IN THIS CHAPTER

- Becoming acquainted with WCF
- Building a WCF service
- Accessing a new service

Online Chapter 6

# Building Web Services with WCF

Windows Communication Foundation (WCF) is just that — the foundation for communication between Windows computers. It just so happens that thanks to open standards like Simple Object Access Protocol (SOAP) and REpresentational State Transfer (REST), WCF can communicate with other software systems (even non-Windows systems).

Although ASMX was really designed to make public services — such as adding an API to a simple web application — WCF is a complete distributed computing platform for Windows.

In the early days of .NET, there was a technology called .NET Remoting that replaced DCOM. DCOM was Distributed Component Object Model, or the commonly accepted way to communicate between distributed components. It was replaced by Remoting when .NET came out. Remoting basically took the principles of DCOM and migrated them to .NET.

WCF isn't like that. WCF is a result of a complete rethinking of distributed computing, based on the understanding that computing is becoming more and more distributed. New protocols for communication come out every day.

The goal here, then, is to define the differences between ASMX and WCF, and see that WCF is a true communications protocol and that ASMX is used solely for adding services to websites. You can use either technology for both tasks, but one is certainly more suited than the other for each. This chapter looks at why WCF works well and then builds a service using it.

# Getting to Know WCF

First there was DCOM. Then there was .NET Remoting. The path to a distributed computing platform for Microsoft has been a long one. Distributed computing is a hard problem to solve, and Microsoft developers continue to hone their platform.

There is SOAP, there are Microsoft binary formats, people are creating custom HTTP contexts — the distributed computing platform is a mess. Something needed to happen to enable you to all take your existing code and make it available across the enterprise.

The designers of WCF (largely Doug Purdy, Don Box, and crew) had two diverse issues. On one hand was ASMX, providing SOAP web service access to business logic. On the other hand was .NET Remoting, providing Microsoft with a custom, black-box format for the transmission of information between components via known network protocols. Online Chapter 5 discusses ASMX and its limitations. Taking a deeper look at Remoting makes the case for a comprehensive platform even clearer (see Figure OC6-1).

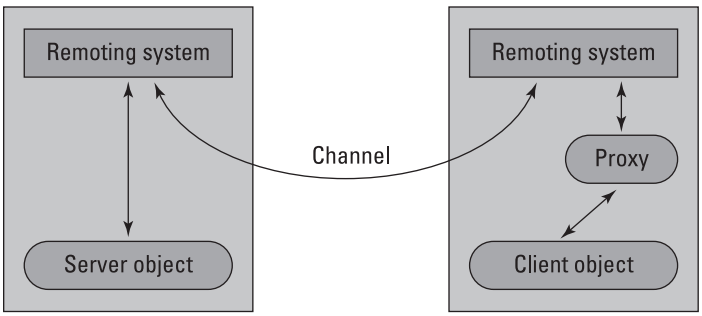


**FIGURE OC6-1:** The Remoting system.

If each of two systems — say, a Client System and a Server System — had a similarly configured remoting system, they could communicate. The problem, of course, is that the remoting systems were never, ever configured correctly (or so it seemed). One side of the equation or the other will make some change, and the

whole system will come to a crashing halt. Clearly, there has to be some product that brings the various formats of remote access together under one umbrella — something that would accept one block of logic and provide multiple service types.

Eventually, WCF was that solution. Starting as an add-on to .NET 2.0, WCF effectively enabled developers to make specific endpoints for a generic connector. Figure OC6-2 shows the original problem illustrated by David Chappel for Microsoft back in 2007.

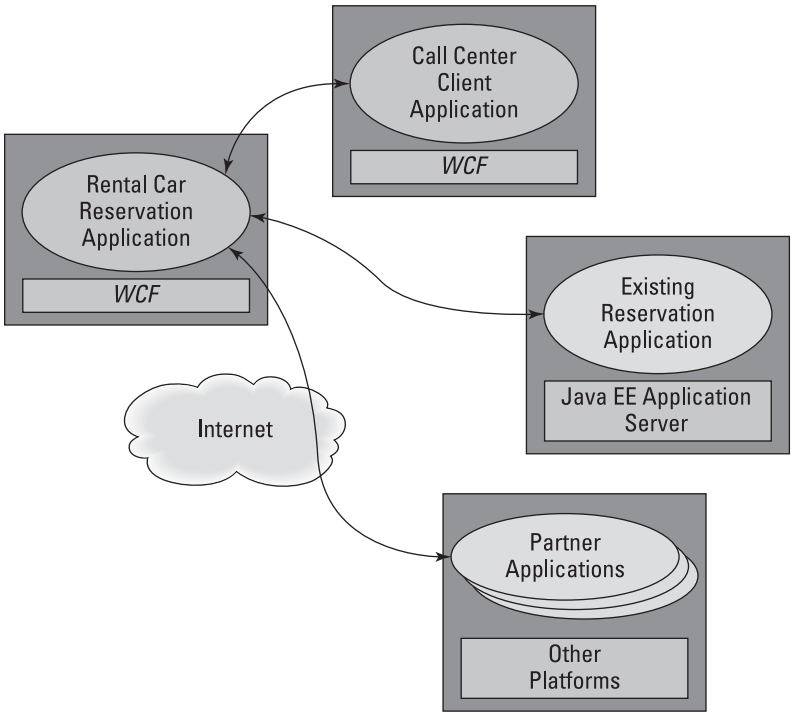


**FIGURE OC6-2:** The original problem to be solved.

A car dealership is trying to build a new reservation application. The business logic needs to be both accessible to outside applications and provide a quality binary transport format for the internal communication. WCF is the answer. It uses configuration to provide various *endpoints* to consuming applications, from SOAP to REST to binary associations that resemble DCOM. It doesn't require configuration on both ends of the pipe, only on the server side. If the server serves it, the client can consume it.

# Creating a WCF Service

As with so much in working with the .NET Framework, creating a WCF service isn't as much about the code as it is about the configuration. The WCF example service starts out much like an ASMX service:

1. **Choose File➪New➪Project.**

   You see the New Project dialog box.

2. **Select the Visual C#\WCF folder.**

   Visual Studio displays the list of projects, as shown in Figure OC6-3.

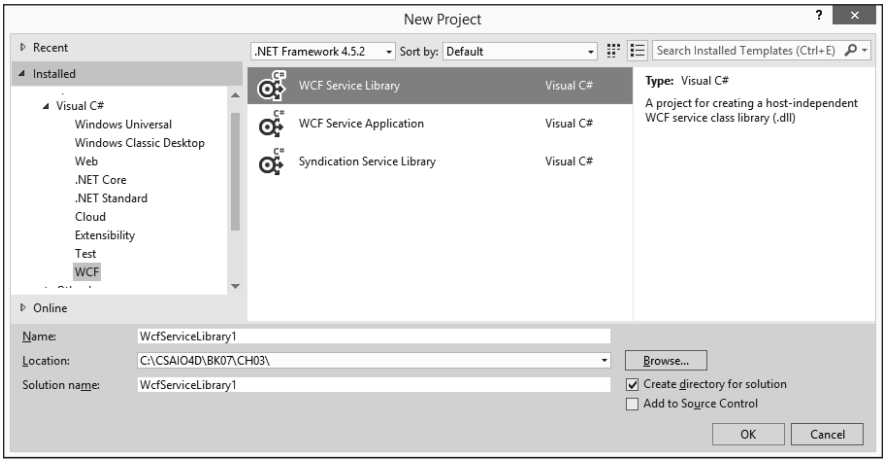


**FIGURE OC6-3:** Starting the new WCF Service.

3. **Highlight the WCF Service Library project type.**

4. **Type** ANewService **in the Name field, type** ServiceApp **in the Solution field, and click OK.**

   Using different names for the project name and the solution name makes it easier to add other projects to the solution. In this case, you need a second project to test the service. The wizard creates a new WCF service library for you with some example code included. You generally see `IService1.cs` opened, which is the interface description for your service library.

## Breaking it down

Look at the template code found in `Service1.cs` for a bit (you likely need to open it by double-clicking it in Solution Explorer). It's nothing like the ASMX code

because, remember, WCF is different. You can get it to do more or less the same thing as ASMX, but it isn't the same technology. Here's what the WCF code looks like:

```
using System;

namespace ANewService
{
    // NOTE: You can use the "Rename" command on the "Refactor" menu to
    // change the class name "Service1" in both code and config file
    // together.
    public class Service1 : IService1
    {
        public string GetData(int value)
        {
            return string.Format("You entered: {0}", value);
        }

        public CompositeType GetDataUsingDataContract(
            CompositeType composite)
        {
            if (composite == null)
            {
                throw new ArgumentNullException("composite");
            }
            if (composite.BoolValue)
            {
                composite.StringValue += "Suffix";
            }
            return composite;
        }
    }
}
```

A few of things are of interest here:

- **There is no `[WebMethod]` decoration.** You don't need it in WCF. The whole project is a service project.
- **The class implements a custom interface, `IService`.** The chapter gets to that in a second.
- **There are two sample methods.** One is the usual old Hello World–style method `GetData`; the other is a more complex example called `GetDataUsingContract`. You can also add asynchronous methods to your service, but the examples in this chapter don't provide them. Asynchronous methods would make it possible to overcome some types of load issues with your applications.

The `GetDataUsingContract` method makes a lot more sense when you take a look at `IService.cs`:

```
using System.Runtime.Serialization;
using System.ServiceModel;

namespace ANewService
{
    // NOTE: You can use the "Rename" command on the "Refactor" menu to
    // change the interface name "IService1" in both code and config file
    // together.
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        string GetData(int value);

        [OperationContract]
        CompositeType GetDataUsingDataContract(CompositeType composite);

        // TODO: Add your service operations here
    }

    // Use a data contract as illustrated in the sample below to add
    // composite types to service operations.

    // You can add XSD files into the project. After building the project,
    // you can directly use the data types defined there, with the
    // namespace "ANewService.ContractType".
    [DataContract]
    public class CompositeType
    {
        bool boolValue = true;
        string stringValue = "Hello ";

        [DataMember]
        public bool BoolValue
        {
            get { return boolValue; }
            set { boolValue = value; }
        }

        [DataMember]
        public string StringValue
        {
            get { return stringValue; }
```

```
            set { stringValue = value; }
        }
    }
}
```

This part of the example code, the interface upon which you base your class, does include decorations of various sorts. Don't let the name `CompositeType` fool you; this code is just a class like any other class.

Line 9 has a `[ServiceContract]` attribute that is similar to the `[WebClass]` of ASMX, and the `[OperationContract]` too, is similar to `[WebMethod]`, at least in usage.

Then you have the `[DataContract]`. This allows you to decorate classes — even in your domain model — with attributes that define which classes and methods get to go out to the service if and when the class is ever called into service.

## Devising a service update

The example in Online Chapter 5 creates a method, `DoAdd()`, that adds two numbers together. You access that method using both a Web Form and a console application. This chapter shows how to perform the same task by using WCF, which means updating the service so that it produces the same result. The changes begin in `IService1.cs`. Obviously, you want a better name than `IService1` for your service, so right-click its entry and choose Rename from the context menu. Type `TheService` and press Enter. Visual Studio makes the required changes in every file of your project.

Delete the `[OperationContract]` entries in the `TheService` interface. The following code replacements will create a version of `DoAdd()` for WCF:

```
[ServiceContract]
public interface TheService
{
    [OperationContract]
    int DoAdd(int Value1, int Value2);

    [OperationContract]
    ResultType DoAddUsingDataContract(InputType composite);
}
```


REMEMBER

You must provide an `[OperationContract]` for each of the access types you plan to support. In addition, anything that provides a data contract must also have a `[DataContract]` defined for it. In this case, you delete the existing `[DataContract]` and replace it with the `[DataContract]` classes shown here.

```
[DataContract]
public class InputType
{
    int Value1 = 1;
    int Value2 = 2;

    [DataMember]
    public int Value1Value
    {
        get { return Value1; }
        set { Value1 = value; }
    }

    [DataMember]
    public int Value2Value
    {
        get { return Value2; }
        set { Value2 = value; }
    }
}

[DataContract]
public class ResultType
{
    int Result = 0;

    [DataMember]
    public int ResultValue
    {
        get { return Result; }
        set { Result = value; }
    }
}
```

TIP

Note that the code has two classes: one for the input and another for the response. If you don't provide this sort of separation, the service becomes hard to use because it's too easy to confuse inputs and outputs. Each of the inputs and outputs must be marked as a `[DataMember]`.

To complete the service update, you must also create methods in the `Service1.cs` file that define how the service actually performs tasks. You must define one method for each of the `[OperationContract]` entries you define. Here are the methods used for this example:

```
public class Service1 : TheService
{
    public int DoAdd(int Value1, int Value2)
```

```
    {
        return Value1 + Value2;
    }

    public ResultType DoAddUsingDataContract(InputType composite)
    {
        if (composite == null)
        {
            throw new ArgumentNullException("composite");
        }

        ResultType result = new ResultType();

        result.ResultValue = composite.Value1Value +
            composite.Value2Value;

        return result;
    }
}
```

Using a data contract does require more work. However, the basic concepts are the same. The biggest difference is that each item has its own special class and you must use those classes when performing any tasks. In this case, you must define a `ResultType` object, `result`, before you can calculate the sum of `composite.Value1Value` and `composite.Value2Value` to return to the caller.

# Using the WCF Test Client utility

As with the ASMX service in Online Chapter 5, you can test your service. However, the WCF approach is more elegant and revealing. Start the service by pressing F5. You see a notification that the service is starting, and then you see the WCF Test Client utility. Note the hierarchical listing of the service content in the tree view in the left pane. Double-click the DoAdd() entry and you see the test page shown in Figure OC6-4.

Type **1** for the Value1 entry and **2** for the Value2 entry in the top-right pane. Click Invoke. The utility will normally display a security warning telling you of the potential for security issues when using the utility. Click OK to close the dialog box. You see the expected output in the bottom-right pane. If desired, you can also see the SOAP request and response by clicking the XML tab, as shown in Figure OC6-5.

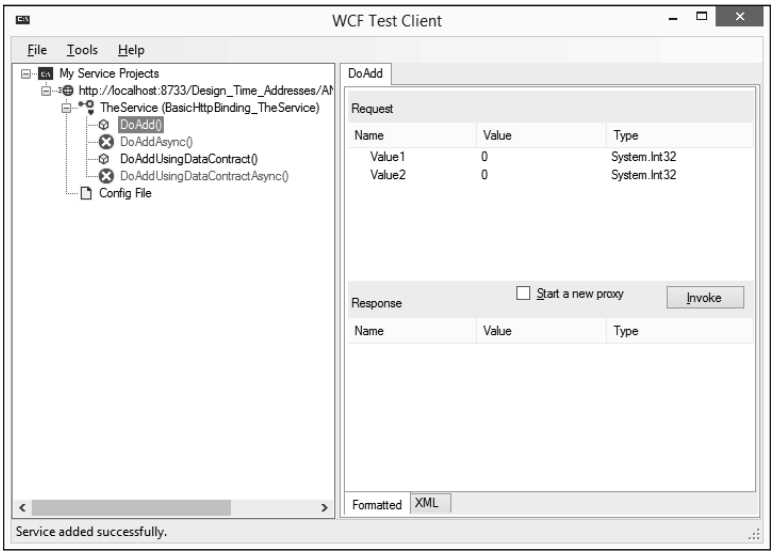


**FIGURE OC6-4:** The DoAdd() test page in the WCF Test Client utility.

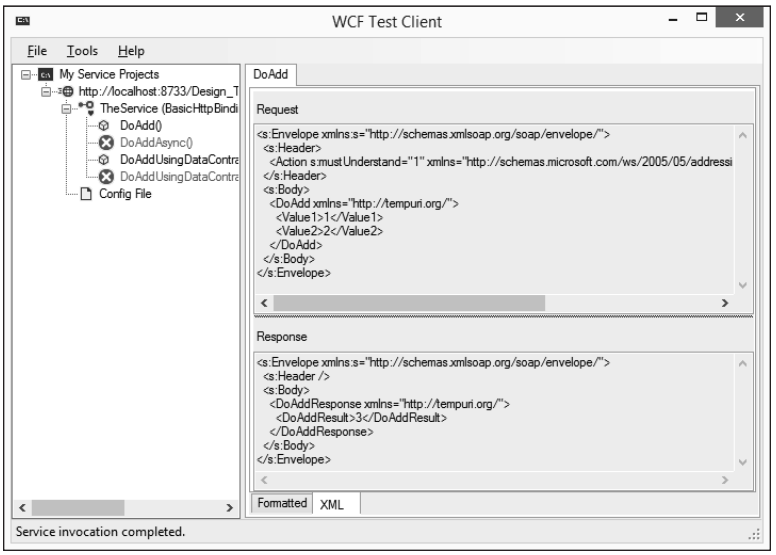


**FIGURE OC6-5:** The XML tab displays the SOAP request and response.

To see how the contract version differs, double-click the DoAddUsingDataContract() entry in the left pane. Note that the pane uses inputs that better match the data contract version. Type **1** in the Value1Value entry and **2** in the Value2Value entry. Click Invoke. You see the result shown in Figure OC6-6. As you can see, the process is the same but the level of information is deeper.

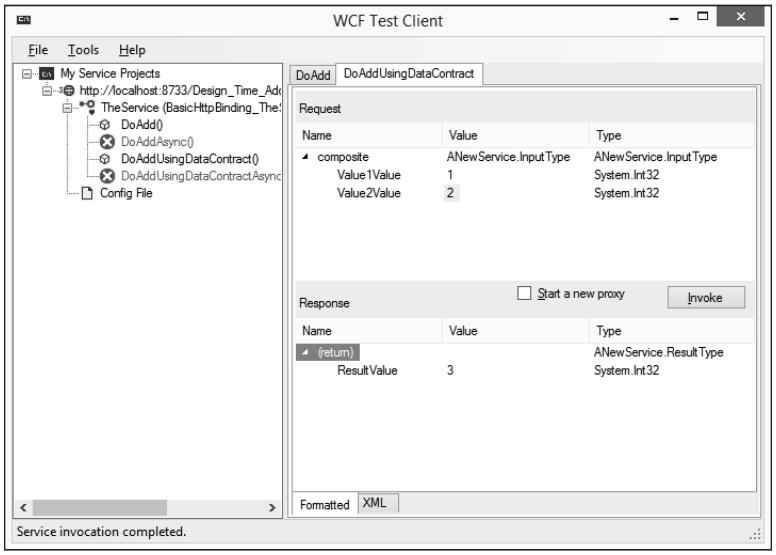


**FIGURE OC6-6:** The DoAddUsing DataContract() test page in the WCF Test Client utility.

# Working with WCF Services in Applications

Your service isn't much good if you can't use it with an application to perform useful tasks. The following sections explore how to use the WCF service that you created and tested in the previous sections to interact with an application.

## Adding the application project

Before you can perform testing, you need a new project. You can't add a Web Forms page directly to the service project as you did in the "Building a test page" section of Online Chapter 5. However, you can add it as a separate project, as you've done when creating web pages in the "Creating a standard project" section in Book 4, Chapter 2, in the book. Make sure to create an Empty project type and name it `WebFormTest`. After you have an empty web project to use, you can add a page to it and configure it using precisely the same directions as you did in the "Building a test page" section of Online Chapter 5.

There is one difference in this project. You actually need a second `Button`. Give the second `Button` these characteristics: `ID="InvokeContract"` and `Text="Contract"`.

## Creating the test code

The test code for `Invoke_Click()` looks surprisingly the same as the code used in the "Building a test page" section of Online Chapter 5. In fact, it is the same as shown here:

```
protected void Invoke_Click(object sender, EventArgs e)
{
    // Create the client.
    UseTheService.TheServiceClient Client =
        new UseTheService.TheServiceClient();

    // Use the client to invoke the web service.
    Result.Text = Client.DoAdd(
        Int32.Parse(Value1.Text),
        Int32.Parse(Value2.Text)).ToString();

}
```

REMEMBER

Whether you create your service using ASMX or WCF doesn't matter to the application. If the interfaces are the same, your user won't notice. The reason that this code works as it does is because the underlying reference that you create hides the details. If you were to browse the underlying reference code, you'd find that ASMX code and WCF code are quite different, but at this level, those differences don't matter.

You do have access to another method of working with the service in this case. A WCF service can provide asynchronous operations, which aren't implemented in this example but can also provide a data contract, which is. The following code shows how to work with the data contract form of the WCF service.

```
protected void InvokeContract_Click(object sender, EventArgs e)
{
    // Create the client.
    UseTheService.TheServiceClient Client =
        new UseTheService.TheServiceClient();

    // Obtain the input data.
    UseTheService.InputType inData =
        new UseTheService.InputType();
    inData.Value1Value = Int32.Parse(Value1.Text);
    inData.Value2Value = Int32.Parse(Value2.Text);

    // Use the client to invoke the web service.
    UseTheService.ResultType outData =
        Client.DoAddUsingDataContract(inData);
```

```
    // Display the data on screen.
    Result.Text = outData.ResultValue.ToString();
}
```

The code begins by creating a client, just as you did before. However, to use the contract, you must create the required data contract objects before making the call to `Client.DoAddUsingDataContract()`. In addition, you must perform extra steps to obtain the output and place it in `Result.Text`. However, using this approach tends to be less error prone because you know that the data is in the correct form before making the call.

## Testing the application

Before you do anything else, make sure that you right-click the WebFormTest entry in Solution Explorer and choose Set As Startup Project from the context menu. Otherwise, Visual Studio will insist on starting your service directly instead. This is by design: The first project in a solution is always the startup project, and sometimes you need to work with the service, rather than the test application.

Press F5 to run the application. What you see is similar to the Web Form provided in Online Chapter, with the addition of a button. The application works the same as in Online Chapter 5. It's interesting to test both the plain and the data contract versions of `DoAdd()`. You can see that they produce the same output and, as far as the user is concerned, there really isn't any difference.