# Table of Contents

<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. TUTORIAL – AO AND ACTIVEMQ QUEUES</td>
<td>2</td>
</tr>
<tr>
<td>1.1 REFERENCES</td>
<td>2</td>
</tr>
<tr>
<td>2. OVERVIEW</td>
<td>2</td>
</tr>
<tr>
<td>3. INSTALL ACTIVEMQ</td>
<td>5</td>
</tr>
<tr>
<td>4. VERIFY THE AO (CLIENT) ENVIRONMENT</td>
<td>6</td>
</tr>
<tr>
<td>5. JNDI PROPERTIES</td>
<td>7</td>
</tr>
<tr>
<td>6. AO ADAPTER CONFIGURATIONS</td>
<td>8</td>
</tr>
<tr>
<td>7. TESTING THE QUEUE</td>
<td>9</td>
</tr>
<tr>
<td>CONSUMER - MONITORING</td>
<td>10</td>
</tr>
<tr>
<td>PRODUCER - ACTOR</td>
<td>17</td>
</tr>
<tr>
<td>8. SYNOPSIS</td>
<td>19</td>
</tr>
</tbody>
</table>
1. Tutorial – AO and ActiveMQ Queues

In this tutorial we’ll standup an ActiveMQ instance and link AO into a Queue within that ActiveMQ environment.

1.1 References

- Title: ActiveMQ
  Author: N/A
  Referenced Revision: N/A

- Title: Tutorial - BMC Atrium Orchestrator JMS Monitor Adapter with Apache ActiveMQ
  Location: [https://communities.bmc.com/communities/docs/DOC-20005](https://communities.bmc.com/communities/docs/DOC-20005)
  Author: caropepe
  Referenced Revision: Jun 26, 2012 11:55 AM

2. Overview

ActiveMQ is an Open Source Message Broker solution. The underlying technology is JMS, so the protocol is somewhat proprietary, but in widespread use. The choice to use ActiveMQ over other JMS or AMPQ brokers is arbitrary. AO does come with out-of-box JMS workflows and adapters to help in the JMS choice.

Message Queues are common in Enterprise Service Bus architecture. They provide an excellent platform for distributed applications to communicate with one another. AO is an excellent platform to integrate Applications in Enterprise Service Bus architecture. At some point you will probably integrate AO with some form of Message Queue to provide larger scope Enterprise integrations. This tutorial discusses and documents using ActiveMQ Queues with AO.

I encourage you to work through the Tutorial by Caropepe in the References section. That tutorial provides an excellent and easy introduction to ActiveMQ and AO. The difference between the two tutorials is the caropepe tutorial uses Topics, this tutorial will focus on Queues.

Queues differ slightly from Topics. Queues are intended to provide data, in the form of a Message, from one application to another. The two applications don’t need to know any details about each other, just what type of message is being passed and how to assemble and disassemble that message for processing. Conceptually, messages within
a Queue are intended for a single Consumer. The messages are designed to pass/provide data from one application to another.

Message Queues: Designed for Application to Application Messaging

Topics are designed for the traditional Publish and Subscribe model. Messages are placed onto a Topic and ‘many’ Consumers Subscribe to that Topic, much like a stock quote system.

Message Topics: Designed for Publish and Subscribe models.
The environment being used for this tutorial is the following:

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
<th>Required?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO</td>
<td>7.6.02.04</td>
<td>✓</td>
<td>Atrium Orchestrator</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>5.6.0</td>
<td>✓</td>
<td>Message Broker, Queues and Topics. <strong>Port - 51515</strong></td>
</tr>
<tr>
<td>ActiveMQ OS</td>
<td>Windows 2008</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AO OS</td>
<td>Windows 2008</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CDP</td>
<td>AO Central</td>
<td>✓</td>
<td>For simplicity, we’ll use a single Peer configuration, CDP only.</td>
</tr>
<tr>
<td>JDK</td>
<td>1.6.0-34</td>
<td></td>
<td>For ActiveMQ. AO comes with its own JRE.</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Galileo</td>
<td></td>
<td>Optional testing section.</td>
</tr>
</tbody>
</table>

For simplicity and brevity, both the ActiveMQ and AO will reside on the same server. This will simplify any issues with Network firewalls, LDAP authentication, etc...

** The default port for ActiveMQ is 61616. The 61616 also happens to be used by AO, so I had to switch the ActiveMQ default port to 51515. You don’t need to actually do this if your ActiveMQ is on its own server though.

For this tutorial the basic steps are the following:
- Install and configure ActiveMQ.
- Verify the AO ActiveMQ components.
- Define a JNDI Properties file for the AO Queue(s).
- Configure an AO JMS Adapter to Consume the Queue.
- Configure an AO JMS Adapter to Produce to the Queue.
- Run the ActiveMQ Example(s) to test the Queue and the AO Monitor Adapter.
- Run the AO Workflows to Produce Messages to the Queue.

The last two steps are optional and intended to solidify the concepts presented in the tutorial. You'll need Eclipse and AO Design Studio to complete these steps.

3. Install ActiveMQ

http://activemq.apache.org/getting-started.html

The short version is:
- Download a binary distribution, I used 5.6.0.
- Unzip/unjar the package into a given directory, I used D:\usr\apps\apache-activemq-5.6.0\.
- Open a command/prompt (console/terminal).
- cd ${MQ-HOME}/bin ... d:\usr\apps\apache-activemq-5.6.0\bin.
- Issue the command, activemq

Depending on the Log level, you should see console messages similar to the following:

D:\>cd usr\apps\apache-activemq-5.6.0\bin
D:\usr\apps\apache-activemq-5.6.0\bin>activemq
Java Runtime: Sun Microsystems Inc. 1.6.0_30 D:\usr\bin\java\jdk-160-30-64\jre
Heap sizes: current=1004928k free=994439k max=1004928k
JVM args: -Dcom.sun.management.jmxremote -Xms1G -Xmx1G -Djava.util.logging.config.file=logging.properties -Dactivemq.classpath=D:\usr\apps\apache-activemq-5.6.0\bin\..conf;D:\usr\apps\apache-activemq-5.6.0\bin\..\conf; -Dactivemq.home=D:\usr\apps\apache-activemq-5.6.0\bin\.. -Dactivemq.base=D:\usr\apps\apache-activemq-5.6.0\bin\.. -Dactivemq.conf=D:\usr\apps\apache-activemq-5.6.0\bin\..\conf -Dactivemq.data=D:\usr\apps\apache-activemq-5.6.0\bin\..\data -Djava.io.tmpdir=D:\usr\apps\apache-activemq-5.6.0\bin\..\data\tmp
ACTIVEMQ_HOME: D:\usr\apps\apache-activemq-5.6.0\bin\..
ACTIVEMQ_BASE: D:\usr\apps\apache-activemq-5.6.0\bin\..
ACTIVEMQ_CONF: D:\usr\apps\apache-activemq-5.6.0\bin\..\conf
ACTIVEMQ_DATA: D:\usr\apps\apache-activemq-5.6.0\bin\..\data
Loading message broker from: xbean:activemq.xml
INFO | Refreshing
org.apache.activemq.xbean.XBeanBrokerFactory$1@671381e7: startup date
[Tue Oct 09 09:00:15 EDT 2012]; root of con
text hierarchy
INFO | PListStore:[D:\usr\apps\apache-activemq-
5.6.0\bin\..\data\localhost\tmp_storage ] started
INFO | Using Persistence Adapter:
KahaDBPersistenceAdapter[D:\usr\apps\apache-activemq-
5.6.0\bin\..\data\kahadb]
INFO | KahaDB is version 4
INFO | Recovering from the journal ...
INFO | Recovery replayed 1 operations from the journal in 0.01
seconds.
INFO | ActiveMQ 5.6.0 JMS Message Broker (localhost) is starting
INFO | For help or more information please see:
http://activemq.apache.org/
INFO | Listening for connections at: tcp://[MQ-SERVER]:51515
INFO | Connector openwire Started
INFO | ActiveMQ JMS Message Broker (localhost, ID:[MQ-SERVER]-61092-
1349787616950-0:1) started
INFO | jetty-7.6.1.v20120215
INFO | ActiveMQ WebConsole initialized.
INFO | started o.e.j.w.WebAppContext{/admin,file:/D:/usr/apps/apache-
activemq-5.6.0/webapps/admin/}
INFO | RESTful file access application at
http://0.0.0.0:8161/fileserver
INFO | Started SelectChannelConnector@0.0.0.0:8161

The Windows dist. comes with a handy BAT file to install ActiveMQ as a Service as well. This can be found in ${ACTIVEMQ-HOME}/bin/[win32|win64]/InstallService.bat.

To simplify the environment I installed the ActiveMQ onto the same server as AO. In a true Enterprise Service Bus architecture, the MQ server would be separate and distinct from the Clients. However, doing this introduced a conflict with AO that also uses the 61616 port. So the ActiveMQ port had to be modified to 51515.

Optional: To modify the default ActiveMQ port, do a text search in the ${MQ-
HOME}/conf/ XML files and replace all occurrences of 61616 with your port number of choice.

Once ActiveMQ is running, navigate your Web Browser to http://${mq-
server}:8161/admin/ and make sure the ActiveMQ Web Console comes up.

### 4. Verify the AO (Client) environment

To verify the AO/ActiveMQ section you can follow the Caropepe tutorial. At the time of writing this tutorial I did discover one anomaly verifying the ActiveMQ jarfiles. The activemq-all-5.4.2.jar file was missing in AO. When I checked the ActiveMQ site, it no
longer offered the 5.4.2, citing Bug so-n-so, please use the 5.4.3. Hence, I downloaded 5.4.3 and put that version out there.

Per the BMC Tutorial, I checked the BMC Servers and verified the following jar files were present:
- activemq-all-5.4.3.jar **
- activemq-core-5.4.2.jar
- spring-2.5.4.jar
- spring-core-2.5.4.jar
- spring-jms-2.5.4.jar

** By mixing the activemq-all-5.4.3.jar with the AO 5.4.2 versions, I didn’t encounter any issues. BUT, you should be aware of this mixture.

5. JNDI Properties

When you read the BMC Base Adapters documentation on the JMS Adapters, it states at the top the following:

Create administered objects at JMS provider (JMS interface) for the ConnectionFactory (an object that a client computer uses to create a connection to the JMS provider), queues (a staging area that contains messages) and topics (a mechanism for publishing messages) in the directory service designated for JNDI (Java Naming and Directory Interface) lookup.

In short, the AO JMS Adapters uses JNDI to lookup the Queue in ActiveMQ.

JNDI does more than that of course, but a deep dive into JNDI is beyond the scope of this tutorial. For simplicity and brevity, we’ll keep the discussion simple. That said, the following is a sample JNDI Properties file that will configure a Queue (ao-timeout) as an administered object that AO can lookup:

```java
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory

# use the following property to configure the default connector
java.naming.provider.url = tcp://$[mq-server]:61616

# use the following property to specify the JNDI name the connection factory
# should appear as.
connectionFactoryNames = connectionFactory, queueConnectionFactory, topicConnectionFactory

# register some queues in JNDI using the form
# queue.[jndiName] = [physicalName]
# queue.MyQueue = example.MyQueue
```
queue.ao-timeout = ao-timeout

# register some topics in JNDI using the form
# topic.[jndiName] = [physicalName]
topic.MyTopic = example.MyTopic

Note, an important concept to keep in the forefront of your mind in this tutorial, AO is the client here, ActiveMQ is the server. It’s easy to start thinking that ActiveMQ is the Client, since AO is our primary application.

This JNDI Properties file then needs to be placed somewhere in the AO Classpath of the CDP to allow the AO Adapters to pick up the Context/Connection Information. I did NOT investigate all possible placements of the file, I simply chose the CDP WEB-INF classes directory. Other options are simply jar the jndi.properties file and place it in a ./lib/ directory that AO uses in the $CLASSPATH.

${AO-HOME}/tomcat/webapps/baocdp/WEB-INF/classes/jndi.properties

Now recycle the AO environment to have it pick up this JNDI Properties file.

6. AO Adapter Configurations

With the ActiveMQ server up and the JNDI in place on AO, let’s configure new JMS Adapters in the AO Environment over to the Queue.

We’ll configure and mount the Consumer (Monitor) adapter first. I found this to be the more difficult one to get working. Your experience may differ.

Please note, I’ve put a placeholder, ${mq-server}, in the configurations that represents what your ActiveMQ server name is. You NEED to replace this placeholder!

- Navigate to AO > Grid Manager > Adapters
- Locate the ro-adapter-jms-monitor on the Adapters in Repository. If you don’t have the JMS Actor and Monitor Adapters, you’ll need to export them from another AO environment as *.aroar files and then Upload them to your AO environment.
- Checkbox the adapter and click the Add to Grid button.
- Click the Configure button next to the Adapter.
- Switch to XML View (usually defaults to it anyway).
- Name the Adapter MQ-TIMEOUT-CONSUMER.
- Paste the following XML into the Properties:

```xml
<config>
    <jndi>
      queue.ao-timeout = ao-timeout
      topic.MyTopic = example.MyTopic
    </jndi>
</config>
```
Navigate to Grid Manager > Peers > CDP.
Click the Edit (Pencil) button.
Click the Enable button.
Locate and select the MQ-TIMEOUT-CONSUMER.
Click the OK button.
Click the OK button.
Monitor the Grid Status and make sure the Adapter gives you a Green light. You can also monitor the grid.log, output will vary depending on your logging levels set in the Peer.
REPEAT all these steps for the Actor Adapter, MQ-TIMEOUT-PRODUCER. Both adapters will be talking to the same Queue.

So far, so good.

Note, If you don’t get a Green Light here are some common issues to resolve:
• Double check the mq-server name.
• Triple check the port number AND verify that nothing else on the ActiveMQ server is using your port. On mine, the default 61616 was actually being used, hence I switched over to 51515.
• Check your firewalls. 9 times out of 10, everything is up and running, but if you can’t tracert or telnet to the server/port, it is probably blocked by a firewall somewhere.

7. Testing the Queue
OK. We’ve covered a lot of material so far. To summarize, at this point you should have:
• ActiveMQ installed and up and running.
• JNDI Properties file in the AO $CLASSPATH.
• AO JMS Adapters configured and Enabled on the CDP.

**Consumer - Monitoring**

To work in the next section, you need to download, setup and configure the ActiveMQ Examples inside an Eclipse Project. Don’t confuse these Examples with the ActiveMQ Web Demos, these are the Java examples in the ActiveMQ download, ./example/ folder. For brevity, the tutorial assumes you are familiar with Eclipse, Java and AO.

• Download the ActiveMQ Examples. These are NOT the Web Demos. These are included in the ActiveMQ server download, ${MQ-HOME}/example/. The Java source code in in ./src. But you can also download them from the ActiveMQ site, [http://activemq.apache.org/examples.html](http://activemq.apache.org/examples.html).
• Unzip the Examples, if not already unpacked.
• Open up Eclipse and create a new Java Project.
• Right click on the Project/src folder and select Import...
• Expand the General item and Select File System from the list.
• Click the Next > button.
• Click the Browse button and locate the directory/folder where you have the Examples/src.
• Click the OK button, then review all the Import dialog settings:
- Click the Finish button and Import the source.

The initial Import brings in all the Java classes, BUT none of the required Libraries. As such, you’ll see a great many Red X’s scattered throughout the source code due to compile errors.
DO not be frightened…Java is like Playdoh…all we need to do is add the required jarfiles to the Eclipse build path and all the errors go away.

You’ll need to copy and add the following Java Libraries (jarfiles) onto the Eclipse Project from the ActiveMQ ./lib/ directory. I usually create a sub-folder in the Eclipse project called ./lib for adding jarfiles. All the following jarfiles can be found in the ActiveMQ ./lib/ and ./lib/optional/ sub-directories:

- activemq-core-5.6.0.jar
- geronimo-j2ee-management_1.1_spec-1.0.1.jar
- geronimo-jms_1.1_spec-1.1.1.jar
- log4j-1.2.16.jar
- slf4j-api-1.6.4.jar
- slf4j-log4j12-1.6.4.jar

Once you add these to the Build Path of the Eclipse Project, the compile errors should be gone.

With the complier errors gone, now select the default package in the ./src Project folder and select Search > File Search from the menu. If you’re like me and the default port of
61616 was changed on ActiveMQ installation, you need to search and replace all instances of 61616 in the source code to the port you setup and configured ActiveMQ to use.

Optional: Do a search for 61616 for Containing text: 61616. Then Click the Replace button and use 51515 (or your port number) to replace the default port.

We now need to copy and adjust a Producer example to fill the Ao Queue, ao-timeout.

- Open up your Eclipse project.
- Copy ProducerTool to ProducerAOTimeout.
- Copy the JNDI Properties file into the ./src folder of the Eclipse Project. This is the same JNDI properties file you placed on AO.
- Open the ProducerAOTimeout class in your Eclipse Java Editor.
- Modify the url to “failover://tcp://[mq-server]:61616“and the subject to “ao-timeout”

```java
private String subject = "ao-timeout";
private boolean topic;
private String user = ActiveMQConnection.DEFAULT_USER;
private String password = ActiveMQConnection.DEFAULT_PASSWORD;
private String url = "failover://tcp://<your-server>:<your-port>";
```

The original ProducerTool class we copied was designed to “create” a Queue on the fly, there is no JNDI lookup in the code. However, for the AO/ActiveMQ validation, we have already defined a Queue in the JNDI Properties and we need AO to perform a JNDI lookup. So we need to modify the example source code to do a simple JNDI Lookup of the queue.

- Locate the run() method.
The original code used an ActiveMQConnectionFactory. In our code we need to create a JNDI Context, then lookup our ConnectionFactory information:

```
// JNDI CONTEXT :
// Setup and create a JNDI Context to use for lookups...
Context context = new InitialContext();
// Create the connection.
System.out.println("URL [" + url + "]");

// CONNECTION FACTORY :
// Establish an MQ Connection Factory...
//ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(user, password, url);
ConnectionFactory connectionFactory = (QueueConnectionFactory) context.lookup("connectionFactory");
connection = connectionFactory.createConnection();
connection.start();
```

- Import the javax.jms.Queue class into the class.
- Define a new Queue just after the connection.start() and just before the Session. Again, we use the JNDI Context to lookup the Queue.

```
connection.start();
Queue queue = (Queue) context.lookup(this.subject);
// Create the session
Session session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);
```

- Comment out or delete the `if (topic) { ... } else { ... }`. This was intended to create a Queue and/or Topic on the fly.

```
// Create the session
Session session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);

[delete]
if (topic) {
    destination = session.createTopic(subject);
} else {
    destination = session.createQueue(subject);
}
[/delete]
```

- Finally, change the Message Producer to use the new Queue defined earlier:

```
// Create the producer.
MessageProducer producer = session.createProducer(queue);
```

- The completed `run()` method:

```
public void run() {
```
Connection connection = null;

try {
    // JNDI CONTEXT :
    // Setup and create a JNDO Context to use for lookups...
    Context context = new InitialContext();
    // Create the connection.
    System.out.println("URL [" + url + "]");
    // CONNECTION FACTORY :
    // Establish an MQ Connection Factory...
    ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(user, password, url);
    ConnectionFactory connectionFactory = (QueueConnectionFactory) context.lookup("connectionFactory");
    connection = connectionFactory.createConnection();
    connection.start();
    Queue queue = (Queue) context.lookup(this.subject);
    Session session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);
    // Create the session
    //            if (topic) {
    //                destination = session.createTopic(subject);
    //            } else {
    //                destination = session.createQueue(subject);
    //            }
    // Create the producer.
    MessageProducer producer = session.createProducer(queue);
    if (persistent) {
        producer.setDeliveryMode(DeliveryMode.PERSISTENT);
    } else {
        producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    }
    if (timeToLive != 0) {
        producer.setTimeToLive(timeToLive);
    }
    // Start sending messages
    sendLoop(session, producer);
    System.out.println("[" + this.getName() + "] Done.");
    synchronized (lockResults) {
        ActiveMQConnection c = (ActiveMQConnection) connection;
        System.out.println("[" + this.getName() + "] Results:
        c.getConnectionStats().dump(new IndentPrinter());
    }
} catch (Exception e) {
    System.out.println("[" + this.getName() + "] Caught: " + e);
    e.printStackTrace();
} finally {
    try {
        connection.close();
    } catch (Exception e) {
        System.out.println("[" + this.getName() + "] Could not close connection.");
    }
}
• Bring up the ActiveMQ Admin Console in your Browser.
• Navigate to the ActiveMQ Queues.
• Now login to your AO (CDP) server and navigate to the grid.log.
• Begin monitoring the grid.log. tail –f grid.log or Notepad++.
• Return your focus to the Eclipse Java class.
• Run the ProducerAOTimeout as a Java Application. When it runs, you’ll see the Eclipse console window fill with messages like the following:

```java
} catch (Throwable ignore) {
}
}
```

Connecting to URL: failover://tcp://${mq-server}:51515 (null: null)
Publishing a Message with size 255 to queue: ao-bppm-recovery
Using non-persistent messages
Sleeping between publish 0 ms
Running 1 parallel threads
URL [failover://tcp://${mq-server}:51515]
   INFO | Successfully connected to tcp://${mq-server}:51515
[Thread-2] Done.
[Thread-2] Results:

• In the ActiveMQ Browser, Refresh the page.
• 10 Messages will be queued to ao-timeout...If the AO Monitor Adapter (MQ-TIMEOUT-CONSUMER) is working, those messages will probably be immediately Dequeued as well.
You should also see messages in the grid.log similar to:

```
09 Oct 2012 10:45:24,615 DEBUG DestinationManager : Message XML is :
<message>
  <destination>ao-timeout</destination>
  <header>
    <JMSDeliveryMode>1</JMSDeliveryMode>
    <JMSExpiration>0</JMSExpiration>
    <JMSPriority>4</JMSPriority>
    <JMSMessageID>ID:LIBP03P-B07INGQ-63075-134979392324-1:1:1:1:10</JMSMessageID>
    <JMSTimestamp>1349793924022</JMSTimestamp>
    <JMSCorrelationID />
    <JMSReplyTo />
  </header>
  <body>
    <text>Message: 9 sent at: Tue Oct 09 10:45:24 EDT 2012</text>
  </body>
</message>
```

**Producer - Actor**

For this section we’ve written a very simple AO Workflow that will setup a message and commit that message to the Queue via our MQ-TIMEOUT-PRODUCER Adapter. You’ll also need the BMC/AO out-of-box workflows in the AutoPilot-AD-Utilities, specifically the JMS workflows.
• Open up your AO Design Studio.
• Create a new Module (MQ).
• Create a new Process in the module, mq-producer-test().
• Drag-n-Drop an Assign activity onto the Canvas.
• Create a single item in that Assign called messages.
• Double click the item row and set the Parameter type: XML.
• Set the value to the following:

  <messages>
  <text>This is my first message</text>
  <text>This is my second message</text>
  <text>This is my third message</text>
  <text>This is my fourth message</text>
  </messages>

• Set the Output, Context type: Context item.
• Set the Output, Context name: messages.
• Save your work and return to the Canvas.
• Drag-n-Drop the AutoPilot:JMS:Send Messages() workflow onto the Canvas after the Assign.
• Double-click the Send Messages workflow and set the following properties:
  o adapter name = String = MQ-TIMEOUT-PRODUCER
  o text messages = Context item = messages
  o destination = String = ao-timeout
• All other Inputs should be empty.
• Click OK and save your work.
• Drag-n-Drop a second Assign activity and name it log-n-event.
• Double-click the log-n-event and click the Logging Tab > Before Execution.
• Enter a simple log message and then log the Context Items, messages and Adapter Response.
• Click OK and Save all you work…return to the Canvas.
• Link the Activities and workflows in the following manner:
  o Start
  o Assign
  o Send Messages()
  o log-n-event
  o End
- Bring up the ActiveMQ Admin Console in your Web Browser.
- Navigate to the ActiveMQ Queues.
- Now login to your AO (CDP) server and navigate to the grid.log.
- Begin monitoring the grid.log. `tail -f grid.log` or Notepad++.
- Return your focus to AO Design Studio.
- Click and Save the Canvas.
- Click the Test Workflow button.
- Begin refreshing the ActiveMQ Web Page...if everything is working, you should see 4 additional message get added to the Queue...probably immediately Dequeued as well.

8. Synopsis

We’ve covered a great deal of material in this tutorial. I hope it works and you had a good experience.
I would finish by saying a few things on this:
The ActiveMQ used in this tutorial was essentially “wide-open”. There was no authentication used and the entire Queue platform was run in an Admin level security.
If you’re using an MQ JMS Broker with authentication and some level of security, then you’ll need to alter parts of this tutorial to get things to happen correctly.
The JNDI was basically ‘client’ side only. ActiveMQ and JNDI in particular has a much more granular control mechanism which is beyond the scope of this tutorial. We essentially placed the JNDI on the client and the Queue was created in a lazy fashion. To secure the Queue and provide a more point-to-point environment like a true Production environment, you’ll need to configure JNDI on the ActiveMQ side and explicitly configure a broker in ActiveMQ to that specific Queue.

Please comment on this tutorial and send any improvements and/or corrections...I would especially like to do a deeper dive into the JNDI aspects and discuss the controls needed to incorporate the point-to-point controls.

Thank You,

adym