Table of Contents

1. Advanced Features Lab

Goals

- Use advanced features of Apache Camel Technology
- Explore Transactions
- Create a new Apache Camel component
- Design an Apache Camel policy to change how a route is started and stopped
- Secure a Jetty endpoint to authenticate HTTP requests

Lab Assets

The lab exercises and solutions are available in the following zip archives:

- https://github.com/gpe-mw-training/camel-labs/archive/v0.3-exercise.zip
- https://github.com/gpe-mw-training/camel-labs/archive/v0.3-solution.zip

1.1. Transactional Client

The purpose of this exercise is to transform an existing Apache Camel JMS endpoint into a transactional client. The JMS component handles commit/rollback activity. If an error occurs during Exchange processing, the component directs ActiveMQ to move the JMS Message to the DLQ.

Three routes were created for this exercise:

- The first route polls a directory and moves files containing the XML payments into the `incomingPayments` queue.
- The second route consumes the JMS messages from the queue, converts the body to a string, transforms the XML string into a `Payment` POJO object and calls the `PaymentBean` object using the `validate` method. If the payment ID type is `???`, the `PaymentBean` class generates an error. If no error occurs, the route continues to process the Apache Camel Exchange. The Exchange (= Payment object) is

unmarshalled into a string and the result is published to an `outgoingPayments` queue.

- The third route consumes the messages from the `outgoingPayments` queue and creates files.

Follow these steps to complete the exercise:

1. In **JBoss Developer Studio**, use **Project Explorer** to open the `camel-jms-transaction` project.
2. Review the Spring Camel XML file and the `PaymentBean` class.
3. Add three new beans to the `META-INF/spring/spring-camel-context.xml` file:
   a. Add a `jmsConnectionFactory` bean to connect the JMS endpoint with the ActiveMQ Broker:
      - Create an `org.apache.activemq.ActiveMQConnectionFactory` bean.
      - Add the `brokerURL` parameter configured to: `tcp://localhost:61616?`.
      - Add a maximum redelivery policy to the URL definition `jms.redeliveryPolicy.maximumRedeliveries=3`.
      - Add a redelivery delay value to the URL definition `jms.redeliveryPolicy.initialRedeliveryDelay=500`.

   b. Add a `transactionManager` bean:
      - Use the **Spring JMS Transaction Manager** to create `org.springframework.jms.connection.JmsTransactionManager`.
      - Add a `connectionFactory` parameter configured to the `jmsConnectionFactory` bean you just created.

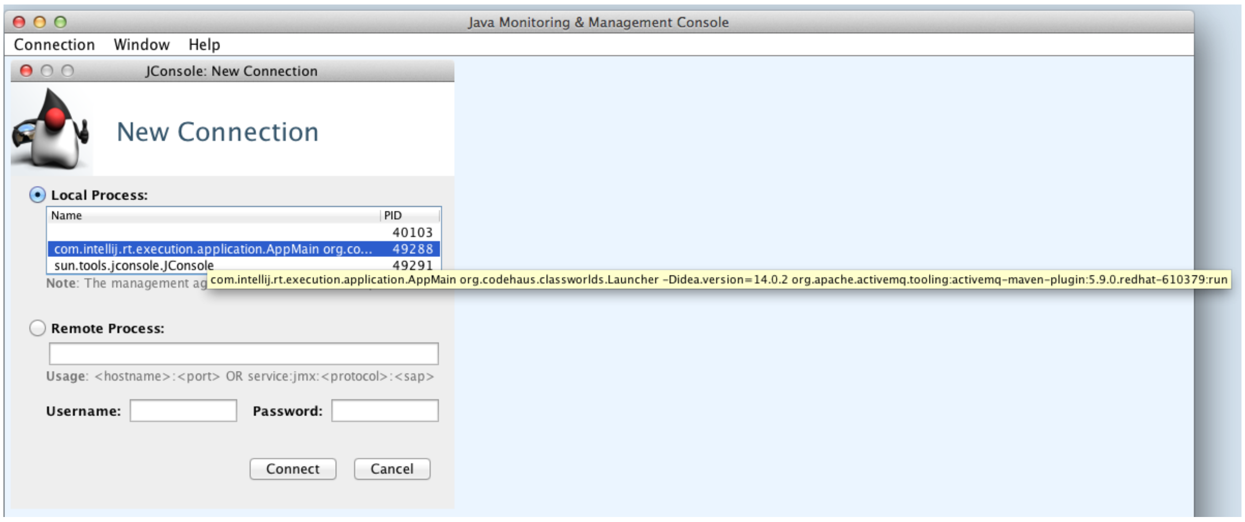
   c. Add an `activemq` bean:
      - Use the **Apache Camel JMS** component `org.apache.camel.component.jms.JmsComponent` to create the bean.
      - Add a `connectionFactory` parameter configured to the `jmsConnectionFactory` bean you created.
      - Add a `transactionManager` parameter configured to the `JmsTransactionManager` bean you created.
      - For each parameter, set the `transacted` property to `true`.
4. Compile the project using the `mvn clean install` command.
5. Test the project.

   a. Use the `mvn activemq:run` command to start the ActiveMQ broker.
   b. Open another command line console and use the `mvn camel:run` command to start the `camel-plugin` that runs the three routes.

6. Verify that a JMS message was rolled back and appears in the DLQ queue.
   a. Launch JConsole to access the broker.

      You can also use other JMX clients.

   b. Review the **Local Process** list and select the one that corresponds to **ActiveMQ**.



Java Monitoring & Management Console

JConsole: New Connection

New Connection

Local Process:

Name	PID
	40103
com.intellij.rt.execution.application.AppMain org.co...	49288
sun.tools.jconsole.JConsole	49291

```
com.intellij.rt.execution.application.AppMain org.codehaus.classworlds.Launcher -Didea.version=14.0.2 org.apache.activemq.tooling:activemq-maven-plugin:5.9.0.redhat-610379:run
```

Note: The management ag

Remote Process:

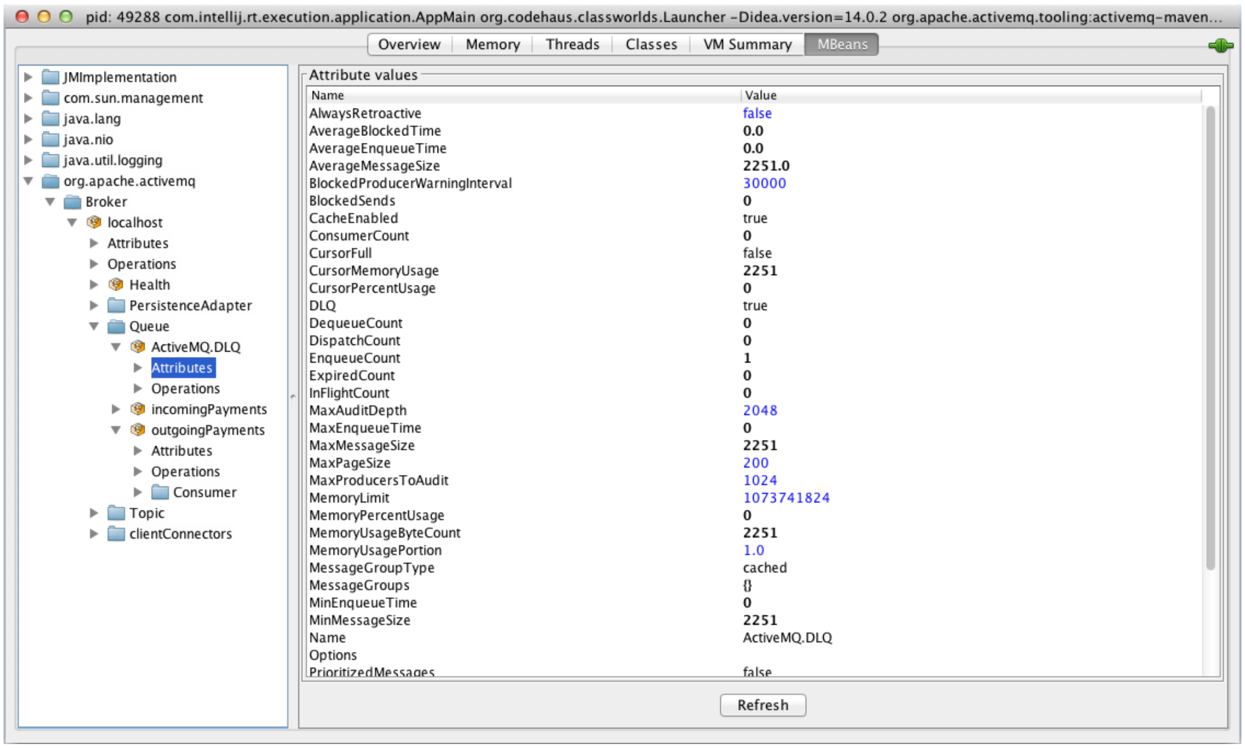
Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username:

Password:

Figure 1. Console - Local Processes

   c. Click the **MBeans** tab, and then expand the `ActiveMQ` tree.
   d. Select the `ActiveMQ.DLQ` queue.
   e. Confirm that one JMS message appears in the queue and contains the unknown payment `???`.

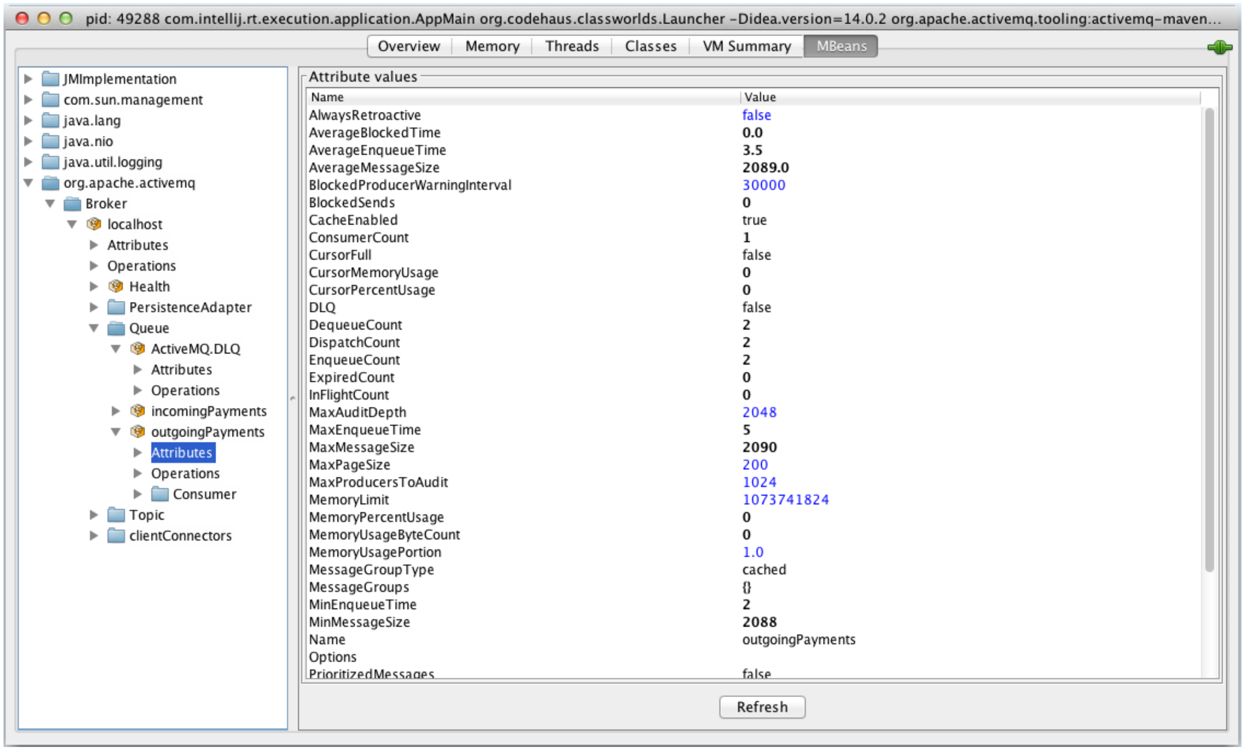

pid: 49288 com.intellij.rt.execution.application.AppMain org.codehaus.classworlds.Launcher -Didea.version=14.0.2 org.apache.activemq.tooling:activemq-maven...
Overview
Memory
Threads
Classes
VM Summary
MBeans
Attribute values
Refresh

```
JMImplementation
com.sun.management
java.lang
java.nio
java.util.logging
org.apache.activemq
  Broker
    localhost
      Attributes
      Operations
      Health
      PersistenceAdapter
      Queue
        ActiveMQ.DLQ
          Attributes
          Operations
        incomingPayments
        outgoingPayments
          Attributes
          Operations
          Consumer
      Topic
      clientConnectors
```

Name	Value
AlwaysRetroactive	false
AverageBlockedTime	0.0
AverageEnqueueTime	0.0
AverageMessageSize	2251.0
BlockedProducerWarningInterval	30000
BlockedSends	0
CacheEnabled	true
ConsumerCount	0
CursorFull	false
CursorMemoryUsage	2251
CursorPercentUsage	0
DLQ	true
DequeueCount	0
DispatchCount	0
EnqueueCount	1
ExpiredCount	0
InFlightCount	0
MaxAuditDepth	2048
MaxEnqueueTime	0
MaxMessageSize	2251
MaxPageSize	200
MaxProducersToAudit	1024
MemoryLimit	1073741824
MemoryPercentUsage	0
MemoryUsageByteCount	2251
MemoryUsagePortion	1.0
MessageGroupType	cached
MessageGroups	{}
MinEnqueueTime	0
MinMessageSize	2251
Name	ActiveMQ.DLQ
Options	
PrioritizedMessages	false

Figure 2. JConsole - DLQ Queue

f. Confirm that two JMS messages (= EUR and US payments) were created and appear in the `outgoingPayments` queue.


pid: 49288 com.intellij.rt.execution.application.AppMain org.codehaus.classworlds.Launcher -Didea.version=14.0.2 org.apache.activemq.tooling:activemq-maven...
Overview
Memory
Threads
Classes
VM Summary
MBeans
Attribute values
Refresh

```
JMImplementation
com.sun.management
java.lang
java.nio
java.util.logging
org.apache.activemq
  Broker
    localhost
      Attributes
      Operations
      Health
      PersistenceAdapter
      Queue
        ActiveMQ.DLQ
          Attributes
          Operations
        incomingPayments
        outgoingPayments
          Attributes
          Operations
          Consumer
      Topic
      clientConnectors
```

Name	Value
AlwaysRetroactive	false
AverageBlockedTime	0.0
AverageEnqueueTime	3.5
AverageMessageSize	2089.0
BlockedProducerWarningInterval	30000
BlockedSends	0
CacheEnabled	true
ConsumerCount	1
CursorFull	false
CursorMemoryUsage	0
CursorPercentUsage	0
DLQ	false
DequeueCount	2
DispatchCount	2
EnqueueCount	2
ExpiredCount	0
InFlightCount	0
MaxAuditDepth	2048
MaxEnqueueTime	5
MaxMessageSize	2090
MaxPageSize	200
MaxProducersToAudit	1024
MemoryLimit	1073741824
MemoryPercentUsage	0
MemoryUsageByteCount	0
MemoryUsagePortion	1.0
MessageGroupType	cached
MessageGroups	{}
MinEnqueueTime	2
MinMessageSize	2088
Name	outgoingPayments
Options	
PrioritizedMessages	false

Figure 3. JConsole - Outgoing Queue

1.2. Transacted Route

The purpose of this exercise is to extend the functionality you created in the last exercise by adding a transacted route to insert payment records into a database. In this exercise, you use an interceptor to identify when the route processor needs a transaction. The interceptor also notifies Apache Camel that a `BeanProcessor`, a nontransactional client, is included in the transaction flow. A `TransactionManager` handles the connection with a database. As in the last exercise, you use the JMS Transaction client to rollback messages not processed into a Dead Letter Queue.

In addition to what is described for the previous exercise, the use case for this exercise includes a `PaymentBean` processor that contains a Spring JDBC Template that inserts the individual payments into the Payments table of the H2, MySQL or PostgresQL database on your local machine.

In this exercise, the route is modified by the addition of an `IdempotentProcessor` which prohibits a message from being processed more than once. The `IdempotentProcessor` was configured to insert the key, which is the name of the file into the table `ProcessedPayments`.

Three routes were designed for this exercise:

- The first route polls a directory and move the files containing the XML payments into an `incomingTxPayments` queue.
- The second route consumes the JMS messages from the queue, and converts the message to a string.
    - The `idempotentProcessor` inserts a record into the database using the name of file assigned by Camel to the property `CamelFileNameOnly` as a key value. Also, the processor checks if the key already exists in the `ProcessedPayments` table.
    - Next, the string is converted from XML format to Java Beans using the JAXB marshaller (review the previous exercise for the full explanation).
    - The `insert` method of the `PaymentBean` processor is called by the Apache Camel route. The method uses the JDBC template of the Spring framework to insert a `Payment` record into the database. If the payment ID type is `???`, an error occurs and Spring rolls back the JDBC transaction.

      When an exception is generated, the JMS client is notified and the JMS Transaction client rolls back the JMS session. The JMS Broker moves the message to the DLQ.

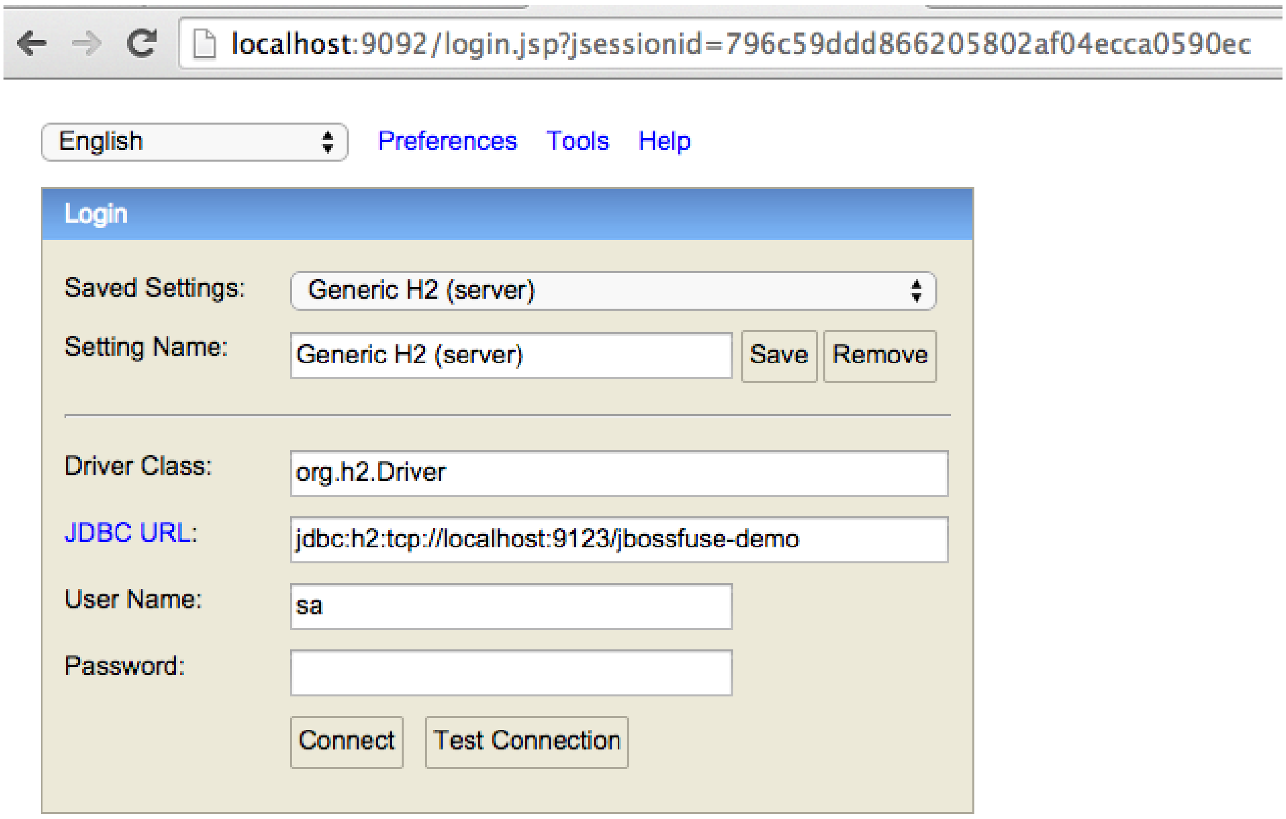
    - If no error occurs, the route continues to process the Apache Camel Exchange. The Exchange (= Payment object) is unmarshalled into a string and the result is published to an `outgoingPayments` queue.
- The third route consumes the messages from the `outgoingPayments` queue and creates files.

Follow these steps to complete the exercise:

1. In **JBoss Developer Studio**, use **Project Explorer** to open the `camel-jdbc-jms-transaction` project.
2. Review the Spring Camel XML file, and configurations of the `TransactionManagers`, `DataSource`, and the `PaymentBean` class.
3. Compile the project using the `mvn clean install` command.
4. Select the **H2** database to run this exercise.

   Because H2 is an embedded Java database, no installation is required on your local machine. Outside this lab environment, you can also use MySQL or PostgresQ databases. Scripts are provided in the `camel-jdbc-jms-transaction/src/main/resources/sql` folder, so that you can create the required tables.

5. Launch the H2 database using the `mvn -P start-h2` command.
6. Open the H2 web console using this address: `http://localhost:9092`.
7. For the **JDBC URL**, enter `jdbc:h2:tcp://localhost:9123/jbossfuse-demo`.


localhost:9092/login.jsp?jsessionid=796c59ddd866205802af04ecca0590ec

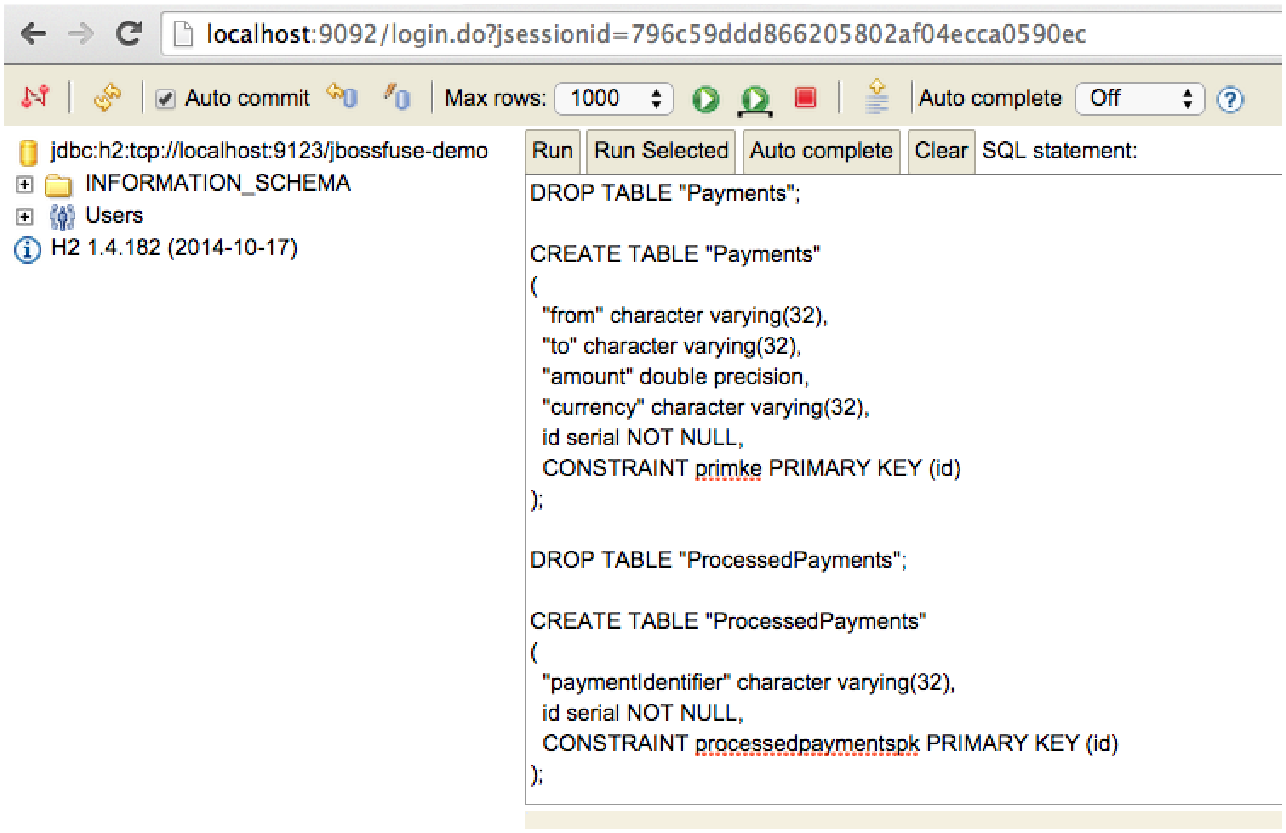
English | Preferences | Tools | Help

Login

Saved Settings:	Generic H2 (server)
Setting Name:	Generic H2 (server)
Driver Class:	org.h2.Driver
JDBC URL:	jdbc:h2:tcp://localhost:9123/jbossfuse-demo
User Name:	sa
Password:	

Figure 4. H2 - Configure

8. Click **Connect**.
9. Create the database using the `db-demo-setup-h2.sql` script, and then paste its content into the **SQL Statement** pane.



localhost:9092/login.do?jsessionid=796c59ddd866205802af04ecca0590ec

Auto commit Max rows: 1000 Auto complete Off

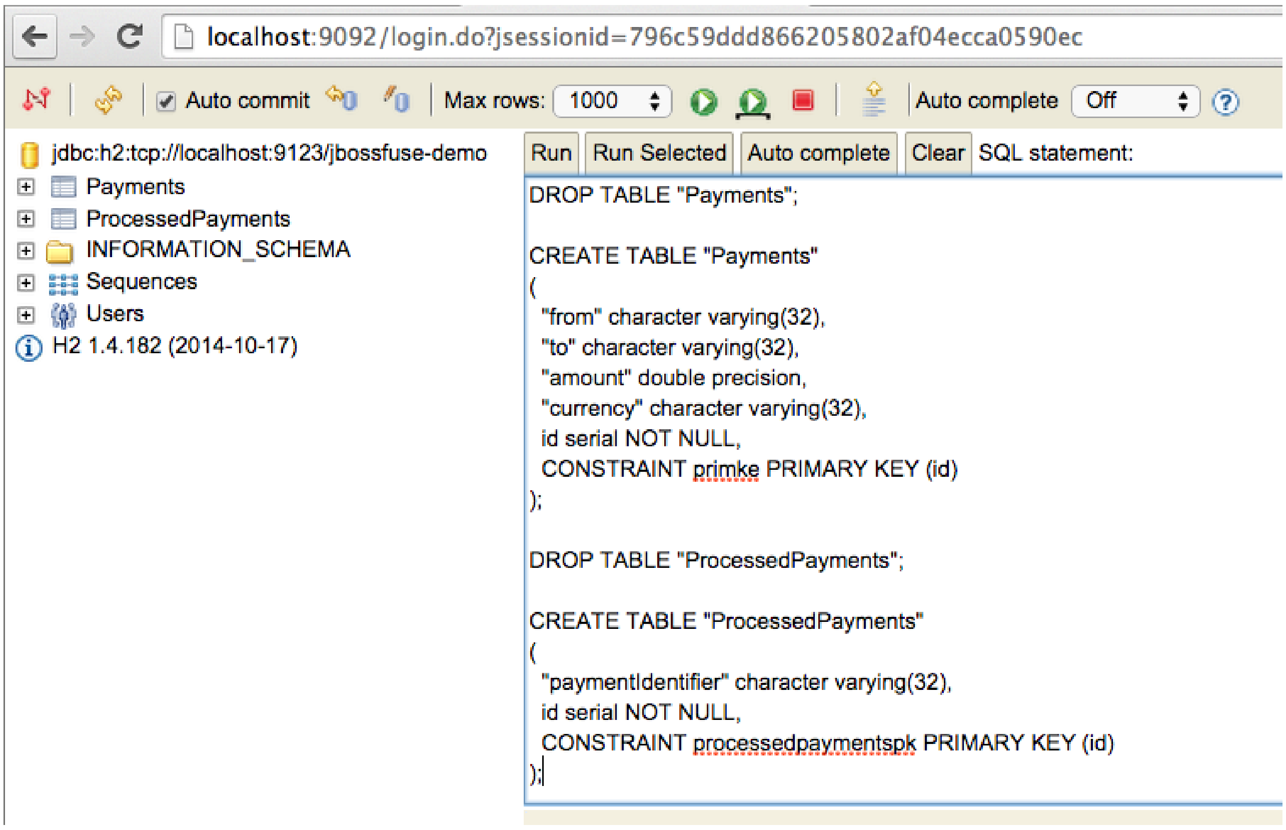
jdbc:h2:tcp://localhost:9123/jbossfuse-demo
INFORMATION_SCHEMA
Users
H2 1.4.182 (2014-10-17)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
DROP TABLE "Payments";

CREATE TABLE "Payments"
(
  "from" character varying(32),
  "to" character varying(32),
  "amount" double precision,
  "currency" character varying(32),
  id serial NOT NULL,
  CONSTRAINT primke PRIMARY KEY (id)
);

DROP TABLE "ProcessedPayments";

CREATE TABLE "ProcessedPayments"
(
  "paymentIdentifier" character varying(32),
  id serial NOT NULL,
  CONSTRAINT processedpaymentspk PRIMARY KEY (id)
);
```

Figure 5. H2 - Create Database

10. Click **Run** and verify that the tables were created.



localhost:9092/login.do?jsessionid=796c59ddd866205802af04ecca0590ec

Auto commit Max rows: 1000 Auto complete Off

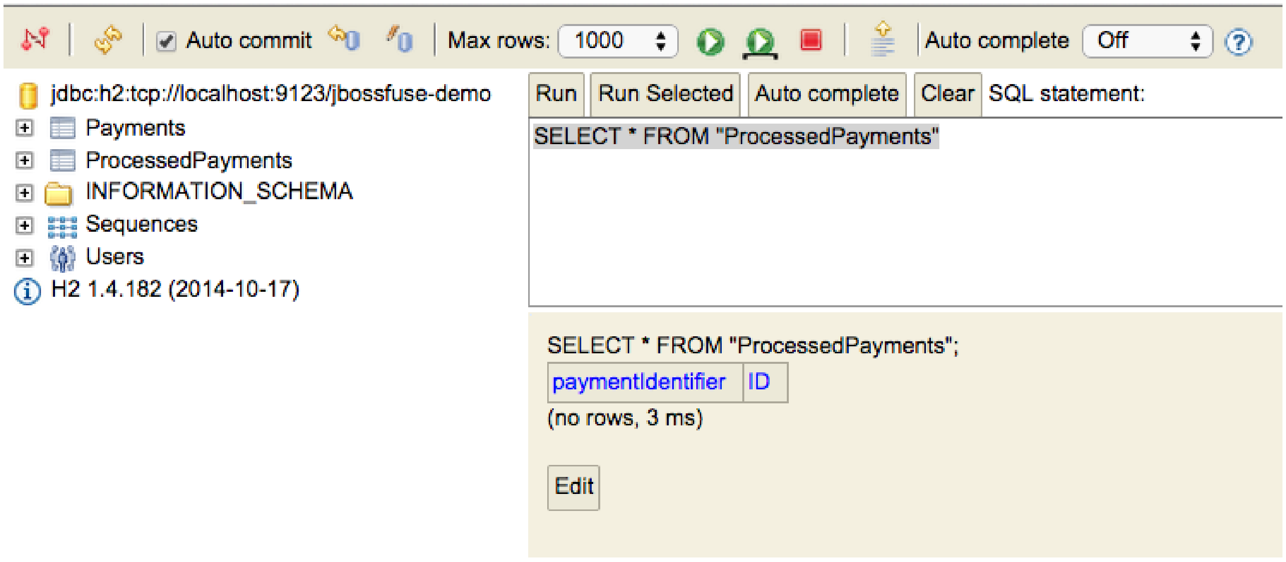
jdbc:h2:tcp://localhost:9123/jbossfuse-demo
Payments
ProcessedPayments
INFORMATION_SCHEMA
Sequences
Users
H2 1.4.182 (2014-10-17)

Run | Run Selected | Auto complete | Clear | SQL statement:

```
DROP TABLE "Payments";

CREATE TABLE "Payments"
(
  "from" character varying(32),
  "to" character varying(32),
  "amount" double precision,
  "currency" character varying(32),
  id serial NOT NULL,
  CONSTRAINT primke PRIMARY KEY (id)
);

DROP TABLE "ProcessedPayments";

CREATE TABLE "ProcessedPayments"
(
  "paymentIdentifier" character varying(32),
  id serial NOT NULL,
  CONSTRAINT processedpaymentspk PRIMARY KEY (id)
);
```

Figure 6. H2 - Database Tables

Alternatively, you can use SQL statements such as `SELECT * FROM "Payments"` and `SELECT * FROM "ProcessedPayments"`.



jdbc:h2:tcp://localhost:9123/jbossfuse-demo
- Payments
- ProcessedPayments
- INFORMATION_SCHEMA
- Sequences
- Users

H2 1.4.182 (2014-10-17)

```
SELECT * FROM "ProcessedPayments"
```

```
SELECT * FROM "ProcessedPayments";
```

paymentIdentifier	ID

(no rows, 3 ms)

Figure 7. H2 - Database Tables SQL

11. Test the project:

    a. Use the `mvn -P amq-run` command to start the ActiveMQ broker.

    b. Open another command line console and use the `mvn -P camel-h2-run` command to start the `camel-plugin` that runs the three routes.

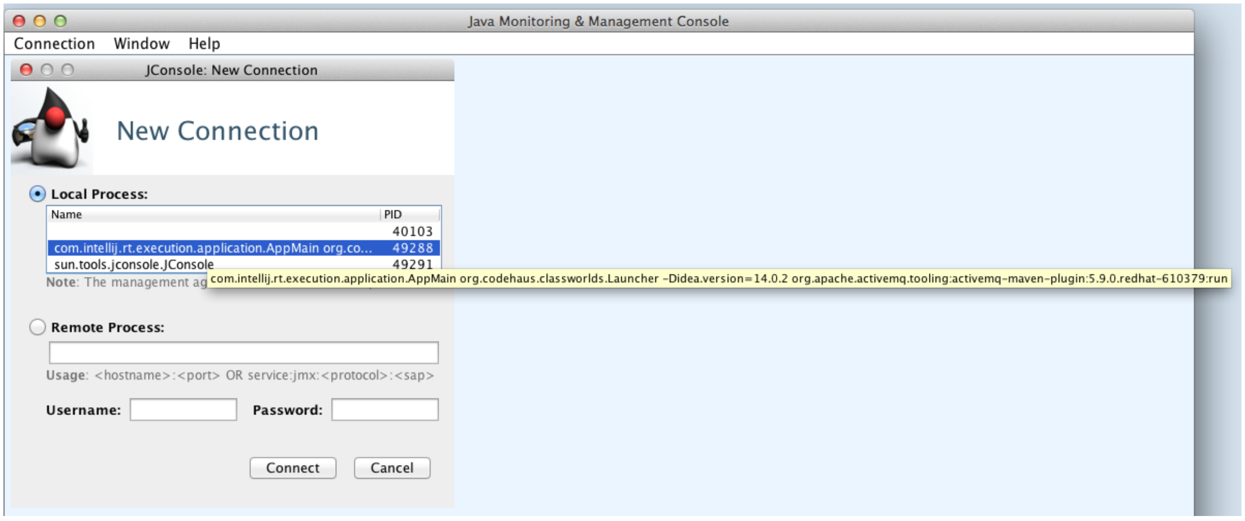
       Additional profiles are available to run Apache Camel with Postgresql or Mysql: `mvn -P camel-postgresql-run` or `mvn -P camel-mysql-run`.

    c. After Apache Camel raises an exception viewable in the console, verify that the messages and the JMS message were rolled back and appear in the Dead Letter Queue.

12. Verify that the payment for this currency type was not saved to the database:

    a. Launch JConsole (or another JMX client) to access the broker.

    b. Review the **Local Process** list and select the one that corresponds to ActiveMQ.



[● Local Process:]

Name	PID
	40103
com.intellij.rt.execution.application.AppMain org.co...	49288
sun.tools.jconsole.JConsole	49291

Note: The management ag

```
com.intellij.rt.execution.application.AppMain org.codehaus.classworlds.Launcher -Didea.version=14.0.2 org.apache.activemq.tooling:activemq-maven-plugin:5.9.0.redhat-610379:run
```

Remote Process:

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

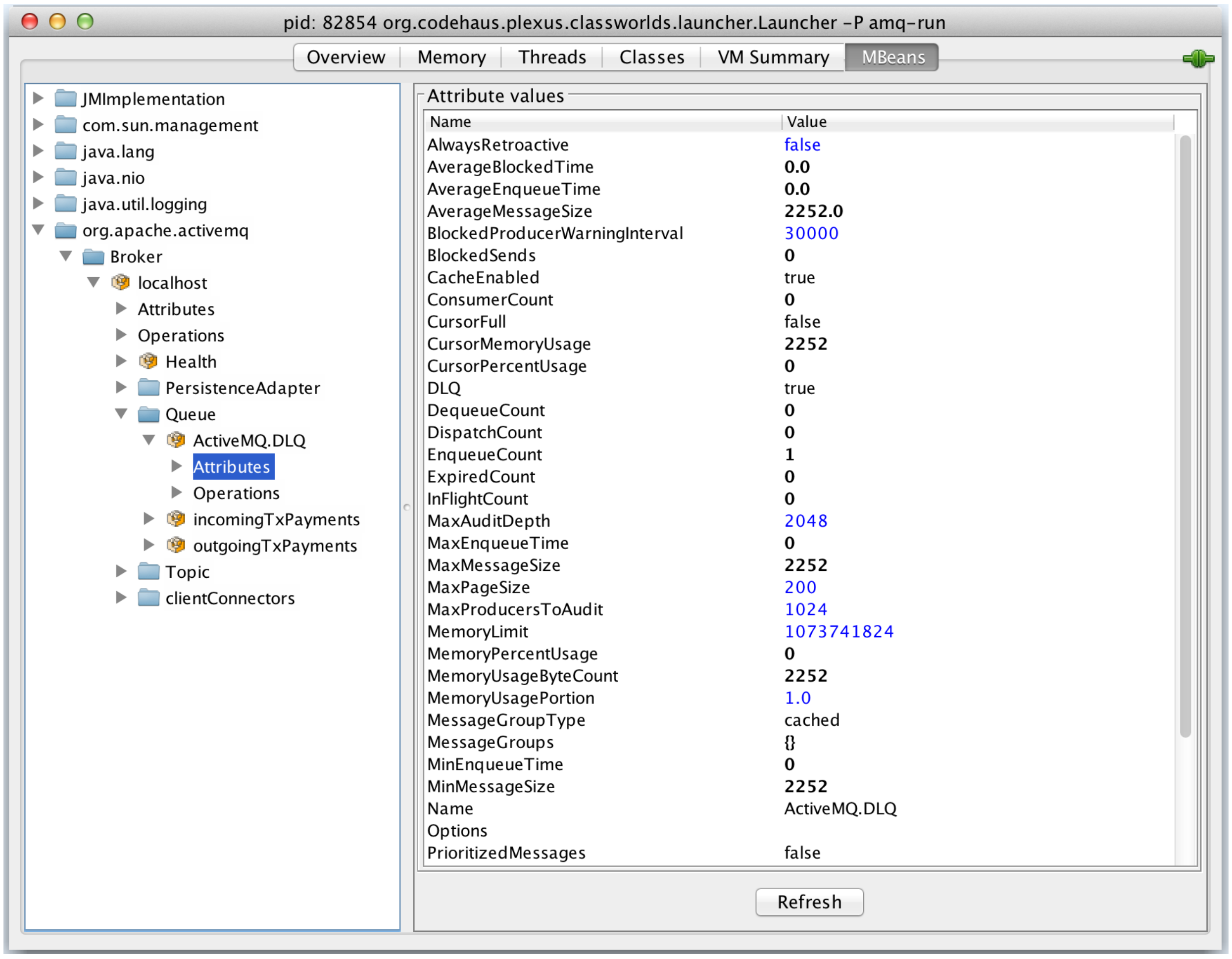
Username: Password:

Figure 8. Console - Local Processes

c. Click the **MBeans** tab, and then expand the `ActiveMQ` tree.

d. Select the `ActiveMQ.DLQ` queue.

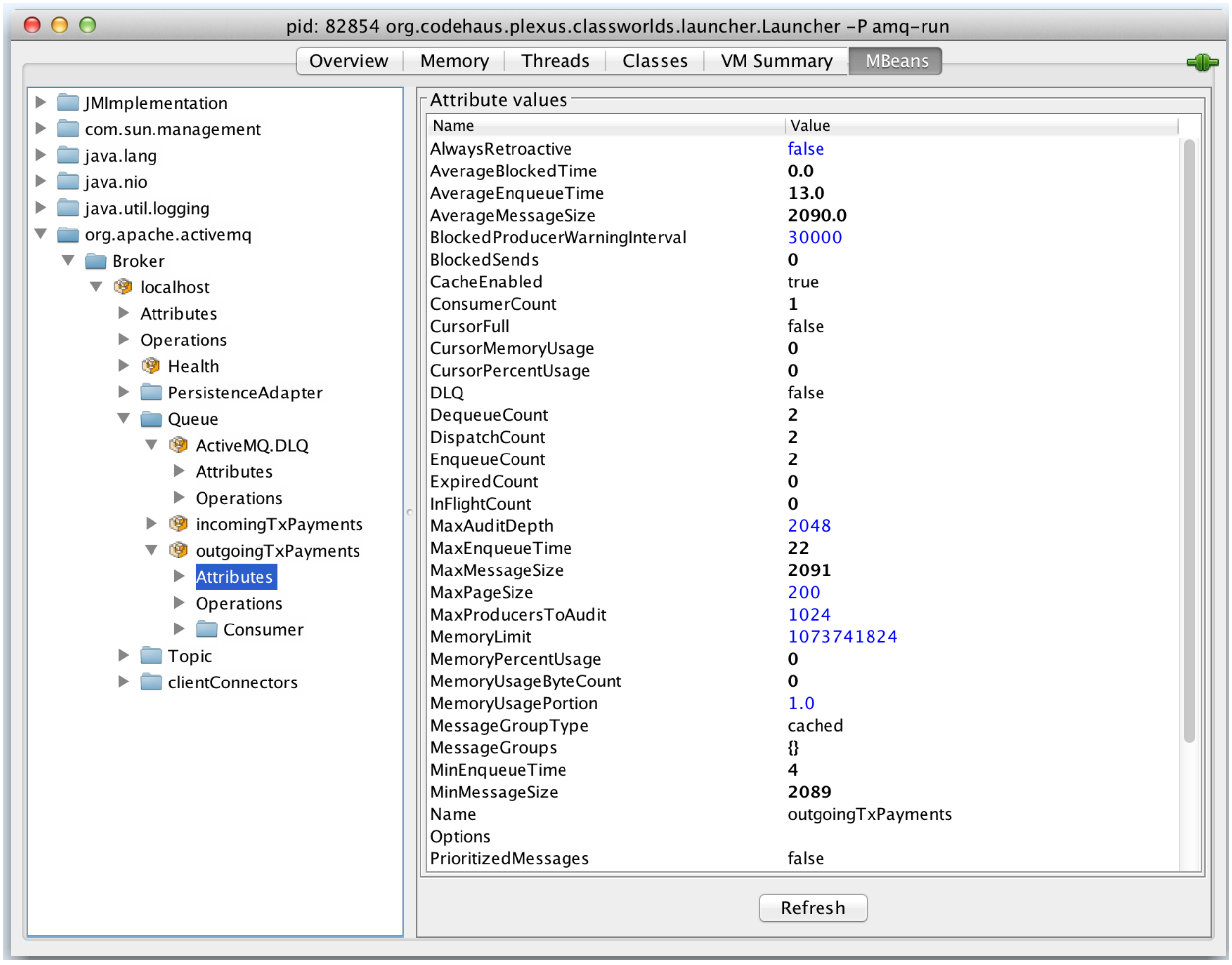
e. Confirm that one JMS message appears in the queue and contains the unknown payment `???`.



Name	Value
AlwaysRetroactive	false
AverageBlockedTime	0.0
AverageEnqueueTime	0.0
AverageMessageSize	2252.0
BlockedProducerWarningInterval	30000
BlockedSends	0
CacheEnabled	true
ConsumerCount	0
CursorFull	false
CursorMemoryUsage	2252
CursorPercentUsage	0
DLQ	true
DequeueCount	0
DispatchCount	0
EnqueueCount	1
ExpiredCount	0
InFlightCount	0
MaxAuditDepth	2048
MaxEnqueueTime	0
MaxMessageSize	2252
MaxPageSize	200
MaxProducersToAudit	1024
MemoryLimit	1073741824
MemoryPercentUsage	0
MemoryUsageByteCount	2252
MemoryUsagePortion	1.0
MessageGroupType	cached
MessageGroups	{}
MinEnqueueTime	0
MinMessageSize	2252
Name	ActiveMQ.DLQ
Options	
PrioritizedMessages	false

Figure 9. JConsole - DLQ Transaction

f. Confirm that two JMS messages (= EUR and US payments) were created and appear in the `outgoingPayments` queue.

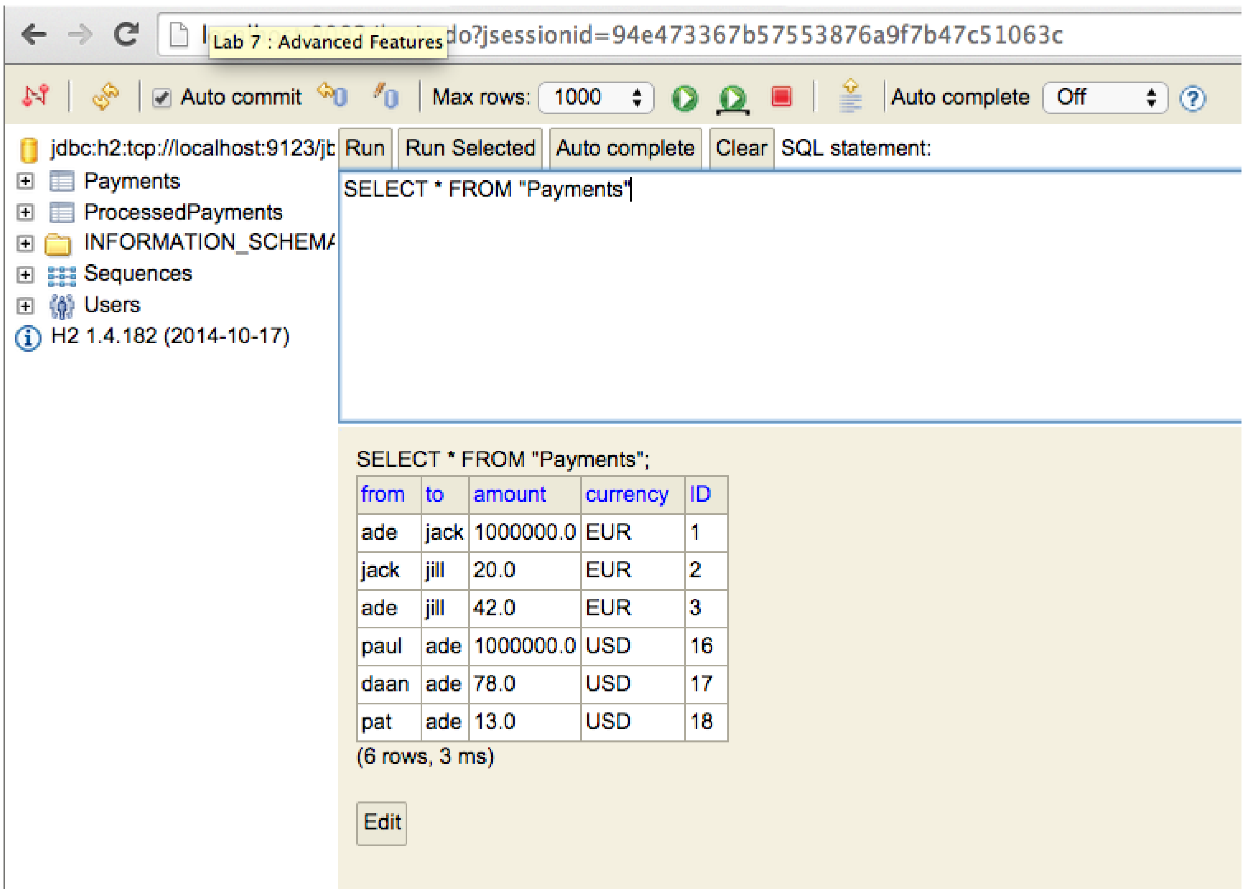


Name	Value
AlwaysRetroactive	false
AverageBlockedTime	0.0
AverageEnqueueTime	13.0
AverageMessageSize	2090.0
BlockedProducerWarningInterval	30000
BlockedSends	0
CacheEnabled	true
ConsumerCount	1
CursorFull	false
CursorMemoryUsage	0
CursorPercentUsage	0
DLQ	false
DequeueCount	2
DispatchCount	2
EnqueueCount	2
ExpiredCount	0
InFlightCount	0
MaxAuditDepth	2048
MaxEnqueueTime	22
MaxMessageSize	2091
MaxPageSize	200
MaxProducersToAudit	1024
MemoryLimit	1073741824
MemoryPercentUsage	0
MemoryUsageByteCount	0
MemoryUsagePortion	1.0
MessageGroupType	cached
MessageGroups	{}
MinEnqueueTime	4
MinMessageSize	2089
Name	outgoingTxPayments
Options	
PrioritizedMessages	false

Figure 10. JConsole - DLQ Outgoing Transactions

g. Open the **H2 Web Console** and connect to the database.

h. Run a `SELECT * FROM "Payments"` query to verify that six individual payments (EUR and USD) were created.



```
SELECT * FROM "Payments"
```

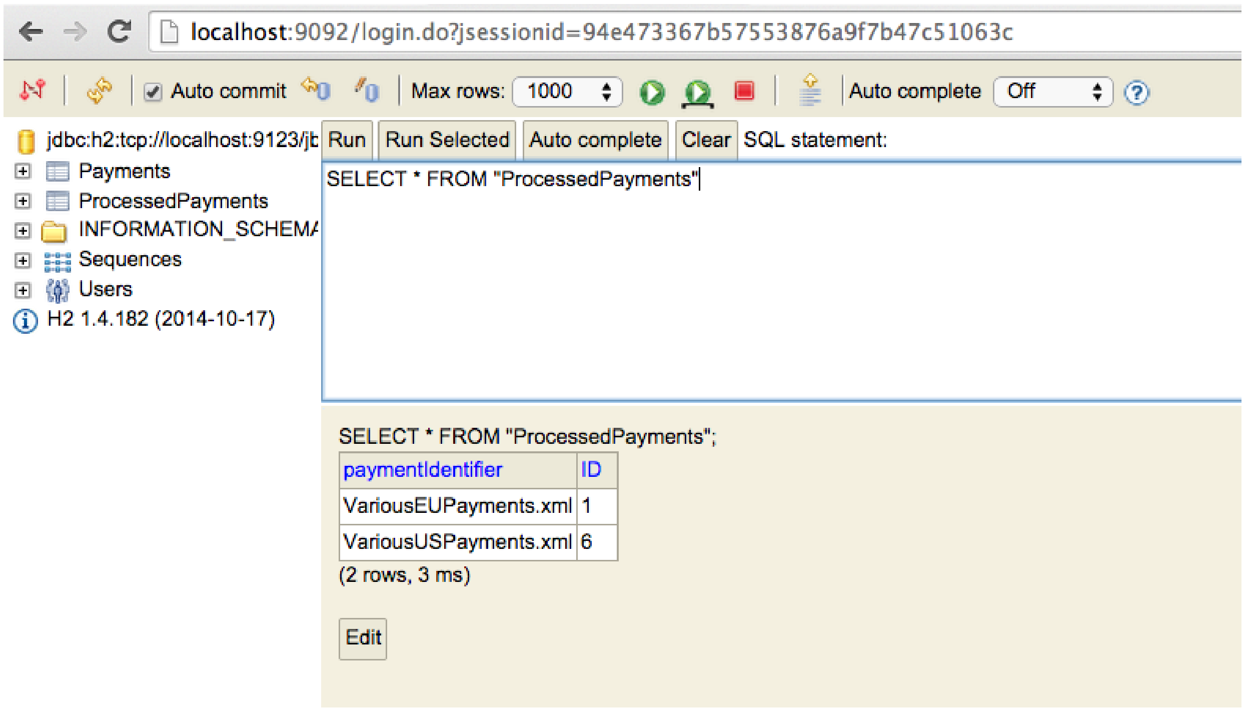
SELECT * FROM "Payments";

from	to	amount	currency	ID
ade	jack	1000000.0	EUR	1
jack	jill	20.0	EUR	2
ade	jill	42.0	EUR	3
paul	ade	1000000.0	USD	16
daan	ade	78.0	USD	17
pat	ade	13.0	USD	18

(6 rows, 3 ms)

Figure 11. H2 - Individual Payments

i. Run a `SELECT * FROM "ProcessedPayments"` query to confirm that only two payment files were processed.



```
SELECT * FROM "ProcessedPayments"
```

SELECT * FROM "ProcessedPayments";

paymentIdentifier	ID
VariousEUPayments.xml	1
VariousUSPayments.xml	6

(2 rows, 3 ms)

Figure 12. H2 - Processed Payments

1.3. Create a New Component

The purpose of this exercise is to create and unit test a new Apache Component, and to confirm that it works. You create a component called `MyComponent` that contains an endpoint, a producer, and a consumer. The consumer extends the `ScheduledPollConsumer` class to automatically generate and poll an exchange. As in the previous exercises, you begin with a skeleton project and add to it.

Follow these steps to complete the exercise:

1. In **JBoss Developer Studio**, use **Project Explorer** to open the `camel-mycomponent` project.
2. Review the skeleton project code: `MyComponent`, `MyComponentConsumer`, `MyComponentEndpoint`, and `MyComponentProducer`.
3. Create a file named `mycomponent` in the `src/main/resources/META-INF/services/org/apache/camel/component` directory.
4. Add the name of the class where the component is declared to the file. When Apache Camel looks for a component, it uses this file to load the class of the component.

   ```
   class=com.redhat.gpe.training.camel.MyComponent
   ```

5. Open the `MyComponent` Java class and in the `createEndpoint()` method, add the missing code to instantiate the `MyComponentEndpoint`, which calls the utility method `setProperties()` to pass the parameters to the bean endpoint in order to set fields and return the endpoint.
6. Review, but do not change, the `MyComponentEndpoint` code, noting how it instantiates the `MyComponentProducer` and `MyComponentConsumer` components.
7. Update the `MyComponentConsumer` code to create an exchange within the `poll()` method and to add a `Hello World!` body message.

   To help you, you can create an exchange using the field endpoint of the class.

8. Enrich the content of the exchange so that the `producer` receives it when it is called within the route:
   a. Open the `MyComponentProducer` class and in the `process()` method, add a new header to the exchange received with the name `MyHeader` and the value `foo`.
   b. Append the following value: `" and the Teacher is crazy !"` to the `Body` received from the exchange.
   c. Compile the project using the `mvn clean compile -DskipTest="true"` command.

   d. Verify that there are no Java compilation errors.

9. If there are no compilation errors, launch the test to verify that your project works correctly.

1.4. Route Policy

The purpose of this exercise is to add a `RoutePolicy` to a route in order to manage its `start/stop`, and to send a body message with the keyword "STOP" to suspend the consumption of this route. Also, you create an exception that is intercepted and `stop` the route sending the messages to this route.

These two routes are used during this exercise:

Route containing the policy

```
from("direct:foo")
   .routePolicy(policy)
   .log("Route direct:foo has been called with the Body : ${body}");
```

Route triggering the messages

```
from("timer:managed").routeId("timer-managed-route")
     .setBody().constant("Hello World")
          .log("Route 'direct:foo' is called")
          .to("direct:foo")
     .setBody().constant("STOP")
          .log("Route 'direct:foo' will be stopped")
          .to("direct:foo")
     .setBody().constant("Hello World")
          .log("Exception will be thrown as the route/consumer has been stopped during
the previous step !")
          .to("direct:foo") ;
```

Exception handling

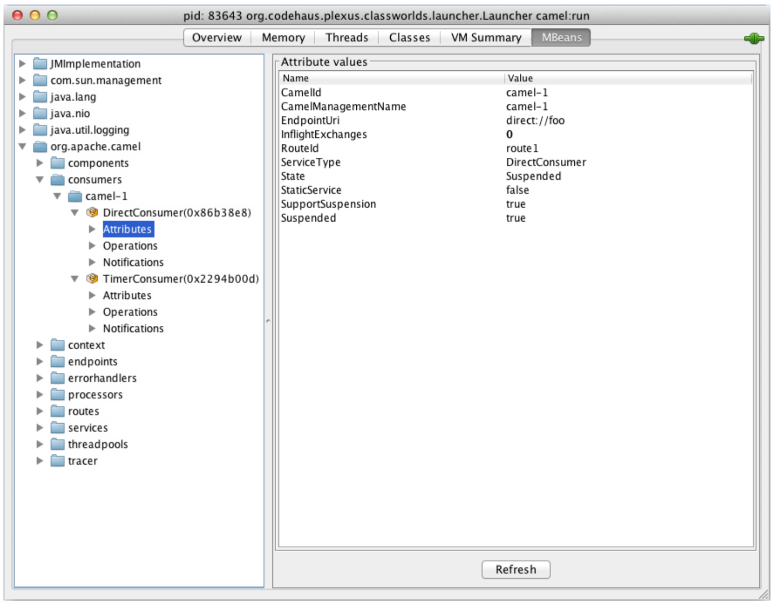
```
 onException(DirectConsumerNotAvailableException.class)
         .handled(true)
         .log("Route 'direct:foo' is suspended so we will close too the consumer of the
timer-managed-route !")
         .process(
         new Processor() {
             @Override
             public void process(Exchange exchange) throws Exception {
                exchange.getContext().getRoute("timer-managed-
route").getConsumer().stop();
             }
         }
 );
```

Follow these steps to complete the exercise:

1. In **JBoss Developer Studio**, use **Project Explorer** to open the `camel-routepolicy` project.
2. Review the code of the skeleton project: `ManagedRoute` and `MyCustomRoutePolicy`.
3. Design a policy to handle the `direct:foo` route.
   a. Open the class `MyCustomRoutePolicy` and override the code of the event/method `onExchangeBegin`. Inspect other available methods by moving to the parent class.
   b. Modify the code to stop the route if the Body message received equals `STOP`.

      To stop the route, use the `stopConsumer(route.getConsumer());` method.

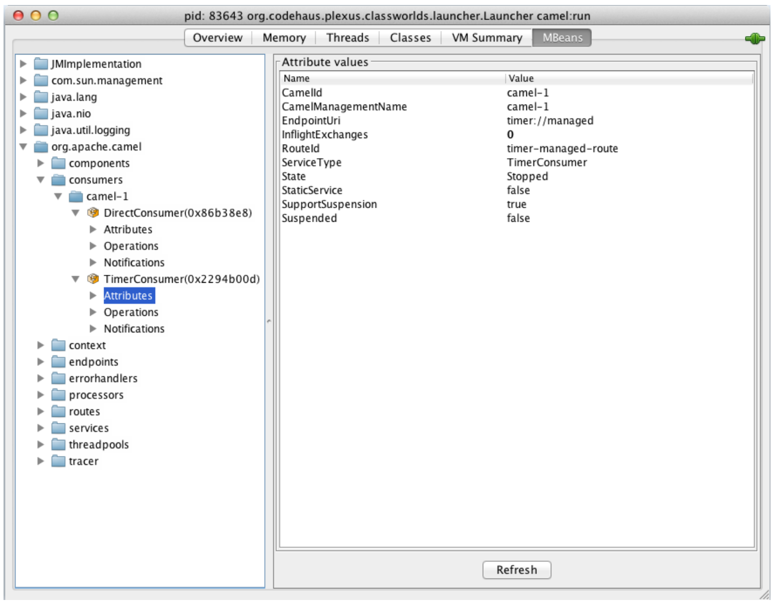
4. Add this route policy to the route you want to manage:
   a. Open the `ManagedRoute` class.
   b. Instantiate the `MyCustomRoutePolicy` class and assign the object to the `policy` field.
   c. Add the required DSL after the `from("direct:foo")` endpoint definition to refer to `RoutePolicy`.
5. Compile the project using the `mvn clean compile` command.
6. Start the Maven Camel goal to run the routes: `mvn camel:run`
7. Check the log generated in the console.
8. Open **JConsole** to verify that the consumer `direct:foo` is suspended.



Name	Value
CamelId	camel-1
CamelManagementName	camel-1
EndpointUri	direct://foo
InflightExchanges	0
RouteId	route1
ServiceType	DirectConsumer
State	Suspended
StaticService	false
SupportSuspension	true
Suspended	true

Figure 13. JConsole - Route Policy 1

9. Verify that the `timer:managed-route` consumer is stopped.

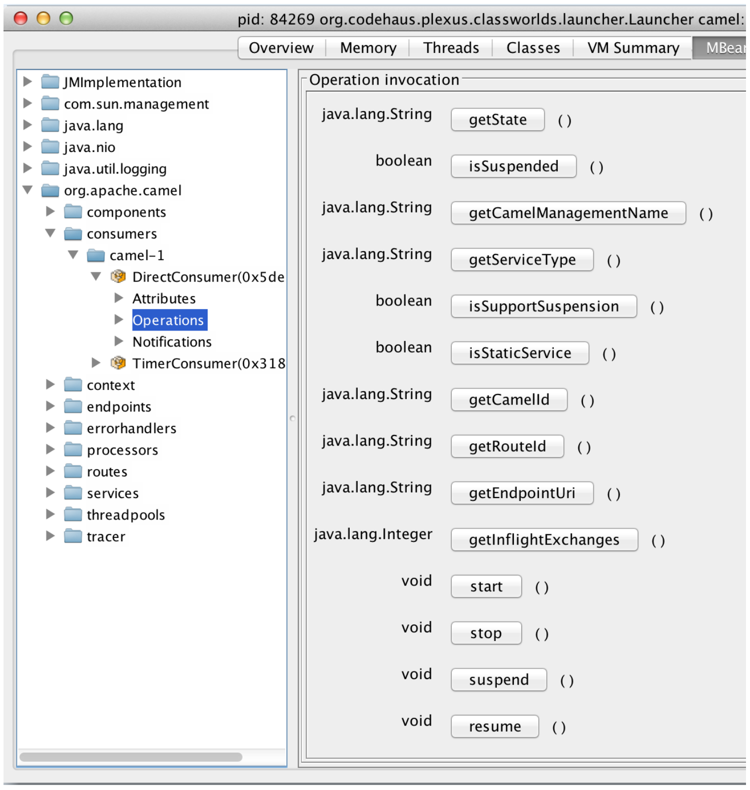


Name	Value
CamelId	camel-1
CamelManagementName	camel-1
EndpointUri	timer://managed
InflightExchanges	0
RouteId	timer-managed-route
ServiceType	TimerConsumer
State	Stopped
StaticService	false
SupportSuspension	true
Suspended	false

Figure 14. JConsole - Route Policy 2

10. Use JConsole to resume or restart the consumers:

    a. Click the **resume** operation of the `DirectConsumer`.



java.lang.String	getState	( )
boolean	isSuspended	( )
java.lang.String	getCamelManagementName	( )
java.lang.String	getServiceType	( )
boolean	isSupportSuspension	( )
boolean	isStaticService	( )
java.lang.String	getCamelId	( )
java.lang.String	getRouteId	( )
java.lang.String	getEndpointUri	( )
java.lang.Integer	getInflightExchanges	( )
void	start	( )
void	stop	( )
void	suspend	( )
void	resume	( )

Figure 15. JConsole - Route Policy Resume

    b. Click the **start** operation of the `TimerConsumer`.

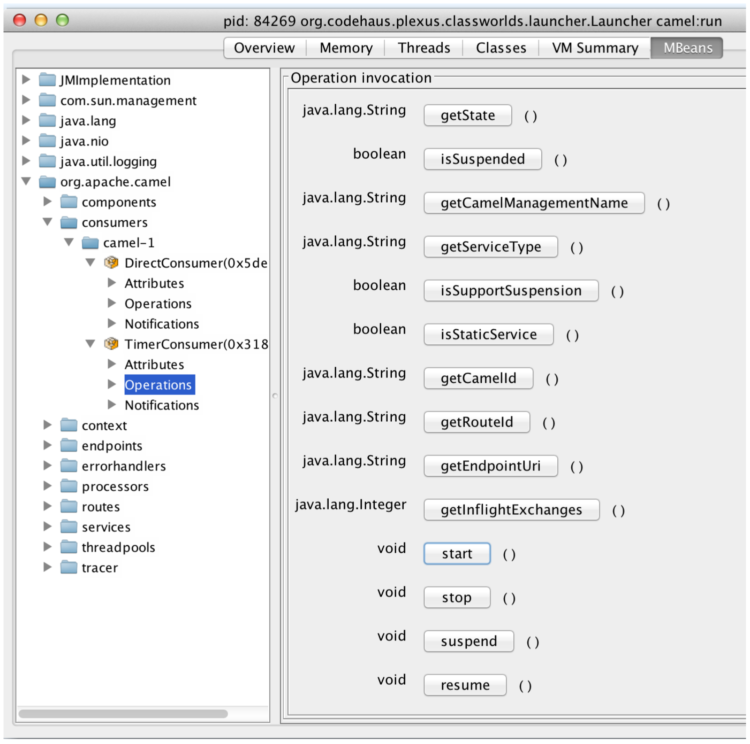


Figure 16. JConsole - Route Policy Start

c. Check the results in the console.

```
2014-12-23 16:49:28,187 [timer://managed] INFO  timer-managed-route            -
Route 'direct:foo' is called
2014-12-23 16:49:28,187 [timer://managed] INFO  route1                         -
Route direct:foo has been called with the Body : Hello World
2014-12-23 16:49:28,187 [timer://managed] INFO  timer-managed-route            -
Route 'direct:foo' will be stopped
2014-12-23 16:49:28,187 [timer://managed] INFO  route1                         -
Route direct:foo has been called with the Body : STOP
2014-12-23 16:49:28,187 [timer://managed] INFO  timer-managed-route            -
Exception will be thrown as the route/consumer has been stopped during the
previous step !
2014-12-23 16:49:28,188 [timer://managed] INFO  timer-managed-route            -
Route 'direct:foo' is suspended so we will close too the consumer of the timer-
managed-route !
```

1.5. Secure an Endpoint

The purpose of this exercise is to secure a Jetty endpoint to authenticate a user of an HTTP request using the basic authentication mechanism. The Jetty web project provides a `ConstraintSecurityHandler` class that you reuse here to set up the authentication solution. You assign the `SecurityHandler` to the URL of the Jetty endpoint. This enables Apache Camel to set up a Jetty Secured Server.

This exercise uses the following route:

```
    from("jetty://http://localhost:9191/demo?handlers=myAuthHandler")
        .transform(constant("<html><body><p>Bye World</p></body></html>"));
```

The route exposes a Jetty web container, and if the user is correctly authenticated, it returns the following HTML response: `<html><body><p>Bye World</p></body></html>`.

The user and password credentials used to authenticate the HTTP users are available in the `src/main/resources/myRealm.properties` file.

Follow these steps to complete the exercise:

1. In **JBoss Developer Studio**, use **Project Explorer** to open the `camel-security` project.
2. Review the code of the skeleton project components: `JettySecuredRoute` and `MySecurityHandler`.
3. Implement the route (defined above) in the `JettySecuredRoute` class.
4. Bind the `ConstraintSecurityHandler` object returned by the `generate()` method of the `MySecurityHandler` class with the `myAuthHandler` key into the `JettySecuredRoute` Main class. Camel uses this binding to search for the object associated with this key in the registry.
5. Start the Apache Camel Route using the `mvn camel:run` command.
6. After you start the route, run the following curl commands to test the endpoint and confirm that it returns the appropriate response:
   a. Use the following command to see the response with the correct user/password:

      ```
      curl --user donald:duck http://localhost:9191/demo
      ```

      The HTTP response should contain: `Bye World`

      ```
      <html><body><p>Bye World</p></body></html>
      ```

   b. Use the following command to see the response with the incorrect user/password:

      ```
      curl --user mickey:mouse http://localhost:9191/demo
      ```

      The HTTP response should contain: `HTTP ERROR: 401`

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=ISO-8859-1"/>
<title>Error 401 Unauthorized</title>
</head>
<body>
<h2>HTTP ERROR: 401</h2>
<p>Problem accessing /demo. Reason:
<pre>    Unauthorized</pre></p>
<hr /><i><small>Powered by Jetty://</small></i>
</body>
</html>
```

Last updated 2015-11-12 12:04:12 EST