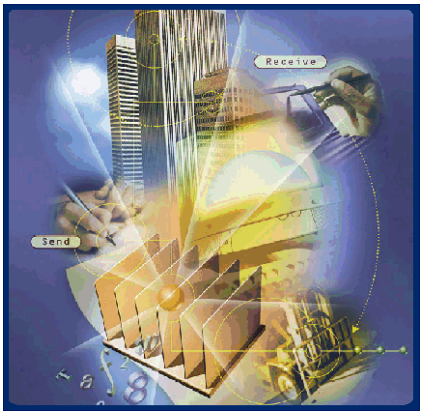
JMS integration into the J2EE platform



This session will address how to integrate a JMS implementation into a web based architecture. We will explore, via coded examples, how to extract information from a Servlet request and publish it to listening clients. The example will also address how MDB's (Message Driven Beans) work, Clustering, DRA (Dynamic Routing Architecture), message filtering and business logic implementation.

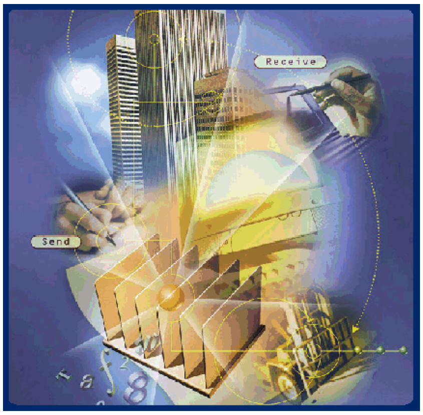
Topics Covered

- A 'Basic' JMS implementation
- The Web and Asynchronous Services
- Message Driven Beans
- Clustering, Failover and Dynamic Routing Architecture

About the Speaker – Andreas Taber

- Founded the Hartmann Software Group
- Was an evangalist, consultant and systems engineer for Sonic Software
- Reseller and service provider of SwiftMQ
- Faculty member at the University of Colorado and the University of Denver

A ‘Basic’ JMS implementation



Let’s get started…

Considerations

- Select a JMS provider … using SwiftMQ
- Set up CLASSPATH to include jms.jar
- Import jms.jar in your java folder
- Connect to server by way of JNDI on either Topics or Queues
- Create a Session
- Obtain a message
- Send or receive the message

…. Let’s look at an example of a queue listener

Basic Example Code - Sender

```
import javax.jms.*;
import javax.naming.*;
import java.util.*;

public class JMSExample
{
      public static void main(String[] args)
      {
            String JMS_URL = “smqp://localhost:4001”;
            String Connection_Name=“plainsocket@router1”;
            String Queue_Name = "queue1@router1";
            try {

         // Perform the JNDI lookup.
          Hashtable env = new Hashtable();

            env.put(Context.INITIAL_CONTEXT_FACTORY,"com.swiftmq.jndi.InitialContextFactoryImpl");
            env.put(Context.PROVIDER_URL,JMS_URL);
            InitialContext ctx = new InitialContext(env);
            QueueConnectionFactory connectionFactory =
                              (QueueConnectionFactory)ctx.lookup(Connection_Name);
            Queue queue = (Queue)ctx.lookup(Queue_Name);

            ctx.close();
```

Sender - Continued

```
// Create connection, session & sender

                QueueConnection connection = connectionFactory.createQueueConnection();
                                QueueSession session =
                connection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
                                QueueSender sender = session.createSender(queue);
                                sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

                                // Send the messages to the queue
                                TextMessage msg = session.createTextMessage();

                                msg.setText("Hello to that one special listener");
                                System.out.println("Sending the following message:\" "+msg.getText() + "\"");
                                sender.send(msg);

                                // Close resources
                                sender.close();
                                session.close();
                                connection.close();

                                System.out.println("\nFinished.");

                } catch (Exception e)
                {
                                System.err.println("Exception: "+e);
                                System.exit(-1);
                }
        }
}
```

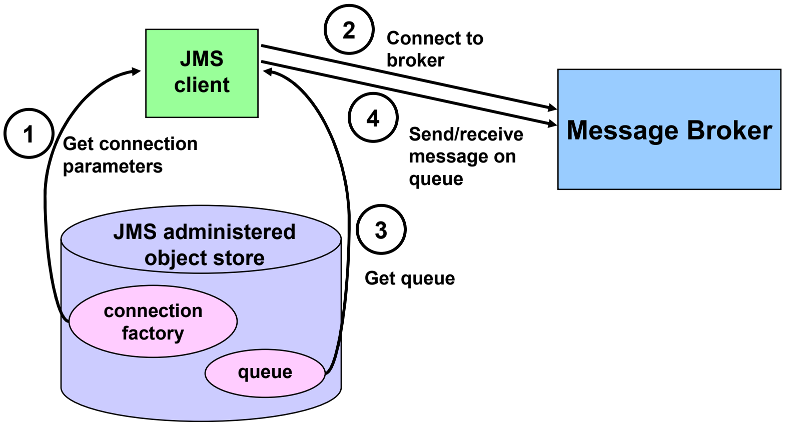
Basic Example Code - Receiver

```
public class P2PReceiver
{
      public static void main(String[] args)
      {
                String JMS_URL = "smqp://localhost:4001";
                String Connection_Name = "plainsocket@router1";
                String Queue_Name = "queue1@router1";

                try {
                                // Perform the JNDI lookup.
                                Hashtable env = new Hashtable();

                env.put(Context.INITIAL_CONTEXT_FACTORY,"com.swiftmq.jndi.InitialContextFactoryImpl");
                                env.put(Context.PROVIDER_URL,JMS_URL);
                                InitialContext ctx = new InitialContext(env);
                                QueueConnectionFactory connectionFactory =
      (QueueConnectionFactory)ctx.lookup(Connection_Name);
                                Queue queue = (Queue)ctx.lookup(Queue_Name);

                                // Important to note that you should close the context thereafter, because
                                // the context holds an active JMS connection.
                                ctx.close();

                                // Create connection, session & receiver
                                QueueConnection connection = connectionFactory.createQueueConnection();
                                QueueSession session =
      connection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
                                QueueReceiver receiver = session.createReceiver(queue);

                                // Start the connection
                                connection.start();
```

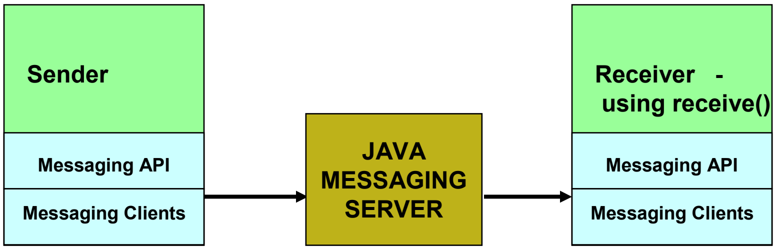
Receiver - Continued

```
// Receive the messages

                        TextMessage msg = (TextMessage)receiver.receive();

                        if(msg instanceof TextMessage)
                        {
                                    System.out.println("Message received: \"" + msg.getText() +"\"");
                        }


                        // Close resources
                        receiver.close();
                        session.close();
                        connection.close();

                        System.out.println("\nFinished.");

            } catch (Exception e)
            {
                        System.err.println("Exception: "+e);
                        System.exit(-1);
            }
    }
}
```

General Overview of Basic Code


JMS client
1
Get connection parameters
2
Connect to broker
4
Send/receive message on queue
Message Broker
JMS administered object store
connection factory
queue
3
Get queue

Review Basic Code



- Queues are not created dynamically – must be done administratively
- Temporary Queues do exist but are limited per the connection that made them
- Must decide the type of connection that will be made i.e. QueueConnection or TopicConnection, then obtain the respective session, then obtain the type of message to send.

Asychnronous Queue Listener that Filters

- **On the sender**

```
TextMessage msg = session.createTextMessage();
msg.setStringProperty(“Bananas”,”Ripe”);
```

- **On the receiver**

```
public class P2PReceiver implements MessageListener
{
        public P2PReceiver()
        {          …

        receiver = session.createReceiver(“queue1@router1”, “Bananas = ‘Ripe’”);
        receiver.setMessageListener(this);
        }

        onMessage(Message message)
        {
        }
}
```

Message selector examples

```
JMSPriority >= 8
Flavor = ‘Chocolate’ AND Quantity > 100
AvailableNow = true
CustomerName = ‘Progress Software’
CustomerName LIKE 'P%'
```

What about Topics?

- Have a one-to-many relationship for producer to consumers
- Can be hierarchichal to provide more refined filtering architecture – must be configured administratively prior to implementation
- Have Durable Subscribers
- Do not have a ‘Browser’

Code Differences

QueueConnection → TopicConnection

QueueSession → TopicSession

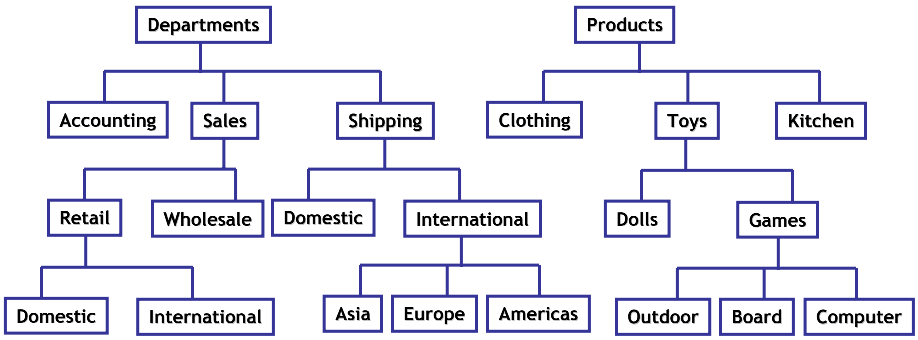
QueueSender → TopicPublisher

QueueReceiver → TopicSubscriber

Has hierarchy:

```
String topicName = “TopTopic.LowerTopic”;
TopicSubscriber subscriber =
session.createSubscriber(topicName,”Selector=‘value’”,false);
```

Hierarchy commands

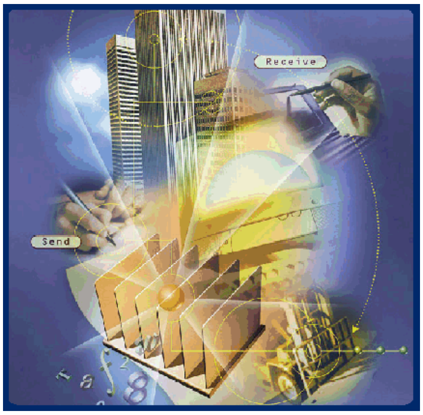


- Departments
  - Accounting
  - Sales
    - Retail
      - Domestic
      - International
    - Wholesale
  - Shipping
    - Domestic
    - International
      - Asia
      - Europe
      - Americas
- Products
  - Clothing
  - Toys
    - Dolls
    - Games
      - Outdoor
      - Board
      - Computer
  - Kitchen

Depending upon the JMS provider, topics may be accessed with wildcards:

Departments.%.Wholesale or *.Wholesale may be the same

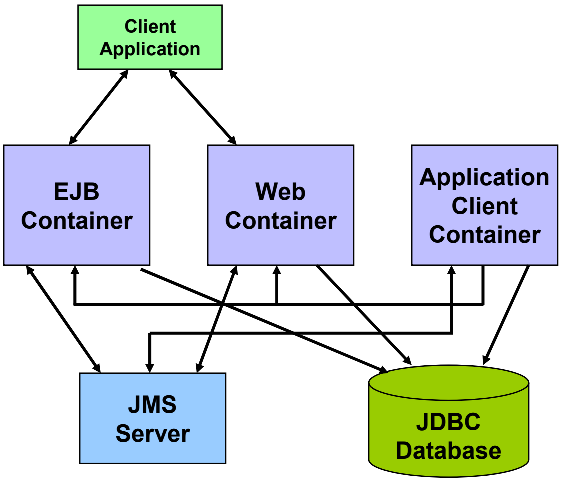
The Web and Asynchronous Services



Looking at how the Java Messaging Service may be employed to provide asynchronous funtionality for web based applications

JMS and the Web

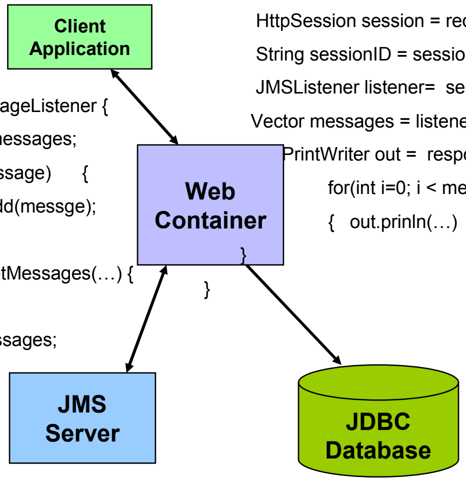
The J2EE platform



Asychronous Web Overview

- Web Client makes a request to a web server which strips the message header from the request via a Servlet
- The Servlet then pushes a message to a JMS server
- The JMS broadcasts the information to a number of listeners
- **A Servlet ‘proxy’ client is created to receive JMS messages that are passed to the web client upon the next request**

Polling



```
Public class JMServlet extends HttpServlet{
    public void doGet(…)    {
        HttpSession session = request.getSession(true);
        String sessionID = session.getId();
        JMSListener listener=  session.getValue(“JMSListener”);
       Vector messages = listener.getMessages(sessionID);
         PrintWriter out =  response.getWriter();
              for(int i=0; i < messages.size(); i++)
              {   out.prinln(…)  }
          }
    }
```

```
…implements MessageListener {
    private Vector messages;
    onMessage(message)    {
        messages.add(messge);
    }
    public Vector getMessages(…) {
            …
            return messages;
    }
}
```

Create Servlet

Create Listener

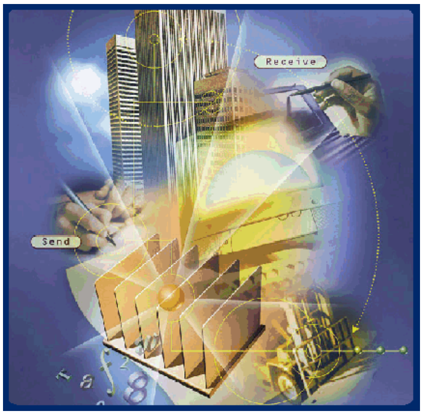
Get sessionID

Get Listener

Get messages

Post response

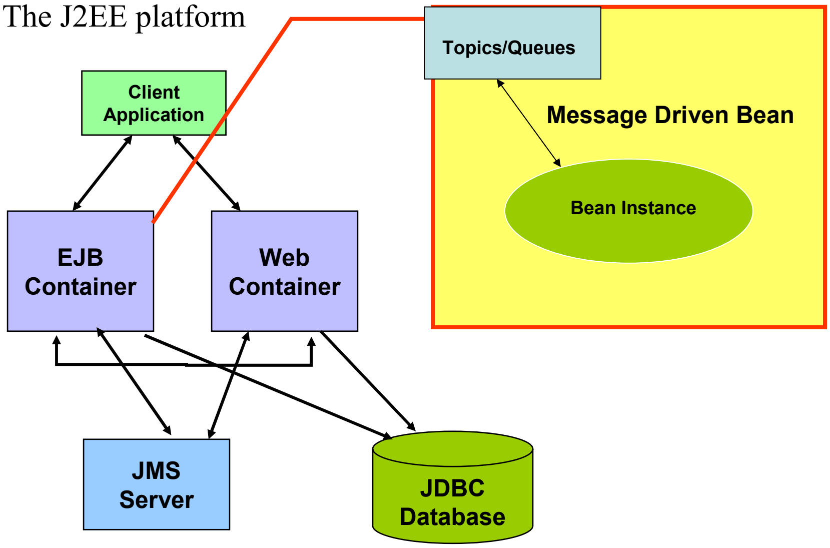
Message Driven Beans



Looking at how Asynchronous message handling has been incorporated into Application Servers

JMS and MDB's

The J2EE platform



What’s Needed

- Application Server that supports the EJB 2.0 standards
- A JMS provider
- A deployment file
- Substitution of existing JMS client code in the place of the onMessage(…) method of the MDB

*** Use is entirely dependent upon application architecture…not necessary if *only* a robust messaging model is required**

Application Servers vs. JMS Providers

- Cost
- Ease of use
- Needs
- Clustering, failover…

MDB Criteria

- Composed of bean class and XML deployment descriptor
- Bean class must implement
  1. Javax.ejb.MessageDrivenBean
     - ejbCreate()
     - ejbRemove()
     - setMessageDrivenContext(MessageDrivenContext mdc
  2. Javax.jms.MessageListener
     - onMessage(Message m);

Deployment Descriptor Example

```
• <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
  JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
  <ejb-jar>
      <enterprise-beans>
          <message-driven>
              <ejb-name>QueueBean</ejb-name>
              <ejb-class>com.queue.mdb.LineUp</ejb-class>
              <transaction-type>Container</transaction-type>
              <jms-acknowledge-mode>auto-acknowledge</jms-acknowledge-
  mode>
              <message-driven-destination>
                  <jms-destination-type>javax.jms.Queue</jms-destination-
  type>
              </message-driven-destination>
          </message-driven>
      </enterprise-beans>
  </ejb-jar>
```

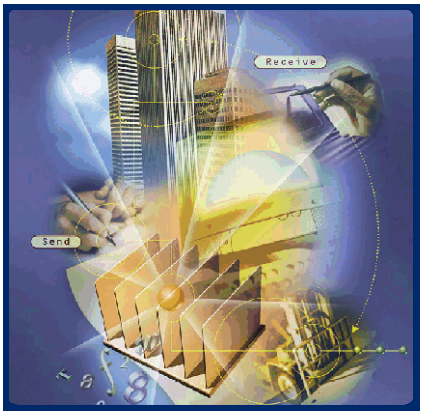
Vendor Specific DD

```
<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD
    WebLogic 6.0.0 EJB//EN"
    "http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd">
    <weblogic-ejb-jar>
        <weblogic-enterprise-bean>
            <ejb-name>QueueBean</ejb-name>
            <message-driven-descriptor>
                <pool>
                    <max-beans-in-free-pool>10</max-beans-in-free-pool>
                    <initial-beans-in-free-pool>2</initial-beans-in-free-pool>
                </pool>
                <destination-jndi-name>logqueue</destination-jndi-name>
            </message-driven-descriptor>
            <jndi-name>QueueBean</jndi-name>
        </weblogic-enterprise-bean>
    </weblogic-ejb-jar>
```

Combined with the Web

- Acts as a JMS client to the JMS provider interfacing with the servlet
- For SOAP messages:
  1. Strip off the SOAP header and send out the requisite information
  2. Use JAXM (Java API for XML Messaging) included in the EJB 2.1 standard

Clustering, Failover and DRA



A fully robust JMS configuration and what it provides

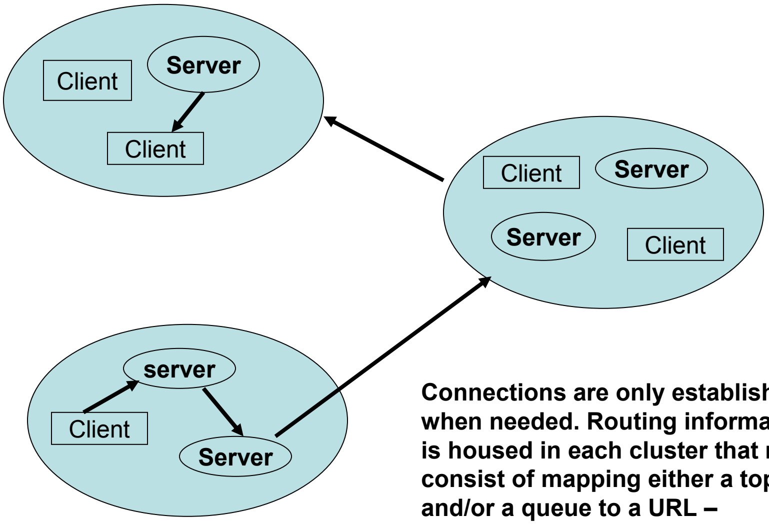
Fundamentals Considerations

- Performance
- Performance
- Performance
- What happens when the JMS server dies?
- What happens if the client dies?
- What happens if my system is under performing?
- How do I connect to a bunch of servers?
- Where’s my business logic?

Clustering and Fail Over

- ➢ Message mirroring is not an option due to performance degradation and guaranteed message delivery
- ➢ Clustering consists of pushing pub/sub messages to the appropriate server to which the listening client is connected
- ➢ If too many clients are connected to the system the next connection will be routed to another server
- ➢ Flow control is an administrative task

Dynamic Routing Architecture



Connections are only established when needed. Routing information is housed in each cluster that may consist of mapping either a topic and/or a queue to a URL – elaborate JNDI schema of sorts

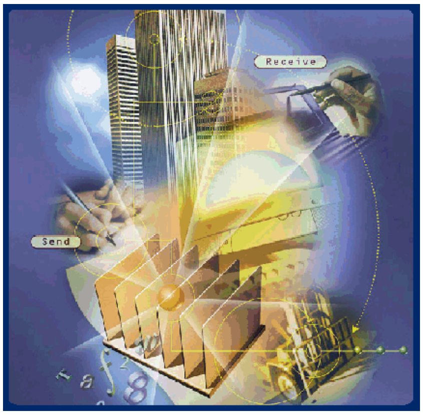
Business Logic

- ➢ Business functionality is achieved via filtering using MessageSelectors or hierarchy configurations when topics are employed
- ➢ Created on the client
  - – What happens if business logic needs to be updated?
  - – How can you design a system requiring continuous business logic changes?

JMS Providers

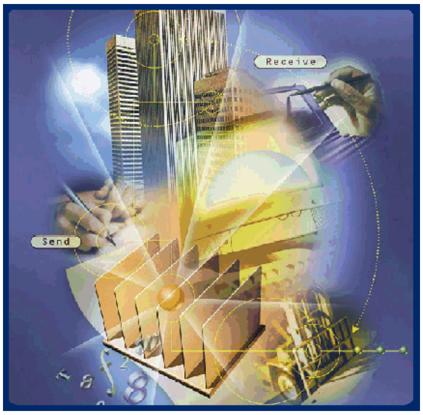
- ➢ SwiftMQ
- ➢ SonicMQ
- ➢ FioranoMQ
- ➢ MQSeries
- ➢ Tibco
- ➢ WebLogic
- ➢ SpiritSoft

Summary



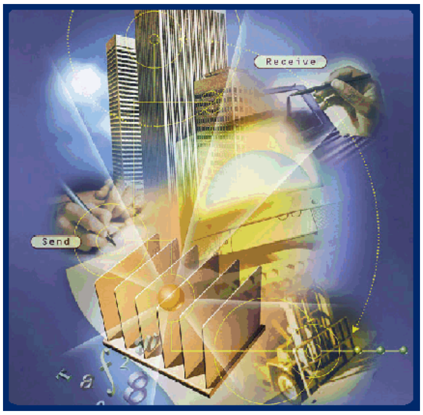
- JMS adopts a loosely coupled, reliable, asynchronous mechanism for information delivery
- Compliments Web based applications very well
- API is easy to implement

About the Hartmann Software Group



- ***Software Training, Services and Products Company***
- ***Expertise in the Object-Oriented Middleware space***
- ***Employ Rule Engine technology to capture business rule functionality into a RETE network***
- ***Deliver affordable solutions and services businesses can bank on!***

Thank you!



Questions?