

# Configuring WCF Using Visual Studio Wizards

Ted Faison  
April 2009

So you've decided to bite the bullet and develop a Windows Communication Foundation (WCF) service. One of the first decisions you'll need to make is how to host the service. WCF offers different hosting solutions. In the most simplistic scenarios, useful mostly for quick demos, you can host the service in a console or Windows Forms application. The service is then only available to clients if the application is running. Another option is to host the service in a Windows NT service. More commonly, WCF services are hosted by IIS. The service is instantiated automatically by IIS when clients make calls to it, using a URL. This article will show the steps involved in creating a service, hosting it in IIS and calling the service from a client application.

## Background

You might think that the process of creating a simple *Hello World* WCF service and client was a simple copy and paste operation. By the sheer length of this article, you can tell that the process is somewhat less than trivial. There are lots and lots of details involved – mostly involving Visual Studio wizards. Just to avoid confusion, I'll assume you don't know anything about the subject and will show you every Visual Studio screen involved. Some people may prefer less information, but others will find the lowest level of details extremely useful.

The overall process of creating a WCF-based system entails creating three separate components and hooking them up via WCF. The three components are:

- The WCF service
- The service host
- The client

The good news is that all the plumbing details are handled by Visual Studio wizards, without you needing to write almost any code. The bad news is that there are dozens of wizard screens to go through and setup.

This article is based on Visual Studio 2005. All the functionality is also supported by Visual Studio 2008, but some screens may look slightly different. My implementation is based on C#, but everything I cover also applies to VB.Net programmers.

## In a Nutshell

In an article with so many sections and figures, it is easy to get lost in the details. Before jumping into the nuts and bolts, here is what we're going to do:

1. Create a DLL with the basic service functionality that we plan to expose via WCF. My service will simply return the current time.
2. Create a Web project to expose the service functionality as a Web Service.

3. Configure the Web project to support dynamic metadata requests, so Visual Studio can automatically create a proxy to the service when creating a client application.
4. Create a client application, add a service proxy to it and call the service through the proxy.

## Creating a service DLL

To create a WCF service, you actually just create a simple **Class Library** DLL project. A given DLL can be exposed (without changes) by any type of host:

- A Web Service
- A Console Application
- A Windows Forms Application
- An NT Service
- Other

Let's get started. Open Visual Studio. Select the menu command **File -> New Project**. In the **New Project** dialog box, select the **Class Library** template and enter the name of the service, as shown in the next figure.

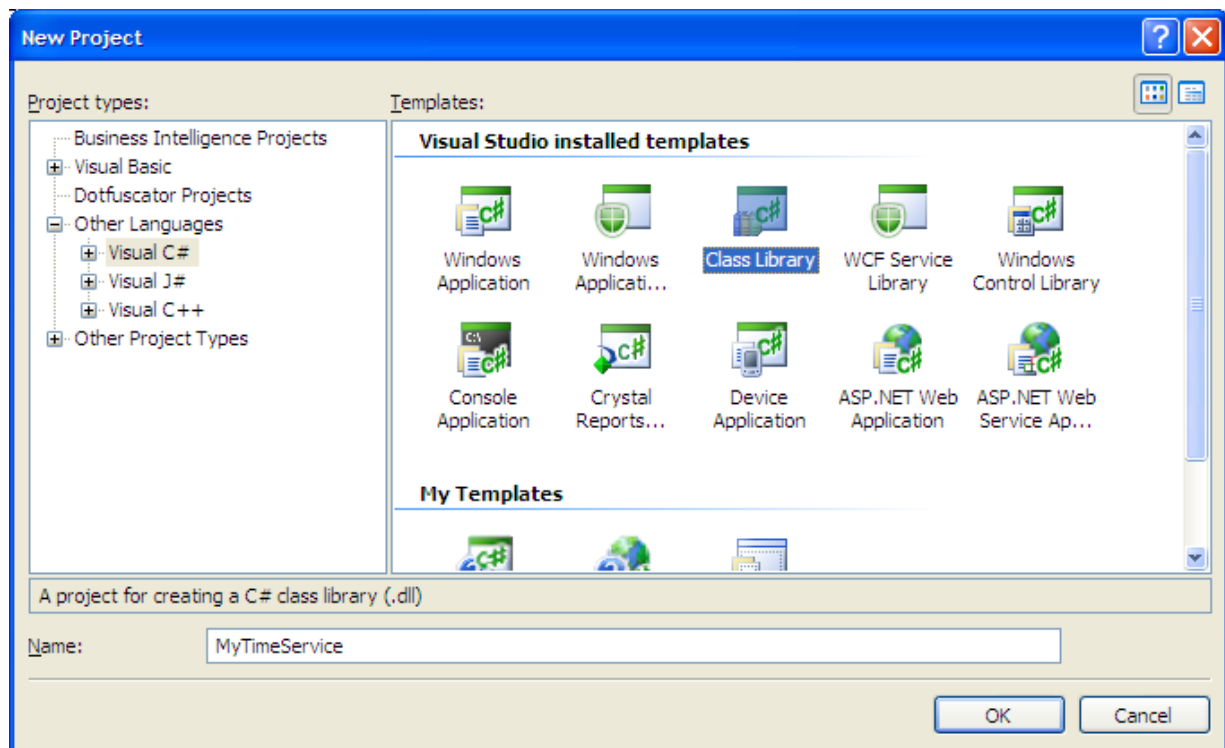
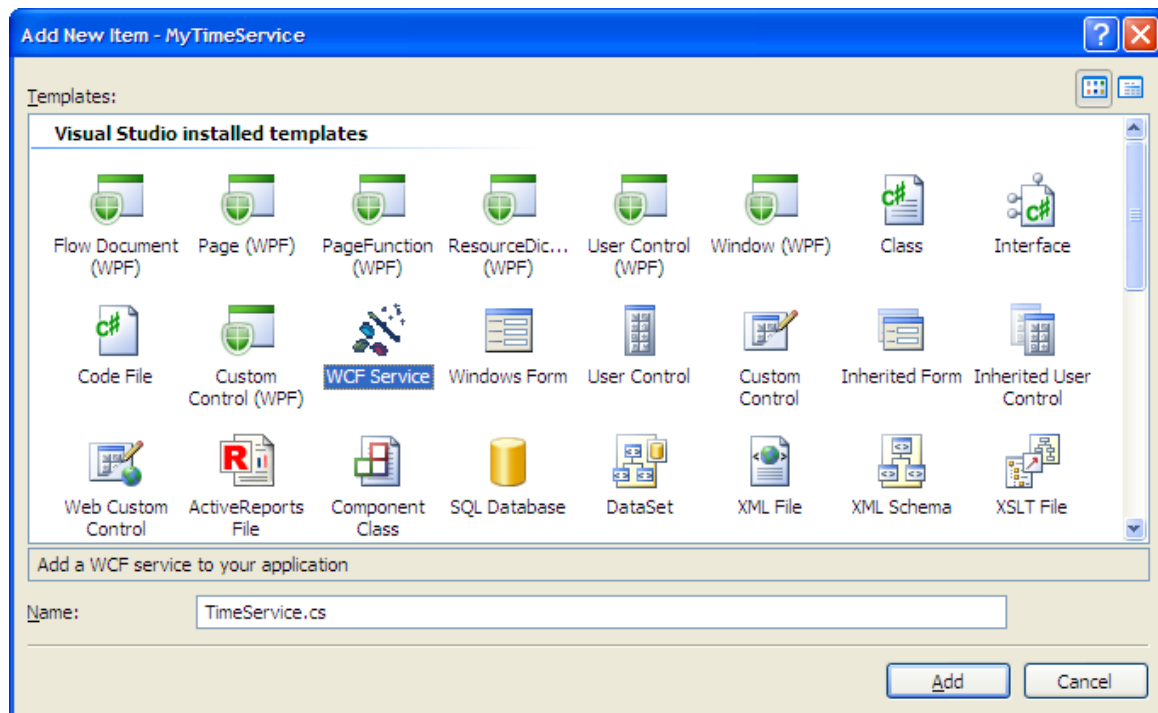


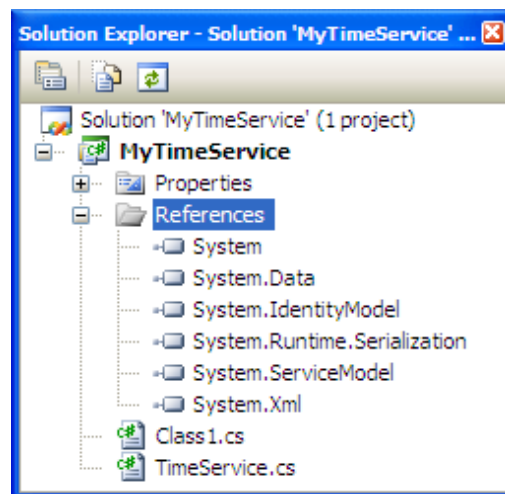
Figure 1 - Creating a service project.

The service described in this article will simply return the current time. As you can see in the figure, I named the service MyTimeService. Next, right click on the MyTimeService project in the Solution Explorer and select **Add New Item** from the context menu. In the **Add New Item** dialog box, select WCF Service, as shown in the next figure.



**Figure 2 – Adding a service class to the project.**

Use `TimeService` for the newly added item. Click the **Add** button. Visual Studio will add the item to the project, and also update the project references to include a number of WCF items, as shown in the next figure.



**Figure 3 – WCF references added to the project automatically.**

To create a WCF service class, you should choose **WCF Service** as the item type. You could have chosen the simple **Class** type. The advantage of using **WCF Service** as the type is that it makes Visual Studio add the WCF references to your project and also creates some useful default code in the generated class. The following listing shows the default

TimeService class created by Visual Studio. The class includes an interface called ITimeService, which is defined as a WCF ServiceContract.

```
using System;
using System.ServiceModel;

namespace MyTimeService
{
    [ServiceContract()]
    public interface ITimeService
    {
        [OperationContract]
        string MyOperation1(string myValue);
    }

    public class TimeService : ITimeService
    {
        public string MyOperation1(string myValue)
        {
            return "Hello: " + myValue;
        }
    }

    internal class MyServiceHost
    {
        internal static ServiceHost myServiceHost = null;

        internal static void StartService()
        {
            // Consider putting the baseAddress in the configuration system
            // and getting it here with AppSettings
            Uri baseAddress =
                new Uri("http://localhost:8080/MyTimeService/TimeService");

            // Instantiate new ServiceHost
            myServiceHost = new ServiceHost(typeof(TimeService), baseAddress);

            myServiceHost.Open();
        }

        internal static void StopService()
        {
            // Call StopService from your shutdown logic (i.e. dispose method)
            if (myServiceHost.State != CommunicationState.Closed)
                myServiceHost.Close();
        }
    }
}
```

**Listing 1 - The default WCF service code created by Visual Studio.**

The default code includes a class called MyServiceHost. If your service will be hosted by IIS, delete the MyServiceHost class, since it won't be used. The default ServiceContract contains an operation named MyOperation1. We'll add a new operation called GetCurrentTime, as highlighted in the following listing.

```

[ServiceContract()]
public interface ITimeService
{
    [OperationContract]
    string MyOperation1(string myValue);

    [OperationContract]
    DateTime GetCurrentTime();
}

```

I'll add a GetCurrentTime implementation to TimeService class:

```

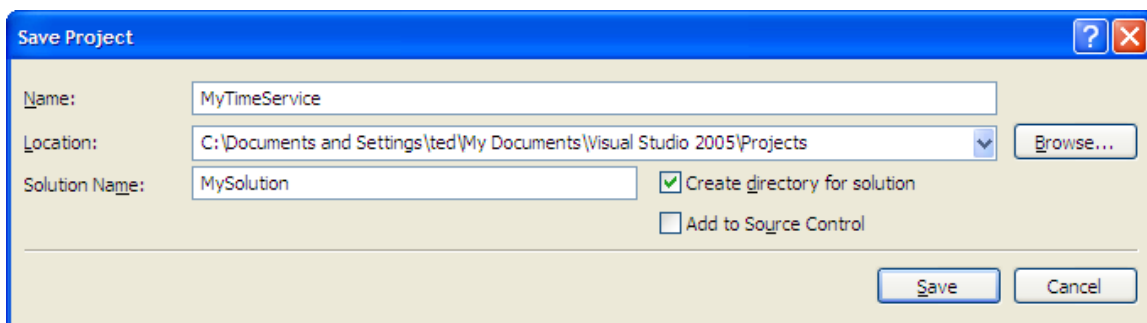
public class TimeService : ITimeService
{
    public string MyOperation1(string myValue)
    {
        return "Hello: " + myValue;
    }

    public DateTime GetCurrentTime()
    {
        return DateTime.Now;
    }
}

```

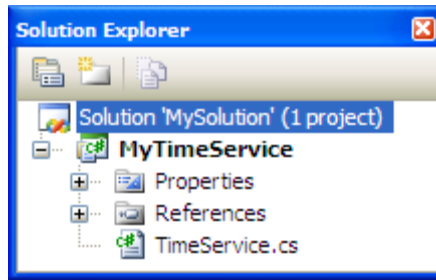
#### Listing 2 - Adding a new operation to the our service.

You can delete the operation MyOperation1 from the interface and class. Also delete the class1 file from the project. Save the solution, using the menu command **File -> Save All**. The **Save Project** dialog box allows you to change the name of the solution, which defaults to the project name. To avoid any confusion between the solution and project files, I renamed the solution to MySolution, as shown in the next figure.



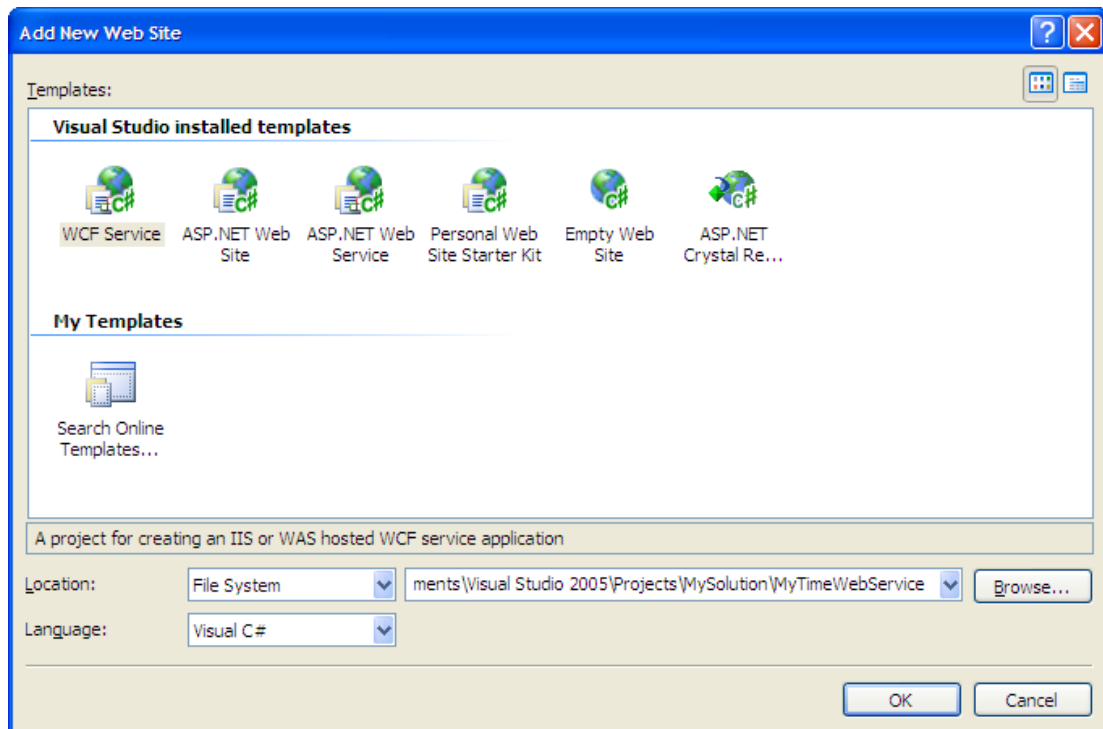
**Figure 4- Saving the project with a new solution name.**

At this point the Solution Explorer should look like the following figure.



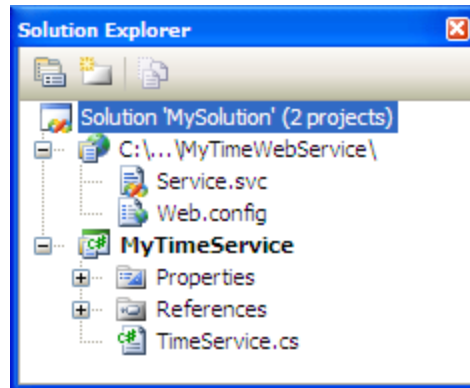
**Figure 5- The Solution Explorer showing the renamed solution.**

The next step is to create a web service that will host the WCF service. Right click on `MySolution` in the Solution Explorer and select **Add -> New Web Site** from the context menu. In the **Add New Web Site** dialog box, select WCF Service, as shown in the next figure.



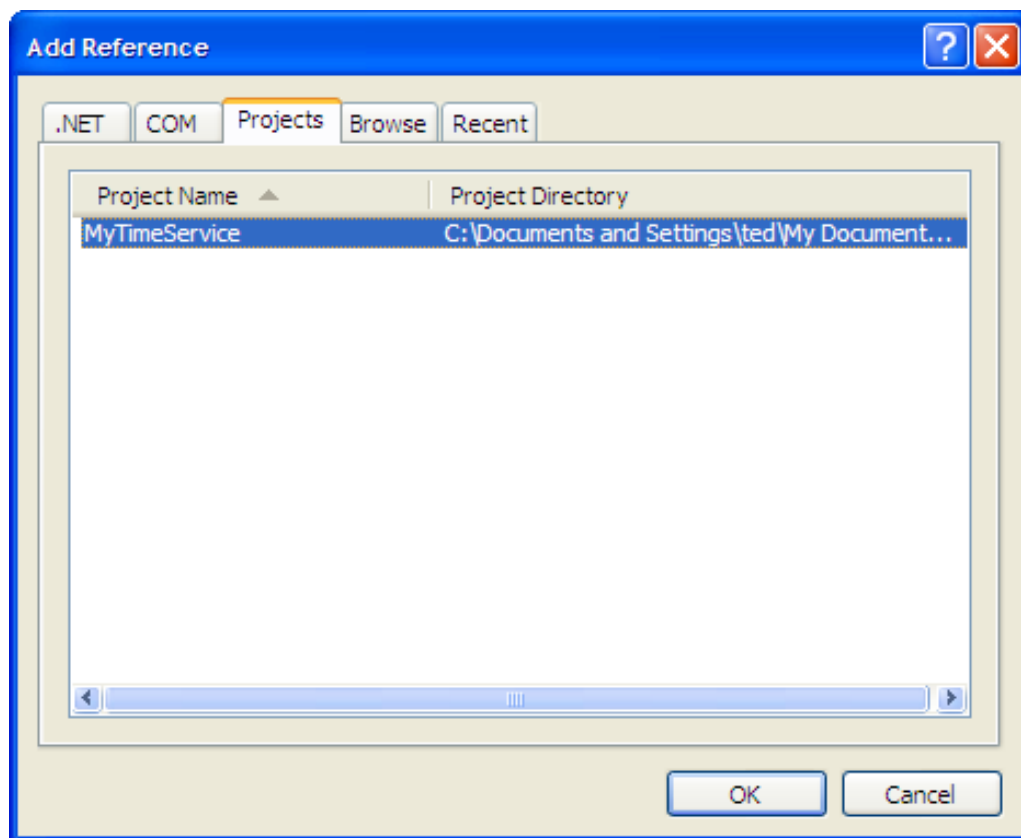
**Figure 6- Creating the host web project for the WCF service.**

For project template, choose **WCF Service**. You can put this web site project pretty much anywhere. I selected a folder under the `MySolution` folder. The created project contains a default service called `service.cs`. You can delete it, since we'll be using `TimeService.cs` (in the `MyTimeService` project) for our service. You can also delete the `App_Code` and `App_Data` items in the Solution. The Solution Explorer should now look like the next figure.



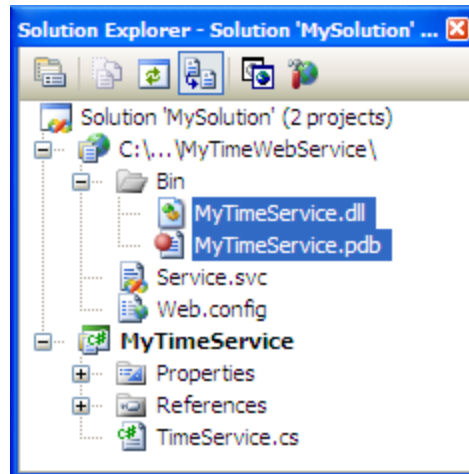
**Figure 7- The new web site added to the Solution Explorer.**

In project `MyTimeWebService`, add a reference to `MyTimeService` to. To do so, right click on the `MyTimeWebService` project node and select **Add Reference** from the context menu. In the **Add Reference** dialog box, go to the **Project** tab and select `MyTimeService`, then click **OK**.



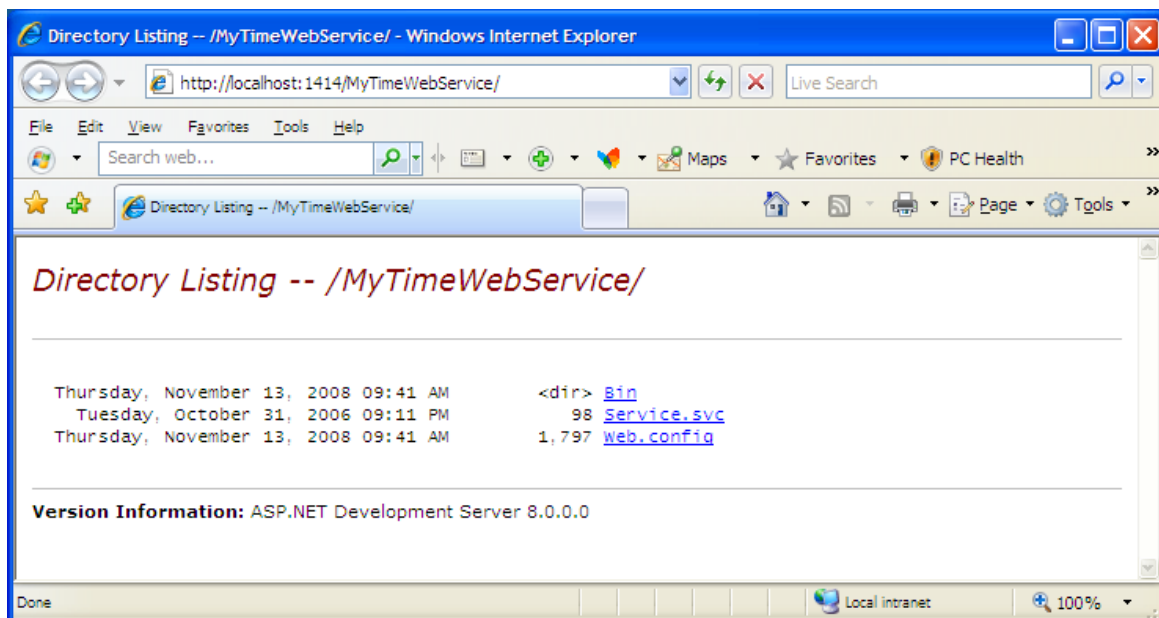
**Figure 8- Adding a project reference to the web project.**

Click **OK** and Visual Studio will add a `Bin` folder to the web site project, with two items, as shown in the next figure.



**Figure 9- The Bin folder added to the web project.**

To verify that everything is working so far, run the web project. To do so, right click on the MyTimeWebService project and select **View in Browser** on the context menu. The following browser page should appear.



**Figure 10- The default page that appears when running the web project.**

Note the URL and port number (1414 in this example) that appears in the **Address** field in Internet Explorer. We'll use this URL later, to test the service and to call it from the client side. This browser window confirms that the web project is configured and running correctly.

Now we need to hook our WCF service into the web project. A standard WCF service should make its metadata available, so clients can use it to easily construct a proxy to the service. When the service host is a web project, WCF metadata is returned by referencing the web's



`Service.svc` resource. The browser page shown in the previous picture shows the web project's base URL. The following URL will access the `Service.svc` resource:

`http://localhost:1414/MyTimeWebService/Service.svc`

We'll use this URL later, to generate the service proxy on the client side.

## Editing the `Service.svc` File

In order for the URL to work, we need to edit the default `Service.svc` file in project `MyTimeWebService`. The following listing shows `Service.svc` before and after the changes.

### Before

```
<% @ServiceHost Language=C# Debug="true"
Service="MyService" CodeBehind="~/App_Code/Service.cs" %>
```

### After

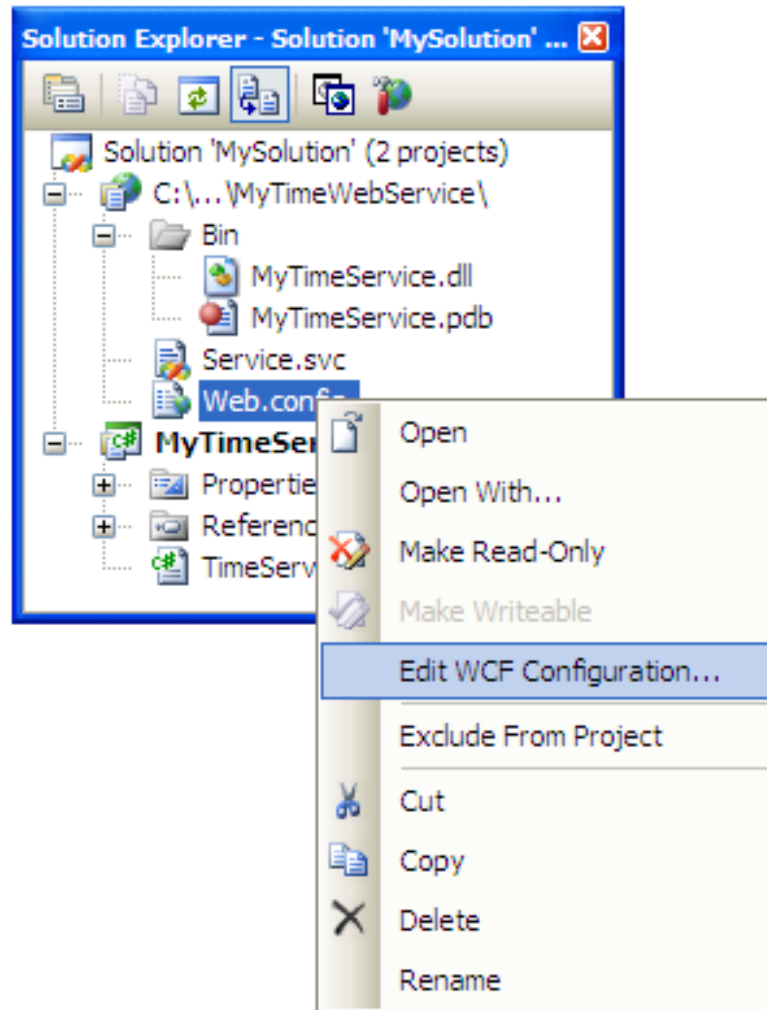
```
<% @ServiceHost Language=C# Debug="true"
Service="MyTimeService.TimeService" %>
```

**Listing 3 - The `Service.svc` file, before and after changing it.**

## Configuring the Service

There is still a long list of things left to do, to setup the configuration files for WCF. You can proceed in two ways: the easy way and the hard way. The easy way is to use Visual Studio's **WCF Configuration Wizard**. The hard way is to edit the XML configuration files manually –something you should consider doing only if you are completely familiar with the schema of the WCF-related configuration XML.

In this article, I'll use the easy way. The first task will be to change the web project's config file to allow metadata retrieval, which will facilitate the configuration of clients. In the Solution Explorer, right-click the `web.config` file of `MyTimeWebService`. From the context menu, select **Edit WCF Configuration**, as shown in the next figure. If this command is not available in the context menu, it means you don't have the right version of the .Net Framework installed. You probably have version 2.0. You'll need version 3.0 or better.

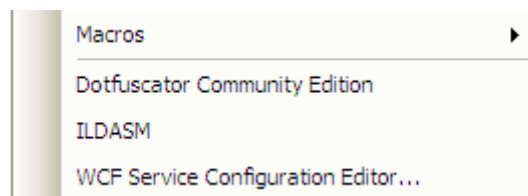


**Figure 11 - Starting the Visual Studio WCF Configuration wizard.**

There are two other ways to start the wizard:

- From the Visual Studio **Tools** menu
- From a command line prompt

In the **Tools** menu, there is an item called **WCF Service Configuration Editor**, as shown in the next figure.



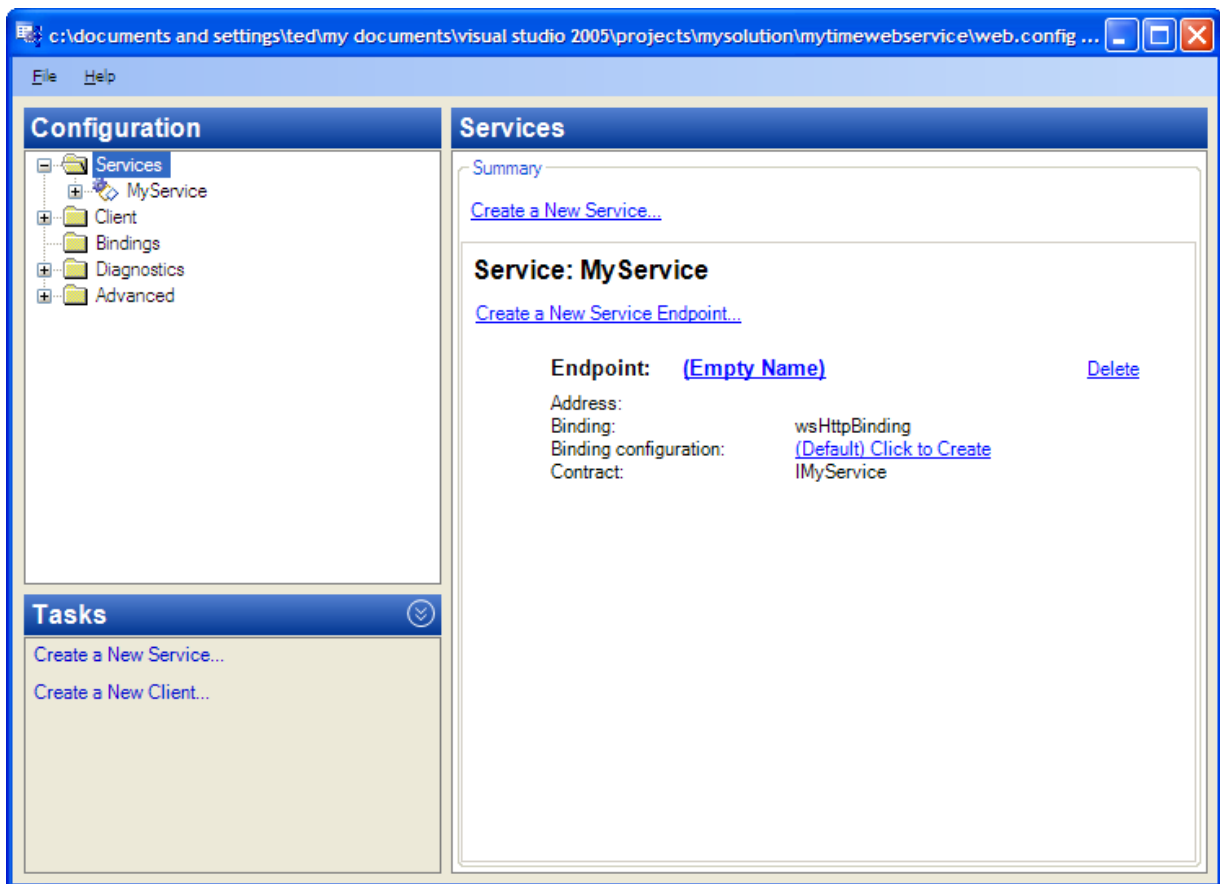
**Figure 12 - Starting the wizard from the Visual Studio Tools menu.**

To run the wizard from the command line, open a Visual Studio command prompt window and run the program `SvcConfigEditor.exe`. The location of the program may vary, depending on the version of Visual Studio you have. On my machine, with Visual Studio 2005, I found it in the folder:

```
C:\Program Files\Microsoft Visual Studio 8\Common7\IDE
```

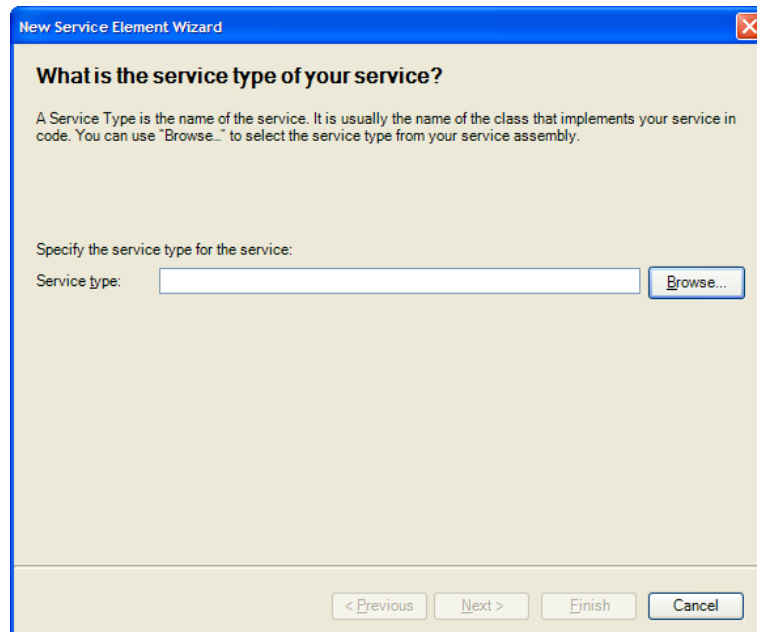
The file may also be located in other Visual Studio folders. The wizard comes with a help file, called `SvcConfigEditor.chm`.

If you run the wizard from the command line, it will start with a blank window. Use the menu **File -> Open** to open the configuration file you want to change. The following figure shows the wizard's initial appearance, after opening the `web.config` file of the `MyTimeWebService` project.



**Figure 13 - The WCF Configuration wizard's main screen.**

In the **Configuration** pane, right click on `MyService` and select **Delete** from the context menu. `MyService` is a default service that we'll replace with our new web service. Click on **Create a New Service** in the **Tasks** pane. The **New Service Element Wizard** will appear, as seen in the next figure.

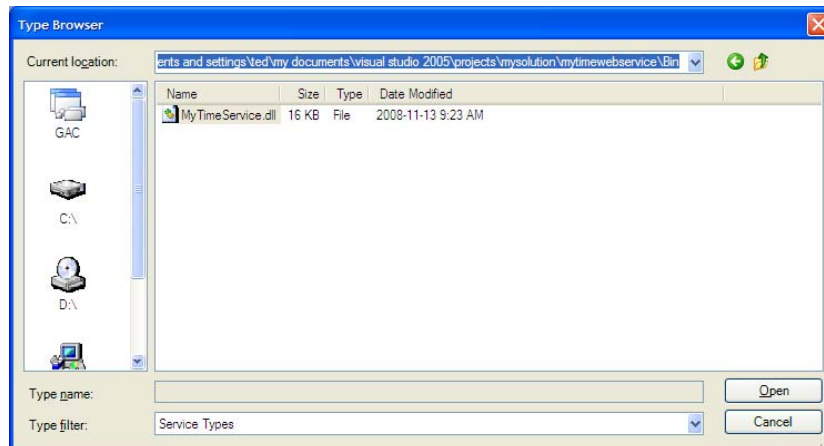


**Figure 14 - The wizard for creating new WCF service elements.**

Enter the path to the `MyTimeService.dll` in the Web Service. In this example, the path is the `Bin` folder of the `MyTimeWebService` project:

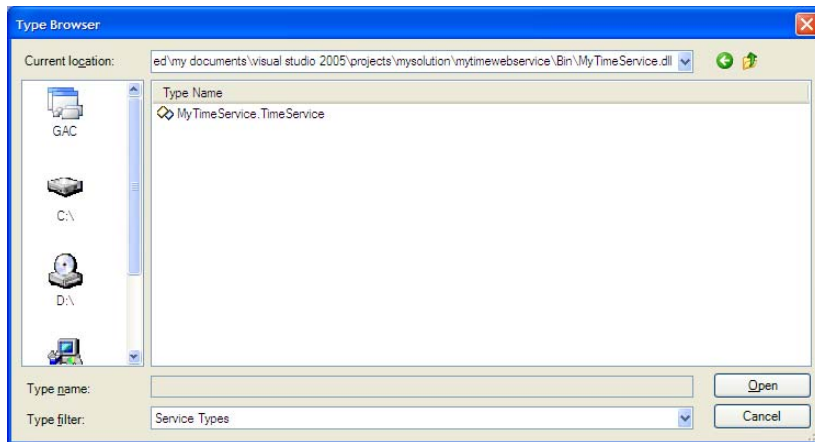
```
...\visual studio 2005\projects\mysolution\mytimeweb-service\Bin
```

After entering the path, click the **Next** button. The wizard will show a list of all the DLLs in the folder, as shown in the next figure.



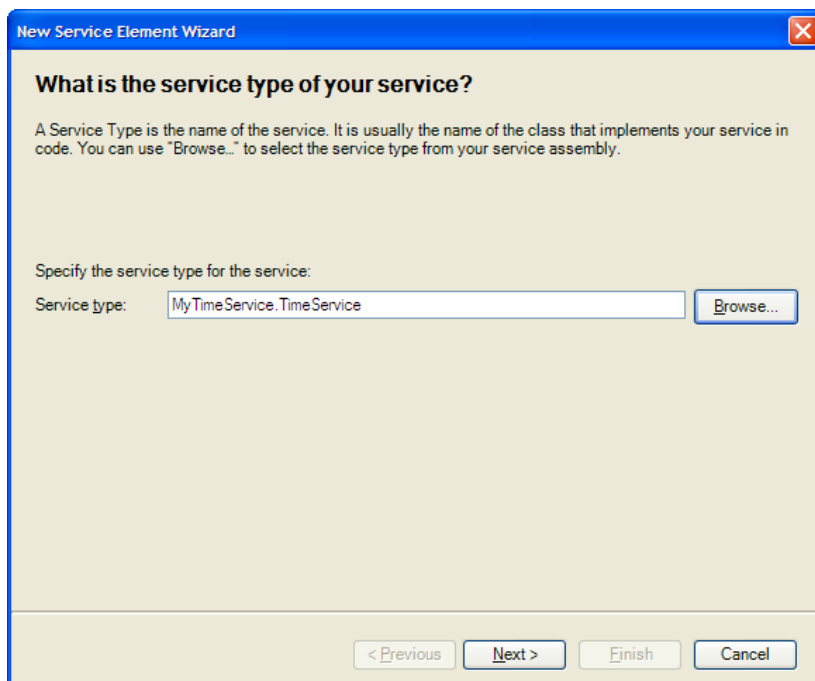
**Figure 15 - The list of service DLLs found by the wizard.**

In the dialog box, double-click `MyTimeService.dll`. The dialog box will then display all the WCF services contained in the DLL. In this example, there is only one, as shown in the next figure.



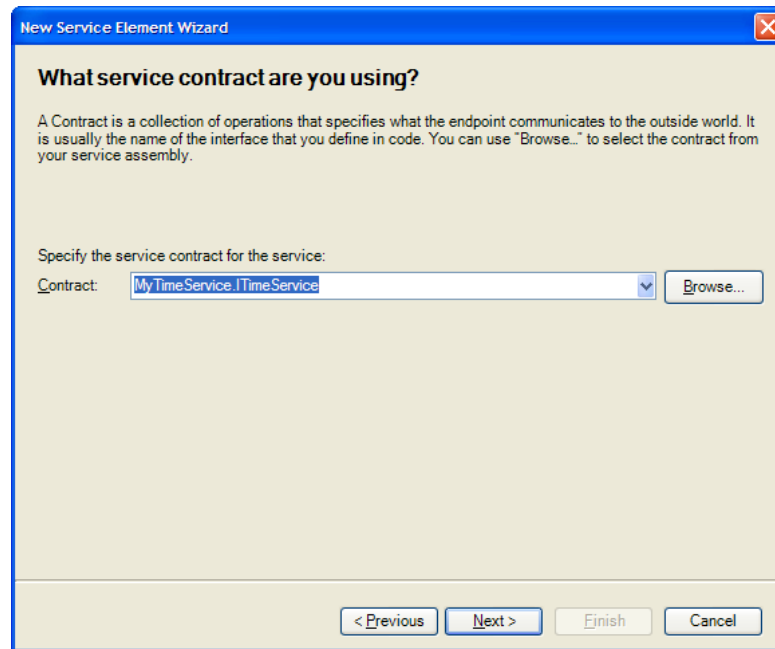
**Figure 16 - The list of services found in the selected DLL.**

Double-click `MyTimeService.TimeService`. The wizard will then start asking you for information about this service, as shown in the next figure.



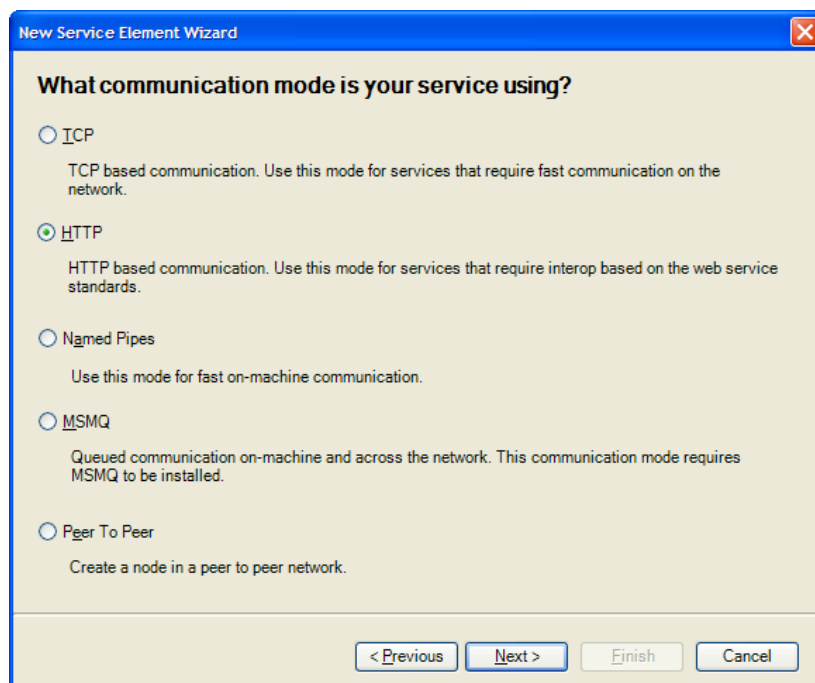
**Figure 17 - The service configuration wizard..**

The **Service type** textbox is automatically filled-in for you. Click **Next** to go to the next screen, shown in the next figure.



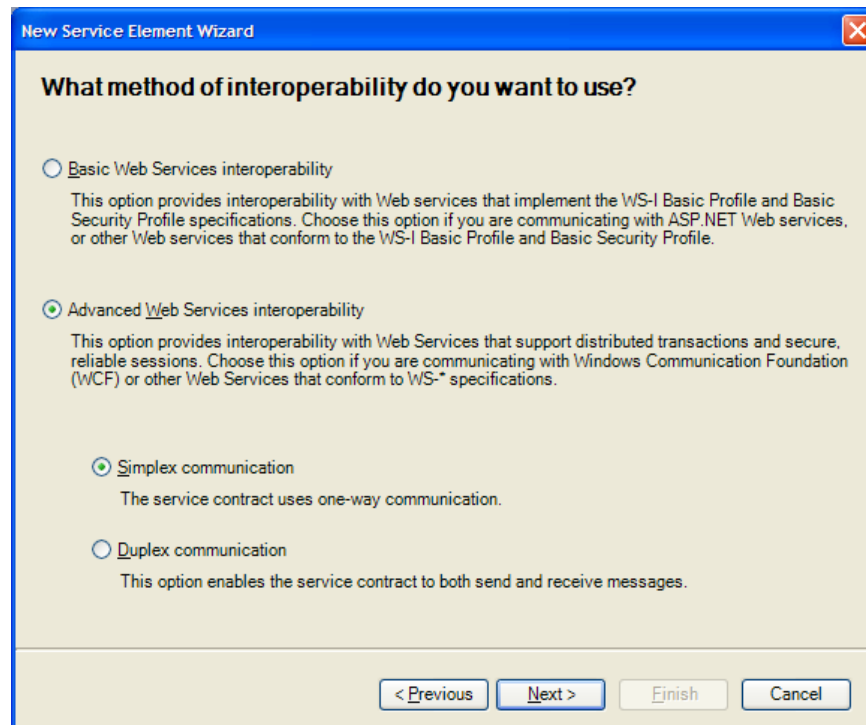
**Figure 18 – Specifying the contract used by the service..**

The **Contract** textbox should already be set. If not, you probably skipped a step or entered incorrect information on one of the previous wizard pages. Enter the contract name as shown in the figure above and click **Next**. The wizard will prompt you for the type of communication mode your service will use, as shown in the next figure.



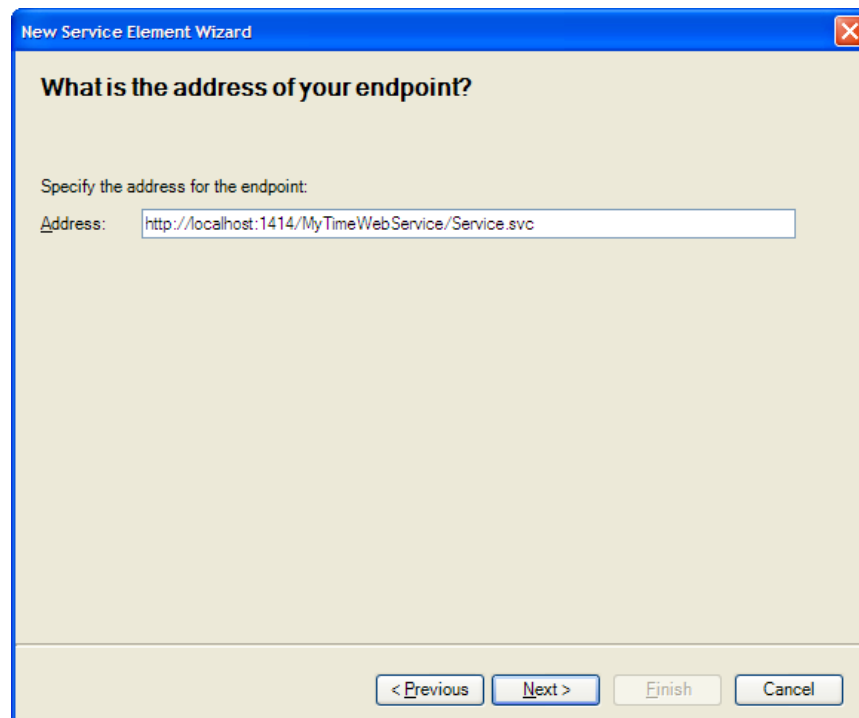
**Figure 19 – Specifying the service's communication mode..**

The default communication mode is HTTP, which is what we want, since we're configuring a web service as the WCF host. Click **Next** to go to the next screen, shown in the following figure.



**Figure 20 – Specifying the web service's interoperability..**

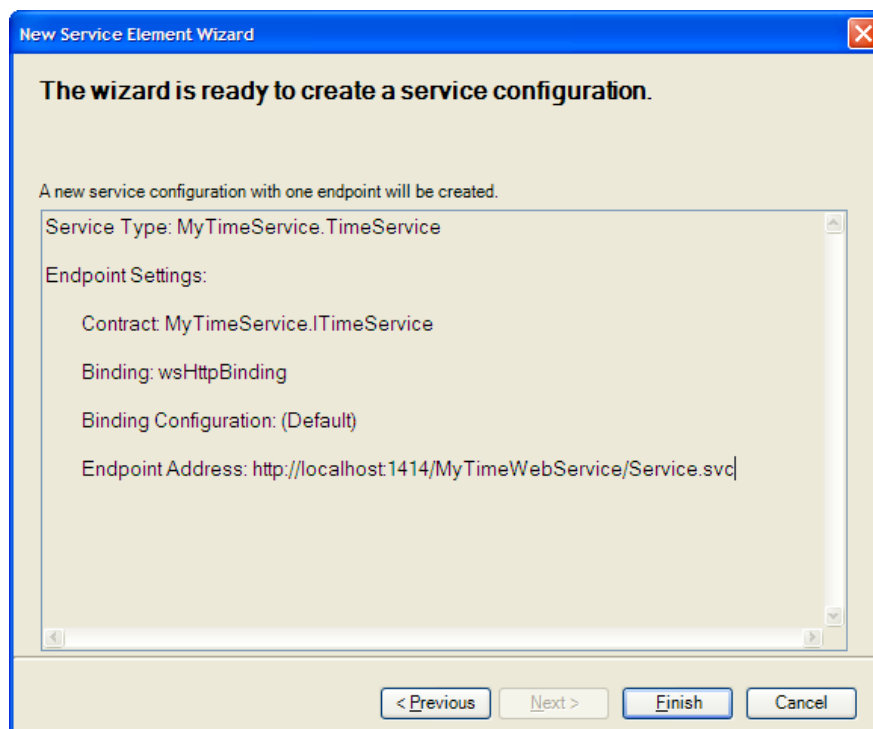
Choose the **Advanced Web Services interoperability** option. This option adds support for WS\* specifications (like WS-Security, MTOM, WS-MetadataExchange and WS-ReliableMessaging) to the project. Click **Next** to set the service's endpoint. See the next figure.



The image shows a Windows-style dialog box titled "New Service Element Wizard". The main heading is "What is the address of your endpoint?". Below this, it says "Specify the address for the endpoint:". There is a text input field labeled "Address:" containing the URL "http://localhost:1414/MyTimeWebService/Service.svc". At the bottom, there are four buttons: "< Previous", "Next >", "Finish", and "Cancel".

**Figure 21 – Specifying the web service’s endpoint.**

Enter the URL for the web service, followed by `Service.svc`. Click **Next** to see the final wizard summary screen, shown in the next figure.

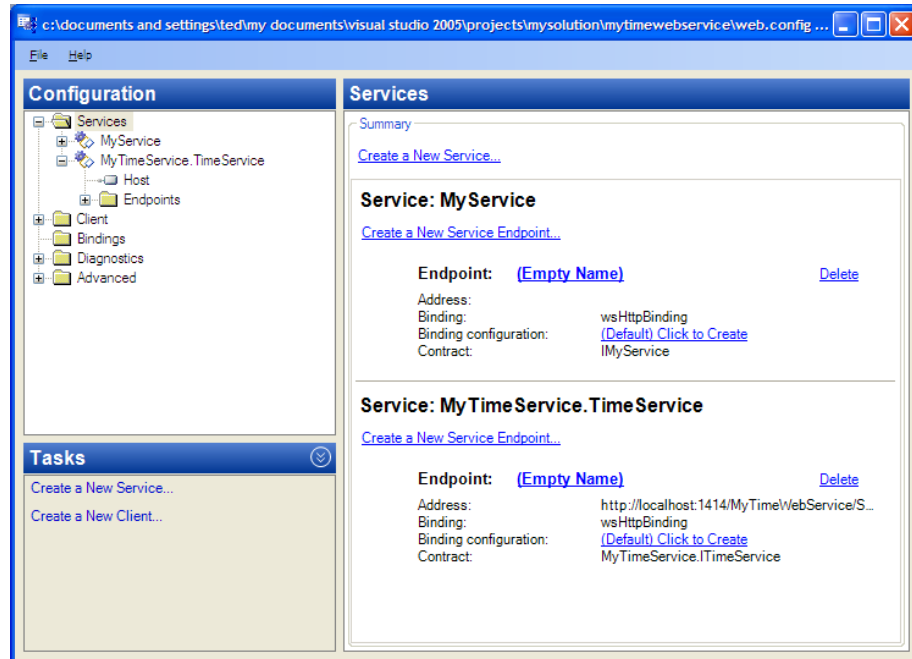


The image shows the same "New Service Element Wizard" dialog box, but at a later step. The main heading is "The wizard is ready to create a service configuration.". Below this, it says "A new service configuration with one endpoint will be created.". There is a scrollable text area containing the following information:  
Service Type: MyTimeService.TimeService  
Endpoint Settings:  
Contract: MyTimeService.ITimeService  
Binding: wsHttpBinding  
Binding Configuration: (Default)  
Endpoint Address: http://localhost:1414/MyTimeWebService/Service.svc  
At the bottom, there are four buttons: "< Previous", "Next >", "Finish", and "Cancel".



**Figure 22 – The final summary page of the service wizard.**

Click **Finish**. The service wizard will end and you'll be taken back to the **WCF Configuration wizard**. All the service settings will be shown, as seen in the next figure.

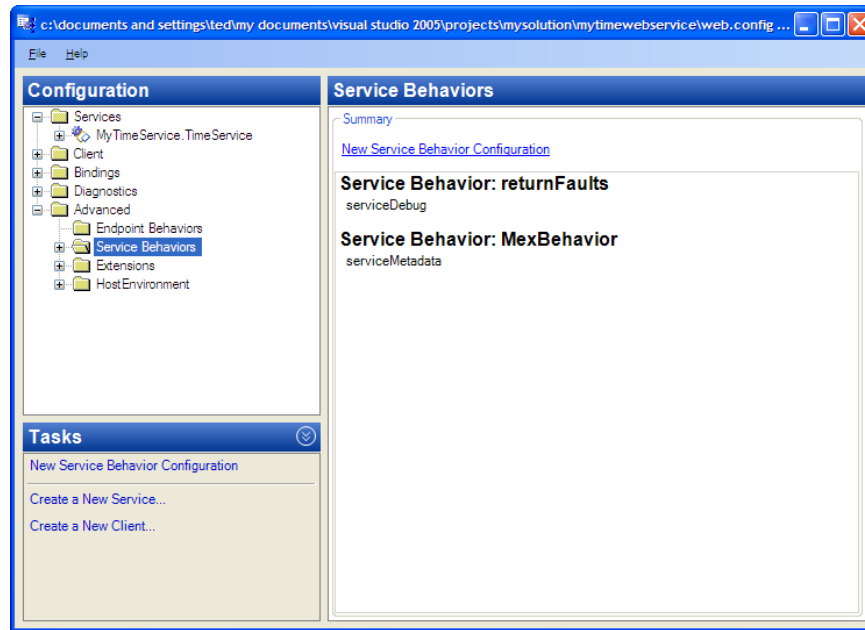


**Figure 23 – The WCF Configuration wizard, showing the configured service.**

You might want to save your work now. To do so, use the **File -> Save** menu or press Ctrl-S.

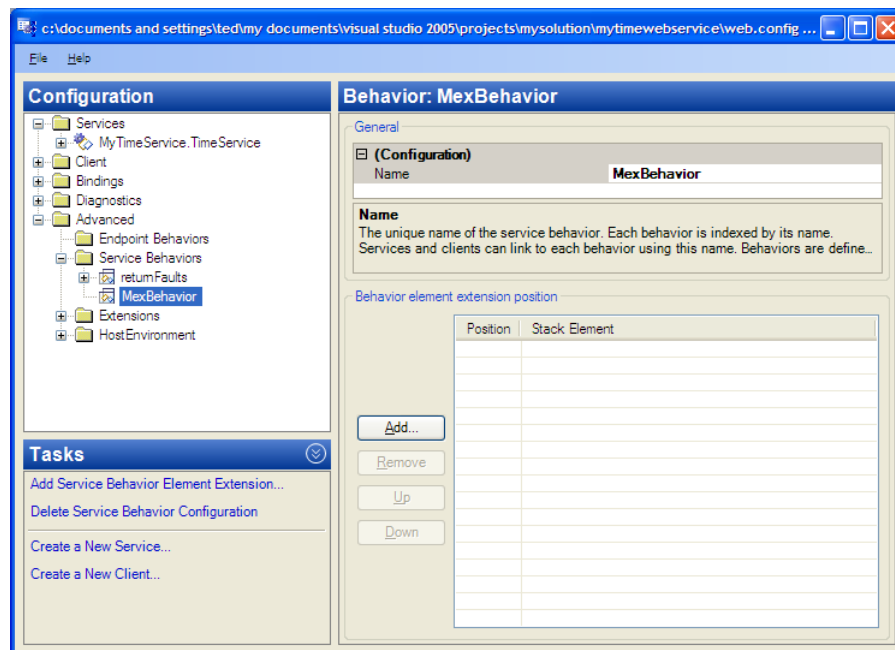
## Enabling Dynamic Retrieval of Metadata

When you create a client that accesses your service, you'll need to create a service proxy in the client project. In order to enable Visual Studio to setup service proxies automatically for clients, a service needs to make its metadata available through the web service. To achieve this, you need to add a WCF behavior specifying that the HTTP `Get` verb is supported for MetadataExchange. You can use the Visual Studio **WCF Configuration wizard** to add this behavior. Select the **Advanced -> Service Behaviors** folder in the **Configuration** pane, as shown in the next figure.



**Figure 24 – Creating a new service behavior for dynamic metadata retrieval.**

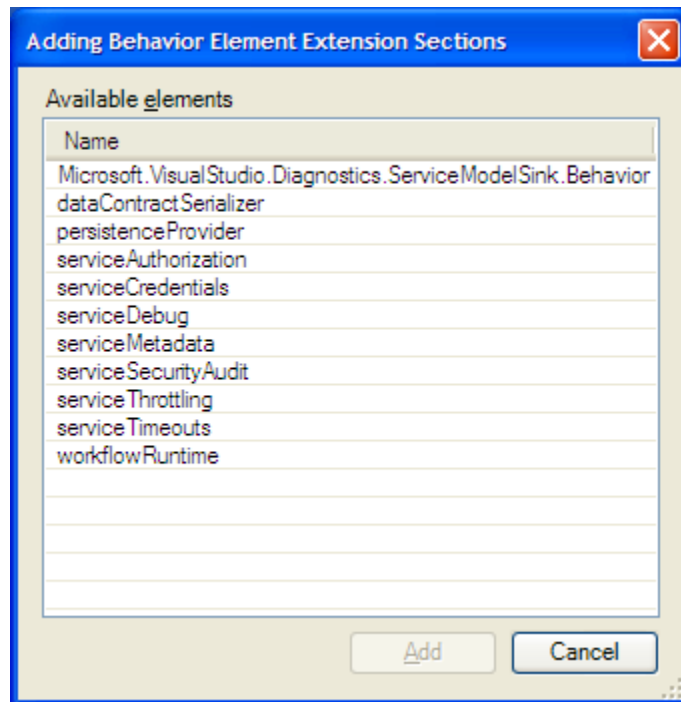
Click the **New Service Behavior Configuration** link in the **Tasks** pane. The **Behavior Configuration** pane will appear, as seen in the next figure.



**Figure 25 – The behavior configuration screen.**

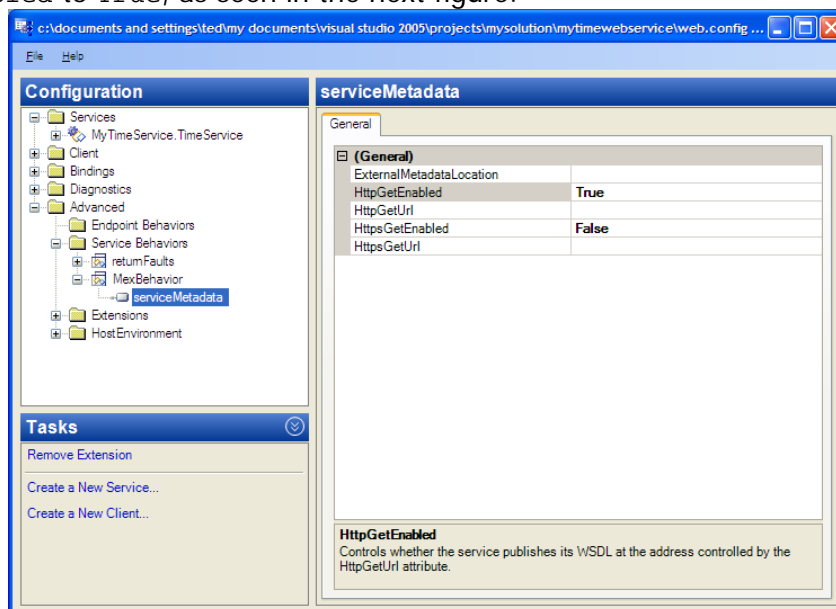
Like most of the configuration items in WCF, behaviors can be named. It's a good idea to name the behavior with something other than the default `NewBehavior` name. A clear name makes your configurations easier to understand. I used `MexBehavior` (Mex for

MetadataExchange). Click the **Add** button to display the list of behaviors that can be added at the service level, as shown in the next figure.



**Figure 26 – The behavior elements available.**

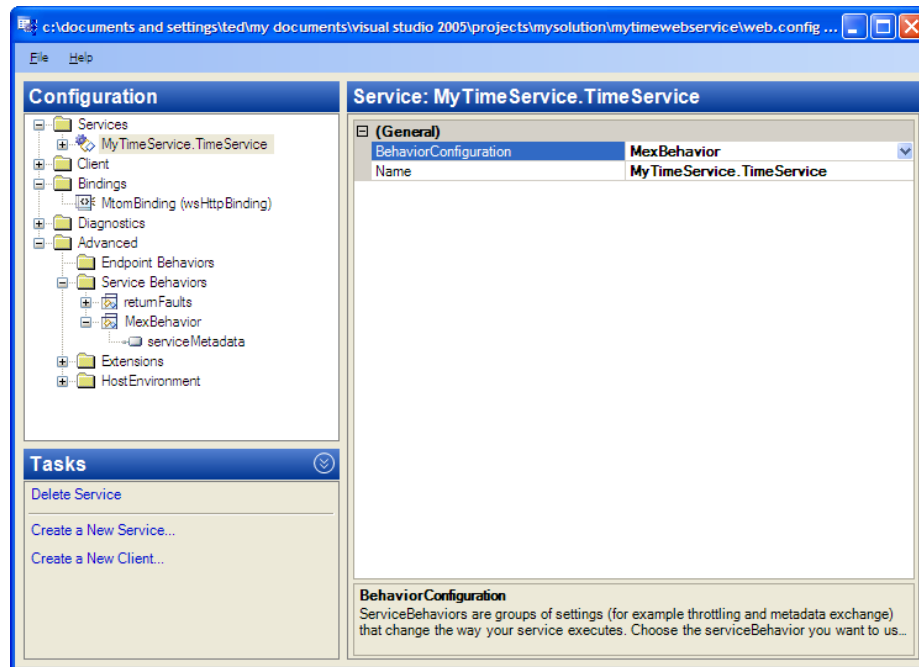
Select the element `serviceMetadata` and click **Add**. Back on the main wizard screen, double click the `serviceMetadata` element in the **Behavior** pane. You'll get see the properties for the `serviceMetadata` behavior element. Change the value of the `HttpGetEnabled` to `True`, as seen in the next figure.



**Figure 27 – Adding support for dynamic metadata retrieval.**

Changing the property value to `True` allows clients to use the HTTP `Get` verb to retrieve the service's metadata. Visual Studio uses this metadata to automatically setup client proxies, as described later.

So far we've only *created* a behavior element that supports dynamic metadata retrieval. To complete the task, we need to actually *use* the element in our WCF service. To do so, click on the **Services -> MyTimeService.TimeService** node in the **Configuration** pane. In the right pane, open the `BehaviorConfiguration` dropdown and select `MexBehavior`, as shown in the next figure.

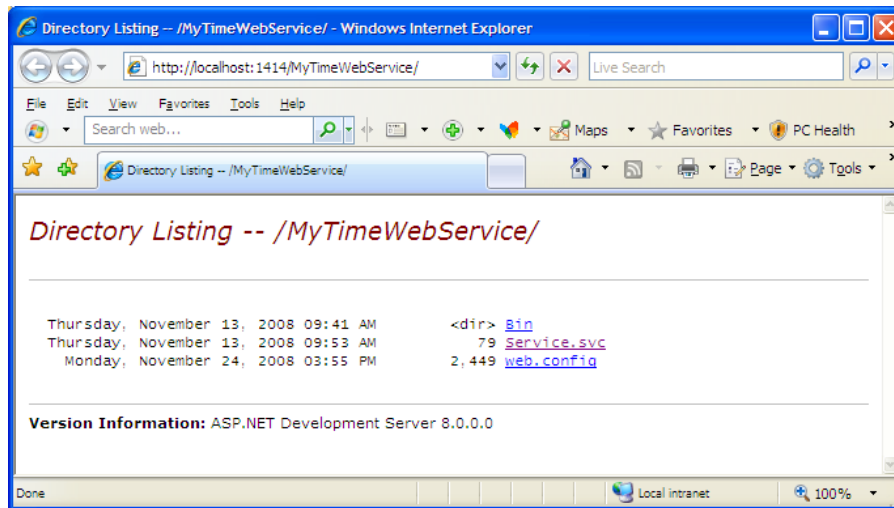


**Figure 28 – Using the newly created behavior in the WCF service.**

Now the service is fully configured to support dynamic client configuration. Select **File -> Save** to save the configuration file.

## Checking the WCF Configuration

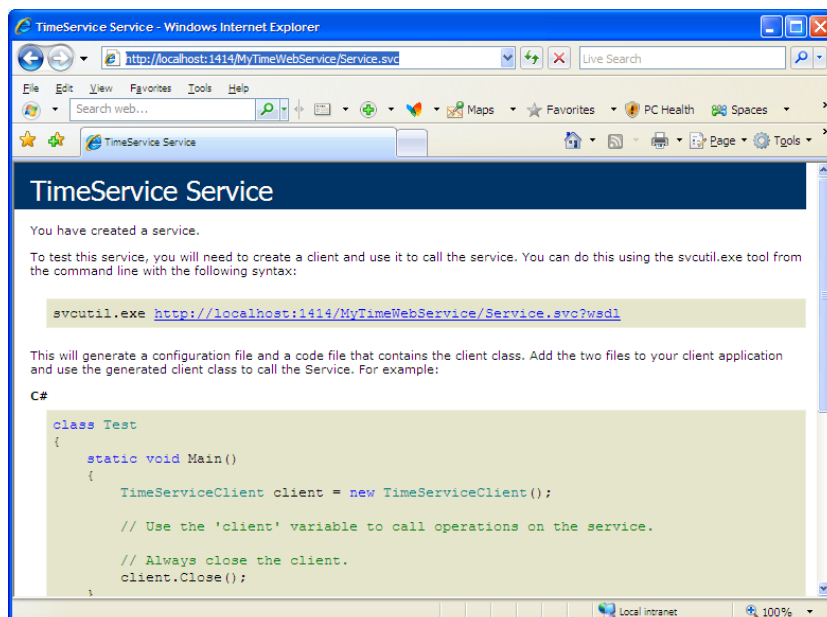
We've been through quite a few screens and changed many properties. If you're not very familiar with the overall process, it's easy to miss something. Before proceeding any further, it's a good idea to stop and ensure that everything is setup correctly. We need to check that our web service is running and responds correctly to queries. To do so, right click on the `MyTimeWebService` project in the Visual Studio Solution Explorer and select **View in Browser** from the context menu. You should get a browser window that shows the directory contents of the web service's root folder, as shown in the next figure.



**Figure 29 – The browser window shown by the View in Browser menu command.**

Notice the port number (1414) in the browser **Address Bar**. The actual port number on your system may be different. The value is set dynamically by Visual Studio, when you create a web project.

Now click on the Service.svc link in the browser window. If everything is configured correctly, the browser should bring up a web page that describes the service, similar to the one in the next figure.



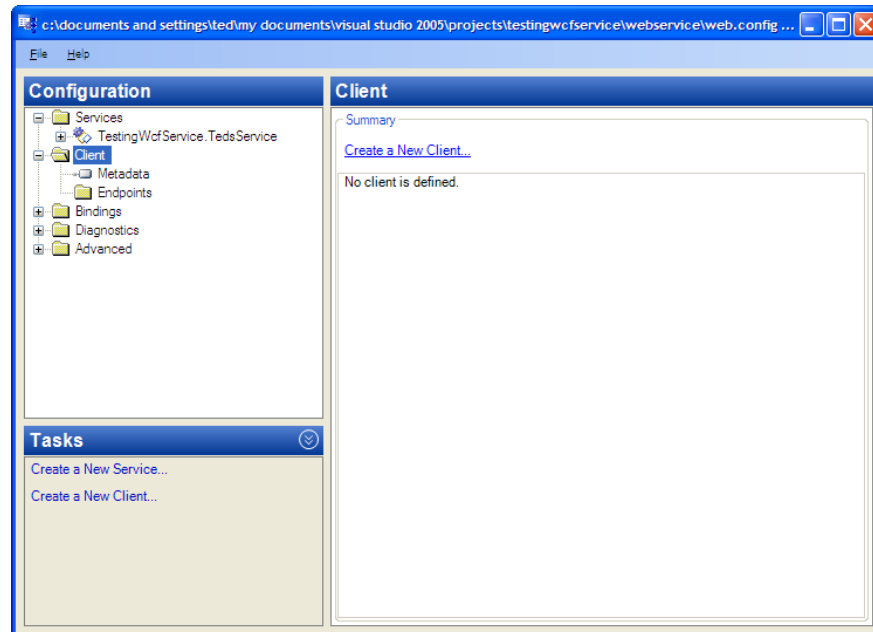
**Figure 30 – Testing the configuration of the WCF service..**

If you don't get this page, you'll have to go back and check your steps. You must have missed something. It is futile to continue any further until you get the right screen here.

## Setting up the Client Configuration

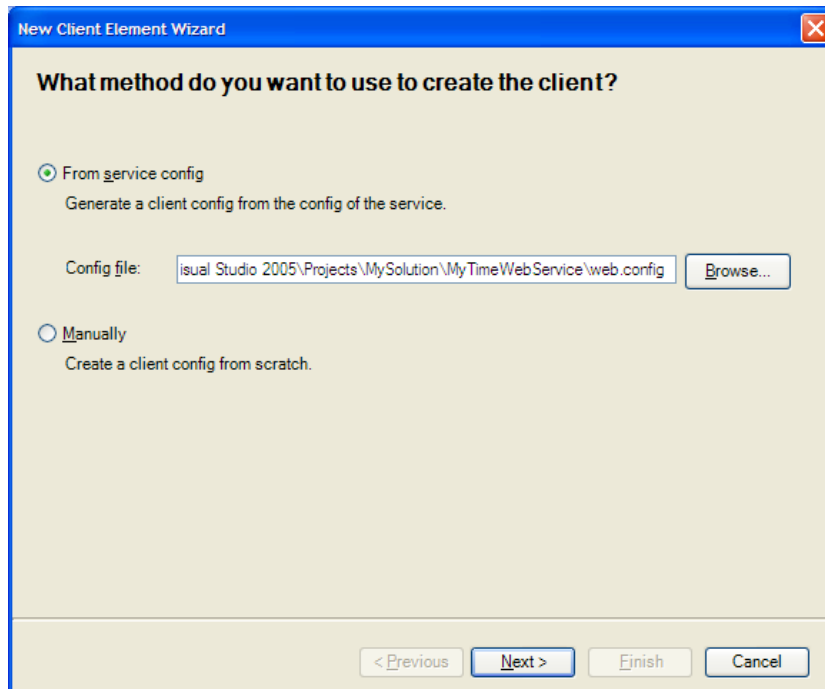
Now that the service is setup correctly, you can create configuration settings for future clients. Here we only create configuration settings; we don't actually create a client project. When we create a client project later on, it will use the client configuration settings we will setup now.

Let's get started. Using the **WCF Configuration wizard**, select the **Client** folder in the **Configuration** pane, as shown in the next figure.



**Figure 31 – The Client folder in the WCF Configuration wizard.**

Click **Create a New Client** in the **Tasks** pane. You'll get the **New Client Element Wizard** screen, shown in the next figure.

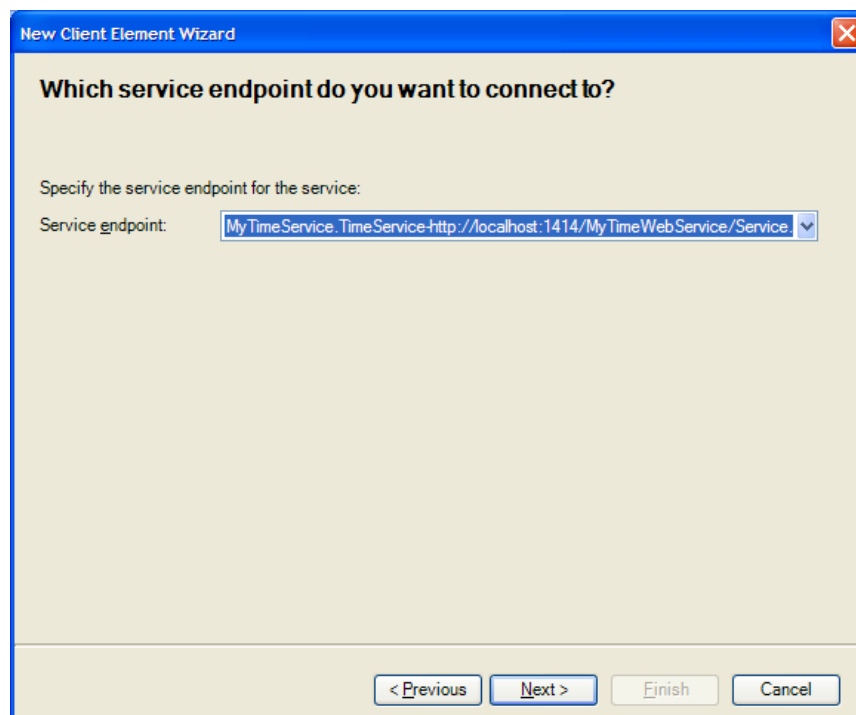


The image shows the first screen of the 'New Client Element Wizard' dialog box. The title bar reads 'New Client Element Wizard'. The main heading is 'What method do you want to use to create the client?'. There are two radio button options: 'From service config' (which is selected) and 'Manually'. Below 'From service config' is the text 'Generate a client config from the config of the service.' Below 'Manually' is the text 'Create a client config from scratch.' Under the 'From service config' option, there is a text box labeled 'Config file:' containing the path 'isual Studio 2005\Projects\MySolution\MyTimeWebService\web.config'. To the right of this text box is a 'Browse...' button. At the bottom of the dialog are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

**Figure 32 – The first screen of the New Client Element Wizard..**

Make sure the **From service config** radio button is checked. This tells the wizard to use all the settings that we created for the service as the basis for the client.

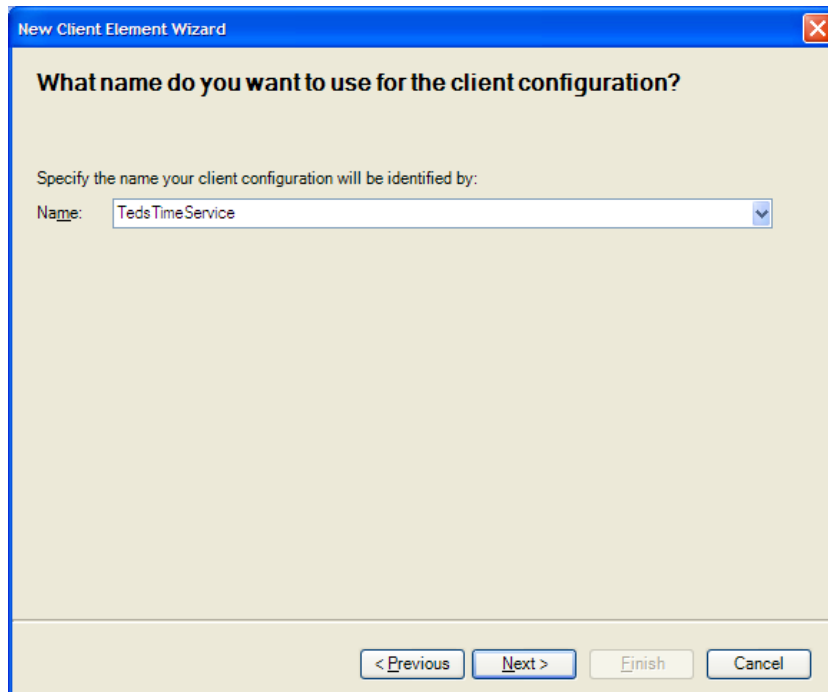
Enter the path to the web service's `web.config` file. Click **Next**. On the next screen, select `MySolution/MyTimeWebService/Service.svc` from the **Service endpoint** dropdown.



The image shows the second screen of the 'New Client Element Wizard' dialog box. The title bar reads 'New Client Element Wizard'. The main heading is 'Which service endpoint do you want to connect to?'. Below this is the text 'Specify the service endpoint for the service:'. There is a dropdown menu labeled 'Service endpoint:' with the selected value 'MyTimeService.TimeService-http://localhost:1414/MyTimeWebService/Service.svc'. At the bottom of the dialog are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

**Figure 33 – Specifying the service endpoint for the client configuration.**

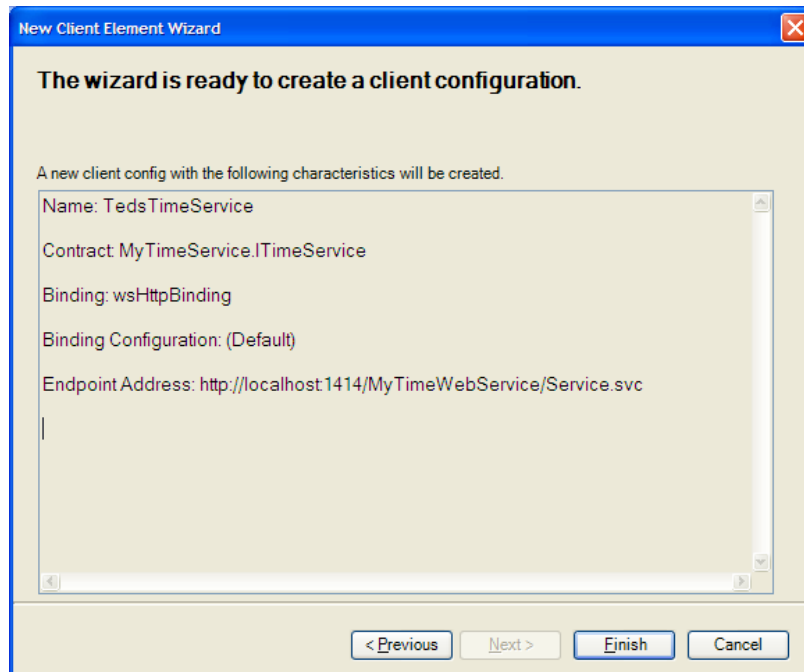
Click **Next**. Enter a name for the client configuration. This is the name that will be used by default by clients for the web service proxy. I entered **TedsTimeService**, as seen in the next figure. The name can't contain embedded spaces. If you include spaces, you'll get a runtime error when someone tries to access the service.

A screenshot of a Windows-style dialog box titled "New Client Element Wizard". The dialog has a blue title bar with a close button in the top right corner. The main area has a light beige background. At the top, it asks "What name do you want to use for the client configuration?". Below this, it says "Specify the name your client configuration will be identified by:". There is a text input field labeled "Name:" containing the text "TedsTimeService". At the bottom of the dialog, there are four buttons: "< Previous", "Next >", "Finish", and "Cancel". The "Next >" button is highlighted with a blue border.

**Figure 34 – Giving a name to the client configuration.**

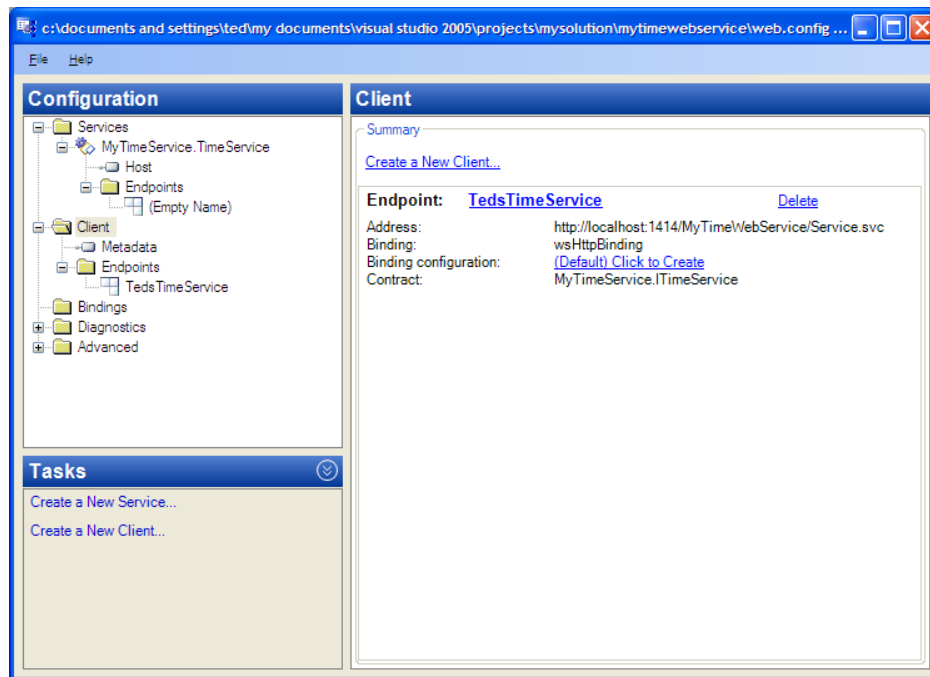
Click **Next** and you'll see the final configuration screen of the wizard, as shown in the next figure.





**Figure 35 – The summary screen showing the client configuration settings.**

Click **Finish**. Having setup the basic client configuration, we need to specify how clients will connect to the service. The *how* part is defined by WCF *binding* element. To create a default binding, select the **Client** node in the **Configuration** pane. The right pane will show the **Client** configuration, as shown in the next figure.



**Figure 36 – The initial client configuration settings.**

In the right pane, click on the **(Default) Click to Create** link. You'll see the initial bindings for the client, shown in the next figure.

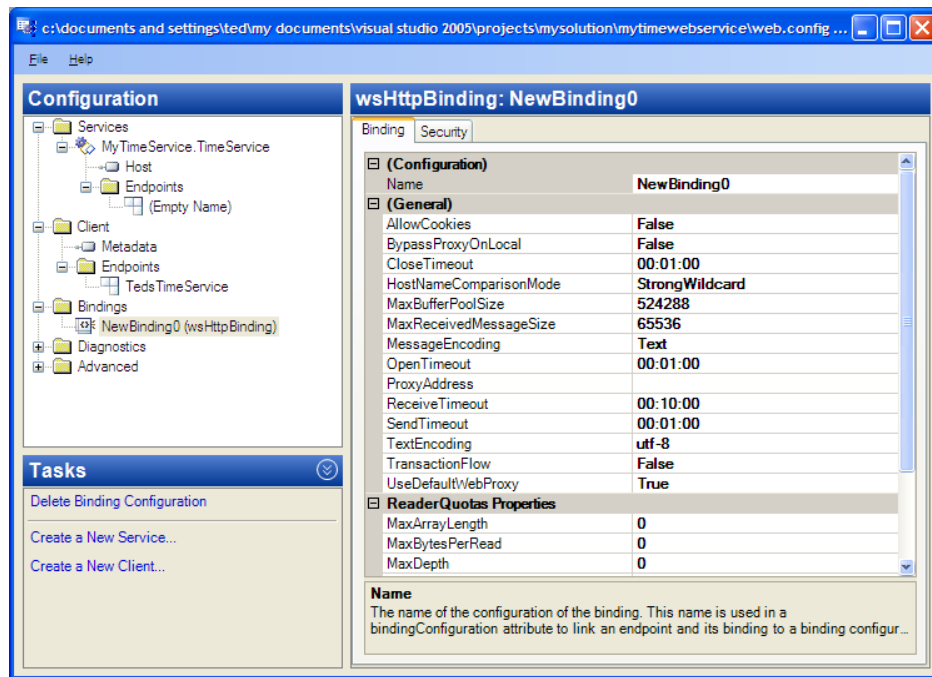


Figure 37 – The initial client configuration settings.

For many situations, the default binding properties will be fine. In the next section, we'll make some changes to support binary attachments. As you can see in the previous figure, you can change all sorts of things that determine how the binding will work. If you don't need to support binary attachments in your web service, skip the next section.

## Adding MTOM Support

If you want to support large attachments, MTOM (Message Transmission Optimization Mechanism) is the way to go. MTOM is a world standard for sending attachments to web service. To enable MTOM, use the **MessageEncoding** property, shown in the previous figure. Change its value from `Text` to `Mtom`. If you expect any web service parameters or return values to exceed the default of 64 KB, change the following binding properties to meet your needs:

- `MaxReceivedMessageSize`
- `MaxArrayLength`
- `MaxBytesPerRead`

Bindings can have a name. We'll call ours `MtomBinding`. To do so, edit the **Name** property at the top of the right pane, changing the value from `NewBinding0` to `MtomBinding`. The next figure shows the binding properties after making changes.

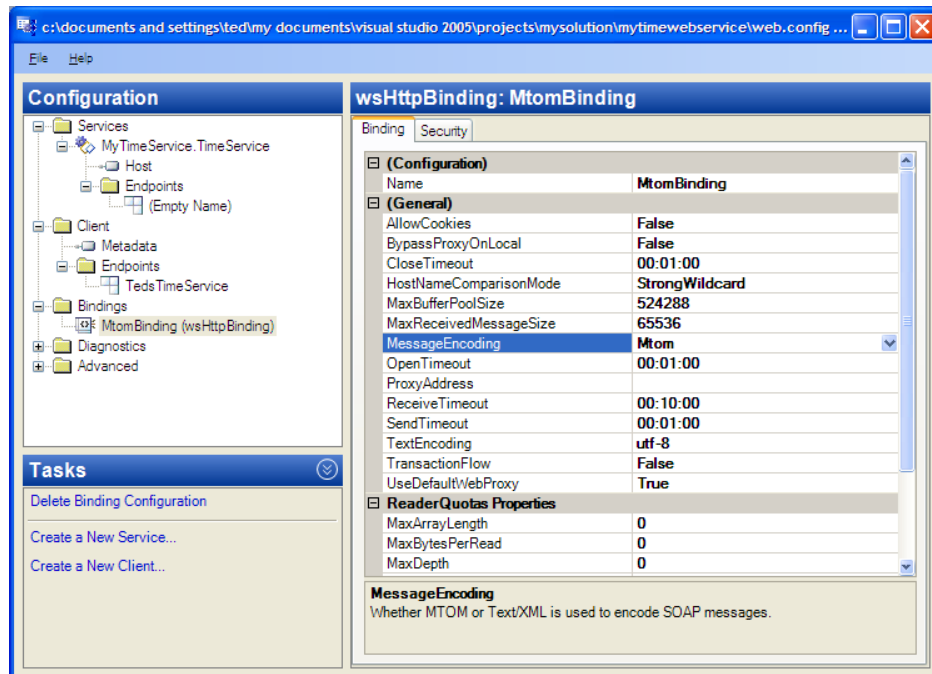
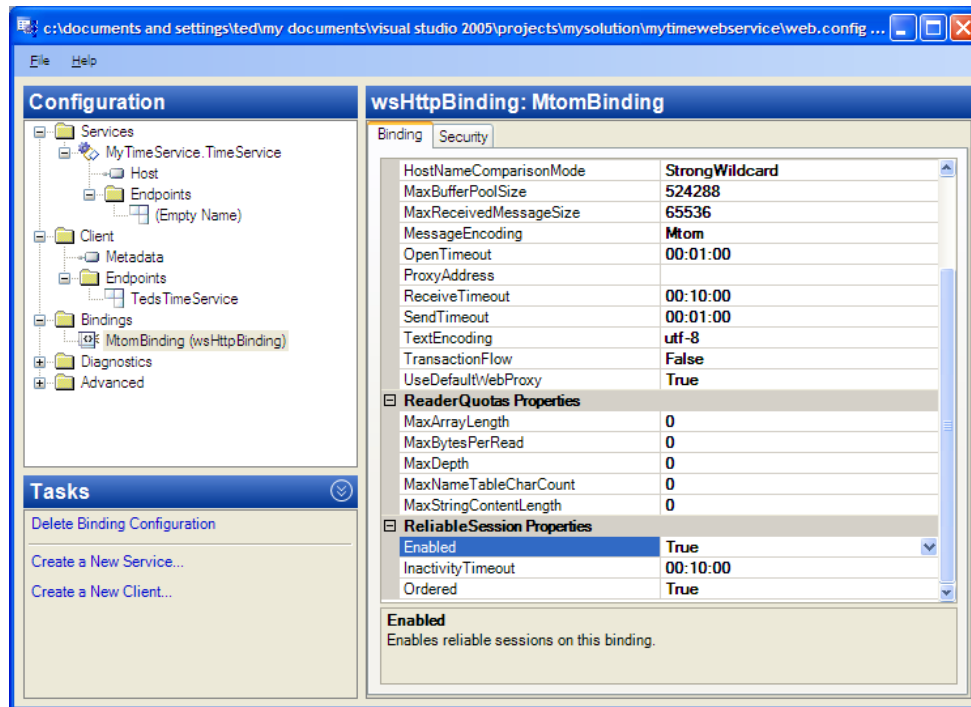


Figure 38 – Adding MTOM support to a binding.

## Adding Reliable Messaging Support

When building distributed systems that rely on network messages, often you need to ensure that messages reach their destination. Lost messages might cause the system to perform incorrectly. Duplicated messages, or messages that arrive out-of-order, might also cause problems. HTTP, via TCP, ensures that messages get to the destination computer (assuming network connectivity is available). But TCP only verifies delivery at the socket level. If the destination computer dies after receiving the message, but before processing it, your distributed system may not function correctly. WS-ReliableMessaging is a WS\* specification to solve this problem. WCF supports a number of WS\* specs, including WS-ReliableMessaging. The **WCF Configuration wizard** makes it very easy to add reliability to a system. Scroll down in the right pane to see the **ReliableSession Properties** section. Set the **Enabled** property to `True`, as shown in the next figure.

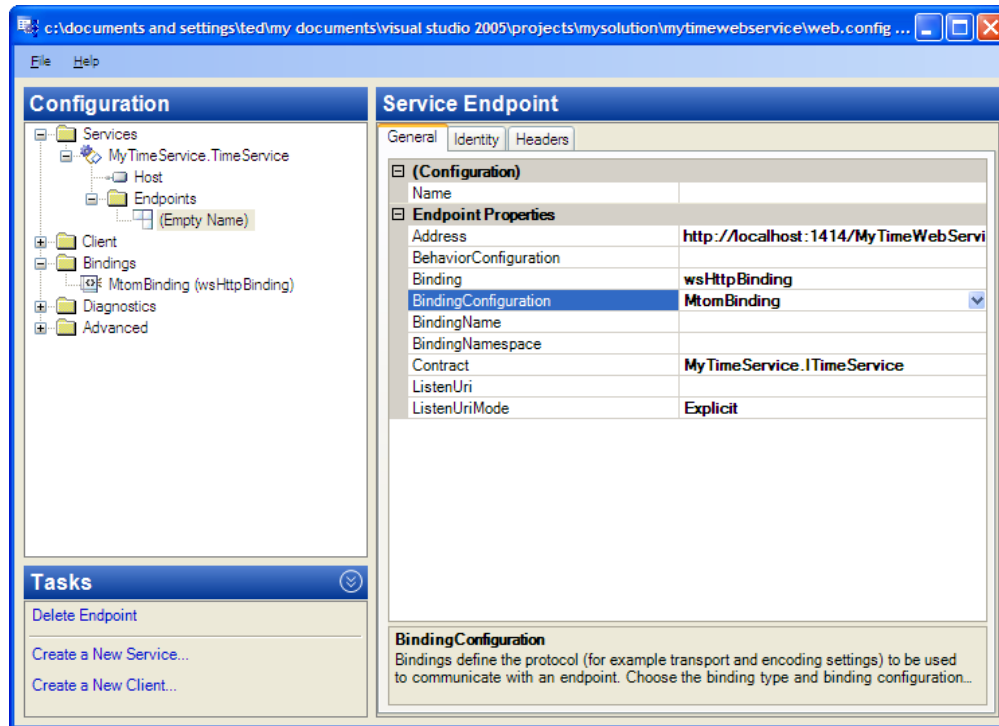


**Figure 39 – Adding ReliableMessaging support.**

To implement reliability at the message level, a system must use sessions, in which messages are numbered sequentially. WCF uses the expression **Reliable Session** for sessions that support WS-ReliableMessaging. By default, messages are delivered to the end application in the same order in which they are sent.

Note: To deliver a received message to an application, WCF calls your application handler method. After the method processes the message and returns control to WCF, an acknowledgement message is returned to the sender. The sender is now certain that the message sent was actually handled by the recipient system. If that system had crashed before or during processing of the message, no acknowledgement would have been returned to the sender. In this case, WCF on the sender side would automatically try to resend the message -- up to a certain number of times.

The last step in configuring reliability for our service is to associate the new binding with the service's endpoint. In the **Configuration** pane, select the node **Services -> MyTimeService.TimeService -> Endpoints -> (Empty Name)**. In the right pane, open the dropdown for the **BindingConfiguration** property and select **MtomBinding**.



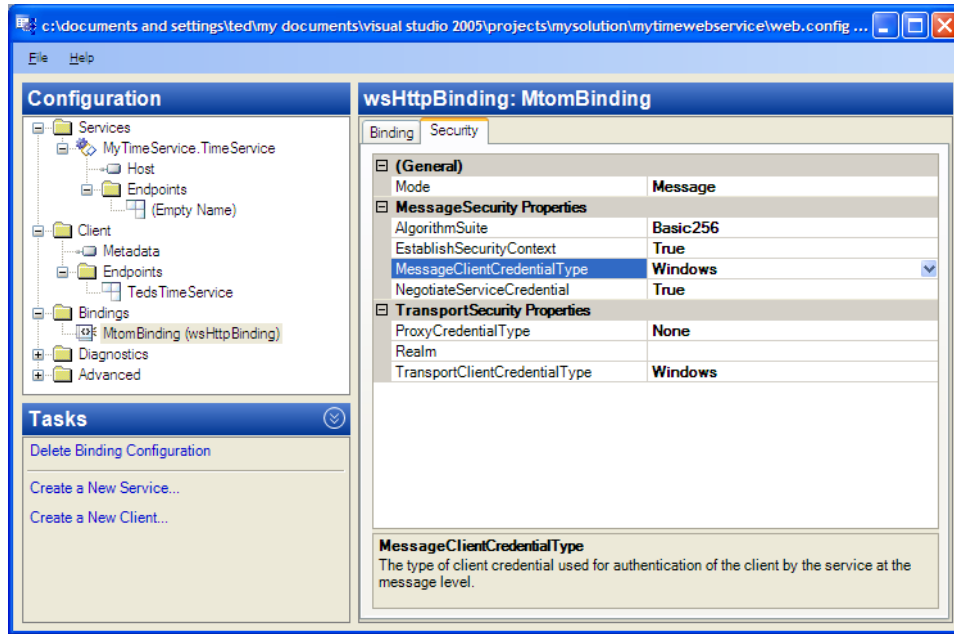
**Figure 40 – Adding Security support.**

Save the configuration by choosing **File -> Save**.

## Adding Security Support

When a distributed system sends confidential information over the wire, security is important. WCF implements security through authentication and encryption. The **WCF Configuration wizard** makes it very easy to setup the security settings for a service.

In WCF, security is applied at the level of a binding. Back in Figure 39 you can see that the right pane has two tabs: **Binding** and **Security**. To enable security, select the **Security** tab and set the **Mode** property, as shown in the next figure.



**Figure 41 – Adding Security support.**

For the Mode property, there are 4 choices:

- *None*. No security is needed.
- *Transport*. Encryption is handled by the transport. Select this option if you plan to use HTTPS. No user credentials are included with messages.
- *Message*. Encryption is handled at the message level, so you could send messages safely even over an HTTP connection. User credentials are included with messages, but everything is encrypted.
- *TransportWithMessageCredential*. Encryption is handled by the transport, but messages also include user credentials.

In the **Message Security Properties** section, in the right pane in the previous figure, you can configure which encryption algorithm will be used at the message level. The algorithm is only used if you set **Mode** to **Message**.

The other properties in the right pane allow you to set up what type of credentials the client will need to provide (a username/password, an X.509 security certificate, Windows credentials or a security token issued by a trusted service).

Save the configuration by choosing **File -> Save**.

## Creating the Client Project

The last piece in the puzzle is the web service client. I'll keep things simple and just create a console application. Right-click on the `MySolution` node in the Visual Studio Solution Explorer and select **Add New Project** from the context menu. In the **Add New Project** dialog, select **Console Application**. Enter a name and location and then click OK.

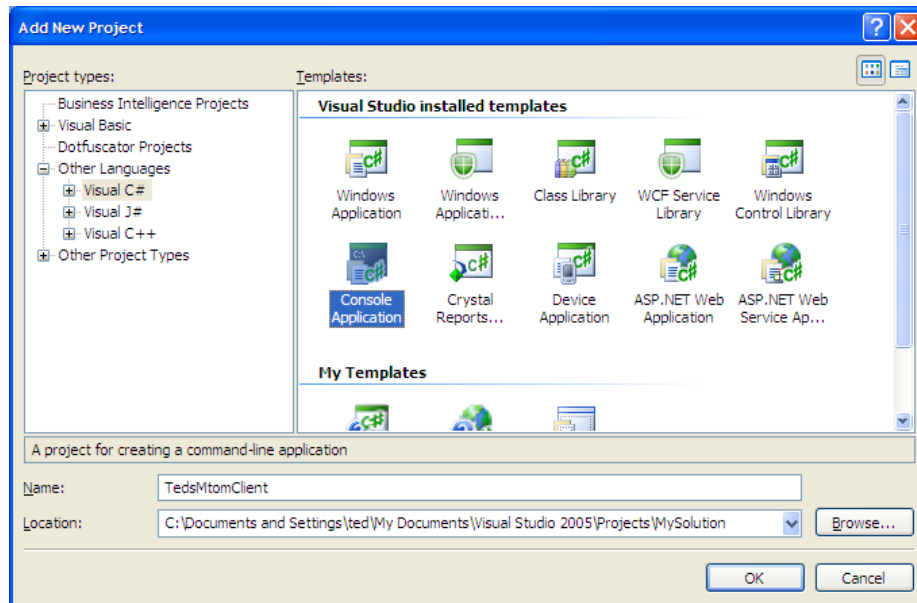


Figure 42 – a Service Reference to client.

In order for the client to be able to call methods of the web service, it must have a Service Reference, which is essentially just a proxy through which web service calls go. To create this proxy, go to the Solution Explorer, right click on the `TedsMtomClient` project and select **Add Service Reference** on the context menu. On the **Add Service Reference** dialog, enter the URI of the service and provide a name for the generated proxy.

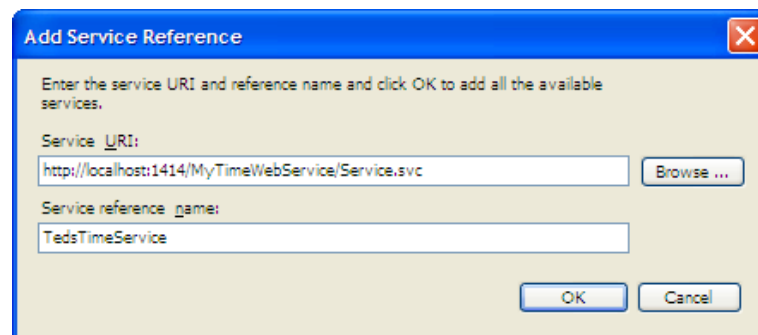
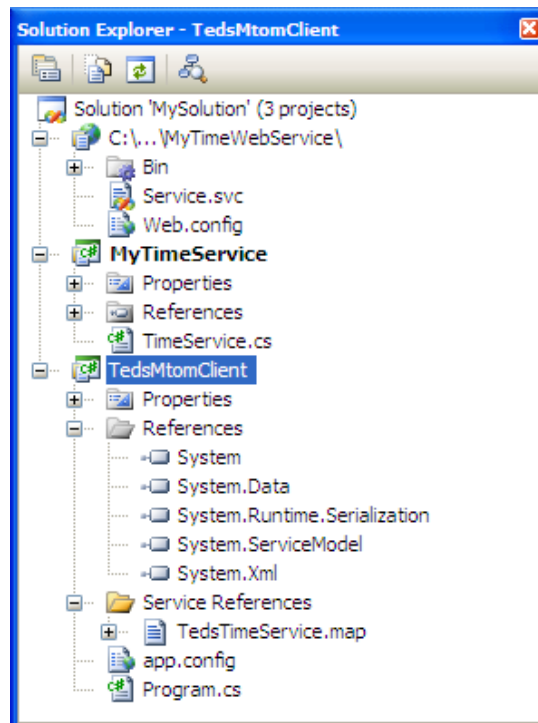


Figure 43 – Adding a Service Reference to a client.

Click the **OK** button. Visual Studio will go and retrieve the metadata from the service and build a local proxy. When all this is done, the Visual Studio Output window will have the following text:

```
C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\svcutil.exe
/noLogo /s /d:"C:\Documents and Settings\ted\Local
Settings\Temp\lfib0uwk.gw4" /config:"C:\Documents and
Settings\ted\Local Settings\Temp\lfib0uwk.gw4\newapp.config"
/mergeConfig /out:"TedsTimeService.cs" /language:csharp
/n:*,TedsMtomClient.TedsTimeService
"http://localhost:1414/MyTimeWebService/Service.svc"
Attempting to download metadata from
'http://localhost:1414/MyTimeWebService/Service.svc' using WS-
Metadata Exchange or DISCO.
Generating files...
C:\Documents and Settings\ted\Local
Settings\Temp\lfib0uwk.gw4\TedsTimeService.cs
C:\Documents and Settings\ted\Local
Settings\Temp\lfib0uwk.gw4\newapp.config
```

A new Service Reference will be created and added to the client project. Visual Studio will also add `System.Runtime.Serialization` and `System.ServiceModel` to the client project references. The Solution Explorer will look like this:



**Figure 44 – The references added by Visual Studio to the client project.**

An `app.config` file is also added to the client project. The config file is derived from the service's config file and looks like this:



```

<?xml version="1.0" encoding="utf-8" ?>
- <configuration>
-   <system.serviceModel>
-       <bindings>
-           <wsHttpBinding>
-               <binding name="WSHttpBinding_ITimeService" closeTimeout="00:01:00"
openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
bypassProxyOnLocal="false" transactionFlow="false"
hostNameComparisonMode="StrongWildcard" maxBufferPoolSize="524288"
maxReceivedMessageSize="65536" messageEncoding="Mtom" textEncoding="utf-8"
useDefaultWebProxy="true" allowCookies="false">
<readerQuotas maxDepth="32" maxStringContentLength="8192"
maxArrayLength="16384" maxBytesPerRead="4096"
maxNameTableCharCount="16384" />
<reliableSession ordered="true" inactivityTimeout="00:10:00" enabled="false" />
-               <security mode="Message">
<transport clientCredentialType="Windows" proxyCredentialType="None"
realm="" />
<message clientCredentialType="Windows" negotiateServiceCredential="true"
algorithmSuite="Default" establishSecurityContext="true" />
</security>
</binding>
</wsHttpBinding>
</bindings>
-       <client>
-           <endpoint address="http://localhost:1414/MyTimeWebService/Service.svc"
binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_ITimeService"
contract="TedsMtomClient.TedsTimeService.ITimeService"
name="WSHttpBinding_ITimeService">
-               <identity>
<userPrincipalName value="ted@AgilityVM.Local" />
</identity>
</endpoint>
</client>
</system.serviceModel>
</configuration>

```

**Listing 4 - The client app.config file created by Visual Studio.**

The client config file will use the same MTOM and security settings as the service. If you later decide to change the MTOM or security settings, you'll need to change both the client and the service configurations. To test the service, you'll need to add some code to the default Program.cs file of client project. The following listing shows an example.

```

using System;
using System.Collections.Generic;
using System.Text;

using TedsMtomClient.TedsTimeService;

namespace TedsMtomClient
{
    class Program

```

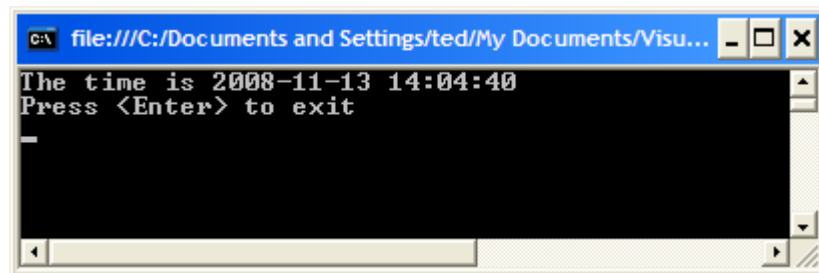
```

{
    static void Main(string[] args)
    {
        TimeServiceClient timeService = new TimeServiceClient();
        DateTime currentTime = timeService.GetCurrentTime();
        Console.WriteLine(string.Format(
            "The time is {0:yyyy-MM-dd HH:mm:ss}",
            currentTime));
        Console.WriteLine("Press <Enter> to exit");
        Console.ReadLine();
    }
}

```

**Listing 5 – A simple client class to test the WCF service.**

Run the client program. If everything works correctly, the `TimeService` service should be instantiated and hosted automatically by `MyTimeWebService`. The following shows the Command Prompt window when running the client.



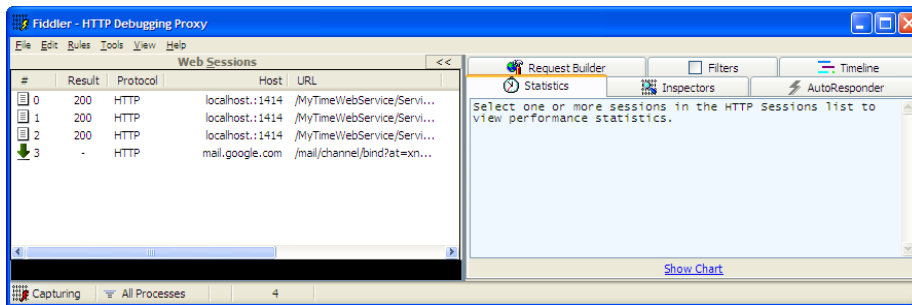
**Figure 45 – The output of a simple client console app.**

## Viewing the HTTP Traffic between client and service

It's great when everything works the first time, but in real life things don't always go as expected. That's when you need support tools to help you figure out what's really going on. When using WCF, there are so many ways to configure the system that it is extremely easy to get something wrong. When your system doesn't seem to work correctly, you often need to see the messages exchanged by client and service. How? One way is to use a tool called **WCF Service Trace Viewer**, which ships with the .Net Framework 3.0 SDK. This is a fairly substantial tool that would require a separate article to cover.

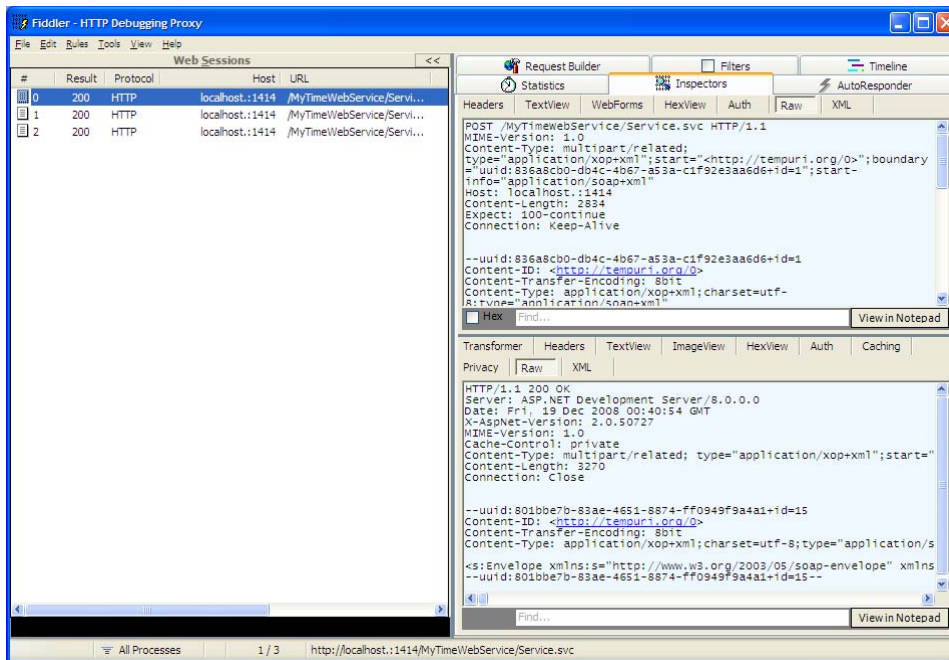
For many situations, an HTTP-capture tool is easier to use and more appropriate. I'll show how to use a freely available HTTP proxy tool called **Fiddler** to capture and show all the HTTP traffic between a client and server. Fiddler installs itself as a web proxy, so all HTTP traffic going to and from the internet on your machine goes through it.

The following figure shows the Fiddler UI after running the Client program.



**Figure 46 – Using Fiddle to inspect the HTTP traffic between our WCF client and service.**

In the left pane Fiddler shows the HTTP traffic. When you select an item, the right pane shows you the details about the associated request and response. The HTTP request is in the upper portion and the HTTP response in the lower portion of the right pane, as shown in the following figure.



**Figure 47 – The HTTP request and response shown in the Fiddler's right pane.**

Both the upper and lower portions of the right pane contain a bunch of tabs. Each tab basically allows you to look at different views of a request or response. The **Raw** view shows the raw characters of the headers and content. I won't describe Fiddler in further detail here. For more information see:

<http://www.fiddler2.com/>

One issue when using an HTTP traffic sniffing tool on a Windows machine is capturing messages sent to the special TCP local loopback address called *localhost*. When sending messages to this address, the Windows HTTP stack doesn't actually pump data all the way

down to the IP layer, where it would appear on the network. Microsoft did things this way to increase performance. The problem with this shortcut is that it prevents network analyzer tools like Wireshark and Fiddler from capturing localhost traffic.

To solve the localhost problem, Fiddler resorts to a simple stratagem. In your application program, wherever you have a URL that uses localhost, replace "localhost" with "localhost.". Notice the dot after the word "localhost". Any messages sent to the host "localhost." will be intercepted by Fiddler. If you look carefully at the messages in the left pane in the previous figure, you'll notice that the URLs have the dot after "localhost". Once you're done testing with Fiddler, make sure to set the localhost URLs back, by removing the dot.

## Conclusion

Windows Communication Foundation is a truly magnificent piece of work – on par with the .Net Framework itself. WCF handles all the heavy lifting, in terms of support for the latest web services standards. In doing so, WCF allows you to spend more time on your application logic and less on communications-related details. Still, you need to know what the various WS\* standards support, because you need to configure WCF to leverage them. WCF configuration is not at all trivial, but the **WCF Configuration wizard** handles many of the details, sparing you from having to master the intricacies of the xml configuration files that WCF relies on. There is little online information about the wizard, so many developers are unaware of its power and usefulness. Hopefully this article will encourage you to use the wizard, handling WCF configuration the easy way.