Creating and Consuming Web Services

March 25, 2002
By: Gregg D. Harrington and Robert P. Lipschutz

Web Services signal a new era of lightweight distributed application development. While Web Services are not intended nor do they have the power to solve every distributed application problem, they are an easy way to create and consume services over the Internet. One of the design goals for Web Services is to allow companies and developers to share services with other companies in a simple way over the Internet.

"Web Services lower the barrier for small companies and make large companies more agile." says Mike Amundsen, President of EraServer.NET, a .NET hosting and XML Web Service provider.

And although interoperability issues are still being worked out, Web Services show strong indications of cross-platform, cross-language compatibility. In this article, we will explain how to create and consume Web Services using Microsoft Visual Studio .NET and the VB .NET language. Throughout this article, when we say Web Services, we'll typically mean Web Services created and consumed in the Microsoft environment. For our story, we used both Release Candidate 3 and the final version that was released on February 13th at the "VSLive!" event in San Francisco. This article will explain the benefits and drawbacks of Web Services, delve into the underlying technology, and provide practical how-to advice on using Web Services in the Microsoft environment. We tried to go deeper than other articles we've seen to provide answers to some of the difficult questions. In total, we think you will like using Web Services in the Visual Studio .NET IDE environment once you understand a few of its limitations. (Note that this is the first of a four-part series.)

The Good, the Bad, and ... Other Stuff to Know

Web Services make it easy to create applications where the consuming application sends up a simple values (e.g. city and state) to exposed methods on a server--the server does some processing, and sends back a simple value (e.g. zip code) to the client. Some applications that lend themselves nicely to Web Services include search, stock quotes, price comparison applications, travel planning, and other applications that lend themselves to request/response architectures.

Web Services have many advantages:

- Based on standard protocols such as SOAP, XML, and HTTP
- Cross-platform and cross-language support
- Simple distributed application development as compared to other techniques
- Type safety for datatypes

In addition, Web Services built and consumed in the Microsoft .NET environment have additional advantages:

- Excellent integration with Visual Studio .NET and Internet Information Server
- Ability to use the same development environment and framework when building client and server applications

If you are comfortable with some simple object oriented concepts like classes and methods, Web Services will be a breeze to use in developing many kinds of applications. If you want to use objects as input parameters or return values, quite common in object oriented programming, there are some limitations. Web Services are not so good at working with complex remote objects. If your goal is to create a complex object on the server side and have a client consume this object and all of its logic, you are in for some difficulties. You'd be better off designing your application differently or using a different approach. Here are some of the limitations we'll discuss:

- Cannot enforce business logic on the client
- Cannot access read-only properties on serializable objects
- Cannot use non-default constructors on serializable objects
- Cannot serialize datatypes such as Collection or HashTable
- Cannot enforce logic in property getters or setters on serializable objects

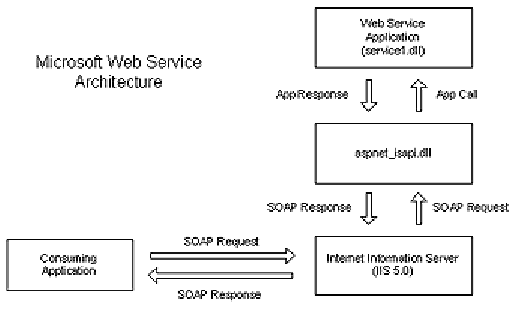
To be fair, Web Services were designed to be simple, with a simple message protocol that does not lend itself to handling complex objects. Unlike other distributed object oriented approaches such as RMI or CORBA, where you get the functionality of a remote class just as you would a local class, Web Services work differently and are more limiting in some respects. These limitations come from the simplicity of the Simple Object Access Protocol (SOAP), and change the way you might write your application as compared with other distributed application approaches. One interesting way to write Web Services applications is with XML formatted documents. If the goal is to return data, then instead of thinking about returning data objects, think about returning data as XML. As said by Yasser Shohoud, President of VBWS, developers using Web Services need to start thinking in terms of messages and XML documents rather than objects and methods." This approach requires a thorough understanding of XML, but any Web Services developer interested in interoperability will need a good understanding of HTTP, XML, SOAP, and WSDL to solve the difficult problems that sometimes arise. In this article, we took the traditional approach of using objects, and then tried to apply Web Services to our thinking. We got the job done and learned some valuable lessons.

Overall, we found Web Services in Visual Studio .NET very pleasing once we got by some of our pre-conceptions. Since VB had become an object-oriented language, we half expected Web Services to provide a powerful remote object framework where remote objects could be used as if they were local. We found instead a simpler, less powerful way of doing things, that in the end, we liked. We think the trade-offs are worth it for many kinds of applications.

What Exactly is a Web Service?

The term Web Services is used quite specifically and describes applications that can be easily consumed by other applications over the Internet using the SOAP specification. The idea is to provide a simple application-to-application interface just like the Web (with its HTML and HTTP specifications, and Web browsers and servers) has provided a simple human-to-application interface. Web Services will be appreciated more by developers and architects than by end-users, but the ultimate benefit will be seen by end-users that gain access to more and better services on the Web.

Let's look at a technical description of Web Services to get on some firm ground. Web Services are applications that expose some or all of their methods to other consuming applications in a well-known standard way. A consuming application can send request messages using SOAP to the exposed methods of a Web Service and expect a SOAP response message back. Each Web Service has what is known as Web Services Description Language (WSDL) that defines the kinds of messages it accepts along with the associated response messages.


Microsoft Web Service Architecture
Web Service Application (service1.dll)
App Response
App Call
aspnet_isapi.dll
SOAP Response
SOAP Request
Consuming Application
SOAP Request
SOAP Response
Internet Information Server (IIS 5.0)

click on image for full view

The SOAP messages are handled first by IIS and then by the ASPNET_ISAPI.DLL.

For example, a weather Web Service might accept a "GetTemperature" message with a parameter of "zip code" and return a "GetTemperatureResponse" message with an integer that indicates the temperature. The VBWS site has a number of simple weather Web Services examples.

The good news for developers is that Visual Studio .NET (and other development tools) hide SOAP and WSDL under the covers making it very easy for developers to build and consume Web Services. The developer works with things developers are used to seeing - classes, methods, and properties. On the consuming side, the developer creates what is called a Web Reference to the Web Service. All of the methods and data exposed by the Web Service, as defined by the WSDL file, show up appearing to be local classes and methods in Visual Studio .NET. A Web Reference works similar to a Visual Studio .NET Reference. For example, when you add a reference (Right Click on Project | Add Reference) to the ADO database component in Visual Studio, you have access to all of the ADO classes and methods. When you add a Web Reference to a Web Service, you have access to all of its exposed classes and methods, and Visual Studio .NET's auto-completion feature (called Intellisense) works just as it does with local classes.

Although we use Microsoft's Visual Basic .NET in this article, Web Services can be created or consumed in any programming language with support for SOAP and WSDL. Other vendors such as IBM, Oracle, Sun, BEA, and Borland to name a few, have Web Services products available or initiatives underway.

Let's talk a little more about the standards, SOAP and WSDL, both of which are handled by the World Wide Web Consortium.

Applications use SOAP to communicate with Web Services. SOAP defines a simple, human-readable message format based on XML, and in the SOAP 1.1 specification, it states a simple binding to HTTP. SOAP is simple to get your hands around. It only takes a few hours to read and understand the specification. The underlying HTTP and XML specifications offer clear advantages. HTTP travels across firewalls, is understood by Web servers, and provides a very simple request/response mechanism--a perfect lightweight distributed application protocol for the Internet. XML provides a clear way to organize information in a message. Although XML is not the most efficient way to send and process data, it is a good choice for interoperability. You can check out the SOAP Version 1.1 W3C Note and the latest version of SOAP Version 1.2 Part 1: Messaging Framework Working Draft and SOAP Version 1.2 Part 2: Adjuncts on the W3C site. Visual Studio .NET supports SOAP version 1.1.

WSDL is an XML format that describes the methods and data available in a Web Service. The WSDL interface provides both human-created descriptions and XML syntax describing the Web Service and its methods. When a developer adds a Web Reference to a Web Service, Visual Studio .NET uses the WSDL file to generate a proxy class for a Web Service, and this proxy class looks just like a local class when exposed in the IDE,. To read more about the WSDL specification, check out the W3C note, Web Services Description Language (WSDL) 1.1.

Comparison to Local Classes and Objects

Perhaps the best way to understand the capabilities and limitations of Web Services is to compare them to other techniques. First, let's compare the consumption of a Web Service to the consumption of a local object. Later, we'll compare Web Services to other distributed application techniques.

Let's look at an example of a simple class called Plane that has private members, public read-write and read-only properties, default and non-default constructors, and a public function.

```
Public Class Plane
    'private members
    Private mName As String
    Private mPassengers As Integer
    Private mRange As Integer
    Private mEngines As Integer
    Private mTailNumber As String

    'read-write properties
    Public Property Name() As String
       Get
          Return mName
       End Get
       Set(ByVal Value As String)
          mName = Value
       End Set
    End Property

    Public Property Passengers() As Integer
       Get
          Return mPassengers
       End Get
       Set(ByVal Value As Integer)
          If Value < 100 Then
             mPassengers = Value
          End If
       End Set
    End Property

    Public Property Engines() As Integer
       Get
          Return mEngines
       End Get
       Set(ByVal Value As Integer)
          mEngines = Value
       End Set
    End Property

    'read-only property
    Public ReadOnly Property PlaneTailNumber()
       Get
          Return mTailNumber
       End Get
    End Property

    'default constructor
    Public Sub New()

    End Sub

    'non-default constructor, takes input parameters
    Public Sub New(ByVal TailNumber As String, ByVal Passengers As Integer, ByVal Range As Integer)
       mName = "Bonanza Man"
       mRange = Range
       mPassengers = Passengers
       mEngines = 1
       mTailNumber = TailNumber
    End Sub

    'public function
    Public Shared Function CalcRange(ByVal Fuel As Integer) As Integer
       'Range increase by 10 miles for each gallon of fuel
       Dim Range As Integer

       Range = Fuel * 10

       Return Range
    End Function
 End Class
```

A piece of code local to this class could access the read-only and read-write properties, both constructors, and the public function. Of course, the private member variables are private to this class and cannot be accessed outside of it.

To use the class, a consuming application would dimension a variable of type Plane and then create a new instance of it; in our example below, we use the parameterized constructor and then display the plane's tail number.

```
 Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
       Dim aPlane As Plane
       aPlane = New Plane("N84DL", 4, 200)

       MessageBox.Show(aPlane.PlaneTailNumber)
    End Sub
```

Converting Our Class to a Web Service

Now, let's compare what we can do local to what a Web Service can do. First off, let's take the simple route and turn this class into a Web Service. We'll soon see why this simple route is not the best route. We create a Web Service project, copy our Plane class into the service1.asmx.vb code file, and make some minor syntax adjustments. In summary, we've added an import of the System.Web.Services namespace, an inherit statement for System.Web.Services, a <WebService()> attribute in front of the class, and a <WebMethod()> attribute just before the public function.

```
 Imports System.Web.Services

 <WebService(Namespace := "http://tempuri.org/")> _
 Public Class Plane
```

```
    Inherits System.Web.Services.WebService

{omitted implementation details}

    'public function
    <WebMethod()> Public Function CalcRange(ByVal fuel As Integer) As Integer
        'Range increase by 10 miles for each gallon of fuel
        Dim range As Integer
        Return range = fuel * 10

    End Function

End Class
```

If we consume this Web Service, all we will have available to us is the single public function exposed by the <WebMethod()> attribute. We've lost access to the properties and the constructors. We can't get at the properties because there is no WebMethod that returns them. So how do we get them back? Well, it turns out that we can get back the read-write property, but not the read-only property.

Some of you may already be guessing the answer. You ask why we don't just make the class serializable which should give us access to the properties, right? Well, it turns out that a class can't be both a Web Service and serializable. But what we can do is split up our plane into two parts, the Web Service part with WebMethods, and a separate serializable class with properties. Then, we can have a WebMethod e.g. "GetPlaneInfo(planeTailNumber) As PlaneDetails" that returns the serializable object we'll call "PlaneDetails" with the plane's properties.

As we split our code into two classes, we could have put the "CalcRange" function implementation details in either class, using the WebMethod itself, or we can have the WebMethod call the implementation from the other class. It's really just a matter of preference. We decided to keep the implementation in PlaneDetails so we simply call this implementation from the WebMethod.

Here's our code.

```
Imports System.Web.Services

<WebService(Namespace:="http://tempuri.org/")> _
Public Class Plane
    Inherits System.Web.Services.WebService

Web Services Designer Generated Code

    <WebMethod()> Public Function CalcRange(ByVal fuel As Integer) As Integer
        'Range increase by 10 miles for each gallon of fuel

        Return PlaneWebService.PlaneDetails.CalcRangeImpl(fuel)
    End Function

    <WebMethod()> Public Function GetPlaneInfo(ByVal planeTailNumber) As PlaneDetails

        'Returns the PlaneDetails serializable object
        Return New PlaneDetails(planeTailNumber, 4, 250)

    End Function

End Class

<Serializable()> Public Class PlaneDetails

    'private members
    Private mName As String
    Private mPassengers As Integer
    Private mRange As Integer
    Private mEngines As Integer
    Private mTailNumber As String

    'read-write property
    Public Property Name() As String
        Get
            Return mName
        End Get
        Set(ByVal Value As String)
            mName = Value
        End Set
    End Property

    Public Property Passengers() As Integer
        Get
            Return mPassengers
        End Get
        Set(ByVal Value As Integer)
            If Value < 100 Then
                mPassengers = Value
            End If
        End Set
    End Property

    Public Property Engines() As Integer
        Get
            Return mEngines
        End Get
        Set(ByVal Value As Integer)
            mEngines = Value
        End Set
    End Property

    'read-only property
    Public ReadOnly Property PlaneTailNumber()
        Get
            Return mTailNumber
        End Get
    End Property

    'default constructor
    Public Sub New()

    End Sub

    'non-default constructor, takes input parameters
    Public Sub New(ByVal planeTailNumber As String, ByVal passengers As Integer, ByVal range As Integer)
        mPassengers = passengers
        mRange = range
        mName = "Bonanza Man"
        mEngines = 1
    End Sub

    'public function
    Public Shared Function CalcRangeImpl(ByVal fuel As Integer) As Integer
        'Range increase by 10 miles for each gallon of fuel
        Dim Range As Integer

        Range = fuel * 10

        Return Range
    End Function

End Class
```

Notice that we now have a Web Service class called Plane that has two WebMethods, and a Serializable class PlaneDetails that includes all of the member variables, properties, constructors, and "CalcRangeImpl" method implementation. So now, a consuming application would have access to the properties in Plane Details.

Our Web Service In Action

Let's see how our Web Service code works. We created a simple Windows application, gave it a Web Reference to our Web Service and tried to expose the plane details:

```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click

        Dim SrvPlane As New localhost.Plane()
        Dim APlane As localhost.PlaneDetails

        APlane = SrvPlane.GetPlaneInfo("N84DL")

        MessageBox.Show(APlane.Name)
        MessageBox.Show(APlane.Passengers)
        MessageBox.Show(APlane.Engines)

    End Sub
```

We find that we could not show the TailNumber property--it simply didn't show up on the client side because it is a read-only property. Read-only properties are not exposed to consuming clients. Again, this is a limitation of the SOAP protocol underlying Web Services. There are many cases where you would want to send down a read-only property to a consuming application. In our example, we want to show the TailNumber to identify the plane. A workaround is to make the TailNumber read-write and ensure that it is not changed by using server logic for enforcement. It's just a little gotcha in Web Services.

Another related limitation is that you can't enforce code in property getters and setters. For example, if you wanted to ensure that the Engines property equaled 1 or 2 (we have small airplanes in our application), you would normally place this logic in the Engine property setter. Since no code can be sent down to the client, this enforcement is ignored. You'll have to enforce it on the server with some server-side validation code written separately in a method.

.NET Remoting and Comparisons to Other Distributed App Techniques

Web Services does not provide enough power for applications where you wish to consume complex objects and enforce business logic on the client. In these cases, consider Microsoft's remoting feature. Remoting is another form of distributed programming that Microsoft has introduced in the .NET framework. Unlike web services, remoting is built on a proprietary binary Microsoft protocol. While remoting is not interoperable with other platforms and languages, in most cases it offers more functionality and performance than web services. For more information on remoting, check out the MSDN site and specifically this excellent article, "Remoting and XML Web Services in Visual Basic .NET".

Web Services come easier than other distributed application development techniques. Distributed application development has mostly been used inside companies for Enterprise Application Integration (EAI) and has historically been the domain of uber-programmers hidden deep in the recesses of big Fortune 500 companies. These hotshots use remote object technologies like Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) or message passing products like IBM MQ Series. These methods suffer from complexity in both development and deployment and require difficult learning curves. In contrast, Web Services opens up the world of distributed programming to the rest of us.

Kollen Glynn, Vice President of Product Development at EarthConnect, a company building Web Services in the financial services space, sums it up nicely, "DCOM and CORBA were painful to use for distributed applications. All of a sudden, it's fun to program again."

Web Services are not for every need. Web Services are designed to be simple and in the process give up some power inherent in other remote object technologies. If you want to do true distributed object oriented programming, with support for distributed read-only properties, serialization of complex types, etc., you'll need to use technologies like RMI or CORBA.

That ends our introductory story on building Web Services. In the next segment, we'll get into much more detail about Web Services from the developer's standpoint. And in the third and forth segments, we'll get even more hands-on with Web Service development and deployment.

Creating and Consuming Web Services

April 2, 2002
By: Gregg D. Harrington and Robert P. Lipschutz

A Simple Web Service from a Developer's Viewpoint

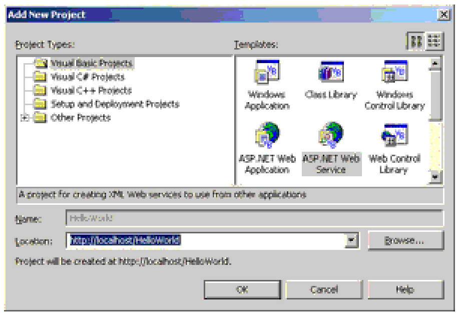
In Part I we introduced you to the general concepts of Web Services and built a simple Web Service. For this segment, we plan to dive deeper into a Microsoft .NET application that makes extensive use of Web Services. It's a simple Price Comparison application with a Windows client, Web application, and three Web Services. We'll start out using a simple "Hello World" application that Microsoft provides to show you the basics of Web Services. Then we'll move over to some home-grown code, and finally to our Price Comparison application. You'll see that we uncover some important discoveries as we dive deeper into development.

You can follow along with us here, or if you'd like, or you can install our project on your own development machine. (You should follow the installation instructions included in the Install.txt file.)

So let's get started with our "Hello World" application.

Visual Studio organizes code using solutions and projects. Typically, an entire application will be housed in a single solution. Let's open Visual Studio .NET and get started.

1. On the **File** menu, click **New | Blank Solution**
2. Name the solution "HelloWorld"
3. In **Solution Explorer**, right-click on the solution name, point to **Add** and then click **New Project**
4. From the Visual Basic Projects folder, click on ASP.NET Web Service and name the project "HelloWorld"
5. Click **OK**



A project for creating XML Web services to use from other applications

Name: HelloWorld

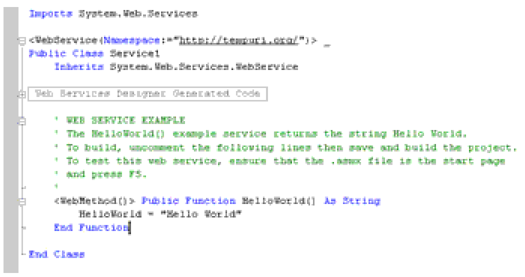
Location: http://localhost/HelloWorld

Project will be created at http://localhost/HelloWorld.

click on image for full view

6. Double-click on the Service1.asmx project and then click on "click here to switch to code view" in the design window.

   The default Web Service file, **Service1.asmx** contains a commented-out HelloWorld <WebMethod>.
7. Uncomment the HelloWorld <WebMethod> as shown below.



```
Imports System.Web.Services

<WebService(Namespace:="http://tempuri.org/")> _
Public Class Service1
    Inherits System.Web.Services.WebService

 Web Services Designer Generated Code

    ' WEB SERVICE EXAMPLE
    ' The HelloWorld() example service returns the string Hello World.
    ' To build, uncomment the following lines then save and build the project.
    ' To test this web service, ensure that the .asmx file is the start page
    ' and press F5.
    '
    <WebMethod()> Public Function HelloWorld() As String
        HelloWorld = "Hello World"
    End Function

End Class
```

click on image for full view

As you know, a typical class will contain members, properties, and methods. From a code perspective, a Web Service is similar to a typical class. One difference is seen in the <WebService()> attribute that prefaces the class name…

```
<WebService(Namespace:="http://tempuri.org/")> _
Public Class Service1
```

Another difference is the <WebMethod> attribute that prefaces any method that will be exposed as a method in the Web Service, in our case the HelloWorld function.

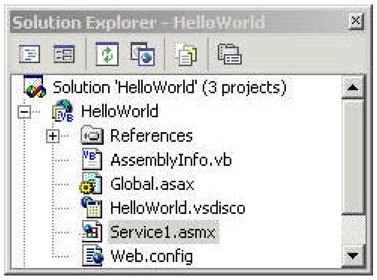
```
<WebMethod()> Public Function HelloWorld() As String
```

A few other things happen when a Web Service is created. Visual Studio .NET creates the helper files required for a Web Service and creates a virtual root folder within Internet Information Server (IIS). You can view the helper files via the Visual Studio.NET Solution Explorer, your Windows file system, and through the Internet Information Services application (Control Panel | Administrative Tools). Let's view the files in the Solution Explorer.

Using the Visual Studio .NET Solution Explorer

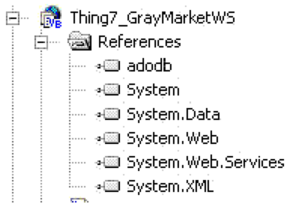
In the solution explorer, we see that a number of primary files have been created for us.

- References folder
- AssemblyInfo.vb
- Global.asax
- HelloWorld.vsdisco
- Service1.asmx
- Web.Config



Visual Studio.NET creates a number of files needed for a Web Service.

References Folder
This is where references to the Common Language Runtime, COM object, and ActiveX objects will appear. You can add to these by right clicking on the project and going to "Add Reference…" For example, we added a reference to the ADODB library for our Web Services for database connectivity.



References provide access to CLR classes such as System and System.Data.

AssemblyInfo.vb
This XML formatted file is dynamically generated when you create a project and includes information about the company, product, and product version.

Global.asax
This file is similar to the Global.asa file used in previous versions of Active Server Pages (ASP). Global.asax executes code based on application-level events that occur, such as when the application starts or a new session begins. For example, we used the "Application_Start" section to initialize the database connection in our Price Comparison application.

```
Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
    ' Fires when the application is started
    ModDatabase.Init("c:\inetpub\OneClickHosting\PDC115\fileUpload\GrayMarket.mdb")
End Sub
```

[Web Service Project Name].vsdisco
This is a Microsoft specific file, i.e. there are no standards around its use in Web Services. A VSDISCO file allows others to find and consume the Web Service(s). A VSDISCO file can perform an iterative search through a hierarchy of folders on a development Web server and point out any ASMX, DISCO, or other VSDISCO files contained therein.

[Web Service Name].asmx
This is the actual Web Service file.

Web.config
This XML-formatted file has many uses ranging from security configuration, to custom application parameters such as a

pointer to the database location. For example, here is the section on authentication.

```
<!--  AUTHENTICATION
     This section sets the authentication policies of the application.
                        Possible modes are "Windows", "Forms", "Passport" and "None"
-->
<authentication mode="Windows" />
```

The Authentication section of the web.config file.

You may also have noticed that our Web Service inherits from the class `System.Web.Services.WebService`.

```
Public Class Service1
    Inherits System.Web.Services.WebService
```

This class provides access to a number of objects. Two interesting ones that come in handy are the session and application objects.

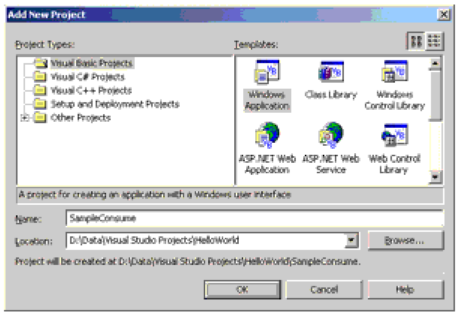
Session
The session object is used to store session information if sessions are enabled by the WebMethod attribute. The session object contains an items collection for storing session variables. The session object maintains state information for individual user connections across multiple HTTP page requests.

Application
The application object is very similar to the Session object in the sense that it stores variables in an items collection. However, it does so for the entire application. The Application object is useful for storing variables whose values are needed by all users.

That's it. We've created a simple Web service. Now, let's consume it from a simple Windows application.

Consuming a Web Service from a Developer's Viewpoint

1. In **Solution Explorer**, right-click on the "HelloWorld" solution name, point to **Add** and then click **New Project**
2. From the **Visual Basic Projects** folder, click on **Windows Application** and name the project "SampleConsume"



A project for creating an application with a Windows user interface

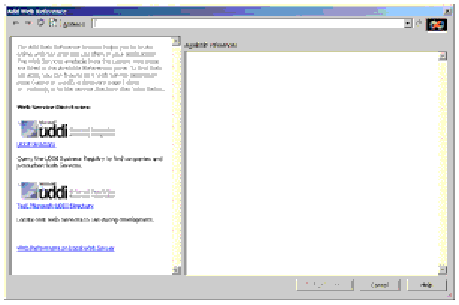
Name: SampleConsume

Location: D:\Data\Visual Studio Projects\HelloWorld

Project will be created at D:\Data\Visual Studio Projects\HelloWorld\SampleConsume.

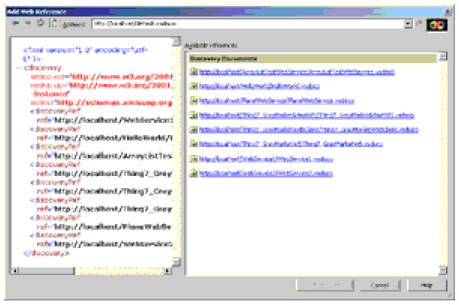
click on image for full view

3. Right-click on the **SampleConsume** project and click **Set As StartUp Project**
   Let's add a Web Reference to our SampleConsume project.
4. Right-click on **SampleConsume** and click **Add Web Reference** to bring up the Add Web Reference Dialog box. The **Add Web Reference** dialog lets you add Web References from any URL location, Microsoft's UDDI directory, Microsoft's Test UDDI Directory, and from the local Web Server on your development box. Since our Web Service is on our development box, we'll choose the local server option.



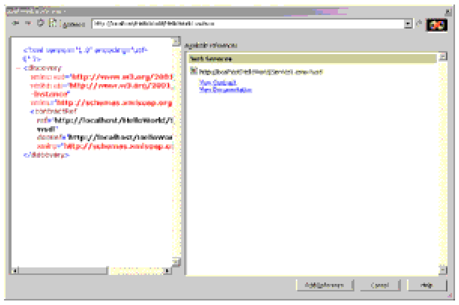
click on image for full view

5. Click on "Web References on Local Web Server".



click on image for full view

6. Click on "http://localhost/HelloWorld/HelloWorld.vsdisco"

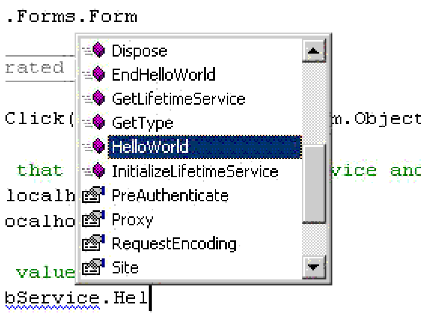


click on image for full view

7. Click on the **Add Reference** button.
   Now that we have a Web Reference to the HelloWorld Web Service, let's create a button and write some simple code to access the HelloWorld Web Service.
8. Double-click on **Form1.vb** to bring up the form designer
9. In the Windows Forms Toolbox, double-click on the **Button** control to create a new button
10. With the button highlighted, name the button **"btnConsume"** in the **Properties** window
11. Double-click on the button to bring up the code behind the btnConsume_Click event
12. Create the following code …

```
Public Class Form1
   Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code " (collapsed for readability)

   Private Sub btnConsume_Click(ByVal sender As System.Object,
            ByVal e As System.EventArgs) Handles btnConsume.Click

      'dimension a new a variable that points to our Web Service
      Dim aWebService As localhost.Service1
      aWebService = New localhost.Service1()

      'Display the return value of our Web Service
      MessageBox.Show(aWebService.HelloWorld())

   End Sub
End Class
```

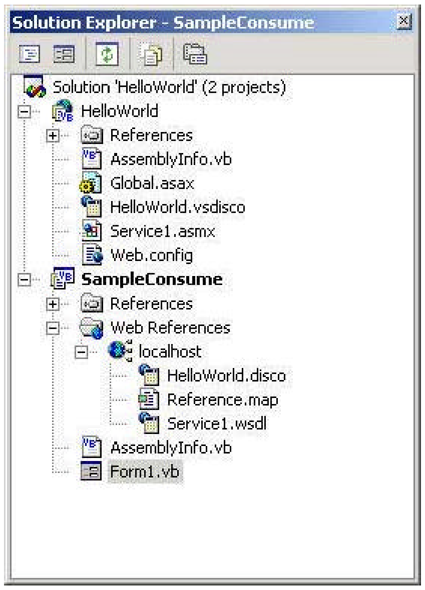
Because we had a Web Reference to the HelloWorld Web Service, we were able to use it as if it was a local class. In this case, it was actually on our local server but it could have just as easily been on a different server. As shown below, the Visual Studio .NET environment provides the same automatic code completion, called Intellisense, as is given to local classes. So after a few simple steps, you are using a remote Web Service as if it was a local class file!



click on image for full view

You may have noticed that when we added a Web Reference, a number of files are added to our SampleConsume project:

- HelloWorld.disco
- Reference.map
- Service1.wsdl



Solution Explorer shows a number of new files.

[Web Service Project Name].disco
Nope, the Bee Gee's have not become Microsoft employees. It's just an unfortunate acronym! A DISCO file, a Microsoft concoction, is an XML pointer file to one or more asmx files. These DISCO files are used to find and consume web services in Visual Studio .NET. However, DISCO is not a W3C standard like WSDL and SOAP.

Reference.map
This file has two parts. The first part is the actual reference.map file that contains XML pointers to each WSDL file. This file is used with another tool called the MSDiscoCodeGenerator to create the Reference.vb file that contains proxy code for the consuming side application.

[Web Service Name].wsdl Known phonetically as the "wis-dle" file, a WSDL (Web Services Description Language) file is an XML formatted document that describes how to consume a Web Service. This file contains such information as methods that are available, parameters for those methods, complex data types, etc.

The Web Reference, Proxy Code, and Method Calls

The following discussion describes what happens when a Web Reference is created and how subsequent method calls are processed.

Adding a Web Reference creates proxy code derived from the contents of the Web Services WSDL file. Visual Studio .NET runs the WSDL.exe on the WSDL file to create this proxy code. The proxy code has enough information in it to provide Visual Studio .NET with such things as Intellisense and object browsing. From a developer's perspective, the Web Service methods appear to be local via this proxy code.

When a method is called by the client application, the actual call is made through the proxy code. This proxy code then changes the request into a properly formatted SOAP message and sends the HTTP request to the Web Service host on default port 80, or port 443 for an SSL-protected transmission. The request is accepted on IIS by the aspnet_isapi.dll and then appropriately dealt with by the code, which will then formulate a proper SOAP response message. The HTTP SOAP response message is then sent to the consuming application.

For both function and subroutine calls, the Web Service returns a SOAP response message This might seem strange knowing that a subroutine typically gives no return value. But the SOAP implementation is tied to HTTP, which being a synchronous protocol, requires a response. The consuming application will wait for a SOAP response before it moves on in its code. Let's take a look at a SOAP message…

Inside a SOAP Request

Below is an example of a simple SOAP request. This request calls the method HelloWorld in the namespace http://tempuri.org/ (the default). As you can see, the SOAP protocol for this example has two basic parts--the envelope and the body (SOAP also defines an optional header which we will not discuss here). The envelope portion must be present in all SOAP messages. The envelope is the top-level element used for parsing the XML SOAP message and includes a link to the schema and encoding specification used by the message--in this case from the SOAP v1.1 specification on the W3C site. The body portion which starts with the <soap:Body> tag determines what the SOAP message was meant to do. In this case, it specifies the HelloWorld tag which we assume (and it does) points to a HelloWorld function on the recipient server.

```
POST /HelloWorld/Service1.asmx HTTP/1.1
Host: localhost
```

```
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/HelloWorld"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <HelloWorld xmlns="http://tempuri.org/" />
  </soap:Body>
</soap:Envelope>
```

Next, let's show a simple SOAP response. It's similar to the SOAP request, but in this case the body has the response data. The body contains the return values for the function called in the request. In this case, it returns the text "Hello World."

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <HelloWorldResponse xmlns="http://tempuri.org/">
      <HelloWorldResult>Hello World</HelloWorldResult>
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

Finding and Interrogating Web Services

If other companies will consume your Web Service, you might ask the question, how will they find my Web Service? One of the initiatives around Web Services is called Universal Description Discovery and Integration (UDDI). You can publish various information about your Web Services within UDDI. The information includes business name, Web Service description, business and industry classifications (such the North American Industry Classification System), and a pointer to the Web Service itself.

Microsoft has 2 UDDI sites, uddi.microsoft.com and http://test.uddi.microsoft.com. The test site is just that, a site where you can figure out how UDDI works and post your information. Once you're pleased with your choices, you can post it on the real site - http://uddi.microsoft.com.

UDDI is still a work in progress. We posted our application to UDDI but only after struggling to understand the meaning of all fields. We suggest you check out the entry for EarthConnect and its Galapogos Web Service to see a well-done entry.

- Browse to http://uddi.microsoft.com
- Search for EarthConnect

click on image for full view

Finding Web Services
Web Services might be on your local development machine, on a server down the hall, or even on a server across the Internet. It really doesn't matter from a coding perspective, but let's shed a little light on how we might find these Web Services and interrogate them to see what they do.

There are really just two parts to this:

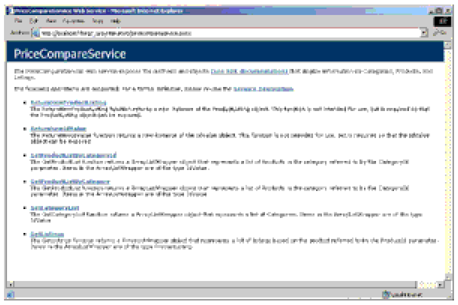
- Finding a Web Service
- Interrogating it to see what it does

Finding a Web Service can be a challenge. As we've said earlier, many companies will post Web Services listing in a UDDI directory such as http://uddi.microsoft.com. This will lead to the Web Service location and most likely a WSDL file. But with UDDI in an early stage of maturity, it can be difficult to find the Web Service you are looking for if it exists. You can also try other sites that are keeping catalogs of Web Services e.g. http://www.xmethods.com. You might have a prior business relationship where someone might email or point you to a Web Service link. In any case, once you have a link to the Web Service or information about the Web Service itself, you can find out a great deal about the service through a simple browser interface.

Obtaining Web Service Information

Let's look at one of the Web Services in our Price Comparison application--called PriceCompareService.

From the Add Web Reference screen or directly from a browser, you can point to any Web Service file--a file with an .asmx extension when created with Visual Studio .NET. Pointing to the file shows the documentation. Below we show the documentation for our PriceCompareService. Notice the URL in the address field. The documentation includes a description of our Web Service and a description for each of the WebMethods.



PriceCompareService

click on image for full view

The documentation is generated from the "Description:=" property placed within <WebService()> and <WebMethods()> attributes. Here is the sample code that generates these descriptions.

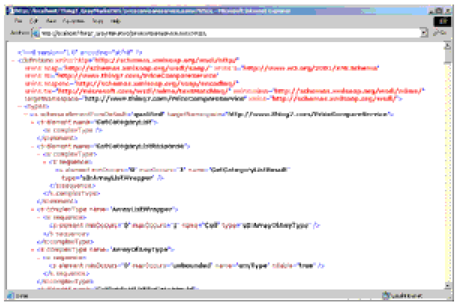
```
<WebService(Namespace:="http://www.thing7.com/PriceCompareService", _
    Description:="The PriceCompareService web service exposes the methods " & _
    "and objects <a href=""http://www.thing7.com/"">
                (see SDK documentation) </a>that display information on Categories,
                Products, and Listings.")> _
Public Class PriceCompareService



    <WebMethod(Description:="The ReturnNewProductListing function returns " & _
        "a new instance of the ProductListing object. This function is " & _
        "not intended for use, but is required so that the ProductListing " & _
        "object can be exposed.")> _
    Public Function ReturnNewProductListing() As ProductListing
```

For simple Web Services, the description fields may provide enough information so the Web Service may be consumed. For many Web Services, however, additional documentation will be required to detail a WebMethod's input parameters, return values and general workings and patterns to the application. Kollen Glynn of EarthConnect.says, "In all but the simplest cases, you need documentation in addition to the WebService and WebMethod descriptions."

In these cases, you can either include a link in the Web Service description as we have done using simple HTML tags or we can override the HTML description entirely and point to a custom Web Page by changing the docRef field on the contractRef node in the associated disco file.

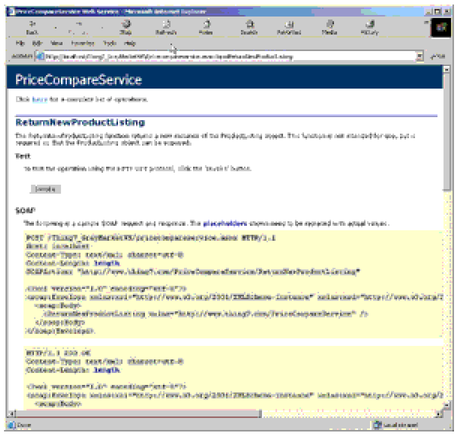
```
<contractRef ref="http://localhost/thing7_GrayMarketWS/AuthService.asmx?wsdl"
docRef="http://localhost/thing7_GrayMarketWS/AuthService.asmx"
xmlns="http://schemas.xmlsoap.org/disco/scl/" />
```

If you desire a lower-level look at the SOAP interface for a Web Service, you can view the WSDL information by tacking on a "?WSDL" at the end of the URL as shown below.

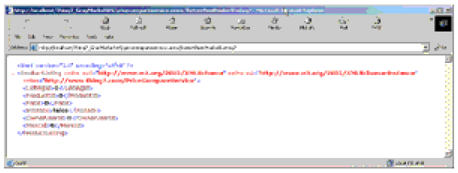


click on image for full view

By clicking on any of the WebMethods in the page, you can drill down to reveal the SOAP interface for the method along with the parameters it accepts. The method can then be tested with some input values and the response seen. For example, below we've clicked on the ReturnNewProductListing method and then on the Invoke button.



click on image for full view



click on image for full view

That wraps up Part II of our tutorial, and in the third and forth segments, we'll get even more hands-on with Web Service development and deployment.

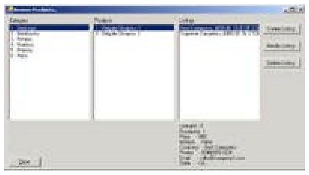
Creating and Consuming Web Services

April 25, 2002
By: Gregg D. Harrington and Robert P. Lipschutz

More Hands-on Application Development

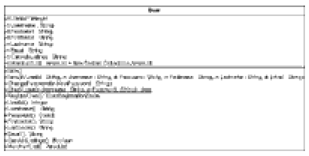
We pick up from the end of Part II, where we discussed creating, finding, and interrogating Web services. In this segment, we'll begin by walking through a real Web service development process to understand some of the nuances. We built a Price Comparison application using two Web service projects, a Web application (using Web Forms), and a Windows application. We expected to run into real problems if we built a real application. Admittedly, our application is not full-blown, but it did have enough complexity to show us many of the ins and outs of creating Web Services. Similar to what we indicated in Part II, you can install our project on your own development machine. (You should follow the installation instructions included in the Install.txt file.)

The Price Comparison user interface shown below illustrates the Windows user interface for the main part of our application.



click on image for full view

OO programmers will usually tackle a problem by designing an object with the right methods and properties to handle the problem. Once the methods and properties are known, the programmer can move on to their implementation. So, not surprisingly, we started with a "User" object. We gave it properties and methods as shown below in Visio UML static structure diagram



click on image for full view

Recall our Plane and PlaneDetails example from Part I, where we needed to separate our WebService from our serializable object. Since User is a serializable class for us, and therefore can't be a Web Service, our WebMethods must be placed in AuthService, our valid Web Service. So let's talk first about the AuthService Web Service.

Namespaces and Web Methods

We've already discussed how we create a Web Service, but let's bring in one more wrinkle here. Each Web Service needs a unique namespace as given by the namespace property within the WebService attribute.

```
<WebService(Namespace:="http://www.thing7.com/AuthService", _
      Description:="The AuthService web service provides " & _
      "the ability to manipulate and access information for " & _
      "the uses of the GrayMarket application.")> _
Public Class AuthService
   Inherits System.Web.Services.WebService
```

The Namespace property has a default value of "http://tempuri.org" and needs to be changed. This will not work with multiple Web Services. The result is that only one of the web services will show up. The suggested naming standard for Web Services uses your own unique URL, e.g. "http://www.thing7.com" with the Web Service name appended-- e.g. Namespace:="http://www.thing7.com/AuthService"

WebMethods
Next we created a WebMethod called CheckUser in the AuthService class, a different method from the CheckUser function in our User object.

```
   <WebMethod(Description:="{description omitted for readability")> _
   Public Function CheckUser(ByVal Username As String, ByVal Password As String) _
         As User
      Return Thing7_GrayMarketWS.User.CheckUser(Username, Password)
   End Function
```

The WebMethod CheckUser in AuthService actually calls our CheckUser function in User. We've chosen to place all of the implementation in the User class where we check the database for a user. Keep this pattern in mind for your own projects. We won't show you the code here, as it is quite verbose database connection logic.

Serializing Objects

Serialization is the process of converting an object or other data type into a form that can be sent across a network (i.e. a data stream). Web Services handle serializing simple data types such as integers, strings, dates, etc. quite well. But you explicitly have to make objects serializable. There's a very nice article on serialization by Rockford Lhotka at Magenic Technologies on the MSDN site called "Object Serialization in Visual Basic .NET".

Notice in our case that our return value for the CheckUser() function is a User object as shown by the "As User" part of the function declaration. For an object to be returned from a WebMethod, it must be serializable.

Let's start with some specific information about what is and isn't serializable using Web Services. The answers lie in the SOAP specification. The SOAP protocol supports all the simple data types such as string, integer, single, double, date, and boolean, so functions that return values of these types are not a problem. Arrays are serializable. Some of the more complex data types such as Collection and Hashtable are not serializable. If you try to serialize an ArrayList, you'll actually get an Array on the consuming side. A very flexible object called DataSet is serializable and is helpful with complex data structures.

As mentioned, if you wish to return your own custom objects, you must declare the class serializable by placing the <Serializable()> attribute tag in front of the class.

```
<Serializable()> Public Class User
```

The "" attribute allows the read-write properties of the class to be serialized, so that it can be used as a return value to a WebMethod, or used as an input parameter of a WebMethod. As mentioned earlier, read-only properties are not serialized nor are write-only properties. These limitations can reduce the elegance of the resulting code and requires that you protect properties with separate code on the server side.

Serializing complex data types

We found some interesting ways to get around some of the serialization issues in Web Services. For example, as mentioned, the collection and ArrayList objects are not serializable. If you try to serialize a collection, you'll get an error. If you try to serialize an ArrayList, you actually get an array back. We really wanted to use an ArrayList for our application and did not want to deal with Arrays with the messy resizing code that accompanies them. So here's how we handled that issue...

We have a WebMethod called GetCategoryList() in our application that returns a list of product categories such as computers, printers, monitors, etc. Since we could not return an ArrayList, we instead returned our own custom object called ArrayListWrapper. This allows us to have a public property Coll that permitted us to access the ArrayList contained within.

```
   <WebMethod(Description:="{description omitted for readability")> _
```

```
    Public Function GetCategoryList() As ArrayListWrapper
    ' Code was omitted for readability.

' ArrayListWrapper was created to mimic ArrayList functionality
<Serializable()> Public Class ArrayListWrapper
    Private mColl As ArrayList

    Public Property Coll() As ArrayList
        Get
            Return mColl
        End Get
        Set(ByVal Value As ArrayList)
            mColl = Value
        End Set
    End Property

    Public Sub Clear()
        mColl = New ArrayList()
    End Sub

    Public Sub Add(ByVal Item As Object)
        mColl.Add(Item)
    End Sub
End Class
```

This also allowed us on the client side to access this object easily without all of the issues with arrays. If you like array programming, by all means, use that technique. We don't!

We declared a private field member called CatList in our frmBrowse code and then used the CatList ArrayListWrapper.

```
    Private CatList As GrayMarketWebService.ArrayListWrapper
```

```
    CatList = SrvPriceCompare.GetCategoryList

    ' list the categories
    For I = 0 To CatList.Coll.GetLength(0) - 1
        Cat = CType(CatList.Coll.GetValue(I), GrayMarketWebService.IdValue)

        Me.lstCategories.Items.Add(Cat.Id & " - " & Cat.Value)
    Next
```

Default Constructors

Constructors are used in object oriented languages like VB .NET to create new instances of objects. The default constructor is indicated by the subroutine name "New", takes no parameters, has no executable code, and creates a default instance of an object. For example, in our User Class the default constructor appears as follows:

```
    Public Sub New()

    End Sub
```

Often developers create non-default or parameterized constructors that construct an object based on a set of input parameters. For example, we used a non-default constructor for our User object that took several parameters such as UserId, Username, Password, Firstname, Lastname, and Email.

```
    Public Sub New(ByVal UserId As String, ByVal Username As String, _
            ByVal Password As String, ByVal Firstname As String, _
            ByVal Lastname As String, ByVal Email As String)
        mUserId = UserId
        mUsername = Username
        mPassword = Password
        mFirstname = Firstname
        mLastname = Lastname
        mEmail = Email
    End Sub
```

Default constructors must be present in all serializable objects that will be returned from a WebMethod. The default constructor is implicit in VB .NET. Non-default constructors are not available to applications consuming the Web Service. Along the same lines, if you have code in the default constructor, this code will not be executed if the object is constructed on the consuming side. This could be of concern if you want to employ some business logic on the consuming side of you application. And this again reinforces the theme of Web Services, in that you can not enforce server-side code on the consuming client.

You can have a non-default constructor in a serializable object, but it will only be available to other code in the same project. If there is a non-default constructor in a serializable object, you need to explicitly include the default constructor.

One way that we found to get around not having access to constructors is to have a WebMethod construct the object for you. For example:

```
<WebMethod(Description:="Neat work around to enable parameterized " & _
      "constructor-like functionality")> _
Public Function ReturnNewUser(ByVal UserId As String, _
      ByVal Username As String, ByVal Password As String, _
      ByVal Firstname As String, ByVal Lastname As String, _
      ByVal Email As String) As User
   Return New User(UserId, Username, Password, Firstname, Lastname, Email)
End Function
```

Overloading

In addition to the serialization issues we faced, we also found some interesting tidbits around overloading. Overloading allows many methods with the same name and return type to exist with different input parameters. During our development, we were not aware of the overloading problems, so we proceeded to write overloaded functions.

```
   <WebMethod(Description:="{description omitted for readability}")> _
   Public Function GetProductList(ByVal CategoryId As Integer) As ArrayListWrapper
      Return GetProductByCategoryId(CategoryId)
   End Function

   <WebMethod(Description:="{description omitted for readability}", _
   Public Function GetProductList(ByVal Category As IdValue) As ArrayListWrapper
      Return GetProductByCategoryId(Category.Id)
   End Function
```

Visual Studio.NET would allow us to write overloaded functions and compile them with no errors or warnings. But when we attempted to add a Web Reference to the Web Service, we started to see problems. The error message below was displayed.

[illegible]

click on image for full view

We would have liked to know about the problem at development time rather than when we tried to consume the service. In any event, with some help from Mike Amundsen at EraServer.net, we solved the problem. It turns out you cannot just simply overload methods as you would when working locally with an object

The problem is that the VisualStudio.NET tries to name the two SOAP messages the same since the SOAP message name comes from the function name. The SOAP message is the name used in the XML SOAP protocol for the method. The problem is that SOAP cannot handle duplicate message names representing different things. However, the VS.NET IDE provides a nice workaround through the use of the "MessageName" parameter on the WebMethod attribute. This parameter changes the message name in the SOAP XML protocol so there is no conflict. We suggest that you make this name verbose so that other people that are not using VS.NET can interpret it correctly. Our example below shows how to use the MessageName parameter.

```
   <WebMethod(Description:="{description omitted for readability}")> _
         MessageName:="GetProductListByCategoryId")> _
   Public Function GetProductList(ByVal CategoryId As Integer) As ArrayListWrapper
      Return GetProductByCategoryId(CategoryId)
   End Function

   <WebMethod(Description:="{description omitted for readability}", _
         MessageName:="GetProductListByCategory")> _
   Public Function GetProductList(ByVal Category As IdValue) As ArrayListWrapper
      Return GetProductByCategoryId(Category.Id)
   End Function
```

We were happy to have overloading as part of our Web Services arsenal.

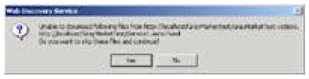
Ambiguous Errors

In programming, there's nothing worse than an error that goes something like this, "Something is wrong with your code." Although Microsoft does a pretty good job overall, when in comes to Web Services and Web References, we found a lot of ambiguous errors that often frustrated us when trying to solve a problem.

For example, the following situations gave rise to ambiguous errors.

1. Creating a Web Reference to a Web Service with a problem (e.g. overloaded function without using the MessageName property)
2. Updating a Web Reference to a Web Service with a problem
3. Forgetting to update the Web Reference to a Web Service whose interface has changed and then running the code

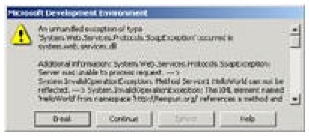
For example, here is the error for cases 1 and 2.


Web Discovery Service

Unable to download following files from http://localhost/[illegible] [illegible]
Do you want to skip these files and continue?

click on image for full view

So do you have any thoughts on what the problem is? Neither did we. And what about the following error we received when we forgot to update our Web Reference before running the code?


Microsoft Development Environment

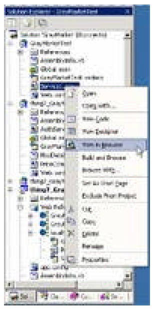
An unhandled exception of type 'System.Web.Services.Protocols.SoapException' occurred in system.web.services.dll

Additional information: System.Web.Services.Protocols.SoapException: Server was unable to process request. ---> System.InvalidOperationException: Method Service1.HelloWorld can not be reflected. ---> System.InvalidOperationException: The XML element named 'HelloWorld' from namespace 'http://tempuri.org/' references a method and

click on image for full view

While this one gives you a little more information it is still misleading. If you weed through it you will find the statement "The XML element named 'HelloWorld' from namespace 'http://tempuri.org/' references a method and a type. Change the method's message name using the attribute or change the type's root element using the XmlRootAttribute." You might be lead to believe that you have a datatype called "HelloWorld". But, no that is not the case. The problem is that we changed code that changed the interface to the Web Service but never updated the Web Reference.

To see a better-formatted version of this message, right-click on the Web Service where there is a problem, and then click "View in Browser" as seen below.



click on image for full view

Server Error in '/Thing2_GrayMarketWS' Application.

[illegible]

click on image for full view

We've reached the end of Part III. In our final chapter coming soon, we'll cover Web Services platform considerations, plus deploying and publishing Web Services.