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

Chapter 7: Clients

ISBN-10: 0-470-08984-9
ISBN-13: 978-0-470-08984-2

Copyright of Wiley Publishing, Inc.
Posted with Permission

7

Clients

Up until now this book has mainly focused WCF service implementations. The last three chapters have discussed the components necessary to build a Windows Communication Foundation service. Chapter 4 discussed addresses, Chapter 5 discussed bindings, and Chapter 6 discussed contracts. Each of these is essential in building a successful service. It is time, however, to change the focus and take a good look at the client, the piece of the equation that utilizes everything you have learned so far.

This chapter covers the following topics:

- Client architecture
- Client communication patterns
- Creating client code
- Defining client bindings and endpoints

Client Architecture

A Windows Communication Foundation client is an application used to invoke functionality exposed by a service. The client application will communicate with the service via a service endpoint. In order to do that the client needs to know several pieces of information about the service, such as the address at which the endpoint is communicating, the binding the service is using, and the service contract. Each of these elements has been discussed in the previous chapters.

A good look under the hood of a client will reveal some important things about its makeup. One of the things you will find is a channel built on binding settings specified in the configuration file. Just to be clear, these bindings are the same bindings that have been discussed in the past couple of chapters. These bindings allow the client and service to appropriately and effectively communicate.

Excerpted from Professional WCF Programming: .NET Development with the Windows(r) Communication Foundation, Wrox Press, www.wrox.com

The second thing you will find is the implementation of the IClientChannel interface. This interface defines the operations that allow developers to control channel functionality, such as closing the client session and disposing of the channel. It exposes the methods and functionality of the SystemServiceModel.ChannelFactory class.

Lastly, you will find a generated service contract, which provides the functionality that turns client method calls into outgoing messages, and turns incoming messages into information that your client application can readily use in the form of return values and output parameters.

Clients communicate with service endpoints through a proxy, as shown in Figure 7-1. A proxy class is what the client manipulates to communicate to a service. This communication takes place via a channel. Once that proxy (and channel) is created, the client can access any exposed methods (service operations) on that endpoint.

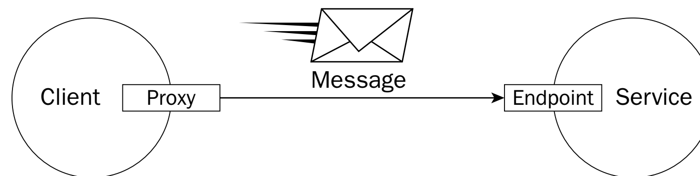

Figure 7-1

There are two ways to create client proxies, both of which are discussed in this chapter:

- The first method is to create the proxy from generated code, that is, code that is automatically generated from the metadata provided by the service. This is done by using the Service Model Metadata Utility Tool Svcutil.exe. The Svcutil utility creates an instance of the derived class ClientBase that is then accessible to the client.
- The second method is by creating the proxy dynamically through code using a ChannelFactory object. This is provided by the System.ServiceModel.ChannelFactory class. This method allows for greater control by the developer, such as creating new client channels from an existing channel factory.

Client Objects

A Windows Communication Foundation client must contain two base object interfaces, the ICommunicationObject interface and the IExtensibleObject interface.

ICommunicationObject

The ICommunicationObject interface is one of the core components that define basic communication object functionality. The responsibility of this object is to define the contract for the basic state for all communication objects in the system; for example, is the communication object opened or closed, or in the process of opening or closing. These objects include channels, listeners, dispatchers, factories, and service hosts.

A state transition is the transition from one state to another; for example, the communication channel transitioning from an “opening” state to an “open” state.

 Excerpted from Professional WCF Programming: .NET Development with the Windows(r) Communication Foundation, Wrox Press, www.wrox.com

This interface defines a set of methods for initiating state transitions. These methods are:

- **Open:** Causes a communication object to transition from the Created state to the Opened state.
- **Close:** Causes a communication object to transition from its current state to the Closed state.
- **Abort:** Causes a communication object to instantly transition from its current state into the Closed state.

This interface also defines notification events for state transitions. These include:

- **Opening:** This event is fired when the communication object transitions from Created to Opened, such as when the Open or BeginOpen method is invoked.
- **Closing:** This event is fired when the communication object transitions from Opened to Closed, such as when the Close or BeginClose method is invoked.
- **Opened:** This event is fired when the communication object is finished transitioning from Opening to Opened.
- **Closed:** This event is fired when the communication object is finished transitioning from Closing to Closed.
- **Faulted:** This event is fired when the communication object enters the Faulted state.

This interface also includes a set of methods that define asynchronous versions of the Open and Close methods:

- **BeginOpen:** Begins an asynchronous operation to open a communication object.
- **BeginClose:** Begins an asynchronous operation to close a communication object.
- **EndOpen:** Completes an asynchronous operation to open a communication object.
- **EndClose:** Completes an asynchronous operation to close a communication object.

This interface has a single State property, of type CommunicationState, which is used to return the current state of the object.

When an ICommunicationObject is instantiated, its default state is Created. This is not readily intuitive because many assume that it's defaulted to Opened. While in the Created state, the ICommunicationObject can be configured but it cannot send or receive communication. For example, any of the events listed earlier can be registered. Once the object is in the Open state, it can send and receive messages, but it no longer can be configured.

The Open method must be called for the object to enter the Opened state. The object will stay in the Open state until its transition to the Closed state is finished. The Close method allows any unfinished work to be completed before transitioning to the Closed state. The Abort method does not exit gracefully, meaning all unfinished work is ignored. The Abort method can also be used to cancel any and all outstanding operations, and that includes outstanding calls to the Close method. Keep in mind that the Abort method will cause any unfinished work to be cancelled. Use transactions, discussed in Chapter 9, if you want work grouped as a single unit.

IExtensibleObject

The IExtensibleObject interface provides extensible behavior in the client, meaning that it enables the object to be involved in custom behaviors. In WCF, the extensible object pattern is used to add new functionality to existing runtime classes, thereby extending current components, as well as adding new state features to an object.

This interface exposes a single property to provide this functionality. The Extensions property, of type IExtenstionCollection, is used to return a collection of extension objects that can then be used to extend the existing runtime classes.

Client Channels

Windows Communication Foundation clients contain two base channel interfaces, the IClientChannel interface and the IContextChannel interface.

IClientChannel

The IClientChannel interface defines the extended ClientBase channel operations. It contains a number of methods and properties that can be used to define the outbound request channel behavior and the request/reply channel behavior of the client application.

For example, the AllowInitializationUI property can be used to tell WCF to open a channel without an explicit call to open it. There are also a small handful of methods that you can use to return the credential information.

The IClientChannel interface inherits from the IContextChannel interface (discussed next) as well as the ICommunicationObject and IExtensibleObject interfaces discussed earlier. This allows client applications to have access to client-side runtime functionality directly.

IContextChannel

The IContextChannel interface defines the session state of the channel. This information includes the SessionId, Input and Output session, as well as the local and remote endpoints that are currently communicating with the client in the session. This information is provided by the following properties:

- **InputSession:** Returns the input session for the channel.
- **OutputSession:** Returns the output session for the channel.
- **LocalAddress:** Returns the local endpoint for the channel.
- **RemoteAddress:** Returns the remote address connected with the channel.
- **SessionId:** Returns the current session identifier.
- **OperationTimeout:** Returns, or sets, the time in which the operation has to complete. If the operation does not complete in the specified time, an exception is thrown.
- **AllowOutputBatching:** Tells WCF to store messages before handing them off to the transport.

There are two AllowOutputBatching properties, one that is applied at the channel level and one that is applied at the message level. Setting the AllowOutputBatching at the message level does not override

 Excerpted from Professional WCF Programming: .NET Development with the Windows(r) Communication Foundation, Wrox Press, www.wrox.com

the channel-level AllowOutputBatching property. If the message-level AllowOutputBatching property is set to true, the message will be sent immediately even if the AllowOutputBatching property is set to true at the channel level.

Keep in mind that the AllowOutputBatching property can affect the performance of the system because you are telling WCF to store outgoing messages in a buffer and send them out with other messages as a group. Your message delivery needs will affect how this setting is configured. Setting this property to true means that message throughput and delivery is essential to you, and setting it to false will reduce latency.

Channel Factories

It is important that you understand the client objects and client channel objects because both of these utilize the ChannelFactory object. The ChannelFactory object is responsible for creating and supporting all the runtime client invocations.

As stated earlier, you can either create clients on demand using the ChannelFactory or by using the Service Model Metadata Utility svcutil.exe. The svcutil utility automatically generates the handling of the ChannelFactory, but as stated before, creating the channels on demand provides you more control over the creation and handling of the channels. For example, you can repeatedly create a new channel from an existing factory.

The following code illustrates using the ChannelFactory to create a channel to a service by specifying the service contract name:

```
EndpointAddress ea = new EndpointAddress("tcp.net://localhost:8000/WCFService");
BasicHttpBinding bb = new BasicHttpBinding();
WCFClientApp.TCP.IServiceClass client =
    ChannelFactory<IServiceClass>.CreateChannel(bb, ea);
client.PlaceOrder(Val1);
```

In this example, the address and binding were specified in code and passed as parameters to the CreateChannel method.

The following section details the ChannelFactory class, which is used to create and manage channels that are used by the clients to send messages and communicate with service endpoints.

ChannelFactory Class

The following sections list many of the important constructors, properties, and methods of the ChannelFactory class.

Constructors

The ChannelFactory class has a single constructor called ChannelFactory, and it is used to instantiate a new instance of the ChannelFactory class, as illustrated in the previous example. The following code snippet, taken from the previous example, shows the instantiation of the ChannelFactory class:

```
WCFClientApp.TCP.IServiceClass client =
    ChannelFactory<IServiceClass>.CreateChannel(bb, ea);
```

Properties

The following properties are exposed by the ChannelFactory class:

- **Credentials:** Returns the credentials used by the client to communicate with the service endpoint, via the channel created by the factory.
- **Endpoint:** Returns the endpoint that the channel created by the factory connect.
- **State:** Returns the value of the current communication object state.

The use of credentials requires a reference to the System.ServiceModel.Description namespace, which needs to be added via a `using` statement:

```
using System.ServiceModel.Description;
```

Once you have access to the System.ServiceModel.Description namespace, you can configure client and service credentials as well as provide credentials for authenticating on the proxy side. The following example illustrates how to provide credentials for proxy side authentication when creating a channel:

```
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

TCP.IServiceClass channel = factory.CreateChannel();

ClientCredentials cc = new ClientCredentials();
cc.UserName.UserName = "scooter";
cc.UserName.Password = "wcfrocks";
factory.Credentials = cc;
```

The following example illustrates how to use the Endpoint property to return the service endpoint on which the channel was produced:

```
 WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

Console.WriteLine(factory.Endpoint);
```

Methods

The following methods are exposed by the ChannelFactory class. The BeginClose, BeginOpen, EndClose, and EndOpen methods are used in asynchronous communication.

- **Abort:** Immediately transitions the communication object from its current state into the *closing* state.
- **BeginClose:** Begins an asynchronous operation to close the current communication object.
- **BeginOpen:** Begins an asynchronous operation to open a communication object.

 Excerpted from Professional WCF Programming: .NET Development with the Windows(r) Communication Foundation, Wrox Press, www.wrox.com

- **Close:** Transitions the object from its current state into the *closed* state.
- **EndClose:** Finishes the asynchronous *close* on the current communication object.
- **EndOpen:** Finishes the asynchronous *open* on the current communication object.
- **Open:** Transitions the object from the *created* state into the *opened* state.

The following can be used to explicitly open a channel:

```
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

TCP.IServiceClass channel = factory.CreateChannel();

factory.Open();

channel.DoSomething();
```

The following can be used to implicitly open a channel:

```
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

TCP.IServiceClass channel = factory.CreateChannel();

channel.DoSomething();
```

The difference between the two previous examples is that by explicitly opening the channel you have more control over the creation and management of the channel.

Be sure to close the channel factory when you are done with it:

```
factory.close();
```

CreateChannel Method

The CreateChannel method creates a channel of a specific type to a specified endpoint. Typically in code you will create a channel that is configured with a specific binding and endpoint. In most of the examples you have seen, a channel has been created to an endpoint that has been configured with a specific binding, as shown here:

```
EndpointAddress ea = new EndpointAddress("tcp.net://localhost:8000/WCFService");
BasicHttpBinding bb = new BasicHttpBinding();
ChannelFactory<TCP.IServiceClass> cf = new
    ChannelFactory<IServiceClass>("BasicHttpBinding_IServiceClass");
TCP.IServiceClass ch = cf.CreateChannel(bb, ea);
textbox1.Text=ch.AddNumbers;
```

The CreateChannel method takes a number of overloads that can be used to create the channel as described in the following table.

Overload	Description
CreateChannel()	Creates a channel of an IChannel type.
CreateChannel(EndpointAddress)	Creates a channel used to send messages to the specified endpoint address.
CreateChannel(String)	Creates a channel used to send messages to a service whose endpoint is configured in a specified way.
CreateChannel(Binding, EndpointAddress)	Creates a channel used to send messages to a service endpoint at the specified endpoint and configured with the specified binding.
CreateChannel(EndpointAddress, Uri)	Creates a channel used to send messages to a service endpoint at the specified endpoint through the specified transport address.
CreateChannel(Binding, EndpointAddress, Uri)	Creates a channel used to send messages to a service endpoint at the specified endpoint and configured with the specified binding and transport address.

Asynchronous communication is discussed later on in this chapter.

Client Communication Patterns

Now that you understand how channels are created and function, this section describes the different types of communication that can take place between the client and the service endpoint.

One-Way

One-way communication is just that, it is communication in a single direction. That direction flows from the client to the service. No reply is sent from the service, and the client does not expect a response. In this scenario, the client sends a message and continues execution.

Figure 7-2 illustrates a one-way communication. The client sends a message to the service, and execution takes place on the service. No response is sent back to the client from the service.

 Excerpted from Professional WCF Programming: .NET Development with the Windows(r) Communication Foundation, Wrox Press, www.wrox.com

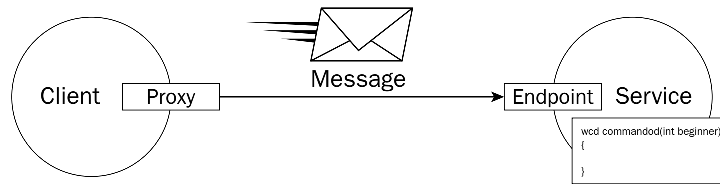

Figure 7-2

Because there is no response from the service in one-way communication, any errors generated by the service during the processing of the message are not communicated back to the client, therefore the client has no idea if the request was successful or not.

For a one-way, single direction communication, the `IsOneWay` parameter on the `[OperationContract]` is set to True. This tells the service that no response is required. The following code example illustrates setting a one-way communication:

```
[ServiceContract]
public interface IServiceClass
{
   [OperationContract(IsOneWay=true)]
   string InitiateOrder();

   [OperationContract]
   BookOrder PlaceOrder(BookOrder request);

   [OperationContract(IsOneWay=true)]
   string FinalizeOrder();
}
```

In this example, the service contains three available operations, two of which are defined as one-way operations. The InitiateOrder and FinalizeOrder operations are defined as one-way operations, whereas the PlaceOrder operation is not. When the client calls the InitiateOrder service operation, it will immediately continue processing without waiting for a response from the service. However, when the client calls the PlaceOrder service operation, it will wait for a response from the service before continuing.

Request-Reply

Request-reply communication means that when the client sends a message to the service, it expects a response from the service. Request-reply communication also means that no further client execution takes place until a response is received from the service.

Figure 7-3 illustrates a request-reply communication. The client sends a message to the service, the service operation takes place, and a responding message is sent back to the client. Further client execution is paused until the responding message is received by the client.