

# **Windows Communication Foundation Using C#**

*Student Guide*

**Revision 4.7**

# **Windows Communication Foundation Using C#**

## **Rev. 4.7**

### **Student Guide**

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Object Innovations.

Product and company names mentioned herein are the trademarks or registered trademarks of their respective owners.



™ is a trademark of Object Innovations.

**Authors:** Robert J. Oberg, Julian Templeman and Ernani Junior Cecon

Copyright ©2015 Object Innovations Enterprises, LLC All rights reserved.

Object Innovations  
877-558-7246  
[www.objectinnovations.com](http://www.objectinnovations.com)

Published in the United States of America.

## Table of Contents (Overview)

Chapter 1	WCF Essentials
Chapter 2	Addresses and Bindings
Chapter 3	Service Contracts
Chapter 4	Instance Management
Chapter 5	Data Contracts
Chapter 6	More about Service Contracts
Chapter 7	Handling Errors
Chapter 8	WCF Security
Chapter 9	WCF Routing
Appendix A	Learning Resources
Appendix B	Hosting in IIS 7.5

# Directory Structure

---

- **Install the course software by running the self-extractor *Install\_WcfCs\_47.exe*.**
- **The course software installs to the root directory *C:\OIC\WcfCs*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02** and so on.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **Demos** directory is provided for performing in-class demonstrations led by the instructor.

# Table of Contents (Detailed)

<b>Chapter 1: WCF Essentials.....</b>	<b>1</b>
What Is WCF?.....	3
WCF Services .....	4
Service Orientation .....	6
WCF and Web Services .....	7
WCF and Web API .....	8
WCF = ABC .....	9
Address, Binding, Contract.....	10
Example – Hello WCF.....	11
Hosting Services .....	12
Demo – Hello WCF .....	13
A Service Contract.....	15
Visual Studio WCF Test Host.....	16
WCF Test Client .....	17
Closing the Test Host Manually .....	19
Self-Hosting .....	20
ServiceHost Class .....	21
Host Life Cycle .....	22
WCF Clients.....	23
Channels.....	24
Demo – A Client for Hello WCF.....	25
ChannelFactory .....	26
Running the Example.....	27
Base Address.....	29
Base Address Example .....	30
Uri Class .....	31
Configuration Files .....	32
Simplified Host Code.....	33
Proxy Initialization .....	34
Metadata Exchange.....	35
Metadata Exchange Example.....	36
Behaviors .....	37
A Service in a Browser .....	38
Proxy Demo – SvcUtil.....	39
Proxy Demo – Visual Studio Proxy.....	41
Standard Endpoints .....	43
Lab 1 .....	44
WCF Architecture.....	45
ServiceHost and ChannelFactory.....	47
Service Contexts and Instances.....	48
Summary .....	49

<b>Chapter 2: Addresses and Bindings .....</b>	<b>55</b>
Addresses .....	57
Transports .....	58
Bindings .....	59
Message Exchange Patterns (MEPs) .....	60
Security .....	61
Choosing a Binding .....	62
HTTP Bindings .....	63
TCP and Named Pipe Bindings .....	65
MSMQ Bindings .....	66
WebSocket and UDP Bindings .....	67
Importance of BasicHttpBinding .....	68
Demo – ASMX Web Service Client .....	69
Working with Endpoints .....	74
Default Endpoints and Bindings .....	75
Default Endpoints Example .....	76
Helper Methods .....	77
ServiceDescription Class .....	78
Multiple Endpoints .....	79
Multiple Endpoints Configuration File .....	81
Multiple Protocol Example .....	82
Simple Host Code .....	83
Lab 2 .....	84
Summary .....	85
<b>Chapter 3: Service Contracts .....</b>	<b>93</b>
Service Contracts at Class Level .....	95
Service Contracts at Interface Level .....	96
Benefits of Interface Level Definition .....	97
A Service with Multiple Contracts .....	98
ServiceContractAttribute .....	100
ServiceContract Attribute Example .....	101
Attributes in WSDL .....	103
Viewing WSDL Files .....	104
Contract Inheritance .....	105
Operation Overloading .....	108
Enabling Operation Overloading .....	109
Operation Overloading Client .....	110
Lab 3A .....	111
Lab 3B .....	112
Summary .....	113
<b>Chapter 4: Instance Management .....</b>	<b>123</b>
Behaviors .....	125
WCF Behaviors .....	126
Configuring Behaviors .....	127

Configuring Behaviors in Code .....	128
WCF Instancing Models .....	129
Per-Call Instancing .....	130
Per-Session Instancing .....	131
Sessions and Threading .....	132
Singleton Instancing .....	133
Which Model to Use? .....	134
PerSession Example.....	135
PerCall Example .....	138
Singleton Example .....	139
Windows Forms WCF Clients .....	140
Lab 4 .....	141
Summary .....	142
<b>Chapter 5: Data Contracts.....</b>	<b>147</b>
Data Contracts.....	149
Data Contract Example .....	150
Data Contract Demonstration .....	151
XSD for Data Contract.....	154
Client Demo .....	156
Arrays.....	158
Array in XML Schema.....	159
Array in Proxy .....	160
Generic Collections.....	161
Generic Collection in XML Schema .....	162
Generic Collection in Proxy .....	163
Lab 5A .....	164
Enumerations in Data Contracts .....	165
Enumeration Data Contract Example .....	166
Employee Client Program.....	167
Saving and Restoring .....	168
Serialization in WCF .....	169
Serialization in .NET .....	170
Serialization Example .....	171
SOAP Serialization .....	172
DataContract Serialization.....	173
JSON Serialization.....	175
Using XmlSerializer .....	176
XmlSerializer Example .....	178
Restoring Data .....	180
Versioning.....	182
New and Missing Members .....	183
Versioning Demonstration.....	184
New Client of Old Service.....	186
Round Trip.....	187
Required Members.....	188

OnDeserializing Event .....	189
Lab 5B .....	190
Summary .....	191
<b>Chapter 6: More about Service Contracts .....</b>	<b>201</b>
Versioning Service Contracts .....	203
Versioning Example.....	204
Version 1 Service.....	205
Version 2 Service.....	206
New Operations .....	207
Version 3 Service.....	208
Version 1 Client / Version 3 Service .....	209
Version 2 Client / Version 2 Service .....	210
Version 3 Client / Version 3 Service .....	211
Message Exchange Patterns.....	212
Request-Reply .....	213
Oneway .....	214
Oneway Example.....	215
Duplex.....	216
Callbacks.....	217
Invoking a Callback.....	218
Callback on the Client.....	219
Asynchronous Proxies .....	220
Threading Considerations.....	221
Task-Based Asynchronous Pattern .....	223
Task-Based Client.....	224
WebSockets.....	225
Lab 6 .....	226
Summary .....	227
<b>Chapter 7: Handling Errors .....</b>	<b>235</b>
Errors in Distributed Systems .....	237
Errors in .NET and WCF .....	238
Demo of WCF Error Behavior.....	239
Service Library Code .....	241
Client Code .....	242
Client Exception Handling.....	243
Exception Handling Demo.....	244
Fault Exceptions .....	246
Faults.....	247
Exception Details in Faults .....	248
Exception Details.....	249
Exception Settings Window.....	251
Exception Details Demo .....	252
Fault Contracts .....	254
Fault Contract Example .....	255



Fault Contract Example – Client .....	256
Custom Faults .....	257
Faulted Channels.....	258
Lab 7 .....	259
Summary .....	260
<b>Chapter 8: WCF Security .....</b>	<b>269</b>
Services and Security.....	271
Security Aspects of Services.....	272
Transfer Security.....	273
Transport Security.....	274
Scenarios for Transport Security .....	275
Configuring Transport Security .....	276
Transport Security Example .....	277
Host's Security Configuration .....	278
Client's Security Configuration.....	279
Message Security .....	280
Scenarios for Message Security.....	281
Configuring Message Security.....	282
Other Security Modes .....	283
Certificates .....	284
Certificate Demo .....	285
Managing Certificates.....	286
Exception Details .....	290
Client Certificate Configuration .....	292
Sending Credentials .....	295
Username Credentials .....	296
Username Example .....	297
Lab 8 .....	300
Summary .....	301
<b>Chapter 9: WCF Routing.....</b>	<b>313</b>
WCF Routing Service .....	315
Protocol Bridging Example .....	316
Service Contract and Implementation.....	317
Service Configuration .....	318
Hosting the Service .....	319
Client Application.....	320
Configuring the Router .....	321
Running the Example.....	323
Router Configuration File .....	324
Routing Contracts .....	327
Message Filters .....	328
EndpointName Message Filter.....	329
EndPointName Router Configuration.....	330
Incoming Endpoints and the Client .....	332

Error Handling .....	333
Backup List Example .....	334
Running Backup List Example .....	335
WCF Routing Scenarios .....	336
Lab 9 .....	339
Summary .....	340
<b>Appendix A: Learning Resources.....</b>	<b>351</b>
<b>Appendix B: Hosting in IIS 7.5.....</b>	<b>355</b>
Internet Information Services .....	356
Installing IIS 7.5 .....	357
WCF with IIS 7.5 .....	358
.NET Framework Version.....	359
Registering ASP.NET .....	362
Demo – Hello WCF .....	363
A Service Contract.....	365
A Website for the Service .....	366
WCF Service Template.....	368
Service Configuration .....	369
Referencing the Class Library .....	370
Examining the Service in the Browser .....	371
WCF Clients.....	372
Creating WCF Clients.....	373
Demo – A Client for Hello WCF.....	374
Service as an IIS Application.....	378
Converting to an Application.....	379
Configuring as an Application.....	380
Moving a WCF Solution.....	381

# Chapter 1

## WCF Essentials

# WCF Essentials

## Objectives

---

*After completing this unit you will be able to:*

- **Explain how WCF unites and extends existing distribution technologies.**
- **Explain the concepts of address, binding, contract and endpoint.**
- **Describe how WCF services can be hosted**
- **Create a simple self-hosted WCF service configured via code.**
- **Implement a client of a WCF service using a Channel Factory.**
- **Use a configuration file to configure a service.**
- **Configure a service to export metadata.**
- **Use metadata to automatically generate a proxy for a service.**
- **Understand the WCF architecture and runtime.**

# What Is WCF?

---

- **Windows Communication Foundation (WCF) is a new service-oriented programming framework for creating distributed applications.**
  - It was previously known as ‘Indigo’ and is part of .NET 3.0 and higher.
- **WCF is designed to provide one mechanism for building connected applications:**
  - Within app domains
  - Across app domains
  - Across machines
- **WCF builds upon and extends existing ways of building distributed applications:**
  - ASMX Web services, .NET Remoting, COM, MSMQ.
- **All these do the same basic job (connecting elements in distributed applications) but they are very different at the programming level, with complex APIs and interactions.**
  - WCF provides one model for programming distributed applications. Developers only need to learn one API.
- **WCF leverages existing mechanisms.**
  - It uses TCP, HTTP and MSMQ for transport.

# WCF Services

---

- **When using WCF, you create and consume services.**
  - A service comprises a set of related operations, which the programmer sees as method calls.
- **Services are described by metadata, which clients can use to determine what operations are available, and how the service can be contacted.**
  - Metadata for WCF services is similar to the WSDL used by web services.
- **Clients and services exchange messages.**
  - A client (which can be another service) communicates with a WCF service by sending and receiving messages. WCF was designed to use SOAP as its messaging mechanism, and SOAP messages can be sent using a number of transports.
  - Using SOAP does not imply that WCF communication is inefficient; efficient binary encodings are employed whenever possible.
  - More recently, REST services over HTTP have been retrofitted to WCF through the REST starter kit.
- **WCF supports several transports out of the box.**
  - TCP, HTTP, HTTPS and MSMQ.
  - Custom transports can be added.

# WCF Services

---

- **WCF supports the WS-\* family of Web service protocols.**
  - The WS-\* family of protocols have been developed by various bodies (including OASIS and W3C) to provide features such as security, transactions and reliable messaging to web services.
- **WCF is very good at interop.**
  - Support for a wide range of transports, encodings and the WS-\* protocols means that WCF services can interoperate with a wide range of platforms and technologies, including J2EE and web services using WS-\* protocols.
- **WCF provides a foundation for service orientation.**
  - WCF helps developers write distributed applications in which loosely coupled services are called by clients and one another.

# Service Orientation

---

- **Service orientation is characterized by four concepts.**
- **Boundaries are explicit**
  - The boundary between client and service is explicit and highly visible, because calls are made via SOAP messages. There is no pretending – as there is in DCOM and Java's RMI – that you are simply playing with a remote object, and can thus ignore the cost of remoting.
- **Services are autonomous**
  - Services are independent entities that each have their own life cycle; they may have been developed completely independently of one another. There is no run-time making sure that services work well together, and so services must be prepared to handle failure situations of all sorts.
- **Share schemas and contracts, not classes**
  - Services are not limited to implementation in OO languages, and so service details cannot be provided in terms of classes. Services should share schemas and contracts, and these are typically described in XML.
- **Use policy-based service compatibility**
  - Services should publish their requirements (i.e. requiring message signing or HTTPS connections) in a machine readable form. This can be used at runtime to ensure compatibility between service and client.



# WCF and Web Services

---

- **Even though it uses SOAP messaging, WCF is more than simply another way of writing Web services.**
  - WCF can be used to write traditional Web services, as well as more sophisticated services that can use the WS-\* protocols. But the design of WCF means that it provides a far more general solution to distribution than Web services.
- **WCF can use several transports.**
  - Web services tend to use HTTP or HTTPS, while WCF is configured to use TCP, HTTP, HTTPS and MSMQ. It is also possible to add new transports, should the need arise.
- **WCF can work throughout the enterprise.**
  - WCF services can be hosted in-process, by a Windows Service or IIS. Message exchange will be optimized to use the most efficient method for exchanging data for a particular scenario.
- **WCF is good at interop.**
  - WCF services can interoperate with a number of different platforms and technologies.
- **WCF is highly customizable.**
  - It is possible to customize almost every part of WCF, adding or modifying transports, encodings and bindings, and plugging in new ‘behaviors’ that affect the way WCF services work.

# WCF and Web API

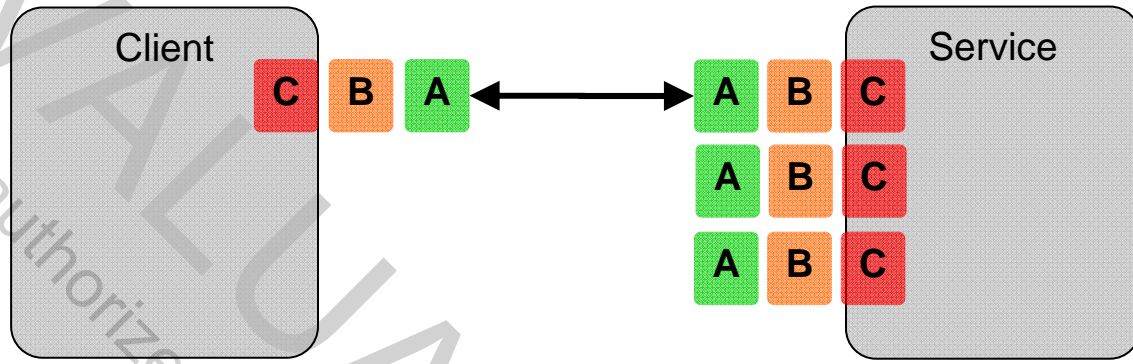
---

- **A recent approach to creating distributed applications is ASP.NET Web API<sup>1</sup>.**
  - ASP.NET Web API is a framework for building and consuming HTTP services.
  - It is built into ASP.NET and can be used by both Web Forms and MVC applications, as well as used standalone.
  - Relying on standard HTTP, Web API facilitates creating services that can reach a wide variety of devices.
  - Visual Studio 2013 provides templates that facilitate creating Web API services.
  - Web API does not dictate a particular architectural style, but it is a great platform for implementing RESTful Web services.
- **Web API was designed from the ground up to work with HTTP and thus has almost universal reach to many devices.**
- **WCF, although it can interoperate with many platforms, does not have this universal reach.**
- **WCF has extensive support of WS-\* and can run over many protocols besides HTTP.**

---

<sup>1</sup> Web API is covered in the Object Innovations course 4147, ASP.NET Web API Essentials Using C#.

# WCF = ABC



**A B C** = an endpoint

**A** = address → Where?

**B** = binding → How?

**C** = contract → What?

# Address, Binding, Contract

---

- **An *address* defines where a service can be found.**
  - It will often be an HTTP address, although other addressing schemes are supported.
- **A *binding* defines how a service can be contacted**
  - Via HTTP, TCP, MSMQ or some custom mechanism.
- **A *contract* defines what a service can do.**
  - In terms of method calls, their arguments and return types.
- **A combination of an address, a binding and a contract is called an *endpoint*.**
  - A service can expose more than one endpoint, and endpoint data can be made available to clients in the form of metadata.

## Example – Hello WCF

---

- **A contract defines a set of operations that a service supports.**
  - Define a contract as an interface decorated with the **ServiceContract** attribute.
  - Decorate operations with the **OperationContract** attribute.

```
[ServiceContract]
public interface IHello
{
    [OperationContract]
    string SayHello(string name);
}
```

- **A service class implements the interface.**
  - And so it has to implement all the operations defined in the contract.

```
public class Hello : IHello
{
    public string SayHello(string name)
    {
        return "Hello, " + name;
    }
}
```

- **Note that this code says nothing about how the client communicates with the service.**
  - It is only the 'C' of the service 'ABC'.

# Hosting Services

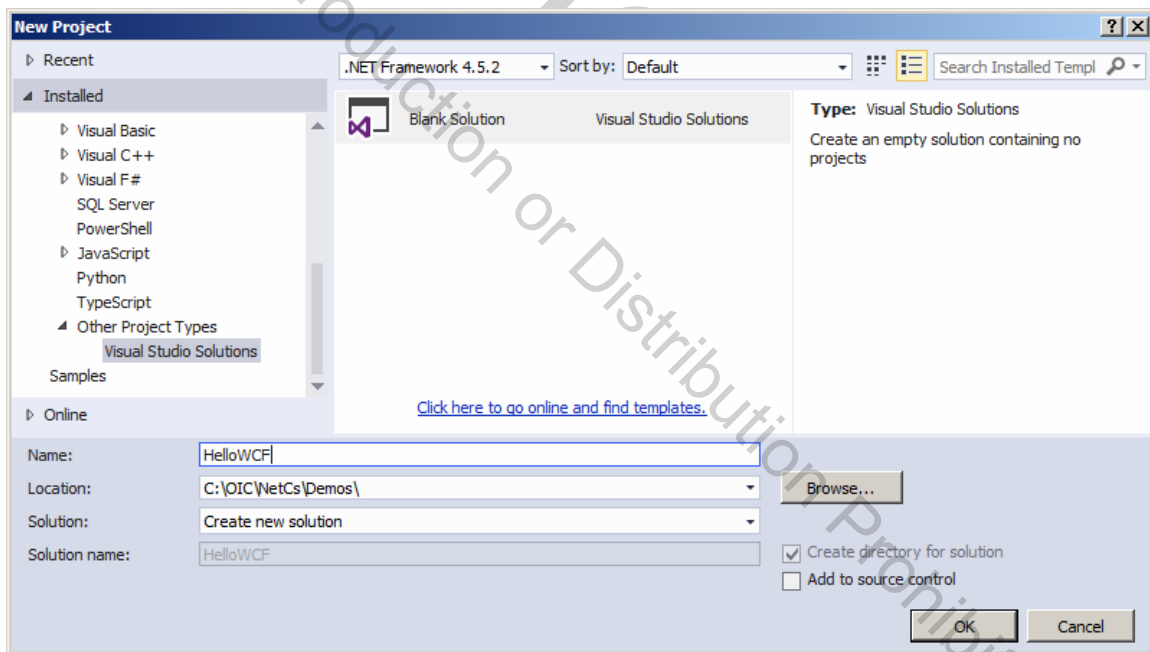
---

- **There are four ways to host a WCF service.**
- **Self-hosting in an EXE.**
  - Use any type of EXE: Console application, Windows application (Windows Forms or WPF), or Windows Service.
  - Need to manage service lifecycle yourself.
- **Hosting in IIS.**
  - IIS will manage the service lifecycle for you, starting the service when the first request comes in.
  - You can only use HTTP and port 80.
  - Configure the service using a **.svc** file.
- **Hosting in Windows Process Activation Service (WAS).**
  - WAS is a feature that is part of Vista, Windows 7 and above, and Windows Server 2008 and above.
  - Similar advantages to hosting in IIS, but you can use other transports and ports as well.
  - WAS also uses **.svc** files.
- **Hosting in Windows Server AppFabric.**
  - This new hosting engine, available on Windows 7 & above & Windows Server 2008 R2 & above, is optimized for hosting WCF and WF (Windows Workflow Foundation) services.

## Demo – Hello WCF

- We'll create a “Hello WCF” service that is self-hosted.
- Our solution will have three projects:
  - A class library implementing the service
  - A console application that hosts the service.
  - A console application that invokes the service

1. Run Visual Studio 2015 as Administrator<sup>2</sup>. Create a new blank solution **HelloWCF** in the **Demos** folder.

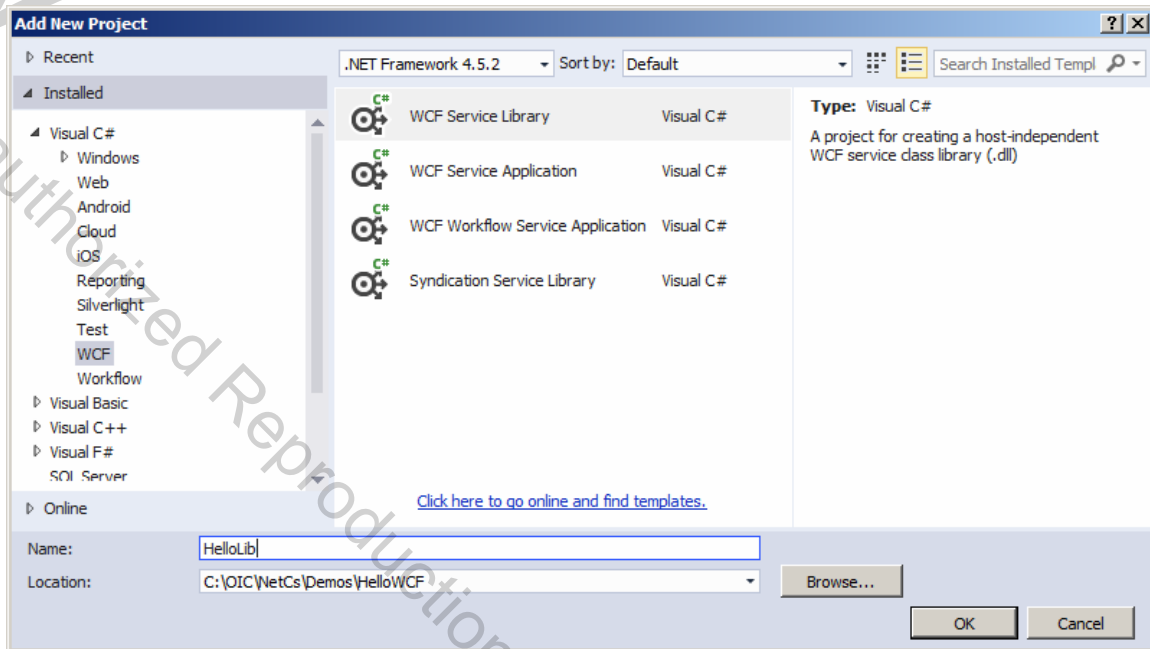


2. In Solution Explorer, right-click over the solution and choose **Add | New Project**.

<sup>2</sup> When working with WCF on Windows Vista and above it is important to always run Visual Studio as Administrator. If you forget, you'll encounter strange errors!

## Demo – Hello WCF (Cont'd)

- From the WCF project types choose the WCF Service Library template. Enter **HelloLib** as the name of your new project.



- Change the name of the file **Service1.cs** to **HelloService.cs**, and **IService1.cs** to **IHelloService.cs**. Say Yes to renaming all references in the project to the corresponding code element.
- Examine the contents of these files, which include comments and starter code for both a Service Contract and also a Data Contract.
- Examine also the file **App.config**. The renaming was not perfect, as under the <baseAddresses> tag there is still use of **Service1**. There is actually a reason for this, as we'll see when we make use of the test programs **WcfSvcHost.exe** and **WcfTestClient.exe**.



## A Service Contract

---

7. Edit **IHelloService.cs** to include only a simple service contract **IHelloService** with one method, **SayHello()**.

```
namespace HelloLib
{
    [ServiceContract]
    public interface IHelloService
    {
        [OperationContract]
        string SayHello(string name);
    }
}
```

8. Edit **HelloService.cs** to implement the service contract.

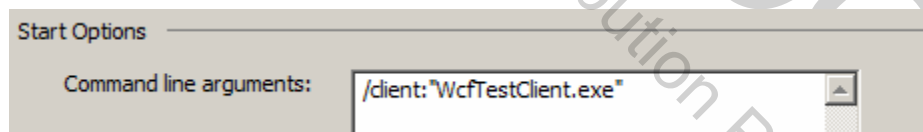
```
namespace HelloLib
{
    public class HelloService : IHelloService
    {
        public string SayHello(string name)
        {
            return "Hello: " + name;
        }
    }
}
```

9. Build the service library project.

# Visual Studio WCF Test Host

---

- **To test the service we need both a host and a client program.**
- **We will implement both, but first let's look at tools provided with Visual Studio.**
  - There is a test host **WcfSvcHost.exe** and a test client **WcfTestClient.exe**.
  - Both are located in<sup>3</sup>:  
\\Program Files\\Microsoft Visual Studio 14.0\\Common7\\IDE
    - This folder is already on the path of the Visual Studio command prompt.
- **A WCF Service Library project created using Visual Studio is set up to invoke the test host and test client automatically.**
  - Look at the Debug tab of the project's properties.



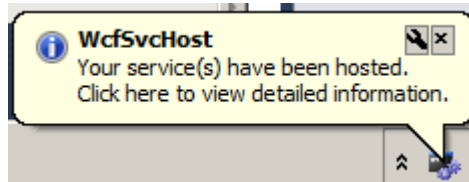
- Also, “Enable the Visual Studio hosting process” is checked.
- When you build and run the class library, the test host will run, and then the test client will start.

---

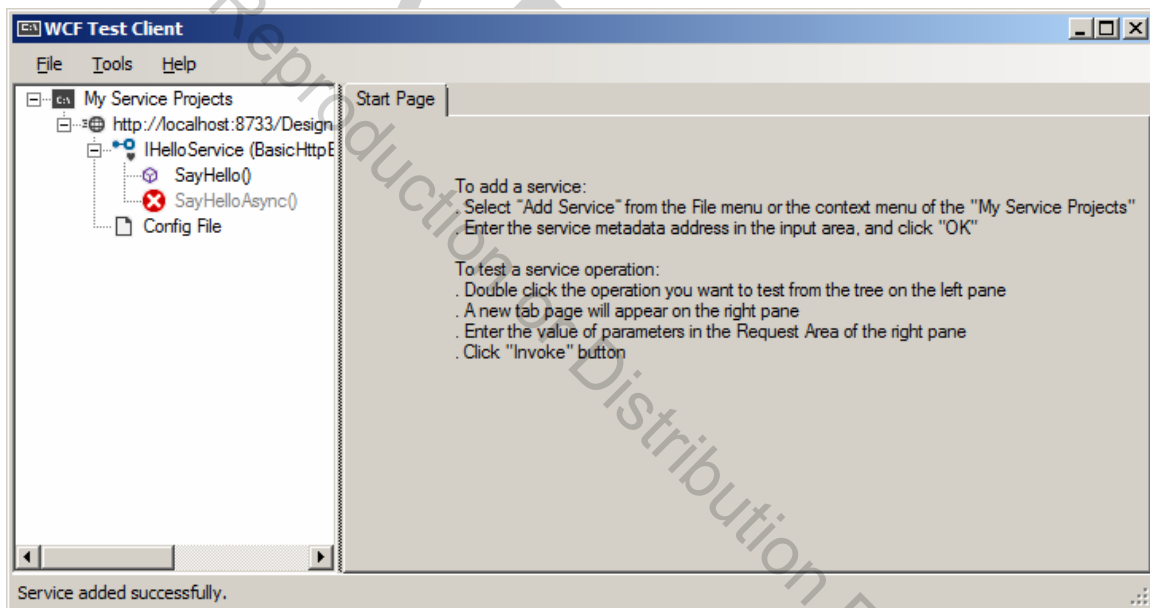
<sup>3</sup> In 64-bit Windows the top-level folder is Program Files (x86).

# WCF Test Client

1. Build the class library project. A bubble will be displayed at the bottom of your screen indicating that **WcfSvcHost** has started.



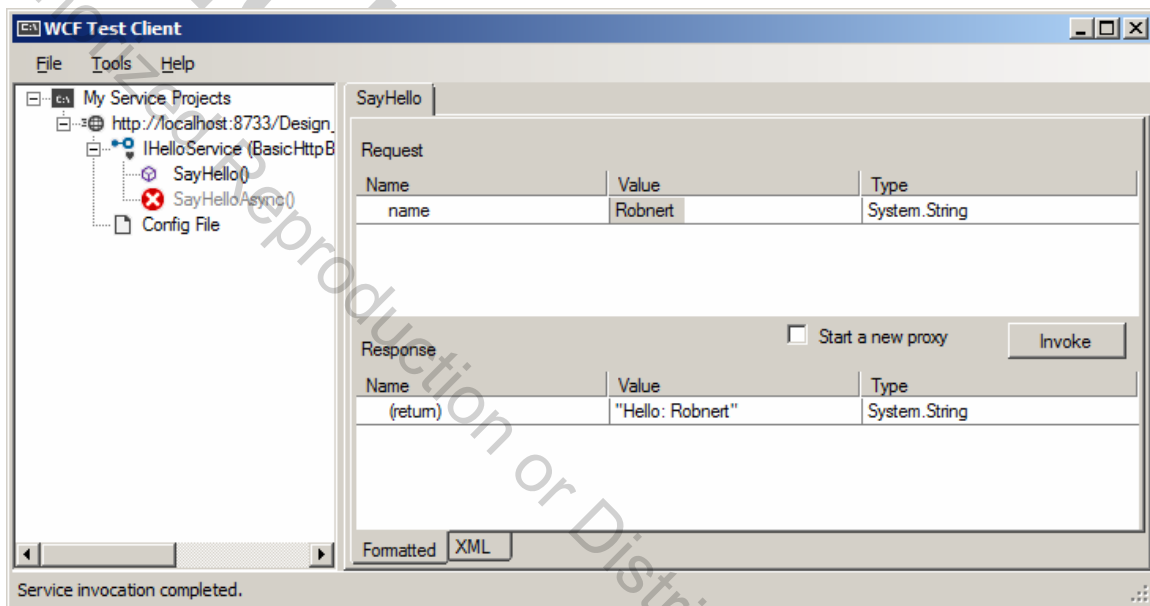
2. **WcfTestClient** will now start, and it has automatically connected to the test host for your service library. A tree view of service projects has been populated with your service project.




3. Double-click on the **SayHello()** method.

## WCF Test Client (Cont'd)

4. You will see a form in which you can enter values for the parameters to **SayHello()**.
5. In our case there is only one parameter. Enter a value for the name and click Invoke. After a few seconds you should see the response. (You may click OK to the security warning and select not to see the warning anymore.)



6. This concludes our quick test. Close the test client by File | Exit or by clicking the . Normally the test host will also close.

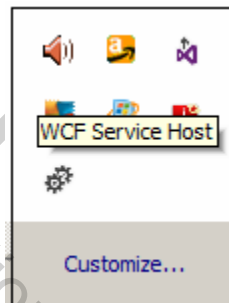
# Closing the Test Host Manually

---

- **A pitfall may be that an instance of WCF Service Host remains running when you don't think it is.**
  - To check for this, display the hidden icons from the task bar.



- You may see an icon for WCF Service Host.



- Right-click over the icon and choose Exit from the context menu.
7. The program at this point is saved in **Chap01\HelloWCF\LibraryOnly**.

# Self-Hosting

---

- **Next we will add a console application that will host our service.**
  1. Right-click over the solution and choose Add | New Project.
  2. From the Windows group choose Console Application. Assign the name **Host** and click OK.
  3. Add a Project reference to **HelloLib** and a Framework reference (under Assemblies) to **System.ServiceModel**. Import these two.

```
using System.ServiceModel;  
using HelloLib;
```

4. Provide this code in **Main()**.

```
using (ServiceHost host = new ServiceHost(  
    typeof(HelloService)))  
{  
    host.AddServiceEndpoint(  
        typeof(HelloService),  
        new BasicHttpBinding(),  
        "http://localhost:8000/HelloService");  
    host.Open();  
  
    Console.WriteLine(  
        "Press ENTER to terminate service host");  
    Console.ReadLine();  
}
```

5. Right-click on the Host project and choose Set as Startup Project from the context menu.
6. Build and run to make sure you get a clean compile and no runtime errors for the host. Press the ENTER key and exit. The project is saved in **HelloWCF\LibraryAndHost**.

# ServiceHost Class

---

- **The *ServiceHost* is used to implement a host.**
  - In self-hosting you instantiate an instance of **ServiceHost** directly.
  - IIS and WAS use **ServiceHost** on your behalf.
- **The main constructor requires a service type and zero or more base addresses.**

```
public ServiceHost(  
    Type serviceType,  
    params Uri[] baseAddresses  
)
```

- **Our example illustrated the simplest case with no base addresses specified.**

```
ServiceHost host = new ServiceHost(  
    typeof(HelloService))
```

- Endpoints must then supply an absolute URI for an address.

```
host.AddServiceEndpoint(  
    typeof(HelloService),    // contract  
    new BasicHttpBinding(),  // binding  
    "http://localhost:8000/HelloService");  
                                // address
```

- Note the “ABC” of address, binding and contract for the endpoint.

# Host Life Cycle

---

- A service host life cycle is controlled by calls to *Open()* and *Close()*.
  - **Open()** allows calls into the host, which are processed by worker threads.
  - **Close()** gracefully exits the host, refusing new calls to the host but allowing calls in progress to complete.
  - The **CloseTimeout** property (10 seconds by default) specifies the length of time the host will wait for the calls in progress to complete before shutting down anyway.
- The **C# *using* statement** facilitates managing the host's life cycle.
  - **ServiceHost** implements the **IDisposable** interface. Exiting a **using** block, either through normal program flow or via an exception, will call the **Dispose()** method, which in turn calls **Close()**.

```
using (ServiceHost host = new ServiceHost(...  
{  
    ...  
    host.Open();  
  
    Console.WriteLine(  
        "Press ENTER to terminate service host");  
    Console.ReadLine();  
}
```



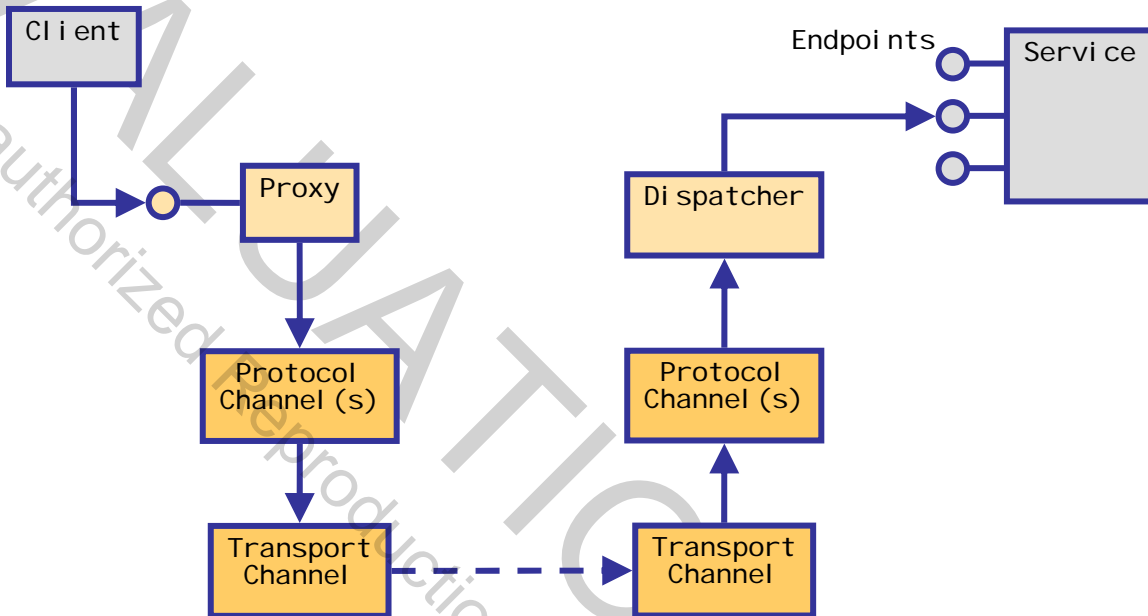
# WCF Clients

---

- **As in all common distribution technologies, clients interact with services through proxies.**
  - The proxy hides details of the communication mechanism being used from the client code.
  - The proxy implements the same interface as the service, so that the client can use exactly the same calls.
- **Proxies are created using metadata provided by the service.**
  - Note that the page shown in the previous page gives instructions for generating a proxy.
- **The ABC (address, binding, contract) information provided by the service can be used to construct a proxy that**
  - Knows where to contact the service.
  - Implements code to use the appropriate communication mechanism (eg. HTTP versus TCP).
  - Implements the operations defined in the service interface.
- **There are two approaches to creating proxies:**
  - “On the fly” using the **ChannelFactory** class.
  - Auto-generate in advance by using a tool such as **SvcUtil.exe** or by adding a Service Reference in Visual Studio.

# Channels

- Channels are used for communication between clients and services in WCF.



- **Client code creates a proxy.**
  - The client often uses a **ChannelFactory** object to create the proxy. The proxy sends messages through the channel stack to the Dispatcher, which makes a call to an endpoint on the service.
- **Service code often uses *ServiceHost* objects to manage WCF services.**

## Demo – A Client for Hello WCF

---

- **We'll create a console client.**
  1. Add a third project to the solution, another Console Application. Specify **Client** as the name of your new project.
  2. Add a reference to the **System.ServiceModel** assembly and import the corresponding namespace<sup>4</sup>.
  3. Provide the **IService** service contract definition.

```
...  
using System.ServiceModel;  
  
namespace Client  
{  
    [ServiceContract]  
    public interface IService  
    {  
        [OperationContract]  
        string SayHello(string name);  
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            ...  
        }  
    }  
}
```

---

<sup>4</sup> A shortcut for importing a namespace is to place the mouse cursor over an unrecognized symbol and press Ctrl+Dot. Select **using ...** from the context menu. You may also right-click over the unrecognized symbol and select Quick Action from the context menu.

# ChannelFactory

---

4. Provide the following code in **Main()**. This initialize a proxy object as an instance of the **ChannelFactory** class. It is based on an endpoint that is compatible with the endpoint in the service.

```
EndpointAddress ep = new EndpointAddress(
    "http://localhost:8000/HelloService");
IHelloService proxy =
    ChannelFactory<IHelloService>.CreateChannel(
        new BasicHttpBinding(), ep);
```

5. Provide the following code to use the proxy object to call the **SayHello()** method of the service and display the result. Exit the client program when the user presses the ENTER key.

```
string result = proxy.SayHello("ChannelFactory");
Console.WriteLine(result);

Console.WriteLine(
    "Press ENTER to terminate client");
Console.ReadLine();
```

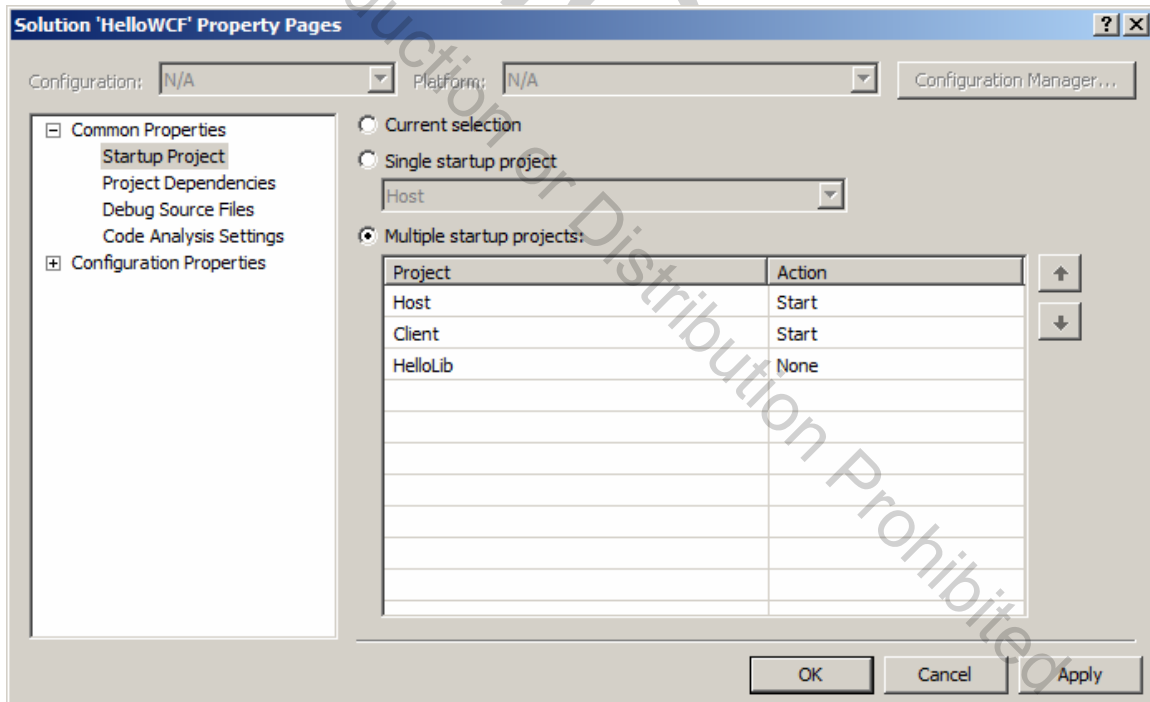
6. Delete the **App.config** files from each project, because they are not used now.<sup>5</sup>
7. Build the solution, now consisting of three projects.

---

<sup>5</sup> The configuration file was used in the service library project to enable generation of metadata, which was used by the test host and test client. We will discuss configuration files and metadata shortly.

## Running the Example

- **One way to run the example is manually to first start the host and then start the client.**
  - Right-click over the project and select Debug | Start new instance from the context menu.
- **Alternatively, you can configure the solution to automatically run the two projects, starting the host first.**
  - Right-click over the solution and select Set StartUp Projects from the context menu. Choose an action of Start for both Host and Client. Move the Host up to top position.



## Running the Example (Cont'd)

---

### Output from Host:

Press ENTER to terminate service host

### Output from Client:

Hello: ChannelFactory

Press ENTER to terminate client

- Terminate first the client and then the host.
- The solution is saved in **HelloWCF\ChannelFactory**.

## Base Address

---

- **Our next example illustrates specifying a base address for the *http://* addressing scheme.**

```
Uri httpBaseAddress =  
    new Uri("http://localhost:8000/");  
using (ServiceHost host = new ServiceHost(  
    typeof(HelloService),    // service type  
    httpBaseAddress))        // base address
```

- **In adding an endpoint we now specify a relative address.**

```
host.AddServiceEndpoint(  
    typeof(HelloService),    // contract  
    new BasicHttpBinding(),  // binding  
    "HelloService");         // relative address  
  
// complete address is  
// http://localhost:8000/HelloService
```

- See Chap01\HelloWCF\BaseAddress.

## Base Address Example

---

- **We've just seen an alternative way of specifying the address for an endpoint.**
  - The service host has one or more base addresses, and endpoints are specified using relative addresses.

```
host.AddServiceEndpoint(  
    typeof(HelloService), // contract  
    new BasicHttpBinding(), // binding  
    "HelloService"); // relative address  
// complete address is  
// http://localhost:8000/HelloService
```

- **We instrument the host code to display all the base addresses before calling *Open()*.**

```
ShowBaseAddresses(host.BaseAddresses);  
...  
static void ShowBaseAddresses(  
    ReadOnlyCollection<Uri> addresses)  
{  
    Console.WriteLine("Base Addresses:");  
    foreach (Uri addr in addresses)  
        Console.WriteLine("    {0}",  
            addr.OriginalString);  
}
```

- **The client program is identical, because we've not changed the contract, address and binding.**
- **Here is the output from running the host:**

```
Base Addresses:  
    http://localhost:8000/  
Press ENTER to terminate service host
```



# Uri Class

---

- **The Uri class encapsulates a uniform resource identifier (URI) and provides easy access to the parts of the URI.**

- **Common properties include:**

AbsolutePath	Absolute path of the URI
IsAbsoluteUri	Is the URI instance absolute?
LocalPath	Local operating system representation of a file name
OriginalString	The original URI string that was passed to the Uri constructor
PortNumber	Port number of the URI
Scheme	Scheme name of the URI (e.g. file, ftp, http, https, net.pipe, net.tcp, etc.)

# Configuration Files

---

- **In WCF there are always two options for supplying configuration information:**
  - In code, like we've done so far.
  - In configuration files such as **App.config** and **Web.config**.
- **As an example, the file *App.config* in the *Host* project provides the same configuration information as the *WCFHello\BaseAddress* code example.**
  - See **WCFHello\Config** for the complete example.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="HelloLib.HelloService">
        <endpoint
          address="HelloService"
          binding="basicHttpBinding"
          contract="HelloLib.IHelloService" />
        <host>
          <baseAddresses>
            <add baseAddress=
              "http://localhost:8000/" />
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

- Again we specify the address by the combination of a base address and a relative address.

## Simplified Host Code

---

- **When we have configuration specified in a configuration file, the host code becomes simpler.**

```
static void Main(string[] args)
{
    using (ServiceHost host = new ServiceHost(
        typeof(HelloLib.HelloService)))
    {
        host.Open();
        ShowBaseAddresses(host.BaseAddresses);
        Console.WriteLine(
            "Press ENTER to terminate service host");
        Console.ReadLine();
        host.Close();
    }
}
...

```

- **The last three examples all had identical client code<sup>6</sup>; we did not change the contract, binding and address, only how it was described.**
  - HelloWCF\ChannelFactory
  - HelloWCF\BaseAddress
  - HelloWCF\Config

---

<sup>6</sup> We made one small change in the client: the name used in the greeting message identifies the example.

# Proxy Initialization

---

- **Up until now we've been creating a proxy directly by using a channel factory.**
- **This approach makes several assumptions:**
  - You know the endpoint address.
  - You have a copy of the server contract definition.
  - You know the required protocols or binding configurations.
- **If you are developing both the service and the client, these are reasonable assumptions.**
- **But in the world of distributed computing when you need to talk to a service you don't own, these are not reasonable assumptions.**
- **When you don't own both the client and the service sides, a more effective approach is to generate the proxy automatically from metadata exported by the service.**
  - The metadata enables you to “know” the things you need to.
  - The process of constructing a proxy is automated through the use of tools.

# Metadata Exchange

---

- **Metadata can be exported by a service through a special endpoint, known as a *metadata exchange endpoint*.**
- **This endpoint enables the generation of a proxy and a configuration file in the client project.**
- **To support this metadata exchange you need to do two things:**
  - Add the metadata exchange endpoint to the host configuration.
  - Enable the metadata exchange behavior.
- **A metadata exchange (mex) endpoint, like other endpoints, requires an address, a binding, and a contract.**
  - For the address you need the base address for the selected binding protocol (we're using http exclusively in this chapter).
  - The contract is the predefined **IMetadataExchange**.

# Metadata Exchange Example

---

- We illustrate metadata exchange with a new variation of our *HelloWCF* example.
  - See **HelloWCF\SvcUtil**. Here is the host's configuration file, with changes indicated in bold.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service
behaviorConfiguration="serviceBehavior"
name="HelloLib.HelloService">
        <endpoint
          address="HelloService"
          binding="basicHttpBinding"
          contract="HelloLib.IHelloService" />
<endpoint binding="mexHttpBinding"
contract="IMetadataExchange"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="serviceBehavior">
          <serviceMetadata httpGetEnabled="True" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

# Behaviors

---

- **A *behavior* affects the service model locally, on either the service side or client side.**
  - A behavior is not part of the metadata and does not affect the contract. They are not shared between service and client.
  - Rather they have a local effect on how the service model processes messages.
- **Service behaviors exist for metadata, debugging, security, serialization and throttling.**
- **Client behaviors exist for debugging, security, serialization, timeouts, and routing.**
- **A new `<behaviors>` element is provided in the configuration file.**
  - A behavior has a **name**.
  - This name is used by the **behaviorConfiguration** attribute of the `<service>` element to tie a service to a behavior<sup>7</sup>.
  - Client behaviors work in a similar manner in the configuration file.
- **Behaviors can also be specified in code.**

---

<sup>7</sup> This use of the behaviorConfiguration attribute is optional in WCF 4.0.

# A Service in a Browser

- A nice feature of WCF is that it enables viewing useful information about the service in a browser.
  - Point the browser to the base address.
  - For our example the base address is **http://localhost:8000/**.



- This useful page tells you how to create and use a proxy in your client program. (If no metadata exchange, you would see nice instructions for implementing it.)



## Proxy Demo – SvcUtil

---

- **The *AutoProxy* folder in the *Demos* directory contains starter code.**
    - It is a copy of **HelloWCF\Config** with **App.config** modified to support metadata exchange, as we have seen.
1. Build the starter solution.
  2. Start the host.
  3. Point your browser to **http://localhost:8000/** and verify that the information about **HelloService** shown on the previous page is displayed. Note the instructions.
  4. Bring up a Visual Studio command prompt and navigate to the **Demos\AutoProxy** folder.
  5. Enter this command (note that it is not case sensitive):  

```
svcutil.exe http://localhost:8000/?wsdl
```
  6. There should be created files **HelloService.cs** and **output.config**.
  7. Copy these files down to the **Client** folder. Rename **output.config** to **App.config**. (If **App.config** already existed, you should edit the configuration information from **output.config** into it.) Add the files to the Client project.
  8. Terminate the host and rebuild the solution.
  9. In **Program.cs** in the Client project remove the **using** statement importing the namespace **System.ServiceModel**. (It is now only needed in the proxy that was generated. You still need a reference to the assembly, though.)

## Proxy Demo – SvcUtil (Cont'd)

---

10. Remove the interface definition.
11. Remove the first two statements in **Main()** setting up an endpoint and initializing a proxy from a channel factory.
12. Examine the file **HelloService.cs** and note that the proxy class is **HelloServiceClient**.
13. Provide the following code to initialize the proxy and to close it when done. Again we'll follow the pattern of the C# **using** statement.

```
static void Main(string[] args)
{
    using (HelloServiceClient proxy =
        new HelloServiceClient())
    {
        string result = proxy.SayHello("SvcUtil");
        Console.WriteLine(result);

        Console.WriteLine(
            "Press ENTER to terminate client");
        Console.ReadLine();
    }
}
```

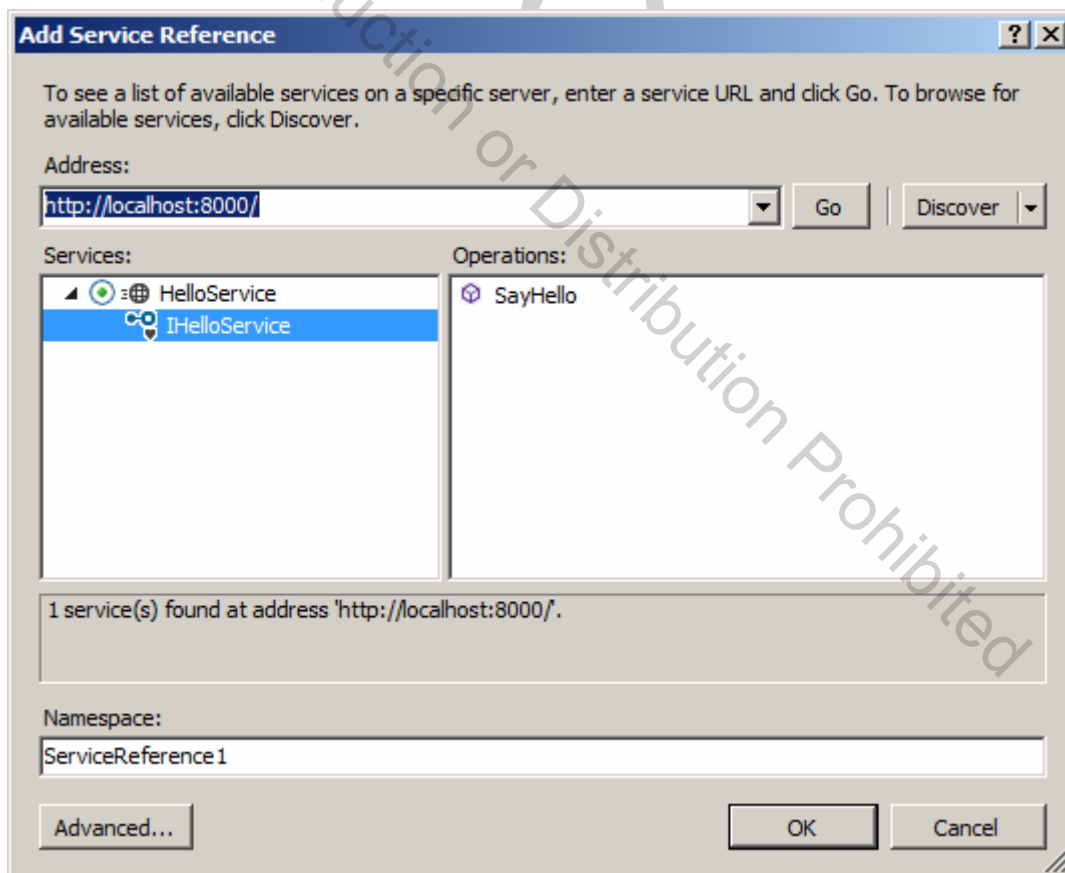
14. Build the solution. Start the host and then the client. You should see this output from the client:

```
Hello: SvcUtil
Press ENTER to terminate client
```

15. Terminate the client and then the host. The completed demo is saved in **HelloWCF\SvcUtil**. The invocation of **svcutil.exe** is done in a small batch file **MakeProxy.bat**


## Proxy Demo – Visual Studio Proxy

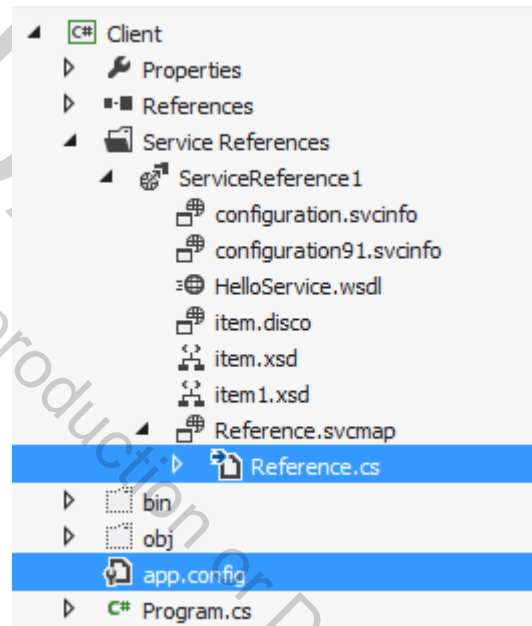
- Continue in the AutoProxy folder from the previous demo.
  1. In the Client project delete the files **HelloService.cs** and **App.config**.
  2. Start the host *outside of Visual Studio*. (Right-click over the file **Host.exe** in the **bin\Debug** folder and choose Run as Administrator from the context menu.)
  3. In Visual Studio right-click over References in Client project and choose Add Service Reference. For the address enter **http://localhost:8000/** and click Go. Expand the tree so that you can see the contract. Leave the namespace as it is. Click OK.



## Proxy Demo – VS Proxy (Cont'd)

---

4. Terminate the host.
5. In Solution Explorer, observe the new files in the Client project. You can see all files by clicking the Show All Files button . The important new files are **Reference.cs** and **app.config**.



6. Examine **Reference.cs**. The proxy class is **HelloServiceClient** and the namespace is **Client.ServiceReference1**. (We could have changed the namespace when we generated the service reference.)
7. All we need to do to use the preceding client program is to import the namespace.

```
using Client.ServiceReference1;
```

8. Build the solution. Test by starting the host and then the client. The completed demo is saved in **HelloWCF\VSProxy**.

# Standard Endpoints

---

- **An endpoint for metadata exchange almost always has exactly the same elements.**
  - To obviate the need for repetitive configuration, WCF 4.5 provides several **standard endpoints**.
  - See the simplification in **HelloWCF\StandardEndpoints**.

```
<configuration>
  <system.serviceModel>
    <standardEndpoints>
      <mexEndpoint>
        <standardEndpoint />
      </mexEndpoint>
    </standardEndpoints>
    <services>
      <service name="HelloLib.HelloService">
        <endpoint address="HelloService"
          binding="basicHttpBinding"
          contract="HelloLib.IHelloService"

          />
        ...
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="True" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

- Other standard endpoints are available for discovery, workflow and web.

# Lab 1

---

## Creating a Simple Service and Client

In this lab, you will use Visual Studio 2015 to create a simple WCF service that is self-hosted. The metadata exchange endpoint is implemented with the help of starter code provided by Visual Studio. You will create a simple Console client program. Create the proxy by adding a Service Reference using Visual Studio. You will also experiment with changing the binding from **basicHttpBinding** to **wsHttpBinding**.

Detailed instructions are contained in the Lab 1 write-up at the end of the chapter.

Suggested time: 50 minutes

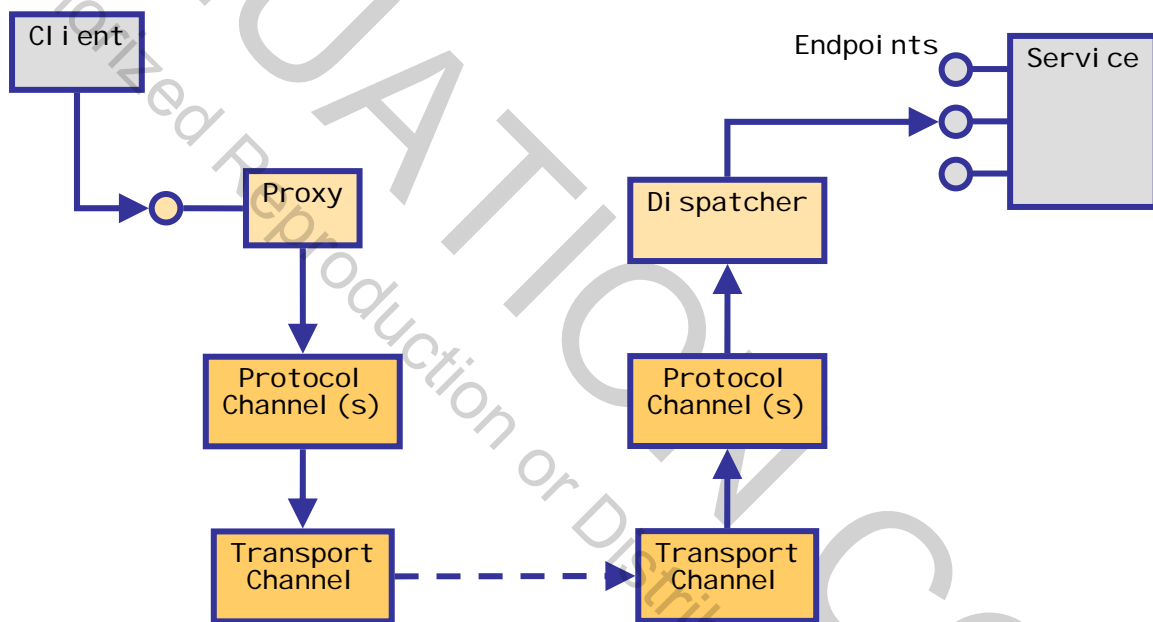
# WCF Architecture

---

- **Services and clients exchange SOAP messages.**
  - But they are not limited to text format; they can be sent in binary if that is more efficient.
- **Channels abstract the communication process.**
  - Channels can be composed, so that the output from one channel acts as the input to another.
  - There are two types of channel.
- **Transport channels implement transport mechanisms**
  - Such as HTTP and TCP/IP.
- **Protocol channels implement elements of the SOAP-based messaging protocol**
  - For example, the security channel implements SOAP security.
  - The channel stack can be specified in configuration files or in code.
- **Behaviors extend or modify service and client operation**
  - For example, whether metadata is published, or authentication is required.
  - Behaviors can be specified in configuration or code.

# WCF Architecture

- The following diagram illustrates the overall architecture of WCF.
  - We looked at this diagram earlier in the chapter when we created a simple proxy using the **ChannelFactory** class.





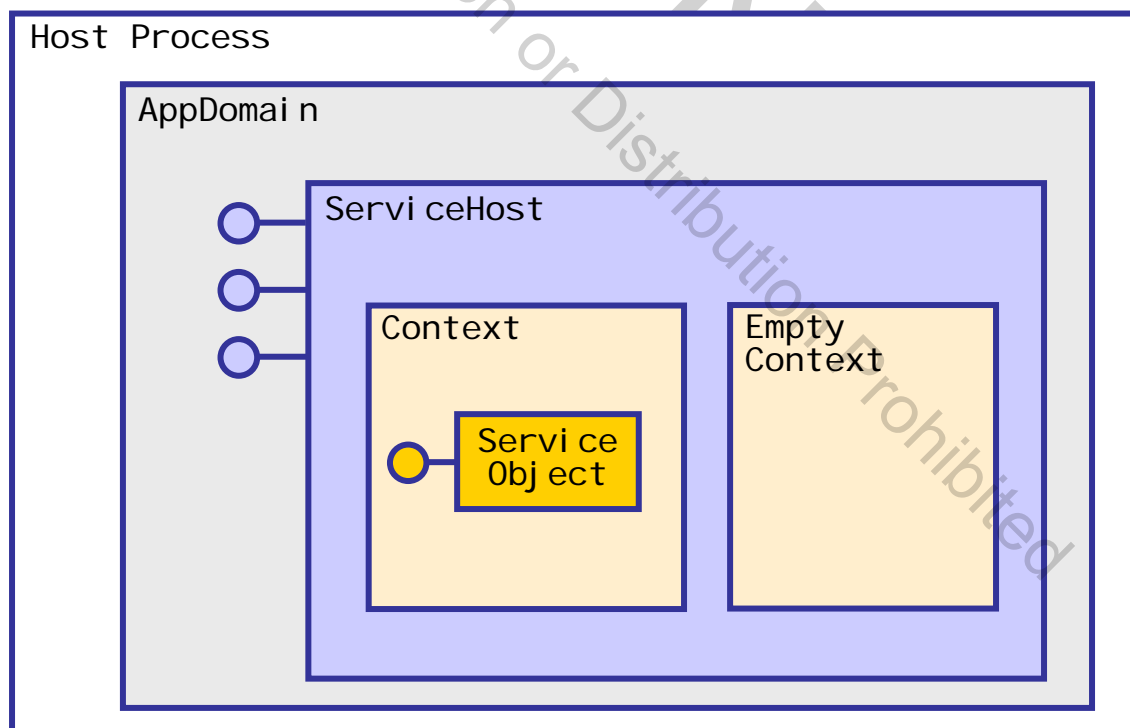
# ServiceHost and ChannelFactory

---

- **ServiceHost is used on the server side.**
  - A ServiceHost is used to host services in code (ie. Self-hosted services). The ServiceHost creates a ServiceDescription object, which consists of a type (implementing the service), a collection of behaviors (which control how the service behaves) and descriptions of one or more endpoints.
  - The WCF runtime uses the ServiceDescription to build the channel stack and configure the endpoints.
- **ChannelFactory is used on the client side.**
  - A ChannelFactory creates a ClientDescription. This consists of a collection of behaviors, and one endpoint. There is no need for a type, because clients don't implement service contracts.
  - The WCF runtime uses the ClientDescription to build the channel stack and proxy.

# Service Contexts and Instances

- **Each .NET host process can contain one or more app domains, and each app domain may contain zero or more ServiceHosts.**
  - Within a ServiceHost, service instances live in contexts; a WCF context is similar to a COM+ (Enterprise Services) context or a .NET context-bound object, in that it provides an environment in which service instances live.
  - Properties and methods on the context allow the developer to control the behavior of the service instance, especially with respect to concurrency and object lifetime. A context can host at most one instance of a service object, so it is possible to have empty contexts.



# Summary

---

- **WCF unifies a number of existing technologies for creating distributed applications.**
- **WCF services are defined by addresses, bindings and contracts.**
- **WCF services can be hosted in IIS or WAS, or they can be self-hosted.**
  - We will use self-hosting throughout this course.
- **WCF services and clients can be configured in code or via XML configuration files.**
- **You can implement a client of a WCF service using a Channel Factory.**
- **A WCF service can be configured to export metadata, which can be used to automatically generate a proxy for a service.**

## Lab 1

### Creating a Simple WFC Service and Client

#### Introduction

In this lab, you will use Visual Studio 2015 to create a simple WCF service that is self-hosted. The metadata exchange endpoint is implemented with the help of starter code provided by Visual Studio. You will create a simple Console client program. Create the proxy by adding a Service Reference using Visual Studio. You will also experiment with changing the binding from **basicHttpBinding** to **wsHttpBinding**.

**Suggested Time:** 50 minutes

**Root Directory:** OIC\WcfCs

**Directories:** Labs\Lab1 (do your work here)  
Chap01\SimpleMath (answer)

#### Part 1: Create the Service Library

1. Start Visual Studio 2015 as Administrator. Create a new blank solution **SimpleMath**.
2. Add a new project **MathLib** to your solution using the WCF Service Library template.
3. Rename the file **IService1.cs** to **IMath.cs** and the file **Service1.cs** to **MathService.cs**.
4. In the files **IMath.cs** and **MathService.cs**, delete the starter code except for the namespace imports and the declaration of the **MathLib** namespace.
5. In the file **IMath.cs**, provide the following simple service contract **IMath**.

```
namespace MathLib
{
    [ServiceContract]
    public interface IMath
    {
        [OperationContract]
        int Add(int x, int y);
        [OperationContract]
        int Subtract(int x, int y);
    }
}
```

6. In the file **MathService.cs**, provide the following implementation:

```
namespace MathLib
```

```

{
    public class MathService : IMath
    {
        public int Add(int x, int y)
        {
            return x + y;
        }

        public int Subtract(int x, int y)
        {
            return x - y;
        }
    }
}

```

7. Edit **App.config** to make sure the service name is **MathLib.MathService** and the contract **MathLib.IMath**. (Visual Studio should have done this rename for you when you changed the names of the files.)
8. Build and run the solution. This should start up both the test host and the test client. Exercise both Add and Subtract.

## Part 2: Create the Host

1. Add a new Console Application **Host** to your solution.
2. Add a reference to the **MathLib** project and the **System.ServiceModel** assembly.
3. In **Program.cs** import the namespaces **System.ServiceModel** and **MathLib**.
4. Provide code in **Main()** to initialize and open a **ServiceHost** object.

```

static void Main(string[] args)
{
    using (ServiceHost host = new ServiceHost(
        typeof(MathService)))
    {
        host.Open();
        Console.WriteLine("Press ENTER to terminate service host");
        Console.ReadLine();
    }
}

```

5. Copy the **App.config** file from the **MathLib** project to the **Host** project and add it to the **Host** project. (You can do this by drag and drop in Solution Explorer.) Edit the base address. Note that we will go with the empty string for the contract address, so we provide a complete URI for the base address. You can delete **App.config** from **MathLib**.

```

<system.serviceModel>
  <services>
    <service name="MathLib.MathService">
      <endpoint address="" binding="basicHttpBinding"
        contract="MathLib.IMath">
        <identity>

```

```

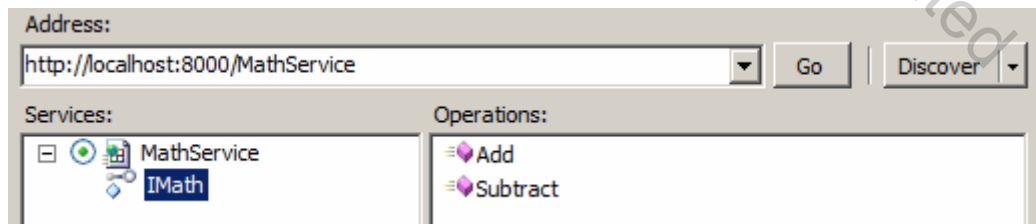
        <dns value="localhost" />
    </identity>
</endpoint>
<endpoint address="mex" binding="mexHttpBinding"
    contract="IMetadataExchange" />
<host>
    <baseAddresses>
        <add baseAddress="http://localhost:8000/MathService" />
    </baseAddresses>
</host>
</service>
</services>
<behaviors>
    <serviceBehaviors>
        <behavior>
            <!-- To avoid disclosing metadata information,
            set the value below to false and remove the metadata endpoint
            above before deployment -->
            <serviceMetadata httpGetEnabled="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="False" />
        </behavior>
    </serviceBehaviors>
</behaviors>
</system.serviceModel>

```

6. Build the solution. Test that the host will start up.

### Part 3: Implement the Console Client

1. Add a new Console project **Client** to your solution.
2. Run the host as Administrator outside of Visual Studio. Verify that it has been correctly implemented by pointing your browser to **http://localhost:8000/MathService**. You should see a page for the **MathService** service. Copy this URI into the clipboard.
3. In the **Client** project right-click over References and choose Add Service Reference.
4. Paste the URI (**http://localhost:8000/MathService**) into the Address: text box.
5. Click Go. Expand the tree to show the **IMath** contract.



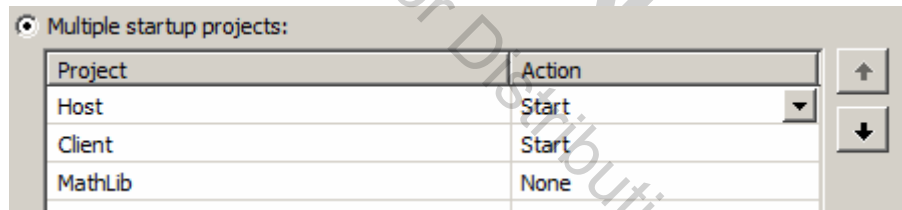
6. Click OK.
7. Close the host.
8. In **Program.cs** import the namespace **Client.ServiceReference1**.
9. Implement a simple test program to assign values to integer variable x and y and invoke both methods of the contract through the proxy.

```

class Program
{
    static void Main(string[] args)
    {
        int x = 7;
        int y = 5;
        using (MathClient proxy = new MathClient())
        {
            int sum = proxy.Add(x, y);
            Console.WriteLine("Sum of {0} and {1} is {2}", x, y, sum);
            int diff = proxy.Subtract(x, y);
            Console.WriteLine("Difference of {0} and {1} is {2}",
                x, y, diff);
        }
        Console.WriteLine("Press ENTER to terminate client");
        Console.ReadLine();
    }
}

```

10. Configure the properties of the solution for multiple startup projects (**Host** and **Client**) with **Host** starting first.



11. Build and run the solution. Verify the output.

```

Sum of 7 and 5 is 12
Difference of 7 and 5 is 2
Press ENTER to terminate client

```

12. Close the client and then the host.

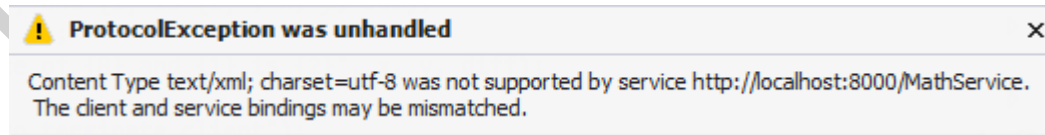
13. Now let's try using a different binding. Edit **App.config** in the Host project to call for **wsHttpBinding**.

```

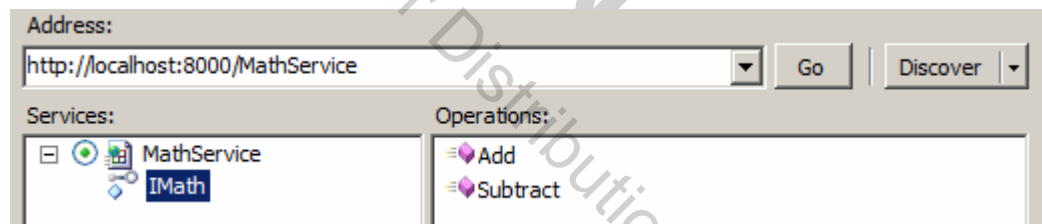
<service name="MathLib.MathService">
  <endpoint address="" binding="wsHttpBinding"
    contract="MathLib.IMath">

```

14. Rebuild and run the solution. The host will startup fine, but you will hit a protocol exception in the client.



15. There is a helpful error message that the client and service bindings may be mismatched. Terminate debugging.
16. If the host is closed, run it again as Administrator outside of Visual Studio.
17. In the Client project delete the Service Reference.
18. Add a Service Reference using the same procedure we used before.
- Run the host as Administrator outside of Visual Studio.
  - In the **Client** project right-click over References and choose Add Service Reference.
  - Enter the URI **http://localhost:8000/MathService** into the Address: text box.
  - Click Go. Expand the tree to show the **IMath** contract. (You could omit expanding the tree – this part is simply to build confidence you have the right service.)



- Click OK.
19. Close the Host.
20. Rebuild and run the solution. Now it should work again!



## **Chapter 8**

### **WCF Security**

EVALUATION COPY  
Unauthorized Reproduction or Distribution Prohibited

# WCF Security

## Objectives

---

*After completing this unit you will be able to:*

- Understand the security aspects of services.
- Explain the difference between Transport and Message security.
- Configure WCF services and clients to communicate over a secure channel.
- Work with certificates to use more security features from WCF.
- Understand how to send credentials from a client to a service.

## Services and Security

---

- **So far we've discussed WCF services without taking security concerns in consideration.**
- **Securing distributed services is about securing communication among different software entities.**
  - The scenarios involved are somewhat similar to traditional client-server applications, but with some nuances that will be covered throughout this chapter.
- **For example, very frequently we are not able to rely on the medium where the message is being transmitted to ensure confidentiality.**
  - Third party software could have access to the messages exchanged between client and service.
  - So it is imperative that actions are taken to ensure that only client and service can read those messages.
- **Several security aspects must be considered when defining how you will secure a service.**

# Security Aspects of Services

---

- **Confidentiality**

- The transmissions should be kept safe in such a way that parties other than the intended receiver are not able to read or understand the content.

- **Integrity**

- The receiver should be assured that the transmission contents were not changed on its way from the sender.

- **Authentication**

- This involves identifying both ends (service and client) in the communication channel.

- **Authorization**

- Once identified, the service must know whether the client is entitled to execute an operation or not.

- **Impersonation**

- The service might be required in some cases to perform operations on behalf of the client.

# Transfer Security

---

- **WCF offers a number of options for securing the information exchanged between a WCF service and a client.**
  - Securing the information on its way from the server to the client is about taking care of two security aspects already mentioned: integrity and confidentiality.
  - This foundational aspect of security is called **transfer security**.
- **We can configure a WCF service to work with the following security modes: Transport, Message, Mixed, Both and None.**
- **Choosing a security mode requires analysis of the specific implementation scenario.**
  - Each mode has characteristics that are suitable for different types of scenarios.
  - We will see some examples in this chapter that will help you decide on the security mode to be used for some of the most popular WCF implementation scenarios.

# Transport Security

---

- **The Transport security mode delegates the security-related work on the communication to the transport layer of the network.**
  - Security is handled by the communication protocol, such as HTTPS or TCP.
- **It provides integrity, confidentiality and authentication.**
  - Integrity and confidentiality exist because transport security encrypts all communication between service and client, so no one can read the messages without the encryption key.
  - Since the client's credentials are contained in the encrypted message, a good level of authentication is also achieved.
- **It is a quick way of implementing security, and no processing is done on the application layer.**
  - For this reason, Transport security has also the best performance among the security modes.
- **The negotiation of encryption details between client and service is done automatically by the communication protocol within the binding used.**

# Scenarios for Transport Security

---

- **Since all the processing is done on the transport layer, it is the best approach to use when a WCF service communicates with a non-WCF client.**
- **Transport security is great for intranet scenarios, where both service and client are on the same network.**
  - The reason for this is that reliance on the transport layer means that this security mode can only guarantee transfer security point-to-point.
  - Therefore, Transport security scenarios must consider service and client connecting directly with each other, with no intermediaries.
- **Transport Security works with all bindings except WSFederationHttpBinding and WSDualHttpBinding.**
- **When you use the following bindings, Transport Security will be enabled by default if you don't specify it in the configuration:**
  - NetTcpBinding
  - NetPeerTcpBinding
  - NetNamedPipeBinding
  - NetMsmqBinding

# Configuring Transport Security

---

- **The most practical way of setting up security in a WCF application is through the configuration file.**
  - Security configuration can also be set up through code, although it is more cumbersome due to the large number of properties.
- **Typically, this is how the configuration looks like:**

```
<system.serviceModel>
...
  <bindings>
    <netTcpBinding>
      <binding name="Secured">
        <security mode="Transport">
          <transport
            clientCredentialType="Windows"
            protectionLevel="EncryptAndSign"
          />
        </security>
      </binding>
    </netTcpBinding>
  </bindings>
```



# Transport Security Example

---

- **Let's see a simple example of a client and a service running on Transport security.**
  - Open the solution in the **Transport** folder in the chapter directory.
- **Build the solution and run the host and the client separately<sup>1</sup>.**
  - Verify that the communication between client and service is working by typing a name and clicking the Greet button. You should see a hello message back from the service in the screen.
- **Now let's take a look at how these applications are set up to communicate securely.**
  - Open **App.config** on the **Host** project.
  - The first to notice on this file when compared to the others we have been dealing with is that the service endpoint has the **bindingConfiguration** property explicitly set.

```
<service behaviorConfiguration="serviceBehavior"
    name="HelloLib.Hello">
  <endpoint address="Hello"
    binding="netTcpBinding" name="netTcp"
    contract="HelloLib.IHello"
    bindingConfiguration="Secured" />
```

...

---

<sup>1</sup> Make sure that you have allowed access through a fire wall before exercising the client.

# Host's Security Configuration

---

- **The next different thing about this configuration file is the <bindings> section.**

- It includes details on security configuration for the binding.
- Note that the binding configuration here has the same name set on the endpoint bindingConfiguration property.

```
<netTcpBinding>
  <binding name="Secured">
    <security mode="Transport">
      <transport clientCredentialType="Windows"
        protectionLevel="EncryptAndSign" />
    </security>
  </binding>
</netTcpBinding>
```

- **Notice the values used for clientCredentialType and protectionLevel properties.**

- The clientCredentialType property is set to Windows, which means that the identity that is running the application process will be sent as a credential.
- The protectionLevel property is set to EncryptAndSign, which means that the service will encrypt the message contents and also append an encrypted checksum representing its “signature” to each message. This provides integrity, privacy and authenticity in the communication.

## Client's Security Configuration

---

- **The configuration on the client is just a bit different from the other examples we've seen.**

```
<netTcpBinding>
  <binding name="netTcp" closeTimeout="00:01:00"
    ...
    transferMode="Buffered">
    <security mode="Transport">
      <transport clientCredentialType="Windows"
        protectionLevel="EncryptAndSign"/>
    </security>
  </binding>
</netTcpBinding>
```

- **Basically, what we see in the client is a configuration that must match the security configuration from the service.**
  - If you try to set the client with Message security and the service with Transport security, an exception will be raised at runtime when a service method is first called.

# Message Security

---

- **When using Message security, the whole message exchanged between service and client is encrypted.**
- **This security mode doesn't rely on the transport layer, since message encryption happens on the application as a WCF feature.**
- **Security is provided end to end, since only client and server can read the contents of the encrypted message.**
- **The use of Message security may introduce a performance hit due to the overhead of encrypting and decrypting messages.**
  - This was not a concern when using Transport security as this work was delegated to the Transport layer.

## Scenarios for Message Security

---

- **Since this transfer mode provides security end to end, it is ideal for Internet scenarios.**
  - In those scenarios, the client and the server are far away from each other so that the message may travel through unknown intermediaries.
  - Hence, ensuring that only the service and the intended client can read messages from each other is imperative.
- **Message Security works with all bindings except NetNamedPipeBinding.**
- **When you use the following bindings, Message Security will be enabled by default if you don't specify it in the configuration:**
  - WSHttpBinding
  - WSFederationHttpBinding
  - WSDualHttpBinding

## Configuring Message Security

---

- **This is how the typical configuration for Message security would look like on the service side:**

```
<system.serviceModel>
...
  <bindings>
    <netTcpBinding>
      <binding name="Secured">
        <security mode="Message">
          <message
            clientCredentialType="Windows" />
        </security>
      </binding>
    </netTcpBinding>
  </bindings>
```

- **The solution in the *Message* folder in the chapter directory contains an example that is very similar to the one shown for Transport security.**
  - The only differences are the Message setting for the security mode, as shown above, and the corresponding change also in the client configuration file.

## Other Security Modes

---

- **Mixed**

- This setting makes WCF use Transport security for confidentiality, integrity and authentication, and also uses Message security to allow sending credentials.
- With this option, we can take advantage of the performance of Transport security and the configuration flexibility of Message security.
- The downside is that we are limited by the Transport security constraints, such as only ensuring security from a point-to-point perspective.

- **Both**

- With this mode, both Transport and Message security are used.
- Using this option will result in processing overhead, since security features will be enabled on both the transport and application layers.

- **None**

- With this setting, all security features from WCF are turned off.

# Certificates

---

- **In order to explore other security scenarios in WCF, we will need to use certificates.**
- **For the purposes of this course, we will create and install test certificates.**
  - In a real world scenario, normally the certificates are purchased from a certificate authority such as VeriSign.
- **The idea behind the use of certificates for service and client communication works as follows:**
  - Assume that there is a Company A which hosts a service and a Company B which has a client application that consumes that service.
  - Company A purchases a certificate and installs it on the machine that runs the service. Then, Company A sends an exported version of that certificate to Company B, which in turn installs that certificate as “trusted” on the machine that runs the client.
  - Similarly, Company B purchases a certificate and installs it on the machine that runs the client. Then, Company B sends an exported version of that certificate to Company A, which in turn installs that certificate as “trusted” on the machine that runs the service.
- **This approach is often called “peer trust”.**



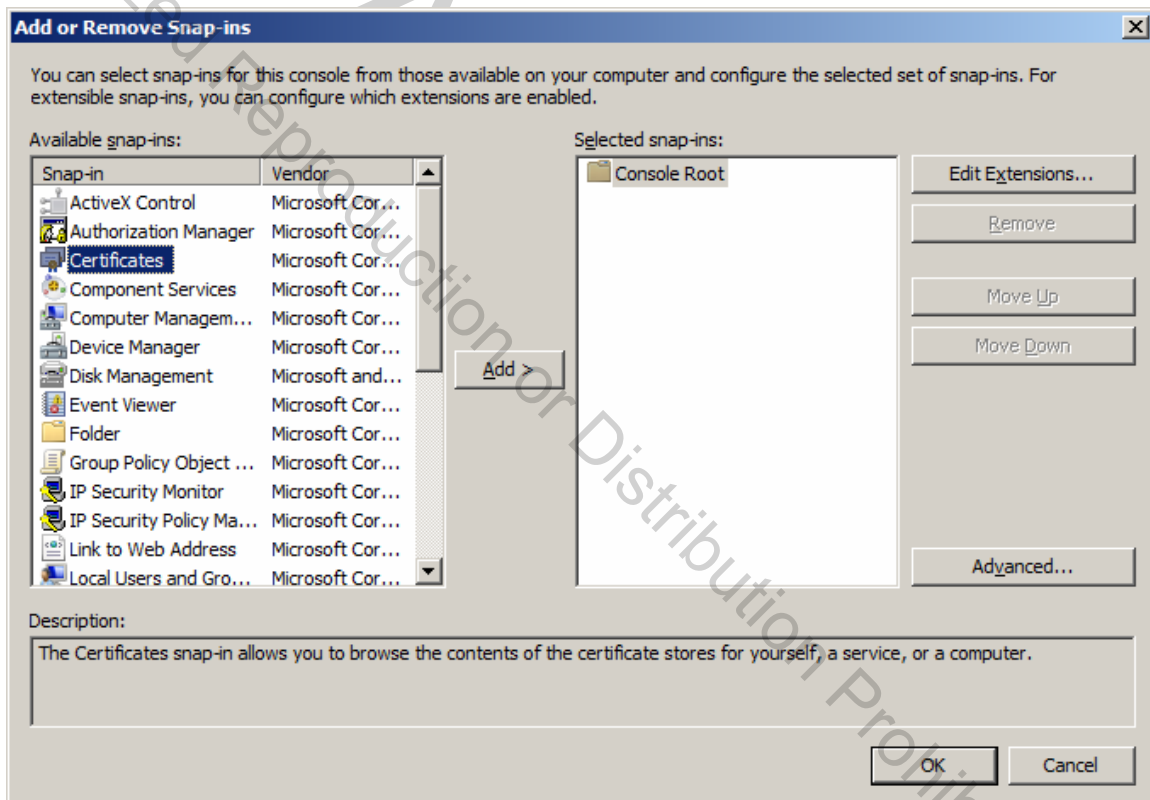
## Certificate Demo

---

- **Let's illustrate how to generate test certificates and use them in a sample WCF application.**
  - For the sake of simplicity, we will do this demo considering that both service and client are running on the same computer. However, the process will be very similar in case you want to run it with service and client on separate computers.
  - Note that you must complete at least Steps 1 through 11 in order to run the supplied solutions for both the **Certificate** example and also the following **UserNameCredential** example, as well as the lab.
- 1. Open a Visual Studio 2015 command prompt, running as Administrator, and navigate to the folder **Demos\Certificate** in the course directory.
- 2. Run the batch file **Cert.bat**.
  - This batch file creates test certificates for the service (OIWCFSservice.cer) and for the client (OIWCFCClient.cer).
  - Note that the test certificate must be generated on the computer that will run the application that will use the certificate to represent itself.

# Managing Certificates

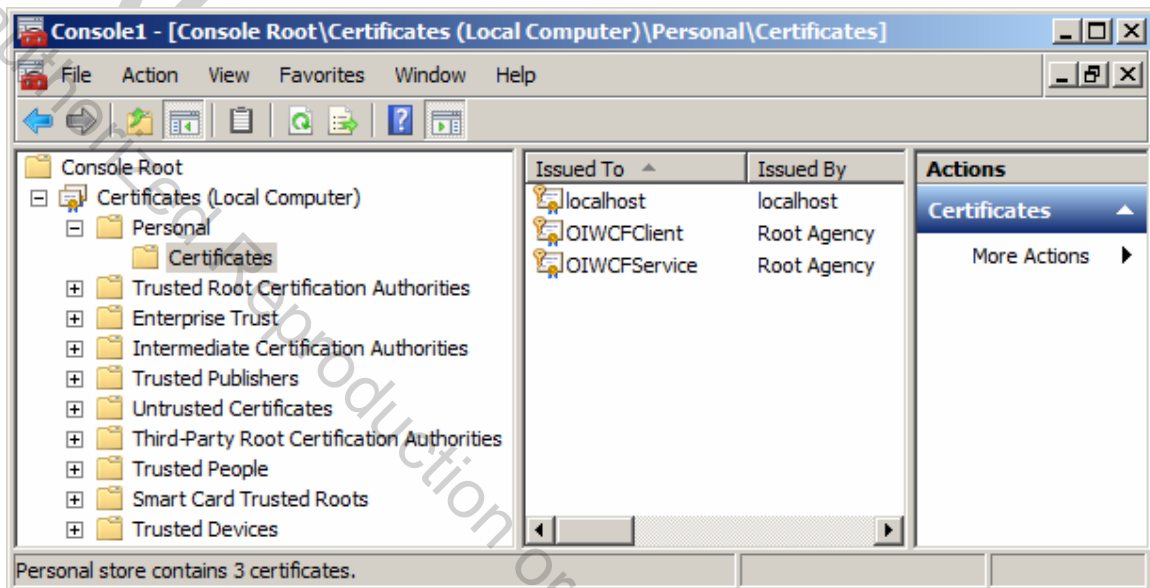
3. In order to manage the certificates mentioned on this chapter, you will need to create a MSC Console with the Certificates snap-in. You can create this by following these steps:
  - Using the Visual Studio 2015 command prompt that you have running, type “mmc” and click **OK**.
  - On the empty MMC Console that opens, click **File**, and then **Add/Remove Snap-in...**



- Select “Certificates”, then click **Add**, select the option “**Computer account**”, and click **Next**. Leave selected: snap-in will manage the local computer, and click **Finish** and **OK**.
- You can save this MMC Console for your future use. Go to **File**, then **Save As...**, and create a msc file on your desktop.

## Certificate Demo (Cont'd)

4. Verify that the certificates are installed on the computer by opening the Certificates snap-in and navigating to Personal\Certificates.
  - Both OIWCFSservice and OIWCFCClient are listed.



- Both the service and the client have their certificates in store to be used when communicating with each other.
  - However, we still need to configure the service certificate to be trusted by the client application, and the client certificate to be trusted by the service.
5. Right-click the OIWCFSservice certificate, point to **All Tasks**, click **Export...**, and then click **Next**.
  6. Say no to the option of exporting the certificate with the private key and click **Next**.
  7. Keep the default format for the certificate (DER encoded binary X.509) and click **Next**.

## Certificate Demo (Cont'd)

---

8. Choose a file name and location for the exported version of the certificate, click **Next** and then click **Finish**. Repeat steps 5 to 8 for the OIWCFCClient certificate.
9. On the Certificates Snap-in, right-click the Trusted People folder, point to All Tasks, click Import..., and then click Next.
10. Put in the path to the exported version of the OIWCFSservice certificate you saved at step 8 and click **Next**.
11. The next screen shows that the certificate will be saved at the Trusted People store, which is what we want, so click **Next**, and then **Finish**. Repeat steps 9 to 11 for the exported version of the OIWCFCClient certificate.
- **Now that we have the necessary certificates properly set up, let's work on the host and client configuration.**
12. Open the solution in the **Demos\Certificate** folder and run both the host and the client to make sure they are working properly.
  - On the client, you should see a greeting message when you type your name and click the Greet button.

## Certificate Demo (Cont'd)

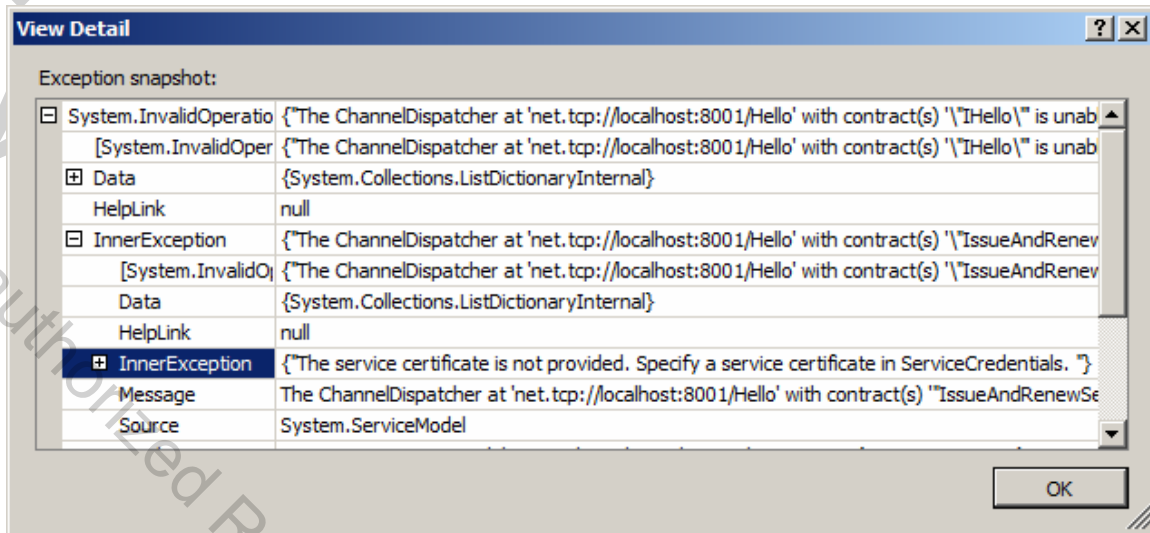
---

13. Open the **App.config** file on the **Host** project. You will notice that it is set to use Message security, with Windows credential type. Change it to use Certificate credential type instead.

```
...  
<binding name="Secured">  
  <security mode="Message">  
    <message clientCredentialType="Certificate"/>  
  </security>  
</binding>
```

14. Build and run the host under the debugger. You will hit an exception, which is pretty much unclear.
- It is very common in WCF that the exceptions are wrapped in multiple layers, so commonly you can find out more details on what happened behind the scenes by digging into the exception's **InnerException** property.
  - You can do that by clicking the **View Detail...** link in Visual Studio when the exception is displayed to see the real issue.

## Exception Details



- The real issue is that the certificate to be used by the service was not provided.

## Certificate Demo (Cont'd)

---

15. Let's add the configuration that solves this issue by specifying a service certificate in the **App.config** file in the **Host** project.

```
...
<behavior name="serviceBehavior">
  <serviceMetadata httpGetEnabled="true" />
  <serviceCredentials>
    <clientCertificate>
      <authentication
        certificateValidationMode="PeerTrust"
        revocationMode="NoCheck"/>
    </clientCertificate>
    <serviceCertificate findValue="OIWCFService"
      storeLocation="LocalMachine"
      storeName="My"
      x509FindType="FindBySubjectName"/>
  </serviceCredentials>
</behavior>
```

- The `clientCertificate` node is specifying how the certificates sent by clients should be validated, meaning basically that the service will look for the client's certificate in the Trusted People store to see whether it should be accepted or not.
  - The `serviceCertificate` element specifies search parameters that indicate where to find the certificate to be used by the service on the local store.
16. Build and run the host. Now the service should be started normally, using the proper certificate.
17. You can now stop the service. You are now at Step 1. A copy of this solution at this point is saved in **Certificate\Step1** in the project directory.

# Client Certificate Configuration

---

- **Now let's add the certificate configuration to the client.**

1. Open the **app.config** file on the **HelloWin** project. You will notice that it is set to use Message security, with Windows credential type. Change it to use Certificate credential type instead.

```
...  
<binding name="Secured" closeTimeout="00:01:00"  
    openTimeout="00:01:00" ...  
    ...  
    <security mode="Message">  
        <message clientCredentialType="Certificate"  
            algorithmSuite="Default" />  
    </security>  
</binding>
```

2. As we learned when configuring the server, we also need to “tell” the client how to find its certificate in the Local Computer store. You can do that by adding a behavior configuration to the config file, which should include the `clientCredentials` configuration element.



# Client Certificate Configuration

## (Cont'd)

---

```
<system.serviceModel>
...
  <behaviors>
    <endpointBehaviors>
      <behavior name="clientBehavior">
        <clientCredentials>
          <clientCertificate
            findValue="OIWCFClient"
            storeLocation="LocalMachine"
            storeName="My"
            x509FindType="FindBySubjectName"/>
          <serviceCertificate>
            <authentication
              certificateValidationMode=
                "PeerTrust"
              revocationMode="NoCheck"/>
          </serviceCertificate>
        </clientCredentials>
      </behavior>
    </endpointBehaviors>
  </behaviors>
</system.serviceModel>
```

- Note that the roles of `clientCertificate` and `serviceCertificate` elements are inverted when compared to the configuration on the service side. Here, the `clientCertificate` element is used to find the client certificate and the `serviceCertificate` is used to configure how the certificate sent by the service should be validated.

# Client Certificate Configuration

## (Cont'd)

---

3. Specify the `behaviorConfiguration` property on the service endpoint to point to the configuration created on the previous step.

```
<endpoint address="..." name="netTcp"
  binding="netTcpBinding"
  contract="ServiceReference1.IHello"
  bindingConfiguration="netTcp"
  behaviorConfiguration="clientBehavior">
</endpoint>
```

4. Build and run the host and the client, type in a name and click the **Greet** button. You will hit an exception saying that the identity check failed for the outgoing message. That can be fixed by specifying a DNS identity for the service endpoint on the client side.

```
<endpoint address=...
  <identity>
    <dns value="OIWCFService"/>
  </identity>
</endpoint>
```

5. Build and run the host and the client, type in a name and click the **Greet** button. Now your WCF solution secured using certificates works perfectly! You are now at Step 2.

- **A copy of this solution is saved in *Certificate\Step2* folder in the chapter directory.**

## Sending Credentials

---

- **In some scenarios, you may be interested in passing credentials from the client to the service.**
  - For example, you may want to implement authorization by giving some clients permission to execute a given operation, but not to others.
- **Here are some alternatives to achieve that:**
  - Windows credentials: the client can pass a windows credential to the service (by default, the one that is running the client's process).
  - ASP.NET Role Provider: you can use the role provider shipped with .NET to authorize users into the service methods.
  - Custom user name validation: you can implement a custom class that will check the username and password from the client at the service.

# Username Credentials

---

- **Let's see a simple example that passes a username and a password to a WCF service.**
  - Open the solution in the **UserNameCredential** folder in the chapter directory.
- **Build and run the host and the client. Type a message on the client and click on the Greet button.**
  - On the server console a message is displayed indicating that the client was able to call the service, which also included the username.

Press ENTER to terminate the service host  
Method SayHello called by user 'oi'

- **Now let's see how this was implemented.**

## Username Example

---

- **Open *App.config* on the Host project. On the binding configuration, the client credential type is set to `UserName`.**

```
<binding name="Secured">
  <security mode="Message">
    <message clientCredentialType="UserName" />
  </security>
</binding>
```

- **When using `UserName` as the client credential type, WCF still requires that a certificate is provided in the communication between client and server.**
  - The reason for that is that a username and password pair allows implementing authorization, but not authentication.
- **Now let's take a look at the `serviceCredentials` element in the service behavior configuration.**

```
<serviceCredentials>
  <userNameAuthentication
    customUserNamePasswordValidatorType=
      "Host.CustomUserNameValidator, Host"
    userNamePasswordValidationMode="Custom" />
  <clientCertificate>
    <authentication
      certificateValidationMode="PeerTrust"
      revocationMode="NoCheck" />
  </clientCertificate>
  <serviceCertificate findValue="OIWCFService"
    storeLocation="LocalMachine" storeName="My"
    x509FindType="FindBySubjectName" />
</serviceCredentials>
```

## Username Example (Cont'd)

---

- **The usernameAuthentication element informs that a custom class was provided to validate credentials.**

- Let's take a look at this class by opening **CustomUserNameValidator.cs** in the **Host** project.

```
public class CustomUserNameValidator :
    UserNamePasswordValidator
{
    public override void Validate(
        string userName, string password)
    {
        if (null == userName || null == password)
        {
            throw new ArgumentNullException();
        }

        if (!(userName == "oi" &&
            password == "io"))
        {
            throw new SecurityTokenException(
                "Unknown username or incorrect password");
        }
    }
}
```

- **For the sake of simplicity, this example just validates a hardcoded username and password.**
- You could extend this approach to use potentially any custom credentials provider.

## Username Example (Cont'd)

---

- **Providing credentials on the client side is very simple.**
  - Open **Form1.cs** on the **HelloWin** project.
  - Note that the client credentials are provided programmatically when the service proxy is created.

```
HelloClient proxy = new HelloClient();  
proxy.ClientCredentials.UserName.UserName = "oi";  
proxy.ClientCredentials.UserName.Password = "io";  
txtResponse.Text = proxy.SayHello(txtName.Text);
```

- **Finally, if you want to get information about the identity of the client at the service side, you can use the *ServiceSecurityContext* class.**
  - See **Hello.cs** in the **HelloLib** WCF library for an example.

```
Console.WriteLine(String.Format(  
    "Method SayHello called by user '{0}'",  
    ServiceSecurityContext.Current.PrimaryIdentity.Name  
));
```

# Lab 8

## Custom User Credentials and Authorization

In this lab you will improve an unsecured contact manager application adding security configuration, certificates support and a custom username validation mechanism. You will also implement authorization by making sure only a specific user can access a given method.



Detailed instructions are contained in the Lab 8 write-up at the end of the chapter.

Suggested time: 90 minutes



## Summary

---

- **Securing services is about securing the communication between different software entities.**
- **Implementing service security involves aspects such as confidentiality, integrity, authentication and authorization.**
- **Transport and Message security modes are available to be used in WCF services depending on the application scenario.**
- **Most of the security configuration of a WCF service can be done in the config file.**
- **Using certificates to implement security allows us to also implement other security mechanisms, such as username credentials.**

## Lab 8

### Custom User Credentials and Authorization

#### Introduction

In this lab you will improve an unsecured contact manager application adding security configuration, certificates support and a custom username validation mechanism. You will also implement authorization by making sure only a specific user can access a given method.



**Suggested Time:** 90 minutes

**Root Directory:** OIC\WcfCs

**Directories:**

<b>Labs\Lab8\ContactMgr</b>	(do your work here)
<b>Chap08\ContactMgr\Step0</b>	(starter code backup)
<b>Chap08\ContactMgr\Step1</b>	(answer to part 1)
<b>Chap08\ContactMgr\Step2</b>	(answer to part 2)
<b>Chap08\ContactMgr\Step3</b>	(answer to part 3)

#### Part 1: Modify Security Configuration

1. Open the starter solution. There are three projects: a service library, a host, and a Windows client program. Build the solution and start the host. Then start the Windows client. You will be able to Add, Remove and Modify contacts. A user interface is provided for credentials, but there is no code yet.
2. On the service library project, open the file **ContactManager.cs**. Notice the implementation of the service methods for adding, removing and modifying a contact.

3. Since we want to support UserName credentials in this application, let's configure it to work with certificates first. Open the file **App.config** in the **Host** project.
4. Add a binding configuration section indicating that the Message security mode will be used and that the client credential type will be Certificate. Remember to give a name to this binding configuration so you can refer it in the endpoint in the next step.

```

...
</services>
<bindings>
  <netTcpBinding>
    <binding name="Secured">
      <security mode="Message">
        <message clientCredentialType="Certificate"/>
      </security>
    </binding>
  </netTcpBinding>
</bindings>
</system.serviceModel>

```

5. On the endpoint definition, specify the name of the binding configuration you just created.

```

<endpoint address="Hello" binding="netTcpBinding"
  name="netTcp" contract="ContactLib.IContactManager"
  bindingConfiguration="Secured"/>

```

6. Now, add configuration to a reference to the **OIWCFService** certificate, which was created previously in this chapter. You should also add configuration for the client certificate to be validated using the **PeerTrust** mode, and the revocation mode should be set to **NoCheck**.

```

...
<serviceMetadata httpGetEnabled="true"/>
<serviceCredentials>
  <clientCertificate>
    <authentication certificateValidationMode="PeerTrust"
      revocationMode="NoCheck"/>
  </clientCertificate>
  <serviceCertificate findValue="OIWCFService"
    storeLocation="LocalMachine" storeName="My"
    x509FindType="FindBySubjectName"/>
</serviceCredentials>
</behavior>
</serviceBehaviors>

```

7. Run the host project to ensure the host can be started with no configuration errors.
8. Now let's configure security for the client to support certificates. Open the file **app.config** on the **ContactWin** project.

9. On the binding configuration section, modify the security mode to **Message**. Remove the transport security node, since we are using message security for this application, and then change the client credential type to **Certificate** on the message node.

```
<bindings>
  <netTcpBinding>
    <binding ...
      ...
      <security mode="Message">
        <message clientCredentialType="Certificate" />
      </security>
    </binding>
  </netTcpBinding>
</bindings>
```

10. Add a behavior configuration section indicating that client credential used will be a certificate. You should also add configuration for the service certificate to be validated using the **PeerTrust** mode, and the revocation mode should be set to **NoCheck**. Remember to give a name to this behavior configuration so you can refer it in the endpoint in the next step.

```
...
<behaviors>
  <endpointBehaviors>
    <behavior name="Secured">
      <clientCredentials>
        <clientCertificate findValue="OIWCFClient"
          storeLocation="LocalMachine" storeName="My"
          x509FindType="FindBySubjectName"/>
        <serviceCertificate>
          <authentication certificateValidationMode="PeerTrust"
            revocationMode="NoCheck"/>
        </serviceCertificate>
      </clientCredentials>
    </behavior>
  </endpointBehaviors>
</behaviors>
</system.serviceModel>
```

11. Specify the name of the newly created behavior configuration within the endpoint definition.

```
<endpoint address="net.tcp://localhost:8001/Hello"
  binding="netTcpBinding" bindingConfiguration="netTcp"
  contract="ServiceReference1.IContactManager"
  name="netTcp" behaviorConfiguration="Secured">
```

12. To complete the certificate configuration on the client, you must set up the DNS identity of the service so that the service certificate can be properly validated. In order to do this, add a **dns** configuration element to the identity configuration of the endpoint with the value **OIWCFService**, since this is assumed to be the DNS name used by the service that owns the OIWCFService certificate. You can also remove the **userPrincipalName** configuration which was added by default by WCF when

creating that configuration file, which won't be needed in our scenario. Here is how the endpoint configuration should look like after your changes:

```
<endpoint address="net.tcp://localhost:8001/Hello"
          binding="netTcpBinding" bindingConfiguration="netTcp"
          contract="ServiceReference1.IContactManager"
          name="netTcp" behaviorConfiguration="Secured">
  <identity>
    <dns value="OIWCFSservice"/>
  </identity>
</endpoint>
```

13. Run the host and then the client. Try to do any operation (Add, Remove or Modify) to ensure the security configuration you've just done didn't break the communication between the service and the client. You are now at Step 1.

## Part 2: Implement Username Credentials Support

1. In this part, we will slightly modify the configuration we just implemented to use Username credentials, which requires that the client and the service are properly set up with certificates first. In this lab we won't use any existing credentials provider like ASP.NET Role Provider or Windows Authentication. We will implement a simple custom username authentication class just for illustration purposes, which you can easily extend to use other custom authentication providers. Let's start by adding a new class file to the host project, by right-clicking the project and selecting Add, then New Item... and then the Class template. Provide **CustomUserNameValidator.cs** as the file name and click OK.
2. We can implement a custom username validator in WCF by extending the **UserNamePasswordValidator** class from the **System.IdentityModel.Selectors** namespace. Before using this namespace, you must add a reference to **System.IdentityModel** on the Host project.
3. Now, implement the **CustomUserNameValidator** class by inheriting the **UserNamePasswordValidator** class and overriding the Validate method, which takes a username and a password as parameters. Note that these credentials will be passed by the client and will reach this code before reaching the service methods, so this is the place to validate and throw the proper exceptions in case the credentials are incorrect. For the sake of simplicity, let's just implement the verification of two acceptable credentials, one for a plain user and another for a manager.

```
class CustomUserNameValidator : UserNamePasswordValidator
{
    public override void Validate(string userName, string password)
    {
        if (userName == null || password == null)
        {
            throw new ArgumentNullException();
        }
    }
}
```

```

    }

    if (!(userName == "user" && password == "user") &&
        !(userName == "manager" && password == "manager"))
    {
        throw new SecurityTokenException(
            "Unknown username or incorrect password");
    }
}

```

4. Import the **System.IdentityModel.Selectors** and the **System.IdentityModel.Tokens** namespaces, since we are using the classes **UserNamePasswordValidator** and **SecurityTokenException** respectively in the code.
5. Open the **App.config** file in the **Host** project. Configure the host to use username authentication by changing the **clientCredentialType** property in the message configuration for the binding.

```

<binding name="Secured">
  <security mode="Message">
    <message clientCredentialType="UserName"/>
  </security>
</binding>

```

6. On the behavior configuration, add a **userNameAuthentication** element to the **serviceCredentials** node, specifying the validation mode as Custom and the class that will be responsible for the validation.

```

<serviceCredentials>
  ...
  <userNameAuthentication
    customUserNamePasswordValidatorType =
      "Host.CustomUserNameValidator, Host"
    userNamePasswordValidationMode="Custom"/>
</serviceCredentials>

```

7. Let's now add some logging to the service to let us know the name of the user who is calling the service operations. On the **ContactManager** class in the service library, add a **LogCall** method which takes a method name as a string parameter.

```

private void LogCall(string methodName)
{
    string currentUser =
        ServiceSecurityContext.Current.PrimaryIdentity.Name;
    Console.WriteLine(String.Format("Method {0} called by user {1}",
        methodName, currentUser));
}

```

8. Inside the method **AddContact**, call **LogCall** passing the method name. Do the same for **RemoveContact**, **ModifyContact** and **GetContacts**.

```

public void AddContact(Contact cont)

```

```
{
    LogCall("AddContact");
    contacts.Add(cont);
}
```

9. Build and run the host application, and then run the client. You will hit a security-related exception, since now the service is expecting username credentials to be passed from the client.
10. Let's now configure the client to work with username credentials. Open **app.config** on the client project and, on the binding configuration, change the message **clientCredentialType** property to **UserName**.

```
<security mode="Message">
  <message clientCredentialType="UserName" />
</security>
```

11. In order to provide username credentials on the client, you need to set the proxy's **ClientCredentials.UserName** property to appropriate values. For now, just to test the connectivity with the service, set the username and password properties to the hard coded values we used in the proxy on the **Form1\_Load** method in the **Form1.cs** file. Note that this must be done before the first call to a service operation.

```
private void Form1_Load(object sender, EventArgs e)
{
    proxy = new ContactManagerClient();
    proxy.ClientCredentials.UserName.UserName = "user";
    proxy.ClientCredentials.UserName.Password = "user";
    ShowContacts();
}
```

12. Build and run host and client. Notice that now the client is back working again with the service, since client credentials are being provided. Also, on the host console window, you will see a log message saying that the **GetContacts** method was called by the user "user". You are now at Step 2.

### Part 3: Implement Authorization

In this part, we will use the modified client interface to allow providing credentials on the fly. We will implement a simple authorization step on the service to require that a specific user credential is provided to execute a given operation. In **Form1.cs** in the client project there are two text boxes to allow the user to enter a username and a password, and a **Set** button to validate the credentials. The **PasswordChar** property of the password text box is set to \*, to avoid displaying the text while entering the value.

1. The application at this point is creating the proxy and loading contacts when the form loads. We will have to change this behavior, since we don't know yet which credentials to use when the form loads. Our purpose is to create the proxy and load

the contacts only when the Set button is clicked. Double-click the Set button to create a click event handler.

2. Remove the call to **ShowContacts()** from the **Form1\_Load()** method and move the code that instantiates the proxy to a new method called **SetupProxy()**, which takes a username and a password as parameters. Also, **proxy** should be initialized to **null**.

```
ContactManagerClient proxy = null;

private void Form1_Load(object sender, EventArgs e)
{
}

private void SetupProxy(string userName, string password)
{
    proxy = new ContactManagerClient();
    proxy.ClientCredentials.UserName.UserName = userName;
    proxy.ClientCredentials.UserName.Password = password;
}
```

3. On the **btnSetCredentials\_Click()** method, check if the proxy is created and, in case it isn't, call the **SetupProxy()** method passing the user-provided credentials, and then call the **ShowContacts()** method.

```
private void btnSetCredentials_Click(object sender, EventArgs e)
{
    if (proxy == null)
    {
        SetupProxy(txtUser.Text, txtPassword.Text);
    }

    ShowContacts();
}
```

4. Build and run host and client. Type "user" as username and password, and click the Set button. You should see the contacts listed appropriately, and a log for the call to **GetContacts()** in the service console window. Now, close the host application only, and try to click the Set button. You will hit an exception. Let's handle that exception in a generic way. Wrap the call to **ShowContacts()** in a try/catch block and display a message to the user. You should also set the proxy to null when the exception is caught, since you won't want to risk using the proxy in an invalid state later. And finally, show a message to the client after calling **ShowContacts()**, to make him aware that the call to the service was done successfully after setting the credentials.

```
try
{
    ShowContacts();
    MessageBox.Show("Authentication successfull");
}
catch (Exception ex)
{
    proxy = null;
    MessageBox.Show(
```



```

    "A problem occurred when communicating with the service: " +
    ex.Message);
}

```

5. Build and run client only. Enter valid credentials and click the Set button. You should see a proper message being displayed, indicating that the service is down. Now bring the service up and click the button again, to ensure the authentication happens successfully. Provide now different (invalid) credentials and click the Set button, and you will notice a weird behavior: the “authentication successful” message will come up! Actually, this is an interface issue, since the user that shows up on the service console log is “user”. That leads us to understand that our interface needs to “detect” when the credentials changed, so we can setup the proxy with the new credential information. You can do so by comparing the credentials on the text boxes with the ones stored on the proxy. Do this in the beginning of the **btnSetCredentials\_Click()** method.

```

private void btnSetCredentials_Click(object sender, EventArgs e)
{
    // If credentials changed, re-create proxy
    if (proxy != null &&
        (proxy.ClientCredentials.UserName.UserName != txtUser.Text ||
         proxy.ClientCredentials.UserName.Password != txtPassword.Text))
    {
        proxy.Close();
        proxy = null;
    }

    if (proxy == null)
    {
        SetupProxy(txtUser.Text, txtPassword.Text);
    }
    ...
}

```

6. Try to reproduce the issue again by setting the credentials as “user” and then “manager”, and you’ll see that the client is now able to detect that the credentials changed when you click the Set button. If you enter invalid credentials, you will hit an exception on the service, which you can ignore by pressing F5 and letting the service run, and you will then get a not so friendly message on the client about a security fault. You can improve the user experience by catching explicitly that exception type, which is **MessageSecurityException**, from the **System.ServiceModel.Security** namespace, and show a friendlier message to the user.

```

try
{
    ShowContacts();
    MessageBox.Show("Authentication successfull");
}
catch (System.ServiceModel.Security.MessageSecurityException)
{
    proxy = null;
    MessageBox.Show("Invalid username or password");
}

```

```

    }
    catch (Exception ex)
    {
        proxy = null;
        MessageBox.Show(
            "A problem occurred when communicating with the service: " +
            ex.Message);
    }
}

```

7. Build and run host and client, and verify that an appropriate message will be displayed when you enter invalid credentials. You will still hit the exception on the service, which is an expected behavior.
8. Now let's add a simple validation on the **btnAdd\_Click()** method to verify if the proxy was setup before the user tried to add a new contact. To achieve this, just verify if the proxy is null in the beginning of the method, and show a message to the user in case it needs to be setup. Do the same for the methods **btnRemove\_Click()** and **btnModify\_Click()**.

```

private void btnAdd_Click(object sender, EventArgs e)
{
    if (proxy == null)
    {
        MessageBox.Show("Please set your credentials first.");
        return;
    }
    ...
}

```

9. Build and run host and client and verify that you cannot use the Add, Remove and Modify buttons before properly setting up credentials.
10. Finally, let's implement an example of authorization on this application. Let's assume that only the user "manager" can delete contacts. To achieve that, you must add validation to the **RemoveContact** method of the **ContactManager** class in the service library.

```

LogCall("RemoveContact");

if (ServiceSecurityContext.Current.PrimaryIdentity.Name != "manager")
{
    throw new ApplicationException("Insufficient privileges");
}

contacts.Remove(cont);

```

11. Build and run host and client, and try to remove a contact using the "user" credential. You will hit an exception on the service, which you can ignore and press F5. Then, you will hit an unhandled exception on the client as well due to the exception raised on the service for insufficient privileges. Let's handle that exception so we can display a friendlier message to the user. Wrap the code from the **btnRemove\_Click()** method in the **Form1.cs** class on the client in a try/catch block, and display the exception message to the user in a message box.

```

try
{
    Contact cont = new Contact();
    cont.FirstName = txtFirst.Text;
    cont.LastName = txtLast.Text;
    proxy.RemoveContact(cont);
    ShowContacts();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}

```

12. Build and run host and client, and try to remove a contact using the “user” credential again. When hitting the exception on the service, press F5 and let it run. You will see a message box on the client side with a not so friendly message. This happened because the exception raised on the service side didn’t have its details propagated to the client. You have two options to get around this situation. The first would be to handle the specific exception type and show a hard coded message to the user, and the other would be to enable exception details to be sent from the service to the client. For our purposes in this lab, let’s do the latter. Open **App.config** in the host project and add the **serviceDebug** tag to the behavior configuration, configuring it to include exception details in faults.

```

<behavior name="serviceBehavior">
    ...
    <serviceDebug includeExceptionDetailInFaults="true"/>
</behavior>

```

13. Build and run host and client, and verify that now a more appropriate message is displayed in the previous use case. However, there is an issue: after the message is displayed, we can’t use the proxy anymore, since it is on the Faulted state. You will see that if you try to add or modify a contact, since you’ll hit an exception on the client. To solve this problem, just call the **SetupProxy()** method again on the catch block on the **btnRemove\_Click()** method, to make sure the proxy is valid even if a fault happens due to insufficient privileges.

```

try
{
    Contact cont = new Contact();
    cont.FirstName = txtFirst.Text;
    cont.LastName = txtLast.Text;
    proxy.RemoveContact(cont);
    ShowContacts();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
    SetupProxy(txtUser.Text, txtPassword.Text);
}

```

14. Build and run host and client, and verify that the application is working correctly after you try to remove a contact with insufficient privileges. Try to modify a contact and then change the credential to “manager”. You will notice an issue: after you change the credentials, the data in the contact you modified was reset to the original value. This happened because the service is running using the default instance context mode, which is per session. Let’s change that so that the service runs in a single instance. Open the **ContactManager.cs** file on the service library and change the **InstanceContextMode** property to Single.

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
public class ContactManager : IContactManager
{
    ...
}
```

15. Build and run host and client, and verify that the application is working correctly. You are now at Step 3.