

Professional WCF Programming: .NET Development with the Windows® Communication Foundation

Chapter 8: Services

ISBN-10: 0-470-08984-9

ISBN-13: 978-0-470-08984-2



Copyright of Wiley Publishing, Inc.
Posted with Permission

8

Services

This chapter focuses on those items that are specific to the service side of Windows Communication Foundation. The chapters up until now have been dealing with topics and concepts that apply to the service and the client, and a lot of that still applies to the service itself. Yet, there are some items that are service specific, and this chapter delves into those.

As a quick recap, Chapters 4 through 6 discussed addresses, bindings, and contracts; concepts that apply to both the service and the client. Chapter 7 focused on the client itself and discussed topics that addressed WCF from the perspective of the client such as consuming WCF services, channels, and communication patterns.

This chapter covers the following topics:

- Overview of WCF service concepts
- Behaviors
- Error handling

Overview

When you write a book you spend a lot of time organizing and laying out the topics and content that you are going to write about. A good portion of that time is trying to figure out the best way to organize and lay out the topics that will provide a smooth flow of content and benefit the reader the most. Do you discuss bindings first, or addresses? Maybe discussing contracts first would make more sense. These are the types of questions that keep an author up at night (besides writing).

It is hard to understand, let alone build, Windows Communication Foundation services without first understanding addresses, bindings, and contracts because that is what defines a WCF service. That is why three chapters were dedicated to those specific topics. With that information firmly

Part II: Programming Windows Communication Foundation

stored on your cerebral hard drive, this section spends a few pages providing an overview of those topics and discusses some aspects that are particular to WCF services.

Service Types

You have read about the different WCF service types a couple of times so far in this book, but they are worth mentioning briefly again. There are three types of services in WCF, and they are the following:

- ❑ Typed
- ❑ Untyped
- ❑ Typed message

Which service type you use is basically determined by how you want your service to accept and return parameters.

Typed

You learned in previous chapters that a typed service is the least complicated of the three service types and they provide most of the functionality that you will need when developing WCF services. Remember that a typed service is a lot like a class method or function, in that it can accept one or more parameters and can return a result.

Another common term for typed services is *parameter model*, which defines specifically what this type of service does. Typed, or parameter, services are not limited to the types of parameters and results that can be passed to them. Typed services can accept simple types and even complex types. Return values are not limited in any way, because return values can also be simple or complex types. However, keep in mind that when passing or returning complex types, data contracts must be defined for each type being passed and returned.

With typed services Windows Communication Foundation handles all of the messaging. This means that as a developer, you do not have to work directly at the message level.

The following example illustrates a typed service. A service contract is defined with two operations. The first operation accepts no parameters, and the second operation accepts two parameters (in this case, both parameters are of type `Int`):

```
[ServiceContract]
public interface IServiceClass
{
    [OperationContract]
    string GetText();

    [OperationContract]
    int MultiplyNumbers(int firstvalue, int secondvalue);
}

public class ServiceClass : IServiceClass
{
    string IServiceClass.GetText()
```

```

    {
        StreamReader sw = new StreamReader(@"c:\wrox\WCFServiceTest.txt");
        return sw.ReadLine();
        //return "Hello World";
    }

    int IServiceClass.MultiplyNumbers(int firstvalue, int secondvalue)
    {
        return firstvalue * secondvalue;
    }
}

```

Typed services also support `ref` and `out` parameters, which have the added benefit of letting the service return multiple results. The following example shows the use of an `out` parameter:

```

public void IServiceClass.MultiplyNumbers(int val1, int val2, out int retval)
{
    retval = val1 * val2;
}

```

As a note, it isn't considered good practice to return more than one parameter in object-oriented programming.

You can see that typed services are simple to work with and are very similar to programming methods you already use.

Untyped

Untyped services are a bit more complicated to work with because they necessitate working directly with the message. In this type of service, you define the messages and their contents. Here you are working at the message level, meaning that message objects are passed back and forth between the client and service, and the service may also return a message object if required. Furthermore, they provide the ability to access the message body, as well as serialize and deserialize the message itself.

The following example is taken from the message contract example in Chapter 6. Instead of sending known types as parameters, message objects are passed between the client and the service:

```

[ServiceContract]
public interface IServiceClass
{
    [OperationContract]
    string InitiateOrder();

    [OperationContract]
    BookOrder PlaceOrder(BookOrder request);

    [OperationContract]
    string FinalizeOrder();
}

[MessageContract]
public class BookOrder

```

Part II: Programming Windows Communication Foundation

```
{

    private string isbn;
    private int quantity;
    private string ordernumber;

    public BookOrder(BookOrder message)
    {
        this.isbn = message.isbn;
        this.quantity = message.quantity;
    }

    [MessageHeader]
    public string ISBN
    {
        get { return isbn; }
        set { isbn = value; }
    }

    [MessageBodyMember]
    public int Quantity
    {
        get { return quantity; }
        set { quantity = value; }
    }

    [MessageBodyMember]
    public string OrderNumber
    {
        get { return ordernumber; }
        set { ordernumber = value; }
    }
}

public class ServiceClass : IServiceClass
{
    string IServiceClass.InitiateOrder()
    {
        return "Initiating Order...";
    }

    public BookOrder PlaceOrder(BookOrder request)
    {
        BookOrder response = new BookOrder(request);
        response.OrderNumber = "12345678";
        return response;
    }

    string IServiceClass.FinalizeOrder()
    {
        return "Order placed successfully.";
    }
}
```

Typed Message

A typed message service, also known as the “message contract model,” is where you define the messages and their contents. Here, the messages are Message Contract attributes, meaning that message contracts are used to define typed message classes, sending custom messages in and out of service operations.

The following example, taken from Chapter 6, shows a service contract along with a defined typed message that is used as a parameter in a typed service operation:

```
[ServiceContract]
public interface IBookOrder
{
    [OperationContract]
    void PlaceOrder(Contract MyContract);
}

[MessageContract]
public class MyContract
{
    [MessageHeader]
    string Title;
    [MessageBodyMember]
    decimal cost;
}
```

Before this chapter gets to the really good information, one last quick review of services and endpoints is in order.

Service Contracts

By now you should have a good grasp of service contracts. A service contains one or more operations, all of which are annotated with specific attributes that define them as WCF services and available service methods, or operations. Methods not tagged with the `[OperationContract]` attribute are normal methods and are not available or accessible to the client, but can be called internally from within the service. This is analogous to making a web service’s method consumable by decorating it with the `[WebMethod]` attribute.

As you are aware, the signature of a contract is critical because it is the mechanism by which the outside world understands the service. The signature of a service is effectively created by defining a method and annotating it with the `[OperationContract]` attribute. A service contract is created by grouping the operations within a defined interface and annotating the interface with the `[OperationContract]` attribute.

The following example shows a simple WCF service and exposed service operations:

```
[ServiceContract]
public interface IServiceClass
{
    [OperationContract]
    string GetText();

    [OperationContract]
}
```

Part II: Programming Windows Communication Foundation

```
    int MultiplyNumbers(int firstvalue, int secondvalue);
}

public class ServiceClass : IServiceClass
{
    string IServiceClass.GetText()
    {
        StreamReader sw = new StreamReader(@"c:\wrox\WCFServiceTest.txt");
        return sw.ReadLine();
        //return "Hello World";
    }

    int IServiceClass.MultiplyNumbers(int firstvalue, int secondvalue)
    {
        return firstvalue * secondvalue;
    }
}
```

Service Endpoints

Clients can only access a service through service endpoints. Endpoints can be defined in code or in a configuration file. A service can have one or more endpoints, and each endpoint must have an address, a binding, and a service contract. Chapters 4, 5, and 6, respectively, discussed each of these in detail.

The following sections provide a review of defining endpoints in code and in a configuration file.

Specifying in Code

The following example illustrates defining an endpoint in code. As stated earlier, an endpoint needs an address, a binding, and a service contract. The first two lines define the addresses through which the service will be accessed.

The next line creates the service host through which the service will be hosted. The addresses are passed as parameters in the service host constructor.

The next two lines define the bindings that the endpoint will use, and the remaining lines call the `AddEndpoint` method of the `ServiceHost` class to add the endpoints, passing the service contract, defined bindings, and defined addresses. The service is then opened and available to clients:

```
Uri bpa = new Uri("net.pipe://localhost/NetNamedPipeBinding");
Uri tcpa = new Uri("net.tcp://localhost:8000/TcpBinding");

sh = new ServiceHost(typeof(ServiceClass), bpa, tcpa);

NetNamedPipeBinding pb = new NetNamedPipeBinding();
NetTcpBinding tcpb = new NetTcpBinding();

ServiceMetadataBehavior mBehave = new ServiceMetadataBehavior();
sh.Description.Behaviors.Add(mBehave);
sh.AddServiceEndpoint(typeof(IMetadataExchange),
```

```

MetadataExchangeBindings.CreateMexTcpBinding(), "mex");

sh.AddServiceEndpoint(typeof(IMetadataExchange),
MetadataExchangeBindings.CreateMexNamedPipeBinding(), "mex");

sh.AddServiceEndpoint(typeof(IServiceClass), pb, bpa);
sh.AddServiceEndpoint(typeof(IServiceClass), tcpb, tcpa);

sh.Open();

```

Specifying in Configuration

The same service endpoints can be defined in a configuration file as shown in the following code. It has been stated a number of times that the preferred method of defining an endpoint is through configuration because of the flexibility and ease of deployment:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="WCFService.ServiceClass"
behaviorConfiguration="metadataSupport">
        <host>
          <baseAddresses>
            <add baseAddress="net.pipe://localhost/WCFService"/>
            <add baseAddress="net.tcp://localhost:8000/WCFService"/>
            <add baseAddress="http://localhost:8080/WCFService"/>
          </baseAddresses>
        </host>
        <endpoint address="tcpmex"
          binding="mexTcpBinding"
          contract="IMetadataExchange"/>
        <endpoint address="namedpipemex"
          binding="mexNamedPipeBinding"
          contract="IMetadataExchange"/>
        <endpoint address="" binding="wsHttpBinding"
contract="WCFService.IServiceClass"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="metadataSupport">
          <serviceMetadata httpGetEnabled="false" httpGetUrl=""/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

When defining an endpoint in configuration, the ServiceHost object automatically scans the configuration file for defined endpoints. Hence, endpoints can be added, modified, or removed non-invasively. This makes adding endpoints and deployment much easier over defining them in code.

Service Behaviors

Up until now the discussion has focused solely on service contracts that define the inputs, outputs, data types, and exposed functionality of a service. Service contracts, when implemented, create a class that when combined with address and binding information makes the service available to clients.

Yet, given all of this information and functionality, the need to control service execution aspects and characteristics of the service is still critical. For example, how do you control threading issues and manage service instances?

The answer is simple: behaviors. Service behaviors are objects that modify and control the runtime characteristics of Windows Communication Foundation services. When a WCF service contract is implemented you then have the ability to shape many of the execution characteristics of the service. These behaviors, or characteristics, are controlled by configuring a runtime property, or through the defining of custom behaviors.

There are two types of behaviors in Windows Communication Foundation: service behaviors and operation behaviors. They are applied just like all the other WCF objects, by adding attributes. The following two sections discuss the `[ServiceBehavior]` attribute and the `[OperationBehavior]` attribute.

Unlike the other attributes that you have learned about so far, such as the `[ServiceContract]` and `[OperationContract]` attributes, which are defined inside an interface, the `[ServiceBehavior]` and `[OperationBehavior]` attributes are applied to the class that implements the interface.

The following code snippet illustrates how the `[ServiceBehavior]` and `[OperationBehavior]` attributes are applied:

```
[ServiceContract]
public interface IServiceClass
{
    [OperationContract]
    int AddNumbers(int number1, int number2);

    [OperationContract]
    int SubtractNumbers(int number1, int number2);
}

[ServiceBehavior]
public class ServiceClass : IServiceClass
{
    [OperationBehavior]
    public int AddNumbers(int number1, int numbers2)
    {
        return number1 + number2;
    }
    [OperationBehavior]
    public int SubtractNumbers(int number1, int numbers2)
    {
        return number1 - number2;
    }
}
```

Just like the other attributes, the `[ServiceBehavior]` and `[OperationBehavior]` attributes have a number of available properties that assist in specifying the behaviors of the service. In the preceding example, the attributes would take on the default values of each property because no properties were explicitly specified.

The following two sections discuss the available properties for both the `[ServiceBehavior]` and `[OperationBehavior]` attributes.

ServiceBehavior Attribute

The `[ServiceBehavior]` attribute comes from the `System.ServiceModel.ServiceBehaviorAttribute` class and specifies the execution behavior of a service contract implementation. The `[ServiceBehavior]` attribute contains the following properties:

- `AddressFilterMode`
- `AutomaticSessionShutdown`
- `ConcurrencyMode`
- `ConfigurationName`
- `IgnoreExtensionDataObject`
- `IncludeExceptionDetailInFaults`
- `InstanceContextMode`
- `ReleaseServiceInstanceOnTransactionComplete`
- `TransactionAutoCompleteOnSessionClose`
- `TransactionIsolationLevel`
- `TransactionTimeout`
- `UseSynchronizedContext`
- `ValidateMustUnderstand`

These properties are discussed in detail in the following sections.

AddressFilterMode

The `AddressFilterMode` property gets or sets the address filter mode of the service. This mode is used by the service dispatcher to route incoming messages to the appropriate endpoint. The modes used by this property come from the `AddressFilterMode` enumeration, and the available values are as follows:

- Any:** Applies a filter that matches on any address of an incoming message.
- Exact:** Applies a filter that matches on the address of an incoming message.
- Prefix:** Applies a filter that matches on the longest prefix of the address of an incoming message.

The following example illustrates using the `AddressFilterMode` property to apply a filter that matches on any address of an incoming message:

Part II: Programming Windows Communication Foundation

```
[ServiceBehavior(AddressFilterMode=AddressFilterMode.Any)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *Exact*.

AutomaticSessionShutdown

The `AutomaticSessionShutdown` property is used to specify whether to close a session automatically when a client closes an output session. If the property value is `true`, the service automatically closes its session when the client closes its output session. The service does not close the session until it has finished processing all remaining messages. For custom control of the session, set the value of this property to `false`.

The following example sets the `AutomaticSessionShutdown` property to a value of *false*. The service will not close any sessions automatically:

```
[ServiceBehavior(AutomaticSessionShutdown=false)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *true*.

An output channel is a channel that can send messages, thus an output session is a session established prior to the acceptance of an output channel, which then sends messages that have a session id for that connection/session. Output sessions are typically not important at the application level. They only come into necessity if you are building channels.

ConcurrencyMode

The `ConcurrencyMode` property specifies the thread support for the service. The modes used by this property come from the `ConcurrencyMode` enumeration, and the available values are as follows:

- ❑ **Single:** The service instance is single threaded and does not accept reentrance calls. If a message arrives while another message is currently being processed, the newly arrived message must wait until the current message is finished being processed.
- ❑ **Multiple:** The service is multi-threaded. In this mode, synchronization and state consistency must be handled manually because no synchronization guarantees are made. This is because threads can change the service object at any given time.
- ❑ **Reentrant:** The service is single threaded and can accept reentrant calls.

The following example sets the `ConcurrencyMode` property to a value of *Single*. The service will be single threaded and not accept reentrant calls:

```
[ServiceBehavior(ConcurrencyMode=ConcurrencyMode.Single)]
public class ServiceClass : IServiceClass
{
    ...
}
```

Using the reentrant mode is useful when one service calls another service, which calls the first service in return. A reentrant service accepts calls when it calls out. Because of possible state inconsistencies, you must be careful to leave your service object state in a consistent state prior to calling out.

Setting this property to a value of *Single* tells the system to limit all instances of the service to a single thread of execution. The biggest benefit of this is that there are no threading issues that you have to concern yourself with. Setting the property to a value of *Multiple* informs the system that service objects can be run on multiple threads, which means thread safety is left up to the developer.

The default value for this property is *Single*.

ConfigurationName

The *ConfigurationName* property specifies or retrieves the value that is used to locate the service element in a configuration file.

The following example sets the *ConfigurationName* to a value of "service":

```
[ServiceBehavior(ConfigurationName="service")]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is the namespace qualified name of the type without the assembly information.

IgnoreExtensionDataObject

The *IgnoreExtensionDataObject* property specifies whether to send unknown serialization data between the service and client. In typical communication, most types are defined, and the service knows how to handle each member. For example, the *BookOrder* type may be defined with *ISBN* and *Quantity* elements, and the service expects these elements. It is possible, however, to send elements that the service is not aware of, such as a *Title* element. In these cases, any Windows Communication Foundation type that implements the *IExtensibleDataObject* interface stores any extra data sent over. In this example, the service would hang on to the *Title* element for serialization later and be re-emitted.

The following example sets the *IgnoreExtensionDataObject* to *true*, which tells the service to ignore any extra data (elements) and to not re-emit them:

```
[ServiceBehavior(IgnoreExtensionDataObject=true)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *false*.

IncludeExceptionDetailInFaults

The *IncludeExceptionDetailInFaults* property specifies what is to be done with unhandled exceptions. When the value of this property is *false*, unhandled exceptions are converted into a *System.ServiceModel.FaultException* of type *System.ServiceModel.ExceptionDetail*.

Part II: Programming Windows Communication Foundation

The root of this goes back to how exceptions are handled in managed applications. Processing errors are represented as Exception objects in managed applications, and in SOAP applications error information is communicated via SOAP fault messages. Windows Communication Foundation utilizes both types of error systems (Exception objects and SOAP fault messages). Therefore, in WCF any managed exception must be converted to a SOAP fault message to be sent over the wire.

This type of information is extremely useful during development to help debug your service, but it is recommended that this property be set to false in production. For more information on this, see the section titled “Handling Exceptions” later in this chapter.

The following example sets the `IncludeExceptionDetailInFaults` property to a value of *true*. The service will then send information back to the client about any internal service method exceptions:

```
[ServiceBehavior(IncludeExceptionDetailInFaults=true)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *false*.

InstanceContextMode

The `InstanceContextMode` property indicates at what point new service objects are created. There is not a direct connection between the service object and the communication channel, therefore the lifetime of the service object is completely independent of the client-service channel. This property specifies the lifetime of the `InstanceContext` object. The lifetime of the user-defined object is the lifetime of the `InstanceContext` by default.

Windows Communication Foundation can create a new `InstanceContext` object for the following situations:

- ❑ **PerCall:** A new `InstanceContext` object is created, and recycled, succeeding each call.
- ❑ **PerSession:** A new `InstanceContext` object is created per session, and the instance is not sharable between multiple sessions.
- ❑ **Single:** A single `InstanceContext` object is created and used for all incoming calls and is not recycled succeeding the calls.

The following example sets the `InstanceContextMode` property to a value of *PerSession*. The service will create a new object when a new communication session is initiated by the client. All calls from the client to the service will be handled by the same service object:

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *PerSession*.

For more information regarding `InstanceContext` and its interaction with WCF services, see the section “`InstanceContext`” later in this chapter.

ReleaseServiceInstanceOnTransactionComplete

The `ReleaseServiceInstanceOnTransactionComplete` property specifies whether the current service object is recycled when the current transaction is complete.

The following example sets the `ReleaseServiceInstanceOnTransactionComplete` property to a value of *true*. The service will recycle the service object:

```
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete=true)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *true*.

TransactionAutoCompleteOnSessionClose

The `TransactionAutoCompleteOnSessionClose` property specifies whether any pending transactions are to be completed when the current session is closed.

The following example sets the `TransactionAutoCompleteOnSessionClose` property to a value of *true*. All pending transactions will be complete when the current session is closed:

```
[ServiceBehavior(TransactionAutoCompleteOnSessionClose=true)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *false*.

TransactionIsolationLevel

The `TransactionIsolationLevel` property specifies the transaction isolation level of the service. The levels used by this property come from the `System.Transactions` namespace and its `IsolationLevel` enumeration. To specify one of the following values, you will need to add a reference to the `System.Transactions` namespace and include a *using* statement to that namespace.

The available values are as follows:

- Chaos:** The pending changes from more highly visible transactions cannot be overwritten.
- ReadCommitted:** Volatile data cannot be read during the transaction, but can be modified.
- ReadUncommitted:** Volatile data can be read and modified during the transaction.
- RepeatableRead:** Volatile data can be read but not modified during the transaction. New data can be added during the transaction.
- Serializable:** Volatile data can be read but not modified during the transaction. No new data can be added during the transaction.

Part II: Programming Windows Communication Foundation

- ❑ **Snapshot:** Volatile data can be read. Prior to modifying any data, the transaction looks to see if the data has been changed by another transaction. An error is raised if the data has been changed. The process allows the current transaction access to previously committed data.
- ❑ **Unspecified:** A different isolation level is being used other than the one specified, and the level is undeterminable.

Volatile data is defined as data that will be affected by a transaction. When creating a transaction you can also specify the isolation level to be applied to that transaction. The level of access that other transactions have to volatile data before your transaction completes is called the isolation level. The isolation level you specify on your transaction, when your transaction is created, determines the access to the volatile data of other transactions before your transaction is finished.

You can see by taking a look at the preceding list that `ReadUncommitted` provides the lowest isolation level. This level allows multiple transactions to read and modify data at the same time, leading to possible data corruption.

The highest level is `Serializable`. Though this level offers the highest protection, it does require that one transaction complete before another transaction begins.

The isolation level you select can have an impact on performance in your application. There are trade-offs that you will probably need to ask yourself about in order to decide which level to select. Do you want multiple transactions operating simultaneously on a data store, which offers little or no protection against data corruption but is the faster performing? Or, does data integrity mean more to you with each transaction completing prior to other transactions being given the OK to operate on the data? These are the questions you need to ask yourself prior to selecting the appropriate isolation level for your environment.

The following example sets the `TransactionIsolationLevel` property to a value of `ReadCommitted`. While the current transaction is reading/modifying data, other transactions can read the volatile data but cannot modify it:

```
[ServiceBehavior(TransactionIsolationLevel=System.Transactions.IsolationLevel.ReadCommitted)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *Unspecified*.

TransactionTimeout

The `TransactionTimeout` property specifies the amount of time, in the form of a `TimeSpan` object, that a transaction has to complete. Here also there are trade-offs in the amount of time given to a transaction to complete or abort. A higher value might result in fewer timeout exceptions but might not be too pleasing to the user, who might think the system is not responding. On the other hand, a lower timeout value might result in more timeout exceptions but would certainly let the user, as well as the developer, know what is happening.

The following example sets the `TransactionTimeout` property to a value of `1`. The transaction must complete in one minute or be automatically aborted and rolled back:

```
[ServiceBehavior(TransactionTimeout="00:01:00")]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *0* (zero).

UseSynchronizedContext

The `UseSynchronizedContext` property specifies whether or not to use the current synchronization context. If the value of this property is set to *true*, all calls to the service will run on the thread specified by the `SynchronizationContext`. The `SynchronousContext` object is used to determine on which thread the service operation will be run.

The following example sets the `UseSynchronizedContext` property to a value of *true*. All calls to the service must run on the same thread that is specified by the `SynchronizationContext`:

```
[ServiceBehavior(UseSynchronizationContext=true)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *true*.

ValidateMustUnderstand

The `ValidateMustUnderstand` property specifies whether or not the application enforces the SOAP *MustUnderstand* header processing. A value of *true* for this property forces the application to perform the SOAP header `MustUnderstand` processing. A value of *false* indicates that no header processing will be performed on incoming messages. Setting the value to *false* means that the application must manually check for headers tagged with `MustUnderstand="true"`. If at least one of the headers is not understood, a fault message must be returned.

The following example sets the `ValidateMustUnderstand` property to a value of *false*. No `MustUnderstand` header processing will occur on incoming messages:

```
[ServiceBehavior(ValidateMustUnderstand=false)]
public class ServiceClass : IServiceClass
{
    ...
}
```

The default value for this property is *true*.

A few properties were not listed or discussed, such as `Name` and `Namespace`, because those properties are common with the other attributes discussed in previous chapters.

The next section discusses the behavior attributes that can be applied to the service operations.

OperationBehavior Attribute

The `OperationBehavior` attribute comes from the `System.ServiceModel.OperationBehaviorAttribute` class and specifies the execution behavior of a service operation. As you read earlier, this attribute is applied directly to the methods of the service implementation class. Like the other attributes, these properties are optional because each property does have a default behavior.

The `[OperationBehavior]` attribute contains the following properties:

- `AutoDisposeParameters`
- `Impersonation`
- `ReleaseInstanceMode`
- `TransactionAutoComplete`
- `TransactionScopeRequired`

These properties are discussed in detail next.

AutoDisposeParameters

The `AutoDisposeParameters` property specifies whether the parameters for the associated method are automatically disposed. Setting the value of this property to *false* will cache the parameter resources and not dispose of them. The service handles the disposing of all disposable parameters such as input, output, and reference parameters.

The following example sets the `AutoDisposeParameters` property to a value of *true*. The service will automatically dispose of the two `int` input parameters:

```
[OperationBehavior(AutoDisposeParameters=true)]
public void AddNumber(int number1, int number2)
{
    result = number1 + number2;
}
```

The default value for this property is *true*.

Impersonation

The `Impersonation` property specifies the impersonation behavior for a service operation for the current service instance. This property lets you identify specific methods in which to execute under the Impersonated caller's identity. When this property is used and `Impersonation` is set to either *Allowed* or *Required*, a binding must also be used that supports impersonation.

It is possible for the service to run with higher credentials than those of the client. If a client passes low-privileged credentials to the service, the service will execute using the higher credentials, allowing the client to use resources that it would not normally be able to use.

The modes used by this property come from the `ImpersonationOption` enumeration and the available values are as follows:

- ❑ **Allowed:** Impersonation is performed if credentials are available and `ImpersonateCallerForAllOperations` is *true*.
- ❑ **NotAllowed:** Impersonation is not performed.
- ❑ **Required:** Impersonation is required.

The following example sets the `Impersonation` property to a value of *Allowed*. The service will execute using the credentials that are passed from the client, or use its own if none are supplied by the client:

```
[OperationBehavior(Impersonation=ImpersonationOption.Allowed)]
public void AddNumber(int number1, int number2)
{
    result = number1 + number2;
}
```

The default value for this property is *NotAllowed*.

ReleaseInstanceMode

The `ReleaseInstanceMode` property is used to determine the recycle point of the service object during the course of an operation. This property utilizes the `InstanceContextMode` property in that the default behavior of this property is to recycle the service object according to the value of `InstanceContextMode` property.

When dealing with threading, WCF makes no guarantees as to the state of the threads. To be safe, set the `InstanceContextMode` property to *PerCall*. This will make sure that you get a new object when your service runs.

When using transactions, this property comes in handy to ensure that volatile data is cleaned up before the method call is processed.

The modes used by this property come from the `ReleaseInstanceMode` enumeration, and the available values are as follows:

- ❑ **AfterCall:** Recycles the object subsequent to the completion of the operation.
- ❑ **BeforeAndAfterCall:** Recycles the object prior to calling the operation and subsequent to the completion of the operation.
- ❑ **BeforeCall:** Recycles the object prior to calling the operation.
- ❑ **None:** Recycles the object according to the `InstanceContextMode`.

The following example sets the `ReleaseInstanceMode` property to a value of *AfterCall*. The service object will be recycled after the successful completion of the operation:

```
[OperationBehavior(ReleaseInstanceMode=ReleaseInstanceMode.AfterCall)]
public void AddNumber(int number1, int number2)
{
    result = number1 + number2;
}
```

The default value for this property is *None*.

TransactionAutoComplete

The `TransactionAutoComplete` property specifies whether to commit the current transaction automatically. If the value of this property is set to *true* and no unhandled exceptions are found, the current transaction is automatically committed. If exceptions do occur, the transaction is cancelled. If the property is set to a value of *false*, the transaction will need to be completed or cancelled directly via code.

The following example sets the `TransactionAutoComplete` property to a value of *true*. All transactions will be automatically committed if no exceptions are found:

```
[OperationBehavior(TransactionAutoComplete=true)]
public void AddNumber(int number1, int number2)
{
    result = number1 + number2;
}
```

The default value for this property is *true*, and is defaulted to this value for a reason. As stated, a value of *false* means that the transaction will need to be completed or aborted manually through code. Letting the system automatically commit the current transaction also lets the system deal with related tasks such as cancelling and rolling back the transaction if an exception occurs. Setting this property to a value of *false* means that the developer will need to deal with those issues as well as others, such as dealing with exceptions. It is best to let the system handle the transactions unless there are specific reasons you want to manually control the transaction.

TransactionScopeRequired

The `TransactionScopeRequired` property specifies whether the associated method requires a transaction scope. If a flowed transaction is available, the method will execute within that transaction, otherwise a new transaction is created and used for that method execution. A flowed transaction is a situation in which a transaction id is passed over the wire and used on the receiving side to perform work by enlisting in the corresponding transaction and executing within the scope of that transaction.

The following example sets the `TransactionScopeRequired` property to a value of *true*. All calls to the service must run on the same thread that is specified by the `SynchronizationContext`:

```
[OperationBehavior(TransactionScopeRequired=true)]
public void AddNumber(int number1, int number2)
{
    result = number1 + number2;
}
```

The default value for this property is *false*.

Using Configuration to Specify Behaviors

The previous two sections showed how to apply service and operation behaviors via code. It is also possible to specify behaviors via configuration. The following configuration file, taken from one of the examples in Chapter 7, shows how to set service throttling:

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

```

    <services>
      <service name ="WCFService.ServiceClass"
behaviorConfiguration="metadataSupport">
        <host>
          <baseAddresses>
            <add baseAddress="net.pipe://localhost/WCFService"/>
            <add baseAddress="net.tcp://localhost:8000/WCFService"/>
            <add baseAddress="http://localhost:8080/WCFService"/>
          </baseAddresses>
        </host>
        <endpoint address="tcpmex"
          binding="mexTcpBinding"
          contract="IMetadataExchange"/>
        <endpoint address="namedpipemex"
          binding="mexNamedPipeBinding"
          contract="IMetadataExchange"/>
        <endpoint address=" " binding="wsDualHttpBinding"
contract="WCFService.IServiceClass"/>
        <!--<endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange"/>-->
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="metadataSupport">
          <serviceDebug includeExceptionDetailInFaults="true"/>
          <serviceMetadata httpGetEnabled="false" httpGetUrl=" " />
          <serviceThrottling maxConcurrentCalls="10" maxConcurrentInstances="5"
maxConcurrentSessions="5"/>
          <serviceSecurityAudit auditLogLocation="Application"
suppressAuditFailure="false"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

The `<serviceDebug>` element contains a number of properties that specify debugging and help information for a Windows Communication Foundation service. In the preceding example, the `IncludeExceptionDetailInFaults` attribute specifies to include exception information in the detail of the SOAP faults, just like the property defined earlier.

Throttling

Windows Communication Foundation lets you set throttling limits on your service. Throttling is the concept of limiting the amount of work a service can accept. You can see in the previous configuration file that limits can be set on concurrent calls, concurrent instances, and concurrent sessions.

Throttling is controlled by adding the `<serviceThrottling>` element within the `<behavior>` element in the service or host application configuration file. You can then specify the attributes of the `<serviceThrottling>` element to define the throttling behavior of your service.

The `maxConcurrentCalls` attribute lets you specify the maximum number of concurrent calls for a service. If the maximum number of concurrent calls has been met when a new call is placed, the call is

Part II: Programming Windows Communication Foundation

queued and will be processed when the number of concurrent calls is below the specified maximum number. The default is 16 but a value of 0 is equal to `Int32.MaxValue`.

The `maxConcurrentInstances` attribute lets you specify the maximum number of concurrent service instances. If a request for a new instance is received and the maximum number has already been reached, the request is queued up and will be completed when the number of instances is below the specified maximum. The default value is `Int32.MaxValue`.

The `maxConcurrentSessions` attribute lets you specify the sessions that service can have. More specifically, it specifies the maximum number of connections to a single service. In reality, it will accept more than the specified limit, but the key is that only the channels below the specified limit will be active. The default value for this attribute is 10, but setting this value to 0 is equal to setting it to `Int32.MaxValue`.

InstanceContext

This chapter has mentioned services and `InstanceContext` a number of times, so it is only fitting to spend a few paragraphs discussing the differences between the two. It was stated earlier that there is not a direct connection between the service object and the communication channel; therefore the lifetime of the service object is completely independent of the client-service channel.

This is where the `InstanceContext` comes in. The `InstanceContext` class is a runtime object whose sole purpose is to bind a channel to an instance of the service object. This is a very useful feature because it allows for the separation of the channel and service. The lifetime of the channel is now disconnected from the lifetime of the service, each being controlled and handled independently.

Why would you want to do this? By managing them separately, you can continue to have open, and maintain, a secure and reliable channel while simultaneously disposing of a service when your transaction or operation is finished. You can keep the channel open and bind it to another instance of a service when you are ready to use it.

By default, when a message is received by the service, a new `InstanceContext` is created. However, this is controlled by setting the `InstanceContextMode` property, discussed a few pages ago. This property controls the exact lifetime of the service object. This property gives you the flexibility to maintain state across multiple calls from the same client, multiple calls from multiple clients, or where state does not need to be maintained over calls at all.

Handling Exceptions

As much as developers would like to think that they write error-free code, all applications will have their fair share of errors that will throw exceptions. Windows Communication Foundation services are no “exception” (pun intended). Therefore, it would be wise to build error handling into your WCF service.

It was stated earlier that Windows Communication Foundation utilizes two types of error systems (Exception objects and SOAP fault messages). Processing errors are represented as Exception objects in managed applications, and in SOAP applications error information is communicated via SOAP fault messages. Therefore, in WCF any managed exception must be converted to a SOAP fault message to be sent over the wire back to the client.

Two types of SOAP faults can be sent back to the client:

- ❑ **Declared:** The operation has a `[FaultContract]` attribute, which specifies a custom SOAP fault type.
- ❑ **Undeclared:** Faults that are not specified in the operation contract.

It is recommended that service operations declare their faults via the `[FaultContract]` attribute. This formally specifies all SOAP faults that a client should expect to receive throughout the course of normal operation. To maximize the level of security, only the information that a client needs to know regarding the exception should be returned in the SOAP fault.

Declared SOAP faults come in handy for architecting interoperable, distributed applications that are stout enough to support any environment and situation. Yet, there may be times when undeclared SOAP faults may be more useful, such as for debugging purposes where unexpected situations can occur and information can be sent back to the client.

One of the ways you can get internal service operation exception information is by setting the `IncludeExceptionDetailInFault` property to true. This property is available on the `ServiceBehaviorAttribute` and `ServiceDebugBehavior` classes and allows the clients to get sensitive information specific to internal service operations, such as personal identifiable information. This information helps in debugging a service application.

The following code snippet shows how to enable `IncludeExceptionDetailInFault` using a configuration file:

```
<behaviors>
  <serviceBehaviors>
    <behavior>
      <serviceDebug includeExceptionDetailInFaults="true"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Due to the amount and type of data returned by setting the `IncludeExceptionDetailInFault` to true, it is recommended that this property be set to true only during development and not be enabled during production.

FaultException

The `FaultException` class represents a SOAP fault and should be used in a service to create an untyped, or undeclared, fault to return to the client, typically for debugging purposes. From the client side, `FaultException` objects returned from the service can be caught and inspected to determine whether an unknown or generic fault has occurred.

The `FaultException` extends the `CommunicationException` object so it is important to catch `FaultExceptions` before catching any `CommunicationException` faults.

Fault exceptions should be thrown when you want the stream to be passed to the constructor, which are then made available to the client and can be called using the `FaultException.ToString` method.

Part II: Programming Windows Communication Foundation

The following example illustrates how to use a try/catch block to catch and manage exceptions thrown from the service:

```
Try
    // call the operations of the service
    ...
catch (FaultException fe)
{
    // do something with the exception
}
```

There are two reasons why you would want to use the `FaultException` class:

- For debugging purposes when a SOAP fault can be sent to the client from a service
- When faults are not part of the service contract and you want to catch SOAP faults on the client

FaultContract Attribute

Although there might be times when the `FaultException` class will be beneficial, such as in the two scenarios just given, the general recommendation is to use the `FaultContract` attribute to return strongly typed SOAP faults.

SOAP-based applications such as Windows Communication Foundation communicate error information using SOAP fault messages. Exception information must be converted from exceptions into SOAP faults prior to sending the information to the client.

Use the `FaultContract` attribute to specify one or more specific exception conditions. These conditions are then added as explicit SOAP fault messages to the WSDL (Web Service Description Language) description, which are returned to the client by the operation. When returning faults, you have two options:

- Using the default service exceptions behaviors
- Specifying how exceptions are mapped to fault messages

The second option, specifying how exceptions are mapped to fault messages, requires that you decide the conditions under which a client needs to be informed of errors. Once these conditions are identified a custom SOAP fault can be defined with the operation tagged as returning a SOAP fault. For example, the `BookOrder` operation might return information to customers informing them that the quantity of books they have ordered is not available or that their credit card could not be processed.

The fault message can contain very sensitive information so there are a few recommendations that should be followed. First, only send information in a SOAP fault back to the client that the client needs to know. Second, set the `ProtectionLevel` property. Failure to follow these recommendations will increase security problems.

This attribute has some of the same properties as the other attributes discussed previously:

- Action:** Defines the action of the fault message.
- DetailType:** Returns the type of the detail object serialized in the message.

- ❑ **Name:** Defines the name of the fault message.
- ❑ **Namespace:** Defines the namespace of the fault message.
- ❑ **HasProtectionLevel:** Specifies the level of protection of the fault message.

As stated earlier, the `IncludeExceptionDetailsInFaults` property can be used to help in debugging. This can be set through the configuration file, and when set, automatically returns exception information to the client, which appears as `FaultException` exceptions.

The following example illustrates how to use the `FaultContract` attribute to return a SOAP fault. The first step is to configure the service appropriately by adding the `FaultContract` attribute to those operations that you want to return SOAP faults:

```
[ServiceContract]
public interface IWCFService
{
    [OperationContract]
    [FaultContract(typeof(OrderFault), ProtectionLevel=ProtectionLevel.EncryptAndSign)]
    String BookOrder(string ISBN, int Quantity)
}

[DataContract]
public class OrderFault
{
    private string info;

    public OrderFault(string Message)
    {
        This.info = Message
    }

    [DataMember]
    public string msg
    {
        Get { return this.info; }
        Set { this.info = value; }
    }
}

class WCFService : IWCFService
{
    public string BookOrder(string ISBN, int Quantity)
    {
        int BooksOnHand = 10;
        //check book quantity vs. order quantity
        If (Quantity <= BooksOnHand)
            return "Order placed"
        else
            throw new FaultException<OrderFault>(new OrderFault("You ordered too many
books"));
    }
}
}
```

Part II: Programming Windows Communication Foundation

On the client side the SOAP fault is received and dealt with, as shown here:

```
private void Form1_Load(object sender, EventArgs e)
{
    TCP.ServiceClassClient client = new
        WCFClientApp.TCP.ServiceClassClient("NetTcpBinding_IServiceClass");
    try
    {
        client.BookOrder("123456", 11);
    }
    catch (FaultException<OrderFault> of)
    {
        MessageBox.Show(of.Detail.Message);
    }
    catch (FaultException unknownFault)
    {
        MessageBox.Show(unknownFault.Message);
    }
}
```

Although this code is a very simple code snippet, it provides the necessary information to return a SOAP fault and have it processed appropriately by the client.

Programming Example

This last section walks you through building a service that uses several of the behaviors you learned about earlier in the chapter. Open the WCFService project you have been working with so far and replace the existing code with the following code:

```
using System;
using System.Transactions;
using System.ServiceModel;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.IO;

namespace WCFService
{
    [ServiceContract(SessionMode = SessionMode.Required)]
    public interface IServiceClass
    {
        [OperationContract]
        string GetText();

        [OperationContract]
        int MultiplyNumbers(int firstvalue, int secondvalue);
    }

    [ServiceBehavior(AutomaticSessionShutdown=true,
        ConcurrencyMode=ConcurrencyMode.Single,
        IncludeExceptionDetailInFaults=false,
```

```

        InstanceContextMode=InstanceContextMode.PerSession,
        UseSynchronizationContext=true)]
public class ServiceClass : IServiceClass
{
    [OperationBehavior(AutoDisposeParameters=true,
        TransactionAutoComplete=true,
        TransactionScopeRequired=true)]
    string IServiceClass.GetText()
    {
        StreamReader sw = new StreamReader(@"c:\wrox\WCFServiceTest.txt");
        return sw.ReadLine();
        //return "Hello World";
    }

    [OperationBehavior(AutoDisposeParameters = true,
        TransactionAutoComplete = true,
        TransactionScopeRequired = true)]
    int IServiceClass.MultiplyNumbers(int firstvalue, int secondvalue)
    {
        return firstvalue * secondvalue;
    }
}
}

```

Take a quick look at the service code. You should notice that the service interface definition is very similar to other examples. What is different is that this class implements the service interface. The class itself is annotated with the `[ServiceBehavior]` attribute, and the corresponding methods are annotated with the `[OperationBehavior]` attribute.

In this example, several properties are applied to the service behavior itself and each operation behavior. For example, the `AutomaticSessionShutdown` property is set to `true`, which automatically closes the session when the client closes its output session.

On the operations themselves, the `TransactionAutoComplete` property is set to `true`, which tells the service to automatically commit the transaction.

Compile the service to make sure no errors are found. The next step is to modify the host application. What needs to be modified here is the configuration file, and the only reason this needs to be modified is because the last example dealt with duplex communication. Because this example does not use duplex communication you need to set it back. Simply change the binding property of the highlighted line from `wsDualHttpBinding` to `wsHttpBinding`:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name ="WCFService.ServiceClass"
        behaviorConfiguration="metadataSupport">
        <host>
          <baseAddresses>
            <add baseAddress="net.pipe://localhost/WCFService"/>
            <add baseAddress="net.tcp://localhost:8000/WCFService"/>

```

Part II: Programming Windows Communication Foundation

```
        <add baseAddress="http://localhost:8080/WCFService"/>
    </baseAddresses>
</host>
<endpoint address="tcpmex"
    binding="mexTcpBinding"
    contract="IMetadataExchange"/>
<endpoint address="namedpipemex"
    binding="mexNamedPipeBinding"
    contract="IMetadataExchange"/>
<endpoint address="" binding="wsHttpBinding"
contract="WCFService.IServiceClass"/>
    <!--<endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange"/>-->
</service>
</services>
<behaviors>
    <serviceBehaviors>
        <behavior name="metadataSupport">
            <serviceMetadata httpGetEnabled="false" httpGetUrl=""/>
        </behavior>
    </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>
```

Compile the host application and press F5 to start the service.

The next step is to modify the client. Because the last example dealt with duplex communication, the client code behind the form and the configuration file contained endpoint definition information dealing with duplex communication. Both need to be modified to operate correctly.

The easiest way to do that is to delete the current service references and re-add them. However, if you feel that the stars are aligned in your favor and want to try to manually modify the configuration file, go right ahead. The rest of you can delete the service references and re-add them. The information to re-add them is the same as past examples, so that won't be covered here. Remember that you will need the host application running to add the service references.

The following client code, however, assumes that you name the service references TCP and NamedPipe and, given the endpoint definitions in the host application configuration file, specify the service URIs as follows:

```
net.pipe://localhost/WCFService/namedpipemex
```

and

```
Net.tcp://localhost:8000/WCFService/tcpmex
```

Once you have your service references added, change the caption of button1 to "Get Text" and change the caption of button2 to "Multiply." Next, modify the code behind the form as follows:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
```

```
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace WCFClientApp
{
    public partial class Form1 : Form
    {
        private int _Selection;
        private int val1 = 5;
        private int val2 = 5;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            radioButton1.Checked = true;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            switch (_Selection)
            {
                case 0:
                    TCP.ServiceClassClient client = new
WCFClientApp.TCP.ServiceClassClient("NetTcpBinding_IServiceClass");
                    textBox1.Text = client.GetText();
                    break;

                case 1:
                    NamedPipe.ServiceClassClient client1 = new
WCFClientApp.NamedPipe.ServiceClassClient("NetNamedPipeBinding_IServiceClass");
                    textBox1.Text = client1.GetText();
                    break;

                case 2:
                    break;
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            switch (_Selection)
```

Part II: Programming Windows Communication Foundation

```
        {
            case 0:
                TCP.ServiceModelClient client = new
                    WCFClientApp.TCP.ServiceModelClient("NetTcpBinding_IServiceClass");
                textBox2.Text = client.MultiplyNumbers(val1, val2).ToString();
                break;

            case 1:
                NamedPipe.ServiceModelClient client1 = new
                    WCFClientApp.NamedPipe.ServiceModelClient("NetNamedPipeBinding_IServiceClass");
                textBox2.Text = client1.MultiplyNumbers(val1, val2).ToString();
                break;

            case 2:
                break;
        }
    }

    private void radioButton1_CheckedChanged(object sender, EventArgs e)
    {
        _Selection = 0;
        textBox1.Text = "";
        textBox2.Text = "";
    }

    private void radioButton2_CheckedChanged(object sender, EventArgs e)
    {
        _Selection = 1;
        textBox1.Text = "";
        textBox2.Text = "";
    }
}
}
```

Compile the client application and press F5 to run the client application. You should be familiar with this form by now, so you can select the transport you want to use and click the appropriate buttons.

Summary

The purpose of this chapter was to provide you with an in-depth look at concepts that are more specific to services. This chapter began by spending a few pages providing an overview of WCF services focusing on how service contracts and their associated endpoints are defined. This information was covered in detail in the past several chapters but provided the basis for the rest of this chapter.

Service behaviors exist so that you can modify the runtime behavior of the service to suit the needs of your application. Therefore it is important that you understand the available options that you can apply to your service in which to control the behavior of your service. Thus, a large portion of this chapter

