Table of Contents

Core Features Lab

Goals

- Understand the core features of JMS, particularly JBoss A-MQ
- Complete lab exercises on durable chat, multiple consumers, request-reply pattern and transacted chat

Lab Assets

The lab exercises are available in the following zip archive:

- GitHub GPE MW Training : Messaging Labs Repository

---

1. Create a Durable Chat Client

Introduction

In this lab exercise you create a durable subscription chat client. The exercise uses an application that catches command line input and issues a message based on encoded text. This exchange is a chat.

The application also listens for incoming messages from the other subscribers published on the chat room topic. To support recovering messages while it was disconnected, the durable property is used to create the subscription between the JMS client and the JBoss A-MQ Broker.

The application publishes messages to a chat topic and creates a simple subscription to the topic using a unique subscriber name, so that it can listen to the messages published by all the subscribers. The sending of messages is artificially slowed with 100ms "sleep" between messages so you can interrupt them when desired.

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 an OpenShift environment, use the correct connection details (URL, `admin` user ID/password) to connect to the OpenShift Enterprise JBoss Fuse cartridge.

2. Start the first chat client:
   a. Open a command line terminal (Windows or UNIX) and navigate to `2_Core_Features_Extensions/durable-chat`.
   b. Run this Maven command:

```
mvn -P chatter -DchatName=chatterOne
```

Do not use spaces in the chat name. Instead of `chatterOne`, you can use any unique name.

```
Tests run: 0, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.3.2:jar (default-jar) @ durable-chat ---
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default) @ durable-chat >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default) @ durable-chat <<<
[INFO]

14:30:38 INFO  Start simple chat client for: one

DurableChat application:
========================
The application will publish messages to the topic://durable.chatter.room topic.
The application also creates a simple subscription to that topic with this name:
'one'to consume any messages published there.

Type some text, and then press Enter to publish it as a Text Message from one.
```

3. Start the second chat client:
   a. Open another command line terminal (Windows or UNIX) and navigate to `2_Core_Features_Extensions/durable-chat`.
   b. Run this Maven command:

```
mvn -P chatter -DchatName=chatterTwo
```

Do not use spaces in the chat name. Instead of `chatterTwo`, you can use any unique name.

```
[INFO]
[INFO] --- maven-jar-plugin:2.3.2:jar (default-jar) @ durable-chat ---
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default) @ durable-chat >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default) @ durable-chat <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default) @ durable-chat ---
14:31:31 INFO  Start simple chat client for: two

DurableChat application:
========================
The application will publish messages to the topic://durable.chatter.room topic.
The application also creates a simple subscription to that topic with this name:
'two'to consume any messages published there.


Type some text, and then press Enter to publish it as a Text Message from two.
```

4. Enter some text in each window.
5. Observe that the messages appear in both chat windows as each chat client listens to the message on the same topic.

Because this chat is over a durable subscription, you can kill one client (Ctrl-C) and continue to chat on the other client. When you bring back the client later (using its *chatName*), it displays the messages sent since it was killed.

6. (Optional) There is no limit to the number of clients as long as the chatName is unique. Experiment with a three-way chat.

The consumer project is designed to continue listening for messages.

7. Press **Ctrl+C** at the command window to kill each client.

2. Create Multiple Consumers

Introduction

In this ActiveMQ exercise you learn how messages are handled and load-balanced when multiple consumers are connected to the same JMS queue.

The project has two parts:

- The consumer project starts a JMS MessageConsumer that reads messages as long as it can find one. For the purpose of this exercise, many consumers will be started in parallel to demonstrate the distribution of the messages.
- The producer project starts a JMS MessageProducer that creates a specified number of JMS messages and then stops and exits. The sending of messages is artificially slowed with 100ms "sleep" between messages so you can interrupt them when desired.

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.
     a. If you are using an OpenShift environment, use the correct connection details (URL, `admin` user ID/password) to connect to the OpenShift Enterprise JBoss Fuse cartridge.
2. Start the first consumer:
   a. Open a command line terminal (Windows or UNIX) and navigate to `2_Core_Features_Extensions/multiple-consumer`.
   b. Run this Maven command:

```
mvn -P consumer
```

```
[INFO]
[INFO] --- maven-jar-plugin:2.3.2:jar (default-jar) @ multiple-consumer ---
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default) @ multiple-consumer >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default) @ multiple-consumer <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default) @ multiple-consumer ---
22:25:38 INFO  Consumer NONAME: Consuming messages from
queue://test.queue.multiple.consumers with 120000ms timeout
```

3. Start the second consumer:
   a. Open a second terminal window and navigate to `2_Core_Features_Extensions/multiple-consumer`.
   b. Run this Maven command:

```
mvn -P consumer
```

```
[INFO]
[INFO] --- maven-jar-plugin:2.3.2:jar (default-jar) @ multiple-consumer ---
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default) @ multiple-consumer >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default) @ multiple-consumer <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default) @ multiple-consumer ---
22:26:43 INFO  Consumer NONAME: Consuming messages from
queue://test.queue.multiple.consumers with 120000ms timeout
```

4. Start the producer:
   a. Open a third terminal and navigate to `2_Core_Features_Extensions/multiple-consumer`.
   b. Run this Maven command:

```
mvn -P producer
```

5. On the Console, observe the following:
   - Each consumer is consuming unique messages.
   - A message consumed by the first consumer is not available for the second consumer, and vice versa.
   - Distribution is defined according to a round robin distribution.

There is no limit on how many consumers can be created. There is also no limit to the number of producers.

6. (Optional) Experiment with shutting down clients while the message processing is underway and observe what happens.

The consumer project is designed to continue listening for messages.

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

3. Create a Request/Reply Pattern

Introduction

In this lab exercise you create a request/reply topology by leveraging ActiveMQ. The JMS client sends an asynchronous request to a queue managed by the JBoss A-MQ broker, requesting that the JMS application consuming the message compute the factorial of the

specified number.

The processed request is a calculated BigDecimal value. It is published as a response into a reply queue as specified by the client ReplyTo JMS header property. The client listens on the reply queue to the messages published and correlates them using the CorrelationID JMS header issued with the original request message. If the message is correlated, then it is removed from a Java Collections object.

The project has two parts:

- The server project starts a JMS MessageConsumer that reads messages (requests) as long as it can find one. The server project also starts a JMS MessageProducer that writes messages (replies) to a reply channel, as specified by the `JMSReplyTo` property.
- The client project starts a JMS MessageProducer that creates the specified number of JMS messages (requests) and then stops and exits. The sending of messages is artificially slowed with 100ms "sleep" between messages so you can interrupt them when desired.

  The client project also starts a JMS MessageConsumer that reads messages (replies) from a temporary queue as long as it can find one or until the process exits.

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 an OpenShift environment, use the correct connection details (URL, `admin` user ID/password) to connect to the OpenShift Enterprise JBoss Fuse cartridge.
2. Starts the server:
   a. Open a command line terminal (Windows or UNIX) and navigate to `2_Core_Features_Extensions/multiple-consumer` directory.
   b. Run this Maven command:

   ```
   mvn -P server
   ```

3. Starts the client:
   a. Open a second terminal window and navigate to `2_Core_Features_Extensions/request-reply`.
   b. Run this Maven command:

   ```
   mvn -P client
   ```

4. View the request and subsequent asynchronous reply messages on the client console:

```
18:08:28 INFO  Sending to destination: queue://test.queue.request a request to
compute factorial(45)
18:08:28 INFO  factorial(45) =
119622220865480194561963161495657715064383733760000000000
18:08:30 INFO  Sending to destination: queue://test.queue.request a request to
compute factorial(50)
18:08:30 INFO  factorial(50) =
30414093201713378043612608166064768844377641568960512000000000000
```

5. Press **Ctrl+C** to exit.

---

4. Create a Transacted Chat

Introduction

This exercise is a modified version of the durable chat exercise. It shows how to use JMS transactions to write messages to a topic (commit) or undo them (roll-back). The messages are published as a batch (`group`) through the JMS transactional session created between the client and the broker.

The listener project starts a transacted chat client that catches user input from the command line and also listens for incoming messages on the chat room topic.

The user must either `COMMIT` the set of previous messages to send them out or `CANCEL` the set to drop the messages without sending them to the other chatters.

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 an OpenShift environment, use the correct connection details (URL, `admin` user ID/password) to connect to the OpenShift Enterprise JBoss Fuse cartridge.
2. Start the first chat client:
   a. Open a command line terminal (Windows or UNIX) and navigate to `2_Core_Features_Extensions/transacted-chat`.
   b. Run this Maven command:

```
mvn -P chatter -DchatName=<Any Name you Like, for example: chatterOne>
```

3. Start the second chat client:

   a. Open another terminal window and navigate to `2_Core_Features_Extensions/transacted-chat`.

   b. Run this Maven command:

```
mvn -P chatter -DchatName=<Any Name you Like (except the name used above), for
example: chatterTwo>
```

Now each chatter can add a few messages, and only if `COMMIT` is entered in a new line are the pending messages sent to the other chatters. If `CANCEL` is entered, the pending messages are dropped, but the user can continue to write and send new messages.

Sample output from first chatter

```
[INFO] --- exec-maven-plugin:1.2.1:java (default) @ transacted-chat ---

Transacted Chat application:
============================

The application user ONE connects to the broker.
The application will stage messages to the topic://transacted.chatter.room topic
until you either commit them or roll them back.
The application also subscribes to that topic to consume any committed messages
published there.

1. Enter text to publish and then press Enter to stage the message.
2. Add a few messages to the transaction batch.
3. Then, either:
     o Enter the text 'COMMIT', and press Enter to publish all the staged
messages.
     o Enter the text 'CANCEL', and press Enter to drop the staged messages
waiting to be sent.


test message to be cancelled
SEND >> 'ONE: test message to be cancelled'
CANCEL
Cancelling messages...
Staged messages have been cleared.
test message to be sent
SEND >> 'ONE: test message to be sent'
COMMIT
Committing messages...
Staged messages have all been sent.
RECEIVED >> 'ONE: test message to be sent'
```

Sample output from second chatter

- Only the `COMMIT` messages are seen. The rest were rolled back and not sent to other chat windows.

```
[INFO] --- exec-maven-plugin:1.2.1:java (default) @ transacted-chat ---

Transacted Chat application:
============================

The application user TWO connects to the broker.
The application will stage messages to the topic://transacted.chatter.room
topic until you either commit them or roll them back.
The application also subscribes to that topic to consume any committed
messages published there.

1. Enter text to publish and then press Enter to stage the message.
2. Add a few messages to the transaction batch.
3. Then, either:
     o Enter the text 'COMMIT', and press Enter to publish all the staged
messages.
     o Enter the text 'CANCEL', and press Enter to drop the staged messages
waiting to be sent.


RECEIVED >> 'ONE: test message to be sent'
```

4. To exit, press **Ctrl+C**.

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