

Introduction to Building Windows Communication Foundation Services

Clemens Vasters
newtelligence AG

September 2005

Summary: Clemens Vasters explains the fundamental concepts of the new connected systems platform for Windows, Windows Communication Foundation (previously code-named "Indigo"), and shows you how to build services and service clients using Windows Communication Foundation's **System.ServiceModel** namespace.

Note This paper and its code samples are written against "Indigo Beta1" and may change during later releases of Windows Communication Foundation.

Contents

[What is Windows Communication Foundation?](#)
[The ABC of Windows Communication Foundation](#)
[Defining Service Contracts](#)
[Defining Data Contracts](#)
[RPC or Messaging?](#)
[Implementing Services](#)
[Hosting Services](#)
[Choosing and Configuring Bindings](#)
[Binding Requirements and Validation](#)
[Clients](#)
[Conclusion](#)

There are just a few rare times when authors of technical articles can introduce an entirely new technology instead of writing about yet another angle of or feature addition to an already existing technology. This is one of these rare times—and instead of jumping straight to the programming classic "Hello World," we'll start exploring the world of Windows Communication Foundation (WCF) in a way more akin to Sesame Street's "ABC". But before we go there, we will look at what Windows Communication Foundation is and why Microsoft is building it.

What is Windows Communication Foundation?

So what is Windows Communication Foundation (WCF)? WCF is Microsoft's next-generation programming platform and runtime system for building, configuring and deploying network-distributed services. The WCF runtime and its `System.ServiceModel` namespace that represents its primary programming interface, is nothing less than the successor and unification of most distributed systems technologies that developers have successfully used to build distributed applications on the Windows platform over the past decade.

Let's start with the following important observation: The current distributed systems technologies, most prominently ASP.NET Web Services (ASMX) along with the Web Service Enhancements (WSE) extensions, the Microsoft Message Queue (MSMQ), the Enterprise Services/COM+ runtime environment, and .NET Remoting are the foundation for countless successful applications. So is there anything fundamentally broken or wrong about these technologies that would prompt Microsoft to replace them all? No, except that there are too many of them.

Using ASMX and WSE, you can build powerful and interoperable service-oriented applications with Web services and Web service clients that allow for cross-platform interoperability and integration, and relatively easy service evolution and versioning strategies through loose coupling. With MSMQ you have a powerful mechanism for scalable, durable and volatile, queued messaging that takes care of getting data reliably from one place to the next. Enterprise Services provides transaction integration across multiple parties performing related work in a distributed environment, allows throttling access to limited, shared resources, allows pooling of object instances to optimize access to expensive-to-initialize resources, features a publish/subscribe mechanism for events, has a rich security model, and sits on top of a proven, platform-integrated, secure, and fast transport. Last but not least, Remoting is the Common Language Runtime (CLR) integrated mechanism for communicating with objects across application domain boundaries and is loved for its very flexible extensibility model, which lets developers rip and replace transports, proxy mechanisms and the way communication channels behave.

All of this is good except that you often have to make an exclusive, explicit choice for one these fundamental technologies. If you need any of the features of Enterprise Services and decide to use it, your implementation strategy will be very different from when you use ASMX. If you need rock-solid reliable messaging decide to send messages via MSMQ instead of risking to send them via plain HTTP, your code will look vastly different from invoking methods on a method-oriented proxy, unless you add some plumbing code yourself or use Enterprise Service's Queued Components feature (which brings us back to the previous point).

The goals of WCF are that you no longer have to make such technology choices upfront and that you can implement any combination of such requirements on a single technology platform and without friction. If you want to build a Web service with reliable communication that supports sessions and transaction flow and extend it to let you inspect the raw messages as they flow into the system, it will be very easy to do so in WCF. Doing this today (and this is not only true for the Microsoft platform) is not entirely impossible, but it would require developers with a lot of system-level background knowledge about all of the aforementioned technologies and quite a bit of time.

The promise is that WCF will make developers more productive, because they will only have to master a single programming model that unifies the feature wealth of ASMX, WSE, Enterprise Services, MSMQ, and Remoting. And it should not be forgotten that WCF is Microsoft's implementation of all these WS-* standards that Microsoft has developed in cooperation with industry partners over the past 5 years—which is promising to ensure broad interoperability across platforms, runtimes and programming languages.

So, after all these bold promises, let's take a look at it:

The ABC of Windows Communication Foundation

"ABC" is the WCF mantra. "ABC" is the key to understanding how a WCF service endpoint is composed. Think Ernie, Bert, Cookie Monster or Big Bird. Remember "ABC".

- "A" stands for Address: Where is the service?
- "B" stands for Binding: How do I talk to the service?
- "C" stands for Contract: What can the service do for me?

Web services zealots who read Web Service Description Language (WSDL) descriptions at the breakfast table will easily recognize these three concepts as the three levels of abstraction expressed in WSDL. So if you live in a world full of angle brackets, you can look at it this way:

- "A" stands for Address—as expressed in the `wsdl:service` section and links `wsdl:binding` to a concrete service endpoint address.
- "B" stands for Binding—as expressed in the `wsdl:binding` section and binds a `wsdl:portType` contract description to a concrete transport, an envelope format and associated policies.
- "C" stands for Contract—as expressed in the `wsdl:portType`, `wsdl:message` and `wsdl:type` sections and describes types, messages, message exchange patterns and operations.

"ABC" means that writing (and configuring) a WCF service is always a three-step process:

- You define a contract and implement it on a service
- You choose or define a service binding that selects a transport along with quality of service, security and other options
- You deploy an endpoint for the contract by binding it (using the binding definition, hence the name) to a network address.

It is important to note is that these three elements are independent. A contract can support many bindings and a binding can support many contracts. A service can have many endpoints (contract bound to address) coexisting and available at the same time. So if you want to expose your service via HTTP and use SOAP 1.1 for maximum interoperability, and also want to expose it via TCP using a binary wire encoding for maximum performance, the two resulting endpoints can reside side-by-side on top of the very same service.

Of course, not all bindings necessarily satisfy the needs of a given service contract or service implementation. If a service does, for instance, absolutely depend on certain security aspects,

reliable messaging, transaction flow, or other features, it can demand that any binding that is used for exposing endpoints must support them. If a binding does not support the demanded features, the WCF runtime will detect the feature mismatch and will refuse to start the service.

Defining Service Contracts

WCF contracts are defined in one of two ways. You can design or otherwise acquire a WSDL contract description and turn it into an WCF contract description using the svcutil.exe tool is included with the WCF runtime binaries. "WCF contract description" means that the tool will generate code that's marked up with metadata attributes telling the WCF runtime about the specifics of the contract definition. Contrary to ASMX, where the wsdl.exe tool imports WSDL descriptions into either a concrete proxy class implementation or a concrete (yet abstract) class server implementation, the WCF tool will generate a contract description as an interface declaration that can be used both for the server-side and for the proxy. So if you wanted to import a contract description for Microsoft's TerraService, the command-line would look likely like this:

```
Svcutil.exe http://terraserver.microsoft.com/TerraService2.asmx  
/useXmlSerializer
```

The resulting output is a source-code file (you can specify the programming language using the /language switch) that contains all necessary data declarations, service interfaces and, for convenience, readily implemented proxy classes.

The more common and natural way to define WCF service contracts is to write the contract declaration not in WSDL, but rather in your preferred programming language. After all, it should be the job of the platform to deal with all the intricate details of WSDL and all the other WS-* specifications and to make sure that services are indeed interoperable. Whether you write your contract declaration in WSDL or by declaring a properly marked up Visual Basic or C# interface should not and indeed does not matter. Making sure that the resulting metadata is equivalent independent of whether you define in C# and export to WSDL, or whether you define in WSDL and import to Visual Basic, and to warn you if an equivalent export or import cannot be performed is the responsibility of WCF. From the WCF perspective, WSDL or a CLR interface declaration are simply different and equally valid ways to express the very same metadata. And as it happens, Visual Basic or C# are a lot friendlier to a programmer's eye than looking at a WSDL declaration—at least for most of us. Therefore, we'll simply leave out all that angle brackets business and look at some code:

```
public interface ICalculate  
{  
    double Add( double a, double b);  
    double Subtract( double a, double b);  
}
```

Above is a very plain and simple CLR interface declaration expressed in C#. If you have used Remoting or Enterprise Services, you will know that any such interface and of all its methods are inherently "remotable" using these technologies. Not so in WCF.

WCF follows the service-orientation tenet of "explicit boundaries". That means that nothing is exposed through WCF unless you explicitly tell the runtime to do so. WCF strictly distinguishes between elements that are private to an application and elements that are made visible and accessible to the outside world. The "public", "private" and "protected" keywords of C# and Visual Basic and the underlying access protection (not security!) mechanisms are meaningful to the code inside an application, but whether an interface or method that's public within the scope of an application shall indeed be publicly visible to the outside world is an entirely different matter altogether. Reversely, there are cases where it is desirable to enforce that a method that's designed exclusively for accessing the application from the outside is not accidentally invoked from within the application. Under such circumstances, a method might be best declared as "private" towards the inside of the application, but is indeed made accessible to outside callers.

```
using System.ServiceModel;

[ServiceContract]
public interface ICalculate
{
    [OperationContract]
    double Add( double a, double b);
    [OperationContract]
    double Subtract( double a, double b);
}
```

Here you can see that the interface has been labeled with the **[ServiceContract]** attribute. That attribute explicitly marks a CLR interface as to carry contract metadata for WCF. Each method is explicitly labeled with the **[OperationContract]** attribute, which is the WCF equivalent to a "public" method. Based on the existence and the additional, optional parameters than can be supplied to these metadata attributes, the WCF runtime can, whenever required, turn the **[ServiceContract]** into an equivalent WSDL **portType** declaration. Each **[OperationContract]** declaration is then mapped to an equivalent WSDL **operation** definition whereby the parameter lists and return values are turned into XML schema declarations reference by the WSDL **types** section and the respective WSDL **message** mappings. We will return once more to these attributes a bit later in this article when we discuss some more specific aspects of contracts.

```
[ServiceContract]
public class Calculator
{
    [OperationContract]
    public double Add(double a, double b) { ... };
    [OperationContract]
    private double Subtract(double a, double b) { ... };
}
```

Above is a very special variation of a contract. The class you see here is a service implementation that does not explicitly implement an interface, but rather declares the contract "inline". This is a special (and not necessarily encouraged) declaration style that allows to port existing code to WCF without having to refactor the code and extracting interfaces from such existing classes. Instead, all methods that shall be exposed on the service edge can be marked up with the appropriate attributes. Candidates for using this declaration style are ports of existing ASMX Web services, Remoting classes, or Enterprise Services service components that—even though best practice guidelines recommend differently—in reality often don't implement interfaces. Note that one of the methods shown is declared as private in C#, but still carries the **[OperationContract]** attribute. Therefore, the method can only be invoked from the outside (or from within the service class itself), but not from inside the service application. Combining explicit interfaces and implicit interfaces is not possible on a service implementation. You have to decide on one of the two implementation styles.

Defining Data Contracts

The operations of the contracts shown in the previous code examples accept only simple types as parameters and the return values are also simple types. If you are dealing with parameter lists of that sort, it is likely that you will—for the most part—ignore the fact that the data you are exchanging between services is serialized from its in-memory object representation into an XML information set and subsequently encoded for the wire transfer (and vice versa, of course).

If you are dealing with predefined WSDL documents or if you want to use complex types as arguments, you need to give a little more consideration to the "data contract", however. While a **service contract** defines the shape and rules for interfaces (portTypes), along with their associated messages and operations, the **data contract** defines the shape and rules for the data that is exchanged through operation's input and output messages.

The split between the **data contract** and the **service contract** is important. Service contracts typically define a logically and semantically related set of operations grouped on an interface and are about how a service behaves. The data contract defined information items that you flow across service boundaries and that are handled with additional logic on the provider and consumer side. If you were looking at it from a (English language) grammar perspective, you could consider operations to be the **predicates** and the data that flows are the **objects**, while the caller is the **subject**. Subjects can do a lot with objects. Every "do" is an operation.

```
[DataContract]
public class Person
{
    [DataMember]
    public int Id;
    [DataMember]
    public string FirstName;
    [DataMember]
    public string LastName;
}
```

Above you see a data contract for a person. The attributes stem from the `System.Runtime.Serialization` namespace that is the home of the new **XmlFormatter** infrastructure. The **XmlFormatter** is similar to the well-known **XmlSerializer** (as well as the **Binary Formatter** and **SoapFormatter**) in purpose, but does things quite a bit differently. A complete discussion of the **XmlFormatter** is beyond the scope of this article, so we will focus on the key differences between this new serialization infrastructure and the ones already present in the .NET Framework.

The most striking and significant difference between the **XmlFormatter** and the **XmlSerializer** is that the new infrastructure intentionally gives the developer **less** control over the XML schema and the way how data is serialized to and deserialized from XML. That seems odd at first consideration, because we are typically used to get increased control and flexibility with each new piece of software, but this particular difference does in fact reflect a tradeoff decision that makes sense—you trade fine-grained control for automatic versioning support of messages and therefore for an easier way of implementing loosely coupled, evolvable data structures that will upgrade and downgrade well between different data contract (and therefore schema) versions. To take advantage of this new functionality, you will have to surrender some control over the schema details to the formatter infrastructure. The **XmlFormatter** will, for instance, not allow you to declare parts of a data contract as XML attributes and other parts as XML elements. All data is always serialized into XML elements.

```
[DataContract]
public class Person
{
    [DataMember]
    public int Id;
    [DataMember]
    public string FirstName;
    [DataMember]
    public string LastName;
    [DataMember(IsOptional = true, VersionAdded = 2)]
    public string MiddleName;
}
```

The code snippet above shows a slight variation of the previous example where a new member was added. By labeling the new member with the **IsOptional** property on the **[DataMember]** attribute set to **true** (the default is **false**) we tell the **XmlFormatter** infrastructure to treat the newly introduced member as an optional member. That means that if we receive data serialized by an older version of the contract (which might be embedded in a client proxy), the member will remain at its default value upon deserialization and the newer version will gracefully accept an older version's input where this element is missing. If **IsOptional** is set to **false** and the respective element is missing in the deserialization input, an exception is raised.

```
[DataContract]
public class Person
{
    [DataMember]
    public int Id;
    [DataMember]
    public string FirstName;
    [DataMember]
    public string LastName;
    [DataMember(IsOptional = true, VersionAdded = 2)]
    public string MiddleName;
    [DataMember(MustUnderstand = true, VersionAdded = 3)]
    public DateTime DateOfBirth;
}
```

When an older version deserializes input from a newer version of a data contract that has additional members, the additional members are ignored, unless the newer contract sets the **MustUnderstand** property for the **[DataMember]** attribute to **true**. Whenever the XmlFormatter deserialization sees an element that is marked with the resulting **mustUnderstand** XML attribute and cannot associate the data with a data member in the destination data contract, an exception will be raised, because **MustUnderstand** means that the data must not be ignored.

The **VersionAdded** property tells the infrastructure in which version the new member was added, which helps sorting the members by version in the XML schema representation and the serialization output whereby earlier members go first. This allows the serialization output of version 2 to properly validate against the schema of version 1 and vice versa.

RPC or Messaging?

WCF is built around service-orientation principles. One of the service-orientation ideas is that services exchange messages rather than procedure calls. The service contract definitions shown earlier in this article, however, seem to contradict this idea, as they do not really look different from the RPC-style definitions that we know from COM, Remoting or Enterprise Services.

This similarity on the surface is not a bad thing. If you want to write services that look and behave similar to "traditional" components, you can do so and don't worry too much about how WCF turns method calls into messages under the surface. WCF services also support sessions and allow very detailed control over the lifetime of service instances. Services built on WCF are therefore not limited to a particular architectural style of "stateless" services as it is the case with ASMX, but you can build services where instances are created and released for each call or a defined sequence of calls, where instances exist for the duration of a session, or you can even build singletons where one service instance is shared by all callers.

What you cannot do, and this is the great difference to RPC technologies like DCOM or Remoting, is to pass objects by reference or pass callback delegates. The reason for this restriction is that a service-oriented technology cannot make as many assumptions about the network as the local area network technologies DCOM or Remoting. Callbacks and object references require a callback path from the server to the client whereby the server can contact the client whenever needed. In an environment where we have to assume that services are distributed across platforms, trust boundaries and wide area networks, this sort of bi-

directional connectivity very often does not exist. Clients may reside behind firewalls, or network address translation (NAT) services, or simply don't actively listen for messages. And even if they do, security restrictions typically mandate that any sender, including those returning calls, will have to authenticate and be authorized at any endpoint. Therefore, implicit backchannels such as those established by callbacks and object references simply don't work in a services world. Instead, such backchannels must be explicitly established using so-called "duplex" conversations, which is a topic that deserves a dedicated article altogether.

If you prefer "messaging style" programming over RPC-style programming, you have multiple choices; you can use so-called typed messages, you can send "raw" WCF **Message** instances, or you can compose the message contents in an operation's parameter list.

```
public enum UpdateConcurrency
{
    UpdateIfModified,
    FailIfModified
}

[DataContract]
public class UpdateBehavior
{
    [DataMember]
    public UpdateConcurrencyMode UpdateConcurrencyMode;
}

[MessageContract]
public class StorePersonMessage
{
    [MessageHeader]
    public UpdateBehavior UpdateBehavior;
    [MessageBody]
    public Person Person;
}

[ServiceContract]
public interface IPeople
{
    [OperationContract]
    public void StorePerson(StorePersonMessage storePersonMessage);
}
```

A "typed message" is an explicitly composed message structure that is labeled with a **[MessageContract]** attribute as shown above and has zero or more **[MessageHeader]** members and zero or more **[MessageBody]** members. As the attribute names suggest, **[MessageHeader]** members are mapped to headers in the SOAP envelope and **[MessageBody]** members are represented as elements inside the SOAP envelope's body section.

```
[ServiceContract]
public interface IPeople
{
    [OperationContract]
    public void StorePerson(
        [MessageHeader] UpdateBehavior UpdateBehavior,
        [MessageBody] Person Person);
}
```

If you want to flow some information in the SOAP envelope's header section, but you don't want to define an explicit message class, you can define the message contract "inline" in the operation's parameter list. Any parameter that is, as shown in the above snippet, labeled with the **[MessageHeader]** attribute will be mapped to the envelope's header section and all other arguments end up in the envelope body. If you want the entire body content or a part of it to be free-form XML, you can use **XmlNode** or **XmlElement**-typed parameter. This is valid for inline message contracts as well as for explicit message contract members.

```
[ServiceContract]
public interface IPeople
{
    [OperationContract]
    public void StorePerson(Message msg);
}
```

If you need complete control over the Message content, you can alternatively use a single `System.ServiceModel.Message` argument with your operations, which gives you immediate access to the WCF **Message**. This class exposes all headers in a collection whereby the content of a range of special headers such as those relevant for security and addressing are exposed through explicit properties. The body content is represented as an **XmlReader**.

```
[ServiceContract]
public interface IMyMessageHandler
{
    [OperationContract(Action="*")]
    public void HandleAnyMessage(Message message);
}
```

With the above examples, WCF is still responsible for dispatching the messages to the correct operation. Messages that do not have an **Action** header that matches one of the contract's **[OperationContract]** Action property values (which defaults to the method name of the operation) are rejected. If you want to accept all messages irrespective of their Action header, you can add an operation where the Action property is set to the wildcard indicator, which is a single asterisk ("*"). If such an operation is present, all messages that cannot be dispatched otherwise are dispatched to that operation and of course, this could also be the only operation on a service contract, which essentially results in a free-form SOAP messaging endpoint.

Recapitulating, it's quite amazing how you can tune your contracts to be anything from purely RPC-style interfaces to what is essentially raw SOAP messaging by combining metadata markup and parameter types, but never have to shift away from the same fundamental programming model.

Implementing Services

To implement a service you choose one or more predefined service contract interfaces and implement them on an ordinary CLR class using the programming language of your choice. Unlike Enterprise Services or Remoting, the service class does not need to be derived from any special base class. As mentioned earlier, special cases are classes with a contract that is defined "inline" and immediately on the class methods, but these do not need a special base class, either.

```
[ServiceContract]
public interface IPeople
{
    [OperationContract( TransactionFlowAllowed = true )]
    public void StorePerson(UpdateBehavior updateBehavior, Person person);
}

public class People : IPeople
{
    public void StorePerson(UpdateBehavior updateBehavior, Person person)
    { ... }
}
```

In most cases, especially when you do not opt to work with messages, the actual service implementation is not very special: You just implement whatever logic you want to implement.

```
[ServiceBehavior(ValidateMustUnderstand = true)]
public class People : IPeople
{
    [OperationBehavior(
        AutoEnlistTransaction = true,
        AutoCompleteTransaction = true,
        ReleaseInstance=ReleaseInstanceMode.BeforeAndAfterCall)]
    public void StorePerson(UpdateBehavior updateBehavior, Person person)
    { ... }
}
```

What is special, however, is that you can ask the WCF runtime to add certain behaviors to your service at runtime, just as with Enterprise Services. If you need, for instance, automatic transaction scopes for your operations or want WCF to make sure that a service instance is never called from more than one concurrent thread (for whatever reason), you can apply **[ServiceBehavior]** and **[OperationBehavior]** attributes to your implementation and WCF will implement the requested behavior "around" your service instance automatically just as Enterprise Services (or COM+) does it for its **ServiceComponent** (or configured components).

The behaviors that are readily available in WCF (unlike Enterprise Services, developers can add their own behaviors, but the details of that are far beyond the scope of this article) resemble those available in Enterprise Services/COM+, but provide far greater flexibility and a higher degree of control. Automatic transactions are a great example for this.

While Enterprise Services defines automatic transaction support on the class-level, WCF lets you define that behavior on a per-operation basis. This way, you can have both, transactional and non-transactional operations on the same service implementation. What is also different is that WCF disassociates "transaction flow" from one operation to the next from whether a transaction scope will be automatically created or whether an existing transaction scope is used for an operation. The capability to accept a foreign transaction from a caller is a property on the **[OperationContract]: TransactionFlowAllowed**. If the caller has a local transaction and is wants to flow the transaction to the remote service, only contracts that explicitly allow transaction flow for the invoked operation will do so.

Whether the service wants to automatically enlist into such a remotely provided transaction is a consideration that is separate from the transaction flow. Even if the contract allows transaction flow, a particular implementation of that contract may indeed opt not to use automatic transactions. Since the concrete is implementation specific, the control property for whether an operation automatically enlists into a transaction sits on the **[OperationBehavior]** attribute: **AutoEnlistTransaction**.

Alongside this property exists the **AutoCompleteTransaction** property, which controls whether the service will automatically mark the transaction as successful (commit) when the operation returns without an exception or as failed (abort) if an exception occurs. And even if this looks very similar to the behavior of the **[AutoComplete]** attribute of Enterprise Services, the behavior is not entirely identical. In Enterprise Services, an "auto complete" method will not only control the transaction vote, but it will also cause the component instance to be discarded. In WCF, the instance behavior is defined separately. Commonly, and in the interest of proper transactional isolation and to not keep intermediate, volatile transaction state in memory after a transaction completes or aborts, the **ReleaseInstance** behavior for a transactional method will be set to **ReleaseInstanceMode.BeforeAndAfterCall**. But while Enterprise Services forces services into that behavior, developers writing WCF services may very well choose to keep instances alive across transaction boundaries—WCF won't try to outsmart developers who know what they're doing.

As mentioned earlier, WCF also supports sessions. Whether session support is required is declared with the **[ServiceContract(Session=true)]** attribute setting and WCF will make sure (we will get to the "how" in a bit) that a session is established for the communication between endpoints using that particular contract. The existence of such a communication session does, however, not immediately control how instances of the service are handled. The correlation between a service instance and a session is explicitly controlled through the **[ServiceBehavior]** property **InstanceMode**. If you want the instance to stay alive for as long as the session lasts, you use **InstanceMode.PrivateSession** and if you want to have a new instance for every call irrespective of the session scope, you use **InstanceMode.PerCall**. In the latter case, you can, if you wanted to, store and recover whatever state you want to associate with the session using the session identifier that's readily available as a value on **OperationContext.Current.SessionIdentifier**.

Hosting Services

Looking at all these instance management options raises the question: Who does that? Up to this point we've dealt with interfaces and the service implementation class, but we're clearly missing something that will activate, manage and discard service instances; we need a service host.

From a developer perspective, WCF provides two alternatives for hosting services, which are both mostly identical under the covers. The easier of the two alternatives is to host services inside an ASP.NET application, the more flexible and more explicit alternative is to host services yourself and in whichever application process you choose.

```
<%@ Service Language="C#"
    CodeBehind="~/App_Code/Service.cs"
    Class="MyService" %>
```

Hosting WCF services in ASP.NET is very simple and straightforward and very similar to the ASMX model. You can either place your entire service implementation in a *.svc file just as with ASP.NET Web services *.asmx files, or you can reference a service implementation residing in a code-behind file or some other assembly. With respect to how the service implementation class is located (and possibly compiled), none of these options differ much from how you would typically create an ASMX Web service—even the attributes of the @Service directive are the same as those for the @WebService directive.

The important difference between WCF and ASMX is that the WCF service will not do anything until you specify precisely how it shall be exposed to the outside world. An ASMX service will happily start talking to the world once you place the *.asmx file into an IIS virtual directory. A WCF service will not talk to anybody until you tell it to do so—and how to do so.

Remember "ABC"? If you place the myservice.svc file at your Web server's root directory, you have an address (<http://www.example.com/myservice.svc>) and therefore the "A", you have a contract (the "C") along with its implementation, but the "B" is missing. We don't have a binding that binds the service to that address—the service doesn't know how to behave on the wire (except that it might, but only might, have to speak HTTP).

```
using (ServiceHost<People> peopleHost = new ServiceHost<People>() )
{
    peopleHost.Open();
    Console.WriteLine("Press ENTER to close");
    Console.ReadLine();
    peopleHost.Close();
}
```

Whenever a file with the *.svc extension is compiled and the respective service class is loaded, the WCF runtime will automatically create a "service host" for the service class. If you choose to host WCF services inside your own process, you have to provide such a service host yourself. Fortunately, this is typically not more complicated than instantiating a **ServiceHost<T>** whereby T is the name of your service implementation class as shown above. A service host starts managing service instances when it is opened and it stops when it is closed.

The **ServiceHost<T>** class contains all necessary functionality to manage instances and sessions and it is also the entry point for all messages directed at a particular service; the service host is the home to one or more endpoints that bind (here's our "B") the service to a transport addresses.

Choosing and Configuring Bindings

As said earlier: A binding binds a service contract implementation to an address. This includes, amongst many other aspects, choosing an appropriate transport, choosing a method how messages are encoded on the wire (are messages sent as angle-bracket XML plaintext or MTOM encoded or binary or in some other way?) and defining how security works, and whether and which reliable message delivery protocol is implemented and whether and how session support functions.

WCF comes with several predefined binding variants that select the most common combinations of transports, encodings and behaviors like the ones mentioned above and in the vast majority of cases you can simply pick one of these predefined bindings and—if at all necessary—tailor it to your needs.

The simplest binding is the **BasicProfileBinding**, which implements a combination of binding elements that is compatible with the WS-I (Web Services Interoperability Organization) Basic Profile for Web Services. It combines the HTTP transport binding element with the (UTF-8) text encoding, transport-level HTTP security, and—unless otherwise configured—the SOAP 1.1 envelope format.

```
using (ServiceHost<People> peopleHost = new ServiceHost<People>() )
{
    peopleHost.AddEndpoint(
        typeof(IPeople),
        new BasicProfileBinding(),
        "http://localhost:8080/people");
    peopleHost.Open();
    Console.WriteLine("Press ENTER to close");
    Console.ReadLine();
    peopleHost.Close();
}
```

To bind a service implementation to an HTTP address using the Basic Profile, we simply need to add an endpoint to the service host that combines a contract, the selected binding and the address as shown above in the explicit hosting code snippet.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.serviceModel>
    <services>
      <service serviceType="SelfHostedService.People">
        <endpoint address="http://localhost:8080/people"
          bindingSectionName="basicProfileBinding"
          contractType="SelfHostedService.IPeople"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

If you don't want to embed the binding details in the source code, and this will be the most common case because it's much more flexible, you can also configure the service endpoints in the application configuration file or, in the case of an ASP.NET hosted service, in the web.config file. To do so, you need a **<system.serviceModel>** configuration section with a **<services>** subsection that contains a **<service>** element for each service your application is hosting. The configured service class is identified by the **serviceType** attribute.

Each endpoint that the service shall expose is configured with an **<endpoint>** element beneath the respective **<service>** element. The **address** attribute specifies the network address of the service, the **bindingSectionName** refers to the predefined or custom binding to be used and the **contractType** points to the interface class that hold the WCF **[ServiceContract]** for the endpoint contract. If a class implements multiple service contracts, you must configure an endpoint for each contract and all contracts must reside at distinct network addresses. A contract implementation for which no endpoint is configured is not reachable—everything is explicit.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.serviceModel>
    <client>
      <endpoint configurationName="myClient"
        address="http://localhost:8080/people"
        bindingSectionName="basicProfileBinding"
        contractType="SelfHostedService.IPeople"/>
    </client>
    <bindings>
      <basicProfileBinding>
        <binding configurationName="MyBpBinding"
          securityMode="HttpAuthenticationOverHttps">
          <httpAuthentication scheme="Digest" realm="example.com"/>
        </binding>
      </basicProfileBinding>
      <wsProfileBinding>
        <binding configurationName="MyWsBinding"
          securityMode="WSSecurityOverHttp">
          <wsSecurity authenticationMode="Windows"
            protectionLevel="EncryptAndSign"/>
        </binding>
      </wsProfileBinding>
      <netProfileTcpBinding>
        <binding configurationName="MyTcpBinding"
          securityMode="WSSecurityOverTcp">
          <wsSecurity authenticationMode="Windows"
            protectionLevel="EncryptAndSign"/>
        </binding>
      </netProfileTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

```
        </binding>
    </netProfileTcpBinding>
</bindings>
<services>
    <service serviceType="SelfHostedService.People">
        <endpoint address="https://localhost:8079/people"
            bindingSectionName="basicProfileBinding"
            bindingConfiguration="MyBpBinding"
            contractType="SelfHostedService.IPeople"/>
        <endpoint address="http://localhost:8080/peopleWS"
            bindingSectionName="wsProfileBinding"
            bindingConfiguration="MyWsBinding"
            contractType="SelfHostedService.IPeople"/>
        <endpoint address="net.tcp://localhost:8081/people"
            bindingSectionName="netProfileTcpBinding"
            bindingConfiguration="MyTcpBinding"
            contractType="SelfHostedService.IPeople"/>
    </service>
</services>
</system.serviceModel>
</configuration>
```

A service can define any number of endpoints for each contract it implements. If you want a service to listen to messages on HTTPS using the **BasicProfileBinding** with transport-level authentication, and on HTTP using the **WSProfileBinding** with WS-Security in-message authentication, alongside the secure-by-default **NetProfileTcpBinding** for binary TCP transport, you can easily do so by adding the respective endpoints to the configuration and clients can subsequently talk to the service through any endpoint they prefer. The service listens to all these endpoints in parallel.

As you can also see in the example above, each of the predefined bindings supports some degree of customization of their respective features. The **<bindings>** section holds a subsection for each of the bindings that WCF predefines and you can add any any number of named customizations to these subsections. The concrete binding that is used for an endpoint is selected by specifying the **bindingSectionName** and the respective **bindingConfigurationName**.

Binding Requirements and Validation

So let's assume you have a service implementation that requires all callers (or all message senders, depending on how you look at it) to be authenticated. Or maybe your service demands that all messages sent to it must be delivered through a reliable transport so that messages cannot get lost on the way. The problem that immediately arises in that case is that the simple, ASMX compatible **BasicProfileBinding** without further configuration is not secured and doesn't support reliable messaging, at all.

Because it's up to the implementer (meant is the person writing the configuration file) to choose a binding, the developer must have a way to define what the minimum requirements for a service endpoint are. Some of these requirements, such as the demand for session support are already expressed on the **[ServiceContract]** attribute and others can be expressed using the **[BindingRequirements]** attribute.

```
[BindingRequirements(  
    TransactionFlowRequirements=RequirementsMode.Require)]  
public class People : IPeople  
{ ... }
```

Using the **[BindingRequirements]** attribute, the developer can impose constraints on the concrete bindings used to expose the service to clients. This is important, for instance, if the service's architecture calls for the service to be fronted with a queue and to accept only messages that are sent using through reliable delivery channel. If the developer would allow an implementer to use an arbitrary transport and binding, this would mean a violation of the architecture rules defined for the service and therefore, possibly, in adverse performance and throughput characteristics under load and, even more severely, in data loss. Since it cannot necessarily be expected from an implementer to know about all the intricate details of the application's architecture, the **[BindingRequirements]** are the developer's way to demand from the WCF runtime that these rules are enforced.

The binding requirements of a service are validated against the actual capabilities of a binding when the service host for the service is opened. If any of the endpoint bindings does not support the binding requirements of a service, an exception is thrown by the respective binding validator (supplied by WCF) and the service host refuses to start in order to avoid inappropriate behavior or, worse, security holes.

Clients

Now we went through all these details of the server side of a service and we haven't even touched the client side, yet. Luckily, WCF is very "symmetric". Most of the things that you know already about addresses, bindings, and service contracts apply to clients in the same way. In fact, implementing a service client is quite often mostly a matter of using the right tool, just as with ASP.NET Web services.

When you install WCF onto a machine where Visual Studio 2005 is already present, WCF adds its own "importer" to Visual Studio. That importer causes WCF contract definitions and WCF service proxies to be emitted alongside the ASP.NET proxies whenever you add a "Web reference" to a service. With that, WCF service clients are implemented for you—almost automatically. What this particular importer doesn't do—at least not with the present Beta bits—is modify the application configuration file to embed the appropriate binding information for the service as it exists in the service metadata.

```
Svcutil.exe /config:myclient.exe.config /out:myclient.cs  
http://www.example.com/myservice.svc?wsdl
```

However, the standalone tool svcutil.exe that was already mentioned right at the beginning of this article is indeed capable of importing not only the contract information and to generate contract proxies, but also has a /config switch, which turns the policy information of a service endpoint's binding metadata into a client-side, custom-binding configuration that is compatible with the service binding.

```
using (ChannelFactory<IPeople> ipeopleFactory =
new ChannelFactory<IPeople>(
    new Uri("http://localhost:8080/people"),
new BasicProfileBinding()))
{
    ipeopleFactory.Open();

    IPeople proxy = ipeopleFactory.CreateChannel();
    proxy.StorePerson( ... );

    ipeopleFactory.Close();
}
```

But you don't really need to use the tools. You can also reference or copy the **[ServiceContract]** declaration directly into your client code. Referencing or copying the service contract declaration code (!) is indeed faithful to the service-orientation spirit of "shared contract"—you are simply copying a metadata definition. Whether that metadata is expressed in WSDL or is expressed using CLR attributes is secondary. And of course, you can also create an assembly that contains the service contract declarations, distribute it, and reference it from both server and client side, even though that causes the two sides to be slightly tighter coupled to each other since they cannot make individual modifications on the contract by adding individual demands and requirements for the bindings. However, whether you want to share contract through WSDL only (using the tools) or whether you want to share the code-based declarations or already compiled metadata—it's entirely your choice. Any of these variants works.

Building a client is as simple creating a configured **ChannelFactory<T>** for the service contract. The channel factory, whose type already embeds the contract (the "C"), must then logically be configured with an address and a binding to complete the WCF "ABC".

Specifying target address and binding can be done explicitly on the channel factory constructor call or, indirectly and just like on the server side, through the application configuration file. Just as on the server side, using configuration is superior to the code-based alternative due to the flexibility and because you can mostly copy and paste the server-side definitions into the client side configuration, if you have access to them.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.serviceModel>
    <client>
      <endpoint configurationName="myClient"
        address="http://localhost:8080/people"
        bindingSectionName="basicProfileBinding"
        contractType="SelfHostedService.IPeople"/>
    </client>
  </system.serviceModel>
</configuration>
```

The link between the configuration file's **<client>** section and the channel factory is established through the **contractType** attribute on one or more of the **<endpoint>** definitions that match the generic type-argument of the channel factory and the **configurationName** value that is matched against the **ChannelFactory<T>** constructor argument for the configuration to pick:

```
using (ChannelFactory<IPeople> ipeopleFactory =
new ChannelFactory<IPeople>("myClient"))
{
    ...
}
```

Once you have a channel factory, independent whether directly or indirectly configured, you can open it, which initiates, just as on the server side a validation process that checks the contract binding requirements against the bindings. If no validation exceptions occur, you can initiate a new session with the service by calling **CreateChannel()** on the channel factory and calling methods on the returned service contract interface proxy implementation.

Conclusion

WCF is a very rich technology platform for building distributed systems, unifying messaging-style and RPC-style communication, efficient platform-optimized binary communication and open-standard based Web services, and a rich set of features enabling robust security, scalable and reliable communication paths and transactional processing.

All these great features aside, it's truly amazing—at least to the author—how simple, clean, and straightforward the programming model is, and how flexibly services can be configured. And even if it may sound a bit unprofessional: Building powerful services with WCF is so easy, it's simply great fun.