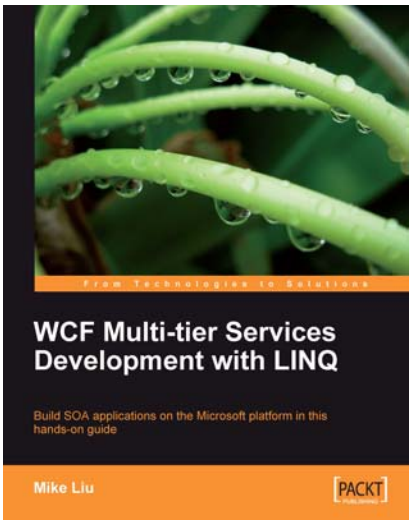
WCF Multi-tier Services Development with LINQ

Mike Liu



Chapter No. 5

"Implementing a WCF Service in the Real World"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Implementing a WCF Service in the Real World"

A synopsis of the book's content

Information on where to buy this book

About the Author

Mike Liu was born in 1966 in China. He studied Mathematics at Nanjing University between 1984 and 1988. After graduating with a Bachelor's degree, he worked as a Programmer / Senior Software Engineer / Architect under Unix and DOS using C/C++, DBase and Oracle. In 1995 he moved to New Zealand and studied Business Computing at the Auckland University of Technology in 1996. During his 5 years New Zealand, he worked as a Senior Software Engineer under Unix and Windows using C/C++, Java, FoxPro, Informix, Oracle and SQL Server. He moved to the United States in 2000, and since then has been working as a Web Developer / Senior Software Engineer / Principal Software Engineer under Unix and Windows using C/C++, C#, Visual Basic, Java, ASP, ASP.NET, Oracle and SQL Server. While working in the United States he studied Software Engineering at Brandeis University, and graduated in 2005 with a Master's degree.

Mike Liu had his first book (*MITT: Multi-user Integrated Table-processing Tool Under Unix*) published in 1993, and had his second book (*Advanced C# Programming*) published in 2003. He became a Sun Certified Java Programmer (SCJP) in 2000, a Microsoft Certified Solution Developer (MCSD) for Visual Studio 6.0 in 2001, and an MCSD for .NET in 2004.

Many thanks to the editors and technical reviewers at Packt Publishing. Without their help, this book won't be of such high quality. Thanks also to my wife, Julia Guo, and my two sons, Kevin and James Liu, for their patience and support while I was working on this book.

WCF Multi-tier Services Development with LINQ

WCF is Microsoft's unified programming model for building service-oriented applications. It enables developers to build secure, reliable, transacted solutions that integrate across platforms and interoperate with existing investments.

If you are a C++/C# developer looking for a book to build real-world WCF services, have probably run into the huge reference tomes currently available in the market. These books are crammed with more information than you need, and most build only simple one-tier WCF services. And if you plan to use LINQ in the data access layer, you will probably need to buy another volume that is just as huge, and just as expensive.

This book is the quickest and easiest way to learn WCF and LINQ in Visual Studio 2008. It is the first book to combine WCF and LINQ in a multi-tier real-world WCF service. Multi-tier services provide separation of concerns and better factoring of code, which gives you better maintainability and the ability to split layers out into separate tiers, for better scalability. WCF and LINQ are both powerful yet complex technologies from Microsoft, but this book will get you through. The mastery of these two topics will quickly get you started on creating service-oriented applications, and will allow you to take your first steps into the world of Service Oriented Architecture, without getting overwhelmed.

This book is a step-by-step tutorial with clear instructions and screenshots to guide you through the creation of a multi-tier real-world WCF service solution. It focuses on the essentials of using WCF and LINQ, rather than providing a reference to every single possibility. It leaves the reference material online where it belongs, and concentrates instead on practical examples, code, and advice.

What This Book Covers

- Creating, hosting, and consuming your first WCF service, in just a few minutes
- Exploring and learning different hosting and debugging options for a WCF service
- Building a multi-tier real-world WCF service from scratch to understand every layer of a WCF service, and applying it to your real work
- Adding exception handling to your WCF services
- Accelerating your WCF service development with Service Factory by applying best practices
- Understanding basic and advanced concepts and features of LINQ and LINQ to SQL
- Communicating securely and reliably with databases by rewriting the data access layer of your WCF service with LINQ to SQL
- Controlling concurrent updates to the databases and adding distributed transaction support to your WCF services

5

Implementing a WCF Service in the Real World

In the previous chapter, we created a basic WCF service. The WCF service we created, `HelloWorldService`, has only one method, called `GetMessage`. Because this is just an example, we implemented this WCF service in one layer only. Both the service interface and implementation are all within one deployable component.

In this chapter and the next one, we will implement a WCF Service, which will be called `RealNorthwindService`, to reflect a real world solution. In this chapter we will separate the service interface layer from the business logic layer, and in the next chapter we will add a data access layer to the service.

In this chapter, we will create and test the WCF service by following these steps:

- Create the project using a WCF Service Library template
- Create the project using a WCF Service Application template
- Create the Service Operation Contracts
- Create the Data Contracts
- Add a Product Entity project
- Add a business logic layer project
- Call the business logic layer from the service interface layer
- Test the service

Why layering a service?

An important aspect of SOA design is that service boundaries should be explicit, which means hiding all the details of the implementation behind the service boundary. This includes revealing or dictating what particular technology was used.

Further more, inside the implementation of a service, the code responsible for the data manipulation should be separated from the code responsible for the business logic. So in the real world it is always a good practice to implement a WCF service in three or more layers. The three layers are the service interface layer, the business logic layer, and the data access layer.

- **Service interface layer**: This layer will include the service contracts and operation contracts that are used to define the service interfaces that will be exposed at the service boundary. Data contracts are also defined to pass in to and out of the service. If any exception is expected to be thrown outside of the service, then Fault contracts will also be defined at this layer.
- **Business logic layer**: This layer will apply the actual business logic to the service operations. It will check the preconditions of each operation, perform business activities, and return any necessary results to the caller of the service.
- **Data access layer**: This layer will take care of all of the tasks needed to access the underlying databases. It will use a specific data adapter to query and update the databases. This layer will handle connections to databases, transaction processing, and concurrency controlling. Neither the service interface layer nor the business logic layer needs to worry about these things.

Layering provides separation of concerns and better factoring of code, which gives you better maintainability and the ability to split layers out into separate physical tiers, for scalability. The data access code should be separated out into its own layer that focuses on performing translation services between the databases and the application domain. Services should be placed in a separate service layer that focuses on performing translation services between the service-oriented external world and the application domain.

The service interface layer will be compiled into a separate class assembly, and hosted in a service host environment. The outside world will only know about and have access to this layer. Whenever a request is received by the service interface layer, the request will be dispatched to the business logic layer, and the business logic layer will get the actual work done. If any database support is needed by the business logic layer, it will always go through the data access layer.

Creating a new solution and project using WCF templates

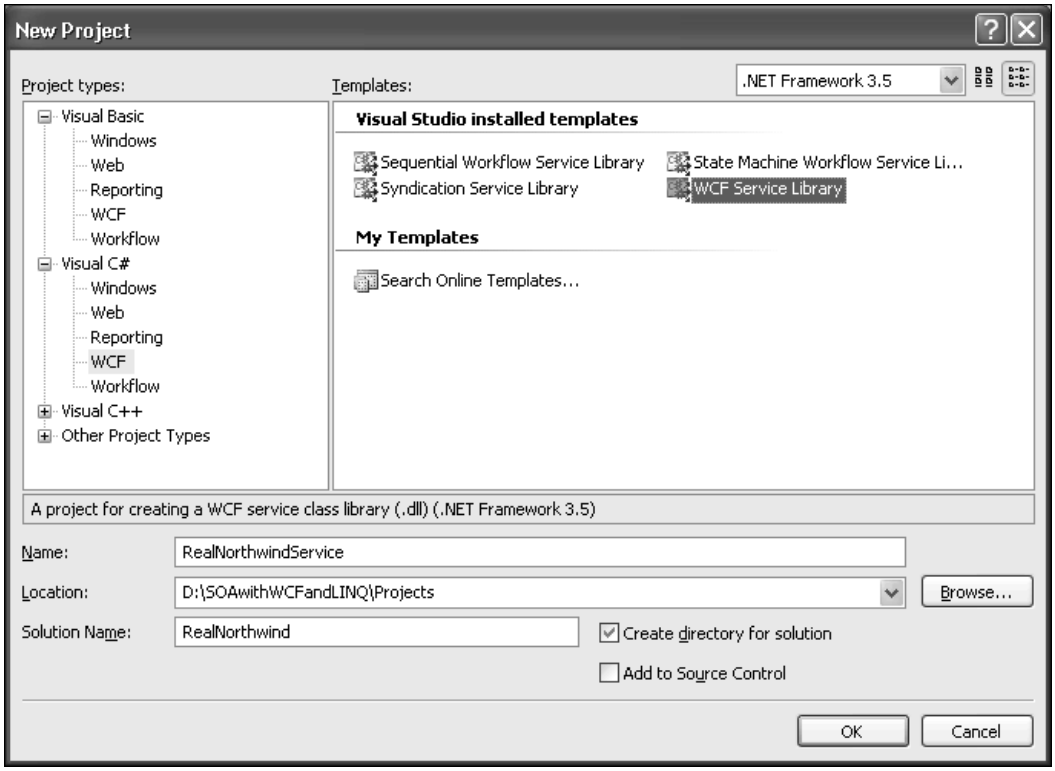
We need to create a new solution for this example, and add a new WCF project to this solution. This time we will use the built-in Visual Studio WCF templates for the new project.

Using the C# WCF service library template

There are two built-in WCF service templates within Visual Studio 2008: Visual Studio WCF Service Library and Visual Studio Service Application. In this section, we will use the service library template, and in the next section, we will use the service application template. Later, we will explain the differences between these two templates and choose the template that we are going to use for this chapter.

Follow these steps to create the `RealNorthwind` solution and the project using service library template:

1. Start Visual Studio 2008, select menu option **File | New | Project…,** and you will see the **New Project** dialog box. Do not open the `HelloWorld` solution from the previous chapter, as from this point onwards, we will create a completely new solution and save it in a different location.
2. In the **New Project** window, specify **Visual C# | WCF** as the project type, **WCF Service Library** as the project template, **RealNorthwindService** as the (project) name, and **RealNorthwind** as the solution name. Make sure that the checkbox **Create directory for solution** checkbox is selected.



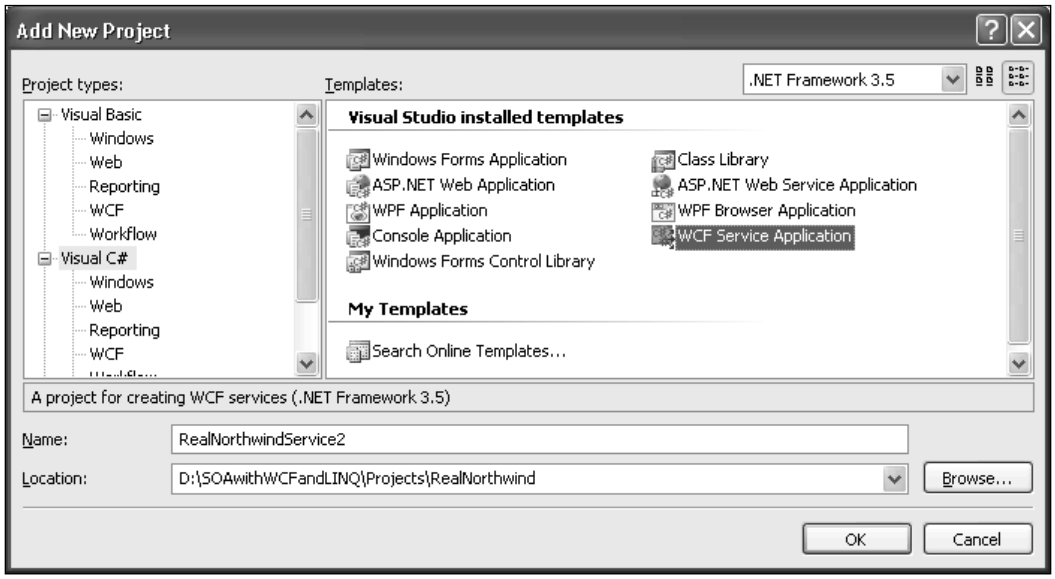
3. Click the **OK** button, and the solution is created with a WCF project inside it. The project already has a `IService1.cs` file to define an service interface and `Service1.cs` to implement the service. It also has an `app.config` file, which we will cover shortly.

Using the C# WCF service application template

Instead of using the Visual Studio WCF Service Library template to create our new WCF project, we can also use the Visual Studio Service Application template to create the new WCF project.

Because we have created the solution, we will add a new project using the Visual Studio WCF Service Application template.

1. Right-click on the solution item in the Solution Explorer, select menu option **Add | New Project...** from the context menu, and you will see the **Add New Project** dialog box.
2. In the **Add New Project** window, specify **Visual C#** as the project type, **WCF Service Application** as the project template, **RealNorthwindService2** as the (project) name, and leave the default location of **D:\SOAwithWCFandLINQ\Projects\RealNorthwind** unchanged.

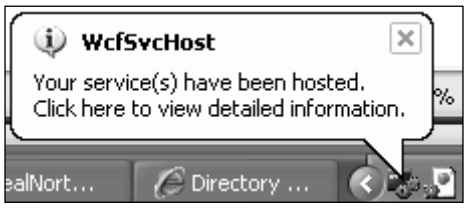


3. Click the **OK** button and the new project will be added to the solution. The project already has an `IService1.cs` file to define a service interface, and `Service1.svc.cs` to implement the service. It also has a `Service1.svc` file, and a `web.config` file, which are used to host the new WCF service. It has also had the necessary references added to the project such as `System.ServiceModel`.

You can follow these steps to test this service:

- Change this new project `RealNorthwindService2` to be the startup project (right-click on it from the Solution Explorer, and select **Set as Startup Project**). Then, run it (*Ctrl+F5* or *F5*). You will see that it can now run. You will see that an ASP.NET Development Server has been started, and a browser is open listing all of the files under the `RealNorthwindService2` project folder. Clicking on the **Service1.svc** file will open the Metadata page of the WCF service in this project. This is the same as we had discussed in the previous chapter for the `HostDevServer` project.
- If you have pressed *F5* in the previous step to run this project, you will see a warning message box asking you if you want to enable debugging for the WCF service. As we said earlier, you can choose enable debugging or just run in non-debugging mode.

You may also have noticed that the WCF Service Host is started together with the ASP.NET Development Server. This is actually another way of hosting a WCF service in Visual Studio 2008. It has been started at this point because, within the same solution, there is a WCF service project (`RealNorthwindService`) created using the WCF Service Library template. We will cover more of this host in a later section.



So far, we have used two different Visual Studio WCF templates to create two projects. The first project, using C# WCF Service Library template, is a more sophisticated one because this project is actually an application containing a WCF service, a hosting application (`WcfSvcHost`), and a WCF Test Client. This means that we don't need to write any other code to host it, and as soon as we have implemented a service, we can use the built-in WCF Test Client to invoke it. This makes it very convenient for WCF development.

The second project, using C# WCF Service Application template, is actually a website. This is the hosting application of the WCF service, so you don't have to create a separate hosting application for the WCF service. This is like a combination of the `HelloWorldService` and the `HostDevServer` applications we created in the previous chapter. As we have already covered them and you now have had a solid understanding of these styles, we will not discuss them any more. But keep in mind that you have this option, although in most cases it is better to keep the WCF service as clean as possible, without any hosting functionalities attached to it.

To focus on the WCF service using the WCF Service Library template, we now need to remove the project `RealNorthwindService2` from the solution.

In the Solution Explorer, right-click on the **RealNorthwindService2** project item, and select **Remove** from the context menu. Then, you will see a warning message box. Click the **OK** button in this message box, and the `RealNorthwindService2` project will be removed from the solution. Note that all the files of this project are still on your hard drive. You will need to delete them using Windows Explorer.

Creating the service interface layer

In the previous section, we created a WCF project using the WCF Service Library template. In this section, we will create the service interface layer contracts.

Because two sample files have already been created for us, we will try to re-use them as much as possible. Then, we will start customizing these two files to create the service contracts.

Creating the service interfaces

To create the service interfaces, we need to open the `IService1.cs` file, and do the following:

1. Change its namespace from `RealNorthwindService` to:
   `MyWCFServices.RealNorthwindService`

2. Change the interface name from **IService1** to **IProductService**. Don't be worried if you see the warning message before the interface definition line, as we will change the `web.config` file in one of the following steps.
3. Change the first operation contract definition from this line:

   ```
   string GetData(int value);
   ```

   To this line:

   ```
   Product GetProduct(int id);
   ```

4. Change the second operation contract definition from this line:

   ```
   CompositeType GetDataUsingDataContract(CompositeType composite);
   ```

   To this line:

   ```
   bool UpdateProduct(Product product);
   ```

5. Change the file's name from **IService1.cs** to **IProductService.cs**.

With these changes, we have defined two service contracts. The first one can be used to get the product details for a specific product ID, while the second one can be used to update a specific product. The product type, which we used to define these service contracts, is still not defined. We will define it right after this section.

The content of the service interface for `RealNorthwindService.ProductService` should look like this now:

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

namespace MyWCFServices.RealNorthwindService
{
    // NOTE: If you change the interface name "IService1" here, you
must also update the reference to "IService1" in App.config.
    [ServiceContract]
    public interface IProductService
    {
        [OperationContract]
        Product GetProduct(int id);

        [OperationContract]
        bool UpdateProduct(Product product);

        // TODO: Add your service operations here
    }
}
```

This is not the whole content of the `IProductService.cs` file. The bottom part of this file now should still have the class `CompositeType`, which we will change to our `Product` type in the next section.

Creating the data contracts

Another important aspect of SOA design is that you shouldn't assume that the consuming application supports a complex object model. A part of the service boundary definition is the data contract definition for the complex types that will be passed as operation parameters or return values.

For maximum interoperability and alignment with SOA principles, you should not pass any .NET specific types such as `DataSet` or `Exceptions` across the service boundary. You should stick to fairly simple data structure objects such as classes with properties, and backing member fields. You can pass objects that have nested complex types such as 'Customer with an Order collection'. However, you shouldn't make any assumption about the consumer being able to support object-oriented constructs such as inheritance, or base-classes for interoperable web services.

In our example, we will create a complex data type to represent a product object. This data contract will have five properties: `ProductID`, `ProductName`, `QuantityPerUnit`, `UnitPrice`, and `Discontinued`. These will be used to communicate with client applications. For example, a supplier may call the web service to update the price of a particular product, or to mark a product for discontinuation.

It is preferable to put data contracts in separate files within a separate assembly, but to simplify our example, we will put the `DataContract` within the same file as the service contract. So, we will modify the file `IProductService.cs` as follows:

1. Change the `DataContract` name from `CompositeType` to `Product`.
2. Change the fields from the following lines:

```
bool boolValue = true;
string stringValue = "Hello ";
```

   To these 7 lines:

```
int productID;
string productName;
string quantityPerUnit;
decimal unitPrice;
bool discontinued;
```

3. Delete the old `BoolValue`, and `StringValue` `DataMember` properties. Then, for each of the above fields, add a `DataMember` property. For example, for `productID`, we will have this `DataMember` property:

```
[DataMember]
public int ProductID
{
   get { return productID; }
   set { productID = value; }
}
```

A better way is to take advantage of the automatic property feature of C#, and add the following `ProductID` `DataMember` without defining the `productID` field:

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

To save some space, we will use the latter format. So, we need to delete all of those field definitions, and add an automatic property for each field, with the first letter capitalized.

The data contract part of the finished service contract file `IProductService.cs` should now look like this:

```
[DataContract]
public class Product
{
        [DataMember]
        public int ProductID { get; set; }
        [DataMember]
        public string ProductName { get; set; }
        [DataMember]
        public string QuantityPerUnit { get; set; }
        [DataMember]
        public decimal UnitPrice { get; set; }
        [DataMember]
        public bool Discontinued { get; set; }
}
```