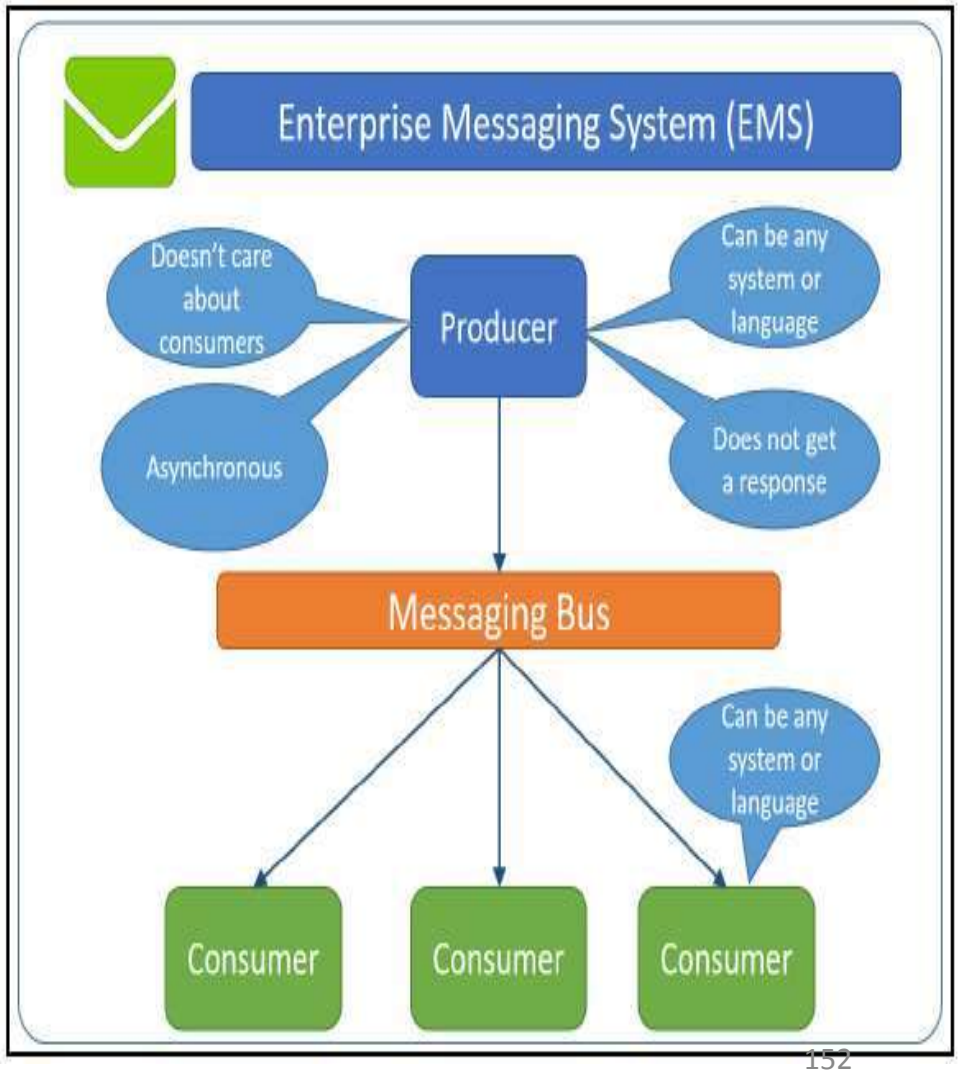
ASSIGNMENT NO. 6	
Assignment No. 6	To develop any distributed application using Messaging System in **Publish - Subscribe paradigm**.
Objective(s):	By the end of this assignment, the student will be able to create, deploy and test Web-service application.
Tools	Eclipse, Java 8, Apache ActiveMQ

ENTERPRISE MESSAGING SYSTEM

- A **specification/standard** that describes a common way for programs to create, send, receive and read distributed enterprise messages.
- Common formats, such **as XML or JSON,** are used to do this.
- EMS recommends the messaging protocols: Data Distribution Service (DDS), Message Queuing (MSMQ), Advanced Message Queuing Protocol (AMQP), or SOAP web services.
- Systems designed with EMS are termed **Message-Oriented Middleware** (MOM).



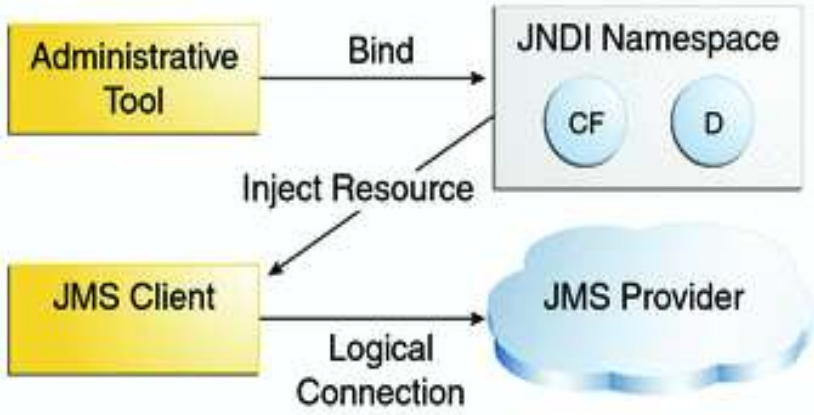
JAVA MESSAGING SERVICE

- The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Messaging enables distributed communication that is **loosely coupled.**
- The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications.
- Messaging is used for communication between software applications or software components.
- A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.

JMS API Architecture

- A JMS application is composed of the following parts:
  - A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features.
  - **JMS clients** are the programs or components, written in the Java programming language, that **produce and consume** messages.
  - **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. The two kinds of JMS administered objects are destinations and connection factories, described in JMS Administered Objects.

- Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

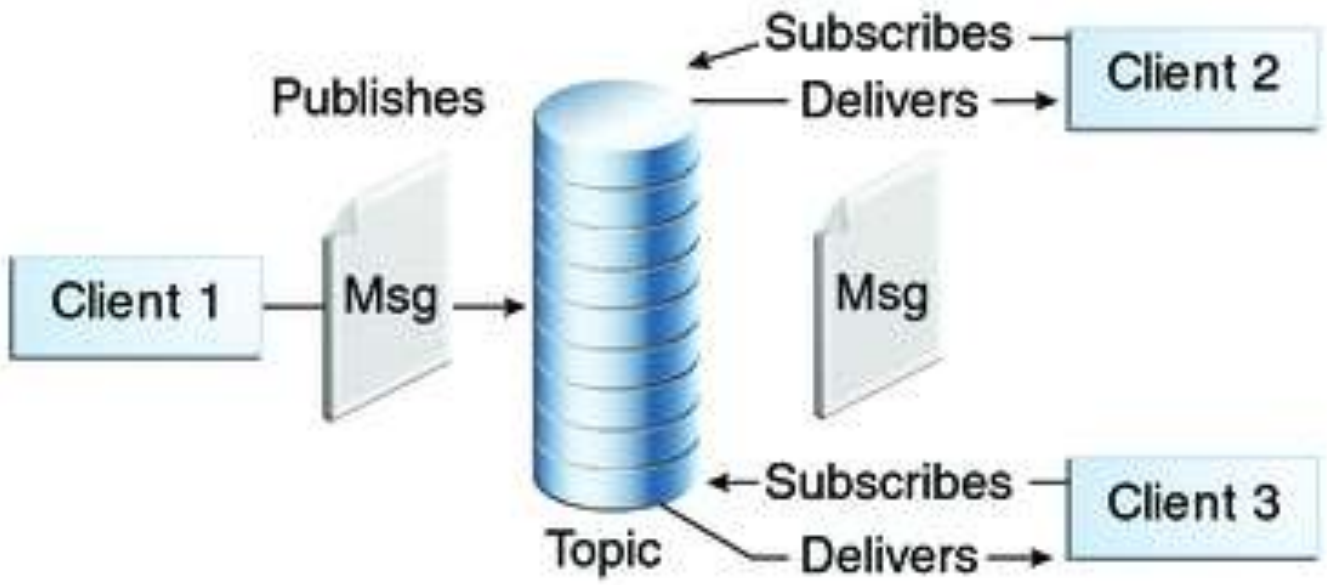


Public Subscribe Messaging Approach

- JMS API support both the point-to-point and the publish/subscribe approach .
- A stand-alone JMS provider can implement one or both domains.
- In a **publish/subscribe** (pub/sub) system, clients address messages to a **topic**, which functions somewhat like a bulletin board.
- Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy.
- The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers.
- Topics retain messages only as long as it takes to distribute them to current subscribers.
- Pub/sub 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.

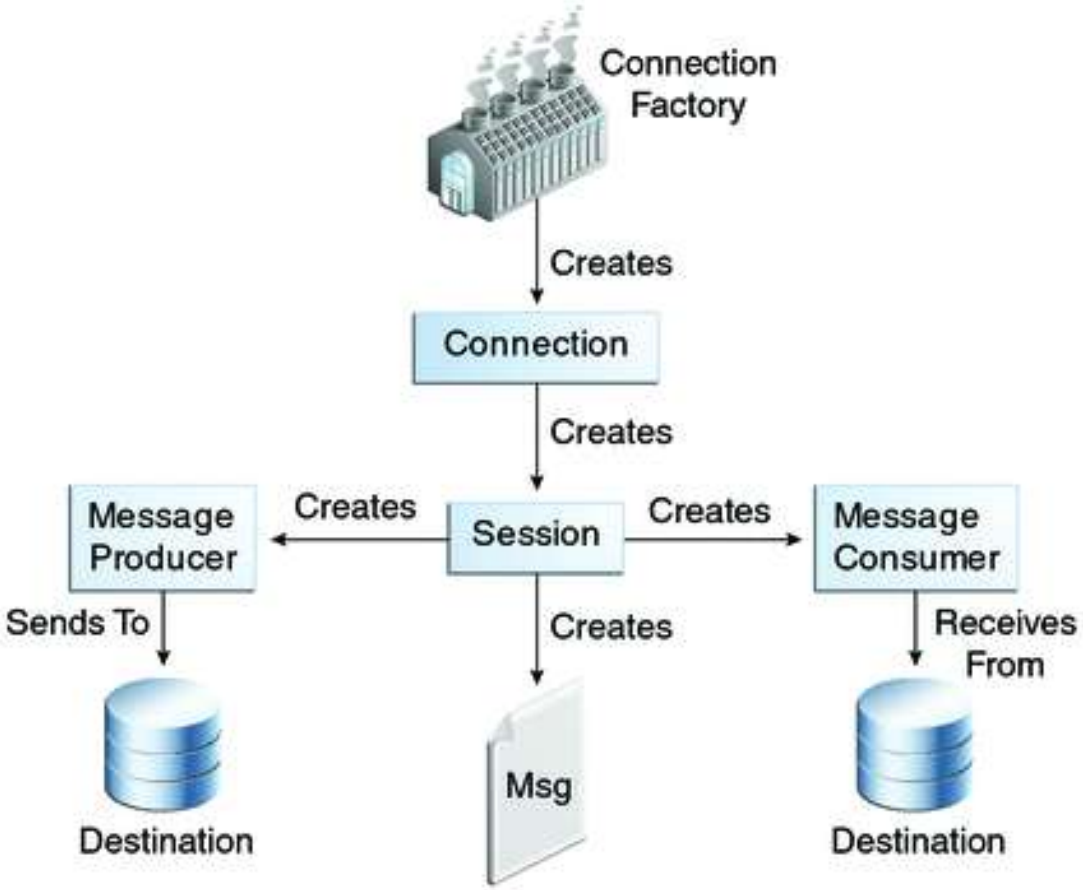
Public Subscribe Messaging Approach

- In a **publish/subscribe** (pub/sub) system, clients address messages to a **topic**, which functions somewhat like a bulletin board.



The JMS API Programming Model

- The basic building blocks of a JMS application are:
- **Administered objects:** connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages



The JMS API Programming Model

- The basic building blocks of a JMS application are:
- **Administered objects:** The management of these objects belongs to provider. (connection factories and destinations).
- **JMS Connection Factories:** A **connection factory** is the object a client uses to create a connection to a provider.
- Each connection factory is an instance of the `CoonectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface.
- **JMS Destinations:** 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 pub/sub messaging domain, destinations are called topics.
- **JMS Connections:** A **connection** encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.
- Before an application completes, you must close any connections you have created.

The JMS API Programming Model

- **JMS Sessions:** A **session** is a single-threaded context for producing and consuming messages.
- Sessions are used to create the Message Producers, Message Consumers, Messages, Topics etc.
- **JMS Message Listeners:** A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

INSTALLING ActiveMQ

- ActiveMQ is an open source message broker written in java.
- **To install the same download Apache ActiveMQ:** http://www.apache.org/dist//activemq/apache-activemq/5.5.0/apache-activemq-5.5.0-bin.tar.gz

```
[suncoma@wso2 ~]$ wget http://www.apache.org/dist//activemq/apache-activemq/5.5.0/apache-activemq-5.5.0-bin.tar.gz
--2011-05-23 04:50:29--  http://www.apache.org/dist//activemq/apache-activemq/5.5.0/apache-activemq-5.5.0-bin.tar.gz
Resolving www.apache.org... 140.211.11.131
Connecting to www.apache.org|140.211.11.131|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 27495046 (26M) [application/x-gzip]
Saving to: `apache-activemq-5.5.0-bin.tar.gz'
```

INSTALLING ActiveMQ

- **Extract the Archive:**
  - If the ActiveMQ start-up script is not executable, change its permissions: `chmod 755 activemq`
- **Run Apache ActiveMQ:**
  - Run ActiveMQ from a command shell:

    `sudo sh activemq start`

```
[suncoma@wso2 bin]$ sudo sh activemq start
INFO: Using default configuration
(you can configure options in one of these file: /etc/default/activemq /home/suncoma/.activemqrc)

INFO: Invoke the following command to create a configuration file
activemq setup [ /etc/default/activemq | /home/suncoma/.activemqrc ]

INFO: Using java '/usr/bin/java'
INFO: Starting - inspect logfiles specified in logging.properties and log4j.properties to get details
INFO: pidfile created : '/opt/apache-activemq-5.5.0/data/activemq.pid' (pid '4081')
[suncoma@wso2 bin]$
```

INSTALLING ActiveMQ

- **Testing the Installation:**
- ActiveMQ's default port is 61616. From another window, run `netstat` and search for port 61616.

```
netstat -an|grep 61616
```

```
[suncoma@wso2 bin]$ netstat -an|grep 61616
tcp        0      0 :::61616                :::*                    LISTEN
[suncoma@wso2 bin]$
```

The port should be open and in LISTEN mode on 61616.

INSTALLING ActiveMQ

- **Monitoring ActiveMQ:**
- You can monitor ActiveMQ, using the Web Console by pointing your browser at:
- `http://localhost:8161/admin`

Publish-Subscriber Implementation

- Create a **new Project** in eclipse with package name **`pubSub`.**
- Create two new class files in the **`src`** directory named **`Publisher.java`** and **`Subscriber.java`**

- In both the java files, import packages java message service (jms) and ActiveMQ.

```
import javax.jms.*;

import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;
```

- `Publisher.java` file will create a **topic** which will be subscribed by the subscribes.

Implementation: Publisher

- In publisher class , the publisher uses the default broker url (URL of JMS server), provided by ActiveMQ.

```
public class Publisher {

    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;
```

- Establish a connection between Publisher and ActiveMQ.
  - Initialize connection between Publisher and the above url using `ActiveMQConnectionFactory()` method and create it with `createConnection()` method.
  - `start()` method starts the created connection.

```
11⊖   public static void main(String[] args) throws JMSException {
12
13        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
14        Connection connection = connectionFactory.createConnection();
15        connection.start();
```

Implementation: Publisher

- Create a session using `createSession()` method

```
17          // JMS messages are sent and received using a Session. We will
18          // create here a non-transactional session object. If you want
19          // to use transactions you should set the first parameter to 'true'
20          Session session = connection.createSession(false,
21                  Session.AUTO_ACKNOWLEDGE);
22
```

- Create a new topic to be published by the publisher with the `createTopic()` method.

```
23          //Set the topic to which the Subscriber will subscribe to
24          Topic topic = session.createTopic("DistributedSystem");
25
```

Implementation: Publisher

- Create a producer object and assign the topic to it from the previous step.

```
26          //Give the topic to the producer
27          MessageProducer producer = session.createProducer(topic);
28
29          // We will send a small text message saying "1.Chapter One"
30          TextMessage message = session.createTextMessage();
31          message.setText("1.Chapter One");
```

- Producer can create and send new messages for the topic using the `createTextMessage()` and `send()` method

```
33          // Here we are sending the message!
34          producer.send(message);
35          System.out.println("Sent message '" + message.getText() + "'");
36
37          connection.close();
```

Implementation: Subscriber

- Create a Subscriber class and create a connection similar to Producer.

```
3  import java.io.IOException;
4
5  import javax.jms.*;
6
7  import org.apache.activemq.ActiveMQConnection;
8  import org.apache.activemq.ActiveMQConnectionFactory;

10 public class Subscriber {
11     // URL of the JMS server
12     private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;

14     // Name of the topic from which we will receive messages from = "DistributedSystem"
15     public static void main(String[] args) throws JMSException {
16         // Getting JMS connection from the server
17         ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
18         Connection connection = connectionFactory.createConnection();
19         connection.start();
```

Implementation: Subscriber

- A subscriber listens to the topics which are published by a publisher by subscribing to the topic .

```
21          //Create a session
22          Session session = connection.createSession(false,
23                  Session.AUTO_ACKNOWLEDGE);
24
25          //This topic is the same as in Publisher
26          Topic topic = session.createTopic("DistributedSystem");
27
```

- The `createConsumer()` method takes the topic as an argument.

```
28          //Give the topic to the producer
29          MessageConsumer consumer = session.createConsumer(topic);
30
```

Implementation: Subscriber

- A `MessageListener` object is used to receive the delivered messages. The `MessageListener` interface has a `onMessage()` method which passes the message to the listener.

```
34        MessageListener listner = new MessageListener() {
35            public void onMessage(Message message) {
36                try {
37                    if (message instanceof TextMessage) {
38                        TextMessage textMessage = (TextMessage) message;
39                        System.out.println("Received message"
40                                + textMessage.getText() + "'");
41                    }
42                } catch (JMSException e) {
43                    System.out.println("Caught:" + e);
44                    e.printStackTrace();
45                }
46            }
47        };
49        consumer.setMessageListener(listner);
```

- Finally we set the message listener and close the connection.

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

Steps to Run and Expected Output

- Run `Subscriber.java` in Eclipse

- Run `Publisher.java` in Eclipse. (After this step the Subscriber.java will show a new Message)

References

JMS API Programming Model:

https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html