**Java Message Services**

JMS (Java Message Service) is an API that provides the facility to create, send and read messages. It provides loosely coupled reliable and asynchronous communication.

JMS is also known as a messaging service.

**Why to use JMS?**

JMS is a messaging service which provides reliable and asynchronous communication to implement the messaging system between Java based applications and software components. The JMS API provides set of interfaces to communicate with Java programs and also defines standard messaging protocols to support the Java programming language.

**Features**

* It provides support for messaging applications in [J2EE](http://www.oracle.com/technetwork/java/javaee/appmodel-135059.html) (Java 2 Platform, Enterprise Edition) technology to interact with other applications.
* It provides a common interface to communicate with messaging implementations.
* The application developed with JMS API, can be deployed in any JMS provider software.
* Developers can easily create messaging enterprise applications by quickly learning JMS API.

In this tutorial will learn to implement JMS model(P2P,Pub-Sub) using Apache Artemis MQ and how to perform operations using filters.

**Software required-**

* Java 1.8 or higher(11 is preferred)\
* IDE
* Apache ActiveMQ Artemis

**Apache ActiveMQ Artemis**

Apache ActiveMQ® is the most popular open source, multi-protocol, Java-based message broker. It supports industry standard protocols so users get the benefits of client choices across a broad range of languages and platforms. Connect from clients written in JavaScript, C, C++, Python, .Net, and more.

#### ActiveMQ Artemis High-performance, non-blocking architecture for the next generation of messaging applications.

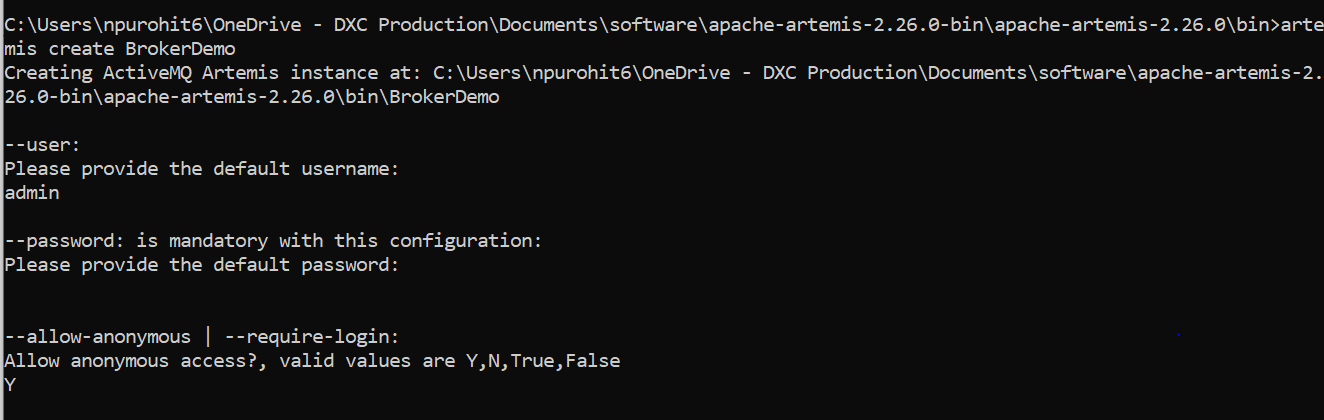
**Installation**

1)Download the zip file from– [ActiveMQ (apache.org)](https://activemq.apache.org/components/artemis/download/)

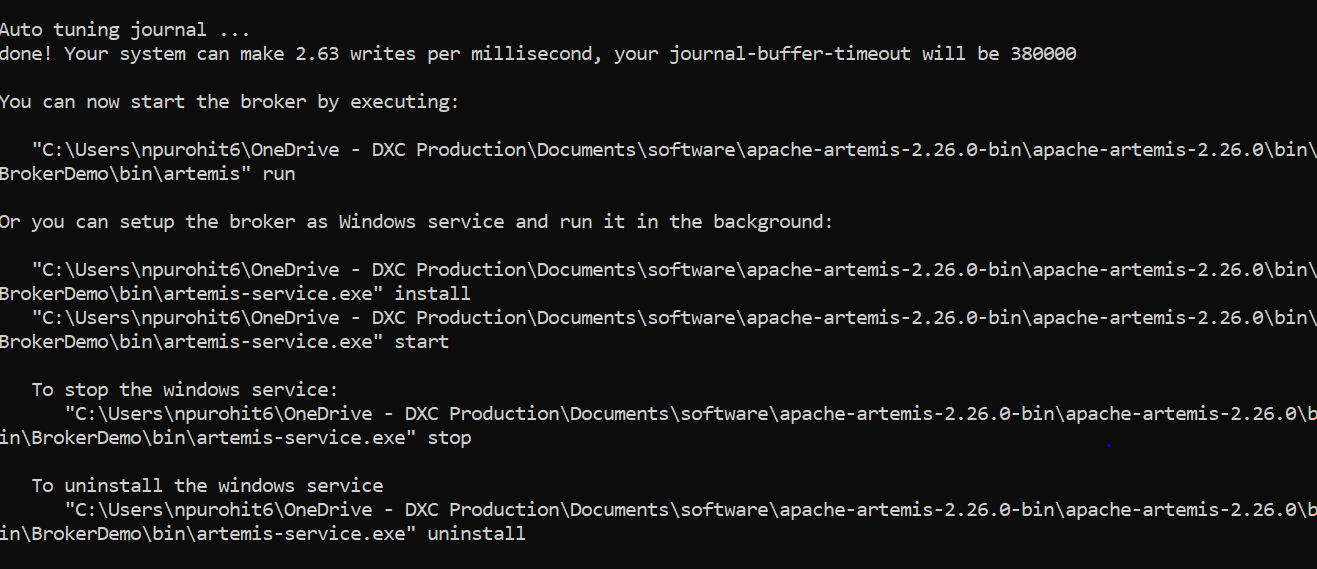
2) Once downloaded, extract it to some convenient folder location, e.g.: D:\apache-artemis-2.26.0 referred as ${ARTEMIS\_HOME}

3)Now go to bin folder and create broker instance using command-

artemis create mybroker



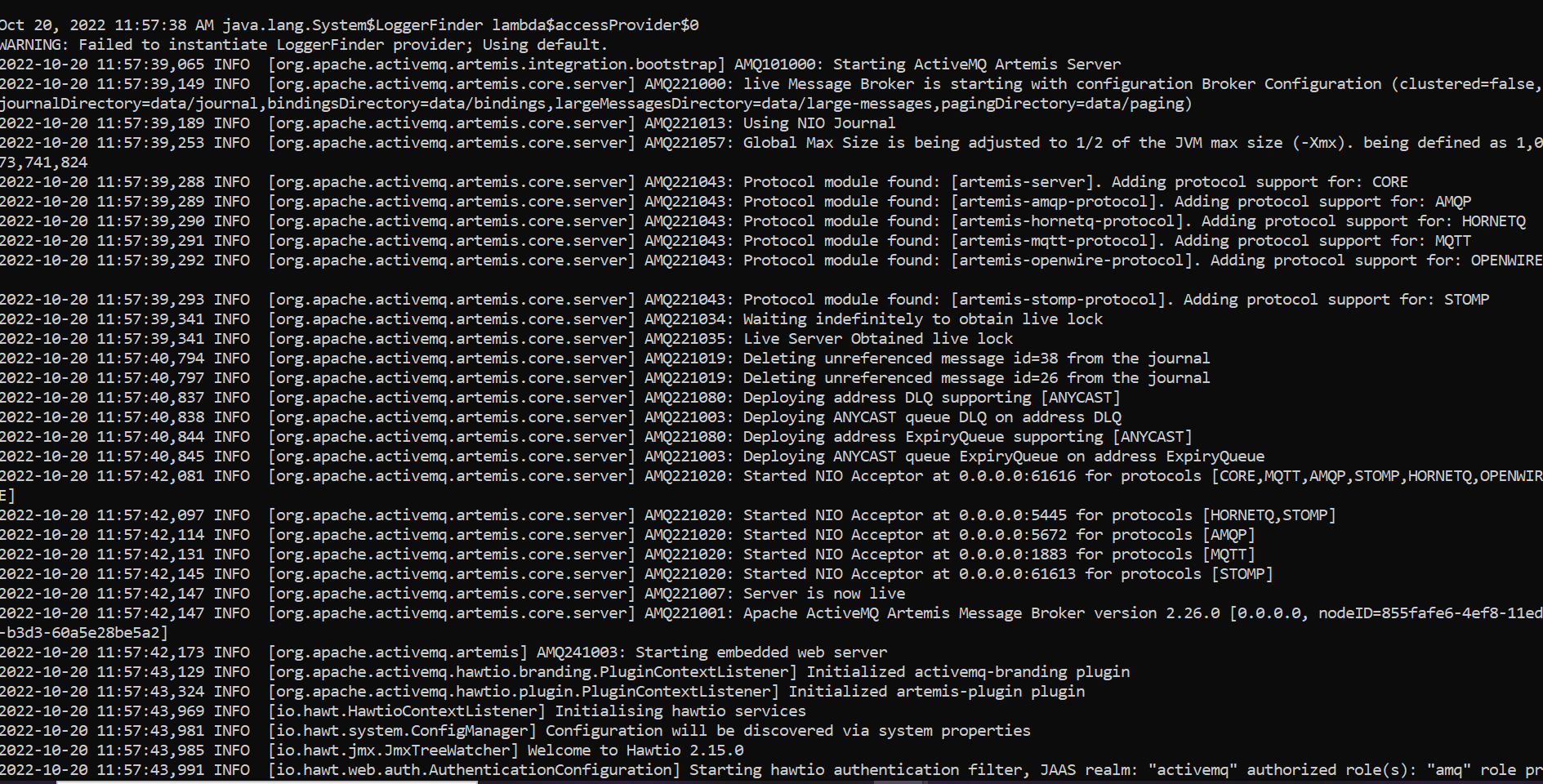
4)provide username and password and broker instance will start running-



A broker instance is a directory containing all the configuration and runtime data, such as logs and data files.

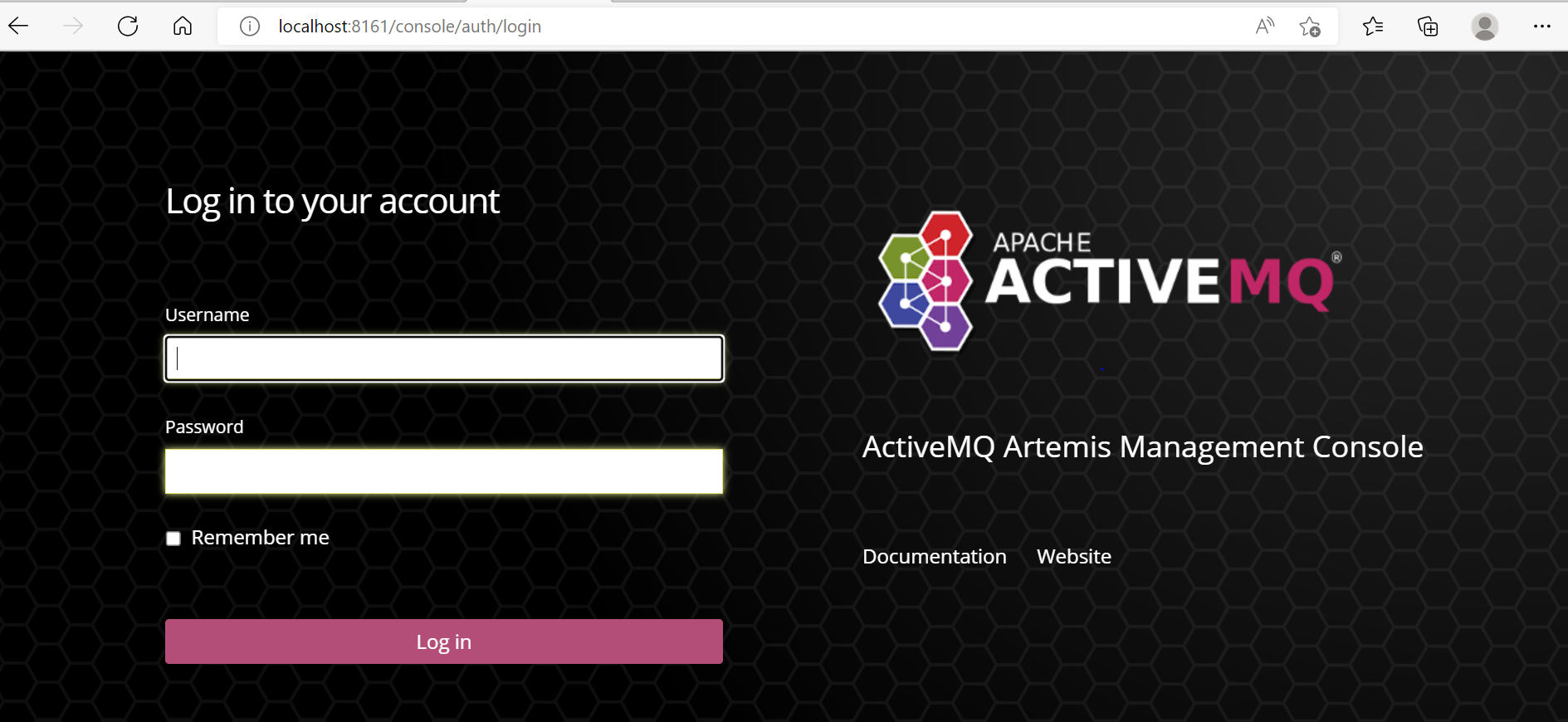
5)Then navigate to folder created in bin folder having same name as that of broker created and navigate to bin folder and type following command to run artemis

${ARTEMIS\_HOME}\bin\BrokerDemo\bin\artemis run



Verify your Broker Instance by opening the console in the browser: <http://localhost:8161/console>

You will see the below login page-



Now broker instance is up and running ,we can proceed with implementation of messaging models.

## **Messaging Domains**

There are two types of messaging domains in JMS.

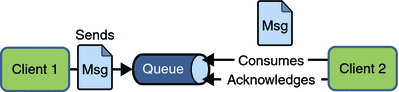
1. Point-to-Point Messaging Domain
2. Publisher/Subscriber Messaging Domain

## **1) Point-to-Point (PTP) Messaging Domain**

In PTP model, one message is delivered to one receiver only. Here, Queue is used as a message oriented middleware (MOM).

The Queue is responsible to hold the message until receiver is ready.

In PTP model, there is no timing dependency between sender and receiver.



1. **To implement P2P model-..c./**
2. **000000000000000000000**

1)Create a maven project and add following dependency in pom.xml-

<dependency>

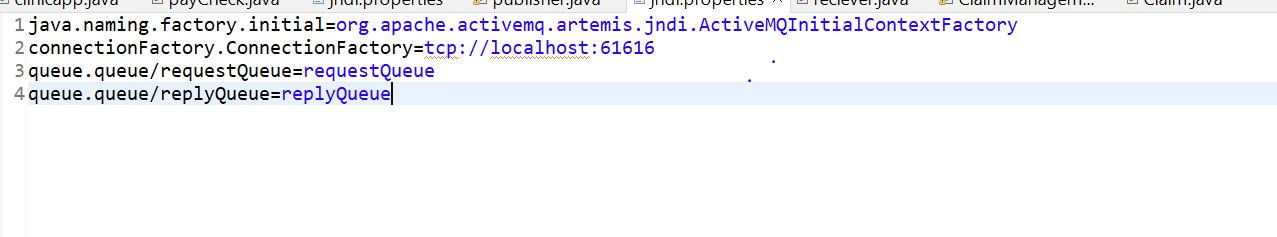
<groupId>org.apache.activemq</groupId>

<artifactId>artemis-jms-client-all</artifactId>

<version>2.6.4</version>

</dependency>

2)Create jndi.properties in src/main/resourses folder-



3)Create a model class as per your logic,here model class patient has been created with following attributes-



Class must be serializable.

4)Create a sender application-

* JNDI lookup of the  queue-

InitialContext ctx = **new** InitialContext();

Queue requestQueue = (Queue) ctx.lookup("queue/requestQueue");

* Creating a connection -

ActiveMQConnectionFactory cf=**new** ActiveMQConnectionFactory();

JMSContext js=cf.createContext();

* Creating producer

JMSProducer pd = js.createProducer();

* Creating ObjectMessage setting entity object-

ObjectMessage om=js.createObjectMessage();

patient p=**new** patient();

p.setId(1);

p.setName("Abc");

p.setPayDue(1000);

p.setPaymentString(20000);

om.setObject(p);

* Sending messages to the queue-

pd.send(requestQueue, om);

This is how sample sender application would look like-



5)Creating receiver application-

We will be receiving message by registering amessage listener object with a message consumer by calling the *setMessageListener()* method and passing in the message listener as a parameter.

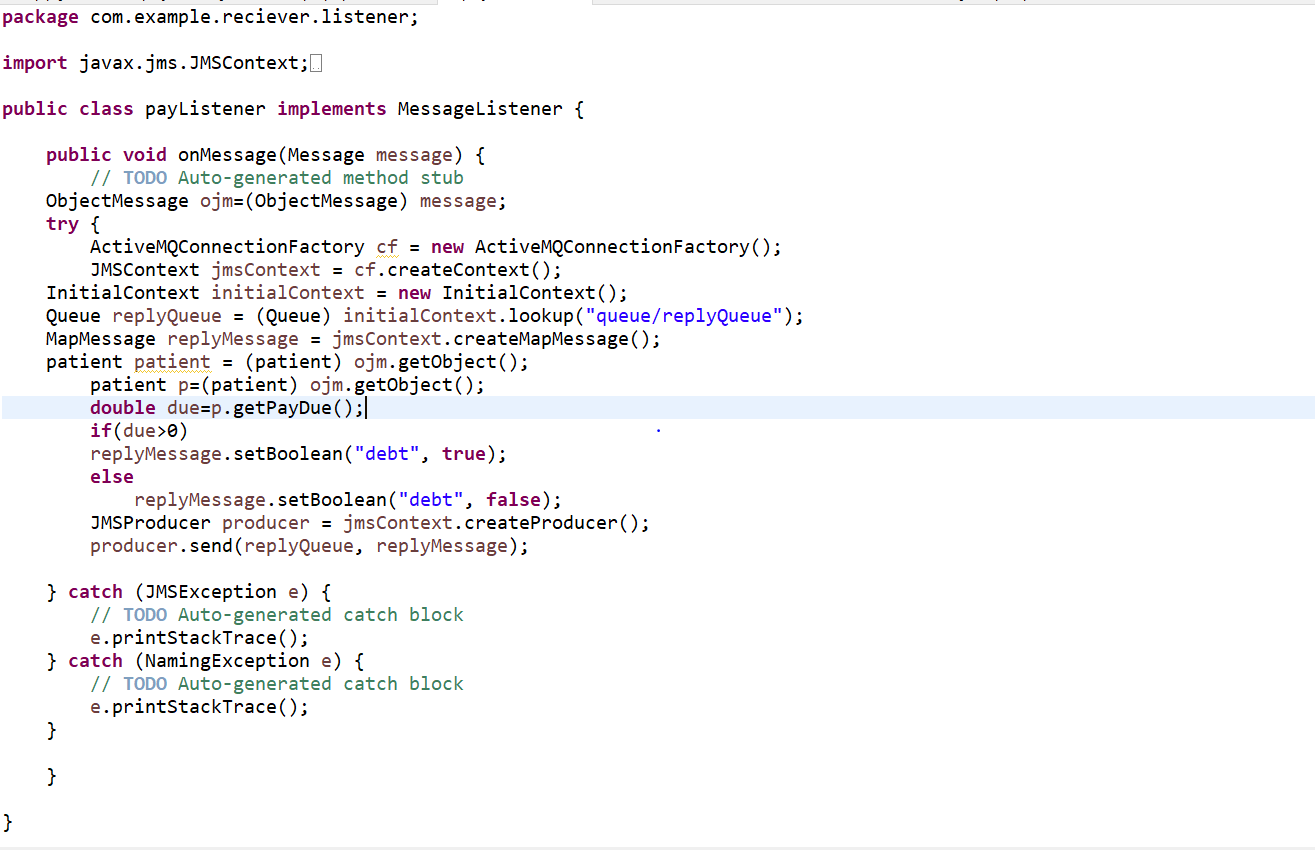
* Create objects similar to that of producer
* Now Instead of creating *producer* create *consumer* this time-

JMSConsumer cons = js.createConsumer(requestQueue);

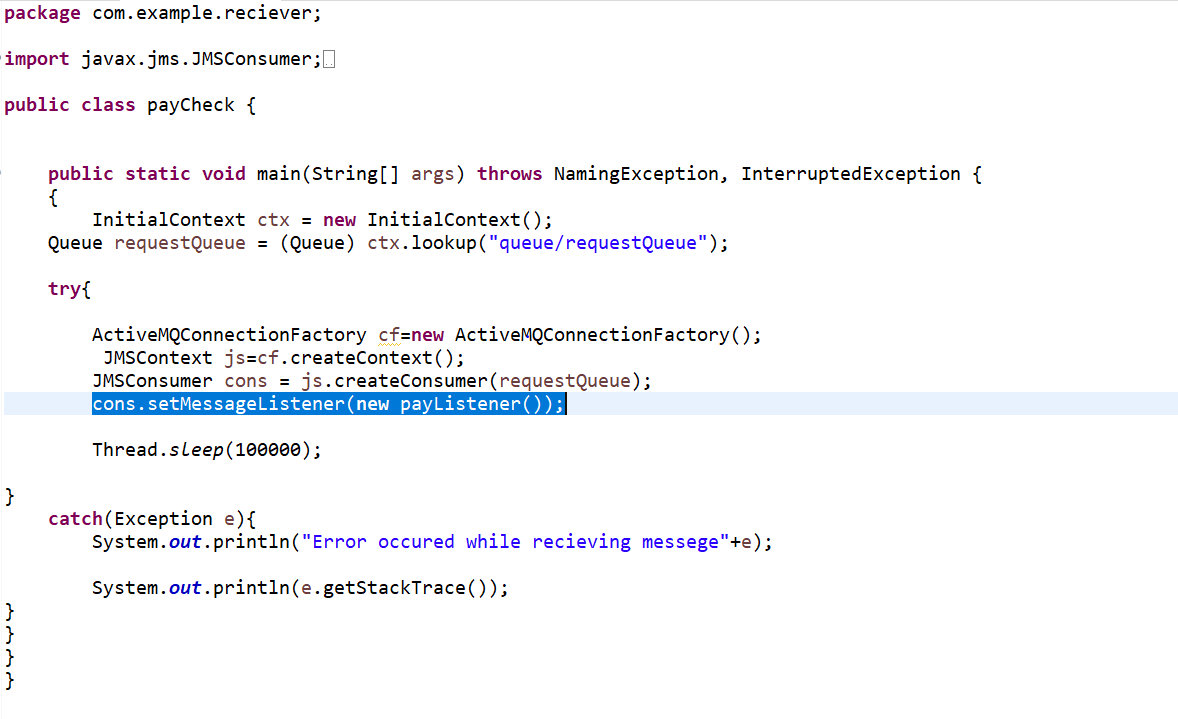
* Register message listener in consumer-

cons.setMessageListener(**new** payListener());

* **Message-Listener’s responsibility:** Whenever a message comes, the onMessage method is getting called automatically. The onMessage method converts the incoming message to a specific type as TextMessage/MapMessage etc and displays its content.
* Create message listener-

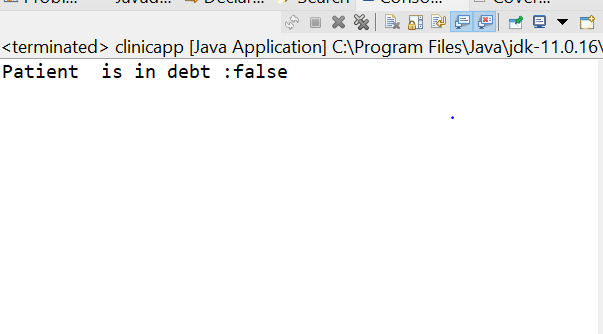


* Receiver application should look like following-



We have created sender application and receiver application.

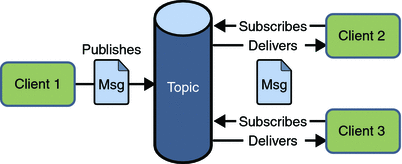
Now run receiver application and then sender application to get output-



## **2) Publisher/Subscriber (Pub/Sub) Messaging Domain**

In Pub/Sub model, one message is delivered to all the subscribers. It is like broadcasting. Here, Topic is used as a message oriented middleware that is responsible to hold and deliver messages.

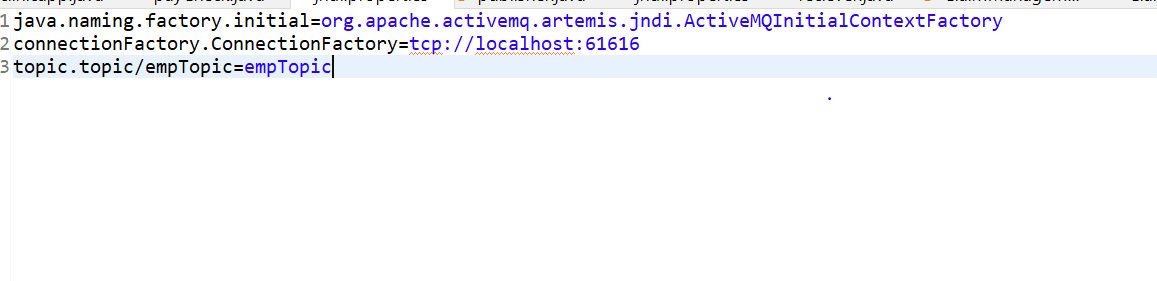
In PTP model, there is timing dependency between publisher and subscriber.



**To implement Pub/Sub model**

1)Create maven application and add ActiveMQ client dependency in pom.xml

2)Create model class and jndi.properties as done in p2p model-



3)Create a publisher application-

* Getting context and looking up *Topic*-

InitialContext ctx = **new** InitialContext();

Topic t=(Topic) ctx.lookup("topic/empTopic");

* Creating a connection -

ActiveMQConnectionFactory cf=**new** ActiveMQConnectionFactory();

JMSContext js=cf.createContext();

* Sending message by publisher-

Patient p=**new** Patient();

p.setId(1);

p.setName("Abc");

p.setPayDue(1000);

p.setPaymentString(20000);

js.createProducer().send(t, p);

sample publisher application-



4)Creating subscriber application-

InitialContext ctx = **new** InitialContext();

Topic t=(Topic) ctx.lookup("topic/empTopic");

**try**{

ActiveMQConnectionFactory cf=**new** ActiveMQConnectionFactory();

JMSContext js=cf.createContext();

JMSConsumer cons = js.createConsumer(t);

Message m=cons.receive();

Patient p = m.getBody(Patient.**class**);

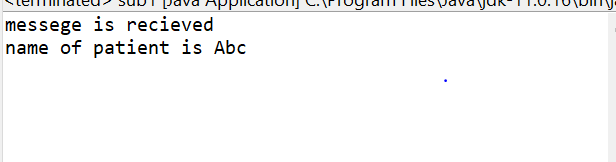
System.***out***.println("messege is recieved");

System.***out***.println("name of patient is "+p.getName());

}



5)Now run both the application to get the output



**Message Filtering in JMS**

Quite often messages delivered to the receiver need to be filtered based on certain criteria. This is why filtering is required.

## Why Message Selectors?

When a message is broadcasted to many receivers, it is useful to place criteria into the Subscription to register specific interests. So only the interested-messages are delivered to the subscribers.

* A Message Selector is a String with syntax based on **SQL92** conditional expression.
* Only messages whose headers and properties match the selector are delivered.
* Message selectors can not reference the message body values.
* Message Selector is evaluated from left to right.
* Selector literals and operators are usually written in Upper Case, however, they are case insensitive.

Use below example to filter messages –

1)Create a maven project and add following-

1)Create a maven project and add following dependency in pom.xml-

<dependency>

<groupId>org.apache.activemq</groupId>

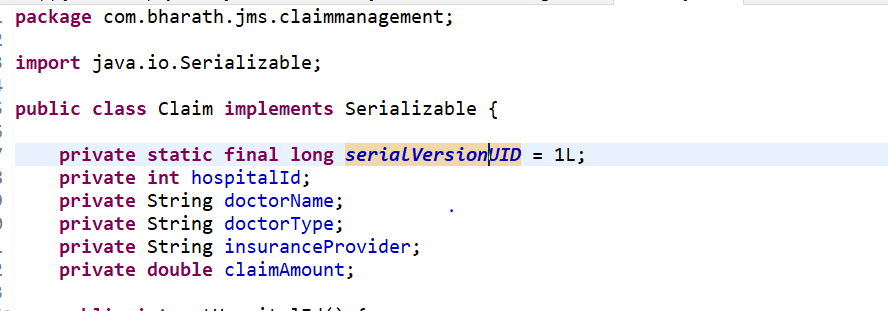
<artifactId>artemis-jms-client-all</artifactId>

<version>2.6.4</version>

</dependency>

2)Create jndi.properties in src/main/resourse filder

3)Create a model class as per your logic,in demo model class patient has been created with following attributes-



4)Create producer in similar way as done in previous application.

5)Now provide message filter while creating consumer-

JMSConsumer consumer = jmsContext.createConsumer(requestQueue, "doctorType IN ('gyna','psych') OR JMSPriority BETWEEN 5 AND 9");

Here we will be using *createConsumer()* method to create consumer but we will be using one with two arguments so that we can pass our filter

6)Sample code for creating consumer and producer-



7)Now run the application to get output.

Please note artemis server should be running while testing any of these applications.