WSO2 ESB for Developers – Fundamentals

Lab Kit: Prerequisites and Prequel to Exercises

Table of content:

Prerequisites: Reading and Installations

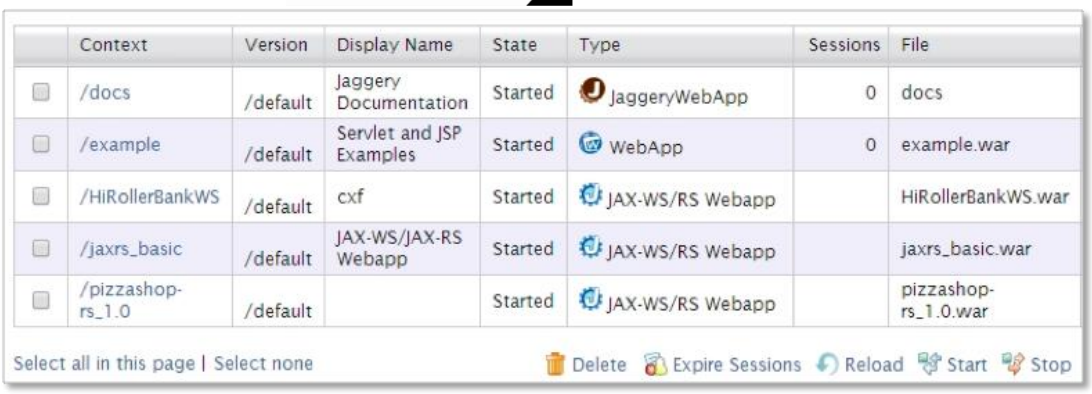
0. Read 'What is Service-oriented architecture?'
   http://www.javaworld.com/article/2071889/soa/what-is-service-oriented-architecture.html
1. Install WSO2 ESB 4.8.1
   http://docs.wso2.com/display/ESB481/Getting+Started
2. (win32 only) Install Patch ESBJAVA-3272 (Resolves configuration redeploy bug)
   https://wso2.org/jira/secure/attachment/33396/org.wso2.carbon.application.upload_4.2.0.jar
   Download jar file and copy to [ESB_HOME]/repository/components/plugin
3. Install WSO2 App Server 5.2.1 (To host your backend services)
   http://docs.wso2.com/display/AS521/Getting+Started
4. Install Eclipse IDE for Java EE Developers (Kepler SR2) and Developer Studio 3.7.0
   https://docs.wso2.org/display/DVS360/Getting+Started
5. Install SoapUI 5.0.0 or higher (Testing tool)
   http://www.soapui.org/
6. Ensure you have installed Java 1.7, **NOT** 1.8

You will need to install a local copy of App Server to host the backend services used during training.

1. If you haven't done so yet, Install AS 5.2.1 (see prerequisites)
2. Set *port-offset* to 1 in [AS_HOME]\repository\conf\carbon.xml to avoid using your ESB ports
3. Start you AS instance (Double-click [AS_HOME]\bin\wso2server.bat
4. In browser, navigate to Mgt Console URL listed in your command screen log
   a. Example: https://localhost:9444/carbon/
   b. Username: **admin** and Password: **admin**
5. Upload the .war files that host your backend services
   a. At left, click on Main tab
   b. In left-navigation bar, under Applications, under Add, click on JAX-WS / JAX-RS
   c. Upload *HiRollerBankWS.war* and *pizzashop-rs_1.0.war*

- *(These can be downloaded from : https://www.dropbox.com/s/1xnehu9vsurckr1/ESBFundamentalsLabs.zip?dl=0)*

6. Under Applications, click on List to view your list of running applications



	Context	Version	Display Name	State	Type	Sessions	File
☐	/docs	/default	Jaggery Documentation	Started	JaggeryWebApp	0	docs
☐	/example	/default	Servlet and JSP Examples	Started	WebApp	0	example.war
☐	/HiRollerBankWS	/default	cxf	Started	JAX-WS/RS Webapp		HiRollerBankWS.war
☐	/jaxrs_basic	/default	JAX-WS/JAX-RS Webapp	Started	JAX-WS/RS Webapp		jaxrs_basic.war
☐	/pizzashop-rs_1.0	/default		Started	JAX-WS/RS Webapp		pizzashop-rs_1.0.war

Select all in this page | Select none Delete Expire Sessions Reload Start Stop

You now have a local AS that will host your backend services.

LAB 00: Getting Started with WSO2 ESB's Management Console

Training Objective: Introduce WSO2 ESB Management Console Tabs and menus, learn how to manage users & roles and list and test WSO2 ESB out-of-the-box deployed services through integrated tools. This will introduce students to WSO2 ESB's Management Console operation.

High-level Steps:

- Know ESB Console Tabs and Options
- Connect to ESB Console as admin User
- Create a tester User
- Create a Role with only test permissions and assign the new tester user to it.
- Test echo service as tester User

Overview of ESB's Management Console Tabs and Options

ESB Console provides a set of options aggregated in four main tabs:

- **Main**
- **Monitor**
- **Configure**
- **Tools**

Under **Main** tab we can find a set of options to manage the following ESB resources:

- **Services**: WSO2 ESB deployed services.
- **Proxy Services**: Virtual services that appear as regular fledged web services to the external clients. There are proxy services of the following types:
    - Pass Through Proxy: A simple proxy service on a specified endpoint that does not perform any processing on the messages.
    - Secure Proxy: A proxy that will process WS-Security on incoming requests and forward them to an unsecured backend service.
    - WSDL Based Proxy: A proxy service out of a WSDL of an existing Web Service.
    - Logging Proxy: A proxy service which logs all the incoming requests and forwards them to a given endpoint.
    - Transformer Proxy: A proxy service which transforms all the incoming requests using XSLT and then forwards them to a given endpoints.
    - Custom Proxy: Launch the proxy service creation wizard.
- **Sequences**: An array of mediators assembled as a message flow.
- **Scheduled Tasks**: Jobs deployed in the WSO2 ESB runtime for periodic execution.
- **Templates**: Prototypes of Endpoints or Sequences that can be used as a base for new objects.
- **Endpoints**: Definitions of external services endpoints and any attributes or semantics that should be followed when communicating with them.
- **Local Entries**: Local configuration registry/repository for resources such as WSDLs, schemas, scripts, etc.
- **Message Processors**: Units of execution that have the ability to connect to a message store and perform message mediation or required data manipulations.
- **Message Stores**: Units of storage (queues) for messages/data exchanged during WSO2 ESB runtime. Queues can refer to local ESB in memory store implementation or to an external MQ product.
- **Priority Executors**: Resources used in high load scenarios where user wants to execute different sequences at different priority levels

- **APIs:** Virtual Web application that provides a convenient approach for filtering and processing HTTP traffic through the service bus.
- **Source View:** ESB full XML configuration (except resources that are stored in registry).
- **Connectors:** Zip modules with components to connect to external applications.
- **Secure Vault Tool:** Tool for encrypting and storing passwords.
- **Carbon Applications: A**pplications developed for WSO2's Product Platform (*.car files).
- **Modules:** Modules developed over WSO2's Product Platform (*.mar files). These extensions can contain processors, executors, etc.
- **Topics:** Group of messages on queues.
- **ESB Artifacts:** Custom developed extensions that can be deployed into the ESB. These extensions can contain custom mediators, tasks, configuration factories and serializers.
- **Registry:** Browse the registry configuration resources.

Under **Monitor** tab we can find a set of options to check the following ESB resources:

- **Application Logs:** Logs of WSO2 ESB deployed services.
- **Transport Statistics:** HTTPS,HTTP & Local Transport statistical data such as min & max package size, timeouts, threads.
- **System Statistics:** System statistical data such as average response time, total requests served, memory allocated.
- **Mediation Statistics:** Mediator execution statistics (requires enabling).
- **System Logs:** Logs of WSO2 ESB operation.
- **SOAP Tracer:** SOAP execution data (requires enabling).
- **Mediation Tracer:** Mediator execution detailed data (requires enabling).
- **Message Flows:** Graphical view of internal message in & out flowing.

Under **Configure** tab we can find a set of options to manage the following ESB resources:

- **Users and Roles:** Users and permissions for actions.
- **Cloud Gateway Agent:** Selectively publish services and data to the cloud through WSO2 Cloud Gateway Agent.
- **User Store Management:** Secondary User Stores.
- **Features:** Installed WSO2 features.
- **KeyStores:** Stores for secret, private and trusted certificates keys.
- **Logging:** WSO2 ESB log4j configuration.
- **Data Sources:** Connection definitions to Databases to be used by WSO2 ESB resources.
- **WS-Discovery:** Proxies that discover and manage service endpoints in your SOA.
- **Mediation Data Publishing:** Mediator statistics for BAM profile publication.
- **BAM Server Profile:** Resource where the user can define a set of event streams that can be used at the BAM mediator declaration time.
- **Server Roles:** Security Server Roles.
- **Multi Tenancy:** Multiple ESB resource instances on a single ESB installation.

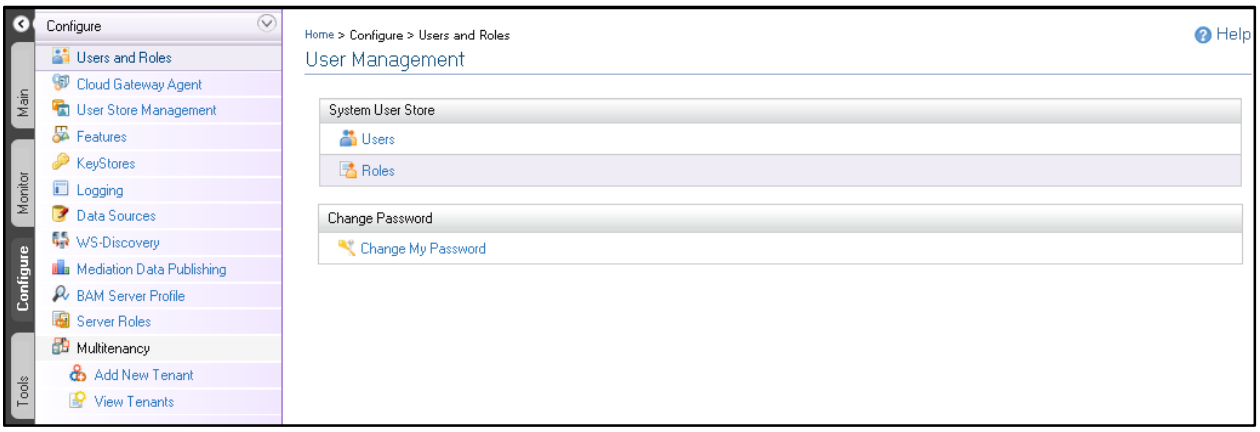
Under **Tools** tab we can find a set of options to manage the following ESB resources:

- **Java2WSDL:** Bottom-up Java Web Service Builder.
- **Try It:** Internat tool for service testing. (only useful when you do not have SOAPUI ready)
- **WSDL Validator:** Tool for validating WSDL files.

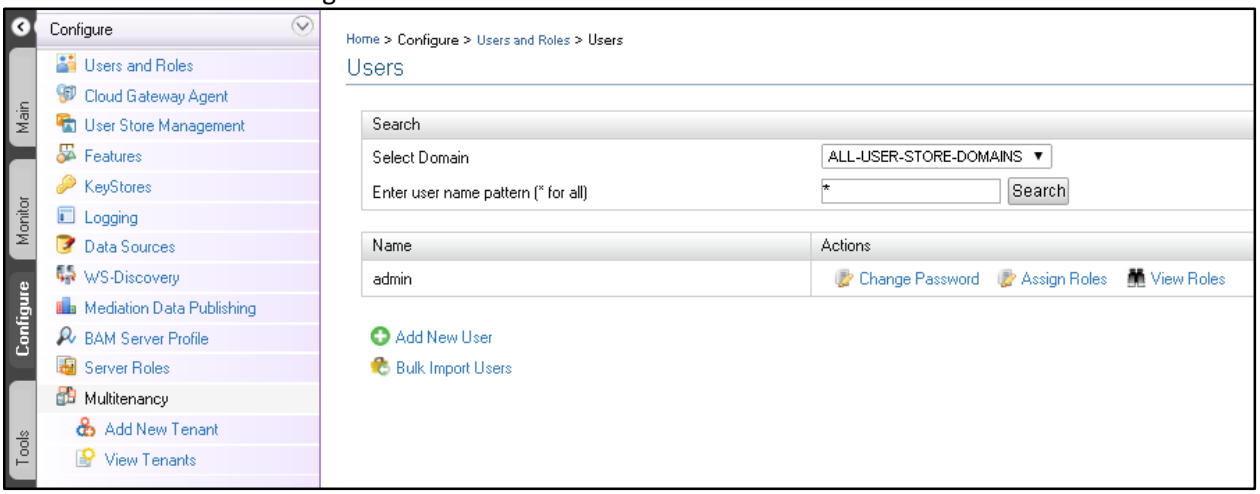
Detailed Instructions:

Add a Tester Role & User

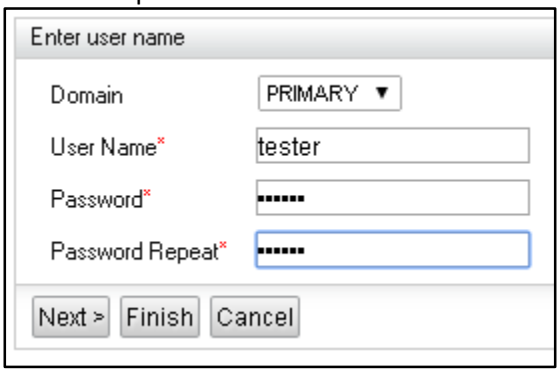
1. Open a browser and access https://localhost:9443/carbon (allow access without a valid certificate)
2. Login as user *admin* and password *admin*
3. Select *Configure* Tab
4. Select *User & Roles* option from menu



5. Choose User management and click *Add New User* action

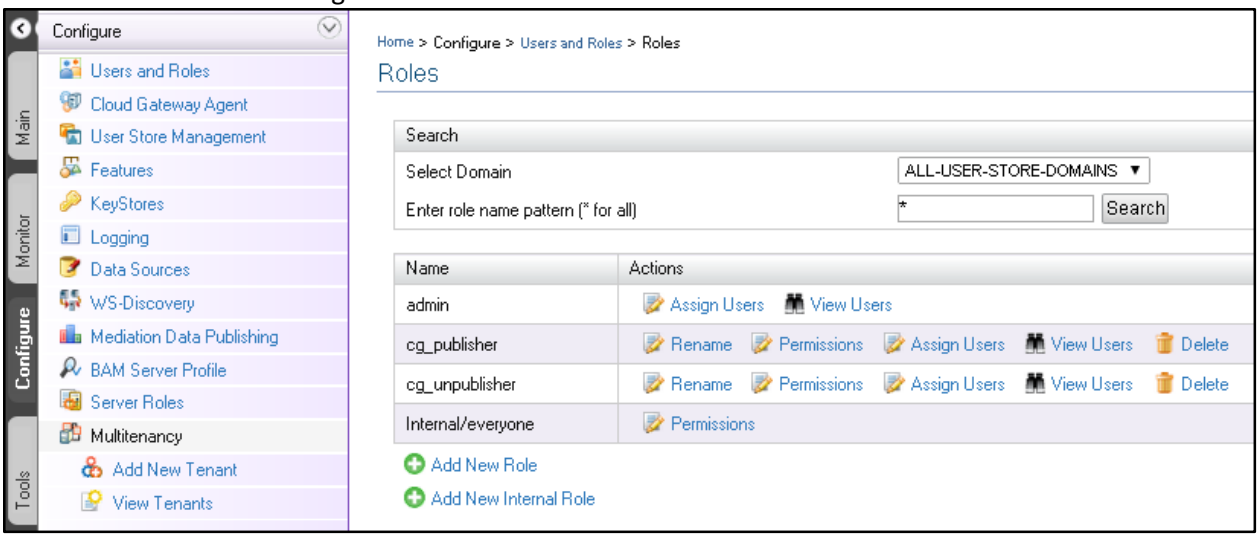


6. Complete with name *tester* and password tester

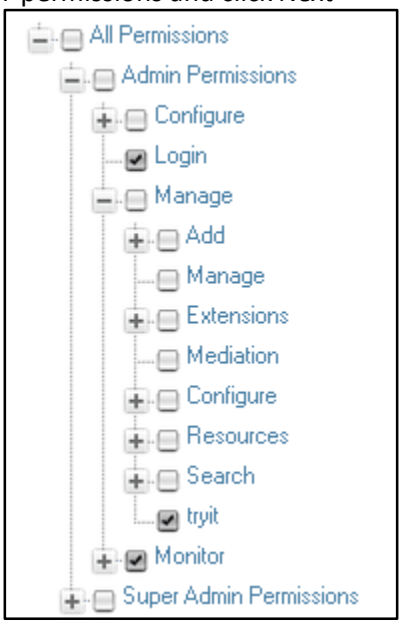


7. Issue *Finish* to complete user creation

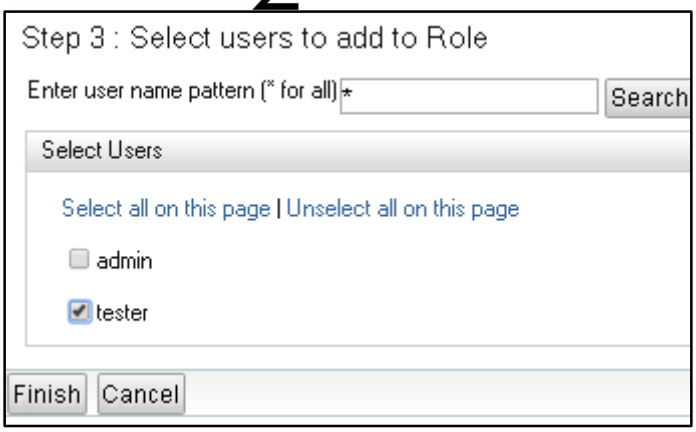
8. Select *User & Roles* option from menu
9. Choose Roles management and click *Add New Role* action



10. Set the role name to *Tester* and click *Next*
11. Select *Login, Tryit & Monitor* permissions and click *Next*



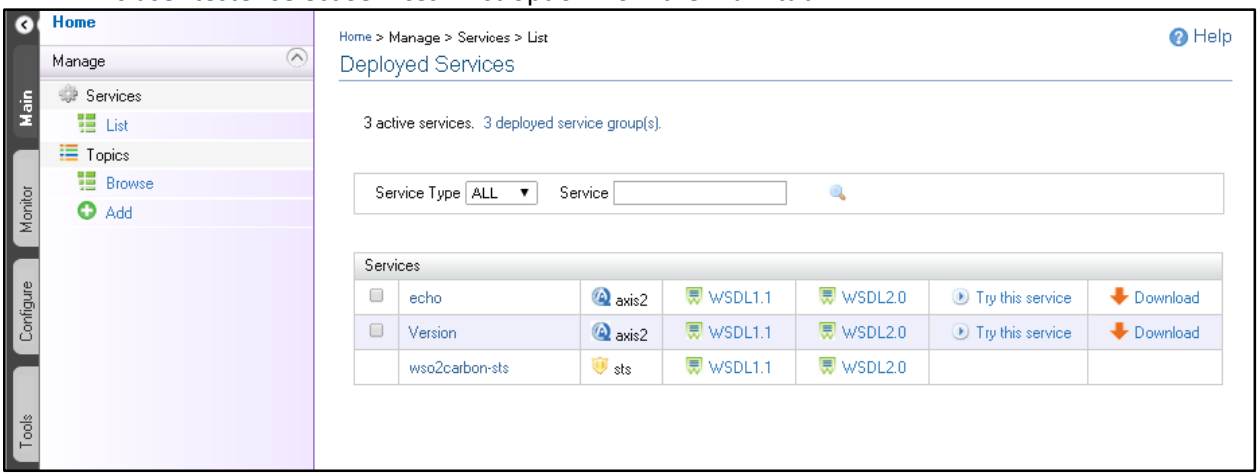
12. Select the user created on previous step to add to role



13. Issue *Finish* to complete role creation and assignation.
14. Logout and Login as user tester, check the tabs and options this user is able to perform.

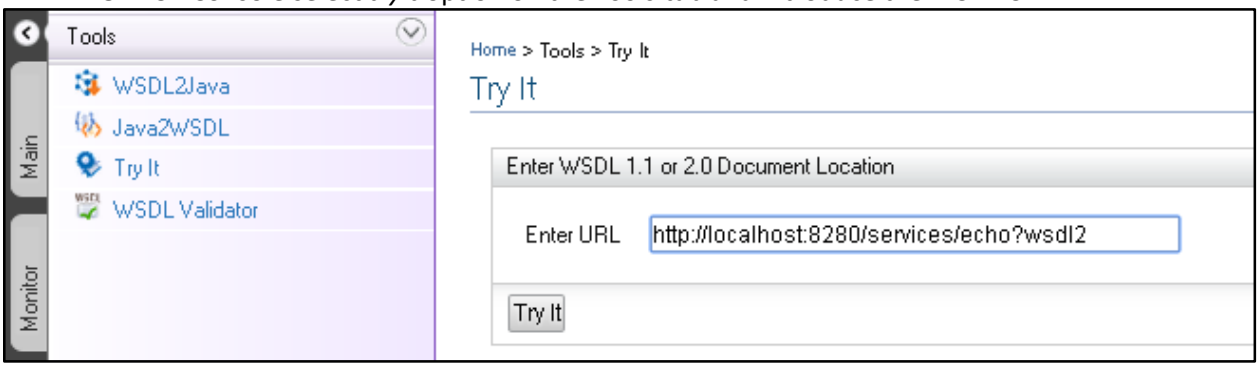
Test the service

1. As user tester select *Services->List* option from the *Main* tab

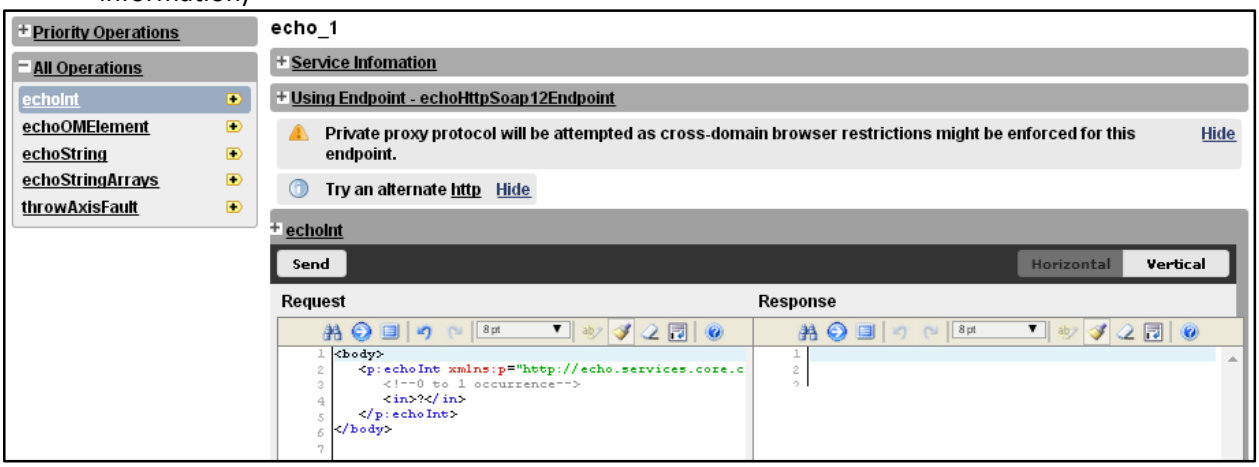


2. Click on WSDL2.0 cell of echo service to access to its definition (A new browser tab is opened).

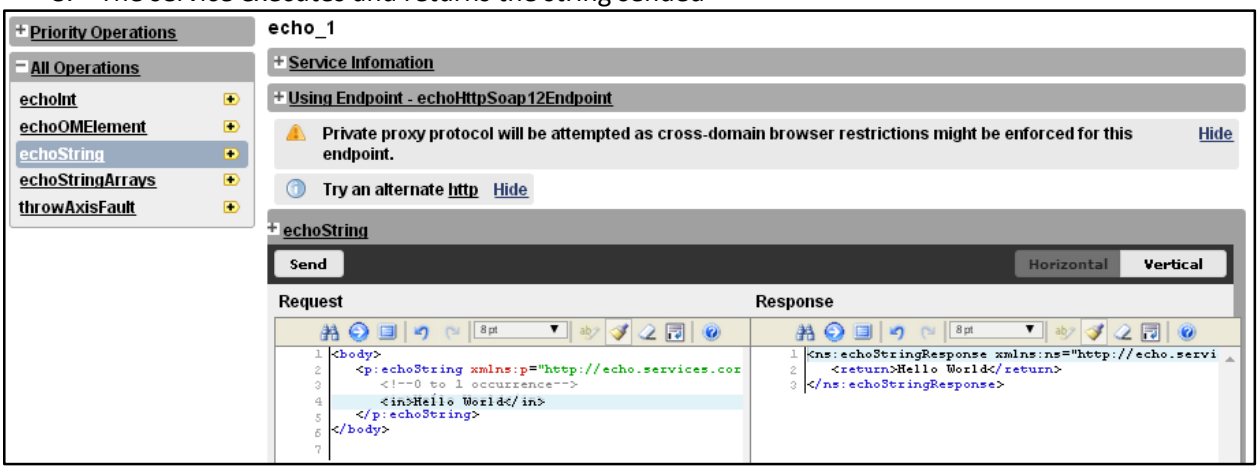
3. Copy the URL of the WSDL definition
4. On ESB Console select *tryIt* option on the *Tools* tab and introduce the WSDL URL



5. Click on Try It to test the service (popup window will be opened showing service detailed information)



6. Select *echoString* operation, introduce a text between <in> and </in> tags of the XML payload body.
7. Click on Send button
8. The service executes and returns the string sended



9. Run SOAPUI and try the same URL

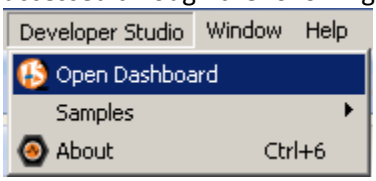
LAB 01: Getting Started with WSO2 ESB DevStudio

Training Objective: Understand WSO2 Developer Studio functionality, learn how set up a project, create a pass-through proxy with a simple sequence and deploy it to WSO2 ESB. This will introduce students to Dev Studio configuration.

Dev Studio Overview:

WSO2 Developer Studio is an IDE solution for WSO2 Application Development where developers can define a project representing a complete Composite Application (C-App) spanning multiple WSO2 products and features.

Dev Studio provides a main dashboard that allows ESB project development, this dashboard can be accessed through the following menu option.



Through the use of this dashBoard, a user can create two different types of ESB projects.

ESB Config Project	An ESB Project that holds ESB resources and can be deployed on a WSO2 ESB installation.
Mediator Project	A Java project with a mediator class that implements org.apache.synapse.mediators.AbstractMediator class.

Once created, an ESB Config Project can hold the following ESB resources:

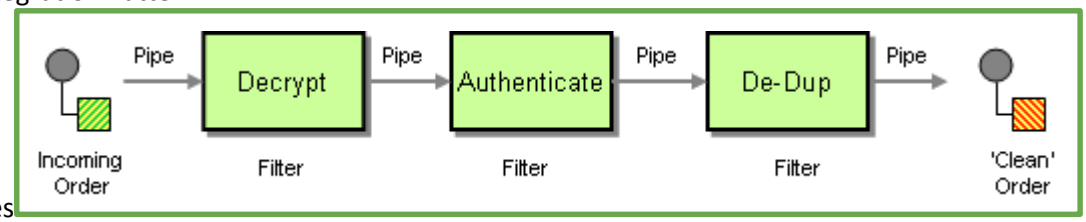
Endpoint	Definitions of external services endpoints and any attributes or semantics that should be followed when communicating with them.
Synapse	Synapse style definition of ESB resources.
Scheduled Task	Jobs in the WSO2 ESB runtime for periodic execution.
Message Processor	Units of execution that have the ability to connect to a message store and perform message mediation or required data manipulations
Proxy Service	Virtual services that appear as regular fledged web services to the external clients. There four types of proxy services: • Pass Through Proxy: A simple proxy service on a specified endpoint that does not perform any processing on the messages. • Secure Proxy: A proxy that will process WS-Security on incoming requests and forward them to an unsecured backend service.

	• WSDL Based Proxy: A proxy service out of a WSDL of an existing Web Service. • Logging Proxy: A proxy service which logs all the incoming requests and forwards them to a given endpoint. • Transformer Proxy: A proxy service which transforms all the incoming requests using XSLT and then forwards them to a given endpoints. • Custom Proxy: Launch the proxy service creation wizard.
Local Entry	Local configuration registry/repository for resources such as WSDLs, schemas, scripts, etc.
Sequence	An array of mediators assembled as a message flow unit.
REST API	Virtual web application that provides a convenient approach for filtering and processing HTTP traffic through the service bus.
Template	Prototypes of Endpoints or Sequences that can be used as a base for new objects.
Message Store	Units of storage (queues) for messages/data exchanged during WSO2 ESB runtime. Queues can refer to local ESB in-memory store implementation or to an external connected MQ product referenced by a JMS datasource.

Business Scenario:
Through a web service, HiRollerBank allows its users to enter a bank account number and receive their current account balance. Going forward, HiRollerBank would like to log all requests coming in.

Enterprise Integration Pattern:

Message Pipes



In this first exercise, you will learn to set up an ESB Configuration Project, which we will reuse throughout the training.
You will be using an existing backend service hosted in your local App Server that returns account balance (i.e. Receives a bank account number and returns the balance).

Backend Services	Location
Accounts Receives a bank account number and returns the balance.	http://localhost:9764/HiRollerBankWS/services/accounts

Follow the steps below to create a simple pass through proxy service that logs every message.

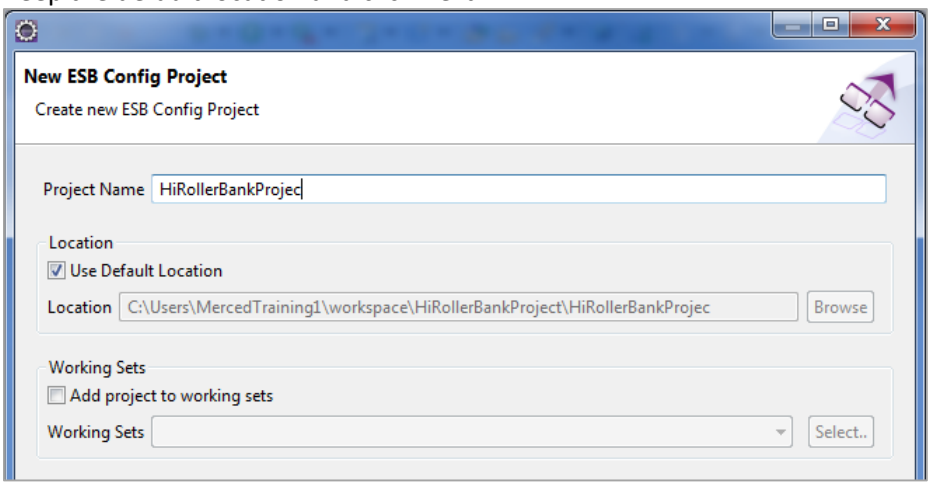
High-level Steps:

- Create an ESB Configuration Project to house your configurations
- Create a simple pass through Proxy Service (i.e. message is not altered)
- Add Log Mediator to existing Proxy Service (i.e. log every message coming through)
- Deploy your new configuration to your ESB (Move artifacts from Dev Studio to WSO2 ESB)
- Test your proxy service (Use Try It or SoapUI)

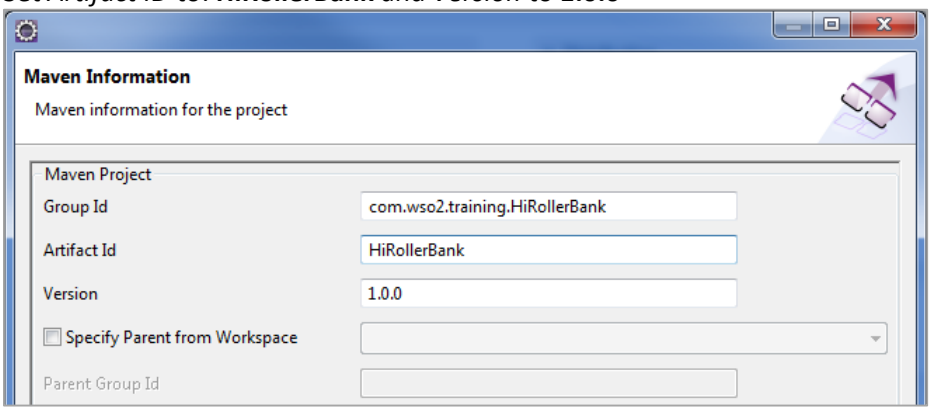
Detailed Instructions:

Create an ESB Configuration Project

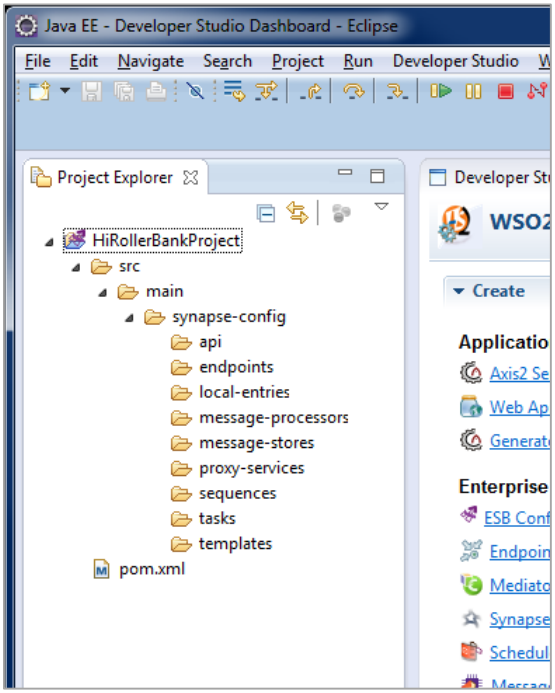
1. In Eclipse, open WSO2 configuration options by going to *Developer Studio → Open Dashboard*
2. Under *Enterprise Service Bus*, click on *ESB Config Project*
    a. Select *New ESB Config Project*
    b. Name your project **HiRollerBankProject**
    c. Keep the default location and click *Next*



3. Since this is a Maven project, you need to provide all the pertinent Maven info:
    a. Set *Group ID* to: **com.wso2.training.HiRollerBank**
    b. Set *Artifact ID* to: **HiRollerBank** and *Version* to **1.0.0**



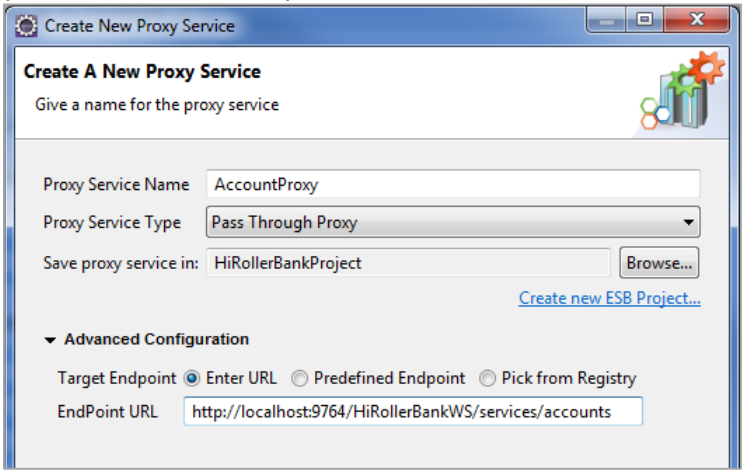
4. Click *Finish* to create a new project which you can view under *Project Explorer*. You can expand this to view all directories, including your *\src\main\synapse-config*. This is where all your ESB configurations will be stored.



Note: WSO2 ESB is based on Apache Synapse ESB, hence the *synapse-config* folder.

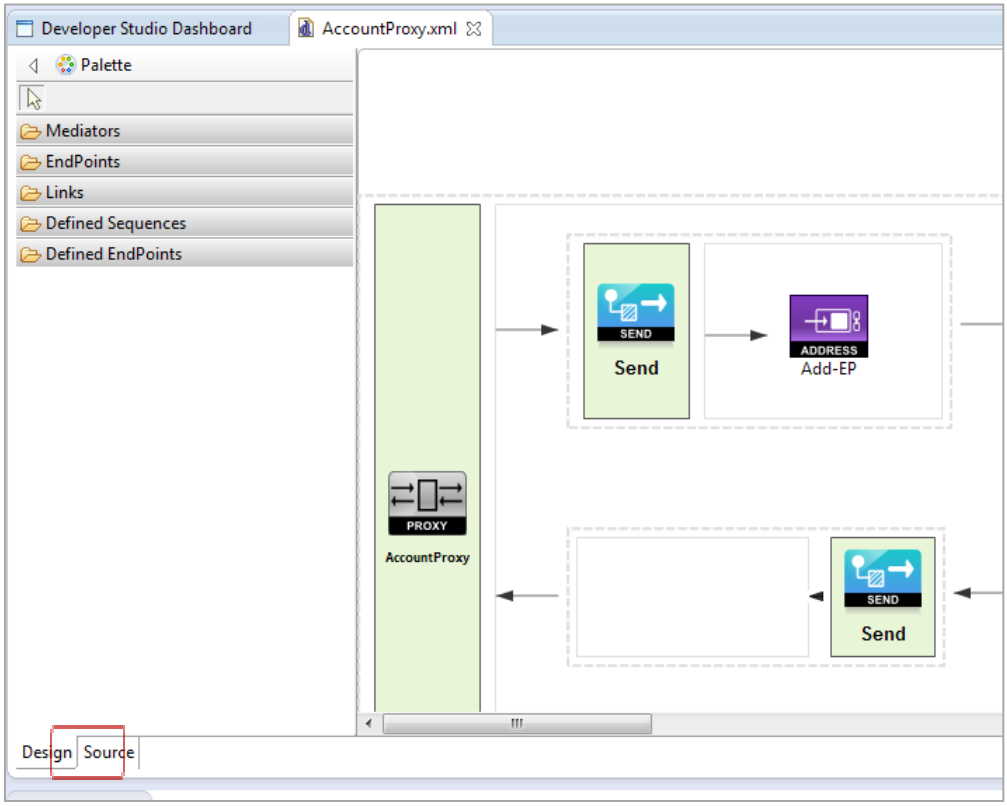
Create a Proxy Service

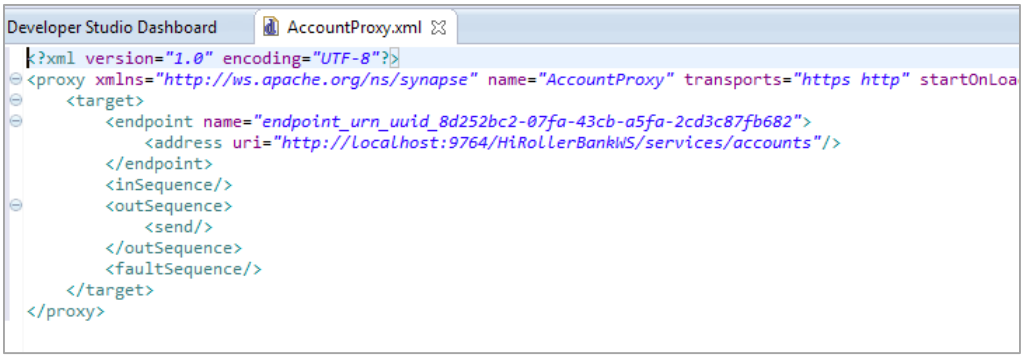
1. Right-click your new *HiRollerBankProject* and select *New➐Proxy Service*
    a. Select *Create a New Proxy Service*
    b. Set *Proxy Service Name* to: **AccountProxy**
    c. Set *Proxy Service Type* to: **Pass Through Proxy**
    d. For *Target Endpoint*, select **Enter URL**
    e. Set *EndPointURL*to:**http://localhost:9764/HiRollerBankWS/services/accounts** (this is your backend service – you can view details of *HiRollerBankWS* in your hosted AS)



f. Click *Finish* to save your new AccountProxy

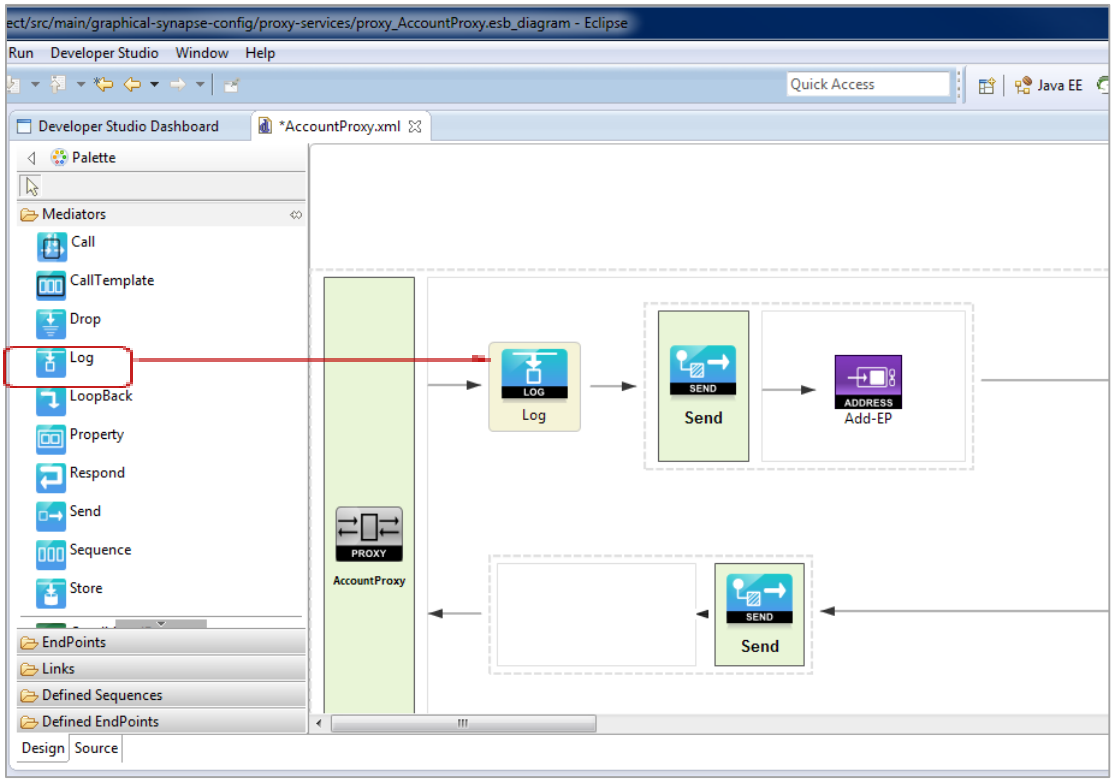
g. View your newly created *AccountProxy.xml* in *Design* and *Source* views





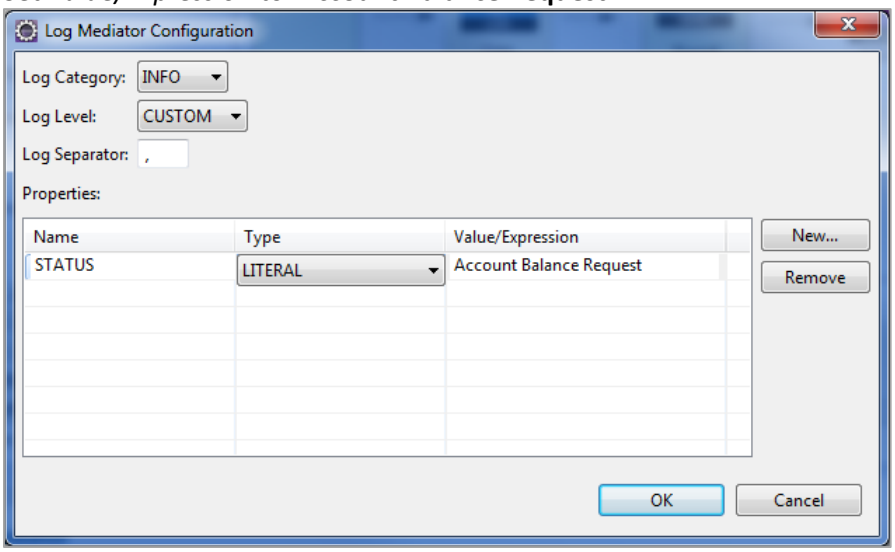
Add Log Mediator to Existing Proxy Service

1. In *Design* view, click on the *Mediators* folder to list the catalog of available mediators
2. Drag the *Log* mediator before the *Send* mediator



3. Right-click on your Log mediator and select *Configure* to view configuration options
    a. Set *Log Level* to: **CUSTOM** and Log Separator to: **,**
    b. Click on *New* to add a new custom property
    c. Set *Property Name* to: **STATUS**
    d. Set *Type* to: **LITERAL**

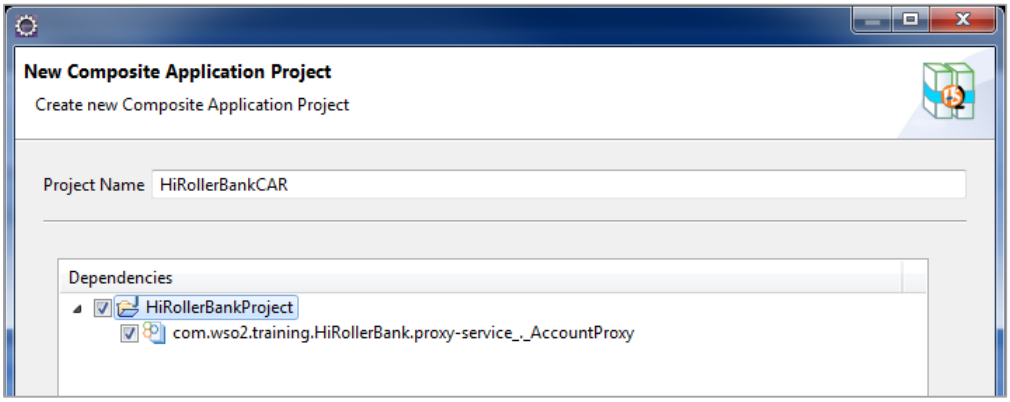
e. Set *Value/Expression* to: **Account Balance Request**



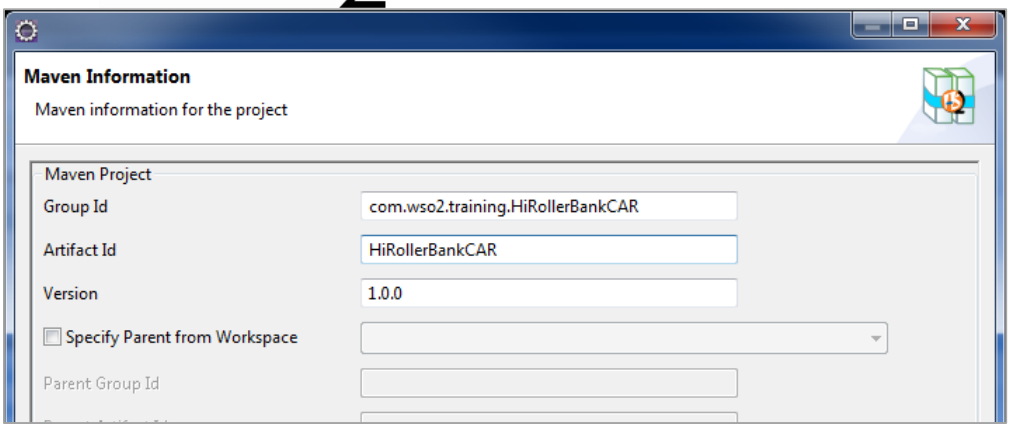
4. Click *OK* to save your changes.
5. In *Source* mode, search for the *<log>…</log>* tags to see your new changes
6. You can close the *AccountProxy.xml* tab to save your changes

Deploy your Configuration to your ESB

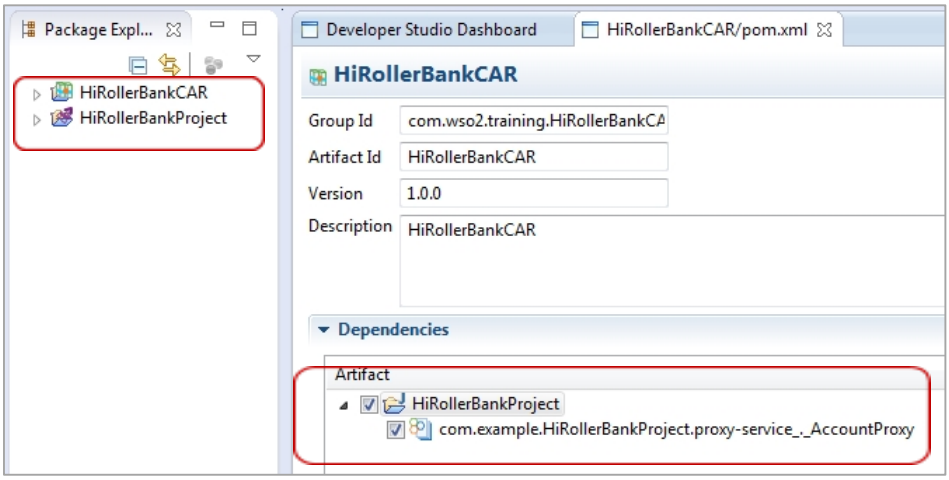
1. Create your CAR file (Collection of deployable artifacts)
   a. Under *Window➹Open Perspective*, select *Java*
   b. In the *Dev Studio Dashboard*, under *Distribution*, click on *Composite App Project*
   c. Set *Project Name* to: **HiRollerBankCAR**
   d. Select your *AccountProxy* service from the list of dependencies and click *Next*



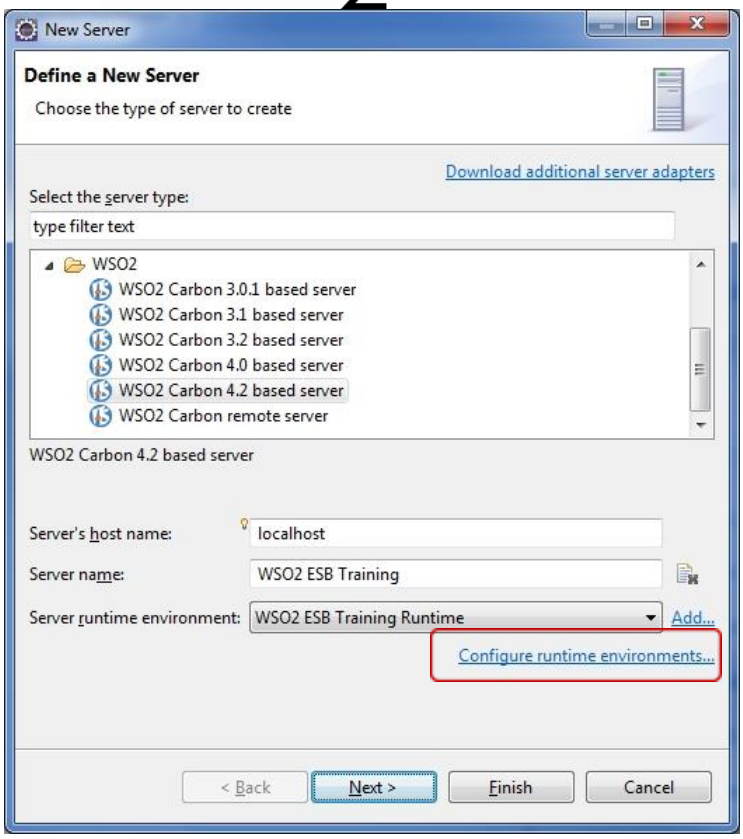
   e. In the Maven Info screen, set *Group ID* to: **com.wso2.training.HiRollerBankCAR**
   f. Set *Artifact ID* to: **HiRollerBankCAR** and *Version* to **1.0.0**



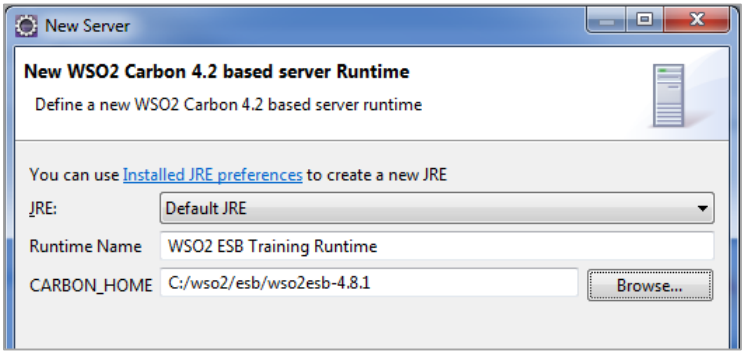
g. Click *Finish* to create your *HiRollerBankCAR/pom.xml* file. You will see your new file in your left-navigation bar under Package Explorer



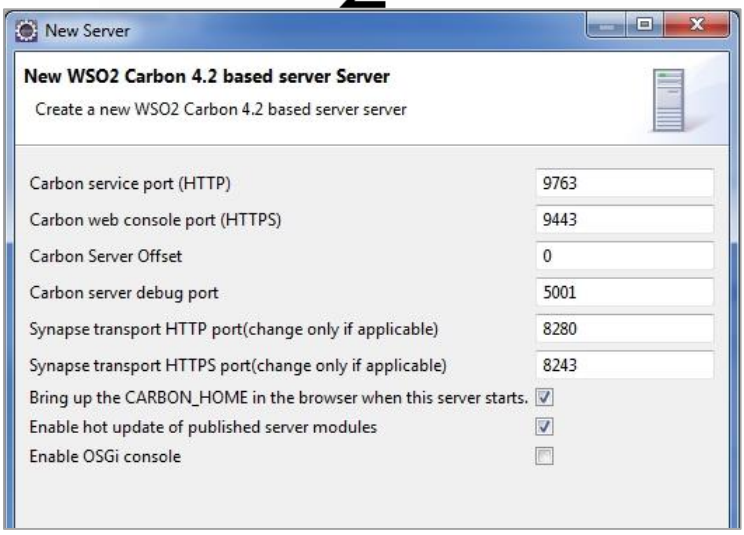
2. Deploy your CAR file to ESB
    a. In the *Dev Studio Dashboard*, under *Add Server*, click on *Server*
    b. Under WSO2, select *WSO2 Carbon 4.2 based server*
    c. Set *Server's Host Name* to: **localhost**
    d. Set *Server Name* to: **WSO2 ESB Training** and click *Configure runtime environments…*



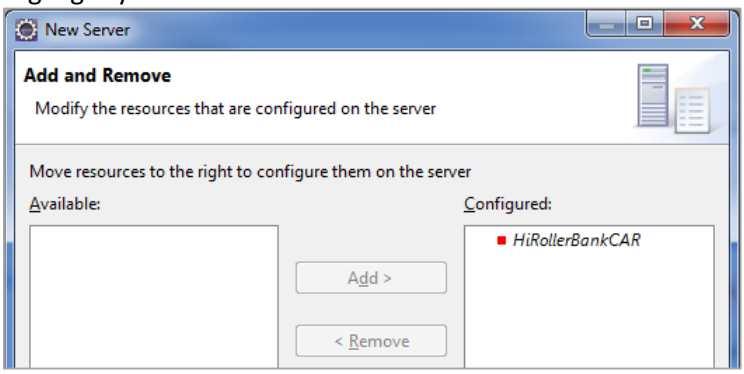
e. Click *on Edit…* if a runtime environment already exists, or *Add...* to create a new one
f. Set *JRE* to: **Default JRE**
g. Set *CARBON_HOME* to the local directory where you installed ESB, and click *Finish*



h. Click *OK* to return to the *Define a New Server* screen and click *Next* on this screen
i. Accept port defaults (unless you changed during ESB installation) and click *Next*

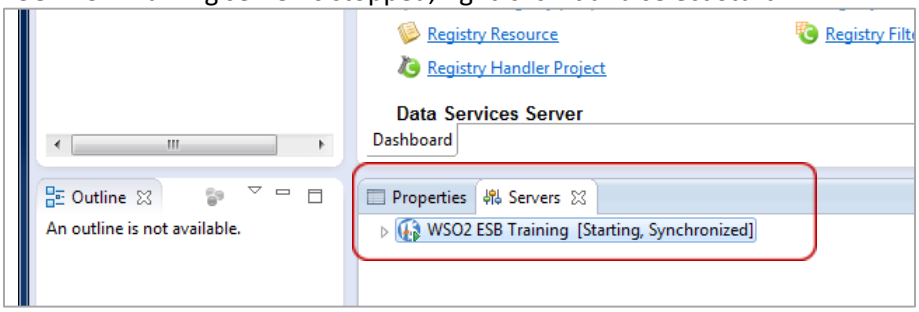


j. Highlight your *HiRollerBankCAR*and click *Add*>and click *Finish*



Test your proxy service

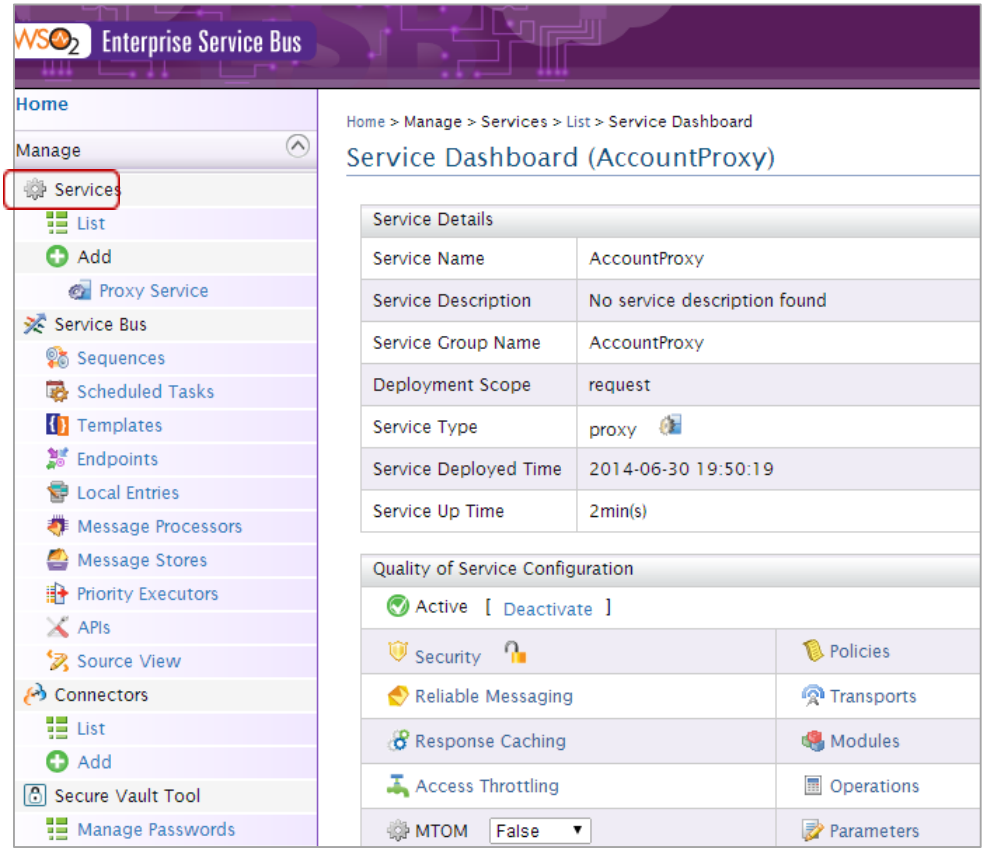
1. If your WSO2 ESB Training server is stopped, right-click it and select *Start*



a. A console tab appears displaying logs. If logs don't show *HiRollerBankCAR* being deployed, follow steps b and c, otherwise skip to d.
b. Drag *HiRollerBankCAR* into your WSO2 ESB Training server icon
c. Logs will start showing deployment of *HiRollerBankCAR*
d. A browser will automatically open *ESB Management Console*
e. Log in using *admin/admin*

f. In the left-nav bar, under *Services*, click *List*

g. You will see your *AccountProxy*listed, click on the name to explore its settings



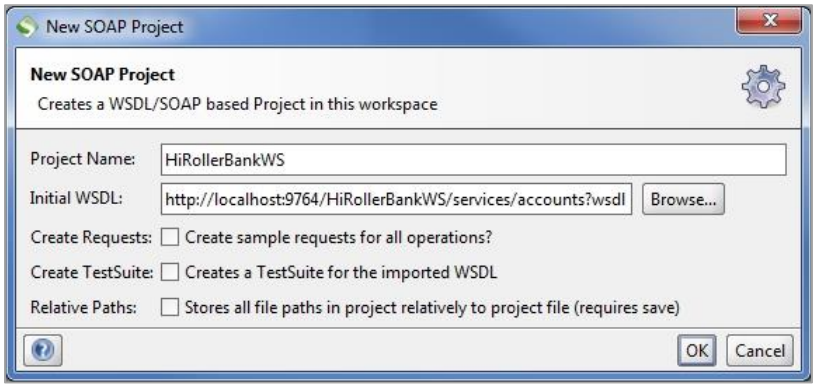
2. In SoapUI, test the existing service and the newly created proxy service

   a. In SoapUI, select File ➊New *SOAPProject*

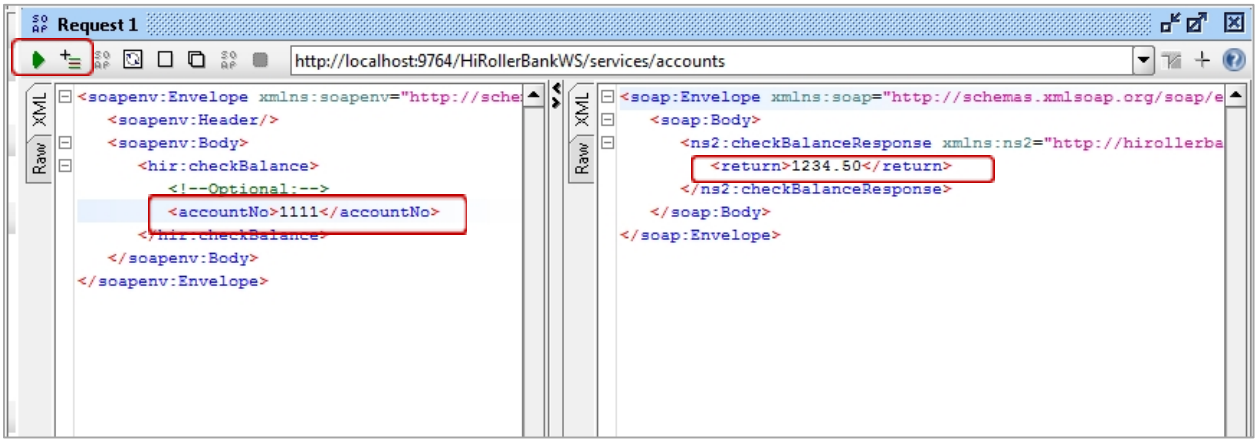
   b. Set *Project Name* to **HiRollerBankWS**

   c. Set *Initial WSDL* to **http://localhost:9764/HiRollerBankWS/services/accounts?wsdl**(verify this by exploring the *HiRollerBankWS* app in AS)

   d. Click *OK*

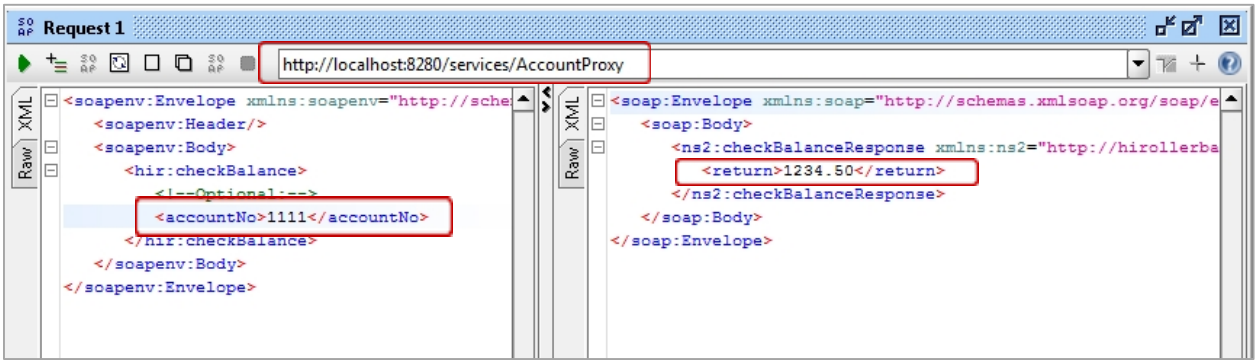


e. On left-nav bar, under the *HiRollerBankWS,* under*checkBalance,* double-click on *Request1*
f. You can leave *Name* as **Request1**
g. In Request window, set the *accountNo* to **1111**by replacing the '?'
h. Click on Green play icon (top right) to send the *request* to the back-end
i. In the response, look for a returned account balance**($1234.50)**

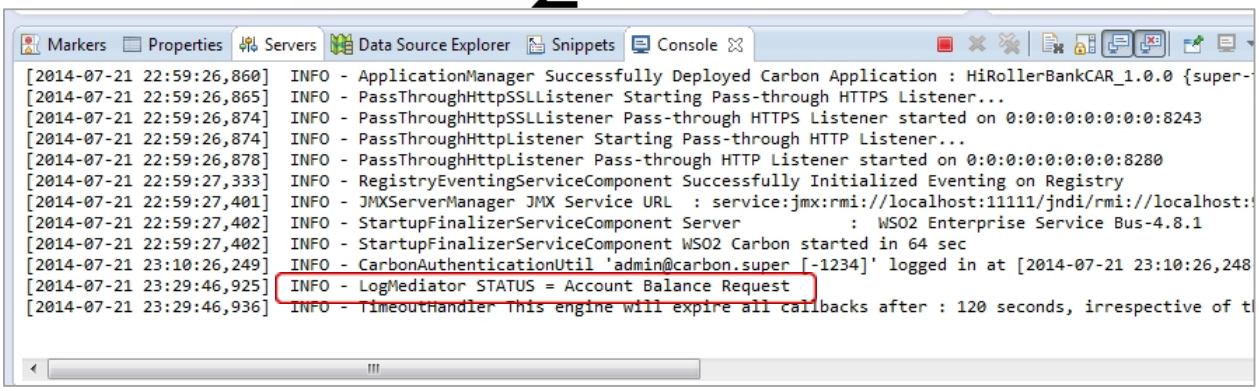


3. You have just hit the accounts service directly and gotten an account balance.

4. Now modify the SoapUI project to connect to our proxy service in WSO2 ESB
   a. Set the URL as **http://localhost:8280/services/AccountProxy**(verify this by exploring the *AccountProxy* service in ESB)
   b. Send the request again and look for the exact same returned account balance



5. Back in Dev Studio, in the console tab, you can verify if your Proxy Service logged a message (LogMediator STATUS = Account Balance Request)

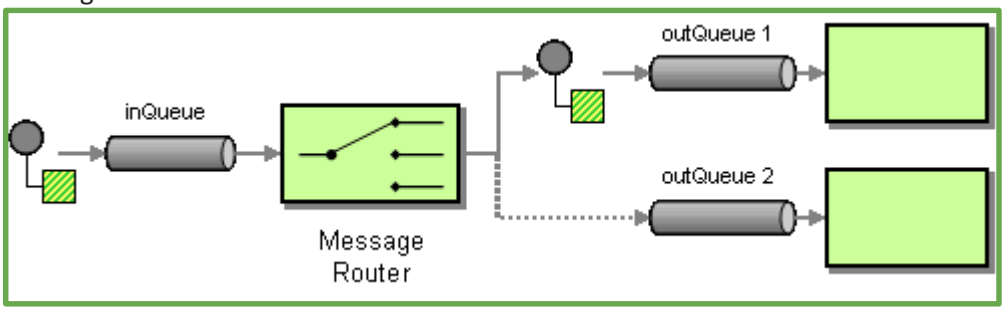

Markers
Properties
Servers
Data Source Explorer
Snippets
Console
[2014-07-21 22:59:26,860] INFO - ApplicationManager Successfully Deployed Carbon Application : HiRollerBankCAR_1.0.0 {super-
[2014-07-21 22:59:26,865] INFO - PassThroughHttpSSLListener Starting Pass-through HTTPS Listener...
[2014-07-21 22:59:26,874] INFO - PassThroughHttpSSLListener Pass-through HTTPS Listener started on 0:0:0:0:0:0:0:0:8243
[2014-07-21 22:59:26,874] INFO - PassThroughHttpListener Starting Pass-through HTTP Listener...
[2014-07-21 22:59:26,878] INFO - PassThroughHttpListener Pass-through HTTP Listener started on 0:0:0:0:0:0:0:0:8280
[2014-07-21 22:59:27,333] INFO - RegistryEventingServiceComponent Successfully Initialized Eventing on Registry
[2014-07-21 22:59:27,401] INFO - JMXServerManager JMX Service URL : service:jmx:rmi://localhost:11111/jndi/rmi://localhost:
[2014-07-21 22:59:27,402] INFO - StartupFinalizerServiceComponent Server : WSO2 Enterprise Service Bus-4.8.1
[2014-07-21 22:59:27,402] INFO - StartupFinalizerServiceComponent WSO2 Carbon started in 64 sec
[2014-07-21 23:10:26,249] INFO - CarbonAuthenticationUtil 'admin@carbon.super [-1234]' logged in at [2014-07-21 23:10:26,248
[2014-07-21 23:29:46,925] INFO - LogMediator STATUS = Account Balance Request
[2014-07-21 23:29:46,936] INFO - TimeoutHandler This engine will expire all callbacks after : 120 seconds, irrespective of t

LAB 02: WSO2 ESB Functionality – Switch Mediator

Training Objective: Using DevStudio, practice configuration of core functionality: Switch Mediator

Business Scenario: As HiRollerBank adds more account types (CHK, SAV) to their services, they would like to route the various requests to the appropriate service, based on the account type. In this exercise, you will perform content-based routing to send the message to a specified service.

Enterprise Integration Pattern:
Message Router



High-level Steps:

- Use a Switch Mediator to check if incoming message is a CHK or a SAV account request.
- Direct messages to a CHK Log Mediator ("CHK balance Request") or to a SAV Log Mediator ("SAV balance Request")
- Redeploy your changes and test your configuration in SoapUI

Message Formats from Client:

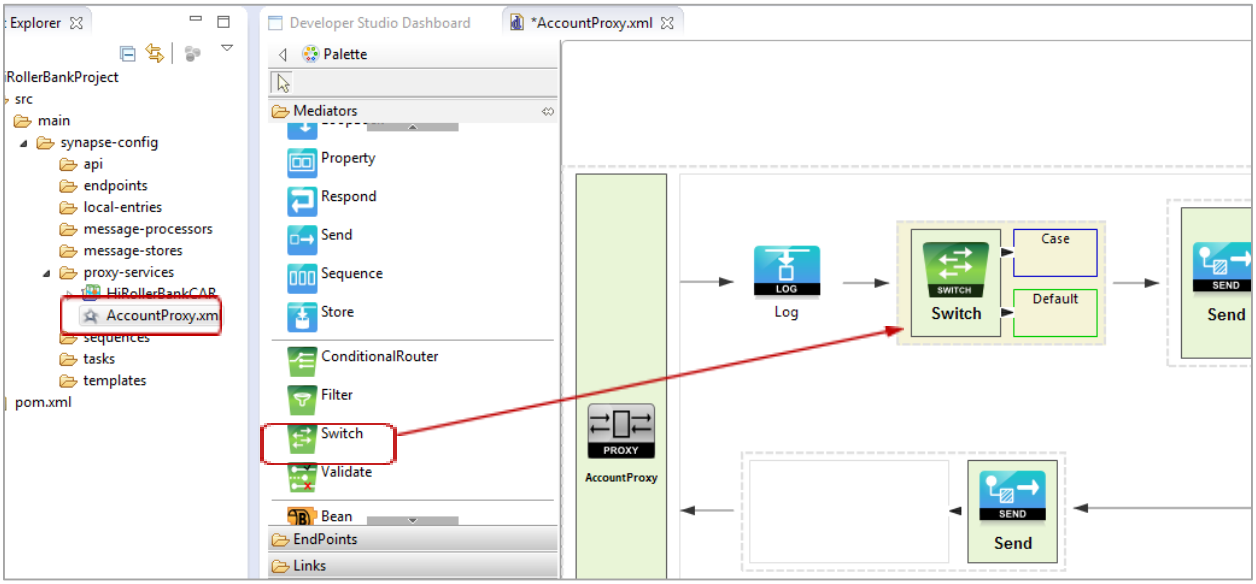
CHK Balance	SAV Balance
<soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:hir="http://hirollerbankws.training.wso2.com/"> <soapenv:Header/> <soapenv:Body> **<hir:checkBalance>** **<!--Optional:-->** **<accountNo>1111</accountNo>** **</hir:checkBalance>** </soapenv:Body> </soapenv:Envelope>	<soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:hir="http://hirollerbankws.training.wso2.com/"> <soapenv:Header/> <soapenv:Body> **<hir:saveBalance>** **<!--Optional:-->** **<accountNo>2222</accountNo>** **</hir:saveBalance>** </soapenv:Body> </soapenv:Envelope>

Detailed Instructions:

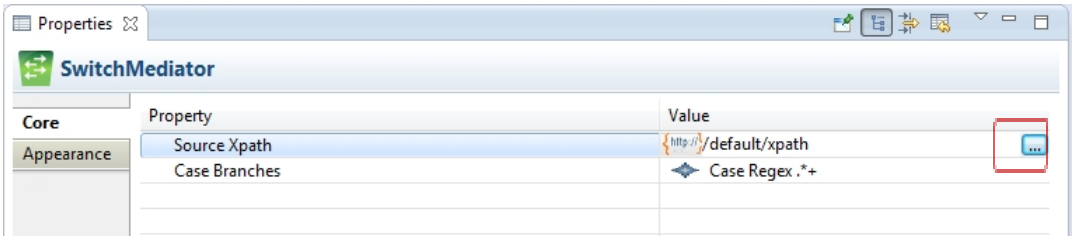
Add a Switch Mediator that has 2 cases: A CHK and a SAV Log Mediator

1. If you haven't done so, in Dev Studio switch to ESB Graphical perspective by clicking on top-right menu

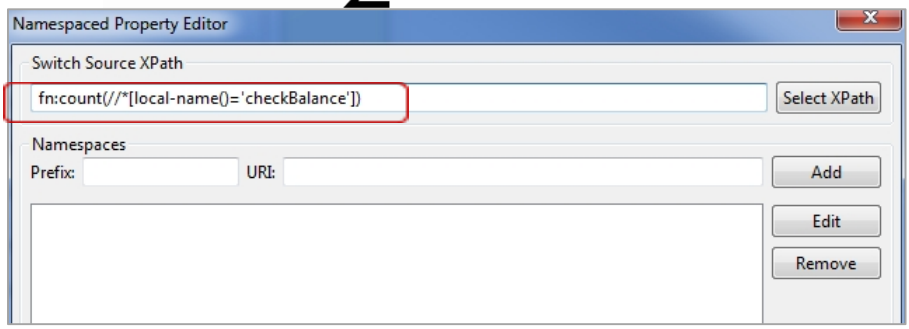
2. Open the *AccountProxy* by double-clicking on *AccountProxy.xml* in your left-nav bar.
3. In *Design* view, click on the *Mediators* folder to list the catalog of available mediators
4. Drag the *Switch* mediator between the existing *Log* mediator and the *Send* mediator



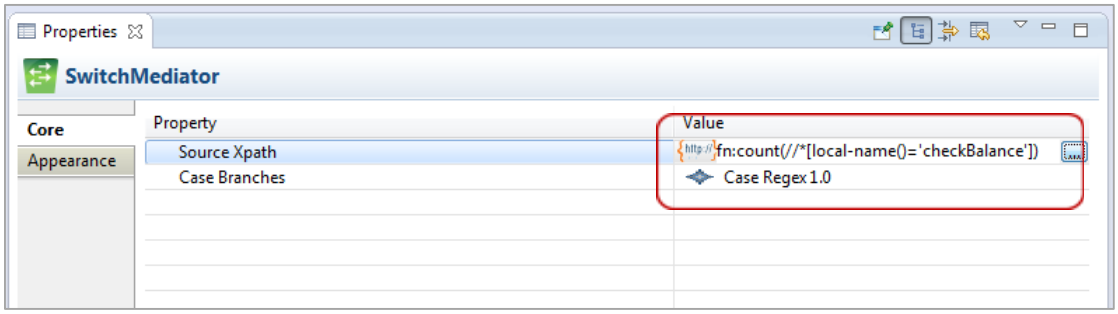
5. Configure the switch condition
    a. Right-click on Switch mediator and select *Show Properties View*
    b. In *Properties* view, click on … to expand the*Source Xpath*definition



    c. Set the Switch Source *Xpath* to **fn:count(//*[local-name()='checkBalance'])**

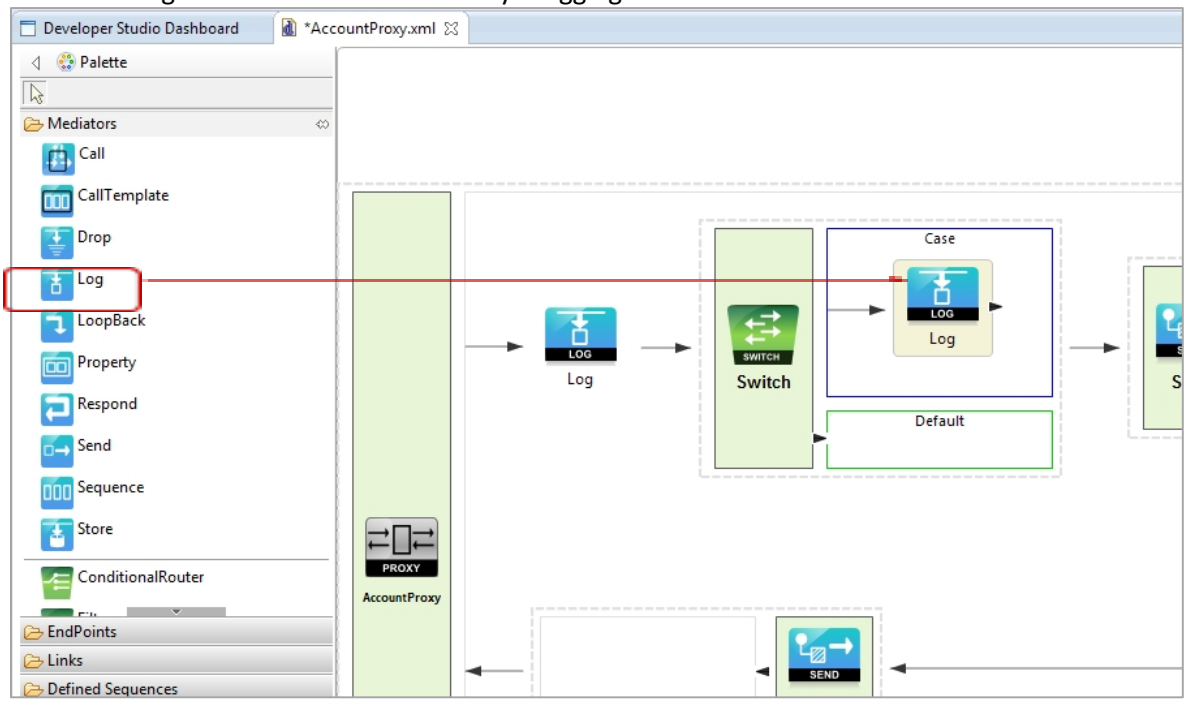


d. Similarly, set the *Case Regex* to **1.0**



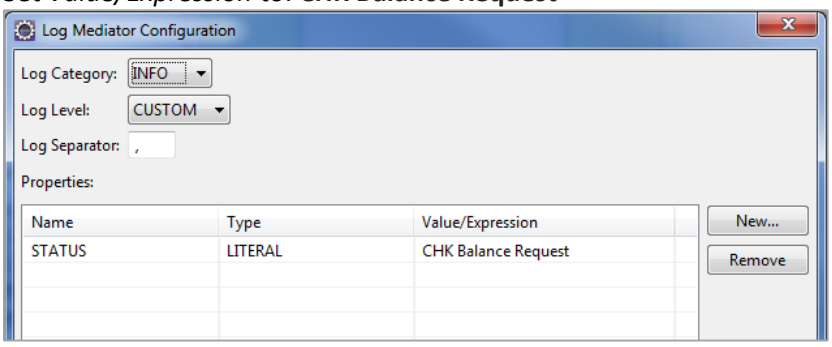
These settings check for the word 'checkBalance' and sets the expression to 1.0 if correct.

6. Add a Log Mediator to *Case* branch by dragging it from the *Mediators* folder

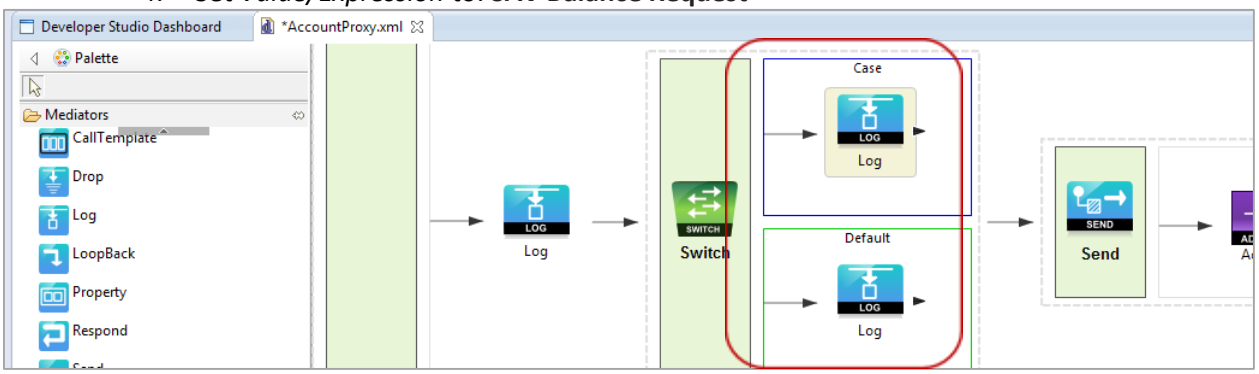


a. Right-click on Log mediator and select *Configure…*
b. Set *Log Level* to: **CUSTOM** and Log Separator to: **,**

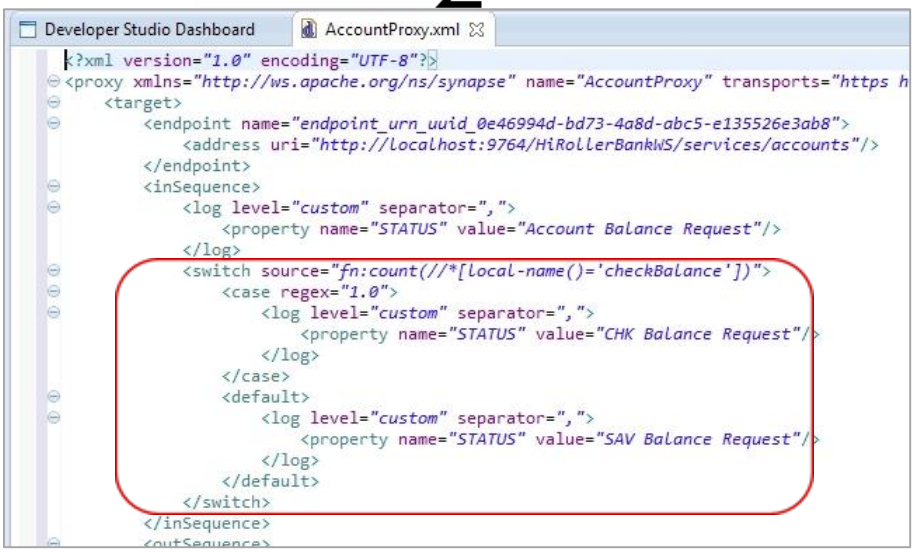
c. Click on *New* to add a new custom property
d. Set *Property Name* to: **STATUS**
e. Set *Type* to: **LITERAL**
f. Set *Value/Expression* to: **CHK Balance Request**



7. Similarly, add a Log Mediator to *Default* branch by dragging it from the *Mediators* folder
   a. Right-click on Log mediator and select *Configure…*
   b. Set *Log Level* to: **CUSTOM** and Log Separator to: **,**
   c. Click on *New* to add a new custom property
   d. Set *Property Name* to: **STATUS**
   e. Set *Type* to: **LITERAL**
   f. Set *Value/Expression* to: **SAV Balance Request**



8. View AccountProxy in Source mode and note the <switch> and <log> tags

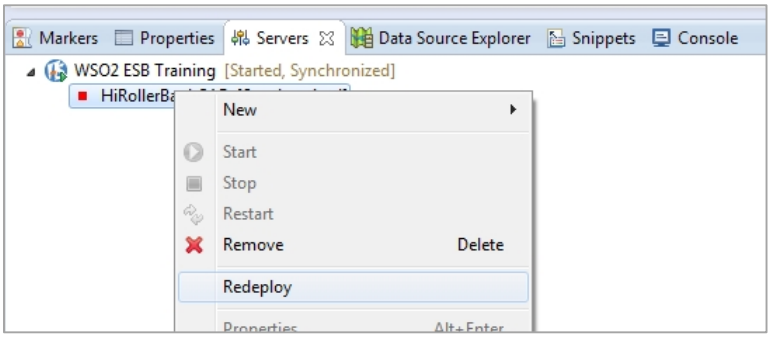


9. Save the changes to the Proxy Service by closing the *AccountProxy.xml* tab

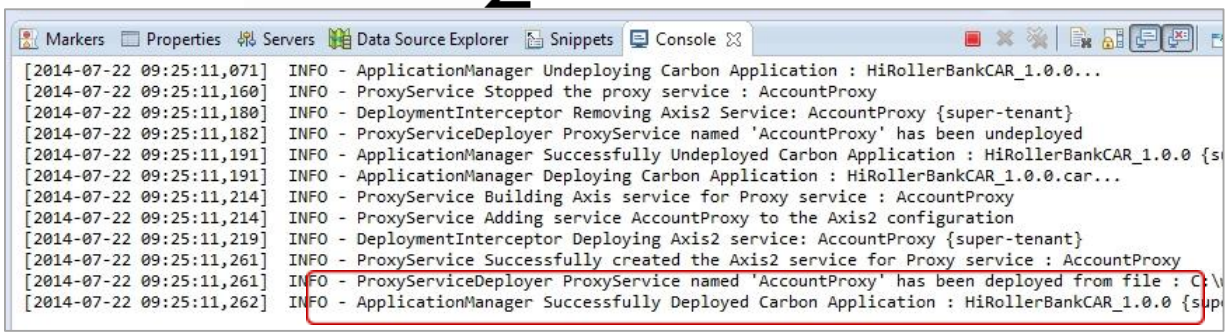
Redeploy the Carbon Application to WSO2 ESB and Test Changes

1. Switch to *Java* Perspective (Top-right corner)
2. Click on the Servers tab in the bottom half of the screen
3. Expand the *WSO2 ESB Training* server
4. If server is stopped, right-click on it and select *Start*
5. Right-click the *HiRollerBankCAR* and click *Redeploy*

Note : you can have the Servers view in your WSO2 perspective. Go to the Properties Tab and select in eclipse menu the "*Window/Show View/Other*" then type *server* and choose *Servers* to make the Servers tab appear in your perspective.

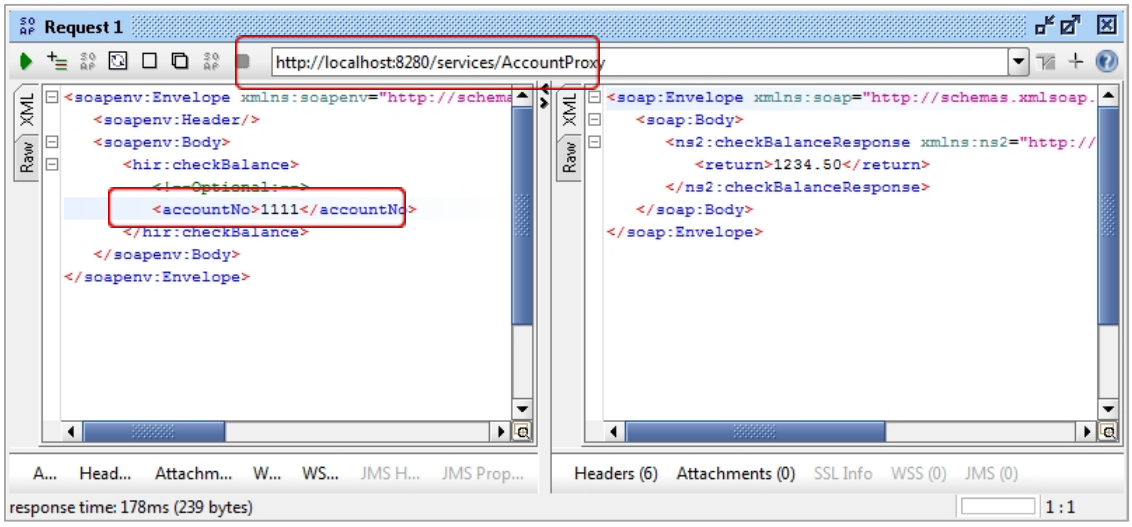


6. In the *Console* tab, verify logs stating that *AccountProxy* has been redeployed.

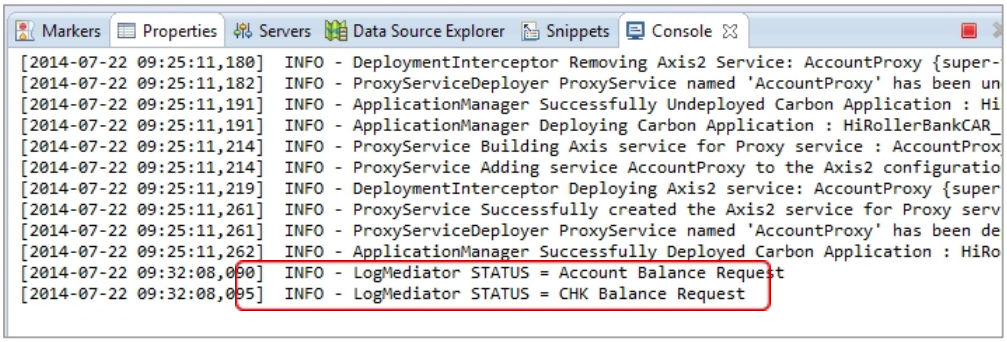


You can also check your redeployed changes in the *ESB Management Console (http://localhost:9443)*: Under *Services*, click on *List* and find your *AccountProxy* service. Click on *Source View* and notice the *<log>* and *<switch>* tags.

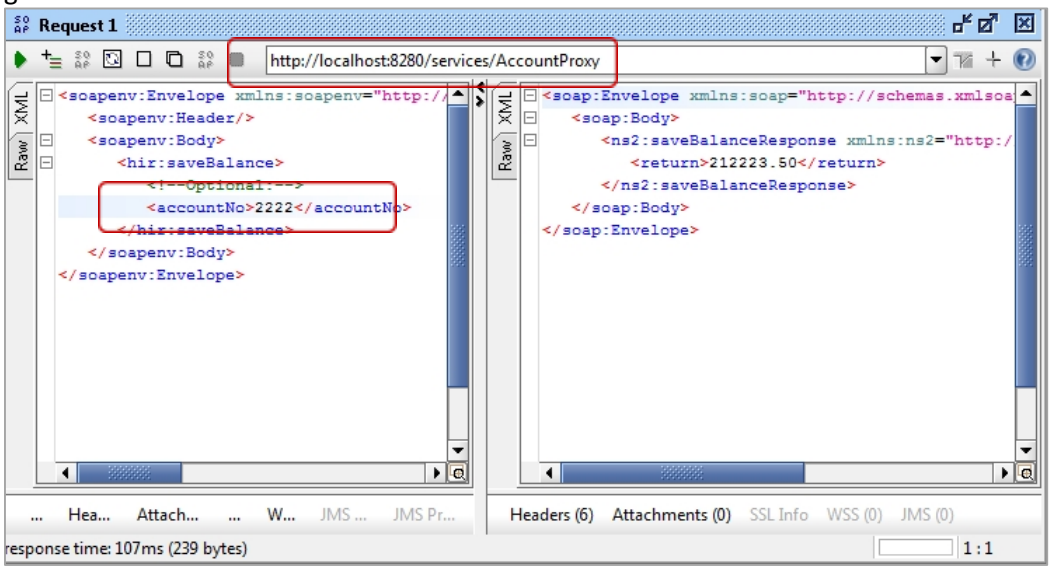
7. Test changes by generating requests for both *checkBalance* and *saveBalance*using SoapUI
    a. In soapUI,right-click on checkBalance and create a new request
    b. Make sure you are hitting **http://localhost:8280/services/AccountProxy**
    c. Set the accountNo to **1111**



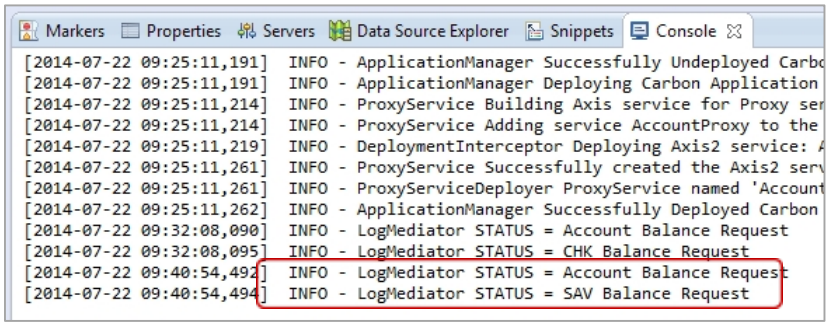
    d. In DevStudio, check your logs in the Console tab



e. Back in SoapUI, right-click on savBalance and create a new request
f. Make sure you are hitting **http://localhost:8280/services/AccountProxy**
g. Set the accountNo to **2222**



h. Lastly, back in DevStudio, check your logs in the Console tab

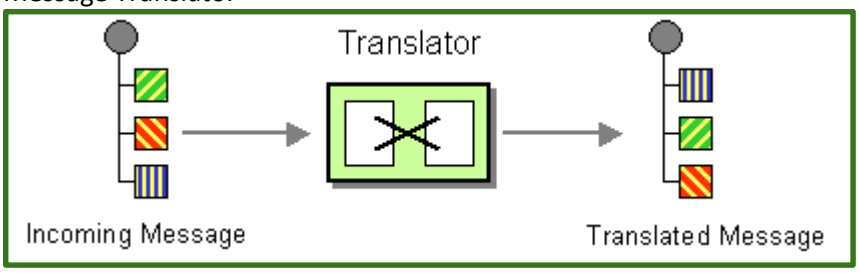


LAB 03: WSO2 ESB Functionality – Message Transformation

Training Objective: Practice configuration of core functionality: Message Transformation

Business Scenario:
In order to accommodate its customers' needs, HiRollerBank will need to change the message format that it accepts from its clients. Since HiRollerBank does not want to change its core service, it will use a proxy service to transform the new-format-messages sent by clients to the existing format that the existing service is expecting.

Enterprise Integration Pattern:
Message Translator



High-level Steps:

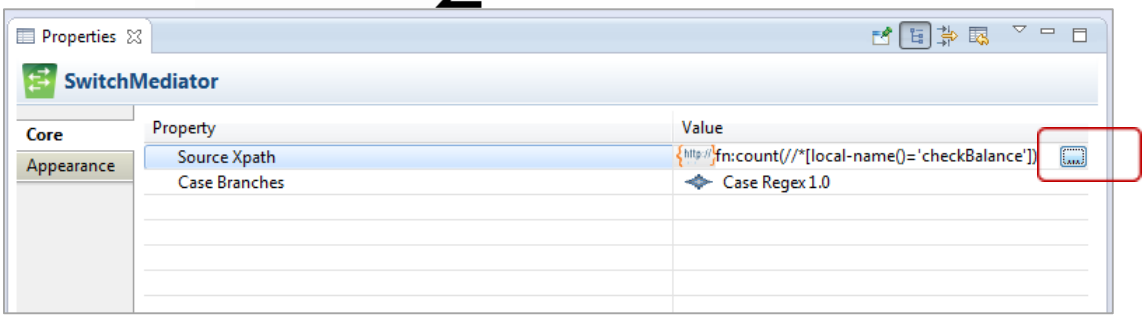
- Edit the *Switch* mediator to recognize the new format
- Use the *PayloadFactory* mediator to change it to the required format
- Redeploy your changes and test your configuration in SoapUI

Old Message format from Client	New Message Format from Client
<soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:hir="http://hirollerbankws.training.wso2.com/"> <soapenv:Header/> <soapenv:Body> **<hir:checkBalance>** **<!--Optional:-->** **<accountNo>1111</accountNo>** **</hir:checkBalance>** </soapenv:Body> </soapenv:Envelope>	<soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:hir="http://hirollerbankws.training.wso2.com/"> <soapenv:Header/> <soapenv:Body> **<hir:balance>** **<hir:type>CHK</hir:type>** **<hir:account>1111</hir:account>** **</hir:balance>** </soapenv:Body> </soapenv:Envelope>

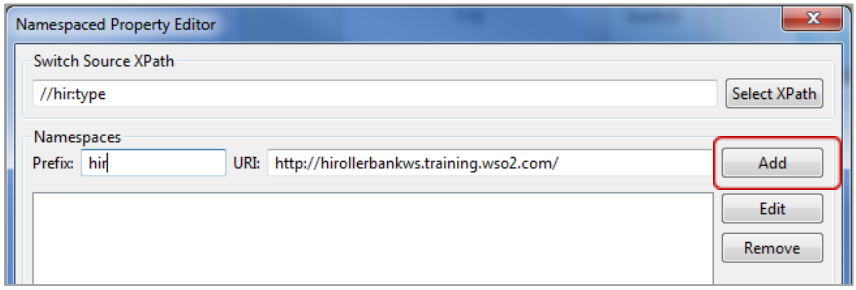
Detailed Instructions:

Edit your Switch mediator and add PayloadFactory mediators to transform message formats

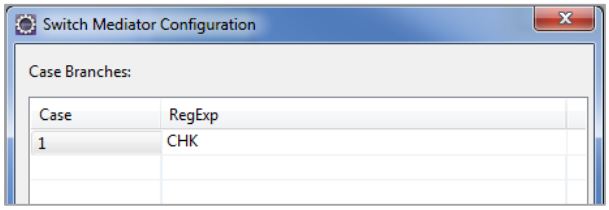
1. In SoapUI, switch to ESB Graphical perspective (top-right corner menu)
2. Open the *AccountProxy* configuration by double-clicking on *AccountProxy.xml*
3. Click on the Switch mediator to edit its properties
4. Edit the Switch condition to accept a new message format



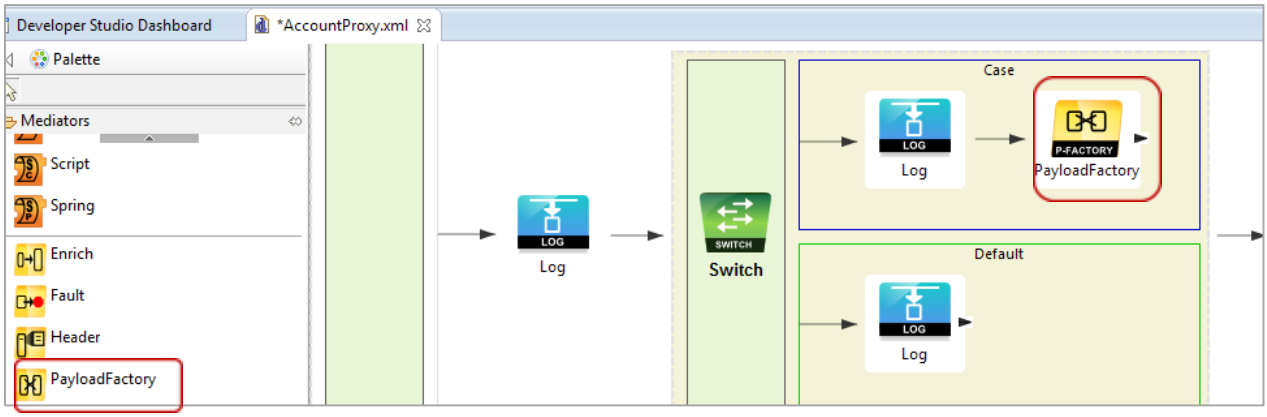
a. Set the *Select Xpath* to **//hir:type**
b. In *Namespaces*, set the *Prefix*to**hir**
c. Set the *URI* to**http://hirollerbankws.training.wso2.com/**
d. Click *Add* to add you new Namespace



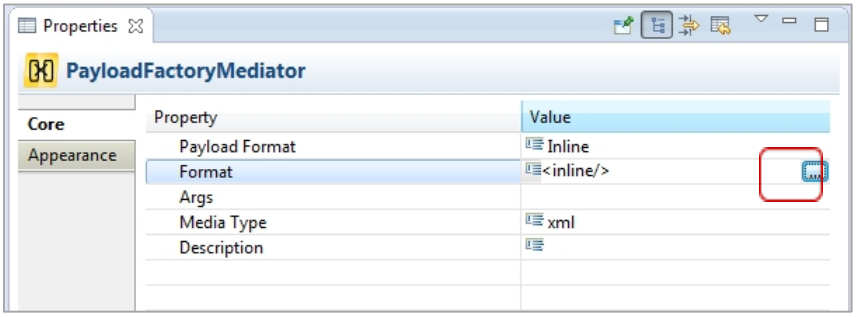
e. Click *OK* to save the configuration
f. Similarly, edit the *Case Regex*to**CHK**



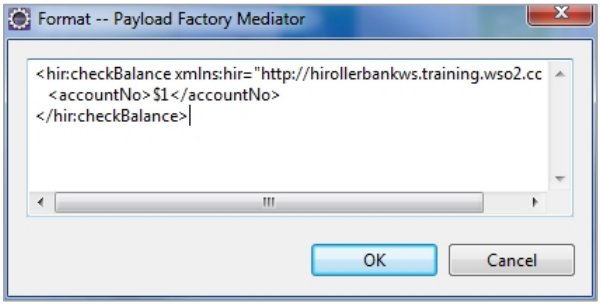
5. In *Design* view, click on the *Mediators* folder to list the catalog of available mediators
6. Drag the *PayloadFactory*mediatorafter the *Log* mediator in the *Case* branch



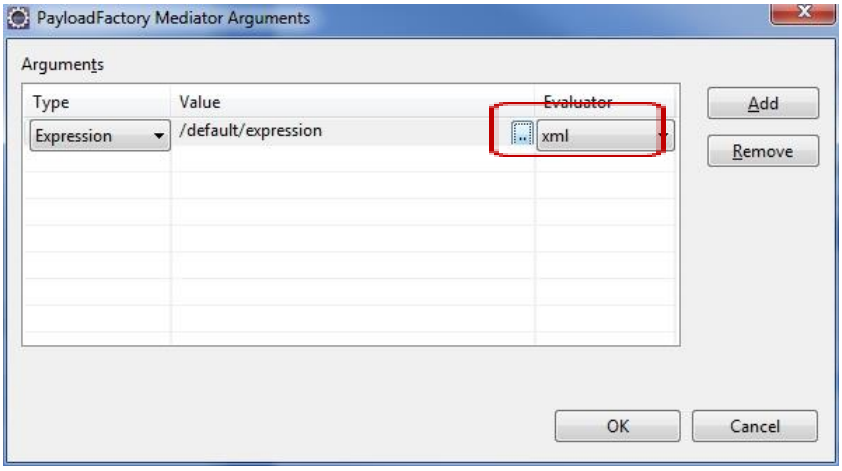
7. Right-click on the PayloadFactory icon to display the properties tab at the bottom and set the following:



   a. Set *Format* to:
   <hir:checkBalancexmlns:hir="http://hirollerbankws.training.wso2.com/">
   <accountNo>$1</accountNo>
   </hir:checkBalance>



   b. Open the *Args* section and click on *Add* to add a new Argument
   c. Set the *Type* to *Expression*, and click on … to edit the Value

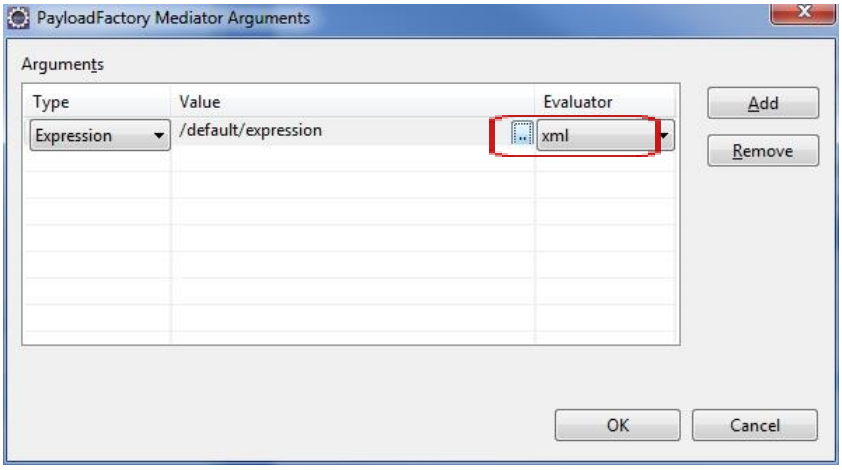


   d. In *Namespaced Property Editor*, set *Argument Expression* to **//hir:account**
   e. In *Namespaces*, set the *Prefix* to **hir**
   f. Set the *URI* to **http://hirollerbankws.training.wso2.com/**

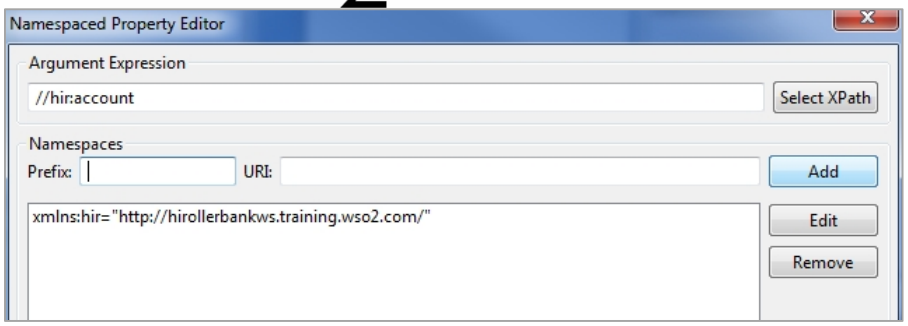
g. Click *Add* to add you new Namespace

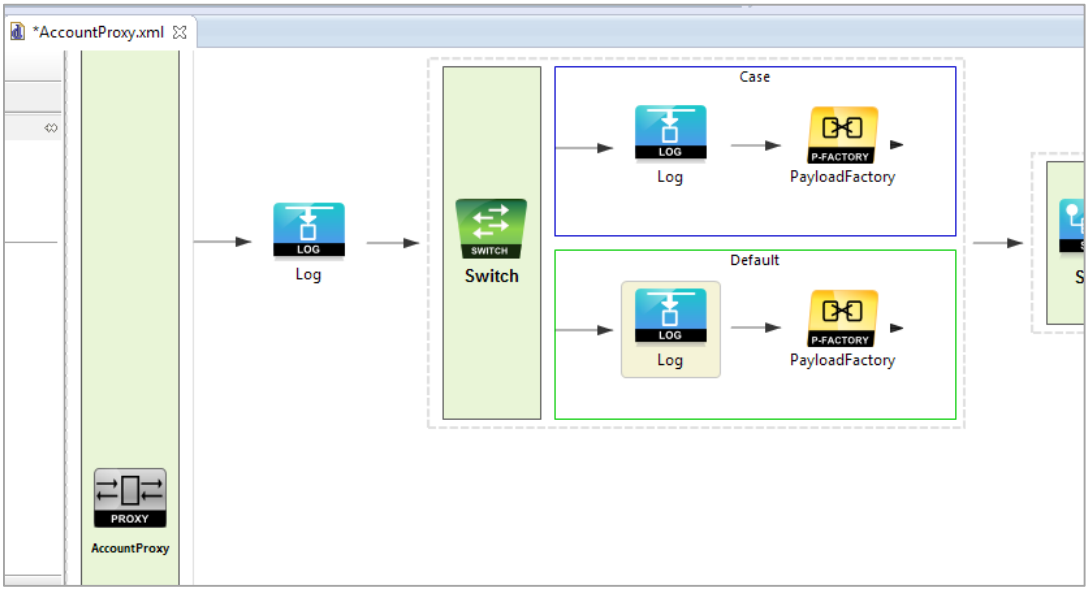
8. In a similar way, drag the *PayloadFactory*mediator after the *Log* mediator in the *Default* branch
   a. Set the *Format* with the following payload
   <hir:saveBalancexmlns:hir="http://hirollerbankws.training.wso2.com/">
   <accountNo>$1</accountNo>
   </hir:saveBalance>

   b. Open the *Args* section and click on *Add* to add a new Argument
   c. Set the *Type* to *Expression*, and click on … to edit the Value



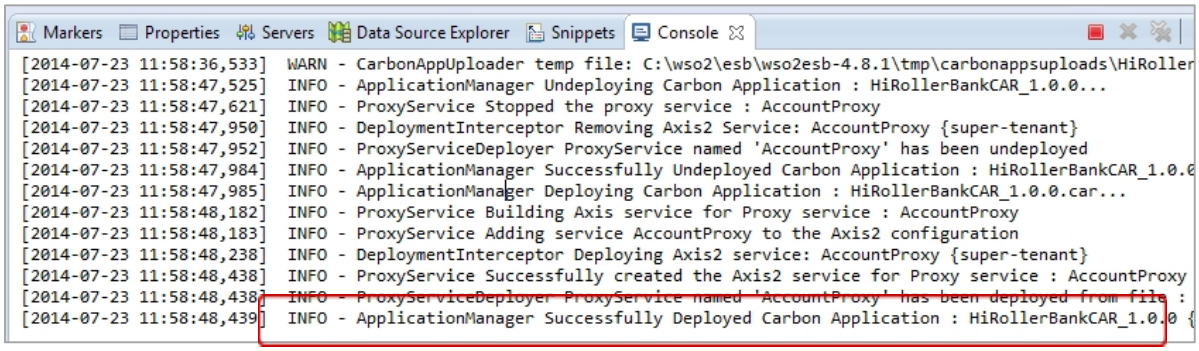
   d. In *Namespaced Property Editor*, set *Argument Expression* to **//hir:account**
   e. In *Namespaces*, set the *Prefix* to **hir**
   f. Set the *URI* to**http://hirollerbankws.training.wso2.com/**
   g. Click *Add* to add you new Namespace





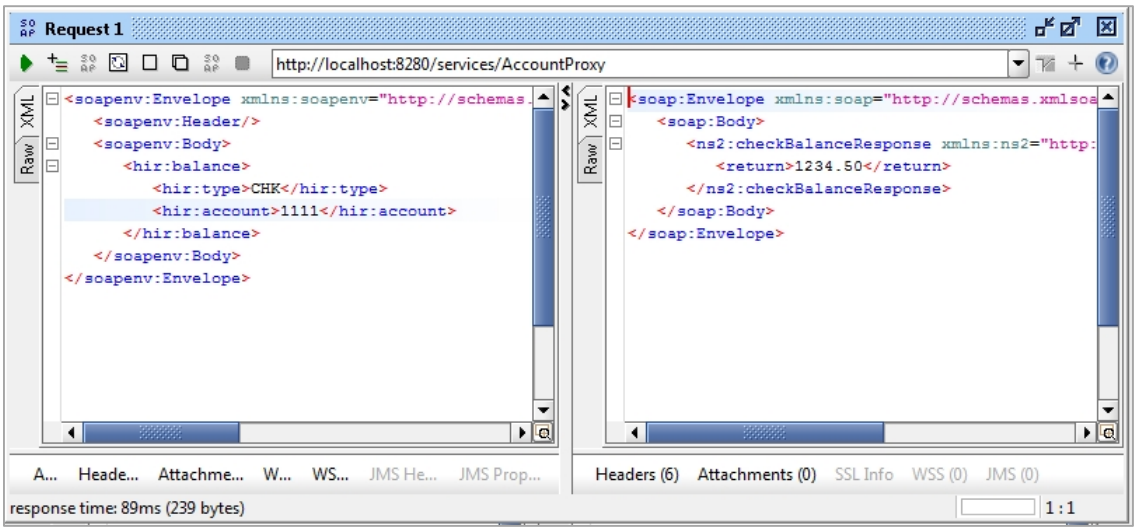
Redeploy the Carbon Application and Test Changes

1. Close your AccountProxy.xml and save changes
2. Switch to Java EE Perspective (Top-right corner)
3. Click on the Servers tab in the bottom half of the screen
4. Right-click the *HiRollerBankCAR* application under the WSO2 ESB Training server and click *Redeploy*



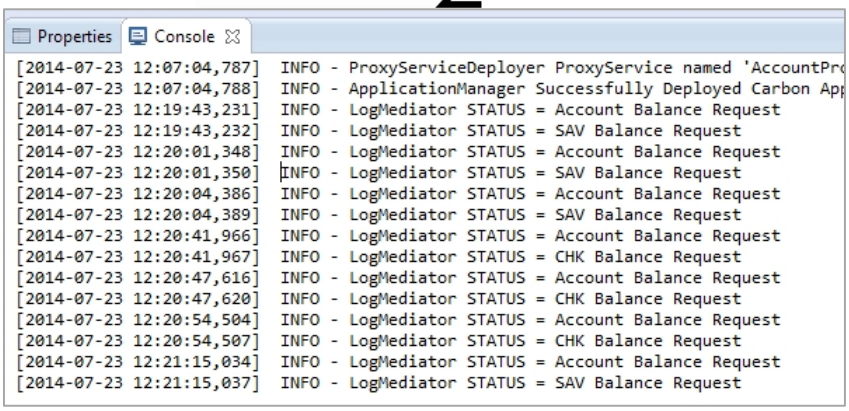
5. In SoapUI, generate a request with the following message format:

<soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:hir="http://hirollerbankws.training.wso2.com/">
<soapenv:Header/>
<soapenv:Body>
<hir:balance>
<hir:type>CHK</hir:type>
<hir:account>1111</hir:account>
</hir:balance>
</soapenv:Body>
</soapenv:Envelope>



6. You can generate a savings request with the following format:
    a. <soapenv:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:hir="http://hirollerbankws.training.wso2.com/">
    <soapenv:Header/>
    <soapenv:Body>
    <hir:balance>
    <hir:type>SAV</hir:type>
    <hir:account>2222</hir:account>
    </hir:balance>
    </soapenv:Body>
    </soapenv:Envelope>

7. Lastly, check your logs for your latest requests:


Properties
Console
[2014-07-23 12:07:04,787] INFO - ProxyServiceDeployer ProxyService named 'AccountPr
[2014-07-23 12:07:04,788] INFO - ApplicationManager Successfully Deployed Carbon Ap
[2014-07-23 12:19:43,231] INFO - LogMediator STATUS = Account Balance Request
[2014-07-23 12:19:43,232] INFO - LogMediator STATUS = SAV Balance Request
[2014-07-23 12:20:01,348] INFO - LogMediator STATUS = Account Balance Request
[2014-07-23 12:20:01,350] INFO - LogMediator STATUS = SAV Balance Request
[2014-07-23 12:20:04,386] INFO - LogMediator STATUS = Account Balance Request
[2014-07-23 12:20:04,389] INFO - LogMediator STATUS = SAV Balance Request
[2014-07-23 12:20:41,966] INFO - LogMediator STATUS = Account Balance Request
[2014-07-23 12:20:41,967] INFO - LogMediator STATUS = CHK Balance Request
[2014-07-23 12:20:47,616] INFO - LogMediator STATUS = Account Balance Request
[2014-07-23 12:20:47,620] INFO - LogMediator STATUS = CHK Balance Request
[2014-07-23 12:20:54,504] INFO - LogMediator STATUS = Account Balance Request
[2014-07-23 12:20:54,507] INFO - LogMediator STATUS = CHK Balance Request
[2014-07-23 12:21:15,034] INFO - LogMediator STATUS = Account Balance Request
[2014-07-23 12:21:15,037] INFO - LogMediator STATUS = SAV Balance Request

LAB 04: WSO2 ESB Functionality – Service Chaining

Training Objective: Practice configuration of core functionality: Service Chaining

Business Scenario: HiRollerBank wants to allow its members to use their mobile phones to find the nearest ATM, according to their location. GPS coordinates received from the mobile phone (client) will be sent to a GeoLocation service to determine the correct zip code. In turn, this zip code will be used against an ATMLocator service to identify all ATMs located in the specified zip code.

WSO2 ESB orchestrates the requests and responses from the various services - to the end user (client), it all looks like one request with one response.Following the steps below, configure sequences with appropriate mediators to achieve the service chaining necessary to provide the right ATM locations to customers.

Backend Services	Location
GeoLocationService Receives GPS coordinates and returns US Postal zip code.	http://localhost:9764/HiRollerBankWS/services/geolocation
ATMLocatorService Receives a zip code and returns street addresses for all ATM in that zip code.	http://localhost:9764/HiRollerBankWS/services/atmlocator

High-level Steps:

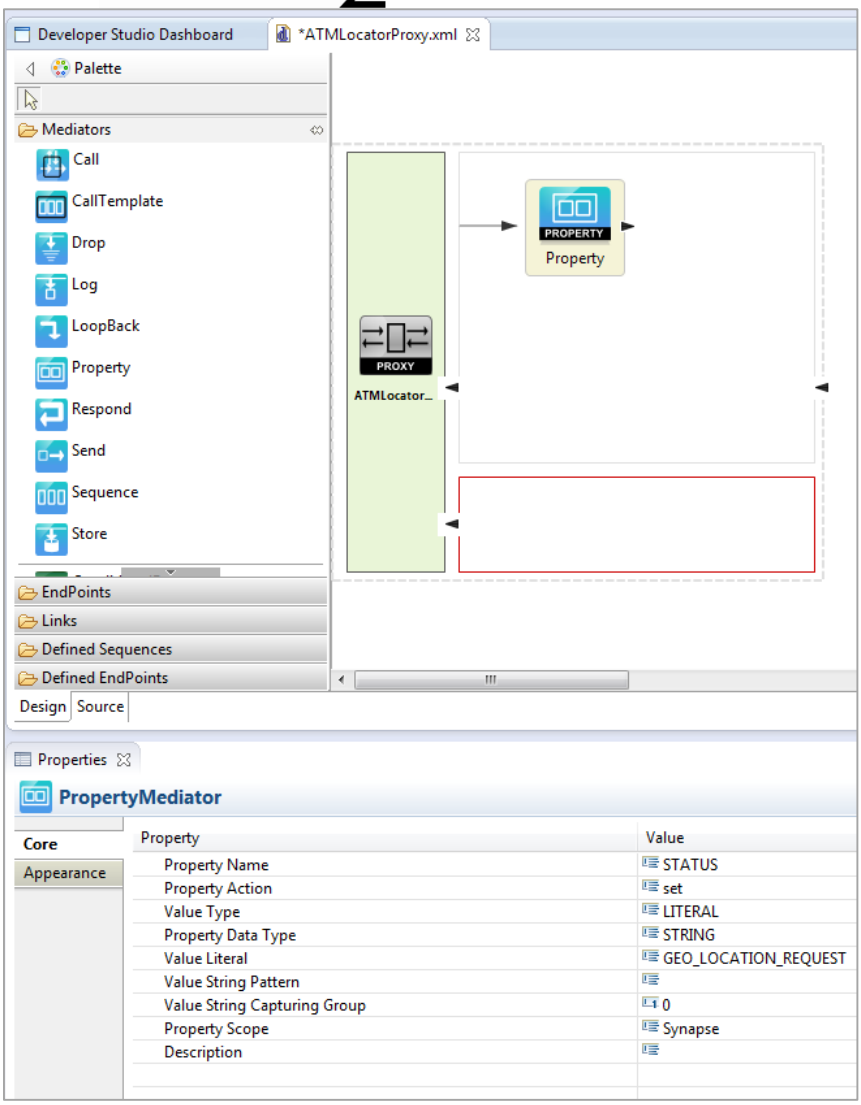
- Create a new proxy service that will serve to coordinate requests and responses.
- Send the request to the first service
- When receiving a response, check whether we still need to make another request to another service, or send answer back to client (use a *Switch* mediator)
- Add you new configuration to your existing CAR file
- Redeploy configuration and test

Detailed Instructions:

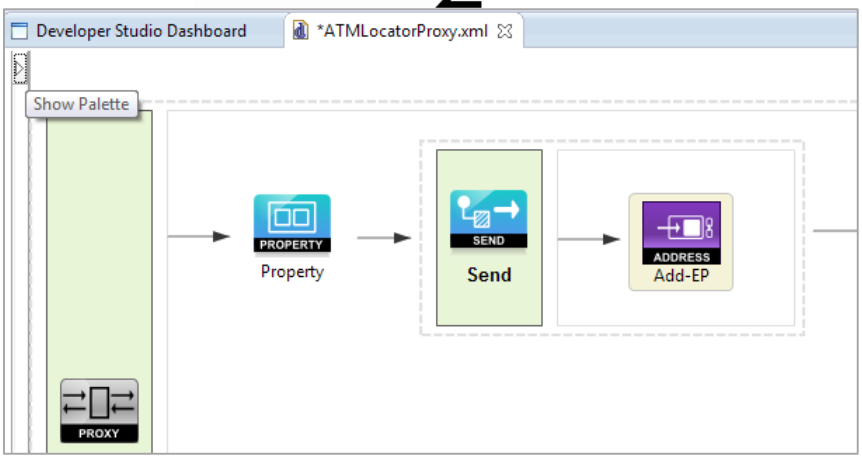
Create the Proxy Service

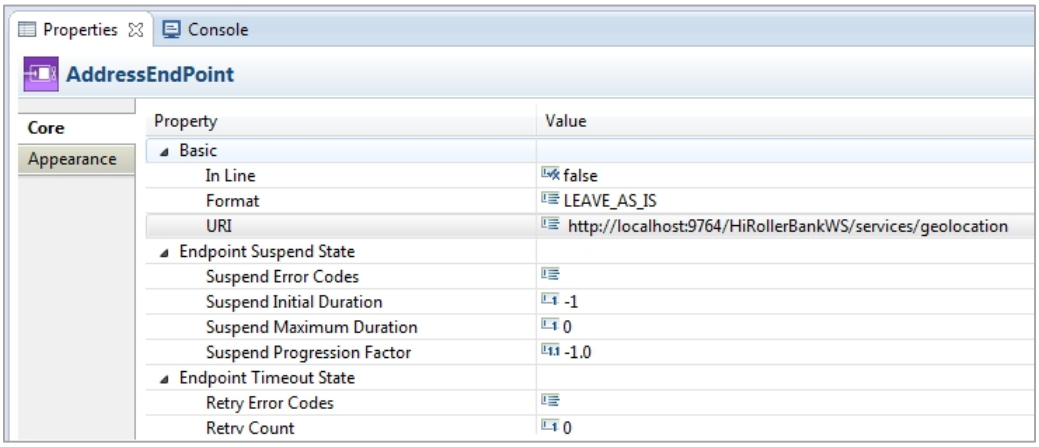
1. In DevStudio, in left-nav bar, right-click on *HiRollerBankProject*, select *New ➔ Proxy Service*
    a. Select *Create a New Proxy Service*
    b. Set *Proxy Service Name* to: **ATMLocatorProxy**
    c. Set *Proxy Service Type* to: **Custom Proxy**
    d. Click *Finish* to build your new proxy service
    e. You can view your newly created *ATMLocatorProxy.xml* in *Design* and *Source* views

2. Drag a *Property*mediator to the *InSequence*. This will be used to save the chaining state
    a. Set *Property Name* to: **STATUS**
    b. Set Value *Type* to: **LITERAL**
    c. Set *Value Literal* to: **GEO_LOCATION_REQUEST**

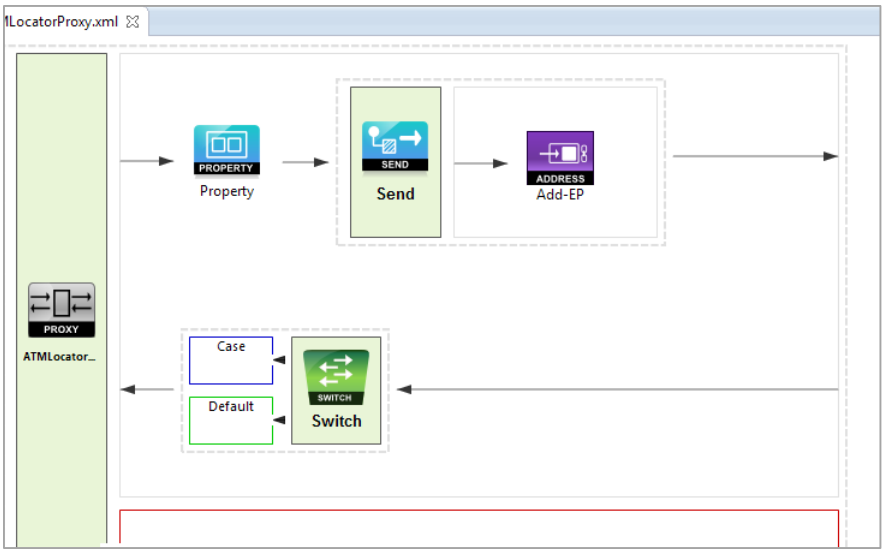


3. Drag a *Send* mediator after the *Property* mediator
4. Drag an *Address Endpoint* inside the *Send* mediator
    a. set its *URI*to **http://localhost:9764/HiRollerBankWS/services/geolocation**
    b. Leave all other *Address Endpoint* properties with their default values

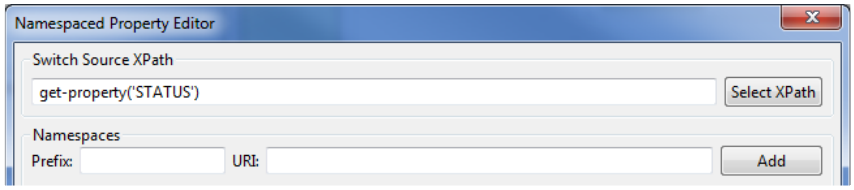




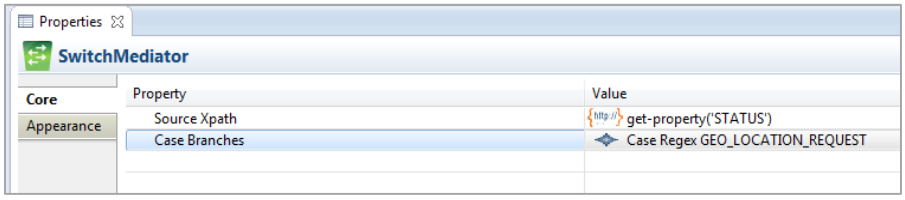
5. Now add a *Switch* mediator in the *OutSequence*



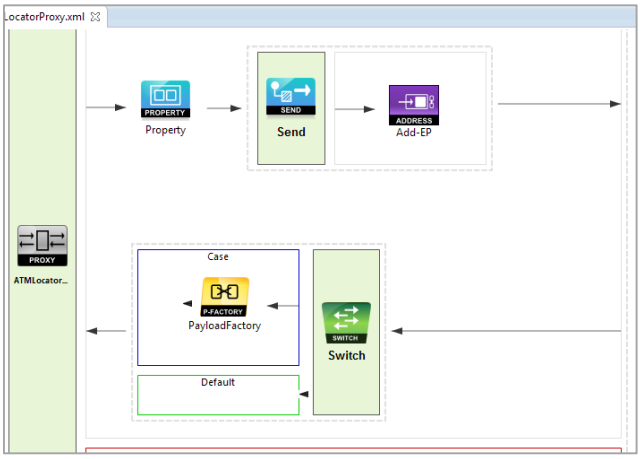
   a. Set *SourceXpath* to **get-property('STATUS')**



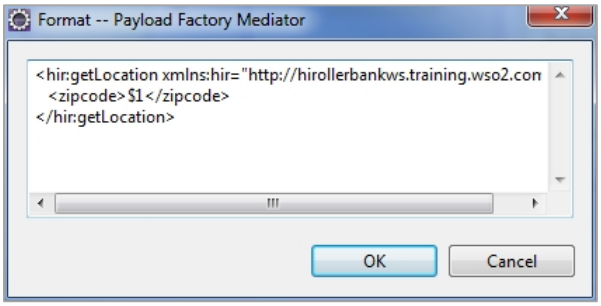
b. Click *OK* to save the configuration
c. Set *Case Regex*to **GEO_LOCATION_REQUEST**



6. In the *Case* branch of the *Switch* mediator, add a *PayloadFactory* mediator

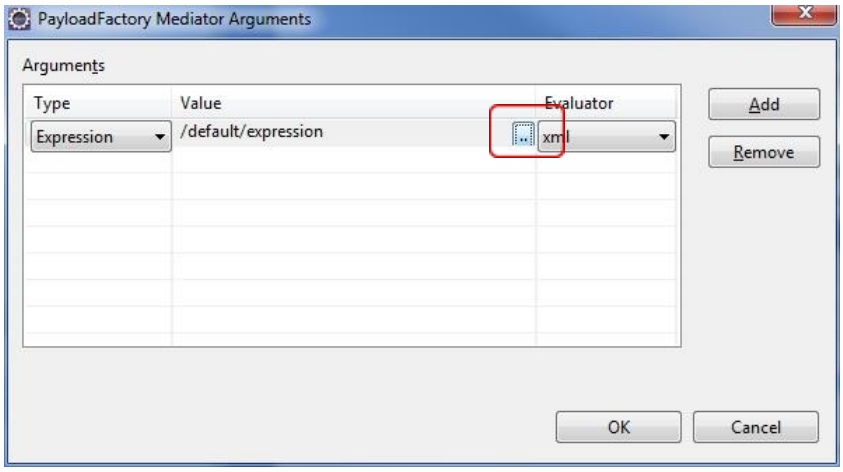


a. Set the *Format* with the following payload
<hir:getLocationxmlns:hir="http://hirollerbankws.training.wso2.com/">
<zipcode>$1</zipcode>
</hir:getLocation>

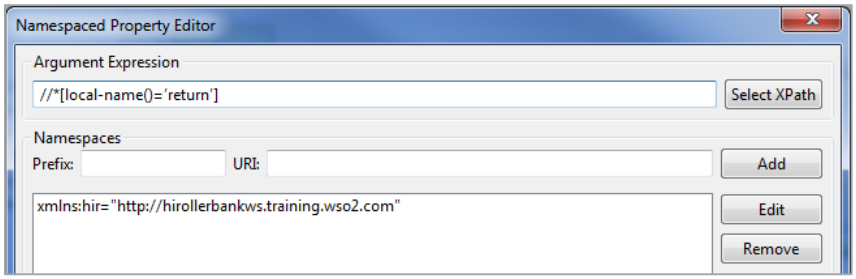


b. Open the *Args* section and click on *Add*to add a new Argument

c. Set the *Type* to *Expression*, and click on … to edit the Value

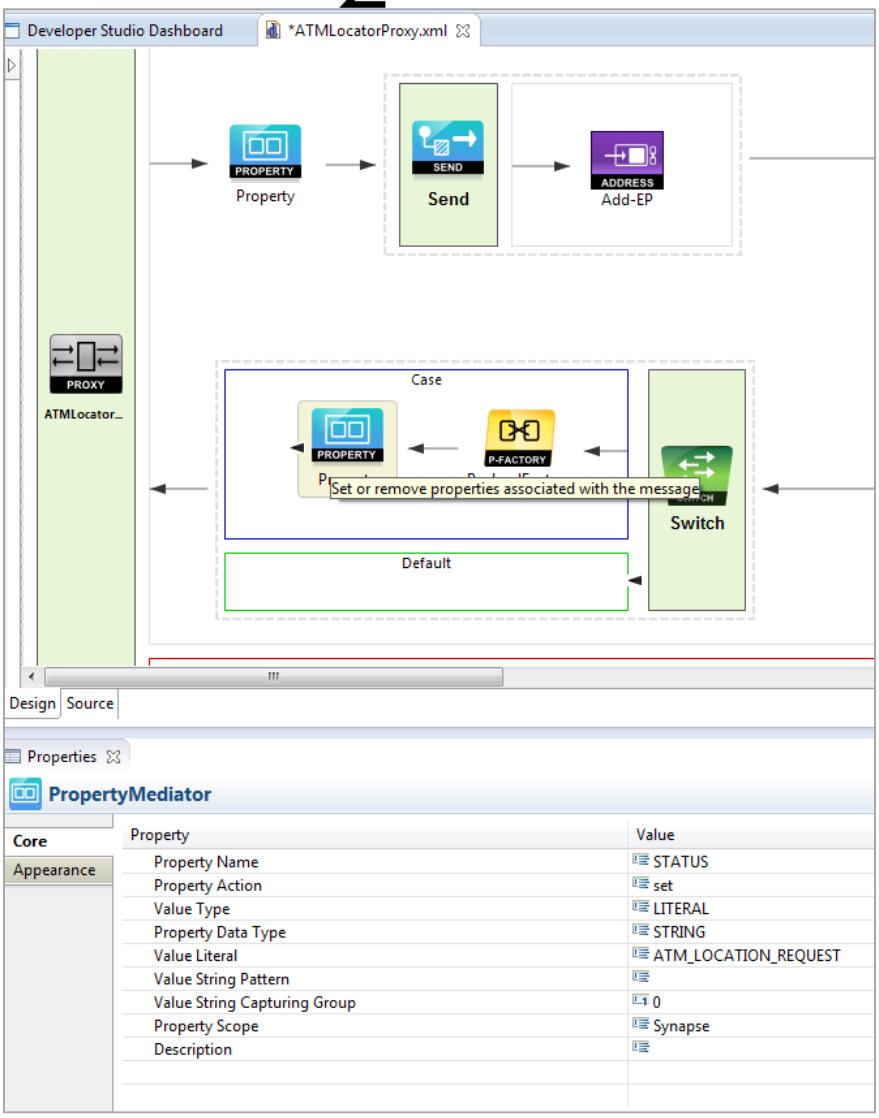


d. Set *Argument Expression* to **//*[local-name()='return']**
e. In *Namespaces*, set the *Prefix*to**hir**
f. Set the *URI* to**http://hirollerbankws.training.wso2.com/**
g. Click *Add* to add you new Namespace

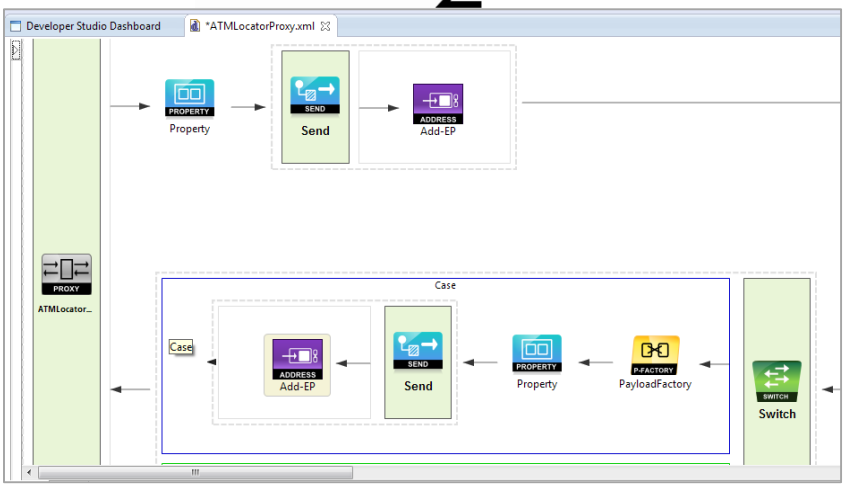


h. Click*OK* to set your Argument

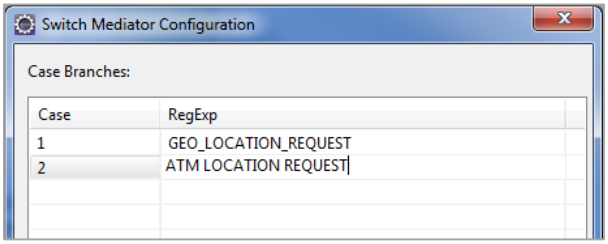
7. Add a new *Property*mediator after the *PayloadFactory* mediator inside the Case block. This will serve to save the chaining state.
   a. Set *Property Name* to: **STATUS**
   b. Set Value *Type* to: **LITERAL**
   c. Set *Value Literal* to: **ATM_LOCATION_REQUEST**



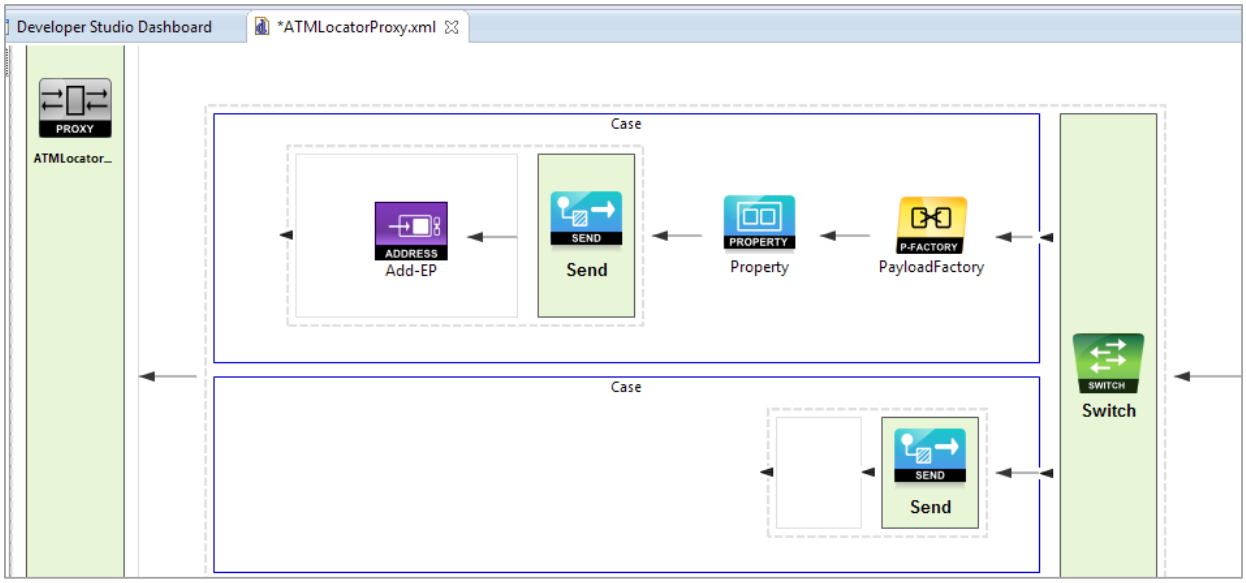
8. Drag a *Send* mediator inside the Case block, after the *Property* mediator
9. Drag an*Address Endpoint* inside the *Send* mediator
    a. Set its URI to **http://localhost:9764/HiRollerBankWS/services/atmlocator**



10. Right-click on the *Switch* mediator, select *Add/Remove Case…*and set *Number of Branches* to **2**
11. Right-click again and select *Configure* to set *RegEx* to **ATM_LOCATION_REQUEST** for Case 2



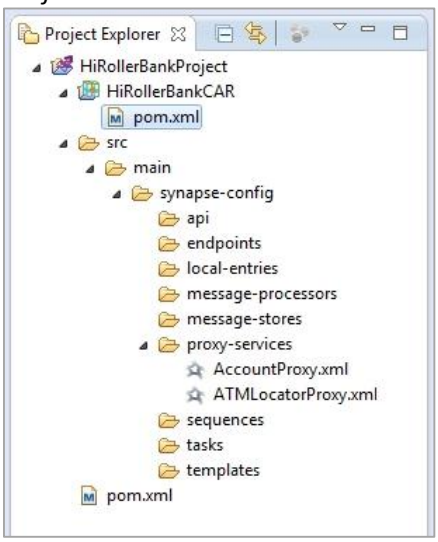
12. Inside the second case, add a *Send* mediator. This will take care of sending completed messages back to the client.



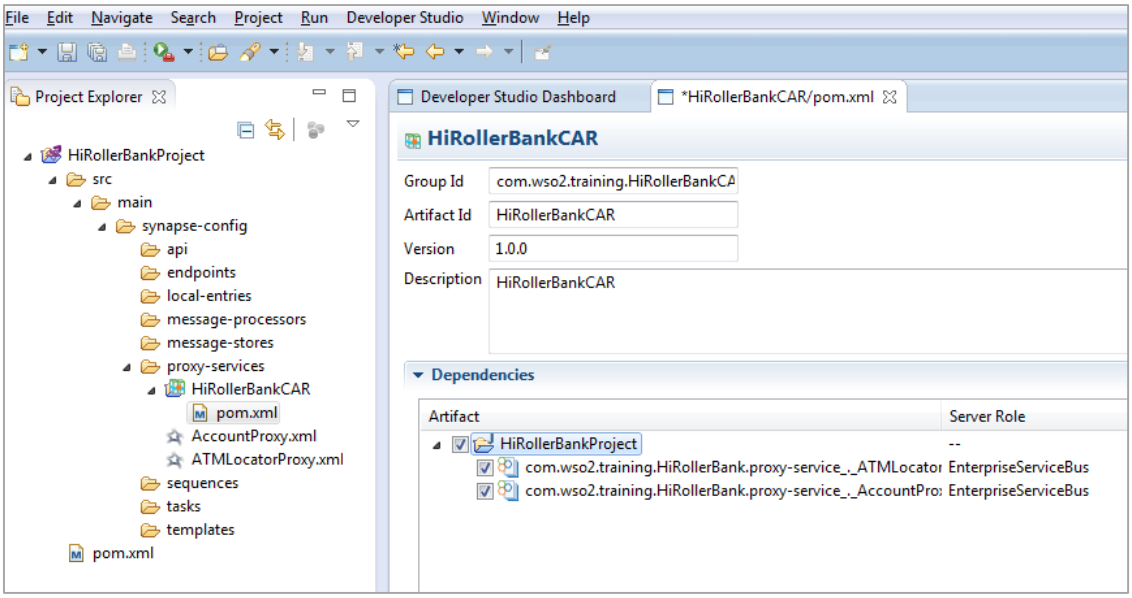
13. Close your *AccountProxy.xml*tab to save changes.

Adding your new Configuration to your Existing CAR file

14. Right-click on the POM file of the *HiRollerBankCAR* file and open with the *Composite Application Project POM Editor*



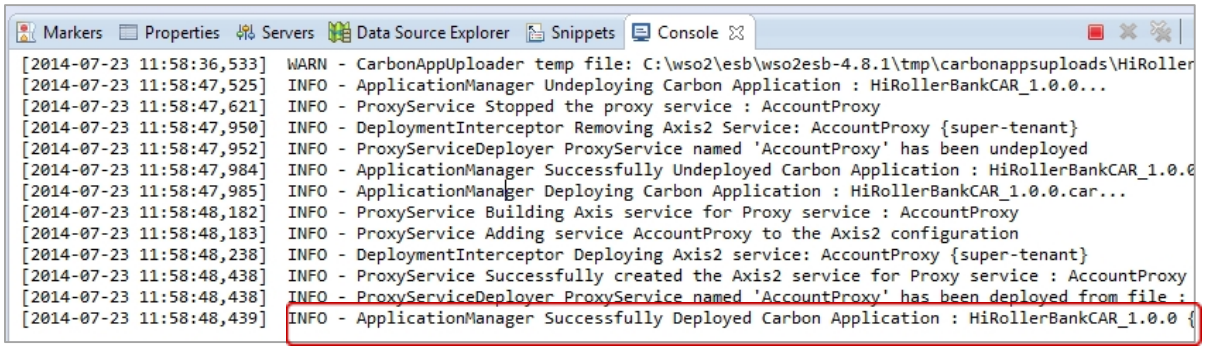
15. Select the newly added Proxy Service (*ATMLocator*) in the list of services bundled into the CAR file.



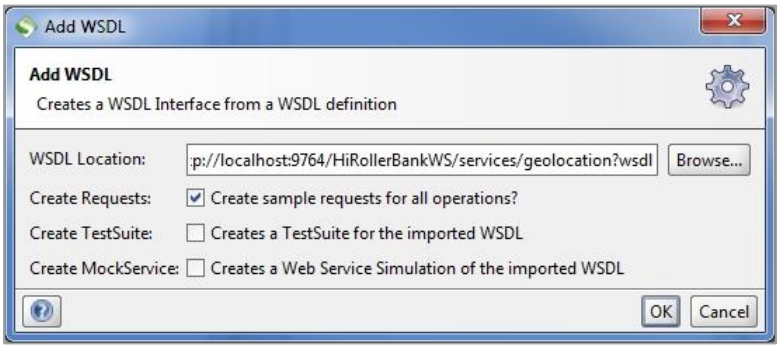
16. Close your *HiRollerCAR/pom.xml* tab to save your changes

Redeploy the Carbon Application and Test Changes

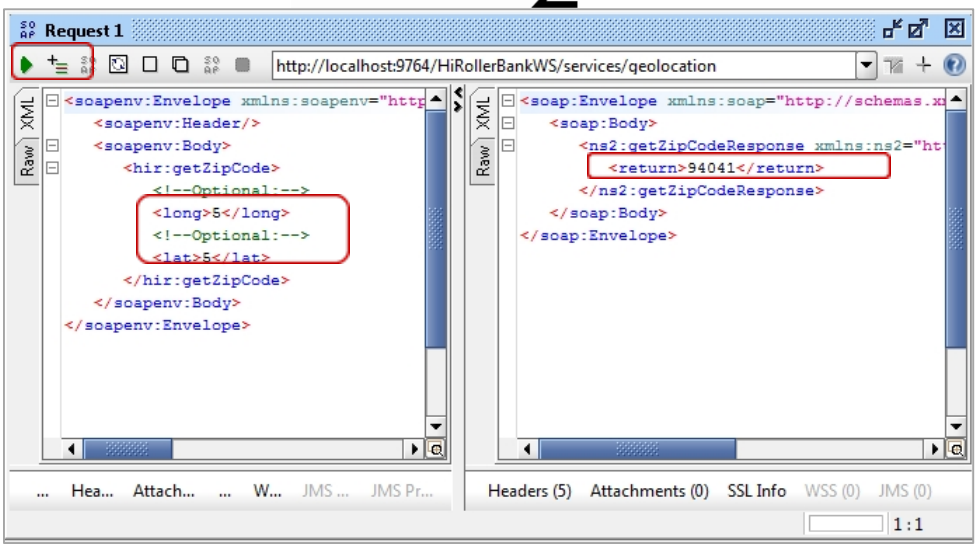
17. Switch to Java Perspective (Top-right corner) or use the Servers tab view added to the current Perspective
18. Click on the Servers tab in the bottom half of the screen
19. Right-click the *HiRollerBankCAR* application under the WSO2 ESB Training server and click *Redeploy*



20. In SoapUI, test the existing service and the newly created proxy service
    a. In SoapUI, right-click on HiRollerBankWS and select *Add WSDL*
    b. Set *WSDL Location* to **http://localhost:9764/HiRollerBankWS/services/geolocation?wsdl**(You can verify this by exploring the *HiRollerBankWS* app in AS)
    c. Click *OK*

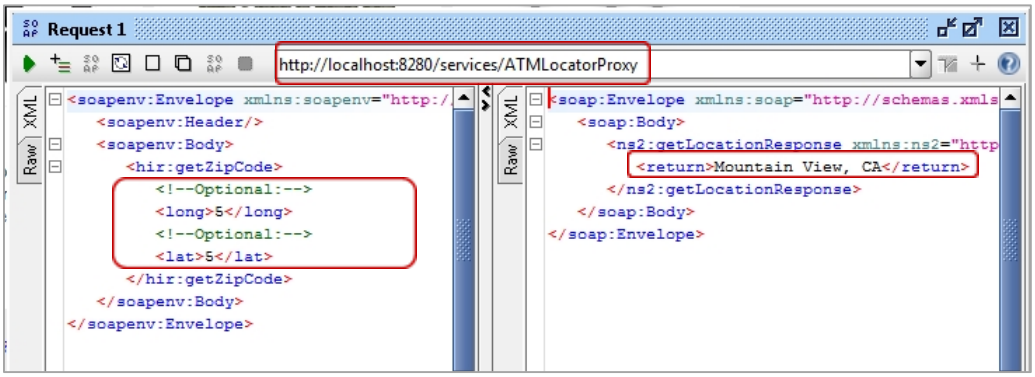


    d. On left-nav bar, under *getZipCode*, double-click on *Request1*
    e. In Request 1 window, set <long> to **5** and <lat> to **5**by replacing the '?'
    f. Click on Green play icon (top right) to send the *request* to the back-end
    g. In the response, look for a returned zipcode**(94041)**



This showed us how when we go directly to the *geoLocation* service, we get the zip code. Now let's try hitting our newly created proxy service which actually chains *geoLocation* and *ATMLocator*.

21. Now modify the SoapUI project to connect to our proxy service in WSO2 ESB
    a. Set the URL to**http://localhost:8280/services/ATMLocatorProxy**(verify this by exploring the *AccountProxy* service in ESB)
    b. Send the request again and look for the returned value



Our *ATMLocator* proxy service chained the various services to get us an ATM Location in Mt. View, CA.

Question : do you fully understand the message flow ?

LAB 04a: WSO2 ESB Functionality – Service Chaining (Call Mediator)

Training Objective: *Running the same scenario as Lab 04, this lab will replace the complex Switch/Send mediators that us the inSequence and outSequence, and instead use the Call/respond mediators which allow for a more logical way to build this solution, using only the inSequence.*

Business Scenario: HiRollerBank wants to allow its members to use their mobile phones to find the nearest ATM, according to their location. GPS coordinates received from the mobile phone (client) will be sent to a GeoLocation service to determine the correct zip code. In turn, this zip code will be used against an ATMLocator service to identify all ATMs located in the specified zip code.

WSO2 ESB orchestrates the requests and responses from the various services - to the end user (client), it all looks like one request with one response. Following the steps below, configure sequences with appropriate mediators to achieve the service chaining necessary to provide the right ATM locations to customers.

Backend Services	Location
GeoLocationService Receives GPS coordinates and returns US Postal zip code.	http://localhost:9764/HiRollerBankWS/services/geolocation
ATMLocatorService Receives a zip code and returns street addresses for all ATM in that zip code.	http://localhost:9764/HiRollerBankWS/services/atmlocator

High-level Steps:

- Create a new proxy service that will serve to coordinate requests and responses.
- Send the request to the first service
- When receiving a response, check whether we still need to make another request to another service, or send answer back to client (use a *Call* mediator)
- Add you new configuration to your existing CAR file
- Redeploy configuration and test

Detailed Instructions:

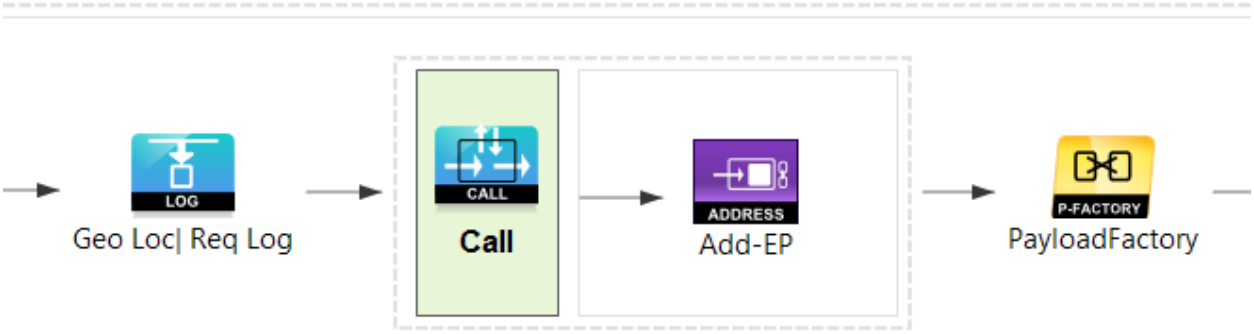
Create the Proxy Service

1. In DevStudio, in left-nav bar, right-click on *HiRollerBankProject*, select *New* ➔ *Proxy Service*
    a. Select *Create a New Proxy Service*
    b. Set *Proxy Service Name* to: **CallMediatorProxy**
    c. Set *Proxy Service Type* to: **Pass Through Proxy**
    d. Set *Target Endpoint* to: **Enter URL**
    e. Set *Endpoint* to: **http://localhost:9764/HiRollerBankWS/services/geolocation**
    f. Click *Finish* to build your new proxy service
    g. You can view your newly created *CallMediatorProxy.xml* in *Design* and *Source* views

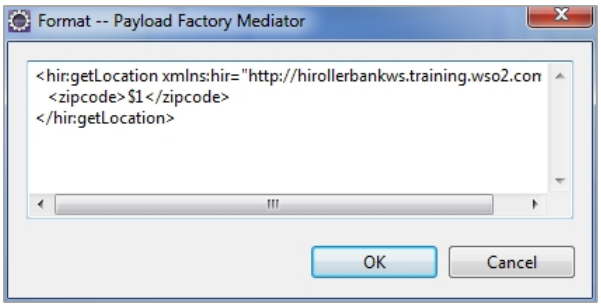
2. Drag a *Call* mediator before the *Send* mediator in the *InSequence*.
3. Add an *Address Endpoint* to the *Call* mediator

a. Set its *URI* to: **http://localhost:9764/HiRollerBankWS/services/geolocation**

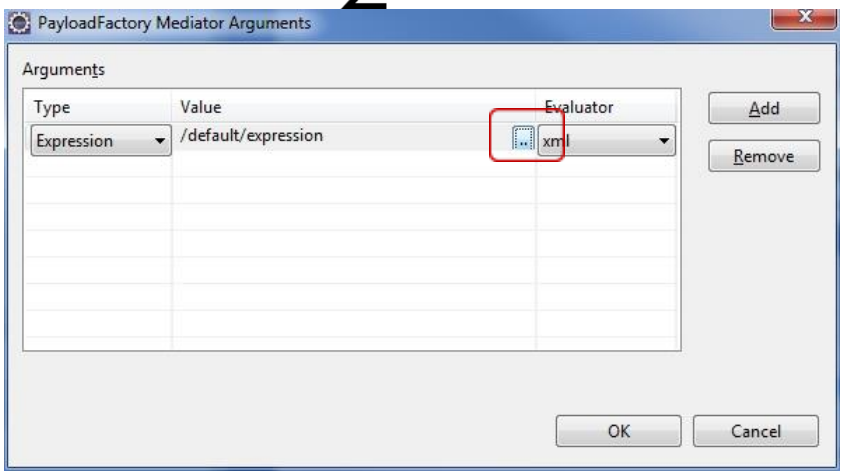
4. Delete the 2 existing *Send* mediators



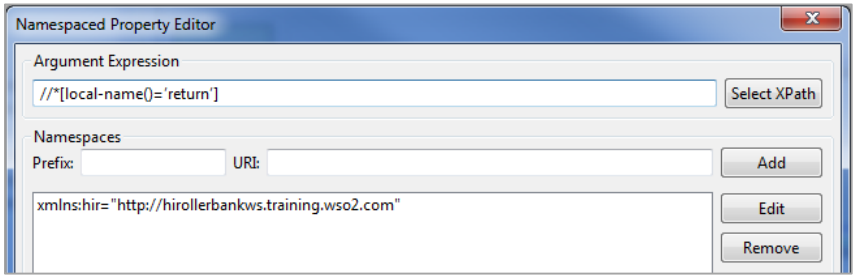
5. Add a *PayloadFactory* mediator and edit its properties
   a. Set the *Format* with the following payload
   <hir:getLocationxmlns:hir="http://hirollerbankws.training.wso2.com/">
   <zipcode>$1</zipcode>
   </hir:getLocation>



   b. Open the *Args* section and click on *Add* to add a new Argument
   c. Set the *Type* to *Expression*, and click on … to edit the Value

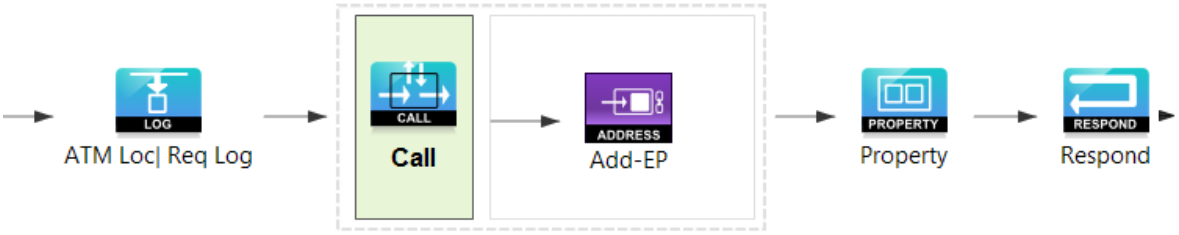


d. Set *Argument Expression* to **//*[local-name()='return']**
e. In *Namespaces*, set the *Prefix* to **hir**
f. Set the *URI* to**http://hirollerbankws.training.wso2.com/**
g. Click *Add* to add you new Namespace



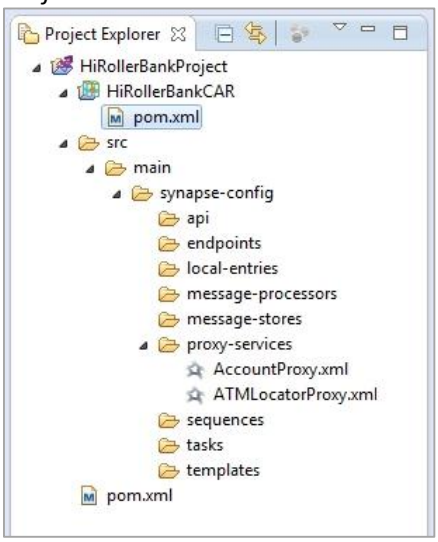
h. Click *OK* to set your Argument

6. Add a new *Call* mediator
7. Drag an *Address Endpoint*inside the *Call* mediator
    a. Set its URI to **http://localhost:9764/HiRollerBankWS/services/atmlocator**
8. Add a *Respond* mediator
9. Close your *CallMediatorProxy.xml* tab to save changes.



Adding your new Configuration to your Existing CAR file

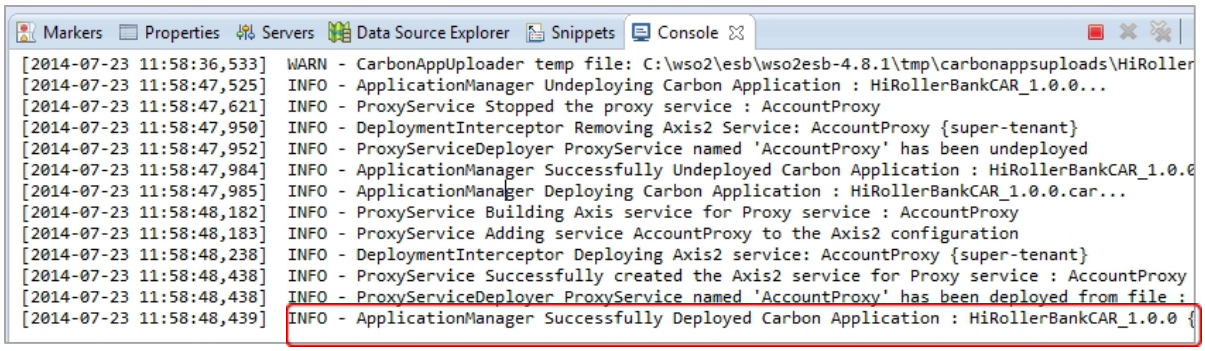
10. Right-click on the POM file of the *HiRollerBankCAR* file and open with the *Composite Application Project POM Editor*



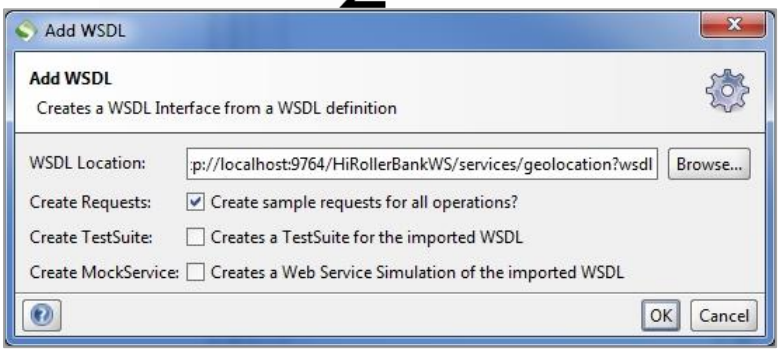
11. Select the newly added Proxy Service (*CallMediatorProxy*) in the list of services bundled into the CAR file.
12. Close your *HiRollerCAR/pom.xml* tab to save your changes

Redeploy the Carbon Application and Test Changes

13. Switch to Java Perspective (Top-right corner)
14. Click on the Servers tab in the bottom half of the screen
15. Right-click the *HiRollerBankCAR* application under the WSO2 ESB Training server and click *Redeploy*



16. In SoapUI, test the existing service and the newly created proxy service
    h. Right-click on HiRollerBankWS and select *Add WSDL*
    i. Set *WSDL Location* to **http://localhost:9764/HiRollerBankWS/services/geolocation?wsdl** (You can verify this by exploring the *HiRollerBankWS* app in AS)
    j. Click *OK*



k. On left-nav bar, under *getZipCode*, double-click on *Request1*
l. In Request 1 window, set <long> to **5** and <lat> to **5** by replacing the '?'
m. Click on Green play icon (top right) to send the *request* to the back-end
n. In the response, look for a returned zip code**(94041)**

LAB 04c: WSO2 ESB Functionality – Store and Forward (Optional)

Training Objective: Practice configuration of core functionality: Store and Forward

Business Scenario: To improve their services, HiRollerBank introduced a new service where customers can provide feedback on the ATM used (ease of use, cleanliness, location convenience, etc.) Using Store and Forward, HiRollerBank will ensure that messages are reliably delivered and stored for future reading/reporting.

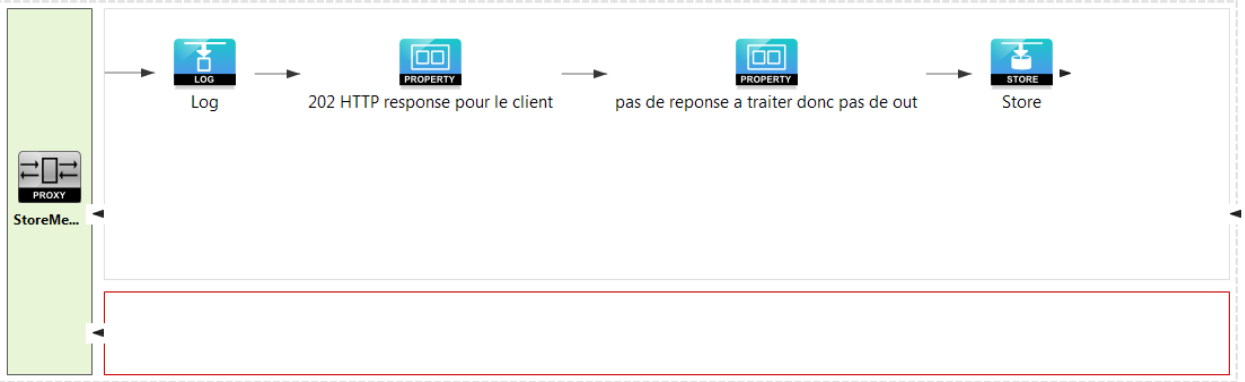
Following the steps below, create a sequence that uses Store and Forward functionality to store customer feedback and at the same time enable reliable messaging (in asynchronous communication, customer is not expecting a response to the feedback they provided).

So far, exercises have been synchronous (i.e. a response is expected). S&F provides reliable messaging for asynchronous scenarios, where no response is expected. In this case, when a customer provides feedback, there is no response expected.

Backend Services	Location
ReceiveFeedbackService Receives freeform text feedback and responds with a '200 OK' message.	http://localhost:9764/HiRollerBankWS/services/receivefeedback

High-level Steps:

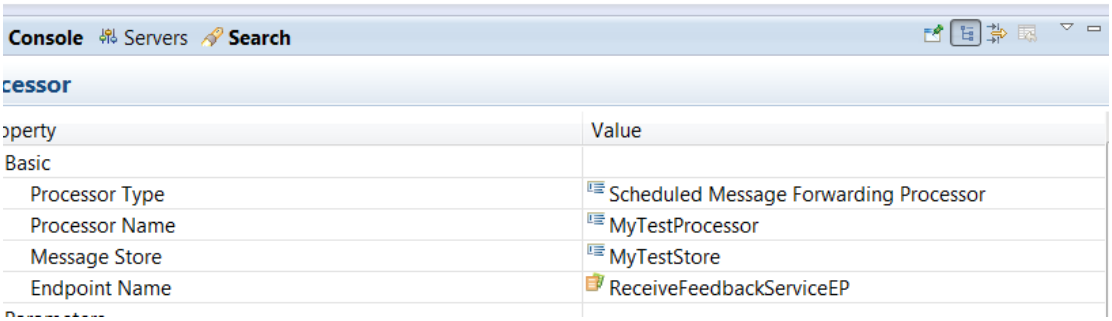
1. Create a Proxy that Stores incoming Messages into a MessageStore
2. Create a Message Processor that Forwards the Message to the Back-end Service



Detailed Instructions:

1. Create a MessageStore
2. Create a Proxy service that stores the message into the store
   - Tell the client it's ok
3. Create a Message Processor that Forwards the Message to the Back-end Service
   - Create a proxy to the feedBackService and use it to log the service usage

- Create an endpoint that targets the feedbackServiceProxy or the service directly (then you will have no logging available)
- Use the endpoint in the messageProcessor



Question : How would you use the messageProcessor to have some loadBalancing feature ?

A similar example can be done by following Sample 702 in the Documentation: https://docs.wso2.com/display/ESB481/Sample+702%3A+Introduction+to+Message+Forwarding+Processor. (Pay attention: in this sample the target is passed by value to the processor. This is not allowed when defining Processor from the Wizards.)

LAB 4c: WSO2 ESB Functionality – Store and Forward (Optional)

Training Objective: Practice configuration of core functionality: Store and Forward

Business Scenario: To improve their services, HiRollerBank introduced a new service where customers can provide feedback on the ATM used (ease of use, cleanliness, location convenience, etc.) Using Store and Forward, HiRollerBank will ensure that messages are reliably delivered and stored for future reading/reporting.

Following the steps below, create a sequence that uses Store and Forward functionality to store customer feedback and at the same time enable reliable messaging (in asynchronous communication, customer is not expecting a response to the feedback they provided).

So far, exercises have been synchronous (i.e. a response is expected). S&F provides reliable messaging for asynchronous scenarios, where no response is expected. In this case, when a customer provides feedback, there is no response expected.

Backend Services	Location

ReceiveFeedbackService Receives freeform text feedback and responds with a '200 OK' message.	http://localhost:9764/HiRollerBankWS/services/receivefeedback

High-level Steps:

1. Create a Proxy that Stores Messages into a MessageStore
2. Create a Message Processor that Forwards the Message to the Back-end Service

Instructions:

4. Create a MessageStore
5. Create a Proxy service that stores the message into the store
6. Create a Message Processor that Forwards the Message to the Back-end Service
    - Create a proxy to the feedBackService and use it to log the service usage
    - Create an endpoint that targets the feedbackServiceProxy or the service directly (then you will have no logging available)
    - Use the endpoint in the messageProcessor

LAB 05: Creating APIs (Proxy Service)

Training Objective: Practice configuration of a simple API with XML to JSON conversion

Business Scenario: PizzaShop allows clients to browse through its food menu and place orders. It has done this providing a backend RESTful service that accepts JSON message format (i.e. the preferred format for mobile clients).
To accommodate all client needs, PizzaShop would also like to accept XML messages. Follow the steps below to configure an API that will translate from XMLto JSON.

Backend Services	Location
MenuService A RESTful service that allows users to browse available pizzas and place orders.	http://localhost:9764/pizzashop-rs_1.0/services/menu

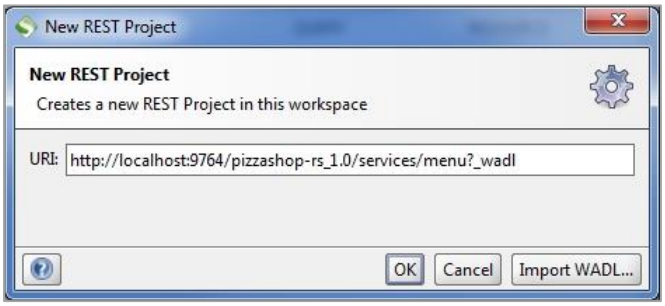
High-level Steps:

- Create a Testing Environment for REST services using SOAP UI
- Build a XMLto JSON Transformation using WSO2 ESB (POSTing of a new order)
- Deploy your new configuration to your ESB (Build new CAR file)Test your proxy service

Detailed Instructions:

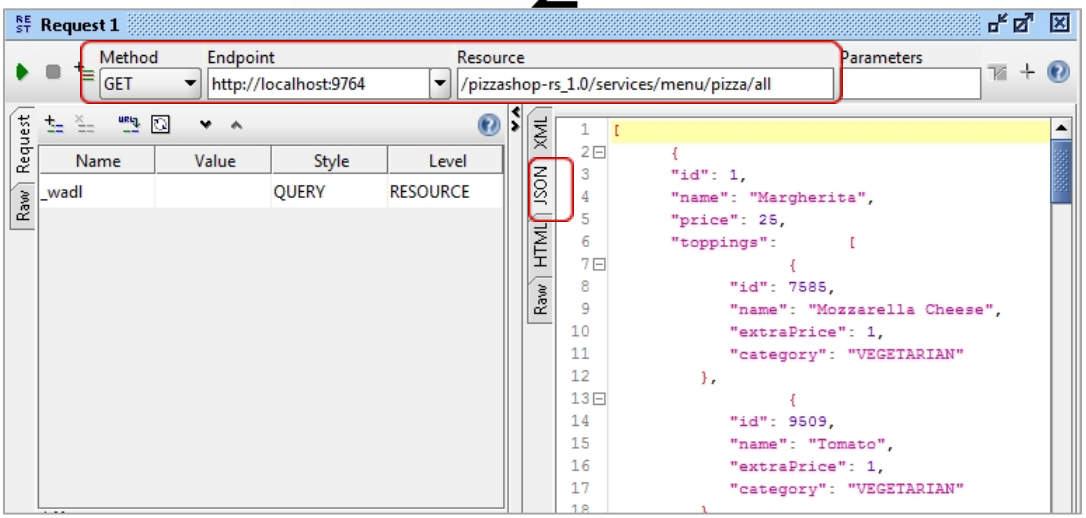
Create a new REST project for REST services

1. In SoapUI, right-click on *Project* and select *New REST Project*
2. Set the *URI* to **http://localhost:9764/pizzashop-rs_1.0/services/menu?_wadl**



Notice how this time, you are using a wadl (which describes RESTful services) versus a wsdl (which describe all other services).

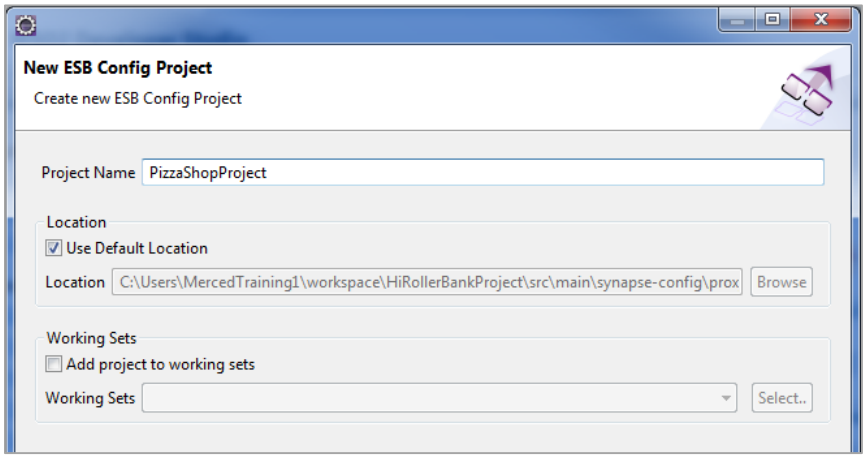
3. In the *Request 1* window, set the following:
    a. Method to **GET**
    b. Endpoint to **http://localhost:9764**
    c. Resource to **/pizzashop-rs_1.0/services/menu/pizza/all**
4. Execute request and switch the *Response* view to *JSON* to see the returned payload



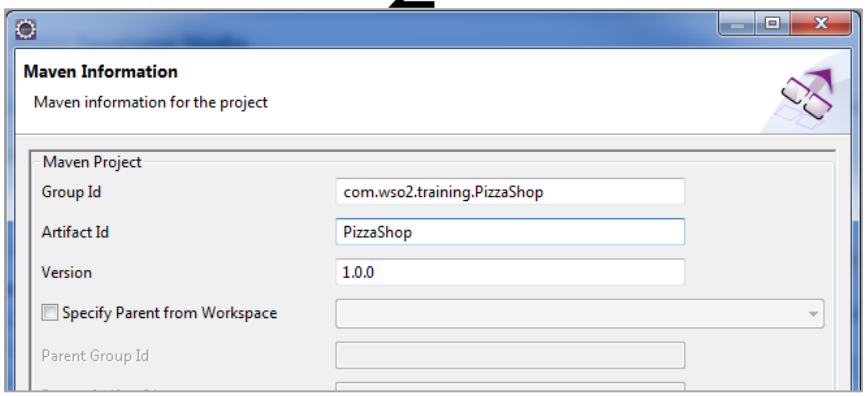
We have just tested the RESTful backend service with SoapUI. Let's now create a proxy service that allows us to do the transformation.

Create an ESB Configuration Project

1. From top menu, choose *File* ➔ *New* ➔ *Project…* and under *WSO2* ➔ *Message Mediation* ➔ *Project Types*, select *ESB Config Project*
    a. Select *New ESB Config Project*
    b. Name your project **PizzaShopProject**
    c. Keep the default location and click *Next*



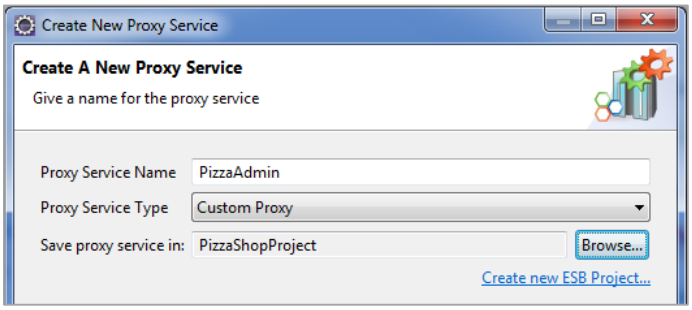
2. Since this is a Maven project, you need to provide all the pertinent Maven info:
    a. Set *Group ID* to: **com.wso2.training.PizzaShop**
    b. Set *Artifact ID* to: **PizzaShop**and*Version*to **1.0.0**



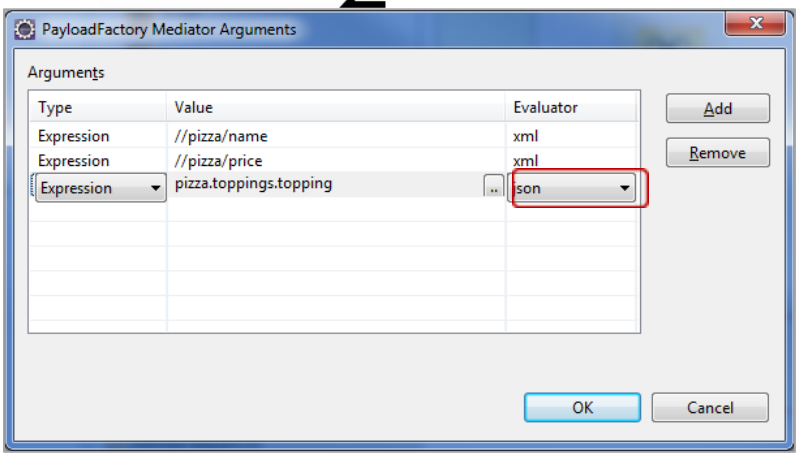
3. Click *Finish* to create a new project which you can view in *Project Explorer*. You can expand this to view all directories, including your */src/main/synapse-config*. This is where all your ESB configurations will be stored.

Create a Proxy Service

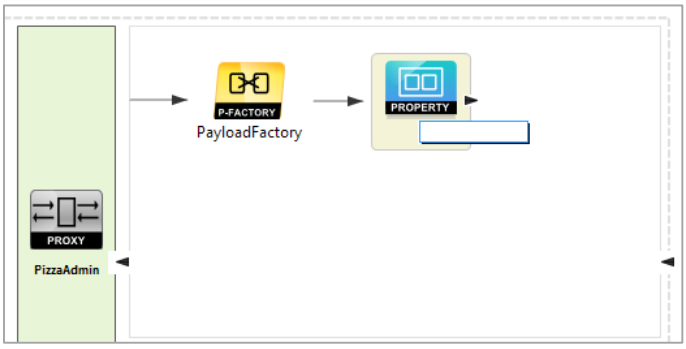
1. In left-nav, right-click on *PizzaShopProject*, select *New*, and click on *Proxy Service*
    a. Select *Create a New Proxy Service*
    b. Set *Proxy Service Name* to: **PizzaAdmin**
    c. Set *Proxy Service Type* to: **Custom Proxy**
    d. Click *Finish* to create your new proxy service



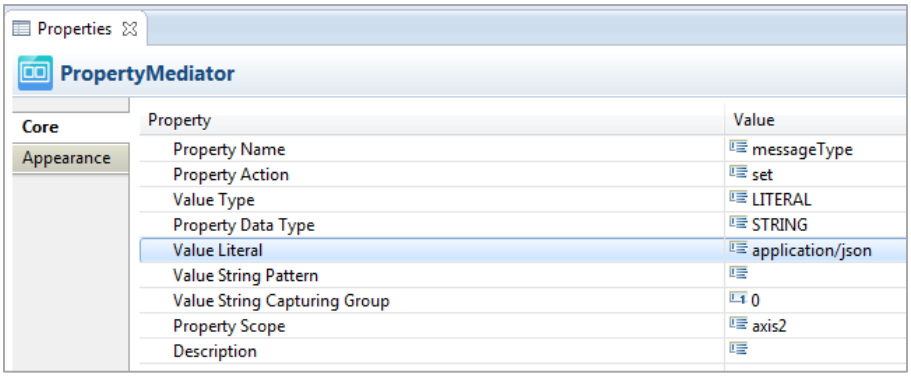
2. Use a *PayloadFactory* to transform from XML to JSON.
    a. Drag a *PayloadFactory* mediator to the *inSequence* of your new proxy service
    b. In *Properties*, set *Media Type* to **json**
    c. Set *Payload Format* to Inline
    d. Next, set *Payload* **format** to **{"pizza": {"name": "$1", "price": $2, "topping": $3}}**
    e. Now add two *Arguments* of type *Expression* with values **//pizza/name** and **//pizza/price**
    f. Add a third *Argument* of type *Expression* with value **pizza.toppings.topping**
    g. For the third *Argument* set the *Evaluator* to **json**



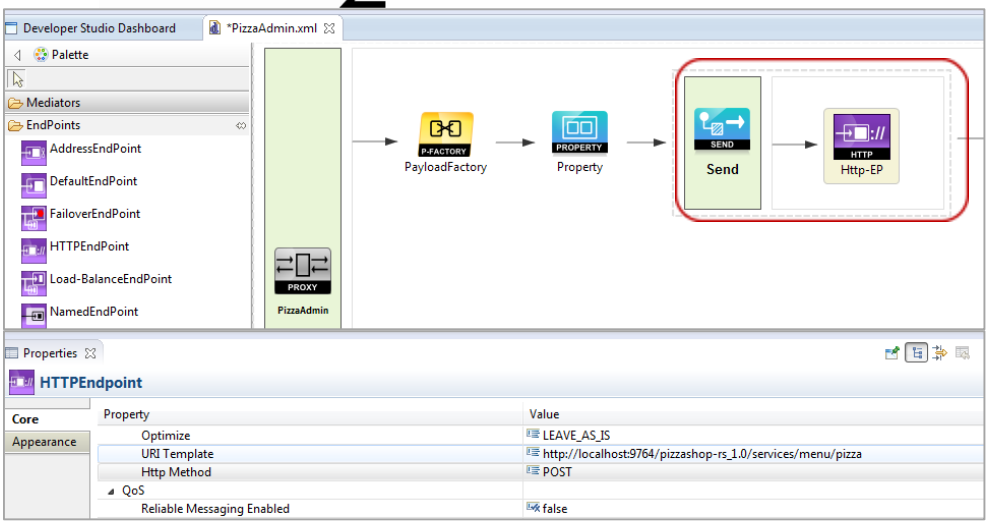
3. Add *Property*mediator after the *PayloadFactory*mediator



   a. In *Properties,* setProperty N*ame* to**messageType**
   b. Set *Value Literal* to**application/json**
   c. Set *PropertyScope* to**axis2**

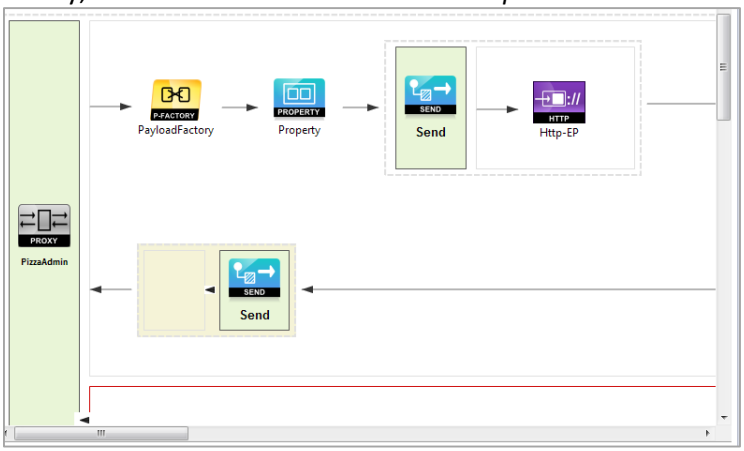


4. Now add a *Send* Mediator after the *Property* mediator
5. Next, add an*HTTP Endpoint* to the *Send* mediator
   a. Set *URI Template* to**http://localhost:9764/pizzashop-rs_1.0/services/menu/pizza**
   b. Set *HTTP Method* as **POST**



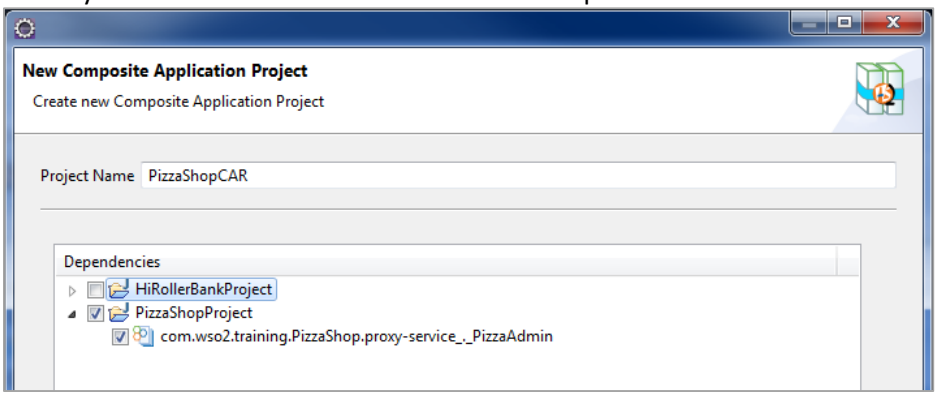
Notice how this time you used an HTTP Endpoint since you are using a RESTful backend service.

6. Finally, Add a *Send* mediator to the *outSequence*

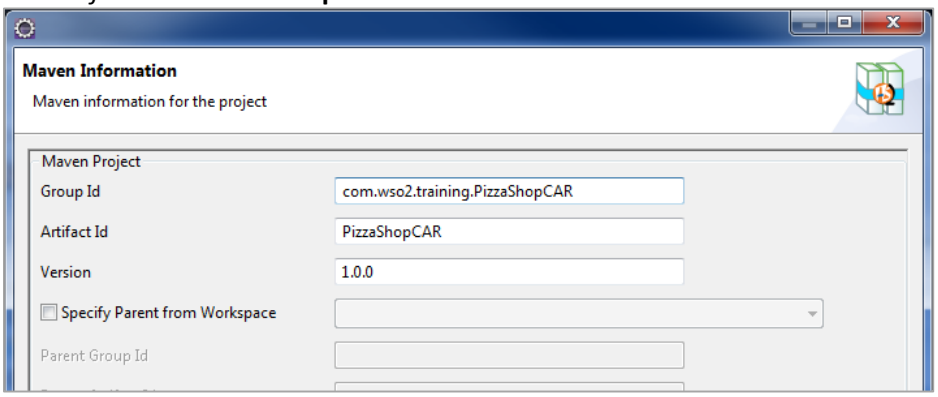


Deploy your Configuration to your ESB and Test

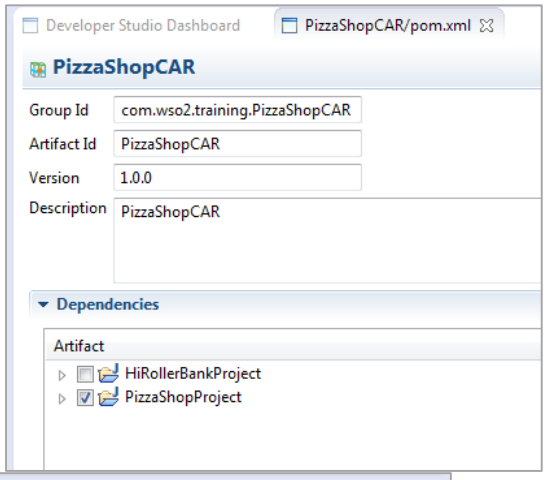
1. Create your CAR file (Collection of deployable artifacts)
    a. Close your *PizzaAdmin.xml* tab to save your changes
    b. In the *Dev Studio Dashboard*, under *Distribution*, click on *Composite App Project*
    c. Set *Project Name* to: **PizzaShopCAR**
    d. Select your *PizzaAdmin*service from the list of dependencies and click *Next*



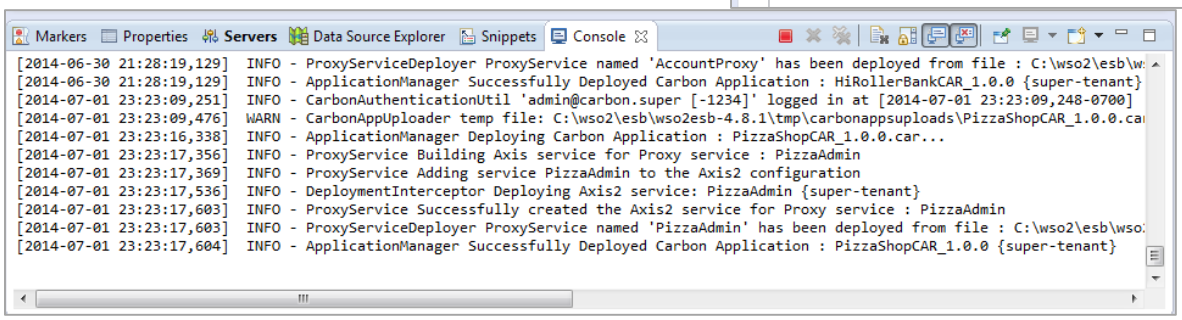
e. Set *Group ID* to: **com.wso2.training.PizzaShopCAR**
f. Set *Artifact ID* to: **PizzaShopCAR** and *Version* to **1.0.0**



g. Click *Finish* to create your *PizzaShopCAR/pom.xml* file

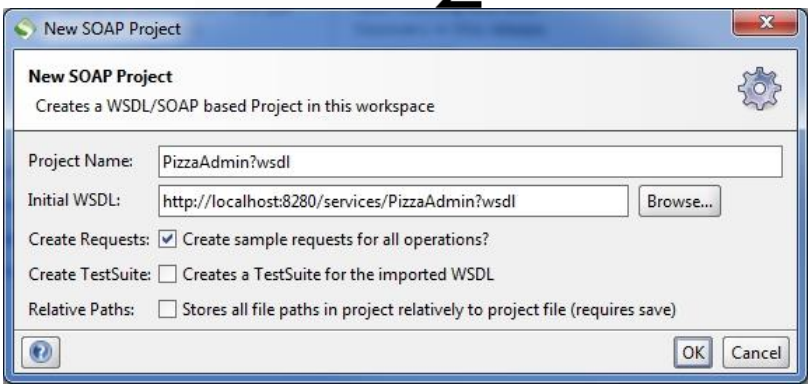


2. Deploy your CAR file to ESB
   a. Switch to Java Perspective (Top-right corner) or use locally added Servers view
   b. Click on the Servers tab in the bottom half of the screen
   c. Right-click on the *WSO2 ESB Training* Server and select *Add and Remove…*
   d. Highlight *PizzaShopCAR* and click *Add*
   e. *Click Finish to deploy your new CAR file*



Test Your Proxy Service

1. In SoapUI, right-click on *Project* and select *New SOAP Project*
2. Set the *Initial WSDL* to **http://localhost:8280/services/PizzaAdmin?wsdl**

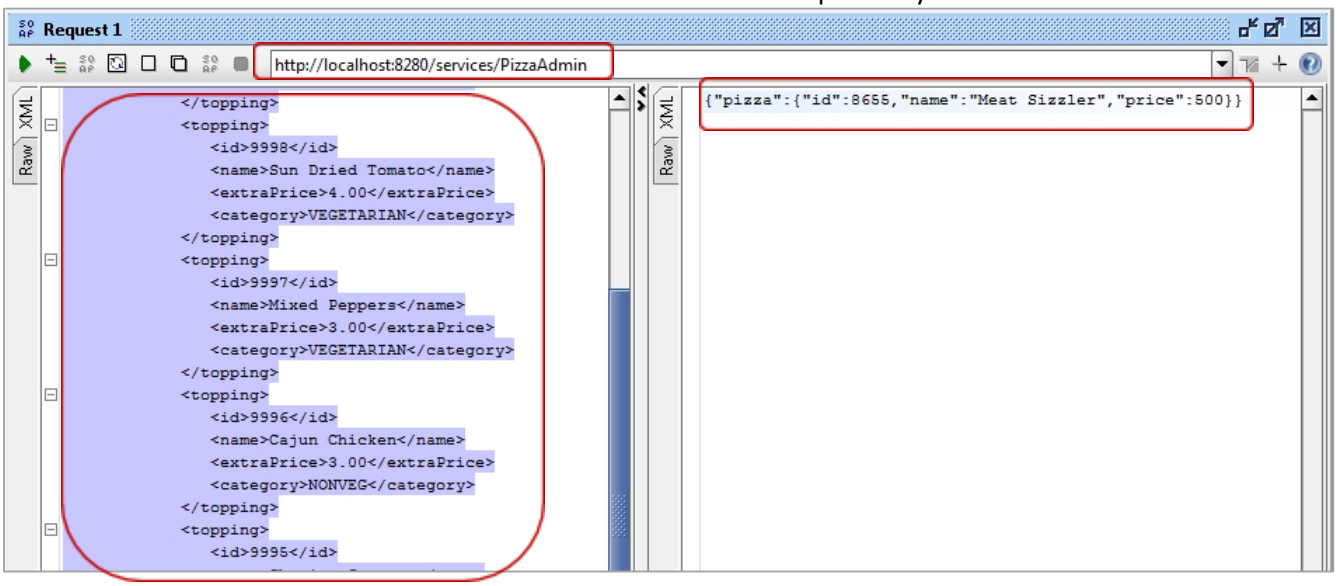


a. Expand the *PizzaAdminSoap_1.0.0* and open *Request 1* under *mediate*
b. Set the endpoint URL to**http://localhost:8280/services/PizzaAdmin**
c. For the message payload, paste the following (*test_payload.txt*):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header/>
   <soapenv:Body>
      <pizza>
         <name>Meat Sizzler</name>
         <price>500.0</price>
         <toppings>
            <topping>
               <id>9999</id>
               <name>Steak</name>
               <extraPrice>4.00</extraPrice>
               <category>NONVEG</category>
            </topping>
            <topping>
               <id>9998</id>
               <name>Sun Dried Tomato</name>
               <extraPrice>4.00</extraPrice>
               <category>VEGETARIAN</category>
            </topping>
            <topping>
               <id>9997</id>
               <name>Mixed Peppers</name>
               <extraPrice>3.00</extraPrice>
               <category>VEGETARIAN</category>
            </topping>
            <topping>
               <id>9996</id>
               <name>Cajun Chicken</name>
               <extraPrice>3.00</extraPrice>
               <category>NONVEG</category>
            </topping>
            <topping>
               <id>9995</id>
               <name>Chorizo Sausage</name>
               <extraPrice>4.00</extraPrice>
               <category>NONVEG</category>
            </topping>
         </toppings>
      </pizza>
   </soapenv:Body>
</soapenv:Envelope>
```

3. Observe the JSON response
    d. The response shows you that PizzaShop took your order for a Meat Sizzler pizza with various toppings and assigned an ID to your order.

e. In summary, your PizzaAdmin proxy service received a XML message, which it in turn converted to the desired format and sent a POST request to your RESTful service



Question: How could you switch the response to XML?
Hint:


```
<payloadFactory media-type="xml">
	<format>
		<pizza xmlns="">
			$1
		</pizza>
	</format>
	<args><arg evaluator="json"expression="$.pizza"/></args>
</payloadFactory>
<property name="messageType"value="text/xml"scope="axis2"type="STRING"description=""/>
```

LAB 06: Creating APIs (HTTP Service)

Training Objective: Practice configuration of invoking a backend service (http endpoint)

Business Scenario: At the application layer, PizzaShop has built new business functionality which utilizes RESTful API. Therefore, PizzaShop needs the ability to communicate with its new backend services using REST.
Following the instructions below, configure a RESTful API.

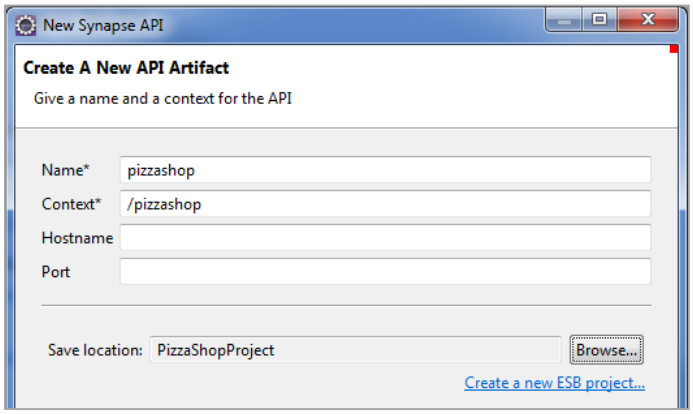
High-level Steps:

- Create a REST API artifact that allows clients to GET a submitted order
- Add you new configuration to your existing CAR file
- Redeploy configuration and Test (create an order and test your REST API)

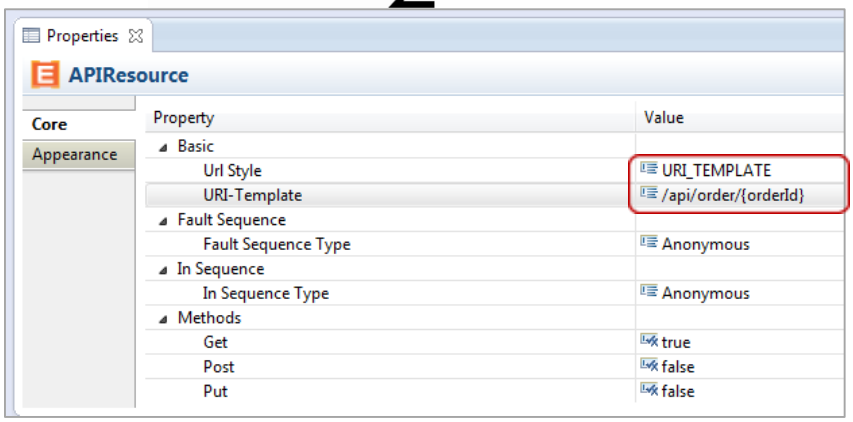
Detailed Instructions:

Create a REST API

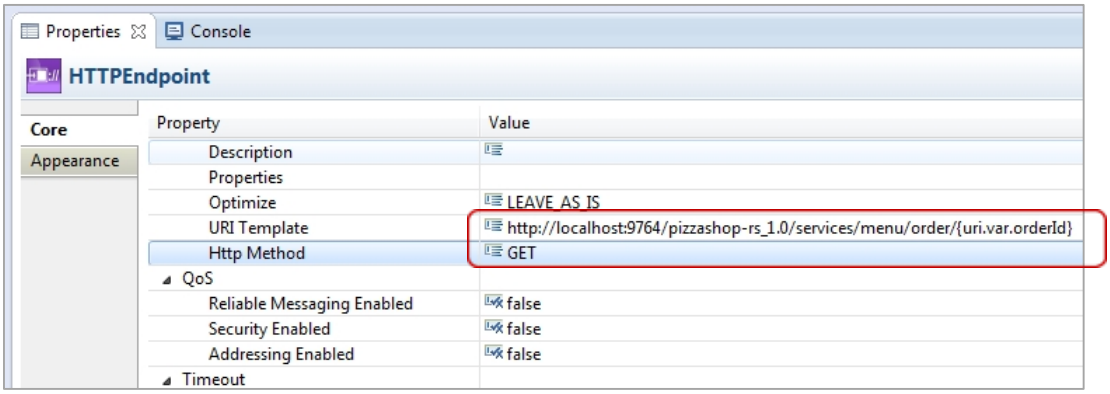
1. In the left-nav, right-click on your *PizzaShopProject* and select *New ➔ Other ➔ WSO2 ➔ Message Mediation ➔ Message Mediation Features ➔ REST API* and click*Next*
    a. Select *Create a New API Artifact*
    b. For your API artifact, set *Name* to: **pizzashop**
    c. Set *Context* to: **/pizzashop**
    d. Click on *Finish* to build your API artifact



    e. Right-click on the *Resource* icon and select *Show Properties View*
    f. In Properties, use dropdown to set *Url Style* to**URI_TEMPLATE**
    g. Set *URI-Template*to**/api/order/{orderId}**

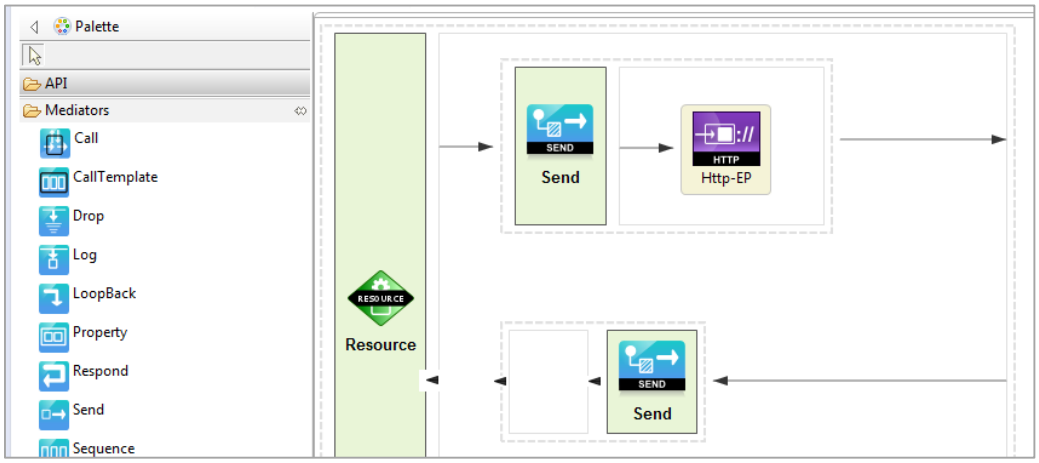


7. Next, add a *Send* Mediator with an *HTTP Endpoint* to the *inSequence*
    a. Set *URI Template*to**http://localhost:9764/pizzashop-rs_1.0/services/menu/order/{uri.var.orderId}**
    b. Set *HTTP Method* to**GET**



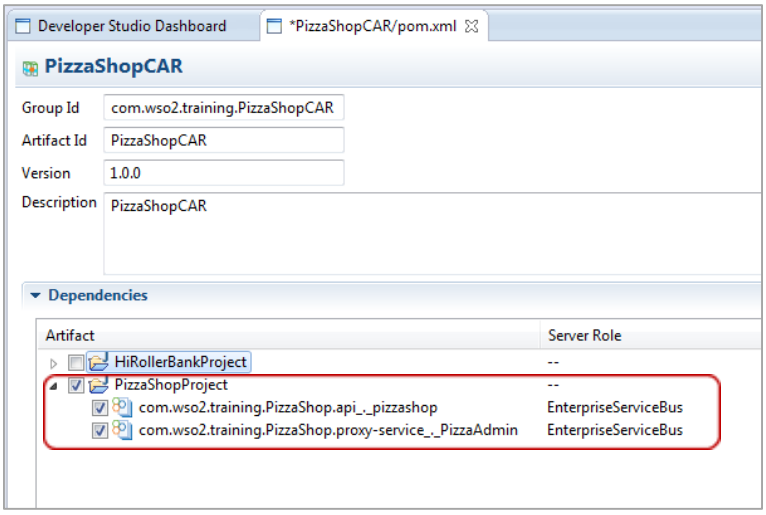
This will allow you to retrieve an order (GET), by providing an order id.

8. Lastly, add a *Send* mediator to the *outSequence*

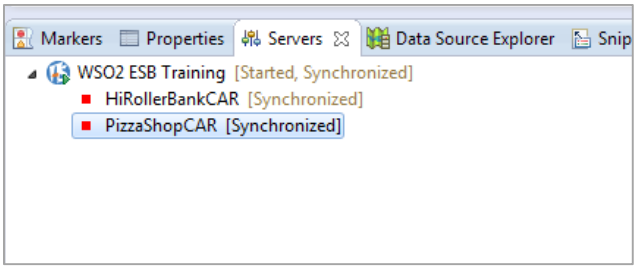


Deploy your Configuration to your ESB

1. Close the *pizzashop.xml* tab to save your changes
2. Right-click on the *pom.xml* file of the *PizzaShopCAR* file and open with the *Composite Application Project POM Editor*
3. In *Dependencies*, select the newly added API artifact to bundle it into the existing CAR file.



4. Close the *PizzaShopCAR/pom.xml* tab to save your changes
5. Change to *Java* perspective (top-right corner menu)
6. In the *Servers* tab, right-click on *PizzaShopCAR* and select *Redeploy*

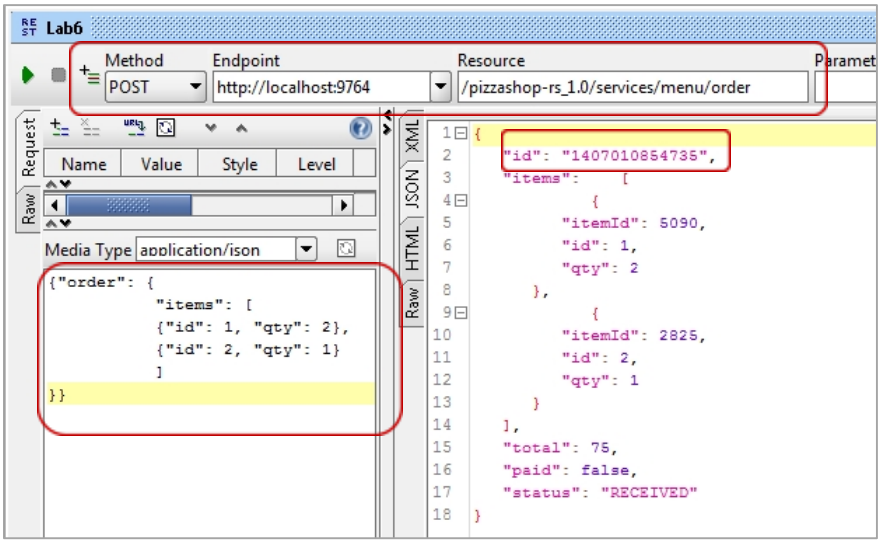


Test Changes

1. Back in soapUI, create a pizza order by doing the following:
    a. Under *REST Project 1*, under *Menu*, right-click on *Menu* and select *New Request*
    b. You can call your new request *Lab06*
    c. Set *Method* to **POST**
    d. Set *Endpoint* to **http://localhost:9764**
    e. Set *Resource* to **/pizzashop-rs_1.0/services/menu/order**
    f. Send the following message (*test_order_payload.txt):*

```
{"order": {
          "items": [
          {"id": 1, "qty": 2},
          {"id": 2, "qty": 1}
          ]
}}
```

2. Switch the *Response* view to *JSON* to see the returned payload



Make a note of the id in the *Response*as we will refer to this as *<your_order_id>* in the next step.

3. In SoapUI, request a status for the order just submitted
    a. Set*Method* to **GET**
    b. Set *Endpoint*to **http://localhost:8280**
    c. Set *Resource* to**/pizzashop/api/order/<your_order_id>**

    Note:
    In some versions of SOAP UI, you are not able to edit the request path. If this is your case, skip these last 3 steps and simply, in a browser, open the URL:**http://localhost:8280/pizzashop/api/order/<your_order_id>**

LAB 07: Creating Simple Task

Training Objective: Introduce the concept of tasks and how simple trigger works

Whenever ESB gets started and initialized, tasks will run periodically in n second intervals. You could limit the number of times that you want to run any task by adding a count attribute with an integer as the value, if the count is not present, this task will run forever.

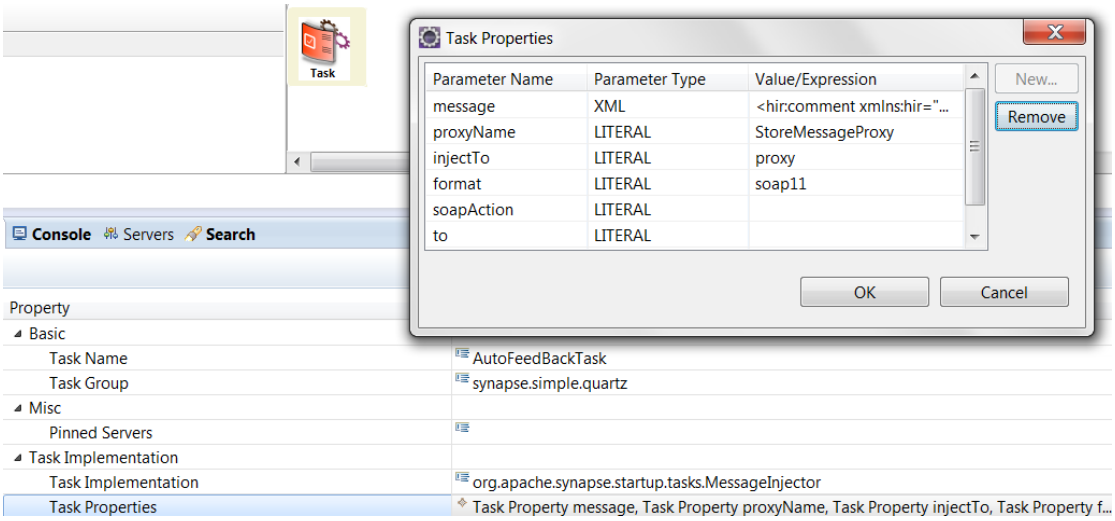
Scenario: Create a simple task that inserts an XML message into the log files every 10 seconds.

Detailed Instructions:

1. In Developer Studio, create new Scheduled Task
   - Set Task Name: **My_ESB_Task** and set Interval to **10 seconds**
2. In Design View, right-click to your Task to configure its properties. Set:
   - SequenceName: **main**
   - Message: **XML**
   - Expression: **<name>My name is Rio</name>**
3. Save your Task
4. Add to CAR file and redeploy, and restart your ESB
5. Review your log files or command line console to see your message every 10 secs

LAB 07a: Creating more Task (Optional)

One can write his own task class implementing the org.apache.synapse.startup.Task interface and implementing the execute method to do the task. For this particular sample, we have used the default implementation MessageInjector to inject a message specified in to a Message Store of the ESB environment.



1. Create a task that will call the former StoreMessageProxy (from previous lab)
   a. Use simple MessageInjector
   b. Use Task Properties to target the proxy using the following xml message:

```
<hir:commentxmlns:hir="http://hirollerbankws.training.wso2.com/">
```

```
        <textxmlns="">This is my message</text>
</hir:comment>
```

LAB 08a: Using Connectors

Training Objective: Practice managing connectors in your ESB instance.
https://docs.wso2.com/display/ESB481/Managing+Connectors+in+Your+ESB+Instance

Business Scenario:
Connect your ESB to be able to receive issues logged on the JIRA tracking tool by providing a JIRA ID. Use a REST API.

High-level Steps:

- Download, install and enable JIRA connector
- Connect to JIRA
- Read content from JIRA (JIRA_ID: APIMANAGER-1808)
- Return response to client

Detailed Instructions:
Download, Install and Enable the JIRA Connector

1. Download **esb-connectors-master.zip** from https://github.com/wso2/esb-connectors/tree/master/distribution
2. Unzip content to ESB_HOME/esb_connectors
3. In ESB UI Management Console:
   To add a connector:
   1. On the Main tab in the ESB Management Console, under Connectors click **Add**.
   2. Click **Browse**, specify the ZIP file, and click **Open**.
   3. Click **Upload**.

   The connector will now appear in the Connectors list and is ready to be enabled in your ESB instance.

   To enable a connector:
   1. On the Main tab in the ESB Management Console, under Connectors click **List** to view your uploaded connectors.

   Click **Enable** next to a connector you want to enable, and then confirm that you want to change its status. Repeat this step for each connector you want to enable.

Configure Connection to JIRA

4. In Developer Studio, right-click on **Project ➐ Import Connector**
   - Upload the JIRA connector in the ESB_HOME/esb-connectors/distribution folder
   - Notice the *JIRA Connector* Folder in your Design View
5. In Developer Studio, Create *New REST API*
   a. Set *Name*: **JIRA_Connector_REST**
   b. Set *Content*: **/jiraREST**
6. Once created, in *Design View*, right-click on **Resource** icon and edit properties:
   a. Set *URL Style:***URI TEMPLATE**
   b. Set *Method*: **GET**

   c. Set *URI_Template*: **/{jiraid}**
7. Drag **Init** mediator (Under **Jira Connector** folder) into the *inSequence*
   a. Set *username*: **esb.fundamentals@gmail.com**
   b. Set *password*: **YAXjtEsusF**
   c. Set *URI*: **https://wso2.org/jira**

   Now, you need a way to capture the JIRA Id for the issue you want to see (i.e. APIMANAGER-1808), so you need to add a property mediator to capture that variable.

8. Add **Property** mediator before **Init** mediator, and set:
   a. Set *Property Name*: **uri_value**
   b. Set *Value Type*: **Expression**
   c. Set *Value Expression*: **get-property('uri.var.jiraid')**
9. Insert a **getIssue** mediator after **Init,** and set:
   a. Set *issueIdorKey*: **{$ctx:uri_value}** (Using the variable defined in **Property** mediator)
10. Add a **Respond** mediator after **getIssue** mediator
    Notice that because we are working in the InSequence, we must use a Respond mediator to return answer to client. A Send mediator can only be used in the OutSequence.

11. To deploy, add dependencies to CAR file
12. Redeploy
13. To test in Soap UI, create REST Client
    a. Method is GET
    b. Resource is http://localhot:8280/jiraREST/APIMANAGER-1808

LAB 08b: Using Connectors - Write mode (Optional)

Training Objective: Keep practicing managing connectors in your ESB instance.

Business Scenario:
Connect your ESB to be able send selected feedback comments to an external GoogleSpreadsheet that can be shared with other readers

High-level Steps:

- Download, install and enable GoogleSpreadSheet connector
- Create a Proxy Service
    1. Read some auto generated fortunes from a rest API
    2. Retrieve the quote from the response
    3. Clone the response
        1. Call the StoreMessageProxy with the quote from the response
        2. Update the spreadsheet with the quote

All operation details are available at:
https://docs.wso2.com/display/ESB481/Google+Spreadsheet+Connector

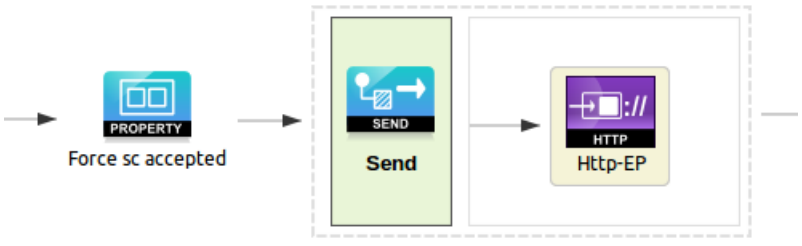
Detailed Instructions:

Download, Install and Enable the GoogleSpreadSheet Connector

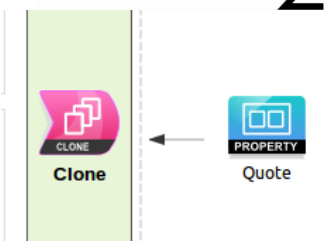
1. Instructions same as the JIRA connector
2. https://github.com/wso2/esb-connectors/tree/master/distribution/google-spreadsheet

Create a proxy service

1. Read fortunes
    a. Add a *Property* mediator to send a 202 response to the client (FORCE_SC_ACCEPTED)
    b. Add a *Send* mediator to call the quote API with an *HTTP Endpoint*
    Set *URL* to: **http://www.iheartquotes.com/api/v1/random?format=json**
    Set *method* to: **get**



2. Retrieve the Quote field from the json response
    Use a *Property* mediator to store the quote string using expression "**json-eval(quote)**"



3. Add *Clone* mediator to split the treatment
    a. Send the quote to the StoreMessageProxy
        i. Set the messageType to text/xml
        ii. Set a header to set no soap action
            name = "Action"
            value = """"
        iii. Add a *payloadFactory* mediator to format a message with the quote
        iv. Add a *Call* mediator to send the feedback

    b. Update a Google spreadsheet
        i. Filter the quote
            1. Don't process quotes that are too short (less than 2 lines)
            2. You can use the Xpath expression

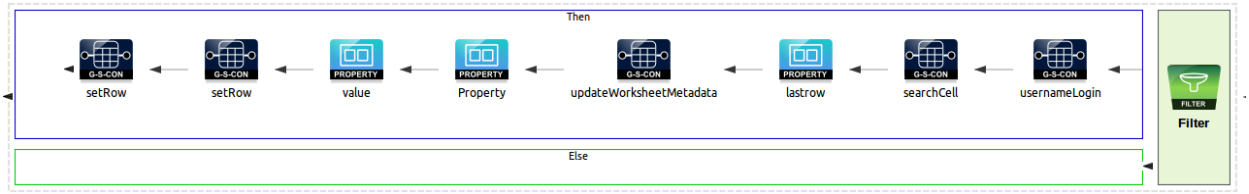
**"(string-length(get-property('quote')) - string-length(translate(string(get-property('quote')), '
', ''))) > 2"**

        ii. Use the connectors
            1. Create a local entry to store connector's credentials
               Call it googleCredential
               Use the following xml body:

```
<googlespreadsheet.init>
    <oauthConsumerKey/>
    <oauthConsumerSecret/>
    <oauthAccessToken/>
    <oauthAccessTokenSecret/>
 </googlespreadsheet.init>
```

            2. Add *usernameLogin* mediator with config key "googleCredential"
               User login = **wso2demoesb**
               User password = **Yademowso2**
            3. Add a *searchCell* mediator to find the last line of the spreadsheet (marked by the text "lastentry")
               spreadsheetName **= longFeddbacks**
               worksheetName **= filtered**
            4. Store the last row number in a property using another Xpath expression:
               translate(.,translate(., '0123456789', ''), '')
            5. Add an *updateWorksheetMetadata* mediator to add a new row at the end of the spreadsheet
               worksheetRows = { fn:string(get-property('lastRowNumber')+1)}

6. Store the csv line containing the date and the quote in a “rowValue” property
   date expression = get-property('SYSTEM_DATE', 'MM/dd/yyyy')
   rowValue expression = concat(get-property('date'),',', translate(get-property('quote'),',',';'))
7. Add a *setRow* mediator to create the new “last row”
   rowId = { fn:string(get-property('lastRowNumber')+1)}
   rowData = lastentry,
8. Add a *setRow* mediator to create the new row with the ”rowValue”
   rowId = {get-property('lastRowNumber')}
   rowData = {get-property('row_value')}
9. Cross your fingers and hope you typed the xPath correctly!



The current Google spreadsheet Connector API we are using do not provide the last row info. The lastentry trick is a workaround.

LAB 08c: multi Sequence - (Optional^2)

Business Scenario:
The former lab has produced an impressive set of mediators to be displayed in the Graphical editor. We have 2 issues there :
One cannot have a global view of the Sequence of Mediators
How can we deal with special error treatment scope ?

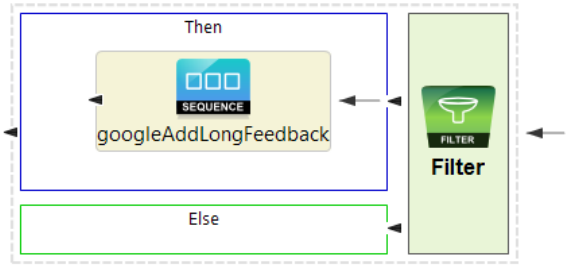
High-level Steps:

- Create a sequence artifact
- Copy all the GoogleConnector part to this sequence
- Replace this sequence of mediators by the named sequence

Detailed Instructions:

1. From Dashboard or right clik in your ESB configuration Project and choose add Sequence
    a. Name it googleAddLongFeedBack
    b. Copy the content of the <then> filter branch
    c. Paste it as the body of the new created sequence

2. Go back to the former **proxy service**
    a. Delete all the content of the <then> filter branch
    b. Replace it with the Defined sequence you can find in the Palette
    c. Enjoy the graphical view and navigation



LAB 09: Configuring Quality of Service (QoS) - Throttling

Throttling is handled by a Throttling mediator. You define a Throttling policy (XML), built easily in Eclipse. Here's an example of the resulting XML:

```
<policy>
 <!-- define throttle policy -->
 <wsp:Policyxmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
        xmlns:throttle="http://www.wso2.org/products/wso2commons/throttle">
   <throttle:ThrottleAssertion>
     <throttle:MaximumConcurrentAccess>10</throttle:MaximumConcurrentAccess>
   </throttle:ThrottleAssertion>
 </wsp:Policy>
</policy>
```

Business Scenario:
In order to ensure timely service to all customers, HiRollerBank wants to limit the amount of messages a customer can send over a period of time so as to not slow down service for other customers. Limit the customer by IP address and only allow them 5 requests/messages every 1 minute. However, HiRollerBank will allow 4 different users from the same customer to make their 5 requests concurrently. Throttling can be set at the ESB level, Sequence level or for REST APIs, at Resource level. This example will throttle at the Sequence level.

Detailed Steps:

1. In Developer Studio, create a *New ESB Config Project* called **QoS**
2. Right-click on this project and click *New → Create new Proxy Service* and set:
    - Proxy Service Name: **ThrottleSequence**
    - Proxy Service Type: **Pass Through Proxy**
    - Target EndPoint: **Enter URL**
    - URL: **http://localhost/HiRollerBankWS/services/accounts**
3. Click Finish to build *ThrottleSequence* Proxy
4. Add a *Throttling* mediator before the *Send* mediator in the *inSequence*
5. Add a *Log* mediator to the *OnAccept* process of the *Throttle* mediator
    - Log Level: **Custom**
    - Log Separator: **,**
    - Properties: **Message Will Pass**
6. Add a *Log* mediator to the *OnReject* process of the *Throttle* mediator
    - Log Level: **Custom**
    - Log Separator: **,**
    - Properties: **Message Will NOT Pass - Past Threshold**
7. For *Throttle* mediator properties, set:
    - Policy Type: **INLINE**
    - For Policy Entries, in *Throttle Policy Editor*:
        - Set Max Concurrent Access: **4**
        - Click *Add* and set:
            - Range: **other** (this means everything, instead of a specific IP)
            - Type: **IP**
            - Max Request Count: **5**

- Unit Time: **60000**
- Prohibit Time Period: **60**
- Access: **Control**

Notice that you could provide an IP in Range and set Access to Allow or Deny to fully allow or block IPs all together.

1. Delete exiting *Send* mediator in *InSequence*
1. Add a *Send* mediator in the *OnAccept* flow, after *Log* mediator
1. Add an *Address EndPoint* inside the *Send* mediator and configure:
    - Set URI to **http://localhost:9764/HiRolleBankWS/services/accounts**
1. Add a *Drop* mediator in the *OnReject* flow, after *Log* mediator
1. To deploy, add to CAR file (pom.xml file) and Redeploy
1. To test, in SOAP UI, create a New SOAP UI project
    - Set *Initial WSDL*to**http://localhost:9764/HiRolleBankWS/services/accounts/?wsdl**
1. Now open*Request1* under *mediate*
    1. Set URL to: **http://localhost:8280/services/ThrottleSequence**
1. Run it 5 times and get successful responses. After 6th, you will not get a response.
1. Check logs to see "*Message Will NOT Pass - Past Threshold*" on 6th attempt.

LAB 10: Configuring Quality of Service (QoS) - Caching

Training Objective: Practice configuration of caching.

Business Scenario:
In order to ensure timely service to all customers, HiRollerBank wants to take advantage of reusing responses it has already delivered; therefore, it will cache responses to avoid requesting a response from the backend service, but instead reuse existing responses.

High-level Steps:

1. Configure the Cache mediator
2. Deploy
3. Test: Make several requests to ESB which will be passed to AS. Shutdown AS - ESB will cache responses and you will still get responses.

Detailed Steps:

1. Right-click your **QoS** project and click *New → Create new Proxy Service,* and set:
    - Proxy Service Name: **CachingMediator**
    - Proxy Service Type: **Pass Through Proxy**
    - Target EndPoint: **Enter URL**
    - URL: **http://localhost/HiRollerBankWS/services/accounts**
2. Click Finish to build *CachingMediator* Proxy
3. Add *Cache* mediator to *inSequence* before the *Send* mediator, and configure:
    - Cache Id: **cache_mediator_test**
    - Cache Scope: **per-mediator**
    - Cache Type: **FINDER**
    - Hash Generator: **org.wso2.caching.digest.DOMHashGenerator**
4. To Deploy, add to CAR file (pom.xml file) and Redeploy
5. To test, in SOAP UI, New SOAP UI project
    - Initial WSDL: **http://localhost:9764/HiRolleBankWS/services/accounts/?wsdl**
6. Open the *Request1* under *mediate*
7. Set URL to: **http://localhost:8280/services/CacheMediator**
8. Make 15-20 requests so that it caches
9. Shutdown App Server so it cannot respond
10. In SoapUI, resend request and you will get a response despite AS being down

LAB 11: Configuring Quality of Service (QoS) - Optional

Training Objective: Practice configuration of WS-Security for outgoing messages
https://docs.wso2.com/display/ESB481/Sample+100%3A+Using+WS-Security+for+Outgoing+Messages

Introduction

This sample demonstrates how you can use the ESB to connect to endpoints with WS-Security for outgoing messages.

In this sample the stock quote client sends a request without WS-Security. The ESB is configured to enable WS-Security as per the policy specified in thepolicy_3.xml file, for outgoing messages to the SecureStockQuoteService endpoint hosted on the Axis2 instance.

Prerequisites

- Download and install the unlimited strength policy files for your JDK before using Apache Rampart. To download the policy files, go to http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html.
- For a list of general prerequisites, see Prerequisites to Start the ESB Samples.

LAB 12: Using Embedded Registry

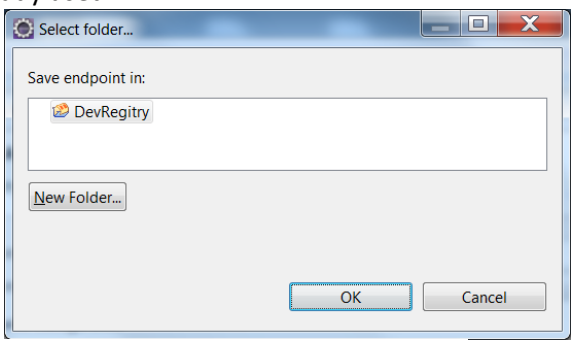
Training Objective: Practice configuration data externalization and Registry deployment

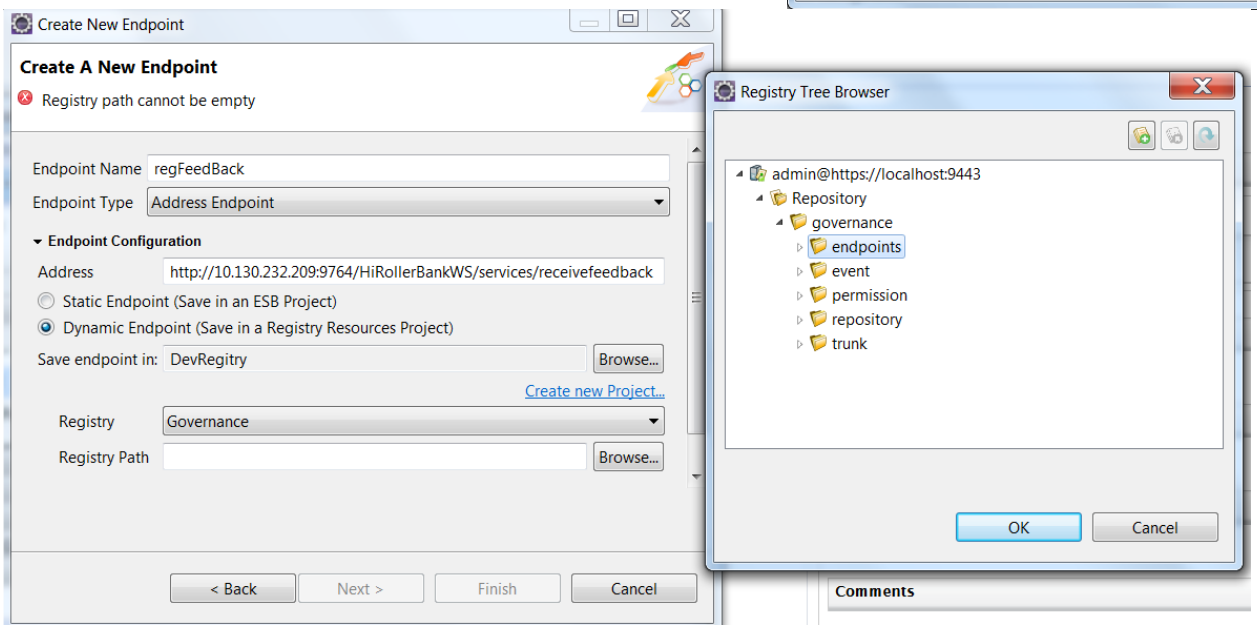
Introduction

This sample demonstrates how you can externalize configuration values to the registry (endpoints for instance) to use the same Capp CAR through all environments.

High-level Steps:

1. Create A Registry Resource Project named **DevRegistry**
2. Create an endpoint in this project that target a previously used httpEndoint from any former lab
    - For example : the feedback service whe have already used
    - Choose “Create new”
    - Name the endpoint “regFeedback”
    - Choose the type (here it is AddressEndoint)
    - choose a Dynamic Endpoint
    - select the Registry Project as a container
    - Target the Governance Registry and browse to point the path
    - You may have to provide the admincredentials

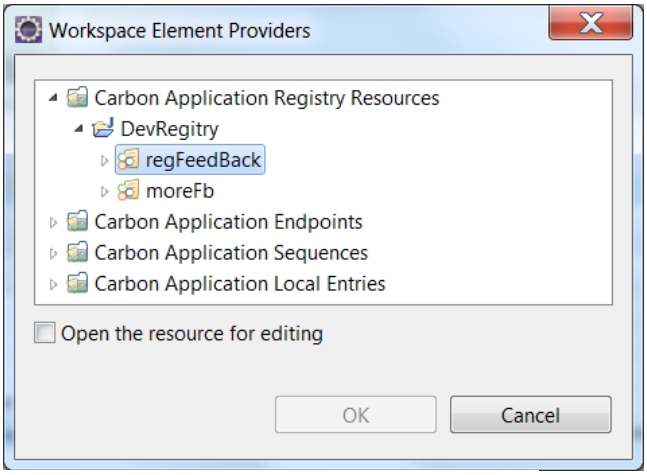




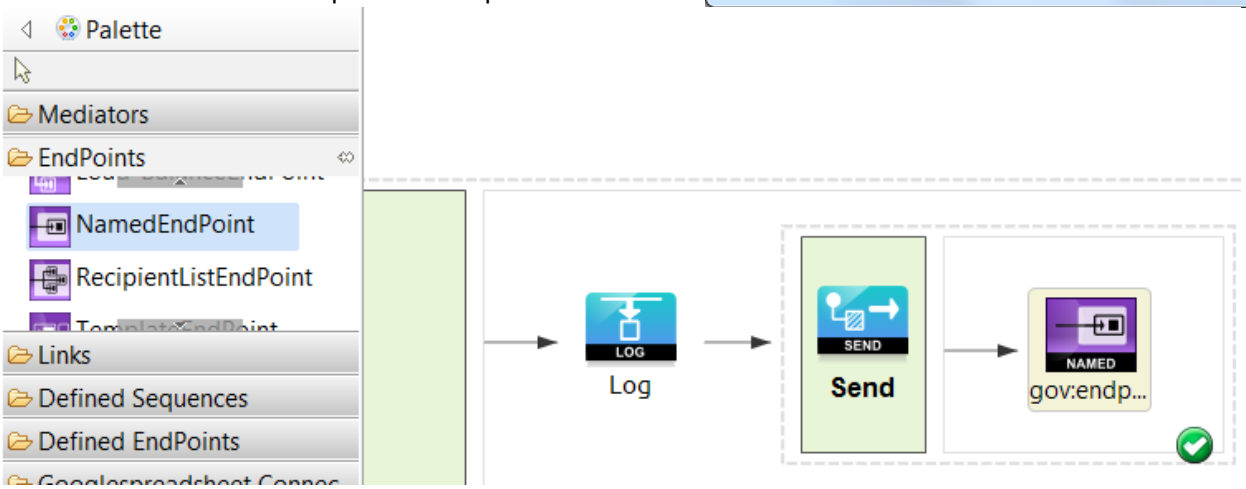
to connect to the registry and your ESB Server must be runnin to make it available.

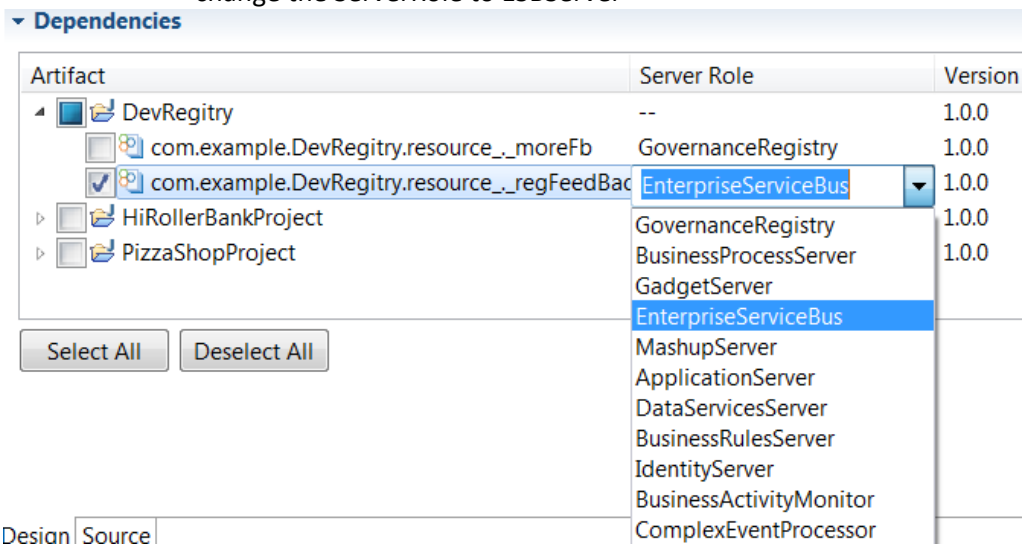
- Browse to the endpoints directory and select it.

3. Now we can use the enpoint in one of our proxy Services
    - Delete a former address enpoint
    - Create a named endpoint and choose from Workspace or Registry



4. How to deploy
    - Create a Capp project to deploy your Registry artifact
    - Check the endpoint on the pom file and change the ServerRole to ESBServer





    - Deploy the new CaR file and see that the endpoint is in the Registry and is used by your proxy