E

Working with JMS 2.0

Java Message Service (JMS) Application Programming Interface (API) is provided by Sun Microsystems and its partner companies to allow Java programs to interact with other messaging implementations. Messaging implementations are applications used for sending and receiving messages over a network. JMS API allows you to establish reliable, asynchronous, and loosely coupled communication among the components of a distributed Java EE application. With the help of this communication, the components can create, read, send, and receive messages by using the JMS API. JMS API contains a set of interfaces, classes, and related semantics that enable components to interact with other messaging systems or implementations.

The javax.jms package provides the classes and interfaces of JMS API. These classes and interfaces are implemented by a JMS provider, which is used to manage the session beans and queues in the Java EE application. Some examples of JMS providers are JBoss Messaging, OpenJMS, and Apache ActiveMQ. It is easier for the programmer to create the JMS application because JMS API is easier to learn and provides sufficient features to communicate with other complicated messaging implementations.

In this appendix, you learn about the need for JMS API; features of JMS API that make it a popular choice to create messaging applications; communication types supported by JMS API; classes, interfaces, and exceptions in JMS API; and JMS API programming model.

Need for JMS API

In a messaging system, a messaging client can deliver and accept messages from any other client. Each client interacts with a messaging agent for creating, sending, receiving, and reading messages. Messaging API, such as JMS API, is needed in contrast to other messaging API, such as remote procedure call (RPC), in the following conditions:

- When a component provider does not need to rely on other component's interface information for the replacement of components of an application
- When application provider wants the application to be executed in all situations, irrespective of whether or not all components of the application are being executed simultaneously
- When the application business model permits a component to deliver the information to another component and continue execution of the messaging service without receiving an immediate response from the receiver

Features of JMS API

With the introduction of JMS API, various vendors have adopted and provided implementation to the JMS API, which can be integrated with the application server using the Java EE Connector architecture (JCA), and accessed using a resource adapter. As a result, JMS API can now offer a complete messaging service for an enterprise. The JMS API in the Java EE platform possesses the following features:

- ❑ Enables the application clients, the Enterprise JavaBeans (EJB) components, and the Web components to deliver or synchronously receive a JMS message. Application clients may receive the JMS messages asynchronously. It is not required by the applets to support the JMS API.
- ❑ Provides the asynchronous consumption of messages by the Message-Driven Bean (MDB), which is a type of enterprise Java Bean.
- ❑ Permits JMS operations and database access operations to occur within a single transaction, because sending and receiving operations can occur in distributed transactions.
- ❑ Supports distributed transactions and permitting the concurrent consumption of messages.

Communication Types Supported by JMS API

As already learned, JMS is a service used to send and receive message through a communication process. JMS API permits the following types of communications:

- ❑ **Asynchronous communication**—Allows JMS provider to send the message to a recipient, as soon as the message arrives. The recipient does not need to make specific requests to receive the messages.
- ❑ **Reliable communication**—Allows JMS API to guarantee that a message is sent exactly once. Lower reliability is meant for applications that can afford to miss messages or accept duplicate messages.
- ❑ **Loosely coupled communication**—Allows a message to be sent to a location by a component; the recipient can then receive the sent message from that location. It is not necessary that for a successful communication, the sender and receiver of the message should be available at the same time. In other words, the sender is not required to have knowledge about the receiver's interface, and vice versa; the only information that a sender or receiver should know is the message format and the location of the message.

Now, let's discuss about the JMS API in detail.

Exploring JMS API

JMS API defines the javax.jms package that contains the classes, interfaces, and exceptions required to create the JMS client application. These JMS client applications are responsible for creating and consuming messages. To understand how an application creates or consumes messages, you need to explore the JMS API.

Let's discuss classes, interfaces, and exceptions in JMS API in detail.

Classes, Interfaces, and Exception in JMS API

The javax.jms package provides a complete set of classes and interfaces, which are used to develop a JMS client application. Some of the important interfaces and classes in the javax.jms package are:

- ❑ The BytesMessage interface
- ❑ The Connection interface
- ❑ The ConnectionFactory interface
- ❑ The MessageConsumer interface
- ❑ The MessageProducer interface
- ❑ The Queue interface
- ❑ The QueueBrowser interface
- ❑ The QueueReceiver interface
- ❑ The QueueSender interface
- ❑ The TopicConnectionFactory interface
- ❑ The XASession interface
- ❑ The QueueRequestor class
- ❑ The TopicRequestor class

Let's discuss these interfaces and classes in detail next.

The BytesMessage Interface

The BytesMessage interface extends the Message interface and sends a message to its receiving point. The message contains a stream of uninterpreted bytes, which are interpreted by the message receiver. The JMS API allows you to use message properties with byte messages; however, the usage of these properties may affect the message format. Therefore, it is advisable not to use the message properties along with the byte message. The methods of the BytesMessage interface depend on the methods of the java.io.DataInputStream class and java.io.DataOutputStream The important methods of the BytesMessage interface are described in Table E.1:

Table E.1: Explaining the Important Methods of the BytesMessage Interface

Methods	Explanation
getBodyLength()	Returns the number of bytes in a message body during its read only mode.
readBoolean()	Reads a boolean value from the message stream containing bytes.
readChar()	Reads a Unicode character value from a message.
readInt()	Reads a signed 32-bit integer from a message.
readLong()	Reads a signed 64-bit integer from a message.
readUTF()	Reads a modified UTF-8 format encoded string from a message.
readBytes(byte[] bytevalue, int bytelength)	Reads a part of a message. If the value of the bytevalue argument is less than the number of remaining bytes to be read, the byte array is filled with bytes from the message. Moreover, the next invocation of the method reads the next increment and the process continues till the specified number of bytes is read.
writeBoolean(boolean myvalue)	Writes a boolean to a message in the form of 1-byte value.
writeByte(byte myvalue)	Writes a byte to a message in the form of 1-byte value.
writeShort(short myvalue)	Writes a short to a message in the form of 1-byte value.
writeChar(char myvalue)	Writes a char to a message in the form of 2-byte value.
writeInt(int myvalue)	Writes an int to a message in the form of four bytes.
writeLong(long myvalue)	Allows writing a long to a message in the form of eight bytes.
writeFloat(float myvalue)	Changes the float argument to an int with the help of Float class floatToIntBits method and then writing the converted int value to a message as a 4-byte quantity.
writeUTF(String value)	Writes a string to a message using UTF-8 encoding.

The Connection Interface

The Connection interface represents a client's active connection with the JMS provider. Provider resources are allocated externally to the Java virtual machine (JVM) by the instance of the Connection interface. In the JMS application, the instance of the Connection interface performs the following tasks:

- Encapsulates an open client connection with a JMS provider. The instance of the Connection interface is usually an open TCP/IP socket from the client to the service provider.
- Defines a distinctive client identifier.
- Supplies the ConnectionMetaData object.
- Provides support for the ExceptionListener object.

A connection instance is a heavyweight instance because its creation involves establishing authentication and communication. Single connection is utilized by majority of clients for all their messaging tasks. The important methods of the Connection interface are described in Table E.2:

Table E.2: Describing the Important Methods of the Connection Interface

Methods	Explanation
close()	Closes the connection.
createSession(boolean transactedvalue, int acknowledgeModevalue)	Creates a Session object. The transactedvalue parameter depicts whether the session is transacted or not. The

Table E.2: Describing the Important Methods of the Connection Interface

Methods	Explanation
	acknowledgeModevalue parameter signifies if the consumer or the client would acknowledge any message on reception of the message. The valid values of the acknowledgeModevalue parameter are: Session.AUTO_ACKNOWLEDGE Session.CLIENT_ACKNOWLEDGE Session.DUPS_OK_ACKNOWLEDGE
getClientID()	Returns the value of client identifier for the connection.
setClientID(String uniqueclientID)	Establishes the value of client identifier for the connection.
getMetaData()	Returns the connection's metadata.
getExceptionListener()	Returns the exception listener object related with the connection. It is not necessary for each connection instance to possess an exception listener associated with it.
setExceptionListener(Exception Listener mylistener)	Establishes an exception listener for the connection. If the JMS provider identifies a problem with the connection, it notifies the registered exception listener of the connection regarding the problem.

The ConnectionFactory Interface

The ConnectionFactory interface encapsulates the set of connection configuration parameters, which are configured by an administrator. A client uses the instance of the ConnectionFactory interface to establish a connection with a JMS provider. The ConnectionFactory instance is an administered object that provides support for concurrent use. Administered objects help administer the JMS API in an enterprise. The JMS client accesses administered objects by looking up the administered objects in a JNDI namespace. Looking up of administered objects by the JMS clients in a JNDI namespace provides the following benefits:

- Conceals provider-specific information from JMS clients
- Abstracts administrative information into Java objects that can be organized and administered from a common management console in an efficient manner
- Enables JMS providers to provide one implementation of administered objects that would be executed for every client

The methods of the ConnectionFactory interface are described in Table E.3:

Table E.3: Describing the Methods of the ConnectionFactory Interface

Methods	Explanation
createConnection()	Establishes a connection with the default user identity in the stopped mode. You must explicitly call the start method of the connection object to initiate successful message delivery.
createConnection(String userNamevalue, String passwordvalue)	Establishes a connection with the user identity passed to the method, in the stopped mode.

The MessageConsumer Interface

The MessageConsumer interface allows a client to receive messages from a destination with the help of the message consumer. To create a message consumer, destination object is passed to the createConsumer() method of the session object. All message consumers inherit the MessageConsumer interface. You can use the MessageConsumer interface to create a message consumer with a message selector. The message selector specifies the criteria based on which the JMS client sends messages to the message consumer. The JMS client can receive the messages asynchronously or synchronously from the message consumer. In case of synchronous receipt, a client may request the next message from a message consumer by invoking the receive method of the message consumer.

In case of an asynchronous delivery, a client registers a message listener object with a message consumer. When messages arrive at the message consumer, they are delivered by the message consumer, which calls the onMessage() method of the message listener.

The methods of the MessageConsumer interface are described in Table E.4:

Table E.4: Describing the Methods of the MessageConsumer Interface

Methods	Explanation
close()	Closes the message consumer
getMessageListener()	Gets the message listener related with the message consumer
setMessageListener(MessageListener listenerValue)	Establishes message listener for a message consumer
receive()	Receives the next message generated for the message consumer
receive(long timeoutvalue)	Receives the next message within the timeout interval limit, which is specified by the timeoutvalue parameter
receiveNoWait()	Receives the next message if the message is instantly available

The MessageProducer Interface

The MessageProducer interface allows the client to deliver messages to a destination. To create a message producer, a destination object is passed to the createProducer method of a session object. All message producers inherit the MessageProducer interface. You can use the MessageProducer interface to create a message producer without providing the destination information. In this case, each send() method operation is provided with the destination. You can also define a default delivery mode, priority, and time to live for messages delivered by a message producer. In addition, you can define the delivery mode, priority, and time to live in case of an individual message.

The methods of the MessageProducer interface are described in Table E.5:

Table E.5: Describing the Methods of the MessageProducer Interface

Methods	Explanation
setDisableMessageID(boolean booleanvalue)	Specifies whether message IDs are disabled or not
getDisableMessageID()	Specifies whether message IDs are disabled or not
setTimeToLive(long timeToLiveValue)	Establishes the default time, in milliseconds, for which a produced message is preserved by the message system
getTimeToLive()	Gets the default time, in milliseconds, that a produced message is preserved by the message system
close()	Closes the message producer
send(Message message, int deliveryModevalue, int priorityvalue, long timeToLivevalue)	Delivers a message to the destination, while providing delivery mode, priority, and time to live for the message

The Queue Interface

A client uses the instance of the Queue interface to provide queue identity to the JMS API. A queue may be utilized to create a MessageConsumer instance or a MessageProducer instance by calling the createProducer() or createConsumer() method of the session object and passing the queue instance as an argument. Queue name, which is provider-specific, is encapsulated by the instance of the Queue interface.

Note that JMS API does not specify the actual time for which messages are retained by a queue.

The methods of the Queue interface are described in Table E.6:

Table E.6: Describing the Methods of the Queue Interface

Methods	Explanation
getQueueName()	Gets the queue name
toString()	Gets the object's string representation

Understanding the QueueBrowser Interface

The QueueBrowser interface helps a client to access messages on a queue without removing the messages. The getEnumeration method of the QueueBrowser interface provides an enumeration, which is utilized to scan the queue's messages. The enumeration object returned by the queue browser may consist of the queue's entire content or only the messages matching a message selector. Note that when messages of a queue are scanned using enumeration, some of the queued messages may have already reached their destinations or may have expired. Queue browser can be created by utilizing a session object or a QueueSession object.

The methods of the QueueBrowser interface are described in Table E.7:

Table E.7: Describing the Methods of the QueueBrowser Interface

Methods	Explanation
getQueue()	Retrieves the queue associated with the queue browser
getMessageSelector()	Gets the message selector expression of the queue browser
getEnumeration()	Retrieves an enumeration, which is used for browsing the current queue messages in the order of reception of messages
close()	Closes the queue browser

The QueueReceiver Interface

The QueueReceiver interface receives the messages delivered to a queue. The JMS API does not specify the manner in which messages are distributed between the queue receivers when there are more than one queue receivers for a single queue. A message selector is denoted by a queue receiver and rejected messages by the message selector remain on the queue. Message selector permits a queue receiver to skip messages, which implies that when the skipped messages are eventually read, the order of the reads does not preserve the partial order provided by the message producer. A QueueReceiver object with no message selector would read messages in the same sequence in which the messages have been produced.

The QueueReceiver interface provides a getQueue() method that allows you to get the queue associated with the QueueReceiver. The following code snippet shows the syntax of the getQueue() method of the QueueReceiver interface:

```
    Queue getQueue() throws JMSException
```

The QueueSender Interface

The QueueSender interface is used by a client to deliver messages to a queue. Usually, a queue is provided when a QueueSender is created. When the send() method of the QueueSender instance is executed, the message cannot be modified by other threads within the client. In case a message is modified during the execution of the send() method of the QueueSender instance, the result of execution of the send() method is uncertain.

The methods of the QueueSender interface are described in Table E.9:

Table E.9: Describing the Methods of the QueueSender Interface

Methods	Explanation
getQueue()	Retrieves the queue related with the QueueSender.
send(Message message, int deliveryModeValue, int priorityValue, long timeToLiveValue)	Delivers a message to the queue. The message parameter signifies the message to send, the deliveryModeValue parameter signifies the delivery mode to be used, the priorityValue parameter signifies the priority for the message, and the timeToLiveValue parameter signifies the message lifetime (in milliseconds).
void send(Queue queue, Message message, int deliveryModeValue, int priorityValue, long timeToLiveValue)	Delivers a message to a queue for an unidentified message producer. The queue parameter signifies the queue object to deliver the message, The message parameter signifies the message to send, the deliveryModeValue parameter signifies the delivery mode to use, the priorityValue parameter specifies the priority for the message, and the timeToLive parameter signifies the message lifetime (in milliseconds).

The TopicConnectionFactory Interface

The TopicConnectionFactory interface is used by the JMS client to create a topic connection with the JMS provider that uses publish/subscribe messaging domain. The TopicConnectionFactory interface generates the topic connection object, using which specific topic-related objects can be created. Table E.10 explains the methods of the TopicConnectionFactory interface:

Table E.10: Describing the Methods of the TopicConnectionFactory Interface

Methods	Explanation
createTopicConnection()	Generates a topic connection in stopped mode, possessing the default user identity. It is necessary to explicitly invoke the start() method of the connection instance to initiate successful message delivery.
createTopicConnection(String userName, String password)	Generates a topic connection in stopped mode, with the user identity passed to the method. It is necessary to explicitly invoke the start() method of the connection object to initiate successful message delivery.

The XASession Interface

The XASession interface provides access to a JMS provider's support for the Java Transaction API (JTA), which is in the form of a javax.transaction.xa.XAResource object. The functionality of the XASession object resembles that of the standard X/Open XA Resource interface. An application server accesses the XA resource of an instance of the XASession interface to control the transactional assignment of a XASession. XAResource supplies some complicated facilities for interleaving multiple transactions, retrieving a list of transactions in progress, and many others. Table E.11 describes the methods of the XASession interface:

Table E.11: Describing the Methods of XASession Interface

Methods	Explanation
getSession()	Gets the session related with the XASession
getXAResource()	Retrieves an XA resource
getTransacted()	Signifies whether the session is in transacted mode or not

Understanding the QueueRequestor Class

The QueueRequestor class generates a TemporaryQueue object, which is a queue generated by the system during connection. The object of the QueueRequestor class provides a request() method that delivers the request message and waits for the reply. The QueueRequestor object is constructed by passing a non-transacted QueueSession object and a destination queue object as parameter to its constructor.

The following code snippet shows the constructor of the QueueRequestor class:

```
    QueueRequestor(QueueSession sessionValue, Queue queueValue) throws JMSException
```

The implementation of the constructor in the preceding code snippet presumes the sessionValue parameter to be non-transacted and the delivery mode of either AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE. The queueValue parameter is the queue object.

Table E.12 describes the methods of the QueueRequester Class:

Table E.12: Explaining the Methods of the QueueRequester Class

Methods	Explanation
request (Message messageValue)	Delivers a request. This method waits for the reply.
close()	Closes the QueueRequestor object and its session.

The TopicRequestor Class

The TopicRequestor helper class makes the work of service requests easier. The constructor of the TopicRequestor class is provided with a non-transacted TopicSession object and a topic. An object of the TopicRequestor class generates a TemporaryTopic object, which is a topic created by the system during connection. This object provides a request() method that delivers the request message and waits for the reply. The following code snippet shows the constructor of the TopicRequestor class:

```
    TopicRequestor(TopicSession sessionValue, Topic topic) throws JMSException
```

This implementation presumes the sessionValue parameter to be non-transacted and the delivery mode to be either AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE. The sessionValue parameter passed to the constructor represents the TopicSession object, to which the topic belongs, and topic parameter represents the topic on which the request/reply calls are performed.

Table E.13 describes the methods of the TopicRequestor class:

Table E.13: Describing the Methods of the TopicRequestor Class

Methods	Explanation
close()	Closes the TopicRequestor and its session. The close() method also closes the TopicSession object, which is passed as a parameter to the constructor of the TopicRequestor class.
request(Message messageValue)	Delivers a request and waits for the reply.

Let's learn about messaging domains in JMS.

Understanding Messaging Domains in JMS

A majority of messaging products support either the point-to-point or the publish/subscribe approach for messaging. JMS supports both types of messaging approach. The JMS specification ensures that a separate domain is provided for both the messaging domains, and defines compliance for both the domain. Implementation of either one or both messaging domains can be provided by a stand-alone JMS provider. A Java EE provider implements both messaging domains.

A majority of JMS API implementations support both types of messaging domains. Few JMS client applications utilize both type of messaging domains in a single application. Therefore, it can be interpreted that JMS API has improved the power and flexibility of messaging products.

Let's discuss both types of messaging domains.

The Point-to-Point Messaging Domain

In the point-to-point (PTP) messaging domain, a PTP application is based on the message queues, senders, and receivers. Every message is addressed to a particular queue, and JMS clients receive messages from the specific queue, which is set up to store their messages. All the messages delivered to queues are preserved in queues till the messages are consumed or expired. The main characteristics of PTP are:

- Provides only one consumer for every message.
- Enables the receiver of the message to receive the message, irrespective of whether the client application at the receiver is being executed or not. Message sender and message receiver, both do not possess timing dependencies.
- Enables the receiver to acknowledge successful message processing.

Figure E.2 shows the PTP messaging domain:

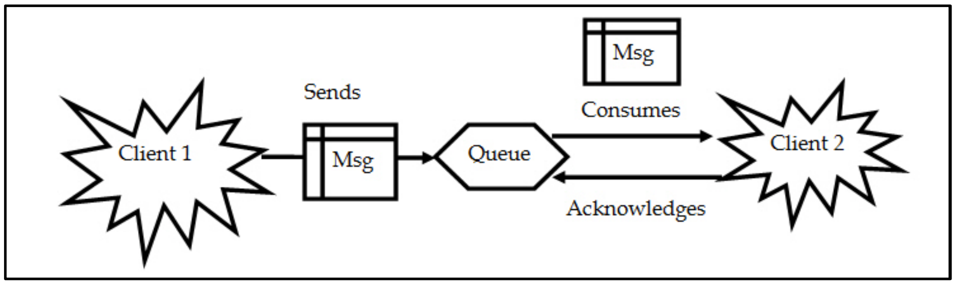

Figure E.2: Showing the Point-to-Point Messaging Domain.

In Figure E.2, Client 1 sends a message to the queue, which is consumed by Client 2.

The Publish/Subscribe Messaging Domain

Publish/subscribe (pub/sub) applications consist of a client that addresses the messages to a topic, which behaves similar to a bulletin board. Publishers and subscribers are usually anonymous and can publish or subscribe the hierarchical content dynamically. Distribution of the messages coming from a topic's multiple publishers to that topic's multiple subscribers is performed by JMS. A topic preserves the messages till the messages are distributed to the current subscribers.

Publish/subscribe messaging domain provides various consumers for every message. Publish/subscribe massaging domain maintains timing dependency between the publishers and the subscribers. A client, which has subscribed to a topic, can receive only the messages from a topic that are published after the client has generated a subscription. The application at a subscriber's end must be executed to consume messages.

JMS API provides solution to the problem of timing dependency in the publish/subscribe messaging domain by permitting subscribers to generate durable subscriptions. Subscribers generating durable subscription can receive the messages even when they are not active. Clients may send messages to multiple recipients using durable subscription. Figure E.3 shows the publish/subscribe messaging domain:

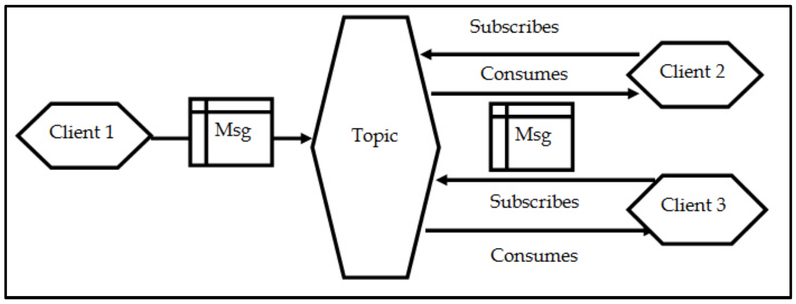

Figure E.3: Showing the Publish/Subscribe Messaging Domain

In Figure E.3, Client 1 publishes a message to the topic, which is consumed by Client 2 and Client 3.

Let's learn about message consumption in JMS.

Understanding Message Consumption in JMS

In general situations, no timing dependency exists between message production and message consumption. JMS specification supports message consumption in the following two ways:

- **Synchronous Consumption**—Enables explicit fetching of the message from the destination by the receiver or the subscriber by invoking the receive method.
- **Asynchronous Consumption**—Enables the JMS client to register a message listener (similar to event listener) with a consumer. When a message arrives at the destination, the JMS provider delivers the message by invoking the message listener's onMessage method. The onMessage method of the message listener processes the message content.

  Now, let's explore the JMS API programming model in detail.

Understanding JMS API Programming Model

To understand the JMS API programming model, you must be acquainted with the main components of a JMS application. The main components of a JMS application are as follows:

- Administered objects
- Connections
- Sessions
- Message producers
- Message consumers
- Message listeners

- Message selectors
- Messages

Figure E.4 shows the main components of the JMS API programming model:

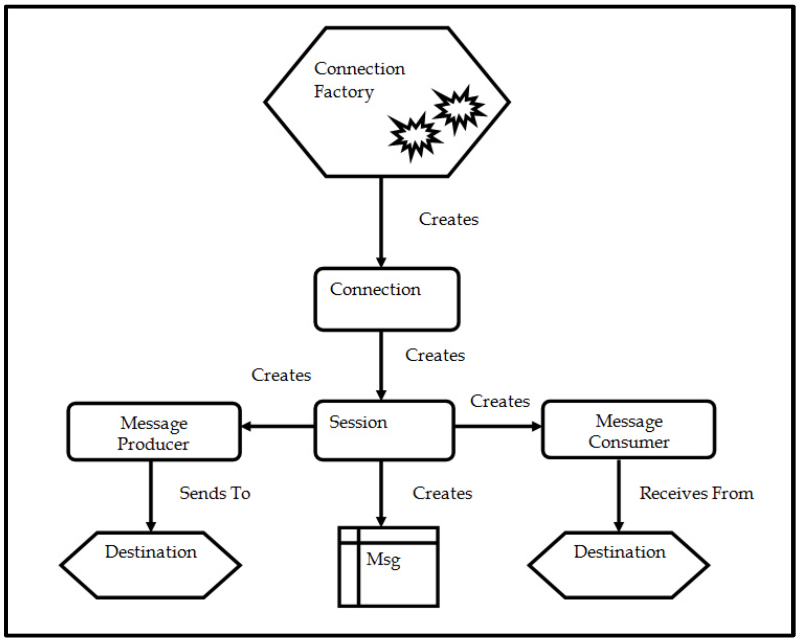

Figure E.4: Showing the JMS API Programming Model

In Figure E.4, a ConnectionFactory instance creates connection, which in turn creates a session. A session creates message, message producer, and message consumer. The message producer then sends the message to the destination using the send() method, and the message consumer receives the message from the destination using it's receive method.

Let's discuss about the various components of the JMS application in detail.

Understanding Administered Objects

JMS Administered Objects are the configurable resources introduced by a JMS Provider that are to be consumed by a JMS client. An administered object enables JMS clients to be executed on multiple JMS API implementations. Administered objects are configured in a JNDI namespace by an administrator. A JMS client can access the administered objects using resource injection. There are two types of administered objects, which are as follows:

- **Connection factory**—Enables the JMS client to create a connection to a JMS provider. Connection configuration parameters, defined by an administrator, are encapsulated by a connection factory instance. Connection factory object is an instance of any of the ConnectionFactory, QueueConnectionFactory, or TopicConnectionFactory interface.
- **Destination**—Enables JMS client to define the targets of messages produced and the source of messages received. Destinations are known as queues in PTP messaging domain and topics in pub/sub messaging domain.

Understanding Connections

A connection object represents a virtual connection of a client with a JMS provider. Session objects can be created using the connection object. Connection object implements the Connection interface. The following code snippet shows the use of the connectionFactory object to create a connection object:

```
    Connection con = connectionFactory.createConnection();
```

Any connection created in an application must be closed before the execution of the application is complete. In case you do not close a connection, resources may not be released by the JMS provider. When a connection is closed, all the associated sessions, message producers and message consumers are also closed. The following code snippet shows how to close a connection:

```
    con.close();
```

Understanding Sessions

A session object, which is a single-threaded context, is used particularly for creating and consuming messages. Session objects are used for creating message producers, message consumers, messages, queue browsers, temporary queues, and topics.

A session object implements the Session interface and can be created with the help of connection object. The following code snippet shows the creation of the session object by using the createSession method of the connection object:

```
    Session sess = connection.createSession(false,
         Session.AUTO_ACKNOWLEDGE);
```

In the preceding code snippet, the first argument passed to the createSession method is the Boolean value, false, which indicates that the session is not transacted. The second argument passed to the createSession method indicates that the session implicitly acknowledges messages after receiving them.

Understanding Message Producers

A message producer object implements the MessageProducer interface and is created using the session object. Message producers are used particularly for sending messages to a destination. The following code snippet demonstrates the creation of a message producer for a destination object, a queue object, or a topic object:

```
    MessageProducer proddest = session.createProducer(dest);
    MessageProducer prodqueue = session.createProducer(queue);
    MessageProducer prodtopic = session.createProducer(topic);
```

In the preceding code snippet, the proddest message producer object is created by passing the destination object, dest, to the createProducer method of the session object. The prodqueue message producer object is created by passing the queue object, queue, to the createProducer method of the session object. The prodtopic message producer object is created by passing the topic object, topic, to the createProducer method of the session object.

Understanding Message Consumers

A message consumer object implements the MessageConsumer interface and is created using a session object. Message consumers are utilized for receiving messages, which are delivered to a destination. To register a JMS client's interest in a destination with a JMS provider, a message consumer is used. Message sending from a destination to the registered destination's consumer is managed by JMS provider. The following code snippet shows the creation of a message consumer using a destination object, a queue object, or a topic object:

```
    MessageConsumer condest = session.createConsumer(dest);
    MessageConsumer conqueue = session.createConsumer(queue);
    MessageConsumer contopic = session.createConsumer(topic);
```

In the preceding code snippet, the condest message consumer object is created by passing the destination object, dest, to the createConsumer method of the session object. The conqueue message consumer object is created by passing the queue object, queue, to the createConsumer method of the session object. The contopic message consumer object is created by passing the topic object, topic, to createConsumer method of session object.

A message consumer object becomes active on creation and can be used to receive messages. The close method of a message consumer object makes the message consumer inactive. To start the message delivery, the connection object is started by invoking its start method.

Understanding Message Listeners

A message listener object is an asynchronous event handler for messages. The message listener object implements the MessageListener interface and possesses one method, onMessage, which defines the actions to be taken on the arrival of a message.

The setMessageListener method registers a message listener with a particular message consumer. The following code snippet demonstrates the registration of a message listener:

```
    Listener myListener = new Listener();
    consumer.setMessageListener(myListener);
```

In the preceding code snippet, the myListener object of a class that implements the MessageListener interface is created and registered with the consumer consumer object by using the setMessageListener method. After a message listener is registered, message delivery can be initiated by calling the start method of the connection object.

Understanding Message Selectors

JMS API message selector is used to filter the messages that a messaging application receives. Message selector permits a message consumer to define the messages of its interest. Filtering of messages is assigned to the JMS provider by a message selector. It consists of a string that expresses an expression. The syntax of the string expression, defined by a message selector, depends on subset of the SQL92 conditional expression syntax.

Understanding Messages

JMS message objects possess a simple, basic, and highly flexible format. Highly flexible format of JMS messages permits the messages to match formats of non-JMS applications on heterogeneous platforms. The three parts of a JMS message are as follows:

- **Message Header**—Consists of predefined fields utilized by both the client and provider to recognize and route messages.
- **Message Properties**—Enables compliance of JMS messaging system with other messaging systems. Message selectors can be created using the message properties.
- **Message Bodies**—Represents the message body. There are five message body formats, known as message types, defined by JMS API, which are used to send and receive data in multiple forms.

For creating a JMS client application, you can refer the MDB application provided in *Chapter 13, Working with EJB 3.1*.

With this, we come to the end of the appendix.