Fundamentals of Java Enterprise Components

Session: 13

Java Message Service Components

Objectives

- Explain JMS API and its utility in an application
- Describe models of message communication
- Explain JMS API architecture
- Explain the JMS programming model
- Explain configuring JMS messages with options
- Use JMS API in Java EE applications

Overview of JMS

Messaging enables integrating Java application architecture components in a loosely coupled manner.

The messaging infrastructure is also known as message oriented middleware, which isolates the developer from the implementation of the messaging system.

Java EE provides the Java Message Service (JMS) API to perform the messaging tasks.

JMS API provides certain interfaces that enable programmers to develop the messaging system in applications.

JMS API and Java EE

JMS has the following features when used as part of a Java EE application:

- Application clients, Enterprise Java beans, and Web components of the application can send and receive asynchronous messages.
- JMS also allows the application clients to set a message listener which gives a notification to the client when a message is received from other components.
- The EJB container has message-driven beans which are responsible for receiving asynchronous messages.
- Multiple message-driven beans can be pooled together for concurrent processing of the messages received.
- JMS messages can be sent and received as part of Java transaction.
- A JMS provider can be integrated using Java EE connector architecture.

JMS API Architecture 1-2

A JMS API can be used to develop an independent application or a messaging module of an enterprise application.

An independent JMS application would comprise the following components:

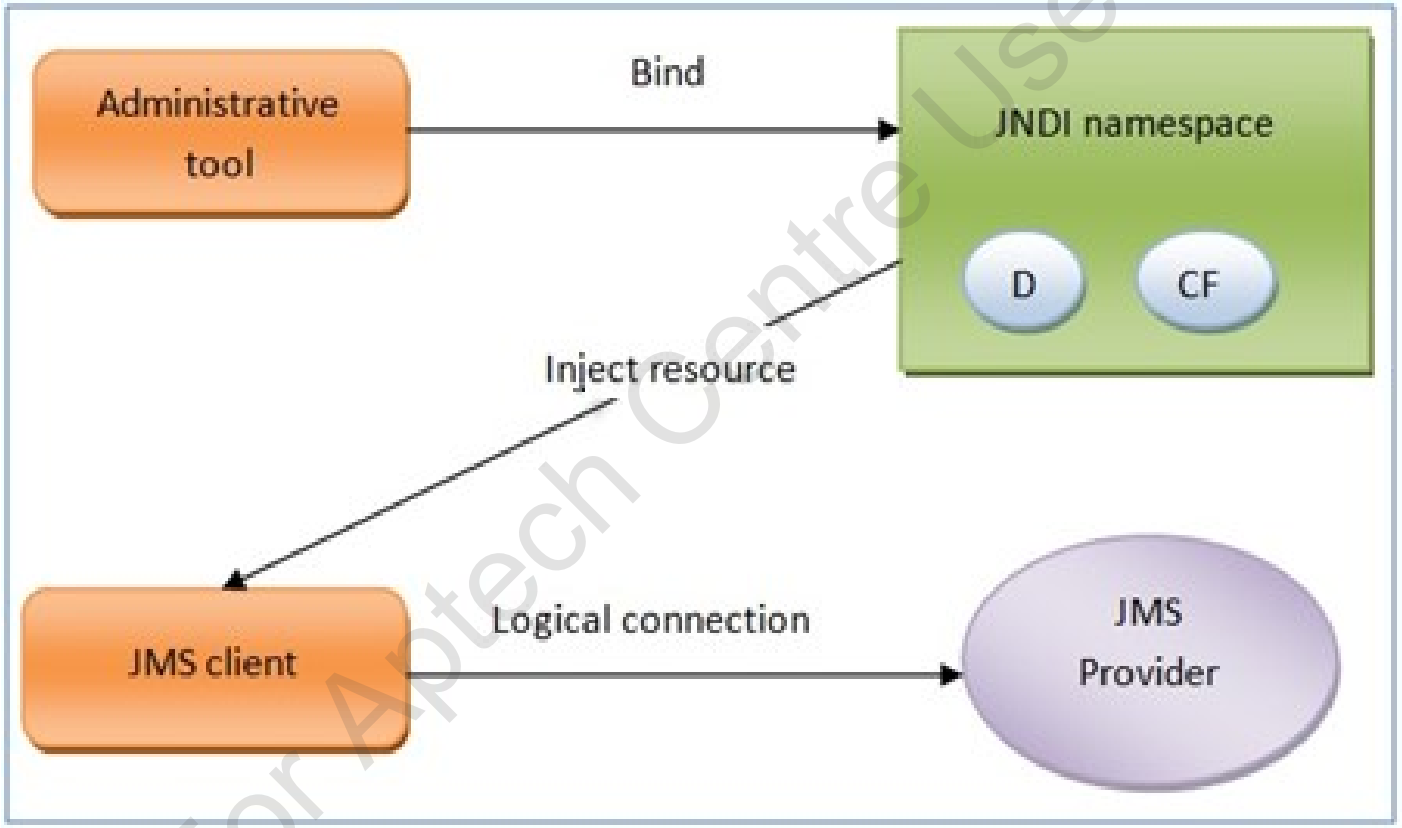
- JMS provider
- JMS clients
- Messages
- Administered objects

There are two types of configuration objects – destinations and connection factories.

The administered objects are created by the administrator and placed in the JNDI namespace and later used through their JNDI names.

JMS API Architecture 2-2

Following figure shows a diagrammatic representation of JMS architecture:



Messaging Models in JMS

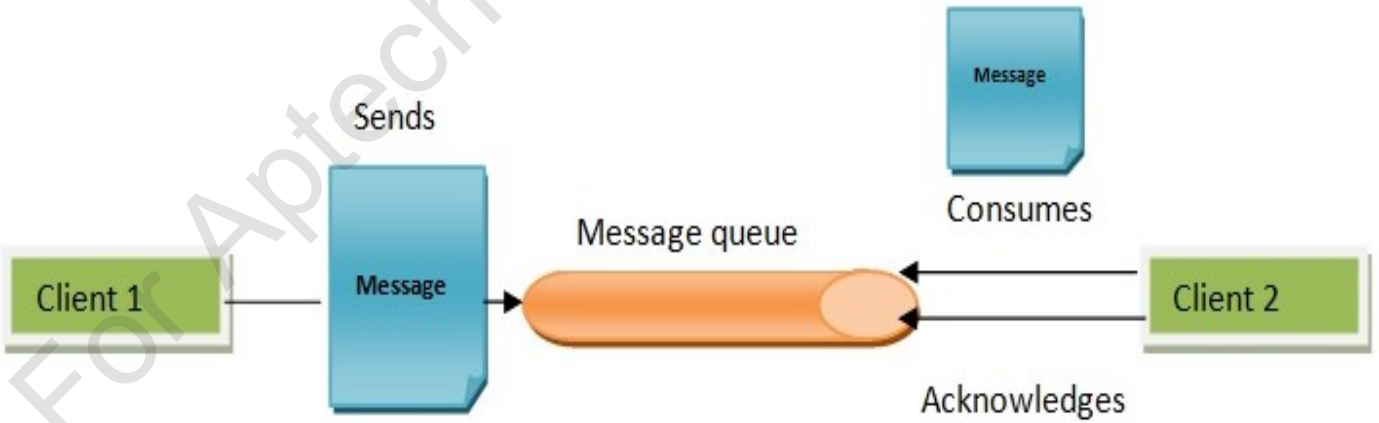
There are two models of messaging used by the JMS providers:

- Point-to-point messaging model
- Publisher-subscribe model

Point-to-point messaging model

- All JMS entities have message queues associated with them.
- Every message sent is destined to a message queue at the destination.
- Every message in this model has only one sender and one receiver.

Following figure demonstrates the point-to-point messaging model:



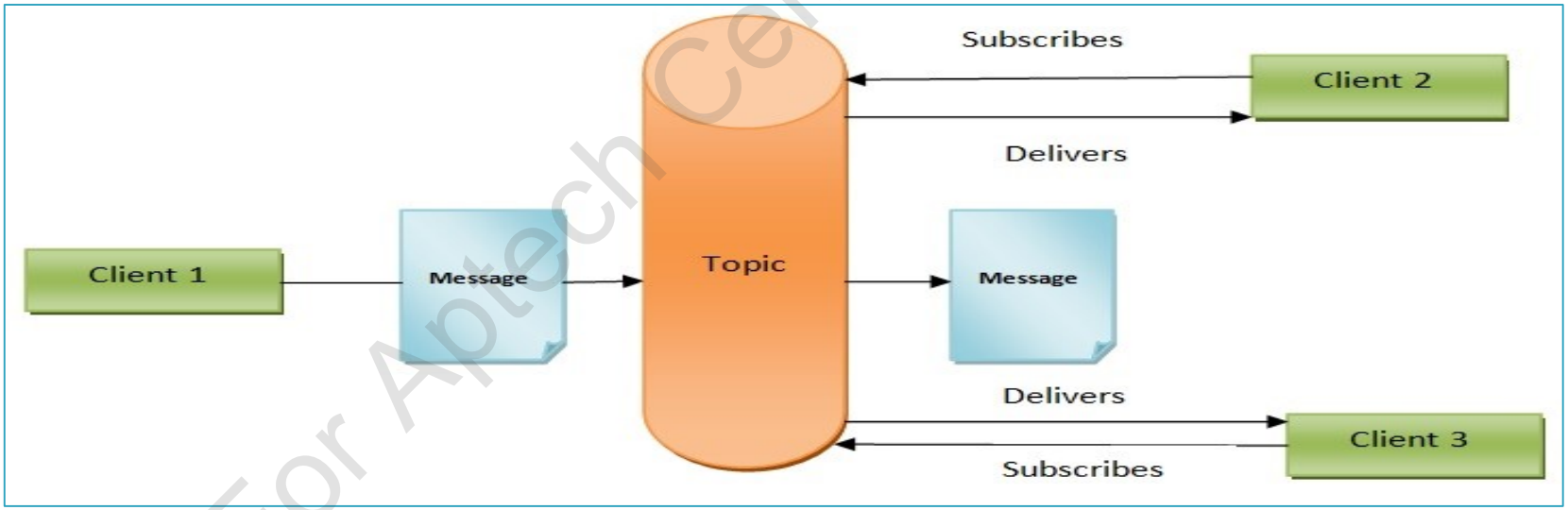
Publish-Subscribe Model

This is a broadcast communication model. All JMS clients are associated with a topic.

The JMS client which intends to receive messages from a topic has to subscribe to the topic and then consume messages.

When there are multiple subscribers to a JMS topic, then the JMS system is responsible for distributing the messages.

Following figure demonstrates publish-subscribe messaging model:



JMS API Programming Model 1-9

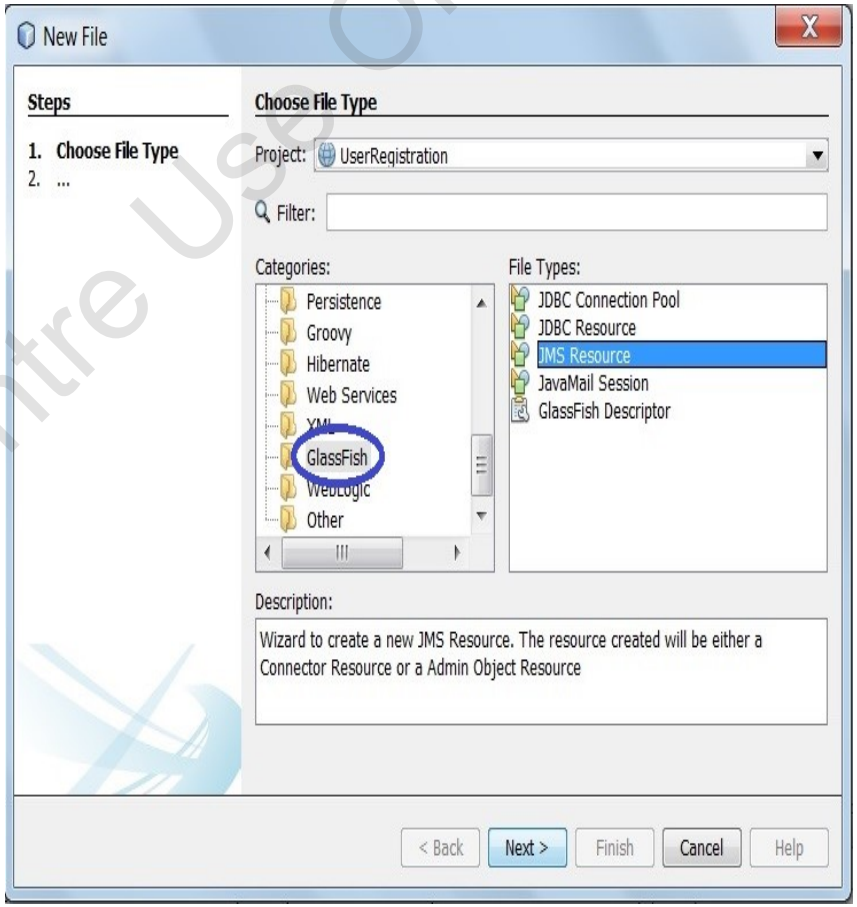
A JMS application comprises the following components:

- Administered objects
- Connections
- Sessions
- JMSContext objects
- Message producers
- Message consumers
- Messages

JMS API Programming Model 2-9

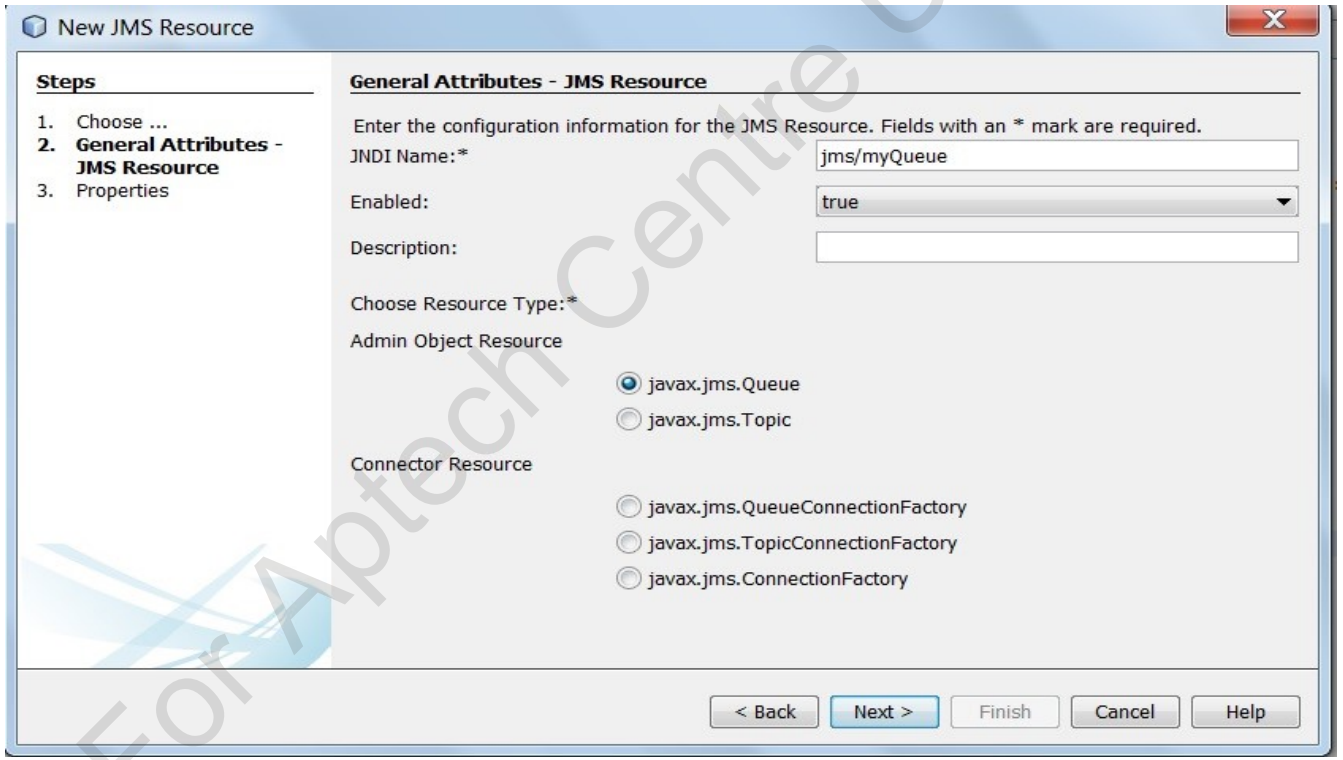
JMS Administered objects

- These are objects which are created by the administrator and are configured in the JNDI namespace of the application.
- There are two types of administered objects in JMS – Connection factories, Destinations.
- These objects are created through Administration Console.
- The figure shows the wizard which creates a JMS resource for the Java EE application.



JMS API Programming Model 3-9

An administered object can be created by adding a new JMS resource. This can be done by adding a new component to the Java EE project and clicking Next. The screen to specify JMS Resource attributes is displayed in the following figure:



JMS API Programming Model 4-9

Connections

- A connection object in the context of JMS implies a virtual connection with the JMS provider.

Sessions

- Sessions comprise message exchange among the message producers and consumers in certain application context.

JMSContext objects

- JMSContext is a state of an application which associates a connection object with a session.
- A JMSContext can be used to create message producers, message consumers, messages, queue browsers and topics.
- A JMSContext object is created by calling the createContext() method on a ConnectionFactory object.

JMS API Programming Model 5-9

Message producers are objects that generate messages in a certain context.

Message consumers are objects that accept messages from JMS clients.

- A message consumer can register with a JMS destination such as queue or topic.
- The receive() method is used to establish a synchronous communication.
- Asynchronous communication is implemented through JMS message listeners by implementing the MessageListener interface.
- The onMessage() method of the listener is invoked when a message is received.
- Following code demonstrates how a message listener is registered with a JMS consumer through the setMessageListener method:

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

JMS API Programming Model 6-9

Consuming Messages from Topics

- Messages are consumed either from queues or topics.
- Messages are consumed from topics through subscriptions.
- Subscriptions can further be durable or non-durable.
- There are two variants for both durable and non-durable subscriptions – shared subscriptions and unshared subscriptions.

JMS Messages

- JMS allows creating messages which can be sent to and received from non-JMS clients.
- A JMS message comprises the following three parts:
  - Message header
  - Properties
  - Message body

JMS API Programming Model 7-9

Message header is a set of administrative fields which identify the source and destination of the message and other administrative aspects of the message. Following table shows the header fields and their description:

Header Field	Description
JMSDestination	Set by the send method of JMS provider. It identifies the destination to which the current JMS message is sent.
JMSDeliveryMode	Set by the send method of the JMS provider. Defines whether the message should be delivered in a persistent mode or non-persistent mode.
JMSDeliveryTime	Specifies any time delay associated with the delivery of the message. This implies the tolerable time delay between the message generation and message consumption.
JMSExpiration	Specifies the validity period of the message.
JMSPriority	Defines a priority of the message set by the send method of the JMS provider.
JMSMessageID	An identifier used to identify the message which is set by the send method of the JMS provider.
JMSTimeStamp	Records the time when the message was generated and sent for delivery. It is set by the send method of the JMS provider.
JMSCorrelationID	Set by the client application to link a message to another.
JMSReplyTo	Set by the client application. It specifies the identifier of the destination to which reply has to be sent.
JMSType	Identifies the structure of the message and the type of data in the body of the message. This is also set by the client application.
JMSRedelivered	Specifies whether the message is being resent.

JMS API Programming Model 8-9

Message properties – Message properties are used to provide compatibility with other messaging systems.

Message body - Following are the message types supported by JMS:

- **TextMessage** – This comprises data as Java strings.
- **MapMessage** – The data has map objects which are key-value pairs. The entities of a map object can be sequentially accessed through enumerator.
- **BytesMessage** – A stream of un-interpreted bytes that match a data format such as JPEG and so on.
- **StreamMessage** – A stream of values which are of primitive types.
- **ObjectMessage** - A serializable object.
- **Message** – Empty message body is also a valid message in JMS. This is used for administrative tasks.

JMS API Programming Model 9-9

JMS Queue Browsers

- The QueueBrowser object can browse through the messages in the queue and interpret the header values of all the messages in the queue.

JMS Topic

- JMS Topic is associated with the publish-subscribe messaging model.
- Stores the messages received from queues as in case of point-to-point communication.

Using Advanced JMS Features

Following are some of the advanced features of JMS that are used to improve the reliability and performance of the Java EE application:

- Controlling Message Acknowledgement
- Specifying Options for Sending Messages
- Creating Temporary Destinations
- Using JMS Local Transactions

Controlling Message Acknowledgement

In a communication session, a message acknowledgement is sent after the message processing is complete.

When the messages are sent as part of a transaction, then the messages are acknowledged once the transaction is committed.

The method of acknowledgement can be defined in the createContext(). Following are the values of the various parameters that can be passed to the createContext() method:

- JMSContext.AUTO_ACKNOWLEDGE
- JMSContext.CLIENT_ACKNOWLEDGE
- JMSContext.DUPS_OK_ACKNOWLEDGE

Specifying Options for Sending Messages

Following options can be set while sending messages to achieve various performance levels of the application:

- Define the persistence of the messages which will ensure that the messages are not lost in case of a JMS provider failure.
- Define priority levels for the messages which will define the order of delivery of the messages. The messages of higher priority are delivered first.
- Define an expiry time for the messages. If the expiry time of a message is lapsed, then the message is discarded and not delivered to the destination.
- Specify the delivery delay for the messages so that they can be scheduled for delivery at the right time.

Creating Temporary Destinations

JMS allows programmers to create temporary destinations for the JMS clients, which will last only as long as the connection exists.

Temporary destinations are objects of classes TemporaryQueue and TemporaryTopic.

The message consumers of these temporary destinations are those which are created during the connection.

They can receive messages from any message producers.

A local transaction can be used to group multiple messages to be sent and received, if all the messages belong to a single JMSContext.

Using JMS API in Java EE Applications

JMS API allows definition of an independent messaging system with different application clients or can be incorporated as a part of Java EE application.

In case of JMS application clients, multiple sessions can be associated with each connection object.

Creating Resources for Java EE Applications

- JMS resources can be added to Java EE applications either through resource injection or through deployment descriptor elements. Following namespaces can be used:
  - java:global namespace
  - java:app namespace
  - java:module namespace
  - java:component

Using Java EE Components to Produce and Synchronously Receive Messages

Though using synchronous messages is not a preferred design of an application, they can be used with definite timeout periods where reliability of messages is required.

A programmer should observe the following practices while managing the JMS resources from the Java EE components or Web components:

- When a JMS resource is required for the span of a business method, then the JMSContext resource has to be created in a try-with-resources block. Creating the resource with try-with-resources block ensures that the resource is closed at the end of the try block.
- The JMS resource has to be injected if the JMS resource has to be maintained for the duration of the transaction or request.

Using Message-Driven Beans to Receive Messages Asynchronously

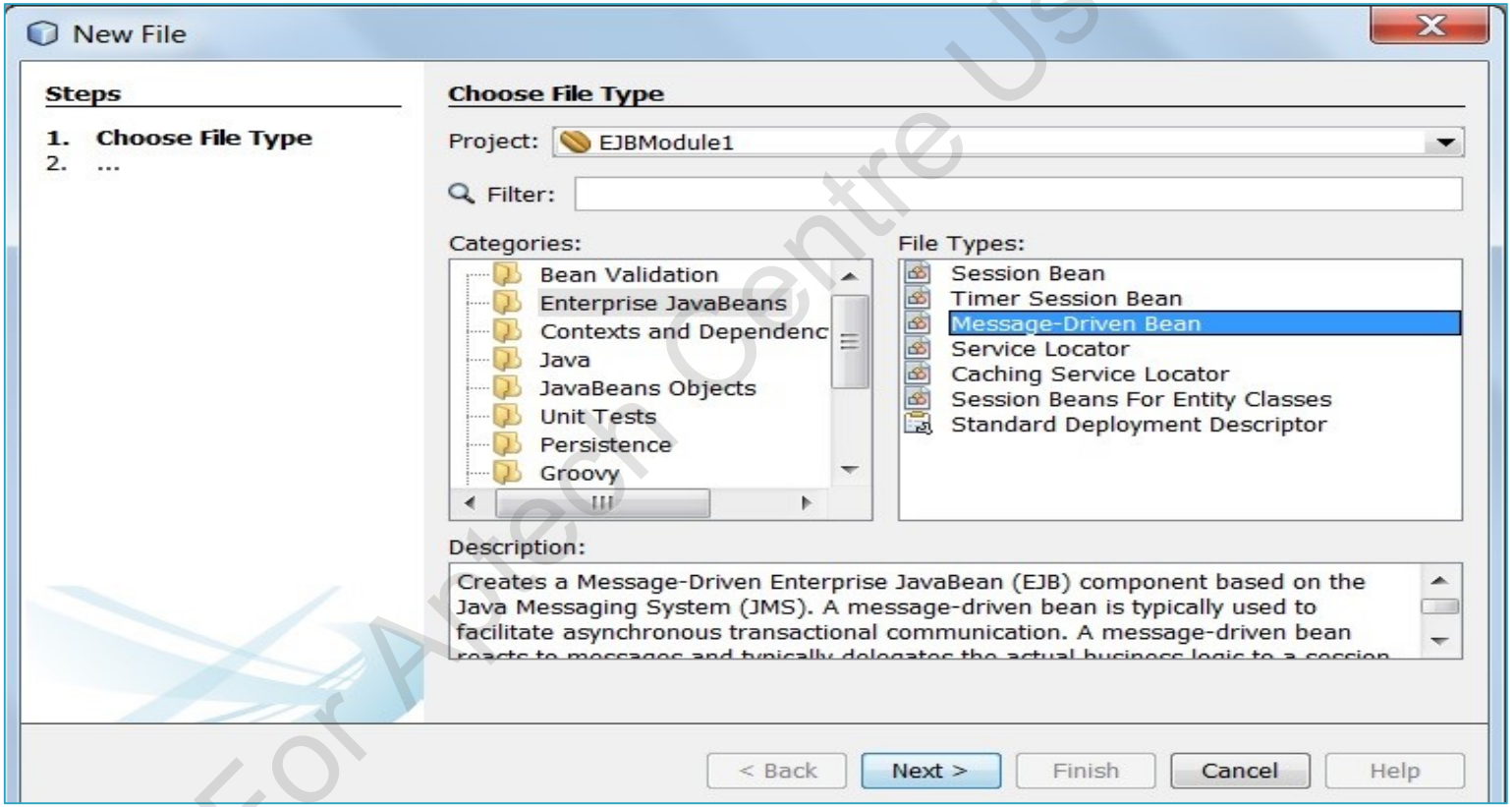
A message-driven bean is an enterprise bean in a Java application which is capable of processing JMS messages asynchronously.

It acts as a message listener for JMS messages. Following are the requirements of a message-driven bean:

- A message-driven bean can be declared in the deployment descriptor or can be injected through resource injection.
- The bean class should be public. It can be inherited by other classes hence, it cannot be abstract or final.
- It should have a public default constructor.

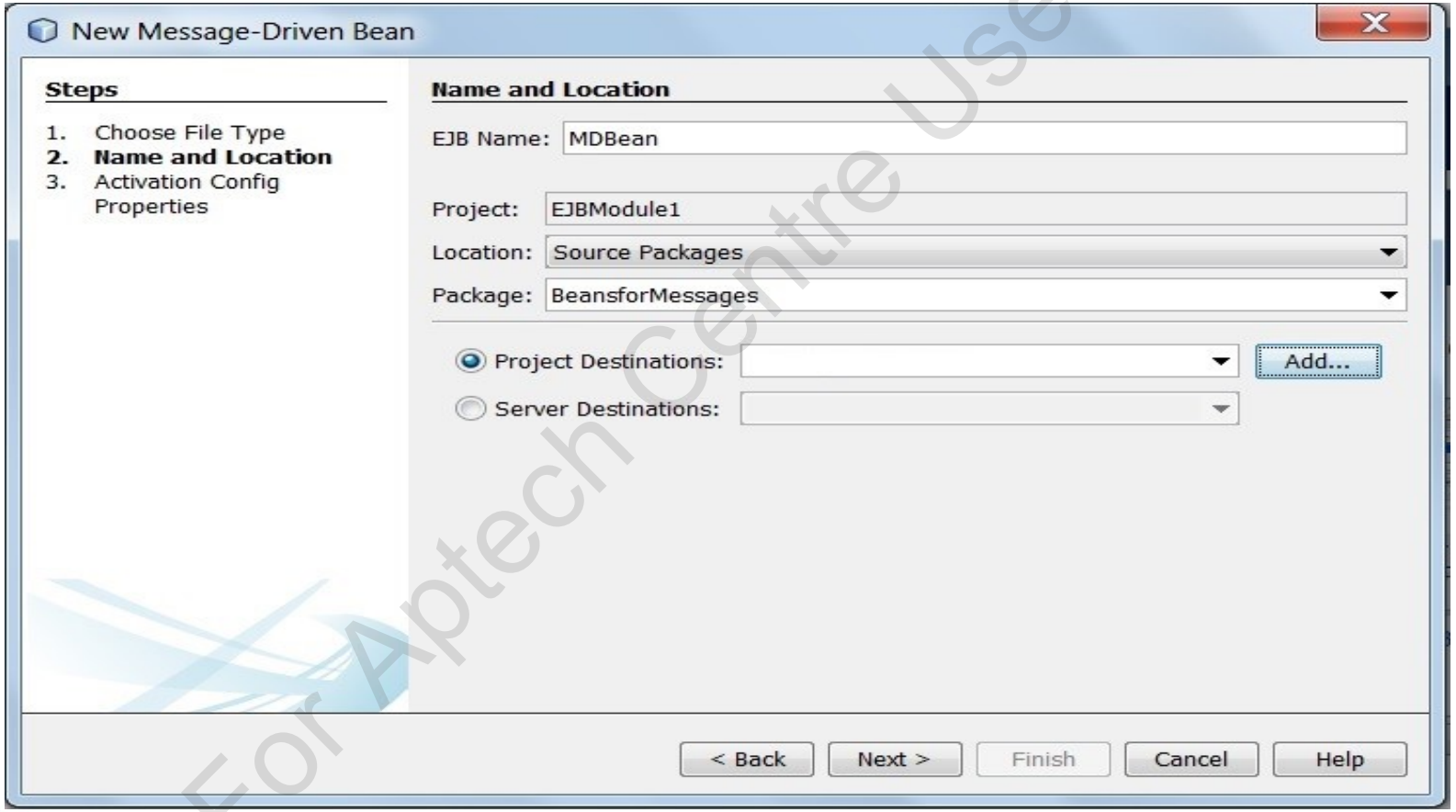
Creating a Message-Driven Bean 1-4

To create a Message-Driven Bean, create an EJB project and right-click the project name. Select New → Other → Enterprise JavaBeans → Message-Driven Bean from the New File dialog box as shown in the following figure:



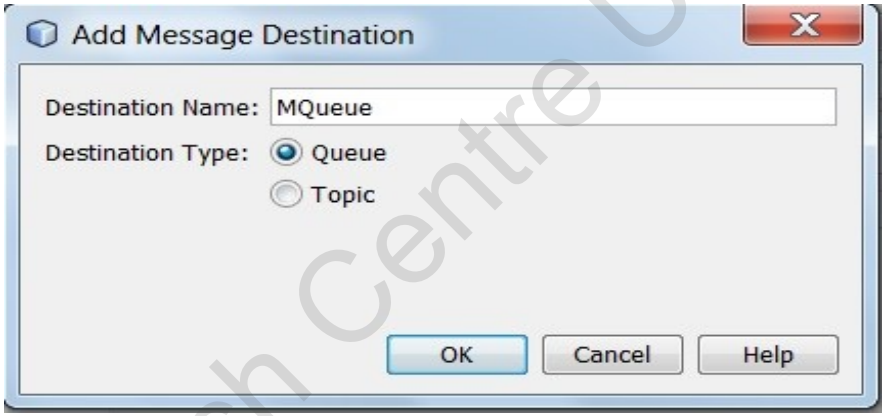
Creating a Message-Driven Bean 2-4

The New Message-Driven Bean dialog box is displayed which prompts for the EJB Name and Package of the message-driven bean, and destination of messages as shown in the following figure:



Creating a Message-Driven Bean 3-4

The destination of the message can be a topic or a queue which has to be selected by clicking Add. The Add Message Destination dialog box is displayed as shown in the following figure:

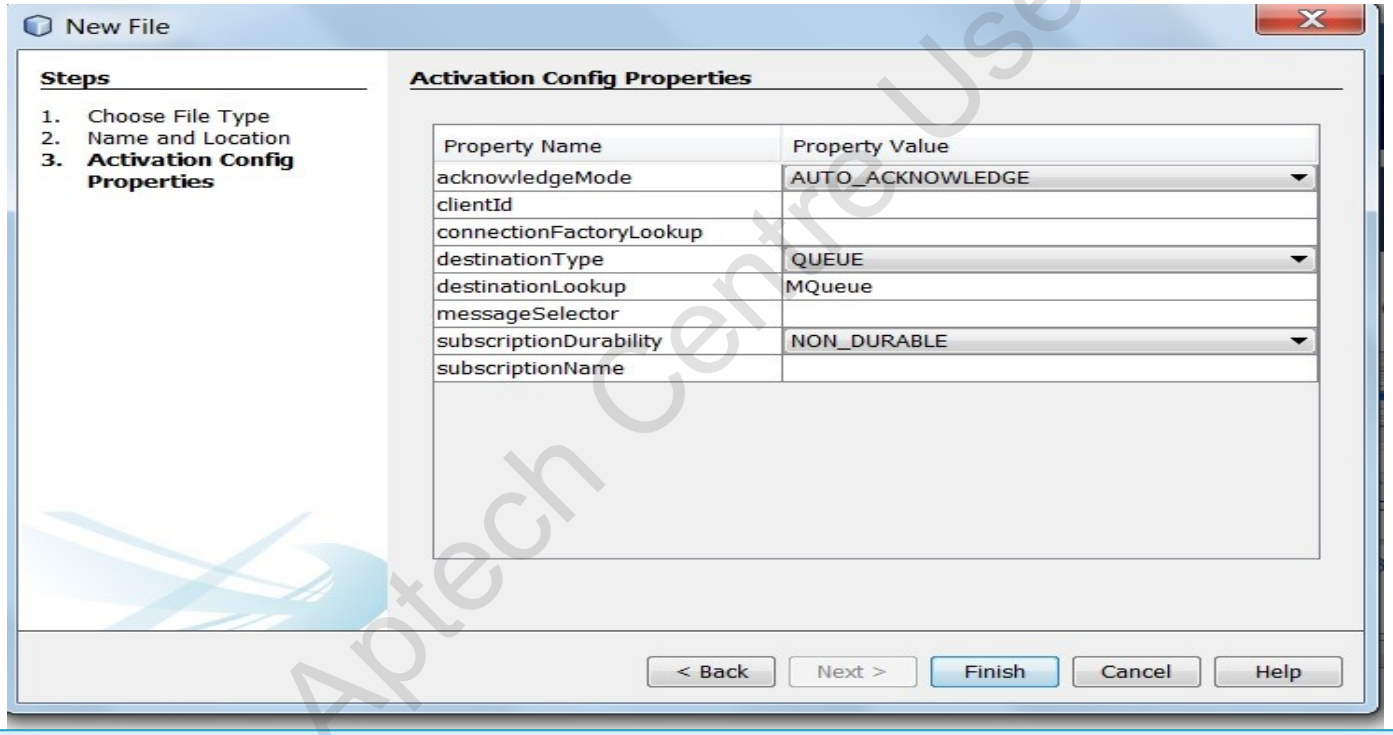


Select Queue. Provide the Destination Name as MQueue and click OK. The name, MQueue will be added to the Project Destinations drop-down.

Creating a Message-Driven Bean 4-4

Click Next. The Activation Config Properties screen is displayed.

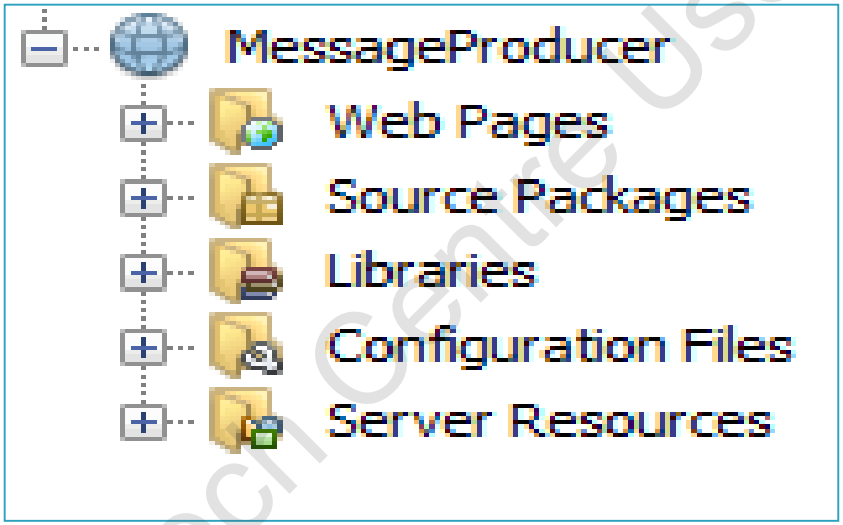
The configuration options for the destination are specified in this screen of the wizard as shown in the following figure:



A message-driven bean is created with the onMessage() method where the developer writes the code to be executed when this method is invoked.

Creating a Simple JMS Application 1-2

Create a Web application in Netbeans IDE, which can be used as the message producer. The MessageProducer Web application has been created using the JSF framework as shown in the following figure:



A typical JMS application will have a Producer entity, which produces the messages. These messages can be stored on a queue/topic and the messages can be accessed by a message receiver from the queue/topic by the consumer.

Creating a Simple JMS Application 2-2

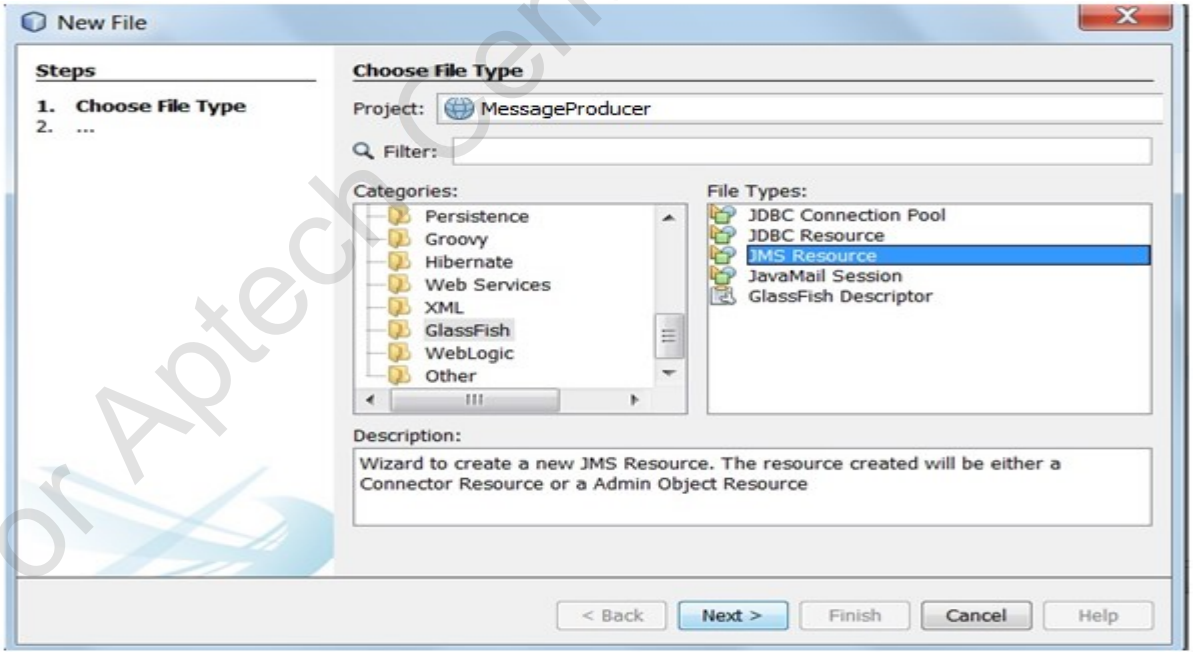
Following is the overview of steps for creating a message producer application:

- Create a Message Producer Web application.
- Create JMS admin objects of Queue and ConnectionFactory on the server.
- Create a JSFManagedBean to send/produce the message to the Queue.
- Create the Message-Driven bean to accept/consume message.

Creating the Queue and Connector Resources 1-4

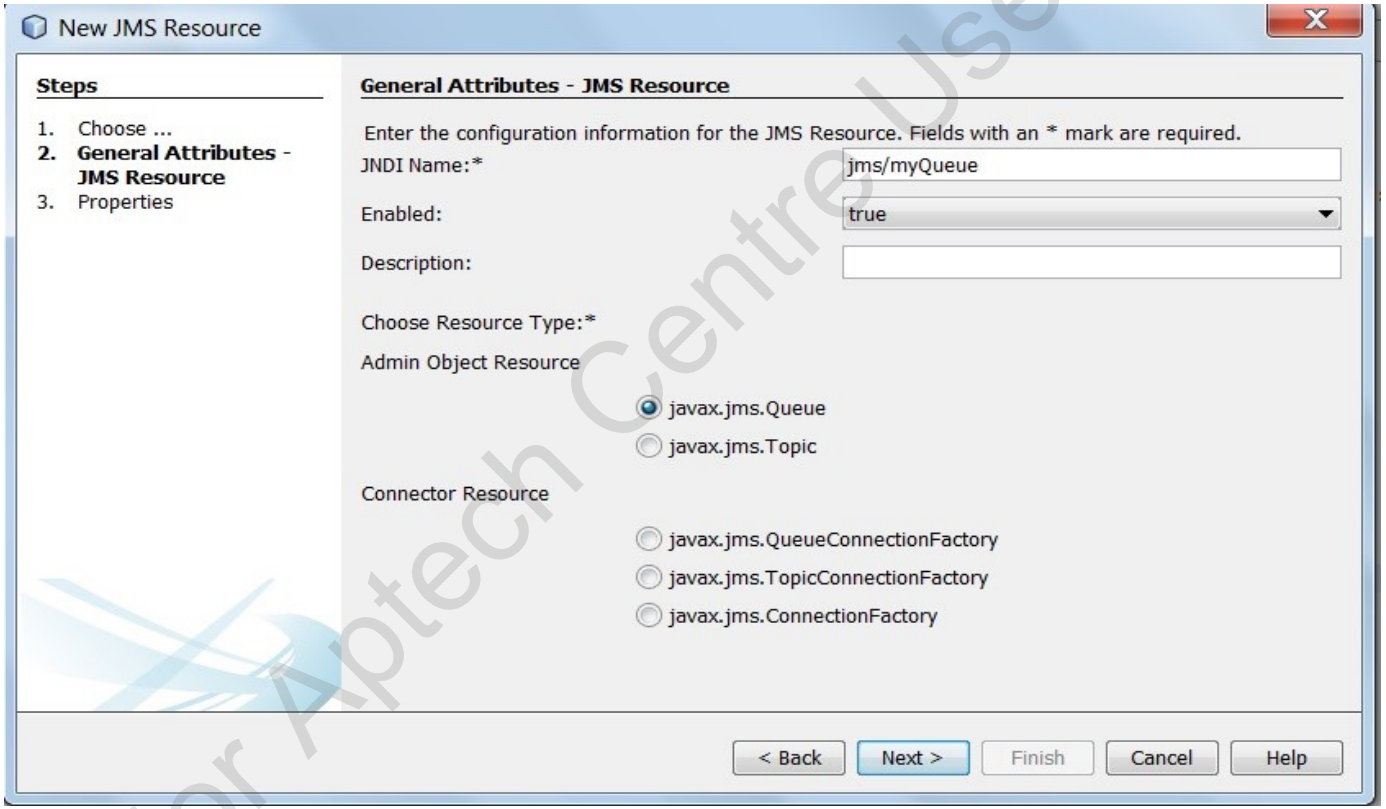
The message queue is created on the server. In this example, consider the instance of GlassFish server to which the JMS resources are added. Right-click the MessageProducer project and select New → Other → GlassFish → JMS Resource from the New File dialog box.

Following figure shows adding JMS resource to the application:



Creating the Queue and Connector Resources 2-4

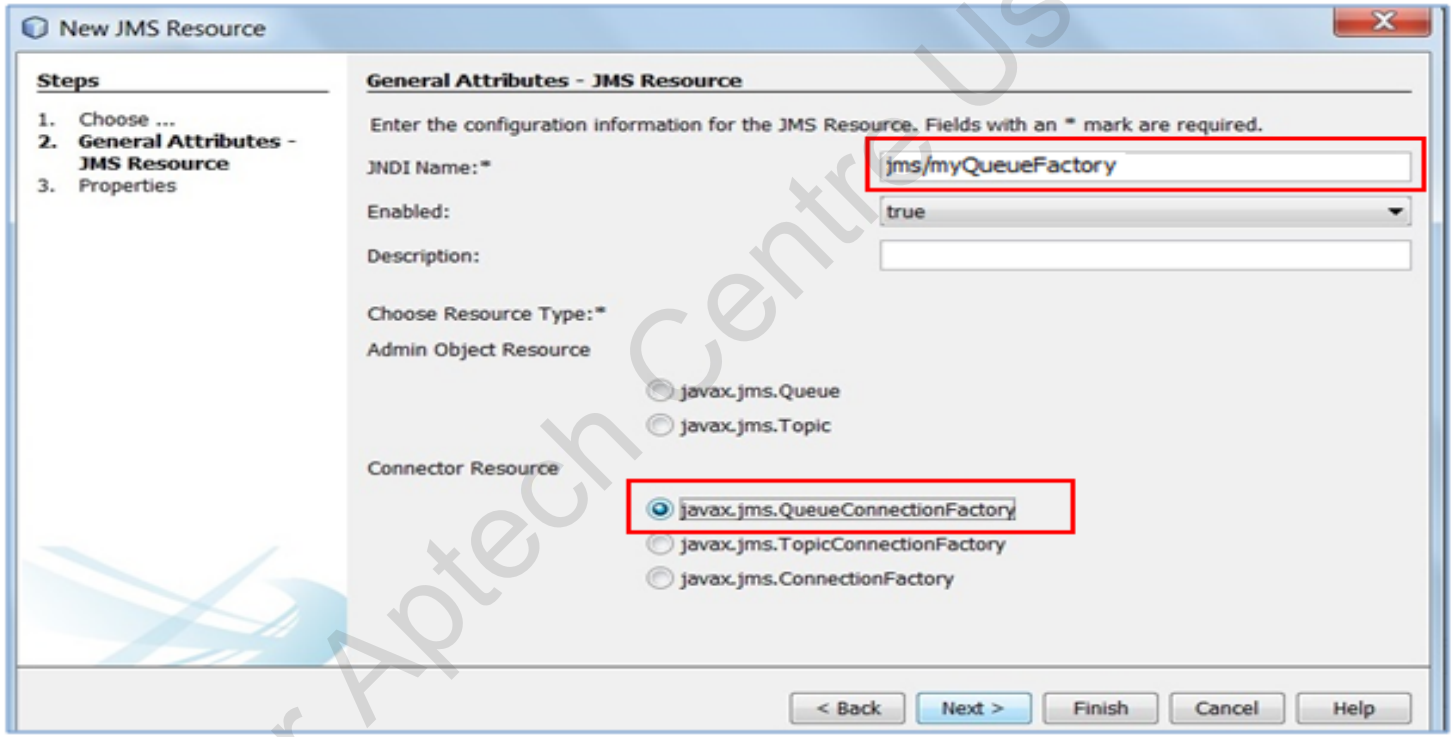
The developer can add a Queue object and a Connector object one after the other through the wizard used to add JMS resource to the server (GlassFish server). Following figure shows the wizard:



Click Next. On the next screen, provide myQueue in the Value field of the Name property. Click Finish.

Creating the Queue and Connector Resources 3-4

Now, create a Connection Factory resource. Right-click the project and select New → Other → GlassFish → JMS Resource from the New File dialog box. Click Next and specify the options as shown in the following figure to create a QueueConnectionFactory object:

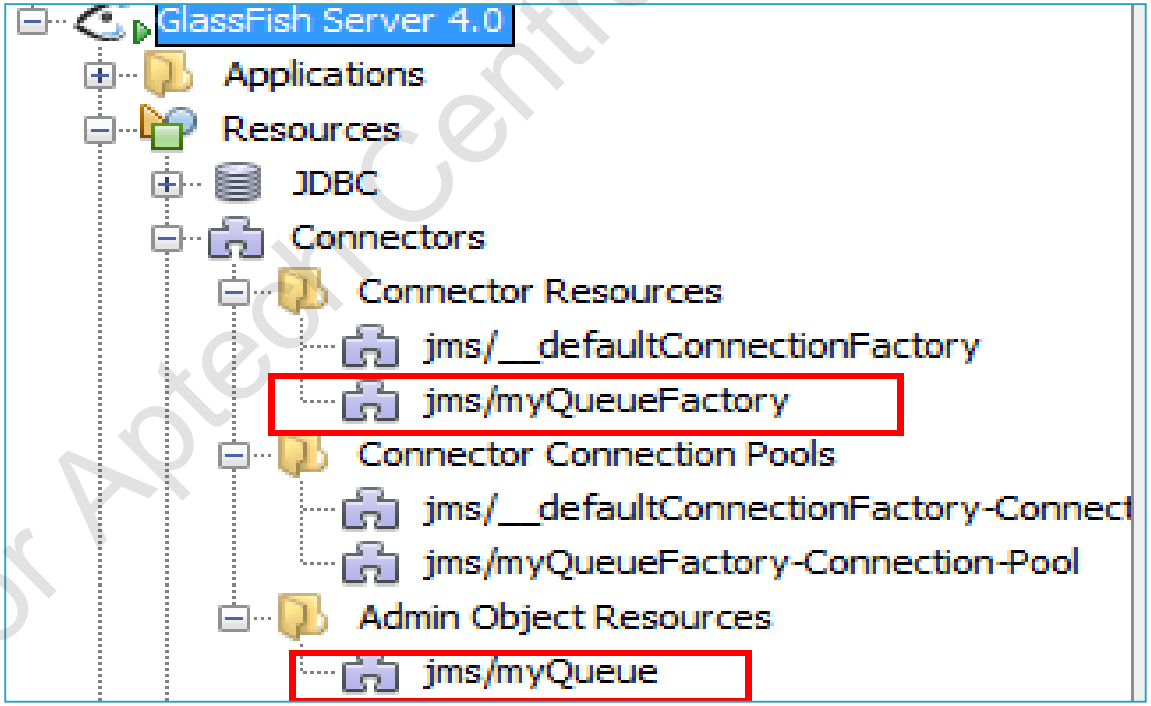


Click Next. The JMS Properties screen is displayed. Here, you can specify additional configuration information. Click Finish.

Creating the Queue and Connector Resources 4-4

Deploy the MessageProducer application on the server.

The Admin Object Resource, jms/myQueue, and a Connector Resource object, jms/myQueueFactory will appear in the respective folders as shown in the following figure:



Creating Message Producer Bean 1-5

To create a JSF ManagedBean that will act as the message producer, right-click the project name and select New → Other → JavaServer Faces → JSF ManagedBean from the New File dialog box.

Click Next. The New JSF ManagedBean dialog box is displayed.

Specify the Class Name of the bean as MessageProducerBean and the Package as mes.

Click Finish. The ManagedBean is created.

Creating Message Producer Bean 2-5

Add a property named 'message' to the bean and create the respective getter and setter methods. Following demonstrates the resultant code in the JSF ManagedBean:

```
package mes;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
@ManagedBean
@RequestScoped
public class MessageProducerBean {
    /**
     * Creates a new instance of MessageProducerBean
     */
    private String message;
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public MessageProducerBean() {
    }
}
```

Creating Message Producer Bean 3-5

In the editor area of MessageProducerBean, press Alt+Insert to open the NetBeans Code Generator feature and select Send JMS Message.

The Send JMS Message dialog box is displayed.

Ensure that Project Destination is set to jms/myQueue and Connection Factory is set to jms/myQueueFactory.

Click OK. NetBeans adds the proper resource declarations to your code for the Queue and ConnectionFactory instances as shown in the given code.

```
@ManagedBean
@RequestScoped
public class MessageProducerBean {
    @Resource(mappedName = "jms/myQueue")
    private Queue myQueue;
    @Inject

@JMSConnectionFactory("jms/myQueueFactory")
    private JMSContext context;
    /**
     * Creates a new instance of
MessageProducerBean
     */
    private String message;
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public MessageProducerBean() {
    }
}
```

Creating Message Producer Bean 4-5

Add a new send() method to the MessageProducerBean.

Following are the steps for adding a new send() method:

- Initial context of the application is invoked.
- Acquire the current instance of FacesContext, to retrieve the user interface state of the application.
- Use the admin objects created on the Web application server (GlassFish) to bind the connection and queue objects to JNDI namespace.
- Create JMS objects – Connection, Session, and MessageProducer.
- Send the message received from the JSF page to the queue.

Creating Message Producer Bean 5-5

Following code demonstrates the send method to be added to the MessageProducerBean:

```
…
public void send() throws NullPointerException, NamingException, JMSException {
        InitialContext initContext = new InitialContext();
        FacesContext facesContext = FacesContext.getCurrentInstance();
        ConnectionFactory factory =(ConnectionFactory)
initContext.lookup("jms/myQueueFactory");
        Destination destination = (Destination) initContext.lookup("jms/myQueue");
        initContext.close();
    //Create JMS objects
    Connection connection = factory.createConnection();
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageProducer sender = session.createProducer(myQueue);
   //Send messages
   TextMessage msg = session.createTextMessage(message);
   sender.send(msg);
   FacesMessage facesMessage = new FacesMessage("Message sent: " + message);
   facesMessage.setSeverity(FacesMessage.SEVERITY_INFO);
   facesContext.addMessage(null, facesMessage);
}
```

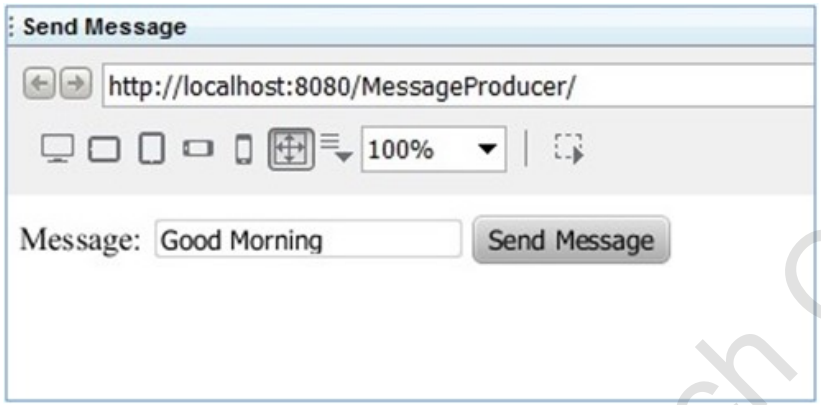
Creating User Interface

Following code demonstrates the code of the JSF page sending the message:

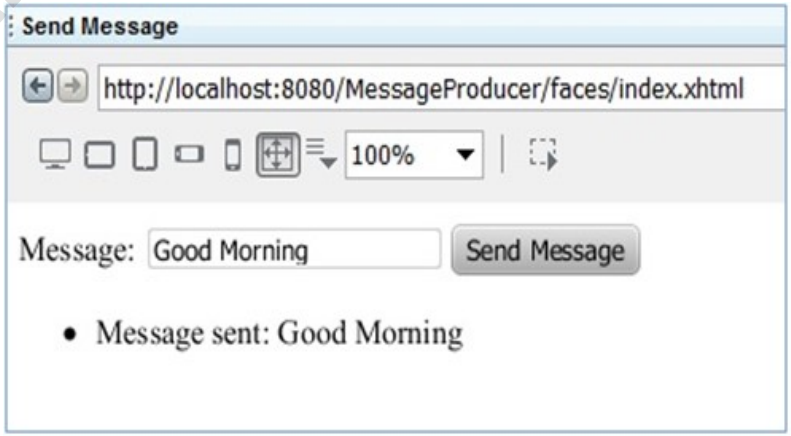
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Send Message</title>
    </h:head>
    <h:body>
        <h:form>
        <h:outputLabel value =" Message :"/>
        <h:inputText id="message" value="#{messageProducerBean.message}"
size="30" />
        <h:commandButton id="Send" value="Send Message"
            action="#{messageProducerBean.send()}"/>

        <h:messages globalOnly ="true"></h:messages>
        </h:form>
    </h:body>
</html>
```

Deploy and Run the Application

Following figure shows the sender page of the MessageProducer.



Following figure shows the message producer page where the user can type a message and on clicking the button the send() method is invoked which sends the message to the message queue:

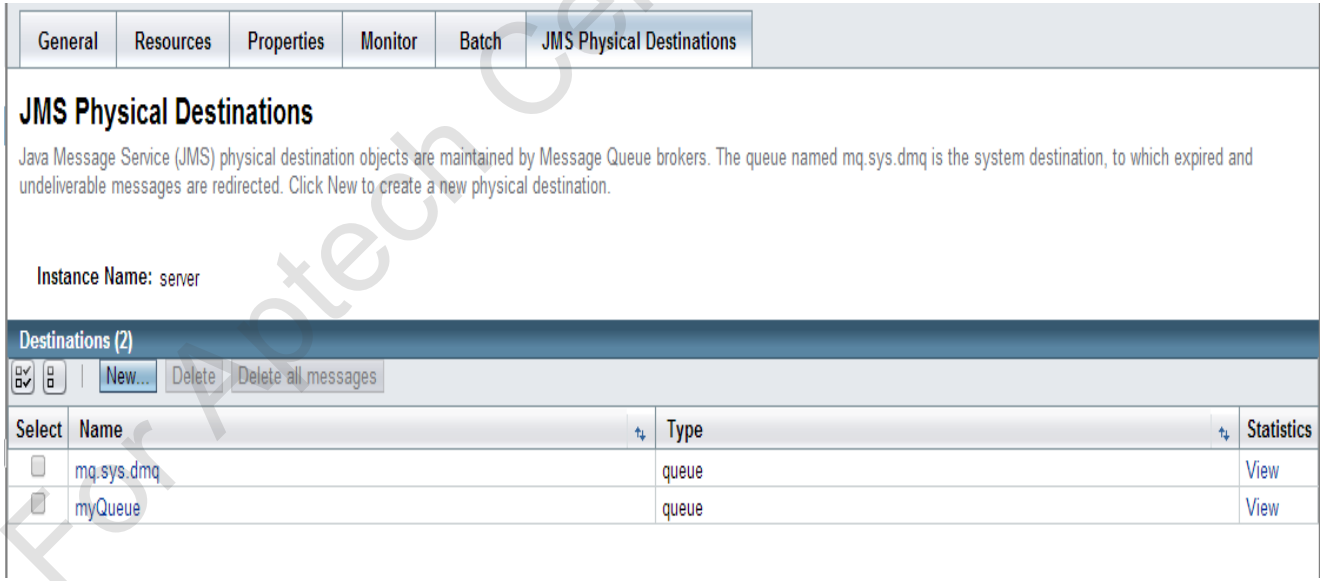


Viewing the Queue on the Admin Console

To view the Queue resource created on the server, right-click the GlassFish Server in the Services tab and select View Admin Console.

The Sever Admin Console is displayed in the browser. Select Server (Admin) from the left pane.

Click JMS Physical Destinations tab. The list of available destinations is displayed as shown in the following figure:

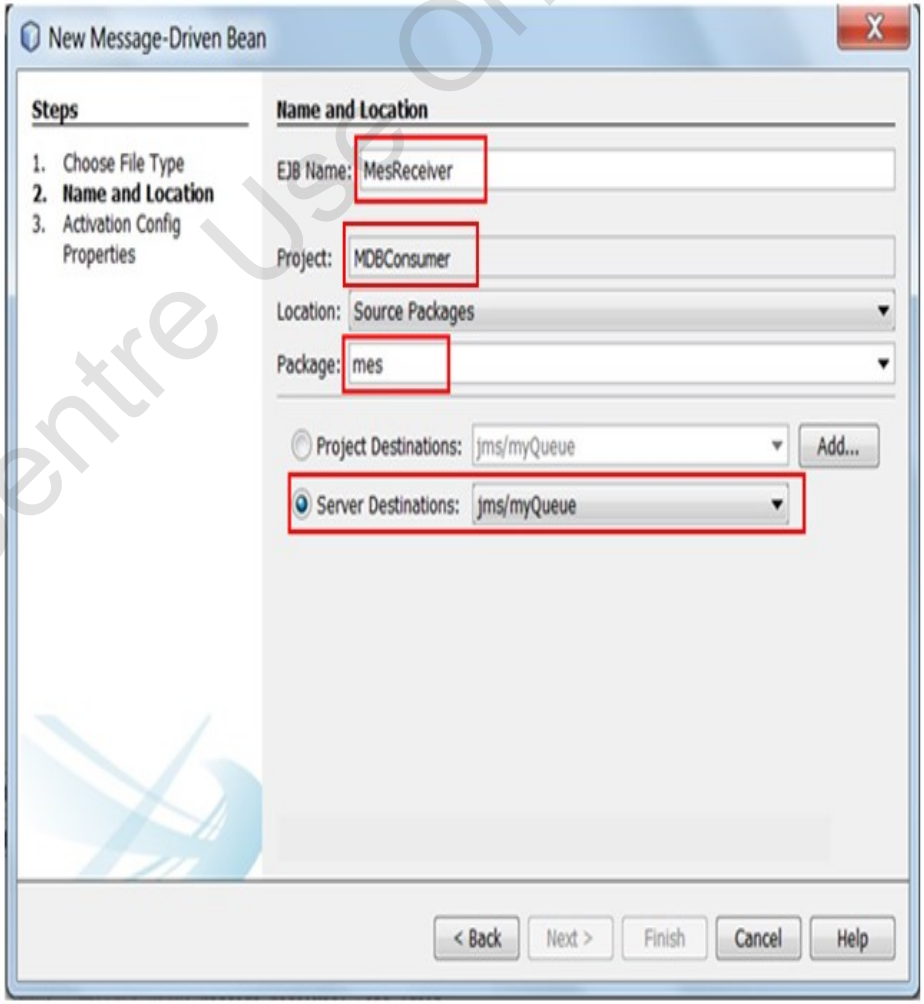


Creating the Message Consumer 1-5

Create a new EJB project by clicking File → New Project → Java EE → EJB Module. Name the project as MDBConsumer. Click Next and then, click Finish.

For the MessageProducer application, a Message-Driven Bean MesReceiver can be created by right-clicking the MDBConsumer project and selecting New → Other → Enterprise JavaBeans → Message-Driven Bean.

Click Next and specify the values for Name and Location as shown in the figure.



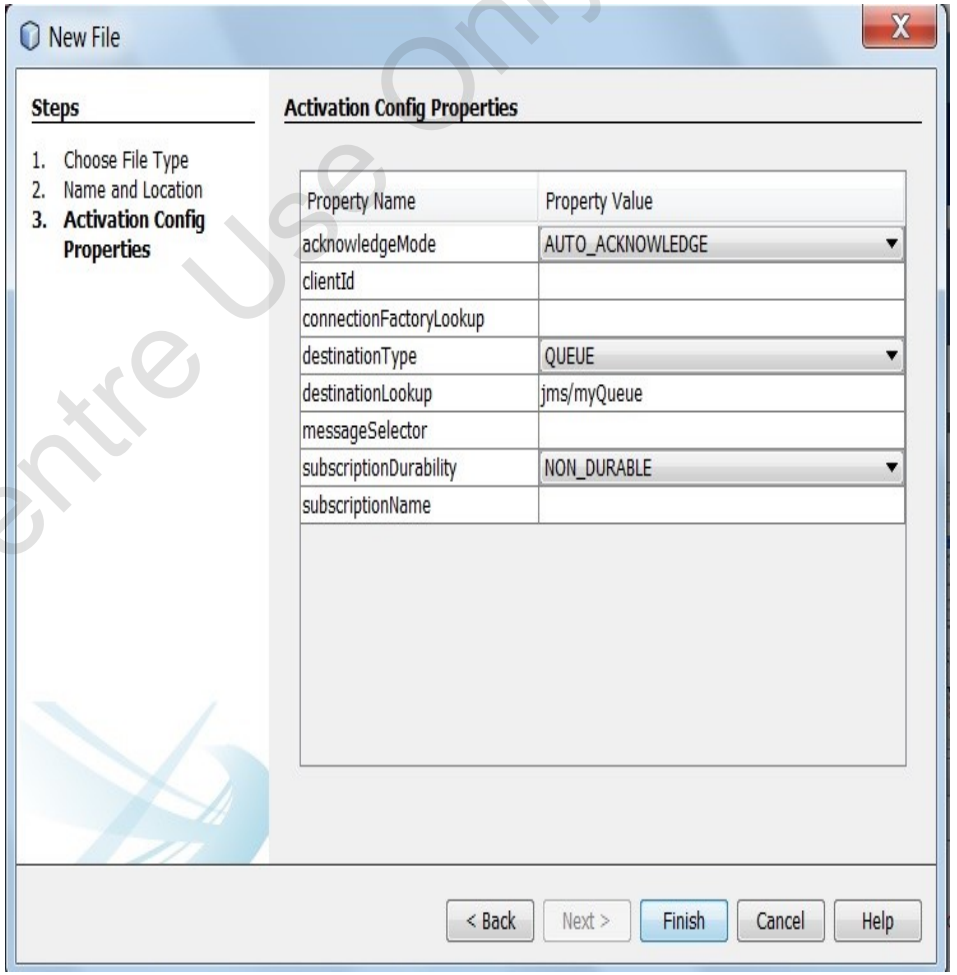
Creating the Message Consumer 2-5

MessageProducerBean sends a message to a Queue object on the server, in this case, a GlassFish Server has been used.

The Message-Driven Bean is configured to receive messages from the destination queue jms/myQueue.

The figure shows the Activation Config properties of the Message-Driven Bean used.

Click Next.



Creating the Message Consumer 3-5

Click Finish. Following code demonstrates the creation of the Message-Driven bean with annotations mapping it to the jms/Queue resource on the server:

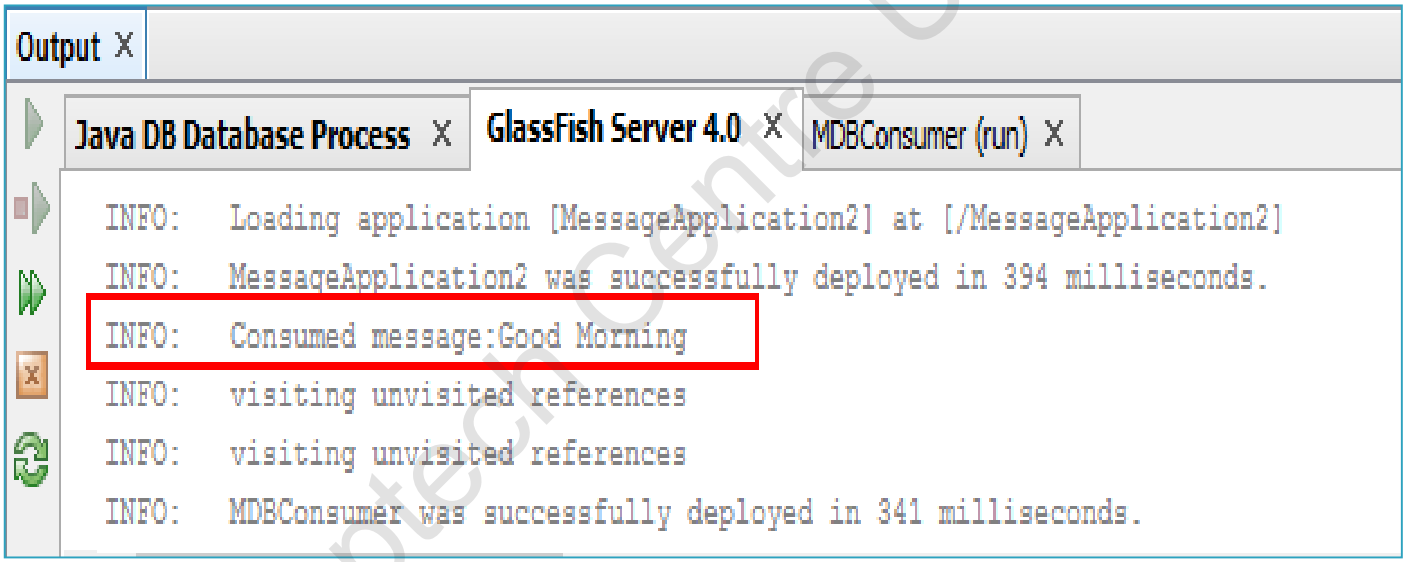
```
package mes;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue
= "jms/myQueue")
})
public class MesReceiver implements MessageListener  {
        public MesReceiver(){

    }
```

Creating the Message Consumer 4-5

```
@Override
    public void onMessage(Message message) {
    if (message instanceof TextMessage) {
         TextMessage msg = (TextMessage) message;
        try {
            System.out.println("Consumed message: " + msg.getText());
        } catch (JMSException ex) {

            Logger.getLogger(MesReceiver.class.getName()).log(Level.SEVERE, null,
ex);
             }
           } else {
              System.out.println("Message of wrong type: " +
message.getClass().getName());

    }

}
}
```

Creating the Message Consumer 5-5

Deploy the MDBConsumer project. In the Output window, click the GlassFish Server tab. The consumed message is visible as shown in the following figure:



Summary

- JMS API is used to set up the messaging infrastructure for a Java EE application. It can also be used to develop an independent messaging system.
- JMS follows two messaging models – point-to-point messaging and publish-subscribe model.
- The message exchange between JMS message producer and receiver is primarily asynchronous.
- JMS API provides for both synchronous and asynchronous communication.
- Synchronous communication is accomplished through queue objects and asynchronous communication through topic objects.
- JMS communication is initiated after the communicating message producer and receiver are bound through JNDI namespace.
- JMS messaging system includes various administrative objects such as message producers, message consumers, connections, sessions, and so on.
- JMS Messages can be handled through Message-Driven Beans. Message-Driven Beans are invoked when a message is received on the destination.