Guide: SOAP

SOAP stands for Simple Object Access Protocol.

SOAP is a type of on-the-wire formatting that can encapsulate entire object trees as XML text. It is typically used with an HTTP-based transport to call an operation in a web service (i.e. through a WSDL). SOAP is a very open specification and how a SOAP message is structured is largely a function of its usage and environment.

All SOAP messages, though, follow a similar format. SOAP currently has two versions of its specification out, 1.1 and 1.2. 1.2 is a popular choice, but as it has not yet been fully adopted by the industry, this article will stick with SOAP 1.1 for its examples.

SOAP 1.1 Envelope Structure

blocked URL

The base SOAP element is the *Envelope*. This element can hold an optional Headers element, which can in turn hold any number or kind of child elements. The *Envelope* then must either have a *Body* or a *Fault*. *Faults* are only allowed in a response. The *Body* element can be either in a request or a response.

The *Headers* element contains optional headers for the service. Only one SOAP header is defined by the SOAP specification: the *mustUnderstand* header, which, if *true*, states all supplied headers must be parsed and validated by the receiving service. All other headers are defined by the specific application.

Basically the *Body* element contains the data for the web service. For requests, it can contain which operation to call, and it always holds all of the data needed for the call. For responses, it can contain the operation that was called and will always contain the return information when the operation completed successfully. *Faults*, on the other hand, are present when an error or exception occurs. This can be anything from an operation not being found to invalid data to internal system problems.

The Body element can contain any child element, but usually only one child element is allowed. The child element can be formatted in a number of different ways, depending on the scenario (see below).

SOAP 1.1 Faults

The fault element MUST contain at least the fault code, the fault string, and the detail, and it MAY contain the fault actor. SOAP defines a subset of fault codes to use, but any valid qualified name can be used. The two most commonly used are *soap:Client* and *soap:Server* errors. *soap:Client* errors describe a problem with the received message or the client communication. *soap:Server* errors describe a problem which occurred on the server during execution of the service. In any case, the fault code is a fully qualified name which is composed of a namespace prefix (that must already be defined) and a local name (which must be a valid name within that namespace).

For example, these fault codes are invalid:

```
<faultcode>ClientError</faultcode>
<faultcode>http://mysite.com/myURI::ClientError</faultcode>
<faultcode>NS1:ClientError</faultcode>  // Where NS1 is not defined as a valid namespace prefix
```

These are valid:

```
<faultcode>SOAP:Client</faultcode>  // Where SOAP is declared in the envelope to be the SOAP namespace
<faultcode xmlns:ns1="myURI">ns1:ClientError</faultcode>
```

The fault string is any summary of the error, and the fault actor is a URI defining the source of the fault.

The detail is essentially an element that specifies detail about the fault, and generally contains an XML element that matches one of the fault messages specified in the corresponding operation's definition (see Operations in the Guide: WSDL article). It is almost always present, although it can be omitted in certain circumstances (e.g. if the fault was encountered outside the SOAP Body element), and it can be empty.

SOAP in HTTP

SOAP is very often sent as an HTTP payload. This uses standard HTTP (with its benefits and limitations) to send the SOAP message as a text payload. The standard HTTP request looks like:

```
<method> <location> HTTP/<version>
<headers>
...

<payload>
```

and the standard response:

```
HTTP/<version> <status code> <reason>
<headers>
...

<payload>
```

The request *<method>* is usually POST or GET (although other methods are available). The *<location>* maps to the address URI in the WSDL as described above. The *<version>* is usually *1.1* (although it can also be *1.0*). The headers are all in the form of *<headerName>: <headerValue>*, and the header value can also be a comma delimited list. At the end of the headers is an extra newline, separating the headers from the actual payload. A few headers are almost always present: *Content-Length*, *Content-Type*, and *Host*, but the *SOAPAction* header is frequently present in web service applications as well.

An example HTTP request with a SOAP payload might be:

```
POST http://server:8080/wsdl/address HTTP/1.1
Content-Length: 2080
Content-Type: text/xml
Host: server:8080
SOAPAction: testOperation

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"><SOAP:Body>...
```

An example HTTP response to the above request might look like:

```
HTTP/1.1 200 OK
Content-Length: 210
Content-Type: text/xml

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"><SOAP:Body>...
```

<See the HTTP specification for more information on message formats and types>.

The only other peculiarity with SOAP over HTTP is that any error in processing the SOAP request will result in a SOAP fault sent and an HTTP status code of 500 (Internal Server Error). Any errors with the SOAP request will result in a response with an HTTP status code of 500 and a SOAP fault containing the error information. All other HTTP-related errors will be sent back with the appropriate HTTP status code as defined by the HTTP specification.

SOAP Bindings with WSDL

The most common usage of WSDL is with SOAP bindings. However, there are many types of bindings and many permutations of possible SOAP message s, 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 message s 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.

Here are the five different SOAP styes:

blocked URL

Figure 3: The Five SOAP Styles

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 benefits 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"/>
  <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.

SOAP Encoding

There is one more "axis" to consider when formatting and parsing a SOAP message: encoding. This is specified in the SOAP input and output body tags in the WSDL for each concrete operation in a binding. The use attribute is either set to *literal* to specify no encoding, or to encoded to specify that an encoding scheme will be used/expected to format the message. The only real encoding scheme being used is SOAP encoding. The SOAP encoding mainly allows for references and SOAP arrays, among some other lesser features. It also will generally add an attribute to specify the type for each of the elements, using the XSI (XML Schema for Instances) *type* tag.

The use of any encoding is not WS-I compliant because there isn't any way to standardize on encodings (because they are WSDL extensions by their definition). This doesn't mean that it is illegal in WSDL to use encodings, but in real-world applications, its use is limited due to the lack of standardization.

Style Summary

blocked URL

Figure 4: SOAP Style Combinations

Note: Typed parts must refer to XSD-defined types (xsd:simpleType or xsd:complexType definitions). Element parts must refer to an XSD-defined element (xsd:element definition).

So, the question becomes, which style should we use? The different styles have their pros and cons. There is also the WS-I profile to consider. Although this profile isn't a "specification" per se, it is essentially a standard that WSDL/SOAP users can rally around. The SOAP and WSDL specifications are somewhat vague in places and downright ambiguous in others (for example, whether or not multiple *parts* are allowed in doc-literal isn't clearly stated in the WSDL or SOAP specifications). The WS-I profile is a way to answer some of these questions by limiting the types of WSDL and SOAP messages allowed in order to clear up ambiguities and irregularities. WS-I compliance is by no means necessary, but you'll find that the WS-I compliant styles are more common in real-world applications.

Document/Literal

Benefits

- Document literal, when only one *part* is used and that *part* is an element, is fully WS-I compliant, meaning that document/literal/element is more common in real-world applications.
- It is more intuitive for those who see web services simply as passing XML documents back and forth (rather than as function calls).
- The message is pretty succinct and easy to parse.
- Less extraneous information means faster transmission and parse speeds.
- The *Body's* content is completely defined by the schema, so it can be completely validated from front to back.

Drawbacks

- Only element parts are compliant to the WS-I profile, which means their use is more common than typed parts.
- If document-style messages are used with more than one *part*, there is no standard way of formatting the SOAP for such a message.
- The *operation* name is not present in the SOAP message and must be sent as extra information (e.g. the “SOAPAction” header in HTTP).
- For those who view web services as function calls, this style doesn't make a lot of sense, since you aren't passing in "parameters" to a function.

Document/Encoded

Benefits

- None over document/literal.

Drawbacks

- No one uses this style because it doesn't provide any benefits over document literal (encoding doesn't make sense for a standard XML document format).
- It is not WS-I compliant.

RPC/Literal

Benefits

- RPC-literal messages with *parts* defined as *types* are WS-I compliant.
- The *operation* name is included with the message, making dispatch easier.
- The procedures are parameterized with *parts*, making the web service call essentially a "function" with parameters and a return value.
- There is no redundant *type* encoding information.

Drawbacks

- SOAP with attachments or anything that uses HREFs aren't available without SOAP encoding.
- The messages are usually a bit longer because of the *operation/part* information.
- The messages cannot be fully validated because only the individual *parts* are described by a schema. The number and name of the *parts* mus t rely on the WSDL to validate correctness.
- RPC-literal messages with parts defined as elements are not WS-I compliant.

RPC/Encoded

Benefits

- Encoding allows for SOAP arrays, references, etc.
- Encoding still has the main advantages of RPC (parameterized functions, operation name included, etc).
- The type encoding information is necessary for data graphs (i.e. hrefs) and derived types.

Drawbacks

- Type info will be included which is usually extraneous, confusing, and redundant.
- Using an encoding scheme is not WS-I compliant because all encodings are extensions.
- You still cannot validate the message from the Body, for the same reasons as in RPC-literal.

Document/Wrapped

Benefits

- WS-I compliant since it is essentially document-literal with one element part.
- Messages can be fully validated because the entire message is described by a schema.
- Message can be easily seen as a function call with parameters, much like RPC.
- No encoded type information
- Operation response element is the *operation* name with *Response* appended to it, thus matching the RPC style exactly.

Drawbacks

- The schema and WSDL get more complicated because the element name in the schema must match the operation name.
- Since an element must be defined that matches the operation, only one operation with this name is allowed. Normally you can have the same operation name in two port types and they'll be two different operations. But with doc-wrapped, the element has to match the operation and you can't overload that (not even by putting the element in a different namespace, since the local name is what counts).
- It is not an industry standard, although it is widely used. It came from Microsoft, but no one has taken up the flag and put the specification into words (so there are still some ambiguous edge cases).

Each of the above styles (except for document/encoded which is never used) are all common. Doc-wrapped is newer but it is definitely gaining ground in standard usage because it gains the benefits of RPC and can still be validated as a full document. RPC messages are useful when mapping web service c alls to C++ or Java methods, because they can be overloaded with different parameters, and the parameters are pretty easy to match up to C++ and Java function arguments. Document-literal can be useful when passing full XML documents around to web service implementations that don't implement function calls (BPEL, HTTP servlets, etc.). It is also useful when you need overloaded functions but only have one parameter, or you don't want the extra overhead of the extra RPC elements.

References

WS-I Profile 1.2: http://www.ws-i.org/Profiles/BasicProfile-1.2.html

WSDL 1.1 spec: http://www.w3.org/TR/wsdl

SOAP 1.1 spec: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

XSD data types spec: http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html

Guide to choosing WSDL styles: http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/