

IBM MQ JMS Application Server Facilities

The JMS Specification attempts to describe a mysterious component called the Application Server Facilities. This topic is also described in the IBM Using Java manual. Neither of these provide a sufficiently clear description to make them understandable, let alone usable. To get to the bottom of the JMS Application Server Facilities, this document attempts to describe these function in plain English, explain why they were provided and, most importantly, how to use them in application projects.

The reader of this document is assumed to be sufficiently familiar with JMS to understand its basic architecture plus the basic JMS classes such as *Session*, *ConnectionFactory*, *MessageListener* and similar.

The rational for the facilities

The facilities start with a simple notion.

A common task of a messaging and queuing solution is to retrieve messages from a queue and process each message. This sounds simple enough. One could create an application that created a *QueueReceiver* and issues the *receive()* call to retrieve a message and process it. Following its processing, the activity could be performed again inside a loop.

Although simple enough, it doesn't appear to scale at all well. If there was more than one message on the queue to be processed, each would have to wait for the preceding message to be retrieved and processed prior to the retrieval and processing of the next. This effectively serializes the processing of messages.

Ideally, an application should be able to process some number of messages in parallel (concurrently). This can be achieved by creating multiple threads within the application and have each thread issue a corresponding receive. Although this would seem to solve the problem, a slew of implementation and performance considerations arise. If there are fewer messages in the queue to be processed than threads available for processing, the threads will consume resources that simply aren't needed. If there are more messages in the queue than threads, some will still have to wait. Extra logic would have to be explicitly implemented to handle such conditions.

If there are multiple applications, perhaps each processing different queues or performing different activities with messages on those queues, each application would have to explicitly include code logic to perform thread management. It would appear that there is

an opportunity to provide a single solution that can accommodate a variety of different queues and processing logic.

The model for the facilities

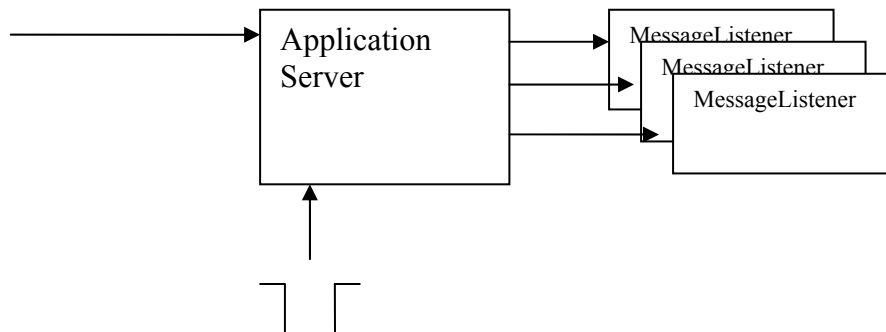
The model presented for the facilities is logically quite simple. Consider a piece of software called an *application server*. To use this application server, the following attributes could be configured to it:

- QueueConnectionFactory
- Queue
- MessageListener
- MessageSelector

When supplied these attributes, the application server will form a connection to the JMS provider (IBM MQ for example) identified by the QueueConnectionFactory, open the Queue specified by the Queue parameter and retrieve messages that match the MessageSelector specification.

For each message retrieved, the user written code identified in the MessageListener will be executed by passing the retrieved message to the *onMessage* method of the MessageListener. If there are multiple messages on the queue, the application server will be able to retrieve more than one message and pass each message to a separate thread for concurrent execution. The application server should also be able to monitor multiple queues with multiple message selectors concurrently.

- QueueConnectionFactory
- Queue
- MessageListner
- MessageSelector



If an application server, as outlined above, were actually implemented, users of such a component would only have to supply parameters and the functional code to perform the requested work on a message when it arrives. The task of connecting to queue managers, reading from queues and other low level implementation details could be hidden from them allowing them to concentrate on the business logic.

This model of an application server is similar to that of the Enterprise Java Bean (EJB) Message Driven Bean (MDB) found in the EJB 2.0 specification.

Within the JMS Specification, guidance is provided on **how** an implementation of an application server could be architected to perform its tasks efficiently. In the specification, this architecture and its supporting Java classes are called the JMS Application Server Facilities.

The remainder of this document will describe this architecture in further detail. It is important to keep in mind the overall goal of the architecture, namely to architect the solution just described.

The JMS Application Server Facilities describes an architecture and set of supporting classes used to implement the asynchronous and concurrent processing of messages arriving on one or more queues.

Facilities components

To provide the framework, the JMS specification introduces some new classes and interfaces that serve the sole purpose of supporting this framework. These classes are:

- ConnectionConsumer
- ServerSession
- ServerSessionPool

Of these, the ConnectionConsumer is implemented by the JMS Provider (IBM MQ for example) and the ServerSession and ServerSessionPool are to be implemented by the application server developer and are defined as Java interfaces.

The purpose and relationship between these classes and the remainder of the JMS classes form the architecture. There is quite a dance between all these components and simply reading the JMS specification does not convey the subtleties.

Before explaining more about these classes, we must understand some additional information about a class we thought we already knew ... the JMS Session class.

The JMS Session class has two methods defined on it that that have the only purpose of implementing the application server framework. These methods are:

- setMessageListener()
- run()

Method: Session.setMessageListener()

Normally, the *setMessageListener()* method name is associated with a JMS Receiver object. In that instance, once a Receiver has been created (and hence associated with a queue), the receiver object's *setMessageListener()* method associates a MessageListener to receive messages that arrive on the JMS queue.

When a MessageListener is associated with a Session via the *setMessageListener()* method, this defines a single MessageListener to be invoked for receipt of any message received by that session. It will be shown that a message is associated with a Session by the ConnectionConsumer (to be discussed). When a message is associated with a Session, it is **not** immediately executed on the MessageListener. Instead, it is processed when the Session's *run()* method is invoked.

Method: Session.run()

The *run()* method on the Session is used to invoke the MessageListener set by the Session's *setMessageListener()* method with a message supplied to the Session by the ConnectionConsumer.

The ConnectionConsumer

The ConnectionConsumer is a class implemented by the JMS Provider (IBM MQ). It is created with parameters which include a Queue and a MessageSelector. From this combination, suitable messages may be retrieved from the queue and passed to a MessageListener.

The ConnectionConsumer is created by invoking the *createConnectionConsumer()* method on the a QueueConnection object. Since the QueueConnection object is created from a QueueConnectionFactory, the ConnectionConsumer has knowledge of information including:

- The QueueConnectionFactory information. This contains the details of how to communicate with the underlying JMS Provider.
- The Queue object. This identifies an individual queue from which messages are to be retrieved.
- The MessageSelector. This identifies the set of messages eligible for retrieval from the queue.

What is missing in order to actually process the messages are two parts:

- The identification of the MessageListener to be invoked by passing a retrieved message to its *onMessage()* method implementation
- A Session in which to retrieve the message. This is where the ServerSession object comes in to play.

The ServerSession

Returning to the original design intent, when multiple messages arrive on a queue, each message is to be supplied to an implemented MessageListener instance for concurrent processing. In order to achieve concurrent execution, multiple threads must be involved. The ServerSession objects are used to associate a Java thread with a JMS Session. The idea here is that a Thread will be associated with the Session and the Session will be associated with a MessageListener. When the ConnectionConsumer retrieves a message from the queue, it will obtain a ServerSession object and pass the message to the JMS Session contained within that ServerSession object. Since the ServerSession also contains a thread, the ServerSession can now arrange for the MessageListener to be passed the message in parallel with other activities.

The implementation of the ServerSession object is the responsibility of the application server creator.

The ServerSession interface contains only two methods, these are:

- getSession()
- start()

Method: ServerSession:getSession()

This method returns the JMS Session that will be associated with the message retrieved by the ConnectionConsumer. This method will be called by ConnectionConsumer.

Method: ServerSession:start()

The *start()* method is called by the internals of the JMS Provider implementation. It is invoked when messages have been associated with the JMS Session managed by the ServerSession object. When called, the implementation of the *start()* method should call the associated Session's *run()* method. This will cause the delivery of the one or more messages that are ready for processing. The *run()* method will not return control to its caller until all the messages have been processed.

It can now be seen that the steps involved in processing a message are:

1. ConnectionConsumer retrieves the message from the queue
2. ConnectionConsumer obtains a ServerSession object instance
3. The ConnectionConsumer retrieves the JMS Session associated with the ServerSession by calling *getSession()*.
4. ConnectionConsumer associates the message with the retrieved Session using JMS Provider proprietary and specific internals.
5. The ConnectionConsumer calls *start()* on the ServerSession which should return immediately.
6. The ServerSession's implementation of *start()* calls the JMS Session's *run()* method on a thread associated with the Session.

At this point it should be clearer that the `ServerSession` knows how to return a JMS Session and provide a thread for the creation and execution of messages for that Session. Since the creation of both threads and Sessions is expensive, having a new `ServerSession` object created for each new message arriving on a monitored queue would be too expensive. To resolve this issue, `ServerSessions` can be pre-created and stored in a collection or pool. When a `ConnectionConsumer` needs the services of a `ServerSession`, it can then retrieve an available `ServerSession` from the pool, use it and then return it to the pool. To achieve this, a `ConnectionConsumer` must have a defined mechanism for interacting with `ServerSessions` in a pool. This is achieved with the JMS defined interface called `ServerSessionPool`.

ServerSessionPool

This interface is defined as part of the `javax.jms` package. The purpose of this object type is to create and manage a pool or collection of `ServerSession` objects. How it is implemented is of little concern to JMS. It must, however, obey the architecture framework rules. It contains a single method to be implemented. The method is called *getServerSession* and has the following signature:

```
ServerSession getServerSession()
```

The *getServerSession()* method is expected to be called by the `ConnectionConsumer` implementation. It should return a `ServerSession` object from the pool of such objects. The implementation may choose to dynamically create a new instance or reuse a previous one. If the implementation decides that there are no further `ServerSession` objects available, it should block until a `ServerSession` object becomes available. Again, the internal implementation of `ServerSessionPool` is of no concern to JMS, only that it complies with the agreed semantics. It is clear that a `ServerSession` can be retrieved from the pool with the *getServerSession()* method but it seems odd that there is no correspondingly architected method to return a `ServerSession` to the pool. This is because the intent is to have a `ServerSession` retrieved by the JMS Provider `ConnectionConsumer` object but have the `ServerSession` returned to the pool when it has finished its task. It is the implementation of the `ServerSession` that knows when the task has been completed and since both it and the `ServerSessionPool` are implemented by the application server author, there can be any association at all between them. There is nothing to prevent an implementation of `ServerSessionPool` from having a method to return a `ServerSession` to the pool, the interface only describes the **must** implement methods and says nothing about other potential methods.

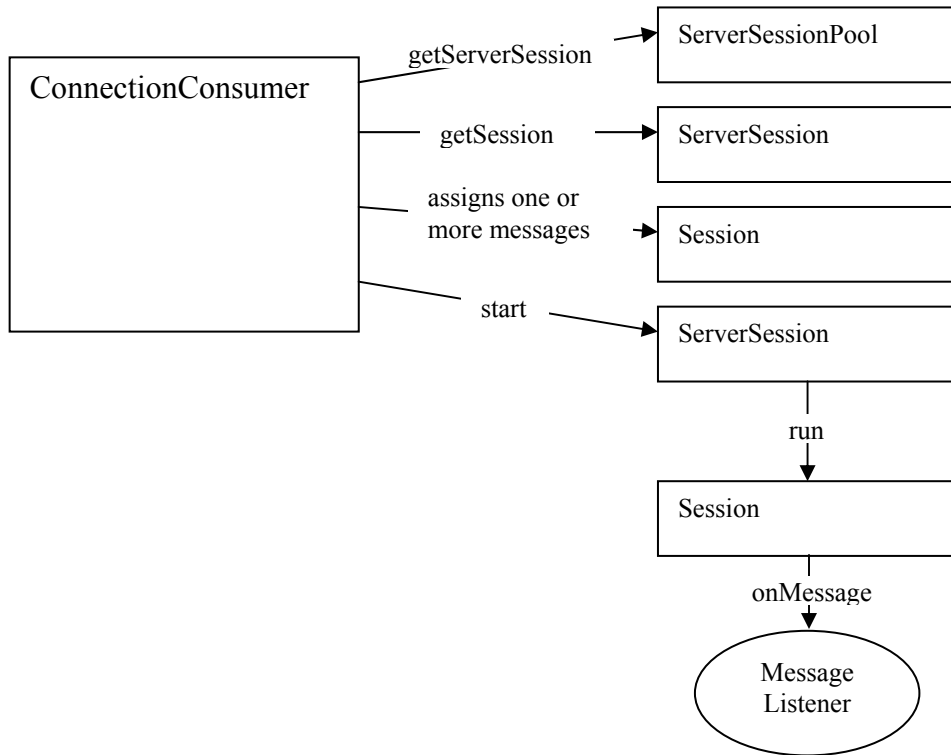
The task of a `ServerSession` is to process a message supplied to it by the `ConnectionConsumer`. The `ConnectionConsumer` informs the `ServerSession` that a message is ready by calling its *start()* method. The `ServerSession` then calls the *run()* method of its associated `Session` and, when *run()* has completed, the `ServerSession` activity has also completed. At this point, it could return itself to the pool.

Putting all this together, the architecture for processing a message now becomes:

1. ConnectionConsumer retrieves the message from the queue
2. ConnectionConsumer obtains a ServerSession object from the ServerSessionPool
3. The ConnectionConsumer retrieves the JMS Session associated with the ServerSession by calling *getSession()*.
4. ConnectionConsumer associates the message with the retrieved Session using JMS Provider internals.
5. The ConnectionConsumer calls *start()* on the ServerSession which should return immediately. The ConnectionConsumer has now passed the message onwards and can continue to process other requests.
6. The ServerSession's implementation of *start()* calls the JMS Session's *run()* method on a thread associated with the Session.
7. When *run()* returns, the ServerSession can return itself to the ServerSessionPool from which it was retrieved.

Role of the ConnectionConsumer

The ConnectionConsumer object is responsible for orchestrating the activities. The following diagram illustrates its activities:



When a message arrives in a queue being monitored by the ConnectionConsumer, it retrieves a ServerSession from the ServerSessionPool by calling the *getServerSession()* method. From the ServerSession, it retrieves the associated Session by calling *getSession()*. It then uses a JMS Provider specific mechanism to assign the message or

messages to the Session. Finally, the ConnectionConsumer calls the ServerSession's *start()* method which requests that the ServerSession invoke the processing of the messages. The ServerSession returns control immediately and runs the Session's *run()* method on a separate thread. The *run()* method calls the MessageListener's *onMessage()* method for each message that was associated with it by the ConnectionConsumer.

IBM MQ Supplied Sample applications

IBM supplies a set of sample application server framework applications. These are located in the <WebSphereMQ>/Tools/java/jms/asf directory. Contained within this directory are the following files:

MyServerSession.java	Implementation of the ServerSession interface.
MyServerSessionPool.java	Implementation of the ServerSessionPool interface.
Load1.java	A trivial JMS application used to load a queue with messages. This program has nothing specific to the application server framework and is supplied here only to places messages on a queue for testing. A trivial JMS application used to load a queue with messages. This program has nothing specific to the application server framework and is supplied here only to places messages on a queue for testing.
Load2.java	Same as Load1.java but with different message contents.

Using the Application Server Facilities

The preceding information provided the basis of the Application Server Facilities. As was seen, the two interfaces, ServerSession and ServerSessionPool are implemented by the application server designer. Assuming that these exist, we will now look at what is required in a JMS client application to utilize the services of the framework.

The key to using the Application Server Framework is the ConnectionConsumer class. An instance of this class is returned from the `javax.jms.QueueConnection.createConnectionConsumer()` method. The ConnectionConsumer is implemented by the JMS Provider (IBM MQ).

The *createConnectionConsumer()* method has the following signature:

```
public ConnectionConsumer createConnectionConsumer(
    Queue queue,
    String messageSelector,
    ServerSessionPool sessionPool,
    int maxMessages)
```

The queue parameter identifies the queue from which messages are to be retrieved. The messageSelector parameter specifies an optional qualifier for messages on the queue. A value of null will return all/any messages. The sessionPool parameter specifies an instance of a ServerSessionPool from which ServerSessions are to be retrieved for

processing the messages. The `maxMessages` parameter specifies the maximum number of messages to load into a single `Session` for processing.

Once the `ConnectionConsumer` has been created and the `QueueConnection` started with its `start()` method, messages will start to be delivered to the `MessageListener` for processing.

It should be noted that the `MessageListener` to be used is **not** specified in the `ConnectionConsumer` creation method. This seems odd. The `MessageListener` to be used appears to need to be associated with the `ServerSessionPool`. Logically, this says that the `ServerSessionPool` contains a pool of `Sessions` for processing an individual `MessageListener`. I don't personally agree with this design.

With this information, a sample application that utilizes the `Application Server Framework` can now be shown:

```
QueueConnectionFactory factory = ...;
Queue queue = ...;
...
QueueConnection connection = factory.createQueueConnection();
SampleServerSessionPool serverSessionPool = new
    SampleServerSessionPool(connection, SampleMessageListener.class);
ConnectionConsumer consumer =
    connection.createConnectionConsumer(
        queue,
        null,
        serverSessionPool,
        5);
connection.start();
Thread.currentThread.sleep(999999999);
```

Walking through this code, we assume the creation of the `QueueConnectionFactory` and `Queue` by normal JMS methods such as a JNDI lookup. The `QueueConnection` is then derived from the `QueueConnectionFactory`.

The `serverSessionPool` is an instance of the, as yet unknown, implementation of the `SampleServerSessionPool` which implements the `ServerSessionPool` interface. It is constructed with a reference to the `QueueConnection` and the `MessageListener` to be acted upon when messages arrive.

The consumer is created and then the connection started. Finally, the thread puts itself to sleep allowing messages to be processed in the background. If the thread didn't suspend, but instead it terminated, the consumer would be released and the asynchronous processing of messages would come to an end.

An implementation of the `ServerSessionPool`

Before the preceding JMS client application can run, an implementation of the `ServerSessionPool` must be obtained. This section describes such an implementation and

is heavily based on the IBM supplied sample contained in the MyServerSessionPool.java file supplied with WebSphere MQ.

The design of the ServerSessionPool class (called SampleServerSessionPool) must implement the ServerSessionPool interface. This means the implementation of the getServerSession() method.

The design intent of the SampleServerSessionPool includes the following:

- create and maintain a collection of ServerSession objects
- return an *unused* ServerSession object when requested
- allow the return of a ServerSession object back into the collection

Construction of the SampleServerSessionPool

The constructor for the SampleServerSessionPool will be passed the number of ServerSessions to maintain. It will create a private array ServerSessions of this size. In addition, since the ServerSessionPool needs to create Session's it will be passed a QueueConnection from which the Sessions may be derived. Finally, the ServerSessionPool will be passed a MessageListener.class to allow it to construct MessageListener instances as needed.

The getServerSession() method

The *getServerSession()* method is the heart of the design. It is to retrieve a ServerSession from the pool which is not in use. The implementation will scan the array of ServerSessions and query each one to determine if it is currently in use. The first not in use ServerSession found will be returned and that ServerSession flagged as in use.

For the case where a ServerSession is requested and none are available, the method implementation must block until one does become available.

SampleServerSessionPool source

```
import javax.jms.*;

public class SampleServerSessionPool implements ServerSessionPool {
    private int capacity; // The capacity of this pool
    private SampleServerSession[] serverSession; // Storage for this pool
    private boolean quit; // 'Stop pool' flag

    //////////////////////////////////////
    // SampleServerSessionPool
    //////////////////////////////////////
    public SampleServerSessionPool(
        Connection connection,
        int capacity,
        Class messageListenerClass)
        throws JMSEException {
        // Allocate storage for the pool
        this.capacity = capacity;
        serverSession = new SampleServerSession[capacity];

        try {
            for (int i = 0; i < capacity; i++) {
```

```

        Session s = ((QueueConnection) connection).createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
        MessageListener ml = (MessageListener)messageListenerClass.newInstance();
        s.setMessageListener(ml);
        serverSession[i] = new SampleServerSession(this, s);
    }

    } catch (Exception e) {
        // Destroy any successfully-created sessions
        for (int i = 0; i < capacity; i++) {
            if (serverSession[i] != null)
                serverSession[i].close();
        }
        throw (new JMSEException(e.toString()));
    }
}

////////////////////////////////////
// getServerSession
////////////////////////////////////
public ServerSession getServerSession() throws JMSEException {
    SampleServerSession chosenServerSession = null;

    synchronized (this) {
        // All the time we are not being told to quit and have not obtained
        // an unused ServerSession, perform the following loop
        while (true) {
            for (int i = 0; i < capacity && chosenServerSession == null; i++) {
                // If we find an unused ServerSession, select it
                if (!serverSession[i].isInUse())
                    chosenServerSession = serverSession[i];
            }

            if (chosenServerSession == null) {
                try {
                    // We haven't been able to find an available ServerSession,
                    // so wait until we are notified either (a) to quit, or
                    // (b) that a ServerSession has returned to the pool
                    wait();
                } catch (InterruptedException ie) {
                    // Ignore
                }
            } else {
                chosenServerSession.setInUse();
                return (chosenServerSession);
            }
            if (quit) {
                throw new JMSEException("MyServerSessionPool is shutting down");
            }
        }
    }
}

////////////////////////////////////
// serverSessionFinished
////////////////////////////////////
void serverSessionFinished() {
    synchronized (this) {
        notifyAll();
    }
}

////////////////////////////////////
// close
////////////////////////////////////
public void close() {
    synchronized (this) {
        quit = true;
        notifyAll();
    }

    // Remember to close down all the ServerSessions in the pool
    for (int i = 0; i < capacity; i++) {
        serverSession[i].close();
    }
}
}
}

```

SampleServerSession source

```
import javax.jms.*;

class SampleServerSession implements ServerSession, Runnable {

    private Thread thread; // The thread and the JMS Session to
    private Session session; // associate with each other
    private SampleServerSessionPool ownerPool;
    // The pool this ServerSession belongs to
    private boolean inUse; // Flags in use of this Server Session
    private boolean ready = false; // Flags holding the state of this
    private boolean quit = false; // ServerSession

    ///////////////////////////////////////////////////////////////////
    // SampleServerSession
    ///////////////////////////////////////////////////////////////////
    SampleServerSession(SampleServerSessionPool ownerPool, Session session) {
        this.ownerPool = ownerPool;
        this.inUse = false;
        this.session = session;

        // Create the thread based on the run method of this class, and start it
        thread = new Thread(this);
        thread.start();
    }

    ///////////////////////////////////////////////////////////////////
    // getSession
    ///////////////////////////////////////////////////////////////////
    public Session getSession() {
        return session;
    }

    ///////////////////////////////////////////////////////////////////
    // start
    // This method will be invoked by the ConnectionConsumer to inform the
    // Session that it is to process the messages that have been associated
    // with it. The implementation sets the ready flag to true and posts a
    // signal that will awake the thread if sleeping.
    ///////////////////////////////////////////////////////////////////
    public void start() {
        synchronized (this) {
            ready = true;
            notify();
        }
    }

    ///////////////////////////////////////////////////////////////////
    // close
    ///////////////////////////////////////////////////////////////////
    public void close() {
        synchronized (this) {
            quit = true;
            notify();
        }

        try {
            session.close();
        } catch (JMSEException e) {
            // ...
        }
    }

    ///////////////////////////////////////////////////////////////////
    // run
    // This method is executed on the thread associated with the Session.
    // The thread is started as soon as it is created. The method immediately
    // puts itself to sleep waiting for a signal. When awoken, it will check
    // that the Session is exiting or is ready to run. If neither are true,
    // it was a false start and will suspend again.
    // If it is ready to run, it will invoke the Session's run() method to
    // process the messages that have been associated with it.
    ///////////////////////////////////////////////////////////////////
    public void run() {
        while (true) {
            synchronized (this) {
                while (!ready && !quit) {
                    try {
                        wait();
                    }
                }
            }
        }
    }
}
```

```

        } catch (InterruptedException ie) {
            // Ignore
        }
    }
    ready = false;
}

if (quit) {
    return;
}
session.run();
inUse = false;
ownerPool.serverSessionFinished();
}
}

////////////////////////////////////
// isInUse
////////////////////////////////////
public boolean isInUse() {
    return inUse;
}

////////////////////////////////////
// setInUse
////////////////////////////////////
public void setInUse() {
    inUse = true;
}
}
}

```

The MessageListener

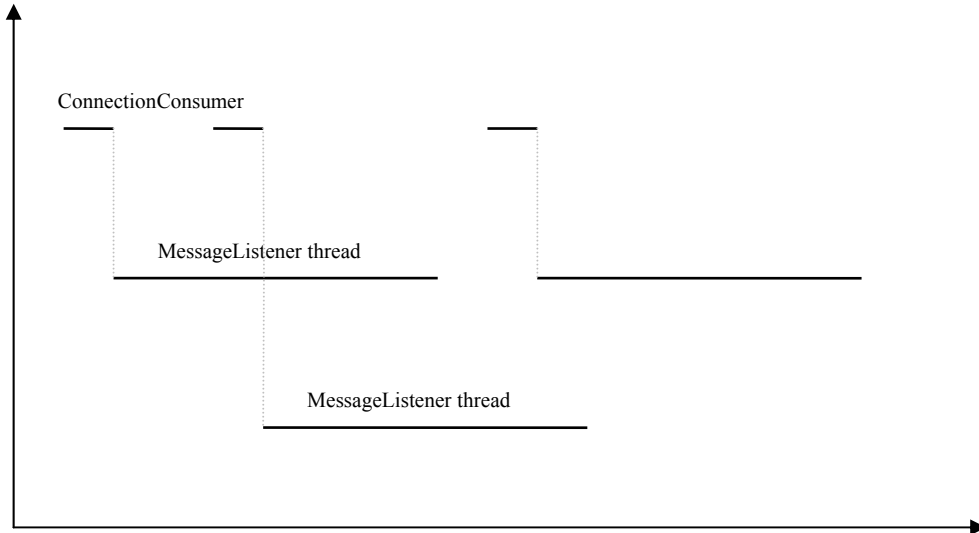
This interface requires only a single method to be implemented. Its signature is:

```
public void onMessage(javax.jms.Message message)
```

Its purpose is to asynchronously accept the delivery of a message that has been received in a queue. The implementer of this interface will place business logic in the body of the method to appropriately process the received message.

In the JMS Application Server Facilities model, the ConnectionConsumer will retrieve one or more messages from its watched queue and deliver these messages to one or more MessageListener implementing classes concurrently. To achieve this concurrency, separate threads are used with each thread having ownership of a MessageListener.

In the following diagram, time is represented along the X axis and parallel thread processing along the Y axis.



When the ConnectionConsumer retrieves a message from the queue, it passes the message to a MessageListener thread for processing. It then starts watching the queue for the next message. This next message can arrive before the preceding message processing has completed. In this case, the new message will be supplied to a parallel thread for executing an instance of the MessageListener.

From this discussion, it can be seen that MessageListener's can execute in parallel. This means that either a single MessageListener instance can be used but **must** be thread safe or else multiple MessageListeners should be used, each independent from the other.

Since the ServerSessionPool owns the threads and the ServerSessions, in the case where multiple MessageListeners should be created, it should also be able to create MessageListeners when needed. The ServerSessionPool could be configured with the java.lang.Class of the MessageListener and it could then invoke the newInstance() method to create an instance as desired.