FREE eBook

LEARNING

wcf

Free unaffiliated eBook created from
Stack Overflow contributors.

#wcf

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: wcf

It is an unofficial and free wcf ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official wcf.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with wcf

Remarks

This section provides an overview of what wcf is, and why a developer might want to use it.

It should also mention any large subjects within wcf, and link out to the related topics. Since the Documentation for wcf is new, you may need to create initial versions of those related topics.

Examples

WCF Restful Service Demo

svc

```
 public class WCFRestfulService : IWCFRestfulService
    {
       public string GetServiceName(int Id)
        {
            return "This is a WCF Restful Service";
        }
    }
```

Interface

```
[ServiceContract(Name = "WCRestfulService ")]
public interface IWCFRestfulService
{
    [OperationContract]
    [WebInvoke(Method = "GET", ResponseFormat = WebMessageFormat.Json, BodyStyle =
WebMessageBodyStyle.Wrapped, UriTemplate = "GetServiceName?Id={Id}")]
    string GetServiceName(int Id);
}
```

svc Markup (Right Click on svc file & click view MarkUp)

```
<%@ ServiceHost Language="C#" Debug="true" Service="NamespaceName.WCFRestfulService"
CodeBehind="WCFRestfulService.svc.cs" %>
```

Web Config

```
<services>
      <service name="NamespaceName.WCFRestfulService"
behaviorConfiguration="ServiceBehaviour">
        <endpoint address="" binding="webHttpBinding"
contract="NamespaceName.IWCFRestfulService" behaviorConfiguration="web"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
```

```
        <behavior name="ServiceBehaviour">
          <!-- To avoid disclosing metadata information, set the values below to false before
deployment -->
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true"/>
          <!-- To receive exception details in faults for debugging purposes, set the value
below to true.  Set to false before deployment to avoid disclosing exception information -->
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
      </serviceBehaviors>
      <endpointBehaviors>
        <behavior name="web">
          <webHttp/>
        </behavior>
      </endpointBehaviors>
    </behaviors>
```

Now Simply Run the Service or host in a port. And access the service using " http://hostname/WCFRestfulService/GetServiceName?Id=1"

Simple WCF service

Minimal requirements for WCF service is one ServiceContract with one OperationContract.

Service contract:

```
[ServiceContract]
public interface IDemoService
{
    [OperationContract]
    CompositeType SampleMethod();
}
```

Service contract implementation:

```
public class DemoService : IDemoService
{
    public CompositeType SampleMethod()
    {
        return new CompositeType { Value = "foo", Quantity = 3 };
    }
}
```

Configuration file:

```
<?xml version="1.0"?>
<configuration>
  <appSettings>
    <add key="aspnet:UseTaskFriendlySynchronizationContext" value="true" />
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5.2" />
    <httpRuntime targetFramework="4.5.2"/>
  </system.web>
  <system.serviceModel>
    <behaviors>
```

```
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="false"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
multipleSiteBindingsEnabled="true" />
  </system.serviceModel>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true"/>
    <directoryBrowse enabled="true"/>
  </system.webServer>
</configuration>
```

DTO:

```
[DataContract]
public class CompositeType
{
    [DataMember]
    public string Value { get; set; }

    [DataMember]
    public int Quantity { get; set; }
}
```

Read Getting started with wcf online: https://riptutorial.com/wcf/topic/992/getting-started-with-wcf

Chapter 2: DataContractSerializer is an Opt-In and Opt-Out serializer.

Introduction

actually its so simple: Opt-In approach says properties that are considered to be part of DataContract must be explicitly marked otherwise will be ignore, while Opt-Out means all of the properties will be assumed to be part of the DataContract unless marked explicitly.

Examples

What is opt in serializer

```
/// <summary>
/// Defines a student.
/// </summary>
[DataContract]
public class Student
{
    /// <summary>
    /// Gets or sets the student number.
    /// </summary>
    [DataMember]
    public string StudentNumber { get; set; }

    /// <summary>
    /// Gets or sets the first name.
    /// </summary>
    [DataMember]
    public string FirstName { get; set; }

    /// <summary>
    /// Gets or sets the last name.
    /// </summary>
    [DataMember]
    public string LastName { get; set; }

    /// <summary>
    /// Gets or sets the marks obtained.
    /// </summary>
    public int MarksObtained { get; set; }
}

/// <summary>
/// A service that provides the operations for students.
/// </summary>
[ServiceContract]
public interface IStudentService
{
    //Service contract code here.
}
```

In above code `StudentNumber`, `FirstName`, `LastName` properties of `Student` class are explicitly marked with `DataMember` attribute as oppose to `MarksObtained`, so `MarksObtained` will be ignored. From ignored it means that `MarksObtained` wont be the part of data going across the wire to / from this service.

What is opt out serializer

Below code represents an example of Opt-Out approach using `Serializable` and `NonSerialized` attributes.

```
/// <summary>
/// Represents a student.
/// </summary>
[Serializable]
public class Student
{
    /// <summary>
    /// Gets or sets student number.
    /// </summary>
    public string StudentNumber { get; set; }

    /// <summary>
    /// Gets or sets first name.
    /// </summary>
    public string FirstName { get; set; }

    /// <summary>
    /// Gets or sets last name.
    /// </summary>
    public string LastName { get; set; }

    /// <summary>
    /// Gets or sets marks obtained.
    /// </summary>
    [NonSerialized]
    public string MarksObtained { get; set; }
}

/// <summary>
/// A service that provides the operations for student.
/// </summary>
[ServiceContract]
public interface IStudentService
{
    //Service contract code here. Example given.

    /// <summary>
    /// Adds a student into the system.
    /// </summary>
    /// <param name="student">Student to be added.</param>
    [OperationContract]
    void AddStudent(Student student);
}
```

In above example, we explicitly marked `MarksObtained` property as `[NonSerialized]` attribute, so it will be ignored except the others. Hope this helps!

Read DataContractSerializer is an Opt-In and Opt-Out serializer. online:

https://riptutorial.com/wcf/topic/9588/datacontractserializer-is-an-opt-in-and-opt-out-serializer-

Chapter 3: Handling exceptions

Remarks

Further reading

More about FaultException: MSDN FaultException

Examples

Showing additional information when an exception occurs

It's important to handle exceptions in your service. When developing the service, you can set the WCF to provide more detailed information, adding this tag to configuration file, usually Web.config:

```
 <serviceDebug includeExceptionDetailInFaults="true"/>
```

This tag must be placed within the serviceBehavior tag, usually like this:

```
<system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
</system.serviceModel>
```

Example of detailed infomation:

> Server stack trace: em System.ServiceModel.Channels.ServiceChannel.HandleReply(ProxyOperationRuntime operation, ProxyRpc& rpc) em System.ServiceModel.Channels.ServiceChannel.Call(String action, Boolean oneway, ProxyOperationRuntime operation, Object[] ins, Object[] outs, TimeSpan timeout) em System.ServiceModel.Channels.ServiceChannelProxy.InvokeService(IMethodCallMessage methodCall, ProxyOperationRuntime operation) em System.ServiceModel.Channels.ServiceChannelProxy.Invoke(IMessage message)
>
> Exception rethrown at [0]: em System.Runtime.Remoting.Proxies.RealProxy.HandleReturnMessage(IMessage reqMsg, IMessage retMsg) em System.Runtime.Remoting.Proxies.RealProxy.PrivateInvoke(MessageData& msgData, Int32 type) em IMyService.GetDataOperation(RequestObterBeneficiario request) em MyServiceClient.GetDataOpration(RequestData request)

This will return to the client detailed information. **During development** this can help out, but when your service **goes to production**, you will no longer keep this because your service can send sensitive data, like your database name or configuration.

Throwing a user-friendly message with FaultException

You can handle exceptions and throw a most friendly message like 'Unavailable service' throwing a FaultException:

```
try
{
    // your service logic here
}
catch (Exception ex)
{
    // You can do something here, like log the original exception
    throw new FaultException("There was a problem processing your request");
}
```

In your client, you can easily get the message:

```
try
{
    // call the service
}
catch (FaultException faultEx)
{
   var errorMessage = faultEx.Message;
   // do something with error message
}
```

You can distinguish the handled exception from other exceptions, like a network error, adding other catches to your code:

```
try
{
    // call the service
}
catch (FaultException faultEx)
{
   var errorMessage = faultEx.Message;
   // do something with error message
}
catch (CommunicationException commProblem)
{
    // Handle the communication error, like trying another endpoint service or logging
}
```

Throwing a FaultException with a Fault Code

The FaultException can also includes a **FaultCode**, that is a string data you can use to pass additional information, so the client can be able to distinguish different exceptions:

```
try
{
    // your service logic here
}
catch (Exception ex)
{
   throw new FaultException("There was a problem processing your request",
      new FaultCode(("01"));
}
```

Getting the FaultCode:

```
try
{
    // call the service
}
catch (FaultException faultEx)
{
   switch (faultEx.Code.Name)
   {
    case "01":
       // do something
       break;
    case "02":
        // do another something
        break
   }
}
```

Decoupling ErrorHandlerAttribute registering from the service implementation

To decouple and reuse the same error logging code for **all** your services you need two boilerplate classes and tuck them away in a library somewhere.

ErrorhandlerAttribute implementing IServiceBehavior. FaultErrorhandler implementing IErrorhandler which logs all the exceptions.

```
[AttributeUsage(AttributeTargets.Class)]
public class ErrorHandlerAttribute : Attribute, IServiceBehavior
{
    Type mErrorType;
    public ErrorHandlerAttribute(Type t)
    {
        if (!typeof(IErrorHandler).IsAssignableFrom(t))
            throw new ArgumentException("Type passed to ErrorHandlerAttribute constructor must
inherit from IErrorHandler");
        mErrorType = t;
    }

    public void AddBindingParameters(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase, Collection<ServiceEndpoint> endpoints, BindingParameterCollection
bindingParameters)
    {
    }

    public void ApplyDispatchBehavior(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase)
```

```
    {
        foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
        {
            dispatcher.ErrorHandlers.Add((IErrorHandler)Activator.CreateInstance(mErrorType));
        }
    }

    public void Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase)
    {
    }
}


class FaultErrorHandler : IErrorHandler
{
    public bool HandleError(Exception error)
    {
        // LOG ERROR
        return false; // false so the session gets aborted
    }

    public void ProvideFault(Exception error, MessageVersion version, ref Message fault)
    {
    }
}
```

Then in your service implementation add the ErrorHandler Attribute passing in the Type instance of FaultErrorHandler. ErrorHandler will construct an instance from that type on which HandleError is called.

```
[ServiceBehavior]
[ErrorHandler(typeof(FaultErrorHandler))]
public class MyService : IMyService
{
}
```

Using a custom error logging framework

It is sometimes useful to integrate a custom error logging framework to ensure all exceptions are logged.

```
[ServiceContract]
[ErrorHandler]
public interface IMyService
{

}

[AttributeUsage(AttributeTargets.Interface)]
public class CustomErrorHandler : Attribute, IContractBehavior, IErrorHandler
{
    public bool HandleError(Exception error)
    {
        return false;
    }
```

```
    public void ProvideFault(Exception error, MessageVersion version, ref Message fault)
    {
        if (error == null)
        {
            return;
        }

        //my custom logging framework
    }

    public void ApplyDispatchBehavior(ContractDescription contractDescription, ServiceEndpoint
endpoint, DispatchRuntime dispatchRuntime)
    {
        dispatchRuntime.ChannelDispatcher.ErrorHandlers.Add(this);
    }

    public void ApplyClientBehavior(ContractDescription contractDescription, ServiceEndpoint
endpoint,
        ClientRuntime clientRuntime)
    {

    }

    public void AddBindingParameters(ContractDescription contractDescription, ServiceEndpoint
endpoint,
        BindingParameterCollection bindingParameters)
    {

    }

    public void Validate(ContractDescription contractDescription, ServiceEndpoint endpoint)
    {

    }
}
```

Read Handling exceptions online: https://riptutorial.com/wcf/topic/5557/handling-exceptions

Chapter 4: How to use a dependency injection container with a WCF service

Examples

How to Configure a WCF Service to Use a Dependency Injection Container (Castle Windsor)

This example has two parts - some boilerplate steps for adding Castle Windsor to your WCF service, and then a simple, concrete example to show how we configure and use Windsor's container.

That makes the example a little bit long. If you already understand using a DI container then you likely only care about the boilerplate steps. If using a DI container is unfamiliar then it takes a little more - seeing the whole thing work from end to end - before it makes sense.

Boilerplate Steps

1. Add the Castle Windsor WCF integration facility Nuget package to your WCF service application. This will add references to Castle Windsor as well as some components specifically for WCF services.

2. Add a Global Application Class (global.asax) to your project: Add > New Item > Visual C# > Web > Global Application Class.
   The code that configures your container must be called from the `Application_Start` method. To keep it organized we can put all of our container configuration in a separate class. We don't have to. It doesn't matter. You'll see this done differently in different examples. What matters is that it's called from the `Application_Start` method because that's your *composition root* - where the application starts. The idea is to configure your container when the application starts and then never touch it directly again. It just stays in the background doing its job.

3. Create a class to configure the container. This does two things:

   - `ContainerInstance.AddFacility<WcfFacility>()` tells Windsor's WCF code to use this particular container when creating instances of your WCF services.
   - `ContainerInstance.Install(FromAssembly.This())` tells Windsor to scan `This` assembly (in other words, your WCF project) looking for classes that implement `IWindsorInstaller`. Those classes will provide instructions to tell your container how to resolve dependencies. (We'll create one a few steps later.)

```
public static class Container
{
    private static readonly IWindsorContainer ContainerInstance = new WindsorContainer();

    public static void Configure()
```

```
    {
        ContainerInstance.AddFacility<WcfFacility>();
        ContainerInstance.Install(FromAssembly.This());
    }
}
```

4. Call this method from `Application_Start` in your global.asax:

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        Container.Configure();
    }
}
```

5. Create an installer. This is just a class that implements `IWindsorInstaller`. This one is empty. It doesn't do anything. We'll add to this class in a few steps. When we call `ContainerInstance.Install(FromAssembly.This())`, `ContainerInstance` gets passed to the `Install` method so that we can register dependencies with the container.

```
public class WindsorInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        // Nothing here yet!
    }
}
```

6. In the markup for any WCF service you create, add this directive. This indicates that the application will use Windsor's WCF facility to create instances of the service, which in turn means that in can inject dependencies when an instance is created.

```
Factory="Castle.Facilities.WcfIntegration.DefaultServiceHostFactory,
Castle.Facilities.WcfIntegration"
```

That ends the boilerplate steps for setting up the service to use a Castle Windsor dependency injection container.

But the example isn't complete unless we configure at least one WCF service for dependency injection. The rest of this isn't boilerplate, just a simple, concrete example.

1. Create a new WCF service called `GreetingService.svc`.
2. Edit the markup. It should look like this:

```
<%@ ServiceHost Language="C#" Debug="true"
    Service="WcfWindsorDocumentation.GreetingService"
    CodeBehind="GreetingService.svc.cs"
    Factory="Castle.Facilities.WcfIntegration.DefaultServiceHostFactory,
Castle.Facilities.WcfIntegration"
 %>
```

3. Replace `IGreetingService` (the service contract) with this:

```
[ServiceContract]
public interface IGreetingService
{
    [OperationContract]
    string GetGreeting();
}
```

4. Replace `GreetingService` (in GreetingService.svc.cs) with this code. Notice that the constructor requires an instance of `IGreetingProvider` which we'll need our container to inject.

```
public class GreetingService : IGreetingService
{
    private readonly IGreetingProvider _greetingProvider;

    public GreetingService(IGreetingProvider greetingProvider)
    {
        _greetingProvider = greetingProvider;
    }

    public string GetGreeting()
    {
        return _greetingProvider.GetGreeting();
    }
}
```

5. Add this implementation of `IGreetingProvider`. It also has a few dependencies of its own which we'll need the container to supply. The specifics of these classes aren't too important. They're just to create something easy to follow.

```
public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IComputerNameProvider
{
    string GetComputerName();
}

public class ComputerNameGreetingProvider : IGreetingProvider
{
    private readonly IComputerNameProvider _computerNameProvider;

    public ComputerNameGreetingProvider(IComputerNameProvider computerNameProvider)
    {
        _computerNameProvider = computerNameProvider;
    }

    public string GetGreeting()
    {
        return string.Concat("Hello from ", _computerNameProvider.GetComputerName());
    }
}

public class EnvironmentComputerNameProvider : IComputerNameProvider
```

```
{
    public string GetComputerName()
    {
        return System.Environment.MachineName;
    }
}
```

6. Now we have all of the classes we need. All that's left is to register dependencies with our container. In other words, we're going to tell the container what classes it needs to create so it knows how to "build" an instance of `GreetingService`. This code goes in our implementation of `IWindsorInstaller` (step 5 of the boilerplate code.)

```
public class WindsorInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(
            Component.For<IGreetingService, GreetingService>(),
            Component.For<IGreetingProvider, ComputerNameGreetingProvider>(),
            Component.For<IComputerNameProvider, EnvironmentComputerNameProvider>());
    }
}
```

This tells the container:

- It's responsible for creating `GreetingService` when needed.
- When it tries to create `GreetingService` it's going to need an `IGreetingProvider`. When it needs that it should create a `ComputerNameGreetingProvider`.
- When it tries to create `ComputerNameGreetingProvider`, that class requires an `IComputerNameProvider`. It should create an instance of `EnvironmentComputerNameProvider` to fulfill that need.

If, somewhere in the process of creating `GreetingService` or one of its dependencies, it comes across a requirement for a dependency that we haven't registered, it will let us know with a helpful exception, like this.

> Missing dependency.
> Component WcfWindsorDocumentation.ComputerNameGreetingProvider has a dependency on WcfWindsorDocumentation.IComputerNameProvider, which could not be resolved.
> Make sure the dependency is correctly registered in the container as a service, or provided as inline argument.

That means that something dependend on `IComputerNameProvider` but there was nothing registered to fulfill that dependency.

This just gets the ball rolling. There's much more to correctly configuring and using a dependency injection container for real-world scenarios. This example only covers what's specific to adding Windsor to a WCF service application. If you use a different container like Autofac or Unity you'll find that while the syntax and details vary, in principle they do the same things and you'll easily spot the similarities.

Read How to use a dependency injection container with a WCF service online: https://riptutorial.com/wcf/topic/5509/how-to-use-a-dependency-injection-container-with-a-wcf-service

Chapter 5: How to: Disable/Enable WCF tracing in C# application code

Examples

One way: use a custom listener defined in your C# code

It took me a while to get this right, so I decided to share a solution because it might save someone else several days of trial and error.

The problem: I want to be able to enable/disable WCF tracing in my C# .NET application and choose the trace output filename. I don't want users editing the .config file, there's too much room for error there.

Here is one solution.

The application's .config file:

```
<?xml version="1.0"?>
<configuration>
  <system.diagnostics>
    <trace autoflush="true"/>
    <sources>
      <source name="System.ServiceModel" switchValue="All">
        <listeners>
          <add name="MyListener"/>
        </listeners>
      </source>
      <source name="System.ServiceModel.MessageLogging" switchValue="All">
        <listeners>
          <add name="MyListener"/>
        </listeners>
      </source>
      <source name="System.ServiceModel.Activation" switchValue="All">
        <listeners>
          <add name="MyListener"/>
        </listeners>
      </source>
      <source name="System.IdentityModel" switchValue="All">
        <listeners>
          <add name="MyListener"/>
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="MyListener" type="MyNamespace.MyXmlListener, MyAssembly"/>
    </sharedListeners>
  </system.diagnostics>
  <system.serviceModel>
    <diagnostics wmiProviderEnabled="true">
      <messageLogging
        logEntireMessage="true"
        logMalformedMessages="true"
```

```
        logMessagesAtServiceLevel="true"
        logMessagesAtTransportLevel="true"
        maxMessagesToLog="1000"
        maxSizeOfMessageToLog="8192"/>
    </diagnostics>
  </system.serviceModel>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

And the C# code:

```
using System;
using System.IO;
using System.Diagnostics;
namespace MyNamespace
{
    public class MyXmlListener : XmlWriterTraceListener
    {
        public static String TraceOutputFilename = String.Empty;

        public static Stream MakeOutputStream()
        {
            if (String.IsNullOrWhiteSpace(TraceOutputFilename))
                return Stream.Null;

            return new FileStream(TraceOutputFilename, FileMode.Create);
        }

        public MyXmlListener ()
            : base(MakeOutputStream())
        { }
    }
}
```

To enable WCF tracing to a file, set TraceOutputFilename before the WCF object is created:

```
MyXmlListener.TraceOutputFilename = "trace.svclog";
```

I've gotten great benefits from this forum, I hope this post pays some of it forward.

Getting the "type" right in the .config file was much more challenging than it should have been, see Specifying Fully Qualified Type Names for setting the "type" correctly in a .config file.

Read How to: Disable/Enable WCF tracing in C# application code online: https://riptutorial.com/wcf/topic/5559/how-to--disable-enable-wcf-tracing-in-csharp-application-code

Chapter 6: Serialization

Examples

Serialization in WCF

> Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Microsoft page Serialization

The following example demonstrates Serialization in WCF:

```
[ServiceContract(Namespace="http://Microsoft.ServiceModel.Samples")]
public interface IPerson
{

    [OperationContract]
    void Add(Person person);

    [DataContract]
    public class Person
    {
        private int id;

        [DataMember]
        public int Age{ set; get;}
    }
}
```

1. `[DataContract]` Attribute is used with the classes. Here it is decorated with `Person` class.
2. `[OperationContract]` is used for methods. Here it is decorated with `Add` method.
3. `[DataMember]` Attribute is used with the properties. those who are decorated with `[DataMember]` Attributes only those will be available for the proxy to access. Here we have 2 properties in that `id` is not accessible and `Age` is accessible.
4. `[DataMember]` Attribute is handy when you don't want to show private fields to outside world and only want to show public properties.
5. With `[DataMember]` Attribute you have some properties stick to it. they are as follows

   Properties of DataMember

a. `IsRequired`can be used like this `[DataMember(IsRequired=true)]`

b. `Name` can be used like this `[DataMember(Name="RegistrationNo")]`

c. `order` can be used like this `[DataMember(order=1)]`

Without specifying attributes, we won't be able to access the class/ method/ property in projects

whom we work with (this exmple wcf service interface).

The way these attributes make the code accessible through individual projects at runtime is called "Serialization".

- With WCF you can communicate with other projects, applications or any other software using serialization, **without** all the work of setting up the endpoints, creating streams manually and maintaining them. Not to mention converting all of the data into bytes and vice versa.

Read Serialization online: https://riptutorial.com/wcf/topic/2491/serialization

Chapter 7: Tracing

Examples

Tracing Settings

WCF tracing is built on top of System.Diagnostics. To use tracing, you should define trace sources in the configuration file or in code.

Tracing is not enabled by default. To activate tracing, you must create a trace listener and set a trace level other than "Off" for the selected trace source in configuration; otherwise, WCF does not generate any traces.

The following example shows how to enable message logging and specify additional options:

```
<configuration>
 <system.diagnostics>
  <sources>
    <source name="System.ServiceModel"
            switchValue="All"
            propagateActivity="true" >
      <listeners>
        <add name="wcf_trace"/>
      </listeners>
    </source>
    <source name="System.ServiceModel.MessageLogging">
      <listeners>
        <add name="wcf_trace"/>
      </listeners>
    </source>
  </sources>
  <sharedListeners>
    <add name="wcf_trace"
         type="System.Diagnostics.XmlWriterTraceListener"
         initializeData="c:\temp\wcf_trace\DistanceService.svclog" />
  </sharedListeners>
 </system.diagnostics>

 <system.serviceModel>
  <diagnostics wmiProviderEnabled="true">
      <messageLogging
           logEntireMessage="true"
           logMalformedMessages="true"
           logMessagesAtServiceLevel="true"
           logMessagesAtTransportLevel="true"
           maxMessagesToLog="3000" />
  </diagnostics>
 </system.serviceModel>
</configuration>
```

initializeData specifies the name of output file for that listener. If you do not specify a file name, a random file name is generated based on the listener type used.

Logging Levels and Options

- Service Level

Messages logged at this layer are about to enter (on receiving) or leave (on sending) user code. If filters have been defined, only messages that match the filters are logged. Otherwise, all messages at the service level are logged. Infrastructure messages (transactions, peer channel, and security) are also logged at this level, except for Reliable Messaging messages. On streamed messages, only the headers are logged. In addition, secure messages are logged decrypted at this level.

- Transport Level

Messages logged at this layer are ready to be encoded or decoded for or after transportation on the wire. If filters have been defined, only messages that match the filters are logged. Otherwise, all messages at the transport layer are logged. All infrastructure messages are logged at this layer, including reliable messaging messages. On streamed messages, only the headers are logged. In addition, secure messages are logged as encrypted at this level, except if a secure transport such as HTTPS is used.

- Malformed Level

Malformed messages are messages that are rejected by the WCF stack at any stage of processing. Malformed messages are logged as-is: encrypted if they are so, with non-proper XML, and so on. maxSizeOfMessageToLog defined the size of the message to be logged as CDATA. By default, maxSizeOfMessageToLog is equal to 256K. For more information about this attribute, see the Other Options section.

If you want to disable the trace source, you should use the logMessagesAtServiceLevel, logMalformedMessages, and logMessagesAtTransportLevel attributes of the messageLogging element instead. You should set all these attributes to false.

Read Tracing online: https://riptutorial.com/wcf/topic/1931/tracing

Chapter 8: WCF - Http and Https on SOAP and REST

Examples

Sample web.config

here is a sample `web.config` in order to have both `http` and `https` support.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <appSettings>
    <add key="aspnet:UseTaskFriendlySynchronizationContext" value="true" />
  </appSettings>

  <system.web>
    <compilation debug="true" targetFramework="4.5.2" />
    <httpRuntime targetFramework="4.5.2" />
  </system.web>

  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="SoapBinding" />
      </basicHttpBinding>
      <basicHttpsBinding>
        <binding name="SecureSoapBinding" />
      </basicHttpsBinding>

      <webHttpBinding>
        <binding name="RestBinding" />
        <binding name="SecureRestBinding">
          <security mode="Transport" />
        </binding>
      </webHttpBinding>

      <mexHttpBinding>
        <binding name="MexBinding" />
      </mexHttpBinding>
      <mexHttpsBinding>
        <binding name="SecureMexBinding" />
      </mexHttpsBinding>
    </bindings>

    <client />

    <services>
      <service behaviorConfiguration="ServiceBehavior" name="Interface.Core">
        <endpoint address="soap" binding="basicHttpBinding" bindingConfiguration="SoapBinding"
name="Soap" contract="Interface.ICore" />
        <endpoint address="soap" binding="basicHttpsBinding"
bindingConfiguration="SecureSoapBinding" name="SecureSoap" contract="Interface.ICore" />
        <endpoint address="" behaviorConfiguration="Web" binding="webHttpBinding"
bindingConfiguration="RestBinding" name="Rest" contract="Interface.ICore" />
```

```
        <endpoint address="" behaviorConfiguration="Web" binding="webHttpBinding"
bindingConfiguration="SecureRestBinding" name="SecureRest" contract="Interface.ICore" />
        <endpoint address="mex" binding="mexHttpBinding" bindingConfiguration="MexBinding"
name="Mex" contract="IMetadataExchange" />
        <endpoint address="mex" binding="mexHttpsBinding"
bindingConfiguration="SecureMexBinding" name="SecureMex" contract="IMetadataExchange" />
      </service>
    </services>

    <behaviors>
      <endpointBehaviors>
        <behavior name="Web">
          <webHttp helpEnabled="true" defaultBodyStyle="Bare"
defaultOutgoingResponseFormat="Json" automaticFormatSelectionEnabled="true" />
        </behavior>
      </endpointBehaviors>

      <serviceBehaviors>
        <behavior name="ServiceBehavior">
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    <protocolMapping>
      <add binding="basicHttpsBinding" scheme="https" />
    </protocolMapping>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
multipleSiteBindingsEnabled="true" />
  </system.serviceModel>

  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true" />
    <directoryBrowse enabled="false" />
  </system.webServer>

</configuration>
```

Read WCF - Http and Https on SOAP and REST online: https://riptutorial.com/wcf/topic/9604/wcf---http-and-https-on-soap-and-rest

Chapter 9: WCF Restful Service

Examples

WCF Resful Service

```
[ServiceContract]
public interface IBookService
{
    [OperationContract]
    [WebGet]
    List<Book> GetBooksList();

    [OperationContract]
    [WebGet(UriTemplate  = "Book/{id}")]
    Book GetBookById(string id);

    [OperationContract]
    [WebInvoke(UriTemplate = "AddBook/{name}")]
    void AddBook(string name);

    [OperationContract]
    [WebInvoke(UriTemplate = "UpdateBook/{id}/{name}")]
    void UpdateBook(string id, string name);

    [OperationContract]
    [WebInvoke(UriTemplate = "DeleteBook/{id}")]
    void DeleteBook(string id);
}
```

Implementing the Service

Now the service implementation part will use the entity framework generated context and entities to perform all the respective operations.

```
public class BookService : IBookService
{
    public List<Book> GetBooksList()
    {
        using (SampleDbEntities entities = new SampleDbEntities())
        {
            return entities.Books.ToList();
        }
    }

    public Book GetBookById(string id)
    {
        try
        {
            int bookId = Convert.ToInt32(id);

            using (SampleDbEntities entities = new SampleDbEntities())
            {
                return entities.Books.SingleOrDefault(book => book.ID == bookId);
            }
```

```
        }
        catch
        {
            throw new FaultException("Something went wrong");
        }
    }

    public void AddBook(string name)
    {
        using (SampleDbEntities entities = new SampleDbEntities())
        {
            Book book = new Book { BookName = name };
            entities.Books.AddObject(book);
            entities.SaveChanges();
        }
    }

    public void UpdateBook(string id, string name)
    {
        try
        {
            int bookId = Convert.ToInt32(id);

            using (SampleDbEntities entities = new SampleDbEntities())
            {
                Book book = entities.Books.SingleOrDefault(b => b.ID == bookId);
                book.BookName = name;
                entities.SaveChanges();
            }
        }
        catch
        {
            throw new FaultException("Something went wrong");
        }
    }

    public void DeleteBook(string id)
    {
        try
        {
            int bookId = Convert.ToInt32(id);

            using (SampleDbEntities entities = new SampleDbEntities())
            {
                Book book = entities.Books.SingleOrDefault(b => b.ID == bookId);
                entities.Books.DeleteObject(book);
                entities.SaveChanges();
            }
        }
        catch
        {
            throw new FaultException("Something went wrong");
        }
    }
}
```

Restful WCF service Configuration

Now from the ServiceContract perspective the service is ready to serve the REST request but to access this service over rest we need to do some changes in the service behavior and binding too.

To make the service available over REST protocol the binding that needs to be used is the webHttpBinding. Also, we need to set the endpoint's behavior configuration and define the webHttp parameter in the endpointBehavior. So our resulting configuration will look something like:

```
<system.serviceModel>
  <services>
    <service name="WcfRestSample.BookService">
      <endpoint address="" behaviorConfiguration="restfulBehavior"
        binding="webHttpBinding" bindingConfiguration="" contract="WcfRestSample.IBook
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost/bookservice" />
        </baseAddresses>
      </host>
    </service>
  </services>
  <behaviors>
    <endpointBehaviors>
      <behavior name="restfulBehavior">
        <webHttp />
      </behavior>
    </endpointBehaviors>
```

Read WCF Restful Service online: https://riptutorial.com/wcf/topic/3041/wcf-restful-service

Chapter 10: WCF Security

Examples

WCF Security

Security is a critical piece of any programming technology or framework for implementing service - oriented applications

WCF has been built from the ground up for providing the necessary security infrastructure at the message and service level.

In the following sections, you see how to use many of the available security settings in WCF, and some common deployment scenarios.

For message protection, WCF supports the two traditional security models, transport security and message security.

The bindings, in addition to specifying the communication protocol and encoding for the services, will also allow you to confi gure the message protection settings and the authentication schema.

Default Security Settings in WCF:

BINDING	SETTINGS
WsHttpBinding	Message Security with Windows Authentication
BasicHttpBinding	No Security
WsFederationHttpBinding	Message Security with Federated Authentication
NetTcpBinding	Transport Security with Windows Authenticatio
NetNamedPipeBinding	Transport Security with Windows Authentication
NetMsmqBinding	Transport Security with Windows Authentication

consider following example:

```
<wsHttpBinding >
  <binding name=”UsernameBinding” >
   <security mode=”Message” >
     <message clientCredentialType=”UserName”/ >
   </security >
  </binding >
</wsHttpBinding >
```

In this example, the service has been confi gured with message security and the username

security token profi le. The rest of the security settings for the binding take the default values.

Security Mode

The security mode setting determines two fundamental security aspects for any service: the security model for message protection and the supported client authentication schema.

Security MODE	Description
None	The service is available for anyone, and the messages are not protected as they go through the transport. When this mode is used, the service is vulnerable to any kind of attack.
Transport	Uses the transport security model for authenticating clients and protecting the messages. This mode provides the advantages and disadvantages discussed in transport security.
Message	Uses the message security model for authenticating clients and protecting the messages. This mode provides the advantages and disadvantages discussed in message security.
Both	Uses the transport security and message security models at the same time for authenticating the service consumers and protecting the messages. This mode is only supported by the MSMQ bindings and requires the same credentials at both levels.
TransportWithMessageCredentials	The message protection is provided by transport, and the credentials for authenticating the service consumers travel as part of the message. This mode provides the flexibility of using any of the credentials or token types supported in message authentication while the service authentication and message protection is performed at transport level.
TransportCredentialOnly	Uses transport security for authenticating clients The service is not authenticated, and the messages,including the client credentials, go as plain text through the transport. This security mode can be useful forscenarios where the kind of information transmitted between the client and the service is not sensitive, although the credentials also get exposed to anyone.

Configure the WsHttpBinding to use transport security with Basic Authentication

```
<bindings >
  <wsHttpBinding >
   <binding name="mybinding" >
    <security mode="Transport" >
     <transport clientCredentialType="Basic"/ >
    </security >
   </binding >
  </wsHttpBinding >
</bindings >
```

Read WCF Security online: https://riptutorial.com/wcf/topic/6021/wcf-security

Chapter 11: Your first service

Examples

1. Your first service and host

Create an interface decorated with a ServiceContract attribute and member functions decorated with the OperationContract attribute.

```
namespace Service
{
    [ServiceContract]
    interface IExample
    {
        [OperationContract]
        string Echo(string s);
    }
 }
```

Create a concrete class implementing this interface and you have the service.

```
namespace Service
{
    public class Example : IExample
    {
        public string Echo(string s)
        {
            return s;
        }
    }
}
```

Then create the host where the service will run from. This can be any type of application. Console, service, GUI application or web server.

```
namespace Console
{
    using Service;

    class Console
    {
        Servicehost mHost;

        public Console()
        {
            mHost = new ServiceHost(typeof(Example));
        }

        public void Open()
        {
            mHost.Open();
        }
```

```
        public void Close()
        {
            mHost.Close();
        }

        public static void Main(string[] args)
        {
            Console host = new Console();

            host.Open();

            Console.Readline();

            host.Close();
        }
    }
}
```

The ServiceHost class will read the configuration file to initialize the service.

```
<system.serviceModel>
  <services>
    <service name="Service.Example">
      <endpoint name="netTcpExample" contract="Service.IExample" binding="netTcpBinding" />
      <host>
        <baseAddresses>
          <add baseAddress="net.tcp://localhost:9000/Example" />
        </baseAddresses>
      </host>
    </service>
  </services>
</system.serviceModel>
```

First Service Client

Let's say you have a service the same as the one defined in the "First service and host" example.

To create a client, define the client configuration section in the system.serviceModel section of your client configuration file.

```
<system.serviceModel>
  <services>
    <client name="Service.Example">
      <endpoint name="netTcpExample" contract="Service.IExample" binding="netTcpBinding"
address="net.tcp://localhost:9000/Example" />
    </client>
  </services>
</system.serviceModel>
```

Then copy the service contract definition from the service:

```
namespace Service
{
    [ServiceContract]
    interface IExample
    {
```

```
        [OperationContract]
        string Echo(string s);
    }
}
```

(NOTE: You could also consume this via adding a binary reference instead to the assembly containing the service contract instead.)

Then you can create the actual client using `ChannelFactory<T>`, and call the operation on the service:

```
namespace Console
{
    using Service;

    class Console
    {
        public static void Main(string[] args)
        {
            var client = new
System.ServiceModel.ChannelFactory<IExample>("Service.Example").CreateChannel();
            var s = client.Echo("Hello World");
            Console.Write(s);
            Console.Read();
        }
    }
}
```

Adding a metadata endpoint to your service

SOAP services can publish metadata that describes the methods that may be invoked by clients. Clients can use tools such as *Visual Studio* to automatically generate code (known as *client proxies*). The proxies hide the complexity of invoking a service. To invoke a service, one merely invokes a method on a client proxy.

First you must add a metadata endpoint to your service. Assuming your service looks the same as the one defined in the "First service and host" example, you can make the following changes to the configuration file.

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="serviceBehaviour">
        <serviceMetadata httpGetEnabled="true" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service name="Service.Example" behaviorConfiguration="serviceBehaviour">
      <endpoint address="mex" binding="mexHttpBinding" name="mexExampleService"
contract="IMetadataExchange" />
      <endpoint name="netTcpExample" contract="Service.IExample" binding="netTcpBinding" />
      <host>
        <baseAddresses>
          <add baseAddress="net.tcp://localhost:9000/Example" />
```

```
          <add baseAddress="http://localhost:8000/Example" />
        </baseAddresses>
      </host>
    </service>
  </services>
</system.serviceModel>
```

The mexHttpbinding exposes the interface over http, so now you can go with a web browser to

http://localhost:8000/Example http://localhost:8000/Example?wsdl

and it will display the service and its metadata.

Create a ServiceHost programmatically

Creating a ServiceHost programmatically (**without** config file) in its most basic form:

```
namespace ConsoleHost
{
  class ConsoleHost
  {
    ServiceHost mHost;

    public Console()
    {
      mHost = new ServiceHost(typeof(Example), new Uri("net.tcp://localhost:9000/Example"));

      NetTcpBinding tcp = new NetTcpBinding();

      mHost.AddServiceEndpoint(typeof(IExample),tcp,"net.tcp://localhost:9000/Example");

    }

    public void Open()
    {
        mHost.Open();
    }

    public void Close()
    {
        mHost.Close();
    }

    public static void Main(string[] args)
    {
        ConsoleHost host = new ConsoleHost();

        host.Open();

        Console.ReadLine();

        host.Close();
    }
  }
}
```

1. Create a ServiceHost instance passing the concrete class type and zero or more

baseaddress Uri's.
2. Construct the desired binding, NetTcpBinding in this case.
3. call AddServiceEndpoint passing the **A**ddress, **B**inding and **C**ontract. (ABC mnemonic for WCF endpoints).
4. Open the host.
5. Keep host open until user press key on console.
6. Close the host.

Programmatically adding a metadata endpoint to a service

When you also want to expose metadata without a config file you can build on the example programmatically creating a ServiceHost:

```
public ConsoleHost()
{
    mHost = new ServiceHost(typeof(Example), new Uri("http://localhost:8000/Example"), new
Uri("net.tcp://9000/Example"));

    NetTcpBinding tcp = new NetTcpBinding();

    mHost.AddServiceEndpoint(typeof(IExample), tcp, "net.tcp://localhost:9000/Example");


    ServiceMetadataBehavior metaBehavior =
mHost.Description.Behaviors.Find<ServiceMetadataBehavior>();

    if (metaBehavior == null)
    {
        metaBehavior = new ServiceMetadataBehavior();
        metaBehavior.MetadataExporter.PolicyVersion = PolicyVersion.Policy15;
        metaBehavior.HttpGetEnabled = true;

        mHost.Description.Behaviors.Add(metaBehavior);
        mHost.AddServiceEndpoint(ServiceMetadataBehavior.MexContractName,
MetadataExchangeBindings.CreateMexHttpBinding(), "mex");
    }

    mHost.Open();
}
```

1. Create a ServiceHost instance passing the concrete class type and zero or more baseaddress Uri's.
2. When you use mexHttpBinding you have to add http://localhost:8000/Example baseaddress
3. Construct the desired binding, NetTcpBinding in this case.
4. call AddServiceEndpoint passing the Address, Binding and Contract. (ABC).
5. Construct a ServiceMetadataBehavior
6. Set HttpGetEnabled to true
7. Add the metadata behavior to the behaviors collection.
8. call AddServiceEndpoint passing the constants for metadata exchange
9. Open the host.

Read Your first service online: https://riptutorial.com/wcf/topic/2333/your-first-service

Credits

S. No	Chapters	Contributors
1	Getting started with wcf	Community, MickyD, Sathish Prabhakaran, Uriil
2	DataContractSerializer is an Opt-In and Opt-Out serializer.	David, Yawar Murtaza
3	Handling exceptions	Kye, Laurijssen, Ricardo Pontual
4	How to use a dependency injection container with a WCF service	Scott Hannen
5	How to: Disable/Enable WCF tracing in C# application code	MikeZ
6	Serialization	Ameya Deshpande, Bardia, Gal Yedidovich
7	Tracing	esiprogrammer, Eugene S.
8	WCF - Http and Https on SOAP and REST	David
9	WCF Restful Service	Nirav Mehta
10	WCF Security	esiprogrammer
11	Your first service	Laurijssen, MickyD, Piyush Parashar, tom redfern