Java Insider

Learning by sharing…

Type text to search here...

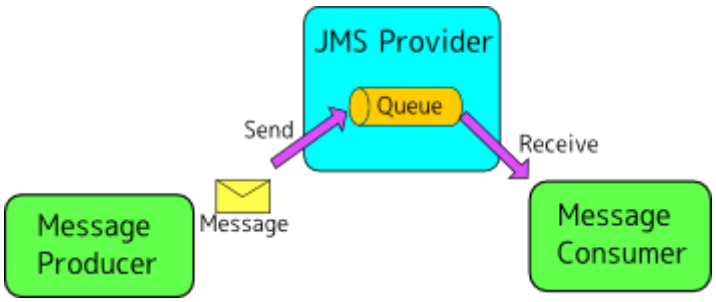
Archive

Posts Tagged ‘JMS with ActiveMQ Sample example’

Simple guide to Java Message Service (JMS) using ActiveMQ

September 25, 2012 Rudra Narayan Garnaik 7 comments

JMS let’s you send messages containing for example a String, array of bytes or a serializable Java object, from one program to another. It doesn’t however use a direct connection from program A to program B, instead the message is sent to a JMS provider and put there in a Queue where it waits until the other program receives it.



MessageProducer is a Java program sending a JMS message to a Queue on the JMS Provider. MessageConsumer is another program which receives that message. These two programs can run on separate machines and all they have to know to communicate is the URL of the JMS Provider. The Provider can be for example a Java EE server, like JBoss or Glassfish. But don’t be afraid, you don’t need a full-blown JEE server to send a JMS message. In this article we will use ActiveMQ which is lightweight and easy to use.

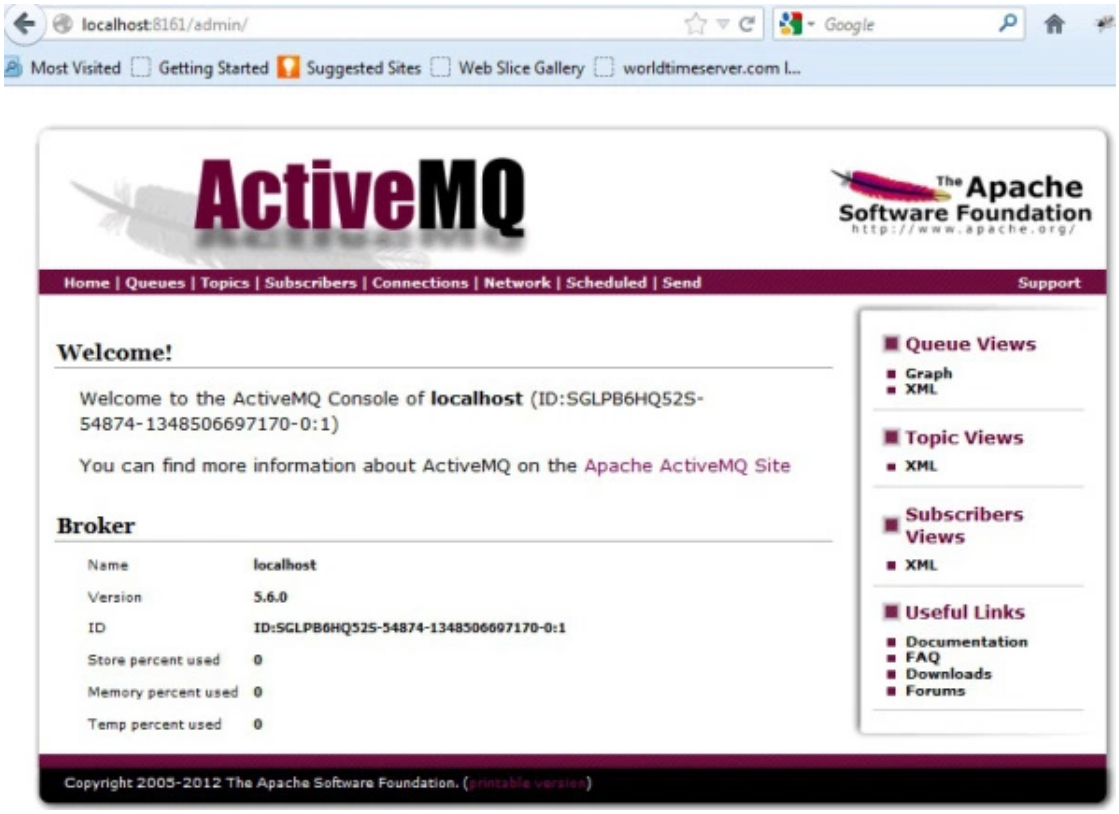
First we need to download ActiveMQ. If you are using Linux, you can get it from this link. For Windows you can use this link. In case the links don’t work, you can find the files in ‘Downloads’ section on ActiveMQ’s webpage.

After the download, extract it to any directory and run the ‘activemq’ program from beneath the ‘{path-where-you-extracted-activemq}/bin’ directory:

You should see a bunch of INFO messages appearing on the terminal:

Now the ActiveMQ server is up and running. You can close it any time by pressing Ctrl-C. ActiveMQ has a nice admin console, where you can see a lot of useful informations and change the settings:

http://localhost:8161/admin/.



Now that we have a JMS provider running, let's write our message producer and consumer programs. For that, you will need to put the ActiveMQ's JAR file on the class path. The file you need is called (for version 5.6.0) 'activemq-all-5.6.0.jar' or something similar and is in the extracted ActiveMQ directory. In Eclipse you could click right mouse button on your project and choose Properties->Java Build Path->Libraries->Add External Library.

Here is the code of the program sending (producing) the messages:

```
package com.vallysoft.activemq.test;

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 Producer {
// 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 = "VALLYSOFTQ"; //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();
// JMS messages are sent and received using a Session. We will
```

```
// create here a non-transactional session object. If you want
// to use transactions you should set the first parameter to 'true'
Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
// Destination represents here our queue 'VALLYSOFTQ' on the
// JMS server. You don't have to do anything special on the
// server to create it, it will be created automatically.
Destination destination = session.createQueue(subject);
// MessageProducer is used for sending messages (as opposed
// to MessageConsumer which is used for receiving them)
MessageProducer producer = session.createProducer(destination);
// We will send a small text message saying 'Hello' in Japanese
TextMessage message = session.createTextMessage("Hello welcome come to vallysoft ActiveMQ!");
// Here we are sending the message!
producer.send(message);
System.out.println("Sentage '" + message.getText() + "'");

connection.close();
}
}
```

There is a lot going on here. The Connection represents our connection with the JMS Provider – ActiveMQ. Be sure not to confuse it with SQL's Connection. 'Destination' represents the Queue on the JMS Provider that we will be sending messages to. In our case, we will send it to Queue called 'VALLYSOFTQ' (it will be automatically created if it didn't exist yet).

What you should note is that there is no mention of who will finally read the message. Actually, the Producer does not know where or who the consumer is! We are just sending messages into queue 'VALLYSOFTQ' and what happens from there to the sent messages is not of Producer's interest any more.

Now let's see how to receive (consume) the sent message. Here is the code for the Consumer class:

```
package com.vallysoft.activemq.test;

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;

/**
* @author RudraG
*
*/
public class Consumer {
// 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 = "VALLYSOFTQ";

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 seding messages
Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

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

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

// Here we receive the message.
// By default this call is blocking, which means it will wait
// for a message to arrive on the queue.
Message message = consumer.receive();

// There are many types of Message and TextMessage
// is just one of them. Producer 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("Receivedage '" + textMessage.getText()
+ "'");
}
connection.close();
}
}
```

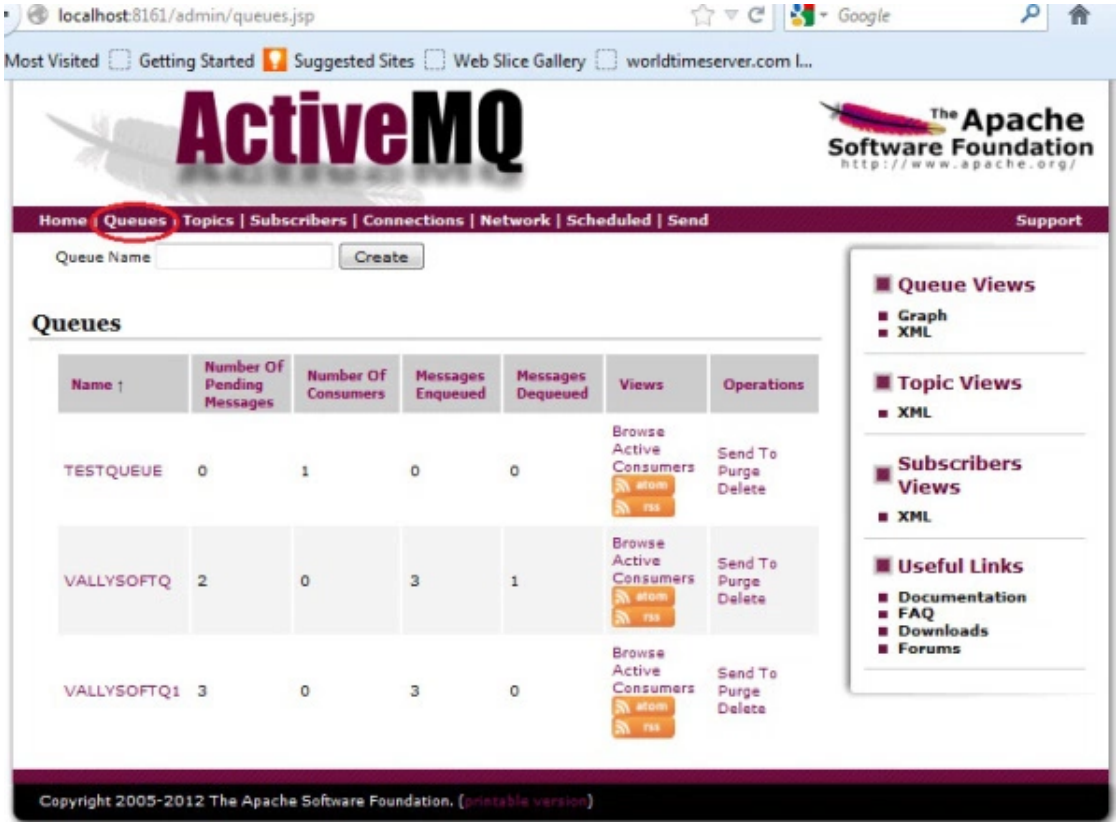
As you see, it looks pretty similar to the Producer's code before. Actually only the part starting from line 35 is substantially different. We produce there a MessageConsumer instead of MessageReceiver and then use it's .receive() method instead of .send(). You can see also an ugly cast from Message to TextMessage but there is nothing we could do about it, because .receive() method just returns interface Message (TextMessage interface extends Message) and there are no separate methods for receiving just TextMessage's.

Compile now both programs remembering about adding ActiveMQ's JAR file to the classpath. Before running them be also sure that the ActiveMQ's instance is running (for example in a separate terminal). First run the Producer program:

```
Successfully connected to tcp://localhost:61616
Sent message : Hello welcome come to vallysoft ActiveMQ!
```

If you see something similar to the output above (especially the 'Sent message' part) then it means that the message was successfully sent and is now inside the VALLYSOFTQ queue. You can enter the Queues section in the ActiveMQ's admin console :

http://localhost:8161/admin/queues.jsp



and see that there is one message sitting in VALLYSOFTQ:

In order to receive that message run now the Consumer program:

```
Successfully connected to tcp://localhost:61616
Received message : Hello welcome come to vallysoft ActiveMQ!
```

If you are getting above input (or something similar) everything went ok. The message was successfully received.

You are now probably thinking “Why would anybody want to do that??”. In fact, the code presented here to transfer just a small text message was pretty big, and you also needed an instance of ActiveMQ running, and dependencies in the classpath and all that…. Instead of using JMS we could use plain TCP/IP with few times less effort. So, what are good points of using JMS compared to simple TCP/IP or Java RMI? Here they are:

- Communication using JMS is **asynchronous**. The producer just sends a message and goes on with his business. If you called a method using RMI you would have to wait until it returns, and there can be cases you just don’t want to lose all that time.
- Take a look at how we run the example above. At the moment we run the Producer to send a message, there was yet no instance of Consumer running. The message was delivered at the moment the Consumer asked ActiveMQ for it. Now compare it to RMI. If we tried to send any request through RMI to a service that is not running yet, we would get a RemoteException. Using JMS let’s you send requests to services that may be currently unavailable. The message will be delivered as soon as the service starts.
- JMS is a way of abstracting the process of sending messages. As you see in the examples above, we are using some custom Queue with name ‘TESTQUEUE’. The producer just knows it has to send to that queue and the consumer takes it from there. Thanks to that we can decouple the producer from the consumer. When a message gets into the queue, we can do a full bunch of stuff with it, like sending it to

other queues, copying it, saving in a database, routing based on its contents and much more. here you can see some of the possibilities.

JMS is widely used as a System Integration solution in big, distributed systems like those of for example banks. There are many books dealing with this huge topic, for example Enterprise Integration Patterns. If you want to learn more about JMS itself you can do it for example on this JMS Tutorial on Sun's webpage.

Share this:

Categories: J2EE, Java Tags: ActiveMQ, ActiveMQ Example, JMS, JMS with ActiveMQ Sample example

RSS feed

Random Posts

- Apache Kafka API Code Sample in Java
- Interface Vs. Abstract class
- String Literals and String Pool
- MVP – Minimum Viable Product Strategy
- Welcome To Java Technical Blog

Tag Cloud

Abstract classes vs Interface ActiveMQ ActiveMQ Example all the best class Class Libraries code check in core java Decorator pattern design pattern in java Design Patterns Development Tools Embedded Java factory design pattern example Factory Method pattern in java factory pattern Graphical user interface Hash function HashMap HashSet Hashtable inheritance Interface and Abstract class Interface Vs. Abstract classes interview interview questions and answers Java Java class file Java Classloader java in embedded space Java Technical Blog Java Virtual Machine JMS JMS with ActiveMQ Sample example Languages Object (computer science) Programming questions Set (abstract data type) singleton singleton design pattern Singleton pattern source code Unified Modeling Language welcome

Categories

- ANDROID
- Cluster
- DBA
- Hibernate
- J2EE
- Java
- Java Design Pattern
- JBoss Richfaces
- JBoss Seam
- JDK7
- JSF
- LINUX
- Load Balancing
- messaging
- MVP
- NFC
- PL/SQL

- Product
- RFID
- Software Solutions
- Spring
- SQL
- Struts 2.0
- UBUNTU
- Uncategorized

Blogroll

- abhiurscute
- Discuss
- Get Inspired
- Get Polling
- Get Support
- Learn WordPress.com
- WordPress Planet
- WordPress.com News

Archives

- May 2017
- May 2016
- May 2014
- January 2014
- December 2013
- July 2013
- May 2013
- March 2013
- January 2013
- December 2012
- November 2012
- October 2012
- September 2012
- May 2012
- December 2011
- August 2011
- July 2011

Meta

- Register
- Log in

Top