WCF Bindings In Depth

Aaron Skonnard

Code download available at: ServiceStation2007_07.exe (162 KB)
Browse the Code Online

Contents

The Windows® Communication Foundation programming model makes it easy for developers to configure services with a variety of wire formats and message protocols. The binding is the secret sauce behind this simplicity. This month I show you in detail how the built-in bindings work, how to configure them, and how to define custom bindings of your own.

What is a Binding?

Ultimately, Windows Communication Foundation is a framework for building services that process XML messages. Windows Communication Foundation allows you to transmit messages using different transport protocols (such as HTTP, TCP, and MSMQ) and using different XML representations (such as text, binary, or MTOM, which is commonly referred to as the message encoding in Windows Communication Foundation). In addition, you can enhance specific messaging interactions with a suite of SOAP protocols, such as the various WS-* specifications around security, reliable messaging, and transactions. All three of these communication concepts—the transport, the message encoding, and the suite of protocols—are central to what happens on the wire at run time.

In Windows Communication Foundation, all of these communication details are handled by the channel stack (see **Figure 1**). Just as its name implies, the channel stack is a stack of channel components that all messages pass through during runtime processing. The bottom-most component is the transport channel. This implements the given transport protocol and reads incoming messages off the wire. The transport channel uses a message encoder to read the incoming bytes into a logical Message object for further processing.

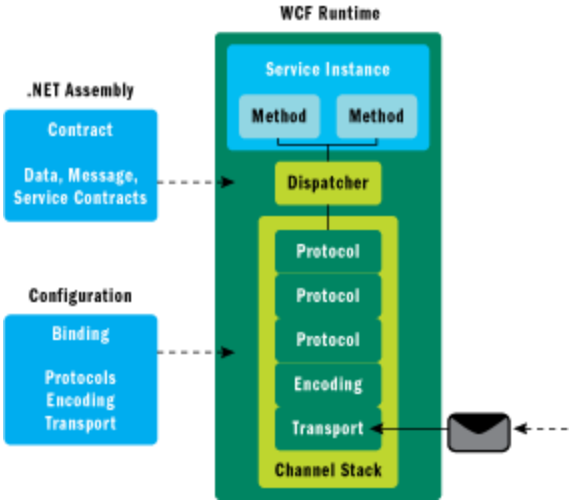

Figure 1 **Bindings and Channel Stacks** (Click the image for a larger view)

After that, the message bubbles up through the rest of the channel stack, giving each protocol channel an opportunity to do its processing, until it eventually reaches the top and Windows Communication Foundation dispatches the final message to your service implementation. Messages undergo significant transformation along the way.

Although the Windows Communication Foundation channel architecture provides a consistent and flexible messaging foundation, it would be too tedious for most developers to work with it directly. When building a channel stack, you have to think carefully about the ordering of the components and whether or not they are compatible with one another. So instead, Windows Communication Foundation provides a simpler abstraction—the concept of endpoints—for indirectly configuring the underlying channel stack.

The idea is that you configure services with one or more endpoints in order to accommodate different communication scenarios. For each endpoint, you specify an address, a binding, and a contract. The address simply specifies the network address where you want to listen for messages while the contract specifies what the messages arriving at the specified address should contain. It's the binding, however, that provides the recipe for building the channel stack needed to properly process the messages. When you load a service, Windows Communication Foundation follows the instructions outlined by the binding description to create each channel stack. The binding binds your service implementation to the wire through the channel stack in the middle (**Figure 1** illustrates this).

This communication model allows you to think about the messaging features you need, in simple terms, while also allowing you to ignore the complexities of actually making it happen in the Windows Communication Foundation runtime environment.

For more background on message processing fundamentals, see my April 2007 column on WCF Messaging Fundamentals.

Built-In Bindings

Within the Windows Communication Foundation programming model, bindings are represented by the System.ServiceModel.Channels.Binding class. All binding classes must derive from this base class. **Figure 2**

summarizes the built-in binding classes that come with Windows Communication Foundation out of the box. Each class derives from Binding and defines a different channel stack configuration through its implementation. These built-in bindings address the most common messaging scenarios that you'll run into today. Transaction flow is always disabled by default—the table shows the protocols that are used by default when you choose to enable transaction flow. Also note that MsmqIntegrationBinding doesn't use a Windows Communication Foundation message encoding—instead it lets you choose a pre-Windows Communication Foundation serialization format.

Figure 2 Windows Communication Foundation Built-In Bindings

Binding Class Name	Transport	Message Encoding	Message Version	Security Mode	Reliable Messaging	Transaction Flow (disabled by default)
BasicHttpBinding	HTTP	Text	SOAP 1.1	None	Not Supported	Not Supported
WSHttpBinding	HTTP	Text	SOAP 1.2 WS-Addressing 1.0	Message	Disabled	WS-AtomicTransactions
WSDualHttpBinding	HTTP	Text	SOAP 1.2 WS-Addressing 1.0	Message	Enabled	WS-AtomicTransactions
WSFederationHttpBinding	HTTP	Text	SOAP 1.2 WS-Addressing 1.0	Message	Disabled	WS-AtomicTransactions
NetTcpBinding	TCP	Binary	SOAP 1.2	Transport	Disabled	OleTransactions
NetPeerTcpBinding	P2P	Binary	SOAP 1.2	Transport	Not Supported	Not Supported
NetNamedPipesBinding	Named Pipes	Binary	SOAP 1.2	Transport	Not Supported	OleTransactions
NetMsmqBinding	MSMQ	Binary	SOAP 1.2	Message	Not Supported	Not Supported
MsmqIntegrationBinding	MSMQ	Not Supported (uses a pre-	Not Supported	Transport	Not Supported	Not Supported

		WCF serialization format)				
CustomBinding	You Decide	You Decide	You Decide	You Decide	You Decide	You Decide

For example, the BasicHttpBinding was designed for scenarios where interoperability is of utmost importance. As a result, BasicHttpBinding uses HTTP for the transport and text for the message encoding. An important aspect of the message encoding is the expected message version, which happens to be SOAP 1.1 for BasicHttpBinding. As for additional protocols, BasicHttpBinding is capable of using transport or message security, but both are disabled by default. The other WS-* protocols are not supported with this binding. As a result, this binding produces a simple channel stack capable of interoperating with any other basic Web services implementation—and this is a great choice when your number-one priority is to make things work.

Then there's WSHttpBinding. This was also designed for interoperability while incorporating the richer Web services protocols for security, reliable messaging, and transactions. As a result, WSHttpBinding also uses HTTP for the transport and text for the message encoding, but it uses SOAP 1.2 along with WS-Addressing 1.0 for the message version—they are needed to carry the additional Web services protocol headers. The binding enables message-based security (via WS-Security and friends) and is capable of supporting WS-ReliableMessaging and WS-AtomicTransactions if you choose to enable them. WSHttpBinding produces a more sophisticated channel stack and will most likely be constrained to enterprise scenarios where integration across frameworks and platforms is required.

And that brings me to NetTcpBinding. Unlike the two HTTP bindings, the various "Net" bindings were not designed for interoperability. In fact, each was designed for optimizing a different communication scenario when you can safely assume you have the Microsoft® .NET Framework 3.0 installed on both sides (this explains why the binding names are prefixed with "Net").

NetTcpBinding uses TCP for the transport, binary for the message encoding, and SOAP 1.2 for the message version. It enables transport security by default and can support transactions if enabled. As you can see, the configuration of this binding focuses on creating a channel stack that will perform better in Windows environments, giving you a great option for replacing your various COM+ and .NET remoting investments.

All you have to do is choose a particular binding based on your communication needs (see **Figure 2**) and Windows Communication Foundation takes care of producing the appropriate channel stack. Programmatically you choose a binding by providing an instance of the desired binding class in your call to ServiceHost.AddServiceHost:

Copy Code

```
using (ServiceHost host = new ServiceHost(typeof(ChatService)))
{
    host.AddServiceEndpoint(typeof(IChat),
        new BasicHttpBinding(), "http://localhost:8080/chat");
    host.AddServiceEndpoint(typeof(IChat),
        new WSHttpBinding(), "http://localhost:8080/chat/secure");
```

```
    host.AddServiceEndpoint(typeof(IChat),
        new NetTcpBinding(), "net.tcp://localhost:8081/chat");

    host.Open();

    ... // remaining code ommitted for brevity
}
```

Or you can specify the binding name when defining endpoints in the host's configuration file. When you specify an endpoint via configuration, the binding name must be written in camel case (for instance basicHttpBinding, wsHttpBinding, or netTcpBinding). **Figure 3** illustrates how to configure the same three endpoints in the host's configuration file.

Figure 3 Configuring Endpoints in Host's Configuration File

Copy Code

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="ChatService">
        <endpoint
           address="http://localhost:8080/chat"
           binding="basicHttpBinding"
           contract="IChat"/>
        <endpoint
           address="http://localhost:8080/chat/secure"
           binding="wsHttpBinding"
           contract="IChat"/>
        <endpoint
           address="net.tcp://localhost:8081/chat"
           binding="netTcpBinding"
           contract="IChat"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

The remaining bindings in **Figure 2** target other common communication scenarios. For example, when you need bidirectional communication over HTTP, you can use WSDualHttpBinding, or when you want to implement asynchronous durable messaging, you can use NetMsmqBinding. When you need peer-to-peer capabilities for rich client applications, you can turn to NetPeerTcpBinding.

You'll be hard-pressed to come up with communication scenarios that aren't already addressed by one of these built-in bindings. However, it's very likely that you'll need to configure one of the built-in bindings in order to accommodate a specific integration issue.

Configuring Bindings

You can configure any of the built-in bindings by taking advantage of the various constructors and properties found on the classes. Once you've instantiated a binding object, you can modify its public writable properties before you pass the object to ServiceHost.AddServiceEndpoint.

The following example illustrates how to configure a BasicHttpBinding object to use MTOM and transport security (HTTPS):

Copy Code

```
BasicHttpBinding basicHttpBinding = new BasicHttpBinding();
basicHttpBinding.MessageEncoding = WSMessageEncoding.Mtom;
basicHttpBinding.Security.Mode = BasicHttpSecurityMode.Transport;

host.AddServiceEndpoint(
    typeof(IChatService),
    basicHttpBinding,
    "http://localhost:8080/chat");
...
```

You can accomplish the same thing entirely within configuration using the <bindings> configuration section. This mechanism allows you to define numerous named binding configurations for any of the built-in binding classes. Then you can apply a particular binding configuration to an endpoint through the <endpoint>'s bindingConfiguration attribute. **Figure 4** illustrates how to create a binding configuration named basicConfig that's also set up to use MTOM and transport security.

Figure 4 Creating a Binding Configuration

Copy Code

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="ChatService">
        <endpoint address="http://localhost:8080/chat"
          binding="basicHttpBinding"
          bindingConfiguration="basicConfig"
          contract="ChatLibrary.IChat" />
        ...
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding name="basicConfig" messageEncoding="Mtom">
          <security mode="Transport"/>
        </binding>
      </basicHttpBinding>
```

```
      ...
    </bindings>
  </system.serviceModel>
</configuration>
```

Notice how the <endpoint> definition refers to basicHttpBinding for the binding name and basicConfig for the binding configuration. You can define multiple binding configurations for each known binding class when multiple endpoints need to use different binding configurations. Take a look at **Figure 5** for an example. In this configuration file, I've created two binding configurations for BasicHttpBinding—both enable transport security but one uses text while the other uses MTOM. The first endpoint is configured to use basicConfig1, the second basicConfig2. Both endpoints use the same binding class but two different configurations.

Figure 5 Binding Configuration Examples

Copy Code

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="ChatService">
        <endpoint address="text"
           binding="basicHttpBinding"
           bindingConfiguration="basicConfig1"
           contract="IChat" />
        <endpoint address="mtom"
           binding="basicHttpBinding"
           bindingConfiguration="basicConfig2"
           contract="IChat" />
        <endpoint address="secure"
           binding="wsHttpBinding"
           bindingConfiguration="wsConfig"
           contract="IChat" />
        <endpoint address=""
           binding="netTcpBinding"
           bindingConfiguration="tcpConfig"
           contract="IChat" />
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding name="basicConfig1" messageEncoding="Text">
          <security mode="Transport"/>
        </binding>
        <binding name="basicConfig2" messageEncoding="Mtom">
          <security mode="Transport"/>
```

```
        </binding>
      </basicHttpBinding>
      <wsHttpBinding>
        <binding name="wsConfig" transactionFlow="true">
          <security mode="TransportWithMessageCredential">
            <message clientCredentialType="UserName"/>
          </security>
          <reliableSession enabled="true" ordered="true"/>
        </binding>
      </wsHttpBinding>
      <netTcpBinding>
        <binding name="tcpConfig" transactionFlow="true"
            transactionProtocol="WSAtomicTransactionOctober2004">
          <security mode="None"/>
          <reliableSession enabled="true" />
        </binding>
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Figure 5 also contains a configuration for WSHttpBinding, named wsConfig, which does several things. It enables transaction flow and ordered reliable messaging. It also changes the security mode to TransportWithMessageCredential and specifies a client credential type of UserName—this means the channel stack will be configured to use transport security (HTTPS) but incoming messages will be expected to contain WS-Security <UsernameToken> elements.

The final binding configuration shown in **Figure 5** changes the defaults for NetTcpBinding. In this example, the binding configuration named tcpConfig enables transaction flow and changes the transaction protocol from OleTransactions (the default) to the October 2004 version of WS-AtomicTransaction. It also disables transport security, which is on by default, and enables WS-ReliableMessaging.

The Windows Communication Foundation binding configuration mechanism can accommodate most of the integration scenarios. However, if luck still finds you stuck dealing with an obscure scenario that you cannot resolve, you can always define a custom binding to meet your exact needs.

Defining a Custom Binding

You define a custom binding by deriving a class from System.ServiceModel.Channels.Binding. Your implementation has one primary responsibility: to produce an ordered collection of BindingElement objects (an object of type BindingElementCollection). The runtime asks the binding instance to do this by calling its CreateBindingElements method at run time. At this point, the binding instance should look at its current configuration, create the various binding element objects it needs, and order them properly in the collection before returning it.

The Windows Communication Foundation runtime walks through the binding element collection and uses it to create the underlying channel stack (this is essentially what I mean by the "recipe"). There's a one-to-one mapping between the binding elements and the objects that end up in the channel stack at run time.
Each of the built-in binding classes derives from Binding and provides a unique implementation of CreateBindingElements. You should whip out .NET Reflector (available at aisto.com/roeder/dotnet) and view the implementation of each class. You'll learn exactly what binding elements each one uses and the order in which they are placed in the collection. This is valuable because you'll be doing the exact same thing when you create a custom binding.
Before you can create a custom binding, however, you must become familiar with the binding element classes that are available to you. A binding element is a class that derives from System.ServiceModel.Channels.BindingElement. Windows Communication Foundation ships with numerous BindingElement classes that are ready to use.
Figures 6 through 9 describe most of the built-in BindingElement classes that are currently found in Windows Communication Foundation. **Figure 6** lists the classes that represent the different transport protocols supported by Windows Communication Foundation. **Figure 7** describes the classes that represent different types of transport security. **Figure 8** describes the classes that represent the different message encodings. And finally, **Figure 9** describes the classes that represent the layered protocols for security, reliable messaging, and transactions.

Figure 9 Protocol Binding Elements

Protocol	Class	Element
Transaction Flow	TransactionFlowBindingElement	<transactionFlow/>
Reliable Messaging	ReliableSessionBindingElement	<reliableSession/>
Security	SecurityBindingElement	<security/>

Figure 8 Message Encoding Binding Elements

Message Encoding	Class	Element
Text	TextMessageEncodingBindingElement	<textMessageEncoding/>
MTOM	MtomMessageEncodingBindingElement	<mtomMessageEncoding/>
Binary	BinaryMessageEncodingBindingElement	<binaryMessageEncoding/>

Figure 7 Transport Security Binding Elements

Transport Security	Class	Element
Windows	WindowsStreamSecurityBindingElement	<windowsStreamSecurity/>
SSL	SslStreamSecurityBindingElement	<sslStreamSecurity/>

Figure 6 Transport Binding Elements

Transport	Class	Element
HTTP	HttpTransportBindingElement	<httpTransport/>
HTTPS	HttpsTransportBindingElement	<httpsTransport/>
TCP	TcpTransportBindingElement	<tcpTransport/>
Named pipes	NamedPipeTransportBindingElement	<namedPipeTransport/>
MSMQ	MsmqTransportBindingElement	<msmqTransport/>
MSMQ	MsmqIntegrationBindingElement	<msmqIntegration/>
P2P	PeerTransportBindingElement	<peerTransport/>

There is another BindingElement not listed in any of these figures. It's called CompositeDuplexBindingElement (<compositeDuplex/>) and it's something of a special case because it doesn't fit into any of these groupings. You use it when you need to create a binding that supports duplex communications.
You create a custom binding by instantiating CustomBinding and adding the desired BindingElement objects to its Elements collection. However, the order in which you add them is very important. Here are the order details:

Transaction Flow (Not Required)
Reliable Messaging (Not Required)
Message Security (Not Required)
Composite Duplex (Not Required)
Message Encoding (Required)
Transport Security (Not Required)
Transport (Required)

You add to the collection from top to bottom. Only the transport binding element is officially required when defining a custom binding. Message encoding is required for each binding, but if you don't specify one, Windows Communication Foundation will add a default encoding for you. The default encoding for HTTP(S) is text and for all other transports it is binary.
Here's an example of creating and using one of the simplest custom bindings possible:

Copy Code

```
...
CustomBinding myHttpBinding = new CustomBinding();
myHttpBinding.Name = "myHttpBinding";
myHttpBinding.Elements.Add(new HttpTransportBindingElement());

host.AddServiceEndpoint(typeof(IChat), myHttpBinding,
    "http://localhost:8080/chat/custom");
...
```

The only BindingElement object I added was an instance of HttpTransportBindingElement. In this case, Windows Communication Foundation will choose to use the text message encoding when it builds the channel stack since it's the default encoding for HTTP. This particular example produces a binding that ends up being identical to the built-in BasicHttpBinding.

However, you now have direct access to the underlying BindingElement objects, which gives you more flexibility and control over the resulting channel stack configuration. When you use one of the built-in binding classes, you're limited to the facade it provides over the underlying BindingElementCollection.

Consider the example shown in **Figure 10**. It configures the individual BindingElement objects before adding them to the collection. The resulting binding uses a different message version than normal, SOAP 1.1 along with the August 2004 version of WS-Addressing, which may come in handy when you need to integrate with older Web services frameworks. It also uses a customized HTTP transport instance.

Figure 10 Configuring the Individual BindingElement Objects

Copy Code

```
...
// instantiate message encoding element and configure
TextMessageEncodingBindingElement text =
    new TextMessageEncodingBindingElement();
text.MessageVersion = MessageVersion.Soap11WSAddressingAugust2004;

// instantiate transport element and configure
HttpTransportBindingElement http = new HttpTransportBindingElement();
http.TransferMode = TransferMode.Streamed;
http.UseDefaultWebProxy = true;

CustomBinding myHttpBinding = new CustomBinding();
myHttpBinding.Name = "myHttpBinding";
myHttpBinding.Elements.Add(text);
myHttpBinding.Elements.Add(http);

host.AddServiceEndpoint(typeof(IChat), myHttpBinding,
    "http://localhost:8080/chat/custom");
...
```

Of course, you can accomplish the same thing entirely within configuration by creating a configuration for <customBinding> as illustrated in **Figure 11**. This technique works like configuring any other built-in binding. You define a configuration for customBinding and then apply it to an endpoint. The difference is that in this case you're defining the entire binding definition within the configuration. You don't get anything for free when using CustomBinding.

Figure 11 Creating a Configuration for

Copy Code

```
<configuration>
  <system.serviceModel>
    <services>
      <service name=”ChatService”>
        <endpoint
          address=”custom”
          binding=”customBinding”
          bindingConfiguration=”myBasicHttpBindingConfiguration”
          contract=”IChat” />
      </service>
    </services>
    <bindings>
      <customBinding>
        <binding name=”myHttpBindingConfiguration”>
          <textMessageEncoding
            messageVersion=”Soap11WSAddressingAugust2004”/>
          <httpTransport
            useDefaultWebProxy=”true” transferMode=”Streamed”/>
        </binding>
      </customBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Figure 12 shows a few more custom binding examples that use some of the other BindingElement classes shown in the figures. The myWSHttpBindingConfiguration configuration is similar to the built-in WSHttpBinding except it uses the binary message encoding and it enables transaction flow and ordered reliable messaging. The myNetTcpBindingConfiguration configuration is like NetTcpBinding except it uses the text message encoding and enables transaction flow.

Figure 12 Custom Binding Configuration Examples

Copy Code

```
<configuration>
  <system.serviceModel>
    ...
    <bindings>
      <customBinding>
        <binding name=”myBasicHttpBindingConfiguration”>
          <textMessageEncoding
            messageVersion=”Soap11WSAddressingAugust2004”/>
          <httpTransport allowCookies=”true”
            useDefaultWebProxy=”true” transferMode=”Streamed”/>
        </binding>
```

```
        <binding name="myWSHttpBindingConfiguration">
          <transactionFlow/>
          <reliableSession ordered="true"/>
          <security authenticationMode="SspiNegotiated"/>
          <binaryMessageEncoding/>
          <httpTransport/>
        </binding>
        <binding name="myNetTcpBindingConfiguration">
          <transactionFlow/>
          <textMessageEncoding/>
          <windowsStreamSecurity/>
          <tcpTransport/>
        </binding>
      </customBinding>
    </bindings>
</configuration>
```

Making Custom Bindings Easy to Use

Although CustomBinding is convenient for one-off customizations, you should define a custom Binding class when you need to make it easier to reuse. When you derive from Binding, you need to override its two abstract members: CreateBindingElements and Scheme. Your implementation of CreateBindingElements is where you create and return the collection of BindingElement objects.

Suppose you want to create a custom HTTP binding that always uses the binary message encoding—call it NetHttpBinding—and assume that you want to let the users of this binding configure two properties on the underlying HttpTransportBindingElement instance—TransferMode and UseDefaultWebProxy. The code in **Figure 13** demonstrates how you can accomplish this.

Figure 13 Custom HTTP Binding

Copy Code

```
public class NetHttpBinding : Binding
{
    private BinaryMessageEncodingBindingElement binary =
        new BinaryMessageEncodingBindingElement();
    private HttpTransportBindingElement http =
        new HttpTransportBindingElement();

    public override BindingElementCollection CreateBindingElements()
    {
        return new BindingElementCollection(
            new BindingElement[] { binary, http });
    }
```

```
    public TransferMode TransferMode
    {
        get { return http.TransferMode; }
        set { http.TransferMode = value; }
    }

    public bool UseDefaultWebProxy
    {
        get { return http.UseDefaultWebProxy; }
        set { http.UseDefaultWebProxy = value; }
    }

    public override string Scheme { get { return "http"; } }
}
```

With the code in place, developers can simply instantiate your NetHttpBinding class and apply the instance to an endpoint as illustrated here:

Copy Code

```
...
NetHttpBinding netHttp = new NetHttpBinding();
netHttp.TransferMode = TransferMode.Streamed;
netHttp.UseDefaultWebProxy = true;

host.AddServiceEndpoint(typeof(IChat), netHttp,
    "http://localhost:8080/chat/nethttp");
...
```

This is much easier than dealing with CustomBinding. But if you want to make it even easier, you may want to make it possible to configure the NetHttpBinding via the Windows Communication Foundation configuration section. In order to do this, you need to implement a few more configuration-related classes. **Figure 14** provides a complete example of the classes you'll need.

Figure 14 Custom Configuration Classes for NetHttpBinding

Copy Code

```
public class NetHttpBindingConfigurationElement : StandardBindingElement
{
    public NetHttpBindingConfigurationElement(string configurationName)
        : base(configurationName) { }

    public NetHttpBindingConfigurationElement()
        : this(null) { }

    protected override Type BindingElementType
```

```
{
    get { return typeof(NetHttpBinding); }
}

[ConfigurationProperty("transferMode",
     DefaultValue = TransferMode.Buffered)]
public TransferMode TransferMode
{
    get { return ((TransferMode)(base["transferMode"])); }
    set { base["transferMode"] = value; }
}

[ConfigurationProperty("useDefaultWebProxy", DefaultValue = false)]
public bool UseDefaultWebProxy
{
    get { return ((bool)(base["useDefaultWebProxy"])); }
    set { base["useDefaultWebProxy"] = value; }
}

protected override ConfigurationPropertyCollection Properties
{
    get
    {
        ConfigurationPropertyCollection properties = base.Properties;
        properties.Add(new ConfigurationProperty("transferMode",
            typeof(TransferMode), TransferMode.Buffered));
        properties.Add(new ConfigurationProperty("useDefaultWebProxy",
            typeof(bool), true));
        return properties;
    }
}

protected override void InitializeFrom(Binding binding)
{
    base.InitializeFrom(binding);
    NetHttpBinding netHttpBinding = ((NetHttpBinding)(binding));
    this.TransferMode = netHttpBinding.TransferMode;
    this.UseDefaultWebProxy = netHttpBinding.UseDefaultWebProxy;
}

protected override void OnApplyConfiguration(Binding binding)
{
    if (binding == null)
    {
```

```
            throw new System.ArgumentNullException("binding");
        }
        if (binding.GetType() != typeof(NetHttpBinding))
        {
            throw new System.ArgumentException(
                "Invalid binding type - expected NetHttpBinding");
        }
        NetHttpBinding netHttpBinding = ((NetHttpBinding)(binding));
        netHttpBinding.TransferMode = this.TransferMode;
        netHttpBinding.UseDefaultWebProxy = this.UseDefaultWebProxy;
    }
}

public class NetHttpBindingSection :
    StandardBindingCollectionElement<NetHttpBinding,
        NetHttpBindingConfigurationElement>
{
}
```

With these classes in place, you can add a binding extension mapping to the configuration file so Windows Communication Foundation knows that netHttpBinding goes with the configuration classes shown in **Figure 14**. Now you can simply specify netHttpBinding for the binding name, and configure it within the <bindings> section like any other binding, as shown in **Figure 15**. (The MSDN® documentation available at msdn2.microsoft.com/aa967561.aspx provides more complete custom binding examples.)

Figure 15 Configuring netHttpBinding

Copy Code

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="ChatService">
        <endpoint
          address="nethttp"
          binding="netHttpBinding"
          bindingConfiguration="myNetHttpBindingConfiguration"
          contract="ChatLibrary.IChat" />
      </service>
    </services>
    <bindings>
      <netHttpBinding>
        <binding
          name="myNetHttpBindingConfiguration"
          transferMode="Streamed"
```

```
          useDefaultWebProxy="true"/>
      </netHttpBinding>
    </bindings>
    <extensions>
      <bindingExtensions>
        <add name="netHttpBinding"
          type="NetHttpBindingSection, NetHttpBindingLibrary" />
      </bindingExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```

I've covered all the major options and components related to implementing a custom binding. The only thing I haven't yet discussed is how to write a custom BindingElement class, which may be necessary if you can't find one that fits your exact needs. For example, say you need to implement a custom transport. In such a case, you'll need to implement the transport channel (which is non-trivial) and a corresponding BindingElement class in order to make it easy to use within a binding. Implementing custom channel components and binding elements are advanced extensibility topics that I'll save for a future column.

Sharing Binding Descriptions

Bindings are all about what happens on the wire. When you choose a particular binding, configure one in a special way, or define a custom binding, it's vital for consumers to be able to discover the exact binding configuration so they can properly integrate with the endpoint. Services can share their binding configurations with consumers by exposing metadata in the form of Web Services Description Language (WSDL) and WS-Policy.

You can enable metadata on a Windows Communication Foundation service through the <serviceMetadata> behavior. This allows developers to browse to the service description using a Web browser. You can also make the metadata available programmatically by enabling MEX (IMetadataExchange) on a separate endpoint.

Once you have metadata enabled, Windows Communication Foundation automatically produces the appropriate WSDL definition for the service upon request. The binding configuration for each endpoint is translated into WS-Policy statements and embedded within the WSDL definition to specify the exact configuration. If you browse to a Windows Communication Foundation-generated WSDL definition, you'll find these policy statements toward the top of the file.

As long as the client is capable of doing everything found in the WS-Policy statement, it should be able to produce the necessary client-side configuration. When using Windows Communication Foundation on the client side, svcutil.exe inspects the policy statements found in the WSDL definition and translates them into a corresponding binding configuration section in the client configuration file (see **Figure 16**). The net result is that the client ends up with an equivalent channel stack for dispatching outgoing messages.

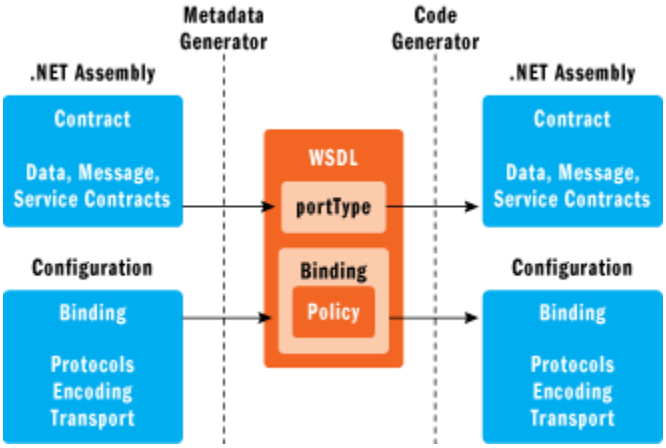

Figure 16 **Sharing Binding Congfigurations via Metadata** (Click the image for a larger view)

The following code illustrates how a Windows Communication Foundation client can automatically access all of the endpoints exposed by a service via MEX at run time:

Copy Code

```
...
ServiceEndpointCollection endpoints =
    MetadataResolver.Resolve(typeof(IChat),
    new EndpointAddress("http://localhost:8080/chat/mex"));
foreach (ServiceEndpoint se in endpoints)
{
    ChannelFactory<IChat> cf = new ChannelFactory<IChat>(
        se.Binding, se.Address);
    IChat client = cf.CreateChannel();
    client.SendMessage(msg);
}
...
```

With this code available on the client, you can add new endpoints to the service, each with a unique binding configuration. Each time you run the client it will invoke all of the endpoints using the correct configuration. This shows that the client is able to automatically discover each endpoint's binding configuration.

One thing that can potentially break interoperability, even when using Windows Communication Foundation on both sides, is when the service employs a custom BindingElement that the client doesn't have access to (such as a custom transport). Obviously, using any type of custom channel component requires agreement between the client and service, and both sides will require an appropriate implementation.

This brings me to my final word of caution. It's easy to configure yourself out of interoperability by getting too fancy with a binding configuration for no good reason. Make sure you know what you're enabling and why. If you don't have a good reason for something, stay with the defaults.

That said, when you do run into situations where interoperability fails when using the defaults, your first goal should be to figure out what's getting in the way and then you can configure the binding (or create a custom binding) that works around the problem at hand.

Conclusion

The Windows Communication Foundation binding architecture provides an elegant model for configuring what happens on the wire. You simply specify a binding and Windows Communication Foundation uses that as the recipe for building the channel stack at run time.

Windows Communication Foundation ships with numerous built-in bindings that come preconfigured to address today's most common communication needs. However, when the built-in bindings don't fit the bill, you can configure them or create custom bindings of your own, making it possible to accommodate any tough integration scenario you encounter.