Issue	.NET Magazine
Section	Features
Main file name	**nm0101jlt8.rtf**
Listing file name	**nm0101jll2.rtf (2 listings)**
Sidebar file name	
Table file name	
Screen capture file names	**nm0110jlf2.bmp, nm0110jlf3.bmp, nm0110jlf4.bmp**
Infographic/illustration file names	**nm0110jlf1.doc**
Photos or book scans	
Special instructions for Art dept.	
Editor	DD
Status	content edited by DD (t4); revised by author (t5); revised by DD (t6); submitted to ME (t7), copy edit by SP (t8)
Spellchecked (set Language to English U.S.)	yes
EN review	approved raw
Character count	text 20,256; listings 2,003; total 22,259
Package length	5.5 pages
ToC blurb	Despite the presence of firewalls on the Internet, .NET gives you connectivity with remote components.

Overline:

W e b S e r v i c e s

Byline:

by Juval Lowy

What you need:

Visual Studio .NET Beta 2 or later

Windows 2000 or Windows XP Beta 2 or later

Resources:

- Microsoft .NET Development: http://msdn.microsoft.com/net

Head:

Web Services Hurdle the Firewall

Deck:

Use Web Services in .NET to invoke components across the Internet just as you do locally.

Web Services support is an exciting feature of the .NET platform, and arguably its single major market driver. Web Services allow one component to invoke methods on another component across the Internet, with the same ease as if the remote component were in a local assembly. Web Services support is analogous in many ways to the support DCOM provides for component connectivity and interoperability within the same local-area network (LAN). To introduce Web Services, I'll first describe the problems facing developers today when trying to connect components in two different Web sites, then walk you through construction and use of a .NET Web Service.

Even though most business Web sites are designed conceptually using the same architecture principles, sites today have a hard time interoperating. The first problem is that the technology used to implement the business logic varies from one Web site to the next. Some sites use DCOM or MTS/COM+ and others use Java and RMI or EJB/CORBA to connect components. Other sites use raw C++ or proprietary technology altogether. Some sites run on Unix, others on Windows. The lack of common ground makes relying on any component technology to connect Web sites impractical. The second problem is that Web sites are contained behind an Internet firewall (see Figure 1).

The firewall is essential for the survival of a business Web site—it protects the site from malicious attacks and maintains data integrity and privacy. Without a firewall, the site is exposed to the unsafe no-man's-land called the Internet. A nonsecured Web site can't conduct business. The problem with a

firewall is that it disallows any kind of binary method invocation, except on pre-designated, guarded ports. As a result, even if all the Web sites use the same component technology, firewalls prevent calls from going through. Component technologies such as DCOM use binary format for method invocations, and because firewalls don't understand the content of the exchanged binary packages, they view the packages as a potential attack and block the calls.

The Internet today is a huge collection of individually useful but isolated islands of business logic and data. Developers handcraft the interoperability solutions between businesses. In fact, many dot-com companies offer exactly this sort of service. However, these solutions are proprietary. Dot-coms provide singular solutions that couple the interacting businesses, and are expensive and time-consuming to implement; versioning is a nightmare. This predicament is holding back the Internet from achieving its true potential.

These problems resemble the issues developers faced in the early '90s, before the dawn of COM, when they were trying to connect components in different binary units across process and machine boundaries. COM, with its language independence, interface definition language (IDL), binary compatibility, and location transparency, solved the problems that had prevented components on the same LAN from interoperating easily.

Web Services are to the Internet what COM is to LAN components. Web Services address the problems of firewall blocking and singular solutions elegantly and simply. The one thing common to business Web sites is that they all understand HTTP. That holds regardless of the Web server used—such as Internet Information Server (IIS) or Apache—or the programming language and component technology used in the sites' middle tier, or the platform they run on. Web Services use HTTP as a transport protocol to transport the call from one Web site to the next, much the same way DCOM uses remote procedure calls (RPC). To go through firewalls, web services serialize the method calls into HTTP-POST, HTTP-GET, or SOAP requests. All three are in text form, so firewalls simply let them through. The Web server receiving the HTTP/SOAP request interpret the request and invoke the call on the appropriate object.

Every Web Service is described in a special XML format called Web Service Description Language (WSDL). WSDL is to Web Services what IDL or a type library is to COM—a way of describing the methods and their parameters, and how to access the service. The WSDL description of the service and its wire representation is independent of component technologies, and is the result of the important players, such as Microsoft and IBM, reaching a consensus on a standard format and data types supported by Web Services.

WSDL is the contract between the service provider and its users. Once published, WSDL binds the service provider to the published method signatures. Changing WSDL or the Web Service methods results in breaking the consumer's code, much the same way changing a published COM interface breaks the client's code. In addition to method definitions, the service description also contains the address (URL) associated with a service, analogous to a COM type library that contains the class identifier (CLSID) of the COM servers implementing the interfaces.

Main Benefit: Decoupling

Web Services also define a rich infrastructure for discovering available Web Services, meaning, discovering what services are available for a particular category of service (such as banking, weather, investments, etc.) Unlike proprietary interoperability solutions, Web Services' main benefit is the high degree of decoupling between the service provider and its client—often called the consumer. By using XML-based messaging over HTTP as the mechanism by which the consumer creates and accesses the service, you can abstract the communication details away, resulting in what looks like a classic client/server component-oriented programming model. Both the client and the Web Service provider are freed from needing any knowledge of each other beyond inputs, outputs, and location. And you can even encapsulate these further, as I'll describe later.

.NET support for Web Services completely encapsulates the underlying Web Services plumbing from both the service provider and its consumer, much the same way Microsoft Foundation Class Library (MFC) encapsulates details such as WinProc, and Active Template Library (ATL) encapsulates class factories or IUnknown implementation. You can develop both a server and a client without ever interacting directly with a WSDL file, service discovery, or SOAP message.

The resulting programming model resembles that of the classic client/server model. Consumers simply create and use components that execute on a remote Web site. Using .NET Web Services, you can create an application that combines Web Services from multiple Web sites, viewing an entire Web site as just one component in your application.

I'll demonstrate this point by showing you how to construct and consume a sample Web Service—an integer arithmetic calculator. The calculator Web Service provides the four arithmetic operations for integers as a Web Service. Bring up the Visual Studio .NET New Project dialog, and select an ASP .NET Web Service project (see Figure 2). Name the project CalculationServices. VS.NET allows you to specify a location for the project files only under the IIS Home Directory.

This location is also known as a virtual directory—a disk folder that has a URL associated with it. By default, all Web Services reside under the home directory IIS folder—usually c:\inetpub\wwwroot—and you can manage them using the IIS Explorer.

As part of the new project, VS.NET creates a skeletal Web Service called Service1 in the Service1.asmx file. You have no use for it. Remove that file from the project. Next, you need to add the calculator Web Service. In the Solution Explorer, right-click on CalculationServices, select Add from the pop-up context menu, then Add New Item. This action brings up the Add New Item dialog. Under Web Project Item, select the Web Service icon, provide SimpleCalculator.asmx as its name, then click on OK.

Right-click on the SimpleCalculator.asmx file in the Solution Explorer and select View Code. This brings up the SimpleCalculator.asmx.cs C# source file associated with the SimpleCalculator.asmx file. The ASMX file contains a reference to the C# file and the C# class in it that implements the Web Service. This level of indirection allows you to refer to an already available Web Service, without having the service's sources in your project.

To expose a class method as a Web Service, you only have to add the [WebMethod] attribute to it. Add the four methods Add(), Subtract(), Divide(), and Multiply(), and the [WebService] attribute, on top of the SimpleCalculator class (see Listing 1).

It's important to emphasize four elements of developing a class that provides a .NET Web Service. First, only methods that have the [WebMethod] attribute on them are exposed. Other methods, public or private, aren't visible to Web consumers; they won't be part of the WSDL generated by VS.NET. Second, deriving from the base class WebService is optional. Doing so provides you with easy access to common ASP .NET objects, such as those for application and session states. Third, for every Web method invocation, .NET creates a new object, and destroys that object when the method returns. In essence, objects providing Web Services should be state-aware objects, such as a COM+ Just-in-time activation object or a transactional object, and retrieve their state and save it in every call.

Finally, the WebService attribute is optional as well. The WebService attribute allows you to specify a Web namespace that contains your service, used much the same way as you use normal .NET namespaces to reduce name collision. If you don't specify a namespace, VS.NET uses http://tempuri.org/ as a default namespace. It's recommended that a published service use a specific 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. VS.NET uses this description in the auto-generated test page.

Now your work developing the Web Service is done. VS.NET does the rest of the work for you, including creating a WSDL file that supports three different protocols for accessing the calculator service—HTTP-GET, HTTP-POST, and SOAP.

Test Your Web Service

Next, you need to test your Web Service. Web Services classes are designed to be accessed both over the Web and over the LAN. You can write a test client that uses the Calculator managed class like any other managed class, and invokes the methods directly by referencing the Calculator's assembly. However, effective testing means testing the Calculator class in the way the client would use it—over the Web.

VS.NET facilitates this sort of testing by generating a test page for your service. Actually, it uses only the HTTP-GET option of accessing the service, but usually that's all you need to step through your code. You can test the calculator Web Service by setting a break point in the Add() method, right-clicking on the SimpleCalculator.asmx file, selecting Set As Start Page, and running the project. VS.NET brings up the test page (see Figure 3).

The test page contains the service description—supplied by you in the WebService attribute—and lists the Web methods exposed by your Web Service. The test page also allows you to view the generated WSDL contract.

Click on the method's link to invoke one of the Web methods. For example, clicking on the Add link brings up a page dedicated to this method (see Figure 4).

Provide parameters to the Add() method and click on the Invoke button. The test page packages the parameters according to the WSDL contract and sends the request to the Web Service. For example, if you provide 2 and 3 as parameters for the Add() method and click on Invoke, VS.NET breaks at your break point, which allows you to step through the code. When the method returns, VS.NET brings up

another Internet Explorer window with the response from the Web Service; in this case that response is an XML string with the int 5 in it:

```
<?xml version="1.0" encoding=
   "utf-8" ?>
<int xmlns=
   "http://CalculationServices.com">
   5</int>
```

This response syntax is according to the HTTP-GET protocol.

An automatic test page is a nice feature, but you learn to appreciate VS.NET's support for Web Services when you develop client-side code. To do that, create a Windows Forms application called CalculatorClient. As a client developer, the last thing you want to do is parse WSDL files, compose HTTP requests, and parse enigmatic XML responses from Web Services. What you want is to create an object, use it, and get on with your life. That's exactly what VS.NET allows you to do, by generating a wrapper class, called a Web Service Proxy in the .NET terminology. The wrapper class is based on the Web Service's WSDL and it completely encapsulates the interaction—message-composing and reply-parsing—required to invoke a Web Service, as well as the service's location—its URL.

Create a Wrapper Class

You create a wrapper class by right-clicking on the CalculatorClient item in the Solution Explorer and selecting Add Web Reference. That brings up the Add Web Reference dialog. Then you specify the Web address of the service you're interested in, in the Address edit box. The left pane in the Add Web Reference dialog is actually an Internet Explorer window. You can navigate in that pane to the desired Web Service's location.

In the CalculatorClient example, the Web Service is on the same machine as the client, so you simply select Web References on the Local Web Server link in the left pane. Alternatively, you can specify any valid address on the Web that contains Web Services. Next, select the CalculationServices link from the right pane and click on Add Reference.

VS.NET adds an item to the Web Reference folder in the client's project in the Solution Explorer, referencing the Web Service, and generates the wrapper class. In the example, from the client's perspective, it's as if a new nested namespace called localhost is available with a new class in it: the SimpleCalculator class.

The wrapper class file is SimpleCalculator.cs; it contains the SimpleCalculator wrapper class definition. All the client has to do now is add a reference to the localhost namespace, create an instance of the SimpleCalculator class, and use it. To do so, add a button to the client form called Add, and add this code to the button clicked event handler:

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

The machine-generated SimpleCalculator wrapper class does all the interaction with the Web Service for you.

Unlike the auto-generated test page, the wrapper class uses SOAP by default to invoke the Web Service. In general, SOAP is a more extensible mechanism than HTTP-GET and HTTP-POST. All three protocols support the same standard set of data types such as int and strings, but SOAP invocations also support classes, structs, and datasets as parameters, as well as by-reference argument parameters. However, SOAP requires a larger payload to be transmitted back and forth, as well as more complex parsing than the simpler HTTP-GET and HTTP-POST formats. Use the WSDL.exe command-line utility if you want to generate a wrapper class that uses HTTP-GET or HTTP-POST formats, and specify the desired protocol as a command-line switch. VS.NET itself uses WSDL.exe to generate the SOAP wrapper class.

Note that the Web method's invocation is synchronous. The wrapper class blocks the client until the method returns from the Web Service or the invocation reaches a timeout. The wrapper class also contains asynchronous method wrappers, with some of the attributes and machine-generated code removed for clarity (see Listing 2). The asynchronous mechanism complies with the standard way of invoking a method asynchronously in .NET using delegates, except the wrapper class already contains the asynchronous methods, saving the trouble of defining a delegate.

Although a full discussion of asynchronous method invocation in .NET is outside the scope of this article, here's a brief description. In a nutshell, the client calls the asynchronous version of the Add() method—the BeginAdd() method—and gets an IAsyncResult object back immediately. The Add() method is executed asynchronously. The client can choose to use the IAsyncResult object to poll for the method completion, or to provide a callback method delegate to be called on the method completion. The client can also call EndAdd(), passing in the IAsyncResult object identifying that particular method invocation because the client might have more then one asynchronous method call in progress. EndAdd blocks the client if the method still executes. Otherwise, EndAdd returns to the client with the results.

The wrapper class's base-class—SoapHttpClientProtocol if you're using a SOAP wrapper class—has a property called Url, which contains the Web Service's address. The client developer can change that address manually to point to a different provider of the same service. However, doing so requires recompilation. A better approach is to package the wrapper class in a separate assembly altogether, and to refer to it only from the client's assembly. Doing so maintains location transparency in the client code, because if the service provider's address changes, only the wrapper class's assembly requires rebuilding. The client's code is unaffected.

Web Services exist to solve the problem of connecting individual Web sites into one computational Web. Even though the SimpleCalculator Web Service is a trivial one, it serves as a vertical slice of what's

required on both the server side and the client side to consume the service, and it demonstrates .NET's superb support for Web Services. You'll also want to explore on your own to learn about topics such as Web Services design guidelines, deploying a Web Service, Web Services debugging and error handling, Web Services state management, Web Services and transactions, Web Services and security, Web Services configuration, versioning a Web Service, and discovering Web Services.

About the Author:

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

Captions:

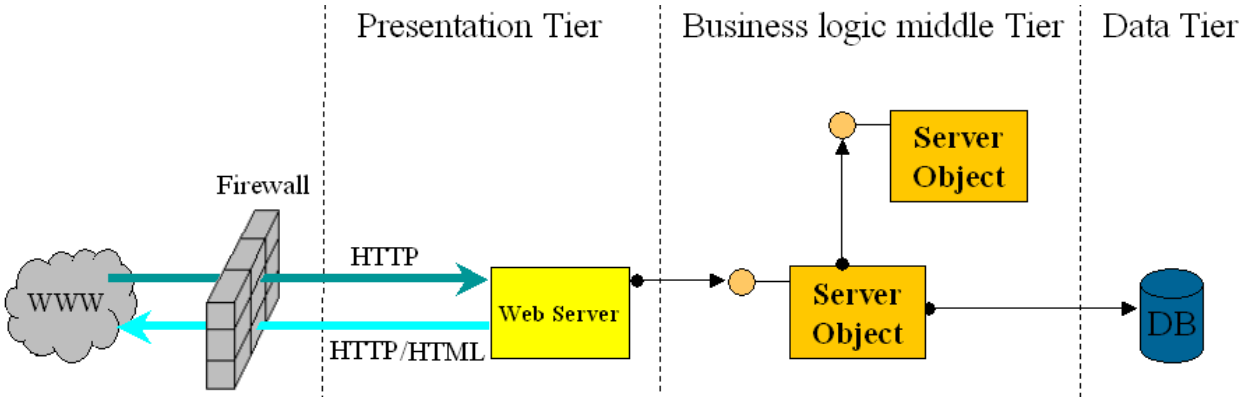
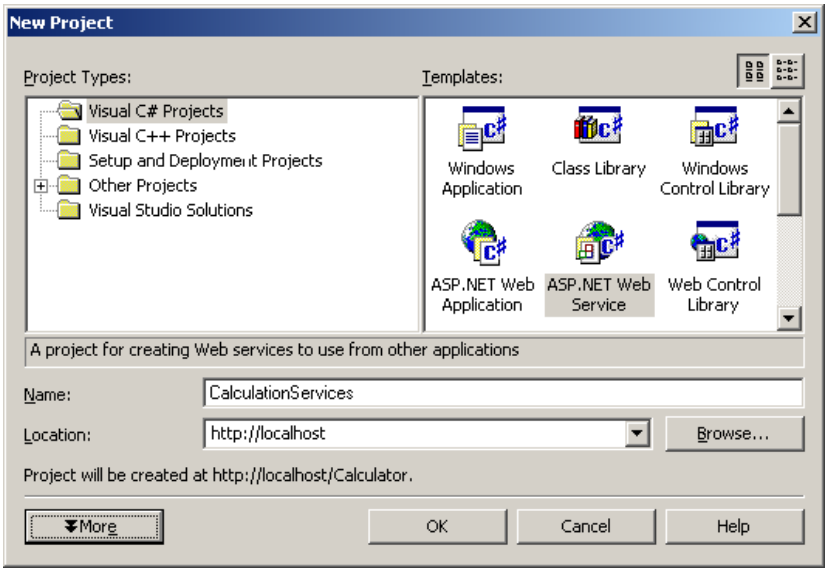


Figure 1

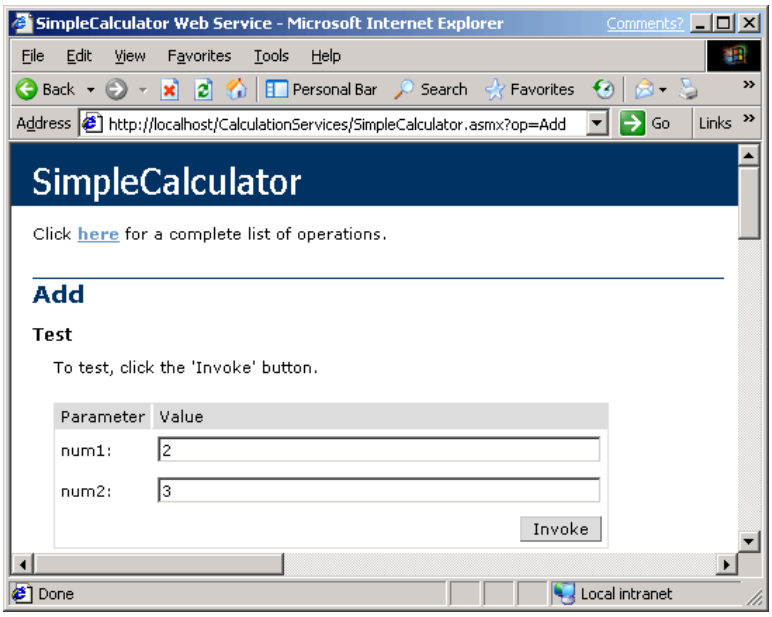
Understand Web Site Architecture. Most Web sites are the same architecturally, consisting of three or more tiers behind a firewall. Even though a given system's platform, language, and component technology is more or less homogeneous, systems have a hard time interoperating. Components developed in one technology can't invoke methods on components developed in another technology, and the firewalls disallow binary calls.

Figure 2



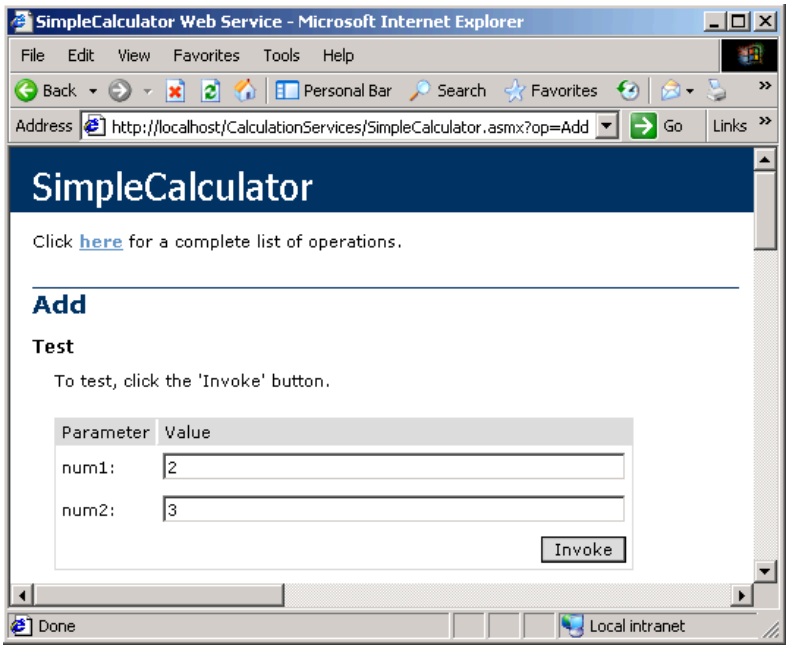
Create a New Web Service. Here in the VS.NET New Project dialog, the files for a new Web Service are located under the Home Directory IIS folder, in a new virtual directory. You can manage that directory from the IIS Explorer.

Figure 3



View the Test Page. A test page generated by VS.NET contains the service description and lists the Web methods exposed by your Web Service. The test page allows you to view your Web Service's WSDL contract in the Service Description link.

Figure 4



Invoke a Web Method. To invoke a specific Web method, VS.NET provides a dedicated page per method. You can provide the parameters, and click on Invoke. VS.NET then presents the returned value as an XML string in a separate window.

Pullquotes:

Web Services are to the Internet what DCOM is to LAN components.

The main benefit of Web Services is the high degree of decoupling between the service provider and its client.

Web Services exist to solve the problem of connecting individual Web sites into one computational Web.

Go Online

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