msdn magazine

THE XML FILES

WS-Security, WebMethods, Generating ASP.NET Web Service Classes

Aaron Skonnard

Download the code for this article: XMLFiles0209.exe (49KB)

Q Lately I've been hearing a lot about WS-Security. What is it, and how is it different from other security standards?

A WS-Security is a new Web Services security specification authored by Microsoft, IBM, and VeriSign. It has generated significant industry-wide interest since it was announced at Tech•Ed 2002. The fundamental difference between WS-Security and current Web-related security standards (SSL, HTTP basic/digest authentication, and so on) is that WS-Security provides a framework for building security information into the SOAP message as opposed to the channel or application protocol.

SOAP provides a simple XML messaging framework for building your own higher-level application protocols or APIs. The SOAP framework consists of three main elements: Envelope, Header, and Body, all of which are defined in the SOAP namespace (http://schemas.xmlsoap.org/soap/envelope/). These elements allow you to frame messages in a standard way that facilitates extensibility and negotiation. The following code shows a typical SOAP Envelope and highlights where the payload and headers go:

```
<soap:Envelope>
  <xmlns:soap="...">
    <soap:Header>
    <!-- standardized, pluggable headers -->
    </soap:Header>
    <soap:Body>
    <!-- payload -->
    </soap:Body>
</soap:Envelope>
```

The SOAP framework also provides standard error representation for informing clients about exceptions. When an error occurs, the SOAP endpoint is required to return a soap:Fault in the Body describing what went wrong.

In addition to this simple messaging framework, the SOAP specification also defines a set of encoding rules for common data structures used in programming languages (mostly a stopgap measure since XML Schema wasn't around at the time), and a standard binding to HTTP and remote procedure calls (RPC). But now that the XML Schema spec is finished, the SOAP encoding rules are considered deprecated and many developers are interested in Web Services over protocols using messaging patterns other than typical RPC (request/response).

As SOAP has shaken out over the last few years, the remaining pillar that really matters is the fundamental XML messaging framework for communication. In this context, however, SOAP doesn't actually provide much value beyond a central rendezvous point for communications. The real added value will come from standardized SOAP headers that Web services can use to negotiate higher-level services like security.

The Microsoft® Global XML Web Services Architecture (GXA) is an initiative to define some of the standard SOAP headers that the industry needs to build higher-level application protocols and services. This is where WS-Security comes in. WS-Security is one of several GXA specifications designed specifically for the SOAP framework.

WS-Security is completely standards-based and extensible, and it provides most of the same security features of other frameworks (like SSL) such as message authentication, message integrity, and message confidentiality. The difference again is that these features are implemented at the SOAP message level. The key benefit of message-level security is that it provides an end-to-end (versus point-to-point) security solution. With WS-Security, security information can span multiple transport or application protocols for a given request (see **Figure 1**). This is crucial in environments where multiple intermediaries (actors), or SOAP routers, are used to do work along the way.

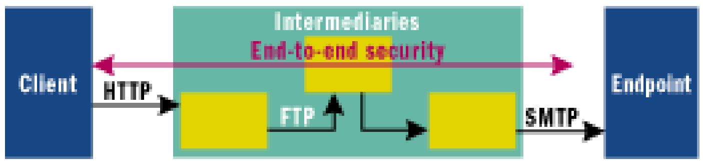


Figure 1 Security Across Protocols

WS-Security utilizes existing security standards like X.509 certificates, Kerberos, XML Signature, and XML Encryption to accomplish all of this. WS-Security is extensible enough, however, to support multiple security tokens for authentication and authorization, multiple trust domains, and multiple encryption technologies. In other words, WS-Security is really a security framework that allows applications to choose the right security technology for the job. It does not define things like how trust is actually established or how keys are exchanged, which are considered outside the scope of the spec.

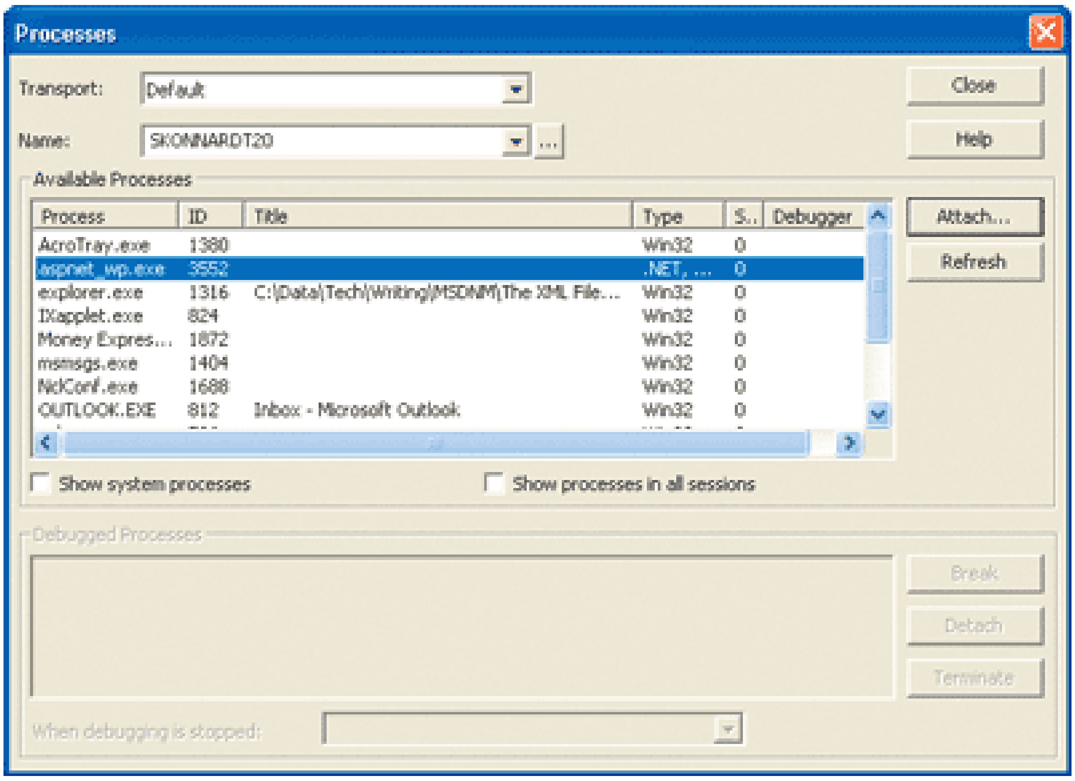
To see something a little more concrete, take a look at a few sample SOAP messages that use WS-Security. The envelope shown in **Figure 2** sends basic user credentials. The code in **Figure 3** uses WS-Security to include an X.509 certificate in the message.

The specification contains several other examples that illustrate how to sign messages for integrity through XML Signature and how to encrypt portions of a SOAP message through XML Encryption. For more information, check out the specification at Web Services Security.

Q How do you debug an ASP.NET WebMethod?

A To debug an ASP.NET WebMethod, first browse to the .asmx file to make sure the ASP.NET application is loaded.

Then, select Processes from the Debug menu and you should see the window shown in **Figure 4**.


Processes

Transport: Default

Name: SKONNARDT20

Available Processes

Process	ID	Title	Type	S..	Debugger
AcroTray.exe	1380		Win32	0	
aspnet_wp.exe	3552		.NET, ...	0	
explorer.exe	1316	C:\Data\Tech\Writing\MSDNM\The XML File...	Win32	0	
IXapplet.exe	824		Win32	0	
Money Expres...	1872		Win32	0	
msmsgs.exe	1404		Win32	0	
NclConf.exe	1688		Win32	0	
OUTLOOK.EXE	812	Inbox - Microsoft Outlook	Win32	0	

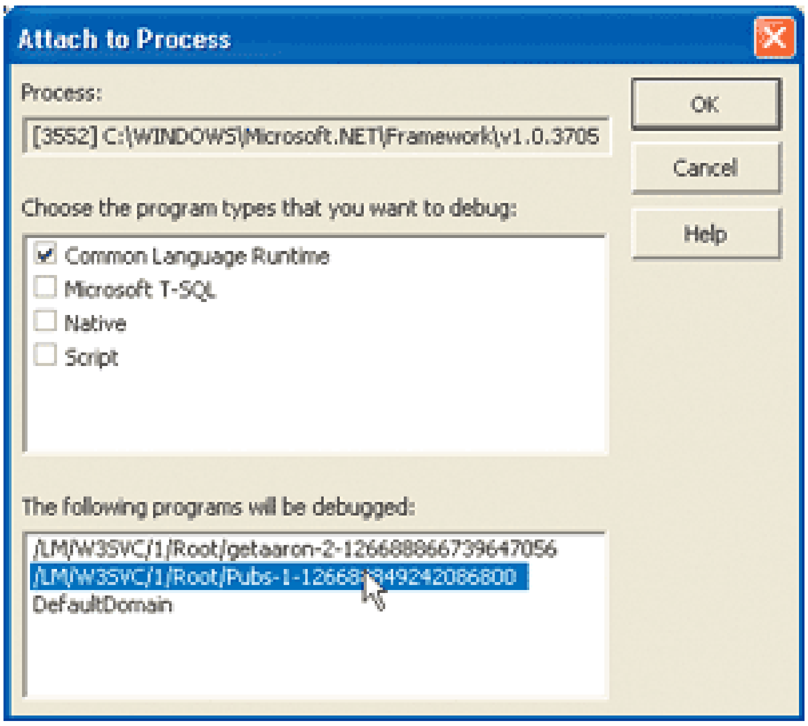
☐ Show system processes ☐ Show processes in all sessions

Debugged Processes

When debugging is stopped:

Figure 4 Processes Window

Next, select aspnet_wp.exe from the list of available processes and click on the Attach button. This should bring up the window shown in **Figure 5**. Find the name of your ASP.NET application in the list of programs at the bottom and press OK. Now, set a breakpoint in your WebMethod and issue either a GET or POST request to the .asmx page. You should hit your breakpoint.


Attach to Process

Process:

[3552] C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705

Choose the program types that you want to debug:

- ☑ Common Language Runtime
- ☐ Microsoft T-SQL
- ☐ Native
- ☐ Script

The following programs will be debugged:

/LM/W3SVC/1/Root/getaaron-2-126688866739647056
/LM/W3SVC/1/Root/Pubs-1-12668?849242086800
DefaultDomain

Figure 5 Attach

Q If I try to return an object graph containing a cycle from an ASP.NET WebMethod, an exception occurs. Why?

A By default, ASP.NET WebMethods use the document-literal style binding for all operations. This means that the input and output messages must be describable using the XML Schema for use in the exposed Web Services Description Language (WSDL) document. Unfortunately, XML Schema doesn't formalize how to describe complex object graph representations, like multiple references to the same object or, in this case, a cycle.

The SOAP section 5 encoding rules did provide built-in support for these more complicated graph scenarios. It was easier for SOAP to deal with this issue because it wasn't defining an entire description language, but rather a simple set of transformation rules. The SOAP encoding uses a combination of id and href attributes to give objects identity (id) and represent references to objects (href). So if you really need to return something like a cycle, use the SOAP encoding for that particular WebMethod by using the SoapRpcMethod attribute, as shown in **Figure 6**. In this case, the SOAP response will look like that in **Figure 7**.

In general, if you care about interoperability, it's better to avoid these situations altogether and design simpler wire-level interfaces that can easily be described with XML Schema.

Q Is it possible to generate an ASP.NET Web Service class from a WSDL document? Is this approach better than simply starting with a class and letting ASP.NET generate the WSDL?

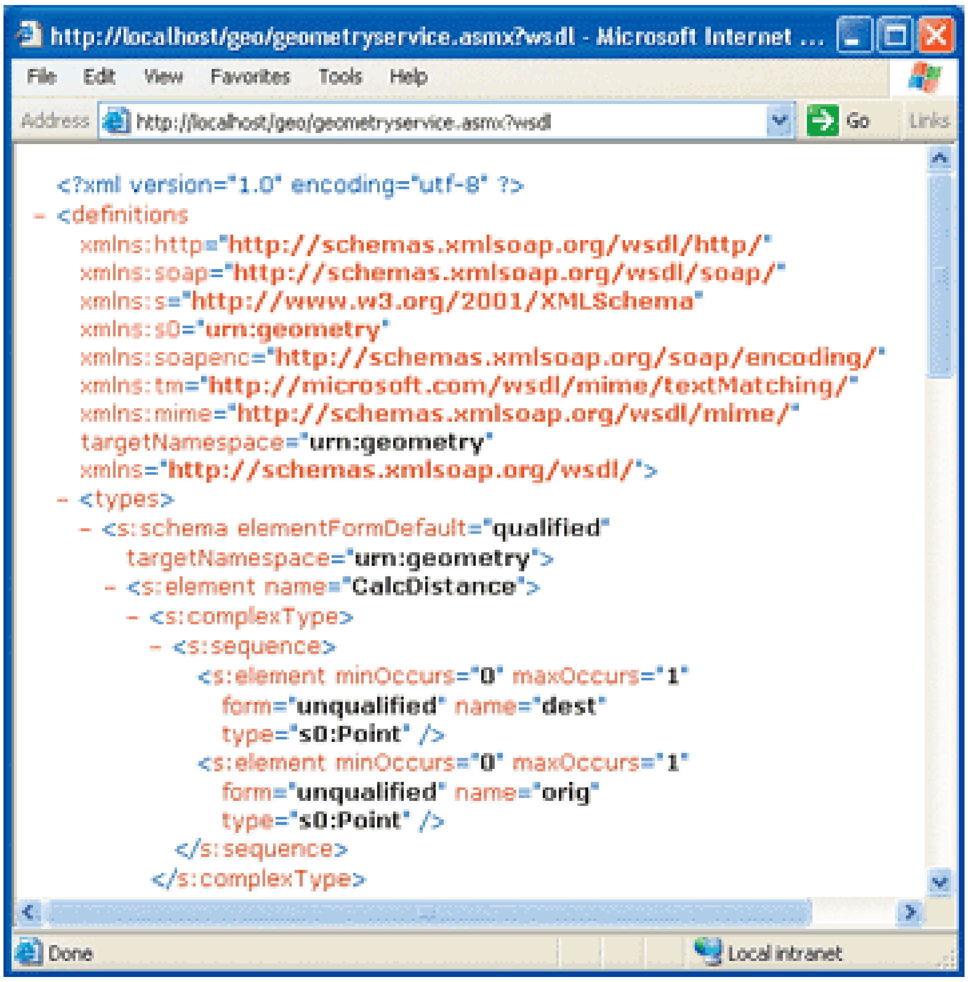
A Yes, you can generate ASP.NET Web Service classes from WSDL. This is a better approach if you care about maintainability and versioning. It's not for the faint of heart, though, since you have to write the WSDL by hand. First I'll explain how ASP.NET generates WSDL documents from class definitions, then I'll show you how to reverse the process.

When you build Web Services using C# or Visual Basic® .NET classes via the ASP.NET WebMethod infrastructure, the runtime has enough information to automatically generate a WSDL document that accurately describes the Web Service. Suppose you have the following GeometryService.asmx file:

```
<%@ WebService language="C#"
     class="GeometryService" %>
using System;
using System.Web.Services;

public class Point {
  public double x;
  public double y;
}
public class GeometryService {
  [WebMethod]
  public double CalcDistance(Point orig,
    Point dest) {
    return Math.Sqrt(Math.Pow(dest.x-
      orig.x, 2) +
      Math.Pow(dest.y-orig.y, 2));
  }
}
```

Issuing an HTTP request against GeometryService.asmx with the ?wsdl query string (http://localhost/geometry/GeometryService.asmx?wsdl) instructs the .asmx handler to dynamically generate a WSDL document (see **Figure 8**).


http://localhost/geo/geometryservice.asmx?wsdl - Microsoft Internet ...
File
Edit
View
Favorites
Tools
Help
Address
http://localhost/geo/geometryservice.asmx?wsdl
Go
Links
Done
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <definitions
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:s0="urn:geometry"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    targetNamespace="urn:geometry"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <types>
    - <s:schema elementFormDefault="qualified"
        targetNamespace="urn:geometry">
      - <s:element name="CalcDistance">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="0" maxOccurs="1"
                form="unqualified" name="dest"
                type="s0:Point" />
              <s:element minOccurs="0" maxOccurs="1"
                form="unqualified" name="orig"
                type="s0:Point" />
            </s:sequence>
          </s:complexType>
```

Figure 8 Dynamically Generated WSDL

By default, ASP.NET WebMethods are described using document-literal bindings in WSDL. This means that literal XML Schema definitions describe the SOAP Body payload for both the request and response messages. The details of the generated XML Schema definitions are derived from the System.Xml.Serialization type mapping rules.

There are a slew of customization attributes found in System.Xml.Serialization and System.Web.Services.Protocols that can be used to control the relationship between common language runtime (CLR) methods marked with [WebMethod] and their corresponding WSDL constructs. To use these attributes, you need to be somewhat familiar with WSDL concepts.

Clients that want to interact with this Web Service can request the WSDL document and determine how to generate appropriate invocation messages as well as how to process the expected response messages. Some clients may even use a tool to generate client-side proxies that map Web Service invocation to a more familiar class-based programming model on the client. .NET provides such a utility, called wsdl.exe, that can be used from the command line:

```
wsdl.exe http://localhost/geometry/GeometryService.asmx?wsdl
```

This will generate a client-side proxy class called GeometryService that can be used as shown here:

```
GeometryService gs = new GeometryService();
Point o = new Point();
Point d = new Point();
o.x = 33.3; o.y = 66.6;
d.x = 66.6; d.y = 33.3;
// invokes GeometryService.asmx and processes response
double dist = gs.CalcDistance(o, d);
```

The fundamental problem with this entire approach is that the server-side programming model hides the true interface, the WSDL contract, lulling developers into believing that Web Services are just methods on a class. The fact that a Web Service operation can map to a method on a C# class is an implementation detail that the other end of the wire knows nothing about. The true interface to a Web Service operation is the wire-level XML contract. This misconception can lead to fragile systems, versioning nightmares, and a dependency on the infrastructure that weakens your ability to diagnose problems. As an example, making subtle changes to the server-side class will break clients that are hardcoded against existing proxies (like the one generated in the previous code snippet) if the WSDL contract changes. It's far from obvious what changes might make the WSDL contract change.

An alternative to this approach is to take manual control of the WSDL definition. This allows you to focus first on defining the interface, which can not only generate client proxies but also serve as a starting point for server-side implementation stubs (see **Figure 9**). Since the WSDL specifies all the details of the wire-level contract, there's no reason plumbing can't exist to automatically generate the correct ASP.NET Web Service class that uses all of the appropriate customization attributes mentioned earlier.

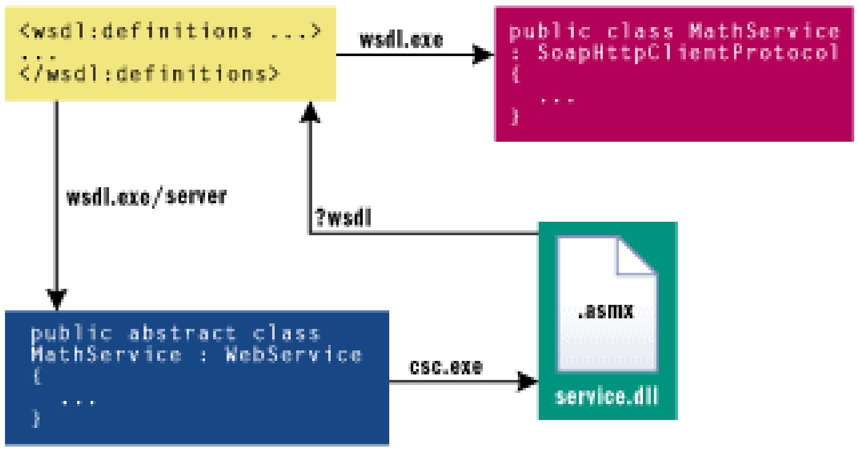


Figure 9 Using WSDL.exe

Wsdl.exe provides this behavior through the /server switch:

```
wsdl.exe /server /out:GeometryServiceBase.cs GeoService.wsdl
```

This generates a file named GeometryServiceBase.cs that contains a class derived from System.Web.Services.WebService. The details of this class depend entirely on the WSDL document.

The first thing you'll notice about the generated class is that it's littered with customization attributes that map the

WSDL constructs onto the generated class and its contained methods. This is great, since you don't have to know anything about the customization attributes, just the underlying WSDL constructs, and you still get to work with simple method definitions. In general, it's a better use of your time to learn WSDL than to learn platform-specific details since you can take the knowledge of the specification with you anywhere.

You'll also notice that the class is marked as abstract, which forces you to derive a new class from it to implement the methods. You should define this class in a file separate from the one that was generated. This was meant to protect you from accidentally losing changes when you regenerate the stub.

When you derive your new class, make sure that you copy all of the customization attributes from the generated class to the derived class (including those on each method). The customization attributes are not inherited by design. Overlooking this step will break the WSDL contract that you started with. Then to finish off the implementation, you simply need to create an .asmx file with a single ASP.NET directive referencing your new class, like so:

```
<%@ WebService language="C#" class="GeometryServiceImpl" %>
```

Now, if you issue an HTTP GET request against this .asmx file, you should get a WSDL document that's equivalent to the one you started with, although it's not the same static file. (Last month I told you how to return the static WSDL document you started with.)

This approach gives you complete control over the WSDL document and forces you to think of the Web Service contract before deciding the implementation details. This is not a new idea. In fact, it's just like every other RPC technology of the past, such as COM and IDL. Granted, there is nothing stopping you from accidentally changing the contract by modifying aspects of the implementation class, but the distinction between interface and implementation makes it less likely to happen. An additional benefit is that this approach protects you from using CLR types in your public interfaces that can't be expressed in XML Schema (since you're defining the interface using XML Schema). Overall, you're better off managing the WSDL yourself, but it does require some effort.

For a complete example of an ASP.NET Web Service implementation generated from WSDL, see **Figure 10**. You can download the complete sample project at the link at the top of this article.

Send your questions and comments for Aaron to xmlfiles@microsoft.com.

Aaron Skonnardis an instructor/researcher at DevelopMentor, where he develops the XML and Web Service-related curriculum. Aaron coauthored *Essential XML Quick Reference* (Addison-Wesley, 2001) and *Essential XML* (Addison-Wesley, 2000). Reach him at http://staff.develop.com/aarons.

From the September 2002 issue of MSDN Magazine.
Get it at your local newsstand, or better yet, *subscribe*.

Figure 2 Sending Basic Credentials

```
<S:Envelope
```

```
  xmlns:S="http://www.w3.org/2001/12/soap-envelope"
  xmlns:ws="http://schemas.xmlsoap.org/ws/2002/04/secext">
    <S:Header>
        <ws:Security>
            <ws:UsernameToken>
                <ws:Username>aarons</ws:Username>
                <ws:Password>snoraa</ws:Password>
            </ws:UsernameToken>
        </wsse:Security>
        •••
    </S:Header>
    •••
</S:Envelope>
```

Figure 3 Sending an X.509 Certificate

```
<S:Envelope
 xmlns:S="http://www.w3.org/2001/12/soap-envelope"
 xmlns:ws="http://schemas.xmlsoap.org/ws/2002/04/secext">
    <S:Header>
            •••
          <wsse:BinarySecurityToken
             Id="myToken"
             ValueType="wsse:X509v3"
             EncodingType="wsse:Base64Binary">
             MIIEZzCCA9CgAwIBAgIQEmtJZc0...
          </wsse:BinarySecurityToken>
            •••
    </S:Header>
    •••
</S:Envelope>
```

Figure 6 Using SoapRpcMethod

```
using System.Web.Services;
using System.Web.Services.Protocols;

public class Person
{
  public string name;
  public Person spouse;
}
public class MyService
{
  [WebMethod]
  [SoapRpcMethod] // remove this line and kaboom!
  public GetAaron()
  {
    Person aaron = new Person();
    aaron.name = "Aaron";
    Person monica = new Person();
    monica.name = "Monica";
    aaron.spouse = monica;
    monica.spouse = aaron;
    return aaron;
  }
}
```

Figure 7 SOAP Response

```
...
<soap:Body ...>
  <tns:GetAaronResponse>
    <GetAaronResult href="#id1" />
  </tns:GetAaronResponse>
  <types:Person id="id1" xsi:type="types:Person">
    <name xsi:type="xsd:string">Aaron</name>
    <spouse href="#id2" />
  </types:Person>
  <types:Person id="id2" xsi:type="types:Person">
    <name xsi:type="xsd:string">Monica</name>
    <spouse href="#id1" />
  </types:Person>
</soap:Body>
...
```

Figure 10 ASP.NET Web Service from WSDL

GeometryService.wsdl

```
<!-- written by Aaron Skonnard -->
<wsdl:definitions
    targetNamespace="urn:geometry"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="urn:geometry"
>
    <wsdl:types>
        <xs:schema targetNamespace="urn:geometry"
          elementFormDefault="unqualified" xmlns:tns="urn:geometry"
          xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <xs:complexType name="Point">
                <xs:sequence>
                    <xs:element name="x" type="xs:double" />
                    <xs:element name="y" type="xs:double" />
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="CalcDistance">
                <xs:sequence>
                    <xs:element name="dest" type="tns:Point" />
                    <xs:element name="orig" type="tns:Point" />
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="CalcDistanceResponse">
                <xs:sequence>
                    <xs:element name="return" type="xs:double" />
                </xs:sequence>
            </xs:complexType>
            <xs:element name="CalcDistance" type="tns:CalcDistance" />
            <xs:element name="CalcDistanceResponse"
              type="tns:CalcDistanceResponse" />
        </xs:schema>
    </wsdl:types>

    <wsdl:message name="CalcDistanceRequestMsg">
        <wsdl:part name="parameters" element="tns:CalcDistance"/>
    </wsdl:message>
    <wsdl:message name="CalcDistanceResponseMsg">
        <wsdl:part name="parameters" element="tns:CalcDistanceResponse"/>
    </wsdl:message>

    <wsdl:portType name="IGeometry">
        <wsdl:operation name="CalcDistance">
            <wsdl:input message="tns:CalcDistanceRequestMsg"/>
            <wsdl:output message="tns:CalcDistanceResponseMsg"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="GeometrySoapBinding" type="tns:IGeometry">
        <soap:binding style="document" transport="http://
          schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="CalcDistance">
            <soap:operation soapAction="urn:geometry#CalcDistance"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="GeometryService">
        <wsdl:port name="GeometryServiceSoapPort"
          binding="tns:GeometrySoapBinding">
            <soap:address location="http://localhost/GeometryService/
              GeometryService.asmx"/>
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

GeometryServiceBase.cs

```
//———————————————————————————————————————
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.3705.0
//
//     Changes to this file may cause incorrect behavior and will be lost
//     if the code is regenerated.
// </autogenerated>
//———————————————————————————————————————
```

```
//
// This source code was auto-generated by wsdl, Version=1.0.3705.0.
//
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;

/// <remarks/>
[System.Web.Services.WebServiceBindingAttribute(Name="GeometrySoapBinding",
  Namespace="urn:geometry")]
public abstract class GeometryService : System.Web.Services.WebService {

    /// <remarks/>
    [System.Web.Services.WebMethodAttribute()]
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute
      ("urn:geometry#CalcDistance", RequestNamespace="urn:geometry",
      ResponseNamespace="urn:geometry",
      Use=System.Web.Services.Description.SoapBindingUse.Literal,
      ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    [return: System.Xml.Serialization.XmlElementAttribute("return",
      Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public abstract System.Double
      CalcDistance([System.Xml.Serialization.XmlElementAttribute
      (Form=System.Xml.Schema.XmlSchemaForm.Unqualified)] Point dest,
      [System.Xml.Serialization.XmlElementAttribute
      (Form=System.Xml.Schema.XmlSchemaForm.Unqualified)] Point orig);
}

/// <remarks/>
[System.Xml.Serialization.XmlTypeAttribute(Namespace="urn:geometry")]
public class Point {

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute
      (Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public System.Double x;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute
      (Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public System.Double y;
}
```

GeometryService.asmx

```
<%@ WebService language="C#" class="GeometryServiceImpl"
  codeBehind="GeometryService.cs" %>
```

GeometryServiceImpl.cs

```
using System;

// my implementation of auto-generated abstract class
```

```
[System.Web.Services.WebService(Name="GeometryService",
  Namespace="urn:geometry")]
[System.Web.Services.WebServiceBindingAttribute(Name="GeometrySoapBinding",
  Namespace="urn:geometry", Location="GeometryService.wsdl")]
public class GeometryServiceImpl : GeometryService
{
    [System.Web.Services.WebMethodAttribute()]
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute
      ("urn:geometry#CalcDistance", Binding="GeometrySoapBinding",
        RequestNamespace="urn:geometry", ResponseNamespace="urn:geometry",
        Use=System.Web.Services.Description.SoapBindingUse.Literal,
        ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    [return: System.Xml.Serialization.XmlElementAttribute("return",
      Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
      public override double CalcDistance([System.Xml.Serialization.Xml
      ElementAttribute(Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
      Point dest, [System.Xml.Serialization.XmlElementAttribute
      (Form=System.Xml.Schema.XmlSchemaForm.Unqualified)] Point orig)
    {
        return Math.Sqrt(Math.Pow(dest.x-orig.x, 2) + Math.Pow
          (dest.y-orig.y, 2));
    }
}
```

©2005 Microsoft Corporation. All rights reserved. Terms of Use | Trademarks | Privacy Statement