Exercise 10

Mediate a service interaction using an ESB to convert from an inbound REST call into an existing SOAP call.

Prior Knowledge
Basic understanding HTTP verbs, REST architecture, SOAP and XML

Objectives
Understand the basic ESB flow, create a flow using the ESB tooling in Eclipse, upload to the ESB using Eclipse Remote Server model, mediate between REST and SOAP.

Software Requirements
(see separate document for installation of these)

- Java Development Kit 8
- Apache Maven 3.3.9 or later
- WSO2 Developer Studio 3.8.0
- Tomcat / Docker
- WSO2 ESB 4.9.0

Overview

In this lab, we are going to take a WSDL/SOAP payment service, which is *very* loosely modeled on a real SOAP API (Barclaycard SmartPay https://www.barclaycard.co.uk/business/accepting-payments/website-payments/web-developer-resources/smartpay#tabbox1).
Our aim is to convert this into a simpler HTTP/JSON interface. We probably won't get as far as any RESTful concepts as we won't have the opportunity to add resources, HATEOAS, etc. But will look at how those could be added with more time.

1. Before we install the ESB we need to host some services to interact with. To do this we are going to use a payment server running in Docker.

   ```
   sudo docker pull pizak/pay
   sudo docker run -d -p 8888:8080 pizak/pay
   ```

   This offers the docker based service (which is a WAR file running in Tomcat) at port 8888.

2. We are also going to intercept all the messages between the ESB and the backend using mitmdump. In a new terminal window start:

   ```
   mitmdump --port 8080 -dd --reverse http://localhost:8888
   ```

   This puts the port back to 8080, but lets us see all the traffic to the backend.

3. Check that it is running:
   Browse: http://localhost:8080/pay/services/paymentSOAP?wsdl

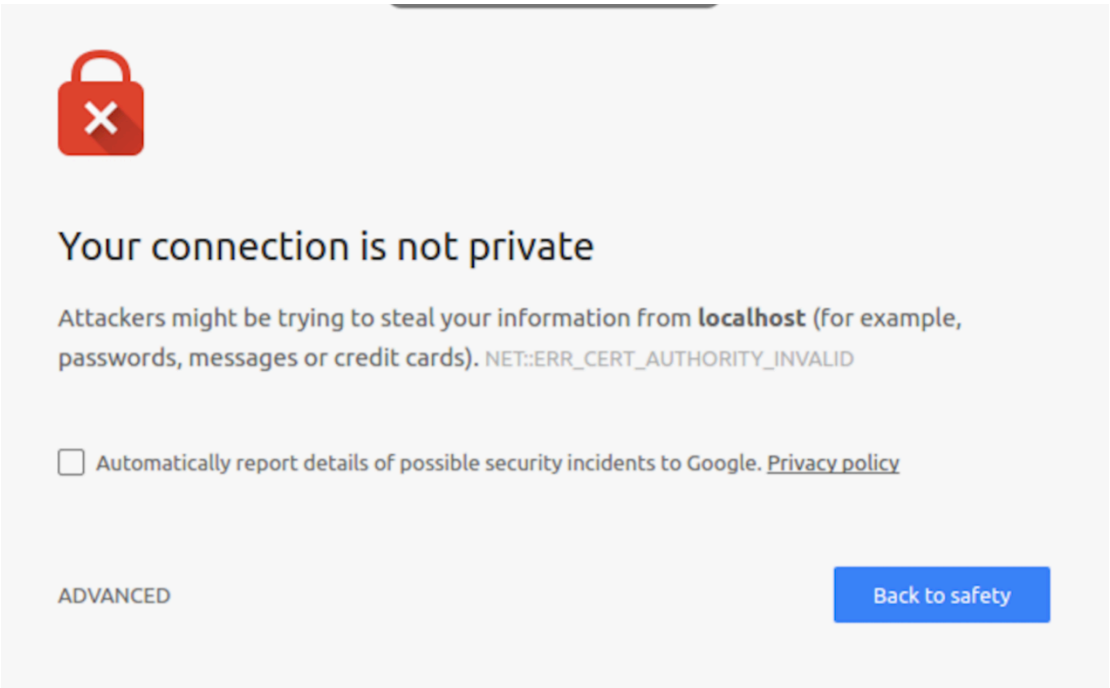
4. Now start up the WSO2 ESB:
   ```
   cd ~/servers/wso2esb-4.9.0/
   bin/wso2server.sh
   ```

5. You should see something like:

   ```
   oxsoa@oxsoa:~/servers/wso2esb-4.9.0$ bin/wso2server.sh
   JAVA_HOME environment variable is set to /usr/lib/jvm/java-8-openjdk-amd64/
   CARBON_HOME environment variable is set to /home/oxsoa/servers/wso2esb-4.9.0
   OpenJDK 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed
   in 8.0
   [2016-06-03 15:57:18,988]  INFO - CarbonCoreActivator Starting WSO2 Carbon...
   [2016-06-03 15:57:18,997]  INFO - CarbonCoreActivator Operating System : Linux 4.4.0-22-
   generic, amd64
   [2016-06-03 15:57:18,998]  INFO - CarbonCoreActivator Java Home        :
   /usr/lib/jvm/java-8-openjdk-amd64/jre
   [2016-06-03 15:57:18,998]  INFO - CarbonCoreActivator Java Version     : 1.8.0_91
   [2016-06-03 15:57:18,998]  INFO - CarbonCoreActivator Java VM          : OpenJDK 64-Bit
   Server VM 25.91-b14,Oracle Corporation
   [2016-06-03 15:57:18,998]  INFO - CarbonCoreActivator Carbon Home      :
   /home/oxsoa/servers/wso2esb-4.9.0
   ...
   ```

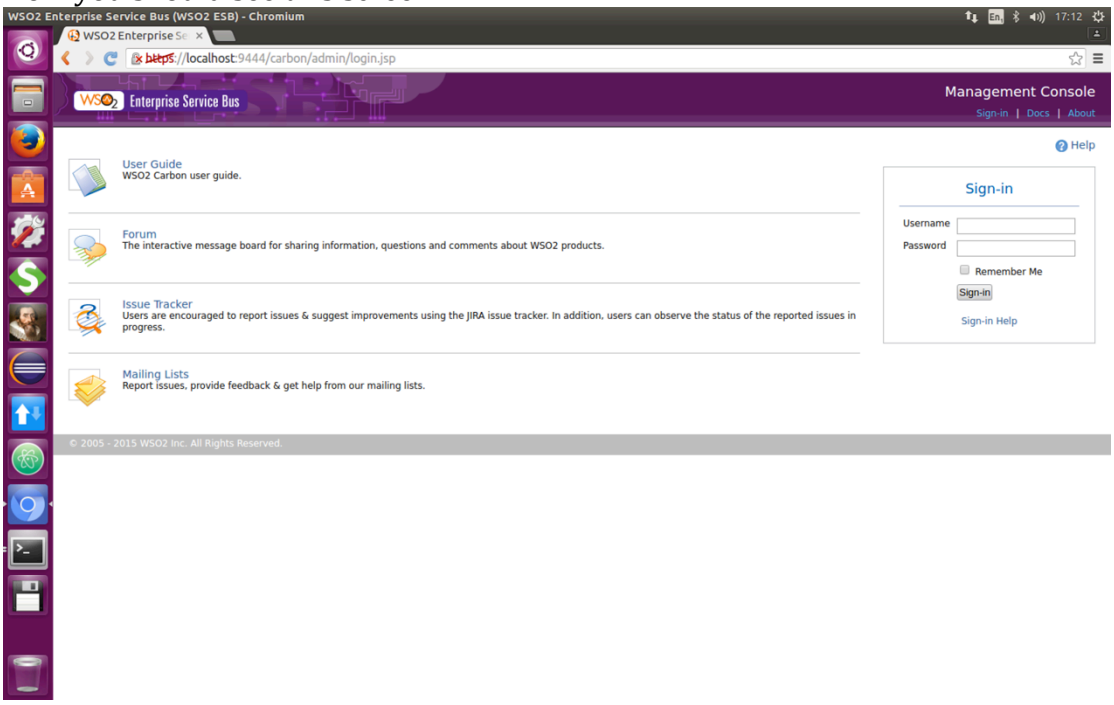
6. Start up a browser and go to https://localhost:9444

7. You may see a page like this:



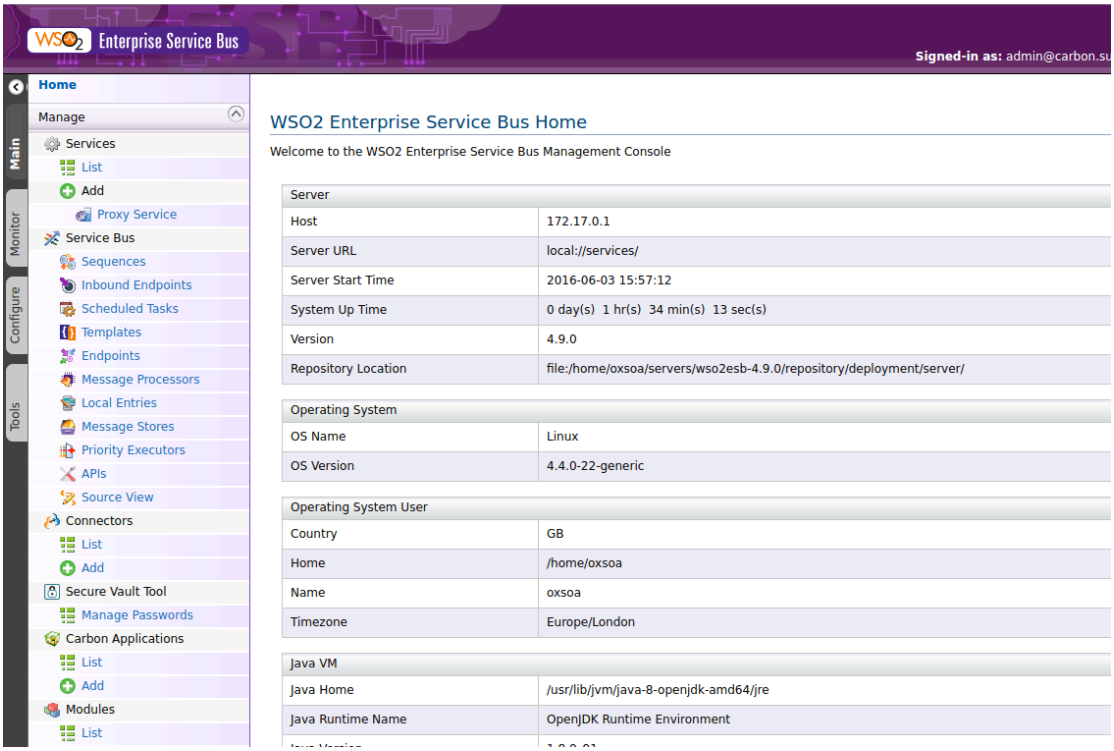
If you do, click on Advanced and Proceed to localhost

8. Now you should see this screen:

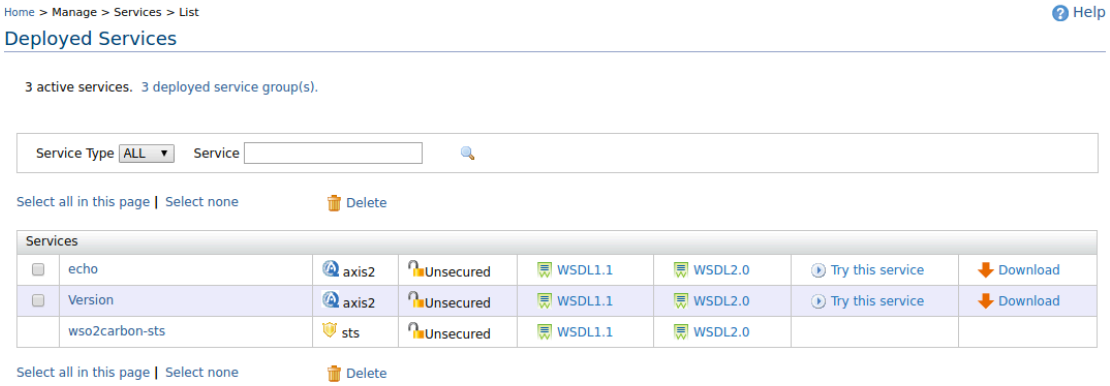


9. Log in with the userid/password: **admin/admin**

10. You should see a Web console like this:

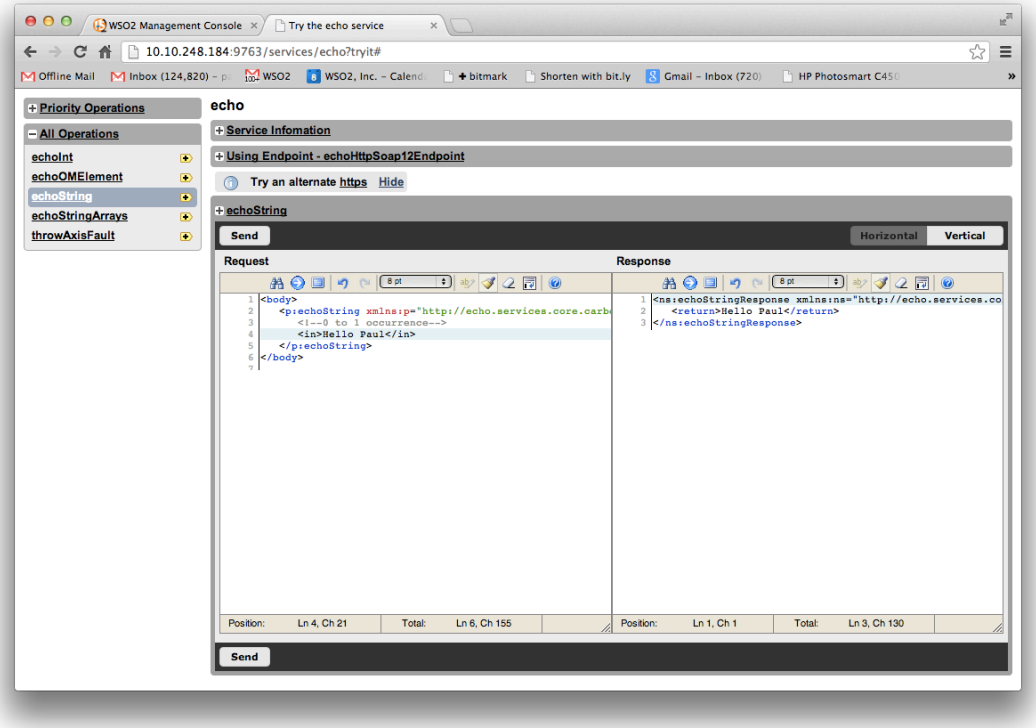


11. Click on **Services->List** in the left hand menu. You should see something like this:



12. To see if the basic “echo” service is working click on “Try this service” next to echo.

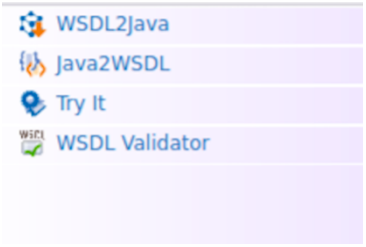
13. You will see a “test” client. (If you don’t Chrome may have blocked a popup, in which case you need to enable popups on this site).



Select the **echoString** operation, modify the XML **(replace the ?)** and click **Send**. If it didn’t work, you might have an odd network setup with VMWare. Try changing the URL to use 127.0.0.1.

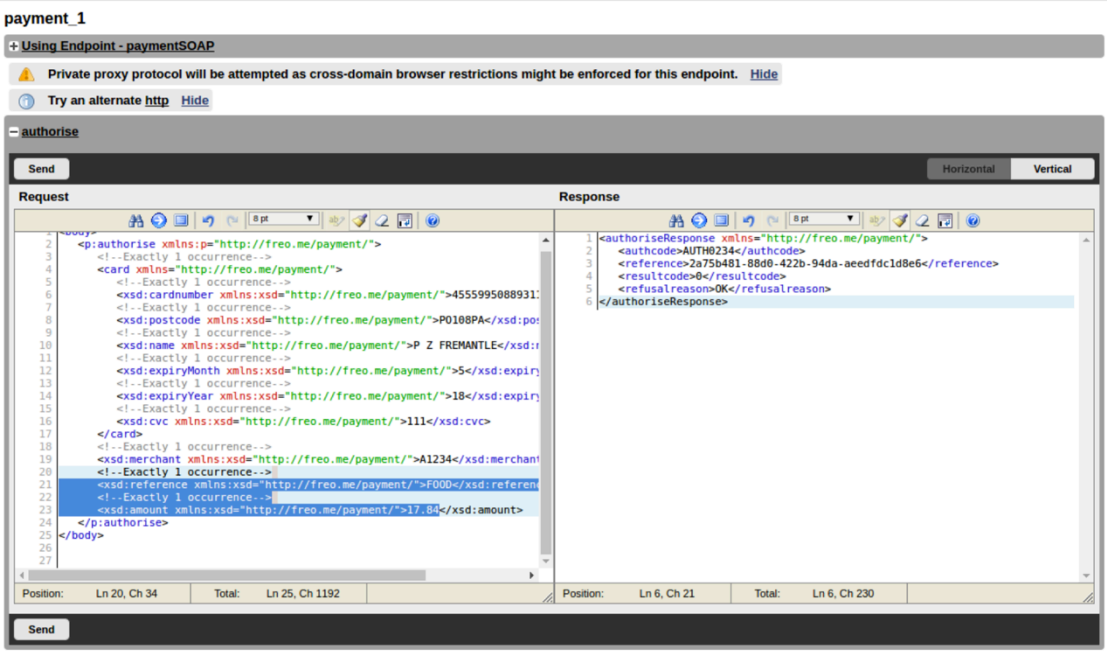
14. Close that tab and now click on the Tools tab on the far left.

15. You should see some new options:



Click on Try It
Now enter the URL of our payment service WSDL (http://localhost:8080/pay/services/paymentSOAP?wsdl) and click TryIt. Make sure that there isn’t another PopUp Blocked moment.

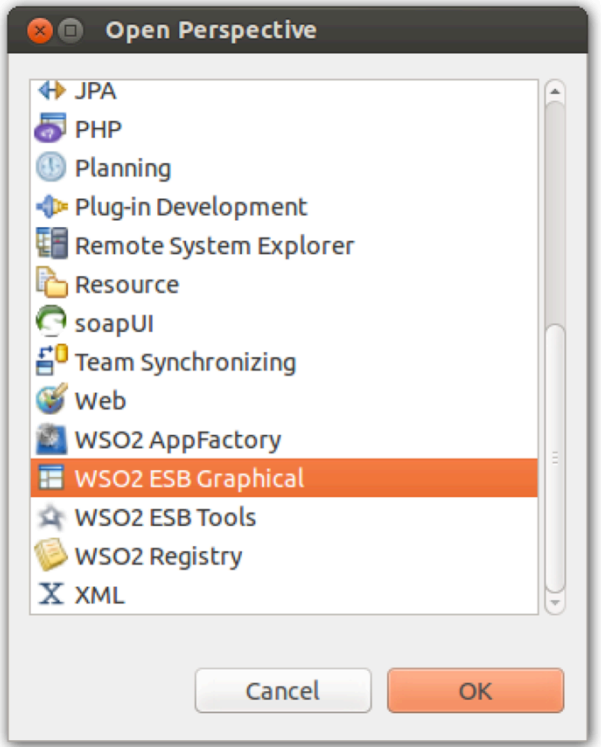
16. This is very like SOAP UI except built into the system. Try both the ping and the authorise methods and see how they work. You should see things like this:



17. Close that tab to get back to the main console.

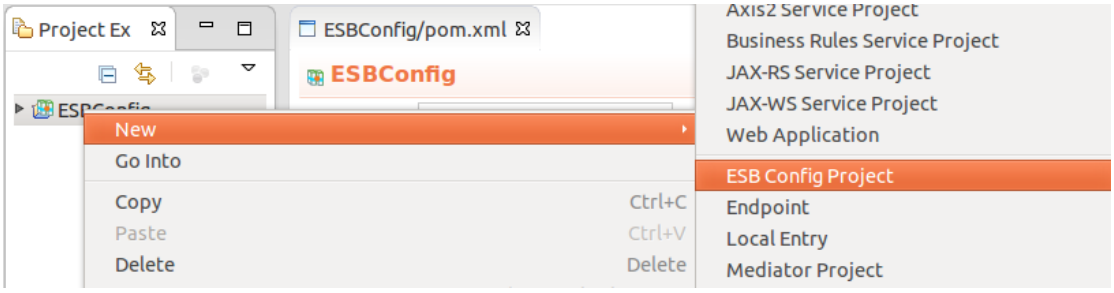
18. We are first going to create a simple RESTful API that bridges the ping service. We want the parameter to be grabbed from the URL using a URL template, and then transformed into an XML document, sent to the SOAP service, and then we will grab the response from the XML and transform into a JSON payload.

19. **In Eclipse**, first open the WSO2 ESB graphical perspective:
Window->Open Perspective->Other:

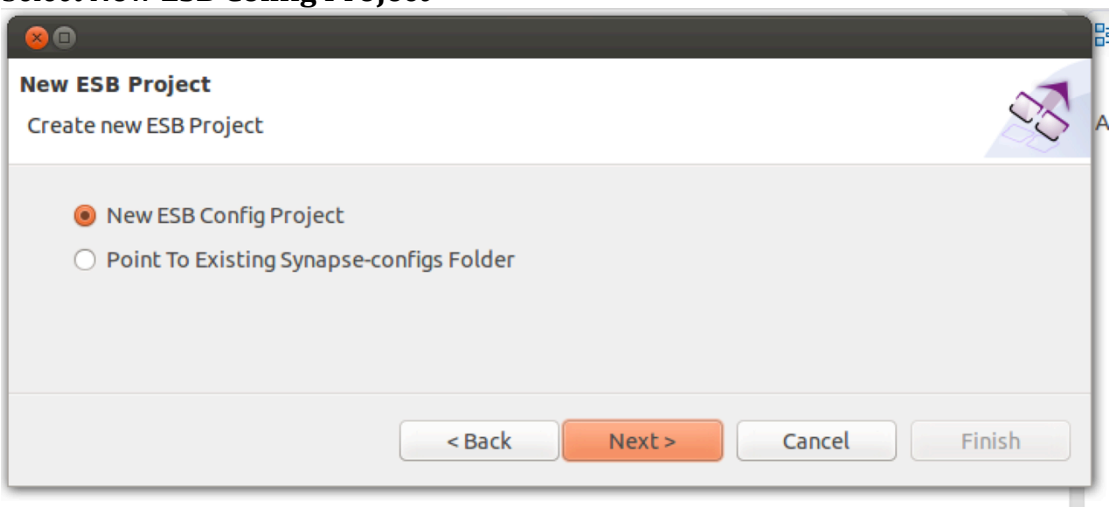


20. Now create a "Composite Application Project":
File -> New -> Composite Application Project
Call it **ESBConfig** and click **Finish**.

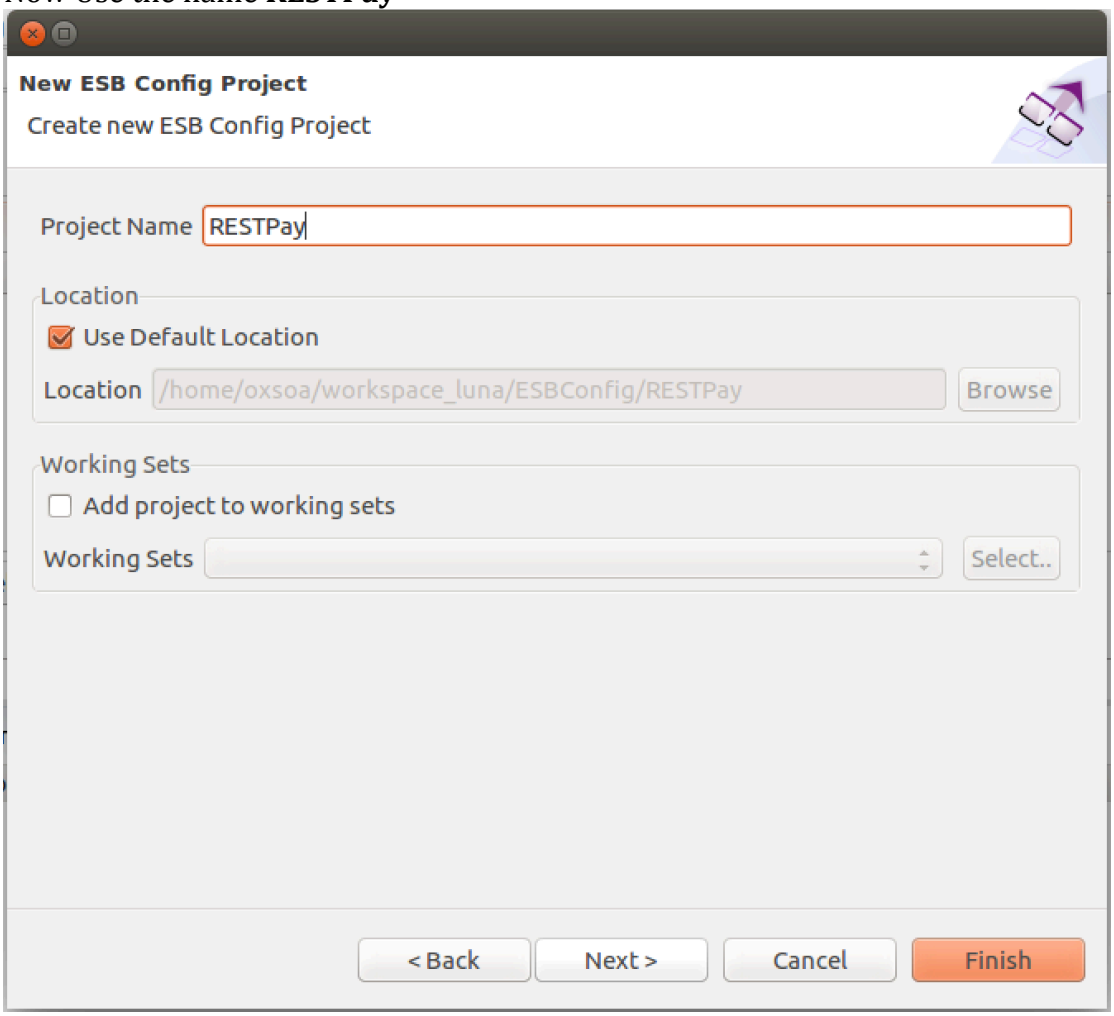
21. Now **right click on the new project** and choose New->ESB Config Project



Select **New ESB Config Project**

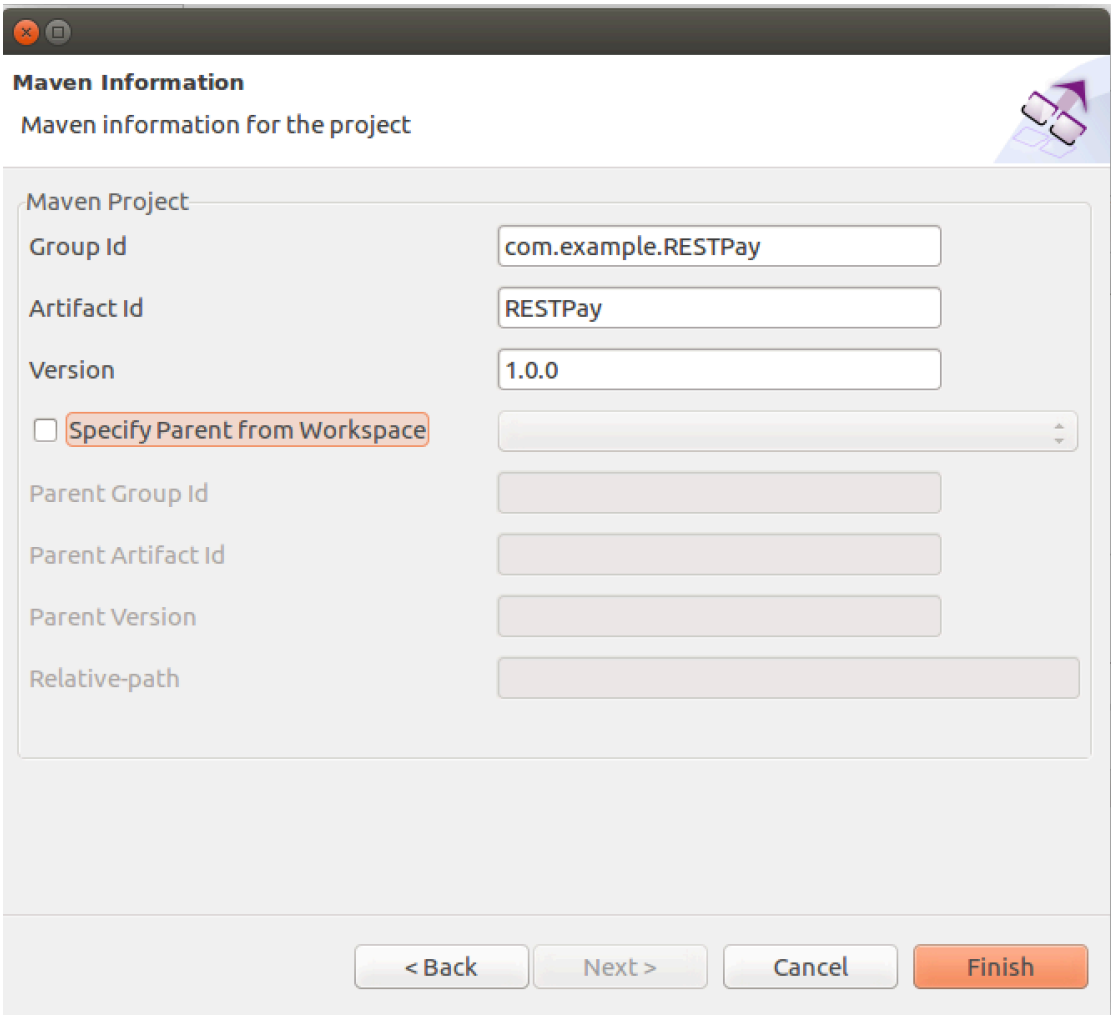


22. Now Use the name **RESTPay**

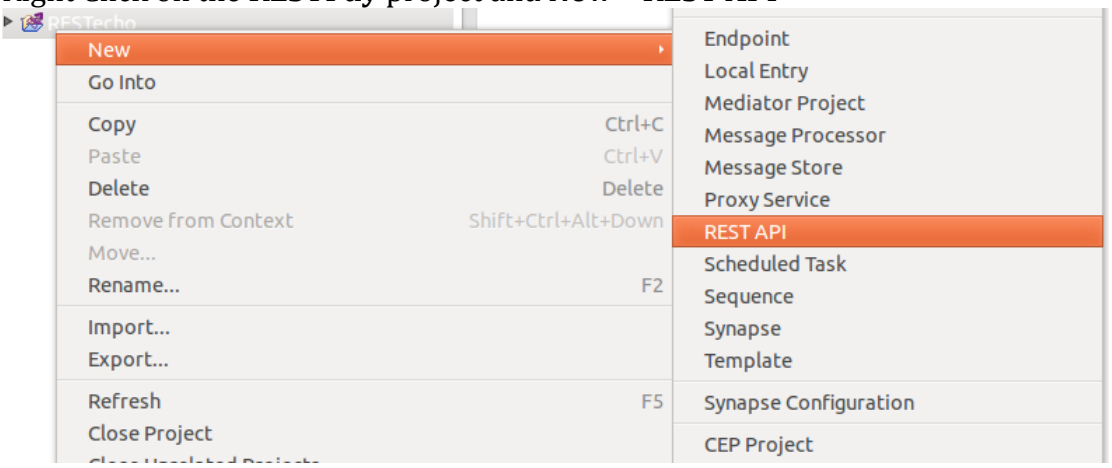


23. Click **Next**

24. Leave the Maven info the same:

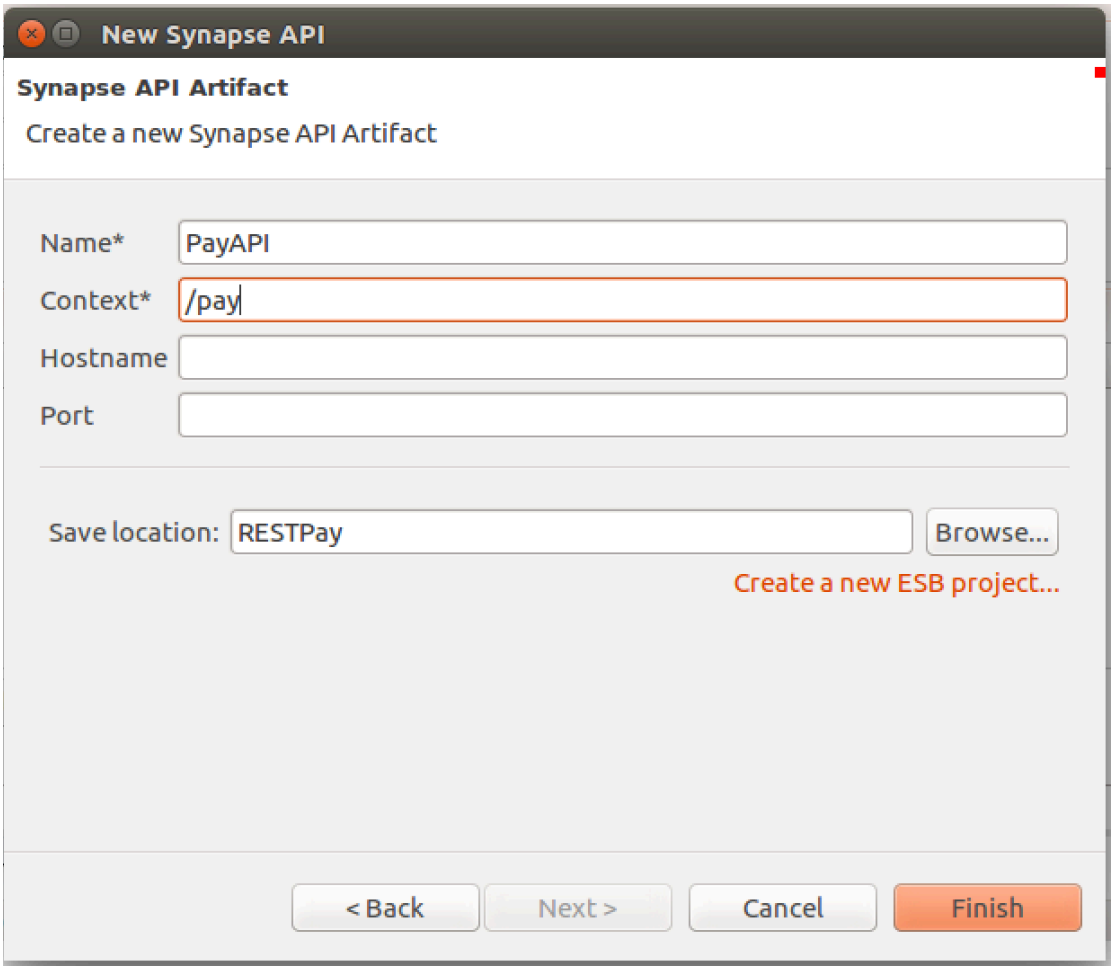


25. Right Click on the **RESTPay** project and **New->REST API**



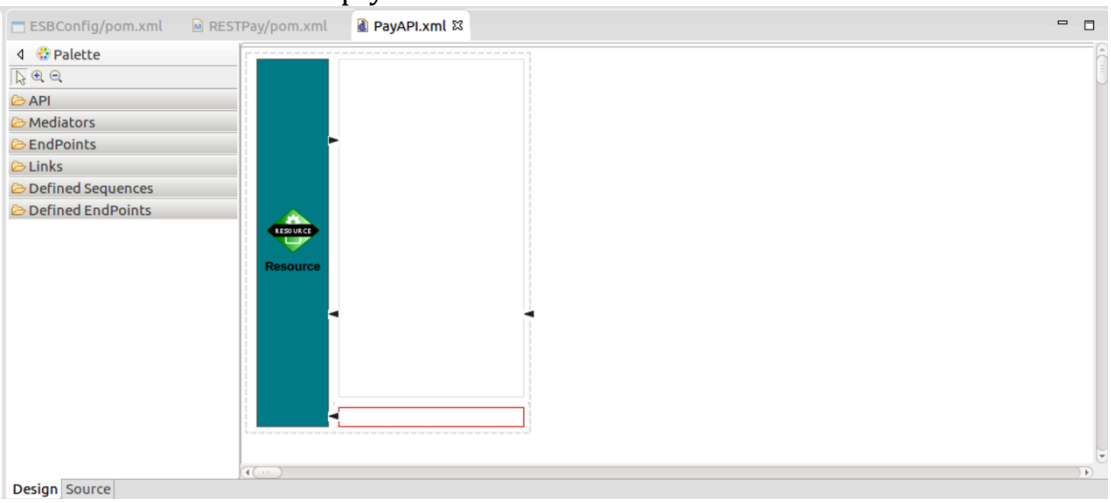
26. Select **Create a New API Artefact** and then **Next**.

27. Use:
    Name: **PayAPI**
    Context: **/pay**

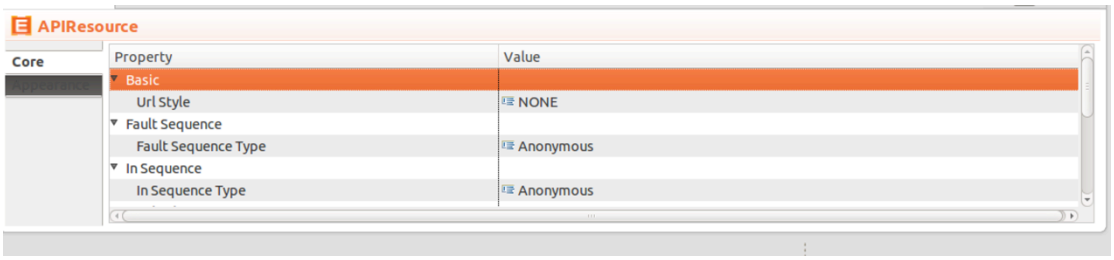


28. **Finish**

29. You should see a nice empty ESB flow like this:



30. First, we need to edit the properties of this resource. Click on the Resource icon, and look at the property editor box.



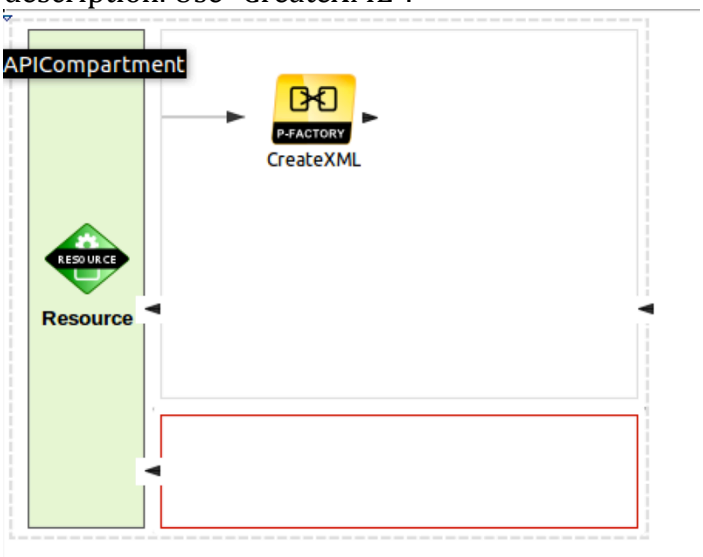
31. Change (or check) the following:

    Url Style: **URI_TEMPLATE** (then hit enter and the next box will appear)
    Uri Template: **/{input}**
    Methods / Get: **True** (should already be this).

    This has said that this is modeling a GET resource with a URI template of: http://*hostname:port/*pay/{input}

32. There are lots of ways to create an XML payload to send to the SOAP service. For example, we could use XSLT, XQuery, or JavaScript. But the simplest way is a mediator called a PayloadFactory that simply populates the body with XML or JSON, and uses a template model to fill in parameters (e.g. $1 is replaced by the first parameter).

    Now expand the Mediators box, and choose the PayloadFactory mediator and drag it over to the upper half of the flow box. It will prompt you for a description. Use "CreateXML":



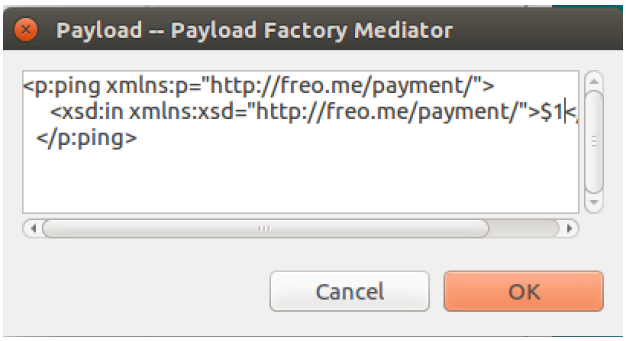
    Now edit the properties of the PayloadFactory.

    The first thing is to make the right XML. We do this by pasting in a sample XML and replacing parts of it with the input parameter from the URL.

To get the sample XML I used **SOAPUI/Try It** against the paymentSOAP service. If you want to do that, please go ahead, otherwise you can enter it from here. I replaced the ? with a $1 which will be augmented by an argument.

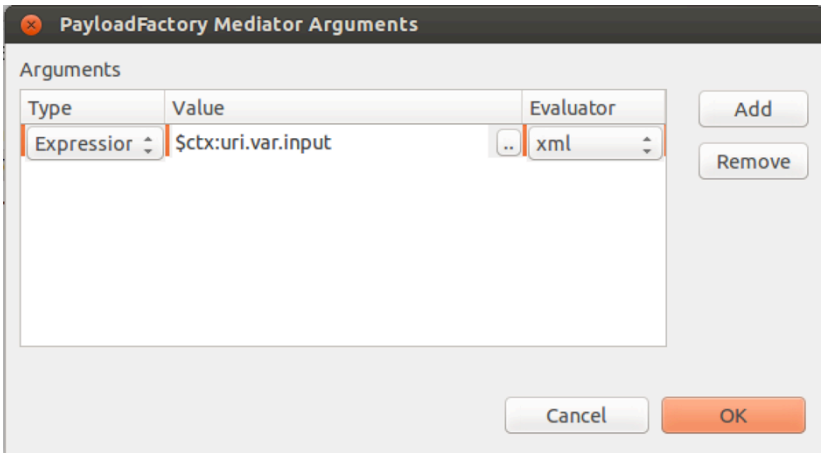
Choose Payload (hit the little button, and then replace <inline/> with:

```
<p:ping xmlns:p="http://freo.me/payment">
<in>$1</in>
</p:ping>
```



33. Now we need to grab the {input} data that came in the URI. We do this by clicking on the button by Args: 
    Then click **Add**. Change the type to **Expression,** and then click the button to edit the expression value.
    Replace **/default/expression** with **$ctx:uri.var.input**

34. It should look like:

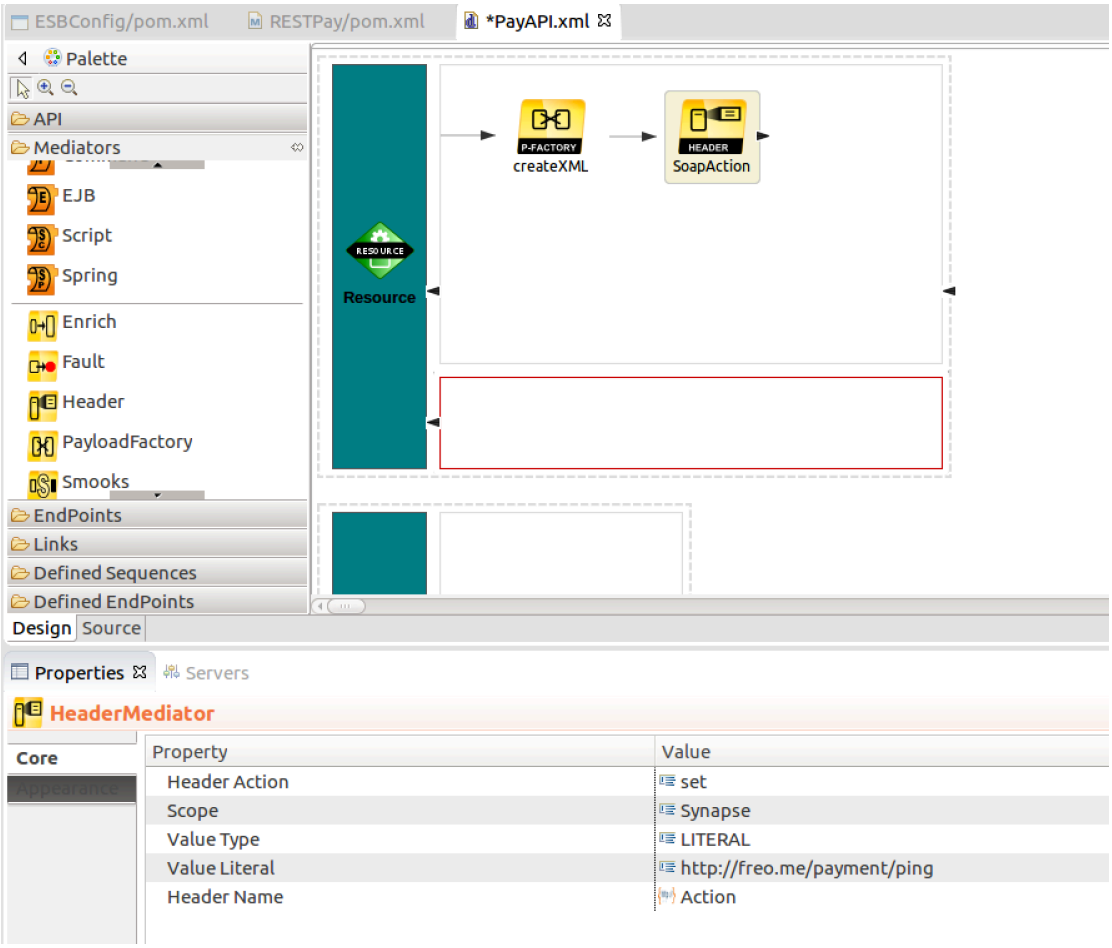


35.
36. Click **OK**
37.
38. Because we are sending the message to a SOAP service, we need a SOAP Action header. We can add that with a **Header** mediator. Grab one of those and drop it to the right of the PayloadFactory. Give it a useful

description (like **Add Soap Action**). Now set the properties as:

Value Literal: **http://freo.me/payment/ping**
Header Name: **Action**

Your screen should look similar to this:

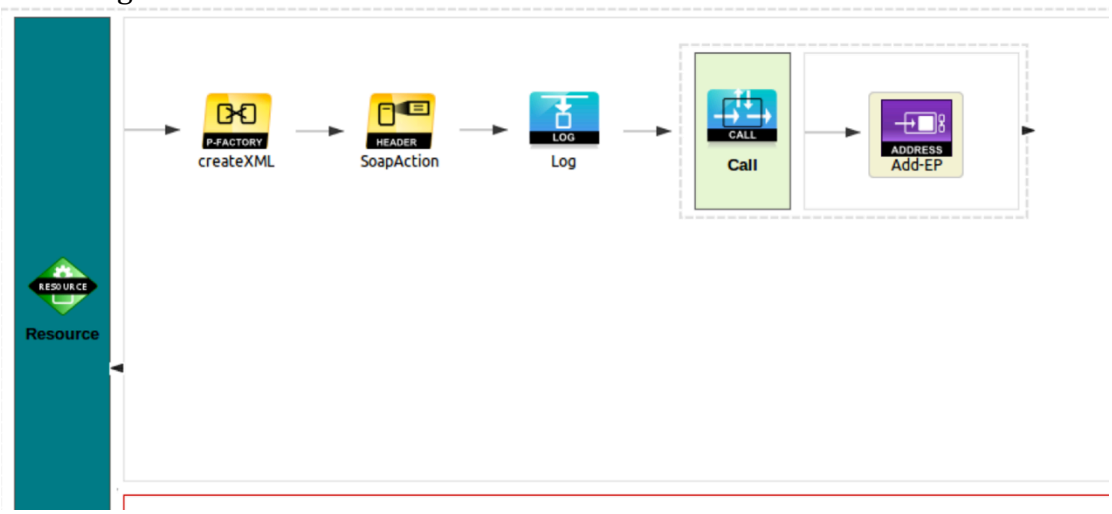


39. Now drop a Log mediator to the right, and set its log level to **FULL**.

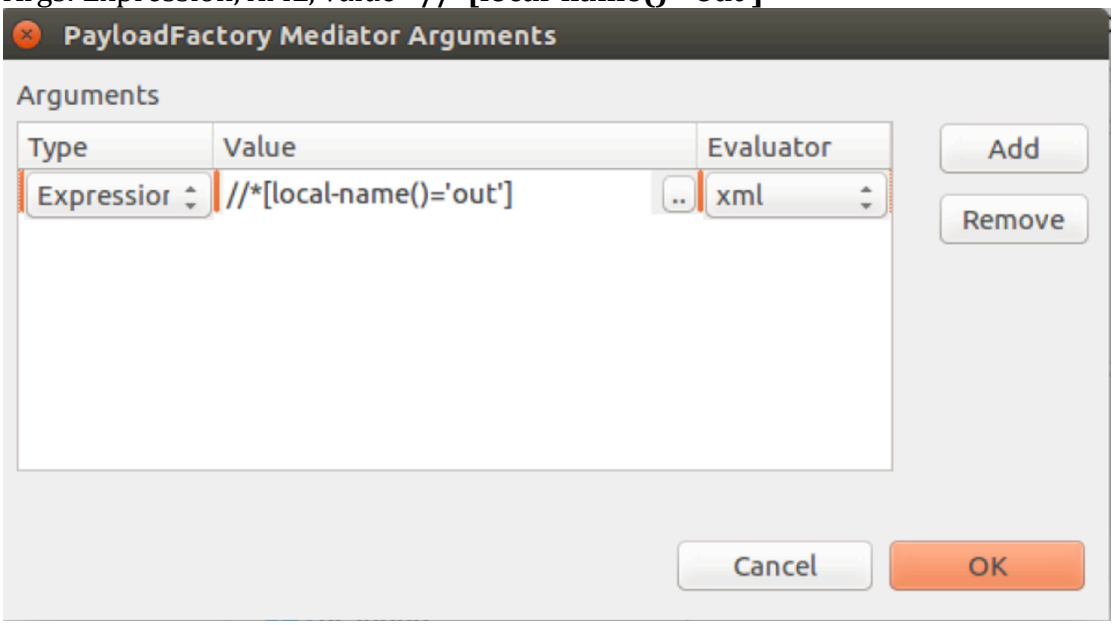
40. We are now ready to send our SOAP message to the SOAP service. Drop a Call Mediator to the right. It will have an empty box inside the mediator.
41.
42. Open the Endpoints section on the left and drop an **Address Endpoint** into the empty box. Edit the description from Add-EP to echoSOAP.
43.
44. In the properties section, under **Basic -> Format**, change the URI from http://www.example.org/service to **http://localhost:8080/pay/services/paymentSOAP**

45. Scroll down the properties until you get to the Misc->Format, and set that to be **soap11**.

Your diagram should look like:



46. After the Call/Endpoint, drop another Log mediator. Call it Logback, and again set the logging level to FULL.
47. To the right of that drop another PayloadFactory, and change its description to **toJSON**.
48. Set its properties as follows:
    Media Type: JSON
    Format: **{ "return": "$1"}**
    Args: Expression, XML, value - **//*[local-name()='out']**



49. This is an XPath expression that finds any element called <out> in any namespace and grabs the value of it.

50. Now drag a Respond Mediator right of the Payload mediator.

51. You have now created an ESB API that will:

    a. Listen at GET /pay/{input}
    b. Extract the {input} value
    c. Construct an XML message
    d. Using the input parameter
    e. Send this as SOAP11 to our server endpoint ping method
    f. Send the response back to the client as JSON

52. Before deploying this in the ESB, take a look at the XML configuration behind this configuration. Click on the Source tab (bottom left corner of the API design pane). Your XML should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<api context="/pay" name="PayAPI" xmlns="http://ws.apache.org/ns/synapse">
  <resource methods="GET" protocol="http" uri-template="/{input}">
    <inSequence>
      <payloadFactory description="createXML" media-type="xml">
        <format>
          <p:ping xmlns:p="http://freo.me/payment/">
            <p:in>$1</p:in>
          </p:ping>
        </format>
        <args>
          <arg evaluator="xml" expression="$ctx:uri.var.input"/>
        </args>
      </payloadFactory>
      <header description="SoapAction" name="Action" scope="default"
         value="http://freo.me/payment/ping"/>
      <log level="full"/>
      <call>
        <endpoint>
          <address format="soap11" trace="disable"
            uri="http://localhost:8080/pay/services/paymentSOAP"/>
        </endpoint>
      </call>
      <log description="logback" level="full"/>
      <payloadFactory description="toJSON" media-type="json">
        <format>{ return: $1 }</format>
        <args>
          <arg evaluator="xml" expression="//out"/>
        </args>
      </payloadFactory>
      <respond description="respond"/>
    </inSequence>
    <outSequence/>
    <faultSequence/>
  </resource>
</api>
```

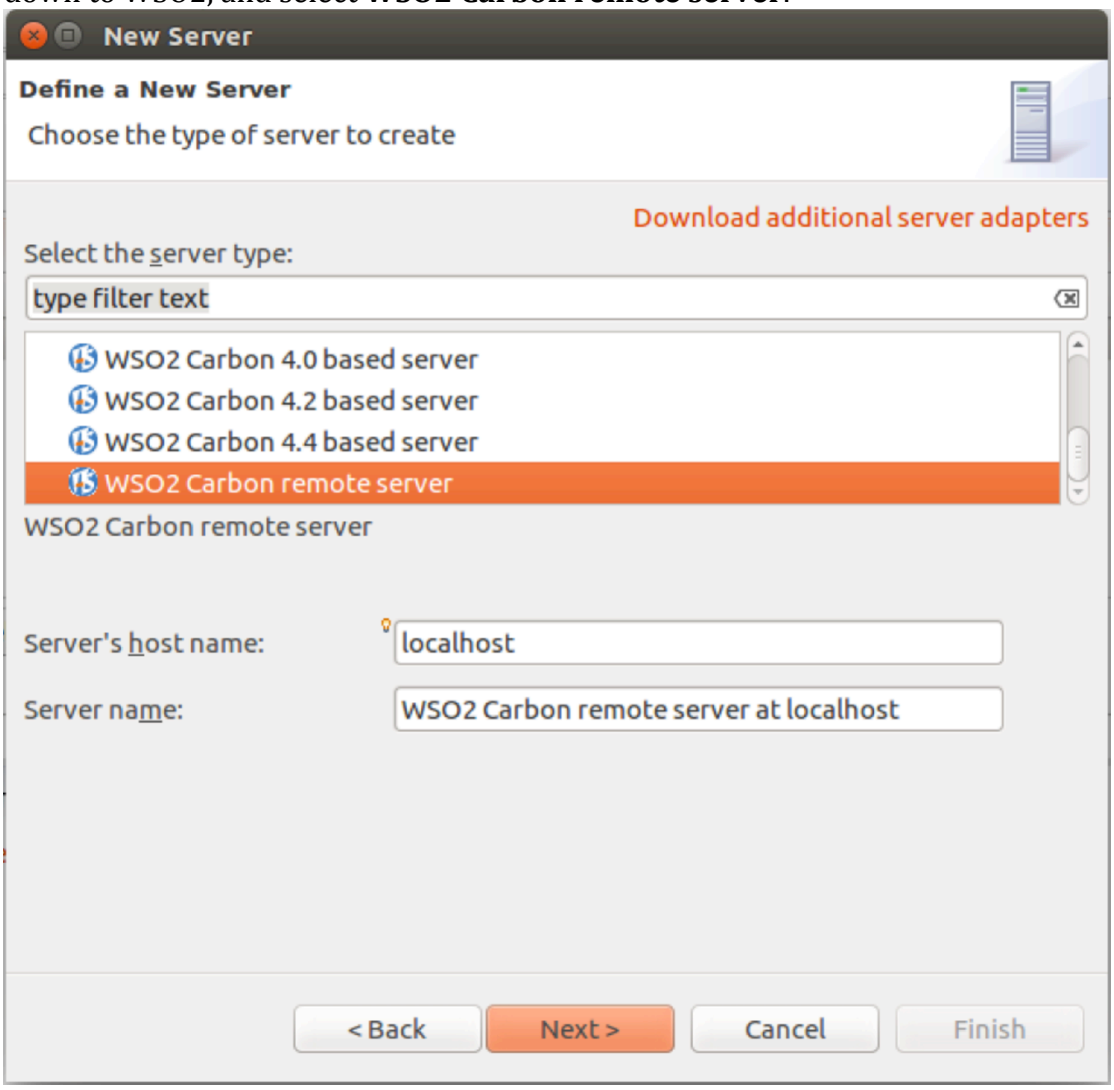
This XML is available at
https://freo.me/ex10-payapi

Firstly, we are defining an API, which is a collection of resource definitions (in the REST style). Each resource is actually implemented by a sequence of flow logic. In this case, we are looking for a GET and mapping it to a simple flow with some mediators. First we create an XML payload, then we send that to an endpoint.

53. In order to test this we need to tell the Eclipse environment about our ESB server.

54. You can check the server is running (in a minute) by browsing https://localhost:9444/. You will need to Proceed past the security warning because by default the server is using a self-signed certificate. The default credentials are **admin/admin.**

55. To add this server to Eclipse, do File->New->Other->Server. Then scroll down to WSO2, and select **WSO2 Carbon remote server**.

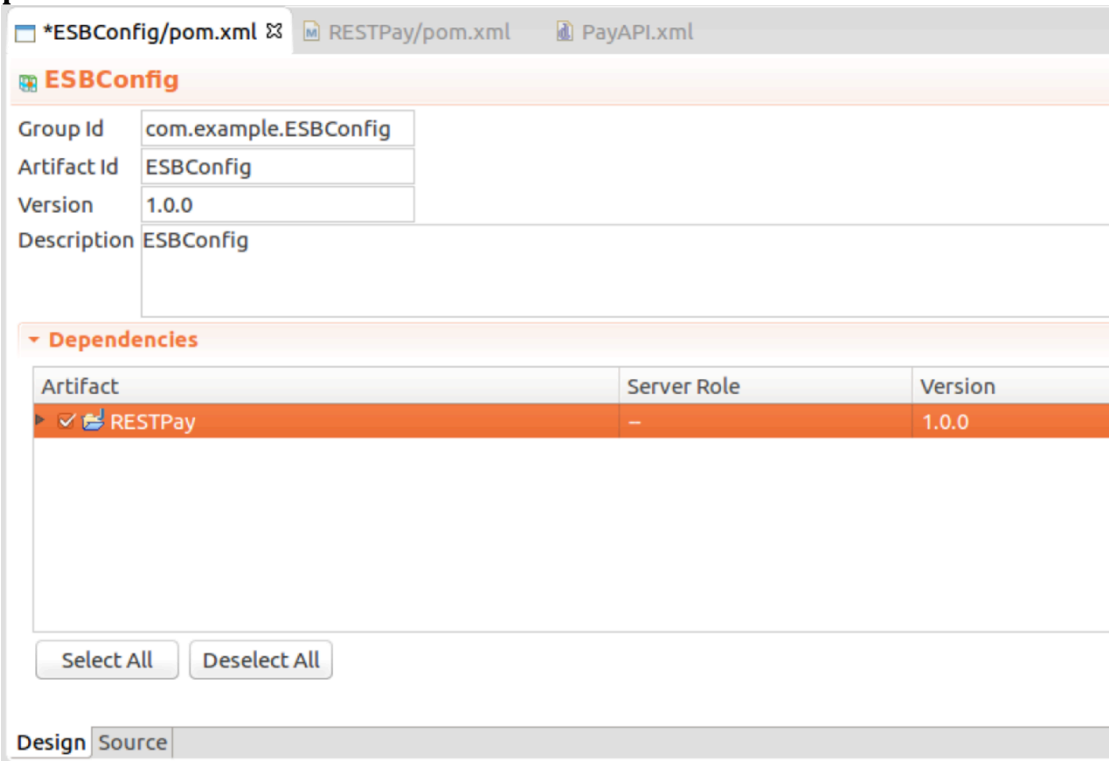


56. Click **Next**.

57. Set the servers URL to be https://localhost:9444/. Test the connection and the credentials. Click **Finish**.

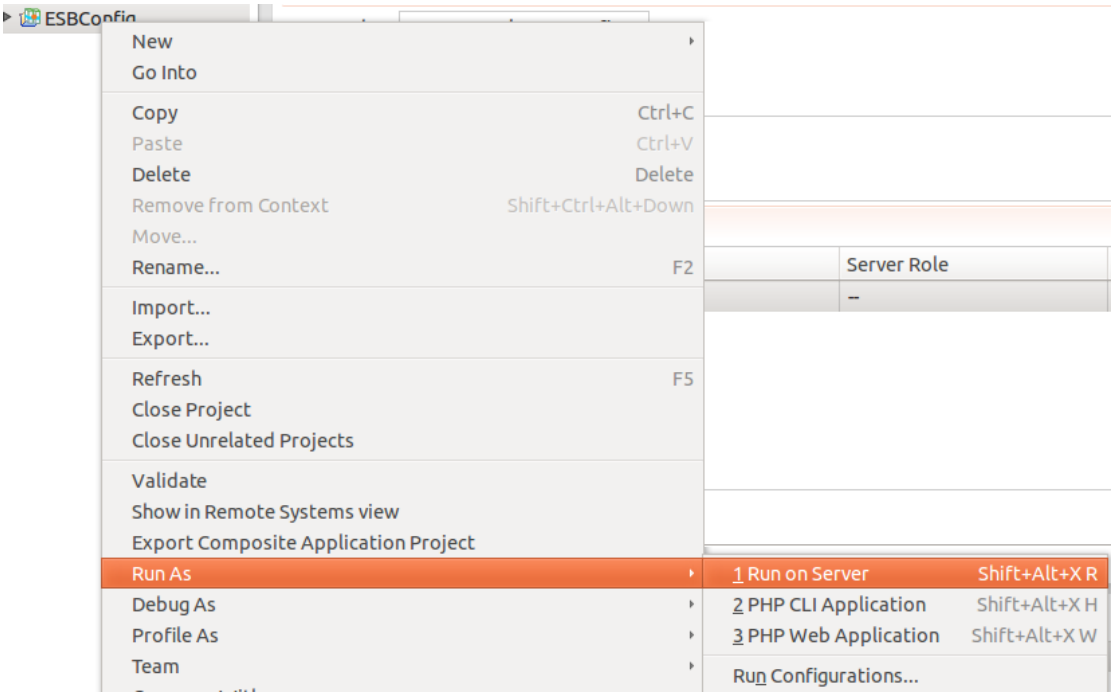
58. You need to make sure the RESTecho ESB config is part of the Composite Application Project. Open up the ESBConfig project and it will open the

pom.xml. Make sure the RESTecho Artifact is checked:

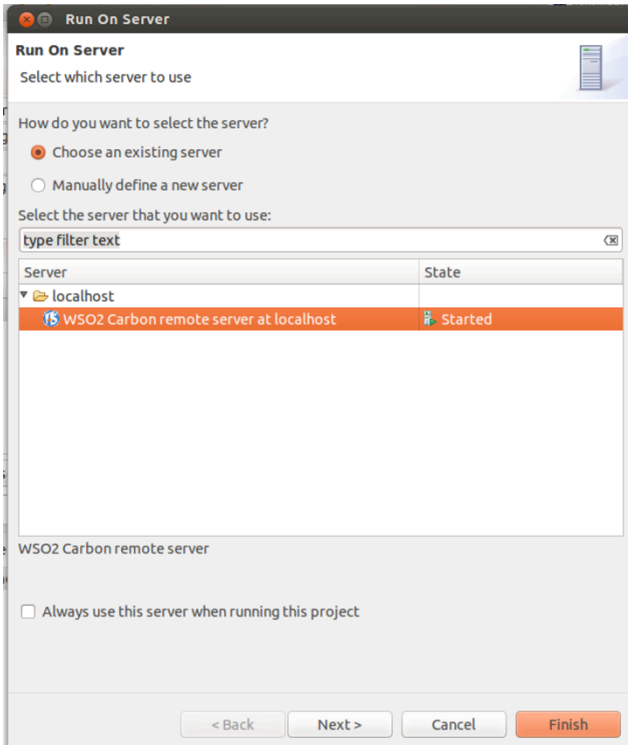


SAVE

59. Now you should be able to run the ESBConfig project on the server. *You may need to restart Eclipse.* To do this right-click on the ESBConfig project and choose **Run As->Run on Server**

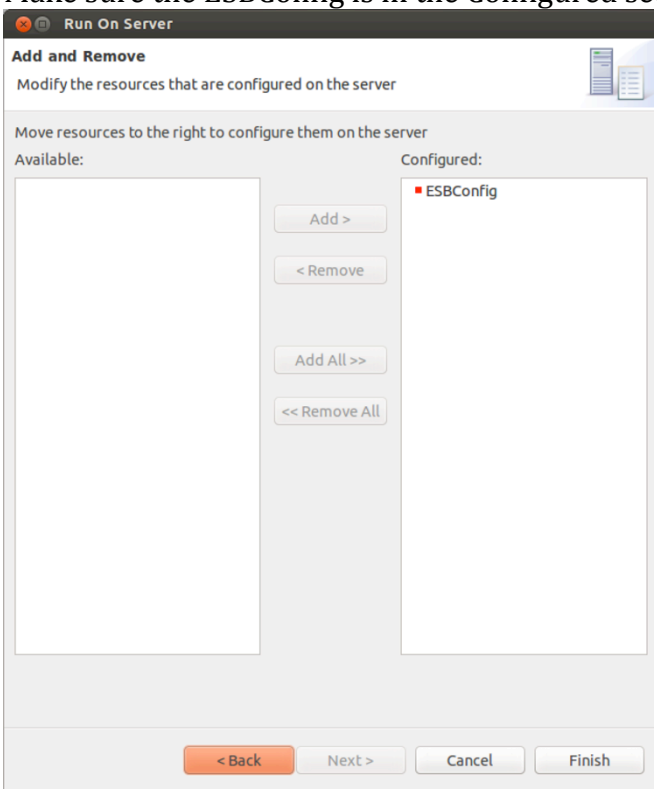


60. Make sure the Carbon Server is selected:



Click Next

61. Make sure the ESBConfig is in the Configured section:

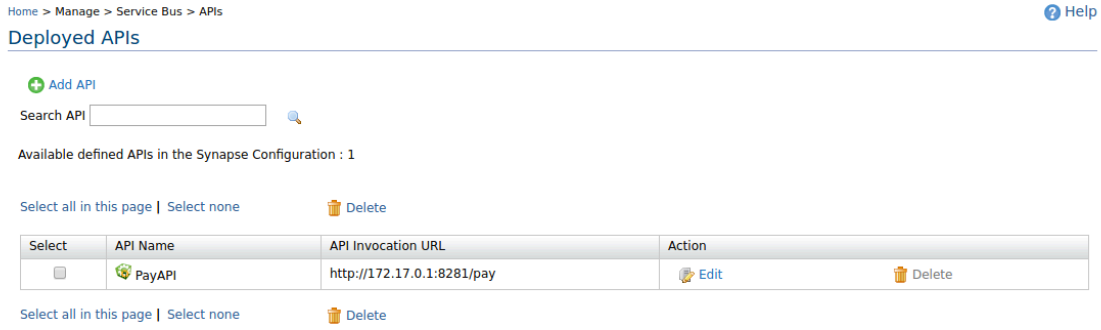


62. Click **Finish**.

63. You might need to wait a second to see it deployed. If you look at your ESB terminal window where the server is running you should see something like:

```
[2016-06-03 18:54:09,090]  INFO - CarbonAuthenticationUtil 'admin@carbon.super [-1234]' logged in at [2016-06-03 18:54
:09,090+0100]
[2016-06-03 18:54:11,502]  INFO - ApplicationManager Deploying Carbon Application : ESBConfig_1.0.0.car...
[2016-06-03 18:54:11,548]  INFO - API Initializing API: PayAPI
[2016-06-03 18:54:11,550]  INFO - APIDeployer API named 'PayAPI' has been deployed from file : /home/oxsoa/servers/wso
2esb-4.9.0/tmp/carbonapps/-1234/1464976451503ESBConfig_1.0.0.car/PayAPI_1.0.0/PayAPI-1.0.0.xml
[2016-06-03 18:54:11,550]  INFO - ApplicationManager Successfully Deployed Carbon Application : ESBConfig_1.0.0 {super
-tenant}
```

64. You can browse to the admin console and see if an API is visible in the API section.



65. Now try the API by browsing http://localhost:8281/pay/test
You should see something like:

66. Check the ESB terminal window and you should see the log messages from the log mediators:

```
[2016-06-03 19:19:40,919]  INFO - LogMediator To: /pay/test, WSAction: http://freo.me/payment/ping, SOAPAction: http:/
/freo.me/payment/ping, MessageID: urn:uuid:164b0197-2962-4d37-a413-aae8ec8feb46, Direction: request, Envelope: <?xml v
ersion='1.0' encoding='utf-8'?><soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"><soapenv:Body
><p:ping xmlns:p="http://freo.me/payment/"><p:in>test</p:in></p:ping></soapenv:Body></soapenv:Envelope>
[2016-06-03 19:19:40,989]  INFO - LogMediator To: http://www.w3.org/2005/08/addressing/anonymous, WSAction: , SOAPActi
on: , MessageID: urn:uuid:7e928fa1-d062-4eb6-a5db-08f53b8d1daa, Direction: request, Envelope: <?xml version='1.0' enco
ding='utf-8'?><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><pingResponse xmlns="ht
tp://freo.me/payment/"><out>test</out></pingResponse></soap:Body></soap:Envelope>
```

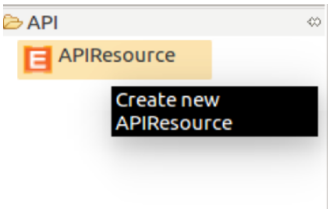
67. Also check out the mitmdump window.

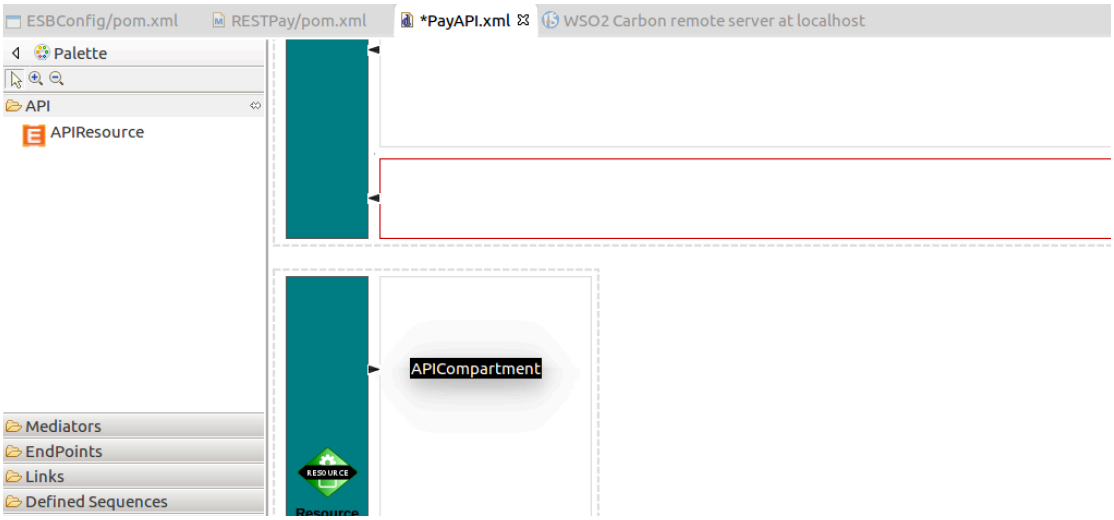
    I know that seemed like a lot of bother to get to a simple end, but in fact, we have now set up to do the next stage!

68. Ok. So that was the ping service! But we really want to do is the actual Payment Service.

69. Go back to the Eclipse editor.

70. In the same API, add a new resource underneath the GET resource. Drag it to underneath.





71. Set up the properties on the resource:
    URL style: **NONE**
    Methods: **POST - True**

Property	Value
Basic	
Url Style	NONE
Protocol	http
Fault Sequence	
Fault Sequence Type	Anonymous
In Sequence	
In Sequence Type	Anonymous
Methods	
Get	false
Post	true
Put	false
Delete	false
Options	false
Head	false
Patch	false
Out Sequence	
Out Sequence Type	Anonymous

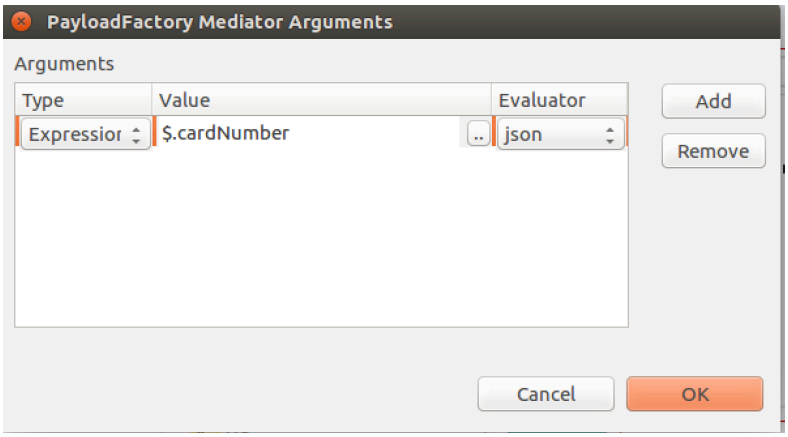
72. We need the same flow of mediators:
    PayloadFactory -> Header -> Log -> Call (Address) -> Log ->
    PayLoadFactory -> Respond

73. Basically, do the same unless it is different! If you prefer XML, once you have created the API Resource, you could cut and paste the XML and make the right edits to that. Or you can use the GUI.

74. The differences this time are:
    First PayloadFactory:
    Firstly, the format of the XML is much more complex!

```
<p:authorise xmlns:p="http://freo.me/payment/">
      <p:card>
            <p:cardnumber>$1</p:cardnumber>
            <p:postcode>$2</p:postcode>
            <p:name>$3</p:name>
            <p:expiryMonth>$4</p:expiryMonth>
            <p:expiryYear>$5</p:expiryYear>
            <p:cvc>$6</p:cvc>
      </p:card>
      <p:merchant>$7</p:merchant>
      <p:reference>$8</p:reference>
      <p:amount>$9</p:amount>
</p:authorise>
```

Secondly, the arguments are not extracted from the URI, but from the JSON body of the message, using JSONPath. These are defined like this:



Here is the proposed incoming JSON:

```
{
  "cardNumber": "4544950403888999",
  "postcode": "PO107XA",
  "name": "P Z FREMANTLE",
  "month": 6,
  "year": 2017,
  "cvc": "999",
  "merchant": "A0001",
  "reference": "test",
  "amount": 11.11
}
```

Hint: You might prefer to edit the XML source of the API design rather than use the UI as its quicker! I quite often create one argument with the GUI and then cut and paste the XML.

The Header mediator needs a different SOAPAction. It should be: http://freo.me/payment/authorise

The address endpoint is the same

The JSON response format in the second PayloadFactory should be adjusted to something useful like:
{"authcode": "$1" , "reference":"$2", "refusalreason": "$3" }

The same model for extracting the responses from the XML SOAP response will work for the results, but we need to extract three arguments. In XML this looks like:

```
<arg evaluator="xml" expression="//*[local-name()='authcode']"/>
<arg evaluator="xml" expression="//*[local-name()='reference']"/>
<arg evaluator="xml" expression="//*[local-name()='refusalreason']"/>
```

75. Your overall XML should look like this Gist here:
    https://freo.me/fullpayapi

76. You can try the API out by sending a JSON POST using Advanced Rest Client.
    The JSON to try is here:
    https://freo.me/cardjson

77. There are lots of ESB samples you can look at here:
    http://docs.wso2.org/display/ESB490/Samples

78. Extension:
    What should the HTTP return code be if the payment is not authorised by the payment service? Can you find a way of acting on the returncode value of the response to produce different flows in the ESB?

79. Big extension!
    The WSO2 ESB stores its configurations in <ESB>/repository/*

    See if you can make a docker build for the ESB and a docker compose that joins the payment XML and the ESB into a single easy deployment.