Table of Contents

1. Camel Project Development Lab

Goals

- Develop an Apache Camel project using Maven archetypes and various DSLs such as Java, XML Spring, or XML Blueprint.
- Log and trace information about the Exchanges.
- Design JUnit tests for the project with a mock endpoint and assertions.
- Debug an integration project.

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. Java DSL

The goal of this exercise is to create an Apache Camel project in Java using a Main class. You register a `MyRouteBuilder` class that implements the following code snippet, and then you run it locally using the the Maven plugin `exec:java`:

```
from_Timer("fired_every5s_delay_of_1s")
  .set_Body("Student")
  .call_A_Bean("toSayHello")
  .log_Message(">> a Camel exercise - $Body ")
```

The code provided is in Java DSL and not Camel DSL.

Follow these steps to complete the Java DSL exercise:

1. In JBoss Developer Studio, use **Project Explorer** to open the `camel-standalone` project.
2. Create a `MyBean` class that contains a `sayHello` method and returns a message as a String. Use `@Body String content` as the input parameter. The `@Body` is an Apache Camel annotation.
3. Create a `CamelStandAloneApp` class in the package directory `com/redhat/gpe/training/camel`.
4. Add a static void Main method.

5. Instantiate the `Camel Main` class (available in the `org.apache.camel.main` package) in the body of the Main method `Main main = ...`.
6. Create the `MyRouteBuilder` class, and make sure you extend it with the `RouteBuilder` class.
7. Add the provided code to the `Configure` method.
8. Register the `MyRouteBuilder` class to the Main object.
9. Enable hangup support, which allows you to terminate the Camel project from a virtual terminal using **Ctrl+C**. Name this method `Main.enableHangupSupport()`.
10. Call the `Main.run()` method.
11. Right-click the Apache Camel route and select **Run as** → **Java Application**.
12. Press **Ctrl+C** to exit the route.
13. Add the `exec:java` plug-in to the `pom.xml` file of the project to enable execution from the command line using `mvn exec:java`.
14. Specify the Main class to be used as a configuration parameter `com.redhat.gpe.training.camel.CamelStandaloneApp`.

The `camel-standalone` project imported into JBoss Developer Studio contains the skeleton of the project. The Apache Maven `pom.xml` file already includes the necessary dependencies.

1.2. Spring XML DSL

The goal of this exercise is to create a standalone Apache Camel project in Spring. This exercise is similar to the one you just completed, but instead of using Java DSL, you use the XML DSL language supported by the Spring framework.

Also, the code snippet used in this exercise is similar to the code snippet used in the previous exercise. However, in this exercise the code resides in a Spring Main class (`org.apache.camel.spring.Main`) that accepts the location of the Spring XML file containing the beans definition as a parameter. And in this exercise, you launch the Camel route using the `camel:run` plug-in instead of the Maven `exec:java` plug-in.

This plug-in is also the plug-in used by JBoss Developer Studio when you right-click the Camel route and select **Run As** → **Camel Local Context**.

Follow these steps to complete the Spring XML DSL exercise:

1. In JBoss Developer Studio, use **Project Explorer** to open the `camel-spring-standalone` project.
2. Create a `MyBean` class that contains a `sayHello` method and returns a message as a String. Use `@Body String content` as the input parameter. The `@Body` is an Apache Camel annotation.
3. Create a `SpringMainApp` class in the `com/redhat/gpe/training/camel` package directory .
4. Add a static void Main method.
5. Instantiate the Main class (available in the package `org.apache.camel.spring.Main`) in the body of the Main method `Main main = ...`.

6. Enable hangup support, which allows you to terminate the Camel project from a virtual terminal using **Ctrl+C**. Name this method `Main.enableHangupSupport()`.
7. Add the location path of the Spring XML file `main.setApplicationContextUri("META-INF/spring/spring-camel-context.xml");`.
8. Call the `Main.run()` method.
9. Create a `spring-camel-context.xml` file in the `src/main/resources/META-INF/spring` directory using this code snippet:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://camel.apache.org/schema/spring
         http://camel.apache.org/schema/spring/camel-spring.xsd">
</beans>
```

10. Define a bean with the `id="myBean"` (i.e., class name of the bean you created) with appropriate `<bean>` tags in the Spring XML file.
11. Add the tag `<camelCon text xmlns="http://camel.apache.org/schema/spring">` to the Apache Camel Route.
12. Right-click the Apache Camel project and select **Run as** → **Java Application** to launch the project.
13. Press **Ctrl+C** to exit the route.
14. In the `pom.xml` file, specify the Apache Maven plug-in.
15. Specify the location of the Spring XML file(s) to use:

```
<configuration>
   <applicationContextUri>META-INF/spring/*.xml</applicationContextUri>
</configuration>
```

16. Run the project in a command line terminal using the `mvn camel:run` command.

1.3. Web

The goal of this exercise is to create an Apache Camel project and package it as a WAR file to be deployed in Apache Tomcat or JBoss Wildfly containers. To achieve this goal, you rely on the Spring Framework to start Spring's root `WebApplicationContext` using a `ServletContextListener` (such as `org.springframework.web.context.ContextLoaderListener`) that you define in the `web.xml` file.

Follow these steps to complete the web exercise:

1. In JBoss Developer Studio, use **Project Explorer** to open the `camel-web` project.
2. Edit the `src/main/webapp/WEB-INF/web.xml` file and add the Spring `org.springframework.web.context.ContextLoaderListener` class as a `<listener>` :

```
<listener>
```

```
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
 </listener>
```

3. In `web.xml`, add the following snippet:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>src/main/resources/webapp/WEB-INF/</param-value>
  </context-param>
```

4. The `applicationContext.xml` file should reside in the directory `src/main/resources/webapp/WEB-INF/` and contain the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:amq="http://activemq.apache.org/schema/core"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
       http://activemq.apache.org/schema/core
         http://activemq.apache.org/schema/core/activemq-core.xsd
       http://camel.apache.org/schema/spring
         http://camel.apache.org/schema/spring/camel-spring.xsd
    ">

    <broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost">

      <persistenceAdapter>
        <kahaDB directory="activemq/kahadb"/>
      </persistenceAdapter>

        <!-- The transport connectors ActiveMQ will listen to -->
        <transportConnectors>
            <transportConnector name="openwire" uri="tcp://localhost:61616" />
        </transportConnectors>
    </broker>

    <!--
        configure the camel activemq component to use the current broker
    -->
    <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent"
        p:brokerURL="vm://localhost?create=false&amp;waitForStart=10000" />

    <camelContext xmlns="http://camel.apache.org/schema/spring">


    </camelContext>

</beans>
```

5. Create an Apache Camel Routes that consumes files from the `src/data` directory.
    - Use the `noop=true` option to avoid deleting the files.
    - Publish files to the `activemq:queue:demo` queue.

6. Create an Apache Camel Routes that consumes JMS Messages from the `activemq:queue:demo` queue.
   - Use the content based router pattern, to assert the Xpath expression `/person/city = 'London'`.
   - If the condition is true, the Route publishes to the `target/messages/uk` directory.
   - If the condition is not true, the Route publishes to the `target/messages/others` directory.
7. Add a log EIP processor right before the file producer in order to log the content of the Exchange body using the simple language.
8. Launch the project locally using either:
   - `mvn jetty:run`
   - `mvn jboss-as:run`
9. Review log messages on the console.

```
2014-10-22 15:51:24,341 [main           ] INFO  SpringCamelContext            - Total 2 routes, of which 2 is started.
2014-10-22 15:51:24,343 [main           ] INFO  SpringCamelContext            - Apache Camel 2.12.0.redhat-610379 (CamelContext: camel-1) started in 0.290 seconds
2014-10-22 15:51:24,346 [main           ] INFO  ContextLoader                 - Root WebApplicationContext: initialization completed in 2212 ms
2014-10-22 15:51:24.377:WARN:oejsh.RequestLogHandler:!RequestLog
2014-10-22 15:51:24.389:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 10 seconds.
2014-10-22 15:51:25,375 [sConsumer[demo]] INFO  route2                        - >> Message : <?xml version="1.0" encoding="UTF-8"?>
<person user="james">
  <firstName>James</firstName>
  <lastName>Strachan</lastName>
  <city>London</city>
</person>
2014-10-22 15:51:25,381 [sConsumer[demo]] INFO  route2                        - >> Message : <?xml version="1.0" encoding="UTF-8"?>
<person user="hiram">
  <firstName>Hiram</firstName>
  <lastName>Chirino</lastName>
  <city>Tampa</city>
</person>
2014-10-22 15:51:25,383 [sConsumer[demo]] INFO  route2                        - >> Message : <?xml version="1.0" encoding="UTF-8"?>
<person user="jon">
  <firstName>Jonathan</firstName>
  <lastName>Anstey</lastName>
  <city>St. John's</city>
</person>
```

Figure 1. Results - Web Console

1.4. Camel Unit test with Mock endpoint

The goal of this exercise is to first create a unit test and then debug the project with the breakpoint set at a processor.

The dependency for the Camel test support classes is the Maven artifact `camel-test`.

Follow these steps to complete this exercise:

1. In the `com/redhat/gpe/training/camel` package, create a class and call it `SimpleCamelUnitTest`.
2. Extend the `CamelTestSupport` class.
3. Override the method `createRouteBuilder()` so that it returns a `RouteBuilder` object.
4. The `RouteBuilder` class should contain this Route:

```
from("direct:start")
  .filter(header("foo").isEqualTo("bar"))
  .to("mock:result");
```

You added a `mock` endpoint to collect the generated exchanges. A filter will be used to discard an exchange based on the expression calculated.

5. Declare a Mock endpoint using the `MockEndpoint` class and name it `resultEndpoint`.
6. Add the `@EndpointInject` annotation with `mock:result` as the URI definition.
7. Set a `template` field with the `ProducerTemplate` definition.
8. Annotate it with the `@Produce` annotation and the `direct:start` URI.
9. Add a test method called `testSendMatchingMessage` that checks messages sent from the Mock endpoint for a `<matched/>` tag.
10. Initialize the `expectedBody` String with the contents of the message body, and the `<matched/>` tag as a suffix.
11. Call the `expectedBodiesReceived(expectedBody)` and `expectedMessageCount(1)` methods at the Mock endpoint.
12. Send a message using the `template.sendBodyAndHeader()` method of the `ProducerTemplate` interface, where the message body contains the `expectedBody` String and the `foo` header contains `bar`.
13. Call the `assertIsSatisfied()` method of the Mock endpoint.
14. Add a second unit test method `testSendNoMatchingMessage()`.
15. The `expectedBody` String is now `<notMatched/>`.
16. Expect to receive no messages from the `expectedMessageCount(0)` endpoint.
17. Send the message with the `foo` Header containing `pub`.
18. Call the `assertIsSatisfied()` method of the Mock endpoint.
19. Run the unit test by right-clicking the project and selecting **Run As → JUnit test**.
20. Check that the test completed successfully.
21. Stop the unit test.

Do not forget to annotate your unit test method with the `@Test` JUnit annotation.

22. Debug your unit test:
    a. Add the following code that enables the `debugger interceptor` in the unit test class:

```
@Override
public boolean isUseDebugger() {
    // must enable debugger
    return true;
}

@Override
protected void debugBefore(Exchange exchange, Processor processor,
                           ProcessorDefinition<?> definition, String id, String shortName) {
    // this method is invoked before we are about to enter the given processor
    // from your Java editor you can just add a breakpoint in the code line below
    log.info("Before " + definition + " with body " + exchange.getIn().getBody());
}
```

b. Add a breakpoint to the line containing `log.info`.

c. Debug your unit test and inspect the contents of the Exchange right before it is sent to the next processor.

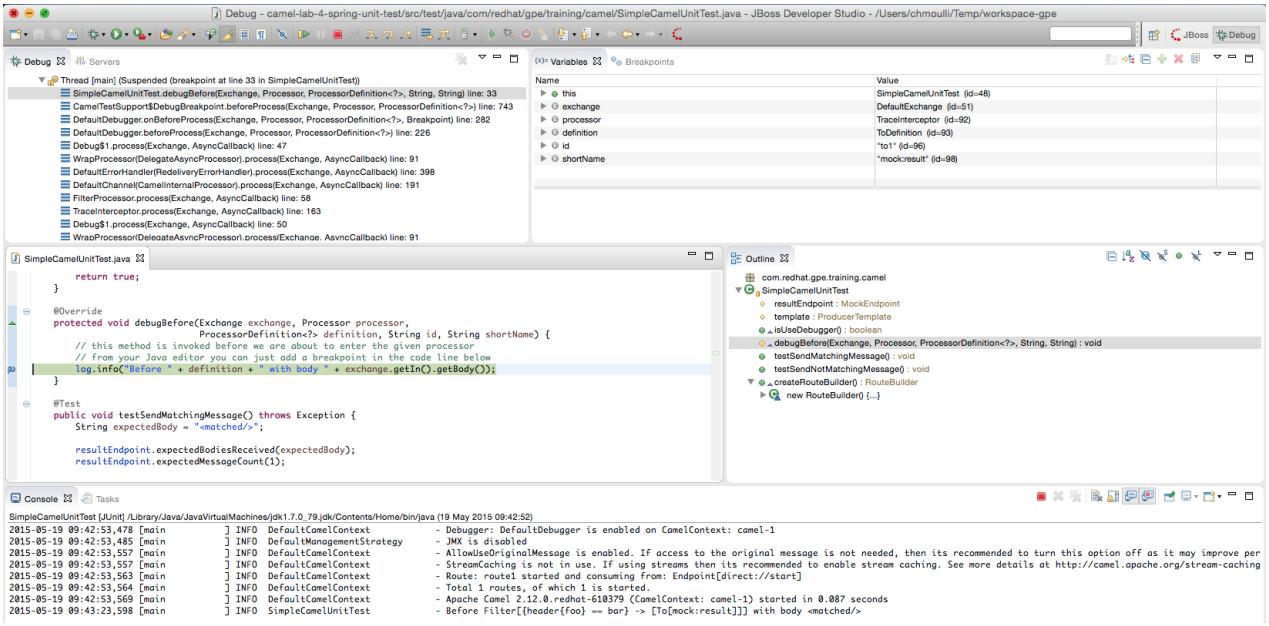

Figure 2. Debug Example

d. Stop the test.

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