Neuron Training: Web Services

Overview

This training will provide you with the knowledge necessary to begin using Neuron as a Web Services intermediary. After this training you will be able to:

- Describe how Neuron functions as a Services Intermediary
- Use Neuron for Location Transparency
- Diagnose issues using Neuron logging
- Use Neuron for binding mediation
- Use Neuron for One-way and Request-Response scenarios
- Use Neuron for Routing, Versioning and Transformation
- Use Neuron to pass REST requests
- Use Neuron as a Service Host including as a host for a façade that calls multiple web services

You should complete the exercises provided at the end of the training to confirm your understanding of the material presented.

Prerequisites

.NET 4.5.1

MSMQ installed and running

MSDTC configured and running

Visual Studio 2013

Neuron Fundamentals Training or equivalent experience

Experience with WCF and Web Services

Concepts

Neuron uses its own pub sub API in combination with Neuron Processes to function as a Services Intermediary. More specifically Neuron exposes two components that connect clients to Neuron and Neuron to Web Services:

- Client Connectors – A Client Connector is a configurable WCF Service Host that launches in its own App Domain and contains an embedded Neuron Party. This Party publishes requests to the bus with a Message Semantic of either Request or Multicast. Request Semantic is used when the

Messaging Pattern is configured as Request-Reply in the Client Connector's Binding Properties tab.

- Service Connectors – A Service Connector is a configurable WCF Service Proxy that launches in its own App Domain and contains an embedded Neuron Party. This Party subscribes to Topics on the bus and depending on if it's required by the configured Messaging Pattern publishes a response from the target service onto the bus.

The fact that Neuron uses its own Parties to route Web Services traffic makes it an extremely powerful intermediary. This is because Processes can be attached to Parties to accomplish nearly any processing required including calling Web Services inline, using databases, injecting custom code, XSLT transforms etc.

This functionality is so complete you can use Neuron to create and host services that normally would be self-hosted or hosted in IIS. That's because the effective use of Neuron Processes and Client Connectors can be used to create a complex real time orchestration.

Neuron endpoints are loosely typed. This means Neuron Endpoints do not require a specific WSDL or contract. This means by default Neuron handles the On Ramp pattern extremely easily. The On Ramp pattern is when you expose a single endpoint that can front multiple backend services or systems.

This also means Neuron does not generate WSDL for its service endpoints. This makes sense when you think about it because if you can send more than one packet to the same endpoint and those packets can be from completely unrelated services what WSDL would you expose?

This does not mean Neuron cannot enforce contracts or a schema. Neuron can use any type of processing including schema validation to enforce a contract through the use of Processes.

Basic Location Transparency

One of the most basic uses of Neuron with Web Services is to provide for Location Transparency. Location Transparency is achieved when the client connects to an endpoint that is different than the actual target service. Among other things, this allows the service to be moved without the client having to change its settings.

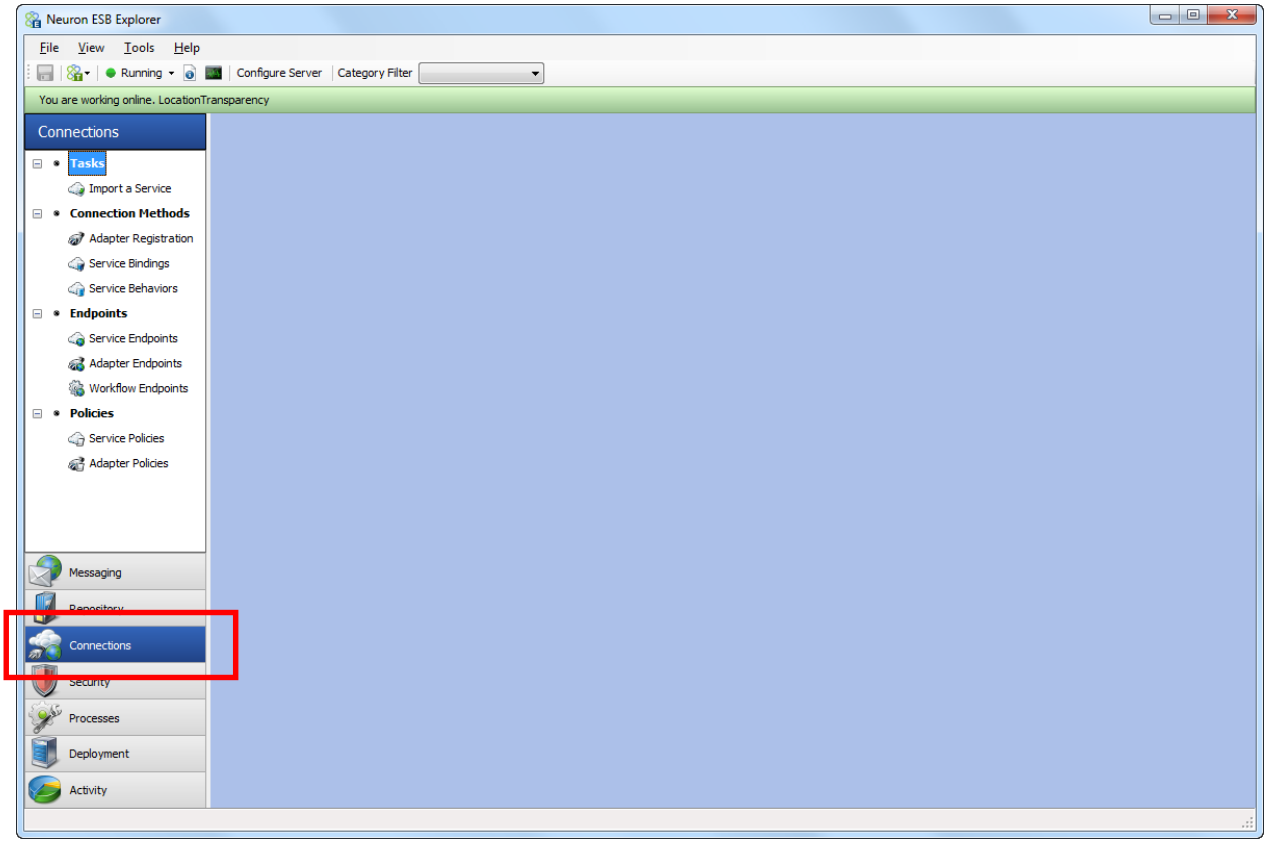
Complete the following steps to prepare for this training section (If any of the following steps are unfamiliar please complete the Neuron Fundamentals training before attempting to continue):

1. Create a new configuration and save it in the folder LocationTransparency
2. Configure Neuron to use the LocationTransparency configuration and set the logging level to verbose.
3. Restart Neuron.
4. Reconnect to Neuron using "Connect" mode in the initial screen so you are working on the live ESB service and add the following (Hint: these are the default names when clicking the New button)

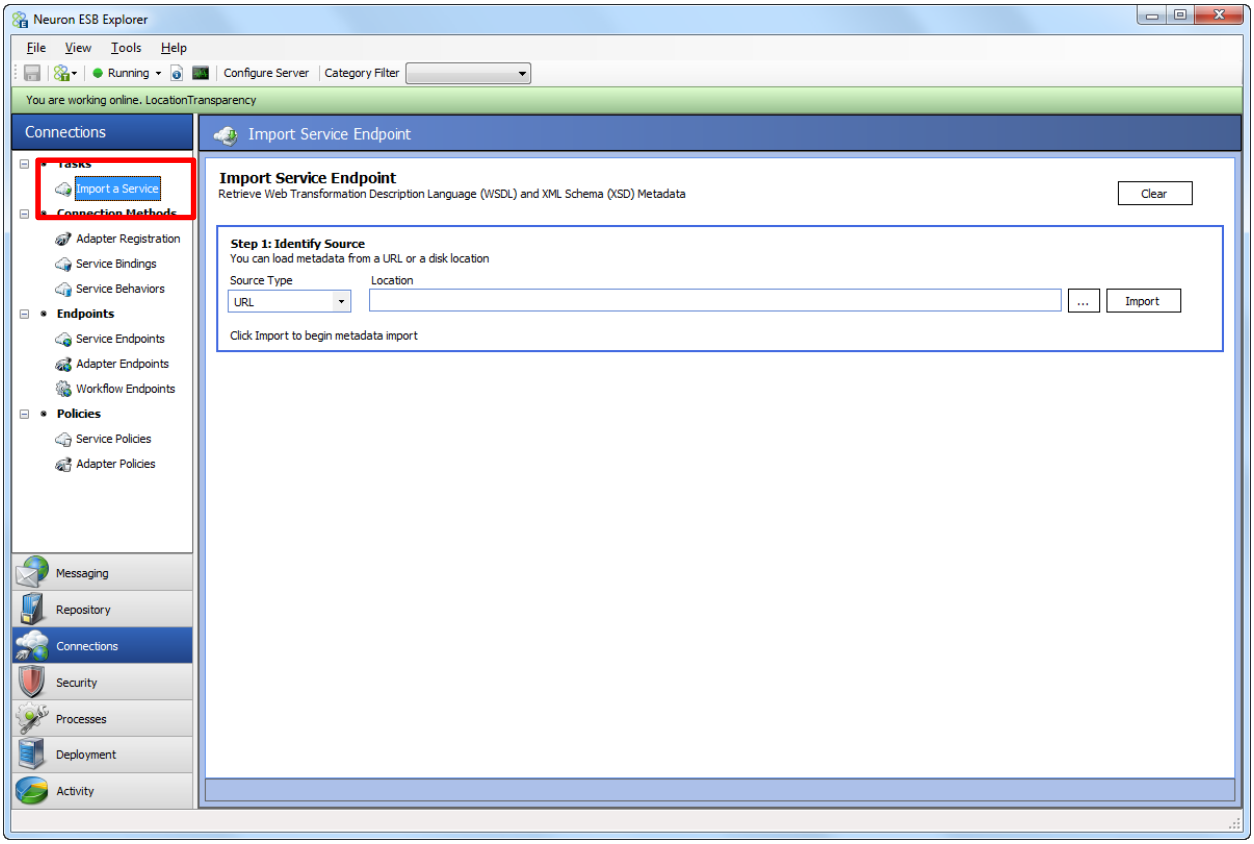
- A Topic called Topic1
- A Publisher called Publisher1 that subscribes to Topic1 (Send + Receive)
- A Publisher called Publisher2 that subscribes to Topic1 (Send + Receive)

5. Save your changes.

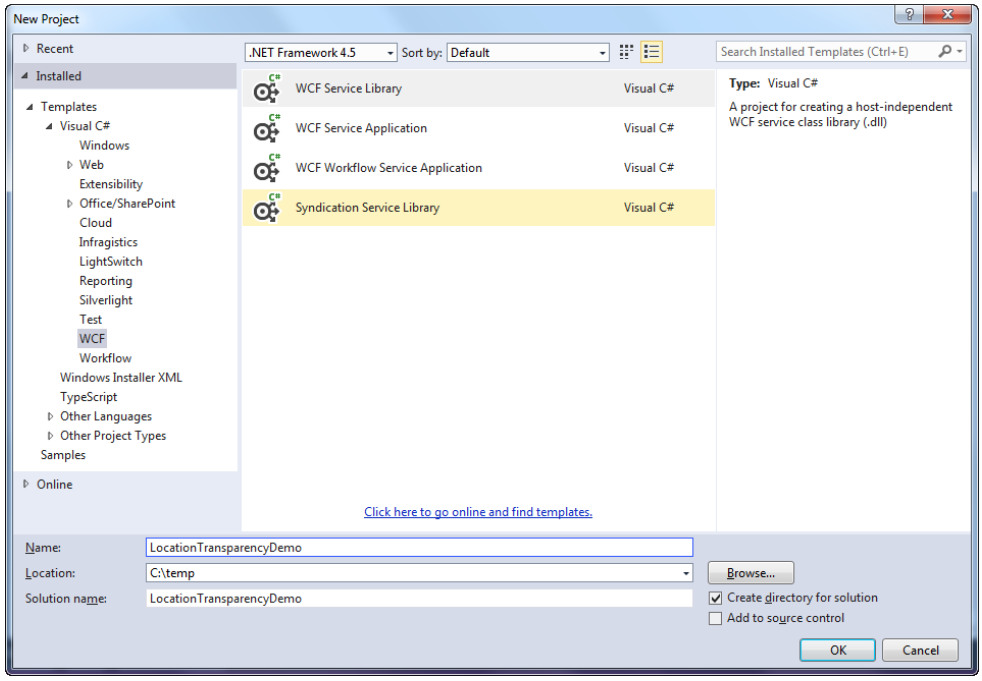
Click the Connections tab



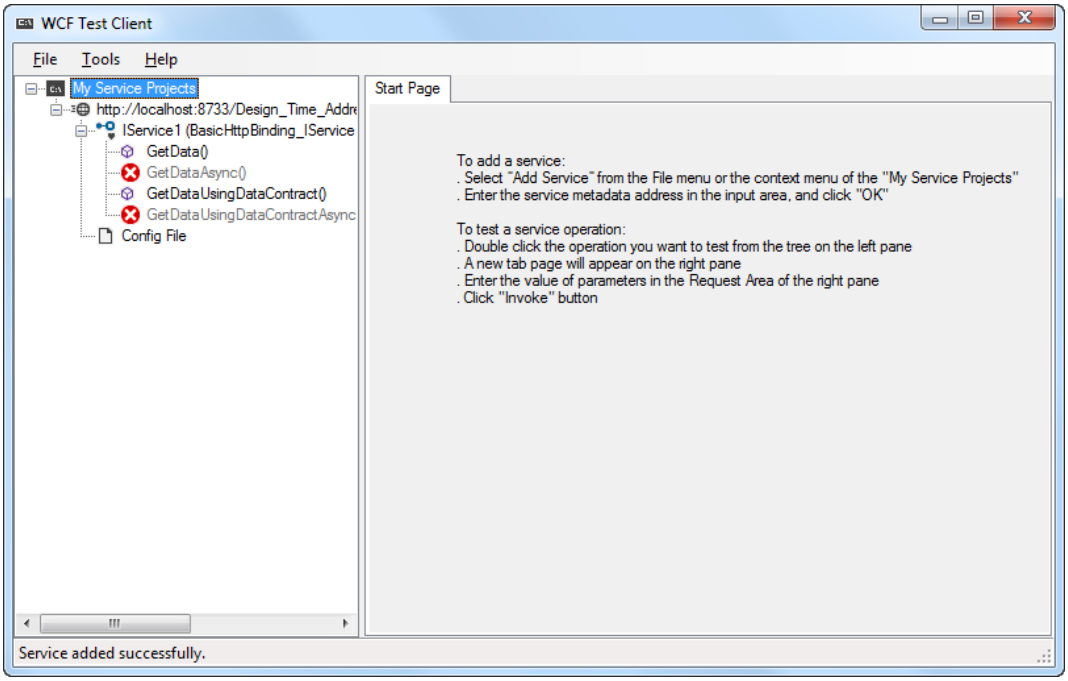
Click Import a Service



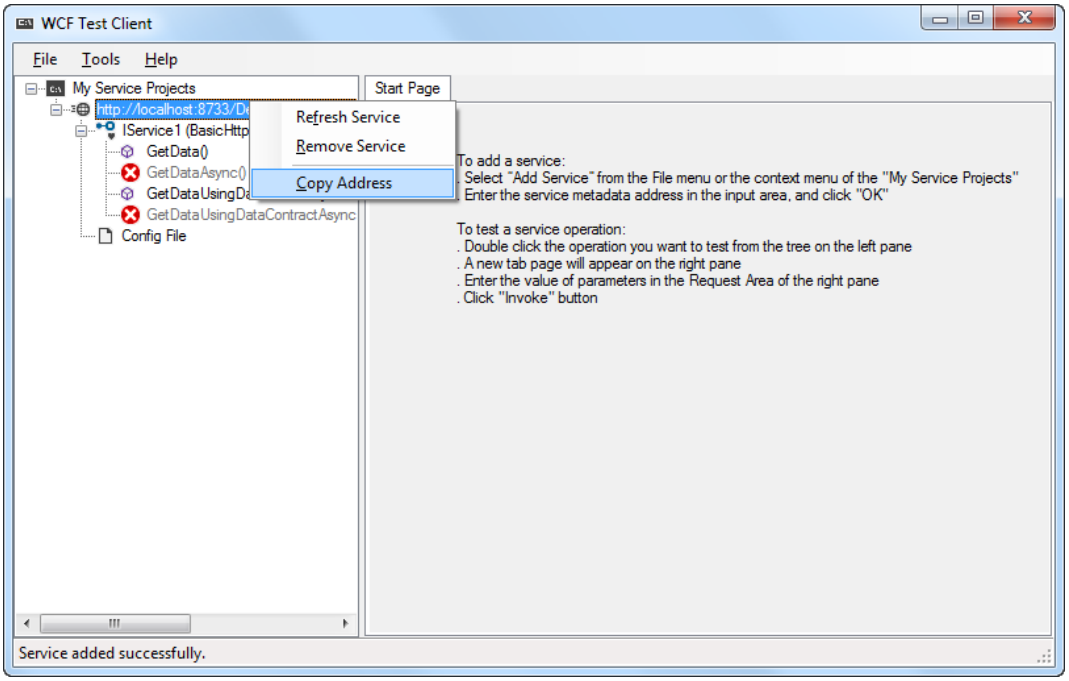
Minimize the Neuron Explorer and open Visual Studio. Create a WCF Service Library Project called LocationTransparencyDemo.



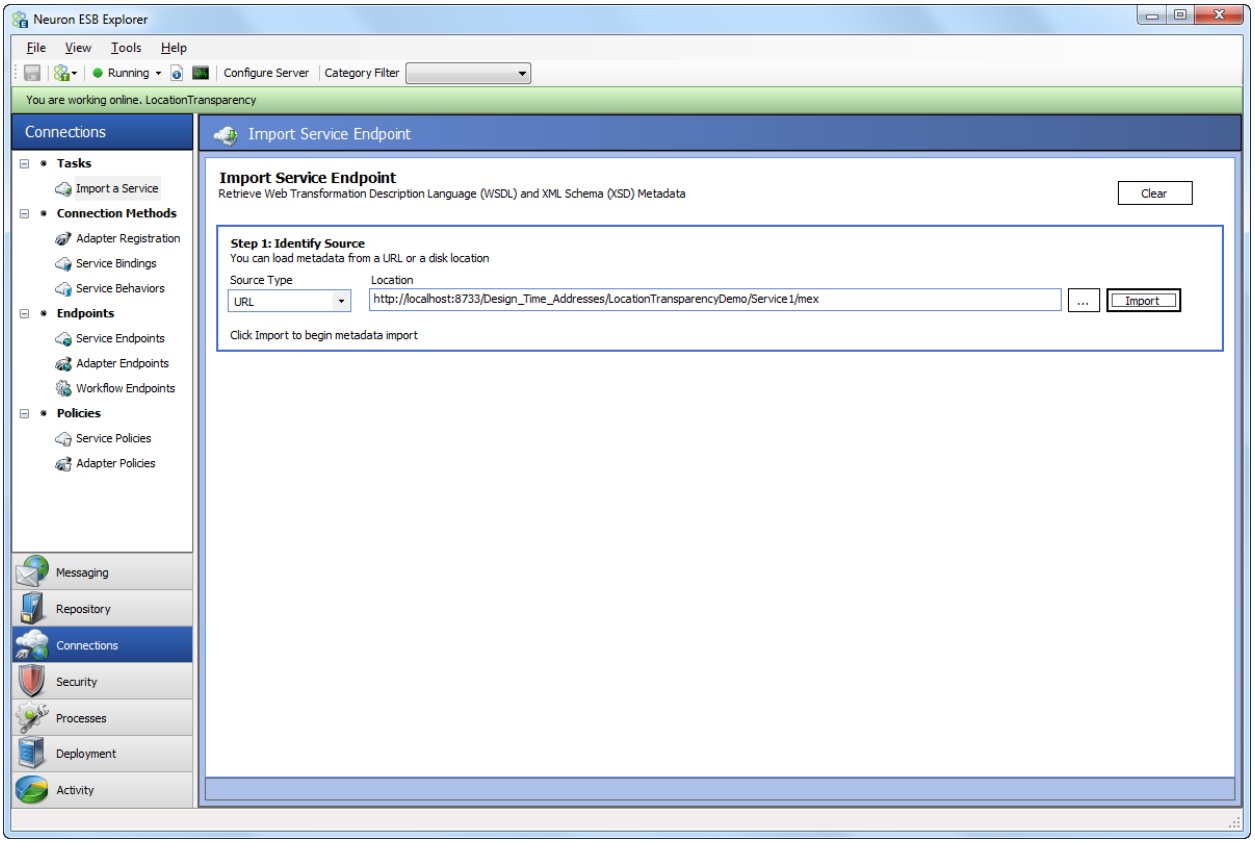
This will create a solution and project that will automatically host a service when you press Ctrl-F5 and start the WCF Test Client.



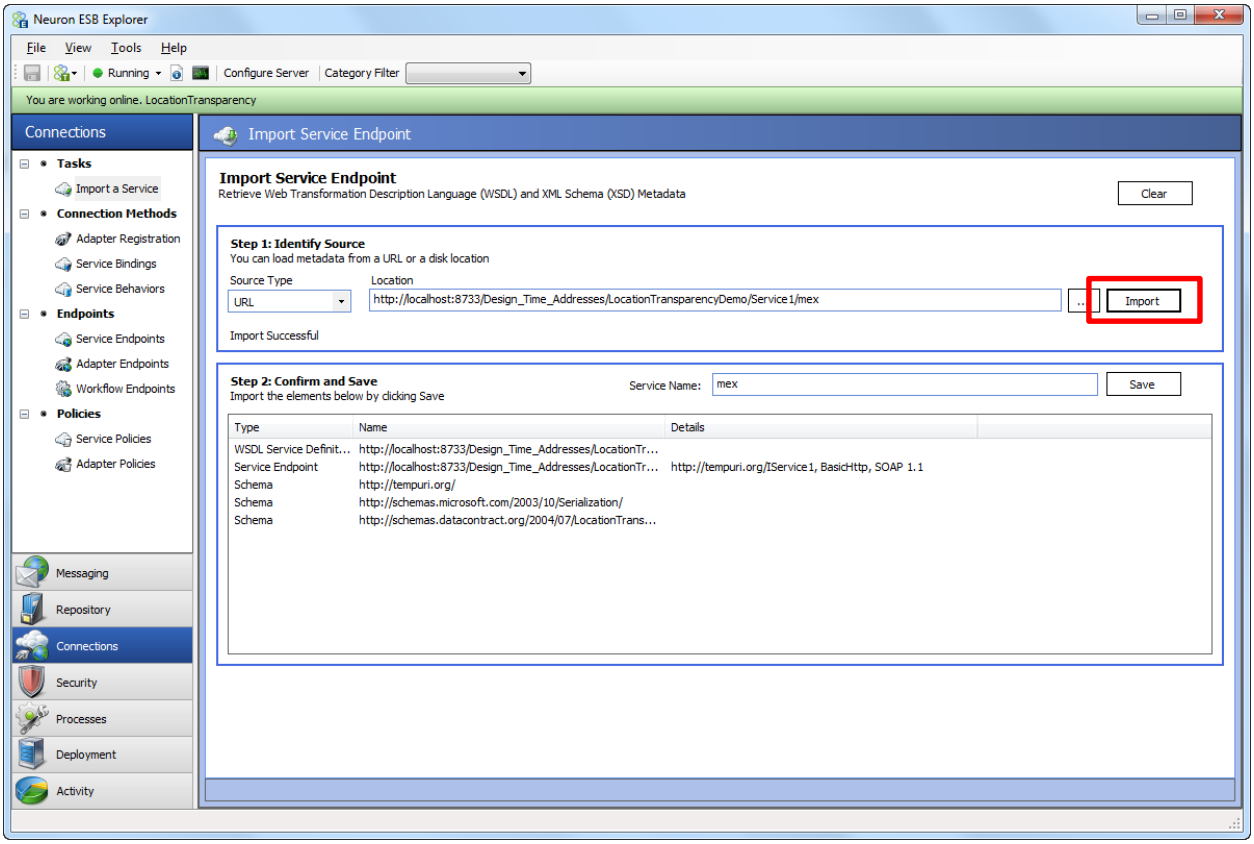
Copy the Address in the WCF Test Client.



Restore the Neuron Explorer window and paste the Service address into the Location test box. Neuron can utilize MEX or WSDL

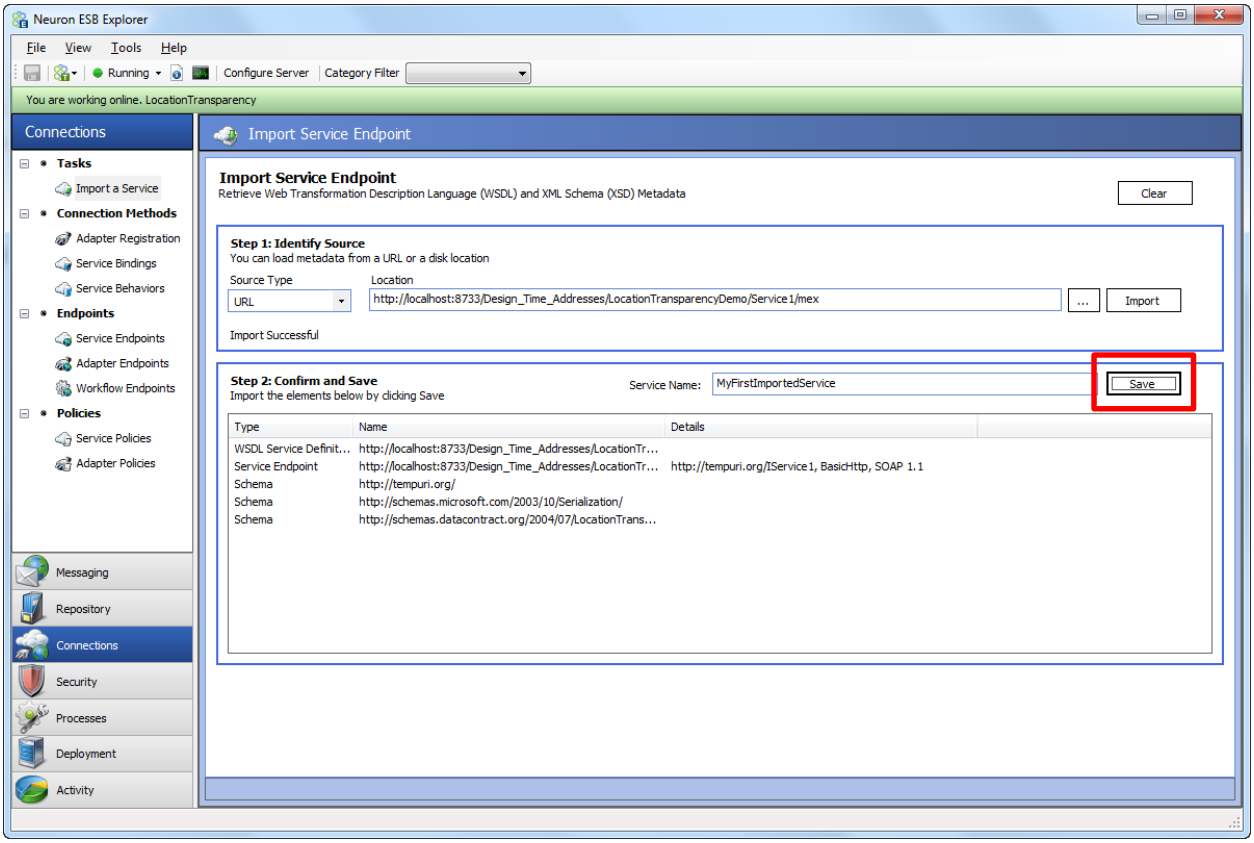


Click the Import button to the right of the ellipses.

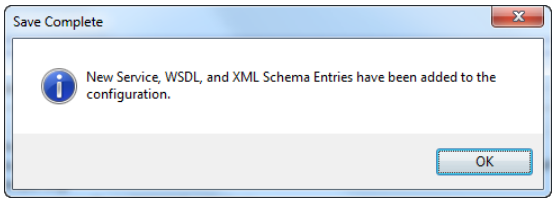


Notice the artifacts imported.

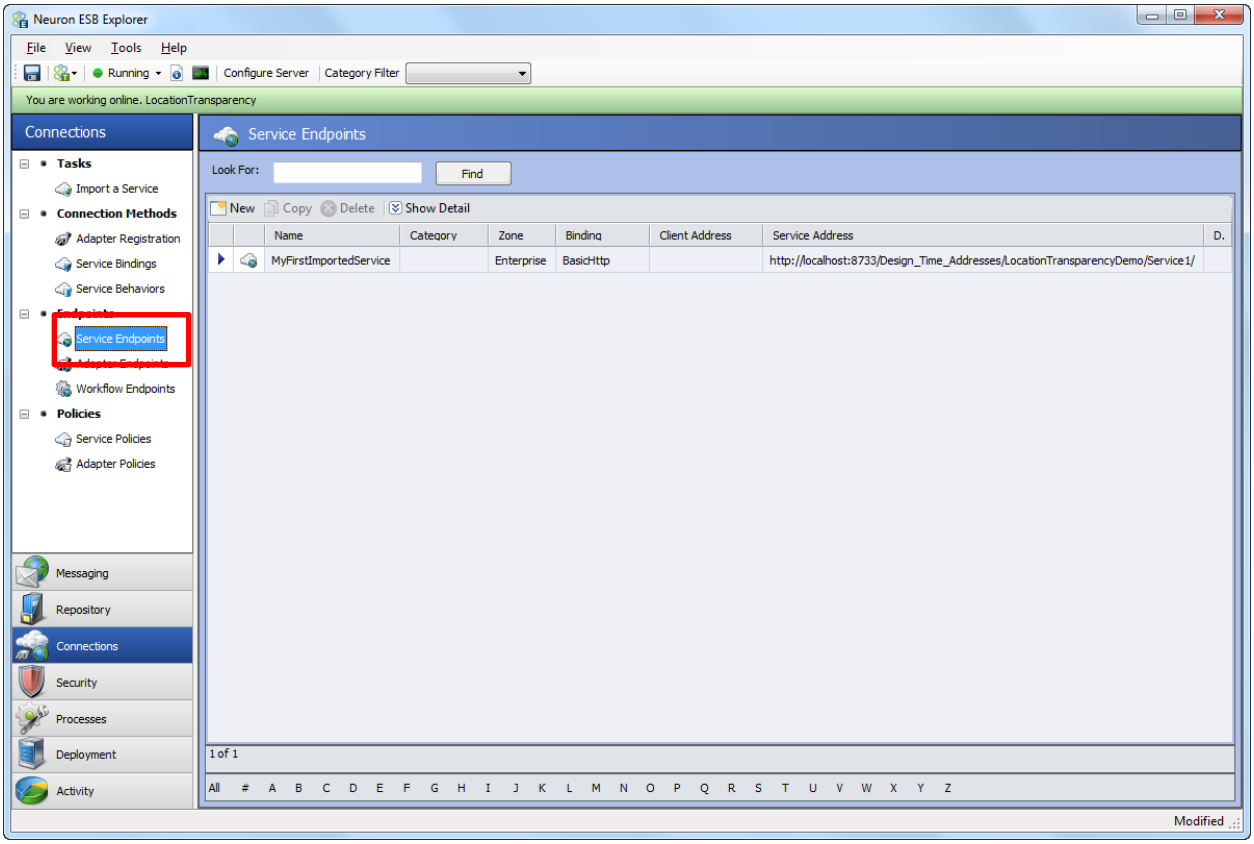
Rename the Service MyFirstImportedService and then click the Save button to the right of the Service Name dialog.



Click OK at the prompt informing you about imported artifacts.

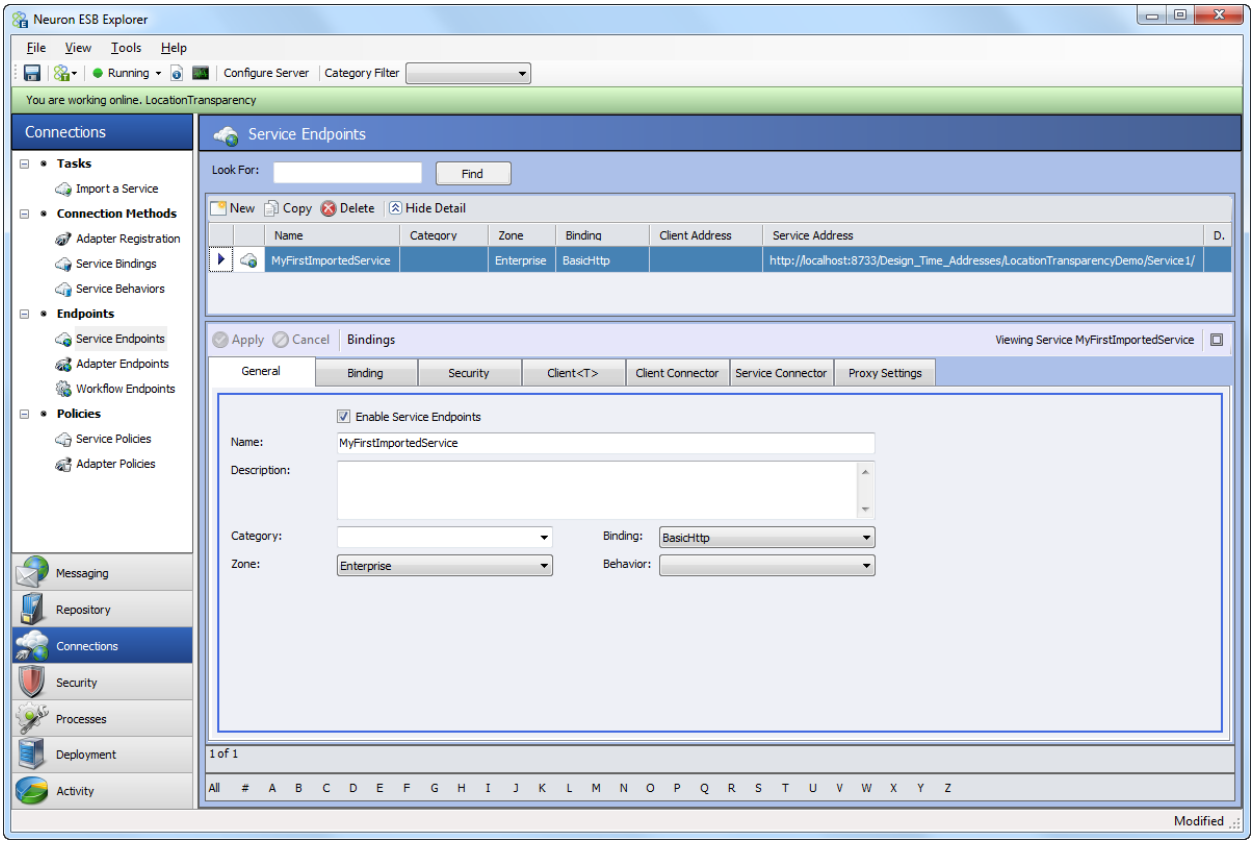


This will take you back to the import screen. Click the Service Endpoints option

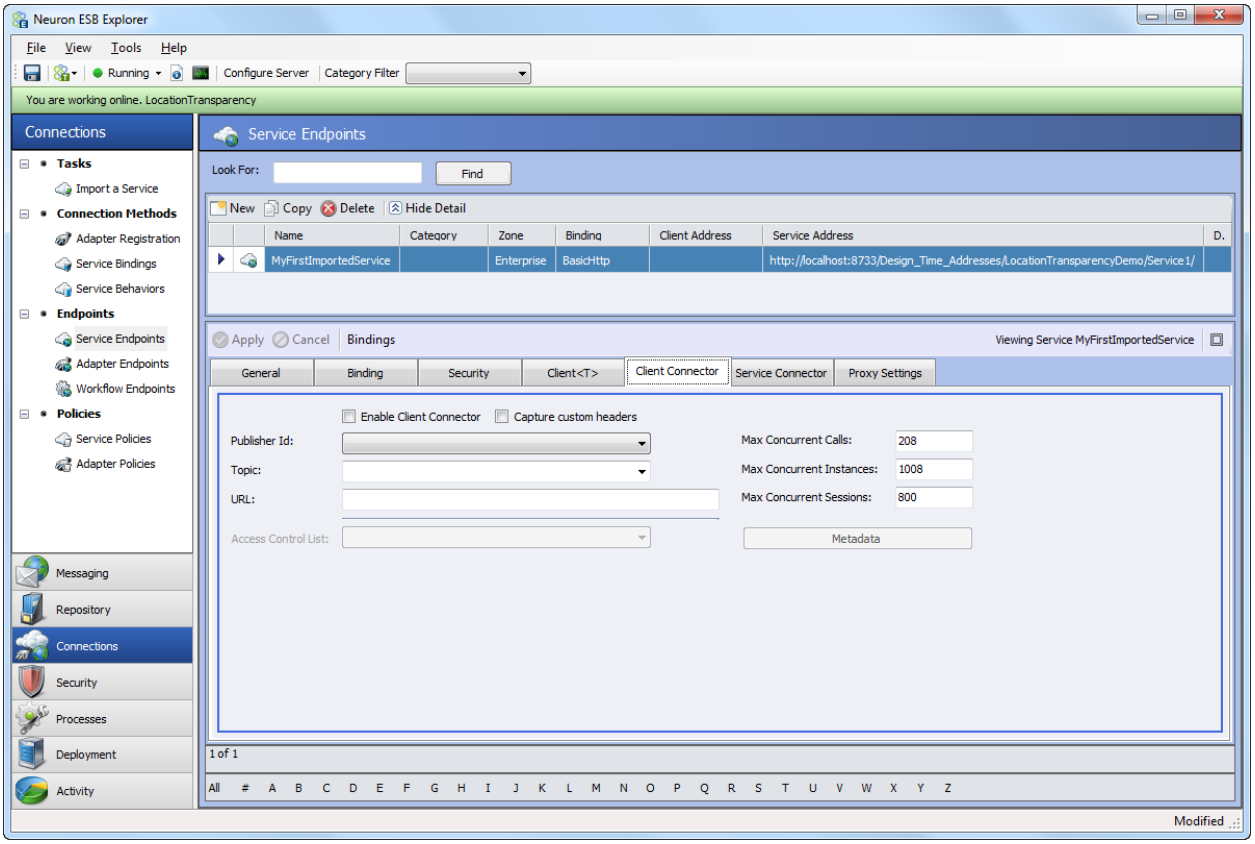


Notice your new Service is listed in the main panel. Notice also that the target Service Address is filled in but the Client Address is not. This is because this service endpoint is going to call a Web service, not host one. We will be creating a client connector next.

Click on your new service in the main panel.

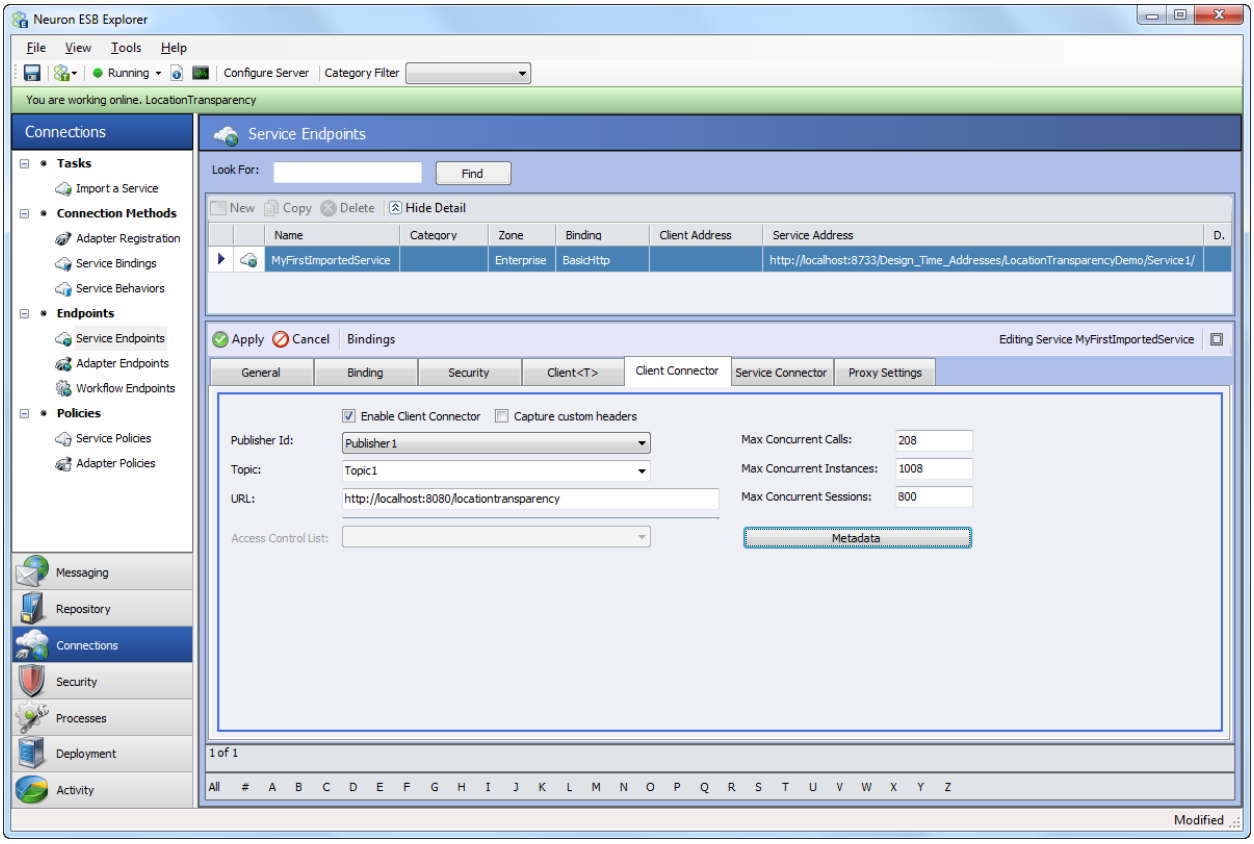


Now click on the tab that says Client Connector



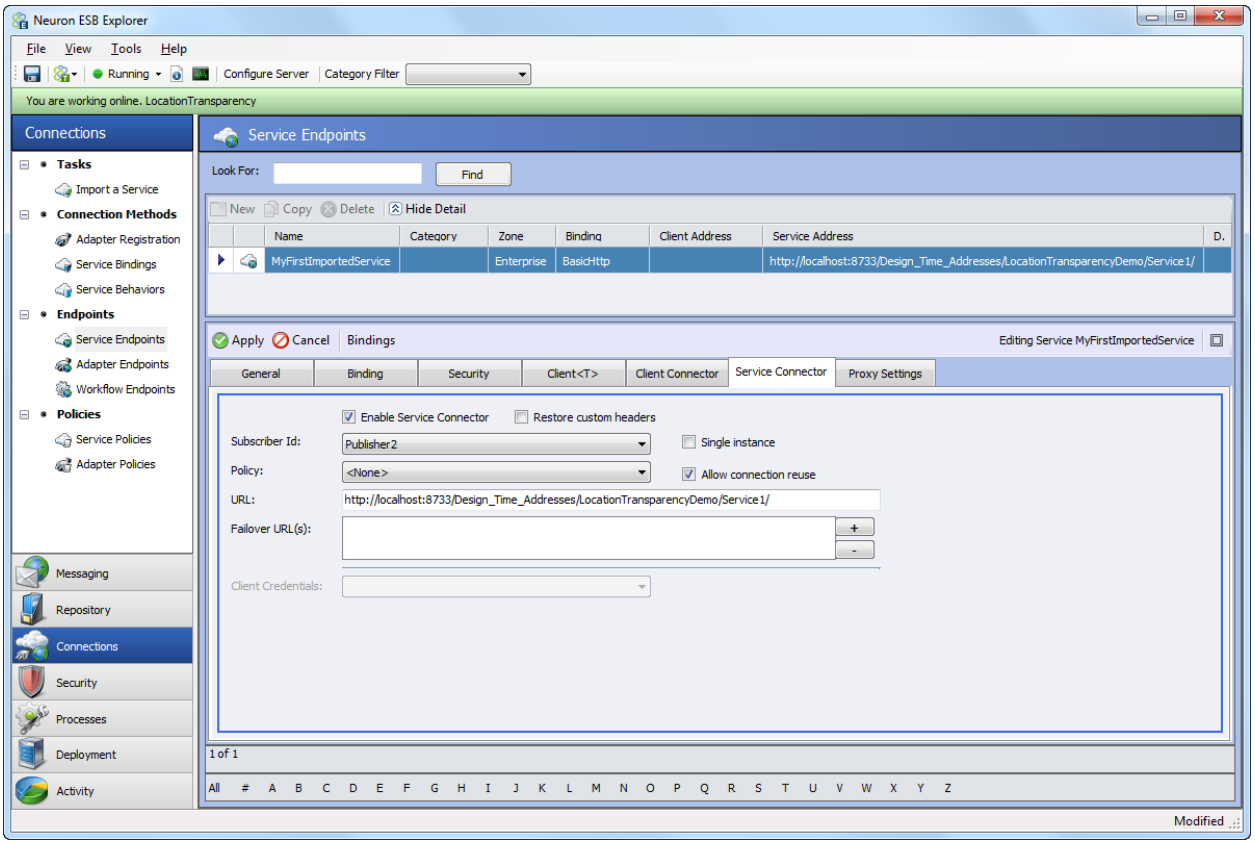
Perform the following:

- Check Enable Client Connector
- Click the Publisher Id dropdown and select Publisher1
- Click the Topic dropdown and select Topic1
- In the URL textbox type in http://localhost:8080/locationtransparency



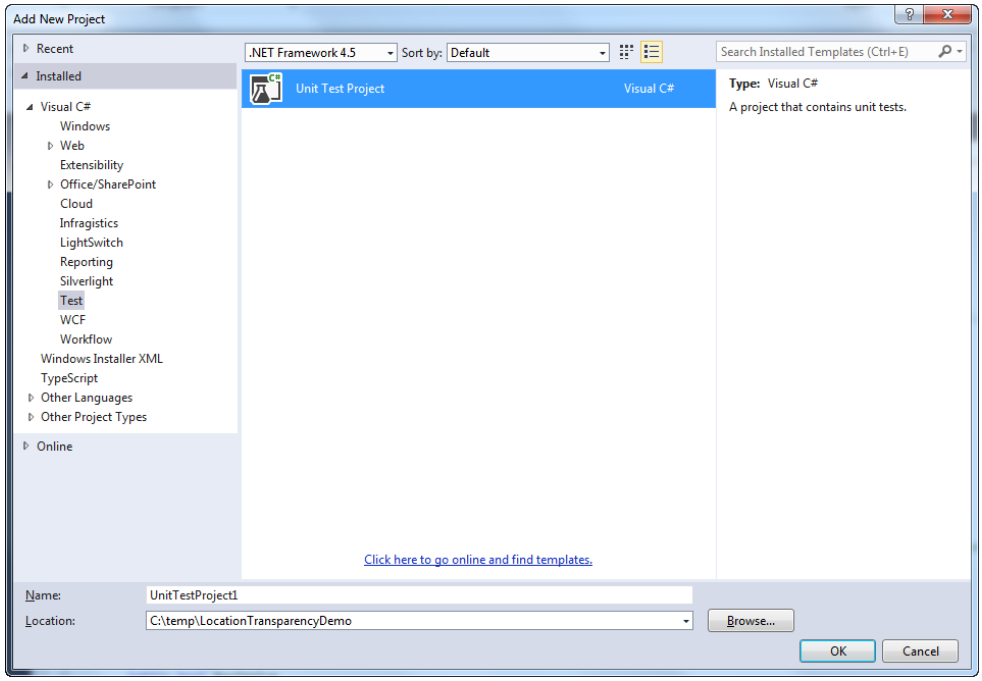
Click the Service Connector tab and perform the following:

- Check Enable Service Connector
- Click the Subscriber Id dropdown and select Publisher2
- Click the Policy dropdown and set the policy to <None>

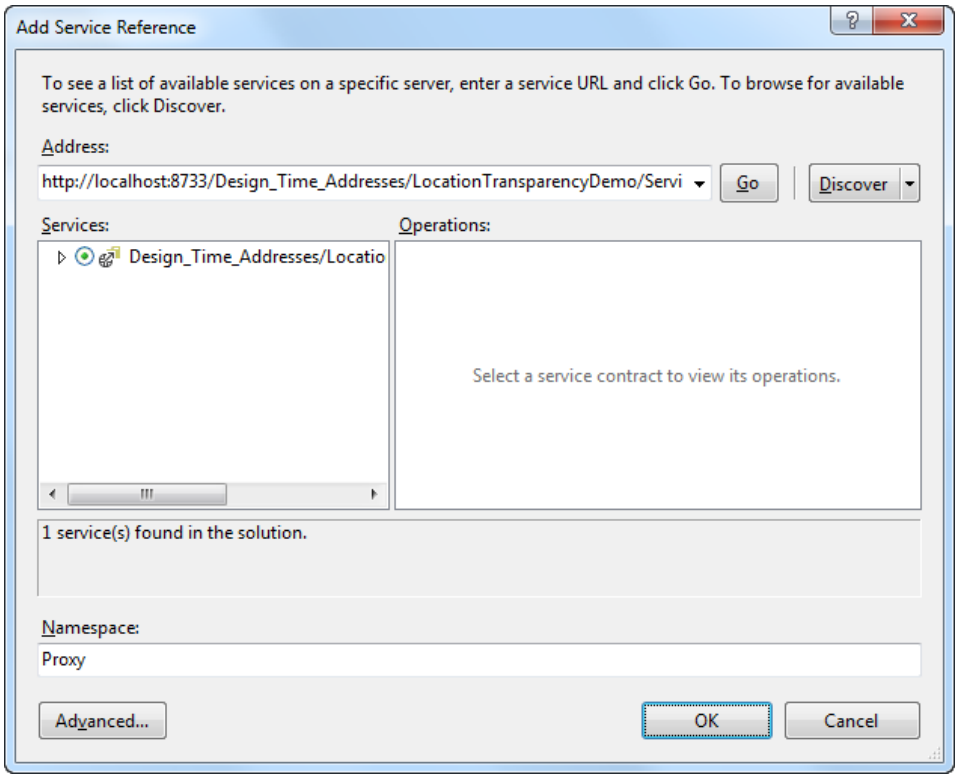


Click Apply and then Click Save.

Minimize Neuron Explorer and return to Visual Studio. Add a new project of type Test to the solution.



Right-click on the service project and select Add->Service Reference. In the Add Service Reference dialog, click the Discover button to automatically add the WCF service. Change the Namespace to Proxy and press the OK button.



Add `using UnitTestProject1.Proxy;` to the using section at the top of the UnitTest1.cs file.

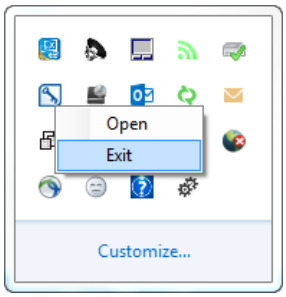
Add the following code to the generated TestMethod1:

```
Console.WriteLine(new Service1Client().GetData(42));
```

Open the App.config file in the UnitTestProject1 project and edit the client endpoint so it uses the address of the Neuron Client Connector instead of connecting to the service directly (modify the highlighted element below):

```
<system.serviceModel>
    <bindings>
        <basicHttpBinding>
            <binding name="BasicHttpBinding_IService1" />
        </basicHttpBinding>
    </bindings>
    <client>
        <endpoint address="http://localhost:8080/locationtransparency"
            binding="basicHttpBinding" bindingConfiguration="BasicHttpBinding_IService1"
            contract="Proxy.IService1" name="BasicHttpBinding_IService1" />
    </client>
</system.serviceModel>
```

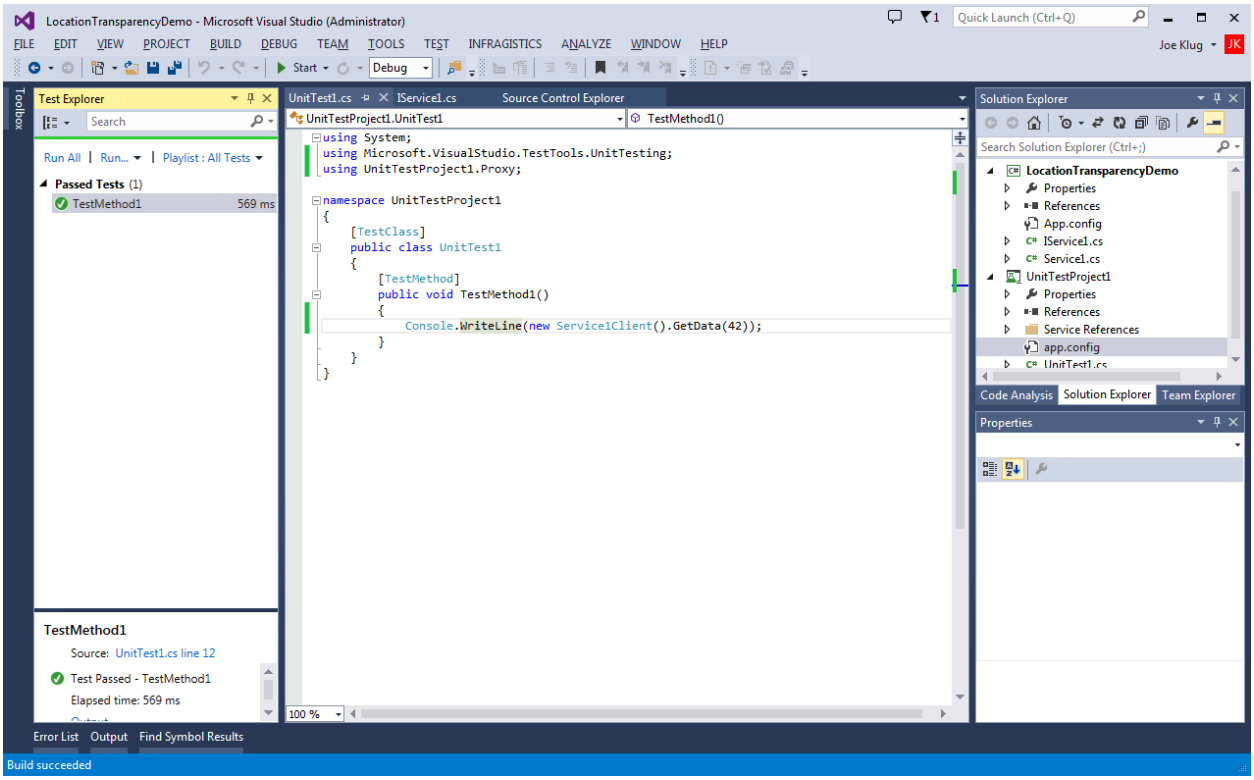
Reset the project and ready it for recompile by stopping the running WCF Service Host. You can usually do this by right-clicking the gear icon in that appears in your taskbar and selecting Exit.



Press F6 to build the solution and then press Ctrl-F5 to restart everything. Minimize the WCF Test Client

If you do not already have the Test Explorer available use the VS Menu to make it available by navigating to Test>Windows>Test Explorer.

Switch to Test View. Highlight TestMethod1, right click and choose Run Selected Tests. After a pause for startup of the test harness you should see the method run and pass



Congratulations you have run your first Web Service Scenario through Neuron!

At first glance this may not seem particularly beneficial from a SOA standpoint because we essentially have a 1-1 mapping of endpoint to client.

But, this simple use of an intermediary can be used to evolve your SOA infrastructure. At this point you have prepared yourself to accomplish the following without requiring any changes to the client or the target service

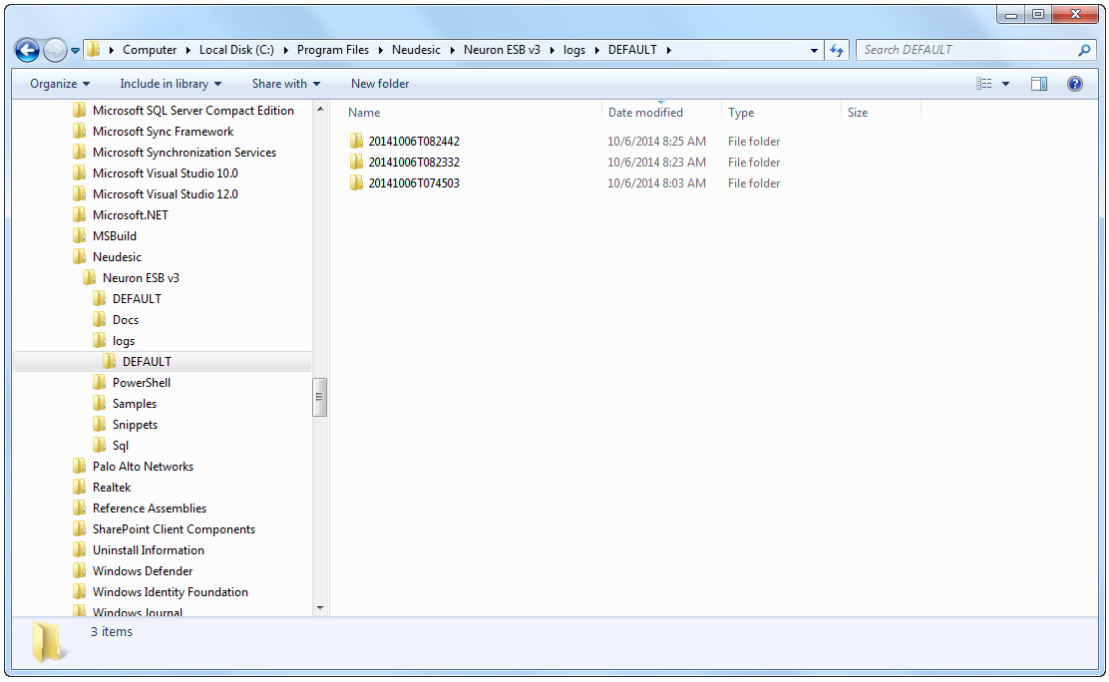
- Move the service address
- Change the Service Binding
- Insert Transforms or custom code
- Log all request and replies
- Broadcast any part of the conversation to interested subscribers
- Insert failover or retry logic
- Use a queued channel for durability
- Insert security logic that guards the target service
- Introduce validation logic
- Migrate to an On Ramp

That's a pretty powerful list of opportunities for such as simple starting point.

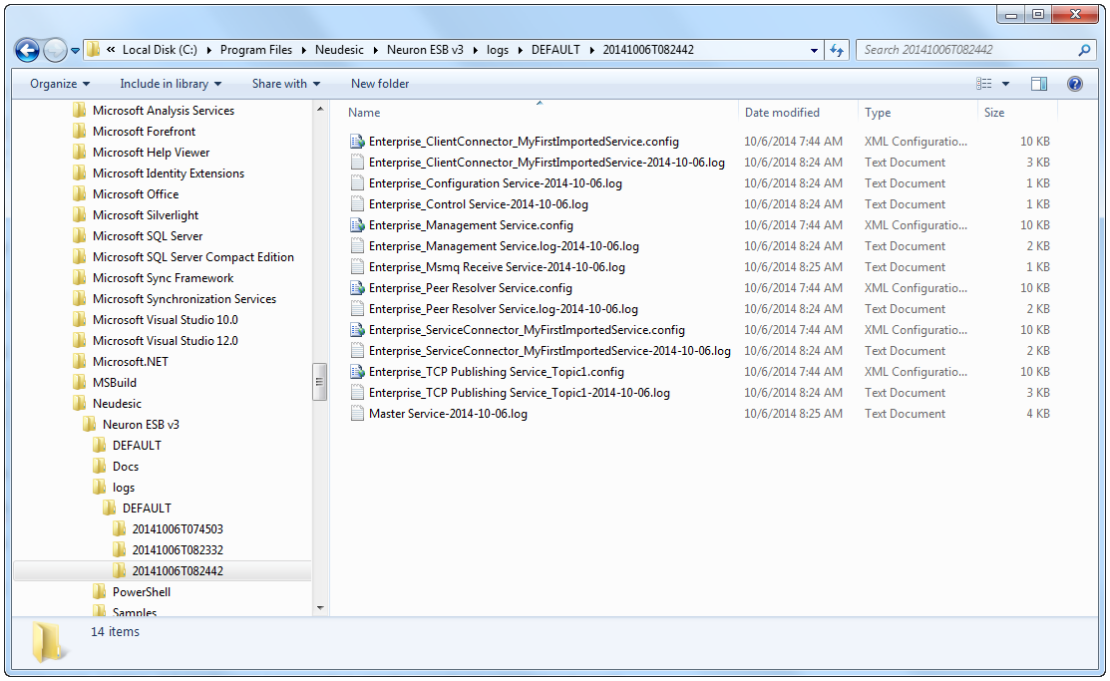
Logging

Logging can be an invaluable tool in understanding your services flow and troubleshooting. We will now examine the logs created by the previous exercise.

Navigate to the logs subdirectory under the Neuron ESB program files folder. There will be a separate subfolder for each Neuron instance you install. If this is your first time working with Neuron, you probably only have a DEFAULT instance installed. You should see multiple subdirectories that correspond to the number of times you've restarted the Neuron ESB service



Sort the directories in reverse order so the latest directory is on top. Navigate to that directory.



If you have named your service as instructed you should see a log for your Client Connector named Enterprise_ClientConnector_MyFirstImportedService-{Todays Date}.log and a log for your Service Connector named Enterprise_ServiceConnector_MyFirstImportedService-{Todays Date}.log

If you have configured logging for maximum verbosity as you were instructed to earlier in this training you will see a wealth of information in both logs including the full SOAP packet.

These logs can be used for gathering packet knowledge for Transforms and for troubleshooting.

Return to Visual Studio and the LocationTransparencyDemo solution. If the WCF Service Host is still running, shut it down. Open Test View and run TestMethod1 again. Wait a minute for the expected failure to occur.

Navigate back to the logs folder pictured above, open the Enterprise_ ServiceConnector_ MyFirstImportedService-{Todays Date}.log and scroll towards the bottom. You should see something similar to a stack trace that begins like this:

```
2014-10-06 08:29:42.243-04:00 [11] WARN  - Communication exception. There was no endpoint
listening at http://localhost:8733/Design_Time_Addresses/LocationTransparencyDemo/Service1/ that
could accept the message. This is often caused by an incorrect address or SOAP action. See
InnerException, if present, for more details.

2014-10-06 08:29:42.278-04:00 [11] DEBUG - Message:

Exception Type: System.ServiceModel.EndpointNotFoundException

Exception Message: There was no endpoint listening at
http://localhost:8733/Design_Time_Addresses/LocationTransparencyDemo/Service1/ that could accept
```

```
the message. This is often caused by an incorrect address or SOAP action. See InnerException, if
present, for more details.

Exception Trace:

Server stack trace:
```

Examine the corresponding Client Connector log.

One final note about logging before the training continues. Verbose logging is great for development and troubleshooting but is discouraged for continuous production use. Verbose logging impacts performance and can use considerable disk space.

Binding Mediation

Binding Mediation occurs when a client connects to an intermediary with one binding then communicates with the target service using another binding. In this exercise, we will modify our test project in the LocationTransparencyDemo solution so that TestMethod1's proxy instance will communicate using BasicHttpBinding as before and a new method we will add cleverly called TestMethod2 will communicate using a WsHttpBinding.

Return to Visual Studio and the LocationTransparencyDemo solution. Modify the code in the Test Project so that TestMethod1 looks like this:

```
[TestMethod]
public void TestMethod1()
{
    Console.WriteLine(new Service1Client("basic").GetData(42));
}
```

Copy TestMethod1 including its TestMethod Attribute below TestMethod1. Change the 1 to a 2 on the end of the method and change "basic" in the Service1Client constructor to "wshttp".

You code should now look like this.

```
[TestMethod]
public void TestMethod1()
{
    Console.WriteLine(new Service1Client("basic").GetData(42));
}

[TestMethod]
public void TestMethod2()
{
    Console.WriteLine(new Service1Client("wshttp").GetData(42));
}
```

Open the app.config file and add a new binding element for wsHttpBinding. Your bindings section should look like this:

```
<bindings>
    <basicHttpBinding>
        <binding name="BasicHttpBinding_IService1" />
    </basicHttpBinding>
  <wsHttpBinding>
    <binding name="WSHttpBinding_IService1">
```

```
      <security mode="None" />
    </binding>
  </wsHttpBinding>
</bindings>
```

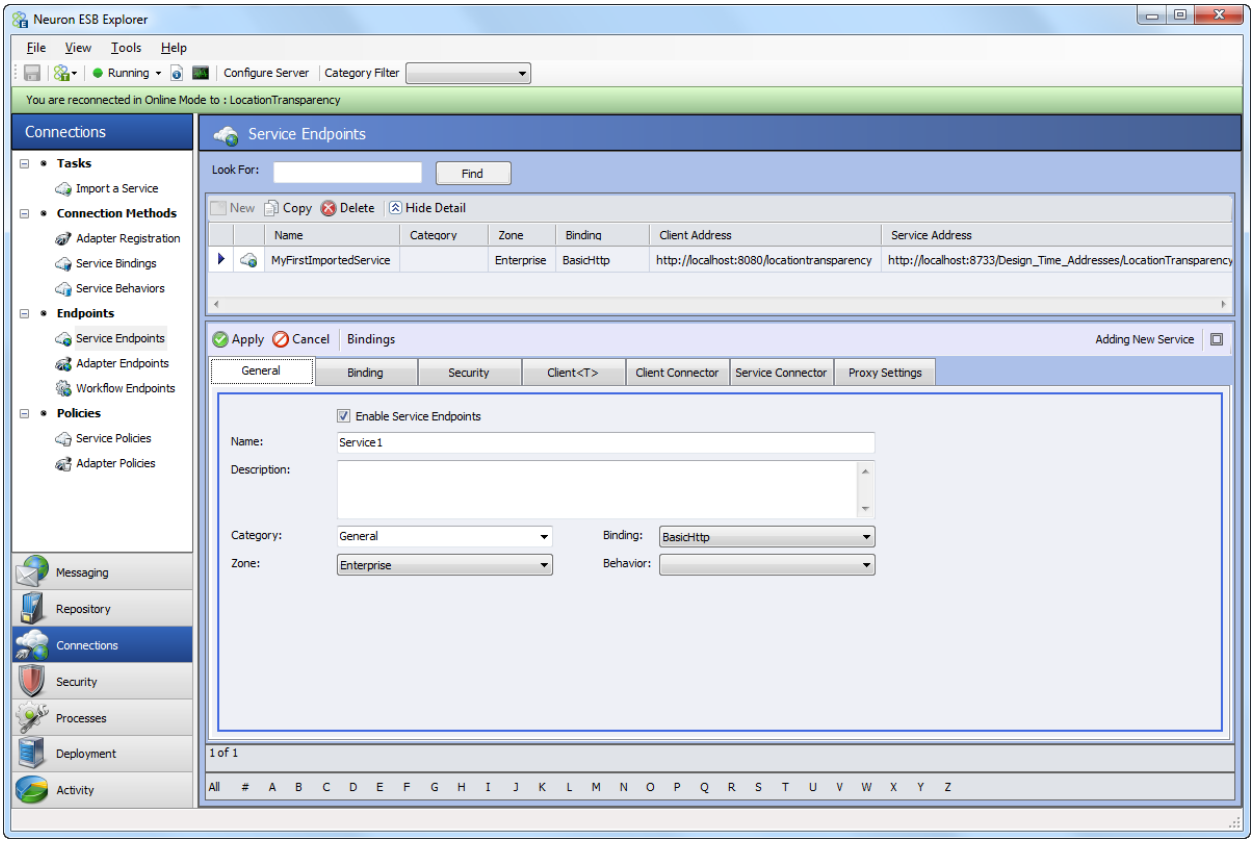
Change the client endpoint name to “wshttp”. Copy the endpoint and change the following attributes

- name=”wshttp”
- address=”http://localhost:8080/bindingmediation”
- binding=”wsHttpBinding”
- bindingConfiguration=”WSHttpBinding_IService1”

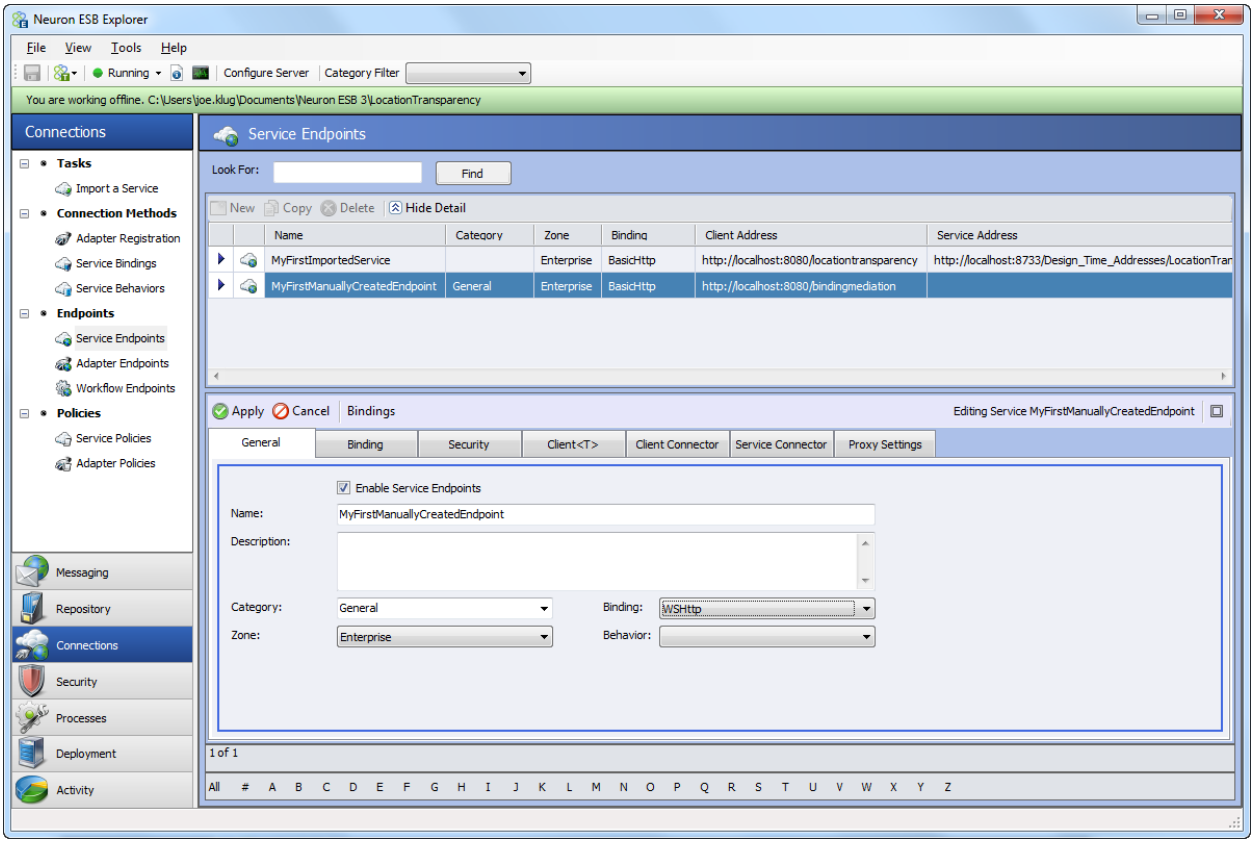
Your client section should now look like this:

```
<client>
    <endpoint address="http://localhost:8080/locationtransparency"
        binding="basicHttpBinding" bindingConfiguration="BasicHttpBinding_IService1"
        contract="Proxy.IService1" name="basic" />
    <endpoint address="http://localhost:8080/bindingmediation"
        binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_IService1"
        contract="Proxy.IService1" name="wshttp" />
</client>
```

Return to Neuron Explorer. Navigate to Connections > Service Endpoints and click the New button in the main panel. Click the General tab of the new Service Endpoint if it is not already chosen. You should see something similar to below:



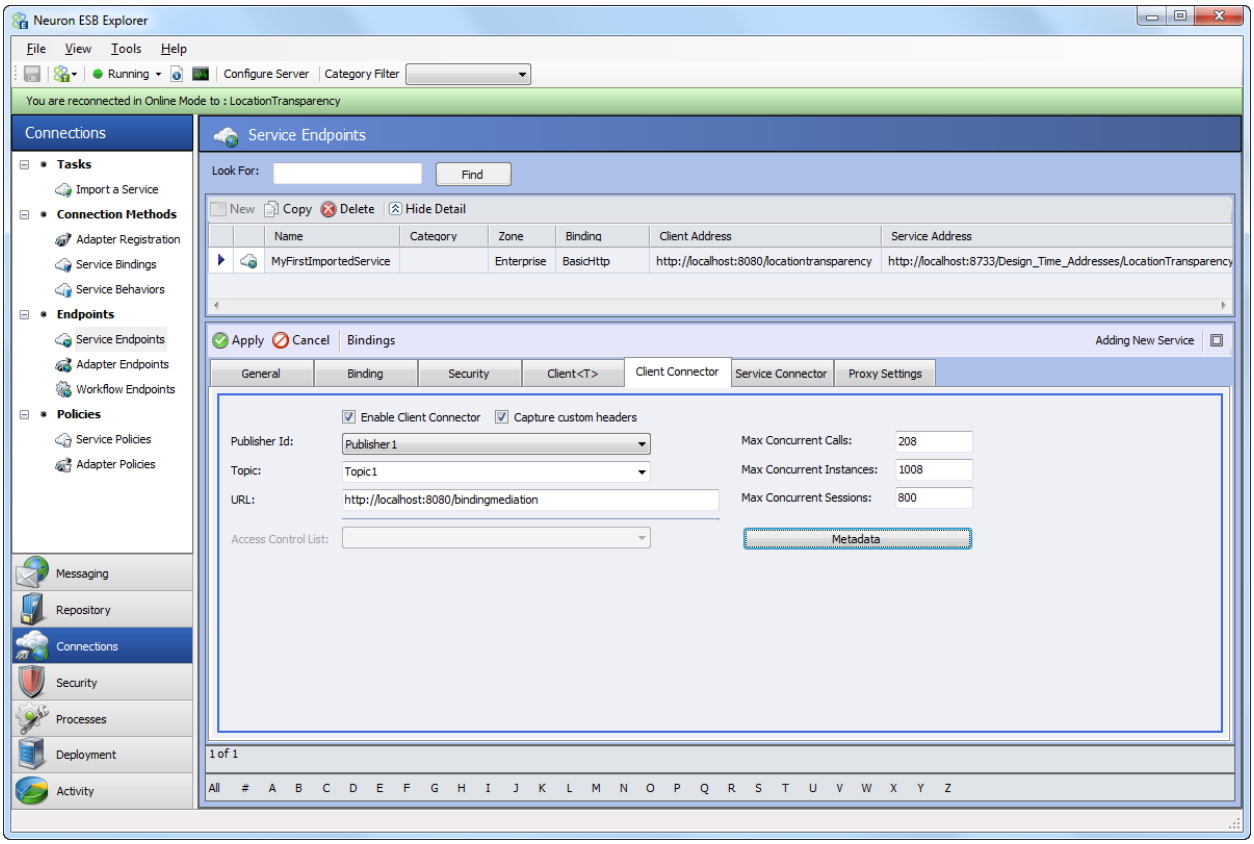
Change the text in the Name text box to MyFirstManuallyCreatedEndpoint and select WSHttp as the Binding:



Click the Client Connector tab and do the following:

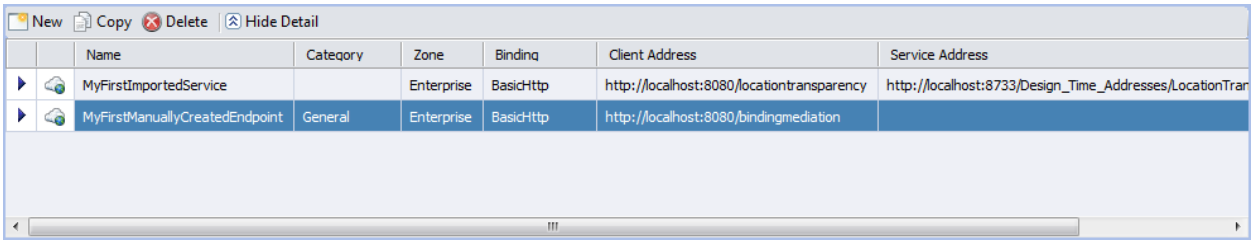
- Check Enable Client Connector
- Click the Publisher Id dropdown and choose Publisher1
- Click the Topic dropdown and choose Topic1
- Enter the following address in the URL test box http://localhost:8080/bindingmediation

Your Client Connector tab should now look like this:



Click Apply and Save.

Your main panel should now look like this.



Notice that the MyFirstManuallyCreatedEndpoint does not have a Service Address. This is actually a more accurate representation of what goes on internally in Neuron then using a single Service Endpoint to maintain the bindings for both a Client Connector and Service Connector. Client Connectors and Service Connectors are not actually connected in any way at runtime. Messages between the two are passed between their respective disconnected app domains using the Neuron API. When you import and enable both the Client Connector and Service Connector in a Single Service Endpoint you are essentially simply making sure they are set to the same WCF binding settings.

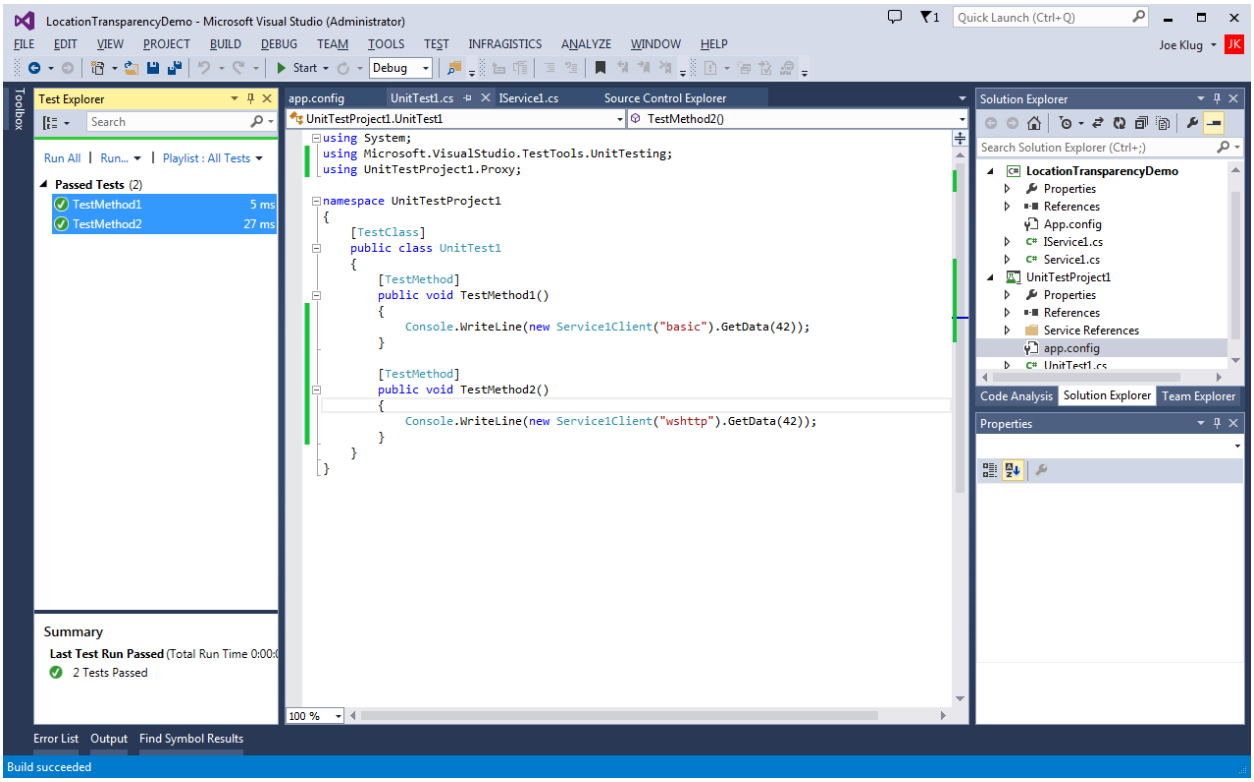
It is very common for experienced Neuron users to create the endpoints manually and split them from the very beginning.

We are now almost ready to run our scenario. Before we do however, navigate back to the log directory. You should now see an additional log named

Enterprise_ClientConnector_MyFirstManuallyCreatedEndpoint-{Todays Date}.log.

Examine the log to make sure everything started up correctly and then close the log.

Return to the LocationTransparencyDemo solution in Visual Studio and press F6 to compile and then Ctrl-F5 to start the WCF Test Client. Minimize the WCF Test Client and then switch to Test View and highlight TestMethod1 and TestMethod2, right-click and select Run Selected Tests:



Observe both tests passing. Open the Client Connector log for your manually created endpoint and observe the difference in the SOAP envelope.

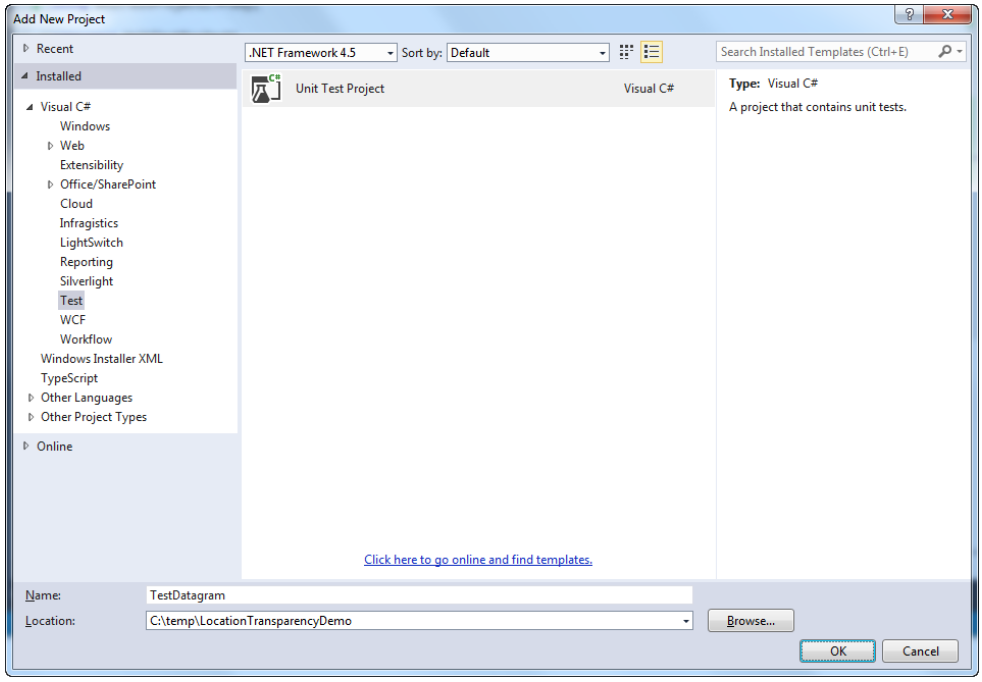
The difference you are seeing is caused by the bindings. By default BasicHttpBinding does not contain any security and the SOAP envelope contains the bare minimum of information. WsHttpBinding on the other hand defaults to Message Security based on the Windows Identity. This creates a large degree of overhead especially if the actual messages you are passing are very small.

When you create an endpoint manually you are responsible for setting the security in the Security tab of the connector to match the security you are trying to achieve with the binding you have selected.

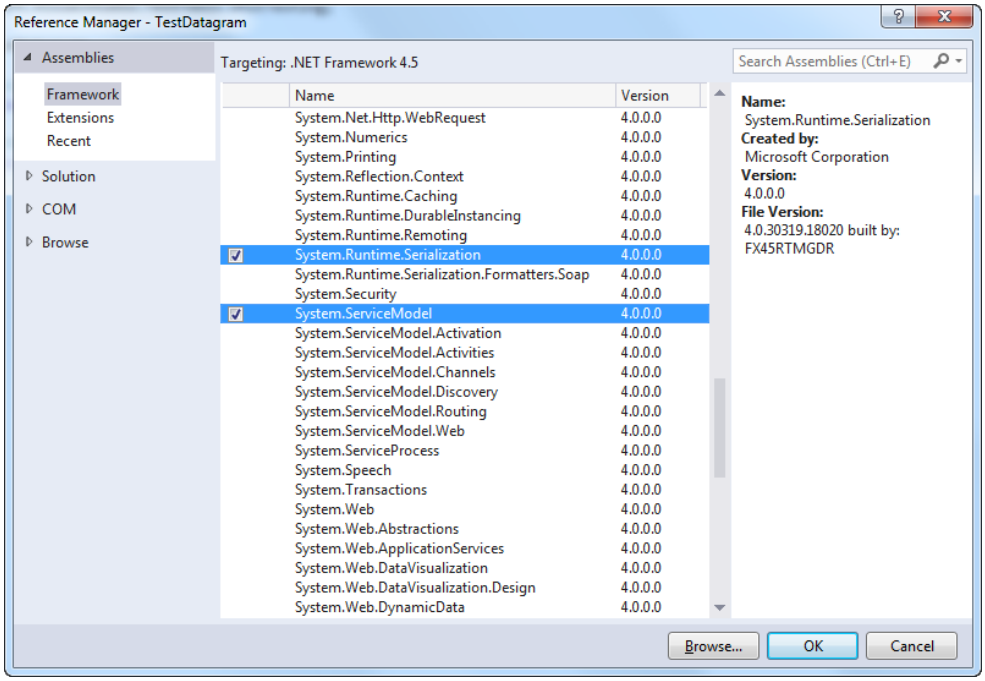
Datagram

So far we have been using the Request-Reply Message exchange pattern exclusively during this training. We are now going to introduce the concept of using the Client Connector as an asynchronous entry point onto the bus.

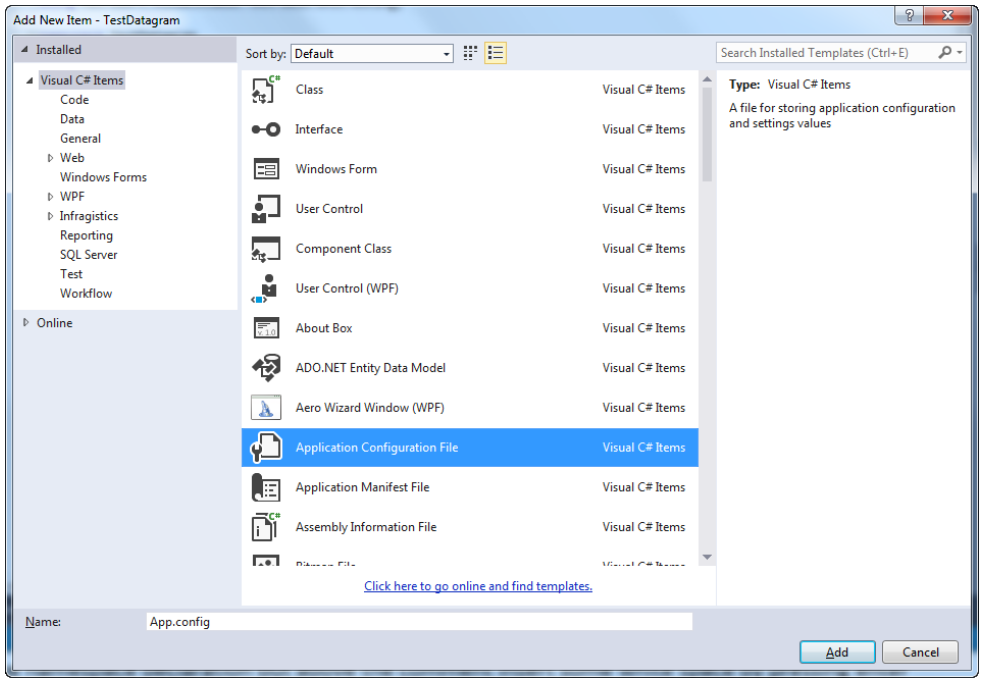
Create a new Visual Studio Project of type Test and call it TestDatagram



Right Click the project and Add References to System.ServiceModel and System.Runtime.Serialization:



Use the Add New Item functionality of the project to add an App.config file:



Now add `using System.ServiceModel;` to the list of using statements.

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 IOnlyDoOneway
{
    [OperationContract(IsOneWay=true)]
    void OneWay(string arg);
}
```

Enter the following code into TestMethod1

```
ChannelFactory<IOnlyDoOneway> oneway = new ChannelFactory<IOnlyDoOneway>("oneway");
IOnlyDoOneway proxy = oneway.CreateChannel();
proxy.OneWay("I love asynchronous communication!");
```

Add the following entry into the App.config file and inside of the existing configuration node:

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

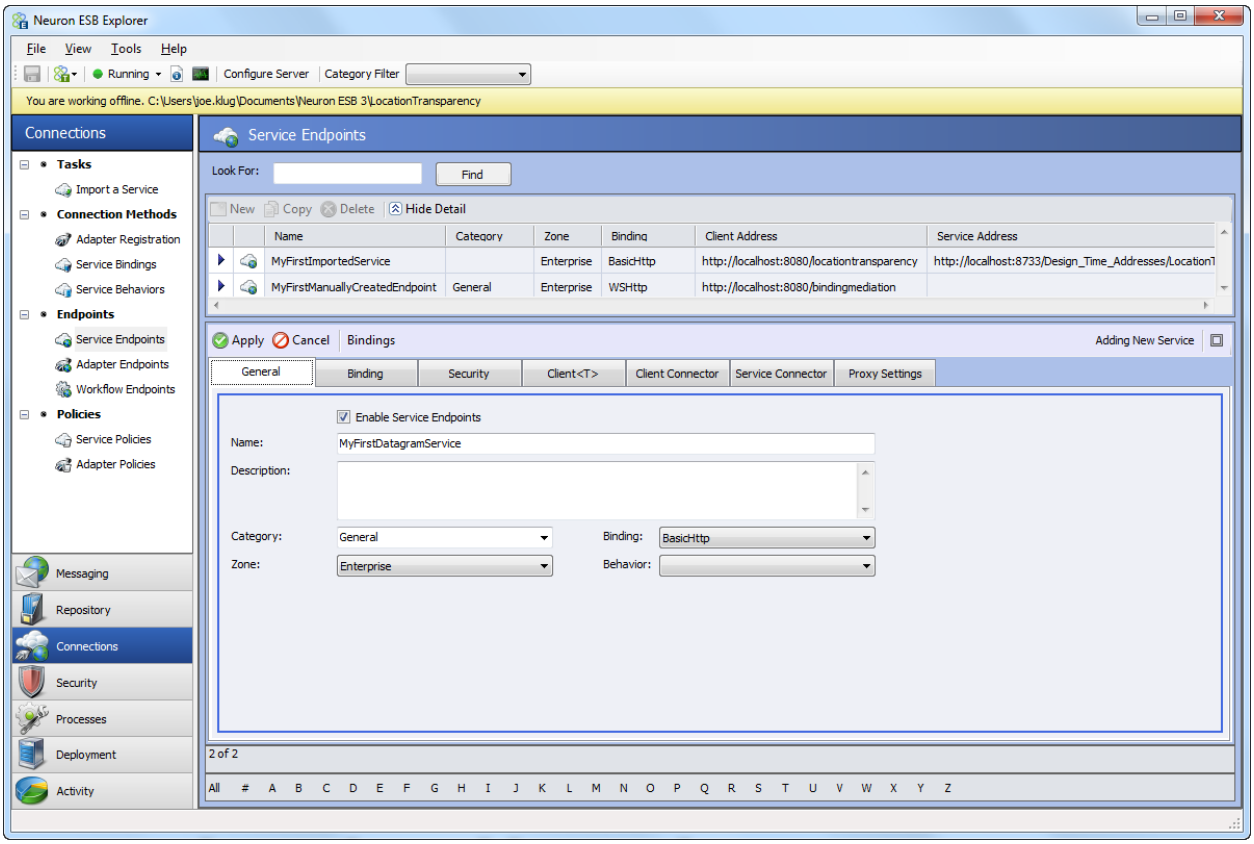
Save all, compile the solution but do not attempt to run it yet.

Restore Neuron Explorer.

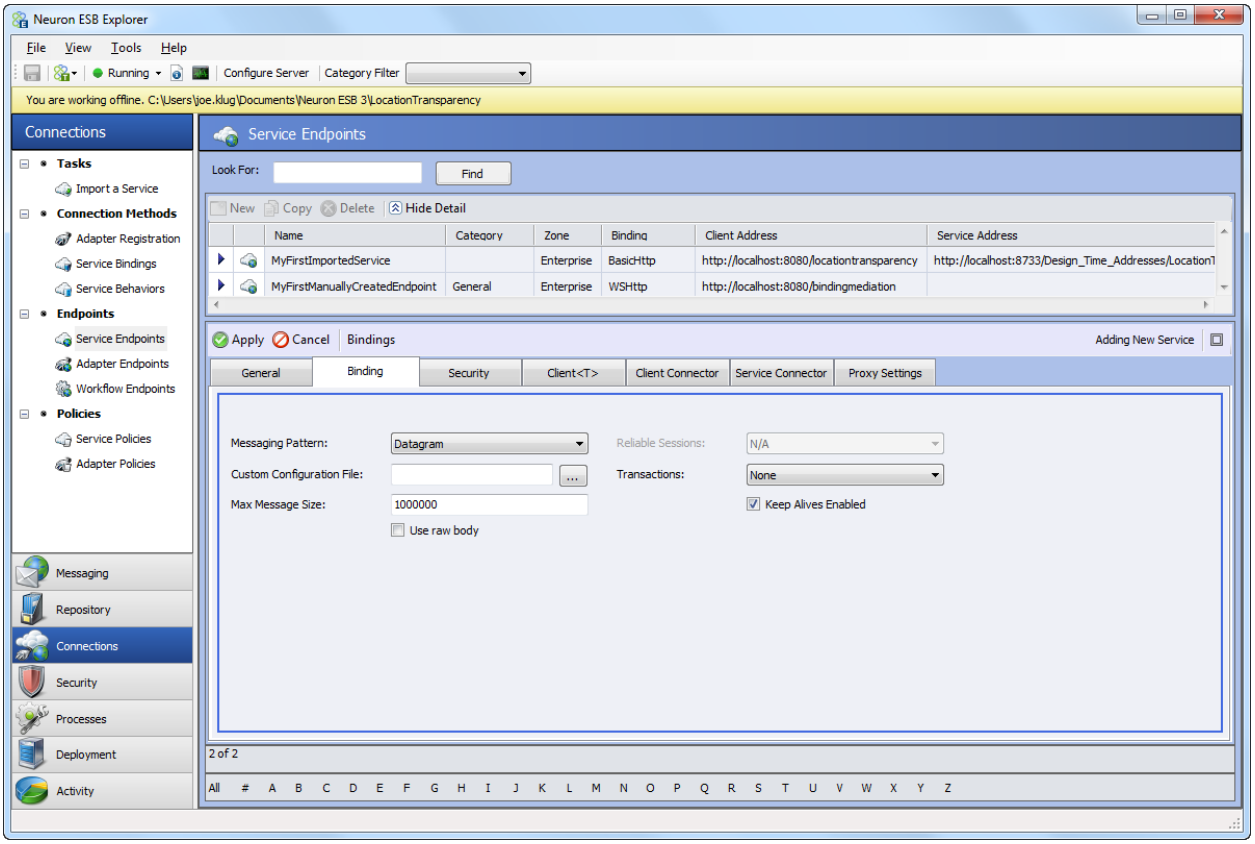
Complete the following steps (Hint: the names of the entities below are using the default when just hitting New):

- Add a new Topic called Topic2
- Add a new Publisher called Publisher3 that subscribes to Topic2 (Send + Receive)
- Add a new Publisher called Publisher4 that subscribes to Topic2 (Send + Receive)
- Save

Now, navigate from the Messaging tab back to Connections > Service Endpoints and click New on the main panel. Change the name in the General tab to MyFirstDatagramService:


Neuron ESB Explorer
File View Tools Help
Running
Configure Server
Category Filter
You are working offline. C:\Users\joe.klug\Documents\Neuron ESB 3\LocationTransparency
Connections
Tasks
Import a Service
Connection Methods
Adapter Registration
Service Bindings
Service Behaviors
Endpoints
Service Endpoints
Adapter Endpoints
Workflow Endpoints
Policies
Service Policies
Adapter Policies
Messaging
Repository
Connections
Security
Processes
Deployment
Activity
Service Endpoints
Look For:
Find
New Copy Delete Hide Detail
Name Category Zone Binding Client Address Service Address
MyFirstImportedService Enterprise BasicHttp http://localhost:8080/locationtransparency http://localhost:8733/Design_Time_Addresses/Location1
MyFirstManuallyCreatedEndpoint General Enterprise WSHttp http://localhost:8080/bindingmediation
Apply Cancel Bindings
Adding New Service
General Binding Security Client<T> Client Connector Service Connector Proxy Settings
Enable Service Endpoints
Name: MyFirstDatagramService
Description:
Category: General
Binding: BasicHttp
Zone: Enterprise
Behavior:
2 of 2
All # A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Click the Binding tab and change the Messaging Pattern to Datagram

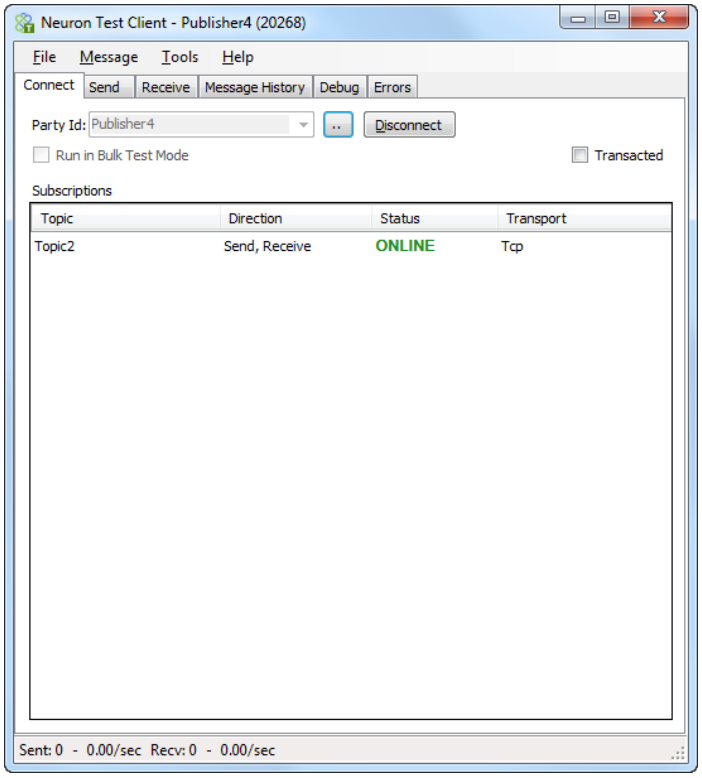


Click the Client Connector tab and do the following:

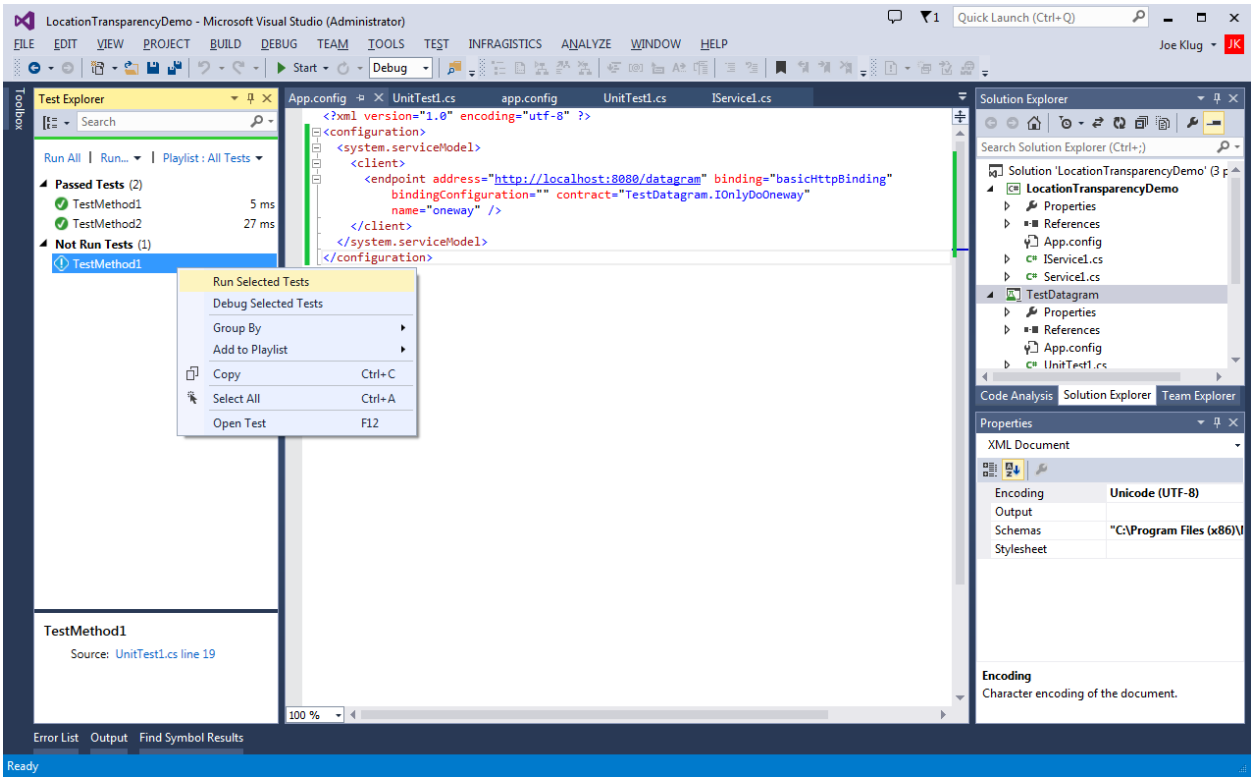
- Check Enable Client Connector
- Click the Publisher Id dropdown to select Publisher3
- Use the Topic dropdown to select Topic2
- Enter `http://localhost:8080/datagram` into the URL textbox

Click Apply and then Save the configuration. Navigate back to the logs directory and examine the newly created log for your new endpoint.

Now, use the Tools menu in Neuron Explorer to launch a Test Client and connect as Publisher4:

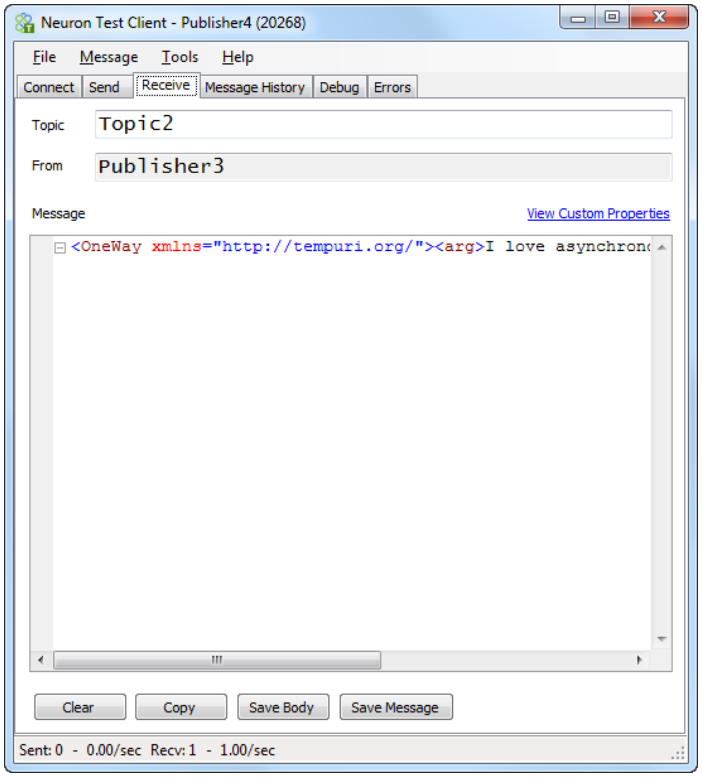


Switch back to Visual Studio and prepare to run TestMethod1:

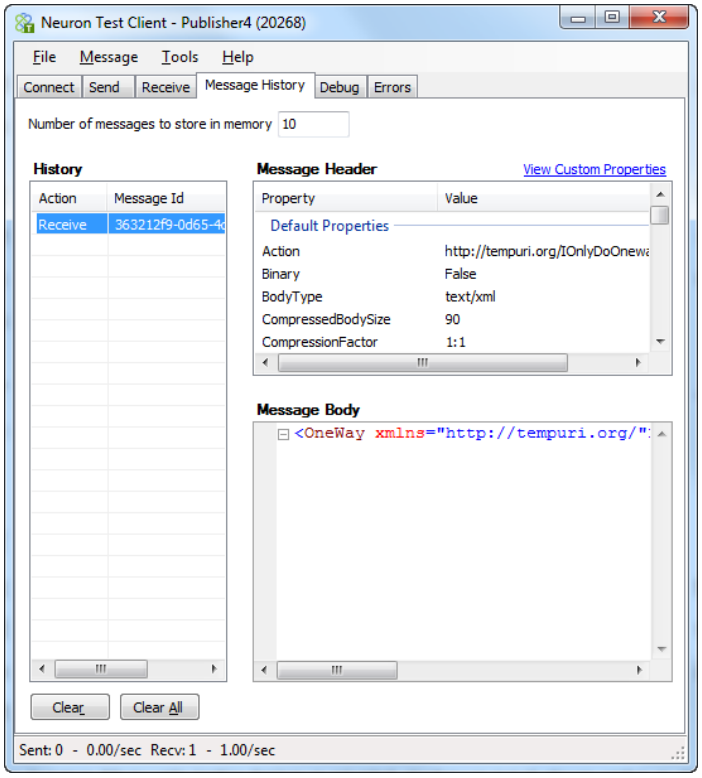


Run the test.

Restore the Test Client and switch to the Receive tab:



Now switch to the Message History tab and view the properties.



By now it should be obvious that crafting a WCF client to send messages to Neuron asynchronously is a straightforward process.

Asynchronous communications are often underutilized in SOA implementations because developers are either unaware of the benefits, are uncomfortable with not receiving a reply or simply used to traditional request-response message patterns that are encouraged by the tools they use to generate services and proxies.

This is unfortunate because designing processes to be asynchronous usually leads to other choices that create a more loosely coupled, scalable architecture.

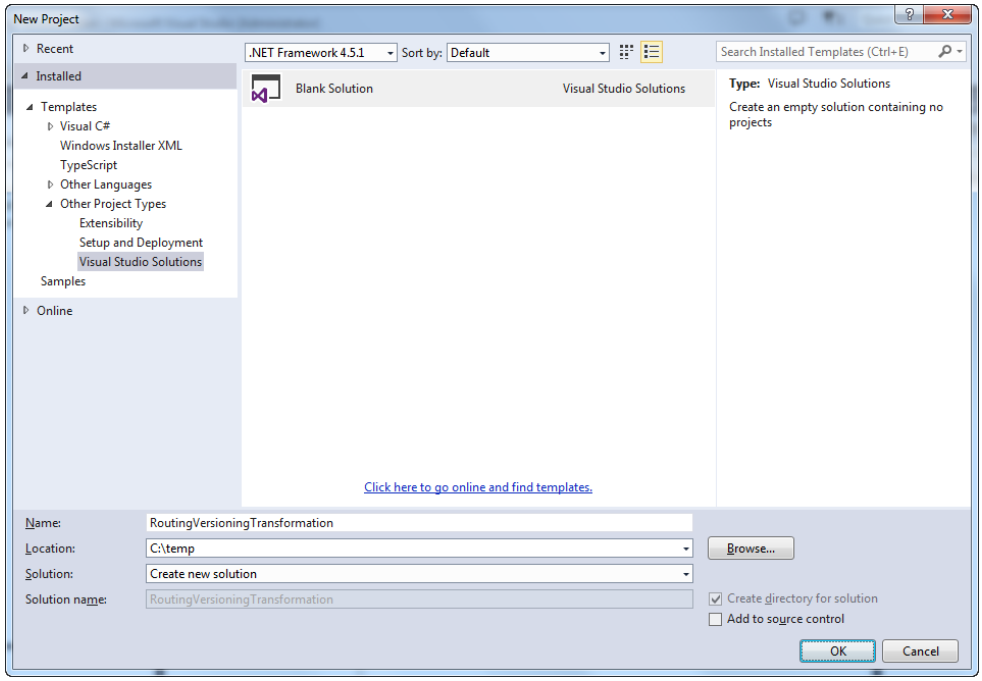
Routing, Versioning and Transformation

At the end of this section of the training you will be equipped with an understanding of Neuron 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.

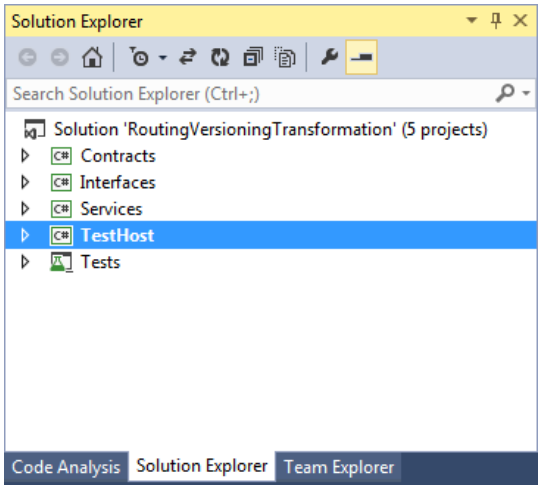
Open Visual Studio and create a new Blank Solution called RoutingVersioningTransformation.



Proceed to do the following in the Blank Solution

1. Create a Class Library project called Contracts
2. Create a Class Library project called Interfaces
3. Create a Class Library project called Services
4. Create a Console Application called TestHost
5. Create a 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:



Add a CodeFile 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
    {
    }
}
```

Add another CodeFile to the project. Rename it ExampleDataContracts.cs.

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.

Switch to the Interfaces Project and insert a CodeFile. 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);
    }
}
```

Add another code 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.

Move onto the Services project. Add a CodeFile and rename it to ExampleMessageContractService.cs.

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();
        }
    }
}
```

Add another CodeFile 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.

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

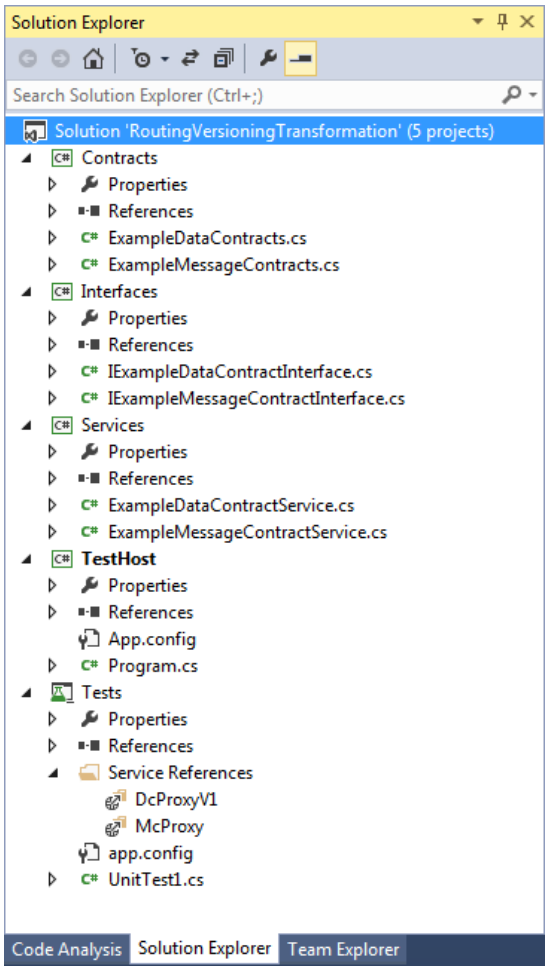
```
using System;
using System.ServiceModel;
using Services;

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();
        }
    }
}
```

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>
```

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.

Shut down the Running TestHost. Your Solution artifacts should now look like the following in Solution Explorer:



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

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

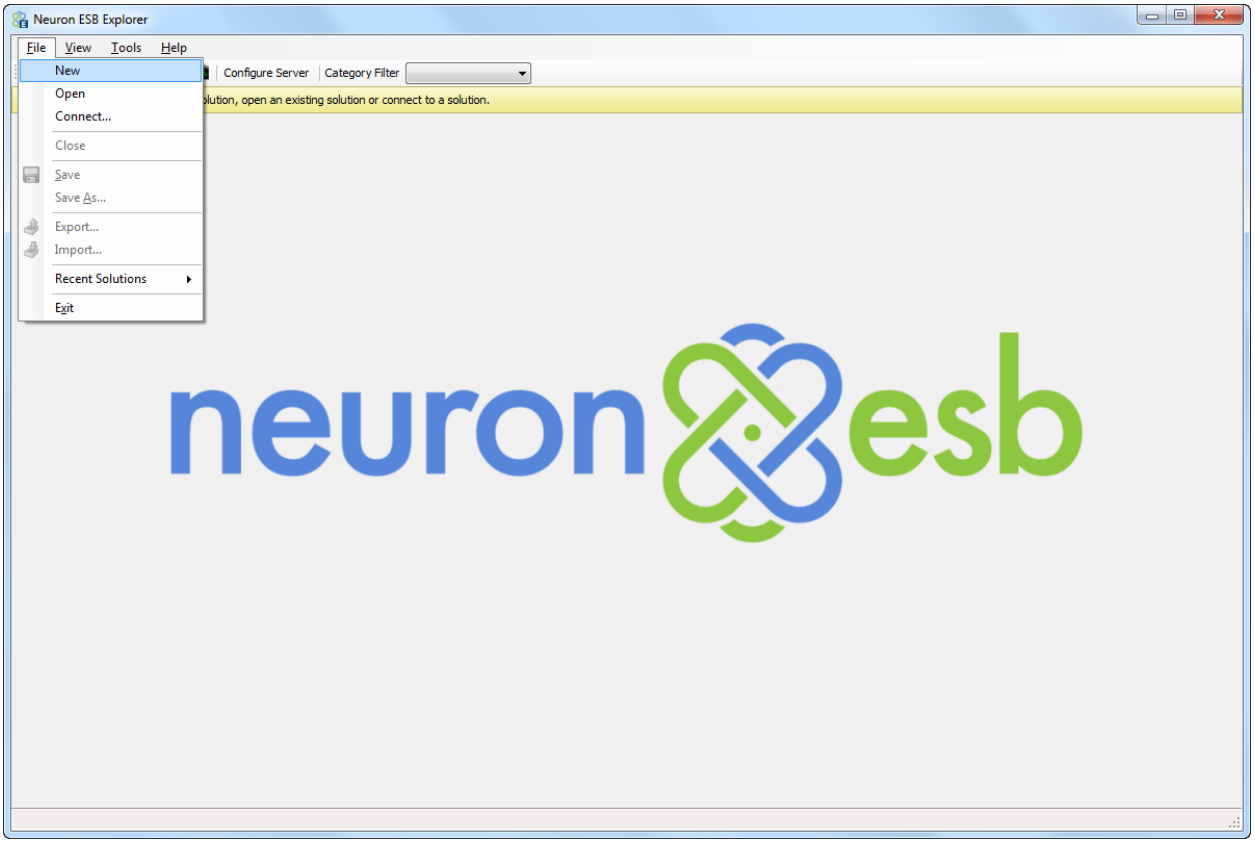
Modify TestMethod1 so it looks as depicted below and add a method named TestMethod2 that contains the code depicted below.

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

[TestMethod]
```

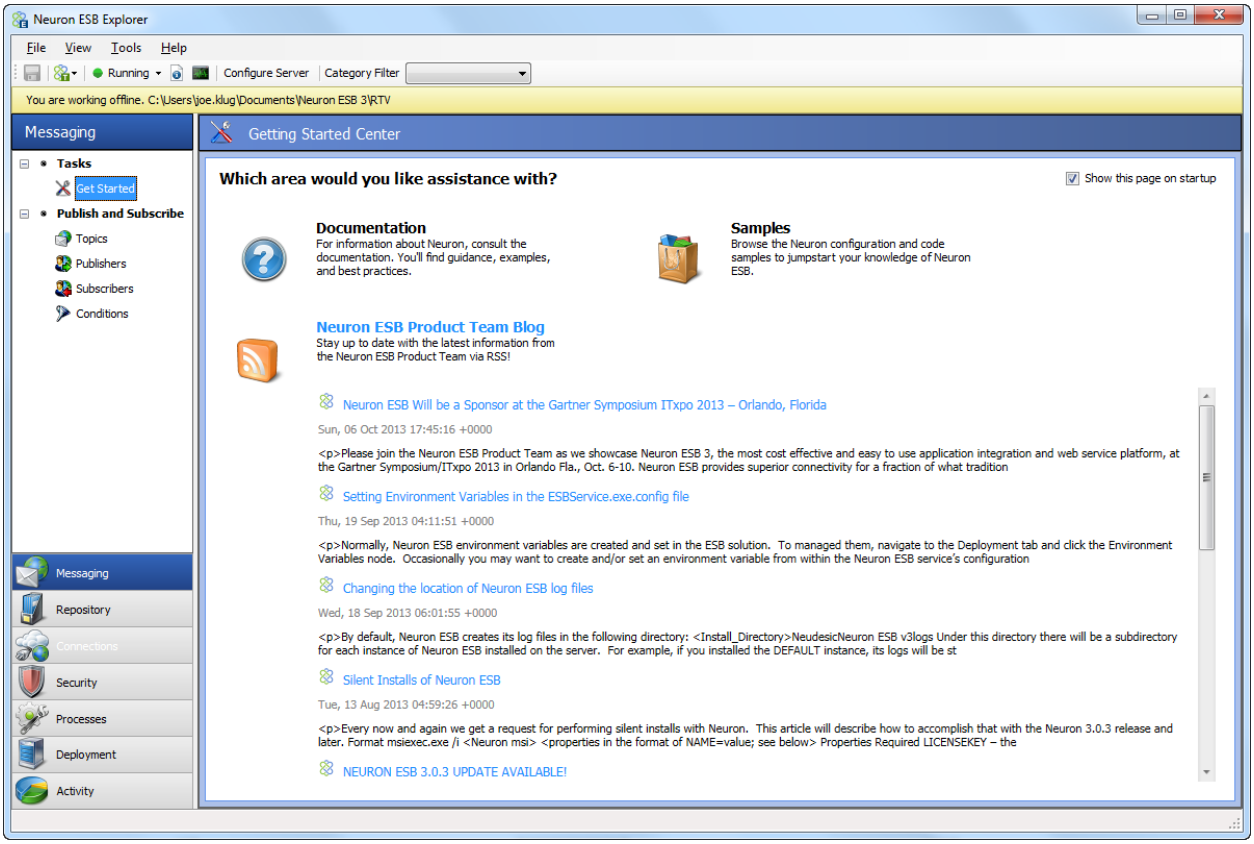
```
public void TestMethod2()
{
      new ExampleDataContractInterfaceClient().DoStuff(new ExampleDcRequest { Argument1 =
"request", Argument2 = 8675309 });
}
```

Press F6 to compile and then press Ctrl-F5 and switch to Test View. Use Test View to run the test methods and verify functionality when calling the services directly.

Leave the TestHost running, minimize the window, open Neuron explorer and create a New configuration:

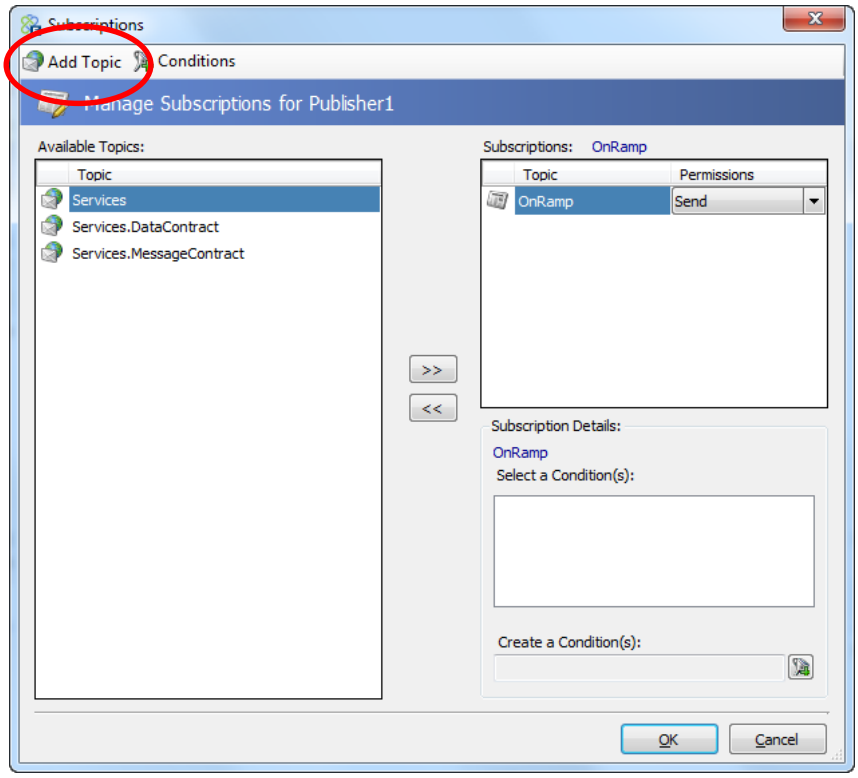


Use the File Menu to save the configuration in the folder RTV

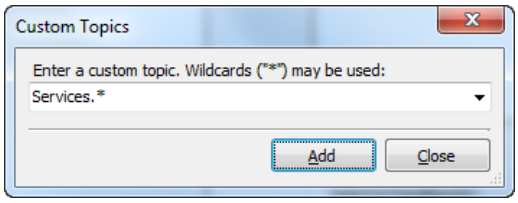


Configure the following:

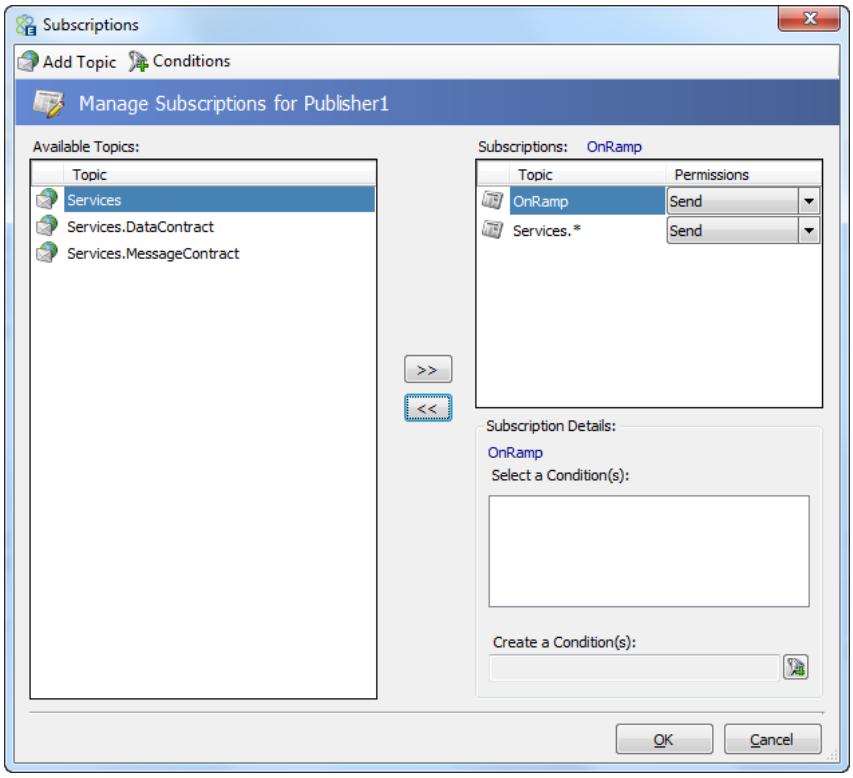
- Add a Topic called OnRamp
- Add a Topic called Services and two Subtopics MessageContract and DataContract
- Add a Publisher called OnRampPublisher with a Send subscription to OnRamp and “Services.*” To create a “.*” subscription, click on the “Add Topic” button on the top of the Subscriptions editor:



Then either select the topic you want to add the wildcard to, or just type in “Services.*” and click the Add button (then click Close):



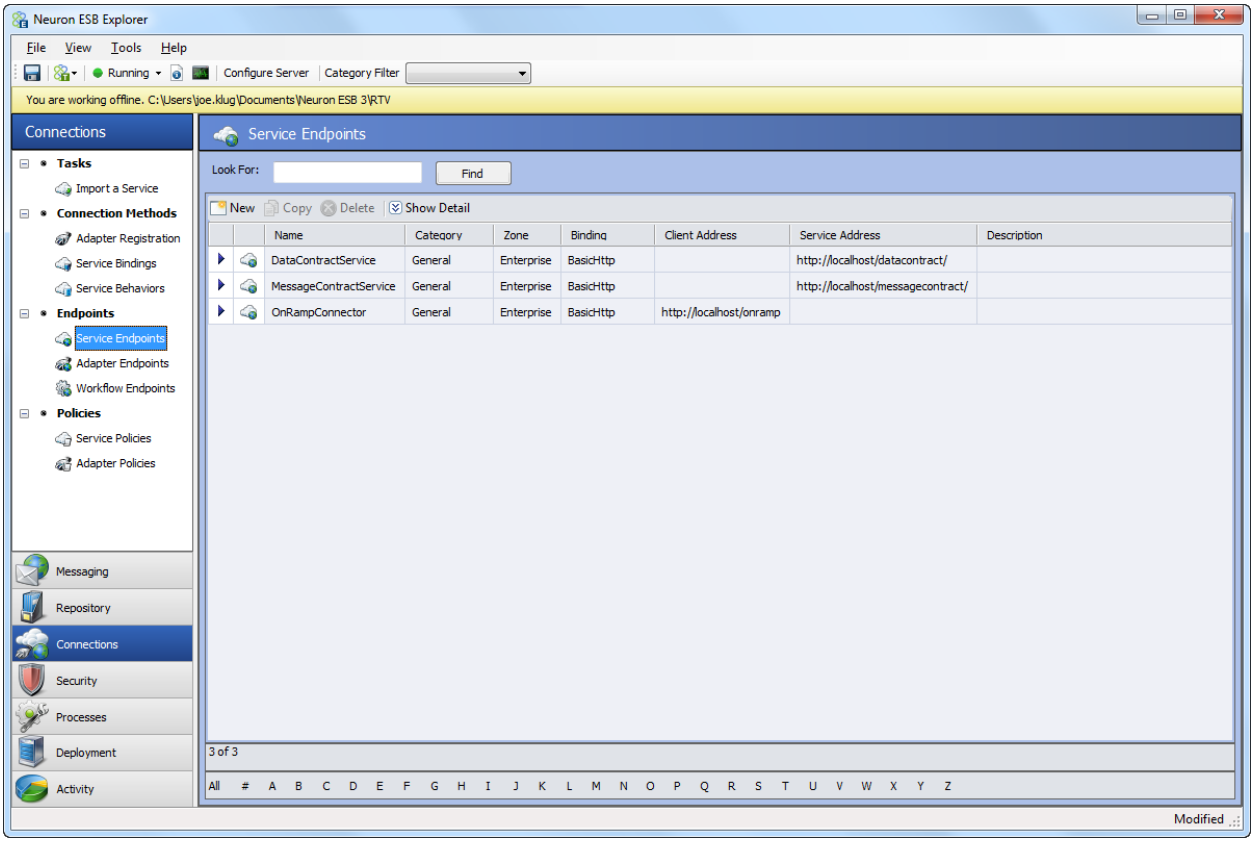
The new wildcard topic will be added to the list of available topics. Select the wildcard topic and add it to the Current Subscription list:



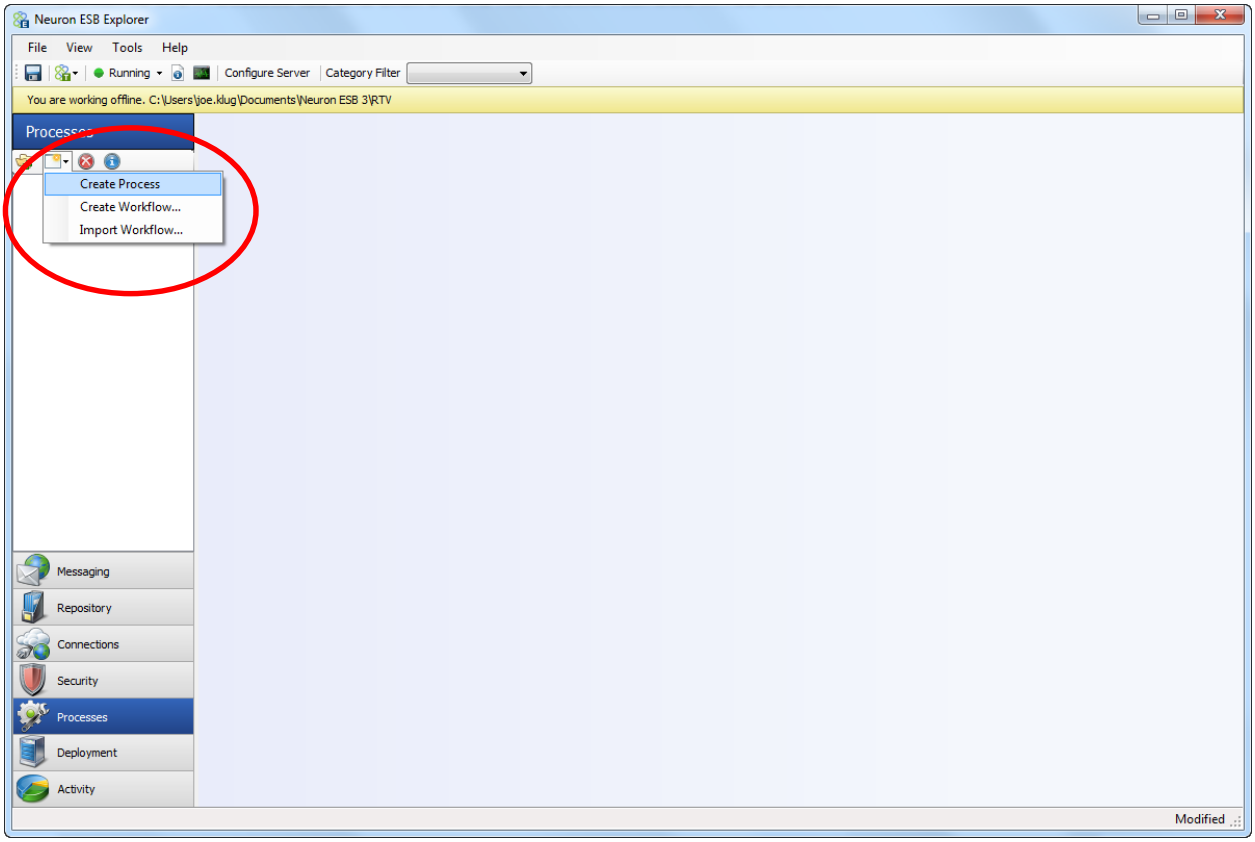
- Add a Subscriber called MessageContractSubscriber with a Receive subscription to Services.MessageContract
- Add a Party called DataContractSubscriber with a Receive subscription to Services.DataContract
- A Client Connector with the following settings:
    - Name: OnRampConnector
    - Binding: BasicHttp
    - Enable Client Connector
    - Publisher Id: OnRampPublisher
    - Topic: OnRamp
    - URL: http://localhost/onramp
- A Service Connector with the following settings (you may import it by using the minimized running TestHost or configure the settings manually):
    - Name: MessageContractService
    - Binding: BasicHttp
    - Enable Service Connector
    - Subscriber Id: MessageContractSubscriber
    - URL: http://localhost/messagecontract/
- A Service Connector with the following settings (you may import it by using the minimized running TestHost or configure the settings manually):
    - Name: DataContractService
    - Binding: BasicHttp
    - Enable Service Connector
    - Subscriber Id: DataContractSubscriber

- URL: http://localhost/datacontract/

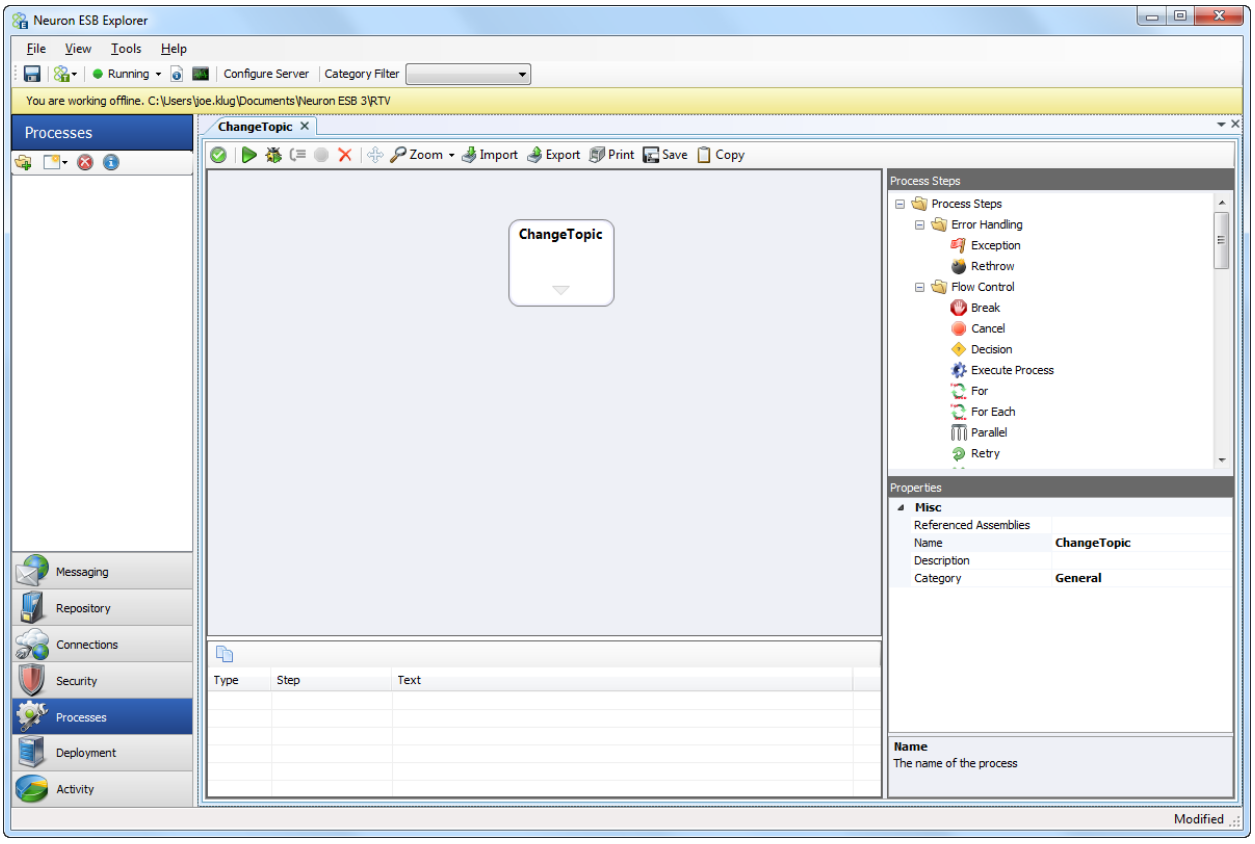
Save the configuration. Neuron explorer should look similar to this:



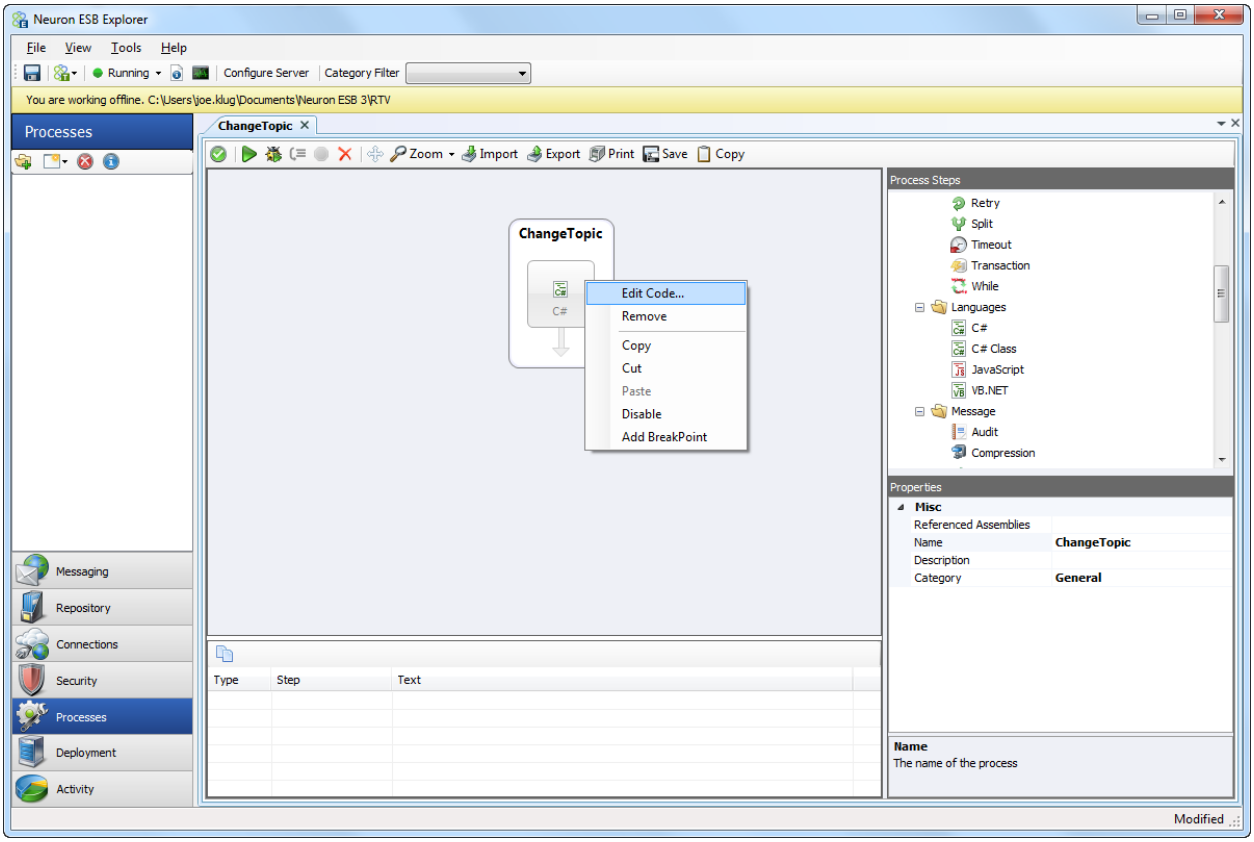
Switch to the Processes tab, click on the New button and select Create Process (circled in red below):



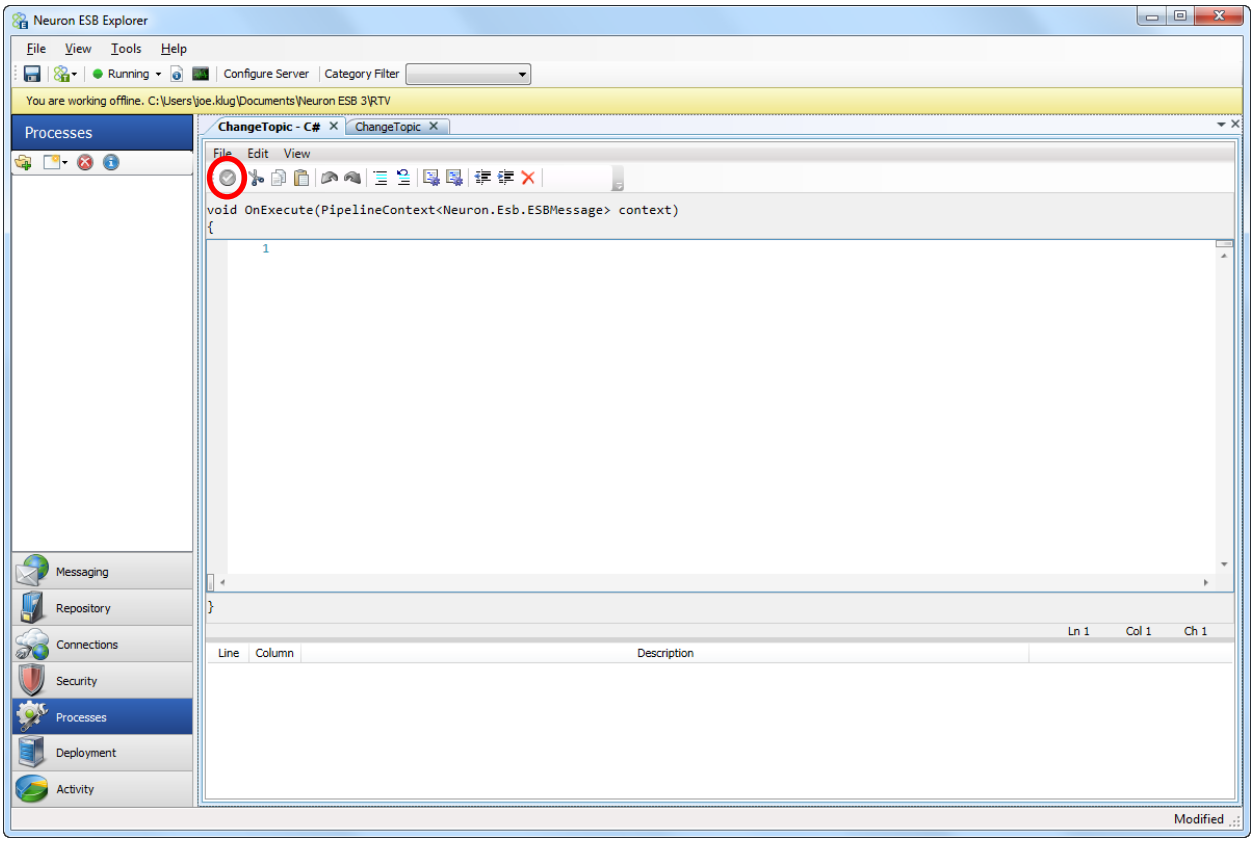
Click the Untitled Process in the designer and change the Name in the Property pane to ChangeTopic:


Neuron ESB Explorer
File
View
Tools
Help
Running
Configure Server
Category Filter
You are working offline. C:\Users\joe.klug\Documents\Neuron ESB 3\RTV
Processes
ChangeTopic
Zoom
Import
Export
Print
Save
Copy
ChangeTopic
Process Steps
Process Steps
Error Handling
Exception
Rethrow
Flow Control
Break
Cancel
Decision
Execute Process
For
For Each
Parallel
Retry
Properties
Misc
Referenced Assemblies
Name
ChangeTopic
Description
Category
General
Type
Step
Text
Name
The name of the process
Messaging
Repository
Connections
Security
Processes
Deployment
Activity
Modified

Drag a C# step onto the process:



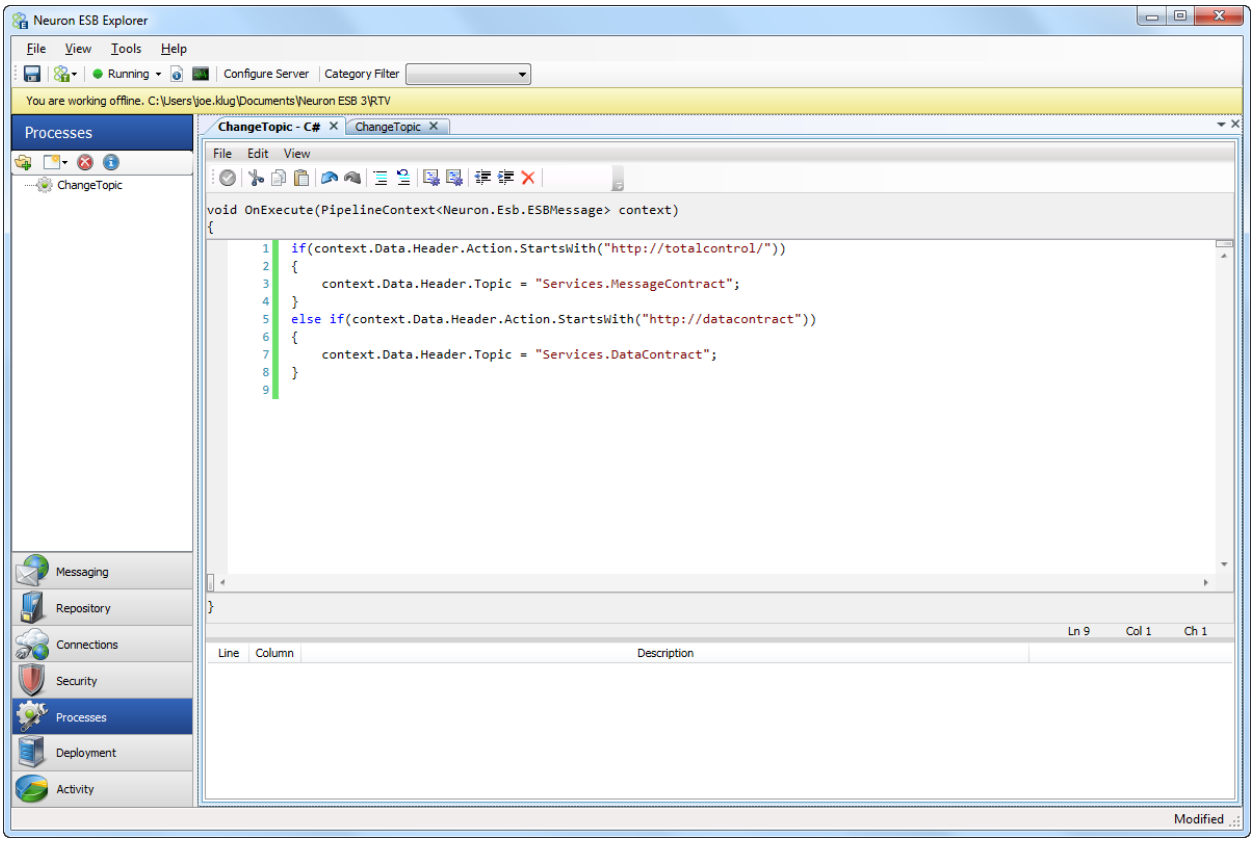
Right click the step and choose Edit Code:



Insert the following code into the Code step

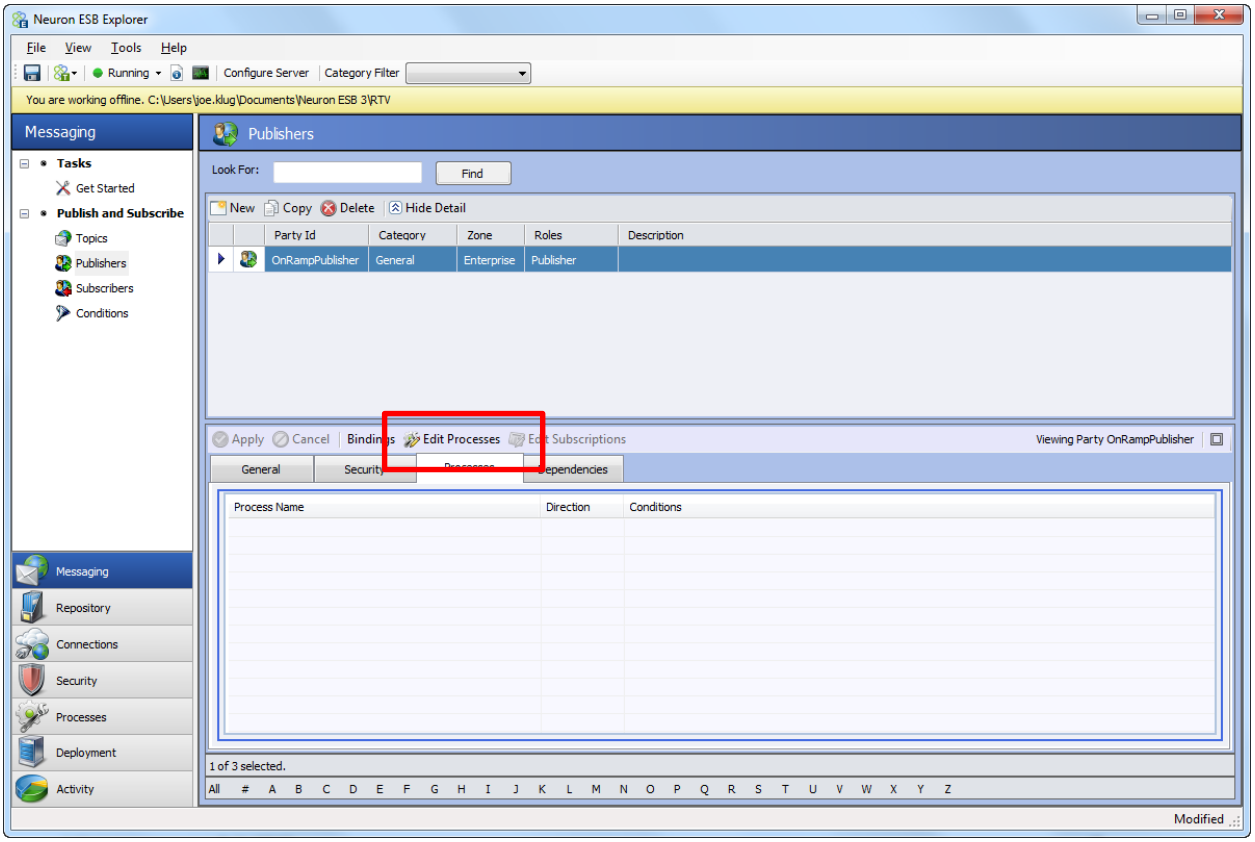
```
if(context.Data.Header.Action.StartsWith("http://totalcontrol/"))
{
        context.Data.Header.Topic = "Services.MessageContract";
}
else if(context.Data.Header.Action.StartsWith("http://datacontract"))
{
        context.Data.Header.Topic = "Services.DataContract";
}
```

Click the Apply button in the Code Step (circled in red above).

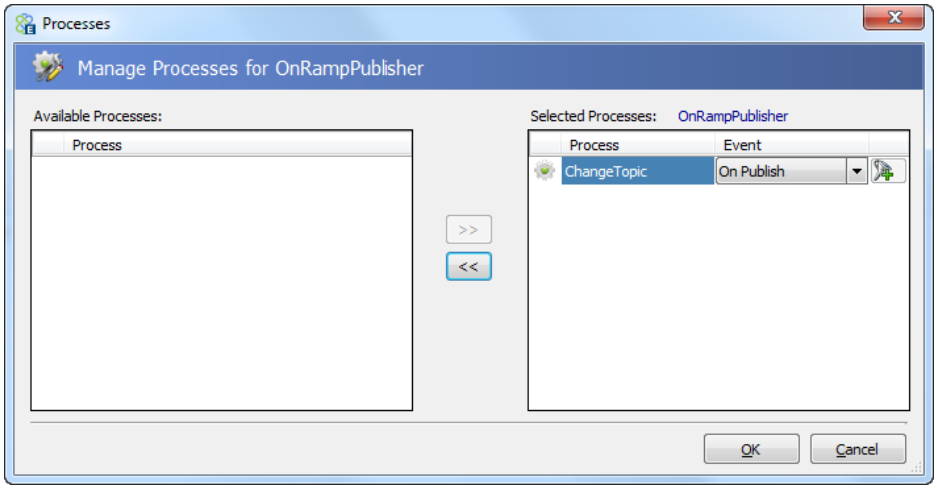


After pressing Apply, you should now see the ChangeTopic Process appear to the left of the Process Designer in the navigation area.

Switch to the Messaging tab, select Publishers, and then select the OnRampPublisher. Click on the Processes tab:

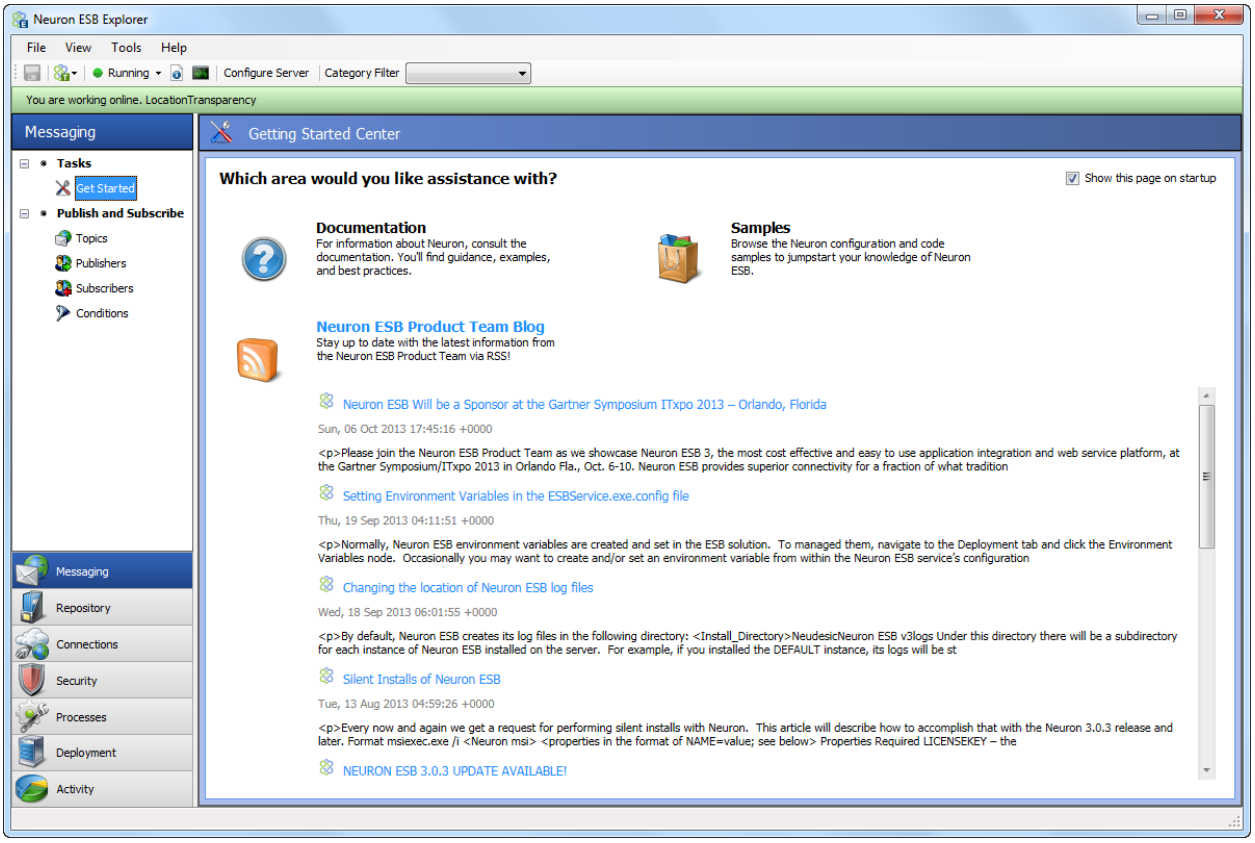


Click the Edit Processes button and add the ChangeTopic Process to the On Publish event:



Press OK on the Manage Processes dialog, Apply the change to the Party and Save the configuration.

Configure Neuron to use the RTV configuration. Restart and use the File Menu to Close and then attach in Connect mode.

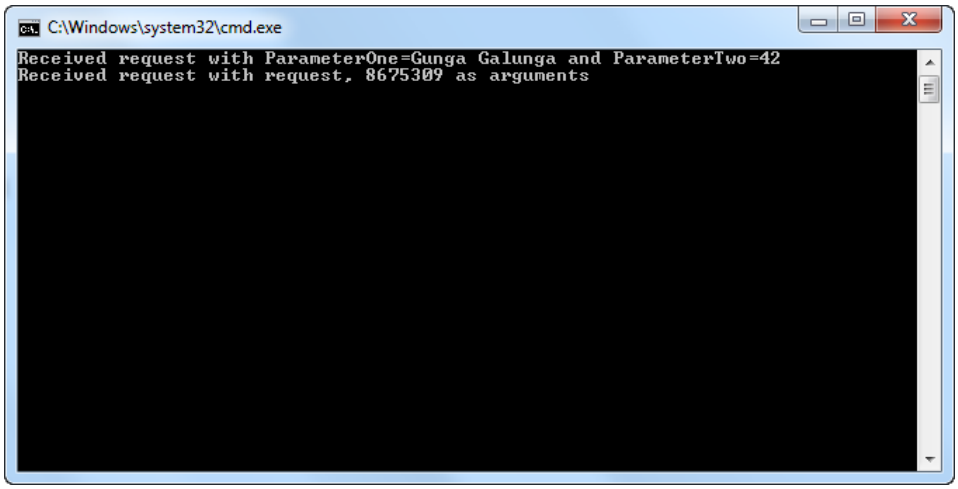


Minimize Neuron Explorer. Restore the TestHost if it is still running and press enter to shut the service down.

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>
```

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



Congratulations! You have just learned how to build an On Ramp in Neuron. For a real production deployment you could use a much more sophisticated dispatching model.

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.

Begin by shutting down the TestHost.

Modify the existing code in ExampleDataContracts.cs by adding highlighted code:

```
[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;
}
```

Modify the code in ExampleDataContractService.cs so that it looks like this

```
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;
}
```

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**.

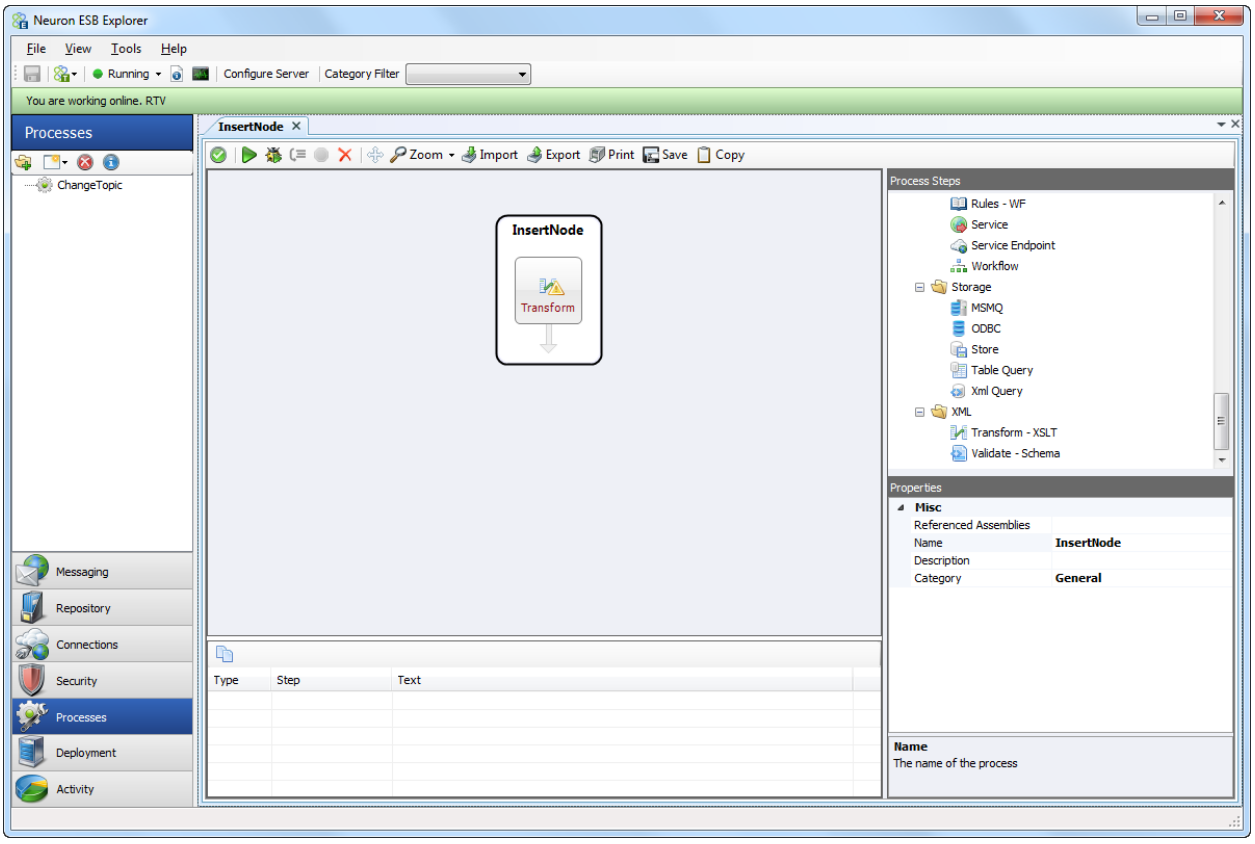
Shutdown the TestHost and add this method to the UnitTest1.cs

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

Change the newly generated address in the Tests App.config to the on ramp:

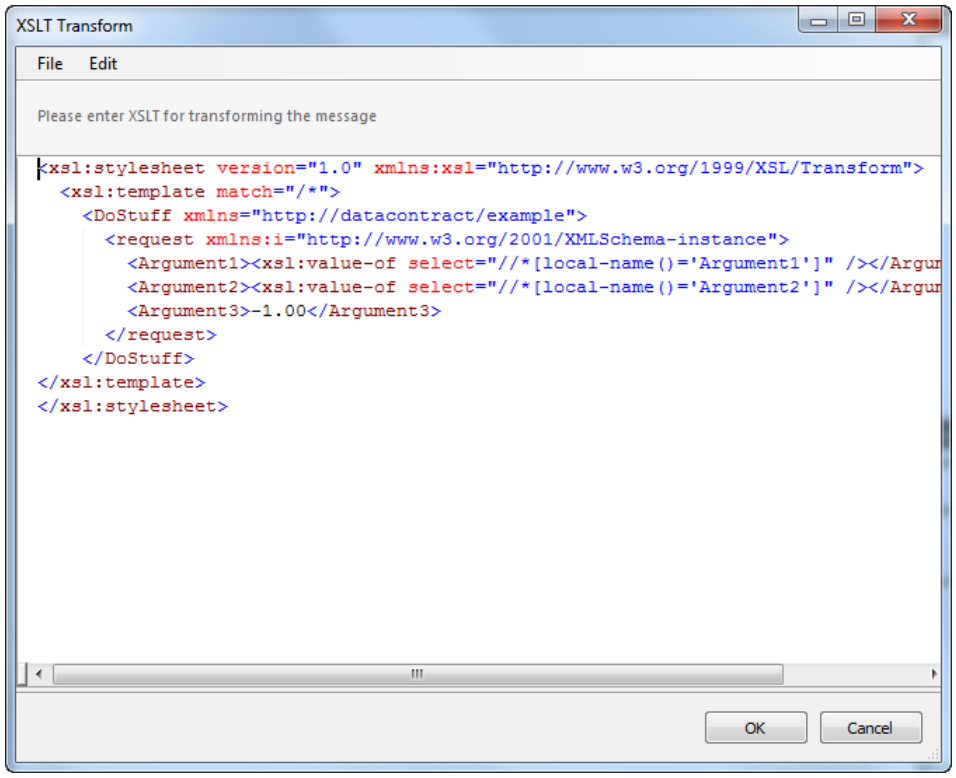
```
<endpoint address="http://localhost/onramp" binding="basicHttpBinding"
    bindingConfiguration="MessageContractService"
    contract="McProxy.IExampleMessageContractInterface"
    name="MessageContractService" />
<endpoint address="http://localhost/onramp/" binding="basicHttpBinding"
    bindingConfiguration="DataContractService"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

Return to Neuron Explorer and click the Processes tab. Click  icon above the ChangeTopic Process to create a new Process. Rename the Process InsertNode and drag a Transform-Xslt Process Step onto the Process designer surface.



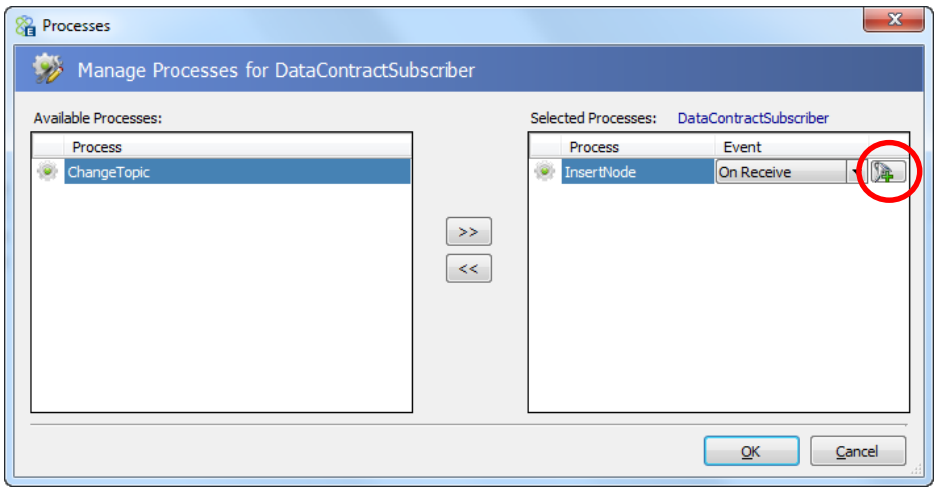
Click the Transform-xslt Step. A property grid will appear. Click the ellipses on the TransformXml property and enter this transform

```
<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>
```

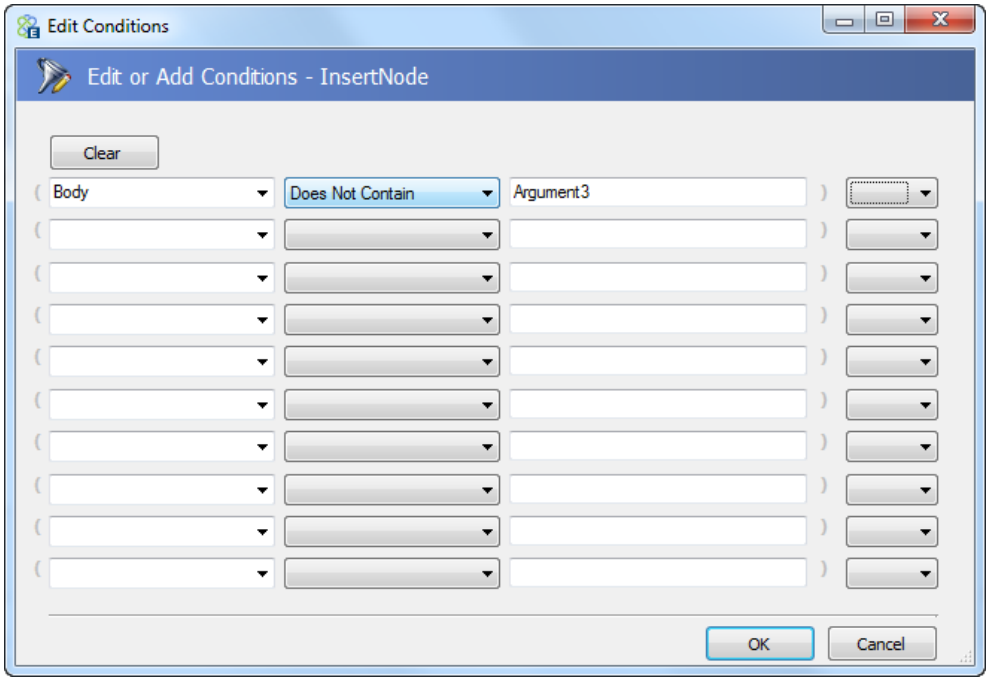


Click Ok on the Transform-Xslt Step and **Apply** the Process then Save the configuration.

Navigate to the Messaging tab and then the DataContractSubscriber Party. Add the InsertNode Process on the On Receive event.



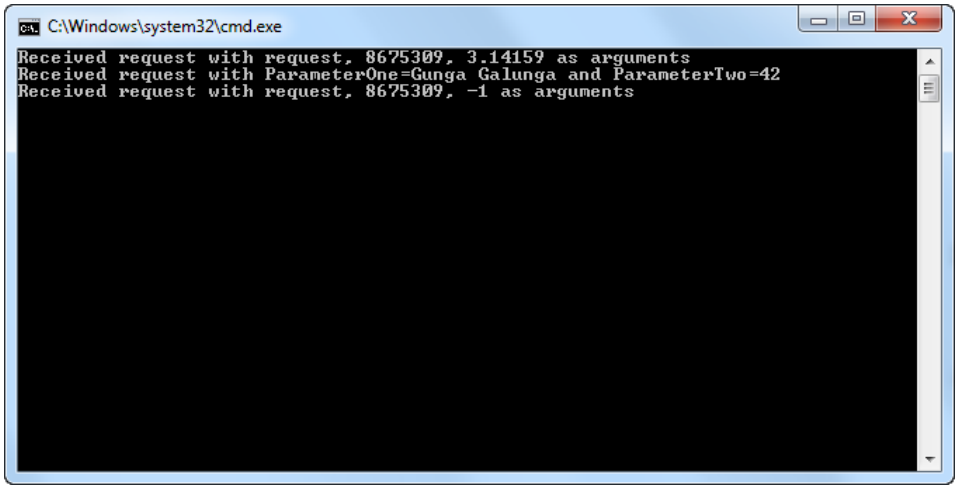
Click the Conditions button next to the Event dropdown (circled in red above) and enter the following:



Click Ok twice and then Apply for the Party. Save the configuration.

Return to Visual Studio. If the TestHost is already running stop it and then press F6 and then Ctrl-F5 to restart.

Switch to Test View and run the tests. Restore the TestHost and observe the output. You should see output similar to this



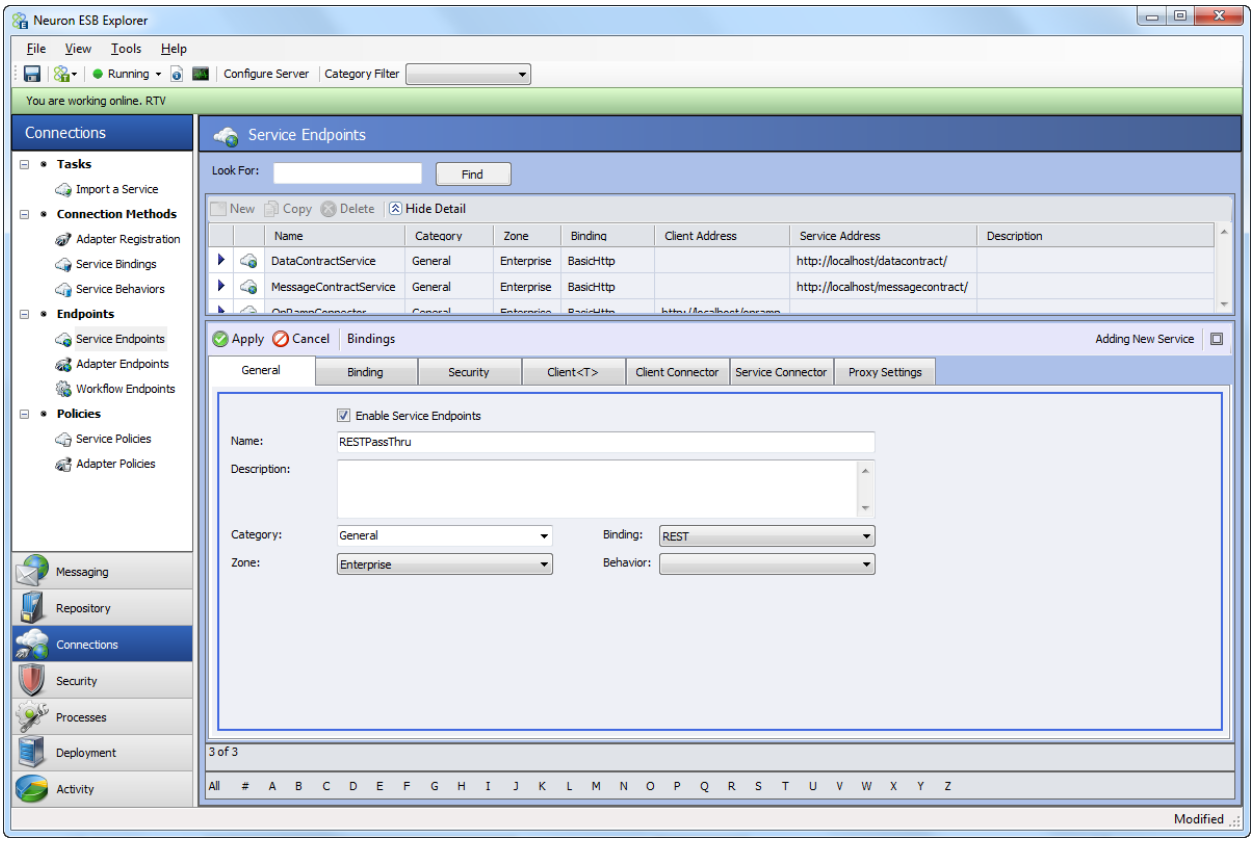
Notice that the process was invoked only when the old client packet was detected.

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

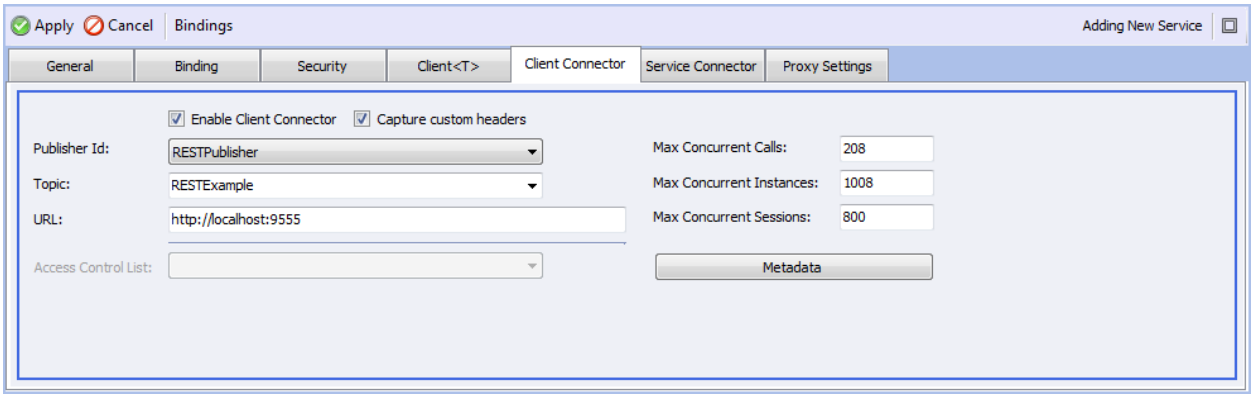
REST

Neuron REST support is straightforward and powerful. Using the same Neuron configuration, add a new Topic named RESTExample. Add a Publisher named RESTPublisher and add a subscription to the RESTExample topic with Send permissions. Add a Subscriber named RESTSubscriber and add a subscription to the RESTExample topic with Receive permissions.

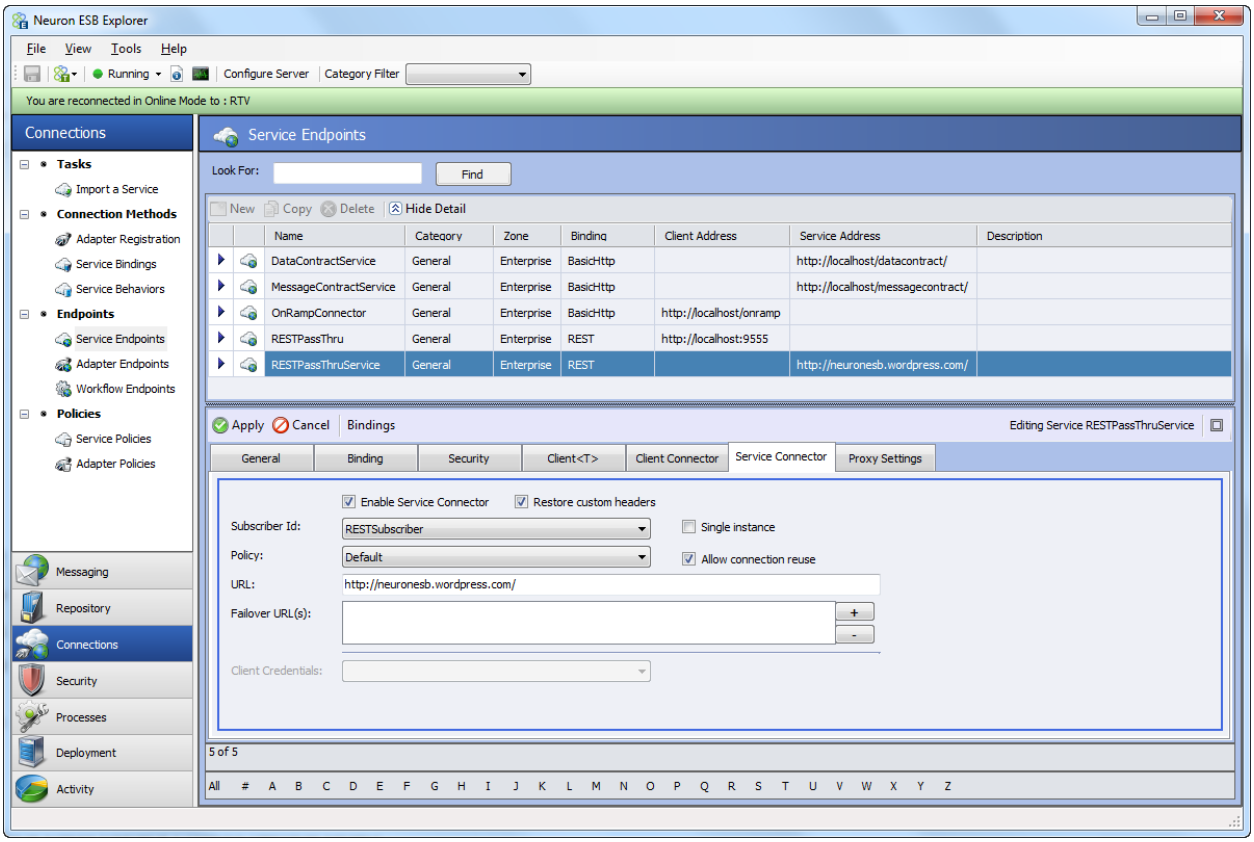
Create a Client Connector and change the Name to RESTPassThru. Change the Binding to REST:



Enable the Client Connector using the RESTPublisher, RESTExample and a URL of http://localhost:9555



Create and enable a separate Service connector named RESTPassThruService using the Party RESTSubscriber and a URL to a search engine or your favorite blog (i.e. http://neuronesb.wordpress.com/).



Save the configuration. Browse to http://localhost:9555 and you will see the website configured on the Service Connector URL. This is fun but in production you are probably going to be using actual RESTful API's. The good news is from a Neuron standpoint it is essentially the same process.

Service Façade

The final walkthrough in this training will cover using Neuron as a Service Façade. This type of pattern is often used in classic SOA approaches where a canonical model is exposed that in reality is composed of information from various sources.

This walkthrough demonstrates a few key Web Services features in business processes:

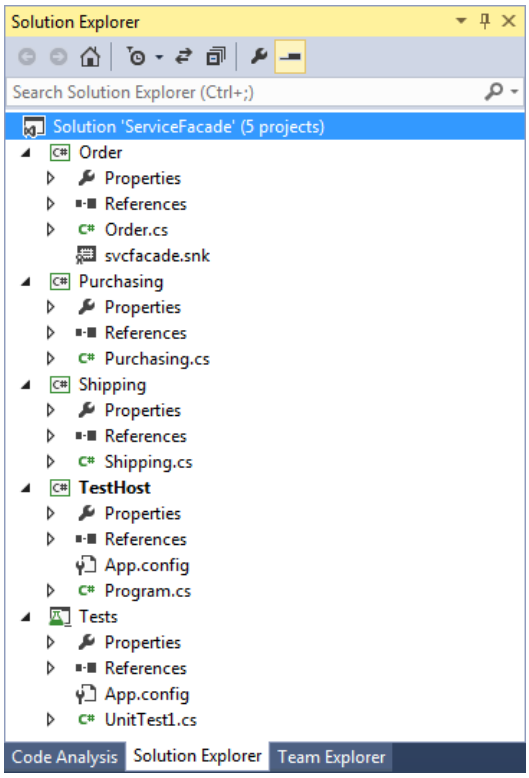
- Service Endpoint Step – This step allows users to call any configured Service Endpoint (i.e. Service Connector) directly, without the need to publish the message to a Topic. This works with either Request/Response or Multicast/Datagram types of messaging patterns.
- Using the Cancel step to send a reply message back to the original sender – Typically the Cancel step will just stop a business process. But when a cancel step is used in a process that is called on the On Publish event, and the message being processed has a request semantic, then the Cancel will automatically return the value of context.Data.

Create a new Visual studio Solution named ServiceFacade.

Complete the following steps:

- Add a Class Library Project named Order
- Sign the project with a new key named svcfacade (we will be GAC-ing this assembly)
- Add a Class Library Project named Purchasing
- Add a Class Library Project named Shipping
- Add a Console Application named TestHost
- Add a Test Application named Tests
- Delete the Class1.cs generated files
- Add a reference to System.ServiceModel and System.Runtime.Serialization to all of the projects. We will not be generating proxies this time.
- Add a Class to OrderProject named Order.cs
- Add a Class to the Purchasing project named Purchasing.cs
- Add a Class to the Shipping project named Shipping.cs
- In the TestHost project, add references to the Purchasing and Shipping projects
- If not already present, add an App.config file to the TestHost project
- In the Tests project, add a reference to the Order project
- If not already present, add an App.config to the Tests project
- Set the TestHost project as the Startup project

Your solution should now look something like this:



Add the following code to the Order.cs file.

```
using System.ServiceModel;
using System.Xml.Serialization;
namespace Order
{
    [MessageContract(WrapperNamespace="http://facade/orders")]
    public class OrderRequest
    {
        [MessageBodyMember(Namespace="http://facade/orders")]
        public string ItemCode;

        [MessageBodyMember(Namespace = "http://facade/orders")]
        public int Qty;
    }

    [MessageContract(WrapperNamespace = "http://facade/orders")]
    [XmlRoot(Namespace="http://facade/orders")]
    public class OrderResponse
    {
        [MessageBodyMember(Namespace = "http://facade/orders")]
        public string TrackingCode;
    }

    [ServiceContract]
    public interface IOrder
    {
        [OperationContract(ReplyAction="*")]
        OrderResponse PlaceOrder(OrderRequest request);
    }
}
```

Compile the project and add the assembly to the GAC.

Add the following code to the Purchasing.cs file

```
using System;
using System.ServiceModel;
namespace Purchasing
{

    [ServiceContract]
    public interface IPurchase
    {
        [OperationContract]
        int Purchase(string itemCode, double qty);
    }

    public class PurchasingService : IPurchase
    {
        public int Purchase(string itemCode, double qty)
        {
            return new Random(itemCode.GetHashCode()).Next();
        }
    }

}
```

Add the following code to the Shipping.cs file

```
using System;
using System.ServiceModel;
namespace Shipping
{
    [ServiceContract]
    public interface IShip
    {
        [OperationContract]
        string ShipOrder(int po);
    }

    public class ShippingService :IShip
    {

        public string ShipOrder(int po)
        {
            return DateTime.Now.Ticks.ToString("X");
        }
    }
}
```

Add the following using statements to the Program.cs file in the TestHost project

```
using System.ServiceModel;
using Purchasing;
using Shipping;
```

Modify the Program.cs file in the TestHost project so the body of the main method is as follows

```
static void Main(string[] args)
{
    var mchost = new ServiceHost(typeof(PurchasingService));
    var dchost = new ServiceHost(typeof(ShippingService));
    mchost.Open();
    dchost.Open();
    Console.ReadLine();
    mchost.Close();
    dchost.Close();
}
```

Add the following entry to the TestHost App.config file 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="Purchasing.PurchasingService">
      <endpoint address="" binding="basicHttpBinding"
          bindingConfiguration="" name="PurchasingService"
          contract="Purchasing.IPurchase" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost/purchasing/" />
        </baseAddresses>
      </host>
    </service>
    <service behaviorConfiguration="TestHostBehavior" name="Shipping.ShippingService">
      <endpoint address="" binding="basicHttpBinding"
          bindingConfiguration="" name="ShippingService"
           contract="Shipping.IShip" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost/shipping/" />
        </baseAddresses>
      </host>
    </service>
  </services>
</system.serviceModel>
```

Add the following using statements to UnitTest1.cs file

```
using Order;
using System.ServiceModel;
```

Update the TestMethod1 :

```
[TestMethod]
public void TestMethod1()
{
    var fac = new ChannelFactory<IOrder>("order");
    var prox = fac.CreateChannel();
    var response = prox.PlaceOrder(new OrderRequest { ItemCode = "i81u812", Qty = 42 });
    Console.WriteLine(response.TrackingCode);
}
```

Open the Tests App.config file and add a client endpoint for the façade:

```
<system.serviceModel>
    <client>
        <endpoint address="http://localhost/facade" binding="basicHttpBinding"
                  bindingConfiguration="" contract="Order.IOrder" name="order" />
    </client>
</system.serviceModel>
```

Compile the solution and start the TestHost (Ctrl-F5).

The Order assembly needs to be put into the Global Assembly Cache:

1. Open the Visual Studio Developer Command Prompt
2. Change directory to the location of the Order assembly

3. Execute the command:

```
gacutil /if Order.dll
```

Restore or start Neuron Explorer. Do the following steps:

- Create a new configuration
- Save the configuration as Façade
- Configure the Neuron ESB Service to run the Façade configuration
- Restart the Neuron ESB Service
- Close and connect in Online mode
- Add a Topic named Façade
- Add a Publisher named FacadePublisher that subscribes to Façade
- Add a Subscriber named FacadeSubscriber that subscribes to Façade
- Add a Client Connector and rename it OrdersClientConnector. Add the following settings
    - Binding: BasicHttp
    - Publisher Id: FacadePublisher
    - Topic: Façade
    - URL: http://localhost/facade
    - Enable Client Connector
- Add a Service Connector and rename it PurchasingServiceConnector. Add the following settings
    - Binding BasicHttp
    - Subscriber Id: FacadeSubscriber
    - URL: http://localhost/purchasing
    - Enable Service Connector
- Add a Service Connector and rename it ShippingServiceConnector. Add the following settings
    - Binding BasicHttp
    - Subscriber Id: FacadeSubscriber
    - URL: http://localhost/shipping
    - Enable Service Connector
- Apply all changes and Save the configuration
- Create a new Process and rename it OrderProcess

Drag a Transform shape onto the Process Designer and add the following transform

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/*">
    <Purchase xmlns="http://tempuri.org/">
      <itemCode>
        <xsl:value-of select="//*[local-name()='ItemCode']" />
      </itemCode>
      <qty>
        <xsl:value-of select="//*[local-name()='Qty']" />
      </qty>
    </Purchase>
  </xsl:template>
</xsl:stylesheet>
```

Drag a C# Step onto the Process designer. Right-click on the code step and select Edit Code…

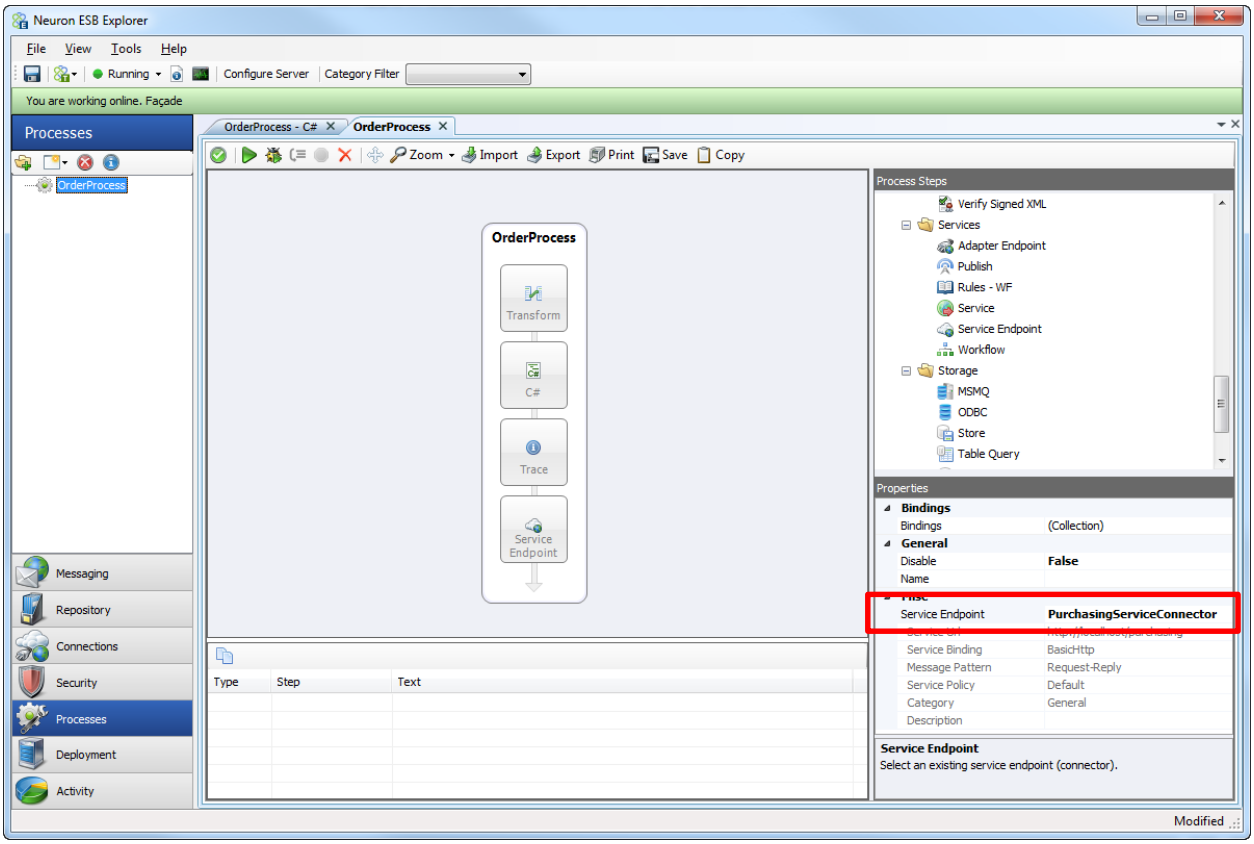
Add the following code to the code shape:

```
context.Data.Header.Action = "http://tempuri.org/IPurchase/Purchase";
```

Click Apply in the Code Editor and return to the designer.

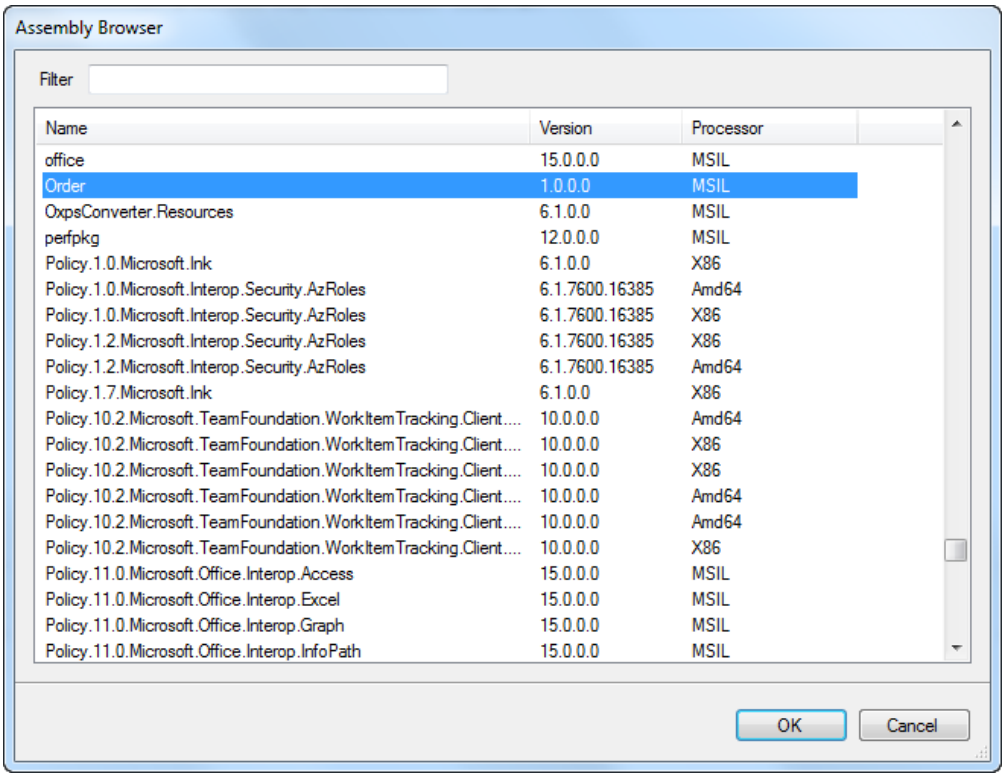
Drag a Trace Step onto the designer. This Step has no properties. We will use it when we test our process.

Drag a Service Endpoint Step onto the designer. Select the Service Endpoint Step and set the Service Endpoint property to PurchasingServiceConnector:



Drag another Trace Step onto the Process designer after the Service Step.

Drag a C# Step onto the Process designer. Right-click on the code step and select Edit Code… Use the File > Manage Referenced Assemblies… toolbar selection to bring up the References dialog and then click the Browse button to bring up the Browse dialog. Select your Order assembly that you previously placed in the GAC.



Click OK and OK again to add the reference. Add the following code to the code shape:

```
Order.OrderResponse response = new Order.OrderResponse();
context.Properties.Add("response",response);
```

Saving our response at this point is technically not required for this scenario however in many scenarios the response has multiple elements that need to be saved along the various steps so knowing how to keep state in memory and restore that state as you progress along the Process is a valuable technique to learn.

Click Apply in the Code Editor and Save the configuration.

Drag a Transform-Xslt Step onto the Process designer now and add the following XSLT to the Step

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/*">
    <ShipOrder xmlns="http://tempuri.org/">
      <po><xsl:value-of select="//*[local-name()='PurchaseResult']" /></po>
    </ShipOrder>
  </xsl:template>
</xsl:stylesheet>
```

Drag another Trace Step onto the Process Designer after the Transform-Xslt Step.

Drag another C# Step onto the Process designer. Right-click on the code step and select Edit Code…

Add the following code to the code shape:

```
context.Data.Header.Action = "http://tempuri.org/IShip/ShipOrder";
```

Click Apply in the Code Editor and return to the designer.

Drag a Service Endpoint Step onto the designer. Select the Service Endpoint Step and set the Service Endpoint property to ShippingServiceConnector.

Drag another Trace Step onto the Process Designer after the Service Step.

Drag a C# Step onto the Process Designer and use the File>Manage Referenced Assemblies dialog to add a reference to Order.dll for this Step. References are stored per Step so you have to do this again even though you added it to a Code Step previously.

After you have added the reference, insert the following code into the Code Step and then click Apply:

```
Order.OrderResponse response = context.Properties["response"] as Order.OrderResponse;
response.TrackingCode =
context.Data.ToXmlDocument().DocumentElement["ShipOrderResult"].InnerText;
context.Data = new Neuron.Esb.ESBMessage(context.Data.Header.Topic,response);
```

Last but definitely not least, **Drag a Cancel step to the end of the Process**. Apply the changes to the process.

The process should now contain this flow (you may have to use the pan tool and adjust the trace window if you want to see all the steps as depicted):

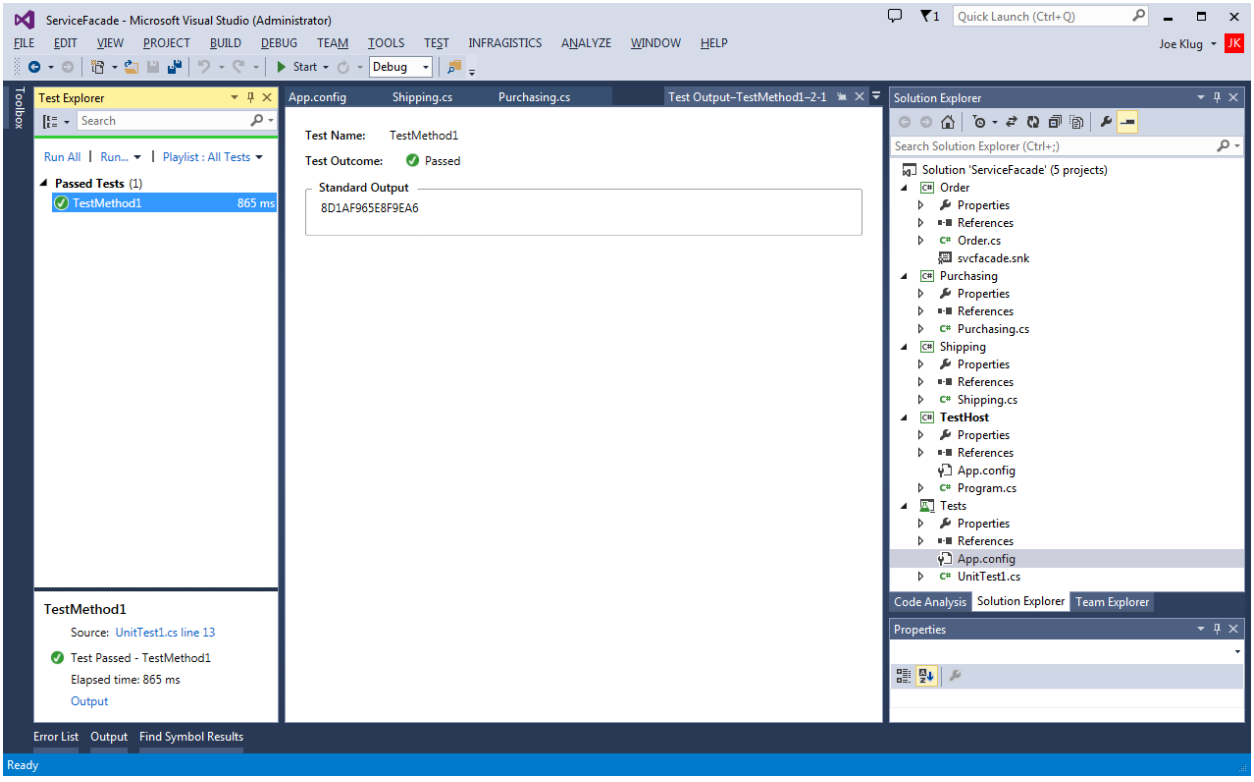

OrderProcess
Transform
C#
Trace
Service Endpoint
Trace
C#
Transform
Trace
C#
Service Endpoint
Trace
C#
Cancel

The Cancel Step is a key component here. The Neuron API Send(..) method is what is used to publish. That method always returns an ESBMessage. If it's a Publish with Request Semantic you will get the response from the Subscriber. When you put a Cancel Step at the end of a Process what you are saying is run all the previous logic and then cancel sending the message to the publishing system. This has the net effect of returning the last value of context.Data to the Send method!

Think of the Cancel acting more like a "Return".

Now associate the OrderProcess process to the Party FacadePublisher, on the On Publish event. Apply the changes to the Party and Save.

Return to Visual Studio. The TestHost should still be running. If it is not, press Ctrl-F5. Switch to Test View and run TestMethod1.



Congratulations yet again! You have completed this training.

Review

Neuron is a powerful Services Intermediary that uses Service Endpoints called Client Connectors to allow Web Service Clients to communicate with the bus. Neuron communicates with services using a Service Connector.

Client Connectors and Service Connectors are not aware of each other. Each Client Connector and Service Connector is hosted in its own app domain.

Client Connectors and Service Connectors both use an embedded Party. When you enable a Party in a Connector, you are using Neuron's pub sub API to publish and subscribe to SOAP and REST requests.

Processes attached to the Party in Client Connectors and Service Connectors can be used to accomplish many tasks common to SOA environments such as Location Transparency, Routing, Versioning and Transformation.

A Client Connector can also be used with a Process that uses the Cancel trick to create a real time message processing environment or hosted service that bypasses the publishing system. This hosted service can utilize any logic you choose including making multiple web service calls.

Exercises

1. Create a Request-Reply OnRamp that accepts messages on a single Client Connector but can send to multiple Service Connectors based on SOAP Action.
2. True or false: Neuron Client Connectors function as a Service Proxy with an embedded Neuron Party?
3. True or false: Neuron produces WSDL and defaults to strongly typed messaging?
4. A Client Connector with a Datagram Messaging Pattern will publish the message as
    a. Request
    b. Multicast
    c. Oneway
5. Create a façade that calls two web services and returns the combined responses. Use the message class so you can dynamically change the client request and response. Use the Publish Step with a Request Semantic instead of the Service Step.
6. True or false: Neuron Client Connectors cannot be secured with Windows Groups?
7. Processes that manipulate a Web Service Response will work as expected if attached to:
    a. the Client Connector Party's On Client Publish
    b. the Service Connector Party's On Client Receive
    c. the Service Connector Party's On Client Publish
8. Accomplish the following:
    1. Import a Service
    2. Set logging to verbose.
    3. In the Visual Studio project there is a file called Arguments for Test Client. Use these value to call the Web Service using a Test Client
9. Which of the following statements are false?
    a. Neuron supports custom WCF configurations.
    b. Neuron supports REST.
    c. Neuron Client Connectors and Service Connectors are linked in memory at runtime.
    d. Neuron cannot use a schema to validate incoming messages.
10. Which of the following statements is false
    a. A Service Connector with Request-Reply set for its Messaging Pattern will ignore Multicast Semantic

b. A Service Connector with Request-Reply set for its Messaging Pattern will ignore Request Semantic
c. A Process attached to the Service Connector's On Client Receive can be used to alter the Semantic thus affecting whether or not the message is delivered to the target service
d. A Process attached to the Service Connector's On Client Publish can be used to alter the Semantic thus affecting whether or not the message is delivered to the target service

11. True or False: A Cancel step can be used to send a reply back to a Web client?
12. True or False: A Service Endpoint step allows you to call a configured Service Connector directly from with within a business process, bypassing the bus infrastructure entirely?

Appendix

The following Artifacts accompany this training:

- Neuron Web Services Answers.docx-Word Document with answers to Exercises that do not require coding
- Neuron Web Services Projects- Directory with solutions to Exercises 1, 5 and 8