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.


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

For More Information:
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.
Implementing a WCF Service in the Real World

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:

```csharp
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
    }
}
```
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 theDataContract 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:
   
   ```csharp
   bool boolValue = true;
   string stringValue = "Hello ";
   ```

   To these 7 lines:

   ```csharp
   int productID;
   string productName;
   string quantityPerUnit;
   decimal unitPrice;
   bool discontinued;
   ```
3. Delete the old BoolValue, and StringValueDataMember properties. Then, for each of the above fields, add a DataMember property. For example, for productID, we will have this DataMember property:

```csharp
[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 ProductIDDataMember without defining the productID field:

```csharp
[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:

```csharp
[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; }
}
```
Implementing the service contracts

To implement the two service interfaces that we defined in the previous section, open the Service1.cs file and do the following:

2. Change the class name from Service1 to ProductService. Make it inherit from the IProductService interface, instead of IService1. The class definition line should be like this:
   ```
   public class ProductService : IProductService
   ```
3. Delete the GetData and GetDataUsingDataContract methods
4. Add the following method, to get a product:
   ```
   public Product GetProduct(int id)
   {
       // TODO: call business logic layer to retrieve product
       Product product = new Product();
       product.ProductID = id;
       product.ProductName = "fake product name from service layer";
       product.UnitPrice = (decimal)10.0;
       return product;
   }
   ```
   In this method, we created a fake product and returned it to the client. Later, we will remove the hard-coded product from this method, and call the business logic to get the real product.
5. Add the following method to update a product:
   ```
   public bool UpdateProduct(Product product)
   {
       // TODO: call business logic layer to update product
       if (product.UnitPrice <= 0)
           return false;
       else
           return true;
   }
   ```
   Also, in this method, we don’t update anything. Instead, we always return true if a valid price is passed in. In one of the following sections, we will implement the business logic to update the product and apply some business logics to the update.
6. Change the file's name from Service1.cs to ProductService.cs. The content of the ProductService.cs file should be like this:

```csharp
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 class name "Service1" here, you must also update the reference to "Service1" in App.config.
    public class ProductService : IProductService
    {
        public Product GetProduct(int id)
        {
            // TODO: call business logic layer to retrieve product
            Product product = new Product();
            product.ProductID = id;
            product.ProductName = "fake product name from service layer";
            product.UnitPrice = (decimal)10;
            return product;
        }
        public bool UpdateProduct(Product product)
        {
            // TODO: call business logic layer to update product
            if (product.UnitPrice <= 0)
                return false;
            else
                return true;
        }
    }
}
```

**Modifying the app.config file**

Because we have changed the service name, we have to make the appropriate changes to the configuration file.
Follow these steps to change the configuration file:

1. Open app.config file from the Solution Explorer.
2. Change the RealNorthwindService string to MyWCFServices.
   RealNorthwindService. This is for the namespace change.
3. Change the Service1 string to ProductService. This is for the actual
   service name change.
4. Change the service address port from 8731 to 8080. This is to prepare for the
   client application.
5. You can also change the Design_Time_Addresses to whatever address you
   want, or delete this part from the service, baseAddress. This can be used to
   test your service locally.

The content of the app.config file should now look like this:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <!-- When deploying the service library project, the content of
       the config file must be added to the host's app.config file.
       System.Configuration does not support config files for
       libraries. -->
  <system.serviceModel>
    <services>
      <service name="MyWCFServices.RealNorthwindService.
                 ProductService" behaviorConfiguration="MyWCFServices.
                 RealNorthwindService.ProductServiceBehavior">
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/Design_Time_  
                  Addresses/MyWCFServices/RealNorthwindService/  
                  ProductService/" />
          </baseAddresses>
        </host>
        <!-- Unless fully qualified, address is relative to base
             address supplied above -->
        <endpoint address="" binding="wsHttpBinding"
                  contract="MyWCFServices.RealNorthwindService.IProductService">
          <!-- Upon deployment, the following identity element should
               be removed or replaced to reflect the identity under which
               the deployed service runs. If removed, WCF will infer an
               appropriate identity automatically. -->
        </endpoint>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```
Testing the service using WCF Test Client

Because we are using the WCF Service Library template in this example, we are now ready to test this web service. As we pointed out when creating this project, this service will be hosted in the Visual Studio 2008 WCF Service Host environment.

```xml
<identity>
  <dns value="localhost"/>
</identity>
</endpoint>
<!-- Metadata Endpoints -->
<!-- The Metadata Exchange endpoint is used by the service to describe itself to clients. -->
<!-- This endpoint does not use a secure binding and should be secured or removed before deployment -->
<endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
</service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name="MyWCFServices.RealNorthwindService.ProductServiceBehavior">
      <!-- To avoid disclosing metadata information, set the value below to false and remove the metadata endpoint above before deployment -->
      <serviceMetadata httpGetEnabled="True"/>
      <!-- To receive exception details in faults for debugging purposes, set the value below to true. Set to false before deployment to avoid disclosing exception information -->
      <serviceDebug includeExceptionDetailInFaults="False" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>
```
This is a new feature of Visual Studio 2008; if you are using Visual Studio 2005, you won't have this built-in functionality.

To start the service, press F5 or Ctrl+F5. The wcfSvcHost will be started and the WCF Test Client is also started. This is a Visual Studio 2008 built-in test client for WCF Service Library projects.

In order to run the WCF Test Client, you have to log in to your machine as a local administrator.

From this WCF Test Client, we can double-click on an operation to test it. First, let us test the GetProduct operation.

1. In the left panel of the client, double-click on the GetProduct operation; the GetProduct Request will be shown on the right-side panel.

2. In this Request panel, specify an integer for the product ID, and click the Invoke button to let the client call the service. You may get a dialog box to warn you about the security of sending information over the network. Click the OK button to acknowledge this warning (you can check the 'In the future, do not show this message' option, so that it won't be displayed again).
Now the message **Invoking Service...** will be displayed in the status bar, as the client is trying to connect to the server. It may take a while for this initial connection to be made, as several things need to be done in the background. Once the connection has been established, a channel will be created and the client will call the service to perform the requested operation. Once the operation has completed on the server side, the response package will be sent back to the client, and the WCF Test Client will display this response in the bottom panel.

If you have started the test client in the debugging mode (by pressing F5), you can set a breakpoint at a line inside the `GetProduct` method in the `RealNorthwindService.cs` file, and when the **Invoke** button is clicked, the breakpoint will be hit so that you can debug the service as we explained earlier.

Note that the response is always the same, no matter what product ID you use to retrieve the product. Specifically, the product name is hard-coded, as shown in the diagram. Moreover, from the client response panel, we can see that several properties of the `Product` object have been assigned default values.
Implementing a WCF Service in the Real World

Also, because the product ID is an integer value from the WCF Test Client, you can only enter an integer for it. If a non-integer value is entered, when you click the **Invoke** button, you will get an error message box to warn you that you have entered the wrong type.

![Microsoft WCF Test Client](image)

'abc' is not a valid value for this type

OK

Now let's test the operation, **UpdateProduct**.

- Double-click the **UpdateProduct** operation in the left panel, and **UpdateProduct** will be shown in the right-side panel, in a new tab.
- Enter a decimal value for the **UnitPrice** parameter, then click the **Invoke** button to test it. Depending on the value you entered in the **UnitPrice** column, you will get a True or False response package back.
The Request/Response packages are displayed in grids by default, but you have the option of displaying them in the XML format. Just select the XML tab from the bottom of the right-hand side panel, and you will see the XML formatted Request/Response packages. From these XML strings, you will discover that they are SOAP messages.

Besides testing operations, you can also look at the configuration settings of the web service. Just double-click on Config File from the left-side panel and the configuration file will be displayed in the right-side panel. This will show you the bindings for the service, the addresses of the service, and the contract for the service.

What you see here for the configuration file is not an exact image of the actual configuration file. It hides some information, such as debugging mode and service behavior, and includes some additional information on reliable sessions and compression mode.
If you are satisfied with the test results, just close the WCF Test Client, and you will go back to Visual Studio IDE. Note that as soon as you close the client, the WCF Service Host is stopped. This is different from hosting a service inside the ASP.NET Development Server, where after you close the client, the ASP.NET Development Server still does not stop.

**Testing the service using our own client**

It is very convenient to test a WCF service using the built-in WCF Test Client, but sometimes, it is desirable to test a WCF service using your own test client. The built-in WCF Test Client is limited to only simple WCF services. So for complex WCF services, we have to create our own test client. For this purpose, we can use the methods we learned earlier, to host the WCF service in IIS, the ASP.NET Development Server, or a managed .NET application, and create a test client to test the service.

In addition to the previous methods we learned, we can also use the built-in WCF Service Host to host the WCF service. So we don't need to create a host application, but just need to create a client. In this section, we will use this hosting method, to save us some time.
First, let us find a way to get the Metadata for the service. From the Visual Studio 2008 built-in WCF Test Client, you can’t examine the WSDL of the service, although the client itself must have used the WSDL to communicate with the service. To see the WSDL outside of the WCF Service Test Client, just copy the address of the service from the configuration file and paste it into a web browser. In our example, the address of the service is: http://localhost:8080/Design_Time_Addresses/MyWCFServices/RealNorthwindService/ProductService/. So, copy and paste this address to a web browser, and we will see the WSDL languages of the service, just as we have seen many times before.

To get the Metadata for the service, the service host application must run. The easiest way to start the RealNorthwindService in the WCF Service Host is to start the WCF Test Client and leave it running.
Now that we know how to get the Metadata for our service, we can start building the test client. We can leave the host application running, and manually generate the proxy classes using the same method that we used earlier. But this time we will let Visual Studio do it for us. So you can close the WCF Test Client for now.

Follow these steps to build your own client to test the WCF service:

1. Add a new Console Application project to the RealNorthwind solution. Let's call it RealNorthwindClient.

2. Add a reference to the WCF service. In the Visual Studio Solution Explorer, right-click on the RealNorthwindClient project, select Add Service Reference ... from the context menu, and you will see the Add Service Reference dialog box.

   ![Add Service Reference dialog box](image)

3. In the Add Service Reference dialog box, type the following address into the Address box, and then click the Go button to connect to the service:

   http://localhost:8080/Design_Time_Addresses/MyWCFServices/RealNorthwindService/ProductService/

   Also, you can simply click the Discover button (or click on the little arrow next to the Discover button, and select Services in Solution) to find this service.

For More Information:
In order to connect to or discover a service in the same solution, you don’t have to start the host application for the service. The WCF Service Host will be automatically started for this purpose. However, if it is not started in advance, it may take a while for the Add Service Reference window to download the required Metadata information for the service.

The ProductService should now be listed on the left-hand side of the window. You can expand it and select the service contract to view its details.

4. Next, let’s change the namespace of this service from ServiceReference1 to ProductServiceRef. This will make the reference meaningful in the code.

5. If you want to make this client run under .NET 2.0, click the Advanced button in the Add Service Reference window, and in the Service Reference Settings pop-up dialog box, click the Add Web Reference button. This will cause the proxy code will be generated based on the .NET 2.0 web services.

In this example, we won’t do this. So, click the Cancel button to discard these changes.
6. Now click the OK button in the Add Service Reference dialog box to add the service reference. You will see that a new folder, named ProductServiceRef, is created under Service References in the Solution Explorer for the RealNorthwindClient project. This folder contains lots of files, including the WSDL file, the service map, and the actual proxy code. If you can't see them, click Show All Files in the Solution Explorer.

A new file, App.config, is also added to the project, as well as several WCF-related references such as System.ServiceModel and System.Runtime.Serialization.

At this point, the proxy code to connect to the WCF service and the required configuration file have both been created and added to the project for us, without us having to enter a single line of code. What we need to do next is to write just a few lines of code to call this service.
Just as we did earlier, we will modify Program.cs to call the WCF service.

1. First, open Program.cs file, and add the following using line to the file:
   ```csharp
   using RealNorthwindClient.ProductServiceRef;
   ```
2. Then, inside the `Main` method, add the following line of code to create a client object:
   ```csharp
   ProductServiceClient client = new ProductServiceClient();
   ```
3. Finally, add the following lines to the file, to call the WCF service to get and update a product:
   ```csharp
   Product product = client.GetProduct(23);
   product.UnitPrice = (decimal)20.0;
   bool result = client.UpdateProduct(product);
   ```

The content of the Program.cs file is:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RealNorthwindClient.ProductServiceRef;

namespace RealNorthwindClient
{
    class Program
    {
        static void Main(string[] args)
        {
            ProductServiceClient client = new ProductServiceClient();

            Product product = client.GetProduct(23);
            Console.WriteLine("product name is " +
                product.ProductName);
            Console.WriteLine("product price is " +
                product.UnitPrice.ToString());

            product.UnitPrice = (decimal)20.0;
            bool result = client.UpdateProduct(product);
            Console.WriteLine("Update result is " +
                result.ToString());
        }
    }
}
```
Now you can run the client application to test the service. Remember that you need to set RealNorthwindClient to be the startup project before pressing F5 or Ctrl+F5.

If you want to start it in debugging mode (F5), you need to add a Console.readLine(); statement to the end of the program, so that you can see the output of the program. The WCF Service Host application will be started automatically before the client is started (but the WCF Test Client won’t be started).

If you want to start the client application in non-debugging mode (Ctrl+F5), you need to start the WCF Service Host application (and the WCF Test Client application) in advance. You can start it from another Visual Studio IDE instance, or you can set the RealNorthwindService as the startup project, start it in the non-debugging mode (Ctrl+F5), leave it running, and then change RealNorthwindClient to be the startup project, and start it in non-debugging mode. Also, you can set the solution to start with multiple projects with the RealNorthwindService as the first project to be run, and RealNorthwindClient as the second project to be run.

The output of this client program is as shown in the following figure:

Adding a business logic layer

Until now, the web service has contained only one layer. In this section, we will add a business logic layer, and define some business rules in this layer.
Adding the product entity project

Before we add the business logic layer, we need to add a project for business entities. The business entities project will hold all business entity object definitions such as products, customers, and orders. These entities will be used across the business logic layer, the data access layer and the service layer. They will be very similar to the data contracts we defined in the previous section, but will not be seen outside of the service. The Product entity will have the same properties as the product contract data, plus some extra properties such as UnitsInStock and ReorderLevel. These properties will be used internally, and shared by all layers of the service. For example, when an order is placed, the UnitsInStock should be updated as well. Also, if the updated UnitsInStock is less than the ReorderLevel, an event should be raised to trigger the re-ordering process.

The business entities by themselves do not act as a layer. They are just pure C# classes representing internal data within the service implementations. There is no logic inside these entities. Also, in our example these entities are very similar to the data contracts (with only two extra fields in the entity class), but in reality the entity classes could be very different from the data contracts, from property names and property types, to data structures.

As with the data contracts, the business entities' classes should be in their own assembly. So, we first need to create a project for them. Just add a new C# class library, RealNorthwindEntities, to the Solution. Then, rename the Class1.cs to ProductEntity.cs, and modify it as follows:

1. Change its namespace from RealNorthwindEntities to MyWCFServices.
2. Change the class name from Class1 to ProductEntity, if it hasn't been changed already
3. Add the following properties to this class:
   ProductID, ProductName, QuantityPerUnit, UnitPrice, Discontinued, UnitsInStock, UnitsOnOrder, ReorderLevel

Five of the above properties are also in the Product service data contract. The last three properties are for use inside the service implementations. Actually, we will use UnitsOnOrder to trigger business logic when updating a product, and update UnitsInStock and ReorderLevel to trigger business logic when saving an order (inside this book, we will not create a service for saving an order, but we assume that this is a required operation and will be implemented later).
Implementing a WCF Service in the Real World

The following is the code list of the ProductEntity class:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MyWCFServices.RealNorthwindEntities
{
    public class ProductEntity
    {
        public int ProductID { get; set; }
        public string ProductName { get; set; }
        public string QuantityPerUnit { get; set; }
        public decimal UnitPrice { get; set; }
        public int UnitsInStock { get; set; }
        public int ReorderLevel { get; set; }
        public int UnitsOnOrder { get; set; }
        public bool Discontinued { get; set; }
    }
}
```

Adding the business logic project

Next, let us create the business logic layer project. Again, we just need to add a new C# class library project, RealNorthwindLogic, to the solution. So, rename the Class1.cs to ProductLogic.cs, and then modify it as follows:

1. Change its namespace from RealNorthwindLogic to MyWCFServices.
2. Change the class name from Class1 to ProductLogic, if it hasn’t been changed
3. Add a reference to the project RealNorthwindEntities, as shown in the following Add Reference image:

![Add Reference](image-url)

For More Information:
Now, we need to add some code to the ProductLogic class.

1. Add the following using line:
   ```csharp
   using MyWCFServices.RealNorthwindEntities;
   ```

2. Add the method GetProduct. It should look like this:
   ```csharp
   public ProductEntity GetProduct(int id)
   {
      // TODO: call data access layer to retrieve product
      ProductEntity p = new ProductEntity();
      p.ProductID = id;
      p.ProductName = "fake product name from business logic layer";
      p.UnitPrice = (decimal)20.00;
      return p;
   }
   ```
   In this method, we create a ProductEntity object, assign values to some of its properties, and return it to the caller. Everything is still hard-coded so far.

   We hard code the product name as "fake product name from business logic layer", so that we know this is a different product from the one returned directly from the service layer.

3. Add the method UpdateProduct, as follows:
   ```csharp
   public bool UpdateProduct(ProductEntity product)
   {
      // TODO: call data access layer to update product
      // first check to see if it is a valid price
      if (product.UnitPrice <= 0)
         return false;
      // ProductName can't be empty
      else if (product.ProductName == null || product.ProductName.Length == 0)
         return false;
      // QuantityPerUnit can't be empty
      else if (product.QuantityPerUnit == null || product.QuantityPerUnit.Length == 0)
         return false;
      // then validate other properties
      else
   ```
Implementing a WCF Service in the Real World

ProductEntity productInDB = GetProduct(product.ProductID);
// invalid product to update
if (productInDB == null)
    return false;
// a product can't be discontinued if there are
// non-fulfilled orders
if (product.Discontinued == true && productInDB.
    UnitsOnOrder > 0)
    return false;
else
    return true;
}

4. Add test logic to the GetProduct method

We still haven't updated anything in a database, but this time, we have
added several pieces of logic to the UpdateProduct method. First, we check
the validity of the UnitPrice property, and return false if it is not a valid
one. We then check the product name and quantity per unit properties, to
make sure they are not empty. We then try to retrieve the product, to see if it
is a valid product to update. We also added a check to make sure that a
supplier can't discontinue a product if there are unfulfilled orders for this
product. However, at this stage, we can't truly enforce this logic, because
when we check the UnitsOnOrder property of a product, it is always 0 as we
didn't assign a value to it in the GetProduct method. For test purposes, we
can change the GetProduct method to include the following line of code:
if(id > 50) p.UnitsOnOrder = 30;

Now, when we test the service, we can select a product with an ID that is
greater than 50, and try to update its Discontinued property to see what
result we will get.

After you put all of this together, the content of the ProductLogic.cs file should be
as follows:

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MyWCFServices.RealNorthwindEntities;
using MyWCFServices.RealNorthwindDAL;

namespace MyWCFServices.RealNorthwindLogic
{
    public class ProductLogic
    {
public ProductEntity GetProduct(int id)
{
    // TODO: call data access layer to retrieve product
    ProductEntity p = new ProductEntity();
    p.ProductID = id;
    p.ProductName = "fake product name from business logic layer";
    //p.UnitPrice = (decimal)20.0;
    if(id > 50) p.UnitsOnOrder = 30;
    return p;
}

public bool UpdateProduct(ProductEntity product)
{
    // TODO: call data access layer to update product
    // first check to see if it is a valid price
    if (product.UnitPrice <= 0)
        return false;
    // ProductName can't be empty
    else if (product.ProductName == null || product.
        ProductName.Length == 0)
        return false;
    // QuantityPerUnit can't be empty
    else if (product.QuantityPerUnit == null || product.
        QuantityPerUnit.Length == 0)
        return false;
    // then validate other properties
    else
    {
        ProductEntity productInDB =
            GetProduct(product.ProductID);
        // invalid product to update
        if (productInDB == null)
            return false;
        // a product can't be discontinued if there are
        // non-fulfilled orders
        else if (product.Discontinued == true && productInDB.
            UnitsOnOrder > 0)
            return false;
        else
            return true;
    }
}
Calling the business logic layer from the service interface layer

We now have the business logic layer ready, and can modify the service contracts to call this layer, so that we can enforce some business logic.

First, we want to make it very clear that we are going to change the service implementations, and not the interfaces. So we will only change the ProductService.cs file.

We will not touch the file IProductService.cs. All of the existing clients (if there are any) that are referencing our service will not notice that we are changing the implementation.

Follow these steps to customize the service interface layer:

1. Add a reference to the business logic layer.

   In order to call a method inside the business logic layer, we need to add a reference to the assembly that the business logic is included in. We will also use the ProductEntity class. So we need a reference to the RealNorthwind-Entities as well.

   To add a reference, from the Solution Explorer, right-click on the project RealNorthwindService, select Add Reference … from the context menu, and select RealNorthwindLogic from the Projects tab. Also, select RealNorthwindEntities as we will need a reference to the ProductEntity inside it. Just hold down the Ctrl key while you are selecting multiple projects. Click the OK button to add references to the selected projects.
2. Now we have added two references. We can add the following two using statements to the ProductService.cs file so that we don't need to type the full names for their classes.

```csharp
using MyWCFServices.RealNorthwindEntities;
using MyWCFServices.RealNorthwindLogic;
```

3. Now, inside the GetProduct method, we can use the following statements to get the product from our business logic layer:

```csharp
ProductLogic productLogic = new ProductLogic();
ProductEntity product = productLogic.GetProduct(id);
```

4. However, we cannot return this product back to the caller, because this product is of the type ProductEntity, which is not the type that the caller is expecting. The caller is expecting a return value of the type Product, which is a data contract defined within the service interface. We need to translate this ProductEntity object to a Product object. To do this, we add the following new method to the ProductService class:

```csharp
private void TranslateProductEntityToProductContractData(
    ProductEntity productEntity,
    Product product)
{
    product.ProductID = productEntity.ProductID;
    product.ProductName = productEntity.ProductName;
    product.QuantityPerUnit = productEntity.QuantityPerUnit;
    product.UnitPrice = productEntity.UnitPrice;
    product.Discontinued = productEntity.Discontinued;
}
```

Inside this translation method, we copy all of the properties from the ProductEntity object to the service contract data object, but not the last three properties—UnitsInStock, UnitsOnOrder, and ReorderLevel. These three properties are used only inside the service implementations. Outside callers cannot see them at all.

The GetProduct method should now look like this:

```csharp
public Product GetProduct(int id)
{
    ProductLogic productLogic = new ProductLogic();
    ProductEntity productEntity = productLogic.GetProduct(id);
    Product product = new Product();
    TranslateProductEntityToProductContractData
        (productEntity, product);
    return product;
}
```
Implementing a WCF Service in the Real World

We can modify the `UpdateProduct` method in the same way, making it like this:

```csharp
public bool UpdateProduct(Product product)
{
    ProductLogic productLogic = new ProductLogic();
    ProductEntity productEntity = new ProductEntity();
    TranslateProductContractDataToProductEntity(
        product, productEntity);
    return productLogic.UpdateProduct(productEntity);
}
```

5. Note that we have to create a new method to translate a product contract data object to a `ProductEntity` object. In translation, we leave the three extra properties unassigned in the `ProductEntity` object, because we know a supplier won’t update these properties. Also, we have to create a `ProductLogic` variable in both the methods, so that we can make it a class member:

```csharp
ProductLogic productLogic = new ProductLogic();
```

The final content of the `ProductService.cs` file is as follows:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
using MyWCFServices.RealNorthwindEntities;
using MyWCFServices.RealNorthwindLogic;

namespace MyWCFServices.RealNorthwindService
{
    // NOTE: If you change the class name "Service1" here, you must also update the reference to "Service1" in App.config.
    public class ProductService : IProductService
    {
        ProductLogic productLogic = new ProductLogic();

        public Product GetProduct(int id)
        {
            /*
             // TODO: call business logic layer to retrieve product
             Product product = new Product();
             product.ProductID = id;
            */
        }
    }
}
```

For More Information:

product.ProductName = "fake product name from service layer";
product.UnitPrice = (decimal)10.0;
*/
ProductEntity productEntity = productLogic.GetProduct(id);
Product product = new Product();
TranslateProductEntityToProductContractData(
    productEntity, product);
return product;
}
public bool UpdateProduct(Product product)
{
    /*
    // TODO: call business logic layer to update product
    if (product.UnitPrice <= 0)
        return false;
    else
        return true;
    */
    ProductEntity productEntity = new ProductEntity();
    TranslateProductContractDataToProductEntity(
        product, productEntity);
    return productLogic.UpdateProduct(productEntity);
}
private void TranslateProductEntityToProductContractData(
    ProductEntity productEntity,
    Product product)
{
    product.ProductID = productEntity.ProductID;
    product.ProductName = productEntity.ProductName;
    product.QuantityPerUnit = productEntity.QuantityPerUnit;
    product.UnitPrice = productEntity.UnitPrice;
    product.Discontinued = productEntity.Discontinued;
}
private void TranslateProductContractDataToProductEntity(
    Product product,
    ProductEntity productEntity)
{
    productEntity.ProductID = product.ProductID;
    productEntity.ProductName = product.ProductName;
    productEntity.QuantityPerUnit = product.QuantityPerUnit;
    productEntity.UnitPrice = product.UnitPrice;
    productEntity.Discontinued = product.Discontinued;
}
Implementing a WCF Service in the Real World

Testing the WCF service with a business logic layer

We can now compile and test the new service with a business logic layer. We will use the WCF Test Client to simplify the process.

1. Make the project RealNorthwindService the startup project
2. Start the WCF Service Host application and WCF Service Test Client, by pressing F5 or Ctrl+F5
3. In the WCF Service Test Client, double-click on the GetProduct operation, to bring up the GetProduct test screen
4. Enter a value of 56 for the ID field and then click the Invoke button

You will see that this time the product is returned from the business logic layer, instead of the service layer. Also, note that the UnitsOnOrder property is not displayed as it is not part of the service contract data type. However, we know that a product has a property UnitsOnOrder, and we will actually use this for our next test.
Chapter 5

Now, let us try to update a product.

1. In the WCF Service Test Client, double-click on the UpdateProduct operation to bring up the UpdateProduct test screen.
2. Enter -10 as the price, and click the Invoke button. You will see that the Response result is False.
3. Enter a valid price, say 25.60, a name, and a quantity per unit, leave the Discontinued property set to False, and then click the Invoke button. You will see that the Response result is now True.
4. Change the Discontinued value from False to True, and click the Invoke button again. The Response result is still True. This is because we didn't change the product ID, and it has defaulted to 0.
5. Change the product ID to 51, leave the Discontinued value as True and product price as 25.60, and click the Invoke button again. This time, you will see that the Response result is False. This is because the business logic layer has checked the UnitsOnOrder and Discontinued properties, and didn't allow us to make the update.
Implementing a WCF Service in the Real World

Summary
In this chapter, we have created a real world WCF service that has a service contract layer, and a business logic layer. The key points in this chapter include:

- WCF Services should have explicit boundaries
- The WCF Service Application template can be used to create WCF services with a hosting web site created within the project
- The WCF Service Library template can be used to create WCF services that will be hosted by the WCF Service Host, and these can be tested using the WCF service Test Client
- The service interface layer should contain only the service contracts, such as the operation contracts, and data contracts
- The business logic layer should contain the implementation of the service
- The business entities represent the internal data of the service shared by all of the layers of the service, and they should not be exposed to the clients

For More Information:
Where to buy this book
You can buy WCF Multi-tier Services Development with LINQ from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.