Instrumenting WCF, Extending WCF using IServiceBehavior, IOperationBehavior, and IParameterInspector



[**Assil**](https://www.codeproject.com/script/Membership/View.aspx?mid=1201825), 9 Jan 2014 [CPOL](http://www.codeproject.com/info/cpol10.aspx)

|  |  |
| --- | --- |
| https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-fill-md.png  https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-empty-md.png | 4.97 (13 votes) |
|  | |  |  |  | | --- | --- | --- | | Rate this: | [vote 1vote 2vote 3vote 4vote 5](https://www.codeproject.com/Articles/598402/Instrumenting-WCF-Extending-WCF) |  | |

This article describes an advanced technique or feature in WCF

* [**Download source (no EXE) - 13.4 KB**](https://www.codeproject.com/KB/dotnet/598402/WCF-noexe.zip)
* [**Download source - 40.7 KB**](https://www.codeproject.com/KB/dotnet/598402/WCF.zip)

Introduction

WCF has become such a big buzz word in the IT solution world, it revolutionary transfers the web services concept to a totally new world, it perfectly implements and extends the Service Oriented Architecture and it also opens the ways to freely design and build flexible solutions.

Background

My intent was to talk about a very important feature of the WCF and that was extensibility, however, I found a very good series of articles online that talk about extensibility in a very detailed and simple professional way that I can’t ever compete with, so I decided to briefly talk about extensibility referring the reader to [that great article by Carlos Figueira](http://blogs.msdn.com/b/carlosfigueira/archive/2011/03/14/wcf-extensibility.aspx) MSDN blog.

In the lines below, I am going to show you how you can extend WCF in a very simple way using three extending points, and in case you want to know more, you can just read the link above.

Using the Code

This article requires you to be well-versed with WCF’s main concepts like service, contracts, behaviors, bindings, etc.

I will build a simple WCF service I will extend the service in three places, or show you how you can extend the service in three places and also show you the execution path by tracing it. Let’s start by considering this simple contract:

Hide   Copy Code

[ServiceContract(Name = "PasswordGenerator")]

public interface IPasswordGenerator

{

[OperationContract(Name = "GeneratePassword")]

string GeneratePassword();

[OperationContract(Name="ParameterizedPasswordGenerator")]

string GeneratePassword(int length);

}

And this implementation:

Hide   Copy Code

public class PasswordGenerator:IPasswordGenerator

{

[OperationBehavior()]

public string GeneratePassword()

{

return "You called GeneratePassword()";

}

[OperationBehavior()]

public string GeneratePassword(int length)

{

return "You called GeneratePassword(int length) overload";

}

}

The host application and the client application are merged in this one small application for brevity and simplicity and because this is out of the scope of our article. In this, I wish to draw your attention to the following points:

1. Using named pipes binding as a choice of many, you can choose whatever you want,
2. The host takes the service class and **not** the interface.
3. The endpoint takes the type of the Interface.
4. Host should be open while using it and you should close it once you are done.
5. The factory uses the class and **not** the interface.
6. It should also be closed when you no longer need it.
7. I used two overloads of GeneratePassword, but because I am using the original assembly not the proxy generated one, that is why they maintained the original names of the methods.

Hide   Copy Code

static void Main(string[] args)

{

ServiceEndpoint serviceEndpoint;

Uri basePipeAddress = new Uri(@"net.Pipe://localhost/Password/Mine");

using (ServiceHost host =

new ServiceHost(typeof(Password.PasswordGenerator), basePipeAddress))

{

serviceEndpoint = host.AddServiceEndpoint(typeof(

Password.IPasswordGenerator), new NetNamedPipeBinding(), string.Empty);

host.Open();

using (var factory = new

ChannelFactory<password.passwordgenerator>(new NetNamedPipeBinding()))

{

var proxy = factory.CreateChannel(serviceEndpoint.Address);

proxy.GeneratePassword();

proxy.GeneratePassword(1500);

factory.Close();

}

host.Close();

}

}

The code above is sufficient to host and consume the service using efficient named pipes binding. So far is just setting up the environment. Now we need to create three extensions and log them all, just to see which one executes when I will not write any useful code in any, just telling you that this is the place where you have full access on the WCF and you can manipulate all its actions. This is the logging class.

Hide   Copy Code

public static class Logger

{

private static void Log(string message)

{

try

{

using (var stream = new StreamWriter(@"G:\log.log", true))

{

stream.WriteLine(message);

stream.Flush();

stream.Close();

}

}

catch (Exception)

{

}

}

public static void Log(string className, string methodName, string parameter)

{

Log(string.Format("Class {0} called method {1} with parameter {2} @ {3}",

className, methodName, parameter, DateTime.Now));

}

}

**First Extension**

Extending the Service behavior by implementing IserviceBehavior interface. This interface can be used to inspect or change the entire service description and runtime.

Hide   Copy Code

public interface IServiceBehavior

{

void Validate(ServiceDescription serviceDescription, ServiceHostBaseserviceHostBase);

void AddBindingParameters(ServiceDescription serviceDescription,

ServiceHostBase serviceHostBase, Collection<ServiceEndpoint>

endpoints,BindingParameterCollection bindingParameters);

void ApplyDispatchBehavior

(ServiceDescription serviceDescription,ServiceHostBase serviceHostBase);

}

Validate: ensure that the service (or the host) doesn’t violate the validation logic on the behavior, it provides the ability to inspect the service host and the service description to confirm that the service can run successfully.

AddBindingParameters: is used to pass information to the binding elements themselves, and it is where you pass custom data to binding elements to support the contract implementation.

It will be invoked once per endpoint defined in the service, unlike other methods that are called only once.

ApplyDispatchBehavior Provides the ability to change run-time property values or insert custom extension objects such as error handlers, message or parameter interceptors, security extensions, and other custom extension objects.

Here is the implementation:

Hide   Copy Code

public class MyServiceBehaviorAttribute : Attribute, IServiceBehavior

{

public void AddBindingParameters(ServiceDescription serviceDescription,

ServiceHostBase serviceHostBase,

Collection<ServiceEndpoint> endpoints,

BindingParameterCollection bindingParameters)

{

Logger.Log("MyServiceBehaviorAttribute",

"AddBindingParameters", serviceDescription.Name);

}

public void ApplyDispatchBehavior(ServiceDescription serviceDescription,

ServiceHostBase serviceHostBase)

{

Logger.Log("MyServiceBehaviorAttribute",

"ApplyDispatchBehavior", serviceDescription.Name);

}

public void Validate(ServiceDescription serviceDescription,

ServiceHostBase serviceHostBase)

{

Logger.Log("MyServiceBehaviorAttribute",

"Validate", serviceDescription.Name);

}

}

**Second Extension**

Extending IOperationBehavior, this behavior‘s scope is limited on a method and we are using it here just to add the third extension. But you can modify it the way you want.

Hide   Copy Code

public interface IOperationBehavior

{

void AddBindingParameters(OperationDescription operationDescription,

BindingParameterCollection bindingParameters);

void ApplyClientBehavior(OperationDescription operationDescription,

ClientOperation clientOperation);

void ApplyDispatchBehavior(OperationDescription operationDescription,

DispatchOperation dispatchOperation);

void Validate(OperationDescription operationDescription);

}

AddBindingParameters: Pass data at runtime to bindings to support custom behavior.

ApplyClientBehavior: Implements a modification or extension of the client across an operation.

ApplyDispatchBehavior: Implements a modification or extension of the service across an operation.

Validate: Implement to confirm that the operation meets some intended criteria.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

public class MyOperationBehavior:Attribute, IOperationBehavior

{

public void AddBindingParameters(OperationDescription operationDescription,

System.ServiceModel.Channels.BindingParameterCollection bindingParameters)

{

Logger.Log("MyOperationBehavior",

"AddBindingParameters", operationDescription.Name);

}

public void ApplyClientBehavior(OperationDescription operationDescription,

System.ServiceModel.Dispatcher.ClientOperation clientOperation)

{

clientOperation.ParameterInspectors.Add(new MyParameterInspector());

Logger.Log("MyOperationBehavior",

"ApplyClientBehavior", operationDescription.Name);

}

public void ApplyDispatchBehavior(OperationDescription operationDescription,

System.ServiceModel.Dispatcher.DispatchOperation dispatchOperation)

{

dispatchOperation.ParameterInspectors.Add(new MyParameterInspector());

Logger.Log("MyOperationBehavior",

"ApplyDispatchBehavior", operationDescription.Name);

}

public void Validate(OperationDescription operationDescription)

{

Logger.Log("MyOperationBehavior", "Validate", operationDescription.Name);

}

}

**Third Extension**

Extending IparameterInspector Interface, it is a way of intercepting requests / responses after all the processing to extract those parameters from incoming messages (or before they’re packaged in outgoing messages) is done.

Hide   Copy Code

public interface IParameterInspector

{

void AfterCall(string operationName,

object[] outputs, object returnValue, object correlationState);

object BeforeCall(string operationName, object[] inputs);

}

AfterCall: Called after client calls are returned and before service responses are sent.

BeforeCall: Called before client calls are sent and after service responses are returned.

Hide   Copy Code

class MyParameterInspector:IParameterInspector

{

public void AfterCall(string operationName, object[] outputs,

object returnValue, object correlationState)

{

Logger.Log("MyParameterInspector", "AfterCall", operationName);

}

public object BeforeCall(string operationName, object[] inputs)

{

Logger.Log("MyParameterInspector", "BeforeCall", operationName);

return null;

}

}

Now we are almost done, the last step is to apply the new behaviors to the service and run it so that we can see the sequence of the calls.

Mind you, you can apply multiple behaviors on a service or on a method either by separating them by a comma or by adding another attribute like I show in the code below:

Hide   Copy Code

**[MyServiceBehaviorAttribute()]**

public class PasswordGenerator:IPasswordGenerator

{

[OperationBehavior(),**MyOperationBehavior**]

public string GeneratePassword()

{

return "You called GeneratePassword()";

}

[OperationBehavior()]

**[MyOperationBehavior]**

public string GeneratePassword(int length)

{

return "You called GeneratePassword(int length) overload";

}

Results

The results show the following:

1. First the Service behavior validates
2. Then the OperationBehavior validates for the first method
3. Then the OperationBehavior validates for the second method
4. Then Servicebehavior AddBinding parameters
5. Then Operation behavior calls AddbindingParameters for each method called
6. Then Servicebehavior ApplyDispatchBehavior
7. Then Operation behavior calls ApplyDispatchBehavior for each method called
8. Finally, it calls before Call and AfterCall for each method called

The result will be something like this:

Hide   Copy Code

Class MyServiceBehaviorAttribute called method Validate

with parameter PasswordGenerator @ 5/27/2013 10:32:26 PM

Class MyOperationBehavior called method Validate

with parameter GeneratePassword @ 5/27/2013 10:32:26 PM

Class MyOperationBehavior called method Validate

with parameter ParameterizedPasswordGenerator @ 5/27/2013 10:32:26 PM

Class MyServiceBehaviorAttribute called method AddBindingParameters

with parameter PasswordGenerator @ 5/27/2013 10:32:26 PM

Class MyOperationBehavior called method AddBindingParameters

with parameter GeneratePassword @ 5/27/2013 10:32:26 PM

Class MyOperationBehavior called method AddBindingParameters

with parameter ParameterizedPasswordGenerator @ 5/27/2013 10:32:26 PM

Class MyServiceBehaviorAttribute called method ApplyDispatchBehavior

with parameter PasswordGenerator @ 5/27/2013 10:32:26 PM

Class MyOperationBehavior called method ApplyDispatchBehavior

with parameter GeneratePassword @ 5/27/2013 10:32:27 PM

Class MyOperationBehavior called method ApplyDispatchBehavior

with parameter ParameterizedPasswordGenerator @ 5/27/2013 10:32:27 PM

Class MyParameterInspector called method BeforeCall

with parameter GeneratePassword @ 5/27/2013 10:32:27 PM

Class MyParameterInspector called method AfterCall

with parameter GeneratePassword @ 5/27/2013 10:32:27 PM

Class MyParameterInspector called method BeforeCall

with parameter ParameterizedPasswordGenerator @ 5/27/2013 10:32:27 PM

Class MyParameterInspector called method AfterCall

with parameter ParameterizedPasswordGenerator @ 5/27/2013 10:32:27 PM

Points of Interest

The bottom-line is that extensibility is a great feature of WCF that added much of the flexibility to it and made it such a powerful tool that you can anticipate a lot of problems, and control the behavior of the service.

References

1. [WCF Extensibility](http://blogs.msdn.com/b/carlosfigueira/archive/2011/03/14/wcf-extensibility.aspx)
2. [IServiceBehavior Interface](http://msdn.microsoft.com/en-us/library/system.servicemodel.description.iservicebehavior.aspx)
3. [IParameterInspector Interface](http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.iparameterinspector.aspx)
4. [WCF Extensibility – IParameterInspector](http://blogs.msdn.com/b/endpoint/archive/2011/04/28/wcf-extensibility-iparameterinspector.aspx)



License

This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)