BI-DIRECTIONAL JMS BRIDGE

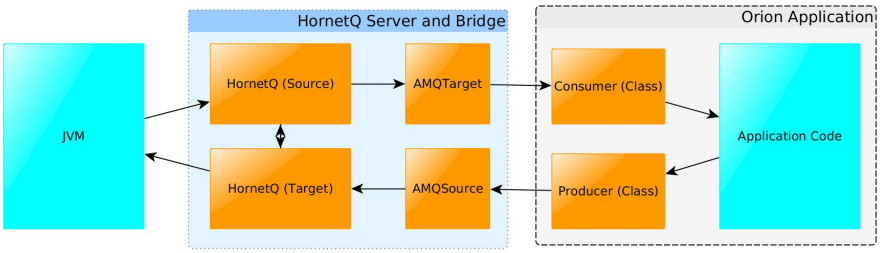
MICHAEL MEDING* , MMEDING@OUTSMARTINC.COM

CONTENTS

ABSTRACT

This article details the steps involved in creating a bi-directional message bridge between two applications using JMS and HornetQ. For this implementation I used Wildfly as my Java application server but you could use Glassfish or simalar and the implementation would be near identical. Additionally, I assume that the reader of this article has prior knowledge of Java EE 7, EJB, RESTful web services, ActiveMQ and Wildfly application servers.

1 OVERVIEW



As shown by the diagram above HornetQ and bridge definitions are contained in wildfly configuration. The orion application simply includes two classes to communicate back and forth across the two bridges. Messages are managed by ActiveMQ. ActiveMQ is included as an additional module

to wildfly and can be accessed by a web console at localhost:61616 when deployed with default settings.

2 SETUP WILDFLY APPLICATION SERVER

```
<!-- placed in the <hornetq-server> definition -->
<jms-destinations>
  <jms-queue name="JMSBridgeSourceQueue">
    <entry name="java:/queue/JMSBridgeSourceQ"/>
    <entry name="java:jboss/exported/jms/queue/JMSBridgeSourceQ"
        />
    <durable>true</durable>
  </jms-queue>
  <jms-queue name="AMQTargetQ">
    <entry name="java:/queue/AMQTargetQ"/>
    <entry name="java:jboss/exported/jms/queue/AMQTargetQ"/>
    <durable>true</durable>
  </jms-queue>
</jms-destinations>

<!-- Added after the <hornetq-server> definition -->

<!-- The forward bridge -->
<jms-bridge name="to-active-mq">
  <source>
    <connection-factory name="ConnectionFactory"/>
    <destination name="queue/JMSBridgeSourceQ"/>
  </source>
  <target>
    <connection-factory name="AMQConnectionFactory"/>
    <destination name="queue/JMSBridgeTargetQ"/>
  </target>
  <quality-of-service>AT_MOST_ONCE</quality-of-service>
  <failure-retry-interval>1000</failure-retry-interval>
  <max-retries>-1</max-retries>
  <max-batch-size>10</max-batch-size>
  <max-batch-time>100</max-batch-time>
</jms-bridge>
<!-- The return bridge -->
<jms-bridge name="from-active-mq">
  <source>
    <connection-factory name="AMQConnectionFactory"/>
    <destination name="queue/AMQSourceQ"/>
  </source>
  <target>
    <connection-factory name="ConnectionFactory"/>
    <destination name="queue/AMQTargetQ"/>
  </target>
  <quality-of-service>AT_MOST_ONCE</quality-of-service>
  <failure-retry-interval>1000</failure-retry-interval>
  <max-retries>-1</max-retries>
```

```
  <max-batch-size>10</max-batch-size>
  <max-batch-time>100</max-batch-time>
</jms-bridge>
}
```

STANDALONE-FULL.XML

This file is the standard configuration file of the Wildfly application server. For this to work properly you must add the HornetQ module to Wildfly before modifying the standalone-full.xml. Instructions to do this can be found easly online. The modifications that must be added for a bi-directional bridge are detailed below. The names given are arbitrary and can be whatever you would like. Throughout this article I follow a pattern of names using source as my origin and target as the endpoint. I would advise following a simalar pattern as it begins to get confusing when trying to detirmine the path of a message. Also keep in mind that the names in jms-destinations must match those in jms-bridge.

3 JVM CODE

3.1 Incomming Side (To JVM)

```
@MessageDriven(activationConfig = {
// these names must match
    @ActivationConfigProperty(propertyName = "destination", propertyValue = "
         java:jboss/exported/jms/queue/AMQTargetQ"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue
         = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "useJNDI", propertyValue = "true"
         ),
    @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue
         = "Auto-acknowledge")
})

public class MessageReceiverAsync implements MessageListener {

    @Override
    public void onMessage(Message message) {
        try {
            TextMessage tm = (TextMessage) message;
            System.out.println("Message received: " + tm.getText());
        } catch (JMSException ex) {
            Logger.getLogger(MessageReceiverAsync.class.getName()).log(Level.
                 SEVERE, null, ex);
        }
    }
}
```

This code implements a listener that as soon as a message is posted to AMQTargetQ (the return bridge) it takes that message and simply prints it out to console. This code runs on the JVM side according to the first diagram above and must be implemented within an EJB to function properly.

3.2 Outgoing Side (From JVM)

```
@Stateless
```

```
public class MessageSender {

    @Inject
    @JMSConnectionFactory("java:comp/DefaultJMSConnectionFactory")
    JMSContext context;

    @Resource(mappedName = "java:jboss/exported/jms/queue/JMSBridgeSourceQ")
    Queue queue;

    public void sendMessage(String message) {
        context.createProducer().send(queue, message);
    }
}
```

This is quite straightforward code which simply puts the message into the Queue. You must ensure that your factory name matches that in your standalone-full.xml. For this example I simply used the default that came with the module.

4 APPLICATION CODE

4.1 Outgoing Side (Producer)

```
try {
  // Create a ConnectionFactory
  ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
       "tcp://opmq1.outsmartinc.com:61616");

  // Create a Connection
  Connection connection = connectionFactory.createConnection();
  connection.start();

  // Create a Session
  Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

  // Create the destination (Topic or Queue)
  Destination destination = session.createQueue("AMQSourceQ");

  // Create a MessageProducer from the Session to the Topic or Queue
  MessageProducer producer = session.createProducer(destination);
  producer.setDeliveryMode(DeliveryMode.PERSISTENT);

  // Create a message
  //String text = "Hello world! From: " + Thread.currentThread().getName() + "
       : " + this.hashCode();
  String text = "new test message";
  TextMessage message = session.createTextMessage(text);

  // Tell the producer to send the message
  System.out.println("Sent message: " + message.hashCode() + " : " + Thread.
       currentThread().getName());
  producer.send(message);

  // Clean up
  session.close();
  connection.close();

} catch (Exception e) {
  System.out.println("Caught: " + e);
  e.printStackTrace();
}
```

The pattern for both the producer and consumer are quite simalar. Once you have a handle on the queue you just push or pull a message from it.

4.2 Incomming Side (Consumer)

```
try {
  // Create a ConnectionFactory
  ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
       "tcp://opmq1.outsmartinc.com:61616");

  // Create a Connection
  Connection connection = connectionFactory.createConnection();
  connection.start();

  // Create a Session
  Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

  // Create the destination (Topic or Queue)
  Destination destination = session.createQueue("JMSBridgeTargetQ");

  // Create a MessageConsumer from the Session to the Topic or Queue
  MessageConsumer consumer = session.createConsumer(destination);

  // Wait for a message
  Message message = consumer.receive(1000);

  if (message instanceof TextMessage) {
    TextMessage textMessage = (TextMessage) message;
    String text = textMessage.getText();
    System.out.println("Received: " + text);
  } else {
    System.out.println("Received: " + message);
  }

  consumer.close();
  session.close();
  connection.close();

  } catch (Exception e) {
    System.out.println("Caught: " + e);
    e.printStackTrace();
}
```

This is not a listener like the JVM side was. This will timeout after 1 second although if more time than that is required then something has gone wrong. Be sure to check the ActiveMQ console for message status.