Develop Interface-Based .NET Web Services

Building interface-based Web Services lets you switch between different service providers with minimal or no changes to client code.

by Juval Lowy

Technology Toolbox

- ☐ VB.NET
- ☑ C#
- ☐ SQL Server 2000
- ☐ ASP .NET
- ☐ XML
- ☐ VB6
- ☑ Other:
  Visual Studio .NET beta 2 or later
  Windows 2000 or Windows XP beta 2 or later

Resources

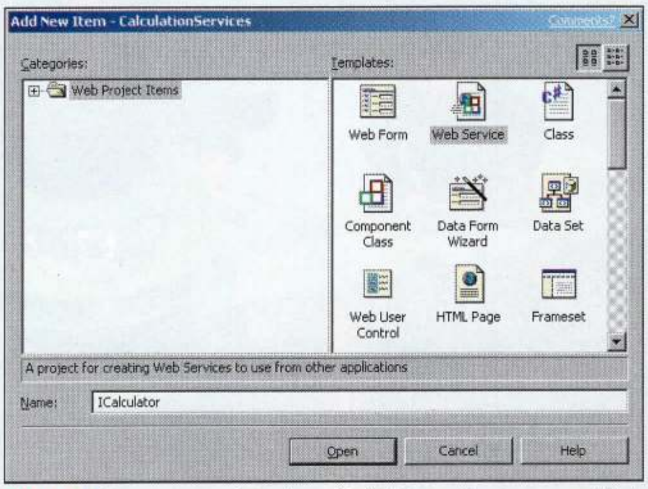
MSDN for Visual Studio .NET

As you well know from traditional component-oriented programming, separating the interface from its implementation provides polymorphism between different implementations of the same service. The separation recognizes the fact that the basic unit of reuse in any application is the interface, not the object implementing it. You can apply this core principle of component-oriented programming to Web Service development as well.

In this case, the interface is a logical grouping of method signatures that act as the contract between the client and the Web Service provider. You can then switch between different providers with minimal or no changes to the client code because the client is written against an abstract service definition (the interface) rather than a particular service implementation.

The Web Services standard supports interfaces (referring to them as ports). But by default, Web Services support in .NET is method-based, not interface-based. So VS.NET, as it exists today, doesn't inherently allow you to develop interface-based Web Services. I'll show you the simple steps required—both on the server and client—to compensate for the lack of interface-based Web Services in VS.NET, allowing you to develop and consume interface-based Web Services. But first I'll set up an example scenario.

Suppose you have a Web Service called SimpleCalculator that provides the four basic arithmetic operations—addition, subtraction, division, and



A project for creating Web Services to use from other applications

Name: ICalculator

Figure 1 Expose an Interface Definition. To expose a Web Service interface definition, add a new Web Service item and give it an interface name. Clicking on the Open button causes VS.NET to create a skeletal Web Service. From there, remove all the implementation code from the wizard-generated code.

multiplication—and a client that consumes the Web Service. You implement the SimpleCalculator Web Service in .NET using C# (see Listing 1).

Simply add the [WebMethod] attribute to the methods you want to expose as Web Services—Add(), Subtract(), Divide(), and Multiply()—and .NET does the rest. Note that having WebService as a base class is optional; deriving from it gives you easy access to common ASP .NET objects, such as those for application and session states. You don't need these objects in this use-case. Also optional is the WebService attribute, but I strongly recommend you use it.

The WebService attribute lets you specify a Web namespace that contains your service, which you can use as you would a normal .NET namespace to reduce name collusion. If you don't specify a namespace, VS.NET uses http://tempuri.org/ as a default. A published service usually uses a specific Uniform Resource Identifier (URI) as its namespace, typically the service provider company's name. The WebService attribute also allows you to provide a free-text description of your service that appears in the auto-generated browser page used by Web Service consumers and during development.

Writing the client code is almost equally trivial. Select Add Web Reference… from the client's project in VS.NET and point the wizard at the site containing the Web Service ASPX file. This makes VS.NET generate a wrapper class—called SimpleCalculator—that the client will use (see Listing 2). The SimpleCalculator wrapper class has all the methods the Web Service developer applied [WebMethod] to as public methods. The wrapper class (sometimes called a "Web Service Proxy" class) encapsulates completely the complex interaction with the remote object. The wrapper class is also the only entity coupled to the service's location—its base class, SoapHttpClientProtocol, has a property called Url that points to the object's location.

The client code uses the wrapper class as if the SimpleCalculator object were a local object:

```
SimpleCalculator calculator;
calculator = new SimpleCalculator();
int result = calculator.Add(2,3);
Debug.Assert(result == 5);
```

Clearly, VS.NET makes invoking Web methods trivial.

But you have a problem: The client ends up programming directly against the "object" providing the service (SimpleCalculator in this case) instead of against a generic abstraction of such a service. You want the SimpleCalculator Web Service polymorphic with a service abstraction—an interface.

For example, imagine the client wants to switch from the SimpleCalculator to a different calculator Web Service, called ScientificCalculator. ScientificCalculator supports the same interface as SimpleCalculator, but it's perhaps faster, cheaper, or more

C# • It All Adds Up With the SimpleCalculator Class

```
[WebService(
Namespace=
"http://CalculationServices.com",
Description = "The SimpleCalculator Web Service provides the four
basic arithmetic operations for integers.")]
public class SimpleCalculator: WebService
{
   public SimpleCalculator(){}
   [WebMethod]
   public int Add(int num1,int num2)
   {
      return num1 + num2;
   }
   [WebMethod]
   public int Subtract(int num1,int num2)
   {
      return num1 - num2;
   }
   [WebMethod]
   public int Divide(int num1,int num2)
   {
      return num1 / num2;
   }
   [WebMethod]
   public int Multiply(int num1,int num2)
   {
      return num1 * num2;
   }
}
```

Listing 1 To expose a class method as a Web Service, simply add the [WebMethod] attribute. Note that deriving from WebService is optional. Using the [WebService] attribute is optional also, but you should use it to provide a service description and containing namespace.

C# • Encapsulate the Interaction

```
public class SimpleCalculator : SoapHttpClientProtocol
{
   public SimpleCalculator()
   {
      Url = "http://www.CalculationServices.com/SimpleCalculator.asmx";
   }

   [SoapDocumentMethod("http://CalculationServices.com/Add")]
   public int Add(int num1,int num2)
   {
      object[] results = Invoke("Add", new object[]{num1,num2});
      return (int)(results[0]);
   }
   // Other method wrappers
}
```

Listing 2 The SimpleCalculator Web Service wrapper class—generated for the Web Service shown in Listing 1—encapsulates the interaction with the Web Service completely and shields the client from the details. It contains the service location in the public Url property, which is defined in the SoapHttpClientProtocol base class.

accurate. You'd like to define a generic calculator interface, the ICalculator interface, and expose it as Web Service:

```
// Imaginary attribute.
// Does not exist in .NET
[WebInterface]
interface ICalculator
{
   int Add(int num1,int num2);
   int Subtract(int num1,int num2);
   int Divide(int num1,int num2);
   int Multiply(int num1,int num2);
}
```

Assuming you could do that (I'll show you how shortly), you can code against only the interface definition instead of a particular implementation of it (see Listing 3).

C# • Prepare for a Service Provider Change

```
// Somewhere in the client code, it
// decides on the service provider:
ICalculator calculator = (ICalculator) new ScientificCalculator();
// or

ICalculator calculator = (ICalculator) new SimpleCalculator();

// This part of the client code is
// polymorphic with any provider of the service:

int result = calculator.Add(2,3);
Debug.Assert(result == 5);
```

Listing 3 Ideally, you want your calculator Web Services polymorphic with a service abstraction—an interface. You can then switch between service providers with minimal or no changes to the client code.

The only thing that changes in the client's code when it switches between service providers is the line that decides the exact interface implementation to use. You can even put that decision in a different assembly than the "main" client's logic, and you can only pass interfaces between the two. Another benefit you can reap from interface-based Web Services: The client can publish the interface definition, enabling different service vendors to implement the client's requirements more easily.

Now you're ready to write the server and client code to work around VS.NET's lack of support for interface-based Web Services.

VS.NET, as it exists today, doesn't inherently allow you to develop interface-based Web Services.

Define and Implement a Web Interface

To enable an interface-based Web Service, first expose the Web Service interface definition. For simplicity's sake, assume the service provider is responsible for both defining and implementing the interface (the client or any third party can expose the interface definition and have anybody implement it, but it requires additional steps).

Create a new Web Service project called CalculationServices. Right-click on the project and select Add Web Service…. In the Add New Item dialog, type "ICalculator" as the interface name and click on Open (see Figure 1).

VS.NET then creates a skeletal Web Service called ICalculator. Open the ICalculator.asmx.cs file and change the ICalculator type definition from "class" to "interface." Remove the derivation from System.Web.Services.WebService, as well. An interface, by definition, has no implementation code—remove the constructor and the InitializeComponent() and Dispose() methods. Finally, remove the commented HelloWorld() method example.

Next, add the interface methods—Add(), Subtract(), Divide(), and Multiply(). Although in principle you could simply apply the [WebMethod] attribute to each interface method to expose the interface as a Web Service definition, you shouldn't in practice because of the [WebService] attribute. This attribute applies only to classes, and it's sealed—you can't subclass it or change it. So you can't assign a namespace and a description to the interface, which I advised you to do earlier. To clear this hurdle, you must provide an interface "shim"—an abstract class that exposes what looks like a pure interface definition. In the ICalculator.asmx.cs file, add the ICalculatorShim pure abstract class definition:

```
[WebService(
Name = "ICalculator",Namespace=
"http://CalculationServices.com",
Description = "This Web Service is only the definition of
the interface. You cannot invoke method calls on it.")]
abstract class ICalculatorShim : ICalculator
{
abstract public int
Add(int num1,int num2);

abstract public int
Subtract(int num1,int num2);

abstract public int
Divide(int num1,int num2);

abstract public int
Multiply(int num1,int num2);
   }
```

Note that because ICalculatorShim is a class, you can use the [WebService] attribute to provide a namespace and description. In addition, you set the Name property of the [WebService] attribute to ICalculator to expose the service definition as ICalculator instead of ICalculatorShim.

You're interested in exposing only the signatures, so you don't need the implementation code. Because you're using an abstract class and abstract methods, VS.NET insists that the ICalculatorShim Web Service have no implementation code—only a service definition.

To verify that all is well so far, set the ICalculator.asmx file as the start page and run the project; the auto-generated browser page presents the ICalculator interface definition. If you try to invoke any of the methods, you should get an error because there's no implementation behind the service.

Next, implement the ICalculator interface on a Web Service class. This is like implementing any other interface in .NET—your Web Service class should inherit from the interface and provide the implementation for its methods. In this example, provide two class implementations: the SimpleCalculator and the ScientificCalculator Web Services (download Listing 4 from the *VSM* Web site; see the Go Online box for details).

Use the Add Web Service… context-menu item again to add the two Web Services. Add a derivation (inheritance) from the ICalculator interface (you can remove the derivation from the WebService base class—it bears no relevance to interface-based Web Services). Add the implementation to the Add(), Subtract(), Divide(), and Multiply() interface methods. You must provide the [WebMethod] attributes for the interface methods you want to expose as Web Services. Without [WebMethod] on a class implementation method, .NET won't expose the method as part of the Web Service.

Write the Client-Side Code

The client needs to add a reference to the type definitions of the interface and the classes implementing it. You can add the interface reference in one of two ways. The first uses the WSDL.exe command-line utility. Using the /server switch, you can instruct WSDL.exe to generate a pure abstract class matching the Web Service definition. Assuming the interface definition resides at http://www.CalculationServices.com/ICalculator.asmx, run the utility with this command line:

```
WSDL.exe /server /out: ICalculatorDef.cs
http://www.CalculationServices.com/ICalculator.asmx
```

Then add the ICalculatorDef.cs source file to the client project. Unfortunately, even though .NET knows about interfaces, the /server switch generates a pure abstract class with abstract methods:

```
public abstract class ICalculator : WebService
{
   [WebMethod]
   [SoapDocumentMethodAttribute("http://
CalculationServices.com/Add"]
   public abstract int Add(int num1, int num2);
   // rest of the ICalculator methods
}
```

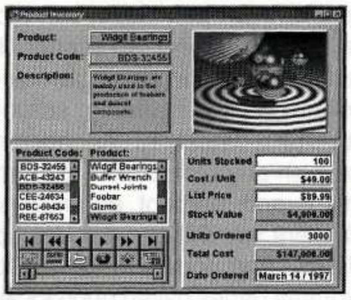
But what you need is an interface definition. Open the ICalculatorDef.cs file, remove the WebService base class, and change the ICalculator definition from abstract class to interface. Remove all the attributes (on ICalculator and its methods) and the public and abstract modifiers from all the methods. You should now have the original interface definition.

The second way a client can import the interface definition: Add a Web reference to the ICalculator Web Service, then extract the interface methods from the wrapper class. To do so, point the Add Web Reference… wizard to the site containing the interface definition. This generates a wrapper class called ICalculator, which exposes the original ICalculator's methods as well as a Web Service wrapper class's other methods. You need only method definitions for an interface, so remove all the interface method bodies and the other methods completely, including the constructor. Remove the SoapClientProtocol base class and the "public" modifier on the interface method. Remove all class and method attributes. Lastly, change the ICalculator definition from "class" to "interface." Essentially, the client should now have the original interface definition.

Next, the client must add a Web reference to the Web Services that provide the interface's implementation. Again, using the Add

The *Fastest* Database Engine

Discover **CodeBase**, the **fastest** database engine on the market that's also fully **xBASE compatible.** Query a million-record table in 0.34 seconds, or read 300,000 records in just 0.59 seconds. It's lightning quick!



ADO/ODBC Support! This application was created without a single line of source code! Use native and 3rd party controls with our new ADO/OLE-DB and ODBC support. Creating database programs has never been easier!

✓ Multi-user file compatible with FoxPro, Clipper and dBASE
✓ Client/Server, multi-user & single user support included
✓ Small DLL loads quickly, simply and use few resources
✓ 100% Portable: Windows, Pocket PC etc. 32 & 64 bit
✓ Supports VB, C, C++, Delphi, Java etc.
✓ Reader's Choice Award for 5 Years
✓ Royalty Free Distribution

FREE 30 Day Test Drive!

SEQUITER® SOFTWARE INC.

www.sequiter.com - info@sequiter.com Phone: (780) 437-2410 Fax: (780) 436-2999

Web Reference… wizard, point the wizard to where those implementations reside. VS.NET generates wrapper classes for those implementations—SimpleCalculator and ScientificCalculator, in this example. These machine-generated wrapper classes won't mention ICalculator. To provide polymorphism with ICalculator, add a derivation from it. The SimpleCalculator and ScientificCalculator classes should now look like those in Listing 5, which you can download.

Here's the client-side design pattern: ICalculator provides the service definition. The Web Service's location is decided at the derived SimpleCalculator or ScientificCalculator classes. The wrapper classes know how to forward the calls to the Web Service, but not how it's implemented on the server side. In fact, all that distinguishes SimpleCalculator from ScientificCalculator is the encapsulated service location or provider.

Finally, you can now write interface-based, polymorphic Web Services code (see Listing 3). You can also make an interesting observation: From the client's perspective in the Web Services world, the location of the service—the URL—is the object.

Web Services will become part of almost every developer's career in the next few years. However, the supporting tools are immature compared to existing design methodologies and component-oriented technologies you've grown accustomed to in the intranet world. Today's challenge is how to combine the two. I hope this article convinces you not to give up on proven and elegant concepts just because VS.NET doesn't support them. With a bit of tweaking and the proper observation that a URL equals an object, you can employ established concepts in a brave new world. VSM

Juval Lowy is a seasoned software architect and the principal of IDesign, a consulting company focused on COM/.NET design. Juval also conducts training classes and gives conference talks on component-oriented design and development. He wrote the book *COM and .NET Component Services – Mastering COM+* (O'Reilly). Reach him through www.componentware.net.

Go Online

Use these DevX Locator+ codes at www.vbpj.com or www.vcdj.com to go directly to these related resources.

VS0110 Download all the code for this issue of *VSM*.

VS0110WS Download the code for this article separately. This article's code includes a basic, non-interface–based Web Service; two interface-based Web Services; a test client that uses them all; and Listings 4 and 5.

VS0110WS_T Read this article online. DevX Premier Club membership is required.

Want to subscribe to the Premier Club? Go to www.devx.com.

New Version

PC-lint 8.0 for C/C++ Bug of the Month #651

```
#include <stdio.h>

struct { int a[3], b; } w[] = { { 1, 2, 3 }, 2 };

int main()
    {
    printf( "w[0].b = %d\n", w[0].b );
    return 0;
    }
```

The programmer expected to print the value '2'. Instead he got '0'. Why? Call if you need a hint or visit our web site at **www.gimpel.com**

PC-lint for C/C++ will catch this and many other bugs. It will analyze a mixed suite of C and C++ modules to uncover bugs, glitches, quirks and inconsistencies.

Not your Grandpa's lint: PC-lint has introduced several spectacular and revolutionary innovations in the art of static program analysis. Taking clues from initializers, assignments, and conditionals, variable and member values are tracked, enabling reports on potential uses of null pointers and out-of-bounds subscripts.

New with Version 8: Interfunction value tracking – Actual argument values are used to initialize parameters; return values are computed; a multi-pass operation (you control the number of passes) allows you to plumb the depths of function behavior to arbitrary levels.

Plus Our Traditional C/C++ Warnings: Uninitialized variables, inherited non-virtual destructors, strong type mismatches, ill-formed macros, inadvertent name-hiding, suspicious expressions, etc., etc.

Full Language Support for ANSI/ISO C and C++.

PC-lint for C/C++ *$239*
Numerous compilers/ libraries supported. Runs on Windows, MS-DOS, and OS/2.

FlexeLint for C/C++
The same great product for other operating systems. Runs on all UNIX systems, VMS, mainframes, etc. Distributed in shrouded C source form. Call for pricing.

30 Day Money Back Guarantee

Gimpel Software

Serving the C/C++ Community for 15 Years.

3207 Hogarth Lane, Collegeville, PA 19426

CALL TODAY (610) 584-4261 Or FAX (610) 584-4266

www.gimpel.com

PA add 6% sales tax.

PC-lint and FlexeLint are trademarks of Gimpel Software