CSC310-A

Module 4

Implementing Windows Communication Foundation

1 Implementing a WCF Service

Windows Communication Foundation lets you implement service-oriented distributed applications inter-operating across LAN, WAN, Intranet and Internet, without being a complex and time-consuming endeavor. Differences between network communication protocols and messaging formats are handled through configuration as well as additional features; such as security, discovery and routing can be easily enabled without affecting business logic. You can get started by learning the basics to build, host and access WCF services in this chapter. The WCF Programming model is based on communication between two entities: a WCF client and a WCF service. While the WCF service can be implemented as a library, you need to build an application to host the service. To build the service, the host and as well as the client you need to reference the following assemblies. You can begin by creating a class library project named **SymBank.Services**. Add the following references. Generate a data model for the **SymBank** database in the project but you need to remove relationships between the tables since bidirectional relationships cannot be serialized.

Standard WCF assemblies to reference

```
System.Runtime.Serialization
System.ServiceModel
```

1.1 ServiceContract & DataContract

A client accesses a WCF service through a service contract. A class or interface can be declared as a service contract by using the **ServiceContract** attribute. However, it would be more preferable to use an interface as a service contract. This allows us to abstract the implementation away from the declaration so that you do not have to distribute any code to the client. Abstraction also allows you to provide multiple and different implementations for the same service. You can also reduce both network and system resources to host multiple service classes in one application.

Declaring a class as the service contract

```
namespace SymBank.Services {
    [System.ServiceModel.ServiceContract]
    public class BankingServices {
                :
    }
}
```

Mark them with **ServiceContract** attribute from **System.ServiceModel** namespace as shown below. You also need to mark each method as **OperationContract** that you will allow the client to access through the interface.

IAccountService service contract : IAccountService.cs

```
[ServiceContract]
public interface IAccountService {
    [OperationContract]void Add(Account item);
    [OperationContract]Account GetAccount(int code);
    [OperationContract]List<Account> GetAccountList();
    [OperationContract]List<Account> GetAccountsForName(string text);
        :
}
```

ITransactionService service contract: ITransactionService.cs

```
[ServiceContract]
public interface ITransactionService {
    [OperationContract]int Debit(int source, decimal amount);
    [OperationContract]int Credit(int source, decimal amount);
    [OperationContract]int Transfer(int source, int target, decimal amount);
}
```

Before C# 4.0, it was necessary to mark data-oriented classes as DataContract and public field or property as DataMember to allow a type and members to be serializable between the WCF client and the service. In C# 4.0, this is not required and all data-oriented are now automatically serializable. Even though objects can reference other objects there must not be any bidirectional relationship between them.

DataContact and DataMembers

```
[DataContract]
public class Account {
    [DataMember]public int Code { get; set; }
    [DataMember]public string Name { get; set; }
    [DataMember]public int Type { get; set; }
    [DataMember]public string ZipCode { get; set; }
    [DataMember]public string Creator { get; set; }
    [DataMember]public DateTime Created { get; set; }
    [DataMember]public decimal Balance { get; set; }
}
```

1.2 FaultException

When an exception occurs, the host will return a generic **FaultException** back to the client and not the original exception to hide the details of the exception from potential hackers. The host will not serialize any exception except FaultException. This can then be used to differentiate between expected exceptions and unexpected exceptions due to bugs in the service. Thus you would need to always catch all exceptions and return a custom FaultException containing only the bare details that you would allow a client to receive.

Implementing IAccountService contract: BankingServices.cs

```
public static string UserName {
    get { return Thread.CurrentPrincipal.Identity.Name; }
}

public void Add(Account item) {
    try { ... }
    catch (Exception ex) {
        throw new FaultException("Add account failed. " + ex.Message);
    }
}

public Account GetAccount(int code) {
    try { ... }
    catch (Exception ex) {
        throw new FaultException("Get account failed. " + ex.Message);
    }
}

public List<Account> GetAccountList() {
    try { ... }
    catch (Exception ex) {
        throw new FaultException("Get account list failed. " + ex.Message);
    }
}

public List<Account> GetAccountsForName(string name) {
    try { ... }
    catch (Exception ex) {
        throw new FaultException(
            "Search accounts list failed. " + ex.Message);
    }
}

public int Debit(int source, decimal amount) {
    try { ... }
    catch (Exception ex) {
        throw new FaultException(
            "Debit account failed. " + ex.Message);
    }
}

public int Credit(int source, decimal amount) {
    try { ... }
    catch (Exception ex) {
        throw new FaultException(
            "Credit account failed. " + ex.Message);
    }
}

public int Transfer(int source, int target, decimal amount) {
    try { ... }
    catch (Exception ex) {
        throw new FaultException(
            "Account transfer failed. " + ex.Message);
    }
}
```

Since it is easier to debug a service locally rather than remotely, you need to test it first to ensure there are no bugs in your code. This allows you to trust your assembly and issues that may occur later on may then be due to networking and configuration issues of WCF rather than because of bugs in your code. Add a console application project named **WCFClient1** and reference the **SymBank.Services** assembly. Write the following code to test the services interfaces from an instantiated service object.

Testing the services locally: WCFClient1\Program.cs

```
static void Main() {
    var obj = new BankingServices();
    IAccountService service1 = obj;
    ITransactionService service2 = obj;
    var list = service1.GetAccountList();
    foreach (var item in list)
        Console.WriteLine("{0},{1},{2}",
            item.Code, item.Name, item.Balance);
    int ref1 = service2.Debit(100, 1000m);
    int ref2 = service2.Credit(200, 500m);
    int ref3 = service2.Transfer(100, 200, 500m);
    Console.WriteLine(ref1);
    Console.WriteLine(ref2);
    Console.WriteLine(ref3);
}
```

2 Hosting a WCF Service

.NET Remoting does not provide authentication, authorization or encryption services. If you need to build secure services, you would have to host it in a web application so you can make use of HTTPS for encryption and ASP.NET for authorization. However this would not be required for WCF services. WCF has a complete set of components that you can enable or disable through behavior configuration. Thus you can still have security regardless of what is the kind of host application include *Console application*, *Windows Forms application* or *WPF application*. You can also build a *Windows service application* to allow you to start and stop the service remotely. You will learn how to build custom host applications in this chapter.

2.1 Service Host

You can now add a console application project to the solution by using the following information. Add reference to the standard WCF assemblies and **SymBank.Services**. To expose a WCF service over the network, you need to create a **ServiceHost** and pass in the service class type. Once a **ServiceHost** has been created, you can call **Open** method to start the host to listen and process incoming requests. You will also be able to stop the service host by calling a **Close** method. The host will also close automatically when the process terminates.

Project information

```
Project Name: WCFHost1
Project Type: Visual C# | Windows | Console Application
Location: C:\CSDEV\SRC\
Solution: Module5
```

Add a **App.Config** file to configure endpoints for the service. An endpoint represent a location that clients connect to access the service. And endpoint is constructed from three parts; the *Address*, the *Binding* and the *Contract* which is basically the ABC of WCF programming. You can write code to access all the endpoints configured for the service for debugging purposes. The endpoints will only be accessible once the service is opened. You can choose to write code to create the endpoints or configure them in the application configuration file. Endpoints are created before the host is opened so it is only possible to access them after calling **Open** method. The following shows how to create a basic host and to verify the endpoints that have been created correctly.

Hosting a WCF service: WCFHost1

```
using System;
using System.ServiceModel;
using SymBank.Services;

class Program {
    static void Main() {
        ServiceHost host = new ServiceHost(typeof(BankingServices));
        host.Open();
        foreach (var endpoint in host.Description.Endpoints) {
            Console.WriteLine(endpoint.Address);
            Console.WriteLine(endpoint.Binding);
            Console.WriteLine(endpoint.Contract.ContractType);
            Console.WriteLine();
        }
        while (true) {
            Console.WriteLine("Host started. Press F12 to terminate.");
            var input = Console.ReadKey(true);
            if (input.Key == ConsoleKey.F12) break;
        }
        host.Close();
    }
}
```

2.2 Address-Binding-Contract

Clients access a service through service endpoints. The endpoint controls the protocol and configuration to access a service. Services usually have more than one endpoint, and each one may have its own characteristics and settings. For example, a service can have one endpoint that listens to HTTP communication and does not require any authentication while another endpoint listens to Transmission Control Protocol (TCP) communication and may requires a client to send a Windows identity in order to pass through the endpoint. Services can have multiple endpoints because of difference in technology, security, reliability, quality of services requirements and other reasons.

Each endpoint is comprised of an Address, a Binding, and a Contract, or in short ABC. Each endpoint references a service contract to allow a host to know what operations it can expect to receive in the request. An endpoint is also defined for a specific binding. A binding refers to the transport type that is used for the communication such as TCP, HTTP, or Named Pipes. Therefore, one endpoint can only listen to a specific transport type. Every endpoint has an address. The address will then uniquely identify a specific endpoint from all of the endpoints to which a host listens. The address is of URL form, and must match the type of transport used by the binding. You can programmatically add endpoints or using a configuration file. Following shows how to write the code to add endpoints for a particular service. This must be done before calling **Open** method on the service host.

Programmatically adding endpoints: WCFHost1\Program.cs

```
host.AddServiceEndpoint(typeof(IAccountService), new BasicHttpBinding(),
    "http://localhost:8080/symbank/services/account");
host.AddServiceEndpoint(typeof(ITransactionService), new BasicHttpBinding()
    "http://localhost:8080/symbank/services/transaction");
host.AddServiceEndpoint(typeof(IAccountService), new NetTcpBinding(),
    "net.tcp://localhost:8081/symbank/services/account");
host.AddServiceEndpoint(typeof(ITransactionService), new NetTcpBinding(),
"net.tcp://localhost:8081/symbank/services/transaction");
```

However if you need to change the endpoint configuration, you would have to rebuild and redeploy the host application. Configuration of endpoints can also be done in the application configuration file. The administrator can then choose to change the service endpoint configuration at any time and restart the service host. Use the configuration section **system.serviceModel** to configure WCF **services**.

Configuration section for WCF services: WCFHost1\App.Config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
            :
        </services>
    </system.serviceModel>
</configuration>
```

Service and endpoints

```
<service name="SymBank.Services.BankingServices">
    <endpoint address="http://localhost:8080/symbank/services/account"
        binding="basicHttpBinding"
        contract="SymBank.Services.IAccountService" />
    <endpoint address="http://localhost:8080/symbank/services/transaction"
        binding="basicHttpBinding"
        contract="SymBank.Services.ITransactionService" />
    <endpoint address="net.tcp://localhost:8081/symbank/services/account"
        binding="netTcpBinding"
        contract="SymBank.Services.IAccountService" />
    <endpoint address="net.tcp://localhost:8081/symbank/services/transaction"
        binding="netTcpBinding"
        contract="SymBank.Services.ITransactionService" />
</service>
```

Since addresses are the mostly the same for each service contract, you can specify the similar part as the base addresses of the service host and the different part as the address in the endpoint. If there are endpoints that uses a different address, you can still use the full address for that endpoint.

Setting Service host base addresses

```
<service name="SymBank.Services.BankingServices">
    <host>
        <baseAddresses>
        <add baseAddress="http://localhost:8080/symbank/services/" />
        <add baseAddress="net.tcp://localhost:8081/symbank/services/" />
        </baseAddresses>
    </host>
    <endpoint address="account" binding="basicHttpBinding"
          contract="SymBank.Services.IAccountService" />
    <endpoint address="transaction" binding="basicHttpBinding"
          contract="SymBank.Services.ITransactionService" />
    <endpoint address="account" binding="netTcpBinding"
          contract="SymBank.Services.IAccountService" />
    <endpoint address="transaction" binding="netTcpBinding"
          contract="SymBank.Services.ITransactionService" />
</service>
```

You can now build and run the host application. Everything should work except if the port number used in the base address is already used for another network service. When this happens, just change the port number to any other unused port. Use the following console command to check what ports are in use.

Check ports in use

```
C:\> netstat -an
```

You can now update the client to access the services remotely. In **WCFClient1**, add references to WCF assemblies. Call **CreateChannel** method on **ChannelFactory<T>** class where **T** represents the service contract where you wish to access. You need to also specify the binding and endpoint address. This method will create a proxy that imitates the service contract specified. When you call methods on the proxy through the service contract, the proxy will communicate with the service through the service host.

Accessing remove services: WCFClient1

```
IAccountService service1 =
    ChannelFactory<IAccountService>.CreateChannel(new NetTcpBinding(),
        new EndpointAddress("net.tcp://localhost:8081/services"));
ITransactionService service2 =
    ChannelFactory<ITransactionService>.CreateChannel(new BasicHttpBinding(),
        new EndpointAddress("http://localhost:8080/services"));
```

2.3 Service Metadata

If you move **BankingServices** class from **SymBank.Services** project to **WCFHost1**, everything still works. As long as a client uses the service remotely, only the service contracts are required on the client-side. This allows us to distribute the library as a *contracts-only* assembly. In this way, the client will not be able to see the code. This improves security as well allowing us to update the service implementation without having to keep updating the client as long as its service contracts are not changed. Better still, we do not deploy the library at all to the client and let them generate their own. However to allow them to be able to generate the service and data contracts without having to write them manually, you can expose the metadata of the service through a **IMetadataExchange** service contract.

WCF consists of a set of optional features called behaviors. Not all of these features are enabled by default on the services, endpoints and bindings. We can add custom behavior configuration sections for services and endpoints to enable or disable these behaviors. You can add a **serviceMetadata** service behavior to the service to enable the service to provide metadata about all the services it provide. You will also need to add an endpoint to expose this service and to determine what protocol can be used to access the service.

Enabling service metadata: WCFHost1\App.Config

```
<behaviors>
    <serviceBehaviors>
        <behavior name="default">
            <serviceMetadata />
        </behavior>
    </serviceBehaviors>
</behaviors>
<services>
    <service behaviorConfiguration="default"
        name="SymBank.Services.BankingServices">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8080/symbank/services/" />
            <add baseAddress="net.tcp://localhost:8081/symbank/services/"/>
          </baseAddresses>
        </host>
        <endpoint address="mex" binding="mexHttpBinding"
            contract="IMetadataExchange" />
                  :
    </service>
</services>
```

You can now add a new console application project named **WCFClient2** to the same solution. However, this project does not reference **SymBank.Services**. Instead right click over the project and select *Add Service Reference…* option. Enter the location of the metadata service. It should be able to retrieve and display the metadata for the service class. Enter **Services** as a namespace for the proxies generated. The name for the proxy class is the same name as the service plus the **Client** keyword. Create an instance of the proxy and access the service through the proxy or through the service interface.

The endpoint and binding configurations will be automatically created for you in the application configuration file. If you only have one endpoint supporting the service type, it will automatically be selected otherwise you need to supply the name of the endpoint configuration to use. In our example, each server has two endpoints; the **http** and **net.tcp**.

Instancing and accessing the proxy: WCFClient2\Program.cs

```
var proxy = new AccountServiceClient("NetTcpBinding_IAccountService");
var service1 = (IAccountService)proxy; var list = service.GetList();
foreach (var item in list) Console.WriteLine("{0},{1},{2}",
        item.Code, item.Name, item.Balance);
proxy.Close();
```

Endpoint configuration used by proxy: App.Config

```
<endpoint address="net.tcp://localhost:8081/symbank/services/account"
    binding="netTcpBinding" bindingConfiguration="NetTcpBinding_IAccountService"
   contract="Services.IAccountService" name="NetTcpBinding_IAccountService">
```

2.4 Hosting in a Windows Service

Add a new console application named **WCFHost2** to the project. Then add the references to **System.Runtime.Serialization**, **System.ServiceModel** and the the other assemblies. Add a *Windows Service* class named **MyService** to the project. Go to code view and write the code to create the service host in the **OnStart** method and shutdown the host in the **OnStop** method.

A Windows Service class: WCFHost2\MyService.cs

```
partial class MyService : ServiceBase {  // ServiceBase from System.ServiceProcess.dll
    public ServiceHost _host;
    public MyService() { ServiceName = "MyService"; }
    protected override void OnStart(string[] args) {
        if (_host != null) OnStop();
        _host = new ServiceHost(typeof(BankingServices));
        _host.Open();
    }
    protected override void OnStop() {
        if (_host != null) { _host.Close(); _host = null; }
}
```

Main method to run the Windows service: Program.cs

```
public class Program {
    public static void Main() {
        ServiceBase.Run(new MyService()); }
}
```

You now need to create the installer for the service. Open the **MyService** class and right-click on the designer window and select **Add Installer** option. When the installer class has been created, use **Properties** window to setup installers. For the service process installer, set the **Account** property to **LocalSystem**. In the service installer set the **DisplayName** and **Description** properties. You can optionally set the **StartType** to **Automatic** instead of **Manual** if you wanted to. Build and use the **installutil** tool to install the Windows service. You can use **/u** option to uninstall the service. If you service fails to start, check *Event Viewer* to view the errors.

Installing the Windows service

```
installutil WFCHost2.exe
```

Uninstalling the Windows service

```
installutil /u WFCHost2.exe
```

3 Secured Services

3.1 Authentication & Authorization

The service is supposed to be already providing the identity of the caller through the following property. However what you do get back is an anonymous identity since the service did not request clients to be authenticated. You can configure security using a binding configuration. The following will then force the client to provide credentials in the transport protocol to be authenticated on the server.

Retrieving the caller's identity

```
public static string UserName {
    get { return Thread.CurrentPrincipal.Identity.Name; }
}
```

Enabling basic authentication: ServiceHost1\App.Config

```
<bindings>
    <basicHttpBinding>
        <binding name="authenticated">
            <security mode="TransportCredentialOnly">
                <transport clientCredentialType="Windows" />
            </security>
        </binding>
    </basicHttpBinding>
<bindings>
```

Assigning binding configuration to endpoint

```
<endpoint address="" binding="basicHttpBinding"
    bindingConfiguration="authenticated"
   contract="SymBank.Services.IAccountService" />
```

You also need to refresh the proxy in the client application so bindings for the proxy will have the same security settings. You can add code in operations to display the identity that is accessing the service to see if the credentials are sent over. To test this, make sure that you are using different accounts to run the client and the service. The client identity should now appear at the server side.

Just because a user is authenticated on a service does not mean that they can simply access everything. You can use Code Access Security to ensure only the right people are allowed access. In CAS, use **PrincipalPermission** attribute to check user or role. You may also write code-based authorization.

Implementing declarative authorization

```
[PrincipalPermission(SecurityAction.Demand, Role="Banking")]
[PrincipalPermission(SecurityAction.Demand, Role="Administrators")]
public List<Account> GetAccountList() {
    Console.WriteLine(UserName);
        :
}
```

Implementing programmatic authorization

```
var principal = Thread.CurrentPrincipal;
if (!(principal.IsInRole("Banking") || principal.IsInRole("Administrator")))
    throw new FaultException("Access denied!");
                :
}
```

3.2 Message Encryption

Note that BasicHttpBinding does not support message encryption. You can only make use of the transport encryption by HTTP/SSL. NetTcpBinding and WSHttpBinding have support for message encryption that is faster and does not require certificates. If you switch the binding on the endpoints from BasicHttpBinding to WSHttpBinding, you can then enable message encryption also through HTTP. Alternatively this can be enabled directly on a NetTcpBinding.

Enabling message security

```
<netTcpBinding>
    <binding name="encrypted">
    <security mode="Message">
        <message algorithmSuite="Basic128"
            clientCredentialType="Windows" />
      </security>
    </binding>
</netTcpBinding>
```

Using encrypted binding configuration

```
<endpoint address="net.tcp://localhost:8090/symbank/services"
    binding="netTcpBinding"
    bindingConfiguration="encrypted"
    contract="SymBank.Services.IAccountService" />
```

Rebuild and then execute the host. Then refresh the client proxy to ensure that it has the same security configuration as the service. Please ensure the the client proxy has been updated to use the same encryption algorithm. If not you should regenerate the proxy again. Note that enabling message security will also enable authentication so all your existing authorization code will continue to work.