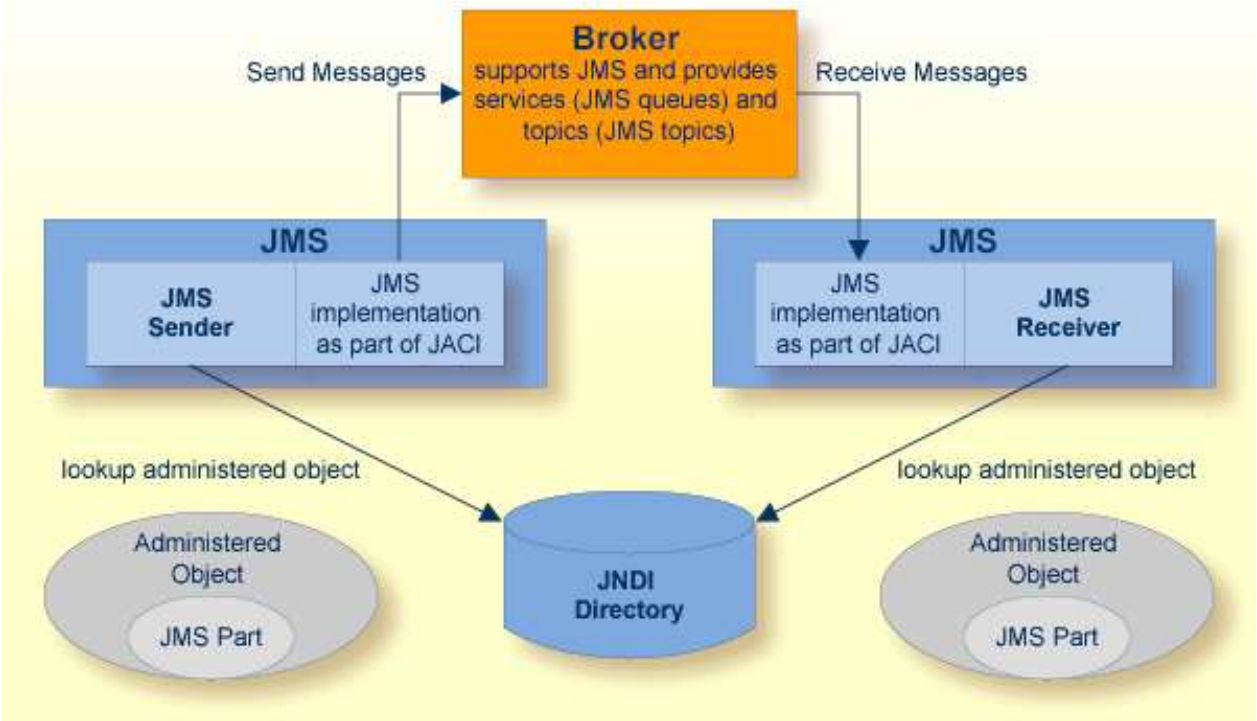
Writing JMS Applications with the EntireX Broker

Java Message Service (JMS) is a standard API for enterprise messaging services. This chapter describes how to write JMS-based applications with the EntireX Broker. It covers the following topics:

- EntireX Broker and JMS
- Writing JMS Applications
- Writing Advanced Applications
- Connecting JMS Applications and non-JMS Applications
- JMS Error Handling

EntireX Broker and JMS

The EntireX Broker is enabled for Java Message Service (JMS). JMS is supported with components on top of Java ACI. JMS in general uses two message models: *point-to-point messaging* and *publish-and-subscribe messaging*. The EntireX Broker supports both messaging models. JMS connections are mapped to Broker. JMS queues are mapped to services of the Broker. JMS topics are mapped to topics of the Broker.



The Broker must have EntireX version 7.1.1 or higher for point-to-point messaging. For publish-and-subscribe messaging, the Broker must have EntireX version 7.2.1 or higher. For point to point, the implementation uses units of work to communicate with the Broker. The configuration of the Broker for JMS includes enabling units of work and configuration of the persistent store for persistent JMS messages. Publish and subscribe uses publications and topics of the Broker. For the administration of the objects in the JNDI directory, use the Message Service Agent of the System Management Hub. See *Message Service Administration using System Management Hub*. The files *entirex.jar* and *exxjms.jar* are required to run JMS applications with the Broker.

Writing JMS Applications

Writing JMS applications with the EntireX Broker requires the following steps:

- Configure the Broker
- Configure the JNDI Provider
- Create the Administered Objects with the JMS Agent of the System Management Hub
- Coding and Compiling Your Application
- Running Your Application
- Examples

Configure the Broker

To enable the Broker for JMS Publish and Subscribe

Edit the Broker attribute file. The following examples show the attributes needed for JMS publish and subscribe. Adapt the numerical values to your needs. The values are examples.

1. Enable the Broker for publish and subscribe.

   ```
   PUBLISH-AND-SUBSCRIBE      = YES
   PUBLICATION-DEFAULT        = UNLIM
   SUBSCRIBER-DEFAULT         = UNLIM
   TOPIC-UPDATES              = YES
   AUTO-COMMIT-FOR-SUBSCRIBER = NO
   ```

   Note:
   Attribute `AUTO-COMMIT-FOR-SUBSCRIBER` is required because committing messages is controlled by JMS.

2. Configure the subscriber store for durable subscribers.

   Use `PSTORE` as subscriber store. For more details of the `PSTORE` configuration, see the *Broker Attributes*.

   ```
   SUBSCRIBER-STORE     = PSTORE
   NUM-SUBSCRIBER-TOTAL = 1000
   NUM-TOPIC-TOTAL      = 1000
   ```

3. Define the topics.

   The following example lists the attributes connected to topics. For a detailed description see the *Broker Attributes*. At least one topic definition with topic "*" is needed to enable the temporary topics of JMS. The names of the topics are restricted to 96 bytes. The lifetime of the messages is controlled by JMS.

```
DEFAULTS                       = TOPIC
ALLOW-DURABLE                  = YES
UNSECURE-SUBSCRIBE             = YES
AUTO-COMMIT-FOR-SUBSCRIBER     = NO
CONVERSION                     = SAGTCHA
LONG-BUFFER-LIMIT              = UNLIM
MAX-PUBLICATION-MESSAGE-LENGTH = 31647
MAX-MESSAGES-IN-PUBLICATION    = 5
PUBLICATION-LIMIT              = UNLIM
PUBLISHER-NONACT               = 5M
SHORT-BUFFER-LIMIT             = UNLIM
SUBSCRIBER-LIMIT               = UNLIM
SUBSCRIBER-NONACT              = 3M
TRANSLATION                    = SAGTCHA
SUBSCRIPTION-EXPIRATION        = 90D
TOPIC                          = *
```

To enable the Broker for JMS point to point

Edit the Broker attribute file as described below:

1. Set `MAX-UOW` to some appropriate value greater than 0.
2. Define the services for JMS.

   JMS queues are mapped to the service class JMS and the service `QUEUE` (or `TMPQUEUE` for temporary queues). The name of the queue is used as the server name. The default Broker attribute file that is installed contains the definitions for JMS. This enables the installed default Broker for JMS with non-persistent messages.

```
* ------------- ENTIREX/JMS example services ------------------------
DEFAULTS             =    SERVICE
  CONV-LIMIT         =      UNLIM
  CONV-NONACT        =         4M
  LONG-BUFFER-LIMIT  =      UNLIM
  NOTIFY-EOC         =         NO
  SERVER-NONACT      =         5M
  SHORT-BUFFER-LIMIT =      UNLIM
 CLASS = JMS,    SERVER = *,      SERVICE = QUEUE,    DEFERRED = yes
 CLASS = JMS,    SERVER = *,      SERVICE = TMPQUEUE, DEFERRED = yes
* -----------------------------------------------------------------
```

3. Configure the persistent store of the Broker (optional).

   Use the Broker attributes `STORE`, `PSTORE`, `PSTORE-TYPE` to configure the persistent store.

   The value `STORE=OFF` corresponds to `DeliveryMode.NON_PERSISTENT` and the value `STORE=BROKER` corresponds to `DeliveryMode.PERSISTENT`.

The defaults set for the Broker are overwritten by the `STORE` attribute of the service and this is overwritten by the value JMS sets.

If you use `DeliveryMode.PERSISTENT` in JMS, you have to use `PSTORE` to define the status of the persistent store and `PSTORE-TYPE` to define the type of persistent store. See *Broker Attributes* for details.

Configure the JNDI Provider

To use administered objects of JMS with a JNDI service provider, configure the JNDI service provider as described below:

To configure the JNDI Service Provider

1. Get the JAR files of the service provider and follow the service provider's documentation to deploy these JAR files.
2. Create or change the file *jndi.properties*. Add the path of this file to the Java classpath.

With the installation of EntireX, the JNDI file system service provider is configured. The JAR files *fscontext.jar* and *providerutil.jar* reside in the subfolder classes of the EntireX installation folder. The JNDI configuration *jndi.properties* is placed in the subfolder *etc*.

Create the Administered Objects with the JMS Agent of the System Management Hub

See *Message Service Administration using System Management Hub*.

Coding and Compiling Your Application

Compile your application with the *gf.javax.jms.jar*. The JAR files from EntireX are not needed. This ensures that the JMS application is portable between JMS providers.

Running Your Application

The following JAR files are required to run your application, in addition to the *gf.javax.jms.jar* and the JAR files for the JNDI provider.

- *entirex.jar*
- *exxjms.jar*
- *Jcup.jar*
- *Jakarta-regexp-1.2.jar*

Examples

Examples for JMS are in the subfolder jms of the examples folder. For a detailed description see the *README.TXT* in this folder. To compile and run the examples use *build.bat* or the *build.xml* script with Ant.

Basic Examples

SenderToQueue.java and *SynchQueueReceiver.java* can be used to send and synchronously receive a single text message using a queue. *SynchTopicExample.java* uses a publisher class and a subscriber class to publish and synchronously receive a single text message using a topic.

Intermediate Examples

The intermediate examples show listeners, conversion and types of messages:

SenderToQueue.java and *AsynchQueueReceiver.java* send a specified number of text messages to a queue and asynchronously receive them using a message listener (TextListener), which is in the file *TextListener.java*.

AsynchTopicExample.java uses a publisher class and a subscriber class to publish five text messages to a topic and asynchronously get them using a message listener (TextListener).

MessageFormats.java writes and reads messages in the five supported message formats. The messages are not sent, so you do not need to specify a queue or topic argument when you run the program.

MessageConversion.java shows that for some message formats, you can write a message using one data type and read it using another.

ObjectMessages.java shows that objects are copied into messages, not passed by reference: once you create a message from a given object, you can change the original object, but the contents of the message do not change.

BytesMessages.java shows how to write, then read a BytesMessage of indeterminate length. It reads the message content from a file.

Advanced Examples

The advanced examples show header fields, selectors, durable subscriptions, acknowledge modes, transacted sessions, and request/reply:

MessageHeadersQueue.java and *MessageHeadersTopic.java* illustrate the use of the JMS message header fields.

TopicSelectors.java shows how to use message header fields as message selectors. The program consists of one publisher and several subscribers. Each subscriber uses a message selector to receive a subset of the messages sent by the publisher.

DurableSubscriberExample.java shows how you can create a durable subscriber that retains messages published to a topic while the subscriber is inactive.

AckEquivExample.java shows that to ensure that a message will not be acknowledged until processing is complete. Use a receiver with `AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE`.

TransactedExample.java demonstrates the use of transactions in a simulated e-commerce application.

RequestReplyQueue.java uses the JMS request/reply facility, which supports situations in which every message sent requires a response.

SampleJMSClient.java and *SampleJMSServer.java* demonstrate sending requests and replies. The server uses a message listener and an exception listener.

Writing Advanced Applications

This section describes the features of JMS and how they are mapped to an EntireX Broker configuration and functions.

Persistent and Non-persistent Messages

For persistent messages, the persistent store of the Broker has to be configured. See *Configure the Broker* and *Broker Attributes* for more information. If the persistent store is disabled, only `DeliveryMode.NON_PERSISTENT` is supported. Sending messages with `DeliveryMode.PERSISTENT` to a Broker without persistent store results in exception "0078 0388: PSI: UOWs canNOT be persisted".

Acknowledge Modes

The acknowledge modes `DUPS_OK_ACKNOWLEDGE`, `AUTO_ACKNOWLEDGE`, and `CLIENT_ACKNOWLEDGE` are supported for non-transacted sessions. In the mode `CLIENT_ACKNOWLEDGE`, the method `acknowledge()` for a message sends a `SYNCPOINT` with option `EOC` to the Broker. For the other modes, the same is done automatically.

Transacted Sessions

For transacted sessions, `commit()` sends a `SYNCPOINT` with option `EOC` to the Broker. This sets the status of the UOW to "accepted" for the sender and "delivered" for the receiver. The `rollback()` method sends a `SYNCPOINT` with option `BACKOUT` to the Broker. This sets the status of the UOW to "backed out" for the sender and "accepted" for the receiver.

Security

EntireX security is supported with the method `ConnectionFactory.createConnection(userName, password)`. This uses a logon to the Broker with user and password.

Receiving Messages with a MessageListener

To receive messages with a MessageListener, implement the `onMessage` method of the interface `MessageListener`. Since this method does not throw JMSExceptions, it is appropriate to set up an `ExceptionListener` for the connection. This listener gets all the exceptions thrown by the `MessageListener`. If the Broker returns a shutdown to the receiver (`BrokerExceptions` with class and code 0010 0050 or 0010 0051), the `ExceptionListener` can handle this exception and stop the connection.

Restrictions

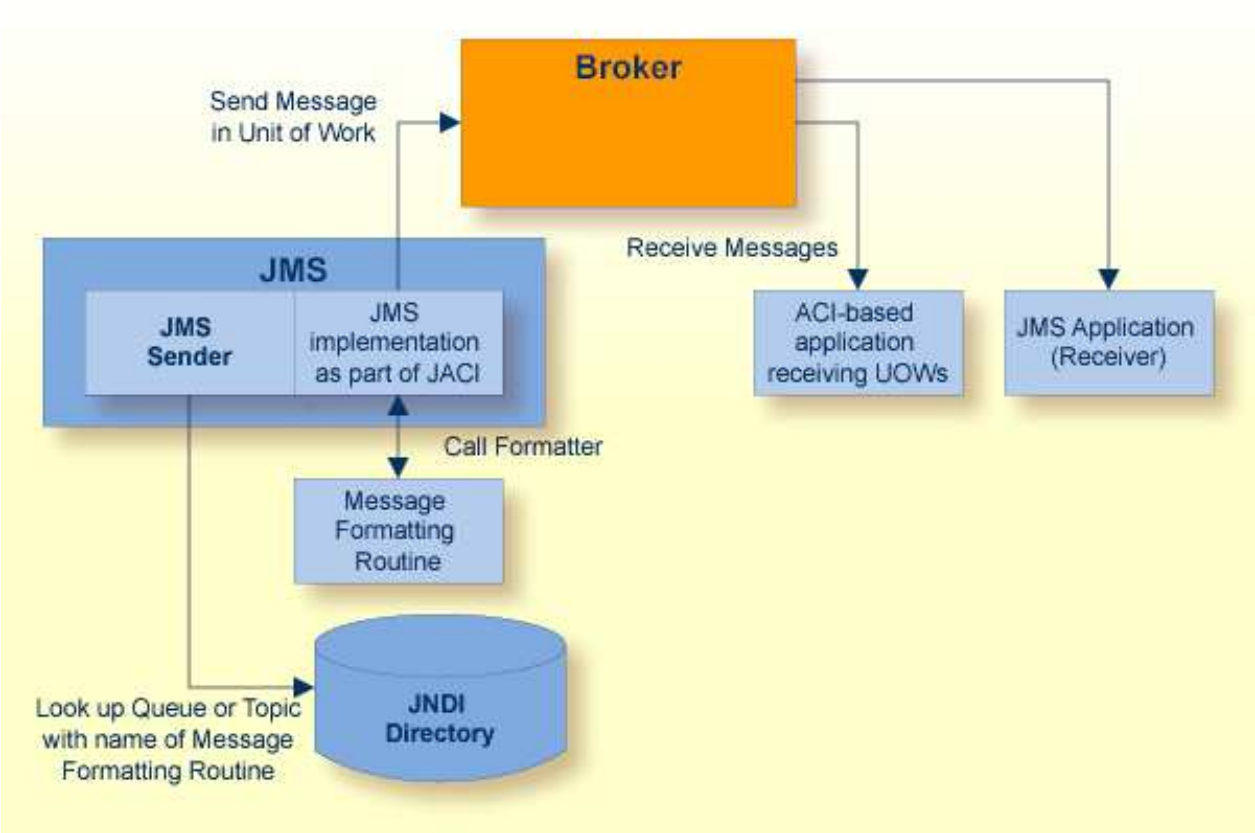
- The property `JMSXDeliveryCount` is not supported.

- Transactions with `XAConnectionFactory`, `XAConnection`, `XASession` are not supported.
- `setDisableMessageID` is ignored. The message ID is always set.
- `setDisableMessageTimestamp` is ignored. The message timestamp is always set.
- Maximum length of subscription name is 32 bytes.
- Maximum length of connection ID is 32 bytes.
- Maximum length of topic names is 96 bytes.
- Maximum length of queue names is 32 bytes.
- Maximum of 1000 receivers and senders in one connection
- Administrative Settings of user ID for connection factories are not supported

Connecting JMS Applications and non-JMS Applications

To connect JMS applications with non-JMS applications you need to modify the format of the messages. This is done by a message formatting RRoutine. The Message formatting routine formats the JMS messages sent by the JMS application in such a way that a non-JMS application can receive them. The routine converts messages from the non-JMS application into JMS messages. The message formatting routine is a user-written class which implements the interface `com.softwareag.entirex.jms.JMSFormatter`. We deliver examples for message formatting routines in the examples folder. These examples can be used as prototypes for your own routines.

The image below illustrates the general concept of JMS-to-non-JMS connections:



To connect JMS applications with non-JMS applications, consider the following aspects:

- Message Format
- Message Encoding
- Transaction Handling
- Configuration

Message Format

Implement your own format with a class that implements JMSFormatter. This format may include the text (for TextMessage) or other data (for BytesMessage, StreamMessage, MapMessage, and ObjectMessage) and properties of JMS.

Message Encoding

Format all data in the message as strings in the default encoding of the JVM. This ensures that translation inside the Broker works. Obey that the Broker translation may change the number of bytes for a field. A second approach is to use the encoding that the non-JMS application needs in the formatter and disable translation or conversion for the queue in the Broker.

Transaction Handling

The non-JMS application has to send the messages in conversations containing one unit of work. The unit of work may contain one or more messages. The JMS application is not able to receive more than one unit of work in a conversation. Do not use the `USTATUS` field of the unit of work. This is reserved for the receiving JMS application.

A receiving non-JMS application receives the messages in units of work. Each unit of work has its own conversation. The unit of work contains one or more messages.

Configuration

For the JMS queues or topics that should connect to non-JMS applications, set the formatter to the name of the class implementing `com.softwareag.entirex.jms.JMSFormatter`. This enables customer-specific formatting of the message for the JMS application. The formatter is set for each queue or each topic individually. Queues and topics connecting only JMS applications do not need a formatter. JMS and non-JMS applications can be mixed in a queue with a formatter. The same applies to topics.

The examples show formatting of text messages with a Natural example application and a Java example application. For detailed instructions on how to run these examples, see the examples folder.

Assume the following scenario: a JMS application sends a message and expects the reply in a special queue. The ACI application has to get the name of the queue from the message and send the reply to this queue. This is achieved in the following manner:

- The JMS application creates a temporary queue for the current session and sets this queue as a `JMSReplyTo` queue in the message. If a message producer sends a message to a destination with a formatter and the `JMSReplyTo` property is used, the same formatter is used for messages received at this `JMSReplyTo` destination.
- The formatter gets the `JMSReplyTo` queue from the JMS message and puts the name and the type of the queue into the ACI message.
- The ACI application reads the name and the type of the queue. If the type is a temporary queue, it uses JMS/<*name of queue*>/TMPQUEUE as `CLASS/SERVER/SERVICE`. If the type is not a temporary queue, it uses JMS/<*name of queue*>/QUEUE as `CLASS/SERVER/SERVICE`.

The same applies to topics, except that the name of the JMS topic can be used "as is" for the Broker topic.

JMS Error Handling

For each JMS API interface, the methods which throw a `BrokerException` wrapped as a `JMSException` are listed.

If a JMS exception wraps a `BrokerException`, `JMSException.getErrorCode` returns the error class and error code from the Broker as *ccccnnnn*, where *cccc* is the class and *nnnn* the code. `JMSException.getMessage` returns `BrokerException.toString`. This is `Broker Error` *`cccc nnnn`*`: <detailed message>`. `JMSException.getLinkedException` returns the `BrokerException`. Using references to a `BrokerException` forces the JMS application to be compiled with the EntireX Java ACI and the application is not provider independent.

A `BrokerException` with error class 0008 (Security or Encryption errors) is thrown as `JMSSecurityException`, a subclass of `JMSException`.

A `BrokerException` with error class 0021 and error code 0043 is thrown as `InvalidDestinationException`, a subclass of `JMSException`.

A `JMSException` may wrap other JVM exceptions. Then the `JMSException.getErrorCode` returns "EntireX JMS". `JMSException.getMessage` returns `Exception.toString` for the wrapped exception, which is set as a linked exception.

JMS Class Connection

Method stop

Every `BrokerException` is thrown, except Broker error 0002 0002. The Broker returns this when the Broker user is already gone due to a timeout. This error is ignored. If a session of this connection has a message listener, this listener forwards these exceptions to the exception listener.

JMS Class Session

Method close

Every `BrokerException` is thrown, except Broker error 0002 0002. The Broker returns this when the Broker user is already gone due to a timeout. This error is ignored. If the session has a message listener, this listener forwards these exceptions to the exception listener of the connection.

Method commit

The error codes 0002 0002, 0003 0003, 0003 0005, 0010 0050, 0010 0051, and 0020 0134 are handled in this method. Every other `BrokerException` is thrown. If the session has a message listener, this listener forwards these exceptions to the exception listener of the connection.

Method createConsumer

Every `BrokerException` is thrown.

Method createProducer

Every `BrokerException` is thrown.

Method createTemporaryQueue

Every `BrokerException` is thrown.

Method createTemporaryTopic

Every `BrokerException` is thrown.

Method rollback

Every `BrokerException` is thrown. If the session has a message listener, this listener forwards these exceptions to the exception listener of the connection.

JMS Class QueueSession

Method createReceiver

Every `BrokerException` is thrown.

Method createSender

Every `BrokerException` is thrown.

Method createTemporaryQueue

Every `BrokerException` is thrown.

JMS Class MessageConsumer

Method close

Every `BrokerException` is thrown, except Broker error 0002 0002. The Broker returns this when the Broker user is already gone due to a timeout. This error is ignored.

Method receive

This method returns null, which indicates "no message received" on the error codes 0003 0003, 0010 0050, 0010 0051, and 0074 0074. The error codes 0002 0002, 0003 0005, 0020 0134, and 0074 0301 are handled in this method. All other error codes throw a `BrokerException` wrapped in a `JMSException`.

Method setMessageListener

Every `BrokerException` is thrown.

JMS Class MessageProducer

Method send

The error codes 0002 0002, 0007 0493, and 0020 0134 are handled in this method. Every other `BrokerException` is thrown.

JMS Class QueueSender

Method send

See JMS class `MessageProducer`, method `send`.

JMS Class TopicPublisher

Method publish

See class `MessageProducer`, method `send`.

JMS Class QueueBrowser

Method getEnumeration

Any error code can occur with the `BrokerException` that is thrown. The error codes 0003 0003, 0010 0050, 0010 0051, 0074 0074 while the messages are being received signal that no more messages are available. The method returns the messages received so far.

Message Listener

The methods `Connection.stop`, `Session.commit` and `Session.rollback` affect the message listener. If these methods throw exceptions, the message listener will forward those exceptions to the exception listener of the connection.

While the message listener is receiving messages, it handles errors as follows. On error codes 0002 0002 and 0020 0134 the listener retries to receive messages. Error codes 0003 0003, 0003 0005, 0074 0074, and 0074 0301 are ignored. A `JMSException` is thrown to the exception listener on error codes 0010 0050 and 0010 0051. Any other error codes is forwarded as a `JMSException` to the exception listener.