WCF Routing Service - Part I: Basic Concept, Simple Routing Service & Content-based Routing

Samir NIGAM, 7 Jun 2014 CPOL

★★★★★ 4.92 (36 votes)

This article describes WCF Routing Service concept, Configuring RoutingService (its endpoint(s), target service(s), message filter(s) and filter table) and content based routing.

Download the sample code - 292.6 KB

Table of Contents

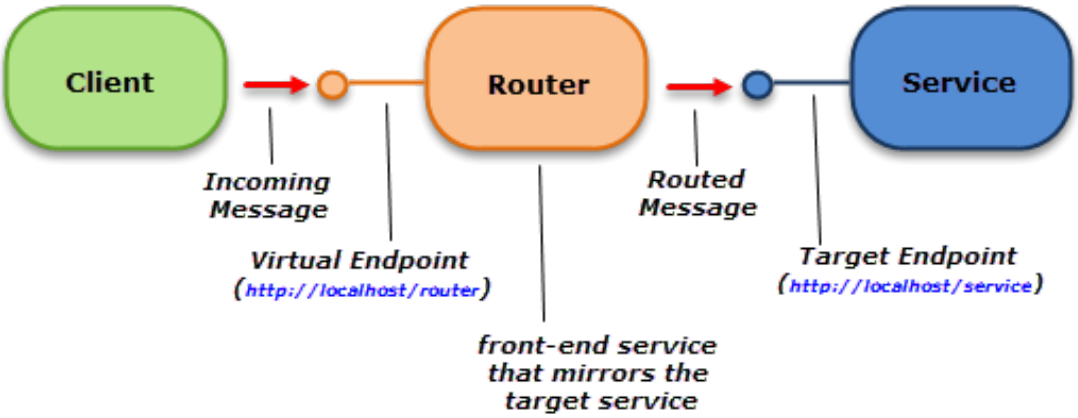
All Posts

Introduction

The **Routing Service** is a generic **SOAP** intermediary that acts as a router. The core functionality of the **Routing Service** is the ability to route incoming messages based on message content (in either the header or the message body) to the actual services hosted in the same machine as the **Router Service** or distributed across the network. Actually **Routing Service** acts as a ***front-end service*** that mirrors the target service(s). The main benefit of the **Routing Service** is to provide location transparency to the client (application) because the client is explicitly decoupled from knowing anything about the actual services that will actually perform tasks on its behalf. Hence it makes possible to perform a variety of different types of intermediate processing within the routing service.

You can use routing in a number of ways. e.g. you can use routing to route incoming messages to the appropriate service(s) by using content-based/context-based routing techniques. You can also use routing to implement a centralized security boundary, protocol bridging, load-balancing or even service versioning.

In routing, **Routing Service** (Router) exposes a virtual endpoint(s) that client application(s) consumes instead of consuming the actual service endpoint(s) and that (virtual endpoint) routes incoming messages from the client application to the appropriate actual service endpoint through an intermediary.



Before **WCF 4.0**, there was not official support of Routing Service in the framework but from **WCF 4.0** onwards there is built-in support of the same.

Understanding the Routing Service

WCF 4.0 came with a new class called `RoutingService` that provides a generic**WCF** routing implementation. The `RoutingService` class can handle routing messages over any **WCF-supported** protocol using a variety of different messaging patterns like one-way, request-response, and duplex messaging. This class is located underneath the `System.ServiceModel.Routing` namespace and you'll need to add reference of **System.ServiceModel.Routing.dll** assembly in your service hosting application.

Below is the definition of the `RoutingService` class (from msdn)

```
[AspNetCompatibilityRequirementsAttribute(RequirementsMode =
                                          AspNetCompatibilityRequirementsMode.Allowed)]
[ServiceBehaviorAttribute(AddressFilterMode = AddressFilterMode.Any, InstanceContextMode
                        = InstanceContextMode.PerSession, UseSynchronizationContext = false,
                                                  ValidateMustUnderstand = false)]
public sealed class RoutingService : ISimplexDatagramRouter, ISimplexSessionRouter,
```

```
                                        IRequestReplyRouter, IDuplexSessionRouter, IDisposable
{ ... }
```

From the above you can see that the `RoutingService` class is defined as a **sealed**class and implements multiple service contracts in order to supports multiple message exchange patterns (**MEP**). Each service contract provides support for a different messaging exchange pattern (**MEP**). Please go through the table down below for details(from msdn)-

Service Contract	Description
IDuplexSessionRouter	Defines the interface required to process messages from duplex session channels.
IRequestReplyRouter	Defines the interface required to process messages from request-reply channels.
ISimplexDatagramRouter	Defines the interface required for processing messages from simplex datagram.
ISimplexSessionRouter	Defines the interface required to process messages from simplex session channels.

Please note that `ISimplexDatagramRouter` and `IRequestReplyRouter`interfaces define generic one-way and request-reply service contract definitions that can be used in conjunction with business-specific service contracts while the other two, `ISimplexSessionRouter` & `IDuplexSessionRouter`, interfaces are session demanding service contracts. `ISimplexSessionRouter` is basically a fire-and-forget operation that takes place within the scope of a session while the`IDuplexSessionRouter` is basically a duplex session aware operation that needs to calling back to the client application within the scope of a session. Please see the definitions of these interfaces down below-

```
[ServiceContract(Namespace = "http://schemas.microsoft.com/netfx/2009/05/routing",
            SessionMode = SessionMode.Allowed)]
public interface ISimplexDatagramRouter
{
   [OperationContract(AsyncPattern = true, IsOneWay = true, Action = "*")]
   IAsyncResult BeginProcessMessage(Message message, AsyncCallback callback, object state);

   void EndProcessMessage(IAsyncResult result);
}

[ServiceContract(Namespace = "http://schemas.microsoft.com/netfx/2009/05/routing",
            SessionMode = SessionMode.Allowed)]
public interface IRequestReplyRouter
{
   [OperationContract(AsyncPattern = true, IsOneWay = false, Action = "*", ReplyAction = "*")]
   [GenericTransactionFlow(TransactionFlowOption.Allowed)]
   IAsyncResult BeginProcessRequest(Message message, AsyncCallback callback, object state);

   Message EndProcessRequest(IAsyncResult result);
}

[ServiceContractAttribute(Namespace = "http://schemas.microsoft.com/netfx/2009/05/routing",
            SessionMode = SessionMode.Required)]
public interface ISimplexSessionRouter
{
   [OperationContractAttribute(AsyncPattern = true, IsOneWay = true, Action = "*")]
   IAsyncResult BeginProcessMessage(Message message, AsyncCallback callback, Object state);

   void EndProcessMessage(IAsyncResult result);
}

public interface IDuplexSessionRouter
{
   [OperationContractAttribute(AsyncPattern = true, IsOneWay = true, Action = "*")]
```

```
    IAsyncResult BeginProcessMessage(Message message, AsyncCallback callback, Object state);

    void EndProcessMessage(IAsyncResult result;)
}
```

The main purpose of the `RoutingService` class is to receive incoming messages from the client applications through the virtual endpoint(s) and to "**route**" them to an appropriate actual service by evaluating each incoming message against a set of message filters. Hence, you can control the routing behavior by defining the message filters, typically in a configuration file.

Hosting the Routing Service

You can host the `RoutingService` just like other **WCF** services using self-hosting or managed hosting techniques. Below is an typical example of self-hosting technique to host `RoutingService` using `ServiceHost` class-

```
var host = new ServiceHost(typeof(RoutingService));

try
{
    host.Open();
    Console.ReadLine();
    host.Close();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    host.Abort();
}
```

Just like other **WCF** services you can also configure the `RoutingService` through configuration file where you define the `RoutingService` endpoint(s),`RoutingService Behavior`, the routing filters and actual services endpoint(s) where finally incoming messages would be routed. Let's try to understand these concepts in coming sections.

Configuring Routing Service Endpoint(s)

You can configure one or more `RoutingService` endpoint(s) by choosing a **WCF**binding and one of the `RoutingService` supported service contracts implemented by the `RoutingService` class as described above (`IRequestReplyRouter`, `ISimplexDatagramRouter`,`ISimplexSessionRouter`, `IDuplexSessionRouter`).

Below is an example of `RoutingService` with two routing endpoints.

```
<services>
    <service name="System.ServiceModel.Routing.RoutingService"><!--Routing Service -->
        <endpoint address="" binding="basicHttpBinding"
                          contract="System.ServiceModel.Routing.IRequestReplyRouter"
name="MessageBroker" /> <!--MessageBroker-->
        <endpoint address="regular" binding="basicHttpBinding"
                          contract="System.ServiceModel.Routing.IRequestReplyRouter" name="Regular" />
<!--Regular-->
```

```
      <host>
         <baseAddresses>
            <add baseAddress="http://localhost:8080/RoutingService/Router" />
         </baseAddresses>
      </host>
   </service>
</services>
```

In the above, first endpoint uses `basicHttpBinding` with `IRequestReplyRouter` service contract (request-reply) and second endpoint uses `wsHttpBinding` with `ISimplexDatagramRouter` service contract (one-way). The endpoints configured above are basically routing endpoints (virtual endpoints or message brokers) that will be consumed by the client applications. Client applications can use one of these endpoints to invoke actual services and each service invocation will be directed directly to the `RoutingService`. When the `RoutingService` receives a message through one of these routing endpoints, it evaluates the message against a set of message filters to determine where to forward the message.

Below is an example of client application endpoints based on above `RoutingService` configuration-

```
<client><!--Client Side Endpoints for Routing Service -->
   <endpoint address="http://localhost:8080/RoutingService/Router" binding="basicHttpBinding"
                  contract="IComplexNumber" name="BasicHttpBinding_IComplexNumber" />
   <endpoint address="http://localhost:8080/RoutingService/Router/regular" binding="basicHttpBinding"
                  contract="IRealNumber" name="BasicHttpBinding_IRealNumber" />
</client>
```

Configuring Routing Service Message Filter(s)

You can configure `RoutingService` with message filters to manage the routing message filters. **WCF 4.0** comes with a `RoutingBehavior` for the same. So in order to configure `RoutingService` with message filters, you need to define a named behavior configuration (say "**routingFilters**") by enabling the `RoutingBehavior` followed by specifying the name of the filter table. After that you need to apply the "**routingFilters**" behavior to the `RoutingService` through the `behaviorConfiguration` attribute. See the example below-

```
<behaviors>
   <serviceBehaviors>
      <behavior name="routingFilters">
         <routing filterTableName="RoutingTable" />
      </behavior>
   </serviceBehaviors>
</behaviors>
<services>
   <service name="System.ServiceModel.Routing.RoutingService" behaviorConfiguration="routingFilters">
        ...
   </service>
</services>
```

Configuring Routing Service Target Service(s)

You will need endpoint definitions for the target actual services intend to route to. You can define these target

endpoints within the WCF `<client>` configuration section like below-

```
<client>
   <endpoint address="http://localhost:8081/ComplexNumberService" binding="basicHttpBinding"
                  contract="*" name="ComplexNumber" />
   <endpoint address="http://localhost:8082/RealNumberService" binding="basicHttpBinding"
                  contract="*" name="RealNumber" />
</client>
```

The **"*"** character in the contract attribute enables the service to accept any incoming messages and not just those specified by a particular service contract.

Defining Routing Service Filter Table

Filter Table determines the routing logic at runtime. You can define the filter table entries within the element. Each entry within the defines a mapping between a routing "**filter**" and a target endpoint. You can define the "**filters**" within theelement. Each entry specifies type of filter along with the filter-specific data e.g. action value, an `XPath` expression, routing endpoint name etc.).

Below is an example of configuring a filter table "`RoutingData`" with two filters that maps two endpoints. Here `EndpointName` filter type has been used.

```
<routing>
   <filters>
      <filter name="ComplexNumberFilter" filterType="EndpointName" filterData="MessageBroker" />
      <filter name="RealNumberFilter" filterType="EndpointName" filterData="Regular" />
   </filters>
   <filterTables>
      <filterTable name="RoutingTable">
         <add filterName="ComplexNumberFilter" endpointName="ComplexNumber" />
         <add filterName="RealNumberFilter" endpointName="RealNumber" />
      </filterTable>
   </filterTables>
</routing>
```

WCF 4.0 comes with several built-in message `filterTypes` that you can use to inspect the content of the incoming messages. Please see the table down below for the details(from msdn) -

Filter Type	Description	Explaination
Action	Uses the ActionMessageFilter class to match messages containing a specific action.	The action to filter upon.
EndpointAddress	Uses theEndpointAddressMessageFilterclass, withIncludeHostNameInComparison ==true to match messages containing a specific address.	The address to filter upon (in the To header).
EndpointAddressPrefix	Uses thePrefixEndpointAddressMessageFilterclass, withIncludeHostNameInComparison ==true to match messages containing a specific address prefix.	The address to filter upon using longest prefix matching.

And	Uses the StrictAndMessageFilterclass that always evaluates both conditions before returning.	filterData is not used; instead filter1 and filter2 have the names of the corresponding message filters (also in the table), which should beANDed together.
Custom	A user-defined type that extends the MessageFilter class and has a constructor taking a string.	The customType attribute is the fully qualified type name of the class to create; filterData is the string to pass to the constructor when creating the filter.
EndpointName	Uses theEndpointNameMessageFilter class to match messages based on the name of the service endpoint they arrived on.	The name of the service endpoint, for example: “serviceEndpoint1”. This should be one of the endpoints exposed on the Routing Service.
MatchAll	Uses the MatchAllMessageFilterclass. This filter matches all arriving messages.	filterData is not used. This filter will always match all messages.
XPath	Uses the XPathMessageFilter class to match specific XPath queries within the message.	The XPath query to use when matching messages.

Features

- Content-based routing
    - Service aggregation
    - Service versioning
    - Priority routing
    - Dynamic configuration
- Context-based routing
- SOAP processing
- Protocol bridging
- Backup endpoints
- Load Balancing
- Multicasting
- Advanced error handling

Demo Service

Now after the description of the `RoutingService`, let's understand it through examples. I've created a demo service `ComplexNumberCalculator` for this purpose. I've defined one data contract `Complex` and one service contract`IComplexNumber,` and then created a `ComplexNumberCalculator` service by implementing the `IComplexNumber` service contract. Please see the code below-

```
[DataContract]
public class Complex
{
    [DataMember]
   public  double Real;

    [DataMember]
    public double Imaginary;
}

[ServiceContract]
public interface IComplexNumber
{
    [OperationContract]
    Complex Add(Complex x, Complex y);

    [OperationContract]
    Complex Subtract(Complex x, Complex y);

    [OperationContract]
    Complex Multiply(Complex x, Complex y);

    [OperationContract]
    Complex Divide(Complex x, Complex y);

    [OperationContract]
    double Modulus(Complex x);

    [OperationContract]
    double Argument(Complex x);

    [OperationContract]
    Complex Conjugate(Complex x);

    [OperationContract]
    Complex Recipocal(Complex x);
}

public class ComplexNumberCalculator : IComplexNumber
{
    public Complex Add(Complex x, Complex y)
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Add");

        var z = new Complex();

        z.Real = x.Real + y.Real;
        z.Imaginary = x.Imaginary + y.Imaginary;

        return z;
    }

    public Complex Subtract(Complex x, Complex y)
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Subtract");

        var z = new Complex();

        z.Real = x.Real - y.Real;
        z.Imaginary = x.Imaginary - y.Imaginary;

        return z;
    }

    public Complex Multiply(Complex x, Complex y)
```

```
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Multiply");

        var z = new Complex();

        z.Real = x.Real * y.Real - x.Imaginary * y.Imaginary ;
        z.Imaginary = x.Real * y.Imaginary + x.Imaginary * y.Real;

        return z;
    }

    public Complex Divide(Complex x, Complex y)
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Divide");

        var z = new Complex();

        var modulusY = this.Modulus(y);

        z.Real = (x.Real * y.Real + x.Imaginary * y.Imaginary) / (modulusY * modulusY);
        z.Imaginary = (x.Imaginary * y.Real - x.Real * y.Imaginary) / (modulusY * modulusY);

        return z;
    }

    public double Modulus(Complex x)
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Modulus");

        var modX = Math.Sqrt(x.Real * x.Real + x.Imaginary * x.Imaginary);

        return modX;
    }

    public Complex Conjugate(Complex x)
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Conjugate");

        var z = new Complex();

        z.Real = x.Real;
        z.Imaginary = -1 * x.Imaginary;

        return z;
    }

    public double Argument(Complex x)
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Argument");

        var argumentX = Math.Atan(x.Imaginary/x.Real);

        return argumentX;
    }

    public Complex Recipocal(Complex x)
    {
        Console.WriteLine("Invoked ComplexNumberCalculator Operation: Recipocal");

        var z = new Complex();

        var modulusX = this.Modulus(x);
        var conjugateX = this.Conjugate(x);

        z.Real = conjugateX.Real / (modulusX * modulusX);
```

```
        z.Imaginary = conjugateX.Imaginary / (modulusX * modulusX);

        return z;
    }
```

I've hosted `ComplexNumberCalculator` service in a windows console application using self-hosting technique as below-

```
var host = new ServiceHost(typeof(ComplexNumberCalculator));

try
{
   host.Open();
   Console.ReadLine();
   host.Close();
}
catch (Exception ex)
{
   Console.WriteLine(ex.Message);
   host.Abort();
}
```

and configured two endpoints, one service endpoint and one standard mex endpoint in order to exchange **metadata** as below.

```
<services>
   <service name="CalculatorService.ComplexNumberCalculator">
      <endpoint address="" binding="basicHttpBinding" contract="CalculatorService.IComplexNumber" />
      <endpoint address="mex" kind="mexEndpoint" />
      <host>
         <baseAddresses>
            <add baseAddress="http://localhost:8081/ComplexNumberService" />
         </baseAddresses>
      </host>
  </service>
</services>
```

I've also enabled service **metadata** by defining a default behavior (by ommiting name) as below-

```
<behaviors>
   <serviceBehaviors>
      <behavior name="">
         <serviceMetadata />
      </behavior>
   </serviceBehaviors>
</behaviors>
```

I'll use this service for all demonstrations throughout this post.

Simple Routing Service using MatchAll filterType

In this example I'm going to configure a simple `RoutingService` that will just pass (route) all incoming messages from our `ComplexNumberCalculator`service's client application to the `ComplexNumberCalculator` service. Here`RoutingService` will just act as an intermediary. I've hosted the`RoutingService` in windows console application using self-hosting technique. You can

host `RoutingService` in **IIS/WAS/Windows Service/AppFabric** as per your need.

First I've configured our `RoutingService` with following endpoint (virtual endpoint) as below-

```
<services>
   <service name="System.ServiceModel.Routing.RoutingService">
      <endpoint address="" binding="basicHttpBinding"
contract="System.ServiceModel.Routing.IRequestReplyRouter"
                          name="VirtualEndpoint"  />
      <host>
         <baseAddresses>
            <add baseAddress="http://localhost:8080/RoutingService/Router" />
         </baseAddresses>
      </host>
   </service>
</services>
```

Please note that here I've used `IRequestReplyRouter` service contract as our`ComplexNumberCalculator` service supports request-reply **MEP**.

Then I've defined an endpoint for our target service: `ComplexNumberCalculator`as below-

```
<client>
   <endpoint address="http://localhost:8081/ComplexNumberService" binding="basicHttpBinding"
                  contract="*" name="ComplexNumber" />
</client>
```

Next I've enabled the `RoutingBehavior` followed by specifying the name of the filter table. I've done this by defining default behavior like below-

```
<behaviors>
   <serviceBehaviors>
      <behavior name="">
         <routing filterTableName="RoutingTable" />
      </behavior>
   </serviceBehaviors>
</behaviors>
```

The next step would be to define our filter table: `RoutingTable` by adding entries to it. But as each entry within the filter table defines the mapping between a routing filter and a target endpoint, we'll define filters first then our filer table. I've defined following filter for our filter table-

```
<routing>
   <filters>
      <filter name="ComplexNumberFilter" filterType="MatchAll" />
   </filters>
...
```

I've used above `MatchAll` filter type that matches all incoming messages. Note that our goal is to just pass all incoming messages from the client to`ComplexNumberCalculator`.

Finally I've configured our filter table: `RoutingTable` with the filter type defined above as below-

```
<filterTables>
   <filterTable name="RoutingTable">
      <add filterName="ComplexNumberFilter" endpointName="ComplexNumber" />
```

```
   </filterTable>
</filterTables>
```

In the above, I've added a single table entry and set the `filterName` attribute to "**ComplexNumberFilter**" (name of the filter type defined above) and `endpointName` attribute to "**ComplexNumber**" (name of the target service endpoint, the ultimate receiver).

At last, I've created a console client application to call the service. I've generated `ComplexNumberCalculator` service code file by using the **svcutil.exe** from the command line like below-

```
svcutil.exe http://localhost:8081/ComplexNumberService/mex
```

and configured the client side endpoint as below-

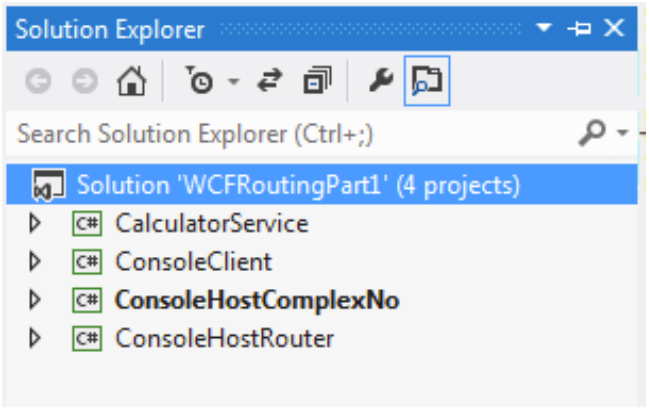
```
<system.serviceModel>
<client>
   <endpoint address="http://localhost:8080/RoutingService/Router" binding="basicHttpBinding"
                  contract="IComplexNumber" name="BasicHttpBinding_IComplexNumber" />
</client>
</system.serviceModel>
```

Notice that I've used here service contract `IComplexNumber` (of `ComplexNumberCalculator` servcie) instead of `IRequestReplyRouter` (of `RoutingService`). `IComplexNumber` service contract will be used to invoke `ComplexNumberCalculator` service's operations by creating client side channel. Below is the client application code-

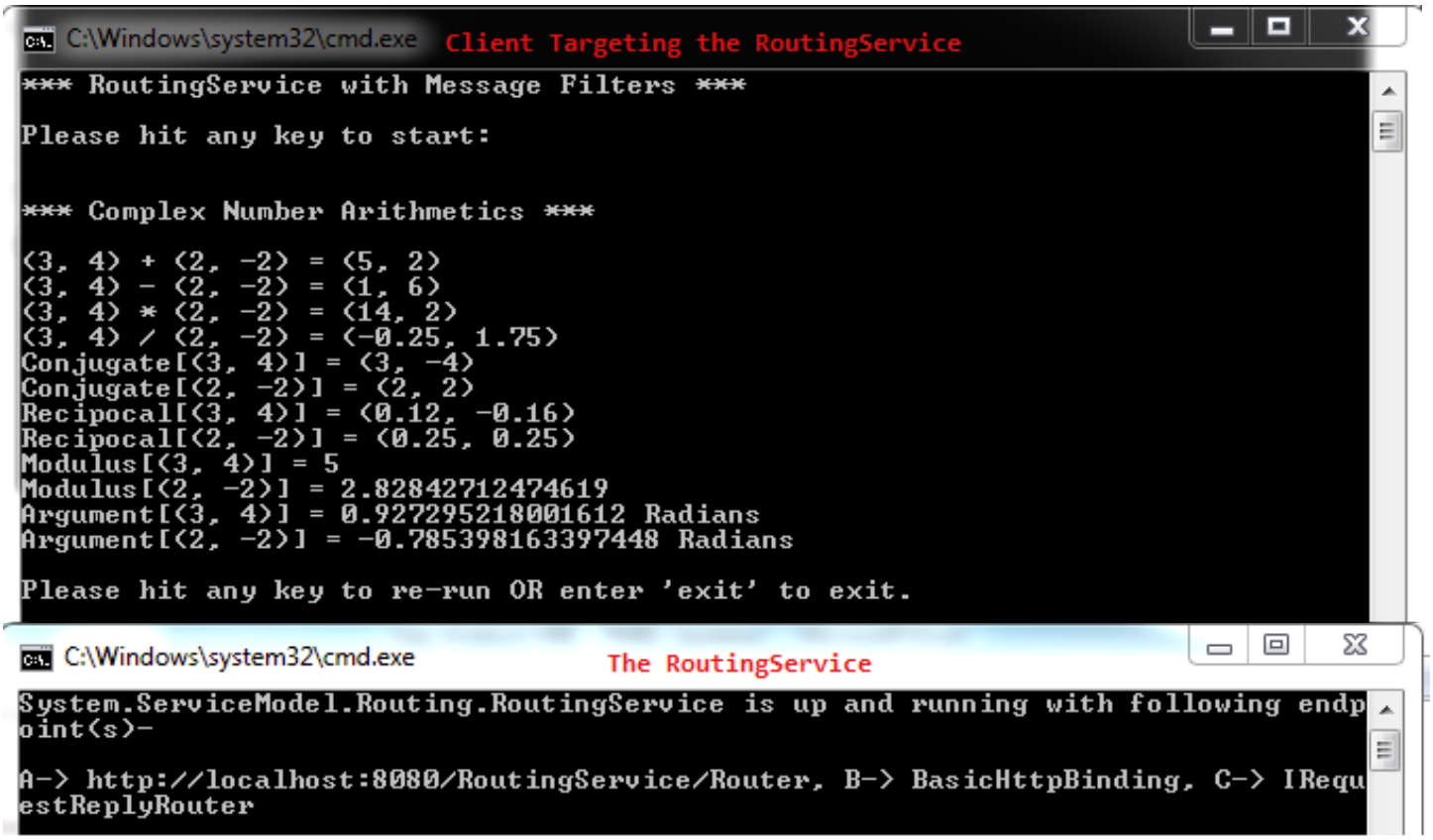
```
var cf = new ChannelFactory<IComplexNumber>("BasicHttpBinding_IComplexNumber");

var channel = cf.CreateChannel();

var z1 = new Complex();
var z2 = new Complex();

z1.Real = 3D;
z1.Imaginary = 4D;

z2.Real = 2D;
z2.Imaginary = -2D;

Console.WriteLine("*** RoutingService with Message Filters ***\n");
Console.WriteLine("Please hit any key to start: ");
string command = Console.ReadLine();

while (command != "exit")
{
   ComplexNumberArithmetics(channel, z1, z2);

   Console.WriteLine("\nPlease hit any key to re-run OR enter 'exit' to exit.");
   command = Console.ReadLine();
}

((IClientChannel)channel).Close();
```

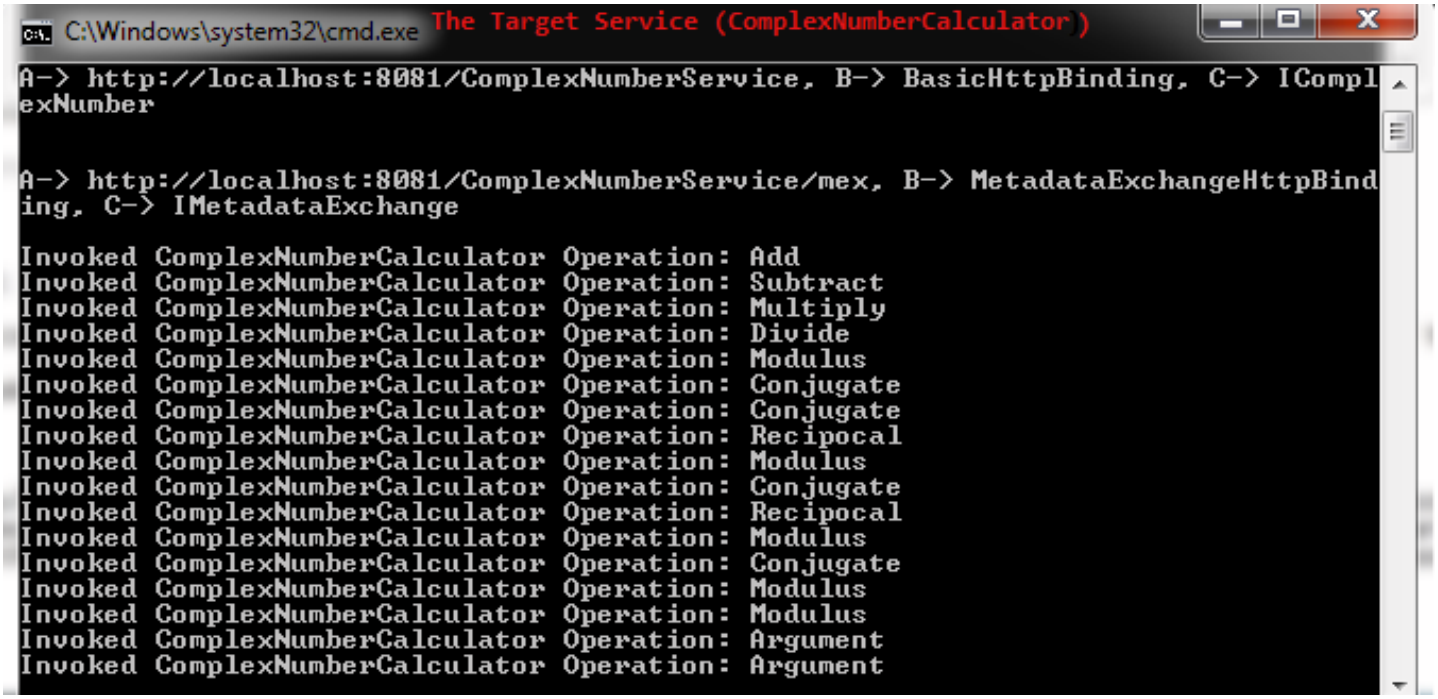
The method `ComplexNumberArithmetics` performs complex number arithmetics using the channel created above (you can find the code of the `ComplexNumberArithmetics` method in the sample).

Before running our demo, just have a quick look of sample project provided with this post-



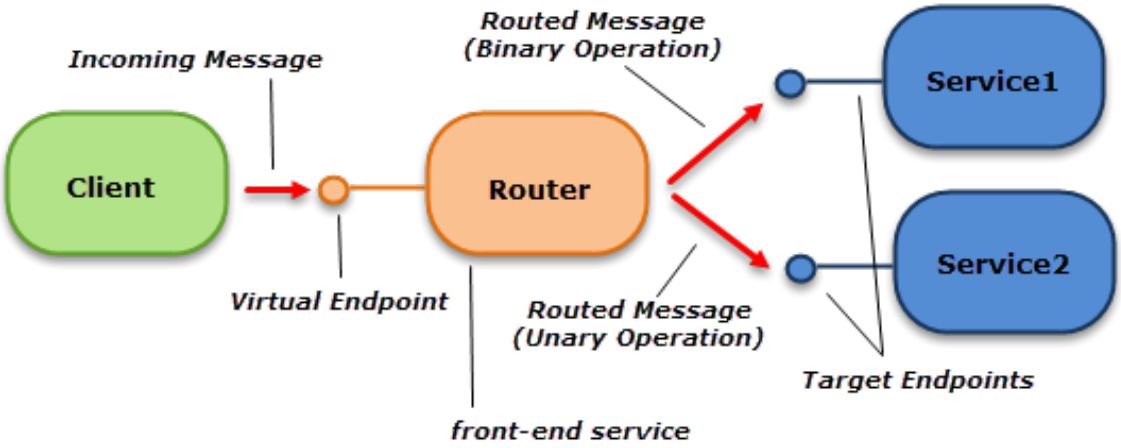
There are four projects in the solution: **CalculatorService**, **ConsoleClient**,**ConsoleHostComplexNo** & **ConsoleHostRouter**. Now set **ConsoleClient**,**ConsoleHostComplexNo** & **ConsoleHostRouter** projects as a **Start Up projects**and hit **Ctrl+F5** keys in order to start the projects. Now press any key on the console client and you can verify that the routing is working properly. you can see that the messages arrive at `ComplexNumberCalculator` service after they are "**routed**" by the intermediary `RoutingService`.





Content Based Routing

In content-based routing techniques, the target service is determined by evaluating the content of a particular incoming message. You can evaluate incoming message**Header** or **Body** to decide the target service endpoint. You can inspect the **SOAP**action of an incoming message or some value inside the message payload such as an element, attribute or header value etc. you can use `Action`, `XPathfilterTypes` to implement content based routing.



Let's consider an example to understand the content based routing with our`ComplexNumberCalculator` service. Suppose that we want to route the **Binary Operations** (Add, Subtract, Multiply & Divide) to `ComplexNumberService1` and the **Unary Operations** (Modulus, Argument, Conjugate & Reciprocal) to`ComplexNumberService2`.

I'll implement the same by two ways; first by using the different`ComplexNumberCalculator` action values within the **SOAP** header and secondly by using the **XPath Expressions**.

Content Based Routing using the Action Values

Lets start with the content-based routing using the action values by updating our`RoutingService`. First I've defined two endpoints for the target services like below-

```
<client>
        <endpoint address="http://localhost:8081/ComplexNumberService1" binding="basicHttpBinding"
                  contract="*" name="BinaryOperation" />
        <endpoint address="http://localhost:8081/ComplexNumberService2" binding="basicHttpBinding"
                  contract="*" name="UnaryOperation" />
</client>
```

Next I've defined filters for each of the different `ComplexNumberCalculator`service action values like below-

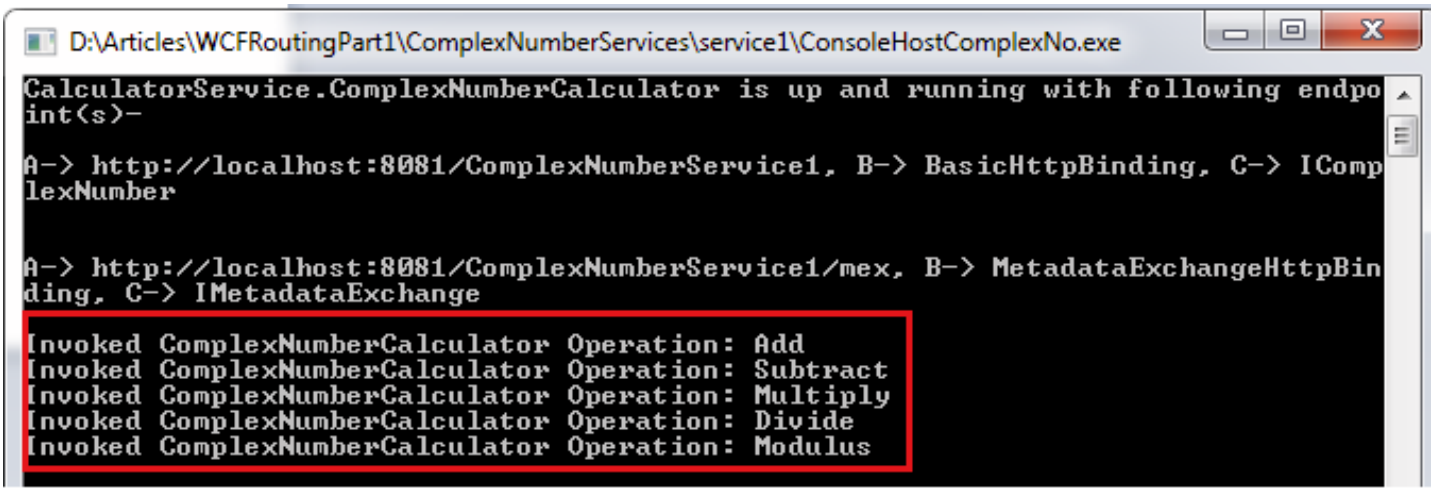
```
<filters>
          <!--Binary Operation-->
          <filter name="AddFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Add" />
          <filter name="SubtractFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Subtract" />
          <filter name="MultiplyFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Multiply" />
          <filter name="DivideFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Divide" />
          <!--Unary Operation-->
          <filter name="ModulusFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Modulus" />
          <filter name="ArgumentFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Argument" />
          <filter name="ConjugateFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Conjugate" />
          <filter name="RecipocalFilter" filterType="Action"
filterData="http://tempuri.org/IComplexNumber/Recipocal" />
</filters>
```

Finally I've mapped **Binary Operations** to the `ComplexNumberService1` endpoint and **UnaryOperations** to the `ComplexNumberService2` endpoint in the filter table: `RoutigTable` like below-

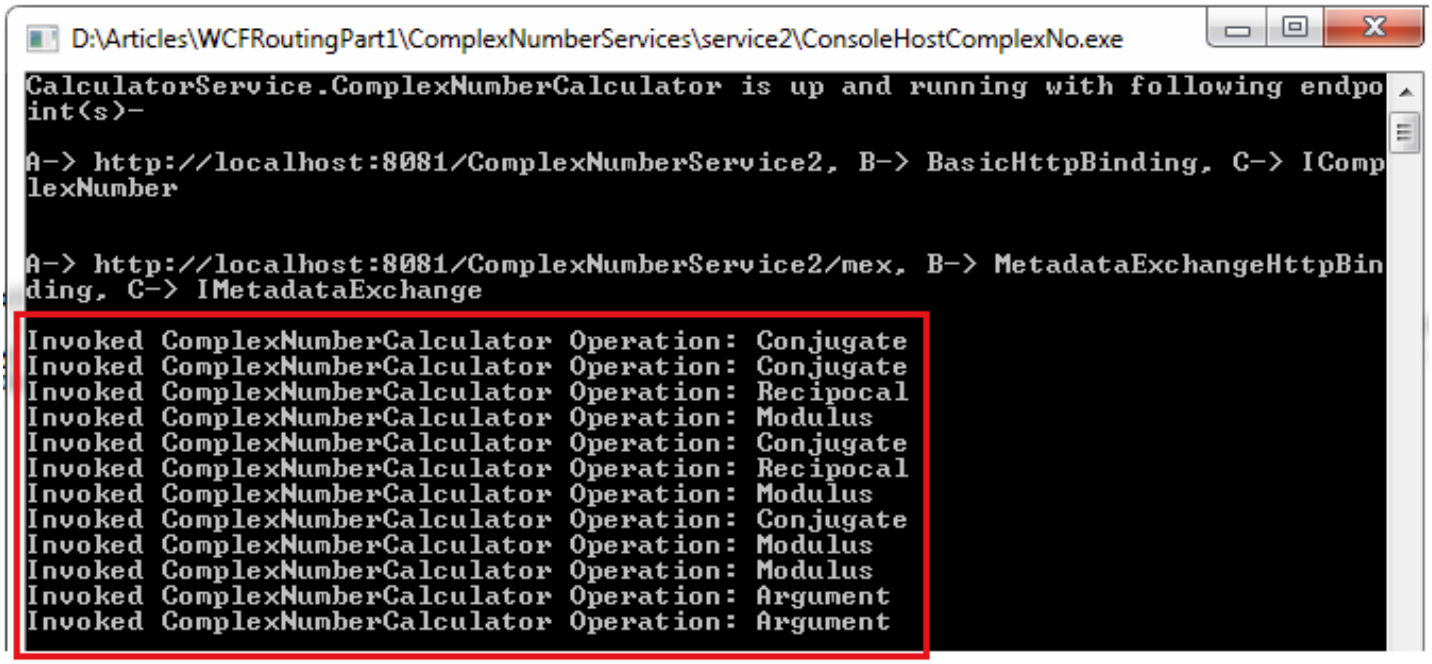
```
<filterTables>
          <filterTable name="RoutingTable">
            <add filterName="AddFilter" endpointName="BinaryOperation" />
            <add filterName="SubtractFilter" endpointName="BinaryOperation" />
            <add filterName="MultiplyFilter" endpointName="BinaryOperation" />
            <add filterName="DivideFilter" endpointName="BinaryOperation" />

            <add filterName="ModulusFilter" endpointName="UnaryOperation" />
            <add filterName="ArgumentFilter" endpointName="UnaryOperation" />
            <add filterName="ConjugateFilter" endpointName="UnaryOperation" />
            <add filterName="RecipocalFilter" endpointName="UnaryOperation" />
          </filterTable>
</filterTables>
```

That's it. Now set **ConsoleClient** & **ConsoleHostRouter** projects as **Start Up projects** and hit **Ctrl+F5** keys in order to run the projects. Next run the**WCFRoutingPart1\ComplexNumberServices\StartAllServices.cmd** file (see the sample code) from the **Visual Studio Developer Command Prompt** (in**Administrator** mode) in order to

start `ComplexNumberService1` and`ComplexNumberService2` services. Finally press any key on the console client and you can verify that the **Binary Operation**s are routing to the`ComplexNumberservice1` service while the **UnaryOperations** are routing to the`ComplexNumberService2` service by the intermediary `RoutingService`.



The Target ComplexNumberCalculator1 Service



The Target ComplexNumberCalculator2 Service

Content Based Routing using the XPath Expressions

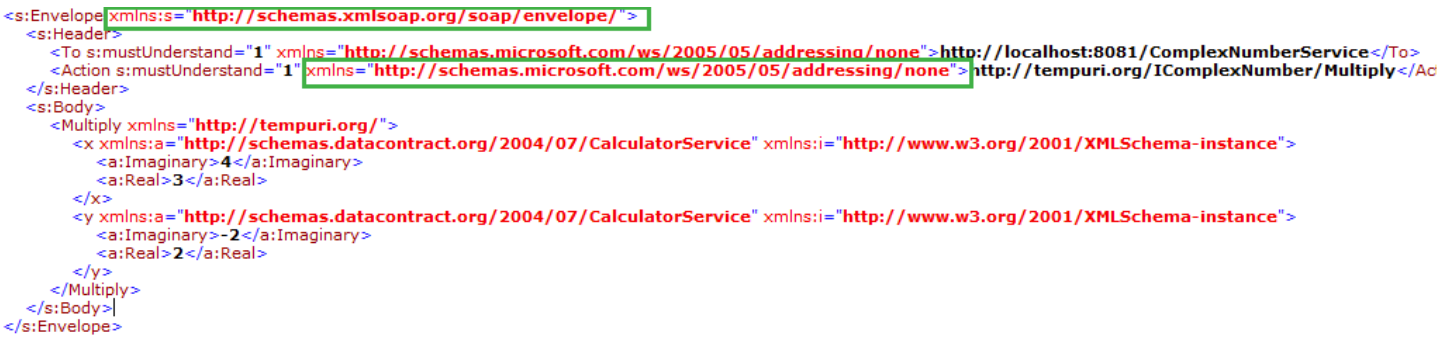
You can use `XPath` filterType to evaluate a variety of different `XPath` expressions against the incoming messages. It is more powerful and flexible and you can use**XPath Expression** to inspect and evaluate any part of the incoming message including **SOAP** headers or the **SOAP** body.

Lets start with the content-based routing using the `XPath filterType` by updating our `RoutingService`. First I've defined a set of namespace prefix bindings using the the `<namespaceTable>` element as below-

```
<namespaceTable>
```

```
            <add prefix="s" namespace="http://schemas.xmlsoap.org/soap/envelope/" />
            <add prefix="wsa" namespace="http://schemas.microsoft.com/ws/2005/05/addressing/none" />
</namespaceTable>
```

See the screen shot below to understand how I defined the namespace prefixes-



Next I've defined filters for each different `ComplexNumberCalculator` service action values using the `XPath filterType` as down below-

```
<filters>
            <!--Binary Operation-->
            <filter name="AddFilter" filterType="XPath" filterData="/s:Envelope/s:Header/wsa:Action
='http://tempuri.org/IComplexNumber/Add'" />
            <filter name="SubtractFilter" filterType="XPath"
filterData="/s:Envelope/s:Header/wsa:Action ='http://tempuri.org/IComplexNumber/Subtract'" />
            <filter name="MultiplyFilter" filterType="XPath"
filterData="/s:Envelope/s:Header/wsa:Action ='http://tempuri.org/IComplexNumber/Multiply'" />
            <filter name="DivideFilter" filterType="XPath" filterData="/s:Envelope/s:Header/wsa:Action
='http://tempuri.org/IComplexNumber/Divide'" />
            <!--Unary Operation-->
            <filter name="ModulusFilter" filterType="XPath" filterData="/s:Envelope/s:Header/wsa:Action
='http://tempuri.org/IComplexNumber/Modulus'" />
            <filter name="ArgumentFilter" filterType="XPath"
filterData="/s:Envelope/s:Header/wsa:Action ='http://tempuri.org/IComplexNumber/Argument'" />
            <filter name="ConjugateFilter" filterType="XPath"
filterData="/s:Envelope/s:Header/wsa:Action ='http://tempuri.org/IComplexNumber/Conjugate'" />
            <filter name="RecipocalFilter" filterType="XPath"
filterData="/s:Envelope/s:Header/wsa:Action ='http://tempuri.org/IComplexNumber/Recipocal'" />
</filters>
```

Notice the **XPath Expression** contained in the `filterData` attribute will be evaluated against the incoming message (the expressions simply inspect the action values).

Now just follow the instructions of the previous example to run the demo, you'll see the same result as before.

`XPath` filter technique is very useful and you can use the same to route messages based on custom **SOAP** headers or the content found within the body of the **SOAP**message.

Conclusion

Routing in **WCF** is a very wide topic. I've covered **WCF Routing Service** concept and explained how to configure a `RoutingService` (endpoint(s), target service(s), message filter(s) and filter table) in this post. Then I've demonstrated a simple`RoutingService` using `MatchAll filterType` and finally explored Content-based routing using **Action** values and **XPath Expressions**. But lot to cover. In the next series of this

article I'll cover some routing topics like protocol bridging, context-based routing, load balancing etc. Till then, happy coding.

History

- 7th Jun, 2014 -- Article updated (Added a new entry for the fourth part of the series in 'All Posts' section)
- 6th Jun, 2014 -- Article updated (added Features section)
- 28th May, 2014 -- Article updated (Added a new entry for the third part of the series in 'All Posts' section)
- 27th May, 2014 -- Article updated (updated the URL of the second part of the series in 'All Posts' section)
- 24th May, 2014 -- Article updated
    - (updated the article's title)
    - (updated the entries of the 'All Posts' section)
- 22nd May, 2014 -- Article updated (corrected typo mistakes)
- 21th May, 2014 -- Article updated (updated the entries of the 'All Posts' section)
- 20th May, 2014 -- Article updated (re-arranged the entries of the 'All Posts' section)
- 19th May, 2014 -- Article updated
    - (Updated the table of contents section-- added an entry of the 'All Posts' section)
    - (Added the 'All Posts' section)
- 14th May, 2014 -- Article updated (Added table of contents section)
- 13th May, 2014 -- Original version posted

License

Share

About the Author

Samir NIGAM

Technical Lead Infogain India Pvt Ltd

India

Samir NIGAM is a **Microsoft Certified Professional**. He is an insightful IT professional with results-driven comprehensive technical skill having rich, hands-on work experience n web-based applications using **ASP.NET**, **C#**, **AJAX**, **Web Service**, **WCF**, **jQuery**, **Microsoft Enterprise Library** , **LINQ**, **MS Entity Framework**, **nHibernate**, **MS SQL Server** & **SSRS**.

He has earned his master degree (**MCA**) from U.P. Technical University, Lucknow, INDIA, his post graduate dipoma (**PGDCA**) from Institute of Engineering and Rural Technology, Allahabad, INDIA and his bachelor degree (**BSc - Mathematics**) from University of Allahabad, Allahabad, INDIA.

He has good knowledge of **Object Oriented Programming**, **n-Tier Architecture**, **SOLID Principle**, and **Algorithm Analysis & Design** as well as good command over cross-browser client side programming using **JavaScript** &**jQuery**,.

Awards:

- **Code Project MVP 2009.**
- **Best ASP.NET article of December 2008.**
- **Best ASP.NET article of June 2008.**