See This in MSDN Library

Web Services Description Language (WSDL) Explained

Carlos C. Tapang
Infotects

July 2001

Summary: Using WSDL, users can automate the generation of proxies for Web services in a truly language- and platform-independent way. (28 printed pages)

Page Options

Contents

Why WSDL?

Are standards like the Internet Protocol imposed by some authority, or do people recognize them as such because the benefits obtained far outweigh the cost of compliance? There have been many standards proposed that did not pan out. Sometimes, standards that do not become widely used are even enforced by law or government regulation: an example is the Ada programming language.

I believe it is the benefits obtained from following a standard that makes it widely accepted. So what matters for railroad services, for example, is that train tracks built by different companies come together, or that products from several companies work together. Several vendors have come together in order to establish SOAP as a standard. Web Services Description Language (WSDL) makes it easy to reap the benefits of SOAP by providing a way for Web service providers and users of such services to work together easily. It is easy for train tracks built by several firms to come together: all they have to agree on is the standard distance between the two rails. For Web services, it's much more complex. We first have to agree on a standard format for specifying interfaces.

It has been argued that SOAP does not really need an interface description language to go with it. If SOAP is a standard for communicating pure content, then it needs a language for describing that content. SOAP messages do carry type information, and so SOAP allows for dynamic determination of type. But I cannot call a function correctly unless I know its name and the number of parameters and the types of each. Without WSDL, I can determine the calling syntax from documentation that must be provided, or by examining wire messages. Either way, a human will have to be involved, and so the process is prone to error. With WSDL, I can automate the generation of proxies for Web services in a truly language- and platform-independent way. Like the IDL file for COM and CORBA, a WSDL file is a contract between client and server.

Note that while WSDL has been designed such that it can express bindings to protocols other than SOAP, our main concern here is WSDL as it relates to SOAP over HTTP. Also, while SOAP is currently used mainly for remote procedure or function calls, WSDL allows the specification of documents for transmission under SOAP. WSDL 1.1 has been submitted to W3C as a NOTE (see http://www.w3.org/TR/wsdl.html).

WSDL Document Structure

When trying to understand any XML document, it helps to have a block diagram. The diagram below illustrates the structure of WSDL, which is an XML document, by showing the relationships among the five sections that make up a WSDL document.

The WSDL document can be divided into two groups of sections. The top group is comprised of Abstract Definitions, while the bottom group consists of Concrete Descriptions. The abstract sections define SOAP messages in a platform- and language-independent manner; they do not contain any machine- nor language-specific elements. This helps define a set of services that several, diverse Web sites can implement. Site-specific matters such as serialization are then relegated to the bottom sections, which contain concrete descriptions.

- **Abstract Definitions**
  - **Types**
    Machine- and language-independent type definitions.
  - **Messages**
    Contains function parameters (inputs separate from outputs) or document descriptions.
  - **PortTypes**
    Refers to message definitions in Messages section to describe function signatures (operation name, input parameters, output pameters).
- **Concrete Descriptions**
  - **Bindings**
    Specifies binding(s) of each operation in the PortTypes section.
  - **Services**
    Specifies port address(es) of each binding.

In the diagram below, arrow connectors represent relationships among the different sections of the document. The dot and arrow connector represents a "refers to" or "uses" relationship. The double-arrow connector represents a "modifier" relationship. The 3-D arrow connector represents a "contains" relationship. Thus, the Messages section uses definitions in the Types section; the PortTypes section uses definitions in the Messages section; the Bindings section refers to the PortTypes section; and the Services section refers to the Bindings section. The PortTypes and Bindings sections contain operation elements, and the Services section contains port elements. Operation elements in the PortTypes section are modified or further described by operation elements in the Bindings section.

In this backgrounder I will be using standard XML terminology to describe the WSDL document. The word "element" refers to an XML element and the word "attribute" refers to an element attribute. Thus:

```
<element attribute="attribute-value">contents</element>
```

Contents may also be made up of one or more elements, in recursive fashion. The root element is the top-most element in which all other elements in a document belong. A child element always belongs to another, parent element.

Note that there can only be one Types section, or no Types section at all. All other sections can have zero, one, or multiple parent elements. For example, the Messages section can have zero or more <message> elements. The WSDL schema requires that all sections appear in a specific order: import, types, message, portType, binding, and service. Each abstract section may be in a separate file by itself and imported into the main document.

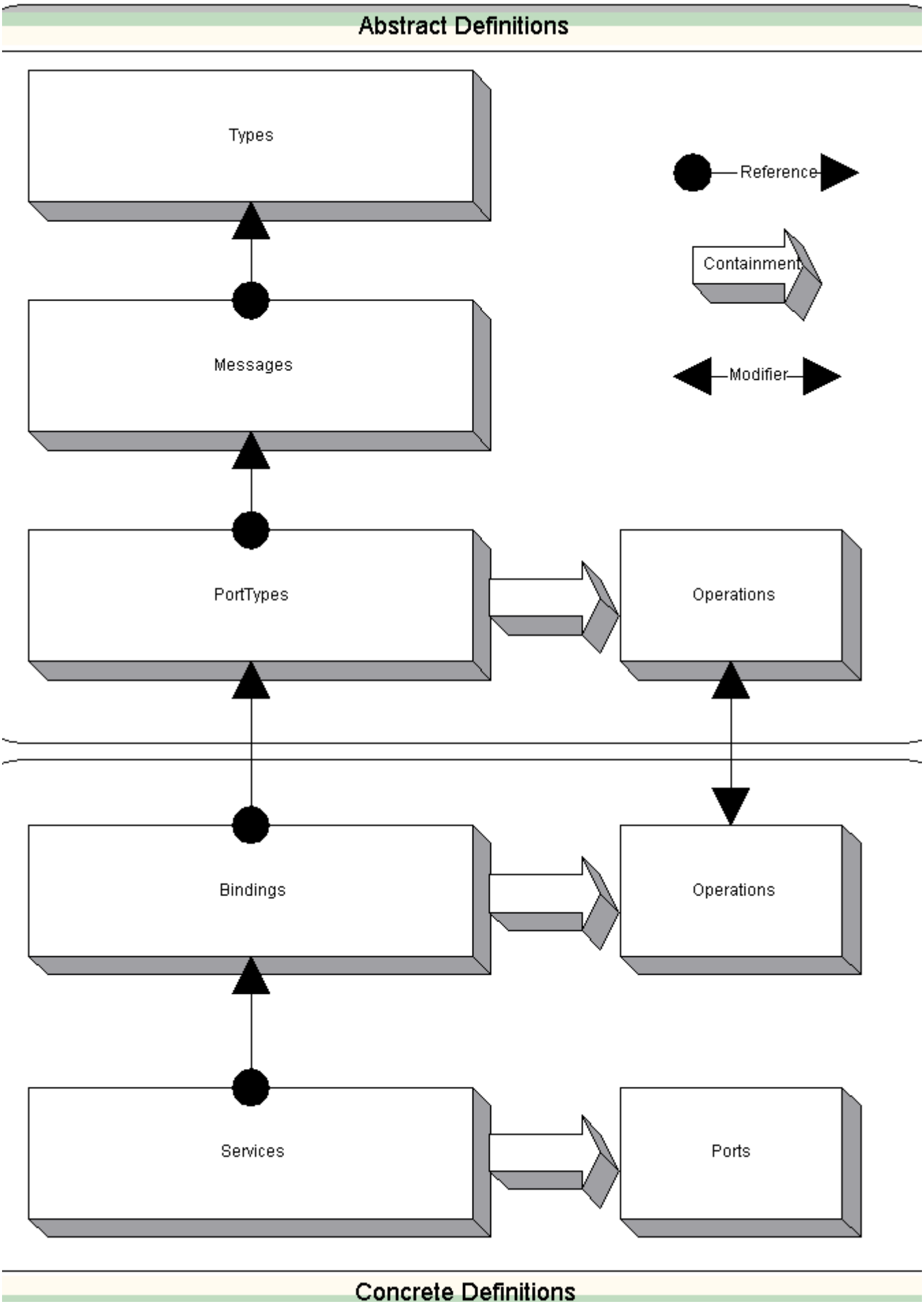
Abstract Definitions
Types
Reference
Containment
Messages
Modifier
PortTypes
Operations
Bindings
Operations
Services
Ports
Concrete Definitions

Figure 1. Abstract and concrete definitions

A Sample WSDL File

Let's dive right into a sample WSDL file to see its structure and how it works. Please be aware that this is a very simple instance of a WSDL document. Our purpose here is simply to illustrate its most salient features. The following sections include more detailed discussions.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="FooSample"
targetNamespace="http://tempuri.org/wsdl/"
xmlns:wsdlns="http://tempuri.org/wsdl/"
xmlns:typens="http://tempuri.org/xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:stk="http://schemas.microsoft.com/soap-toolkit/wsdl-extension"
xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
<schema targetNamespace="http://tempuri.org/xsd"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
elementFormDefault="qualified" >
</schema>
</types>

<message name="Simple.foo">
<part name="arg" type="xsd:int"/>
</message>

<message name="Simple.fooResponse">
<part name="result" type="xsd:int"/>
</message>

<portType name="SimplePortType">
<operation name="foo" parameterOrder="arg" >
<input message="wsdlns:Simple.foo"/>
<output message="wsdlns:Simple.fooResponse"/>
</operation>
</portType>

<binding name="SimpleBinding" type="wsdlns:SimplePortType">
<stk:binding preferredEncoding="UTF-8" />
<soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="foo">
<soap:operation
soapAction="http://tempuri.org/action/Simple.foo"/>
<input>
<soap:body use="encoded" namespace="http://tempuri.org/message/"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</input>
<output>
<soap:body use="encoded" namespace="http://tempuri.org/message/"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</output>
</operation>
</binding>

<service name="FOOSAMPLEService">
<port name="SimplePort" binding="wsdlns:SimpleBinding">
<soap:address location="http://carlos:8080/FooSample/FooSample.asp"/>
</port>
</service>
</definitions>
```

What follows is a bird's eye view of the sample document. Later, I will discuss each section in more detail.

The first line declares that the document is XML. Although it is not required, it helps the XML parser to determine whether to parse the WSDL file at all or signal an error. The second line is the root element in the WSDL document: <definitions>.

There are several namespace attributes (namespace declarations) attached to the root element, as well as in the <schema> child element of the <types> element.

The <types> element comprises the Types section. This section may be omitted if there are no data types that need to be declared. In the sample WSDL, there are no application-specific types declared but I use the Types section anyway just to declare schema namespaces used in the document.

The <message> elements comprise the Messages section. If we consider operations as functions, then a <message> element defines the parameters to that function. Each <part> child element in the <message> element corresponds to a parameter. Input parameters are defined in a single <message> element, separate from output parameters, which are in their own <message> element. Parameters that are both input and output have their corresponding <part> elements in both input and output <message> elements. The name of an output <message> element ends in "Response", as in "fooResponse", by convention. Each <part> element has a name and type attribute, just as a function parameter has both a name and type.

When used for document exchange, WSDL allows the use of <message> elements to describe the document to be exchanged.

The type of a <part> element can be an XSD base type, a SOAP defined type (soapenc), a WSDL defined type (wsdl), or a Types section defined type.

There can be zero, one, or more <portType> elements in the PortTypes section. Because abstract PortType definitions can be place in a separate file, it is possible to have zero <portType> element in a WSDL file.' The sample above shows only one <portType> element. As you can see, a <portType> element defines one or more operations in <operation> elements. The sample shows only one <operation> element, named "foo". This name is equivalent to a function name. The <operation> element can have one, two, or three child elements: the <input>, <output>, and <fault> elements. The message attribute in each <input> and <output> element refers to the relevant <message> element in the Messages section. Thus, the whole <portType> element in the sample is equivalent to the following C function declaration:

```
int foo(int arg);
```

This example shows how much more verbose XML is compared to C. (Including <message> elements, in the sample it took 12 lines of XML to express the same single-line function declaration.)

The Bindings section can have zero, one, or more <binding> elements. Its purpose is to specify how each <operation> cal and response is sent over the wire. The Services section can also have zero, one, or more <service> elements. It contains <port> elements, each of which refers to a <binding> element in the Bindings section. Both the Bindings and Services sections comprise the concrete descriptions of a WSDL document.

Namespaces

In both the root element <definitions> and child element <schema> are namespace attributes:

```
<definitions name="FooSample"
targetNamespace="http://tempuri.org/wsdl/"
xmlns:wsdlns="http://tempuri.org/wsdl/"
xmlns:typens="http://tempuri.org/xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:stk="http://schemas.microsoft.com/soap-toolkit/wsdl-extension"
xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
<schema targetNamespace="http://tempuri.org/xsd"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
elementFormDefault="qualified" >
</schema>
</types>
```

Each namespace attribute declares a shorthand for each namespace that is used in the document. For instance "xmlns:xsd" defines a shorthand (xsd) for the namespace http://www.w3.org/2001/XMLSchema. This allows references to

this namespace later in the document simply by prefixing (or "qualifying") a name with "xsd:" as in "xsd:int", which is a qualified type name. Normal scoping rules apply for the shorthand prefixes. That is, a prefix defined in an element only holds within that element.

What are namespaces for? The purpose of namespaces is to avoid naming conflicts. If I establish a Web service whose WSDL file contains an element with a name of "foo", and you want to use my Web service in conjunction with another, complementary service, then without namespaces that other Web service can not use the name "foo" anywhere in its WSDL file. Both services can use the same name only if they mean exactly the same thing in both instances. With two distinct namespaces, my Web service "foo" can mean a different thing than the other Web service "foo". In your client, you would have to refer to my "foo" by qualifying it. For example, http://www.infotects.com/fooService#foo is a fully qualified name which can be equivalent to "carlos:foo" if I declare carlos as a shorthand for http://www.infotects.com/fooService. Note that URI's are used for namespaces to guarantee their uniqueness and also to allow locators in the document. The location pointed to by the URI does not have to correspond to a real Web location. A GUID can also be used instead of, or in addition to, a URI. For example, the GUID "335DB901-D44A-11D4-A96E-0080AD76435D" is a valid namespace designator.

The targetNamespace attribute declares a namespace to which all names declared in an element will belong. In the sample WSDL file, the targetNamespace for <definitions> is http://tempuri.org/wsdl. This means that all names declared in this WSDL document belong to this namespace. The <schema> element has its own targetNamespace attribute with a value of http://tempuri.org/xsd so that all names defined in this <schema> element belong to this namespace instead of the main target namespace.

The following line in the <schema> element declares a default namespace. All unqualified names in the schema belong to this namespace.

```
xmlns="http://www.w3.org/2001/XMLSchema"
```

SOAP Messages

One way of looking at a WSDL file is that, for the clients and servers that use it, it determines what gets sent on the wire. Although SOAP uses low-level protocols such as IP and HTTP, the application determines the high-level protocol that it uses between a particular client and a particular server. In other words, given an operation, say "echoInt" that simply echoes back an input integer, the count of parameters, the type of each parameter, and how those parameters are to be sent through the wire (serialization) make up an application-specific protocol. Such protocol can be specified in many ways, and I believe the best way is to use WSDL. If we look at it this way, WSDL is not just an "interface contract"; it is also a protocol specification language. It is precisely what we need if we are to go beyond "fixed" protocols such as IP and HTTP towards application-specific protocols.

WSDL can specify whether SOAP messages conform to either rpc or document styles. An rpc-style message, as used in the example, looks like a function call with zero or more parameters. A document-style message is flatter and requires less nesting levels. The XML messages below are sent and received as a result of parsing the sample WSDL file using the SoapClient object in MS SOAP Toolkit 2.0 (MSTK2).

Sent from client to make a function call "foo(5131953)":

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<m:foo xmlns:m="http://tempuri.org/message/">
<arg>5131953</arg>
</m:foo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Received back from server (response):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
```

```
<SOAP-ENV:Body>
<SOAPSDK1:fooResponse xmlns:SOAPSDK1="http://tempuri.org/message/">
<result>5131953</result>
</SOAPSDK1:fooResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Both the function call message and its response are valid XML. A SOAP message consists of an <Envelope> element that contains an optional <Header> element and at a minimum, one <body> element. Both sent and received messages have a single <Body> element in the main <Envelope> element. The rpc function call message body has an element named after the operation name "foo", while the response body has a "fooResponse" element. The foo element has one part, <arg>, which is the single argument, as described in the sample WSDL. The fooResponse likewise has a single <result> part. Note how the encodingStyle, envelope, and message namespace is as prescribed in the WSDL Bindings section repeated here:

```
<binding name="SimpleBinding" type="wsdlns:SimplePortType">
     <stk:binding preferredEncoding="UTF-8" />
        <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="foo">
           <soap:operation
              soapAction="http://tempuri.org/action/Simple.foo"/>
              <input>
                   <soap:body use="encoded"
                        namespace="http://tempuri.org/message/"
                        encodingStyle=
                              "http://schemas.xmlsoap.org/soap/encoding/" />
              </input>
              <output>
                   <soap:body use="encoded"
                        namespace="http://tempuri.org/message/"
                        encodingStyle=
                              "http://schemas.xmlsoap.org/soap/encoding/" />
              </output>
        </operation>
      </binding>
```

XML Schema in the Types and Messages Sections of WSDL

WSDL data typing is based on "XML Schema: Datatypes" (XSD) which is now a W3C Recommendation. There are three different versions of this document (1999, 2000/10, and 2001), and declaring it in as one of the namespace attributes in the <definitions> element specifies which version is used in a particular WSDL file:

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

In this article I will consider the 2001 version only. Proponents of the WSDL standard strongly recommend the use of the 2001 version.

In this and the following sections, the following prefixes or namespace shorthands are used:

Prefix	Equivalent Namespace	Description
soapenc	http://schemas.xmlsoap.org/soap/encoding	SOAP 1.1 encoding
wsdl	http://schemas.xmlsoap.org/wsdl/soap	WSDL 1.1
xsd	http://www.w3.org/2001/XMLSchema	XML Schema

XSD Base Types

The following table, taken right out of the MSTK2 documentation, enumerates all XSD base types supported by MSTK2. It shows how the WSDL reader, both on the client and server sides, maps XSD types to variant types and corresponding types in VB, C++, and IDL.

XSD (Soap) Type	Variant type	VB	C++	IDL	Comments
anyURI	VT_BSTR	String	BSTR	BSTR	
base64Binary	VT_ARRAY | VT_UI1	Byte()	SAFEARRAY	SAFEARRAY(unsigned char)	
boolean	VT_BOOL	Boolean	VARIANT_BOOL	VARIANT_BOOL	

byte	VT_I2	Integer	short	short	Range validated on conversion.
date	VT_DATE	Date	DATE	DATE	Time set to oo:oo:oo
dateTime	VT_DATE	Date	DATE	DATE	
double	VT_R8	Double	double	double	
duration	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
ENTITIES	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
ENTITY	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
float	VT_R4	Single	float	float	
gDay	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
gMonth	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
gMonthDay	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
gYear	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
gYearMonth	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
ID	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
IDREF	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
IDREFS	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
int	VT_I4	Long	long	long	
integer	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.
language	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
long	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.
Name	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
NCName	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
negativeInteger	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.
NMTOKEN	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
NMTOKENS	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
nonNegativeInteger	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.

nonPositiveInteger	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.
normalizedString	VT_BSTR	String	BSTR	BSTR	
NOTATION	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
number	VT_DECIMAL	Variant	DECIMAL	DECIMAL	
positiveInteger	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.
QName	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
short	VT_I2	Integer	short	short	
string	VT_BSTR	String	BSTR	BSTR	
time	VT_DATE	Date	DATE	DATE	Day set to December 30, 1899
token	VT_BSTR	String	BSTR	BSTR	No validation or conversion performed
unsignedByte	VT_UI1	Byte	unsigned char	unsigned char	
unsignedInt	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.
unsignedLong	VT_DECIMAL	Variant	DECIMAL	DECIMAL	Range validated on conversion.
unsignedShort	VT_UI4	Long	Long	Long	Range validated on conversion.

XSD defines two sets of built-in data types: primitive and derived. It is instructive to examine the hierarchy of built-in data types in http://www.w3.org/TR/2001/PR-xmlschema-2-20010330.

Complex Types

The XML Schema allows for definition of complex types, which in C would be structs. For example, to define the equivalent of the following C struct:

```
typedef  struct {
   string  firstName;
   string  lastName;
   long    ageInYears;
   float    weightInLbs;
   float    heightInInches;
} PERSON;
```

we can write in XML Schema:

```
<xsd:complexType name="PERSON">
   <xsd:sequence>
      <xsd:element name="firstName" type="xsd:string"/>
      <xsd:element name="lastName" type="xsd:string"/>
      <xsd:element name="ageInYears" type="xsd:int"/>
      <xsd:element name="weightInLbs" type="xsd:float"/>
      <xsd:element name="heightInInches" type="xsd:float"/>
   </xsd:sequence>
</xsd:complexType>
```

However, <complexType> can express more than just the equivalent of a struct. It can have other child elements other than <sequence>. Instead of a <sequence>, I could have used <all>:

```
<xsd:complexType name="PERSON">
   <xsd:all>
      <xsd:element name="firstName" type="xsd:string"/>
      <xsd:element name="lastName" type="xsd:string"/>
      <xsd:element name="ageInYears" type="xsd:int"/>
      <xsd:element name="weightInLbs" type="xsd:float"/>
      <xsd:element name="heightInInches" type="xsd:float"/>
   </xsd:all>
```

```
</xsd:complexType>
```

This would have meant that the member variable <element>s can come in any order, and each one is optional. That would not have been exactly the way a C struct should be.

Note the use of built-in datatypes string, int, float in the sample. A C string is also a string in XML, and a float is a float. But a C long is an XML int (as in the table above).

In a WSDL file, the Types section is where complex types such as the one above can be declared. For example, I can declare the PERSON type in the following manner and use it in the Messages section:

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions Ã‚Â… >
<types>
     <schema targetNamespace="someNamespace"
            xmlns:typens="someNamespace" >
     <xsd:complexType name="PERSON">
        <xsd:sequence>
           <xsd:element name="firstName" type="xsd:string"/>
           <xsd:element name="lastName" type="xsd:string"/>
           <xsd:element name="ageInYears" type="xsd:int"/>
           <xsd:element name="weightInLbs" type="xsd:float"/>
           <xsd:element name="heightInInches" type="xsd:float"/>
        </xsd:sequence>
     </xsd:complexType>
     </schema>
</types>

<message name="addPerson">
     <part name="person" type="typens:PERSON"/>
</message>

<message name="addPersonResponse">
     <part name="result" type="xsd:int"/>
</message>

</definitions>
```

In the above example, the first message has a name of "addPerson", and it has one <part> whose type is "PERSON". The type PERSON is declared as a complex type in the Types section.

If we use a complete WSDL file with the above fragment when initializing the MSTK2 SoapClient, it will parse the file successfully. Still, it won't be able to send a function call to <addPerson>. This is because SoapClient doesn't know how to handle complex types by itself; it needs a custom type mapper to handle the complex type. The MSTK2 documentation has a sample application that includes a custom type mapper.

There is another way to relate a <part> element to a type declaration. This other way of associating type to a <part> is to use an element instead of a type attribute. In the next example (below) I declare two elements in the Types section ("Person" and "Gender"), which I then refer to in the "addPerson" <message> using an element attribute.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions Ã‚Â… >
<types>
     <schema targetNamespace="someNamespace"
            xmlns:typens="someNamespace" >
     <element name="Person">
     <xsd:complexType>
        <xsd:sequence>
           <xsd:element name="firstName" type="xsd:string"/>
           <xsd:element name="lastName" type="xsd:string"/>
           <xsd:element name="ageInYears" type="xsd:int"/>
           <xsd:element name="weightInLbs" type="xsd:float"/>
           <xsd:element name="heightInInches" type="xsd:float"/>
        </xsd:sequence>
     </xsd:complexType>
     </element>
     <element name="Gender">
     <xsd:simpleType>
        <xsd:restriction base="xsd:string">
```

```
            <xsd:enumeration value="Male" />
            <xsd:enumeration value="Female" />
         </xsd:restriction>
      </xsd:simpleType>
      </element>
      </schema>
</types>

<message name="addPerson">
      <part name="who" element="typens:Person"/>
      <part name="sex" element="typens:Gender"/>
</message>

<message name="addPersonResponse">
      <part name="result" type="xsd:int"/>
</message>
</definitions>
```

The Gender <element> in the Types section has embedded in it an anonymous enumeration type whose possible values are "Male" and "Female". I then refer to that element in the "addPerson" <message> by using an element attribute instead of a type attribute.

What is the difference between "element" and "type" attributes when associating a particular type to a <part>? Using the "type" attribute, we can describe a part that can assume several types (just like a variant), which we can't do when using the "element" attribute. The example below illustrates this.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions Ã‚Â… >
<types>
      <schema targetNamespace="someNamespace"
             xmlns:typens="someNamespace">
      <xsd:complexType name="PERSON">
         <xsd:sequence>
            <xsd:element name="firstName" type="xsd:string"/>
            <xsd:element name="lastName" type="xsd:string"/>
            <xsd:element name="ageInYears" type="xsd:int"/>
            <xsd:element name="weightInLbs" type="xsd:float"/>
            <xsd:element name="heightInInches" type="xsd:float"/>
         </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="femalePerson">
         <xsd:complexContent>
            <xsd:extension base="typens:PERSON" >
            <xsd:element name="favoriteLipstick" type="xsd:string" />
            </xsd:extension>
         </xsd:complexContent>
      </xsd:complexType>
      <xsd:complexType name="malePerson">
         <xsd:complexContent>
            <xsd:extension base="typens:PERSON" >
            <xsd:element name="favoriteShavingLotion" type="xsd:string" />
            </xsd:extension>
         </xsd:complexContent>
      </xsd:complexType>
      <xsd:complexType name="maleOrFemalePerson">
         <xsd:choice>
            <xsd:element name="fArg" type="typens:femalePerson" >
            <xsd:element name="mArg" type="typens:malePerson" />
         </xsd:choice>
      </xsd:complexType>
      </schema>
</types>

<message name="addPerson">
      <part name="person" type="typens:maleOrFemalePerson"/>
</message>

<message name="addPersonResponse">
      <part name="result" type="xsd:int"/>
</message>

</definitions>
```

The above example also illustrates the use of derivation by extension. Both "femalePerson" and "malePerson" are derived from "PERSON". Each has an extra element: "favoriteLipstick" for "femalePerson" and "favoriteShavingLotion" for "malePerson". The two derived types are combined into a single complex type, "maleOrFemalePerson", using the <choice> construct. Finally, in the "addPerson" <message>, the combined type is referred to by the "person" <part>. This <part> or parameter can then be either a "femalePerson" or a "malePerson".

Arrays

XSD provides the <list> construct for declaring an array of items delimited by whitespace. However, SOAP does not use XSD lists to encode arrays. Instead, it defines its own type for arrays, the "SOAP-ENC:Array". The following example shows how to derive from this type to declare a type for a single-dimension array of integers:

```
<xsd:complexType name="ArrayOfInt">
   <xsd:complexContent>
      <xsd:restriction base="soapenc:Array">
         <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

A new complex type is declared by deriving from soapenc:Array using derivation by restriction. An attribute for the complex type is then declared, the arrayType attribute. The reference to "soapenc:arrayType" actually does the arrayType attribute declaration as follows:

```
<xsd:attribute name="arrayType" type="xsd:string"/>
```

The wsdl:arrayType attribute value then determines the type of each of the members of the array. The array items can also be of a complex type:

```
<xsd:complexType name="ArrayOfPERSON">
   <xsd:complexContent>
      <xsd:restriction base="soapenc:Array">
         <attribute ref="soapenc:arrayType"
               wsdl:arrayType="typens:PERSON[]"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

WSDL requires that the type name for an array is the concatenation of "ArrayOf" and the type of the items in the array. It is then clear that from the name alone, "ArrayOfPERSON" is an array of PERSON structs. Below I use ArrayOfPERSON to declare a <message> for adding not just one PERSON but a number of PERSONs:

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions Ã‚Â… >
<types>
     <schema targetNamespace="someNamespace"
            xmlns:typens="someNamespace" >
     <xsd:complexType name="PERSON">
        <xsd:sequence>
           <xsd:element name="firstName" type="xsd:string"/>
           <xsd:element name="lastName" type="xsd:string"/>
           <xsd:element name="ageInYears" type="xsd:int"/>
           <xsd:element name="weightInLbs" type="xsd:float"/>
           <xsd:element name="heightInInches" type="xsd:float"/>
        </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="ArrayOfPERSON">
        <xsd:complexContent>
           <xsd:restriction base="soapenc:Array">
              <attribute ref="soapenc:arrayType"
           wsdl:arrayType="typens:PERSON[]"/>
           </xsd:restriction>
        </xsd:complexContent>
     </xsd:complexType>
     </schema>
</types>
```

```
<message name="addPersons">
     <part name="person" type="typens:ArrayOfPERSON"/>
</message>

<message name="addPersonResponse">
     <part name="result" type="xsd:int"/>
</message>

</definitions>
```

<portType> and <operation> Elements

A PortType defines a number of operations in the abstract. Operation elements within a PortType define the syntax for calling all methods in the PortType. Each operation element declares the name of the method, the parameters (using <message> elements), and types of each (<part> elements declared in every <message>).

There can be several <portType> elements within a WSDL document. Each <portType> element groups together a number of related operations in much the same way that a COM interface groups a number of methods.

In an <operation> element, there can be at most one <input> element, at most one <output> element, and at most one <fault> element. Each of these three elements has a name and a message attribute.

What is the purpose of the name attribute in the <input>, <output>, and <fault> elements? It can be used to distinguish between two operations that have the same name (overloading). For instance, consider the following two C functions with the same name but dissimilar parameters:

```
void foo(int arg);
void foo(string arg);
```

This kind of overloading can be expressed in WSDL as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="fooDescription"
     targetNamespace="http://tempuri.org/wsdl/"
     xmlns:wsdlns="http://tempuri.org/wsdl/"
     xmlns:typens="http://tempuri.org/xsd"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:stk="http://schemas.microsoft.com/soap-toolkit/wsdl-
            extension"
     xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
     <schema targetNamespace="http://tempuri.org/xsd"
              xmlns="http://www.w3.org/2001/XMLSchema"
              xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
                    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
           elementFormDefault="qualified" >
</schema>
</types>

<message name="foo1">
     <part name="arg" type="xsd:int"/>
</message>

<message name="foo2">
     <part name="arg" type="xsd:string"/>
</message>

<portType name="fooSamplePortType">
     <operation name="foo" parameterOrder="arg " >
          <input name="foo1" message="wsdlns:foo1"/>
     </operation>
     <operation name="foo" parameterOrder="arg " >
          <input name="foo2" message="wsdlns:foo2"/>
     </operation>
</portType>

<binding name="fooSampleBinding" type="wsdlns:fooSamplePortType">
<stk:binding preferredEncoding="UTF-8" />
```

```
     <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http"/>
     <operation name="foo">
     <soap:operation soapAction="http://tempuri.org/action/foo1"/>
          <input name="foo1">
     <soap:body use="encoded" namespace="http://tempuri.org/message/"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
     </input>
</operation>
<operation name="foo">
     <soap:operation soapAction="http://tempuri.org/action/foo2"/>
          <input name="foo2">
               <soap:body use="encoded"
               namespace="http://tempuri.org/message/"
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               />
          </input>
</operation>
</binding>

     <service name="FOOService">
          <port name="fooSamplePort" binding="fooSampleBinding">
               <soap:address
                  location="http://carlos:8080/fooService/foo.asp"/>
          </port>
     </service>
</definitions>
```

At the time of this writing, none of the SOAP implementations are able to do the overloading of operation names. It's important for Java-based clients because Java-based servers use interfaces that utilize the overloading feature of Java. For COM-based clients, it's not important because COM does not support overloading.

<binding> and <operation> Elements

The Binding section is where the protocol, serialization, and encoding on the wire are fully specified. Whereas the Types, Messages, and PortType sections deal with data content in the abstract, the Binding section is where the physical details of data transmission is dealt with. The Binding section concretizes the abstractions made in the first three sections.

The separation of binding specifications from data and message declarations means that service providers who engage in the same type of business can standardize on a set of operations (portType). Each provider can then differentiate from another by providing custom bindings. It helps that WSDL also has an import construct so that the abstract definitions can be put in their own file, separate from the Bindings and Services sections, which can be distributed among service providers for whom the abstract definitions will have been established as a standard. For example, banks can standardize on a set of banking operations that are then accurately described in an abstract WSDL document. Each bank is then still free to "customize" on an underlying protocol, serialization optimizations, and encoding.

Below is the Binding section of the overloading sample WSDL, repeated here so I can discuss its details:

```
<binding name="fooSampleBinding" type="wsdlns:fooSamplePortType">
     <stk:binding preferredEncoding="UTF-8" />
     <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http"/>
     <operation name="foo">
          <soap:operation soapAction="http://tempuri.org/action/foo1"/>
               <input name="foo1">
                    <soap:body use="encoded"
                         namespace="http://tempuri.org/message/"
                         encodingStyle=
                        "http://schemas.xmlsoap.org/soap/encoding/" />
               </input>
     </operation>
     <operation name="foo">
          <soap:operation soapAction="http://tempuri.org/action/foo2"/>
               <input name="foo2">
                    <soap:body use="encoded"
                         namespace="http://tempuri.org/message/"
                         encodingStyle=
                        "http://schemas.xmlsoap.org/soap/encoding/" />
               </input>
```

```
    </operation>
</binding>
```

The <binding> element is given a name (in this case "fooSampleBinding") so that the <port> element in the Services section can refer to it. It has a "type" attribute that refers to a <portType>, which in this case is "wsdlns:fooSamplePortType". The second line is an MSTK2 extension element, <stk:binding>, which specifies the preferredEncoding, "UTF-8".

The <soap:binding> element specifies the style ("rpc") and transport used. The transport attribute refers to a namespace, which signifies that the HTTP SOAP protocol is used.

There are two <operation> elements with the same name, "foo". What distinguishes between the two operations are the two different <input> names, "foo1" and "foo2". The <soap:operation> element within both <operation> elements has the same "**soapAction**" attribute, which is a URI. The **soapAction** attribute is a SOAP-specific URI that simply gets used in the SOAP message verbatim. The resulting SOAP message has a **SOAPAction** header and this URI in the <soap:operation> element becomes its value. The **soapAction** attribute is required for HTTP binding but should not be present for non-HTTP binding. Its use is not clear at this writing. It appears that it can be used to help distinguish between the two "foo" operations in this particular case. SOAP 1.1 states that **soapAction** is used to identify the "intent" of the message. It suggests that the server can use this attribute to route the message without having to parse the whole message. In practice, its use varies. The <soap:operation> element can also contain another attribute, the "style" attribute that is used if it is necessary to override the style specified in the <soap:binding> element for this particular operation.

The <operation> element can contain <input>, <output>, and <fault> elements which all correspond to the same elements in the PortTypes section. Only the <input> element is present in the example above. Each of these three elements has an optional "name" attribute that can be used, as in this case, to distinguish among operations with the same name. Inside the <input> element in the example is a <soap:body> element that specifies what gets into the <body> of the resulting SOAP message. This element has the following attributes:

- **Use**
  This is for specifying whether the data is "encoded" or "literal". "Literal" means that the resulting SOAP message contains data formatted exactly as specified in the abstract definitions (Types, Messages, and PortTypes sections). "Encoded" means that the "**encodingStyle**" attribute (see below) determines the encoding.
- **Namespace**
  Each SOAP message body can have its own namespace to prevent name clashing. The URI specified in this attribute is used verbatim in the resulting SOAP message.
- **EncodingStyle**
  For SOAP encoding, this should have the URI value of "http://schemas.xmlsoap.org/soap/encoding".

Document-style Binding

In the previous section, the <soap:binding> element has a type attribute, which is set to "rpc". This attribute, when set to "document" changes the serialization of messages on the wire. Instead of function signatures, the messages are now document transmissions. In this type of binding, the <message> elements define document formats instead of function signatures. As an example, consider the following WSDL fragment:

```
<definitions
  xmlns:stns="(SchemaTNS)"
  xmlns:wtns="(WsdlTNS)"
  targetNamespace="(WsdlTNS)">

<schema targetNamespace="(SchemaTNS)"
    elementFormDefault="qualified">
  <element name="SimpleElement" type="xsd:int"/>
  <element name="CompositElement" type="stns:CompositeType"/>
  <complexType name="CompositeType">
    <all>
      <element name='a' type="xsd:int"/>
      <element name='b' type="xsd:string"/>
    </all>
  </complexType>
</schema>

<message...>
  <part name='p1' type="stns:CompositeType"/>
```

```
   <part name='p2' type="xsd:int"/>
   <part name='p3' element="stns:SimpleElement"/>
   <part name='p4' element="stns:CompositeElement"/>
 </message>
 Ã‚Â…
 </definitions>
```

The schema has two elements, SimpleElement and CompositeElement, and a type declared (CompositeType). The only <message> element declared has four parts: p1, which is of type CompositeType; p2, which is of type int; p3, which is a SimpleElement; and p4, which is a CompositeElement. Below is a table that compares four kinds of bindings as determined by use/type: rpc/literal, document/literal, rpc/encoded, and document/encoded. The table shows what appears on the wire for each kind of binding.

rpc / literal	**document / literal / type=**

`<operation name="method1" style="rpc" ...>`
`<input>`
`<soap:body parts="p1 p2 p3 p4"`
`use="literal"`
`namespace="(MessageNS)"/>`
`</input>`
`</operation>`
 	
on the wire:
`<soapenv:body... xmlns:mns="(MessageNS)"`
`xmlns:stns="(SchemaTNS)">`
`<mns:method1>`
`<mns:p1>`
`<stns:a>123</stns:a>`
`<stns:b>hello</stns:b>`
`</mns:p1>`
`<mns:p2>123</mns:p2>`
`<mns:p3>`
 	
`<stns:SimpleElement>`
 	
`123`
 	
`</stns:SimpleElement>`
`</mns:p3>`
`<mns:p4>`
`<stns:CompositeElement>`
`<stns:a>123</stns:a>`
`<stns:b>hello</stns:b>`
`</stns:CompositeElement>`
`</mns:p4>`
`</mns:method1>`
`</soapenv:body>`
	`<operation name="method1"`
	`style="document" ...>`
	`<input>`
	`<soap:body parts="p1" use="literal">`
	`</input>`
	`</operation>`
	
	on the wire:
	
	`<soapenv:body... xmlns:stns="(SchemaTNS)">`
	`<stns:a>123</stns:a>`
	`<stns:b>hello</stns:b>`
	`</soapenv:body>`

rpc / encoded	**document / literal / element=**

`<operation name="method1" style="rpc" ...>`
`<input>`
`<soap:body parts="p1 p2" use="encoded"`
`encoding=`
`"http://schemas.xmlsoap.org/soap/encoding/"`
`namespace="(MessageNS)"/>`
`</input>`
`</operation>`
 	
on the wire:
`<soapenv:body... xmlns:mns="(MessageNS)">`
`<mns:method1>`
`<p1 HREF="#1" TARGET="_self"/>`
`<p2>123</p2>`
	`<operation name="method1"`
	`style="document" ...>`
	`<input>`
	`<soap:body parts="p3 p4"`
	
	`use="literal">`
	`</input>`
	`</operation>`
	
	on the wire:
	
	`<soapenv:body... xmlns:stns="(SchemaTNS)">`
	`<stns:SimpleElement>`

</mns:method1> <mns:CompositeType id="#1"> <a>123 hello </mns:CompositeType> </soapenv:body>	123 </stns:SimpleElement> <stns:CompositeElement> <stns:a>123</stns:a> <stns:b>hello</stns:b> </stns:CompositeElement> </soapenv:body>
	document / encoded <operation name="method1" style="document" ...> <input> <soap:body parts="p1 p2" use="encoded" encoding= "http://schemas.xmlsoap.org/soap/encoding/" namespace="(MessageNS)"/> </input> </operation> **on the wire:** <soapenv:body... xmlns:mns="(MessageNS)"> <mns:CompositeType> <a>123 hello </mns:CompositeType> <soapenc:int>123</soapenc:int> </soapenv:body>

<service> and <port> Elements

A service is a set of <port> elements. Each <port> element associates a location with a <binding> in a one-to-one fashion. If there is more than one <port> element associated with the same <binding>, then the additional URL locations can be used as alternates.

There can be more than one <service> element in a WSDL document. There are many uses for allowing multiple <service> elements. One of them is to group together ports according to URL destination. Thus, I can redirect all my stock quote requests simply by using another <service>, and my client program would still work because in this type of service grouping the protocol is invariant with respect to different services. Another use of multiple <service> elements is to classify the ports according to the underlying protocol. For example, I can put all HTTP ports in one <service>, and all SMTP ports in another. My client can then search for the <service> that matches the protocol that it can deal with.

```
<service name="FOOService">
     <port name="fooSamplePort" binding="fooSampleBinding">
          <soap:address
            location="http://carlos:8080/fooService/foo.asp"/>
     </port>
</service>
```

Within one WSDL document, the <service> "name" attribute distinguishes one service from another. Because there can be several ports in a service, the ports also have a "name" attribute.

Summary

In this article I have described the most salient SOAP-specific aspects of the WSDL document. It should be mentioned that WSDL is not limited to describing SOAP over HTTP. WSDL is expressive enough for describing SOAP using HTTP-POST, HTTP-GET, SMTP, and others. With WSDL, SOAP is much easier to deal with both for developers and users. I believe that WSDL and SOAP together will usher in a whole new class of applications that utilize Web services distributed over the net.

WSDL has a number of XML elements in its namespace. The following table summarizes those elements, their attributes,

and contents:

Element	Attribute(s)	Contents (children)
<definitions>	name targetNamespace xmlns (other namespaces)	<types> <message> <portType> <binding> <service>
<types>	(none)	<xsd:schema>
<message>	name	<part>
<portType>	name	<operation>
<binding>	name type	<operation>
<service>	name	<port>
<part>	name type	(empty)
<operation>	name parameterOrder	<input> <output> <fault>
<input>	name message	(empty)
<output>	name message	(empty)
<fault>	name message	(empty)
<port>	name binding	<soap:address>

Resources:

1. WSDL 1.1
2. SOAP 1.1
3. XML Schema Primer
4. MS SOAP Toolkit Download Site
5. A tool for translating IDL to WSDL
6. Free Web Services resources including a WSDL to VB proxy generator
7. PocketSOAP: SOAP related components, tools & source code

Top of Page