Novell exteNd Composer™ JMS Connect

5.0

USER'S GUIDE

Novell®

Legal Notices

Copyright © 2000, 2001, 2002, 2003, 2004 SilverStream Software, LLC. All rights reserved.

Title to the Software and its documentation, and patents, copyrights and all other property rights applicable thereto, shall at all times remain solely and exclusively with SilverStream and its licensors, and you shall not take any action inconsistent with such title. The Software is protected by copyright laws and international treaty provisions. You shall not remove any copyright notices or other proprietary notices from the Software or its documentation, and you must reproduce such notices on all copies or extracts of the Software or its documentation. You do not acquire any rights of ownership in the Software.

Novell, Inc.

1800 South Novell Place

Provo, UT 85606

www.novell.com

exteNd Composer JMS Connect ***User's Guide***

January 2004

Online Documentation: To access the online documentation for this and other Novell products, and to get updates, see www.novell.com/documentation.

Novell Trademarks

eDirectory is a trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

Jakarta-Regexp Copyright ©1999 The Apache Software Foundation. All rights reserved. Xalan Copyright ©1999 The Apache Software Foundation. All rights reserved. Xerces Copyright ©1999-2000 The Apache Software Foundation. All rights reserved. Jakarta-Regexp , Xalan and Xerces software is licensed by The Apache Software Foundation and redistribution and use of Jakarta-Regexp, Xalan and Xerces in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notices, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "The Jakarta Project", "Jakarta-Regexp", "Xerces", "Xalan" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their name, without prior written permission of The Apache Software Foundation. THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright ©1996-2000 Autonomy, Inc.

Copyright ©2000 Brett McLaughlin & Jason Hunter. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution. 3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org. 4. Products derived from this software may

not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org). THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This Software is derived in part from the SSLava™ Toolkit, which is Copyright ©1996-1998 by Phaos Technology Corporation. All Rights Reserved. Customer is prohibited from accessing the functionality of the Phaos software.

The code of this project is released under a BSD-like license [license.txt]: Copyright 2000-2002 (C) Intalio Inc. All Rights Reserved. Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The name "ExoLab" must not be used to endorse or promote products derived from this Software without prior written permission of Intalio Inc. For written permission, please contact info@exolab.org. 4. Products derived from this Software may not be called "Castor" nor may "Castor" appear in their names without prior written permission of Intalio Inc. Exolab, Castor, and Intalio are trademarks of Intalio Inc. 5. Due credit should be given to the ExoLab Project (http://www.exolab.org/). THIS SOFTWARE IS PROVIDED BY INTALIO AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE DISCLAIMED. IN NO EVENT SHALL INTALIO OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

About This Guide **9**

1 **Welcome to exteNd Composer and JMS** **11**

About exteNd Connects 12
What Is the JMS Connect? 13
What Needs Does JMS Address? 13
What Is Enterprise Messaging? 14
What Are Message Queues? 15
Will My Message-Based Application Be Slow? 16
Is Messaging Reliable? 16
Can Messages Be Part of Transactions? 17
What Is Point-to-Point Messaging? 17
What Is Publish/Subscribe Messaging? 18
What About Delivery Guarantees? 19
How Are Messages Structured? 20
  Header Information 20
  Body Types 21
How Are Messages Retrieved? 22
Message Filtering 23
Request-Response versus Store/Forward 23
What Does JMS Not Cover? 24
About exteNd's JMS Component 24

2 **Getting Started with the JMS Component Editor** **27**

Creating a JMS Connection Resource 27
  About Expression-Driven Connections 28
  About Queue Connections 29
  About Topic Connections 36
Creating XML Templates for Your Component 42

3 **Creating a JMS Component** **43**

Before Creating a JMS Component 43
About the JMS Component Editor Window 48
About the Native Environment Pane 49

4 **Creating JMS Actions** **51**

About Actions 51
Actions Unique to the JMS Component Editor 52
  Options Tab 52
  Message Body Tab 53
  Message Header Tab 54

The Send Message Action . . . 55
Priority, Mode, and Time to Live . . . 55
Destination Queue/Topic. . . . 55
Return Address . . . 57
The Browse Messages Action . . . 64
The Receive Message Action . . . 69
The Message Transaction Action . . . 75
What Happens When I Issue a Commit? . . . 75
What Happens When I Issue a Rollback? . . . 76
What Happens if I Leave the Session Unresolved? . . . 76
What Actions Are Included in a Message Transaction? . . . 76
What Can I Use Message Transactions For? . . . 77
Using Other Actions in the JMS Component Editor . . . 79

5 **Working with Messages** **81**
Mapping Data into the Message Header . . . 81
Limitations on Header Mapping . . . 83
Mapping Data to Custom Properties . . . 84
Limitations on Property Mapping. . . . 85
Working with XML Messages . . . 85
Working with Copybook Messages . . . 91
Copybook Message Setup . . . 91
Copybooks and the Native Environment Pane. . . . 92
Copybook-Specific Context Menu Items . . . 94
Mapping Data Between Copybook and DOMs. . . . 96
Working with Message Filters (Selectors). . . . 98
Limitations on Filtering . . . 100
Filtering by Body Type . . . 100
Request-Response Messaging . . . 101
Temporary Queues . . . 102
ECMAScript and the JMS Connect . . . 104
ECMAScript Method Summary . . . 107

6 **The JMS Service** **111**
About the JMS Service . . . 111
Multiple Listeners . . . 112
Creating a JMS Service . . . 112
Deployment of the JMS Service. . . . 117
How Do I Manage Deployed JMS Services? . . . 118

A **JMS Glossary** **121**

B **Message Selector Syntax** **127**
Literals . . . 127
Identifiers . . . 127

Expressions 128
Comparisons 129
Null Values 130
Special Considerations. 130

C **Message Headers and Properties** **131**
Header Fields Defined by JMS. 131
JMSCorrelationID 131
JMSDeliveryMode 131
JMSDestination 131
JMSExpiration 132
JMSMessageID 132
JMSPriority 132
JMSRedelivered 132
JMSReplyTo 132
JMSTimestamp 133
JMSType 133
Message Properties 133
JMS-Defined Properties. 133
Provider-Specific Properties 134
User-Defined Properties 134

About This Guide

Purpose

This guide describes how to use the JMS Component Editor, which is the design-time portion of the exteNd Composer JMS Connect.

Audience

This book is for systems analysts, programmers, and others who intend to build applications or services that require a Message Oriented Middleware component, where the host MOM system in question is compatible with Sun Microsystems' Java Message Service API.

Prerequisites

This book assumes prior familiarity with the exteNd Composer design-time environment and Composer application-building metaphors. You should also be familiar with MOM and JMS concepts.

Additional documentation

For the complete set of Novell exteNd Director documentation, see the Novell Documentation Web Site:

http://www.novell.com/documentation-index/index.jsp.

1 Welcome to exteNd Composer and JMS

Welcome to the *Novell exteNd JMS Connect User's Guide*. This Guide is a companion to the *exteNd Composer User's Guide*, which details how to use all the features of Composer except for the various Connect Component Editors. So, if you haven't looked at the *Composer User's Guide* yet, please familiarize yourself with it before using this Guide.

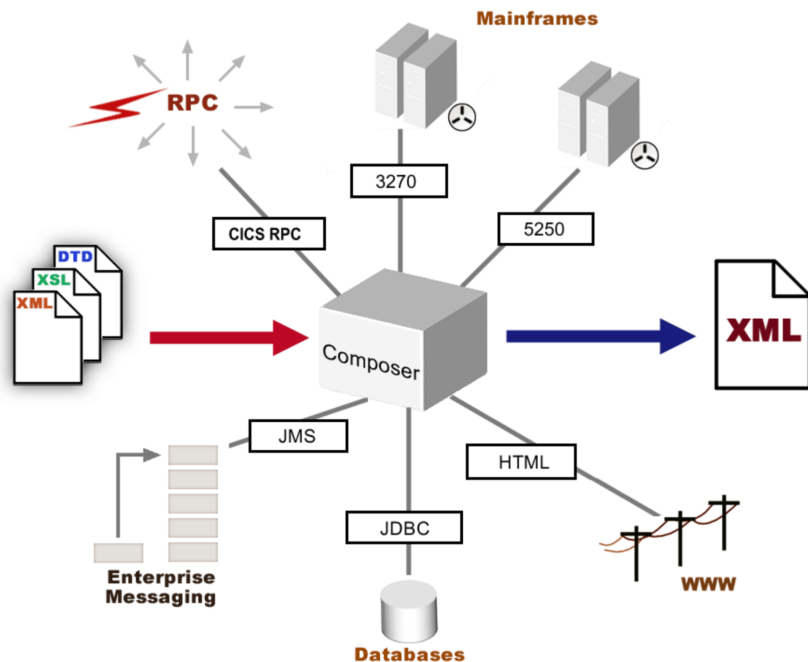
exteNd Composer provides separate Component Editors for each Connect, such as the JMS Connector. The special features of each Component Editor are described in individual Guides like this one.

Before you begin working with the JMS Connect, you must have it installed into your existing exteNd Composer. Likewise, before you can run any Services built with this connector in the exteNd Server environment, you must have already installed the corresponding Server software for this connector into exteNd Server.

NOTE: To be successful with this Component Editor, you should be familiar with Message Oriented Middleware (MOM) concepts and the particular MOM environment (e.g., MQSeries) into which you will be deploying. While the pages that follow offer a brief introduction to important enterprise-messaging concepts, a truly comprehensive approach is beyond the scope of this guide. In any event, the discussion offered here is in no way meant to substitute for the documentation supplied by your JMS provider.

About exteNd Connects

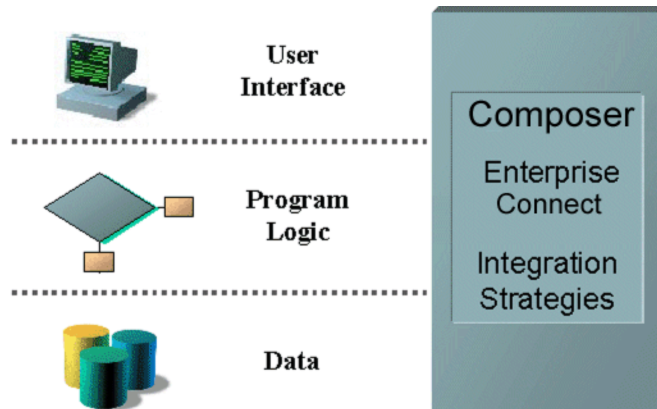
Novell exteNd is built upon a simple hub and spoke architecture. (See illustration below.) The hub is a robust XML transformation engine that accepts XML documents, processes the documents, and returns an XML document. The spokes or Connects are plug-in modules that "XML enable" sources of data that are not XML-aware, bringing their data into the hub for processing as XML. These data sources can be anything from legacy COBOL/VSAM managed information to Message Queues to HTML pages. exteNd Connects can be categorized by the integration strategy each one employs to XML enable an information source. The integration strategies are a reflection of the major divisions used in modern systems designs for Internet based computing architectures. Depending on your B2Bi needs, exteNd can integrate your business systems at the User Interface, Program Logic, and/or Data levels.



Hub and spoke architecture allows exteNd to provide enterprise-wide XML integration via Connects (EEs).

What Is the JMS Connect?

Java Messaging Service (JMS) is a Java-based interface for using Message Oriented Middleware (MOM) services, such as provided by IBM's MQSeries or Progress Software's SonicMQ. In order for distributed applications running on Java application servers to make full use of messaging systems, Java-language clients and Java middle-tier services must have a common way to "speak to" enterprise messaging products. JMS provides that capability.



Novell exteNd can integrate business systems at the User Interface, Program Logic, and/or Data levels.

The exteNd JMS Connect allows you to create Components that can send, receive, and/or browse messages in queues administered by a JMS-based MOM system, using transacted or non-transacted sessions. JMS-enabled exteNd services are able to enjoy the dual benefits of asynchronous processing and transport-layer independence that characterize enterprise messaging. Using the JMS Connect, you will be able to create powerful, flexible applications that make optimal use of system resources while carrying out potentially complex operations involving remote invocation of objects, assured "once only" delivery of notifications, and/or distributed transactions.

What Needs Does JMS Address?

The JMS standard was built with several goals in mind:

- Provide an Application Programming Interface suitable for creating and manipulating messages in formats compatible with existing MOM products.

- Support many different message-content types, including messages containing Java objects.
- Facilitate the development of heterogeneous applications that span operating systems, machine architectures, transport mechanisms, and computer languages.

JMS is a broadly applicable Java API that is intended to be layered over a wide range of existing and future Message Oriented Middleware systems, much the same way that JNDI (the Java Name and Directory Interface) is layered over existing name and directory services.

The complete JMS specification is available at **http://java.sun.com/products/jms/**.

What Is Enterprise Messaging?

An enterprise messaging system provides for the transport and storage of messages. *Messages*, in this context, are packets of information that are produced and/or consumed primarily by enterprise applications (rather than humans). They may contain key-value pairs, XML documents, serialized Java objects, or arbitrary byte streams.

One of the main attractions of Message Oriented Middleware is its ability to serve as an abstraction layer that hides the details of message transport and delivery from diverse clients that may need to communicate across networks that use different communication protocols. By acting as a communications gateway, MOM shields clients from connectivity issues that would otherwise impede development of distributed applications.

Another aspect of enterprise messaging that makes products like MQSeries and SonicMQ so useful is their ability to link processes in *asynchronous* fashion. Asynchronous processing means that the exchange of data between parties does not depend on either party being in direct, realtime contact with the other. The alternative to asynchronous processing is *synchronous* processing, wherein a host and a client (or a peer and a peer) *must* be in continuous conversation with each other for the entire duration of a session, without interruption. (An example of a synchronous interchange would be the use of a Remote Procedure Call in a CICS environment.) While synchronous operations are required for some types of interactions, there are many kinds of business processes that do *not* require synchronous communication between participants. For such processes, asynchronous interaction generally makes the most efficient use of resources and can dramatically improve system productivity.

A synchronous process is analogous to a restaurant in which every customer orders his meal directly through a conversation with the chef and the kitchen takes no other orders while the current customer's meal is being cooked. Customers line up and must wait, one by one, for every meal to be individually prepared. Asynchronous processing would be analogous to the more familiar scenario of waiters and waitresses conveying orders between customers, kitchen, and bar concurrently. In the latter case, the waiters serve as a messaging channel to the kitchen, where orders are "queued up" and finished meals are dispersed on an as-available basis. In this example (as in many business processes), much better efficiencies are possible with asynchronous order processing than with synchronous order processing.

The time-domain decoupling afforded by messaging systems helps make robust, fail-safe operation possible. One party can be busy—or even offline—when the other party sends (or receives) its message. The sender can continue processing without needing to wait for an acknowledgement from the receiver. A network or server can go down, yet not affect the transmission or receipt of a message.

What Are Message Queues?

Asynchronous messaging depends on the fact that messages are sent not to clients, per se, but to *queues*, which exist independently of the client processes that use them.

A *queue* is a holding area or repository in which data elements (messages, in this case) are stored for eventual retrieval. In a MOM environment, client applications needn't know how message queues are structured, maintained, or stored; the details of queue management are handled by the MOM vendor (or "JMS provider"). Like server nodes, queues are often *clustered* for purposes of reliability, scalability, and load balancing.

While FIFO (first-in/first-out) and LIFO (last-in/first-out) queues are familiar constructs in computing, *no order of retrieval is presumed in a message queue*. Rather, the retrieval order is open. This means custom prioritization schemes can be applied to messages so that retrieval order (i.e., consumption order) is dependent on a client's needs. Properly exploited, this feature can lead to more efficient overall system operation. Processing of low-priority messages can be deferred to a time when system resources are available; low-priority items needn't interfere with the processing of high-priority ones.

The process of inspecting messages without removing them from the queue is called *browsing*.

NOTE: The length of time messages are held in a queue, the maximum number of messages a queue can handle, and the manner in which resource overruns are handled are not defined by the JMS standard. Consult your MOM vendor's documentation to learn more about these issues.

Will My Message-Based Application Be Slow?

While some latency occurs in all messaging systems, this does not mean that applications that use messaging are, of necessity, slow. The asynchronous processing made possible by messaging affords the possibility of multitasking inside an application, which could (depending on the application) actually boost throughput. For example, while a customer adds items to his shopping cart, the shopping-cart app can trigger an inventory-checking component, while another component can calculate shipping charges, while another component pulls customer information out of a database, etc., all operations taking place concurrently.

The choice of messaging model (Point-to-Point versus Publish/Subscribe) has important implications for latency. See "What Is Point-to-Point Messaging?" and "What Is Publish/Subscribe Messaging?" further below.

Is Messaging Reliable?

While the quality-of-service guarantees offered in Message Oriented Middleware solutions can vary greatly, and while real-world reliability often depends on administrative issues (such as cluster size and available resources), *all* JMS-based messaging services are required to offer *assured, once-only* delivery of messages as an option for applications where reliability is paramount. JMS also allows for configurations that provide a less robust quality of service, so that in cases where *speed* of delivery might be more important than assured, once-only delivery, a tailored solution can be built. The reliability of JMS-based messaging solutions is thus configurable.

In general, strong reliability guarantees are a common feature of JMS-based systems.

Can Messages Be Part of Transactions?

One of the things that makes JMS messaging attractive from a reliability standpoint is that message sessions can (optionally) incorporate *transaction control*. A transacted session groups an arbitrary set of produced and/or consumed messages into a single logical unit of work. When a transaction *commits*, all of its inputs (in terms of messages) are acknowledged and all outputs are sent. When a transacted message session *rolls back*, any produced messages are destroyed and any messages consumed during the session are recovered.

An an example, suppose an application builds a group of five messages. A requirement of the application is that the entire group of five messages must be sent as a batch; or else none of the five must be sent. Using a JMS Component, the application could be structured such that messages are built and sent individually, but if a connection closes prematurely (or any other error condition happens, involving any of the five messages), the entire group is rolled back.

NOTE: JMS does not require that MOM products support *distributed* transactions. But if such support exists, JMS does require that such support be implemented via the JTA (Java Transactions API) *XAResource* interface. Consult your JMS provider's documentation to see what kind of distributed transaction support, if any, is available in your MOM environment.

NOTE: Since distributed transactions are controlled via JTA, the use of message-session *commit* or *rollback* commands in this context will cause a JMS *TransactionInProgressException* to be thrown.

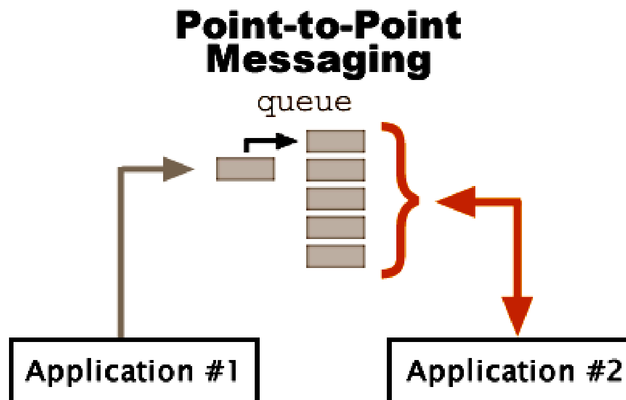
What Is Point-to-Point Messaging?

Two main messaging paradigms are implemented by MOM vendors: *Point-to-Point* (PTP) messaging, and *Publish/Subscribe* (which is discussed further below). Some vendors implement one or the other; some implement both.

NOTE: Point-to-point does *not* imply a synchronous connection in the context of messaging (as it does in some other contexts, such as discussions of RPC).

In the PTP model, any JMS client can—in theory—send messages to any other JMS client, subject only to administrative constraints. PTP is an asynchronous, queue-based, peer-to-peer model in which queues are typically created administratively and have indefinite lifespans. A queue is always available to receive and hold messages sent to it whether the client or clients using that queue are online or not.

With PTP, a queue functions much like a mailbox. One application might send messages to a queue; another application might retrieve messages from the same queue. A common case is that a client will have all its messages delivered to a single queue.

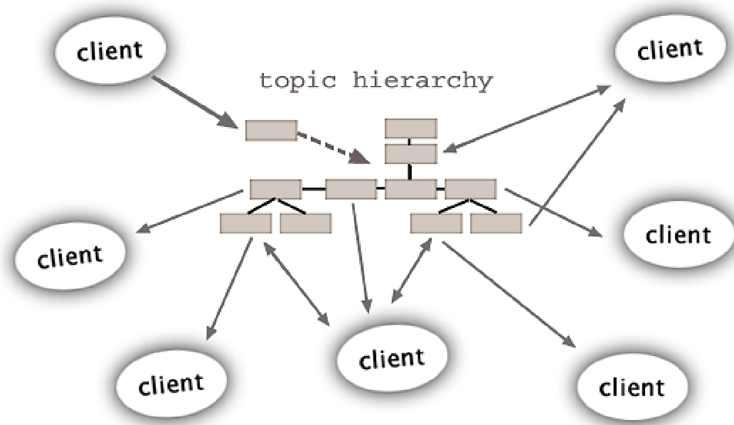


The Point-to-Point model is a queue-based, peer-to-peer model in which queues act, essentially, as mailboxes. Client applications can post to a queue, or (as with Application #2 above) browse, retrieve messages one-by-one, or continuously poll the message queue. Optionally, clients can implement a MessageListener that will act on messages as they are received.

What Is Publish/Subscribe Messaging?

The Publish/Subscribe (or pub/sub) messaging model—implemented by some (but not all) MOM vendors—differs from Point-to-Point in the following ways:

- Queues are typically shared by multiple clients.
- Queues are organized hierarchically into nodes called *topics*. (This is a typical implementation scheme, although in fact JMS places no restrictions on what a topic can represent.)
- Each topic acts as a kind of mini-message-broker that accumulates and distributes messages addressed to it.



In a Publish/Subscribe system, queues are usually organized hierarchically into nodes called topics. Clients may subscribe and/or publish to any number of topics.

Clients in this kind of system use message producer/consumer objects called *TopicPublishers* and *TopicSubscribers*. A client may subscribe to more than one topic; and a client may be both a subscriber and a publisher.

TopicSubscribers can be *durable* or *non-durable*. If a client needs to have access to *all* messages on a given topic (including ones that may be published when the subscriber is offline), a durable TopicSubscriber must be used. Otherwise, the client will have access only to messages that are queued during the lifetime of a given message-retrieval session.

NOTE: Messages are served to subscribers in serial fashion. Because topics are shared resources (and because only one subscriber can be serviced at a time), the potential for latency is somewhat greater in pub/sub messaging than in PTP.

What About Delivery Guarantees?

JMS supports two modes of message delivery.

- The PERSISTENT mode instructs the message broker to write the message to a secure store to insure that the message is not lost in transit due to a system failure.

- The NON_PERSISTENT mode does not require the JMS provider to log the message to stable storage; thus, the message can, in theory, be lost. (The tradeoff here is one of performance. There is less overhead with a NON_PERSISTENT message.)

A JMS provider is required to deliver a NON_PERSISTENT message *at-most-once*. This means that while a message may sometimes be lost, it will never be delivered twice.

By contrast, the delivery guarantee for PERSISTENT messages is *once-and-only-once*. This means a provider failure must not cause a message to be lost in transit; and the message must not be delivered more than once.

NOTE: Once-and-only-once delivery has the important limitation that it cannot and does not guarantee against message loss due to message expiration, resource overruns, or administrative destruction criteria. *Configuring a message system for maximum reliability requires a thorough understanding of administrative issues surrounding the particular MOM solution in use.*

How Are Messages Structured?

Enterprise messaging products treat messages as persistible, lightweight entities that consist of a header, a body, and (in the case of JMS-aware products) a property list.The *header* contains fields used for message routing and identification. The *body* contains the application data being sent. *Properties* provide a mechanism for adding arbitrary descriptors (actually implemented as extra header fields) to messages so that clients—or their providers—can select or “filter” messages on the basis of application-specific criteria.

Header Information

Among the message characteristics defined in the header are an *expiration time* (which sets the message’s useful life); *message priority* (based on a number ranking, from zero to nine); and *delivery mod*e (PERSISTENT or NON_PERSISTENT).

The header fields defined by JMS are:

- JMSCorrelationID
- JMSDestination
- JMSDeliveryMode
- JMSExpiration
- JMSPriority
- JMSMessageID

- JMSTimestamp
- JMSRedelivered
- JMSReplyTo
- JMSType

In addition to these predefined header fields (which every JMS message is required to have), there are JMS-defined property fields (many of which are optional), provider-specific properties, and user properties.

The semantics of the various header and property fields are discussed in detail in Appendix C.

Body Types

JMS defines five message body types:

1. **MapMessage**—a message in which the body consists of a set of key-value pairs, wherein the keys are Java *Strings* and the values are Java primitive types. Entries may be accessed by sequential enumeration or randomly by name. (The ordering of key-value pairs is undefined.)
2. **TextMessage**—a message in which the body is a *java.lang.String*.
3. **StreamMessage**—a message whose body consists of a stream of Java primitive values (which are filled and read sequentially).
4. **ObjectMessage**—a message containing a Serializable Java object. (For collections of objects, one of the *collection classes* defined in JDK 1.2 can be used.)
5. **BytesMessage**—a message comprising any arbitrary stream of uninterrupted bytes. (This category is intended for encoding a binary payload, or a special payload to match a vendor's native message format.) In Composer, the content of a BytesMessage should be Base64-encoded.

NOTE: Regardless of type, all JMS messages are *read-only* once posted to a queue.

The JMS Connect allows you to define message payloads using any of the five canonical JMS body types. In addition, the JMS Connect offers two predefined message types (which are actually wrappers for two of the predefined JMS body types):

- **XML**—Allows you to send or receive an XML document (based on any XML template of your choosing) as a message. The complete DOM representation of the XML document is available to you for mapping and/or manipulation via ECMAScript and XPath; and you can add new nodes to the template document. This message type wrappers the JMS-defined TextMessage type.
- **Copybook**—Allows you to send or receive a COBOL copybook (as a BytesMessage).

See Chapter 4 for more information.

How Are Messages Retrieved?

JMS provides two mechanisms for client retrieval of messages:

1. Synchronous message retrieval, wherein a timeout value can be specified for terminating the session should no response occur.
2. Asynchronous retrieval via a *MessageListener* object whose `onMessage()` method contains application logic for processing incoming messages.

NOTE: Here, the terms "synchronous" and "asynchronous" refer to the queue/client communication session rather than any relationship between *sender* and client. Senders can always post to queues, whether or not receiving clients are online; in that sense, all messages are asynchronously received.

In synchronous retrieval, a timeout value can be specified in milliseconds. If no expiration value is specified, the "receive" session will block indefinitely, until a message arrives. On the other hand, if a *zero* wait time is specified, queued messages that meet the applicable selection criteria (if any) will be retrieved—and the session terminated—immediately.

Asynchronous retrieval treats messages like events and allows clients to be notified immediately (and take action on) messages as they arrive. Application logic triggered by an `onMessage()` handler processes messages transparently, with a minimum of latency. A broadcaster/listener metaphor applies in this case.

One way to think of it is that in a synchronous-retrieval scenario, the client is *pulling* data from the queue; in the asynchronous case, the queue is *pushing* data at the client.

Message Filtering

Some applications need to filter and/or categorize the messages they send or receive. In some instances, the receiving application can simply inspect the message body and decide—from the message contents—whether the message should be acted upon, or discarded. But it is often more efficient for selection criteria to be exposed in the message header, so that the message body need not be parsed in order to determine if the message is one that should be acted upon.

Exposing message selection hints in the header portion of a message is a common tactic when multiple receiving apps are pointed at the same queue. The application that is best suited to handling a given message type can harvest just the messages it needs, while other applications can act on messages better suited for *them*. Administratively, it is more efficient to set up one queue (with multiple receivers accessing it) than to set up multiple queues, each with a dedicated receiver.

Another factor to consider is that when potential selection criteria are visible (via the header) to a JMS provider, the provider can avoid delivering messages to clients that might not need them. In effect, filtering can be delegated to the JMS provider. (This strategy is important in Publish/Subscribe messaging.)

JMS defines a *message selector* that can be used for screening messages on a queue. The selector is an expression (with syntax similar to SQL92) that evaluates to true or false when header field and/or property values are substituted for their corresponding identifiers in the selector.

The exteNd JMS Connect implements selectors in the Message Filter tab of the Native Environment pane for all Browse and Receive actions; this allows you to filter incoming messages according to whatever criteria you select.

See Chapter 4 and Appendix B for more information.

Request-Response versus Store/Forward

A *request-response* scenario involves an application sending a message in anticipation of receiving a reply. For example, a credit-clearing application might package customer information into a message and send that message to a queue, where a receiving application retrieves the message, performs the necessary database queries and other processing, then replies to the original message.

This is different from the *store/forward* or “fire and forget” type of scenario in which a message producer simply places a message on a queue and terminates (or goes on to other processing). Messages that are sent in this fashion are sometimes called *datagrams*.

The JMS Connect supports both kinds of scenarios. However, the request-response scenario must (in this version of the connector) be implemented using individual Send Message and Receive Message actions. (That is, there is no *one* action type that encapsulates a request-response session.) If the request message and response message share the same queue, the associated send and receive actions can occur serially in the same JMS Component. But if the outgoing message will be placed on a different queue than the incoming reply, then two separate JMS Components must be created, since only one queue can be used per JMS Component.

See Chapter 4 for additional information.

What Does JMS *Not* Cover?

The JMS standard defines numerous message-system behaviors and data types but does not address administrative concerns, performance tuning, security, configuration issues, nor a variety of other JMS-provider functions.

Among the areas *not* addressed by JMS are:

- Load balancing
- Scalability
- Transparent failover
- System-wide error notifications or warnings
- User authentication
- Secure transport of messages (privacy)
- Communications protocols
- Message type definitions stored in a repository

Consult your MOM vendor’s documentation for information about any of these features.

About exteNd’s JMS Component

The JMS Connect creates JMS Components which can be incorporated into exteNd services. Much like the XML Map Component, the JMS Component is designed to map, transform, and transfer data between incoming or outgoing messages and XML templates. It is specialized to make JMS calls into JMS-aware messaging systems; automatically fill out needed header information based on information you supply via a setup wizard; and handle details of packaging message contents according to constraints imposed by JMS.

Like any data-exchange operation, the JMS Component relies on a Connection Resource. The Connection Resource in turn specifies important information regarding ports, channels, user identity, password, queue location, and so forth. Once you've set up a JMS Connection Resource, you can use it to set up a JMS Component that sends messages to (or retrieves messages from) the queue specified in the resource.

2 Getting Started with the JMS Component Editor

As with other exteNd Connects, creating a usable JMS Component actually begins with creating a JMS Connection resource, via which communication can occur with a message queue or topic. You will also want to prepare any XML template documents (XML skeletons, DTDs, and/or XSL stylesheets) with which your component will work. Getting these items ready is the subject of this chapter.

Creating a JMS Connection Resource

Before you can make use of a JMS Component, you must create a Connection Resource to access the queue or topic your component will be sending messages to or receiving messages from.

Every Connect, including the JMS Connect, uses its own Connection Resource type. The different Connection Resources (for JDBC, JMS, ECI, etc.) require varying numbers and types of parameters, appropriate to the external data source in question. The setup wizard changes appearance dynamically to reflect this.

Once you create a Connection Resource, you can reuse it for various JMS components that you create, rather than creating a new connection each time. Also, a Connection Resource, once created, can to some degree be *self-configuring* in that its data fields can be linked to ECMAScript expressions that control the parameter values associated with the connection (see below).

About Expression-Driven Connections

The “Create a New Connection Resource” wizard will let you specify connection parameters in two ways: as Constants or as Expressions. By default, the wizard’s parameter-entry fields are *constant-based*, which means that the value you enter for any given parameter is utilized, unchanged, every time the connection is used. An *expression-based* parameter, by contrast, gets its value programmatically, at runtime, via an ECMAScript expression which you supply in the wizard at design time. The value of an expression-driven parameter, therefore, can be different each time a connection is used, depending on the conditions prevailing at runtime.

For example, one very simple use of an expression-driven parameter in a JMS Connection would be to define the Connection User name as a PROJECT Variable. (From Composer’s main menubar, choose **Tools**, then **Configuration**, then select the **Project Variables** tab.) Then you could assign the value of the PROJECT Variable to the Connection User parameter. This way, when you deploy the project, you use the “Project Variable Remapping Panel” feature of the Deployment Wizard to update the Connection User name to a value appropriate for the final production environment.

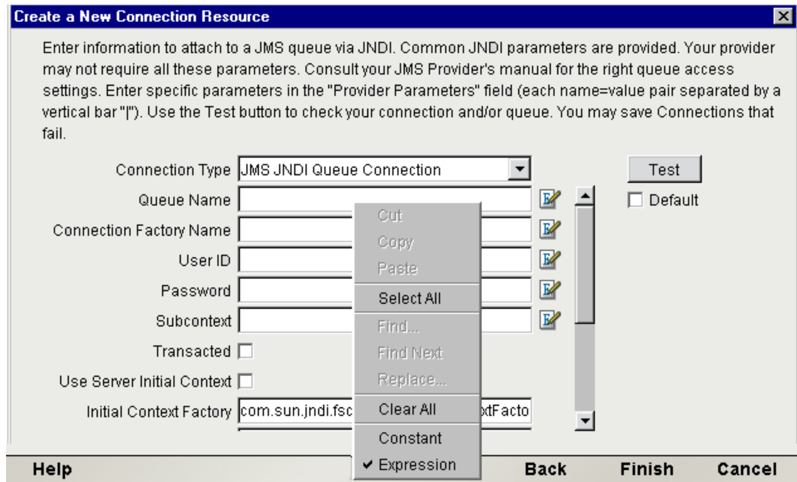
As another example, suppose Queue1 in your MOM environment is scheduled for maintenance on the 15th day of every month, in which case Queue2 should be used instead. You could assign an expression to the Queue Name parameter of the connection:

```
(new Date).getDate() == 15 ? "Queue1" : "Queue2"
```

You can also use an ECMAScript expression to read information from a file on disk, call a Java object in the Application Server, etc. Thus, the use of expressions to provide parameter information brings great flexibility and power.

➢ **To switch a parameter to Expression-driven mode**

1. Position the cursor in the field that you want to attach an expression to. (Note that this does not apply to the Connection Type field nor to checkboxes.)
2. Click the right mouse button to bring up a context menu.



3 Select **Expression** from the menu. A blue Expression Editor icon appears to the right of the parameter field.

4 Type an ECMAScript expression into the field, or click the Expression Editor button and use the pick lists in the Expression Editor to build an expression that evaluates to a valid parameter value at runtime.

About Queue Connections

In Message Oriented Middleware systems, queues are administered resources that JMS gains access to via its own *administered objects*. JMS administered objects encapsulate information about destinations and connections in such a way that client apps can use these objects through interfaces that remain portable.

JMS requires that administered objects (ConnectionFactories and Destinations) be placed in a JNDI namespace. Therefore, the connection resources needed by an application can always be obtained via JNDI. But if the name(s) of the provider's ConnectionFactory objects are known to a given Java application, that application can create its own connection(s) *without* going through JNDI.

exteNd offers a connection-via-JNDI facility by default (since JNDI access to administered objects is *guaranteed* to be available in every JMS MOM). But in the case of IBM's MQSeries, exteNd also offers the option of obtaining queue connections directly using MQSeries classes (which is to say, without going through JNDI). This offers the user easier setup options, with a more vendor-tailored user interface.

NOTE: If a vendor-specific “direct connection” facility is available, it will be listed as a pulldown-menu choice in exteNd Composer’s “Create a New Connection Resource” Wizard. You can also verify the availability of provider-specific connection facilities by checking the contents of <PROVIDERS> under the JMS <COMPONENT_FACTORY> node of your **xconfig.xml** file. (The **xconfig.xml** file is located in your **Composer/bin** directory).

➢ To create a JMS Queue Connection Resource Using JNDI

NOTE: The settings in the following graphics are typical for a JNDI connection to a JBrokerMQ destination.

1 Select **File>New>xObject**, then open the **Resource** tab and select **Connection**. The “Create a New Connection Resource” Wizard appears.

Create a New Connection Resource

A Connection resource is used to establish communications with an Connector data source or with a server using HTTP authentication. You need to create connections for each type of data source or each HTTP server you wish to communicate with. Enter a name and, optionally, a description for this Connection. The name will appear in the Composer Detail Pane and in choice lists when you are prompted for objects in Composer. The name may not contain the characters: \ / : ? " < > . | Names are case insensitive.

Name:

SampleJMS

Description:

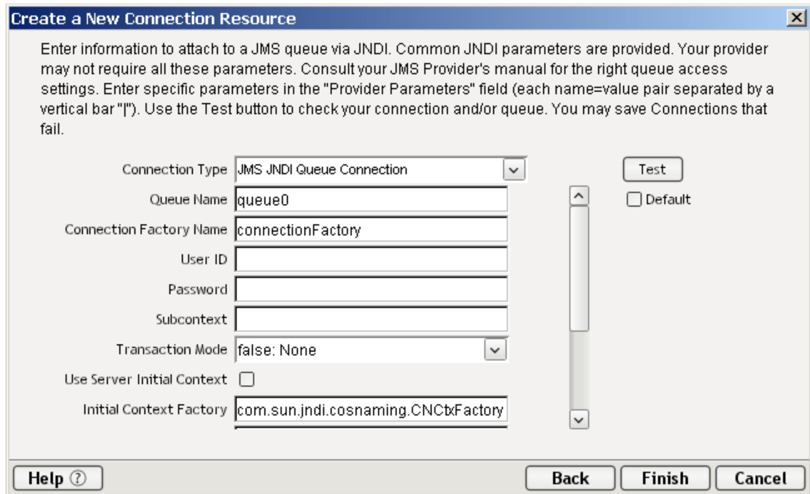
Purpose:
Input:
Output:
Remarks:

Help ⓘ Back Next Cancel

2 Type a **Name** for the connection object.

3 Optionally, type **Description** text.

4 Click **Next**.



5 Using the Connection Type pulldown menu, select **JMS JNDI Queue Connection** (for Point-to-Point messaging). The contents of the pane will update to reflect the setup information needed for the particular connection type you've chosen.

6 Enter the name of the queue you want to use in the first field (called **Queue Name**).

7 In the **Connection Factory Name** field, enter the QueueConnectionFactory if you are creating a queue connection, or TopicConnectionFactory if you are creating a topic connection.

8 Enter **Connection User** and **Connection Password** info, as applicable. (Optional)

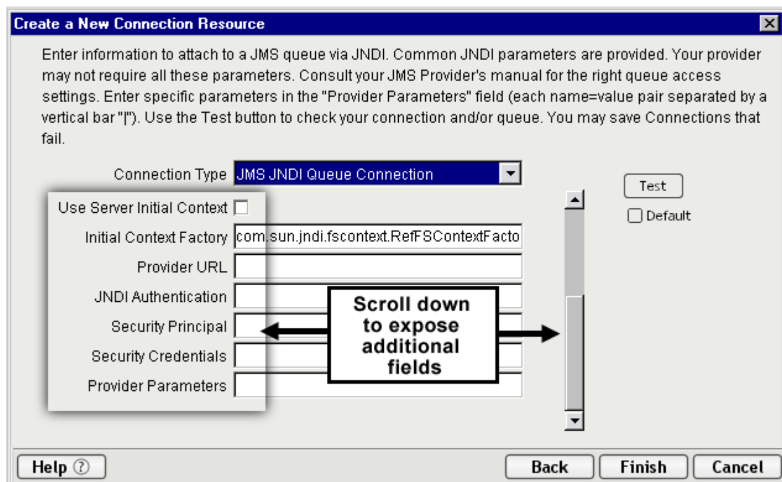
9 Enter the JNDI **Subcontext**, if needed. (Optional)

10 Under **Transaction Mode**, enter False: None, True: Local or XA: Server if you intend to issue session-level Commit or Rollback commands in your JMS Component.

NOTE: Issuing JMS Commit or Rollback statements in a JMS Component's action model *without* the proper Transaction Mode selected will result in exceptions being thrown.

11 Check the **Use Server Initial Context** box if you would like your service, once deployed, to obtain a ConnectionFactory locally, on the server, at runtime. This means you do not have to carry out steps 15 to 19 below in order for the deployed service to obtain a queue or topic connection. However, if you intend to send and receive live messages over the connection at *design* time, you should also complete all applicable steps below, because Composer needs to be able to find the ConnectionFactory objects it needs on the remote host. The following settings are aimed at helping Composer establish connections remotely.

12 Using the **vertical scrollbar** to the right of the text fields, scroll down to expose the remaining fields in the dialog. See below.



13 In the **Initial Context Factory** field, enter the name of your system's JNDI context factory, such as **com.sun.jndi.fscontext.RefFSContextFactory**. (Contact your administrator to obtain this information.)

14 In the **Provider URI** field, enter the URI representing the location of the JMS provider's (or MOM vendor's) JNDI context resources. For example, this may look something like **iiop://localhost:3506** or **file:///D/MQSeries/java/fscontext**.

15 (Optional) In the **JNDI Authentication** field, enter any required JNDI Authentication string (as provided by your administrator).

16 (Optional) In the **Security Principal** field, enter any required JNDI Security Principal's name (as provided by your administrator).

17 (Optional) In the **Security Credentials** field, enter any required JNDI Security Credential string (as provided by your administrator).

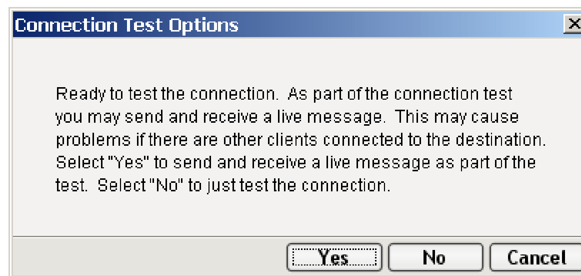
18 (Optional) In the **Provider Parameters** field, enter any provider-specific name/value pairs that are necessary for the MOM environment in which you are operating. Separate name/value pairs by a pipe character (|). For example, parameters for an LDAP provider could be:

```
java.naming.security.authentication = value |
java.naming.security.credentials = value |
java.naming.security.principal = value
```

NOTE: Spaces are shown here for clarity. Do not use spaces in your provider-param string.

19 Check the **Default** checkbox (which is normally unchecked) if you would like this Connection Resource to appear in setup dialogs by default when you create JMS Components.

20 Click **Test** to see if your connection is successful. The Test Options dialog appears.



21 The Test Options dialog asks if you want to send a live message as part of the test of the connection's integrity. Clicking the **Yes** button causes Composer to send a live message (of type *TextMessage*, with a unique CorrelationID) to the queue or topic for which you're establishing a connection.

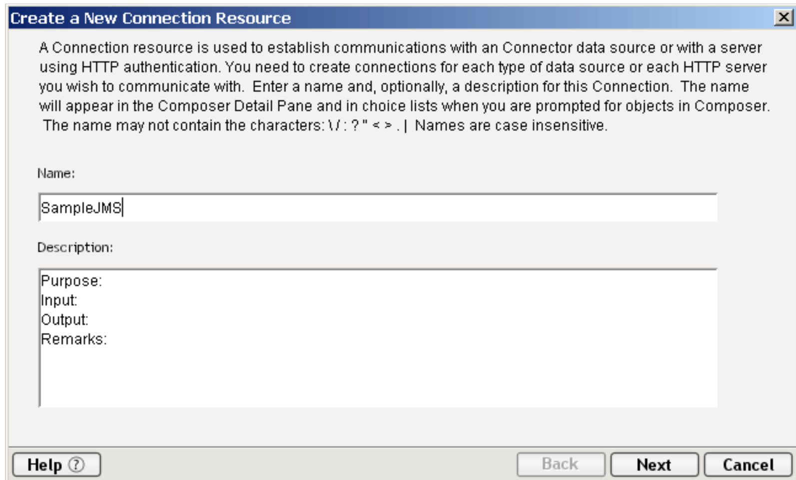
NOTE: Use care not to send this test message in a production environment (i.e., using a live queue, with potentially many listeners) unless you are reasonably certain that any existing applications in that environment won't be adversely affected.

Click **No** if you wish to create the necessary connection objects but not send any test message.

22 Click **Finish**. The newly-created connection resource xObject appears in Composer's Connection Resource detail pane.

➢ To create an MQSeries Queue Connection Resource

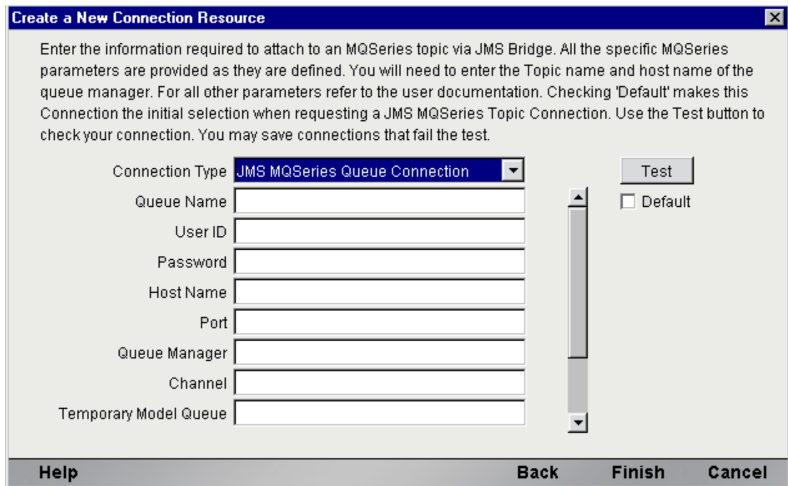
1 Select **File>New>xObject**, then open the **Resource** tab and select **Connection**. The "Create a New Connection Resource" Wizard appears.



2 Type a **Name** for the connection object.

3 Optionally, type **Description** text.

4 Click **Next** to go to the Connection Info pane of the Wizard.



5 Using the Connection Type pulldown menu, choose **JMS MQSeries Queue** (for Point-to-Point messaging). The contents of the pane will update to reflect the setup information needed for the particular connection type you've chosen.

6 Enter the name of the queue you want to use in the first field (called **Queue Name**).

7 Optionally enter a username in the **Connection User** field.

8 Optionally enter your password in the **Connection Password** field.

9 In the **Host Name** field, enter the name of your system's MQSeries Host Machine Name. (Contact your administrator, if need be, for this information.)

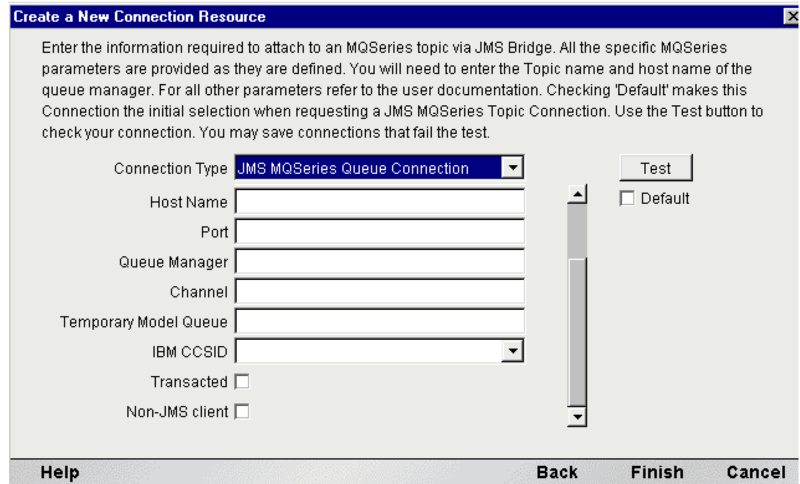
10 In the **Port** field, enter the MQSeries Host Machine Port Number. (Contact your administrator, if need be, for this information.)

11 In the **Queue Manager** field, enter the MQSeries Queue Manager name for this queue (as provided by your administrator).

12 In the **Channel** field, enter the MQSeries Host Machine Channel Name (as provided by your administrator).

13 If you want to specify a temporary model queue, do so in the **Temporary Model Queue** field.

14 Using the **vertical scrollbar** provided, scroll down to expose the remaining fields of the dialog. (In this case, two checkboxes are exposed.) See below.



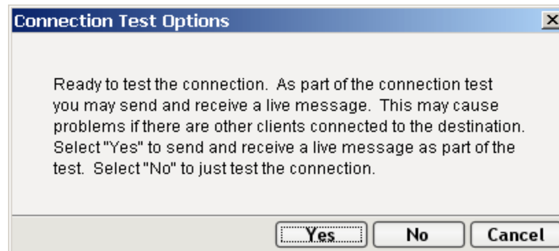
15 Under **Transaction Mode**, enter False: None, True: Local or XA: Server if you intend to issue session-level Commit or Rollback commands in your JMS Component.

NOTE: Issuing JMS Commit or Rollback statements in a JMS Component's action model *without* the proper Transaction Mode selected will result in exceptions being thrown.

16 Check the **Non-JMS Client** checkbox if you wish the connection to be set up using MOM-native facilities. (In this case, this means that MQSeries objects will be obtained directly, using vendor-proprietary calls.)

17 Check the **Default** checkbox (which is normally unchecked) if you would like this Connection Resource to appear in setup dialogs by default when you create JMS Components.

18 Click **Test** to see if your connection is successful. The Test Options dialog appears.



19 The Test Options dialog asks if you want to send a live message as part of the test of the connection's integrity. Clicking the **Yes** button causes Composer to send a live message (of type *TextMessage*, with a unique CorrelationID) to the queue or topic for which you're establishing a connection.

NOTE: Use care not to send this test message in a production environment (i.e., using a live queue, with potentially many listeners) unless you are reasonably certain that any existing applications in that environment won't be adversely affected.

Click **No** if you wish to create the necessary connection objects but not send any test message.

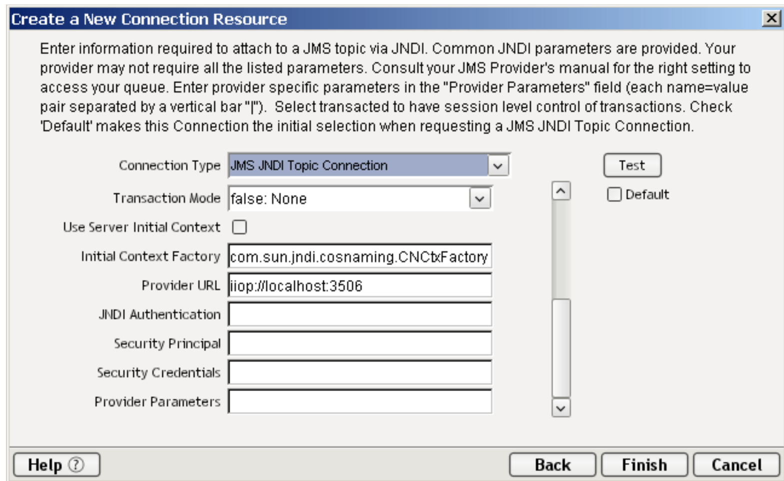
20 Click **Finish**. The newly-created connection resource xObject appears in Composer's Connection Resource detail pane.

About Topic Connections

When a queue is used in a Publish/Subscribe context (see "What Is Publish/Subscribe Messaging?"), it is called a *topic*. The differences between queues and topics are primarily administrative rather than functional. Therefore, all of the comments regarding Queue Connections in the preceding section apply equally to Topic Connections as well, except for the fact that in a Publish/Subscribe context, browsing is not defined. If you need to use a Browse action, you *must* connect to a queue, not a topic.

➢ To create a JNDI Topic Connection Resource

1. Select **File>New>xObject**, then open the **Resource** tab and select **Connection**. The "Create a New Connection Resource" Wizard appears.
2. Follow **Steps 2, 3,** and **4** for filling out the first panel in the wizard as described further above under "To create a JMS Queue Connection Resource Using JNDI" (page 30).
3. Click **Finish** on the first panel of the wizard. A new panel appears:
4. Using the Connection Type pulldown menu, select **JMS JNDI Topic Connection** (for Pub/Sub messaging). The contents of the pane will update to reflect the setup information needed for this connection type.
5. Enter the name of the Topic in the first field (called **Topic Name**).
6. Enter a **Durable Subscriber** name. (Optional)
7. Enter a **Client ID**.
8. In the **Connection Factory Name** field, enter the TopicConnectionFactory name.
9. Enter **Connection User** and **Connection Password** info, as applicable. (Optional)
10. Enter the JNDI **Subcontext**, if needed. (Optional)
11. Check the **No Local Messages** checkbox if you want to keep your component or service from receiving messages that it may be sending to a topic that it is listening on. (In other words, check this box if you want to keep a component from receiving its own messages.)
12. Scroll down to expose the rest of the panel's fields. See below.



13 Under **Transaction Mode**, enter False: None, True: Local or XA: Server if you intend to issue session-level Commit or Rollback commands in your JMS Component.

NOTE: Issuing JMS Commit or Rollback statements in a JMS Component's action model *without* the proper Transaction Mode selected will result in exceptions being thrown.

14 Check the **Use Server Initial Context** box if you would like your service, once deployed, to obtain a ConnectionFactory locally, on the server, at runtime. This means you do not have to carry out steps 15 to 19 below in order for the deployed service to obtain a queue or topic connection. However, if you intend to send and receive live messages over the connection at *design* time, you should complete all applicable steps below, because Composer needs to be able to find the ConnectionFactory objects it needs on the remote host. The following settings are aimed at helping Composer establish connections remotely.

15 In the **Initial Context Factory** field, enter the name of your system's JNDI context factory, such as **com.sun.jndi.fscontext.RefFSContextFactory**. (Contact your administrator to obtain this information.)

16 In the **Provider URI** field, enter the URI representing the location of the JMS provider's (or MOM vendor's) JNDI context resources. For example, this may look something like **iiop://localhost:3506** or **file:///D/MQSeries/java/fscontext**.

17 (Optional) In the **JNDI Authentication** field, enter any required JNDI Authentication string (as provided by your administrator).

18 (Optional) In the **Security Principal** field, enter any required JNDI Security Principal's name (as provided by your administrator).

19 (Optional) In the **Security Credentials** field, enter any required JNDI Security Credential string (as provided by your administrator).

20 (Optional) In the **Provider Parameters** field, enter any provider-specific name/value pairs that are necessary for the MOM environment in which you are operating. Separate name/value pairs by a pipe character (|). For example, parameters for an LDAP provider could be:

```
java.naming.security.authentication = value |
java.naming.security.credentials = value |
java.naming.security.principal = value
```

NOTE: Spaces are shown here for clarity. Do not use spaces in your provider-param string.

21 Check the **Default** checkbox (which is normally unchecked) if you would like this Connection Resource to appear in setup dialogs by default when you create JMS Components.

22 Optionally click the **Test** button to test the connection.

23 Click Finish to exit the wizard.

➢ To create an MQSeries Topic Connection Resource

1 Select **File>New>xObject** then open the **Resource** tab and select **Connection**. The "Create a New Connection Resource" Wizard appears.

2 Follow **Steps 2, 3,** and **4** for filling out the first panel in the wizard as described further above under "To create a JMS Queue Connection Resource Using JNDI".

3 Click **Finish** on the first panel of the wizard. A new panel appears:

Create a New Connection Resource

Enter the information required to attach to an MQSeries queue via JMS Bridge. All the specific MQSeries parameters are provided as they are defined. You will need to enter the Queue name and host name of the queue manager. For all other parameters refer to the user documentation. Checking 'Default' makes this Connection the initial selection when requesting a JMS MQSeries Topic Connection. Use the Test button to check your connection. You may save connections that fail the test.

Connection Type: JMS MQSeries Topic Connection
Topic Name
Durable Subscriber
Client ID
User ID
Password
Host Name
Port
Queue Manager

Test
Default

Help Back Finish Cancel

4 Enter the name of the Topic in the first field (called **Topic Name**).

5 Enter a **Durable Subscriber** name. (Optional)

6 Enter a **Client ID**. (Optional)

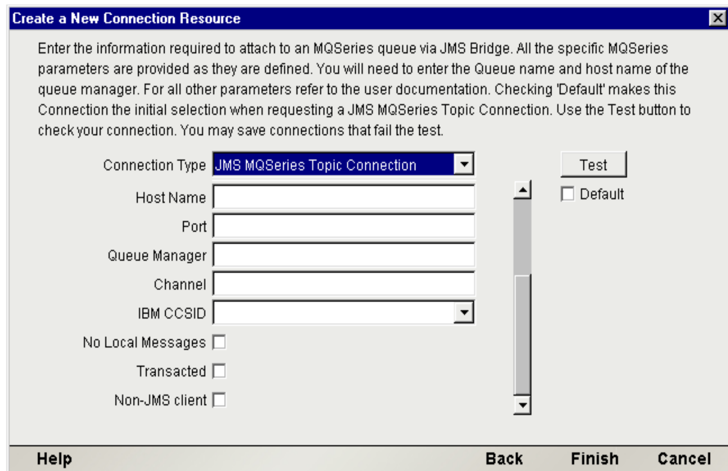
7 Enter **User ID** and **Password** info, as applicable. (Optional)

8 Enter the **Host Name**. (Optional)

9 Enter the **Port**. (Optional)

10 Enter the MQSeries **Queue Manager** name, if you want to specify one. (See your MQSeries documentation for advice on when and when not to enforce queue-manager associations.)

11 Scroll down to expose the rest of the fields in this panel of the wizard. See below.



12 Optionally enter a **Channel**.

13 Check the **No Local Messages** checkbox if you want to keep your component or service from receiving messages that it may be sending to a topic that it is listening on. (In other words, check this box if you want to keep a component from receiving its own messages.)

14 Check the **Transacted** checkbox (which is unchecked by default) if you intend to issue session-level Commit or Rollback commands in your JMS Component.

NOTE: Issuing JMS Commit or Rollback statements in a JMS Component's action model *without* the Transacted checkbox being checked will result in exceptions being thrown.

15 Check the **Non-JMS Client** checkbox if you will be using provider-native message delivery on this topic (ignoring JMS header information). This option would be of interest if your component were sending messages intended for a receiver that was using only MQSeries-native (non-JMS-aware) messaging features.

16 Check the **Default** checkbox (which is normally unchecked) if you would like this Connection Resource to appear in setup dialogs by default when you create JMS Components.

17 Optionally click the **Test** button to test the connection.

18 Click **Finish** to exit the wizard.

Creating XML Templates for Your Component

In addition to a connection resource, a JMS Component may also use XML stub documents, associated DTDs, and/or XSL stylesheets to aid in the mapping of message information. If you intend to use such documents, you should add them to an XML Template resource at this time so that you have sample documents for designing your component. (See Chapter 5, *Creating XML Templates,* in the *exteNd Composer User's Guide* for more information.)

Also, if your component design calls for any other xObject resources such as custom scripts or code table maps, it is best to create these before creating the JMS Component. For more information, see *Creating Custom Scripts* in the *Composer User's Guide*.

3 Creating a JMS Component

This chapter outlines the process of creating a JMS Component for use in an exteNd web service. At the same time, the semantics and usage of message header fields and properties are discussed in the context of JMS messaging; and tips are given for making the most effective use of JMS-based exteNd web services. You should familiarize yourself with this chapter before creating and deploying web services that rely on components created with the JMS Connect.

Before Creating a JMS Component

Creating a JMS component requires that you know the answers to the following questions:

- Which XML template documents (and/or COBOL copybooks) will you to need in order to map data into or out of messages? (For more information on XML Template resources, see *Creating a New XML Template* in the *Composer User's Guide*.)

- Which JMS Connection resource will your component(s) use? As part of the process of creating a JMS component, you can select an existing JMS connection, or you can create a new one. (If you create the connection beforehand, then it is available to all new JMS components.) See the previous chapter for step-by-step information on how to set up a JMS Connection resource.

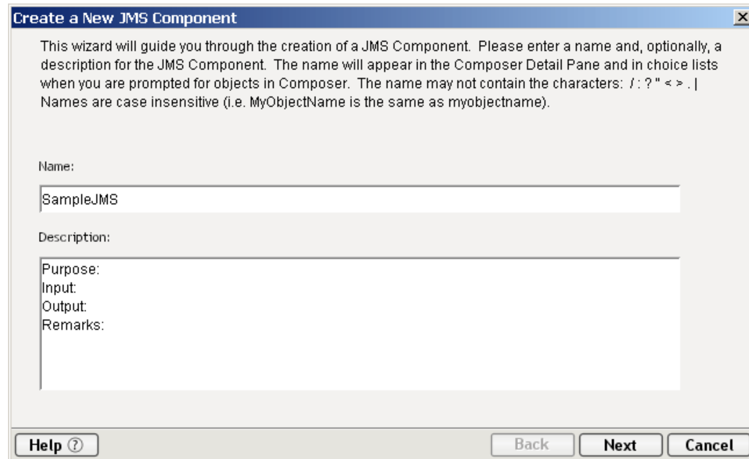
- Will you be creating a Browse Messages action? (See "The Browse Messages Action".) If so, you will need to choose a *queue* connection as your connection resource. Browsing is not defined on *topic* connections.

- Will your service be triggered by messages arriving at a queue/topic? If so, you will need to deploy your service as a JMS Service. (See Chapter 6, "The JMS Service".)

- Will your message session be transacted? If you intend to issue Commit or Rollback commands, you must enable transactions in the particular JMS Connection Resource you intend to use (by checking the “Transacted” checkbox in the Connection Resource setup wizard). For more information, see the discussion of the “Transacted” checkbox on page 33.
- Will you be *sending* messages, or *receiving* them? You can send as well as receive messages inside of a single JMS Component—but only if the same queue (or connection resource) is used. *If you will be sending or receiving to/from two or more different queues or topics, you must create separate components: one for each connection resource.*

➢ To create a new JMS Component:

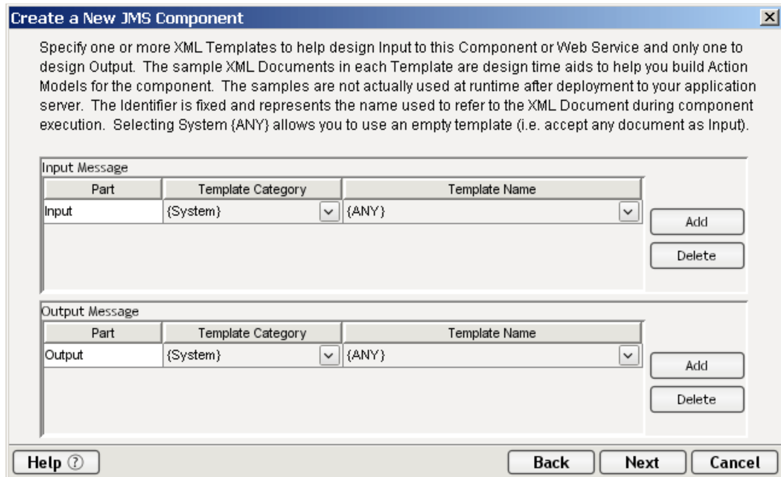
1 Select **File>New>xObject** then open the **Component** tab and select **JMS**. The “Create a New JMS Component” Wizard appears.



2 Enter a **Name** for the new JMS Component.

3 Optionally, type **Description** text.

4 Click **Next**. The XML Templates Info panel of the “Create a New JMS Component” Wizard appears.



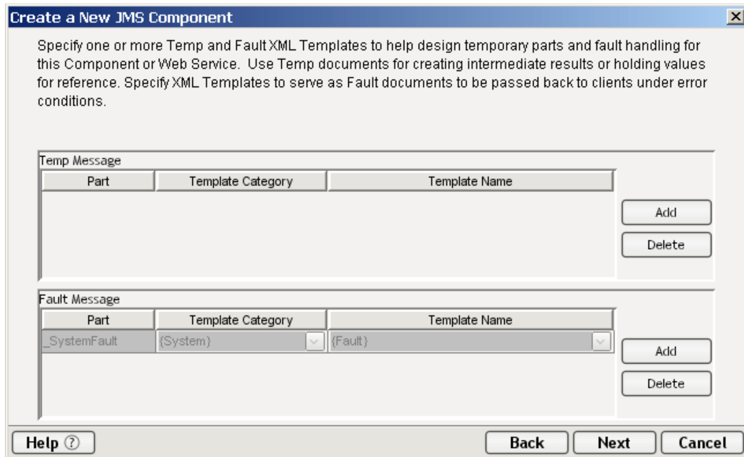
5 Specify the Input and Output templates as follows.

- Type in a name for the template under **Part** if you wish the name to appear in the DOM as something other than “Input”.
- Select a **Template Category** if it is different than the default category.
- Select a **Template Name** from the list of XML templates in the selected **Template Category**.
- To add additional input XML templates, click **Add** and choose a **Template Category** and **Template Name** for each.
- To remove an input XML template, select an entry and click **Delete**.

6 Select an XML template for use as an Output Part using the same steps outlined above.

NOTE: You can specify an input or output XML template that contains no structure by selecting {System}{ANY} as the Input or Output template. For more information, see “Creating an Output Part without Using a Template” in the User’s Guide.

7 Click **Next**. The XML Temp/Fault Template Info panel of the New HP3000 Terminal Component Wizard appears.

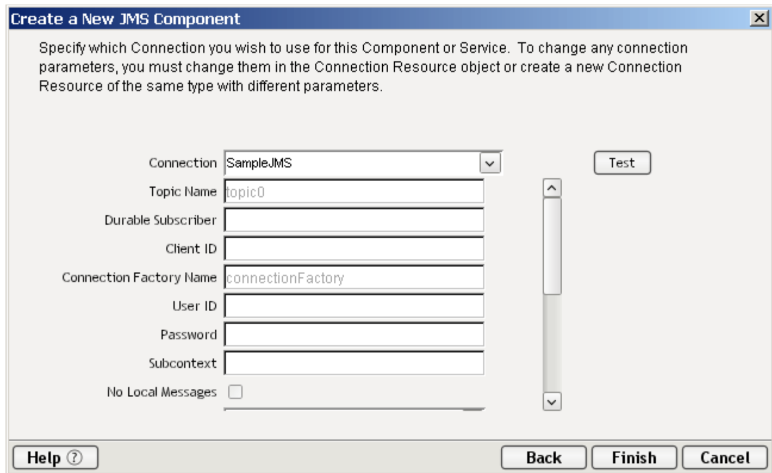


8 If desired, specify a template to be used as a scratchpad under the “Temp Message” pane of the dialog window. This can be useful if you need a place to hold values that will only be used temporarily during the execution of your component or are for reference only. Specify the templates as indicated in Step 6 above.

9 Under the “Fault Message” pane, select an XML template to be used to pass back to clients when an error condition occurs.

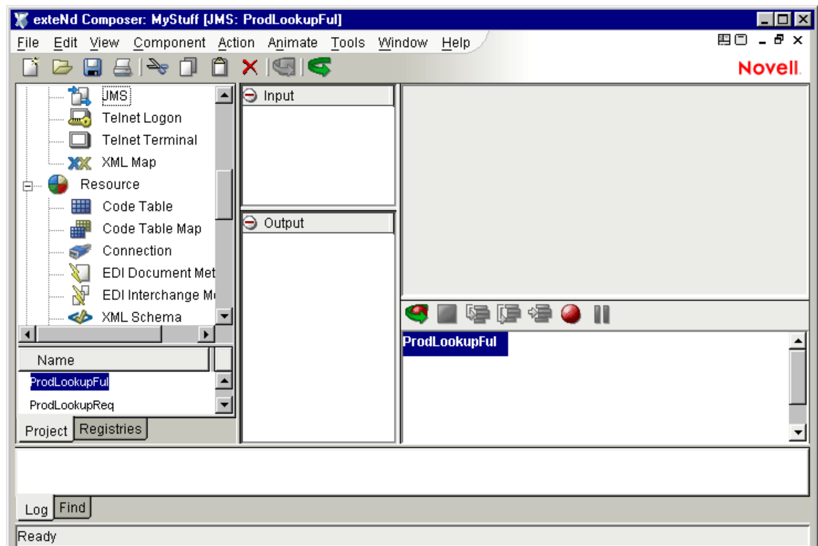
10 As above, to add additional temp or fault XML templates, click **Add** and choose a Template Category and Template Name for each. Repeat as many times as desired. To *remove* an XML template, select an entry and click **Delete**.

11 Click **Next.** The Connection Info panel of the “Create a New JMS Component” Wizard appears.



12 Select a **Connection** type from the pulldown list. (The pulldown list choices reflect the available JMS Connection Resources that were created earlier. For more information on creating JMS Connection Resources, see "Creating a JMS Connection Resource" on page 27).

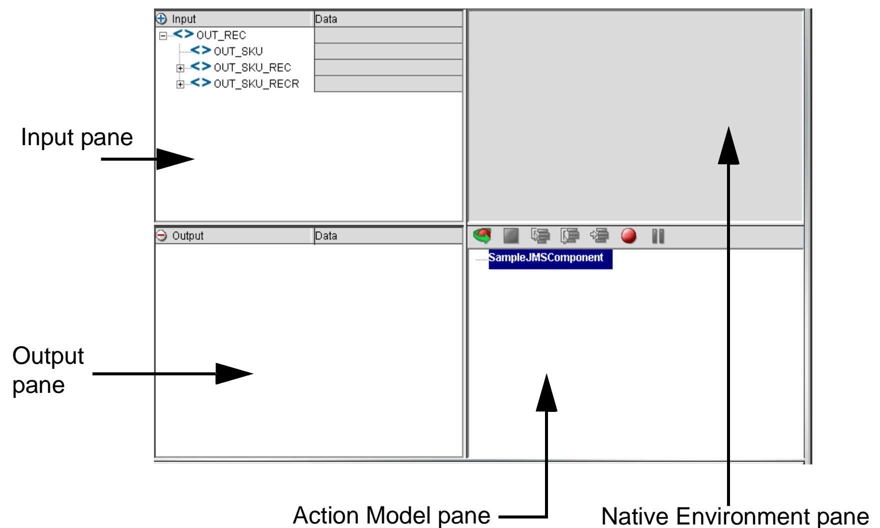
13 Click **Finish**. The component is created and the JMS Component Editor appears.



About the JMS Component Editor Window

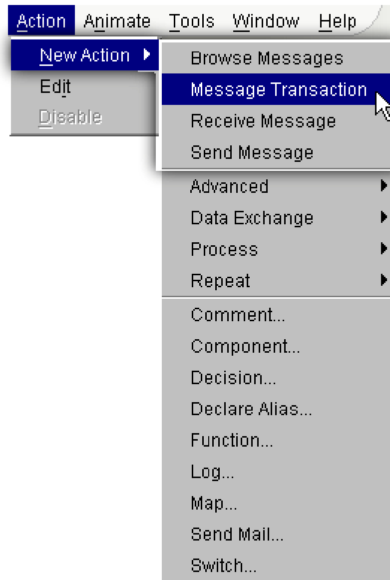
The JMS Component Editor is similar in appearance to the XML Map Component Editor window (and in fact includes all the functionality of the XML Map Component Editor, plus additional Action types specific to messaging). Like all other Component Editors, the JMS Component Editor window includes an Action Model pane (which is typically in the lower right corner, although it doesn't have to be), a Native Environment pane (upper right), and mapping panes for input, output, and/or temporary DOMs.

The Native Environment page appears as a grey pane until you create or highlight a Message Action, at which time it displays a message status pane containing either two or three tabs, depending on whether the current Message Action involves browsing, sending, or receiving.



If you activate the Action menu (or do a right-mouse-click inside the Action Model pane) and select New Action, you will see that all of the same actions that are available in the XML Map Component are also available in the JMS Component, but with four new action types:

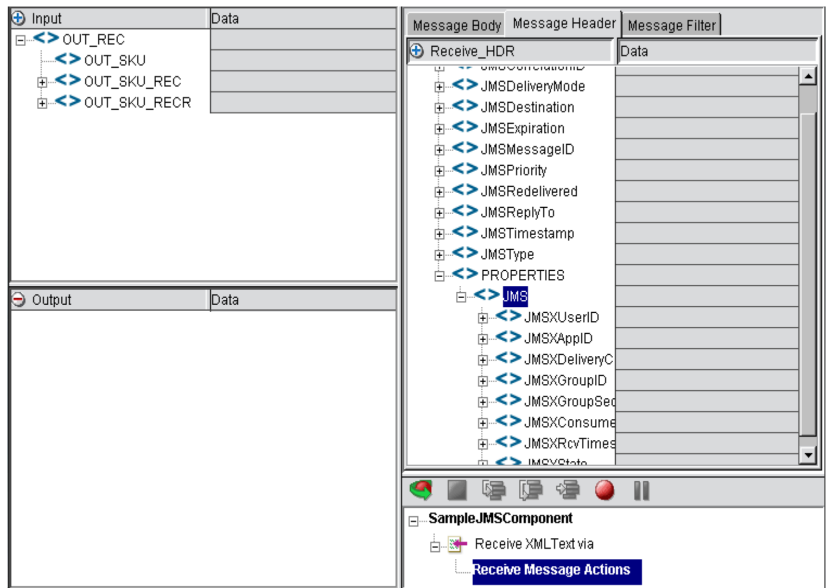
- Browse Messages
- Message Transaction
- Receive Message
- Send Message



The four JMS-specific actions are the subject of the next chapter.

About the Native Environment Pane

The JMS Component Editor's Native Environment pane (which is initially grey) will display various types of information associated with a message whenever a JMS Message Action is highlighted in the Action Model pane. The available categories of information (indicated by tabs at the top of the pane) include Message Body, Message Header, and Message Filter. The first two categories are common to all message actions (Send, Receive, and Browse). The Message Filter category, however, is visible only for Receive and Browse.



In the above illustration, the Message Header tab has been selected (with a Receive Message action highlighted in the Action Model pane). The Native Environment pane has been enlarged to show all available message headers and properties. For information on message header and property field usage, see “Message Headers and Properties” on page 131.

When the Message Body tab is selected, the Native Environment Pane will display content in a manner appropriate to the body type. For example, if the message contains an XML document, the Native Environment Pane will display a DOM tree; whereas if the message contains a COBOL copybook, it will display the contents of the copybook. See “Actions Unique to the JMS Component Editor”(starting on page 52) for additional information.

When the Message Filter tab is selected, the Native Environment Pane will display a selector-edit area. See “Working with Message Filters (Selectors)” on page 98 for more information.

4 Creating JMS Actions

About Actions

An *action* is similar to a programming statement in that it performs a specific, well-defined task, often with input in the form of parameters. Related actions are often chained together to form a single functional unit. In exteNd, this functional unit is the Component; actions that make up the Component are part of an *action list* or *Action Model*. (Please see the chapters in the *Composer User's Guide* devoted to Actions.)

The JMS Component Editor allows you to create actions that involve sending, receiving, or browsing messages, optionally as part of a *transaction*. The powerful XML mapping capabilities of exteNd Composer allows you to map XML information between messages and DOMs with ease, while also permitting the transformation of data with business logic. JMS Components created in exteNd Composer thus bring sophisticated messaging capabilities to XML integration applications.

Within a Component, an Action Model is made up of a list of related actions that work together to achieve a desired result. As an example, in a JMS Component, an Action Model might contain actions that read order data from a queue, map the data to a temporary XML document, perform data transformations on specific line items, and map the converted data to an output XML document.

The Action Model mentioned above would be composed of several discrete actions. These actions would:

- Perform a Read Message action, perhaps with the aid of filtering
- Map message contents to a temporary XML document
- Transform data items using a Code Table
- Optionally execute other Components
- Map the results to an Output XML document

Actions Unique to the JMS Component Editor

The JMS Component editor contains all the core functionality of exteNd Composer's XML Map Component editor, plus four connector-specific action types:

- Browse Message
- Message Transaction
- Receive Message
- Send Message

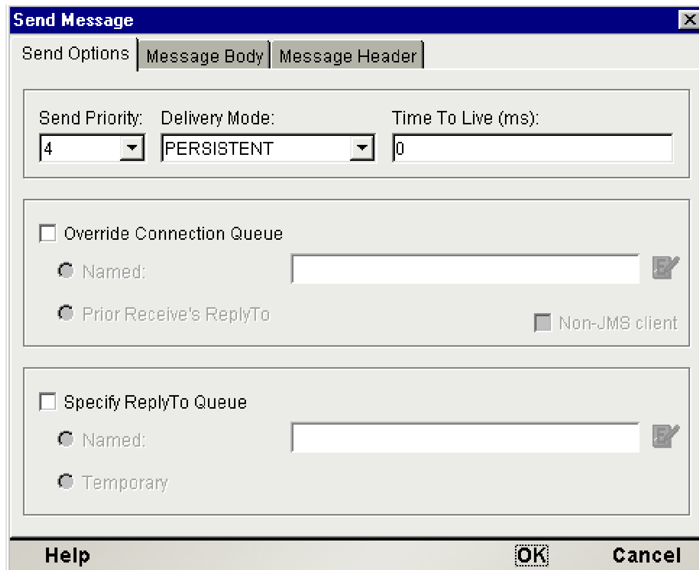
Except for the Message Transaction action type (which simply allows you to place Commit and/or Rollback statements in a JMS Component; see "The Message Transaction Action" further below), the various message actions all share a common setup dialog. The setup dialog has three tabs:

- An Options tab (labelled Send Options, Browse Options, or Receive Options, as appropriate to the action type)
- Message Body tab
- Message Header tab

The Browse and Receive dialogs also have a Filter tab (discussed under "The Browse Messages Action" and "The Receive Message Action" further below).

Options Tab

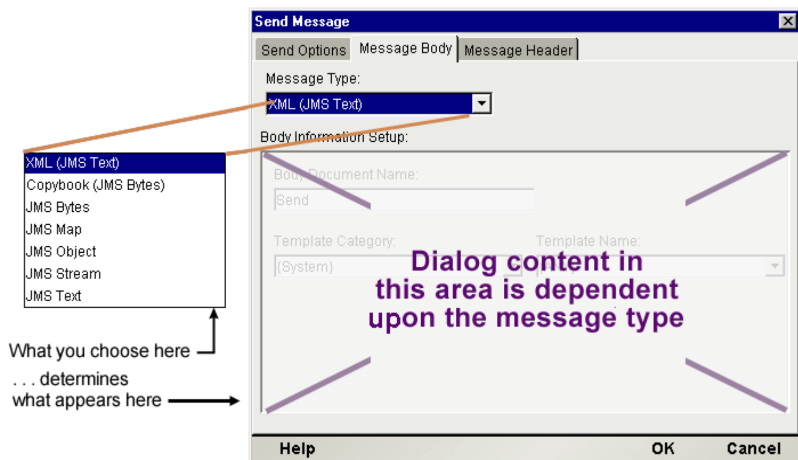
The Options tab exposes options specific to the type of action in question (Send, Browse, or Receive). For example, the Send Message action has an Options panel that looks like this:



The options shown in this tab are *action*-specific, but not *message-type*-specific. That is to say, the appearance of this tab will not be different for a BytesMessage than for a MapMessage.

Message Body Tab

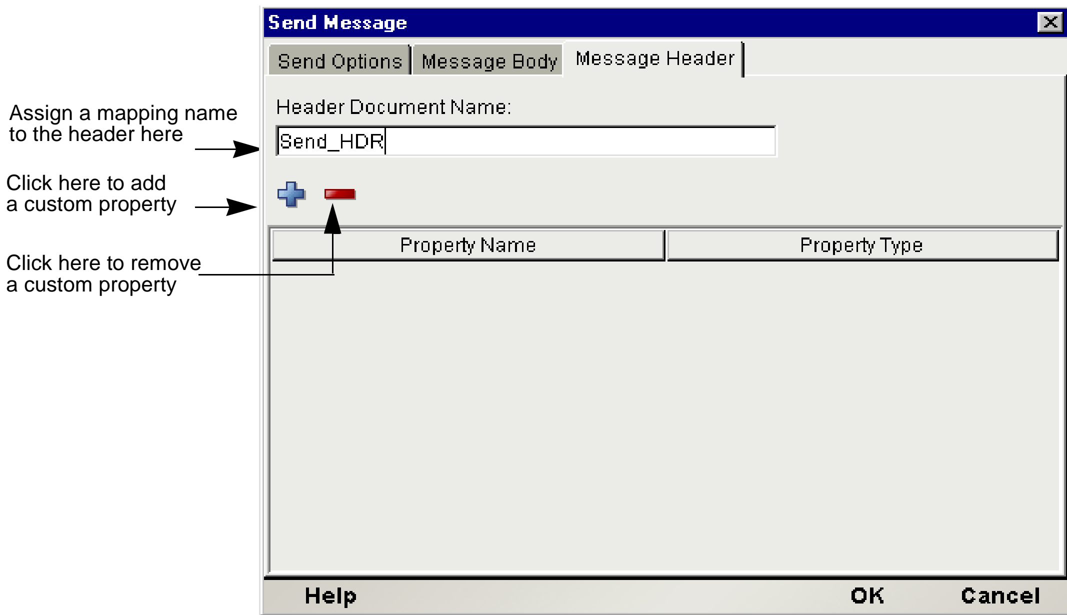
The **Message Body** tab brings up a pane containing setup parameters that differ depending on the message type:



This pane contains a **Message Type** pulldown menu as shown above, allowing access to the XML and Copybook types plus the five predefined JMS message types. Depending on what you select here, the Body Information Setup pane in the lower portion of the dialog will change. (See the sections that follow for a more complete description of the various fields and their usage.)

Message Header Tab

The **Message Header** tab brings up a pane that looks the same for all message action types:



This tab is where you can optionally create *custom properties* (equivalent to custom header fields) to supplement the built-in header fields defined by JMS.

Under Header Document Name, you must provide a name that can later be used as a target label for purposes of mapping *values* to *property fields* in the Native Environment pane. (Fields cannot be assigned values directly in this dialog.) The default name, in the case of the Send Message action, is Send_HDR.

The Send Message Action

The Send Message action can be used to post messages to a queue or topic. The message priority, delivery mode (persistent or non-persistent), and Time to Live can be specified on an action-by-action basis. In addition, you can specify the destination queue or topic on an action-by-action basis and you can optionally specify a named queue in the JMSReplyTo field of the outgoing message's header.

The button groups in the Send Options tab of the Send Message action require separate discussion. There are three button groups. The top group allows you to specify settings related to quality of service. The middle group allows you to specify a destination queue or different from the one you specified in the Connection Resource for the component. The bottom group is for letting you specify a "return address" for your outbound message. Each button group will be discussed in turn.

Priority, Mode, and Time to Live

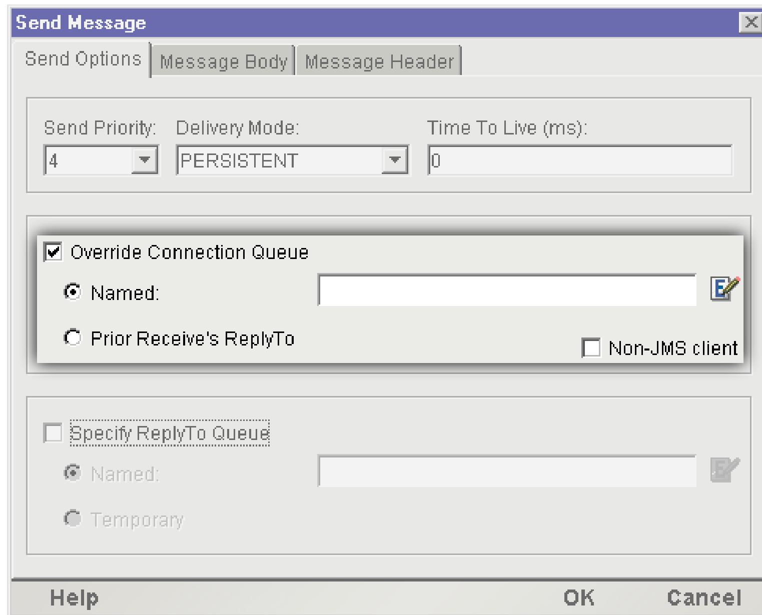
The top button group in the Send Options panel allows you to specify properties important to quality of service:

- **Send Priority**—Lets you assign a priority to the message, from one to nine. The JMS-defined default priority is four. Use the pulldown menu to override the default. (Note that the implementation details associated with message priority are not specified by the JMS standard.)
- **Delivery Mode**—Allows you to specify whether the message will be persisted to storage en route to its destination (for maximum reliability) or delivered more quickly but without recoverability.
- **Time to Live**—Lets you assign a maximum lifetime to the message, in milliseconds. If a zero value is specified, the message will not expire.

Destination Queue/Topic

The default destination queue for your Send Message action will be the one specified in the JMS Connection Resource for the component. (To change the default destination from within a JMS Component, choose **File > Component > Connection Info** and select a different queue from the pulldown menu.)

If you want to override the default behavior, you can specify destination queues or topics for Send actions on an action-by-action basis. To *override* the default behavior, simply check the **Override Connection Queue/Topic** checkbox in the Send Options panel of the Send Message dialog:



Checking the Override Connection Destination box (see above) enables two radio buttons underneath.

- The **Named** radio button allows you to enter a queue or topic name in the accompanying text field (as a quoted string) or specify the queue/topic using ECMAScript. The ability to specify a script here means that you can base the choice of queue or topic on custom logic involving conditionals and/or values obtained at runtime. To create a script that specifies the named queue, click the Expression icon to the right of the text field; this will bring up the Expression Editor dialog. Type your script in the Expression Editor (or build it with the aid of the picklists shown there) and click OK. Your script will appear in the text field. Obviously, the script must ultimately evaluate to a string representing the name of a particular queue or topic.

  NOTE: When you are using an IBM MQSeries queue, you can specify a fully qualified queue URI in the Use Named text field, such as:

  ```
  queue://qmanager/queue1?CLIENTTYPE=0
  ```

- The second radio button in the "Override" button group, **Prior Receive's ReplyTo**, will (if selected) cause the outgoing message to go to the queue or topic specified in the *JMSReplyTo* field of the *last received message having a non-empty JMSReplyTo field*. You would select this radio button if your Send Message action is specifically aimed at replying to an incoming message (a prior Receive Message action in the same component).

NOTE: The *JMSReplyTo* field that is referenced for purposes of this radio button is the last *JMSReplyTo* field associated with the last received message (within this component) *that has a non-empty JMSReplyTo field*. For example: If your component has three Receive Message actions—A, B, and C (in that order)—and the *JMSReplyTo* field was empty on B and C but non-empty for Message A, setting the **Use Prior Receive's ReplyTo** radio button will cause your Send Message action to use the queue specified by Message A. If *all three* messages were to have non-empty ReplyTo fields, and you wanted to reply to Message A, you would *not* set the **Use Prior Receive** radio button. Instead, you would select **Use Named** and specify an Expression that grabs the *JMSReplyTo* element from the header DOM of Message A.

MQSeries-Specific Behavior

In the Override Connection button group, a special **Non-JMS Client** checkbox will appear if your connection resource specifies an IBM MQSeries message queue. You can check this box if you are sending a message to a non-JMS message consumer, which is to say, a user of native MQSeries services. When this option is used, the assumption is that the receiving process is a non-Java message consumer that has no knowledge of (for example) JMS rules for header encoding and decoding. Therefore, when you use this option, you should not assume that any header information will be received by the message recipient, or that the recipient would know what to do with such information even if it was received. You should not, for example, specify user-defined header properties of any kind, nor map values into JMSCorrelationID nor JMSType header fields.

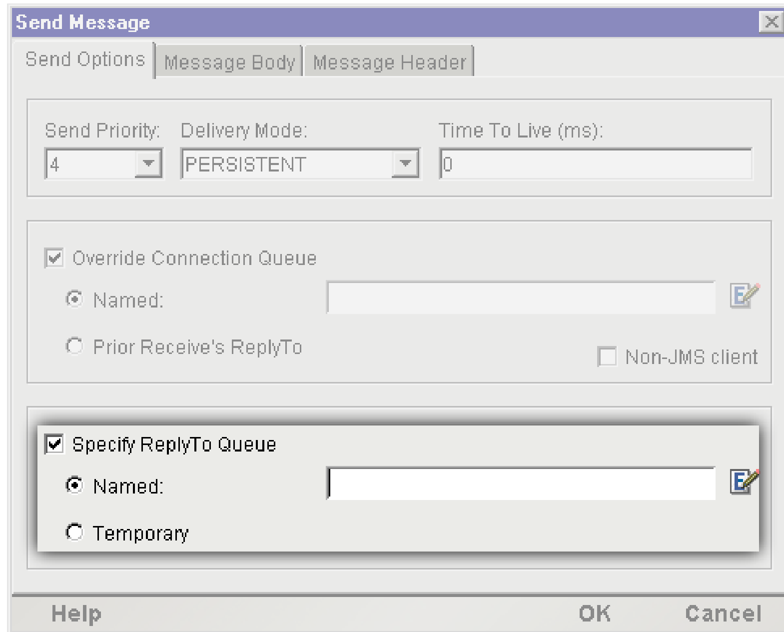
Notwithstanding the above, it's important to remember that the queue manager (which *is* JMS-aware) will still use certain header values even if the message is intended for a non-JMS client. In general, any header fields that have to do with quality of service (such as JMSDeliveryMode or JMSExpiration) will be honored by the queue manager, even if **Non-JMS Client** is checked.

Note that when you are using an IBM MQSeries queue, you can specify a fully qualified queue URI in the **Use Named** text field of the Override Connection Queue control group, such as:

```
queue://qmanager/queue1?CLIENTTYPE=0
```

Return Address

The lowermost button group in the Send Options panel gives you control over the value that the *JMSReplyTo* header field will have in your outgoing message.



By default, the *JMSReplyTo* field in an outgoing message is empty. You will typically change this behavior if you want your outgoing message to trigger a reply on the part of a receiver, or if your outgoing message *might* elicit a reply (such as an error report) at least some of the time. To override the default behavior (and specify a return address in the *JMSReplyTo* field), first check the **Specify ReplyTo** checkbox (see above), then select one of the two radio buttons underneath.

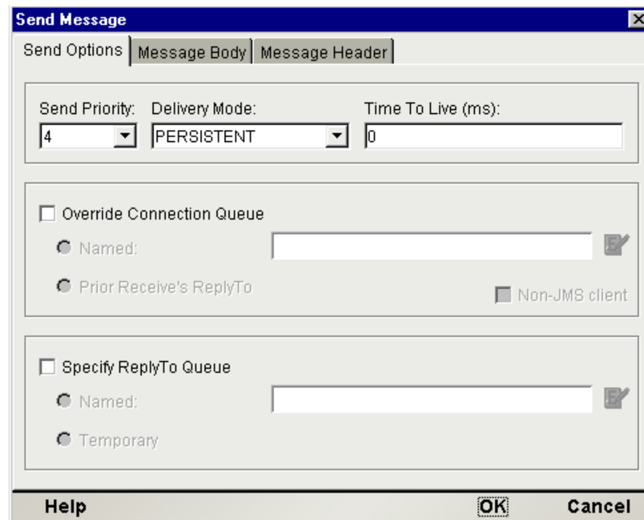
There are two possible reply scenarios: *asynchronous* and *synchronous*. (It's important to remember that in *neither* case can your component be *assured* of receiving a reply.) The radio buttons are meant to accommodate these two scenarios.

- The **Named** radio button implies an asynchronous scenario. When you select this radio button, you are specifying a queue or topic name in the outbound message's *JMSReplyTo* field. In essence, you are saying to the receiver: "If a reply is sent, be sure it goes to this address." You must type a queue name (or topic name) manually, enclosed in quotes, in the associated text field; or create an ECMAScript expression that evaluates to a queue/topic name.

- The **Temporary** radio button is designed to be used in cases where you expect to receive a reply in a synchronous manner. When you select this radio button, a temporary queue is created for purposes of receiving an immediate reply, and the outbound message has the temporary queue name in its *JMSReplyTo* header field so that the receiving process knows where to send its response. (The temporary queue will exist only for the lifetime of your component. See “Temporary Queues” in the next chapter.)

➢ To create a Send Message action

1. Create or open a JMS Component (as described in the previous chapter).
2. Highlight a line in the Action Model where you want to place the Send Message action. The new action will be inserted below the line you highlight.
3. From the **Action** menu, select **New Action**, then **Send Message**. The Send Message setup dialog appears, with the Send Options panel displayed.



4. From the **Send Priority** pulldown menu, select a priority ranking (from 0 to 9) for your message. Zero represents the lowest priority; 9, the highest.

   NOTE: The details of how this priority value is implemented are not defined by the JMS specification. Consult your MOM vendor’s documentation for details.

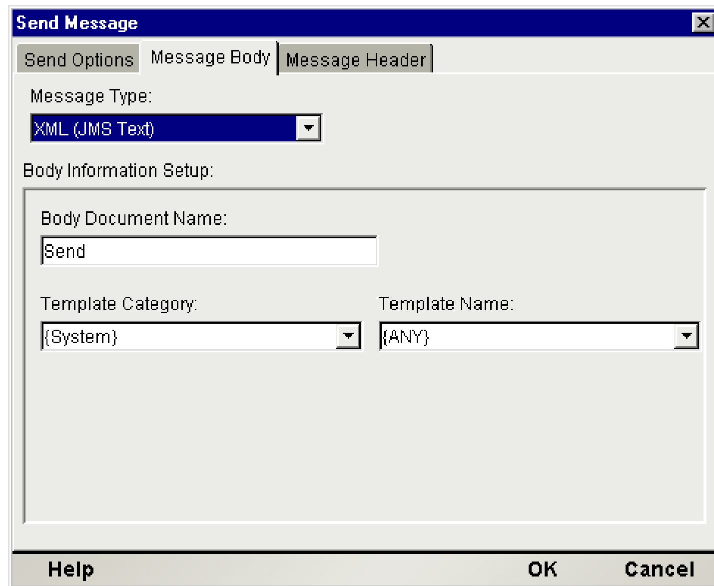
5. From the **Delivery Mode** pulldown menu, select PERSISTENT or NON_PERSISTENT. (See “What About Delivery Guarantees?” on page 19 for more information on the meaning of these terms.)

6 Enter a millisecond value in **Time To Live** to give your message a finite lifespan; or else enter zero, if you do not want your message to expire. (See “Message Headers and Properties” on page 131 for more information on this setting.)

7 If you wish to send your message to a queue or topic other than the one specified in your connection resource, click the **Override Connection Queue** checkbox and select the appropriate radio button as described in the discussion above under .

8 If you wish to specify a value in the JMSReplyTo field of your outgoing message’s header, click the Specify ReplyTo checkbox and select the appropriate radio button as described in the discussion above.

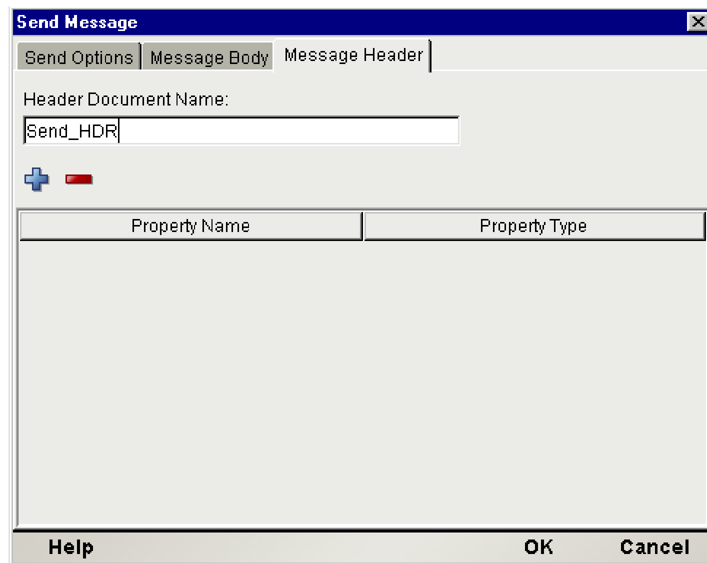
9 Click the Message Body tab. A new pane appears.



10 Select one of the seven available message types from the **Message Type** pulldown menu. (In this example, the XML type is selected, indicating that the posted message will contain an XML-formatted text document.)

11 In the Body Information Setup portion of the dialog, enter information as necessary. *This portion of the dialog will vary in appearance depending on the Message Type selected*. In this example, the Message Type is XML; hence, you are prompted to enter information for Body Document Name (i.e., the name you want to apply to the Message Body DOM), Template Category (the XML Template resource name), and Template Name (name of the XML stub document you want to apply to the Message Body, if appropriate). See "Using Other Actions in the JMS Component Editor" on page 79 for additional information.

12 If you would like to add your own custom Properties to the message header, click on the **Message Header** tab. The Message Header pane appears.



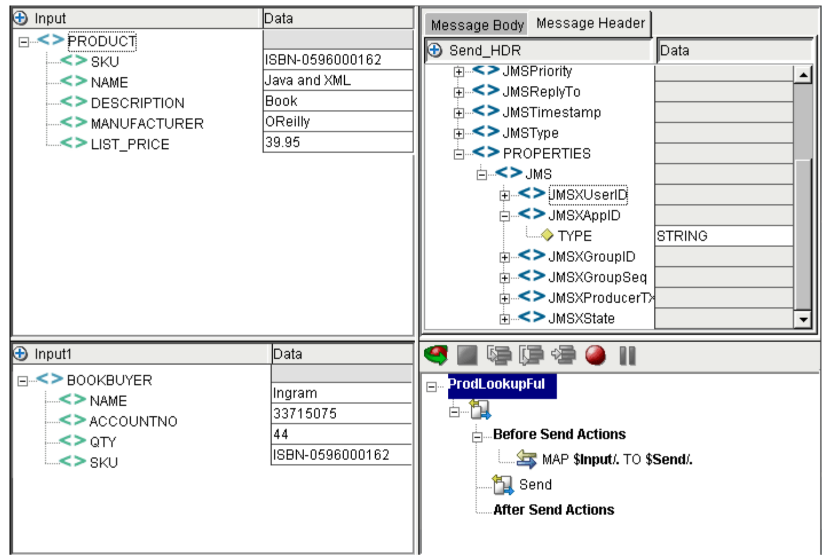
13 Enter a **Header Document Name** (or accept the default name, which begins with Send_HDR). This name will be shown in the Native Environment Pane as the name of the header tree, for purposes of mapping values to fields.

14 Click the plus (+) icon to add a Property. Type the name of the custom property under **Property Name** and click in the **Property Type** column to bring up a menu of available data types. Choose the data type appropriate to your property. In the above example, a single custom property called SKU_PREFIX has been created, specified as a String.

NOTE: You must go to the Native Environment Panel to specify *values* for your properties. This dialog merely creates the empty properties.

15 Click the plus (+) icon as many times as needed to add extra properties. Fill out the Name and Type information for each one.

16 Click **OK**. The JMS Component editor main window appears, containing the new Send action in the action list.



Note that when the Message Header tab is selected in the Native Environment pane (above), a User property called SKU_PREFIX is visible. This is the custom property we created in the setup dialog.

Before Send vs. After Send

When a Send Message action has been created, the action list shows “Before Send Maps” and “After Send Maps” lines. The reason there are two map lists is that some JMS-defined header fields are empty before the Send action is executed; after the Send, the same fields contain data supplied either by the JMS provider or by the JMS Component’s internal methods. In particular, the fields that are populated *after* Send time are:

- JMSDestination
- JMSDeliveryMode
- JMSExpiration
- JMSPriority
- JMSMessageID
- JMSTimestamp
- JMSRedelivered

Any data mapped into these header locations *prior* to sending a message will be overwritten at Send time. These fields should be considered read-only; and the data in them should be considered valid only *after* a Send.

The header fields that are writable are:

- JMSCorrelationID
- JMSType

You can double-click in these fields (in the Native Environment Pane) to enter data manually, or you can use them as drop targets in drag-and-drop mapping from DOMs.

NOTE: The *JMSReplyTo* field is writable, but not through drag-and-drop or direct editing. To populate this field, you must use the controls provided in the Send Options tab of the Send Message dialog.

The following graphic shows what the Native Environment pane might look like *after* a message has been sent (that is to say, after fields have been auto-populated):

Message Body | Message Header

Send_HDR	Data
MSGHEADER	
JMSCorrelationID	
JMSDeliveryMode	PERSISTENT
JMSDestination	queue:///clq_default_opal
JMSExpiration	Wed Dec 31 19:00:00 EST 1969
JMSMessageID	ID:414d5120514d5f6f70616c2e67656d6cc185a63913a0
JMSPriority	4
JMSReplyTo	
JMSTimestamp	Thu Dec 07 11:42:35 EST 2000
JMSType	
PROPERTIES	

If you wish to use the data in read-only fields for logging purposes, debugging, mapping to an Output DOM, etc., you should add the relevant Map actions *below* the “After Send Maps” line in the Action model. If you have input-DOM data that you wish to map to a header field (such as JMSCorrelationID), you can use the drag-and-drop method to create mappings between input DOM elements and header fields as described on page 81.

The Browse Messages Action

A *browse* operation allows your application to inspect messages from a queue without causing those messages to be removed from the queue. That is to say, after a browse operation, all messages are still available on the queue for any consuming application to obtain.

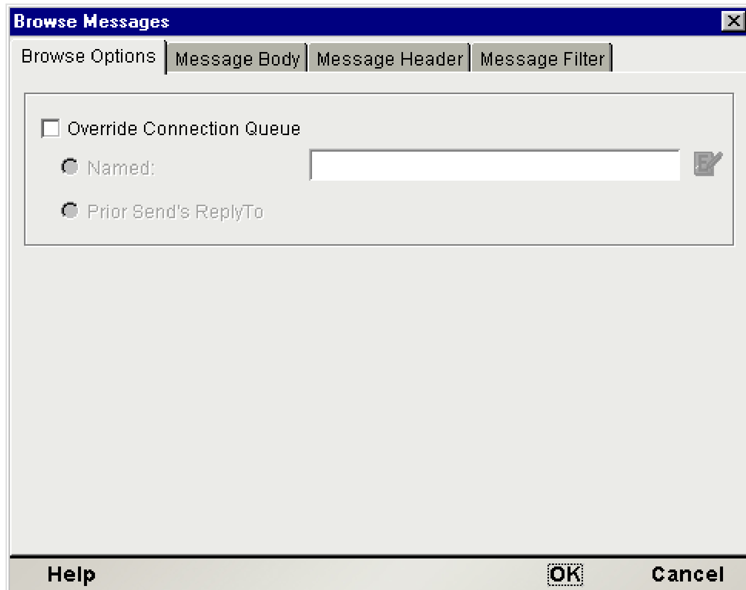
In response to a browse request, a queue manager will return a *java.util.Enumeration* containing *all* available messages, unless a Message Filter (or "selector") has been specified, in which case only those messages matching the selector statement will be returned. (For information on using selectors, see "Working with Message Filters (Selectors)" on page 98.)

In a JMS Component, the Browse Messages action is used to browse messages from a *queue*. By default, the queue that will be browsed is the one specified in the JMS Connection Resource for the component. (To change the *default* queue from within a JMS Component, choose **File > Component > Connection Info** and selected a different queue from the pulldown menu.) You can also override the default on an action-by-action basis (see below).

NOTE: Browsing is a Point-to-Point operation only. In Publish/Subscribe, browsing is not defined. Your connection resource should be configured to use a *queue* (not a topic) if you will be using the Browse Messages action.

➤ **To create a Browse Messages action**

1. Create or open a JMS Component (as described in the previous chapter).
2. Highlight a line in the Action Model where you want to place the Browse Messages action. The new action will be inserted below the line you highlight.
3. From the Action menu, select **New Action**, then **Browse Messages**. The Browse Message dialog appears, with the Browse Options panel displayed.

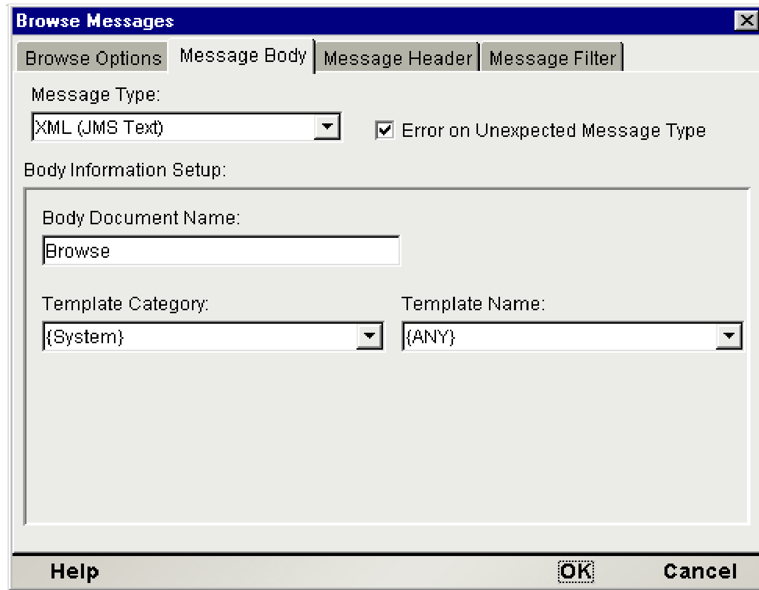


4 (Optional) If you would like to browse a message queue other than the one specified in your component's connection resource, check the **Override Connection Queue** checkbox (which is unchecked by default).

- Click the **Use Named** radio button if you wish to specify a queue name manually. (Type the queue name in the accompanying text field, surrounded in quotes, or build an ECMAScript expression that evaluates to a queue name.)
- Click the **Use Sent Message ReplyTo Field** radio button if you wish to browse a queue that was specified in the last Send Message action. (To obtain the destination queue name, exteNd will search for the last received message that had a non-empty *JMSReplyTo* field. This may or may not be the same as the last received message.)

5 If desired, specify a Filter expression in the text field under **Filter**. (See Appendix B, "Message Filter Syntax," for more information.) This is an ECMAScript expression, so be sure to wrap strings in quotes.

6 Click the **Message Body** tab. A new panel appears.



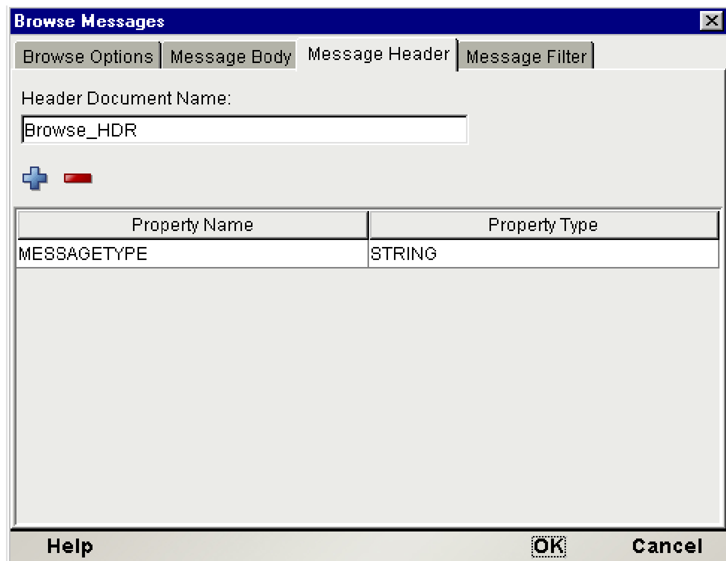
7 From the **Message Type** pulldown menu, select the message type that corresponds to the kind of message your component is designed to process. This will preconfigure the Message Body to receive and store incoming data in a format that's acceptable to your application's requirements.

NOTE: This choice *does not* tell exteNd Composer to filter out unwanted message types. A browse operation, under JMS, *always* returns an enumeration of all available messages, regardless of body type.

8 Uncheck the **Error on Invalid Message Type** checkbox (checked by default) if you do *not* want exteNd Composer to throw an exception when an unexpected message body type is encountered. Normally, a JMS process is designed to "understand" and handle one specific JMS message type (such as JMSText), which usually means that if an unexpected message type (such as JMSBytes instead of JMSText) is encountered, processing errors eventually occur. Usually, it is better for processing problems to be discovered earlier rather than later; hence, the default state of this checkbox is *checked*. It is quite possible, however, especially for testing purposes, that you might want your JMS Component to handle all available messages *regardless* of body type. In that case, you'd want to uncheck the box.

9 In the **Body Information Setup** portion of the dialog, enter whatever information might be appropriate to the Message Type. (This portion of the dialog will change in appearance according to the Message Type that you've chosen.) In this example, the Message Type is XML; hence, you are prompted to enter information for Body Document Name (i.e., the name you want to apply to the Message Body DOM), Template Category (the XML Template resource name), and Template Name (name of the XML stub document you want to apply to the Message Body, if appropriate). See "Using Other Actions in the JMS Component Editor" on page 79 for additional information.

10 Click the **Message Header** tab. A new panel appears.



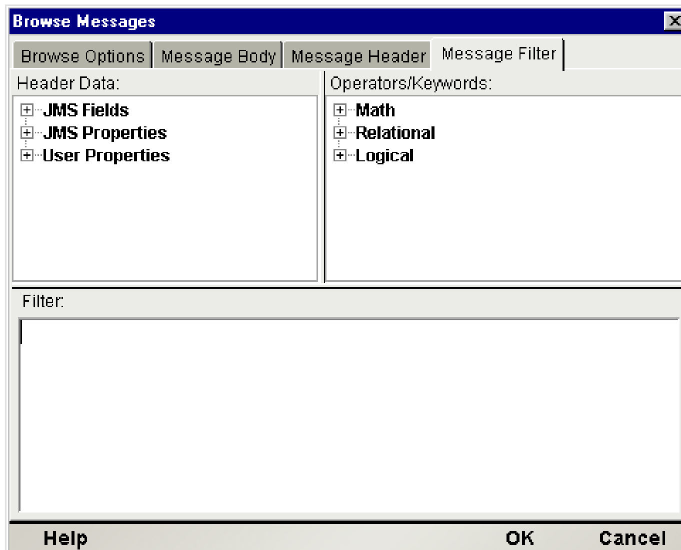
11 Enter a **Mapping Name** (or accept the default, which begins with Browse_HDR). This name will be shown in the Native Environment Pane as the name of the header tree, for mapping purposes.

12 Click the plus (+) icon to add a Property. Type the name of the custom property under **Property Name** and click in the **Property Type** column to bring up a menu of available data types. Choose the data type appropriate to your property.

NOTE: The purpose of this procedure is to build a property list that corresponds to the anticipated property list of the incoming message(s). If any incoming messages have property fields that aren't accounted for here, the extra fields will be ignored and any associated data will be lost.

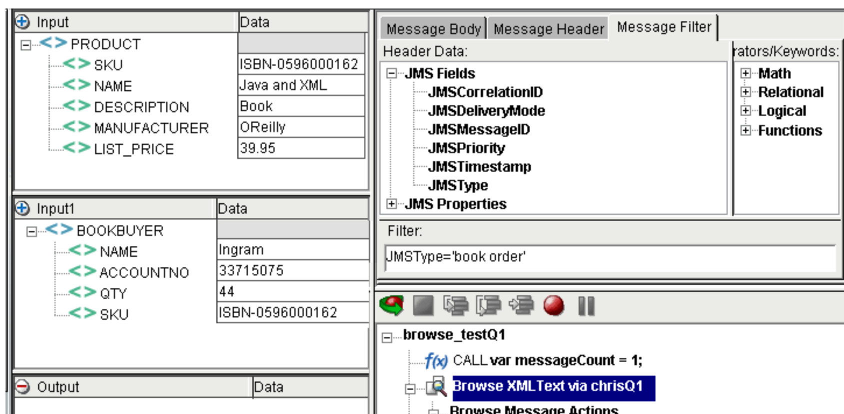
13 Click the plus (+) icon as many times as needed to add extra properties. Fill out the Name and Type information for each one.

14 Click the **Message Filter** tab. A new panel appears.



15 If you want to apply a JMS Message Filter to your Browse operation, enter it (and/or build it using the picklist items) in the text area in the lower half of the dialog. (See “Working with Message Filters (Selectors)” on page 98 for more information.)

16 Click **OK**. The JMS Component editor main window appears, with the new Browse action shown in the action list. (See illsutration.)



Iterating through Messages

When you create a new Browse action, the line “Browse Messages: Type = . . .” appears in the action list, followed by a line “Browse Message Maps,” followed (in turn) by a line that begins:

```
WHILE JMSMESSAGE.hasMessages() . . .
```

Because a JMS browse operation always returns a *list* of every available message (subject to filtering constraints; see “Working with Message Filters (Selectors)” on page 98), the JMS Connect automatically constructs a `WHILE` loop as part of every Browse Messages action. This loop iterates through each available message. As part of your action model, you can place whatever Map actions (or other processing) you might need, within the loop. You can also exercise loop control (using the component editor’s Break and Continue commands) as you would for any other loop.

Note that as with the Receive Message action (described below), the Native Environment pane for the Browse Messages action contains Message Body, Message Header, and Message Filter tabs. Working with these tabs is the subject of the next chapter.

The Receive Message Action

A *receive* operation allows your application to retrieve a message from a queue. The act of receiving a message causes that message to be destructively removed from the queue. The only exception to this occurs when a message session is *transacted* and a rollback takes place; in that instance, although a Receive Message action takes place, the message is ultimately not consumed. It remains on the queue.

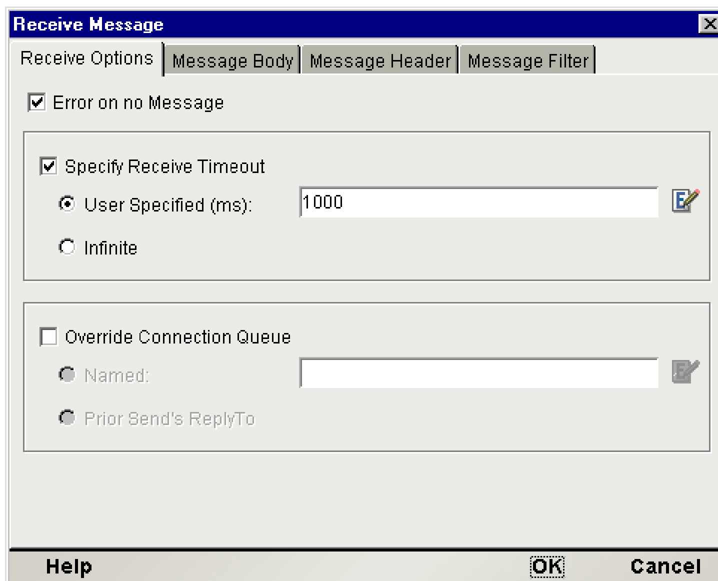
In response to a *receive* request, a queue or topic manager will return the first available message, except when a Message Filter (or “selector”) has been specified, in which case only the first available message *that matches the selector statement* will be returned.

In a JMS Component, the Receive Message action is used to obtain a message from a queue or topic. The queue or topic that will be used is the one specified in the JMS Connection Resource for the component. (To change queue/topic from within a JMS Component, choose **File > Component > Connection Info** and selected a different queue or topic from the pulldown menu.)

NOTE: Since the Receive Message action retrieves, at most, a single message at a time, you will need to construct a Repeat/While loop in order to receive *all* messages from a queue or topic. The Repeat While action should be designed in such a way that looping terminates when the *JMSMessageID* header field is empty.

➢ To create a Receive Message action

1. Create or open a JMS Component (as described in the previous chapter).
2. Highlight a line in the Action Model where you want to place the Receive Message action. The new action will be inserted below the line you highlight.
3. From the **Action** menu, select **New Action**, then **Receive Message**. The Receive Message dialog appears, with the Receive Options panel showing.



4. Check the **Error on No Message** checkbox (which is checked by default) if you want an exception to be thrown whenever no message is received within the timeout period. Note that the default timeout period (unless one is user-specified; see step 6) is NO_WAIT, or zero.

   NOTE: If this box is left checked, you should wrap your Receive action in a Try/On Error action and execute appropriate recovery steps in the On Error branch. But whether you check the checkbox or not, you should anticipate (and make sure your application can gracefully handle) the possibility that the queue is empty when your Receive action executes.

5. Check the **Specify Receive Timeout** checkbox (unchecked by default) if you wish to specify a timeout value other than the default of NO_WAIT (zero). If the checkbox is *not* checked, the action will simply check the queue or topic and retrieve a message if one is available, then return immediately without waiting. To force the action to block for a specified time, you must check this checkbox and specify a wait time.

- Select the **User Specified** radio button if you want to enter your own timeout value. Enter the timeout value (in milliseconds) in the accompanying text field, or create an ECMAScript expression that evaluates to a suitable number.

- Select the **Infinite** radio button if you want the Receive action to block indefinitely (until a message is received).

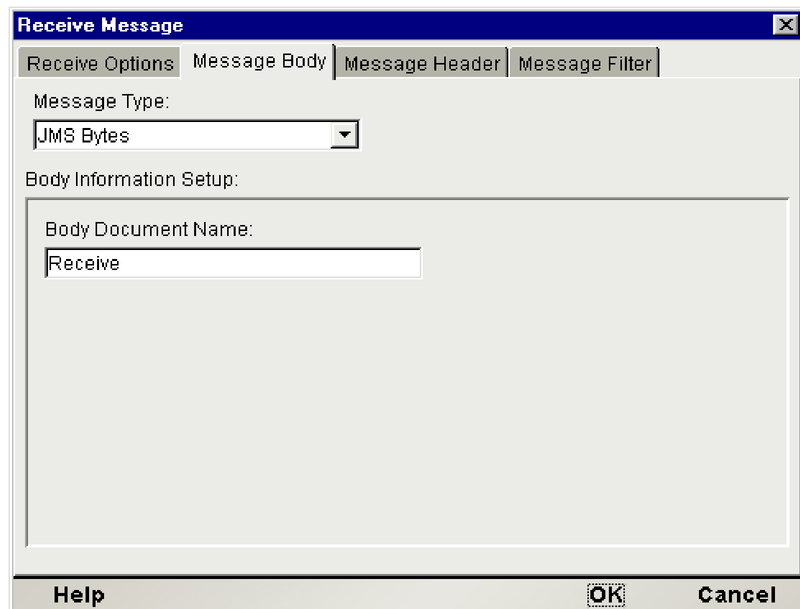
6 Check the **Override Connection Queue** checkbox (unchecked by default) if you want to specify a queue or topic other than the one given in the component's connection resource.

  - Select the **Use Named** radio button (the default) if you want to explicitly specify a given queue or topic. Type the queue or topic name in the accompanying text field, in quotation marks, or enter an ECMAScript expression that evaluates to a queue or topic name.

  - Select the **Use Prior Message ReplyTo Field** radio button if you want to use the queue or topic named in the *JMSReplyTo* field of the last sent message that contains a non-empty *JMSReplyTo* header field.

  NOTE: If your action is waiting for a reply to a previously sent message, you should have specified an appropriate timeout value in Step 6 above. Some testing may be required in order to determine the timeout value that provides the best ratio of safety to performance.

7 Click the **Message Body** tab. A new panel appears.



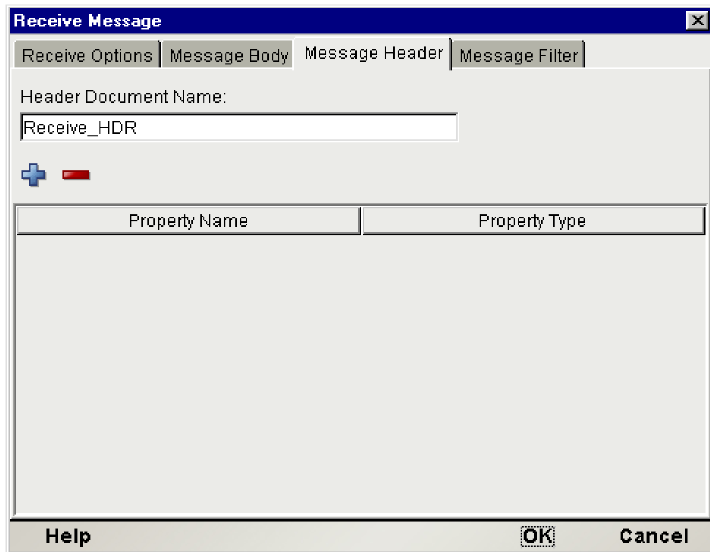
8 From the **Message Type** pulldown menu, select the message type that corresponds to the kind of message your component is designed to receive. This will preconfigure the Message Body to receive and store incoming data in a format that's acceptable to your application's requirements.

NOTE: It's important that your application take steps to ensure that only messages of the appropriate body type are received. Ordinarily, this will not be a problem since message-producing applications typically send their messages only to consuming applications that have been designed to receive them. Should your application happen to receive messages with a body type incompatible with the type you select in the Receive Message dialog, an exception will be thrown. If your application will be receiving messages from a queue that contains many different kinds of message body types, it is advisable that you design a Message Filter (selector statement) that can distinguish just the messages that are appropriate for your application.

9 Under **Body Document Name**, enter the name you wish to associate with the body of the incoming message, for purposes of DOM context (or accept the default, which begins with "Receive").

NOTE: This portion of the dialog will change in appearance according to the Message Type that you've chosen.) In the screen shot shown above, the Message Type is XML; hence, you are prompted to enter information for **Body Document Name** (i.e., the name you want to apply to the Message Body DOM), **Template Category** (the XML Template resource name), and **Template Name** (name of the XML stub document you want to apply to the Message Body, if appropriate). See "Using Other Actions in the JMS Component Editor" on page 79 for additional information.

10 Click the **Message Header** tab. The Message Header pane appears.



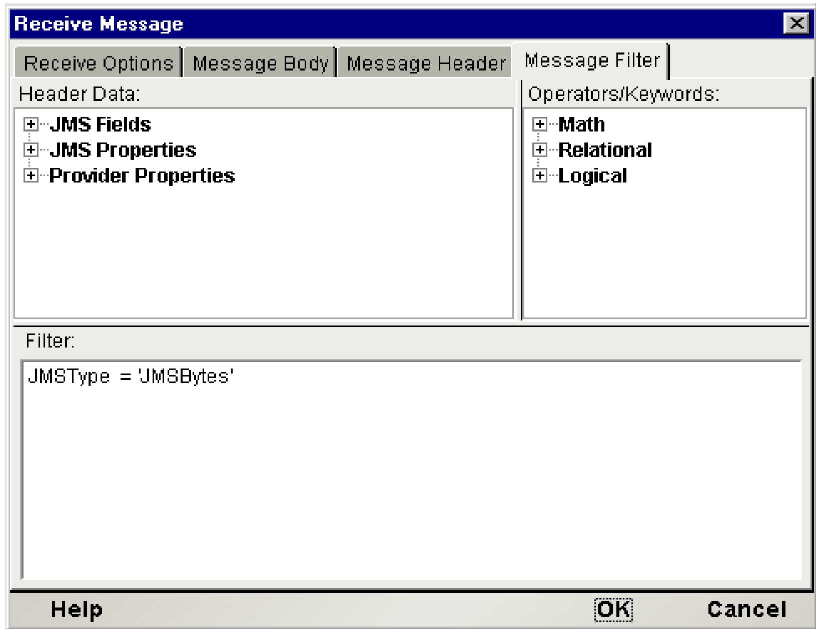
11 Enter a **Mapping Name** (or accept the default name, which begins with "Receive_HDR"). This name will be shown in the Native Environment Pane as the name of the header tree, for mapping purposes.

12 Click the plus (+) icon to add a Property. Type the name of the custom property under **Property Name** and click in the **Property Type** column to bring up a menu of available data types. Choose the data type appropriate to your property.

NOTE: The purpose of this procedure is to build a property list that corresponds to the anticipated property list of the incoming message. If the incoming message has property fields that aren't accounted for here, the extra fields will be ignored and any associated data will be lost.

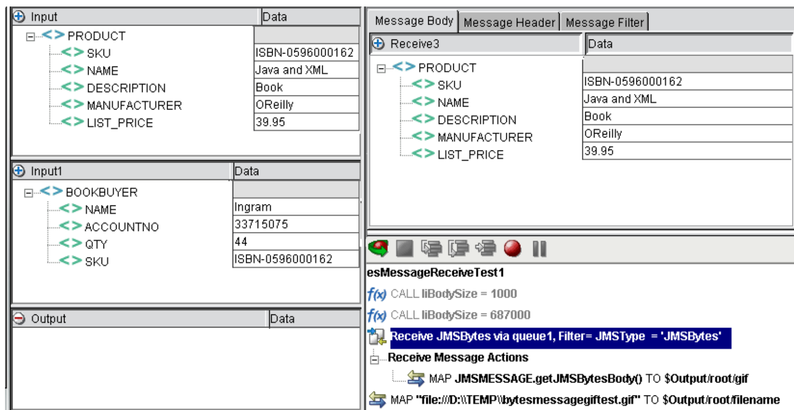
13 Click the plus (+) icon as many times as needed to add extra properties. Fill out the Name and Type information for each one.

14 Click the **Message Filter** tab. A new panel appears.



15 If you want to apply a JMS Message Filter to your Browse operation, enter it (and/or build it using the picklist items) in the text area in the lower half of the dialog. (See “Working with Message Filters (Selectors)” on page 98 for more information.)

16 Click **OK**. The JMS Component editor main window appears, and a new Receive action is displayed in the action list (see illustration).



When you create a new Receive action, the line “Receive Message . . .” appears in the action list, followed by a line “Receive Message Maps.” Under “Receive Message Maps,” you can insert any Map actions or other processing needed to make use of information contained in the received message.

Unlike a Browse Messages operation, a Receive Message action is limited to retrieving *one message at a time*. Therefore, the JMS Connect does *not* include a `WHILE` loop as part of every Receive Message action.

NOTE: To iterate through all available messages, you should enclose your Receive Message action (and any associated processing) within a Repeat While action that terminates when the JMSMessageID field is empty.

Note that as with the Browse Messages action (described above), the Native Environment Pane for the Receive Message action contains Message Body, Message Header, and Message Filter tabs. Working with these tabs is covered in the next chapter.

The Message Transaction Action

The Message Transaction action allows you to group two or more message-related actions into one logical unit of work that is executed (or not executed) atomically. When a transaction *commits*, all of its inputs (in terms of messages) are acknowledged and all outputs are sent. When a transacted message session *rolls back*, any produced messages are destroyed and any messages consumed during the session are recovered.

In order to utilize the Message Transaction calls within a JMS Component, it’s essential that the session occur in Transacted mode. The way one configures this is by setting the “Transacted” checkbox in the JMS Connection resource for the component. (The procedure for doing this is explained below. Also see “Creating a JMS Connection Resource” in Chapter 2.) When the JMS session associated by the JMS Component is in Transacted mode, you can safely use Commit and Rollback statements (via the Message Transaction action) in your Action Model.

What Happens When I Issue a Commit?

When you issue a Commit, every message action (starting either from the beginning of the Action Model or the last Commit or Rollback statement, as appropriate) is executed, which means every message produced up to that point is sent and every message consumed up to that point is acknowledged. (On a *receive* operation, message acknowledgement signals the queue manager that it is okay to destructively remove the message from the queue.)

What Happens When I Issue a Rollback?

When you issue a Rollback, every message action (starting either from the beginning of the Action Model or the last Commit or Rollback statement) is nullified, which means every message produced up to that point is destroyed (not sent) and every message consumed up to that point is recovered (i.e., allowed to remain on the queue, as if nothing had happened).

What Happens if I Leave the Session Unresolved?

An ambiguous state can arise if a transacted session uses no Commit or Rollback statements at all; or if an Action Model that contains several properly committed (and/or rolled back) actions ends with a message action that is neither committed nor rolled back. Interestingly, MOM vendors differ significantly in their handling of this situation. Some automatically commit unresolved actions; others roll back anything that is not committed.

The exteNd JMS Connect enforces an *automatic-rollback* protocol in situations where transactions are left uncommitted. (This is similar to the behavior of the JDBC Connect, where checking “Allow SQL Transactions” in the connection resource setup dialog causes automatic rollback to occur if the final SQL statement in a transacted JDBC Component uses neither a commit nor a rollback. See Chapter 2 of the *JDBC Connect* guide.) That is to say, any message actions that have occurred since the last Commit or Rollback statement will be rolled back when the session closes. Due to the way JMS “wrappers” MOM services, this behavior takes precedence over the MOM’s normal behavior. No matter whether your MOM defaults to automatic commit or automatic rollback, the JMS Connect guarantees one consistent behavior: automatic rollback of uncommitted messages.

What Actions Are Included in a Message Transaction?

Because the JMS Component allows the use of non-message-related Transaction actions (**New Action > Advanced > Transaction**) as well as Message Transaction actions, it’s important to understand the difference between the two.

The Message Transaction action calls `commit()` and `rollback()` methods that are scoped to the JMS *session*. Therefore, these methods cover *message-related processes only*. For example, while a Message Transaction command could roll back the sending of a message, it could never roll back a database operation conducted by an external call to a JDBC Component from within the Action Model of a JMS Component. If your Action Model contains a Receive Message action, followed by a Component action (involving a call to a JDBC Component), followed by a Send Message action, and you want to roll back everything (including the database operation) in the event of an error condition, you would need to demarcate the transaction using **New Action > Advanced > Transaction** (which is available in all Components) rather than
New Action > Message Transaction (which is specific to the JMS Component). The latter would roll back message operations only.

What Can I Use Message Transactions For?

A typical use of JMS transaction demarcation is to force "all or none" behavior in a set of related message actions. For example, suppose a set of messages (involving an order acknowledgement, a vendor notification, and a back-end query) must all be sent at once, or else not sent at all. Complex business logic may be involved in the determination of whether the sending of the messages should be authorized. Using a transacted JMS Component, the session could be set up such that messages are "sent" by default, but on any error condition or unsuccessful return from any point in the process, any Send Message (and/or Receive Message) operations are simply rolled back.

Another possible use of a transacted session involves nondestructive reading of messages. Normally, when an application reads a message off a queue, that message is permanently and irrecoverably removed from the queue. But if a JMS application reads a message in a transacted session, it can inspect the contents of the message before deciding whether or not to roll the session back. If the session *is* rolled back, the message will remain on the queue as if nothing happened. Why is this more useful than simply browsing? (Recall that browsing, unlike receiving, does not destructively consume messages.) Consider the case of an application that needs to inspect the *body* of a message before deciding whether to take action on that message. With browsing, the workflow would be:

1. Browse.
2. Inspect message contents.
3. If the message needs processing, *read* it from the queue (so as to ensure its removal from the queue); and process the message contents. Otherwise, do nothing.

Notice that the application must first do a browse, then a read operation. This represents two trips to the queue. With a transacted session, the workflow is:

1. Read the message off the queue.

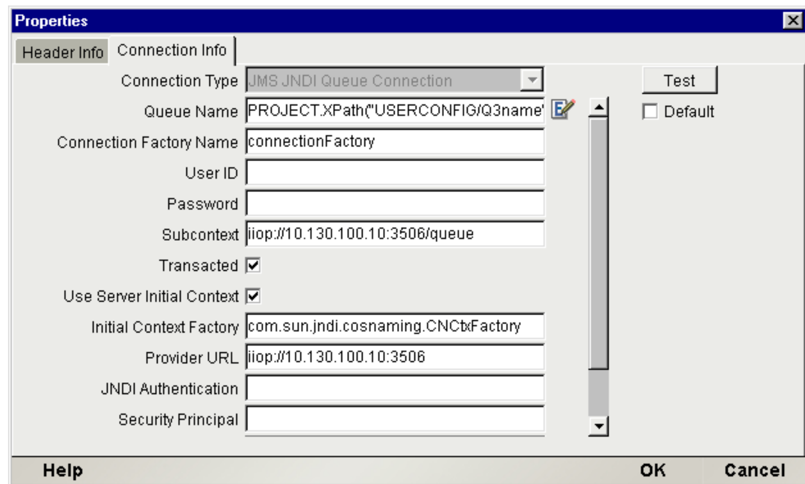
2. Inspect message contents.

3. If the message requires processing, take action. Otherwise, roll back the session.

In this case, there is only one trip to the queue (to read the message). If the message is not suitable, rolling back leaves the message on the queue as if nothing had happened. No matter whether the message is usable or not, there has been only one trip to the queue.

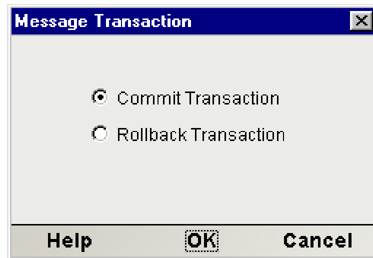
➢ To create a Message Transaction action

1 *Check to be sure the JMS Connection resource is transaction-enabled.* In the navigation pane of the exteNd Composer main screen, click Connection, then Resource; then doubleclick the appropriate JMS Connection resource in the detail pane. Select the **Connection Info** tab of the Properties dialog. The following screen appears:



Notice the **Transacted** checkbox in the lower part of the dialog. *This box must be checked if you are going to employ Message Transaction commands in your JMS Component.* If the box is left unchecked, Message Transaction calls in your component will cause exceptions to be thrown.

2 From the main menu, select **Action**, then **New Action**, then **Message Transaction**. A dialog box appears:



3 Select **Commit Transaction** or **Rollback Transaction** as appropriate.

NOTE: There is no need to issue a Begin statement. The Begin is implicit, based on your having checked the “Transacted” checkbox in the connection resource (see above). Checking this checkbox places the entire JMS session within a transacted context.

4 Click **OK**. The appropriate statement is inserted in the action list.

Using Other Actions in the JMS Component Editor

In addition to the Send Message, Browse Message, Receive Message, and Message Transaction actions, you have all the standard Basic and Advanced Composer actions at your disposal as well. The complete listing of Basic Composer Actions can be found in Chapter 7 of the *Composer User’s Guide*. Chapter 8 contains a listing of the more Advanced Actions available to you.

5 Working with Messages

The primary purpose of the JMS Connector is to allow you to leverage the power of Message Oriented Middleware (MOM) in your exteNd services, making possible many types of interactions between front- and back-end portions of a business application. To fully exploit this capability requires that you understand, in some detail, the ways in which messages can convey various forms of content via JMS Components. This chapter explains the process of putting messages to work.

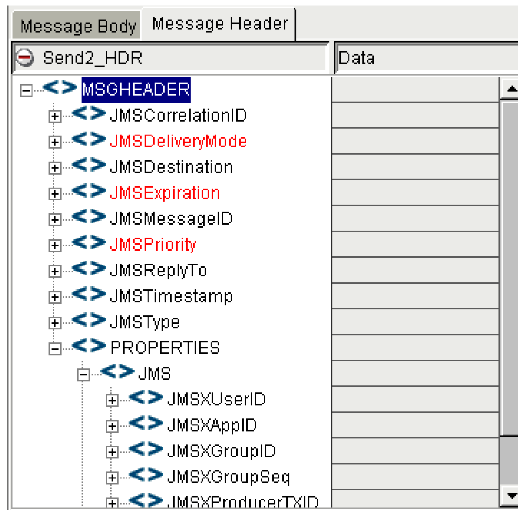
In this chapter, you will learn:

- How to map data into and out of JMS Message Headers
- How to map data into and out of Custom Properties
- How to map data from and to DOMs and message bodies
- How to work with the special XML and Copybook message types
- How to employ Message Filters (JMS Selectors) to receive or browse messages selectively, based on application-defined criteria
- How to use exteNd's JMS-related ECMAScript extensions to manipulate message data

Before reading this chapter, you should have already read "Getting Started with the JMS Component Editor" beginning on page 27, "Creating a JMS Component" on page 43, and "Creating JMS Actions" on page 51; and you should already be familiar with exteNd Composer, as well as the concept of Map actions.

Mapping Data into the Message Header

When you click on the Message Header tab in the Native Environment pane, a tree view of the JMS header is displayed:



This header (which contains property info as well as JMS-defined header fields) constitutes a unique DOM for mapping purposes. In the example shown above, the header DOM has been named Send_HDR. (This name was set in the message action setup dialog; see page 61.) The Send_HDR DOM can serve as a target for mapping data into the message header. It can also serve as the data source for mapping to elements of an output DOM.

Because JMS defines a number of preexisting fields, a message-header DOM tree is associated with *all* messages (and appears automatically in the Native Environment pane whenever you select the Message Header tab); hence, you can use the *drag-and-drop* technique to map data from any portion of any input DOM straight into the message header (subject to the limitations outlined below), or in the opposite direction. Simply click on an input node, in any visible DOM pane, and drag over to the desired spot in the message header, then release the mouse button. The appropriate Map action is added to the Action Model automatically.

Limitations on Header Mapping

Most JMS-defined header fields are *read-only* (or intended for use by the JMS provider) and therefore unavailable for use as drop targets. Only **JMSCorrelationID** and **JMSType** are intended to be mapped *into* by drag-and-drop. (If you attempt to drag an item into any other fields, you will see the "forbidden drag operation" symbol shown at left, along with the warning "Write-restricted drop target" in the status line of the component editor window.) JMSReplyTo is writable, but only through Send Options in the Send dialog (see discussion below). For more information about JMS header fieldsand their meanings, see Appendix C.

JMSCorrelationID

The **JMSCorrelationID** field is intended to serve as a tracking or control field for use by applications. A typical use of **JMSCorrelationID** is to serve as an identification string for request/response purposes. For instance, in the example shown further below (see "Actions Unique to the JMS Component Editor"beginning on page 52), the sending app may want to identify the business transaction represented by the current message with an unambiguous tracking number that may have significance for nonrepudiation. Using a Function action, one could assign a unique value, at runtime, to the **JMSCorrelationID** field consisting of the current ECMAScript date in milliseconds, plus the BOOKBUYER/NAME value (from the Input1 DOM), separated by a hyphen:

```
(Number(new Date)).toString() + '-' +
   Input1.XPath("BOOKBUYER/NAME")
```

At runtime, this will cause a value similar to `'976128839742-Ingram'` to appear in the outgoing message's **JMSCorrelationID** header field. A receiving application could take note of this value and place it in *its* outgoing messages. In this way, individual elements of a distributed application are able to identify this customer order as a unique order and operate on it in coherent fashion.

NOTE: JMS specifies that application-assigned **JMSCorrelationID** values must not begin with 'ID:', since that prefix is reserved for the use of JMS providers.

JMSType

The **JMSType** field can hold any string. A typical use of **JMSType** on outgoing messages is to hold sentinel values that receiving apps (pointing at the same queue) can inspect for filtering purposes. Depending on the nature of the application, you might choose to map this value from an input DOM, use a Code Table, base the value on an ECMAScript expression, hard-code a particular value, etc.

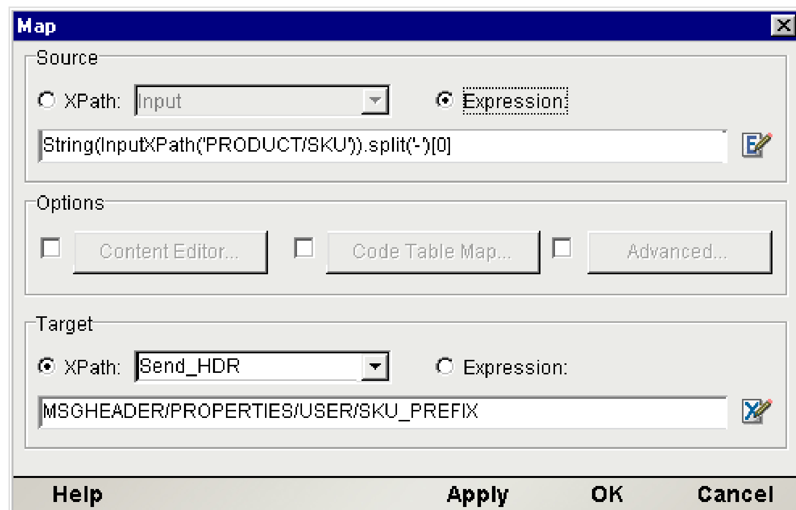
JMSReplyTo

The **JMSReplyTo** field is writable, but not through drag-and-drop or direct editring. To put a “return address” value in this field, use the Send Options tab in the Send Message dialog (as explained in the previous chapter). In the Send Options tab, there is a checkbox called Specify ReplyTo Queue/Topic. When this checkbox is checked, you can enter a queue name (in quotes) in the associated text field, or specify an ECMAScript expression that will evaluate to a queue name at runtime.

Mapping Data to Custom Properties

Any custom properties that you defined when creating the message action will appear automatically in the Message Header DOM tree. You can map data to these properties in any of the usual ways: drag-and-drop, ECMAScript, etc.

In the example further below, we have a custom property called SKU_PREFIX. Suppose we’d like to map a portion of the Input DOM’s PRODUCT/SKU data into this item: in particular we want just the first portion of the SKU, up to (but not including) the first hyphen. One way to accomplish this would be to highlight the SKU_PREFIX node of the Message Header tree, then do a right-mouse-click and select **Map . . .** in order to bring up the Map dialog window.



Under Source, we’ve clicked the Expression radio button and entered an ECMAScript expression of:

```
String(Input.XPath('PRODUCT/SKU')).split('-')[0]
```

Here, we use the String object's `split()` method to split the SKU string at every hyphen. The `split()` method returns an array, the zeroth member of which is the first substring, up to the first occurrence of the delimiter (in this case, the hyphen). Hence, this expression, when applied to the string "ISBN-0596000162", returns "ISBN."

A receiving application, perhaps tailored to handle just messages with a SKU_PREFIX value of "ISBN," could selectively pull messages of this type from a queue, ignoring all others. Filtering of this sort is done with Message Selectors.

Limitations on Property Mapping

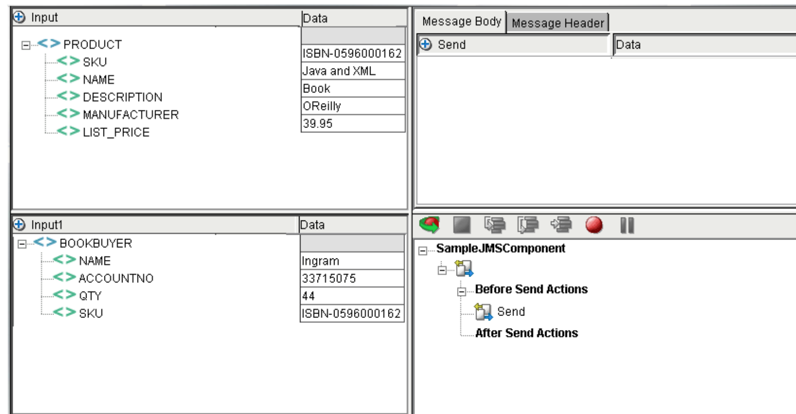
All user-defined Properties are read/write-enabled, which means that they can serve as drop targets for drag-and-drop mapping. The only restriction on this involves attempts to map *incompatible data types*. For example, if you attempt to drag a String value to a Property that has been defined as an Integer value, you will see the "invalid drag operation" icon as well as a status message (in the lower left corner of the component editor window) of "Invalid drag value for drop target: INTEGER." This is true for all header-field drag-and-drop operations: exteNd Composer will perform automatic type-checking during drag operations and prevent you from violating type constraints.

Working with XML Messages

A common use of messaging is to send an *XML document* (or documents) to a queue. For example, orders taken in real time over the web might be handed off to a back-end fulfillment system via a message queue. The back-end application could retrieve orders on a fixed schedule or by polling or listening for orders as they arrive.

In the following example, a publishing company is receiving book orders via the web from wholesales and distributors. The requirement is for a JMS component that can take information from two XML sources (one representing customer-submitted information and the other representing product information obtained via a database lookup) and transform the information into a new XML document, which will ultimately be sent to a message queue.

First, a Send Message action is created using the XML message type (as described in the previous chapter; see "Creating JMS Actions").



With the Message Body tab of the Native Environment pane selected, the body of the message is initially empty.

Notice that this particular component uses two input DOMs: *Input* (containing product information) and *Input1* (containing customer information). These DOMs could originate in many ways. The Input DOM might represent data pulled from a database via a JDBC Component. Information in the Input1 DOM could be customer information that arrived via the web (or via a message that was processed by another JMS Component).

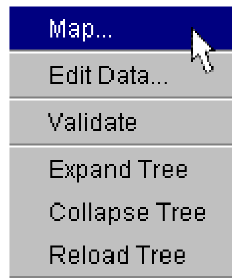
NOTE: In this example, an Output DOM is not shown, because the Message Body will contain this component's output. (Display of the default Output DOM has been suppressed using **View > Window Layout > XML Layout**.)

Mapping Data into the Message Body

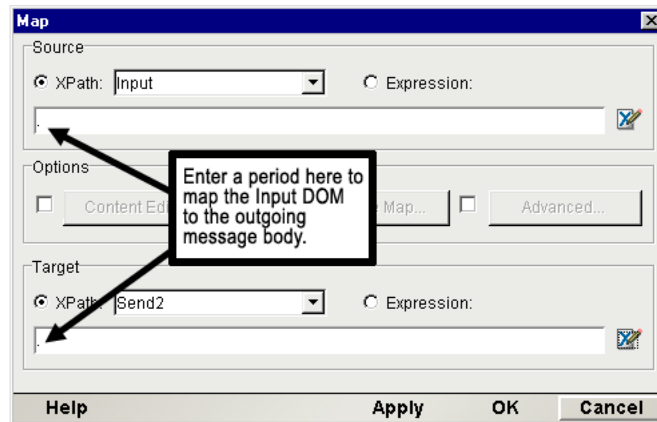
On certain occasions, you may want to map an *entire* XML document into a message body; in other cases, you might just want to map a *portion* of an XML document into a message body. We will discuss each case in turn.

Mapping an Entire XML Document into a Message Body

With the Message Body tab selected in the Native Environment Pane, right-mouse-click inside the empty Body area to bring up a context menu; then select **Map** from the menu.

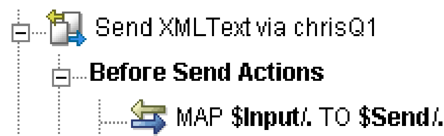


The standard XML Map Action dialog will appear:

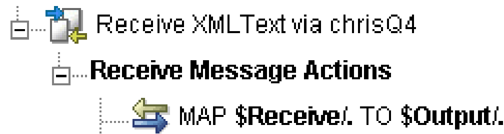


The default source DOM will be **Input**. (Use the pulldown menu to select another DOM, if you do not want to use Input.) Inside the Source text field of the Map dialog, enter a period (.), signifying that you want to map the entire source DOM to the target.

The default target will be **Send** (or whatever the name of the outgoing message is, as specified when the Send action was created). In the text field under Send, enter a period. Dismiss the dialog. You should now see a Map action in your action model that looks like:



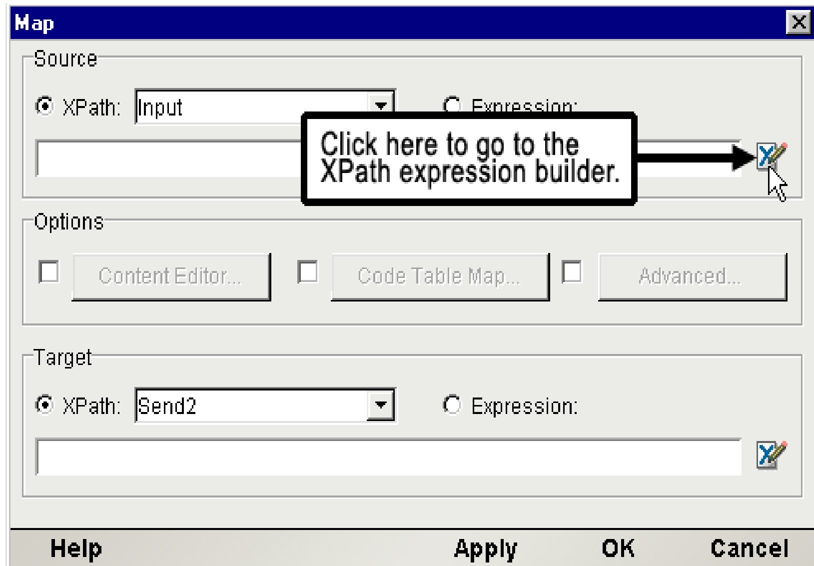
You can also map all of a message from Receive to Output as part of a Receive action using the same procedure. The result of this would be:



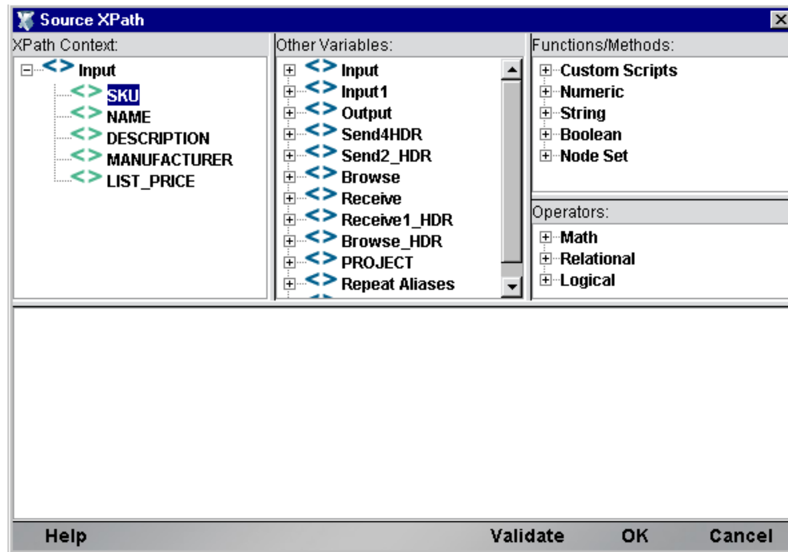
Mapping a Portion of an XML Document into a Message Body

To map a *portion* of an XML document into the Message Body, start by doing a right-mouse-click inside the empty area of the Native Environment pane (with the Message Body tab selected). This brings up a contextual menu.

Select the **Map . . .** command. This will bring up the Map dialog.

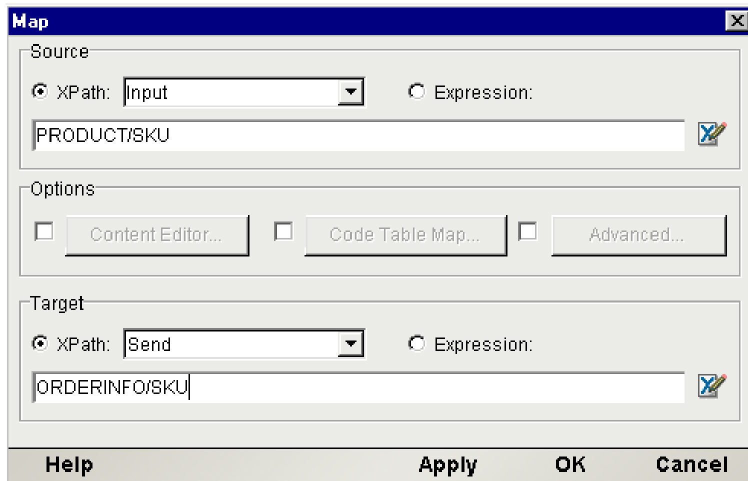


In the Map dialog, Input is shown as the default Source DOM and Send is shown as the default Target. (You can choose different Source and Target DOMs using the pulldown menus.) If you know the XPath fragment that you want to use as the source, type it in the box provided; otherwise, click on the blue Expression Editor icon at right. Clicking the Expression Editor icon brings up the Expression Editor dialog for the Source XPath.

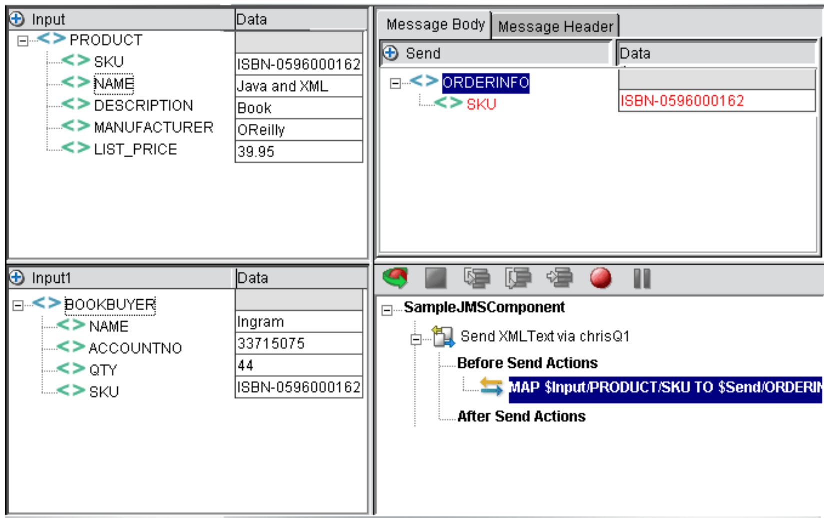


With the aid of the pick lists in the top portion of this dialog, you can build an XPath fragment and/or an ECMAScript expression simply by pointing and clicking. In this case, we've expanded the tree view of the Input DOM (in the upper left) to show the complete Input tree structure. Doubleclicking the SKU item in the tree causes PRODUCT/SKU (i.e., the XPath fragment for that portion of the tree) to appear automatically in the lower portion of the dialog. When we click OK, the XPath information appears in the appropriate place in the Map dialog.

To cause PRODUCT/SKU information to be mapped from Input to an XPath location of ORDERINFO/SKU in the message body, we type ORDERINFO/SKU in the Target portion of the Map dialog:



When we click OK, the map dialog disappears and we're able to see the result of our mapping in the JMS Component Editor main window:



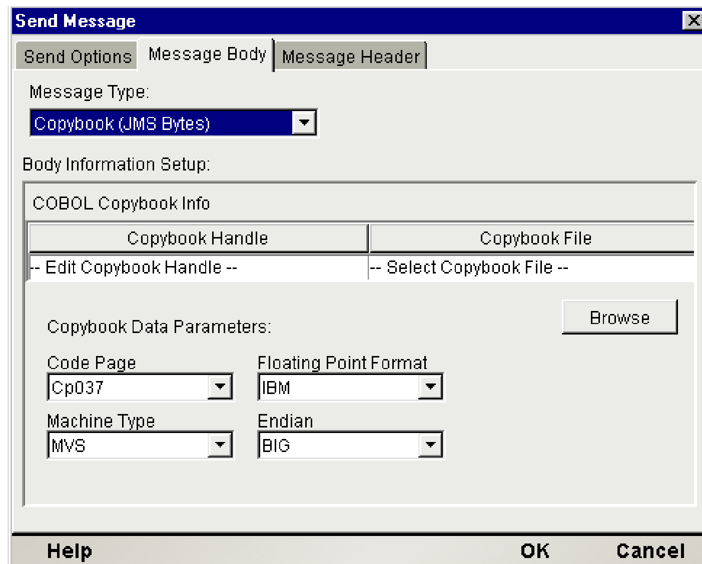
This procedure can be repeated as many times as necessary to populate the message body with data. Alternatively, you can use Function actions (in conjunction with ECMAScript DOM methods) to create XML nodes in the message body programmatically.

Working with Copybook Messages

One of the most powerful features of the JMS Connect is its ability to send, receive, and browse messages whose payloads consist of COBOL copybooks. Being able to use copybooks in messages allows the application Composer to leverage a wide range of legacy system interactions in exteNd services. This is especially true when the CICS RPC Connect is used in conjunction with the JMS Connect. For example, a copybook received as a message in a JMS Component can be transformed in accordance with business needs and used as the input to an RPC session, all within the same service.

Copybook Message Setup

When you create a Send Message, Receive Message, or Browse Messages action and you specify "Copybook (JMS Bytes)" in the pulldown menu for Message Type, the bottom portion of the setup dialog (under Body Information Setup) will contain fields that allow you to associate a copybook with the message action.



Under **Copybook Handle**, you can type an arbitrary text string that identifies the copybook for mapping purposes in the JMS Component editor.

The copybook's file name should show under **Copybook File**. If it does not, click the nearby **Browse** button and find the copybook you wish to use; its name will then appear in the Copybook File area.

Under **Copybook Data Parameters**, you will find four pulldown menus that allow you to choose the Code Page type, Floating Point format, Machine Type, and byte order (Endian), as appropriate to the target environment.

Code Page

Supported character encodings vary somewhat according to the version of the Java 2 runtime environment that is present. The Code Page pulldown menu will list all of the character encodings supported by the Java runtime being used.

Floating Point Format

The two floating-point formats supported by the JMS Connect are IEEE-754 and IBM formats.

Machine Type

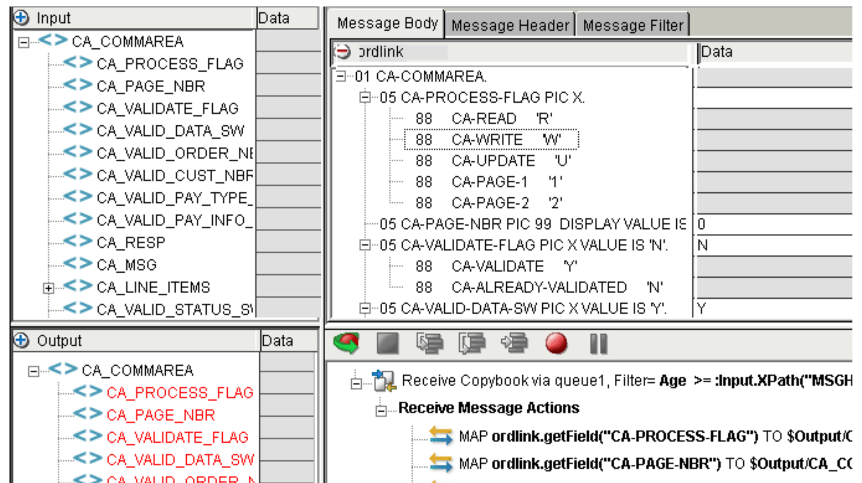
Machine Type refers to the target platform that will ultimately receive or process the copybook in question. The available choices are MVS, OS/2, NT, or AIX.

Endian

The two choices, BIG or LITTLE, refer to the native byte-order representation of the target platform. Intel architectures use a *little-endian* addressing scheme in which the least significant byte of a multi-byte entity is represented at the lowest memory address. Most other machine architectures are *big-endian*.

Copybooks and the Native Environment Pane

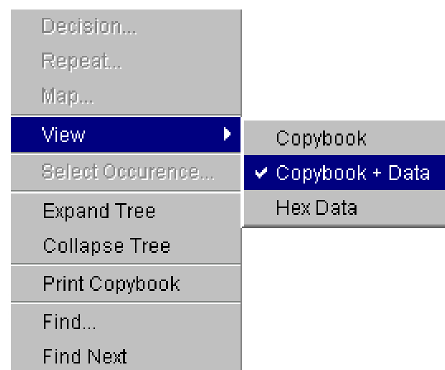
When you are in the JMS Component editor main window and you select (or highlight) a message action involving a copybook, the Native Environment pane displays information contained in the copybook that you selected in the message-action setup dialog.



In this case, a Receive Message action has been highlighted. The associated copybook is displayed in the Native Environment Pane in the upper right. There are three display modes for coypbooks in the Native Environment Pane:

- Copybook
- Copybook + Data (shown above)
- Hex Data

To select a different view, perform a right-mouse-click inside the Native Environment Pane and choose a View from the contextual menu:



The Hex Data view creates a standard hexadecimal data view of the copybook contents.

```
Message Body | Message Header | Message Filter
ordlink
0000:  2030304E 59202020 20303030 30303036  | 00NY     0000006|
0010:  38592020 20202020 20202020 20202020  |8Y              |
0020:  20202020 20202020 20202020 20202020  |                |
0030:  20202020 20202020 20202020 20202020  |                |
0040:  20202020 20202020 20202020 20202020  |                |
0050:  20202020 20202020 20202020 20204E00  |              N.|
0060:  00000000 00000000 00000000 00000000  |................|
0070:  00000000 00000000 00000000 00000000  |................|
0080:  00000000 00000000 00000000 00000000  |................|
0090:  00000000 00000000 00000000 00000000  |................|
00A0:  00000000 00000000 00000000 00002030  |.............. 0|
00B0:  30303030 30202020 20202020 20202020  |00000           |
```

This view is neither editable nor mappable; it is designed primarily as a troubleshooting and debugging aid.

Copybook-Specific Context Menu Items

The contextual menu that appears when you perform a right-mouse-click within a copybook's Native Environment Pane contains a number of copybook-specific commands, as explained below.

Native Pane Menu	Description
Decision	Enabled when a REDEFINE statement in the copybook is highlighted. When you click on Decision, a dialog box appears prompting you to enter a Decision expression that determines when to use the Redefined data descriptors.
Repeat	Enabled when an OCCURS statement in the copybook is highlighted. When you click on Repeat, a dialog box appears prompting you to enter information specifying the Target for the Repeat action.
Map	Active in the input or output pane. When you highlight a statement and click on Map, a dialog box appears prompting you to enter information.

Native Pane Menu	Description
View - Copybook	Active in the native environment pane. When you highlight a statement and click on View/Copybook, the pane will reflect the copybook format
View - Copybook and Data	Active in the native environment pane. When you highlight a statement and click on View/Copybook and Data, the pane will reflect the copybook and any data mapped into the copybook or placed there as output from executing the program.
View Hex Data	Active in the native environment pane. When you highlight a statement and click on View/Hex Data, the pane will reflect the copybook and any data mapped into the copybook or placed there as an output from executing the program in Hexidecimal format.
Select Occurrences	Enabled when an OCCURS statement in the Copybook is highlighted. Since each occurrence of the OCCURS clause is not displayed, when you click on Select Occurrences, a dialog box appears prompting you to select which occurrence you wish to see the data for. Enter a number. NOTE: The array and/or data structure is numbered from 0-5.
Expand Tree	Displays all Copybook nodes beneath the selected data descriptor.
Collapse Tree	Hides Copybook nodes beneath the selected data descriptor.
Copy	Enabled when in the View/Hex Data format. Allows you to highlight a block of text and copy it then paste.
Print Copybook	Allows you to print the copybook.
Find	Allows you to perform a search within the copybook in all data views.
Find Next	Allows you to search the next item in all data views.

Mapping Data Between Copybook and DOMs

When a copybook is visible in the Native Environment Pane, in any view except the Hex Data view, you can use the drag-and-drop technique to map XML DOM elements to copybook fields or vice versa. Mapping *into* the copybook will ordinarily be done from Input DOMs as part of Send Message actions. Mapping *out of* the copybook will ordinarily be done with an Output DOM as a target, for Browse or Receive actions.

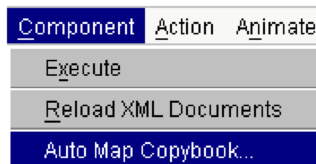
NOTE: The JMS Connect performs certain behind-the-scenes checks as part of every drag-and-drop operation. If a particular operation is forbidden, you will see the “forbidden drag operation” symbol, as well as an error message in the Component Editor’s status line, and no “drop” will occur when you release the mouse button.

Mappings between DOMs and copybook fields can also be created using the **Map . . .** command in the **Action** > **New Action** menu (from the main menubar) or the **Map . . .** command of the contextual menus. To access a contextual menu, right-mouse-click on any element in a DOM or copybook. The advantage of using the contextual-menu technique is that the Map dialog will contain appropriate XPath information for the DOM/copybook element you clicked on, already filled out.

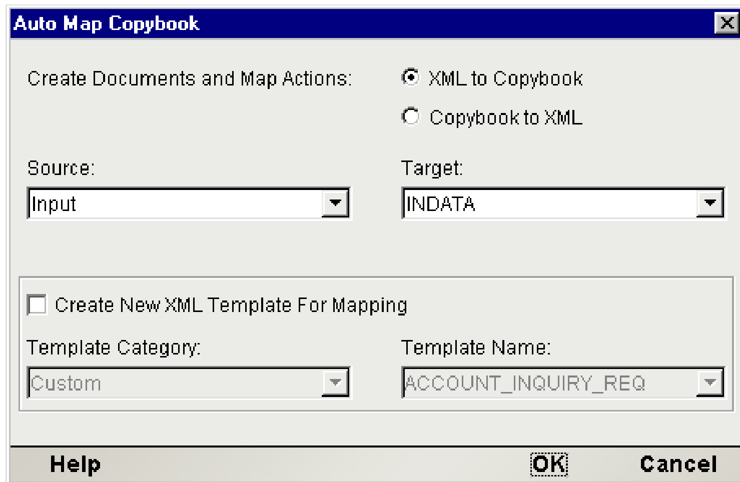
Mappings can also be created programmatically, using ECMAScript, inside Function actions.

Auto Map Copybook

Using the mapping techniques described above, you may have to perform numerous keyboard and/or mouse operations to create required mappings to or from DOM elements and copybook fields. When the copybook is more than a few lines long, this quickly becomes a tedious affair. To make it easier to create large numbers of mappings, the JMS Connect provides an **Auto Map Copybook** command under the **Component** menu (on the main menubar):



Selecting this command brings up the following dialog:



This dialog allows you to create batch mapping actions between DOMs and copybooks (in either direction) quickly and easily. Optionally, it allows you to create a new XML template document, based on a copybook, which can be saved permanently in the event you want to reuse it.

NOTE: The **Auto Map Copybook** dialog will appear only if a copybook is showing in the Native Environment Pane. (This, in turn, requires that a copybook-related message action be highlighted in the Component Editor's action list.)

➢ To create a batch mapping of XML elements to copybook fields (or vice versa)

1. Select the appropriate radio button in the upper right portion of the Auto Map Copybook dialog (**XML to Copybook**, or **Copybook to XML**).
2. Select the desired **Source** and **Target** documents from the two pulldown menus.
3. Click **OK**. The Component Editor Window appears and new Map actions are shown in your Action Model.

NOTE: You may want to inspect the Map action list that is thus created and remove any actions that are not relevant to your project.

➢ **To create a new XML Template document based on a copybook**

1. Select the appropriate radio button in the upper right portion of the Auto Map Copybook dialog (**XML to Copybook**, or **Copybook to XML**). Selecting **XML to Copybook** indicates that you wish to create an Input document and Map Actions to move data into the Copybook. Selecting **Copybook to XML** indicates that you wish to create an Output document and Map Actions to move host program output data from the Copybook to the Ouput document.

2. Select the desired **Source** and **Target** documents from the two pulldown menus.

3. In the lower portion of the dialog, check the **Create New XML Template for Mapping** checkbox. This indicates that *in addition to* the batch-mapping operation (described above), you wish to create a new XML Template document. Checking the checkbox will activate the lowermost items in the dialog: Template Category and Template Name.

4. Select a **Template Category** if it is different than the default category, or enter a New Template Category Name over an existing entry.

5. Select a **Template Name** from the list of XML templates, or enter a New Template Name over an existing entry.

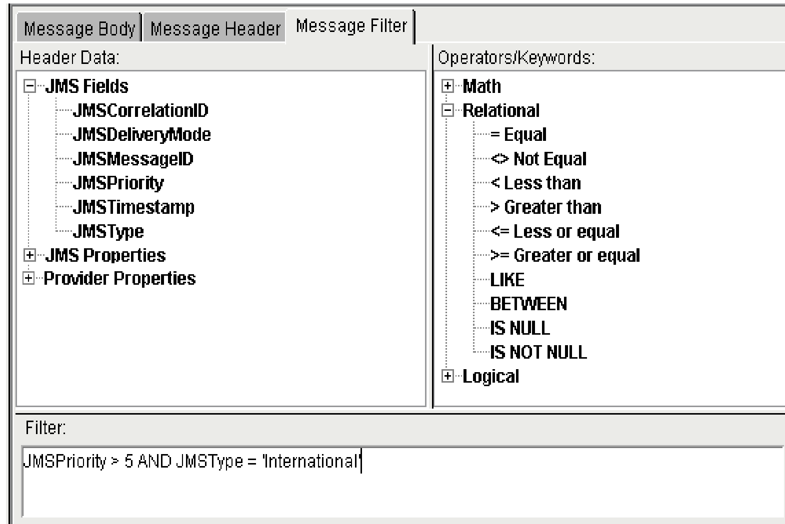
6. Click **OK.** The Component Editor Window appears with the Input Template created. (Also, the new template document appears on disk in the **xmlcategories** folder under your project's main folder, and the new template category appears under XML Templates in the navigation pane of Composer's main window.)

Working with Message Filters (Selectors)

The JMS standard allows *message filtering* (that is, selective retrieval of messages) to occur based on user-determined criteria. Filtering occurs via a JMS *message selector*, which is a String containing a statement (with SQL-like syntax) that will evaluate to true or false. The selector statement will normally refer to one or more JMS message header fields and/or custom properties. (Message selectors *cannot* reference message body values.) Based on data exposed in the message header, the message will either be selected for retrieval, or not selected.

Selector setup occurs in the JMS Connect by way of the Native Environment Pane. The Browse Messages and Receive Message actions (for all message types) cause a Message Filter tab to be included in the Native Environment Pane, along with tabs for Message Body and Message Header. The Message Filter tab exposes a text-edit field where you can place a selector expression. The expression can be typed manually or constructed, in whole or in part, by doubleclicking pick-list entries from the upper part of the tab.

As an example, consider the following Message Filter:



In this example, the selector expression requires that JMSPriority be greater than 5 *and* that JMSType exactly equal the string ‘International’. In the upper part of the tab, under Header Data, the JMS Fields node of the picktree is expanded to reveal the available header fields. (Not all of the ten JMS-defined header fields can be used for filtering; see “Limitations on Filtering” below.) JMSType is highlighted; doubleclicking it places “JMSType” in the Filter field, at the cursor’s location. Operators and keywords can likewise be “picked” using the pick lists in the upper right, to form an expression (or part of an expression) without typing.

Using the filter expression example shown, any Browse Messages action will retrieve from the queue *only* those messages with a JMSType of “International” and a priority greater than 5. All other messages will be ignored.

In a Receive Message action, the filter would result in the *first available message* that meets the criteria being removed from the queue.

Other selector expression examples can be found in Appendix B.

NOTE: You should escape any colon characters (":") in your selector expressions with a backslash, since exteNd treats colons in SQL statements and JMS selectors as a marker, indicating the presence (after the colon) of an evaluatable script expression.

Limitations on Filtering

JMS imposes a restriction on which header fields can be used for filtering.Header field references inside selector statements are restricted to the following fields:

- JMSCorrelationID
- JMSDeliveryMode (integer)
- JMSMessageID
- JMSPriority (integer)
- JMSTimestamp (long)
- JMSType

Data types can be important in selector statements, since if the comparison of non-like types is attempted, the statement will always return *false*. In the above list, all fields are Strings, except for JMSPriority, which is of type `integer`, and JMSTimestamp, which is of type `long`.

NOTE: Normally, JMSDeliveryMode is an integer, but in a selector context it will have the String value 'PERSISTENT' or 'NON_PERSISTENT'.

Custom, application-defined property fields can be used as the basis of selector statements. But if a selector references a property that does not exist, the value of the operation will be unknown. That's because SQL semantics treat NULL values as unknowns; and any operation involving an unknown value produces an unknown value. For a selector that references a NULL value to be useful, it must return true or false. The only way this can happen is if you use the IS NULL or IS NOT NULL operator to convert an unknown value into a boolean result.

Filtering by Body Type

You should be aware that JMS makes no provision for filtering by *message body type* (BytesMessage, StreamMessage, etc.) per se. If a client application requires access to this information, the relevant body-type info should be exposed in a custom property or header field by the sending application. A JMS receiver has no *a priori* way of knowing what the body type of a message is.

Notwithstanding this, in order for the JMS Connect to know what to do with the contents of an arriving message, it is necessary for *some* value to be selected in the Message Type field of the Receive Message (or Browse Messages) setup dialog's Message Body tab (see illustration, page 72). The value given in the Receive Message setup dialog will *not* be used for filtering, however, since there is no way for a JMS application to filter messages by body type (as just explained).

NOTE: It's important that your application take steps to ensure that only messages of the appropriate body type are received. Ordinarily, this will not be a problem since message-producing applications typically send their messages only to consuming applications that have been specifically designed to receive them. Nevertheless, should your application happen to receive messages with a body type incompatible with the type you select in the Receive Message dialog, exceptions will be thrown.

If your application will be receiving/browsing messages from a queue that contains many different kinds of message body types, it is advisable that you expose body-type info in a message property or header field and design a Message Filter (selector statement) that the receiving app can use to distinguish just the messages that are appropriate from those that are not.

Request-Response Messaging

Roundtrip request-response messaging is a common scenario in applications that use messaging. Some typical use cases are:

- Your application sends a message to a queue or topic with the expectation of receiving an immediate reply (or replies). For example, a manufacturer may want to purchase a particular kind of part; to do this, it may broadcast a request for a bid on the item by publishing a message to a topic. Listeners on that topic may be set up to respond immediately with an acknowledgement (or an actual bid).
- Your application sends a message to a queue or topic without any expectation of getting a response; but for purposes of error notification, your app specifies a ReplyTo destination in the outgoing message. (The ReplyTo queue might be in an entirely different domain, and might be set up solely to accumulate error reports.) Thus, your app sends messages out to one queue, but asks that error reports be sent to *another* queue.
- Your application is a JMS Service that listens on a queue or topic and is designed to process the incoming message, then *reply* to it.

In each case, the sender needs to supply a queue or topic name in the *JMSReplyTo* header field of the sent message. This can be done using the Send Options tab of the Send Message dialog; see the section called "Return Address" under "The Send Message Action" in the previous chapter.

In the case where your application is *responding* to a query, you can set up your Send Message action so that it automatically uses the queue or topic that was present in the *JMSReplyTo* field of the last received message. In the Send Options tab of the Send Message dialog, there is a radio button called Use Prior Message ReplyTo Field, provided for this purpose. (See "Destination Queue/Topic" under "The Send Message Action" in the previous chapter.)

Temporary Queues

When your application is soliciting an immediate response, and you intend to block until a reply is received (or until a specified timeout value is reached), it is often convenient to create a *temporary queue* dedicated to receiving the reply. The advantages of using a temporary queue are:

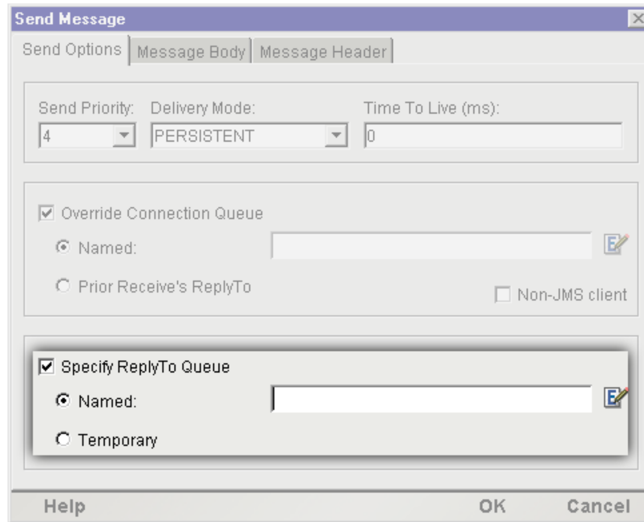
- Isolation from other processes (no client other than the one receiving your message knows about the temporary queue). This can simplify application design by minimizing or even eliminating the need for message filters.
- From a resource and administration standpoint, temporary queues are cheap, since they are created on the fly and destroyed immediately after they have served their purpose.

To specify that a temporary queue be used for replies to a message, check the Use Temporary radio button in the Send Options tab of the Send Message dialog. (See "Return Address" under "The Send Message Action" in the previous chapter.) A temporary queue will be created for you automatically and its name will be placed in the *JMSReplyTo* field of your outgoing message. The queue will then exist for the lifetime of your component.

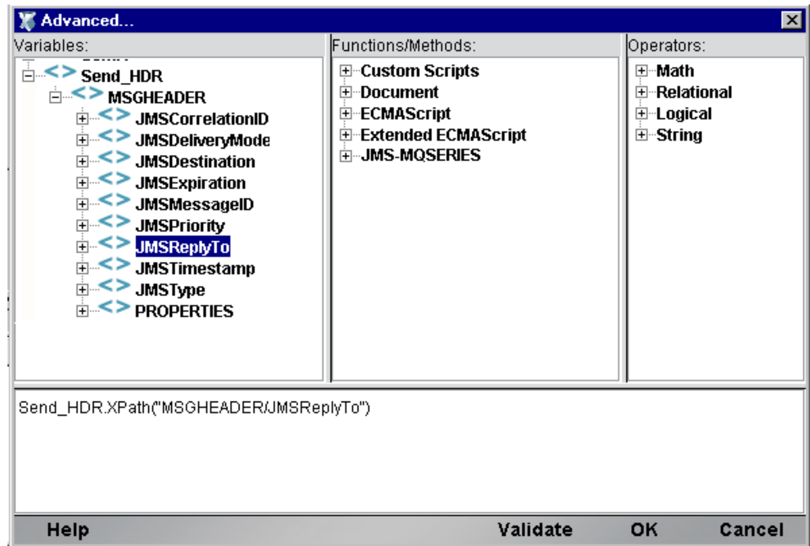
Multiple Temporary Queues

In cases where multiple outgoing messages will be sent in the lifetime of a single component, multiple temporary queues will be created. By default, the JMS Connectwill create a *unique* temporary queue every time an outgoing message is created (if the Use Temporary radio button has been set). If you want *multiple* outgoing messages to specify the *same* temporary queue in their ReplyTo fields, you can do it in the following fashion:

1. Create your first Send Message action in the usual way, checking the **Use Temporary** radio button in the Send Options tab. (See "Return Address" under "The Send Message Action" in the previous chapter.)
2. In your next Send Message action, *do not* check the **Temporary** radio button (since this will cause a new, unique temporary queue to be used). Instead, check the **Named** radio button.



3 Click the **Expression** icon to the right of the text field. The Expression Builder dialog appears.



4 In the Variable picktree (upper left), open the send header node (default name **Send_HDR**) for the *first sent message*. Expand the **MSGHEADER** node to reveal all of the JMS header field names.

5. Doubleclick the **JMSReplyTo** entry in the picklist. An ECMAScript expression appears in the edit field below.
6. Click **OK** to go back to the Send Message dialog.
7. Set any other message parameters that you might need to specify for this message, then click **OK** to close the Send Message dialog.
8. Repeat steps 2 through 7 for each subsequent Send Message action in your component that will use the original temporary queue.

ECMAScript and the JMS Connect

The JMS Connect exposes a number of JMS-related ECMAScript extensions that you can use in your own Function and Map actions to extend the functionality of your JMS Components and services. For the most part, the extensions comprise "get" and "set" methods for manipulating the body content of messages.

Access to JMS-related ECMAScript extensions is available via pick lists in the Expression Builder dialog. (The Expression Builder, in turn, is available from the Map action and Function action dialogs. See example below.) You should note that the methods exposed via the Expression Builder's pick lists are exposed in *context-sensitive* fashion. For example, if you are working with a Copybook Message, the methods exposed via the picktree will correspond to copybook-related operations, whereas if your Message action involves a Bytes Message, the exposed ECMAScript methods will relate to working with the JMS Bytes Message body type.

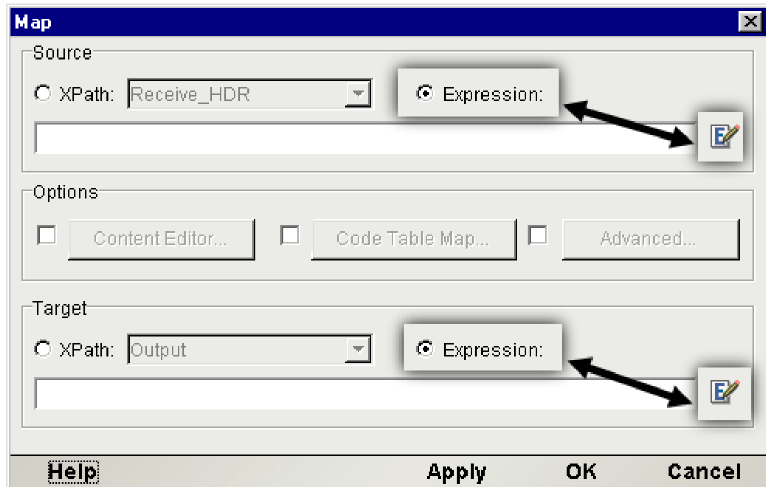
In the example below, we show how to attach content to a JMS Bytes Message using ECMAScript. Unlike most other message types, the Bytes Message type has no user interface for mapping (no Body tab in the Native Environment Pane). Hence, ECMAScript is the *only* way to attach content to the body of a Bytes Message.

NOTE: The JMS Object Message also has no user interface for mapping. You must use ECMAScript to attach content to an Object Message. (Typically, you will rely on ECMAScript's Packages mechanism to call Java code to retrieve Serializable objects associated with Object Messages. See Chapter 10 of the *exteNd Composer User's Guide*.)

➤ **To work with JMS-related ECMAScript extensions in the Expression Editor**

1. Create a **Send Message** (or other JMS) action. Select a body type corresponding to the type of content your message will have. For this example, a Bytes Message will be used, but the principles demonstrated here will apply to all body types.

2 With the Message action highlighted, create a new Map action. In the example below, we have selected **Action** from the main menu, then **New Action**, then **Map**, to create a new Map action.

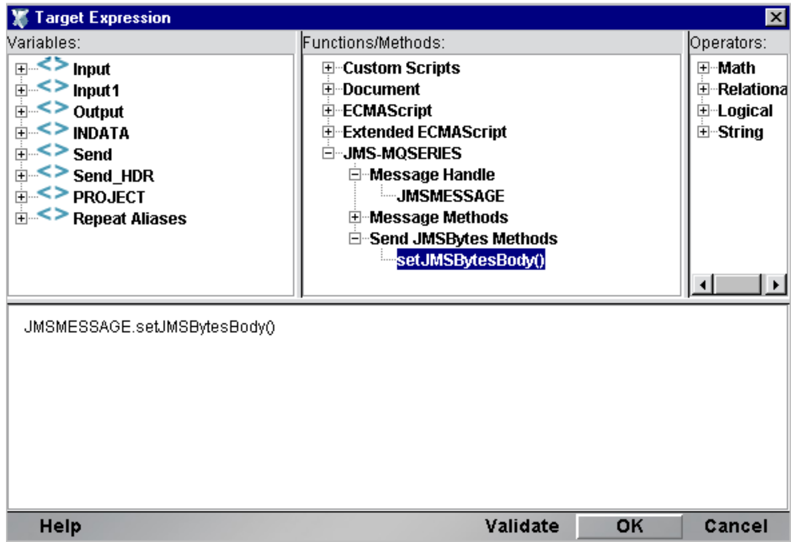


3 Click the **Expression** radio button for Source as well as Target.

4 Type the Source expression representing the source of your message-body data, or (alternatively) click the **Expression Editor icon** next to the text-edit area for Source. The Expression Editor window appear will appear.

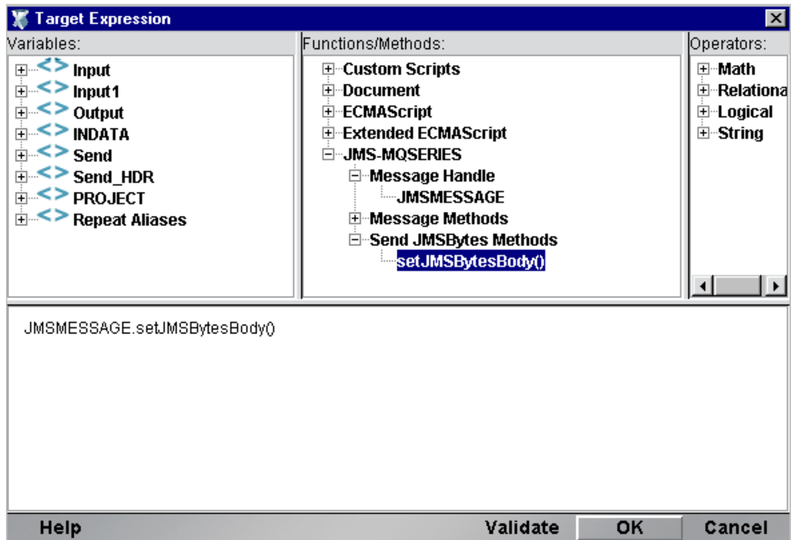
NOTE: In this example, we will map data from the PRODUCT/NAME XPath of our Input DOM, but you can also map from other sources. For example, you could use the `File` constructor and `readAll()` method to obtain data from a file on disk. (The “Extended ECMAScript” pick list contains File I/O and other methods.)

5 Type the Target expression, or build it using the Expression editor. Click the **Expression Editor icon** next to the text-edit area for Target. The Expression Editor window appears.



6 In the upper middle part of the Expression Editor, click the **JMS-MQSERIES** item to expand the picktree to reveal nodes labeled Message Handle, Message Methods, and Send JMSBytes Methods.

NOTE: If you do not see any JMS-related nodes in the picktree, it is because your Map or Function action was not associated with a Message action. Be sure to highlight a Message action (or related section of the action list, such as “Before Send Maps”) prior to creating the Map or Function action.



7. Expand the various nodes of the tree under JMS-MQSERIES. You will see terminal nodes with names like **JMSMESSAGE**, **getJMSMessage()**, and **setJMSBytesBody()**. Doubleclick these names (and/or other leaf nodes in other picklist windows) as necessary to build the desired ECMAScript expression. The relevant labels appear in the text-edit portion of the window as you doubleclick.
8. Dismiss all dialogs by clicking **OK**. Your Map action appears in the component's Action list.

To attach content to a message using ECMAScript, use the target expression:

```
JMSMESSAGE.setJMSBytesBody( );
```

No arguments are necessary. This "setter" method will take data from the Source part of the Map action, convert it to the appropriate format (a byte array, in the case of a Bytes Message), and attach it to the body of the message. Corresponding "getter" methods operate in similar fashion, although some of these methods require arguments. The calling conventions are set forth below.

ECMAScript Method Summary

Available JMS-related ECMAScript extensions and usage are as follows. Most methods will be called on the JMSMESSAGE handle; the exceptions are Copybook methods (called on the Copybook handle) and CopybookField methods (called on CopybookField objects).

```
Message getJMSMessage()
```

When called on the JMSMESSAGE handle, this method returns a JMS Message object. To work with a specific type of JMS Message body, caste the returned message to the appropriate type. For example:

```
TextMessage lMsg = (TextMessage)getJMSMessage()
```

```
String getJMSMsgBody()
```

Call this method on JMSMESSAGE to obtain the body of a message as a String.

```
String getJMSMsgType()
```

Call this method on JMSMESSAGE to obtain the body type of a message as a String. The return value will be one of *JMSText, JMSBytes, JMSMap, JMSObject*, or *JMSStream*. Notice that no special XML or Copybook type is ever returned since these are not JMS-defined types.

`CopybookField getField(String cobolDataDesc)`

Example:

Suppose you have the following copybooks:

COMMAREA

05 INDATA

10PARTID

05 OUTDATA

10PARTID

Perhaps you're interested in referencing the first PARTID (under INDATA), but you don't want the PARTID under OUTDATA. To resolve the duplicate name issue, reference the parent `cobolDataDesc` as follows (assuming a Copybook Handle of MYCOPYBOOK):

```
MYCOPYBOOK.getField("PARTID IN INDATA")
```

The returned CopybookField object has two methods: `toString()` and `setValue()`.

`void setValue(Object aValue)`

This method sets the value for a CopybookField object.

`String toString()`

Returns the value set for the CopybookField object.

`String getJMSBytesBody(int aiBufSize)`

Gets the value for a JMS BytesMessage object's body as a String representation. The aibufSize parameter is the size of the body in bytes. Returns a String.

`setJMSBytesBody()`

Sets the body for a JMS BytesMessage object.

`String getJMSBytesBodyAsBytes(int aiSize)`

Returns a Java byte[ ] Object of size specified by aiSize.

`setJMSBytesBodyAsBytes()`

Returns a Java byte[ ] Object.

`String getJMSMapField(String asName, String asType)`

Gets the value for a JMS MapMessage object body field. Returns a String.

`setJMSMapField(String asName, String asType)`

Sets the name and type of a JMS MapMessage object body field.

`Serializable getJMSObjectBody()`

Gets the value for a JMS ObjectMessage object body (after a Receive Message action). Returns a Serializable object.

`void setJMSMsgProperty(String asName, String asType, String asValue)`

Sets a JMS header of a given name and type to a given value.

`setJMSObjectBody(Serializable aObject)`

Sets the body of a JMS ObjectMessage object.

`String getJMSStreamField(String asName, String asType)`

Gets the value for a JMS StreamMessage object body field. Returns a String.

`setJMSStreamField(String asName, String asType)`

This method sets the value of a JMS StreamMessage object body field.

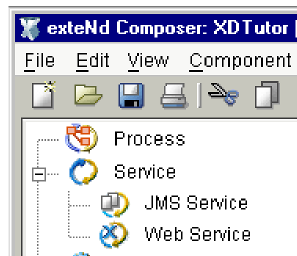
`String getJMSTextBody()`

Gets the value for a JMS TextMessage object body. Returns a String.

6 The JMS Service

JMS defines a mechanism, called the *MessageListener* object, whereby message consumers can be asynchronously notified whenever a message has been published to a queue or topic. This gives a receiving application the ability to treat incoming messages as events: Instead of the application having to go out and pull messages off a topic, messages are—in effect—*pushed* toward the application by the queue or topic manager.

To take advantage of this capability, the JMS Connect introduces a new type of exteNd Composer xObject called the *JMS Service* (which appears as a category under Service in the main Composer window).



About the JMS Service

Like other exteNd services, the JMS Service can call external Components, perform XML Interchange actions, carry out Log actions, execute Function actions, etc. (See “Creating a Service” in the *Composer User’s Guide*.) But the JMS Service differs from other services in a number of important ways:

- The JMS Service is triggered by an *incoming message* (from a queue or topic).
- To assure proper processing of the incoming message, the JMS Service must contain one (and only one) Receive Message action.

NOTE: With exteNd version 2.7 and subsequent, you can put Send Message actions inside a JMS Service, so that a listener can reply to incoming messages directly (instead of having to call another component).

It's important to note that JMS Components do *not,* in general, need to be packaged inside JMS Services. The distinguishing characteristic of a JMS Service is not its *content* but its *triggering mechanism*. The JMS Service is designed to be triggered by the arrival of a message at a queue or topic. Services that are designed to be triggered from HTTP servlets should be created as Web Services, even if they use JMS Components. See "Creating a Service" in the most recent edition of the *Composer User's Guide*.

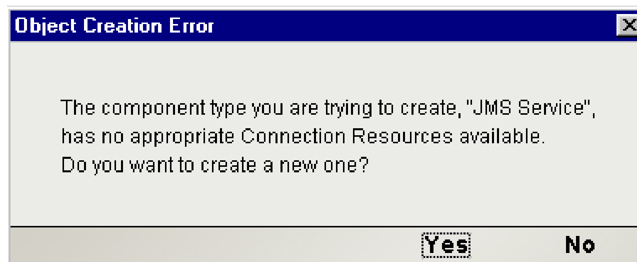
Multiple Listeners

A useful capability offered by the JMS Connector is the ability to deploy a JMS Service with multiple listeners. This makes it possible to have multiple instances of the same JMS Service running at the same time.

Creating a multiple-listener service is merely a matter of designing a JMS Service and deploying it using the same deployment wizard that you normally use. The JMS Service panel of the deployment wizard now has a Count field (see graphic in the section titled "Deployment of the JMS Service" further below) in which you can specify the number of listeners you want to deploy. Once deployed, the listeners can be administered from an HTML-based console window (see discussion further below under "How Do I Manage Deployed JMS Services?").

Creating a JMS Service

The JMS Service is created much like any other service. Before creating a JMS Service, however, you should already have created a JMS Connection resource for the queue or topic from which your service will be receiving messages. (See "Creating a JMS Connection Resource" for more information.) If you omit this step, you will see an error message similar to the one below:

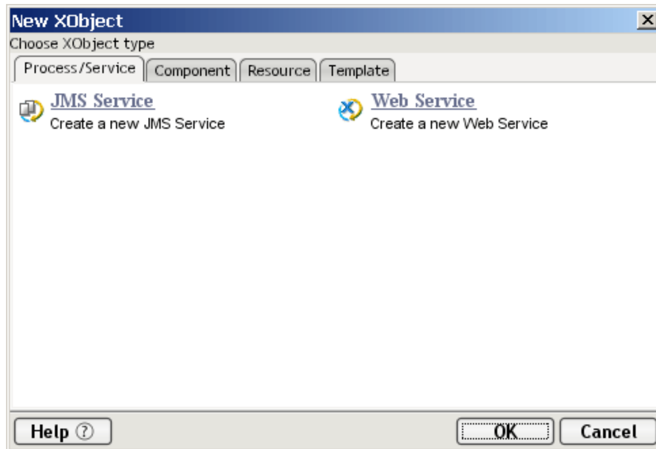


Click **Yes** in this dialog if you wish to create a new JMS Connection resource on the fly.

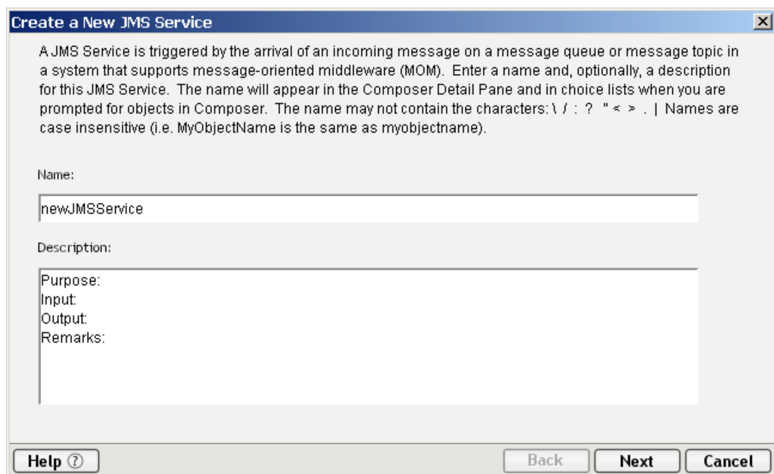
The following discussion assumes that you have already created a connection resource for use with your JMS Service.

➤ To create a new JMS Service

1. From Composer's main menu, select **File**, then **New>xObject**, then open the **Process/Service** tab and select **JMS Service**.



2. In the first panel of the "Create a New JMS Service Component" wizard, type the **Name** you want the service to have and (optionally) **Description** information.



3 Click **Next** to display the Templates panel.

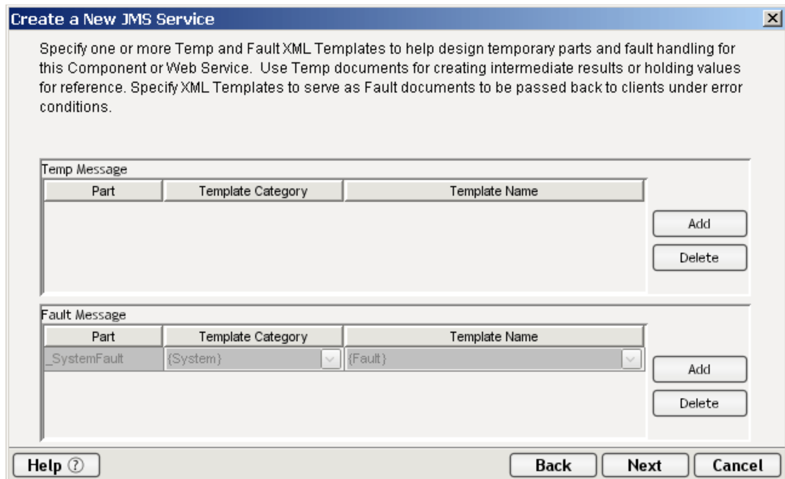
4 Specify the Input and Output templates as follows.

- Type in a name for the template under **Part** if you wish the name to appear in the DOM as something other than “Input”.
- Select a **Template Category** if it is different than the default category.
- Select a **Template Name** from the list of XML templates in the selected **Template Category**.
- To add additional input XML templates, click **Add** and choose a **Template Category** and **Template Name** for each.
- To remove an input XML template, select an entry and click **Delete**.

5 Select an XML template for use as an Output Part using the same steps outlined above.

NOTE: You can specify an input or output XML template that contains no structure by selecting {System}{ANY} as the Input or Output template. For more information, see “Creating an Output Part without Using a Template” in the User’s Guide.

6 Click **Next**. The XML Temp/Fault Template Info panel of the New HP3000 Terminal Component Wizard appears.

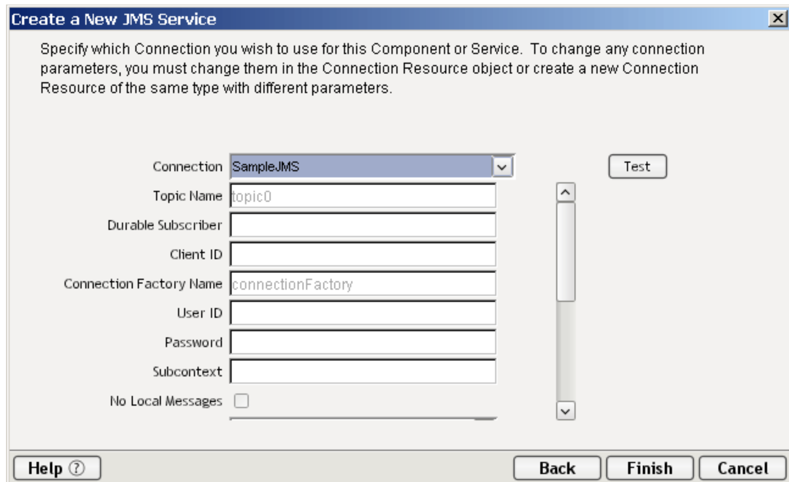


7 If desired, specify a template to be used as a scratchpad under the "Temp Message" pane of the dialog window. This can be useful if you need a place to hold values that will only be used temporarily during the execution of your component or are for reference only. Specify the templates as indicated in Step 6 above.

8 Under the "Fault Message" pane, select an XML template to be used to pass back to clients when an error condition occurs.

9 As above, to add additional temp or fault XML templates, click **Add** and choose a Template Category and Template Name for each. Repeat as many times as desired. To *remove* an XML template, select an entry and click **Delete**.

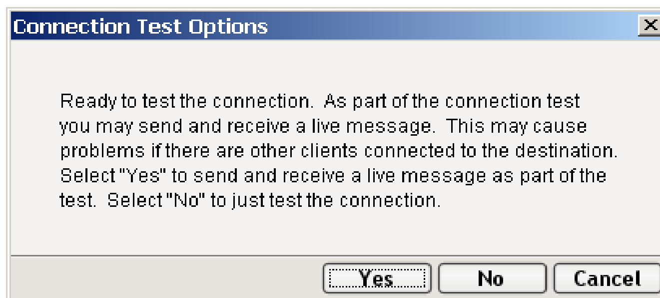
10 Click **Next** to bring up the final panel of the wizard.



11 Select a **Connection** from available queues and/or topics shown in the pulldown menu.

NOTE: Fields underneath the Connection menu will be greyed out (disabled). If you need to change the information displayed in any of these fields, you can do so by opening the appropriate Connection Resource from Composer's main window (after first dismissing this dialog).

12 Click **Test** to see if your connection is successful. The Test Options dialog appears.



13 The Test Options dialog asks if you want to send a live message as part of the test of the connection's integrity. Clicking the **Yes** button causes Composer to send a live message (of type *TextMessage*, with a unique CorrelationID) to the queue or topic for which you're establishing a connection.

NOTE: Use care not to send this test message in a production environment (i.e., using a live queue, with potentially many listeners) unless you are reasonably certain that any existing applications in that environment won't be adversely affected.

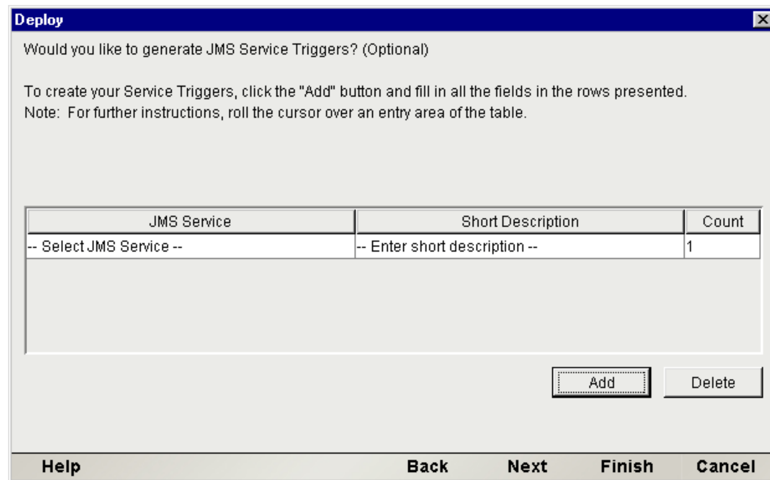
Click **No** if you wish to create the necessary connection objects but not send any test message.

14 Click **Finish**. The JMS Service component is created and the Service Editor window appears.

Deployment of the JMS Service

A project containing JMS Service objects is deployed the same way as any other project, using the same Deployment Wizard. See the chapter called "Deploying a Project" in your *exteNd Composer Enterprise Server User's Guide* for step-by-step instructions on how to deploy a project.

The only part of the Deployment Wizard that is different for users of the JMS Connect is that the wizard includes a special panel as shown below.



The purpose of this panel is to allow you to create a *MessageListener* object for each JMS Service that you wish to deploy, automatically associating each service with the `onMessage()` handler of the appropriate listener object. (At deployment time, listeners are registered with the JMS queue or topic connections specified in your respective JMS Service components.)

➢ **To create a JMS Service Trigger**

1. Click the **Add** button in the above panel. A new row appears in the inset pane.
2. Click in the space where it says "-- Select JMS Service **--".** A pulldown menu appears, listing the names of all available JMS Service objects in your project.
3. Select a JMS Service object and release the mouse button.
4. Under "Short Description," enter the plain-text descriptive info you would like to associate with this JMS Service.

   NOTE: This field cannot be left blank.
5. Under "Count," enter the number of listeners you would like to associate with this service. The default value is one.
6. Repeat the above procedure as many times as necessary to add new JMS Services to the deployment.
7. Click **Next** or **Finish**, as appropriate.

The listener objects for each JMS Service in your project will now exist on the server. If you are using the Novell application server, there is no need to restart the server; listener objects are active as soon as deployment is complete.

How Do I Manage Deployed JMS Services?

Once a project containing JMS Services has been deployed, the *MessageListener* objects for the various services will be actively listening for messages each time you (re)start your server. To start/stop these services individually, or to remove them from the server altogether, you need to gain access to the exteNd JMS Services Console. This browser-based console will allow you to see a list of JMS Services (along with the descriptive info you supplied in the deployment wizard), the status of each service (active or inactive), the running tally (Count) of messages received, and other administrative information. You will also see buttons labeled Start/Stop and Remove (one per service).

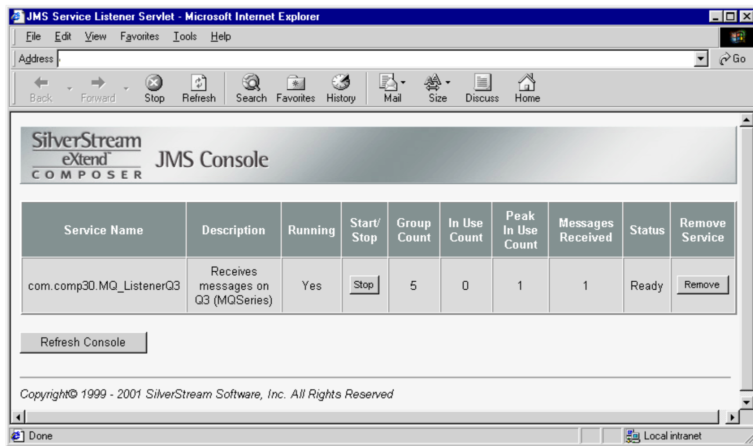
➢ **To gain access to the exteNd JMS Services Console**

1. Be sure your application server is running.
2. Launch your web browser and go to

   http://[hostname]/extendComposer/jmsConsole

   where **[hostname]** is the name (and :port) of your server; for example, "localhost:80."

3 The console window appears, listing any JMS Services that have been deployed.



4 To stop a JMS Service, hit the appropriate **Stop** button. (The button will then change to Start.)

NOTE: If messages are in the process of being handled by a service at the time of the **Stop** command, there may be some delay before the service actually exits. Hit the **Refresh** button periodically until the “Running” column of the console says No for the service(s) in question. The exact amount of latency you can expect when stopping a service is impossible to predict, since it depends on traffic conditions and vendor-specific JMS implementation details. You should consult your provider’s documentation for information that may be helpful in suspending execution of a pub/sub topic.

5 To remove a JMS Service permanently, hit the appropriate **Remove** button.

Note that if a message is waiting on a queue or topic, hitting **Start** on the console page (to reinitiate a service) will cause the service’s `onMessage()` method to be called immediately, but the Count field of the console (which normally displays a running total of messages received) will not update automatically. To make the Count display correctly, hit the browser’s Refresh button after Starting a service.

A JMS Glossary

Administered Object

JMS defines two administered objects: *Destinations* and *ConnectionFactories*. The former kind of object associates a topic name or queue with a physical resource; the latter exposes the methods by which a client can connect to a JMS provider's service daemon. Both kinds of objects involve resources that are under administrative control. JMS clients are able to find administered objects by looking them up in a namespace using JNDI.

Asynchronous Delivery

In Publish/Subscribe messaging, *asynchronous delivery* occurs when the message broker (or topic manager) calls the *MessageListener*'s `onMessage()` method. In the synchronous case, by contrast, the receiving application obtains messages by requesting them.

BytesMessage

One of the five JMS-defined message types. The body of this type of message consists literally of a byte array; hence, it can represent any kind of payload.

CICS

Customer Information Control System: An IBM protocol for conducting and monitoring transactions with mainframes.

Commit

In a transacted message session, calling the session's `commit()` method causes any produced messages to be (irreversibly) sent and any consumed messages to be *acknowledged* (and thus removed from the queue). See also *Transaction* and *Rollback*.

Connection

A connection represents the collection of server-side and client-side objects needed to establish and manage *sessions*. Connections can be of the QueueConnection type or the TopicConnection type. Creation of connections occurs via a ConnectionFactory object (accessible via JNDI).

Copybook

A record structure consisting of individually defined COBOL data descriptors.

Datagram

Although not a JMS-defined term, the word *datagram* is frequently used in messaging. It generally refers to a short message, often an administrative notification of some kind, sent in "fire and forget" manner (i.e., with no expectation of any reply).

Destination

JMS Queues and Topics extend `javax.jms.Destination.` Thus, a JMS destination is equivalent to a queue or topic. Destinations are created administratively and bound to a JNDI name at the time of creation.

DOM

Document Object Model. An industry standard way of describing or representing the containment hierarchy for an XML or HTML file.

Durable Subscriber

A message receiver in a Publish/Subscribe setting (see "Publish/Subscribe," below) can register to receive messages even when offline. Such a subscriber is said to be *durable*, since the receiver's status persists beyond any given session.

Endian

Term used to describe the order in which bytes are stored in computer memory. Big-endian is an order in which the most significant value in the sequence is stored at the lowest memory address (the opposite of little-endian). Intel has traditionally used a little-endian architecture, where most other chip makers have favored a big-endian architecture.

JMS

Java Messaging Service. A Sun-developed Java interface for message services, defining indsutry-standard objects and behaviors for Message Oriented Middleware. A JMS-compliant MOM implements the interfaces defined in JMS.

JMS Provider

Any MOM system that implements the JMS interface.

JNDI

Java Naming and Directory Interface: a standard extension to the Java platform, giving Java applications a unified interface to multiple naming and directory services.

JTA

Java Transaction API: a Java API for delimiting distributed transactions.

MapMessage

One of five JMS-defined message types, consisting of name/value pairs. The keys are Java *Strings* and the values are Java primitive types.

MessageListener

A Java interface that applications can implement in a Publish/Subscribe system that allows the application to receive automatic notification of incoming messages.

MOM

Message Oriented Middleware. A software system (e.g., IBM's MQSeries software) that implements enterprise messaging.

Native Environment Pane

A pane in the JMS Component Editor that displays various attributes (such as header fields and body content) associated with a message.

NON_PERSISTENT

One of the two JMS-defined delivery modes (the other being PERSISTENT; see below), guaranteeing at-most-once delivery. Because the message is not written to nonvolatile storage at any point, system outages can result in loss of the message when this mode is used; however, overhead is lower with this mode than with PERSISTENT.

ObjectMessage

One of the five JMS-defined message types, in which the body contains a serialized Java object.

PERSISTENT

One of the two JMS-defined delivery modes (the other being NON_PERSISTENT). Use of the PERSISTENT mode guarantees that a message will be delivered once and only once. The message is written to nonvolatile storage to avoid any possibility of loss in transit.

Point-to-Point (PTP)

One of two main messaging paradigms in popular use (the other being Publish/Subscribe messaging). In PTP systems, queues are not organized by topic but instead typically “belong” to dedicated receivers (client apps), who treat queues much like mailboxes. Clients send messages to and receive messages from other clients with a minimum of administrative intervention. Receiving apps may

optionally implement *selectors* (or filters) that allow for preferential retrieval of messages based on special criteria.

Publish/Subscribe

One of two main messaging paradigms in popular use (the other being Point-to-Point messaging). In Publish/Subscribe, queues are often called *topics*. (See below.) They differ from ordinary queues in that topics are designed to be shared by numerous "listeners," whereas in Point-to-Point messaging a queue is customarily associated with one receiving app (or at least a small, well-defined number of users). Because topics are shared, a message is not removed from a topic until every registered listener has received it. Also, filtering (which in Point-to-Point messaging is done via message *selectors*) is under administrative control in Pub/Sub systems, rather than being under the control of receiving apps. Clients that post messages to a topic are said to be "publishers," while clients that consume those messages are "subscribers."

Queue

In JMS-based messaging systems, messages are sent not to clients but to *queues*—which is to say, storage repositories set up to handle messages. Interposing queues between senders and receivers assures that even when a client is unavailable, messages addressed to the client are still able to be cached for later retrieval. Queues are typically created administratively and exposed to message clients as static resources.

RPC

Remote Procedure Call: A protocol in which a program or procedure is remotely invoked via a synchronous session with a mainframe or server.

Rollback

In a transacted message session, calling the session's rollback() method causes any produced messages to be discarded (not sent) and any consumed messages to be left on the queue as if nothing happened. See also *Commit* and *Transaction*.

Selectors

In Point-to-Point messaging, a client can use a *selector* to filter messages based on header content. The selector is basically a conditional statement (i.e., a statement that evaluates to *true* or *false*) involving a header or property value, written in a syntax similar to SQL92. (See Appendix C.)

Session

A session is a lightweight JMS object for producing and consuming messages. A session retains retrieved messages until they have been acknowledged. All send and receive actions are scoped to sessions.

SQL92

An implementation of Structured Query Language (commonly used for database queries); the basis for JMS message selector syntax.

StreamMessage

One of the five JMS-defined message types, wherein the body of the message consists of Java primitive values. The body of this type of message is intended to be read sequentially using methods like `readLong(), readString()`, etc.

TextMessage

One of five JMS-defined message types. The body of a TextMessage is a String.

Time-to-Live

The effective lifespan of a message. Message expiration is calculated on the basis of the time when the message was sent plus the Time-to-Live value.

Topic

In Publish/Subscribe messaging (see "Publish/Subscribe" above), messages queues are often called *topics*. In essence, a topic *is* a queue; it differs from a queue mostly in the way it is administered. Topics are typically shared by many users and may form nodes in a content hierarchy (although this is not a requirement of JMS). Many users may "publish" to one or more topics. Conversely, a given topic may have many "subscribers."

Transaction

JMS message sessions can group one or more receive, send, or browse actions into a *transaction*, which means all operations are conducted as a unit. If a transaction succeeds, all of its constituent operations succeed. If a transaction fails, all operations are "rolled back" to restore the original state that existed before the session began. Note that JMS commit and rollback methods are scoped to the JMS *session* and therefore do not affect other program operations.

B Message Selector Syntax

A message selector is a *String* containing an expression that, if it evaluates to TRUE, will result in messages being selected, or if FALSE results in messages being neglected. The syntax of the JMS selector expression is based on a subset of SQL92. The order of evaluation of a message selector is from left to right within precedence level; but parentheses can be used to alter the evaluation order. For consistency, predefined selector literals and operator names are shown in upper case below (but are nevertheless case-insensitive).

A selector can contain tokens, operators, and expressions conforming to the rules outlined hereunder.

Literals

A string literal is enclosed in single quotes. If a string literal is to contain an included single quote, it can be represented by a doubled single quote: for example, 'its' and 'it''s'. As with Java *String* literals, the Unicode character encoding is presumed.

An *exact* numeric literal is a numeric value without a decimal point, such as 59, -257, +82, etc. Numbers in the range of Java *long* are supported. Exact numeric literals use the Java integer literal syntax.

An *approximate* numeric literal is a numeric value in scientific notation, such as 7E4, -27.9E2 or a numeric value with a decimal such as 7., -95.7, +16.2; numbers in the range of Java *double* are supported. Approximate literals use the Java floating point literal syntax.

A boolean literal can have a value of *TRUE* or *FALSE*.

Identifiers

Identifiers can be either header field references or property references. An identifier is a character sequence that begins with a Java-identifier start character and is followed by characters that are Java-identifier part characters. An identifier start character is any character for which the method *Character.isJavaIdentifierStart()* returns *true*. This includes underscore and **$**. An identifier part character is any character for which the method *Character.isJavaIdentifierPart()* returns true.

Identifiers cannot be *NULL*, *TRUE*,no*r FALS*E.

Identifiers cannot be *NOT, AND, OR, BETWEEN, LIKE, IN*, nor *IS*.

Identifiers are case-sensitive.

Message header field references are restricted to *JMSDeliveryMod*e, *JMSPriorit*y, *JMSMessageID*, *JMSTimestamp, JMSCorrelationID*, and *JMSTyp*e.

JMSMessageID, JMSCorrelationID, and *JMSType* values may be *null* and if so are treated as a NULL value.

Any name beginning with 'JMSX' is a JMS-defined property name.

Any name beginning with 'JMS_' is a provider-specific property name.

Any name that does not begin with 'JMS' is an application-specific property name. If a non-existent property is referenced, its value is NULL. If it does exist, its value is the corresponding property value.

Whitespace is the same as that defined for Java: space, horizontal tab, form feed and/or line terminator.

Expressions

A selector is a conditional expression. Any selector that evaluates to *true* matches; a selector that evaluates to *false* or unknown does not match.

Arithmetic expressions are composed of arithmetic operators, identifiers with numeric values, numeric literals and/or other arithmetic expressions.

Conditional expressions are composed of comparison operators, logical operators, identifiers with boolean values, boolean literals, and/or other conditional expressions.

Standard bracketing () for ordering expression evaluation is supported.

Logical operators in precedence order: `NOT, AND, OR`

Comparison operators: `=, >, >=, <, <=, <>` (not equal)

Only *like* type values can be compared. One exception is that it is valid to compare exact numeric values and approximate numeric values. (The necessary type conversion is conducted according to the rules of Java numeric promotion.) If the comparison of non-like type values is attempted, the selector is always false.

String and *Boolean* comparisons are restricted to = (equal) and <> (not equal). Two strings are equal *if and only if* they contain the same sequence of characters.

Arithmetic operators in precedence order:

+, - unary

*, / multiplication and division

+, - addition and subtraction

NOTE: Arithmetic operations must use Java numeric promotion.

Comparisons

- *arithmetic-expr1* [NOT] BETWEEN *arithmetic-expr2* and *arithmetic-expr3*

Example:

`age BETWEEN 15 and 19` is equivalent to `age >= 15 AND age <= 19`

`age NOT BETWEEN 15 and 19` is equivalent to `age < 15 OR age > 19`

- *identifier* [NOT] IN *(string-literal*1, *string-literal*2,...), where *identifier*is a *String* or NULL value.

Example: `Country IN (' UK', 'US', 'France')`

is true for 'UK' and false for 'Peru'. It is equivalent to the expression:

`(Country = ' UK') OR (Country = ' US') OR (Country = ' France')`

Example: `Country NOT IN (' UK', 'US', 'France')`

is false for 'UK' and true for 'Peru'. It is equivalent to the expression:

`NOT ((Country = ' UK') OR (Country = ' US') OR (Country = ' France'))`

If *identifier* in an IN or NOT IN operation is NULL, the value of the operation is unknown.

- *identifier* [NOT] LIKE *pattern-value* [ESCAPE *escape-characte*r], where *identifier* has a *String* value; *pattern-value* is a string literal where '_' (underscore) stands for any single character; '%' stands for any sequence of characters (including the empty sequence); and all other characters stand for themselves. The optional *escape-character* is a single-character string literal whose character is used to escape the special meaning of the '_' and '%' in *pattern-value.*

`phone LIKE '12%3'` is true for '123' or '12993' and false for '1234'

`phone NOT LIKE '12%3'` is false for '123' and '12993' and true for '1234'

`word LIKE 'l_se'` is true for 'lose' and false for 'loose'

`underscored LIKE '_%' ESCAPE '\'` is true for '_foo' and false for 'bar'

If *identifier* in a LIKE or NOT LIKE operation is NULL, the value of the operation is unknown.

- *identifier* IS NULL tests for a null header field value, or a missing property value.
- *identifier* IS NOT NULL tests for the existence of a non null header field value or property value.

The following message selector selects messages with a message type of *car* and color of *red* and weight greater than 3500 lbs:

```
"JMSType = 'car' AND color = 'red' AND weight > 3500"
```

Null Values

As noted above, header fields and property values may be NULL. The evaluation of selector expressions containing NULL values is defined by SQL 92 NULL semantics. I.e., SQL treats a NULL value as unknown. Comparison or arithmetic with an unknown value always yields an unknown value. The IS NULL and IS NOT NULL operators convert an unknown header or property value intoTRUE or FALSE values.

Special Considerations

When used in a message selector, *JMSDeliveryMode* will have the value 'PERSISTENT' or 'NON_PERSISTENT'.

Date and time values should use the standard Java long millis value. When including a date or time literal in a message selector, it should be an integer literal for a millis value. The standard way to produce millis values is to use *java.util.Calendar*. Although SQL supports fixed decimal comparison and arithmetic, JMS message selectors do not. (This is the reason for restricting exact numeric literals to non-decimals.)

SQL comments are *not* supported.

C Message Headers and Properties

Header Fields Defined by JMS

In JMS, all messages support the same set of predefined header fields, which are described below. Note that most header fields will have their values set automatically at runtime either by exteNd Composer or the MOM vendor.

JMSCorrelationID

A JMS client can use the *JMSCorrelationID* header to associated one message with another (for request/response situations). This field can hold an arbitrary string value, obtained by mapping a node value from an Input DOM, or perhaps created dynamically with the aid of ECMAScript, etc. Use of this field is not mandatory.

JMSDeliveryMode

This header field contains the delivery mode (PERSISTENT or NON_PERSISTENT) specified when the message was sent. At the start of a send session, this header field is ignored; after the send has been accomplished, it holds the delivery mode specified by the sending method.

This field will be filled out for you, using the persistency value you chose in the Send Message setup wizard. (See next chapter.)

JMSDestination

The *JMSDestination* header field is ignored at the time a message is sent; after the send, it contains the destination object specified by the sending message.

You do not need to enter anything manually in this field, since the necessary connection information (i.e., choice of destination queue) was automatically set when you first created the JMS Component.

JMSExpiration

You will not need to enter anything manually into this field. The Send Message setup wizard will prompt you for (among other things) a Time-to-Live for the outgoing message. During the "send" session, when the message is actually ready to be sent, exteNd will calculate the message's expiration time as the sum of the Time-to-Live value and the current UTC time (both values in milliseconds). After the send is completed, the message's *JMSExpiration* header field will contain this sum. If the Time-to-Live value was zero when the Send Message action was created, the message will have no expiration value (which means it will not expire).

> NOTE: In most JMS-based MOMs, clients never receive expired messages.

JMSMessageID

The *JMSMessageID* value uniquely identifies a message in the MOM environment. It is set automatically by the JMS provider and is read-only (and only after a message has been sent).

JMSPriority

The *JMSPriority* field holds a string value containing one of ten values ('0' through '9') reflecting the message's priority. This field value is filled out automatically with the value you supplied in the Send Message setup wizard. A value of '0' to '4' indicates a range of normal priorities (with '4' being the default); '5' to '9' are gradations of *expedited* priority.

> NOTE: The manner by which priority settings determine message ordering in a queue is not defined by JMS. Consult your MOM vendor's documentation for information about how this feature might affect message ordering.

JMSRedelivered

If a client application receives a message that has the *JMSRedelivered* marker set, it is possible that the queue manager tried to deliver the message earlier, but the message was not acknowledged by the client (perhaps due to a system failure). This field is under the control of the queue manager or message broker. It is not under application control.

JMSReplyTo

The *JMSReplyTo* field is designed to contain a Destination supplied by a client when a message is sent. It represents the destination where a reply to the message (if any) should be sent.

> NOTE: This header field is not currently exposed as a write-enabled item in exteNd Composer's JMS Component Editor.

JMSTimestamp

This field contains the time that a message was handed off to a provider to be sent. It may or may not be the actual transmission time, depending on whether (for instance) the message's "send" session is under transactional control.

JMSType

This user-settable field contains an arbitrary string supplied by a client when a message is created. The sender can assign any value to *JMSType* that a receiver might find useful. For example, application-defined *JMSType* values could facilitate message filtering by making it possible for various receivers to handle various specific message types.

> NOTE: Some JMS providers store message type definitions in a repository and may expect runtime values in *JMSType* that correspond to these type definitions. If this is the case with your MOM environment, use symbolic values for *JMSType* that correspond to legal values defined in the applicable repository. (Consult your MOM documentation for details.)

Message Properties

Message properties serve, in effect, as extra header fields. JMS allows for three broad categories of properties: JMS-defined properties, provider-specific properties, and user-defined properties. JMS Connect supports all three categories, although JMS does not *require* applications to support properties (other than *JMSXGroupID* and *JMSXGroupSeq*; see below).

Property values (if not null) must be of type *boolean, byte, short, int, long, float, double*, or *String*. The allowable values for specific predefined properties are described further below.

Property values, if present, are set by the sender prior to sending a message. When a message is received by a client, all properties are read-only. Any attempt by the client to set a property value on a retrieved message will result in a *MessageNotWriteableException* being thrown.

JMS-Defined Properties

JMS defines (and the JMS Connect exposes) a number of JMS-specific property fields that can optionally be populated by message clients and/or providers. These JMS-defined properties, which are prefixed with 'JMSX', include:

- JMSXUserID (String) — Arbitrary string identifying the user who is sending the message. (This is intended to be set by the provider during a send operation.)
- JMSXAppID (String) — Identity of the sending application. (This is intended to be set by the provider during a send operation.)
- JMSXDeliveryCount (int) — The number of message delivery attempts. (Set by provider.)

- JMSXGroupID (String) — The (client-settable) identity of the message group that this message is a member of. Intended for use by clients who are sending messages in batches.
- JMSXGroupSeq (int) — The (client-settable) sequence number of this message within a group.
- JMSXProducerTXID (String) — Identifier of the transaction within which this message was produced (set by the provider).
- JMSXConsumerTXID (String) — Identifier of the transaction within which this message was consumed (set by the provider).
- JMSXRcvTimestamp (long) — The time when a message was delivered to its ultimate consumer (set by provider).
- JMSXState (int) — One of 1 (waiting), 2 (ready), 3 (expired), 4 (retained). Not relevant to the client app; for internal use of the provider.

Provider-Specific Properties

JMS allows providers to define their own public property names, with a prefix of "JMS_<vendor name>" (e.g., `JMS_IBM` is the prefix for IBM-defined properties). Although the JMS Connect exposes these fields in JMS message header tree views, they are really intended for the JMS provider's use.

When IBM's MQSeries is the provider, exteNd's JMS Connect exposes three vendor-specific properties: `JMS_IBM_MsgType`, `JMS_IBM_PutApplType`, and `JMS_IBM_Format`. After a message has been received by a JMS Component, these fields will typically be populated with MQSeries-specific control information. On outgoing messages, you can either supply appropriate values in these fields yourself, or leave them blank. See the MQSeries *Application Programming Reference* for information on the semantics of these fields.

User-Defined Properties

JMS allows users to define their own custom properties, and the JMS Connect exposes this functionality in the JMS Component Editor as described further below. There is no restriction on the number or kinds of user-defined property fields that can be attached to a message, except that the names of user-defined properties must (like all headers and properties) obey the syntax rules for message selector identifiers.

Index

A

acknowledgement 75
Action Model 51
Action Model pane 48
actions 51
administered objects 29, 121
After Send Maps 62
ambiguous transaction state 76
assured once-only delivery 16
asynchronous delivery 121
asynchronous processing 14
asynchronous retrieval 22
asynchronous triggering 111
at-most-once 20
authentication 24, 32, 38
automatic-rollback protocol 76

B

batch mapping XML elements 97
Before Send Maps 62
blocking/polling 22
body type
  filtering for 72
body types 21
Break command 69
broadcaster/listener 22
Browse Messages action 64
  queues and 43
browser console 118
browsing 64
browsing vs. reading 77
Bytes Message 104
BytesMessage 21

C

CICS 14
CICS RPC Enterprise Enabler 91
COBOL 91, 122
COBOL copybook See copybook
colon 100
commit 17, 31, 35, 38, 41, 75
  automatic 76
comparison operators 128
component editor 48
ConnectionFactories 29
connection resource
  creating 27
connections 29
  MQSeries Topic 39
  provider-specific 33
  queue 29
  topic 36
context-sensitive picklists 104
Continue command 69
copybook 22, 50, 122
custom header property 84
custom properties 54
custom property 67

D

database operation 77
datagram 23, 122
data types 85
delivery guarantees 19
deployment issues 117
destination 122
destinations 29
destinations, changing 55
destructive removal 69
distributed transactions 17
drag-and-drop 83
drop targets 83
DTDs 42
durable subscriber 19, 122

E

ECMAScript 81, 83, 89
  getters & setters 107
  method summary 107
ECMAScript extensions 104

Error on No Message checkbox 70
exceptions 31, 35, 38, 41, 72, 101
    TransactionInProgressException 17
Expand Tree 95
expression, selector 128
Expression-Driven Connections 28
Expression Editor 88

F

failover 24
FIFO (first-in/first-out) 15
filter 23, 85
filtering
    body type 100
    limitations on 100
fire and forget 23
forbidden drag operation 83

G

getField() 108
getJMSBytesBody() 108
getJMSMapField() 109
getJMSObjectBody() 109
getJMSStreamField() 109
getter methods 107

H

hasMessages() 69
header, message
    before send 62
    mapping data into 81, 83
Header Document Name 61
Host Name 35
HTTP servlets 112
hub and spoke architecture 12

I

IBM 13, 29
IBM message properties 134
identifier, selector 127
Initial Context Factory, JNDI 32, 38
IS NULL 130
iterating through all messages 69
Iterating through Messages 69

J

java.util.Enumeration 64
JDBC 77
JMS Component
    creating a new 44
JMS Connection resource 113
JMSCorrelationID 20, 63, 83, 100, 131
JMSDeliveryMode 20, 62, 100, 131
JMSDestination 20, 62, 131
JMSExpiration 20, 62, 132
JMS_IBM_Format 134
JMS_IBM_MsgType 134
JMS_IBM_PutApplType 134
JMSMESSAGE 69
JMSMessageID 20, 62, 69, 100, 132
    loop termination based on 75
JMS-MQSERIES 106
JMSPriority 20, 62, 99, 100, 132
JMSRedelivered 21, 62, 132
JMSReplyTo 21, 132
JMS Service 111
JMS Services
    removing 118
    starting and stopping 118
JMS Services Console 118
JMS Service Trigger 118
JMS standard 13
    what's not covered by 24
JMSTimestamp 21, 62, 100, 133
JMSType 21, 83, 99, 100, 133
JMSXAppID 133
JMSXConsumerTXID 134
JMSXDeliveryCount 133

JMSXGroupID 134
JMSXGroupSeq 134
JMSXProducerTXID 134
JMSXRcvTimestamp 134
JMSXState 134
JMSXUserID 133
JNDI 29
JTA (Java Transactions API) 17

L

latency 16, 19, 22
limitations
  on property mapping 85
literals, selector 127
little-endian 92
load balancing 24
loop control 69
looping on Receive 69

M

mailbox 18
Map command 88
MapMessage 21
mapping, headers and 83
Mapping Name 67
message
  iteration 69
  structure 20
message acknowledgement 75
Message Body DOM 67
message broker 18
Message Filter 64, 69, 72, 99
Message Filter tab 23
MessageListener 22, 111, 117, 118
message properties 133
message queues 15
messages
  asynchronous retrieval of 22
  copybook 22
  filtering 23
  iteration 69
  lifespan of 16
  maximum number on a queue 16
  read-only nature of 21
  selectors 23
  type definitions 24
  XML 22
message selector See selector
Message Transaction action 75, 78
Message Transaction dialog 78
Message Type pulldown menu 66
millisecond values 60
model queue 35
MQSeries 11, 13, 14, 29, 34, 134
MQSeries Host Machine 35
MQSeries queue 33
multiple listeners 112
multitasking 16

N

Native Environment pane 48, 49, 63
nondestructive read 77
Non-JMS Client 36
NON_PERSISTENT 100
NON_PERSISTENT mode 20
nonrepudiation 83

O

Object Message 104
ObjectMessage 21
once-and-only-once 20
onMessage() 22
OS/2 92
Override Connection Queue 65

P

password 35
performance issues 16
PERSISTENT 100
PERSISTENT mode 19
pick lists 104
Point-to-Point
  browsing and 64
Point-to-Point (PTP) 17, 123
priority 20
privacy 24

properties 20, 133
properties, custom 61
Property Name 67
Property Type 67
providers 30
provider-specific properties 134
Provider URL 32, 38
PTP See Point-to-Point
Publish/Subscribe 124
Publish/Subscribe (pub/sub) 18
    browsing not defined 64
pulling vs. pushing data 22

Q

quality-of-service 16
Queue Manager 35
queues
    browsing 64
    changing 64
    clustered 15
    empty 70
    in pub/sub 18
    temporary model 35
    using two or more 44

R

receive
    blocking during 22
Receive Message action 69
Receive Message Maps 75
reliability 16, 20
Remote Procedure Call 14
Repeat While action 69
replying to messages 101
repository 24
request-response 23, 83
request-response messaging 101
resource overruns 16, 20
rollback 17, 31, 35, 38, 41, 69, 75
RPC 17, 124

S

sample documents 42
scalability 24
scope of transaction control 77
Security Principal 32, 39
Select Occurrences 95
selector 23, 64, 69, 72
    grammar 127
Send Message action 55
serializable Java object 21
service trigger 118
setJMSBytesBody() 108
setJMSMapField() 109
setJMSMsgProperty() 109
setJMSObjectBody() 109
setJMSStreamField() 109
setter methods 107
setValue() 108
shopping-cart app 16
SonicMQ 13, 14
SQL92 23, 130
stopping JMS Service listeners 119
store/forward 23
StreamMessage 21
stub documents 42
stylesheets 42

T

temporary model queue 35
Temp XML Document 46, 115
Test Options dialog 36
TextMessage 21
timeout value 22
topic connections 36
    browsing not allowed 43
TopicPublishers 19
topics 18
TopicSubscribers 19
Transacted checkbox 31, 35, 38, 41, 44, 78
transaction control 17, 75
    scope of 77
TransactionInProgressException 17

Try/On Error action 70

U

unresolved transactions 76
Use Prior Message ReplyTo Field 102
username 35
Use Sent Message ReplyTo Field 65

W

WHILE loop 69

X

XAResource interface 17
xconfig.xml 30
XML Map Component 48
XML stub document 61, 67
XML templates 42, 67
XPath 83, 88
XPath() method 84
XSL 42