RED HAT® JBOSS® MIDDLEWARE

redhat.

RED HAT® JBOSS® FUSE

Hands-on Labs

Author	**Pablo Szuster** **Wayne Toh**
Version	**3**
Fuse Version:	**FIS 2.0**

Index

RedHat OpenShift Introduction

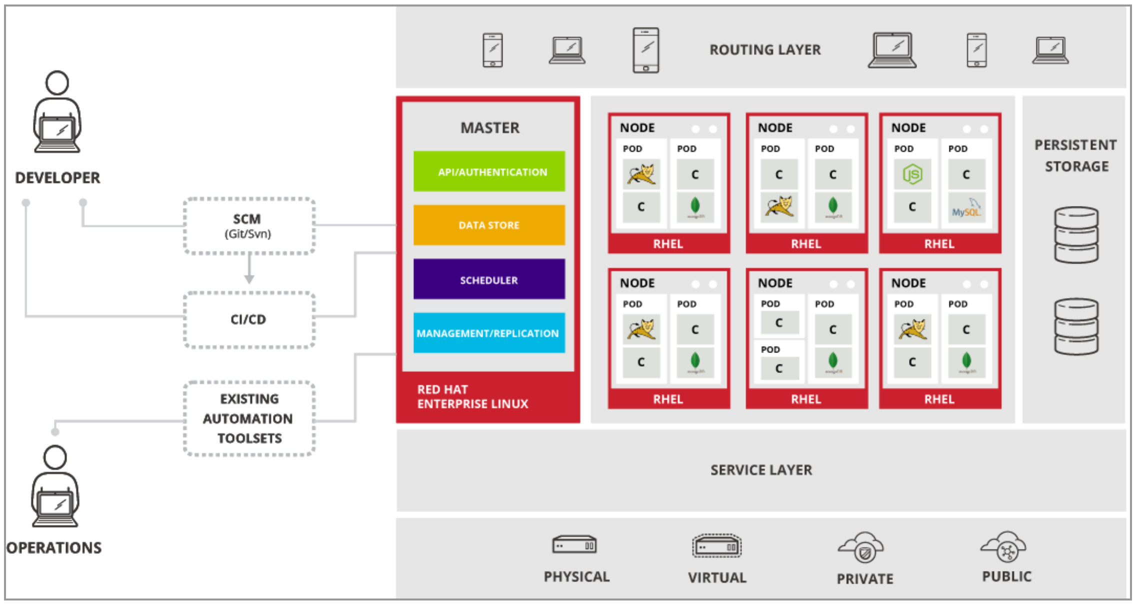
OpenShift Enterprise by Red Hat is a Platform as a Service (PaaS) that provides developers and IT organizations with a cloud application platform for deploying new applications on secure, scalable resources with minimal configuration and management overhead. OpenShift Enterprise supports a wide selection of programming languages and frameworks, such as Java, Ruby, and PHP.

Built on Red Hat Enterprise Linux and Google Kubernetes, OpenShift Enterprise provides a secure and scalable multi-tenant operating system for today's enterprise-class applications, while providing integrated application runtimes and libraries. OpenShift Enterprise brings the OpenShift PaaS platform to customer data centers, enabling organizations to implement a private PaaS that meets security, privacy, compliance, and governance requirements.

OpenShift v3 is a layered system designed to expose underlying Docker and Kubernetes concepts as accurately as possible, with a focus on easy composition of applications by a developer. For example, install Ruby, push code, and add MySQL.

Docker provides the abstraction for packaging and creating Linux-based, lightweight containers. Kubernetes provides the cluster management and orchestrates Docker containers on multiple hosts. OpenShift Enterprise adds:

- Source code management, builds, and deployments for developers
- Managing and promoting images at scale as they flow through your system
- Application management at scale
- Team and user tracking for organizing a large developer organization



Within OpenShift Enterprise, Kubernetes manages containerized applications across a set of containers or hosts and provides mechanisms for deployment, maintenance, and

application-scaling. Docker packages, instantiates, and runs containerized applications. A Kubernetes cluster consists of one or more masters and a set of nodes.
The master is the host or hosts that contain the master components, including the API server, controller manager server, and etcd. The master manages nodes in its Kubernetes cluster and schedules pods to run on nodes.

A node provides the runtime environments for containers. Each node in a Kubernetes cluster has the required services to be managed by the master. Nodes also have the required services to run pods, including Docker, a kubelet, and a service proxy. OpenShift Enterprise creates nodes from a cloud provider, physical systems, or virtual systems.

CORE CONCEPTS
Container
The basic units of OpenShift Enterprise applications are called containers. Linux container technologies are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources. Many application instances can be running in containers on a single host without visibility into each others' processes, files, network, and so on. Typically, each container provides a single service (often called a "micro-service"), such as a web server or a database, though containers can be used for arbitrary workloads.

The Linux kernel has been incorporating capabilities for container technologies for years. More recently the Docker project has developed a convenient management interface for Linux containers on a host. OpenShift Enterprise and Kubernetes add the ability to orchestrate Docker containers across multi-host installations.

Docker containers are based on Docker images. A Docker image is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing its needs and capabilities. You can think of it as a packaging technology. Docker containers only have access to resources defined in the image unless you give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, OpenShift Enterprise can provide redundancy and horizontal scaling for a service packaged into an image.

You can use Docker directly to build images, but OpenShift Enterprise also supplies builders that assist with creating an image by adding your code or configuration to existing images.

Docker Registry
A Docker registry is a service for storing and retrieving Docker images. A registry contains a collection of one or more Docker image repositories. Each image repository contains one or more tagged images. Docker provides its own registry, the Docker Hub, but you may also use private or third-party registries. Red Hat provides a Docker registry at registry.access.redhat.com for subscribers. OpenShift Enterprise can also supply its own internal registry for managing custom Docker images.
Pod

OpenShift Enterprise leverages the Kubernetes concept of a pod, which is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

Pods are the rough equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Pods have a lifecycle; they are defined, then they are assigned to run on a node, then they run until their container(s) exit or they are removed for some other reason. Pods, depending on policy and exit code, may be removed after exiting, or may be retained in order to enable access to the logs of their containers.

OpenShift Enterprise treats pods as largely immutable; changes cannot be made to a pod definition while it is running. OpenShift Enterprise implements changes by terminating an existing pod and recreating it with modified configuration, base image(s), or both. Pods are also treated as expendable, and do not maintain state when recreated. Therefore pods should usually be managed by higher-level controllers, rather than directly by users.

Service
A Kubernetes service serves as an internal load balancer. It identifies a set of replicated pods in order to proxy the connections it receives to them. Backing pods can be added to or removed from a service arbitrarily while the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent internal address.

Services are assigned an IP address and port pair that, when accessed, proxy to an appropriate backing pod. A service uses a label selector to find all the containers running that provide a certain network service on a certain port.

Project
A project is a Kubernetes namespace with additional annotations, and is the central vehicle by which access to resources for regular users is managed. A project allows a community of users to organize and manage their content in isolation from other communities. Users must be given access to projects by administrators, or if allowed to create projects, automatically have access to their own projects.

Build
A build is the process of transforming input parameters into a resulting object. Most often, the process is used to transform input parameters or source code into a runnable image. A BuildConfig object is the definition of the entire build process.

OpenShift Enterprise leverages Kubernetes by creating Docker containers from build images and pushing them to a Docker registry.

Build objects share common characteristics: inputs for a build, the need to complete a build process, logging the build process, publishing resources from successful builds, and publishing the final status of the build. Builds take advantage of resource restrictions, specifying limitations on resources such as CPU usage, memory usage, and build or pod execution time.

The OpenShift Enterprise build system provides extensible support for build strategies that are based on selectable types specified in the build API. There are three build strategies available:

- Docker build: invokes the plain docker build command, and it therefore expects a repository with a Dockerfile and all required artifacts in it to produce a runnable image.
- Source-to-Image (S2I) build: is a tool for building reproducible Docker images. It produces ready-to-run images by injecting application source into a Docker image and assembling a new Docker image. The new image incorporates the base image (the builder) and built source and is ready to use with the docker run command. S2I supports incremental builds, which re-use previously downloaded dependencies, previously built artifacts, etc.
- Custom build: The Custom build strategy allows developers to define a specific builder image responsible for the entire build process. Using your own builder image allows you to customize your build process.

Replication Controller
A replication controller ensures that a specified number of replicas of a pod are running at all times. If pods exit or are deleted, the replication controller acts to instantiate more up to the defined number. Likewise, if there are more running than desired, it deletes as many as necessary to match the defined amount.
A replication controller configuration consists of:

1. The number of replicas desired (which can be adjusted at runtime).
2. A pod definition to use when creating a replicated pod.
3. A selector for identifying managed pods.

Route
An OpenShift Enterprise route is a way to expose a service by giving it an externally-reachable hostname like www.example.com.

A defined route and the endpoints identified by its service can be consumed by a router to provide named connectivity that allows external clients to reach your applications. Each route consists of a route name (limited to 63 characters), service selector, and (optionally) security configuration.

xPaaS
JBoss xPaaS services provide the powerful capabilities of JBoss Middleware as cloud-based services on OpenShift. With the ability to run these services as containerized deployments across a variety of different environments, you can trust Red Hat JBoss Middleware to be your

solution of choice for enterprise applications.
(https://access.redhat.com/documentation/en/red-hat-jboss-middleware-for-openshift/)
Currently, the following are available:

- Red Hat JBoss Enterprise Application Platform
- Red Hat JBoss Web Server
- Red Hat JBoss A-MQ
- Red Hat JBoss Fuse (Fuse Integration Services)
- Red Hat JBoss BRMS (Decision Server)
- Red Hat JBoss Data Grid
- Red Hat JBoss SSO image.
- Red Hat JBoss Data Virtualization (Data Services)
- Red Hat JBoss BPMS (Process Server)
- Red Hat Java

Apache Camel Introduction

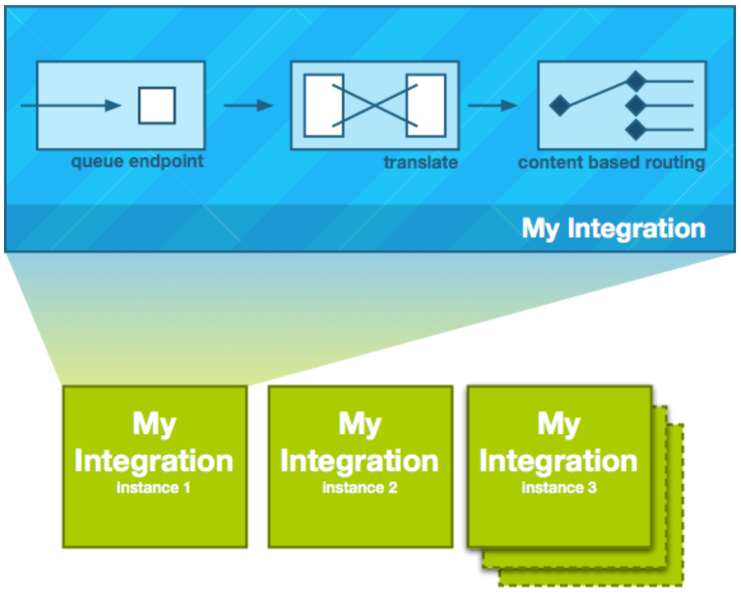
Building complex systems from scratch is a very costly endeavor, and one that's almost never successful. An effective and less risky alternative is to assemble a system like a jigsaw puzzle from existing, proven components. We depend daily on a multitude of such integrated systems, making possible everything from phone communications, financial transactions, and healthcare to travel planning and entertainment.
You can't finalize a jigsaw puzzle until you have a complete set of pieces that plug into each other simply, seamlessly, and robustly. That holds true for system integration projects as well. But whereas jigsaw puzzle pieces are made to plug into each other, the systems we integrate rarely are. Integration frameworks aim to fill this gap.
As an integrator, you're less concerned about how the system you integrate works and more focused on how to interoperate with it from the outside. A good integration framework provides simple, manageable abstractions for the complex systems you're integrating and the "glue" for plugging them together seamlessly. Apache Camel is such an integration framework.

At the core of the Camel framework is a routing engine, or more precisely a routing engine builder. It allows you to define your own routing rules, decide from which sources to accept messages, and determine how to process and send those messages to other destinations. Camel uses an integration language that allows you to define complex routing rules, akin to business processes. One of the fundamental principles of Camel is that it makes no assumptions about the type of data you need to process. This is an important point, because it gives you, the developer, an opportunity to integrate any kind of system, without the need to convert your data to a canonical format. Camel offers higher-level abstractions that allow you to interact with various systems using the same API regardless of the protocol or data type the systems are using. Components in Camel provide specific implementations of the API that target different protocols and data types. Out of the box, Camel comes with support for over 80 protocols and data types. Its extensible and modular architecture allows you to implement and seamlessly plug in support for your own protocols, proprietary or not.
These architectural choices eliminate the need for unnecessary conversions and make Camel not only faster but also very lean.

Fuse Integration Services Introduction

Red Hat JBoss Fuse Integration Services provides a set of tools and containerized xPaaS images that enable development, deployment, and management of integration microservices within OpenShift.

So imagine you have an integration service with a route that starts with a message queue based endpoint, then you apply some transformation on the message, and based on its content, you decide where to go next.
When you deploy this integration service, it becomes an independent unit of schedule and scale on the OpenShift platform, allowing for easy addition or removal of more replicas of this route to make sure your SLAs are maintained.
Fuse Integration Services enables creation of pattern-based and modular microservices-style integration services that can be deployed, removed, and redeployed in a continuous fashion using the underlying OpenShift capabilities for more rapid connections.Also, each route deployment, no matter the number of replicas it might have, can be seen as a single unique service with its load being balanced across all of your replicas. Client applications see one endpoint, and the platform takes care of spreading the load.


queue endpoint
translate
content based routing
My Integration
My Integration instance 1
My Integration instance 2
My Integration instance 3

You can start using Fuse Integration Services by creating an application and deploying it to OpenShift using one of the following OpenShift Source-to-Image (S2I) application development workflows:

S2I binary workflow
S2I with build input from a binary source. This workflow is characterised by the fact that the build is partly executed on the developer's own machine. After building a binary package locally, this workflow hands off the binary package to OpenShift. For more details, see Binary Source from the OpenShift 3.3 Developer Guide.

S2I source workflow
S2I with build input from a Git source. This workflow is characterised by the fact that the build is executed entirely on the OpenShift server. For more details, see Git Source from the OpenShift 3.3 Developer Guide.

Red Hat JBoss Fuse provide two popular way to solve these problem by using two frameworks,

Spring Boot
Spring Boot is the most popular framework for microservice these days, due to it's simplicity to get started, by adding the Spring boot starter module that conveniently connects the dependency needed, and allowing developer to select the libraries they need in the creation time. To start the application, simple execute a command line to bring it up, yes, it's just a FAT JAR, without the redundant or unneeded dependency, therefore it's extremely lightweight.

Karaf
The OSGi technology is a set of specifications that define a dynamic component system for Java. These specifications enable a development model where applications are (dynamically) composed of many different (reusable) components

- The internal implementation is private to each bundle,
- The behaviour exposed by the bundle is described by its stated 'Capabilities',
- The dependencies a bundle has on its local environment are described by its stated 'Requirements'.
- Finally semantic versioning is used. A bundle's Capabilities are versioned (major.minor.micro). Meanwhile a bundle's Requirements specify the acceptable version ranges within which the Capabilities of third parties must fall.

Due to strong isolation OSGi bundles may be dynamically loaded or unloaded from a running JVM. As the bundles are self-describing, a process know as 'resolution' may be used to ensure that all inter-related bundles are automatically loaded into the runtime and wired together.

OSGi fit into the general microservices definition: lightweight communications, independently deployable etc, but they can also mitigate some of the criticisms of microservices cited above. Service boundaries are hard to predict in advance, so rather than forcing all services to run as separate processes, OSGi allows the decision to go remote to be deferred until deployment.

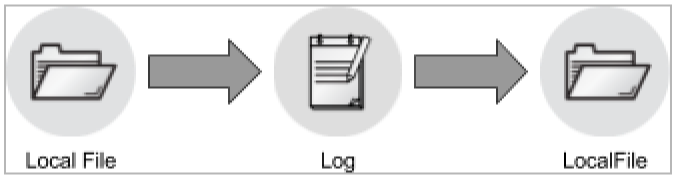
OSGi has a flexible capabilities model for handling service (and other) dependencies. Components specify the capabilities they provide and those they require, which facilitates automated resolution and deployment.

Notes

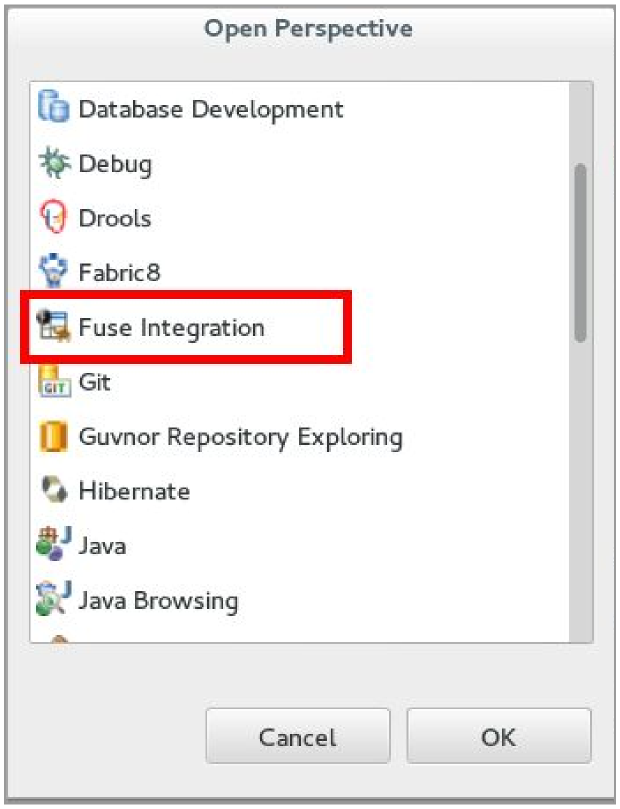
ⓘ	*You can access the documentation for every component by clicking on the Properties -> Documentation tab.*
ⓘ	After finishing a lab it is recommended to delete the corresponding OpenShift project.

Lab 1 - Files

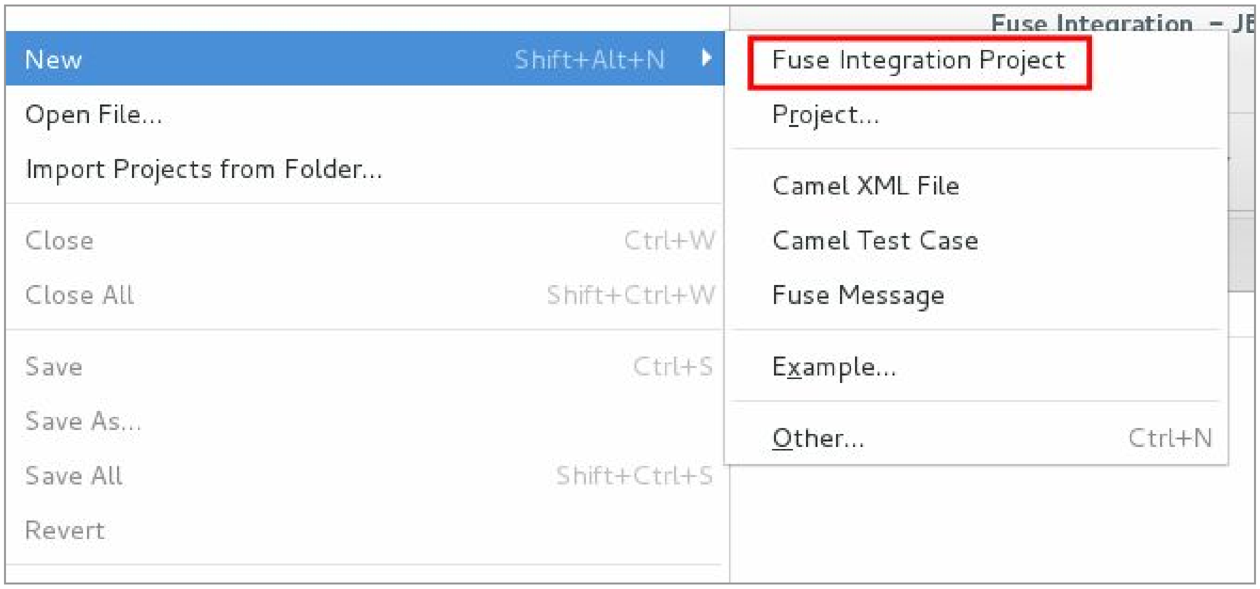
In this lab you will create a camel route that reads files from a file system directory, logs a message and transfer their content to another directory.



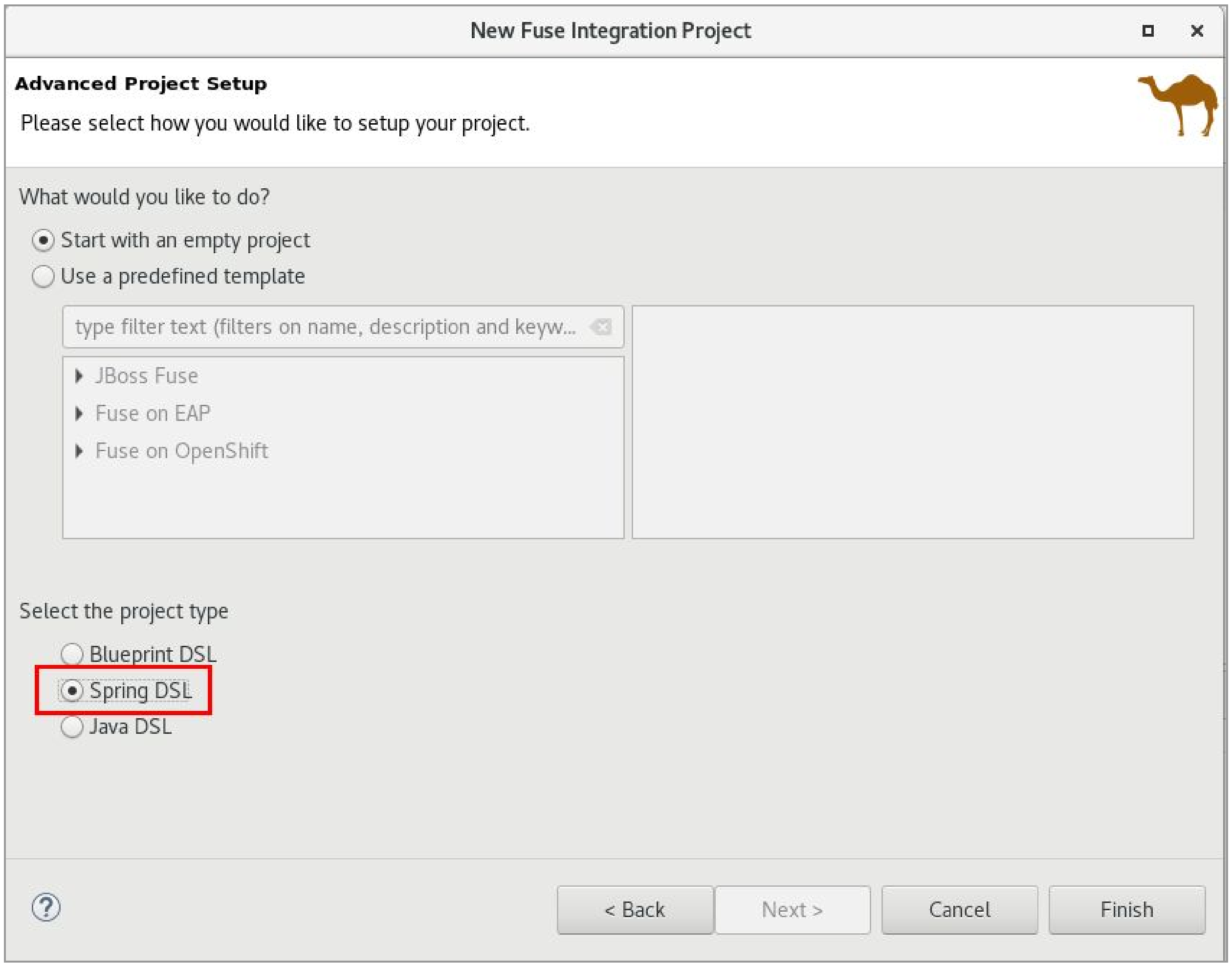
1. Switch to the **Fuse Integration** perspective.



2. Select **File -> New -> Fuse Integration Project**



3. Enter "**Lab1**" as the project name, and click **Next**.
4. Accept the defaults and click **Next**.
5. Select **Spring DSL** as the project type and click **Finish**.


New Fuse Integration Project

Advanced Project Setup

Please select how you would like to setup your project.

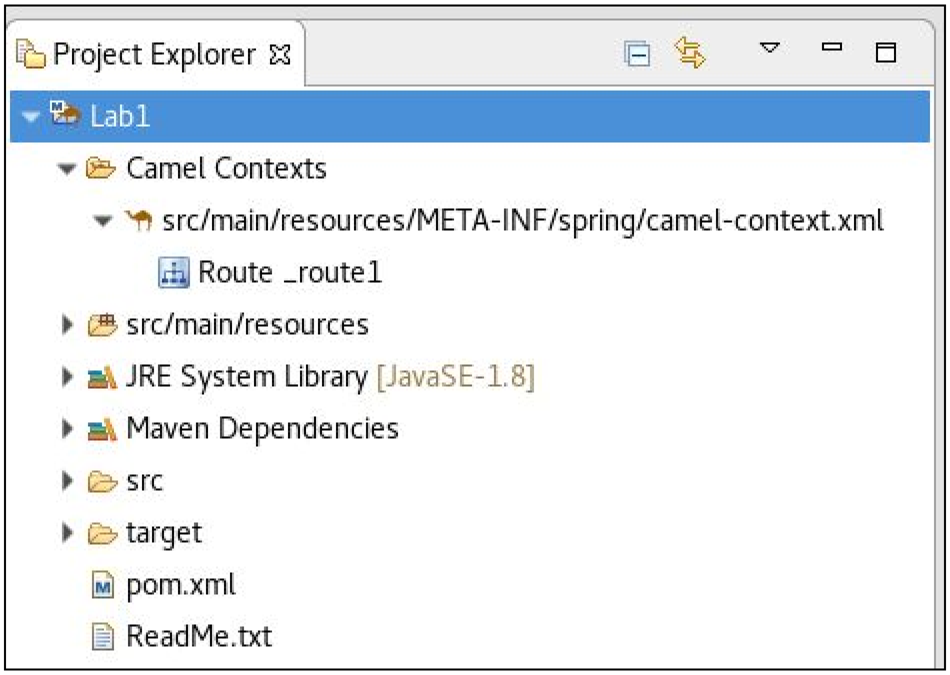
What would you like to do?

(●) Start with an empty project
( ) Use a predefined template

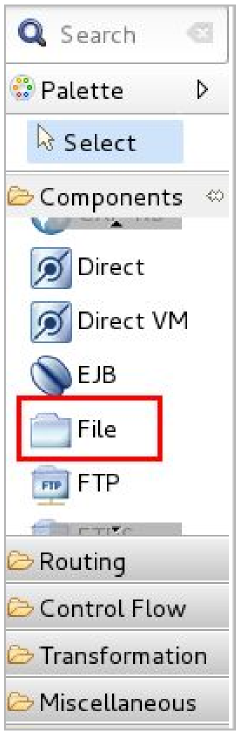
Select the project type

( ) Blueprint DSL
(●) Spring DSL
( ) Java DSL

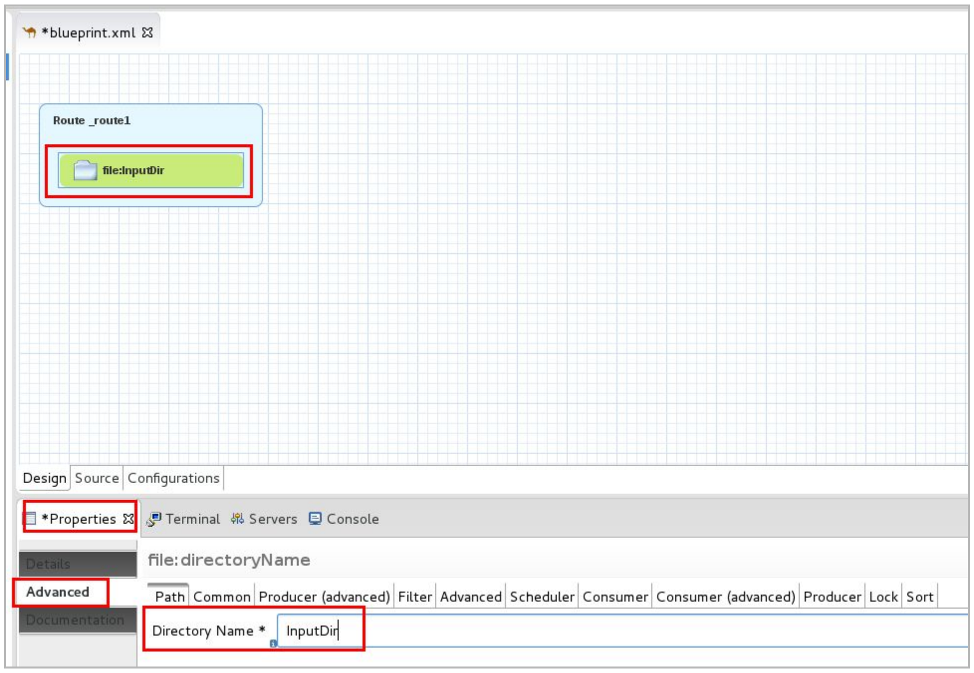
6. A new Fuse project is displayed in the Project Explorer:



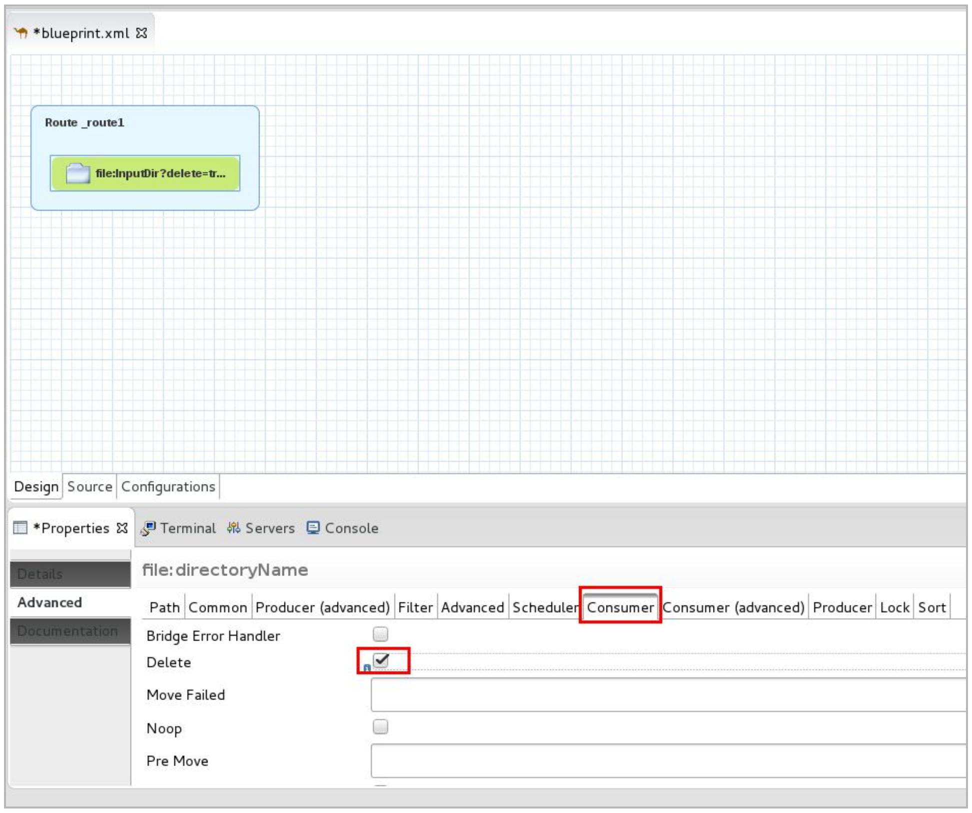
7. Double-click on the camel context file under **Camel Contexts**.
8. Click on the **Design** tab.
9. Drag a **File** component from the Components palette and drop it into the existing route.



   a. Click on the **Properties** tab -> **Advanced**
      i. **Path->Directory Name**: InputDir
      ii. **Consumer->Delete**: checked



Directory Name * InputDir

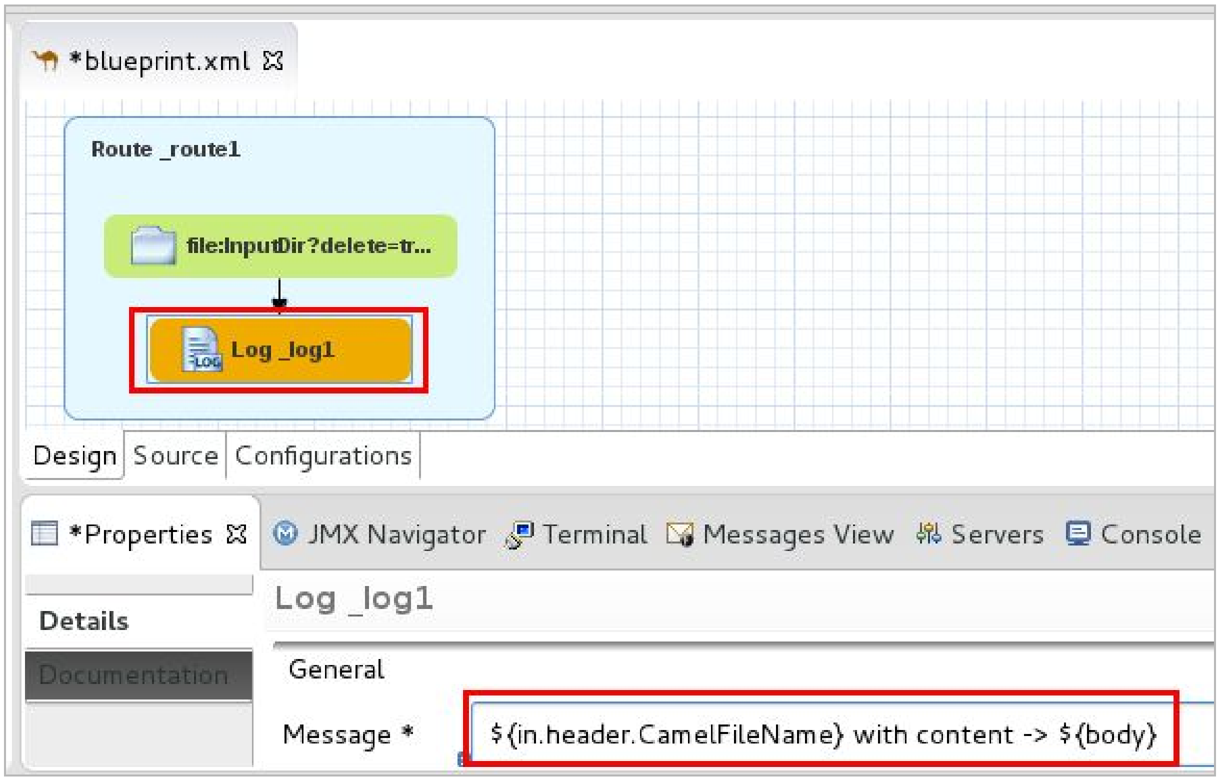


Bridge Error Handler	☐
Delete	✓
Move Failed	
Noop	☐
Pre Move	

The File component has a lot of configuration options, some are general , some apply when it acts as a Consumer (reads files) and some when it acts as a Producer (writes files).

	• *Directory Name: is the filesystem dir where it will be reading/writing files.* • *Consumer-> Delete means it will delete the file after processing it..*

10. Drag a **Log** component from the **Components** palette and drop it onto the file component.
11. Click on the **Properties** tab
    a. **Message**: ${in.header.CamelFileName} with content -> ${body}

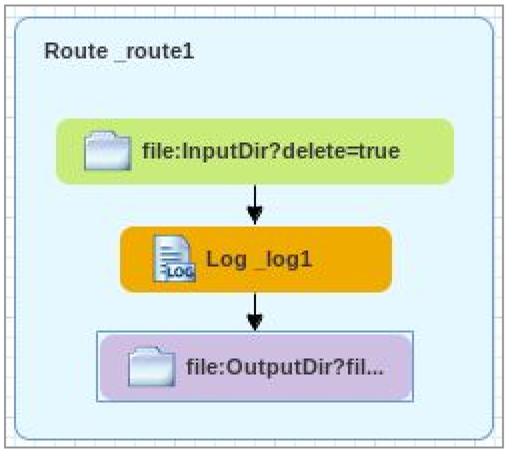


Message * ${in.header.CamelFileName} with content -> ${body}

ⓘ	*The Log component logs message exchanges to the underlying logging mechanism. You can use a simple language expression to construct the message.* *https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-component-reference/chapter-99-log*

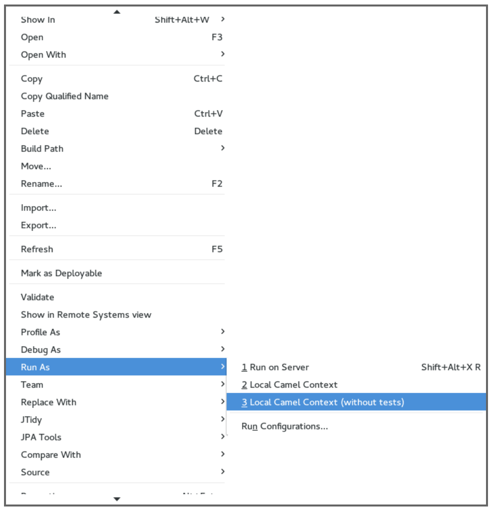
12. Drag a **File** component from the **Components** palette and drop it onto the **Log** component.
    a. Click on the **Properties** -> **Advanced** ->**Path**
        i. **Directory Name**: OutputDir
    b. Click on the **Properties** -> **Advanced** -> **Common**
        i. **File Name:** ${date:now:yyyyMMddhhmmss}-read.xml

13. Save the file (Ctrl+S).

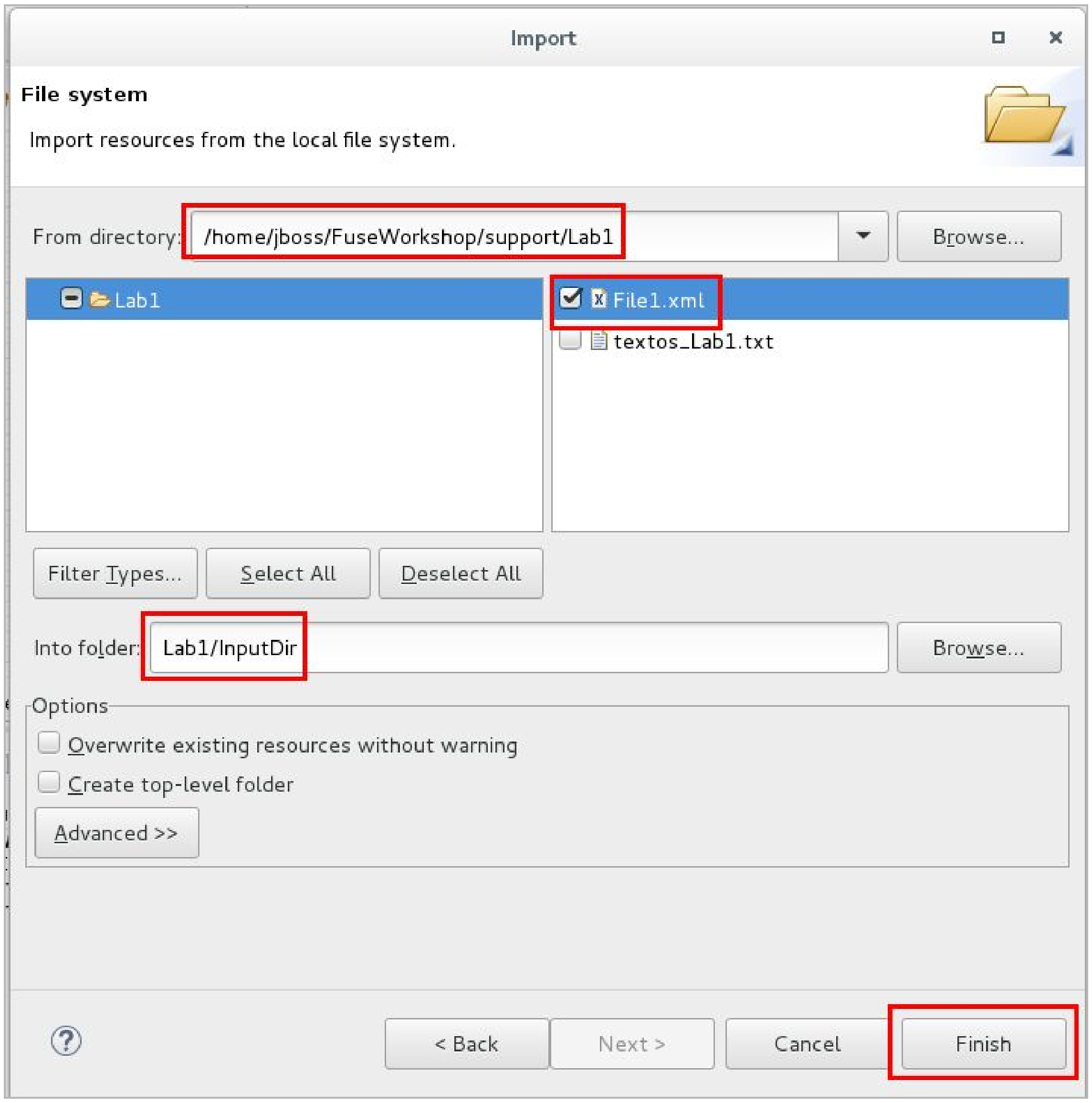


The expressions used in the properties of the Log and setHeader components are written in "simple expression language". They begin with a "${" and end with "}", and use reserved words to access parts of the message e.g: body, in.headers or functions e.g.: date. For more information: [*https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-development-guide/chapter-30-the-simple-language*](https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-development-guide/chapter-30-the-simple-language)

14. Right-click **camel-context.xml**, **Run As => Local Camel Context (without tests)**



15. Refresh the **Lab1** project by clicking on its name and pressing **F5**.
16. Right-click on the automatically created **InputDir** folder and click **Import..**
17. Select **General->File System** and click **Next**.
18. Browse to **~/FuseWorkshop/support/Lab1** and click **OK**.
19. Select **File1.xml** and click **Finish**.



From directory: /home/jboss/FuseWorkshop/support/Lab1

Lab1

☑ File1.xml
☐ textos_Lab1.txt

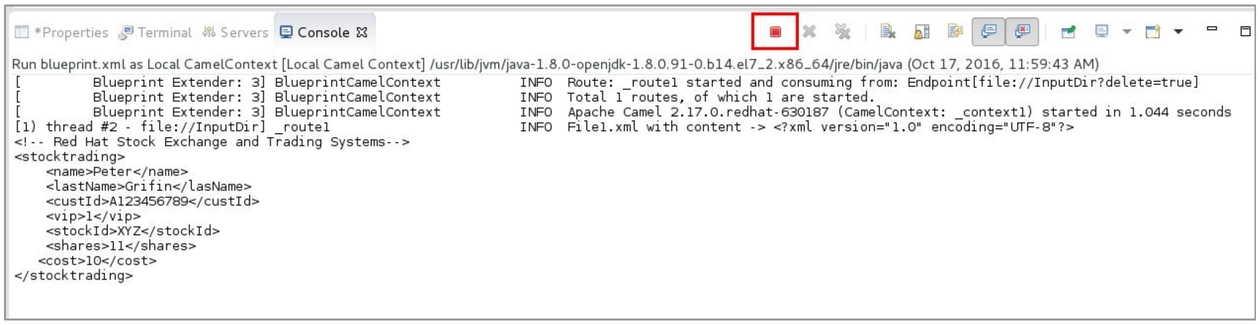
Into folder: Lab1/InputDir

Options

☐ Overwrite existing resources without warning

☐ Create top-level folder

20. Watch the console log and verify the route ran without errors.
21. Click on **Lab1** and press F5 to refresh.
22. Expand **OutputDir** folder, and verify there’s a new file with current date as its name.
23. Check its contents and delete it.
24. Close all open files.
25. Stop the Camel route

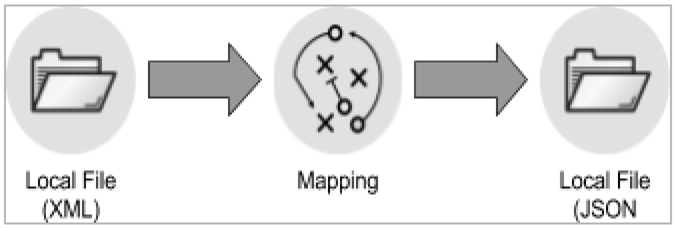


Run blueprint.xml as Local CamelContext [Local Camel Context] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.91-0.b14.el7_2.x86_64/jre/bin/java (Oct 17, 2016, 11:59:43 AM)

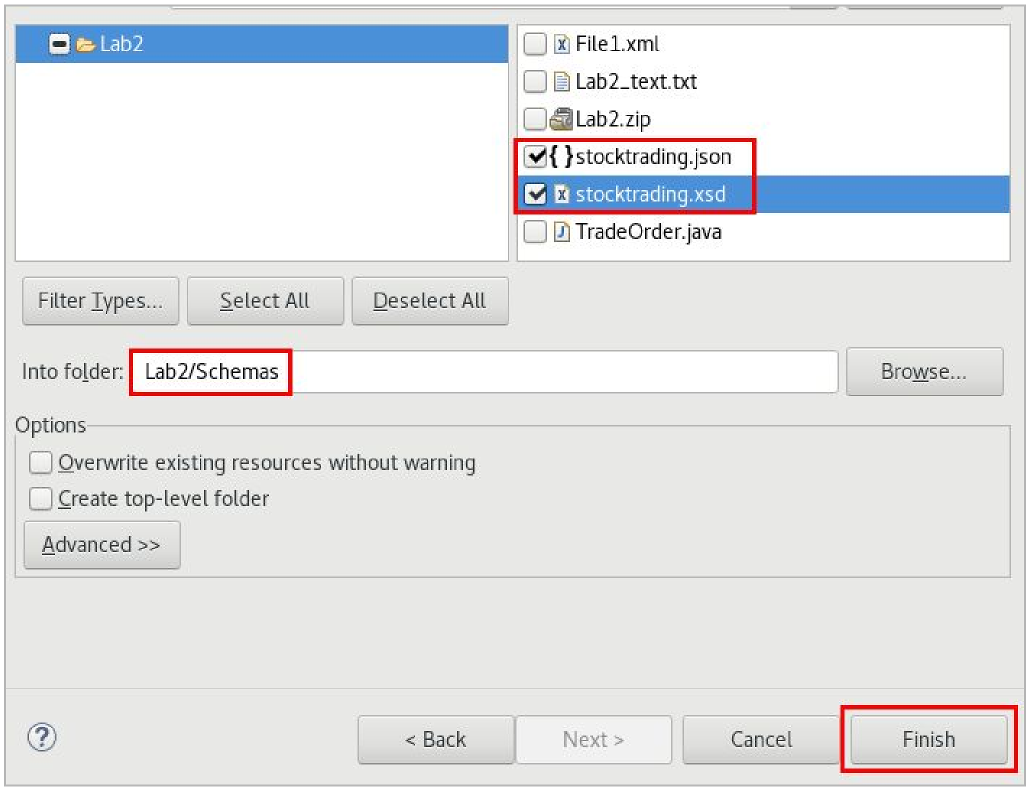
```
[         Blueprint Extender: 3] BlueprintCamelContext          INFO  Route: _route1 started and consuming from: Endpoint[file://InputDir?delete=true]
[         Blueprint Extender: 3] BlueprintCamelContext          INFO  Total 1 routes, of which 1 are started.
[         Blueprint Extender: 3] BlueprintCamelContext          INFO  Apache Camel 2.17.0.redhat-630187 (CamelContext: _context1) started in 1.044 seconds
[1) thread #2 - file://InputDir] _route1                        INFO  File1.xml with content -> <?xml version="1.0" encoding="UTF-8"?>
<!-- Red Hat Stock Exchange and Trading Systems-->
<stocktrading>
    <name>Peter</name>
    <lastName>Grifin</lasName>
    <custId>A123456789</custId>
    <vip>1</vip>
    <stockId>XYZ</stockId>
    <shares>11</shares>
  <cost>10</cost>
</stocktrading>
```

Lab 2 - Data Transformation

In this lab you will transform an XML file to a JSON file using a Data Mapper. This component has a visual transformation tool that allows you to map fields from the source structure to the target structure.



1. Click on the menu **File->Import..**
2. Select **General->** "**Existing Project into Workspace**" and click Next.
3. Click on "**Select archive file**" and browse to **~/FuseWorkshop/support/Lab2/Lab2.zip**
4. Click **Finish.**
5. Right-click on **Lab2** project and select **New -> Folder**
6. Enter "**Schemas**" and click **Finish.**
7. Right-click on the **Schemas** folder -> **Import**
8. Select **General -> File System** and browse to **~/FuseWorkshop/support/Lab2/**
9. Select **stocktrading.xsd** and **stocktrading.json** and click **Finish.**



Lab2

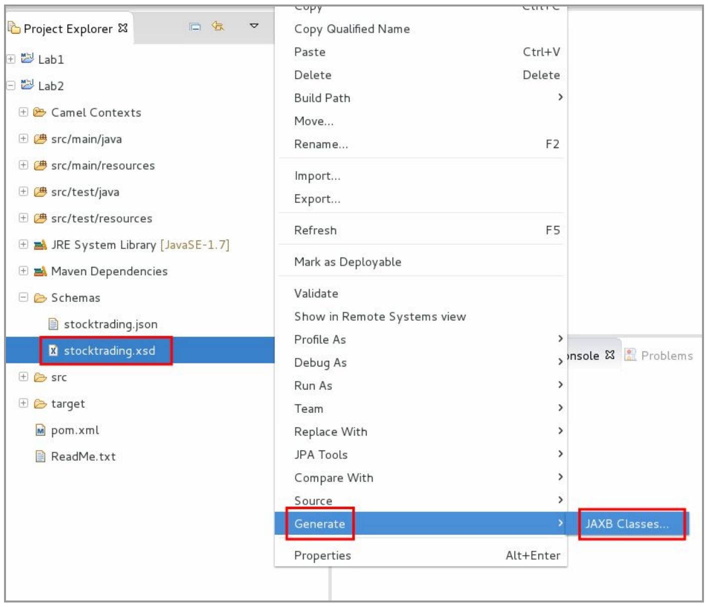
- [ ] File1.xml
- [ ] Lab2_text.txt
- [ ] Lab2.zip
- [x] stocktrading.json
- [x] stocktrading.xsd
- [ ] TradeOrder.java

Into folder: Lab2/Schemas

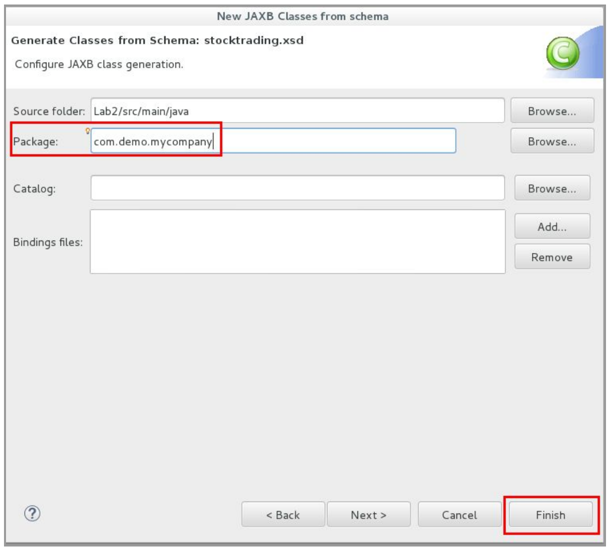
Options

- [ ] Overwrite existing resources without warning
- [ ] Create top-level folder

10. Right-click on the imported **stocktrading.xsd** file and select **Generate->JAXB Classes**



11. Select **Lab2** as destination project and click **Next**.
12. Enter **com.demo.mycompany** as package and click **Finish.**
13. Click **Yes** in the warning dialog.


New JAXB Classes from schema

Generate Classes from Schema: stocktrading.xsd

Configure JAXB class generation.

Source folder: Lab2/src/main/java

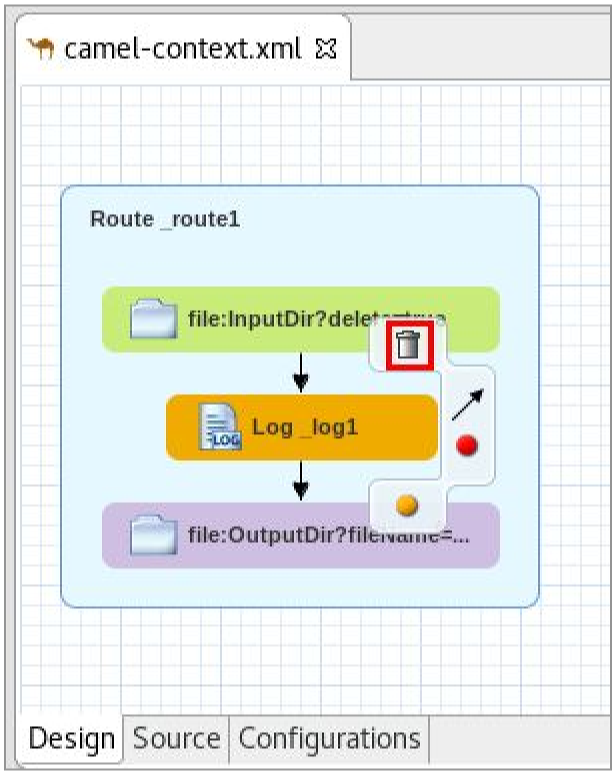
Package: com.demo.mycompany

Catalog:

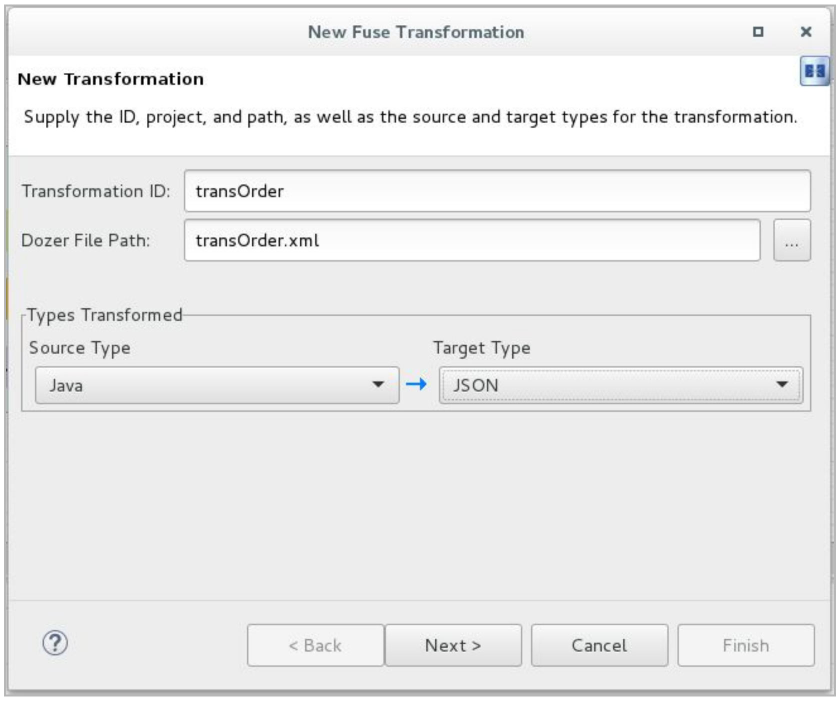
Bindings files:

	Java Architecture for XML Binding (JAXB) allows Java developers to map Java classes to XML representations. JAXB provides two main features: the ability to marshal Java objects into XML and the inverse, i.e. to unmarshal XML back into Java objects.

14. Open **camel-context.xml** to modify the Camel route.
15. Delete the **Log** component from the route.



16. Save the file (Ctrl+S).
17. Drag “**Data Transformation**” component from the **Transformation** palette and drop it onto the arrow between the **File** and the second **File** components.
18. In the **New Fuse Transformation** dialog, enter:
    a. **Transform ID**: transOrder
    b. **Dozer file path**: transOrder.xml
    c. **Source Type:** Java
    d. **Target Type**: JSON



New Transformation

Supply the ID, project, and path, as well as the source and target types for the transformation.

Transformation ID: transOrder

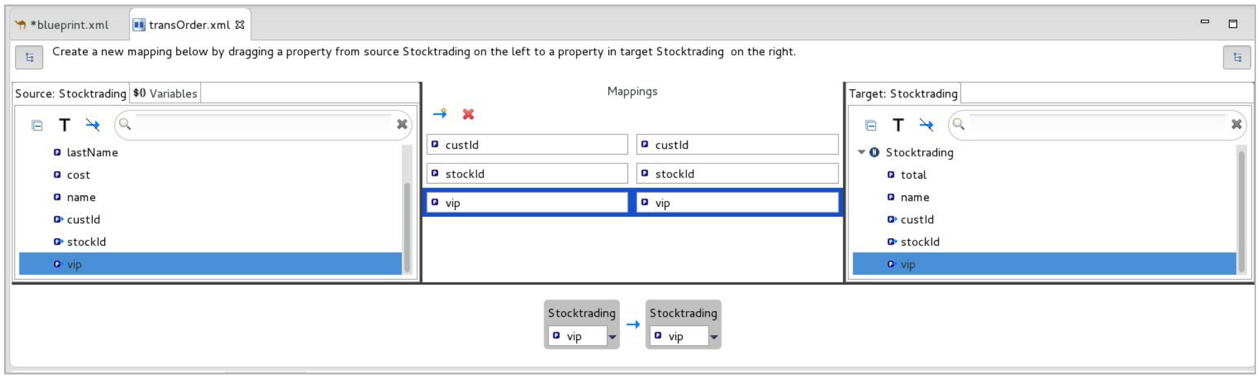
Dozer File Path: transOrder.xml

Types Transformed

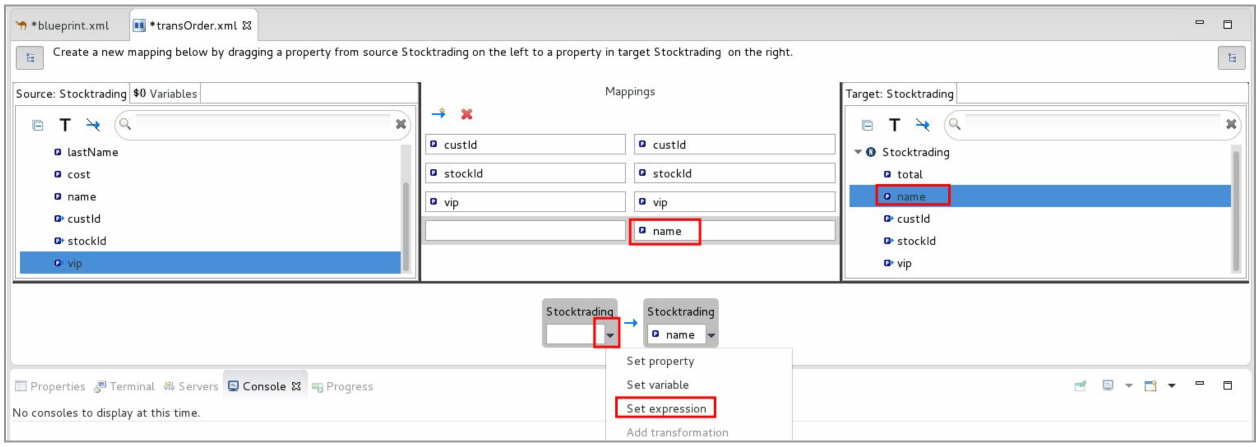
Source Type: Java

Target Type: JSON

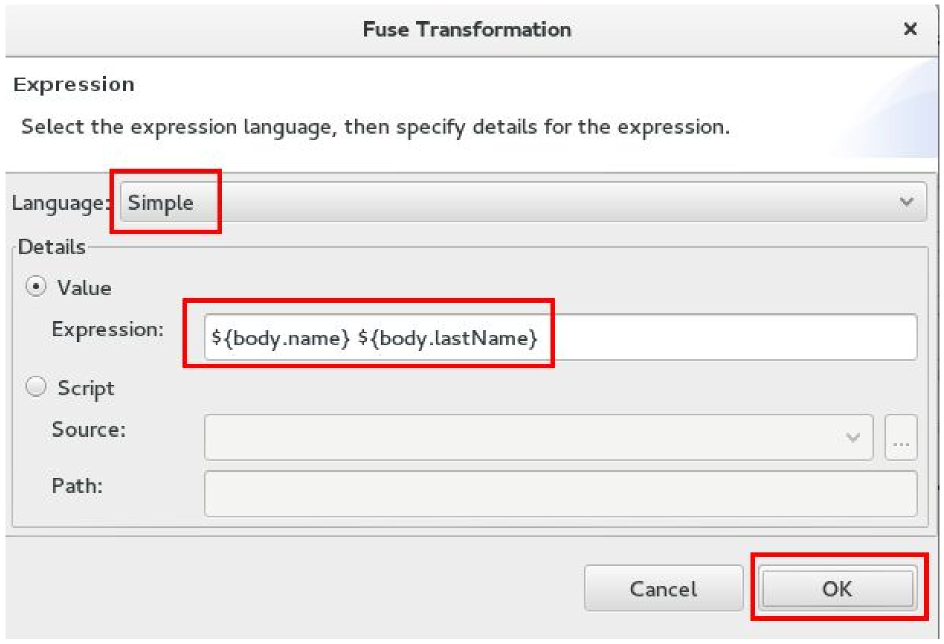
19. Click **Next**
20. **Source Type (Java)**:
    a. Click on the browse button [...] and enter **Stocktrading** in the dialog and select the first class in the list.
    b. Click **OK** and **Next.**
21. **Target Type (JSON)**:
    a. Select “JSON Instance Document”.
    b. Browse for the **stocktrading.json** file.
22. Click **Finish**.
23. In the **Data Transformation Editor**, link each of the following source fields with their corresponding target fields by dragging and dropping them:
    a. custId
    b. stockId
    c. vip



24. Create a new mapping by clicking on the button.
25. Drag and drop the **name** <u>target</u> field to the right box.
26. Click on the arrow in the left box and select **Set expression**.



27. Select **Simple** as the Language, and enter: ***${body.name} ${body.lastName}*** as the Expression.
28. Click **OK**.


Fuse Transformation

Expression

Select the expression language, then specify details for the expression.

Language: Simple

Details

Value

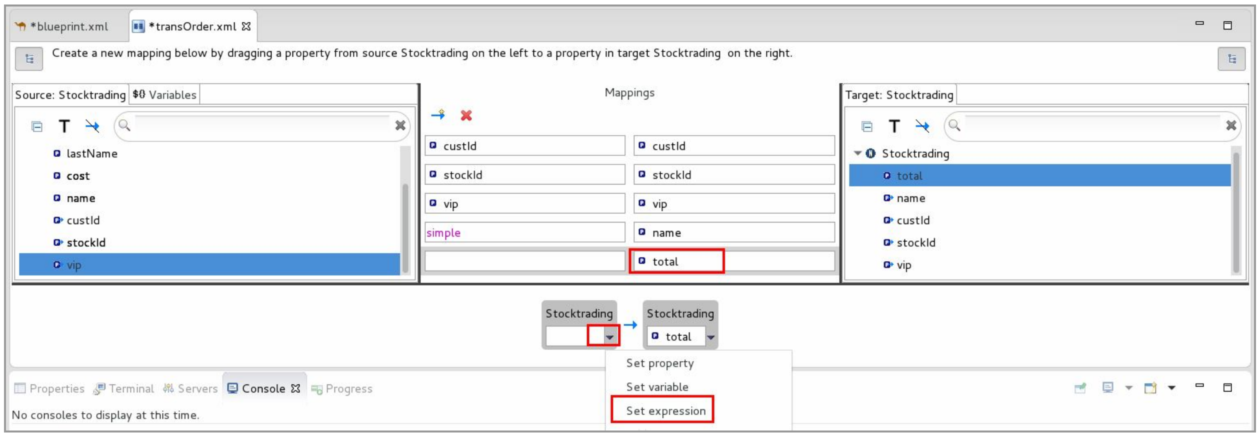
Expression: ${body.name} ${body.lastName}

Script

Source:

Path:

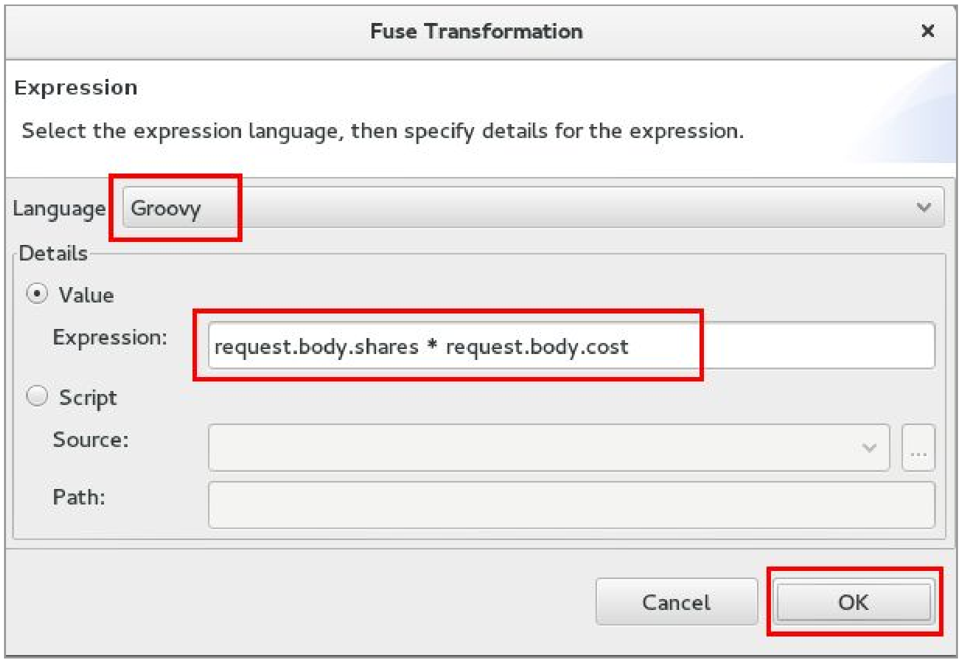
29. Create another mapping by clicking on the button again.
30. Drag and drop the **total** target field to the right box.
31. Click on the arrow in the left box and select **Set expression**.


Create a new mapping below by dragging a property from source Stocktrading on the left to a property in target Stocktrading on the right.

Source	Target
custId	custId
stockId	stockId
vip	vip
simple	name
	total

No consoles to display at this time.

32. Select **Groovy** as the Language, and enter the following as Expression: ***request.body.shares * request.body.cost***
33. Click **OK**.


Fuse Transformation

Expression

Select the expression language, then specify details for the expression.

Language: Groovy

Details

Value

Expression: request.body.shares * request.body.cost

Script

Source:

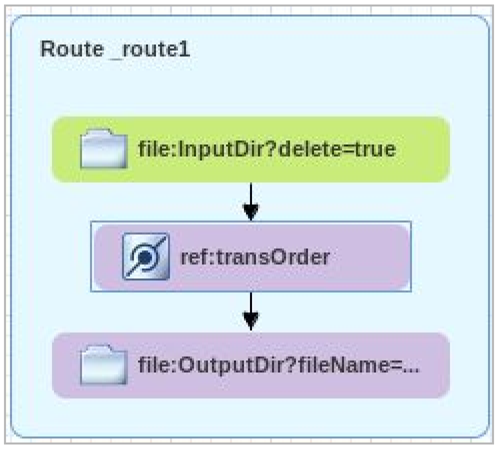
Path:

34. Close the Data Transformation editor.

ⓘ	*Groovy is a Java-based scripting language that allows quick parsing of object. The Groovy support is part of the camel-script module.* *https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-development-guide/chapter-17-groovy*

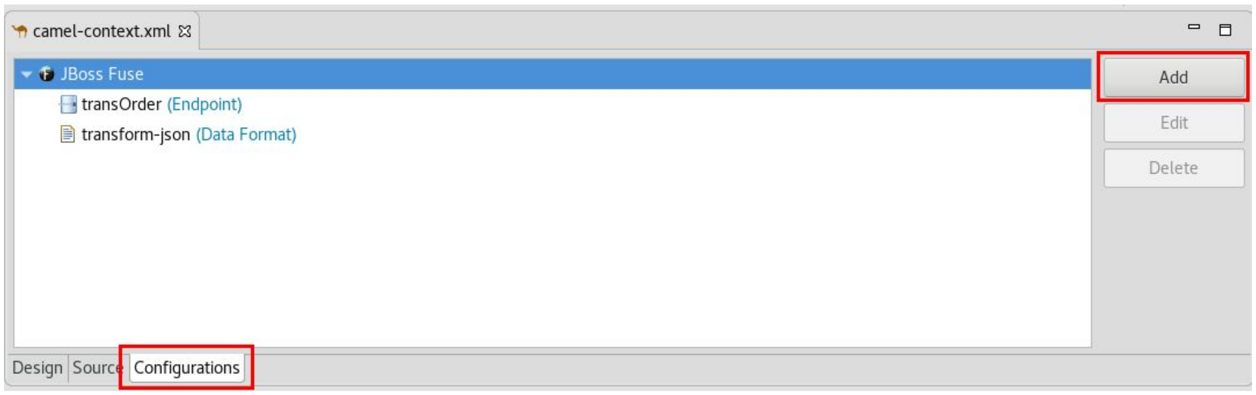
ⓘ	*http://dozer.sourceforge.net/* *Dozer is a Java Bean to Java Bean mapper that recursively copies data from one object to another. Typically, these Java Beans will be of different complex types.* *More information in:* *https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.2.1/html/Apache_Camel_Component_Reference/IDU-Dozer.html*

35. Save the file (Ctrl+S).

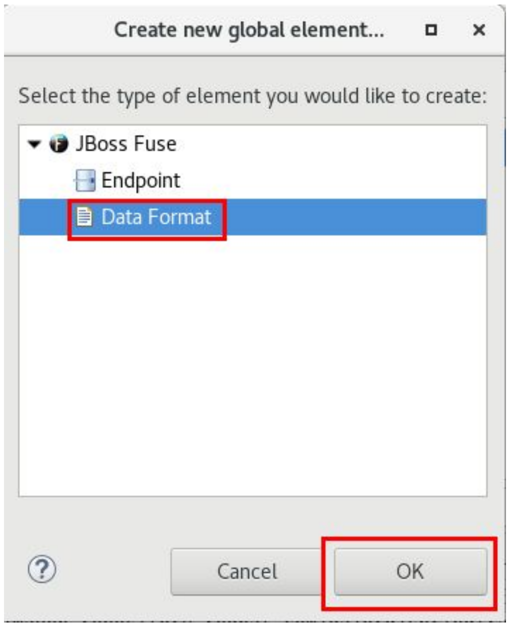


36. Click on the **Configurations** tab.

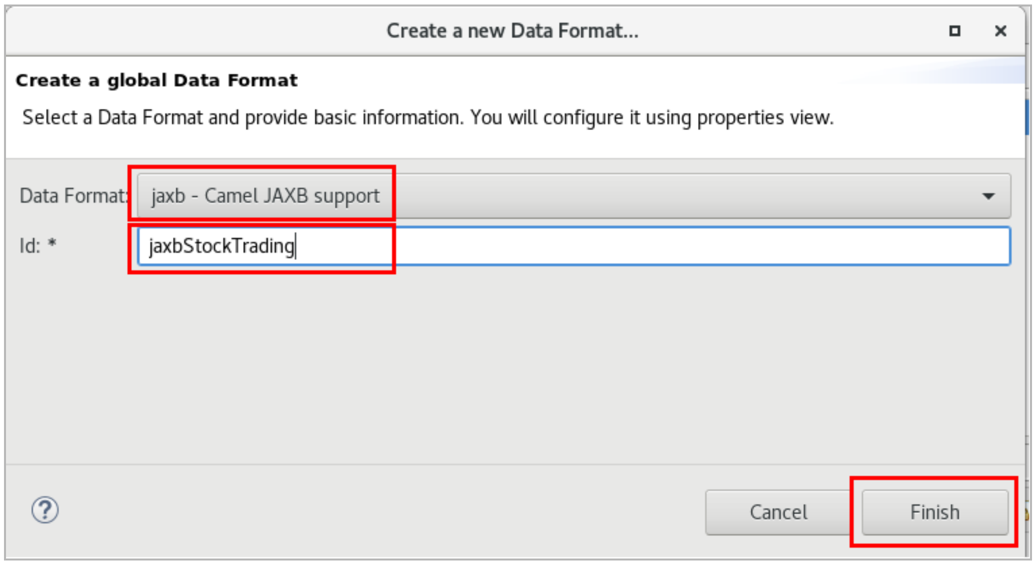
37. Click on the **Add** button.



38. Select **Data Format** and click **OK.**


Create new global element...
Select the type of element you would like to create:
JBoss Fuse
Endpoint
Data Format
Cancel
OK

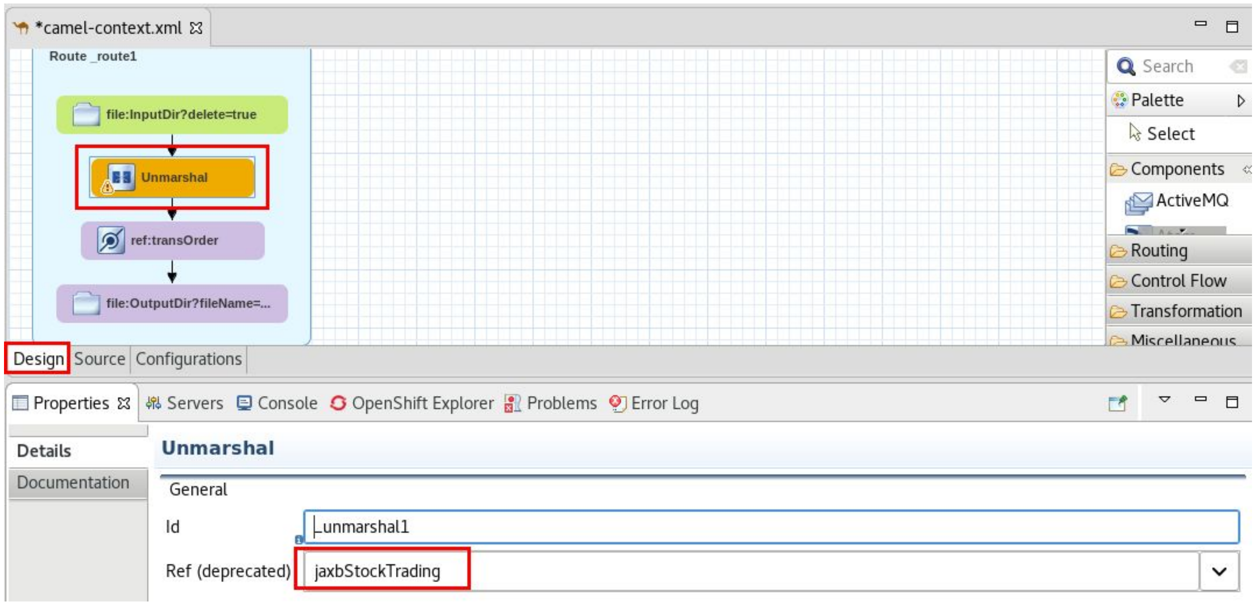
39. Select **jaxb** and enter **jaxbStockTrading** as the id.
40. Click **Finish**.


Create a new Data Format...
Create a global Data Format
Select a Data Format and provide basic information. You will configure it using properties view.
Data Format:
jaxb - Camel JAXB support
Id: *
jaxbStockTrading
Cancel
Finish

41. In the Properties tab, enter:

   a. **Context Path:** com.demo.mycompany

42. Click on the **Design** tab.
43. Drag an **Unmarshal** component from the **Transformation** palette and drop it onto the arrow between the **File** and **Data Transformation** components.
44. Click on the properties tab and select:
    a. **Ref: jaxbStockTrading**



Unmarshal

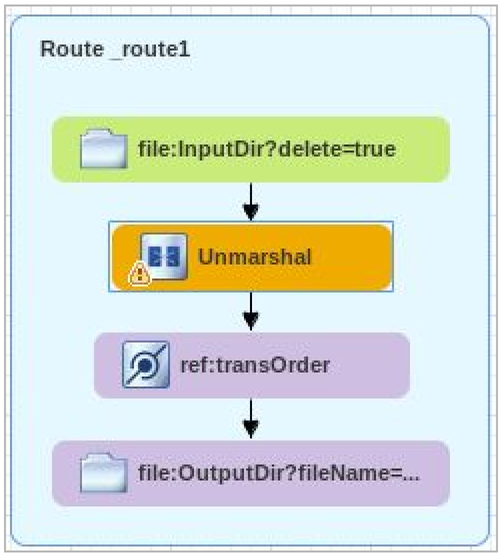
General

Id: _unmarshal1

Ref (deprecated): jaxbStockTrading

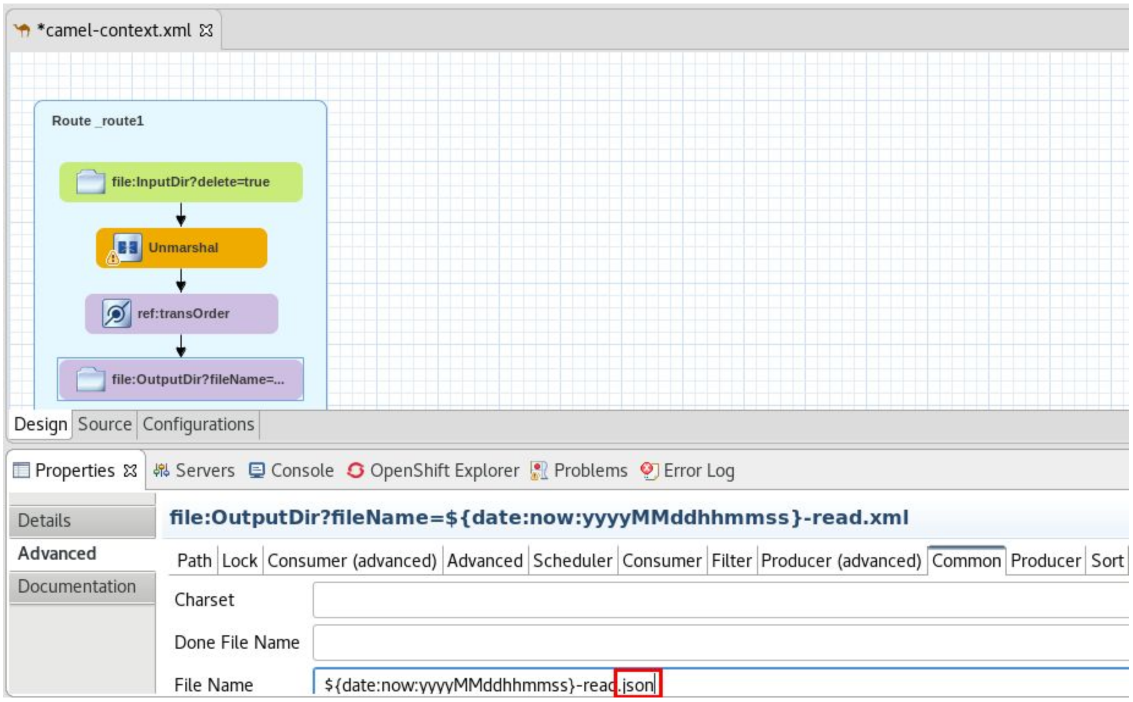
ⓘ	*The Marshal component serializes a bean to some binary or textual format for transmission over some transport via a Camel component.* *The Unmarshal component is the opposite, and parses a payload into a specified bean type.* *In both cases you have to define a data format to use (http://camel.apache.org/data-format.html)*

ⓘ	*This Unmarshal component will process the incoming XML file and generate a Java object using the JAXB annotated class generated earlier (com.demo.mycompany.Stocktrading)*



.

45. Edit the second **File** component's properties:
    a. **Advanced -> Common-> File Name**: change file extension to ".json"



file:OutputDir?fileName=${date:now:yyyyMMddhhmmss}-read.xml

Charset

Done File Name

File Name: ${date:now:yyyyMMddhhmmss}-read.json

46. Save the file (Ctrl+S).

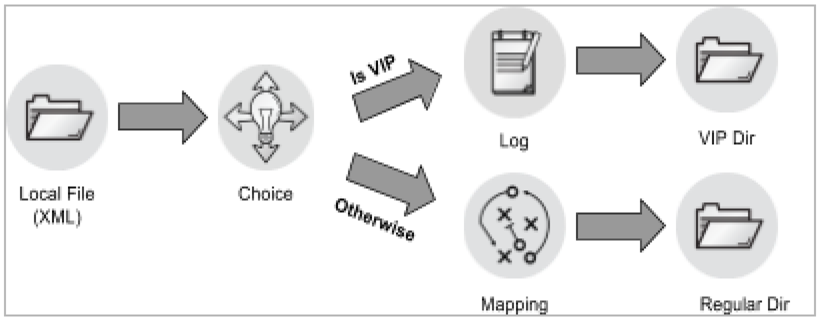
ⓘ	If an error appears in the **camel-context.xml,** click on the **Source** tab, search for the **objectFactory="false"** attribute and delete it.

47. Run the camel route by right-clicking **blueprint.xml**, **Run As -> "Local Camel Context (without tests)"**
48. Right-click on the **InputDir** folder and select **Import**.
    a. If the folder doesn't appear in the folder tree, refresh it (F5).
49. Browse to **~/FuseWorkshop/support/Lab2** and select **File1.xml**.
50. Click **Finish**.
51. Expand **OutputDir**.
52. Inspect the JSON file contents and verify that the **name** property is the concatenation of **name + last name**, and **total** is the multiplication of **shares** per **cost.**
53. Close all the open files.
54. Stop the camel route by clicking the stop button 🟥.

Lab 3 - Routing

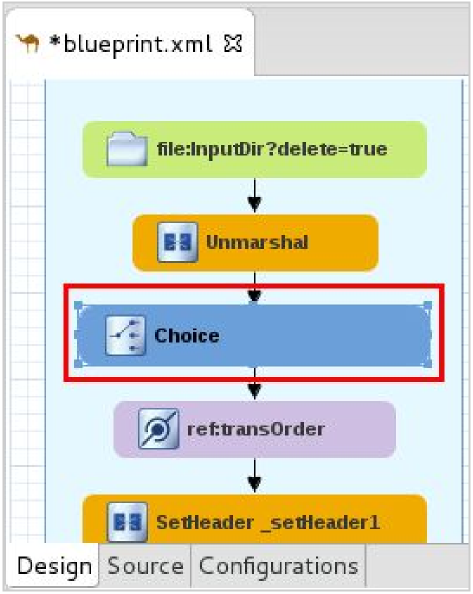
Part A - Content Based Routing

In this lab you will route the request based on its contents. If customer status is VIP, then the file will go to a VIP folder, otherwise it will be transformed to JSON and be sent to a Regular folder.

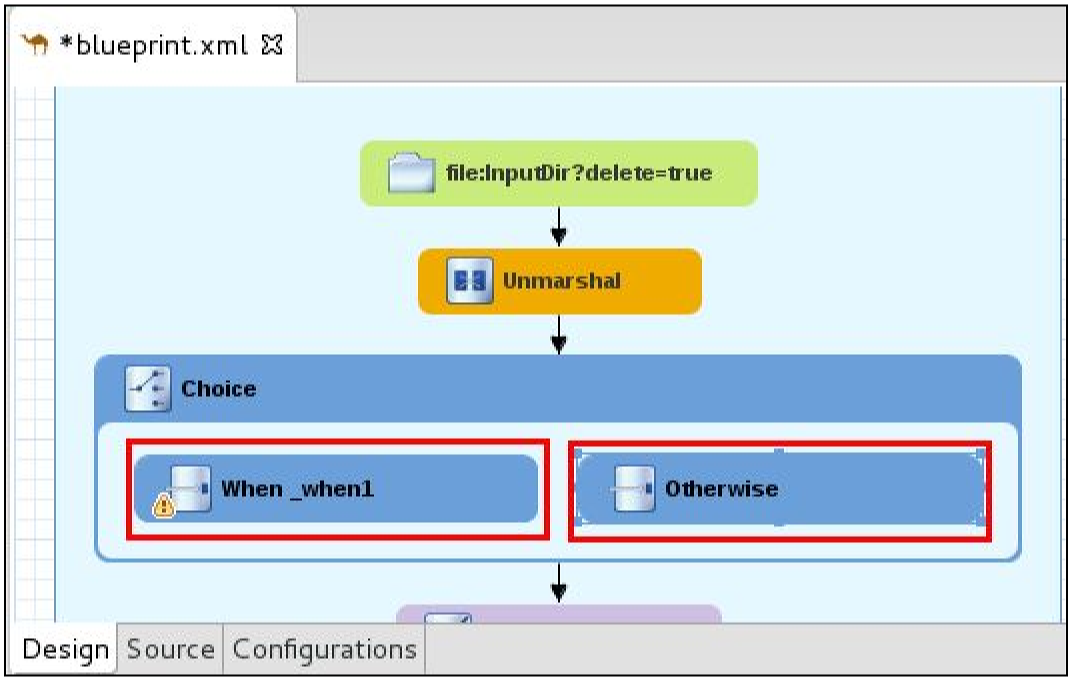


This is the implementation of the Content Based Router - Enterprise Integration Pattern (https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-development-guide/chapter-8-message-routing#MsgRout-Fig-CBRP)

1. Import an “Existing project into Workspace” from **~/FuseWorkshop/support/Lab3/Lab3.zip**”.
2. Open **camel-context.xml**.
3. Drag a **Choice** component from the **Routing** palette and drop it onto the arrow between the **Unmarshal** and the **Data Transformation** components.

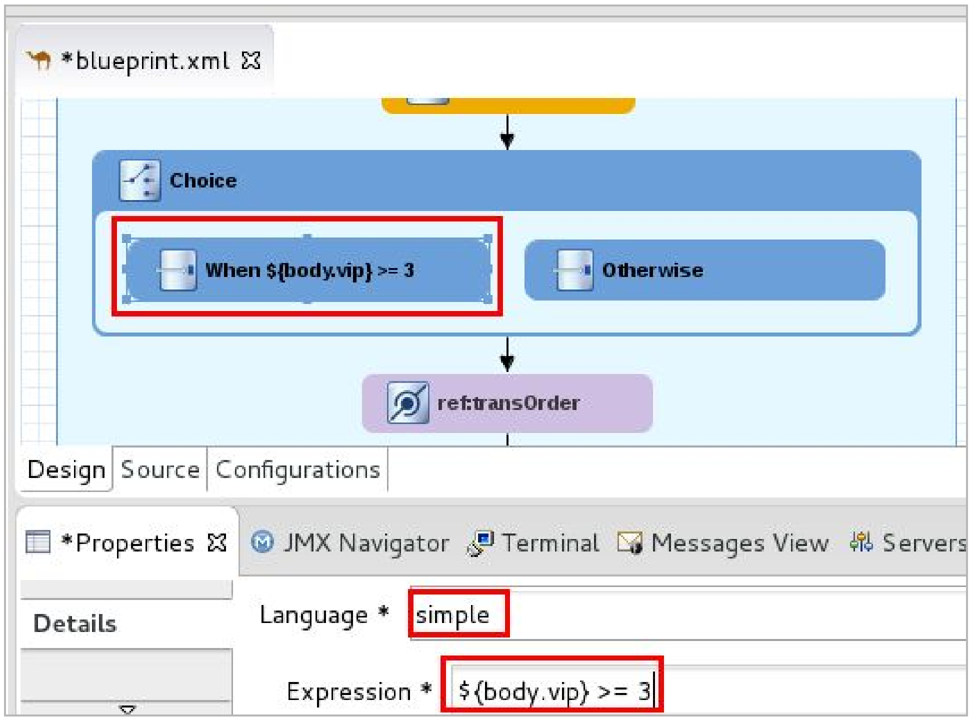


4. Drag and Drop a **When** and an **Otherwise** component inside the **Choice** component.



You could add as many ***When*** *components as needed, one for each condition.*

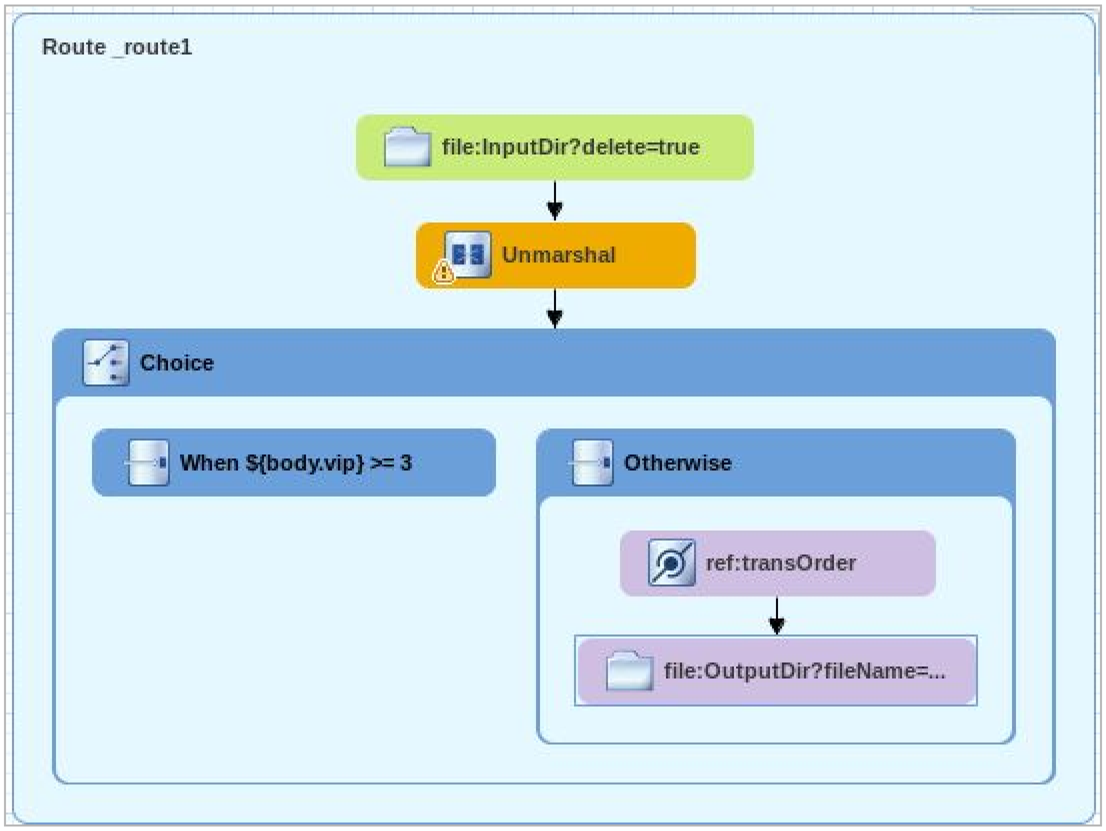
5. Click on the **When** component and click on the **Properties** tab.
   a. **Language:** simple
   b. **Expression**: ${body.vip} >= 3
      i. **Note:** there's a space before and after the ">=" operator.



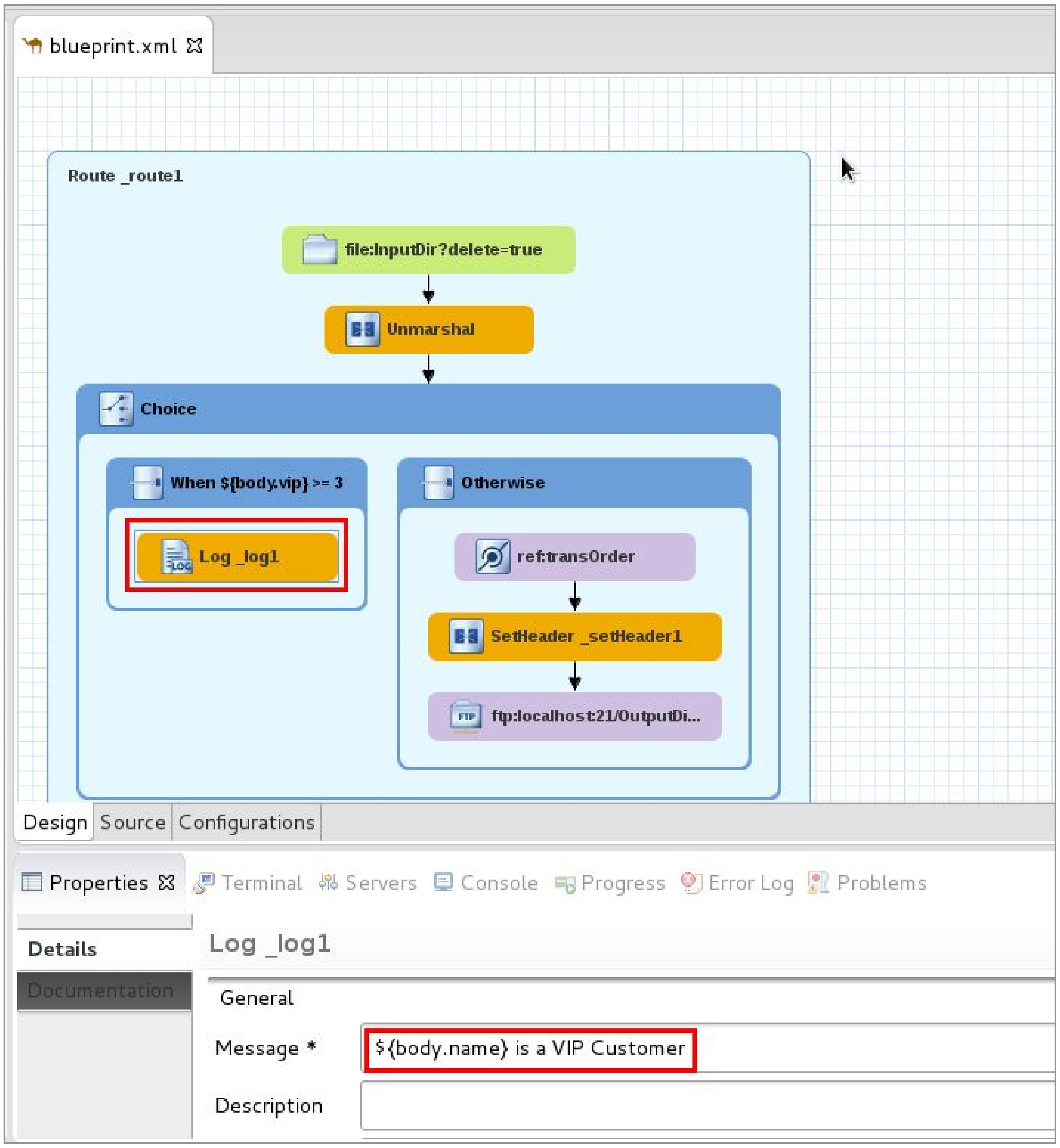
Language * simple

Expression * ${body.vip} >= 3

6. Drag the following existing components and drop them into the **Otherwise** box:
    a. **ref:TransOrder** (DataTransformation)
    b. **File**



7. Drag a **Log** component from the Components palette and drop it onto the **When** component.
8. Edit the new **Log** Component:
    a. **Message**: ${body.name} is a VIP Customer



Log _log1

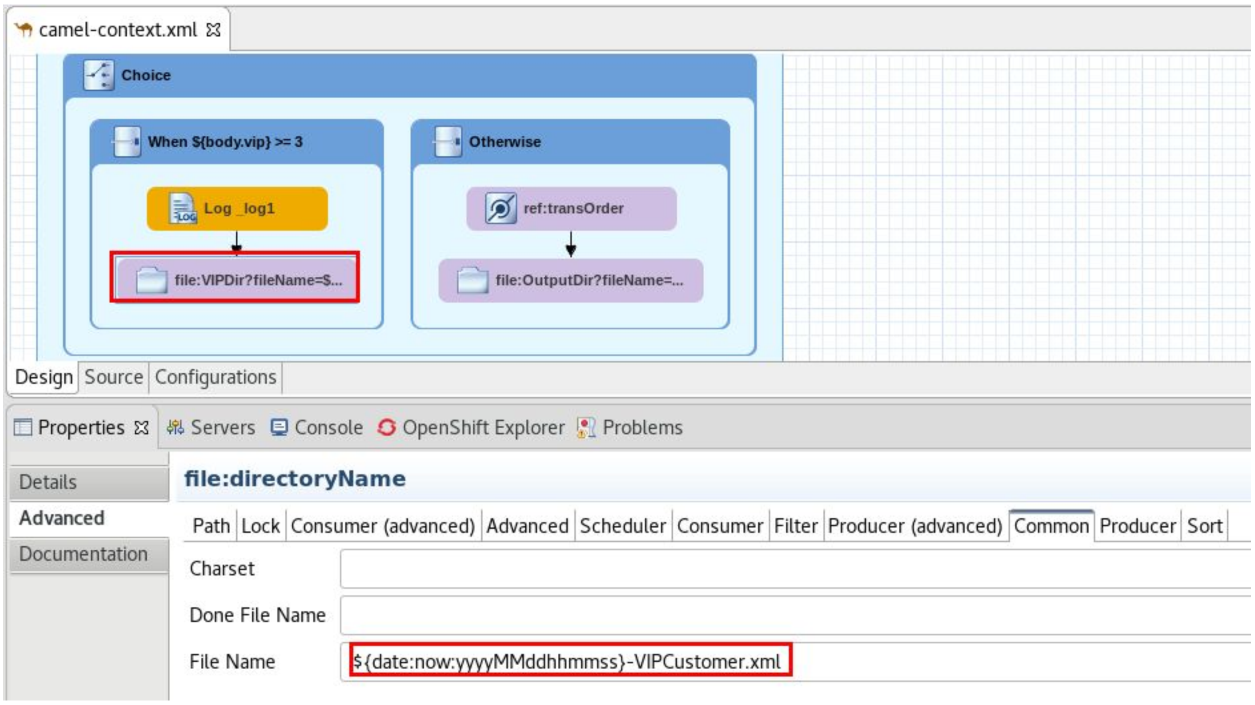
General

Message * ${body.name} is a VIP Customer

Description

9. Drag a new **File** component from the Components palette, and drop it onto the **Log** component:
    a. **Advanced -> Path -> Directory Name:** VIPDir
    b. **Advanced -> Common -> File Name**: ${date:now:yyyyMMddhhmmss}-VIPCustomer.xml

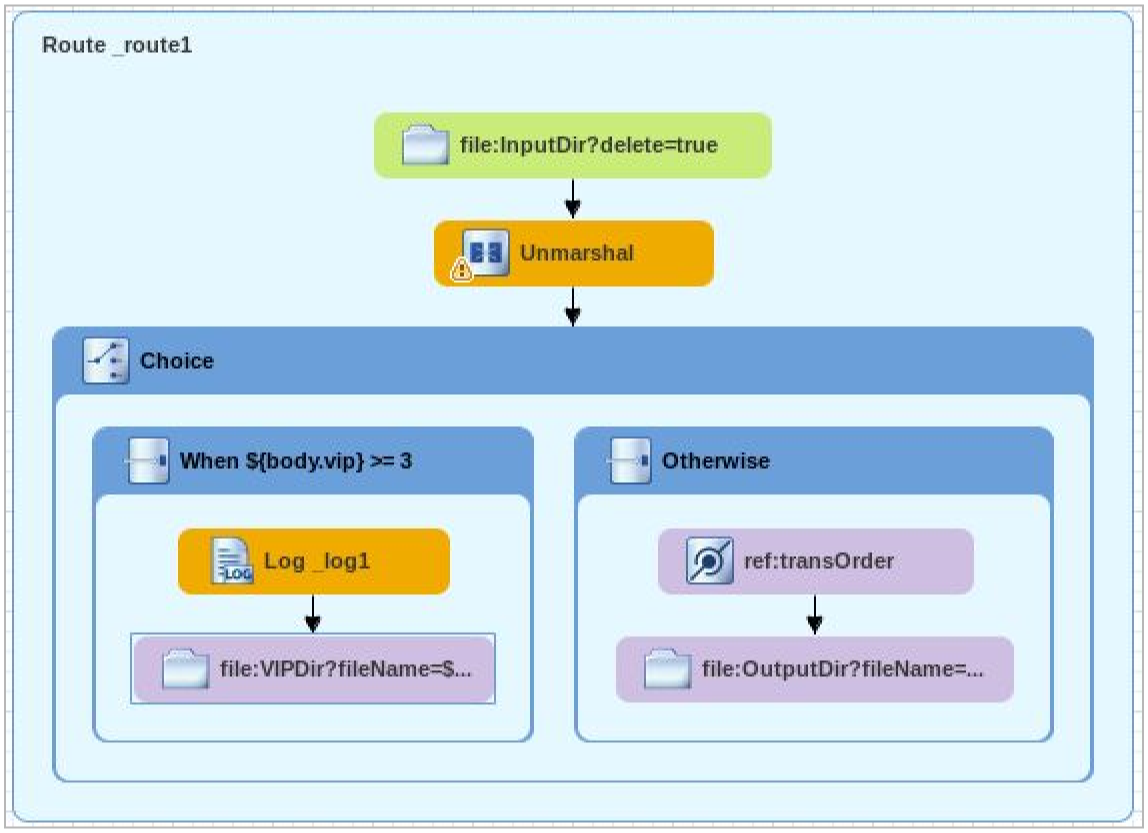
10. Save the file (Ctrl+S).



Charset

Done File Name

File Name: ${date:now:yyyyMMddhhmmss}-VIPCustomer.xml

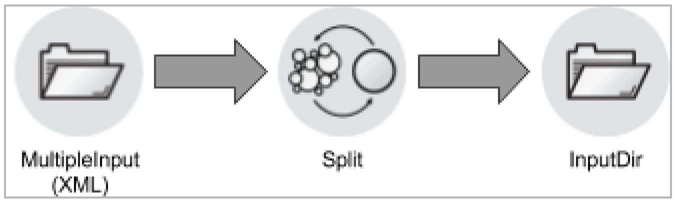


11. Run the route by right-clicking **camel-context.xml -> Run As -> Local Camel Context (without tests)**
12. Right-Click on the **InputDir** folder.
13. Select **General -> File System.**
14. Import **File2.xml** from **~/FuseWorkshop/support/Lab3**.

15. Verify that this time, the file is routed through the VIP branch, so there is a log message and the file ends up in the “***VIPdir***” folder.
16. Close all the open files.
17. Stop the camel route by clicking the stop button 🟥.

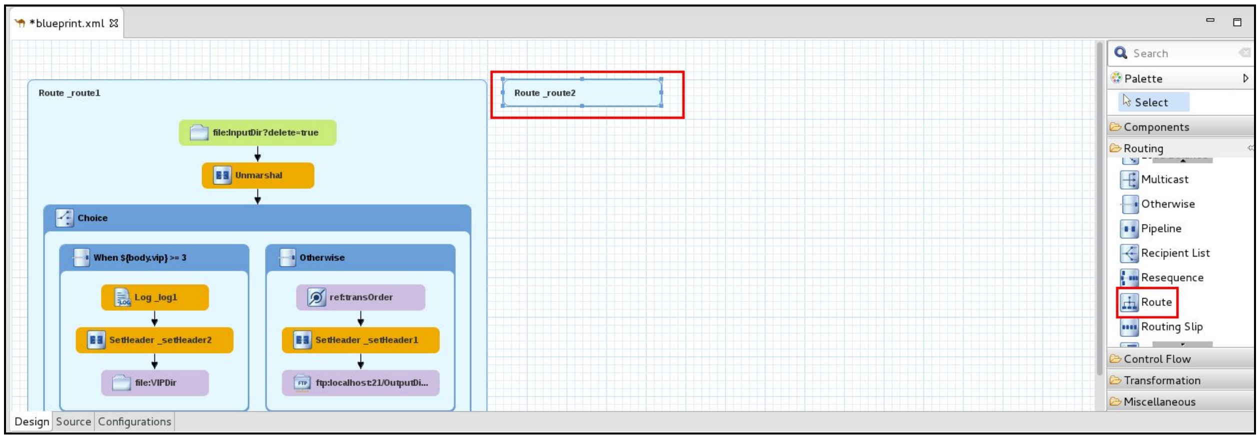
Part B - Message Splitting

In this lab you will add a Split component, to split a multiple-instance message into separate messages and process them individually.

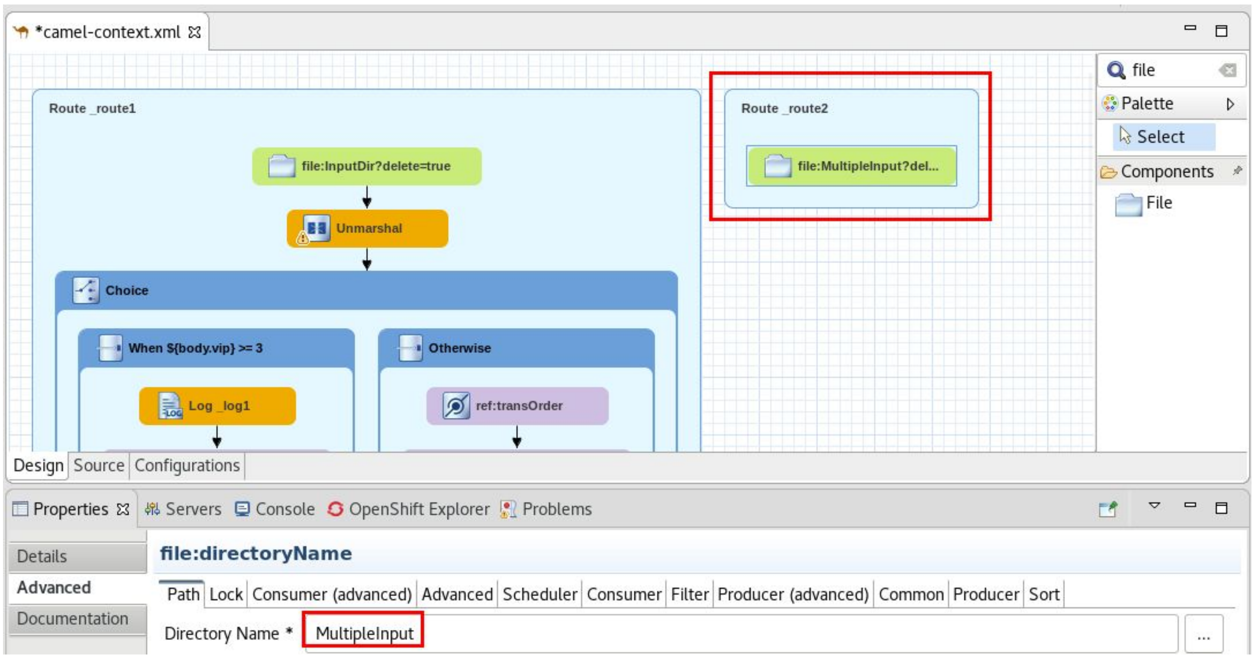


This component implements the Splitter - Enterprise Integration Pattern (https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-development-guide/chapter-8-message-routing#MsgRout-Splitter)

1. Open the **camel-context.xml** file under **Camel Context**.
2. Drag and Drop a **Route** component from the **Routing** palette into the Canvas.



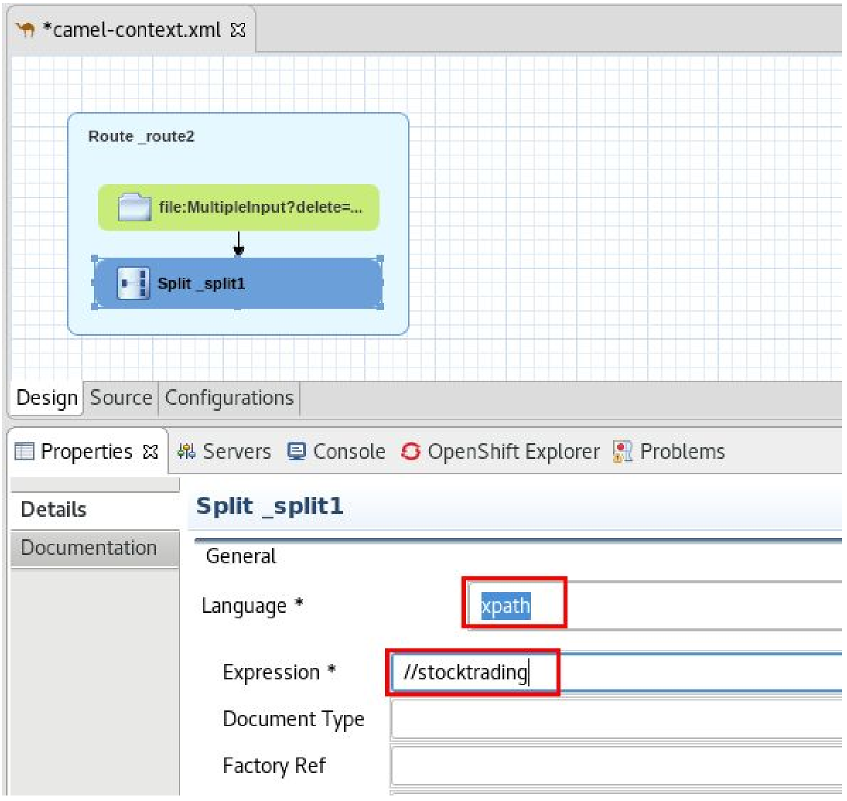
3. Drag a **File** component from the Components palette and drop it onto the new empty route.
4. Click on the Properties tab and enter the following values:
    a. **Advanced->Path -> Directory Name**: MultipleInput
    b. **Consumer->Delete**: True



file:directoryName

Directory Name * MultipleInput

5. Drag a **Split** component from the **Routing** palette and drop it onto the **file** component.
6. Edit **Split** properties:
    a. **Language**: xpath
    b. **Expression:** //stocktrading



Split _split1

Language * xpath

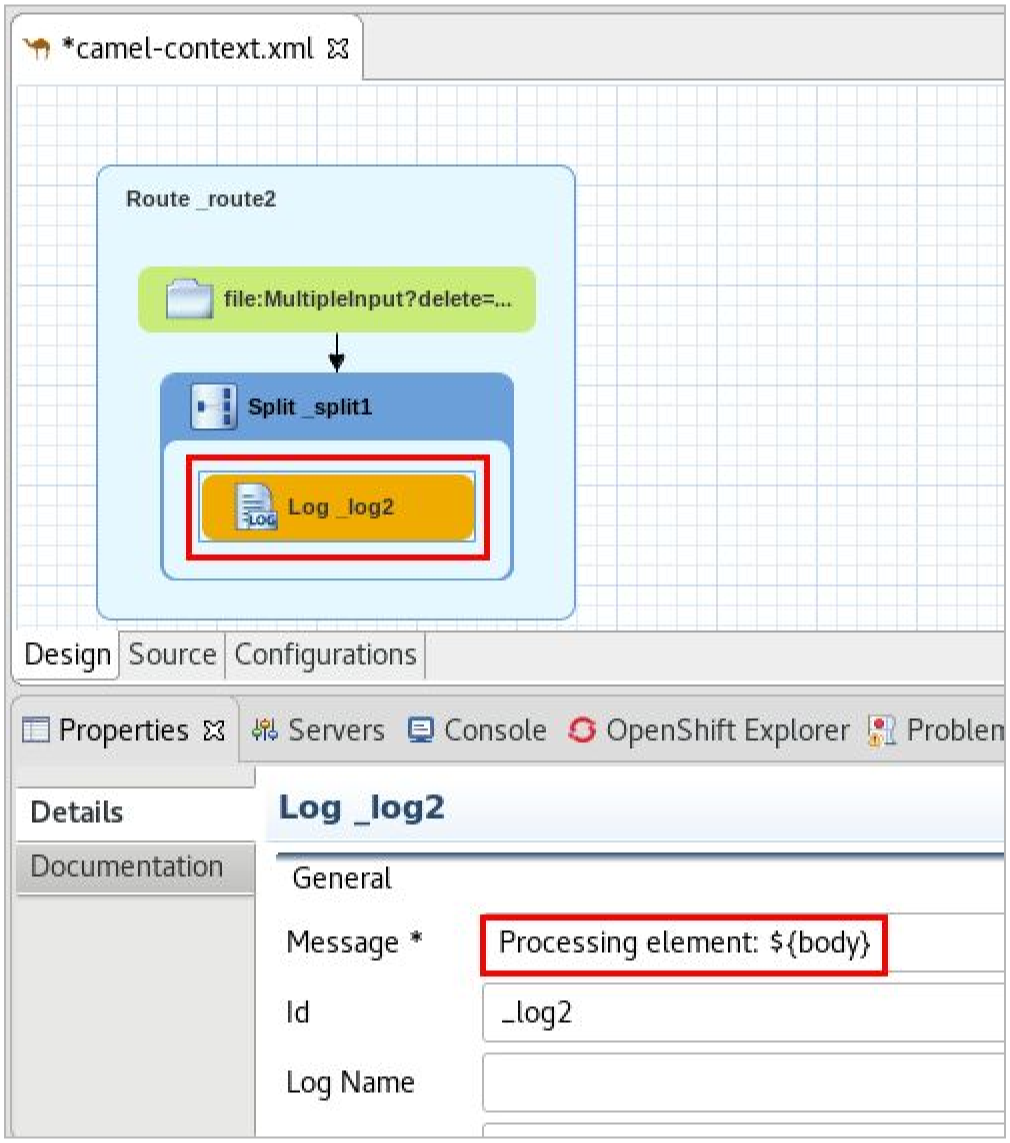
Expression * //stocktrading

Document Type

Factory Ref

7. Drag a **Log** component from the Components palette, and drop it onto the Split component.

a. **Message**: Processing element: ${body}

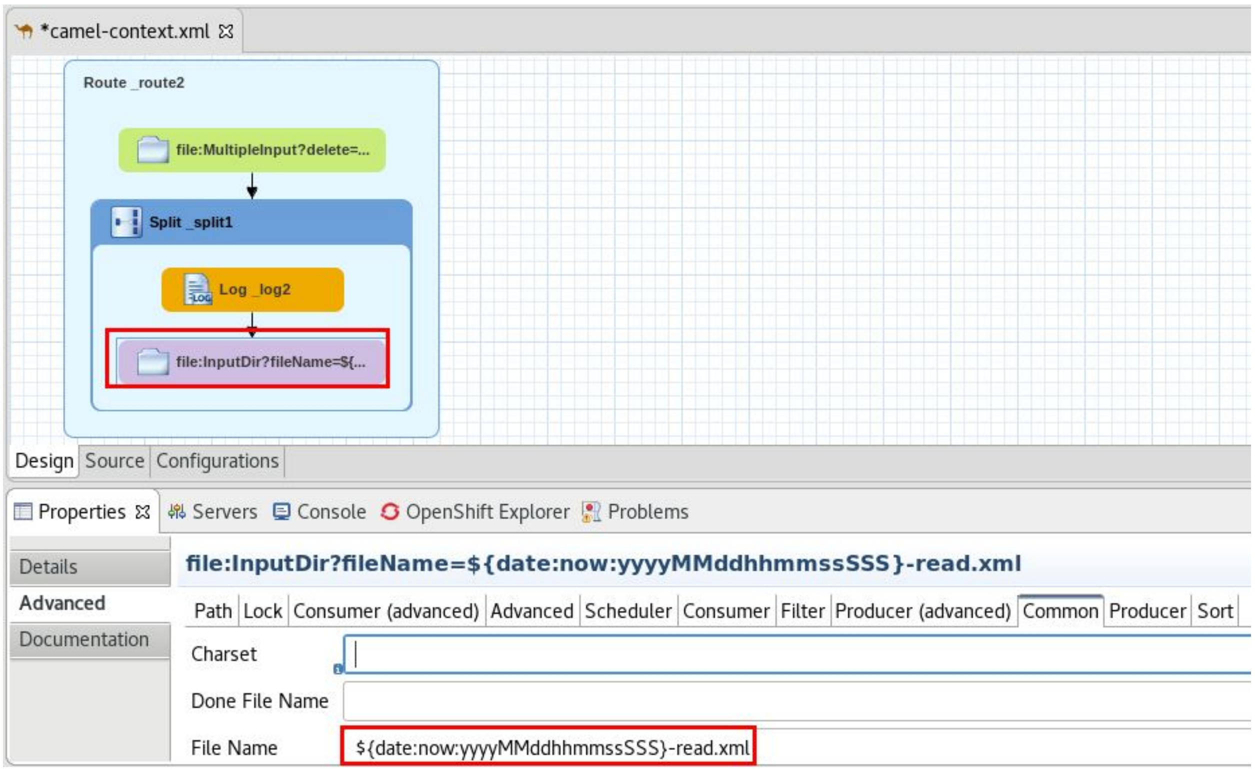


Message * Processing element: ${body}

Id _log2

Log Name

8. Drag a **File** endpoint and drop it onto the **Log** component.
   a. **Advanced -> Path -> Directory Name**:InputDir
   b. Advanced -> Common -> File Name:
   ${date:now:yyyyMMddhhmmssSSS}-read.xml
9. Save the file (Ctrl+S).



file:InputDir?fileName=${date:now:yyyyMMddhhmmssSSS}-read.xml

Charset

Done File Name

File Name: ${date:now:yyyyMMddhhmmssSSS}-read.xml

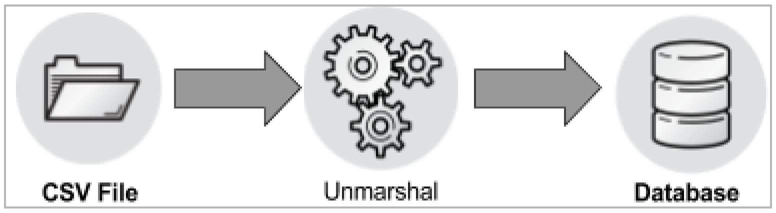
10. Inspect the contents of **~/FuseWorkshop/support/Lab3/File3.xml.**

For each element in the XML document, the splitter will generate a new Message, which will be processed by the subsequent nodes in the route. If we set the "parallelProcessing" option to true in the splitter component, all the splitted messages would be processed concurrently.

11. Run the routes by right-clicking **camel-context.xml -> Run As -> Local Camel Context (without tests)**
12. Right-click on the automatically created "***MultipleInput***" folder in the **Project Explorer**.
13. Select **Import** -> **File System.**
14. Browse to **~/FuseWorkshop/support/Lab3/** ,select **File3.xml** and click **Finish.**
15. This should fire route #2, which will generate a file for each element in the incoming XML file.
16. Then, each generated file (in the InputDir folder) should trigger an instance of route#1 to process it.
17. Close all open files.
18. Stop the camel routes by clicking the stop button. 🟥

Lab 4 - Database

In this lab you will process a CSV file and insert a record in a Database for each line. To do this, you will use the a Bindy component to parse the CSV file, and a JPA component to insert rows in a PostgreSQL DB running on OpenShift.



JPA: Enables you to store and retrieve Java objects from persistent storage using EJB 3's Java Persistence Architecture (JPA), which is a standard interface layer that wraps Object/Relational Mapping (ORM) products such as OpenJPA, Hibernate, TopLink, and so on.
https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-component-reference/chapter-86-jpa

Bindy: Enables the parsing and binding of non-structured data via Java Beans. These Java Beans consists of binding mappings defined with annotations.
https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-component-reference/chapter-14-bindy

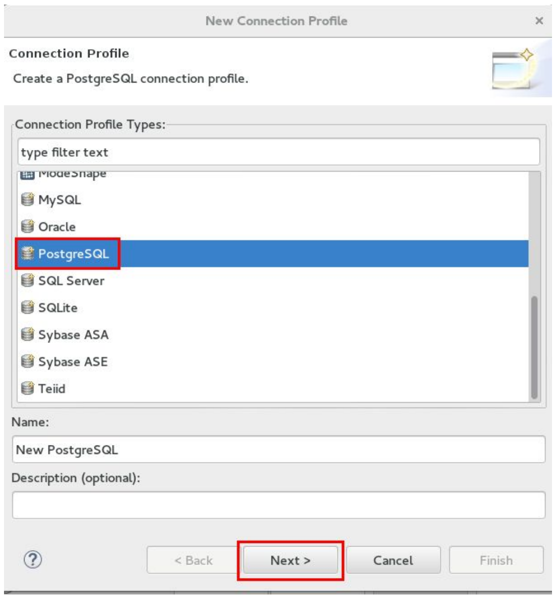
The database has been prepared for you with the following details:

Database: sampledb
URL: jdbc:postgresql://13.229.117.241:31291/sampledb
Username: fis
Password: training

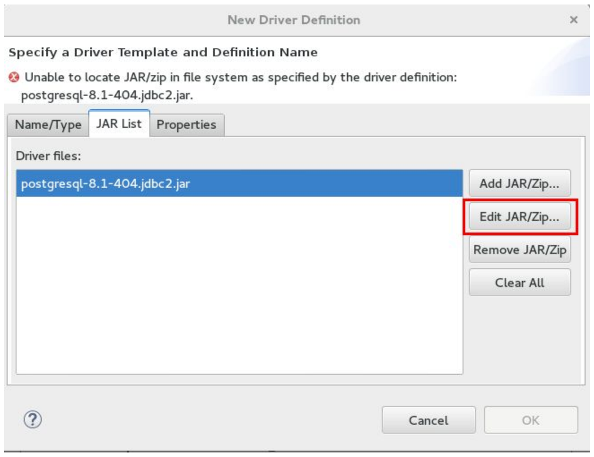
Labs

1. Click **File -> Import.**
2. Select **General -> Existing Project into Workspace.**
3. Select **archive file**.
4. Browse to **~/FuseWorkshop/support/Lab4/** and select **Lab4.zip**.

5. Disregard the errors.
6. Right-click on the **Lab4** project, and select **New-> JPA Entities from tables.**
7. Click on the “add connections” button.
8. Select **PostgreSQL** and click **Next**.



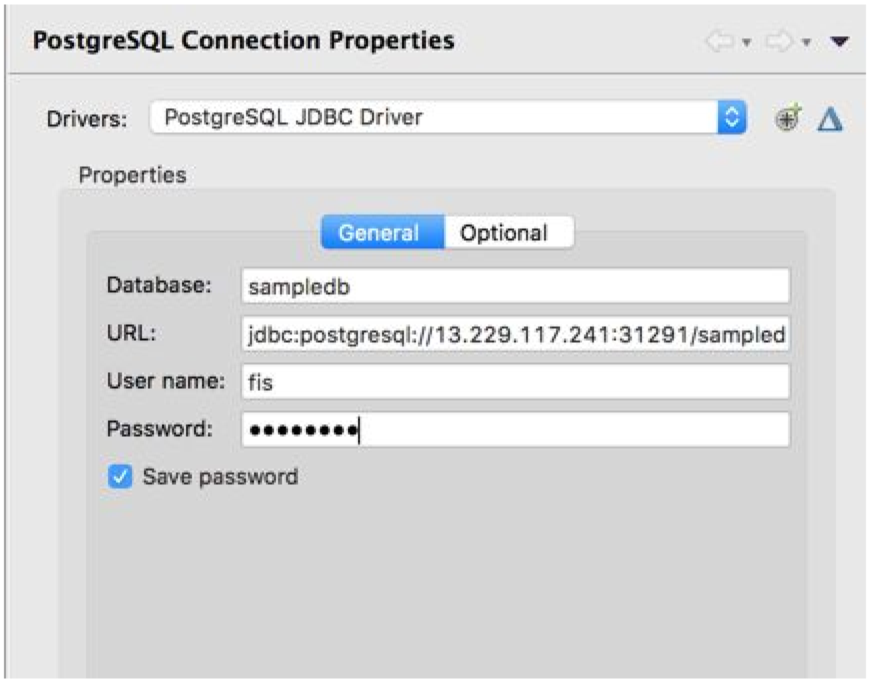
9. Click on the “New driver definition” button .
10. Select **PostgreSQL JDBC Driver** and click on the **JAR List** tab.
11. Select the driver file and click on the **Edit JAR/Zip** button.


New Driver Definition

Specify a Driver Template and Definition Name

Unable to locate JAR/zip in file system as specified by the driver definition: postgresql-8.1-404.jdbc2.jar.

12. Browse to **~/FuseWorkshop/support/Lab4** and select **postgresql-9.4-1201.jdbc4.jar**
13. Click on the **Properties** tab and complete the following:
    a. **Connection URL**: jdbc:postgresql://13.229.117.241:31291/sampledb
    b. **Database Name**: sampledb
    c. **User Name**: fis
    d. **Password**: training


PostgreSQL Connection Properties

Drivers: PostgreSQL JDBC Driver

Properties

General | Optional

Database: sampledb

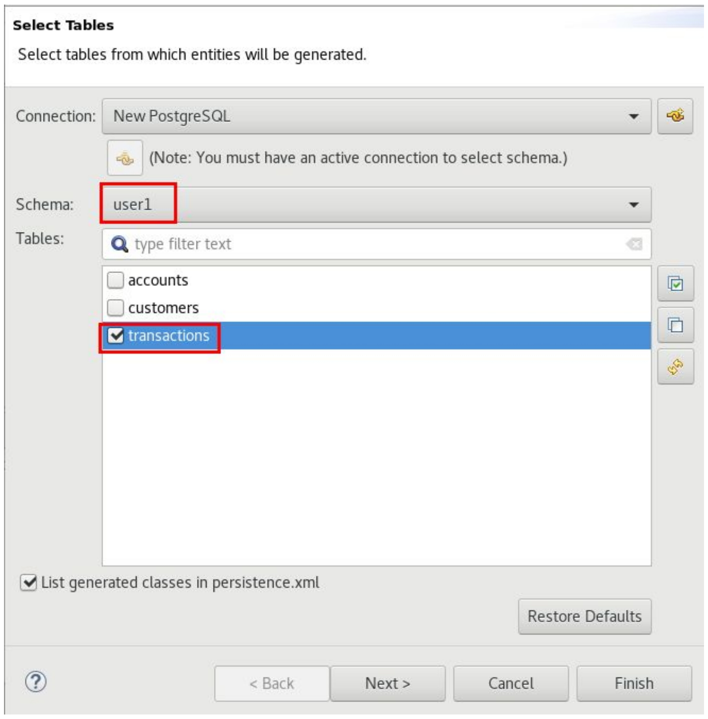
URL: jdbc:postgresql://13.229.117.241:31291/sampled

User name: fis

Password: ••••••••

✓ Save password

14. Click **OK**.
15. Click on the **Test Connection** button to validate the connection.
16. Click **Finish**.
17. In the **Select Tables** dialog, select the **fis** schema and check the **transactions** table.


Select Tables

Select tables from which entities will be generated.

Connection: New PostgreSQL

(Note: You must have an active connection to select schema.)

Schema: user1

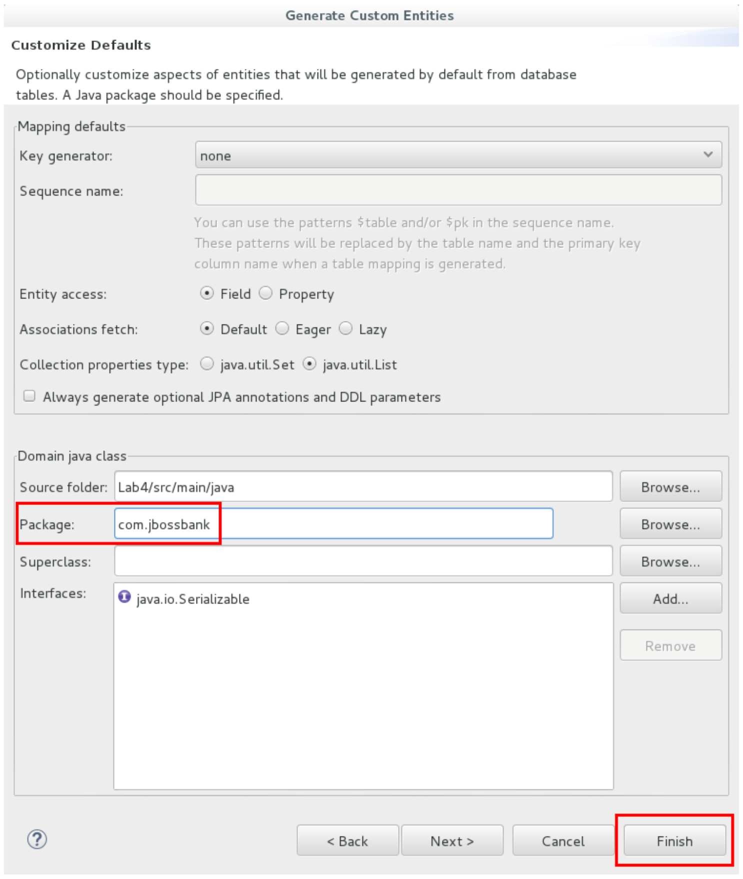
Tables:

- [ ] accounts
- [ ] customers
- [x] transactions

✓ List generated classes in persistence.xml

18. Click **Next**.
19. Click **Next.**
20. In the **Customize Defaults dialog**, in the **Domain java class** area, enter:
    a. **package:** com.jbossbank
    b. Source folder: src/main/java

21. Click **Finish.**


Generate Custom Entities

Customize Defaults

Optionally customize aspects of entities that will be generated by default from database tables. A Java package should be specified.

Mapping defaults

Key generator: none

Sequence name:

You can use the patterns $table and/or $pk in the sequence name. These patterns will be replaced by the table name and the primary key column name when a table mapping is generated.

Entity access: [✓ Field] Property

Associations fetch: [✓ Default] Eager Lazy

Collection properties type: java.util.Set [✓ java.util.List]

☐ Always generate optional JPA annotations and DDL parameters

Domain java class

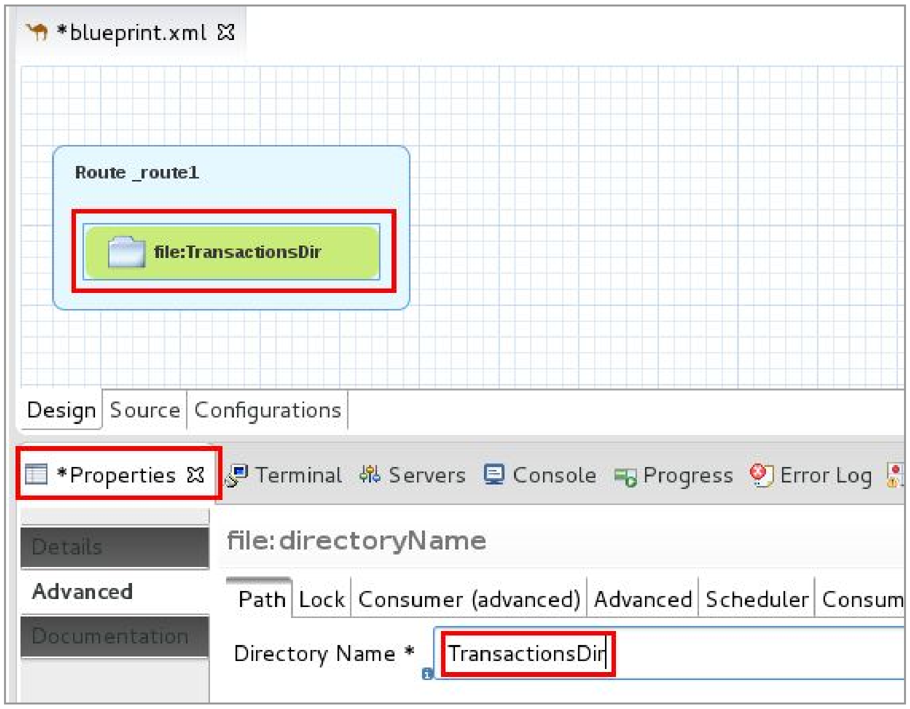
Source folder: Lab4/src/main/java

Package: com.jbossbank

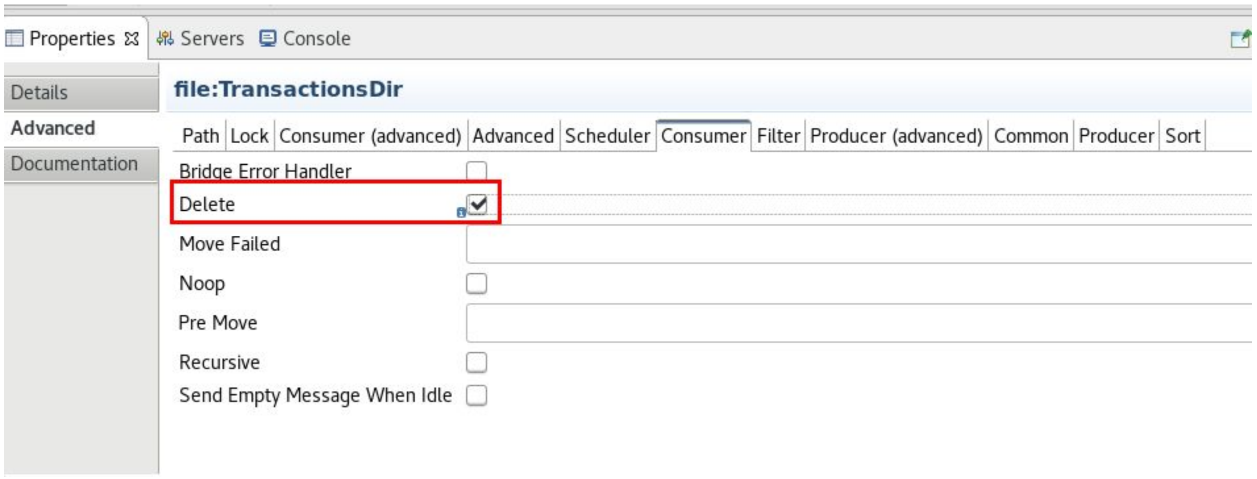
Superclass:

Interfaces: java.io.Serializable

22. A new java class that represents the Transactions table has been created under **src/main/java/com.jbossbank**.
23. Open **camel-context.xml** by double-clicking it.
24. Drag a **File** component from the Components palette and drop it into the Route.
    a. Edit its properties
        i. **Advanced -> Path -> Directory Name:** TransactionsDir
        ii. **Advanced -> Consumer -> Delete: true**

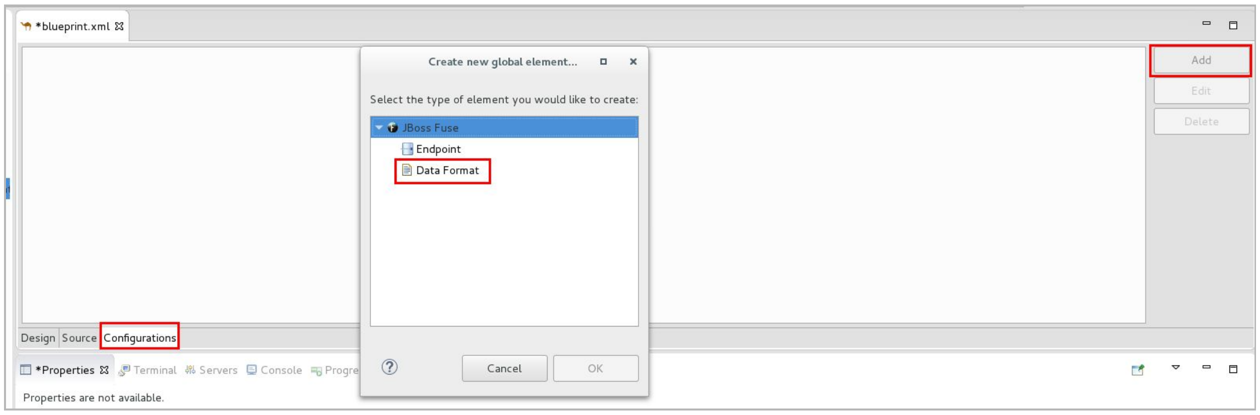


Directory Name * TransactionsDir



Bridge Error Handler ☐
Delete ✓
Move Failed
Noop ☐
Pre Move
Recursive ☐
Send Empty Message When Idle ☐

25. Click on the **Configurations** tab.
26. Click on the **Add** button.
27. Select **Data Format** and click **OK**.

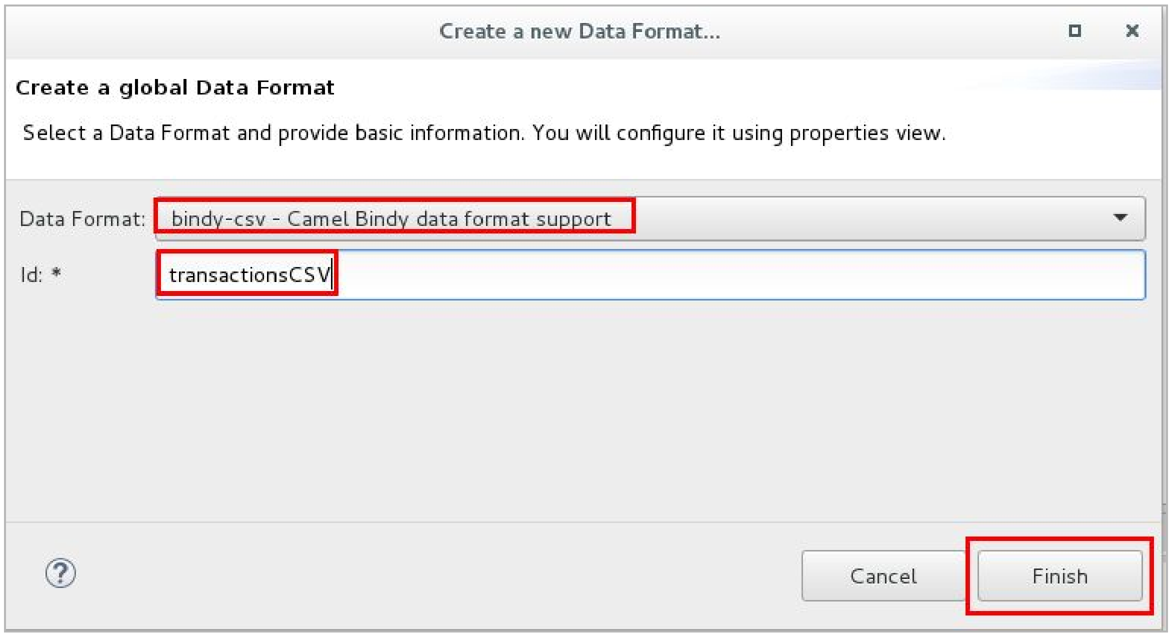


Create new global element...

Select the type of element you would like to create:

Properties are not available.

28. Select "**bindy-csv**" as the **Data Format**.
29. Enter "transactionsCSV" as **Id**.
30. Click on **Finish**.



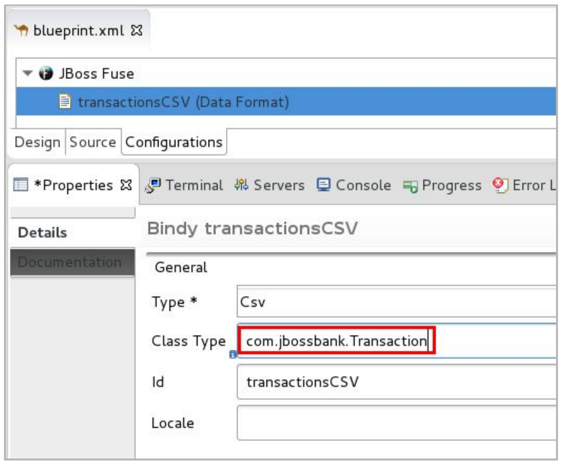
Create a global Data Format

Select a Data Format and provide basic information. You will configure it using properties view.

Data Format: bindy-csv - Camel Bindy data format support

Id: * transactionsCSV

31. In the properties tab, enter "com.jbossbank.Transaction" as **Class Type**.



Bindy transactionsCSV

General

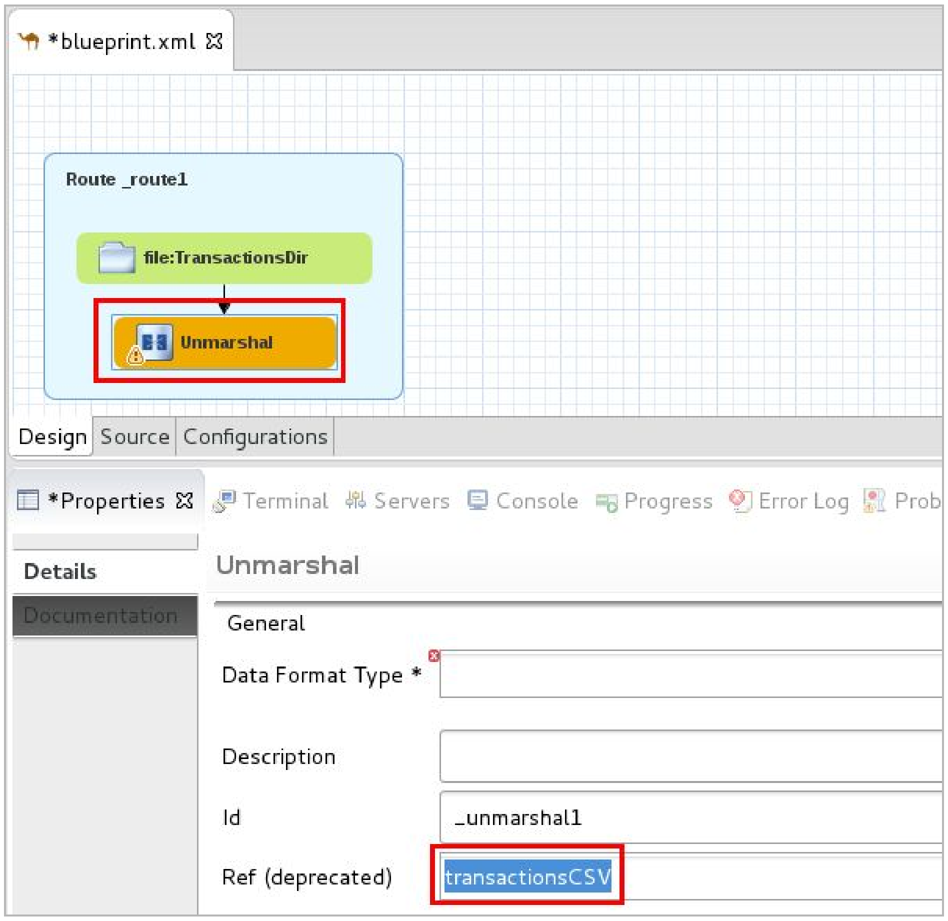
Type * Csv

Class Type com.jbossbank.Transaction

Id transactionsCSV

Locale

32. Click on the **Design** tab.
33. Drag an **Unmarshal** component from the Transformation palette and drop it onto the **File** component.
    a. Click on the **Ref** dropdown and select **transactionsCSV**.



Unmarshal

General

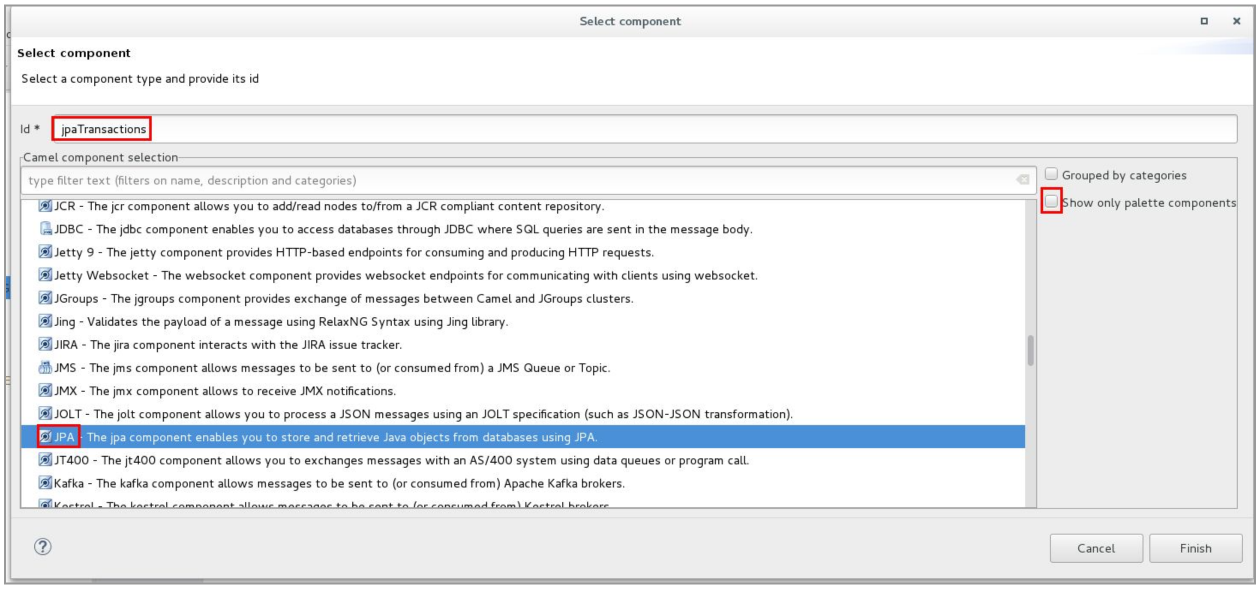
Data Format Type *

Description

Id _unmarshal1

Ref (deprecated) transactionsCSV

34. Drag a **Generic** component from the Components palette and drop it onto the Unmarshal component.
35. Enter “jpaTransactions” as the **Id**.
36. Uncheck “**Show only palette components”.**
37. Select **JPA**.
38. Click **Finish**.



Select component

Select a component type and provide its id

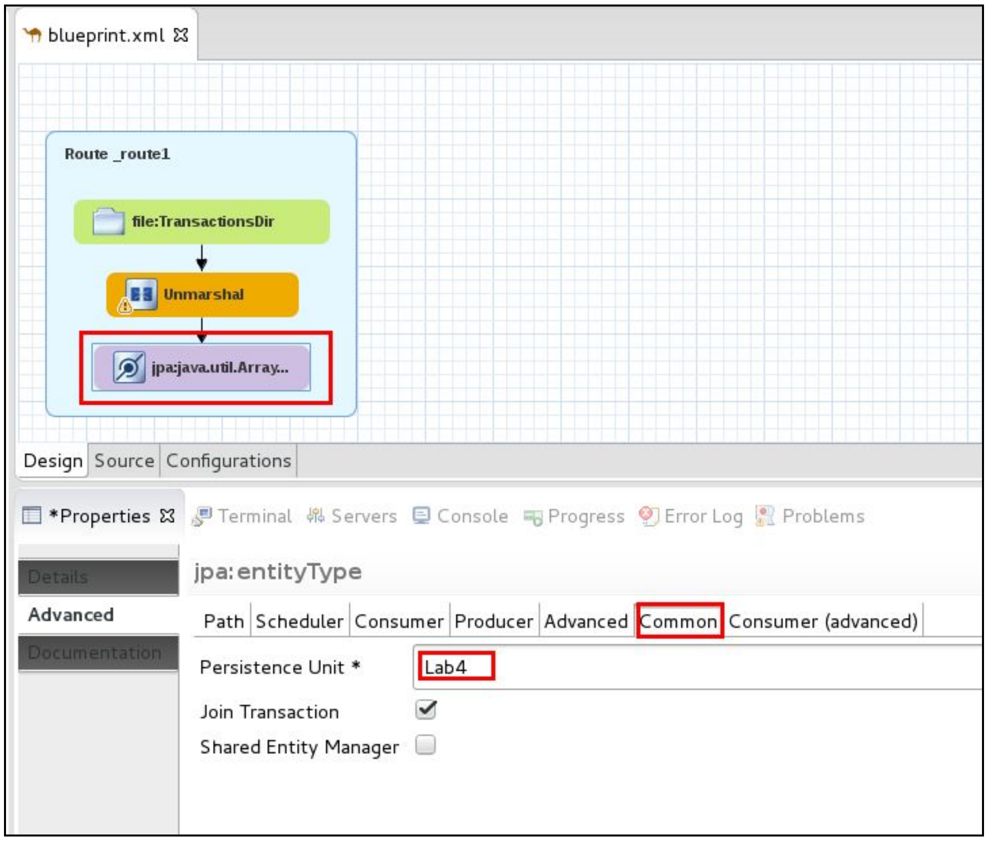
Id * jpaTransactions

Camel component selection

type filter text (filters on name, description and categories)

JCR - The jcr component allows you to add/read nodes to/from a JCR compliant content repository.
JDBC - The jdbc component enables you to access databases through JDBC where SQL queries are sent in the message body.
Jetty 9 - The jetty component provides HTTP-based endpoints for consuming and producing HTTP requests.
Jetty Websocket - The websocket component provides websocket endpoints for communicating with clients using websocket.
JGroups - The jgroups component provides exchange of messages between Camel and JGroups clusters.
Jing - Validates the payload of a message using RelaxNG Syntax using Jing library.
JIRA - The jira component interacts with the JIRA issue tracker.
JMS - The jms component allows messages to be sent to (or consumed from) a JMS Queue or Topic.
JMX - The jmx component allows to receive JMX notifications.
JOLT - The jolt component allows you to process a JSON messages using an JOLT specification (such as JSON-JSON transformation).
JPA - The jpa component enables you to store and retrieve Java objects from databases using JPA.
JT400 - The jt400 component allows you to exchanges messages with an AS/400 system using data queues or program call.
Kafka - The kafka component allows messages to be sent to (or consumed from) Apache Kafka brokers.
Kestrel - The kestrel component allows messages to be sent to (or consumed from) Kestrel brokers.

39. Link the **Unmarshal** component to the **JPA** component.
40. Click on the **JPA** component and edit its properties:
    i. **Advanced-> Path**
        1. **Entity Type:** java.util.ArrayList
    ii. **Advanced-> Common**
        1. **Persistence Unit:** Lab4

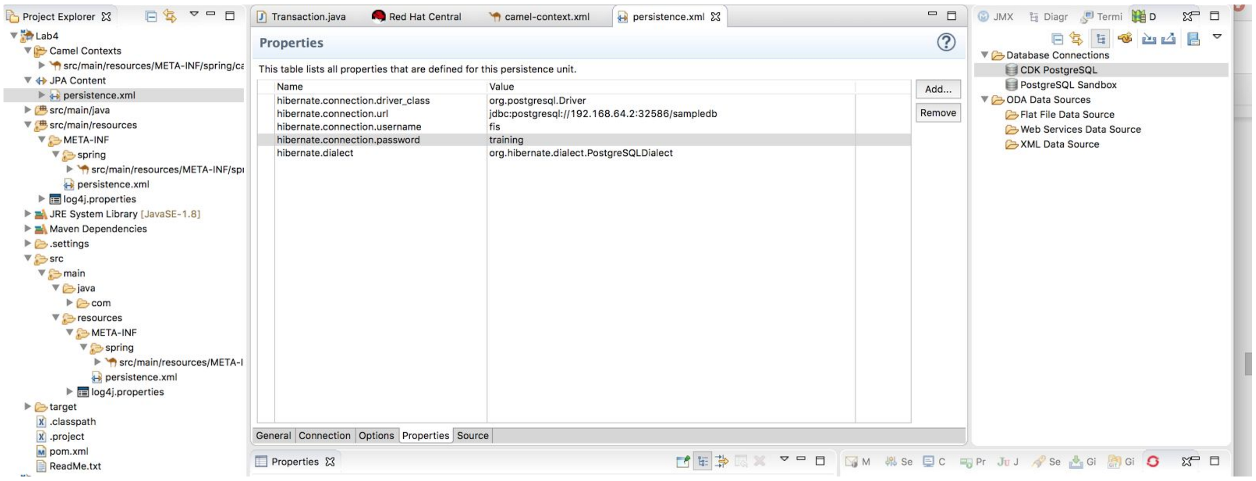


jpa:entityType

Persistence Unit * Lab4
Join Transaction ✓
Shared Entity Manager ☐

We define java.util.ArrayList as the Entity Type because we will be sending a list of transactions (read and unmarshalled from the csv file) to be inserted in the table.

41. Save the blueprint file (Ctrl+S).
42. Right-click on **src/main/resources/META-INF** folder and select **Import -> General -> File System**
43. Browse to **~/FuseWorkshop/support/Lab4** and select **persistence.xml**.
44. Open **persistence.xml** file under **JPA Content**, and review its contents.

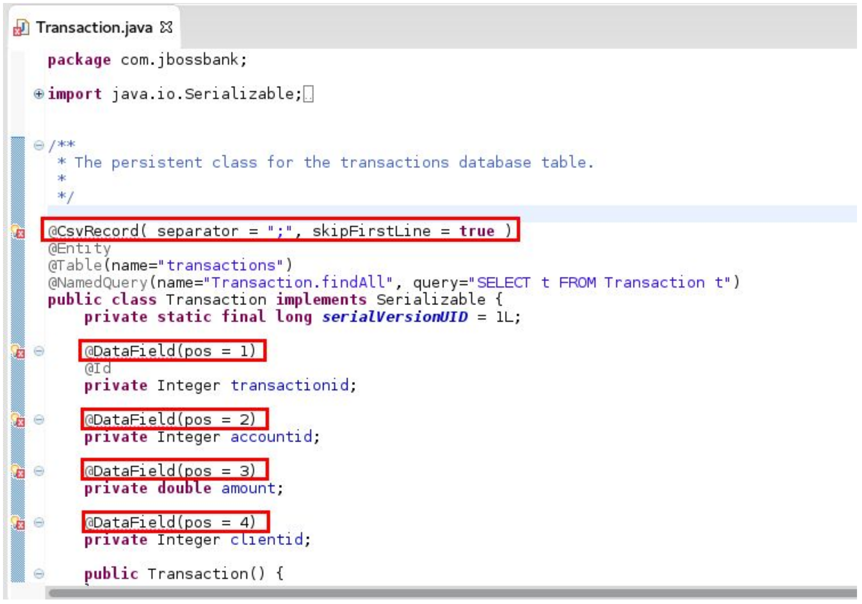


Properties

This table lists all properties that are defined for this persistence unit.

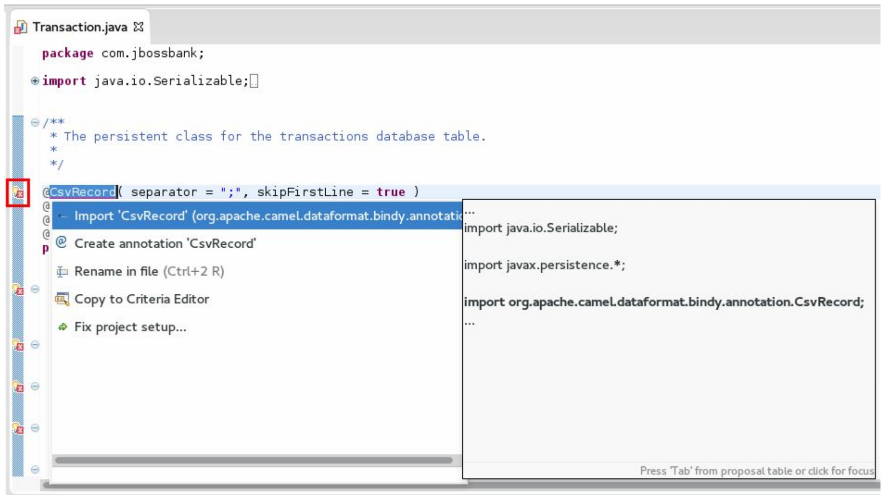
Name	Value
hibernate.connection.driver_class	org.postgresql.Driver
hibernate.connection.url	jdbc:postgresql://192.168.64.2:32586/sampledb
hibernate.connection.username	fis
hibernate.connection.password	training
hibernate.dialect	org.hibernate.dialect.PostgreSQLDialect

45. Open **Transaction.java** under src/main/java/com.jbossbank.
46. Add the following annotation before “@Entity”: **@CsvRecord(separator = ";" , skipFirstLine = true)**
47. Add the following annotation before the **transactionid** property: **@DataField(pos = 1)**
48. Add the following annotation before the **accountid** property: **@DataField(pos = 2)**
49. Add the following annotation before the **amount** property: **@DataField(pos = 3)**
50. Add the following annotation before the **clientid** property: **@DataField(pos = 4)**



```
package com.jbossbank;

import java.io.Serializable;

/**
 * The persistent class for the transactions database table.
 *
 */
@CsvRecord( separator = ";", skipFirstLine = true )
@Entity
@Table(name="transactions")
@NamedQuery(name="Transaction.findAll", query="SELECT t FROM Transaction t")
public class Transaction implements Serializable {
    private static final long serialVersionUID = 1L;

    @DataField(pos = 1)
    @Id
    private Integer transactionid;

    @DataField(pos = 2)
    private Integer accountid;

    @DataField(pos = 3)
    private double amount;

    @DataField(pos = 4)
    private Integer clientid;

    public Transaction() {
```

51. Click on the error marker next to **@CsvRecord** annotation and select the first suggestion (Import CsvRecord)

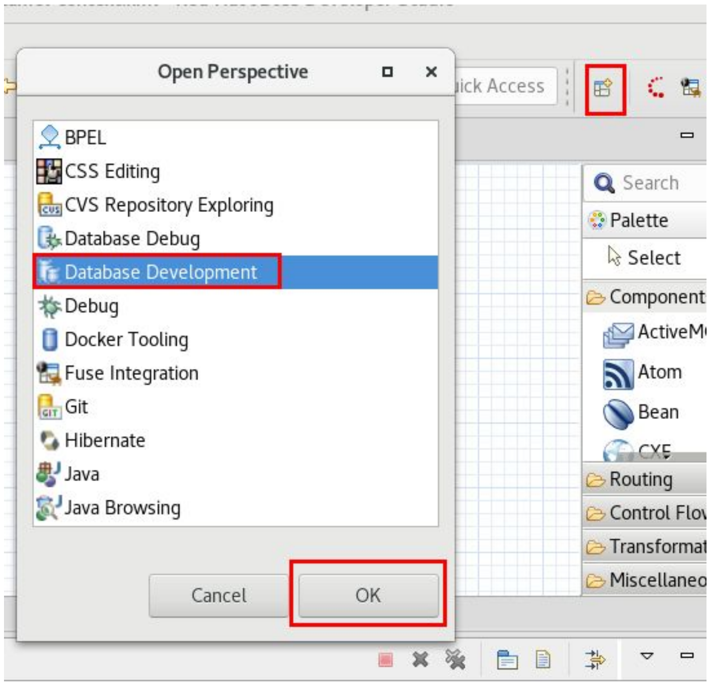


```
package com.jbossbank;

import java.io.Serializable;

/**
 * The persistent class for the transactions database table.
 *
 */
@CsvRecord( separator = ";", skipFirstLine = true )
```

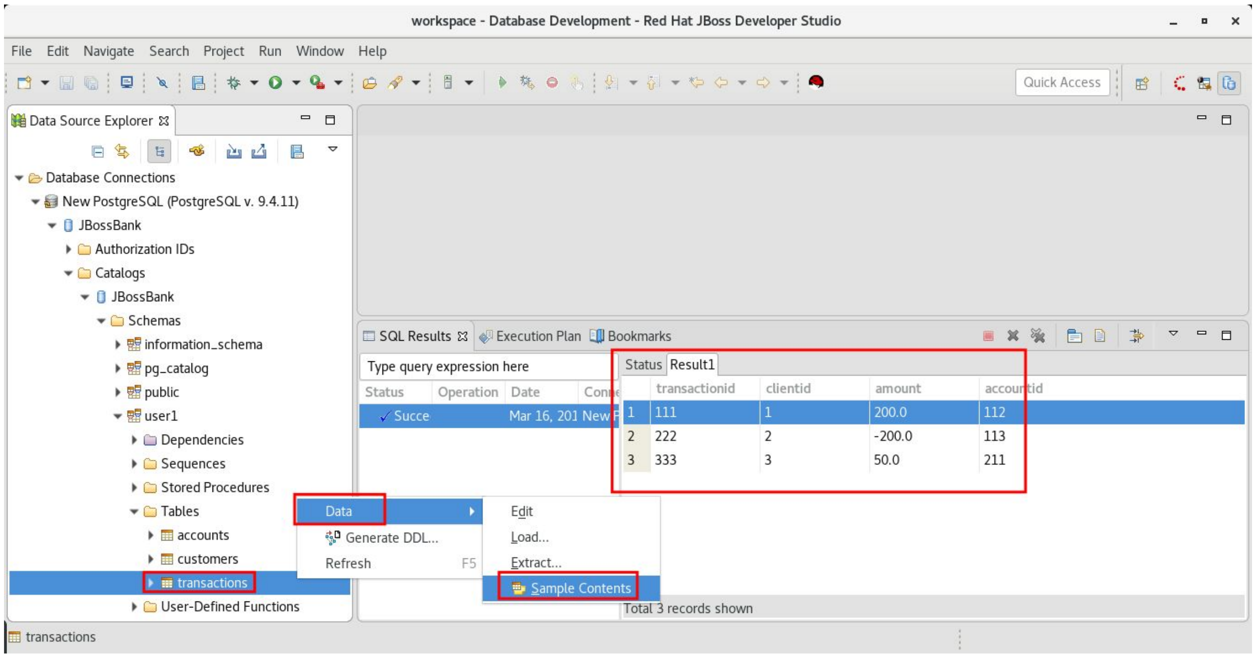
Transaction.java
Import 'CsvRecord' (org.apache.camel.dataformat.bindy.annotati
Create annotation 'CsvRecord'
Rename in file (Ctrl+2 R)
Copy to Criteria Editor
Fix project setup...
Press 'Tab' from proposal table or click for focus

```
...
import java.io.Serializable;

import javax.persistence.*;

import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
...
```

52. Click on the error marker next to any **@DataField** annotation and select the first suggestion (Import DataField).
53. Save the file and close it (Ctrl+S).
54. Run the Camel Route by right-clicking **camel-context.xml**, **Run As -> Local Camel Context (without tests).**
55. Once the route is running, right-click on the **TransactionsDir** folder.
    a. If it doesn't appear, just refresh Lab4 project (F5).
56. Click **Import**, **General -> File System**
57. Browse to **~/FuseWorkshop/support/Lab4,** and select **Transactions.csv**
58. Click **Finish**.
59. The route will process the CSV file and insert each line into the PostgreSQL database.
60. Stop the camel route by clicking on the **Stop** button 🟥.
61. Close all open files.
62. Open the **Database Development** perspective.



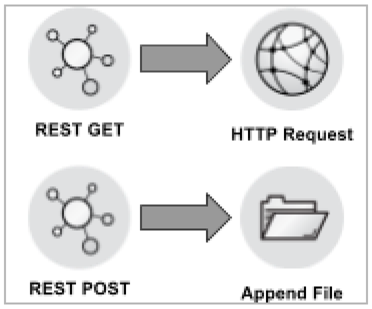
63. In the **Data Source Explorer**, expand **Database Connections -> New PostgreSQL -> database -> Catalogs -> sampledb -> Schemas -> public -> Tables**
64. Right-click on the **transactions table** and select **Data -> Sample contents.**



	transactionid	clientid	amount	accountid
1	111	1	200.0	112
2	222	2	-200.0	113
3	333	3	50.0	211

Lab 5 - REST Web Services

In this lab you will create a route that exposes a REST Web Service using the REST DSL and deploy it to OpenShift.



This web service will expose two operations: 1) A (GET) Orders request that will query a web API for an order, and 2) A (POST) operation that appends the contents of the request to a file.

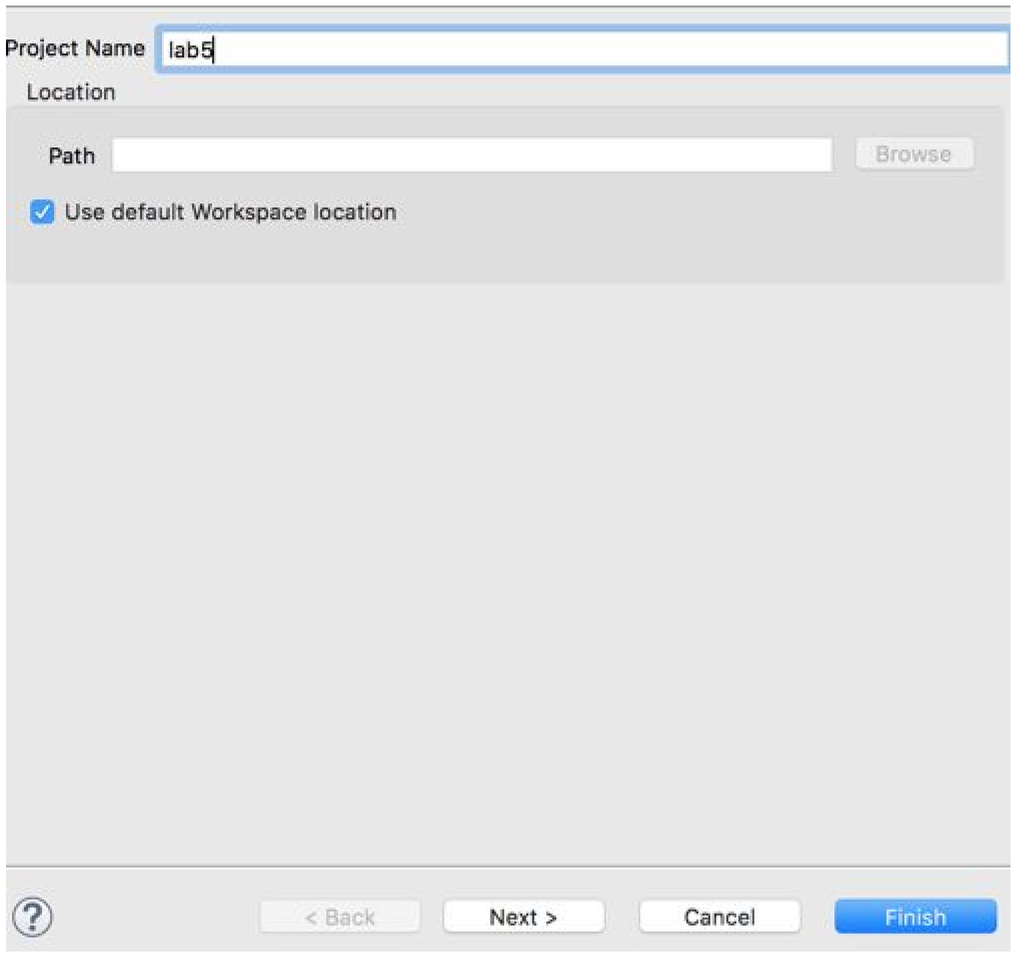
Apache Camel supports multiple approaches to defining REST services. In particular, Apache Camel provides the REST DSL (Domain Specific Language), which is a simple but powerful fluent API that can be layered over any REST component and provides integration with Swagger.

https://access.redhat.com/documentation/en/red-hat-jboss-fuse/6.3/paged/apache-camel-development-guide/chapter-4-defining-rest-services

Other alternatives when using REST are: CXF RS, Spark Rest, and Restlet

1. Go to Openshift Web Console, Select **New->Project.**
2. Enter:
    a. **Project Name:** lab5-<your username>
    b. **Display Name:** Lab5
    c. **Description:** Lab5 - REST
3. Click **Finish**.

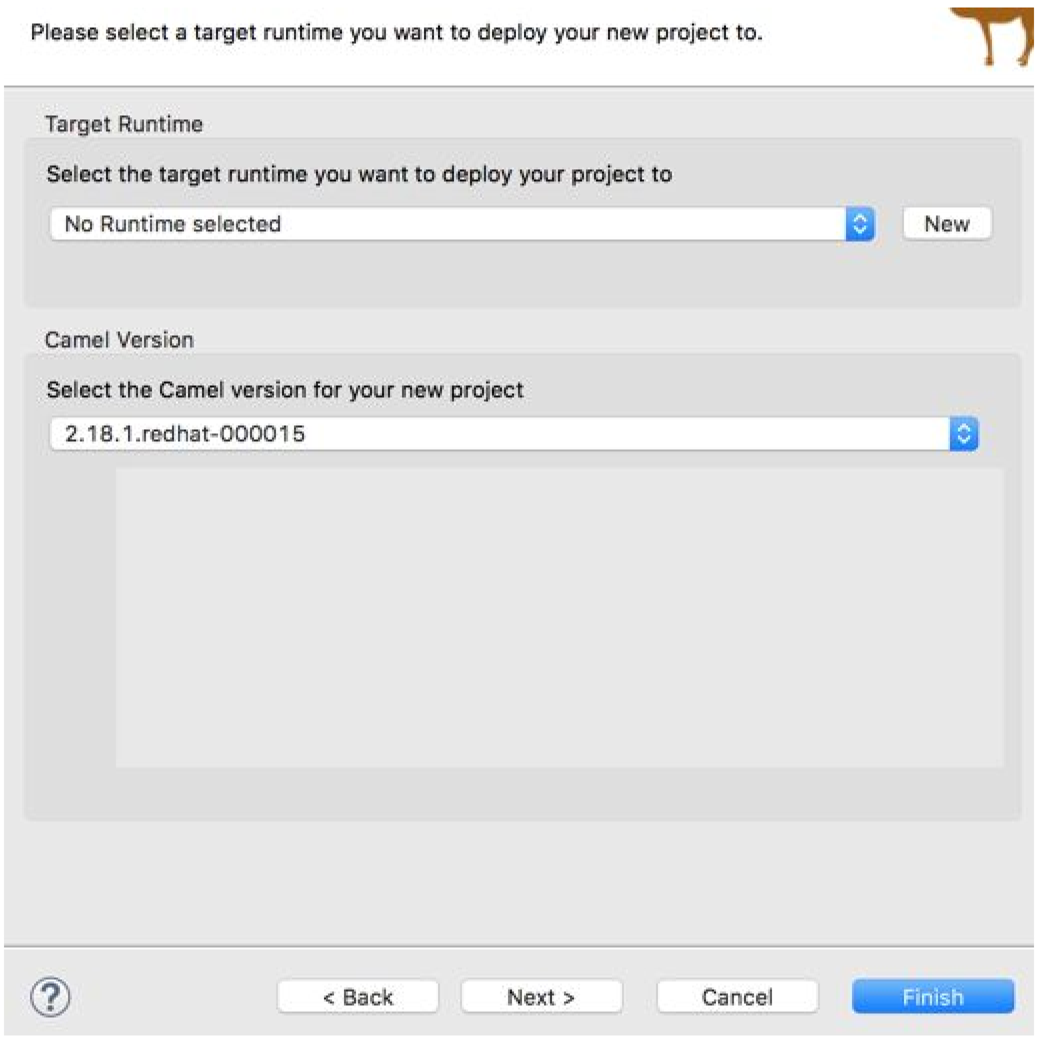
On the JBDS, Select New -> Fuse Integration Project


Project Name lab5

Location

Path

✓ Use default Workspace location


Please select a target runtime you want to deploy your new project to.

Target Runtime

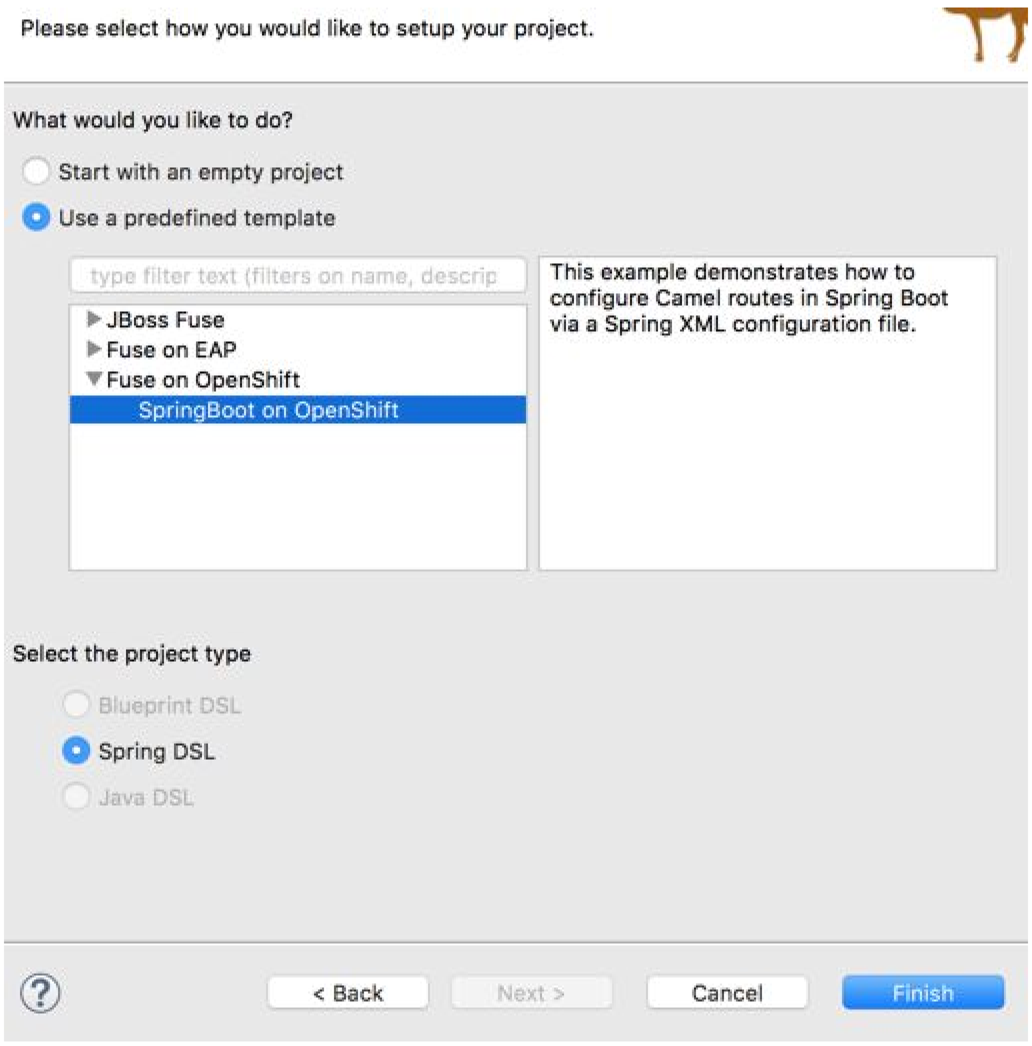
Select the target runtime you want to deploy your project to

No Runtime selected

Camel Version

Select the Camel version for your new project

2.18.1.redhat-000015


Please select how you would like to setup your project.

What would you like to do?

- ○ Start with an empty project
- ● Use a predefined template

type filter text (filters on name, descrip

- ▶ JBoss Fuse
- ▶ Fuse on EAP
- ▼ Fuse on OpenShift
  - SpringBoot on OpenShift

This example demonstrates how to configure Camel routes in Spring Boot via a Spring XML configuration file.

Select the project type

- ○ Blueprint DSL
- ● Spring DSL
- ○ Java DSL

Once the project has been created, open up the Camel context xml and replace the content with the following.

```
<bean class="org.springframework.boot.context.embedded.ServletRegistrationBean"
id="camelServlet">
       <property name="name" value="CamelServlet"/>
       <property name="loadOnStartup" value="1"/>
       <property name="servlet">
              <bean class="org.apache.camel.component.servlet.CamelHttpTransportServlet"
id="camelServlet" />
       </property>
       <property name="urlMappings" value="/*" />
    </bean>
    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

        <restConfiguration component="servlet"/>
        <rest path="/say">
            <get outType="java.lang.String" produces="application/json" uri="/hello">
                <to uri="direct:hello"/>
            </get>
        </rest>
        <route id="_route1">
            <from id="_from1" uri="direct:hello"/>
            <setBody id="_setBody1">
                <constant>{"result": "Hello"}</constant>
            </setBody>
        </route>
    </camelContext>
```

Add the following to the pom.xml in the project

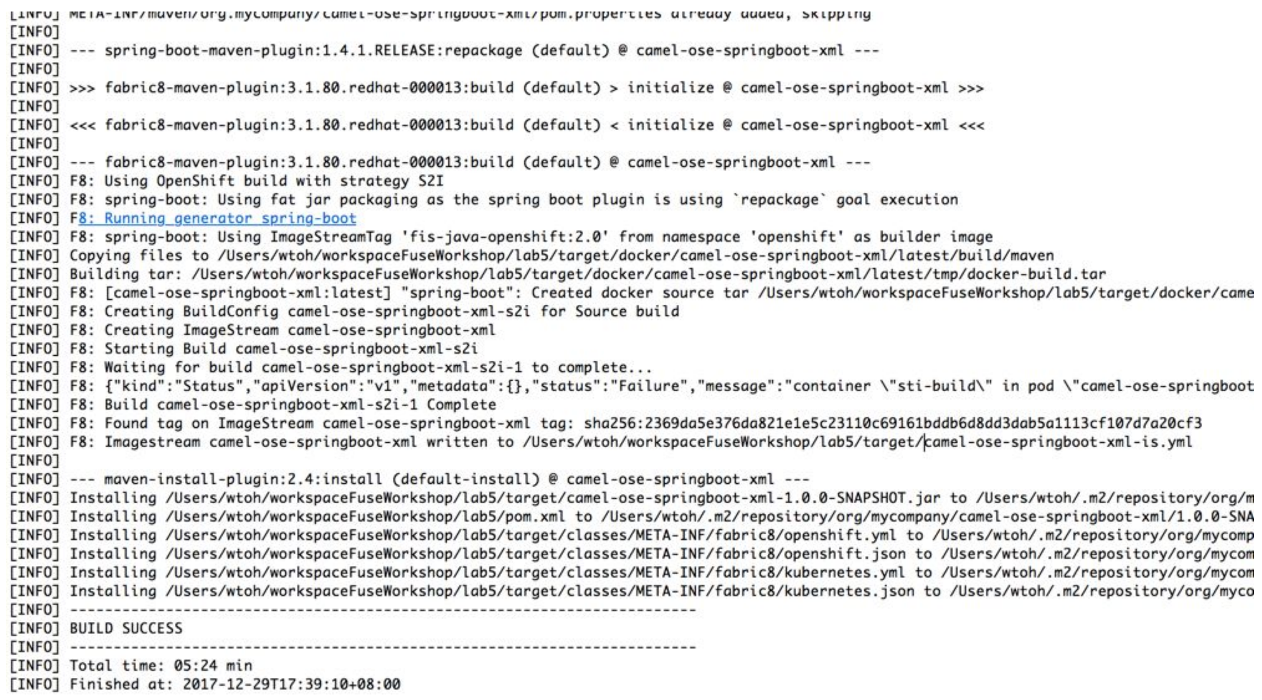
```
       <dependency>
       <groupId>org.apache.camel</groupId>
       <artifactId>camel-servlet-starter</artifactId>
       <version>2.18.1.redhat-000015</version>
     </dependency>
```

Execute the following on command line:

```
oc login https://master.ocp37-wayne.ddns.net:8443 --insecure-skip-tls-verify=false
```

```
oc project lab5-<your username>
```

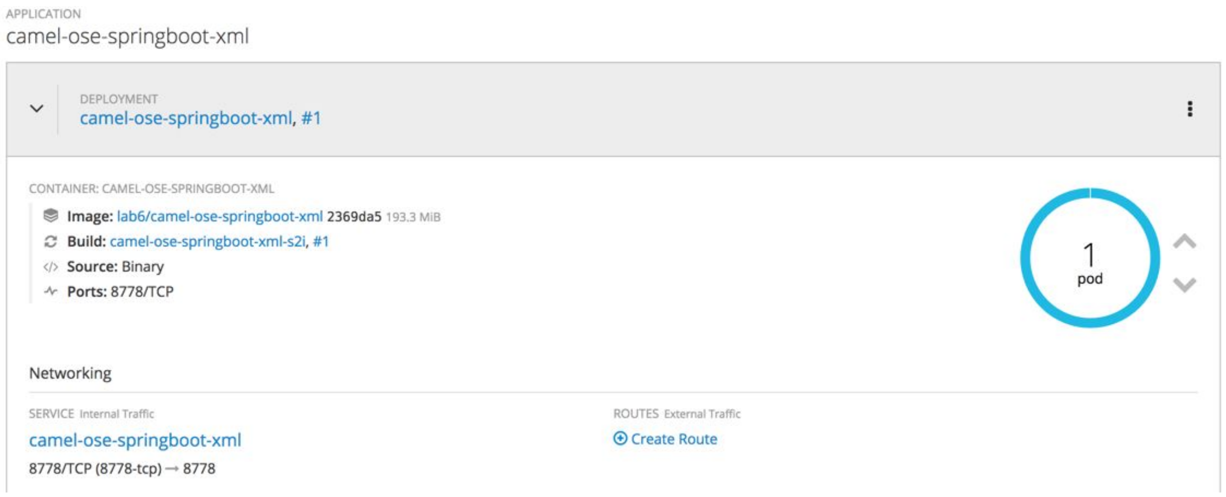
Right click on the project in JBDS and choose “run as maven install”



```
[INFO] META-INF/maven/org.mycompany/camel-ose-springboot-xml/pom.properties already added, skipping
[INFO]
[INFO] --- spring-boot-maven-plugin:1.4.1.RELEASE:repackage (default) @ camel-ose-springboot-xml ---
[INFO]
[INFO] >>> fabric8-maven-plugin:3.1.80.redhat-000013:build (default) > initialize @ camel-ose-springboot-xml >>>
[INFO]
[INFO] <<< fabric8-maven-plugin:3.1.80.redhat-000013:build (default) < initialize @ camel-ose-springboot-xml <<<
[INFO]
[INFO] --- fabric8-maven-plugin:3.1.80.redhat-000013:build (default) @ camel-ose-springboot-xml ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: spring-boot: Using fat jar packaging as the spring boot plugin is using `repackage` goal execution
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using ImageStreamTag 'fis-java-openshift:2.0' from namespace 'openshift' as builder image
[INFO] Copying files to /Users/wtoh/workspaceFuseWorkshop/lab5/target/docker/camel-ose-springboot-xml/latest/build/maven
[INFO] Building tar: /Users/wtoh/workspaceFuseWorkshop/lab5/target/docker/camel-ose-springboot-xml/latest/tmp/docker-build.tar
[INFO] F8: [camel-ose-springboot-xml:latest] "spring-boot": Created docker source tar /Users/wtoh/workspaceFuseWorkshop/lab5/target/docker/came
[INFO] F8: Creating BuildConfig camel-ose-springboot-xml-s2i for Source build
[INFO] F8: Creating ImageStream camel-ose-springboot-xml
[INFO] F8: Starting Build camel-ose-springboot-xml-s2i
[INFO] F8: Waiting for build camel-ose-springboot-xml-s2i-1 to complete...
[INFO] F8: {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Failure","message":"container \"sti-build\" in pod \"camel-ose-springboot
[INFO] F8: Build camel-ose-springboot-xml-s2i-1 Complete
[INFO] F8: Found tag on ImageStream camel-ose-springboot-xml tag: sha256:2369da5e376da821e1e5c23110c69161bddb6d8dd3dab5a1113cf107d7a20cf3
[INFO] F8: Imagestream camel-ose-springboot-xml written to /Users/wtoh/workspaceFuseWorkshop/lab5/target/camel-ose-springboot-xml-is.yml
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ camel-ose-springboot-xml ---
[INFO] Installing /Users/wtoh/workspaceFuseWorkshop/lab5/target/camel-ose-springboot-xml-1.0.0-SNAPSHOT.jar to /Users/wtoh/.m2/repository/org/m
[INFO] Installing /Users/wtoh/workspaceFuseWorkshop/lab5/pom.xml to /Users/wtoh/.m2/repository/org/mycompany/camel-ose-springboot-xml/1.0.0-SNA
[INFO] Installing /Users/wtoh/workspaceFuseWorkshop/lab5/target/classes/META-INF/fabric8/openshift.yml to /Users/wtoh/.m2/repository/org/mycomp
[INFO] Installing /Users/wtoh/workspaceFuseWorkshop/lab5/target/classes/META-INF/fabric8/openshift.json to /Users/wtoh/.m2/repository/org/mycom
[INFO] Installing /Users/wtoh/workspaceFuseWorkshop/lab5/target/classes/META-INF/fabric8/kubernetes.yml to /Users/wtoh/.m2/repository/org/mycom
[INFO] Installing /Users/wtoh/workspaceFuseWorkshop/lab5/target/classes/META-INF/fabric8/kubernetes.json to /Users/wtoh/.m2/repository/org/myco
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 05:24 min
[INFO] Finished at: 2017-12-29T17:39:10+08:00
```

Execute the following on the command line:

```
oc new-app camel-ose-springboot-xml
```



APPLICATION
camel-ose-springboot-xml

DEPLOYMENT
camel-ose-springboot-xml, #1

CONTAINER: CAMEL-OSE-SPRINGBOOT-XML
Image: lab6/camel-ose-springboot-xml 2369da5 193.3 MiB
Build: camel-ose-springboot-xml-s2i, #1
Source: Binary
Ports: 8778/TCP

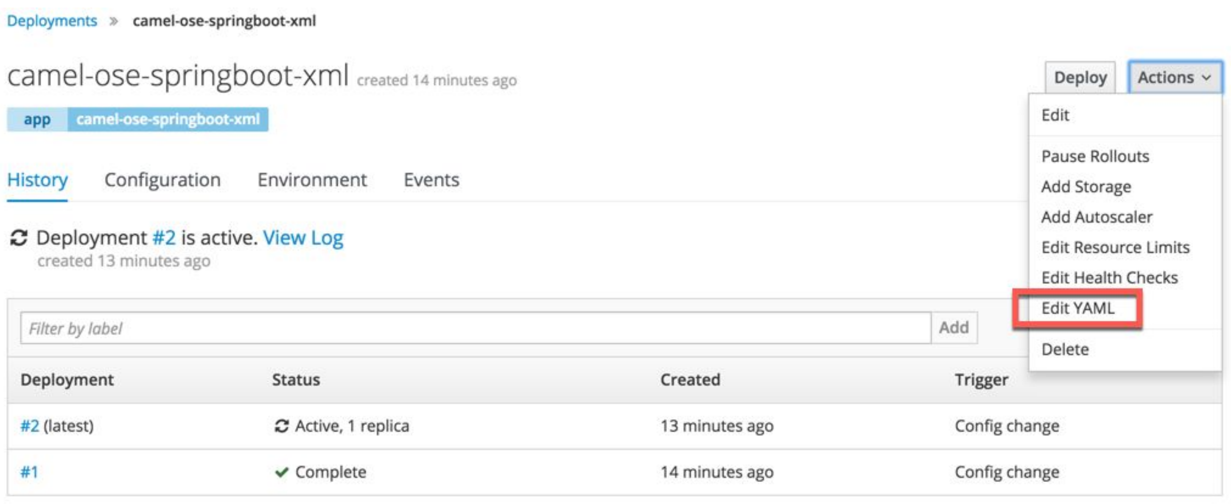
Networking

SERVICE Internal Traffic
camel-ose-springboot-xml
8778/TCP (8778-tcp) → 8778

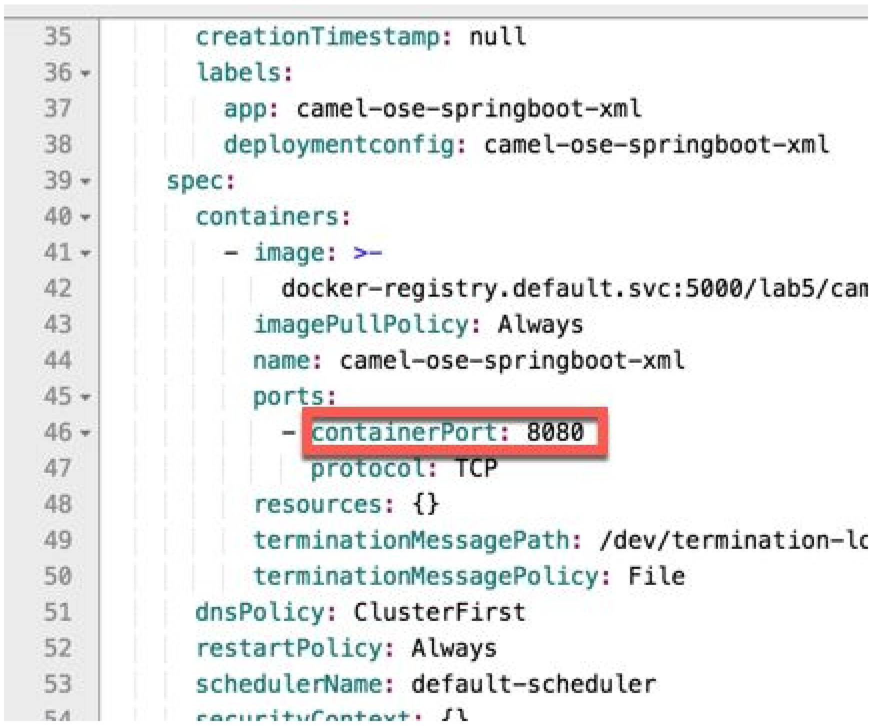
ROUTES External Traffic
Create Route

The pod should come up and running

Edit the deployment configuration by clicking on the “Edit YAML”

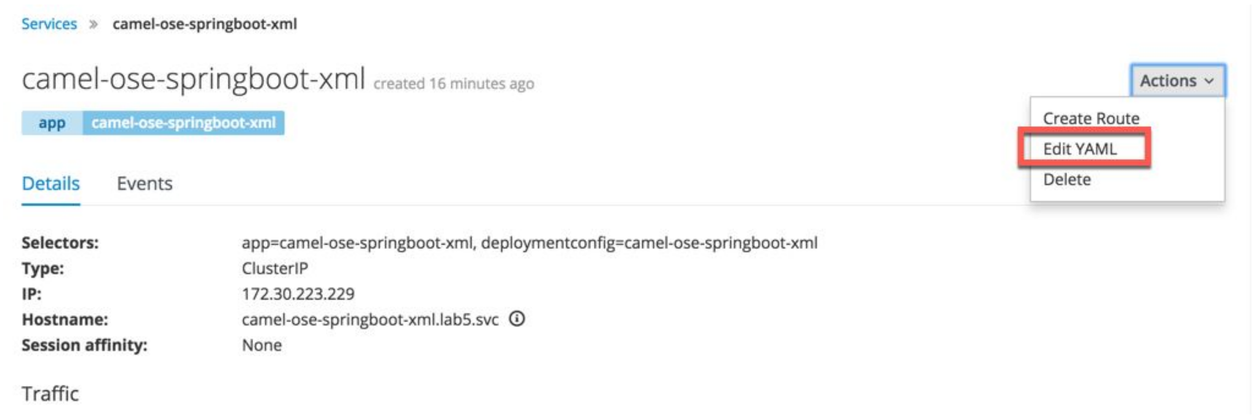


Deployment	Status	Created	Trigger
#2 (latest)	Active, 1 replica	13 minutes ago	Config change
#1	Complete	14 minutes ago	Config change

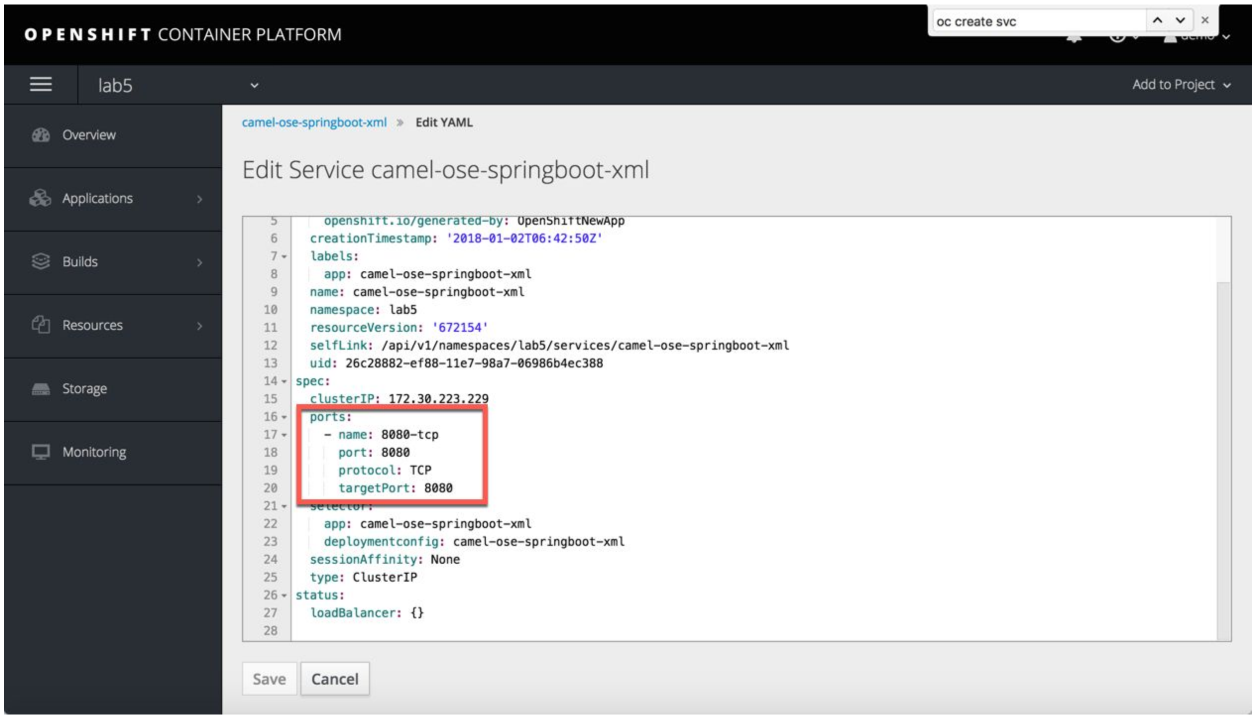


```
      creationTimestamp: null
      labels:
        app: camel-ose-springboot-xml
        deploymentconfig: camel-ose-springboot-xml
    spec:
      containers:
        - image: >-
            docker-registry.default.svc:5000/lab5/cam
          imagePullPolicy: Always
          name: camel-ose-springboot-xml
          ports:
            - containerPort: 8080
              protocol: TCP
          resources: {}
          terminationMessagePath: /dev/termination-lc
          terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
```

Edit the service configuration by clicking on the “Edit YAML”


Services » camel-ose-springboot-xml
camel-ose-springboot-xml created 16 minutes ago
app camel-ose-springboot-xml
Actions
Create Route
Edit YAML
Delete
Details
Events
Traffic

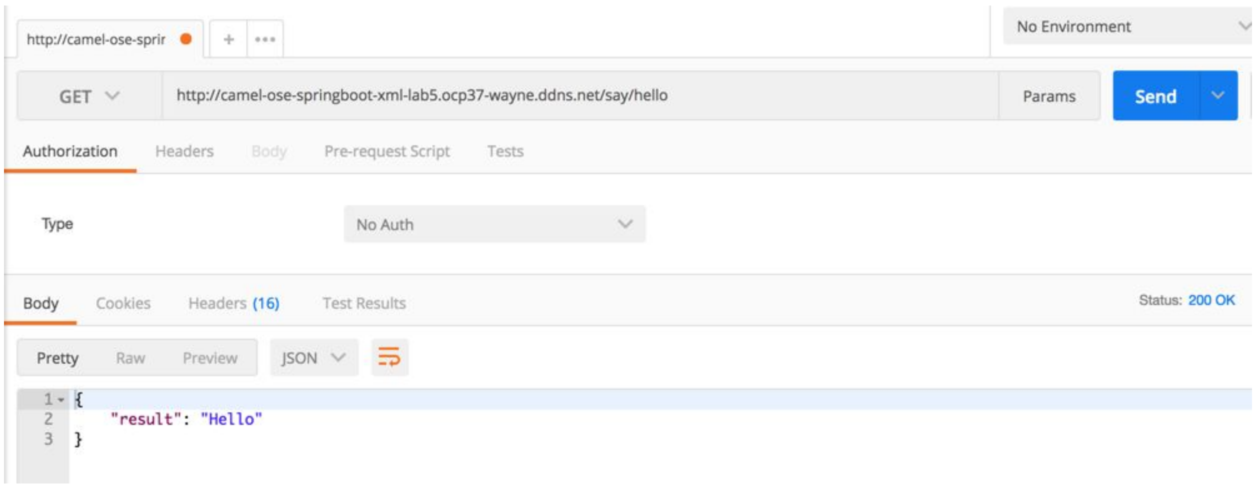
Selectors: app=camel-ose-springboot-xml, deploymentconfig=camel-ose-springboot-xml
Type: ClusterIP
IP: 172.30.223.229
Hostname: camel-ose-springboot-xml.lab5.svc
Session affinity: None


oc create svc
OPENSHIFT CONTAINER PLATFORM
lab5
Add to Project
Overview
Applications
Builds
Resources
Storage
Monitoring
camel-ose-springboot-xml » Edit YAML
Edit Service camel-ose-springboot-xml
Save
Cancel

```
    openshift.io/generated-by: OpenShiftNewApp
  creationTimestamp: '2018-01-02T06:42:50Z'
  labels:
    app: camel-ose-springboot-xml
  name: camel-ose-springboot-xml
  namespace: lab5
  resourceVersion: '672154'
  selfLink: /api/v1/namespaces/lab5/services/camel-ose-springboot-xml
  uid: 26c28882-ef88-11e7-98a7-06986b4ec388
spec:
  clusterIP: 172.30.223.229
  ports:
    - name: 8080-tcp
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: camel-ose-springboot-xml
    deploymentconfig: camel-ose-springboot-xml
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

Click on the “Create Route” button to access the application. You should now be able to access the application at the route.

Below is a screenshot of the REST output using PostMan.



http://camel-ose-springboot-xml-lab5.ocp37-wayne.ddns.net/say/hello

Type: No Auth

Status: 200 OK

```
{
    "result": "Hello"
}
```