

# Introduction to Windows Communication Foundation



Ken Casada  
Microsoft Switzerland  
[ken.casada@microsoft.com](mailto:ken.casada@microsoft.com)

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2007 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, MS, Windows, Windows NT, MSDN, Active Directory, BizTalk, SQL Server, SharePoint, Outlook, PowerPoint, FrontPage, Visual Basic, Visual C++, Visual J++, Visual InterDev, Visual SourceSafe, Visual C#, Visual J#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

## Contents

INTRODUCTION.....	4
BEFORE GETTING STARTED .....	5
CONVENTION USED FOR THE HOL.....	5
The ABC of Windows Communication Foundation.....	6
Defining Contracts and Implementing the Service .....	7
Hosting the WCF Service and Endpoints .....	12
Building the WCF Client .....	21
Using Multiple Endpoints .....	26
Protecting your Service .....	30
IIS Version 7: support for non-HTTP protocols (Optional part) .....	39

## INTRODUCTION

The goal of this lab is to provide you with the basics understanding of Windows Communication Foundation (WCF). WCF has been released for the first time in November 2006 as part of the .NET Framework 3.0 and recently updated with the .NET Framework 3.5 release. Before you start with the HOL it's probably a good idea to say what WCF is and why Microsoft built it.

WCF enables you to build powerful service-oriented systems, based on connected services and applications. The WCF runtime and its *System.ServiceModel* namespace that represents its primary programming interface, is nothing less than the successor and unification of most distributed systems technologies that developers have successfully used to build distributed applications on the Windows platform over the past decade. When designing distributed applications in the past, you frequently had to choose a specific technology, such as ASP.NET Web Services (ASMX) along with the Web Service Enhancements (WSE) extensions, the Microsoft Message Queue (MSMQ), the Enterprise Services/COM+ runtime environment, and .NET Remoting.

Now it comes automatically the question if there was something wrong with all these technologies. Actually nothing was wrong, except that there were too many of them and that the choice between one of them had a fundamental impact on the architecture of your solutions.

The goal of WCF is also that you no longer have to make such technology choices upfront and that you can implement any combination of requirements on a single technology platform and without friction.

The promise is that WCF will make developers more productive, because they will only have to master a single programming model that unifies the feature wealth of ASMX, WSE, Enterprise Services, MSMQ, and Remoting.

## BEFORE GETTING STARTED

To complete the hands-on-lab you need:

- Visual Studio 2008 (not the express edition)  
(evaluation version available here: <http://msdn2.microsoft.com/en-us/vstudio/products/aa700831.aspx>)
- Microsoft .NET Framework 3.5 (included in Visual Studio 2008)  
(downloadable here: <http://go.microsoft.com/?linkid=7755937>)
- Windows Server 2003 SP1, Windows Server 2008, Windows XP SP2, or Windows Vista (better with SP1)



To complete the optional part of the lab you need:

- **Windows Vista (better with SP1) or Windows Server 2008**  
→ IIS version 7 is required!
- **Following Windows components must be installed:**
  - IIS Metabase and IIS6 Configuration Compatibility
  - ASP.NET
  - IIS Management Console
  - Windows Communication Foundation HTTP Activation
  - Windows Communication Foundation Non-HTTP Activation

## CONVENTION USED FOR THE HOL



notes, tips



code snipped available

## The ABC of Windows Communication Foundation

What are the ABCs of WCF? “ABC” stands for address, binding and contract.

When you define a service you have to specify some important information such as **what** this service can do (Contract), **how** can it be accessed (Binding) and **where** it is available (Address). These three pieces of information are encapsulated in endpoints. So an endpoint is the “gateway” to the outside world, containing information that clients need to know in order to use a service.

What makes WCF really interesting is that you can have one service with multiple endpoints and each of these endpoints can differ in the address, in the binding requirements, or contract getting implemented. This means that you can have one service, which is flexible and interoperable for any application requirements.

"ABC" means also another thing: writing (and configuring) a WCF service is always a three-step process:

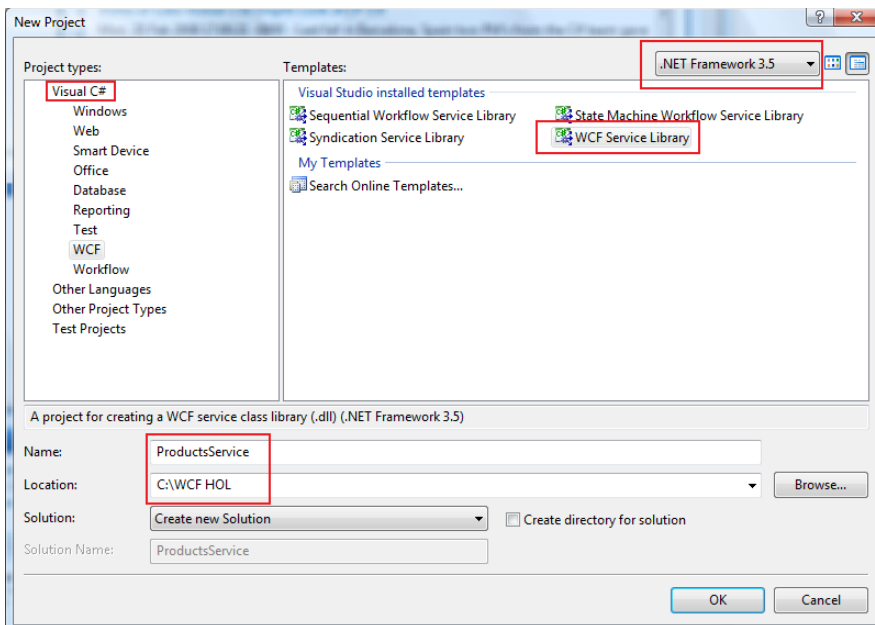
- You define a contract and you implement it on a service.
- You choose or define a service binding that selects a transport (HTTP, tcp, ...) along with quality of service, security and other options.
- You deploy an endpoint (ABC) for the contract by binding it (using the binding definition) to a network address.

Now that you have some basics information it's time to start creating your first WCF service by first defining the contract.

## Defining Contracts and Implementing the Service

The structure of a WCF service enables you to adopt a “contract-first” approach to development. You therefore have to define an interface (or contract) that your service will implement and then build a service that conforms to this contract. With this approach you can focus on the design of your service.

- 1.1 Create the **C:\WCF HOL** folder.
- 1.2 Click the **Start|Programs|Microsoft Visual Studio 2008|Microsoft Visual Studio 2008** menu command. *If you are using UAC (User Access Control), then start Visual Studio 2008 as an administrator.*
- 1.3 On the **File** menu, point to **New**, and click **Project....**
- 1.4 Select the **WCF Service Library** template from the **Visual C#/WCF** section. As during the lab we will use LINQ to XML to load data, be sure you have selected the .NET Framework 3.5 as Target Framework.
- 1.5 Uncheck the **Create directory for solution** check box.
- 1.6 Provide a name for the new project by entering *ProductsService* in the **Name** box and provide a **location** by entering *C:\WCF HOL* as shown in the following picture:



- 1.7 Click **OK**.

As you can see Visual Studio 2008 generates a WCF Library Project. Inside this project you find two files: the *IService1.cs* that you will use to define your own interface (the contract) and the *Service1.cs* that you will use to define the implementation of your service; an *App.config* File is also present.

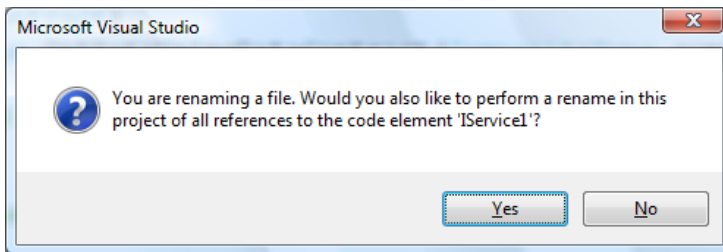


If you have a look inside one of these two files you will notice that WCF makes use of the classes defined in the *System.Runtime.Serialization* namespace (*using ...*) to convert objects into a stream of data for transmitting over the network (a process known as *serialization*) and to convert a stream of data received from the network back into objects (*deserialization*).

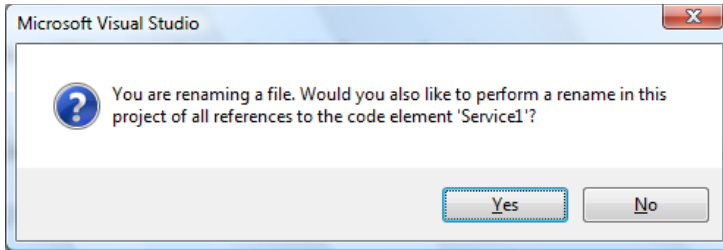
- 1.8 As the goal of the HOL is to learn the WCF technology with a step by step approach, we are not going to use the whole code and files provided by the Visual Studio 2008 template.

As first delete the **App.Config** file from the ProductsService project.

- 1.9 Rename the *IService1.cs* file in **IProductsService.cs**. If the following dialog appears press the **yes** button.



- 1.10 Rename the *Service1.cs* file in **ProductsService.cs**. If the following dialog appears press the **yes** button.



- 1.11 Clean the code inside the *IProductsService.cs* by removing the code defined for the *IProductsService* interface and deleting the *CompositeType* class that is part of the VS template. Inside the *IProductsService.cs* file you should now have the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;

namespace ProductsService
{
    [ServiceContract]
    public interface IProductsService
    {
    }
}
```

Note that the interface is decorated with the **ServiceContract** attribute. As the name says, this attribute indicates that an interface (or a class) defines a service contract in a Windows Communication Foundation (WCF) application.



By default, the namespace and name of a WCF contract are the namespace and name of the interface to which the *ServiceContract* attribute is added (in our specific case *ProductsService.IProductsService*). One can alter the default name of a contract using the *Namespace* and *Name* parameters of the *ServiceContract* attribute as

```
[ServiceContract(Namespace="MyNamespace", Name="MyContract")]
public interface IProductsService
```



- 1.12 Instead of using the `CompositeType` class we are going to define our own class. Add the **Product** class shown below to the *ProductsService* namespace inside the *IProductsService* file.



```
[DataContract]
public class Product
{
    [DataMember]
    public int ProductID { get; set; }

    [DataMember]
    public string ProductName { get; set; }

    [DataMember]
    public double UnitPrice { get; set; }

    [DataMember]
    public int StockLevel { get; set; }

    [DataMember]
    public string Category { get; set; }
}
```

The **DataContract** attribute identifies the class as a type that can be serialized and deserialized as an XML stream by WCF. The **DataMember** attribute specifies that the member is part of a data contract and is serializable (any member not tagged in this way will not be serialized).

- 1.13 Inside the *IProductsService* interface define all methods you want to expose:



```
[ServiceContract]
public interface IProductsService
{
    [OperationContract]
    List<Product> GetProductsList();

    [OperationContract]
    List<Product> GetProductsByCategory(string CategoryName);

    [OperationContract]
    Product GetProduct(int ProductID);

    [OperationContract]
    int GetStockLevel(int ProductID);
}
```

Note that the **OperationContract** attribute indicates that a method defines an operation that is part of the service contract in a WCF application.

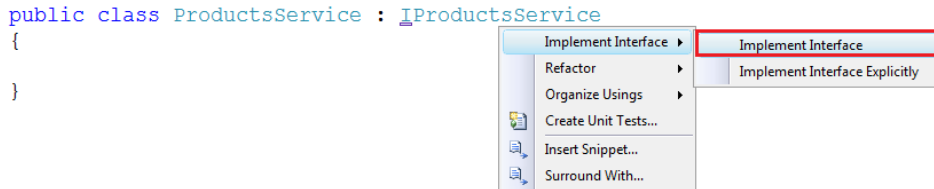


By default, the name on an operation included in a WCF contract is the name of the method to which the *OperationContract* attribute is added. One can alter the default name of an operation with the *Name* parameter of the *OperationContract* attribute as

```
[OperationContract (Name="MyOperation")]
List<Product> GetProductsList();
```

1.14 Now that you have defined the shape of the WCF service (using the *ServiceContract* and *OperationContract*) and have specified the structure of the data passed to the WCF service (using the *DataContract* and *DataMember*) it's time to implement the *ProductsService*.

Open the *ProductsService.cs* file and remove all code inside the *ProductsService* class, then right-click on *IProductsService* and select *Implement Interface* as shown in the following figure:



1.15 You now have to implement the four methods you defined previously within the interface as shown in the following code and then **add** `using System.Xml.Linq;` on top of the *ProductsService.cs* file. Note that to keep things simple we will get data from an XML file and to do that we will use LINQ to XML. Note also that we will put the source *Products.xml* file in the bin directory later for execution:



```
public List<Product> GetProductsList()
{
    var productList = from p in XElement.Load("Products.xml").Elements()
                      select new Product
                      {
                          ProductID = int.Parse(p.Attribute("ProductID").Value),
                          ProductName = p.Element("ProductName").Value,
                          UnitPrice = double.Parse(p.Element("UnitPrice").Value),
                          StockLevel = int.Parse(p.Element("StockLevel").Value),
                          Category = p.Element("Category").Value
                      };

    return productList.ToList();
}

public List<Product> GetProductsByCategory(string CategoryName)
{
    var productList = from p in XElement.Load("Products.xml").Elements()
                      where p.Element("Category").Value == CategoryName
                      select new Product
                      {
                          ProductID = int.Parse(p.Attribute("ProductID").Value),
                          ProductName = p.Element("ProductName").Value,
                          UnitPrice = double.Parse(p.Element("UnitPrice").Value),
                          StockLevel = int.Parse(p.Element("StockLevel").Value),
                          Category = p.Element("Category").Value
                      };

    return productList.ToList();
}

public Product GetProduct(int ProductID)
{
    Product prod = (
        from p in XElement.Load("Products.xml").Elements()
        where p.Attribute("ProductID").Value == ProductID.ToString()
        select new Product
        {
            ProductID = int.Parse(p.Attribute("ProductID").Value),
            ProductName = p.Element("ProductName").Value,
```

```

        UnitPrice = double.Parse(p.Element("UnitPrice").Value),
        StockLevel = int.Parse(p.Element("StockLevel").Value),
        Category = p.Element("Category").Value
    }
    ).FirstOrDefault();

    return prod;
}

public int GetStockLevel(int ProductID)
{
    int stockLevel =
        (
            from p in XElement.Load("Products.xml").Elements()
            where p.Attribute("ProductID").Value == ProductID.ToString()
            select int.Parse(p.Element("StockLevel").Value)
        ).FirstOrDefault();

    return stockLevel;
}

```

- 1.16 Choose **Build/Build Solution** from the Visual Studio 2008 menu to ensure that there are no programming errors.

## Hosting the WCF Service and Endpoints

In order to run and make a WCF service accessible to clients you have to host it. Before we see what are the available options for hosting a WCF service we first have to define what are the tasks that a hosting application has to perform: first a host application has to start and stop the service, must listen for request coming from client application and direct them to the service and finally send any responses from the service back to the client application.

But before you start to implement the hosting application, it is important to clarify a little bit better the concept of endpoint. In fact endpoints are a key factor, because the way a host application makes a service available to clients is exactly by providing endpoints to which clients can send requests.

As you already know an endpoint contains 3 pieces of information: the address, the binding and the contract. So far you have already seen what the contract is and how you implement it. Let's also have a look at the address and at the binding:

- Address: specifies the address of the service. The address depends on several factors, including the transport protocol being used. The following list show you the root of an address (base address scheme) depending on the chosen transport protocol:

<u>Transport Protocol</u>	<u>Base Address Scheme</u>
HTTP	http
TCP	net.tcp
Named Pipes	net.pipe
MSMQ	net.msmq

- Binding: describes how a client can connect to the service and the format of the data expected by the service and responds to the following questions: what transport protocol do the client needs to use? What's the encoding format of messages, XML or binary? What are the security and the transactional requirements of the service?

A WCF binding is also a combination of binding element and one special category of binding element consists of those that implement protocols for transporting messages. One of those is the binding element that implements the hypertext transport protocol (HTTP). Another is the binding element that implements the transmission control protocol (TCP). Another special category of binding element consists of those that implement protocols for encoding messages. One of those is a binding element for encoding XML as text. Another is a binding element for encoding XML in a binary format.

A WCF binding is a collection of bindings elements that must include at least one transport protocol binding element, one or more message-encoding protocol binding element, and zero or more other binding elements.

Bindings may be defined by selecting individual binding elements, either in code, or in configuration. However, WCF provides a number of classes that represent common selections of bindings elements. Those classes are referred to as the standard binding. One of the standard bindings is the BasicHttpBinding. The BasicHttpBinding represents the combination of HTTP transport binding element and the binding element for encoding XML in text format.

The following table lists the available standard bindings within the WCF Library version 3.0:

BasicHttpBinding	This binding conforms to the WS-I Basic Profile 1.1. It can use the HTTP and HTTPS protocols and encodes messages as XML text. This binding is used to maintain compatibility with client applications previously developed to access .asmx Web services.
WSHttpBinding	HTTP communication in conformity with WS-* protocols. It supports the HTTP and HTTPS transport protocols.
WSDualHttpBinding	Similar to WSHttpBinding, but it is suitable for handling duplex communications. Duplex messaging enables a client and service to perform two-way communication without requiring any form of synchronization (the receiver of a message may transmit any number of responses over a period of time).
WSFederationBinding	This binding supports the WS-Federation specifications. WS-Federation defines several models for providing federated security, based on the W-WS-Trust, WS-Security, and WS-SecureConversation specifications. It is based on HTTP communication, in which access to the resources of a service can be controlled based on credentials issued by an explicitly-identified credential provider.
NetTcpBinding	Secure, reliable, high-performance communication between Windows Communication Foundation software entities across a network. Binary encoding is used. Supports transactions, reliable session and secure communication. It is typically used in an intranet scenario between computers using the Windows OS.
NetPeerTcpBinding	Communication between Windows Communication Foundation software entities via Windows Peer-to-Peer Networking. Supports secure communication and reliable ordered delivery of messages. Binary encoding is used.
NetNamedPipeBinding	Secure, reliable, high-performance communication between Windows Communication Foundation software entities on the same machine.
NetMsmqBinding	Communication between Windows Communication Foundation software entities via Microsoft Message Queuing (MSMQ). MSMQ is used as the transport to transmit messages between client application and service (both implemented using WCF). This binding enables temporal isolation (messages are stored in a message queue, so the client and the service do not both have to be running at the same time. Binary encoding is used; secure, reliable sessions and transactions are also supported.
MsmqIntegrationBinding	For building WCF applications that sends or receives messages from another software entity via MSMQ. For communication with existing applications that use MSMQ message queues.

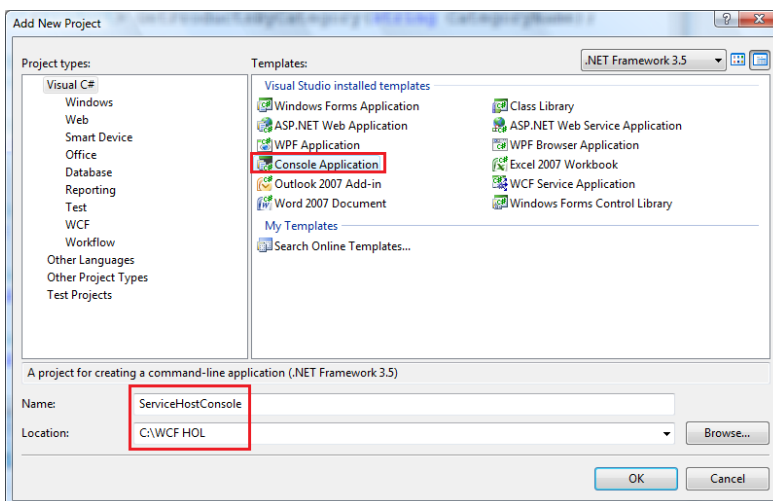
Now that you have a better idea of what an endpoint is, it's time to implement the hosting application that will expose our endpoints. But what are the available options to host a WCF service?

1. You can create your own custom hosting application. Any .NET 3.0 Application can be programmed to be a hosting application.
2. You can host it in a Windows service so that the WCF service is available as long as Windows is running.
3. You can host it in Internet Information Server (IIS) and so expose HTTP Endpoints.
4. If you are using IIS version 7.0 (part of Windows Vista and Windows Server 2008) you can use Windows Activation Services, or WAS. Using WAS, you can configure and host a WCF service without needing to run the World Wide Web Publishing Service and more interesting you can also host non-HTTP WCF services (option that is not available with previous IIS version). We will use WAS in the last optional part of the lab.

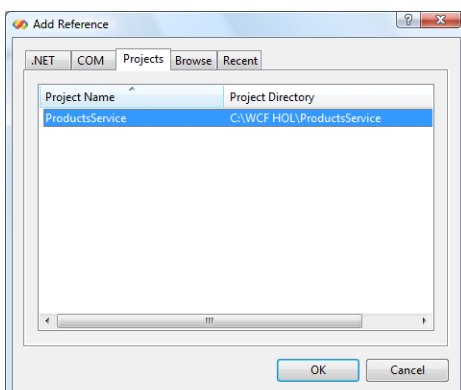
In our specific case we will start by implementing the first option, a custom host application (in our case a ConsoleApplication) that the user will have to run in order to start and stop the service.

1.17 In the *Solution Explorer* right-click on the solution and select **Add|New Project....**

1.18 Provide a name for the new project by entering *ServiceHostConsole* in the **Name** box, provide a **location** by entering *C:\WCF HOL* as shown in the following picture and click **OK**.



1.19 You now have to add a reference to the new created project. To do this select **Project|Add Reference...** from the Visual Studio 2008 menus, and, from the **Projects** tab, add a reference to the **ProductsService** project.



- 1.20 Select **Project|Add Reference...** from the Visual Studio 2008 menus, and, from the **.NET** tab, add a reference to the **System.ServiceModel** assembly.
- 1.21 Modify the contents of the **Programs.cs** file module as shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace ServiceHostConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            Type serviceType = typeof(ProductsService.ProductsService);

            using (ServiceHost host = new ServiceHost(serviceType))
            {
                host.Open();

                Console.WriteLine("The Product Service is available");

                Console.ReadLine();

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

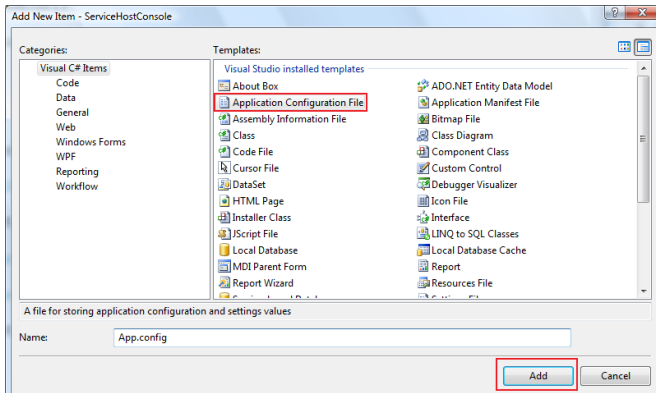
*ServiceHost* is the class provided by the Windows Communication Foundation Service Model for programming .NET applications to host Windows Communication Foundation endpoints within application domain.

In the code above a constructor of the *ServiceHost* class is given information to identify the service type of the service that is to be hosted. What the *ServiceHost* object is doing, is instantiating a service object from an assembly holding the service class (*ProductsService.ProductsService*), configure the endpoints of the service by using bindings provided in an application configuration file (next step) or in code, apply any security settings required by the service, and create listener objects for each address that you specify. In our specific case we didn't specify any address (no second parameter by the *ServiceHost* constructor) because we want to define everything (address and binding) as part of one or more endpoints inside an application configuration file. This is exactly what we are going to do in the following steps.

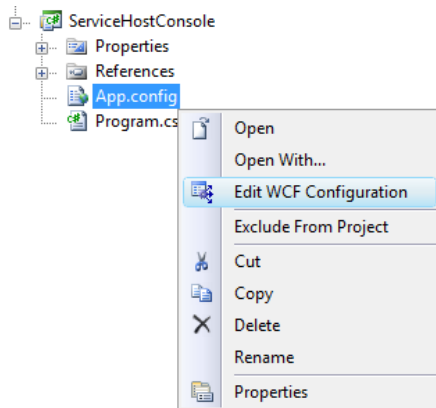


Specifying an address and binding for an endpoint does not require writing any code, and is customarily the work of an administrator rather than a programmer. Providing an address and a binding can be done in code. However, that would require having to modify the code in order to change the address and the binding of the endpoint. A key innovation of the Windows Communication Foundation is to separate how software is programmed from how it communicates, which is what the bindings specify. So, generally, one avoids the option of specifying the addresses and bindings of endpoints in code.

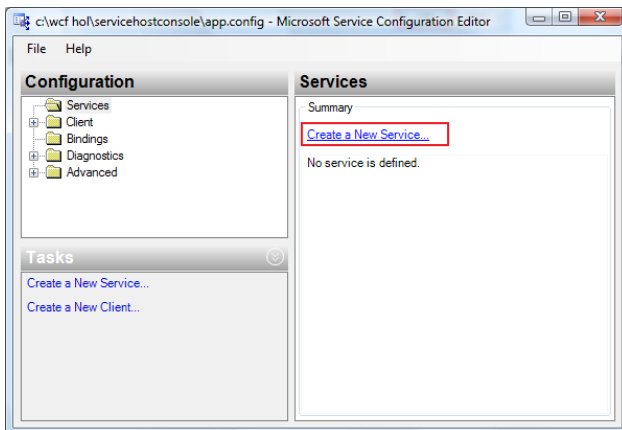
- 1.22 Right-click the *ServiceHostConsole* project, select **Add | New Item...** and add an *Application Configuration File* as shown in the following figure:



- 1.23 Right-Click the added *App.Config* file and select **Edit WCF Configuration** as shown in the following figure. This will start the *Microsoft WCF Configuration Editor* tool, which has been completely integrated within Visual Studio 2008.

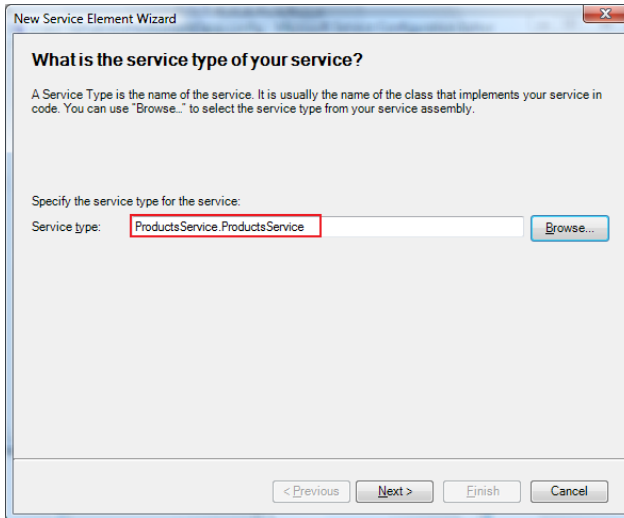


- 1.24 Click on the *Create a New Service ...* link as shown in the following figure. The *New Service Elements Wizard* starts.

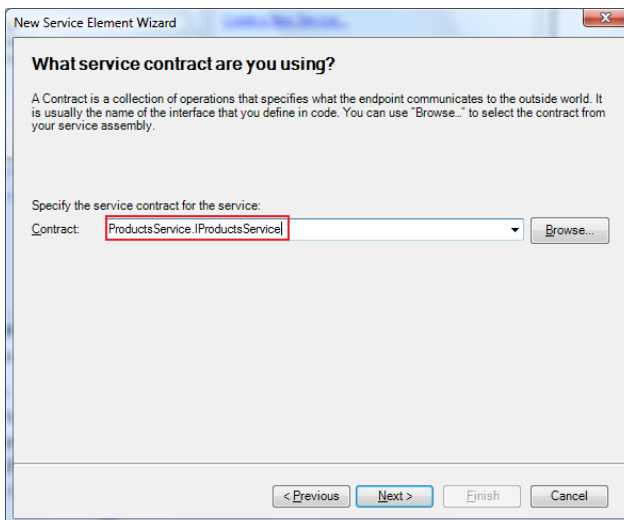




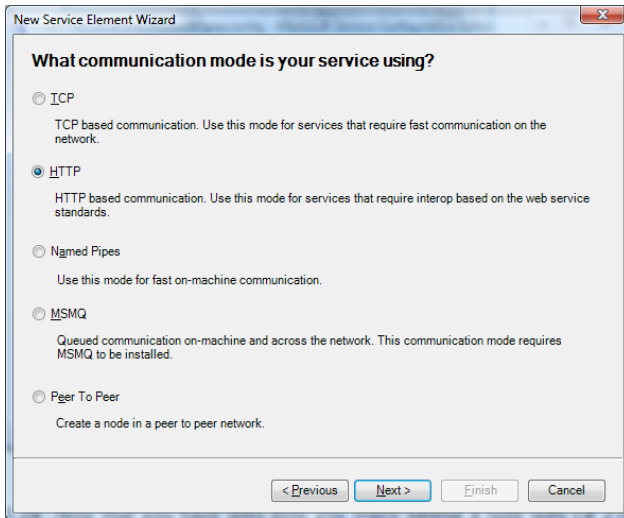
- 1.25 As *Service type* put **ProductsService.ProductsService**, then press on the **Next** button. If you prefer you can also click on *Browse*, double-click on the *ProductsService.dll* you find in the *c:\wcf\hol\servicehostconsole\bin\Debug\* folder and then select the *ProductsService.ProductsService* Type Name. Then press **Next**.



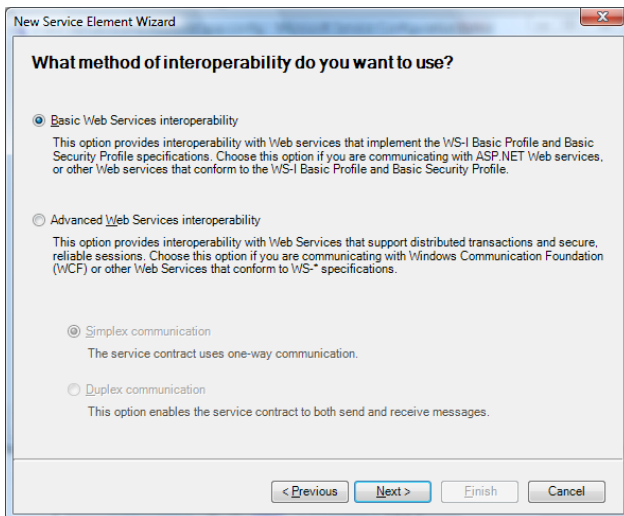
- 1.26 Define the contract by putting **ProductsService.IProductsService** in the contract field and then press the **Next** button (also in this case it is possible to use the browse button and select the *ProductsService.dll* which contains the service type definition as well as the contract).



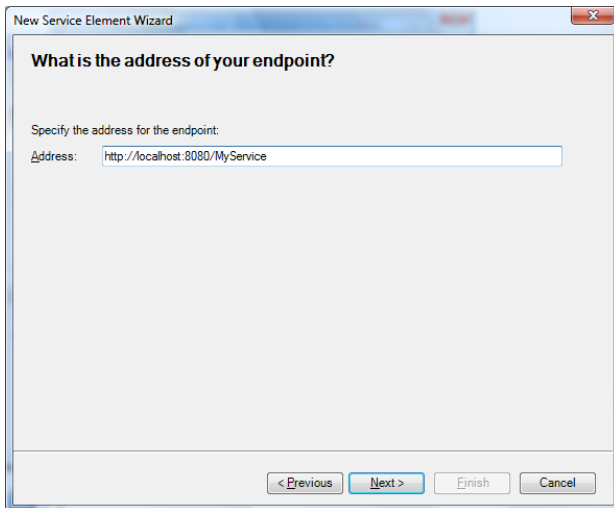
- 1.27 You know have to choose the communication mode (in other words the transport protocol). In this case we will start by using HTTP. Choose **HTTP** and then press **Next**.



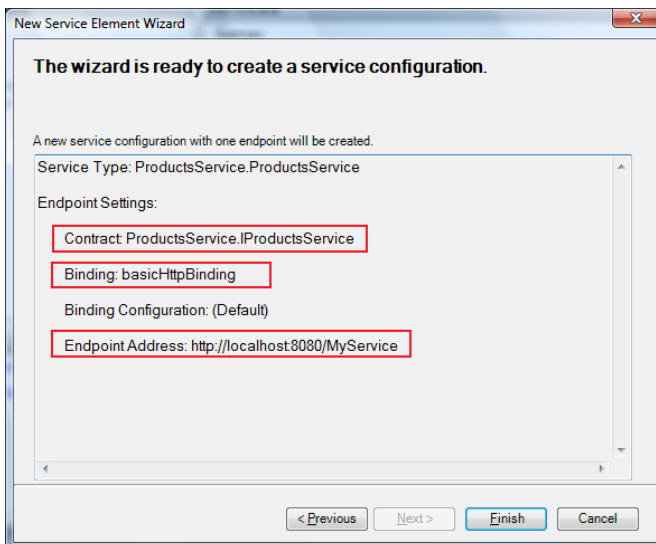
- 1.28 As we want to use the *BasicHttpBinding* select **Basic Web Services interoperability** as shown in the following figure and then press **Next**.



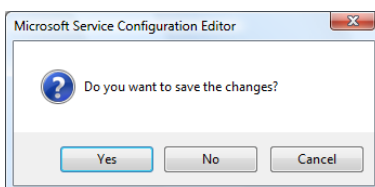
- 1.29 As address put <http://localhost:8080/MyService>. The address is the location that the application hosting the service uses to advertise the service. The address that you specify following the scheme, machine, and port (in this case *MyService*) is just a logical identifier that the WCF service uses to advertise the service to clients.



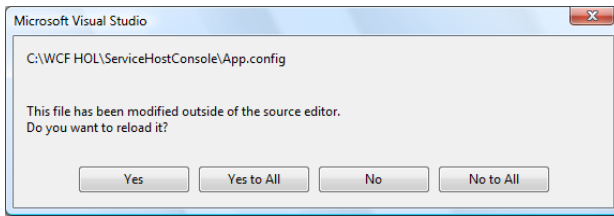
- 1.30 Click on **Next** and then on **Finish**. As you can see from the following dialog you have defined your first EndPoint with <http://localhost:8080/MyService> as Address, *BasicHttpBinding* as Binding and *ProductsService.IProductService* as Contract.



- 1.31 Close the *Microsoft Service Configuration Editor* tool and when the following SaveAs dialog appears press **yes**.



1.32 If the following dialog appears press **Yes**.

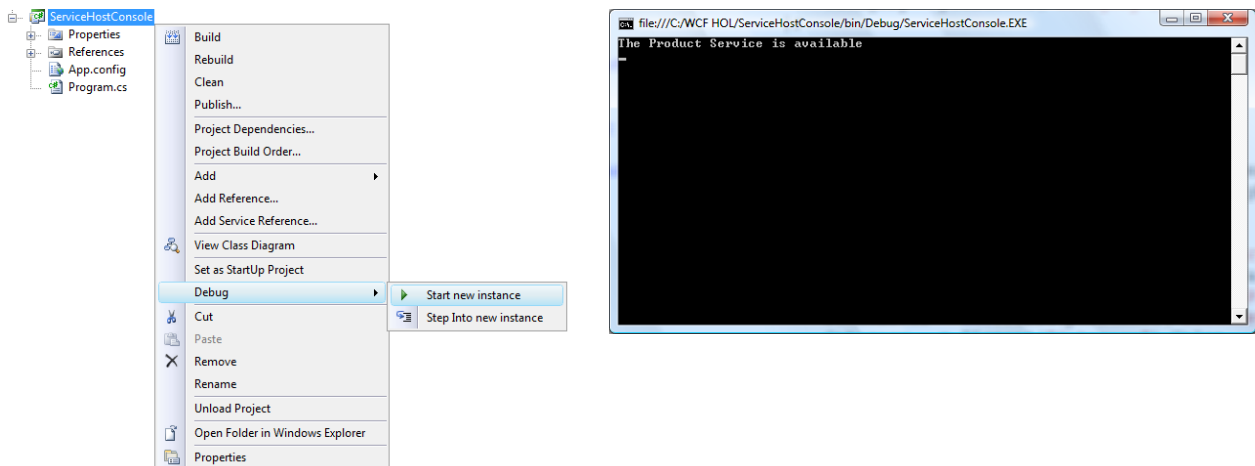


1.33 If you now have a look at the changed App.config file you see that a `<system.serviceModel>` tag with Endpoint information has been added to it.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="ProductsService.ProductsService">
        <endpoint
          address="http://localhost:8080/MyService"
          binding="basicHttpBinding"
          bindingConfiguration=""
          contract="ProductsService.IProductsService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

1.34 Copy the **Products.xml** file provided in the `\HOL_Material\Data\` folder in the `C:\WCF HOL\ServiceHostConsole\bin\Debug` folder (the bin directory of the hosting application).

1.35 Start a new instance of the *ServiceHostConsole* application. The hosting application should start and you should obtain a console dialog telling you that the product service is available. What's happening? The hosting application is listening on <http://localhost:8080/MyService> for request coming from client. In the next step we will implement our client application. Press ENTER to close both console applications.



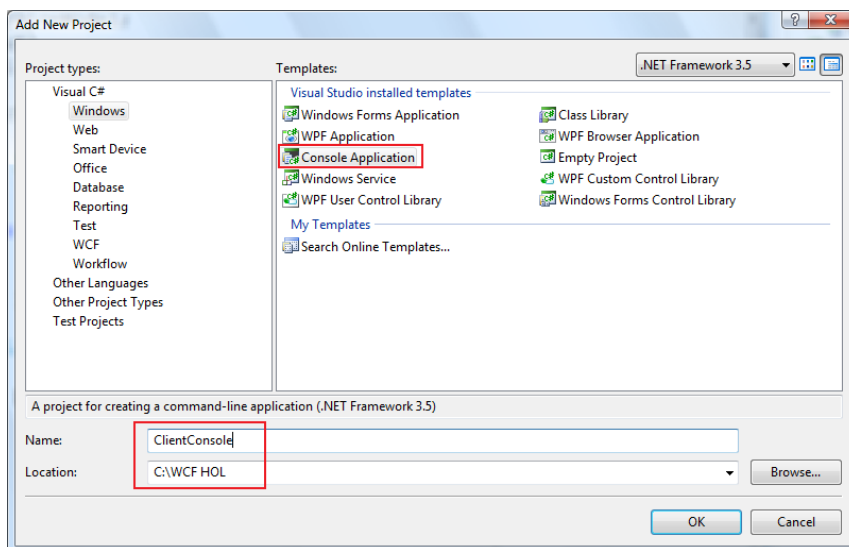
## Building the WCF Client

As client we will also use a Console Application and we will proceed in the following way:

- First we are going to create the new console application project
- Second, in order to be able to communicate with the WCF service we need to create a client proxy class
- Third, implement the client console application making use of the generated proxy class.
- Last, define a client endpoint that reflect the one on the server side

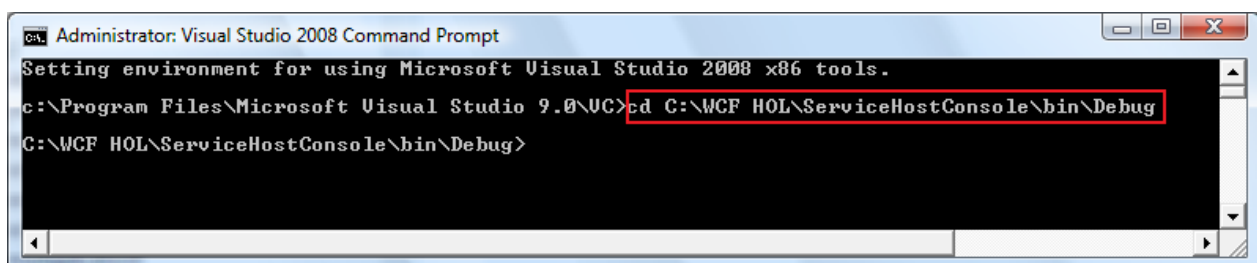
1.36 To start, add a new Console Application Project to the solution file. To do this in the *Solution Explorer* right-click on the solution and select **Add|New Project....**

1.37 Provide a name for the new project by entering *ClientConsole* in the **Name** box, provide a **location** by entering *C:\WCF HOL* as shown in the following picture and click **OK**.

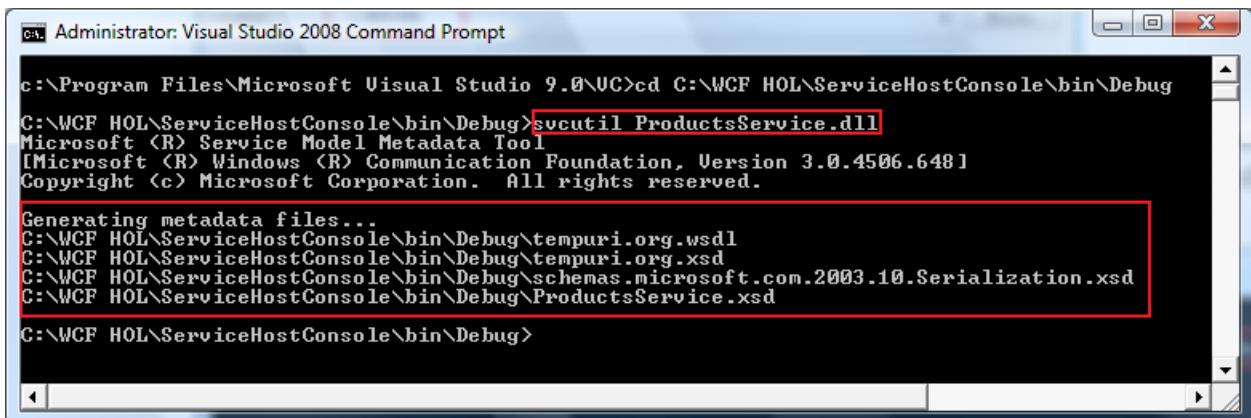


1.38 We need to add a reference to the *System.ServiceModel* and *System.Runtime.Serialization* assemblies in order to communicate with the WCF service. Select **Project|Add Reference...** from the Visual Studio 2008 menus, and, from the **.NET** tab, add a reference to the two mentioned assemblies.

1.39 We now have to create the proxy class. To do that open the *Visual Studio 2008 Command Prompt* and move to the *C:\WCF HOL\ServiceHostConsole\bin\Debug* folder where the *ProductsService.dll* is located (to do this run the *cd* command as shown in the following figure).



- 1.40 Run the `svcutil ProductsService.dll` command as shown in the following figure. This command extracts the definition of the `ProductsService` and the other types from the assembly. Note that a `wsdl` file (a file which contains the contract of our WCF service) has also been generated.



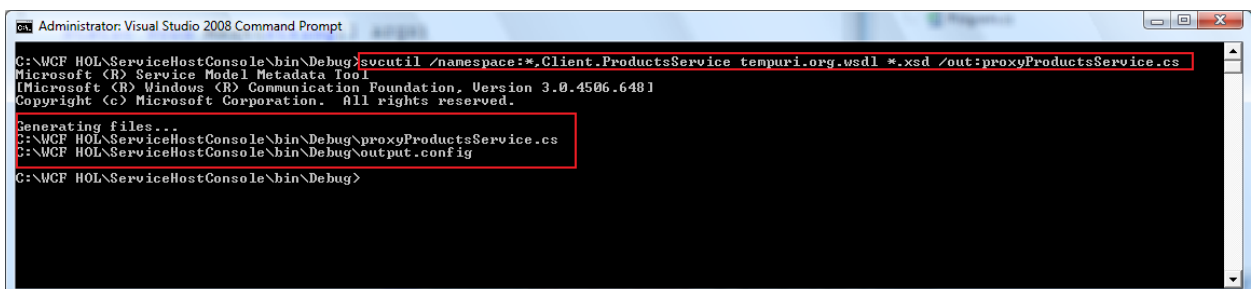
```
Administrator: Visual Studio 2008 Command Prompt
c:\Program Files\Microsoft Visual Studio 9.0\VC>cd C:\WCF HOL\ServiceHostConsole\bin\Debug
C:\WCF HOL\ServiceHostConsole\bin\Debug>svcutil ProductsService.dll
Microsoft (R) Service Model Metadata Tool
[Microsoft (R) Windows (R) Communication Foundation, Version 3.0.4506.6481]
Copyright (c) Microsoft Corporation. All rights reserved.

Generating metadata files...
C:\WCF HOL\ServiceHostConsole\bin\Debug>tempuri.org.wsdl
C:\WCF HOL\ServiceHostConsole\bin\Debug>tempuri.org.xsd
C:\WCF HOL\ServiceHostConsole\bin\Debug>schemas.microsoft.com.2003.10.Serialization.xsd
C:\WCF HOL\ServiceHostConsole\bin\Debug>ProductsService.xsd
C:\WCF HOL\ServiceHostConsole\bin\Debug>
```

- 1.41 You can now use the `WSDL` file and the `XML schemas (*.xsd)` to generate the proxy class. With the following commands we want to generate a `C#` source file (`proxyProductsService.cs`) containing a class that can be used as a proxy for the `WCF` service. With the `/namespace` option we tell the utility to generate a `Client.ProductsService` namespace. Run



`svcutil /namespace:*,Client.ProductsService tempuri.org.wsdl *.xsd /out:proxyProductsService.cs`

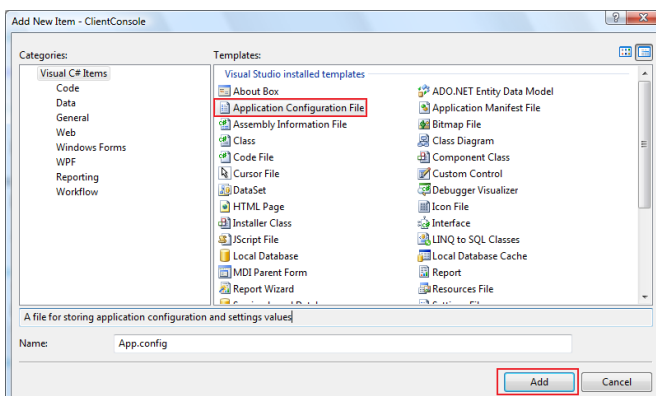


```
Administrator: Visual Studio 2008 Command Prompt
C:\WCF HOL\ServiceHostConsole\bin\Debug>svcutil /namespace:*,Client.ProductsService tempuri.org.wsdl *.xsd /out:proxyProductsService.cs
Microsoft (R) Service Model Metadata Tool
[Microsoft (R) Windows (R) Communication Foundation, Version 3.0.4506.6481]
Copyright (c) Microsoft Corporation. All rights reserved.

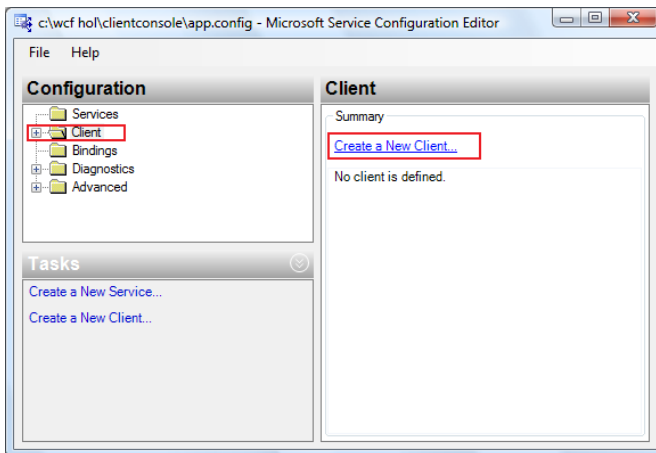
Generating files...
C:\WCF HOL\ServiceHostConsole\bin\Debug>proxyProductsService.cs
C:\WCF HOL\ServiceHostConsole\bin\Debug>output.config
C:\WCF HOL\ServiceHostConsole\bin\Debug>
```

and the `proxyProductsService.cs` and the `output.config` file will be generated! You are now ready to integrate these two files inside the client application.

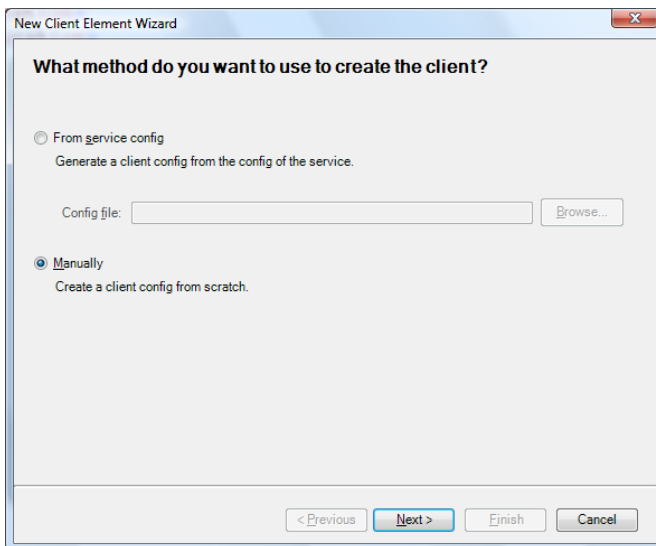
- 1.42 Right-click the `ClientConsole` project and select **Add|Existing Item....** And select the `C:\WCF HOL\ServiceHostConsole\bin\Debug\proxyProductsService.cs` and press **Add**.
- 1.43 Instead of using the generated `output.config` file we want to generate our own. Right-click the `ClientConsole` project, select **Add|New Item...** and add an `Application Configuration File` as shown in the following figure:



- 1.44 Right-Click the added *App.Config* file and select **Edit WCF Configuration**. The *Microsoft Service Configuration Editor* tool will start.
- 1.45 Select the *Client* section and then click on the *Create a New Client ...* link as shown in the following figure. The *New Client Elements Wizard* starts.

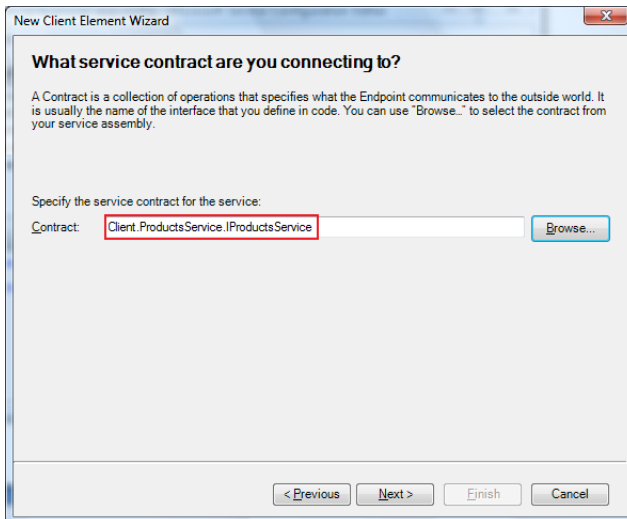


- 1.46 The wizard offers you two options: you can create a config file taking as input the one you have defined on the server or you can create one from scratch. Choose the second one (*Manually*).



1.47 As Service type put **ProductsService.ProductsService**, then press on the **Next** button.

1.48 As contract type **Client.ProductsService.IProductsService** or if you prefer you can also click on *Browse*, double-click on the *ClientConsole.exe* file you find in the *C:\WCF HOL\ClientConsole\bin\Debug* folder and then select the *Client.ProductsService.IProductsService* Type Name (the information about the interface is stored in the client EXE file). Press **Next**.



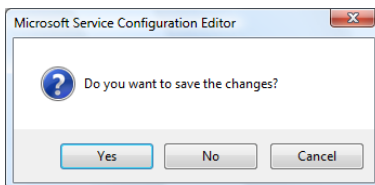
1.49 As communication mode select once again **HTTP**.

1.50 As we want to use the *BasicHttpBinding* select gain **Basic Web Services interoperability** and press **Next**.

1.51 As address put <http://localhost:8080/MyService> (the same we defined for the server side endpoint) and press **Next**.

1.52 As name for the client configuration leave the default value provided and press **Next**. Finally press **Finish**.

1.53 Close the *Microsoft Service Configuration Editor* tool and when the following dialog appears press **yes**.



1.54 If you now have a look at the changed App.config file you see that a `<system.serviceModel>` tag with Endpoint information has been added to it.

```
<system.serviceModel>
  <client>
    <endpoint
      address="http://localhost:8080/MyService" binding="basicHttpBinding"
      bindingConfiguration=""
      contract="Client.ProductsService.IProductsService"
      name="" />
    </client>
  </system.serviceModel>
```



- 1.55 Open the *Program.cs* file of the *ClientConsole* application and add the following code inside the *Main* method:



```
static void Main(string[] args)
{
    Console.WriteLine("When the hosting application is ready press ENTER");
    Console.ReadLine();

    int prodID = 1;

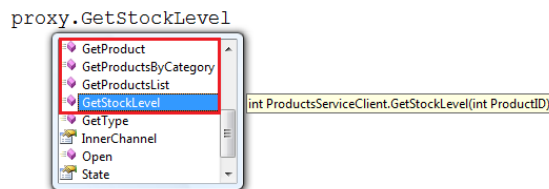
    Client.ProductsService.ProductsServiceClient
        proxy = new Client.ProductsService.ProductsServiceClient();

    int stockLevel = proxy.GetStockLevel(prodID);

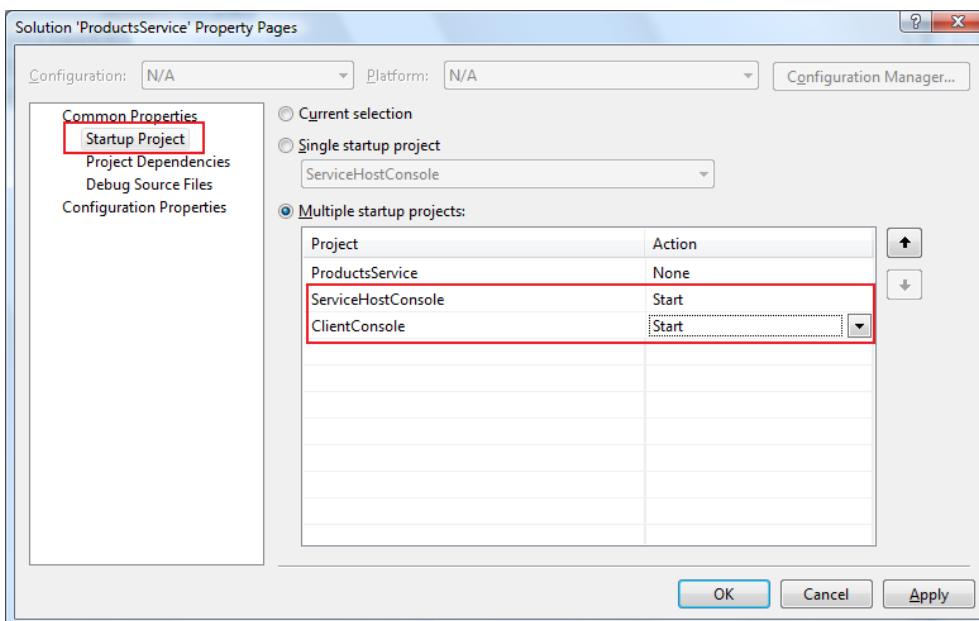
    Console.WriteLine("Stock Level for productID {0} is {1}",
        prodID.ToString(),
        stockLevel.ToString());

    Console.ReadLine();
}
```

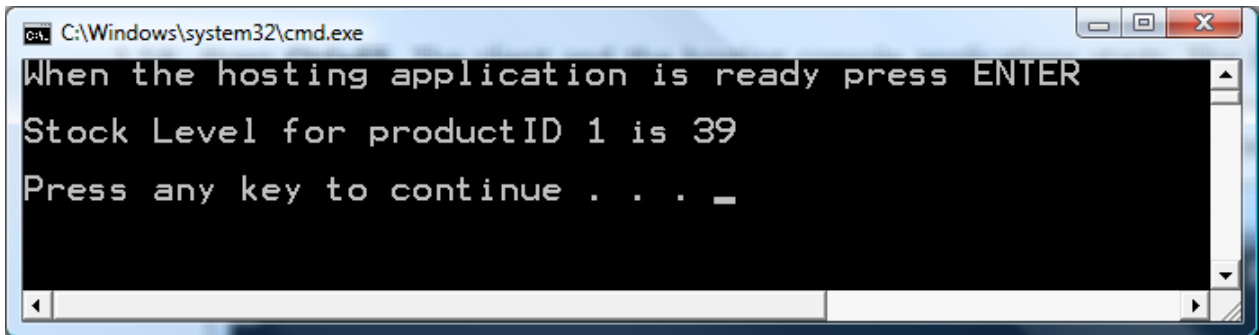
As you can see you are using the `Client.ProductsService.ProductsServiceClient` proxy client that has been generated inside the *proxyProductsService.cs* file. You can now access and use all methods exposed by the WCF service as shown from the following figure:



- 1.56 We are now ready to do our first test. As we need to start multiple project inside the same solution, right-click the *ProductsService* solution file and select *Properties*. When the following dialog appears, select *Multiple startup projects* in the *Common Properties | Startup Project* section. Then select *Start* as Action by the *ServiceHostConsole* and *ClientConsole* projects and then press **OK**.



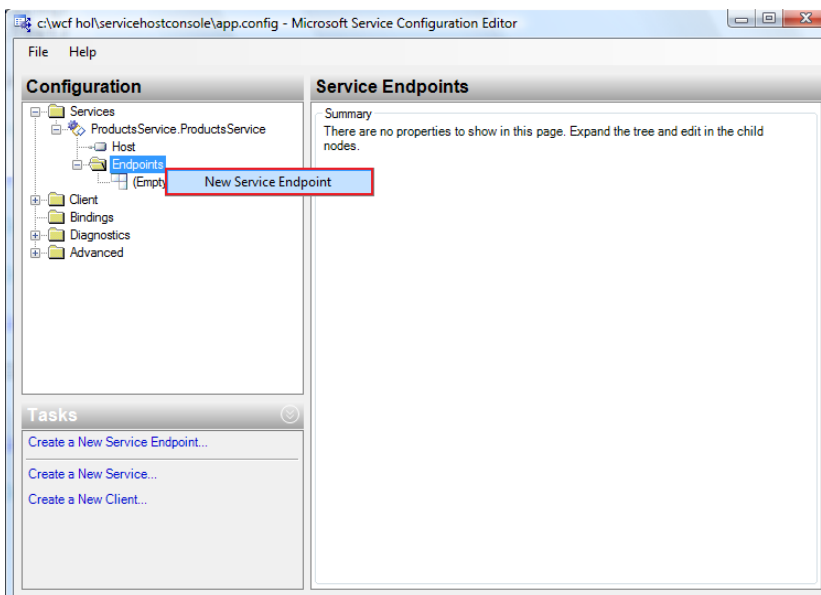
- 1.57 Press **Ctrl+F5**. The client and the hosting console applications starts. The hosting application is listening for requests coming from client. Now execute the `proxy.GetStockLevel(prodID)` call (call to the WCF service) by pressing **ENTER** inside the client console application window. Then press ENTER to close both applications.



## Using Multiple Endpoints

So far we have just used the HTTP protocol. Of course this is a good choice if we want for instance communicate outside our organization. What if we need to have performance for communication inside the organization? We can use the TCP protocol, which is more efficient, compared to the HTTP one. What can you do to solve this problem? You can define multiple endpoints: one for external client accessing the service using HTTP, and another for internal windows clients accessing the service using TCP. This is exactly what you are going to implement in the following steps:

- 1.58 Open the App.Config file of the *ServiceHostConsole* application by using the WCF Configuration tool (if you don't remember how to do it have a look at step 1.23).
- 1.59 Expand the *Services* node, then the *ProductsService.ProductsService* one and then right-click on the *Endpoints* folder and select **New Service Endpoint** as shown in the following figure:

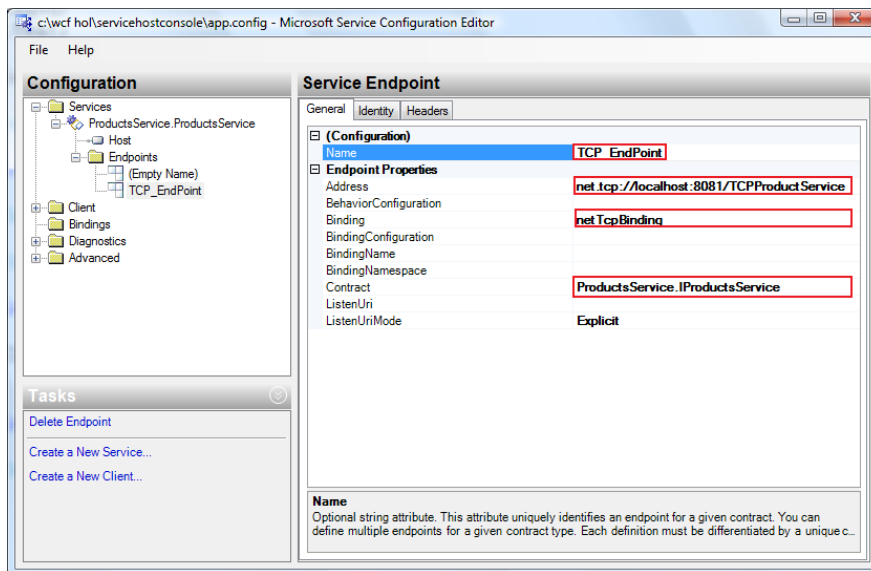


A new endpoint entry has been created.

1.60 Set the following properties as shown below:

Properties	Value
Name	TCP_EndPoint
Address	net.tcp://localhost:8081/TCPProductService
Binding	netTcpBinding (choose it from the combobox)
Contract	ProductsService.IProductsService

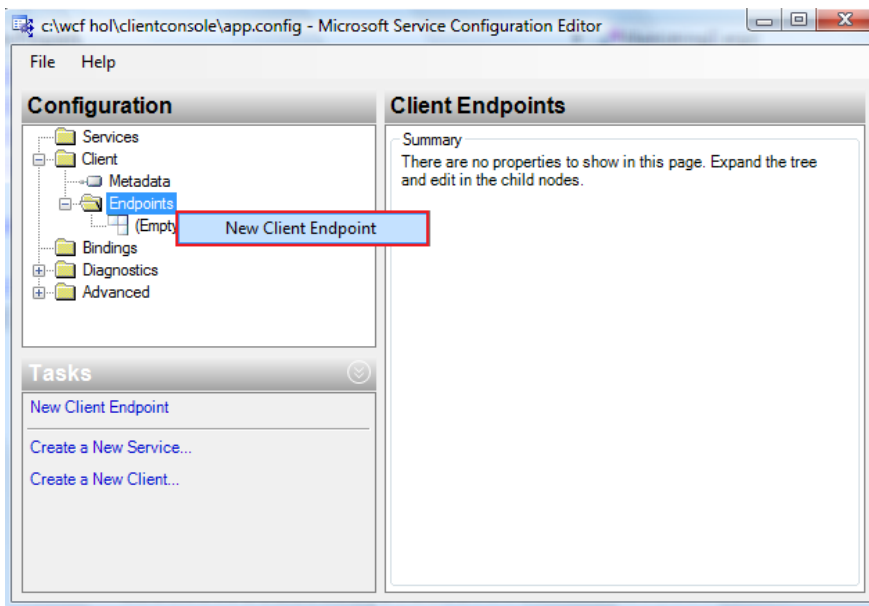
Note that by the address the schema is **net.tcp** (not *http*); the binding is again a provided WCF standard binding, while the contract is the same as the one of the HTTP Endpoint. Now you should have the following configuration:



1.61 Change the *Name* property to **HTTP\_EndPoint** of the other EndPoint (the one displayed as (Empty name) using the HTTP protocol).

1.62 Close the tool by saving the changes you made.

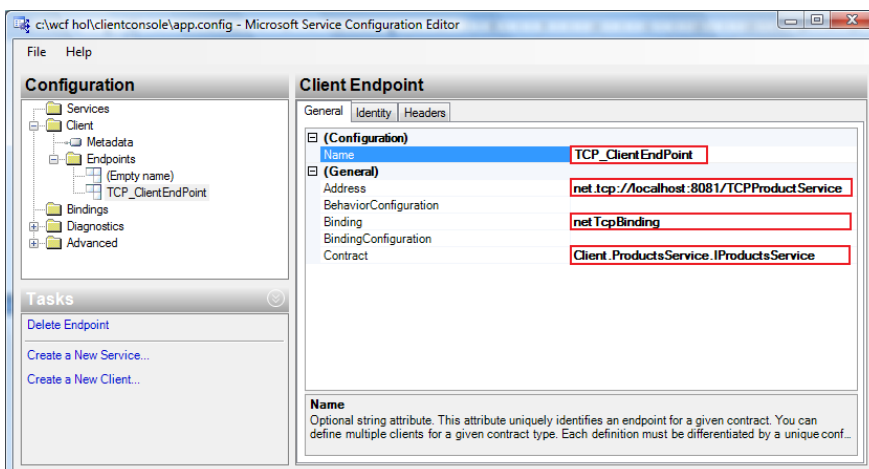
- 1.63 Open the App.Config file of the *ClientConsole* application by using the WCF Configuration tool. Expand the *Client* node and then right-click on the *Endpoints* folder and select **New Client Endpoint** as shown in the following figure:



- 1.64 Set the following properties as shown below:

Properties	Value
Name	TCP_ClientEndPoint
Address	net.tcp://localhost:8081/TCPProductService
Binding	netTcpBinding
Contract	Client.ProductsService.IProductsService

Note that the contract contains the namespace you defined during the creation of the proxy class with the *svcutil* utility (see step 1.41). Now you should have the following configuration.



- 1.65 Change the *Name* property to **HTTP\_ClientEndPoint** of the other EndPoint (the one displayed as (Empty name) using the HTTP protocol).
- 1.66 Close the tool by saving the changes you made.

- 1.67 You now have to tell the client application either to use the TCP protocol or the HTTP one. To do this open the *Program.cs* file from the *ClientConsole* application and add to the proxy class constructor (as first parameter) the name of the **client** endpoint you want to use as shown below. In our case we are going to use the TCP protocol.

```
static void Main(string[] args)
{
    Console.WriteLine("When the hosting application is ready press ENTER");

    Console.ReadLine();

    int prodID = 1;

    Client.ProductsService.ProductsServiceClient
    proxy = new Client.ProductsService.ProductsServiceClient("TCP_ClientEndPoint");

    int stockLevel = proxy.GetStockLevel(prodID);

    Console.WriteLine("Stock Level for productID {0} is {1}",
        prodID.ToString(),
        stockLevel.ToString());

    Console.ReadLine();
}
```

- 1.68 Press **Ctrl+F5** to start again the client and the hosting application. Press **ENTER** by the client application and check that everything is still working fine. **If a Windows Security Alert appears, click *Unblock* to allow the service to access the TCP port.** This time you are communicating with the WCF service using the TCP protocol!

## Protecting your Service

When you build a service and you are sending and receiving messages you can perform authentication and encryption at the **transport level** (SSL) and/or at the **message level**.

At this point our hosting application exposes 2 endpoints: one based on the TCP protocol (netTCPBinding) and the other using HTTP (BasicHttpBinding).

BasicHttpBinding (which conforms to the WS-BasicProfile 1.1 specifications) supports transport level security. The support for message level security is only available in combination with the WS-Security specifications, in which case you would therefore use the WsHttpBinding binding. The limitation of using WsHttpBinding is that you can then not be interoperable with ordinary ASP.NET Web Services. In fact ordinary ASP.NET Web Services don't implement the WS-Security specifications, while the WsHttpBinding as the name says does. But what you can do if you want to implement message confidentiality and still be interoperable with ordinary ASP.NET Web Services? You can use BasicHttpBinding with transport level security (SSL). And this is exactly what we are going to do for the HTTP Binding.

For the NetTcpBinding you can choose between using transport level security (SSL over TCP) or using message level security with the option to choose the encryption algorithm (and no need to configure SSL).

Remember that the scenario of our application is to have two endpoints: one for external client (running also on others platforms and using technologies other than .NET) accessing the service using HTTP and another for internal **windows** clients (to be used inside the organization) accessing the service using TCP.

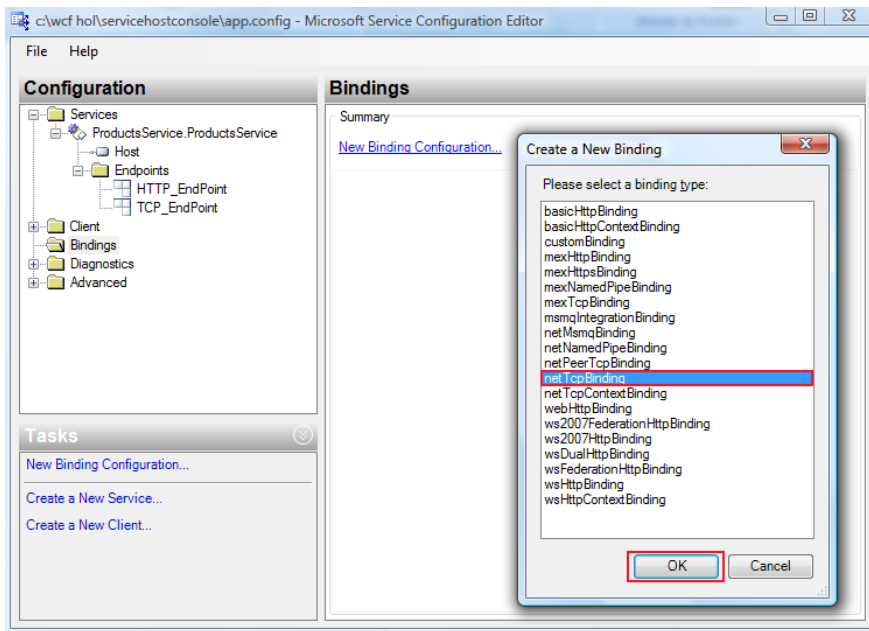
As in both cases we want to guarantee message confidentiality we need to implement security. In the case of the HTTP binding we are going to use transport level security (HTTP + SSL = HTTPS), while for the TCP one we are going to use message level security using Windows Integrated Security. This is the typical implementation used for a Windows environment. Note that when you use Windows Integrated Security, username and password are not transmitted as clear text and therefore you don't need to implement security at the transport level (no need of SSL).

Now let's start to protect the TCP Service at the message level.



Note that Transport level encryption is usually more efficient than message level encryption, but it requires more configuration tasks for the administrator (you need to install and configure an SSL certificate).

- 1.69 Open the App.Config file of the *ServiceHostConsole* application by using the WCF Configuration tool.
- 1.70 Right-click the *Bindings* folder and then click *New Binding Configuration...* In the *Create New Binding* dialog, select the **netTcpBinding** binding type and then click OK as shown in the following figure.



- 1.71 The tool generates a binding configuration with the default settings for the netTcpBinding.
- 1.72 In the right pane of the tool (in the *Binding* tab), change the *Name* property of the binding to **ProductsServiceTCPBindingConfig**.
- 1.73 Click on the *Security* tab and change the *Mode* property to **Message** (we want to use message level encryption). Leave the *AlgorithmSuite* property to **Basic256** and the *MessageClientCredentialType* property to **Windows**. With this configuration clients are expected to provide a valid Windows username and password.



Note that the *TransportWithMessageCredential* is also available as *Mode* property option. This is a mixed solution, where message level security is used for authentication purpose, while encryption is performed at the transport level.

- 1.74 Now you need to bind the TCP binding configuration you just created to the TCP Endpoint. To do this, click on the *TCP\_EndPoint* and set the *BindingConfiguration* property under the *General* tab by choosing **ProductsServiceTCPBindingConfig**.
- 1.75 Close the tool by saving the configuration.
- 1.76 On the server side we are now ready. As we need to do the same configuration changes for the Client Endpoint, open the *App.config* file of the *ClientConsole* application with the *WCF Service Configuration Editor* tool.
- 1.77 In the *WCF Service Configuration Editor*, right-click the *Bindings* folder and then click **New Binding Configuration**.
- 1.78 In the *Create New Binding* dialog, select **netTcpBinding** and then click OK.
- 1.79 Change the *Name* property of the binding to **ProductsServiceClientTCPBinding**.

- 1.80 Click the security tag and change the *Mode* property to **Message**. Leave the *AlgorithmSuite* property to **Basic256** and the *MessageClientCredentialType* property to **Windows**.
- 1.81 Click on the *TCP\_ClientEndPoint* and set the *BindingConfiguration* property under the *General* tab by choosing **ProductsServiceClientTCPBinding**.
- 1.82 Close the tool saving the configuration.
- 1.83 Press **Ctrl+F5** and check that everything is still working. This time the communication between the client and the server (included client's identity) is encrypted.

It's now time to protect the HTTP connection at the Transport Level using SSL. What do we have to do? First we will make the necessary configuration changes inside our client and server hosting applications. Once we are done, we need to create a Test SSL certificate. Of course this SSL certificate must not be used in a production environment, but it's just for development purpose. When you deploy a real production service, you need to buy one from a certification authority (such as Verign or Thawte). Instead of buying one you can always use Windows Certificate Services, which enables an organization to generate its own certificates.

...but let's start with the configuration at the application level.

- 1.84 First you need to tell the client application to use HTTP Endpoint. To do this open the *Program.cs* file from the *ClientConsole* application and change the parameter of the proxy class constructor to the name of the http client endpoint (**HTTP\_ClientEndPoint**) as shown below.

```
static void Main(string[] args)
{
    Console.WriteLine("When the hosting application is ready press ENTER");

    Console.ReadLine();

    int prodID = 1;

    Client.ProductsService.ProductsServiceClient
        proxy = new Client.ProductsService.ProductsServiceClient("HTTP_ClientEndPoint");

    int stockLevel = proxy.GetStockLevel(prodID);

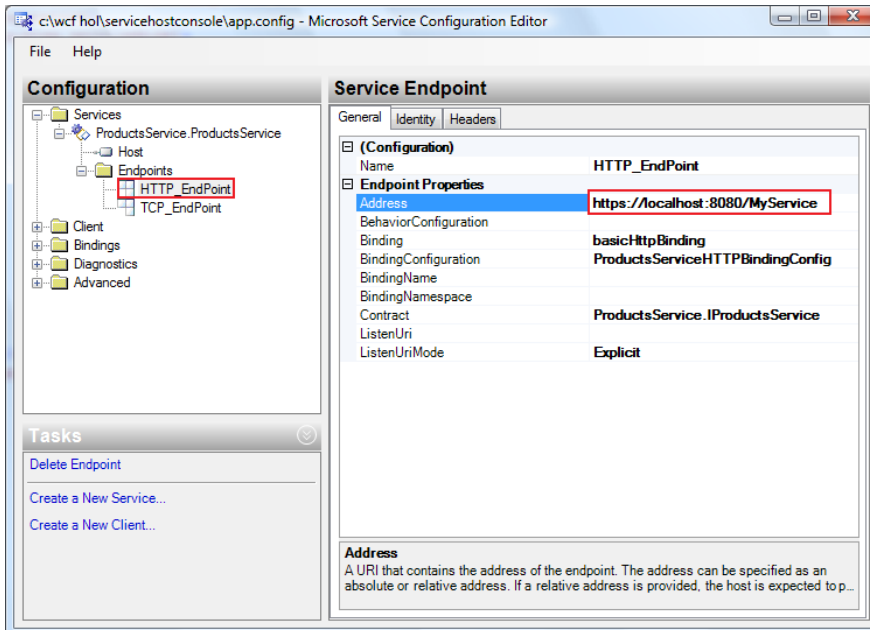
    Console.WriteLine("Stock Level for productID {0} is {1}",
        prodID.ToString(),
        stockLevel.ToString());

    Console.ReadLine();
}
```

- 1.85 Open once again the *App.Config* file of the *ServiceHost* application with the *WCF Service Configuration Editor* tool.
- 1.86 Right-click the *Bindings* folder, click *New Binding Configuration....*, select the **basicHttpBinding** binding type and then click OK.
- 1.87 Change the *Name* property of the new created binding to **ProductsServiceHTTPBindingConfig**.
- 1.88 Click the *Security* tab and set the *Mode* property to **Transport** (we want to use transport level security → SSL).

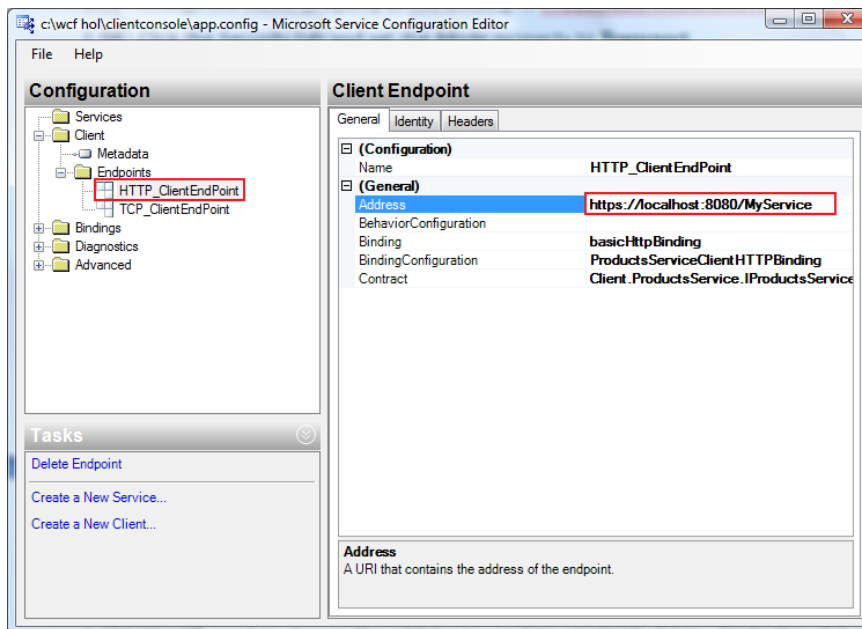


- 1.89 Set the *TransportClientCredentialType* to **Basic**. In this way in order to communicate with the service, client applications must provide a valid username and password and because of the use of SSL everything is encrypted at the transport level.
- 1.90 Now you need to bind the HTTP binding configuration you just created to the HTTP Endpoint. To do this, click on the *HTTP\_EndPoint* and set the *BindingConfiguration* property under the *General* tab by choosing **ProductsServiceHTTPBindingConfig**
- 1.91 As we are using SSL we need to make a small change to the address of the HTTP Endpoint: we need to change the schema of the address by adding a 's' to http. To do this click on the *HTTP\_EndPoint* and change the *Address* property to <https://localhost:8080/MyService> as shown in the following figure.



- 1.92 Close the tool by saving the configuration.
- 1.93 Now we need to do the same configuration changes for the Client HTTP Endpoint. To do this, open the *App.config* file of the *ClientConsole* application with the *WCF Service Configuration Editor* tool.
- 1.94 In the *WCF Service Configuration Editor*, right-click the *Bindings* folder and then click **New Binding Configuration**.
- 1.95 In the *Create New Binding* dialog, select **basicHttpBinding** and then click OK.
- 1.96 Change the *Name* property of the binding to **ProductsServiceClientHTTPBinding**.
- 1.97 Click the *Security* tab and set the *Mode* property to **Transport**.
- 1.98 Set the *TransportClientCredentialType* to **Basic**.
- 1.99 Click on the *HTTP\_ClientEndPoint* and set the *BindingConfiguration* property under the *General* tab by choosing **ProductsServiceClientHTTPBinding**.

- 1.100 We still need to change the address to use https instead of http. To do this click on the *HTTP\_ClientEndPoint* and change the *Address* property to <https://localhost:8080/MyService> as shown in the following figure.



- 1.101 Close the tool saving the configuration.
- 1.102 We aren't ready to start the whole solution yet. We still need to create and configure an SSL certificate for the WCF service and add some code to the client application that provide the client credentials to the WCF service. Let's start with the SSL certificate and therefore you first need to create a test SSL certificate by using the *makecert* utility. Open the Visual Studio 2008 command prompt windows and type the following command (*If you are using User Access Control, then start the command prompt as an administrator*).

```
makecert -sr LocalMachine -ss My -n CN=HTTPS-Server -sky exchange -sk HTTPS-Key
```

This command creates a certificate in the local computer store location (LocalMachine) and places it in the Personal folder (My). Be sure that the commands executes with success (*Succeeded* appears).

- 1.103 Now that the SSL certificate has been created we need to bind it with a HTTP port number. To do this we are going to use another utility that takes as parameter the thumbprint of the generated certificate. Therefore we first need to retrieve the thumbprint of our SSL certificate. This information can be obtained by using the *Certificates Microsoft Management Console snap-in*. To open it run the following command in the command prompt window:

```
mmc
```

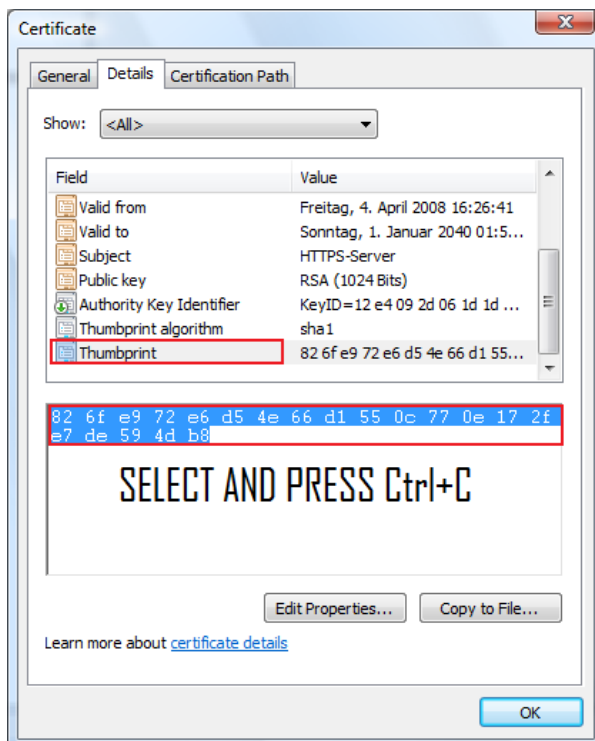
- 1.104 When the Microsoft Management Console opens, in the *File* menu click **Add | Remove Snap-In**.
- 1.105 In the *Add/Remove Snap-In* dialog, click **Add**.
- 1.106 In the *Add Standalone Snap-In* dialog, select the **Certificate** snap-in and then click **Add**.
- 1.107 In the *Certificate Snap-In* dialog, select **Computer account** and then click **Next**.
- 1.108 In the *Select Computer* dialog, select **Local Computer** and then click **Finish**.
- 1.109 In the *Add Standalone Snap-In* dialog, click **Close**.

1.110 In the *Add/Remove Snap-In* dialog, click **OK**.

1.111 In the left pane of the tool, expand the *Console Root* node, expand the *Certificates (Local Computer)* node, expand the *Personal* folder and then click the *Certificates* folder. On the right side you can now see the test certificate you created before (**HTTPS-Server**).

1.112 Double-click the *HTTPS-Server* certificate.

1.113 In the *Certificate* dialog, click the **Details** tab and select the **Thumbprint** property. Now copy (using **Ctrl+C**) or make a note of the hexadecimal string displayed in the lower part of the window.



1.114 Return to the command prompt window.

1.115 If you are using **Windows Vista** run the following command replacing the certhash with your own Thumbprint → **remove SPACES** from your hexadecimal string!

```
netsh http add sslcert ipport=0.0.0.0:8080 certhash=826fe972e6d54e66d1550c770e172fe7de594db8  
appid={00112233-4455-6677-8899-AABBCCDDEEFF}
```

**OR**

If you are using **Windows XP** run the following command replacing the hexadecimal string following the *-h* parameter with your own Thumbprint → **remove SPACES** from your hexadecimal string!

```
httpcfg set ssl -i 0.0.0.0:8080 -h 826fe972e6d54e66d1550c770e172fe7de594db8
```

If you can't find the httpcfg tool please download the [WinXP SP2 Support Tool](#) or [Win2003 version](#)

Be sure to obtain such a message or something similar: "SSL Certificate successfully added"

1.116 In order to test our application we need to implement a workaround. In fact when a client receives the public key from the server, the WCF runtime attempts to check whenever the certificate/public key is valid and that the authority that issued it is trusted. Of course with our test SSL certificate this check would fail. To workaround this problem we can add some code to the client that override the certificate validation check. **This is not production code, it's just because we are using a test certificate!**

Add the *PermissiveCertificatePolicy* class to the *ClientConsole* namespace inside the *Program.cs* file of the *ClientConsole* application and do all the necessary marked changes :



```
...

using System.Security.Cryptography.X509Certificates;
using System.Net;

namespace ClientConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            ...

            PermissiveCertificatePolicy.Enact("CN=HTTPS-Server");

            Client.ProductsService.ProductsServiceClient proxy =
                new Client.ProductsService.ProductsServiceClient("HTTP_ClientEndPoint");
            ...
        }
    }
}

class PermissiveCertificatePolicy
{
    string subjectName;
    static PermissiveCertificatePolicy currentPolicy;

    PermissiveCertificatePolicy(string subjectName)
    {
        this.subjectName = subjectName;

        ServicePointManager.ServerCertificateValidationCallback += new
            System.Net.Security.RemoteCertificateValidationCallback(
                RemoteCertValidate);
    }

    public static void Enact(string subjectName)
    {
        currentPolicy = new PermissiveCertificatePolicy(subjectName);
    }

    bool RemoteCertValidate(object sender,
                            X509Certificate cert,
                            X509Chain chain,
                            System.Net.Security.SslPolicyErrors error)
    {
        if (cert.Subject == subjectName)
        {
            return true;
        }

        return false;
    }
}
```

1.117 We still need a way for the client to provide username and password to send to the WCF service. To do this you can use the *Username* property of the *ClientCredentials*, which hold username and password. You need of course to provide your own credentials (a valid username and password). Open the Main method of the *ClientConsole* application and make the necessary changes as shown below.



```
static void Main(string[] args)
{
    Console.WriteLine("When the hosting application is ready press ENTER");

    Console.ReadLine();

    //dirty code to read user input
    Console.WriteLine("Username: ");
    string username = Console.ReadLine();
    Console.WriteLine("Password: ");

    ConsoleKeyInfo cki = new ConsoleKeyInfo();
    StringBuilder pw = new StringBuilder();

    do
    {
        cki = Console.ReadKey(true);
        if (cki.Key != ConsoleKey.Enter)
            pw.Append(cki.KeyChar.ToString());
    } while (cki.Key != ConsoleKey.Enter);

    int prodID = 1;

    ...

    proxy.ClientCredentials.UserName.UserName = username;
    proxy.ClientCredentials.UserName.Password = pw.ToString();

    int stockLevel = proxy.GetStockLevel(prodID);

    Console.WriteLine("Stock Level for productID {0} is {1}",
        prodID.ToString(),
        stockLevel.ToString());

    Console.ReadLine();
}
```

- 1.118 What we still need to implement it's a way to get the identity of the user running the client application. To do this open the *ProductsService.cs* file in the ProductsService project and make the necessary marked changes as shown in the code below. Note that to keep things simple we are just writing the *username* using a *Console.WriteLine* statement (the result will appear inside the *ServiceHostConsole* application window).

```
public int GetStockLevel(int ProductID)
{
    string username = System.Threading.Thread.CurrentPrincipal.Identity.Name;

    Console.WriteLine("Username is {0}", username);

    int stockLevel =
        (
            from p in XElement.Load("Products.xml").Elements()
            where p.Attribute("ProductID").Value == ProductID.ToString()
            select int.Parse(p.Element("StockLevel").Value)
        ).FirstOrDefault();

    return stockLevel;
}
```

- 1.119 Recompile the whole solution and press **Ctrl+F5**. Press ENTER inside the client application and when prompt for the username, please provide the name of your domain or computer (if you are not a member of a domain) followed by a backslash and by the username (DOMAIN\username or COMPUTER\_NAME\username) and then press ENTER. Then provide the password and press ENTER again.

Note that once the call to the *GetStockLevel* method is executed, the username you provided will appear inside the service host console application window.

- 1.120 You can now close you solution and Visual Studio 2008.

### **Congratulation!**

**You have reached the end of this Hands-On Lab. I hope you enjoyed it and you learned quite a lot of interesting things. If you still have some energy left ☺ there is another optional part where you will see how you can host NON-HTTP WCF Services inside IIS version 7, otherwise you can always use it as a reference.**

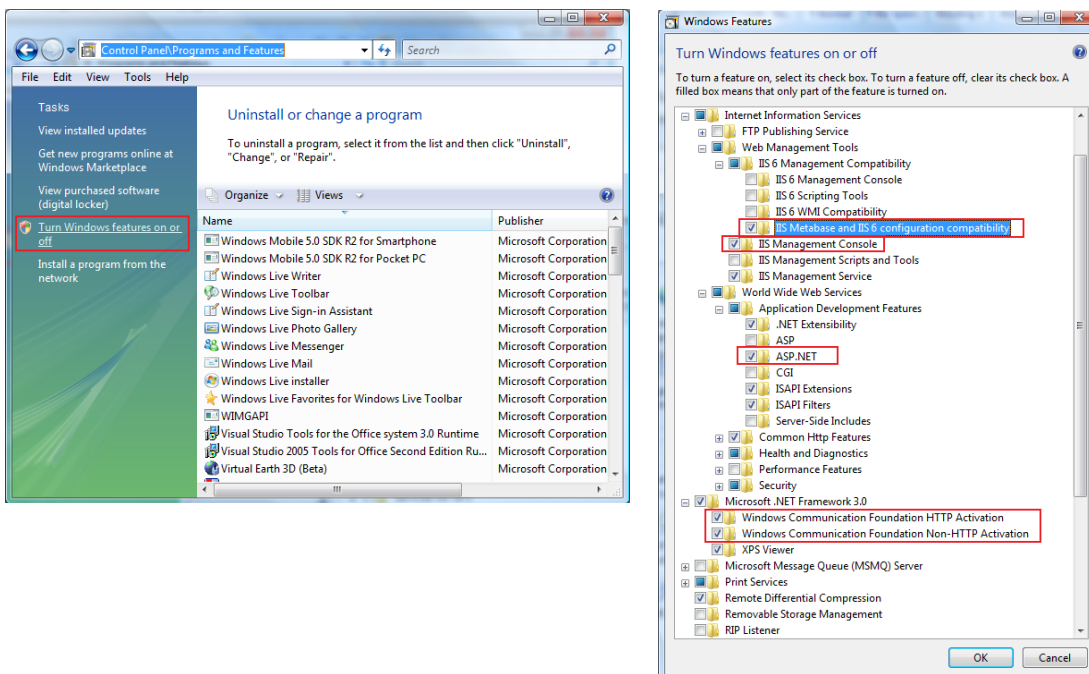
## IIS Version 7: support for non-HTTP protocols (Optional part)

To complete this optional part you need **Internet Information Server version 7**. This means that you need Windows Vista (ideally with SP1) or Windows Server 2008, because these are the only OS where you can install IIS7.

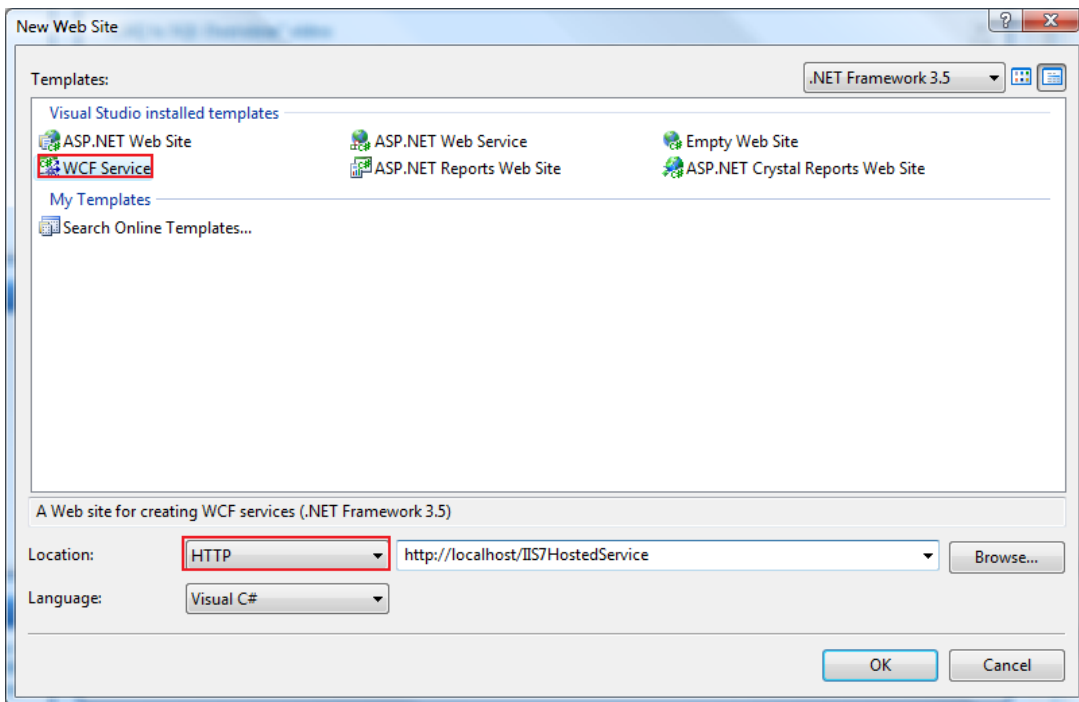
What you are going to do is using the new Windows Process Activation Service (WAS) component of IIS7. With WAS it is in fact possible to host WCF services beyond HTTP and without having to install the whole IIS package. Of course this offers you a lot of advantages: you can host **non-HTTP** WCF service endpoint inside IIS7 (something that wasn't possible with previous IIS version → only HTTP endpoints were supported) and therefore profiting from IIS built-in feature like process recycling, process isolation, easier support of HTTPS and XCopy deployment, which allows you to deploy a new version of a service by replacing the DLL's representing the service (while when you host your service for instance inside a Windows Service you need to stop the service and then start it again).

Of course hosting one service just in one place (in our case IIS7) can make things simpler. You can in fact configure your service to support multiple protocols by defining multiple endpoints. Therefore you can for example define a TCP endpoint (that uses the TCP protocol) that serves requests coming from your intranet (in terms of performance is better) and then define another endpoint, an HTTP endpoint, that communicates with the external world (through a firewall) and serves requests coming also from consumer not necessarily based on Microsoft technologies.

1.121 Now that we have seen some of the big advantages of WAS in relation with IIS7, it's time to build our sample. But before you start, be sure that you are using Windows Vista or Windows Server 2008. Then open the **Control panel | Programs and features** window and click on **Turns on Windows features on or off** as shown in the following figure on left side. When the following dialog on the right side appears, check that the red marked components are already installed. If not select them and then click on OK.

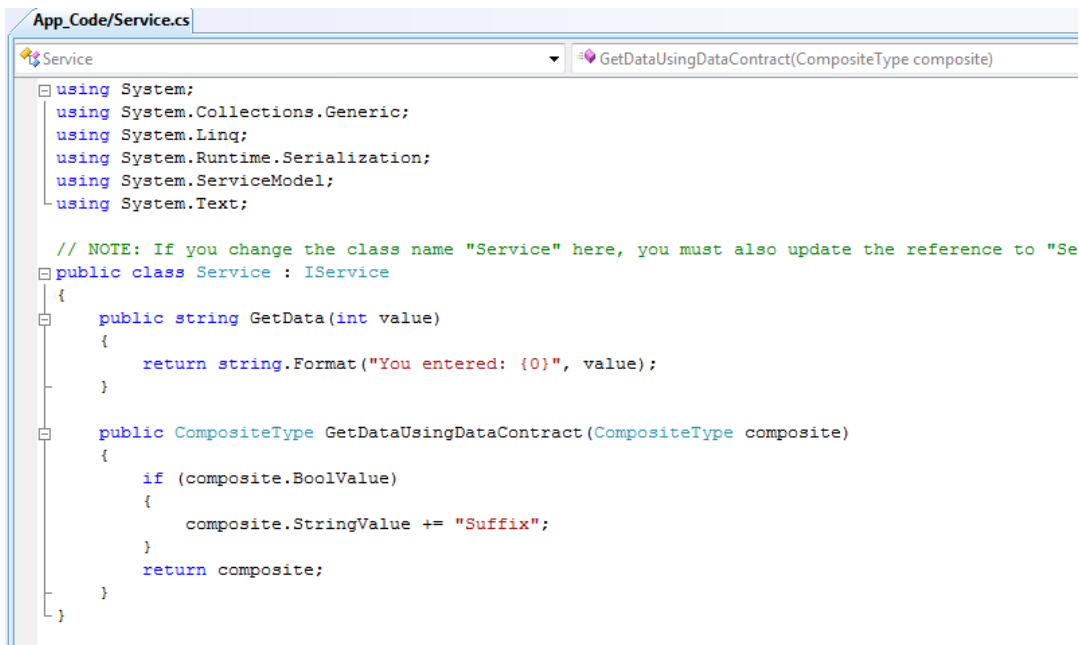


- 1.122 It's now time to open Visual Studio 2008 and create a new Web Site project (location HTTP) using the *WCF Service* template and name it *IIS7HostedService* as shown in the following image (If you are using User Access Control, then start Visual Studio 2008 as an administrator).



Note that by using the WCF Service template, the service will automatically be deployed to IIS (it is exactly what we want to do...).

- 1.123 If you now open the generated *Service.cs* file, you will see that a sample service with 2 operations (*GetData* and *GetDataUsingDataContract*) has already been created for you. For our sample we will use *GetData*).





1.124 As we want to modify the return value of the *GetData* method, by adding information on the used protocol, modify the method in the following way:

```
public string GetData(int value)
{
    string usedProtocol = OperationContext.Current.Channel.LocalAddress.Uri.Scheme;

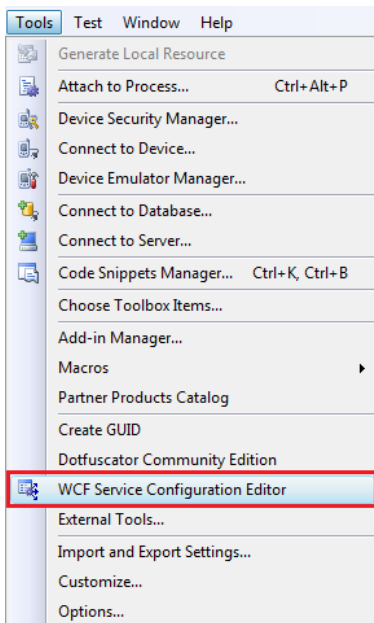
    return string.Format("You entered: {0} and you used protocol {1}", value, usedProtocol);
}
```



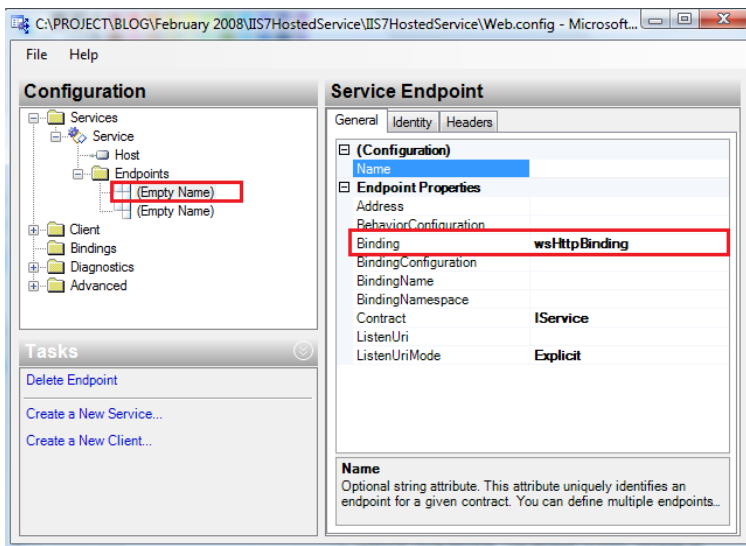
Notice that as we are going to host our service inside IIS, Visual Studio created a *Service.svc* file. This file is a service definition file required by IIS, which tells IIS how to identify the assembly containing the service implementation → a *ServiceHost* instance will then be created for this specific type.

Note also that the service definition file (*Service.svc*) must have the same name as the Web service and have the *.svc* suffix!

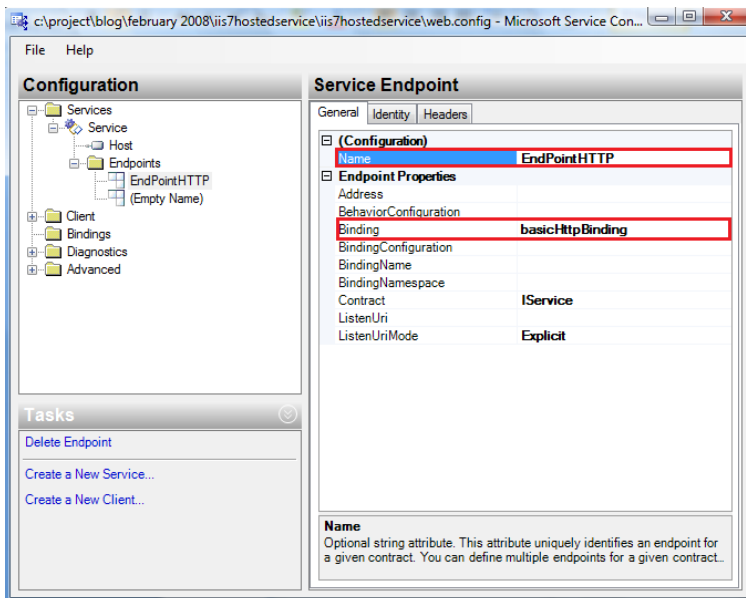
1.125 You can now open the *web.config* file by using the integrated *WCS Service Configuration Editor Tool* provided inside Visual Studio 2008.



- 1.126 Once you have opened the mentioned config file, you can expand the Services node and you will see that inside the Endpoints folder, 2 Endpoints have already been created. Select the one that uses the *wsHttpBinding* Binding.

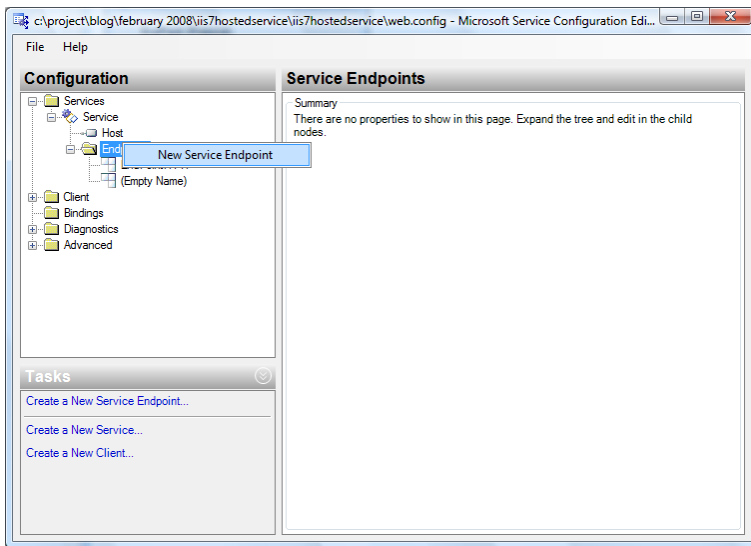


- 1.127 Change the **Name** property of the selected endpoint with *EndPointHTTP* and change the **Binding** property by selecting the *basicHttpBinding* as shown in the following figure:



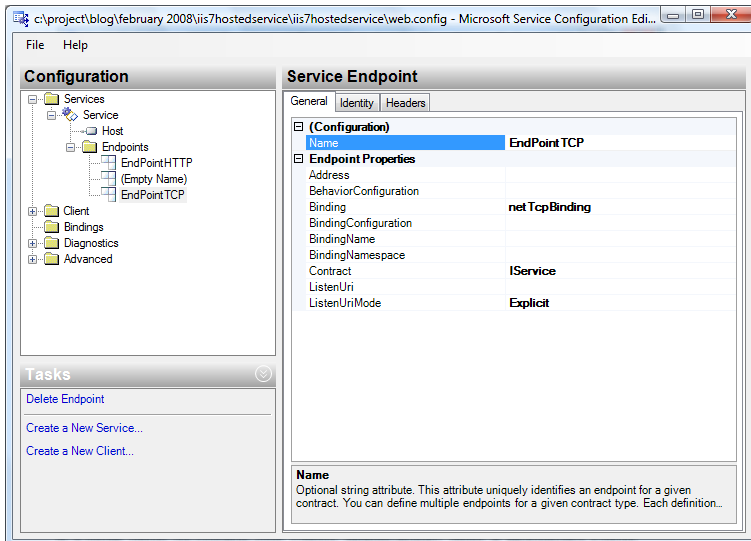
Note that the main difference between the *basicHttpBinding* and the *wsHttpBinding*, is that the first one conforms to the WS-I Basic Profile 1.1 (use this binding if you want to maintain compatibility with client developed to access the “old” ASMX-based web service), while the *wsHttpBinding* conforms to the WS-\* specifications.

1.128 Add now a second endpoint by selecting *New Service Endpoint* as shown in the following figure



and set the following properties.

Properties	Value
Name	EndPointTCP
Binding	netTcpBinding
Contract	IService

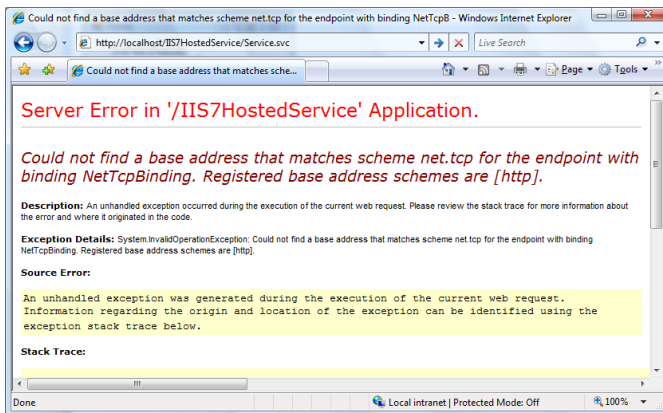


Note that you don't need to define the address part of the server side endpoint. Why that? Because as we are hosting a service in IIS, IIS use a combination of the URL of the WCF service and the name of the service definition file as the endpoint. This means that in this specific case the address of the endpoint that will be exposed from IIS is <net.tcp://localhost/IIS7HostedService/Service.svc> (*net.tcp* as root because we are using the TCP protocol).

Note also that instead of using *basicHttpBinding* we use this time *netTcpBinding* (the Contract remains the same). This also shows how it is easy with WCF to implement a service that can handle multiple protocols (it's just a configuration change inside a configuration file!).

1.129 **Close** the tool by saving the changes you made.

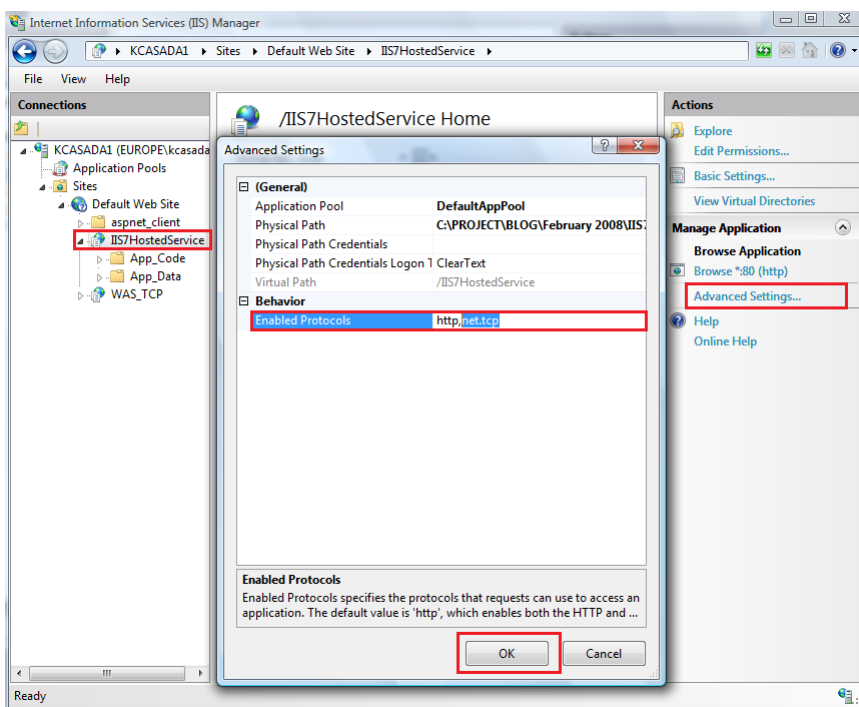
1.130 If you now open the browser and you enter <http://localhost/IIS7HostedService/Service.svc> (we are trying to access the http endpoint) you will get the following error:



In this case we are trying to access the HTTP Endpoint, but there is something wrong with the TCP Endpoint. The error message says that our Web Site doesn't support TCP; no address match the schema *net.tcp*. How is it possible? Actually we have defined a TCP endpoint... What do we have to do?

The problem is that we need to explicitly activate the support for the TCP protocol within IIS7. To solve this problem you have to proceed in different way: if you are using Windows Vista **with SP1** or Windows Server 2008 then proceed to the next step (1.131), otherwise (if you are using Windows Vista **without SP1**) proceed to step 1.132.

1.131 Open the IIS7 management console and look at the advance setting of our *IIS7HostedService* Web Application. You see that in the *Enabled Protocols* property just *http* is defined. **Add *net.tcp*** (separated by a comma) to the **Enabled Protocols** property as shown in the following figure and then press **OK**.



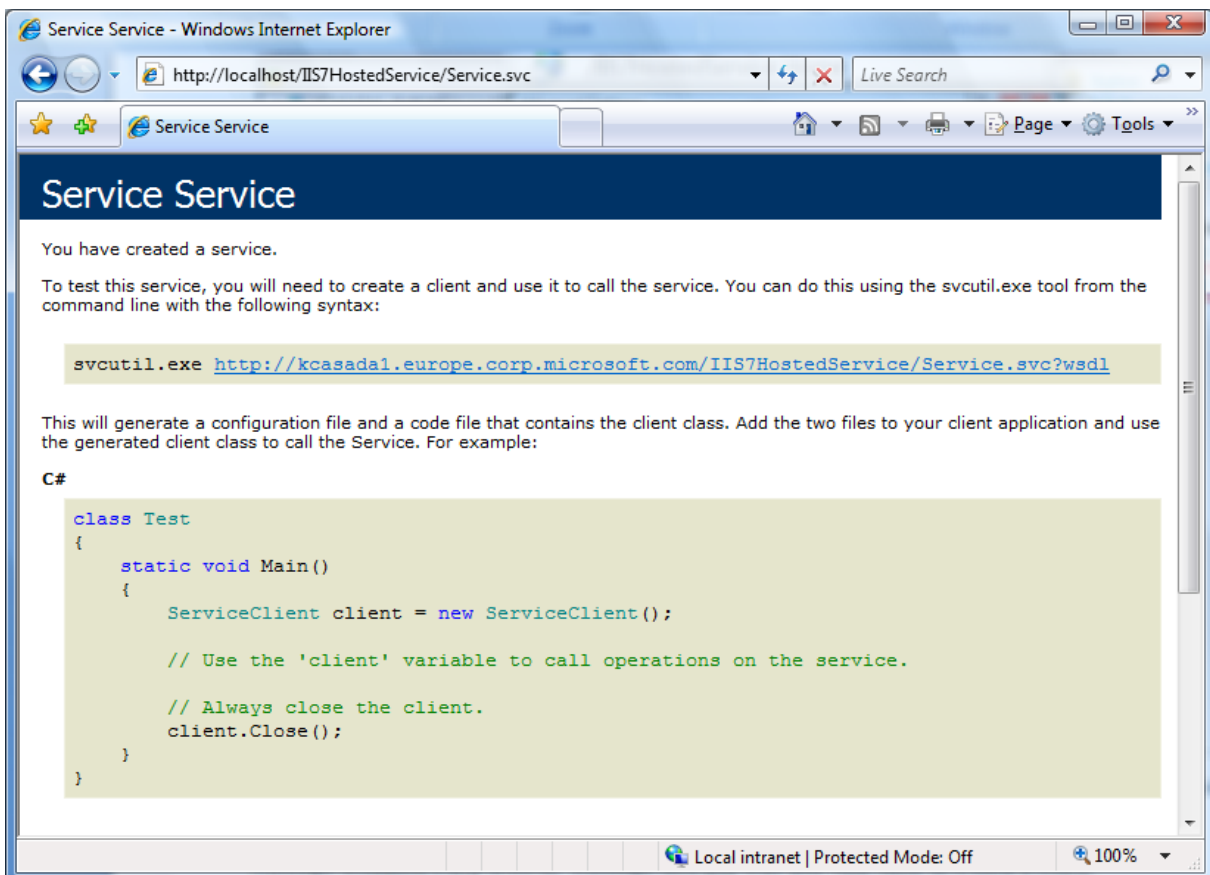
- 1.132 If you are using Windows Vista (**without SP1**) you have to locate and open the `%windir%\system32\inetsrv\config\applicationhost.config` file, locate the configuration entry for our application (`IIS7HostedService`) and **add** the `enabledProtocols` attribute with the necessary values as shown below. *If you are using User Access Control, then start the program you are using the open the file as an administrator.*

```
<application path="/IIS7HostedService"
             enabledProtocols="http,net.tcp">
```

**Save** and **close** the file.

Now our service is able to respond also to TCP requests.

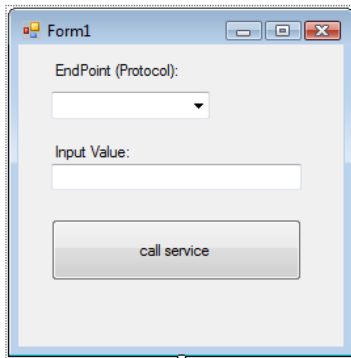
- 1.133 Go back to the browser window and click refresh. What happened? The error message has disappeared and another message tells you that you now have to create a client proxy class. This is exactly what you are going to do now.



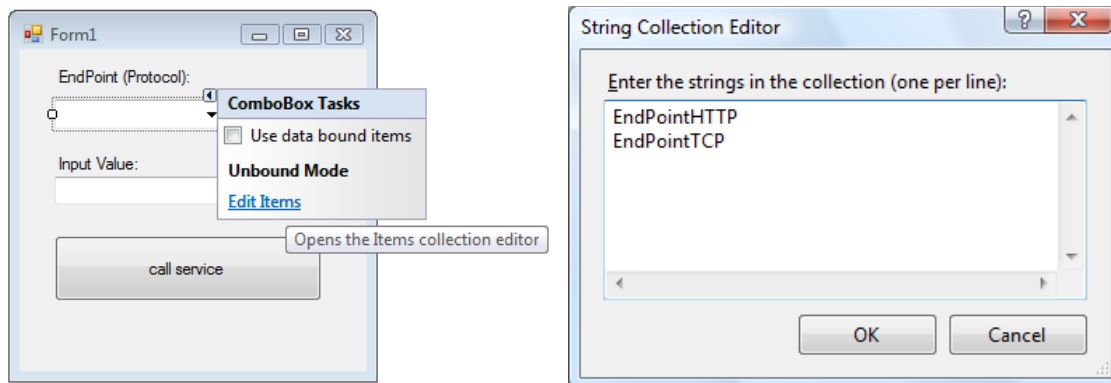
1.134 To do this you first have to create a Windows Form client application.

To do this add a Windows Form Application project to your solution and open the form designer. Drag&Drop a *comboBox* control, a *TextBox* Control and a *Button* control. Add a *label* control on top of the comboBox control and name it *EndPoint (Protocols)*. Add another *Label* control on top of the TextBox control and name it *Input Value*. Change the *Text* property of the button control to *call service*.

You should have something that looks like the following dialog:

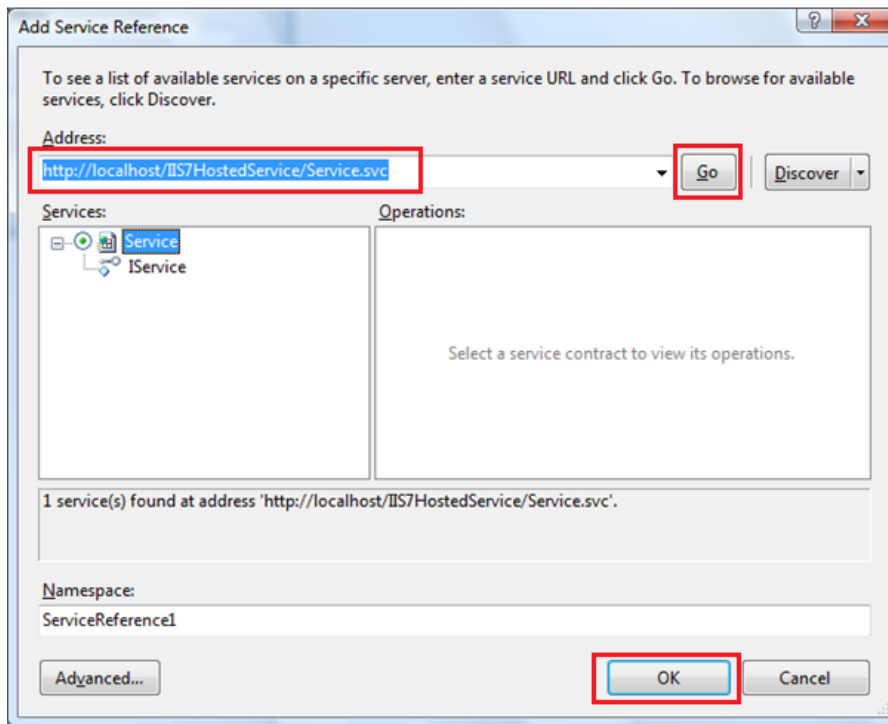


1.135 Fill the *Combox* control with fixed values by selecting *Edit Items* and entering the client endpoints name (*EndPointHTTP*, *EndPointTCP*) you defined before inside the WCF Configuration Tool, as shown in the following figure:



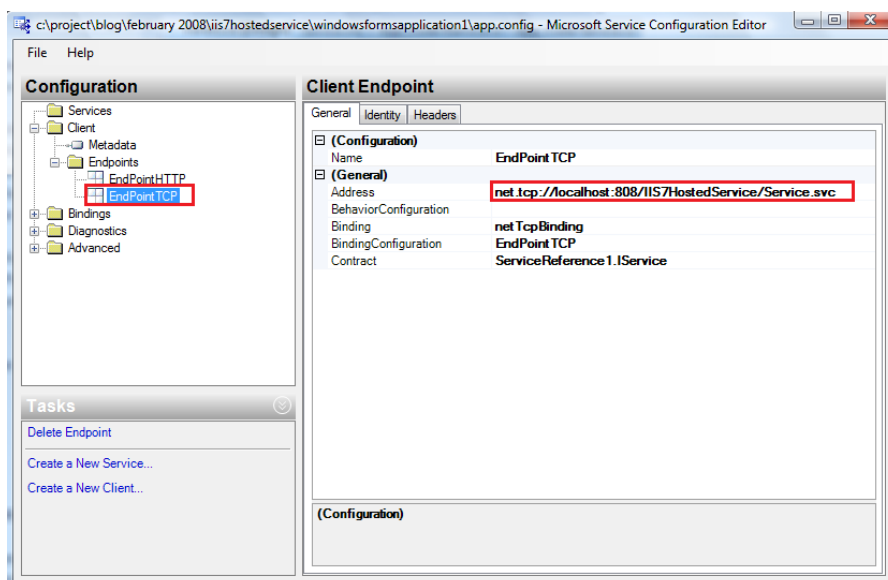
On the UI side you are now ready.

- 1.136 You still need to implement some code. Before you do that you still have to generate the proxy class. To do that *right-click* the Windows Form project and select **Add Service Reference**. Inside the *Add Service Reference* dialog enter the URL that point to the .svc file (<http://localhost/IIS7HostedService/Service.svc>: the same URL you entered before in the browser window), click Go and then OK.

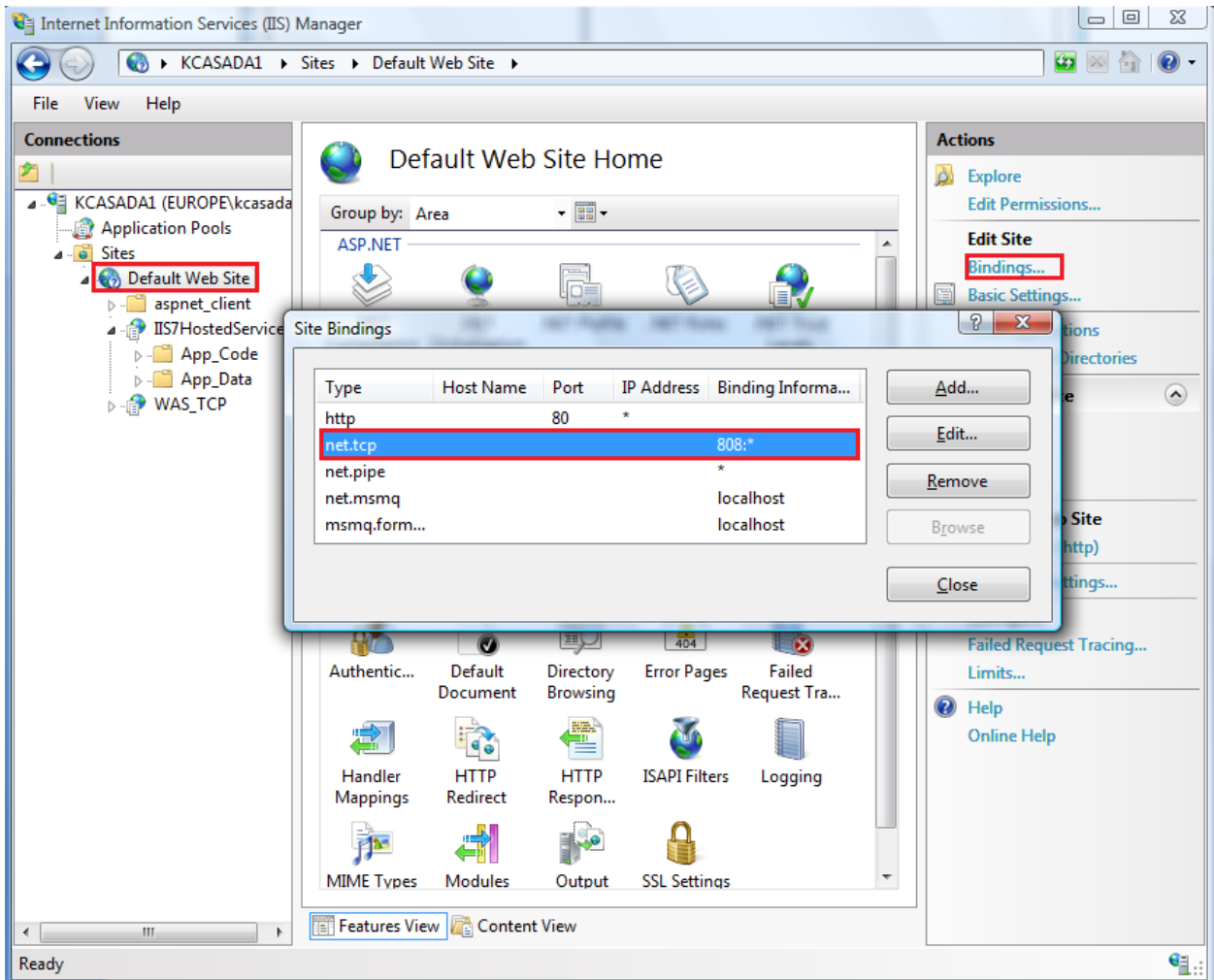


The proxy class has been generated! Moreover inside the *app.config* file two endpoints that reflect the endpoints we defined for the service have been generated.

- 1.137 You still need to make a small change to the *address* property of the client *TCP endpoint* using the WCF Configuration Editor tool. Replace the generated address with the following address as shown below.  
*net.tcp://localhost:808/IIS7HostedService/Service.svc*



Note that we replaced the machine name with *localhost* (it wasn't really necessary) and that we added the TCP port (808). But where you can find out the TCP port number? It's easy. Open the IIS management console, select the main Site and select *Bindings...* as shown in the following picture. As you can notice, for the *net.tcp* protocol the port 808 has been defined.





1.138 You now have to implement the *button\_click* EventHandler of our *call service* Button defined in the client application as shown in the following figure.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        ServiceReference1.ServiceClient proxy =
            new WindowsFormsApplication1.ServiceReference1.ServiceClient(comboBox1.SelectedItem.ToString());

        string retValue = proxy.GetData(int.Parse(textBox1.Text));

        MessageBox.Show(String.Format("{0}", retValue));
    }
}
```

Note that when we instantiate the proxy class (ServiceClient) we pass the name of the endpoint we want to use (value that is retrieved from our combobox). To the *GetData* method we then pass the value of the *TextBox* (it must be a number otherwise the application will throw an exception). The result of the service call will be shown inside a *MessageBox*.

1.139 Now you just need to test your application. Set the Windows Form Application as start up project, then press Ctrl+F5 and try to call the service (once by using TCP and once by using HTTP).

1.140 Did it work? I hope so 😊