WebSphere Message Broker

End user Application Support

Version 6 Release 0

WebSphere Message Broker

End user Application Support

Version 6 Release 0

Note

Before using this information and the product it supports, read the information in the Notices appendix.

Fourth Edition (July 2006)

This edition applies to IBM® WebSphere® Message Broker Version 6.0 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this topic collection. v

Part 1. End user Application Support 1

Supporting end-user applications. 3
End-user application support 3
Enabling WebSphere MQ applications 20
Enabling WebSphere MQ Everyplace applications . 23
Enabling WebSphere MQ Telemetry Transport applications 24
Working with Web service applications 27

Part 2. Reference 53

End-user application support 55
Application transports. 55
Application environments 109

Part 3. Appendixes 113

Appendix. Notices 115
Trademarks 117

Index 119

About this topic collection

This PDF has been created from the WebSphere Message Broker Version 6.0 (Fix Pack 2 update, July 2006) information center topics. Always refer to the WebSphere Message Broker online information center to access the most current information. The information center is periodically updated on the document update site and this PDF and others that you can download from that Web site might not contain the most current information.

The topic content included in the PDF does not include the ″Related Links″ sections provided in the online topics. Links within the topic content itself are included, but are active only if they link to another topic in the same PDF collection. Links to topics outside this topic collection are also shown, but these attempt to link to a PDF that is called after the topic identifier (for example, ac12340_.pdf) and therefore fail. Use the online information to navigate freely between topics.

Feedback: do not provide feedback on this PDF. Refer to the online information to ensure that you have access to the most current information, and use the Feedback link that appears at the end of each topic to report any errors or suggestions for improvement. Using the Feedback link provides precise information about the location of your comment.

The content of these topics is created for viewing online; you might find that the formatting and presentation of some figures, tables, examples, and so on are not optimized for the printed page. Text highlighting might also have a different appearance.

Part 1. End user Application Support

Supporting end-user applications 3
End-user application support 3
Application communication models. 4
Application programming interfaces 7
Application transports 9
Application environments 19
Enabling WebSphere MQ applications 20
Defining WebSphere MQ resources 20
Securing WebSphere MQ resources 21
Building the message model for the IDoc parser 22
Enabling WebSphere MQ Everyplace applications . 23
Enabling WebSphere MQ Telemetry Transport applications 24
Designing Telemetry applications 24
An example message flow to support Telemetry clients 25
Working with Web service applications 27
Web services, WSDL and message flows 28
Working with HTTP flows 33
Web services scenarios. 36

Supporting end-user applications

Your end-user applications can connect to the broker using one of the following transports:

- WebSphere MQ clients connect using the WebSphere MQ Enterprise Transport. SAP clients also use this transport.
- WebSphere MQ Everyplace™ clients connect using the WebSphere MQ Mobile Transport.
- Multicast JMS clients connect using the WebSphere MQ Multicast Transport.
- Real-time JMS clients connect using the WebSphere MQ Real-time Transport.
- SCADA clients connect using the WebSphere MQ Telemetry Transport.
- Web services clients connect using the WebSphere MQ Web Services Transport.
- JMS clients connect using the WebSphere Broker JMS Transport.
- Additional clients can connect using alternative transports if you have installed user-defined nodes and parsers that support them.

The topics in this section provide further information for particular clients including special tasks that you must take to enable end-user applications and WebSphere Message Broker brokers to communicate, and examples of supported scenarios and configurations.

- “End-user application support”
- “Enabling WebSphere MQ applications” on page 20
- “Enabling WebSphere MQ Everyplace applications” on page 23
- “Enabling WebSphere MQ Telemetry Transport applications” on page 24
- “Working with Web service applications” on page 27

End-user application support

You can connect a variety of end-user applications to the WebSphere Message Broker brokers, and take advantage of the routing, aggregation, and transformation facilities that it provides.

WebSphere Message Broker supports two application communication models:

1. Point-to-point
2. Publish/subscribe

These are defined in “Application communication models” on page 4.

Applications that use these models can connect to the broker using the following transports and protocols:

- “WebSphere MQ Enterprise Transport” on page 9
- “WebSphere MQ Mobile Transport” on page 10
- “WebSphere MQ Multicast Transport” on page 10
- “WebSphere MQ Real-time Transport” on page 11
- “WebSphere MQ Telemetry Transport” on page 11
- “WebSphere MQ Web Services Transport” on page 12
- “WebSphere Broker JMS Transport” on page 15

You can configure message flows to support these communication models and clients connecting over any one of these transports. Your message flows can be specific to one protocol, or can receive messages from applications communicating across one protocol and deliver messages to applications connecting across any one or more alternative protocols, with the broker providing automatic conversion between these protocols. You can also provide point-to-point and publish/subscribe support in a single message flow.

All message flows can support messages crossing from all transports to all other supported transports, Therefore if you start the message flow with an input node that supports messages from clients that connect through one transport, you can end it with any of the supported output nodes (including user-defined output nodes); you do not have to include the complementary output node.

For example, you can design the flow to receive WebSphere MQ messages and generate output messages to SCADA devices, or to receive messages from SCADA devices and generate output messages for real-time or multicast application clients. However, certain restrictions do apply depending on the transport being used: for example, messages published persistently through an MQInput node are not guaranteed to be delivered to the subscribers over WebSphere MQ Real-time Transport, because this transport does not support assured delivery.

For example, you can create a message flow that receives a message from a WebSphere MQ application. The message flow constructs a publication message from its contents, and publishes the message through a Publication node from where real-time subscribers that use JMS can register their interest in the publications. The message flow can also include an MQReply or MQOutput node to provide a confirmation message that indicates to another application that the publication has been implemented.

Application communication models

Applications can use the services of a broker by sending messages to it and receiving messages from it, across one of the supported transport protocols.

The way that they do this depends on the protocol itself, the programming interface that they use, and the communication model that they adopt.

WebSphere Message Broker supports two end-user application communication models:
1. "Point-to-point"
2. "Publish/subscribe" on page 6

A single application can mix the two styles, if appropriate. In a mixed scenario, the message flow that processes the messages for this application contains at least one output node and at least one publication node, in addition to one or more input nodes.

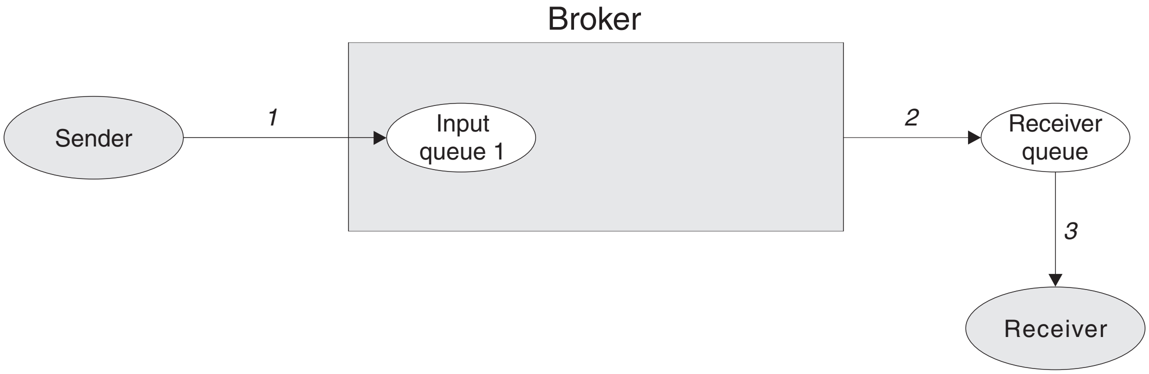
The programming interfaces that you can code in your applications are described in "Application programming interfaces" on page 7.

Point-to-point

Point-to-point applications use a request/reply or client/server model, or broadcast a message to many target applications using distribution lists. Other applications send one-way send-and-forget or datagram traffic. They exchange information with known partners. Each application is aware of the identity of the one or more applications with which it is communicating. You can create and configure message flows to process both send-and-forget and request/reply messages, and deploy them to your brokers.

The text and diagrams below illustrate the send-and-forget and request/reply models. The diagrams assume that the applications are using the WebSphere MQ Enterprise Transport protocol. The model is identical for other protocols, although the resource through which a message is sent or received will not be a WebSphere MQ queue.

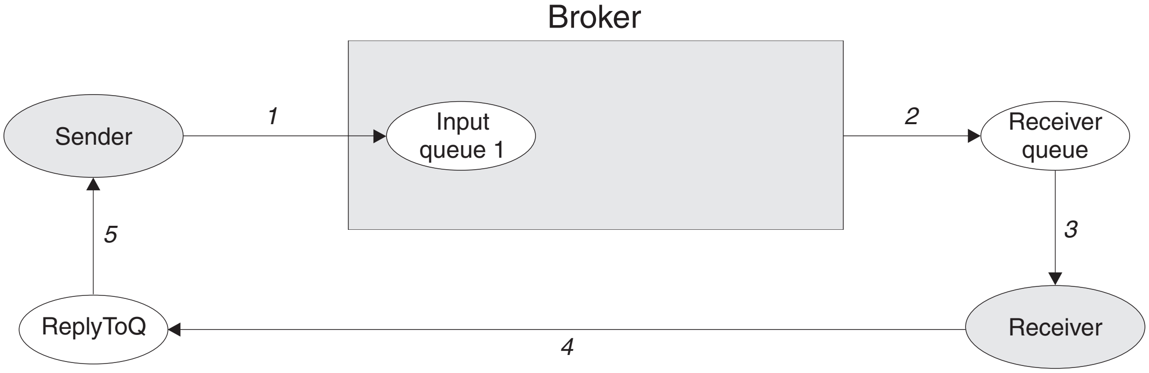
In the send-and-forget model, an application sends a message but does not expect a reply. Another application might optionally receive a message as a result of the message sent by the first application. It is possible that no message is sent by the message flow (for example, if the sending message just requested a database update). In the diagram, the sender puts a message onto the input queue of a message flow at the broker (1). The output from the message flow is put onto the receiver's queue (2), from where the receiver can get it (3).



With request/reply messaging, after the receiver receives a request message it sends a reply back to the sender. The request message is handled as described for send-and-forget messages. There are two possibilities for the reply:

1. The receiver sends the reply message directly back to the sender, without involving the broker. The message is sent to the *ReplyToQ* in the message descriptor (MQMD) of the request message, which is passed unchanged by the broker. (If your applications are not using WebSphere MQ, you must use some other technique to determine the reply destination.)

   In the diagram below, the sender puts a message onto the input queue of a message flow at the broker (1). The output from the message flow is put onto the receiver's queue (2), from where the receiver gets it (3). The receiver sends the reply directly to the ReplyToQ of the sender (4), from where the sender can get it (5).



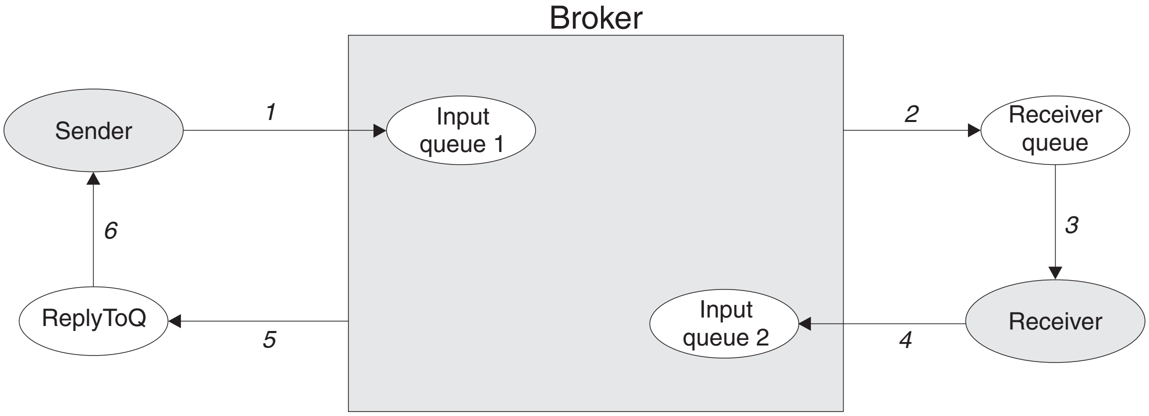
2. The receiver sends the reply message to a reply message flow in the broker, so that it can be processed before reaching the sender. In this case, the broker must replace the sender's *ReplyToQ* in the MQMD of the request message with the input queue name of the reply message flow.

   The output of this reply message flow must go to the sender's *ReplyToQ*. If the name is fixed, there is no problem; if it is not, some means of associating this queue with the reply message is needed.

   You can do this, for example, by including a Database or DataInsert node in the first message flow that stores the reply destination information, which can be retrieved by the second message flow.

   Alternatively, the relevant details in the message descriptor can be copied into a folder in the MQRFH2 header, and carried with the message.

   In the diagram below, the sender puts a message onto the input queue of the first message flow at the broker (1). The output from the message flow is put onto the receiver's queue (2), from where the receiver gets it (3). The receiver sends the reply to the input queue of the second message flow at the broker (4). After processing the reply, the broker sends it to the ReplyToQ of the sender (5), from where the sender can get it (6). (In this case, the output node of the second message flow needs to know the ReplyToQ of the sender.)



Existing applications that you have written using the point-to-point model can run unchanged in a WebSphere Message Broker environment if they use one of the supported protocols to communicate with the broker.

You can enhance and extend your existing application function by using the facilities of the broker to include additional partners. For example, an application that handles similar data but in a different format can participate because the original message can be transformed by a message flow in the broker into the expected format, without the need to change the sending or receiving application.

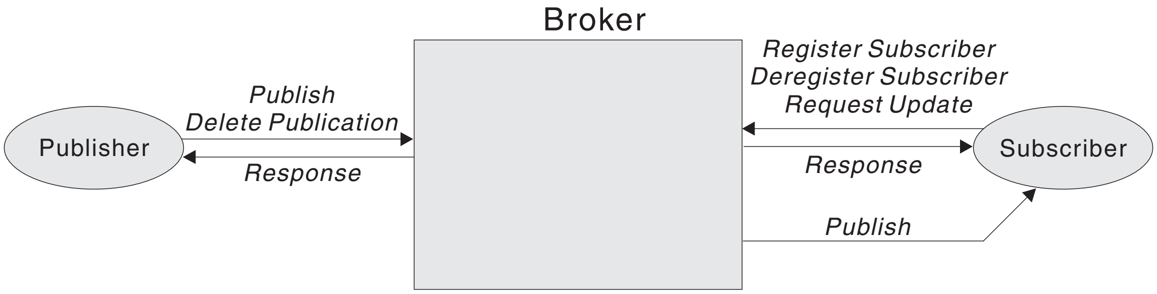
If you identify a message that needs additional application processing, you can create another copy of the message in the message flow, and send it to a new application developed to provide that processing. The original applications are unaware of the new action on the message and continue to work unchanged.

Publish/subscribe

The publish/subscribe application communication model involves applications known as publishers and applications known as subscribers. Publishers make messages available by publishing on specific topics. Subscribers receive messages by subscribing to topics. An individual application can be both a publisher and a subscriber.

Messages published by any one publisher can be received by any number of subscribers. Subscribers might also receive messages, on the same or different topics, from any number of publishers.

In the diagram below, the publisher can send Publish or Delete Publication messages to the broker. The broker forwards the Publish message to subscribers that have a matching subscription. The subscriber can send Register Subscriber, Deregister Subscriber, or Request Update messages to the broker. Optional Response messages from the broker are sent to the publisher and subscriber.



If you have existing end-user applications that are written to the publish/subscribe model, for example using the MQI or AMI, you can probably integrate these applications into a WebSphere Message Broker broker domain without change.

You can also modify these applications, or write new ones, to take advantage of the sophisticated publish/subscribe processing that is provided, particularly for subscribers.

The publish/subscribe model, and the processing provided by WebSphere Message Broker, is described fully in further topics available through the related links listed below.

Application programming interfaces

WebSphere Message Broker does not provide any unique programming interfaces, but supports several programming interfaces that are already in use by messaging applications today.

- Message Queue Interface (MQI)

  The MQI provides a small number of calls that allow an application to interact with other applications across a WebSphere MQ network of queue managers. The calls support a large range of parameters that allow a rich choice of processing options for each and every message.

  Client applications using the MQI can run on any supported WebSphere MQ operating system, and therefore any limitations as to language or function are defined by the relevant product for that operating system.

  The MQI is described in the *WebSphere MQ Application Programming Guide* and *WebSphere MQ Application Programming Reference*. These books also give details of the programming language and operating system support available for clients that use this interface.
- Application Messaging Interface (AMI)

The AMI is designed to simplify the application programmer’s task by centralizing the selection of optional parameters outside the application program. It also provides support for the more advanced functions available from the message broker. The AMI is designed for general messaging applications with and without a broker.

The principal functions of the AMI are administrator-defined packets of options known as policies and services. An application specifies a service to determine the underlying messaging support required, and associates a policy with sending or receiving a message to control attributes for message processing, such as priority.

The policies and services mean that the application does not have to understand details of the MQRFH2 header and the MQI interface.

Client applications using the AMI are restricted to the operating systems and programming languages supported by this interface. Check the current level of the *WebSphere MQ Application Messaging Interface* book for details, or visit the WebSphere MQ web site.

- Java Message Service (JMS)

  The JMS is an application programming interface that provides Java language functions for handling messages. Developed by messaging vendors, including IBM in partnership with Sun Microsystems, Inc., the JMS API provides a common interface to access different enterprise messaging systems, including WebSphere MQ. This interface is appropriate for point-to-point and publish/subscribe applications.

  Messaging clients in JMS are called *JMS clients*, and the messaging system is called the *JMS provider*. A *JMS application* is a business system that comprises JMS clients and at least one JMS provider. Client applications using the JMS interface are written in the Java programming language, and are therefore restricted to the levels of JVM that are supported on the operating system in question. For further information, see the *WebSphere MQ Using Java* book, or visit the WebSphere MQ web site.
- WebSphere MQ Everyplace programming calls

  These calls are described in “WebSphere MQ Mobile Transport” on page 56.
- SCADA device protocol publication messages.

  These messages are described in “WebSphere MQ Telemetry Transport” on page 58.

If you have existing end-user applications that are written to these interfaces, they can typically run unchanged in a broker environment. You must create the message flows to interact with these applications across the supported protocols, using the appropriate input and output nodes. WebSphere Message Broker provides built-in input and output nodes for its supported protocols and you can create your own user-defined nodes to support additional protocols if you choose.

You can also create new end-user applications to interact with the broker.

Message headers

WebSphere Message Broker provides parsers for a large number of WebSphere MQ headers, and can therefore accept messages that contain these headers across the WebSphere MQ Enterprise Transport, WebSphere MQ Mobile Transport, and WebSphere MQ Telemetry Transport protocols.

Messages must include a WebSphere MQ Message Descriptor (MQMD) as the first header, which must precede user or application data in every message. The MQMD contains basic control information that must travel with the message, including:
- The message identifier
- The destination of the reply, if one is to be sent
- Reply and report options (for example, confirm on delivery report)
- The format of any following data in the message

When a message is processed by a WebSphere Message Broker broker, it typically (but not necessarily) has one or more additional headers. The header following the MQMD is always identified in the format field within the MQMD, and itself contains another format field to identify either the header that follows, or the format of the user data.

The additional headers can include:

MQRFH
The Rules and Formatting header is used by WebSphere MQ Publish/Subscribe.

MQRFH2
The MQRFH2 is an updated version of MQRFH and allows Unicode strings to be transported without translation, and it can carry numeric data types. The MQRFH2 header carries a description of the message contents, so that WebSphere Message Broker can select the correct message parser when content-based processing is carried out on the message. In addition, this header contains publish/subscribe command messages. Messages created by the SCADAInput node always include and MQRFH2 header.

Use the MQRFH2 header in all new applications written for the WebSphere Message Broker environment that use a supported protocol based on WebSphere MQ technology. The MQRFH2 header should be immediately before the body of the message (that is, the last header).

If an MQRFH2 header is not included (which is normally the case of the application uses a supported protocol that is not based on WebSphere MQ technology), you must configure the message flow that processes its messages to specify the message characteristics (by setting the input node properties).

Application transports

This section covers the following topics:
- “WebSphere MQ Enterprise Transport”
- “WebSphere MQ Mobile Transport” on page 10
- “WebSphere MQ Multicast Transport” on page 10
- “WebSphere MQ Real-time Transport” on page 11
- “WebSphere MQ Telemetry Transport” on page 11
- “WebSphere MQ Web Services Transport” on page 12
- “WebSphere Broker JMS Transport” on page 15

WebSphere MQ Enterprise Transport

WebSphere MQ Enterprise Transport is a service that connects applications to messaging middleware.

This transport is used by WebSphere MQ clients or application programs that are written to the Application Messaging Interface (AMI) or Message Queue Interface

(MQI). The client uses the services provided by the message flows deployed within one or more brokers in the broker domain by interacting with the queues serviced by those message flows.

Application clients can use one of two techniques for gaining access to a broker's services:

- An application can use a WebSphere MQ client connection. The application can be running on the same system as the queue manager to which it connects, or on a different system. It can connect to a queue manager supporting a broker, or to any other queue manager in the WebSphere MQ network that has a defined path to the broker's queue manager.

  You can use all of the WebSphere MQ clients supported by WebSphere MQ Version 5.3, giving you the freedom to connect applications running in a wide variety of environments into your broker domain.
- An application can use a local connection to a queue manager. If it uses this method, the client must execute on the same system. It can connect to a queue manager supporting a broker, or to any other queue manager in the WebSphere MQ network that has a defined path to the broker's queue manager. (This is not possible on z/OS, where clients are not supported.)

WebSphere Message Broker does not impose any particular conditions or restrictions on applications.

WebSphere MQ Mobile Transport

WebSphere MQ Mobile Transport is a service that connects mobile and wireless applications that use WebSphere MQ Everyplace.

The operation of WebSphere MQ Everyplace is different from that of WebSphere Message Broker, which means that there are different concepts involved in using a broker to operate with WebSphere MQ Everyplace. In particular, the message format is different, and messages must be converted.

This transport cannot be used directly with the broker, instead WebSphere MQ Everyplace is installed separately, and an MQe Gateway configured on it that acts as an intermediary between MQe devices and the broker.

For more details on how to do this, see “Enabling WebSphere MQ Everyplace applications” on page 23.

For help with configuring WebSphere MQ Everyplace, see the documentation supplied with that product.

If you have message flows containing MQe nodes from a previous version broker and you want to use them, see how to migrate them in Migrating a message flow that contains WebSphere MQ Everyplace nodes.

WebSphere MQ Multicast Transport

WebSphere MQ Multicast Transport is a service that connects dedicated JMS application clients and is optimized for high volume, one-to-many publish/subscribe topologies.

WebSphere Message Broker provides support to allow these clients to communicate with other applications through message flows in a broker.

The ability to communicate with a broker means that JMS application multicast clients can communicate with applications that use other supported protocols and transports.

WebSphere MQ Real-time Transport

WebSphere MQ Real-time Transport is a service that connects dedicated JMS application clients.

These clients participate in publish/subscribe scenarios, sending messages across IP connections to other internet and intranet applications. WebSphere Message Broker provides support to allow these clients to communicate with other applications through message flows in a broker.

The ability to communicate with a broker means that JMS application clients can communicate with applications that use other supported protocols and transports.

WebSphere MQ Telemetry Transport

WebSphere MQ Telemetry Transport is a lightweight publish/subscribe protocol flowing over TCP/IP for remote sensors and control devices through low bandwidth communications.

This protocol is used by specialized applications on small footprint devices that require a low bandwidth communication, typically for remote data acquisition and process control.

A typical system might comprise several hundred client devices communicating with a single WebSphere Message Broker, where each client is identified by a unique ID. A single broker can manage a maximum of approximately 2000 clients.

WebSphere Message Broker uses the SCADAInput node to receive messages from WebSphere MQ Telemetry Transport client applications. The node interacts with a TCP/IP port to receive the messages.

Output is typically returned to the client application using a Publication node which embeds a SCADAOutput node. The Publication node filters and sends output from a message flow to subscribers who have registered an interest in a particular set of topics. If an application is using WebSphere MQ, the Publication node puts the message to the WebSphere MQ queue on the queue manager. For WebSphere MQ Telemetry Transport applications, the embedded SCADAOutput node routes the message to a subscribing WebSphere MQ Telemetry Transport client using a TCP/IP port.

SCADAInput nodes are available on all platforms except z/OS.

It is unlikely that you will use the SCADAOutput node directly, unless you write your own publication node for advanced applications.

Unlike WebSphere MQ and WebSphere MQ Everyplace, WebSphere MQ Telemetry Transport does not provide any security, although you can encrypt data, if required.

Quality of Service: You can specify a maximum Quality of Service (QoS) in a WebSphere MQ Telemetry Transport subscription message. QoS is similar to persistence in WebSphere MQ. There are three levels of QoS:

QoS0 ″At most once″ delivery. Delivery is not assured; acknowledgment is not expected.

QoS1 ″At least once″ delivery. Successful delivery is assured and an acknowledgment sent.

QoS2 ″Exactly once″ delivery. Similar to QoS1, but the message is assured not to be duplicated.

For further information about QoS see Quality of Service levels and flows

WebSphere MQ Web Services Transport

WebSphere MQ Web Services Transport connects Web services and clients that use the HTTP protocol for messaging.

WebSphere Message Broker allows Web service providers and their clients to communicate with:

- each other
- other applications using different protocols

through message flows in a broker. The topics linked below describe the WebSphere MQ Web Services Transport. Web service applications can also use other transports as described in "Web services, WSDL and message flows" on page 28.

Tip: You can create your own WSDL files, or generate WSDL files from message set definitions that you have created in the Message Brokers Toolkit, and use these files with tools such as Rational Application Developer or Microsoft Visual Studio .NET to build Web services applications that connect to WebSphere Message Broker. When you generate the WSDL file, you can specify one or more of the following bindings to be created:

- SOAP over HTTP (using WebSphere MQ Web Services Transport)
- SOAP over JMS (using WebSphere MQ Real-time Transport)

For more detail on WebSphere MQ Web Services Transport, refer to the following topics:

- Message format
- Headers
- Example messages

For more information, follow the Web services link on the developerWorks Web site.

For help with using HTTPS see Implementing SSL authentication.

Web services message format: Messages that pass across this transport have an XML format that provides platform-independence. The XML content that defines specific operations, and error situations (SOAP Fault), is defined by the SOAP standard. Because delivery is not guaranteed, applications must provide recovery in the event of system or network failures.

A message contains components that are appropriate to its type:

Command string
A request message contains the command POST followed by a URL. The command string ends with HTTP/1.0.

Status line
A response message contains a status line that starts with HTTP/1.0 , and

indicates the success or failure of the request with a code and text, for example 200 OK, 404 Not Found, 500 Server Error.

Headers
The headers used by the Web services protocol are HTTP headers that consist of a name followed by the characters colon and (optionally) a space, followed by the value (contents) of the header. Each header is followed by the characters <CR><LF>. The final header is followed by <CR><LF><CR><LF>, which creates a blank line. Data following this line is interpreted as message body data. Headers are described in "Web services headers."

Body The message body is the data following the headers in the message (that is, following the characters <CR><LF><CR><LF>). For an inbound message received by the HTTPInput or HTTPRequest node, this data is typically parsed according to the settings of the message properties in the node. For outbound messages, the data is typically parsed according to the settings of the message properties in the Properties folder received by the HTTPReply or HTTPRequest node that parses the message from message tree to bit stream.

In most cases, the body of a message is XML. However, an inbound message can contain a different format if the receiving node is configured to handle that format and the appropriate parsers are available. An outbound message can contain a body of any format that is accepted by the Web service to which it is sent.

An error that is generated by a component other than a Web service (for example, 404 Not Found) might result in a message that is in HTML format.

The bit stream containing headers and body are parsed and represented within the message tree when an input request is received by an HTTPInput node, or when a response from a Web service is received by the HTTPRequest node. A bit stream is created by parsers from the appropriate parts of the message tree when a reply is sent to the client by the HTTPReply node, and when a request is sent to a Web service by the HTTPRequest node. For further details about these actions, see the individual node descriptions.

Web services headers: When a message is received from a Web service client or Web server, the HTTPInput or HTTPRequest node that receives that message must parse the HTTP headers to create elements in the message tree. When an HTTPReply or HTTPRequest message sends a message to a Web service client or Web server, it parses the HTTP headers from the message tree into a bit stream.

The HTTP headers in a message depend on the type of message that is processed. There are four message types recognized in a message flow, and a parser is associated with each of these.

1. Input. An input message is received by the HTTPInput node from a Web service client. The HTTP headers in the input message (data up to and including the CRLFCRLF) are parsed by the HTTPInput parser and are included in the message tree under the correlation name HTTPInput. The headers shown in the following table are expected in an input message; others might also be present.

Header	Content	Example
Host	The hostname to which the client issued the message.	localhost
Content-Length	The length of the body of the input message in decimal (that follows the CRLFCRLF after the last header).	520
Content-Type	The type of the body data.	text/xml; charset=utf-8
SOAPAction		"" (empty string)

2. Reply. A reply message is sent by the HTTPReply node to the Web service client that sent the corresponding input message. The Web service headers in the reply message are created in the message tree under the correlation name HTTPReply, which is also the name of the parser used to parse that part of the message tree to a bit stream. You can choose to create your own HTTPReply header in a Compute node, or you can configure the HTTPReply node to create it using default values, or values taken from the HTTPReply or HTTPResponse trees in the input message, or both. If the HTTPReply node creates a default HTTPReply header, it contains the Web service headers and values shown in the following table.

Header	Value
Content-Length (if present in the input message)	The calculated length of the reply message body in decimal.
Content-Type	text/xml; charset=utf-8

3. Request. A request message is sent to a Web server by the HTTPRequest node. The HTTP headers in this message must be created in the message tree under the correlation name HTTPRequest, and are parsed by the HTTPRequest parser when the message tree is parsed to a bit stream. You can choose to create your own HTTPRequest header in a Compute node, or you can configure the HTTPRequest node to create it using default values, or values taken from the HTTPInput or HTTPRequest trees in the input message, or both. If the HTTPRequest node creates a default HTTPRequest header, it contains the Web service headers and values shown in the following table.

Header	Value
Host	Value set in the *Default Web Service URL* property.
Content-Length	The calculated length of the request message body in decimal.
Content-Type	text/xml; charset=utf-8
SOAPAction	"" (empty string)

4. Response. A response message is received by the HTTPRequest node from the Web server to which the corresponding request message was sent. The HTTP headers in the response message (data up to and including the CRLFCRLF) are parsed by the HTTPResponse parser and are included in the message tree under the correlation name HTTPResponse. The header shown in the following table is expected in a response message (though not required); others might also be present.

Header	Content	Example
Content-Length	The length of the response message body in decimal.	1585

"Web Services example messages" provides example messages that include these headers.

Web Services example messages: The following message provides an example of a request sent by an HTTPRequest node to a Web service that provides a look-up service:

```
POST /greenpages/servlet/rpcrouter HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 520
SOAP Action: ""
Cookie: JSESSIONID=0000B5OSLFIUDMQZFAUXKHD5ZDQ:-1

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schema.xmlsoap.org.soap/envelope/"
                   xmlns:xsi="http://www.w3/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getUserByName xmlns:ns1="http://tempuri.org/imb.GreenPages"
                   SOAP-ENV:encodingStyle="http:/schemas.xmlsoap.org/soap/encoding/">
<nameField xsl:type="xsd:string">bloggs, joe</nameField>
</ns1:getUserByName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The following message is an example of the response that might be returned to the HTTPRequest node:

```
HTTP/1.0 200 OK
Server: WebSphere Application Server/4.0
Content-Type: text/xml; charset=utf-8
Content-Length: 1585
Content-Language: en
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schema.xmlsoap.org.soap/envelope/"
                   xmlns:xsi="http://www.w3/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getUserByNameResponse xmlns:ns1="http://tempuri.org/imb.GreenPages"
                   SOAP-ENV:encodingStyle="http:/schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="http://www.greenpages.com/schemas/GreenPagesRemoteInterface"
        xsi:type="ns2:imb.UserRecord">
<fullName xsi:type="xsd:string">Joseph Bloggs</fullName>
<empNum xsi:type="xsd:int">65874</empNum>
<deskPhone xsi:type="xsd:string">(718)545-3623</deskPhone>
</return>
</ns1:getUserByNameResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

WebSphere Broker JMS Transport

The WebSphere Broker JMS Transport uses JMS as the connection protocol for sending and receiving messages in a Java environment.

The WebSphere Broker JMS Transport can be used to:

- Receive a JMS message as input.
- Create a JMS message for output.
- Inter-operate with message flows that do not expect JMS messages.

The exchange of JMS messages is achieved by using two built-in nodes, the JMSInput node and the JMSOutput node. These two nodes allow a message flow to receive messages from JMS destinations, or to send messages to JMS destinations. These destinations are accessible through connection to a JMS provider.

Two transformation nodes allow the JMSInput and JMSOutput nodes to inter-operate with nodes that expect a propagated message to contain an MQMD (and RFH2) header. These nodes are the JMSMQTransform node and the MQJMSTransform node. The JMSMQTransform node takes the output of the JMSInput node and produces a message that can be handled by an MQOutput node. The MQJMSTransform node transforms a message with an MQMD (and RFH2) header into a message that is expected by the JMSOutput node.

The JMS nodes can be used in applications where messages are produced and consumed from a variety of JMS destinations. In sending and receiving messages, the JMS nodes are behaving like JMS clients.

The JMS nodes work with the WebSphere MQ JMS provider, WebSphere Application Server Version 6.0, the IBM Service Integration Bus, and any JMS provider that conforms to the Java Message Service Specification, version 1.1.

The JMS Nodes sample sample is provided to help you start using the WebSphere Broker JMS Transport.

Support for JMS messages: In previous versions of WebSphere Message Broker, the support for JMS messages extended only to JMS provision. In WebSphere Message Broker Version 6.0, brokering value has been added to enable the broker to behave like a JMS client.

JMS provision

In previous versions of WebSphere Message Broker, the WebSphere MQ Real-time Transport enabled support for JMS provision; the built in nodes, Real-timeInput node, Real-timeOptimizedFlow node, and Publication node, allow JMS applications to communicate with applications that use other supported protocols and transports.

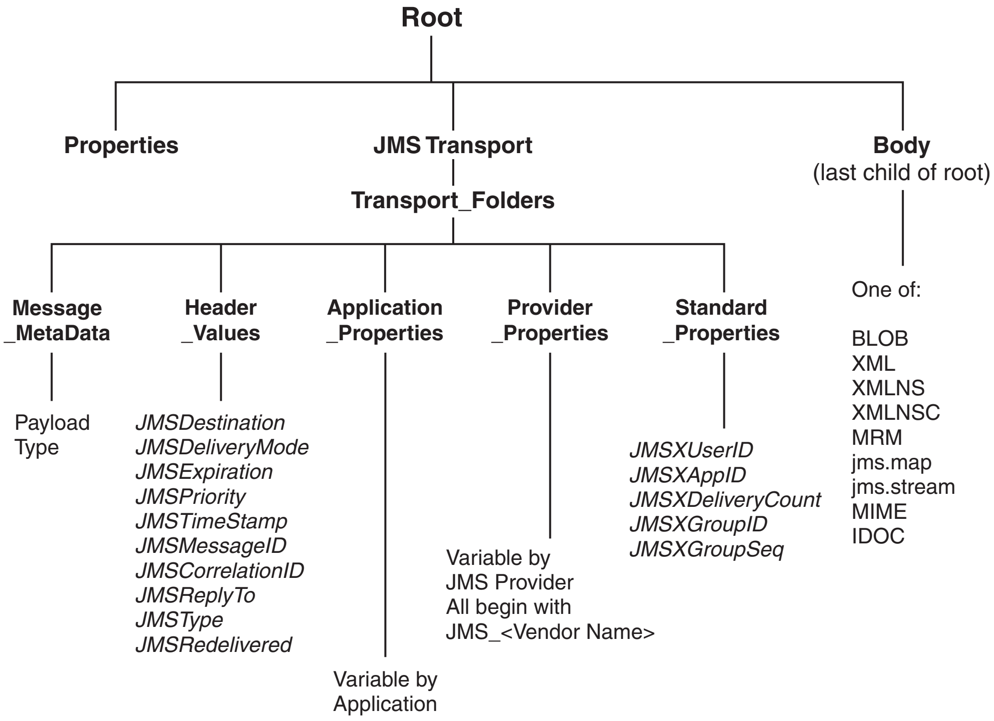
In this implementation, the Real-time node acts as a server for a JMS client, where the client can be WebSphere MQ.

JMS brokering

WebSphere Message Broker Version 6.0 adds brokering value to a JMS network. Four new built-in nodes, JMSInput node, JMSOutput node, JMSMQTransform node, and MQJMSTransform node, provide support for the broker to act like a JMS client. JMS messages can be sent and received, and can be transformed into other message formats.

Simplified JMS message representation: At the JMSInput node, a message is received as a Java object and not as a bit stream wire format (as would be the case with an MQInput node). The message does not populate an MQMD and RFH2 header, but instead populates a new message tree that represents a JMS message in a more native way.

To represent a JMS message in a message tree, a new canonical form has been created. This new message tree allows for representation of JMS message header data, and message properties. The JMS message tree is in a format that is recognizable to Java programmers.

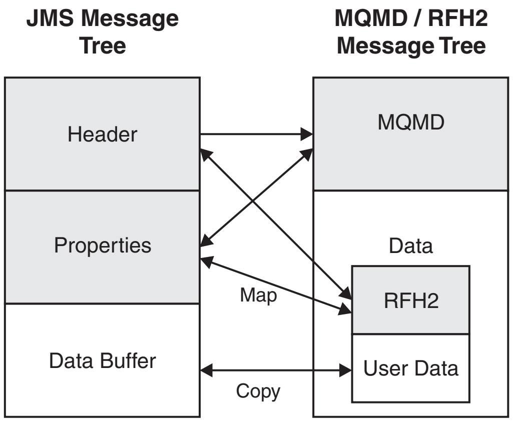


For details about the structure and content of the JMS message tree, see "Representation of messages across the JMS Transport" on page 90.

JMS message transformation: The JMSInput and JMSOutput nodes expect JMS messages and therefore expect a native JMS message tree representation. The following two nodes are available when a WebSphere MQ JMS message tree is expected but the representation is that of a JMS message tree:

- The JMSMQTransform node
- The MQJMSTransform node

These nodes do not have any configurable attributes. JMSMQTransform transforms a native JMS message tree to a WebSphere MQ JMS message tree, and MQJMSTransform achieves transformation in the opposite direction. The following diagram provides an overview of the mapping scheme that is used:



The above mapping uses the same scheme as the WebSphere MQ JMS provider to convert between a JMS message and an MQMD or MQRFH2 message.

When transforming between an WebSphere MQ message tree and a native JMS message tree, the transformation nodes copy elements from different parts of a message tree.

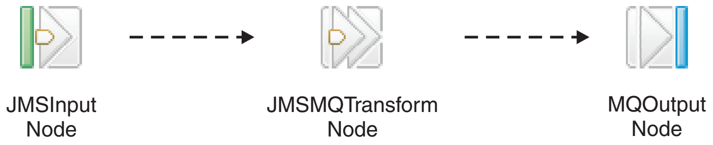
- For a WebSphere MQ JMS message tree:

  Elements are taken from the MQMD, MQRFH2, and UserData, and are copied into the appropriate JMS folders
- For a native JMS message tree:

  Elements are taken from the JMS header and properties, and are copied into the appropriate MQMD and RFH2 folders.

Example message flow scenario: JMSInput to MQOutput

Figure 16 JMSInput to MQOutput

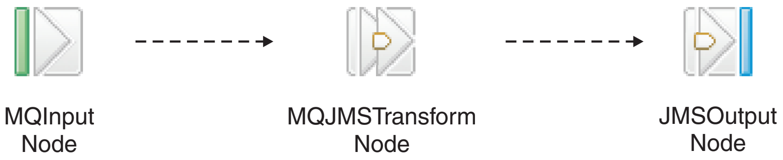


1. A JMSInput node is configured to subscribe to Topic ABC.
2. An application that is connected to the JMS server publishes on topic ABC.
3. A publication is received at the JMSInput node.
4. The node extracts data from the JMS message.
5. The JMS message is passed to the JMSMQTransform node where the message is converted to an MQ message.
6. The MQOutput node receives the MQ message, and publishes the message on a WebSphere MQ queue.

The final destination is to be an WebSphere MQ queue, and therefore the message must pass through a JMSMQTransform node to convert the message tree to a WebSphere MQ JMS format before it reaches the MQOutput node.

Example message flow scenario: MQInput to JMSOutput

Figure 17 HTTPInput to JMSOutput



1. An MQInput node receives a message from a WebSphere MQ queue.
2. The MQInput node creates an MQ message.
3. The MQ message is passed to the MQJMSTransform node where the message tree is converted to a JMS format.
4. The JMSOutput node receives the JMS message, extracts data from the JMS message, and creates a Java JMS message.
5. The JMSOutput node publishes the JMS message on Topic XYZ.

Note: These two examples are not intended to be exclusive, but rather demonstrate the range of solutions that can be achieved when using the JMS Transport. For example, although it is not shown here, the message could be passed to a Compute node or a JavaCompute node and the contents could be modified as required.

Refer to the JMS Nodes sample sample for examples of the JMS nodes being used in message flows.

Connection to different JMS providers: The JMSInput and JMSOutput nodes are compatible with, and work with, any JMS provider that conforms to the Java Message Service Specification, version 1.1. Futhermore, in order for these nodes to participate in a WebSphere Message Broker coordinated transaction, the JMS Provider must support the XAResource interface as defined in the Java Message Service Specification, version 1.1

Application environments

This section covers the following topics:
- "Web services, WSDL and message flows" on page 28

Web service applications

A Web service is defined by a WSDL definition, which describes both the logical interface of the service and its physical binding.

The broker can act as any of:
- Web service provider
- Web service client
- intermediary

and typically sends and receives SOAP messages at runtime.

For information on using the message broker in conjunction with Web services technologies see "Working with Web service applications" on page 27.

For general information on working with Web services in Message Brokers Toolkit see “Working with Web service applications” on page 27.

Enabling WebSphere MQ applications

If you want to connect WebSphere MQ client applications to the broker, you must define and secure the required resources. If you are connecting SAP clients, you must create metadata files to support their use.

- “Defining WebSphere MQ resources”
- “Securing WebSphere MQ resources” on page 21
- “Building the message model for the IDoc parser” on page 22

Defining WebSphere MQ resources

An application client can run on a system anywhere in the WebSphere MQ network. Because WebSphere Message Broker clients must use WebSphere MQ facilities to connect to the broker, and to interact with it (using the MQI and AMI), the setup of clients for WebSphere Message Broker is identical to that for clients for an WebSphere MQ server.

To support client connections to a broker:

1. If the application runs on the same system as the broker, it can establish a local connection with the broker’s queue manager using MQCONN, and you do not have to define any WebSphere MQ resources to support it.
2. If the application runs on the same system as another queue manager in the broker network, it can establish a local connection to that queue manager. In this scenario, you must define the appropriate resource to support communications between the queue manager to which the client has connected and the queue manager that hosts the broker that provides the required service.
3. If the application runs on a system that does not support a queue manager, it must make a client connection to a queue manager on another system. The queue manager to which it connects can be one of the following:
   - The broker’s queue manager

     You must set up the appropriate client connection and server connection definitions to support this option.
   - Another queue manager in the network

     You must set up the appropriate client connection and server connection definitions to support this option, and ensure that definitions are in place to support communications between the queue manager to which the client has connected and the queue manager that hosts the broker that provides the required service.

An application can only get messages from queues owned by the queue manager to which it is connected (this is true for all WebSphere MQ applications). Therefore, if an application expects to receive messages from a queue populated by a service within a particular broker and owned by that broker’s queue manager, it must connect to that broker’s queue manager (using a local or WebSphere MQ client connection).

An application that puts messages, however, can be connected to any queue manager in the network, as long as the queue manager can resolve the target destination in some way. In all cases, the queue manager to which the client application is connected must know the location of the queue or queues to which the application puts messages (for example using remote queue definitions).

When you define a WebSphere MQ queue as a node for a message flow, you must not give it a name that starts with SYSTEM_BROKER. This is reserved for queues defined for internal use by WebSphere Message Broker.

If your application is a subscriber, receiving messages published by other applications, it can specify a temporary dynamic queue as its subscriber queue. If it does so, the broker automatically deregisters the subscription when the queue is deleted.

For more details about applications, putting and getting messages, and the use of WebSphere MQ clients, see *WebSphere MQ Clients* and the *WebSphere MQ Application Programming Guide*.

Securing WebSphere MQ resources

Secure the WebSphere MQ resources that your configuration requires.

This section does not apply to z/OS.

WebSphere Message Broker depends on a number of WebSphere MQ resources to operate successfully. You must control access to these resources to ensure that the product components can access the resources on which they depend, and that these same resources are protected from other users.

Some authorizations are granted on your behalf when commands are issued. Others depend on the configuration of your broker domain.

- When you issue the command **mqsicreatebroker**, it grants put and get authority on your behalf to the group **mqbrkrs** for the following queues:
  - SYSTEM.BROKER.ADMIN.QUEUE
  - SYSTEM.BROKER.CONTROL.QUEUE
  - SYSTEM.BROKER.EXECUTIONGROUP.QUEUE
  - SYSTEM.BROKER.EXECUTIONGROUP.REPLY
  - SYSTEM.BROKER.INTERBROKER.QUEUE
  - SYSTEM.BROKER.MODEL.QUEUE
- When you issue the command **mqsicreateconfigmgr** it grants put and get authority on your behalf to the group **mqbrkrs** for the following queues:
  - SYSTEM.BROKER.CONFIG.QUEUE
  - SYSTEM.BROKER.CONFIG.REPLY
  - SYSTEM.BROKER.ADMIN.REPLY
  - SYSTEM.BROKER.SECURITY.REPLY
  - SYSTEM.BROKER.MODEL.QUEUE
- When you issue the command **mqsicreateusernameserver**, it grants put and get authority on your behalf to the group **mqbrkrs** for the following queues:
  - SYSTEM.BROKER.SECURITY.QUEUE
  - SYSTEM.BROKER.MODEL.QUEUE
- When you issue the command mqsicreateaclentry, it grants put and get authority on your behalf to the resource or user that you have specified for the command parameters -p or -u for the following queues:
  - SYSTEM.BROKER.CONFIG.QUEUE
  - SYSTEM.BROKER.CONFIG.REPLY

- If you have created WebSphere Message Broker components to run on different queue managers, the transmission queues that you define to handle the message traffic between the queue managers must have put and setall authority granted to the local **mqbrkrs** group, or to the service user ID of the component supported by the queue manager on which the transmission queue is defined.
- When you start the workbench, it connects to the Configuration Manager using a WebSphere MQ client/server connection. For details of WebSphere MQ channel security refer to ″Setting up WebSphere MQ client security″ in the *WebSphere MQ Clients* book.
- When you create and deploy a message flow, grant:
  1. get and inq authority to each input queue identified in an MQInput node, for the broker's ServiceUserID.
  2. put and inq authority to each output queue identified in an MQOutput node, or by an MQReply node, for the broker's ServiceUserID.
  3. get authority to each output queue identified in an MQOutput node or an MQReply node to the user ID under which a receiving or subscribing client application runs.
  4. put authority to each input queue identified in an MQInput node to the user ID under which a sending or publishing client application runs.

Building the message model for the IDoc parser

To use WebSphere Message Broker's capability to transform SAP IDoc data, you must model the segment data of the IDoc in WebSphere Message Broker.

The messages sent to and received from SAP applications are processed by the IDoc parser, which requires a message model to interpret the data correctly. This topic documents how to build the message model. An example prebuilt version of a message model is provided in SupportPac IA0F.

Obtaining the IDoc

Use the following procedure to create an import file of the required IDoc data for the Message Brokers Toolkit to use:

1. Logon to a SAP system
2. Run the supplied transaction `we60` that extracts the IDoc data as a C header file:
   a. For *ObjectName* select the IDoc type of interest, for example `MATMAS02`.
   b. Select the version of the record type. A version 4 IDoc is type **3**.
   c. Select **C-header** from the Documentation menu.
   d. Enter the filename for the output from the transaction when prompted. The C representation of the IDoc is saved to this file.
3. Use the Perl script `hdrfiddle.pl`, supplied in SupportPac IA0F, to ensure that the output file form you created is of the correct format to import into the Message Brokers Toolkit.

Modeling the IDoc

Use the following procedure to create your message model:

1. Use the New Message Set Project wizard to create a message set for your IDoc, for example `matmas02`, and to create a Custom Wire Format (CWF) physical format.
2. Use the message set editor to change the message set Runtime Parser property to IDOC.

3. Use the New Message Definition File wizard or the mqsicreatemsgdefs command to import the C representation of the IDoc into the new message set. When you are using the wizard or command, specify the following settings:
   - Create messages for the segments that appear in the IDoc.
   - Use the String Encoding option to import char arrays as fixed length strings.
   - Use the Padding Char for String option to make space (“ ”) the padding character used.

   Importing from C contains information on using the New Message Definition File wizard.
4. Rename each message to be the name of the segment it represents in uppercase letters. For example, for a segment called *E2MARAMM* the message must be named *E2MARAMM*.

The message set must be added to a broker archive file and deployed to a broker execution group, because the IDoc parser uses the MRM parser to parse the IDoc segments you have defined.

The Runtime Parser property, the message set name, and the Custom Wire Format name are used when you create a message flow to process an IDoc. They appear as the values of the Message Domain, Message Set and Message Format properties respectively of an MQinput node. Do not supply a Message Type property on the MQInput node, because the IDoc parser uses the contents of the SAP defined IDoc DD field segnam to provide the name of the message.

Accessing fields of the IDoc using ESQL

Use the ESQL editor Content Assist to fill in the SAP-defined fields of the IDoc. Invoke Content Assist from the Edit menu or, on some systems, by pressing Ctrl+Space if that has not been assigned to another function.

When you get to the `sdatatag` tag in an ESQL statement, the next tag is `MRM`, which must be entered manually, and then the field name to be manipulated. Specify the name of the field within the message segment here, instead of the name of the message segment.

For example, the following code sets the segment name of the IDoc:

```
SET OutputRoot.IDOC.DD[I].segnam  = 'E2MAKTM001';
```

The following example sets the `msgfn` field within the `E2MAKTM001` segment:

```
SET OutputRoot.IDOC.DD[I].sdatatag.MRM.msgfn = '006';
```

Enabling WebSphere MQ Everyplace applications

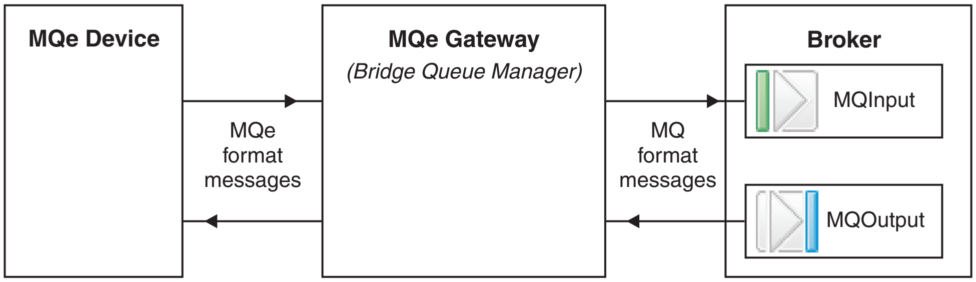
WebSphere MQ Mobile Transport is a service that connects mobile and wireless applications that use WebSphere MQ Everyplace.

WebSphere MQ Everyplace is an application designed primarily for messaging to, from, and between pervasive devices. These are typically small, handheld devices, such as mobile phones and PDAs.

The operation of WebSphere MQ Everyplace is different from that of WebSphere Message Broker, which means that there are different concepts involved in using a broker to operate with WebSphere MQ Everyplace. In particular, the message format is different, and messages must be converted.

Communication between WebSphere MQ Everyplace and a broker is achieved through the use of an external MQe Gateway (also called a *bridge-queue-manager*). WebSphere MQ Everyplace must be installed separately, and its MQe Gateway configured appropriately. Your flows use MQInput and MQOutput nodes to communicate with the gateway using MQ format messages. The gateway then communicates with MQe devices using MQe format messages.

The following diagram illustrates this visually:



For help with configuring WebSphere MQ Everyplace, see the documentation supplied with that product.

If you have message flows containing MQe nodes from a previous version broker and you want to use them, see how to migrate them in Migrating a message flow that contains WebSphere MQ Everyplace nodes.

Enabling WebSphere MQ Telemetry Transport applications

If you want to connect WebSphere MQ Telemetry Transport client applications to the broker, review the following guidance information:
- Designing applications
- An example message flow

The following SupportPacs are available from the WebSphere MQ SupportPacs Web page, and provide additional information to help you develop WebSphere MQ Telemetry Transport applications:
- IA92 Java implementation of WebSphere MQ Telemetry Transport
- IA93 C implementation of WebSphere MQ Telemetry Transport

Designing Telemetry applications

When you create a message flow to receive messages from Telemetry clients, you must include at least one SCADAInput node. Configure its properties to define the port on which it listens for new messages. If your message flow sends messages to Telemetry clients, you must include either a Publication node, or a SCADAOutput node (the Publication node includes an embedded SCADAOutput node).

You must deploy message flows that contain SCADAInput and SCADAOutput nodes to a single execution group within a broker. If you send messages to Telemetry clients through a Publication node, the message flow that contains that node must also be in the same execution group as a SCADAInput node, even if you do not have a message flow that receives messages from Telemetry clients.

This is because the properties of the SCADAInput node identify the TCP/IP port that is used for communication with the clients, and the characteristics of how messages are handled.

Starting and stopping listeners

Start and stop a WebSphere MQ Telemetry Transport listener using a publish message with the topic `$SYS/SCADA/MQIsdpListener/<port_number>`. Set the Payload part of the message set to ON or OFF. Replace `<port_number>` with the single port that you want to start or stop, or with `all` to start or stop all ports on the system that are designated as SCADA ports.

Improving message handling

The number of messages that are handled by a message flow depends both on message throughput and on response times. Review the guidance in Optimizing message flow response times and Optimizing message flow throughput. In addition, you must consider the Quality of Service that you choose for messages received from, or published to, Telemetry clients. This is described in "Choosing Quality of Service."

Choosing Quality of Service

Quality of Service determines the reliability of message delivery. Review the circumstances of the messages that are processed; in some situations, message loss might be acceptable. For other scenarios, message delivery might need to be guaranteed. The Quality of Service options, QoS0, QoS1, and QoS2, are described in "WebSphere MQ Telemetry Transport Quality of Service levels and flows" on page 58.

If you choose to guarantee message delivery, the broker must take additional actions to preserve the message until it is certain that it has been delivered. This affects broker and client performance, so you must balance the need for speed of message processing with the need to ensure message delivery.

If you choose QoS1 or QoS2, which indicates the message must be delivered at least, or only, once, the broker and client must provide a certain level of acknowledgment. The broker must store the message so that it can resend it if the appropriate acknowledgments are not received.

The broker stores messages in its database. This can affect message handling if the broker is unable to complete input to or output from the database when required; the broker might stop processing messages if this happens. If your broker database is DB2, turn off DB2 next key locking to avoid these deadlock problems. Issue the following command in a DB2 command window to make this change:

```
db2set DB2_RR_TO_RS=YES
```

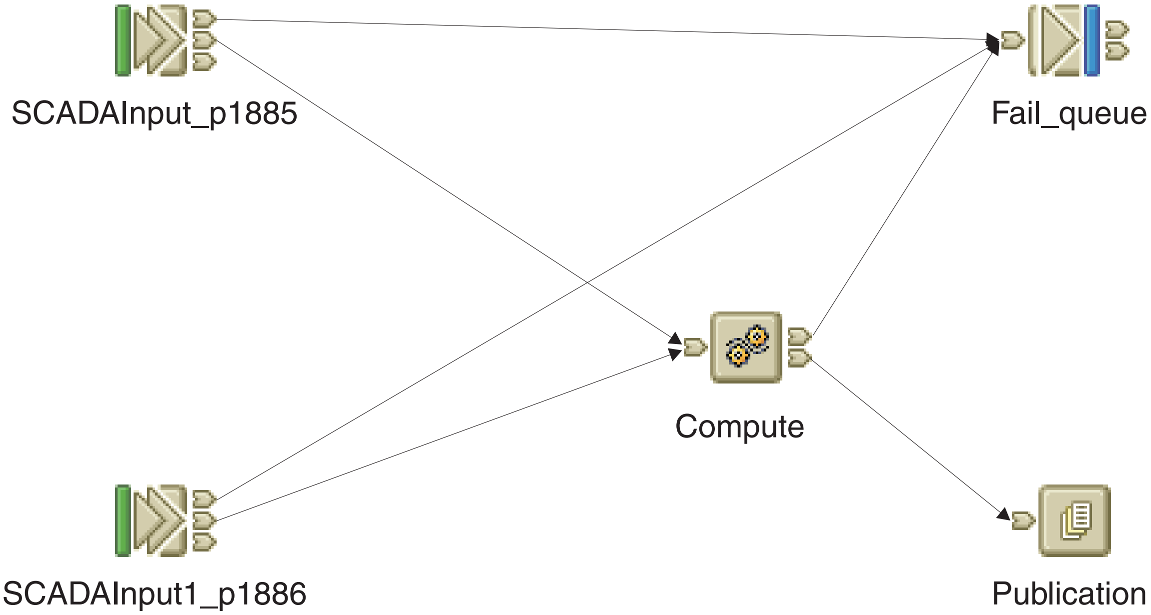
Restart the DB2 database manager for this change to take effect.

If you choose QoS0, message delivery is not guaranteed. The broker does not store the messages.

An example message flow to support Telemetry clients

This example message flow includes two SCADAInput nodes configured to listen on different TCP/IP ports. When a message is received without error, the input node propagates it to a Compute node that manipulates the input message content and generates a number of output messages. The Compute node propagates the output messages to a Publication node which publishes them to registered subscribers that use the WebSphere MQ protocol. Any errors in the flow are propagated to an MQOutput node, which records the error messages on a queue to be processed later.

The Telemetry clients generate events, for example to indicate a change of state, or to confirm that they are still active. The clients are programmed to batch up events and send them to the broker every 15 minutes. The message flow is designed to extract individual event messages from the batch message and publish these.



The Compute node has been configured with ESQL that splits the batched input message into individual event messages. A WHILE loop iterates over each message in the batch. The MQMD is copied from the input message to each output message. Relevant fields are copied from the input tree to the output tree. Each output message is a JMSText message which is built by setting the User properties in the usr folder within the MQRFH2 header. Each message is passed to the Publication node by a PROPAGATE statement.

Here is an example input batch message that contains two events:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited by Mary Bright -->
<events d_tstamp="20040417103118">
  <StateChange topic="LCUnit/12345/StateChange"
    d_tstamp="20040417104439" i_state="1" i_old_state="0">
    <![CDATA[Changing state from 'Starting' to 'Payload' because
    'The startup routine is complete']]&gt;
  </StateChange>
  <Heartbeat topic="LCUnit/12345/Heartbeat"
    d_tstamp="20040417105126" i_state="1">
    <d_tstamp>20040417104948</d_tstamp>
    <i_state>1</i_state>
  </Heartbeat>
</events>
```

The ESQL module that processes messages of this format is shown below:

```
CREATE COMPUTE MODULE messageflow_Compute
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
 DECLARE BatchTime CHAR;
 SET BatchTime = InputRoot.XML.events.d_tstamp;
 DECLARE Count INTEGER CARDINALITY(InputRoot.XML.events.*[]);
 DECLARE I INTEGER 2;

 WHILE I <= Count DO
  SET OutputRoot.Properties.MessageDomain = 'XML';
  SET OutputRoot.XML = NULL;
  SET OutputRoot.MQMD = InputRoot.MQMD;
  SET OutputRoot.MQRFH2.CodedCharSetId = 1208;
  SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR   ';
  SET OutputRoot.MQRFH2.(MQRFH2.Field)NameValueCCSID = 1208;
  SET OutputRoot.MQRFH2.psc.Topic = InputRoot.XML.events.*[I].topic;
  SET OutputRoot.MQRFH2.usr.*[] = InputRoot.XML.events.*[I].(XML.Attribute)*[];
  SET OutputRoot.MQRFH2.usr.b_time = BatchTime;
  SET OutputRoot.XML.Body.Text = InputRoot.XML.events.*[I].(XML.CDataSection)*'
  SET I = I + 1;
  PROPAGATE;
 END WHILE;

 RETURN FALSE;
   END;
END MODULE;
```

The message flow has two input nodes to increase the message handling capability. You can use any number of input nodes. You can also change the message flow property *Additional Instances* to increase the number of processes that service the message flow. If you have many hundreds of clients, you might find that you have to handle a high message load on two or more brokers. Use one or more of these techniques to find an acceptable level of message processing.

This example shows only one way in which Telemetry client messages can be handled. Change this message flow, or create a new one, to meet your own Telemetry requirements.

Working with Web service applications

A Web service is defined by a WSDL definition, which describes both the logical interface of the service and its physical binding.

The broker can act as any of:

- Web service provider
- Web service client
- intermediary

and typically sends and receives SOAP messages at runtime.

This section provides a brief overview of Web services and then describes how the broker can work with the related SOAP and WSDL technologies to interact with Web services.

This section covers the following topics:
- “Web services, WSDL and message flows”
- “Working with HTTP flows” on page 33
- “Web services scenarios” on page 36

Web services, WSDL and message flows

The term Web service here refers to an interface described by WSDL and accessible over one of the transports supported by the broker, generally HTTP, JMS or MQ.

For descriptions of the transports see:
- HTTP: “WebSphere MQ Web Services Transport” on page 12
- JMS: “WebSphere Broker JMS Transport” on page 15
- MQ: “WebSphere MQ Enterprise Transport” on page 9

WSDL

The logical part of a WSDL definition describes the format of the business messages to be exchanged and can be imported by the broker to create a message model at design time. (See Importing from WSDL).

The physical part of a WSDL definition describes the actual end-points for the service and the protocol used to exchange messages. The physical binding information is not imported at design time. You must implement the protocol and end-point details in a message flow, as follows:
- Using the binding SOAP/HTTP, in which case you would implement a flow using HTTP nodes (HTTPInput and HTTPReply if a flow implements a Web service, or HTTPRequest if a flow invokes a Web service).
- Using the binding SOAP/JMS, where you would implement a flow using JMS or MQ nodes.
- You can configure message flows that receive input messages from clients using one transport, and interact with a Web service or legacy application using another.
- You can propagate a message to more than one location – for instance the Web service response to be returned to a client by an HTTPReply node might first be sent to an auditing application using an MQOutput node (making any required adjustments to the message headers).

It is also possible to take an existing message model and *generate* a WSDL definition; in this case you must supply the physical binding information. (See Generating a Web Service Definition from a message set).

The WSDL bindings discussed here are specifically those supported directly by the WSDL Generator wizard:
- SOAP/HTTP
- SOAP/JMS

that is, those in which the message format used is SOAP, which can be parsed as XML in the MRM domain. However, note that the general definition of a Web service is broader than this and would allow message formats other than SOAP (for example XML-RPC) and transports other than HTTP or JMS (for example SMTP).

Your starting point is typically:

- an existing message model which is to be exposed to a client (requiring WSDL generation)
- an existing WSDL definition which will enable the broker to interact with a Web service (requiring WSDL import)

In both cases the broker will receive messages at runtime based on the WSDL which was generated or imported at design time.

(For more information on WSDL see Relationship of WSDL to Message Model and "WSDL validation" on page 33).

The scenarios

A broker message flow can:
- invoke external Web services
- implement a Web service
- do both

If the broker needs to interact with an existing Web service:
A WSDL definition for that service can be imported into a message set using the new WSDL importer. The resulting message set contains message definitions which model the SOAP envelope and the content of the corresponding SOAP messages. A flow developer can use these definitions to validate and work with an incoming message – for example defining a mapping to transform a SOAP request message into a SOAP response message.

If the broker already has existing message definitions which are going to be used in a Web services context:
The broker can export those definitions as a WSDL file which may then be consumed by other toolkits such as .NET.

To import WSDL:
Use the new option on the New Message Definition File wizard, or use the command `mqsicreatemsgsetsfromwsdl`. (See Importing data structures)

To export WSDL:
Set up categories using the Message Category File Wizard and Message Category Editor (See Working with a message category file) and then invoking the WSDL Generator Wizard (See Generating a Web Service Definition from a message set).

The pages in this section give an overview of the main Web services technologies:
- "Web services applications - SOAP"
- "Web services applications - WSDL" on page 31

Web services applications - SOAP

SOAP

SOAP is an XML based language defined by the W3C for sending data between applications. SOAP is transport and platform neutral.

Not all Web services use SOAP, but it is probably the most common message format for new Web services.

SOAP Message

A SOAP message comprises an envelope containing:

- an optional header (containing one or more header blocks)
- a mandatory body.

The content of the header and body is typically defined by WSDL.

SOAP style

SOAP defines two styles:

- **RPC** The SOAP body corresponds to a method call.
- **document**
  The SOAP body is typically a coarser-grained XML document and is defined explicitly by XML schema.

SOAP encodings

SOAP also defines two encodings:

- **SOAP encoding**
  With SOAP encoding the content is defined using an encoding scheme which implies a specific mapping to language-specific types.
- **literal** With literal encoding the SOAP content is defined explicitly by some schema (generally XML Schema).

SOAP style and encoding combinations

The above allows for four possible SOAP style and encoding combinations, but only the following three are meaningful and supported by the WSDL importer and generator:

- document - literal
- RPC - literal
- RPC - encoded (supported for WSDL importer only)

SOAP Versions

Two versions of SOAP are currently encountered: 1.1 and 1.2. SOAP 1.1 is perhaps more common, but has some interoperability issues (mainly concerned with the use of SOAP encoding) which are addressed by a separate standard: the WS-I Basic Profile.

More information

See the W3C at: http://www.w3.org and in particular the SOAP 1.1 and SOAP 1.2 documents at: http://www.w3.org/TR/soap.

See the WS-I at: http://www.ws-i.org/ and in particular the WS-I Basic Profile under http://www.ws-i.org/deliverables.

Web services applications - WSDL

WSDL

WSDL stands for Web Services Description Language and is an XML-based language that is defined by the W3C for describing an interface between applications.

A WSDL definition tells a client how to compose a Web services request and describes the interface that is provided by the server.

service

The WSDL document defines a *service* in terms of a number of:

- *ports* (WSDL 1.1)
- *endpoints* (WSDL 1.2)

These define where the service is available.

ports/endpoints

Each named port also defines a mechanism for accessing it – a *binding*. There is a separate binding for each supported protocol; for example, SOAP over HTTP. Each binding refers to a named:

- *portType* (WSDL 1.1)
- *interface* (WSDL 1.2)

bindings and portTypes/interfaces

- A binding defines the message wire format and transport details.
- A portType/interface is the logical interface to the Web service.

Both binding and portType/interface define *operations*.

operations

Each operation comprises *input* and *output* elements defined in terms of messages or message parts.

The optional parameterOrder attribute is only applicable for rpc-style WSDL. The value of the parameterOrder attribute is an ordered list of WSDL message parts providing a hint to a Web service provider about how the parts map to a method signature.

message elements

The *message* elements define a logical message in terms of one or more *parts*. (Each part might correspond to a parameter on a method call.)

A part is defined either as an XML Schema element or as an XML Schema type.

These elements and types are fully defined by either or both of the following:

- a WSDL <types> section
- imported schema (.xsd) files that provide the definitions for the WSDL part definitions.

More information

See the W3C at: http://www.w3.org and in particular the WSDL 1.1 document at: http://www.w3.org/TR/wsdl.

See the WS-I at: http://www.ws-i.org/ and in particular the WS-I Basic Profile under http://www.ws-i.org/deliverables.

WSDL operation types:

An operation type defines the expected use of the WSDL input, output and fault elements in the WSDL definition.

Operation types fall into two categories:

- request-style
- solicit-style - this type is not widely used and is not WS-I compliant.

Input and output messages:

- An input message is always the one being sent to the Web service
- An output message is always the one returned from the Web service

Effect of styles on input/output:

- In request-style, the input message is sent first; that is, the interaction is initiated by the client
- In solicit-style, the output message is sent first; that is, the interaction is initiated by the Web service

The operation type does impose some requirements on the flow (for example, whether the broker is initiating the interaction or listening). It is your responsibility to construct an appropriate flow for the operation type.

The presence and ordering of input and output messages implicitly defines the operation type in WSDL 1.1. For example a WSDL operation defining a single wsdl:input followed by a single wsdl:output implies that the operation type is request-response. Zero or more (*) wsdl:fault definitions may also appear and the Web service would return either the expected output message or one of the defined faults messages.

Table 1. WSDL 1.1 Operation Types

Operation Type	Description	Analogous to...	Order of Messages
request-response	client calls Web service – response expected	method call	Input output fault*
solicit-response	Web service solicits client – response expected	callback	output input fault*
one-way	client calls Web service – no response expected	procedure – no return	input
notification	Web service calls client – no response expected	signal – no response	output

WSDL validation: The W3C standards for Web services allow for more than one interpretation, and the WS-I (Web services Interoperability Organization) introduced a separate standard called the Basic Profile to tighten up their use. For example, the WS-I Basic Profile does not allow the use of SOAP encoding, even though this is often seen in practice.

The WS-I Validator can be used to check your WSDL definitions against the Basic Profile. The validator can be run either:

- manually against a specific .wsdl resource in the workbench. This will allow you to investigate and fix any WS-I compliance problems. Any validation issues are displayed as task list errors and warnings. The behavior of the validator can be controlled under **Preferences** → **Web services** → **WS-I Compliance**.
- automatically on importing a WSDL definition (by using the WSDL Importer wizard or the mqsicreatemsgdefsfromwsdl command) or WSDL generation (by the WSDL Generator wizard). In this case errors are displayed and also written to a report file.

  You are able to set a validation failure action to control the outcome of the validation as follows:

  ignore Any warnings from the WS-I validator will be written to the log file as informational. The user is unaware of the messages unless they view the log file. The user will be able to continue with the process.

  warn Any warnings from the WS-I validator will be written to the log file as warnings. Any messages that generate a warning will be highlighted with yellow symbols on the next panel of the wizard and the user will be able to continue with the process.

  fail Any warnings from the WS-I validator will be written to the log file as errors. Any messages that generate an error will be highlighted with red symbols on the next panel of the wizard and the user will not be able to continue with the process until the errors in the original WSDL definition are corrected and it is revalidated.

  You have to set the failure action to warn or ignore if you want to use features which are not WS-I compliant such as SOAP encoding.

Working with HTTP flows

This topic provides information which may be useful if you are using HTTP message flows to interact with Web services. You may find it useful to read this in conjunction with the subsequent “Web services scenarios” on page 36 section.

HTTPS

For help with using HTTPS see Implementing SSL authentication.

Setting the HTTP Status Code for a reply

The default HTTP Status Code is 200, which means success. If you want a different status code to be returned, then either:

- Set your status code in the field Destination.HTTP.ReplyStatusCode in the LocalEnvironment (correlation name OutputLocalEnvironment). This overrides any status code set in an HTTPResponseHeader as described below.
- If an HTTPRequest node precedes the HTTPReply node in your flow, then an HTTPResponseHeader will have been created in the logical tree, representing the HTTP headers in the response from another Web service. For convenience, if the HTTPReply node property Generate default HTTP headers from reply or response is selected, then values

from this header are used as defaults in creating the reply message, allowing you to set your status code in the field X-Original-HTTP-Status-Code in the HTTPResponseHeader. You can also set this field in an HTTPReplyHeader, but you are recommended to use the LocalEnvironment instead as described above.

Using the LocalEnvironment.Destination.HTTP.RequestIdentifier

When the HTTPInput node receives an input request message, it sets the LocalEnvironment field Destination.HTTP.RequestIdentifier to a unique value that identifies the Web service client that sent the request. You can refer to this value and you can save it to another location if appropriate.

For example, if you design a pair of message flows that interact with an existing WebSphere MQ application (as described in "Broker calls existing Web service" on page 37), you can save this value in the request flow and restore it in the reply flow to ensure that the correct client receives the reply. If you do so, you must not change the data, and you must retain the data as a BLOB.

The HTTPReply node extracts this value from LocalEnvironment and sets up the reply so that it is sent to the specific client.

If you design a message flow that includes both an HTTPInput and an HTTPReply node, the value is set into the LocalEnvironment by the HTTPInput node, but the HTTPReply node does not use it. Therefore if your message flow includes both nodes and a Compute node in the same flow, you do not have to include the LocalEnvironment tree when you specify which components of the message tree are copied from input message to output message by the Compute node (the *Compute mode* property).

Setting the HTTPRequest node URL dynamically

You can set the property *Default Web service URL* on the HTTPRequest node to determine the destination URL for a Web service request. You can configure a Compute node before the HTTPRequest node within the message flow to override the value set in the property. Code ESQL that stores a URL string in LocalEnvironment.Destination.HTTP.RequestURL; this is retrieved by the HTTPRequest node and used in place of the node property value.

Although you can also set the request URL in the special header X-Original-HTTP-URL in the HTTPRequestHeader section of the request message (which overrides all other settings) in a Compute node, you are recommended to use the LocalEnvironment content for this purpose.

Setting *Generate default HTTP headers from reply or response* for the HTTPReply node

If you select the *Generate default HTTP headers from reply or response* check box on the HTTPReply node properties dialog, the node includes a minimum set of headers in the response that is sent to the Web service client.

If you want to set any headers explicitly, create them in an HTTPReplyHeader. For example, a Compute node propagating a message in the XMLNS domain and wishing to modify the Content-Type could do so as follows:

```
CALL CopyMessageHeaders();
SET OutputRoot.HTTPReplyHeader."Content-Type" = 'text/xml';
SET OutputRoot.XMLNS = InputRoot.XMLNS;
```

In the particular case of Content-Type, do not set this header using the ContentType property unless you are working in the MIME domain. The ContentType property is specifically intended to set the value of Content-Type used in MIME.

The full set of HTTP headers used in the request is built by selecting the headers using the following algorithm:

1. Select any headers in an HTTPReplyHeader.
2. If no Content-Type header is defined so far, create one using any non-empty value in the ContentType property.
3. Select any headers in an HTTPResponseHeader (an HTTPResponsHeader is propagated on return from an HTTPRequest node).
4. If no Content-Type header is defined so far, create one with the default value `text/xml; charset=utf-8`
5. Create or overwrite the Content-Length header.

Attention: The HTTPReply node always rewrites the Content-Length header, even if you have cleared the *Generate default HTTP headers from reply or response* check box. This is to ensure that the content is correct.

If an HTTPReplyHeader section existed in the message received by the HTTPReply node, and the Output terminal of the HTTPReply node is connected, the HTTPReplyHeader section is updated with any changed or added values.

Setting *Generate default HTTP headers from input* **for the HTTPRequest node**

If you select the *Generate default HTTP headers from input* check box on the HTTPRequest node properties dialog, the node includes a minimum set of headers in the request that is sent to the server.

If you want to set any headers explicitly, create them in an HTTPRequestHeader. For example, a Compute node propagating a message in the XMLNS domain and wishing to modify the Content-Type could do so as follows:

```
CALL CopyMessageHeaders();
SET OutputRoot.HTTPRequestHeader."Content-Type" = 'text/xml';
SET OutputRoot.XMLNS = InputRoot.XMLNS;
```

In the particular case of Content-Type, do not set this header using the ContentType property unless you are working in the MIME domain. The ContentType property is specifically intended to set the value of Content-Type used in MIME.

The full set of HTTP headers used in the request is built by selecting the headers using the following algorithm:

1. Set the Host header, based on either the request URL or the incoming HTTPRequestHeader section of the message.
2. Select any headers in an HTTPRequestHeader.
3. If no Content-Type header is defined so far, create one using any non-empty value in the ContentType property.
4. Select any headers in an HTTPInputHeader (an HTTPInputHeader is created automatically by an HTTPInput node).

5. If no Content-Type header is defined so far, create one with the default value `text/xml; charset=utf-8`
6. If no SOAPAction header is defined so far, create one with the default value `''`
7. Create or overwrite the Content-Length header.

Attention: The HTTPRequest node always rewrites the Content-Length header, even if you have cleared the *Generate default HTTP headers from input or request* check box. This is to ensure that the content is correct.

If an HTTPRequestHeader exists in the received message, the HTTPRequestHeader is updated with any changed or added values.

Collecting HTTPListener trace if you have problems with HTTP

If you have problems with HTTP, you can trace the HTTPListener:

1. Start trace using the mqsichangetrace command.
2. View the HTTPListener trace log using the mqsireadlog command with the HTTPListener qualifier for the `-b` parameter.

Web services scenarios

This section describes the basic Web services scenarios. It is organized according to the role played by the broker.

You want the broker to invoke a Web service:

See "Broker calls existing Web service" on page 37

You want the broker to provide a new Web service wrapper to a non-web-service application:

See "Broker implements new Web service interface" on page 41

You already have a Web service client and you want the broker to make some existing non-web-service functionality available to it:

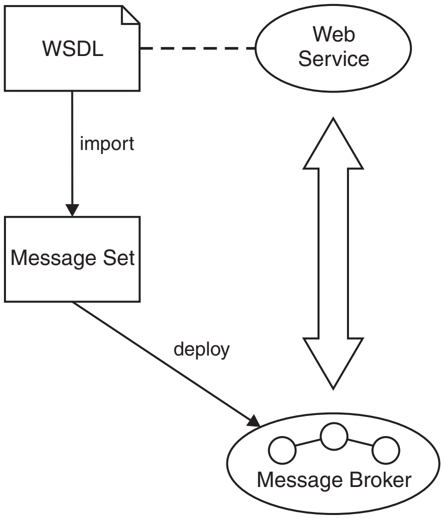
See "Broker implements existing Web service interface" on page 46

You have a non-web-service client and want the broker to make a corresponding Web service available to it:

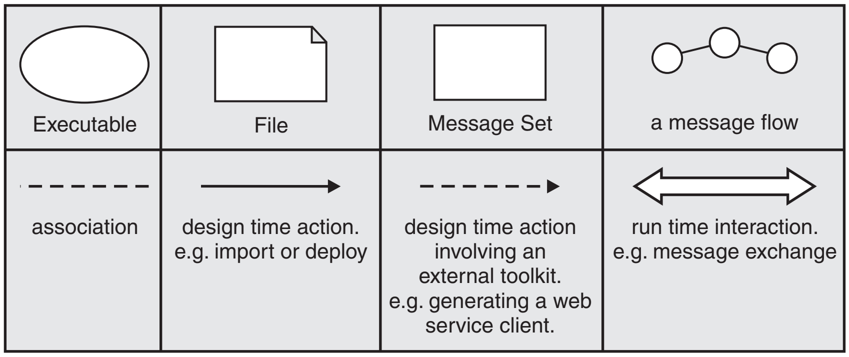
See "Broker implements non-web-service interface to new Web service" on page 50

Broker calls existing Web service

In this scenario, the broker invokes an existing Web service implementation.



Key to symbols:



Executable	File	Message Set	a message flow
association	design time action. e.g. import or deploy	design time action involving an external toolkit. e.g. generating a web service client.	run time interaction. e.g. message exchange

Possible uses

- You have a message flow and you want it to use a function already available as a Web service.
- You have an existing Web service and you want to provide a different interface to it. This could be an alternative Web services interface or a WebSphere MQ interface.
- You have an existing Web service and you want to change its implementation in some way without changing its interface; that is, the broker acts as an intermediary to the Web service. For instance a message flow could be used to enable auditing, or to transparently propagate the Web service response to another application.

Design steps

1. Import WSDL to create a message set containing definitions for the SOAP messages described by the WSDL.
2. Create a message flow to invoke the Web service. If the Web service transport is HTTP, then an HTTPrequest node would be used. If the Web service transport is JMS then appropriate JMS or WebSphere MQ nodes would be used.

Runtime

Your message flow creates an appropriately formatted Web service request, invokes the Web service, and parses the Web service response.

Example 1

In this example the broker acts as an intermediary to the Web service.

1. Create a message flow with nodes: HTTPInput, HTTPRequest, and HTTPReply.
2. The Web service client generates a Web service request. This is directed to the HTTPInput node by the broker.
3. The message flow processes the message in some way.
4. The HTTPRequest node sends a request to the original Web service and receives the response.
5. The flow generates a Web service response and puts it to the HTTPReply node. The response might be based in full or in part on the response received by the HTTPRequest node.
6. The broker sends the response to the Web service client.

If another application enabled for WebSphere MQ required the information in a different format then the message flow could arrange for the message to be propagated first to an MQOutput node for transmission to this application before continuing on to the end of the flow and the HTTPReply node which sends the expected Web service response. Appropriate Compute nodes would be required to transform the message (if required) and to manipulate the message headers (e.g. adding an MQMD header).

Example 2

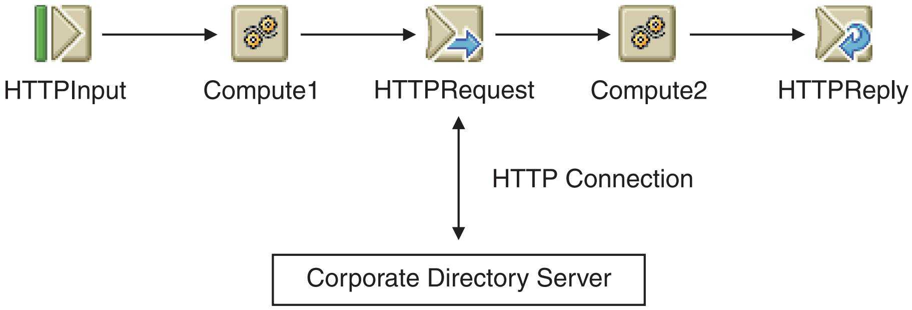
In this example the broker provides auditing for a Web service.

1. You design a message flow that includes an HTTPInput node connected to a Warehouse node.
2. The input message received from the Web service client is stored in the database by the Warehouse node.
3. The Warehouse node passes the message on to an HTTPRequest node, which interacts with the Web service.
4. When the response is received, the HTTPRequest node propagates the response to the HTTPReply node.
5. The HTTPReply node generates the response for the Web service client.

Example 3

The WebSphere MQ flow interacts with a Web service

You design a WebSphere MQ message flow that accesses a Web service across an HTTP connection. For example, you create a message flow that supports a process for the Human Resource department of your company. The process must retrieve employee ID numbers and enhance the message with this information. Employee IDs are maintained in the company's employee directory which is accessed through a Web service.

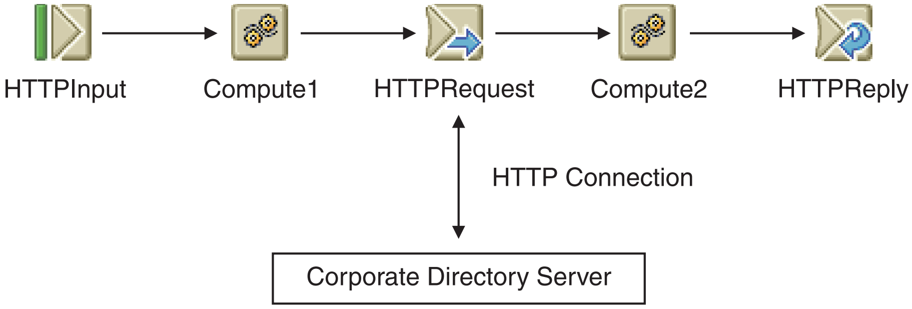


In this scenario, you typically clear the *Replace input message with Web service response* check box in the HTTPRequest node properties, and place the response from the corporate directory server in a temporary location in the message tree specified in the *Response message location in tree* property in the same node. In Compute2, you code ESQL to unpack the result, and augment the final message as appropriate.

Example 4

The message flow acts as a Web service intermediary

You design a message flow that interacts with a Web service that has updated its interface on behalf of Web service clients that are not aware of the new interface. This new message flow allows older clients to access the server using the new interfaces without updating their own interfaces.



Code ESQL in Compute1 to map the client request to a server request, and in Compute2 to map the server response to the client reply. You could define these request, reply, and response messages in the MRM domain to simplify the transformation from one format to another.

You can configure the HTTPRequest node to generate HTTP headers from the headers that are received by the HTTPInput node, which allows for cookies and other application-specific headers to be passed through. The HTTPReply node can provide an equivalent task to extract headers from the response from the Web service to return to the originating client. If you want this to be done, select the appropriate *Generate default HTTP headers from.....* check box on both the HTTPRequest and HTTPReply nodes.

In most scenarios, the original request is of no value, and you need only the response from the service to be able to generate the client reply message. If so, select the property *Replace input message with Web service response* on the HTTPRequest node. If you do want to preserve any data from the input request, you can store this in the LocalEnvironment in Compute1 and retrieve it in Compute2 for inclusion in the reply.

Broker calls existing Web service - detail:

This is an overview of a typical end-to-end scenario where the broker invokes a Web service.

There are two basic business propositions:

- You want to make an existing Web service available via a different interface. This could be an alternative Web services interface or an interface to WebSphere MQ.
- You want to incorporate the Web service into your business logic.

It is reasonable to assume that a WSDL definition is available for the Web service.

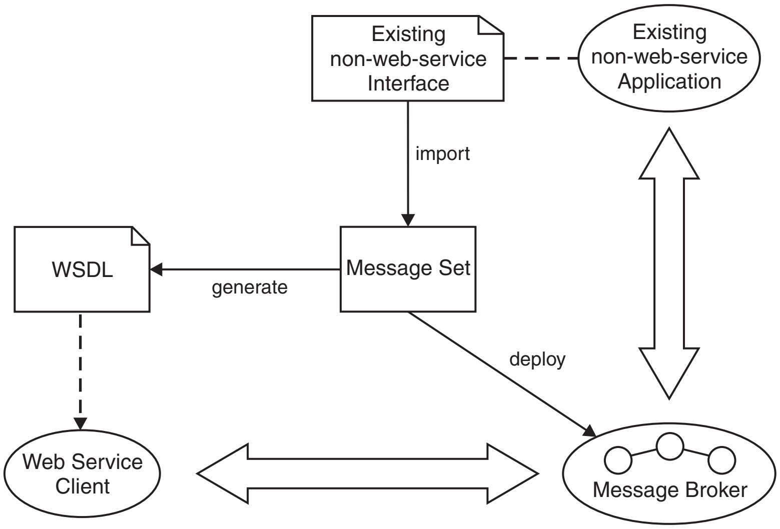
1. Import the WSDL definition for the Web service to create a message model for the payload data (see Importing data structures).
2. The WSDL import step will result in the appropriate SOAP mxsds being automatically included in the message set. Specifically this includes the SOAP envelope mxsd and – if required – the SOAP encoding mxsd.
3. Implement a message flow to make the Web service request – i.e. to act as the Web services client. The endpoint nodes might be MQ for integration with an existing system. The flow includes an HTTPRequest node to emit the Web service request and receive the response. The user must create the required SOAP content by populating a SOAP envelope instance. This might be done via a mapping node or created directly in ESQL:

```
DECLARE SOAPENV NAMESPACE 'http://schemas.xmlsoap.org/soap/envelope/';

SET OutputRoot.Properties.MessageSet = 'soap1';
SET OutputRoot.Properties.MessageType = 'Envelope';
SET OutputRoot.Properties.MessageFormat = 'XML1';

SET OutputRoot.MRM.SOAPENV:Body.addEntryResponse.cid = 'abc123';
.
.
```

4. The HTTPRequest node properties related to the request are:
   - domain: ″MRM″
   - set: the message set containing the SOAP Envelope definition
   - type: ″Envelope″
   - format: ″XML1″
   - whether the Web service request is to consist of the whole input message or just part of it
5. The HTTPRequest node properties related to the response are:
   - option to automatically redirect for HTTP status codes 300-399
   - whether a valid Web service response replaces the input message in the tree (default) or is attached at a specific location in the original tree
   - likewise, whether a Web service error replaces the input message in the tree (default) or is attached at a specific location in the original tree
   - the domain, set, type, and format (as in step 4 above). Used by the broker to create the logical tree (portion) from the bit stream
6. Once the response is available in the tree, you can process it using a mapping node or using ESQL. If the actual name of the response message is unknown (it isn't defined in the WSDL and appending Response to the request name is only a convention) then the user could write ESQL to check the children of the (arbitrarily named) response element.

Broker implements new Web service interface

In this Web service scenario, the broker provides a Web service interface to an existing non-web-service application.



Existing non-web-service Interface
Existing non-web-service Application
import
WSDL
generate
Message Set
deploy
Web Service Client
Message Broker

Key to symbols:

⬭ Executable	🗎 File	▭ Message Set	○—○—○ a message flow
- - - - - - association	⟶ design time action. e.g. import or deploy	- - - - - -▸ design time action involving an external toolkit. e.g. generating a web service client.	⇔ run time interaction. e.g. message exchange

This is sometimes referred to as a Web service facade. The design of the Web service interface will typically involve some regrouping, restriction or enhancement of the existing interface, and is not constrained by an existing WSDL definition.

Possible uses

The broker provides a Web services interface to an existing application, optionally providing other mix-in capabilities such as auditing the requests made.

Over time the implementation can be changed without affecting the interface presented to the Web services client.

Design steps

1. Create a message set for the business messages, possibly by importing an existing interface definition such as a C header file or COBOL copybook.
2. Generate a WSDL definition from the message set.

3. Use a SOAP toolkit such as Rational Application Developer to create a suitable Web services client based on the WSDL.
4. Develop a message flow to implement the Web service.

Runtime

Your message flow receives a Web service request, converts it into a form expected by the existing application and invokes the existing application. The response from the existing application is converted into a valid Web service response.

Example 1

An existing CICS application has a COBOL copybook interface.

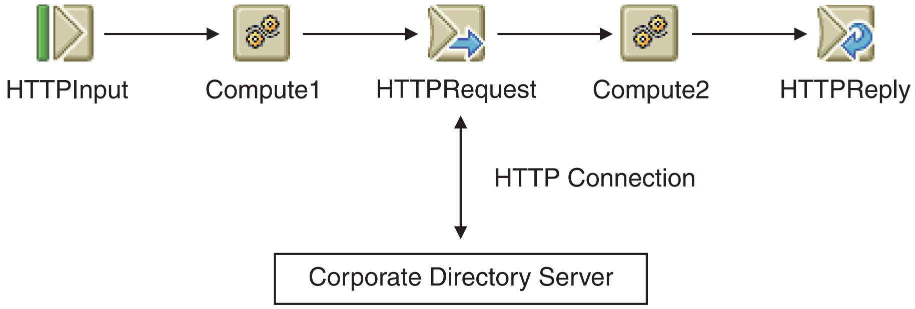
1. Import the COBOL copybook to create a message model.
2. Create a message flow with nodes: HTTPInput, HTTPReply and CICS.
3. Use the HTTPInput and HTTPReply nodes to provide the Web service facade.
4. Use the CICS node to invoke the original CICS implementation.

Example 2

The message flow is invoked as a Web service

You modify the design of an existing message flow to interact with Web service clients across HTTP. The existing message flow takes a well-defined input message, and the client can use WSDL exported from the WebSphere Message Broker tooling in invoking the message flow.

Most message flows that currently use WebSphere MQ for input or output can be adapted to use HTTP as a replacement or additional protocol.



You can model the input message in the MRM domain and generate WSDL from the model, or you could process a generic XML or XMLNS domain message. If you have defined the message in the MRM domain, you can configure the HTTPInput node to validate the input message. The node generates an exception if the message does not conform to the model.

You can configure the HTTPReply node to generate a set of default HTTP headers for the reply message sent to the client. This reduces the modifications that you must make to convert the message flow from one that processes WebSphere MQ messages to a flow that processes HTTP messages.

Example 3

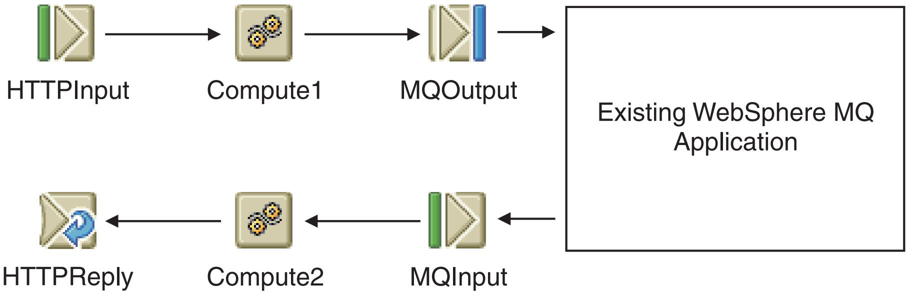
The message flow provides access to a WebSphere MQ application

You design two message flows that receive requests from and send replies to Web service clients, and interact with an existing WebSphere MQ application that has not been adapted to communicate across HTTP.

The first message flow receives inbound requests from Web service clients in an HTTPInput node. It includes a Compute node to transform the request in some way and an MQOutput node to send the modified request to the WebSphere MQ application.

The second message flow receives the response from that application in an MQInput node. The message is passed to a Compute node that transforms the message and propagates it to an HTTPReply node that sends it as a reply to the original Web service client.

Although the transformations completed by each Compute node might be trivial, the ESQL code in the first must save HTTP state information that is retrieved by the second to ensure that the replies from the WebSphere MQ application are returned to the client that sent the original request.



The first message flow receives the message, does whatever transformations are needed, and encodes the HTTP request identifier in the outbound message. (The request identifier could also be stored in a database if you prefer). The HTTPInput node provides the request identifier as a field in the LocalEnvironment tree called Destination.HTTP.RequestIdentifier and Compute1 can read and store this value.

The second message flow receives the reply message and transforms it back into the client message format. Compute2 reads the HTTP request identifier from the message, and sets LocalEnvironment.Destination.HTTP.RequestIdentifier using this value. The HTTPReply node uses the request identifier to ensure that the message reaches the correct HTTP client.

The implementation of this scenario requires correct handling of the MQMD. Messages that come into a message flow across HTTP must have an MQMD added before they are sent to an MQOutput node. Also, any messages coming in across WebSphere MQ must have the MQMD removed before being sent into an HTTPReply or HTTPRequest node (unless including an MQMD in the HTTP stream is desired).

In the ESQL module for Compute1, include a statement like this:

```
SET OutputRoot.XML.A.MessageID =
   CAST(InputLocalEnvironment.Destination.HTTP.RequestIdentifier AS CHARACTER);
```

In the ESQL module for Compute2, code a statement like this:

```
SET OutputLocalEnvironment.Destination.HTTP.RequestIdentifier =
   CAST(InputRoot.XML.A.MessageID AS BLOB);
```

Broker implements new Web service - detail:

This is an overview of a typical end-to-end scenario where the broker implements a Web service.

An existing C or COBOL based system offers some business logic which can usefully be exposed as a Web service.

There is some mechanism for the broker to invoke operations on the existing system (that is, the system exposes an interface to the broker). Typically the existing system would be enabled for WebSphere MQ, meaning that it receives MQ messages containing application data, dispatches these to the underlying implementation and then packages the return values as an MQ response. The data structures supplied to and returned by these existing operations are defined in a C header file or COBOL copybook.

The Web service is to offer an interface based on the operations already exposed by the existing system. This interface might comprise all the existing operations, some subset of them and/or composite operations.

To define your interface:

1. Look at the business function offered by the existing system
2. Select the subset of this business function to be exposed
3. Decide how that is to be represented in the interface (as many discrete operations, or as fewer multipurpose operations)

A basic decision is whether the Web service interface will be RPC-style or document-style. (See "Web services, WSDL and message flows" on page 28 and Relationship of WSDL to Message Model).

- An RPC-style interface is generally designed to map on to an underlying set of operations provided by some API, and the individual operations (method calls) have relatively small payloads.
- A document-style interface might have fewer operations each with a larger payload – for instance this might be a document representing a loan request.

The WS-I (see http://www.ws-i.org/) recommend the use only of document-style WSDL, but many older Web services use RPC-style.

To implement the scenario:

1. Import the existing API data structures – for example using the C importer. If document-style WSDL is to be used, you must have the importer wizard create the required global elements in the broker model. The WS-I recommends that the Web service payload be namespace qualified, so the user should specify the target namespace on the import.

   At this stage you have a message model for the data to be used in invoking the existing operations.
2. Generate the WSDL definition. Unless you have already created the required Message Categories, create a Message Category for each Web services operation to be exposed and associate the existing broker messages with the appropriate SOAP roles (input, output or fault). (See Working with a message category file)
   - If you chose document-style WSDL, the message set itself is not modified.
   - If you chose RPC-style WSDL, messages corresponding to the request and response for each WSDL operation are created and added to the message set automatically

3. The WSDL generation step will result in the appropriate SOAP mxsds (message definition files) being automatically included in the message set. Specifically this includes the SOAP envelope mxsd and (if the WSDL style is RPC-encoded) the SOAP encoding mxsd.
4. If you are creating a new Web services client, use the generated WSDL with your chosen Web services client technology. You do this with a tool other than WebSphere Message Broker, for example Rational Application Developer or .NET.
5. Implement a message flow to receive the Web service request, that is, to act as the Web services provider. The endpoint nodes are HTTP or MQ depending on the transport used by the client. The input node properties are:
   - domain: ″MRM″
   - set: the message set containing the SOAP Envelope definition
   - type: ″Envelope″
   - format ″XML1″
6. When invoked by the flow, the parser creates a logical tree comprising the SOAP envelope as defined by the pre-canned SOAP mxsd. Parsing continues automatically at the *attachment points* within the SOAP envelope (SOAP body and header), attempting to match against other message definitions in the message set. Apply validation at the input node if required.

   At this stage you have a logical tree, but you do not know which SOAP payload has been received. You could check the HTTP SOAPAction/action field to determine the likely content but this only works for HTTP. (Use of SOAPAction is not recommended by the WS-I.)
7. You can provide mapping from the allowed payload messages to the required messages from the existing system. For example you can have a single mapping definition with messages `message1a` and `message1b` both mapped to the same target `message2`.

   Alternatively, separate mappings could be provided for each message type – or for groups of related message types. This approach might result in more manageable and reusable mapping definitions. The disadvantage is that the user has to determine which payload they have received before they can apply the mapping. This can be done in ESQL as follows:

   ```
   DECLARE SOAPENV NAMESPACE 'http://schemas.xmlsoap.org/soap/envelope/';
   SET contentType = FIELDNAME(InputBody.SOAPENV:Body.*[<]);
   IF (contentType = 'foo') ...
   ```

   or using a field reference, for example:

   ```
   DECLARE R REFERENCE TO InputRoot.MRM.SOAPENV:Body.*[<];
   IF (FIELDNAME(R) = "foo") ...
   ```

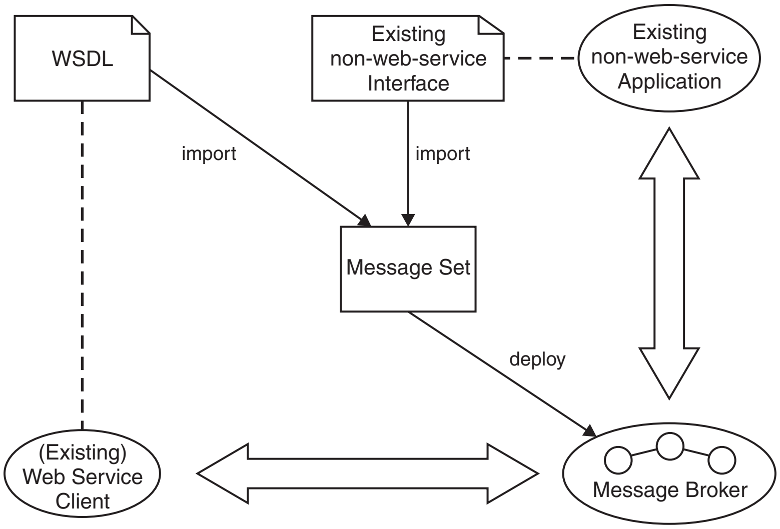
   Following that, now that the content is known, the flow could branch appropriately (for example using a RouteToLabel node) with each branch having content-specific Mapping and/or Compute nodes. For simple flows all the logic could be kept in a single Compute node if required.

   Now invoke the existing system (typically over WebSphere MQ), retrieve any expected response, and propagate the Web service reply. The dataflow designer must account for the possibility that the business application does not send the expected response in a reasonable period of time.

For a similar scenario, see the sample: Web Service Host sample.

Broker implements existing Web service interface

In this Web service scenario, the broker provides a Web services interface to an existing non-web-service application. The Web service is constrained to an existing WSDL definition.


WSDL
Existing non-web-service Interface
Existing non-web-service Application
import
import
Message Set
deploy
(Existing) Web Service Client
Message Broker

Key to symbols:

⬭ Executable	🗎 File	□ Message Set	o—o—o a message flow
- - - - - - association	——→ design time action. e.g. import or deploy	- - - - - → design time action involving an external toolkit. e.g. generating a web service client.	⇔ run time interaction. e.g. message exchange

Possible uses

- The broker provides a Web service implementation with a different quality of service from existing implementations.
- The broker provides a migration strategy for the existing implementation.

Design steps

1. Import WSDL to create a message set containing definitions for the SOAP messages described by the WSDL.
2. Adapt the message set for the required existing interface, possibly by importing an existing interface definition such as a C header file or COBOL copybook.
3. Develop a message flow to implement the Web service.

Runtime

Your message flow receives a Web service request, converts it into a form expected by the existing application and invokes the existing application. The response from the existing application is converted into a valid Web service response.

Example 1

In this example, an existing HTTP Web service client provides information on a given subject (stock prices or exchange rates, for example). You want to replace this service with an inhouse database lookup solution, but want to make no changes to the clients since these are widely deployed.

1. Create a message flow including an HTTPInput node that receives requests from your clients.
2. The HTTPInput node connects to a Compute node that retrieves the required information from the database and generates a new output message, in the form of a Web service response including this new data.
3. The Compute node propagates the message to the HTTPReply node, which generates the response for the Web service client.

Example 2

In this example, an existing, C or Cobol based, system can usefully be exposed as a Web service. There is also a constraint on what the Web service must look like – we already have the WSDL definition for a widely distributed Web services client which gives your users access to a particular business capability. The role of the broker will be to offer the same interface to a new implementation based on the existing system. Perhaps the original Web services provider offers a different quality of service or is to be discontinued for some reason.

1. Import the existing application interface.
2. Import an existing WSDL definition for the client.
3. Create a flow to implement the Web service interface and mediate with the existing application.

(See Importing from WSDL).

Broker implements existing Web service interface - detail:

This is an overview of a typical end-to-end scenario where you have a Web service client and you want the broker to make some existing non-web-service functionality available to it.

An existing C or Cobol based system offers some business logic which can usefully be exposed as a Web service.

Like the previous example ("Broker implements new Web service - detail" on page 43), there is some mechanism for the broker to invoke operations on the existing system (i.e. the system exposes an interface to the broker). Typically the existing system would be enabled for WebSphere MQ, meaning that it receives MQ messages containing application data, dispatches these to the underlying implementation and then packages the return values as an MQ response. The data structures supplied to and returned by these existing operations are defined in a C header file or Cobol copybook.

However in this example there is also a constraint on what the Web service must look like, that is, we already have the WSDL definition for the Web services client.

A possible scenario would be that a widely distributed Web services client already gives users access to a particular business capability and the role of the broker will be to offer the same interface to a new implementation based on the existing system. Perhaps the original Web services provider offers a different quality of service or is to be discontinued for some reason.

As before, the broker can invoke the existing system function over WebSphere MQ.

To implement the scenario:

1. Import the existing API data structures – for example using the C importer. If document-style WSDL is to be used, you must have the importer wizard create the required global elements in the broker model. The WS-I (see http://www.ws-i.org/) recommends that the Web service payload be namespace qualified, so the user should specify the target namespace on the import.

   At this stage you have a message model for the data to be used in invoking the existing operations.
2. Import the existing WSDL file to create the appropriate message model for the expected instance documents (see Importing data structures). The flow will parse the corresponding SOAP request and will need to transform to and from the required existing data format. You can inspect the imported message definitions and use the ESQL and/or Mapping editors to help create the flow. You do not create category files or generate WSDL from the broker model.
3. The WSDL import step will result in the appropriate SOAP mxsds being automatically included in the message set. Specifically this includes the SOAP envelope mxsd and – if required – the SOAP encoding mxsd.
4. Implement a message flow to receive the Web service request, that is, to act as the Web services provider. The endpoint nodes are HTTP or MQ depending on the transport used by the client. The input node properties are:
   - domain: ″MRM″
   - set: the message set containing the SOAP Envelope definition
   - type: ″Envelope″
   - format ″XML1″
5. When invoked by the flow, the parser creates a logical tree comprising the SOAP envelope as defined by the pre-canned SOAP mxsd. Parsing continues automatically at the *attachment points* within the SOAP envelope (SOAP body and header), attempting to match against other message definitions in the message set. Apply validation at the input node if required.

   At this stage you have a logical tree, but you do not know which SOAP payload has been received. You could check the HTTP SOAPAction/action field to determine the likely content but this only works for HTTP. (Use of SOAPAction is not recommended by the WS-I.)
6. You can provide a single mapping from the allowed payload messages to the required messages from the existing system. For example a single mapping definition could map messages `message1a` and `message1b` to the same target `message2`.

   Alternatively, separate mappings could be provided for each message type – or for groups of related message types. This approach may result in more manageable and reusable mapping definitions. The disadvantage is that the user has to determine which payload they have received before they can apply the mapping. This can be done in ESQL as follows:

   ```
   DECLARE SOAPENV NAMESPACE 'http://schemas.xmlsoap.org/soap/envelope/';
   SET contentType = FIELDNAME(InputBody.SOAPENV:Body.*[<]);
   IF (contentType = 'foo') ...
   ```

or using a field reference, for example:

```
DECLARE R REFERENCE TO InputRoot.MRM.SOAPENV:Body.*[<];
IF (FIELDNAME(R) = "foo") ...
```

Once the content is known, the flow could branch appropriately (e.g. RouteToLabel) with each branch having content-specific Mapping and/or Compute nodes. For simple flows all the logic could be kept in a single Compute node if required.

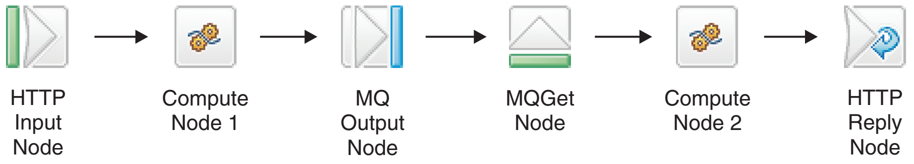
Now invoke the existing system (typically over WebSphere MQ), retrieve any expected response, and propagate the Web service reply. This is like the previous scenario ("Broker implements new Web service - detail" on page 43), except that the dataflow must also map between the data format used by the Web service client and the format used by the existing system enabled for WebSphere MQ. The dataflow designer must account for the possibility that the business application does not send the expected response in a reasonable period of time.

SOAP over JMS using MQGet node:

This is an overview of implementing a SOAP/JMS Web service using the MQGet node and describes the costs and benefits of this approach.

When using a synchronous request-reply pattern, the request message would be sent using an MQOutput node and the reply would then be received inline with an MQGet node.

A flow like the one in the following simplified diagram would enable synchronous SOAP-over-HTTP to SOAP-over-JMS translation:



The above flow provides a simple mechanism to implement a synchronous translation of HTTP to SOAP-over-JMS in which each incoming HTTP request is fully processed and responded to before moving onto the next. Failure of the SOAP-over-JMS service would be handled by the setting of a timeout on the MQGet node. Another advantage of this flow model is that the whole request-reply translation is handled in a single transaction enabling simple rollback and recovery.

This synchronous flow might not be appropriate in cases where the SOAP-over-JMS processing might be long running, since processing of other incoming HTTP requests will not be initiated until the current request is completed. In these cases the Message Flow would be better designed asynchronously, with separate request and reply flows, with any required correlation context being saved to a storage medium in the request flow and restored during the reply flow. A possible solution is to use a queue to store the required correlation context and use an MQGet node in the reply flow to retrieve it. This flow design allows the HTTP requests to be dispatched to the SOAP-over-JMS service as they are received and replies delivered back to HTTP in the order they are received. This would prevent, for example, fast running enquiry requests being blocked by a longer running update request.

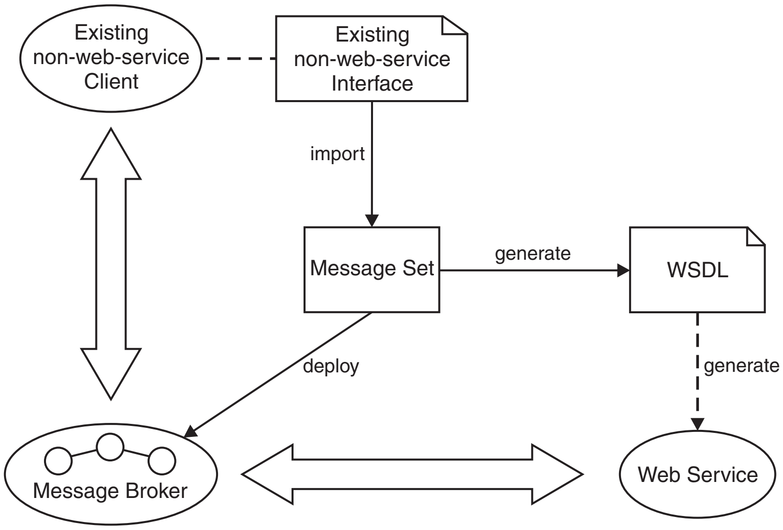
The asynchronous request-reply scenario is described in Using an MQGet node in a request-response flow.

The asynchronous request-reply scenario is also detailed in the sample Coordinated Request Reply sample which could be adapted for Web Service usage.

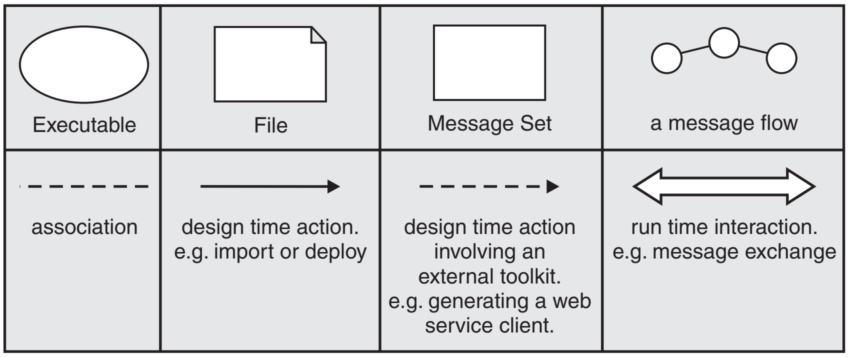
Another Web services scenario is described in the sample: Web Service Host sample.

Broker implements non-web-service interface to new Web service

In this Web service scenario, the broker provides backward compatibility for existing non-web-service clients to invoke a new Web services implementation provided by a SOAP toolkit.



Key to symbols:


Executable
File
Message Set
a message flow
association
design time action. e.g. import or deploy
design time action involving an external toolkit. e.g. generating a web service client.
run time interaction. e.g. message exchange

Possible uses

You want to migrate an application to a Web service implementation, for example an EJB implementation hosted by an application server to offer better scalability.

However, a significant number of your users have existing clients which cannot be immediately replaced. The broker allows the existing clients to use the new Web service implementation.

Design steps

1. Create a message set for the business messages, possibly by importing an existing interface definition such as a C header file or COBOL copybook.
2. Generate a WSDL definition from the message set.
3. Use a SOAP toolkit or application server to create a suitable Web services implementation based on the WSDL
4. Develop a message flow to mediate between the original existing client and the new Web service.

Runtime

Your message flow receives a request from the existing client, converts it into a Web services request and invokes the Web service. The response from the Web service is converted into a form understood by the existing client.

Part 2. Reference

End-user application support 55
Application transports 55
WebSphere MQ Enterprise Transport 55
WebSphere MQ Mobile Transport 56
WebSphere MQ Multicast Transport 56
WebSphere MQ Real-time Transport 57
WebSphere MQ Telemetry Transport 58
WebSphere MQ Web Services Transport 85
WebSphere Broker JMS Transport 86
Application environments 109
MQGet node message processing 109

End-user application support

The following reference information helps you to enable your end-user applications for use with WebSphere Message Broker:

- "WebSphere MQ Enterprise Transport"
- "WebSphere MQ Mobile Transport" on page 56
- "WebSphere MQ Multicast Transport" on page 56
- "WebSphere MQ Real-time Transport" on page 57
- "WebSphere MQ Telemetry Transport" on page 58
- "WebSphere MQ Web Services Transport" on page 85
- "WebSphere Broker JMS Transport" on page 86

For information about application compiler support, and connectivity between applications and brokers, refer to the relevant information for your operating system. For WebSphere MQ applications, further information is provided in the *WebSphere MQ Quick Beginnings* book for that system.

Application transports

This section covers the following topics:

- "WebSphere MQ Enterprise Transport"
- "WebSphere MQ Mobile Transport" on page 56
- "WebSphere MQ Multicast Transport" on page 56
- "WebSphere MQ Real-time Transport" on page 57
- "WebSphere MQ Telemetry Transport" on page 58
- "WebSphere MQ Web Services Transport" on page 85
- "WebSphere Broker JMS Transport" on page 86

WebSphere MQ Enterprise Transport

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

The WebSphere MQ Enterprise Transport provides all the reliable messaging features available in WebSphere MQ. This transport provides persistent and non-persistent messaging and supports transactions. To use the WebSphere MQ Enterprise Transport, you must deploy a message flow that contains an MQInput node to your broker. If this message flow sends output messages to other WebSphere MQ applications, it must also include an MQOutput, MQReply, or Publication Node.

The queue specified in the MQInput node determines the queue on which the broker receives publications from publishing applications. Subscribers connect to the broker by sending a registration request to the broker's SYSTEM.BROKER.CONTROL.QUEUE. The subscriber specifies a queue on which they want to receive any publications on the registered topic in the registration request.

The WebSphere MQ Enterprise Transport is a queued transport, applications communicate with the broker by writing data to and reading data from message queues. Use the WebSphere MQ Enterprise Transport when you require assured delivery of messages or need to use transactional support. There are overheads involved in using the WebSphere MQ Enterprise Transport, therefore it does not offer the same levels of performance and scalability as the WebSphere MQ Real-time Transport.

All WebSphere Message Broker applications, like WebSphere MQ applications, can use all the supported WebSphere MQ interfaces to put messages to the message flow queues. In fact, every WebSphere MQ application is a potential WebSphere Message Broker application.

These applications use one of two techniques to gain access to the broker's services:

- An application can use a WebSphere MQ client connection. You can use all the WebSphere MQ clients supported by Version 5.1 or later. This allows you to connect applications running in a wide variety of environments into your broker domain. An application running on the same system as the queue manager to which it connects can also use a client connection.
- An application running on the same system as a broker can use a local connection to the queue manager that hosts that broker.

Receiving applications can get the messages put to the output queue or queues of a message flow when they have been processed by that message flow. The applications must be connected, either by a client/server connection, or via a local connection, to the queue manager that owns the queue or queues defined as the target for their messages. If the message flow provides a publish/subscribe service, the publication node puts the messages to the queue specified by the subscriber as its local receiver queue.

Applications that connect using WebSphere MQ Enterprise Transport use a mixture of point-to point and publish/subscribe models.

The following built-in nodes are provided to support this protocol:

- MQInput
- MQOutput
- MQReply
- Publication

WebSphere MQ Mobile Transport

WebSphere MQ Mobile Transport is used exclusively by WebSphere MQ Everyplace clients.

This transport cannot be used directly with the broker, instead WebSphere MQ Everyplace is installed separately, and an MQe Gateway configured on it that acts as an intermediary between MQe devices and the broker.

For more details on how to do this, see "Enabling WebSphere MQ Everyplace applications" on page 23.

WebSphere MQ Multicast Transport

Applications that connect using WebSphere MQ Multicast Transport and the JMS API use predominantly the publish/subscribe model. The applications must be multicast-enabled to use this protocol.

To use the WebSphere MQ Multicast Transport, you must deploy a message flow that contains a Real-timeOptimizedFlow node or a Real-timeInput node to your broker. The message flow can send output messages to other real-time applications, using either the Real-timeOptimizedFlow or the Publication node.

This protocol is a non-queued transport: applications communicate with the broker by writing data directly to TCP/IP ports and the input nodes are configured with a TCP/IP port number on which the broker listens for incoming connections. Client applications that use the WebSphere MQ Multicast Transport connect to this port.

The following built-in nodes are provided to support this protocol:
- Real-timeInput
- Real-timeOptimizedFlow
- Publication

WebSphere MQ Real-time Transport

WebSphere MQ Real-time Transport is a lightweight protocol optimized for use with nonpersistent messaging. It is used exclusively by Java Message Service (JMS) clients and provides high levels of scalability and message throughput.

To use the WebSphere MQ Real-time Transport, you must deploy a message flow that contains a Real-timeOptimizedFlow node or a Real-timeInput node to your broker. (The Real-timeInput node is an input node and the Real-timeOptimizedFlow node is a complete message flow that provides a high performance publish/subscribe message flow.) The message flow can send output messages to other real-time applications, using either the Real-timeOptimizedFlow or the Publication node.

This protocol is a non-queued transport: applications communicate with the broker by writing data directly to TCP/IP ports and the input nodes are configured with a TCP/IP port number on which the broker listens for incoming connections. Client applications that use the WebSphere MQ Real-time Transport connect to this port.

WebSphere MQ Real-time Transport is suited for applications and environments where you need to send large numbers of messages, or where messages are to be sent to large numbers of client applications. Use this protocol for applications that must rely on the quality of service provided by TCP/IP but do not need persistent delivery.

For example, you can use this protocol in situations where a piece of data is updated very frequently, such as updating a scoreboard for a sporting event or updating a share price on a stock ticker. Because this is a lightweight protocol, it offers higher levels of performance for nonpersistent messaging than WebSphere MQ Enterprise Transport. The WebSphere MQ Real-time Transport does not provide any facilities for persistent messaging or durable subscriptions.

You can generate WSDL files from message set definitions that you have created in the workbench, and use these files with tools such as WebSphere Studio Application Developer Integration Edition or Microsoft Visual Studio.NET to build JMS client applications that connect to WebSphere Message Broker. When you generate the WSDL file, you can specify one or more of the following bindings to be created:
- SOAP over HTTP

- SOAP over JMS
- JMS TextMessage

The first of these is supported using the WebSphere MQ Web Services Transport. The other two are supported using the WebSphere MQ Real-time Transport.

Applications that connect using WebSphere MQ Real-time Transport and the JMS use predominantly the publish/subscribe model.

The following built-in nodes are provided to support this protocol:
- Real-timeInput
- Real-timeOptimizedFlow
- Publication

WebSphere MQ Telemetry Transport

Client applications use WebSphere MQ Telemetry Transport to send messages to SCADAInput nodes and receive message from Publication or SCADAOutput nodes in a message flow.

The transport is described in the following topics:
- Quality of Service levels and flows
- Message format
- Command messages

WebSphere MQ Telemetry Transport Quality of Service levels and flows

WebSphere MQ Telemetry Transport delivers messages according to the levels defined in a Quality of Service (QoS). The levels are described below:

QoS level 0 At most once delivery

The message is delivered according to the best efforts of the underlying TCP/IP network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the broker either once or not at all.

The table below shows the QoS level 0 protocol flow.

Client	Message and direction	Broker
QoS = 0	PUBLISH ---------->	**Action:** Publish message to subscribers

QoS level 1 At least once delivery

The receipt of a message by the broker is acknowledged by a PUBACK message. If there is an identified failure of either the communications link or the sending device, or the acknowledgment message is not received after a specified period of time, the sender resends the message with the DUP bit set in the message header. The message arrives at the broker at least once. Both SUBSCRIBE and UNSUBSCRIBE messages use QoS level 1.

A message with QoS level 1 has a Message ID in the message header.

The table below shows the QoS level 1 protocol flow.

Client	Message and direction	Broker

QoS = 1
DUP = 0
Message ID = x
	PUBLISH
	---------->
		Actions:
		• Store message in database
		• Publish message to subscribers

Action: Discard message	PUBACK <----------	

If the client does not receive a PUBACK message (either within a time period defined in the application, or if a failure is detected and the communications session is restarted), the client resends the PUBLISH message with the DUP flag set.

When it receives a duplicate message from the client, the broker republishes the message to the subscribers, and sends another PUBACK message.

QoS level 2 Exactly once delivery

Additional protocol flows above QoS level 1 ensure that duplicate messages are not delivered to the receiving application. This is the highest level of delivery, for use when duplicate messages are not acceptable. There is an increase in network traffic, but it is usually acceptable because of the importance of the message content.

A message with QoS level 2 has a Message ID in the message header.

The table below shows the QoS level 2 protocol flow.

Client	Message and direction	Broker

QoS = 2
DUP = 0
Message ID = x
	PUBLISH
	---------->
		Action: Store message in database

	PUBREC <----------	Message ID = x
Message ID = x	PUBREL ---------->	**Actions:** • Update database • Publish message to subscribers
Action: Discard message	PUBCOMP <----------	Message ID = x

If a failure is detected, or after a defined time period, each part of the protocol flow is retried with the DUP bit set. The additional protocol flows ensure that the message is delivered to subscribers once only.

Because QoS1 and QoS2 indicate that messages must be delivered, the broker stores messages in a database. If the broker has problems accessing this data, messages might be lost. For more details, and actions you can take to reduce these problems, see “Designing Telemetry applications” on page 24.

Assumptions for QoS levels 1 and 2:

In any network, it is possible for devices or communication links to fail. If this happens, one end of the link might not know what is happening at the other end; these are known as in doubt windows. In these scenarios assumptions have to be made about the reliability of the devices and networks involved in message delivery.

WebSphere MQ Telemetry Transport assumes that the client and broker are generally reliable, and that the communications channel is more likely to be unreliable. If the client device fails, it is typically a catastrophic failure, rather than a transient one. The possibility of recovering data from the device is low. Some devices have non-volatile storage, for example flash ROM. The provision of more persistent storage on the client device protects the most critical data from some modes of failure.

Beyond the basic failure of the communications link, the failure mode matrix becomes complex, resulting in more scenarios than the specification for WebSphere MQ Telemetry Transport can handle.

The time delay (retry interval) before resending a message that has not been acknowledged is specific to the application, and is not defined by the protocol specification.

WebSphere MQ Telemetry Transport message format

The message header for each WebSphere MQ Telemetry Transport command message contains a fixed header. Some messages also require a variable header and a payload. The format for each part of the message header is described in the following topics:

- Fixed header
- Payload
- Variable header

WebSphere MQ Telemetry Transport fixed header:

The message header for each WebSphere MQ Telemetry Transport command message contains a fixed header. The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	Remaining Length							

Byte 1

Contains the Message Type and Flags (Dup, QoS level, and RETAIN) fields.

Byte 2

(At least one byte) contains the Remaining Length field.

The fields are described in the following sections. All data values are in big-endian order: higher order bytes precede lower order bytes. A 16-bit word is presented on the wire as Most Significant Byte (MSB), followed by Least Significant Byte (LSB).

Message Type:

Position: byte 1, bits 7-4.

Represented as a 4-bit unsigned value. The enumerations for this version of the protocol are shown in the table below.

Mnemonic	Enumeration	Description
Reserved	0	Reserved
CONNECT	1	Client request to connect to Broker
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received (assured delivery part 1)
PUBREL	6	Publish Release (assured delivery part 2)
PUBCOMP	7	Publish Complete (assured delivery part 3)
SUBSCRIBE	8	Client Subscribe request
SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	PING Request
PINGRESP	13	PING Response
DISCONNECT	14	Client is Disconnecting
Reserved	15	Reserved

Flags:

The remaining bits of byte 1 contain the fields DUP, QoS, and RETAIN. The bit positions are encoded to represent the flags as shown in the table below.

Bit position	Name	Description
3	DUP	Duplicate delivery
2-1	QoS	Quality of Service
0	RETAIN	RETAIN flag

DUP

Position: byte 1, bit 3.

This flag is set when the client or broker attempts to re-deliver a PUBLISH message. This applies to messages where the value of QoS is greater than zero (0), and an acknowledgment is required. When the DUP bit is set, the variable header includes a Message ID.

QoS

Position: byte 1, bits 2-1.

This flag indicates the level of assurance for delivery of a PUBLISH message. The QoS levels are shown in the table below.

QoS value	bit 2	bit 1	Description		
0	0	0	At most once	Fire and Forget	<=1
1	0	1	At least once	Acknowledged delivery	>=1

QoS value	bit 2	bit 1	Description		
2	1	0	Exactly once	Assured delivery	=1
3	1	1	Reserved		

RETAIN

Position: byte 1, bit 0.

When set, the Retain flag indicates that the broker holds the message, and sends it as an initial message to new subscribers to this topic. This means that a new client connecting to the broker can quickly establish the current number of topics. This is useful where publishers send messages on a ″report by exception″ basis, and it might be some time before a new subscriber receives data on a particular topic. The data has a value of retained or Last Known Good (LKG).

After sending a SUBSCRIBE message to one or more topics, a subscriber receives a SUBACK message, followed by one message for each newly subscribed topic that has a retained value. The retained value is published from the broker to the subscriber with the Retain flag set and with the same QoS with which it was originally published, and is therefore subject to the usual QoS delivery assurances. The Retain flag is set in the message to the subscribers, to distinguish it from ″live″ data so that it is handled appropriately by the subscriber.

Because a broker might no longer hold a previously Retained PUBLISH message, there is no guarantee that the subscriber will receive an initial Retained PUBLISH message on a topic.

Remaining Length:

Position: byte 2.

Represents the number of bytes remaining within the current message, including data in the variable header and the payload.

The variable length encoding scheme uses a single byte for messages up to 127 bytes long. Longer messages are handled as follows. Seven bits of each byte encode the Remaining Length data, and the eighth bit indicates any following bytes in the representation. Each byte encodes 128 values and a ″continuation bit″. For example, the number 64 decimal is encoded as a single byte, decimal value 64, hex 0x40. The number 321 decimal (=128x2 + 65) is encoded as two bytes, least significant first. The first byte is 2+128 = 130. Note that the top bit is set to indicate at least one following byte. The second byte is 65.

The protocol limits the number of bytes in the representation to a maximum of four. This allows applications to send messages of up to 268 435 455 (256 MB). The representation of this number on the wire is: 0xFF, 0xFF, 0xFF, 0x7F.

The table below shows the Remaining Length values represented by increasing numbers of bytes.

Digits	From	To
1	0 (0x00)	127 (0x7F)
2	128 (0x80, 0x01)	16 383 (0xFF, 0x7F)
3	16 384 (0x80, 0x80, 0x01)	2 097 151 (0xFF, 0xFF, 0x7F)

Digits	From	To
4	2 097 152 (0x80, 0x80, 0x80, 0x01)	268 435 455 (0xFF, 0xFF, 0xFF, 0x7F)

The algorithm for encoding a decimal number (X) into the variable length encoding scheme is as follows:

```
do
  digit = X MOD 128
  x = X DIV 128
  // if there are more digits to encode, set the top bit of this digit
  if ( X > 0 )
    digit = digit OR 0x80
  endif
  'output' digit
while ( X > 0 )
```

where MOD is the modulo operator (% in C), DIV is integer division (/ in C), and OR is bit-wise or (| in C).

The algorithm for decoding the Remaining Length field is as follows:

```
multiplier = 1
     value = 0
     do
          digit = 'next digit from stream'
          value += (digit AND 127) * multiplier;
          multiplier *= 128;
     while ((digit AND 128) != 0);
```

where AND is the bit-wise and operator (& in C).

When this algorithm terminates, value contains the Remaining Length in bytes.

Remaining Length encoding is not part of the variable header. The number of bytes used to encode the Remaining Length does not contribute to the value of the Remaining Length. The variable length ″extension bytes″ are part of the fixed header, not the variable header.

WebSphere MQ Telemetry Transport payload:

The following types of WebSphere MQ Telemetry Transport command message have a payload in the message header:

CONNECT
The payload contains one or three UTF-8 encoded strings. The first string uniquely identifies the client to the broker. The second string is the Will topic, and the third string is the Will message. The second and third strings are present only if the Will flag is set in the CONNECT Flags byte.

SUBSCRIBE
The payload contains a list of topic names to which the client can subscribe, and the QoS level. These strings are UTF-encoded.

SUBACK
The payload contains a list of granted QoS levels. These are the QoS levels at which the administrators for the broker have permitted the client to subscribe to a particular Topic Name. Granted QoS levels are listed in the same order as the topic names in the corresponding SUBSCRIBE message.

The payload part of a PUBLISH message contains application-specific data only. No assumptions are made about the nature or content of the data, and this part of the message is treated as a BLOB.

If you want an application to apply compression to the payload data, you need to define in the application the appropriate payload flag fields to handle the compression details. You cannot define application-specific flags in the fixed or variable headers.

WebSphere MQ Telemetry Transport variable header:

The message header for some types of WebSphere MQ Telemetry Transport command message contains a variable header. It resides between the fixed header and the payload.

The format of the variable header fields are described in the following topics, in the order in which they must appear in the header:

The variable length Remaining Length field is not part of the variable header. The bytes of the Remaining Length field do not contribute to the byte count of the Remaining Length value. This value only takes account of the variable header and the payload. See Fixed header for more information.

- Protocol name
- Protocol version
- Connect flags
- Keep Alive timer
- Connect return code
- Topic name
- Message identifier

WebSphere MQ Telemetry Transport protocol name:

The protocol name is present in the variable header of a WebSphere MQ Telemetry Transport CONNECT message. This field is a UTF-encoded string that represents the protocol name MQIsdp, capitalized as shown.

WebSphere MQ Telemetry Transport protocol version:

The protocol version is present in the variable header of a CONNECT message.

The field is an 8-bit unsigned value that represents the revision level of the protocol used by the client. The value of the Protocol version field for the current version of the protocol, 3 (0x03), is shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Protocol Version							
	0	0	0	0	0	0	1	1

WebSphere MQ Telemetry Transport connect flags:

The Clean start, Will, Will QoS, and Retain flags are present in the variable header of a CONNECT message.

Clean start flag:

Position: bit 1 of the Connect flags byte.

Returns the client to a known, ″clean″ state with the broker. If the flag is set, the broker discards any outstanding messages, deletes all subscriptions for the client, and resets the Message ID to 1. The client proceeds without the risk of any data from previous connections interfering with the current connection. The format of the Clean start flag is shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Reserved	Reserved	Will Retain	Will QoS		Will Flag	Clean Start	Reserved
	x	x	x	x	x	x		x

Bits 7, 6, and 0 of this byte are not used in the current version of the protocol. They are reserved for future use.

Will flag:

Position: bit 2 of the Connect flags byte.

The Will message defines that a message is published on behalf of the client by the broker when either an I/O error is encountered by the broker during communication with the client, or the client fails to communicate within the Keep Alive timer schedule. Sending a Will message is not triggered by the broker receiving a DISCONNECT message from the client.

If the Will flag is set, the Will QoS and Will Retain fields must be present in the Connect flags byte, and the Will Topic and Will Message fields must be present in the payload.

The format of the Will flag is shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Reserved	Reserved	Will Retain	Will QoS		Will Flag	Clean Start	Reserved
	x	x	x	x	x		x	x

Bits 7, 6, and 0 of this byte are not used in the current version of the protocol. They are reserved for future use.

Will QoS:

Position: bits 4 and 3 of the Connect flags byte.

A connecting client specifies the QoS level in the Will QoS field for a Will message that is sent in the event that the client is disconnected involuntarily. The Will message is defined in the payload of a CONNECT message.

If the Will flag is set, the Will QoS field is mandatory, otherwise its value is disregarded.

The value of Will QoS is 0 (0x00), 1 (0x01), or 2 (0x02). The Will QoS flag is shown in the table below.

bit	7	6	5	4	3	2	1	0
	Reserved	Reserved	Will Retain	Will QoS		Will Flag	Clean Start	Reserved
	x	x	x			1	x	x

Bits 7, 6, and 0 of this byte are not used in the current version of the protocol. They are reserved for future use.

Will Retain flag:

Position: bit 5 of the Connect flags byte.

The Will Retain flag indicates whether or not broker should retain the Will message which is published by the broker on behalf of the client in the event that the client is disconnected unexpectedly.

The Will Retain flag is mandatory if the Will flag is set, otherwise, it is disregarded. The format of the Will Retain flag is shown in the table below.

bit	7	6	5	4	3	2	1	0
	Reserved	Reserved	Will Retain	Will QoS		Will Flag	Clean Start	Reserved
	x	x		x	x	1	x	x

Bits 7, 6, and 0 of this byte are not used in the current version of the protocol. They are reserved for future use.

WebSphere MQ Telemetry Transport Keep Alive timer:

The Keep Alive timer is present in the variable header of a WebSphere MQ Telemetry Transport CONNECT message.

The Keep Alive timer, measured in seconds, defines the maximum time interval between messages received from a client. It enables the broker to detect that the network connection to a client has dropped, without having to wait for the long TCP/IP timeout. The client has a responsibility to send a message within each Keep Alive time period. In the absence of a data-related message during the time period, the client sends a PINGREQ message, which the broker acknowledges with a PINGRESP message.

If the broker does not receive a message from the client within one and a half times the Keep Alive time period (the client is allowed ″grace″ of half a time period), it disconnects the client as if the client had sent a DISCONNECT message. This action does not impact any of the client’s subscriptions. See “DISCONNECT Disconnect notification” on page 71 for more details.

The Keep Alive timer is a 16-bit value that represents the number of seconds for the time period. The actual value is application-specific, but a typical value is a few minutes. The maximum value is approximately 18 hours. A value of zero (0) means the client is not disconnected.

The format of the Keep Alive timer is shown in the table below. The ordering of the 2 bytes of the Keep Alive Timer is MSB, then LSB (big-endian).

bit	7	6	5	4	3	2	1	0
	Keep Alive MSB							
	Keep Alive LSB							

WebSphere MQ Telemetry Transport connect return code:

The connect return code is resent in the variable header of a WebSphere MQ Telemetry Transport CONNACK message.

This field defines a one byte unsigned return code. The meanings of the values, shown in the tables below, are specific to the message type. A return code of zero (0) usually indicates success.

Enumeration	HEX	Meaning
0	0x00	Connection Accepted
1	0x01	Connection Refused: unacceptable protocol version
2	0x02	Connection Refused: identifier rejected
3	0x03	Connection Refused: broker unavailable
4-255		Reserved for future use

bit	7	6	5	4	3	2	1	0
	Return Code							

WebSphere MQ Telemetry Transport topic name:

The topic name is present in the variable header of a WebSphere MQ Telemetry Transport PUBLISH message.

The topic name is the key that identifies the information channel to which payload data is published. Subscribers use the key to identify the information channels on which they want to receive published information.

The topic name is a UTF-encoded string. See “WebSphere MQ Telemetry Transport and UTF-8” on page 84 for more information. Topic name has an upper length limit of 32,767 characters.

WebSphere MQ Telemetry Transport message identifier:

The message identifier is present in the variable header of the following WebSphere MQ Telemetry Transport messages: PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE, UNSUBACK.

The Message Identifier (Message ID) field is only present in messages where the QoS bits in the fixed header indicate QoS levels 1 or 2. See Quality of Service levels and flows for more information.

The Message ID is a 16-bit unsigned integer. It typically increases by exactly one from one message to the next, but is not required to do so. This assumes that there are never more than 65,535 messages ″in flight″ between one particular client-broker pair at any time.

The ordering of the two bytes of the Message Identifier is MSB, then LSB (big-endian).

Do not use Message ID 0. It is reserved as an invalid Message ID.

bit	7	6	5	4	3	2	1	0
	Message Identifier MSB							
	Message Identifier LSB							

WebSphere MQ Telemetry Transport command messages

Follow the links below for details of the supported WebSphere MQ Telemetry Transport command messages.
- "CONNACK Acknowledge connection request"
- "CONNECT Client requests a connection to a broker" on page 69
- "DISCONNECT Disconnect notification" on page 71
- "PINGREQ PING request" on page 71
- "PINGRESP PING response" on page 72
- "PUBACK Publish acknowledgment" on page 73
- "PUBCOMP Assured publish complete (part 3)" on page 73
- "PUBLISH Publish message" on page 74
- "PUBREC Assured publish received (part 1)" on page 76
- "PUBREL Assured Publish Release (part 2)" on page 77
- "SUBACK Subscription acknowledgment" on page 78
- "SUBSCRIBE Subscribe to named topics" on page 80
- "UNSUBACK Unsubscribe acknowledgment" on page 82
- "UNSUBSCRIBE Unsubscribe from named topics" on page 83

CONNACK Acknowledge connection request:

The CONNACK message is the message sent by the broker in response to a CONNECT request from a client.

Fixed header:

The fixed header format is shown in the table below.

bit	7	6	5	4	3	2	1	0
byte 1	Message type (2)				DUP flag	QoS flags		RETAIN
	0	0	1	0	x	x	x	x
byte 2	Remaining Length (2)							
	0	0	0	0	0	0	1	0

The DUP, QoS and RETAIN flags are not used in the CONNACK message.

Variable header:

The variable header format is shown in the table below.

	Description	7	6	5	4	3	2	1	0
Topic Name Compression Response									

	Description	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Reserved values. Not used.	x	x	x	x	x	x	x	x
Connect Return Code									
byte 2	Return Code								

The values for the one byte unsigned Connect return code field are shown in the table below.

Enumeration	**HEX**	**Meaning**
0	0x00	Connection Accepted
1	0x01	Connection Refused: unacceptable protocol version
2	0x02	Connection Refused: identifier rejected
3	0x03	Connection Refused: broker unavailable
4-255		Reserved for future use

Return code 2 (identifier rejected) is sent if the unique client identifier is not between 1 and 23 characters in length.

Payload:

There is no payload.

CONNECT Client requests a connection to a broker:

When a TCP/IP socket connection is established between the client and the broker, a protocol level session is required. It is assumed that the direction of connection is client to broker, and that the client supports broker listener functionality.

Fixed header:

The fixed header format is shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (1)				DUP flag	QoS level		RETAIN
	0	0	0	1	x	x	x	x
byte 2	Remaining Length							

The DUP, QoS, and RETAIN flags are not used in the CONNECT message.

Remaining Length is the length of the variable header (12 bytes) and the length of the Payload. This can be a multibyte field.

Variable header:

An example of the format of the variable header is shown in the table below.

	Description	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Protocol Name									

	Description	7	6	5	4	3	2	1	0
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (6)	0	0	0	0	0	1	1	0
byte 3	′M′	0	1	0	0	1	1	0	1
byte 4	′Q′	0	1	0	1	0	0	0	1
byte 5	′I′	0	1	0	0	1	0	0	1
byte 6	′s′	0	1	1	1	0	0	1	1
byte 7	′d′	0	1	1	0	0	1	0	0
byte 8	′p′	0	1	1	1	0	0	0	0
Protocol Version Number									
byte 9	Version (3)	0	0	0	0	0	0	1	1
Connect Flags									
byte 10	Will RETAIN (0) Will QoS (01) Will flag (1) Clean Start (1)	x	x	0	0	1	1	1	x
Keep Alive timer									
byte 11	Keep Alive MSB (0)	0	0	0	0	0	0	0	0
byte 12	Keep Alive LSB (10)	0	0	0	0	1	0	1	0

Clean Start flag

Set (1).

Keep Alive timer

Set to 10 seconds (0x000A).

Will message

- Will flag is set (1)
- Will QoS field is 1
- Will RETAIN flag is clear (0)

Payload:

The payload of the CONNECT message contains one or three UTF-encoded strings. If the Will flag is set in the Connect flags byte in the variable header, the payload must contain all three UTF-encoded strings:

Client Identifier

The first UTF-encoded string. The Client Identifier (Client ID) is between 1 and 23 characters long, and uniquely identifies the client to the broker. It must be unique across all clients connecting to a single broker, and is the key in handling Message IDs messages with QoS levels 1 and 2. If the Client ID contains more than 23 characters, the broker responds to the CONNECT message with a CONNACK return code 2: Identifier Rejected.

Will Topic

The second UTF-encoded string. The Will Message is published to the Will Topic. The QoS level is defined by the Will QoS field, and the RETAIN status is defined by the Will RETAIN flag in the variable header.

Will Message

The third UTF-encoded string. The Will Message defines the content of the message that is published to the Will Topic if the client is unexpectedly disconnected.

Although the Will Message is UTF-encoded in the CONNECT message, when it is published to the Will Topic only the bytes of the message are sent, not the first two length bytes. The message sent when the broker executes the Will Message is raw ASCII, not UTF-encoded.

Response:

The broker sends a CONNACK message in response to a CONNECT message from a client.

If the client does not receive a CONNACK message from the broker within a ″reasonable″ amount of time, the client closes the TCP/IP socket connection, and restarts the session by opening a socket to the broker and issuing a CONNECT message. A ″reasonable″ amount of time depends on the type of application and the communications infrastructure.

DISCONNECT Disconnect notification:

The DISCONNECT message is sent from the client to the broker to indicate that it is about to close its TCP/IP connection. This allows for a clean disconnection, rather than just dropping the line.

Sending the DISCONNECT message does not affect existing subscriptions. They are persistent until they either explicitly unsubscribed, or if there is a clean start. The broker retains QoS 1 and QoS 1 messages for topics to which the client is unsubscribed until the client reconnects. QoS 0 messages are not retained, since they are delivered on a best efforts basis.

Fixed header:

The fixed header format is shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (14)				DUP flag	QoS level		RETAIN
	1	1	1	0	x	x	x	x
byte 2	Remaining Length (0)							
	0	0	0	0	0	0	0	0

The DUP, QoS, and RETAIN flags are not used in the DISCONNECT message.

Payload:

There is no payload.

Variable header:

There is no variable header.

PINGREQ PING request:

The PINGREQ message is an "are you alive" message that is sent from or received by a connected client.

Fixed header:

The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (12)				DUP flag	QoS level		RETAIN
	1	1	0	0	x	x	x	x
byte 2	Remaining Length (0)							
	0	0	0	0	0	0	0	0

The DUP, QoS, and RETAIN flags are not used.

Variable Header:

There is no variable header.

Payload:

There is no payload.

Response:

The response to a PINGREQ message is a PINGRESP message.

PINGRESP PING response:

A PINGRESP message is the response to a PINGREQ message and means "yes I am alive". Keep Alive messages flow in either direction, sent either by a connected client or the broker.

Fixed header:

The table below shows the fixed header format:

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (13)				DUP flag	QoS level		RETAIN
	1	1	0	1	x	x	x	x
byte 2	Remaining Length (0)							
	0	0	0	0	0	0	0	0

The DUP, QoS, and RETAIN flags are not used.

Payload:

There is no payload.

Variable header:

There is no variable header.

PUBACK Publish acknowledgment:

A PUBACK message is the response to a PUBLISH message with QoS level 1. A PUBACK message is sent by a broker in response to a PUBLISH message from a publishing client, and by a subscriber in response to a PUBLISH message from the broker.

Fixed header:

The table below shows the format of the fixed header.

bit	7	6	5	4	3	2	1	0
byte 1	Message Type (4)				DUP flag	QoS level		RETAIN
	0	1	0	0	x	x	x	x
byte 2	Remaining Length (2)							
	0	0	0	0	0	0	1	0

QoS level
Not used.

DUP flag
Not used.

RETAIN flag
Not used.

Remaining Length field
This is the length of the variable header (2 bytes). It can be a multibyte field.

Variable header:

Contains the Message Identifier (Message ID) for the PUBLISH message that is being acknowledged. The table below shows the format of the variable header.

bit	7	6	5	4	3	2	1	0
byte 1	Message ID MSB							
byte 2	Message ID LSB							

Payload:

There is no payload.

Actions:

When the client receives the PUBACK message, it discards the original message, because it is also received (and logged) by the broker.

PUBCOMP Assured publish complete (part 3):

This message is either the response from the broker to a PUBREL message from a publisher, or the response from a subscriber to a PUBREL message from the broker. It is the fourth and last message in the QoS 2 protocol flow.

Fixed header:

The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (7)				DUP flag	QoS level		RETAIN
	0	1	1	1	x	x	x	x
byte 2	Remaining Length (2)							
	0	0	0	0	0	0	1	0

QoS level
Not used.

DUP flag
Not used.

RETAIN flag
Not used.

Remaining Length field
The length of the variable header (2 bytes). It can be a multibyte field.

Variable header:

The variable header contains the same Message ID as the acknowledged PUBREL message.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message ID MSB							
byte 2	Message ID LSB							

Payload:

There is no payload.

Actions:

When the client receives a PUBCOMP message, it discards the original message because it has been delivered, exactly once, to the broker.

PUBLISH Publish message:

A PUBLISH message is sent by a client to a broker for distribution to interested subscribers. Each PUBLISH message is associated with a topic name (also known as the Subject or Channel). This is a hierarchical name space that defines a taxonomy of information sources for which subscribers can register an interest. A message that is published to a specific topic name is delivered to connected subscribers for that topic.

To maintain symmetry, if a client subscribes to one or more topics, any message published to those topics are sent by the broker to the client as a PUBLISH message.

Fixed header:

The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message type (3)				DUP flag	QoS level		RETAIN
	0	0	1	1	0	0	1	0
byte 2	Remaining Length							

QoS level
Set to 1.

DUP flag
Set to zero (0). This means that the message is being sent for the first time.

For messages with QoS level 1 or level 2 that are being re-sent because a failure has been detected, the DUP bit is set to 1. This indicates to the broker that the message might duplicate a message that has already been received. The significance of this information to the broker depends upon the QoS level. The DUP bit is not used for messages with QoS level 0.

RETAIN flag
Set to zero. This means do not retain.

Remaining Length field
The length of the variable header plus the length of the payload. It can be a multibyte field.

Variable header:

The variable header contains the following fields:

Topic name
A UTF-encoded string.

Message ID
Present for messages with QoS level 1 and QoS level 2.

Typically, the protocol library is responsible for generating the Message ID and passing it back to the publishing application, possibly as a return handle. This approach avoids the risk of multiple applications, or publishing threads, running on a single client generating duplicate Message IDs.

A Message ID must not be used in the variable header for messages with QoS level 0.

The Message ID is a 16-bit unsigned integer, which typically increases by exactly one from one message to the next, but is not required to do so. The ordering of the 2 bytes of the Message Identifier is MSB, then LSB (big-endian).

Message ID 0 (that is, 0x0000) is reserved as an invalid Message ID, and must not be used.

The table below shows an example variable header for a PUBLISH message.

Field	**Value**
Topic Name:	″a/b″
QoS level	1
Message ID:	10

The format of the variable header in this case is shown in the table below.

	Description	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Topic Name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (3)	0	0	0	0	0	0	1	1
byte 3	′a′ (0x61)	0	1	1	0	0	0	0	1
byte 4	′/′ (0x2F)	0	0	1	0	1	1	1	1
byte 5	′b′ (0x62)	0	1	1	0	0	0	1	0
Message Identifier									
byte 6	Message ID MSB (0)	0	0	0	0	0	0	0	0
byte 7	Message ID LSB (10)	0	0	0	0	1	0	1	0

Payload:

Contains the data for publishing. The content and format of the data is application specific. The Remaining Length field in the fixed header includes both the variable header length and the payload length.

Response:

The response to a PUBLISH message depends on the QoS level. The table below shows the expected responses.

QoS Level	**Expected response**
QoS 0	None
QoS 1	PUBACK
QoS 2	PUBREC

Actions:

PUBLISH messages can be sent either from a publisher to the broker, or from the broker to a subscriber. The action of the recipient when it receives a message depends on the QoS level of the message:

QoS 0 Make the message available to any interested parties.

QoS 1 Log the message to persistent storage, make it available to any interested parties, and return a PUBACK message to the sender.

QoS 2 Log the message to persistent storage, do not make it available to interested parties yet, and return a PUBREC message to the sender.

If the broker receives the message, interested parties means subscribers to the topic of the PUBLISH message. If a subscriber receives the message, interested parties means the application on the client which has subscribed to one or more topics, and is waiting for a message from the broker.

PUBREC Assured publish received (part 1):

A PUBREC message is the response to a PUBLISH message with QoS level 2. It is the second message of the QoS level 2 protocol flow. A PUBREC message is sent by the broker in response to a PUBLISH message from a publishing client, or by a subscriber in response to a PUBLISH message from the broker.

Fixed header:

The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (5)				DUP flag	QoS level		RETAIN
	0	1	0	1	x	x	x	x
byte 2	Remaining Length (2)							
	0	0	0	0	0	0	1	0

QoS level
Not used.

DUP flag
Not used.

RETAIN flag
Not used.

Remaining Length field
The length of the variable header (2 bytes). It can be a multibyte field.

Variable header:

The variable header contains the Message ID for the acknowledged PUBLISH. The table below shows the format of the variable header.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message ID MSB							
byte 2	Message ID LSB							

Payload:

There is no payload.

Actions:

When it receives a PUBREC message, the recipient sends a PUBREL message to the sender with the same Message ID as the PUBREC message.

PUBREL Assured Publish Release (part 2):

A PUBREL message is the response either from a publisher to a PUBREC message from the broker, or from the broker to a PUBREC message from a subscriber. It is the third message in the QoS 2 protocol flow.

Fixed header:

The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (6)				DUP flag	QoS level		RETAIN

bit	7	6	5	4	3	2	1	0
	0	1	1	0	x	x	x	x
byte 2	Remaining Length (2)							
	0	0	0	0	0	0	1	0

QoS level
Not used.

DUP flag
Not used.

RETAIN flag
Not used.

Remaining Length field
The length of the variable header (2 bytes). It can be a multibyte field.

Variable header:

The variable header contains the same Message ID as the PUBREC message that is being acknowledged. The table below shows the format of the variable header.

bit	7	6	5	4	3	2	1	0
byte 1	Message ID MSB							
byte 2	Message ID LSB							

Payload:

There is no payload.

Actions:

When the broker receives a PUBREL message from a publisher, the broker makes the original message available to interested subscribers, and sends a PUBCOMP message with the same Message ID to the publisher. When a subscriber receives a PUBREL message from the broker, the subscriber makes the message available to the subscribing application and sends a PUBCOMP message to the broker.

SUBACK Subscription acknowledgment:

A SUBACK message is sent by the broker to the client to confirm receipt of a SUBSCRIBE message.

A SUBACK message contains a list of granted QoS levels. These are the levels at which the administrators for the broker permit the client to subscribe to a specific topic name. In the current version of the protocol, the broker always grants the QoS level requested by the subscriber. The order of granted QoS levels in the SUBACK message matches the order of the topic Nnames in the corresponding SUBSCRIBE message.

Fixed header:

The table below shows the format of the fixed header.

bit	7	6	5	4	3	2	1	0
byte 1	Message Type (9)				DUP flag	QoS level		RETAIN
	1	0	0	1	x	x	x	x
byte 2	Remaining Length							

QoS level
Not used.

DUP flag
Not used.

RETAIN flag
Not used.

Remaining Length field
The length of the variable header. It can be a multibyte field.

Variable header:

The variable header contains the Message ID for the SUBSCRIBE message that is being acknowledged. The table below shows the format of the variable header.

bit	7	6	5	4	3	2	1	0
byte 1	Message ID MSB							
byte 2	Message ID LSB							

Payload:

The payload contains a vector of granted QoS levels. Each level corresponds to a topic name in the corresponding SUBSCRIBE message. The order of QoS levels in the SUBACK message matches the order of topic name and Requested QoS pairs in the SUBSCRIBE message. The Message ID in the variable header enables you to match SUBACK messages with the corresponding SUBSCRIBE messages.

The table below shows the Granted QoS field encoded in a byte.

bit	7	6	5	4	3	2	1	0
	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	QoS level	
	x	x	x	x	x	x		

The upper 6 bits of this byte are not used in the current version of the protocol. They are reserved for future use.

The table below shows an example payload.

Granted QoS	0
Granted QoS	2

The table below shows the format of this payload.

	Description	7	6	5	4	3	2	1	0
Requested QoS									

	Description	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Granted QoS (0)	x	x	x	x	x	x	0	0
Granted QoS									
byte 2	Granted QoS (2)	x	x	x	x	x	x	1	0

SUBSCRIBE Subscribe to named topics:

The SUBSCRIBE message allows a client to register an interest in one or more topic names with the broker. Messages published to these topics are delivered from the broker to the client as PUBLISH messages. The SUBSCRIBE message also specifies the QoS level at which the subscriber wants to receive published messages.

Fixed header:

The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (8)				DUP flag	QoS level		RETAIN
	1	0	0	0	0	0	1	x
byte 2	Remaining Length							

QoS level

SUBSCRIBE messages use QoS level 1 to acknowledge multiple subscription requests. The corresponding SUBACK message is identified by matching the Message ID. This also handles SUBSCRIBE messages retries in the same way as PUBLISH messages.

DUP flag

In this example the DUP flag is set to zero (0) to indicate that the message is being sent for the first time. If this message is being re-sent because a SUBACK message has not arrived after a specified timeout period, the DUP bit is set to indicate to the broker that it might be a duplicate of a message already received.

RETAIN flag

Not used.

Remaining Length field

The length of the payload. It can be a multibyte field.

Variable header:

The variable header contains a Message ID because a SUBSCRIBE message has a QoS level of 1.

Typically, the protocol library generates the Message ID, and passes it back to the publishing application, for example as a return handle. This prevents multiple applications, or multiple publishing threads, running on a single client from generating duplicate Message IDs.

Message ID 0 (0x0000) is reserved as an invalid Message ID, and must not be used. The Message ID is a 16-bit unsigned integer, which typically increases by exactly one from one message to the next, but is not required to do so. The two bytes of the Message ID are ordered as MSB, followed by LSB (big-endian).

The table below shows an example format for the variable header with a Message ID of 10.

	Description	7	6	5	4	3	2	1	0
Message Identifier									
byte 1	Message ID MSB (0)	0	0	0	0	0	0	0	0
byte 2	Message ID LSB (10)	0	0	0	0	1	0	1	0

Payload:

The payload of a SUBSCRIBE message contains a list of topic names to which the client wants to subscribe, and the QoS level at which the client wants to receive the messages. The strings are UTF-encoded, and the QoS level occupies 2 bits of a single byte. These topic/QoS pairs are packed contiguously as shown in the example payload in the table below.

Topic name	″a/b″
Requested QoS	1
Topic name	″c/d″
Requested QoS	2

Topic names in a SUBSCRIBE message are not compressed.

The format of the example payload is shown in the table below.

	Description	7	6	5	4	3	2	1	0
Topic name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (3)	0	0	0	0	0	0	1	1
byte 3	′a′ (0x61)	0	1	1	0	0	0	0	1
byte 4	′/′ (0x2F)	0	0	1	0	1	1	1	1
byte 5	′b′ (0x62)	0	1	1	0	0	0	1	0
Requested QoS									
byte 6	Requested QoS (1)	x	x	x	x	x	x	0	1
Topic Name									
byte 7	Length MSB (0)	0	0	0	0	0	0	0	0
byte 8	Length LSB (3)	0	0	0	0	0	0	1	1
byte 9	′c′ (0x63)	0	1	1	0	0	0	1	1
byte 10	′/′ (0x2F)	0	0	1	0	1	1	1	1
byte 11	′d′ (0x64)	0	1	1	0	0	1	0	0
Requested QoS									
byte 12	Requested QoS (2)	x	x	x	x	x	x	1	0

Assuming that the requested QoS level is granted, the client receives PUBLISH messages at less than or equal to this level, depending on the QoS level of the original message from the publisher. For example, if a client has a QoS level 1 subscription to a particular topic, then a QoS level 0 PUBLISH message to that topic is delivered to the client at QoS level 0. A QoS level 2 PUBLISH message to the same topic is downgraded to QoS level 1 for delivery to the client.

A corollary to this is that subscribing to a topic at QoS level 2 is equivalent to saying ″I would like to receive messages on this topic at the QoS at which they are published″.

The Requested QoS field is encoded in the byte following each UTF-encoded topic name as shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	QoS level	
	x	x	x	x	x	x		

The upper 6 bits of this byte are not used in the current version of the protocol. They are reserved for future use.

Response:

When it receives a SUBSCRIBE message from a client, the broker responds with a SUBACK message.

UNSUBACK Unsubscribe acknowledgment:

The UNSUBACK message is sent by the broker to the client to confirm receipt of an UNSUBSCRIBE message.

Fixed header:

The table below shows the fixed header format.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Type (11)				DUP flag	QoS level		RETAIN
	1	0	1	1	x	x	x	x
byte 2	Remaining length (2)							
	0	0	0	0	0	0	1	0

QoS level
Not used.

DUP flag
Not used.

RETAIN flag
Not used.

Remaining Length
The length of the Variable Header (2 bytes).

Variable header:

The variable header contains the Message ID for the UNSUBSCRIBE message that is being acknowledged. The table below shows the format of the variable header.

bit	7	6	5	4	3	2	1	0
byte 1	Message ID MSB							
byte 2	Message ID LSB							

Payload:

There is no payload.

UNSUBSCRIBE Unsubscribe from named topics:

An UNSUBSCRIBE message is sent by the client to the broker to unsubscribe from named topics.

Fixed header:

The table below shows an example fixed header format.

bit	7	6	5	4	3	2	1	0
byte 1	Message Type (10)				DUP flag	QoS level		RETAIN
	1	0	1	0	0	0	1	x
byte 2	Remaining Length							

QoS level
The level of QoS is 1 to acknowledge multiple unsubscribe requests. The corresponding UNSUBACK message is identified by the Message ID. Retries are handled in the same way as PUBLISH messages.

DUP flag
In this example the DUP flag is set to zero (0) to indicate that the message is being sent for the first time.

If this message is being re-sent because a SUBACK message has not arrived after a specified timeout period, the DUP bit is set to indicate to the broker that it might be a duplicate of a message already received.

RETAIN flag
Not used.

Remaining Length
This is the length of the Payload. It can be a multibyte field.

Variable header:

The variable header contains a Message ID because an UNSUBSCRIBE message has a QoS level of 1.

Typically, the protocol library generates the Message ID, and passes it back to the publishing application, for example as a return handle. This prevents multiple applications, or multiple publishing threads, running on a single client from generating duplicate Message IDs.

Message ID 0 (0x0000) is reserved as an invalid Message ID, and must not be used. The Message ID is a 16-bit unsigned integer, which typically increases by exactly one from one message to the next, but is not required to do so. The two bytes of the Message ID are ordered as MSB, followed by LSB (big-endian).

The table below shows an example format for the variable header with a Message ID of 10.

	Description	7	6	5	4	3	2	1	0
Message Identifier									
byte 1	Message ID MSB (0)	0	0	0	0	0	0	0	0
byte 2	Message ID LSB (10)	0	0	0	0	1	0	1	0

Payload:

The client unsubscribes from the list of topics named in the payload. The strings are UTF-encoded and are packed contiguously. Topic names in a UNSUBSCRIBE message are not compressed. The table below shows an example payload.

Topic Name	″a/b″
Topic Name	″c/d″

The table below shows the format of this payload.

	Description	7	6	5	4	3	2	1	0
Topic Name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (3)	0	0	0	0	0	0	1	1
byte 3	′a′ (0x61)	0	1	1	0	0	0	0	1
byte 4	′/′ (0x2F)	0	0	1	0	1	1	1	1
byte 5	′b′ (0x62)	0	1	1	0	0	0	1	0
Topic Name									
byte 6	Length MSB (0)	0	0	0	0	0	0	0	0
byte 7	Length LSB (3)	0	0	0	0	0	0	1	1
byte 8	′c′ (0x63)	0	1	1	0	0	0	1	1
byte 9	′/′ (0x2F)	0	0	1	0	1	1	1	1
byte 10	′d′ (0x64)	0	1	1	0	0	1	0	0

Response:

The broker sends an UNSUBACK to a client in response to an UNSUBSCRIBE message.

WebSphere MQ Telemetry Transport and UTF-8

UTF-8 is an efficient encoding of Unicode character-strings that optimizes the encoding of ASCII characters in support of text-based communications.

The WebSphere MQ Telemetry Transport protocol uses a subset of UTF-8. Only single byte (non-extended) characters are supported.

The UTF string format is shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Length MSB							
byte 2	Message Length LSB							
bytes 3 ...	Encoded Character Data							

Message Length is the number of bytes of encoded string characters, not the number of characters. For ASCII strings, however, these are the same. The format of encoded characters for ASCII codes 0x01 to 0x7F are shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0	ASCII code of character						

For example, the ASCII text string OTWP is encoded in UTF-8 as shown in the table below.

bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message Length MSB (0x00)							
	0	0	0	0	0	0	0	0
byte 2	Message Length LSB (0x04)							
	0	0	0	0	0	1	0	0
byte 3	’O’ (0x4F)							
	0	1	0	0	1	1	1	1
byte 4	’T’ (0x54)							
	0	1	0	1	0	1	0	0
byte 5	’W’ (0x57)							
	0	1	0	1	0	1	1	1
byte 6	’P’ (0x50)							
	0	1	0	1	0	0	0	0

The Java writeUTF() and readUTF() data stream methods use this format.

WebSphere MQ Web Services Transport

Clients that use the WebSphere MQ Web Services Transport define messages in XML and use the standardized HTTP protocol running over TCP/IP.

To use the WebSphere MQ Web Services Transport, you must deploy a message flow that contains an HTTPInput or HTTPRequest node to your broker. If this message flow sends output messages to other WebSphere MQ Web Services Transport applications, it must also include an HTTPReply node.

Applications that connect using WebSphere MQ Web Services Transport and HTTP use predominantly the point-to-point model.

The following built-in nodes are provided to support this protocol:
- HTTPInput
- HTTPReply
- HTTPRequest

WebSphere Broker JMS Transport

Use the WebSphere Broker JMS Transport to send and receive JMS messages. These messages must conform to the Java Message Service Specification, version 1.1.

To use the WebSphere Broker JMS Transport, you must deploy a message flow that contains a JMSInput node or a JMSOutput node. You can also include a JMSMQTransform node or an MQJMSTransform node.

The following topics contain information about JMS, JMS messages, and JMS messages in WebSphere Message Broker:

- "JMS message structure" on page 87
- "JMS message types" on page 88
- "Representation of messages across the JMS Transport" on page 90
- "JMS message as input" on page 91
- "JMS message for output" on page 96
- "JNDI administered objects" on page 97

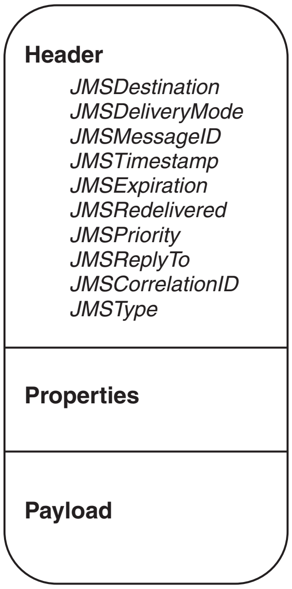
For information about the JMS transport, configurable properties for the JMS nodes, and troubleshooting, refer to the following topics:

- "JMS provision" on page 98
- "JMS brokering" on page 98
  - "JMS Transactionality" on page 98
  - "JMS message selector" on page 102
  - "JMS properties for application communication models" on page 103
  - "JMS message domain properties" on page 104
- "Troubleshooting JMS nodes" on page 106

The JMS Nodes sample sample is provided to help you start using the WebSphere Broker JMS Transport.

JMS message structure

The following figure depicts the JMS message structure:



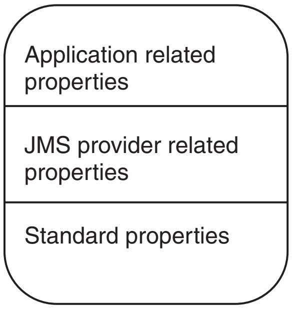
Header

A header must be present in every JMS message, and it is assigned automatically. Most of the values in the header are set by the JMS provider when the message is put to a JMS destination. Some values can be declared by the JMS client when it creates a JMS session, or when it creates the message consumer or producer; for example, *JMSDeliveryMode*, *JMSExpiration*, *JMSReplyTo*, and *JMSCorrelationID* are created when the JMS client creates a JMS session or creates the message consumer or producer.

The data elements of each header comprise name-value pairs and they can be any of the Java following types: Boolean, byte, short, char, long, int, float, double, string or byte[ ].

Properties

The properties are optional and can be divided into the following subsections:


Application related properties
JMS provider related properties
Standard properties

- *Application related properties*

A Java application can assign application related properties, and these are set before the message is delivered. The property names of the application are meaningful only to the sending and receiving applications.

- *Provider related properties*

  Every JMS provider can define proprietary properties that can be set either by the client or automatically by the provider. Provider related properties are prefixed with *JMS_* followed by the vendor name and the specific property name. For example, the WebSphere MQ JMS client sets the provider property to be *JMS_IBM_MsgType*.
- *Standard properties*

  These properties are set by the JMS provider when a message is sent. The JMS provider vendor can choose to not support any standard properties, to support some standard properties, or to support all standard properties. Standard property names start with *JMSX*; for example: *JMSXUserid* or *JMSXDeliveryCount*.

The properties are handled as name-value pairs and they can be any of the Java following types: Boolean, byte, short, char, long, int, float, double, string or byte[ ].

Payload

The payload type defines the JMS message. It can be one of the six JMS message types that are described in "JMS message types."

JMS does not define a wire format. The Java Message Service Specification, version 1.1 describes the physical representation of how a message is structured.

JMS message types

JMS defines six message interface types; a base message type and five subtypes. The message types are defined according to the type of the message *payload*, where the payload is the body of a message that holds the content. JMS specifies only the interface and does not specify the implementation. This allows for vendor specific implementation and transportation of messages while using a common interface.

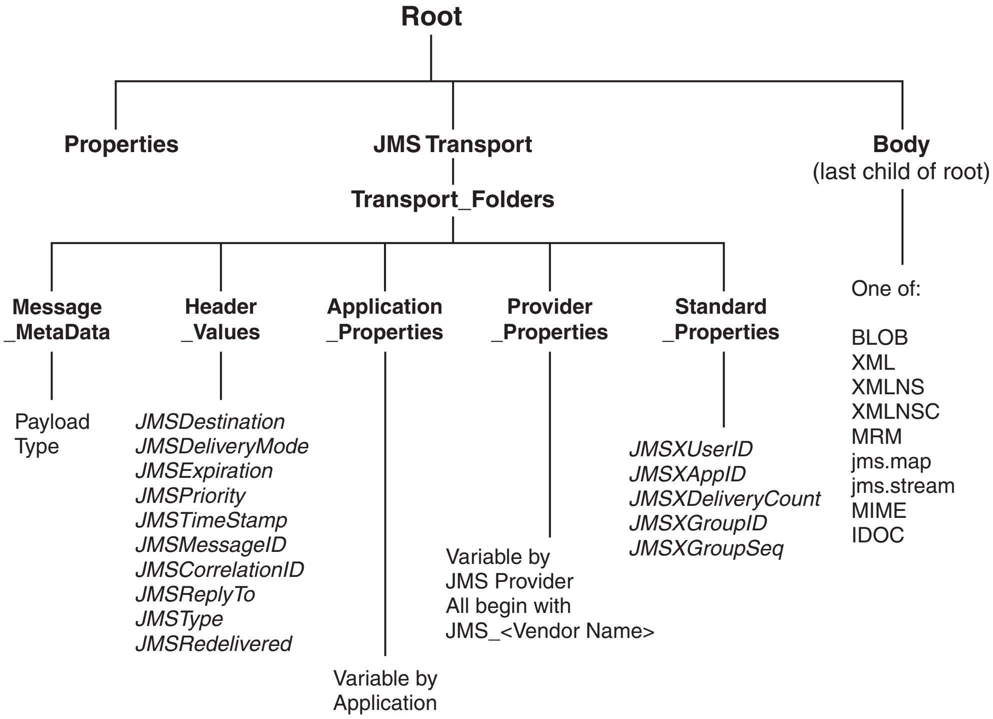
The following table describes the six message types:

Message type	Description
Message	The base class. This message type is used for event notification and does not have a payload.
BytesMessage	The payload is stored as an array of bytes. This message type is useful for exchanging data in an application's native format and when JMS is used as a transport between two systems, where the JMS client does not know the message payload type.
TextMessage	Data is stored as a string. This message type is useful for exchanging simple text messages and for more complex character data, such as XML documents.

StreamMessage	A Stream message is a sequence of primitive Java types. The message object keeps track of the order and the types of these primitives within the stream. Formal conversion rules apply; for example, an exception is thrown if a JMS application tries to read a double value as a short value. Refer to the Java Message Service Specification, version 1.1 for a full list of the conversion rules. `21ABCDEFGH32.345` is an example of a StreamMessage payload. It consists of the following three fields: • an Integer, `21` • a String, `ABCDEFGH` • a Float, `32.345` If the data structure is unknown, the generic method `readObject()` can be used to return the next object in the stream. If the structure of the data is known, the JMS client can be specific about the type of object being accessed.
MapMessage	The payload of a MapMessage is stored as a set of name-value pairs. The name is defined as a string and the value is typed. The MapMessage is useful for delivering keyed data that can change from one message to the next. `NumberOfCopies:5` is an example of a MapMessage payload, where `NumberOfCopies` is the key and `5` is the value. Data can be accessed by using `getMapNames()`, which returns a Java Enumeration object. It is possible to iterate through the MapMessage by using `hasMoreElements()` to retrieve the mapped name-value pairs.
ObjectMessage	The Object message carries a serializable Java Object as its payload. It is useful for exchanging Java objects.

Representation of messages across the JMS Transport

The following figure depicts the JMS message tree that is used by the JMSInput and JMSOutput nodes:



JMSTransport

- *Header_Values* subfolder:

  This subfolder is mandatory and is always created.

  JMS message properties are optional. If they are present in input messages, they are stored in the appropriate property subfolder.
- Properties subfolders:

  JMS message properties are optional. If present in the message, they are stored in the appropriate properties subfolder.
- *Message_MetaData* subfolder:

  This subfolder is included in order to preserve the payload type of the JMS message. It is used by the JMSOutput node when creating a JMS message. The payload type can be one of the following values:

Message type	Payload values
Base JMS message with no payload	jms_none
TextMessage	jms_text
BytesMessage	jms_byte
MapMessage	jms_map
StreamMessage	jms_stream
ObjectMessage	jms_object

Body

The message payload is stored in the body folder, which is the last child of Root. The payload is transferred by using one of the following message domain parsers:

- XML
- XMLNS
- XMLNSC
- BLOB
- JMSMap
- JMSStream
- MRM
- MIME
- IDOC

JMS message as input

The JMS message is a Java object and therefore it is not possible to parse the message as a bit stream. When the message is received, the header data, property data, and payload data are extracted by using the JMS API. For information on the header, properties, and payload of JMS messages, refer to “JMS message structure” on page 87.

The following topics describe how the different parts of the JMS message are obtained, and how the message is parsed:

- “JMS input message header and property data”
- “JMS Message payload” on page 93
- “JMS message payload and appropriate parser” on page 94
- “Order of precedence for deriving the message domain” on page 95

JMS input message header and property data:
Header data

This section describes how the JMSInput node obtains header and property data from JMS messages.

The JMSInput node extracts header data from messages by using JMS API methods. Header data is stored as name-value pairs in the *Header_Values* folder. The API methods return the value; for example, to get the value for the header field *JMSTimestamp*, the JMSInput node uses the `getJMSTimestamp( )` method. A similar method is provided for each of the following fixed header fields:

- JMSDestination
- JMSDeliveryMode
- JMSExpiration
- JMSPriority
- JMSTimeStamp
- JMSMessageID
- JMSCorrelationID
- JMSReplyTo
- JMSType
- JMSRedelivered

Property data

In a similar way to how the header data is obtained, the JMSInput node extracts property data from messages by using JMS API methods. Property data is stored as name-value pairs in the properties folders. The API method returns a value for every property name with which it is supplied.

XML representation of header and property data

The JMSInput node uses the header and property data to create an XML representation of the JMSTransport folders. The node passes the XML data to the JMSTransport parser as a byte array. The byte array is then used to populate or to refresh the elements in the message tree. The JMSTransport parser is a new parser type.

Preservation of Java type

A scheme is not required to preserve knowledge of the Java type because the header value Java types are fixed and known. The JMS message properties are optional, therefore a scheme is required to preserve the Java type of the property values. The scheme used is that which is implemented by the WebSphere MQ JMS client and the Real-timeInput node.

Java type information is represented as a metadata in the form of a keyword `dt='DataType'`where *Datatype* is a string. The Java type is passed in the XML as part of the element name `<ElementName dt='DataType'>Value</ElementName>`. *Datatype* can be any of the following values:

Datatype value	Definition
String	Any sequence of characters, excluding `<` and `&`
Boolean	The character `0` or `1`, where `1` is equal to "true"
bin.hex	Hexadecimal digits representing octets
I1	A number, expressed using the digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -128 to 127 inclusive.
I2	A number, expressed using the digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -32768 to 32767 inclusive.
I4	A number, expressed using the digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -2147483648 to 2147483647 inclusive.
I8	A number, expressed using the digits 0..9, with optional sign (no fractions or exponent). Must lie in the range -9223372036854775808 to 92233720368547750807 inclusive.
int	A number, expressed using the digits 0..9, with optional sign (no fractions or exponent). Must lie in the same range as the datatype value`I8`. This number can be used in place of one of the `I*` types if the sender does not want to associate a particular precision with the property.
R4	A floating point number, expressed using the digits 0..9, optional sign, optional fractional digits, optional exponent. Magnitude <= 3.40282347E+38, and >= 1.175E-37

R8	A floating point number, expressed using the digits 0..9, optional sign, optional fractional digits, optional exponent. Magnitude <= 1.7976931348623E+308, and >= 2.225E-307

JMS Message payload:

The payload for some of the JMS message types can be extracted as a whole from the message object by using the JMS API. The payload is passed as a bit stream to a broker parser. This is true for the following message types:

- BytesMessage
- TextMessage
- ObjectMessage

  Additional processing is required to deal with the ObjectMessage payload because the JMS ObjectMessage payload is a serialized Java Object.

  The JMSInput node obtains the payload by calling `getObject( )` on the message. `getObject( )` returns a de-serialized object of the original class. This class definition must be made available to the JMSInput node, and you should ensure that it is accessible through the broker's Java class path. (The class path is defined in the mqsiprofile batch file, which is in the broker's executable directory; for example, on Windows, this is mqsiprofile.cmd in the *install_dir*/bin directory.) The JMSInput node invokes the BLOB parser, which creates the message body by using a bit stream that is created from the object.

  The Java Object can be subsequently re-serialized in a JavaCompute Node or a user-defined extension, and is updated by using its method calls.

The payload for MapMessage and StreamMessage can be extracted only as individual elements and must be reformatted by the JMSInput node before it can be used to create the message body.

- **MapMessage payload**

  The JMSMap domain is a synonym for the broker XML parser, which expects a stream of XML data. MapMessage payload data however, is extracted as sets of name-value pairs from the message object. The JMS API is used to obtain the name-value pairs.

  The JMSInput node appends each name-value pair to a bit stream as an XML element and value, and preserves the type of the value by using the `dt=` attribute.

  The following example shows the XML that is generated by the JMSInput node for the MapMessage payload:

```
<map>
    <Item_8_of_10_Char dt='char'>A</Item_8_of_10_Char>
     <Item_5_of_10_Double dt='r8'>999999.0</Item_5_of_10_Double>
     <Item_10_of_10_String>Last Map Item</Item_10_of_10_String>
     <Item_9_of_10_Boolean dt='boolean'>0</Item_9_of_10_Boolean>
     <Item_2_of_10_Integrer dt='i4'>999</Item_2_of_10_Integrer>
     <Item_3_of_10_Short dt='i2'>9999</Item_3_of_10_Short>
     <Item_7_of_10_Byte dt='i1'>9</Item_7_of_10_Byte>
     <Item_6_of_10_Float dt='r4'>2.24</Item_6_of_10_Float>
     <Item_1_of_10_String>P2P Map Msg Number:1</Item_1_of_10_String>
     <Item_4_of_10_Long dt='i8'>99999</Item_4_of_10_Long>
          </map>
```

  In this example, the message contains 10 fields. The field names have been generated by a JMS Client application, and take the form `item_n_of_x_t`, where:

- n is the sequence number in which the item was added to the message,
- x is the total number of items in the map,
- t is the type of the value.

The map data is not returned from the JMS API the order in which it was received.

- **StreamMessage payload**

  The StreamMessage payload data is a sequence of fields, where each field has a specific type. The fields do not have associated names and so a default element name elt is used to generate the XML elements. Similar to the MapMessage, the JMS API allows for fields only to be retrieved individually. The JMSInput node extracts fields from the JMS message and appends each to a bit stream in XML format.

  The following is an example of the XML that is generated by the JMSInput node for the StreamMessage payload:

```
<stream>
     <elt>P2P Stream Message  Number :7</elt>
     <elt dt='i4'>999</elt>
     <elt dt='i2'>9999</elt>
     <elt dt='i8'>99999</elt>
     <elt dt='r8'>999999.0</elt>
     <elt dt='r4'>2.24</elt>
     <elt dt='i1'>9</elt>
     <elt dt='char'>A</elt>
     <elt dt='boolean'>0</elt>
     <elt>Last Stream Item</elt>
</stream>
```

  In this example, 10 typed values are added to the StreamMessage by a JMS client application.

JMS message payload and appropriate parser:

When the JMSInput node creates a message body from the JMS message payload, the appropriate parser for that payload is determined. Therefore, the JMSInput node must know the type of JMS message that it expects to receive. A JMS message is defined by the payload type, and the JMSInput node extracts the payload from the JMS message by using the JMS API.

The following JMSInput node properties allow you to specify the type of JMS message that the node expects to receive:

- Message Domain
- Message Set
- Message Type
- Message Format

The Message Domain can be set to one of the following values:

Domain	Usage

blank	This corresponds to the blank domain in an MQInput node. The node derives the Message Domain from the JMSType header field providing that the value conforms to a proprietary URI format, see "Order of precedence for deriving the message domain." If the JMSType value is blank or does not match this URI format, the node sets the Message Domain according to the JMS Message Java Class type. Refer to "Order of precedence for deriving the message domain" for more information.
BLOB	The node expects bit stream data from ByteMessage, ObjectMessage, or TextMessage.
XML	The node expects a TextMessage with an XML payload.
XMLNS	The node expects a TextMessage with an XMLNS payload.
XMLNSC	The node expects a TextMessage with an XMLNSC payload.
MRM	The node expects to receive a TextMessage or ByteMessage. If the message set, type, and format are not supplied in the JMSInput node then the JMSType header field must be set.
JMSMap	The node expects to receive a MapMessage only.
JMSStream	The node expects to receive a StreamMessage only.
MIME	The node expects a TextMessage or ByteMessage with a MIME (Multipurpose Internet Mail Extension).
IDOC	The node expects a TextMessage or ByteMessage with an IDOC payload.

Order of precedence for deriving the message domain:

When a JMS Message is received by the JMSInput node, the message domain is derived according to the following criteria and in the following order of precedence:

1. The Message Domain property is set to a specific domain type.

   In this case, the node expects to receive only the following JMS message types:

Message domain	**Valid JMS message types**				
	BytesMessage	TextMessage	MapMessage	StreamMessage	ObjectMessage
BLOB	X	X			X
XML		X			
XMLNS		X			
XMLNSC		X			
MRM	X	X			
JMSMap			X		
JMSStream				X	
MIME	X	X			
IDOC	X	X			

   If a JMS Message type is received, which is not valid for the Message Domain that is configured in the JMSInput node, the node issues a warning and backs out the message either to the source JMS provider destination, or to the backout destination.

2. The Message Domain property is *blank* (default). The JMSType header value from the JMS input Message is set according to the URI format shown below:

JMSType	Broker domain
mcd://domain/[set]/[type]/[?format=fmt]	MRM
mcd://XML or mcd://xml	XML
mcd://XMLNS or mcd://xmlns	XMLNS
mcd://XMLNSC or mcd://xmlnsc	XMLNSC
mcd://IDOC or mcd://idoc	IDOC
mcd://MIME or mcd://mime	MIME

   If a JMS Message type is received, which is not valid for the Message Domain configured in the JMSType header, the node issues a warning and backs out the message either to the source JMS provider destination, or to the backout destination.

   Messages received in the MRM domain

   Messages that are received in the MRM domain must have a JMSType header field that is set in accordance with the following format (which is also used in JMS provision):

   ```
   mcd://domain/[set]/[type]/[?format=fmt]
   ```

   If the JMSType field does not conform to this format, the message is handled in the BLOB domain. For example,

   ```
   mcd://MRM/SWIFTXML2005/{http://SWIFT/2005}:Document/?format=SWIFT
   ```

3. The Message Domain property is *blank* (default) and the JMSType header value from the JMS input Message is also blank.

   The message domain is set according to the JMS Message Java Class as follows:

TextMessage	XML
BytesMessage	BLOB
MapMessage	JMSMap
StreamMessage	JMSStream
ObjectMessage	BLOB

JMS message for output

When the JMSOutput node receives a JMS message, it invokes the JMSTransport parser to return an XML bit stream containing the JMSTransport section of the message. The node extracts the *Message_MetaData* and obtains the payload type information to identify which JMS message type to create for output. If the *Message_MetaData* folder is not present, the output node creates a BytesMessage by default.

Header data

The JMSOutput node extracts the JMS header data from the XML string and uses this data to populate the values for the JMS header fields in the message.

Property data

The JMSOutput node extracts the property values from the XML string. The XML elements contain type information that identifies which Java Object type to create for each property value.

Message payload

The message payload is obtained from the JMS message as a bit stream. For TextMessage and BytesMessage payloads, the bit stream can be passed to the JMS API directly to create the appropriate payload.

For MapMessage and StreamMessage payloads, the individual elements must be extracted from the XML bit stream. The output node calls the appropriate JMS API method to create the map or stream fields in the message.

For an ObjectMessage payload, the JMSOutput node re-serializes the bit stream payload by using the object class. The object class must be available in the broker's Java class path. The class path is defined in the mqsiprofile batch file, which is in the broker's executable directory; for example, on Windows, this is mqsiprofile.cmd in the *install_dir*/bin directory.

Message publication

The message is published to the JMS destination that has been specified as an attribute of the JMSOutput node. However, if the *JMSReplyTo* header field is set in the JMS message, the JMSOutput node treats the message as a reply to a previous request, and publishes the message to the JMS destination of the previous request.

JNDI administered objects

JNDI (Java Naming and Directory Interface) is a standard Java extension that provides a uniform API for accessing a variety of directory and naming services.

JMS clients use JNDI to browse a naming service in order to obtain references to administered objects. *Administered objects* are JMS connection factory and JMS destination objects, where JMS destination objects are topics and queues. Administered objects are created and configured by a system administrator.

To create and configure JNDI administered objects, refer to the JMS provider documentation. If you are using the WebSphere MQ JMS provider, see the sample JMSAdmin definitions file that is included with WebSphere MQ and refer to the *WebSphere MQ Using Java* book.

Location of JNDI administered objects

JNDI administered objects are stored in the bindings. This can be either file system based or LDAP based. LDAP (Lightweight Directory Access Protocol) is a software protocol that enables anyone to locate organizations, individuals, and other resources; for example, locating files and devices in a network, either on the public Internet or on a corporate intranet.

LDAP is part of X.500, which is a standard for directory services in a network.

Naming service

A naming service associates names with distributed objects so that the administered objects are located by using names and not complex network addresses. JNDI provides an abstraction that hides the specifics of the naming service, which makes client applications more portable.

A JMS client specifies a *JNDI InitialContext* to obtain a JNDI connection to the JMS messaging server. The InitialContext is the starting point in any JNDI lookup and acts like the root of a file system. The JMS directory service that is being used determines the properties that are used to create an InitialContext.

JMS provision

For information about JMS provision in WebSphere Message Broker, refer to the WebSphere MQ Real-time Transport, and the Real-timeInput, Real-timeOptimizedFlow, and Publication nodes.

- "Support for JMS messages" on page 16
- Real-timeInput node
- Real-timeOptimizedFlow node
- Publication node
- "WebSphere MQ Real-time Transport" on page 11

JMS brokering

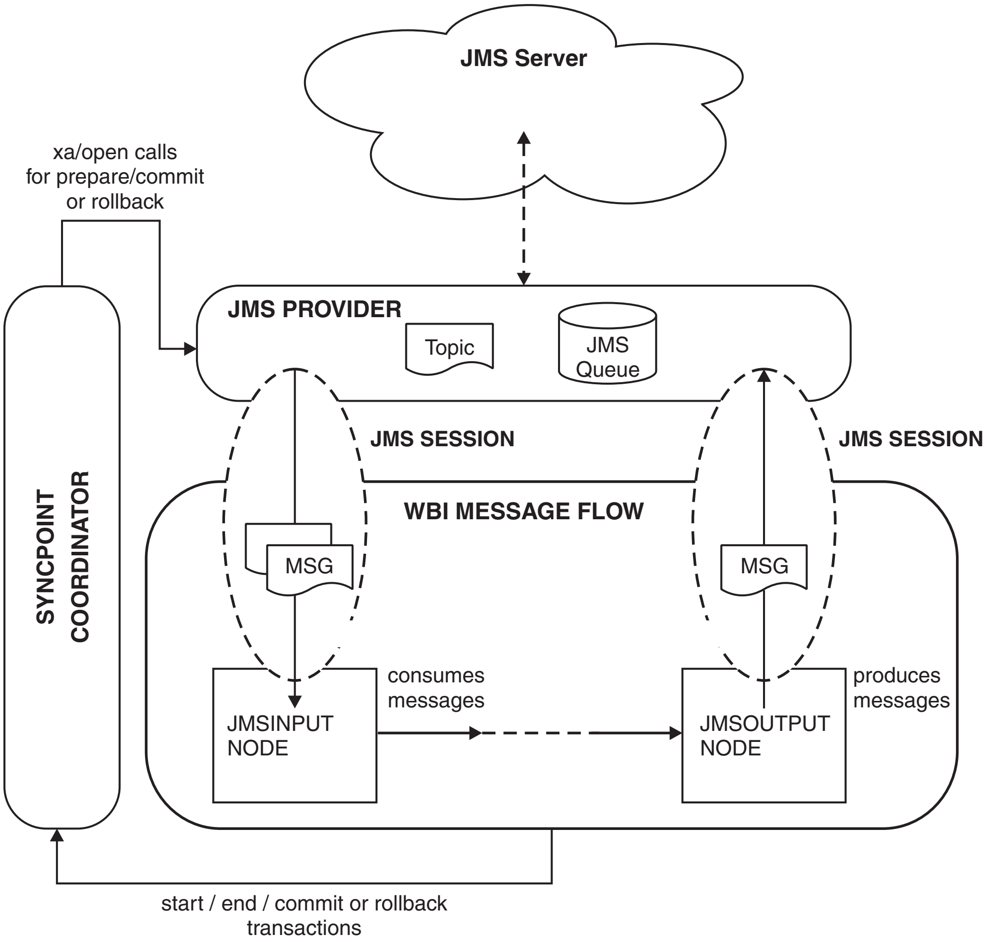
This section contains information to help you configure the properties on the JMSInput and JMSOutput nodes.

The two transformation nodes, JMSMQTransform and MQJMSTransform do not have configurable properties.

- "JMS Transactionality"
- "JMS message selector" on page 102
- "JMS properties for application communication models" on page 103
- "JMS message domain properties" on page 104

JMS Transactionality: JMS destinations that supply messages to a JMSInput node, or receive messages from a JMSOutput node, can be syncpoint coordinated as part of a message flow global transaction.

Transactions involving the Syncpoint coordinator



In this diagram, messages are consumed from a topic by a JMSInput node, and are produced to a JMS queue JMSOutput node. The nodes are connected with and are in session with a JMS provider. Any message flow input node can inform the external Syncpoint Coordinator when a message flow transaction starts and ends, and whether any resources that have been touched by the flow should be committed or rolled back.

The Syncpoint Coordinator sends XA/Open compliant requests to all participating Resource Managers to inform them to prepare. Any changes are then either committed or rolled back. Resource Managers such as WebSphere MQ, DB2 and any XA compliant JMS provider can participate in a global transaction. The external Syncpoint Coordinator is WebSphere MQ on distributed platforms, and RRS (Resource Recovery Services) on z/OS.

The JMSInput node and JMSOutput node can participate in a global transaction only if the JMS provider to which they connect supports the XA/Open interface through the JMS XAResource Class. An example JMS provider is the WebSphere MQ Java Client.

In-doubt transactions

In-doubt transactions can occur when a Resource Manager does not reply to a call from the Syncpoint manager, where the call is to commit or to rollback resources.

During start up of the broker's WebSphere MQ queue manager, an initial recovery step is taken to ensure that any in-doubt transactions are resolved before the broker message flows start to process new input. A JMS provider that participates in broker global transactions is included in this recovery step.

Configuration to enable global transaction support

Additional configuration is required to enable global transaction support for the JMSInput and JMSOutput node. You must complete the following steps:

1. Set the Message Flow property *Coordinated Transaction* to *yes*.
2. For each JMSInput or JMSOutput node that is required to participate in the global transaction, set the Advanced property *Transaction Mode* to *global*.
3. Create a queue connection factory and supply either a default name, *recoverXAQCF* or supply a user defined name. Refer to the JMSInput or JMSOutput node for further details on creating JNDI administered objects.
4. On distributed platforms, an WebSphere MQ administration task is required prior to deployment. This task is needed to register a broker component with the queue manager. The component, which is referred to a Switch file, is a shared library (a DLL on Windows).

   When the broker's WebSphere MQ queue manager starts up, it loads the Switch file. The Switch file forwards XA/Open transaction calls from the Syncpoint Coordinator to the JMS Provider. This ensures that the JMS resources that participate in the transaction can be coordinated in synchronization with other Resource managers that are involved in the same transaction.

   The task varies by platform.

   - **Linux and UNIX**

     For the broker's queue manager for each JMS provider that can be used by a JMSInput node, place a stanza entry in an initialization file, for example qm.ini. The following is an example of a stanza entry that can be added when you are using WebSphere MQ Java as the JMS provider:

     ```
     XAResourceManager:
      Name=WBIWMQJMS
         SwitchFile=/<Installation Path>/lib/JMSSwitch.so
         XAOpenString=<Initial Context Factory>,
                   <location of JNDI bindings>'
                   <LDAP Principal>,
                   <LDAP Credentials>,
                   <Recovery Connection Factory Name>
         ThreadOfControl=THREAD
     ```

     Where:

     `<Installation Path>` is the location of the WebSphere Message Broker installation. This value is mandatory.

     The parameters that are supplied on the `XAOpenString` are comma delimited and positional. Any missing optional parameter must be represented by a comma if other parameters are provided later in the string.

     – `<Initial Context Factory>` This is the Initial Context Factory identifier for the JMS provider. This value is mandatory.

     – `<Location of JNDI bindings>` This is either the file path to the bindings file, or the LDAP directory location of the JNDI administered objects that can be used to create an initial context factory for the JMS connection. When supplying the file path to the bindings file, do not include the file name. Refer to the JMSInput or JMSOutput node for further details on creating the JNDI administered objects. This value is mandatory.

- <LDAP Principal> This is an optional parameter that is used to specify the principal (user ID) that might be required when an LDAP database is used to hold the JNDI administered objects.
- <LDAP Credentials> This is an optional parameter that is used to specify the Credentials (password) that might be required if a password protected LDAP database is used to hold the JNDI administered objects.
- <Recovery Connection Factory Name> This is an optional parameter that is used to specify the name of a Queue Connection Factory object in the JNDI administered objects for recovery purposes, when the non default name is required.

You must specify a stanza in the broker's queue manager .ini file for each JMS provider that you want to use, that is, there should be one stanza for each new JMS provider, where the JMS provider can be specified by any JMSInput or JMSOutput node included in a message flow that is running on a broker.

The values for the Initial Context factory and Location of JNDI bindings in the stanza must match those specified in the JMSInput or JMSOutput nodes in the message flows.

Any LDAP parameters must match those which have been specified by using the mqsicreatebroker or mqsichangebroker command.

The Recovery Factory Name must match a Queue Connection Factory name that is created in the JNDI administered objects. If this is omitted, a default factory called recoverXAQCF is used. In either case this value must refer to a JNDI administered object that has already been created.

The following is an example stanza:

```
XAOpenString=com.sun.jndi.fscontext.RefFSContextFactory,
     /u/myJndiFileLocation,
     ,
     ,
     myRecoveryQCFName
```

Where the LDAP parameters are omitted, but a user defined Queue Connection Factory is specified for recovery.

- **On Windows platforms**

  As with Linux and Unix, the same information is required on Windows but it is configured using the WebSphere MQ Explorer or WebSphere MQ Services snap-in depending on which version of WebSphere MQ you are using. On Windows, the Switch file is called JMSSwitch.dll. Refer to the WebSphere MQ System Administration Guide for details on how to update the qm.ini file. The extra entry, called the XACloseString, should match the values provided for the XAOpenString.

- **On z/OS**

  In WebSphere Message Broker the only JMS provider currently supported is the IBM WebSphere MQ Java Client. The only transport mode currently supported for the client is BIND mode. No further configuration steps are required

For further information, see the section on Configuring for coordinated transactions section within the JMSInput node and JMSOutput node topics.

5. The JMS provider can supply additional jar files that are required for transactional support. Refer to the JMS provider documentation for more information. For instance, on distributed platforms (not z/OS), the WebSphere MQ JMS provider supplies an extra jar file com.ibm,mqetclient.jar. This jar must also be added to the broker shared_classes directory. On Windows, this

directory is C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes. For more information, refer to the section on making the JMS provider client available to the JMS Nodes in the following topics: JMSInput node.

Choice of JMS Provider

Any JMS provider that conforms to the Java Message Service Specification, version 1.1 and that supports the JMS XAResource API through the JMS session can be used if transaction coordination is required.

If the message designer has specified a non XA compliant provider, the non transactional mode only is supported. In this case, you must set the *Transaction mode* property to *no* for all JMSInput and JMSOutput nodes.

JMS message selector:

A message selector allows a JMS consumer to be more selective about the messages that it receives from a particular topic or queue.

A message selector uses message properties and headers as criteria in conditional expressions. These expressions use boolean logic to declare which messages should be delivered to a client, such as the JMSInput node.

The following table demonstrates the construction of a message selector. It comprises an identifier, such as the *JMSPriority* header, or an application controlled property *myProperty1*. The selector string must specify an operator followed by a literal.

Element	Valid values
Identifiers	• Property or header field reference (such as *JMSPriority*, *myProperty1*) • The following values are not possible: NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, IS
Operators	AND, OR, LIKE, BETWEEN, =, <>, <, >, <=, >=, IS NULL, IS NOT NULL
Literals	• The two boolean literals, TRUE and FALSE • Exact number literals that have no decimal point; for example, +25, -399, 40 • Approximate number literals. These can use scientific notation or decimal; for example, -21.4E4, 5E2, +34.4928

The JMSInput node provides a free format string *PropertySelector*, to specify selectors that filter or include application properties. The node also has attributes for specific header properties, where the identifier is implicit and is generated by the node. For the header selectors, the operator and literal part of the string must be specified.

If more than one selector is specified the node generates a composite selector string, where the individual selector strings are concatenated with the `AND` operator, and each selector string part is wrapped with parentheses.

The following are examples for each of the selector attributes:

Selector attribute	**Description**
PropertySelector	`OrderValue > 100.00` This string is used directly as shown.
TimeStamp	`BETWEEN 1057576423231 AND  10575788993265` Messages that are put between these two Java Times only (where Java time is milliseconds since 01 Jan 1970) is delivered to the JMSInput node. In this case the string generated is prefixed with the identifier *JMSTimestamp*.
Delivery Mode	`PERSISTENT` This means that only messages marked by the sender as being `PERSISTENT` should be delivered to the JMSInput node. In this case the string that is generated is prefixed with the identifier *JMSDeliveryMode*.
Priority	`>= 5 AND <= 8` This means that only messages marked by the sender as having a priority `5`, `6`, `7`, or `8` should be delivered to the JMSInput node. In this case the string generated is prefixed with the identifier *JMSPriority*.
Message ID	`> WMBRK123456` This will return messages whose Message ID is greater than the value specified. In this case the string generated is prefixed with the identifier *JMSMessageID*.
Redelivered	`FALSE` This means that messages that have not been redelivered should be received by the node. In this case the string generated is prefixed with the identifier *JMSRedlivered*.
Correlation ID	`= WMBRKABCDEFG` This returns messages whose Correlation ID is equal to the value `WMBRKBABCDEFG`. In this case, the string generated is prefixed with the identifier *JMSCorrelationID*.

JMS properties for application communication models:

JMS clients can operate with both publish/subscribe and point-to-point messages. The publish/subscribe and point-to-point application communication models use virtual channels called destinations. In the publish/subscribe model, the destinations are topics. For the point-to-point model, the destinations are known as queues.

The following application communication model properties can be configured for JMSInput and JMSOutput nodes:

Property	**Description**

Connection Factory Name	A string name that is passed to JNDI to look up the administered connection factory object. The connection factory object is used to create a connection to the JMS destination. • For a client operating as a publish/subscribe client, the connection factory name will be for a TopicConnectionFactory. • For a client operating as a point-to-point client, the connection factory name will be for a QueueConnectionFactory.
Subscription Topic	The string name that is passed to JNDI to look up the JMS topic destination. The topic is used to create a JMS session when the node is being used to process publish/subscribe messages.
Durable Subscription ID	This is a JMSInput node property only. It is a string identifier that is specified if the node is to subscribe to a durable subscription topic. A durable subscription is one that outlasts the client's connection to a message server. When a durable subscriber is disconnected from the server, it is the responsibility of the server to store messages that are published. Therefore, when the durable subscriber reconnects, the message server sends all the unexpired messages. Durable subscriptions cannot be unsubscribed from a message flow. It requires a separate administration task to unsubscribe a previously registered durable subscription. Some JMS providers supply an administration tool to perform this action.
Source Queue	The string name that will be passed to JNDI to look up the JMS queue destination. The queue is used to create a JMS session when the node is being used to process point-to-point messages.

The Subscription Topic and Source Queue properties are mutually exclusive because they configure the node to work with either the Publish/Subscribe message model or the Point-To-Point message model.

ADurable subscription ID is not valid without a Subscriber Topic property.

JMS message domain properties:

The JMSInput node can receive all of the JMS message types that are specified in the Java Message Service Specification, version 1.1. For more information see "JMS message types" on page 88.

The following JMSInput node properties can be specified:

Property	Description

Message Domain	• ***blank*** The messages to be received are for either JMS TextMessage or BytesMessage. The JMS message header field is set according to the format: `mcd://domain/[set]/[type]/[?format=fmt]` Valid JMS Message types for each domain are described in “JMS message types” on page 88. • **BLOB** Messages in this domain are processed by the BLOB parser. The BLOB parser is a program that interprets a bit stream or message tree that represents a message that belongs to the BLOB domain, and generates the corresponding tree from the bit stream on input, or a bit stream from the tree on output. • **XML**, **XMLNS**, and **XMLNSC** For messages conforming to the W3C XML standard. See The XML domains. • **MIME** For messages conforming to the MIME standard implemented by WebSphere Message Broker. See The MIME domain. • **MRM** For modelling a wide range of messages including XML, fixed-format binary and formatted text. See The MRM domain. • **JMSMap** or **JMSStream** For messages produced by the WebSphere MQ implementation of the Java Messaging Service standard. See The JMS domains. • **IDOC** The IDOC domain processes messages that are sent to the broker by SAP R3 clients across the MQSeries® link for R3.
Message Set	If the node is expecting a TextMessage or a BytesMessage that contains a payload to be interpreted using the MRM domain, the Message Set property should be set to the appropriate MRM Message Set ID. This is mandatory when using a Message Domain of MRM. The default value is blank (not set).
Message Type	If the Message Set property has been specified, you have the option to specify the Message Type. This property is used when you define multipart messages. See Multipart messages for a definition and explanation of multipart messages. The value that you specify is used as an absolute or relative path to the innermost message from the outermost message. The value is used as a prefix to the value of the Message Type property that is specified for the outermost message. The Message Set property value is specified either in the MQRFH2 header of the message, or in the input node of the message flow. If you set a value, it must be in the form `id1/id2/.../idnu` where `id1` is the identifier of the outermost message, `id2` is the identifier of the next element or message, and `idn` is the identifier of the innermost message. The default value is blank (not set).

Message Format	If you are using the MRM domain, you can model a variety of different physical representations using named physical formats. • Use the Custom Wire Format (CWF) physical format to model fixed format messages from legacy applications written in C, COBOL, PL/1 and other languages. This support includes the ability to create a message model directly from a C header file or COBOL copybook. • Use the XML Wire Format (XML) physical format to model XML messages, including those that use XML namespaces. This support includes the ability to create a message model directly from an XML DTD or XML Schema file. • Use the Tagged Delimited String Format (TDS) physical format to model formatted text messages, where the field content is identified by tags, is separated by specific de-limiters, or both. This support is rich enough to model industry standards such as SWIFT, EDIFACT and X12.

Troubleshooting JMS nodes

This section contains information to help you troubleshoot the JMS nodes.

- "Managing badly formed messages"
- "Diagnosing problems when using globally coordinated transactions:" on page 107
- "Problems with JNDI Administered Objects" on page 107
-

In all cases of error, if the underlying cause is a JMSException that has been thrown by the JMS provider, the broker bip event message will include the text message from the JMSException to aid diagnosis.

Managing badly formed messages:

If a message cannot be processed by the JMSInput node, or has been rolled back as part of a Global transaction, the message is backed out to the source destination. The message is then redelivered to the JMSInput node.

To prevent badly formed messages from interrupting the processing of valid messages, the node properties can be configured as follows:

Backout Destination	This is a JMS destination where backed out messages are routed if the JMS message property *JMSX_DeliveryCount*, which is set by the JMS provider, exceeds the backout threshold. The JMS destination must be applicable to the message model being used by the node; for example, if a subscription topic has been configured on the node, the JMS destination must also be a topic.

Backout Threshold	This is the integer value that controls a message is sent to the backout destination. A threshold value of 3 means that, if the JMSInput node receives a Message where the value of the *JMSX_DeliveryCount* property exceeds 3, the message is sent to the backout destination and is removed from the source destination.

Diagnosing problems when using globally coordinated transactions::

In addition to the broker service trace, another trace log is provided to diagnose problems that could occur when a JMSInput or JMSOutput node participates in a global message flow transaction. That is, at least one JMSInput or JMSOutput node in the message flow has the *Transaction Mode* property set to *global* and the message flow attribute *Coordinated Transaction* set to *yes*.

To capture the trace log, complete the following steps:

1. Define an environment variable called `XAJMS_TRACEFILE` that is available to the broker Queue Manager.
2. Set the value of the environment variable. This should be a character string that represents the location and file name of the trace log. For example, on windows the variable could be configured as follows:

   ```
   XAJMS_TRACEFILE = c:\JMSSwitchLog
   ```
3. When the broker Queue Manager starts, it performs a recovery step to resolve any previous broker transactions that the JMS provider considers to be in-doubt. This Queue Manager process will output two trace logs during this stage. The two trace logs are:
   - *<XAJMS_TRACEFILE value><PID>*`.txt`, where `PID` is the process ID of the Queue manager start process. This file is produced from the broker's JMSSwitch library, see "JMS Transactionality" on page 98 for more information.

     Using the above example value for the variable produces a file called `JMSSwitchLog2596.txt`, where the Queue Manager start up process ID was 2596.
   - *<XAJMS_TRACEFILE><XARecoveryTrace>*`.txt` which is produced by the broker's recovery component that connects to the JMS provider.
4. After the broker's Queue Manager has completed recovery, the broker starts and creates a file called *<XAJMS_TRACEFILE value><PID>*`.txt`, where `PID` is the process ID of the Queue manager start process. This file is produced from the broker's JMSSwitch library, see "JMS Transactionality" on page 98 for more information.

Neither of these trace files require extra formatting.

This is not applicable to Z/OS.

Problems with JNDI Administered Objects:

Description of problem: The JMSInput or JMSOutput node is unable to obtain the Initial Context Factory or a JNDI administered object such as the Connection Factory or JMS Destination, and a bip4640 message is issued.

Corrective action

1. Verify that the JNDI bindings have been correctly built, and can be reached at the location specified in the node.
2. Check that the values specified in the node for the Initial Context , Connection Factory Name, and Source Queue or Destination Queue exist in the JNDI bindings.
3. Ensure that the correct keyword is used to match the location of the bindings
   - file:/ when the administered objects have been created in a .bindings file
   - ldap:/ when the administered objects exist in an LDAP directory
   - iiop:/ when corba is used to access the administered objects
4. When the bindings are file based do not specify the .bindings filename in the node attribute.
5. Ensure that the Initial Context Factory name does not include a filepath.
6. Ensure that a JMS Destination (Topic or Source Queue or Destination Queue) specified in the node attributes exists in the JNDI administered objects.
7. Ensure that the JMS Provider Java .jars have been placed into the broker shared-classes directory on distributed platforms, or on z/OS that these jars have been defined to the broker CLASSPATH and any native libraries defined in the broker LIBPATH.

The JMS Nodes will continue to attempt to obtain the JNDI administered objects. Correct any problems and rebuild the bindings. The JMS node should automatically detect the changes and attempt to start.

Description of problem: A JMSInput or JMSOutput Node is unable to connect for a JMS provider and issues a bip4648 message.

Corrective action:

1. Verify that the JMS Provider server is running. Start it if it is offline.
2. Verify that the JMS Provider server is available from the broker environment.
3. Ensure that the JMS Provider Java .jars have been placed into the broker shared-classes directory on distributed platforms, or on z/OS that these jars have been defined to the broker CLASSPATH and any native libraries defined in the broker LIBPATH.

The JMS Nodes will continue to attempt to connect to the JMS provider. Correct any problems and the JMS node should automatically detect the changes and attempt to connect to the provider.

Description of problem: A JMSInput or JMSOutput Node is unable to obtain a JMS destination and issues a bip4642 message.

Corrective action

1. Investigate the cause of the problem described by the JMS exception message that may be included in the bip event message.
2. Check that the name of the JMS destination attribute defined in the node attribute (Topic, Source Queue or Desination Queue) has been correctly defined in the JNDI administered objects.
3. Verify that the underlying system resource used by the JMS provider for the JMS Destination has been configured correctly

Application environments

This section covers the following topics:
- "MQGet node message processing"

MQGet node message processing

The following diagrams illustrate the processing done within the MQGet node to:

- Propagate the message
- Construct the Output tree LocalEnvironment
- Construct the Output tree message

Propagating the message

1. If there is an MQMD in the input tree it is used, otherwise a default MQMD is used.
2. A default MQGMO is created, then if there is a GMO in the input tree it is used to modify the default one according to the node attributes.
3. The MQGet call is made to WebSphere MQ.
4. The Return Code (CC) from the call is analysed, and the message propagated accordingly as follows:

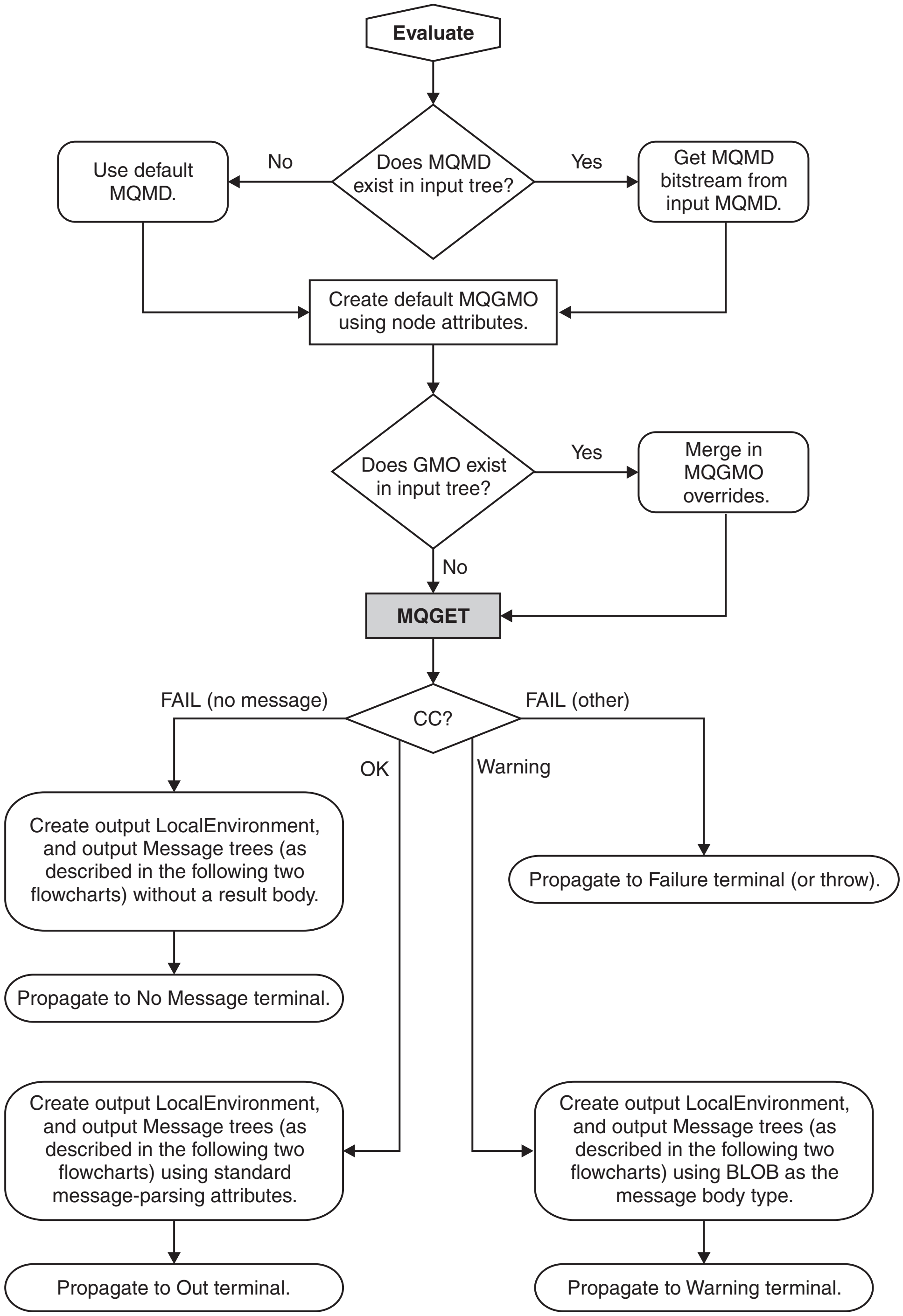
   OK Create output LocalEnvironment and output Message trees using standard message-parsing techniques. Propagate to Out terminal.

   Warning
   Create output LocalEnvironment and output Message trees using BLOB as the message body type. If connected, propagate to Warning terminal. Otherwise, no propagation occurs and the flow ends.

   Fail (no message)
   Create output LocalEnvironment and output Message trees by copying the input. If connected, propagate to No Message terminal. Otherwise, no propagation occurs. The output message that is propagated to the No Message terminal is constructed from the input message only, according to the values of the Generate Mode and Copy Message, or Copy Local Environment properties.

   Fail (other)
   Propagate to Failure terminal. If the Failure terminal is not connected the broker throws an exception and returns control to the closest previous node that can process it. See Handling errors in message flows for more information.

The following diagram illustrates this visually:


Evaluate
Does MQMD exist in input tree?
No
Yes
Use default MQMD.
Get MQMD bitstream from input MQMD.
Create default MQGMO using node attributes.
Does GMO exist in input tree?
Yes
Merge in MQGMO overrides.
No
MQGET
CC?
FAIL (no message)
FAIL (other)
OK
Warning
Create output LocalEnvironment, and output Message trees (as described in the following two flowcharts) without a result body.
Propagate to No Message terminal.
Propagate to Failure terminal (or throw).
Create output LocalEnvironment, and output Message trees (as described in the following two flowcharts) using standard message-parsing attributes.
Create output LocalEnvironment, and output Message trees (as described in the following two flowcharts) using BLOB as the message body type.
Propagate to Out terminal.
Propagate to Warning terminal.

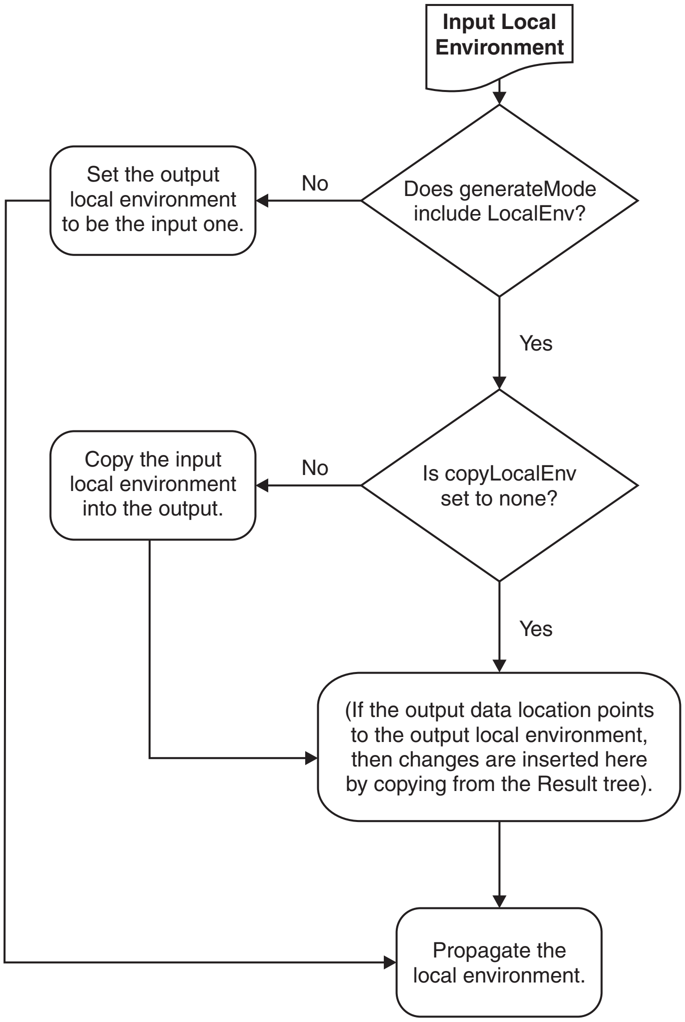
Constructing the Output LocalEnvironment

1. If the **generateMode** attribute on the MQGet node is not one of the options that includes `LocalEnv`, then set the output local environment to be the input, and propagate the local environment.

   Note: In this case, no updates that go into OutputLocalEnvironment will be propagated downstream.
2. Otherwise, if the **copyLocalEnv** attribute is not set to `none`, then copy the input local environment into the output.

3. If the output data location points to the output local environment, then changes are inserted here by copying from the result tree.
4. The local environment is propagated.

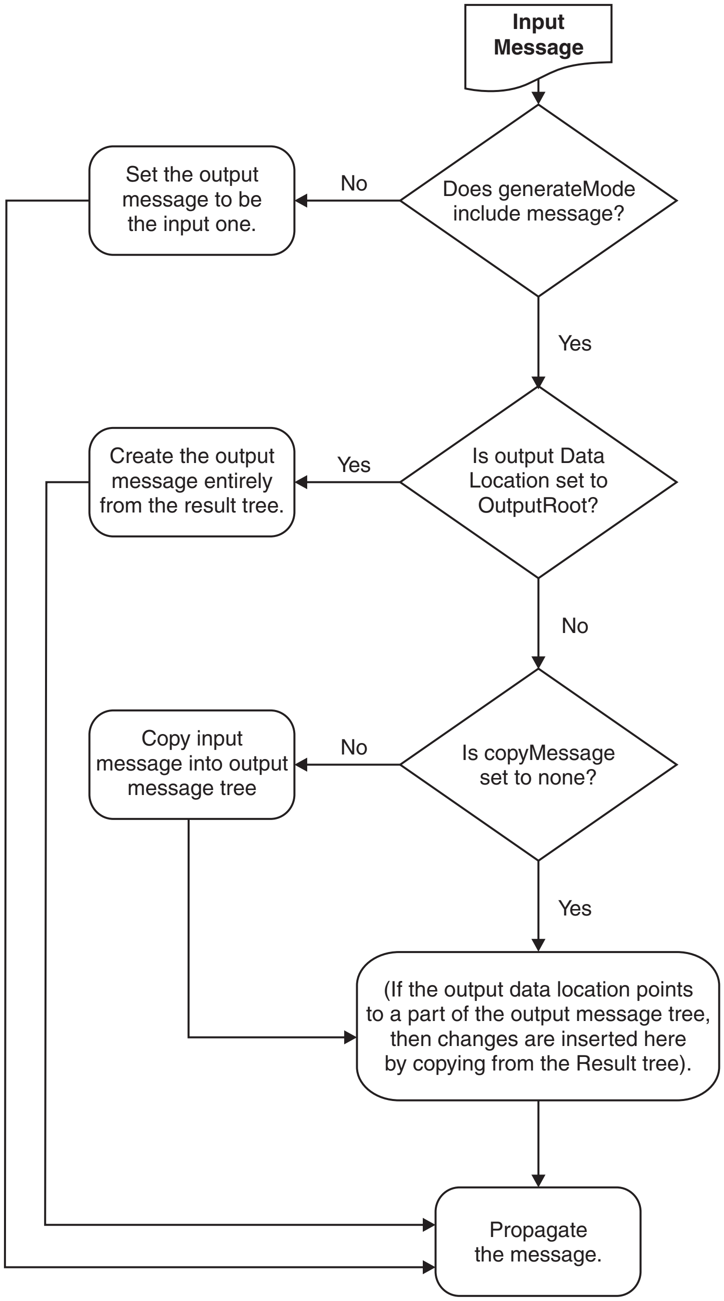
The following diagram illustrates this visually:


Input Local Environment
Does generateMode include LocalEnv?
No
Set the output local environment to be the input one.
Yes
Is copyLocalEnv set to none?
No
Copy the input local environment into the output.
Yes
(If the output data location points to the output local environment, then changes are inserted here by copying from the Result tree).
Propagate the local environment.

Constructing the Output message

1. If **generateMode** does not include message, then set the output message to be the input one. Goto step 5.
2. If **output Data Location** is set to `OutputRoot`, then create the output message entirely from the result tree. Goto step 5.
3. If **copyMessage** is not set to `none`, then copy the input message into the output message tree.
4. If the output data location points to a part of the output message tree, then changes are inserted here by copying from the result tree at the point defined by result Data Location.
5. The message is propagated.

The following diagram illustrates this visually:


Input Message
Does generateMode include message?
No
Set the output message to be the input one.
Yes
Is output Data Location set to OutputRoot?
Yes
Create the output message entirely from the result tree.
No
Is copyMessage set to none?
No
Copy input message into output message tree
Yes
(If the output data location points to a part of the output message tree, then changes are inserted here by copying from the Result tree).
Propagate the message.

For detailed examples of how to exploit this, see Using an MQGet node in a request-response flow.

Part 3. Appendixes

Appendix. Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032,
Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

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

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

COPYRIGHT LICENSE:

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

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _*enter the year or years*_. All rights reserved.

Trademarks

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

AIX
CICS
Cloudscape
DB2
DB2 Connect
DB2 Universal Database
developerWorks
Domino
Everyplace
FFST
First Failure Support Technology
IBM
IBMLink
IMS
IMS/ESA
iSeries
Language Environment
Lotus
MQSeries
MVS
NetView
OS/400
OS/390
POWER
pSeries
RACF
Rational
Redbooks
RETAIN
RS/6000
SupportPac
S/390
Tivoli
VisualAge
WebSphere
xSeries
z/OS
zSeries

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

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

Intel and Pentium are trademarks or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

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

Index

A

APIs 7
application clients 3
  MQGet node message processing 109
  multicast 10
  real-time (JMS) 11
  supporting 3
  telemetry (SCADA) 11
    command messages 68
    designing applications 24
    example message flow 25
    message format 60
    Quality of Service 58
    UTF-8 84
    variable header contents 64
  Web services 12
    applications 27
    call existing 37
    existing as new Web service 43
    existing to existing Web service 47
    existing Web service 39
    HTTP flows 33
    implement existing interface 46
    implement existing interface to new 50
    implement new 41
    message flows 28
    scenarios 36
    SOAP applications 29
    SOAP over JMS 49
    WSDL 28
    WSDL applications 31
    WSDL operation types 32
  WebSphere Broker JMS Transport 15
  WebSphere MQ 9
    building IDoc message model 22
    defining 20
    securing 21
  WebSphere MQ Everyplace 10
Application environments 19
Application Messaging Interface (AMI) 7
application programming interfaces 7
  Application Messaging Interface (AMI) 7
  Java Message Service (JMS) 7
  Message Queue Interface (MQI) 7
application transports 9

C

configuration, transactionality 98

E

end-user applications 3
  communication models 4
  supporting 3

H

headers
  MQRFH 7
  MQRFH2 7

I

IDoc
  message model 22

J

JMS 98
  broker domain configuration 104
  brokering 98
    connecting providers 19
    JMS providers 19
    message representation 17
    support for JMS messages 16
    transforming messages 17
  brokering and provision 15
  configuration, broker domain 104
  configuration, message model 103
  configuration, message selector 102
  creating a message for output 96
  deriving the parser 94
  header and property data 91
  JNDI administered objects 97
  message as input 91
  message domain 95
    order of precedence 95
  message model configuration 103
  message selector configuration 102
  message structure 87
  message types 88
  output message 96
  parser 94
  payload processing 93
  preservation of Java type 91
  provision 98
  receiving a message 91
  representation of messages 90
  troubleshooting 106
JNDI administered objects 97

M

message definitions
  importing from WSDL
    WSDL validation 33
message headers 7
message models
  IDoc 22
Message Queue Interface (MQI) 7
messages 7

P

point-to-point communication model 4
publish/subscribe communication model 4

Q

Quality of Service 11

T

trademarks 117
transactionality configuration 98

W

Web services
  example messages 15
  headers 13
  message format 12
WebSphere MQ Enterprise Transport 9
WebSphere MQ Everyplace 10
WebSphere MQ Mobile Transport 10
WebSphere MQ Multicast Transport 10
WebSphere MQ Real-time Transport 11
WebSphere MQ Telemetry Transport 11
WebSphere MQ Web Services Transport 12
WSDL
  validation 33

Printed in USA