Table of Contents

Persistent Reliable Messaging Lab

Goals

- Work with JBoss A-MQ failover, network of brokers, reliable messaging, and persistence features to enhance the JMS specification
- Complete exercises on failover, dynamic failover, network of brokers, and persistence

Lab Assets

The lab exercises are available in the following zip archive:

- GitHub GPE MW Training : Messaging Labs Repository

---

1. Set Up Static Failover

In these exercises, you use JBoss AM-Q to set up a failover protocol between a JMS client and two brokers. The two brokers are configured using a *Shared File System Master Slave* topology, so they compete with each other to acquire a lock on the file system.

1.1. Set Up Static Failover on Local Server

This exercise requires two JBoss A-MQ brokers configured with their own XML configuration files. The configurations are provided in the `brokerOne` and `brokerTwo` folders of `4_Persistence_Reliable_Messaging/failover`. To do this exercise, other brokers running on the local machine must be shut down.

Introduction

The project has four parts:

- The `brokerOne` project is the first message broker.
- The `brokerTwo` project is the second message broker.

- The consumer project starts a JMS MessageConsumer that reads messages as long as it can find one. The consumer is artificially slowed down so that you can test switching back and forth between message brokers.
- The producer project starts a JMS MessageProducer that creates the specified number of JMS messages and then stops and exits.

Procedure

1. If any other brokers are running on the local machine, shut them down.
2. Start the `brokerOne` message broker:
   a. Open a command line terminal (Windows or UNIX) and navigate to `4_Persistence_Reliable_Messaging/failover`.
   b. Run this Maven command:

```
mvn -P brokerOne
```

   - This starts the first JBoss A-MQ message broker on port 61616 and displays a rolling log file.
3. Start the `brokerTwo` message broker:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/failover`.
   b. Run this Maven command:

```
mvn -P brokerTwo
```

   - This starts the second JBoss A-MQ message broker.
   - The broker is started, but the transport connector is not started.
   - Only the JMX connection is available.

   c. Verify that the second broker is in standby mode because it cannot acquire the lock on the file system:

```
[INFO] Database ../shared-broker-data/kahadb/lock is locked... waiting 10 seconds for
the database to be unlocked. Reason: java.io.IOException: File '../shared-broker-
data/kahadb/lock' could not be locked.
```

4. Start the consumer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/failover`.
   b. Run this Maven command:

```
mvn -P consumer
```

5. Start the producer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/failover`.
   b. Run this Maven command:

```
mvn -P producer
```

6. Check failover behavior:
   a. Determine which broker is active by checking for the following log message in the server log:

```
'ActiveMQ JMS Message Broker ... started'
```

   b. After some messages are consumed, press **Ctrl+C** to shut down the active broker.
      - Observe that the other broker becomes active and that messages from the consumer and producer indicate that they switched to the other broker.

The process of switching to the other broker takes a few seconds because the locked broker waits 10 seconds before checking if the lock can be acquired.

   c. Take note of the following message, which is seen in the producer and the consumer when the first broker is shut down and the failover broker reconnects:

```
10:43:58 WARN  Transport (tcp://localhost/127.0.0.1:61616@59322) failed, reason:  ,
attempting to automatically reconnect
java.io.EOFException
 at java.io.DataInputStream.readInt(DataInputStream.java:392)
 at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:258)
 at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java:221)
 at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:213)
 at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:196)
 at java.lang.Thread.run(Thread.java:744)
10:44:02 INFO  Successfully reconnected to tcp://localhost:62616
```

The consumer project is designed to continue listening for messages.

7. To exit the consumer project, enter **Ctrl+C** for each consumer.

Failover Notes

The configuration of the client is defined with the class `Producer.java` and `Consumer.java` and contains a `String uri` to connect to the broker using the failover protocol. The addresses of the two brokers are defined as follows:

```
String uri = "failover:(tcp://localhost:61616,tcp://localhost:62616)";
```

By default, the protocol randomly chooses a URI from the list and attempts to establish a network connection. If it does not succeed, or if it subsequently fails, a new network connection is established to an alternative URI from the list. Thus, each broker acts as a failover for the other and can take over if the other broker is down. Clients automatically reconnect to the second broker and consume messages.

If both brokers are shut down and one broker comes back up quickly, the message exchange proceeds without failing.

1.2. Set Up Static Failover on OpenShift Enterprise

Introduction

This exercise requires two AMQ brokers set up on the OpenShift Enterprise JBoss A-MQ cartridge with a master/slave topology (the same configuration used for local usage in the previous module). Deploying the same Fuse Fabric broker profile within the two containers ensures that they start as a master/slave group because they use the same KahaDB storage location, and therefore compete to acquire the lock on the file system. This allows the brokers to function in a client failover mode.

The project has four parts:

- The `brokerOne` container is the first message broker.
- The `brokerTwo` container is the second message broker.
- The consumer project starts a JMS MessageConsumer that reads messages as long as it can find one.
- The producer project starts a JMS MessageProducer that creates the specified number of JMS messages and then stops and exists.

Procedure

1. If any other brokers are running on the local machine, shut them down. Do not skip this step.
2. Log in to the JBoss Fuse Management Console of your OpenShift environment.
3. Create and select a new broker profile:
   a. Go to the **MQ** tab.
   b. Click the **Broker** button to create a new broker profile.
   c. In the **Broker name** field, enter `masterSlaveBroker`.
   d. In the **Kind** field, select **MasterSlave** from the drop-down list

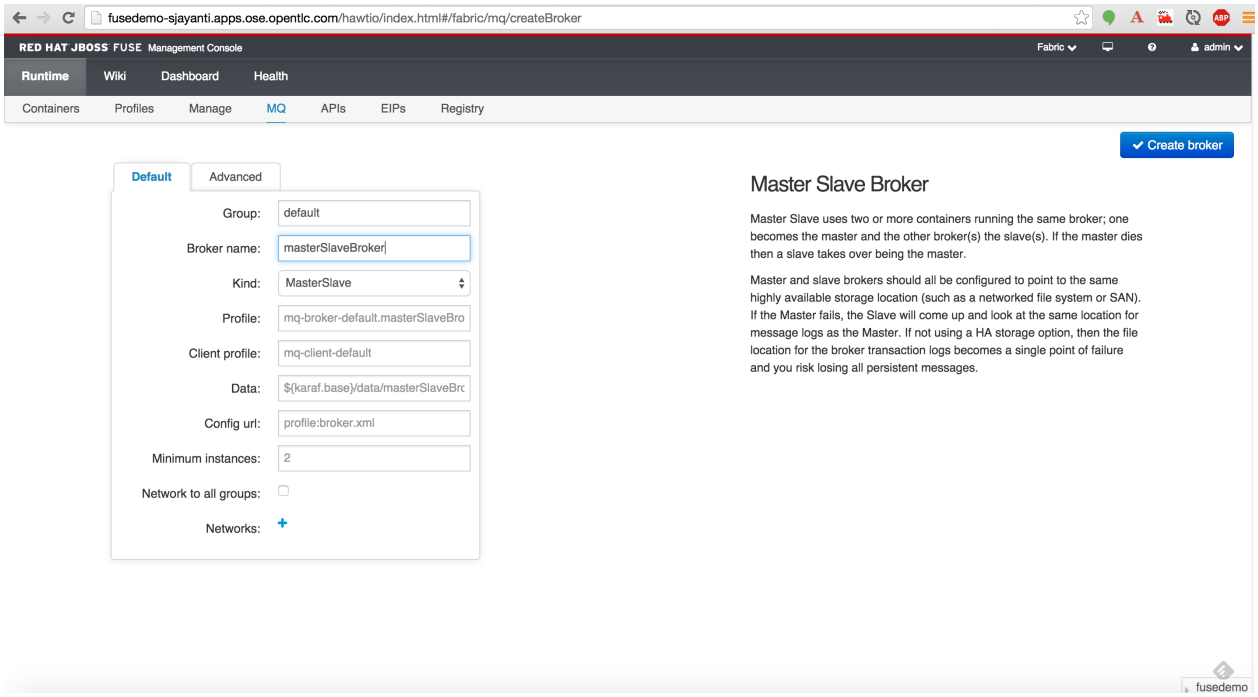

Figure 1. New broker profile

e. Click **Create broker** to create the Fabric broker profile.

f. Verify that the broker profile appears in your list of brokers.

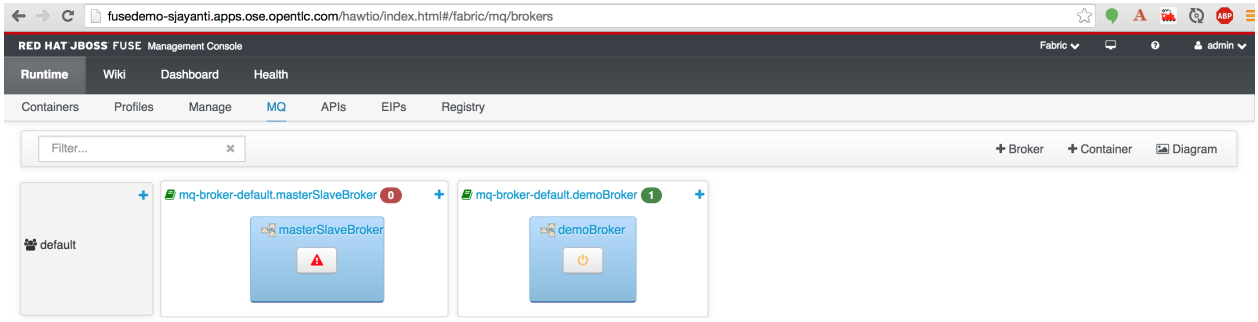

Figure 2. List of brokers

g. Select the **masterSlaveBroker** fabric profile.

4. Create two containers:

a. Click the red warning button.

b. In the **Container type** field, select **openshift**.

c. Fill in the **OpenShift Login** and **OpenShift Password** fields and log in to your OpenShift account.

d. In the **Container Name** field, enter `mqmasterslave`.

e. Leave the defaults for the other fields.

- Notice that the default number of containers is 2.

f. Click **Create and start container**.

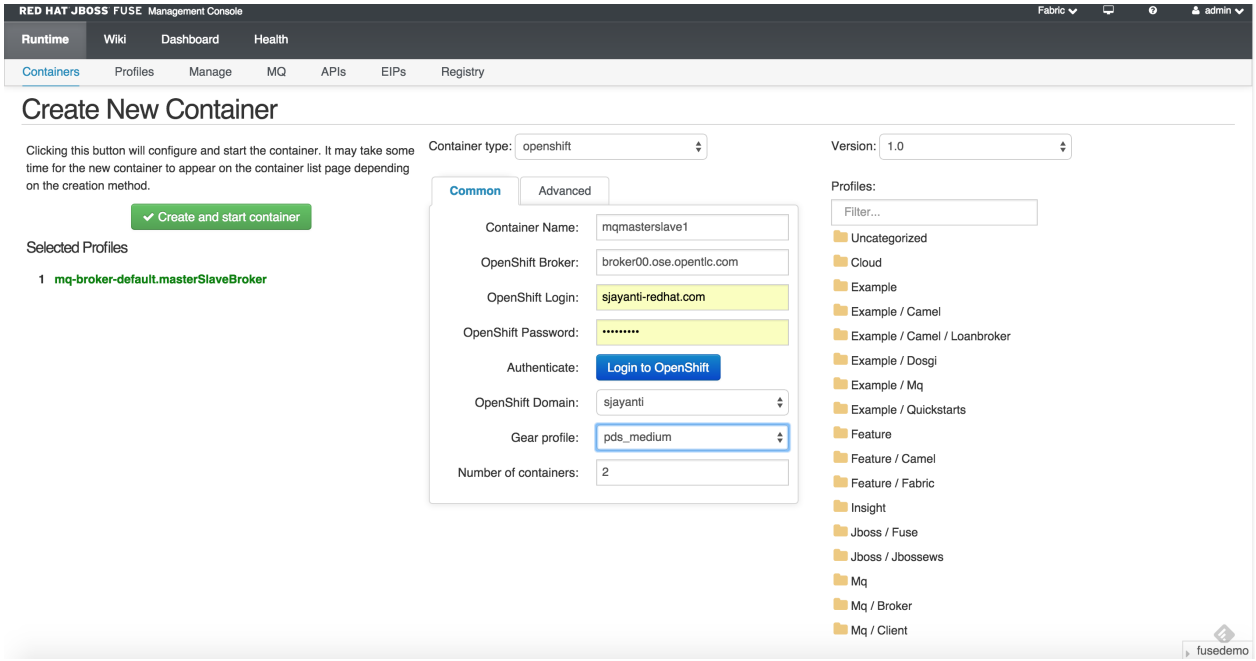

Figure 3. Create new container

g. Observe that your new containers are created with the names `mqmasterslave1` and `mqmasterslave2`.

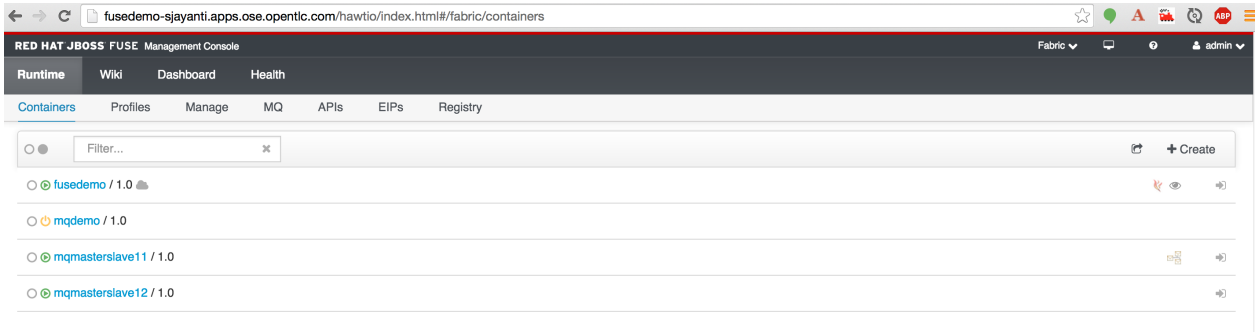

Figure 4. New containers: `mqmasterslave1` and `mqmasterslave2`

Because this is a master/slave setup, after the JBoss Fuse containers are started, only one of them has an active broker URL.

5. Verify the active broker by checking the registry view under `fabric/registry/clusters/fusemq/default/{session_id _of_the_broker}`.
   - If the broker is active, then its transport protocol service is displayed: `"tcp://hostname:portNumber"`.

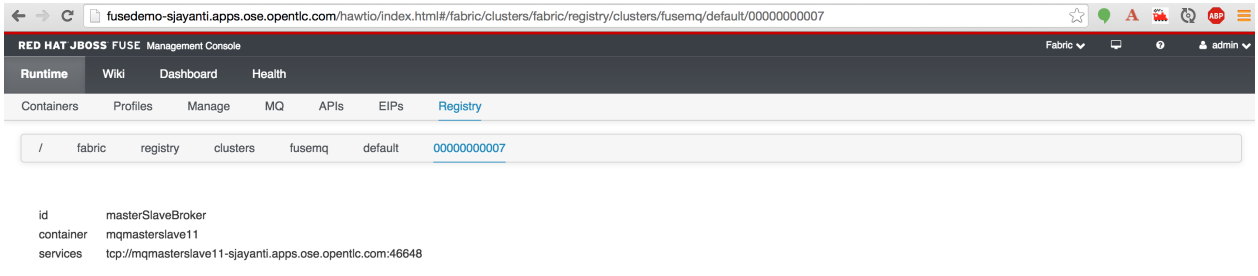

Figure 5. `masterSlaveBroker` with transport protocol

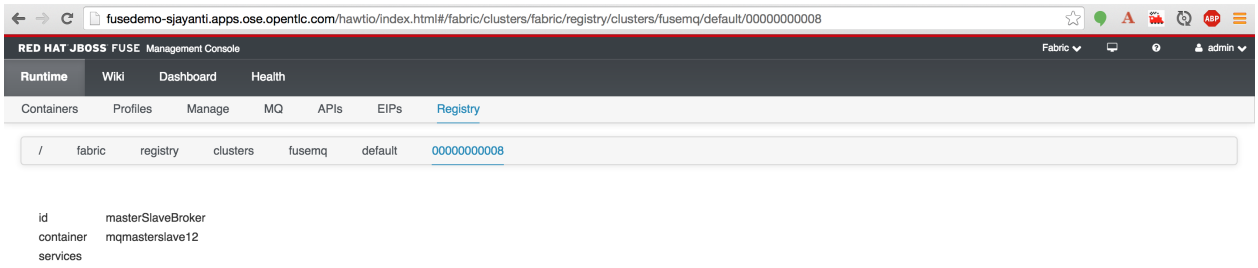

Figure 6. `masterSlaveBroker` container

6. Shut down the first container and notice that you now see the broker URL for the second container.

Record both broker URLs. They are required to configure the client.

7. Edit the `jndi.properties` file located in the `src/main/resources` directory of the failover project:
   a. Edit the *java.naming.provider.url* to use the URL below:

```
failover:(tcp://mqmasterslave11-
sjayanti.apps.ose.opentlc.com:46648,tcp://mqmasterslave12-
sjayanti.apps.ose.opentlc.com:43093)
```

The broker URLs should correspond to your OpenShift Enterprise broker URLs.

   b. Edit `SimpleConsumer.java` and `SimpleProducer.java` to provide the correct login/password credentials for your OpenShift Enterprise broker.
   c. Save your changes and compile the project:

```
mvn clean compile
```

8. Start the consumer:
   a. Open a command line terminal (Windows or UNIX) and navigate to `4_Persistence_Reliable_Messaging/failover`.
   b. Run this Maven command:

```
mvn -P consumer
```

9. Start the producer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/failover`.
   b. Run this Maven command:

```
mvn -P producer
```

- After the producer successfully connects to the broker, it starts sending messages.

10. Check the failover process:
    a. As messages are being sent, go to the Management Console and stop the container that is running the master broker.
    b. Verify the following:
        - The other broker becomes active
        - Messages state that the consumer and provider switched to the other broker
        - Messages continue to be processed

The consumer project is designed to continue listening for messages.

11. To exit the consumer project, press **Ctrl+C** for each consumer.

---

2. Set Up Dynamic Failover

Introduction

In this exercise you set up a dynamic failover topology.

Instead of defining the TCP addresses of the two brokers to be used with the failover protocol of the client, as you did for the static failover exercise, in this exercise you use the multicast discovery protocol to discover the brokers running in your local network.

The procedure sets up the failover protocol and runs the brokers on a local machine.

This exercise uses two specific XML configuration files, `broker1` and `broker2`, which are provided in the `src/main/resources/conf` directory of `4_Persistence_Reliable_Messaging/failover-dynamic`. No other brokers can be running on the local server.

The project has four parts:

- The `brokerOne` project is the first message broker.
- The `brokerTwo` project is the second message broker.
- The consumer project starts a JMS MessageConsumer that reads messages as long as it can find one.
- The producer project starts a JMS MessageProducer that creates the specified number of JMS Messages and then stops and exits. The producer is artificially slowed down so that you can test switching back and forth between message brokers.

Unlike in the previous lab exercise, in this exercise, both brokers use a different persistence store, so there is no shared persistence for failover.

Procedure

1. If any other brokers are running on the local machine, shut them down. Do not skip this step.
2. Start the `brokerOne` message broker:
   a. Open a command line terminal (Windows or UNIX) and navigate to `4_Persistence_Reliable_Messaging/failover-dynamic`.
   b. Run this Maven command:

```
mvn -P brokerOne
```

   - This starts the first ActiveMQ message broker on port 61616 and displays a rolling log file.
3. Start the `brokerTwo` message broker:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/failover-dynamic`.
   b. Run this Maven command:

```
mvn -P brokerTwo
```

   - This starts the second ActiveMQ message broker on port 61617 and displays a rolling log file.
4. Start the consumer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/failover-dynamic`.
   b. Run this Maven command:

```
mvn -P consumer
```

5. Check the broker to which the client is connected by looking at the console output:

```
[INFO] --- exec-maven-plugin:1.2.1.jbossorg-3:java (default) @ failover-dynamic ---
18:01:11 INFO  Adding new broker connection URL: tcp://dabou.local:61617
18:01:11 INFO  Adding new broker connection URL: tcp://dabou.local:61616
18:01:11 INFO  Successfully connected to tcp://dabou.local:61617
```

6. Start the producer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/failover-dynamic`.
   b. Run this Maven command:

```
mvn -P producer
```

7. Check the failover process:
   a. After some messages are consumed by the client, press **Ctrl+C** to shut down the broker identified in step 5 and notice that the consumer switches to the other broker.

```
Consumer

18:05:33 WARN  Transport (tcp://dabou.local/192.168.1.80:61616@53024) failed, reason:
, attempting to automatically reconnect
java.io.EOFException
 at java.io.DataInputStream.readInt(DataInputStream.java:392)
 at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:258)
 at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java:221)
 at org.apache.activemq.transport.tcp.TcpTransport.doRun(TcpTransport.java:213)
 at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:196)
 at java.lang.Thread.run(Thread.java:745)
18:05:33 INFO  Successfully reconnected to tcp://dabou.local:61617
```

This message is seen in both the producer and the consumer when the first broker is shut down.

   b. (Optional) On the client side, change the name of the group to a group that doesn't exist and observe what happens.

The consumer project is designed to continue listening for messages.

8. To exit the consumer project, press **Ctrl+C** for each consumer.

Dynamic Failover Notes

The discovery multicast protocol is configured at the broker side. Both brokers communicate via multicast in the `default` group. For `brokerOne`, you can see this in `broker01.xml` in the `discoveryUri` property:

```
<transportConnectors>
    <transportConnector name="openwire" uri="tcp://0.0.0.0:61616"
discoveryUri="multicast://default" />
</transportConnectors>
```

And in the `brokerTwo` `discoveryUri` property:

```
<transportConnectors>
    <transportConnector name="openwire" uri="tcp://0.0.0.0:61617"
discoveryUri="multicast://default" />
</transportConnectors>
```

The only difference is the URI definition, because the two brokers listen on different TCP ports,

61616 and 61617.

On the client side, the Java class of the producer and the consumer uses the following URL to connect to the broker:

```
discovery:multicast://default?group=default
```

This is where the multicast discovery protocol is defined and where the default group of the brokers is accessed.

From a connectivity point of view, dynamic failover works the same as static failover. A client randomly chooses a URI from the list provided in its failover URI string. After the connection is established, the list of available brokers is updated to reflect the brokers discovered in the `default` group. If the connection fails, the client randomly selects a new URI from its dynamically generated list of brokers.

If the new broker is configured to supply a failover list, the new broker updates the client's list.

---

3. Set Up a Network of Brokers (Localhost)

This lab is for setting up a network of brokers on localhost. If you are using an OpenShift environment, skip this exercise and go to the next section in this lab: *Set Up a Network of Brokers (OpenShift Environment)*.

Introduction

In this exercise you set up a network of three brokers, using JBoss A-MQ technology. The first broker contains a network connector with a URI string that points to the second and third brokers. This setup allows a message produced on `brokerOne` to be forward and consumed by `brokerTwo` or `brokerThree`.

This exercise requires three specific A-MQ broker configuration files, which are provided in the `broker01`, `broker02`, and `broker03` directories of `4_Persistence_Reliable_Messaging/network-of-brokers`. Other brokers in the local server must be shut down to do this exercise.

The project has five parts:

- The `broker01` project is the first message broker.
- The `broker02` project is the second message broker.
- The `broker03` project is the third message broker.
- The consumer project starts a JMS MessageConsumer that reads messages as long as it can find one.
- The producer project starts a JMS MessageProducer that creates the specified number JMS

messages and then stops and exits. The sending of messages is artificially slowed with 100ms "sleep" between messages to enable you to interrupt them when desired.

Procedure

1. If any other brokers are running on the local machine, shut them down. Do not skip this step.
2. Start the `brokerOne` message broker:
   a. Open a command line terminal (Windows or UNIX) and navigate to `4_Persistence_Reliable_Messaging/network-of-brokers`.
   b. Run this Maven command:

```
mvn -P brokerOne
```

   - This starts the first JBoss A-MQ message broker on port 61616 and displays a rolling log file.

If brokers 2 and 3 are not yet started, then you see this warning message in the `brokerOne` log:

```
[INFO] Establishing network connection from vm://broker1?
async=false&network=true to tcp://localhost:62616
[INFO] Connector vm://broker1 started
[INFO] broker1 Shutting down
[INFO] Connector vm://broker1 stopped
[INFO] broker1 bridge to Unknown stopped
[WARNING] Could not start network bridge between: vm://broker1?
async=false&network=true and: tcp://localhost:62616 due to:
java.net.ConnectException: Connection refused
```

3. Start the `brokerTwo` message broker:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/network-of-brokers`.
   b. Run this Maven command:

```
mvn -P brokerTwo
```

   - This starts the second ActiveMQ message broker on port 62616 and displays a rolling log file.
4. Start the `brokerThree` message broker:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/network-of-brokers`.
   b. Run this Maven command:

```
mvn -P brokerThree
```

```
```

- This starts the third ActiveMQ message broker on port 63616 and displays a rolling log file.

5. Start the consumer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/network-of-brokers`.
   b. Run this Maven command:

```
mvn -P consumer
```

6. Start the producer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/network-of-brokers`.
   b. Run this Maven command:

```
mvn -P producer
```

7. Create the network of brokers by connecting `brokerOne` through a network connector to `brokerTwo` and `brokerThree`.

```
<networkConnectors>
    <networkConnector uri="static:(tcp://localhost:62616,tcp://localhost:63616)"
                      prefetchSize="1000"/>
</networkConnectors>
```

The consumer project is designed to continue listening for messages.

8. To exit the consumer project, press **Ctrl+C** for each consumer.

Broker Network Notes (Localhost)

This procedure creates a one-way forward of messages from the first broker to one of the other two brokers. On the client side (producer and consumer Java class), the broker TCP connection to be used is passed as a Java system property.

The producer is connected by default to `brokerOne`:

```
// figure out who to connect to:
String broker = System.getProperty("broker", "localhost:61616");
String uri = "tcp://" + broker;
```

The consumer is connected by default to `brokerTwo`:

```
// figure out who to connect to:
```

```
String broker = System.getProperty("broker", "localhost:62616");

String uri = "tcp://" + broker;
```

You can change the consumer connection to `brokerThree` by passing the system property as a Maven parameter with the TCP URL to be connected:

```
mvn -P consumer -Dbroker=localhost:63616
```

If two consumers are started in parallel and are respectively connected to `brokerTwo` and `brokerThree`, then messages are load-balanced. The first consumer gets the odd messages, and the second consumer gets the even messages.

4. Set Up a Network of Brokers (OpenShift Environment)

Introduction

In this exercise, you set up a network of brokers in JBoss A-MQ running on OpenShift Enterprise. The exercise requires JBoss A-MQ brokers configured in a different way from a standalone broker. You reuse the master/slave broker and JBoss Fuse containers created earlier, and create an additional broker to join to them in a network of brokers configuration.

The project has five parts:

- The `networkofbroker01` container is the first message broker.
- The `masterslavecontainer01` container is the second message broker.
- The `masterslavecontainer02` container is the third message broker.
- The consumer project starts a JMS MessageConsumer that reads messages as long as it can find one. The consumer is artificially slowed down so that you can test switching back and forth between the message brokers.
- The producer project starts a JMS MessageProducer that creates the specified number JMS messages and then stops and exits. The sending of messages is artificially slowed with 100ms "sleep" between messages to enable you to interrupt them when desired.

Procedure

1. Create a new broker profile:
   a. Log in to the JBoss Fuse Management Console.
   b. Go to the **MQ** tab.
   c. Click **Broker** to create a new broker profile.
   d. Complete the fields of the **Default** tab:
      - In the **Kind** field, select **StandAlone** from the drop-down list

- In the **Broker name** field, enter `networkOfBroker`.
- In the **Group** field , enter `receiver`.
- In the **Networks** field, select **default**, the name used when you created the master/slave topology.

You should have created `mqmasterslavebroker` in the `default` group earlier. If it was not, enter the name of the group within which `mqmasterslavebroker` was created.

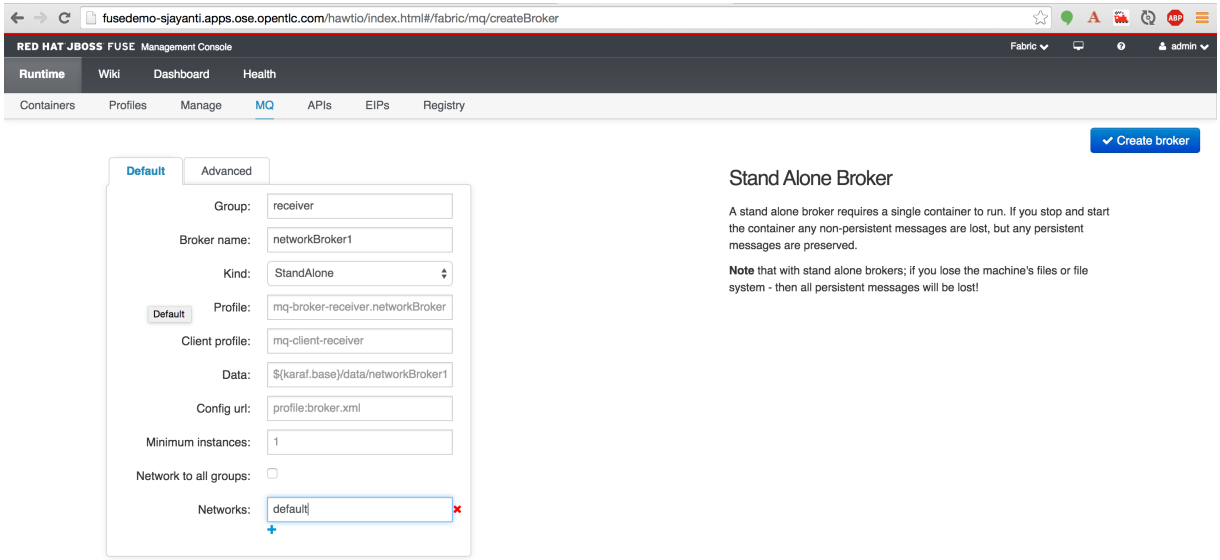

Figure 7. Broker profile

e. Click **Create broker** to create the broker profile.
   - You should see the broker profile in your list of brokers.

2. Create a container:
   a. Click the red warning button in the broker.
   b. For container type, select **openshift**.
   c. Log in to your OpenShift account.
   d. In the **Container Name** field, enter `networkofbroker`.
   e. Click **Create and start container.**
      - You should see your new container in the list of Fuse containers.

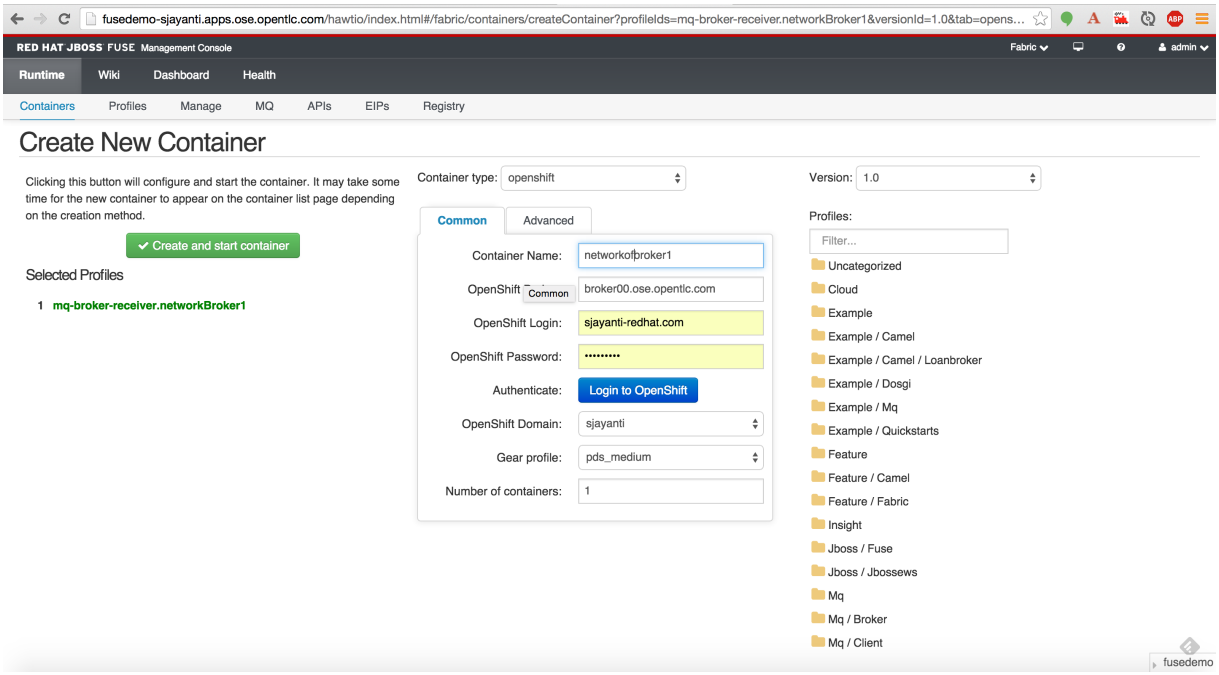

Figure 8. Create new container

3. Verify that the master/slave broker containers `masterslavecontainer01` and `masterslavecontainer02` are started.
4. Start the consumer:
   a. Open a command line terminal (Windows or UNIX) and navigate to `4_Persistence_Reliable_Messaging/network-of-brokers`.
   b. Run this Maven command:

```
mvn -P consumer
```

5. Start the producer:
   a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/network-of-brokers`.
   b. Run this Maven command:

```
mvn -P producer
```

The network of brokers is created by connecting the first broker (`networkofbroker`) through a network connector to the master/slave brokers. This allows for a one way transfer of messages from `networkofbroker` to the master broker part of the master/slave topology.

6. Because messages are sent to `networkofbroker`, ensure that the URL for this broker is provided in the `String broker` of the producer (see *SimpleProducer.java class*).

```
// figure out who to connect to:
String broker = System.getProperty("broker", “networkofbrokers1-
```

```
sjayanti.apps.ose.opentlc.com:60523”);
String uri = "tcp://" + broker;
```

7. Ensure that the userid/password and the connection URL are provided to the JBoss Fuse cartridge on OpenShift Enterprise:
   a. To check which broker is the master and active, open the **Registry** view under `fabric/registry/clusters/fusemq/default/{session_id _of_the_broker}` and check if the `"tcp://hostname:portNumber"` transport protocol service is displayed.
   b. Note the port number. You need it it to access the broker.

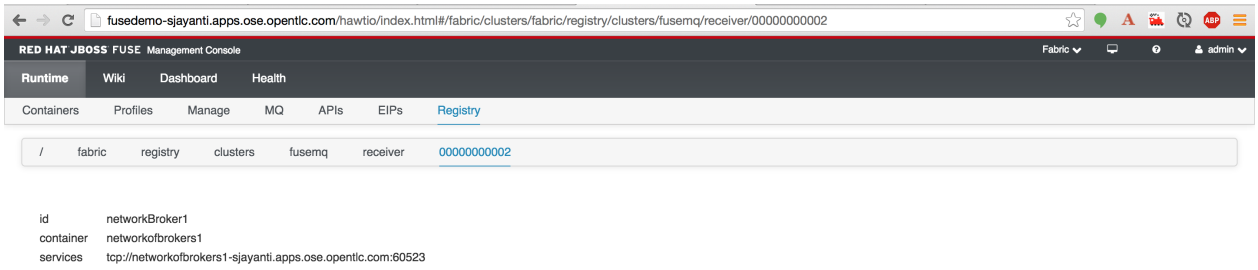

Figure 9. Registry view

Network of Brokers Notes (OpenShift)

Instead of changing the code and recompiling it with `mvn compile` before running `mvn -Pconsumer` or `mvn -Pproducer`, you can also change the URL of the connection to access the broker running on OpenShift Enterprise. You can do this by setting the broker's Java System Property using an Apache Maven parameter:

```
mvn -Pproducer -Dbroker=networkofbroker_user.apps_ose_opentlc.com:port_number
```

`networkofbroker_user.apps_ose_opentlc.com` is the JBoss Fuse OpenShift Enterprise gear running the `networkofbroker` Fuse container.

As with the machine hostname, you can use the **Registry** view of the JBoss Fuse Management Console to retrieve the port number of the broker instance from `fabric/registry/clusters/fusemq/receiver/`.

The JMS consumer is connected to one of the brokers of the master/slave topology, as defined in the SimpleConsumer Java class:

```
// figure out who to connect to:
String broker = System.getProperty("broker", "mqmasterslave11-
sjayanti.apps.ose.opentlc.com:46648");

String uri = "tcp://" + broker;
```

You can change the consumer to connect to the master/slave brokers using the failover protocol:

```
// figure out who to connect to:
String broker = System.getProperty("broker", "failover:(tcp://mqmasterslave11-
sjayanti.apps.ose.opentlc.com:46648,tcp://mqmasterslave12-
sjayanti.apps.ose.opentlc.com:43093)");
String uri = broker;
```

Now, if the JBoss Fuse container `mqmasterslavecontainer1` is shut down while the messages are consumed by the JBoss A-MQ broker, the consumer reconnects to the broker running `mqmasterslavecontainer2` and continues to consume messages.

---

5. Use KahaDB-Based Persistence

Introduction

In this exercise you use KahaDB-based JMS persistence with JBoss A-MQ.

The project has three parts:

- The root project starts the ActiveMQ message broker.
- The consumer project starts a JMS MessageConsumer that reads messages as long as it can find one.
- The producer project starts a JMS MessageProducer that creates the specified number of JMS messages and then stops and exits.

The exercise includes two scenarios. First you test with a persistent producer and ensure that messages persist during a broker restart. Second, you test with a non-persistent producer and confirm that messages do not persist after a restart.

Procedure

1. If it is not already running, start the message broker as described in the Module 1 lab.
   - By default, the connection details are provided for connecting to the broker on localhost.
   - If you are using the OpenShift environment, use the correct connection details (URL, `admin` user ID/password) to connect to the OpenShift Enterprise JBoss Fuse cartridge.
   - The message brokers (both OpenShift and localhost) have persistence defined in these settings:

     ```
     <persistenceAdapter>
       <kahaDB directory="./target/activemq-data/kahadb"/>
     </persistenceAdapter>
     ```

2. Start the producer:
   a. Check that the following line in `SimpleProducer.java` is commented:

      ```
      //producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
      ```

b. Open a command line terminal (Windows or UNIX) and navigate to `4_Persistence_Reliable_Messaging/persistence`.

c. Run this Maven command:

```
mvn -P producer
```

d. Confirm that all 100 messages are sent to the broker.

3. Stop the broker by doing one of the following:
   - If you are using a local broker, press **Ctrl+C** in the terminal window where the broker was started.
   - If you are using an OpenShift environment, stop the `mqdemo` container.
4. Start the message broker as described in the Module 1 lab.
5. Start the consumer:

a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/persistence`.

b. Run this Maven command:

```
mvn -P consumer
```

c. Confirm that the consumer received all 100 messages sent by the producer before the broker was restarted.

d. Exit the consumer by pressing **Ctrl+C**.

6. Start the **non_persistent** producer:

a. Uncomment the following line in `SimpleProducer.java`:

```
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

b. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/persistence`.

c. Run this Maven command:

```
mvn -P producer
```

d. Confirm that all 100 messages are sent to the broker.

7. Stop the broker by doing one of the following:
   - If you are using local broker, press **Ctrl+C** in the terminal window where the broker was started.
   - If you are using an OpenShift environment, stop the `mqdemo` container.
8. Restart the broker.

9. Start the consumer:
    a. Open another terminal window and navigate to `4_Persistence_Reliable_Messaging/persistence`..
    b. Run this Maven command:

```
mvn -P consumer
```

    c. Confirm that the consumer did not receive any messages from the broker.
10. To exit the consumer project, press **Ctrl+C**.

Last updated 2015-10-27 21:13:22 EDT