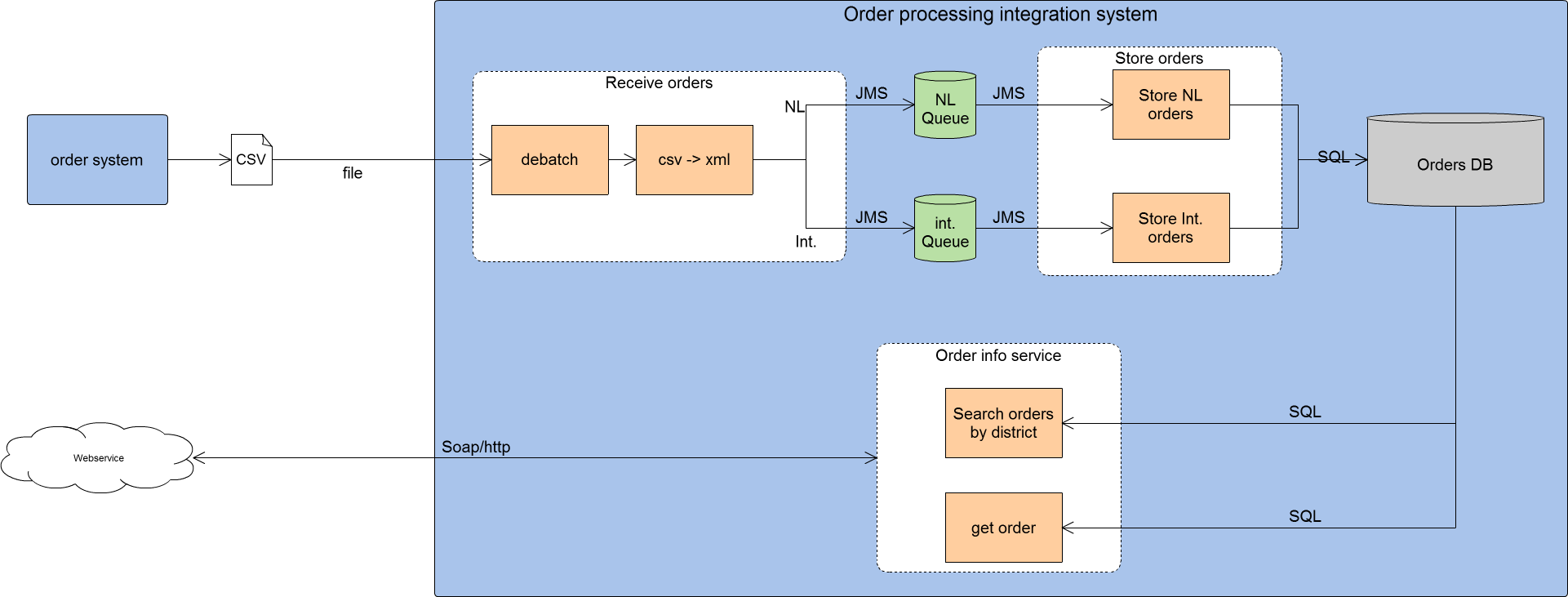
JBoss Fuse Hands on coding session

In the JBoss Fuse hands on session (aka Rubix codefest JBoss) we are going to (try to) build an end to end integration solution based of three distinct parts or use cases which build on each other. The first use case will be documented step by step in this document. For the other two use cases general tips and tricks are available in the PowerPoint.

# Integration solution

In this hands on session we are going to create an end-to-end integration solution for processing and querying orders. The orders are received from an external system in bulk csv format. The integration solution is responsible for receiving the csv file, transforming it into an xml cdm format, routing the order based on country code and storing it into the order database. For management purposes a webservice is available for querying orders. The webservice has two operations one for retrieving order IDs based on district (NL or international) and one operation for retrieving the order based on ID. This fictional integration solution does cover a variety of different message types, protocols and messaging patterns. For example csv and xml messages are covered, JMS, SQL and webservices are covered and both synchronous and asynchronous messaging is covered.

Graphically the overall integration solution is depicted as below:



# Use case 1 Receive orders

The first use case we are going to develop is an interface which will perform the following tasks:

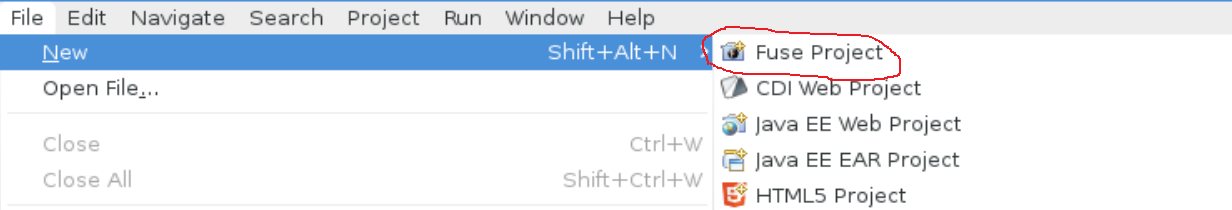
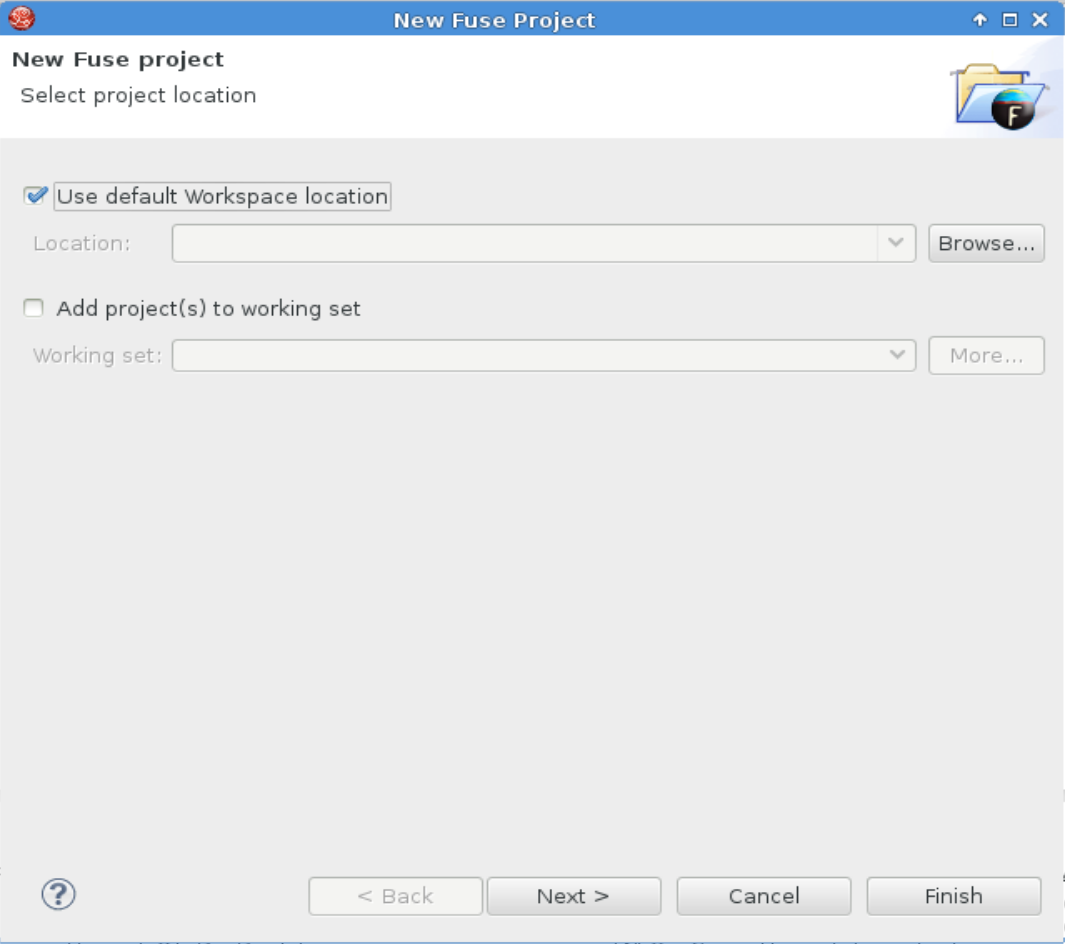
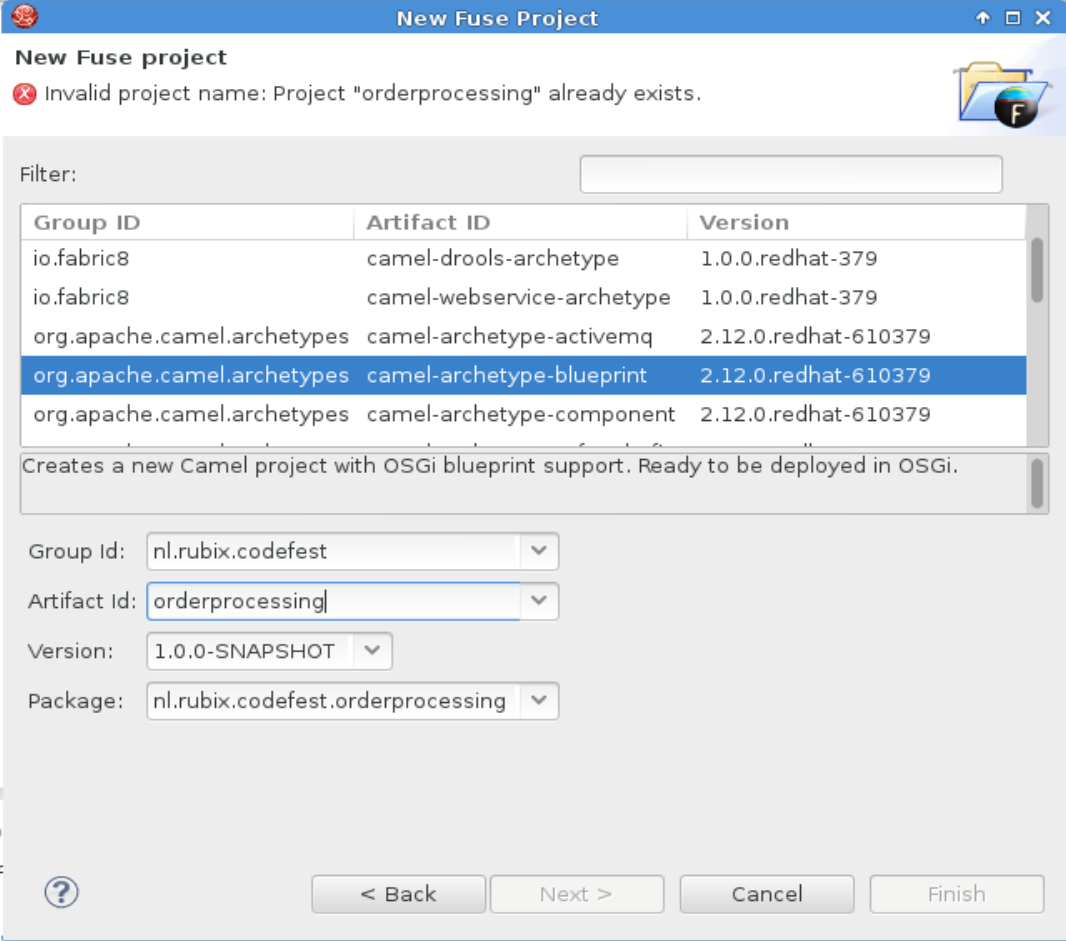
* Receive a csv file containing bulk orders
* Marshal the csv file into a Java object
* Debatch the bulk orders
* Transform the message into a CDM xml message
* Route the message based on countryCode
* Publish NL orders into an nl orders JMS queue
* Publish international orders into an international orders JMS queue

But before we are able to develop the interface we need to setup a JBoss Fuse project.

## Creating and setup of a new JBoss Fuse project

In JBDS (JBoss Developer Studio) when the integration stack is installed correctly the option to create a new Fuse project becomes available.

### Create a new Fuse project

1. Create a new Fuse project
2. Use the default workspace location and click next
3. Creating a Fuse project in JDBS uses maven as a build system so we need to select an archetype and configure the groupid and artifactid   
   In the archetype list select camel-archetype-blueprint  
   since some of the resources like Java classes are provided it is important to enter the groupid and artifactid in the same manner as depicted above, so:  
   groupid: nl.rubix.codefest  
   artifactid: orderprocessing
4. Click finish to create the Fuse project

### Clean up the Fuse project

The Camel blueprint archetype comes with an example Camel Context and route. However, these sample code provided by the archetype is not required so we are going to do some clean-up.

Delete the following artifacts from the project:

* /orderprocessing/src/main/resources/OSGI-INF/blueprint/blueprint.xml
* /orderprocessing/src/main/java/nl/rubix/codefest/orderprocessing/Hello.java
* /orderprocessing/src/main/java/nl/rubix/codefest/orderprocessing/HelloBean.java
* /orderprocessing/src/test/java/nl/rubix/codefest/orderprocessing/RouteTest.java

Now we have a completely empty Fuse project and are ready to build our use cases!

### Setup provided resources in the Fuse project

In order to focus completely on the development of interfaces some resources are provided beforehand. These resources will speed up some of the development tasks more unrelated with JBoss Fuse.

It is essential these resources are present in the project and are located in the specified location[[1]](#footnote-1).

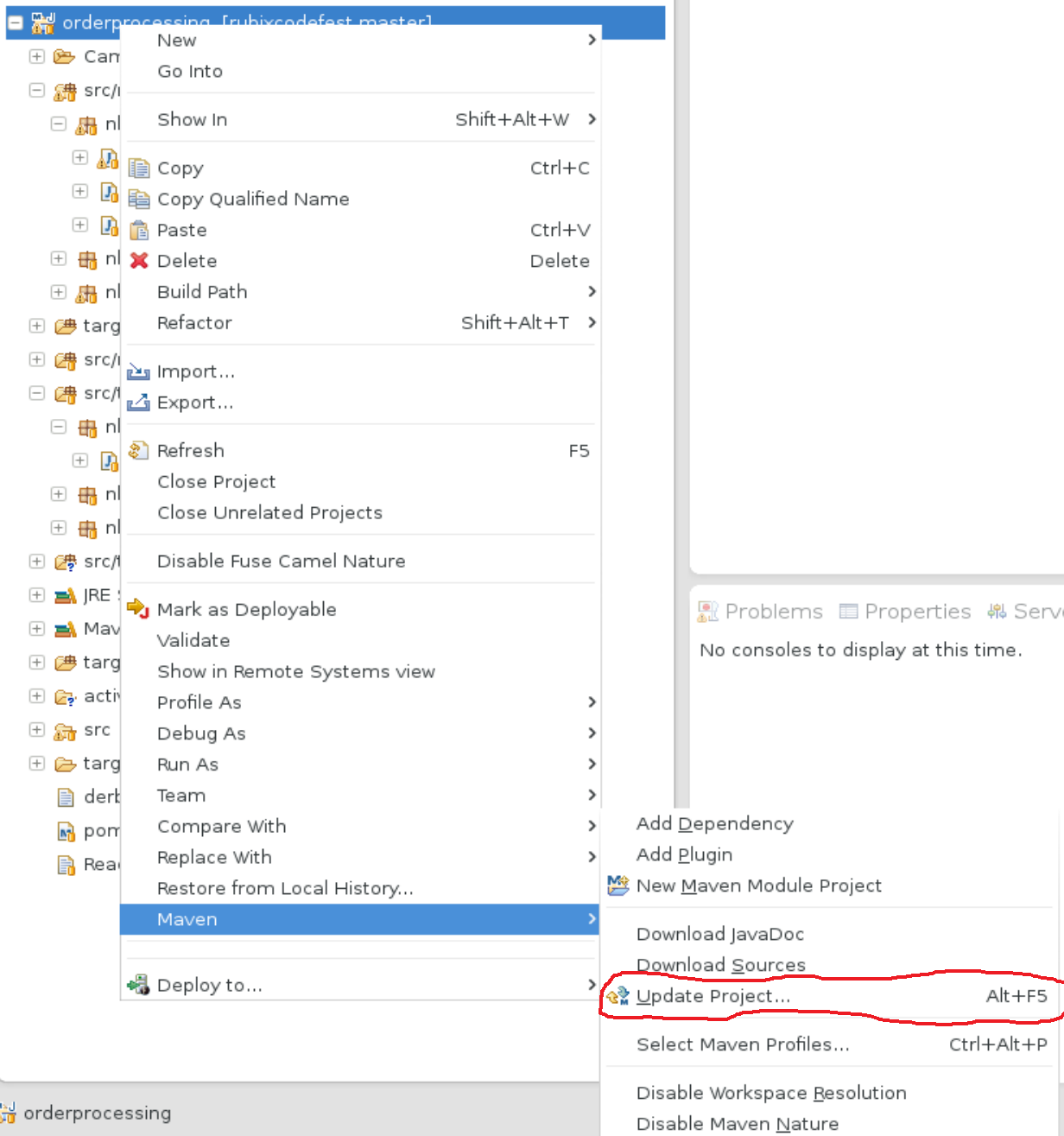
The following resources are supplied

* DatabaseBean.java – setup class for embedded Derby DB
* OrderInfoService.wsdl – wsdl for the webservice
* Order.xsd – xsd for the order message
* Orders.csv – input csv file
* Pom.xml – the maven pom file with all the dependencies and plugins
* Shared-resources.xml – containing the embedded broker and database used in the other Camel contexts
* Sql.properties – properties file containing the SQL queries for interacting with the database

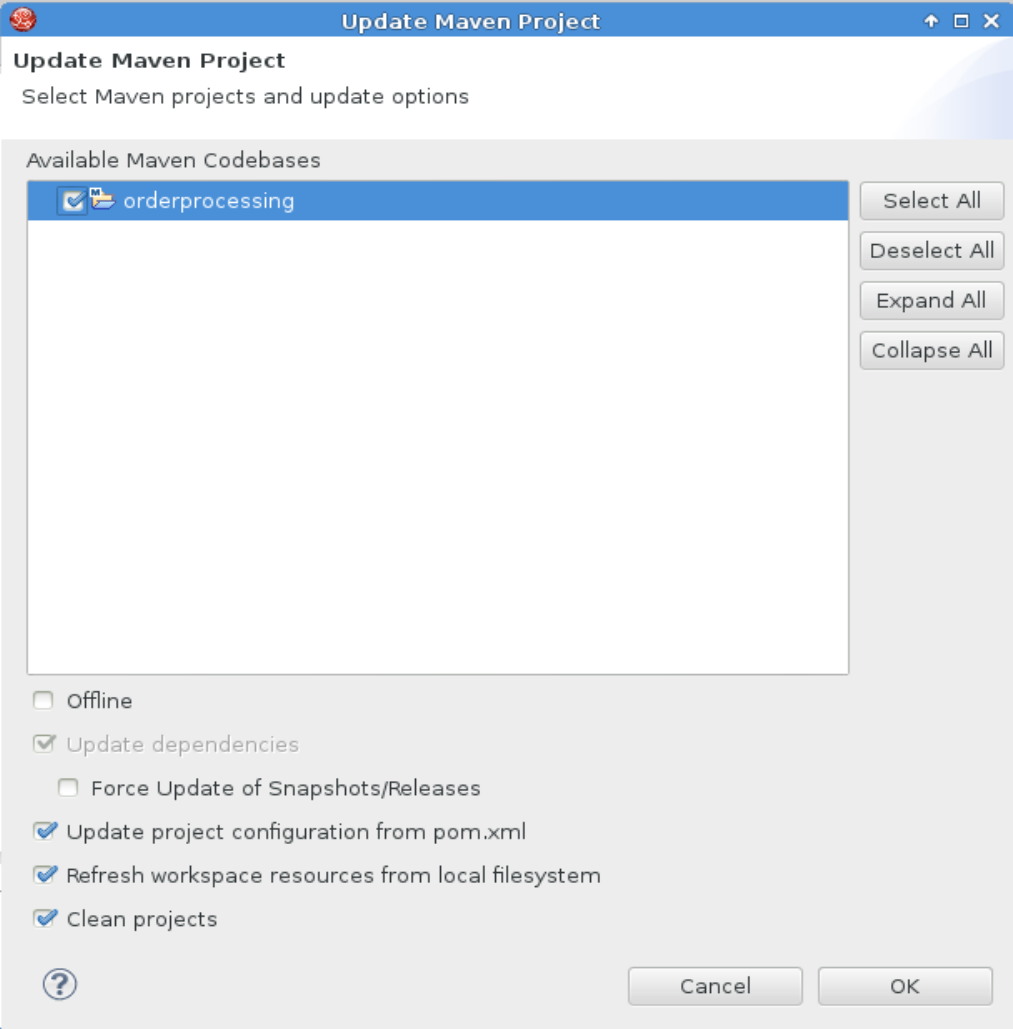
The resources can be found on Github in the folder resources. Add the resources to the project as described in the table below:

|  |  |  |
| --- | --- | --- |
| Resource | location | |
| DatabaseBean.java | /orderprocessing/src/main/java/nl/rubix/codefest/orderprocessing/sql/ (java package = nl.rubix.codefest.orderprocessing.sql) |
| OrderInfoService.wsdl | /orderprocessing/src/main/resources/wsdl/ |
| Order.xsd | /orderprocessing/src/main/resources/xsd/ |
| Orders.csv | Not used inside the project but in the file location used for testing |
| Pom.xml | (project root folder overwrite the generated pom.xml with this one) |
| Shared-resources.xml | /orderprocessing/src/main/resources/OSGI-INF/blueprint/ |
| Sql.properties | /orderprocessing/src/main/resources/ |

When all the resources are added to the project you will most likely get an error stating that the project needs to be updated. To update the project: right click the project -> maven -> update project



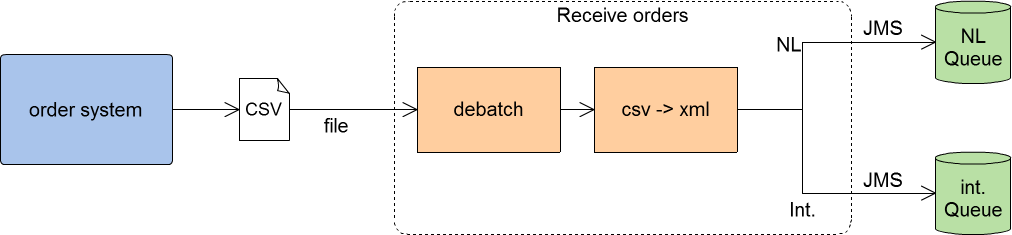
Accept all the defaults and click OK



The project is now ready for developing the first use case!

## Starting development

The use case we are going to develop contains one Camel context with two routes.



The first route will be receiving the csv file, marshal it to a Java bean, debatch it using the splitter EIP, transform it to a CDM message and call the second route.

The second route will implement a content based router which will route orders based on country code and publish the order on a JMS queue.

### Receiving and processing the CSV file

The orders are received from an external system in the form of a CSV file which contains multiple orders. The format of the CSV file is:

Orderid, amount, price, country

And every order is located on a separate line.

Receiving files in Fuse using Apache Camel is very straightforward, however we need to be able to parse and process the CSV file in order to extract the data from it. To do this we need to create a custom dataformat based on Camel Bindy: <http://camel.apache.org/bindy.html>

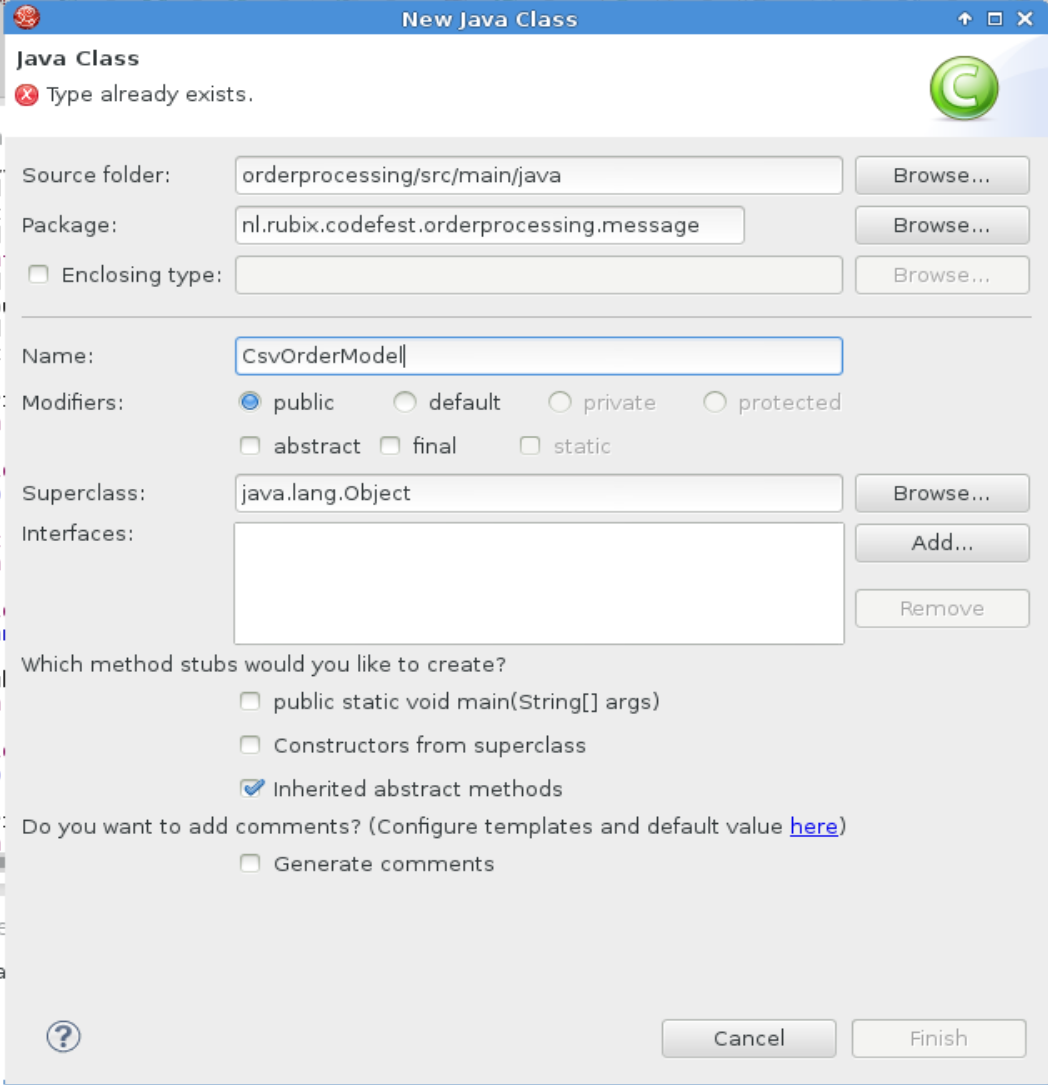
This dataformat can be used in Camel in the (un)marshal processors so the CSV files is marshalled into a Java bean.

We have to implement the following steps in order to receive and process the CSV file:

1. Create the Camel Bindy dataformat POJO
2. Create a new Camel context and setup the dataformat created in step 1
3. Create a Camel route that uses a file poller and marshal the CSV file using the dataformat configured in step 2

#### Create the Camel Bindy dataformat POJO

A custom Camel Bindy dataformat is just a POJO so create a new Java Class by right clicking the src/main/java folder and click new class.

* Create a new package:”nl.rubix.codefest.orderprocessing.message”
* Name the Class CsvOrderModel  
  

In the (empty) java class we are first going to create a regular bean with the following instance variables:

* orderId
* amount
* price
* country

package nl**.**rubix**.**codefest**.**orderprocessing**.**message**;**

public class CsvOrderModel **{**

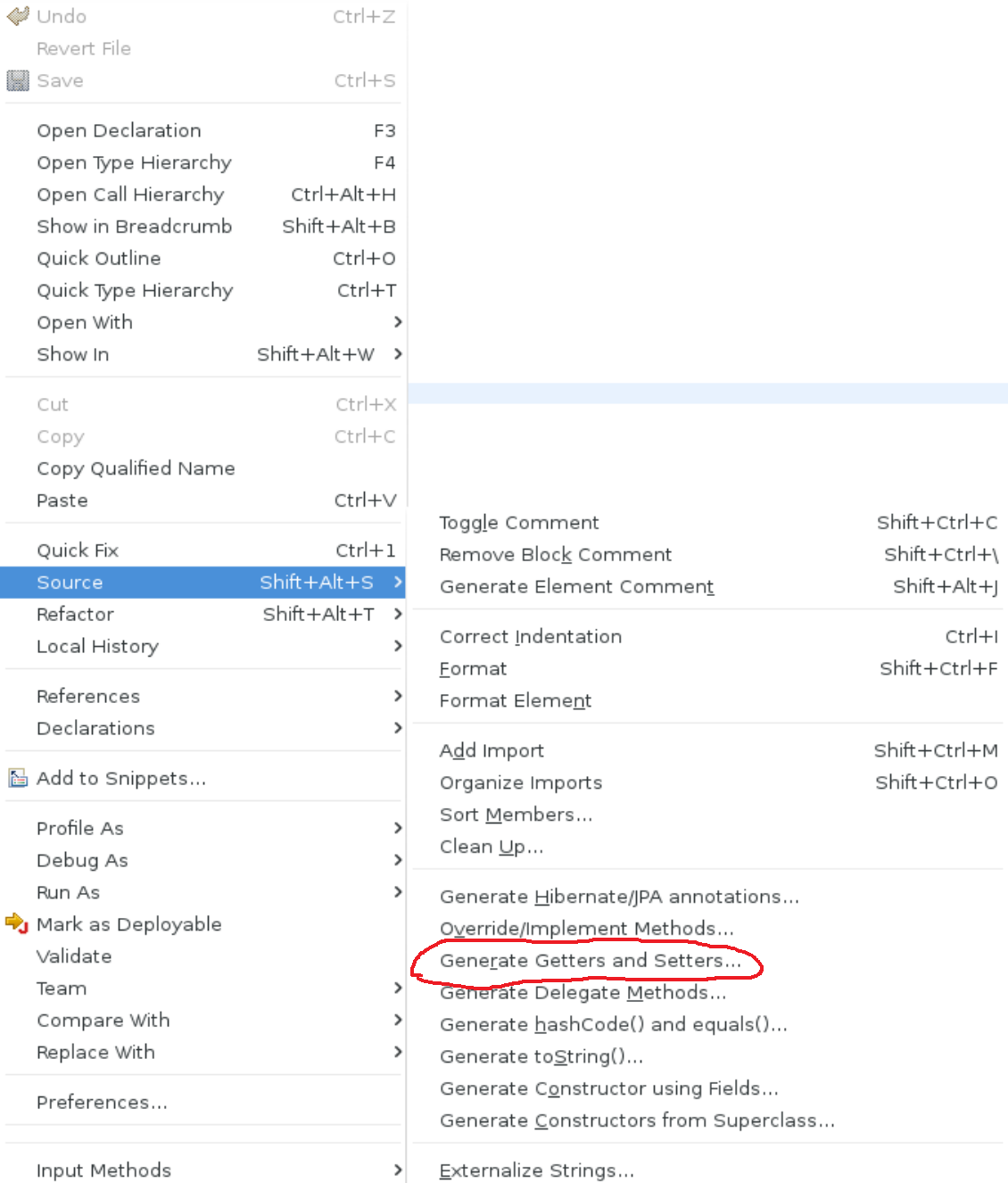
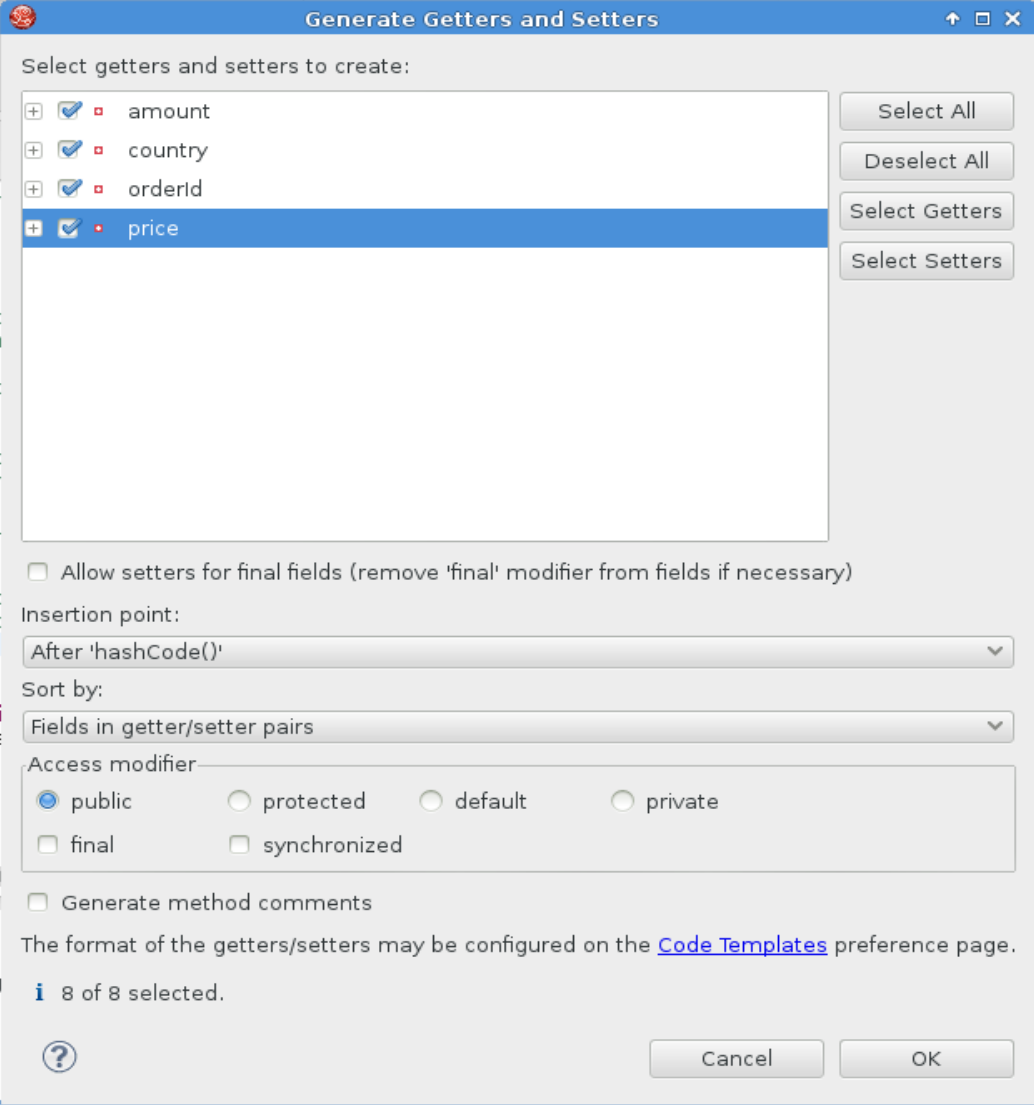
private String orderId**;**

private int amount**;**

private Double price**;**

private String country**;**

use JBDS to generate getters and setters.

* Right click within the class -> source -> generate getters and setters 
* Select all the fields and click ok 

Now we have a regular Java Bean, the next step is to add the bindy components to it using Java annotations.

There are two different things we need to add to our Java bean.

1. Csv record information
2. Datafield declerations

* Above the class decleration add the following Java annotation:   
  @CsvRecord**(**separator**=**","**,** crlf **=**"UNIX"**)**

public class CsvOrderModel **{**

* Above each instance variable set the corresponding position in the CSV file with the following annotations:

@DataField**(**pos **=** 1**)**

private String orderId**;**

@DataField**(**pos **=** 2**)**

private int amount**;**

@DataField**(**pos **=** 3**)**

private Double price**;**

@DataField**(**pos **=** 4**)**

private String country**;**

Press Ctrl+Shift+O to resolve the missing imports:   
import org.apache.camel.dataformat.bindy.annotation.CsvRecord;

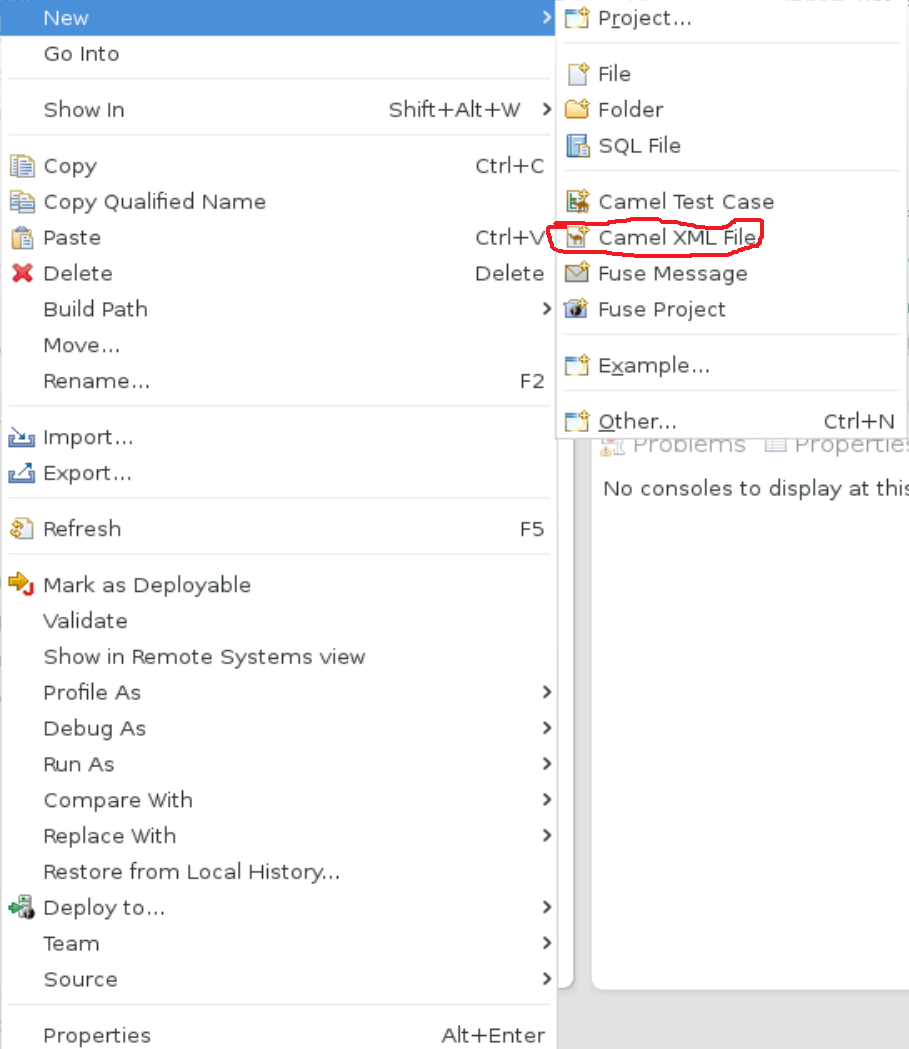
import org.apache.camel.dataformat.bindy.annotation.DataField;

Now our Camel bindy dataformat is complete and we can use it inside a Camel Context and route.

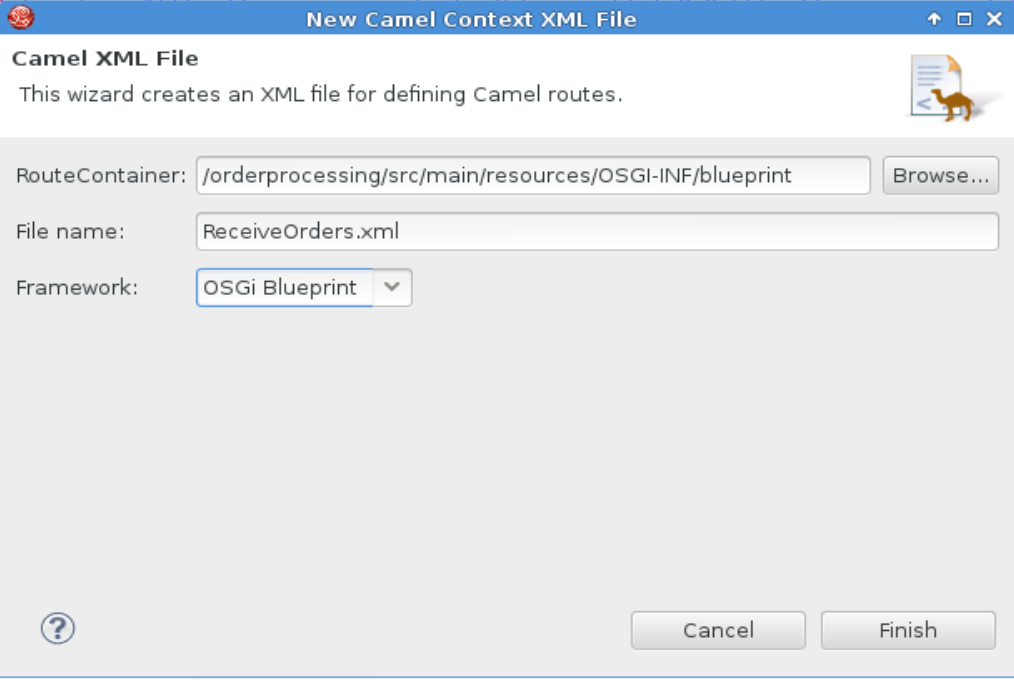
#### Create a new Camel context and setup the dataformat created in step 1

Interfaces and services in JBoss Fuse are implemented in Camel Contexts and routes. For this use case we are going to create a new Camel Context which contains two routes. Since we use the OSGi Blueprint framework all Camel Contexts are stored in the following location inside the project: /orderprocessing/src/main/resources/OSGI-INF/blueprint[[2]](#footnote-2)

Right click the blueprint folder and select new -> Camel Xml File



**Name the Camel Context ReceiveOrders.xml and make sure to select OSGi Blueprint in the framework drop down!**



Click finish and the Camel Context is created and the Fuse IDE opens the route editor.

Switch to the source view by clicking on the source tab at the bottom of the editor.

Inside the camelContext tags create a new dataformat by adding the following code:

<?xml version=**"1.0"** encoding=**"UTF-8"**?>

<blueprint xmlns=**"http://www.osgi.org/xmlns/blueprint/v1.0.0"**

xmlns:camel=**"http://camel.apache.org/schema/blueprint"**

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

xsi:schemaLocation=**"http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd**

**http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-blueprint.xsd"**>

<camelContext trace=**"false"** xmlns=**"http://camel.apache.org/schema/blueprint"**>

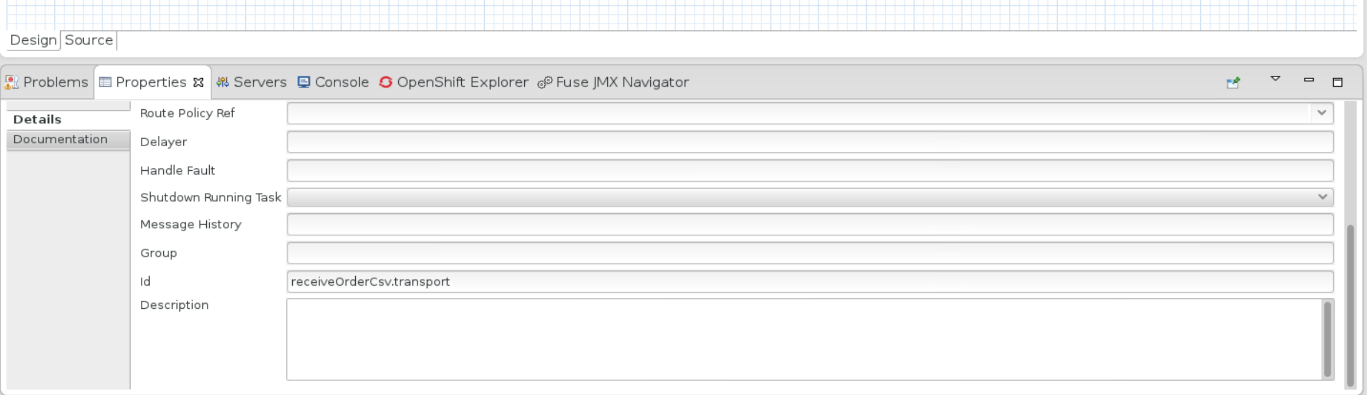
<dataFormats>

<bindy type=**"Csv"** packages=**"nl.rubix.codefest.orderprocessing.message"** id=**"csvordermodel"**/>

</dataFormats>

Now we can start by building our route.

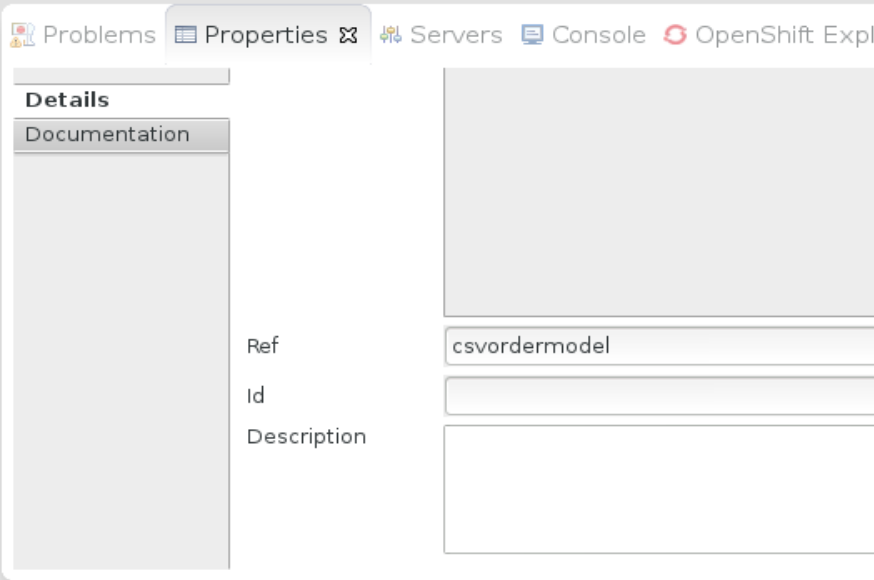
Switch back to the design view and select properties. Scroll down and fill in a custom id for the route, this is not required but it is a good habit to get into.



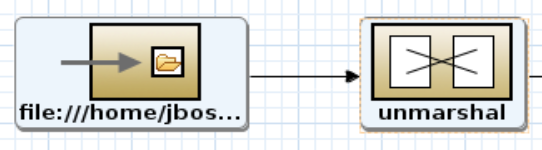
Select the Palette and drag and endpoint to the canvas in the URI field fill in the path of the folder where Camel will receive the CSV file.

For example: <file:///home/jboss/Documents/orderprocessing/in/>

Next drag an unmarshal processor on the canvas (the unmarshal processor is located under the Transformation section of the Palette). And select properties, scroll down and fill in the csvordermodel into the ref field:



Note: the csvordermodel is the id of our bindy dataformat we created earlier. Use the arrow to connect the endpoint and the unmarshal processors and save the route. The route should now look like this:



Now we have created a file poller interface which receives a csv file and processes it into a Java bean we can use in the rest of our interface.

The next step is to debatch the message and transform it to a CDM xml message.

### Debatching and transforming the message

Debatching and transforming the message is a two-step process, first we are going to debatch the message. The individual items we are going to transform into a CDM message.

#### Debatching the message

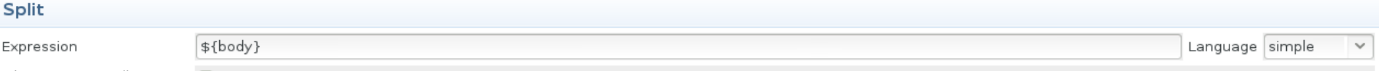
The output of the unmarshal processor is a List of Map, from the documentation:

The collection returned is a List of Map objects. Each Map within the list contains the model objects that were marshalled out of each line of the CSV. The reason behind this is that each line can correspond to more than one object. This can be confusing when you simply expect one object to be returned per line.

Each object can be retrieve using its class name.

Since a List is an iterable object debatching it is very straightforward. From the Palette drag the Split processor onto the canvas. The split processor implements the Splitter EIP. Using the simple expression language we can configure the splitter to split the body. Since the body consists of a Java iterable type the splitter splits the message and creates a new message for every item in the iterable object (in our case a list).

In the properties of the split processor enter ${body} in the expression and verify simple is selected as the language.



Next connect the unmarshal processor with the split processor and save the route. The route now looks like this:



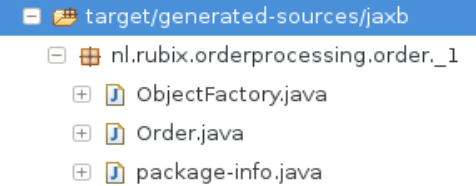
#### Transforming the message

The next step is to transform the message into a CDM xml format. There are numerous ways in Camel to perform message transformation, for example: xquery, XSLT, bean mappers. However one of the most common ways is to execute transformations in a custom Java Bean or custom processor. In this use case we are going to implement a custom processor for performing the message transformation. In use case 3 we will explore the custom bean method. Note, custom processors are not only usefull for performing message tranformations. All the Enterprise Integration Patterns, for example the split pattern used previously, are processors. So, custom processors can also be used to extend the functionality of Camel with custom functionality.

Our source and target message. In a message transformation there is always a source and a target message. Our source message is the split order in the format of a Java Map (remember the Camel bindy outputs a List of Map after the split we have individual Map objects). The target message format is our CDM xml message. However in order to perform a transformation between a Java object and an xml message we have to take an additional step. In this step we use JAXB to create a Java object representation of our xml message. In this case we can transform one java object to another and use a marshal to create the xml document based on the Java object created by JAXB.

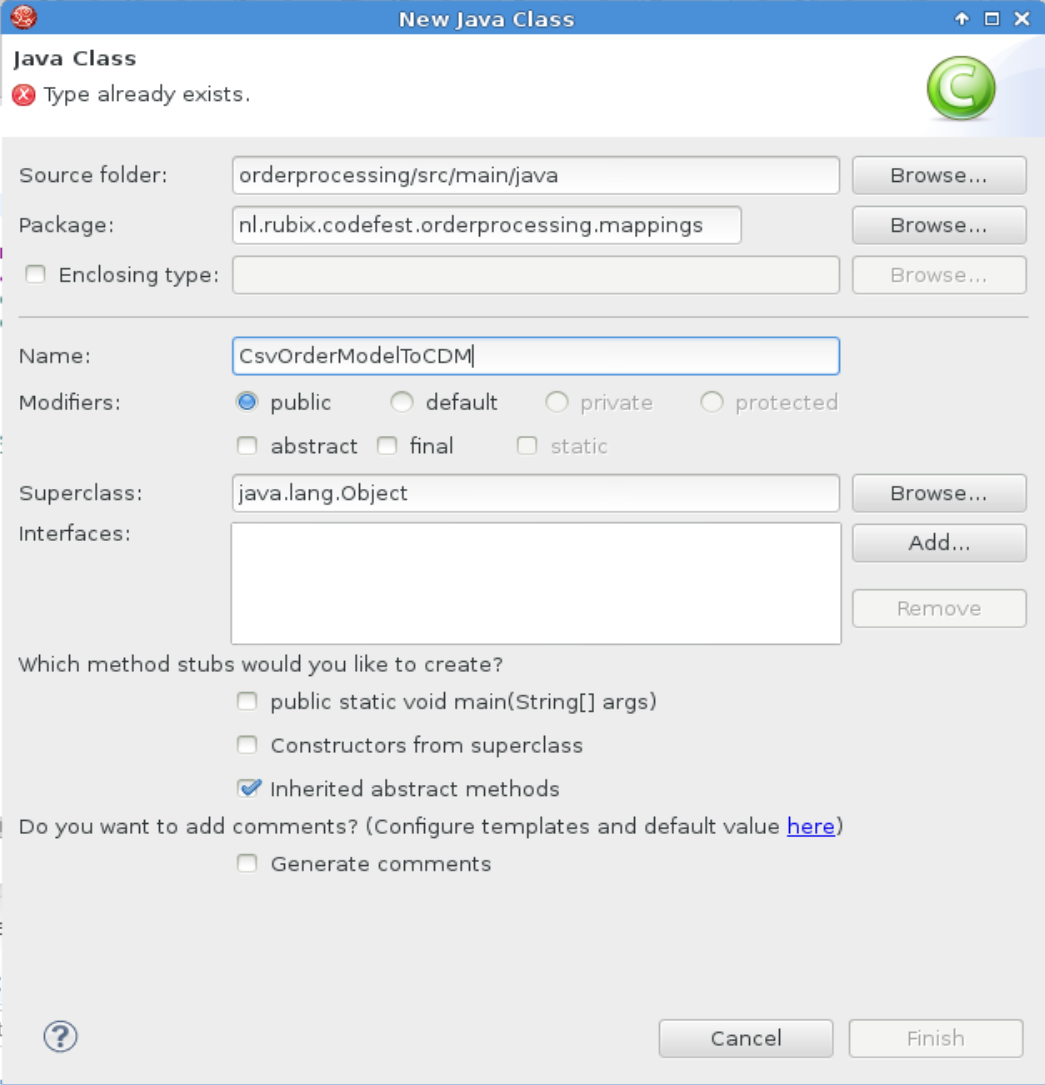
The CDM xml message is represented in the order.xsd located in the resources/xsd folder. The maven plugin in the pom.xml file already generated the JAXB classes and placed them in the /orderprocessing/target/generated-sources/jaxb folder.

The content of this folder should look like this:



Order.java contains the Java class representation of the xsd. Our custom processor will transform the CsvOrderModel.java into an Order.java object.

To create the custom processor create a new java class in the src/main/java. To organize our project create a sub package mappings and name the class CsvOrderModelToCDM.



And click finish.

In order to create a custom processor we have to implement the Processor interface provided by Camel and add the process method.

public class CsvOrderModelToCDM **implements** Processor**{**

@Override

public void process**(**Exchange exchange**)** **throws** Exception **{**

use Ctrl+Shift+O to resolve the missing imports.

The parameter of the process method is the entire Camel Exchange. So the first action is to extract the message from the exchange.

// retrieve the in message from the Camel exchange

Message in **=** exchange**.**getIn**();**

// retrieve the body of the in message, this body contains the split output from the bindy unmarshal, which is a Map<String, CsvOrderModel>

Map**<**String**,** CsvOrderModel**>** body **=** **(**Map**<**String**,** CsvOrderModel**>)** in**.**getBody**();**

// retrieve the CsvOrderModel object from the map

CsvOrderModel csvOrder **=** body**.**get**(**CsvOrderModel**.**class**.**getName**());**

Now we have extracted the CsvOrderModel object we can perform the mapping. Use the getters and setters of the objects to bind the different fields together.

//create the data transformation of from the CsvOrderModel to the generated Order class

// use the generated object factory to create a new order object

ObjectFactory objFac **=** **new** ObjectFactory**();**

Order order **=** objFac**.**createOrder**();**

// create the data mepping by assinging the setters of the order object to the getters of the csvOrder object

order**.**setOrderId**(**csvOrder**.**getOrderId**());**

// since totalPrice is a BigDecimal and price is a Double we need to convert the type

BigDecimal bdPrice **=** **new** BigDecimal**(**csvOrder**.**getPrice**(),** MathContext**.**DECIMAL64**);**

order**.**setTotalPrice**(**bdPrice**);**

// since totalAmount is a BigInteger and amount is an int we need to convert the type

BigInteger biAmount **=** BigInteger**.**valueOf**(**csvOrder**.**getAmount**());**

order**.**setTotalAmount**(**biAmount**);**

order**.**setCountryCode**(**csvOrder**.**getCountry**());**

// it is a best practice to use the in message in the exchange, when instictively you want to use the out message. THe reason for using the in message

// is because all headers and properties of the message are deleted if you use the out message. Using the in message preserves all the properties and headers

in**.**setBody**(**order**);**

use Ctrl+Shift+O to resolve the missing imports.

Now we created our custom processor class we can use it inside the Camel route.

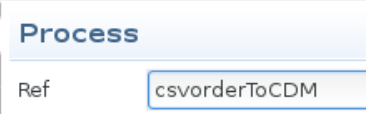
To use the custom processor we have to first instantiate it inside the Blueprint context. This is done by declaring a bean outside the camelContext tags but inside the blueprint tags.

</camelContext>

<bean id=**"csvorderToCDM"** class=**"nl.rubix.codefest.orderprocessing.mappings.CsvOrderModelToCDM"**/>

</blueprint>

Back in the design view drag a Process processor onto the canvas and in the properties select csvorderToCDM from the drop down list.



Check in the source view the process tag resides within the split tags.

The entire route should look like this in the source view:

<route customId=**"true"** id=**"receiveOrderCsv.transport"**>

<from uri=**"file:///home/jboss/Documents/orderprocessing/in/"**/>

<unmarshal ref=**"csvordermodel"**/>

<split>

<simple>**${body}**</simple>

<process ref=**"csvorderToCDM"**/>

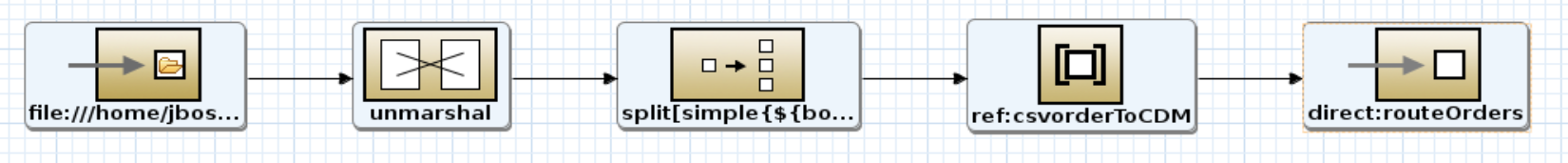
</split>

</route>

The next thing we will add is an endpoint to call the route we will create up next.

Back in the design view drag another endpoint to the canvas and enter: direct:routeOrders in the URI. The direct protocol is used in Camel to call routes who reside inside the same Camel Context.

The complete route looks like this in the design view:



In the source view the route looks like this:

<route customId=**"true"** id=**"receiveOrderCsv.transport"**>

<from uri=**"file:///home/jboss/Documents/orderprocessing/in/"**/>

<unmarshal ref=**"csvordermodel"**/>

<split>

<simple>**${body}**</simple>

<process ref=**"csvorderToCDM"**/>

<to uri=**"direct:routeOrders"**/>

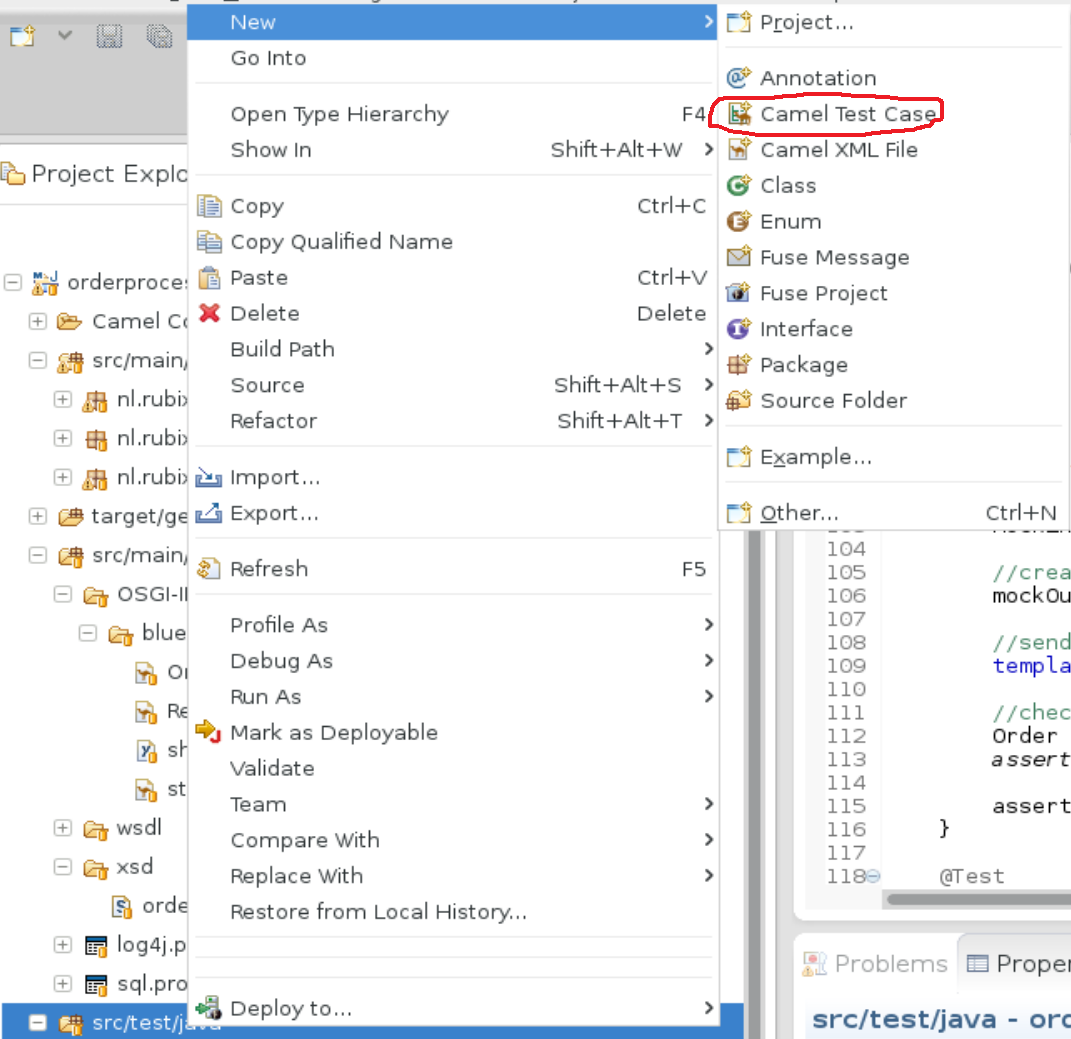
</split>

</route>

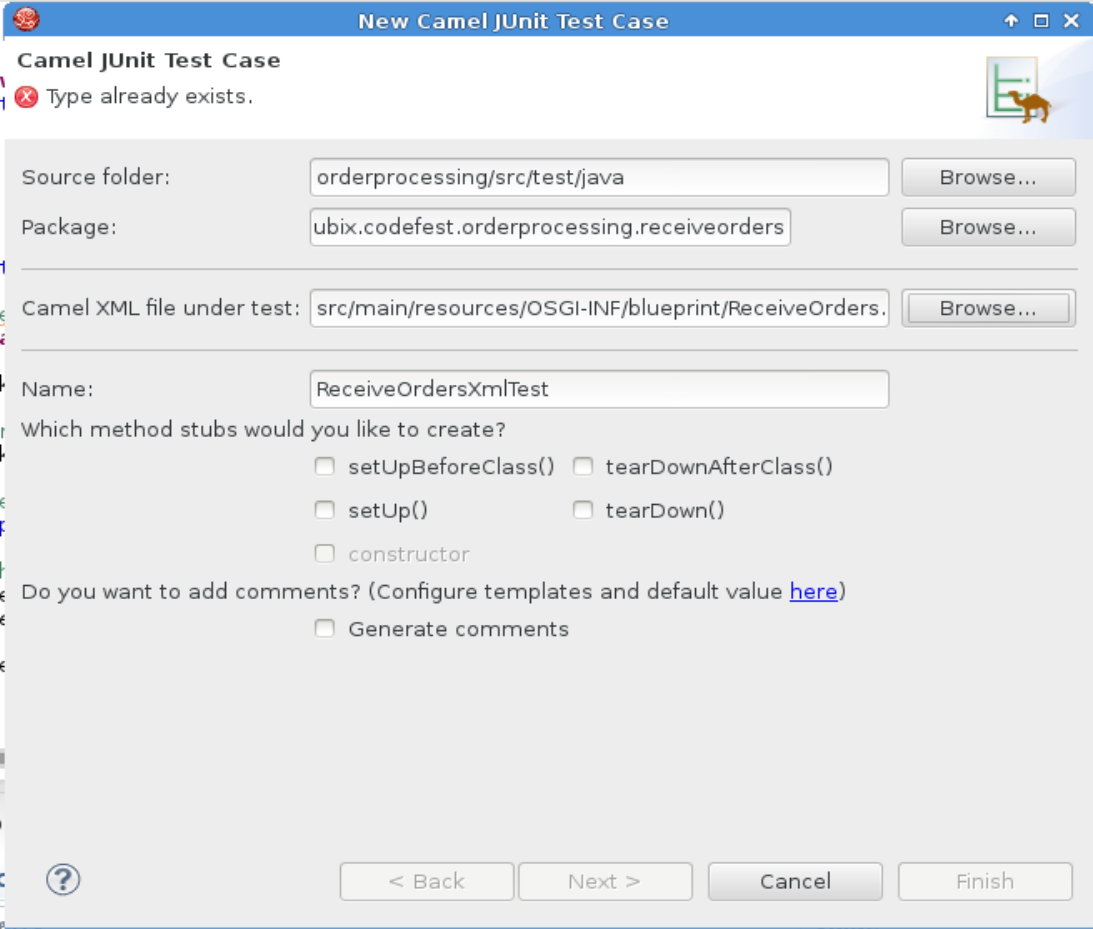
Now the first Camel route is complete. However we still need to write our unittest.

## Writing a unittest

Camel has an extensive testing framework based on Junit. To create a new Camel testcase right click src/test/java -> new -> Camel Test Case

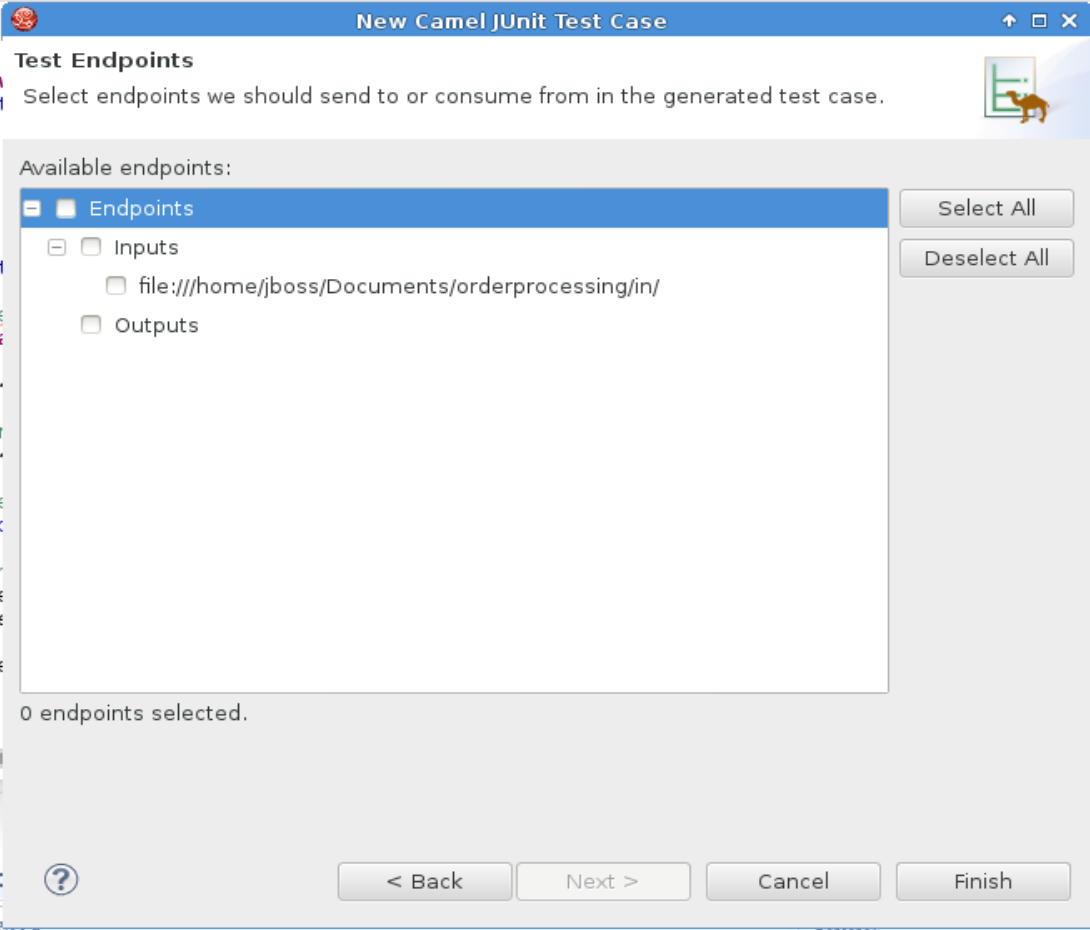


Under package create a receiveorders package, next click the third browse butten (next to Camel XML file under test) and select the Camel context created earlier.



Click next.

Deselect all the options.



Click finish.

This will create a new java class extending CamelBlueprintTestSupport. When the unittest is run (for example during a maven build. The code inside the @Test methods are executed as unittests. We will implement a @Test method with our unittest.

The unittest we will implement will test the following scenario:

We are going to replace the file poller endpoint with a direct endpoint, and the to direct endpoint with a mock endpoint. This enables us to call and listen to these endpoints inside our unittest. In essence we are mocking our endpoints. We will then send a test request containing three orders to the endpoint, this must result into three messages sent to the mock endpoint. We will inspect one message sent to the mock endpoint to verify the message transformation was executed as expected.

The Blueprint context is already configured in the unittest by the wizard. However, some of our resources are in the shared-resources.xml context. So we need to add this context to the unittest case.

Extend the getBlueprintDescriptor method so it looks like this:

@Override

protected String getBlueprintDescriptor**()** **{**

**return** "/OSGI-INF/blueprint/ReceiveOrders.xml,/OSGI-INF/blueprint/shared-resources.xml"**;**

**}**

**Note: the / at the beginning of the paths!**

### Mocking endpoints

As described above we are going to mock the endpoints in the Camel route in order to use them in the unittest. To accomplish this we are going to use the Camel interceptor. <http://camel.apache.org/intercept.html>

To use an interceptor we first need to add the following method to our test class:

@Override

public boolean isUseAdviceWith**()** **{**

**return** **true;**

**}**

The next step is to implement the interceptor within the test method. Add the following code to the test method:

@Test

public void testReceiveOrderCsvTransportRoute**()** **throws** Exception**{**

context**.**getRouteDefinition**(**"receiveOrderCsv.transport"**).**adviceWith**(**context**,** **new** AdviceWithRouteBuilder**()** **{**

@Override

public void configure**()** **throws** Exception **{**

replaceFromWith**(**"direct:csv.unmarshal"**);**

interceptSendToEndpoint**(**"direct:routeOrders"**).**skipSendToOriginalEndpoint**().**to**(**"mock:out"**);**

**}**

**});**

context**.**start**();**

**}**

This will retrieve the “receiveOrderCsv.transport” route from the context and replaces the from endpoint with “direct:csv:unmarshal” and modify the “direct:routeOrders” endpoint to “mock:out”

Finish the unittest by adding the following code:

// a dummy test request

final String request **=** "1,10,100,nl\n" **+** "2,1,10,us\n" **+** "3,100,1000,nl\n"**;**

MockEndpoint mockOut **=** getMockEndpoint**(**"mock:out"**);**

//create an assertion that the mockOut endpoint receives 3 messages

mockOut**.**expectedMessageCount**(**3**);**

//send the test message to the route

template**.**sendBody**(**"direct:csv.unmarshal"**,**request**);**

//check one message for expected output

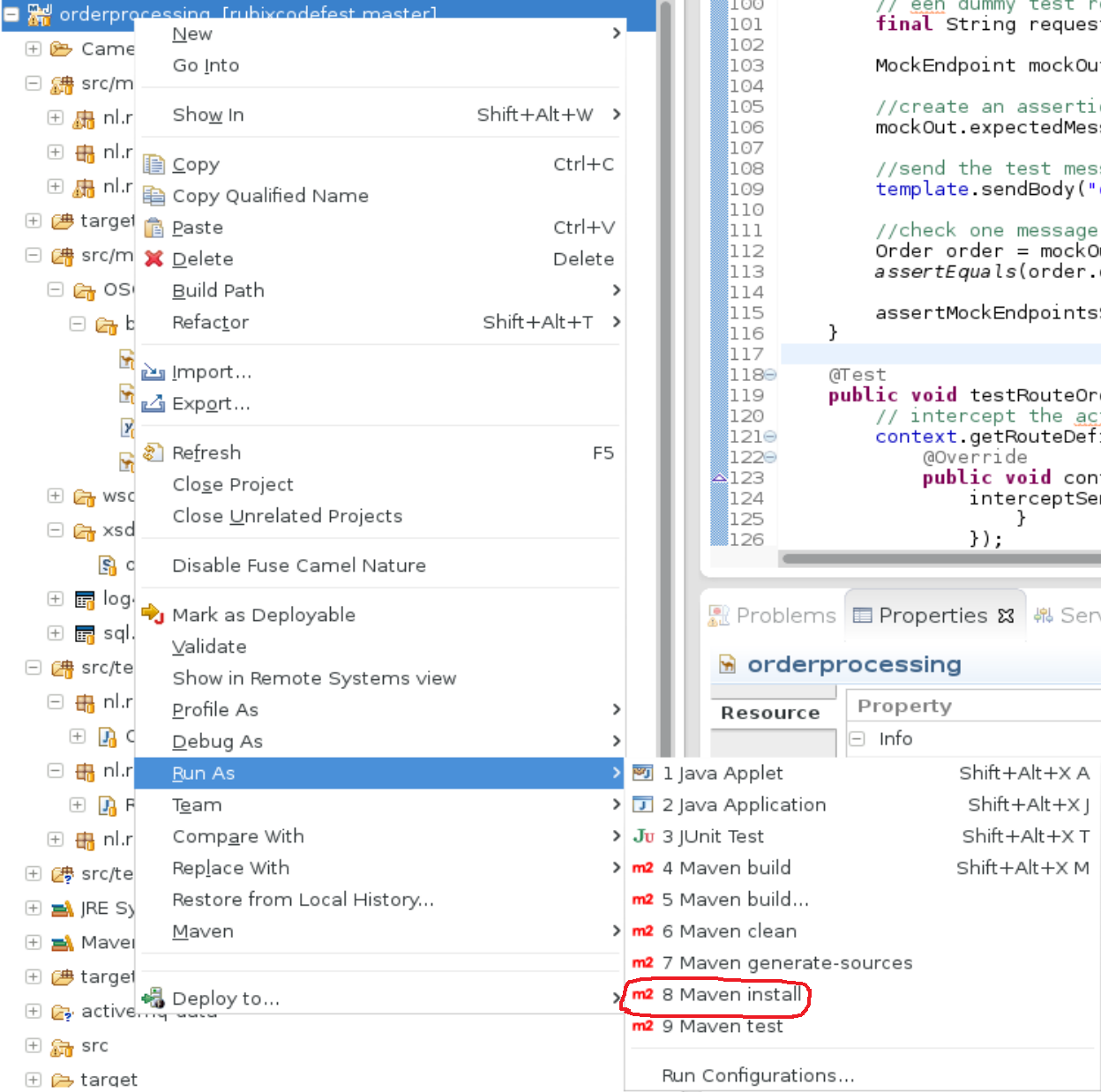
Order order **=** mockOut**.**getExchanges**().**get**(**0**).**getIn**().**getBody**(**Order**.**class**);**

assertEquals**(**order**.**getOrderId**(),**"1"**);**

assertMockEndpointsSatisfied**();**

press Ctrl+Shift+O to resolve the missing imports and save the unittest.

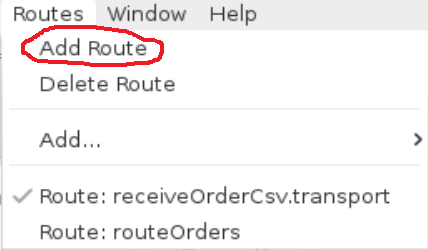
Run the unittest by executing mvn install.



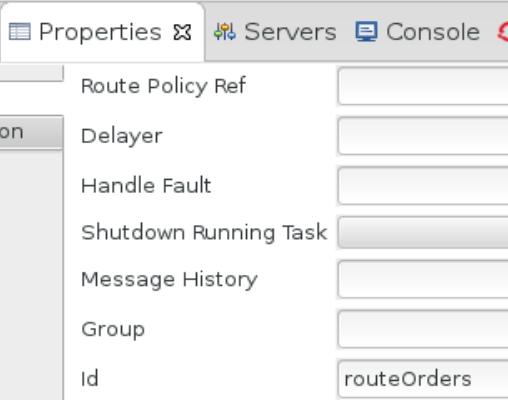
If everything is implemented correctly a SUCCESS message should appear.

## Creating the second route

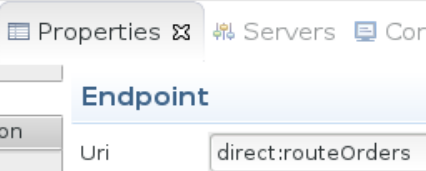
To finish up the first use case we need to implement a second route. To create another route open the ReceiveOrders.xml Context and in the top menu select routes -> add route



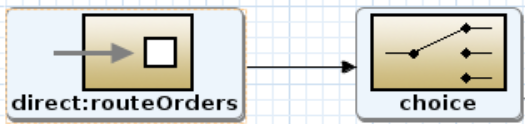
In the properties view enter a custom id: routeOrders



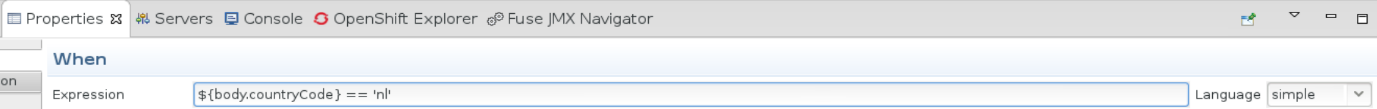
In the design view drag an endpoint from the Palette to the canvas and enter the following URI: direct:routeOrders



Next drag a choice from the Palette (From the routing section) connect the endpoint to the choice.



Next drag a when from the Palette onto the canvas and connect the choice to the when. In the properties view of the when enter the expression ${body.countryCode} == 'nl' and verify the language is set to simple.



Note, using the simple expression language we can access instance variables of the object residing in the body of the message. In this case the object in the body is the generated JAXB object of Order.

After the when drag a marshal from the Palette and connect the when to the marshal. Before we can configure the marshal we have to extend the dataformats we have configured in our Camel context. We are going to extend the dataformat with our generated JAXB object.

In the source view of the context extend the dataformats tag so the end result looks like this:

<camelContext trace=**"false"** xmlns=**"http://camel.apache.org/schema/blueprint"**>

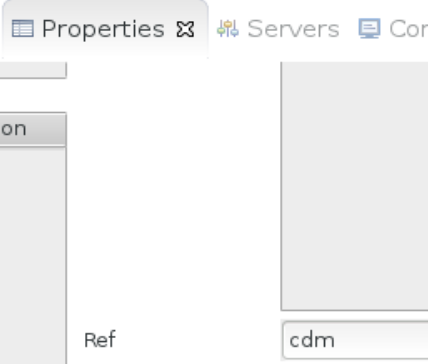
<dataFormats>

<bindy type=**"Csv"** packages=**"nl.rubix.codefest.orderprocessing.message"** id=**"csvordermodel"**/>

<jaxb contextPath=**"nl.rubix.orderprocessing.order.\_1"** prettyPrint=**"true"** id=**"cdm"**/>

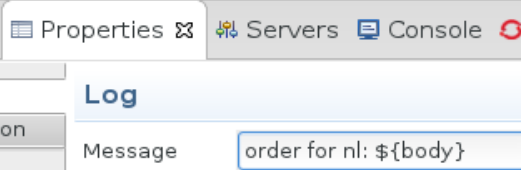
</dataFormats>

Now in the properties view of the marshal set the ref to cdm.

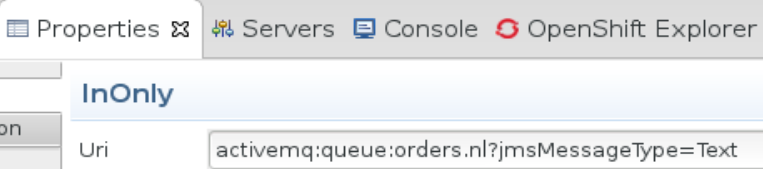


This will marshal the JAXB object into an xml document based on the order.xsd.

Next drag a log from the Palette onto the canvas and connect the marshal to the log. In the properties of the log enter the following message: order for nl: ${body}



Next drag an inOnly (from the transform section) onto the canvas. In the properties add the following URI: activemq:queue:orders.nl?jmsMessageType=Text



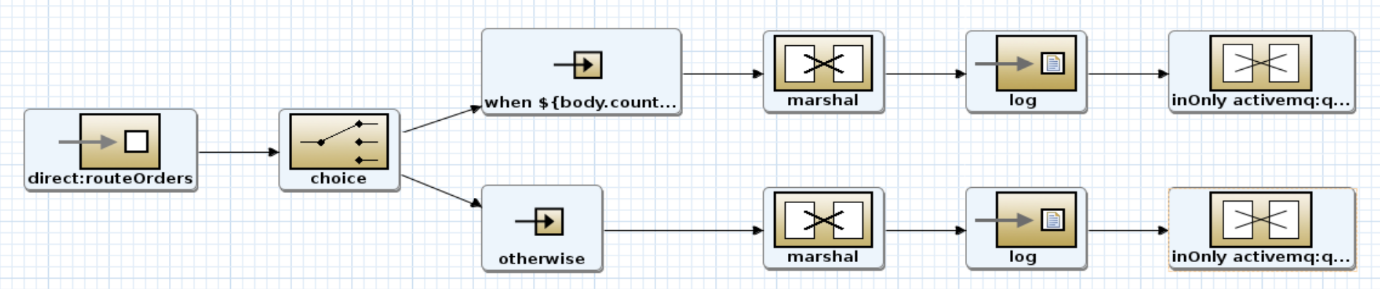
Note, the inOnly will set the message exchange pattern to inOnly so the endpoint is called in a fire and forget manner. In this case the message is published on a JMS queue hosted by ActiveMQ named orders.nl

We have finished the first branche of our choice now the second one.

Drag a otherwise from the Palette and connect it to the choice. After the otherwise, drag a marshal from the Palette and connect it to the otherwise. Again use cdm in the ref just like above. After the marshal drag a log from the Palette and enter the following message: international order

After the log drag a inOnly to the canvas and configure the following endpoint URI: activemq:queue:orders.int?jmsMessageType=Text

Save the Context. The route should look like this in the design view:



In the source view the route looks like this:

<route customId=**"true"** id=**"routeOrders"**>

<from uri=**"direct:routeOrders"**/>

<choice>

<when>

<simple>**${body.countryCode} == 'nl'**</simple>

<marshal ref=**"cdm"**/>

<log message=**"order for nl: ${body}"**/>

<inOnly uri=**"activemq:queue:orders.nl?jmsMessageType=Text"**/>

</when>

<otherwise>

<marshal ref=**"cdm"**/>

<log message=**"international order"**/>

<inOnly uri=**"activemq:queue:orders.int?jmsMessageType=Text"**/>

</otherwise>

</choice>

</route>

## Extending the test class

Now we have a second route we can extend our unittest class with additional test methods testing our new route.

For the new unittest we are again going through the same procedure used in the first unittest of mocking the endpoints using an interceptor. We will than send a dummy request message to the route and observe the routing is performed correctly.

### Creating a new unittest

In the test class create a new method and annotate the method with @Test.

@Test

public void testRouteOrders**()** **throws** Exception**{**

since the route already uses a direct endpoint we only have to mock the activemq endpoint.

// intercept the activemq endpoint so we can use it in the unittest

context**.**getRouteDefinition**(**"routeOrders"**).**adviceWith**(**context**,** **new** AdviceWithRouteBuilder**()** **{**

@Override

public void configure**()** **throws** Exception **{**

interceptSendToEndpoint**(**"activemq:queue:orders.nl?jmsMessageType=Text"**).**skipSendToOriginalEndpoint**().**to**(**"mock:out"**);**

**}**

**});**

context**.**start**();**

To implement the functionality of the unittest add the following code:

//create a test message

Order testRequest **=** **new** Order**();**

testRequest**.**setOrderId**(**"1"**);**

testRequest**.**setTotalAmount**(new** BigInteger**(**"100"**));**

testRequest**.**setTotalPrice**(new** BigDecimal**(**100.1**));**

testRequest**.**setCountryCode**(**"nl"**);**

//fetch the mockendpoint

MockEndpoint mockOut **=** getMockEndpoint**(**"mock:out"**);**

//state how many messages we expect our mockendpoint to receive

mockOut**.**setExpectedMessageCount**(**1**);**

//send the testrequest to the from endpoint of the route

template**.**sendBody**(**"direct:routeOrders"**,** testRequest**);**

//verify our expected message count

assertMockEndpointsSatisfied**();**

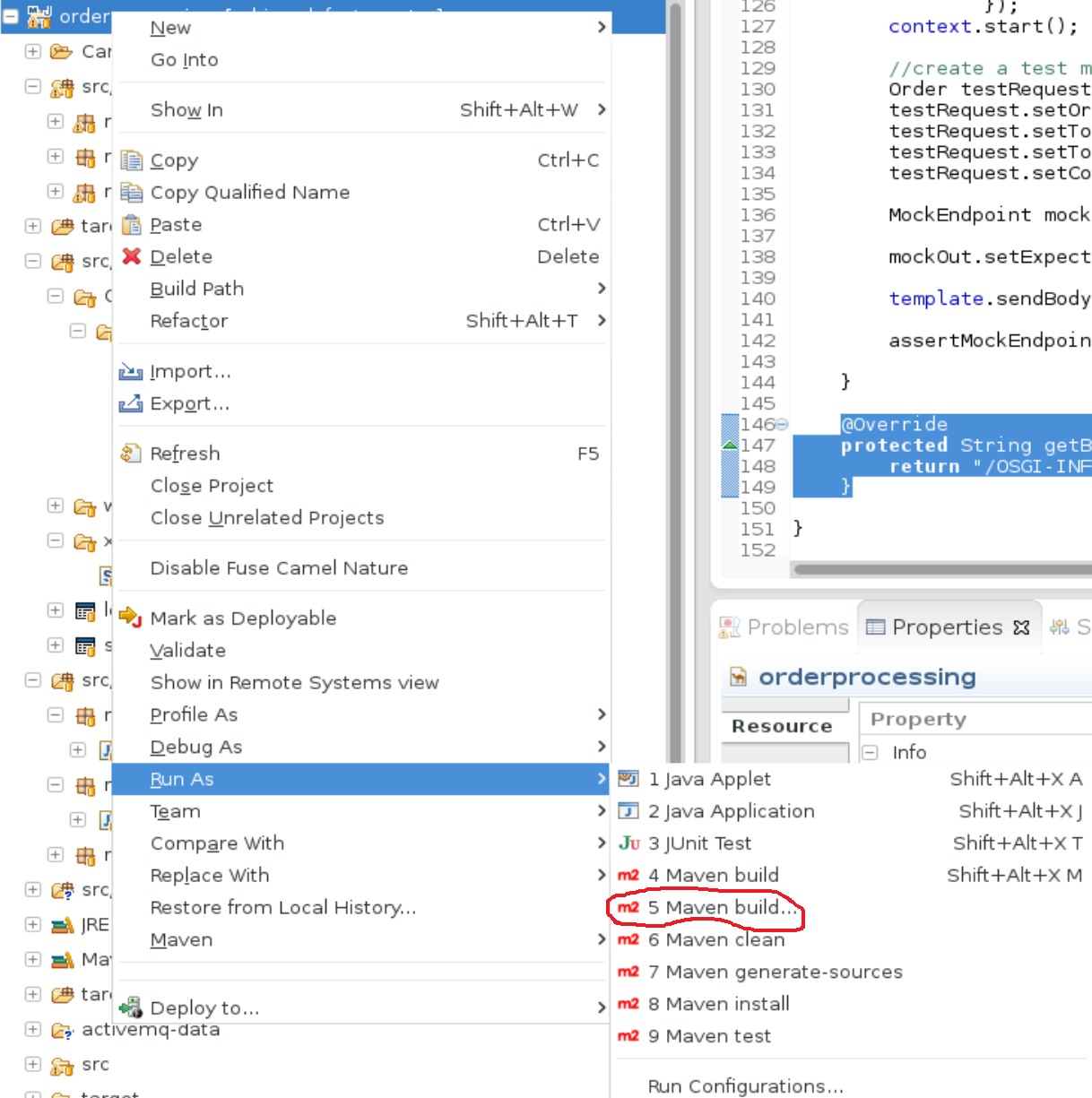
Save the unittest and run mvn Install to run all the unittest again.

## Running the entire project within JBDS

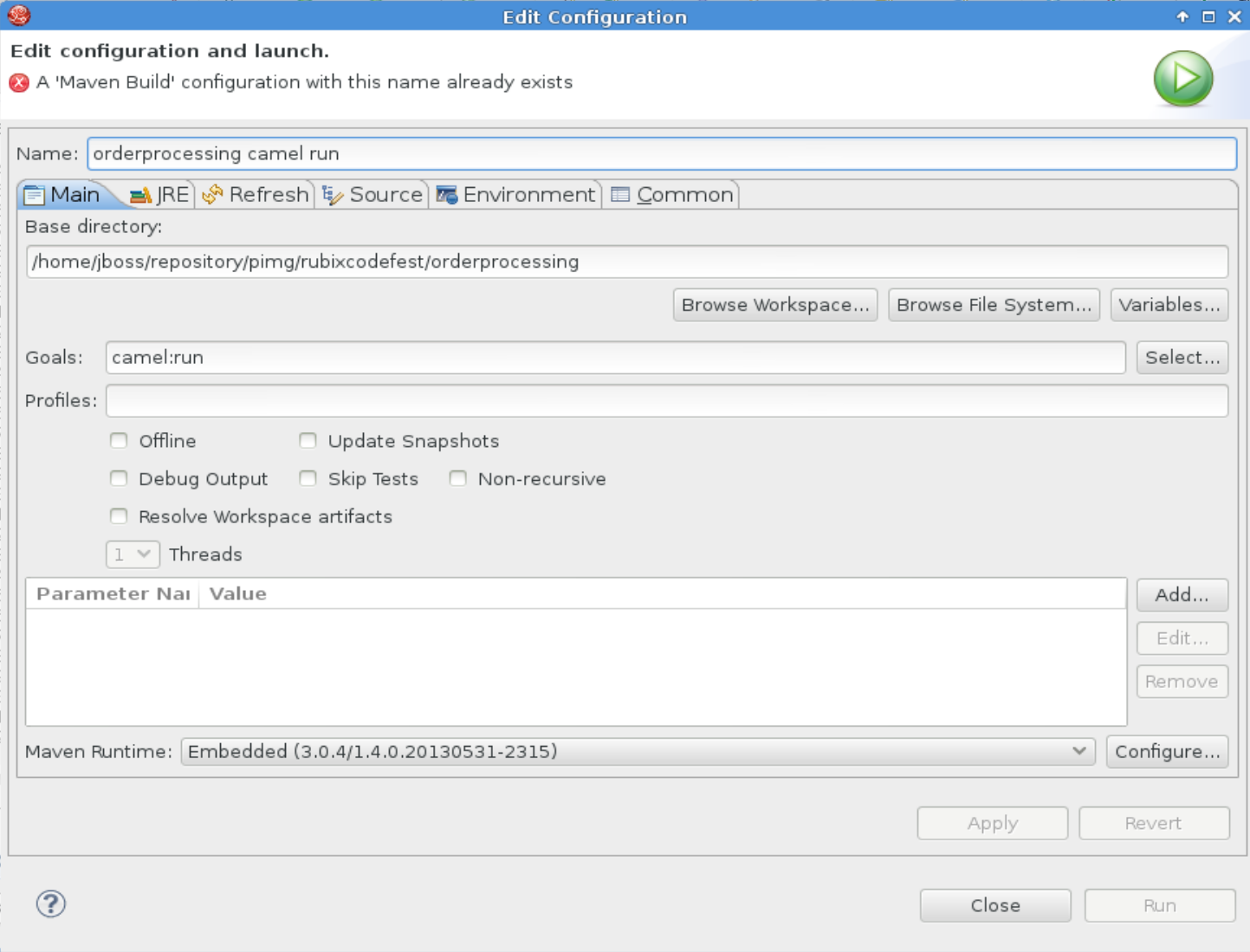
It is also possible to run the entire Fuse project within JBDS. This is done by executing the maven goal camel:run

However this maven goal is not present in the run as menu, so we have to configure a custom maven goal.

Right click on the project -> run as -> Maven build…



Enter camel run in the name and camel:run in the goal and click run



Now the blueprint container is started and all Camel Contexts present in the Fuse projects are loaded. After the container is started you can test both routes by dropping the orders.csv in the folder you configured in the from endpoint in the first route. When the file is consumed observe the logging output in the console.

Congratulations you finished the first use case!!! Now it is time to get started with the second use case.

1. Even though some resources are not required for the first use case, they will have to be present in the project. For example, the WSDL is only used in use case 3. However, the Maven plugin in the POM expects the WSDL to be present in the specified location and will throw an error if it is not present. [↑](#footnote-ref-1)
2. This location differs based on the framework used. For example Spring uses META-INF instead of OSGI\_INF. [↑](#footnote-ref-2)