ATL Server: High Performance C++ on .NET

PRANISH KUMAR,
JASJIT SINGH GREWAL,
BOGDAN CRIVAT,
AND ERIC LEE

Apress™

ISBN (pbk): 1-59059-128-3

Printed and bound in the United States of America 12345678910

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Manager and Copy Editor: Nicole LeClerc

Proofreader: Lori Bring

Compositor: Argosy Publishing

Indexer: Valerie Perry

Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 10

Web Services

WEB SERVICES **REPRESENT A** new philosophy in application development and design. Based on standard protocols, Web services introduce a way of allowing applications to take advantage of Internet communication.

Web services enable applications to take advantage of the Internet by allowing them to make procedural calls and exchange data over the Web. By relying on Extensible Markup Language (XML) as the packaging, Web services allow communications between programs running all over the world, regardless of the underlying platform.

ATL Server has been designed to allow native C++ developers to easily create Web services on the Windows platform. The ATL Server Web service support has been extensively tested against Web services created using other tools and technologies (ASP.NET, Apache, and others) to ensure ease of use and interoperability.

In this chapter you'll examine what exactly Web services are, how they work, and how you can use ATL Server to take advantage of them in your own applications (or to migrate existing components forward). This chapter assumes you are somewhat familiar with XML and XML namespaces.

Introducing Web Services

The two main protocols in the ATL Server implementation of Web services are Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL). SOAP is the protocol for a Web service message, and WSDL is for defining the interface for a service being called.

In this section you'll also look at Universal Description, Discovery, and Integration (UDDI), a publishing service that allows developers to publish their Web services so that other developers may discover and use them. In addition, UDDI allows developers looking for Web services to find and consume those that they have access to.

For more information on SOAP and WSDL, please visit the following Web pages on the World Wide Web Consortium's (W3C's) site:

- Simple Object Access Protocol (SOAP): `http://www.w3.org/TR/SOAP/`
- Web Services Description Language (WSDL): `http://www.w3.org/TR/WSDL.html`
- XML Schema Part 0: Primer: `http://www.w3.org/TR/xmlschema-0/`

- XML Schema Part 1: Structures: `http://www.w3.org/TR/xmlschema-1/`
- XML Schema Part 2: Datatypes: `http://www.w3.org/TR/xmlschema-2/`

SOAP

SOAP is a protocol for the exchange of information in a distributed environment, which is achieved by the exchange of SOAP "messages." You'll see some examples of SOAP messages later in this section.

For many C++ developers, it's probably convenient to consider SOAP as a style of remote procedure call (RPC) using HTTP as the transport and XML as the data format or packaging. Although this definition is an oversimplification, it should provide a good sense of context. It's important to note, however, that neither SOAP nor the ATL Server SOAP support is limited to HTTP as the transport mechanism. Later on, you'll see how users can plug in their own transport mechanisms into the ATL Server SOAP support.

A SOAP message is an XML document with predefined elements that may have user-defined data as subelements. The basic format of a SOAP message as described in section 4 of the SOAP specification is as follows:

```
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Header>
    <!-- user data -->
  </SOAP:Header>
  <SOAP:Body>
    <!-- user data -->
  </SOAP:Body>
</SOAP:Envelope>
```

The `Header` element of the SOAP message is optional. The `Envelope` and `Body` elements are required in the SOAP message. You can find a full description of the SOAP message format in section 4 of the SOAP specification. Here's an example of a simple SOAP message:

```
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <m:HelloWorld xmlns:m="Some-URI">
    <inputString>StringValue</inputString>
    </m:HelloWorld>
  </SOAP:Body>
</SOAP:Envelope>
```

Notice that the optional `Header` element has been omitted. The user data under the `Body` element is one possible encoding of a `Hello World` SOAP message. When viewed as an RPC message, the element `HelloWorld` is a wrapper element under `Body` that denotes the function name. `inputString` is a parameter to the function `HelloWorld` and has the value `StringValue`.

With that, let's continue on to the other major protocol in ATL Server Web services: WSDL.

WSDL

WSDL is an XML format for describing network services as a series of endpoints containing either document-oriented or procedure-oriented information. WSDL isn't specific to SOAP, but it has a predefined syntax for describing SOAP messages. For those familiar with COM, it may be convenient to think of WSDL as a Web service version of Interface Definition Language (IDL). Again, this is an oversimplification that should help provide some context.

A WSDL document is an XML document, and it uses XML Schemas to describe the format of the messages. (Extensible Schema Definition, or XSD, is described in the specifications you can find at the Web addresses we presented at the start of this section.) Listing 10-1 presents an example of a simple WSDL document.

Listing 10-1. A Simple WSDL Document

```
<?xml version="1.0"?>
<!-- ATL Server generated Web Service Description -->
<definitions
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:s0="http://mynamespace"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:atls="http://tempuri.org/vc/atl/server/"
  targetNamespace="http://mynamespace"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
>
  <types>
    <s:schema targetNamespace=http://mynamespace
 attributeFormDefault="qualified" elementFormDefault="qualified">
      <s:simpleType name="MyEnumeration">
        <s:restriction base="s:string">
```

```
            <s:enumeration value="Value1"/>
            <s:enumeration value="Value2"/>
            <s:enumeration value="Value3"/>
          </s:restriction>
        </s:simpleType>
        <s:complexType name="MyStruct">
          <s:sequence>
            <s:element name="EnumValue" type="s0:MyEnumeration"/>
            <s:element name="UIntValue" type="s:unsignedInt"/>
          </s:sequence>
        </s:complexType>
      </s:schema>
    </types>
    <message name="MyMethodIn">
      <part name="Parameter1" type="s0:MyStruct"/>
    </message>
    <message name="MyMethodOut">
      <part name="return" type="s0:MyStruct"/>
    </message>
    <portType name="MyServiceSoap">
      <operation name="MyMethod">
        <input message="s0:MyMethodIn"/>
        <output message="s0:MyMethodOut"/>
      </operation>
    </portType>
    <binding name="MyServiceSoap" type="s0:MyServiceSoap">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
      <operation name="MyMethod">
        <soap:operation soapAction="#MyMethod" style="rpc"/>
        <input>
          <soap:body use="encoded" namespace="http://mynamespace"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </input>
        <output>
          <soap:body use="encoded" namespace="http://mynamespace"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>
      </operation>
    </binding>
```

```
  <service name="MyService">
    <port name="MyServiceSoap" binding="s0:MyServiceSoap">
      <soap:address location="http://localhost/MyService.dll?Handler=MyService"/>
    </port>
  </service>
</definitions>
```

All WSDL documents begin with a `definitions` tag, which may also contain XML namespace declarations. The `definitions` tag is followed by the `types` tag, which in turn may contain `schema` tags, which contain XSD type definitions. In Listing 10-1, the `schema` element contains a `simpleType` element and a `complexType` element. The `simpleType` element in Listing 10-1 is used to define an enumeration named `MyEnumeration`, with the values `Value1`, `Value2`, and `Value3`. You may also use the `simpleType` element to extend or restrict other "primitive" types (e.g., to restrict the range of an `unsignedInt`). The `complexType` element in Listing 10-1 is used to define a struct named `MyStruct`, with the field `EnumValue`, which is of type `MyEnumeration`, and `UIntValue`, which is of type `unsignedInt`. You may also use the `complexType` element to define arrays, SOAP messages, and other types and elements. The `types` section is followed by a series of `message` elements, which are used to define the contents of a SOAP message. The `message` elements contain message `part` elements, which reference XSD types. The `message` elements are followed by one or more `portType` elements, which compose the individual messages into *operations,* which form a complete SOAP interaction (request/response). The `portType` element is in turn followed by a `binding` element, which references a `portType` element and its `operations` to `bind` the operations to specific SOAP protocols, namespaces, encoding styles, and so on. The final element is the `service` element, which references a `binding` element and provides a specific location URL at which the service can be invoked.

This is a very high-level view of WSDL. Listing 10-1 is an example of a WSDL document that is produced by the default configuration and settings of ATL Server Web services. A WSDL document will vary depending on the type of messages that are being passed, the encoding style, the transport mechanism, and other factors. WSDL provides an extensibility method that allows for custom elements to be inserted into WSDL documents. For more details on WSDL, consult the specifications noted previously.

Creating a Web Service

In this section you'll learn what's involved in creating a Web service and how ATL Server helps simplify many of the tasks involved in this process for you.

Creating a Web Service by Hand

To create a Web service by hand (without using any libraries or frameworks), you have to complete a number of tasks. The following steps outline the tasks you normally need to perform to correctly handle the reception of a SOAP message:

1. Determine the intended recipient of the message (e.g., the `HelloWorld` function).

2. Parse the XML of the message and marshal the parameters (e.g., `inputString`) into real C++ data types (e.g., a string).

3. Call the intended recipient of the message (`HelloWorld`) with the expected parameters (`inputString`).

4. After the function, take the output parameters and return value and generate a SOAP HTTP response message to send back to the client.

This doesn't even account for a situation where you intend the Web service to be callable from any client, in which case WSDL for the Web service also needs to be generated.

As you can clearly see, there's a lot of code involved in creating a Web service infrastructure, and most of this code actually needs to be duplicated for every exposed function. All of these issues must be resolved before you can focus on implementing your Web service functionality. Plus, all of this work only enables you to create Web service services—it doesn't include the infrastructure code required to create a Web service client!

Creating a Web Service with ATL Server

ATL Server is designed to solve the problems mentioned in the preceding section for you and make creating a Web service easy. ATL Server does this by allowing you to focus on implementing your application logic and not on the underlying infrastructure code.

Using the ATL Server Web Service Wizard

From the Visual Studio .NET New Project dialog box, choose the Visual C++ Projects folder, and then choose the ATL Server Web Service project.

You'll notice that the wizard dialog box that appears is nearly identical to that of the ATL Server Project Wizard described earlier. In fact, it's the same wizard, just

with different default settings. The Application Options tab has the Create as Web Service box checked by default. Almost all the options and settings described earlier for ATL Server are available for ATL Server Web services. A few options are unavailable (they're grayed out). In the Application Options tab, the Validation Support and Stencil Processing Support options aren't available, because those options apply only to ATL Server projects that will handle Web page requests (i.e., SRF-based pages). Similarly, the locale and codepage options are unavailable because Web services don't use SRF files. On the Developer Support Options tab, the Attributed Code check box can't be unchecked, because ATL Server Web services require the use of attributes.

The ATL Server Web Service Wizard will generate the following Web service files (assuming that the project's name is MyProject):

- *MyProject.h:* Contains your Web service implementation
- *MyProject.disco:* The Web service .disco file
- *MyProject.htm:* A description of the Web service

The default Web service generated by the wizard is a simple "Hello World"–style Web service that shows how a basic ATL Server Web service works:

```
[
  uuid("989438E7-DC64-4C1E-9B7D-18AE00BA8EE2"),
  object
]
__interface IMyProjectService
{
  // HelloWorld is a sample ATL Server Web service method. It shows how to
  // declare a Web service method and its in-parameters and out-parameters
  [id(1)] HRESULT HelloWorld([in] BSTR bstrInput, [out, retval] BSTR *bstrOutput);
  // TODO: Add additional Web service methods here
};
```

First, embedded IDL is used to declare a COM interface that describes the Web service.

Through attributes, embedded IDL is now available to all COM developers and Web service developers. The COM-like syntax was chosen for a few very specific reasons. The benefit of choosing a COM-like syntax is that it enables COM developers to protect their existing investment. In terms of a similar coding style, it helps COM developers protect their existing investment in their knowledge of COM development. In terms of similar code (embedded IDL), it enables COM developers to easily protect their investment in existing code, making it easy for

these developers to expose existing or new COM objects as Web services if they desire (with just a few lines of code).

The IDL attributes are used to specify the parameters of the Web service methods being exposed. In the wizard-generated code example, the `HelloWorld` method has two parameters, both of which are BSTRs: `bstrInput` and `bstrOutput` (in the next section we describe Web service types and their mappings to XSD). The IDL `in` attribute is used to specify that the `bstrInput` parameter is a part of the SOAP *request*, and the IDL `out` attribute is used to specify that the `bstrOutput` parameter is part of the SOAP *response* (in the next section we describe all the SOAP attributes in detail). Listing 10-2 shows the sample "Hello World" Web service.

Listing 10-2. "Hello World" Web Service

```
[
  request_handler(name="Default", sdl="GenProject1WSDL"),
  soap_handler(
    name="MyProjectService",
    namespace="urn:MyProjectService",
    protocol="soap"
  )
]
class CMyProjectService : public IMyProjectService
{
public:
// This is a sample Web service method that shows how to use the
// soap_method attribute to expose a method as a Web method
  [ soap_method ]
  HRESULT HelloWorld(/*[in]*/ BSTR bstrInput, /*[out, retval]*/ BSTR *bstrOutput)
  {
    CComBSTR bstrOut(L"Hello ");
    bstrOut += bstrInput;
    bstrOut += L"!";
    *bstrOutput = bstrOut.Detach();
    return S_OK;
  }
// TODO: Add additional Web service methods here
}; // class CMyProjectService
```

The `CMyProjectService` class implements the Web service described by the interface we examined previously.

The `request_handler` attribute (described earlier in the context of normal ATL Server Web applications) now has the additional `sdl` parameter, which specifies the handler name for retrieving the WSDL for the Web service. The `soap_handler` attribute specifies that the request handler is also a SOAP Web service (i.e., it will contain methods that will need to be able to decode incoming SOAP and encode outgoing messages as SOAP). The class inherits from the `IMyProjectService` interface and implements the `HelloWorld` method.

The `soap_method` attribute specifies which methods from the `IMyProjectService` interface are to be exposed via SOAP. The `HelloWorld` method is implemented as any COM method would be implemented, without any special processing required as a Web service method.

Consuming ATL Server Web Services

In this section we describe how to consume ATL Server Web services using Visual Studio .NET's Add Web Reference dialog box. You can use the options on this dialog box to consume any kind of Web service that exposes a WSDL description. The dialog box generates a proxy class that you can use to invoke a method on the Web service by simply calling the matching method in the proxy class. The dialog box generates an ATL Server native C++ proxy for C++ projects (in 2002), in 2003 managed C++ projects generate managed C++ proxies (native remains ATL Server).

In Solution Explorer, right-click the project to which you want to add the Web service proxy class and choose Add Web Reference. Enter the location of the Web service's WSDL for the address, for example, **http://localhost/MyProjectService/MyProjectService.dll?Handler=GetWSDL**. The Add Web Reference dialog box lists the WSDL that you've selected. You have the option of viewing the documentation for the Web service (if it exists). Click the Add Reference button to generate the proxy class. For example, if you're adding a Web reference to a Visual C++ project, an ATL Server proxy class will be generated using the sproxy.exe tool. The Build Output window should appear and show something like the following:

```
------ Build started: Project: Proxy1, Configuration: Debug Win32
Creating Web service proxy file...
/out:MyProjectService.h
Build log was saved at "file://c:\Code\Proxy\Debug\BuildLog.htm"
Proxy1 - 0 error(s), 0 warning(s)
---------------------- Done ----------------------
    Build: 1 succeeded, 0 failed, 0 skipped
```

As the output suggests, the Web service proxy class is generated in the file MyProjectService.h.

The generated proxy class will have methods that map to the methods in the Web service. In this example, there will be a `HelloWorld` method in the proxy class that can be called to invoke the `HelloWorld` method in the Web service:

```
HRESULT HelloWorld(
  BSTR bstrInput,
  BSTR* __retval
);
```

To invoke the method, create an instance of the proxy class and simply call the method as you would normally:

```
#include "MyProjectService.h"
int main()
{
  CoInitialize(NULL);
  {
    MyProjectService::CMyProjectService svc;
    CComBSTR bstrOut;
    svc.HelloWorld(CComBSTR(L"Joe"), &bstrOut);
    printf("%ws\n", bstrOut);
    return 0;
  }
  CoUninitialize();
}
```

When you run your program, it will output `Hello Joe!`.

You've covered the basics of creating and consuming ATL Server Web services in Visual Studio .NET. Now it's time to become more familiar with using ATL Server.

Using ATL Server Web Services

In this section you'll learn how to use ATL Server to create Web services. You'll begin by looking at the architecture of an ATL Server Web service and then see how a Web service request hooks into the ATL Server architecture.

Next, you'll investigate the SAX XML parser used to parse the XML of the incoming SOAP requests/responses. You'll move on to implement ATL Server Web services using attributes. You'll then see how common types are supported by ATL Server.

ATL Server Web Service Architecture

From ATL Server's point of view, a Web service is simply another request handler. The SOAP details are handled in the implementation of the request handler itself, which in turn dispatches to the user code once the XML has been marshaled to C++ types and marshals the returned C++ types back into XML for the response. Because the Web service is just another request handler, it can take advantage of all the services that a regular request handler can. It has access to all services that live in the ISAPI extension, and it can provide its own services as well. Figure 10-1 illustrates the lifetime of a typical request.

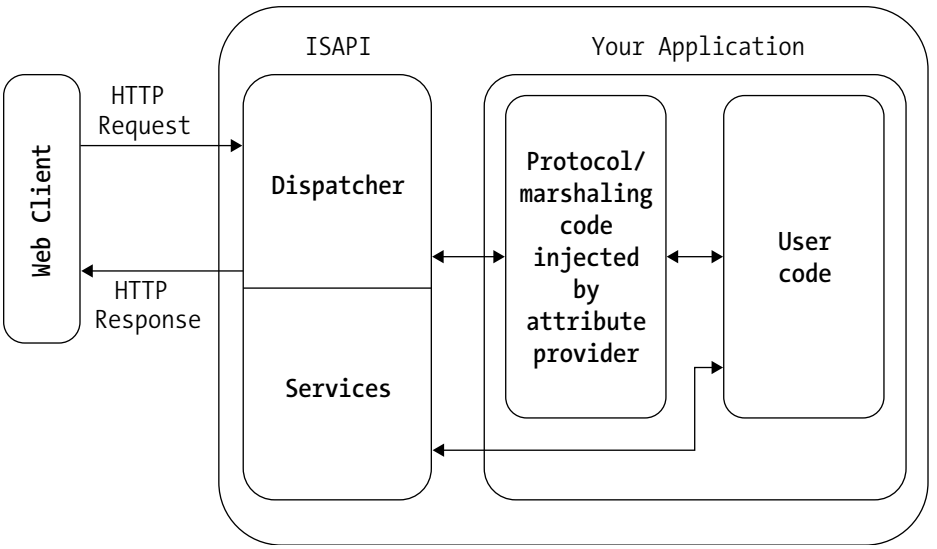

Figure 10-1. Application architecture for an ATL Server SOAP application

SAX

SOAP is based on XML, and the XML must be parsed somehow. ATL Server uses SAX to parse XML. The primary reasons SAX was chosen are performance and scalability. SAX is considerably faster than MSXML DOM for parsing, and it's considerably more lightweight as well—SAX uses significantly less memory and significantly fewer allocations than MSXML DOM.

Implementing Web Services

In the following sections you'll explore the various aspects of implementing ATL Server Web services. First, we cover the Visual C++ attributes that are provided by ATL Server for Web services. Then we cover the various types supported by ATL Server and how they map to XSD. We also describe how user-defined types, such as structs and enums, can be used with ATL Server. Finally, we cover how to use arrays in the Web service.

ATL Server Web Service Attributes

Just as ATL Server provides attributes to simplify the task of creating Web applications, it provides attributes to simplify the task of creating Web services. One important difference, however, is that while it's possible to create Web applications without the use of the `request_handler` and `tag_name` attributes, ATL Server Web services *require* the use of attributes. (Technically, it's possible to create Web services without attributes, because it's always possible to view the injected code; however, this won't be portable to future versions of ATL Server.)

ATL Server provides the following attributes for creating Web services: `soap_handler`, `soap_header`, and `soap_method`. Also, the `sdl` parameter of the `request_handler` attribute applies only when it's used in conjunction with the `soap_handler` attribute.

soap_handler

The `soap_handler` attribute applies to classes, and it designates that the class will handle SOAP requests. This ensures that the code to handle the marshaling of the XML gets injected by the attribute provider. Additionally, a different base class will be injected from the one that's normally injected by the `request_handler` attribute. The `soap_handler` attribute can appear only once per class.

The `soap_handler` attribute has five parameters: `name`, `namespace`, `protocol`, `style`, and `use`.

- `name`: This is the user-provided name for the Web service. This name appears in the WSDL as the name of the Web service and is used by sproxy.exe in generating the Web service proxy class. If this parameter isn't specified, the name of the class is used.

- `namespace`: This parameter is the user-provided namespace for the Web service, the XML namespace to which all user-defined types and methods will belong. It's the namespace that will be used to validate incoming SOAP messages. If this parameter isn't specified, the XML namespace will be based upon the name of the class. For example, if the class name is `CMyWebService`, the namespace will be `urn:CMyWebService`. Developers should choose a specific, unique namespace that properly distinguishes their Web service.

- `protocol`: This is a reserved parameter in the version of ATL Server that ships as part of Visual Studio .NET. The only permissible value is `soap`. In the future, other Web service protocols or extended SOAP protocols may be supported.

- `style`: This is the SOAP "style" to be used for the format of the SOAP messages. The permissible values are `rpc` and `document`; `rpc` is the default value. This parameter describes whether the SOAP messages are intended as remote procedure calls or XML documents.

- `use`: This is the SOAP "use" for the SOAP messages. The permissible values are `encoded` and `literal`; `encoded` is the default value. The `use` parameter indicates whether the SOAP messages are to be encoded using SOAP section 5 encoding rules, or whether they're describing the concrete XML schema of the message.

In the version of ATL Server that ships with Visual Studio .NET, the only permissible style/use combinations are `rpc/encoded` and `document/literal`.

CAUTION *When you use* `document/literal`*, multidimensional arrays aren't supported as SOAP headers or as parameters on a method exposed via SOAP.*

When the `soap_handler` attribute is used in conjunction with the `request_handler` attribute, its `sdl` parameter may be used to specify the handler name used to retrieve the WSDL for the Web service. If the handler name isn't specified, it defaults to a value based on the class name. For example, if the class name is `CMyWebService`, the `sdl` parameter defaults to `GenCMyWebServiceWSDL`.

soap_method

The `soap_method` attribute applies to methods, and it designates that the method on which it appears will be exposed via SOAP. The `soap_method` attribute can appear only once per method.

The `soap_method` attribute has one parameter: `name`.

- `name`: This parameter is the user-specified name for the exposed method. This is the name that will be used in the WSDL and that clients will need to use when invoking the Web service. If this parameter isn't specified, the name of the method will be used.

CAUTION *The method on which this attribute is placed must be an implementation of an interface method defined in an embedded IDL interface; otherwise, it will result in a compiler error. The reason for this is that unless the method is an interface method, the ATL Server framework can't determine which parameters are* `in` *parameters, which are* `out` *parameters, and if a return value is specified.*

soap_header

The `soap_header` attribute applies to methods, and it indicates that the method on which it appears will have the specified SOAP header in its SOAP message. Headers must be member variables. The `soap_header` attribute is optional and may appear one or more times per method. The `soap_header` attribute may appear only on methods that also have the `soap_method` attribute, although the `soap_method` attribute doesn't require the `soap_header` attribute.

The `soap_header` attribute has four parameters: `value`, `required`, `in`, and `out`.

- `value`: This is the name of the member variable that's being sent or received as a header. Variable size arrays may not be used as SOAP headers. The user must specify this parameter. There's no default value.

- `required`: This is a boolean parameter that indicates whether the specified header is optional or not. The default value for this parameter is `false`. If this parameter is set to `true`, the header will be sent as a SOAP `mustUnderstand` header, and the WSDL will indicate that the client should send the header as

a `mustUnderstand` header. If a required header isn't received, ATL Server will return a SOAP fault.

- `in`: This is a boolean parameter that indicates whether the specified header is expected as part of a request. The default value is `true`.

- `out`: This is a boolean parameter that indicates whether the specified header should be sent as part of the response of the method. The default value is `true`.

CAUTION *When a* `soap_header` *attribute appears on a method, each instance must have a unique "value" parameter. That is, the same header can't appear as more than one SOAP header for a particular method.*

Types

In this section you'll look at the types supported by the ATL Server implementation of Web services. You'll see how C++ data types are mapped to XSD data types.

ATL Server supports all native C++ data types. It also defines a special type, `ATLSOAP_BLOB`, that's used to send binary data over SOAP. Table 10-1 shows the native types that are supported, along with their corresponding XSD data type mappings.

Table 10-1. ATL Server Data Types

C++ DATA TYPE	XSD DATA TYPE
`bool`	`Boolean`
`char`	`Byte`
`_int8`	`Byte`
`unsigned char`	`unsignedByte`
`unsigned _int8`	`unsignedByte`
`short`	`Short`

Table 10-1. ATL Server Data Types (Continued)

C++ DATA TYPE	XSD DATA TYPE
_int16	Short
unsigned short	unsignedShort
unsigned_int16	unsignedShort
wchar_t	unsignedShort
Int	int
_int32	int
long	int
unsigned int	unsignedInt
unsigned_int32	unsignedInt
unsigned long	unsignedInt
_int64	long
unsigned_int64	unsignedLong
double	double
float	dloat
BSTR	string
ATLSOAP_BLOB	base64Binary

ATL Server currently has no way to represent XSD types not listed in Table 10-1. You may use `typedef`s in place of direct references to native types. You must take care to ensure that the `typedef`s have the expected results. For example, in Visual C++ .NET, BSTR is the only type that ATL Server will map to string. If another string type is used, for example `LPCSTR`, it will be mapped to `const char *`, which ATL Server will attempt to map to an array of bytes, which isn't an efficient way to represent strings in SOAP. ATL Server will treat all pointer types as arrays. In these cases, the user is required to specify the size of the array by using the `size_is` attribute. The upcoming section on arrays describes arrays in detail.

ATL Server supports user-defined structs and enums, which we describe in detail in later sections of this chapter. ATL Server doesn't support unions in Visual C++ .NET. ATL Server doesn't support templatized types or template instantiations as SOAP types.

Arrays

ATL Server supports arrays in two forms: fixed-size arrays and variable-sized arrays. The arrays can contain any primitive or user-defined type that's supported by ATL Server. Fixed-size arrays are arrays of the form

```
int arr[5];
BSTR arr[2][3];
```

Variable-sized arrays are of the form

```
int *arr;
```

When you use variable-sized arrays, you must specify the size of the array with the `size_is` attribute. This is required to ensure that the array is marshaled correctly and safely. The `size_is` attribute references a parameter or struct field that specifies the size of the array. When the array is an `in`-only parameter, the `size_is` attribute is optional. If it appears for `in`-only parameters, the parameter specified in the `size_is` attribute will contain the number of array elements sent by client. The parameter referenced by the `size_is` attribute must have the same "in" and "out" attributes as the array to which it is applied. So a `size_is` parameter for an `out` array must also be an `out` parameter, a `size_is` parameter for an `in` array must also be an `in` parameter, and a `size_is` parameter for an `in/out` array must also be an `in/out` parameter. Listing 10-3 shows an example of this.

Listing 10-3. Sample Web Service Using Variable-Sized Arrays

```
[ uuid("643cd054-24b0-4f34-b4a1-642519836fe8"), object ]
__interface IRetArray
{
  [id(1)] HRESULT retArray(
[out] int *nSize,
[out, retval,
size_is(*nSize)] int **arrOut);
};
[
  request_handler(name="Default", sdl="retArraySDL"),
  soap_handler(name="RetArray",
  namespace="http://retArray ",
  protocol="soap")
]
class CRetArray : public IRetArray
{
public:
```

```
  [ soap_method ]
  HRESULT retArray(int *nSize, int **arrOut)
  {
    *nMax = 10;
    *arrOut = (int *)GetMemMgr()->Allocate(*nMax*sizeof(int));
    for (int i=0; i<*nMax; i++)
      (*arrOut)[i] = i;
    return S_OK;
  }
};
```

The `size_is` attribute appears in the IDL definition on the `arrOut` parameter and references the `nSize` parameter (we explain the call to `GetMemMgr()->Allocate` in detail in the section "Memory Management").

ATL Server doesn't support variable-length arrays of more than one dimension.

Fixed-size arrays don't require a `size_is` attribute, because the size of the array is part of its type. Fixed-size may also be multidimensional. Listing 10-4 shows an example of a Web service that uses multidimensional arrays.

Listing 10-4. Sample Web Service Using Multidimensional Arrays

```
[ uuid("643cd054-24b0-4f34-b4a1-642519836fe8"), object ]
__interface IRetArray
{
[id(1)] HRESULT retArray([out] int arrOut[3][3]);
};
[
  request_handler(name="Default", sdl="retArraySDL"),
  soap_handler(name="RetArray",
  namespace="http://retArray ",
  protocol="soap")
]
class CRetArray : public IRetArray
{
public:
  [ soap_method ]
  HRESULT retArray(int arrOut[3][3])
  {
    for (int i=0; i<3; i++)
   {
      for (int j=0; j<3; j++)
      {
        arrOut[i][j] = i*3+j;
```

```
      }
    }
  return S_OK;
  }
};
```

Structs

ATL Server supports user-defined structs. For most common structs, simply define and use the struct as you normally would. Listing 10-5 shows a sample Web service that uses structs.

Listing 10-5. Sample Web Service Using Structs

```
[ export ]
struct MyStruct
{
  BSTR strValue;
  int nValue;
};

// IStructService - Web service interface declaration
//
[
  uuid("4EA08537-12F7-4DC7-ABE5-483CFE0F4FE0"),
  object
]
__interface IStructService
{
  [id(1)] HRESULT StructTest([in] MyStruct tIn, [out, retval] MyStruct *tOut);
};

// StructService - Web service implementation
//
[
  request_handler(name="Default", sdl="GenStructWSDL"),
  soap_handler(
    name="StructService",
    namespace="urn:StructService",
    protocol="soap"
  )
]
class CStructService :
  public IStructService
```

```
{
public:
  [ soap_method ]
  HRESULT StructTest(/*[in]*/ MyStruct tIn, /*[out, retval]*/ MyStruct *tOut)
  {
    tOut->strValue = SysAllocString(tIn.strValue);
    tOut->nValue = tIn.nValue;
    return S_OK;
  }
}; // class CStructService
```

The `export` attribute is only necessary if you plan for your Web service to also be used as a COM object. If you don't plan on using your Web service as a COM object, it's completely harmless to leave it on your struct definition.

Structs can contain fields of nearly any type, including nested struct and enum fields. They can also contain array fields. You can use fixed-size arrays just as you would normally. When you use variable-length arrays, however, you must specify the array size, just as you do when you use a variable-length array as a parameter. Listing 10-6 shows a sample Web service that uses a struct that contains a variable-sized array.

Listing 10-6. Sample Web Service Using a Struct That Contains a Variable-Sized Array

```
[ export ]
struct MyStruct
{
  [size_is(nSize)] int *arr;
  int nSize;
};

// IStructService - Web service interface declaration
//
[
  uuid("4EA08537-12F7-4DC7-ABE5-483CFE0F4FE0"),
  object
]
__interface IStructService
{
  [id(1)] HRESULT StructTest([in] MyStruct tIn, [out, retval] MyStruct *tOut);
};


// StructService - Web service implementation
```

```
//
[
  request_handler(name="Default", sdl="GenStructWSDL"),
  soap_handler(
    name="StructService",
    namespace="urn:StructService",
    protocol="soap"
  )
]
class CStructService :
  public IStructService
{
public:
  [ soap_method ]
  HRESULT StructTest(/*[in]*/ MyStruct tIn, /*[out, retval]*/ MyStruct *tOut)
  {
    // set the size of the array
    // tIn.nSize will contain the number of array elements marshaled
    tOut->nSize = tIn.nSize;
    tOut->arr =
reinterpret_cast<int *>(GetMemMgr()->Allocate(
    tIn.nSize*sizeof(int)));
    if (!tOut->arr)
    {
      return E_OUTOFMEMORY;
    }
    for (int i=0; i<tIn.nSize; i++)
    {
      tOut->arr[i] = tIn.arr[i];
    }
    return S_OK;
  }
}; // class CStructService
```

Note that `tIn.nSize` will contain the number of array elements marshaled in, independent of the value that's sent in the client request. Thus, a malicious user can't spoof the number of array elements, which could otherwise result in walking past the end of an array. The `nSize` field of `tOut` will tell the ATL Server framework how many array elements to marshal back to the user.

Enums

ATL Server supports enums. You can use enums in ATL Server exactly as you would normally. Listing 10-7 shows a sample Web service that uses enums.

Listing 10-7. Sample Web Service Using Enums

```
[ export ]
enum MyEnum { Value1, Value2, Value3, Value4 };

// IEnumService - Web service interface declaration
//
[
  uuid("A745E7CB-AD49-41EB-B36C-D533B812EC64"),
  object
]
__interface IEnumService
{
  [id(1)] HRESULT TestEnum([in] MyEnum eIn, [out, retval] MyEnum *eOut);
};


// EnumService - Web service implementation
//
[
  request_handler(name="Default", sdl="GenEnumWSDL"),
  soap_handler(
    name="EnumService",
    namespace="urn:EnumService",
    protocol="soap"
  )
]
class CEnumService :
  public IEnumService
{
public:

  [ soap_method ]
  HRESULT TestEnum(/*[in]*/ MyEnum eIn, /*[out, retval]*/ MyEnum *eOut)
  {
    if (eIn == Value4)
    {
      *eOut = Value1;
    }
```

```
    else
    {
      *eOut = (MyEnum)(eIn+1);
    }
    return S_OK;
  }
}; // class CEnumService
```

Again, the `export` attribute on the enum declaration is only necessary when you plan for your Web service to also be used as a COM object. If you don't plan on using your Web service as a COM object, it's completely harmless to leave it on your enum definition.

BLOB Types

ATL Server supports BLOBs through the framework-defined `ATLSOAP_BLOB` struct. The definition of `ATLSOAP_BLOB` is as follows:

```
[ export ]
typedef struct _tagATLSOAP_BLOB
{
  unsigned long size;
  unsigned char *data;
} ATLSOAP_BLOB;
```

The `data` field is the raw bytes contained in the BLOB, and the `size` field indicates the number of bytes in the data field. The `ATLSOAP_BLOB` type maps to the XSD `base64Binary` type, hence the data is base64-encoded before being put on the wire. The memory for the `data` field is allocated in the same way arrays are allocated. Listing 10-8 shows a sample Web service using the `ATLSOAP_BLOB` type.

Listing 10-8. Sample Web Service Using ATLSOAP_BLOB

```
[
  uuid("41AF710A-EC7B-4FD5-B1C4-CBB58406AEF8"),
  object
]
__interface IBlobService
{
  [id(1)] HRESULT BlobTest(
[in] ATLSOAP_BLOB blobIn,
[out, retval] ATLSOAP_BLOB *blobOut);
};
```

```
// BlobService - Web service implementation
//
[
  request_handler(name="Default", sdl="GenBlobWSDL"),
  soap_handler(
    name="BlobService",
    namespace="urn:BlobService",
    protocol="soap"
  )
]
class CBlobService :
  public IBlobService
{
public:
  [ soap_method ]
  HRESULT BlobTest(
/*[in]*/ ATLSOAP_BLOB blobIn,
/*[out, retval]*/ ATLSOAP_BLOB *blobOut)
  {
    blobOut->size = blobIn.size;
    blobOut->data =
reinterpret_cast<unsigned char *>(GetMemMgr()->Allocate(
    blobIn.size));
    memcpy(blobOut->data, blobIn.data, blobIn.size);
    return S_OK;
  }
}; // class CBlobService
```

When sending binary data, you need to use the `ATLSOAP_BLOB` struct; otherwise, ATL Server won't treat the data as binary and won't perform the proper encodings to ensure correct transport.

Restricted Types

The only type restriction that ATL Server has is the use of variable-length multidimensional arrays. For example, the code in Listing 10-9 will result in a compiler error.

Listing 10-9. Illegal Input Header

```
[
  uuid("23E070EF-C8B5-4A0F-A299-FB50ABD6CD03"),
  object
]
__interface IRestrictedTypesService
{
  [id(1)] HRESULT Illegal([in] BSTR **arrInput);
};

 [
  request_handler(name="Default", sdl="GenRestrictedTypesWSDL"),
  soap_handler(
    name="RestrictedTypesService",
    namespace="urn:RestrictedTypesService",
    protocol="soap"
  )
]
class CRestrictedTypesService :
  public IRestrictedTypesService
{
public:

  [ soap_method ]
  HRESULT Illegal(/*[in]*/ BSTR **arrInput)
  {
    arrInput;
    return S_OK;
  }
}; // class CRestrictedTypesService
```

Listing 10-9 will result in the following compiler error message:

```
error C2338: soap_method
        Atl Attribute Provider : error ATL2213: "arrInput" parameter of method
"Illegal" has too many indirections. In parameters cannot have more than 1
indirection.
```

Variable-length multidimensional arrays aren't allowed as `in` parameters, `out` parameters, struct fields, or SOAP headers. The restriction is due to implementation details relating to memory management.

SOAP Headers

In this section you'll look at how SOAP headers are defined and used. As described earlier, SOAP headers are defined using the `soap_header` attribute. SOAP headers designate member variables of the class to be sent or received as SOAP headers on a per-method basis. Listing 10-10 shows a sample Web service that uses SOAP headers.

Listing 10-10. Sample Web Service Using SOAP Headers

```
 [
  uuid("E8F59246-F4CB-4B8D-8FO9-1F8C79F5A825"),
  object
]
__interface IHeader1Service
{
  [id(1)] HRESULT HeaderMethod([out, retval] BSTR *ReturnValue);
};

 [
  request_handler(name="Default", sdl="GenHeader1WSDL"),
  soap_handler(
    name="Header1Service",
    namespace="urn:Header1Service",
    protocol="soap"
  )
]
class CHeader1Service :
  public IHeader1Service
{
public:

  BSTR HeaderValue;

  [ soap_method ]
  [ soap_header(value="HeaderValue", required=false, in=true, out=false) ]
  HRESULT HeaderMethod(/*[out, retval]*/ BSTR *ReturnValue)
  {
    if (HeaderValue != NULL)
    {
      *ReturnValue = SysAllocString(HeaderValue);
    }
```

```
    else
    {
      *ReturnValue = NULL;
    }

    return S_OK;
  }

}; // class CHeader1Service
```

In Listing 10-10, the `HeaderMethod` SOAP method declares that the `HeaderValue` member variable be used as a SOAP header for the method. The `soap_header` attribute's parameters declare that the header isn't a required header, which means that its absence won't result in an error; that the header is an `in` header, which means it's expected as part of the SOAP request packet; and that the header isn't an `out` header, which means that it won't be sent back to the client as part of the SOAP response packet. If the `required` parameter to the `soap_header` attribute is set to `true`, the header *must* be present if it's an `in` header. If the header isn't present, the ATL Server framework will return an error to the client. Required headers also impact the WSDL that's generated for the Web service by making the header a `mustUnderstand` header. Any `mustUnderstand` headers must be recognized by the SOAP processor; if they aren't, they're required to return an error.

SOAP headers must be public member variables. If a private or protected member variable is used, it will result in a compiler error. Again, this is due to implementation details of the ATL Server framework. In future versions, protected or private members might be permitted.

Any type that's supported by ATL Server may be used as a SOAP header, with the exception of variable-length arrays. ATL Server has no way to retrieve marshaling information about the size of the arrays as it can with the `size_is` attribute in IDL interface and struct definitions, hence it can't marshal and clean up the array. You may still use variable-length arrays inside of structs that are used as SOAP headers, however—you just can't use them directly as SOAP headers.

SOAP headers are automatically cleaned up by the ATL Server framework; however, users must initialize the values themselves, as they would with any other member variable. If custom cleanup is required for a SOAP header, users should override the `CleanupHeaders` function in their `soap_handler` class (see Chapter 19 for more details on custom handling).

SOAP Faults

In this section you'll look at how ATL Server handles SOAP faults. *SOAP faults* are the way Web services convey error and status information to a client. A SOAP fault is a special type of message, and it defines four subelements (see section 4.4 of the SOAP 1.1 specification): `faultcode`, `faultstring`, `faultactor`, and `detail`.

- `faultcode`: The `faultcode` element is intended to provide an algorithmic mechanism for identifying the fault. SOAP defines four default fault codes:
  - `VersionMismatch` means the SOAP processor found an invalid namespace for the SOAP envelope element.
  - `MustUnderstand` means that the SOAP processor encountered a header marked as `mustUnderstand`, which it didn't recognize.
  - `Client` means the client request is incorrect.
  - `Server` means that the error occurred on the server, rather than for some reason relating to the client request.
- `faultstring`: The `faultstring` element is intended to provide a human-readable description of the error.
- `faultactor`: The `faultactor` element is intended to provide information about who caused the fault within a message path.
- `detail`: The `detail` element is intended to provide application-specific error information.

ATL Server represents SOAP faults through the `CSoapFault` class, which has member variables to represent each of the preceding subelements. We describe how to retrieve fault information from the client later in the "ATL Server Web Service Client" section. For now, we'll explain how to return custom SOAP faults from the Web service.

ATL Server will automatically return faults for errors that occur while marshaling the SOAP request. This includes VersionMismatch, MustUnderstand, Server, and Client faults. Users can return custom SOAP faults by calling the `SoapFault()` function. Listing 10-11 shows a sample Web service that returns a custom SOAP fault.

Listing 10-11. Sample Web Service Using a Custom SOAP Fault

```
[
  uuid("31F3025O-D5BB-4O22-B6E2-CEA65EC7BO6D"),
  object
]
__interface IFault1Service
{
  [id(1)] HRESULT FaultTest([in] BSTR bstrInput);
};


 [
  request_handler(name="Default", sdl="GenFault1WSDL"),
  soap_handler(
    name="Fault1Service",
    namespace="urn:Fault1Service",
    protocol="soap"
  )
]
class CFault1Service :
  public IFault1Service
{
private:

  bool IsInvalidArg(BSTR bstrInput)
  {
    bstrInput;
    return true;
  }

public:
  [ soap_method ]
  HRESULT FaultTest(/*[in]*/ BSTR bstrInput)
  {
    if (IsInvalidArg(bstrInput))
    {
      SoapFault(SOAP_E_CLIENT, L"Invalid Argument", sizeof("Invalid Argument")-1);
      return E_INVALIDARG;
    }

    return S_OK;
  }
}; // class CFault1Service
```

In Listing 10-11, the FaultTest method checks the input to ensure it's a valid value; if it isn't, it returns a SOAP fault with a custom error message. Additionally, ATL Server will attempt to find an appropriate error message for an HRESULT error using the FormatMessage API. In Listing 10-11, FaultTest could have also returned E_INVALIDARG, and ATL Server would have loaded the appropriate error message using FormatMessage.

Users can also use the CSoapFault class directly by filling in the fields that represent the subelements and then calling the GenerateFault method with a class derived from IWriteStream. Listing 10-12 shows a sample Web service that uses the GenerateFault method to return a custom SOAP fault.

Listing 10-12. Sample Web Service Using the GenerateFault Method to Return a SOAP Fault

```
[
  uuid("2E55C132-0E5A-4EE9-9CAA-0B4824738D6B"),
  object
]
__interface IFault2Service
{
  [id(1)] HRESULT FaultTest([in] BSTR bstrInput);
};

 [
  request_handler(name="Default", sdl="GenFault2WSDL"),
  soap_handler(
    name="Fault2Service",
    namespace="urn:Fault2Service",
    protocol="soap"
  )
]
class CFault2Service :
  public IFault2Service
{
private:

  bool IsInvalidArg(BSTR bstrInput)
  {
    bstrInput;

    return true;
  }

public:
```

```
  [ soap_method ]
  HRESULT FaultTest(/*[in]*/ BSTR bstrInput)
  {
    if (IsInvalidArg(bstrInput))
    {
      CSoapFault fault;
      fault.m_soapErrCode = SOAP_E_CLIENT;
      fault.m_strDetail = L"Invalid Argument";
      fault.GenerateFault(m_pHttpResponse);

      return E_INVALIDARG;
    }
    return S_OK;
  }
}; // class CFault2Service
```

Memory Management

In this section you'll look at how memory is managed in an ATL Server Web service. In general, users will never have to free memory themselves, provided they allocate the memory in the way required by the ATL Server framework.

ATL Server follows COM rules with respect to memory allocation: `out` parameters must be NULL or must be able to be deallocated. For `in/out` parameters, users should free the memory before assigning into it; otherwise, the same rules as for `out` parameters apply. There are three cases when users will have to manage memory: when dealing with strings, when dealing with variable-length arrays, and when dealing with `ATLSOAP_BLOB`s.

When dealing with strings (BSTRs), you should allocate memory using `SysAllocString*` and free memory with `SysFreeString`. In other words, you allocate and free memory in the same way you would normally when dealing with BSTRs.

For variable-length arrays and `ATLSOAP_BLOB`s, you should allocate memory as explained in "Types" section previously, using the `IAtlMemMgr` interface that's returned from the `GetMemMgr()` method. The `IAtlMemMgr` interface is defined as follows:

```
__interface __declspec(uuid("654F7EF5-CFDF-4df9-A450-6C6A13C622C0")) IAtlMemMgr
{
public:
  void* Allocate( size_t nBytes ) throw();
  void Free( void* p ) throw();
  void* Reallocate( void* p, size_t nBytes ) throw();
  size_t GetSize( void* p ) throw();
};
```

By default, ATL Server uses a per-thread heap for its allocations. Users can provide their own `IAtlMemMgr` using the `SetMemMgr` method. After calling `SetMemMgr`, ATL Server will use the passed-in `IAtlMemMgr` for all its allocations, and it will also be returned from the `GetMemMgr()` method. Users should set a different memory manager when they are using asynchronous Web services (see Chapter 19 for more details).

The ATL Server framework will automatically handle the cleanup of memory after the processing of a Web service request. ATL Server Web service clients, however, will have to manage the memory after a Web service proxy method invocation themselves. We describe this process further in the next section.

ATL Server Web Service Clients

In this section you'll look at ATL Server Web service clients in more detail. You'll examine how they differ from ATL Server Web services and how they're similar. First, you'll look at the type support in ATL Server Web service clients.

Types

Table 10-2 shows how the ATL Server Web service proxy class generated by sproxy.exe maps XSD types to C++ data types.

Table 10-2. XML Data Type to C++ Data Type Mapping

XML SCHEMA DATA TYPE	C++ DATA TYPE (SPROXY)
boolean	bool
byte	char
unsignedByte	unsigned char
short	short
unsignedShort	unsigned short
int	int
unsignedInt	unsigned int
long	__int64
integer	__int64
nonPositiveInteger	__int64
negativeInteger	__int64

Table 10-2. XML Data Type to C++ Data Type Mapping (Continued)

XML SCHEMA DATA TYPE	C++ DATA TYPE (SPROXY)
unsignedLong	unsigned __int64
nonNegativeInteger	unsigned __int64
positiveInteger	unsigned __int64
decimal	double
double	double
float	float
string	BSTR
hexBinary	ATLSOAP_BLOB
base64Binary	ATLSOAP_BLOB
dateTime	BSTR
time	BSTR
date	BSTR
gMonth	BSTR
gYearMonth	BSTR
gYear	BSTR
gMonthDay	BSTR
gDay	BSTR
duration	BSTR
anyURI	BSTR
ENTITIES	BSTR
ENTITY	BSTR
ID	BSTR
IDREF	BSTR
IDREFS	BSTR
language	BSTR
Name	BSTR
NCName	BSTR
NMTOKEN	BSTR

Table 10-2. XML Data Type to C++ Data Type Mapping (Continued)

XML SCHEMA DATA TYPE	C++ DATA TYPE (SPROXY)
NMTOKENS	BSTR
normalizedString	BSTR
NOTATION	BSTR
QName	BSTR
token	BSTR

The mapping is the reverse of the mapping from C++ types to XSD types on the Web service side. Types that aren't directly supported, such as `normalizedString`, are represented as strings (BSTRs).

User-defined types, such as structs and enums, are also extracted from the XSD and appropriate definitions are emitted by sproxy.exe. When sproxy.exe emits the definition for a function that has a variable-length array as input or output, or when it encounters a struct that has a variable-length array field, it will emit a parameter or field that's used as the `size_is` for that array. The name of the parameter or field will be of the form `__[`*`parameter or field name`*`]_nSizeIs`. For `in` parameters, users are required to pass in the number of elements in the array so that the ATL Server framework knows how many elements to marshal. For `out` parameters, ATL Server will fill in this value with the number of elements that were marshaled. The `size_is` parameter/field will appear directly after the array parameter/field to which it applies in the function/struct definition. We present examples of this in the next section.

Memory Management

In this section you'll examine memory management in ATL Server Web service clients. Unlike ATL Server Web services, where ATL Server controls the full lifetime of the request and hence the data for the request, ATL Server controls neither the lifetime of the input parameter nor the output parameters on the client side. Users must manage much of the memory on the client side themselves; however, ATL Server provides several helper functions to make the job easier.

Strings on the client are managed just as they are on the server (i.e., using the `SysAllocString*` and `SysFreeString` functions). `CComBSTR` or `_bstr_t` can be used to simplify the task.

Arrays are allocated as they are on the server (i.e., using the proxy class's `GetMemMgr()` method to get the `IAtlMemMgr` interface and then invoking the `Allocate()` method to allocate the memory). Arrays can then be freed using `IAtlMemMgr`'s `Free()` method function provided by ATL Server. Listing 10-13 shows how array memory should be managed on the client.

Listing 10-13. Managing Array Memory on the Client

```
CWebServiceProxy proxy;
int *pArrInput = proxy.GetMemMgr()->Allocate(10*sizeof(int));
for (int i=0; i<10; i++)
{
  pArrInput[i] = i;
}
int *pArrOutput;
int nSize = 0;
HRESULT hr = proxy.EchoArray(pArrInput, 10, &pArrOutput, &nSize);
if (SUCCEEDED(hr))
{
  proxy.GetMemMgr()->Free(pArrOutput);
}
proxy.GetMemMgr()->Free(pArrInput);
```

Note the use of the `size_is` fields in Listing 10-13. The "10" represents the number of elements in the input array to marshal, and the `nSize` parameter is used to return the number of elements marshaled by the framework.

Structs can be cleaned up using the `AtlCleanupValueEx` template function. This function ensures that all struct fields, including strings, arrays, and nested structs, are cleaned up properly. Listing 10-14 shows how to manage struct memory on the client.

Listing 10-14. Managing Struct Memory on the Client

```
CWebServiceProxy proxy;
WebServiceStruct wsStruct;
wsStruct.s = SysAllocString(L"string");
wsStruct.arr = proxy.GetMemMgr()->Allocate(10*sizeof(int));
wsStruct.__arr_nSizeIs = 10;
for (int i=0; i<10; i++)
{
  wsStruct.arr[i] = i;
}
WebServiceStruct wsStructOut;
HRESULT hr = proxy.EchoStruct(wsStruct, &wsStructOut);
if (SUCCEEDED(hr))
{
  AtlCleanupValueEx(&wsStructOut, proxy.GetMemMgr());
}
AtlCleanupValueEx(&wsStruct, proxy.GetMemMgr());
```

Note the use of the `size_is` field for the preceding struct. This is used for the same marshaling purposes as in the previous array example.

Cleanup of arrays of structs can also be simplified using the `AtlCleanupArrayEx` or `AtlCleanupArrayMDEx` template function. The latter function handles cleanup of multidimensional arrays. Listing 10-15 shows how to manage the memory of arrays of structs on the client.

Listing 10-15. Cleaning Up Arrays of Structs

```
CWebServiceProxy proxy;
WebServiceStruct *pArrInput =
proxy.GetMemMgr()->Allocate(
    10*sizeof(WebServiceStruct));
for (int i=0; i<10; i++)
{
  pArrInput[i].s = SysAllocString(L"String");
  pArrInput[i].arr = proxy.GetMemMgr()->Allocate(10*sizeof(int));
  for (int j=0; j<10; j++)
  {
    pArrInput[i].arr[j] = j;
  }
pArrInput[i].__arr_nSizeIs = 10;
}
WebServiceStruct *pArrOutput;
int nSize = 0;
HRESULT hr = proxy.EchoStructArray(pArrInput, 10, &pArrOutput, &nSize);
if (SUCCEEDED(hr))
{
  AtlCleanupArrayEx(pArrOutput, nSize, proxy.GetMemMgr());
  proxy.GetMemMgr()->Free(pArrOutput);
}
AtlCleanupArrayEx(pArrOutput, 10, proxy.GetMemMgr());
proxy.GetMemMgr()->Free(pArrInput);
```

NOTE *Users must still free the top-level array manually.*

Multidimensional arrays can be cleaned up using the `AtlCleanupArrayMDEx` template function. The only difference between this function and the `AtlCleanupArrayEx` function is that instead of taking a count of the elements,

it takes an array containing information about the dimensions of the array. For example:

```
WebServiceStruct arrInput[2][3];
// Web service calls
int arrInputSize[] = {2, 2, 3};
AtlCleanpuArrayMDEx(&arrInput, arrInputSize, proxy.GetMemMgr());
```

The `arrInputSize` array indicates that this is a two-dimensional array (first element). The remaining elements describe each dimension's size.

Error Handling

In this section you'll look at how errors are reported and handled in ATL Server Web service clients.

When a SOAP request fails, the proxy class will set an error state that can be retrieved using the `GetClientError()` function. This function returns a `SOAPCLIENT_ERROR` enum value that describes the type of error. The enum is defined as follows:

```
// client error states
enum SOAPCLIENT_ERROR
{
  SOAPCLIENT_SUCCESS=0,           // everything succeeded
  SOAPCLIENT_INITIALIZE_ERROR,    // initialization failed - most
                                  // likely an MSXML installation
                                  // problem
  SOAPCLIENT_OUTOFMEMORY,         // out of memory
  SOAPCLIENT_GENERATE_ERROR,      // failed in generating the response
  SOAPCLIENT_CONNECT_ERROR,       // failed connecting to server
  SOAPCLIENT_SEND_ERROR,          // failed in sending message
  SOAPCLIENT_SERVER_ERROR,        // server error
  SOAPCLIENT_SOAPFAULT,           // a SOAP Fault was returned by the server
  SOAPCLIENT_PARSEFAULT_ERROR,    // failed in parsing SOAP fault
  SOAPCLIENT_READ_ERROR,          // failed in reading response
  SOAPCLIENT_PARSE_ERROR          // failed in parsing response
};
```

The errors are essentially as described in the comments next to the enum values. The most relevant error is probably the `SOAPCLIENT_SOAPFAULT` error. The `SOAPCLIENT_SOAPFAULT` error is returned when the Web service being invoked returns a SOAP fault. Information about the SOAP fault can be retrieved using the proxy class's `m_fault` member variable, which is a `CSoapFault`. The same fields that

we described in the earlier section on SOAP faults will be filled in according to the information returned by the Web service. Listing 10-16 shows how to retrieve error information from the proxy class.

Listing 10-16. Using SOAP Fault Information on the Client

```
HRESULT hr = proxy.WebMethod();
if (FAILED(hr))
{
  SOAPCLIENT_ERROR soapErr = proxy.GetClientError();
  switch(soapErr)
  {
    case SOAPCLIENT_INITIALIZE_ERROR :
      printf("initialization failed: check MSXML installation\n");
      break;
    case SOAPCLIENT_OUTOFMEMORY :
      printf("out of memory\n");
      break;
    case SOAPCLIENT_GENERATE_ERROR :
      printf("failed while generating request\n");
      break;
    case SOAPCLIENT_CONNECT_ERROR :
      printf("failed to connect to server\n");
      break;
    case SOAPCLIENT_SEND_ERROR :
      printf("failed while sending SOAP request\n");
      break;
    case SOAPCLIENT_SERVER_ERROR :
      printf("server error : %d\n", proxy.GetStatusCode());
      break;
    case SOAPCLIENT_PARSEFAULT_ERROR :
      printf("failed in parsing fault\n");
      break;
    case SOAPCLIENT_READ_ERROR :
      printf("failed while reading response\n");
      break;
    case SOAPCLIENT_PARSE_ERROR :
      printf("failed while parsing response\n");
      break;
    case SOAPCLIENT_SOAPFAULT :
      printf("SOAP Fault:\n"
                "fault code : %ws\n"
                "fault string : %ws\n"
                "fault detail : %ws\n",
```

```
                proxy.m_fault.m_strFaultCode,
                proxy.m_fault.m_strFaultString,
                proxy.m_fault.m_strDetail);
      break;
    default:
      printf("unknown error\n");
  }
}
```

Note the use of the proxy class's `GetStatusCode` method. This method retrieves the HTTP code that is returned by the server.

Conclusion

In this chapter you explored the basic uses and functionality of the ATL Server Web service support. You examined how to create Web services. You saw how to use user-defined types, such as structs and enums, in Web services. You also saw how arrays can be used within the ATL Server Web service framework. You learned how memory is managed in Web services and how to use and create SOAP faults from within a Web service. In addition, you examined the client side of all of these areas. You should now be able to create basic Web services using ATL Server.

One of the great things about ATL Server is that many of the classes are useful on applications other than Web applications and Web services. In the next chapter we examine how you can take advantage of some of the core ATL Server classes to solve problems you may face in many of your non-Web applications.