Neuron ESB Training: Web Services

Overview

Many companies rely on web services to perform vital tasks within their organization. When looking to implement an integration strategy, it is important for them to know and understand how Neuron ESB will be able to support the web services they currently use, as well as provide them flexibility for expanding and maintaining their web services done the road.

This document will provide you with an understanding of how Neuron ESB can support the use of services in your integration strategy. We will cover the following topics as part of this tutorial:

- Service Endpoints
- One-Way Scenario
- Request-Reply Scenario
- REST Requests
- Using Service Endpoints with Business Processes
- Using Web Service Security
- Using Certificates
- Using OAuth
- Routing and Versioning

Prerequisites

The scenarios in this tutorial require you to have the following prerequisites

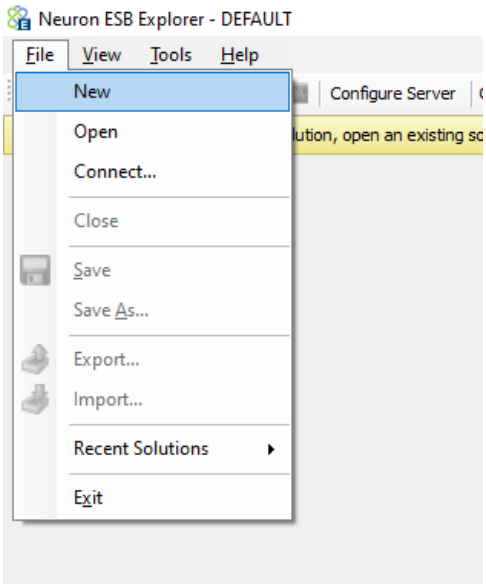
- NET 4.7.2
    - (Neuron ESB Version 3.7 requires 4.7.2, Version 3.7.5 requires 4.8)
- Visual Studio 2013
- Understanding of Visual Studio and how to work within its environment
- An instance of Neuron ESB installed on your environment
- Neuron ESB Fundamentals Training or equivalent experience
- Experience with Web Services

Tutorial Prep

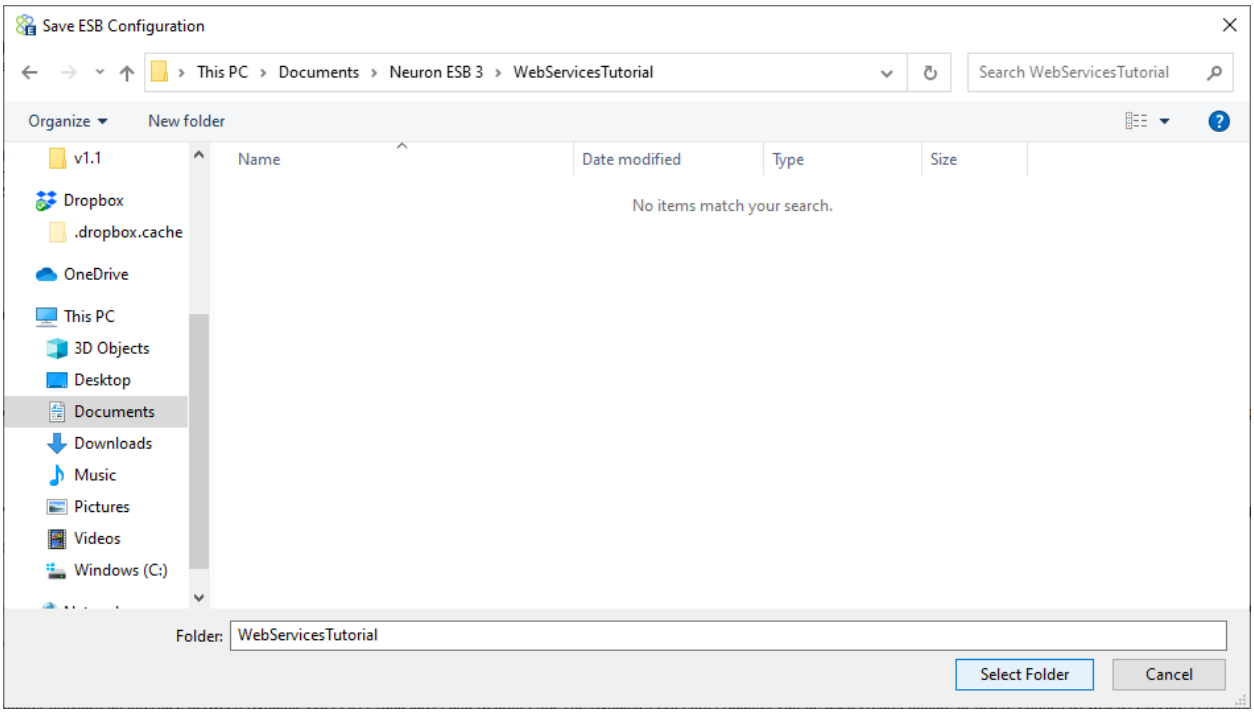
In order to complete this tutorial, you will need both a Neuron ESB configuration and a Visual Studio Project. The majority of both will be filled in as we go along. However, there is some initial prep work that can be done prior to moving into specific scenarios, that we can do to prepare.

Neuron ESB Configuration

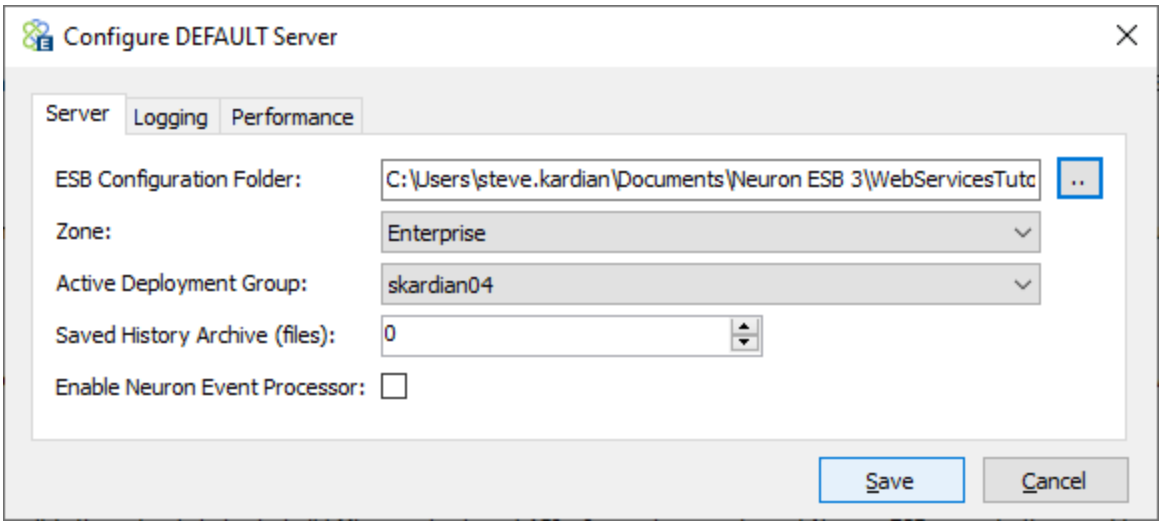
1. Open the Neuron ESB Explorer and select New from the File menu.

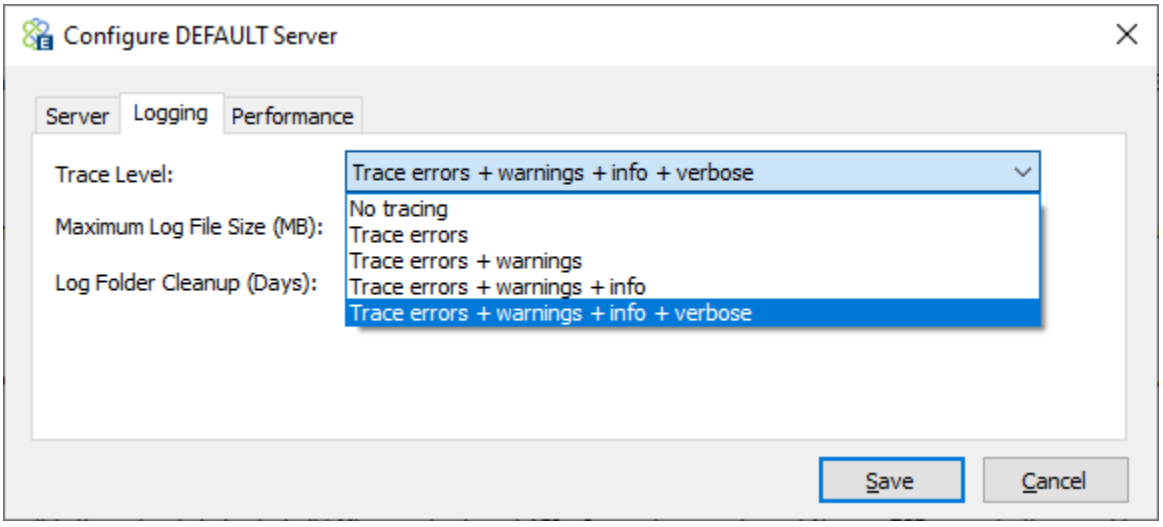


2. Save the configuration, naming it WebServicesTutorial

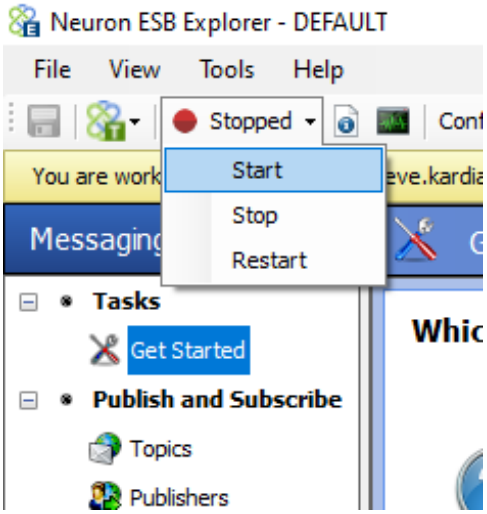


3. Configure the Neuron ESB instance to use the WebServicesTutorial, with the logging level set to Verbose



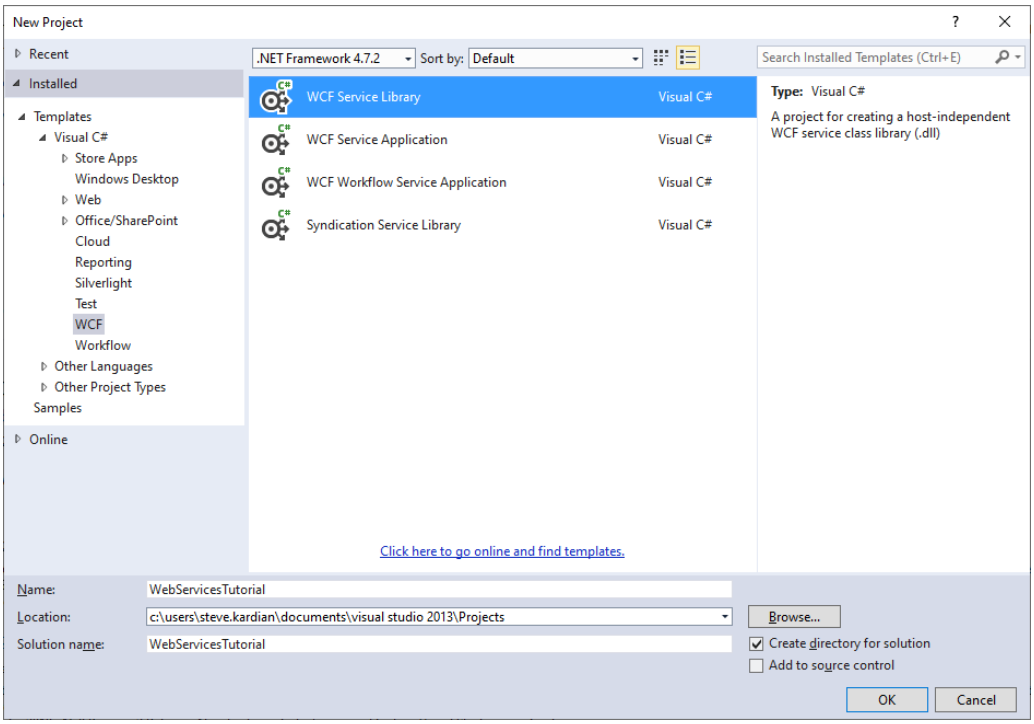


4. Start the Neuron ESB instance. If the Neuron ESB instance is already running, chose restart instead

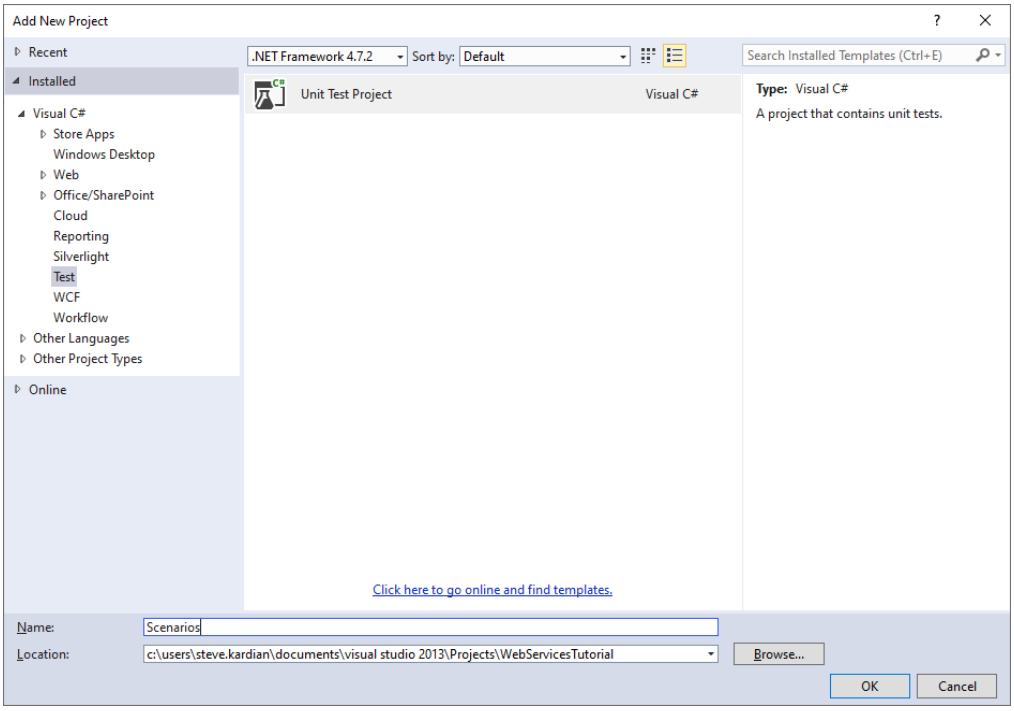


Visual Studio Project

1. Create a new Visual Studio Project of type WCF service library, named WebServicesTutorial, ensuring that it is targeting the 4.7.2 framework



2. Add a project of type Unit Test, called Scenarios, to the solution.



3. Right click the Scenarios project, select Add -> Reference and add System.ServiceModel and System.Runtime.Serialization to the project.
4. Right click the Scenarios project, select Add -> New Item and add an Application Configuration File to the project (leave the default name of App.config)
5. Open the UnitTest1 class, of the Scenarios project, and add `using System.ServiceModel;` to the list of using statements.

Service Endpoints

Neuron ESB includes a Service Broker built on the Microsoft Windows Communication Foundation (WCF). The Service Broker enables organizations to deploy Neuron ESB as a Service Gateway, providing mediation, security, hosting and a number of other services. Neuron ESB is built on the principal of SOA and Microservices, hence it's a natural choice to also build Microservice based APIs. Neuron ESB, allows organizations to easily create, test and deploy REST APIs or SOAP based services.

> *Neuron ESB is built on top of the Microsoft Windows Communication Foundation (WCF). This allows users to easily extend and customize Neuron ESB Service Endpoints using custom Behaviors, Bindings and, even Configuration Files. https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf*

There are two core components within Neuron ESB that facilitate basic service capabilities, Service Connectors and Client Connectors.

Service Connectors are essentially registrations within Neuron ESB that point to existing services (i.e. REST or SOAP APIs) hosted within an organization, partner or cloud domain.

Client Connectors on the other hand are REST or SOAP APIs created using the Neuron ESB toolset (e.g. Neuron ESB Explorer) and hosted by Neuron ESB. These are live services that can accept external requests, initiate business processing logic and/or broker to external services or applications.

SOAP vs REST

SOAP vs REST is a question that is often asked when creating a web service. Is one better than the other? Which one should be used? The answer, unfortunately, is not simple and straightforward. Each has their own benefits and disadvantages, and most times the decision comes down to the scenario one is trying to achieve. Having a firm understanding of both SOAP and REST, as well as the difference between them, is usually the only way to truly make the right decision.

SOAP

SOAP, which stands for Simple Object Access Protocol, is a method of transferring messages, or small amounts of information, over the Internet. SOAP messages are formatted in XML and are typically sent using HTTP (hypertext transfer protocol). Both are widely supported data transmission standards.

The XML used by SOAP to make requests and receive responses can become extremely complex. In some programming languages, you need to build those requests manually, which becomes problematic because SOAP is intolerant of errors. However, other languages can use shortcuts that SOAP provides. They can help you reduce the effort required to create the request and to parse the response. In fact, when working with .NET languages, you never even see the XML.

This is due in part to the Web Services Description Language (WSDL). This is another file that's associated with SOAP. It provides a definition of how the web service works, so that when you create a reference to it, the IDE can completely automate the process. So, the difficulty of using SOAP depends to a large degree on the language you use.

One of the most important SOAP features is built-in error handling. If there's a problem with your request, the response contains error information that you can use to fix the problem. Given that you might not own the Web service, this particular feature is extremely important; otherwise you would be left guessing as to why things didn't work. The error reporting even provides standardized codes so that it's possible to automate some error handling tasks in your code.

Benefits

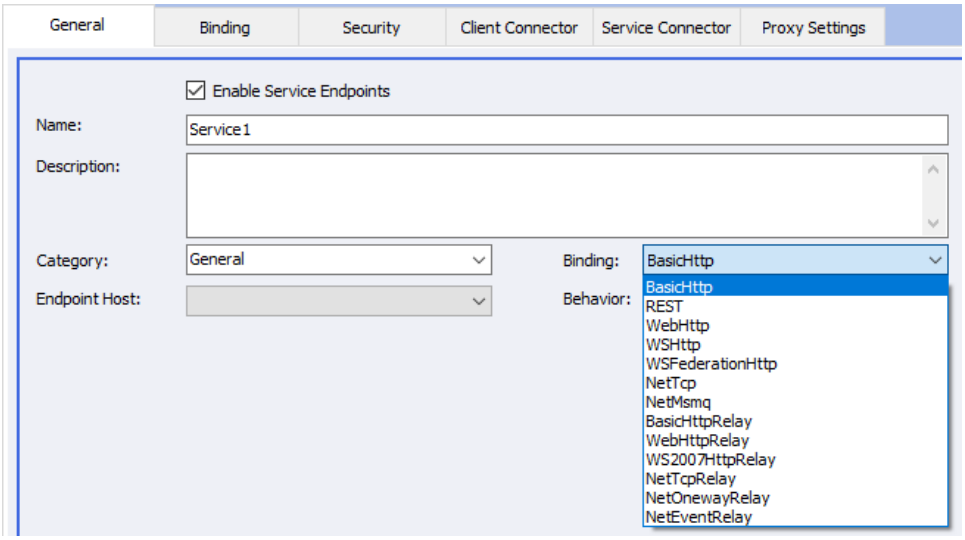
- SOAP's standard HTTP protocol makes it easier for it to operate across firewalls and proxies without modifications to the SOAP protocol itself.
- While it's rarely needed, some use cases require greater transactional reliability than what can be achieved with HTTP (which limits REST in this capacity). If you need ACID-compliant transactions, SOAP is the way to go.
- In some cases, designing SOAP services can actually be less complex compared to REST. For web services that support complex operations, requiring content and context to be maintained, designing a SOAP service requires less coding in the application layer for transactions, security, trust, and other elements.

- SOAP is highly extensible through other protocols and technologies. In addition to WS-Security, SOAP supports WS-Addressing, WS-Coordination, WS-ReliableMessaging, and a host of other web services standards.

In Neuron ESB

A service endpoint in Neuron ESB, whether it be a client connector or a service connector, can be configured to use SOAP by selecting a SOAP binding from the Binding dropdown list on the General tab of the endpoint. The following bindings are SOAP bindings:

- BasicHttp (SOAP 1.1)
- WSHttp (SOAP 1.2)
- WSFederatedHttp (SOAP 1.2)
- BasicHttpRelay (SOAP 1.1)
- WS2007HttpRelay (SOAP 1.2)



REST

REST, or Representational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. REST-compliant systems, often called RESTful systems, are characterized by how they are stateless and separate the concerns of client and server.

Instead of using XML to make a request, REST (usually) relies on a simple URL. In some situations, you must provide additional information, but most web services using REST rely exclusively on the URL approach. REST can use four different HTTP 1.1 verbs (GET, POST, PUT, and DELETE) to perform tasks.

REST doesn't have to use XML to provide the response. You can find REST-based web services that output the data in Command Separated Value (CSV), JavaScript Object Notation (JSON) and Really Simple Syndication (RSS). The point is you can obtain the output you need, in a form that's easy to parse within the language you're using for your application.

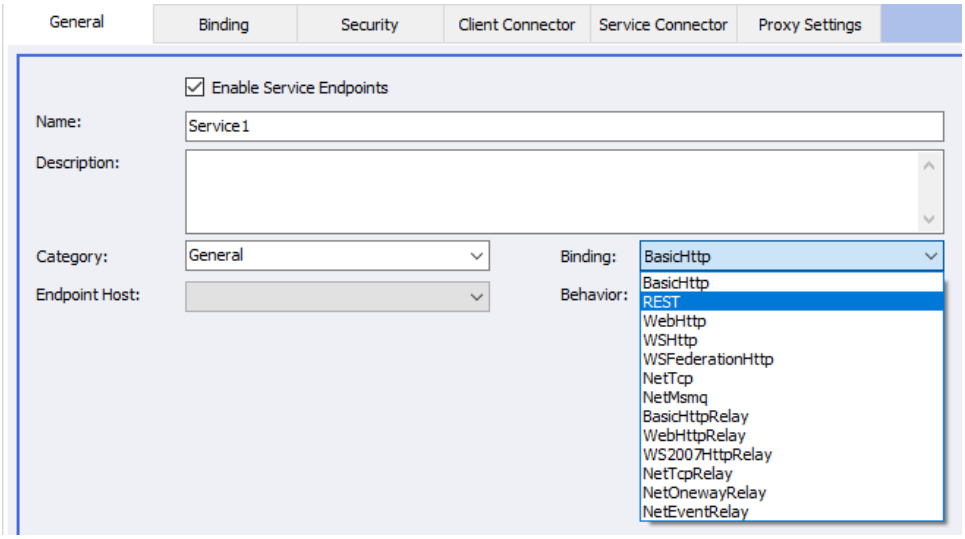
Benefits

- REST allows a greater variety of data formats, whereas SOAP only allows XML.
- Coupled with JSON (which typically works better with data and offers faster parsing), REST is generally considered easier to work with.
- Thanks to JSON, REST offers better support for browser clients.
- REST provides superior performance, particularly through caching for information that's not altered and not dynamic.
- REST is generally faster and uses less bandwidth. It's also easier to integrate with existing websites with no need to refactor site infrastructure. This enables developers to work faster rather than spend time rewriting a site from scratch. Instead, they can simply add additional functionality.

In Neuron ESB

A service endpoint in Neuron ESB, whether it be a client connector or a service connector, can be configured to use REST by selecting a REST binding from the Binding dropdown list on the General tab of the endpoint. The following bindings are REST bindings:

- REST
- WebHttp
- WebHttpRelay



More information on WCF Bindings can be found at https://docs.microsoft.com/en-us/dotnet/framework/wcf/bindings-overview

Enforcement of Contracts

Neuron ESB Service Endpoints are loosely typed. This means Neuron ESB Service Endpoints do not require a specific WSDL/Swagger or contract at runtime. This also means Neuron ESB does not generate

WSDL or Swagger for its service endpoints. This allows clients to send more than one request message type to the same endpoint and those requests can be from completely unrelated services.

Alternatively, Neuron ESB does allow uses to import WSDL and Swagger documents and then assign these to Client Connectors. However, they are not used for runtime validation.

This does not mean that Neuron ESB cannot enforce contracts or a schema. Business processes can be attached to service endpoints, with business process steps, such as XML or JSON Schema validation, implemented to enforce a contract.

Service Routing vs Dynamic Routing

Service routing (or Static Routing) is the process by which Neuron ESB uses custom business logic defined by the user in order to determine which Service Connector to route the request to. The Service Connector to route to can be determined by any number of ways including using information in the headers of the message, content in the body of the message, custom business logic or any combination thereof, evaluated at runtime.

In some cases, rather than creating a unique Service Connector for every possible URL or Local Path, it becomes more manageable to set the properties of the Service Connector at runtime, such as the URL, Local Path, SOAP Action, REST Method, Credential properties, etc. This minimizes the endpoints that are required to be managed as well as reduces the overall runtime overhead of the solution. In other cases, the user may only be able to resolve the appropriate elements of the Service Connector at runtime, usually from elements of the incoming request. In this case, Dynamic Routing becomes the preferred solution. Neuron ESB provides Service and Dynamic Routing capabilities using Business Processes, Messaging or Service Routing Tables.

Using Business Processes

Business Processes can be attached to Neuron ESB Client and Service Connectors, regardless of whether they are used as part of the pub/sub model, to accomplish nearly any processing required including calling Web Services inline, using databases, injecting custom code, XSLT transforms etc.

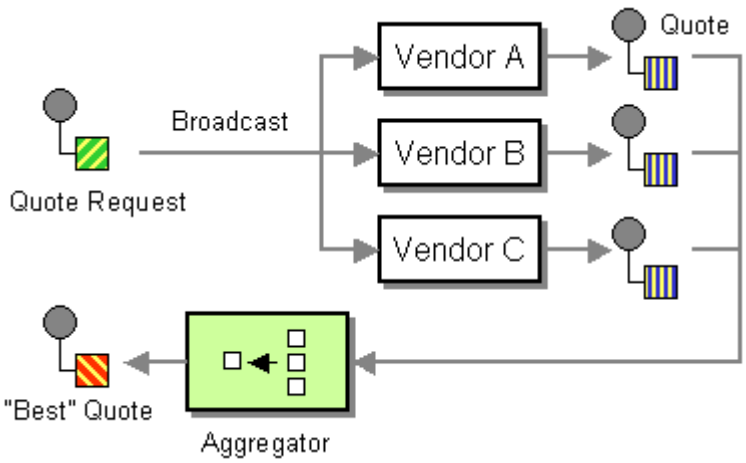
Neuron ESB Business Process Documentation can be found here: https://www.neuronesb.com/article/kbtopic/doc-development-business-processes/

This allows organizations to use Neuron ESB to create and host services that normally would be self-hosted or hosted in IIS. That's because Neuron ESB Business Processes and Client Connectors can be used to create a complex real time orchestration.

This functionality also allows organizations to use Business Processes to determine the correct Service Connector to route the message to. A single Business Process can route a request message to multiple Service Connectors either sequentially or in parallel. Additionally, users can alter elements of the service request (i.e. URL, Action, Local Path, etc.) dynamically as described in the Dynamic Routing section above.

Routing via Business Processes is recommended when custom business logic needs to be executed upon

Scatter Gather Pattern – This sample ships with Neuron ESB and presents a simple and reliable design using the Neuron Business Process Designer and Runtime to resolve a common scatter-gather application integration problem described in the book Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.



https://www.neuronesb.com/article/kb/scatter-gather-pattern/

receipt of the request message prior to sending that message to a Service Connector. Whether that logic is data transformation, aggregation, dispersal or any other unique business logic which can only be achieved by a Business Process. Routing via Business Processes is also mandatory when routing from an endpoint other than a Client Connector to a Service Connector, such as from an Adapter Endpoint. Otherwise simple pass thru routing and dynamic routing can be easily configured without a Business Process when the Service Route Processing Mode is used.

Processing Modes

Messaging

When enabled for “Messaging”, both the Client Connector and Service Connector can participate in the pub/sub Messaging infrastructure that Neuron ESB provides, essentially decoupling one from the other using a Topic. When “Messaging” is enabled, a Client Connector receives the inbound request and “publishes” it to a Topic. A “subscribing” Service Connector would then receive the request, call the external service, publishing the response back to the Topic, where the response would be forwarded back to the waiting Client Connector.

This can be useful if you are doing either of the two:

- One Way Service Calls – In this scenario, the service configuration is a Datagram vs a Request-Reply call, the calling client is not expecting a reply
- Request-Reply Web Service Call with a long running process – in this scenario, the calling client may need an immediate ack/nack returned, but the original request requires more extensive processing than a normal request wait time would allow. For example, an incoming order request that requires a long running workflow to process the request, which normally may take minutes, hours or days to complete.

Using Messaging to decouple services always requires the following (with a notable exception):

- A Publisher configured for the Client Connector (unless configured to use “Business Process” vs “Messaging”)
- A Subscriber configured for the Service Connector
- One or more Topics configured that both the Publisher and Subscriber has rights to

Because a Publisher and Subscriber are always required, it means Business Processes can be attached to them and subsequently executed to either pre or post process a request or reply message.

Additionally, when Messaging is used to decouple Service Endpoints, there is usually more overhead required at runtime. Some considerations include the following:

- Increased Complexity
    - Need to create a Topic and Publisher before configuring a Client Connector.
    - Need to create a Subscriber and Topic before configuring a Service Connector
- Reduced Runtime Utilization
    - The Topic Publishing Service must be created.
    - More CPU and Memory required for both Service and Client Connector
- Slower Startup Time
    - The Topic, and Service Connectors must be loaded by the master runtime and endpoint host
- Higher Latency and reduced concurrency
    - All requests are brokered by a Publisher object through the Messaging System

Generally, if working with pure Request-Reply scenarios that do not require an intermediate long running workflow, users should choose either the “Business Process” or “Service Route” processing mode option.

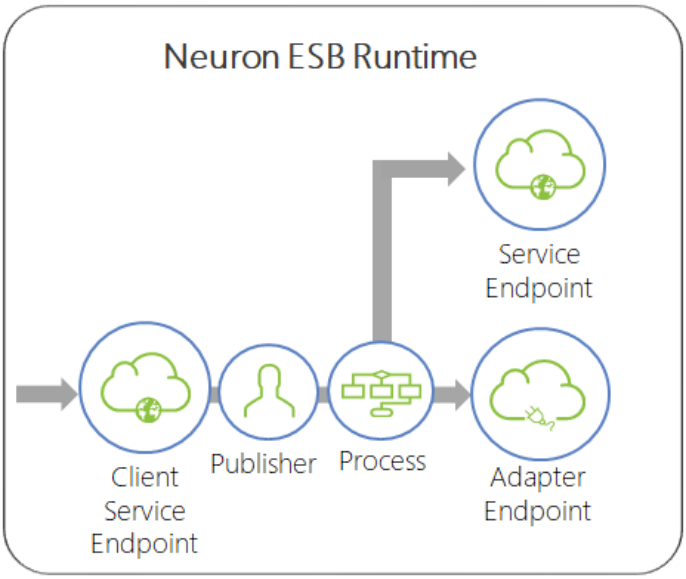
> *When using Messaging to decouple service endpoints, the general rule of thumb should always be to use TCP based Topics rather than a Rabbit MQ or MSMQ based Topic, unless the subscribing endpoint is a Workflow Endpoint. Anytime a queued Topic is used to decouple request/reply calls, all calls are essentially sequentially processed due to the nature of FIFO Queues.*

Business Process

Users can choose to disable the Messaging option located on the respective tab of the Client Connector (i.e. publisher role) and Service Connector (i.e. subscriber role) of the Service Endpoint.

Service Connectors

Just as with Adapter Endpoints, subscribe side Service Endpoints (i.e. Service Connectors) can be executed directly within a Business Process or Workflow.

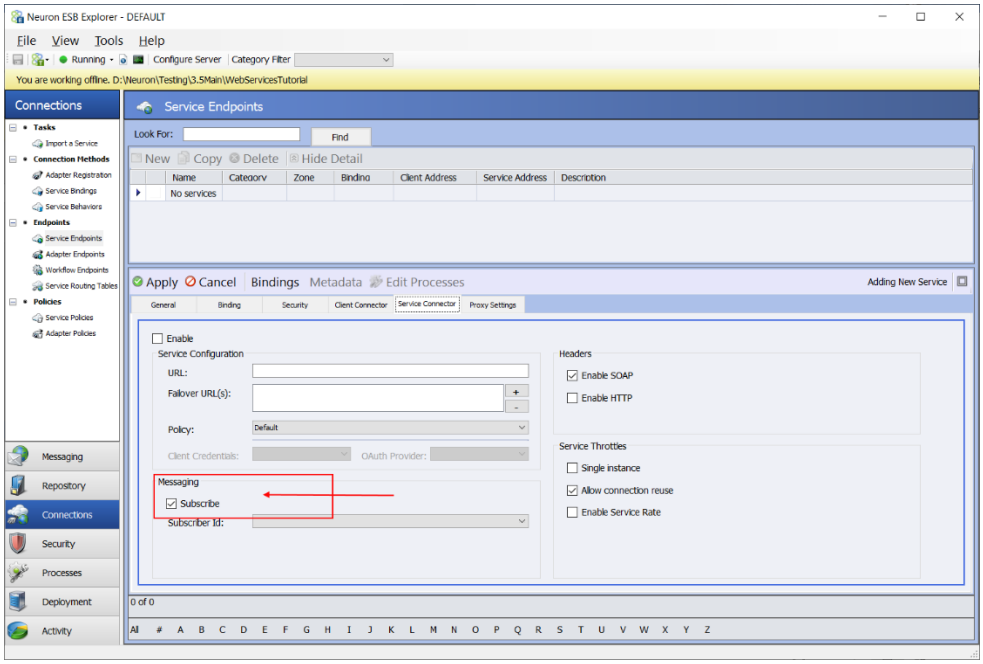


In Neuron ESB, Adapter and Service Endpoints can be executed directly by a Neuron ESB Business Process rather than having the original request forwarded across the Neuron ESB Messaging sub system.

When users specifically choose to disable the Messaging option located on the Service Connector tab of the Service Endpoint, it allows the developer to create a Service Connector without needing to create a Topic or Subscriber. At runtime, the Service Endpoint can be called directly within the Business Process or Workflow but will not be a “live” subscriber to messages published by other systems. The immediate benefits to this approach are:

- Reduced Complexity

    - No need to create a Topic or Subscriber before configuring a Service Connector.
- Reduced Runtime Utilization
    - The Service Connector does not need to be loaded in memory with its own resources. The Topic Publishing Service is never created.
- Faster Startup Time
    - Since there is no subscribing Service Connector, the master runtime or endpoint host does not attempt to start it as a manageable endpoint.
    - There is no Topic that needs to be created and loaded by the master runtime.



Neuron ESB Explorer – Service Connector User Interface. This has a Messaging option which can be enabled or disabled by selecting the "Subscribe" checkbox encircled in Red.

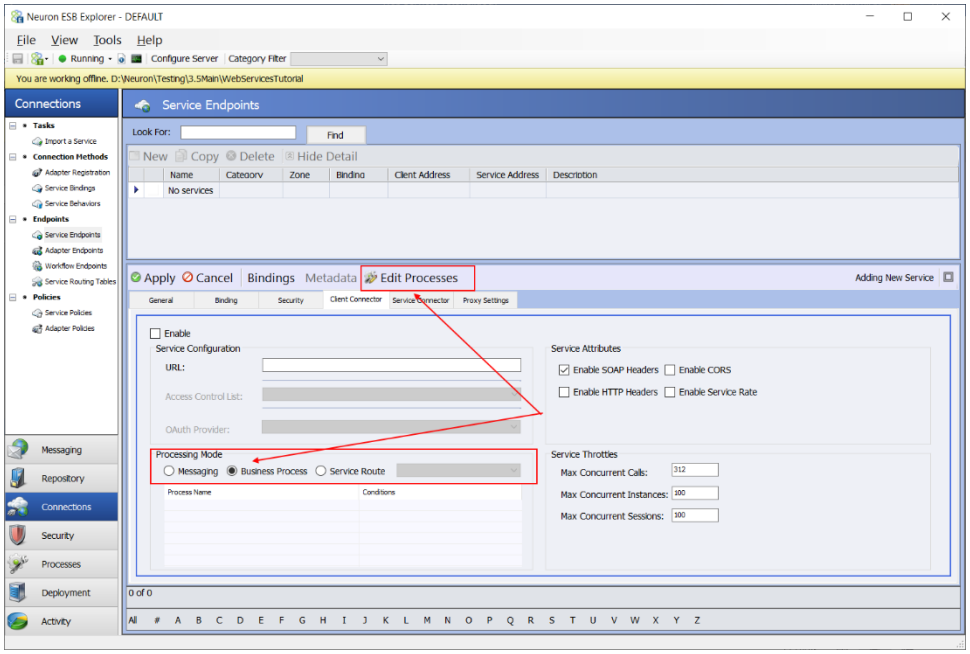
Client Connectors

Although Messaging can also be disabled for Client Connectors, it is for a different purpose than that of Service Connectors. Rather than disabling live hosting of the Service Endpoint, selecting the "Business Process" mode allows the user to configure a Client Connector to call and execute Business Processes directly, without the need to configure the Client Connector with a Publisher and Topic.

Users can easily expose a Business Process as a web service (REST or SOAP) in 3 simple steps:

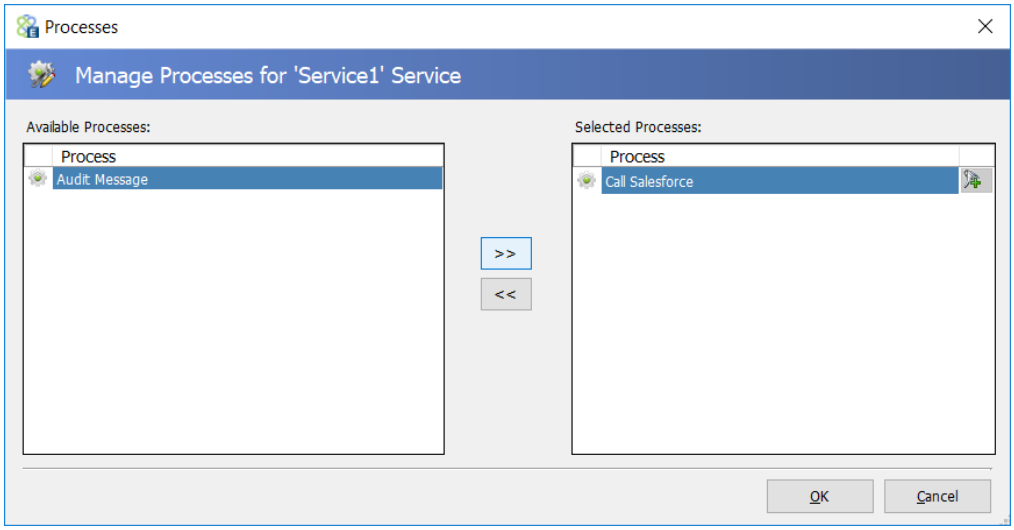
1. *Create a Business Process*
2. *Create a Client Connector*
3. *Configure the Client Connector to call the Business Process*

Selecting the Business Process option of the Processing Mode section of the Client Connector effectively disables the Messaging option and allows users to select one or more Business Processes to execute directly from the Service Endpoint UI.



Neuron ESB Explorer – Client Connector User Interface. This has a Processing Mode group box which contains a Messaging, Service Route and Business Process mode option. If Messaging is selected, controls appear to allow users to select a Publisher and Topic. If Business Process is selected, users can select the Edit Processes toolbar button to select Business Processes to execute (encircled in Red).

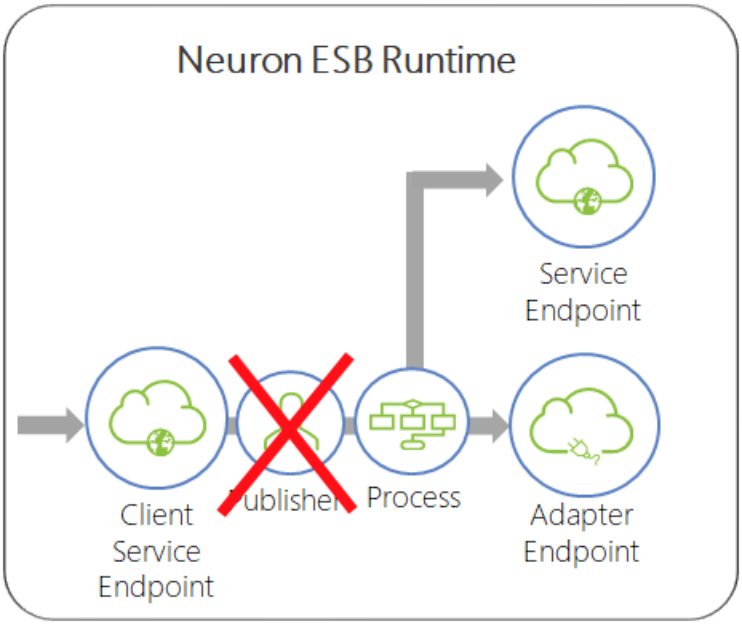
Selecting the "Edit Processes" toolbar button, displays the same Processes dialog used to select Business Processes for Publishers and Subscribers.



Processes dialog – Launched from the Edit Processes toolbar button of the Client Connector. Allows user to select one or more Business Processes for the Client Connector to directly execute.

Once a Client Connector is configured to execute Business Processes directly through the new Processing Mode option, developers do not need to create a Publisher or Topic as part of the configuration process. Additionally, when creating Business Processes, users can forego the placement of the Cancel Process Step as the last step in the Business Process when dealing with request/reply patterns. Client Connectors directly executing Business Processes incur other benefits as well including

- Reduced Complexity
    - No need to create a Topic or Publisher before configuring a Client Connector.
    - No mapping of Business Process to Publisher and Publisher to Client Connector
    - No need to use a Cancel Process Step in Business Processes to return the reply message to the calling client or prevent a message from being forwarded to a Topic.
- Reduced Runtime Utilization
    - The Topic Publishing Service is never created.
    - Less processing and memory consumed by the Client Connector
- Faster Startup Time
    - The Topic is not created and loaded by the master runtime.
- Better Performance
    - All requests are no longer being brokered by a Publisher object, latency is decreased, and concurrency is increased when compared to the "Messaging" process mode.



In Neuron ESB, Client Connectors can execute Business Processes directly, without an intermediate Publisher brokering messages across the Neuron ESB Messaging sub system.

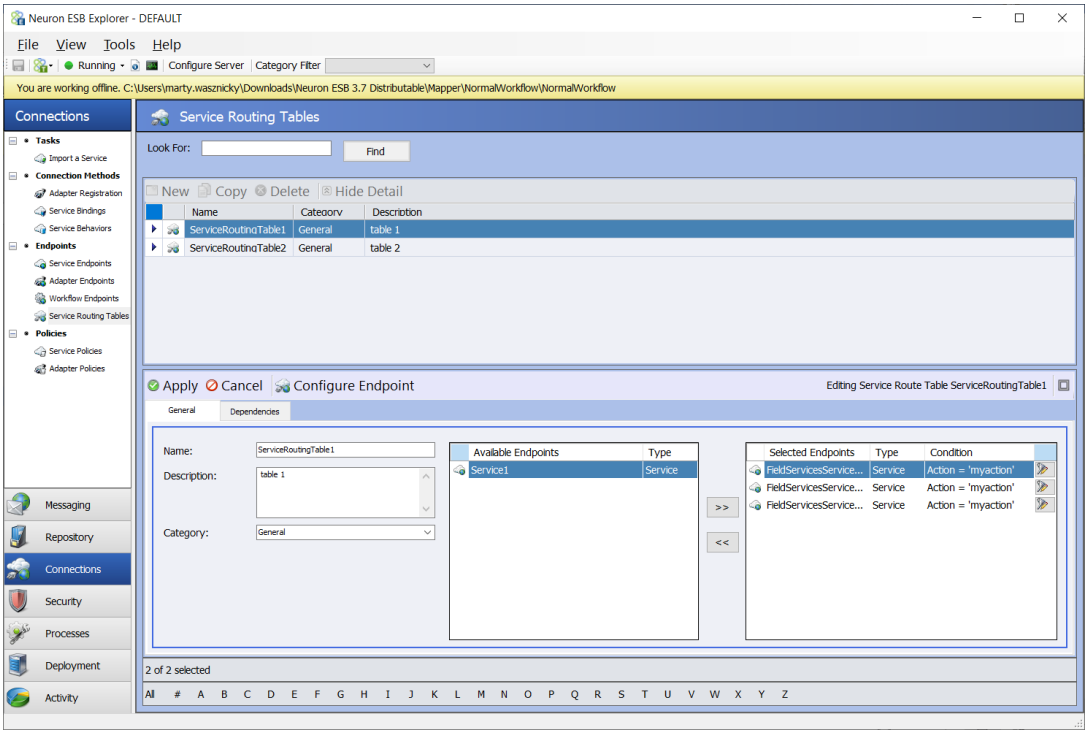
Limitations

Client Connectors configured for direct execution of Business Processes should be the preferred mode of operation for users. This will generally provide simpler and more efficient execution of messages and Business Processes. Even though in this configuration there is no Publisher or Topic associated with an endpoint, messages received by the Client Connector can still be published to topics using the Publish Process Step within the Business Processes it's executing. All Topic transports are supported. The only

use case that is not supported is using MSMQ based topics when the Publish Step is configured with the Request Semantic.

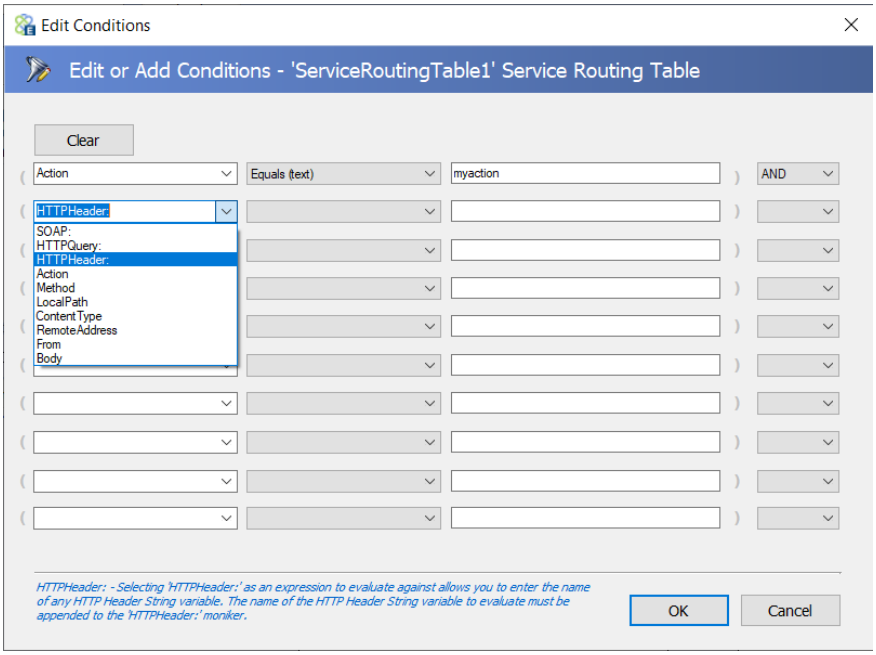
Service Route

Service Routing Tables (introduced in Neuron ESB version 3.7.5) provide users with the ability to setup pre-defined sets of Service Connectors that a client connector can use to determine where to route messages received by that client connector, based on conditional checks.



Neuron ESB Explorer – Service Routing Table User Interface. Users can select existing Service Connectors to route to for Pass thru scenarios or, create multiple instances of existing Service Connectors for Dynamic Routing scenarios. Which Service Connector instance to route to can be determined by configuring AND/OR conditions.

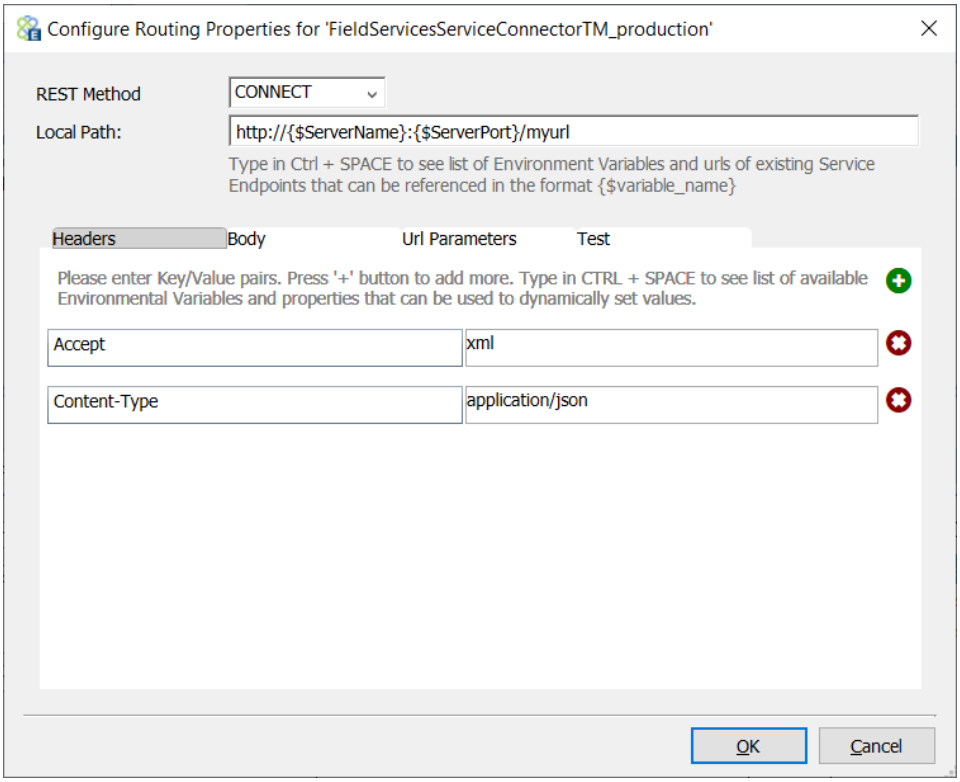
Routing tables can use header properties (SOAP headers, SOAP Action, HTTP headers, Query String variables, local path, etc.…) or parts of the message body in their conditional checks in order to determine which service a message should be delivered to. These conditional checks can be used individually or chained together to form a much more intricate and precise condition.



Neuron ESB Explorer – Condition Editor for the Service Routing Table. Conditions are evaluated against the incoming request at runtime to determine which Service Connector instance to route.

They can also be configured using Environment Variables and to return a Test Response Message, rather than forwarding the call directly to the external service. This is useful in load testing and QA environments.

Service Routing Table not only allow users to specify the conditions that governs which endpoint to send the request to but can also dynamically assign properties of the endpoint when those conditions are met. The configurable aspects of the service connector can use static input as well as environment variables. This allows for configuration of pass thru and dynamic routing solutions without the need for an intermediate Business Process or code.



Neuron ESB Explorer – Service Endpoint Configuration Editor. Opened by selecting the "Configure Endpoint" short cut menu on the Service Connector instance within the Service Routing Table. Allows users to use any element of the incoming request, including Environment Variables to determine how the Service Connector will be configured at runtime. If not used, the default configuration and URL of the Service Connector is used.

Routing via a Service Routing Table is recommended when business logic does not have to be executed on the message prior to delivery of that message to a Service Connector, such as in the case of a passthrough or dynamic routing scenario. Some of the use cases this processing mode more easily enables are:

- Dynamic Routing based on conditions
- REST to SOAP. New clients can use java script friendly REST API to inter operate with legacy WCF services.
- SOAP to REST. Existing WCF clients can upgrade to new REST APIs without changing their code.
- POST to GET. On many occasions simple websites such as WordPress can't do POST easily.
- Query String to Custom Header and vice versa. If many method calls require the same query string, they can be simplified by moving query string to HTTP header and vice versa.
- Return a Test Response instead of the service endpoint response message. Test Response can contain custom HTTP headers or query strings to facilitate testing. It can also contain selected properties of request body.
- Append standard query strings such as API key automatically to service endpoint request
- Route to different endpoints or append alternative paths/query strings based on the scope in OAuth token
- Apply different service policy to different Users based on their OAuth token.
  This requires creating multiple service endpoints for same service and then applying business conditions that match token contents to route to the endpoint with appropriate policy.

- Use Local Path to map to query strings. This can support converting traditional query string-based services to support RESTful Service URLs.

Regarding the last bullet, consider the following service:

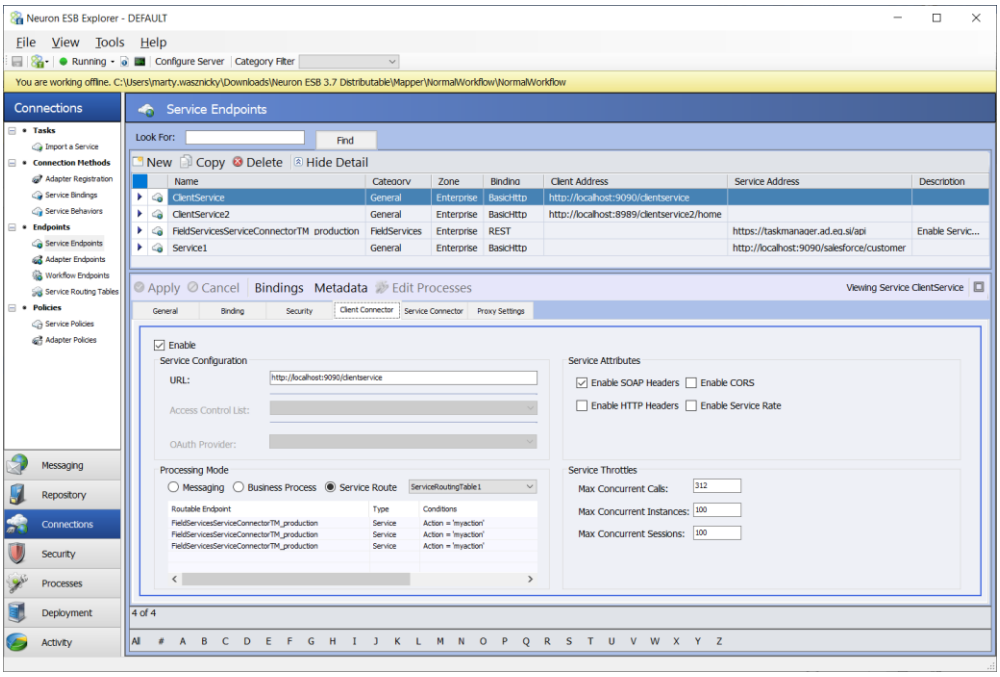
https://Getprice.svc?ProductName=”Shirt”&Size=”Medium”

The sample URL is neither bookmark or search engine friendly. However, the following URL is:

https://Getprivce.svc/Shirt/Medium

This allows consumers to use Local Path instead of Query strings to adapt an older service to newer RESTful URLs.

Once a Service Routing Table has been created in the Neuron ESB Explorer, it can be directly assigned to a Client Connector via the Processing Mode option.

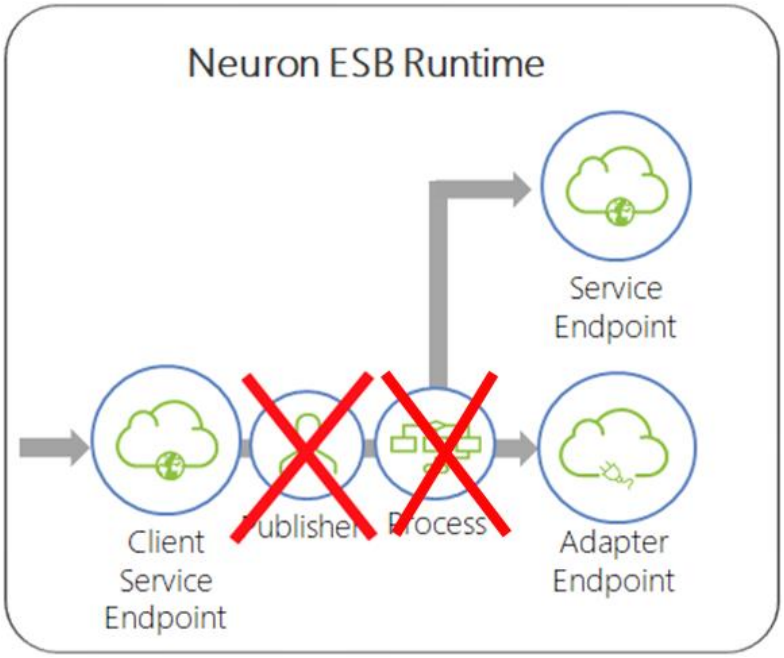


Neuron ESB Explorer – Client Connector User Interface. This has a Processing Mode group box which contains a Messaging, Service Route and Business Process mode option. If Service Route is selected, controls appear to allow users to select the Service Route Table to use.

Client Connectors configured to use Service Routes provide the following benefits

- Reduced Complexity

    - No need to create a Topic or Publisher before configuring a Client Connector.
    - No need to create a Business Process
- Reduced Runtime Utilization
    - The Topic Publishing Service is never created.
    - Less processing and memory consumed by the Client Connector
- Faster Startup Time
    - The Topic is not created and loaded by the master runtime.
- Better Performance
    - All requests are no longer being brokered by a Publisher object or preprocessed by a Business Process; latency is substantially less than all other options while at the same time increasing concurrency.



In Neuron ESB, Client Connectors can route directly to Service Endpoints without the need for a Business Processes or, an intermediate Publisher brokering messages across the Neuron ESB Messaging sub system.

Scenarios

One Way Scenario

Neuron ESB supports client connectors operating as part of multiple message exchange patterns (MEP). The first MEP that we are going to take a look at is the one-way pattern, in which a message is delivered to Neuron ESB via a client connector, but the service is not expecting a response or additional data to be returned to it.

The following steps will prepare the Visual Studio Project to run the test scenario for a one-way message.

1. Open the UnitTest1 class of your Scenarios project
2. Below the namespace declaration but above the class declaration insert some white space by pressing enter and input the following code into the editor.

```
[ServiceContract]
public interface IOneWay
{
    [OperationContract(IsOneWay=true)]
    void OneWay(string arg);
}
```

3. Rename TestMethod1 to OneWayTest
4. Add the following code to OneWayTest

```
ChannelFactory<IOneWay> oneway = new ChannelFactory<IOneWay>("oneway");
IOneWay onewayProxy = oneway.CreateChannel();
onewayProxy.OneWay("No need to respond. I am just one way.");
```

5. Open the App.config file that you created and add the following configuration section inside the `<configuration>` element.

```
<system.serviceModel>
    <client>
      <endpoint address="http://localhost:8080/oneway" binding="basicHttpBinding"
          bindingConfiguration="" contract="Scenarios.IOneWay"
          name="oneway" />
    </client>
  </system.serviceModel>
```

6. Press F6 to save and compile the solution without running it

The Neuron ESB configuration will also need to be configured in order to support the one-way test. The following steps will create the proper artifacts.

7. Restore the Neuron ESB Explorer window
8. Complete the following tasks in the Neuron ESB Explorer
    a. Add a new topic called OneWayTopic
    b. Add a new publisher called OneWayPublisher with a send subscription to OneWayTopic
    c. Add a new subscriber called OneWaySubscriber with a receive subscription to OneWayTopic
    d. Save your changes
9. Navigate to Connections -> Service Endpoints
10. Click on the New button in the main panel
    a. Change the name on the General Tab to OneWayService
    b. Change the Endpoint Host on the General Tab to Neuron ESB Default Host
11. Click on the Binding tab
    a. Change the Messaging Pattern drop down selection for Request-Reply to Datagram, as Datagram is a synonym for one way.
12. Click the Client Connector Tab
    a. Check the Enable checkbox to enable the Client Connector
    b. Enter `http://localhost:8080/oneway` in the URL textbox
    c. Select OneWayPublisher from the Publisher Id drop down list
    d. Select OneWayTopic from the Topic drop down list
13. Click Apply

14. Save your changes

With both the Visual Studio Project and the Neuron ESB configuration prepared, the one-way scenario is ready to be tested.

15. Click on the Tools Menu option for the Neuron ESB Explorer and select Test Client -> 1 Test Client to launch a single test client
16. Select OneWaySubscriber from the Party Id drop down list and click Connect
17. Click on the Receive Tab of the test client
18. Return to Visual Studio and open the Test Explorer Window (This can be opened from the Test Menu option -> Windows -> Test Explorer)
19. Right click OneWayTest and select Run Selected Tests
20. Return to the Test Client to view the message sent from your Visual Studio Project to the Neuron ESB Explorer via the Client Connector that you created

Request-Reply Scenario

The request-reply MEP is one in which the service endpoint receives a message from a source and then returns a reply to that source. It is one of the most common MEPs for a developer to implement in an integration environment, either through necessity or because it has been encouraged by the tools they use to generate services and proxies.

The following steps will prepare the Visual Studio Project to run the test scenario for a request-reply interaction.

1. Open the UnitTest1 class of your Scenarios project
2. Below the namespace declaration but above the class declaration insert some white space by pressing enter and input the following code into the editor

```
[ServiceContract]
public interface IRequestReply
{
    [OperationContract]
    string RequestReply(string arg);
}
```

3. Create a new test method named RequestReplyTest
4. Add the following code to RequestReplyTest

```
ChannelFactory<IRequestReply> requestReply = new
ChannelFactory<IRequestReply>("requestreply");
IRequestReply requestReplyProxy = requestReply.CreateChannel();
requestReplyProxy.RequestReply("I am request reply so I would love to hear back from
you.");
```

5. Open the App.config file that you created and add the following configuration section inside the `<client>` element.

```
<endpoint address="http://localhost:8080/requestreply" binding="basicHttpBinding"
bindingConfiguration="" contract="Scenarios.IRequestReply" name="requestreply" />
```

6. Press F6 to save and compile the solution without running it

The Neuron ESB configuration will also need to be configured in order to support the request-reply test. The following steps will create the proper artifacts.

7. Restore the Neuron ESB Explorer window
8. Complete the following tasks in the Neuron ESB Explorer
    a. Add a new topic called RequestReplyTopic
    b. Add a new publisher called RequestReplyPublisher with a send subscription to RequestReplyTopic
    c. Add a new subscriber called RequestReplySubscriber with a receive subscription to RequestReplyTopic
    d. Save your changes
9. Navigate to Connections -> Service Endpoints
10. Click on the New button in the main panel
    a. Change the name on the General Tab to RequestReplyService
    b. Change the Endpoint Host on the General Tab to Neuron ESB Default Host
11. Click the Client Connector Tab
    a. Check the Enable checkbox to enable the Client Connector
    b. Enter `http://localhost:8080/RequestReply` in the URL textbox
    c. Select RequestReplyPublisher from the Publisher Id drop down list
    d. Select RequestReplyTopic from the Topic drop down list
12. Click Apply
13. Save your changes

With both the Visual Studio Project and the Neuron ESB configuration prepared, the request-reply scenario is ready to be tested.

14. Click on the Tools Menu option for the Neuron ESB Explorer and select Test Client -> 1 Test Client to launch a single test client
15. Select RequestReplySubscriber from the Party Id drop down list and click Connect
16. Click on the Receive Tab of the test client
17. Return to Visual Studio and open the Test Explorer Window (This can be opened from the Test Menu option -> Windows -> Test Explorer
18. Right click RequestReplyTest and select Run Selected Tests
19. Return to the Neuron ESB Test Client and you will see your message in the Receive Tab.
20. Wait 1 minute to allow the service call to timeout
21. Return to Visual Studio to see that the test has failed.
22. Rerun the test in Visual Studio
23. Return to the Neuron ESB Test Client
24. The message count on the Receive tab now says 2 showing that the new message has also been received
25. Click on the Send tab and enter the following message

```
<RequestReplyResponse xmlns="http://tempuri.org/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
        <arg>Nice to meet you</arg>
</RequestReplyResponse>
```

26. Change the semantic drop-down selection to Reply
27. Click the Send button
28. Return to Visual Studio to see that the test has now been passed successfully as the reply has been received

REST Requests

Neuron ESB allows organizations to use REST bindings in addition to SOAP bindings in your configuration. When using the REST binding, the REST protocol (i.e. HTTP/JSON) will be used for the endpoint.

The Neuron ESB configuration will need to be configured in order to support the REST requests test. The following steps will create the proper artifacts.

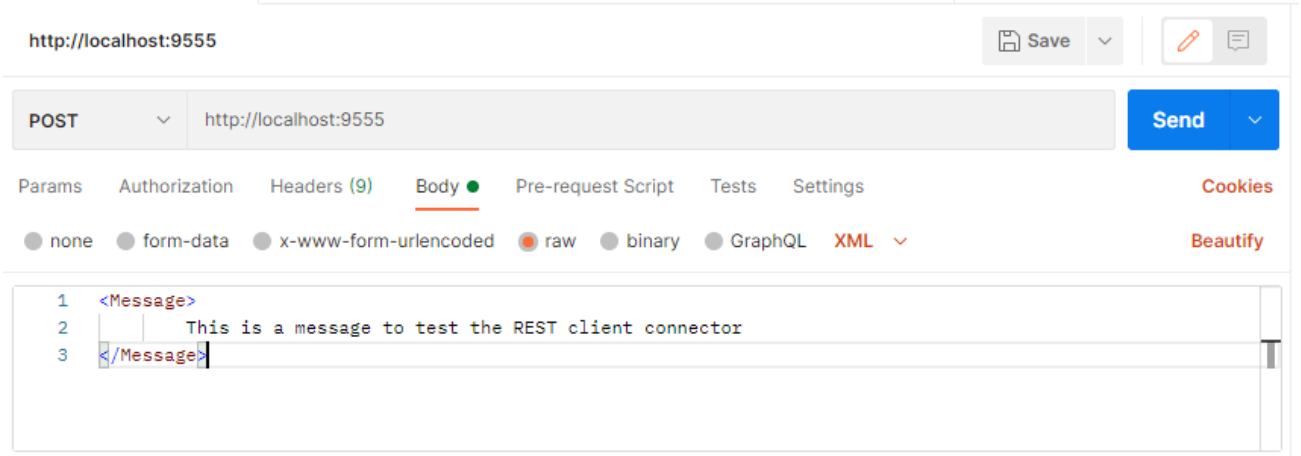
1. Complete the following tasks in the Neuron ESB Explorer
    a. Add a new topic called RESTTopic
    b. Add a new publisher called RESTPublisher with a send subscription to RESTTopic
    c. Add a new subscriber called RESTSubscriber with a receive subscription to RESTTopic
2. Save your changes
3. Navigate to Connections -> Service Endpoints
4. Click on the New button in the main panel
    a. Change the name on the General Tab to RESTService
    b. Change the Binding on the General Tab to REST
    c. Change the Endpoint Host on the General Tab to Neuron ESB Default Host
5. Click the Client Connector Tab
    a. Check the Enable checkbox to enable the Client Connector
    b. Enter `http://localhost:9555` in the URL textbox
    c. Select RESTPublisher from the Publisher Id drop down list
    d. Select RESTTopic from the Topic drop down list
6. Click Apply
7. Save your changes
8. Click on the Tools Menu option for the Neuron ESB Explorer and select Test Client -> 1 Test Client to launch a single test client
9. Select RESTSubscriber from the Party Id drop down list and click Connect
10. Click on the Receive Tab of the test client

With the Neuron ESB configuration prepared, the REST request is ready to be tested. In order to do this, you will need to use a REST client such as Postman, the Firefox REST client or Fiddler.

11. Open a REST client which will send the message
12. Send the following message as a GET request to `http://localhost:9555`

```
<Message>
```

```
        This is a message to test the REST client connector
</Message>
```



13. Return to the Neuron ESB Test Client
14. You will see your message in the Receive tab of the Test Client
15. Click on the Send tab
16. Enter the following message

```
<Response>
      This is a response to the test message
</Response>
```

17. Change the semantic drop-down list to Reply
18. Click Send
19. Return to your REST client to see the reply message has been delivered to the caller.

Using Service Endpoints with Business Processes

Business Processes provide services with a wide variety of functional logic which can be implemented as messages are received by or sent through service endpoints.

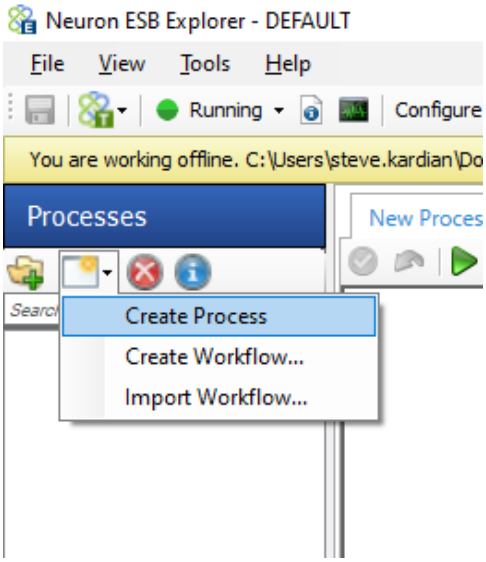
Both Client Connectors and Service Connectors can be used in conjunction with Business Processes, outside of the normal inclusion of Business Process as part of the party hosting the endpoint.

Client Connectors

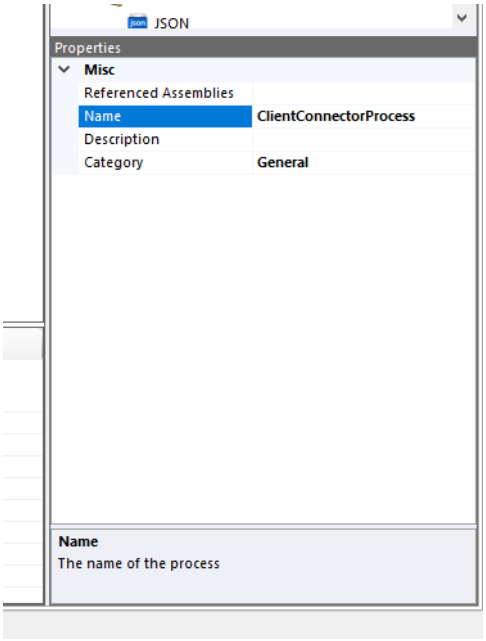
Client connectors can be used outside of the pub/sub model, directly executing one or more Business Processes instead of passing it to a subscriber via a topic. This enables you to encapsulate all the necessary processing of messages into a Business Process without having to worry about the overhead of the pub/sub model, in scenarios where the request message does not need to be passed to a multitude of subscribers.

The following steps will create a simple Business Process which will change the body of the message to a pre-defined string.

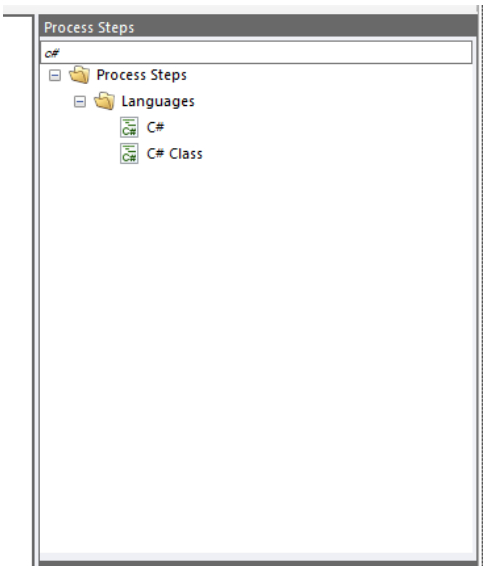
1. In the Neuron ESB Explorer click on Processes
2. Click the New button in the Process Library and select Create Process



3. Click on the New Process element in the Design Canvas
4. In the Properties Grid, on the lower right-hand side of the Neuron ESB Explorer, enter ClientConnetorProcess into the Name textbox

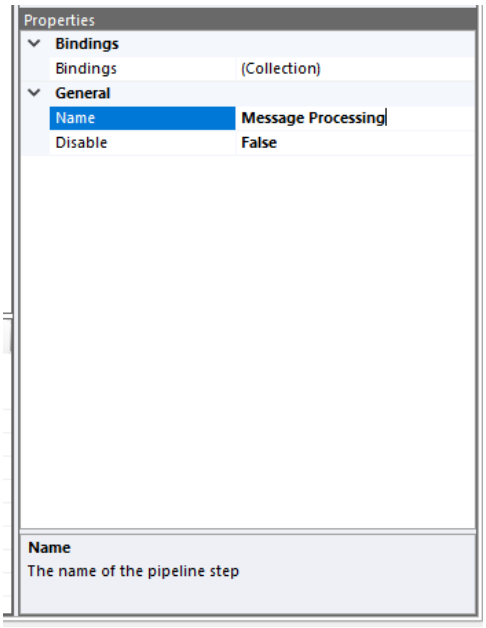


5. In the Search textbox of the Process Step Library enter C#



6. Drag a C# Process Step onto the Design Canvas and drop it in the element named ClientConnectorProcess

7. Click on the C# step in the Design Canvas and use the Property Grid to change its name to Message Processing



8. Double click the Message Processing Process Step in the Design Canvas to open the Language Editor Tab.
9. In the Language Editor tab enter the following code

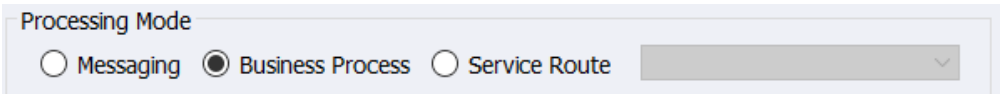
```
context.Data.Text = "This message has been processed";
```

10. Apply your changes and close the Language Editor tab

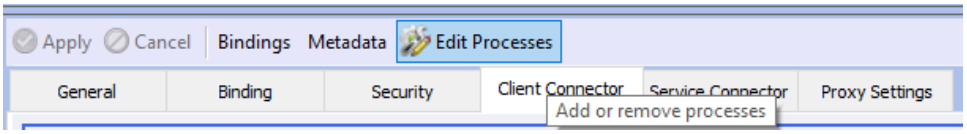
11. Apply your changes to the process

We are now going to use this process directly connected to our REST service endpoint. This will simulate a scenario in which you want a business process to receive a message from a Client Connector, perform the necessary processing of that message, and then return a response to the caller, without having to use the pub/sub model or have another application send a message back.

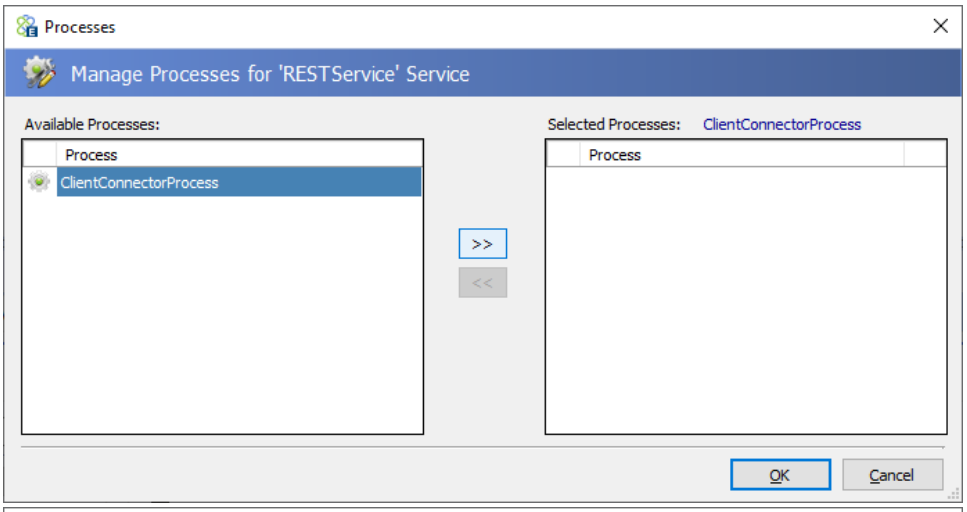
1. Navigate to Connections -> Service Endpoints
2. Click on the RESTService service endpoint
3. Click on the Client Connector Tab
4. Under Processing Mode change the radio button from Messaging to Business Process

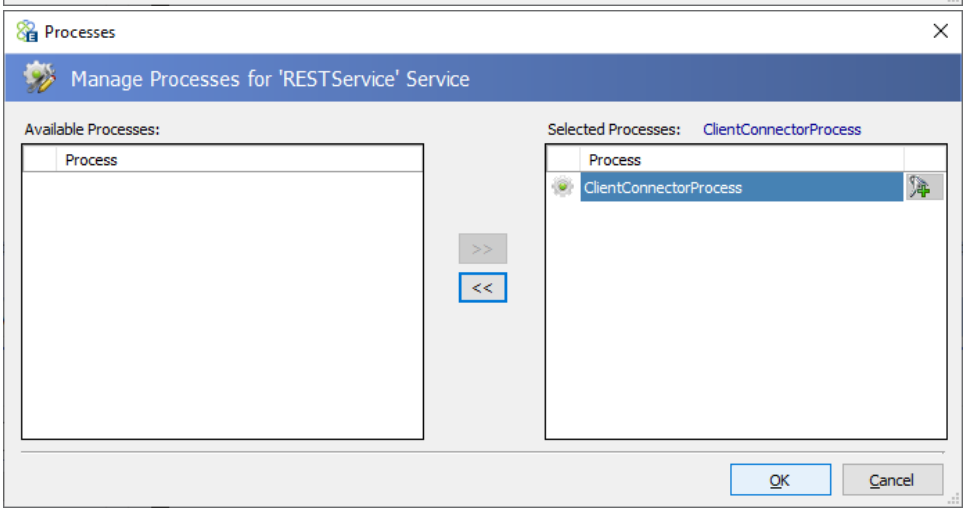


5. Click on Edit Processes, above the Service Endpoint Tabs

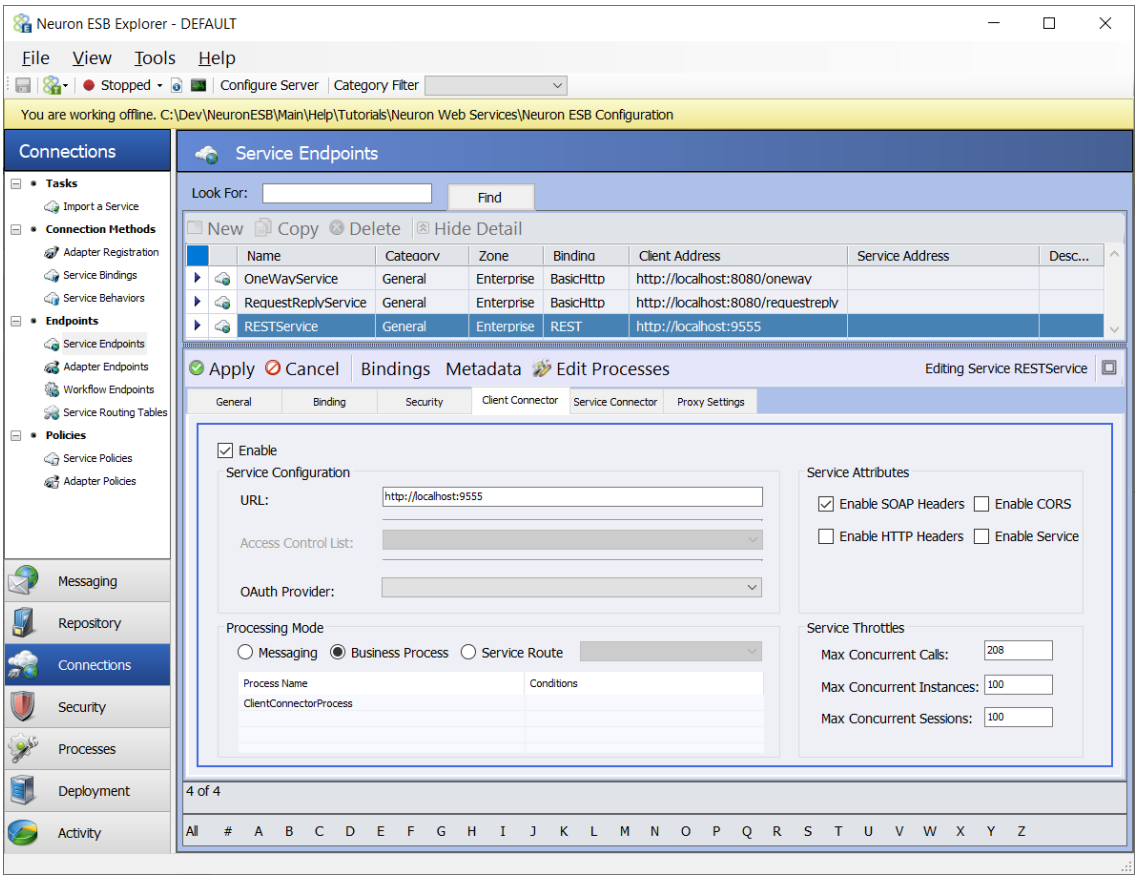


6. Select the ClientConnectorProcess and add it to the list of Selected Processes

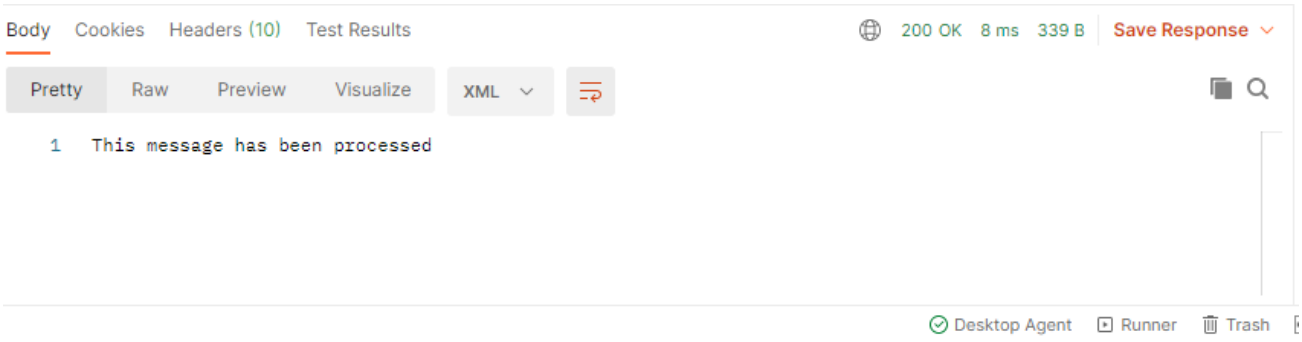




7. Click the OK button to add the process



8. Click Apply
9. Save your changes
10. Return to your REST client and send the message provided in the REST Requests section of this tutorial
11. Look at the response to your REST call and you will see the that Neuron returned the message that you created in the C# step of your process.



Unlike our REST request example, we did not need to launch a test client or send a message back to the caller. The business process handled all of that for us, without the need to participate in the pub/sub model of Neuron ESB.

Service Connectors

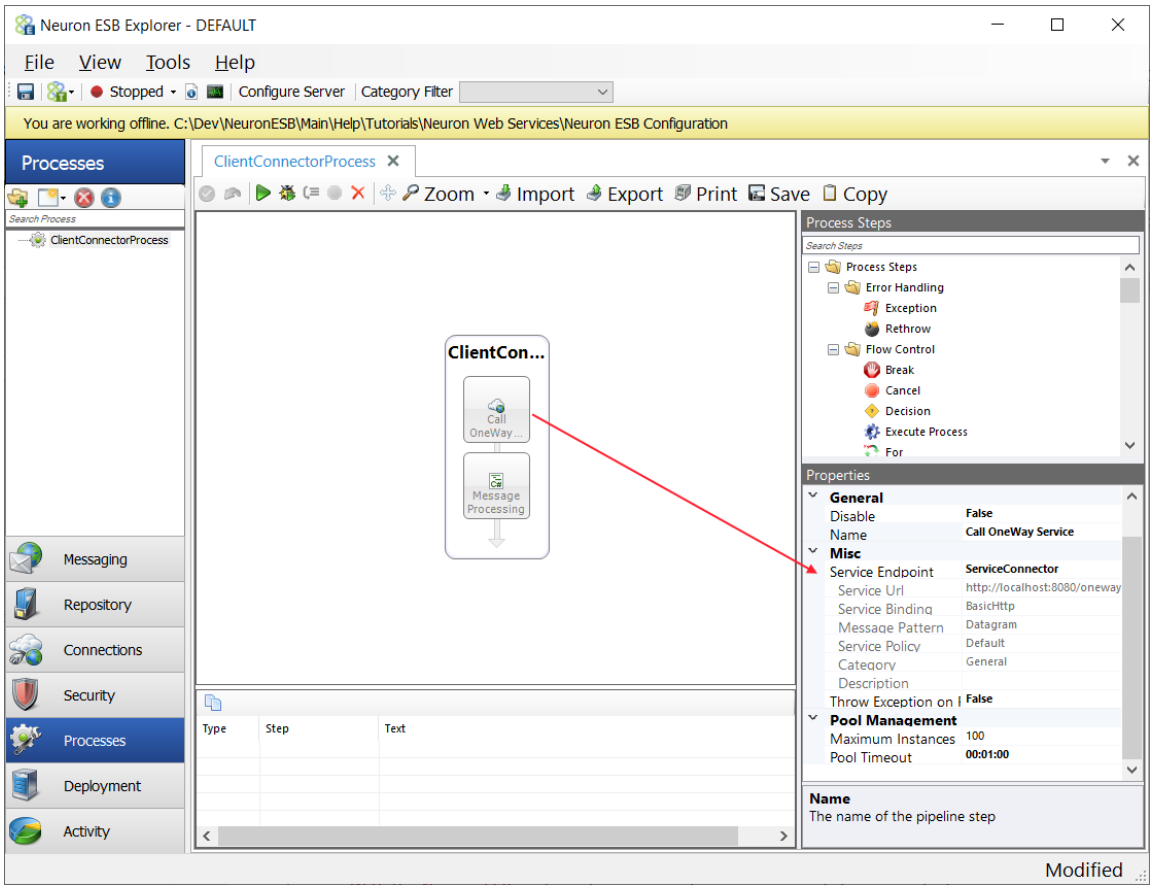
Service Connectors can also be used with Business Processes or Workflows, outside of the pub/sub model. A Service Connector that is used only within a Business Process or Workflow never receives a message from the bus via a Party Subscription, and will not be initialized from the start, like Service Connectors participating in the pub/sub model are. The Service Connector will only be initialized when it is called from a Business Process or Workflow.

The following steps will prepare a Service Connector to be used only outside of the pub/sub model.

1. Navigate to Connections -> Service Endpoints
2. Click on the New button in the main panel
    a. Change the name on the General Tab to ServiceConnector
    b. Leave the Binding on the General Tab as BasicHttp
    c. Change the Endpoint Host on the General Tab to Neuron ESB Default Host
    d. Uncheck the "Enable Service Endpoints" option on the General Tab
3. Click on the Binding tab
    a. Change the Messaging Pattern drop down selection for Request-Reply to Datagram, as Datagram is a synonym for one way.
4. Click the Service Connector Tab
    a. Check the Enable checkbox to enable the Service Connector
    b. Enter `http://localhost:8080/oneway` in the URL textbox
    c. Uncheck the Subscribe box in the Messaging section to remove the service connector from the pub/sub model
5. Apply and Save your changes

A service connector, which is configured to be used only outside the pub/sub model, needs to be integrated into a Business Process or Workflow in order to receive messages. For this scenario we will integrate the Service Connector into the REST Request Business Process and point it to the Client Connector that we created for the one-way scenario.

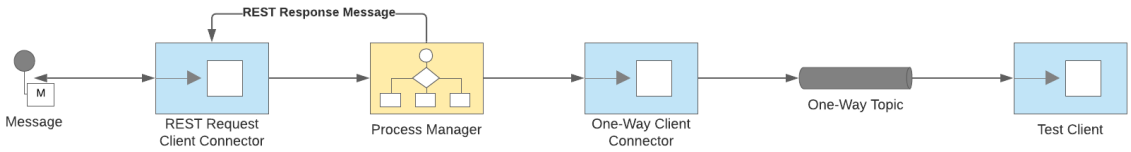
6. Click on Processes
7. Open the ClientConnectorProcess
8. In the Search textbox of the Process Step Library enter Service
9. Drag a Service Endpoint Process Step onto the Design Canvas and drop it onto the element named ClientConnectorProcess above the Process Step named Message Processing
10. Click the Service Endpoint Process Step and in the Property Grid perform the following:
    a. Change the name to Call OneWay Service
    b. Select ServiceConnector from the Service Endpoint drop-down list



11. Apply and Save your changes

With the Neuron ESB configuration prepared we are now ready to run our test.

The expectations for a successful test are that the Client Connector will receive the message from the REST Client and pass that to its Business Process, as was the case in the previous scenario. However, as the Business Process executes it will encounter our new Service Endpoint Process Step and send the message to the Client Connector we previously created for the one-way scenario. That Client Connector will use the pub/sub model to deliver the message to the Neuron ESB Test Client, the same as we saw in the one-way scenario, while the Business Process returns the response to the REST Client.



The following steps will prepare the testing environment.

12. Click on the Tools Menu option for the Neuron ESB Explorer and select Test Client -> 1 Test Client to launch a single test client

13. Select OneWaySubscriber from the Party Id drop down list and click Connect
14. Click on the Receive Tab of the test client
15. Return to your REST client and send the same message that we have used in the previous REST examples.
16. Look at the response from the REST service and you will see that it is the same as in the Client Connector example.
17. Return to the Test Client and you will see the message that was initially sent to the REST service.

This scenario illustrates how messages can be sent to other services, and even await responses from external services before continuing to process a message, without having to go through the pub/sub model.

Web Service Security

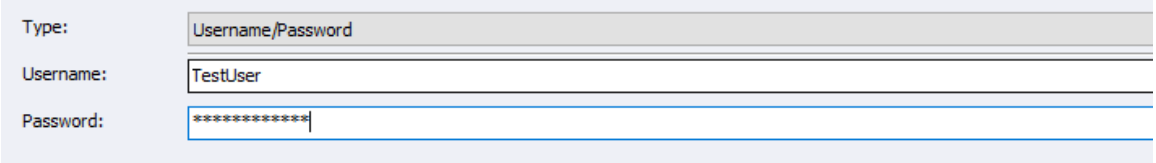
Sometimes it is necessary to use security in conjunction with a web service. Whether that is because you are using an external service that requires it, or because you wish to implement it on the services that Neuron ESB hosts for you.

1. Navigate to Security -> Credentials. This is where you can create and store credentials to be used with your services.
2. Click on the New button in the main panel.

On the General tab is where you will name your credential and determine the type of credential that you are creating. Neuron ESB supports 3 types of credentials

1. Username / Password
2. Windows Domain
3. Certificate

Once you have selected the appropriate type of credential, the property grid for your credential will display fields that need to be filled in for that credential type. For example, the Username / Password credential type shows a textbox for both the username and the password.



Credentials created in this manner can be used with Service Connector service endpoints.

3. Enter Test Credential in the Name textbox
4. Enter TestUser in the Username textbox
5. Enter Password in the Password textbox
6. Apply and Save your changes

For Client Connector service endpoints, you need to take an additional step, which is to create an Access Control List.

Access Control Lists are groups of credentials which can be used with a Client Connector service. When reaching out to an external service you are going to use a specific credential in order to authenticate with that service. However, when hosting your own service, you may need to have more than one credential that can authenticate with that service. For example, if you have a service exposed to 5 different vendors and they each have their own username / password combination in order to authenticate with the service. This is the role that access control lists fill within Neuron ESB. They allow you to specify either a single, or multiple, credentials that the service will accept.

1. Click Access Control Lists
2. Click New in the main panel

Like credentials, Access Control Lists support 3 types of lists

1. Username / Password
2. Windows Domain
3. Certificate

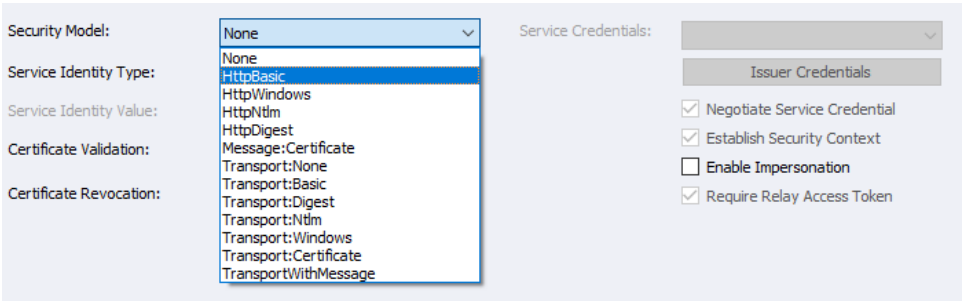
Each Access Control List can only accept a specific credential type, so you cannot have an Access Control List that has both username / password and windows domain credentials.

3. Enter Test ACL in the Name textbox
4. Select Username / Password from the type drop-down list
5. Check the checkbox next to Test Credential
6. Apply and Save your changes

Now that your credentials and Access Control Lists are created, they need to be applied to the appropriate web services.

1. Navigate to Connections -> Service Endpoints
2. Click New on the main panel
3. Click on the Security tab

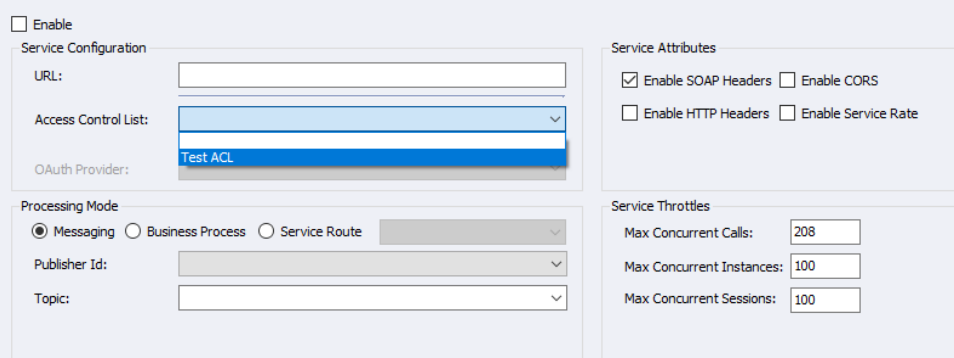
The Security Model drop-down list on the Security tab of a service endpoint will show you all the available security options for the binding type that you have selected. In this case the Binding is BasicHttp so the drop-down list is showing only the security models available for that binding.



More information about WCF bindings and security can be found at https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/bindings-and-security

4. Select HttpBasic
5. Click on the Client Connector tab

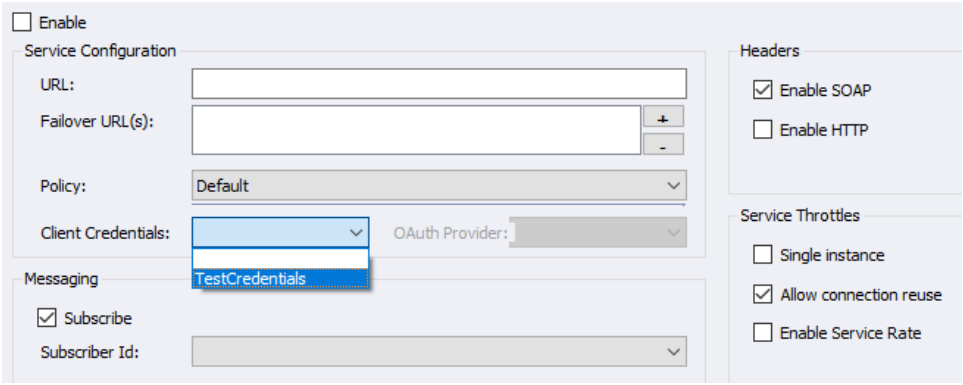
You will notice that the Access Control List drop-down is now enabled and if you click on it you will see the Test ACL access control list that you created listed there. As HttpBasic security supports username / password credentials, any access control list that is of type username / password would appear here. If the security model that you selected only supported windows domain or certificates, then you would not see the Test ACL access control list.



Selecting an Access Control List from the drop-down will automatically apply that Access Control List to your Client Connector once you apply and save your service endpoint.

6. Click on the Service Connector tab

You will notice that the Client Credentials drop-down list is now enabled, and selecting it shows the Test Credentials credential that we created earlier. Just like with Access Control Lists, the Client Credentials drop-down list only shows credentials supported by the security model that you selected.



Selecting a credential from the drop-down list will automatically apply that credential to your service connector once you apply and save your changes.

7. Click cancel as we do not want to save this service endpoint

Using Certificates

Certificates can be used as part of a Client Connector or a Service Connector, as they are credentials they are created in the same way as we did for the username / password credential in the previous example. However, there are a few differences that we need to note in regard to certificates and how to use them properly with Neuron ESB.

Dual Type Security Models

Certain security models support two types of security when being implemented.

Example:

Transport:Basic allows you to specify a username / password credential on the Service Connector tab, or a username / password Access Control List on the Client Connector tab, the same way that HttpBasic does. However, this type of security model also allows you to specify a certificate credential on the security tab, in the Service Credentials drop-down list. This is because Transport:Basic supports the use of one or both types of security on the service.

Certificates and Client Connectors

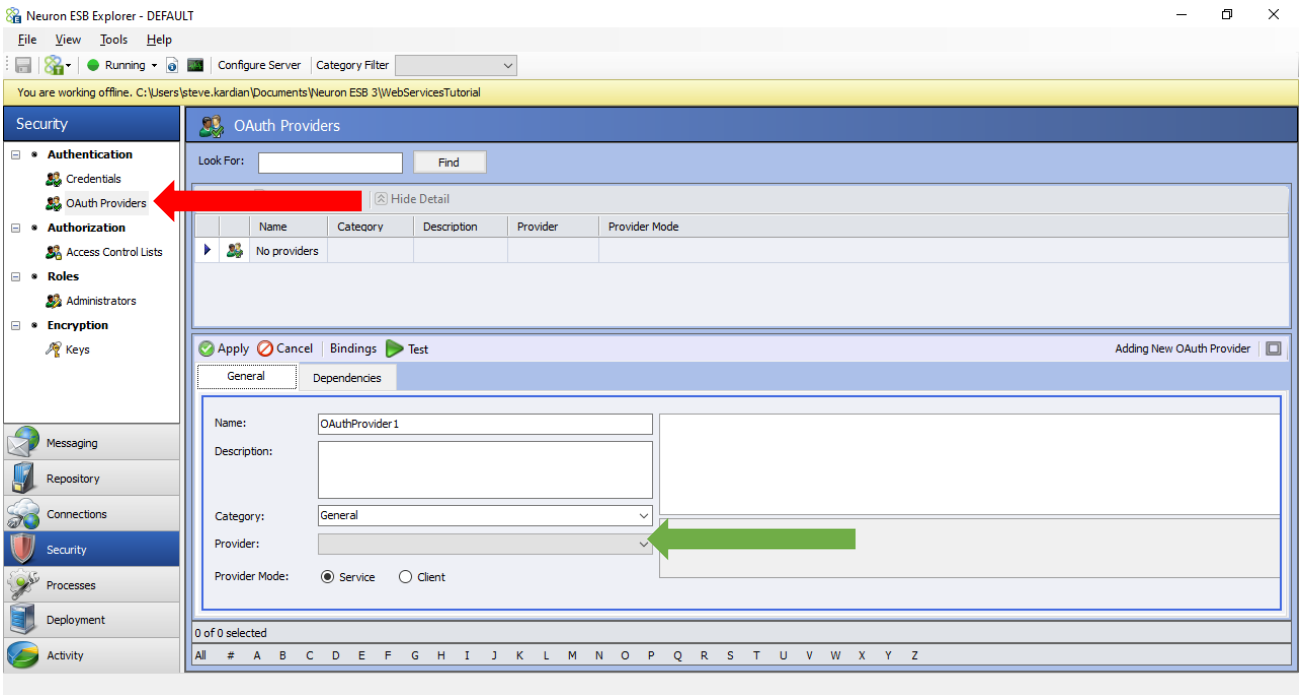
Using a certificate with a Client Connector requires that you first register the certificate on the Client Connectors port. There is an article on the Neuron ESB website that will walk you through how to achieve this.

https://www.neuronesb.com/article/kb/configuring-ssl/

OAuth

Neuron ESB supports OAuth from common providers such as Facebook, Google and LinkedIn, but it also supports custom OAuth providers.

Configuring an OAuth provider for web services is very simple but will require you to coordinate with the authorization service. Most authorization services will require you to register your application with them in order to obtain an identifier and secret value that is used to identify your application and authorize your application to act on behalf of you or your application's users.



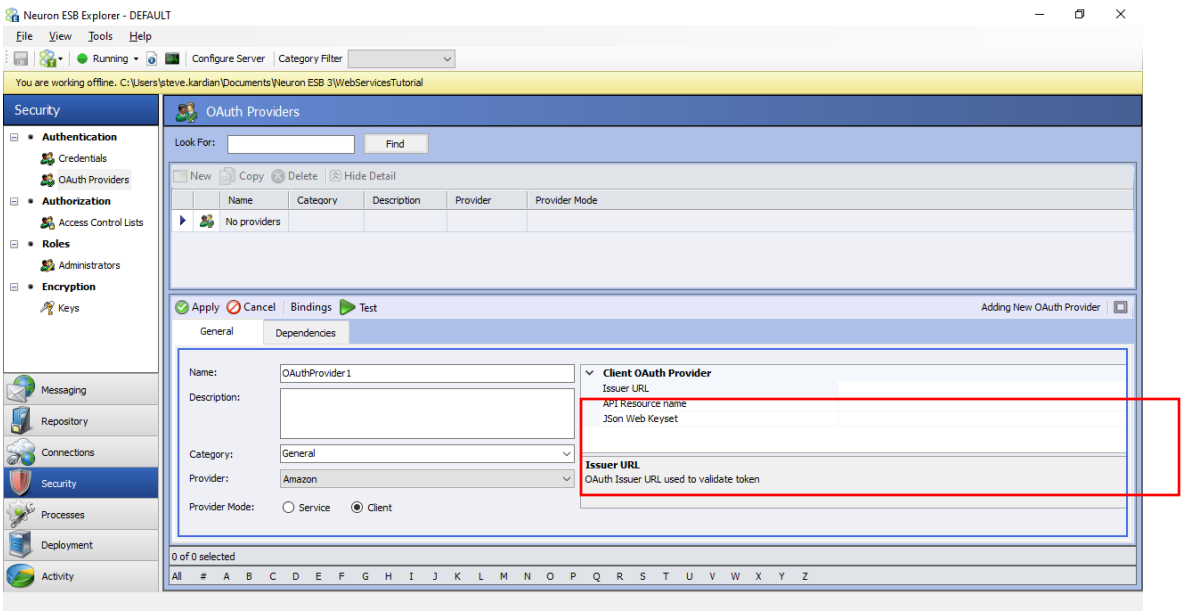
Client Connector OAuth Provider

Navigate to the Security tab in the Neuron ESB Explorer and select the OAuth Providers node in the tree. Create a new OAuth provider and enter in a name. At the bottom of the details view is a radio button which by default has Service selected. Change the selection to Client to create an OAuth provider for a service connector. Above those radio buttons is a dropdown list of providers. Select the authorization service provider that you want to use to obtain an access token for your application to use.

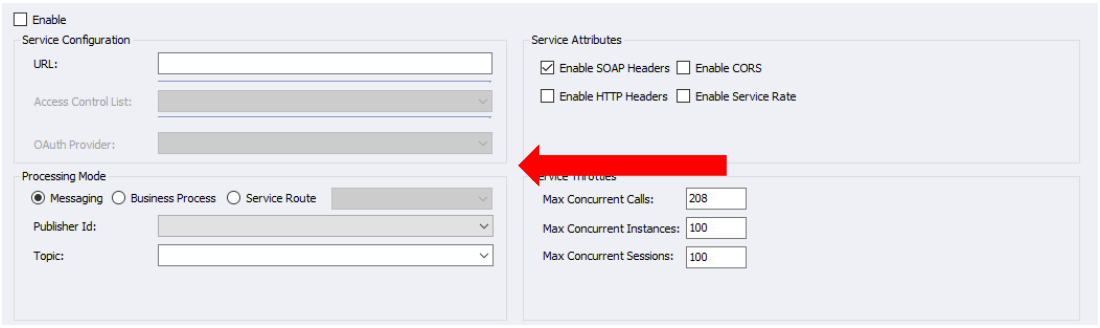
Client side OAuth 2.0 Providers

- Amazon
- Azure Active Directory
- Thinktecture Identity Server
- Peregrine Connect (default)

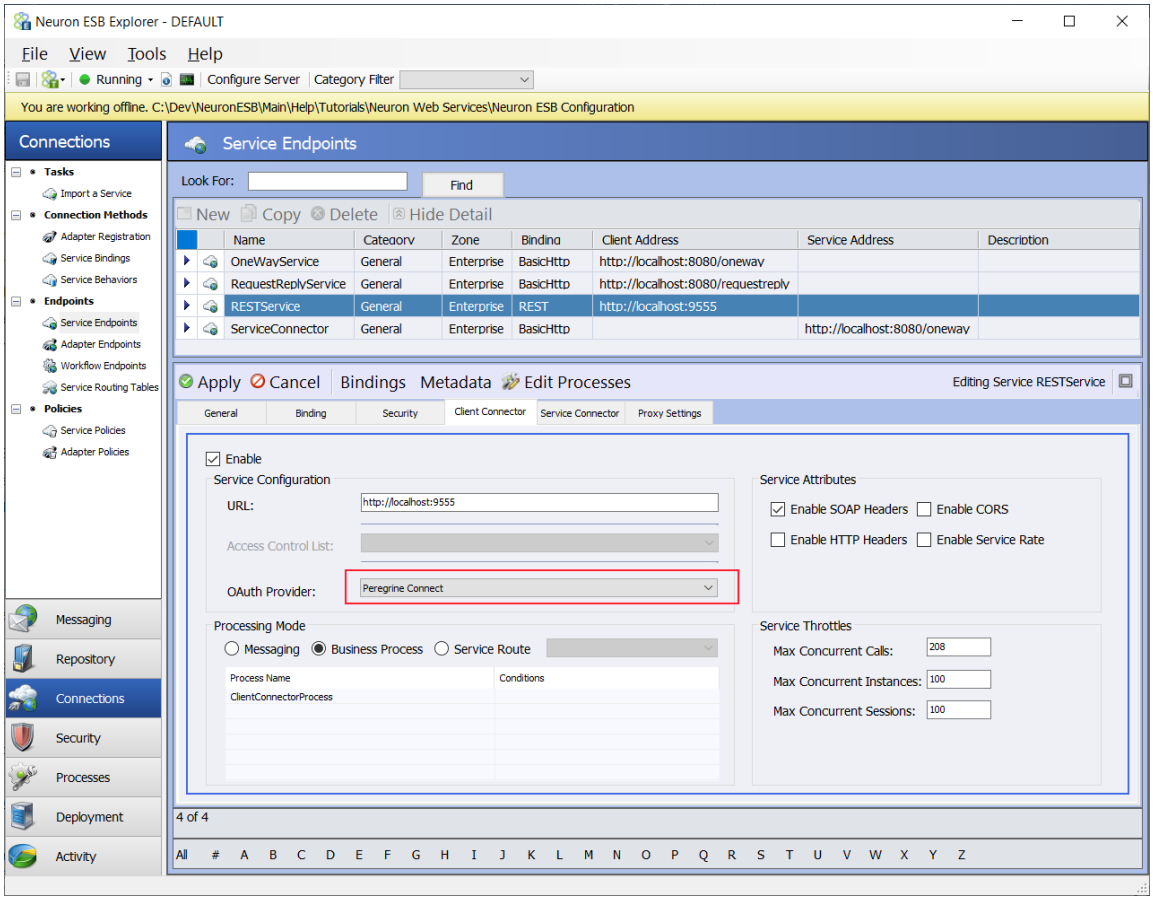
You will next need to enter in the information for the OAuth provider that you selected. For example, selecting Amazon as the provider will require you to enter the Issuer URL, API Resource Name and Json Web Keyset.



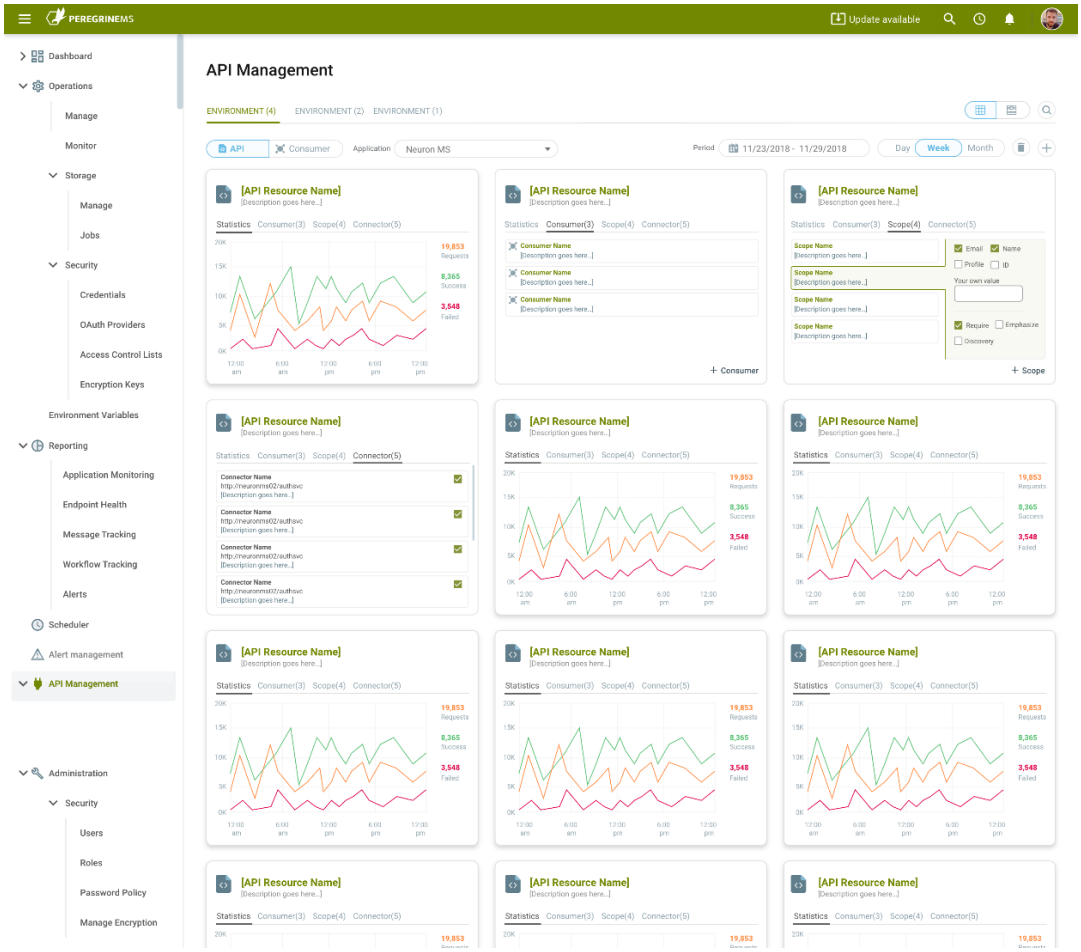
Once an OAuth provider has been configured, it can be used by a RESTful client connector to authorize requests sent to the service. All that is necessary is to associate your OAuth provider with the client connector using the drop-down list located on the client connector detail tab.



Alternatively, you can choose “Peregrine Connect” from the OAuth Provider drop down list as shown below.



In that case, the Client Connector will use the API Management features of the Peregrine Connect Management Suite to provide OAuth capabilities. Organizations can use the Peregrine Connect Management Suite (shown below) to create API Resources (from Neuron ESB Client Connectors), Consumers, issue Tokens, create Scopes and generate standard reporting.



Peregrine Connect Management Suite – API Management is included in the suite and fully integrated with Neuron ESB. Organizations can use the Management Suite to manage Neuron ESB Environments, monitor applications, create and generate alerts, schedule Business Processes and manage their APIs in a fully secure environment.

Service Connector OAuth Provider

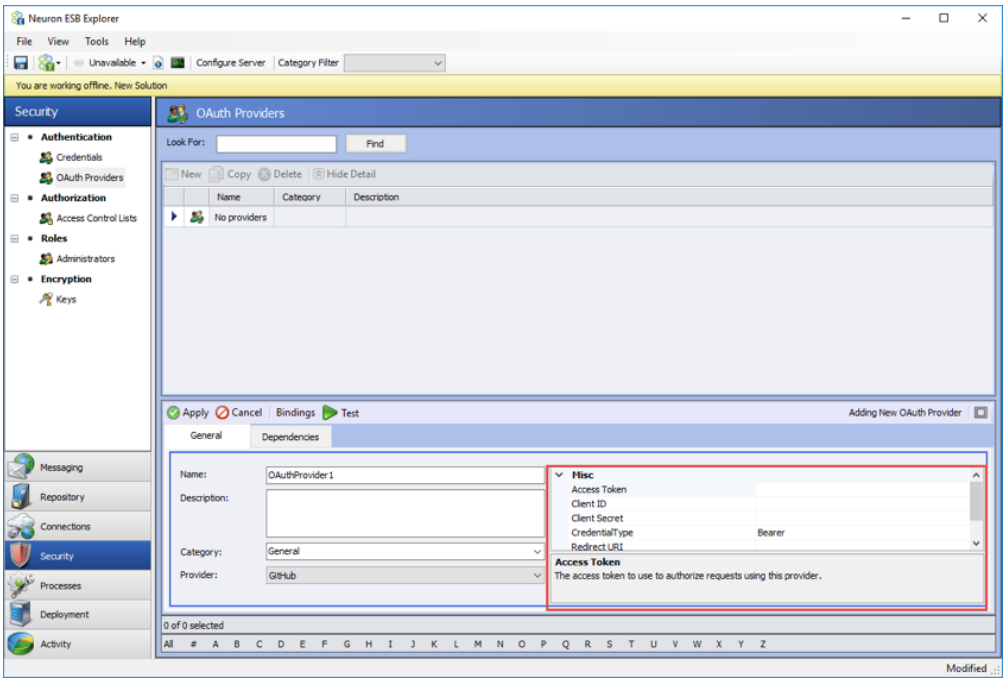
Navigate to the Security tab in the Neuron ESB Explorer and select the OAuth Providers node in the tree. Create a new OAuth provider and enter in a name. At the bottom of the details view is a radio button which by default has Service selected. Leave the default value to create an OAuth provider for a service connector. Above those radio buttons is a dropdown list of providers. Select the authorization service provider that you want to use to obtain an access token for your application to use.

Server side OAuth 2.0 Providers

- ADP
- Amazon
- Azure Active Directory/ADFS
- Box.com
- Dropbox
- Facebook
- Foursquare
- GitHub
- Google
- HubSpot
- Instagram
- LinkedIn
- Salesforce
- ServiceNow
- SoundCloud
- SourceForge
- Thinktecture Identity Server
- Tumblr
- Twitter
- Yahoo!

You will next need to register your application with the authorization service. For example, to use Facebook to obtain an access token to authorize access to remote web services, you will need to create an application at the Facebook Developer website.

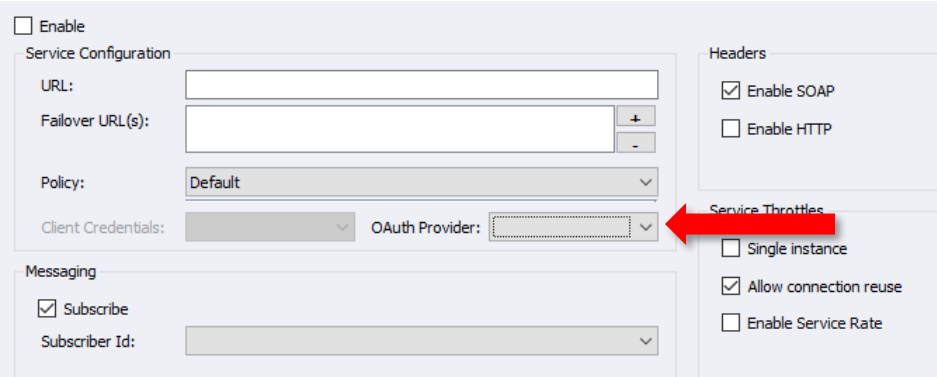
After creating your application, you will be able to access the client identifier and client secret. These values are used to identify your application to the authorization service and to authenticate the access tokens and codes from the authorization service. When you have these values, you should register them in the OAuth provider's properties in Neuron Explorer



You will typically also need to register a callback URI with the authorization service when you register your application. The callback URI is used as part of the OAuth 2.0 protocol to redirect the user back to your application after signing into the authorization service. You can use any URL. During the OAuth authentication process, Neuron ESB's OAuth providers will use the URI to determine when the authorization process has completed successfully in order to obtain the access token from the authorization service for the OAuth provider.

After your OAuth provider is configured, you can test the provider within Neuron ESB Explorer using the web-based flows. By clicking on the Test button in the toolbar for the detail view, a web browser will be opened, and you will be able to validate that the client identifier, secret value, and redirect URI are correct by authenticating with the authorization service and obtaining an access token.

Once an OAuth provider has been configured, it can be used by a RESTful service connector to invoke web services from a remote web service. All that is necessary is to associate your OAuth provider with the service connector using the drop-down list located on the service connector detail tab



Once the service connector has been configured with an OAuth provider, the service connector will use the OAuth provider's access token to invoke the remote web service. The access token will be passed to the remote web service as a bearer token using RFC 6750.

For information on Custom OAuth providers please read the following article:

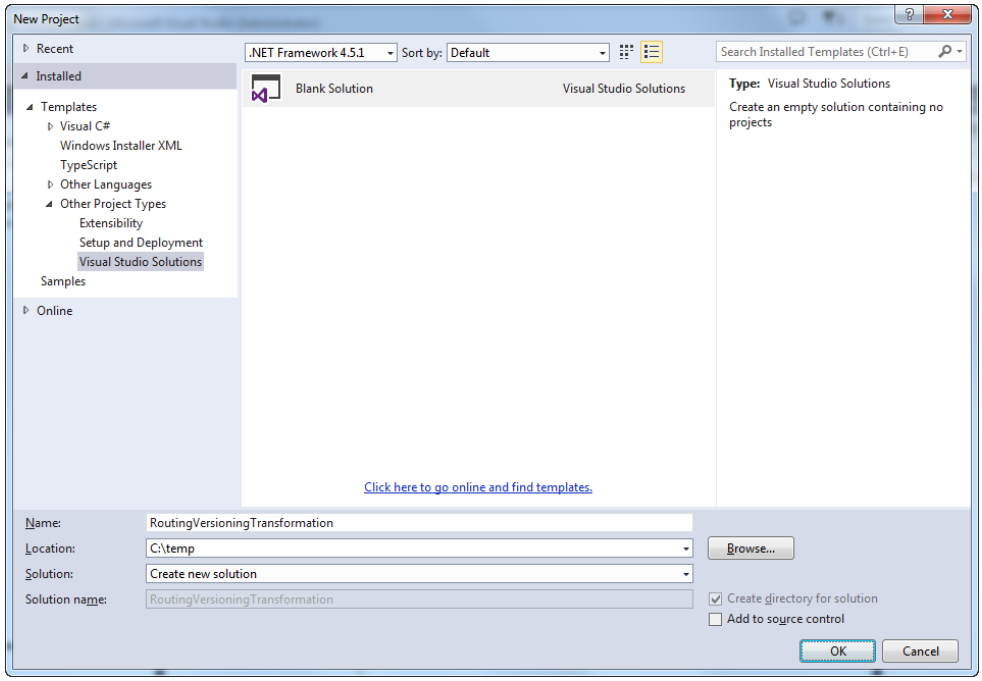
https://www.neuronesb.com/article/custom-oauth-providers/

Routing and Versioning

At the end of this section of the training you will be equipped with an understanding of Neuron ESB as an intermediary that will provide you with the foundation to tackle some of the most common challenges facing organizations today as more and more of their infrastructures are built using web services.

Before we begin though let's take a slight detour and dive into a discussion of versioning. What is versioning? Usually you want one of two things to happen with versioning. Either you want to deploy a version of the service and be able to **route** old clients to the old service and new clients to the new service. Or, you want to be able to deploy a new service at the original address and be able to **transform** packets coming from old clients as they are delivered to the updated service. So versioning then at runtime either involves **routing** or **transformation** or in rare cases both.

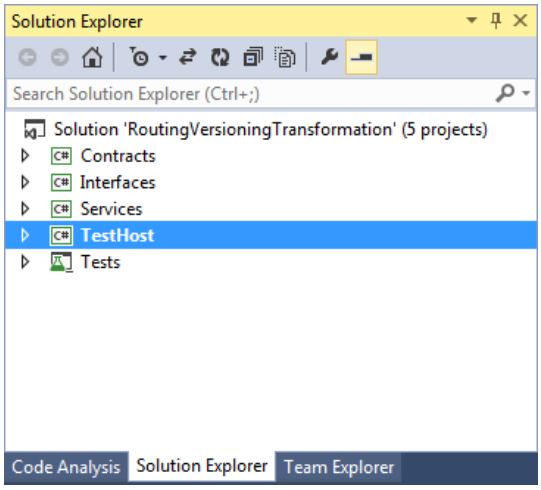
Open Visual Studio and create a new Blank Solution called RoutingVersioningTransformation.



Proceed to do the following in the Blank Solution

1. Create a Visual C# Class Library project called Contracts
2. Create a Visual C# Class Library project called Interfaces
3. Create a Visual C# Class Library project called Services
4. Create a Console Application called TestHost
5. Create a Unit Test Project called Tests
6. In the Interfaces Project add a reference to the Contracts Project
7. In the Services Project add a reference to the Contracts and Interfaces Projects
8. In the TestHost Project add a reference to the Contracts, Interfaces and Services Projects.
9. If it doesn't already exist, add an application configuration file to the TestHost Project
10. Remove the generated Class1.cs files from the Class Library projects.
11. Add references to System.ServiceModel and System.Runtime.Serialization to all of the projects with the exception of Tests. (We will use the Test Host once it is built to generate proxies to use for our test runs.)
12. Set TestHost as the Startup project

Your project structure should look similar to this:



13. Add a C# Class file to the Contracts project. Rename it ExampleMessageContracts.cs. Insert the following code:

```
using System.ServiceModel;
using System.Xml;
using System.Xml.Serialization;

namespace Contracts
{
    [MessageContract(IsWrapped=true)]
    public class ExampleRequest
    {
        [MessageBodyMember(Order=0)]
        [XmlElement(IsNullable=false)]
        public string ParameterOne;

        [MessageBodyMember(Order = 1)]
        [XmlElement(IsNullable = false)]
        public int ParameterTwo;
    }

    [MessageContract(IsWrapped = true)]
    public class ExampleResponse
    {
    }
}
```

14. Add another C# Class file to the project. Rename it ExampleDataContracts.cs.
15. Insert the following code:

```
using System.Runtime.Serialization;

namespace Contracts
{
    [DataContract(Namespace="http://datacontract/example")]
    public class ExampleDcRequest
    {
        [DataMember(IsRequired=true,Order=1)]
        public string Argument1;

        [DataMember(IsRequired=true,Order=2)]
        public int Argument2;
    }
}
```

That's it for the Contracts for now. Later we will return to this project and alter the DataContract above to simulate a typical versioning scenario.

16. Switch to the Interfaces Project and insert a C# Interface file. Rename it IExampleMessageContractInterface.cs. Insert the following code:

```
using System.ServiceModel;

using Contracts;

namespace Interfaces
{
    [ServiceContract(Namespace="http://totalcontrol/services/")]
    public interface IExampleMessageContractInterface
    {
        [OperationContract(Action="http://totalcontrol/example/request",
ReplyAction="http://totalcontrol/example/response")]
        [XmlSerializerFormat(Style = OperationFormatStyle.Document, SupportFaults =
true, Use = OperationFormatUse.Literal)]
        ExampleResponse DoRequest(ExampleRequest request);
    }
}
```

17. Add another C# Interface file and rename it to IExampleDataContractInterface.cs. Insert the following code:

```
using System.ServiceModel;
using Contracts;

namespace Interfaces
{
    [ServiceContract(Namespace="http://datacontract/example")]
    public interface IExampleDataContractInterface
    {
        [OperationContract]
        string DoStuff(ExampleDcRequest request);
    }
}
```

We're done with interfaces.

18. Move onto the Services project. Add a C# Class File and rename it to ExampleMessageContractService.cs.
19. Insert the following code:

```
using System;
using System.ServiceModel;
using Contracts;
using Interfaces;

namespace Services
{
    [ServiceBehavior(Name="MessageContractService",Namespace="http://totalcontrol")]
    public class ExampleMessageContractService : IExampleMessageContractInterface
    {
        public ExampleResponse DoRequest(ExampleRequest request)
        {
            Console.WriteLine("Received request with ParameterOne={0} and
ParameterTwo={1}", request.ParameterOne, request.ParameterTwo);
            return new ExampleResponse();
        }
    }
}
```

20. Add another C# Class File and rename it to ExampleDataContractService.cs. Insert the following code:

```
using System;
using Contracts;
using Interfaces;

namespace Services
{
    public class ExampleDataContractService :IExampleDataContractInterface
    {
        public string DoStuff(ExampleDcRequest request)
        {
            Console.WriteLine("Received request with {0}, {1} as arguments",
request.Argument1, request.Argument2);
            return "Ok. I did stuff at " + DateTime.Now;
        }
    }
}
```

We're done for now with our services implementation. We will return when we simulate our versioning scenario.

21. Open the Program.cs file in the TestHost project and insert the following code:

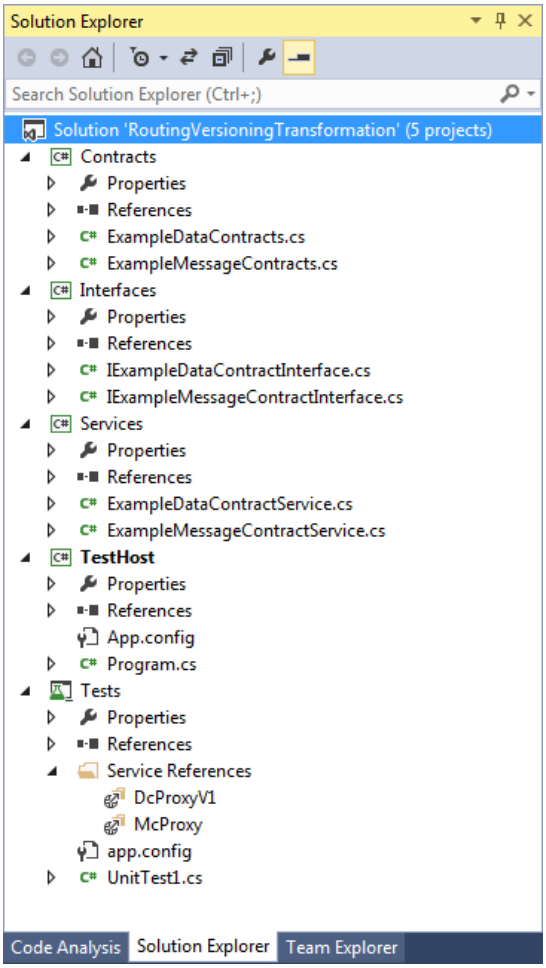
```
using System;
using System.ServiceModel;
using Services;

namespace TestHost
{
    class Program
    {
        static void Main(string[] args)
        {
            var mchost = new ServiceHost(typeof(ExampleMessageContractService));
            var dchost = new ServiceHost(typeof(ExampleDataContractService));
            mchost.Open();
            dchost.Open();
            Console.ReadLine();
            mchost.Close();
            dchost.Close();
        }
    }
}
```

22. Open the App.config file for the TestHost project and insert the following between the configuration elements:

```
<system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="TestHostBehavior">
          <serviceDebug includeExceptionDetailInFaults="true" />
          <serviceMetadata httpGetEnabled="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service behaviorConfiguration="TestHostBehavior"
name="Services.ExampleMessageContractService">
        <endpoint address="" binding="basicHttpBinding"
            bindingConfiguration="" name="MessageContractService"
bindingNamespace="http://totalcontrol"
            contract="Interfaces.IExampleMessageContractInterface" />
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost/messagecontract/" />
          </baseAddresses>
        </host>
      </service>
      <service behaviorConfiguration="TestHostBehavior"
name="Services.ExampleDataContractService">
        <endpoint address="" binding="basicHttpBinding"
            bindingConfiguration="" name="DataContractService"
bindingNamespace="http://generated"
            contract="Interfaces.IExampleDataContractInterface" />
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost/datacontract/" />
          </baseAddresses>
        </host>
      </service>
    </services>
</system.serviceModel>
```

23. Press F6 to compile the solution, then press Ctrl-F5 to launch the TestHost. Add a Service Reference to the Tests Project using the URL http://localhost/messagecontract/ and use the namespace McProxy. Add another Service Reference to http://localhost/datacontract/ and use the namespace DcProxyV1.
24. Shut down the Running TestHost. Your Solution artifacts should now look like the following in Solution Explorer:



25. Open the UnitTest1.cs file in the Tests project and insert the following using statements at the top:

```
using Tests.McProxy;
using Tests.DcProxyV1;
```

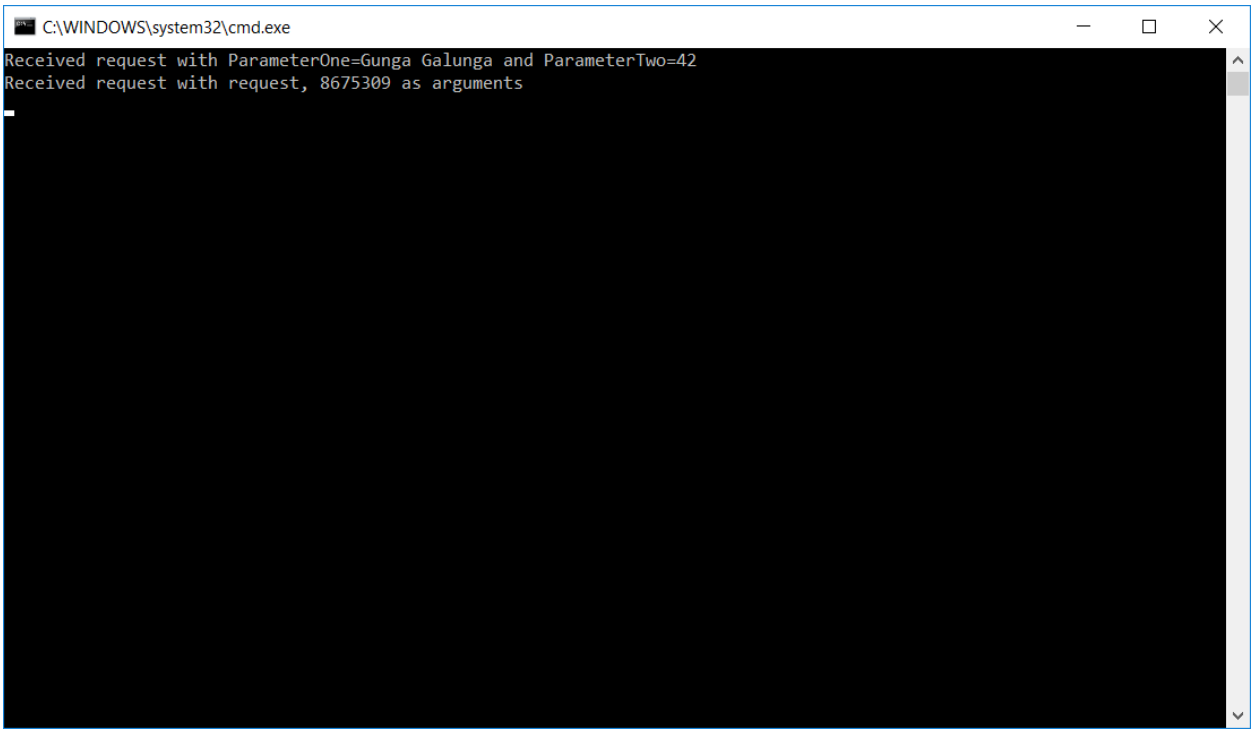
26. Modify TestMethod1 so it looks as depicted below and add a method named TestMethod2 that contains the code depicted below.

```
[TestMethod]
public void TestMethod1()
{
    new ExampleMessageContractInterfaceClient().DoRequest("Gunga Galunga", 42);
}

[TestMethod]
```

```
public void TestMethod2()
{
      new ExampleDataContractInterfaceClient().DoStuff(new ExampleDcRequest {
Argument1 = "request", Argument2 = 8675309 });
}
```

27. Press F6 to compile and then press Ctrl-F5 and switch to Test Explorer. Use Test Explorer to run the test methods and verify functionality when calling the services directly. The following should appear in the console application:



28. Leave the TestHost running
29. Return to the Neuron Explorer and perform the following
30. Add a Topic called OnRamp
31. Add a Topic called Services and two Subtopics MessageContract and DataContract
32. Add a Publisher called OnRampPublisher with a Send subscription to OnRamp and “Services.*”
33. Add a Subscriber called MessageContractSubscriber with a Receive subscription to Services.MessageContract
34. Add a Party called DataContractSubscriber with a Receive subscription to Services.DataContract
35. A Client Connector with the following settings:
    a. Name: OnRampConnector
    b. Binding: BasicHttp
    c. Endpoint Host: Neuron ESB Default Host
    d. Enable Client Connector
    e. Publisher Id: OnRampPublisher

f. Topic: OnRamp
g. URL: http://localhost/onramp

36. A Service Connector with the following settings (you may import it by using the minimized running TestHost or configure the settings manually):
    a. Name: MessageContractService
    b. Binding: BasicHttp
    c. Endpoint Host: Neuron ESB Default Host
    d. Enable Service Connector
    e. Subscriber Id: MessageContractSubscriber
    f. URL: http://localhost/messagecontract/
37. A Service Connector with the following settings (you may import it by using the minimized running TestHost or configure the settings manually):
    a. Name: DataContractService
    b. Binding: BasicHttp
    c. Endpoint Host: Neuron ESB Default Host
    d. Enable Service Connector
    e. Subscriber Id: DataContractSubscriber
    f. URL: http://localhost/datacontract/
38. Switch to the Processes tab,
39. Click on the New button and select Create Process
40. Click the Untitled Process in the designer and change the Name in the Property Grid to ChangeTopic
41. In the Search textbox of the Process Step Library enter C#
42. Drag a C# step onto the Design Canvas and drop it in the element named ChangeTopic
43. Right click the C# process step and choose Edit Code to open the Language Editor
44. In the Language Editor enter the following code

```
if(context.Data.Header.Action.StartsWith("http://totalcontrol/"))
{
    context.Data.Header.Topic = "Services.MessageContract";
}
else if(context.Data.Header.Action.StartsWith("http://datacontract"))
{
    context.Data.Header.Topic = "Services.DataContract";
}
```

45. Apply your changes in the Language Editor
46. Navigate to Messaging -> Publishers
47. Select the OnRampPublisher
48. Select the Processes tab
49. Click on Edit Processes
50. Add the ChangeTopic Process to the On Publish event
51. Click Ok
52. Apply and Save your Changes

The ChangeTopic Business Process will run everytime a request is made to the onramp "http://localhost/onramp" URL configured in the Client Connector, "OnRampConnector". When the

message is received, its SOAP Action is evaluated and, depending on the value, the internal Topic to publish the request on will be changed to the Topic that the target Service Endpoint is subscribing to. Essentially, this is using Topic based pub/sub messaging to decouple and dynamically route inbound requests to the correct outbound URLs, represented by subscribing Service Connectors.

53. Restore the TestHost if it is still running and press enter to shut the service down
54. Modify the address entries in the Tests project App.config to point to the onramp

```
<client>
    <endpoint address="http://localhost/onramp" binding="basicHttpBinding"
        bindingConfiguration="MessageContractService"
        contract="McProxy.IExampleMessageContractInterface"
        name="MessageContractService" />
    <endpoint address="http://localhost/onramp" binding="basicHttpBinding"
        bindingConfiguration="DataContractService"
        contract="DcProxyV1.IExampleDataContractInterface"
        name="DataContractService" />
</client>
```

55. Press Ctrl –F5 and then minimize the TestHost. Switch to Test View and run the Test Methods then restore the TestHost window

At this point you know the fundamentals of how to build a powerful and flexible routing mechanism that relieves clients of having to deal with multiple service addresses and multiple bindings.

Let's now add versioning into the mix. Suppose you were given this mandate.

> *Modify an existing service by adding a required element to support new functionality to an existing service without affecting current client code.*

With Neuron there are several approaches to solving this problem. We will work through one way and after you have completed that exercise you will be well on your way to being capable of developing your own patterns.

1. Begin by shutting down the TestHost.
2. Modify the existing code in ExampleDataContracts.cs located in the Contracts project by adding the highlighted code below:

```
[DataContract(Namespace="http://datacontract/example")]
    public class ExampleDcRequest
    {
    [DataMember(IsRequired=true,Order=1)]
    public string Argument1;

    [DataMember(IsRequired=true,Order=2)]
    public int Argument2;

    [DataMember(IsRequired = true, Order = 3)]
    public double Argument3;
}
```

3. Modify the code in ExampleDataContractService.cs located in the Services project so that it looks like the following:

```
public string DoStuff(ExampleDcRequest request)
```

```
{
    Console.WriteLine("Received request with {0}, {1}, {2} as arguments",
    request.Argument1, request.Argument2,request.Argument3);
    return "Ok. I did stuff at " + DateTime.Now;
}
```

4. Press F6 to compile and then press Ctrl-F5 and add a Service Reference to the Tests Project to http://localhost/datacontract/. This time use the namespace **DcProxyV2**.
5. Shutdown the TestHost and add the method below to the UnitTest1.cs file in the Tests project:

```
[TestMethod]
public void TestMethod3()
{
    new  Tests.DcProxyV2.ExampleDataContractInterfaceClient().DoStuff(
    new Tests.DcProxyV2.ExampleDcRequest { Argument1 = "request", Argument2 = 8675309,
Argument3 = 3.14159 });
}
```

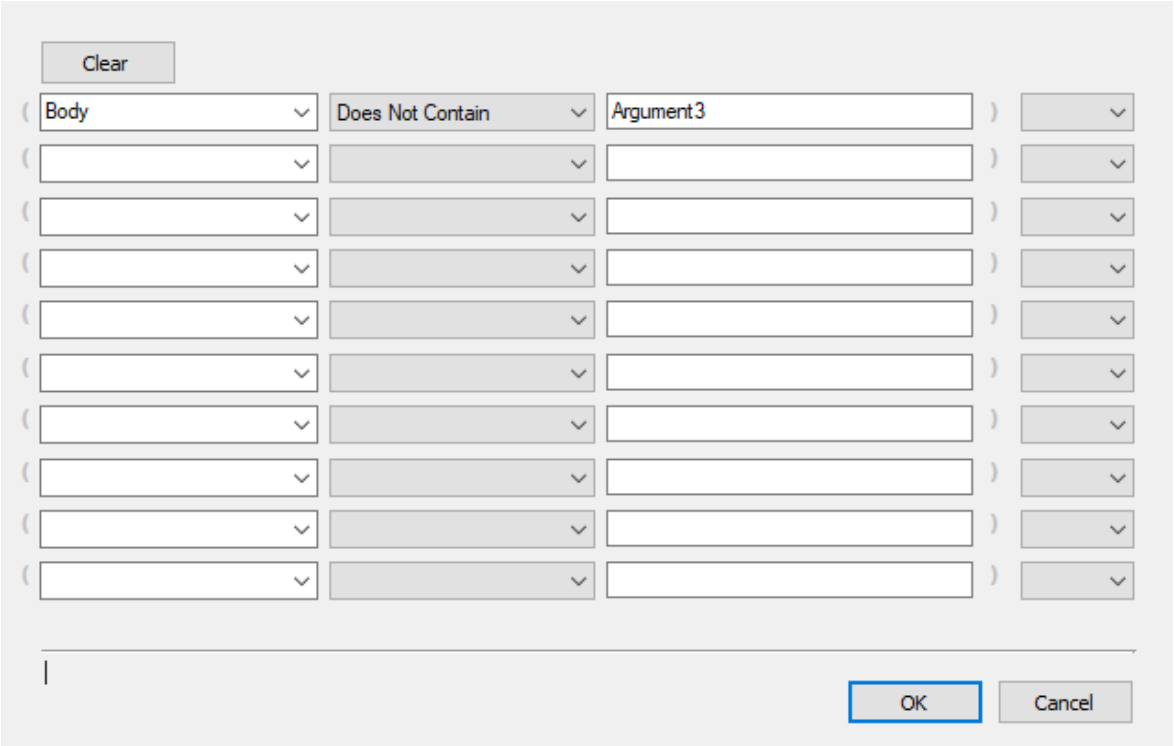
6. Change the newly generated address in the Tests App.config to the on ramp to use the localhost address as follows:

```
<endpoint address="http://localhost/onramp/" binding="basicHttpBinding"
    bindingConfiguration="MessageContractService"
    contract="McProxy.IExampleMessageContractInterface"
    name="MessageContractService" />
<endpoint address="http://localhost/onramp/" binding="basicHttpBinding"
    bindingConfiguration="DataContractService"
    contract="DcProxyV1.IExampleDataContractInterface"
    name="DataContractService" />
<endpoint address="http://localhost/onramp/" binding="basicHttpBinding"
    bindingConfiguration="DataContractService1"
    contract="DcProxyV2.IExampleDataContractInterface"
    name="DataContractService1" />
```

7. Return to Neuron Explorer and click the Processes tab.
8. Create a new Process.
9. Rename the Process InsertNode and drag a Transform-Xslt Process Step onto the Design Canvas
10. Click the Transform-xslt Step.
11. In the Property Grid click the ellipses on the TransformXml property and enter this transform

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/*">
    <DoStuff xmlns="http://datacontract/example">
      <request xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <Argument1><xsl:value-of select="//*[local-name()='Argument1']" /></Argument1>
        <Argument2><xsl:value-of select="//*[local-name()='Argument2']" /></Argument2>
        <Argument3>-1.00</Argument3>
      </request>
    </DoStuff>
</xsl:template>
</xsl:stylesheet>
```

12. Click Ok
13. Apply and Save your changes
14. Navigate to the Messaging tab and then the DataContractSubscriber Party.
15. Select the Processes tab and click the Edit Processes button.
16. Add the InsertNode Process to the On Receive event
17. Click the  button next to the Event dropdown and enter the following



The Conditions dialog allows users to create restrictions on when/if a Business Process is executed for an incoming message.

18. Click Ok twice and then Apply your changes and Save the configuration
19. Return to Visual Studio. If the TestHost is already running stop it and then press F6 and then Ctrl-F5 to restart
20. Switch to Test View and run the tests. Restore the TestHost and observe the output

Notice that the process was invoked only when the old client packet was detected.

Service Routing Tables

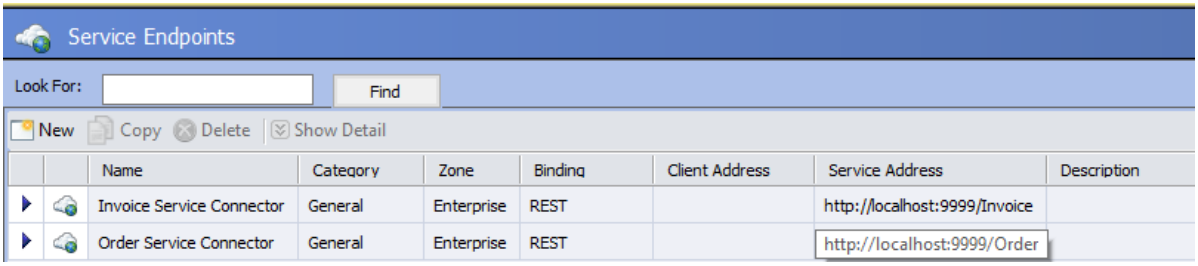
This section will show you how to setup a service routing table and associate it with a client connector in order to route messages based on the message body. Service Routing tables can be used to build (through configuration) simple pass thru or dynamic routing solutions.

The following steps will prepare the service routing table in the Neuron ESB configuration.

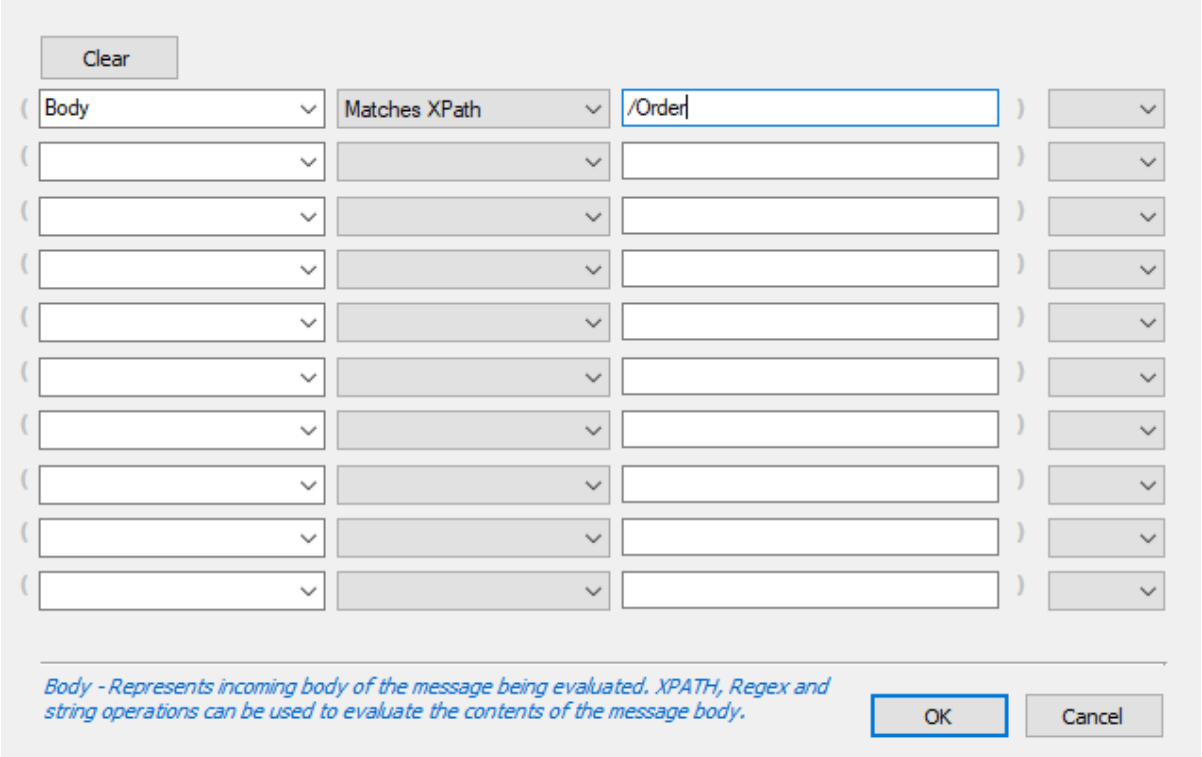
1. Navigate to Connections -> Service Endpoints
2. Click the “New” button to create a new service connector.
3. On the General tab
    a. Enter “Order Service Connector” in the name field.
    b. Change the Binding dropdown to REST
4. On the Service Connector tab

a. Enable the Service Connector by checking the enabled checkbox
b. Enter “http:localhost:9999/Order” in the URL field

5. Click the Apply button
6. Repeat this process to create an “Invoice Service Connector” with a URL of “http://localhost:9999/Invoice”



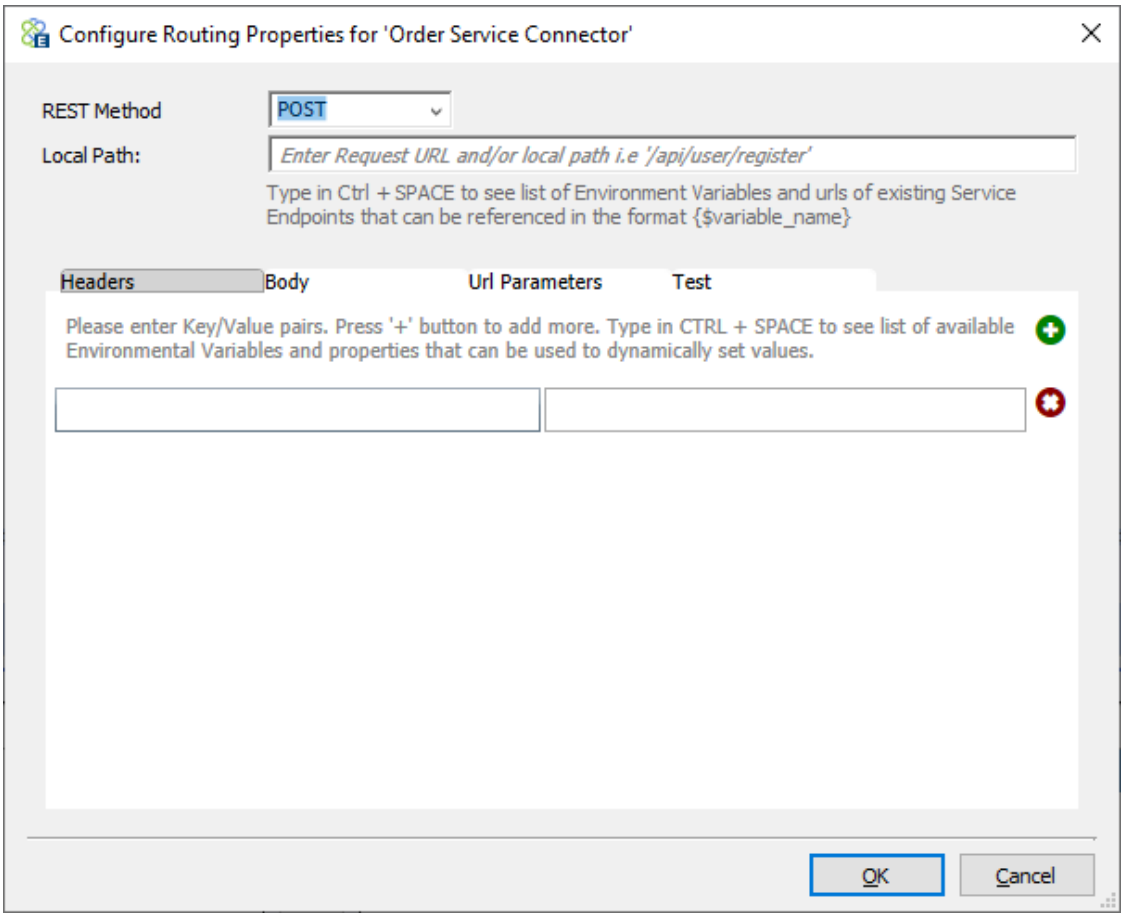
7. Navigate to Connections -> Service Routing Tables
8. Move the Order Service Connector to Select Endpoints by clicking the >> button
9. With the Order Service Connector in the Selected Endpoints box click on the  button to launch the Edit Conditions dialog
10. In the Edit Conditions dialog
    a. Select Body from the first dropdown list
    b. Select Matches XPath in the second dropdown list
    c. Enter /Order in the third dropdown list.



This will create a condition for the message that should the body include the XPath /Order it will be routed to the service connector Order Service Connector.

11. Highlight the Order Service Connector in the Selected Endpoints pane then click Configure Endpoint (right clicking the endpoint will also display the short cut menu)

12. Change the REST Method to POST



13. Repeat this process for the Invoice Service Connector with a condition of body matches XPath "/Invoice"
14. Click Apply to save the Service Routing Table.
15. Navigate to Connections -> Service Endpoints
16. Click new to create a Client Connector
17. On the General tab
    a. Enter REST Client Connector in the name field
    b. Select REST from the binding dropdown list
18. On the Client Connector tab

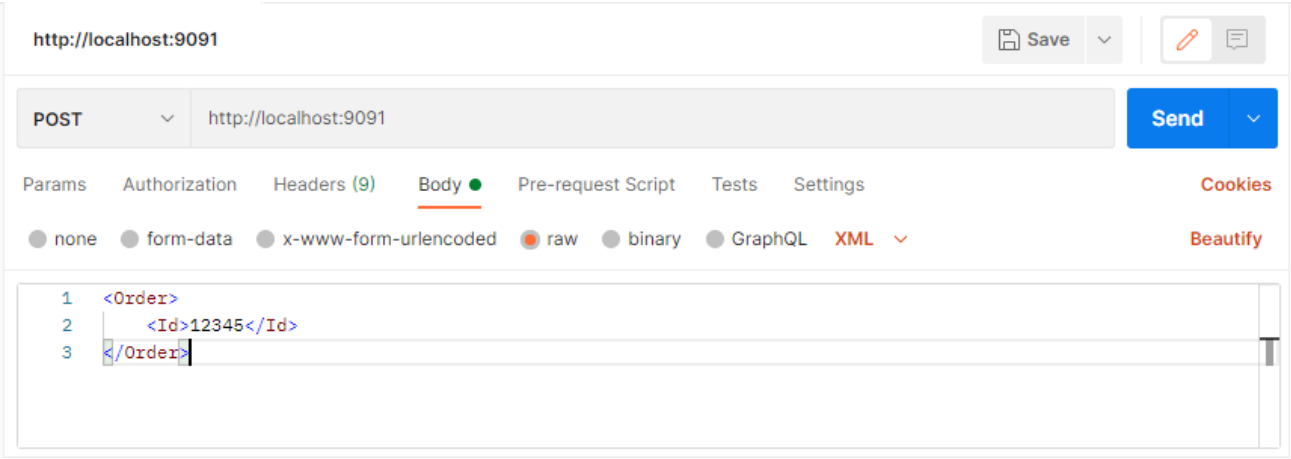
a. Enable the client connector by checking the enable checkbox
b. Enter “http://localhost:9091” in the URL field
c. In Processing Mode select the Service Route radio button and then select ServiceRoutingTable1 from the dropdown list.

19. Apply your changes
20. Save the configuration

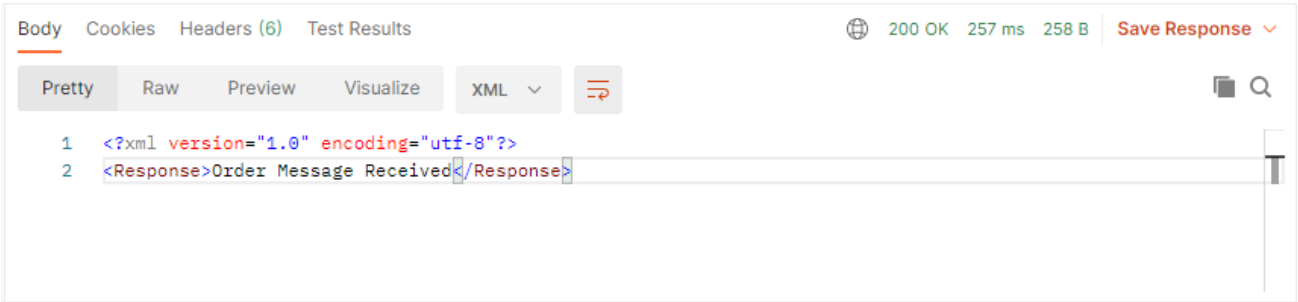
With the setup of the Neuron ESB configuration completed, the testing environment needs to be prepared. For this scenario an external service is required. This service is provided in the Service Routing Table Project folder.

21. Navigate to Service Routing Table Project\SRT_Example\bin\Debug and double click SRT_Example.Service.exe to launch the WCF service to receive the routed message
22. Launch a REST client such as Postman or the Firefox REST Client
23. Select POST as the REST method and enter “http://localhost:9091” in the URL field
24. In the Body pane enter the following XML

```
<Order>
        <Id>12345</Id>
</Order>
```



25. Send the message
26. Notice that the response from the service indicates that an Order message was received.



27. Repeat the process with the following XML as the body

```
<Invoice>
        <Id>12345</Id>
</Invoice>
```

28. Notice that this time the response from the service indicates that an Invoice message was received.

While this is a very simplified example, it illustrates how service routing tables can be used to identify and route messages to the correct service endpoint, making them a powerful tool in an organization's overall integration strategy.

With this knowledge about the fundamentals of Neuron ESB Business Processes and conditions and some practice you should be prepared for pretty much any routing, transformation or versioning challenge.

Review

In today's enterprise architecture the use of services, and the need to connect to them, is more important than ever. Neuron ESB's service endpoints not only allows organizations to integrate external services into their overall enterprise architecture, but also provide the ability to host services directly from the Neuron platform to facilitate the receipt of messages and data.

Client Connectors can be used to quickly and easily spin up services, hosted by Neuron ESB, which receive data directly at the middle ware level.

Service Connectors allow organizations to easily integrate external services into their integration strategies.

Service endpoints can leverage the power of Neuron ESB Business Processes, which provide logical functionality to the messages being processed, both when part of the pub/sub model or when being used outside of it.

Neuron ESB provides service endpoints with the ability to implement WCF security models. This allows organizations to integrate with external services which require authentication and enables client connectors to enforce security protocols.

Credentials used by service endpoints, when implementing security, can be either Windows Domain, Username/Password or Certificates.

Service Connectors and Client Connectors can leverage OAuth in order to communicate with external services or receive requests from external clients.

Business Processes and Service Routing Tables can be used to implement static and dynamic routing inside of Neuron ESB. While Business Processes provide logical functionality in addition to their Service Routing abilities, Service Routing Tables are faster and have a lower latency.

Quiz

1. How many types of service endpoints are available in Neuron ESB?
2. What are the different types of service endpoints called?
3. Neuron ESB produces WSDL and defaults to strongly typed messaging?
    a. True
    b. False
4. What two features provide service routing capabilities?
5. A service endpoint step allows you to call a configured service connector directly from with within a business process, bypassing pub/sub model entirely?
    a. True
    b. False
6. Upon what framework are the service security models based?
7. Neuron ESB supports OAuth?
    a. True
    b. False
8. Which are types of credentials that Neuron ESB supports?
    a. Windows Domain
    b. Username/Password
    c. Certificates
    d. All of the above
9. What do client connectors use in order to determine which credentials have access to a service?
10. Service routing tables provide logical processing as well as routing?
    a. True
    b. False
11. Business processes are slower, with a higher latency, than service routing tables?
    a. True
    b. False

Appendix

The following artifacts accompany this training:

- Web Service Tutorials document
- Web Service Tutorials answer key

- Neuron ESB Configuration
- Service Routing Table Project for Visual Studio