

APACHE ACTIVEMQ COOKBOOK

Hot Recipes for the ActiveMQ Messaging Server



Apache
ACTIVEMQ



Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Apache ActiveMQ Cookbook

Contents

1	ActiveMQ "Hello World" Example	1
1.1	Introduction	1
1.2	How to setup ActiveMQ	1
1.3	Using ActiveMQ in a Java project (example using eclipse)	2
1.4	Output	5
1.5	Conclusion	7
2	ActiveMQ Monitoring Tutorial	8
2.1	Introduction	8
2.2	Monitor ActiveMQ JVM	8
2.3	Monitor ActiveMQ Web Console	9
2.4	Monitor ActiveMQ File System	10
2.5	Monitor ActiveMQ Application	10
2.6	Java Example	11
2.6.1	Technologies used	11
2.6.2	Data Model	11
2.6.3	Monitor Service	13
2.6.4	Quartz Job	15
2.6.5	Monitor Application	16
2.6.6	Monitor Application execution	17
2.7	Summary	17
2.8	Download the Source Code	17
3	ActiveMQ Best Practices Tutorial	18
3.1	Introduction	18
3.2	Install an ActiveMQ Server	18
3.3	Start the ActiveMQ Server	18
3.4	Monitor the ActiveMQ Server	18
3.5	Business Use Cases	19
3.6	Define JMS Message	19
3.6.1	Message Destination	19

3.6.2	Message Header	19
3.6.3	Message Body	19
3.6.4	Configure Virtual Topic	20
3.7	ActiveMQ Java Client Library	20
3.8	Publish Message Application	21
3.8.1	ActiveMQMessgeProducer	21
3.8.2	ActiveMQMessgeProducerTest	23
3.8.3	Execution Output	25
3.8.4	OnBoardNewCustomerApp	25
3.9	Consume Message Application	25
3.9.1	ActiveMQMessageConsumer	25
3.9.2	ActiveMQMessageConsumerMainApp	28
3.9.3	Execution Output	30
3.10	Integration with Spring JMS	31
3.10.1	Add Spring JMS dependency	31
3.10.2	Configure Spring Beans	31
3.10.3	MessageSender	33
3.10.4	BillingAppListener	34
3.10.5	SupportAppListener	35
3.10.6	ConfigBillingforNewCustomerApp	36
3.10.7	ConfigSupportforNewCustomerApp	37
3.10.8	Run as Distributed Systems	37
3.11	Integrating with Tomcat	38
3.11.1	Configure Tomcat Resource	38
3.11.2	Look up JNDI Resource	38
3.12	Common Problems	38
3.12.1	Slow Consumer Application	39
3.12.2	ActiveMQ Sends Unwanted Messages to Virtual Topic Queue	39
3.12.3	Exception Handler	39
3.13	Summary	39
3.14	Download the Source Code	39
4	ActiveMQ Distributed Queue Tutorial	40
4.1	Introduction	40
4.2	ActiveMQ Server Installation	40
4.3	Producer Java Application	40
4.3.1	MessageProducerApp	40
4.3.2	QueueMessageProducer	41
4.3.3	Export MessageProducerApp as a Jar	42

4.4	Consumer Java Application	42
4.4.1	MessageConsumerApp	42
4.4.2	QueueMessageConsumer	43
4.4.3	Common Utils	44
4.4.4	Export MessageConsumerApp as a Jar	46
4.5	Distributed Queue in a Static Network of Brokers	46
4.5.1	Configure a Static Network of Brokers	47
4.5.2	Verify the AMQ Brokers - Part I	47
4.5.3	Execute the Consumer Application	49
4.5.4	Execute the Publisher Application	50
4.5.5	Verify the AMQ Brokers - Part II	52
4.6	Distributed Queue in a Dynamic Network of Brokers	53
4.6.1	Configure a Dynamic Network of Brokers	54
4.6.2	Verify the AMQ Brokers - Part I	54
4.6.3	Execute the Consumer Application	55
4.6.4	Execute the Publisher Application	55
4.6.5	Verify the AMQ Brokers - Part II	56
4.7	Summary	58
4.8	Download the Source Code	58
5	ActiveMQ BrokerService Example	59
5.1	Introduction	59
5.2	Embed a broker in ActiveMQ	59
5.2.1	Embed a broker in ActiveMQ using Java code (using BrokerService API)	60
5.2.2	Embed a broker in ActiveMQ using Spring 2.0	62
5.2.3	Embed a broker in ActiveMQ using XBean	62
5.2.4	Embed a broker in ActiveMQ using BrokerFactory	63
5.3	Conclusion	65
6	ActiveMQ Load Balancing Example	66
6.1	Introduction	66
6.2	The Component Diagram	66
6.3	Technologies used	67
6.4	Start two ActiveMQ Brokers	67
6.4.1	Configure ActiveMQ with Non-Default Port	67
6.4.1.1	Update <code>activemq.xml</code>	67
6.4.1.2	Update <code>jetty.xml</code>	68
6.4.2	Start ActiveMQ Brokers	68
6.5	Producer Load Balancing Example	69

6.5.1	Dependency	69
6.5.2	Constants	70
6.5.3	Spring configuration	70
6.5.4	MessageSender	72
6.5.5	MessageProducerApp	73
6.5.6	Execute MessageProducerApp	74
6.6	Consumer Load Balancing Example	74
6.6.1	MessageConsumerWithPrefetch	74
6.6.2	MessageConsumerApp	75
6.6.3	Execute MessageConsumerApp in Eclipse	76
6.6.4	Execute MessageConsumerApp via Jar command	76
6.6.5	Summary	77
6.6.6	Things to consider	77
6.7	Conclusion	77
6.8	Download the Source Code	78
7	ActiveMQ Failover Example	79
7.1	Introduction	79
7.2	ActiveMQ Server Installation	79
7.3	ActiveMQ Server Configuration	79
7.3.1	Two Standalone ActiveMQ Brokers	80
7.3.2	Master/Slave ActiveMQ Brokers	82
7.3.3	Network of Brokers	84
7.3.3.1	Static Network of Brokers	84
7.3.3.2	Dynamic Network of Brokers	87
7.4	Create Java Client Applications	92
7.4.1	Common Data	92
7.4.2	QueueMessageProducer	94
7.4.3	MessageProducerApp	96
7.4.4	QueueMessageConsumer	96
7.4.5	MessageConsumerApp	97
7.5	Demo Time	98
7.5.1	Two Standalone Brokers	98
7.5.2	Master/Slave Brokers	100
7.5.3	Static Network of Brokers	101
7.5.4	Dynamic Network of Brokers	102
7.6	Summary	106
7.7	Download the Source Code	106

8	ActiveMQ Advisory Example	107
8.1	Introduction	107
8.2	Configuration for ActiveMQ Advisory	107
8.3	Using ActiveMQ Advisory	108
8.3.1	Example 1 - Using ActiveMQ Advisory in a simple Java project (example using Eclipse)	108
8.3.2	Example 2 - Monitoring ActiveMQ for a number of events (example using Eclipse)	111
8.4	Conclusion	118
9	ActiveMQ File Transfer Example	119
9.1	Introduction	119
9.2	JMS Message Type	119
9.3	Business Use Case	120
9.4	File Transfer Application	120
9.4.1	Technologies Used	120
9.4.2	Dependency	121
9.4.3	Constants	121
9.4.4	File Manager	121
9.4.5	Save File Application	122
9.4.6	Message Consumer	122
9.4.7	Send File Application	125
9.4.8	Message Producer	125
9.5	Demo Time	128
9.5.1	Start ConsumeFileApp	128
9.5.2	Start SendFileApp	128
9.5.3	Verify the Transferred Files	129
9.6	Summary	130
9.7	Download the Source Code	130

Copyright (c) Exelixis Media P.C., 2017

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

Preface

Apache ActiveMQ is an open source message broker written in Java together with a full Java Message Service (JMS) client. It provides "Enterprise Features" which in this case means fostering the communication from more than one client or server. Supported clients include Java via JMS 1.1 as well as several other "cross language" clients. The communication is managed with features such as computer clustering and ability to use any database as a JMS persistence provider besides virtual memory, cache, and journal persistency.

The ActiveMQ project was originally created by its founders from LogicBlaze in 2004, as an open source message broker, hosted by CodeHaus. The code and ActiveMQ trademark were donated to the Apache Software Foundation in 2007, where the founders continued to develop the codebase with the extended Apache community.

ActiveMQ employs several modes for high availability, including both file-system and database row-level locking mechanisms, sharing of the persistence store via a shared filesystem, or true replication using Apache ZooKeeper. A robust horizontal scaling mechanism called a Network of Brokers, is also supported out of the box. In the enterprise, ActiveMQ is celebrated for its flexibility in configuration, and its support for a relatively large number of transport protocols, including OpenWire, STOMP, MQTT, AMQP, REST, and WebSockets.

ActiveMQ is used in enterprise service bus implementations such as Apache ServiceMix and Mule. Other projects using ActiveMQ include Apache Camel and Apache CXF in SOA infrastructure projects. (Source: https://en.wikipedia.org/wiki/Apache_ActiveMQ)

In this book, we provide a series of tutorials on Apache ActiveMQ that will help you kick-start your own projects. We cover a wide range of topics, from Apache ActiveMQ Best Practices, to Load Balancing and File Transfer Examples. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at <https://www.javacodegeeks.com/>

Chapter 1

ActiveMQ "Hello World" Example

In this example, we shall show you how to make use of **ActiveMQ** as a message broker for exchanging messages between applications connected via a network. Before starting with our example, it is expected that we have a basic understanding of **JMS concepts** and Java/J2EE. JMS stands for Java Messaging Service, which is a JAVA API that helps communication between two applications connected via a network or two components within an application that communicate by exchanging messages. The messages are exchanged in an asynchronous and reliable manner.

There are several JMS providers like TIBCO EMS, JBOSSMQ etc., which act as Message Oriented Middleware (MOM) and ActiveMQ (Apache) is one of them. The software versions that we will be using to build this example is Java v8 and ActiveMQ v 5.14.5.

1.1 Introduction

ActiveMQ is an open source, message broker (MOM-Message Oriented Middleware) that implements the JMS API. It is lightweight and open source. In this example, we will see how to use **ActiveMQ** as a **JMS provider**.

1.2 How to setup ActiveMQ

- Download ActiveMQ from [Download ActiveMQ](#)
- Extract the ActiveMQ downloaded zip file to any location in your computer
- Open command prompt and go to the bin directory of extracted activemq folder
- Type `activemq.bat start`, to start activemq
- The command prompt will state that activemq has been started by this statement - `INFO | Apache ActiveMQ 5.14.5 (localhost, ID:xxxx) started (please refer screenshot)`. Please refer the activeMQ image below

```

Command Prompt - activemq.bat start
INFO | Connector amqp started
INFO | Listening for connections at: stomp://DESKTOP-FDUV3GG:61613?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector stomp started
INFO | Listening for connections at: mqtt://DESKTOP-FDUV3GG:15837?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector mqtt started
WARN | ServletContext@o.e.j.s.ServletContextHandler@7efaad5a{/,null,STARTING} has uncovered http methods for path: /
INFO | Listening for connections at ws://DESKTOP-FDUV3GG:61614?maximumConnections=1000&wireFormat.maxFrameSize=104857600
INFO | Connector ws started
INFO | Apache ActiveMQ 5.14.5 (localhost, ID:DESKTOP-FDUV3GG-52741-1494136555907-0:1) started
INFO | For help or more information please see: http://activemq.apache.org
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | ActiveMQ WebConsole available at http://0.0.0.0:8161/
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | jolokia-agent: Using policy access restrictor classpath:/jolokia-access.xml
    
```

Figure 1.1: Running ActiveMQ bat

- You can also try to open ActiveMQ console using the link "<https://localhost:8161/admin/>" with *admin/admin* as username and password. It should open as below:



Figure 1.2: ActiveMQ Console

- The ActiveMQ console gives us information about the queue and the topic which we can monitor when we exchange messages

1.3 Using ActiveMQ in a Java project (example using eclipse)

- Let us now create a dynamic web project in eclipse and create our `MessageSender` and `MessageReceiver` classes to see how a message is exchanged using ActiveMQ
- Copy the jar files from extracted ActiveMQ/lib folder in your system to the lib folder of dynamic web project just created in eclipse
- Create two classes one for sending the message (`MessageSender.java`) and another for receiving the message (`MessageReceiver.java`). Please refer code snippets below

This class is used to send a text message to the queue. The JMS class `MessageProducer.java` has a method `send()`, which will be used here to send the message.

MessageSender.java

```
package com.activemq.sender;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

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

public class MessageSender {

    //URL of the JMS server. DEFAULT_BROKER_URL will just mean that JMS server is on localhost
    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;

    // default broker URL is : tcp://localhost:61616"
    private static String subject = "JCG_QUEUE"; // Queue Name.You can create any/many queue names as per your requirement.

    public static void main(String[] args) throws JMSException {
        // Getting JMS connection from the server and starting it
        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
        Connection connection = connectionFactory.createConnection();
        connection.start();

        //Creating a non transactional session to send/receive JMS message.
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);

        //Destination represents here our queue 'JCG_QUEUE' on the JMS server.
        //The queue will be created automatically on the server.
        Destination destination = session.createQueue(subject);

        // MessageProducer is used for sending messages to the queue.
        MessageProducer producer = session.createProducer(destination);

        // We will send a small text message saying 'Hello World!!!'
        TextMessage message = session
            .createTextMessage("Hello !!! Welcome to the world of ActiveMQ.");

        // Here we are sending our message!
        producer.send(message);

        System.out.println("JCG printing@@ '" + message.getText() + "'");
        connection.close();
    }
}
```

This class is used to receive the text message from the queue. The JMS class `MessageConsumer.java` has a method `receive()`, which will be used here to receive the message.

MessageReceiver.java

```
package com.activemq.receiver;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;

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

public class MessageReceiver {

    // URL of the JMS server
    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;
    // default broker URL is : tcp://localhost:61616"

    // Name of the queue we will receive messages from
    private static String subject = "JCG_QUEUE";

    public static void main(String[] args) throws JMSException {
        // Getting JMS connection from the server
        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
        Connection connection = connectionFactory.createConnection();
        connection.start();

        // Creating session for sending messages
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);

        // Getting the queue 'JCG_QUEUE'
        Destination destination = session.createQueue(subject);

        // MessageConsumer is used for receiving (consuming) messages
        MessageConsumer consumer = session.createConsumer(destination);

        // Here we receive the message.
        Message message = consumer.receive();

        // We will be using TextMessage in our example. MessageProducer sent us a ←
        // TextMessage
        // so we must cast to it to get access to its .getText() method.
        if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            System.out.println("Received message '" + textMessage.getText() + " ←
                '");
        }
        connection.close();
    }
}
```

So now we have the following things:

- A JMS provider is running (Running ActiveMQ as shown in image above)
- We have our MessageSender and MessageReceiver classes
- Next we will see how a message is exchanged using Active MQ but before that here is a brief about the classes that we have used in our code

Connection → It represents the connection with the JMS provider - ActiveMQ. It is not a database connection.

Destination → It represents a destination on our message provider ActiveMQ i.e. it represents a queue where we will be sending our messages. In our case we are sending it to queue "JCG_QUEUE" which will be automatically created once the MessageSender class is executed.

MessageProducer → It represents a producer to send a message to the queue.

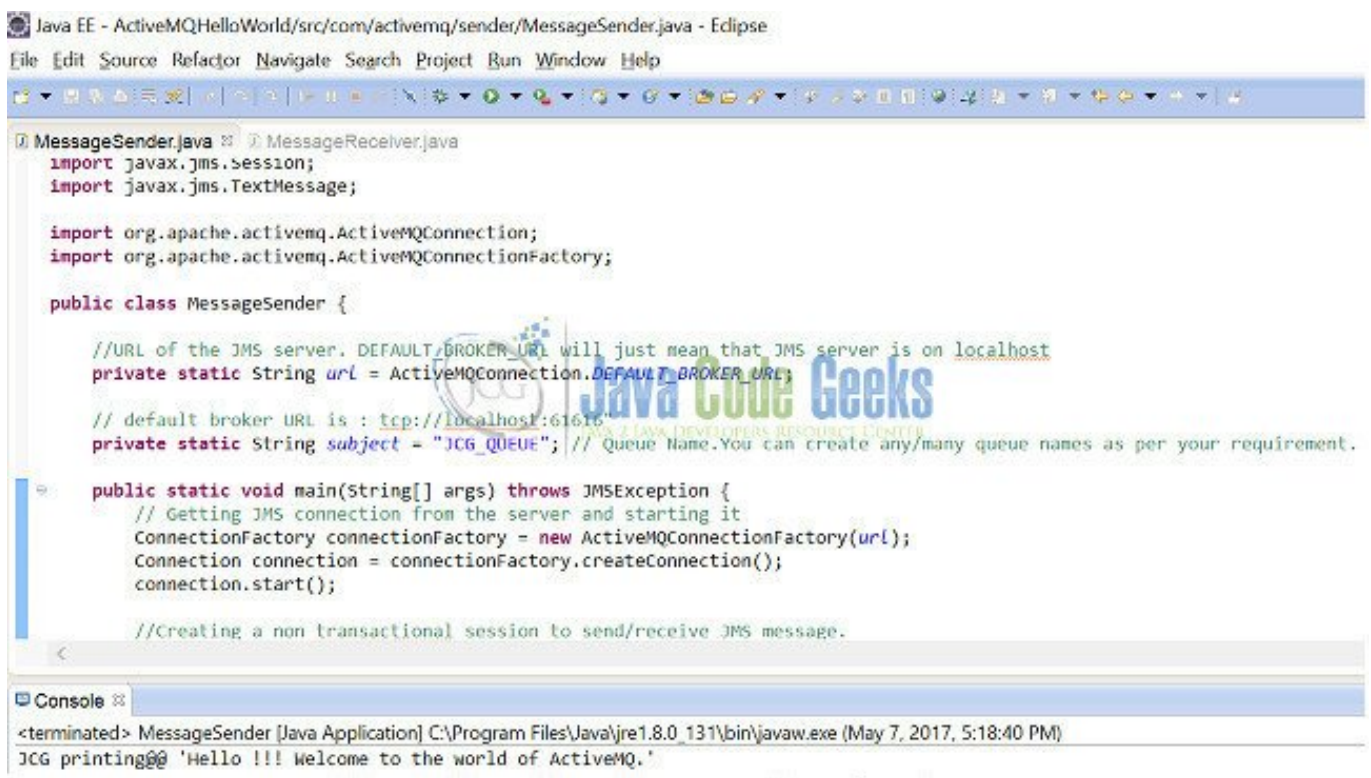
MessageConsumer → It represents a receiver to receive a message from the queue.

1.4 Output

We will be running our java classes written above to check how the message is exchanged.

Please follow the steps below:

- In the eclipse Right Click on MessageSender.java → Run As→Java Application to see if our message is sent to the queue. The Hello message after being successfully sent to the queue gets printed in eclipse output console



The screenshot shows the Eclipse IDE interface. The top part displays the source code for MessageSender.java. The code includes imports for javax.jms.Session, javax.jms.TextMessage, org.apache.activemq.ActiveMQConnection, and org.apache.activemq.ActiveMQConnectionFactory. The main method creates an ActiveMQConnectionFactory with the default URL, creates a connection, and starts it. A comment indicates that a non-transactional session will be used for sending/receiving JMS messages. The console output at the bottom shows the execution of the Java application, with the message "JCG printing@ 'Hello !!! welcome to the world of ActiveMQ.'" printed to the console.

```
Java EE - ActiveMQHelloWorld/src/com/activemq/sender/MessageSender.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

MessageSender.java MessageReceiver.java
import javax.jms.Session;
import javax.jms.TextMessage;

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

public class MessageSender {

    //URL of the JMS server. DEFAULT_BROKER_URL will just mean that JMS server is on localhost
    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;

    // default broker URL is : tcp://localhost:61616"
    private static String subject = "JCG_QUEUE"; // Queue Name.You can create any/many queue names as per your requirement.

    public static void main(String[] args) throws JMSException {
        // Getting JMS connection from the server and starting it
        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
        Connection connection = connectionFactory.createConnection();
        connection.start();

        //Creating a non transactional session to send/receive JMS message.
    }
}

Console
<terminated> MessageSender [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (May 7, 2017, 5:18:40 PM)
JCG printing@ 'Hello !!! welcome to the world of ActiveMQ.'
```

Figure 1.3: MessageSender.java

- We can also check our ActiveMQ console→Queues tab, to see the number of pending messages in our queue after running the MessageSender.java



Figure 1.4: ActiveMQ console with pending message

The producer task is done. It will just create a queue and send a message to it. Now who is going to read that message, it will not be mentioned in the producer.

Next comes our `MessageReceiver` class which has a connection and a destination same as explained earlier. Here we also mention the same queue name that we have used in the `MessageSender` class to know that we are going to pick the message from the same queue. The `consumer.receive()` is used to receive the message. Once the class is run, the message is received from the queue and gets printed in eclipse output console when we will execute the class.

- In the eclipse Right Click on `MessageReceiver.java` → `Run As` → `Java Application` to see if our message is received from the queue. The hello message after being successfully received gets printed in eclipse output console.

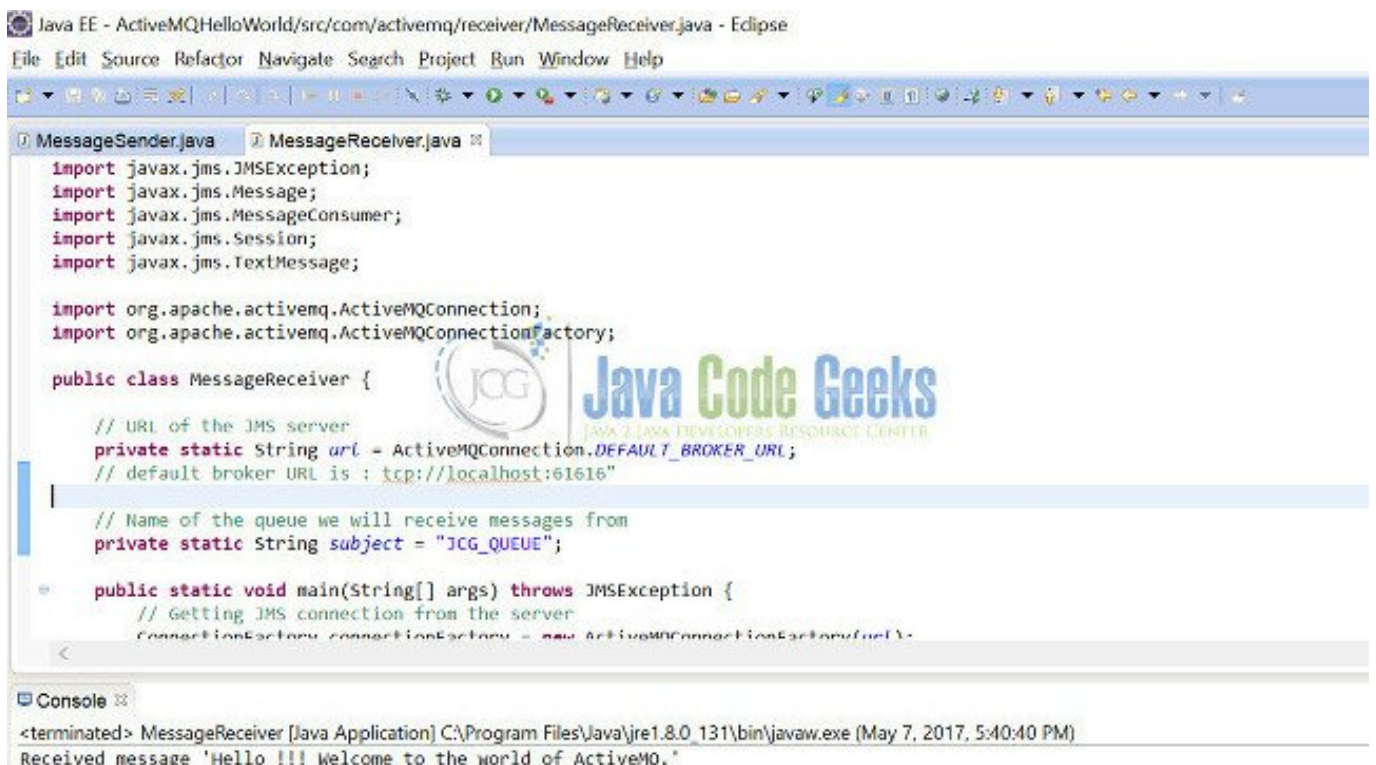


Figure 1.5: MessageReceiver.java

- We can also check our ActiveMQ console→Queues tab, to see the number of dequeued messages from our queue after running the MessageReceiver.java



Figure 1.6: ActiveMQ console with received message

1.5 Conclusion

Through this example, we have learned how to configure ActiveMQ and use it to exchange any text message in an asynchronous manner. Both the sender and receiver classes are independent and we just have to configure the queue name that we want to send message to and receive message from.

Chapter 2

ActiveMQ Monitoring Tutorial

Apache ActiveMQ (AMQ) is an open source messaging server written in Java which implements JMS 1.1 specifications. In this example, I will explain how to monitor an AMQ server.

2.1 Introduction

Apache ActiveMQ (AMQ) is an open source messaging server written in Java which implements JMS 1.1 specifications. Most of the enterprise applications use AMQ as a building block for their MOM infrastructure, so it's critical that the AMQ server functions as expected. In this example, I will explain how to monitor an AMQ server from these four aspects:

- Java Virtual Machine (JVM)
- AMQ Web Console
- File System
- Message Broker

2.2 Monitor ActiveMQ JVM

AMQ is a JVM. [Oracle provides free monitoring tools](#) to monitor the JVM's CPU, Memory and threads.

- [Java Mission Control](#)
 - [jcmd Utility](#)
 - [Java visualVM](#)
 - [JConsole Utility](#)
 - [jmap Utility](#)
 - [jps Utility](#)
 - [jstack Utility](#)
 - [jstat Utility](#)
 - [jstatd Daemon](#)
 - [visualgc Utility](#)
-

- [Native Tools](#)

Open Source JVM monitor tools:

- [Stagemonitor](#)
- [Pinpoint](#)
- [MoSKito](#)
- [Glowroot](#)
- [Kamon](#)
- [ServOne](#)

2.3 Monitor ActiveMQ Web Console

AMQ provides a [web console](#) (Ex: <https://localhost:8161/admin/index.jsp>) to enable users to administrate and monitor the AMQ. The administrator can check any queue which has no consumer or has large amounts of pending messages. AMQ also provides the XML feeds to retrieve the information from the queues. (Ex: <https://localhost:8161/admin/xml/queues.jsp>). Users can use a web application monitoring tool to monitor the web console index web page and the XML feed page.

Web application monitoring tools:

- [updown](#)
 - [Pingometer](#)
 - [Uptime Robot](#)
 - [Pingdom](#)
 - [StatusCake](#)
 - [New Relic](#)
 - [Monitis](#)
 - [StatusOK](#)
 - [Monitority](#)
 - [Montastic](#)
 - [AppBeat](#)
 - [Uptrends](#)
 - [HostTracker](#)
 - [Site24x7](#)
 - [SiteScope](#)
-

2.4 Monitor ActiveMQ File System

AMQ uses `log4j` to generate three log files under the `data` directory with the default settings.

- `activemq.log`
- `audit.log`
- `wrapper.log`

The log file's information and location can be changed by following these [instructions](#). These log files grow as time goes by. Users can use a file system monitoring tool to monitor the disk space.

File system monitoring tools:

- [Watch 4 Folder](#)
- [Directory Monitor](#)
- [TheFolderSpy](#)
- [Track Folder Changes](#)
- [FolderChangesView](#)
- [Windows Explorer Tracker](#)
- [Spy-The-Spy](#)
- [SpyMe Tools](#)
- [Disk Pulse](#)
- [File Alert Monitor](#)

2.5 Monitor ActiveMQ Application

AMQ is a [message broker](#) which transfers the message from the sender to the receiver. Users monitor the AMQ broker to ensure that the messaging broker is in a healthy state.

AMQ broker monitoring tools:

- [AMQ JMX MBeans](#)
 - [AMQ Advisory Message](#)
 - [Visualisation](#)
 - [Statistics](#)
 - [jmxtrans](#)
 - [ActiveMQ Monitor \(AMon\)](#)
 - [Apache ActiveMQBrowser](#)
 - [HermesJMS](#)
 - [HermesJMS/soapUI](#)
 - [Hyperic HQ and Hyperic HQ Enterprise](#)
 - [FuseHQ](#) (based on Hyperic HQ Enterprise)
-

- [iTKO LISA Test](#)
- [Geronimo Administration Console](#) (JMS Resources)
- [Media Driver Integrated Console](#)
- [Hawt](#)
- [Jolokia](#)

2.6 Java Example

In this example, I will demonstrate how to build a simple Java scheduler application which monitors AMQ queues and then sends notifications when it detects a queue with no consumers or a large amount of pending messages.

2.6.1 Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Quartz 2.2.1 (2.x will do fine)
- Eclipse Neon (Any Java IDE would work)

2.6.2 Data Model

AMQ provides a XML feeds to retrieve the information from the queues.

<https://localhost:8161/admin/xml/queues.jsp>

```
<queues>
  <queue name="test.queue">
    <stats size="0" consumerCount="0" enqueueCount="0" dequeueCount="0" />
    <feed>
      <atom>queueBrowse/test.queue?view=rss&feedType=atom_1.0</atom>
      <rss>queueBrowse/test.queue?view=rss&feedType=rss_2.0</rss>
    </feed>
  </queue>
</queues>
```

I create a `ListOfActiveMqQueue` to map to the root of the XML element `queues` .

`ListOfActiveMqQueue.java`

```
package jcg.demo.model;

import java.util.List;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "queues")
public class ListOfActiveMqQueue {
    private List queue;

    public List getQueue() {
        return queue;
    }
}
```

```
    }

    public void setQueue(List queue) {
        this.queue = queue;
    }
}
```

I create a `ActiveMqQueue` to map to the child element `queue` .

`ActiveMqQueue.java`

```
package jcg.demo.model;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="queue")
@XmlAccessorType(XmlAccessType.FIELD)
public class ActiveMqQueue {

    @XmlAttribute
    private String name;
    private Stats stats;

    public String getName() {
        return name;
    }

    public Stats getStats() {
        return stats;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setStats(Stats stats) {
        this.stats = stats;
    }
}
```

I create a `Stats` to map to the child element `stats` .

`Stats.java`

```
package jcg.demo.model;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Stats {

    @XmlAttribute
    private Integer size;
    @XmlAttribute
    private int consumerCount;
    @XmlAttribute
```

```
private int enqueueCount;
@XmlAttribute
private int dequeueCount;

public Integer getSize() {
    return size;
}

public int getConsumerCount() {
    return consumerCount;
}

public int getEnqueueCount() {
    return enqueueCount;
}

public int getDequeueCount() {
    return dequeueCount;
}

public void setSize(Integer size) {
    this.size = size;
}

public void setConsumerCount(int consumerCount) {
    this.consumerCount = consumerCount;
}

public void setEnqueueCount(int enqueueCount) {
    this.enqueueCount = enqueueCount;
}

public void setDequeueCount(int dequeueCount) {
    this.dequeueCount = dequeueCount;
}
}
```

2.6.3 Monitor Service

Create a monitor service which monitors the given AMQ server's queues and invokes the `NotificationService` to send an email when it detects that the queue has no consumers or there is a queue with too many pending messages.

`ActiveMQMonitorService.java`

```
package jcg.demo.service;

import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

import jcg.demo.model.ActiveMqQueue;
import jcg.demo.model.NotificationEmail;
import jcg.demo.model.ListOfActiveMqQueue;

public class ActiveMQMonitorService {

    private ActiveMqQueueTransformer transformer = new ActiveMqQueueTransformer();
    private NotificationService notService = new NotificationService();
}
```

```
@SuppressWarnings("resource")
public void monitorAndNotify(String brokerUrl, String username, String password, ←
    int maxPendingMessageLimit) {
    System.out.println("monitorAndNotify starts for " + brokerUrl);
    NotificationEmail email = dummyEmail();
    InputStream xmlFeedData = readXmlFeeds(brokerUrl, username, password);

    ListOfActiveMqQueue activeMqXmlData = transformer.convertFromInputStream( ←
        xmlFeedData);
    if (activeMqXmlData != null) {
        for (ActiveMqQueue queue : activeMqXmlData.getQueue()) {
            if (queue.getStats().getConsumerCount() == 0) {
                email.setSubject("Must check activeMQ, queue: " + ←
                    queue.getName() + " has no consumer.");
                notService.sendEmail(email);
            }
            else{
                int pendingMessageCounts = queue.getStats().getSize ←
                    () - queue.getStats().getEnqueueCount();
                if (pendingMessageCounts > maxPendingMessageLimit){
                    email.setSubject("Must check activeMQ, ←
                        queue: " + queue.getName() + " has large ←
                        pending message. ");
                    notService.sendEmail(email);
                }
            }
        }
    }
    System.out.println("monitorAndNotify completes for " + brokerUrl);
}

private InputStream readXmlFeeds(String brokerUrl, String username, String password ←
    ) {
    try {
        URL url = new URL(brokerUrl);
        URLConnection uc = url.openConnection();

        String userpass = username + ":" + password;
        String basicAuth = "Basic " + javax.xml.bind.DatatypeConverter. ←
            printBase64Binary(userpass.getBytes());

        uc.setRequestProperty("Authorization", basicAuth);

        return uc.getInputStream();

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }

    return null;
}

private NotificationEmail dummyEmail() {
    NotificationEmail noConsumerEmail = new NotificationEmail();
    noConsumerEmail.setFromAddress("monitorApp@noreply.com");
    noConsumerEmail.setToAddress("activeMQServerMonitorTeam@test.com");
    noConsumerEmail.setBody("test email body message");
    return noConsumerEmail;
}
```



```
}

```

- Line 22: Read the AMQ queue data
- Line 24: Transform the AMQ queue data from `InputStream` to `ListOfActiveMqQueue`
- Line 27: Send notification when detects the queue has no consumer
- Line 33: Send notification when detects the queue has large pending messages

2.6.4 Quartz Job

Create a Quartz job to monitor the AMQ server.

QuartzJob.java

```
package jcg.demo.scheduler.quartz2;

import java.time.LocalDateTime;
import java.util.List;

import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

import jcg.demo.model.NotificationEmail;
import jcg.demo.service.ActiveMQMonitorService;
import jcg.demo.service.NotificationService;

/**
 * This class implements Quartz Job interface, anything you wish to be executed
 * by the Quartz Scheduler should be here it should invokes business class to
 * perform task.
 *
 * @author Mary.Zheng
 *
 */
public class QuartzJob implements Job {

    private static final String LOGIN = "admin";
    private static final int MAX_PENDING_MESSAGE_SIZE_LIMIT = 10;
    private String brokerXmlUrl = "https://localhost:8161/admin/xml/queues.jsp";

    private ActiveMQMonitorService activeMqMonitorService = new ActiveMQMonitorService ←
        ();

    @Override
    public void execute(JobExecutionContext context) throws JobExecutionException {
        LocalDateTime localTime = LocalDateTime.now();
        System.out.println(Thread.currentThread().getName() + ": Run QuartzJob at " ←
            + localTime.toString());

        activeMqMonitorService.monitorAndNotify(brokerXmlUrl, LOGIN, LOGIN, ←
            MAX_PENDING_MESSAGE_SIZE_LIMIT);
    }
}
```

2.6.5 Monitor Application

Create a monitor application which runs every minutes to monitor the given AMQ server's queues and sends notifications when it detects that the queue has no consumers or there is a queue with too many pending messages.

QuartzSchedulerApp.java

```
package jcg.demo.scheduler.quartz2;

import java.util.List;

import org.quartz.CronScheduleBuilder;
import org.quartz.JobBuilder;
import org.quartz.JobDetail;
import org.quartz.JobExecutionContext;
import org.quartz.Scheduler;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.impl.StdSchedulerFactory;

/**
 * This application schedule a job to run every minute
 *
 * @author Mary.Zheng
 *
 */
public class QuartzSchedulerApp {

    private static final String TRIGGER_NAME = "MyTriggerName";
    private static final String GROUP = "simple_Group";
    private static final String JOB_NAME = "someJob";
    private static Scheduler scheduler;

    public static void main(String[] args) throws Exception {
        System.out.println("QuartzSchedulerApp main thread: " + Thread. ←
            currentThread().getName());

        scheduler = new StdSchedulerFactory().getScheduler();
        scheduler.start();

        List currentJobs = scheduler.getCurrentlyExecutingJobs();
        for (JobExecutionContext currJob : currentJobs) {
            System.out.println("running job" + currJob.toString() + currJob. ←
                getJobDetail());
        }

        Trigger trigger = buildCronSchedulerTrigger();
        scheduleJob(trigger);
    }

    private static void scheduleJob(Trigger trigger) throws Exception {

        JobDetail someJobDetail = JobBuilder.newJob(QuartzJob.class).withIdentity( ←
            JOB_NAME, GROUP).build();

        scheduler.scheduleJob(someJobDetail, trigger);
    }

    private static Trigger buildCronSchedulerTrigger() {
        String CRON_EXPRESSION = "0 * * * * ?";
    }
}
```

```
        Trigger trigger = TriggerBuilder.newTrigger().withIdentity(TRIGGER_NAME, ←
            GROUP)
            .withSchedule(CronScheduleBuilder.cronSchedule( ←
                CRON_EXPRESSION)).build();

        return trigger;
    }
}
```

2.6.6 Monitor Application execution

Start the AMQ server locally and then execute the monitor application.

Output

```
QuartzSchedulerApp main thread: main
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
DefaultQuartzScheduler_Worker-1: Run QuartzJob at 2017-12-28T14:15:00.058
monitorAndNotify starts for https://localhost:8161/admin/xml/queues.jsp
NotificationService send email EmailNotification [ toAddress=activeMQServerMonitorTeam@test ←
    .com, subject=Must check activeMQ, queue: test.queue has no consumer., body=test email ←
    body message]
monitorAndNotify completes for https://localhost:8161/admin/xml/queues.jsp
DefaultQuartzScheduler_Worker-2: Run QuartzJob at 2017-12-28T14:16:00.002
monitorAndNotify starts for https://localhost:8161/admin/xml/queues.jsp
NotificationService send email EmailNotification [ toAddress=activeMQServerMonitorTeam@test ←
    .com, subject=Must check activeMQ, queue: test.queue has no consumer., body=test email ←
    body message]
monitorAndNotify completes for https://localhost:8161/admin/xml/queues.jsp
```

2.7 Summary

In this example, I outlined how to monitor an AMQ server from four aspects: JVM, AMQ web console, the data file system, and the message broker. I also built a simple Java monitor application to monitor the AMQ based on the web console. There are lots of monitoring tools in the market. Use your best judgement to pick the best monitoring tool for your business. Here are few items to consider when choosing a monitoring tool:

- any security concern?
- any impact to the AMQ server?
- any interface to your notification tool?

2.8 Download the Source Code

This example consists of a Quartz scheduler to monitor an AMQ server based on the web console queue data. You can download the full source code of this example here: [Apache ActiveMQ Monitoring Tutorial](#)

Chapter 3

ActiveMQ Best Practices Tutorial

Apache ActiveMQ is an open source messaging server written in Java which implements JMS 1.1 specifications. In this chapter, you will learn how to develop a few Java applications which integrate ActiveMQ to send and receive messages to and from destinations. If you already know how to install and configure ActiveMQ, you can skip the first four chapters.

3.1 Introduction

Apache ActiveMQ (AMQ) is [JMS 1.1](#) implementation from [the Apache Software Foundation](#).

AMQ is a [message broker](#) which translates the messages from the sender to the receiver. Message brokers are the building blocks of [message-oriented middleware](#) (MOM) architecture.

AMQ is one of the best open source messaging and [Integration Patterns](#) server. It provides a communication between applications, as well as fulfills both notification and inter-operation needs among the applications.

3.2 Install an ActiveMQ Server

Most of business applications treat the AMQ as an infrastructure resource. We will install an AMQ server as a standalone server in this chapter. Follow these [instructions](#), we installed the AMQ 5.15.0.

3.3 Start the ActiveMQ Server

Navigate to `\apache-activemq-5.15.0\bin\win64` directory and click on the `activemq.bat` to start the server.

The output below demonstrates that the server started successfully.

server.log

```
jvm 1 | INFO | Apache ActiveMQ 5.15.0 (localhost, ID:SL2LS431841 ↵  
-57319-1512184574307-0:1) started  
jvm 1 | INFO | For help or more information please see: https://activemq.apache.org
```

3.4 Monitor the ActiveMQ Server

AMQ provides a web console application to monitor and administrate. After the AMQ server starts, follow the steps below to launch the web console.

- Open a Browser: Chrome, IE, Firefox, etc
- Enter the URL: `localhost:8161/admin/index.php`
- Enter `admin/admin` as username/password

Here you should see the "Welcome" page. Users can send, read, and delete messages via the web console.

3.5 Business Use Cases

Company X provides services to customers. Each new customer will be set up at billing and support systems.

In this chapter, we will demonstrate how to build customer on-boarding process, billing system, support application, and integrate them via AMQ:

- `OnBoardNewCustomerApp` which sets up new customers and sends the new customer events to ActiveMQ customer topic
- `ConfigBillingForNewCustomerApp` which listens to the new customer events from the virtual topic and configures it into the billing application
- `ConfigSupportForNewCustomerApp` which listens to the new customer events from the virtual topic and configures it into the support application

3.6 Define JMS Message

3.6.1 Message Destination

For this business use case, both billing and support systems get notified when new customer joins. We choose the publish/subscribe message pattern to build the `OnBoardNewCustomerApp` which publishes the customer event to AMQ broker topic: `VirtualTopic.Customer.Topic`. There are three special characters reserved by AMQ when naming the destination:

- `.` is used to separate names in a path
- `*` is used to match any name in a path
- `>` is used to recursively match any destination starting from this name

3.6.2 Message Header

The message header provides meta data about the message used by both clients and the AMQ brokers. There are sets of pre-defined JMS message header. Giving two examples below:

- `JMSXGroupID`: utilize this if you want some group of message to always go to same consumer
- `JMXCorrelationId`: use this to link the message together

3.6.3 Message Body

The message body is the actual message that integrates the applications together. For this example, the message is Json format of the `CustomerEvent`.

`CustomerEvent`

```
package jcg.demo.model;

public class CustomerEvent {
    private String type;
    private Integer customerId;

    public CustomerEvent(String type, Integer customerId) {
        this.type = type;
        this.customerId = customerId;
    }

    public String getType() {
        return type;
    }

    public Integer getCustomerId() {
        return customerId;
    }

    public String toString() {
        return "CustomerEvent: type(" + type + "), customerId(" + customerId + ")";
    }

    public String getCustomerDetailUri() {
        return "https://localhost:8080/support/customer/" + customerId;
    }
}
```

3.6.4 Configure Virtual Topic

AMQ server installation comes with a ready to use configuration file. Modify the `activemq.xml` to add below to allow AMQ Broker forwards the messages from any topic named as `VirtualTopic.*.Topic` to any virtual topic destination with name starts as `Consumer.*`.

`activemq.xml`

```
<destinationInterceptors>
  <virtualDestinationInterceptor>
    <virtualDestinations>
      <virtualTopic name="VirtualTopic.>" prefix="Consumer.*" selectorAware="↔
        false"/>
      <virtualTopic name="JCG.>" prefix="VTC.*" selectorAware="true"/>
    </virtualDestinations>
  </virtualDestinationInterceptor>
</destinationInterceptors>
```

- line 4: Configure Virtual Topic to disable `selectorAware`
- line 4: Configure Virtual Topic to enable `selectorAware`

Restart the AMQ server after the configuration file updates.

3.7 ActiveMQ Java Client Library

Add ActiveMQ Java library to the project `pom.xml`.

`pom.xml`

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-client</artifactId>
  <version>5.15.0</version>
</dependency>
```

3.8 Publish Message Application

In this example, you will see how to create `ActiveMQMessageProducer` to send the messages.

3.8.1 ActiveMQMessageProducer

A Java class wraps the ActiveMQ Java API to send the messages.

`ActiveMQMessageProducer`

```
package jcg.demo.activemq;

import java.util.Random;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.RedeliveryPolicy;

import com.google.gson.Gson;

import jcg.demo.jms.util.DataUtil;

/**
 * A simple message producer which sends the message to the ActiveMQ Broker.
 *
 * @author Mary.Zheng
 *
 */
public class ActiveMQMessageProducer {

    private static final String ACTION_ID_HEADER = "actionId";
    private static final String ACTION_HEADER = "action";

    private ConnectionFactory connFactory;
    private Connection connection;
    private Session session;
    private Destination destination;
    // https://docs.oracle.com/javaee/7/api/javax/jms/MessageProducer.html
    private MessageProducer msgProducer;

    private String activeMqBrokerUri;
    private String username;
    private String password;
```

```
public ActiveMQMessageProducer(final String activeMqBrokerUri, final String ↵
    username, final String password) {
    super();
    this.activeMqBrokerUri = activeMqBrokerUri;
    this.username = username;
    this.password = password;
}

public void setup(final boolean transacted, final boolean isDestinationTopic, final ↵
    String destinationName)
    throws JMSEException {
    setConnectionFactory(activeMqBrokerUri, username, password);
    setConnection();
    setSession(transacted);
    setDestination(isDestinationTopic, destinationName);
    setMsgProducer();
}

public void close() throws JMSEException {
    if (msgProducer != null) {
        msgProducer.close();
        msgProducer = null;
    }

    if (session != null) {
        session.close();
        session = null;
    }

    if (connection != null) {
        connection.close();
        connection = null;
    }
}

public void commit(final boolean transacted) throws JMSEException {
    if (transacted) {
        session.commit();
    }
}

public void sendMessage(final String actionVal) throws JMSEException {
    TextMessage textMessage = buildTextMessageWithProperty(actionVal);
    msgProducer.send(destination, textMessage);
    // msgProducer.send(textMessage, DeliveryMode.NON_PERSISTENT, 0, 0);
}

private TextMessage buildTextMessageWithProperty(final String action) throws ↵
    JMSEException {
    Gson gson = new Gson();
    String eventMsg = gson.toJson(DataUtil.buildDummyCustomerEvent());
    TextMessage textMessage = session.createTextMessage(eventMsg);

    Random rand = new Random();
    int value = rand.nextInt(100);
    textMessage.setStringProperty(ACTION_HEADER, action);
    textMessage.setStringProperty(ACTION_ID_HEADER, String.valueOf(value));

    return textMessage;
}
```



```

private void setDdestination(final boolean isDestinationTopic, final String ↵
destinationName) throws JMSEException {
    if (isDestinationTopic) {
        destination = session.createTopic(destinationName);
    } else {
        destination = session.createQueue(destinationName);
    }
}

private void setMsgProducer() throws JMSEException {
    msgProducer = session.createProducer(destination);
}

private void setSession(final boolean transacted) throws JMSEException {
    // transacted=true for better performance to push message in batch mode
    session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);
}

private void setConnection() throws JMSEException {
    connection = connFactory.createConnection();
    connection.start();
}

private void setConnectionFactory(final String activeMqBrokerUri, final String ↵
username, final String password) {
    connFactory = new ActiveMQConnectionFactory(username, password, ↵
        activeMqBrokerUri);

    ((ActiveMQConnectionFactory) connFactory).setUseAsyncSend(true);

    RedeliveryPolicy policy = ((ActiveMQConnectionFactory) connFactory). ↵
        getRedeliveryPolicy();
    policy.setInitialRedeliveryDelay(500);
    policy.setBackOffMultiplier(2);
    policy.setUseExponentialBackOff(true);
    policy.setMaximumRedeliveries(2);
}
}

```

- line 51-55: Wire connection, session with correct order. Spring JMS Dependency Injection takes care of it for you
- line 58-73: Close connection. Spring JMS takes care of it for you
- line 84: Define the durability of message. All message are durable by default. We can turn off to get better performance

3.8.2 ActiveMQMessgeProducerTest

This Junit test sends the messages to various destinations. This is my convenient way to send the message to the destination.

ActiveMQMessgeProducerTest

```

package jcg.demo.activemq;

import javax.jms.JMSEException;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

```

```
import jcg.demo.jms.util.DataUtil;

public class ActiveMQMessageProducerTest {

    private ActiveMQMessageProducer msgQueueSender;

    @Before
    public void setup() {
        msgQueueSender = new ActiveMQMessageProducer("tcp://localhost:61616", " ←
            admin", "admin");
    }

    @After
    public void cleanup() throws JMSEException {
        msgQueueSender.close();
    }

    @Test
    public void send_msg_to_no_transaction_Queue() throws JMSEException {
        msgQueueSender.setup(false, false, DataUtil.TEST_GROUP1_QUEUE_1);
        msgQueueSender.sendMessage("JCG");
    }

    @Test
    public void send_msg_to_Group2_Queue1() throws JMSEException {
        msgQueueSender.setup(false, false, DataUtil.TEST_GROUP2_QUEUE_1);
        msgQueueSender.sendMessage("JCG");
    }

    @Test
    public void send_msg_to_transaction_Group1_Queue2() throws JMSEException {
        msgQueueSender.setup(true, false, DataUtil.TEST_GROUP1_QUEUE_2);
        msgQueueSender.sendMessage("DEMO");
        msgQueueSender.commit(true);
    }

    @Test
    public void send_msg_to_no_transaction_Group1_Topic() throws JMSEException {
        msgQueueSender.setup(false, true, DataUtil.TEST_GROUP1_TOPIC);
        msgQueueSender.sendMessage("MZHENG");
    }

    @Test
    public void send_msg_to_Virtual_Topic() throws JMSEException {
        msgQueueSender.setup(false, true, DataUtil.CUSTOMER_VTC_TOPIC);
        msgQueueSender.sendMessage("MZHENG");
    }

    @Test
    public void send_msg_to_Virtual_Topic_WithSelector() throws JMSEException {
        msgQueueSender.setup(false, true, DataUtil.TEST_VTC_TOPIC_SELECTOR);
        msgQueueSender.sendMessage("DZONE");
    }
}
```

- line 27-28: Send to queue test.group1.queue1
- line 33-34: Send to queue test.group2.queue1
- line 39-41: Send to queue test.group1.queue2

- line 46-47: Send to normal topic `test.group1.topic`
- line 52-53: Send to selector unaware topic `VirtualTopic.Customer.Topic`
- line 58-59: Send to selector aware topic `JCG.Mary.Topic`

3.8.3 Execution Output

We ran the `ActiveMQMessageProducerTest` to send message to three queues and three topics. You can verify by viewing the AMQ web console. There are one pending messages in each of three queues: `test.group1.queue1`, `test.group1.queue2`, and `test.group2.queue1`.

There is one messages in each of three topics: `JCG.Mary.Topic`, `test.group1.topic` and `VirtualTopic.Customer.Topic`.

3.8.4 OnBoardNewCustomerApp

`OnBoardNewCustomerApp` sends the new customer message to the `VirtualTopic.Customer.Topic`.

`OnBoardNewCustomerApp`

```
package jcg.demo.activemq.app;

import jcg.demo.activemq.ActiveMQMessageProducer;
import jcg.demo.jms.util.DataUtil;

public class OnBoardNewCustomerApp {
    public static void main(String[] args) {
        ActiveMQMessageProducer msgQueueSender = new ActiveMQMessageProducer("tcp ←
        ://localhost:61616", "admin", "admin");
        try {
            msgQueueSender.setup(false, true, DataUtil.CUSTOMER_VTC_TOPIC);
            msgQueueSender.sendMessage("CUSTOMER");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Execute `OnBoardNewCustomerApp` sends a customer message to the `VirtualTopic.Customer.Topic`. However, since there is no consumer yet, so AMQ Broker will not send any message to the virtual topic queue yet.

3.9 Consume Message Application

3.9.1 ActiveMQMessageConsumer

A message consumer utilizes AMQ java API.

`ActiveMQMessageConsumer`

```
package jcg.demo.activemq;

import java.util.Enumeration;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
```

```
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;

import org.apache.activemq.ActiveMQConnectionFactory;

/**
 * A simple message consumer which consumes the message from ActiveMQ Broker.
 *
 * @author Mary.Zheng
 *
 */
public class ActiveMQMessageConsumer implements MessageListener {

    private String activeMqBrokerUri;
    private String username;
    private String password;

    private boolean isDestinationTopic;
    private String destinationName;
    private String selector;
    private String clientId;

    public ActiveMQMessageConsumer(String activeMqBrokerUri, String username, String password) {
        super();
        this.activeMqBrokerUri = activeMqBrokerUri;
        this.username = username;
        this.password = password;
    }

    public void run() throws JMSEException {
        ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(username, password, activeMqBrokerUri);
        if (clientId != null) {
            factory.setClientID(clientId);
        }
        Connection connection = factory.createConnection();
        if (clientId != null) {
            connection.setClientID(clientId);
        }
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        ;

        setComsumer(session);

        connection.start();
        System.out.println(Thread.currentThread().getName() + ": ActiveMQMessageConsumer Waiting for messages at "
            + destinationName);
    }

    private void setComsumer(Session session) throws JMSEException {
        MessageConsumer consumer = null;
        if (isDestinationTopic) {
            Topic topic = session.createTopic(destinationName);

            if (selector == null) {
                consumer = session.createConsumer(topic);
            }
        }
    }
}
```

```
        } else {
            consumer = session.createConsumer(topic, selector);
        }
    } else {
        Destination destination = session.createQueue(destinationName);

        if (selector == null) {
            consumer = session.createConsumer(destination);
        } else {
            consumer = session.createConsumer(destination, selector);
        }
    }

    consumer.setMessageListener(this);
}

@Override
public void onMessage(Message message) {

    String msg;
    try {
        msg = String.format(
            "[%s]: ActiveMQMessageConsumer Received message ←
            from [ %s] - Headers: [ %s] Message: [ %s ]",
            Thread.currentThread().getName(), destinationName, ←
            getPropertyNames(message),
            ((TextMessage) message).getText());
        System.out.println(msg);
    } catch (JMSEException e) {
        e.printStackTrace();
    }
}

private String getPropertyNames(Message message) throws JMSEException {
    String props = "";
    @SuppressWarnings("unchecked")
    Enumeration properties = message.getPropertyNames();
    while (properties.hasMoreElements()) {
        String propKey = properties.nextElement();
        props += propKey + "=" + message.getStringProperty(propKey) + " ";
    }
    return props;
}

public void setSelector(String selector) {
    this.selector = selector;
}

public boolean isDestinationTopic() {
    return isDestinationTopic;
}

public String getDestinationName() {
    return destinationName;
}

public String getSelector() {
    return selector;
}

public String getClientId() {
```

```

        return clientId;
    }

    public void setDestinationTopic(boolean isDestinationTopic) {
        this.isDestinationTopic = isDestinationTopic;
    }

    public void setDestinationName(String destinationName) {
        this.destinationName = destinationName;
    }

    public void setClientId(String clientId) {
        this.clientId = clientId;
    }
}

```

- line 23: Create ActiveMQMessageConsumer by implementing `javax.jms.MessageListener`
- line 44: Set connection `clientId`
- line 62: Create a topic
- line 65: Create message consumer from a topic without selector
- line 67: Create message consumer from a topic with selector
- line 70: Create a queue
- line 73: Create message consumer from a queue without selector
- line 75: Create message consumer from a queue with selector
- line 79: Register message listener
- line 83: Override the `onMessage`

3.9.2 ActiveMQMessageConsumerMainApp

Create `ActiveMQMessageConsumerMainApp` to consume from various destinations.

`ActiveMQMessageConsumerMainApp`

```

package jcg.demo.activemq.app;

import javax.jms.JMSEException;

import jcg.demo.activemq.ActiveMQMessageConsumer;
import jcg.demo.jms.util.DataUtil;

public class ActiveMQMessageConsumerMainApp {

    public static void main(String[] args) {

        consumeCustomerVTCQueue();
        consumerVTCQueueWithSelector();
        consumeGroup1Topic();
        consumeAllGroup2();
        consume_queue_with_prefetchsize();

    }

    private static void consumeCustomerVTCQueue() {

```

```
// the message in the topic before this subscriber starts will not be
// picked up.
ActiveMQMessageConsumer queueMsgListener = new ActiveMQMessageConsumer("tcp ←
    ://localhost:61616", "admin",
        "admin");
queueMsgListener.setDestinationName("Consumer.zheng." + DataUtil. ←
    CUSTOMER_VTC_TOPIC);

try {
    queueMsgListener.run();
} catch (JMSEException e) {

    e.printStackTrace();
}

}

private static void consumerVTCQueueWithSelector() {
ActiveMQMessageConsumer queueMsgListener = new ActiveMQMessageConsumer("tcp ←
    ://localhost:61616", "admin",
        "admin");
queueMsgListener.setDestinationName("VTC.DZONE." + DataUtil. ←
    TEST_VTC_TOPIC_SELECTOR);
queueMsgListener.setSelector("action='DZONE'");
try {
    queueMsgListener.run();
} catch (JMSEException e) {

    e.printStackTrace();
}
}

private static void consumeGroup1Topic() {
ActiveMQMessageConsumer queueMsgListener = new ActiveMQMessageConsumer("tcp ←
    ://localhost:61616", "admin",
        "admin");
queueMsgListener.setDestinationName(DataUtil.TEST_GROUP1_TOPIC);
queueMsgListener.setDestinationTopic(true);

try {
    queueMsgListener.run();
} catch (JMSEException e) {

    e.printStackTrace();
}
}

private static void consumeAllGroup2() {
ActiveMQMessageConsumer queueMsgListener = new ActiveMQMessageConsumer("tcp ←
    ://localhost:61616", "admin",
        "admin");
queueMsgListener.setDestinationName("*.group2.*");

try {
    queueMsgListener.run();
} catch (JMSEException e) {

    e.printStackTrace();
}
}

private static void exclusive_queue_Consumer() {
ActiveMQMessageConsumer queueMsgListener = new ActiveMQMessageConsumer("tcp ←
```

```

        ://localhost:61616", "admin",
        "admin");
queueMsgListener.setDestinationName(DataUtil.TEST_GROUP2_QUEUE_2 + "? ←
consumer.exclusive=true");

try {
    queueMsgListener.run();
} catch (JMSEException e) {

    e.printStackTrace();
}

}

private static void consume_queue_with_prefetchsize() {
    ActiveMQMessageConsumer queueMsgListener = new ActiveMQMessageConsumer("tcp ←
://localhost:61616", "admin",
    "admin");
queueMsgListener.setDestinationName(DataUtil.TEST_GROUP1_QUEUE_2 + "? ←
consumer.prefetchSize=10");

try {
    queueMsgListener.run();
} catch (JMSEException e) {

    e.printStackTrace();
}

}
}

```

- line 25: Consume from virtual topic `Consumer.zheng.VirtualTopic.Customer.Topic`
- line 38-39: Consume from virtual topic queue `VTC.DZONE.JCG.Mary.Topic` which message selector set as `action='DZONE'`
- line 51: Consume from topic `test.group1.topic`
- line 65: Consume from any queue name matches the `*.group2.*`
- line 78: Set exclusive message consumer. It will fail over if one consumer is down then the other will be picked to continue
- line 91: Set `preFetch` size for the consumer

3.9.3 Execution Output

Now, started the `ActiveMQMessageConsumerMainApp`. Here is the application output:

ActiveMQMessageConsumerMainApp Output

```

main: ActiveMQMessageConsumer Waiting for messages at Consumer.zheng.VirtualTopic.Customer. ←
Topic
main: ActiveMQMessageConsumer Waiting for messages at VTC.DZONE.JCG.Mary.Topic
main: ActiveMQMessageConsumer Waiting for messages at test.group1.topic
main: ActiveMQMessageConsumer Waiting for messages at *.group2.*
[ActiveMQ Session Task-1]: ActiveMQMessageConsumer Received message from [ *.group2.*] - ←
Headers: [ action=JCG actionId=40 ] Message: [ {"type":"NEWCUSTOMER","customerId":79} ]
main: ActiveMQMessageConsumer Waiting for messages at test.group1.queue2?consumer. ←
prefetchSize=10
[ActiveMQ Session Task-1]: ActiveMQMessageConsumer Received message from [ test.group1. ←
queue2?consumer.prefetchSize=10] - Headers: [ action=DEMO actionId=84 ] Message: [ {" ←
type":"NEWCUSTOMER","customerId":28} ]

```


Now execute `OnBoardNewConsumerApp` a couple times. Here you see two lines printed out from the running consumer application console as the output below.

ActiveMQMessageConsumerMainApp Output Continue

```
[ActiveMQ Session Task-1]: ActiveMQMessageConsumer Received message from [ Consumer.zheng. ←  
VirtualTopic.Customer.Topic] - Headers: [ action=CUSTOMER actionId=15 ] Message: [ {" ←  
type": "NEWCUSTOMER", "customerId":51} ]  
[ActiveMQ Session Task-2]: ActiveMQMessageConsumer Received message from [ Consumer.zheng. ←  
VirtualTopic.Customer.Topic] - Headers: [ action=CUSTOMER actionId=75 ] Message: [ {" ←  
type": "NEWCUSTOMER", "customerId":73} ]
```

Always verify and confirm via the AMQ web console.

3.10 Integration with Spring JMS

Spring JMS provides a JMS integration framework that simplifies the use of the JMS API.

3.10.1 Add Spring JMS dependency

Add Spring JMS library to the project `pom.xml`.

`pom.xml`

```
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-core</artifactId>  
    <version>4.1.5.RELEASE</version>  
</dependency>  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-context</artifactId>  
    <version>4.1.5.RELEASE</version>  
</dependency>  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-jms</artifactId>  
    <version>4.1.5.RELEASE</version>  
</dependency>
```

3.10.2 Configure Spring Beans

Add Spring JMS Beans to the context.

`JmsConfig`

```
package jcg.demo.spring.jms.config;  
  
import javax.jms.ConnectionFactory;  
  
import org.apache.activemq.ActiveMQConnectionFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.jms.annotation.EnableJms;  
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;  
import org.springframework.jms.connection.CachingConnectionFactory;
```

```
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.destination.DestinationResolver;
import org.springframework.jms.support.destination.DynamicDestinationResolver;

import jcg.demo.spring.jms.component.JmsExceptionListener;

@Configuration
@EnableJms
@ComponentScan(basePackages = "jcg.demo.spring.jms.component, jcg.demo.spring.service")
public class JmsConfig {

    private String concurrency = "1-10";
    private String brokerURI = "tcp://localhost:61616";

    @Autowired
    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory(
        JmsExceptionListener jmsExceptionListener) {
        DefaultJmsListenerContainerFactory factory = new
            DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(jmsConnectionFactory(jmsExceptionListener));
        factory.setDestinationResolver(destinationResolver());
        factory.setConcurrency(concurrency);
        factory.setPubSubDomain(false);
        return factory;
    }

    @Bean
    @Autowired
    public ConnectionFactory jmsConnectionFactory(JmsExceptionListener
        jmsExceptionListener) {
        return createJmsConnectionFactory(brokerURI, jmsExceptionListener);
    }

    private ConnectionFactory createJmsConnectionFactory(String brokerURI,
        JmsExceptionListener jmsExceptionListener) {
        ActiveMQConnectionFactory activeMQConnectionFactory = new
            ActiveMQConnectionFactory(brokerURI);
        activeMQConnectionFactory.setExceptionListener(jmsExceptionListener);

        CachingConnectionFactory connectionFactory = new CachingConnectionFactory(
            activeMQConnectionFactory);
        return connectionFactory;
    }

    @Bean(name = "jmsQueueTemplate")
    @Autowired
    public JmsTemplate createJmsQueueTemplate(ConnectionFactory jmsConnectionFactory) {
        return new JmsTemplate(jmsConnectionFactory);
    }

    @Bean(name = "jmsTopicTemplate")
    @Autowired
    public JmsTemplate createJmsTopicTemplate(ConnectionFactory jmsConnectionFactory) {
        JmsTemplate template = new JmsTemplate(jmsConnectionFactory);
        template.setPubSubDomain(true);
        return template;
    }

    @Bean
    public DestinationResolver destinationResolver() {
        return new DynamicDestinationResolver();
    }
}
```

```
    }  
}
```

As you seen here, the order to create these Beans is managed by the Spring Dependency Injection.

3.10.3 MessageSender

A class to send messages based on Spring JMS framework.

MessageSender

```
package jcg.demo.spring.jms.component;  
  
import java.util.Map;  
  
import javax.jms.JMSEException;  
import javax.jms.Message;  
import javax.jms.Session;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jms.core.JmsTemplate;  
import org.springframework.jms.core.MessageCreator;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MessageSender {  
  
    @Autowired  
    private JmsTemplate jmsQueueTemplate;  
  
    @Autowired  
    private JmsTemplate jmsTopicTemplate;  
  
    public void postToQueue(final String queueName, final String message) {  
  
        MessageCreator messageCreator = new MessageCreator() {  
  
            @Override  
            public Message createMessage(Session session) throws JMSEException {  
                return session.createTextMessage(message);  
            }  
        };  
  
        jmsQueueTemplate.send(queueName, messageCreator);  
  
    }  
  
    public void postToQueue(final String queueName, Map headers, final String message) ←  
    {  
  
        jmsQueueTemplate.send(queueName, new MessageCreator() {  
  
            @Override  
            public Message createMessage(Session session) throws JMSEException {  
                Message msg = session.createTextMessage(message);  
                headers.forEach((k, v) -> {  
                    try {  
                        msg.setStringProperty(k, v);  
                    } catch (JMSEException e) {  
                        System.out.println(  

```

```

        String.format("JMS fails to ←
                        set the Header value '% ←
                        s' to property '%s'", v, ←
                        k));
    }
    });
    return msg;
}
});
}

public void postToTopic(final String topicName, Map headers, final String message) ←
{
    jmsTopicTemplate.send(topicName, new MessageCreator() {

        @Override
        public Message createMessage(Session session) throws JMSEException {
            Message msg = session.createTextMessage(message);
            headers.forEach((k, v) -> {
                try {
                    msg.setStringProperty(k, v);
                } catch (JMSEException e) {
                    System.out.println(
                        String.format("JMS fails to ←
                                set the Header value '% ←
                                s' to property '%s'", v, ←
                                k));
                }
            });
            return msg;
        }
    });
}
}
}

```

As you seen here, the `MessageSender` is simpler than the `ActiveMQMessageProducer` created at step 3.8.1.

3.10.4 BillingAppListener

A listener listens the new customer events and integrates with billing system.

BillingAppListener

```

package jcg.demo.spring.jms.component;

import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.TextMessage;

import org.apache.activemq.command.ActiveMQQueue;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

import jcg.demo.jms.util.DataUtil;
import jcg.demo.model.CustomerEvent;
import jcg.demo.spring.service.BillingService;
import jcg.demo.spring.service.MessageTransformer;

```

```

@Component
public class BillingAppListener {

    @Autowired
    private JmsTemplate jmsQueueTemplate;

    @Autowired
    private BillingService billingService;

    @Autowired
    private MessageTransformer msgTransformer;

    private String queueName = "Consumer.Billing." + DataUtil.CUSTOMER_VTC_TOPIC;

    public String receiveMessage() throws JMSEException {
        System.out.println(Thread.currentThread().getName() + ": BillingAppListener ←
            receiveMessage.");

        Destination destination = new ActiveMQQueue(queueName);
        TextMessage textMessage = (TextMessage) jmsQueueTemplate.receive(←
            destination);

        CustomerEvent customerEvt = msgTransformer.fromJson(textMessage.getText(), ←
            CustomerEvent.class);
        return billingService.handleNewCustomer(customerEvt);
    }
}

```

As you seen here, this class is simpler than the `ActiveMQMessageConsumer` created at step 3.9.1.

3.10.5 SupportAppListener

A listener listens the new customer events and integrates with the support system.

SupportAppListener

```

package jcg.demo.spring.jms.component;

import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.TextMessage;

import org.apache.activemq.command.ActiveMQQueue;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

import jcg.demo.jms.util.DataUtil;
import jcg.demo.model.CustomerEvent;
import jcg.demo.spring.service.MessageTransformer;
import jcg.demo.spring.service.SupportService;

@Component
public class SupportAppListener {

    @Autowired
    private JmsTemplate jmsQueueTemplate;

    @Autowired
    private SupportService supportService;
}

```

```

@Autowired
private MessageTransformer msgTransformer;

private String queueName = "Consumer.Support." + DataUtil.CUSTOMER_VTC_TOPIC;

public String receiveMessage() throws JMSEException {
    System.out.println(Thread.currentThread().getName() + ": SupportAppListener ←
        receiveMessage.");

    Destination destination = new ActiveMQQueue(queueName);
    TextMessage textMessage = (TextMessage) jmsQueueTemplate.receive(←
        destination);

    CustomerEvent customerEvt = msgTransformer.fromJson(textMessage.getText(), ←
        CustomerEvent.class);
    return supportService.handleNewCustomer(customerEvt);
}
}

```

3.10.6 ConfigBillingforNewCustomerApp

Configure a Spring context to consume the new customer events to integrates with the billing system.

ConfigBillingforNewCustomerApp

```

package jcg.demo.spring.jms.app;

import java.net.URISyntaxException;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Configuration;

import com.google.gson.Gson;

import jcg.demo.spring.jms.component.BillingAppListener;
import jcg.demo.spring.jms.config.JmsConfig;

@Configuration
public class ConfigBillingForNewCustomerApp {
    public static void main(String[] args) throws URISyntaxException, Exception {
        Gson gson = new Gson();

        AnnotationConfigApplicationContext context = new ←
            AnnotationConfigApplicationContext(JmsConfig.class);
        context.register(ConfigBillingForNewCustomerApp.class);

        try {

            BillingAppListener billingAppListener = (BillingAppListener) ←
                context.getBean("billingAppListener");

            System.out.println("ConfigBillingForewCustomerApp receives " + ←
                billingAppListener.receiveMessage());

        } finally {
            context.close();
        }
    }
}

```

3.10.7 ConfigSupportforNewCustomerApp

Configure a Spring context to consume the new customer events to integrates with the support system.

ConfigSupportforNewCustomerApp

```
package jcg.demo.spring.jms.app;

import java.net.URISyntaxException;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Configuration;

import com.google.gson.Gson;

import jcg.demo.spring.jms.component.SupportAppListener;
import jcg.demo.spring.jms.config.JmsConfig;

@Configuration
public class ConfigSupportForNewCustomerApp {
    public static void main(String[] args) throws URISyntaxException, Exception {
        Gson gson = new Gson();

        AnnotationConfigApplicationContext context = new
            AnnotationConfigApplicationContext(JmsConfig.class);
        context.register(ConfigSupportForNewCustomerApp.class);

        try {
            SupportAppListener supportAppListener = (SupportAppListener)
                context.getBean("supportAppListener");
            System.out.println("supportAppListener receives " +
                supportAppListener.receiveMessage());
        } finally {
            context.close();
        }
    }
}
```

3.10.8 Run as Distributed Systems

By far, we built one Java JMS application - OnBoardNewCustomerApp and two Spring JMS applications: ConfigBillingForNewCustomerApp and ConfigSupportForNewCustomerApp. Now it's the time to run them together to enable the onboarding customer process integrates with both billing and support system.

ConfigBillingForNewCustomerApp Output

```
main: ConfigBillingForNewCustomerApp receiveMessage.
```

ConfigSupportForNewCustomerApp Output

```
main: ConfigSupportForNewCustomerAppreceiveMessage.
```

Execute the OnBoardNewCustomerApp. Here you will see both consumer received the customer message and processed them.

ConfigBillingForNewCustomerApp Output Continue

```
ConfigBillingForewCustomerApp receives BillingService handleNewCustomer CustomerEvent: type
(NEWCUSTOMER), customerId(41)
```

ConfigSupportForNewCustomerApp Output Continue

```
ConfigSupportForNewCustomerApp receives SupportService handleNewCustomer CustomerEvent: ←  
type (NEWCUSTOMER), customerId(41)
```

You just witnessed a working distributed system.

3.11 Integrating with Tomcat

3.11.1 Configure Tomcat Resource

Configure Tomcat context.xml with AMQ resource as below.

context.xml

```
<Resource name="jms/ConnectionFactory" global="jms/ConnectionFactory" auth="Container"  
  type="org.apache.activemq.ActiveMQConnectionFactory"  
  factory="org.apache.activemq.jndi.JNDIReferenceFactory"  
  brokerURL="tcp://localhost:61616"  
  userName="admin"  
  password="admin"  
  useEmbeddedBroker="false"/>
```

3.11.2 Look up JNDI Resource

Use `jndiContext.lookup` to look up the `ActiveMQConnectionFactory` from the JNDI resource.

JmsConfig

```
private ConnectionFactory createJmsConnectionFactory(String jndiName, JMSExceptionListener ←  
  exceptionListener) {  
    CachingConnectionFactory connectionFactory = null;  
    try {  
        Context jndiContext = new InitialContext();  
        Context envContext = (Context) jndiContext.lookup("java:comp/env");  
        ActiveMQConnectionFactory activeMQConnectionFactory = ( ←  
            ActiveMQConnectionFactory) envContext.lookup(jndiName);  
        connectionFactory = new CachingConnectionFactory( ←  
            activeMQConnectionFactory);  
        connectionFactory.setExceptionListener(exceptionListener);  
    } catch (NamingException e) {  
        String msg = String.format("Unable to get JMS container with name % ←  
            s ", jndiName);  
        throw new RuntimeException(msg, e);  
    }  
    return connectionFactory;  
}
```

3.12 Common Problems

There are three common problems when developing an ActiveMQ application.

3.12.1 Slow Consumer Application

When the AMQ console shows that there are growing numbers of pending messages. It indicates that the consumer's application is slower than the producer publishes the messages. There are several ways to address this issue:

- The publishers publish the messages with a similar speed to the consumers consuming the messages
- The publishers publish the messages to different destinations to reduce the total messages consumers consume
- The consumers improve the speed it takes to process the message by separating any long processes from the main thread to an asynchronous thread

3.12.2 ActiveMQ Sends Unwanted Messages to Virtual Topic Queue

There a **bug** found in an AMQ broker which sends unwanted messages to the virtual queue when selector is defined. Our solution is let the applications handle the selector by setting the `selectorAware` to false.

3.12.3 Exception Handler

Some applications redeliver the message back to destination when it encounters an exception. This may jam up the destination if it fails again. The better solution is to have separate exception handler to deal with any exceptions.

3.13 Summary

In this chapter, we outlined the steps to install the configure the AMQ server and demonstrated:

- how to install and configure
- how to build AMQ applications via ActiveMQ library
- how to build AMQ applications with Spring JMS framework
- how to integrate with Tomcat web container

We also described three common problems when developing an AMQ application.

3.14 Download the Source Code

This example builds several java applications to send and receive messages via the AMQ broker. You can download the full source code of this example here: [Apache ActiveMQ Best Practices Tutorial](#)

Chapter 4

ActiveMQ Distributed Queue Tutorial

Apache ActiveMQ (AMQ) is an open source messaging server written in Java, which implements JMS 1.1 specifications. In this article, I will demonstrate how to utilize a distributed queue within a group of AMQ brokers.

4.1 Introduction

Apache ActiveMQ (AMQ) is a message broker which transfers the messages from the sender to the receiver.

A **distributed queue** is a single unit of Java Message Service (JMS) queues that are accessible as a single, logical queue to a client. The members of the unit are usually distributed across multiple servers within a cluster, with each queue member belonging to a separate JMS server.

AMQ provides network connectors to connect AMQ servers as a cluster. In a network of AMQ servers, the messages in a queue at Broker A can be consumed by a client from a different broker.

In this example, I will demonstrate how a distributed queue works in AMQ brokers.

4.2 ActiveMQ Server Installation

Follow these [instructions](#) to install an AMQ server. Then use the AMQ admin command: `activemq-admin create ${brokerName}` to create a server instance.

Click [here](#) for details.

4.3 Producer Java Application

4.3.1 MessageProducerApp

Create MessageProducerApp.

MessageProducerApp.java

```
package jcg.demo.activemq;

import jcg.demo.util.DataUtils;
import jcg.demo.util.InputData;

public class MessageProducerApp {

    public static void main(String[] args) {
```

```
        InputData brokerTestData = DataUtils.readTestData();

        if (brokerTestData == null) {
            System.out.println("Wrong input");
        } else {
            QueueMessageProducer queProducer = new QueueMessageProducer( ←
                brokerTestData.getBrokerUrl(), DataUtils.ADMIN,
                DataUtils.ADMIN);
            queProducer.sendDummyMessages(brokerTestData.getQueueName());
        }
    }
}
```

4.3.2 QueueMessageProducer

Create QueueMessageProducer.

QueueMessageProducer.java

```
package jcq.demo.activemq;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

import jcq.demo.util.DataUtils;

/**
 * A simple message producer which sends the message to ActiveMQ Broker
 *
 * @author Mary.Zheng
 *
 */
public class QueueMessageProducer {

    private String activeMqBrokerUri;
    private String username;
    private String password;

    public QueueMessageProducer(String activeMqBrokerUri, String username, String ←
        password) {
        super();
        this.activeMqBrokerUri = activeMqBrokerUri;
        this.username = username;
        this.password = password;
    }

    public void sendDummyMessages(String queueName) {
        System.out.println("QueueMessageProducer started " + this.activeMqBrokerUri ←
            );
        ConnectionFactory connFactory = null;
        Connection connection = null;
        Session session = null;
        MessageProducer msgProducer = null;
        try {
```

```

        connFactory = new ActiveMQConnectionFactory(username, password, ←
            activeMqBrokerUri);
        connection = connFactory.createConnection();
        connection.start();
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE) ←
            ;
        msgProducer = session.createProducer(session.createQueue(queueName) ←
            );

        for (int i = 0; i < DataUtils.MESSAGE_SIZE; i++) {
            TextMessage textMessage = session.createTextMessage( ←
                DataUtils.buildDummyMessage(i));
            msgProducer.send(textMessage);
            Thread.sleep(30000);
        }
        System.out.println("QueueMessageProducer completed");
    } catch (JMSEException | InterruptedException e) {
        System.out.println("Caught exception: " + e.getMessage());
    }
    try {
        if (msgProducer != null) {
            msgProducer.close();
        }
        if (session != null) {
            session.close();
        }
        if (connection != null) {
            connection.close();
        }
    } catch (Throwable ignore) {
    }
}
}

```

- Line 49: Sleep 30 seconds after sending a message for demonstrating a slow producer

4.3.3 Export MessageProducerApp as a Jar

Export MessageProducerApp as activemq-msgproducerApp.jar

4.4 Consumer Java Application

4.4.1 MessageConsumerApp

Create MessageConsumerApp.

MessageConsumerApp.java

```

package jcg.demo.activemq;

import javax.jms.JMSEException;

import jcg.demo.util.DataUtils;
import jcg.demo.util.InputData;

public class MessageConsumerApp {

    public static void main(String[] args) {

```

```
        InputData brokerTestData = DataUtils.readTestData();
        if (brokerTestData == null) {
            System.out.println("Wrong input");
        } else {
            QueueMessageConsumer queueMsgListener = new QueueMessageConsumer( ←
                brokerTestData.getBrokerUrl(), DataUtils.ADMIN,
                DataUtils.ADMIN);
            queueMsgListener.setDestinationName(brokerTestData.getQueueName());

            try {
                queueMsgListener.run();
            } catch (JMSEException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- Line 16: Test AMQ broker URL
- Line 18: Test queue name

4.4.2 QueueMessageConsumer

Create QueueMessageConsumer.

QueueMessageConsumer.java

```
package jcg.demo.activemq;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQTextMessage;

/**
 * A simple message consumer which consumes the message from ActiveMQ Broker
 *
 * @author Mary.Zheng
 *
 */
public class QueueMessageConsumer implements MessageListener {

    private String activeMqBrokerUri;
    private String username;
    private String password;
    private String destinationName;

    public QueueMessageConsumer(String activeMqBrokerUri, String username, String ←
        password) {
        super();
        this.activeMqBrokerUri = activeMqBrokerUri;
        this.username = username;
    }
}
```

```

        this.password = password;
    }

    public void run() throws JMSEException {
        ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(username, ←
            password, activeMqBrokerUri);
        Connection connection = factory.createConnection();
        connection.setClientID(getClientId());
        connection.start();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE) ←
            ;

        Destination destination = session.createQueue(destinationName);

        MessageConsumer consumer = session.createConsumer(destination);
        consumer.setMessageListener(this);

        System.out.println(String.format("QueueMessageConsumer Waiting for messages ←
            at queue='%s' broker='%s'",
                destinationName, this.activeMqBrokerUri));
    }

    @Override
    public void onMessage(Message message) {
        if (message instanceof ActiveMQTextMessage) {
            ActiveMQTextMessage amqMessage = (ActiveMQTextMessage) message;
            try {
                String msg = String.format("QueueMessageConsumer Received ←
                    message [ %s ]", amqMessage.getText());
                System.out.println(msg);
            } catch (JMSEException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("QueueMessageConsumer Received non-text message" ←
                );
        }
    }

    public String getDestinationName() {
        return destinationName;
    }

    public void setDestinationName(String destinationName) {
        this.destinationName = destinationName;
    }

    private String getClientId() {
        return "MzhengClient_" + destinationName + "_" + activeMqBrokerUri.replace( ←
            "tcp://localhost:", "");
    }
}

```

- Line 37: Set connection clientID
- Line 74: Set client Id from the queue name and the broker port

4.4.3 Common Utils

Create DataUtils .

DataUtils.java

```

package jcg.demo.util;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Scanner;

import org.springframework.util.StringUtils;

/**
 * The data utility used in this Demo
 *
 * @author Mary.Zheng
 *
 */
public final class DataUtils {

    private static final String INPUT_PROMPT_1 = "Enter Broker URL(tcp://$host:$port): ←
        ";
    private static final String INPUT_PROMPT_2 = "Enter Queue Name: ";
    public static final int MESSAGE_SIZE = 10;
    public static final String ADMIN = "admin";

    public static String buildDummyMessage(int value) {
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        LocalDateTime now = LocalDateTime.now();

        return "dummy message [" + value + "], created at " + dtf.format(now);
    }

    public static InputData readTestData() {
        InputData testData = null;

        try (Scanner scanIn = new Scanner(System.in)) {
            System.out.println(INPUT_PROMPT_1);
            String brokerUrl = scanIn.nextLine();

            System.out.println(INPUT_PROMPT_2);
            String queueName = scanIn.nextLine();

            if (StringUtils.isEmpty(queueName) || StringUtils.isEmpty(brokerUrl ←
                )) {
                return testData;
            }
            testData = new InputData( brokerUrl, queueName);
        }

        return testData;
    }
}

```

- Line 26: Include the message born time in the message body for demonstrating purpose

Create InputData to hold the testing data.

InputData.java

```

package jcg.demo.util;

/**

```

```
* The input data for this demo.
*
* @author Mary.Zheng
*
*/
public class InputData {

    private String brokerUrl;
    private String queueName;

    public InputData(String brokerUrl, String queueName) {
        super();
        this.brokerUrl = brokerUrl.trim();
        this.queueName = queueName.trim();
    }

    public String getBrokerUrl() {
        return brokerUrl;
    }

    public void setBrokerUrl(String brokerUrl) {
        this.brokerUrl = brokerUrl;
    }

    public String getQueueName() {
        return queueName;
    }

    public void setQueueName(String queueName) {
        this.queueName = queueName;
    }
}
```

4.4.4 Export MessageConsumerApp as a Jar

Export MessageConsumerApp as activemq-msgConsumerApp.jar

4.5 Distributed Queue in a Static Network of Brokers

In this example, Producer-1 sends messages to Queue.1 at Broker-1. Consumer-1 receives the messages from Queue.1 at Broker-3. Queue.1 is the distributed queue. It's useful when the producer and consumer applications cannot be in the same AMQ server.

Image below shows a distribute queue (queue.1) in Brokers-1 and Broker-3 .

Apache ActiveMQ Distributed Queue – Static Network of Brokers

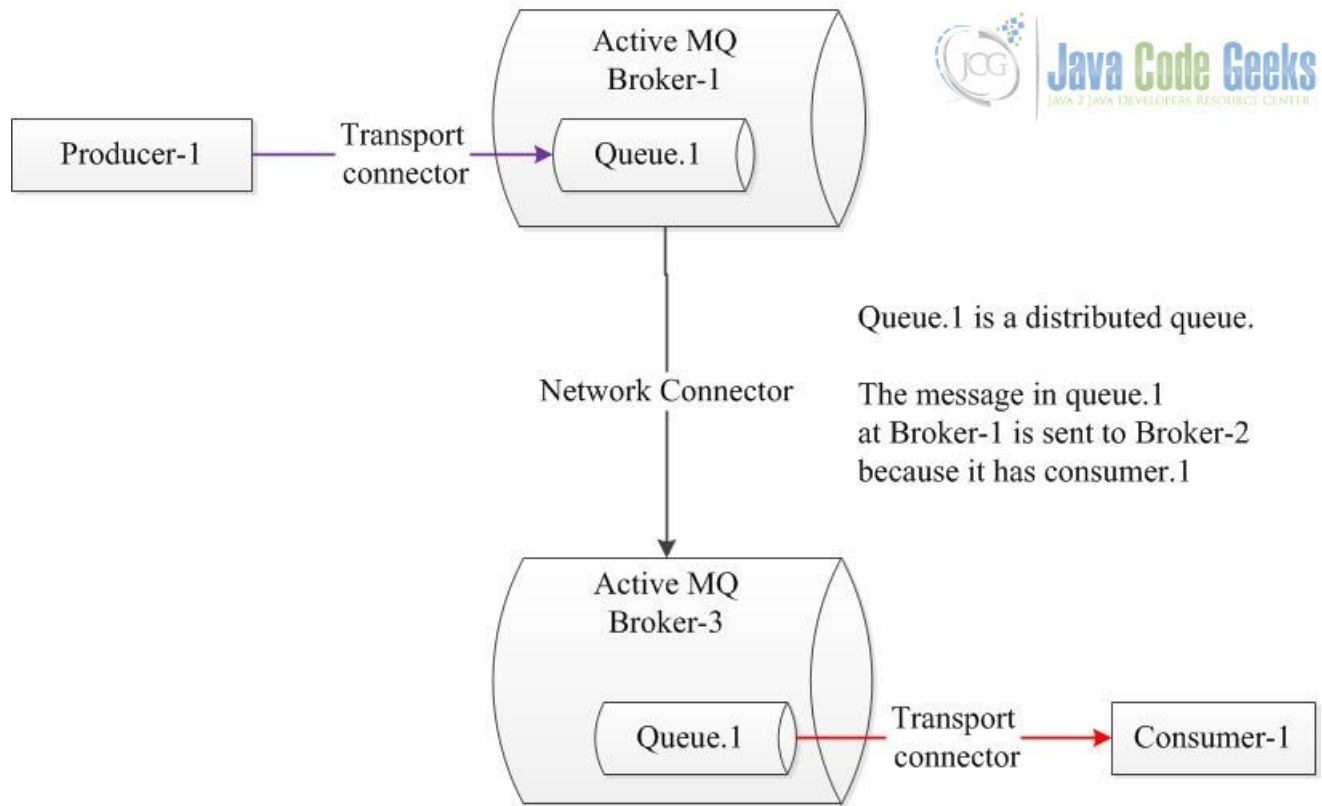


Figure 4.1: distributed queue 1

4.5.1 Configure a Static Network of Brokers

Configure a network of Broker-1 and Broker-3 :

broker-1	..\cluster\broker-1	61816	8861	..\data
broker-3	..\cluster\broker-3	61516	5161	\broker-3\data

Click [here](#) for the configuration details.

4.5.2 Verify the AMQ Brokers - Part I

Start both Broker-1 and Broker-3 .

Navigate to the AMQ web console to view the connections details.

localhost:8861/admin/connections.jsp

Apps Managed bookmarks Allen's Folder applications confluen activeMQ Bookmarks Post

Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send

Connections

Connector openwire

Name ↑	Remote Address	Active	Slow
MzhengClient:queue.1:61816	tcp://127.0.0.1:55408	true	false
nc:61516-61816:broker-3:outbound	tcp://127.0.0.1:55633	true	false

Network Connectors

Name	Message TTL	Consumer TTL	Dynamic Only	Conduit Subscriptions	Bridge Temps	Decrease Priorities

Copyright 2005-2015 The Apache Software Foundation.

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Figure 4.2: Broker-1 connection

Note: Broker-1 connection client name is defined at step 4.4.2.

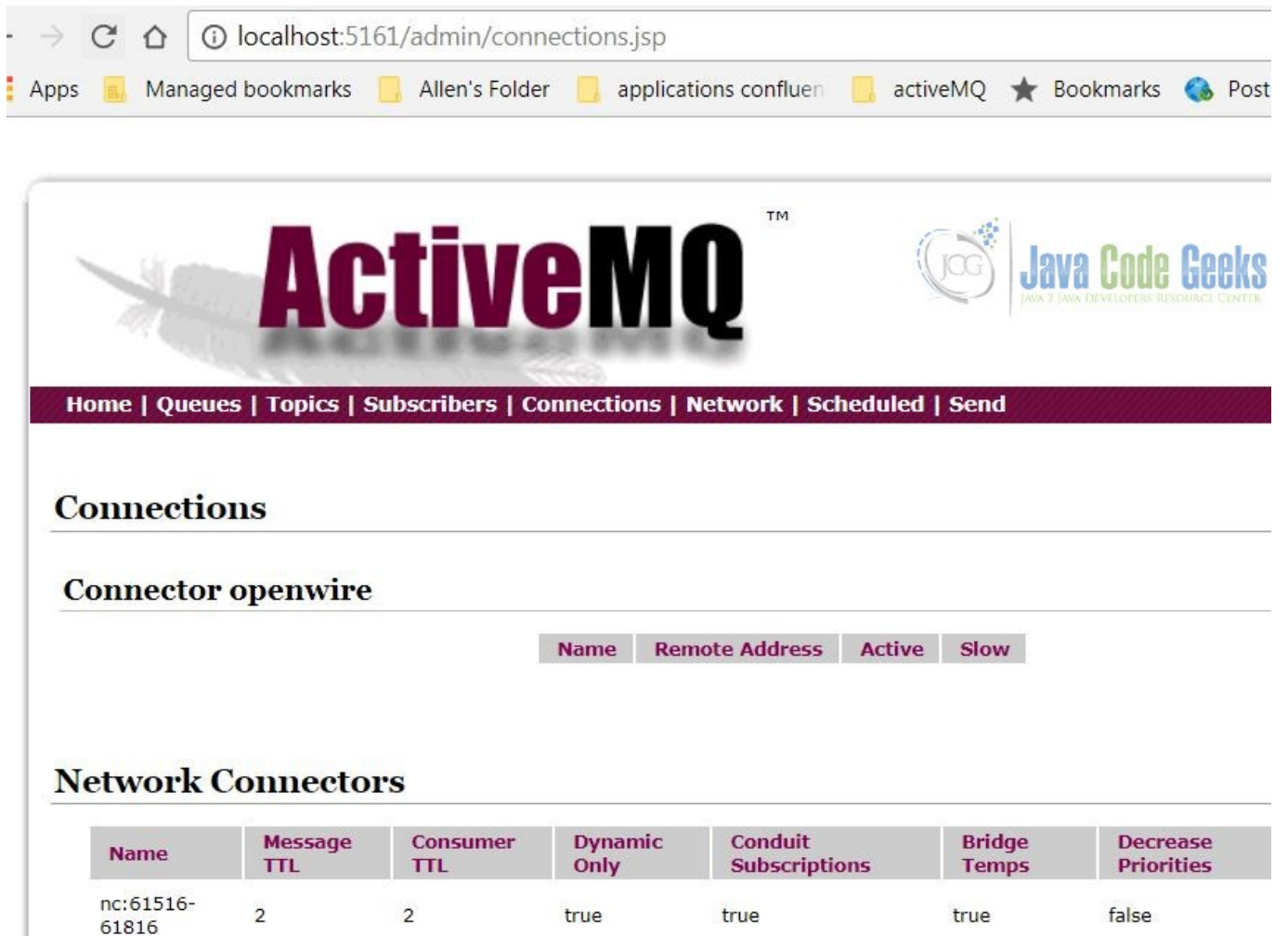


Figure 4.3: Broker-3 connection

Note: Broker-3 has a network connector to Broker-1.

4.5.3 Execute the Consumer Application

Enter `java -jar activemq-msgConsumerApp.jar` to start MessageConsumerApp .

MessageConsumerApp Output

```
C:\Users\shu.shan\Desktop>java -jar activemq-msgConsumerApp.jar
Enter Broker URL(tcp://$host:$port):
tcp://localhost:61816
Enter Queue Name:
queue.1
QueueMessageConsumer Waiting for messages at queue='queue.1' broker='tcp://localhost:61816'
```

- Line 3: Enter Broker-1 URL
- Line 5: Enter queue name queue . 1

4.5.4 Execute the Publisher Application

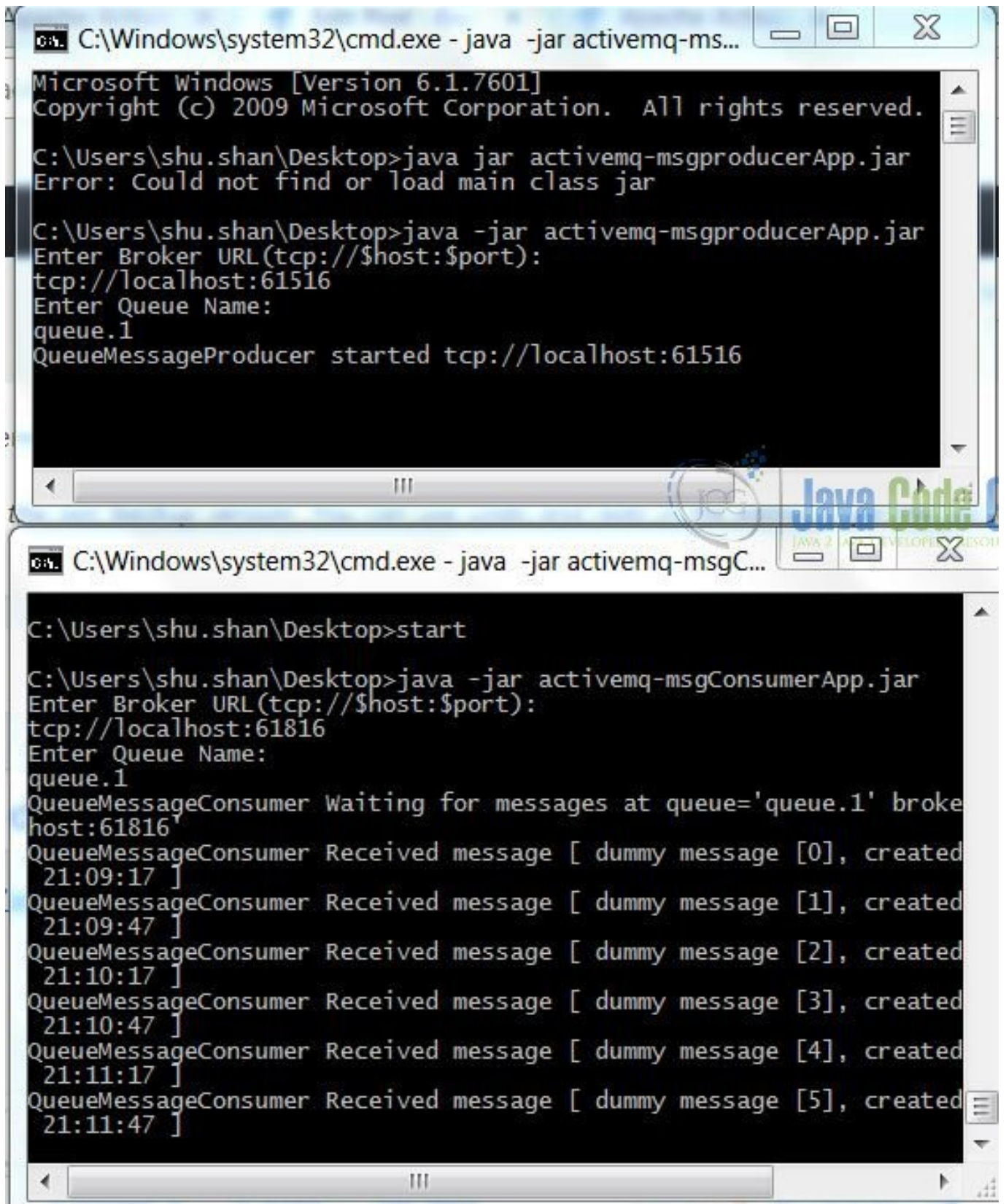
While MessageConsumerApp is running, enter `java -jar activemq-msgproducerApp` to start MessageProducerApp.

MessageProducerApp output

```
C:\Users\shu.shan\Desktop>java -jar activemq-msgproducerApp.jar
Enter Broker URL(tcp://$host:$port):
tcp://localhost:61516
Enter Queue Name:
queue.1
QueueMessageProducer started tcp://localhost:61516
```

- Line 3: Enter Broker-3 URL
- Line 5: Enter queue name `queue.1`

Image below shows both applications are running.



```
C:\Windows\system32\cmd.exe - java -jar activemq-ms...
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\shu.shan\Desktop>java jar activemq-msgproducerApp.jar
Error: Could not find or load main class jar

C:\Users\shu.shan\Desktop>java -jar activemq-msgproducerApp.jar
Enter Broker URL(tcp://$host:$port):
tcp://localhost:61516
Enter Queue Name:
queue.1
QueueMessageProducer started tcp://localhost:61516

C:\Windows\system32\cmd.exe - java -jar activemq-msgC...
C:\Users\shu.shan\Desktop>start

C:\Users\shu.shan\Desktop>java -jar activemq-msgConsumerApp.jar
Enter Broker URL(tcp://$host:$port):
tcp://localhost:61816
Enter Queue Name:
queue.1
QueueMessageConsumer waiting for messages at queue='queue.1' broke
host:61816'
QueueMessageConsumer Received message [ dummy message [0], created
21:09:17 ]
QueueMessageConsumer Received message [ dummy message [1], created
21:09:47 ]
QueueMessageConsumer Received message [ dummy message [2], created
21:10:17 ]
QueueMessageConsumer Received message [ dummy message [3], created
21:10:47 ]
QueueMessageConsumer Received message [ dummy message [4], created
21:11:17 ]
QueueMessageConsumer Received message [ dummy message [5], created
21:11:47 ]
```

Figure 4.4: Execution of Application

4.5.5 Verify the AMQ Brokers - Part II

Navigate to Broker-1 web console, click queues to see queue . 1 , lastly, click on its active consumers link.

Image Below shows queue . 1 's active consumer - Mzhengclient-queue . 1_61816 at broker-1 .

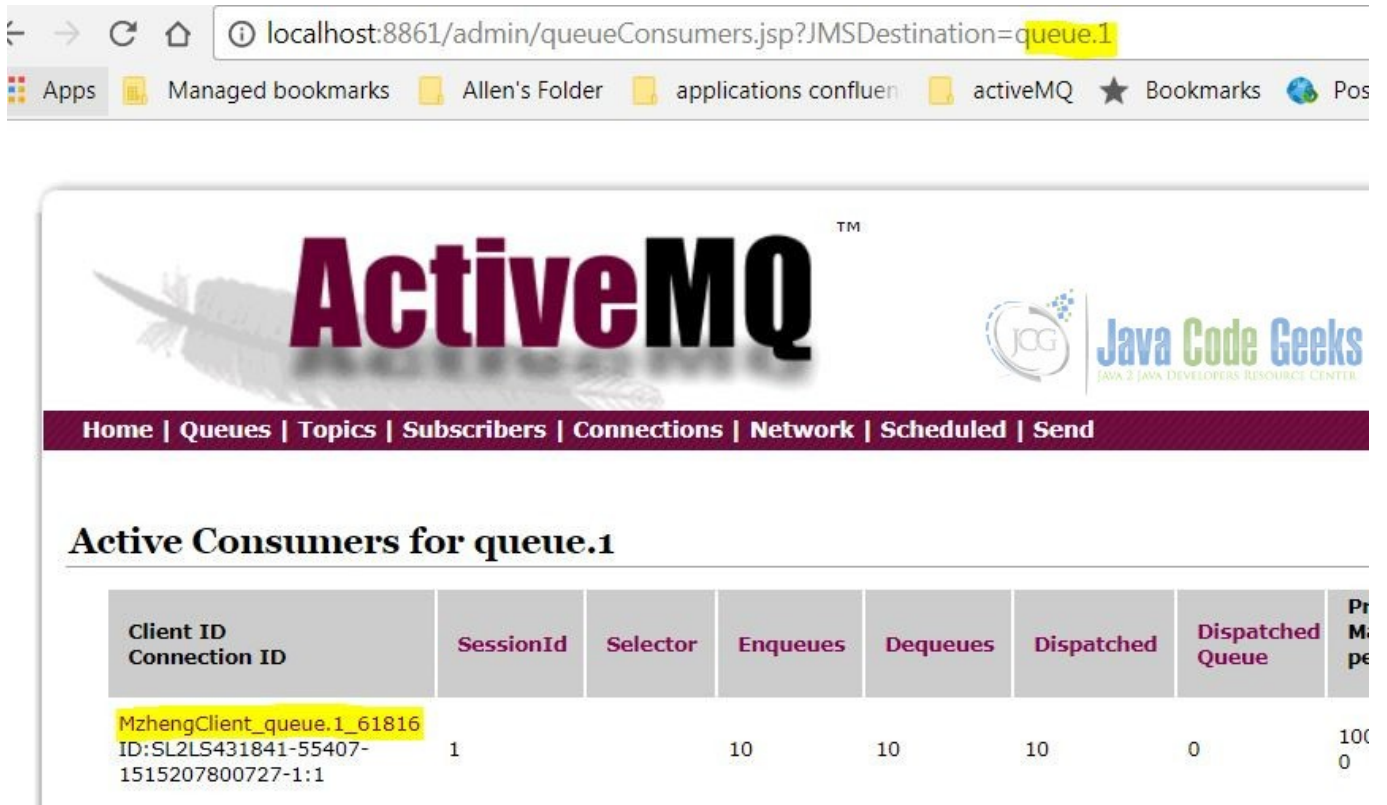


Figure 4.5: Broker-1 queue.1 consumer

Image Below shows queue . 1 's active consumer - nc:61516-61816_broker-1_inbound_broker-3 at broker-3 .



Figure 4.6: Broker-3 Consumer

Note: queue . 1 is the distributed queue via the broker’s connect connector.

4.6 Distributed Queue in a Dynamic Network of Brokers

In this example, Producer-1 sends messages to queue . 1 at Dynamic-Broker1 , Producer-2 also sends messages to queue . 1 at Dynamic-Broker2 , Consumer-1 receives the messages from Queue . 1 at Dynamic-Broker3 . Queue . 1 is the distributed queue. It’s useful to share the load among multiple producers and support in-order delivery when processing the messages.

Diagram below shows a distributed queue(Queue . 1) among three brokers.

Apache ActiveMQ Distributed Queue – Dynamic Network of Brokers

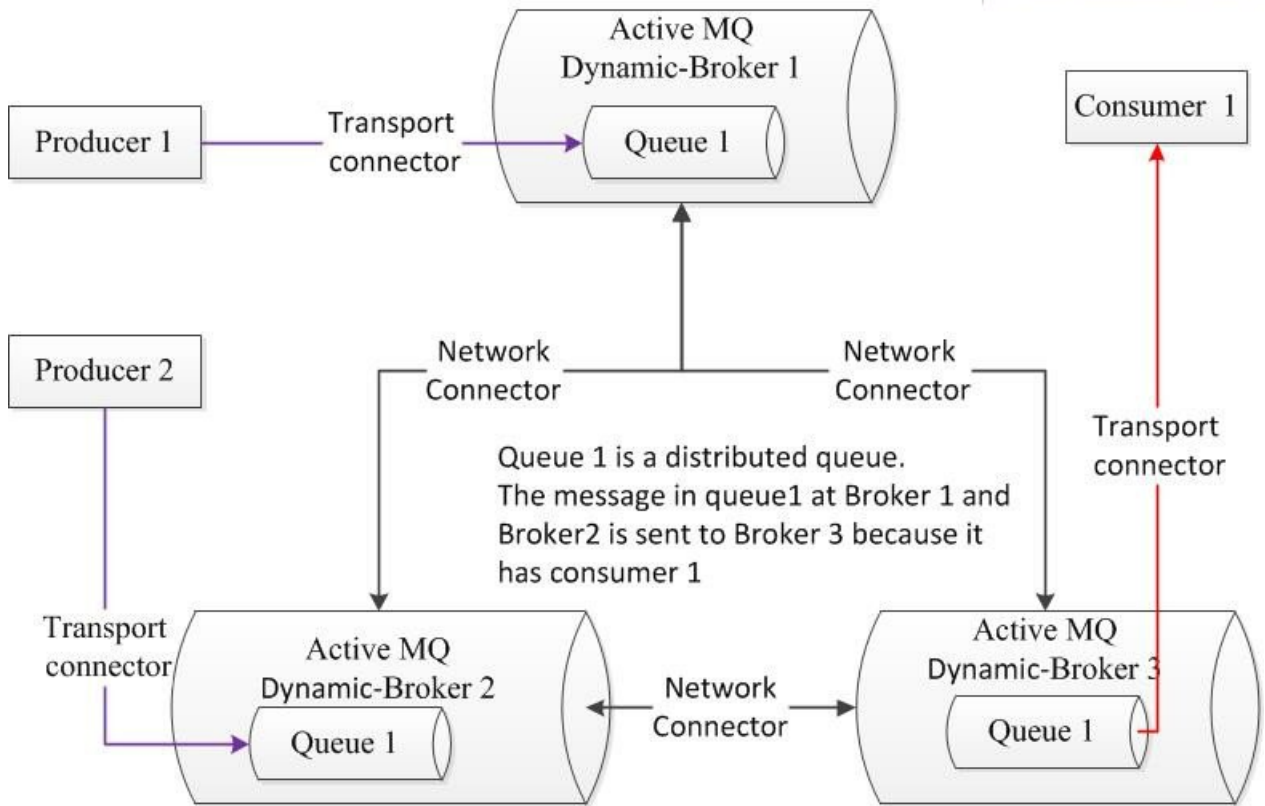


Figure 4.7: distributed queue

4.6.1 Configure a Dynamic Network of Brokers

Configure a dynamic network of brokers with three brokers:

dynamic-broker1	..\cluster\dynamic-broker1	61626	8166	..\dynamic-broker1\data
dynamic-broker2	..\cluster\dynamic-broker2	61636	8164	..\dynamic-broker2\data
dynamic-broker3	..\cluster\dynamic-broker3	61646	8165	..\dynamic-broker3\data

Click [here](#) for the configuration details.

4.6.2 Verify the AMQ Brokers - Part I

Start all three dynamic-brokers. Navigate to the AMQ web console to view the connections details.

Image below shows `Dynamic-broker1 (8166)` connections.

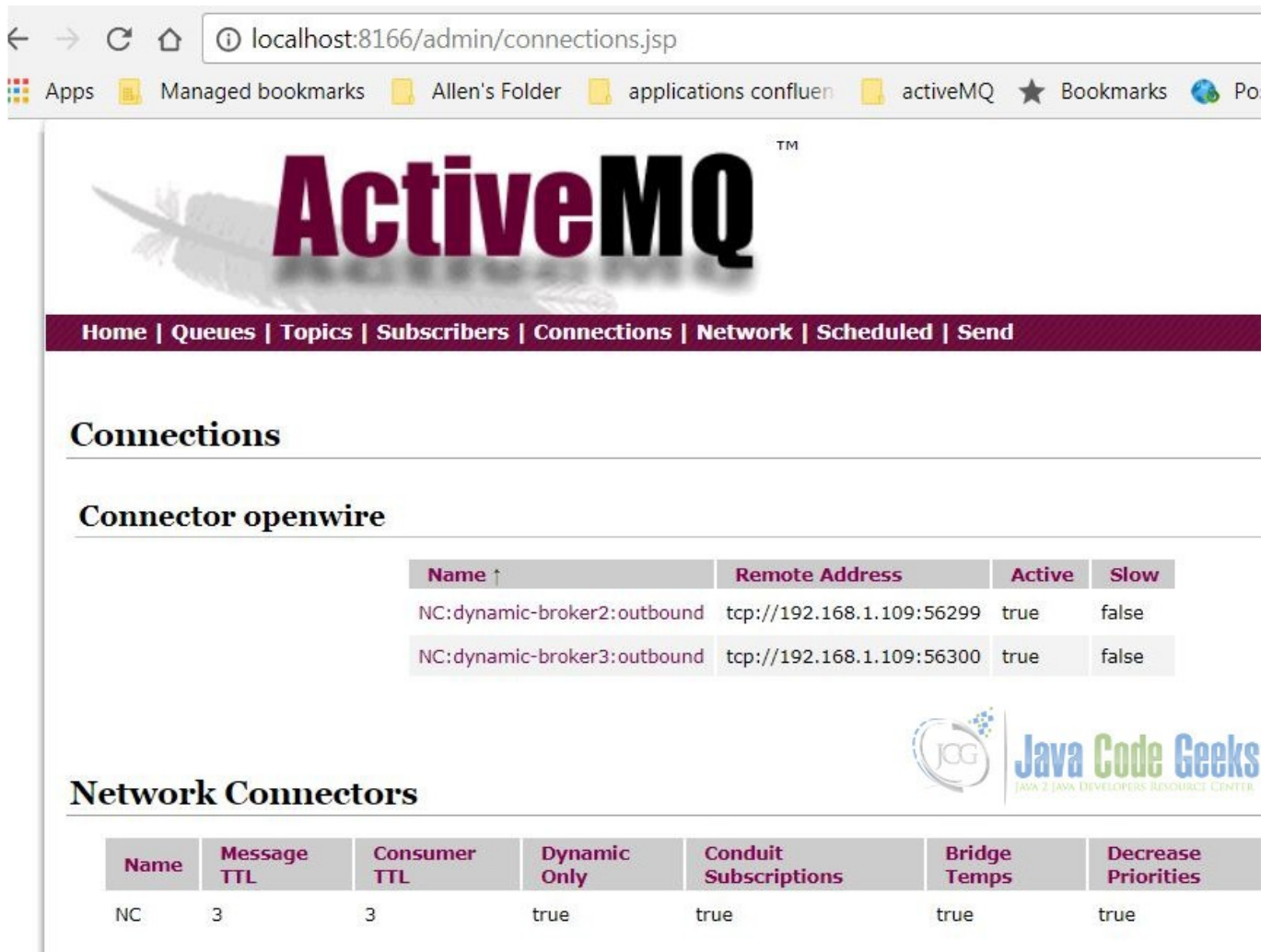


Figure 4.8: Dynamic-Broker1 Connections

4.6.3 Execute the Consumer Application

Enter `java -jar activemq-msgConsumerApp.jar` to start MessageConsumerApp at Dynamic-broker2 .

MessageConsumerApp Output

```
C:\\Users\\shu.shan\\Desktop>java -jar activemq-msgConsumerApp.jar
Enter Broker URL(tcp://$host:$port):
tcp://localhost:61636
Enter Queue Name:
queue.1
QueueMessageConsumer Waiting for messages at queue='queue.1' broker='tcp://localhost:61636'
```

4.6.4 Execute the Publisher Application

While MessageConsumerApp is running, enter `java -jar activemq-msgproducerApp` to start MessageProducerApp twice, one for Dynamic-broker1 , the other for Dynamic-broker3 .

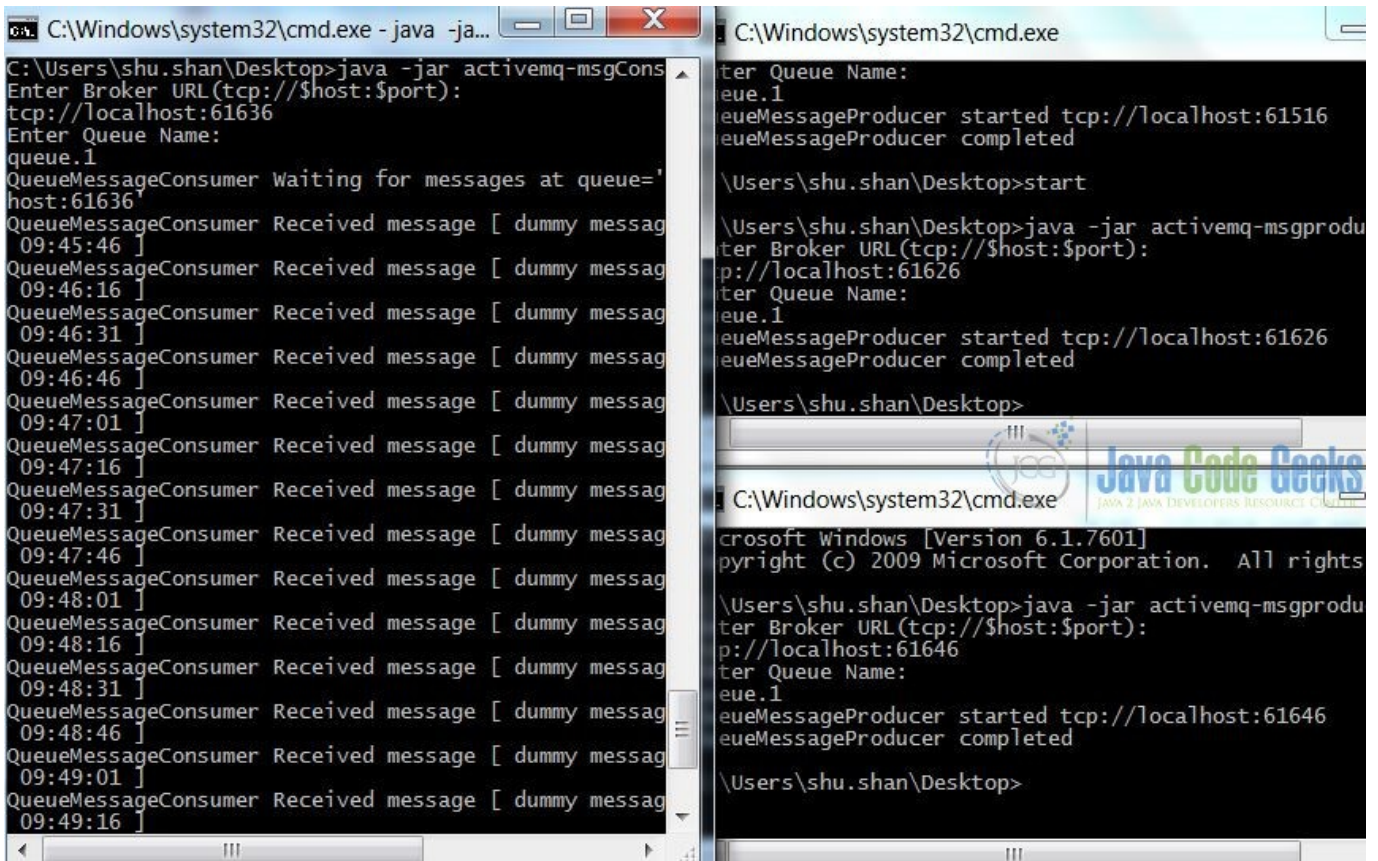


Figure 4.9: Application Execution Output

Note: The consumer listens to queue.1 at Dynamic-Broker2 while two publishers publish the messages to queue.1 at Dynamic-Broker1 and Dynamic-Broker3. The consumer processed the messages based on the message's born time.

4.6.5 Verify the AMQ Brokers - Part II

Navigate to Dynamic-Broker2 web console, click queues to see queue.1, lastly, click on its active consumers link.

Image below shows queue.1's active consumer - Mzhengclient-queue.1_61636 at broker-3.

localhost:8164/admin/queueConsumers.jsp?JMSDestination=queue.1

Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send

Active Consumers for queue.1

Client ID Connection ID	SessionId	Selector	Enqueues	Dequeues	Dispatched	Dispatched Queue	Pre Max per
MzhengClient_queue.1_61636 ID:SL2LS431841-56536-1515252333491-1:1	1		0	0	0	0	1000

Figure 4.10: Dynamic-Broker2 Consumer

Image below shows queue.1 at Dynamic-broker3 has two active consumers via the network of brokers.

The screenshot shows the ActiveMQ web console interface. At the top, the browser address bar displays `localhost:8165/admin/queueConsumers.jsp?JMSDestination=queue.1`. The page header includes navigation links: Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send. The main heading is "Active Consumers for queue.1". Below this is a table listing active consumers.

Client ID Connection ID ↑	SessionId	Selector	Enqueues	Dequeues	Dispatched	Dispatched Queue	Pre Max pen
NC_dynamic-broker1_inbound_dynamic-broker3 dynamic-broker3->dynamic-broker1-56430-1515252105800-7:1	1		0	0	0	0	1 0
NC_dynamic-broker2_inbound_dynamic-broker3 dynamic-broker3->dynamic-broker2-56430-1515252105800-5:1	1		0	0	0	0	1 0

Figure 4.11: Dynamic-Broker3 consumers

Note: `queue.1` is the distributed queue via the broker's connect connector.

4.7 Summary

In this article, I demonstrated two cases of a distributed queue by utilizing AMQ with a network of brokers. AMQ network of brokers also provides high availability to the client. Click [here](#) for more details on high availability.

Distributed Queue provides support for deliveries where subscribers receives messages in the same order they have been published. Besides Apache ActiveMQ, [IBM MQ](#), [RabbitMQ](#), [HornetQ](#), and [Apache Kafka](#) also support Distributed Queues.

4.8 Download the Source Code

This example builds two java applications to send and receive messages via the AMQ broker. You can download the full source code of this example here: [Apache ActiveMQ Distributed Queue Tutorial](#)

Chapter 5

ActiveMQ BrokerService Example

Using this example, we will be learning about Apache ActiveMQ BrokerService and various other ways in which a broker in ActiveMQ can be started and configured. But before we begin with our example, it is expected that we have a basic understanding of **JMS concepts**, **ActiveMQ** and **Java/J2EE**. JMS stands for Java Messaging API and Apache ActiveMQ is an open source message broker written in Java together with a full Java Message Service (JMS) client.

Apache ActiveMQ is packed up with enormous features but via this example, we will see how we embed a broker inside a connection and the usage of ActiveMQ BrokerService API.

5.1 Introduction

ActiveMQ has a BrokerService API that manages the lifecycle of an ActiveMQ Broker. In order to exchange messages, producers and consumers need to connect to the broker. If one wants to connect to the broker over the network then a transport connector is needed. If the client is within the same application as the broker and both are sharing the same JVM, then the broker can be connected using Virtual Machine Protocol. Based on the configured transport connector, broker knows how to accept and listen to connections from clients.

The `org.apache.activemq.broker.BrokerService` class implements the `Service` interface and has two subclasses:

- `SslBrokerService`
- `XBeanBrokerService`

The `org.apache.activemq.broker.SslBrokerService` is a `BrokerService` that allows access to the key and trust managers used by SSL connections. There is no reason to use this class unless SSL is being used AND the key and trust managers need to be specified from within the code. In fact, if the URI passed to this class does not have an "ssl" scheme, this class will pass all the work on to its superclass.

The `org.apache.activemq.broker.XBeanBrokerService` is an ActiveMQ Message Broker. It consists of a number of transport connectors, network connectors and a bunch of properties which can be used to configure the broker as its lazily created.

Lets see few examples on how to use `BrokerService` API and how to embed a broker in ActiveMQ.

5.2 Embed a broker in ActiveMQ

In many messaging topologies there are JMS Brokers (server side) and a JMS client side. Often it makes sense to deploy a broker within your JVM to optimise away a network hop; making the networking of JMS as efficient as pure RMI, but with all the usual JMS features of location independence, reliability, load balancing etc.

There are various ways to embed a broker in ActiveMQ depending on if we are using Java, Spring, XBean or using the Broker-Factory.

5.2.1 Embed a broker in ActiveMQ using Java code (using BrokerService API)

We can use embedded ActiveMQ which means we can create object of `BrokerService` class and then use java code to configure it instead of regular approach of using `activemq.xml` file. Lets try that by creating a sample application. First create `EmbeddedBrokerService` class. In this class we will just create object of `BrokerService`. Add a connector and then call `brokerService.start()` to start the broker. After that we will use our `MessageSender` and `MessageReceiver` classes to send and receive a text message.

Please refer the code snippet below for our `EmbeddedBrokerService` class. The `MessageSender` and `MessageReceiver` classes can be referred from [here](#).

`EmbeddedBrokerService.java`

```
import org.apache.activemq.broker.BrokerService;

public class EmbeddedBrokerService {

    public static void main(String[] args) throws Exception {
        BrokerService broker = new BrokerService();
        broker.setUseJmx(true);
        broker.addConnector("tcp://localhost:61616");
        broker.start();
        System.out.println("Broker Started!!!");
        // now lets wait forever to avoid the JVM terminating immediately
        Object lock = new Object();
        synchronized (lock) {
            lock.wait();
        }
    }
}
```

Output:

We will be running our `EmbeddedBrokerService` class which starts the broker and then run the `MessageSender` and `MessageReceiver` classes to see how a message is exchanged.

Please follow the steps below:

- In the eclipse Right Click on `EmbeddedBrokerService.java` → Run As→Java Application, to start the broker
- In the same way run the `MessageSender` and `MessageReceiver` classes by Right Click on the class → Run As→Java Application to see if our message is sent to the queue. The Hello message after being successfully sent to the queue gets printed in eclipse output console
- The same message after being received from the queue gets printed in eclipse output console
- Please refer images below to check the output of all the three classes

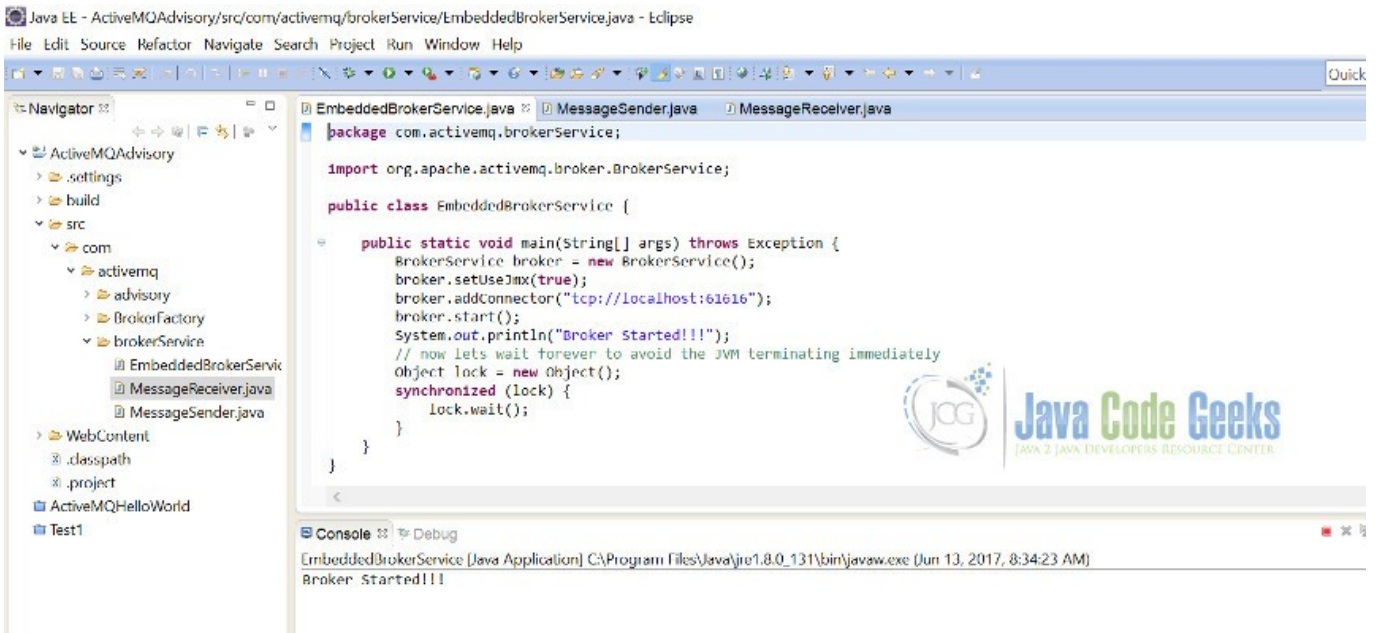


Figure 5.1: Eclipse console showing BrokerService is started

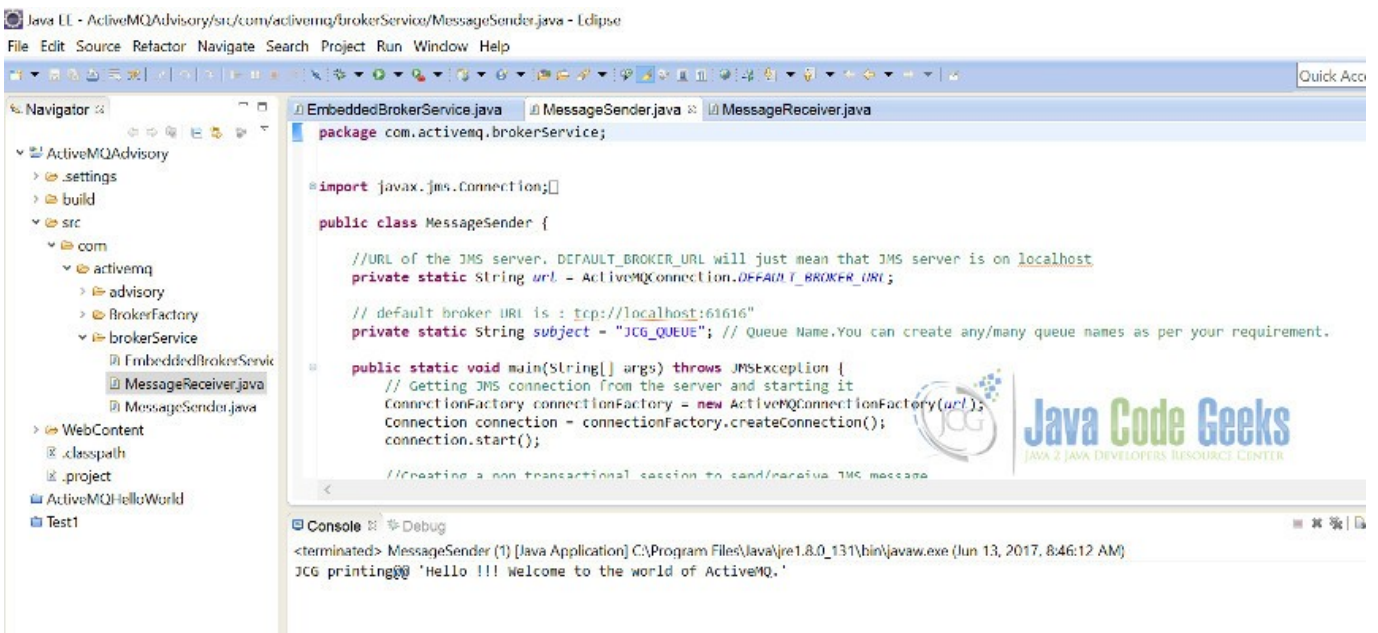


Figure 5.2: MessageSender output

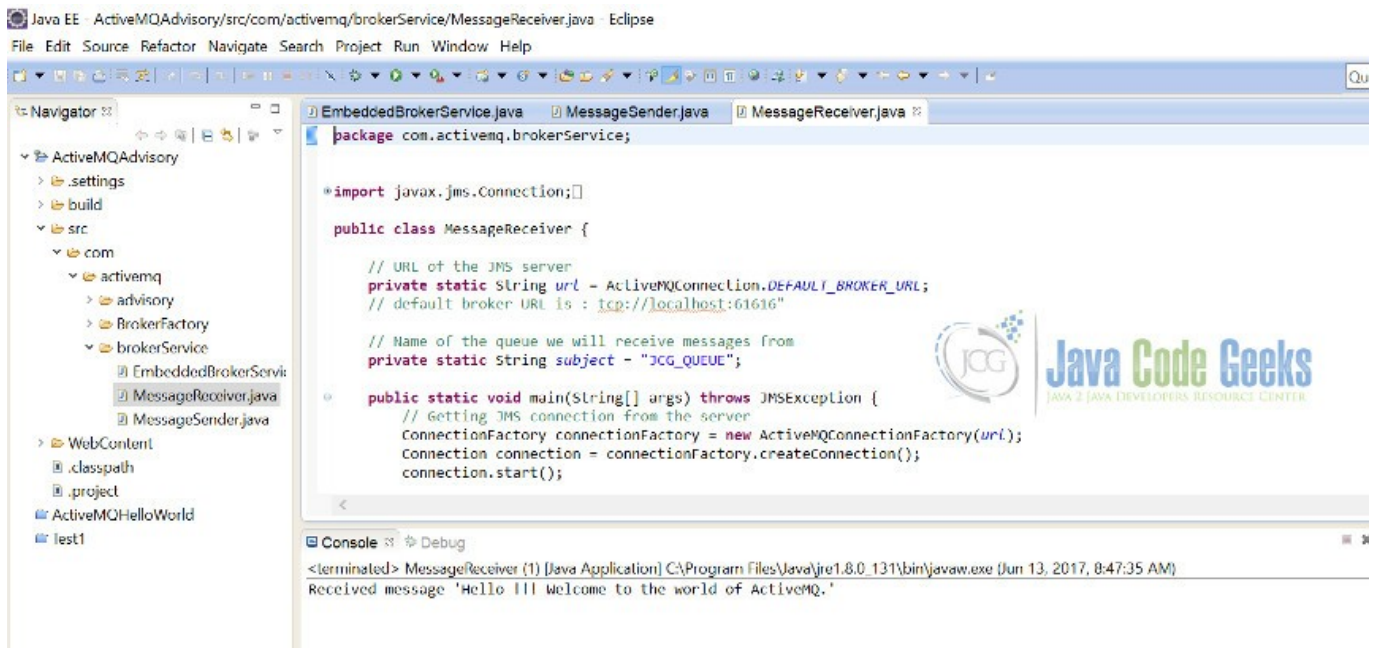


Figure 5.3: MessageReceiver output

5.2.2 Embed a broker in ActiveMQ using Spring 2.0

We can embed the ActiveMQ broker XML inside any regular Spring.xml file, if we are using the new XML Schema-based configuration of Spring. Here is an example of a regular Spring 2.0 XML file which also configures a broker. This allows us to configure JMS artifacts like destinations and connection factories together with the entire broker.

Spring.xml

```
<beans
  xmlns="https://www.springframework.org/schema/beans"
  xmlns:amq="https://activemq.apache.org/schema/core"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↵
    springframework.org/schema/beans/spring-beans-2.0.xsd
  https://activemq.apache.org/schema/core https://activemq.apache.org/schema/core/activemq- ↵
    core.xsd">

  <amq:broker useJmx="false" persistent="false">
  <amq:transportConnectors>
  <amq:transportConnector uri="tcp://localhost:0" />
  </amq:transportConnectors>
  </amq:broker>

  <amq:connectionFactory id="jmsFactory" brokerURL="vm://localhost"/>
</beans>
```

5.2.3 Embed a broker in ActiveMQ using XBean

If we are already using XBean then we can just mix and match our Spring/XBean XML configuration with ActiveMQ's configuration.

Spring.xml


```

<beans
xmlns="https://www.springframework.org/schema/beans"
xmlns:amq="https://activemq.apache.org/schema/core"
xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↵
    springframework.org/schema/beans/spring-beans-2.0.xsd
https://activemq.apache.org/schema/core https://activemq.apache.org/schema/core/activemq- ↵
    core.xsd">

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

<broker useJmx="true" xmlns="https://activemq.apache.org/schema/core">

<persistenceFactory>
<kahaDB directory="${basedir}/target" />
</persistenceFactory>

<transportConnectors>
<transportConnector uri="tcp://localhost:61636" />
</transportConnectors>

</broker>
</beans>

```

5.2.4 Embed a broker in ActiveMQ using BrokerFactory

There is a helper class called `BrokerFactory` which can be used to create a broker via URI for configuration. Instead of explicitly instantiating a broker service, we can use `BrokerFactory` to create a broker. All we need to do is pass the URI configuration.

- Let us now create a `BrokerFactoryExample.java`
- The URI passed is `broker://(tcp://localhost:61616)?brokerName=myJCGBroker`. It will use the broker scheme to instantiate the specific `BrokerFactory`. It uses the `brokerURI` to extract the configuration parameters for the broker service. It internally instantiates a `BrokerService` and then directly configures the pojo model
- Please refer the code snippet below for details

`BrokerFactoryExample.java`

```

package com.activemq.BrokerFactory;

public class BrokerFactoryExample {
    public static void main(String[] args) throws Exception {
        String brokerName = "myJCGBroker";
        String brokerSchemeUrl = "broker://(tcp://localhost:61616)?brokerName="
            + brokerName;
        Utility.createBrokerSendReceiveMessage(brokerSchemeUrl, brokerName);
    }
}

```

- Next we create a Utility class, in order to connect to the broker and create a connection factory first
- If the client is within the same JVM as that of broker then we need to use `vm://brokerName` virtual machine protocol scheme
- Then we just create connection and start it. After that just create a session and send and receive a message

- Please refer the code snippet below for details

Utility.java

```
package com.activemq.BrokerFactory;

import javax.jms.Connection;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.broker.BrokerFactory;
import org.apache.activemq.broker.BrokerRegistry;
import org.apache.activemq.broker.BrokerService;

public class Utility {
    public static void createBrokerSendMessage(String brokerSchemeUrl,
        String brokerName) throws Exception {
        BrokerService brokerService = BrokerFactory
            .createBroker(brokerSchemeUrl);
        startBrokerSendMessage(brokerService);
    }

    public static void startBrokerSendMessage(BrokerService brokerService) ←
        throws Exception {
        brokerService.start();
        String brokerName = brokerService.getBrokerName();
        System.out.println("Broker " + brokerName
            + " started? " + brokerService.isStarted());

        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory ←
            (
                "vm://" + brokerService.getBrokerName() + "?create=false");
        Connection connection = connectionFactory.createConnection();
        connection.start();

        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        try {
            Queue destination = session.createQueue("Q");
            MessageProducer producer = session.createProducer(destination);
            Message message = session.createTextMessage("Hi!");
            System.out.println("Sending Hi!....");
            producer.send(message);
            MessageConsumer consumer = session.createConsumer(destination);
            System.out.println("Message received "
                + ((TextMessage) consumer.receive()).getText());
        } finally {
            session.close();
            connection.close();
            BrokerRegistry.getInstance().lookup(brokerName).stop();
        }
    }
}
```

Output:

We will be running our BrokerFactoryExample class which starts the broker and then use utility class to send and receive a message.

Please follow the steps below:

- In the eclipse Right Click on BrokerFactoryExample.java → Run As→Java Application, to start the broker
- The utility class methods are called to create a connection and exchange a text message
- The same message after being received from the queue gets printed in eclipse output console
- Please refer the image below to check the output

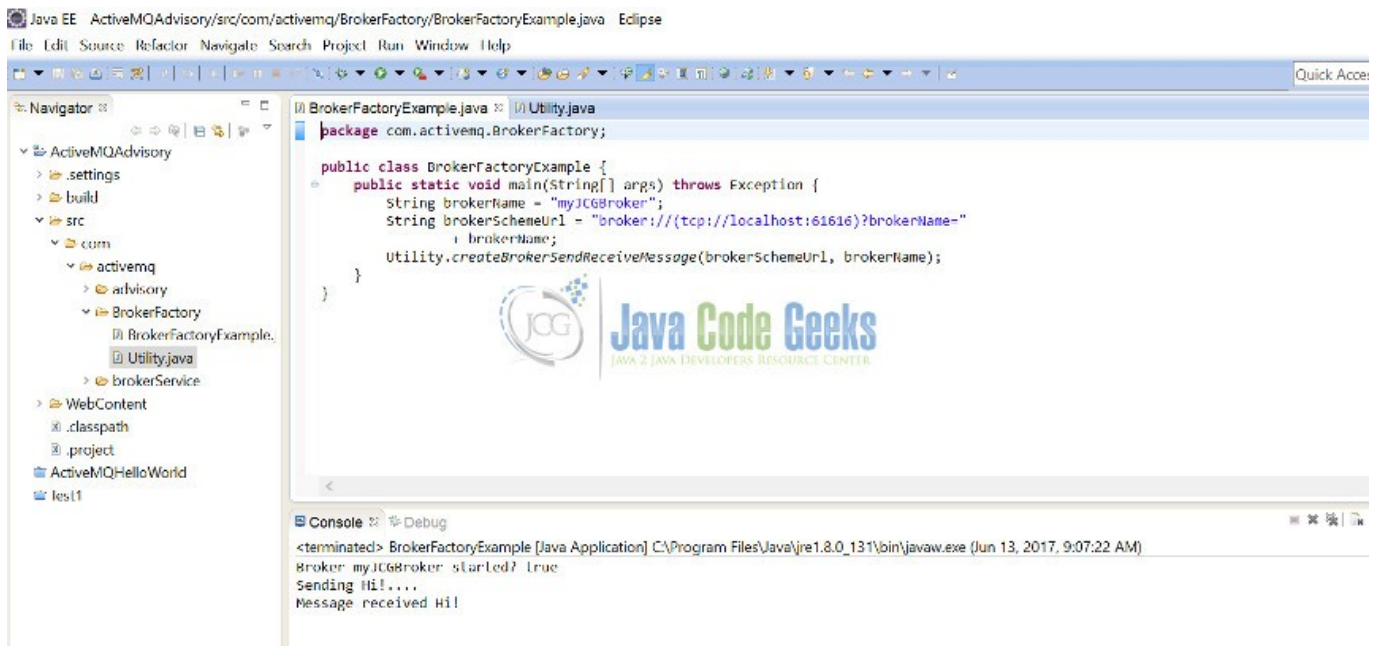


Figure 5.4: BrokerFactory example

5.3 Conclusion

Through this example, we have learned various ways to embed a broker in ActiveMQ depending on if we are using Java, Spring, XBean or using the BrokerFactory. We have also seen the usage of BrokerService API.

Chapter 6

ActiveMQ Load Balancing Example

6.1 Introduction

Apache ActiveMQ (AMQ) is a **message broker** which transfers the message from the sender to the receiver. **Load Balancing** is the process of distributing data across services for better performance.

In this example, we will demonstrate how to build a load-balanced AMQ client application.

6.2 The Component Diagram

In this example, we will demonstrate two forms of load balancing outlined in the diagram:

- A message producer sends messages to multiple AMQ brokers
- Messages in a queue are consumed by multiple competing consumers

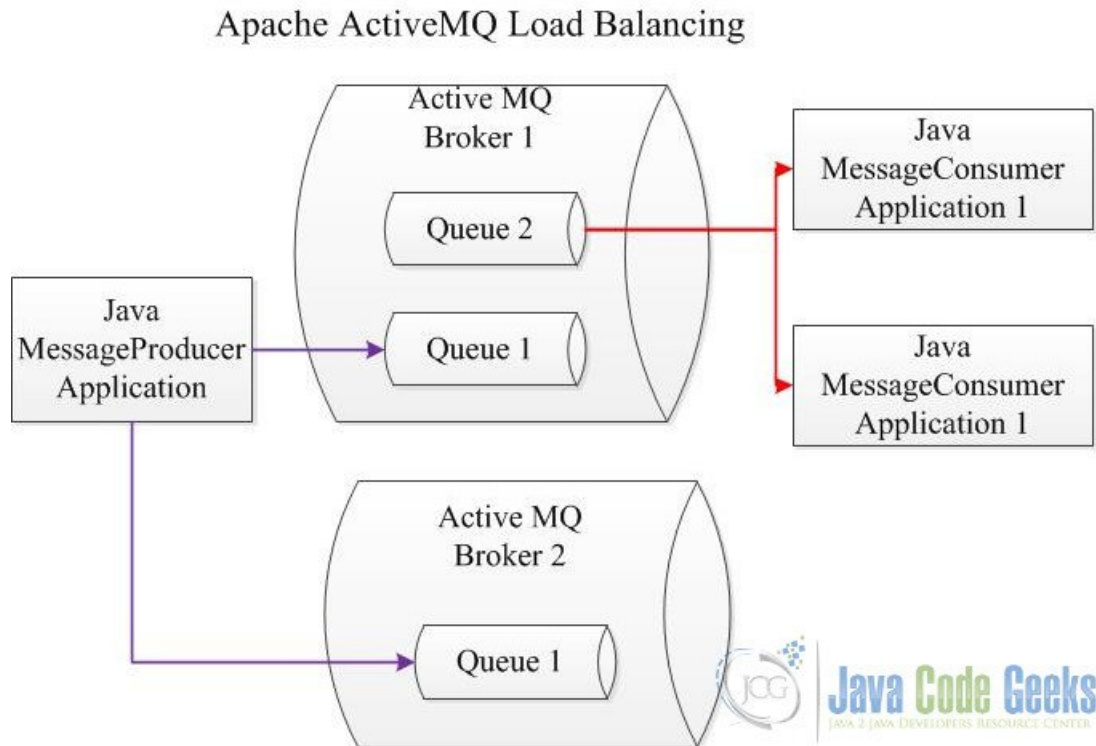


Figure 6.1: AMQ Load Balancing

6.3 Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Apache ActiveMQ 5.8.0 and 5.15.0 (others will do fine)
- Spring JMS 4.1.5.RELEASE (others will do fine)
- Eclipse Neon (Any Java IDE would work)

6.4 Start two ActiveMQ Brokers

6.4.1 Configure ActiveMQ with Non-Default Port

6.4.1.1 Update activemq.xml

Navigate to the `..\apache-activemq-5.8.0\conf` directory. Update the `activemq.xml` file at the `transportConnector` element.

`activemq.xml` `transportConnectors`

```
<transportConnectors>
  <transportConnector name="openwire" uri="tcp://0.0.0.0:61716?maximumConnections ←
    =1000&wireformat.maxFrameSize=104857600"/>
</transportConnectors>
```

6.4.1.2 Update jetty.xml

Go to the `..\apache-activemq-5.8.0\conf` directory. Update the `jetty.xml` file at the bean element `jetty.xml port`

```
<bean id="Connector" class="org.eclipse.jetty.server.nio.SelectChannelConnector">
  <property name="port" value="8761" />
</bean>
```

6.4.2 Start ActiveMQ Brokers

In this example, we will start two AMQ instances:

- Broker 1 - AMQ 5.15.0 at default port 61616/8161
- Broker 2 - AMQ 5.8.0 at port 61716/8761

Go to the `..\apache-activemq-5.x.0\bin` directory. Then click the `activemq.bat` file.

If you can go to `https://localhost:8161/admin/index.jsp`, then the broker 1 is started fine.

You do the same for broker 2 at `https://localhost:8761/admin/index.jsp`.

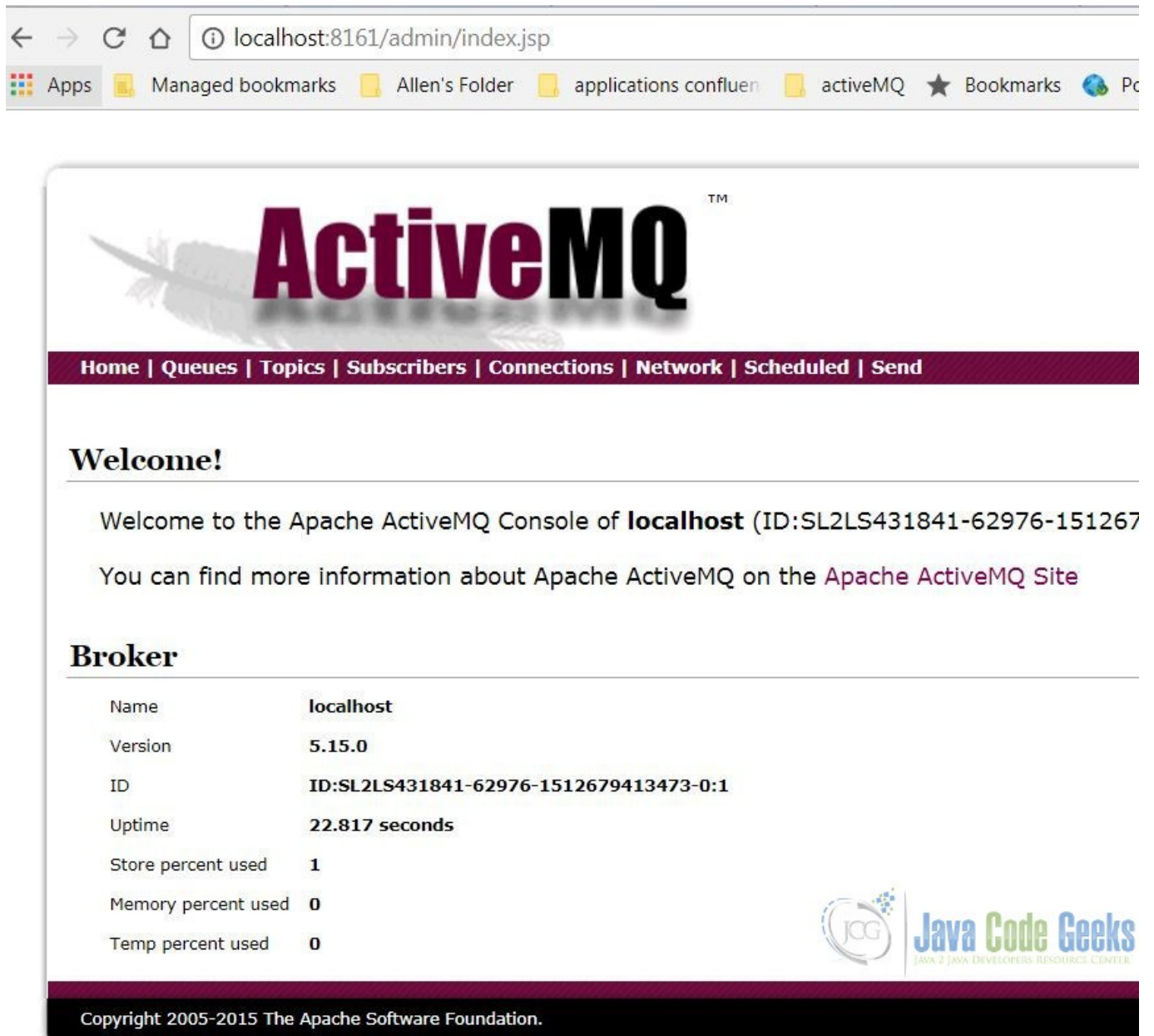


Figure 6.2: AMQ Broker 1 Started

6.5 Producer Load Balancing Example

In this example, we will demonstrate how to build `MessageSender` which uses the Round-robin method to send messages to two AMQ brokers.

6.5.1 Dependency

Add dependency to Maven pom.xml.

pom.xml

```
<dependencies>
```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>4.1.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>4.1.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jms</artifactId>
  <version>4.1.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-all</artifactId>
  <version>5.15.0</version>
</dependency>
</dependencies>
```

6.5.2 Constants

There are five constants value used in this example.

DemoConstants

```
package jcg.demo.util;

import java.util.Random;

/**
 * The constant data used in this Demo
 * @author Mary.Zheng
 *
 */
public final class DemoConstants{

    public static final int MESSAGE_SIZE = 100;
    public static final String PRODUCER_DESTINATION = "test.queue.lb.producer";
    public static final String CONSUMER_DESTINATION = "test.queue.lb.consumer";

    public static String BROKER_1_URI = "tcp://localhost:61616";
    public static String BROKER_2_URI = "tcp://localhost:61716";

    public static String buildDummyMessage() {
        Random rand = new Random();
        int value = rand.nextInt(MESSAGE_SIZE);
        return "dummy message " + value;
    }
}
```

6.5.3 Spring configuration

Add the JMS Spring configuration.

JmsConfig


```
package jcg.demo.spring.config;

import javax.jms.ConnectionFactory;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.annotation.EnableJms;
import org.springframework.jms.connection.CachingConnectionFactory;
import org.springframework.jms.core.JmsTemplate;

import jcg.demo.spring.jms.component.JmsExceptionListener;
import jcg.demo.util.DemoConstants;

@Configuration
@EnableJms
@ComponentScan(basePackages = "jcg.demo.spring.jms.component")
public class JmsConfig {

    @Bean
    @Autowired
    public ConnectionFactory jmsConnectionFactory(JmsExceptionListener ←
        jmsExceptionListener) {
        return createJmsConnectionFactory(DemoConstants.BROKER_1_URI, ←
            jmsExceptionListener);
    }

    @Bean
    @Autowired
    public ConnectionFactory jmsConnectionFactory_2(JmsExceptionListener ←
        jmsExceptionListener) {
        return createJmsConnectionFactory(DemoConstants.BROKER_2_URI, ←
            jmsExceptionListener);
    }

    private ConnectionFactory createJmsConnectionFactory(String brokerURI, ←
        JmsExceptionListener jmsExceptionListener) {
        ActiveMQConnectionFactory activeMQConnectionFactory = new ←
            ActiveMQConnectionFactory(brokerURI);
        activeMQConnectionFactory.setExceptionListener(jmsExceptionListener);

        CachingConnectionFactory pooledConnection = new CachingConnectionFactory( ←
            activeMQConnectionFactory);
        return pooledConnection;
    }

    @Bean(name = "jmsQueueTemplate_1")
    @Autowired
    public JmsTemplate createJmsQueueTemplate(ConnectionFactory jmsConnectionFactory) {
        return new JmsTemplate(jmsConnectionFactory);
    }

    @Bean(name = "jmsQueueTemplate_2")
    @Autowired
    public JmsTemplate createJmsQueueTemplate_2(ConnectionFactory ←
        jmsConnectionFactory_2) {
        return new JmsTemplate(jmsConnectionFactory_2);
    }
}
```

- line 25: Create connection factory to broker 1
- line 31: Create connection factory to broker 2
- line 42: Create `JmsTemplate` to broker 1
- line 48: Create `JmsTemplate` to broker 2

6.5.4 MessageSender

Create `MessageSender` Spring component to send messages.

`MessageSender`

```
package jcg.demo.spring.jms.component;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Component;

/**
 * This is Spring component which finds the load-balanced JmsTemplate via
 * Round-Robin from the list of available JmsQueueTemplates to send the message
 *
 * @author Mary.Zheng
 *
 */
@Component
public class MessageSender {
    @Autowired
    private List jmsQueueTemplates = new ArrayList();

    private AtomicInteger current = new AtomicInteger(0);

    private JmsTemplate findJmsTemplate_LB() {
        int cur = current.getAndIncrement();
        int index = cur % jmsQueueTemplates.size();
        System.out.println("\t\tFind Load balanced JmsTemplate[ " + index + " ]");

        return jmsQueueTemplates.get(index);
    }

    public void postToQueue(final String queueName, final String message) {
        System.out.println("MessageSender postToQueue started");
        this.findJmsTemplate_LB().send(queueName, new MessageCreator() {

            @Override
            public Message createMessage(Session session) throws JMSEException {
                return session.createTextMessage(message);
            }
        });
    }
}
```

- line 25: Spring dependency injection will add both broker's `JmsTemplate` to `jmsQueueTemplates`
- line 30-36: Use Round-robin logic to find the `JmsTemplate`
- line 40: Send the message with load-balanced `JmsTemplate`

6.5.5 MessageProducerApp

Create MessageProducer application.

MessageProducerApp

```
package jcg.demo.activemqqlb.producer;

import java.util.Scanner;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Configuration;

import jcg.demo.spring.config.JmsConfig;
import jcg.demo.spring.jms.component.MessageSender;
import jcg.demo.util.DemoConstants;

@Configuration
public class MessageProducerApp {
    public static void main(String[] args) throws Exception {

        try (AnnotationConfigApplicationContext context = new ←
            AnnotationConfigApplicationContext(JmsConfig.class)) {
            context.register(MessageProducerApp.class);
            String queueName = readDestination();

            MessageSender springJmsProducer = (MessageSender) context.getBean(" ←
                messageSender");

            for (int i = 0; i < DemoConstants.MESSAGE_SIZE; i++) {
                springJmsProducer.postToQueue(queueName, DemoConstants. ←
                    buildDummyMessage());
            }
        }

        private static String readDestination() {
            System.out.println("Enter Destination: P - Producer, C - Consumer : ");

            try (Scanner scanIn = new Scanner(System.in)) {
                String inputString = scanIn.nextLine();
                scanIn.close();
                if (inputString.equalsIgnoreCase("P")) {
                    return DemoConstants.PRODUCER_DESTINATION;
                }
                return DemoConstants.CONSUMER_DESTINATION;
            }
        }
    }
}
```

- line 16: Start Spring context from `JmsConfig`
- line 20: Get `messageSender` Spring bean

6.5.6 Execute MessageProducerApp

Below is the application output when you input P on the prompt. Make sure both brokers are running.

Execution Output

```
Enter Destination: P - Producer, C - Consumer :
P
MessageSender postToQueue started
Find Load balanced JmsTemplate[ 0 ]
MessageSender postToQueue started
Find Load balanced JmsTemplate[ 1 ]
MessageSender postToQueue started
Find Load balanced JmsTemplate[ 0 ]
MessageSender postToQueue started
Find Load balanced JmsTemplate[ 1 ]
.....
```

As you see here, two `JmsTemplates` take turns to send a total of 100 messages to their connected broker. Go to <https://localhost:8161/admin/queues.jsp> for broker 1 and <https://localhost:8761/admin/queues.jsp> for broker 2. You should see that each broker has 50 pending messages at `test.queue.lb.producer`.

6.6 Consumer Load Balancing Example

In this example, we will demonstrate how to build the `MessageConsumerApp` which consumes the messages from a queue. We also show how to run two of them concurrently.

6.6.1 MessageConsumerWithPrefetch

AMQ brokers set default prefetch size 1000, so we have to set the prefetch size to 1 to allow two consumers consume messages concurrently.

MessageConsumerWithPrefetch

```
package jcg.demo.activemq.lb.consumer;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

/**
 * A simple message consumer which consumes the message from ActiveMQ Broker
 * with pre-fetch size set to 1 instead of default 1000.
 *
 * @author Mary.Zheng
 *
 */
public class MessageConsumerWithPrefetch implements MessageListener {

    private static final String JMS_PREFETCH_POLICY_ALL_1 = "?jms.prefetchPolicy.all=1" ←
    ;
    private String activeMqBrokerUri;
```

```

private String username;
private String password;
private String destinationName;

public MessageConsumerWithPrefetch(String activeMqBrokerUri, String username, ↵
String password) {
    super();
    this.activeMqBrokerUri = activeMqBrokerUri + JMS_PREFETCH_POLICY_ALL_1;
    this.username = username;
    this.password = password;
}

public void run() throws JMSEException {
    ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(username, ↵
password, activeMqBrokerUri);
    Connection connection = factory.createConnection();
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE) ↵
;
    setComsumer(session);
    connection.start();

    System.out.println(String.format("MessageConsumerWithPrefetch Waiting for ↵
messages at %s from %s",
destinationName, this.activeMqBrokerUri));
}

private void setComsumer(Session session) throws JMSEException {
    Destination destination = session.createQueue(destinationName);
    MessageConsumer consumer = session.createConsumer(destination);
    consumer.setMessageListener(this);
}

@Override
public void onMessage(Message message) {
    String msg;
    try {
        msg = String.format("MessageConsumerWithPrefetch Received message [ ↵
%s ]",
((TextMessage) message).getText());
        Thread.sleep(10000); // sleep for 10 seconds
        System.out.println(msg);
    } catch (JMSEException | InterruptedException e) {
        e.printStackTrace();
    }
}

public String getDestinationName() {
    return destinationName;
}

public void setDestinationName(String destinationName) {
    this.destinationName = destinationName;
}
}

```

- line 23, 31 : Set the AMQ prefetchPolicy

6.6.2 MessageConsumerApp

Create MessageConsumerApp which consumes from the consumer queue based on the selected broker.

MessageConsumerApp

```
package jcg.demo.activemq_lb.consumer;

import java.util.Scanner;

import javax.jms.JMSEException;

import jcg.demo.util.DemoConstants;

public class MessageConsumerApp {

    public static void main(String[] args) {
        String brokerUri = readBrokerInstance();

        consume_queue_with_prefetchsize(brokerUri);
    }

    private static void consume_queue_with_prefetchsize(String brokerUri) {
        MessageConsumerWithPrefetch queueMsgListener = new ←
            MessageConsumerWithPrefetch(brokerUri, "admin", "admin");
        queueMsgListener.setDestinationName(DemoConstants.CONSUMER_DESTINATION);

        try {
            queueMsgListener.run();
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }

    private static String readBrokerInstance() {
        System.out.println("MessageConsumerApp listens at Broker Instance ( 1 or 2 ←
            ): ");

        try (Scanner scanIn = new Scanner(System.in)) {
            String inputString = scanIn.nextLine();
            scanIn.close();
            if (inputString.equalsIgnoreCase("1")) {
                return DemoConstants.BROKER_1_URI;
            }
            return DemoConstants.BROKER_2_URI;
        }
    }
}
```

6.6.3 Execute MessageConsumerApp in Eclipse

Starts the MessageConsumerApp via Eclipse.

MessageConsumerApp Output

```
MessageConsumerApp listens at Broker Instance ( 1 or 2 ):
1
MessageConsumerWithPrefetch Waiting for messages at test.queue.lb.consumer from tcp:// ←
localhost:61616?jms.prefetchPolicy.all=1
```

6.6.4 Execute MessageConsumerApp via Jar command

First, **export** the MessageConsumerApp as a jar: `activemq-lb.jar`. Open the command prompt and enter the command `java -jar activemq-lb.jar`.

MessageConsumerApp Output

```
C:\JDK8_CTLSS\Java Code Geek Examples>java -jar activemq-lb.jar
MessageConsumerApp listens at Broker Instance ( 1 or 2 ):
1
MessageConsumerWithPrefetch Waiting for messages at test.queue.lb.consumer from
tcp://localhost:61616?jms.prefetchPolicy.all=1
```

6.6.5 Summary

There are two consumer applications listening at `test.queue.lb.consumer` after the steps 6.6.3 and 6.6.4, Monitoring both output while executing the `MessageProducerApp` built at step 6.5.5 to send 100 messages to `test.queue.lb.consumer`. You should see both consumers are receiving the messages. The screenshot below shows both consumers consumed 25 messages from `test.queue.lb.consumer` after all messages are processed.

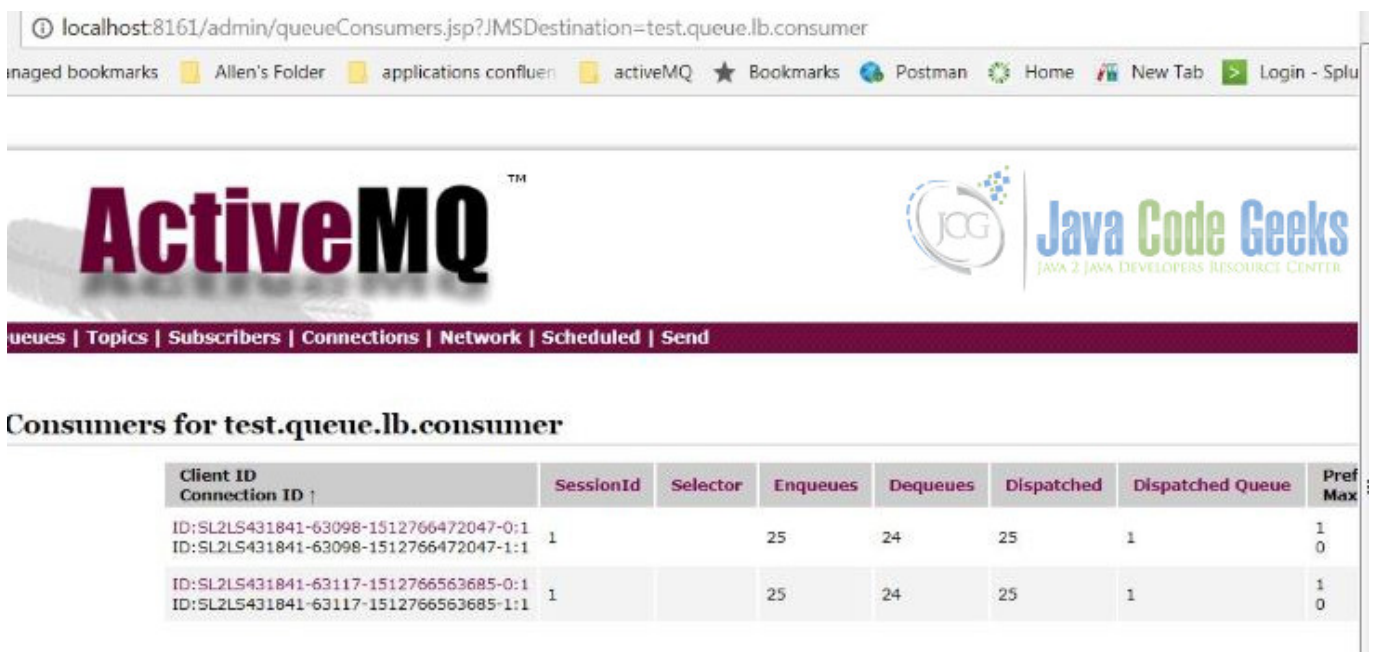


Figure 6.3: AMQ Consumer Load Balancing

6.6.6 Things to consider

The AMQ message is dispatched based on the First-In, First-Out(FIFO) algorithm. If the messages must be processed based on the order entered, then running the consumer concurrently must be planned accordingly to avoid an error. Please check out [ActiveMQ tutorial](#) for detail.

6.7 Conclusion

In this example, we built two Java AMQ client applications:

- `MessageProducerApp` sends the message to two AMQ brokers via Round-robin algorithm to reduce the data load at each AMQ Broker
- Two `MessageConsumerApps` consume the messages from the same queue to reduce the data load at the AMQ queue

6.8 Download the Source Code

This example built two Java AMQ client applications (producer and consumer) to achieve the load balancing requirement. You can download the full source code of this example here: [Apache ActiveMQ Load Balancing Example](#)

Chapter 7

ActiveMQ Failover Example

Apache ActiveMQ is an open source messaging server written in Java, which implements JMS 1.1 specifications. In this example, I will demonstrate how to configure a group of AMQ brokers to make the system fault-tolerant.

7.1 Introduction

Apache ActiveMQ (AMQ) is a **message broker** which transfers the messages from the sender to the receiver.

Failover is a procedure by which a system automatically transfers control to a duplicate system when it detects a fault or failure.

AMQ failover transport protocol enables an application automatically reconnect to a broker when a failure is detected while establishing a connection.

In this example, I will demonstrate:

- how to create eight AMQ broker instances
- how to configure AMQ broker instances with four network typologies
- how to build a Java client application which still functions when one of the AMQ broker fails

7.2 ActiveMQ Server Installation

Follow these **instructions** to install an AMQ server. Then use AMQ admin command: `activemq-admin create ${brokerName}` to create a server instance.

Create eight AMQ server instances:

```
activemq-admin create ..\standalone\broker1
activemq-admin create ..\standalone\broker2
activemq-admin create ..\cluster\broker-1
activemq-admin create ..\cluster\broker-2
activemq-admin create ..\cluster\broker-3
activemq-admin create ..\cluster\dynamic-broker1
activemq-admin create ..\cluster\dynamic-broker2
activemq-admin create ..\cluster\dynamic-broker3
```

7.3 ActiveMQ Server Configuration

AMQ `activemq-admin` command copies the AMQ server to the user defined location. Need to configure the default value with the steps below:

- Navigate to the AMQ instance folder. Ex: `standalone\broker1`

- Edit the `activemq.xml` at `transportConnector` and `networkConnector`
- Edit the `jetty.xml` with a different web port number
- Edit the Windows batch file at `ACTIVEMQ_CONF` and `ACTIVEMQ_DATA`

7.3.1 Two Standalone ActiveMQ Brokers

Use the steps above, configure two standalone AMQ brokers:

Broker Name	Home Path	Openwire Port	Web Port	Data Path
broker1	..\standalone\broker1	61616	8161	broker1\data
broker2	..\standalone\broker2	61716	7161	broker2\data

Figure 7.1: Standalone AMQ brokers

```

C:\MaryZheng\tools\apache-activemq-5.15.0\standalone>tree
Folder PATH listing for volume OSDisk
Volume serial number is 3A10-C6D4
C:
├── broker1
│   ├── bin
│   ├── conf
│   └── data
│       ├── kahadb
│       └── tmp
├── broker2
│   ├── bin
│   ├── conf
│   └── data
│       ├── kahadb
│       └── tmp
└── ...
C:\MaryZheng\tools\apache-activemq-5.15.0\standalone>

```

Figure 7.2: Two Standalone AMQ Brokers

Here is an example of `broker1.bat` file.

`broker1.bat`

```

@echo off

set ACTIVEMQ_HOME="C:/MaryZheng/tools/apache-activemq-5.15.0"
set ACTIVEMQ_BASE="C:/MaryZheng/tools/apache-activemq-5.15.0/standalone/broker1"
set ACTIVEMQ_CONF=%ACTIVEMQ_BASE%/conf
set ACTIVEMQ_DATA=%ACTIVEMQ_BASE%/data

set PARAM=%1
:getParam

```

```

shift
if "%1"==" " goto end
set PARAM=%PARAM% %1
goto getParam
:end

%ACTIVEMQ_HOME%/bin/activemq %PARAM%

```

Start broker1 with the command: `broker1.bat start`.

broker1 server log

```

C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\bin>broker1.bat start
Java Runtime: Oracle Corporation 1.8.0_31 C:\MaryZheng\tools\java\jdk1.8.0_31\jre
Heap sizes: current=1005056k free=984084k max=1005056k
JVM args: -Dcom.sun.management.jmxremote -Xms1G -Xmx1G -Djava.util.logging.config.file=
logging.properties -Djava.security.auth.login.config=C:/MaryZheng/too
ls/apache-activemq-5.15.0/standalone/broker1/conf\login.config -Dactivemq.classpath=C:/
MaryZheng/tools/apache-activemq-5.15.0/standalone/broker1/conf;C:/MaryZhe
ng/tools/apache-activemq-5.15.0/standalone/broker1/conf;C:/MaryZheng/tools/apache-activemq
-5.15.0/conf; -Dactivemq.home=C:/MaryZheng/tools/apache-activemq-5.15.
0 -Dactivemq.base=C:/MaryZheng/tools/apache-activemq-5.15.0/standalone/broker1 -Dactivemq.
conf=C:/MaryZheng/tools/apache-activemq-5.15.0/standalone/broker1/conf
-Dactivemq.data=C:/MaryZheng/tools/apache-activemq-5.15.0/standalone/broker1/data -Djava.
io.tmpdir=C:/MaryZheng/tools/apache-activemq-5.15.0/standalone/broker1
/data\tmp
Extensions classpath:
[C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\lib,C:\MaryZheng\
tools\apache-activemq-5.15.0\lib,C:\MaryZheng\tools\apache-activemq-5.15.0\stan
dalone\broker1\lib\camel,C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\
broker1\lib\optional,C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1
\
lib\web,C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\lib\extra,C:\
MaryZheng\tools\apache-activemq-5.15.0\lib\camel,C:\MaryZheng\tools\apache-act
ivemq-5.15.0\lib\optional,C:\MaryZheng\tools\apache-activemq-5.15.0\lib\web,C:\
MaryZheng\tools\apache-activemq-5.15.0\lib\extra]
ACTIVEMQ_HOME: C:\MaryZheng\tools\apache-activemq-5.15.0
ACTIVEMQ_BASE: C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1
ACTIVEMQ_CONF: C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\conf
ACTIVEMQ_DATA: C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\data
Loading message broker from: xbean:activemq.xml
INFO | Refreshing org.apache.activemq.xbean.XBeanBrokerFactory$1@1753acfe: startup date [
Sat Dec 16 07:05:53 CST 2017]; root of context hierarchy
INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\MaryZheng\tools\apache-
activemq-5.15.0\standalone\broker1\data\kahadb]
INFO | KahaDB is version 6
INFO | Recovering from the journal @1:65574
INFO | Recovery replayed 15 operations from the journal in 0.032 seconds.
INFO | PListStore:[C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\data
\broker1\tmp_storage] started
INFO | Apache ActiveMQ 5.15.0 (broker1, ID:SL2LS431841-50062-1513429555523-0:1) is
starting
INFO | Listening for connections at: tcp://SL2LS431841:61616?maximumConnections=1000&
wireFormat.maxFrameSize=104857600
INFO | Connector openwire started
INFO | Apache ActiveMQ 5.15.0 (broker1, ID:SL2LS431841-50062-1513429555523-0:1) started
INFO | For help or more information please see: https://activemq.apache.org
WARN | Store limit is 102400 mb (current store usage is 0 mb). The data directory: C:\
MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\data\kahadb only has
1146 mb of usable space. - resetting to maximum available disk space: 1146 mb
WARN | Temporary Store limit is 51200 mb (current store usage is 0 mb). The data directory
: C:\MaryZheng\tools\apache-activemq-5.15.0\standalone\broker1\data only has 1146
mb of usable space. - resetting to maximum available disk space: 1146 mb

```

```
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | ActiveMQ WebConsole available at https://0.0.0.0:8161/
INFO | ActiveMQ Jolokia REST API available at https://0.0.0.0:8161/api/jolokia/
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | jolokia-agent: Using policy access restrictor classpath:/jolokia-access.xml
```

- Line 1: start the broker1
- Line 2: Java Runtime is JDK8
- Line 15-18: important setting values
- Line 27: broker1 openwire port
- Line 29: broker1 starts
- Line 34: broker1 web console starts

Repeat the steps to configure other brokers.

Note: Pay special attention to the highlighted lines. Verify the AMQ via web console.

7.3.2 Master/Slave ActiveMQ Brokers

In a Master/Slave topology, the Master provides services to the client, the Slave is at standby mode and gets promoted when the Master fails. There are three kinds of Master/Slave configurations:

- [Shared File System Master Slave](#)
- [JDBC Master Slave](#)
- [Replicated LevelDB Store](#)

Configure Master/Slave brokers with "Shared File System":

Broker Name	Home Path	Openwire Port	Web Port	Data Path
broker-1	..\cluster\broker-1	61816	8861	..\data
broker-2	..\cluster\broker-2	61826	8862	..\data

Figure 7.3: Configure Master/Slave brokers with Shared File System

Note: The configuration steps for Mater/Slave are same as the standalone server. The difference is that Master and Slave must share the same data.

Slave server log

```
C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\broker-2\bin>broker-2.bat start
Java Runtime: Oracle Corporation 1.8.0_31 C:\MaryZheng\tools\java\jdk1.8.0_31\jre
Heap sizes: current=1005056k free=984084k max=1005056k
JVM args: -Dcom.sun.management.jmxremote -Xms1G -Xmx1G -Djava.util.logging.config.file=
logging.properties -Djava.security.auth.login.config=C:/MaryZheng/tools/apache-
activemq-5.15.0/cluster/broker-2/conf\login.config -Dactivemq.classpath=C:/
MaryZheng/tools/apache-activemq-5.15.0/cluster/broker-2/conf;C:/MaryZheng/tools/
apache-activemq-5.15.0/cluster/broker-2/conf;C:/MaryZheng/tools/apache-activemq
-5.15.0/conf; -Dactivemq.home=C:/MaryZheng/tools/apache-activemq-5.15.0 -Dac
```

```
tivemq.base=C:/MaryZheng/tools/apache-activemq-5.15.0/cluster/broker-2 -Dactivemq.conf=C:/ ←
MaryZheng/tools/apache-activemq-5.15.0/cluster/broker-2/conf -Dactivemq.data=C:/ ←
MaryZheng/tools/apache-activemq-5.15.0/data -Djava.io.tmpdir=C:/MaryZheng/tools/apache- ←
activemq-5.15.0/data\tmp
Extensions classpath:
[C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\broker-2\lib,C:\MaryZheng\ ←
tools\apache-activemq-5.15.0\lib,C:\MaryZheng\tools\apache-activemq-5.15.0\ ←
cluste
r\broker-2\lib\camel,C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\broker-2\ ←
lib\optional,C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\broker-2\lib\web,
C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\broker-2\lib\extra,C:\MaryZheng\ ←
tools\apache-activemq-5.15.0\lib\camel,C:\MaryZheng\tools\apache-activemq-5.15
.0\lib\optional,C:\MaryZheng\tools\apache-activemq-5.15.0\lib\web,C:\MaryZheng\ ←
tools\apache-activemq-5.15.0\lib\extra]
ACTIVEMQ_HOME: C:\MaryZheng\tools\apache-activemq-5.15.0
ACTIVEMQ_BASE: C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\broker-2
ACTIVEMQ_CONF: C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\broker-2\conf
ACTIVEMQ_DATA: C:\MaryZheng\tools\apache-activemq-5.15.0\data
Loading message broker from: xbean:activemq.xml
INFO | Refreshing org.apache.activemq.xbean.XBeanBrokerFactory$1@1753acfe: startup date [ ←
Sat Dec 16 08:05:25 CST 2017]; root of context hierarchy
INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\MaryZheng\tools\apache- ←
activemq-5.15.0\data\kahadb]
INFO | Database C:\MaryZheng\tools\apache-activemq-5.15.0\data\kahadb\lock is locked ←
by another server. This broker is now in slave mode waiting a lock to be acquired
```

- Line 1: start the broker
- Line 14: display ACTIVEMQ_DATA location
- Line 18: indicate that the broker is the Slave

Stop the Master and watch the Slave get promoted to the Master.

Slave broker promotes to Master log

```
INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\MaryZheng\tools\apache- ←
activemq-5.15.0\data\kahadb]
INFO | Database C:\MaryZheng\tools\apache-activemq-5.15.0\data\kahadb\lock is locked ←
by another server. This broker is now in slave mode waiting a lock to be acquired
INFO | KahaDB is version 6
INFO | Recovering from the journal @1:31536026
INFO | Recovery replayed 97 operations from the journal in 0.042 seconds.
INFO | PListStore:[C:\MaryZheng\tools\apache-activemq-5.15.0\data\broker-2\ ←
tmp_storage] started
INFO | Apache ActiveMQ 5.15.0 (broker-2, ID:SL2LS431841-50406-1513433598677-0:1) is ←
starting
INFO | Listening for connections at: tcp://SL2LS431841:61826?maximumConnections=1000& ←
wireFormat.maxFrameSize=104857600
INFO | Connector openwire started
INFO | Apache ActiveMQ 5.15.0 (broker-2, ID:SL2LS431841-50406-1513433598677-0:1) started
INFO | For help or more information please see: https://activemq.apache.org
WARN | Store limit is 102400 mb (current store usage is 31 mb). The data directory: C:\ ←
MaryZheng\tools\apache-activemq-5.15.0\data\kahadb only has 1173 mb of usable space ←
. - resetting to maximum available disk space: 1173 mb
WARN | Temporary Store limit is 51200 mb (current store usage is 0 mb). The data directory ←
: C:\MaryZheng\tools\apache-activemq-5.15.0\data only has 1141 mb of usable space. - ←
resetting to maximum available disk space: 1141 mb
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | ActiveMQ WebConsole available at https://0.0.0.0:8961/
INFO | ActiveMQ Jolokia REST API available at https://0.0.0.0:8961/api/jolokia/
```

```
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | jolokia-agent: Using policy access restrictor classpath:/jolokia-access.xml
```

- Line 2: slave indication
- Line 3-7: promote the Slave to the Master

7.3.3 Network of Brokers

AMQ provides a `networkConnector` to connect two brokers together and three options at the `transportConnector`: `updateClusterClients`, `rebalanceClusterClients` and `updateClusterClientsOnRemove`.

7.3.3.1 Static Network of Brokers

In a static network of brokers, the `networkConnector` connects the broker to a list of brokers.

Configure a network of three brokers:

Broker Name	Home Path	Openwire Port	Web Port	Data Path
broker-1	..\clusterbroker-1	61816	8162	..\data
broker-2	..\clusterbroker-2	61826	8961	..\data
broker-3	..\clusterbroker-3	61516	5161	\\broker-3\data

Figure 7.4: Network of three brokers

The static network of brokers' `activemq.xml`.

`activemq.xml`

```
<?xml version="1.0" encoding="UTF-8"?><beans xmlns="https://www.springframework.org/schema/beans" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd https://activemq.apache.org/schema/core https://activemq.apache.org/schema/core/activemq-core.xsd">

  <!-- Allows us to use system properties as variables in this configuration file -->
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      <value>file:${activemq.conf}/credentials.properties</value>
    </property>
  </bean>

  <!-- Allows accessing the server log -->
  <bean class="io.fabric8.insight.log.log4j.Log4jLogQuery" destroy-method="stop" id="logQuery" init-method="start" lazy-init="false" scope="singleton">
  </bean>

  <!--
  The <broker> element is used to configure the ActiveMQ broker.
  -->
  <broker xmlns="https://activemq.apache.org/schema/core" brokerName="broker-3" dataDirectory="${activemq.data}">
```

```

<destinationInterceptors>
  <virtualDestinationInterceptor>
    <virtualDestinations>
      <virtualTopic name="VirtualTopic.>" prefix="Consumer.*" ←
        selectorAware="false"/>
      <virtualTopic name="JCG.>" prefix="VTC.*" selectorAware="true"/>
    </virtualDestinations>
  </virtualDestinationInterceptor>
</destinationInterceptors>

<destinationPolicy>
  <policyMap>
    <policyEntries>
      <policyEntry topic="">
        <!-- The constantPendingMessageLimitStrategy is used to prevent
              slow topic consumers to block producers and affect other consumers
              by limiting the number of messages that are retained
              For more information, see:
              https://activemq.apache.org/slow-consumer-handling.html

              -->
        <pendingMessageLimitStrategy>
          <constantPendingMessageLimitStrategy limit="1000"/>
        </pendingMessageLimitStrategy>
      </policyEntry>
    </policyEntries>
  </policyMap>
</destinationPolicy>

<!--
  The managementContext is used to configure how ActiveMQ is exposed in
  JMX. By default, ActiveMQ uses the MBean server that is started by
  the JVM. For more information, see:
  https://activemq.apache.org/jmx.html
-->
<managementContext>
  <managementContext createConnector="false"/>
</managementContext>

<!--
  Configure message persistence for the broker. The default persistence
  mechanism is the KahaDB store (identified by the kahaDB tag).
  For more information, see:
  https://activemq.apache.org/persistence.html
-->
<persistenceAdapter>
  <kahaDB directory="${activemq.data}/kahadb"/>
</persistenceAdapter>

<!--
  The systemUsage controls the maximum amount of space the broker will
  use before disabling caching and/or slowing down producers. For more ←
  information, see:
  https://activemq.apache.org/producer-flow-control.html
-->
<systemUsage>
  <systemUsage>

```

```

        <memoryUsage>
            <memoryUsage percentOfJvmHeap="70"/>
        </memoryUsage>
        <storeUsage>
            <storeUsage limit="100 gb"/>
        </storeUsage>
        <tempUsage>
            <tempUsage limit="50 gb"/>
        </tempUsage>
    </systemUsage>
</systemUsage>

<networkConnectors>
    <networkConnector name="amq3-nc"
        uri="static:(failover:(tcp://0.0.0.0:61816,tcp://0.0.0.0:61826))"
        dynamicOnly="true"
        networkTTL="3"
        duplex="true"/>
</networkConnectors>

    <!--
        The transport connectors expose ActiveMQ over a given protocol to
        clients and other brokers. For more information, see:

        https://activemq.apache.org/configuring-transport.html
    -->
    <transportConnectors>
        <!-- DOS protection, limit concurrent connections to 1000 and frame size to 100 ←
            MB -->
        <transportConnector name="openwire" rebalanceClusterClients="true" ←
            updateClusterClients="true" updateClusterClientsOnRemove="true" uri="tcp: ←
                //0.0.0.0:61516?maximumConnections=1000&wireFormat.maxFrameSize=104857600"/>
    </transportConnectors>

    <!-- destroy the spring context on shutdown to stop jetty -->
    <shutdownHooks>
        <bean xmlns="https://www.springframework.org/schema/beans" class="org.apache. ←
            activemq.hooks.SpringContextHook"/>
    </shutdownHooks>

</broker>

    <!--
        Enable web consoles, REST and Ajax APIs and demos
        The web consoles requires by default login, you can disable this in the jetty.xml ←
            file

        Take a look at ${ACTIVEMQ_HOME}/conf/jetty.xml for more details
    -->
    <import resource="jetty.xml"/>

</beans>

```

- Line 91- 97: configure as static network of brokers
- Line 107: enable updateClusterClients etc options

Keep the Master/Slave server running and start Broker 3. The Master broker server log shows the it has connected to Broker 3.
master log


```
INFO | Connector vm://broker-2 started
INFO | Started responder end of duplex bridge amq3-nc@ID:SL2LS431841 ←
-64674-1513436259188-0:1
INFO | Network connection between vm://broker-2#0 and tcp:///192.168.1.109:64676@61826 ( ←
broker-3) has been established.
```

Line 3: the network bridge is established between the broker 3 and 2

Note: Verify the AMQ via web console, you should see the network connector detail under Connection tab.

7.3.3.2 Dynamic Network of Brokers

Dynamic network of brokers auto-detects the broker within the network. Configure three brokers:

Broker Name	Home Path	Openwire Port	Web Port	Data Path
broker-1	..\cluster\dynamic-broker1	61626	8163	..\dynamic-broker1\data
broker-2	..\cluster\dynamic-broker2	61636	8164	..\dynamic-broker2\data
broker-3	..\cluster\dynamic-broker3	61646	8165	..\dynamic-broker3\data

Figure 7.5: Configure three brokers

The image below shows six brokers under the cluster directory after steps 7.3.2 and 7.3.3.

```

C:\MaryZheng\tools\apache-activemq-5.15.0\cluster>tree
Folder PATH listing for volume OSDisk
Volume serial number is 3A10-C6D4
C:.\
├── broker-1
│   ├── bin
│   └── conf
├── broker-2
│   ├── bin
│   └── conf
├── broker-3
│   ├── bin
│   ├── conf
│   └── data
│       ├── kahadb
│       └── tmp
├── dynamic-broker1
│   ├── bin
│   ├── conf
│   └── data
│       ├── kahadb
│       └── tmp
├── dynamic-broker2
│   ├── bin
│   ├── conf
│   └── data
│       ├── kahadb
│       └── tmp
└── dynamic-broker3
    ├── bin
    └── conf
        └── data

```

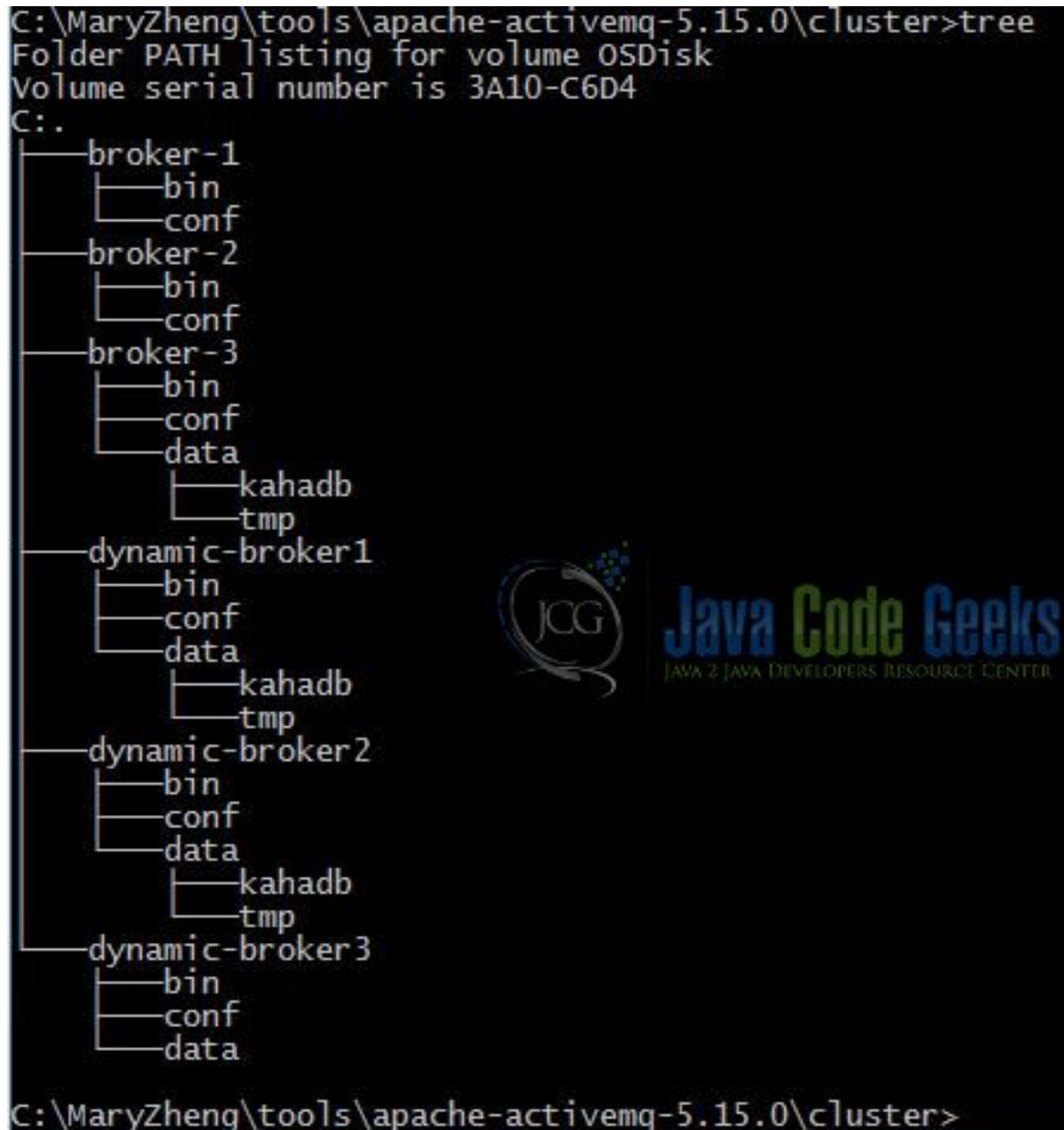


Figure 7.6: Cluster of AMQ Brokers

Configuration file example for dynamic Broker 1.

dynamic broker1 activemq.xml

```

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="https://www.springframework.org/schema/ ←
beans" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https: ←
//www.springframework.org/schema/beans https://www.springframework.org/schema/beans/ ←
spring-beans.xsd https://activemq.apache.org/schema/core https://activemq.apache.org/ ←
schema/core/activemq-core.xsd">

<!-- Allows us to use system properties as variables in this configuration file -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <value>file:${activemq.conf}/credentials.properties</value>

```

```

    </property>
</bean>

<!-- Allows accessing the server log -->
<bean class="io.fabric8.insight.log.log4j.Log4jLogQuery" destroy-method="stop" id=" ←
    logQuery" init-method="start" lazy-init="false" scope="singleton">
</bean>

<!--
    The <broker> element is used to configure the ActiveMQ broker.
-->
<broker xmlns="https://activemq.apache.org/schema/core" brokerName="dynamic-broker1" ←
    dataDirectory="${activemq.data}">

<destinationInterceptors>
    <virtualDestinationInterceptor>
        <virtualDestinations>
            <virtualTopic name="VirtualTopic.>" prefix="Consumer.*" ←
                selectorAware="false"/>
            <virtualTopic name="JCG.>" prefix="VTC.*" selectorAware="true"/>
        </virtualDestinations>
    </virtualDestinationInterceptor>
</destinationInterceptors>

<destinationPolicy>
    <policyMap>
        <policyEntries>
            <policyEntry topic="">
                <!-- The constantPendingMessageLimitStrategy is used to prevent
                    slow topic consumers to block producers and affect other consumers
                    by limiting the number of messages that are retained
                    For more information, see:

                    https://activemq.apache.org/slow-consumer-handling.html
                -->
                <pendingMessageLimitStrategy>
                    <constantPendingMessageLimitStrategy limit="1000"/>
                </pendingMessageLimitStrategy>
            </policyEntry>
        </policyEntries>
    </policyMap>
</destinationPolicy>

<!--
    The managementContext is used to configure how ActiveMQ is exposed in
    JMX. By default, ActiveMQ uses the MBean server that is started by
    the JVM. For more information, see:

    https://activemq.apache.org/jmx.html
-->
<managementContext>
    <managementContext createConnector="false"/>
</managementContext>

<!--
    Configure message persistence for the broker. The default persistence
    mechanism is the KahaDB store (identified by the kahaDB tag).
    For more information, see:

    https://activemq.apache.org/persistence.html
-->

```

```
<persistenceAdapter>
  <kahaDB directory="{activemq.data}/kahadb"/>
</persistenceAdapter>

<!--
The systemUsage controls the maximum amount of space the broker will
use before disabling caching and/or slowing down producers. For more ←
information, see:
https://activemq.apache.org/producer-flow-control.html
-->
<systemUsage>
  <systemUsage>
    <memoryUsage>
      <memoryUsage percentOfJvmHeap="70"/>
    </memoryUsage>
    <storeUsage>
      <storeUsage limit="100 gb"/>
    </storeUsage>
    <tempUsage>
      <tempUsage limit="50 gb"/>
    </tempUsage>
  </systemUsage>
</systemUsage>

<networkConnectors>
  <networkConnector uri="multicast://default"
    dynamicOnly="true"
    networkTTL="3"
    prefetchSize="1"
    decreaseNetworkConsumerPriority="true" />
</networkConnectors>

<!--
The transport connectors expose ActiveMQ over a given protocol to
clients and other brokers. For more information, see:

https://activemq.apache.org/configuring-transport.html
-->
<transportConnectors>
  <!-- DOS protection, limit concurrent connections to 1000 and frame size to 100 ←
  MB -->
  <transportConnector name="openwire" rebalanceClusterClients="true" ←
    updateClusterClients="true" updateClusterClientsOnRemove="true" uri="tcp: ←
    //0.0.0.0:61626?maximumConnections=1000&wireFormat.maxFrameSize=104857600" ←
    discoveryUri="multicast://default" />
</transportConnectors>

<!-- destroy the spring context on shutdown to stop jetty -->
<shutdownHooks>
  <bean xmlns="https://www.springframework.org/schema/beans" class="org.apache. ←
    activemq.hooks.SpringContextHook"/>
</shutdownHooks>

</broker>

<!--
Enable web consoles, REST and Ajax APIs and demos
The web consoles requires by default login, you can disable this in the jetty.xml ←
file
```

```

    Take a look at ${ACTIVEMQ_HOME}/conf/jetty.xml for more details
-->
<import resource="jetty.xml"/>

</beans>

```

- **Line 91-97:** set uri="multicast://default" at networkConnector
- **Line 107:** set discoveryUri="multicast://default" and rebalanceClusterClients="true" update ClusterClients="true" updateClusterClientsOnRemove="true"

Start three dynamic brokers.

Dynamic broker3 server.log

```

C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\dynamic-broker3\bin>dynamic-broker3 ←
.bat start
Java Runtime: Oracle Corporation 1.8.0_31 C:\MaryZheng\tools\java\jdk1.8.0_31\jre
Heap sizes: current=1005056k free=984084k max=1005056k
JVM args: -Dcom.sun.management.jmxremote -Xms1G -Xmx1G -Djava.util.logging.config.file= ←
logging.properties -Djava.security.auth.login.config=C:/MaryZheng/too
ls/apache-activemq-5.15.0/cluster/dynamic-broker3/conf\login.config -Dactivemq.classpath=C ←
:/MaryZheng/tools/apache-activemq-5.15.0/cluster/dynamic-broker3/conf;
C:/MaryZheng/tools/apache-activemq-5.15.0/cluster/dynamic-broker3/conf;C:/MaryZheng/tools/ ←
apache-activemq-5.15.0/conf; -Dactivemq.home=C:/MaryZheng/tools/apache-activemq-5.15.0 - ←
Dactivemq.base=C:/MaryZheng/tools/apache-activemq-5.15.0/clust
r/dynamic-broker3 -Dactivemq.conf=C:/MaryZheng/tools/apache-activemq-5.15.0/cluster/dynamic ←
broker3/conf -Dactivemq.data=C:/MaryZheng/tools/apache-activemq-5.15.0/cluster/dynamic- ←
broker3/data -Djava.io.tmpdir=C:/MaryZheng/tools/apache-activemq-5.15.0/cluster/dynamic- ←
broker3/data\tmp
Extensions classpath:
[C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\dynamic-broker3\lib,C:\ ←
MaryZheng\tools\apache-activemq-5.15.0\lib,C:\MaryZheng\tools\apache-activemq ←
-5.15.0\cluster\dynamic-broker3\lib\camel,C:\MaryZheng\tools\apache-activemq ←
-5.15.0\cluster\dynamic-broker3\lib\optional,C:\MaryZheng\tools\apache-activemq ←
-5.15.0\cluster\dynamic-broker3\lib\web,C:\MaryZheng\tools\apache-activemq ←
-5.15.0\cluster\dynamic-broker3\lib\extra,C:\MaryZheng\tools\apache-activemq ←
-5.15.0\lib\camel,C:\MaryZheng\tools\apache-activemq-5.15.0\lib\optional,C:\ ←
MaryZheng\tools\apache-activemq-5.15.0\lib\web,C:\MaryZheng\tools\apache- ←
activemq-5.15.0\lib\extra]
ACTIVEMQ_HOME: C:\MaryZheng\tools\apache-activemq-5.15.0
ACTIVEMQ_BASE: C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\dynamic-broker3
ACTIVEMQ_CONF: C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\dynamic-broker3\conf
ACTIVEMQ_DATA: C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\dynamic-broker3\data
Loading message broker from: xbean:activemq.xml
INFO | Refreshing org.apache.activemq.xbean.XBeanBrokerFactory$1@7c16905e: startup date [ ←
Sat Dec 16 09:48:42 CST 2017]; root of context hierarchy
INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\MaryZheng\tools\apache- ←
activemq-5.15.0\cluster\dynamic-broker3\data\kahadb]
INFO | PListStore:[C:\MaryZheng\tools\apache-activemq-5.15.0\cluster\dynamic-broker3 ←
\data\dynamic-broker3\tmp_storage] started
INFO | Apache ActiveMQ 5.15.0 (dynamic-broker3, ID:SL2LS431841-65244-1513439325237-0:1) is ←
starting
INFO | Listening for connections at: tcp://SL2LS431841:61646?maximumConnections=1000& ←
wireFormat.maxFrameSize=104857600
INFO | Connector openwire started
INFO | Network Connector DiscoveryNetworkConnector:NC:BrokerService[dynamic-broker3] ←
started
INFO | Apache ActiveMQ 5.15.0 (dynamic-broker3, ID:SL2LS431841-65244-1513439325237-0:1) ←
started
INFO | For help or more information please see: https://activemq.apache.org

```

```

WARN | Store limit is 102400 mb (current store usage is 0 mb). The data directory: C:\\ ←
      MaryZheng\\tools\\apache-activemq-5.15.0\\cluster\\dynamic-broker3\\data\\kahadb only ←
      has 1154 mb of usable space. - resetting to maximum available disk space: 1154 mb
WARN | Temporary Store limit is 51200 mb (current store usage is 0 mb). The data directory ←
      : C:\\MaryZheng\\tools\\apache-activemq-5.15.0\\cluster\\dynamic-broker3\\data only has ←
      1154 mb of usable space. - resetting to maximum available disk space: 1154 mb
INFO | Establishing network connection from vm://dynamic-broker3 to tcp://SL2LS431841 ←
      :61646
INFO | Connector vm://dynamic-broker3 started
INFO | dynamic-broker3 Shutting down NC
INFO | dynamic-broker3 bridge to Unknown stopped
WARN | Transport Connection to: tcp://192.168.1.109:65245 failed: java.io.EOFException
INFO | Connector vm://dynamic-broker3 stopped
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | ActiveMQ WebConsole available at https://0.0.0.0:8164/
INFO | ActiveMQ Jolokia REST API available at https://0.0.0.0:8164/api/jolokia/
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | No Spring WebApplicationInitializer types detected on classpath
INFO | jolokia-agent: Using policy access restrictor classpath:/jolokia-access.xml
INFO | Establishing network connection from vm://dynamic-broker3 to tcp://SL2LS431841 ←
      :61636
INFO | Connector vm://dynamic-broker3 started
INFO | Network connection between vm://dynamic-broker3#2 and tcp://SL2LS431841 ←
      /192.168.1.109:61636@65254 (dynamic-broker2) has been established.
INFO | Establishing network connection from vm://dynamic-broker3 to tcp://SL2LS431841 ←
      :61626
INFO | Network connection between vm://dynamic-broker3#4 and tcp://SL2LS431841 ←
      /192.168.1.109:61626@65266 (dynamic-broker1) has been established.

```

- Line 22: dynamic Broker 3 starts as a standalone broker
- Line 26-27: dynamic Broker 3 starts as a networked broker
- Line 38: establish the network connection from Broker 3 to 2
- Line 40: the network connection is established between Broker 3 and 2
- Line 41: establish the network connection from Broker 3 to 1
- Line 42: the network connection is established between Broker 3 and 1

Note: Try to stop any of these dynamic brokers and watch the other broker's server log. Verify the connection via AMQ web console.

7.4 Create Java Client Applications

Create two Java AMQ client applications. One is a producer application which sends ten dummy messages to a `test.queue`. The other is a consumer application which consumes the messages from the `test.queue`.

7.4.1 Common Data

Create a common data class to hold the data used in the Demo.

DemoDataUtils

```

package jcg.demo.util;

import java.util.Random;

```

```
import java.util.Scanner;

/**
 * The constant data utility used in this Demo
 *
 * @author Mary.Zheng
 *
 */
public final class DemoDataUtils {

    public static final int MESSAGE_SIZE = 10;
    public static final String DESTINATION = "test.queue";

    private static final String COMMA = ",";
    private static final String FAILOVER = "failover(";

    private static final String HALF_MINUTE_TIMEOUT = ")?timeout=30000";

    private static String STANDALONE_BROKER_1 = "tcp://localhost:61616";
    private static String STANDALONE_BROKER_2 = "tcp://localhost:61716";

    private static String MASTER_BROKER_1 = "tcp://localhost:61816";
    private static String SLAVE_BROKER_2 = "tcp://localhost:61826";

    private static String[] STATIC_NC_BROKER = { "tcp://localhost:61516", ←
        MASTER_BROKER_1, SLAVE_BROKER_2 };
    private static String[] DYNAMIC_NC_BROKER = { "tcp://localhost:61626", "tcp:// ←
        localhost:61636",
            "tcp://localhost:61646" };

    private static String getStaticNCBroker() {
        return STATIC_NC_BROKER[0];
    }

    private static String getDynamicNCBroker() {
        Random rand = new Random();
        int value = rand.nextInt(3);
        return DYNAMIC_NC_BROKER[value];
    }

    public static String buildDummyMessage(int value) {
        return "dummy message " + value;
    }

    private static String getFailOverURI(String... uris) {

        String[] brokerURIs = uris;
        StringBuffer foUrl = new StringBuffer(FAILOVER);

        int brokerUriSize = brokerURIs.length;
        if (brokerUriSize == 1) {
            foUrl.append(brokerURIs[0]);
        } else {
            for (String brokerUri : brokerURIs) {
                foUrl.append(brokerUri);
                if (!brokerUri.equalsIgnoreCase(brokerURIs[brokerUriSize - ←
                    1])) {
                    foUrl.append(COMMA);
                }
            }
        }
    }
}
```

```

        foUrl.append(HALF_MINUTE_TIMEOUT);// fast fails
        return foUrl.toString();
    }

    public static String readFailoverURL() {
        String promptMessage = "Enter Demo Type for Failover:"
            + "\n\t1 - Stand Alone Brokers \n\t2 - Master-Slave ↔
              ↔
              Brokers \n\t3 - Network of Brokers(Static) \n\t4 - ↔
              ↔
              Network of Brokers(Dynamic): ";
        System.out.println(promptMessage);
        String failoverUrl = null;

        try (Scanner scanIn = new Scanner(System.in)) {
            String inputString = scanIn.nextLine();
            scanIn.close();
            switch (inputString) {
                case "1":
                    failoverUrl = DemoDataUtils.getFailOverURI( ↔
                        STANDALONE_BROKER_1, STANDALONE_BROKER_2);
                    break;
                case "2":
                    failoverUrl = DemoDataUtils.getFailOverURI(MASTER_BROKER_1, ↔
                        SLAVE_BROKER_2);
                    break;
                case "3":
                    failoverUrl = DemoDataUtils.getFailOverURI( ↔
                        getStaticNCBroker());
                    break;
                case "4":
                    failoverUrl = DemoDataUtils.getFailOverURI( ↔
                        getDynamicNCBroker());
                    break;
            }
        }

        return failoverUrl;
    }
}

```

- Line 33: client knows only the static broker
- Line 37-39: reduce the broker load with total number of broker in dynamic network of brokers
- Line 78: client knows all the brokers in the standalone broker topology
- Line 81: client knows all the brokers in the Master/Slave broker topology
- Line 84: client knows only the static broker
- Line 87: client knows any one of the brokers in the dynamic network

7.4.2 QueueMessageProducer

Create QueueMessageProducer .

QueueMessageProducer

```

package jcg.demo.activemq.failover;

import javax.jms.Connection;

```



```
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

import jcg.demo.util.DemoDataUtils;

/**
 * A simple message producer which sends the message to ActiveMQ Broker
 *
 * @author Mary.Zheng
 *
 */
public class QueueMessageProducer {

    private String activeMqBrokerUri;
    private String username;
    private String password;

    public QueueMessageProducer(String activeMqBrokerUri, String username, String password) {
        super();
        this.activeMqBrokerUri = activeMqBrokerUri;
        this.username = username;
        this.password = password;
    }

    public void sendDummyMessages(String queueName) {
        System.out.println("QueueMessageProducer started " + this.activeMqBrokerUri);

        ConnectionFactory connFactory = null;
        Connection connection = null;
        Session session = null;
        MessageProducer msgProducer = null;
        try {
            connFactory = new ActiveMQConnectionFactory(username, password, activeMqBrokerUri);
            connection = connFactory.createConnection();
            connection.start();
            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            msgProducer = session.createProducer(session.createQueue(queueName));

            for (int i = 0; i < DemoDataUtils.MESSAGE_SIZE; i++) {
                TextMessage textMessage = session.createTextMessage(DemoDataUtils.buildDummyMessage(i));
                msgProducer.send(textMessage);
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e) {}
            }
            System.out.println("QueueMessageProducer completed");
        } catch (JMSEException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
        try {
            if (msgProducer != null) {

```

```
        msgProducer.close();
    }
    if (session != null) {
        session.close();
    }
    if (connection != null) {
        connection.close();
    }
} catch (Throwable ignore) {
}
}
```

7.4.3 MessageProducerApp

A Java application which sends ten messages at 10 seconds intervals.

MessageProducerApp

```
package jcg.demo.activemq.failover;

import jcg.demo.util.DemoDataUtils;

public class MessageProducerApp {

    public static void main(String[] args) {
        String failoverUrl = DemoDataUtils.readFailoverURL();

        if (failoverUrl == null) {
            System.out.println("Wrong input");
        } else {
            QueueMessageProducer queProducer = new QueueMessageProducer( ←
                failoverUrl, "admin", "admin");
            queProducer.sendDummyMessages(DemoDataUtils.DESTINATION);
        }
    }
}
```

7.4.4 QueueMessageConsumer

Create QueueMessageConsumer.

QueueMessageConsumer

```
package jcg.demo.activemq.failover;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

/**
 * A simple message consumer which consumes the message from ActiveMQ Broker
 */
```

```
*
* @author Mary.Zheng
*
*/
public class QueueMessageConsumer implements MessageListener {

    private String activeMqBrokerUri;
    private String username;
    private String password;
    private String destinationName;

    public QueueMessageConsumer(String activeMqBrokerUri, String username, String ←
        password) {
        super();
        this.activeMqBrokerUri = activeMqBrokerUri;
        this.username = username;
        this.password = password;
    }

    public void run() throws JMSEException {
        ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(username, ←
            password, activeMqBrokerUri);
        Connection connection = factory.createConnection();
        connection.start();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE) ←
            ;
        Destination destination = session.createQueue(destinationName);
        MessageConsumer consumer = session.createConsumer(destination);
        consumer.setMessageListener(this);

        System.out.println(String.format("QueueMessageConsumer Waiting for messages ←
            at %s %s",
                destinationName, this.activeMqBrokerUri));
    }

    @Override
    public void onMessage(Message message) {
        String msg;
        try {
            msg = String.format("QueueMessageConsumer Received message [ %s ]",
                ((TextMessage) message).getText());
            Thread.sleep(10000); // sleep for 10 seconds
            System.out.println(msg);
        } catch (JMSEException | InterruptedException e) {
            e.printStackTrace();
        }
    }

    public String getDestinationName() {
        return destinationName;
    }

    public void setDestinationName(String destinationName) {
        this.destinationName = destinationName;
    }
}
```

7.4.5 MessageConsumerApp

A Java application which consumes messages at 10 seconds intervals.

MessageConsumerApp

```

package jcg.demo.activemq.failover;

import javax.jms.JMSEException;

import jcg.demo.util.DemoDataUtils;

public class MessageConsumerApp {

    public static void main(String[] args) {

        String failoverUrl = DemoDataUtils.readFailoverURL();

        if (failoverUrl == null) {
            System.out.println("Wrong input");
        } else {
            QueueMessageConsumer queueMsgListener = new QueueMessageConsumer(
                failoverUrl, "admin", "admin");
            queueMsgListener.setDestinationName(DemoDataUtils.DESTINATION);

            try {
                queueMsgListener.run();
            } catch (JMSEException e) {
                e.printStackTrace();
            }
        }
    }
}

```

7.5 Demo Time

This is the most fun moment of this example. I will show the Java application built at chapter 7.4 continue functions when the connected AMQ broker is down. Start eight brokers configured at chapter 7.3, then start the `MessageConsumerApp` and `MessageProducerApp`. While both programs are running, stop the connected AMQ broker. Both applications auto-detect the failure and then reconnect to a different broker.

7.5.1 Two Standalone Brokers

Below you can find the `MessageProducerApp` output.

MessageProducerApp output

```

Enter Demo Type for Failover:
    1 - Stand Alone Brokers
    2 - Master-Slave Brokers
    3 - Network of Brokers(Static)
    4 - Network of Brokers(Dynamic):
1
QueueMessageProducer started failover:(tcp://localhost:61616,tcp://localhost:61716)?timeout=30000
INFO | Successfully connected to tcp://localhost:61616
WARN | Transport (tcp://localhost:61616) failed , attempting to automatically reconnect:
{}
java.io.EOFException
    at java.io.DataInputStream.readInt(DataInputStream.java:392)
    at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:268)

```

```

    at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java ←
      :240)
    at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:232)
    at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:215)
    at java.lang.Thread.run(Thread.java:745)
INFO | Successfully reconnected to tcp://localhost:61716
QueueMessageProducer completed

```

- Line 6: input from user
- Line 7: client failover URI with both servers
- Line 8: client connects to `broker1 (61616)`
- Line 9: client detects the connection failed and try to reconnect
- Line 18: client reconnects to `broker2 (61716)`

Verify via AMQ web console, notice that there is one message at broker 1 and 9 messages at broker 2. (the number can vary depends on the shut down timing).

Below you can find the `MessageConsumerApp` output.

MessageConsumerApp output

```

Enter Demo Type for Failover:
  1 - Stand Alone Brokers
  2 - Master-Slave Brokers
  3 - Network of Brokers(Static)
  4 - Network of Brokers(Dynamic) :
1
INFO | Successfully connected to tcp://localhost:61716
QueueMessageConsumer Waiting for messages at test.queue failover:(tcp://localhost:61616,tcp ←
://localhost:61716)?timeout=30000
QueueMessageConsumer Received message [ dummy message 1 ]
QueueMessageConsumer Received message [ dummy message 2 ]
QueueMessageConsumer Received message [ dummy message 3 ]
QueueMessageConsumer Received message [ dummy message 4 ]
QueueMessageConsumer Received message [ dummy message 5 ]
QueueMessageConsumer Received message [ dummy message 6 ]
QueueMessageConsumer Received message [ dummy message 7 ]
QueueMessageConsumer Received message [ dummy message 8 ]
QueueMessageConsumer Received message [ dummy message 9 ]
QueueMessageConsumer Received message [ dummy message 0 ]
WARN | Transport (tcp://localhost:61616) failed , attempting to automatically reconnect: ←
{}
java.io.EOFException
  at java.io.DataInputStream.readInt(DataInputStream.java:392)
  at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:268)
  at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java ←
:240)
  at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:232)
  at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:215)
  at java.lang.Thread.run(Thread.java:745)
INFO | Successfully reconnected to tcp://localhost:61716

```

- Line 6: input from user
- Line 7: client connects to `broker2 (61716)`
- Line 19: client detects the connection failed and try to reconnect
- Line 27: client reconnects to `broker2 (61716)`

7.5.2 Master/Slave Brokers

Repeat above steps for the Master/Slave brokers.

Execution for Master/Slave brokers

```

Enter Demo Type for Failover:
    1 - Stand Alone Brokers
    2 - Master-Slave Brokers
    3 - Network of Brokers(Static)
    4 - Network of Brokers(Dynamic):
2
QueueMessageProducer started failover:(tcp://localhost:61816,tcp://localhost:61826)?timeout ←
=30000
INFO | Successfully connected to tcp://localhost:61826
WARN | Transport (tcp://localhost:61826) failed , attempting to automatically reconnect: ←
{}
java.io.EOFException
    at java.io.DataInputStream.readInt(DataInputStream.java:392)
    at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:268)
    at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java ←
:240)
    at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:232)
    at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:215)
    at java.lang.Thread.run(Thread.java:745)
INFO | Successfully reconnected to tcp://localhost:61816
QueueMessageProducer completed

*****

Enter Demo Type for Failover:
    1 - Stand Alone Brokers
    2 - Master-Slave Brokers
    3 - Network of Brokers(Static)
    4 - Network of Brokers(Dynamic):
2
INFO | Successfully connected to tcp://localhost:61826
QueueMessageConsumer Waiting for messages at test.queue failover:(tcp://localhost:61816,tcp ←
://localhost:61826)?timeout=30000
QueueMessageConsumer Received message [ dummy message 0 ]
WARN | Transport (tcp://localhost:61826) failed , attempting to automatically reconnect: ←
{}
java.io.EOFException
    at java.io.DataInputStream.readInt(DataInputStream.java:392)
    at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:268)
    at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java ←
:240)
    at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:232)
    at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:215)
    at java.lang.Thread.run(Thread.java:745)
QueueMessageConsumer Received message [ dummy message 1 ]
INFO | Successfully reconnected to tcp://localhost:61816
WARN | ID:SL2LS431841-51754-1513648383975-1:1:1:1 suppressing duplicate delivery on ←
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ←
consumerId = ID:SL2LS431841-51754-1513648383975-1:1:1:1, destination = queue://test. ←
queue, message = ActiveMQTextMessage {commandId = 6, responseRequired = true, messageId ←
= ID:SL2LS431841-50044-1513371045088-1:1:1:1:2, originalDestination = null, ←
originalTransactionId = null, producerId = ID:SL2LS431841-50044-1513371045088-1:1:1:1, ←
destination = queue://test.queue, transactionId = null, expiration = 0, timestamp = ←
1513371056504, arrival = 0, brokerInTime = 1513371056505, brokerOutTime = ←
1513648412924, correlationId = null, replyTo = null, persistent = true, type = null, ←
priority = 4, groupId = null, groupSequence = 0, targetConsumerId = null, compressed = ←
false, userID = null, content = org.apache.activemq.util.ByteSequence@59b92659, ←

```

```

    marshalledProperties = null, dataStructure = null, redeliveryCounter = 0, size = 0, ←
    properties = null, readOnlyProperties = true, readOnlyBody = true, droppable = false, ←
    jmsXGroupFirstForConsumer = false, text = dummy message 1}, redeliveryCounter = 0}
QueueMessageConsumer Received message [ dummy message 2 ]
QueueMessageConsumer Received message [ dummy message 3 ]
QueueMessageConsumer Received message [ dummy message 4 ]
QueueMessageConsumer Received message [ dummy message 5 ]
QueueMessageConsumer Received message [ dummy message 6 ]
QueueMessageConsumer Received message [ dummy message 7 ]
QueueMessageConsumer Received message [ dummy message 8 ]
QueueMessageConsumer Received message [ dummy message 9 ]
QueueMessageConsumer Received message [ dummy message 0 ]
QueueMessageConsumer Received message [ dummy message 1 ]
QueueMessageConsumer Received message [ dummy message 2 ]
QueueMessageConsumer Received message [ dummy message 3 ]
QueueMessageConsumer Received message [ dummy message 4 ]
QueueMessageConsumer Received message [ dummy message 5 ]
QueueMessageConsumer Received message [ dummy message 6 ]
QueueMessageConsumer Received message [ dummy message 7 ]
QueueMessageConsumer Received message [ dummy message 8 ]

```

- Line 6: input from user
- Line 7: client failover URI with both servers
- Line 8: client connects to server at 61826
- Line 9: client detects the connection failed and try to reconnect
- Line 17: client reconnects to server at 61816
- Line 27: input from user
- Line 28: client connects to server at 61826
- Line 31: client detects the connection failed and try to reconnect
- Line 40: client reconnects to server at 61816

7.5.3 Static Network of Brokers

Repeat above steps for the static network of brokers.

Execution for static network of brokers

```

Enter Demo Type for Failover:
    1 - Stand Alone Brokers
    2 - Master-Slave Brokers
    3 - Network of Brokers(Static)
    4 - Network of Brokers(Dynamic):
3
QueueMessageProducer started failover:(tcp://localhost:61516)?timeout=30000
INFO | Successfully connected to tcp://localhost:61516
INFO | Successfully reconnected to tcp://SL2LS431841:61516
WARN | Transport (tcp://SL2LS431841:61516) failed , attempting to automatically reconnect: ←
    {}
java.io.EOFException
    at java.io.DataInputStream.readInt(DataInputStream.java:392)
    at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:268)
    at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java: ←
        :240)
    at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:232)

```

```

        at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:215)
        at java.lang.Thread.run(Thread.java:745)
INFO | Successfully reconnected to tcp://SL2LS431841:61816
QueueMessageProducer completed

Enter Demo Type for Failover:
    1 - Stand Alone Brokers
    2 - Master-Slave Brokers
    3 - Network of Brokers(Static)
    4 - Network of Brokers(Dynamic):
3
INFO | Successfully connected to tcp://localhost:61516
INFO | Successfully reconnected to tcp://SL2LS431841:61816
QueueMessageConsumer Waiting for messages at test.queue failover:(tcp://localhost:61516)? ←
    timeout=30000
QueueMessageConsumer Received message [ dummy message 0 ]
QueueMessageConsumer Received message [ dummy message 0 ]
QueueMessageConsumer Received message [ dummy message 1 ]
QueueMessageConsumer Received message [ dummy message 2 ]
QueueMessageConsumer Received message [ dummy message 3 ]
QueueMessageConsumer Received message [ dummy message 4 ]
QueueMessageConsumer Received message [ dummy message 5 ]
QueueMessageConsumer Received message [ dummy message 6 ]
QueueMessageConsumer Received message [ dummy message 7 ]
QueueMessageConsumer Received message [ dummy message 8 ]
QueueMessageConsumer Received message [ dummy message 9 ]

```

- Line 6: input from user
- Line 7: client failover URI with both servers
- Line 8-9: client connects to server at 61516
- Line 10: client detects the connection failed and try to reconnect
- Line 18: client reconnects to server at 61816
- Line 27: input from user
- Line 28: client connects to server at 61516
- Line 29: client connects to server at 61816
- Line 30: client failover URI is static

7.5.4 Dynamic Network of Brokers

Repeat above steps for the dynamic network of brokers.

Execution output for dynamic network of brokers

```

Enter Demo Type for Failover:
    1 - Stand Alone Brokers
    2 - Master-Slave Brokers
    3 - Network of Brokers(Static)
    4 - Network of Brokers(Dynamic):
4
QueueMessageProducer started failover:(tcp://localhost:61646)?timeout=30000
INFO | Successfully connected to tcp://localhost:61646
INFO | Successfully reconnected to tcp://SL2LS431841:61626

```



```

WARN | Transport (tcp://SL2LS431841:61626) failed , attempting to automatically reconnect: ←
{}
java.io.EOFException
  at java.io.DataInputStream.readInt(DataInputStream.java:392)
  at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:268)
  at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java ←
:240)
  at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:232)
  at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:215)
  at java.lang.Thread.run(Thread.java:745)
INFO | Successfully reconnected to tcp://SL2LS431841:61646
QueueMessageProducer completed

*****

Enter Demo Type for Failover:
  1 - Stand Alone Brokers
  2 - Master-Slave Brokers
  3 - Network of Brokers(Static)
  4 - Network of Brokers(Dynamic):
4
INFO | Successfully connected to tcp://localhost:61636
INFO | Successfully reconnected to tcp://SL2LS431841:61626
QueueMessageConsumer Waiting for messages at test.queue failover:(tcp://localhost:61636)? ←
timeout=30000
QueueMessageConsumer Received message [ dummy message 0 ]
QueueMessageConsumer Received message [ dummy message 1 ]
QueueMessageConsumer Received message [ dummy message 2 ]
QueueMessageConsumer Received message [ dummy message 3 ]
QueueMessageConsumer Received message [ dummy message 4 ]
QueueMessageConsumer Received message [ dummy message 5 ]
INFO | Successfully reconnected to tcp://SL2LS431841:61636
QueueMessageConsumer Received message [ dummy message 6 ]
QueueMessageConsumer Received message [ dummy message 7 ]
QueueMessageConsumer Received message [ dummy message 4 ]
QueueMessageConsumer Received message [ dummy message 9 ]
QueueMessageConsumer Received message [ dummy message 5 ]
QueueMessageConsumer Received message [ dummy message 7 ]
QueueMessageConsumer Received message [ dummy message 6 ]
QueueMessageConsumer Received message [ dummy message 8 ]
QueueMessageConsumer Received message [ dummy message 9 ]
QueueMessageConsumer Received message [ dummy message 8 ]
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ←
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ←
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ←
queue, message = ActiveMQTextMessage {commandId = 11, responseRequired = true, ←
messageId = ID:SL2LS431841-54039-1513676753374-1:1:1:1:7, originalDestination = null, ←
originalTransactionId = null, producerId = dynamic-broker1->dynamic-broker2 ←
-54099-1513676883550-5:2:1:1, destination = queue://test.queue, transactionId = null, ←
expiration = 0, timestamp = 1513676813826, arrival = 0, brokerInTime = 1513676968988, ←
brokerOutTime = 1513676968989, correlationId = null, replyTo = null, persistent = true, ←
type = null, priority = 4, groupId = null, groupSequence = 0, targetConsumerId = null, ←
compressed = false, userID = null, content = org.apache.activemq.util. ←
ByteSequence@6fb0f2ba, marshalledProperties = null, dataStructure = null, ←
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ←
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ←
message 6}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ←
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ←
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ←

```

```
queue, message = ActiveMQTextMessage {commandId = 12, responseRequired = true, ↵
messageId = ID:SL2LS431841-54039-1513676753374-1:1:1:1:8, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker1->dynamic-broker2 ↵
-54099-1513676883550-5:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676823835, arrival = 0, brokerInTime = 1513676968999, ↵
brokerOutTime = 1513676969000, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupId = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@1133bfa0, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 7}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 13, responseRequired = true, ↵
messageId = ID:SL2LS431841-54039-1513676753374-1:1:1:1:10, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker1->dynamic-broker2 ↵
-54099-1513676883550-5:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676843872, arrival = 0, brokerInTime = 1513676969006, ↵
brokerOutTime = 1513676969007, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupId = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@6c74ff09, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 9}, redeliveryCounter = 1}
INFO | Successfully reconnected to tcp://SL2LS431841:61626
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 7, responseRequired = true, messageId ↵
= ID:SL2LS431841-53789-1513676183711-1:1:1:1:5, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker2->dynamic-broker1 ↵
-54081-1513676839955-7:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676224129, arrival = 0, brokerInTime = 1513676902269, ↵
brokerOutTime = 1513677166569, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupId = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@4ec525c9, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 4}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 9, responseRequired = true, messageId ↵
= ID:SL2LS431841-53789-1513676183711-1:1:1:1:6, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker3->dynamic-broker1 ↵
-54012-1513676733269-11:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676234150, arrival = 0, brokerInTime = 1513676902309, ↵
brokerOutTime = 1513677166570, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupId = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@4002d261, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 5}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 8, responseRequired = true, messageId ↵
```

```
= ID:SL2LS431841-53789-1513676183711-1:1:1:8, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker2->dynamic-broker1 ↵
-54081-1513676839955-7:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676254168, arrival = 0, brokerInTime = 1513676902311, ↵
brokerOutTime = 1513677166570, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupID = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@443d4eb9, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 7}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 10, responseRequired = true, ↵
messageId = ID:SL2LS431841-53789-1513676183711-1:1:1:7, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker3->dynamic-broker1 ↵
-54012-1513676733269-11:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676244161, arrival = 0, brokerInTime = 1513676902317, ↵
brokerOutTime = 1513677166570, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupID = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@13fd85a, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 6}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 9, responseRequired = true, messageId ↵
= ID:SL2LS431841-53789-1513676183711-1:1:1:9, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker2->dynamic-broker1 ↵
-54081-1513676839955-7:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676264175, arrival = 0, brokerInTime = 1513676902319, ↵
brokerOutTime = 1513677166571, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupID = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@66b34ed9, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 8}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 11, responseRequired = true, ↵
messageId = ID:SL2LS431841-53789-1513676183711-1:1:1:10, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker3->dynamic-broker1 ↵
-54012-1513676733269-11:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676274182, arrival = 0, brokerInTime = 1513676902323, ↵
brokerOutTime = 1513677166577, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupID = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@30f419b4, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 9}, redeliveryCounter = 1}
WARN | ID:SL2LS431841-54111-1513676901862-1:1:1:1 suppressing duplicate delivery on ↵
connection, poison acking: MessageDispatch {commandId = 0, responseRequired = false, ↵
consumerId = ID:SL2LS431841-54111-1513676901862-1:1:1:1, destination = queue://test. ↵
queue, message = ActiveMQTextMessage {commandId = 10, responseRequired = true, ↵
messageId = ID:SL2LS431841-54039-1513676753374-1:1:1:9, originalDestination = null, ↵
originalTransactionId = null, producerId = dynamic-broker2->dynamic-broker1 ↵
```

```
-54081-1513676839955-7:2:1:1, destination = queue://test.queue, transactionId = null, ↵
expiration = 0, timestamp = 1513676833866, arrival = 0, brokerInTime = 1513676902327, ↵
brokerOutTime = 1513677166585, correlationId = null, replyTo = null, persistent = true, ↵
type = null, priority = 4, groupId = null, groupSequence = 0, targetConsumerId = null, ↵
compressed = false, userID = null, content = org.apache.activemq.util. ↵
ByteSequence@1e92d195, marshalledProperties = null, dataStructure = null, ↵
redeliveryCounter = 1, size = 0, properties = null, readOnlyProperties = true, ↵
readOnlyBody = true, droppable = false, jmsXGroupFirstForConsumer = false, text = dummy ↵
message 8}, redeliveryCounter = 1}
```

- Line 6: user input
- Line 7: client failover URI
- Line 8: client connects to server at 61646
- Line 9: client connects to server at 61626
- Line 10: client detects server 61626 failed
- Line 18: client reconnects to server 61646
- Line 30: user input
- Line 31: client connects to server at 61636
- Line 32: client connects to server at 61626
- Line 33: client failover URI
- Line 40: client reconnects to server 61636
- Line 54: client reconnects to server 61626

7.6 Summary

AMQ provides a network connector to bridge any two brokers and provides failover transport connector to allow the client application to connect to a list of AMQ brokers. If the connection from the client to one broker fails, then the failover transport connector will automatically try to connect to the next broker and will keep trying until the connection is established or a retry limit is reached.

In the Java client application, we demonstrate that sometime the failed broker cause one message get lost in the standalone topology. We also demonstrate that the dynamic network broker not only provides the failover function but also reduces the load at each broker.

Topology	Message Loss	Client Friendly	Load Balanced
Standalone	Possible	No	No
Master/Slave	No	No	No
Static Network of Brokers	No	Yes	No
Dynamic Network of Brokers	No	Yes	Yes

Figure 7.7: Table

7.7 Download the Source Code

This example built two Java AMQ client applications along with four AMQ brokers typologies configuration files. You can download the full source code of this example here: [Apache ActiveMQ Failover Example](#)

Chapter 8

ActiveMQ Advisory Example

In this example, we will be discussing about Apache **ActiveMQ Advisory**. But before we start with our example, it is expected that we have a basic understanding of **JMS concepts**, **ActiveMQ** and **Java/J2EE**. JMS stands for Java Messaging API and ActiveMQ is a java based implementation of JMS. ActiveMQ acts as a message broker. It is open source and helps in asynchronous message transfer between producer and consumer using queues/topics.

So, whenever some activity occurs on ActiveMQ , there is a course of action that can be taken to receive notifications using **ActiveMQ Advisory** messages. For example, we can setup notifications on producer, consumer or destination.

8.1 Introduction

ActiveMQ Advisory acts as an administrative channel to monitor our system via JMS messages. We can get information on what is happening with our producers, consumers and destination topics/queues. In this chapter we will see how a consumer gets notified whenever a message is consumed. We will be using Java v8 and Active MQ v 5.14.5.

8.2 Configuration for ActiveMQ Advisory

- Download ActiveMQ from [ActiveMQ download](#) link
- Extract the ActiveMQ downloaded zip file to any location in your computer
- Goto conf directory of extracted activemq folder
- Open activemq.xml and search for xml tag `policyEntry` . It will be like this:

```
policyEntry topic=">"
```

- Change topic to queue (as we will be using queue in our example) and add `advisoryForDelivery` tag to setup advisory messages. The `>` matches all queues.

```
policyEntry queue=">" advisoryForDelivery="true"
```

- Now start activemq as described here [Say Hello To ActiveMQ](#)

8.3 Using ActiveMQ Advisory

8.3.1 Example 1 - Using ActiveMQ Advisory in a simple Java project (example using Eclipse)

- In this example, we will see how ActiveMQ advisory works in a simple way. The listener method `onMessage()` gets triggered whenever a message is received by the Receiver
- Let us now create a dynamic web project in eclipse and create our Sender and Receiver classes to see how a message is exchanged using ActiveMQ. A notification will be received when a consumer consumes a message
- The Sender and Receiver classes are created as separate threads and implements the Runnable interface
- The Receiver class implements `MessageListener` and hence override the `onMessage()` method
- The main method starts the Receiver thread, sleep for some time and then starts the Sender thread. So the moment Sender starts sending messages, the `onMessage()` method in Receiver class gets invoked
- Please refer the code snippet below for our Sender and Receiver classes

Sender.java

```
package com.activemq.advisory;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.DeliveryMode;
import javax.jms.Destination;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

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

public class Sender implements Runnable{

    //URL of the JMS server. DEFAULT_BROKER_URL will just mean that JMS server is on localhost
    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;

    // default broker URL is : tcp://localhost:61616"
    private static String subject = "JCG_QUEUE"; // Queue Name.You can create any/many queue names as per your requirement.

    @Override
    public void run() {
        try {
            // Getting JMS connection from the server and starting it
            ConnectionFactory activeMQConnectionFactory = new ActiveMQConnectionFactory(url);
            Connection connection = activeMQConnectionFactory.createConnection();
            connection.start();

            //Creating a non transactional session to send/receive JMS message.
            Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

            //Destination represents here our queue 'JCG_QUEUE' on the JMS server.
            //The queue will be created automatically on the server.
        }
    }
}
```

```

        Destination destination = session.createQueue(subject);

        // MessageProducer is used for sending messages to the queue.
        MessageProducer producer = session.createProducer(destination);
        producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

        // We will be sending text messages
        for (int i = 0; i < 100; i++) {
            String text = "JCG Message Count " + i;
            TextMessage message = session.createTextMessage(text);
            // Here we are sending our message!
            producer.send(message);
            System.out.println("JCG printing Sent message: " + message) ←
                ;
        }
        session.close();
        connection.close();
    } catch (Exception e) {
        System.out.println("Caught Exception: " + e);
        e.printStackTrace();
    }
}
}

```

Receiver.java

```

package com.activemq.advisory;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQTextMessage;

public class Receiver implements Runnable, ExceptionListener, MessageListener {

    // URL of the JMS server
    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;
    // default broker URL is : tcp://localhost:61616"

    // Name of the queue we will receive messages from
    private static String subject = "JCG_QUEUE";

    public Receiver() {

        try {
            // Getting JMS connection from the server
            ConnectionFactory activeMQConnectionFactory = new ←
                ActiveMQConnectionFactory(
                    url);
            Connection connection = activeMQConnectionFactory
                .createConnection();
            connection.start();

```

```

        connection.setExceptionListener(this);

        // Creating session for receiving messages
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);

        // Getting the queue 'JCG_QUEUE'
        Destination destination = session.createQueue(subject);

        // MessageConsumer is used for receiving (consuming) messages
        MessageConsumer consumer = session.createConsumer(destination);
        consumer.setMessageListener(this); // Setting message listener
    } catch (Exception e) {
        System.out.println("Caught exception: " + e);
        e.printStackTrace();
    }
}

public void run() {
    // Make Consumer a Daemon thread.
    while (true) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public synchronized void onException(JMSEException ex) {
    System.out.println("JCG ActiveMQ JMS Exception occurred. Shutting down ↵
        client.");
}

public void onMessage(Message msg) {
    ActiveMQTextMessage tm = (ActiveMQTextMessage) msg;
    try {
        System.out.println(" JCG inside onMessage:: Received Message::" + ↵
            tm.getText());
    } catch (JMSEException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws Exception {
    (new Thread(new Receiver())).start();
    Thread.sleep(10000);
    (new Thread(new Sender())).start();
}
}

```

Output:

We will be running our Receiver class which starts both Sender and Receiver threads written above to check how the onMessage () method is invoked.

Please follow the steps below:

- In the eclipse Right Click on Receiver.java → Run As → Java Application, to see if our message is sent to the queue and received by the Receiver class

- Check the eclipse console output for the sent and received message. Whenever a message is received, the onMessage () method is invoked

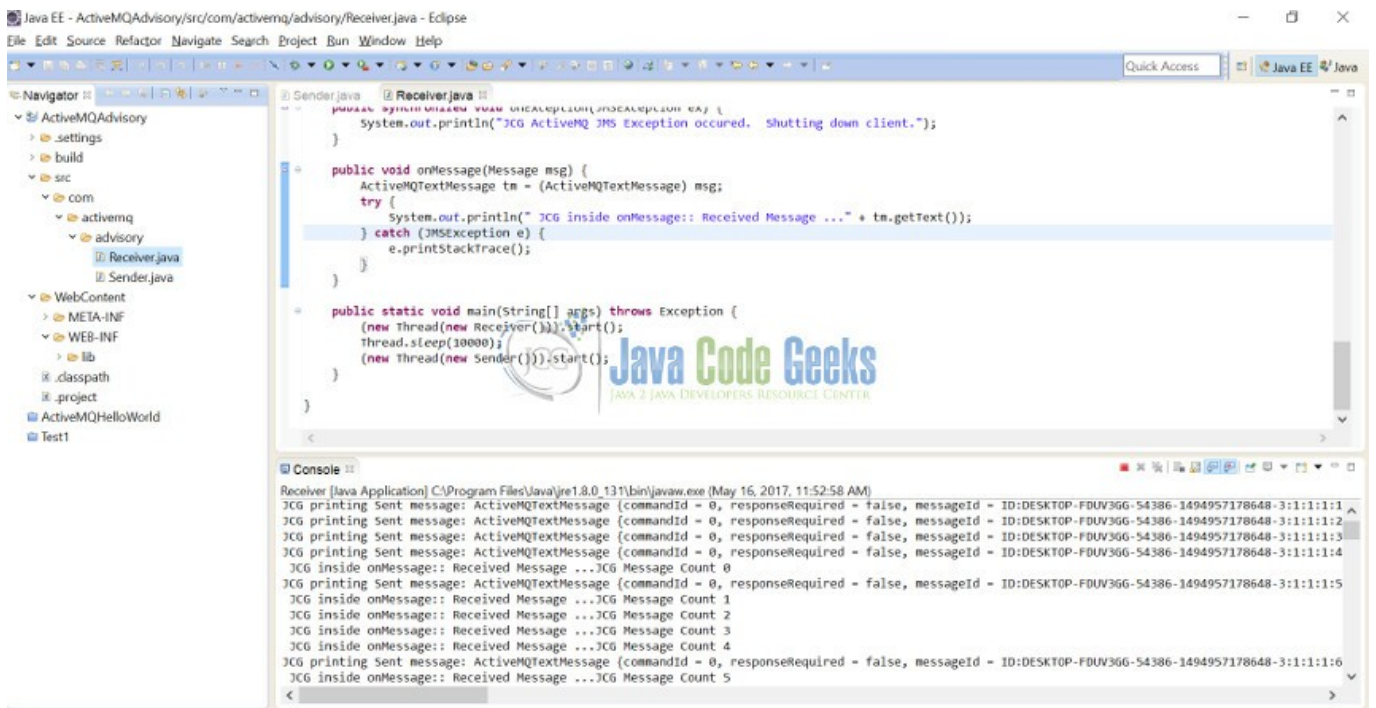


Figure 8.1: Eclipse console showing onMessage() being called

- We can also check our ActiveMQ console→Queues tab, to see the number of Pending/Enqueued/Dequeued messages in our queue after running the program



Figure 8.2: ActiveMQ Console

8.3.2 Example 2 - Monitoring ActiveMQ for a number of events (example using Eclipse)

- In this example we will use two classes AdvisorySrc and AdvisoryConsumer

- Using `AdvisorySrc`, the broker will produce Advisory messages
- `AdvisoryConsumer` demonstrates how we can implement those broker advisories
- The ActiveMQ broker generates advisory messages for a number of different events that occur on the broker
- The client applications can subscribe to special topics where the events are sent in order to monitor activity on the broker
- The advisory messages are nothing but simple JMS message objects that have some properties to provide event related information
- The `AdvisoryConsumer` class that we have written listens for events related to `MessageProducer` and `MessageConsumer`, being added and removed from the broker for a particular destination
- In this example, we will be watching a queue named `JCG_QUEUE`
- Let us now create a dynamic web project in eclipse and create our `AdvisorySrc` and `AdvisoryConsumer` classes. Please refer the code below

AdvisorySrc.java

```
package com.activemq.advisory;

import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;

public class AdvisorySrc implements Runnable {

    private final String connectionUri = "tcp://localhost:61616";
    private ActiveMQConnectionFactory connectionFactory;
    private Connection connection;
    private Session session;
    private Destination destination;
    private final Random rand = new Random();

    public void run() {
        try {
            connectionFactory = new ActiveMQConnectionFactory(connectionUri);
            connection = connectionFactory.createConnection();
            connection.start();
            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            destination = session.createQueue("JCG_QUEUE");

            TimeUnit.SECONDS.sleep(rand.nextInt(10));

            MessageProducer producer = session.createProducer(destination);
            producer.send(session.createTextMessage());

            TimeUnit.SECONDS.sleep(rand.nextInt(10));

            MessageConsumer consumer = session.createConsumer(destination);
            consumer.receive();
        }
    }
}
```

```
        TimeUnit.SECONDS.sleep(rand.nextInt(30));

        System.out.print(".");

        if (connection != null) {
            connection.close();
        }

    } catch (Exception ex) {}
}

public static void main(String[] args) {
    System.out.println("Starting Advisory Message Source !!!!");
    try {
        ExecutorService service = Executors.newFixedThreadPool(10);
        for (int i = 0; i < 20; ++i) {
            service.execute(new AdvisorySrc());
        }
        service.shutdown();
        service.awaitTermination(5, TimeUnit.MINUTES);
        System.out.println();

    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    System.out.println("Finished running the Advisory Message Source");
}
}
```

AdvisoryConsumer.java

```
package com.activemq.advisory;

import java.util.concurrent.TimeUnit;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.advisory.AdvisorySupport;

public class AdvisoryConsumer implements MessageListener {

    private final String connectionUri = "tcp://localhost:61616";
    private ActiveMQConnectionFactory connectionFactory;
    private Connection connection;
    private Session session;
    private Destination destination;
    private MessageConsumer advisoryConsumer;
    private Destination monitored;

    public void before() throws Exception {
        connectionFactory = new ActiveMQConnectionFactory(connectionUri);
        connection = connectionFactory.createConnection();
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        monitored = session.createQueue("JCG_QUEUE");
        destination = session.createTopic(
```

```

        AdvisorySupport.getConsumerAdvisoryTopic(monitored).getPhysicalName() + "," +
        AdvisorySupport.getProducerAdvisoryTopic(monitored).getPhysicalName());
        advisoryConsumer = session.createConsumer(destination);
        advisoryConsumer.setMessageListener(this);
        connection.start();
    }

    public void after() throws Exception {
        if (connection != null) {
            connection.close();
        }
    }

    public void onMessage(Message message) {
        try {
            Destination source = message.getJMSDestination();
            if (source.equals(AdvisorySupport.getConsumerAdvisoryTopic(monitored))) {
                int consumerCount = message.getIntProperty("consumerCount");
                System.out.println("New Consumer Advisory, Consumer Count: " + consumerCount);
            } else if (source.equals(AdvisorySupport.getProducerAdvisoryTopic(monitored))) {
                int producerCount = message.getIntProperty("producerCount");
                System.out.println("New Producer Advisory, Producer Count: " + producerCount);
            }
        } catch (JMSEException e) {
        }
    }

    public void run() throws Exception {
        TimeUnit.MINUTES.sleep(10);
    }

    public static void main(String[] args) {
        AdvisoryConsumer example = new AdvisoryConsumer();
        System.out.println("Starting Advisory Consumer example now...");
        try {
            example.before();
            example.run();
            example.after();
        } catch (Exception e) {
            System.out.println("Caught an exception during the example: " + e.getMessage());
        }
        System.out.println("Finished running the Advisory Consumer example.");
    }
}

```

Output:

ActiveMQ must be running. Next, we will be running our `AdvisorySrc` and `AdvisoryConsumer` classes to see how the `onMessage()` method is invoked.

Please follow the steps below:

- In the eclipse Right Click on `AdvisoryConsumer.java` → `Run As` → `Java Application`. We can see the main method printing the message

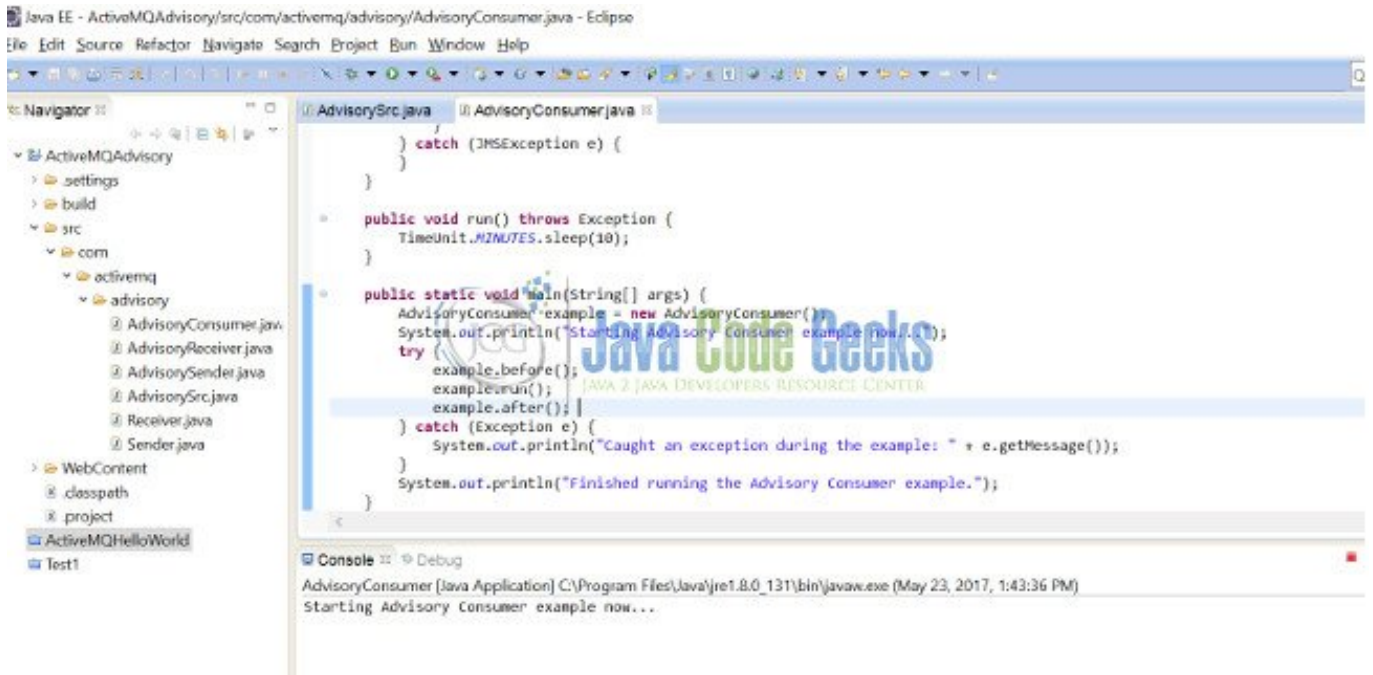


Figure 8.3: Eclipse output console

- In the eclipse Right Click on `AdvisorySrc.java` → Run As→Java Application. When you run `AdvisorySrc`, check eclipse console for both `AdvisorySrc` and `AdvisoryConsumer` classes. We can see output like the following that indicates the application is receiving advisory messages from the broker

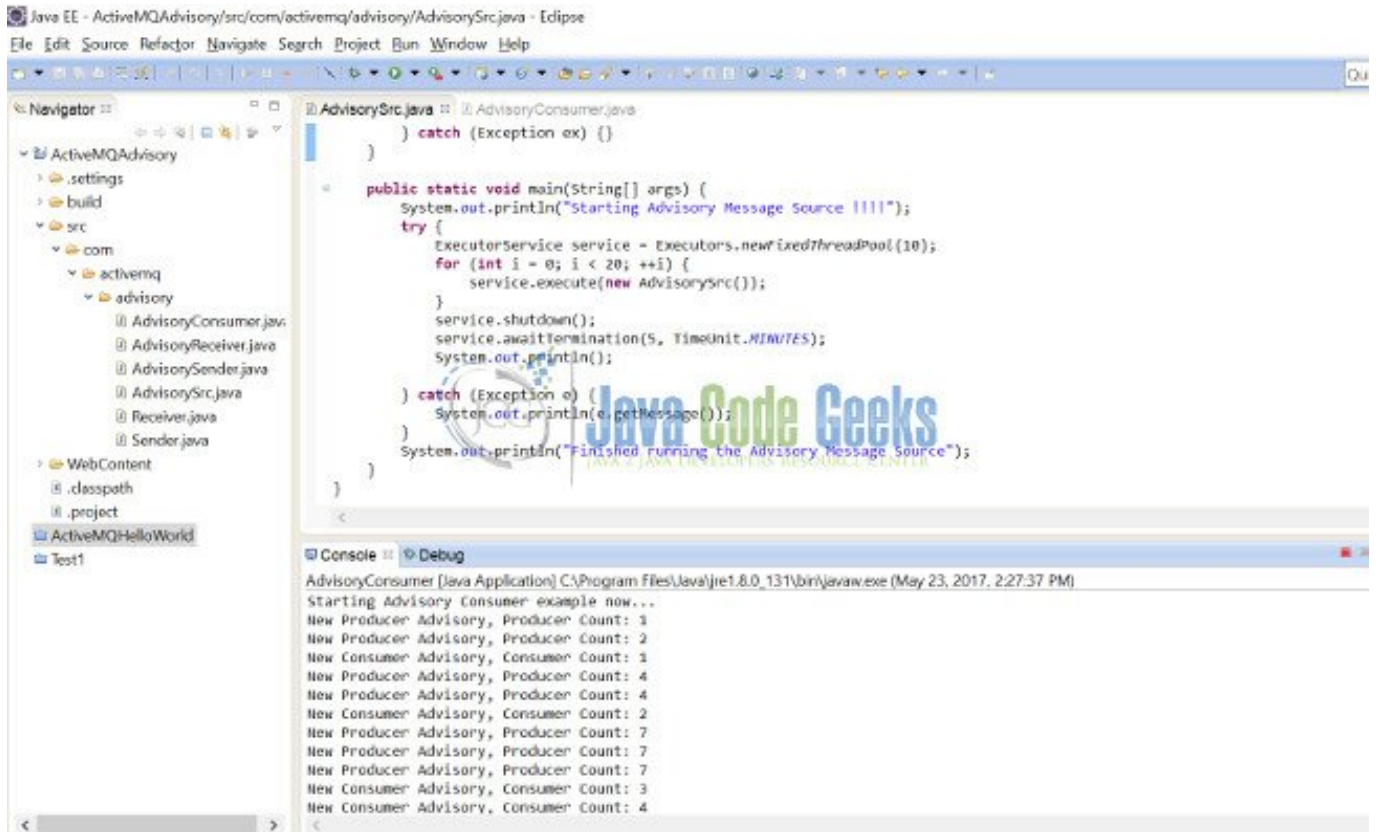


Figure 8.4: Eclipse console showing appln receiving msgs from broker

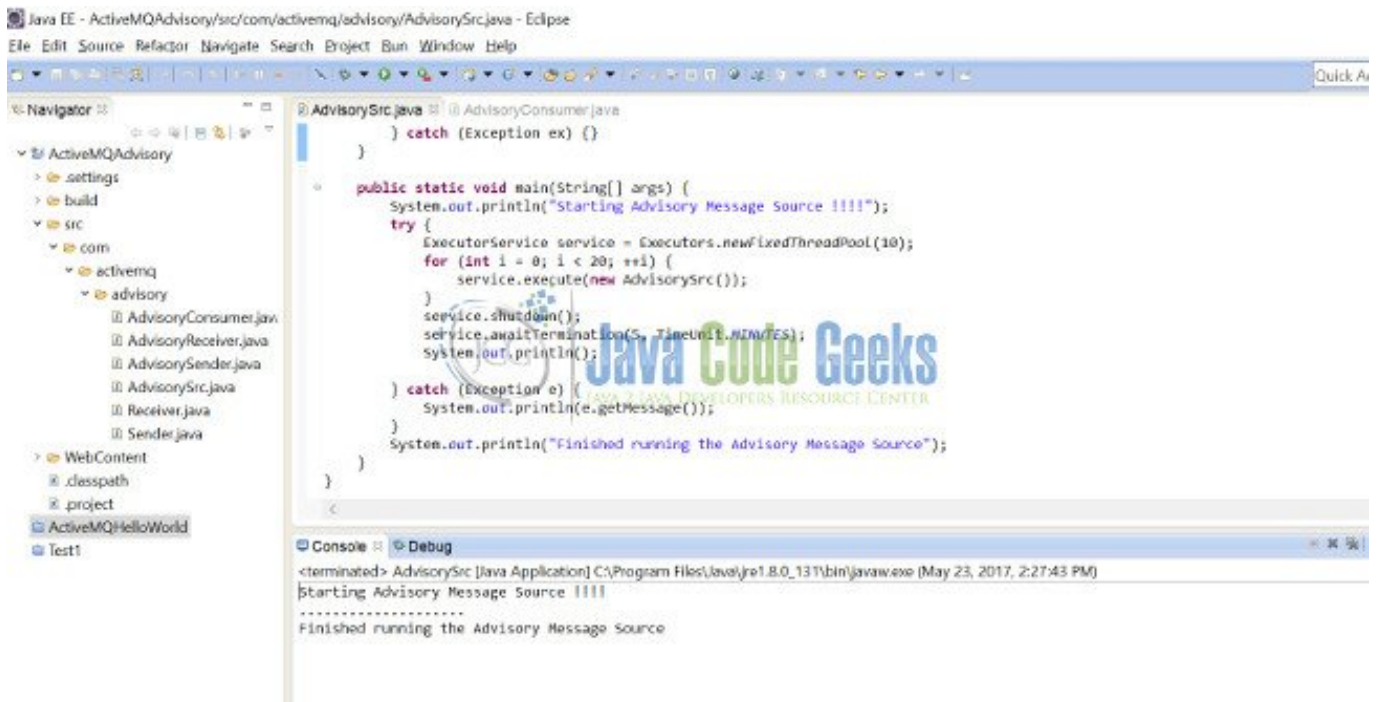


Figure 8.5: Eclipse console for AdvisorySrc.java

As you see the `AdvisoryConsumer` class, we are subscribing to two different topics on the broker (ActiveMQ). The topics for subscribing are **ConsumerAdvisoryTopic** and **ProducerAdvisoryTopic**. ActiveMQ client library provides a convenience class `AdvisorySupport` for fetching the various advisory topics.

- We can also check our ActiveMQ console→Queues tab and ActiveMQ console→Topics tab, to see the number of Pending/Enqueued/Dequeued messages in our queue and the topics our client has subscribed to, after running the program. Refer the screenshots below



Figure 8.6: ActiveMQ Queues tab

Name	Number of Consumers	Messages Enqueued	Messages Dequeued	Operations
ActiveMQ.Advisory.Connection	0	42	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Consumer.Queue.JCG_QUEUE	0	40	40	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.MessageDelivered.Queue.JCG_QUEUE	0	20	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Producer.Queue.JCG_QUEUE	0	40	40	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Queue	0	2	0	Send To Active Subscribers Active Producers Delete

Figure 8.7: ActiveMQ Topics subscribed

8.4 Conclusion

Through this example, we have learned how to configure ActiveMQ Advisory messages and use those messages as some kind of notification when a message is consumed by the consumer. We have also seen how a client application can subscribe to different topics on the broker.

Chapter 9

ActiveMQ File Transfer Example

9.1 Introduction

Apache ActiveMQ (AMQ) is a **message broker** which transfers messages from sender to receiver.

In this example, I will build two simple AMQ applications which will transfer files from one location to another:

- A producer sends a file via `BytesMessage` or `BlobMessage`
- A consumer receives the `BytesMessage` or `BlobMessage` and saves it as a file

9.2 JMS Message Type

JMS defines six different message types.

AMQ has deprecated `StreamMessage` and added `BlobMessage`. Message types:

Message Type	Contents	Purpose
<code>TextMessage</code>	A <code>java.lang.String</code> object	Exchanges simple text messages, such as <code>XML</code> and <code>Json</code>
<code>MapMessage</code>	A set of name-value pairs, with names as <code>String</code> objects and values as primitive types in the Java programming language.	Exchanges key-value data
<code>ObjectMessage</code>	A <code>Serializable</code> object in the Java programming language.	Exchanges Java objects.
<code>StreamMessage</code>	A stream of primitive values in the Java programming language, filled and read sequentially.	Deprecated within AMQ.
<code>BytesMessage</code>	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.	Exchanges data in a format that is native to the application, and when JMS is used as a transport between two systems, where the JMS client does not know the message payload type.
<code>BlobMessage</code>	Binary Large Object (BLOB).	Added by AMQ.

Figure 9.1: Message Types

In this example, I will demonstrate how to transfer a file via `BytesMessage` and `BlobMessage`.

9.3 Business Use Case

Businesses exchange information by transferring files from one location to another. In this example, it transfers nine files from C:\temp\input to C:\temp\output . These nine files belong to commonly used file types.

The image below shows file details as well as the output directory details.

```
C:\temp>dir input
Volume in drive C is OSDisk
Volume Serial Number is 3A10-C6D4

Directory of C:\temp\input

01/11/2018  09:26 AM    <DIR>          .
01/11/2018  09:26 AM    <DIR>          ..
08/23/2016  12:18 PM             126,025 10-18 Year Swim Lessons- SUMMER.docx
10/17/2017  12:17 PM             70,971 2017_18 _Schedule_chess.pdf
12/28/2017  03:52 PM             10,932 activemq-monitor-demo.zip
01/05/2018  09:49 AM             98,816 activeMQ.vsd
01/24/2017  11:10 AM             32,971 JVM_memory.PNG
01/09/2018  09:16 PM              112 site_cag.txt
01/10/2018  10:12 AM             8,761 test.xlsx
06/07/2017  09:36 AM              49 test2.ppt
07/13/2009  11:32 PM             620,888 Tulips.jpg
          9 File(s)          969,525 bytes
          2 Dir(s)    3,363,618,816 bytes free

C:\temp>tree
Folder PATH listing for volume OSDisk
Volume serial number is 3A10-C6D4
C:.
├── input
├── output
│   ├── blob
│   └── bytes
C:\temp>
```





Figure 9.2: Input files

9.4 File Transfer Application

9.4.1 Technologies Used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Apache ActiveMQ 5.15.0 (others will do fine)
- Eclipse Neon (Any Java IDE would work)

9.4.2 Dependency

Add dependency to Maven pom.xml.

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-all</artifactId>
    <version>5.15.0</version>
  </dependency>

  <dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.5</version>
  </dependency>
</dependencies>
```

9.4.3 Constants

There are eight constants value used in this example.

Constants.java

```
package jcg.demo;

/**
 * The constants for this demo.
 *
 * @author Mary.Zheng
 *
 */
public class Constants {
  public static final String FILE_INPUT_DIRECTORY = "C:\\temp\\input";
  public static final String FILE_NAME = "fileName";

  public static final String FILE_OUTPUT_BYTE_DIRECTORY = "C:\\temp\\output\\bytes\\" ←
  ;
  public static final String FILE_OUTPUT_BLOB_DIRECTORY = "C:\\temp\\output\\blob\\";

  public static final String TEST_QUEUE = "test.queue";
  public static final String TEST_BROKER_URL = "tcp://localhost:61716";
  public static final String ADMIN = "admin";

  public static final String BLOB_FILESERVER = "?jms.blobTransferPolicy. ←
  defaultUploadUrl=https://localhost:8761/fileserver/";
}
```

- Line 20: AMQ BlobMessage requires a file server

9.4.4 File Manager

Create a file manager to read and write a file via bytes array.

FileAsByteArrayManager.java

```
package jcg.demo.file;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

public class FileAsByteArrayManager {

    public byte[] readfileAsBytes(File file) throws IOException {
        try (RandomAccessFile accessFile = new RandomAccessFile(file, "r")) {
            byte[] bytes = new byte[(int) accessFile.length()];
            accessFile.readFully(bytes);
            return bytes;
        }
    }

    public void writeFile(byte[] bytes, String fileName) throws IOException {
        File file = new File(fileName);
        try (RandomAccessFile accessFile = new RandomAccessFile(file, "rw")) {
            accessFile.write(bytes);
        }
    }
}
```

9.4.5 Save File Application

Create a Java application which receives the messages and save them to `c:\temp\output\` with the same file name as before.

ConsumeFileApp.java

```
package jcg.demo;

import javax.jms.JMSEException;

import jcg.demo.activemq.QueueMessageConsumer;

public class ConsumeFileApp {

    public static void main(String[] args) {

        QueueMessageConsumer queueMsgListener = new QueueMessageConsumer(Constants. ←
            TEST_BROKER_URL, Constants.ADMIN,
            Constants.ADMIN);
        queueMsgListener.setDestinationName(Constants.TEST_QUEUE);

        try {
            queueMsgListener.run();
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}
```

9.4.6 Message Consumer

Create the class `QueueMessageConsumer`.

QueueMessageConsumer.java

```
package jcg.demo.activemq;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.time.Duration;
import java.time.Instant;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQBlobMessage;
import org.apache.activemq.command.ActiveMQBytesMessage;
import org.apache.activemq.command.ActiveMQTextMessage;
import org.apache.commons.io.IOUtils;

import jcg.demo.Constants;
import jcg.demo.file.FileAsByteArrayManager;

/**
 * A message consumer which consumes the message from ActiveMQ Broker
 *
 * @author Mary.Zheng
 *
 */
public class QueueMessageConsumer implements MessageListener {

    private String activeMqBrokerUri;
    private String username;
    private String password;
    private String destinationName;
    private FileAsByteArrayManager fileManager = new FileAsByteArrayManager();

    public QueueMessageConsumer(String activeMqBrokerUri, String username, String ←
password) {
        super();
        this.activeMqBrokerUri = activeMqBrokerUri;
        this.username = username;
        this.password = password;
    }

    public void run() throws JMSEException {
        ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(username, ←
password, activeMqBrokerUri);
        Connection connection = factory.createConnection();
        connection.start();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE) ←
;

        Destination destination = session.createQueue(destinationName);

        MessageConsumer consumer = session.createConsumer(destination);
        consumer.setMessageListener(this);

        System.out.println(String.format("QueueMessageConsumer Waiting for messages ←
```

```

        at queue='%s' broker='%s'",
            destinationName, this.activeMqBrokerUri));
    }

    @Override
    public void onMessage(Message message) {

        try {
            String filename = message.getStringProperty(Constants.FILE_NAME);

            Instant start = Instant.now();

            if (message instanceof ActiveMQTextMessage) {
                handleTextMessage((ActiveMQTextMessage) message);
            } else if (message instanceof ActiveMQBlobMessage) {
                handleBlobMessage((ActiveMQBlobMessage) message, filename);
            } else if (message instanceof ActiveMQBytesMessage) {
                handleBytesMessage((ActiveMQBytesMessage) message, filename ←
            );
            } else {
                System.out.println("test");
            }

            Instant end = Instant.now();
            System.out
                .println("Consumed message with filename [" + ←
                    filename + "], took " + Duration.between(start, ←
                    end));

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void handleBytesMessage(ActiveMQBytesMessage bytesMessage, String filename)
        throws IOException, JMSEException {
        String outputFileName = Constants.FILE_OUTPUT_BYTE_DIRECTORY + filename;
        fileManager.writeFile(bytesMessage.getContent().getData(), outputFileName);
        System.out.println("Received ActiveMQBytesMessage message");
    }

    private void handleBlobMessage(ActiveMQBlobMessage blobMessage, String filename)
        throws FileNotFoundException, IOException, JMSEException {
        // for 1mb or bigger message
        String outputFileName = Constants.FILE_OUTPUT_BLOB_DIRECTORY + filename;
        InputStream in = blobMessage.getInputStream();
        fileManager.writeFile(IUtils.toByteArray(in), outputFileName);
        System.out.println("Received ActiveMQBlobMessage message");
    }

    private void handleTextMessage(ActiveMQTextMessage txtMessage) throws JMSEException ←
    {
        String msg = String.format("Received ActiveMQTextMessage [ %s ]", ←
            txtMessage.getText());
        System.out.println(msg);
    }

    public String getDestinationName() {
        return destinationName;
    }

    public void setDestinationName(String destinationName) {

```

```

        this.destinationName = destinationName;
    }
}

```

- Line 73: Receives the file content as BlobMessage
- Line 75: Receives the file content as BytesMessage

9.4.7 Send File Application

Create a Java application to send nine files at C:\temp\input as either BytesMessage or BlobMessage to AMQ.

SendFileApp.java

```

package jcg.demo;

import java.util.Scanner;

import jcg.demo.activemq.QueueMessageProducer;

public class SendFileApp {

    public static void main(String[] args) {
        try {

            QueueMessageProducer queProducer = new QueueMessageProducer( ←
                Constants.TEST_BROKER_URL, Constants.ADMIN,
                Constants.ADMIN);

            System.out.println("Enter message type for transferring file:"
                + "\n\t1 - File as BytesMessage \n\t2 - File as ←
                BlobMessage");
            try (Scanner scanIn = new Scanner(System.in)) {
                String inputFileType = scanIn.nextLine();
                switch (inputFileType) {
                    case "1":
                        queProducer.sendBytesMessages(Constants.TEST_QUEUE) ←
                            ;
                        break;
                    case "2":
                        queProducer.sendBlobMessages(Constants.TEST_QUEUE);
                        break;
                    default:
                        System.out.println("Wrong input");
                }
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

9.4.8 Message Producer

Create a class QueueMessageProducer .

QueueMessageProducer.java

```
package jcg.demo.activemq;

import java.io.File;
import java.io.IOException;
import java.time.Duration;
import java.time.Instant;

import javax.jms.BytesMessage;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.StreamMessage;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.ActiveMQSession;
import org.apache.activemq.BlobMessage;

import jcg.demo.Constants;
import jcg.demo.file.FileAsByteArrayManager;

/**
 * A message producer which sends the file message to ActiveMQ Broker
 *
 * @author Mary.Zheng
 *
 */
public class QueueMessageProducer {

    private String activeMqBrokerUri;
    private String username;
    private String password;

    private ActiveMQSession session;
    private MessageProducer msgProducer;
    private ConnectionFactory connFactory;
    private Connection connection;

    private FileAsByteArrayManager fileManager = new FileAsByteArrayManager();

    public QueueMessageProducer(String activeMqBrokerUri, String username, String ←
        password) {
        super();
        this.activeMqBrokerUri = activeMqBrokerUri;
        this.username = username;
        this.password = password;
    }

    private void setup() throws JMSEException {
        connFactory = new ActiveMQConnectionFactory(username, password, ←
            activeMqBrokerUri);
        connection = connFactory.createConnection();
        connection.start();
        session = (ActiveMQSession) connection.createSession(false, Session. ←
            AUTO_ACKNOWLEDGE);
    }

    private void close() {
        try {
            if (msgProducer != null) {
```



```
        msgProducer.close();
    }
    if (session != null) {
        session.close();
    }
    if (connection != null) {
        connection.close();
    }
} catch (Throwable ignore) {
}
}

public void sendBytesMessages(String queueName) throws JMSEException, IOException {
    setup();

    msgProducer = session.createProducer(session.createQueue(queueName));

    File[] files = new File(Constants.FILE_INPUT_DIRECTORY).listFiles();
    for (File file : files) {
        if (file.isFile()) {
            sendFileAsBytesMessage(file);
        }
    }

    close();
}

public void sendBlobMessages(String queueName) throws JMSEException {
    this.activeMqBrokerUri = activeMqBrokerUri + Constants.BLOB_FILESERVER;
    setup();

    msgProducer = session.createProducer(session.createQueue(queueName));

    File[] files = new File(Constants.FILE_INPUT_DIRECTORY).listFiles();
    for (File file : files) {
        if (file.isFile()) {
            sendFileAsBlobMessage(file);
        }
    }

    close();
}

private void sendFileAsBlobMessage(File file) throws JMSEException {
    Instant start = Instant.now();
    BlobMessage blobMessage = session.createBlobMessage(file);
    blobMessage.setStringProperty(Constants.FILE_NAME, file.getName());
    msgProducer.send(blobMessage);
    Instant end = Instant.now();
    System.out.println("sendFileAsBlobMessage for [" + file.getName() + "], ←
        took " + Duration.between(start, end));
}

private void sendFileAsBytesMessage(File file) throws JMSEException, IOException {
    Instant start = Instant.now();
    BytesMessage bytesMessage = session.createBytesMessage();
    bytesMessage.setStringProperty(Constants.FILE_NAME, file.getName());
    bytesMessage.writeBytes(fileManager.readFileAsBytes(file));
    msgProducer.send(bytesMessage);
    Instant end = Instant.now();
}
```

```

        System.out.println("sendFileAsBytesMessage for [" + file.getName() + "], ←
            took " + Duration.between(start, end));
    }
}

```

- Line 89: AMQ BlobMessage requires a file server

9.5 Demo Time

9.5.1 Start ConsumeFileApp

Start the ConsumeFileApp .

ConsumeFileApp Output

```

QueueMessageConsumer Waiting for messages at queue='test.queue' broker='tcp://localhost ←
:61716'

```

9.5.2 Start SendFileApp

Run SendFileApp with BytesMessage

Send BytesMessage Output

```

Enter message type for transferring file:
    1 - File as BytesMessage
    2 - File as BlobMessage
1
sendFileAsBytesMessage for [10-18 Year Swim Lessons- SUMMER.docx], took PT0.02S
sendFileAsBytesMessage for [2017_18 _Schedule_chess.pdf], took PT0.009S
sendFileAsBytesMessage for [activemq-monitor-demo.zip], took PT0.008S
sendFileAsBytesMessage for [activeMQ.vsd], took PT0.01S
sendFileAsBytesMessage for [JVM_memory.PNG], took PT0.008S
sendFileAsBytesMessage for [site_cag.txt], took PT0.006S
sendFileAsBytesMessage for [test.xlsx], took PT0.009S
sendFileAsBytesMessage for [test2.ppt], took PT0.008S
sendFileAsBytesMessage for [Tulips.jpg], took PT0.018S

```

The ConsumeFileApp output:

ConsumeFileApp Output

```

QueueMessageConsumer Waiting for messages at queue='test.queue' broker='tcp://localhost ←
:61716'
Received ActiveMQBytesMessage message
Consumed message with filename [10-18 Year Swim Lessons- SUMMER.docx], took PT0.002S
Received ActiveMQBytesMessage message
Consumed message with filename [2017_18 _Schedule_chess.pdf], took PT0.002S
Received ActiveMQBytesMessage message
Consumed message with filename [activemq-monitor-demo.zip], took PT0.001S
Received ActiveMQBytesMessage message
Consumed message with filename [activeMQ.vsd], took PT0.001S
Received ActiveMQBytesMessage message
Consumed message with filename [JVM_memory.PNG], took PT0.002S
Received ActiveMQBytesMessage message
Consumed message with filename [site_cag.txt], took PT0.001S
Received ActiveMQBytesMessage message
Consumed message with filename [test.xlsx], took PT0.001S

```

```
Received ActiveMQBytesMessage message
Consumed message with filename [test2.ppt], took PT0.001S
Received ActiveMQBytesMessage message
Consumed message with filename [Tulips.jpg], took PT0.004S
```

Run SendFileApp for BlobMessage

```
Enter message type for transferring file:
    1 - File as BytesMessage
    2 - File as BlobMessage
2
sendFileAsBlobMessage for [10-18 Year Swim Lessons- SUMMER.docx], took PT0.048S
sendFileAsBlobMessage for [2017_18 _Schedule_chess.pdf], took PT0.021S
sendFileAsBlobMessage for [activemq-monitor-demo.zip], took PT0.01S
sendFileAsBlobMessage for [activeMQ.vsd], took PT0.02S
sendFileAsBlobMessage for [JVM_memory.PNG], took PT0.012S
sendFileAsBlobMessage for [site_cag.txt], took PT0.011S
sendFileAsBlobMessage for [test.xlsx], took PT0.015S
sendFileAsBlobMessage for [test2.ppt], took PT0.012S
sendFileAsBlobMessage for [Tulips.jpg], took PT0.029S
```

- **Line 2:** BlobMessage took longer (28 ms) than BytesMessage to send 10-18 Year Swim Lessons-SUMMER.docx

The ConsumeFileApp output:

ConsumeFileApp Output

```
Received ActiveMQBlobMessage message
Consumed message with filename [10-18 Year Swim Lessons- SUMMER.docx], took PT0.044S
Received ActiveMQBlobMessage message
Consumed message with filename [2017_18 _Schedule_chess.pdf], took PT0.011S
Received ActiveMQBlobMessage message
Consumed message with filename [activemq-monitor-demo.zip], took PT0.007S
Received ActiveMQBlobMessage message
Consumed message with filename [activeMQ.vsd], took PT0.01S
Received ActiveMQBlobMessage message
Consumed message with filename [JVM_memory.PNG], took PT0.006S
Received ActiveMQBlobMessage message
Consumed message with filename [site_cag.txt], took PT0.005S
Received ActiveMQBlobMessage message
Consumed message with filename [test.xlsx], took PT0.006S
Received ActiveMQBlobMessage message
Consumed message with filename [test2.ppt], took PT0.005S
Received ActiveMQBlobMessage message
Consumed message with filename [Tulips.jpg], took PT0.021S
```

- **Line 2:** BlobMessage took longer (42 ms) than BytesMessage to save the 10-18 Year Swim Lessons-SUMMER.docx

9.5.3 Verify the Transferred Files

Check the files at the output directory. The files at C:\temp\blob\ and C:\temp\bytes\ are the same as the ones from the C:\temp\input directory.

The image below shows the saved files at blob directory. Click to open and compare to the input files.

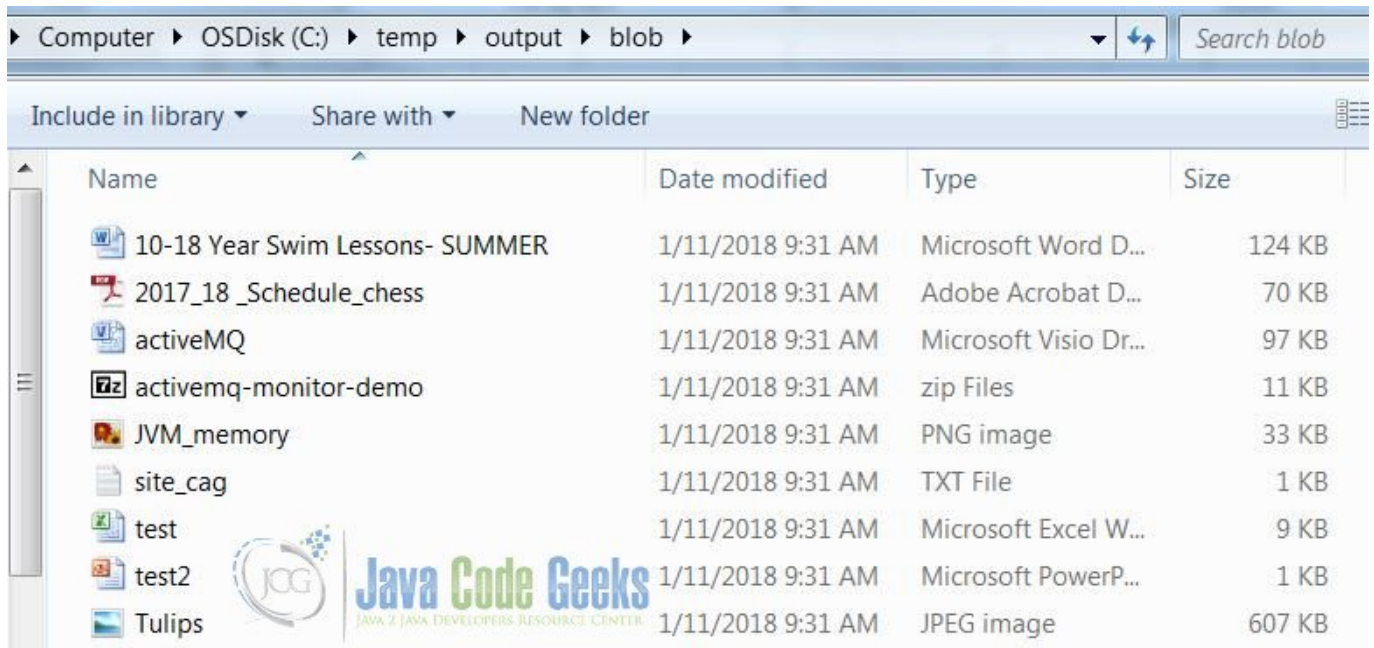


Figure 9.3: Files transferred via BlobMessage

9.6 Summary

In this example, I built two Java AMQ client applications:

- `SendFileApp` sends the file to the AMQ via `ByteMessage` and `BlobMessage`
- `ConsumeFileApp` receives the message from the AMQ and saves it to a different location

I compared the total time to send and receive two different message types and found out that `ByteMessage` is faster than `BlobMessage`.

9.7 Download the Source Code

This example consists of two applications to send and receive file data based on the `ByteMessage` and `BlobMessage`. You can download the full source code of this example here: [Apache ActiveMQ File Transfer Example](#)