Messaging

JMS

Messaging

Messaging is a method of communication between software components or applications.

A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages.
Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

Messaging

	Send	Synchronously Receive	Asynchronously Receive
Application clients	YES	YES	YES
Enterprise JavaBeans (EJB) components	YES	YES	NO
Web components	YES	YES	NO
Message Driven Beans			

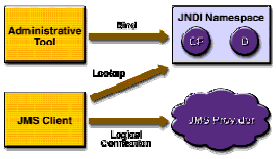
Messaging

A *JMS provider* is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the J2EE platform at release 1.3 and later includes a JMS provider.

JMS clients are the programs or components, written in the Java programming language, that produce and consume messages. Any J2EE application component can act as a JMS client.

Messages are the objects that communicate information between JMS clients.

Administered objects are preconfigured JMS objects created by an administrator for the use of clients.



D=Destination
CF=Connection Factory

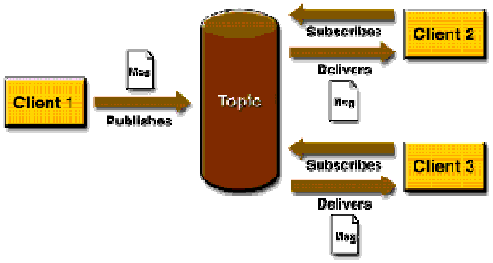
Messaging Domains: PtP

Point-to-Point Messaging Domain
A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire.

Messaging Domains: publish-subscribe

Publish/subscribe messaging has the following characteristics.
Each message can have multiple consumers.

Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.



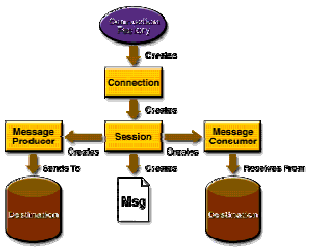
Message Consumption

Synchronously: A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.

Asynchronously: A client can register a *message listener* with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage method, which acts on the contents of the message.

Roles in the JMS scenario

Administered Objects: connection factories and destinations
Connections
Sessions
Message Producers
Message Consumers
Messages



Getting connections

```
Context ctx = new InitialContext();
ConnectionFactory connectionFactory1 =
        (ConnectionFactory)   ctx.lookup("jms/QueueConnectionFactory");
ConnectionFactory connectionFactory2 =
        (ConnectionFactory)   ctx.lookup("jms/TopicConnectionFactory");

Connection connection = connectionFactory1.createConnection();
```

Before an application completes, you must close any connections that you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

```
connection.close();
```

Destinations and Sessions

A *destination* is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues. In the pub/sub messaging domain, destinations are called topics.

```
Destination myDest = (Topic) ctx.lookup("jms/MyTopic");
Destination myDest = (Queue) ctx.lookup("jms/MyQueue");
```

A *session* is a single-threaded context for producing and consuming messages. You use sessions to create message producers, message consumers, and messages. Sessions serialize the execution of message listeners

```
//non transacted session, no acknowledgement
Session session =
        connection.createSession(false,   Session.AUTO_ACKNOWLEDGE);
//transacted session, no acknowledgement
Session session = connection.createSession(true, 0);
```

Message Producer

A *message producer* is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.
You use a Session to create a MessageProducer for a destination. Here, the first example creates a producer for the destination myQueue, and the second for the destination myTopic:

```
MessageProducer producer = session.createProducer(myQueue);
MessageProducer producer = session.createProducer(myTopic);
```

After you have created a message producer, you can use it to send messages by using the send method:

```
producer.send(message);
```

Message Consumer

A *message consumer* is an object that is created by a session and used for receiving messages sent to a destination. It implements the MessageConsumer interface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

You use a Session to create a MessageConsumer for either a queue or a topic:

```
MessageConsumer consumer = session.createConsumer(myQueue);
MessageConsumer consumer = session.createConsumer(myTopic);
```

You use the receive method to consume a message synchronously. You can use this method at any time after you call the start method:

```
connection.start();
Message m = consumer.receive();
Message m = consumer.receive(1000); // time out after a second
```

Message Listener

A *message listener* is an object that implements the MessageListener interface and acts as an asynchronous event handler for messages. The MessageListener interface contains one method, onMessage(Message m), where you define the actions to be taken when a message arrives.

You register the message listener with a specific MessageConsumer by using the setMessageListener method. For example, if you Listener implements the MessageListener interface:

```
MessageListener myListener = new Listener();
consumer.setMessageListener(myListener);
Connection.start();  //If you call start before you register the message
        listener, you are likely to miss messages.
```

When message delivery begins, the JMS provider automatically calls the message listener's onMessage method whenever a message is delivered.

A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on the type of destination for which the message consumer was created.

Message Types

TextMessage A java.lang.String object (for example, the contents of an Extensible Markup Language file).

MapMessage A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.

BytesMessage A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.

StreamMessage A stream of primitive values in the Java programming language, filled and read sequentially.

ObjectMessage A Serializable object in the Java programming language. Message Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

Exchanging messages

Producer:

```
TextMessage message = session.createTextMessage();
message.setText(msg_text); // msg_text is a String
producer.send(message);
```

Consumer:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
        TextMessage message = (TextMessage) m;
        System.out.println("Reading message: " + message.getText());
} else {   // Handle error }
```

Messages also have headers and optional Properties

Message Driven Beans

A *message-driven bean* is an enterprise bean that allows J2EE applications to process messages asynchronously. It normally acts as a JMS message listener. The messages can be sent by any J2EE component--an application client, another enterprise bean, or a Web component--or by a JMS application or system that does not use J2EE technology. Message-driven beans can process either JMS messages or other kinds of messages.

The main difference between a message-driven bean and other enterprise beans is that a message-driven bean has no home or remote interface. Instead, it has only a bean class.

Structure of a Message Driven Bean:

Your message-driven bean class must implement the following in addition to the onMessage method:

The Interfaces:
javax.ejb.MessageDrivenBean and javax.jms.MessageListener.

The methods:
public void ejbCreate() {}
If your message-driven bean produces messages or does synchronous receives from another destination, you use its ejbCreate method to look up JMS API connection factories and destinations and to create the JMS API connection.
public void ejbRemove() {}
If you used the message-driven bean's ejbCreate method to create a JMS API connection, you ordinarily use the ejbRemove method to close the connection.
public void setMessageDrivenContext(MessageDrivenContext mdc) {}
A MessageDrivenContext object provides some additional methods that you can use for transaction management.

Example – Sender 1

```
package com.mydomain;
import java.io.*; import java.util.*;
import javax.jms.*; import javax.naming.*;
import javax.servlet.*; import javax.servlet.http.*;
public class ProducerServlet extends HttpServlet {
    Context context = null;
    QueueConnection queueConnection = null;
    QueueSession queueSession = null;
    Queue queue = null;
    QueueSender queueSender = null;
    TextMessage message = null;
    boolean esito = true;
```

Esempio tratto da:
www.javaportal.it/docs/jms.htm
Autore: Giulio Rambelli

Example – Sender 2

```
/* uso l'IP di destinazione preso dal file web.xml in cui si trova come init-
param per inizializzare l'InitialContext  */
   public void init() {
         String destinationIP = getInitParameter("destinationIP");
         Hashtable env = new Hashtable();
         env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                  "org.jnp.interfaces.NamingContextFactory");
         env.put("java.naming.factory.url.pkgs",
                  "org.jboss.naming:org.jnp.interfaces");
         env.put(javax.naming.Context.PROVIDER_URL, destinationIP);
         try {
                  context = new InitialContext(env);
                  getQueueSender();
                  context.close();
         } catch(Exception exc) {
                  System.out.println(exc.toString());
                  esito = false;
         }
   }
```

Example – Sender 3

```
/* prendo la factory da cui ottenere una QueueConnection e la Queue
dall'albero JNDI inizializzo i vari oggetti che mi serviranno per l'invio del
messaggio e creo il messaggio stesso (per adesso vuoto). */

   public void getQueueSender() throws JMSException,NamingException {
      QueueConnectionFactory queueFactory =
        (QueueConnectionFactory)context.lookup("ConnectionFactory");
      queueConnection = queueFactory.createQueueConnection();
      queueSession = queueConnection.createQueueSession(false,
                  javax.jms.Session.AUTO_ACKNOWLEDGE);
      queue = (Queue)context.lookup("queue/A");
      queueSender = queueSession.createSender(queue);
      message = queueSession.createTextMessage();
   }
```

Example – Sender 4

```
/* setto il testo immesso dall'utente al TextMessage e lo invio. */

    public void sendMsg(String testo) throws JMSException {
          message.setText(testo);
          queueSender.send(queue, message);
    }
```

Example – Sender 5

```
public void service(HttpServletRequest request, HttpServletResponse
                response) throws IOException, ServletException {
        String whereToForward = null;
        String testo = request.getParameter("testo");
        if (esito) {
                try {
                        sendMsg(testo);
                        whereToForward = "success";
                } catch (JMSException jme) {
                        System.out.println(jme.toString());
                        whereToForward = "failure";
                }
        } else { whereToForward = "failure"; }
        System.out.println("whereToForward: " + whereToForward);
        request.getRequestDispatcher("/" + whereToForward +
        ".jsp").forward(request,response);
    }
```

Example – Sender 6

```
public void destroy() {
        try {
                if(queueSender != null)queueSender.close();
        } catch (JMSException jme) {   System.out.println(jme.toString());
        } finally { queueSender = null; }
        try {
                if(queueSession != null)queueSession.close();
        } catch (JMSException jme) {   System.out.println(jme.toString());
        } finally { queueSession = null;}
        try {
                if(queueConnection != null)queueConnection.close();
        } catch (JMSException jme) {   System.out.println(jme.toString());
        } finally { queueConnection = null; }
   }
}
```

Example – web.xml

```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
    <servlet-name>producer</servlet-name>
    <servlet-class>com.mydomain.ProducerServlet</servlet-class>
    <init-param>
      <param-name>destinationIP</param-name>
      <param-value>127.0.0.1</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>producer</servlet-name>
    <url-pattern>ProducerServlet</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Example – index.html

```
<html>
<head>
<title>Prova jms</title>
</head>
<body>
<form action="ProducerServlet">
<input type="text" name="testo">
<input type="submit" value="INVIA!">
</form>
</body>
</html>
```

Example – success.jsp

```
<html>
<head>
<title>Success!</title>
</head>
<body>
<center>
<h3>Messaggio <%= request.getParameter("testo") %> accodato con
successo</h3>
<a href="index.html">TORNA</a>
</center>
</body>
</html>
```

Example – failure.jsp

```
<html>
<head>
<title>Failure!</title>
</head>
<body>
<center>
<h3>Messaggio <%= request.getParameter("testo") %> non
accodato!!!</h3>
<a href="index.html">TORNA</a>
</center>
</body>
</html>
```

Example – Consumer 1

```
package ejb.mdb;
import java.io.*; import javax.ejb.*; import javax.jms.*;
import javax.naming.*; import javax.ejb.EJBException;
public class GenericMDBean implements MessageDrivenBean,
                    MessageListener {
   String filePath = new String();
/* al momento della creazione dell'mdb viene inizializzato il Context
e viene preso il path al file su cui scrivere presente come <env-entry>
nel file ejb-jar.xml. */
   public void ejbCreate() {
         try {
            InitialContext ctx = new InitialContext();
            filePath = (String)ctx.lookup("java:comp/env/destinazione");
            ctx.close();
         } catch(NamingException ne) {
            throw new RuntimeException(ne.toString());}
   }
   public void ejbRemove() {}
   public void setMessageDrivenContext(MessageDrivenContext mdc)
         throws EJBException {}
```

Example – Consumer 2

```
public void onMessage(Message message) {
        if(message instanceof TextMessage) {
            TextMessage textMsg = (TextMessage)message;
            try {
                String testo = textMsg.getText();
                scriviTesto(testo);
            } catch(Exception exc) {
                throw new RuntimeException(exc.toString());
            }
        }
    }
    public void scriviTesto(String testo) throws Exception {
        PrintWriter pw =
                new PrintWriter(new FileWriter(new File(filePath), true));
        BufferedWriter bw = new BufferedWriter(pw);
        bw.write(testo);
        bw.newLine();
        bw.flush();
        bw.close();
    }
}
```

Example – Consumer ejb-jar.xml


```
<?xml version="1.0"?>  <!DOCTYPE ejb-jar ...>
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>GenericMDBean</ejb-name>
      <ejb-class>ejb.mdb.GenericMDBean</ejb-class>
      <message-selector></message-selector>
      <transaction-type>Bean</transaction-type>
      <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
      <message-driven-destination>
      <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
      <env-entry>
        <description>Il path e il nome del file su cui scrivere</description>
        <env-entry-name>destinazione</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>C:\destinazione.txt</env-entry-value>
      </env-entry>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Example – Consumer jboss.xml

```
<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>GenericMDBean</ejb-name>
      <configuration-name>Standard Message Driven Bean
      </configuration-name>
      <destination-jndi-name>queue/A</destination-jndi-name>
    </message-driven>
  </enterprise-beans>
</jboss>
```

Complementi sulle Servlet

Dispatching, monitoring etc.

Dispatching

```
RequestDispatcher dispatch =
        cntx.getRequestDispatcher("/SecondServlet");
dispatch.forward(req,res);
```

```
RequestDispatcher dispatch =
        cntx.getRequestDispatcher("/SecondServlet");
dispatch.include(req,res);
```

Dispatching example

```
package servlets;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletContext;
import javax.servlet.RequestDispatcher;

 public class SecondServlet extends HttpServlet  {
public void doGet(HttpServletRequest req,HttpServletResponse res)
    throws IOException,ServletException {
  Printer out=res.getWriter();
      System.out.println("Second Servlet Called");
  }
}
```

Dispatching example

```
package servlets;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletContext;
import javax.servlet.RequestDispatcher;

 public class FirstServlet extends HttpServlet  {
public void doGet(HttpServletRequest req,HttpServletResponse res)
throws IOException,ServletException {
  Printer out=res.getWriter();
      out.println("First Servlet Called");
      ServletConfig config = getServletConfig();
  ServletContext cntx = config.getServletContext();
  RequestDispatcher dispatch =
            cntx.getRequestDispatcher("/SecondServlet");
  dispatch.forward(req,res);
}
}
```

Dispatching example

```
   <servlet>
<servlet-name>FirstServlet</servlet-name>
<servlet-class>servlets.FirstServlet</servlet-class>
</servlet>

<servlet>
<servlet-name>SecondServlet</servlet-name>
<servlet-class>servlets.SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>FirstServlet</servlet-name>
<url-pattern>/firstservlet/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
<servlet-name>SecondServlet</servlet-name>
<url-pattern>/SecondServlet/*</url-pattern>
</servlet-mapping>
```

Monitoring Servlets Lifecycle

Web context	Initialization and Destruction	ServletContextListener	ServletContextEvent
	Attribute added, removed, or replaced	ServletContextAttributeListener	ServletContextAttributeEvent
Session	Creation, invalidation, activation, passivation, and timeout	HttpSessionListenerHttpSessionActivationListener	HttpSessionEvent
	Attribute added, removed, or replaced	HttpSessionAttributeListener	HttpSessionBindingEvent
Request	A servlet request has started being processed by Web components	ServletRequestListener	ServletRequestEvent
	Attribute added, removed, or replaced	ServletRequestAttributeListener	ServletRequestAttributeEvent

Monitoring Servlets Lifecycle - Example

```
/* File : ApplicationWatch.java */
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;
public class ApplicationWatch implements
ServletContextListener {
public static long applicationInitialized = 0L;
/* Application Startup Event */
public void contextInitialized(ServletContextEvent ce) {
applicationInitialized = System.currentTimeMillis(); }
/* Application Shutdown Event */
public void contextDestroyed(ServletContextEvent ce) {}
}
```

Monitoring Servlets Lifecycle - Example

```
/* File : SessionCounter.java */
import javax.servlet.http.HttpSessionListener;
import javax.servlet.http.HttpSessionEvent;
public class SessionCounter implements
HttpSessionListener {
private static int activeSessions = 0;
/* Session Creation Event */
public void sessionCreated(HttpSessionEvent se) {
activeSessions++; }
/* Session Invalidation Event */
public void sessionDestroyed(HttpSessionEvent se) {
if(activeSessions > 0) activeSessions--; }
public static int getActiveSessions() { return
activeSessions; }
}
```

Monitoring Servlets Lifecycle - Example

```
<!-- Web.xml -->
<?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>
<!-- Listeners -->
<listener>
    <listener-class>
com.stardeveloper.web.listener.SessionCounter </listener-
class>
</listener>
<listener>
    <listener-class>
com.stardeveloper.web.listener.ApplicationWatch </listener-
class>
</listener>
</web-app>
```

Scope Objects

Web context	ServletContext	Web components within web context servlet.getServletConfig().getServletContext
Session	HttpSession	Web components handling requests that belong to a session
Request	ServletRequest	Web component handling the request
Page	PageContext	Web component in the JSP page

Main Methods:
Object getAttribute(String name)
void setAttribute(String name, Object o)
Enumeration getAttributeNames()

Filters (javax.servlet.filter)

Other classes that preprocess/postprocess request/response

A filter is an object than perform filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.

Filters perform filtering in the doFilter method. Every Filter has access to a FilterConfig object from which it can obtain its initialization parameters, a reference to the ServletContext which it can use, for example, to load resources needed for filtering tasks.

Filters are configured in the deployment descriptor of a web application

Examples that have been identified for this design are
1) Authentication Filters
2) Logging and Auditing Filters
3) Image conversion Filters
4) Data compression Filters
5) Encryption Filters
6) Tokenizing Filters
7) Filters that trigger resource access events
8) XSL/T filters
9) Mime-type chain Filter