

A Quick look at the Windows Communication Foundation

by Mark Smith

WCF is Microsoft's über framework for communication combining all the best features of Web services, .NET remoting and MSMQ into one platform. It allows developers to focus more on the services they want build instead of the network plumbing used to expose them. Like .asmx web services, WCF is driven through attributes – building a service involves defining the contract (generally as an interface) and decorating it with attributes to supply the intent:

```
[ServiceContract]  
interface IReverser  
{  
    [OperationContract]  
    string Reverse(string input);  
}
```

The **[ServiceContract]** attribute marks the service itself, much the same way **[WebService]** is used in the .asmx framework. The **[OperationContract]** allows WCF to expose that method from the service – without it, WCF will ignore the method altogether just like **[WebMethod]**.

The service itself would then supply an implementation of the above interface and ask WCF to expose the implementation through an endpoint. The endpoint describes the location of the service (address), the protocol and mechanics used to transfer information (binding) and the methods exposed (contract). This information can be done in code, but is generally contained in the configuration file as it tends to be based on where the service is being deployed and the expected client locations. In this sense, WCF behaves much like remoting, which had the same sort of information stored in a configuration file.

While WCF allows you to interact directly with the messages passed along the wire (through the **Message** class), it's more common to use .NET types in the operation contract definitions and allow the framework to serialize basic and complex types into messages. Basic types such as **System.String** and **System.Int32** can be serialized natively by WCF, but complex types need a little help from the developer. Like remoting, we need to signal our intent to allow a complex type to be serialized and we can supply that in one of three ways: **[Serializable]**, **[XmlSerializerFormat]**, or **[DataContract]** which is the preferred way. The first two options allow for an easy transition from an existing remoting or web services solution.

As an example, consider a system that exposes the pubs database and allows a remote client to retrieve the collection of titles:

```
Book[] GetTitles();
```

In this case, the **Book** class would be marked as **[Serializable]** enabling it to be marshaled from the server to the client through the remoting infrastructure. This would allow a Windows Forms client to bind to the array and display the data through a grid or form. Converting this to a WCF solution is easy because the framework still respects the serializable attribute:

```
[ServiceContract]
interface IPubsDatabase
{
    [OperationContract]
    Book[] GetTitles();
}
```

The Windows Forms client would now utilize WCF to retrieve the data (in a secure, authenticated fashion if necessary) and would get an array of Book objects – just like the remoting version. In this case, all fields (public and private) are serialized and carried over to the client copy.

In the same way .asmx web services, which rely on the **XmlSerializer**, can be easily ported to WCF by decorating the service contract interface with the **[XmlSerializerFormat]** attribute. This indicates to WCF that it should utilize the same serialization method utilized by .asmx. Changing from .asmx to WCF involves updating the attributes on the service implementation/interface and either changing the host or moving the code into a .svc file so it gets managed by WCF. Data structures utilized by the web service contract would not need to change. The normal **XmlSerializer** rules are followed in this case (i.e. only public fields/properties are exposed and a public, default constructor is required):

```
public class Book
{
    public string Author, Title;
    public Book() { }
    public Book(string title, string author) { ... }
}
```

```
[ServiceContract]
[XmlSerializerFormat]
public class Library
{
    [OperationContract]
    public Book GetMostPopularBook() {
        return new Book("Harry Potter and the Deathly Hallows",
            "J.K. Rowling");
    }
}
```

While both of these mechanisms will provide a workable service, the preferred method in WCF is to utilize a set of new attributes to define an explicit mapping from the .NET type into a data stream. It

starts with the `[DataContract]` attribute which is applied to the complex type itself, and then `[DataMember]` which is used to expose specific fields and properties of the type. The visibility of the field/property is no longer a factor. Both of these attributes are actually in the `System.Runtime.Serialization` assembly:

`[DataContract]`

```
class Book
{
    [DataMember]
    string Author;
    [DataMember]
    public string Title;
    public string ISBN; // will not be sent to client
    public Book(string title, string author) { ... }
}
```

`[ServiceContract]`

```
public class Library
{
    [OperationContract]
    public Book GetMostPopularBook() {
        return new Book("Harry Potter and the Deathly Hallows",
            "J.K. Rowling");
    }
}
```

This provides an explicit contract to determine what is exposed to the client. The `DataMember` attributes determine the schema information that allows the client to generate an equivalent data structure to manage the information.

An easy way to get WCF to generate these data contracts is through the `svcutil.exe` tool. This tool serves the same purpose as `wSDL.exe` in the `.asmx` world – it generates proxies and data structures from metadata information. The `svcutil.exe` tool has a nice contract-first feature which enables it to take an `.XSD` describing a data structure and it will generate the appropriate data contract to represent it in code. For example, given the following `xsd` definition:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="Book" targetNamespace="http://tempuri.org/Book.xsd"
elementFormDefault="qualified" xmlns="http://tempuri.org/Book.xsd"
xmlns:mstns="http://tempuri.org/Book.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="Book">
    <xs:sequence>
      <xs:element name="Author" type="xs:string" />
      <xs:element name="Title" type="xs:string" />
      <xs:element name="ISBN" type="xs:string" />
      <xs:element name="Price" type="xs:decimal" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```
</xs:sequence>  
</xs:complexType>  
</xs:schema>
```

We can generate the C# code to model this in WCF by calling `svcutil.exe` and passing the **/donly** switch:

```
Svcutil.exe /donly books.xsd
```

This will create a new C# source file describing our data structure in WCF form. The same .XSD could also be used by other tools to generate technology-specific versions of this data structure – allowing us to ensure that we don’t break a primary rule of service orientation – sharing based on schema instead of types.

Microsoft has put a lot of thought and effort into creating an extensible platform that existing solutions can migrate to with minimal effort while still allowing new solutions to take advantage of the latest standards and architecture styles. It truly is a “best-of-breed” technology.