Create backend SOAP µ-service

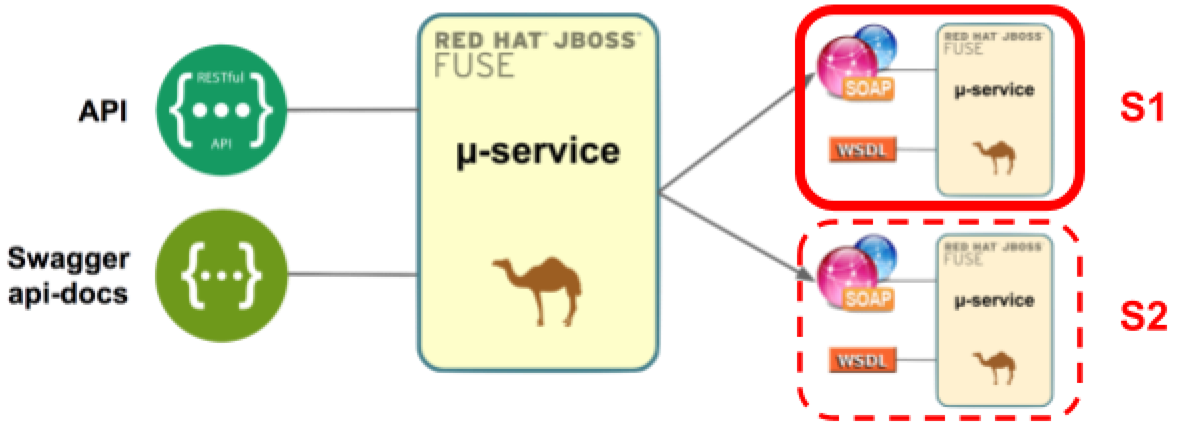
Prerequisites

This tutorial has been built and tested using the following product versions:

- JBDS [11.0.0] GA
- OpenShift [3.6.173.0.5] (Minishift)
- SoapUI 5.3.0

This project is a prerequisite to complete the orchestration REST service. It guides the reader to create a service S1 (see picture below).

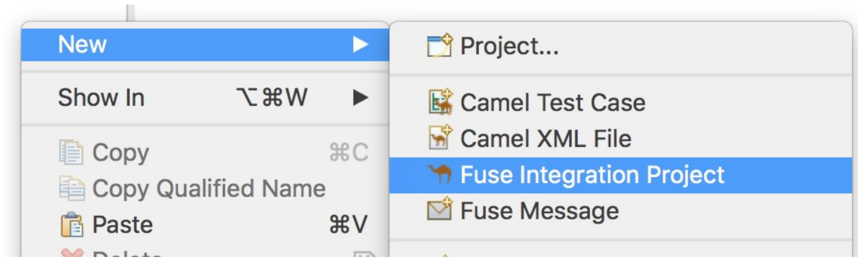
The document is valid also to create a service S2 (also a requirement), simply follow the same instructions using the name ‘S2’ instead.



Overview of full demo project

Project setup

Create a new Fuse project from JBDS

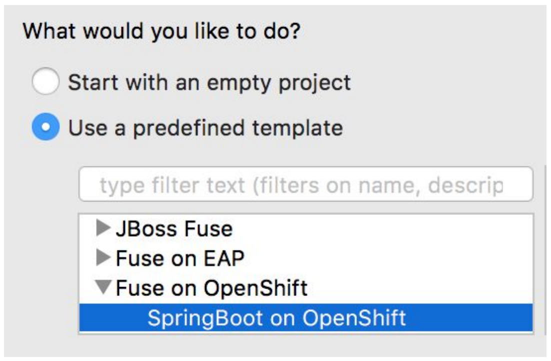


Project Name **“s1”**

Click ‘Next’ twice:

Next >

Next >



What would you like to do?

Start with an empty project

Use a predefined template

type filter text (filters on name, descrip

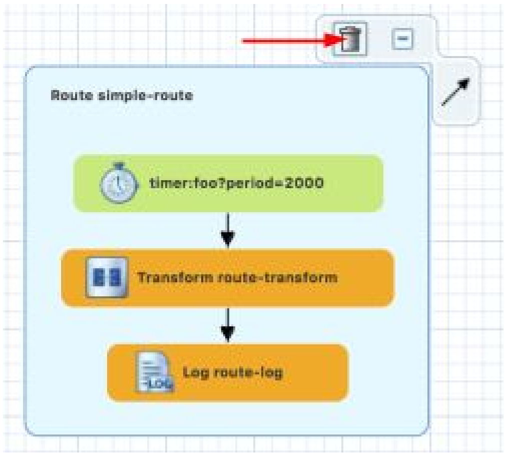
JBoss Fuse

Fuse on EAP

Fuse on OpenShift

SpringBoot on OpenShift

Delete default route:



Open the POM file (Project’s definition).
Update (for convenience) the Maven coordinates to:

```
    <groupId>org.demo</groupId>
    <artifactId>s1</artifactId>
    <version>1.0.0</version>
```

Also add the following POM dependency which is required to setup CXF

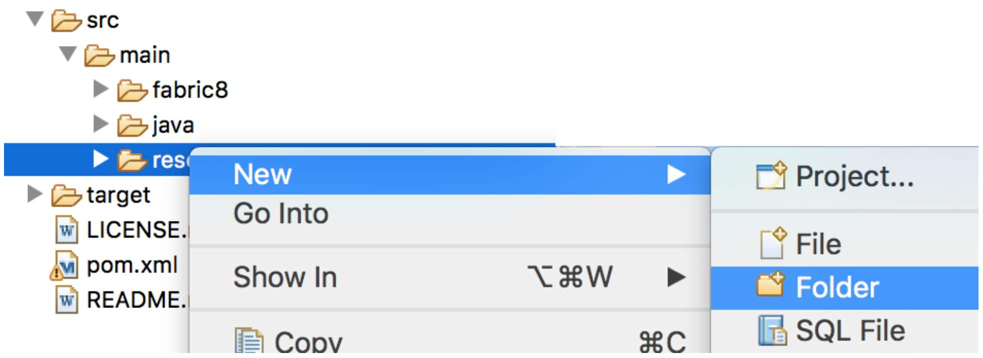
```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-spring-boot-starter-jaxws</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jaxb</artifactId>
</dependency>
```

Preparation of WSDL Interface

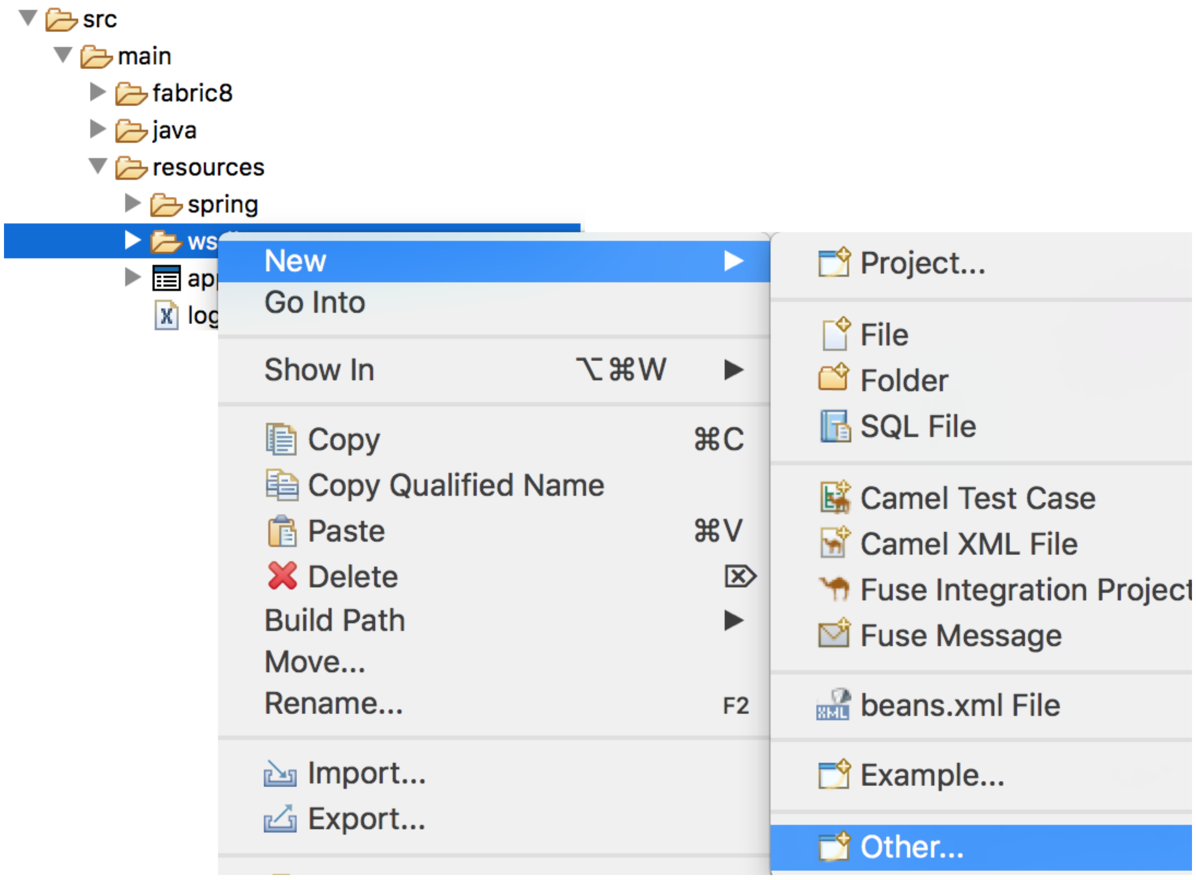
Under

```
src/main/resources
```

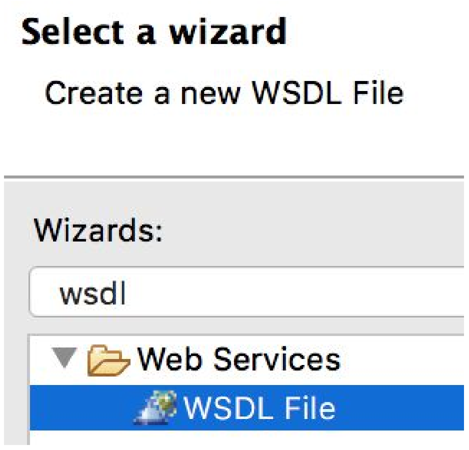
Create a new folder **‘wsdl’**



Create a WSDL file:

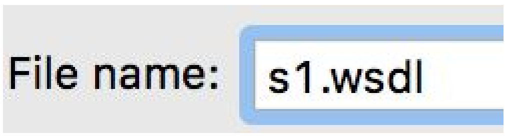


Open the wizard to create the WSDL file:
From the **'wsdl'** folder, right-click and select **New** → **Other…**

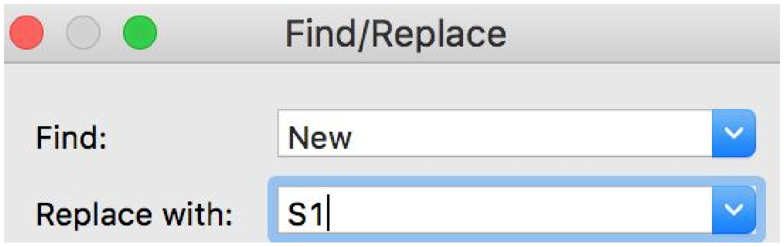

Select a wizard

Create a new WSDL File

Wizards: wsdl


File name: s1.wsdl

Open the WSDL and make some modification to customise the service

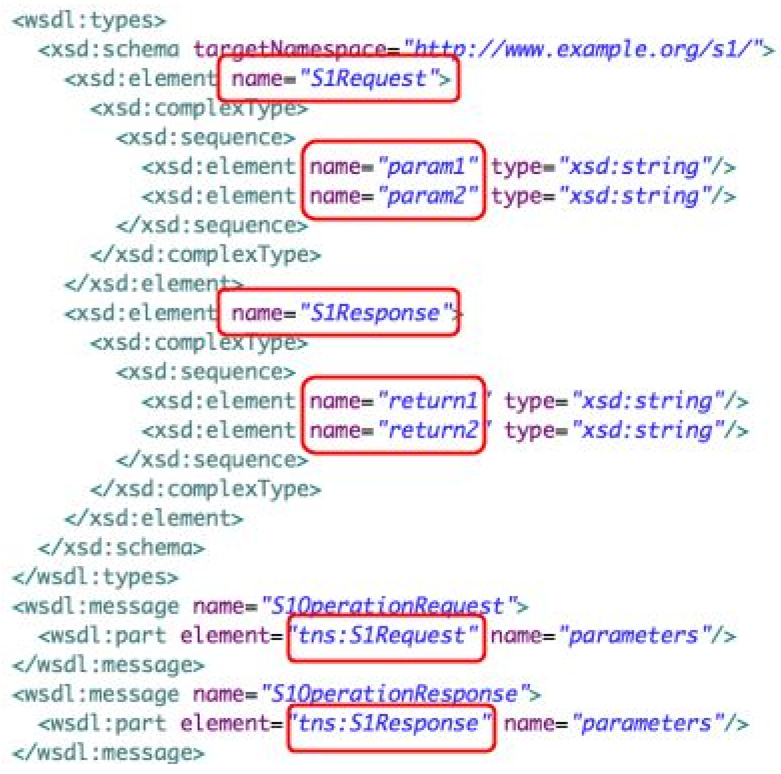


Find: New

Replace with: S1

Modify the Schema a bit:

- 2 parameters in/out
- Element names to request/response



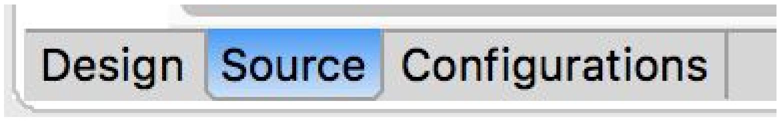
```
<wsdl:types>
  <xsd:schema targetNamespace="http://www.example.org/s1/">
    <xsd:element name="S1Request">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="param1" type="xsd:string"/>
          <xsd:element name="param2" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="S1Response">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="return1" type="xsd:string"/>
          <xsd:element name="return2" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>
<wsdl:message name="S1OperationRequest">
  <wsdl:part element="tns:S1Request" name="parameters"/>
</wsdl:message>
<wsdl:message name="S1OperationResponse">
  <wsdl:part element="tns:S1Response" name="parameters"/>
</wsdl:message>
```

Add POM plugin:
(this will generate the sources and classes necessary for the SOAP endpoints)

```
<plugin>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-codegen-plugin</artifactId>
        <executions>
          <execution>
                <id>generate-sources</id>
                <phase>generate-sources</phase>
                <configuration>
                  <sourceRoot>${basedir}/target/generated/src/main/java</sourceRoot>
                  <wsdlRoot>${basedir}/src/main/resources/wsdl</wsdlRoot>
                </configuration>
                <goals>
                  <goal>wsdl2java</goal>
                </goals>
          </execution>
        </executions>
</plugin>
```

Preparation of SOAP endpoint

Open **'Camel-Context.xml'** in Source view



a. Add the namespace:

xmlns:cxf="http://camel.apache.org/schema/cxf"

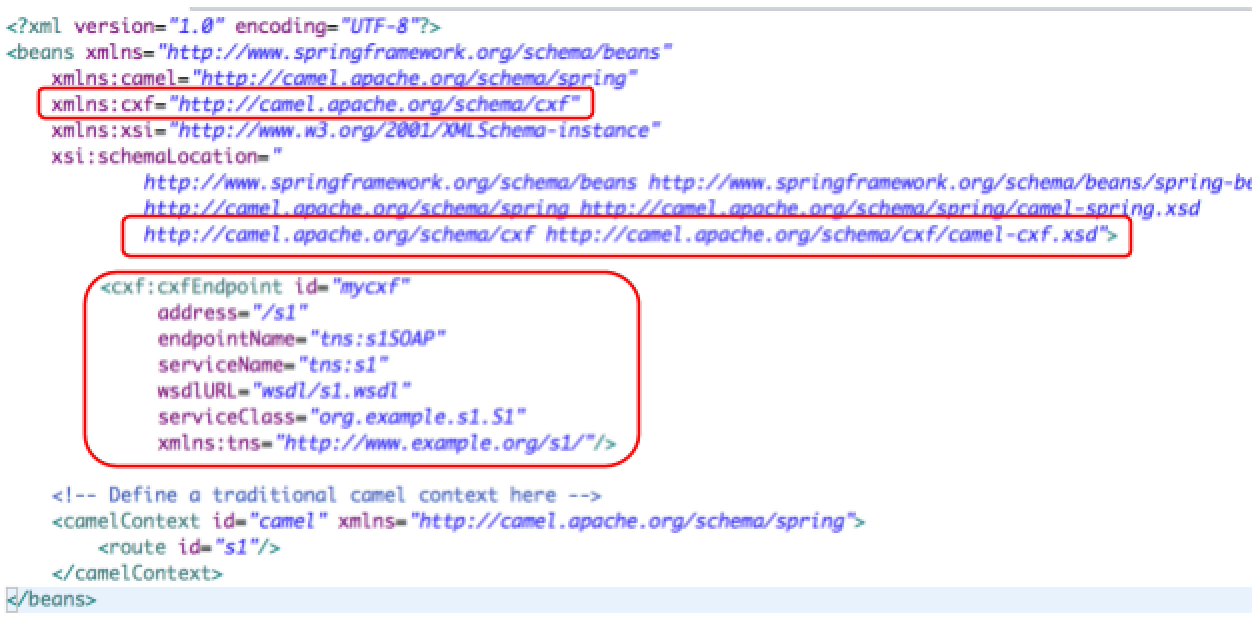
with 'xsi:schemaLocation' set to

http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd

b. and the endpoint in the Bean part (outside the Camel context part):

```
<cxf:cxfEndpoint id="mycxf"
        address="/s1"
        endpointName="tns:s1SOAP"
        serviceName="tns:s1"
        wsdlURL="wsdl/s1.wsdl"
        serviceClass="org.example.s1.S1"
        xmlns:tns="http://www.example.org/s1/"/>
```

Should look like this:



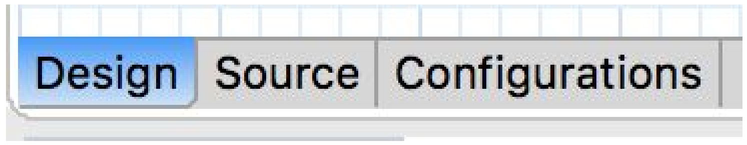
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:camel="http://camel.apache.org/schema/spring"
    xmlns:cxf="http://camel.apache.org/schema/cxf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        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

        <cxf:cxfEndpoint id="mycxf"
             address="/s1"
             endpointName="tns:s1SOAP"
             serviceName="tns:s1"
             wsdlURL="wsdl/s1.wsdl"
             serviceClass="org.example.s1.S1"
             xmlns:tns="http://www.example.org/s1/"/>

    <!-- Define a traditional camel context here -->
    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
        <route id="s1"/>
    </camelContext>
</beans>
```

Let’s also prepare a demo response for S1 by defining a Bean as follows:
(the class will be autogenerated by the POM plugin we included)

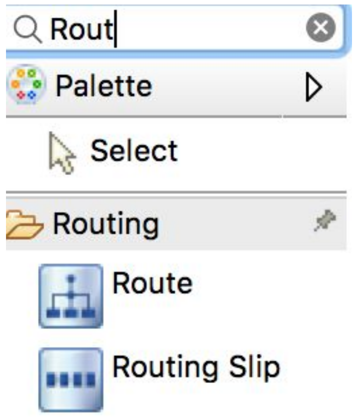
```
<bean id="response" class="org.example.s1.S1Response">
   <property name="return1" value="value1 S1"/>
   <property name="return2" value="value2 S1"/>
</bean>
```

Create Route Definition

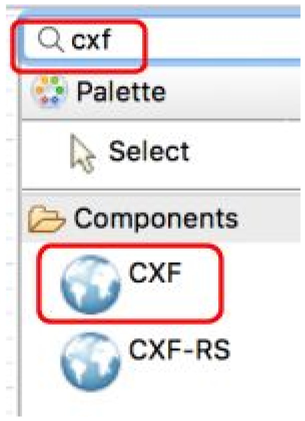
Change to ‘Design’ view



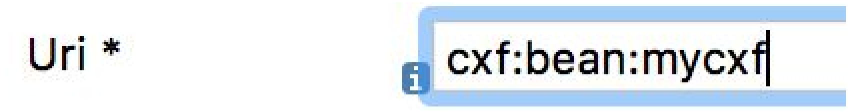
Create new Route (type in search bar) and drag and drop ‘Route’ to the drawing pane.



Choose from the palette the CXF starting activity:



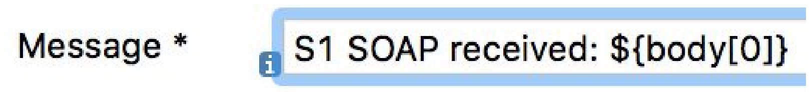
Define the URI as:



Uri * cxf:bean:mycxf

Where ‘mycxf’ denotes our previously CXF defined Endpoint.

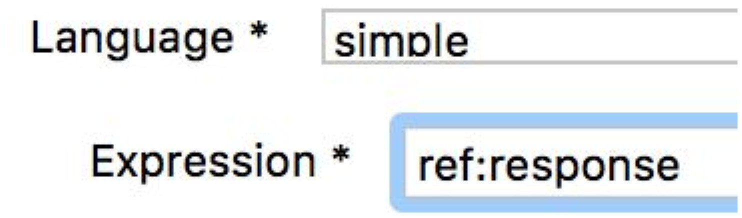
We also include a **Log** activity with the following message:



Message * S1 SOAP received: ${body[0]}

(CXF places response content in index 0)

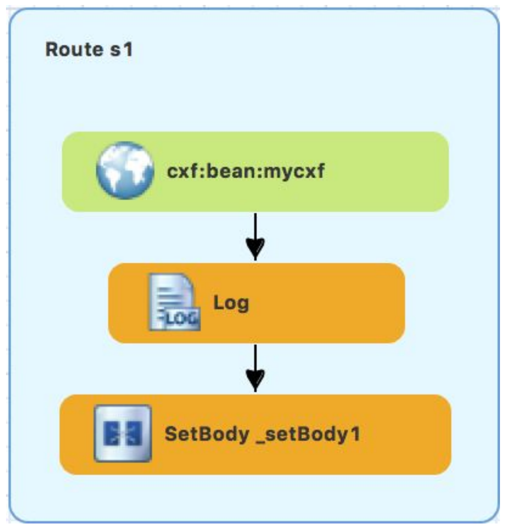
We also include a **'setBody'** activity to inject the Bean response as follows:



Language * simple

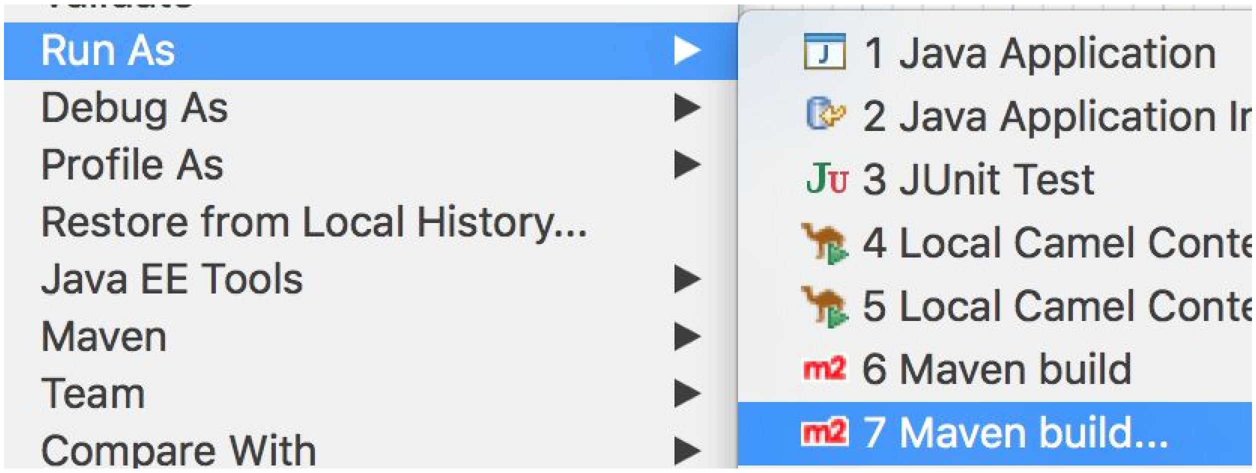
Expression * ref:response

Our Route has been completed and should look like:

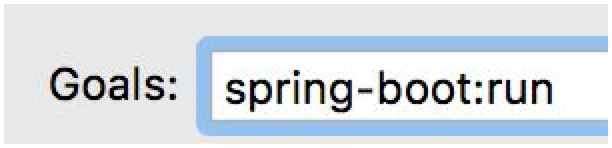


Test locally

Build with Maven:



Using the goal ‘spring-boot:run’



Goals: spring-boot:run

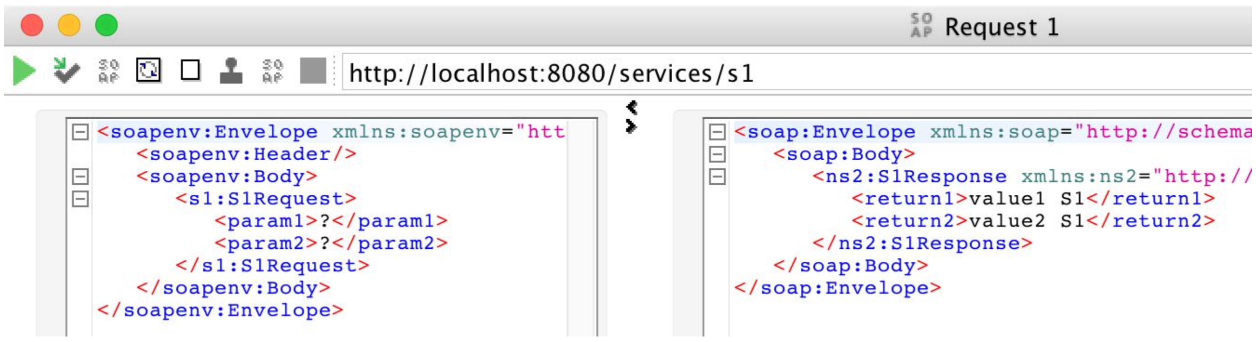
Click **‘Run’**

Eclipse should launch a SpringBoot application successfully
(if not, you’ll have to fix the problems)

You can now test the service from SOAPUI for instance.
Create a new project from a WSDL by providing the default URL:

```
“http://localhost:8080/services/s1?wsdl”
```

SOAPUI should succeed to obtain a response:



```
http://localhost:8080/services/s1
```

```
<soapenv:Envelope xmlns:soapenv="htt
   <soapenv:Header/>
   <soapenv:Body>
      <s1:S1Request>
         <param1>?</param1>
         <param2>?</param2>
      </s1:S1Request>
   </soapenv:Body>
</soapenv:Envelope>
```

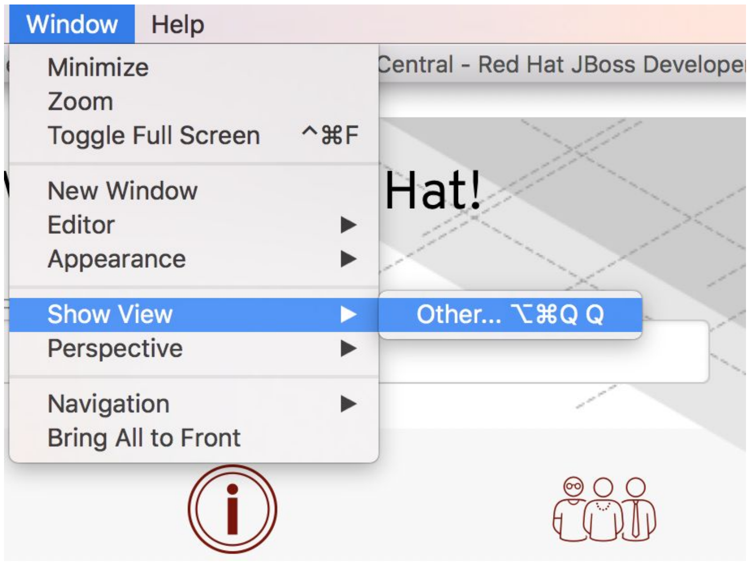
```
<soap:Envelope xmlns:soap="http://schema
   <soap:Body>
      <ns2:S1Response xmlns:ns2="http://
         <return1>value1 S1</return1>
         <return2>value2 S1</return2>
      </ns2:S1Response>
   </soap:Body>
</soap:Envelope>
```

You’ve successfully created a SOAP µ-service !!

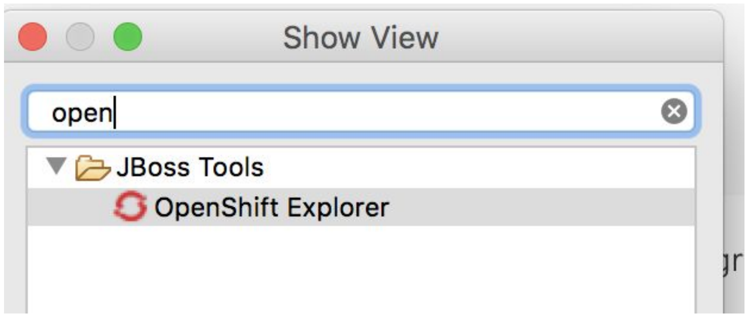
Deploy in OpenShift

(in this example 'minishift')

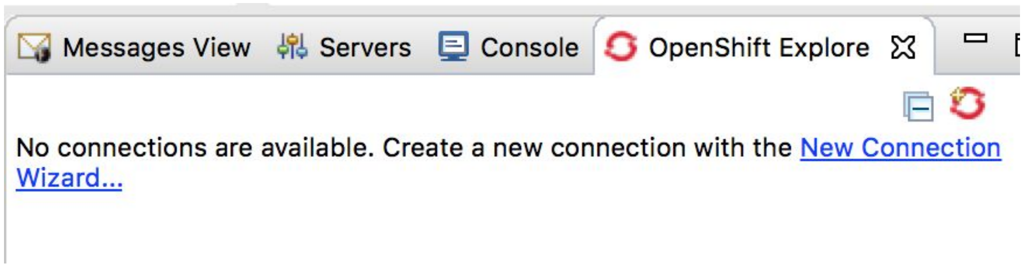
Open the **'OpenShift Explorer'** View:



Type in the search bar to find the OpenShift view:

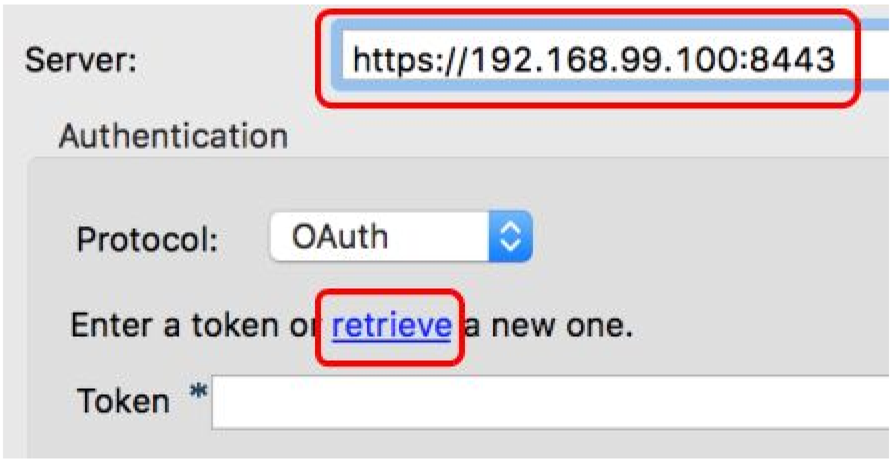


The first time you'll be presented with the following Wizard link:



No connections are available. Create a new connection with the New Connection Wizard...

Enter your environment’s address and click ‘retrieve’ to pre-populate the Token

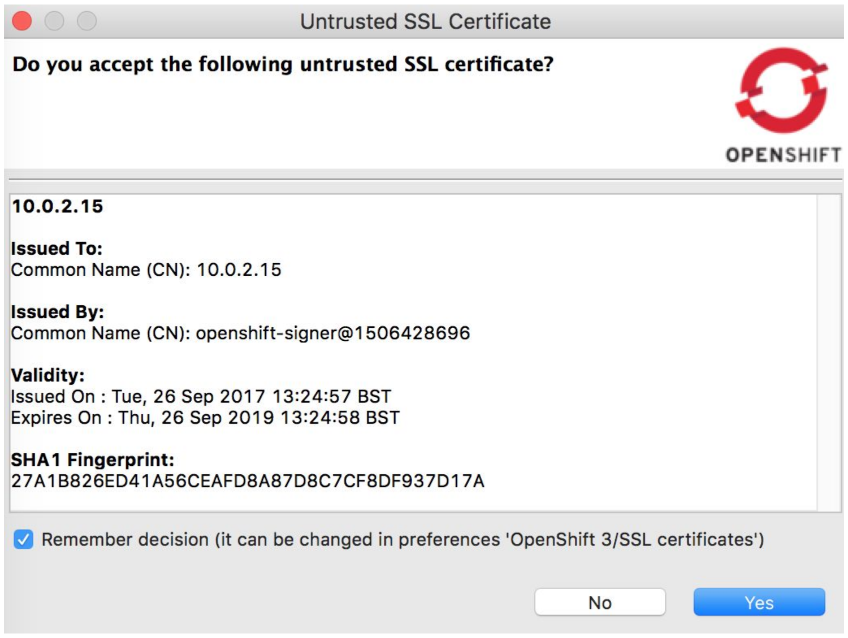


Server: https://192.168.99.100:8443

Protocol: OAuth

Enter a token or retrieve a new one.

You’ll be asked to accept the SSL certificate, click ‘Yes’



Untrusted SSL Certificate

Do you accept the following untrusted SSL certificate?

10.0.2.15

Issued To:
Common Name (CN): 10.0.2.15

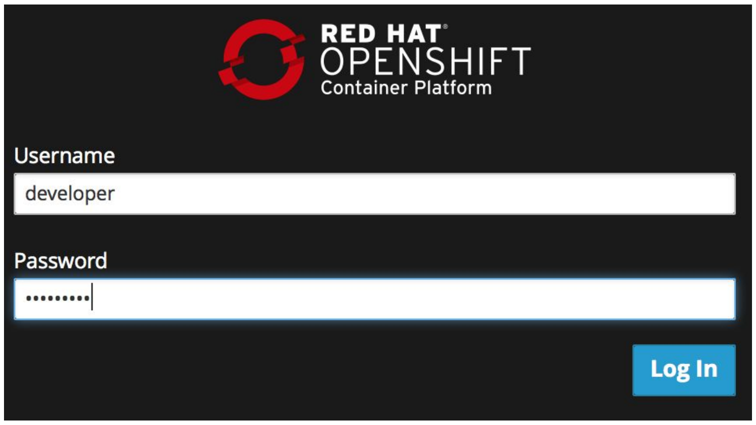
Issued By:
Common Name (CN): openshift-signer@1506428696

Validity:
Issued On : Tue, 26 Sep 2017 13:24:57 BST
Expires On : Thu, 26 Sep 2019 13:24:58 BST

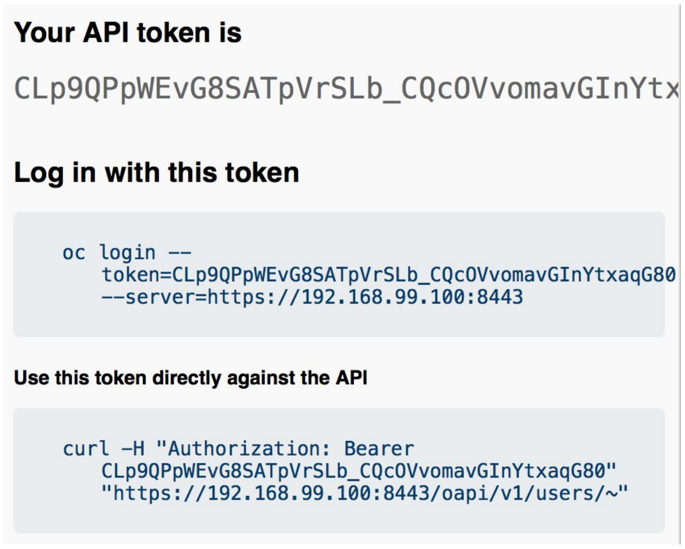
SHA1 Fingerprint:
27A1B826ED41A56CEAFD8A87D8C7CF8DF937D17A

✓ Remember decision (it can be changed in preferences 'OpenShift 3/SSL certificates')

Then type in your username/password
(minishift default developer/developer)



A token is generated, click 'close'



Your API token is

CLp9QPpWEvG8SATpVrSLb_CQcOVvomavGInYtx

Log in with this token

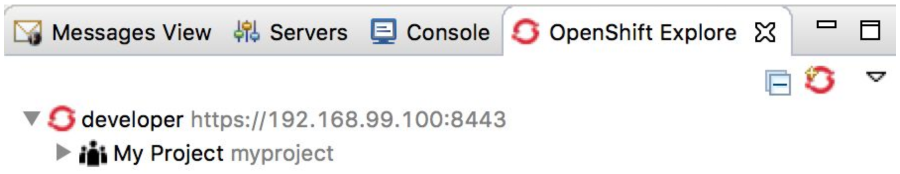
```
oc login --
   token=CLp9QPpWEvG8SATpVrSLb_CQcOVvomavGInYtxaqG80
   --server=https://192.168.99.100:8443
```

Use this token directly against the API

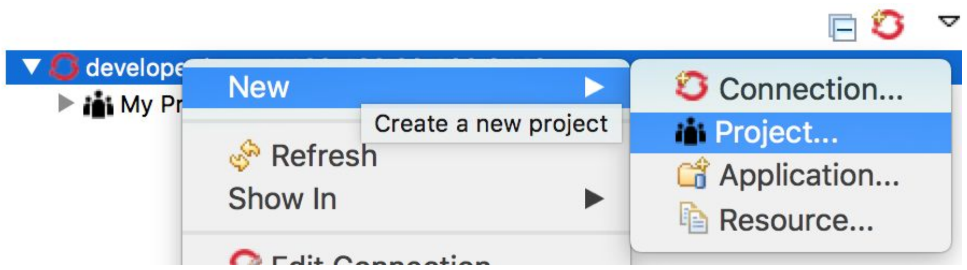
```
curl -H "Authorization: Bearer
   CLp9QPpWEvG8SATpVrSLb_CQcOVvomavGInYtxaqG80"
   "https://192.168.99.100:8443/oapi/v1/users/~"
```

Then click **'Finish'** on the Wizard's final step.

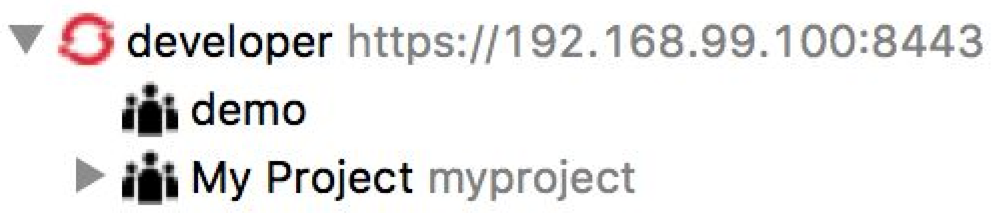
Your OpenShift environment will be viewable from JBDS:



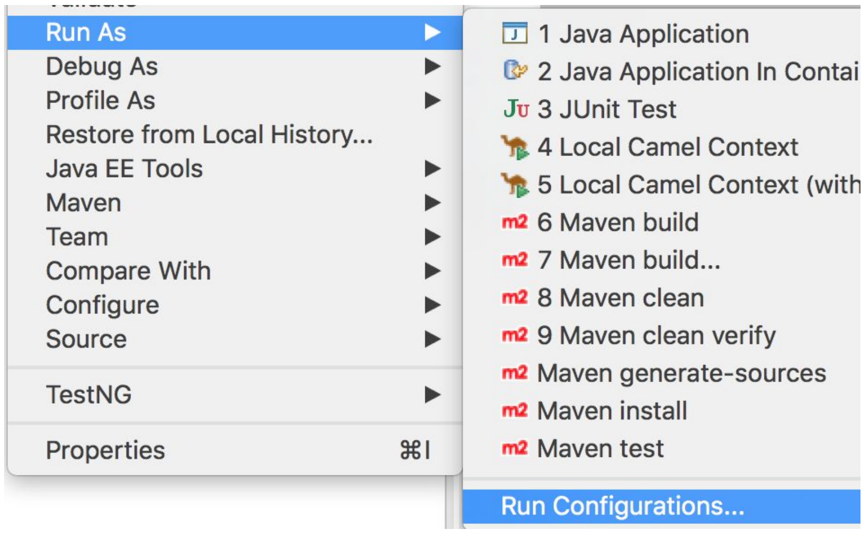
Create a new Project **'demo'**



You should see:

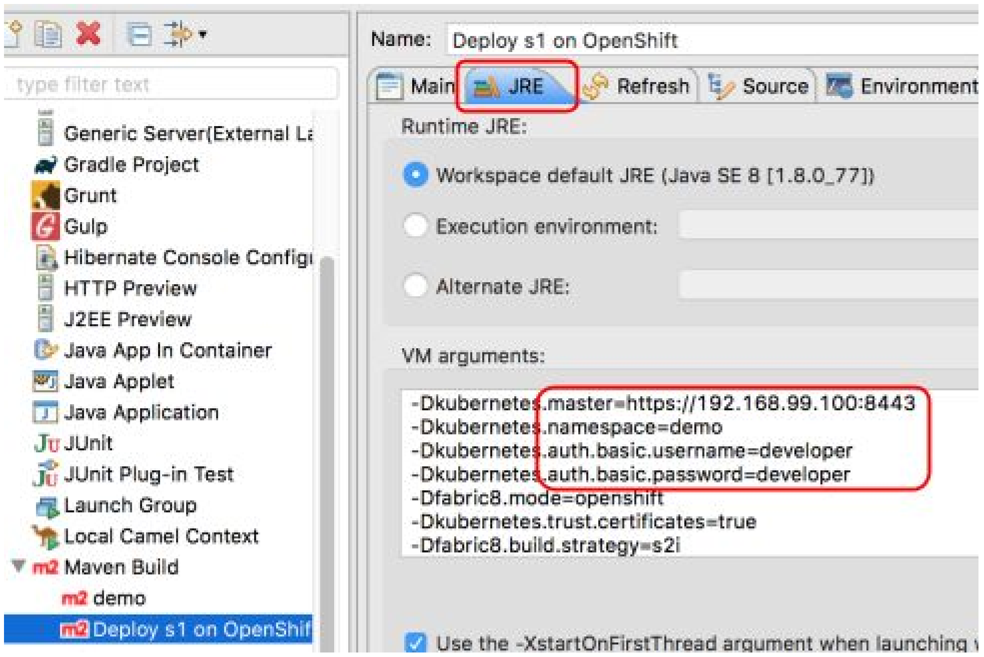


Now from the Project's root, right click and select **Run As → Run Configurations…**



And ensure the following JRE parameters are properly set:

```
-Dkubernetes.master=https://192.168.99.100:8443
-Dkubernetes.namespace=demo
-Dkubernetes.auth.basic.username=developer
-Dkubernetes.auth.basic.password=developer
```



Name: Deploy s1 on OpenShift

Main | JRE | Refresh | Source | Environment

Runtime JRE:

● Workspace default JRE (Java SE 8 [1.8.0_77])

○ Execution environment:

○ Alternate JRE:

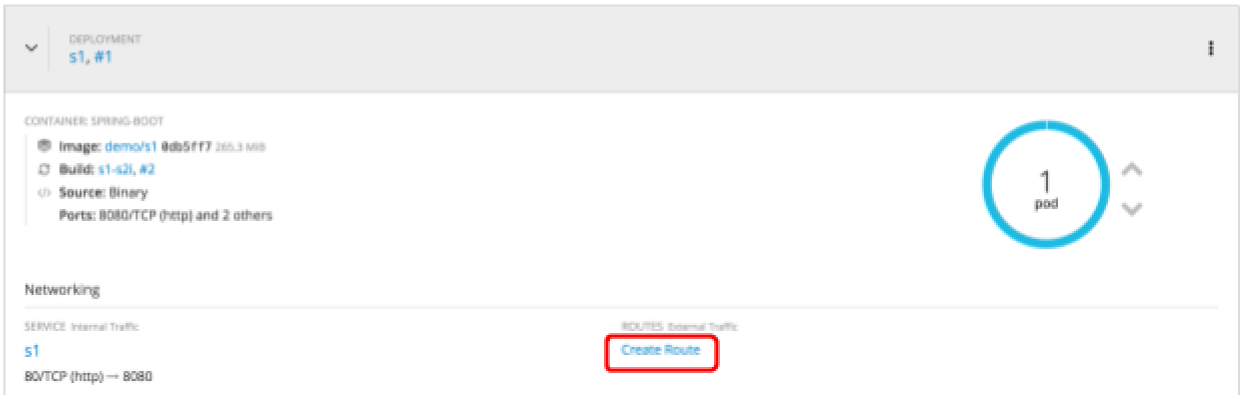
VM arguments:

```
-Dkubernetes.master=https://192.168.99.100:8443
-Dkubernetes.namespace=demo
-Dkubernetes.auth.basic.username=developer
-Dkubernetes.auth.basic.password=developer
-Dfabric8.mode=openshift
-Dkubernetes.trust.certificates=true
-Dfabric8.build.strategy=s2i
```

✓ Use the -XstartOnFirstThread argument when launching

Click **'Run'** and JBDS should trigger the deployment in OpenShift.

Once deployed, you should be able from the console to visualise the pod running. Create a route to expose the service so that it can be tested externally:



DEPLOYMENT
s1, #1

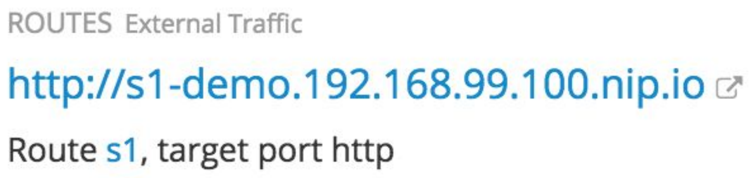
CONTAINER: SPRING-BOOT
Image: demo/s1 0db5ff7 265.3 MiB
Build: s1-s2i, #2
Source: Binary
Ports: 8080/TCP (http) and 2 others

Networking

SERVICE Internal Traffic
s1
80/TCP (http) → 8080

ROUTES External Traffic
Create Route

Once the Route was created (use all default values), you should see the following:



ROUTES External Traffic

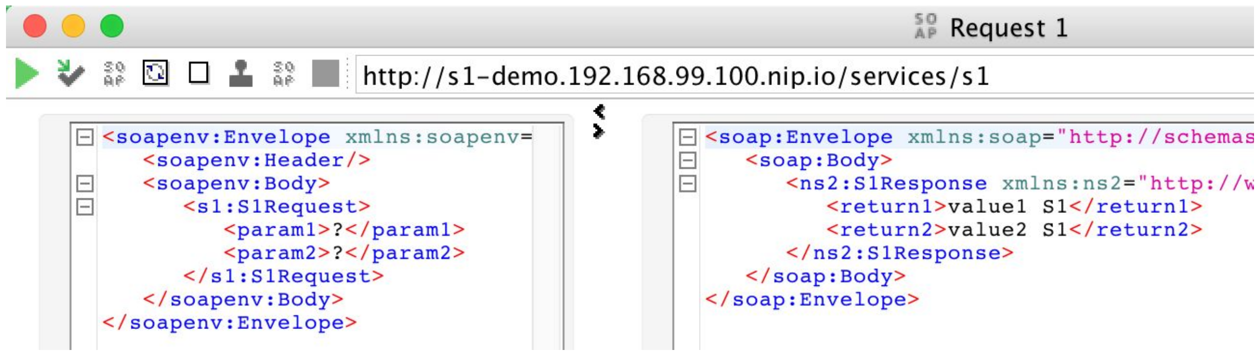
http://s1-demo.192.168.99.100.nip.io

Route s1, target port http

You should now be able to test the service from SOAPUI again using the above URL address. Create a new project from a WSDL by providing the OpenShift URL:

```
“http://s1-demo.192.168.99.100.nip.io/services/s1?wsdl”
```

Fire a test, should get a response as shown below:



http://s1-demo.192.168.99.100.nip.io/services/s1

```
<soapenv:Envelope xmlns:soapenv=
   <soapenv:Header/>
   <soapenv:Body>
      <s1:S1Request>
         <param1>?</param1>
         <param2>?</param2>
      </s1:S1Request>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap:Envelope xmlns:soap="http://schemas
   <soap:Body>
      <ns2:S1Response xmlns:ns2="http://w
         <return1>value1 S1</return1>
         <return2>value2 S1</return2>
      </ns2:S1Response>
   </soap:Body>
</soap:Envelope>
```

You’ve successfully deployed the SOAP service in OpenShift !!