WebSphere MQ

Transport for SOAP with Asynchronous Extensions

SC34-6656-00

WebSphere MQ

IBM

Transport for SOAP with Asynchronous Extensions

SC34-6656-00

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix B, "Notices," on page 85.

First edition (May 2005)

This edition of the book applies to the following products:

- IBM WebSphere MQ, Version 6.0

and to any subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

About this book

This book describes the facilities available in WebSphere® MQ to handle SOAP (Simple Object Access Protocol) formatted messages.

It also includes the facilities for asynchronous messaging included in SupportPac™ MA0V.

Who this book is for

This information is for anyone involved in the design or implementation of WebSphere MQ systems that are to process SOAP messages

What you need to know to understand this book

You need:

- Knowledge of the SOAP protocol
- Knowledge of the Apache Axis or Microsoft® .NET host environments.
- Experience of the procedural WebSphere MQ application programming interface as described in the WebSphere MQ Application Programming Guide, or familiarity with the content of the other WebSphere MQ publications

Terms used in this book

The term *WebSphere MQ transport for SOAP* is sometimes abbreviated to *SOAP/WebSphere MQ*.

The term *UNIX systems* denotes the following operating systems:

- AIX
- HP-UX
- Linux
- Solaris

The term *Linux* denotes:

- Linux® (x86 platform)
- Linux (POWER™ platform)
- Linux (zSeries® platform)

The term *Windows* denotes the following Windows operating systems:

- Windows 2000
- Windows XP
- Windows 2003

The term *stem format*, used of a file location, indicates a full path including a filename but with the file extension omitted.

The variable *mqmtop* represents the name of the base directory where WebSphere MQ is installed.

- On AIX®, the actual name of the directory is /usr/mqm

- On other UNIX® systems, the actual name of the directory is /opt/mqm
- On Windows systems, the default directory is C:/Program Files/IBM/WebSphere MQ but you might have chosen to install to a different directory.

The separator character in pathnames is generally shown as "/". If you use Windows 2000, substitute the backslash (\) character.

How to use this book

Part 1 of the book describes SOAP concepts and gives instructions on installing WebSphere MQ transport for SOAP and testing the installation. Part 2 describes how to create and deploy Web services and Web service clients, and Part 3 contains reference information in support of Part 2.

Part 1. Introduction to WebSphere MQ transport for SOAP

Chapter 1. Getting started

This chapter describes WebSphere MQ transport for SOAP and how to use it at a high level.

What is SOAP?

SOAP, the Simple Object Access Protocol, is a protocol for exchange of information in a decentralized, distributed environment. It is an XML (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 information see the W3C website http://www.w3.org/2000/xp/Group.

What is a Web service?

In general there are two parts to a Web service:

- The requester, or client, which issues a request to a server.
- The server. This can be an external web server such as IBM® WebSphere Application Server (WAS), but it need not be. Bespoke service code running on the server does some processing on behalf of the client, and typically sends a reply to the client.

The Web request has two parts

- The Uniform Resource Identifier (URI) of a service.
- A stream of data which the remote server processes and responds to. This is often a SOAP or other web service request and response in XML.

What is WebSphere MQ transport for SOAP?

WebSphere MQ transport for SOAP allows you to send SOAP formatted messages, used in conjunction with Web services, over WebSphere MQ.

It is implemented for either Apache Axis or Microsoft .NET host environments. Axis is available on UNIX or Windows platforms, and .NET on Windows only.

WebSphere MQ transport for SOAP allows interoperation with WebSphere Application Server and CICS®.

Figure 1 on page 4 illustrates how WebSphere MQ transport for SOAP fits into a Web service design. This diagram shows a process where a client application calls a target Web service and obtains a response from the service.

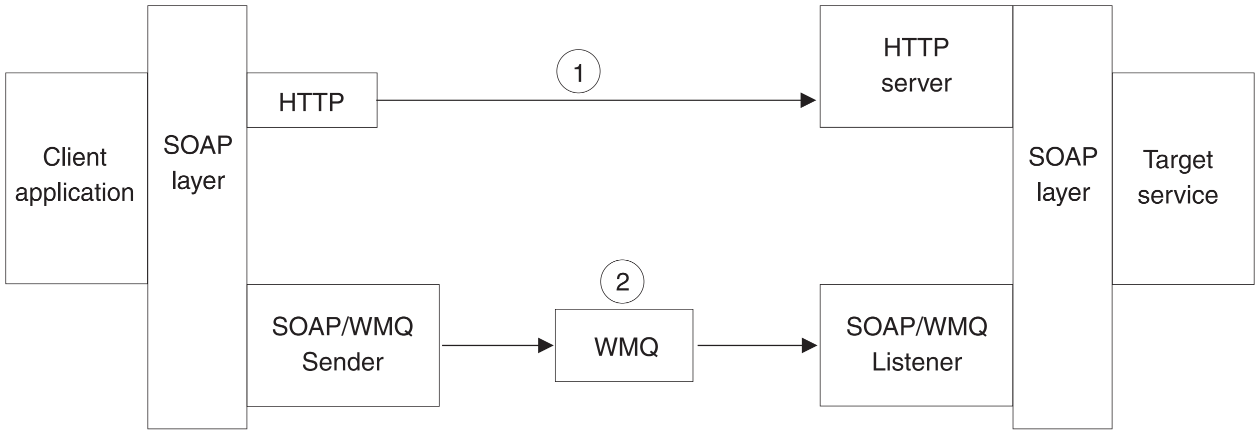

Figure 1. Overview of WebSphere MQ transport for SOAP

First consider the case where HTTP is being used as a transport between the client application and target web service (labelled ″1″ in Figure 1). The client passes the details of the required call to the SOAP layer, which prepares a request for invocation of the service as a SOAP formatted request. This request is dispatched to the server system, where an HTTP server receives the request and passes it through the SOAP layer for decoding and invocation of the service. The response message is processed synchronously by the service, but can be handled either synchronously or asynchronously by the client.

WebSphere MQ transport for SOAP (labelled ″2″ in Figure 1) provides an alternative transport to HTTP. The advantages of using SOAP/WebSphere MQ over SOAP/HTTP include options for:

- Assured delivery.
- Integration with, and reuse of, existing WebSphere MQ infrastructure.
- Use of existing WebSphere MQ security.
- Use of WebSphere MQ clustering, for load balancing and enhanced reliability and availability.

For HTTP the URI is of the format `http://address:[port]/function`, for example `http://localhost:7080/MyRequest`. This would direct the request to the same machine where the application is running (localhost), using port 7080, and the server can select which application is to receive the data based on the MyRequest data.

For SOAP/WebSphere MQ, the URI is of the format `jms:/queue?name=value&name=value...`. This is fully described in Chapter 6, “Specifying the URI,” on page 29.

Apache Axis client applications must be written in Java™. Microsoft .NET client applications must be written in C#, Visual Basic, or other .NET common language runtime (CLR) languages. The target services must be written in Java if using Axis, and in any .NET CLR language if using Microsoft .NET. It is possible to use Axis clients with .NET services, or .NET clients with Axis services, but see “Interoperability” on page 10.

What are senders and listeners in WebSphere MQ transport for SOAP?

Senders
A sender is called by the infrastructure (Axis or .NET) and writes a SOAP request for invocation of a service. In WebSphere MQ transport for SOAP the sender causes the request to be put to a WebSphere MQ request queue, setting up any specific expiry, persistence and priority options. Senders are fully described in "Senders" on page 13.

Listeners
A SOAP/WebSphere MQ listener process waits for incoming messages and then invokes the target Web Service through the Web Services infrastructure and waits for the response. The term *listener* is used here in its standard WebServices sense: these listeners listen on request queues for WebSphere MQ messages and are completely distinct from the standard WebSphere MQ listener invoked by the runmqlsr command. Listeners are fully described in "Listeners" on page 12.

Chapter 2. Overview of WebSphere MQ transport for SOAP

How WebSphere MQ transport for SOAP processes Web services requests

In its synchronous implementation, WebSphere MQ transport for SOAP is based on a traditional request/response model. In the simplest case, using proxy classes, the client program sees this as a remote procedure call. The SOAP/WebSphere MQ client puts a message to a WebSphere MQ queue. This may be the request queue on the local queue manager or may be a remote queue in which case the message is then transported using WebSphere MQ to the appropriate SOAP/WebSphere MQ request queue. A dedicated SOAP/WebSphere MQ listener process monitors the request queue for incoming messages and then routes them through to the target Web Service using the appropriate host infrastructure. Two distinct types of SOAP/WebSphere MQ listener are provided, one for Apache Axis Web Services (SimpleJavaListener) and one for Microsoft .NET web services (amqwSOAPNETListener). Note that these listeners are distinct from the standard WebSphere MQ listener invoked by the **runmqlsr** command.

The sequence of control for this is as follows:

```
Client:    Client Program (provided by user)
              Client Proxy (generated by infrastructure [optional for Axis])
                 WebServices runtime (Axis or .NET infrastructure)
                    SOAP/WebSphere MQ Sender
                       WebSphere MQ
                            |
Server:                 WebSphere MQ
                    SOAP/WebSphere MQ Listener
                 WebServices runtime (Axis or .NET infrastructure)
           Service Program (provided by user)
```

The client proxy is shown as optional for Axis because Axis supports three programming styles, only one of which requires a proxy. .NET always requires a proxy. See “Basic Web service client programming” on page 25 for details of the different programming styles.

Figure 1 on page 4 shows WebSphere MQ positioned between the SOAP/WebSphere MQ sender and listener, providing transport between the client application and the target Web service. Figure 2 on page 8 and Figure 3 on page 9 expand on that figure, showing queues and queue managers within WebSphere MQ, for synchronous messaging. Figure 2 on page 8 illustrates one queue manager (QM1) associated with the sender and one (QM2) with the listener. The sender puts a message on a transmission queue on QM1, from where it is transported to the request queue on QM2. The request queue is monitored by the listener. The listener invokes a Web service and then returns a message via a transmission queue on QM2 to a response queue on QM1. The listener can put messages on the dead-letter queue if a request fails. The sender can be local to QM1 or connected by a WebSphere MQ client connection. The listener can similarly be local to QM2 or connected by a WebSphere MQ client connection.

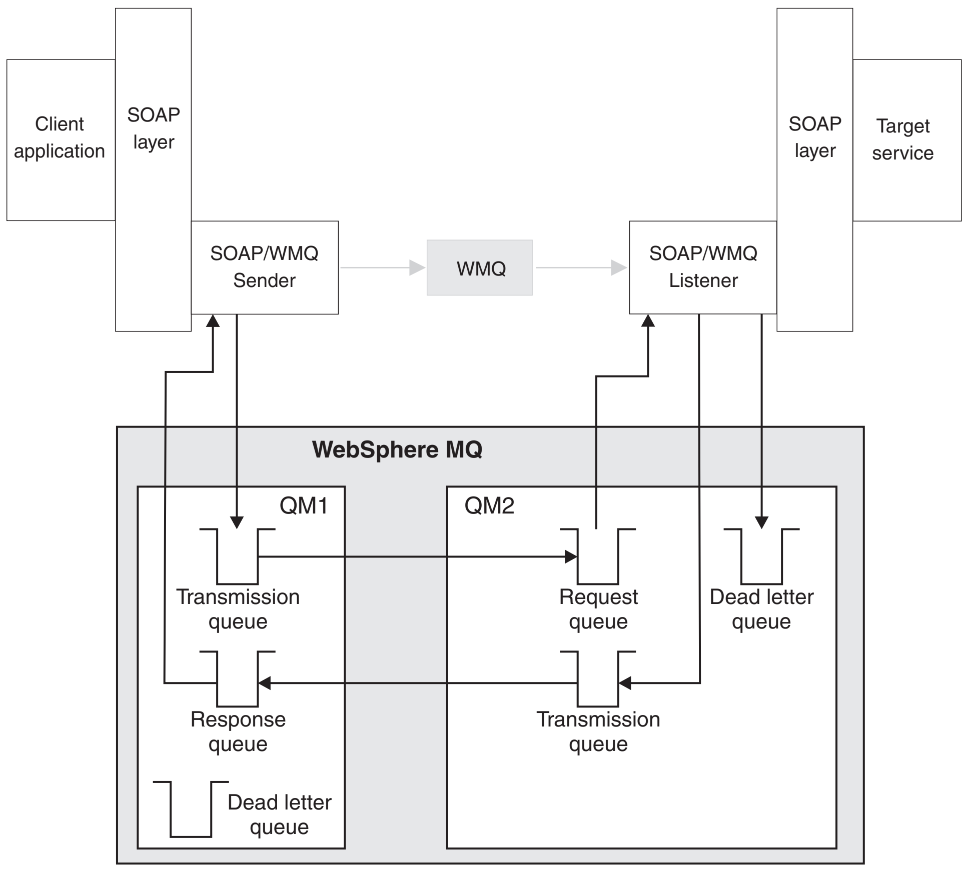

Figure 2. Queues used by SOAP/WebSphere MQ (separate queue managers)

Figure 3 on page 9 illustrates a single queue manager (QM2) servicing both the sender and the listener. The sender places a message on the request queue, which is monitored by the listener. The listener returns a message to a response queue. The listener will put messages onto the dead letter queue if a response message cannot be returned. The sender and listener can be local to QM2 or connected by a WebSphere MQ client connection.

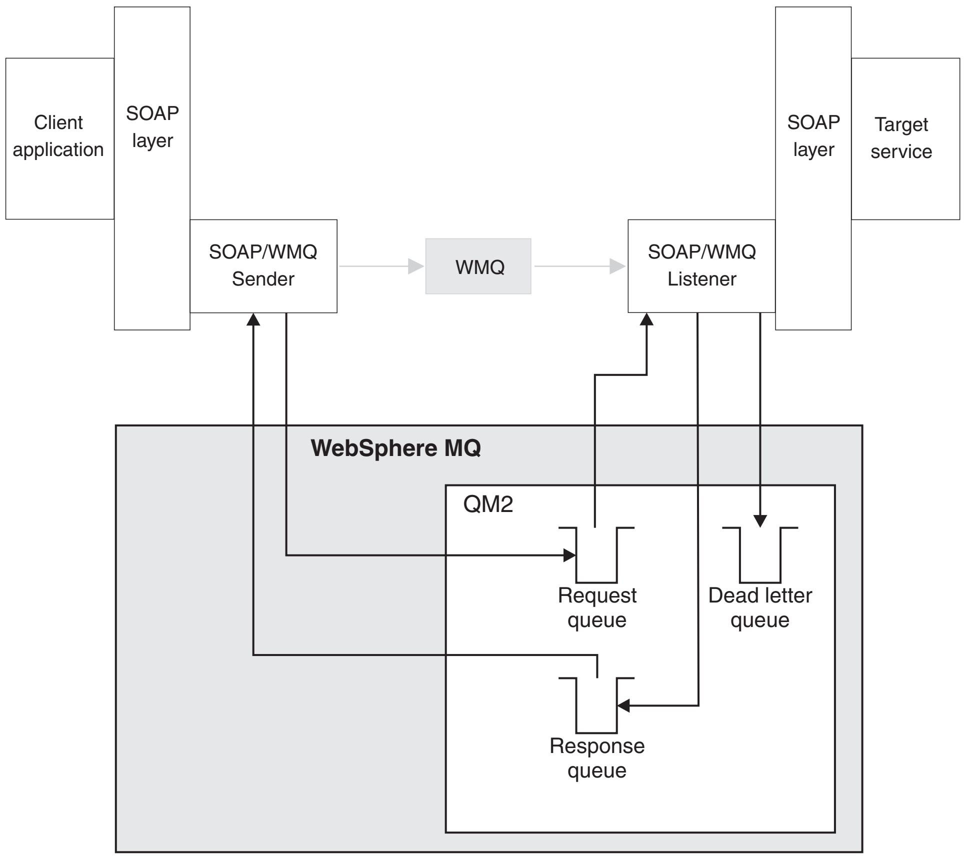

Figure 3. Queues used by SOAP/WebSphere MQ (single queue manager)

In the synchronous implementation, processing is as follows:

1. A client program calls the appropriate WebServices framework in the same way as it would for HTTP transport, except that it must also register the ’jms:’ prefix (see “Basic Web service client programming” on page 25).
2. The Axis or .NET framework marshals the call into a SOAP request message exactly as for SOAP/HTTP.
3. A WebSphere MQ service is identified by a URI prefixed with ’jms:’. When the framework identifies the ’jms:’ URI, it calls the WebSphere MQ transport sender code; com.ib.mq.soap.transport.jms.WMQSender (for Axis) or IBM.WMQSOAP.MQWebRequest (for .NET). If the framework encounters a URI with an ’http:’ prefix, it calls the standard SOAP over http sender.
4. The SOAP message is transported by the WebSphere MQ sender over WebSphere MQ, via the request queue, ready for the SimpleJavaListener (for Java) or amqwSOAPNETListener (for .NET) to receive it.

   The SOAP/WebSphere MQ listeners are standalone processes but are multithreaded with a tailorable number of threads.
5. The SOAP/WebSphere MQ listener reads the incoming SOAP request, and passes it to the appropriate Web service infrastructure.
6. The Web service infrastructure parses the SOAP request message and invokes the service, exactly as it would have done for a message that arrived on an HTTP transport
7. The infrastructure formats the response into a SOAP response message and returns it to the SOAP/WebSphere MQ listener.

8. The listener returns the message via the response queue over WebSphere MQ to the SOAP/WebSphere MQ sender, which passes it to the client Web service infrastructure.
9. The client infrastructure parses the response SOAP message and hands the result back to the client application.

Each application context is served by a separate WebSphere MQ request queue. The application context is controlled in Axis by ensuring that the SOAP/WebSphere MQ listener and service execute in the appropriate directory and with the correct CLASSPATH. Application context is controlled in .NET by the SOAP/WebSphere MQ listener executing the service in a context created by a call to ApplicationHost.CreateApplicationHost. This call specifies the target execution directory. Each service then operates in the directory in which it was deployed. The queues are generated automatically by the SOAP/WebSphere MQ deployment utility, which also generates the infrastructure necessary for handling the queues. See Chapter 7, "Deployment," on page 33 for details of the deployment utility.

Interoperability

WebSphere MQ transport for SOAP does not guarantee interoperability between different host environments such as Apache Axis or .NET. This is because there are many different standards for SOAP and many implementations of SOAP environments, and it is those implementations that determine the specifics of each SOAP message. In addition, there are various different options for formatting the details of a service within a particular implementation (for example, 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.

It is the responsibility of the service provider or developer to identify the SOAP implementations within which the service is to be supported and to ensure that the formatting style is compatible with those implementations. This process is, however, independent from WebSphere MQ; when interoperability is established over HTTP, then it will also be established if WebSphere MQ transport for SOAP is used in place of HTTP.

There are preferred SOAP options for interoperability as defined by the WebServices Interoperability group (WSI, http://www.ws-i.org/). The samples provided with SOAP/WebSphere MQ reflect these options.

WebSphere MQ transport for SOAP uses the same message and URI formats as WebSphere Application Server (WAS) and CICS and so allows interoperability with those products. The message format is described in "Constructing message headers" on page 55 and the URI format in Chapter 6, "Specifying the URI," on page 29. For more information, see "Interoperation with WebSphere Application Server" and "Interoperation with CICS" on page 11.

Interoperation with WebSphere Application Server

To allow SOAP/WebSphere MQ to interoperate with WAS, an APAR has to be applied to WAS. The APAR depends on the version of WAS you have installed:

- For WAS 5.x, APAR PK05013
- For WAS 6.x, APAR PK05012

Deployment of a SOAP/WebSphere MQ Web service using the supplied deployment utility generates appropriate WSDL and a URI for a standard WAS client to use directly: the only special requirement is that ’nojndi.jar’ be in the classpath of the WAS client at runtime.

Normal deployment of a WAS/SOAP/JMS Web service will generate WSDL and a URI that includes JNDI (Java Naming and Directory Interface) references. WebSphere MQ transport for SOAP does not use a JNDI, whereas WAS/SOAP/JMS and other SOAP/JMS implementations generally do. To make the service available to SOAP/WebSphere MQ does not require any change to the runtime, but does require the extension of the WSDL to include a ’pure MQ’ binding with a JNDI-free URI (see the description of the initialContextFactory in Chapter 6, “Specifying the URI,” on page 29). This binding may be a replacement for or addition to the standard JNDI-dependent binding. You must write your own deployment process to generate this new WSDL and URI.

Interoperation with CICS

To allow SOAP/WebSphere MQ to interoperate with a CICS/SOAP client or CICS/SOAP service, APARs have to be applied to both WebSphere MQ and CICS. These APARs are:

- For CICS, APAR PK04615
- For WebSphere MQ, APAR IC46192

If these APARs are not applied, you will experience an MQRFH2 format error.

The Nojndi mechanism

The Nojndi mechanism enables JMS programs (such as WAS SOAP/JMS support) that use JNDI interfaces to use the same URI format as MQ programs that do not use JNDI. Nojndi uses parsing (rather than a repository) to provide the appropriate JMS/WebSphere MQ objects.

The URI contains specific WebSphere MQ queue manager and queue names. These names are parsed and used directly by SOAP/WebSphere MQ support. SOAP/JMS support uses the initialContextFactory specification to decide which JNDI implementation to use. ’initalContextFactory=com.ibm.mq.jms.Nojndi’ will direct it to the Nojndi, which is an implementation of the JNDI interface.

A conventional JNDI implementation looks up input strings in a repository. The repository looks up these inputs based on its configuration, and JNDI returns the result as Java objects. In the case of SOAP/JMS, the input strings are determined by the connectionFactory and destination in the URI, and the JNDI layer then returns the result as appropriate ConnectionFactory and Queue objects. Where the JMS implementation is JMS/WebSphere MQ, these will be objects of the subclasses MQConnectionFactory and MQQueue.

By contrast, the Nojndi implementation operates by parsing the input strings, and does not use a repository. This parsing matches the parsing performed by the SOAP/WebSphere MQ implementation. It is still fed input strings based on the connectionFactory and destination in the URI, and still produces as a result objects of the subclasses MQConnectionFactory and MQQueue. Thus the WAS Web services client will access the same MQ queue managers and queues as the SOAP/WebSphere MQ implementation. No change is needed to the client (other than the presence of ’nojndi.jar’), and it will still be able to use other JNDI based services (that do not use Nojndi) within the same session.

Messages

A SOAP/WebSphere MQ format message is a WebSphere MQ message containing a SOAP message within its body. SOAP/WebSphere MQ provides no constraint on this body; it relies on the host Web services environment to provide SOAP formatting and parsing services.

Listeners

The SimpleJavaListener listener is provided for Axis web services and the amqwSOAPNETlistener listener for Microsoft .NET services. The term *listener* is used here in its standard WebServices sense: these listeners listen on request queues for WebSphere MQ messages and are completely distinct from the standard WebSphere MQ listener invoked by the runmqlsr command.

The SOAP/WebSphere MQ listeners pass SOAP messages as WebSphere MQ messages with a body consisting of a stream of bytes with no assumed structure. If the format is incorrect, the listener generates a report message.

The incoming messages must be encoded in UTF-8. Any response will also be written in UTF-8.

The listener performs a basic integrity check of the incoming request message. The following checks are made:

1. That the basic structure of the MQRFH2 is intact.
2. That all required MQRFH2 fields are present (for example soapAction).
3. That the format of the main body of the incoming message is MQFMT_NONE.

Both the Java and .NET SOAP/WebSphere MQ listeners generate a report message if a request message is badly formed or has an illegal format. This report message is processed according to the report options specified in the sender (see “Report messages” on page 49) and has the feedback code set as described in “Feedback” on page 56.

A SOAP/WebSphere MQ listener passes the endpointURL and soapAction fields in the MQRFH2 component of the message to the SOAP infrastructure to enable that infrastructure to correctly identify and call the target service. The listener does not validate these fields. They are automatically set in the supplied SOAP/WebSphere MQ senders.

The listener invokes the service through the Web Services infrastructure and waits for the response. The listener processes the response message as follows:

1. The SOAP/WebSphere MQ listener sets the correlation ID in the response message according to the report option in the request message. See *WebSphere MQ Application Programming Reference* for details of report options.
2. The listener passes back the same Expiry, Persistence, and Priority settings in the response message as were specified in the request message.
3. The listener also sends report messages back to clients in some circumstances. See “Report messages” on page 13 for more information about report messages.

You can configure the SOAP/WebSphere MQ listeners to be started as WebSphere MQ services in the supplied deployment utility by using the -s option. See “The deployment utility” on page 34 for details.

You can vary the behavior of a SOAP/WebSphere MQ listener by setting various parameters, which are described in "Listeners" on page 43.

Report messages

Report messages can be generated either by WebSphere MQ transport for SOAP or by WebSphere MQ itself in a number of circumstances, depending on the report options specified by the sender when constructing a request message (see "Report messages" on page 49). Note that the report options specified by the SOAP/WebSphere MQ senders (MQRO_EXCEPTION_WITH_FULL_DATA, MQRO_EXPIRATION_WITH_FULL_DATA and MQRO_DISCARD) cannot be tailored.

For more information about report messages and report options in WebSphere MQ transport for SOAP, see Chapter 8, "Senders and listeners," on page 43. For more information about report messages in WebSphere MQ in general, see *WebSphere MQ Application Programming Guide*.

Senders

For Axis web services, a sender is implemented in the final class com.ib.mq.soap.transport.jms.WMQSender which is derived from the org.apache.axis.handlers.BasicHandler class. For Microsoft .NET services, a sender is implemented in the sealed class IBM.WMQSOAP.MQWebRequest. This class is derived from System.Net.WebRequest and System.Net.IwebRequestCreate.

The supplied SOAP/WebSphere MQ sender puts a SOAP request for invocation of a service to a WebSphere MQ request queue. The sender sets fields in the WebSphere MQ MQRFH2 header according to options specified in the URI, or according to defaults. The options that can be specified in the URI are detailed in Chapter 6, "Specifying the URI," on page 29; all the fields in the MQRFH2 are described in "Constructing the MQRFH2 header" on page 59. If you need to change the behavior of a sender beyond what is possible using the URI options, you will have to write your own senders. See "Writing WebSphere MQ transport for SOAP senders" on page 53.

In the synchronous environment, the Java sender blocks after placing the message until it has read a response from the response queue. If no response is received within a given timeout interval the sender throws an exception. If a response is received within the timeout interval the response message is returned to the client via the Axis framework. Your client application must be able to handle these response messages.

The .NET sender creates an MQWebResponse object to read the response message from the response queue and return it to the client.

Diagnostics

Trace facilities in WebSphere MQ transport for SOAP are integrated with the standard WebSphere MQ diagnostic facilities and (for Java) with the WebSphere MQ Java diagnostic classes. See *WebSphere MQ System Administration Guide* for full details of problem determination in WebSphere MQ.

On Windows platforms, when running SOAP/WebSphere MQ listeners as services, the associated process names in the Windows Task Manager are displayed as either java.exe (for Java listeners) or amqwSOAPNETlistener (for .NET listeners). This can make it difficult to identify the specific process for a given service. To associate a process with a service, execute the WebSphere MQ runmqsc utility and issue the command ’display svstatus(*service-name*)’. This displays the PID of the process for the service.

Chapter 3. Installation

SOAP/WebSphere MQ is installable as part of the standard WebSphere MQ install mechanism. It is included as part of the ″Java Messaging and SOAP Transport″ install option in both the server and client installation CDs.

On Windows the binaries, command files, DLLs and executables are installed into the *mqmtop*/bin directory. The SOAP/WebSphere MQ jar files and external jar files used by SOAP/WebSphere MQ are installed into *mqmtop*/java/lib. The samples (including installation verification test (IVT)) are installed to *mqmtop*/tools/soap/samples. Installation on Windows includes registration to the Microsoft .NET Global Assembly Cache of various .DLL and .EXE files.

On Unix systems, the SOAP/WebSphere MQ shell scripts are installed into the *mqmtop*/bin directory. These are symbolically linked to the /usr/bin directory according to the usual WebSphere MQ convention. On these platforms there are no SOAP/WebSphere MQ executables or shared libraries. The jar files are installed into *mqmtop*/java/lib. The samples (including IVT) are installed to *mqmtop*/samp/soap.

What is installed

The following components are provided with WebSphere MQ transport for SOAP:
- WebSphere MQ sender transport code for both Apache Axis and Microsoft .NET environments
- Microsoft .NET and Apache Axis SOAP/WebSphere MQ listeners for polling request queues and invoking target services
- A deployment tool, for defining web services to the host infrastructure.
- Sample source code for a deployment tool.
- Sample Web client and Web service software that can be built, deployed and tested. Sample programs are listed in “Samples” on page 22.
- An Installation Verification Test (IVT) system for running the supplied demonstrations
- Various set up and utility scripts

Prerequisites

The prerequisites for WebSphere MQ transport for SOAP are:
- IBM Java 2 SDK and Runtime environment, V 1.4.2
- One or both of
  - Apache Axis V1.1
  - Microsoft .Net Framework redistributable V1.1 with Service Pack 1
- For .NET only, one of
  - Microsoft .NET Framework SDK V1.1
  - Microsoft Visual Studio .NET 2003
- On Windows® 2000 only, Microsoft Internet Information Services

The IBM Java SDK and Runtime environment are included in the WebSphere MQ installation media. On AIX, Linux and Windows, they are installed automatically but on Solaris and HP-UX they must be selected at installation time.

A version of the Apache Axis runtime is also included in the WebSphere MQ installation media, in the prereqs/axis directory, together with a text file, axis_readme.txt, giving instructions on how to install it. It is not installed as part of the WebSphere MQ installation process. The supplied Axis.jar has been specifically patched (with code published on the Apache Axis Web site). We recommend that you use this version of the Axis runtime with WebSphere MQ transport for SOAP, rather than any other version that you might already have installed. Note that IBM does not provide technical support for Apache Axis. If you have technical problems with Axis, or any queries about it, you should contact the Apache Software Foundation. Contact details are given in Appendix A, “Apache software license,” on page 83.

On Windows 2000 only, you must have installed Microsoft Internet Information Services (IIS) to be able to deploy and run Microsoft .NET services. It is not necessary for it to remain installed if the services you are invoking will be using only WebSphere MQ SOAP transport but it is necessary that IIS has at least been installed at some stage previously before making a service deployment. If the Microsoft .NET framework has been installed before IIS, you must use the aspnet_regiis utility to register IIS to the framework. The location of the aspnet_regiis.exe utility might vary with different versions of the Microsoft .NET framework, but it is typically located in: %SystemRoot%/Microsoft.NET/Framework/*version number*/aspnet_regiis. If multiple versions are installed, use only the executable for the version of .NET you are using.

On Windows 2003 only, though you do not need to install IIS, you must run aspnet_regiis to register IIS to the framework.

The installation process does not check for the presence and availability of the prerequisite software items. You must verify that they are installed first.

For .NET only, you must register the WebSphere MQ transport for SOAP files with the Global Assembly Cache. If .NET is already installed when you install WebSphere MQ, registration is performed automatically at installation. If you install .NET after WebSphere MQ, the registration is performed automatically when the IVT is first run (see “Testing your SOAP/WebSphere MQ installation”), or you can run amqwregisterDotNet.cmd to perform registration. You can also run amqwregisterDotNet.cmd to force reregistration at any stage. Once made, this registration survives system restarts and subsequent reregistration is not normally necessary.

Testing your SOAP/WebSphere MQ installation

An installation verification test suite (IVT) is provided with WebSphere MQ transport for SOAP. This runs a number of demonstration applications and ensures the environment is correctly set up after installation.

Before running the IVT, set the environment variable WMQSOAP_HOME to specify the WebSphere MQ installation directory. If you are using the IVT to test asynchronous functionality, build the asynchronous samples using regenDemoasync.cmd (on Windows) or regenDemoasync.sh (on UNIX) and the transactional samples using regenTranDemoasync.cmd or regenTranDemoasync.sh

Change to a directory under which you want the IVT to deploy. Run the IVT by executing the runivt.cmd script on Windows or the runivt.sh script on Unix systems. The script is located in the samples directory. The full set of tests can be executed by entering the command ″runivt″ with no arguments. The IVT starts the listeners it requires during the tests and by default closes them down before exiting. If you want to leave the listeners running after the IVT has completed, specify the ″hold″ option as the last argument on the runivt command line. The listeners will be started in separate command windows; for this reason it is necessary to be using an X-Windows session when using the IVT on Unix systems. On Windows platforms, the IVT utility by default uses a configuration file called ivttests.txt that details the various tests to be performed. On Unix systems the file is called ivttests_unix.txt. To use a different configuration file, for example if you want to run your own tests, specify the ″-c filename″ option.

The configuration file is a plain text file that describes the tests that can be executed. Each test is defined over 5 lines. For example, for the IVT test labeled ″Dotnet″ the entries in the configuration file are:

```
Dotnet
WMQ transport test: C# to .NET (Asmx)
SQCS2DotNet
DOC reply is: 77.77
dotnet
```

The first line (″Dotnet″) is the name of the test. This can be specified as an argument to the runivt script to specify which test(s) should be run. The second line (″WMQ transport test: C# to .NET (Asmx)″) is a description of the test. The third line (″SQCS2DotNet″) is the command that the IVT will execute to start the client application. The fourth line (″DOC reply is: 77.77″) is the expected reply string from the client. This must be the last actual line output by the client for the test to have been deemed to pass. The fifth line (″dotnet″) is the name of the SOAP/WebSphere MQ listener that the IVT will start in order for the service request to be processed. Valid listener names are ″dotnet″ for the SOAP/WebSphere MQ amqwSOAPNETlistener and ″JMSax″ for the SOAP/WebSphere MQ SimpleJavaListener.

If you want to use the IVT to run only a single test, the name of the test should be the first argument supplied to the utility.

To run two or more tests, supply the names of the tests to be run as arguments to the utility. For example, to run the ″dotnet″ and ″AxisProxy″ tests, invoke the IVT as follows:

```
runivt dotnet axisproxy
```

To leave the listeners running add the ″hold″ parameter to the end of the command.

All arguments to the IVT are case insensitive. The IVT configuration file can contain comment lines (indicated by a ′#′ character in the first character of a line) and blank lines.

For example, to run just the ″dotnet″ test and leave the listeners running, invoke the IVT as follows:

```
runivt dotnet hold
```

This produces output similar to the following:

```
define qlocal(SYSTEM.SOAP.RESPONSE.QUEUE) BOTHRESH(3) completed OK.
define qmodel(SYSTEM.SOAP.MODEL.RESPONSE.QUEUE) BOTHRESH(3) DEFTYPE(PERMDYN) DEFSOPT(SHARED) SHARE
completed OK.
define qlocal(SYSTEM.SOAP.SIDE.QUEUE) completed OK.
define channel(TESTCHANNEL) CHLTYPE(SVRCONN) TRPTYPE(TCP) REPLACE completed OK.
define channel(TESTSSLCHANNEL) CHLTYPE(SVRCONN) TRPTYPE(TCP) SSLCIPH(DES_SHA_EXPORT) REPLACE completed OK.

----- [Dotnet] --------------------------------
WMQ transport test: C# to .NET (Asmx)
--- client: SQCS2DotNet  jms:/queue?desination=SOAPN.demos@WMQSOAP.DEMO.QM&connectionFactory=connectQueueM
anager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE&targetService=StockQuoteDotNet.asmx&in
initialContextFactory=com.ibm.mq.jms.Nojndi

RPC reply is: 88.88
OK.
----------------------------------
C:/temp/demos>rem - generated by DeployWMQService.java at 08-Mar-2005 11:05:04
C:/temp/demos>call amqwsetcp.cmd
===========================================
1 tests run, of which 0 failed.
```

The list of tests which can be run using the IVT is as follows. These tests are described in “Samples” on page 22.

- SQAxis2Axis
- SQAxis2DotNet (Windows only)
- SQAxis2AxisAsyncMany
- SQAxis2AxisAsyncRequestResponse
- SQCS2Axis
- SQCS2DotNet (Windows only)
- SQCS2DotNetAsyncMany (Windows only)
- SQCS2DotNetAsyncRequestResponse (Windows only)

Part 2. Using WebSphere MQ transport for SOAP

Chapter 4. Creating and deploying a Web service using WebSphere MQ transport for SOAP

This chapter explains, at a high level, the steps you must carry out to develop and implement a Web service and a client to invoke that service.

Target Web services need to be processed through a series of deployment steps. These steps define the target service to the Axis or .NET host infrastructure, generate proxy methods to simplify the process of invoking the service from the client, prepare a script file to start one of the service listeners and perform some queue and process configuration within WebSphere MQ.

Preparing service code

Java

A Java service that has been compiled into classes can be used without modification, provided that the type of any arguments to the methods in the web service are supported by the Axis engine. Refer to Axis documentation for further details. There is a restriction on the use of complex objects as arguments to the service. This is detailed in "Basic service programming" on page 26.

.NET

A .NET service that has already been prepared as an HTTP Web service does not need to be further modified for use as a WebSphere MQ Web service but does need to be redeployed through a SOAP/WebSphere MQ deployment process. See "Basic service programming" on page 26 for more details about how to prepare service code for use as a SOAP/WebSphere MQ service, especially when the service uses external classes, and for an example of a .NET class modified to run as a Web service.

Deploying a service

Deployment is the process of configuring the host web services infrastructure (Axis or Microsoft .NET) to recognize the prepared web service.WebSphere MQ transport for SOAP supplies a sample deployment utility and specimen command files. For more details, see Chapter 7, "Deployment," on page 33.

Preparing client code

You code client applications almost identically to the equivalent applications for SOAP/HTTP over the same host infrastructure (Axis or .NET) and you can continue to use your standard development environment. However, you must add a registration call to the client to register the WebSphere MQ SOAP transport sender, so that the Web services framework at the client environment is willing to accept a URI prefixed with 'jms:'. For more details, see"Basic Web service client programming" on page 25.

Linking a client

Compile and link Java clients using your standard Java compiler. Make all the required classes available by setting up the CLASSPATH using the amqwsetcp.sh script.

A sample build script, msdemobuild.cmd is included in the samples directory. This builds the .NET samples and you can use it as a basis of scripts to build your own client programs.

Run amqwclientconfig to create client-config.wsdd in your deployment directory. This is unnecessary if you are recompiling a client and the client-config.wsdd is already prepared. amqwclientconfig is run automatically when preparing the sample programs using regenDemo, regenDemoAsync or regenTranDemoAsync.

Executing a client

A client application is executed in the same way as any other application on its host infrastructure. You must, however, set the CLASSPATH and WMQSOAP_HOME environment variables for Java clients or the PATH variable for .NET clients.

If you experience security problems using a SOAP/WebSphere MQ client installed on a network drive, you can use the .NET Framework configuration tool mscorcfg.msc to customize your .NET security settings. Refer to Microsoft .NET documentation for details of this utility.

Starting listeners

The deployment process (see “The deployment utility” on page 34) automatically creates wrapper scripts in the generated server directory that set up and invoke the listener. Two scripts are generated to start the listener; one starts it as a WebSphere MQ service and the other starts it directly (the latter script is used when a listener is started by triggering). You can also start a listener manually. See “Listeners” on page 43 for more details.

Samples

Sample services and client applications are supplied for both Java and .NET and for both synchronous and asynchronous use. The synchronous samples are built using regenDemo.cmd (on Windows) or regenDemo.sh (on UNIX). The asynchronous samples must be built using regenDemoasync.cmd (on Windows) or regenDemoasync.sh (on UNIX) and the transactional samples using regenTranDemoasync.cmd or regenTranDemoasync.sh. Run these commands after running regenDemo to build the synchronous samples. The samples are based on a Stock Quote service that takes a request for a stock quote and provides the stock quote. On Windows, samples are installed to *mqmtop*/tools/soap/samples. On UNIX systems, the samples are installed to *mqmtop*/samp/soap.

The samples generate logs by redirecting output from WebSphere MQ utilities such as runmqsc to a log file. If you are running the samples under Windows, Notepad and other editors can cause certain characters to be displayed wrongly in some European languages. If you view the log by issuing the **type** command, Windows displays the characters correctly.

Samples for Java

The following sample services and client applications are supplied for the Java environment. Most of these can be run using the supplied IVT. Those that cannot are indicated.

StockQuoteAxis.java
Defines the stock quote service. This file is used to generate the proxies required by the client code.

SQAxis2Axis.java
This sample provides an example of a request to an Axis service providing stock quotes.

SQAxis2DotNet.java
This sample provides an example of a request to a .NET service providing stock quotes.

SQAxis2AxisAsyncMany.java
This sample provides an example of making several asynchronous requests to an Axis service and then waiting for responses and processing the responses when they are received.

SQAxis2AxisAsyncRequestResponse.java
This sample provides an example of making an asynchronous request to an Axis service and then waiting for the response and processing the response when it is received, either in the same or a different process.

SQAxis2AxisAsyncReqRespNtfy.java
This sample demonstrates the use of the onInitilialize() and onTerminate() methods when implementing the IResponseListenerNotification interface and specifies the use of this to the response listener. The methods are called when the response listener starts and stops, and also if the response listener terminates because of an exception.

This sample is not included in the IVT, and is left for you to run manually if required. Instructions for running this sample are supplied in “Sample programs” on page 79.

SQAxis2AxisAsyncReqRespTran.java
This sample provides an example of the use of a transactional client request to an Axis service. It is not included in the IVT, and is left for you to run manually if required. Transactional processing is described in Chapter 11, “Transactional processing,” on page 65; instructions for running this sample are supplied in “Sample programs” on page 66.

Samples for .NET

The following sample programs are supplied for the .NET environment. Most of these can be run using the supplied IVT; those that cannot are indicated.:

StockQuoteDotNet.asmx
Defines the stock quote service. This file is used to generate the proxies required by the client code.

SQCS2Axis.cs, SQVB2Axis.vb
These samples provide an example of a request to an Axis service providing stock quotes. The samples provide examples coded in C# and Visual Basic respectively.

SQCS2DotNet.cs, SQCS2DotNet.vb
These samples provide an example of a request to a .NET service providing stock quotes. The samples provide examples coded in C# and Visual Basic respectively.

SQDNNoninline.asmx, SQDNNoninline.asmx.cs, SQCS2DNNoninline.cs
These 3 files provide an example of the use of non inline code using the codebehind technique in an 'asmx' file. These are not part of the IVT. To run this sample, do the following:

1. Compile the SQDNNoninline.asmx.cs and build a .dll from the resulting object file.
2. Put the .dll in the 'bin' subdirectory of the directory from where the service is run.
3. The file SQCS2DNNoninline.cs is a sample of the client code that invokes the service. The client invokes the service with a 'targetService' of SQDNNoninline.asmx

SQCS2DotNetAsyncMany.cs
This sample provides an example of making several asynchronous requests to a .NET service and then waiting for responses and processing the responses when they are received.

SQCS2DotNetAsyncRequestResponse.cs
This sample provides an example of making an asynchronous request to a .NET service and then waiting for the response and processing the response when it is received, in either the same or a different process.

SQCS2DotNetAsyncReqRespNtfy.cs
This sample demonstrates the use of the onInitilialize() and onTerminate() methods when implementing the IResponseListenerNotification interface and specifies the use of this to the response listener. The methods are called when the response listener starts and stops, and also if the response listener terminates because of an exception.

This sample is not included in the IVT, and is left for you to run manually if required. Instructions for running this sample are supplied in “Sample programs” on page 79.

SQCS2DotNetAsyncReqRespTran.cs, StockQuoteDotNetTran.asmx.cs,StockQuoteDotNetTran.asmx
These samples provide an example of the use of a transactional client request to a .NET service. These samples are not included in the IVT, and are left for you to run manually if required. Transactional processing is described in Chapter 11, “Transactional processing,” on page 65; instructions for running this sample are supplied in “Sample programs” on page 66.

Chapter 5. Programming for WebSphere MQ transport for SOAP

This chapter discusses issues to do with the writing of client applications and Web services. After a note on the languages available, it considers writing client applications, firstly in Java, with examples of two styles, and then in .NET, with an example. It then considers writing Web services, firstly in Java and then in .NET, with an example of a .NET class prepared as a Web service.

Languages supported

Apache Axis client and service applications must be written in Java.

Microsoft .NET client and service applications must be written in C#, Visual Basic, or other .NET CLR languages.

Basic Web service client programming

You code client applications almost identically to the equivalent applications for SOAP/HTTP and you can continue to use your standard development environment. However, you must add a registration call to the client to register the WebSphere MQ transport for SOAP transport sender, so that the WebServices framework at the client environment is willing to accept a URI prefixed with 'jms:'. This call depends on the programming language, as follows:

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

C# IBM.WMQSOAP.Register.Extension();

Visual Basic
IBM.WMQSOAP.Register.Extension()

Examples of these calls are given in the following sections.

Java

For Java, WebSphere MQ provides access to web services using the Apache Axis Web Services infrastructure. Refer to the Axis documentation for full information on how to use the infrastructure.

Axis supports three programming styles: SOAP style, WSDL style and PROXY style. The key features of these three styles can be summarized as follows:

SOAP style
Assumes that the client knows about the location and signature of the service and does not use a WSDL definition of the service.

WSDL style
Uses WSDL to locate the service, but still relies on the client to know the signature and prepare the parameters accordingly.

Proxy style
Assumes that a proxy to the service has been pregenerated from WSDL. The client calls the service via an instantiation of the proxy object and the service signature is checked at compile-time. This is likely to be the simplest and easiest option.

All three of these styles are supported in the SOAP/WebSphere MQ Java client environment. However, SOAP style offers limited flexibility and ease of use and samples are provided only for WSDL and Proxy styles.

WSDL style

Sample *mqmtop*/tools/soap/samples/java/clients/soap.clients.WsdlClients.java shows an example of a simple Java client WebSphere MQ transport test. This calls an Axis service from an Axis client environment using WSDL Axis calls. The programmer is responsible for referencing the correct WSDL (which can be held locally or accessed over HTTP), and using appropriate ports and bindings.

Proxy style

Sample *mqmtop*/tools/soap/samples/java/clients/soap.clients.SqAxis2Axis.java is an example of a simple Java WebSphere MQ transport test. This calls an Axis service from an Axis client environment using automatically generated proxy classes. The programmer must reference the correct proxies, and the proxies will have been generated to get the remaining information correct.

.NET

For .NET, WebSphere MQ provides access to web services using the Microsoft .NET SDK. Refer to the .NET documentation for information about using the .NET infrastructure.

SOAP/WebSphere MQ only supports the proxy programming style for .NET clients. There is no equivalent to the Axis SOAP or WSDL programming styles in the .NET environment.

Proxy style

mqmtop/tools/soap/samples/dotnet/clients/SQCS2DotNet.cs is an example of a simple C# WebSphere MQ transport test. This calls a .NET service from a .NET client environment using automatically generated proxy classes. The programmer must reference the correct proxies, and the proxies will have been generated to get the remaining information correct:

Basic service programming

Java

A Web service that has been compiled into classes can be used without modification, provided that the types of any arguments to the methods in the web service are supported by the Axis engine. Refer to Axis documentation for further details.

If the service uses a complex object as an argument, or returns one, that object must comply to the Java Bean specification.

The supplied deployment utility does not support the case where a service returns an object in a different package to the service itself. If you wish to do this, you can write your own deployment utility based on the supplied sample, or capture the commands produced by the supplied utility, using the **-v** option, and amend them to produce a tailored script.

If the service uses classes that are external to the Axis infrastructure and the SOAP/WebSphere MQ run time environment, you must amend the generated script that starts or defines the listeners, changing the CLASSPATH to include the services required. You can do this in any of the following ways:

- Amend the CLASSPATH directly in the script after the call to amqwsetcp.
- Create a service-specific script to customize the CLASSPATH and invoke this script in the generated script after the call to amqwsetcp.
- Create a customized deployment process to customize the CLASSPATH in the generated script automatically.

.NET

A service that has already been prepared as an HTTP Web service does not need to be further modified for use as a WebSphere MQ Web service but it does need to be redeployed through a SOAP/WebSphere MQ deployment process. If the service code has not previously been prepared an HTTP Web service you must modify it to declare it as a web service and to identify how each method's parameters should be formatted. You must also check that any arguments to the methods of the service are compatible with the environment. Figure 4 shows a .NET class that has been prepared as a web service. The additions made are shown in bold type.

```
<%@ WebService Language="C#" Class="StockQuoteDotNet" %>

using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Services.Description;
using System.Threading;

[WebService (Namespace="http://dotnet.server")]
public class StockQuoteDotNet {

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

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

Figure 4. Example of .NET service programming

If the web service uses classes that are external to the .NET infrastructure and the SOAP/WebSphere MQ run time environment, you must write and build the service source code as non-inline. This means the source for the service is separated from the asmx file. The asmx file must declare the name of the associated source file with the "codebehind" keyword and the source must be compiled prior to deployment.

A sample non-inline service program is supplied with WebSphere MQ transport for SOAP. This is the SQCS2DNNonInline sample described in "Samples for .NET" on page 23.

Chapter 6. Specifying the URI

A web service is specified using a Universal Resource Identifier (URI). This section specifies the URI format that is supported in WebSphere MQ transport for SOAP. This URI format permits a comprehensive degree of control over SOAP/WebSphere MQ specific parameters and options when accessing target services. This format is compatible with WebSphere Application Server (WAS) and with CICS facilitating the integration of WebSphere MQ with both those products provided that the prerequisite APARs have been applied. These APARs are listed in "Interoperability" on page 10.

The URI syntax is as follows:

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

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

Parameter names and values are listed below. Parameter names are case sensitive, as are 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 unrecognised parameters are included, they are ignored.

If you store a URI in an XML string, you must represent the ampersand character as ″&amp;″. Similarly, if a URI is coded in a script, take care to escape characters such as **&** which would otherwise be interpreted by the shell.

Examples of URIs are given in "Sample URIs" on page 31.

Parameter names and values

destination
: This parameter is required and should be the first parameter in the URI after the initial ′jms:/queue′ string.

  The name of the request queue: either a WebSphere MQ queue name, or a queue name and queue manager name connected by an @ symbol, for example `SOAPN.trandemos@WMQSOAP.DEMO.QM`. Note that WebSphere MQ Publish/Subscribe is not supported.

connectionFactory
: This parameter is required. For the syntax of this parameter, see "The connectionFactory parameter" on page 30.

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 (see "Interoperation with WebSphere Application Server" on page 10).

timeout
: The time, in milliseconds, that the client will wait for a response message.

Overrides any values set by the infrastructure or client application. If not specified, the application value (if specified) or infrastructure default is used.

targetService

This option is mandatory for accessing .NET services. In the .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 as the Axis infrastructure permits SOAP/WebSphere MQ listeners to access multiple services. If it is specified in the Axis environment it will override the default Axis mechanism.

The *value* for this parameter is a service name. For a .NET service the service name should be specified with no directory qualification as .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`.

timeToLive

Specifies the expiry time of the message in milliseconds. The default is zero, which indicates an unlimited lifetime.

Note: No relationship is enforced between timeout and expiry.

persistence

Specifies the message persistence. Following standard JMS conventions, this is specified as a number, with the following meanings:

0	No persistence specified; WebSphere MQ treats this as PersistenceAsQDef. This is the default.
1	The message is non persistent.
2	The message is persistent

priority

Specifies the message priority. Valid values are in the range 0 (low) to 9 (high). The default is environment specific, for WebSphere MQ the default is 0.

replyDestination

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

The connectionFactory parameter

The connection factory parameter's syntax is as follows:

```
connectionFactory=name(value)name(value)...
```

where **name** is a sub-parameter name and *(value)* is an appropriate value, and the **name***(value)* element is repeated as necessary. There is no separator between occurrences of **name***(value)*.

Sub-parameter names and values are as follows. If you are using SSL, you must add further SSL-specific sub-parameters, as detailed in "SSL-related options in the URI" on page 61. All the sub-parameters are optional; if none are to be set, you must code the connectionFactory parameter as **connectionFactory=()**.

connectQueueManager
Specifies the queue manager to which the client will connect. The default is blank.

binding
Which type of binding should be used on the queue manager connection. If the binding option is not specified but options appropriate to a client binding are specified (such as **clientConnection**), the sender code assumes a client type binding. If no client type attributes are specified and no binding type is specified, the default is ″auto″ which means that the client will attempt a server connection first. If this is not successful a client connection will then be attempted. If ″server″ is specified as the binding type, then the client will not attempt a client bindings connection if the server connection fails. Other options are ″client″, where it is known a server bindings connection would not be appropriate, or ″xaclient″ (xaclient applies to .NET only). The SOAP/WebSphere MQ sender code checks the URI for any inconsistencies in the specified options. For example if the URI specifies ″binding=server″ but also had client type parameters specified such as ″clientConnection=″ or SSL parameters, an error message is displayed by the SOAP/WebSphere MQ sender and the request fails.

clientChannel
Specifies the channel to be used when a 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
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 hostname (for example `MACH1.ABC.COM`) or 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 can include the port number, for example `MACH1.ABC.COM(123)`.

Sample URIs

This is an example of a simple URI for an Axis service:

```
jms:/queue?destination=myQ&connectionFactory=()
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

This is an example of a simple URI for a .NET service:

```
jms:/queue?destination=myQ&connectionFactory=()&targetService=MyService.asmx
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

Only the required parameters are supplied (**targetService** is required for .NET services only), and **connectionFactory** is given no options.

In this Axis example, **connectionFactory** contains a number of options:

```
jms:/queue?destination=myQ@myRQM&connectionFactory=connectQueueManager(myconnQM)
binding(client)clientChannel(myChannel)clientConnection(myConnection)
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

In this Axis example, the **sslPeerName** option of **connectionFactory** has also been specified. The value of sslPeerName itself contains name value pairs and significant embedded blanks:

```
jms:/queue?destination=myQ@myRQM&connectionFactory=connectQueueManager(myconnQM)
binding(client)clientChannel(myChannel)clientConnection(myConnection)
sslPeerName(CN=MQ Test 1,O=IBM,S=Hampshire,C=GB)
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

Request queues

If you do not specify the name of the request queue in a URI, it is determined by the deployment process and depends on how the service was named to the deployment process. The service name might either be a simple filename or might include a relative path name. The queue name is generated by removing the extension from the filename and replacing any file separator characters with periods. For example a service name of dotnetDemos/server/StockQuoteAxis.cs gives a queuename of dotnetDemos.server.StockQuoteAxis.

As for any WebSphere MQ queue, the queue name must be no longer than 48 characters. You can override the default queue name on deployment to specify a shorter name. You might also need to override the queue name to ensure unique names. If the deployment utility generates a queue name longer than 48 characters it truncates it but this could result in duplicate queue names. See Chapter 7, "Deployment," on page 33.

Response queues

The default name of the response queue is SYSTEM.SOAP.RESPONSE.QUEUE. You can override this by specifying the replyToQueue parameter in the target URI. See Chapter 6, "Specifying the URI," on page 29 for further details. The demonstration programs create this default queue automatically in the demonstration queue manager. A script is provided (setupSOAPWMQ.cmd in Windows or setupSOAPWMQ.sh in UNIX) that configures MQ with default SOAP/WebSphere MQ queue configurations for a specified queue manager. If you nominate a non-default response queue name, it is your responsibility to ensure the queue is properly defined.

Dynamic response queues

Permanent and temporary dynamic response queues are supported. To use a dynamic response queue, specify a model queue name in the replyToQueue field of the SOAP/WebSphere MQ URI. A model queue called SYSTEM.SOAP.MODEL.RESPONSE.QUEUE is defined for a specified queue manager by the script setupWMQSOAP.cmd (for Windows) or setupWMQSOAP.sh (for UNIX). The model queue is set up as a permanent dynamic queue but you can change it according to your requirements. Alternatively, you can create your own model queue.

Long term asynchronous service calls use permanent dynamic queues. For more information, see Chapter 12, "Asynchronous messaging," on page 71.

Chapter 7. Deployment

Deployment is the process of configuring the host web services infrastructure (Axis or Microsoft .NET) to recognize the prepared web service.

A deployment utility is provided as part of WebSphere MQ transport for SOAP. This consists of a Java program, com.ibm.mq.soap.util.amqwdeployWMQService, and command files amqwdeployWMQService.sh and amqwdeployWMQService.cmd, which invoke it. Source code functionally equivalent to amqwdeployWMQService is also supplied so you can use it as the basis of your own deployment utility.

The supplied deployment utility undertakes the following activities:

1. Validate an optional supplied URI to be used in the deployment process. (Though optional, a URI is normally supplied to avoid the need to specify a URI at run time.)
2. Prepare the WSDL from the service code.
3. (Axis only) Prepare deployment descriptor files and create or update server-config.wsdd, which defines services to Axis.
4. Generate client proxies from the WSDL.
5. Prepare scripts for invoking and stopping a SOAP/WebSphere MQ listener on the Web Service platform
6. Configure WebSphere MQ with the required queues and processes necessary to implement the service.

You cannot deploy from existing WSDL using the provided deployment utility: you can only use it by nominating the source file of a service. If you want to deploy from WSDL you will have to write your own deployment procedure.

After deployment using the provided utility it is possible for clients to invoke services through a special proxy class generated by the deployment procedure. In the Axis environment, use of a proxy is generally be simpler than the alternatives of using low level calls or by the use of a WSDL configuration file.

You might need to do further configuration, for example:

- If the client is operating with server bindings to a service on a different machine, create channel definitions and enable communication between queue managers located on the two machines.
- If the client application is to invoke the service with WebSphere MQ client bindings and there is no local queue manager at the client, create and configure a server connection channel to enable the client and server to communicate.
- Configure SSL communications if required on client bindings. It is necessary to set up the WebSphere MQ channel definitions appropriately if you want to use SSL and also to prepare a key repository file and import keys and certificates into it. See Chapter 10, "Using SSL with WebSphere MQ transport for SOAP," on page 61 for details.
- If the client and server are on different machines, either deploy on the server machine and copy the proxies to the client machine, or deploy on both machines and remove the redundant elements from each platform.

The deployment utility

The deployment utility prepares a service class for use as a Web service using WebSphere MQ as the transport.

The deployment utility amqwdeployWMQService.java is most easily called from the supplied command files amqdeployWMQService.cmd and amqdeployWMQService.sh. Change to the root directory of the directory in which the source exists before calling amqdeployWMQService.

A source code example of a deployment utility program is supplied, so you can develop your own customized deployment procedures for your specific environment. See "Customizing the deployment process" on page 53 for details.

Syntax of amqwdeployWMQService

The calling syntax for the UNIX shell is:

```
./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]
```

and for the Windows command file:

```
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]
```

Where:

-f className
The name of the class to be deployed. For Java, this must be fully qualified by the package. It can be specified as a path name with directory separators or as a class name with period separators. For a .NET service, although the directory can be specified, Java proxies are always located in the package dotNetService.

If you specify a URI with the -u option and within the URI specify the targetService, the deployment utility checks that the className you have specified matches that service. If the class and service specified do not match, it displays an error message and exits.

-a integrityOption
Allows the default behavior of SOAP/WebSphere MQ listeners to be changed when it is not possible to put a failed request message on the dead-letter queue. **integrityOption** can take one of the following values:

DefaultMsgIntegrity
For non-persistent messages, the listener displays a warning message and continues to execute with the original message being discarded. For persistent messages, it displays an error message, backs out the request message so it remains on the request queue and exits. This default mode applies if the -a flag is omitted, or if it is specified with no option.

LowMsgIntegrity
For both persistent and non-persistent messages, the listener displays a warning and continues to execute, discarding the message.

HighMsgIntegrity
: For both persistent and non-persistent messages, the listener displays an error message, backs out the request message so it remains on the request queue and exits.

The deployment utility checks for the compatibility of the -x and -a flags. If ″-x none″ is specified, then ″-a LowMsgIntegrity″ must be specified. If the flags are incompatible it exits with an error message and with no deploy steps having been undertaken.

-b bothresh
A numeric value specifying the backout threshold setting that is to be set on the request queue. The default is 3.

-c operation
Specifies which part of the deployment process to be executed. **operation** is one of the following options:

allAxis
Perform all compile and setup steps for an Axis/Java service.

compileJava
Compile the Java service (.java to .class).

genAxisWsdl
Generate WSDL (.class to .wsdl).

axisDeploy
Deploy the class file (.wsdl to .wsdd, apply .wsdd).

genProxiestoAxis
Generate proxies (.wsdl to .java and .class).

genAxisWMQBits
Set up WebSphere MQ queues, SOAP/WebSphere MQ listeners and triggers for an Axis service.

allAsmx
Perform all setup steps for a .NET service

genAsmxWsdl
Generate WSDL (.asmx to .wsdl).

genProxiesToDotNet
Generate proxies (.wsdl to .java, .class, .cs and .vb)

genAsmxWMQBits
Set up WebSphere MQ queues, SOAP/WebSphere MQ listeners and triggers

startWMQMonitor
Start the trigger monitor for SOAP/WebSphere MQ services. See “Starting listeners by triggering” on page 49.

If this option is omitted, the default is allAxis if the className given with the -f parameter has a .java extension, and allAsmx if it has an asmx extension.

-i passContext
Specifies whether the listeners should pass identity context. This parameter can take the values **passContext** or **ownContext**. If the parameter is omitted, the default is to pass context. (See“Context” on page 50).

-n num
Number of threads to be specified in the startup scripts for the

SOAP/WebSphere MQ listener. The default is 10. Consider increasing this number if you have high message throughput or when services are being called asynchronously.

-r Specifies that any existing request queue or trigger monitor queue (if -tmq was specified) will be explicitly replaced. By specifying -r, you ensure the queues will be recreated with standard default attributes and with no messages. If the -r option is not used then any existing queue definitions will not be altered nor message entries deleted. By not specifying -r, you ensure that any customized queue attributes are preserved.

-s Configure the listener to be executed as a WebSphere MQ service. This option is mutually exclusive to the -tmq option.

If -s and -tmq are both specified, the deployment utility displays an error message and exits.

-tmp programName
Specifies a trigger monitor program. This option might be used in a UNIX environment to circumvent limitations of the setuid trigger monitor runmqtrm.

-tmq queueName
Specifies a trigger monitor queue name. If this option is used then WebSphere MQ process definitions are made to configure automatic triggering of SOAP/WebSphere MQ listeners with the associated trigger monitor queue name. If the option is not specified then no triggering configuration is made by the deployment utility. This option is mutually exclusive to the -s option.

If -s and -tmq are both specified, the deployment utility displays an error message and exits.

-u URI

Specifies a URI. By specifying the URI in this way at deploy time, the need to supply the URI with every client invocation is removed.

If a target service is specified in this URI, it must match the class supplied in the -f option. If a request queue is not specified in this URI, the queue name is generated as follows:

1. The full path name given in the -f parameter is taken and the file extension removed.
2. Any directory separator characters are replaced with period characters.
3. Any embedded spaces are replaced with underscore characters.
4. For a .NET service on Windows, a colon after any drive prefix is replaced with a period. The drive prefix itself is left intact.
5. The name is prefixed with ″SOAPJ.″ for Java services or with ″SOAPN.″ for .NET services.
6. The path name is truncated to no more than 48 characters, including the ″SOAPJ.″ or ″SOAPN.″ prefix. On platforms other than Windows, this is done by taking the ″SOAPJ.″ prefix and then appending up to a maximum of the rightmost 42 bytes. On Windows systems, the SOAPN. prefix is taken. Then, if the service being deployed is a .NET service, the first character and period following arel also taken if these originally denoted a drive prefix. This leaves a maximum of either 42 or 40 characters which will be taken from the right side of the string.

It is possible in some environments that a queue name generated by the supplied deployment utility might not be unique. Although there is protection against this via the validation process described in “Queue and directory validation" on page 38, you might choose to safeguard further against this by either restructuring the deployment directory hierarchy or by customizing the supplied deployment process.

It is only possible to nominate a single URI to the deployment utility. This URI is used in both the default client and listener configurations built by the deployment utility. One implication of this is that if a binding=client option is specified in the URI given to the deployment utility, then a configuration is built that assumes binding=client both at the sender and the listener. You might want to use the binding=auto option if you have no local queue manager on the client side and therefore require a client connection at the sender and a server connection at the listener. If you require different URIs at the client and listener you can either modify the configuration built by the deployment utility or build your own deployment utility.

-v Sets verbose output from external commands. (Error messages are displayed whether or not -v is set.) This can be useful for creating customized deployment scripts.

-x transactionality
The form of transactional control the listener should run under. **transactionality** can be set to one of the following values:

onePhase
WebSphere MQ one-phase support is used. If the system fails during processing, the request message is redelivered to the application. WebSphere MQ transactions ensure that the response messages are written exactly once. If the -x flag is omitted, or is used without a qualifying option, this is the default option assumed.

twoPhase
Two-phase support is used. If other resources are coordinated resource managers and the service is written appropriately the message is delivered exactly once with a single committed execution of the service.

This option applies to server bindings connections only.

none No transactional support. If the system fails during processing, the request message can be lost (even if persistent). The service might or might not have executed, and response, report or dead-letter messages might or might not have been written.

The deployment utility checks for the compatibility of the -x and -a flags. See the description of the -a flag for details.

-? Print out a help text describing how the command should be used.

SSL options
Options that can be specified for use with client connections over a channel configured to run in SSL mode are specified in Chapter 10, "Using SSL with WebSphere MQ transport for SOAP," on page 61.

Outputs

All outputs are generated into the ./generated subdirectory, and subdirectories of ./generated.

Before deploying a service, you should delete the ./generated subdirectory. This is particularly important if you are redeploying a modified service.

Files needed only by the client are placed in: ./generated/client and its subdirectories.

Files needed only by the server are placed in ./generated/server and its subdirectories.

The outputs are:
- classes: The Java service source file is compiled (into ./generated/server)
- wsdl: ./generated/*className*_Wmq.wsdl: wsdl
- wsdd (Axis service deployment files):
  - ./generated/server-config.wsdd
  - ./generated/client-config.wsdd
  - ./generated/server/*classDirectory/className*_deploy.wsdd
  - ./generated/server/*classDirectory/className*_undeploy.wsdd
- wsdd: Axis source for Java and .Net proxies (in ./generated/client and subdirectories)
- compiled Java proxies (in ./generated/client and subdirectories)
- Listener scripts: (in ./generated/server). Six scripts are generated:
  - startWMQJListener.cmd
  - startWMQJListener.sh
  - startWMQNListener.cmd
  - endWMQJListener.cmd
  - endWMQJListener.sh
  - endWMQNListener.cmd

Queue and directory validation

If you explicitly specify a request queue name, you might accidentally use an existing queue name. If you use generated queue names, a truncated queue name might not be unique. To protect against these problems, the deploy utility looks for the presence of an existing start up script in the generated/server subdirectory of the deployment directory. It then checks the URI specified to the listener in that script. It also checks whether the request queue already exists. Having made these checks, the deployment utility then takes one of the following six actions as appropriate:
- Request queue does not already exist
  - The listener start up script is not found in the generated/server directory. This is the case where no previous service has been deployed from this directory.

    Deployment continues with no warnings or errors.
  - The listener start up script is found but the request queue in the URI does not match that being used by the deployment utility. This is the case where a previously deployed service in this directory was deployed with a different queue.

    The deployment utility displays an error message and exits.
  - The listener start up script is found and the request queue matches that being used by the deploy utility This might indicate an incomplete or corrupted, but compatible, previous deployment.

    The deployment utility displays a warning message because the start up file was valid but the queue did not exist. Deployment continues and the request queue is created.

- Request Queue does already exist
  - The listener start up script is not found in the generated/server directory. This might indicate the queue is already in use for services deployed in another directory, or for some other application.

    The deployment utility displays an error message and exits.
  - The listener start up script is found but the request queue in the URI does not match that being used by the deployment utility. This is the case when a service has already been deployed in this directory, but using a different queue.

    The deployment utility displays an error message and exits.
  - The listener start up script is found and the request queue matches that being used by the deploy utility. This occurs when a service has already been deployed in this directory using the same queue.

    Deployment continues with no warnings or errors.

Using the deployment utility

Run the deployment utility using the amqwdeployWMQService command. Use the -f flag to specify the name of the source file. For Axis services this is the Java source file, and for .NET services, the .asmx file. The following example illustrates the use of anqwdeployWMQService for Axis

```
amqwdeployWMQService -f javaDemos/service/StockQuoteAxis.java
```

The following example illustrates the use of anqwdeployWMQService for .NET.

```
amqwdeployWMQService -f StockQuoteDotNet.asmx
```

Deployment performs the following actions. Most of the results of these actions are placed in the `generated` subdirectory of the directory from which the deployment is made, which is created if it does not already exist. If you are redeploying a service that has already been deployed, you should first delete the `generated` subdirectory. In the following examples amqwdeployWMQService is running from c:/temp/soap, so output is placed in c:/temp/soap/generated.

1. For Java services, compile the source into the c:/temp/soap/generated/server subdirectory, for example:

   ```
   c:/temp/soap/generated/server/javaDemos/service/StockQuoteAxis.class
   ```

   .NET services that are not written using code embedded in the .ASMX file must be compiled before calling anqwdeployWMQService.
2. Generate the appropriate WSDL. This is created in c:/temp/soap/generated/*className*_Wmq.wsdl, for example

   ```
   c:/temp/soap/generated/javaDemos.service.StockQuoteAxis_Wmq.wsdl
   ```
3. For Java services, prepare deployment descriptor files (*className*_deploy.wsdd and *className*_undeploy.wsdd), for example:

   ```
   c:/temp/soap/generated/server/javaDemos/service/StockQuoteAxis_deploy.wsdd
   ```

   and

   ```
   c:/temp/soap/generated/server/javaDemos/service/StockQuoteAxis_undeploy.wsdd
   ```

   and deploy into the execution directory to create or update c:/temp/soap/generated/server-config.wsdd.
4. Generate the appropriate proxies for Java, C# and Visual Basic from this WSDL. On Windows, proxies are generated for all three client languages regardless of the language in which the service is written. The C# and VB proxies are placed

directly into the c:/temp/soap/generated directory. The Java proxies are placed in the ./remote subdirectory of the original package and file directories to prevent confusion with the original classes. For example, the .Net proxies might be placed in:

```
c:/temp/soap/generated/client/StockQuoteAxisService.cs
```

and

```
c:/temp/soap/generated/client/StockQuoteAxisService.vb
```

and the various Java files might be placed in:

```
c:/temp/soap/generated/client/javaDemos/service/remote/*.java
```

5. Compile the Java proxies: for example to

   ```
   c:/temp/soap/generated/client/javaDemos/service/remote/*.class
   ```
6. Create a WebSphere MQ queue to hold requests to the service. The queue name is of the form SOAPJ.*directory*, for example: SOAPJ.demos.
7. Prepare a file to start the SOAP/WebSphere MQ listener that will process this queue, for example

   ```
   c:/temp/soap/generated/server/startWmqJListener.cmd
   ```
8. If the -tmq option has been used, then prepare WebSphere MQ definitions that will permit the SOAP/WebSphere MQ listener process to be automatically triggered. for example:

   WebSphere MQ Process: SOAPJ.demos WMQ

   Trigger Initiation Queue: SYSTEM.SOAP.INIT.QUEUE

The WSDL and the proxies generated from it will include the appropriate URI to call the service, for example:

```
jms:/queue?destination=myQ&connectionFactory=()&targetService=myService
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

Errors

On Windows, if errors are reported from amqswsdl, try issuing the following command:

```
%windir%/Microsoft.NET/Framework/version number/aspnet_regiis.exe -ir
```

This generally applies to systems where IIS has not been installed or IIS has been installed after NET. (For more information about installing IIS and .NET, and registering IIS, see “Prerequisites” on page 15.) The aspnet_regiis utility sets up the necessary registry keys to allow Windows to execute files with the .asmx extension as services. This is normally first encountered when amqswsdl generates the wsdl files at deploy time. If you use a customized deployment procedure that does not include this step, the registry keys are still required to permit the listener to invoke the services.

Restriction on deployment directory length

The deployment utility uses the APPLICID attribute of the runmqsc DEFINE PROCESS command to contain a command to start the listener. This will have the name of the deployment directory embedded in it. WebSphere MQ imposes a maximum length of 256 on the APPLICID field which in turn means there is a limit on the maximum length of the deployment directory.

For Java services, this limit is as follows:

- UNIX: 218
- Windows: 197 minus queue_name_length

For .NET services the limit is

- Windows: 209 minus length of service_name

where ″service_name″ is the name of the input service file without any extension. (For example, ″StockQuoteDotNet.asmx″ would have a service name of StockQuoteDotNet, which has a length of 16 characters, and so the corresponding maximum deployment directory length would be 193).

If you are using triggering, the deployment utility checks whether the limit for APPLICID is exceeded. If the limit is exceeded, the utility does not attempt to define the triggering process; it displays an error message and the deployment process fails with no deployment steps having been taken.

For more information on the DEFINE PROCESS command, see *WebSphere MQ Script (MQSC) Command Reference*.

Chapter 8. Senders and listeners

Senders

The SOAP/WebSphere MQ Java sender

A Java sender is implemented in the final class com.ibm.mq.soap.transport.jms.WMQSender which is derived from the org.apache.axis.handlers.BasicHandler class. When the Axis host environment detects the ″jms:″ URI prefix that was previously registered to the environment, the sender attempts to put the message on the specified request queue with any specific expiry, persistence and priority options.

In the synchronous environment, the sender then blocks until it has read a response from the response queue. The response message is then returned to the client. If no response is received within the timeout interval set in the URI, the sender throws an exception.

The SOAP/WebSphere MQ .NET sender

The .NET sender is implemented in the sealed class IBM.WMQSOAP. MQWebRequest. This class is derived from System.Net.WebRequest and System.Net.IwebRequestCreate. When the .NET environment detects the ″jms:″ URI prefix it attempts to place the message on the specified request queue, setting up any specific expiry, persistence and priority options.

In the synchronous environment, the method then creates an IBM.WMQSOAP.MQWebResponse object which reads the response message from the response queue and then returns it to the client.

Listeners

The deployment process (see “The deployment utility” on page 34) automatically creates wrapper scripts in the generated/server directory that set up and invoke the listener. Scripts are generated to start the listener as a WebSphere MQ service or by triggering. You can also start a listener manually.

The scripts to start listeners are named as follows:

startWMQJListener.sh
to start a Java listener under UNIX

startWMQJListener.cmd
to start a Java listener under Windows

startWMQNListener.cmd
to start a .NET listener under Windows

The scripts to define and start listeners as a WebSphere MQ service are named as follows:

defineWMQJListener.sh
to define and start a Java listener as a service under UNIX

defineWMQJListener.cmd
to define and start a Java listener as a service under Windows

defineWMQNListener.cmd
to define and start a .NET listener as a service under Windows

These invoke the start scripts described above.

The deployment process also generates scripts to stop listeners, named as follows:

endWMQJListener.sh
to end a Java listener under UNIX

endWMQJListener.cmd
to end a Java listener under Windows

endWMQNListener.cmd
to end a .NET listener under Windows

Java SOAP/WebSphere MQ listener

The Java SOAP/WebSphere MQ listener is implemented in the class com.ibm.mq.soap.axis.transport.jms.SimpleJavaListener.

The Java SOAP/WebSphere MQ listener calling syntax is:

```
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] [-?]
```

where

-u Specifies the URI of the service to be invoked. This parameter is required.

-a Allows the default behavior to be customized when it is not possible to write a failed request message to the dead letter queue.

DefaultMsgIntegrity
For non-persistent messages, the listener displays a warning message and continues to execute with the original message being discarded. For persistent messages, it displays an error message, backs out the request message so it remains on the request queue and exits. This default mode applies if the -a flag is omitted, or if it is specified with no option.

LowMsgIntegrity
For both persistent and non-persistent messages, the listener displays a warning and continues to execute, discarding the message.

HighMsgIntegrity
For both persistent and non-persistent messages, the listener displays an error message, backs out the request message so it remains on the request queue and exits. This mode is mutually exclusive with the -x none option. If these options are both specified, the listener displays an error message and exit.

-d Time, in milliseconds, for the SOAP/WebSphere MQ listener to stay alive after no request messages have been received (on any thread, if you are running multiple threads). If set to -1, the listener stays alive indefinitely. This is the default.

-i Specifies whether or not the listener should attempt to pass identity context.

passContext
The listener uses the sender’s context. This is the default.

owncontext
: The listener uses the context under which it was started

For more details of context passing, see "Context" on page 50.

-n Specifies the number of SOAP/WebSphere MQ listener threads required. The default is 10.

-v Display warning messages if any of the following options are specified in the -u argument:

- reply to queue
- timeout
- expiry
- persistence
- priority
- targetService

These options apply only to SOAP/WebSphere MQ clients. If you use the same URI for the listener and don't want to be warned that you are using redundant parameters, omit the -v parameter to suppress the warning messages. Whether or not -v is set or warning messages are output, the listener executes as normal.

-x Indicates what form of transactional control the listener should run under. Options can be set on this flag as follows:

onePhase
: WebSphere MQ one-phase support is used. If the system fails during processing, the request message can be redelivered to the application. WebSphere MQ transactions assure that the message is written exactly once. This is the default.

twoPhase
: Two-phase support is used. This is based on WebSphere MQ coordination. As long as other resources are coordinated resource managers and the service is written appropriately the message is delivered exactly once with a single committed execution of the service. This option applies only to server bindings (see "The connectionFactory parameter" on page 30).

none No transactional support. If the system fails during processing, the request message might be lost, even if it is persistent. The service might or might not have executed, and response, report or dead-letter queue messages might or might not have been written. If the -x none option is used, then the ″-a LowMsgIntegrity″ option is mandated and the listener exits on start-up with an error message if the latter is not specified.

The queue of the URI determines the queue that the listener will monitor for service requests. If you are starting the listener manually, you normally run this command from a command prompt. The CLASSPATH should first be correctly set up by invoking the amqwsetcp.sh script. The Axis configuration directory defaults to the current working directory and the Axis configuration filename defaults to server-config.wsdd.

-? Provide a usage statement. The usage statement is displayed and the listener then exits.

For example:

```
 java com.ibm.mq.soap.transport.jms.SimpleJavaListener
-u "jms:/queue?destination=myQ&connectionFactory=()
&initialContextFactory=com.ibm.mq.jms.Nojndi"
-n 20
```

In the above example, the Java virtual machine is invoked to run the program com.ibm.mq.soap.transport.jms.SimpleJavaListener. Two arguments are supplied, the WebSphere MQ URI, which is identified with the -u qualifier, and the number of listener threads to start, which is identified with the -n parameter. The URI has the same form as for the client, but is used in a slightly different way. Optional values in the URI can be used to control the way the connection occurs (client/server bindings, which queue manager, and so on). These are: connectQueueManager, binding, clientChannel, clientConnection, sslCipherSuite, sslPeerName, sslKeyResetCount, sslKeyStore, sslKeystorePassword, ssFipsRequired, sslTrustStore, sslTrustStorePassword, and sslLDAPCRLServers. Other optional values in the URI are only relevant to clients and are ignored, but a warning message can be issued; see the explanation of the -v parameter.

.NET SOAP/WebSphere MQ listener

The .NET SOAP/WebSphere MQ listener is implemented in amqwSOAPNETlistener.exe.

The calling syntax is:

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

where:

-u Specifies the URI of the service to be invoked. This option is required.

-a Allows the default behavior to be customized when it is not possible to write a failed request message to the dead letter queue.

DefaultMsgIntegrity
For non-persistent messages, the listener displays a warning message and continues to execute with the original message being discarded. For persistent messages, it displays an error message, backs out the request message so it remains on the request queue and exits. This default mode applies if the -a flag is omitted, or if it is specified with no option.

LowMsgIntegrity
For both persistent and non-persistent messages, the listener displays a warning and continues to execute, discarding the message.

HighMsgIntegrity
For both persistent and non-persistent messages, the listener displays an error message, backs out the request message so it remains on the request queue and exits. This mode is mutually exclusive with the -x none option. If these options are both specified, the listener displays an error message and exits.

-d Time, in milliseconds, for SOAP/WebSphere MQ listener to stay alive after no request messages have been received (on any thread, if you are running multiple threads). If set to -1, the listener stays alive indefinitely. This is the default.

-i Specifies whether or not the listener should attempt to pass identity context.

passContext
The listener uses the sender's context. This is the default.

owncontext
The listener uses the context under which it was started

For more details of context passing, see "Context" on page 50.

-n Specifies the number of SOAP/WebSphere MQ listener threads required. The default is 10.

-v Display warning messages if any of the following options are specified in the -u argument:

- reply to queue
- timeout
- expiry
- persistence
- priority
- targetService

These options apply only to SOAP/WebSphere MQ clients. If you use the same URI for the listener and don't want to be warned that you are using redundant parameters, omit the -v parameter to suppress the warning messages. Whether or not -v is set or warning messages are output, the listener executes as normal.

-w Physical directory containing web service. The default is 'c:\\inetpub\\wwwroot\\<Application>\\' (extracted from Queue if not specified)

-x Indicates what form of transactional control the listener should run under. Options can be set on this flag as follows:

onePhase
WebSphere MQ one-phase support is used. If the system fails during processing, the request message can be redelivered to the application. WebSphere MQ transactions assure that the message is written exactly once. This is the default

twoPhase
two-phase support is used. As long as other resources are coordinated resource managers and the service is written appropriately the message is delivered exactly once with a single committed execution of the service. This option applies only to server bindings (see "The connectionFactory parameter" on page 30).

none No transactional support. If the system fails during processing, the request message might be lost, even if it is persistent). The service might or might not have executed, and response, report or dead-letter queue messages might or might not have been written. If the -x none option is used, then the ″-a LowMsgIntegrity″ option is mandated and the listener exits on start-up with an error message if the latter is not specified.

The queue of the URI determines the queue that the listener will monitor for service requests. If you are starting the listener manually, you normally run this command from a command prompt.

-? Provide a usage statement. The usage statement is displayed and the listener then exits.

For example:

```
amqwSOAPNETlistener -u "jms:/queue?destination=myQ&connectionFactory=()
&targetService=myService&initialContextFactory=com.ibm.mq.jms.Nojndi"
-w C:/wmqsoap/demos -n 20
```

In the above example, the listener is started by running the program amqwSOAPNETlistener. Three arguments are supplied, the WebSphere MQ URI, which is identified with the -u qualifier, the directory the service is located in, which is identified by the -w parameter, and the number of listener threads to start, which is identified with the -n parameter. The URI has the same form as for the client, but is used in a slightly different way. Optional values in the URI might be used to control the way the connection occurs (client/server bindings, which queue manager, and so on). These are: connectQueueManager, binding, clientChannel, clientConnection, sslKeyRepository, sslCipherSpec, sslPeerName, sslKeyResetCount, sslCryptoHardware, sslFipsRequired and sslLDAPCRLServers. Other optional values in the URI are relevant only to clients and are ignored, but a warning message can be issued; see the explanation of the -v parameter.

Channel definition tables

As an alternative to creating a client connection channel definition by setting certain properties of a ConnectionFactory object, you can use client connection channel definitions that are stored in a client channel definition table. Channel definition tables are described in *WebSphere MQ Using Java* and *WebSphere MQ Clients*. They are specified in the normal manner for the Java and .NET environments, as detailed below.

Java

The channel definition table is specified as a URI (for example file://*mqmtop*/qmgrs/*QUEUEMANAGERNAME*/@ipcc/AMQCLCHL.TAB). The system property for the channel definition table URI is com.ibm.mq.soap.transport.jms.mqchlurl. This URI is passed to the SOAP/WebSphere MQ client (or listener) via a system property on the command line. You cannot set the channel definition table URI via the SOAP/WebSphere MQ URI.

If clientChannel, clientConnection, or any SSL options are specified in the SOAP/WebSphere MQ URI or as system properties when a SOAP client (or listener) makes a WebSphere MQ client connection, these take priority over any channel definition table. In these circumstances the channel definition table is not used and a warning message to this effect is displayed.

.NET

The .NET classes for WebSphere MQ support the use of client definition tables through the environment variables MQCHLLIB and MQCHLTAB. MQCHLLIB specifies the directory where the table is located and MQCHLTAB specifies the actual filename of the table. You cannot specify the channel definition table directly in a SOAP/WebSphere MQ URI.

Starting listeners by triggering

You can cause SOAP/WebSphere MQ listeners to start automatically by using triggering. This is done by setting the **-tmq** option and supplying a trigger queue name when you run the deployment utility. You can also specify a trigger monitor program by setting the **-tmp** option.

For example:

```
amqwdeployWMQService -f javaDemos/service/StockQuoteAxis.java -tmq trigger.monitor.queue
-tmp trigmon
```

When SOAP/WebSphere MQ listeners are activated by the trigger monitor process, they normally run under the user ID that invokes the trigger monitor. However, if the trigger monitor is activated on UNIX systems with a setuid trigger monitor, such as the supplied runmqtrm utility, the listeners execute under the mqm user id. Where security is an issue, you must ensure that the listeners are started under appropriate user IDs

Triggering is described in *WebSphere MQ Application Programming Guide* and *WebSphere MQ Intercommunication*. The deployment utility is described in Chapter 7, “Deployment,” on page 33.

Terminating listeners

The deployment utility generates a script in the generated/server directory that you can use to close down a listener. The script is called: endWMQJListener.cmd (for Windows) or endWMQJListener.sh (for UNIX) for Java services, and endWMQNListener.cmd for .NET services.

You can also terminate listeners by setting GET DISABLED on the request queue.

Report messages

The WebSphere MQ transport for SOAP sender code sets the MQRO_EXCEPTION_WITH_FULL_DATA and MQRO_EXPIRATION_WITH_FULL_DATA report options. This results in report messages being written to the response queue in the event of an exception or message expiry condition. SOAP/WebSphere MQ listeners also generates report messages if the format of the request message is not recognized or fails a basic integrity check of the MQRFH2 header, or if the backout/retry threshold is exceeded while a SOAP/WebSphere MQ listener is trying to process the request. The use of these report options means that report messages contain the entire originating request message. Your client application should get these messages and process them appropriately.

The WebSphere MQ transport for SOAP sender code sets the MQRO_DISCARD report option. This option causes a message to be discarded after a report message has been returned, rather than being written to the dead-letter queue. The provided SOAP/WebSphere MQ listeners honour the case where MQRO_DISCARD has not been set. If SOAP/WebSphere MQ generates a report message but fails in the process of sending the report, the normal behavior is that the report message is sent to the dead-letter queue (DLQ). Ensure that your DLQ handler handles these messages correctly. (See *WebSphere MQ System Administration Guide* for information about DLQ handlers.) If the sender’s report options do not meet your requirements you will have to write your own senders to use different

MQRO_EXCEPTION and MQRO_DISCARD report options. (See“Writing WebSphere MQ transport for SOAP senders” on page 53 for information about writing your own senders.)

If an error occurs when attempting to write to the dead-letter queue (for example if the dead letter queue is full), a message is written to the WebSphere MQ error log. If the listener is running in its default transactional mode of OnePhase with a non-persistent request message, the original message is not left in the request queue and the SOAP/WebSphere MQ listener continues to execute. If the request message is persistent it is backed out to the request queue and the listener exits. In this case the request queue is set to get-inhibited to prevent an accidental triggered restart.

Context

Both the Java and .NET SOAP/WebSphere MQ listeners have runtime options for specifying whether they pass identity context. The default if the parameter is not specified is that they pass context.

If the listener is to pass identity context, it checks at runtime that it has authority both to save the context from the request queue and to pass it to the response queue when opening the response queue. If the listener does not have sufficient authority the message is put on the dead-letter queue with the return code set to that returned from the failed open call on the response queue. The listener then continues to process subsequent incoming messages as normal. If the listener does have the required authority and the open call is successful, the listener sets the identity context of the original request message into the response message.

If the listener is started and told not to pass identity context, the listener honours the runtime option and does not pass context. The returned context reflects the user ID under which the listener is running rather than the user ID which created the original request message.

The fields in the origin context are not set by SOAP/WebSphere MQ but by the queue manager.

See *WebSphere MQ Application Programming Guide* for details of identity context and origin context.

Part 3. Further considerations

Chapter 9. Customizing WebSphere MQ transport for SOAP

This chapter discusses how to customize WebSphere MQ transport for SOAP. There are three areas for customization:

- deployment, see "Customizing the deployment process"
- senders, see "Writing WebSphere MQ transport for SOAP senders"
- listeners, see "Writing WebSphere MQ transport for SOAP listeners" on page 55

Source code for a sample deployment utility is installed into *mqmtop*/tools/soap/samples on Windows systems or *mqmtop*/samp/soap on UNIX systems. You can use this sample as the basis of you own deployment utility, as required.

Customizing the deployment process

The behavior of the supplied deployment utility is described in Chapter 7, "Deployment," on page 33. That chapter describes how to control the behavior of the deployment utility using its optional parameters. If you want to change the deployment process beyond the capabilities of the supplied utility, perhaps to deploy from .wsdl rather than from source code, you can write your own utility. If you want to your utility to work in a similar way to the supplied utility, you can find related source code in the samples directory as amqwdeployWMQService.java.

You can also customize deployment by running the provided utility with the **-v** option set, and capturing the commands that are output. You can then assemble these into a script and edit them appropriately, for example using different arguments in the Java2WSDL or WSDL2Java steps.

Writing WebSphere MQ transport for SOAP senders

Attention: This is recommended for advanced users only.

The behavior of the supplied senders is described in "Senders" on page 43. Though the supplied senders can be replaced with your own code, sample source code is not supplied.

One function of the sender is to construct the required message headers. If you implement your own sender code and these fields are not properly set, results are undefined. The contents of the message headers are described in "Constructing message headers" on page 55.

If the sender's report options do not meet your requirements (see "Report messages" on page 49), you will have to write your own senders to use different MQRO_EXCEPTION and MQRO_DISCARD report options. The SOAP/WebSphere MQ listeners honour such settings as follows (The notation MQRO_EXCEPTION_* indicates the use of either MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA or MQRO_EXCEPTION_WITH_FULL_DATA):

MQRO_EXCEPTION_* enabled, MQRO_DISCARD enabled
Default behavior. Report messages are automatically generated if necessary

and the original request discarded. If a report message could not be returned to the response queue the report message is sent to the dead letter queue.

MQRO_EXCEPTION_* enabled, MQRO_DISCARD not enabled
Report messages are automatically generated if necessary and the original message is sent to the dead letter queue. If the report message could not be returned to the response queue it is also be sent to the dead-letter queue. In this case there are therefore two dead letter queue entries for the failed request.

MQRO_EXCEPTION_* not enabled, MQRO_DISCARD not enabled
Report messages are not automatically generated when the incoming format is not recognized or the backout retry count is exceeded. The original request message is however written to the dead letter queue when a report would otherwise have been generated.

MQRO_EXCEPTION_* not enabled, MQRO_DISCARD enabled
Report messages are not automatically generated when the incoming format is not recognized or the backout retry count is exceeded. The message is not sent to the dead letter queue. This means that there is no return notification that the client can inspect and the original request message is lost.

The following lists show whether other report options are honoured in the SOAP/WebSphere MQ listeners.

- Report options supported with SOAP/WebSphere MQ:
  - MQRO_EXCEPTION
  - MQRO_EXCEPTION_WITH_DATA
  - MQRO_EXCEPTION_WITH_FULL_DATA
  - MQRO_DEAD_LETTER_Q
  - MQRO_DISCARD_MSG
  - MQRO_NONE
  - MQRO_NEW_MSG_ID
  - MQRO_PASS_MSG_ID
  - MQRO_COPY_MSG_ID_TO_CORREL_ID
  - MQRO_PASS_CORREL_ID
- Report options handled by the queue manager with no intervention from SOAP/WebSphere MQ:
  - MQRO_COA
  - MQRO_COA_WITH_DATA
  - MQRO_COA_WITH_FULL_DATA
  - MQRO_COD
  - MQRO_COD_WITH_DATA
  - MQRO_COD_WITH_FULL_DATA
  - MQRO_EXPIRATION
  - MQRO_EXPIRATION_WITH_DATA
  - MQRO_EXPIRATION_WITH_FULL_DATA
- Unsupported report options:
  - MQRO_PAN
  - MQRO_NAN

Writing WebSphere MQ transport for SOAP listeners

Attention: This is recommended for advanced users only.

The behavior of the supplied listeners is described in "Listeners" on page 43. Though the supplied listeners can be replaced with your own code, sample source code is not supplied.

Constructing message headers

A SOAP/WebSphere MQ format message consists of a standard WebSphere MQ header (MQMD), a WebSphere MQ Rules and formatting header (MQRFH2), and a body that contains a standard SOAP message. SOAP/WebSphere MQ provides no constraint on this body; it regards it simply as an unformatted byte string and relies on the host Web services environment to provide SOAP formatting and parsing services.

Constructing the message descriptor

This section describes how the message descriptor (MQMD) is set up by the supplied senders. If you implement your own sender code and these fields are not properly set, results are undefined. Each message descriptor field can be set differently for the three types of SOAP/WebSphere MQ message (Request, Response, and Report). Where the setting is common across the three types they are grouped together as "All". For more details of the MQMD, see *WebSphere MQ Application Programming Reference*.

StrucId
Structure identifier.

All message types: MQMD_STRUC_ID

Version
Structure version number.

All message types: MQMD_VERSION_2

Report
Options for report messages.

- Request: SOAP/WebSphere MQ sets the option MQRO_COPY_MSG_ID_TO_CORREL_ID in this flag so that the WebSphere MQ generated message ID is automatically returned in any response message as the correlation ID. It also sets MQRO_EXCEPTION, MQRO_EXPIRY, and MQRO_DISCARD so that report messages are returned to the client if an exception is generated or the request message expires, and so that a message is discarded after a report message has been returned in the event of an error, rather than being written to the Dead Letter Queue.
- Other types: This field is not used.

MsgType
Message type.

- Request: MQMT_REQUEST
- Response: MQMT_REPLY
- Report: MQMT_REPORT

Expiry Message lifetime.

- Request: By default this field is set to MQEI_UNLIMITED, meaning an unlimited expiry time. However this can be overridden by specifying the Expiry option in the target URI. (See Chapter 6, “Specifying the URI,” on page 29).
- Response: The value of this field is passed by WebSphere MQ from a request message to any corresponding response message.
- Report: The field is set to MQEI_UNLIMITED.

Feedback
Feedback or reason code.

- Request, Response: Not used
- Report: For report messages that are generated by WebSphere MQ this field is automatically set according to normal conventions. Report messages that are generated directly by the SOAP/WebSphere MQ listeners are raised either because the backout retry threshold has been exceeded or because the format of the request message was not recognized. If the backout retry threshold was exceeded the field is set to MQRC_BACKOUT_THRESHOLD_REACHED. For problems with the request message format, the feedback is set to one of the following values:
  - MQRCCF_MD_FORMAT_ERROR - The message is not recognized as having an MQRFH2 header.
  - MQRC_RFH_PARM_MISSING - A required parameter (for example, soapAction) in the MQRFH2 is missing.
  - MQRC_RFH_FORMAT_ERROR - A basic integrity check of the MQRFH2 failed (for example, internal lengths are corrupt).
  - MQRC_RFH_ERROR - The MQRFH2 passed an integrity check, but the body of the message is not set to MQFMT_NONE.

Encoding
Numeric encoding of message data.

All: platform default.

CodedCharSetId
Character set identifier of message data.

All: UTF-8.

Format
Format name of message data.

- Request, Response: MQFMT_RF_HEADER_2.
- Report: For report messages generated by WebSphere MQ the format is automatically set according to normal WebSphere MQ conventions. Where SOAP/WebSphere MQ explicitly creates a report message, the format of the original request is preserved.

Priority
Message priority.

- Request: The priority is by default set to MQC.MQPRI_PRIORITY_AS_Q_DEF. It can be overridden in the URI. See Chapter 6, “Specifying the URI,” on page 29.
- Response, Report: The value of this field is passed by WebSphere MQ from a request message to any corresponding response or report message.

Persistence
Message persistence.
- Request: The default persistence is taken from the queue definitions.
- Response, Report: The value of this field is passed by WebSphere MQ from a request message to any corresponding response or report message.

MsgId Message identifier.
- Request, Report: This field is automatically completed by WebSphere MQ and is unique to each message.
- Response: The supplied SOAP/WebSphere MQ senders request that the MsgId should be uniquely generated in a response message. If you write your own sender, the SOAP/WebSphere MQ listeners honour other options supported in WebSphere MQ for setting the MsgId.

CorrelId
Correlation identifier.
- Request: This field is used only in asynchronous service requests where it holds the client id. See Chapter 12, "Asynchronous messaging," on page 71 for more information.
- Report, One way: This field is not used.
- Response: The supplied SOAP/WebSphere MQ senders request that the CorrelId should be set in a response message to the MsgId of the request message. If you write your own sender, the SOAP/WebSphere MQ listeners honour other options supported in WebSphere MQ for setting the CorrelId.

BackoutCount
Backout counter .
- Request: This field is used to detect messages that should be discarded or put to the dead-letter queue.
- Response, Report: This field is not used.

ReplyToQ
Name of reply queue.
- Request: SOAP/WebSphere MQ sets the reply-to queue to SYSTEM.SOAP.RESPONSE.QUEUE. This default can be overridden by setting the target URI appropriately.

Response, Report: Not used.

ReplyToQMgr
Name of reply queue manager.

All: Set by the WebSphere MQ queue manager.

UserIdentifier
User identifier.
- Request, Report: This field is not used.
- Response: How this and other fields in the Identity Context are set in response messages depends on the authorization the listener is running under. See "Context" on page 50 for more details.

AccountingToken
Accounting token.

All: This field is not used.

ApplIdentityData
Application data relating to identity.

All: This field is not used.

PutApplType
Type of application that put the message.

All: This field is set by the queue manager.

PutApplName
Name of application that put the message.

All: This field is completed by the queue manager. In the Java environment it is normally set to ″java.exe″ and in the .NET environment to ″dir/progName.exe″.

PutDate
Date when message was put.

All: This field is completed by the queue manager.

PutTime
Time when message was put.

All: This field is completed by the queue manager.

ApplOriginData
Application data relating to origin

This field is completed by the queue manager.

GroupId
Group identifier

- Request, Report: This field is not used.
- Response: Although there is no specific use of the GroupId field in SOAP/WebSphere MQ, the field is preserved by SOAP/WebSphere MQ listeners.

MsgSeqNumber
Sequence number of logical message within group

- Request, Report: This field is not used
- Response: This field is preserved by SOAP/WebSphere MQ listeners.

Offset Data offset in physical message

- Request, Report: This field is not set.
- Response: This field is preserved by SOAP/WebSphere MQ listeners.

MsgFlags
Message flags

- Request, Report: This field is not used..
- Response: This field is preserved by SOAP/WebSphere MQ listeners.

OriginalLength

Length of original message

Request, Response: This field is not set by SOAP/WebSphere MQ.

Report: The field is set to the length of the original request message.

Constructing the MQRFH2 header

This section describes how the supplied senders set up the second message header, which is in the MQRFH2 format. The settings are common across the three message types. For more details of the MQRFH2, see *WebSphere MQ Application Programming Reference*.

Fixed portion

StrucId
Structure identifier

MQRFH_STRUC_ID

Version
Structure version number

MQRFH_VERSION_2

StrucLength
Total length of this header including all NameValueLength and NameValueData fields

Encoding
Numeric encoding of data that follows last NameValueData field

CodedCharSetId
Character set identifier of data that follows last NameValueData field

Set to UTF-8

Format
Format name of data that follows last NameValueData field

Flags Flags

NameValueCCSID
Character set identifier of NameValueData

Set to UTF-8

Variable portion

The variable portion of an MQRFH2 header consists of a number of occurrences of the fields NameValueLength and NameValueData. This is described in *WebSphere MQ Application Programming Reference*. WebSphere MQ transport for SOAP uses the same NameValueLength and NameValueData fields as WebSphere MQ JMS messages. These are detailed in *WebSphere MQ Using Java*. The NameValueData fields containing the <mcd> and <jms> folders must be set up as described in *WebSphere MQ Using Java*. The <mcd>, <jms> and <usr> folders (and no others) must occur in that order. The NameValueData field containing the <usr> folder must be set up as follows

NameValueLength
Length of matching NameValueData. Must be a multiple of four.

NameValueData
Contains a folder named <usr>, which contains five name/value pairs, as follows:

contentType
This always contains the string ″text/xml; charset=utf-8″.

endpointURL
URI of the web service to be invoked.

targetService
A copy of the targetService field from endpointURL.

soapAction
This field is mandatory for .NET services but optional for Axis services. Its value depends on the service to be invoked.

transportVersion
This always contains the value 1.

clientId
String used to distinguish between the responses to different requests for the same service.

sideQueueReason
Indicates whether this is a queue mapping or context entry

sideQueueBaseName
Base name of response queue

sideQueueExpandedQueueName
Real name of response queue

Chapter 10. Using SSL with WebSphere MQ transport for SOAP

WebSphere MQ transport for SOAP provides several SSL options that can be specified in the WebSphere MQ URI for use with client connections over a channel configured to run in SSL mode. There are differences in these options between the .NET and Java environments but the SOAP/WebSphere MQ senders and listeners process the SSL options that are applicable to that particular environment and ignore those which are not.

The presence or absence of the sslCipherSpec option for .NET clients and the sslCipherSuite option for Java clients determines whether SSL is used or not. If the option is not specified in the URI then by default SSL is not used and all other SSL options are ignored. All SSL options are optional except where indicated.

For WebSphere MQ clients, where a local queue manager is not used, you should set the SSL attributes in the URI or channel definition table. On the server, you should set them using the facilities of WebSphere MQ. By default, the standard WebSphere MQ SSL option ″Always authenticate parties initiating connections to this channel definition″ is set when enabling SSL on the channel. This means that clients are required to authenticate themselves before SSL communication can commence. They do this by sending their certificate to the server system. If this option is not set, then SSL communications are established without client authentication. If using client authentication, it is essential that the client's key repository has a certificate assigned which is acceptable to the queue manager. For additional security, WebSphere MQ channels can be configured to only accept certificates the Distinguished Names of which match a required set of values. If an SSL Peer Name is set on a channel, the client's certificate must match the values specified in SSL Peer Name. Refer to *WebSphere MQ Security* for details on the use and specification of the SSL Peer Name parameter for WebSphere MQ channels. The parameter is called SSLPEER when it is used in the MQSC DEFINE CHANNEL command.

In SOAP/WebSphere MQ the only difference in this specification is that the entire SSL Peer Name string in the URI for these connections has to be enclosed in parentheses. This is shown in the following example: SSLPeerName=″(CN=MQ Test 1,O=IBM,S=Hampshire,C=GB)″

For more details on the CipherSpecs and CipherSuites supported, refer to *WebSphere MQ Security* and to *WebSphere MQ Using Java*. For information about using the MQSCO structure on an MQCONNX call, see *WebSphere MQ Application Programming Reference*.

SSL-related options in the URI

The SSL options provided are:

sslKeyRepository=KeyRepository
For SSL enabled client connections, this specifies the location of the SSL key repository in which SSL keys and certificates are stored. This is specified in ″stem″ format, that is, a full path with file name but with the file extension

omitted. The effect is the same as setting the KeyRepository field in the MQSCO structure on an MQCONNX call (see *WebSphere MQ Application Programming Reference* for details).

This property applies to the .NET client environment only and is mandatory if sslCipherSpec is set. It is ignored in the Java environment or if sslCipherSpec is null.

sslCipherSpec=CipherSpec
For SSL enabled client connection, this specifies the SSL CipherSpec used on the channel. For more information about CipherSpecs, including a list of the CipherSpecs that can be used with WebSphere MQ SSL support, see *WebSphere MQ Security*.

This property applies to the .NET client environment only and is mandatory if SSL is being used. It is ignored in the Java environment.

sslCipherSuite=CipherSuite
For SSL enabled client connection, this specifies the SSL CipherSuite used on the channel. For more information about CipherSuites including a list of CipherSuites that can be used with WebSphere MQ SSL support, see *WebSphere MQ Using Java*.

This property applies to the Java client environment only and is mandatory in if SSL is being used. It is ignored in the .NET environment.

sslPeerName=PeerName
For SSL enabled client connections, this specifies an SSL peer name. The format of an SSL peer name is described in *WebSphere MQ Script (MQSC) Command Reference*.

This property is ignored if sslCipherSpec (for .NET) or sslCipherSuite (for Java) is null.

sslKeyResetCount=bytecount
For SSL enabled client connections, this specifies the number of bytes passed across an SSL channel before the SSL secret key must be renegotiated. To disable the renegotiation of SSL keys the field can either be omitted or set to 0. The effect is the same as setting the KeyResetCount field in the MQSCO structure on an MQCONNX call (see *WebSphere MQ Application Programming Reference* for details).

This property is ignored if sslCipherSpec (for .NET) or sslCipherSuite (for Java) is null.

This property should not be used in certain Java environments, see *WebSphere MQ Using Java* for details.

sslCryptoHardware=cryptographic hardware details
For SSL enabled client connections, this specifies details relating to the cryptographic hardware to be used. The possible values for this field, and the effect of setting it, are the same as for the CryptoHardware field of the MQSCO structure on an MQCONNX call (see *WebSphere MQ Application Programming Reference* for details).

This property applies to the .NET environment only. It is ignored in the Java environment or if sslCipherSpec is null.

sslFipsRequired=YES | NO
For SSL enabled client connections, this specifies whether the CipherSpecs or CipherSuites requested must use FIPS-certified cryptography in WebSphere MQ. The default value is NO. The effect of setting this field is the same as

setting the FipsRequired field of the MQSCO structure on an MQCONNX call (see *WebSphere MQ Application Programming Reference* for details).

This property is ignored if sslCipherSpec (for .NET) or sslCipherSuite (for Java) is null.

sslKeyStore=key store name
For SSL enabled client connections, this specifies the JSSE key store.

This property applies to the Java environment only. It is ignored in the .NET environment or if sslCipherSuite is null. For information about keystores, see *WebSphere MQ Using Java*.

sslKeyStorePassword=password
For SSL enabled client connections, this specifies the password for the JSSE key store.

This property applies to the Java environment only. It is ignored in the .NET environment or if sslCipherSuite is null. For information about keystores, see *WebSphere MQ Using Java*.

sslLDAPCRLServers=LDAP server list
For SSL enabled client connections, this specifies a list of LDAP servers to be used for Certificate Revocation List checking This string must consist of a sequence of space-delimited LDAP URIs of the form ldap://host[:port]. If no port is specified, the LDAP default of 389 is assumed. The certificate provided by the queue manager is checked against one of the listed LDAP CRL servers; if found, the connection fails. Each LDAP server is tried in turn until connectivity is established to one of them. If it is impossible to connect to any of those specified, the certificate is rejected. Once a connection has been successfully established to one of them, the certificate is accepted or rejected depending on the CRLs present on that LDAP server. If sslLDAPCRLServers is set to null (the default), the queue manager's certificate is not checked against a Certificate Revocation List. An error message is displayed if the supplied list of LDAP URIs is not valid. The effect of setting this field is the same as that of including MQAIR records and accessing them from an MQSCO structure on an MQCONNX call (see *WebSphere MQ Application Programming Reference*).

This property is ignored if sslCipherSpec (for .NET) or sslCipherSuite (for Java) is null.

sslTrustStore
For SSL enabled client connections, this specifies the JSSE trust store.

This property applies to the Java environment only. It is ignored in the .NET environment or if sslCipherSuite is null. For information about truststores, see *WebSphere MQ Using Java*.

sslTrustStorePassword
For SSL enabled client connections, this specifies the password for the JSSE trust store.

This property applies to the Java environment only. It is ignored in the .NET environment or if sslCipherSuite is null. For information about truststores, see *WebSphere MQ Using Java*.

If you use Java, the first SSL connection from a SOAP/WebSphere MQ client causes the following SSL parameters to become fixed for subsequent connections on this client process:

- sslKeyStore
- sslKeyStorePassword

- sslTrustStore
- sslTrustStorePassword
- sslFipsRequired
- sslLDAPCRLservers

The effect of varying these parameters on subsequent connections from this client is undefined.

If you use .NET, the first SSL connection from a SOAP/WebSphere MQ client causes the following SSL parameters to become fixed for subsequent connections on this client process:
- sslKeyRepository
- sslCryptoHardware
- sslFipsRequired
- sslLDAPCRLservers

The effect of varying these parameters on subsequent connections from this client is undefined. These parameters are reset if all SSL connections become inactive and a new SSL connection is subsequently made.

The following properties can also be specified as system properties:
- sslKeyStore
- sslKeyStorePassword
- sslTrustStore
- sslTrustStorePassword

If they are specified both as system properties and in the URI, and the values differ, the deployment utility displays a warning and the URI values take precedence.

Chapter 11. Transactional processing

SOAP/WebSphere MQ can be used to transport messages in the context of Web Services transactions. However, you must ensure that if you send messages using SOAP/WebSphere MQ within a Web Services transaction they do not attempt to use local one-phase transactions.

The only transaction coordinator supported for Axis services is WebSphere MQ Java classes and the only transaction coordinator supported for .NET services. is .NET MTS.

Java and .NET clients

Running the standard request/response model transactionally within SOAP/WebSphere MQ uses a standard three transaction pattern:

1. A client sends a request, and performs other actions.
2. A listener accepts the request, executes it, and sends a response.
3. The client accepts the response, and performs other actions.

The client send and client accept must be in different transactions. This means that you must code your client process as a transactional object using the asynchronous request/response pattern.

In the Java environment your client application must issue MQQueueManager.begin() and MQQueueManger.commit() or MQQueueManager.backout() calls as appropriate. In .NET, you must write a class derived from System.EnterpriseServices.ServicedComponent and call ContextUtil.setComplete() or ContextUtil.setAbort() (or use equivalent autocompletion) as appropriate.

Java and .NET listeners

Transactionality at the listener is independent of that at the client. The listener provides support for transactionality in the WebSphere MQ Version 6.0 product and does not depend on SupportPac MA0V for this, whereas to use a client transactionally the SupportPac will be required

Use the -x option in the deployment utility to specify the level of transactional control to be used by the listener (one phase, two phase or none). See the description of the -x option in “The deployment utility” on page 34 for full details.

Multiple connections in transactions

If you are using Java, there are limitations on the use of multiple connections in transactions. (This applies only to Java; there are no equivalent issues in the .Net environment.) If you want to make explicit use of WebSphere MQ in Web services and Web services clients, in addition to the implicit WebSphere MQ calls provided by the SOAP/WebSphere MQ infrastructure, you must be aware of these and obtain connections (MQQueueManager objects) accordingly. Issue a begin() call on the queue manager object to initiate a transaction.

Transactional processing

On the client side transactions are managed by the client code (both in sending the request message, and in the client managed handling of the asynchronous response). The client code must obtain MQQueueManager objects using a MQ_QMGR_ASSOCIATION value of ASSOCIATE_THREAD. This name/value pair must be set in the properties `Hashtable props` and used in one of the constructors

```
public MQQueueManager(String qmName, Hashtable props)
public MQQueueManager(String qmName, Hashtable props, ConnectionManager cm)
public MQQueueManager(String qmName, Hashtable props, MQConnectionManager cm)
```

for example

```
Hashtable props = new Hashtable();
props.put(MQ_QMGR_ASSOCIATION, new Integer(ASSOCIATE_THREAD));
MQQueueManager myQM = new MQQueueManager("", props);
```

If you require coordination with database resources, you must follow the installation and usage rules set out in *WebSphere MQ Using Java*, in the section on *JTA/JDBC coordination using WebSphere MQ base Java*. In particular, the database connection must be created using MQQueueManager.getJDBCConnection(XADataSource).

Sample programs

Sample programs are provided to demonstrate transactional processing. On Windows, two samples are provided, these are SQCS2DotNetAsyncReqRespTran.cs and SQAxis2AxisAsyncReqRespTran.java. On UNIX only the Java version is supplied. Prepare these programs in a separate directory to that used for the other supplied samples (see "Samples" on page 22) because they use different request queues and different scripts to start and stop the listeners.

Build the transactional samples with the regenTranDemoasync.cmd script on Windows or regenTranDemoasync.sh on UNIX systems.

The samples take the following arguments:

command
Mandatory. Must be one of *request*, *response* or *requestresponse*.

url
The SOAP/WebSphere MQ URI for the service.

clientId
This is a string which can be used to group different client requests together.

tranArg
String argument supplied to the service - if this is set to "rollback" the listener's request/response transaction will abort. Any other value has no effect.

messageIntegrity
Must be one of *defaultmsgintegrity*, *lowmsgintegrity*, *highmsgintegrity*. These have the same meanings as when they are used in the **-a** option of the deployment utility, see "The deployment utility" on page 34.

The samples must either be run with just the command argument or with all the other arguments specified. For example, you cannot supply just the **command** and **tranArg** arguments.

To demonstrate a synchronous request and response being transacted within a single process, run the program once with **command** set to *requestresponse*. To demonstrate transactionality with an asynchronous service that lasts beyond the lifetime of the client processor, run the program twice, first with **command** set to **request** and then with **command** set to *response*.

If you specify **tranArg** as *rollback* the test service aborts the listener transaction, which causes the listener's request/response transaction to be backed out. The samples behave differently in Java and .NET in these circumstances.

- In .NET, the request message is backed out by setting ContextUtil.MyTransactionVote=TransactionVote.Abort. This results in the request message being left on the request queue and being reprocessed by the listener until the backout retry threshold (3) is exceeded. A report message is then written to the response queue.
- In Java, the service calls System.Exit(). This is not recommended practice for production applications but is done to demonstrate the effect of a catastrophic failure in the service. In this case the listener exits and the message is backed out so it is left on the request queue. Separate invocations of the listener repeat this behaviour, with the listener exiting and the request message being backed out until the backout/retry count is exceeded. At this point a report message is sent to the response queue.

Examples - Java

The sample commands that follow are formatted for clarity, enter the commands without line breaks.

Example of running the request and response separately

To make the request, invoke the sample program using the following command:

```
java soap.clients.SQAxis2AxisAsyncReqRespTran request
    "jms:/queue?destination=SOAPJ.trandemos@WMQSOAP.DEMO.QM&connectionFactory=
    connectQueueManager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
    &targetService=StockQuoteDotNetTran.asmx
    &initialContextFactory=com.ibm.mq.jms.Nojndi"
    testJavaClient
    commit
    defaultmsgintegrity
```

The program responds with

```
Calling commit
Async service(s) successfully requested.
```

To receive the response asynchronously, invoke the sample program using the following command:

```
java soap.clients.SQAxis2AxisAsyncReqRespTran response
    "jms:/queue?destination=SOAPJ.trandemos@WMQSOAP.DEMO.QM&connectionFactory=
    connectQueueManager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
    &targetService=StockQuoteDotNetTran.asmx
    &initialContextFactory=com.ibm.mq.jms.Nojndi"
    testJavaClient
    commit
    defaultmsgintegrity
```

The program responds with

```
ASYNC RPC reply is: 55.25
All Async response(s) successfully received.
```

Example of running the request and response together

To make the request and receive the response within the same process, invoke the sample program using the following command:

```
java soap.clients.SQAxis2AxisAsyncReqRespTran requestresponse
    "jms:/queue?destination=SOAPJ.trandemos@WMQSOAP.DEMO.QM&connectionFactory=
    connectQueueManager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
    &targetService=StockQuoteDotNetTran.asmx
    &initialContextFactory=com.ibm.mq.jms.Nojndi"
    testJavaClient
    commit
    defaultmsgintegrity
```

The program responds with:

```
Calling commit
Async service(s) successfully requested.
ASYNC RPC reply is: 55.25
All Async response(s) successfully received.
```

Examples - .NET

The sample commands that follow are formatted for clarity, enter the commands without line breaks.

Example of running the request and response separately

To make the request, invoke the sample program using the following command:

```
SQCS2DotNetAsyncReqRespTran request
    "jms:/queue?destination=SOAPN.trandemos@WMQSOAP.DEMO.QM&connectionFactory=
    connectQueueManager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
    &targetService=StockQuoteDotNetTran.asmx
    &initialContextFactory=com.ibm.mq.jms.Nojndi"
    testNetClient
    commit
    defaultmsgintegrity
```

The program responds with

```
Completing transaction
Async service(s) successfully requested.
```

To receive the response asynchronously, invoke the sample program using the following command:

```
SQCS2DotNetAsyncReqRespTran response
    "jms:/queue?destination=SOAPN.trandemos@WMQSOAP.DEMO.QM&connectionFactory=
    connectQueueManager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
    &targetService=StockQuoteDotNetTran.asmx
    &initialContextFactory=com.ibm.mq.jms.Nojndi"
    testNetClient
    commit
    defaultmsgintegrity
```

The program responds with

```
  CallBackFunction: intran=False
  CallBackFunction: intran=True
ASYNC response is: 88.88
CallBack function is running transactionally
All Async response(s) successfully received.
```

Example of running the request and response together

To make the request and receive the response within the same process, invoke the sample program using the following command:

```
SQCS2DotNetAsyncReqRespTran requestresponse
    "jms:/queue?destination=SOAPN.trandemos@WMQSOAP.DEMO.QM&connectionFactory=
    connectQueueManager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
    &targetService=StockQuoteDotNetTran.asmx
    &initialContextFactory=com.ibm.mq.jms.Nojndi"
    testNetClient
    commit
    defaultmsgintegrity
```

The program responds with:

```
Completing transaction
Async service(s) successfully requested.
  CallBackFunction: intran=False
  CallBackFunction: intran=True
ASYNC response is: 88.88
CallBack function is running transactionally
All Async response(s) successfully received.
```

Chapter 12. Asynchronous messaging

What is asynchronous messaging?

As used in WebSphere MQ, the term ″asynchronous messaging″ means a method of communication between programs in which a program places a message on a message queue, then proceeds with its own processing without waiting for a reply to its message. This contrasts with synchronous messaging, in which a program places a message on a message queue and then waits for a reply to its message before resuming its own processing.

In the context of WebSphere MQ transport for SOAP, long term asynchrony refers to the ability of a client to invoke a service request from one process and then retrieve the response from the service in a different process. Such a facility might be required where it is not practical or efficient to prolong the lifetime of a client until the response is received. Some services might not be designed to return a response immediately but might return a set of responses at some future point. Other services might be able to process the request immediately but might then be unable to return the response owing to unstable network connections. For both these situations, the ability to decouple the processing of the client's request and response across separate processes is a valuable option.

In the Microsoft .NET Environment, short term asynchrony means asynchronous operation within the context of a single client process.

Short-term asynchronous messaging

Short-term asynchronous messaging is available in the .NET environment using Microsoft's proprietary .NET asynchronous interface. The request must be submitted by and the response returned to the same process. For details, consult the relevant Microsoft documentation. Short-term asynchrony is not available in the Axis environment.

.NET short-term asynchronous messaging is supported in WebSphere MQ transport for SOAP but no samples are provided.

Long-term asynchronous messaging using the MA0V SupportPac

This section contains details of the facilities provided by the WebSphere MQ SupportPac MA0V ″Asynchronous support in the WebSphere MQ transport for SOAP″ to support asynchronous messaging.

The use of long term asynchrony affects the invoking client application and the transport sender code only. Implications for the application are summarized in the following sections. The SOAP/WebSphere MQ listener and the Web service itself do not require any specific options or customization in order to work with an asynchronous client application.

What is installed

Additional software is installed by the SupportPac to implement asynchronous messaging support. This is principally in the files com.ibm.mq.soapasync_MA0V.jar (for Java) and amqsoapasync_MA0V.dll (for .NET).

Using a long-term asynchronous client

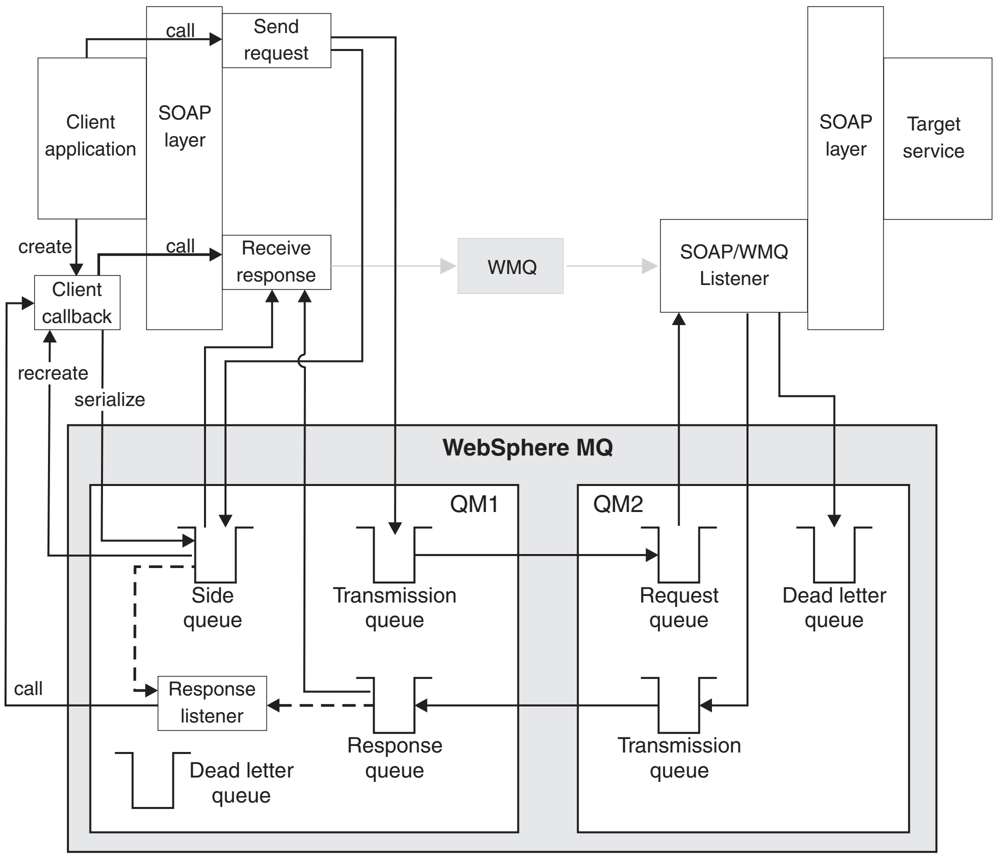

Figure 5. Overview of asynchronous client operation

An asynchronous client application works as follows:

1. The client starts an asynchronous response listener on the response queue.
2. The client makes an asynchronous request, specifying a callback object (see "Callback objects" on page 78 for an explanation of callback objects).
3. The callback object is serialized to the side queue.
4. When a response arrives, the asynchronous response listener retrieves and reconstitutes the callback object.
5. The callback object processes the response
6. The client stops the client asynchronous response listener

Note: Steps 1 and 2 can be performed in either order.

The request and response can be handled in separate processes.

Starting an asynchronous response listener

The client starts an asynchronous response listener by instantiating a ResponseListener object. The constructor takes a URI as an argument that specifies where the response listener should look for the response message. The ResponseListener object has the following alternative constructors:

- Java

```
ResponseListener(String aURL) throws WMQSOAPException
ResponseListener(String aURL, Object aClientId) throws
WMQSOAPException
```

```
ResponseListener(String aURL, Object aClientId,
IResponseListenerNotification errorHandler) throws WMQSOAPException
```

```
ResponseListener(String aURL, Object aClientId,
IResponseListenerNotification errorHandler, int aMsgIntegrity) throws
WMQSOAPException
```

- C#

```
ResponseListener(String aURL)
```

```
ResponseListener(String aURL, object aClientId)
```

```
ResponseListener(String aURL, object aClientId,
IResponseListenerNotification notifier)
```

```
ResponseListener(String aURL, object aClientId,
IResponseListenerNotification notifier, int msgIntegrity)
```

ClientId must be a non-negative integer and defaults to 0. (See "Distinguishing between responses for different requests" on page 75 for an explanation of client Ids)

errorHandler defaults to null (no notification of errors will occur).

msgIntegrity defaults to DEFAULT_MSG_INTEGRITY

The following C# code segment illustrates invocation of the response listener:

```
 // Create proxy object so we know URL for the response listener
  StockQuote stockobj = new StockQuote();

  // Now create (and start) the response listener
  m_responseListener = new WMQSOAP.ResponseListener(stockobj.Url);
```

Figure 6. Sample code: starting an asynchronous response listener

This call causes the response listener to be created and to start within a separate execution thread.

Making an asynchronous request

A client makes an asynchronous request as follows:

1. The client indicates to the sender code that it intends to operate in asynchronous mode. To do this it instantiates an object derived from the class WMQSOAP.AsyncCallBack. This object implements the callback function which will be invoked by the response listener.
2. The client calls the method WMQSOAP.Async.Request(), passing the callback object. The boolean isTransacted flag specifies whether the sender and response listener perform syncpointed operations.
3. The client invokes the service as in the synchronous case (see "Basic Web service client programming" on page 25).
4. The transport sender code serializes the callback object and puts it on a dedicated queue referred to as the side queue.
5. The transport sender code returns a WMQSOAP.AsyncResponseExpectedException exception to the client, which indicates that the sender code has correctly interpreted the call as an asynchronous request.

Asynchronous messaging

The following C# code segment illustrates the asynchronous registration and call procedure:

```
    // Create the proxy to make the call
    StockQuoteDotNet stockobj = new StockQuoteDotNet();

    // Create a context object, and remember some context in it
    testStateClass requestContext = new testStateClass();
    requestContext.myCall = "XXX";

    // Make the request part of the call, ignore the exception
    String myClient="client";
    try  {
        WMQSOAP.Async.Request(requestContext, myClient);
        res = stockobj.getQuote("XXX");
    } catch (WMQSOAP.AsyncResponseExpectedException) {}
```

Figure 7. Sample code: making an asynchronous request

Processing the response

The asynchronous response listener monitors the response queue for the service. When a reply is received on the response queue, it is processed as follows:

1. The listener retrieves the corresponding message from the side queue.
2. The listener reconstitutes the callback object and calls its CallBackFunction() method so that this can retrieve the response message.
3. The callback object registers that it wishes to receive an asynchronous response by calling the method Async.Response().
4. The callback object makes a dummy invocation of the service to extract the response.
5. The transport code detects it is being requested to retrieve a waiting asynchronous response and retrieves the response message without reinvoking the service.

The following is an example of a C# CallBack class derived in a client. This implements the CallBackFunction() and demonstrates the above process.

```
[Serializable] class testStateClass : WMQSOAP.Async.CallBack
{
    public string myCall = null;  // used for context
    public override void CallBackFunction() {  // used for callback
        // get proxy
        StockQuoteDotNet stockobj = new StockQuoteDotNet();

        // make call to get response
        WMQSOAP.Async.Response(this);
        Single res = stockobj.getQuote("unused dummy parameter");
        Console.WriteLine("ASYNC response to " + myCall + " is: " + res);
    }
}
```

Figure 8. Sample code: a CallBack class

Transactions in asynchronous messaging are managed by the client code. The client must use the Java calls begin(), commit() and abort() or the .NET System.EnterpriseServices.ServicedComponent attributes or calls.

If a callback object is not found: If a response listener cannot find a message on the side queue for a particular response message, its behavior depends on the value of the msgIntegrity argument used in the MQResponseListener start method. This can be set to one of the following values:

For Java:

- 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

For .NET.:

- WMQSOAP.ErrorHandler.DEFAULT_MSG_INTEGRITY
- WMQSOAP.ErrorHandler.LOW_MSG_INTEGRITY
- WMQSOAP.ErrorHandler.HIGH_MSG_INTEGRITY

These have the same meanings as the DefaultMsgIntegrity, LowMsgIntegrity and HighMsgIntegrity integrity options used when starting a normal listener. See "Listeners" on page 43 for explanations of these values and the actions the listener will take. Note that if you set isTransacted to false, you must use LOW_MSG_INTEGRITY; the response listener will exit with an error message if this is not set.

Stopping the asynchronous response listener

When the client has received all the responses it needs to, it should stop the response listener with a call to its Stop() method. Make this call on the instance of the ResponseListener object, for example `m_responseListener.Stop();`. Any callback that is being processed when the call is made will run to completion before the listener terminates.

Distinguishing between responses for different requests

To distinguish between the responses to different requests for the same service and between responses to synchronous and asynchronous requests, you can use client IDs. If you specify a client ID in the registration call for the client, that client ID is used for all requests issued by that client. The SOAP/WebSphere MQ listeners copy the client ID into the response messages. The response listener can then be started by the client so as to retrieve only messages that match that client ID. The use of the client ID is illustrated in the following code fragment:

```
// prepare for request call
WMQSOAP.Async.Request(requestContext, myClientID);

// start listener call
m_responseListener = new WMQSOAP.ResponseListener(stockobj.Url, myClientID);
```

You must ensure that the client Id is unique within the side queue for your application. The amqwAsyncConfig utility can be used to list all the client Ids used by a queue manager, for example:

```
amqwAsyncConfig -qm WMQSOAP.DEMO.QM
```

Housekeeping

The use of client Ids involves the creation of permanent dynamic queues. It is most efficient to retain these queues while the client Id is still in use but when they are no longer needed you should remove them using the AsyncConfig utility, described in this section.

The utility exists as a Java class, a .NET class and as command scripts, which call the Java class. The Java and .NET classes are similar, as described below.

For Java:

```
static public Hashtable AsyncConfig.enumClients
                (MQQueueManager qm,
                String clientIdFilter,
                String baseQNameFilter,
                String dynQNameFilter,
                String envFilter,
                java.io.PrintStream printer,
                boolean toDelete)  throws WMQSOAPException
```

For .NET:

```
static public Hashtable AsyncConfig.EnumClients
                (MQQueueManager qm,
                string clientIdFilter,
                string baseQNameFilter,
                string dynQNameFilter,
                string envFilter,
                StreamWriter printer,
                bool toDelete)
```

where the properties have the following meanings

qm
queue manager in use

clientIdFilter
filter for clientId

baseQNameFilter
filter for base queue name

dynQNameFilter
filter for the dynamic queue name

envFilter
filter for environment [″java″ or ″dotnet″]

printer
PrintStream or StreamWriter object for list output: no output if null

toDelete
if true, matching entries will be deleted (both the side queue callback object and the dynamic queue); if false, they will not.

All the filters use Java regular expression syntax; use a null string to match any entry.

The return value is a Hashtable of all matching clients. The key is object[], which contains longName, baseName and environment; the value is the corresponding object[], containing dynamic queue name and exception. The exception is generally null, but if an exception is raised on a particular entry, it will be saved in the Hashtable. The utility will not throw an exception for an error in an individual entry but it will throw an exception for a problem with the WebSphere MQ infrastructure, for example if the side queue cannot be read.

Command script: The AsyncConfig utility is also provided as a command script named amqwAsyncConfig.sh in UNIX and amqwAsyncConfig.cmd in Windows. The syntax is:

```
amqwAsyncConfig [-qm] [-clientId] [-baseQ] [-dynQ] [-java] [-dotnet] [-list] [-delete]
```

where the properties have the following meanings

-qm
queue manager name to use (defaults to default queue manager)

-clientId
filter for clientId (defaults to no filtering)

-baseQ
filter for base queue name (defaults to no filtering)

-dynQ
filter for the dynamic queue name (defaults to no filtering)

-java
show dynamic queues for support of Java

-dotnet
show dynamic queues for support of .Net

-list
list output to terminal

-delete
if set, matching entries will be deleted (both the side queue entry and the dynamic queue)

If neither -java nor -dotnet is specified, both are included.

If neither -list nor -delete is specified, -list is assumed.

If -delete is specified, both -clientId and -baseQ must be specified.

The filters -clientId and -baseQ use Java regular expression syntax. If the command is called from a shell, use the appropriate escape mechanism to pass wildcards to the utility. For example from a Windows command line, you would use quotation marks: `amqwAsyncConfig -baseQ ".*"`

For example:
- `amqwAsyncConfig` lists full information for the default queue manager.
- `amqwAsyncConfig -clientId Client1 -baseQ .* -delete` deletes all information about clientId Client1, for all base queues.
- `amqwAsyncConfig -clientId .* -baseQ .* -delete` deletes all information about all clientIds.
- `amqwAsyncConfig -dynQ TESTBASE_Jfred4210A7F5066A0220` shows the usage of the specified dynamic queue.
- `amqwAsyncConfig -dynQ TESTBASE_J.*` shows the usage of dynamic queues with names beginning TESTBASE_J

Sample output is shown in Figure 9.

```
clientId(fred) baseQ(TESTBASE)  dynQ(TESTBASE_Jfred4210A7F5067F0220) env(java)
clientId(fred) baseQ(TESTBASE)  dynQ(TESTBASE_J4210A7F5087F0220) env(java)
clientId(fredfredfredfredfredfredfredfredfredfredfredfred) baseQ(TESTBASE)
    dynQ(TESTBASE_Jfredfredfredfredfredfr4210A7F50A7F0220) env(java)
```

Figure 9. Sample output from amqwAsyncConfig

Callback objects

An asynchronous request call includes a serializable 'callback' object. This object is preserved as a byte string and is reconstituted when the response is processed, even from a separate process. This serialized context is stored as a context message in the side queue, SYSTEM.SOAP.SIDE.QUEUE with its correlation Id set to the message Id of the request message. It is picked up and returned to the application when the asynchronous response is processed. The callback object provides two features; an entry point to call and context information identifying which request this is a response to.

You can create and configure the side queue using the UNIX shell script setupSOAPWMQ.sh or the Windows command file setupSOAPWMQ.cmd. This script is automatically called from the regenDemo utility. If you want to change the characteristics of the side queue, for example to change the MaxQueueDepth from its default value of 1000, you can edit the appropriate script, or alter the queue.

Purging redundant context messages

Under certain error conditions, such as when a response can never be returned for a failed request, context messages can be left in the side queue and these need to be periodically removed. For this purpose, a Java utility is provided to remove context messages in a given side queue that are older than a specified age. A Windows command file amqwCleanSideQueue.cmd and a UNIX shell script amqwCleanSideQueue.sh are provided to make the use of the utility easier. The calling syntax of these scripts is:

```
amqwCleanSideQueue [-d days] [-h hours] [-mn mins] [-m queue-manager]
[-q side-queue] [-?]
```

-d *days*
: Specifies an age in days; messages older than this age will be removed. Defaults to 1.

-h *hours*
: Specifies an age in hours; messages older than this age will be removed.

-mn *mins*
: Specifies an age in minutes; messages older than this age will be removed.

-m *queue-manager*
: Specifies the queue manager that contains the side queue.

-q *side-queue*
: Specifies the name of the side queue. Defaults to SYSTEM.SOAP.SIDE.QUEUE.

-?
: Displays help information describing how the command should be used.

Only one of the -d, -h and -mn options is permitted. If none are specified, the default is `-d 1`, one day.

The utility prompts for confirmation before deleting any messages, displaying the target queue manager, queue name and age. It displays error messages for any messages that it cannot delete. It also displays the total number of messages that were successfully deleted before it exits.

Linking

For Java, the com.ibm.mq.soapasync_MA0V.jar filename is always included in the standard classpath which is used when SOAP/WebSphere MQ is run as part of WebSphere MQ. No special action is needed to make the asynchronous facilities available other than installing MA0V.

For .NET, the clients must be explicitly linked with amqsoapasync_MA0V.dll and amqsoap.dll if asynchronous support is to be used. If only synchronous support is required, they must be linked only with amqsoap.dll.

Errors

If you try to build asynchronous clients when the SupportPac is not present, the code will not compile. The compiler will issue a ″Class not found″ error.

If you build an asynchronous client with the SupportPac present, but then run the resulting object code in an environment that supports only synchronous code, you will receive a ″Class not found″ error.

Sample programs

Some asynchronous sample programs are provided as part of the IVT. See "Samples" on page 22 for details of these.

Sample programs SQAxis2AxisAsyncReqRespNtfy.java and SQCS2DotNetAsyncReqRespNtfy.cs are also provided. These samples use an additional class, DemoAsyncNotifier, this is embedded in SQCS2DotNetAsyncReqRespNtfy.cs for .NET and is a separate class for Java. This class implements the IResponseListenerNotification interface and provides implementations of the onInitialize() and onTerminate() methods. A DemoAsyncNotifier object is passed into the response listener which uses it to call your specific initialize and terminate methods. This sample demonstrates how you can take appropriate action when the asynchronous response listener starts or when it terminates.

These samples are run as follows.

Java sample

First start the SimpleJavaListener as detailed in "Java SOAP/WebSphere MQ listener" on page 44. Then invoke the program twice, first with the **request** parameter:

```
C:\temp>java soap.clients.SQAxis2AxisAsyncReqRespNtfy request myClient
```

The program will reply with:

```
Async service(s) successfully requested.
```

Invoke the program a second time, with the **response** parameter:

```
C:\temp>java soap.clients.SQAxis2AxisAsyncReqRespNtfy response myClient
```

The program will reply with :

```
The DemoAsyncNotifier onInitialize method has been called.
ASYNC RPC reply is: 55.25
All Async response(s) successfully received.
The DemoAsyncNotifier onTerminate method has been called.
```

.NET sample

First start the amqwSOAPNetListener as detailed in ".NET SOAP/WebSphere MQ listener" on page 46. Then invoke the program twice, first with the **request** parameter:

```
C:\temp>SQCS2DotNetAsyncReqRespNtfy request myClient
```

The program will reply with:

```
Async service(s) successfully requested.
```

Invoke the program a second time, with the **response** parameter:

```
C:\temp>SQCS2DotNetAsyncReqRespNtfy response myClient
```

The program will reply with :

```
The DemoAsyncNotifier onInitialize method has been called.
ASYNC response is: 88.88
All Async response(s) successfully received.
The DemoAsyncNotifier onTerminate method has been called.
```

Note the messages confirming that onInitialize() and onTerminate() have been called.

Part 4. Appendixes

Appendix A. Apache software license

Appendix B. Notices

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	CICS	IBM
IBMLink™	POWER	SupportPac
WebSphere	zSeries	

Java and all Java based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Microsoft and Windows are trademarks or registered 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.

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

Index

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–816151
  - From within the U.K., use 01962–816151
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in USA

SC34-6656-00

Spine information:

IBM WebSphere MQ Transport for SOAP with Asynchronous Extensions