Message Queuing in Java (simple version)

This tutorial will show you how to develop a Java Web application that communicates with a back-end component using Java Message Service (JMS) – a standard Java API for message queuing. The tutorial relies on NetBeans and the Glassfish application server deployed on the ATS server. You can probably also complete the tutorial using another IDE (e.g. Eclipse), but the instructions below are given for NetBeans.
The tutorial is to a large extent based on a tutorial by Masoud Kalali:
http://weblogs.java.net/blog/kalali/archive/2006/05/step_by_step_to.html

Part A – Creating the JMS queue

First of all, you will need to create a JMS queue on the Glassfish server. This can be done using the administration console of the Glassfish server (the login details were given in the practical session in week 2). You need to create two things: a JMS "Connection Factory" to manage the JMS connections (i.e. sending messages), and a JMS "Destination Resource" (i.e. the queue itself).

- To create the connection factory, expand the menu items "Resources", "JMS Resources", "Connection Factories", then add a new connection factory with the following parameters:
  - JNDI Name: jms/yourLogin_ConnectionFactory
  - Type: javax.jmsConnectionFactory
  - Description: some description here
- To create the JMS destination resources, go to the menu item Destination Resources, and create a new destination resource with the following parameters:
  - JNDI Name: jms/yourLogin_Queue
  - Physical destination: yourLogin_Queue
  - Type: javax.JMS.Queue
  - Description: some description here

Now you have a JMS configuration ready to serve Message-Driven Beans (MDBs).

Part B – Creating the Message-Driven Bean and the Web Application

A message-driven bean is a type of Enterprise Java Bean (EJB) that reacts to incoming events, which are typically messages arriving over a JMS queue.
We want to create a Web application that communicates with an MDB. Rather than creating two separate projects, it is more convenient that you create a single project of type "Java enterprise application". A Java Enterprise Application has by default one EJB module and one web module. When creating your project, please give it a name like yourLogin_JMS so that you do not have name clashes with other students.

- Create an MDB by going to the project view, right-clicking on the EJB module and the option to create a new MDB, with the following parameters:
  - EJB Name : yourLogin_MDB
  - package : mdbs
  - Mapped Name : jms/yourLogin_Queue
- The previous step creates the skeleton code for an MDB. This MDB is configured to react to messages on the JMS queue you previously created. What you have to do now is to implement the onMessage(...) method. Below we give the skeleton of the code. You need to do include a few packages (e.g. javax.jms.* and others) to make it complete.

```
@Resource
 private MessageDrivenContext mdc;
```

```
   public void onMessage(Message message) {

      TextMessage msg = null;
      try {
         if (message instanceof TextMessage) {
            msg = (TextMessage) message;
            System.out.println("A Message received in TMDB: " +
                  msg.getText());
         } else {
           System.out.println("Message of wrong type: " +
                  message.getClass().getName());

         }
      } catch (JMSException e) {
         e.printStackTrace();
         mdc.setRollbackOnly();
      } catch (Throwable te) {
         te.printStackTrace();
      }
   }
```

- Let's now implement a Web application that prompts the user for a text message. Expand the web application node in your prject, and double-click on index.jsp. Change the content of the JSP to something like the following:

```
<html>
   <head>

      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>JSP Page</title>

   </head>
   <body>
<center>
<form action="sendMessage">
         <table  cellspacing="20" >

            <tbody>
               <tr>
                  <td>Enter some message: </td>
                  <td><input type="text" name="message" value="Enter your message here"
width="30" /></td>

               </tr>
            </tbody>
         </table>
            <input type="submit" value="Send the message" name="send" />
         </center>
      </form>
   </body>
</html>
```

- We now need to implement a servlet that will handle the data submitted through this simple form. This servlet will then act as the client of the MDB by forwarding the received text message to the MDB created earlier. To do this, right-click on Web application node and select New -> Servlet. Change its Name to sendMessage (the action associated to the form you previously developed) and give it your login as a package name, then click finish.
- Change the processRequest(..) method body to the following one (you will need to include the packages javax.jms.* plus a some Java naming packages – NetBeans will give you hints as to which packages you need to include).

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        //start the JMS connection using the connection factory created earlier
        try{
            Context ctx = new InitialContext();
            ConnectionFactory     connectionFactory =
(ConnectionFactory)ctx.lookup("jms/tConnectionFactory");
            Queue     queue = (Queue)ctx.lookup("jms/tQueue");
            javax.jms.Connection  connection = connectionFactory.createConnection();
            javax.jms.Session        session =
connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer = session.createProducer(queue);
            TextMessage message = session.createTextMessage();
            message.setText(request.getParameter("message"));
            System.out.println( "It come from Servlet:"+ message.getText());
            messageProducer.send(message);

            //message sent
            //show what we have done in this servlet
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet sendMessage</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<center>");
            out.print("Servlet Send this message <h2>"+request.getParameter("message") + "</h2>  to
this Queue : <h2>"+queue.getQueueName()+"</h2>");
            out.println("</center>");
            out.println("</body>");
            out.println("</html>");

        } catch(Exception ex){
            ex.printStackTrace();
        }
        out.close();
    }
```

- You should now be able to build, deploy and test your project. You can do this in two ways. You can build on NetBeans, and find the ".ear" file produced by NetBeans (note: the EAR file extension stands for Enterprise Archive). Then upload this file to Enteprise Application to Glassfish through the administration console (look for Applications → Enterprise Applications and add a new entry to the list of Enterprise Applications). You can find the web application at the following URL: URL_OF_ ATS_SERVER:portNumber/yourLogin_JMS-war/

(where "yourLogin_JMS" should be replaced by the name you have to your Java Enterprise Application project upon creation).

- Another option is to register the Glassfish server that is running in ATS with the NetBeans server manager. For this, you should go to the tab "services" on NetBeans, right-click on "servers", then add a new server, and follow the screens, making sure that you select the option "register remote server". At some point you will have to enter the URL and login details of the Glassfish instance on the ATS server. Then you can go to the Project tab on NetBeans, right-click on the name of your Java Enterprise Application project, select "Properties", then select the "Run" category, then change the change the "Server" property to the remote Glassfish server (probably this will be the one at the bottom of the list among the available servers).
- When you access the JSP you created, you can enter any text message. Your text message should then appear in the logs of the Glassfish server. To see the logs of the Glassfish server go to the admin console, then select "Common Tasks" and "search the logs".

Part C – Creating a chain of processing nodes

In the previous part, a servlet receives the input text from the user and puts it into a message queue which is connected with a message-driven bean. Supposedly, this MDB will do some processing on this request and possibly pass it on to another processing node if further processing is required.

- Create a second JMS message queue and a second MDB that consumes messages from this queue and prints them out (in the same way as the first MDB you implemented above).
- Modify the first message-driven bean so that after reading a message from the first queue and printing it out, it put this message into the second queue so that the message is then read by the second queue. Note: you can reuse the same connection factory that you previously created, there is no need her e to create a second connection factory.
- Warning: At any cost, please avoid that the first MDB puts a message into the first queue. Why? Because this means that the first MDB will be reading a message from the first queue and putting it back into this same queue, creating an infinite processing loop!

Part D – If you want to go further

Try to modufy JSP and the servlet so that the user can input (in an HTML form) all the data fields required to produce an xCBL puchase order (see notes of the practical session of Week 4). When the data is submitted to the "sendMessage" servlet, the servlet will produce an xCBL purchase order and send it to the first MDB using the first queue that you created in Part B. The first MDB will then read the purchase order in xCBL format and transform it into a purchase order in RosettaNet format using the XSL developed in Week 4. The first MDB will then put the purchase order in RosettaNet format into the second message queue so that it is later consumed by the second MDB.