

2

Building ASP.NET Web Services

The first chapter explained what Web Services are and why you need them. In this chapter we will look at how you can build Web Services with ASP.NET.

We will discuss the various options we have when building Web Services. Examples will be shown primarily in C#, but we will also look at some VB.NET syntax for building Web Services. We will use the built-in test support in ASP.NET to test these services. The next chapter will show you how you can use the Web Services from ASP.NET.

As this is an introductory chapter, the examples will be kept simple.

More specifically, we will look at:

- ❑ The minimum requirements needed for building a Web Service with ASP.NET.
- ❑ How to build a Web Service using a text editor such as Notepad.
- ❑ How to use Visual Studio .NET to build Web Services.
- ❑ The attributes and properties we can use to control the Web Service behavior.

Building a Web Service using Notepad

Visual Studio .NET has excellent support for building Web Services. Nevertheless, it is very easy to build Web Services without the support of VS.NET. In fact, to build Web Services we really only need:

- ❑ A text editor.
- ❑ A .NET compiler, such as the C# or VB.NET compilers.

There are many reasons why you may want to build a Web Service without VS.NET. It may be too expensive or you may prefer a different text editor or development environment. Many developers, even within Microsoft, like the Emacs editor. Many large-scale projects also use makefiles to control the build process.

In the upcoming section we will use Notepad, the world's most available text editor to build a Web Service. In Windows XP, Notepad also has an option to enable line numbers, so there is one less reason to use a different text editor. But before we dig into the details of Web Services let's look at the example that we will use in this chapter and the next to illustrate the main concepts of ASP.NET Web Services.

The Application Settings Example

In this and the next chapter we will use an example that exposes the application settings of a web application as a Web Service. In ASP.NET, application settings may be stored in two places, in the `Application` object and in the `web.config` file.

If you have used classic ASP, you probably know the `Application` object already. It is a dictionary where you can store key/value pairs. It is typically used to store application-wide settings. One common use of the `Application` object is to cache application data that changes infrequently. In classic ASP, many people also use the `Application` object to store configuration settings such as a database connection string.

In ASP.NET we will typically store application configuration settings in the `web.config` file. This file is placed in the web application's root directory and can be used to configure many aspects of the web application. The `web.config` file contains several sections, including sections for authentication, security, error handling, and Web Services. We'll see several examples of configuration settings affecting Web Services later in this chapter.

For the purposes of this opening explanation, we'll look at one of the `Web.config` setting sections now; a section in which you can define your own application settings, the `appSettings` section. Here is a database connection string in the `appSettings` section:

```
<configuration>
  <appSettings>
    <add key="ConnectionString" value="user id=sa; password=;database=pubs;
                                     server=localhost" />
  </appSettings>
</configuration>
```

The name of the key, `ConnectionString`, is an arbitrary name we are using for the purposes of this example. We could use whatever name we want.

Here is a simple C# class that allows us to write strings to the `Application` object, retrieve strings from the `Application` object, and retrieve strings from the `appSettings` section of the `web.config` file. The class is available with the download package for this book as `Chapter2/wwwroot/AppServiceCS/AppService.cs`:

```
using System.Configuration;
using System.Web;
```

```

public class AppService
{
    public void SetAppState(string key, string value)
    {
        HttpApplicationState Application;
        Application = HttpContext.Current.Application;

        Application.Lock();
        Application[key] = value;
        Application.Unlock();
    }

    public string GetAppState(string key)
    {
        HttpApplicationState Application;
        Application = HttpContext.Current.Application;

        if (Application[key] == null)
            return null;
        else if (Application[key] is System.String)
            return Application[key].ToString();
        else
            return null;
    }

    public string GetAppSettings(string key)
    {
        return ConfigurationSettings.AppSettings[key];
    }
}

```

The `SetAppState()` function writes a string value to the `Application` object and associates it with the given key:

```

public void SetAppState(string key, string value)
{
    HttpApplicationState Application;
    Application = HttpContext.Current.Application;

    Application.Lock();
    Application[key] = value;
    Application.Unlock();
}

```

It uses the `Application` property of the `HttpContext` class. The context is always available from ASP.NET code. Later in this chapter we will see another way of getting at the `Application` object.

To be on the safe side, the function synchronizes access to the `Application` object by calling `Lock()` before writing to the `Application` object and `Unlock()` after writing. This is necessary since many threads may access the function at the same time.

The `GetAppState()` function retrieves the value of a key:

```
public string GetAppState(string key)
{
    HttpApplicationState Application;
    Application = HttpContext.Current.Application;

    if (Application[key] == null)
        return null;
    else if (Application[key] is System.String)
        return Application[key].ToString();
    else
        return null;
}
```

It checks if the value stored is really a string before returning it, since we are only dealing with strings in this service, not other objects that may be stored in the Application object.

The last function in the class, `GetAppSettings()`, retrieves an application setting from the `web.config` file.

```
public string GetAppSettings(string key)
{
    return ConfigurationSettings.AppSettings[key];
}
```

The function uses the `AppSettings` property of the `ConfigurationSettings` class in the .NET Framework to get this value. This property is a dictionary of all key/value pairs in the `web.config` file. Note that the `AppSettings` property is read-only. To modify the application settings we have to edit the `web.config` file.

The class in question is a regular class, and is not yet exposed as a Web Service. As it stands it can be used from `.aspx` or code-behind files within an ASP.NET web application. The next section will look at how we can expose this class to the world outside the ASP.NET web application: we are of course going to expose it as a Web Service.

Why would we want to expose this class as a Web Service? One reason might be to integrate classic ASP pages and ASP.NET pages. If you have an ASP application you may want to develop new pages in ASP.NET to take advantage of all the new features of ASP.NET. ASP and ASP.NET pages can co-exist in the same web application. However, the `Application` object of ASP.NET is separate from the `Application` object of ASP. The "Application Settings" Web Service can be used from ASP to get at the ASP.NET application state and configuration. This allows us to modify application settings such as database connection strings in one place. We will see an example of an ASP page accessing the Web Service later in this chapter.

The Application Settings service can also be used if you want to expose the application settings to other applications such as other web applications, Windows applications etc.

If you want to expose application settings as a Web Service, you should secure your Web Service so that only authorized clients can access it. Chapters 13 and 14 describes security with Web Services in detail.

For this kind of service, we can use the **IP Address and Domain Name Restrictions** settings of IIS to only allow access to this service from the same machine. To get at these settings, right-click on the Web Service file in IIS, select **Properties**, and then select **File Security**.

Exposing Application Settings as a Web Service

To expose our class as a Web Service with ASP.NET we have to do the following:

- ❑ Place our class in a file with an .asmx extension.
- ❑ Add a `WebService` directive to the top of the page.
- ❑ Add a `WebMethod` attribute to the methods we want to expose to the world.

We have other options as well when we want to expose a class as an ASP.NET Web Service, but these are the minimum requirements. The other options will be shown later.

The .asmx file is the entry point into the Web Service. The code for the Web Service can either be in the .asmx file or in a code-behind file as we'll see later.

The .asmx extension is used for ASP.NET Web Services in the same way that the .aspx extension is used for web pages. The code in the .asmx file can be any .NET language. Let us first look at how we can expose our class as a Web Service with all the code in a single file.

C#

Here is the C# code for exposing the code as a Web Service. The changes to the original code are highlighted:

```
<%@ WebService Language="C#" class="AppService" %>

using System.Configuration;
using System.Web;
using System.Web.Services;

public class AppService
{
    [WebMethod]
    public void SetAppState(string key, string value)
    {
        HttpApplicationState Application;
        Application = HttpContext.Current.Application;

        Application.Lock();
        Application[key] = value;
        Application.UnLock();
    }

    [WebMethod]
    public string GetAppState(string key)
    {
        HttpApplicationState Application;
        Application = HttpContext.Current.Application;

        if (Application[key] == null)
            return null;
        else if (Application[key] is System.String)
            return Application[key].ToString();
        else
            return null;
    }
}
```

```
[WebMethod]
public string GetAppSettings(string key)
{
    return ConfigurationSettings.AppSettings[key];
}
```

As you can see, we don't have to write any program logic ourselves to expose a Web Service. We simply add the `WebService` directive and decorate all our methods with a `WebMethod` attribute.

The `WebService` directive names the class as one to be exposed as a Web Service. We also specify that the language used in the `.asmx` file is C# so that ASP.NET knows which compiler to invoke when compiling the file.

The `.asmx` file may contain several classes, but only one class can be exposed as a Web Service.

The `WebMethod` attribute is part of the `System.Web.Services` namespace. The namespace is referenced at the beginning of the file. In C# this is done by a `using` statement. It is not really necessary to reference the namespace, as we could also have fully qualified the attribute:

```
[System.Web.Services.WebMethod]
public void SetAppState(string key, string value)
```

Referencing the namespace is a good idea, however. It makes the code more readable.

The Web Service class and all methods you want to expose in the Web Service must be declared as public.

If we add the `WebMethod` attribute to a private function, we won't get an error. The function will simply be missing from the Web Service and we won't be able to call it.

VB.NET

The VB.NET code is almost identical (except, of course, some syntactical differences):

```
<%@ WebService class="AppService" %>

Imports System.Configuration
Imports System.Web
Imports System.Web.Services

Public Class AppService

    <WebMethod> Public Sub SetAppState(ByVal key As String, ByVal value As String)
        Dim Application As HttpApplicationState
```

```

        Application = HttpContext.Current.Application

        Application.Lock()
        Application(key) = value
        Application.Unlock()
    End Sub

    <WebMethod> Public Function GetAppState(ByVal key As String) As String
        Dim Application As HttpApplicationState
        Application = HttpContext.Current.Application

        If Application(key) Is Nothing Then
            Return Nothing
        ElseIf TypeOf Application(key) Is String Then
            Return Application(key)
        Else
            Return Nothing
        End If
    End Function

    <WebMethod> Public Function GetAppSettings(ByVal key As String) As String
        Return ConfigurationSettings.AppSettings(key)
    End Function

End Class

```

The only real difference here is that we don't have to specify what the language is since VB.NET is the default language. The default compiler for ASP.NET is specified in the machine-wide configuration file. The location of this file is:

```
%SystemRoot%\Microsoft.NET\Framework\<version>\CONFIG\machine.config
```

You can override the default language in the local web.config file:

```

<configuration>
  <system.web>
    <compilation defaultLanguage="C#" />
  </system.web>
</configuration>

```

This sets the default language to C# for all ASP.NET files in the application. Other applications will still use the compiler specified in the machine-wide configuration file.

Deploying the Web Service

Deploying a Web Service is very simple. ASP.NET Web Services have the same easy deployment model as ASP and ASP.NET web pages. All we have to do is place the .asmx file in the directory of a web application.

Changing the Web Service is just as easy; we just change the source file and hit save, as in classic ASP.

If we do not have a web application, we can create one using the Internet Information Services (IIS) administration tool. We simply create a virtual directory and point this virtual directory to the folder in which we put our .asmx file.

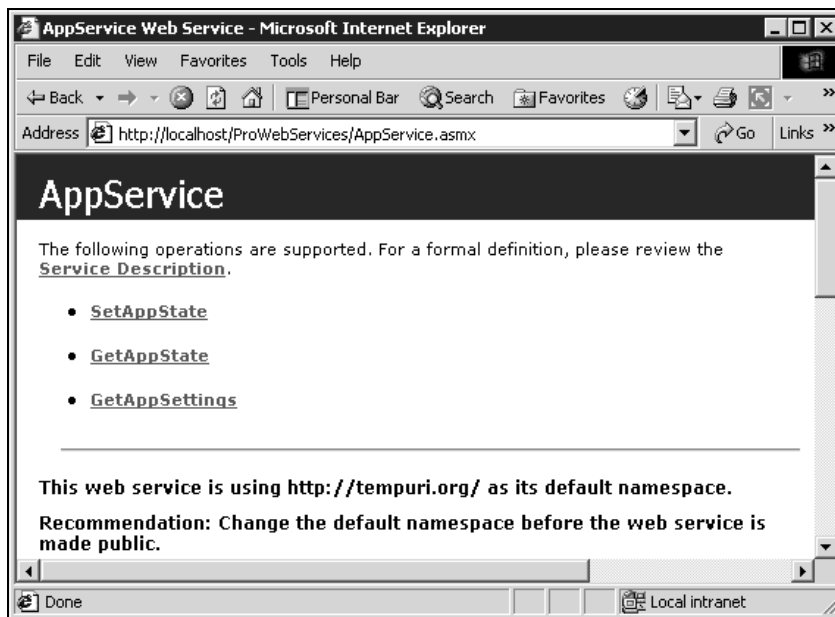
For the Application Settings example, we will assume that we have created a virtual directory on our machine at the location `http://localhost/ProWebServices/ch2`.

Testing the Web Service

So just by creating an `.asmx` file and adding a few attributes, we have created a Web Service. But wait; there's more! In addition to exposing the method as a Web Service with SOAP serialization, ASP.NET also gives us:

- ❑ Test pages we can use to test our Web Service.
- ❑ A structured description of the Web Service in the form of a Web Service Description Language (WSDL) file.

We don't have to write a test client just to test that our Web Service is functioning. We simply point our browser at the `.asmx` file:



There is note at the bottom saying that the namespace is `http://tempuri.org` and that this should be changed. To change the namespace you simply add a `WebService` directive to the Web Service class:

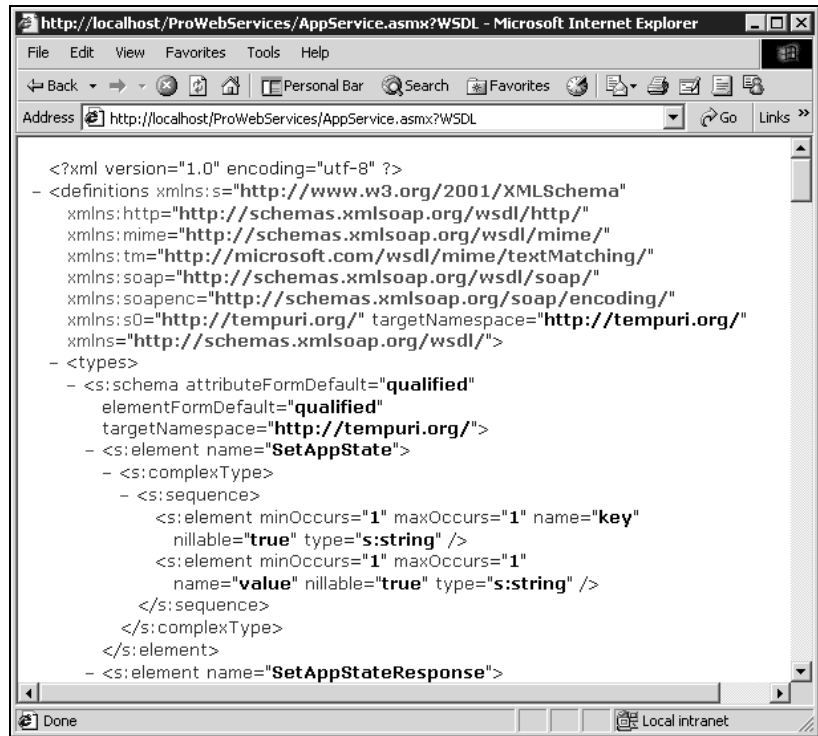
```
<%@ WebService class="AppService" %>

using System.Configuration;
using System.Web;
using System.Web.Services;

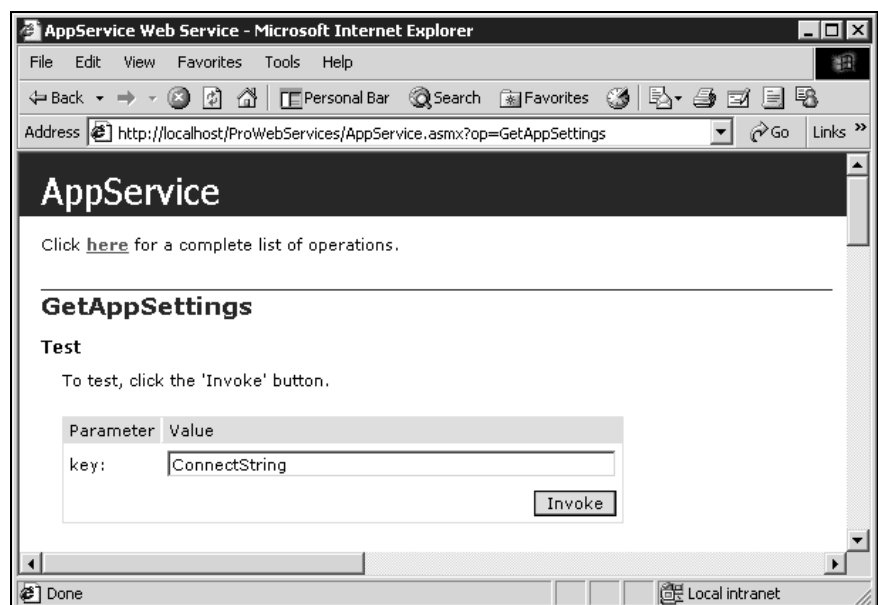
[WebService(Namespace="http://www.wrox.com/services")]
public class AppService
{
    ... code removed.
}
```


The WebService directive is described later in this chapter.

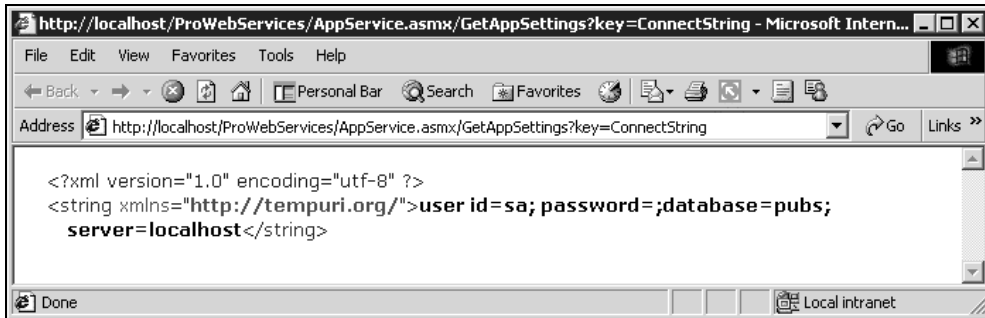
The generated page has a link to the service description. Here is a screen shot of the WSDL:



The generated page also allows us to test the individual methods of the Web Service. If we click on the **GetAppSettings** link, a test page where we can test this method appears. This test page has input fields for the input parameters of the Web Service:



We can now enter values for the field and click Invoke. This invokes the Web Service method and returns the XML result:



As was mentioned in Chapter 1, ASP.NET supports three protocols for calling Web Services: SOAP, HTTP-GET, and HTTP-POST. The test page uses HTTP-GET. With HTTP-GET, parameters are passed in the URL, as can be seen in the screen shot above. HTTP-GET returns the response as XML: in this case a single string.

The test pages make it easy for us to test our Web Services. We don't have to write a test client just to see if our methods are functioning.

With HTTP-GET and HTTP-POST, parameters are passed as name/value pairs. HTTP-GET passes the parameters in the URL; HTTP-POST passes the parameters in the HTTP request message. With SOAP, parameters are passed in an XML string. SOAP thus allows complex structures to be passed in to the Web Service.

If our Web Service takes complex parameters, we cannot use HTTP-GET or HTTP-POST and ASP.NET will not be able to render test pages. We can only test methods that take simple name/value parameters.

The test pages are rendered by ASP.NET on the fly, and as such they never exist as a file on your disk. The file that does the rendering is

```
%SystemRoot%\Microsoft.NET\Framework\<version>\CONFIG\DefaultWsdHelpGenerator.aspx
```

It may be that we want to modify the look or behavior of this page. If so, we simply copy the `DefaultWsdHelpGenerator.aspx` file to our Web Service directory, modify and rename it, and then change the `web.config` file to point to our new rendering page (for your convenience we have already added a customized rendering page to the code download package, `MyWsdHelpGenerator.aspx`):

```
<configuration>
  <system.web>
    <webServices>
      <wsdlHelpGenerator href="MyWsdHelpGenerator.aspx" />
    </webServices>
  </system.web>
</configuration>
```

The help file generator uses HTTP-GET to test the Web Service. You can modify the generator to use HTTP-POST instead. The first lines of the help generator .aspx file look like this:

```
<html>
  <script language="C#" runat="server">

    // set this to true if you want to see a POST test form
    //instead of a GET test form
    //bool showPost = true; // false is default
```

The showPost variable is, by default, set to false. If you set this variable to true, the test pages will use HTTP-POST instead of HTTP-GET.

Using the Web Service from classic ASP

One motivation for this Web Service is to synchronize application values between ASP and ASP.NET applications. Here is an example that uses the Web Service from the global.asa file of an ASP application to get the connect string from an ASP.NET web.config file:

```
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">
Option Explicit

Sub Application_OnStart()
    Application("ConnectionString") = GetAppSettings("ConnectionString")
End Sub
```

The Application_OnStart() event is fired when the ASP application starts. It calls the GetAppSettings method and stores the result in the ASP Application object.

```
Function GetAppSettings(key)

    Dim url, xmlhttp, dom, node

    'Call Web Service using HTTP-GET
    url = "http://localhost/ProWebServices/AppService.asmx/"
    url = url & "GetAppSettings?key=" & key

    Set xmlhttp = Server.CreateObject("Microsoft.XMLHTTP")
    Call xmlhttp.Open("GET", url, False)
    Call xmlhttp.send

    'Parse result
    Set dom = Server.CreateObject("Microsoft.XMLDOM")
    dom.Load(xmlhttp.responseBody)
    Set node = dom.SelectSingleNode("//string")

    If Not node Is Nothing Then
        GetAppSettings = node.text
    End If

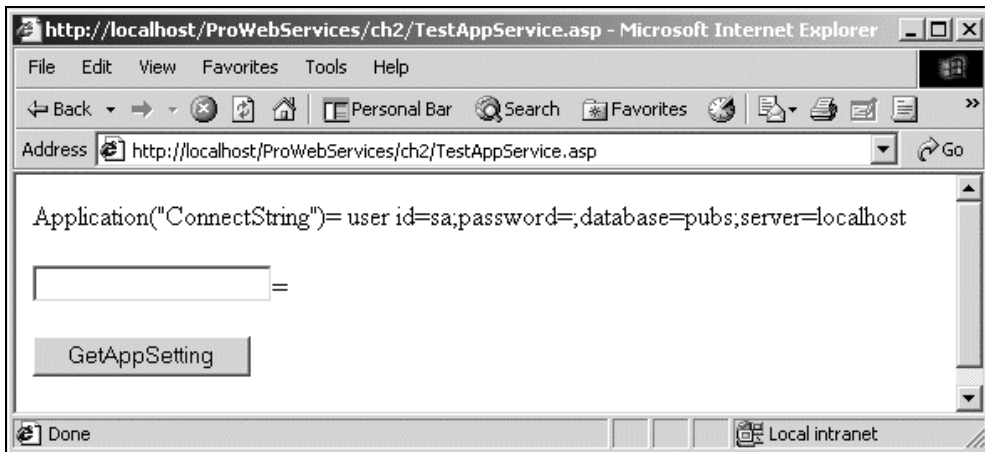
End Function

</SCRIPT>
```

The `GetAppSettings()` method uses the `XMLHTTP` object to issue a HTTP-GET to the Web Service. The `XMLDOM` object is used to search the result for the return value.

If you want to run this code you may need to change the URL in the `GetAppSettings()` function to match where you have installed the AppService Web Service.

In the download for this chapter there is a file that allows you to test the Web Service from classic ASP. The file is `Chapter2\wwwroot\TestAppService.asp`. This test file writes out the value of `Application("ConnectionString")` as set in `global.asa`. The test file also allows you to exercise the AppService Web Service. The below screen shot shows the test page in the browser:



This example illustrates one of the advantages of the HTTP-GET and HTTP-POST protocols. If you don't have any special SOAP tool support, it is much easier to format a request using these protocols than to obey by the rules of SOAP. To use SOAP you must format a valid SOAP XML string and set the appropriate HTTP headers.

Note that if the connection string is changed in the `web.config` file, the ASP application must be restarted for the changes to be updated in the ASP Application object.

Using a Separate Assembly

In our first example we placed the code for the Web Service in the `.asmx` file. The `.asmx` file is compiled to Intermediate Language (IL) code the first time anyone makes a request to the service.

We can also place the Web Service class in a separate code-behind file and have it compiled to a separate assembly. You then need two files:

- ❑ One `.asmx` file with the `WebService` directive
- ❑ One file with the Web Service class.

The `.asmx` file contains a single line, the `WebService` directive:

```
<%@ WebService class="AppService,AppServiceAssembly"%>
```

The `class` property names the class and, optionally, the assembly.

To create the assembly, you must compile the Web Service class. If you use C#, move the class in to a separate file and name it, for example `AppService.cs`. To compile the C# class, use the following command line:

```
csc /out:bin/AppServiceAssembly.dll /t:library  
/r:System.dll,System.web.services.dll,System.web.dll AppService.cs
```

We must reference the assembly files used in our Web Service class, in this case `System.dll`, `System.web.dll`, and `System.web.services.dll`. The compiled assembly, `AppServiceAssembly.dll`, contains the IL code. It must be placed in the `bin` directory of the web application.

In the `WebService` directive above we named the assembly. We could have also just named the class, like this:

```
<%@ WebService class="AppService" %>
```

If we don't name the assembly in the directive, ASP.NET searches all assemblies in the `bin` directory for the class.

Why Use a Separate Assembly?

What are the advantages and disadvantages of using a separate assembly? The answer is: it depends.

If you have your code in a separate assembly it may also be used as a normal assembly from within the web application and from other applications:

- ❑ We can use the class directly within our web application without incurring the overhead of SOAP and HTTP calls. Thus the same class may be used both internally within the application and externally as a Web Service.
- ❑ We can also use all the other features of an assembly such as versioning.

If we do use a separate assembly, we must manually invoke the compiler when we modify the class file. This makes deployment a bit harder, as we can't simply edit and hit save. On the other hand, it can be better to detect programming errors at compile time instead of having the first user hitting the service detect the error. The user of a Web Service is typically another program. If the calling program does not report errors properly, it may be hard to figure out what has caused the error.

If we have the source code in the `.asmx` file we also deploy the source code. We may not want to expose the source code to everyone who has access to the server where our Web Service is hosted. The compiled IL code is harder to read, modify and tamper with than un-compiled C# or VB.NET code.

Building a Web Service with Visual Studio .NET

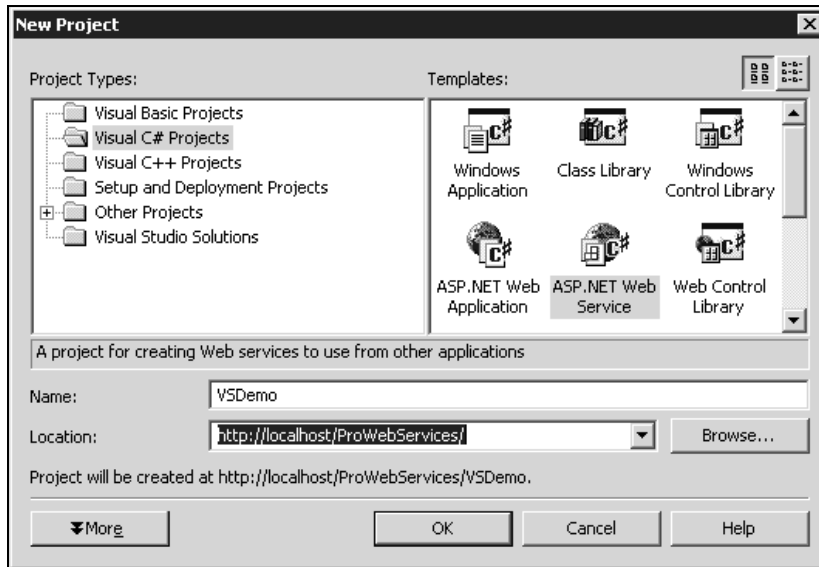
The previous section showed how we can create a Web Service with ASP.NET simply by using a text editor and the compilers that ship with the .NET Framework.

We can also use the support of Visual Studio .NET to create Web Services.

Building and Running a HelloWorld Example

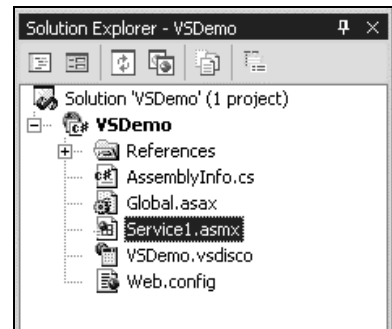
To demonstrate how to build a Web Service with Visual Studio .NET, we are going to use the simplest example of all, the HelloWorld example. In fact, when we create a Web Service project with Visual Studio .NET, it automatically creates a HelloWorld example as a starting point.

To follow this example, start up Visual Studio .NET and create a new C# ASP.NET Web Service Project.



In the dialog that appears, you can name the project. Let's name it VSDemo. You also specify the location of the project. VS.NET automatically creates a virtual directory in IIS named VSDemo at this location.

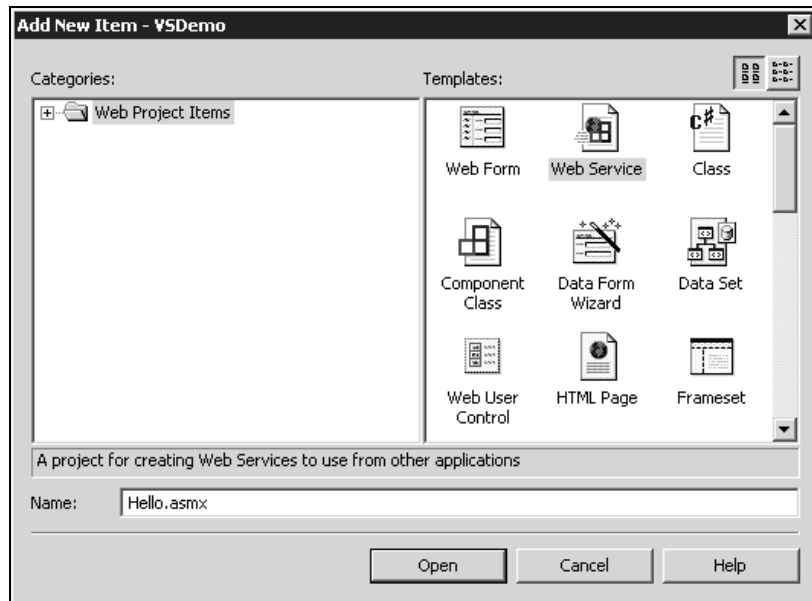
When we click OK, VS.NET gives us a number of files. The screenshot below shows the Solution Explorer with the files created by VS.NET.



One of the files is named `Service1.asmx` and contains the actual Web Service code. The name of the Web Service class is also `Service1`. In the real world, we wouldn't want all our services to be named 'Service1', so the first thing we need to do is come up with a better name. Since we are building a HelloWorld example, let's name the Web Service `Hello`.

The service is deliberately not named HelloWorld. The Web Service class will contain a method named HelloWorld, and a C# class cannot have a method with the same name as the class. In C#, a method with the same name as the class is a constructor.

When we rename the `.asmx` file, VS.NET does not propagate the changes to the code. Instead of manually going through the code and changing the name, it is often easier to just delete the `Service1.asmx` file and add a new Web Service file with a more meaningful name. We can add a new Web Service file by selecting File | Add New Item.



Let us take a look at the code generated by VS.NET for the HelloWorld example:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace VSDemo
{
```

```
/// <summary>
/// Summary description for Hello.
/// </summary>
public class Hello : System.Web.Services.WebService
{
    public Hello()
    {
        //CODEGEN: This call is required by the ASP.NET Web Services Designer
        InitializeComponent();
    }

    #region Component Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
    }
    #endregion

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
    }

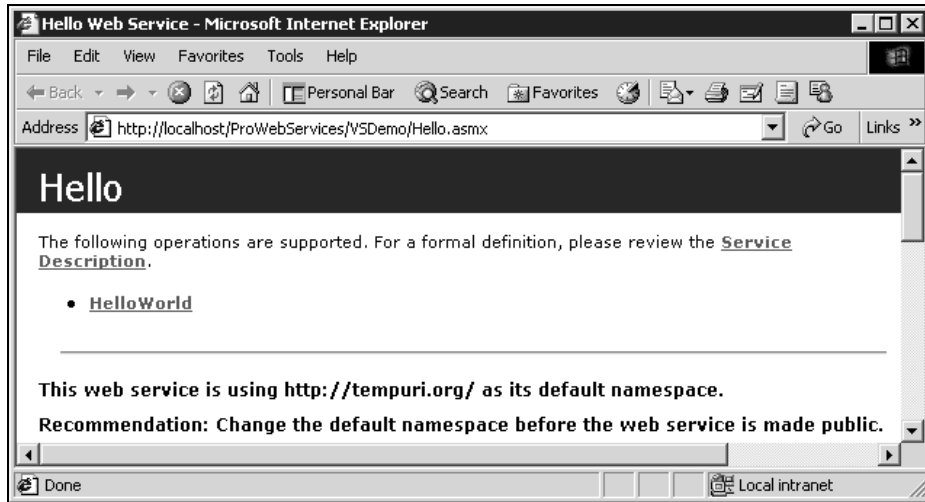
    // WEB SERVICE EXAMPLE
    // The HelloWorld() example service returns the string Hello World
    // To build, uncomment the following lines then save and build the project
    // To test this Web Service, press F5

    // [WebMethod]
    // public string HelloWorld()
    // {
    //     return "Hello World";
    // }
}
```

The code contains a HelloWorld example. You can see that the code is similar to the code we created earlier.

Visual Studio .NET also adds some code so that you can work with the visual designer. Using the designer you can drag components from the toolbox into your Web Service. These may be components such as timer components, event log components, and so on.

To run the Web Service, simply uncomment the HelloWorld example and set the `Hello.asmx` file as the start page for the project by right-clicking on it in the Solution Explorer and selecting **Set As Start Page**. Then hit **Start** in the **Debug** drop down menu. VS.NET will compile your code and run it: ASP.NET renders the test page as we saw before:



Selecting HelloWorld will bring up a test page where you can invoke the HelloWorld function.

The code generated for a new Web Service is copied from a template. You can modify the template for new Web Services.

The C# template is stored in the file:

```
\Program Files\Microsoft Visual Studio  
.NET\VC#\DesignerTemplates\1033\NewWebServiceCode.cs
```

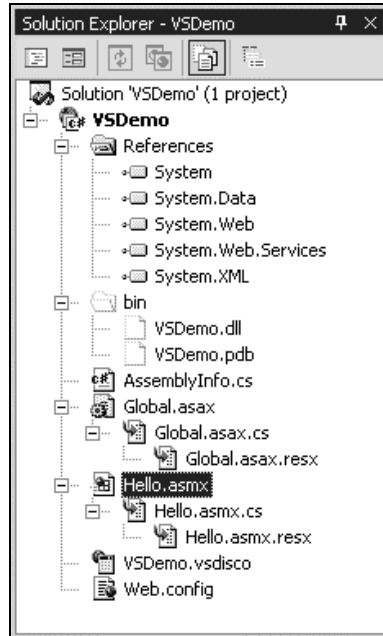
The VB.NET template is stored in the file:

```
\Program Files\Microsoft Visual Studio  
.NET\Vb7\VBWizards\DesignerTemplates\1033\NewWebServiceCode.vb
```

Generating the skeleton for the Web Service is easy. Now the hard work starts, making the Web Service do something useful. This is up to you!

The Anatomy of a VS.NET Generated Web Service

When we create a new Web Service project with VS.NET, quite a number of files are generated. There are actually more files than meet the eye at first. If we select **Show All Files** in the Solution Explorer we will see some additional files:



File/Folder	Description
References	The references are not files, but references to assemblies used by the Web Service. They are stored in the VS.NET project file.
Bin	The bin folder contains the assemblies for our Web Service. You can view the VSDemo.dll using the IL disassembler.
AssemblyInfo.cs	The AssemblyInfo.cs file contains information such as title, description, and version that is added to the assembly.
Global.asax	The Global.asax file can be used for Web Services the same way it is used for web applications. You can add code to the events available in the Global.asax file.
Global.asax.cs	
Global.asax.resx	
Hello.asmx	The Hello.asmx file contains the WebService directive.
Hello.asmx.cs	The Hello.asmx.cs file contains the code-behind for the Web Service.
Hello.asmx.resx	A file consisting of XML entries for resource added in the designer. Is normally empty.
VSDemo.vsdico	The Web Service discovery file. See Chapter 7 for more information about Disco.
Web.config	The configuration file for the Web Service.

The `Hello.aspx` file contains a single line, the Web Service directive (to open this file from Visual Studio.NET, right-click on the file and select **Open With...** then select the **Source Code (Text) Editor**):

```
<%@ WebService Language="c#" Codebehind="Hello.aspx.cs" Class="VSDemo.Hello" %>
```

This is the same directive we saw earlier. Note the `Codebehind` attribute. This attribute is not used by ASP.NET. It is used by Visual Studio to associate a code-behind file with the `.asmx` file.

Visual Studio .NET uses code-behind for Web Services. If you don't want to use code-behind in VS.NET, remove the `Codebehind` attribute from the `.asmx` file and delete the codebehind file.

Why Use Visual Studio .NET?

So if it is so easy to create Web Services with a simple text editor such as Notepad, why should we use Visual Studio .NET? The answer is simple: VS.NET is a fantastic tool.

VS.NET gives us an integrated development environment where we can develop, debug, test and deploy the Web Service. We also get Intellisense, which completes the code for us.

As we have seen, VS.NET also does some of the plumbing for us; it creates files and a virtual directory in IIS where we can develop and test our Web Service.

More On Building Web Services

In this section, we are going to look beyond just VS.NET, at the options we have to configure a Web Service using the various directives and attributes that are available with ASP.NET. We will look at:

- ❑ The `WebService` directive
- ❑ The `WebService` attribute
- ❑ The `WebMethod` attribute
- ❑ The `WebService` class

WebService Directive

The `WebService` directive is placed on the first line of an `.asmx` file. It specifies the class to expose as a Web Service.

```
<%@ WebService Language="c#" Codebehind="Hello.aspx.cs"  
Class="VSDemo.Hello,VSDemo" %>
```

The directive is required for ASP.NET Web Services. If the class for the Web Service is in a code-behind file, this line will be the only line in the `.asmx` file.

The directive may use the following attributes:

- ☐ Language
- ☐ Codebehind
- ☐ Class

Language

The Language attribute is optional. It specifies the language to use to compile the Web Service. Any .NET compiler installed on the system may be specified. By default the installed compilers are C#, VB.NET and JScript.NET. These are specified using the values VB, C#, or JS.

The default language is VB.NET as specified in the `machine.config` file, unless it is overridden in the `web.config` file.

Codebehind

The Codebehind attribute is optional. It is used by Visual Studio .NET to find the code-behind file for the `.asmx` file so that when you click on the `.asmx` file it can open the code-behind file. The attribute is only used in Visual Studio .NET, it has no effect when the Web Service is executing.

Class

The Class attribute specifies the class to expose as a Web Service. If the class is within a namespace, the namespace is specified, as in the following example. In this example, the namespace is `VSDemo` and the class is `Hello`:

```
<%@ WebService class="VSDemo.Hello"%>
```

If the class is not within a namespace, we specify the class directly:

```
<%@ WebService class="Hello"%>
```

The class attribute may optionally also specify the assembly where the class exists:

```
<%@ WebService class="VSDemo.Hello,VSDemo"%>
```

If the assembly is not specified, ASP.NET searches all assemblies in the `bin` directory for the `Hello` class.

WebService Attribute

The `WebService` attribute is used to add additional information about a Web Service such as a description of the Web Service. The attribute is implemented by the `System.Web.Services.WebServiceAttribute` class and may use any of the following properties:

- ☐ Description
- ☐ Name
- ☐ Namespace

The attribute is added to the Web Service class as follows:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace VSDemo
{
    [WebService(Description="This is a simple test service",
        Name="Hello Universe",
        Namespace="http://www.wrox.com/services")]
    public class Hello : System.Web.Services.WebService
    {
        public Hello()
    }
}
```

This attribute is not required to build Web Services. The attribute and its properties are added to the metadata of the Web Service. The metadata is stored in the assembly along with the compiled code.

Now, let's go on to see what each property does.

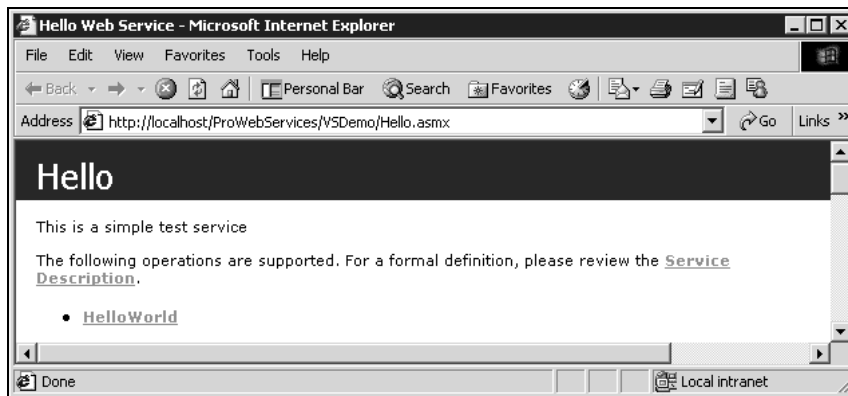
Description

The Description property is used to add a description to the Web Service:

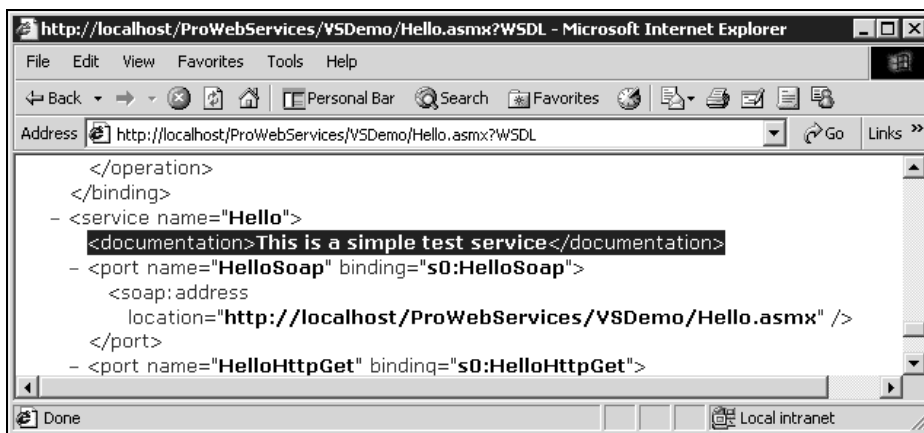
```
[WebService(Description="This is a simple test service")]

public class Hello : System.Web.Services.WebService
{
}
```

ASP.NET uses the description when rendering test pages.



When ASP.NET creates the WSDL file it also includes the description in the <documentation> element of the WSDL service description, which we can view by clicking the **Service Description** link on the test page:

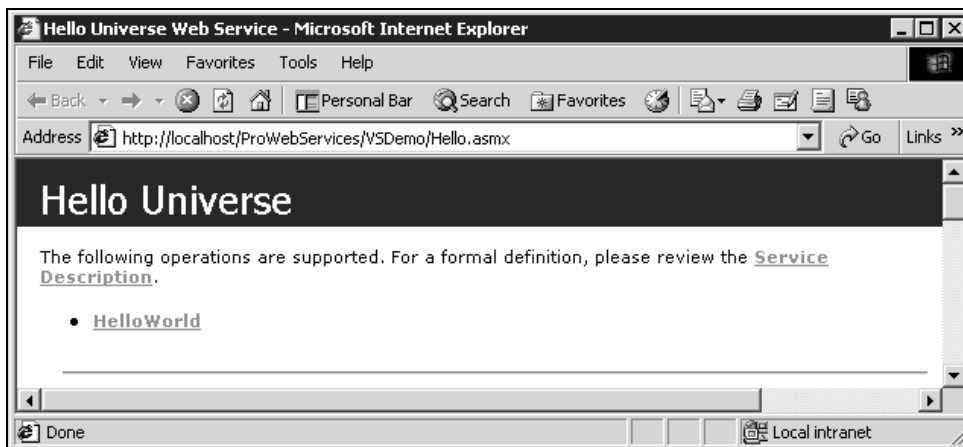


Name

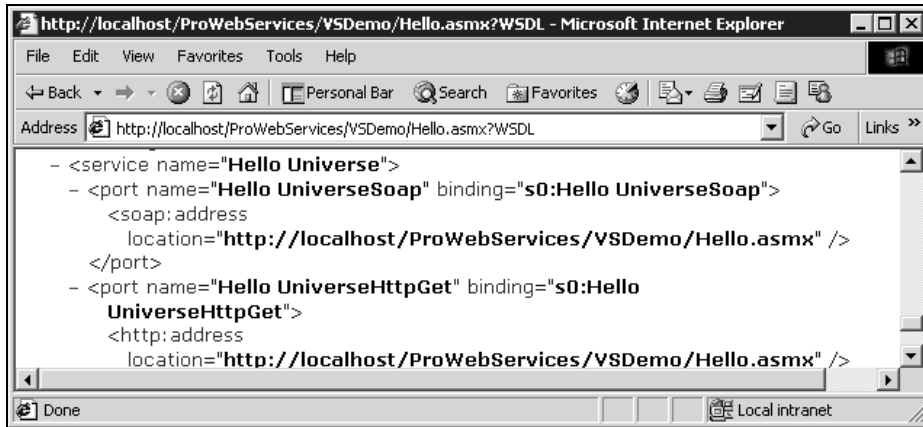
By default, the name of the Web Service is the same as the name of the class. The Name property is used to give the Web Service a different name from that of the class. We'll discuss why in a moment, here is how you use the property:

```
[WebService(Name="Hello Universe")]  
public class Hello : System.Web.Services.WebService  
{
```

The value specified becomes the name of the Web Service. Here is the test page:



The new name is also exposed to the outside world through the WSDL:



Why would we want to give the Web Service a different name? It may be a matter of taste: we can use a name that is different from the internal name of the class. If we use an internal naming convention for classes for instance, we may not want to use the convention when exposing the class as a Web Service.

We may also give the Web Service a name that we are not allowed to use for our class. This may for instance be a name that is reserved in the language. Say we are building a trade application and we want a Web Service with the name 'Imports'. We are not allowed to use this as the name of a VB.NET class since Imports is a key word in VB.NET. However, by using the Name property we can give the Web Service the 'Imports' name. Another example is if you develop your Web Service in C# and you want the service to have the same name as a method. You saw earlier that we named our HelloWorld Web Service 'Hello' to avoid this issue. By using the Name property you can name the service HelloWorld and still have a method with the same name.

All the generated test pages we have seen up to now have had a message on them:

This Web Service is using http://tempuri.org/ as its default namespace.

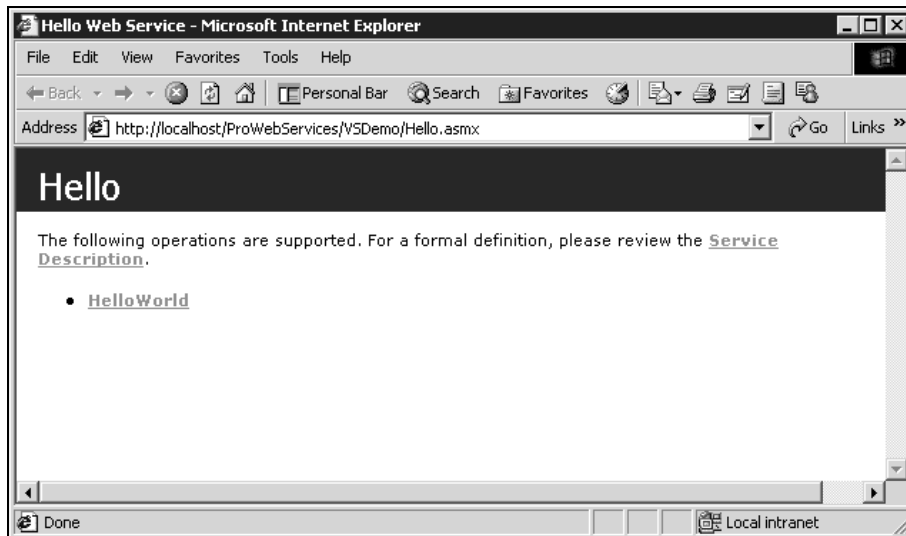
Recommendation: Change the default namespace before the Web Service is made public.

The pages also explain how you change the namespace. You do that by setting the Namespace property of the WebService attribute:

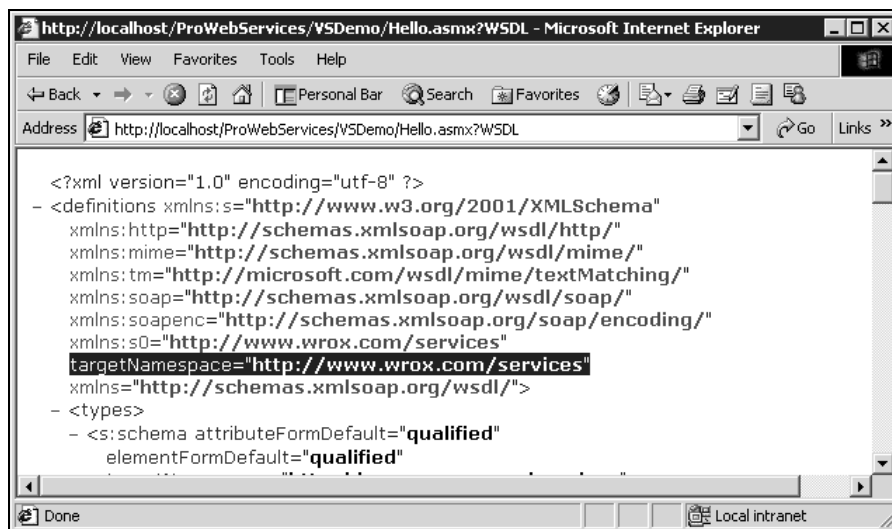
```
[WebService(Namespace="http://www.wrox.com/services")]
public class Hello : System.Web.Services.WebService
{
```

The namespace uniquely identifies your Web Service methods and allows two different Web Services to have methods with the same name.

When you set the Namespace property the warning disappears:



The namespace is also used in the WSDL:



The namespace is a URI or a Universal Resource Identifier. This may be a URL, but it need not be. It need not point to a valid HTTP address. It should be unique so we should use a name that we control. It is a good idea to use your company's domain name as a starting point.

If we don't specify the namespace property, ASP.NET uses `http://tempuri.org/`. This name should only be used during development and should be changed before we go live with our Web Service.

WebMethod Attribute

We have now seen the `WebService` directive and the `WebService` attribute. They apply to the Web Service as a whole.

The `WebMethod` attribute is added to each method we want to expose as a Web Service.

The attribute is implemented by the `System.Web.Services.WebMethodAttribute` class and it supports a number of properties that control the behavior of the methods.

The following properties are available:

- ☐ `CacheDuration`
- ☐ `Description`
- ☐ `EnableSession`
- ☐ `MessageName`
- ☐ `TransactionOption`
- ☐ `BufferResponse`

Here we will describe each of the properties with simple examples. The examples can be found in the `Properties.asmx` file in the code download. Caching with Web Services is described in more detail in Chapter 12. Session state and Web Services are covered in detail in Chapter 10. Transaction handling and Web Services is covered in Chapter 11.

CacheDuration

ASP.NET has built-in support for caching the data on the server. Web Services can use the caching support of ASP.NET to cache the result of a web method.

Caching makes sense when the data returned from the Web Service does not change often and many clients call the service with the same parameters to get the same result. Good examples of this are Web Services for news feeds, currency services returning the currency conversion rate, stock quotes and so on.

Caching the result of a web method does not make sense when the results are highly dynamic. If a currency service actually takes in the amount to convert it does not make much sense to cache the result as each request will most likely be different.

The `CacheDuration` property is used to enable caching of the web method result:

```
[WebMethod(CacheDuration=10)]
public string GetTime()
{
    return DateTime.Now.ToString("T");
}
```

The duration is specified in seconds. The example above sets the cache to expire after 10 seconds. The default value is 0, which means that caching is disabled.

If the web method takes parameters, the parameters are used as keys for the cache. The result will be cached for each combination of the parameters. Here is an extension of the `GetTime()` function above that takes the format string as a parameter:

```
[WebMethod(CacheDuration=10)]
public string GetTime(string format)
{
    return DateTime.Now.ToString(format);
}
```

The result will be different for each format string. If the format string is 'd' the time will be returned in the format 'M/d/yyyy', if the format string is 'g', the time will be returned in the format 'M/d/yyyy HH:mm aa' and so on. If the web method is called within 10 seconds with the same format string, the result will be fetched from the cache.

Using the `CacheDuration` property, results from the Web Service are cached on the server on which the Web Service is running. You should also consider adding caching on the client calling the Web Service.

If the client is an ASP.NET page, we can use the ASP.NET cache to cache the result of the Web Service on the client side.

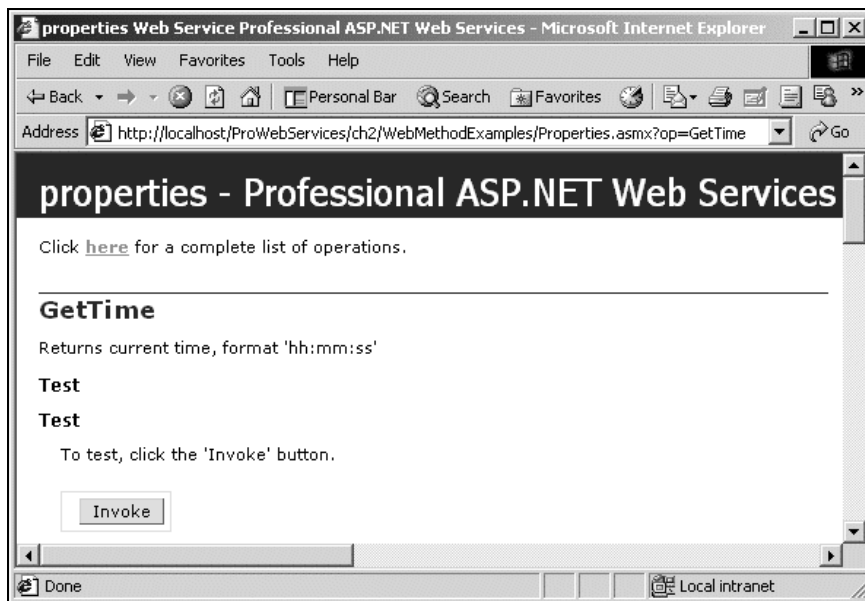
Description

The `Description` property of the `WebMethod` attribute is used to describe the web method in the same way that the `description` property of the `WebService` attribute is used to describe a Web Service.

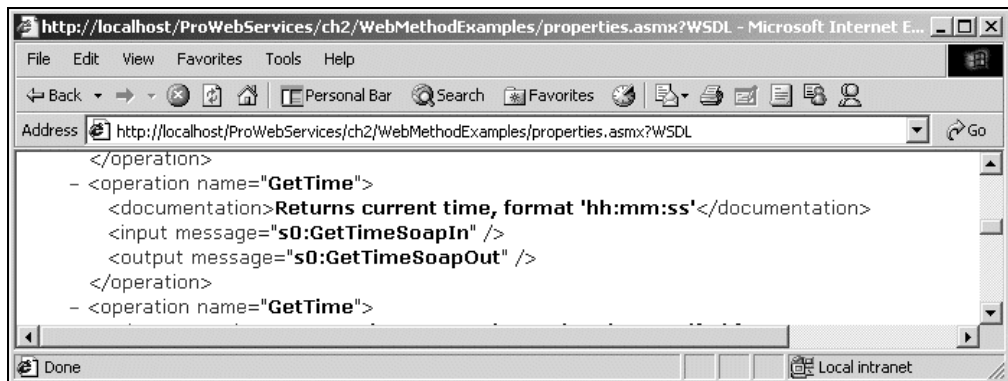
Here is an example where a description is added to the `GetTime()` method:

```
[WebMethod(Description="Returns current time, format 'hh:mm:ss'")]
public string GetTime()
{
    return DateTime.Now.ToString("T");
}
```

The description is added to the test pages:



and to the <documentation> element in the WSDL:



Adding a textual description to your Web Service and to the methods is good practice. It makes your Web Service easier to use.

EnableSession

The best practice for ASP web applications has been to disable session state. This best practice applies to Web Services as well. By default, Web Services do not support session state. Most Web Services should be designed to be stateless to achieve Internet scalability, as session state consumes memory for each client on the server.

There are cases, however, where we might want to enable session state for Web Services. We can enable session state for individual methods. To enable session state for a Web Service method we use the `EnableSession` property of the `WebMethod` attribute. Here is an example of a function that counts the number of times it is called for each user.

```
[WebMethod(EnableSession=true)]
public int UserCount()
{
    int count;
    System.Web.SessionState.HttpSessionState Session;

    Session = System.Web.HttpContext.Current.Session;

    if (Session["UsageCount"] == null)
        count = 1;
    else
        count = (int) Session["UsageCount"] + 1;

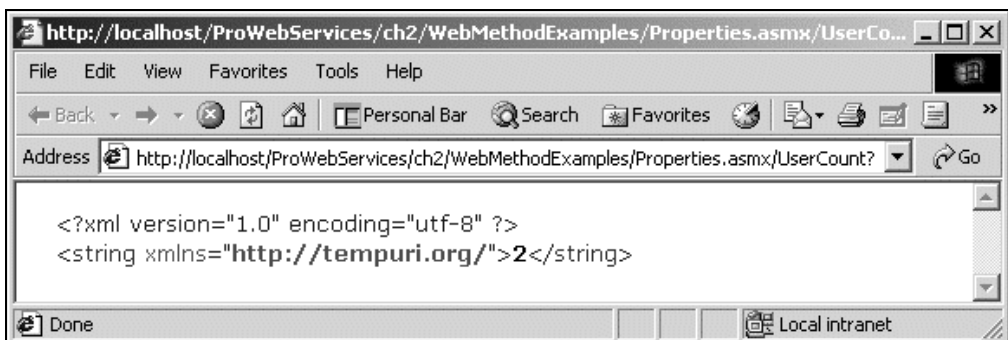
    Session["UsageCount"] = count;
    return count;
}
```

Session state is enabled by setting the `EnableSession` property to `true`. The `Session` object is used within the function to keep a usage count for each user.

Session state handling is not specified in the SOAP specification. As is often the case with Web Services, you must rely on the support of the underlying infrastructure. ASP.NET relies on HTTP cookies to support session state. The session cookie stores a session ID and ASP.NET uses the session ID to associate the client with the session state on the server.

Session handling is not specified in the SOAP specification, different SOAP implementations may handle session state differently.

When you test the `UserCount` function from a browser using the ASP.NET test pages, you get the expected behavior. Each time you refresh the browser, the count increases:



If you open a new browser instance, you will see the count start from 1 again. Note that if you open several browser instances from a test page, they will share cookies. You need to open a brand new browser instance from e.g. the Start menu.

The next chapter will show you how to generate a proxy for the Web Service. If you call the Web Service through the generated proxy you need to supply the proxy with what is called a "cookie container". The cookie container is used to host the cookies. The proxy object has a `cookieContainer` property for this purpose.

In the download available with this chapter is a Windows Forms application that can be used to test the session state handling. The form for this application is available under `/ProWebServices/ch2/TestApp/Form1.cs`. See the readme file that comes with the code for setup instructions.

The class contains a private `cookieContainer` object. This class is defined in the `System.Net` namespace:

```
public class Form1 : System.Windows.Forms.Form
{
    private static System.Net.CookieContainer cookieContainer =
        new System.Net.CookieContainer()
```

We make the variable static so that each time the callback is called, the same object will be used. If the `cookieContainer` variable is not declared as static, we will get a new `cookieContainer` object each time we click the button.

Here is an example of a button callback function in the Windows Forms application. This function calls the `UserCount` web method and displays a message box:

```
private void button5_Click(object sender, System.EventArgs e)
{
    WebMethodExamples.properties service;
    string message;

    service = new WebMethodExamples.properties();
    service.CookieContainer = cookieContainer;

    message = "Calling UserCount() returns ";
    message = message + service.UserCount();
    MessageBox.Show(message, "WebMethodExamples");
}
```

We create the Web Service proxy and set the `CookieContainer` property of the proxy to our static cookie container object. When we then call the Web Service, the session cookie will be sent to the service and we are able to use session state.

If we press the button three times we see that the count is 3.

The Web Service client may be a web application. In this case we have three parties involved;

- ☐ The user(s) of the web application
- ☐ The web application
- ☐ The Web Service

The web application needs to store the cookie container in an appropriate place. If each user of the web application needs a separate session, we can store the `cookieContainer` object in the `Session` object of the web application. Note that the session state of the web application is different from the session state of the Web Service.

If we need a session for the web application, we can store the `cookieContainer` object in the `Application` object of the web application.

Chapter 10 has more information on session handling in Web Services.

MessageName

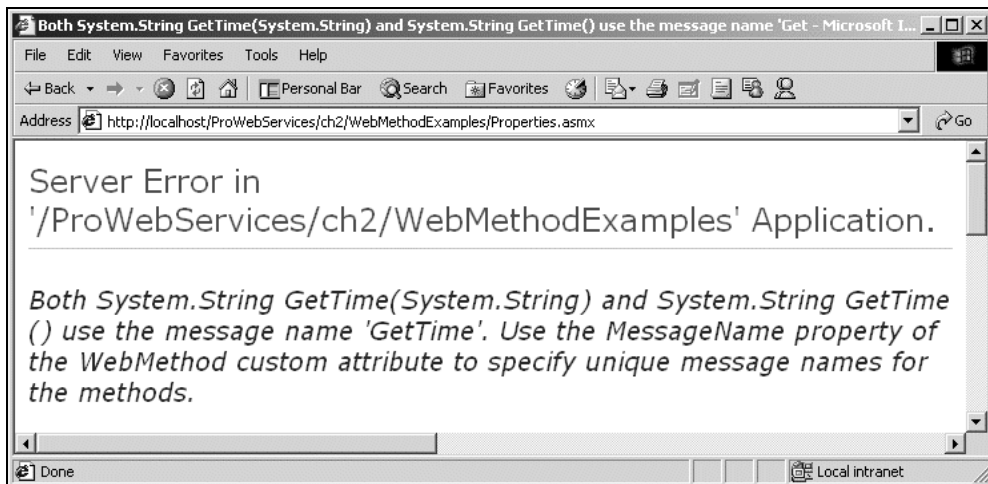
When we expose a method, the name of the Web Service method is by default the same as the name of the method in our class. There are cases when we want to give the web method a different name than the class method.

One example is with overloaded methods. In our `CacheDuration` samples we saw two different methods for returning the time:

```
[WebMethod]
public string GetTime()
{
    return DateTime.Now.ToString("T");
}

[WebMethod]
public string GetTime(string format)
{
    return DateTime.Now.ToString(format);
}
```

Since C# and VB.NET allow overloaded methods, this code compiles just fine. However, if we try to use the Web Service we will get an error. We can see this if we browse to the test page:

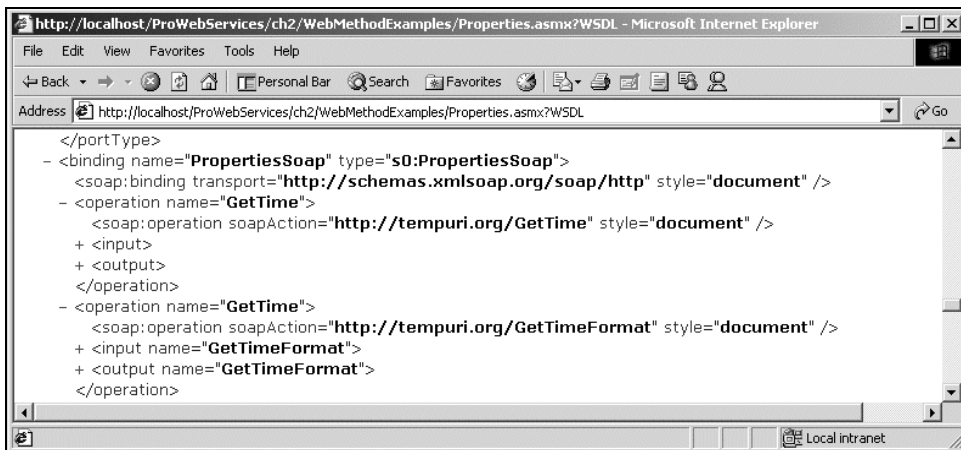


We need to supply unique names for each web method. One way to do this is to rename one method. Another way is to use the `MessageName` property to give the web method a new name:

```
[WebMethod]
public string GetTime()
{
    return DateTime.Now.ToString("T");
}

[WebMethod(MessageName="GetTimeFormat")]
public string GetTime(string format)
{
    return DateTime.Now.ToString(format);
}
```

The web method is now named `GetTimeFormat`. When the method is called, the SOAP message will identify the message as `GetTimeFormat`. The WSDL will now contain two messages, one `GetTime` message and one `GetTimeFormat` message. It will also show two `GetTime` operations:



When we build a proxy as shown in the next chapter, we will again get two `GetTime` functions, each with a different signature.

TransactionOption

ASP.NET is integrated with the transactional services of COM+. We can set ASP.NET pages to be transactional. We don't need to build COM+ components and register them in Component Services. We simply mark our ASP.NET page with a transaction attribute, and all code within that page will be in the same transaction.

ASP.NET Web Services support the same model. We can mark our web method as transactional. All code within that method will be executed in the same transaction. Note that with ASP.NET we can specify transactional support on a method level. This contrasts with COM+ components where transactional support has to be declared at the class level.

To set transaction support for a web method, we use the `TransactionOption` property of the `WebMethod` attribute as in the following example, which is part of the `Pubs.asmx` Web Service in the code download:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.EnterpriseServices;

namespace WebMethodExamples
{
    public class pubs : System.Web.Services.WebService
    {
        public pubs()
        {
            //CODEGEN: This call is required by the ASP.NET Web Services Designer
            InitializeComponent();
        }

        [WebMethod(TransactionOption=TransactionOption.Required)]
        public int UpdatePublisher(string name, bool blowup)
        {
            string conn;
            string sql;
            int rows;

            conn = "server=localhost;uid=sa;pwd=;database=pubs";
            sql = "update publishers set pub_name='" + name + "' where
pub_id=0736";

            SqlConnection connection = new SqlConnection(conn);
            SqlCommand command = new SqlCommand(sql, connection);
            connection.Open();
            rows = command.ExecuteNonQuery();

            if (blowup)
                throw new Exception("You killed me!");

            return rows;
        }
    }
}
```

The transactional services live in the `System.EnterpriseServices` namespace. So we need to set a reference to the `System.EnterpriseServices` assembly found in the `System.EnterpriseServices.dll`.

The method updates a specific row in the `publishers` table in the `pubs` database. This database is a demo database that comes with SQL Server 2000. The method takes two parameters. The first is the name of the publisher. The second is a Boolean that is used to simulate errors. If this is set to `true`, the web method will raise an error. When an error is raised, all database updates executed will be rolled back. The method returns the number of rows updated.

If you are familiar with MTS or COM+, you may ask why we don't call `SetComplete()` and `SetAbort()`. This is because COM+ has a feature called "AutoComplete" that is turned on for Web Services. If a method returns without an error, this is considered as a `SetComplete()`. If the method raises an error, this is considered a `SetAbort()`.

`System.EnterpriseServices.TransactionOption` is an enumeration with the following possible values:

- ☐ Disabled
- ☐ NotSupported
- ☐ Required
- ☐ RequiresNew
- ☐ Supported

A web method will always be the root of a transaction, so the only value that makes sense for a Web Service is `Required`. `RequiresNew` will have the same effect; turning on transactional support.

A Web Service cannot participate in an ongoing transaction. It will always start a new transaction. The Web Service and the client cannot share transaction context.

If a transacted Web Service calls COM+ transacted objects, these objects are able to participate in the transaction of the Web Service. If a transacted Web Service calls another transacted Web Service, they will not be part of the same transactions. Transactions are covered in more detail in Chapter 11.

BufferResponse

The `BufferResponse` property allows you to control when data returned from the Web Service is sent to the client. By default, the `BufferResponse` property is set to `true`. This means that the entire result is serialized before it is sent to the client. By setting this property to `false`, ASP.NET will start returning output as it is serialized.

```
[WebMethod(BufferResponse=false)]
public DataSet GetLargeResultset()
{
    // do something
}
```

Setting `BufferResponse` to `False` does only make sense when the Web Service returns a large amount of data.

The Web Service method will always complete execution before anything is returned. Buffering on/off relates to the serialization that takes place after the method has executed. With buffering turned off, the first part of the result is serialized and sent. Then the next part of the result is serialized and sent, and so on.

Deriving from the WebService Class

There is one more option for implementing Web Services that we have not yet discussed. A Web Service class may inherit from a class in the `System.Web.Services` namespace, the `WebService` class. You may have seen this already, since the code emitted by Visual Studio .NET inherits from this class:

```
public class Service1 : System.Web.Services.WebService
{
    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

Inheriting from the `WebService` class is mainly a convenience; it does not add any functionality to our Web Service.

The convenience is that the `WebService` class provides direct access to the `Web.HTTPContext` object of the current request. We saw earlier in this chapter that to get at the `Application` object we had to write the following code:

```
HttpApplicationState Application;
Application = HttpContext.Current.Application;
```

By inheriting from the `WebService` class we can access the `Application` object directly. The code above becomes unnecessary, so we can write:

```
public class Service1 : System.Web.Services.WebService
{
    [WebMethod]
    public void SetAppState(string key, string value)
    {
        Application.Lock();
        Application[key] = value;
        Application.Unlock();
    }
}
```

The `WebService` class adds the following properties:

Property	Description
Application	The <code>Application</code> object holding the application state of the Web Service.
Context	The <code>Web.HTTPContext</code> object of the current request.

Property	Description
Server	The <code>HTTPServerUtility</code> object of the current request. This is similar to the <code>Server</code> object found in ASP. It contains, for instance, the <code>CreateObject()</code> function.
Session	The <code>Session</code> object holding the session state of the Web Service. See the description of the <code>EnableSession</code> property earlier.
User	The <code>User</code> object is used to get access to security information such as whether the user is authenticated and the name of the authenticated user. See Chapter 13 for more information about security.

Since the .NET Framework only supports single inheritance, inheriting from `WebService` means that our class cannot inherit from other classes. This is really the only reason not to inherit from `WebService`.

An interesting point with inheriting from `WebService` is that `WebService` is derived from the `System.MarshalByRefObject` class. This class is the base class for .NET Remoting.

This allows our class to be remoted using Remoting as well. Remoting is a Microsoft proprietary technology available in the .NET Framework. It allows objects to be passed by reference between machines and application domains. .NET Remoting is somewhat similar to DCOM but is much more flexible. We can configure how data should be serialized, if it should use binary, XML, or other formats, and how data should be transported: using TCP, HTTP etc.

By inheriting from `WebService` (and indirectly from `MarshalByRefObject`) we have several options for our class:

- ❑ The class can be used directly within an application, without going through serialization at all.
- ❑ The class can be exposed as an ASP.NET Web Service.
- ❑ We can configure the class to use .NET Remoting and for example we could use a binary formatter over TCP.

If you want to know more about .NET Remoting, you can read the upcoming book *Professional C# Web Services* ISBN 1861004397 also by Wrox Press.

Summary

This chapter has showed how we can create basic Web Services using ASP.NET. We can create a Web Service using any text editor or we can use Visual Studio .NET.

An ASP.NET Web Service exists in an `.asmx` file. The code for the service can be in the same file or in a code-behind file.

By adding a directive and a few attributes to our code, a .NET class is exposed as a Web Service. ASP.NET gives us:

- ❑ Support for HTTP-GET, HTTP-POST and SOAP.
- ❑ Test pages we can use to test the Web Service.
- ❑ A description of the Web Service in the form of a WSDL file.

The following table summarizes the main options we have when building ASP.NET Web Services:

Option	Mandatory	Description
Web Service Directive	Yes	Added as first line in an <code>.asmx</code> file.
<code>WebService</code> attribute	No	Added to the class. Used to set the namespace, add a description, and give the Web Service a name other than the name of the class.
<code>WebMethod</code> attribute	Yes	Add to public methods you want to expose. May also add attributes to specify caching, enable session state, enable transactions etc.
Inherit from <code>WebService</code> class	No	Gives direct access to intrinsic objects such as the <code>Application</code> object.

So now that we have seen how we can create Web Services using ASP.NET, let us see how we can turn around and consume Web Services from ASP.NET.