WSDL Basic Outlines – By Debadatta Mishra

Structure of WSDL

<wsdl:definitions>

<wsdl:types>
XSD Schema
</wsdl:types>

<wsdl:message>
Typically, a message corresponds to an operation.
The message contains the information needed to perform the operation.
</wsdl:message>

<wsdl:portType>
Named operations that can be performed.
It contains information about the name of the operations like add(), sub(), multiply() etc.
</wsdl:portType>

<wsdl:binding>
Defines protocol for communication
</wsdl:binding>

<wsdl:service>
Contains a set of related end points called port.
Port means URL for the web service invocation in a layman term.
</wsdl:service>

</wsdl:definitions>

Message

Typically, a message corresponds to an operation. The message contains the information needed to perform the operation. Each message is made up of one or more logical parts. Each part is associated with a message-typing attribute. The message name attribute provides a unique name among all messages. The part name attribute provides a unique name among all the parts of the enclosing message. Parts are a description of the logical content of a message. In RPC binding, a binding may reference the name of a part in order to specify binding-specific information about the part. A part may represent a parameter in the message; the bindings define the actual meaning of the part. Messages were removed in WSDL 2.0, in which XML schema types for defining bodies of inputs, outputs and faults are referred to simply and directly.

The WSDL message element defines an abstract message that can serve as the input or output of an operation. Messages consist of one or more part elements, where each part is associated with either an element (when using document style) or a type (when using RPC style).
The structure of message is given below.

```
<wsdl:message name="addRequest">
    <wsdl:part name="parameters" element="ns:add"/>
</wsdl:message>
<wsdl:message name="addResponse">
      <wsdl:part name="parameters" element="ns:addResponse"/>
</wsdl:message>
```

PortType

Defines a Web service, the operations that can be performed, and the messages that are used to perform the operation.
The WSDL portType element defines a group of operations, also known as an interface in most environments. Unfortunately, the term "portType" is quite confusing so you're better off using the term "interface" in conversation. WSDL 1.2 has already removed "portType" and replaced it with "interface" in the current draft of the language. In relation with Java, we can PortType contains methods or method name.

The structure of PortType is given below.

```
<wsdl:portType name="SimpleWebServicePortType">
        <wsdl:operation name="add">
                <wsdl:input message="tns:addRequest" wsaw:Action="urn:add"/>
                <wsdl:output message="tns:addResponse" wsaw:Action="urn:addResponse"/>
        </wsdl:operation>
</wsdl:portType>
```

Binding
Binding defines the protocol used to execute an operation of the WSDL.
Binding stores information about the operation name or method name and contains input and output message interpretaion.
Binding also provides information about the format of the request and response XML payload.

The structure of Binding is given below.

```
<wsdl:binding name="SimpleWebServiceSoap11Binding" type="tns:SimpleWebServicePortType">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="add">
                <soap:operation soapAction="urn:add" style="document"/>
                <wsdl:input>
                        <soap:body use="literal"/>
                </wsdl:input>
                <wsdl:output>
                        <soap:body use="literal"/>
                </wsdl:output>
        </wsdl:operation>
</wsdl:binding>
```

Service
Service provides the information about the end points.
The WSDL service element defines a collection of ports, or endpoints, that expose a particular binding.
The structure of Service is given below.

```
<wsdl:service name="SimpleWebService">
        <wsdl:port name="SimpleWebServiceHttpSoap11Endpoint" binding="tns:SimpleWebServiceSoap11Binding">
                <soap:address location="http://localhost:8080/axis2/services/SimpleWebService"/>
        </wsdl:port>
        <wsdl:port name="SimpleWebServiceHttpSoap12Endpoint" binding="tns:SimpleWebServiceSoap12Binding">
                <soap12:address location="http://localhost:8080/axis2/services/SimpleWebService"/>
        </wsdl:port>
        <wsdl:port name="SimpleWebServiceHttpEndpoint" binding="tns:SimpleWebServiceHttpBinding">
                <http:address location="http://localhost:8080/axis2/services/SimpleWebService"/>
        </wsdl:port>
</wsdl:service>
```

SOAP Binding Style

Style defines the structure of xml payload for <soap:body>

Document Style : The content of the <soap:body> is specified by XML Schema defined the the <WSDL:TYPE> section of wsdl. The document style indicates that the soap body contains XML document which can be validated against pre-defined XML schema

RPC Style : **In case of RPC style , the structure of <soap:body> is as per SOAP 1.1 specification. In case of RPC style <soap:body> contains XML representation of a method.**

**** Style : Define the structure of XML payload for the <soap:body>**

**** Use : Defines the structure of data value. It defines the encoding scheme for the data value.**

Example

Encoded Use

<x xsi:type="xsd:int">5</x>
<y xsi:type="xsd:float">10.0</y>

Literal Use

<X>5</X>
<Y>10.0</Y>

Style and Use in WSDL – By Debadatta Mishra

The "Style" Attribute

WSDL 1.1 specifies the style of the binding as either RPC or document. This choice corresponds to how the SOAP payload - i.e., how the contents of the <soap:Body> element - can be structured. Style is about the format of the soap message.

Here are some details of how each style affects the contents of <soap:Body>:
Document: the content of <soap:Body> is specified by XML Schema defined in the <wsdl:type> section. It does not need to follow specific SOAP conventions. In short, the SOAP message is sent as one "document" in the <soap:Body> element without additional formatting rules having to be considered. Document style is the default choice.
RPC: The structure of an RPC style <soap:Body> element needs to comply with the rules specified in detail in Section 7 of the SOAP 1.1 specification. According to these rules, <soap:Body> may contain only one element that is named after the operation, and all parameters must be represented as sub-elements of this wrapper element.

Section 7 of the SOAP 1.1 specification

To make a method call, the following information is needed:
The URI of the target object
A method name
An optional method signature
The parameters to the method
Optional header data

The "Use" Attribute

The use attribute specifies the encoding rules of the SOAP message. This is also done within the <wsdl:binding> element, as seen in the example above. The value can be encoded or literal. It refers to the serialization rules followed by the SOAP client and the SOAP server to interpret the contents of the <Body> element in the SOAP payload.
use="literal" means that the type definitions literally follow an XML schema definition.
use="encoded" refers to the representation of application data in XML, usually according to the SOAP encoding rules of the SOAP 1.1 specification. The rules to encode and interpret a SOAP body are in a URL specified by the encodingStyle attribute. Encoded is the appropriate choice where non-treelike structures are concerned, because all others can be perfectly described in XML Schema.

Style : Document/RPC
Use : Encoded/Literal

5. SOAP Encoding

The SOAP encoding style is based on a simple type system that is a generalization of the common features found in type systems in programming languages, databases and semi-structured data. A type either is a simple (scalar) type or is a compound type constructed as a composite of several parts, each with a type. This is described in more detail below. This section defines rules for serialization of a graph of typed objects. It operates on two levels. First, given a schema in any notation consistent with the type system described, a schema for an XML grammar may be constructed. Second, given a type-system schema and a particular graph of values conforming to that schema, an XML instance may be constructed. In reverse, given an XML instance produced in accordance with these rules, and given also the original schema, a copy of the original value graph may be constructed.
The namespace identifier for the elements and attributes defined in this section is "http://schemas.xmlsoap.org/soap/encoding/". The encoding samples shown assume all namespace declarations are at a higher element level.
Use of the data model and encoding style described in this section is encouraged but not required; other data models and encodings can be used in conjunction with SOAP (see section 4.1.1).

5.1 Rules for Encoding Types in XML

XML allows very flexible encoding of data. SOAP defines a narrower set of rules for encoding. This section defines the encoding rules at a high level, and the next section describes the encoding rules for specific types when

they require more detail. The encodings described in this section can be used in conjunction with the mapping of RPC calls and responses specified in Section 7.
To describe encoding, the following terminology is used:

A "value" is a string, the name of a measurement (number, date, enumeration, etc.) or a composite of several such primitive values. All values are of specific types.
A "simple value" is one without named parts. Examples of simple values are particular strings, integers, enumerated values etc.
A "compound value" is an aggregate of relations to other values. Examples of Compound Values are particular purchase orders, stock reports, street addresses, etc.
Within a compound value, each related value is potentially distinguished by a role name, ordinal or both. This is called its "accessor." Examples of compound values include particular Purchase Orders, Stock Reports etc. Arrays are also compound values. It is possible to have compound values with several accessors each named the same, as for example, RDF does.
An "array" is a compound value in which ordinal position serves as the only distinction among member values.
A "struct" is a compound value in which accessor name is the only distinction among member values, and no accessor has the same name as any other.
A "simple type" is a class of simple values. Examples of simple types are the classes called "string," "integer," enumeration classes, etc.
A "compound type" is a class of compound values. An example of a compound type is the class of purchase order values sharing the same accessors (shipTo, totalCost, etc.) though with potentially different values (and perhaps further constrained by limits on certain values).
Within a compound type, if an accessor has a name that is distinct within that type but is not distinct with respect to other types, that is, the name plus the type together are needed to make a unique identification, the name is called "locally scoped." If however the name is based in part on a Uniform Resource Identifier, directly or indirectly, such that the name alone is sufficient to uniquely identify the accessor irrespective of the type within which it appears, the name is called "universally scoped."
Given the information in the schema relative to which a graph of values is serialized, it is possible to determine that some values can only be related by a single instance of an accessor. For others, it is not possible to make this determination. If only one accessor can reference it, a value is considered "single-reference". If referenced by more than one, actually or potentially, it is "multi-reference." Note that it is possible for a certain value to be considered "single-reference" relative to one schema and "multi-reference" relative to another.
Syntactically, an element may be "independent" or "embedded." An independent element is any element appearing at the top level of a serialization. All others are embedded elements.
Although it is possible to use the xsi:type attribute such that a graph of values is self-describing both in its structure and the types of its values, the serialization rules permit that the types of values MAY be determinate only by reference to a schema. Such schemas MAY be in the notation described by "XML Schema Part 1: Structures" [10] and "XML Schema Part 2: Datatypes" [11] or MAY be in any other notation. Note also that, while the serialization rules apply to compound types other than arrays and structs, many schemas will contain only struct and array types.

The rules for serialization are as follows:
All values are represented as element content. A multi-reference value MUST be represented as the content of an independent element. A single-reference value SHOULD not be (but MAY be).
For each element containing a value, the type of the value MUST be represented by at least one of the following conditions: (a) the containing element instance contains an xsi:type attribute, (b) the containing element instance is itself contained within an element containing a (possibly defaulted) SOAP-ENC:arrayType attribute or (c) or the name of the element bears a definite relation to the type, that type then determinable from a schema.
A simple value is represented as character data, that is, without any subelements. Every simple value must have a type that is either listed in the XML Schemas Specification, part 2 [11] or whose source type is listed therein (see also section 5.2).
A Compound Value is encoded as a sequence of elements, each accessor represented by an embedded element whose name corresponds to the name of the accessor. Accessors whose names are local to their containing types have unqualified element names; all others have qualified names (see also section 5.4).
A multi-reference simple or compound value is encoded as an independent element containing a local, unqualified attribute named "id" and of type "ID" per the XML Specification [7]. Each accessor to this value is an empty element having a local, unqualified attribute named "href" and of type "uri-reference" per the XML Schema

Specification [11], with a "href" attribute value of a URI fragment identifier referencing the corresponding independent element.
Strings and byte arrays are represented as multi-reference simple types, but special rules allow them to be represented efficiently for common cases (see also section 5.2.1 and 5.2.3). An accessor to a string or byte-array value MAY have an attribute named "id" and of type "ID" per the XML Specification [7]. If so, all other accessors to the same value are encoded as empty elements having a local, unqualified attribute named "href" and of type "uri-reference" per the XML Schema Specification [11], with a "href" attribute value of a URI fragment identifier referencing the single element containing the value.
It is permissible to encode several references to a value as though these were references to several distinct values, but only when from context it is known that the meaning of the XML instance is unaltered.
Arrays are compound values (see also section 5.4.2). SOAP arrays are defined as having a type of "SOAP-ENC:Array" or a type derived there from.

SOAP arrays have one or more dimensions (rank) whose members are distinguished by ordinal position. An array value is represented as a series of elements reflecting the array, with members appearing in ascending ordinal sequence. For multi-dimensional arrays the dimension on the right side varies most rapidly. Each member element is named as an independent element (see rule 2).
SOAP arrays can be single-reference or multi-reference values, and consequently may be represented as the content of either an embedded or independent element.
SOAP arrays MUST contain a "SOAP-ENC:arrayType" attribute whose value specifies the type of the contained elements as well as the dimension(s) of the array. The value of the "SOAP-ENC:arrayType" attribute is defined as follows:

```
arrayTypeValue = atype asize
atype          = QName *( rank )
rank           = "[" *( "," ) "]"
asize          = "[" #length "]"
length         = 1*DIGIT
```

The "atype" construct is the type name of the contained elements expressed as a QName as would appear in the "type" attribute of an XML Schema element declaration and acts as a type constraint (meaning that all values of contained elements are asserted to conform to the indicated type; that is, the type cited in SOAP-ENC:arrayType must be the type or a supertype of every array member). In the case of arrays of arrays or "jagged arrays", the type component is encoded as the "innermost" type name followed by a rank construct for each level of nested arrays starting from 1. Multi-dimensional arrays are encoded using a comma for each dimension starting from 1.

The "asize" construct contains a comma separated list of zero, one, or more integers indicating the lengths of each dimension of the array. A value of zero integers indicates that no particular quantity is asserted but that the size may be determined by inspection of the actual members.

For example, an array with 5 members of type array of integers would have an arrayTypeValue value of "int[][5]" of which the atype value is "int[]" and the asize value is "[5]". Likewise, an array with 3 members of type two-dimensional arrays of integers would have an arrayTypeValue value of "int[,][3]" of which the atype value is "int[,]" and the asize value is "[3]".

A SOAP array member MAY contain a "SOAP-ENC:offset" attribute indicating the offset position of that item in the enclosing array. This can be used to indicate the offset position of a partially represented array (see section 5.4.2.1). Likewise, an array member MAY contain a "SOAP-ENC:position" attribute indicating the position of that item in the enclosing array. This can be used to describe members of sparse arrays (see section 5.4.2.2). The value of the "SOAP-ENC:offset" and the "SOAP-ENC:position" attribute is defined as follows:

```
arrayPoint = "[" #length "]"
```

with offsets and positions based at 0.
A NULL value or a default value MAY be represented by omission of the accessor element. A NULL value MAY also be indicated by an accessor element containing the attribute xsi:null with value '1' or possibly other application-dependent attributes and values.

There are 5 types style/use models in WSDl
RPC/encoded
RPC/literal
Document/encoded
Document/literal
Document/Literal Wrapped

RPC/encoded
In case RPC-Encoded style, the soap request will be like the following.

Request Soap Message for RPC-Encoded

```
<soap:envelope>
   <soap:body>
      <myMethod>
         <x xsi:type="xsd:int">5</x>
         <y xsi:type="xsd:float">5.0</y>
      </myMethod>
   </soap:body>
</soap:envelope>
```

```
function myMethod(int x , float y) {
}

public void myMethod(int x , float y) {
}
```

Encoding
<x xsi:type=”xsd:int”>

WSDL for RPC-Encoded

```
<message name="myMethodRequest">
   <part name="x" type="xsd:int"/>
   <part name="y" type="xsd:float"/>
</message>
<message name="empty"/>

<portType name="PT">
   <operation name="myMethod">
      <input message="myMethodRequest"/>
      <output message="empty"/>
   </operation>
</portType>
```

JAX-RPC follows RPC-Encoded style of WSDL

Advantages
The operation name appears in the message, so the receiver has an easy time dispatching this message to the implementation of the operation.
Soap request is the complete representation of the method called myMethod.

Disadvantages
The various web services specifications are sometimes inconsistent and unclear. The WS-I organization was formed to clear up the issues with the specs. It has defined a number of profiles which dictate how you should write your web services to be interoperable.

The type encoding info (such as xsi:type="xsd:int") is usually just overhead which degrades throughput performance.
You cannot easily validate this message since only the <x ...>5</x> and <y ...>5.0</y> lines contain things defined in a schema; the rest of the soap:body contents comes from WSDL definitions.
Although it is legal WSDL, RPC/encoded is not WS-I compliant.

1. **Encoding is an overhead**
2. **Unable to validate**
3. **Performance Issue**

RPC/literal

Soap Request Message

```
<soap:envelope>
   <soap:body>
      <myMethod>
         <x>5</x>
         <y>5.0</y>
      </myMethod>
   </soap:body>
</soap:envelope>
```

RPC-Literal is same as RPC-Encoded. But RPC-Literal does not have encoding info.

WSDL

```
<message name="myMethodRequest">
   <part name="x" type="xsd:int"/>
   <part name="y" type="xsd:float"/>
</message>
<message name="empty"/>

<portType name="PT">
   <operation name="myMethod">
      <input message="myMethodRequest"/>
      <output message="empty"/>
   </operation>
</portType>
```

Advantages

The WSDL is still about as straightforward as it is possible for WSDL to be.
The operation name still appears in the message.
The type encoding info is eliminated.
RPC/literal is WS-I compliant.

Disadvatages

You still cannot easily validate this message since only the <x ...>5</x> and <y ...>5.0</y> lines contain things defined in a schema; the rest of the soap:body contents comes from WSDL definitions.

Document/encoded

Nobody follows this style. **It is not WS-I compliant**. So let's move on.

Document/literal

Soap Request Message

```
<soap:envelope>
   <soap:body>
      <xElement>5</xElement>
      <yElement>5.0</yElement>
   </soap:body>
</soap:envelope>
```

WSDL

```
<types>
   <schema>
      <element name="xElement" type="xsd:int"/>
      <element name="yElement" type="xsd:float"/>
   </schema>
</types>

<message name="myMethodRequest">
   <part name="x" element="xElement"/>
```

```
   <part name="y" element="yElement"/>
</message>
<message name="empty"/>

<portType name="PT">
   <operation name="myMethod">
      <input message="myMethodRequest"/>
      <output message="empty"/>
   </operation>
</portType>
```

Advantages

There is no type encoding info.
You can finally validate this message with any XML validator. Everything within the soap:body is defined in a schema.
Document/literal is WS-I compliant, but with restrictions

Disadvantages

The WSDL is getting a bit more complicated. This is a very minor weakness, however, since WSDL is not meant to be read by humans.
The operation name in the SOAP message is lost. Without the name, dispatching can be difficult, and sometimes impossible.
WS-I only allows one child of the soap:body in a SOAP message. As you can see in Listing 7, this example's soap:body has two children.
The document/literal style seems to have merely rearranged the strengths and weaknesses from the RPC/literal model. You can validate the message, but you lose the operation name.

What is encoding here ?

The type encoding info (such as xsi:type="xsd:int")

Document/literal wrapped

Soap Request Message

```
<soap:envelope>
   <soap:body>
      <myMethod>
         <x>5</x>
         <y>5.0</y>
      </myMethod>
   </soap:body>
</soap:envelope>
```

WSDL

```
<types>
   <schema>
      <element name="myMethod">
         <complexType>
            <sequence>
               <element name="x" type="xsd:int"/>
               <element name="y" type="xsd:float"/>
            </sequence>
         </complexType>
      </element>
      <element name="myMethodResponse">
         <complexType/>
      </element>
   </schema>
```

```
</types>
<message name="myMethodRequest">
   <part name="parameters" element="myMethod"/>
</message>
<message name="empty">
   <part name="parameters" element="myMethodResponse"/>
</message>

<portType name="PT">
   <operation name="myMethod">
      <input message="myMethodRequest"/>
      <output message="empty"/>
   </operation>
</portType>
```

Notice that this SOAP message looks remarkably like the RPC/literal SOAP message in Listing 5. You might say it looks exactly like the RPC/literal SOAP message, but there's a subtle difference. In the RPC/literal SOAP message, the <myMethod> child of <soap:body> was the name of the operation. In the document/literal wrapped SOAP message, the <myMethod> clause is the name of the wrapper element which the single input message's part refers to. It just so happens that one of the characteristics of the wrapped pattern is that the name of the input element is the same as the name of the operation. This pattern is a sly way of putting the operation name back into the SOAP message.
These are the basic characteristics of the document/literal wrapped pattern:
The input message has a single part.
The part is an element.
The element has the same name as the operation.
The element's complex type has no attributes.
Here are the strengths and weaknesses of this approach:

Advantages

There is no type encoding info.
Everything that appears in the soap:body is defined by the schema, so you can easily validate this message.
Once again, you have the method name in the SOAP message.
Document/literal is WS-I compliant, and the wrapped pattern meets the WS-I restriction that the SOAP message's soap:body has only one child.

Disadvantages

The WSDL is even more complicated.

Note: Apache Axis2,Spring WS,JAX-WS and Metro generates Document-Literal wrapped type of WSDL. JAX-RPC generated RPC-Encoded type wsdl.

An example is given below.
If you generate the WSDL using Apache Axis2, it will generate Document-Literal-Wrapped style WSDL.
The structure of the WSDL will be like this.

```
<wsdl:binding name="SimpleWebServiceSoap11Binding" type="tns:SimpleWebServicePortType">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<wsdl:operation name="add"> <!-- This is the operation name -->
        <soap:operation soapAction="urn:add" style="document"/>
        <wsdl:input>
                <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
                <soap:body use="literal"/>
        </wsdl:output>
</wsdl:operation>
</wsdl:binding>
```

Request Message

```
<?xml version="1.0" encoding="utf-16"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <add xmlns="http://service.simplewebservice/xsd">
      <number1>12</number1>
      <numberStr>13</numberStr>
    </add>
  </soap:Body>
</soap:Envelope>
```

Response Message

```
<?xml version="1.0" encoding="utf-16"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns:addResponse xmlns:ns="http://service.simplewebservice/xsd">
      <ns:return>25</ns:return>
    </ns:addResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

If you use JAX-RPC, it will generate RPC-Encoded style wsdl. The structure of the WSDL will be like this.

```
<binding name="SimpleWebServiceBinding" type="tns:SimpleWebService">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                style="rpc" />
        <operation name="add">
                <soap:operation soapAction="" />
                <input>
                        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                use="encoded" namespace="http://deba.org/wsdl" />
                </input>
                <output>
                        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                use="encoded" namespace="http://deba.org/wsdl" />
                </output>
        </operation>
</binding>
```

Request Message

```
<?xml version="1.0" encoding="utf-16"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://deba.org/wsdl"
xmlns:types="http://deba.org/wsdl/encodedTypes" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:add>
      <int_1 xsi:type="xsd:int">12</int_1>
      <String_2 xsi:type="xsd:string">13</String_2>
    </tns:add>
  </soap:Body>
</soap:Envelope>
```

Response Message

```
<?xml version="1.0" encoding="utf-16"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns0="http://deba.org/types"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <env:Body>
    <ans1:addResponse xmlns:ans1="http://deba.org/wsdl">
      <result xsi:type="xsd:string">25</result>
    </ans1:addResponse>
  </env:Body>
</env:Envelope>
```

What is part ?

Part is a logical concept found inside message to distinguish between request and response message.

```
<wsdl:message name="addRequest">
        <wsdl:part name="parameters" element="ns:add"/>
</wsdl:message>
<wsdl:message name="addResponse">
        <wsdl:part name="parameters" element="ns:addResponse"/>
</wsdl:message>
```

Document Vs RPC and Literal Vs Encoded in WSDL

Style may be considered as the format of the soap message.

RPC vs Document styles

The body of an RPC (remote procedure call) style SOAP message is constructed in a specific way, which is defined in the SOAP standard. It is built around the assumption that you want to call the web service just like you would call a normal function or method that is part of your application code. The message body contains an XML element for each "parameter" of the method. These parameter elements are wrapped in an XML element which contains the name of the method that is being called. The response returns a single value (encoded in XML), just like a programmatic method. The WSDL code for a RPC-style web service is less complex than that of a document-style web service, but this isn't a big deal since WSDLs aren't meant to be handled by humans. In case of RPC style the a web service method is represented as an XML.

A RPC-style request:

```
<soap:envelope>
  <soap:body>
    <multiply>    <!-- web method name -->
      <a>2.0</a>  <!-- first parameter -->
      <b>7</b>    <!-- second parameter -->
    </multiply>
  </soap:body>
</soap:envelope>
```

```
function multiply( int a , int b 0 {
}
```

A document style web service, on the other hand, contains no restrictions for how the SOAP body must be constructed. It allows you to include whatever XML data you want and also to include a schema for this XML. This means that the client's and server's application code must do the marshalling and unmarshalling work. This contrasts with RPC in which the marshalling/unmarshalling process is part of the standard, so presumably should be handled by whatever SOAP library you are using. The WSDL code for a document-style web service is much more complex than that of a RPC-style web service, but this isn't a big deal since WSDLs aren't meant to be handled by humans.

A document-style request:

```
<soap:envelope>
  <soap:body>
    <!-- arbitrary XML -->
    <movies xmlns="http://www.myfavoritemovies.com">
      <movie>
        <title>2001: A Space Odyssey</title>
        <released>1968</released>
      </movie>
      <movie>
        <title>Donnie Darko</title>
        <released>2001</released>
      </movie>
    </movies>
  </soap:body>
</soap:envelope>
```

The main downside of the RPC style is that it is tightly coupled to the application code (that is, if you decide you want to call these web methods like normal methods--this is not a requirement, but this is what the RPC style was designed for). This means that if you want to change the order of the parmeters or change the types of those parameters, this change will affect the definition of the web service itself (just as it would affect the definition of a normal function or method).

Document style services do not have this issue because they are loosely coupled with the application code--the application must handle the marshalling and unmarshalling of the XML data separately. For example, with a document style service, it doesn't matter if the programmer decides to use a "float" instead of an "int" to represent

a particular parameter because it's all converted to XML text in the end.

The main downside of the document style is that there is no standard way of determining which method of the web service the request is for. It's easy to get around this limitation, but, however it's done, it must be done manually by the application code. (Note: The "document/literal wrapped" style removes this limitation; read on for more details.)

Another point to note about the document style is that there are no rules for how the SOAP body must be formatted. This can either be seen as a downside or a strength, depending on your perspective. It's a strength if you are looking for the freedom to handle the message the way you want, but a downside if you don't want to have to do the extra marshalling/unmarshalling work that it requires.

Downside of RPC styles

1. **RPC style is tightly coupled to the application code.**
2. **If you change the function parameters, you will face problem.**

Style is about the format of the SOAP request and response message.
Use is about the interpretation of request and response message.

Literal Vs Encoded

Literal vs. encode deals specifically with the interpretation of the XML payload within the SOAP message.
literal" means "what you see is what you get"- this is just plain XML data.
"literal" implies that you should need nothing more other than the WSDL to establish communication with the web service endpoint. "literal" is also the only messaging mode that is endorsed by the WS-I Basic Profile.

"encoded" means that there is an additional set of rules outside of WSDL that imbue the XML data with some meaning. These rules specify how "something" is encoded/serialized to XML and then later decoded/de-serialized from XML back to "something". This set of rules (encoding) is identified by the encodingStyle attribute.
That encoding was originally designed to help in the serialization and de-serialization of data structures used in most programming languages even of entire object graphs (something not supported under XML Schema). However it was ambiguous enough so that two separate vendor implementations of the encoding may not work together. As a result, if for example you run an Axis web service endpoint with an RPC service style (rpc/encoded messaging mode) other SOAP stacks (like .NET) may not be able to interact with it perfectly because of a slight difference in interpretation of the SOAP encoding.

Literal means that the SOAP body follows an XML schema, which is included in the web service's WSDL document. As long as the client has access to the WSDL, it knows exactly how each message is formatted.

Encoded, on the other hand, means that the SOAP body does not follow a schema, but still follows a specific format which the client is expected to already know. It is not endorsed by the WS-I standard because there can be slight differences in the way in which various programming languages and web service frameworks interpret these formatting rules, leading to incompatabilities.

This makes for 4 different style/encoding combinations:
RPC/encoded - RPC-style message that formats its body according to the rules defined in the SOAP standard (which are not always exact and can lead to incompatabilities).
RPC/literal - RPC-style message that formats its body according to a schema that reflects the rules defined in the SOAP standard. This schema is included in the WSDL.
document/encoded - Document-style message that does not include a schema (nobody uses this in practice).
document/literal - Document-style message that formats its body according to a schema. This schema is included in the WSDL.

There's also a 5th type. It isn't an official standard but it is used a lot in practice. It came into being to compensate for document/literal's main shortcoming of not having a standard way of specifying the web method name:

document/literal wrapped - The same as document/literal, but wraps the contents of the body in an element with the same name as the web service method (just like RPC-style messages). This is what web services implemented

in Java use by default.

The most common usage of WSDL is with SOAP bindings. However, there are many types of bindings and many permutations of possible SOAP messages, which can be a bit confusing.

SOAP messages are categorized into two main styles: document and RPC. Document styles are based around sending XML documents back and forth while RPC (Remote Procedure Call) messages are based around sending function calls in and getting a return value. To further complicate matters, message parts (as described above) can either be a type or an element. This will also affect how the SOAP message is formatted. Finally, SOAP messages can either be encoded or literal (no encoding). The encoding rarely affects the SOAP message a great deal, but in some circumstances (HREFs for example) encoding can change the resulting SOAP message.

Document Style

Document style SOAP messages are based around XML documents. The SOAP Body element, in effect, becomes the root element of the document. This means that document style messages are really not supposed to have more than one part, because the message is supposed to be a document, not a parameter list.

If the part is a type, then the SOAP Body element becomes that type. For the XSD and WSDL message:

XSD

```
<complexType name="foo">
  <sequence>
    <element name="sub1" type="xsd:integer"/>
    <element name="sub2" type="xsd:string"/>
  </sequence>
</complexType>
<element name="MyElement" type="foo"/>
```

WSDL Message section

```
<message name="docTypeIn">
  <part name="doesntMatter" type="fooType"/>
</message>
```

WSDL binding section:

```
<binding name="MyBinding" type="MyPortType">
 <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="oper1">
    <soap:operation soapAction="oper1"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
 </operation>
</binding>
```

SOAP Request

```
 <soap:Body>
   <sub1>42</sub1>
   <sub2>The answer to everything</sub2>
 </soap:Body>
```

The response would look similar, except with the output message instead. Note that the SOAP Body element becomes the fooType type and since the fooType type holds a sequence of two elements: sub1 and sub2, the SOAP Body will hold two elements: sub1 and sub2.

If the part is an element, then the SOAP Body will contain that element as a child. For the above XSD and this WSDL message section (the WSDL binding section is the same):

```
<message name="docTypeIn">
```

```
  <part name="doesntMatter" element="MyElement"/>
</message>
```
The resulting document-style SOAP message would look like:

```
 <soap:Body>
   <MyElement>
     <sub1>42</sub1>
     <sub2>The answer to everything</sub2>
   </MyElement>
 </soap:Body>
```
The element name will be printed as a fully qualified name. In this example <MyElement> is a simple local name with no namespace, but if a namespace were present, it would get printed as <ns0:MyElement xmlns:ns0="myURI">. In general, the subelements of this parent element are NOT fully qualified. There is a flag in the XML Schema specification that says to qualify all children of all elements, but it is not common to use it.

Note that with multiple parts, one COULD define different elements and those elements could be placed in the Body in sequence. This is usually what happens when multiple parts are specified for document-style, element messages, but still bear in mind that this is not standard nor advised. Also, note that the part name doesn't matter in the slightest. There is no place that the part name gets printed in document-style SOAP messages, not even with elements.

There is one more form of document-style SOAP messages called document-wrapped. If you look at the above messages, you will notice that there is no indication of which operation to call. Looking at the WSDLs above, there are two pieces of information necessary to locate a web service: the URI or "location", and the name of the operation. The URI is provided by the transport protocol, in this case the HTTP location in the HTTP request line. However, the operation name is still missing. As you will see below with RPC, the operation name CAN be sent in the SOAP payload, but another common (and usually necessary for document-style messages) way to transmit the operation name via the SOAPAction header in the HTTP header section.

This can be cumbersome and prone to error, because we are now relying on the transport to relay the information about which specific operation to call. The transport directing us to the proper WSDL port makes a lot of sense, but after that, the transport's duties are finished. However, as you will see below, RPC style SOAP messages have their own drawbacks. Fortunately there is a pretty clever way of getting the best of both worlds here.

If we take a document-style SOAP message with an element part, and make sure to name the part the exact operation name, then we, in effect, transmit the operation as part of the SOAP message, making the SOAPAction HTTP header unnecessary. This style of SOAP message is really document-style with an element part, but it is commonly referred to as doc-wrapped and is most commonly used in .NET applications.

Here's an example of doc-wrapped.

XSD:
```
<complexType name="operationParams">
  <sequence>
    <element name="param1" type="xsd:integer"/>
    <element name="param2" type="xsd:string"/>
  </sequence>
</complexType>
<element name="doTestOperation" type="operationParams"/>
<element name="doTestOperationResponse" type="xsd:integer"/>
```
WSDL message section:
```
<message name="doTestOperationIn">
  <part name="doesntMatter" element="doTestOperation"/>
</message>
<message name="doTestOperationOut">
  <part name="doesntMatter" element="doTestOperationResponse"/>
</message>
```

WSDL binding section:

```
<binding name="MyBinding" type="MyPortType">
 <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="oper1">
   <soap:operation soapAction="oper1"/>
   <input>
     <soap:body use="literal"/>
   </input>
   <output>
     <soap:body use="literal"/>
   </output>
 </operation>
</binding>
```

SOAP Request:

```
 <soap:Body>
   <doTestOperation>
     <param1>42</param1>
     <param2>The answer to everything</param2>
   </doTestOperation>
 </soap:Body>
```

SOAP Response:

```
 <soap:Body>
   <doTestOperationResponse>
     42
   </doTestOperationResponse>
 </soap:Body>
```

The SOAPAction header can be omitted (but, if present, it MUST be equal to the operation name, in this case: doTestOperation, and the parameters to this operation are easy to specify and read. In effect, this style gets many of the benfits of RPC without a lot of the drawbacks (we'll cover RPC in the next section). For this reason, it is a very common SOAP style, used in many applications (as noted, most commonly in .NET).

RPC style

So, we've covered the document styles and the drawbacks. Basically, document styles are based around sending XML documents, whereas web services in general tend to represent function calls. These documents have problems representing simple function parameters and there is also the problem of needing a transport level header to specify which function to invoke. Document-style messages can also only specify one part and send a single XML document in the message.

RPC, on the other hand, was created to represent function calls. RPC-style SOAP messages contain a wrapping element that specifies the operation name, and that element contains one child element for each of the function parameters. This allows multiple parts to be specified as either; simple types, complex types, or elements. It also means that the SOAPAction header is unnecessary and can be omitted with RPC-style messages.

Both RPC with typed parts and with element parts have the same structure, with the only difference being the information under the part element.

Here is the general format for RPC-style SOAP messages.

XSD:

```
<complexType name="fooType">
  <sequence>
    <element name="sub1" type="xsd:integer"/>
    <element name="sub2" type="xsd:string"/>
  </sequence>
</complexType>
<element name="MyElement" type="fooType"/>
```

WSDL message section:

```
<message name="doTestOperationIn">
  <part name="part1" element="doTestOperation"/>
  <part name="part2" type="doTestOperation"/>
</message>
<message name="doTestOperationOut">
  <part name="result" type="xsd:string"/>
</message>
```

WSDL binding section:

```
<binding name="MyBinding" type="MyPortType">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http">http://schemas.xmlsoap.org/soap/http"/>
  <operation name="oper1">
    <soap:operation soapAction="oper1"/>
    <input>
      <soap:body use="literal" namespace="wrapperURIIn"/>
    </input>
    <output>
      <soap:body use="literal" namespace="wrapperURIOut"/>
    </output>
  </operation>
</binding>
```

SOAP Request:

```
 <soap:Body>
   <ns1:oper1 xmlns:ns1="wrapperURIIn">
     <part1>
       <MyElement>
         <sub1>42</sub1>
         <sub2>The answer to everything</sub2>
       </MyElement>
     </part1>
     <part2>
       <sub1>76</sub1>
       <sub2>Trombones leading the parade</sub2>
     </part2>
   </ns1:oper1>
 </soap:Body>
```

SOAP Response:

```
 <soap:Body>
   <ns1:oper1Response xmlns:ns1="wrapperURIOut">
     <result>34</result>
   </ns1:oper1>
 </soap:Body>
```

You'll notice that, like document-style, the part with the element contains the fully-qualified element, while the part that's a type becomes the specified type.

You'll also notice that RPC, unlike document-style, treats web service calls as functions with parameters and return values. Whereas document-style just passes around documents for requests and responses, RPC passes around function name, parameters, and result. The operation name in the request is qualified by a namespace specified by the SOAP input body declaration in the WSDL as the namespace attribute. By convention, the operation name in the response will have the tag Response appended to it to show that it is indeed a response, and it also is qualified by a namespace, in this case the namespace attribute in the SOAP output body declaration. This is also known as the wrapper namespace, or the on-the-wire namespace, because it is only used when the SOAP message is being sent and received. After the SOAP message is parsed, that namespace is no longer relevant.

There are two communication style models that are used to translate a WSDL binding to a SOAP message body.

They are: **Document, and RPC**
The advantage of using a Document style model is that you can structure the SOAP body any way you want it as long as the content of the SOAP message body is any arbitrary XML instance. The Document style is also referred to as Message-Oriented style.

However, with an RPC style model, the structure of the SOAP request body must contain both the operation name and the set of method parameters. The RPC style model assumes a specific structure to the XML instance contained in the message body.

Furthermore, there are two encoding use models that are used to translate a WSDL binding to a SOAP message.
They are: **literal, and encoded**
When using a literal use model, the body contents should conform to a user-defined XML-schema(XSD) structure. The advantage is two-fold. For one, you can validate the message body with the user-defined XML-schema, moreover, you can also transform the message using a transformation language like XSLT.

With a (SOAP) encoded use model, the message has to use XSD datatypes, but the structure of the message need not conform to any user-defined XML schema. This makes it difficult to validate the message body or use XSLT based transformations on the message body.

The combination of the different style and use models give us four different ways to translate a WSDL binding to a **SOAP message.**
Document/literal
Document/encoded
RPC/literal
RPC/encoded