UNIT 3 EJB: DEPLOYING ENTERPRISE JAVA BEANS

Structure **Page Nos.**

3.0 INTRODUCTION

In the previous unit, we have learnt the basics of Enterprise java beans and different type of java beans. In this unit we shall learn how to create and deploy Enterprise java beans. "Enterprise JavaBeans" (EJBs) are distributed network aware components for developing secure, scalable, transactional and multi-user components in a J2EE environment. EJBs are a collection of Java classes, interfaces and XML files adhering to given rules. In J2EE all the components run inside their own containers. As we have learnt, JSP Servlets have their own web container and run inside that container. Similarly, EJBs run inside EJB container. The container provides certain built-in services to EJBs which the EJBs use to function. Now, we will learn how to create our first Enterprise JavaBean and we will then deploy this EJB on a production class, open source, and free EJB Server; JBoss.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- provide an overview of XML;
- discuss how XML differs from HMTL;
- understand the SGML and its use;
- analyse the difference between SGML and XML;
- understand the basic goals of development of XML;
- explain the basic structure of XML document and its different components;
- understand what DTD is and how do prepare DTD for an XML document;
- learn what an XML parser is and its varieties, and
- differentiate between the different types of entities and their uses.

3.2 DEVELOPING THE FIRST SESSION EJB

We have already learnt in the previous unit about Session EJBs which are responsible for maintaining business logic or processing logic in our J2EE applications. Session beans are the simplest beans and are easy to develop so, first, we will develop Session beans. Every EJB class file has two accompanying interfaces and one XML file. The two interfaces are :

- **Remote Interfaces:** Remote interface is what the client gets to work with, in other words Remote interface should contain the methods you want to expose to your clients.

- **Home Interfaces:** Home interface is actually EJB builder and should contain methods used to create Remote interfaces for your EJB. By default, the Home interface must contain at least one create() method.
  The actual implementation of these interfaces, our Session EJB class file remains hidden from the clients.

The XML file will be named as “ejb-jar.xml” and will be used to configure the EJBs during deployment. So, in essence our EJB, FirstEJB, which we are going to create consists of the following files :

```
Ejb
        session
                First.java
                FirstHome.java
                FirstEJB.java
META-INF
          ejb-jar.xml
```

“First.java” will be the Remote interface we talked about. “FirstHome.java” is our Home interface and “FirstEJB.java” is the actual EJB class file. The ejb-jar.xml deployment descriptor file goes in the META-INF folder.

Now, create a new folder under the C:\Projects folder which we had created earlier, and name it “EJB”. Now, create a new sub-folder under C:\Projects\EJB folder and name it “FirstEJB”. Create a new folder "src" for Java source files in the "FirstEJB' folder. The directory structure should look like the following :

```
C:\Projects
   TomcatJBoss
   EJB
     FirstEJB
       Src
```

Now, create directory structure according to the package ignou.mca.ejb.session in the "src" folder. If, you know how package structure is built, it shouldn't be a problem. Anyhow, the final directory structure should like the following :

```
C:\Projects
  TomcatJBoss
```

```
EJB
   FirstEJB
     src
       ignou
         mca
           ejb
             session
```

First.java :

Now, we shall create the First.java source file in ignou/mca/ejb/session folder. Type the following code in it:

```
/* First.java */

package ignou.mca.ejb.session;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface First extends EJBObject {

   public String getTime() throws RemoteException;
}
```

Now save this file as first.java

First line is the package statement which indicates that the first interface belongs to the ignou.mca.ejb.session package:

```
package com.stardeveloper.ejb.session;
```

The next two lines are import statements for importing the required classes from the packages.

import javax.ejb.EJBObject;

import java.rmi.RemoteException;

Then, comes the interface declaration line which depicts that this is an interface with name “First” which extends an existing interface javax.ejb.EJBObject.

Note: Every Remote interface *must* always extend EJBObject interface. It is a requirement, not an option.

public interface First extends EJBObject {

Now, we have to declare methods in Remote interface which we want to be called for the client (which the client can access and call). For simplicity, we will only declare a single method, getTime(); which will return a String object containing current time.

```
public String getTime() throws RemoteException;
```

We should notice that there are no {} parenthesis in this method as has been declared in an interface. Remote interface method's must also throw RemoteException because EJBs are distributed components and during the call to an EJB, due to some network problem, exceptional events can arise, so all Remote interface method's must declare that they can throw RemoteException in Remote interfaces.

FirstHome.java :

Now, let us create the home interface for our FirstEJB. Home interface is used to create and get access to Remote interfaces. Create a new FirstHome.java source file in ignou/mca/ejb/session package. Type the following code in it:

```
/* FirstHome.java */
package com.stardeveloper.ejb.session;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface FirstHome extends EJBHome {

   public First create() throws CreateException, RemoteException;
}
```

Now, save this file as FirstHome.java.

First few lines are package and import statements. Next, we have declared our FirstHome interface which extends javax.ejb.EJBHome interface.

Note: All Home interfaces *must* extend EJBHome interface.

```
public interface FirstHome extends EJBHome {
```

Then, we declared a single create() method which returns an instance of First Remote interface. Notice that all methods in Home interfaces as well must also declare that they can throw RemoteException. One other exception that must be declared and thrown is CreateException.

```
public First create() throws CreateException, RemoteException;
```

We are done with creating Remote and Home interfaces for our FirstEJB. These two are the only things which our client will see, the client will remain absolutely blind as far as the actual implementation class, FirstEJB is concerned.

FirstEJB.java :

FirstEJB is going to be our main EJB class. Create a new FirstEJB.java source file in

ignou/mca/ejb/session folder. Type the following code in it:

```
/* FirstEJB.java */
package ignou.mca.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.EJBException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
import java.util.Date;

public class FirstEJB implements SessionBean {

   public String getTime() {
      return "Time is : " + new Date().toString();
   }
   public void ejbCreate() {}
   public void ejbPassivate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
   public void setSessionContext(SessionContext context) {}
}
```

Now, save this file as FirstEJB.java.

The first few lines are again the package and import statements. Next, we have declared our FirstEJB class and made it implement javax.ejb.SessionBean interface.

Note: All Session bean implementation classes must implement SessionBean interface.

```
public class FirstEJB implements SessionBean
```

Our first method is getTime() which had been declared in our First Remote interface. We implement that method here in our FirstEJB class. It simply returns get date and time as can be seen below:

```
public String getTime() {

      return "Time is : " + new Date().toString();

   }
```

Then comes 5 callback methods which are part of SessionBean interface and since we are implementing SessionBean interface, we have to provide empty implementations of these methods.

ejb-jar.xml :

Let’s now create the EJB deployment descriptor file for our FirstEJB Session bean. Create a new ejb-jar.xml file in the FirstEJB/META-INF folder. Copy and paste the following text in it:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems,
   Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
```

```
  <description></description>
  <enterprise-beans>
    <session>
      <display-name>FirstEJB</display-name>
      <ejb-name>First</ejb-name>
      <home>ignou.mca.ejb.session.FirstHome</home>
      <remote>ignou.mca.ejb.session.First</remote>
      <ejb-class>ignou.mca.ejb.session.FirstEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
     <method>
        <ejb-name>First</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Supports</trans-attribute>
    </container-transaction>

    <security-role>
      <description>Users</description>
      <role-name>users</role-name>
    </security-role>
  </assembly-descriptor>
</ejb-jar>
```

One or more EJBs are packaged inside a JAR (.jar) file. There should be only one ejb-jar.xml file in an EJB JAR file. So ejb-jar.xml contains deployment description for one or more than one EJBs. Now as we learned in an earlier unit, there are 3 types of EJBs so ejb-jar.xml should be able to contain deployment description for all 3 types of EJBs.

Our ejb-jar.xml file for FirstEJB contains deployment description for the only EJB we have developed; FirstEJB. Since it is a Session bean, it's deployment description is contained inside <session></session> tags.

```
<ejb-jar>
  <description></description>
  <enterprise-beans>
    <session></session>
  </enterprise-beans>
</ejb-jar>
```

Now, let us discuss different deployment descriptor tags inside the <session></session> tag. First is the <ejb-name> tag. The value of this tag should be name of EJB i.e. any name you think should point to your Session EJB. In our case it's value is "First". Then, comes <home>, <remote> and <ejb-class> tags which contain complete path to Home, Remote and EJB implementation classes. Then comes <session-type> tag whose value is either "Stateless" or "Stateful". In our case it is "Stateless" because our Session bean is stateless. Last tag is <transaction-type>, whose value can be either "Container" or "Bean". The Transactions for our FirstEJB will be managed by the container.

```
<ejb-jar>
  <description></description>
  <enterprise-beans>
    <session>
      <display-name>FirstEJB</display-name>
      <ejb-name>First</ejb-name>
      <home>ignou.mca.ejb.session.FirstHome</home>
      <remote>ignou.mca.ejb.session.First</remote>
      <ejb-class>ignou.mca.ejb.session.FirstEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Then there is a <container-transaction> tag, which indicates that all methods of FirstEJB "support" transactions.

Compiling the EJB Java Source Files :

Now, our directory and file structure till now looks something like the following:

```
C:\Projects
  TomcatJBoss
  EJB
    FirstEJB
      src
        ignou
          mca
            ejb
              session
                First.java
                FirstHome.java
                FirstEJB.java
      META-INF
        ejb-jar.xml
```

We can compile all the Java source file by using a command like the following on the command prompt:

```
C:\Projects\EJB\FirstEJB\src\ignou\mca\ejb\session>
          javac -verbose -classpath %CLASSPATH%;C:\JBoss\client\jboss-j2ee.jar
    -d C:\Projects\EJB\FirstEJB *.java
```

Note: The point to remember is to make sure jboss-j2ee.jar (which contains J2EE package classes) in the CLASSPATH, or you will get errors when trying to compile these classes.

If, you have installed JBoss Server in a separate directory then, substitute the path to jboss-j2ee.jar with the one present on your system. The point is to put jboss-j2ee.jar in the CLASSPATH for the javac, so that all EJB source files compile successfully.

3.3 PACKAGING EJB SOURCE FILES INTO A JAR FILE

Till now our directory and file structure should look something like the following:

```
C:\Projects
  TomcatJBoss
  EJB
    FirstEJB
      ignou
        mca
          ejb
            session
              First.class
              FirstHome.class
              FirstEJB.class
      META-INF
        ejb-jar.xml
      src
        ignou
          mca
            ejb
              session
                First.java
                FirstHome.java
                FirstEJB.java
```

Now, to package the class files and XML descriptor files together, we will use the concept of jar file, which act as a container. For creating the jar file, run the following command at the command prompt:

```
C:\Projects\EJB\FirstEJB>jar cvfM FirstEJB.jar com META-INF
```

After running this command, EJB JAR file will be created with the name FirstEJB.jar in the FirstEJB folder. After this Jboss configuration file is to be added.

Adding JBoss specific configuration file :

Till now our FirstEJB.jar file contains generic EJB files and deployment description. To run it on JBoss we will have to add one other file into the META-INF folder of this JAR file, called jboss.xml. This file contains the JNDI mapping of FirstEJB.
So, create a new jboss.xml file in the FirstEJB/META-INF folder where ejb-jar.xml file is present and type the following text in it:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
        "http://www.jboss.org/j2ee/dtd/jboss.dtd">
<jboss>
        <enterprise-beans>
                <session>
                        <ejb-name>First</ejb-name>
                        <jndi-name>ejb/First</jndi-name>
                </session>
        </enterprise-beans>
</jboss>
```

Now, we will add this new jboss.xml file into our existing FirstEJB.jar file by running the following command at the DOS prompt:

```
C:\Projects\EJB\FirstEJB>jar uvfM FirstEJB.jar META-INF
```

Now, we will deploy the FirstEJB.jar on the JBoss Server.

3.4 DEPLOYING JAR FILE ON JBOSS SERVER

Now, we will deploy the FirstEJB.jar file on the Jboss Server by copying the FirstEJB.jar file and pasting it into the C:\JBoss\deploy folder. If JBoss is running, you should see text messages appearing on the console that this EJB is being deployed and finally deployed and started.

Creating the Client JSP Page:

Now we will create a new WEB-INF folder in C:\Projects\TomcatJBoss folder. Then create a new web.xml file and type the following text in it :

```
<?xml version= “1.0” encoding= “ISO-8859-1”?>

<!DOCTYPE web-app
   PUBLIC “-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN”
   “http://java.sun.com/j2ee/dtds/web-app_2.3.dtd”>

<web-app>
</web-app>
```

As we can see, this web.xml is almost empty and is not doing anything useful. We still created it because Tomcat will throw an error if you try to access this /jboss context that we had created earlier without creating a /WEB-INF/web.xml file.

FirstEJB.jsp JSP Client page :

We will now create a new JSP page in the C:\Projects\TomcatJBoss folder and save it as "firstEJB.jsp". Now type the code in it :

```
<%@ page import= “javax.naming.InitialContext,
             javax.naming.Context,
             java.util.Properties,
             ignou.mca.ejb.session.First,
             ignou.mca.ejb.session.FirstHome"%>
<%
   long t1 = System.currentTimeMillis();
   Properties props = new Properties();
      props.put(Context.INITIAL_CONTEXT_FACTORY,
         "org.jnp.interfaces.NamingContextFactory");
      props.put(Context.PROVIDER_URL, "localhost:1099");

   Context ctx = new InitialContext(props);
   FirstHome home = (FirstHome)ctx.lookup("ejb/First");
   First bean = home.create();
   String time = bean.getTime();
   bean.remove();
```

```
    ctx.close();
    long t2 = System.currentTimeMillis();
%>
<html>
<head>
    <style>p { font-family:Verdana;font-size:12px; }</style>
</head>
<body>
<p>Message received from bean = "<%= time %>".<br>Time taken :
    <%= (t2 - t1) %> ms.</p>
</body>
</html>
```

Now, save this file as firstEJB.jsp as JSP page.

As we can see, the code to connect to an *external* JNDI/EJB Server is extremely simple. First, we have created a Properties object and put certain values for Context.INITIAL_CONTEXT_FACTORY and Context.PROVIDER_URL properties.

The value for Context.INITIAL_CONTEXT_FACTORY is the interface provided by JBoss and the value for Context.PROVIDER_URL is the location:port number where JBoss is running. Both these properties are required to connect to an external JNDI Server.

```
    Properties props = new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.PROVIDER_URL, "localhost:1099");
```

We then get hold of that external JNDI Context object by creating a new InitialContext() object, it's argument being the Properties object we had created earlier.

```
Context ctx = new InitialContext(props);
```

Next, we have used that external JNDI Context handle to lookup our FirstEJB running on JBoss. Notice that the argument to Context.lookup("ejb/First") is the same value we had put in the jboss.xml file to bind our FirstEJB to this name in the JNDI context. We have used that same value again to look for it. Once our lookup is successful, we cast it to our FirstHome Home interface.

```
    FirstHome home = (FirstHome)ctx.lookup("ejb/First");
```

We have then used create method of our FirstHome Home interface to get an instance of the Remote interface; First. We will now use the methods of our EJB's remote interface (First).

```
    First bean = home.create();
```

We then called the getTime() method we had created in our EJB to get the current time from JBoss Server and saved it in a temporary String object.

```
String time = bean.getTime();
```

Once we are done with our Session EJB, we use the remove method to tell the JBoss Server that we no longer need this bean instance. After this we close the external JNDI Context.

```
    bean.remove();
    ctx.close();
```

After this retrieved value from getTime() method displayed on the user screen.

3.5 RUNNING THE CLIENT/JSP PAGE

Before trying to run firstJSP.jsp page, we have to do one other thing. Copy following files from C:\JBoss\client folder and paste them in the C:\Projects\TomcatJBoss\WEB-INF\lib folder. It is a must, without it firstEJB.jsp page will not run.

```
connector.jar
deploy.jar
jaas.jar
jboss-client.jar
jboss-j2ee.jar
jbossmq-client.jar
jbosssx-client.jar
jndi.jar
jnp-client.jar
```

we will also copy the FirstEJB.jar file from C:\Projects\EJB\FirstEJB folder to the C:\Projects\TomcatJBoss\WEB-INF\lib folder. Without it the Tomcat will not be able to compile firstEJB.jsp JSP page.

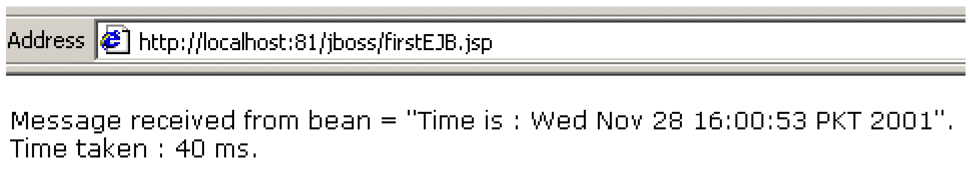
We are now ready to run our firstEJB.jsp page

Running firstEJB.jsp JSP page :

Now, start JBoss Server if, it is not already running. Also start Tomcat Server. If it is already running, then stop it and restart it again. Open your browser and access the following page :

```
http://localhost:8080/jboss/firstEJB.jsp
```

Please substitute port number “8080” above with the port number where your Tomcat Server is running. By default it is “8080” you will see a result like the following :



```
http://localhost:81/jboss/firstEJB.jsp

Message received from bean = "Time is : Wed Nov 28 16:00:53 PKT 2001".
Time taken : 40 ms.
```

Figure 1: firstEJB.jsp Client View

3.6 MESSAGE-DRIVEN BEAN

Message-driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging system, such as the Java™ API for XML Messaging (JAX-M). Message-driven beans asynchronously consume messages from a message destination, such as a JMS queue or topic.

Message-driven beans are components that receive incoming enterprise messages from a messaging provider. The primary responsibility of a message-driven bean is to process messages, because the bean's container automatically manages other aspects of the message-driven bean's environment. Message-driven beans contain business logic for handling received messages. A message-driven bean's business logic may key off the contents of the received message or may be driven by the mere fact of receiving the message. Its business logic may include such operations as initiating a step in a workflow, doing some computation, or sending a message.

A message-driven bean is essentially an application code that is invoked when a message arrives at a particular destination. With this type of messaging, an application—either a J2EE component or an external enterprise messaging client—acts as a message producer and sends a message that is delivered to a message destination. The container activates an instance of the correct type of message-driven bean from its pool of message-driven beans, and the bean instance consumes the message from the message destination. As a message-driven bean is stateless, any instance of the matching type of message-driven bean can process any message. Thus, message-driven beans are programmed in a similar manner as stateless session beans.

The advantage of message-driven beans is that they allow a loose coupling between the message producer and the message consumer, thus, reducing the dependencies between separate components. In addition, the EJB container handles the setup tasks required for asynchronous messaging, such as registering the bean as a message listener, acknowledging message delivery, handling re-deliveries in case of exceptions, and so forth. A component other than a message-driven bean would otherwise have to perform these low-level tasks.

3.7 IMPLEMENTING A MESSAGE-DRIVEN BEAN

Now, we will learn how to implement message driven beans. Message-driven beans are much like stateless session beans, having the same life cycle as stateless session beans but not having a component or home interface. The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. A message-driven bean class must also implement an ejbCreate method, even though the bean has no home interface. As they do not expose a component or home interface, clients cannot directly access message-driven beans. Like session beans, message-driven beans may be used to drive workflow processes. However, the arrival of a particular message initiates the process.

Implementing a message-driven bean is fairly straightforward. A message-driven bean extends two interfaces: javax.ejb.MessageDrivenBean and a message listener interface corresponding to the specific messaging system. (For example, when using the JMS messaging system, the bean extends the javax.jms.MessageListener interface.) The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

We can provide an empty implementation of the ejbCreate and setMessageDrivenContext methods. These methods are typically used to look up objects from the bean's JNDI environment, such as references to other beans and resource references. If the message-driven bean sends messages or receives synchronous communication from another destination, you use the ejbCreate method to look up the JMS connection factories and destinations and to create the JMS connection. The implementation of the ejbRemove method can also be left empty. However, if the ejbCreate method obtained any resources, such as a JMS connection, you should use the ejbRemove method to close those resources.

The methods of the message listener interface are the principal methods of interest to the developer. These methods contain the business logic that the bean executes upon receipt of a message. The EJB container invokes these methods defined on the message-driven bean class when a message arrives for the bean to service.

A developer decides how a message-driven bean should handle a particular message and codes this logic into the listener methods. For example, the message-driven bean might simply pass the message to another enterprise bean component via a synchronous method invocation, send the message to another message destination, or perform some business logic to handle the message itself and update a database.

A message-driven bean can be associated with configuration properties that are specific to the messaging system it uses. A developer can use the bean's XML deployment descriptor to include the property names and values that the container can use when connecting the bean with its messaging system.

3.8 JMS AND MESSAGE-DRIVEN BEANS

The EJB architecture requires the container to support message-driven beans that can receive JMS messages. You can think of message-driven beans as message listeners that consume messages from a JMS destination. A JMS destination may be a queue or a topic. When the destination is a queue, there is only one message producer, or sender, and one message consumer. When the destination is a topic, a message producer publishes messages to the topic, and any number of consumers may consume the topic's messages.

A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class. The onMessage method contains the business logic that the message-driven bean executes upon receipt of a message. The bean typically examines the message and executes the actions necessary to process it. This may include invoking other components.

The onMessage method has one parameter, the JMS message itself, and this parameter may be any valid JMS message type. The method tests whether the message is the expected type, such as a JMS TextMessage type, and then casts the message to that type and extracts from the message the information it needs.

Because the method does not include a throws clause, no application exceptions may be thrown during processing.

The EJB architecture defines several configuration properties for JMS-based message-driven beans. These properties allow the container to appropriately configure the bean and link it to the JMS message provider during deployment. These properties include the following:

- **DestinationType:** Either a javax.jms.Queue if the bean is to receive messages from a JMS queue, or a javax.jms.Topic if the bean is to receive messages from a JMS topic.
- **SubscriptionDurability:** Used for JMS topics to indicate whether the bean is to receive messages from a durable or a nondurable subscription. A durable subscription has the advantage of receiving a message even if the EJB server is temporarily offline.
- **AcknowledgeMode:** When message-driven beans use bean-managed transactions, this property indicates to the container the manner in which the delivery of JMS messages is to be acknowledged. (When beans use container-managed transactions, the message is acknowledged when the transaction commits.) Its values may be Auto-acknowledge, which is the default, or Dups-ok-acknowledge. The Auto-acknowledge mode indicates that the container should acknowledge messages as soon as the onMessage method returns. The Dups-ok-acknowledge mode indicates that the

container may lazily acknowledge messages, which could cause duplicate messages to be delivered.

- **MessageSelector**— Allows a developer to provide a JMS message selector expression to filter messages in the queue or topic. Using a message selector expression ensures that only messages that satisfy the selector are delivered to the bean.

3.9 MESSAGE-DRIVEN BEAN AND TRANSACTIONS

As messaging systems often have full-fledged transactional capabilities, message consumption can be grouped into a single transaction with such other transactional work as database access. This means that a bean developer can choose to make a message-driven bean invocation part of a transaction. Keep in mind, however, that if the message-driven bean participates in a transaction, you must also be using container-managed transaction demarcation. The deployment descriptor transaction attribute, which for a message-driven bean can be either Required or NotSupported, determines whether the bean participates in a transaction.

When a message-driven bean's transaction attribute is set to Required, the message delivery from the message destination to the message-driven bean is part of the subsequent transactional work undertaken by the bean. By having the message-driven bean be part of a transaction, you ensure that message delivery takes place. If the subsequent transaction fails, the message delivery is rolled back along with the other transactional work. The message remains available in the message destination until picked up by another message-driven bean instance. Note that, the message sender and the message receiver, which is the message-driven bean, do not share the same transaction. Thus, the sender and the receiver communicate in a loosely coupled but reliable manner.

If the message-driven bean's transactional attribute is NotSupported, it consumes the message outside of any subsequent transactional work. Should that transaction not complete, the message is still considered consumed and will be lost.

It is also possible to use bean-managed transaction demarcation with a message-driven bean. With bean-managed transaction demarcation, however, the message delivery is not be part of the transaction, because the transaction starts within the onMessage method.

3.10 MESSAGE-DRIVEN BEAN USAGE

Bean developers should consider using message-driven beans under certain circumstances:

- To have messages automatically delivered.
- To implement asynchronous messaging.
- To integrate applications in a loosely coupled but reliable manner.
- To have message delivery drive other events in the system workflow.
- To create message selectors, whereby specific messages serve as triggers for subsequent actions.

3.11 EXAMPLE OF MESSAGE-DRIVEN BEAN

Now, we will learn how to write a message driven bean. Consider,PayrollMDB, a message-driven bean that follows the requirements of the EJB 2.1 architecture, consisting of the PayrollMDB class and associated deployment descriptors. The following Code shows the complete code for the PayrollMDB implementation class:

```
Public class PayrollMDB implements MessageDrivenBean,
        MessageListener {

    private PayrollLocal payroll;

    public void setMessageDrivenContext(MessageDrivenContext mdc) {
        try {
            InitialContext ictx = new InitialContext();
            PayrollLocalHome payrollHome = (PayrollLocalHome)
                ictx.lookup("java:comp/env/ejb/PayrollEJB");
            payroll = payrollHome.create();
        } catch ( Exception ex ) {
            throw new EJBException("Unable to get Payroll bean", ex);
        }
    }

    public void ejbCreate() { }

    public void ejbRemove() { }

    public void onMessage(Message msg) {
        MapMessage map = (MapMessage)msg;
        try {
            int emplNumber = map.getInt("Employee");
            double deduction = map.getDouble("PayrollDeduction");

            payroll.setBenefitsDeduction(emplNumber, deduction);
        } catch ( Exception ex ) {
            throw new EJBException(ex);
        }
    }
}
```

The PayrollMDB class implements two interfaces: javax.ejb.MessageDrivenBean and javax.jms.MessageListener. The JMS MessageListener interface allows the bean to receive JMS messages with the onMessage method. The MessageDrivenBean interface defines a message-driven bean's life-cycle methods called for, by the EJB container.

Of three such life-cycle methods, only one is of interest to PayrollMDB—setMessageDrivenContext. The container calls for setMessageDrivenContext immediately after the PayrollMDB bean instance is created. PayrollMDB uses this method to obtain a local reference to the Payroll stateless session bean, first looking up the PayrollLocalHome local home object and then invoking its create method. The setMessageDrivenContext method then stores the local reference to the stateless bean in the instance variable payroll for later use in the onMessage method.

The ejbCreate and ejbRemove life-cycle methods are empty. They can be used for any initialisation and cleanup that the bean needs to do.

The real work of the PayrollMDB bean is done in the onMessage method. The container calls the onMessage method when a JMS message is received in the PayrollQueue queue. The msg parameter is a JMS message that contains the message sent by the Benefits Enrollment application or other enterprise application. The method typecasts the received message to a JMS MapMessage message type, which is a special JMS message type that contains property-value pairs and is particularly useful when receiving messages sent by non-Java applications. The EJB container's JMS provider may convert a message of MapMessage type either from or to a messaging product–specific format.

Once the message is in the proper type or format, the onMessage method retrieves the message data: the employee number and payroll deduction amount, using the Employee and PayrollDeduction properties, respectively. The method then invokes the local business method setBenefitsDeduction on the Payroll stateless session bean method to perform the update of the employee’s payroll information in PayrollDatabase.

The PayrollMDB bean's deployment descriptor declares its transaction attribute as Required, indicating that the container starts a transaction before invoking the onMessage method and to make the message delivery part of the transaction. This ensures that the Payroll stateless session bean performs its database update as part of the same transaction and that message delivery and database update are atomic. If, an exception occurs, the transaction is rolled back, and the message will be delivered again. By using the Required transaction attribute for the message-driven bean, the developer can be confident that the database update will eventually take place.

PayrollEJB Local Interfaces

In the PayrollMDB means we have noticed that it uses the local interfaces of the PayrollEJB stateless session bean. These interfaces provide the same functionality as the remote interfaces described earlier. Since, these interfaces are local, they can be since accessed only by local clients deployed in the same JVM as the PayrollEJB bean. As a result, the PayrollMDB bean can use these local interfaces because it is deployed together with the PayrollEJB bean in the payroll department's application server. The following is the code for the local interfaces :

```
public interface PayrollLocal extends EJBLocalObject {
   void setBenefitsDeduction(int emplNumber, double deduction)
      throws PayrollException;
   double getBenefitsDeduction(int emplNumber)
      throws PayrollException;
   double getSalary(int emplNumber)
      throws PayrollException;
   void setSalary(int emplNumber, double salary)
      throws PayrollException;
}

public interface PayrollLocalHome extends EJBLocalHome {
   PayrollLocal create() throws CreateException;

}
```

Check Your Progress 1

1) State True/ False

   a) By default Home interface must contain at least one create() method. T ☐ F ☐

   b) Message-driven beans always synchronously consume messages from a message destination, such as a JMS queue. T ☐ F ☐

   c) The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. T ☐ F ☐

d) A message-driven bean that consumes JMS messages is not always required to implement the javax.jms.MessageListener interface. T ☐ F ☐

2) Explain two interfaces associated with EJB class files.

…………………………………………………………………………………..…

….……………………………………………………………………………..…

…………………………………………………………………………………

3) Explain the Message Driven bean and its advantages.

…………………………………………………………………………………..…

….……………………………………………………………………………..…

…………………………………………………………………………………

4) Explain the methods required to implement the Message Driven Bean.

…………………………………………………………………………………..…

….……………………………………………………………………………..…

…………………………………………………………………………………

5) How does Message Driven Bean consumes the messages from the JMS?

…………………………………………………………………………………..…

….……………………………………………………………………………..…

…………………………………………………………………………………

6) Explain the various configuration properties of JMS based Message Driven Beans offered by EJB architecture.

…………………………………………………………………………………..…

….……………………………………………………………………………..…

…………………………………………………………………………………

7) Explain the various circumstances under which Message Driven Bean can be used ?

…………………………………………………………………………………..…

….……………………………………………………………………………..…

…………………………………………………………………………………

3.12 SUMMARY

In this unit we have learnt what comprises an Enterprise JavaBean and how to develop and deploy an EJB. We created an EJB, deployed it on JBoss Server and called it from a JSP page running on Tomcat Server in a separate process. EJBs are collection of Java classes, interfaces and XML files adhering to given rules. In J2EE all the components run inside their own containers. EJBs run inside EJB container. The container provides certain built-in services to EJBs which the EJBs use to function.

Every EJB class file has two accompanying interfaces and one XML file. The two interfaces are Remote Interfaces and home interface. Remote interface contains the methods that developer wants to expose to the clients. Home interface is actually EJB builder and should contain methods used to create Remote interfaces for the EJB. By default Home interface must contain at least one create() method. The actual implementation remains hidden from the clients.

Every Remote interface must always extend EJBObject interface. It is a requirement, not an option. Message driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging system, such as the Java API for XML Messaging (JAX-M). Message-driven beans are components that receive incoming enterprise messages from a messaging provider. Message-driven beans contain business logic for handling received messages. A message-driven bean’s business logic may key off the contents of the received message or may be driven by the mere fact of receiving the message. Its business logic may include such operations as initiating a step in a workflow, doing some computation, or sending a message. The advantage of message-driven beans is that they allow a loose coupling between the message producer and the message consumer, thus reducing the dependencies between separate components.

Message-driven beans are much like stateless session beans, having the same life cycle as stateless session beans but not having a component or home interface. The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. A message-driven bean class must also implement an ejbCreate method, even though the bean has no home interface. The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

The EJB architecture requires the container to support message-driven beans that can receive JMS messages. Message-driven beans act as message listeners that consume messages from a JMS destination. A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class.

Messaging systems often have full-fledged transactional capabilities, message consumption can be grouped into a single transaction with such other transactional work as database access. This means that a bean developer can choose to make a message-driven bean invocation part of a transaction.

Message driven can be used under the various circumstances like when messages to automatically delivered, to implement asynchronous messaging and to create message selectors.

3.13 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) True/ False

   a) True
   b) False
   c) True
   d) False

Explanatory Answers

2) There are two interfaces associated with every EJB class file :

   Remote Interfaces: Remote interface is what the client gets to work with, in other words Remote interface should contain the methods you want to expose to your clients.

   Home Interfaces: Home interface is actually the EJB builder and should contain methods used to create Remote interfaces for your EJB. By default, Home interface must contain at least one create() method.

The actual implementation of these interfaces, our Session EJB class file remains hidden from the clients.

3) Message-driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging system, such as the Java API for XML Messaging (JAX-M). The primary responsibility of a message-driven bean is to process messages, because the bean's container automatically manages other aspects of the message-driven bean's environment. Message-driven beans contain business logic for handling received messages.

A message-driven bean is essentially an application code that is invoked when a message arrives at a particular destination. With this type of messaging, an application—either a J2EE component or an external enterprise messaging client—acts as a message producer and sends a message that is delivered to a message destination.

Advantages

The advantage of message-driven beans is that they allow loose coupling between the message producer and the message consumer, thus, reducing the dependencies between separate components. In addition, the EJB container handles the setup tasks required for asynchronous messaging, such as registering the bean as a message listener, acknowledging message delivery, handling re-deliveries in case of exceptions, and so forth. A component other than a message-driven bean would otherwise have to perform these low-level tasks.

4) The message-driven bean must implements two interfaces: javax.ejb.MessageDrivenBean and a message listener interface corresponding to the specific messaging system. (For example, when using the JMS messaging system, the bean extends the javax.jms.MessageListener interface.) The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

There can be empty implementation of the ejbCreate and setMessageDrivenContext methods. These methods are typically used to look up objects from the bean's JNDI environment, such as references to other beans and resource references.

5) Message-driven beans act as message listeners that consume messages from a JMS destination. A JMS destination may be a queue or a topic. A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class. The onMessage method contains the business logic that the message-driven bean executes upon receipt of a message. The bean typically examines the message and executes the actions necessary to process it. This may include invoking other components.

6) The onMessage method has one parameter, the JMS message itself, and this parameter may be any valid JMS message type. The method tests whether the message is the expected type, such as a JMS TextMessage type, and then casts the message to that type and extracts from the message the information it needs.

7) The EJB architecture defines several configuration properties for JMS-based message-driven beans, which are described as the following:

DestinationType: Either a javax.jms.Queue if the bean is to receive messages from a JMS queue, or a javax.jms.Topic if the bean is to receive messages from a JMS topic.

SubscriptionDurability: It is used for JMS topics to indicate whether the bean is to receive messages from a durable or a nondurable subscription. A durable subscription has the advantage of receiving a message even if the EJB server is temporarily offline.

AcknowledgeMode: When message-driven beans use bean-managed transactions, this property indicates to the container the process of acknowledging the delivery of JMS messages. (When beans use container-managed transactions, the message is acknowledged once the transaction commits.) Its values may be Auto-acknowledge, which is the default, or Dups-ok-acknowledge.

MessageSelector: It allows a developer to provide a JMS message selector expression to filter messages in the queue or topic. Using a message selector expression ensures that only messages that satisfy the selector are delivered to the bean.

Bean developers may consider using message-driven beans under certain circumstances:

- To have messages automatically delivered.
- To implement asynchronous messaging.
- To integrate applications in a loosely coupled but reliable manner.
- To have message delivery drive other events in the system workflow.
- To create message selectors, whereby specific messages serve as triggers for subsequent actions.

3.14 FURTHER READINGS/REFERENCES

- Justin Couch and Daniel H. Steinberg, *Java 2 Enterprise Edition Bible,* Hungry Minds, Inc.

- Paco Gomez and Peter Zadronzy, *Professional Java 2 Enterprise Edition with BEA Weblogic Server,* WROX Press Ltd

- Budi Kurniawan, *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions,* New Riders Publishing

- Richard Monson-Haefel, *Enterprise JavaBeans (3rd Edition),* O'Reilly

- Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, Beth Stearns, *Applying Enterprise JavaBeans™:Component-Based Development for the J2EE™ Platform,* Second Edition, Pearson Education

- Robert Englander, *Developing Java Beans,* O'Reilly Media

Reference websites:

- www.javaworld.com
- www.j2eeolympus.com
- www.samspublishing.com
- www.oreilly.com
- www.stardeveloper.com
- www.roseindia.net
- www.e-docs.bea.com
- www.java.sun.com