Exercise 8

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 7
- Apache Maven 3.0.4 or later
- WSO2 Developer Studio 3.3.0
- WSO2 App Server 5.2.0
- WSO2 ESB 4.8.0

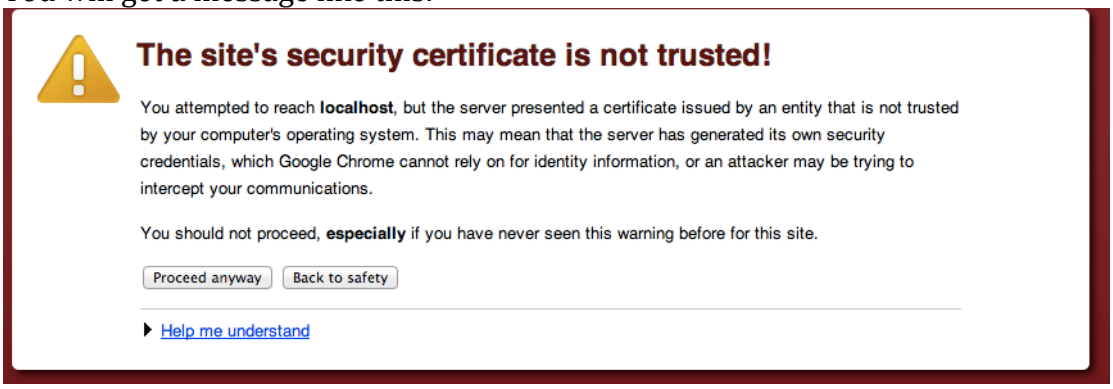
1. Before we install the ESB we need to host some services to interact with. To do this we are going to use the WSO2 Application Server.

2. cd ~/servers/wso2as-5.2.0/

3. Start up the server:
   `bin/wso2server.sh`

4. In Chrome, browse to
   https://localhost:9443 [Note the SSL URL]

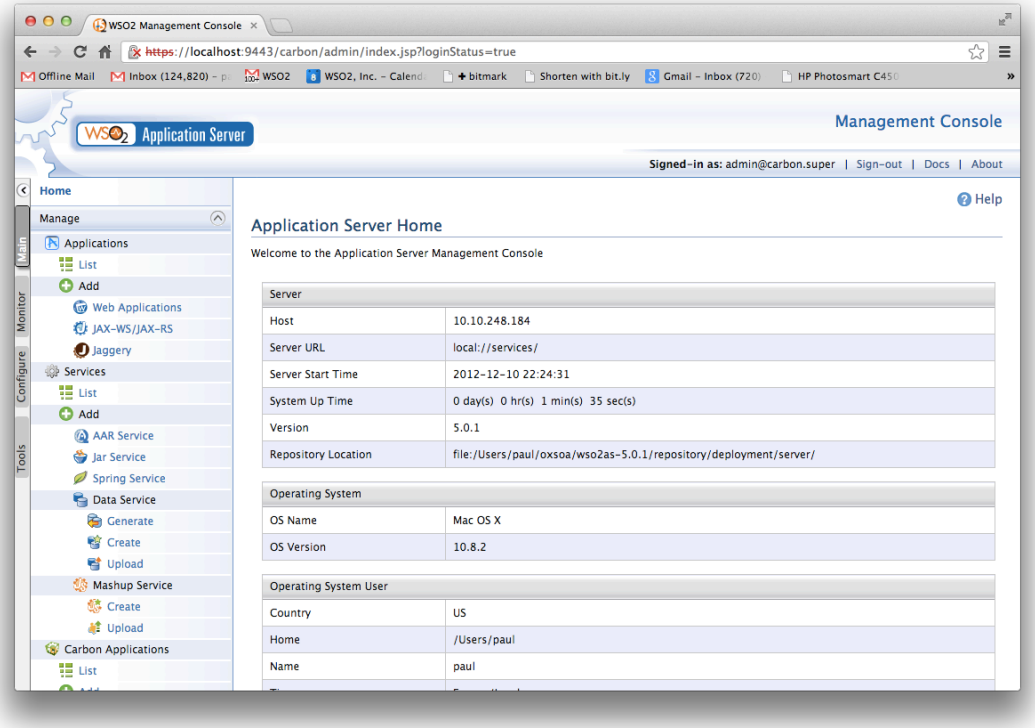
   You will get a message like this:



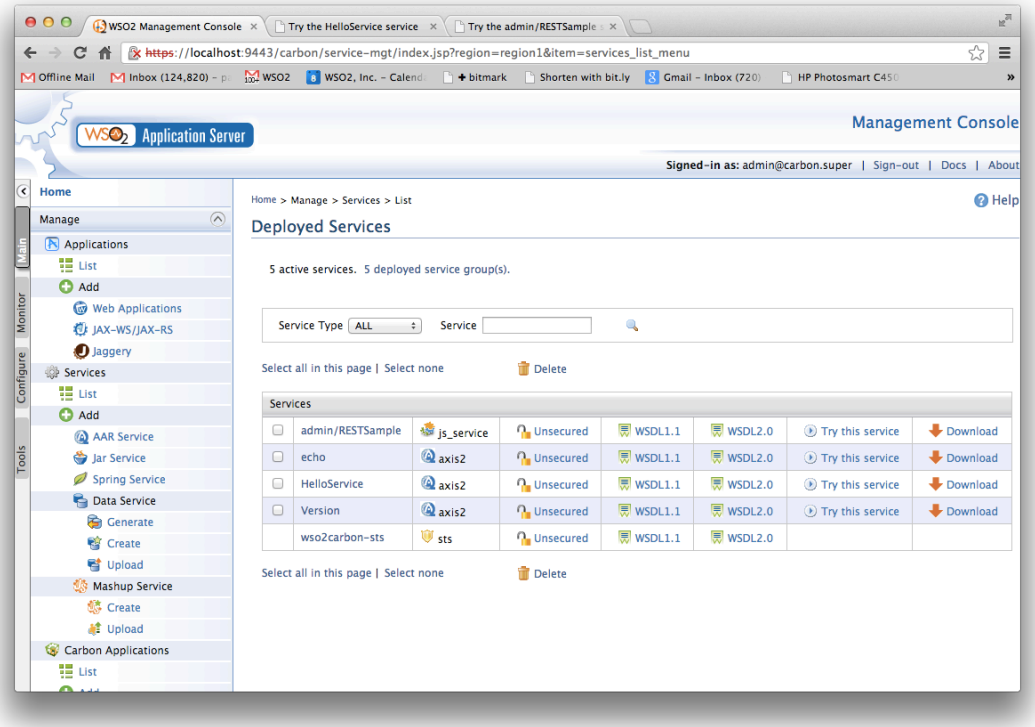
   This is because we haven't installed a "proper" certificate into the server. Click **Proceed Anyway**

5. Use **admin/admin** as username and password. Click **Remember Me** and sign in:

6. You should see a Web Console like this:

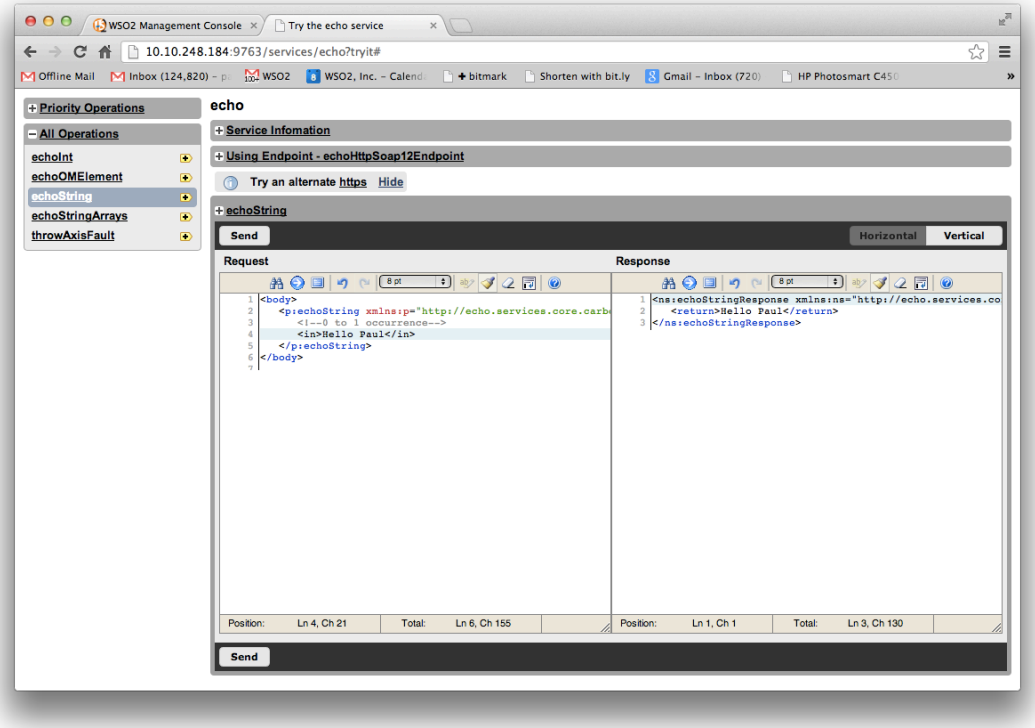


7. Click on **Services->List** in the left hand menu. You should see something like this:



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

9. You will see a “test” client.



10. 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.

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

12. The Starbucks Outlet Service should be in your Downloads directory. [If not then download the Starbucks Outlet Service:
Browse to http://freo.me/UtN4Mj]

13. Click on **Services/Add/AAR Service.**

14. Click on **Choose File**. Browse to the saved StarbucksOutletService.aar This is an Axis2 Web Service application. Click **Upload**

15. Wait a minute for this to deploy, then Click on **Services/List** again.

16. Now Click on **Try this service** for the StarbucksOutletService.

17. Create an order, list orders, pay for it, etc. Get a feel for the SOAP API.

18. Go to the Monitor tab and look at some stats, logs, etc.

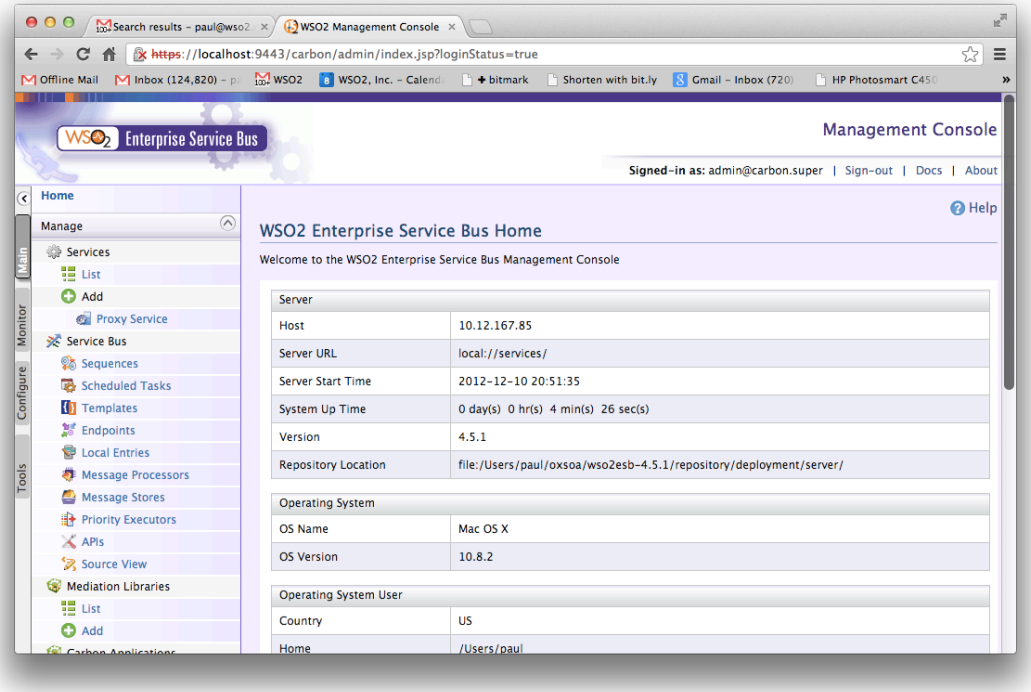
19. Now start a new Terminal window
`cd ~servers/wso2esb-4.8.0`

20. Start the server up
    `bin/wso2server.sh`

21. In Chrome, browse to
    https://localhost:9444 [Note the SSL URL]

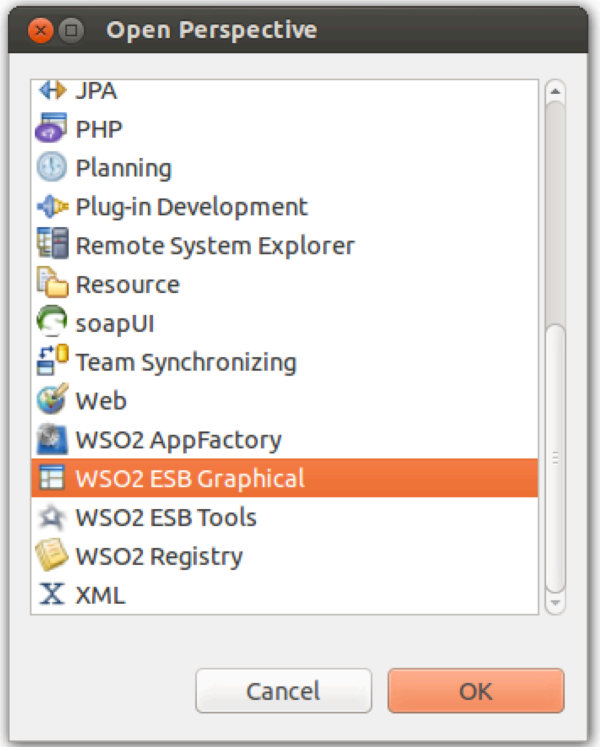
    This is one port higher. By default the ESB and AppServer share the same port (which means you can't run them on the same machine). To solve this we changed the default ports by editing the offset parameter in the ESB's repository/conf/carbon.xml file.

22. Login as before. You should see a similar (but different) console. Compare to the App Server screens.



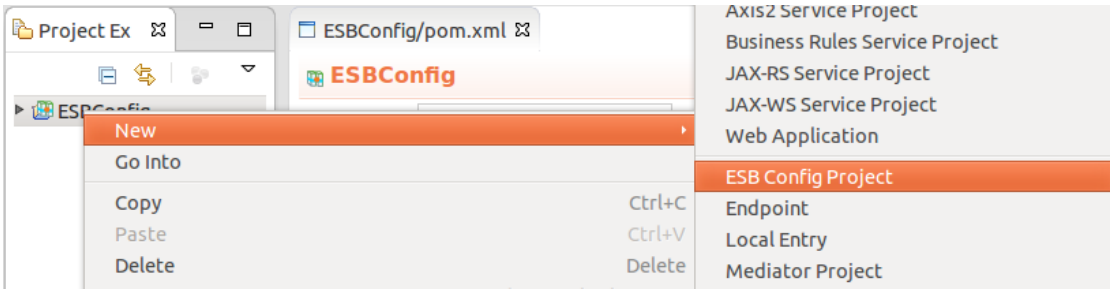
23. We are going to create a simple RESTful API that bridges the App Server's echo service. We want the parameter to be grabbed from the URL using a URL template, and then transformed into an XML document, sent to the SOAP service, and then we will grab the response from the XML and transform into a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

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

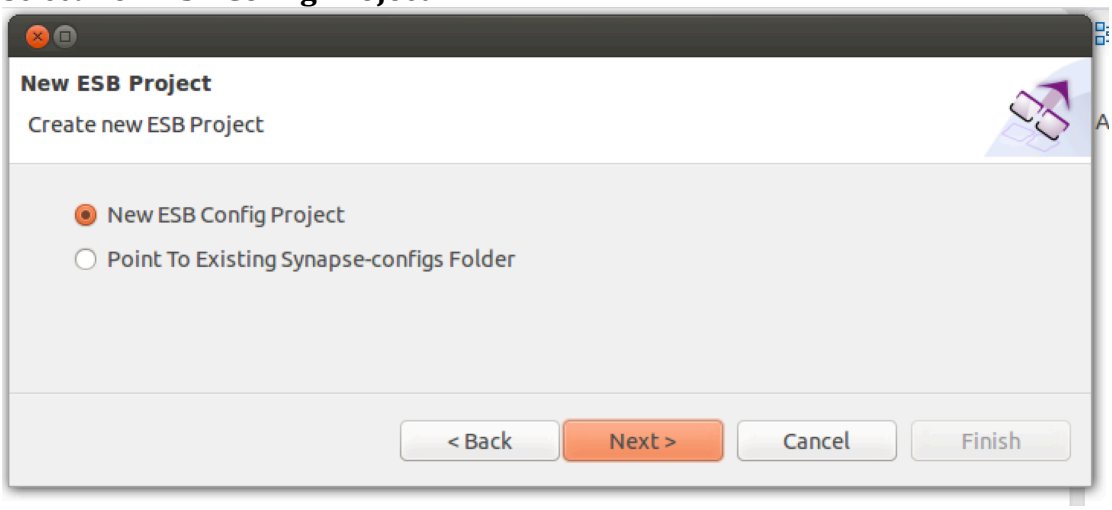


25. Now create a “Composite Application Project”:
    File -> New -> Composite Application Project
    Call it **ESBConfig** and click **Finish**.

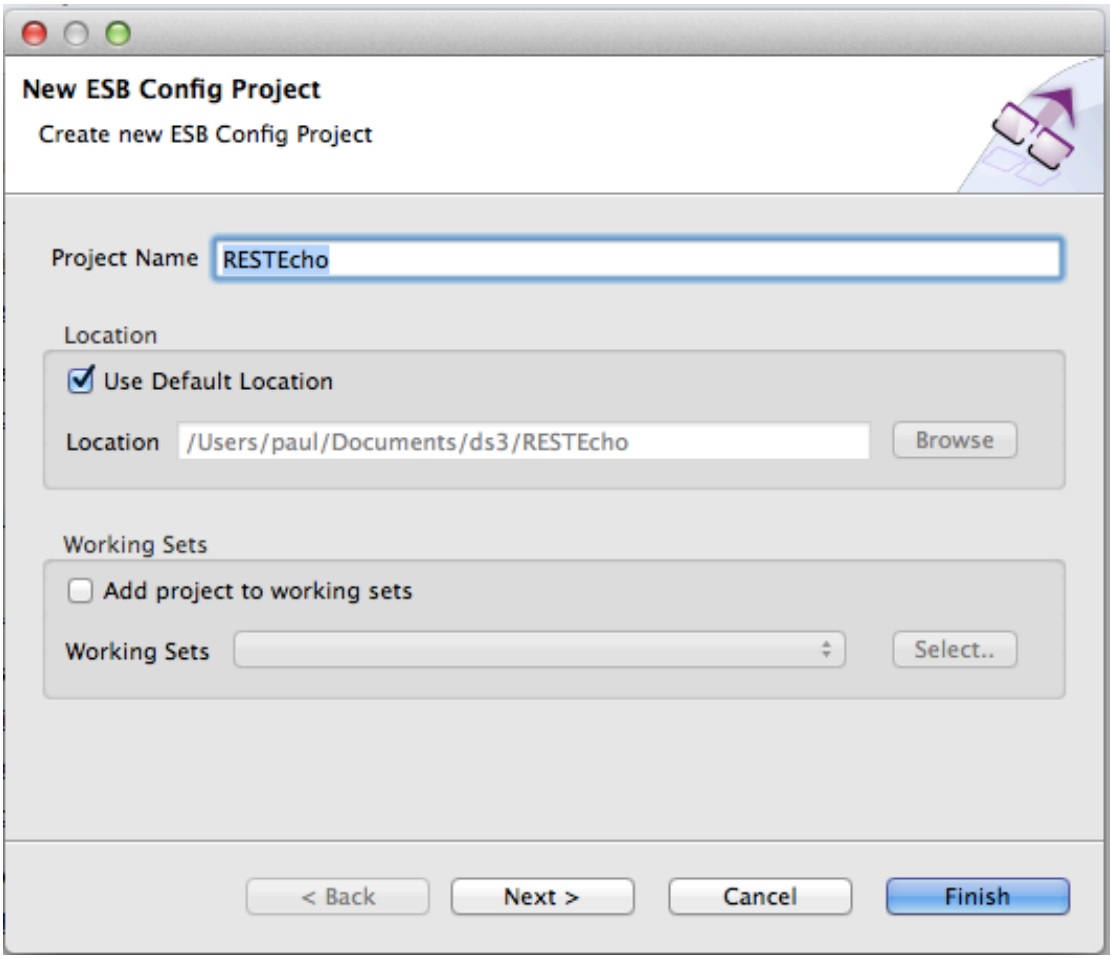
26. Now right click on the new project and choose New->ESB Config Project



Select **New ESB Config Project**

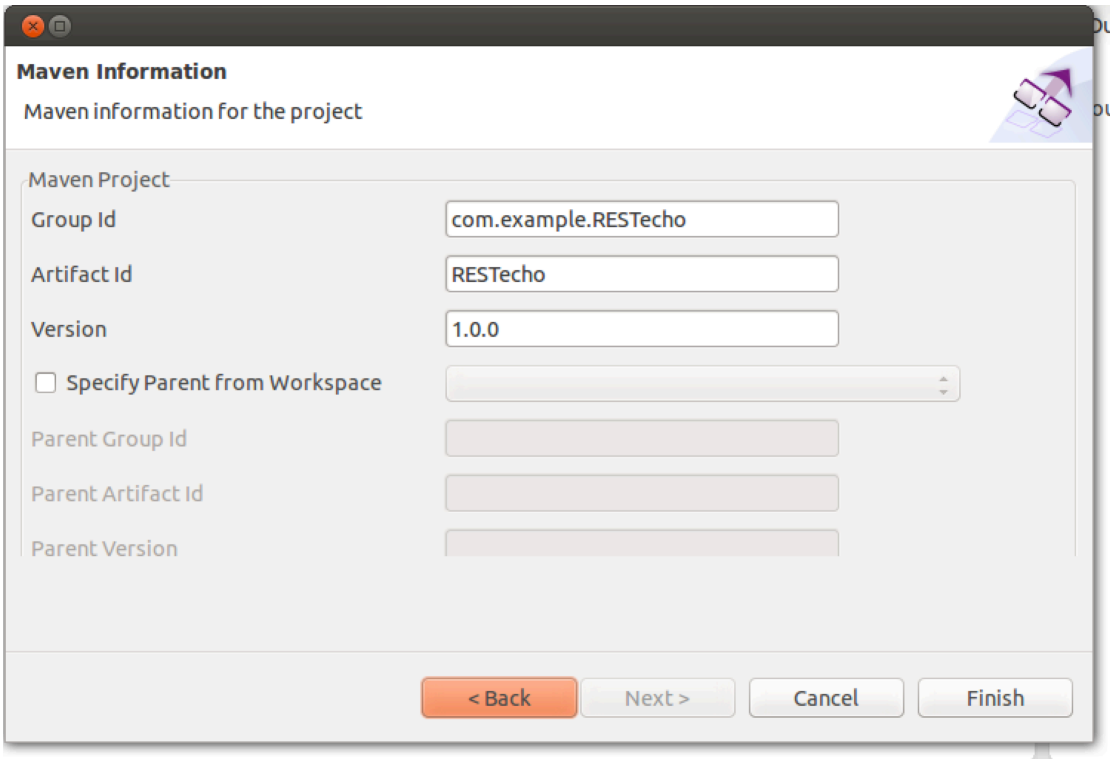


27. Now Use the name **RESTEcho**



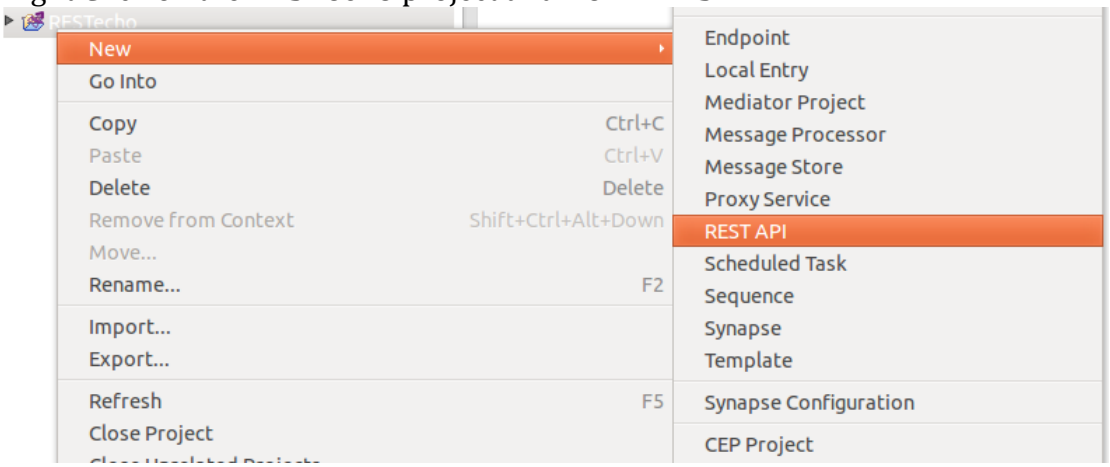
28. Click **Next**

29. Leave the Maven info the same:



30. Click Finish

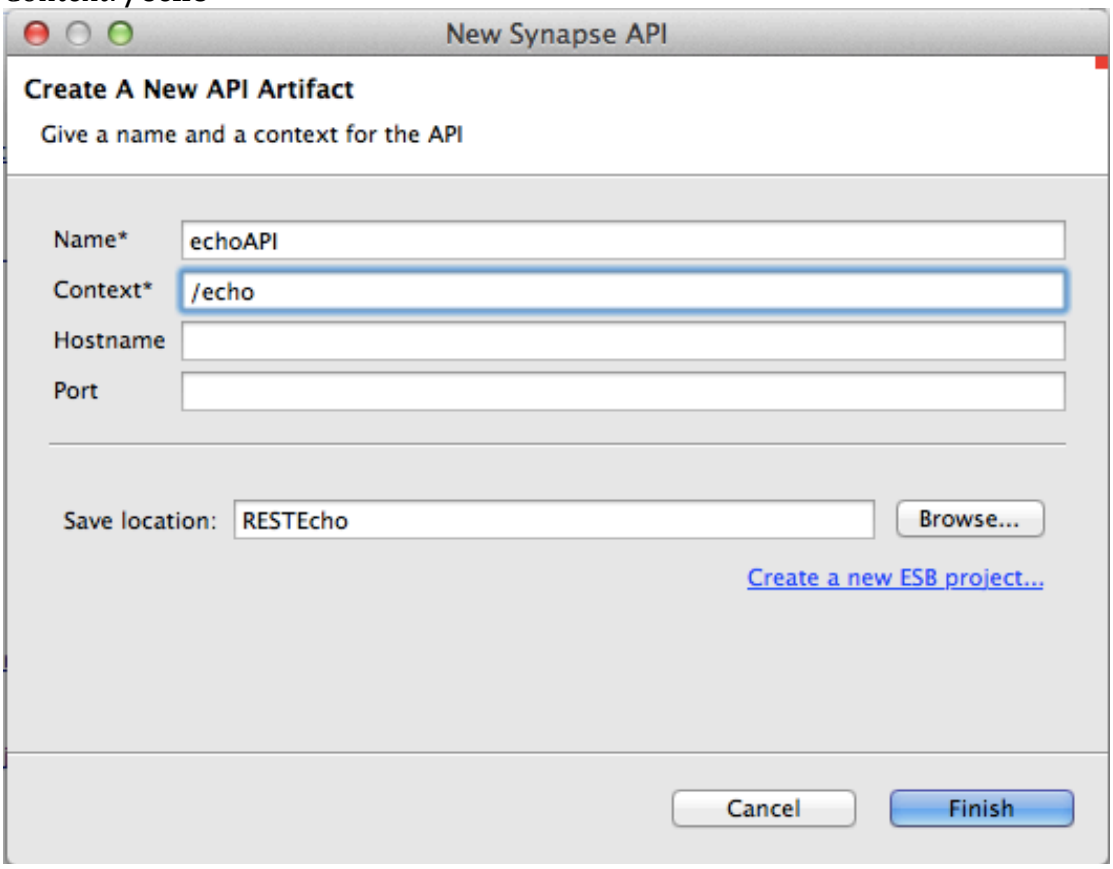
31. Right Click on the **RESTecho** project and **New->REST API**



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

33. Use:

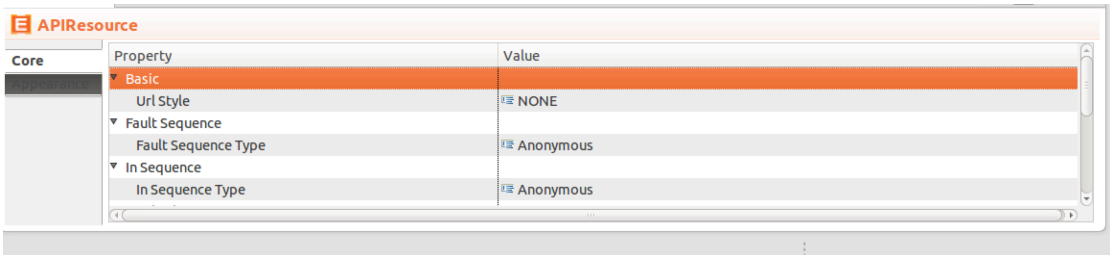
    Name: **echoAPI**
    Context: **/echo**



34. **Finish**

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

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



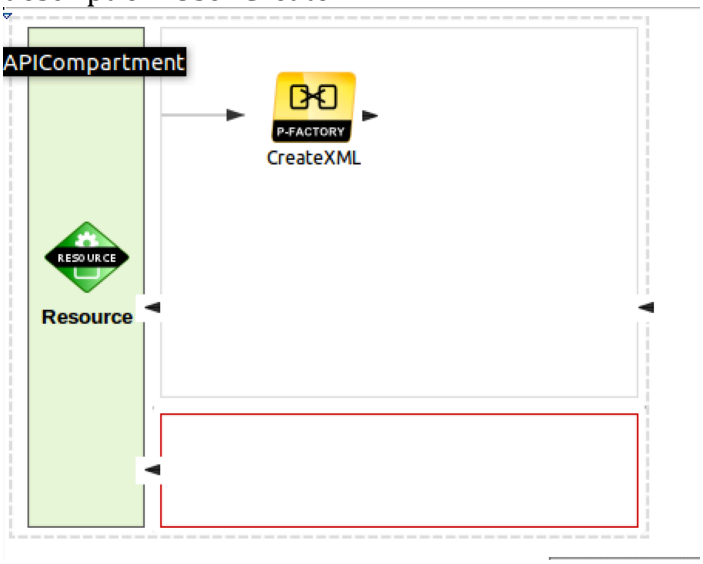
37. Change (or check) the following:

    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*/echo/{input}

38. 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”:



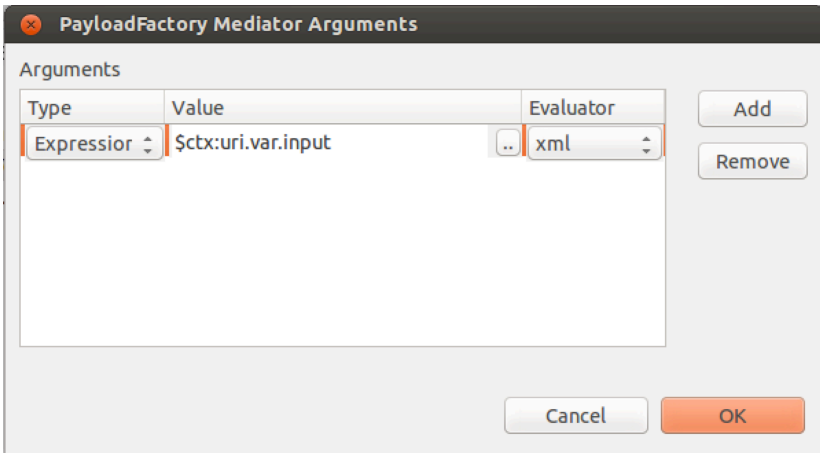
39. Now edit the properties of the PayloadFactory.

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 against the Echo service. If you want to do that, please go ahead, otherwise you can enter it from here. Choose Format (hit the little button, and then replace <inline/> with:

```
<p:echoString
xmlns:p="http://echo.services.core.carbon.wso2.org">
<in>$1</in>
</p:echoString>
```

40. 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**

It should look like:



41. Click **OK**

42. 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: **urn:echoString**
    Header Name: **Action**

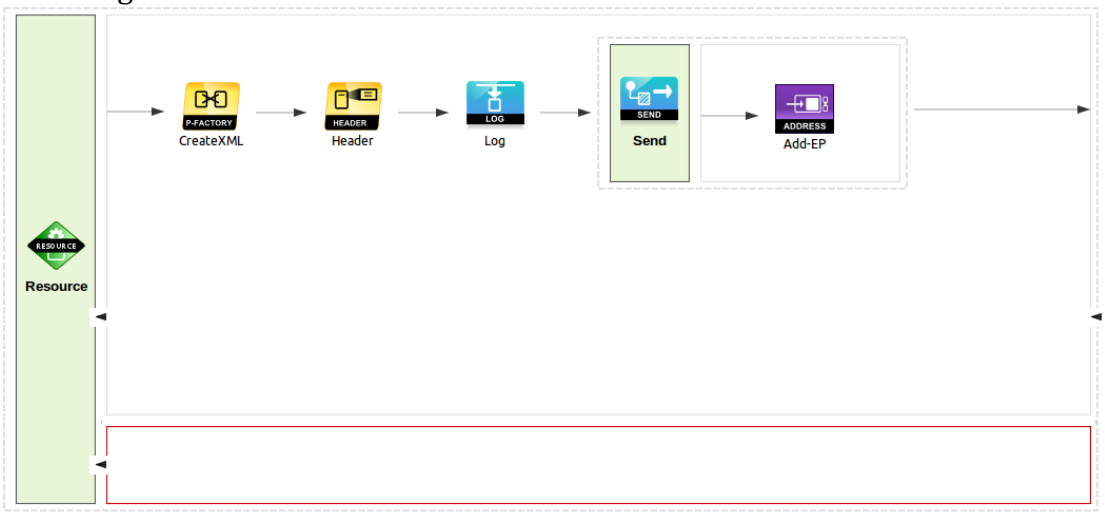
43. Now drop a Log mediator to the right, and set its log level to **FULL**.

44. We are now ready to send our SOAP message to the SOAP service. Drop a Send Mediator to the right. It will have an empty box inside the mediator.

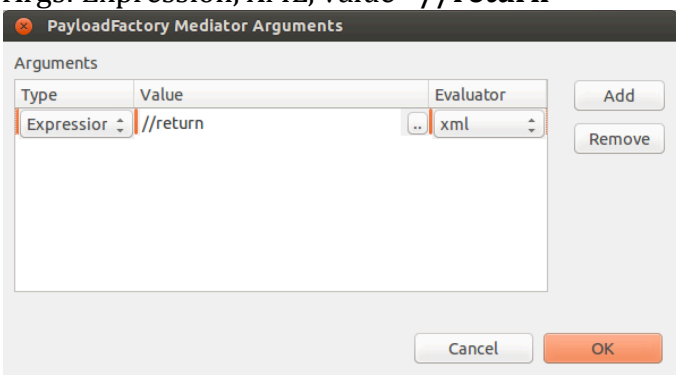
45. Open the Endpoints section on the left and drop an **Address Endpoint** into the empty box. Edit the description from Add-EP to echoSOAP.

46. In the properties section, under **Basic**, change the URI from http://www.example.org/service to **http://localhost:9763/services/echo**

47. Scroll down the properties until you get to the Misc->Format, and set that to be **soap11**.
Your diagram should look like:



48. Underneath the previous line on the return path, now drop another Log mediator, and again set the log level to **FULL.**
49. To the left of that drop another PayloadFactory, and change its description to **toJSON**.
50. Set its properties as follows:
Media Type: JSON
Format: **{ return: "$1"}**
Args: Expression, XML, value - **//return**



This will grab the value of the first element called <return> and use that as the value of a JSON string.

51. Unfortunately due to a bug/feature/oversight (which is being fixed), despite using json in the PayloadFactory, the ESB still thinks the media type of the message is XML which came back from the SOAP service. We need to fix that.

    Drop a property mediator to the left of the PayloadFactory. Set the description to be **jsonMediaType.**

    Now edit the properties:
    Property Name: **messageType**
    Property Action: **set**
    Value Literal: **application/json**
    Property Scope: **axis2**

52. Now drag another Send Mediator left of the property mediator. You can leave the address empty because we are implicitly sending this back to the client on the open HTTP channel.

53. You have now created an ESB API that will:
    - Listen at /echo/{input}
    - Extract the {name} value
    - Construct an XML message
        - Using the name parameter
    - Send this as SOAP11 to our server endpoint
    - Send the response back to the client

54. 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 xmlns="http://ws.apache.org/ns/synapse" name="echoAPI" context="/echo">
    <resource methods="GET" uri-template="/{input}">
        <inSequence>
            <payloadFactory media-type="xml" description="CreateXML">
                <format>
                    <p:echoString xmlns:p="http://echo.services.core.carbon.wso2.org">
                        <in>$1</in>
                    </p:echoString>
                </format>
                <args>
                    <arg evaluator="xml" expression="$ctx:uri.var.input"/>
                </args>
            </payloadFactory>
            <header name="Action" scope="default" value="urn:echoString"/>
            <log level="full" description="Log"/>
            <send>
                <endpoint>
                    <address uri="http://localhost:9763/services/echo" format="soap11"/>
                </endpoint>
            </send>
        </inSequence>
        <outSequence>
            <log level="full" description="log again"/>
            <payloadFactory media-type="json" description="toJSON">
                <format>{ return: "$1"}</format>
                <args>
                    <arg evaluator="xml" expression="//return"/>
                </args>
            </payloadFactory>
            <property name="messageType" value="application/json" scope="axis2"
type="STRING" description="jsonMediaType"/>
            <send/>
        </outSequence>
        <faultSequence/>
    </resource>
</api>
```

This XML is available at https://gist.github.com/pzfreo/7660591

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 two mediators. First we create an XML payload, then we send that to an endpoint.

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

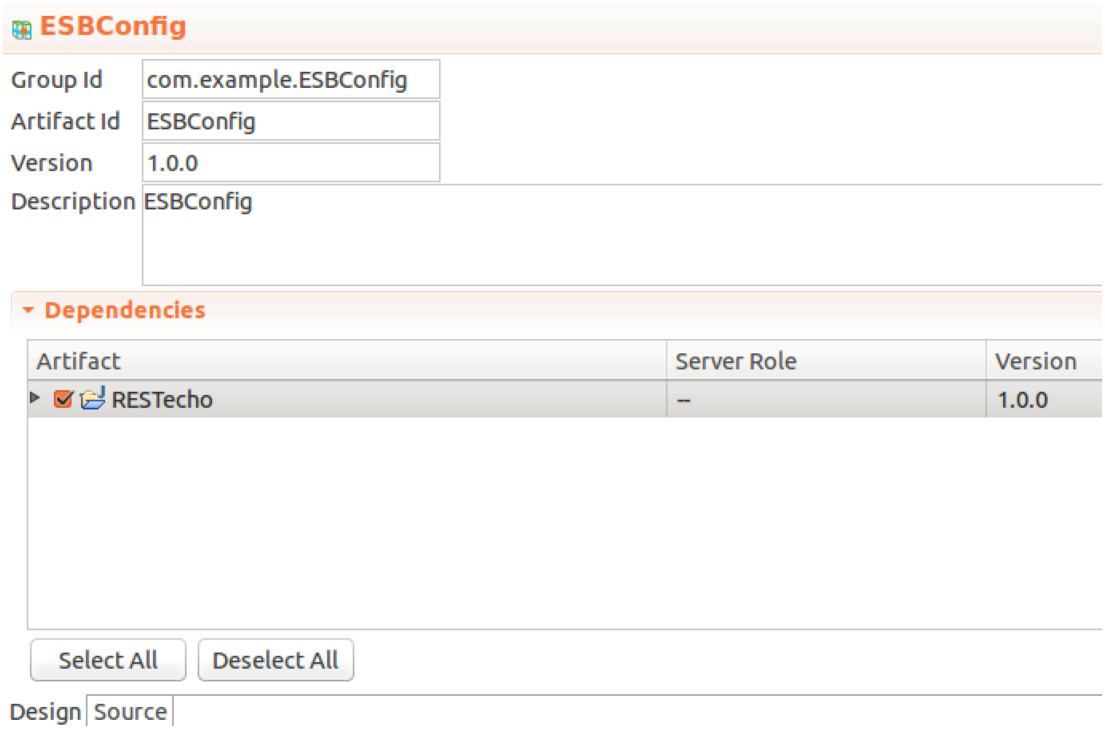
56. You can check the server is running (in a minute) by browsing https://localhost:9444/. You will need to Proceed past the security warning because by default the server is using a self-signed certificate. The default credentials are **admin/admin.**

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

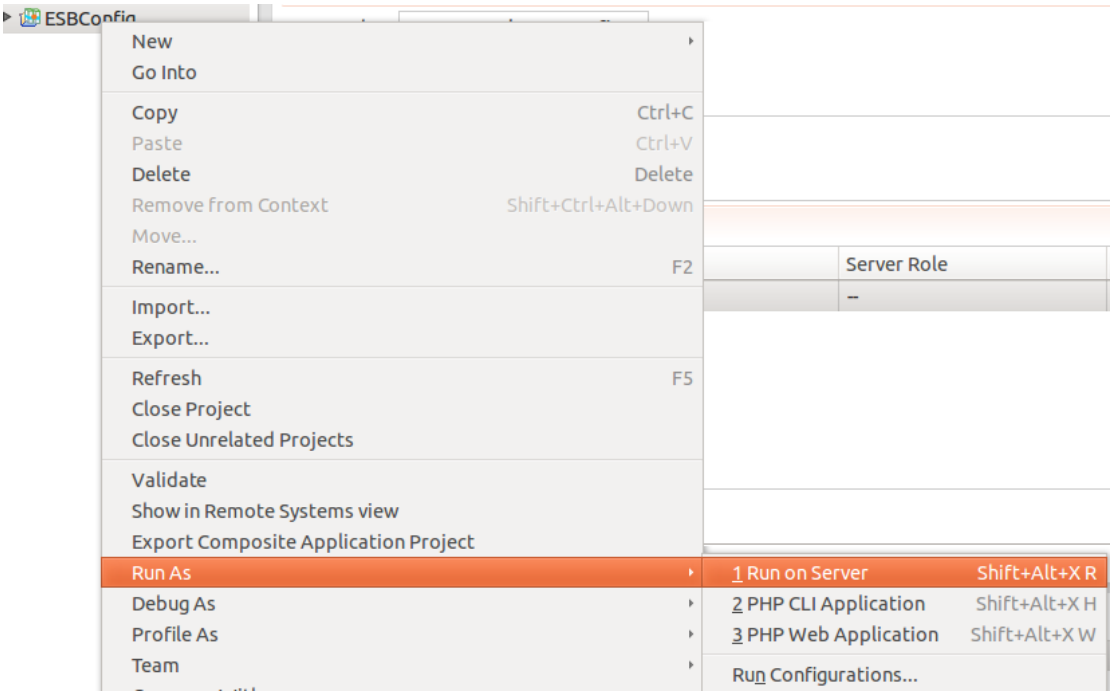
58. Click **Next**.

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

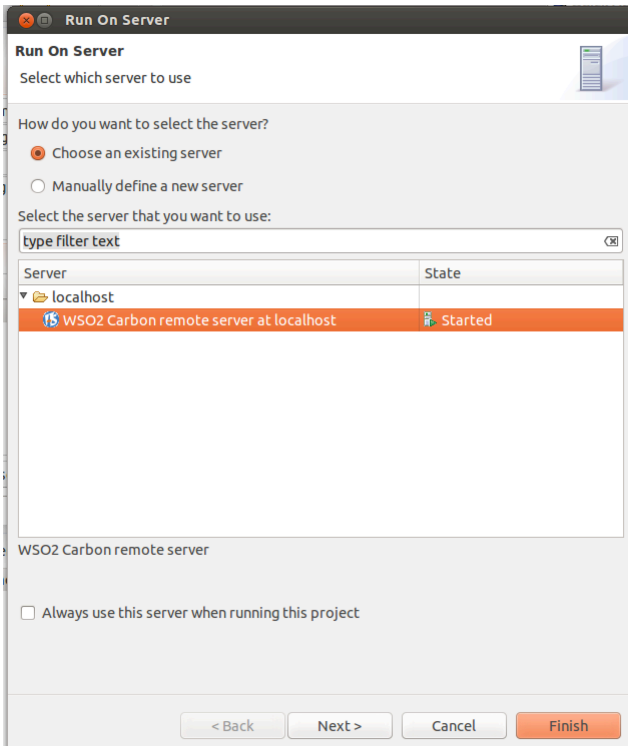
60. You need to make sure the RESTecho ESB config is part of the Composite Application Project. Open up the ESBConfig project and it will open the pom.xml. Make sure the RESTecho Artifact is checked:



61. 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**

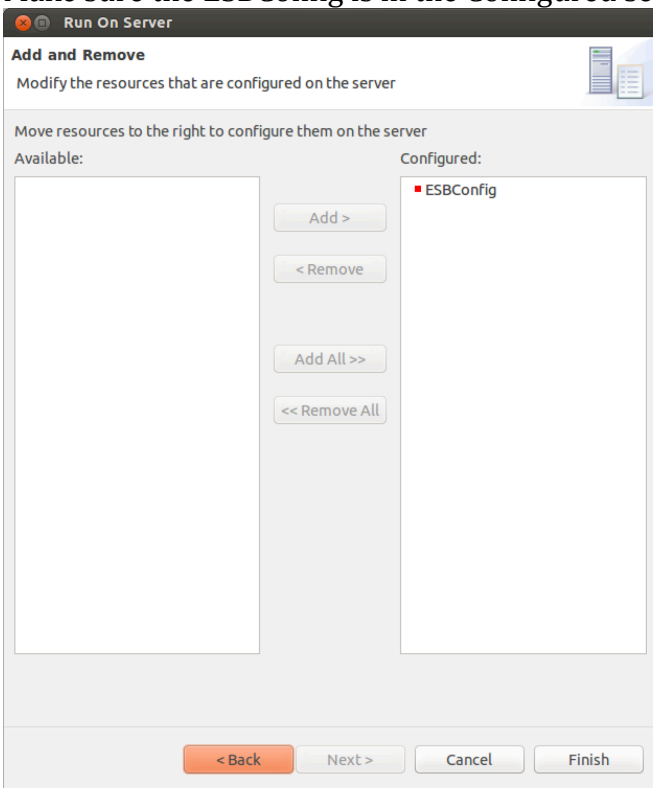


62. Make sure the Carbon Server is selected:



Click Next

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

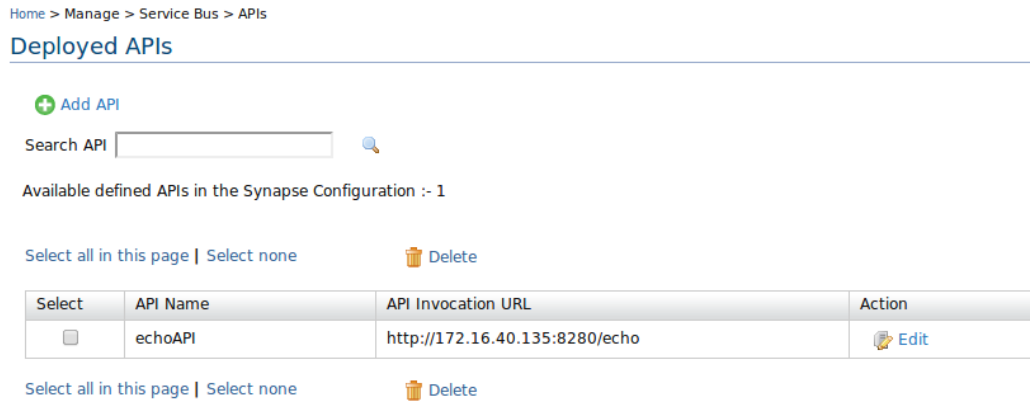


64. Click **Finish**.

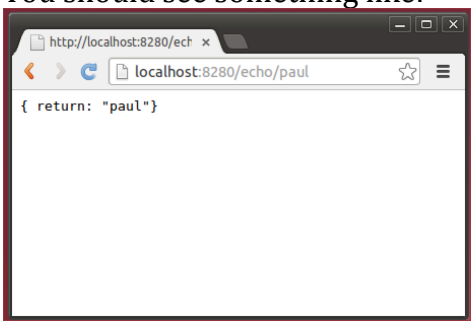
65. If you look at your ESB terminal window where the server is running you should see something like:

```
[2013-11-26 16:09:48,830]  INFO - ApplicationManager
Deploying Carbon Application :
ESBConfig_1.0.0.car...
[2013-11-26 16:09:48,834]  INFO - API Initializing
API: echoAPI
[2013-11-26 16:09:48,834]  INFO - APIDeployer API
named 'echoAPI' has been deployed from file :
/home/ox-soa/servers/wso2esb-
4.8.0/repository/carbonapps/work/1385482188830ESBCon
fig_1.0.0.car/echoAPI_1.0.0/echoAPI-1.0.0.xml
[2013-11-26 16:09:48,834]  INFO - ApplicationManager
Successfully Deployed Carbon Application :
ESBConfig_1.0.0 {super-tenant}
```

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



67. Now try the API by browsing http://localhost:8281/echo/paul
You should see something like:



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

```
[2013-11-26 16:44:51,324]  INFO - LogMediator To: /echo/paul,
WSAction: urn:echoString, SOAPAction: urn:echoString,
MessageID: urn:uuid:3176a1f4-4329-4bf6-93d4-9fbbe67dc902,
Direction: request, Envelope: <?xml version="1.0"
encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://www.w3.org/2003/05/soap-
envelope"><soapenv:Body><p:echoString
xmlns:p="http://echo.services.core.carbon.wso2.org"><in
xmlns="http://ws.apache.org/ns/synapse">paul</in></p:echoString
></soapenv:Body></soapenv:Envelope>
[2013-11-26 16:44:51,330]  INFO - TimeoutHandler This engine
will expire all callbacks after : 120 seconds, irrespective of
the timeout action, after the specified or optional timeout
[2013-11-26 16:44:51,358]  INFO - LogMediator To:
http://www.w3.org/2005/08/addressing/anonymous, WSAction: ,
SOAPAction: , MessageID: urn:uuid:694d3459-4369-4c80-9e8c-
b365368e8009, Direction: response, Envelope: <?xml
version="1.0" encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soap
env:Body><ns:echoStringResponse
xmlns:ns="http://echo.services.core.carbon.wso2.org"><return>pa
ul</return></ns:echoStringResponse></soapenv:Body></soapenv:Env
elope>
```

69. Extensions:

    The ESB would allow you to do this completely “inline” without calling an external service, since the logic is pretty simple. Create a new API that does this. Hint: use the Respond mediator.

70. There are lots of ESB samples you can look at here: http://docs.wso2.org/display/ESB480/Samples