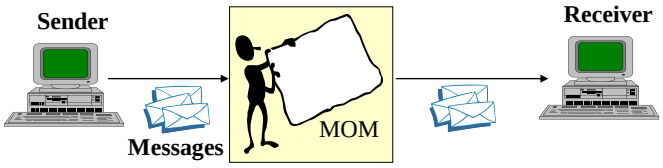
Messaging System

By Võ Văn Hải

Enterprise Message System – EMS

- Message is a data, which is used for communication.
- Message is used through a middle object (such as MOM or MQ) for sending and receiving the events and data between software component/ application.
- Provide facilities (synchronous/ asynchronous) for creating, sending, and receiving message.
- Application A communication Application B by sending a message via the MOM's application programming interface.



Message Oriented Middleware
Or
Message Queue

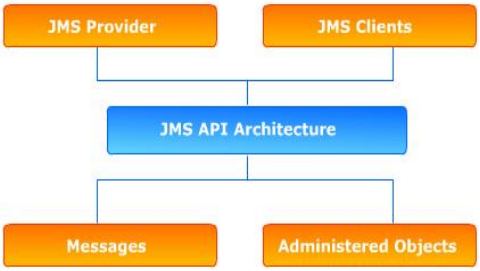
Message Oriented Middleware -MOM

- Apply to a distributed system for sending and receiving message between software components on network.
- Provide facilities for fault tolerance, load balancing, scalability, and transaction.
- Use formatting message and network protocol in processing.
- Venders:
  - HornetMQ
  - ActiveMQ
  - MSMQ
  - …

Java Message Service

- JMS is a Java-based messaging system that provides the facility of creating, sending, receiving, and reading messages to Java enterprise applications.
- The JMS API defines a common set of interfaces and associated semantics that allow creation of applications that use JMS to communicate with other applications.
- The JMS API enables communication that is:
  - Asynchronous
  - Reliable*(no duplicate, no missed msg to send, one-time delivery)*

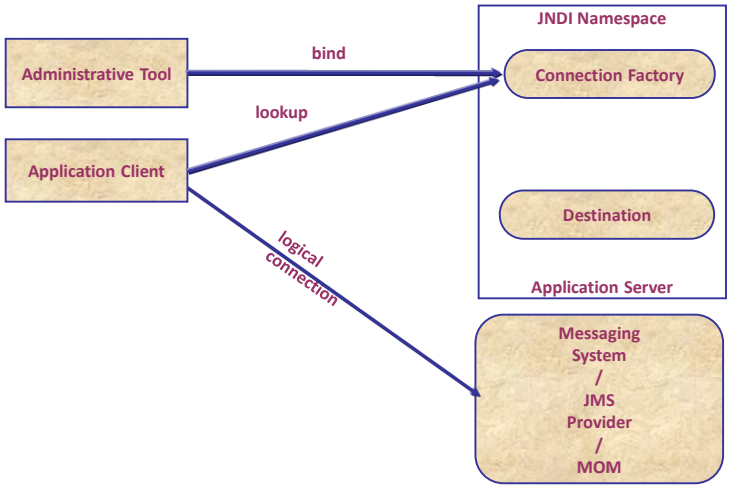
JMS Architecture



- JMS Provider: A messaging system that implements the JMS interfaces and provides administrative and control features
- Clients: Java applications that send or receive JMS messages. A message sender is called the Producer, and the recipient is called a Consumer
- Messages: Objects that communicate information between JMS clients
- Administered objects: Preconfigured JMS objects created by an administrator for the use of clients.

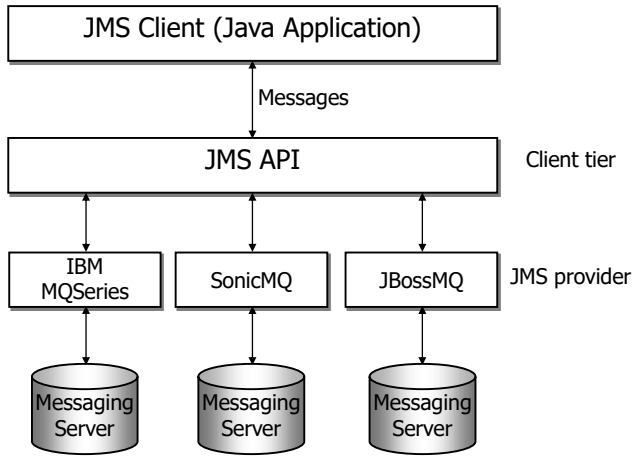
5

JMS Participants

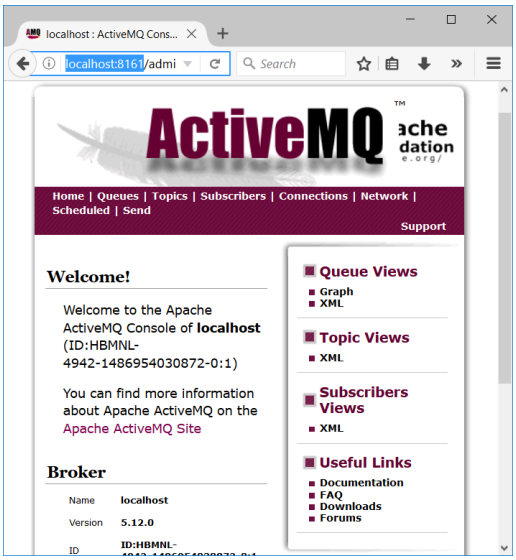


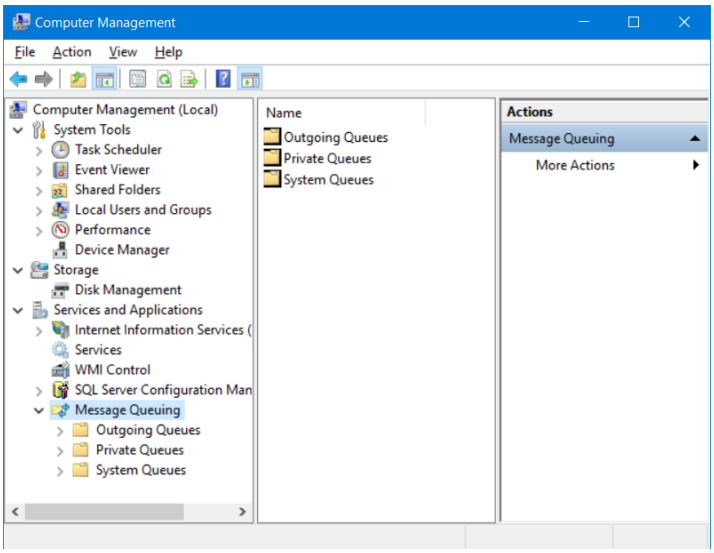
6

JMS – Connection From Client



Sample MOM Control Panel





Message Structure

- Data type is used to communication between software components/ applications.
- JMS messages have a specific format and have 3 parts:
  - Header
  - Properties (optional)
  - Body (optional)

9

Message Header

- Contains a number of fields that both the JMS clients and providers can use to identify and to route messages

Header Fields	Description
JMS Destination	Determine a destination store messages
JMSDeliveryMode	Determine a mode delivered message to consumers
JMSExpiration	The life cycle of message on destination
JMSPriority	Priority of message in deliver process.
JMSMessageID	A unique identifier of Message
JMSTimestamp	The sending time of message
JMSCorrelationID	A identifier message correlate with current message (Ex: message A is replied from message B)
JMSReplyTo	A responsing mode
JMSType	A message structure.
JMSRedelivered	Determine message delivered over once times

10

Message Properties

- The parameters required by a message are specified using message properties
- Provide compatibility with other messaging systems and also creating message selectors
- All message properties are pre-fixed with JMS
- Some predefined properties are included in the JMS API.
- All the header fields such as JMSMessageID and JMSDestination are instances of message properties
- Can use either predefined properties or user-defined properties in your JMS application
- Developer can define message properties using methods like setStringProperty(), setDoubleProperty(), and setShortProperty() of javax.jms.Message interface

11

Message Bodies

- The body of a message is nothing but the contents of the message. To send or receive messages in different formats, JMS defines 5 message body formats which are as follows

Message Type	Usage
TextMessage	For sending data in the form of a text.
MapMessage	Sending data in the form of key,value pairs.
BytesMessage	Sending binary data.
StreamMessage	Sending Java primitives as a stream of values.
ObjectMessage	Sending a Serializable Java Object as a message.
Message	Sending a message with only JMS headers and properties with no body associated with it.

12

Transaction

As an example, consider Code Snippet 1:

```
QueueConnection qConn =
                    qcf.createQueueConnection();
QueueSession session = qConn.createQueueSession
              (false, Session.AUTO_ACKNOWLEDGE);
```

A boolean value of true means that all the messages will be sent only if the commit() method will be called.

To check whether all the messages, which will be sent or received, will be transacted or not.

13

Transaction

```
    …
    QueueSession session=null;
    try {
    session = qConn.createQueueSession(true,
Session.AUTO_ACKNOWLEDGE);
    Queue queue = (Queue) //jndi lookup
    QueueSender sender = session.createSender(queue);
    TextMessage msg1 = session.createTextMessage();
    TextMessage msg2 = session.createTextMessage();
    msg1.setText("Some message 1");
    msg2.setText("Some message 2");
      session.commit();
    }catch(Exception e) {
      session.rollback();
    }
    …
```

As soon as the commit() or rollback() invocation completes, it signifies the end of one transaction and all the operations after this would automatically start in a new transaction till another commit() or rollback() doesn't complete the transactions.

14

Message Acknowledged

- Until a JMS message has been acknowledged, it is not considered to be successfully consumed.
- For the successful comsumption of a message, a client need to acknowledge.
- In transacted session, acknowledge happen automatically when a transaction is commited.
- In non-transacted session, message acknowledge depend on the value of the second parameter pass to the createQueueSession() or createTopicSession() method.

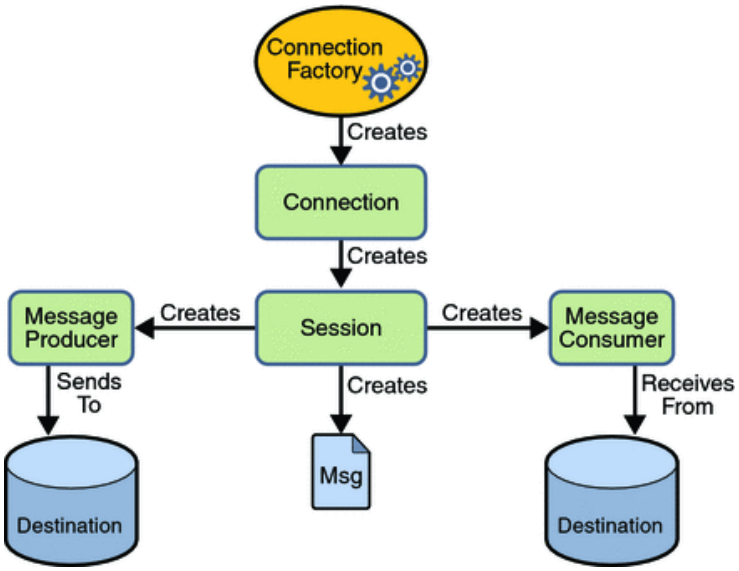
15

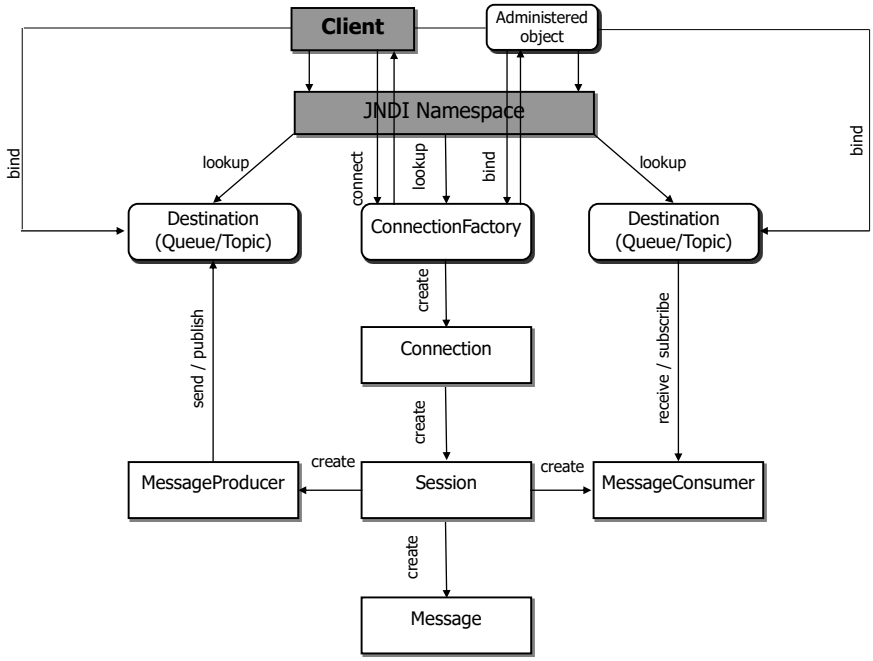
Message acknowledge modes

- **AUTO_ACKNOWLEDGE**
  - JMS session is responsible for automatically acknowledging the message received
  - Acknowledge synchronous after the receive() method call or asynchronous after the onMessage() callback method call.
- **CLIENT_ACKNOWLEDGE**
  - JMS Client is responsible for manually acknowledging the message using the acknowledge() method after sucessfully consuming the message.
  - Acknowleding even a single message in the session causes all the messages to be acknowledged automatically.
- **DUPS_OK_ACKNOWLEDGE**:
  - The JMS session can lazily acknowledge for message after they are consumed. (The consumer can tolerate duplicate message)

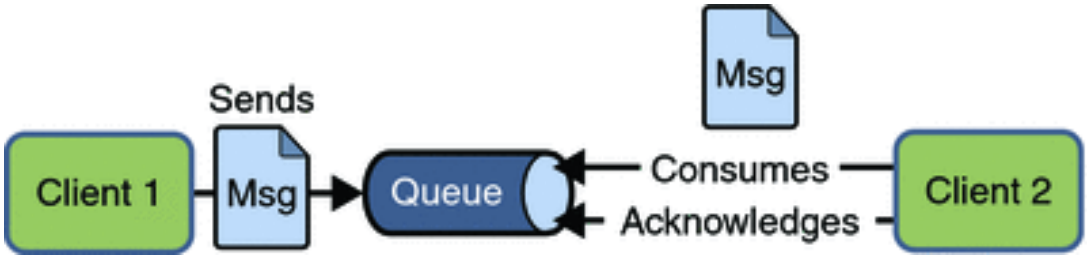
16

JMS Programming model

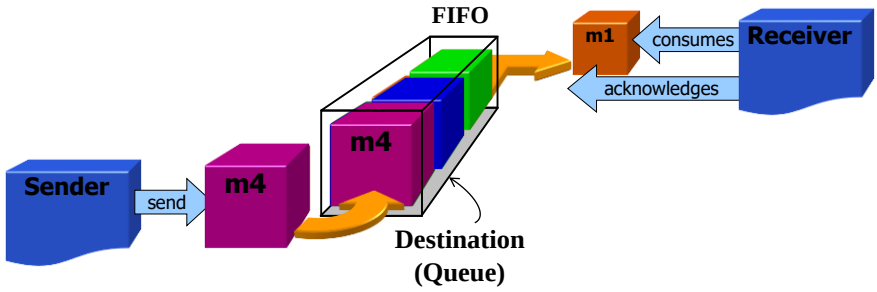




Point to point messaging model



- Producer sends the message to a specified queue within JMS provider and the only one of the consumers who listening to that queue receives that message.



Queue sender with JAVA – ActiveMQ

```
public static void main(String[] args) throws Exception{
    BasicConfigurator.configure();
    Properties settings=new Properties();
    settings.setProperty(Context.INITIAL_CONTEXT_FACTORY,
            "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
    settings.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");
    Context ctx=new InitialContext(settings);
    Object obj=ctx.lookup("ConnectionFactory");
    ConnectionFactory factory=(ConnectionFactory)obj;
    Destination destination=(Destination) ctx.lookup("dynamicQueues/thanthidet");
    Connection con=factory.createConnection("admin","admin");
    con.start();
    Session session=con.createSession(
            /*transaction*/false,
            /*ACK*/Session.AUTO_ACKNOWLEDGE
            );
    MessageProducer producer = session.createProducer(destination);
    Message msg=session.createTextMessage("hello mesage from ActiveMQ");
    producer.send(msg);
    session.close();
    con.close();
    System.out.println("Finished...");
}
```

Queue receiver JAVA – ActiveMQ

```
public static void main(String[] args) throws Exception {
    BasicConfigurator.configure();
    Properties props = new Properties();
    props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
            "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
    props.setProperty(Context.PROVIDER_URL,"tcp://localhost:61616");
    InitialContext jndi= new InitialContext(props);
    ConnectionFactory conFactory = (ConnectionFactory) jndi.lookup("ConnectionFactory");
    Connection connection = conFactory.createConnection();
    connection.start();
    Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
    Queue destination = (Queue) jndi.lookup("dynamicQueues/TEOLEO");
    MessageConsumer consumer = session.createConsumer(destination);
    consumer.setMessageListener(new MessageListener() {
        public void onMessage(Message msg) {
            try {
                if(msg instanceof TextMessage){
                    TextMessage tmsg=(TextMessage)msg;
                    System.out.println("----------"+tmsg.getText());
                }
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    });
```

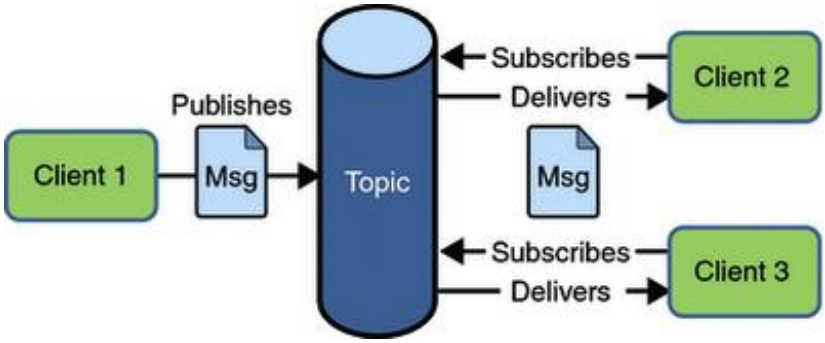
Queue sender with C# – ActiveMQ

```
static void Main(string[] args)
{
    try
    {
        IConnectionFactory fac = new ConnectionFactory("tcp://localhost:61616");
        IConnection con = fac.CreateConnection();
        con.Start();
        ISession session = con.CreateSession(AcknowledgementMode.AutoAcknowledge);

        ActiveMQQueue queue = new ActiveMQQueue("TEOLEO");

        IMessageProducer producer = session.CreateProducer(queue);
        IMessage msg = new ActiveMQTextMessage("hola pakon " + i);
        producer.Send(msg);
        Console.WriteLine("send ok ");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.StackTrace);
    }
    Console.ReadLine();
}
```

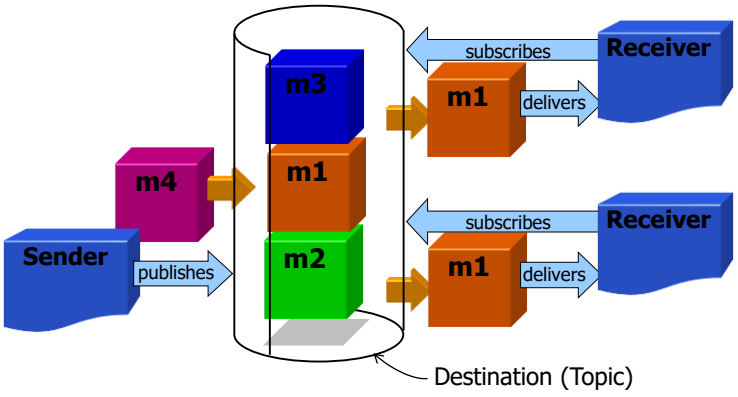
Queue receiver C# – ActiveMQ

```
static void Main(string[] args){
    try{
        IConnectionFactory fac = new ConnectionFactory("tcp://localhost:61616");
        IConnection con = fac.CreateConnection();
        con.Start();
        ISession session = con.CreateSession(AcknowledgementMode.AutoAcknowledge);
        ActiveMQQueue queue = new ActiveMQQueue("TEOLEO");
        IMessageConsumer consumer = session.CreateConsumer(queue);
        consumer.Listener += consumer_Listener;
    }
    catch (Exception ex){
        Console.WriteLine(ex.StackTrace);
    }
    Console.ReadLine();
}

static void consumer_Listener(IMessage message){
    if (message is ActiveMQTextMessage){
        ActiveMQTextMessage msg = (ActiveMQTextMessage)message;
        Console.WriteLine(msg.Text);
    }
}
```

Publisher Subscriber Model



- Publisher publishes the message to a specified topic within JMS provider and all the subscribers who subscribed for that topic receive the message. Note that only the active subscribers receive the message.



Pub/Sub sender JAVA

```
public static void main(String[] args) throws Exception {
    Properties props = new Properties();
    props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
            "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
    props.setProperty(Context.PROVIDER_URL,"tcp://localhost:61616");
    InitialContext jndi= new InitialContext(props);
    ConnectionFactory conFactory = (ConnectionFactory) jndi.lookup("TopicConnectionFactory");
    Connection connection = conFactory.createConnection();
    connection.start();
    Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
    Topic topic=(Topic)jndi.lookup("dynamicTopics/teoTopic");

    MessageProducer producer = session.createProducer(topic);
    TextMessage message = session.createTextMessage("Hello World! ");
    producer.send(message);
    connection.close();
}
```

Pub/Sub receiver with asynchronous JAVA

```
public static void main(String[] args) throws Exception {
    Properties props = new Properties();
    props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
            "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
    props.setProperty(Context.PROVIDER_URL,"tcp://localhost:61616");
    InitialContext jndi= new InitialContext(props);
    ConnectionFactory conFactory = (ConnectionFactory) jndi.lookup("TopicConnectionFactory");
    Connection connection = conFactory.createConnection();
    connection.start();
    Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
    Topic topic=(Topic)jndi.lookup("dynamicTopics/teoTopic");
    MessageConsumer consumer=session.createConsumer(topic);
    consumer.setMessageListener(new MessageListener() {
        public void onMessage(Message msg) {
            try {
                if(msg instanceof TextMessage){
                    TextMessage tmsg=(TextMessage)msg;
                    System.out.println(tmsg.getText());
                }
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    });
}
```

Pub/Sub sender C#

```
using Apache.NMS;
using Apache.NMS.ActiveMQ;
using Apache.NMS.ActiveMQ.Commands;

static void Main(string[] args)
{
    String url = "tcp://localhost:61616";
    IConnectionFactory factory = new ConnectionFactory(url);
    IConnection connection = factory.CreateConnection();
    connection.Start();
    ISession session = connection.CreateSession(
        AcknowledgementMode.AutoAcknowledge);
    ActiveMQTopic topic = new ActiveMQTopic("teoTopic");
    IMessageProducer producer = session.CreateProducer(topic);
    ITextMessage txtMessage = session.CreateTextMessage("Hello world");
    producer.Send(txtMessage);
    connection.Close();
}
```

Pub/Sub receiver with asynchronous C#

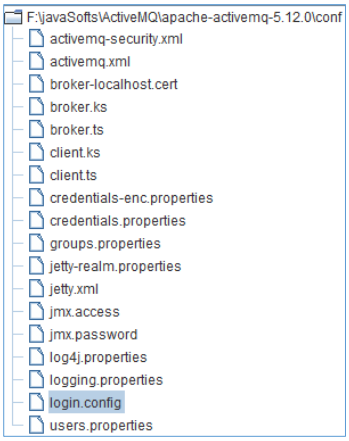
```
static void Main(string[] args)
{
    String url = "tcp://localhost:61616";
    IConnectionFactory factory = new ConnectionFactory(url);
    IConnection connection = factory.CreateConnection();
    connection.Start();
    ISession session = connection.CreateSession(
        AcknowledgementMode.AutoAcknowledge);
    ActiveMQTopic topic = new ActiveMQTopic("teoTopic");

    IMessageConsumer consumer = session.CreateConsumer(topic);
    consumer.Listener += consumer_Listener;
    Console.ReadLine();
}

static void consumer_Listener(IMessage message)
{
    ITextMessage textMessage = message as ITextMessage;
    Console.WriteLine(textMessage.Text);
}
```

Displaying thread

```
delegate void SetTextCallback(string text);
private void SetText(string text)
        {
            // InvokeRequired required compares the thread ID of the
            // calling thread to the thread ID of the creating thread.
            // If these threads are different, it returns true.
            if (this.MessagesRichTextBox.InvokeRequired)
            {
                SetTextCallback callback = new SetTextCallback(SetText);
                this.Invoke(callback, new object[] { text });
            }
            else
                this.MessagesRichTextBox.AppendText(text + "\n");
        }
```

Security

- ActiveMQ 4.x and greater provides pluggable security through various different providers.
- The most common providers are:
  - JAAS for authentication
  - A default authorization mechanism using a simple XML configuration file.
- Typically you configure JAAS using a config file and set the java.security.auth.login.config system property to point to it.
- If no system property is specified then by default the ActiveMQ JAAS plugin will look for login.config on the classpath and use that



Authentication settings

%ACTIVEMQ_HOME%\conf\login.config

```
activemq {
    org.apache.activemq.jaas.PropertiesLoginModule required
    org.apache.activemq.jaas.properties.user="users.properties"
    org.apache.activemq.jaas.properties.group="groups.properties"
    reload=true;
};
```

%ACTIVEMQ_HOME%\conf\groups.properties

```
#group_name=users_list
admins=admin,teo
users=ty
dhcn=men
guests=guest
```

%ACTIVEMQ_HOME%\conf\users.properties

```
#username=password
admin=admin
teo=123
ty=456
men=123
```

%ACTIVEMQ_HOME%\conf\activemq.xml

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerN
    <!--Enable it for security-->
    <!---->
    <plugins>
        <jaasAuthenticationPlugin configuration="activemq" />
        <authorizationPlugin>
    </plugins>
```

Simple Authentication Plugin

- If you have modest authentication requirements you can use SimpleAuthenticationPlugin. With this plugin you can define users and groups directly in the broker's XML configuration.

```
<plugins>
    <!-- Configure authentication; Username, passwords and groups -->
    <simpleAuthenticationPlugin anonymousAccessAllowed="true">
        <users>
            <authenticationUser username="system" password="manager" groups="users,admins"/>
            <authenticationUser username="user" password="password" groups="users"/>
            <authenticationUser username="guest" password="password" groups="guests"/>
        </users>
    </simpleAuthenticationPlugin>
```

- Users and groups defined in this way can be later used with the appropriate authorization plugin

Authorization (1)

- In ActiveMQ we use a number of operations which you can associate with user roles and either individual queues or topics or you can use wildcards to attach to hierarchies of topics and queues.

Operation	Description
read	You can browse and consume from the destination
write	You can send messages to the destination
admin	You can lazily create the destination if it does not yet exist. This allows you fine grained control over which new destinations can be dynamically created in what part of the queue/topic hierarchy

Authorization (2)

```
<!--  Lets configure a destination based authorization mechanism -->
<authorizationPlugin>
    <map>
        <authorizationMap>
            <authorizationEntries>
                <authorizationEntry queue=">" read="admins" write="admins" admin="admins" />
                <authorizationEntry queue="USERS.>" read="users" write="users" admin="users" />
                <authorizationEntry queue="GUEST.>" read="guests" write="guests,users"
                                    admin="guests,users" />
                <authorizationEntry queue="TEST.Q" read="guests" write="guests" />
                <authorizationEntry topic=">" read="admins" write="admins" admin="admins" />
                <authorizationEntry topic="USERS.>" read="users" write="users" admin="users" />
                <authorizationEntry topic="GUEST.>" read="guests" write="guests,users"
                                    admin="guests,users" />
                <authorizationEntry topic="ActiveMQ.Advisory.>" read="guests,users"
                                    write="guests,users" admin="guests,users"/>
            </authorizationEntries>

            <tempDestinationAuthorizationEntry>
                <tempDestinationAuthorizationEntry read="admin" write="admin" admin="admin"/>
            </tempDestinationAuthorizationEntry>
        </authorizationMap>
    </map>
</authorizationPlugin>
```

Message level authorization

- Sometimes it can be usefull to add authorization at the message level instead of at the connection level.
- This is done by creating a java class *MessageAuthorizationPolicy* thus forcing you to implement the method

```
boolean isAllowedToConsume(ConnectionContextcontext, Message message)
```

- Here you can implement your own criteries of authorization.
- To put the authorization policy in good use you need to install and configure it into your ActiveMQ setup. Basicly it is done via these steps:
  - Compile your class into a JAR file
  - Put the JAR in ApacheMQ's lib folder
  - Add a proper messageAuthorizationPolicy element to your configuration
  - Restart ActiveMQ