Inq and JMS

Mini Guide

Table of Contents

Document History

Version	Date	Notes
0.1	04/10/10	Initial Version

1 Introduction

This document describes how JMS functionality is available in Inq. JMS is the Java API and specification for connecting to and exchanging messages between cooperating processes via a message broker.

This document assumes the reader is familiar with Inq in general. Further information on the Inq language is available at the Inqwell website. JMS itself is described at JavaSoft. The JMS api can be found here.

2 The GlassFish[tm] Message Broker

2.1 Installing GlassFish

JMS in Inq has been tested against GlassFish[tm] message broker, known as Open Message Queue. In order to run the examples as-is you will need to install the GlassFish JMS implementation on your development machine – the latest version can be downloaded from https://mq.dev.java.net/.

The current Inq build has no JNDI support at this time, so those things that are typically retrieved from a LDAP store such as `javax.jms.ConnectionFactory` implementations and queues/topics are instead created directly. In order to support this there is a class `com.inqwell.any.jms.JMSHelper` that explicitly references the GlassFish class `com.sun.messaging.ConnectionFactory` (destinations are created in Inq script) so this class must be available at build time. The file `imq.jar`, the GlassFish runtime, must `$INQHOME/lib/endorsed` for this purpose.

Unzip the download somewhere convenient to create the installer. You will have

```
<somewhere-convenient>/openmq4_4-installer
```

although the exact directory name may have changed with more recent releases, possibly.

There is a `README` file in this directory with installation instructions. I have installed mine to `~/apps/MessageQueue44`.

2.2 Starting The Broker

Having installed GlassFish, you will have a directory structure something like this:

```
tom@think2:~/apps/MessageQueue44$ ls -l
total 28
drwxr-xr-x 2 tom tom 4096 2009-11-19 08:52 bin
drwxr-xr-x 3 tom tom 4096 2009-11-19 08:52 etc
drwxr-xr-x 4 tom tom 4096 2008-09-24 12:35 install
drwxr-xr-x 7 tom tom 4096 2010-08-31 10:03 mq
drwxr-xr-x 5 tom tom 4096 2009-11-19 08:50 nss
drwxr-xr-x 3 tom tom 4096 2009-11-19 08:52 pkg
drwxr-xr-x 4 tom tom 4096 2010-03-10 08:23 var
```

In a tty window, change directory to `~/apps/MessageQueue44/mq/bin` and invoke the following script:

```
./imqbrokerd -tty -loglevel DEBUGHIGH
```

The broker should eventually produce a 'ready' log message.

2.3 Using Other Broker Implementations

The Inq JMS functionality is implemented strictly in line with the JMS specification, so any JMS implementation should work just as well. With the present lack of JNDI you will need to provide your own equivalent of `com.inqwell.any.jms.JMSHelper` and modify the examples accordingly. This is covered in the section on running the Example Applications

3 The Inq JMS Entities

The Inq bindings for JMS follow closely the interfaces defined in `javax.jms`. The following steps are typical:

1. Obtain a `ConnectionFactory`. Until JNDI is supported, this must be achieved by calling a Java method (which may itself do so) using the `callmethod` Inq function. The examples call `com.inqwell.any.jms.JMSHelper.getJMSConnectionFactory`.
2. Using the `ConnectionFactory` and the Inq function `mqcreateconnection()`, create a `Connection` to the broker. This function takes as arguments a connection factory and optional user-name and password.
3. Using the `Connection`, create a `Session` using the Inq function `mqcreatesession()`. This function requires a `Connection` argument, a boolean specifying whether the session is transacted and, if it is not, an acknowledgement mode. A `Session` is the entity through which queues and topics (commonly known as destinations) can be explicitly created, temporary destinations acquired, producers and consumers that use destinations and queue browsers created.
4. Create a destination, that is a topic or a queue. Again, destinations may normally be looked up via JNDI however they can also be created with `mqcreatetopic()` and `mqcreatequeue()`. These functions require a `Session` and a destination name, which is a string.
5. Create a `Producer` or a `Consumer`, through which messages are sent or received, using `mqcreateproducer()` and `mqcreateconsumer()`. These functions require a `Session` and a destination. Consumers can specify a message selector – a string that filters the messages the consumer will present out of all those that are sent to its associated destination. Consumers that are associated with topics can also specify whether messages that are produced by their own connection will be delivered – the `nolocal` argument.
6. Create text, map, stream, bytes or object messages, populate them with data and send them. The functions are `mqcreate<type>message()` and `mqsend()`. There is also a message that carries no payload (other than the JMS specified and user-defined properties that all messages can carry) created by `mqcreatemessage()`.
7. Receive messages either synchronously using `mqreceive()` or asynchronously if a message listener function has been established using `mqsetmessagelistener()`. Whether messages are processed synchronously or asynchronously is a property of the `Session`.

The Inq scripting concepts that are employed are

1. Built-in functions – these all begin `mq` ;
2. Access to functionality and data using properties (accessed as map children).

3.1 Sessions and Processes

The JMS specification says that a `Session` is a single-threaded context through which messaging entities may be created and used. In Inq, a `Process` is a single-threaded execution context providing a node space

and a transaction.

A process is started by the environment (for example when a client connects to a server) or explicitly using the `spawn()` function. Inq creates a thread for the process which waits on the process's input channel. The process then responds to service requests sent to it using the `send()` function. This threading model makes it natural that a given JMS session is managed by a given Inq process.

3.1.1 Synchronous Sessions

The `mqcreatesession()` function creates a synchronous session, that is one whose messages are received by a blocking call to `mqreceive()`. This synchronous model is one of the message processing options in JMS which in the Inq world is suitable for simple stand-alone utilities, such as an alert processor that emails exceptions. However it does not fit particularly well with enterprise server applications.

While the Inq process thread, woken by a service request, is blocked in `mqreceive()` it is unable to respond to further service requests or indeed any other kind of event it may have solicited from the Inq environment. In effect, the `mqreceive()` call, perhaps inside a loop construct, has become the process's *event loop*, usurping that model provided by Inq. If that is suitable (as it may be for a very simple application and as used in some of the examples) then all well and good. Otherwise an asynchronous session is preferable.

3.1.2 Asynchronous Sessions

While an Inq process defines a thread, this thread does not have to be the only one that can use the process's context. The *plugin* environment, for example, makes no assumptions about the threading model that calls back to it and in the graphical client, the Inq process representing the user interface is shared between its own thread and that of the Java graphics system. The node space and transaction however are single-threaded and the Inq environment ensures that, where a process is shared amongst two or more threads, only one can make use of it at any one time.

In JMS, a session (or more accurately a consumer associated with a destination) is made asynchronous by establishing a message listener on the consumer. Inq provides the `mqsetmessagelistener()` function to allow received messages to be processed by the specified Inq scripted function. Any number of consumers can have message listeners set for them, however the JMS specification says that only one of these will ever run at once (where the consumers were all created from a single session).

Inq further manages the thread model by ensuring that, even if different JMS sessions are used, only one message listener function (which uses the Inq process in which it was established) will ever run at once. Therefore, if parallel processing of message destinations is required, separate Inq processes must be started for each.

The use of message listeners makes the JMS thread that calls back on them and the Inq process thread interleave. The Inq process is thus still able to respond to service requests (for example should other parts of the application wish to change its behaviour) and the concept of its *event loop* is maintained, while being spread across the processing of JMS messages and Inq events.

3.1.3 Transacted and Non-transacted Sessions

The arguments when creating a session specify:

- whether the session is transacted;
- if it is non-transacted, what its JMS acknowledgement mode is.

An acknowledgement is confirmation to the JMS implementation that the message has been delivered and

processed successfully by the receiver. JMS specifies the following acknowledgement modes

`AUTO_ACKNOWLEDGE` - acknowledgement is implicit after returning from `mqreceive()` (message not necessarily actually processed) or the listener function (message presumably processed).

`CLIENT_ACKNOWLEDGE` - explicit acknowledgement by calling `mqacknowledge()`.

`DUPS_OK_ACKNOWLEDGE` - acknowledgement is automatic but lazy (JMS Specification refers) resulting in possible duplicate delivery, which the receiver must be able to handle.

Notwithstanding the JMS Session/Inq Process pattern noted earlier, an Inq process can create any number of non-transacted sessions with any acknowledgement mode.

A process creates a transacted session as follows:

```
any mySession = mqcreatesession(myConnection, true);
```

The second argument indicates the session is transacted. In this case, the acknowledgement mode is not relevant so Inq allows it to be omitted. When a process creates a transacted session its Inq transaction context takes over the commit or rollback of the session. A process can create at most one transacted session.

JMS defines a transaction as all the messages consumed and produced between calls to `javax.jms.Session.commit()`. If message processing is unable to complete for any reason, the application calls `javax.jms.Session.rollback()`. The JMS implementation will then redeliver any messages consumed in the transaction and not send any that were produced.

There are no Inq functions that correspond to their `javax.jms.Session` counterparts for commit and rollback. Instead, by default these operations are carried out by the process's root transaction. If script uses nested transactions then it may elect to commit the JMS session at the current level as follows:

```
transaction
{
  mqsetcommit(true);
    .
    // message processing
    .
}  // Inq commits JMS transaction
```

By calling `mqsetcommit(true)` the current transaction is instructed to commit (or rollback) the process's JMS transacted session. If committed, the single message consumed, necessarily outside any nested level, and any produced will be committed to as the current transaction commits. The process may continue to produce messages within enclosing transaction(s) and these will be committed by them or by the root transaction, depending on the prevailing `mqsetcommit` value.

If the transaction fails the JMS session is rolled back. In this case the JMS implementation will endeavour to redeliver the original consumed message while not sending any that were produced.

An Inq transaction also manages any relational database resources bound to typedef instances. It commits these before committing the JMS session. There is therefore the possibility that the JMS session commit may fail while the database commit has succeeded. In this scenario the application must either be capable of processing the redelivered message without error or, by inspecting the received message's `JMSRedelivered` property, take some other appropriate action.

4 Example Applications

4.1 GlassFish Examples in Inq

The GlassFish JMS implementation comes with a number of examples and a readme file for them in `.../MessageQueue44/mq/examples/jms`. These have been reproduced in Inq in the `$INQHOME/examples/jms` directory.

In addition to these there is a queue browser example and an example of how XML can be used as the message payload.

4.1.1 SenderToQueue.inq

Sends a number of *text* messages to a named queue. There doesn't need to be anyone ready to consume these messages as they will remain in the queue (according to their time-to-live) until someone does. Finishes by sending an empty control message.

```
inq -in SenderToQueue.inq -name fooQueue
```

4.1.2 Synch/AsynchQueueExample.inq

Complementary to SenderToQueue, demonstrates queue message consumption using synchronous and asynchronous sessions. Reads all text messages until a non-text message, interpreted as the empty control message, is received.

```
inq -in SynchQueueExample.inq -name fooQueue
```

4.1.3 The Other GlassFish Reimplementations

All the other examples produce and consume messages within the single application, starting Inq processes as necessary. Comments from the original Java sources have been included and the overall structure aped as much as possible. The `…/MessageQueue44/mq/examples/jms/README` file refers.

4.1.4 XMLTopicExample.inq

This example is the same as AsynchTopicExample.inq except that it uses XML inside a text message, generating it on the sender side and parsing on the receiver side.

4.1.5 QueueBrowser.inq

Sometimes its useful to look at the messages in a queue and optionally drain them, especially if the

consuming application you are developing crashes leaving the messages in the queue in an invalid state (well you were using transactions though, right?).

```
inq -in QueueBrowser.inq -queues fooQueue [barQueue …] [-drain]
```

This will dump out messages in the specified queue(s) and consume them if the `-drain` option is specified.