Nothin' but ASP.NET

**Web Services with ASP.NET**

Rob Howard  
Microsoft Corporation

February 22, 2001

Web Services are the underpinning of Microsoft's .NET strategy. The concepts and the innovations behind this initiative have struck a chord with developer's building the next generation of Internet applications.

In this month's column, we're going to take a look at the features within ASP.NET to enable Web Services. Before we dig into the technical details let's start with an overview of Web Services.

**Web Services Overview**

A Web Service is programmable application logic accessible via standard Web protocols. One of these Web protocols is the Simple Object Access Protocol (SOAP). SOAP is a W3C submitted note (as of May 2000) that uses standards based technologies (XML for data description and HTTP for transport) to encode and transmit application data.

Consumers of a Web Service do not need to know anything about the platform, object model, or programming language used to implement the service; they only need to understand how to send and receive SOAP messages (HTTP and XML).

**Soap Message**

A SOAP message consists of several elements, most notably an envelope. The envelope encapsulates the data transmitted within the SOAP message. Below is a simple SOAP message complete with HTTP headers:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl01_code');" \o "Copy Code)

POST /demo/MSDN/PerfCounter.asmx HTTP/1.1

Connection: Keep-Alive

Content-Length: 150

Content-Type: text/xml

Host: localhost

User-Agent: MS Web Services Client Protocol 1.0.2204.19

SOAPAction: "http://tempuri.org/PerfCounters"

<?xml version="1.0"?>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"

xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<soap:Body>

<PerfCounters xmlns="http://tempuri.org/"/>

</soap:Body>

</soap:Envelope>

In the example above, we see the HTTP headers for the request, including the HTTP SOAPAction header, which is optionally used by the server for routing the SOAP message. Following the HTTP headers we find the body of the HTTP message. The body of the HTTP message is the SOAP request for a PerfCounters Web Service, which we are going to build.

Unfortunately we don't have nearly enough room in this column to discuss SOAP in depth. To learn more about SOAP, please see the [SOAP Developer Resources page](http://msdn.microsoft.com/library/?url=/nhp/default.asp?contentid=28000523). Here you can find the public specification for SOAP 1.1 as well as articles and other relevant resources.

**ASP.NET Web Services**

Web Services are simple and easy to understand. It is possible, in fact, to author a simple application that surfaces data as XML conforming to the SOAP specification. It would also be relatively straightforward to build an application capable of receiving SOAP messages over HTTP and deriving meaningful value out of it. For those of you familiar with PERL, this could simply be a matter of using RegEx to parse the value out of the XML result; it's just another string.

However, just as we use frameworks such as ASP and ASP.NET to build Web applications, we would much rather use a framework for building Web Services. The reasoning is quite logical. We don't need to reinvent the plumbing—that is, at a high level, the capability to serialize our data as XML, transport the data using HTTP, and de-serialize the XML back to meaningful data. Instead, we want a framework that makes building Web Services easy, allowing us to focus on the application logic not the plumbing. ASP.NET provides this framework for us.

From a developer's point of view, if you have ever written application logic, you have the required skills to author ASP.NET Web Services. More importantly, if you're at all familiar with ASP or ASP.NET application services, (application state memory, and so on) you can also leverage these skills when you build ASP.NET Web Services.

**Exposing**

For the purpose of example, we're going to write a Web Service that exposes Web application performance counters. Performance counters provide us with details about the behavior of our application, such as the number of active sessions or the number of requests served. We don't always have local server access to our Web server, and if we have a farm of servers we might want to expose the performance counters from all these servers and aggregate them in a central location.

**Starting with a Simple Example**

Rather than jumping straight into the Performance Counters example, let's start with some very simple application logic so we can see what we need to do to expose our logic as a Web Service. We'll use an **Add()** method that accepts two Integers and returns their sum. Below is this simple Visual Basic logic:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl03_code');" \o "Copy Code)

Public Class MyMath

Public Function Add(a As Integer, b As Integer) As Integer

Return a + b

End Function

End Class

We could use this class and its method as follows:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl04_code');" \o "Copy Code)

Dim mymath As new MyMath

Dim result As Integer

result = mymath.Add(10, 20)

To expose the above class, MyMath, as an ASP.NET Web Service we need to move the application logic into a \*.asmx file. Just as we use the extension \*.aspx for ASP.NET Pages, we use \*.asmx to tell ASP.NET that the file is an ASP.NET Web Service.

After we created the \*.asmx source file and add our application logic, we need to make a few more small changes:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl05_code');" \o "Copy Code)

<%@ WebService Language="VB" Class="MyMath" %>

Public Class MyMath

Public Function <WebMethod()>Add(a As Integer, b As Integer) As Integer

Return a + b

End Function

End Class

**Changes to our source**

The changes we've made to the \*.asmx file include adding a **WebService** directive that names both the Language as well as the Class we're exposing as a Web Service. The **WebService** directive is required, as we must tell ASP.NET the class that contains the application logic. Next, we've added a **<WebMethod()>** attribute to our Add() function declaration. An attribute is a declarative code element that lets us change the behavior of our application logic without necessarily writing more code. In the case of the **<WebMethod()>** attribute, this tells ASP.NET that the method with this attribute is to be treated as 'Web callable'. Web callable in the sense that ASP.NET does the necessary work for this method to support SOAP.

Now that we've seen what needs to be done to enable application logic as Web callable, let's look at a more relevant sample.

**Performance Counter Web Service**

Below is application logic that gives us access to the Windows® performance counters, with the changes for ASP.NET Web Services. The file we've created is PerfCounter.asmx:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl06_code');" \o "Copy Code)

<%@ WebService language="VB" class="PerfCounters" %>

Imports System.Xml.Serialization

Imports System.Web.Services

Imports System.Diagnostics

Public Class PerfCounters

  Inherits WebService

  ' Returns a Counter class

  Public Function <WebMethod()>GetCounters() As Counters

    Dim c As new Counters

    ' Application Name

    c.ApplicationName              = IISAppName

    ' System specific

    c.WorkerProcessRestarts = Poll(0, "Worker Process Restarts")

    c.WorkerProcessRunning  = Poll(0, "Worker Process Running")

    c.ApplicationsRunning   = Poll(0, "Applications Running")

    c.RequestsQueued        = Poll(0, "Requests Queued")

    ' Application Specific

    c.RequestsTotal         = Poll(1, "Requests Total")

    c.RequestsFailed        = Poll(1, "Requests Failed")

    c.RequestsSucceeded     = Poll(1, "Requests Succeeded")

    c.ActiveSessions        = Poll(1, "Sessions Active")

    Return c

  End Function

  Private Function Poll(counterType As Integer, counter As String) As Integer

     Dim PerfCounter As PerformanceCounter

     If (counterType = 0)

       PerfCounter = new PerformanceCounter("ASP Plus System", counter, "")

     Else

       PerfCounter = new PerformanceCounter("ASP Plus Applications", counter, IISAppName)

     End If

     Return PerfCounter.NextValue().ToInt32()

  End Function

  Private Function IISAppName() As String

    Dim AppName As String

    AppName = Context.Request.ServerVariables("APPL\_MD\_PATH")

    AppName = AppName.Replace("/"C, "\_"C)

    Return AppName

  End Function

End Class

Public Class Counters

  Public ApplicationName As String

  Public WorkerProcessRestarts As Integer

  Public WorkerProcessRunning As Integer

  Public ApplicationsRunning As Integer

  Public RequestsQueued As Integer

  Public RequestsTotal As Integer

  Public RequestsFailed As Integer

  Public RequestsSucceeded As Integer

  Public ActiveSessions As Integer

End Class

Again we see that we've declared a **WebService** directive at the top of the file noting both the language and the class. The class that contains the Web callable method is **PerfCounters**. Within **PerfCounters** we find a single method, **GetCounters()**, with the **<WebMethod()>** attribute. **GetCounters()** returns an instance of another class, **Counters**.

When we call **GetCounters()**, the method creates a new instance of the **Counter** class and begins to set its public members; note, these public members should be implemented as properties, but I chose to save the space for the purpose of the article.

When the **Counter** class' members are set, we're setting them with the returned result of a call to a private method **Poll()**. **Poll()** is responsible for doing the actual work of polling the systems performance counters and returning a result.

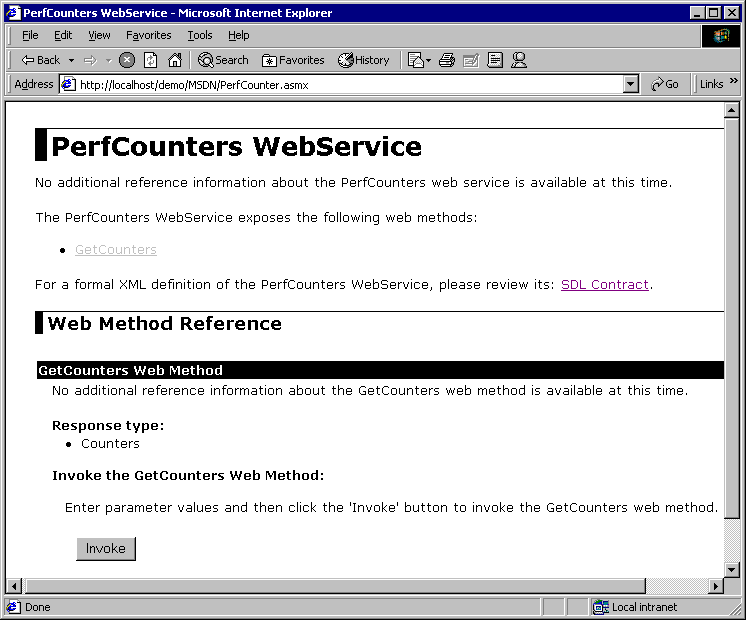
Finally, the last method, IISAppName(), returns the value of the server variable APPL\_MD\_PATH and replaces '/' characters with '\_' characters; this value is used as the application name within the performance counters.

Now that we've built the service, let's take a look at how we test it.

**Testing Web Services**

Now that we've authored this ASP.NET Web Service, how do we test it? The consumer of a Web Service is another application, but ASP.NET provides a simple browser interface to our Web Service that we can use for testing or documentation purposes.

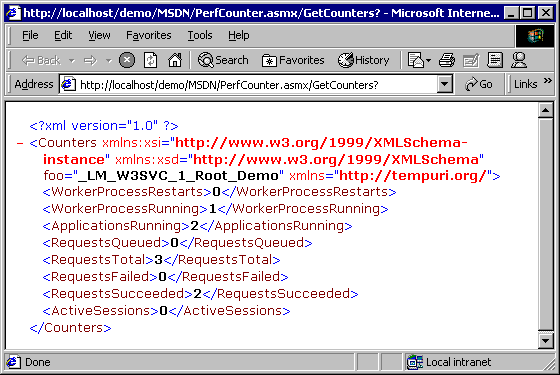
Since our service is exposed as a resource available from our Web server, we can simply open a browser and make a request for that resource. Doing so provides us with a nice HTML-based Web Service Help page that lets people learn about what our service provides:



**Figure 1. HTML-based Web Service Help page**

ASP.NET generates the above page for us, and we can use it to test our service (note the HTML Invoke button within the GetCounters Web Method section) and access the XML contract language used to describe what our service offers; we'll be coming back to the XML contract language momentarily.

If we press the Invoke button, a new browser window is opened, and a request is made to our service using HTTP-Get; one of the three supported protocols used by ASP.NET Web Services:



**Figure 2. Example of the new browser window that is created when pressing the Invoke button.**

The XML returned is a valid XML document that describes all of the settings we identified in our Counters class. However, it is not SOAP. SOAP is the default protocol that is used when we do application-to-application communication.

Although we didn't discuss it in this article, we can customize our help page quite extensively. This is done by making some changes to the ASP.NET configuration system, or modifying the DefaultSDLHelpGenerator.aspx. I would recommend not modifying the DefaultSDLHelpGenerator.aspx, as this is the template used for all our Web Services. Instead, make a copy of it and reference the copied version in the application's configuration that makes use of it.

Now that we've discussed authoring and testing our Web Service, let's make use of it.

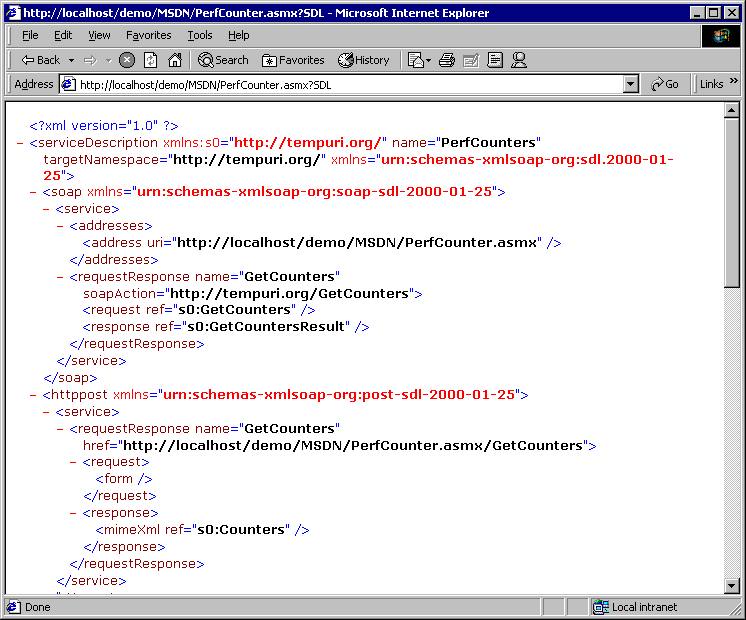
**Consuming**

We have several options for consuming Web Services. Since this article is about ASP.NET, we'll focus on .NET technologies that can consume Web Services. However, I should point out that any platform or framework that understands SOAP should be able to communicate with our Web Service. Building the Web Service with ASP.NET does not mean that the service is only available to other Microsoft applications.

Consumers of a Web Service need to know what the service offers—for example, what its Web callable method look like. Therefore, all Web Services optionally share another common XML document: a contract (note, Web Services built with ASP.NET always have a contract provided automatically).

**Contract**

In the examples above when we discussed testing a Web Service, we didn't discuss the link found within Web Service Help Page: SDL Contract. If we were to follow that link, instead of pressing the Invoke button for the **GetCounters()** Web Method, we would be presented with the following XML document:



**Figure 3. XML document presented when following the link found within the Web Service Help Page**

This XML document is a contract that describes our Web Service. It details the protocols supported as well as the semantics for calling and returning values. It additionally defines an XML schema for our **Counters** class.

Tools can use this XML schema to build proxy classes for our Web Service. A proxy class is a class that looks and feels like a local object, but it is in fact doing the work to serialize, send, receive, and de-serialize our method request to a SOAP endpoint.

**Note**Beta 1 of .NET surfaces an "SDL—Service Description Language" contract, Beta 2 will switch to use the more recent "WSDL—Web Service Description Language" contract. Semantically they are very different. WSDL is the collaborative work of Microsoft, IBM, and several other companies to better standardize the XML contract language.

We have various options for consuming Web Services, however, I'd like to call out three in particular:

* Visual Studio .NET: —Visual Studio .NET does the work of creating the proxy from the SDL or WSDL and adds the appropriate code to our project. This is done by simply selecting Project | Web References, and then pointing at a valid contract. Note that for beta 1 the contract must be SDL.
* Command Line Tools: —The .NET SDK ships with a tool called WebServiceUtil.exe that accepts an SDL contract and can generate the proxy source code for Visual Basic .NET, C#, or JScript.NET.
* IE 5.5. Behavior: —A browser specific behavior that allows for rich client interaction with SOAP end-points. For those of you familiar with Remote Scripting, you're going to love this! To learn more about the IE 5.5 behavior, please see [WebService Behavior](http://msdn.microsoft.com/en-us/library/ms531032.aspx).

Unfortunately, we don't have the space to discuss these three options in detail. However, I thought it would be worthwhile to briefly cover building a proxy with the command line tool, as this is applicable to those who have installed .NET; not just those that have Visual Studio .NET.

**Command line tool**

.NET, whether you install it as part of Visual Studio .NET or the .NET SDK, includes a command line proxy generation tool called WebServiceUtil.exe. The path to this command line tool, as well as several other command line tools, is added to our path when we installed .NET.

WebServiceUtil.exe allows us to name a SDL, or contract, as one of the command line arguments and the tool can then generate the source code for a proxy to our Web Service.

If, for example, we were to save the SDL from our PerfCounters.asmx example, we could use WebServiceUtil.exe to generate a Visual Basic .NET proxy to this Web Service:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl08_code');" \o "Copy Code)

WebServiceUtil.exe /command:proxy PerfCounter.sdl /language:VB

This generates a source file PerfCounters.vb that we now need to compile.

Using the VB.NET command line compiler, vbc.exe, we can compile our VB source file:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl09_code');" \o "Copy Code)

vbc /t:library /r:system.web.dll /r:system.web.services.dll /r:system.xml.serialization.dll perfcounters.vb

What we've done with the command line compiler is specify that we want to create a library (dll) rather than an executable (exe), and in addition to naming the source file to compile, we've specified some .NET assemblies (libraries containing classes our source file requires) as arguments to the compiler.

The result is PerfCounters.dll, a complete proxy to our PerfCounters.asmx ASP.NET Web Service that we can now use in .NET applications to communicate via SOAP to our Web Service.

Let's use this proxy to build a simple ASP.NET page that consumes and uses our Web Service.

**Using the Web Service**

First we need to deploy the compiled proxy, known as an assembly, to a Web application's \bin directory. Although we haven't discussed deploying compiled code in this column yet (yet another topic for a future column), suffice to say that to 'register' an assembly on the system simply requires copying the \*.dll to a Web application's \bin directory. This is a feature of .NET, but the use of the \bin directory is specific for ASP.NET.

To make things simple, we'll create a bin directory off of the server's root directory, c:\inetpub\wwwroot\bin for example. A \bin directory must exist in an application root, either the root of the Web or a folder marked as an application in IIS.

Next, we copy our assembly, PerfCounters.dll, to our \bin directory. We can now author our ASP.NET page, which we'll deploy to c:\inetpub\wwwroot. We'll call it PerfCountersConsume.aspx:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_code');" \o "Copy Code)

<Script runat="server">

Public Sub Page\_Load(sender As Object, e As EventArgs)

Dim perfcounters As New PerfCounters

Dim counters As Counters

counters = perfcounters.GetCounters()

webapp.InnerHtml = counters.ApplicationName

restarts.InnerHtml = counters.WorkerProcessRestarts.ToString()

procrunning.InnerHtml = counters.WorkerProcessRunning.ToString()

apprunning.InnerHtml = counters.ApplicationsRunning.ToString()

queued.InnerHtml = counters.RequestsQueued.ToString()

totalrequests.InnerHtml = counters.RequestsTotal.ToString()

failedrequests.InnerHtml = counters.RequestsFailed.ToString()

succeededrequests.InnerHtml = counters.RequestsSucceeded.ToString()

activesessions.InnerHtml = counters.ActiveSessions.ToString()

End Sub

</Script>

Web Application: <B id="webapp" runat="server"/><BR>

Process Restarts: <B id="restarts" runat="server"/><BR>

Processes Running: <B id="procrunning" runat="server"/><BR>

Applications Running: <B id="apprunning" runat="server"/><BR>

Requests Queued: <B id="queued" runat="server"/><BR>

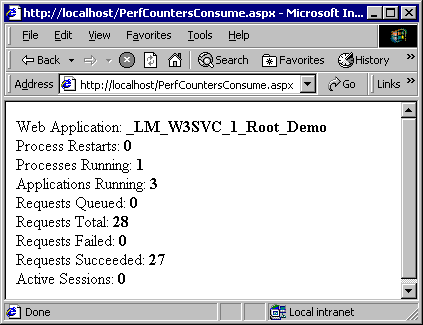
Requests Total: <B id="totalrequests" runat="server"/><BR>

Requests Failed: <B id="failedrequests" runat="server"/><BR>

Requests Succeeded: <B id="succeededrequests" runat="server"/><BR>

Active Sessions: <B id="activesessions" runat="server"/><BR>

The code above creates an instance of our proxy class PerfCounters (available to us since it's a registered assembly in our \bin directory) calls its **GetCounters()** method and returns an instance of a **Counters** class. We then use the instance of the **Counters** class, counters, to request its member variables and populate ASP.NET server controls. The result is below:



**Figure 4. ASP.NET server controls**

**Summary**

This column has taken a very high level overview of ASP.NET Web Services. There's quite a bit of detail that we either glossed over or didn't cover at all, for example security, use of session state, extensions, and so on. In next month's column we're going to look at a more advanced feature of ASP.NET Web Services, extensions, that we can use for building attributes that allow us to trace the request/response of our ASP.NET Web Service.

**Rob Howard** is a program manager for ASP.NET on the .NET Framework team. He spends whatever spare time he has either with his family or fly fishing in Eastern Washington.