JMS en aplicaciones reales (2)

Índice

1. JMS con MDBs, EJBs y XML

En esta sesión vamos a presentar una aplicación ejemplo basada en EJBs que hace uso de mensajes. Concretamente se trata de que:

1. Un cliente construye y envía una petición a una cola.

2. La petición es recogida (modo asíncrono) por un bean de mensajes.

3. El bean invoca (síncrono) a un EJB de sesión para realizar la petición del cliente.

4. El EJB devuelve el resultado al bean de mensajes.

5. El bean empaqueta el resultado en un mensaje y lo envía a otra cola.

6. Un segundo cliente recoge la respuesta.

2. ClientSend produce y envía mensaje XML

El primer paso consiste en construir un cliente que construya y envíe un mensaje con la información necesaria para el EJB. Dicha información se pedirá en línea al usuario y se irá construyendo un mensaje XML. El código fuente del ejemplo se encuentra en ClientSend.java

En el método principal de esta clase se va construyendo un mensaje como el siguiente

`<stocktrade action=buy symbol=BEAS numShares=10 \>`

sobre un `String` llamado `buffer`. Todo ello se hace dentro de un bucle `do` hasta poder enviar el mensaje o bien recibir la opción *quit*.

```
do {
      buffer.append("<stocktrade action=");
      transactionSent = false;

      //get action from user
      System.out.print("Action (\"Buy\" or \"Sell\", \"quit\" to
quit): \n");
      line = msgStream.readLine();
      if (line != null && line.trim().length() != 0) {
        quitNow = line.equalsIgnoreCase("quit");
        buy = line.equalsIgnoreCase("buy");
        sell = line.equalsIgnoreCase("sell");
        buffer.append("\""+(buy ? "buy" : "sell")+"\"");
        if ((buy || sell) && !quitNow) {
```

```
          buffer.append(" symbol=");
          do {

            //get stock symbol from user
            System.out.print("Symbol (\"BEAS\" or \"MSFT\", \"quit\"
to quit): \n");
            line = msgStream.readLine();
            if (line != null && line.trim().length() != 0) {
              quitNow = line.equalsIgnoreCase("quit");
              beas = line.equalsIgnoreCase("BEAS");
              msft = line.equalsIgnoreCase("MSFT");

              if (beas == true) buffer.append("\""+ "BEAS" + "\"");
              if (msft == true) buffer.append("\""+ "MSFT" + "\"");

              if ((beas || msft) && !quitNow) {
                 do {
                   // get number of shares from user
                   System.out.print("Number of shares (\"quit\" to
quit): \n");
                   buffer.append(" numShares=");
                   line = msgStream.readLine();
                   if (line != null && line.trim().length() != 0) {
                     quitNow = line.equalsIgnoreCase("quit");
                     buffer.append("\""+line+"\"");
                     buffer.append("/>");
...
```

Si finalmente se envia el mensaje se llamará al método `send()`

```
System.out.println("Sending trade information...");
// send message to JMS queue
client.send(buffer.toString());
buffer.delete(0,buffer.length());
System.out.println("Your message has been sent to queue");
transactionSent = true;
```

donde en dicho método hay que destacar que la variable `xmsg` es un objeto `weblogic.jms.extensions.XMLMessage` lo cual especifica contenido XML y esto facilita el filtrado de mensajes que es más complejo en este caso.

```
public void send(String message) throws JMSException {
    xmsg.setText(message);
    qsender.send(xmsg);
   }
```

El objeto `xmsg` se creó en el método `init()`, donde típicamente inicializamos el contexto:

```
xmsg = ((WLQueueSession)qsession).createXMLMessage();
```

siendo la clase `weblogic.jms.extensions.WLQueueSession` una extensión de la clase `Session` aportando nuevos campos no soportados.

Por otro lado, la cola a la que se envía este mensaje es `weblogic.examples.jms.messageformat.exampleQueueSend`.

3. MessageTraderBean recoge el mensaje XML y lo parsea

Un EJB de mensajes (Message Driven Bean o MDB) es una sublclase de `weblogic.ejb.GenericMessageDrivenBean` que implementa la interfaz `MessageListener` (ver el código completo en MessageTraderBean.java). Toda la actividad realizada por este bean se codifica el método `OnMessage()`. Como siempre lo primero es recibir el mensaje XML de la cola. Pero además hay que parsearlo antes de invocar al EJB que realizará la acción especificada en el mensaje. El código de la primera parte de este método, donde se lee el mensaje y se invoca al parser

```
public void onMessage(Message msg) {
   log("MessageTraderBean.onMessage(msg)");
   xmlmessage = (XMLMessage) msg;
   RequestHandler rh = new RequestHandler();
   try {
     String txt = xmlmessage.getText();

     SAXParserFactory fact = SAXParserFactory.newInstance();
     SAXParser sp = fact.newSAXParser();

     InputSource inSource = new InputSource();
     inSource.setCharacterStream(new StringReader(txt));
     sp.parse(inSource, rh);
   } catch(JMSException ex) {
     System.err.println("An exception occurred: "+ex.getMessage());
   } catch (Exception e) {
     System.err.println("An exception occurred: "+e.getMessage());
     return;
   }
```

3.1. Parsing XML: definición del RequestHandler

Antes de realizar el parsing se especifica un manejador de eventos. Esto se hace en la clase `RequestHandler` que extiende `org.xml.sax.helpers.DefaultHandler`. El código completo del manejador de eventos se encuentra en MessageTraderBean.java.

En términos generales dicho manejador tiene un constructor (vacío en este caso) y una serie

de métodos asociados a los eventos detectables durante el proceso de parsing.

1. El primer evento a detectar es el comienzo del documento (o mensaje, como en este caso) XML:

```
public void startDocument() throws SAXException {
    nl();
    nl();
    trade = new Hashtable<String,String>();
    log("START DOCUMENT");
    nl();
    log("<?xml version='1.0' encoding='UTF-8'?>");
  }
```

donde `nl()` comienza una nueva linea y la identa de forma adecuada. Los métodos del manejador van a ir construyendo una `Hashtable` denominada `trade` que se devolverá finalmente como resultado.El método `log()` imprime en pantalla

2.Una vez se ha detectado el comienzo del documento XML, se notificará un evento cada vez que comience un nuevo elemento

```
public void startElement(String uri, String localName, String qName,
                         Attributes attrs)
    throws SAXException
  {
    indentLevel++;
    nl();
    log("ELEMENT: " +qName);
            if (attrs != null) {
        for (int i = 0; i < attrs.getLength (); i++) {
          nl();
          trade.put(attrs.getQName(i), attrs.getValue(i));
          log("   ATTR: " +attrs.getQName(i)+" = "+attrs.getValue(i));
        }
      }
   }
```

Esta notificación procede del parser y la información sobre el elemento se encuentra en el objeto de la clase `Attributtes`. Antes se incrementa la identación.

3. La notificación del final de un elemento solo tiene como efecto el decremento de la identación.

```
public void endElement(String name) throws SAXException {
     nl();
     log ("END_ELM: " + name);
     indentLevel--;
   }
```

4.Una vez comenzado un elemento, el método `characters()` lee su contenido.

```
public void characters (char buf [], int offset, int len)
    throws SAXException
  {
    nl();
    String s = new String(buf, offset, len);
    if (!s.trim().equals("")) log ("CHARS:   " + s);
  }
```

5. Finalmente el método `endDocument()` notifica el final del documento XLM.

```
public void endDocument() throws SAXException {
    nl();
    log("END DOCUMENT");
    try {
      nl();
    } catch (Exception e) {
      throw new SAXException ("I/O error", e);
    }
```

y el método `getData()` devuelve la tabla hash con el contenido parseado:

```
public Hashtable<String,String> getData() { return trade;}
```

3.2. Parsing XML con SAX

El proceso de parsing en sí viene dirigido por un objeto de la clase `javax.xml.parsers.SAXParser`. En `onMessage()` tenemos:

```
try {
      String txt = xmlmessage.getText();

      SAXParserFactory fact = SAXParserFactory.newInstance();
      SAXParser sp = fact.newSAXParser();

      InputSource inSource = new InputSource();
      inSource.setCharacterStream(new StringReader(txt));
      sp.parse(inSource, rh);
    } catch(JMSException ex) {
      System.err.println("An exception occurred: "+ex.getMessage());
    } catch (Exception e) {
      System.err.println("An exception occurred: "+e.getMessage());
      return;
    }
```

con la llamada al método `parse()` que es el que va a generar los eventos que analizará el objeto `RequestHandler` llamado `rh`.

4. Invocación a TraderBean

Seguimos dentro del método `onMessage()`, ya que el resultado del parsing se ha recogido en una tabla hash invocando al manejador:

```
java.util.Hashtable<String,String> trade = rh.getData();
```

A partir de aquí el proceso sigue los pasos siguientes:

1. Crear un EJB `Trader`llamado `trader`.

2. Construir un objeto `TradeResult` llamado `tr`.

3. Pasarle a `tr` los datos de la operación consultando la tabla hash

4. Invocar el método pertinente del EJB.

Estos pasos se realizar en el siguiente fragmento de código:

```
try {
      // Create a trader object.
      Context ctxt          =  new InitialContext();
      TraderHome brokerage = (TraderHome)
ctxt.lookup(TRADER_EJB_JNDI);

      // Give this trader a name.
      Trader trader = brokerage.create();

      // Do trade.
      if (((String) trade.get("action")).equals("buy")) {
        TradeResult tr = trader.buy("Erin", (String)
trade.get("symbol"),
                          Integer.parseInt((String)
trade.get("numShares")));
....
```

Así pues, el control se cede al EJB cuyo código completo se muestra en TraderBean.java. El objeto `TraderBean` extiende la clase `GenericSessionBean`Una de las acciones a realizar puede ser por ejemplo comprar, soportada por el método `buy()`:

```
@RemoteMethod()
  public TradeResult buy(String customerName, String stockSymbol, int
shares)
```

```
    throws ProcessingErrorException
  {
    log("buy ("+customerName+", "+stockSymbol+", "+shares+")");

    double price = getStockPrice(stockSymbol);
    tradingBalance -= (shares * price); // subtract purchases from
cash account

    return new TradeResult(shares, price, TradeResult.BUY);
  }
```

5. MessageTraderBean recibe el resultado y envía un mensaje XML

Siguiendo en el método `onMessage()` del bean de mensajes, el EJB le ha devuelto el resultado de la operación en el objeto `TradeResult` llamado `tr`. Así pues, se construye un nuevo mensaje sobre la cadena `buffer`. Este mensaje XML será del tipo:

```
<traderesult action=bought symbol=BEAS numShares=10 price=2.0
changeInAccount=2000.0\>
```

Siendo el fragmento de código:

```
buffer.append("<traderesult action='bought' "+
            "symbol='"+(String) trade.get("symbol")+
            "' numShares='"+ tr.getNumberTraded() +
            "' price ='"+ tr.getPrice() +"' "+
            "changeInAccount ='"+trader.getBalance() +"'/>");
```

Finalmente, todavía dentro de `onMessage()` se pasa a `String` la variable `buffer` y se invoca el método `send()`:

```
Context ctx = getInitialContext();
       init(ctx, QUEUE);
       send(buffer.toString());
```

Siendo `QUEUE="java:comp/env/jms/exampleQueueReceive"` cuyo nombre JNDI es precisamente `weblogic.examples.jms.messageformat.exampleQueueReceive`. La correspondencia entre los nombres JNDI y los recursos a los que accede el EJB se especifica, en EJB 3.0, en la cabecera del mismo, junto con otros elementos:

```
@JndiName(remote = "jms.Messageformat")
@MessageDriven(maxBeansInFreePool = "1000",
               destinationType = "javax.jms.Queue",
               initialBeansInFreePool = "0",
```

```
                transTimeoutSeconds = "600",
                defaultTransaction =
MessageDriven.DefaultTransaction.REQUIRED,
                durable = Constants.Bool.FALSE,
                ejbName = "jmsMessageformat",
                destinationJndiName =
"weblogic.examples.jms.messageformat.exampleQueueSend")
@ResourceEnvRefs({
    @ResourceEnvRef(name = "jms/exampleQueueReceive",
                    type = "javax.jms.Queue",
                    jndiName =
"weblogic.examples.jms.messageformat.exampleQueueReceive")
})
@ResourceRefs({
    @ResourceRef(auth = ResourceRef.Auth.CONTAINER,
                 jndiName =
"weblogic.examples.jms.messageformat.QueueConnectionFactory",
                 name = "jms/QCF",
                 type =
"weblogic.examples.jms.messageformat.QueueConnectionFactory",
                 sharingScope = ResourceRef.SharingScope.SHAREABLE)
})
```

6. ClientReceive consulta el resultado en otra cola

Un segundo cliente, cuyo código se puede consultar en ClientReceive.java, se conecta a la cola a la que se envió el resultado de la operación y en su método `receive()` realiza el parsing y muestra el contenido final del mensaje; algo así como:

```
START DOCUMENT

?xml version='1.0' encoding='UTF-8'?

    ELEMENT: traderesult

       ATTR: action = bought

       ATTR: symbol = BEAS

       ATTR: numShares = 10

       ATTR: price = 2.0

       ATTR: changeInAccount = 2000.0

    END_ELM: traderesult

END DOCUMENT
```

7. Despliegue y ejecución del ejemplo

Las instrucciones para el despliegue de todos los ejemplos ya se han referenciado con anterioridad. En este caso, y desplegando en el menú izquierdo la pestaña correspondiente a *Messaging* podemos acceder a todos los ejemplos de JMS. Concretamente seleccionaremos *Using JMS and Message-Driven EJB to Transmit XML*.

En este caso el despliegue es algo más complejo que en los casos anterioroes ya que intervienen EJBs. Se contemplan los siguientes pasos:

1. Establecer las variables de entorno.

2. Ejecutar `ant build` para compilar.

3. Ejecutar `ant deploy`

4. Invocar la consola de administración

5. En la consola, activar el modo de hacer cambios en la configuración haciendo click en *Lock & Make Changes*

6. Expandir en el panel izquierdo el nodo **Deployments**.

7. Expandir **jmsMsgFormatEar** en la tabla de deployments.

8. Seleccionar **JMSMessageFormatJMS** y consultar los recursos JMS asociados.

9.Seleccionar cada uno de los recursos y revisar su configuración. En el tab de **Targets** comprobad que el SubDeployment seleccionado es **examplesJMSServer** y el servidor JMS es **examplesJMSServer**. Seleccionar update y click en **Save**.

10. En el *Change Center* click en *Activate Changes*

En cuanto a la ejecución del ejemplo se puede lanzar el cliente que construye y envía el mensaje XML con `ant run.send`. Desde otro shell se puede lanzar el cliente que va a recibir y mostrar el resultado con `ant run.receive`.