Feb. 9, 2000

# Inside Soap by Don Box

The Simple Object Access Protocol (SOAP) is a minimal set of conventions for invoking code using XML and HTTP. DevelopMentor, Microsoft, and UserLand Software submitted SOAP to the IETF as an Internet Draft in December 1999 (available here). Since then, numerous application server/ORB vendors have announced support for the protocol as an Internet-friendly alternative to Microsoft's DCOM, Sun's RMI, and OMG's CORBA/IIOP (see the SOAP FAQ for a list of supporting vendors and products). SOAP utilizes the existing HTTP-centric fabric of the Internet to carry method requests that are encoded as XML both for ease of parsing as well as platform/language agnosticism.

SOAP walks a very precarious tightrope, balancing the needs of developers using sophisticated type-centric technologies like Java and CORBA against the desires of the casual Perl or Tcl programmer writing CGI scripts. This tightrope is similar to the one walked by the W3C Schemas Working Group, who have had to design a metadata format that satisfies the needs of object and database technologies, while at the same time addressing the problem of describing document markup. While SOAP does not mandate the use of XML Schemas, it was certainly designed with them in mind. XML Schemas offer an excellent way to describe SOAP types and endpoints, as their type model matches that of SOAP very closely.

## A Top-Down View

SOAP allows methods to be invoked against endpoints over HTTP. A SOAP endpoint is identified by a URL (just like any HTTP-based resource). A SOAP method is uniquely identified by a namespace URI and an NCName. The NCName maps to the symbolic name of the method. The namespace URI scopes the method name, much like an interface name scopes a method in Java, CORBA, or COM. SOAP method requests are transported in HTTP POST requests. They must have a `SOAPMethodName` HTTP header indicating the method being invoked. The following is a minimal SOAP HTTP header:

```
POST /objectURI HTTP/1.1
Host: www.foo.com
SOAPMethodName: urn:develop-com:IBank#getBalance
Content-Type: text/xml
Content-Length: nnnn
```

This HTTP header indicates that the getBalance method (from the `urn:develop-com:IBank` namespace) should be invoked against the endpoint identified by `http://www.foo.com/objectURI`.

The HTTP payload of a SOAP method request is an XML document that contains the information needed to invoke the request. Assuming that all that is needed to get a bank balance is an account number, the HTTP payload of the request would look something like this:

```
<?xml version='1.0'?>
<SOAP:Envelope
  xmlns:SOAP='urn:schemas-xmlsoap-org:soap.v1'>
 <SOAP:Body>
  <i:getBalance
    xmlns:i='urn:develop-com:IBank'>
   <account>23619-22A</account>
  </i:getBalance>
 </SOAP:Body>
</SOAP:Envelope>
```

After drilling through the `SOAP:Envelope` and `SOAP:Body` elements, note that "root" element of `SOAP:Body` is an element whose namespace-qualified tag name matches the `SOAPMethodName` HTTP header exactly. This redundancy is to allow the HTTP-based infrastructure (proxies, firewalls, web server software) to process the call without parsing XML, while also allowing the XML payload to stand independent of the surrounding HTTP message. Since all that was needed to invoke the `getBalance` method was an account number, only one child element appears below the `i:getBalance` element.

Upon receiving this request, the server-side software is expected to execute some code that corresponds to `getBalance`. How this happens is completely outside the scope of the SOAP protocol. Here are some possible reactions to the request:

1. A CGI program may run.
2. An Apache module may be called.
3. An ASP or JSP page may be processed.
4. A Java Servlet or ISAPI extension may be invoked.
5. A servant may be dispatched inside a CORBA ORB.

6. An XSLT may be run against the request.
7. A human may read the request and start typing a response (unlikely, but legal SOAP!).

Once the server-side operation has executed, an HTTP response message will be returned to the client containing the results of the operation. There are no SOAP-specific HTTP response headers. However, the HTTP payload will contain an XML document that contains the results of the operation. The results will be inside an element whose name matches the method name suffixed by "Response." Here's an example response message (including the HTTP header):

```
200 OK
Content-Type: text/xml
Content-Length: nnnn

<?xml version='1.0'?>
<SOAP:Envelope
   xmlns:SOAP='urn:schemas-xmlsoap-org:soap.v1'>
 <SOAP:Body>
  <i:getBalanceResponse
     xmlns:i='urn:develop-com:IBank'>
   <amount>45.21</amount>
  </i:getBalanceResponse>
 </SOAP:Body>
</SOAP:Envelope>
```

That's it. SOAP endpoints are just URLs. SOAP methods are just a pair of XML element declarations identified by a namespace URI and an NCName.

## A Bottom-Up View

Now that we have looked at a simple SOAP method call, it is useful to dissect the SOAP protocol from the bottom-up. Figure 1 shows the implied layering model of SOAP. While the SOAP specification is not organized according to this figure, the figure acts as a reasonable decomposition of the SOAP protocol. Note that the core of SOAP is the XML 1.0 recommendation and XML Namespaces. This reflects the fact that SOAP is simply an application of XML.

The next layer is the XML Schemas specification. While SOAP does not mandate the use of XML Schemas, it was designed to allow them to act as its type description language. Additionally, several "XML Schema-isms" appear in the SOAP specification. In particular, SOAP's use of the `xsi:type` attribute. Note that neither of these two layers are SOAP-specific. Rather, these are two technologies that SOAP utilizes. The first "new" layer added by SOAP is

the element-normal-form encoding style described by section 8 of the SOAP specification.

**Figure 1: SOAP Layers**

## Encoding Instances

Section 8 of the SOAP specification describes the rules used to encode instances of types. The section 8 rules describe an element-normal-form encoding style, in which all properties of an instance are encoded as child elements, never as attributes. Consider the following Java class definition:

```
public class Person
{
  String name;
  double age;
}
```

The section 8-compliant encoding of an instance of this type would look like this:

```
<Person xmlns='someURI'>
  <name>Don Box</name>
  <age>37</age>
</Person>
```

From an XML Schemas perspective, this assumes that the class definition shown above would yield the following schema definition:

```
<schema
  xmlns='http://www.w3.org/1999/XMLSchema'
  targetNamespace='someURI'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'
  xmlns:this='someURI'>

 <type name='Person'>
  <element name='name'
    type='xsd:string' />
  <element name='age'
    type='xsd:double' />
  <anyAttribute
    namespace='urn:schemas-xmlsoap-org:soap.v1' />
  </type>

 <element name='Person' type='this:Person' />

</schema>
```

Subordinate objects are simply encoded directly beneath the accessor element that describes the referring field. Consider the following Java class:

```
public class Marriage
{
  Person husband;
  Person wife;
}
```

The section 8-compliant encoding of an instance of this type would look like this:

```
<Marriage xmlns='uriForMarriage'>
 <husband>
  <name>Don Box</name>
  <age>37</age>
 </husband>

 <wife>
  <name>Barbara Box</name>
  <age>27</age>
 </wife>
</Marriage>
```

Readers familiar with Don Park's SML work may be feeling a bit of déjà vu here. While SOAP is not strictly SML, the section 8 encoding rules have an SML-like flavor, at least for relatively simple types. One departure from SML is section 8's treatment of shared instances.

In many programming environments, it is possible for one instance to be referred to from multiple locations. For example, consider the following Java code:

```
Marriage wedding = new Marriage();
```

```
wedding.husband = new Person();
wedding.husband.name = 'Don Box';
wedding.husband.age = 37;
wedding.wife = wedding.husband;
```

In this case, the wife and husband fields both refer to the same object. If this usage is allowed for instances of class `Marriage`, then the husband and wife fields would be encoded as *multi-ref accessors*. Multi-ref accessors have no child elements. Rather, they have a lone attribute, `soap:href`, that contains a fragment identifier to an independent element containing the serialized instance. The following is an encoding of the `Marriage` object shown above using multi-ref accessors.

```
<Marriage
  xmlns='uriForMarriage'>
 <husband
   soap:href='#id-1' />
 <wife
   soap:href='#id-1' />
</Marriage>

<Person
  xmlns='someURI'
  soap:id='id-1'>

 <name>Don Box</name>
 <age>37</age>
</Person>
```

In this and all other examples, assume that the namespace URI for SOAP (`urn:schemas-xmlsoap-org:soap.v1`) has been aliased to the `soap` prefix.

## The SOAP Envelope

Looking back at Figure 1, the next layer in the SOAP protocol is the `SOAP:Envelope` construct. SOAP defines the "Envelope" type as a serialization scope. An `Envelope` contains an optional `Header` element followed by a mandatory `Body` element. The `Header` element contains a collection of header entries that act as annotations to the root element of `Body`. The first child element of the `Body` is the root of the instance graph held by the `Envelope`. For example, to encode an instance of `Person` inside an `Envelope`, one would write this:

```
<soap:Envelope
  xmlns:soap='uriForSoap'>

 <soap:Body>
  <Person xmlns='someURI'>
   <name>Don Box</name>
   <age>37</age>
  </Person>
 </soap:Body>

</soap:Envelope>
```

When multi-ref accessors are used, the independent elements they refer to are serialized as children of either the `soap:Header` or `soap:Body` elements:

```
<soap:Envelope
  xmlns:soap='uriForSoap'>

 <soap:Body>
  <Marriage
    xmlns='uriForMarriage'>
   <husband soap:href='#id-1' />
   <wife soap:href='#id-1' />
  </Marriage>

  <Person xmlns='someURI'
    soap:id='id-1'>
   <name>Don Box</name>
```

```
    <age>37</age>
   </Person>
  </soap:Body>

</soap:Envelope>
```

The `SOAP:Header` element follows the same form as the `SOAP:Body` element. However, it may have more than one "root," and each can be marked optional or mandatory using the `SOAP:mustUnderstand` attribute.

## SOAP Methods

The next layer in the SOAP protocol is the SOAP method. A SOAP method is simply a request and an optional response. Both the request and response are encoded as a serialized instance of a type. The type of the request is simply a `<type>` whose fields correspond to the in and in-out parameters of the method. Consider the following CORBA IDL method declaration:

```
float f(in float a1, inout float a2, out float a3);
```

The XML Schema definition for the request and response would look like this:

```
<schema
  targetNamespace='interfaceURI' >

 <type name='f'>
  <element name='a1' type='float' />
  <element name='a2' type='float' />
  <anyAttribute
    namespace='uriForSoap' />
 </type>

 <type name='fResponse' >
  <element name='a2' type='float' />
  <element name='result' type='float' />
  <anyAttribute
    namespace='uriForSoap' />
 </type>

 <element name='f' type='f' />

 <element name='fResponse' type='fResponse' />

</schema>
```

Technically, the `<f>` and `<fResponse>` elements could be transmitted using any transport available. However, SOAP codifies the transport of SOAP methods over HTTP, shown as the final layer in Figure 1. The primary facet of the mapping to HTTP is the mandatory use of the `SOAPMethodName` HTTP header in the POST request. This header must match the tag name of the root element of `SOAP:Body` exactly. To invoke this method against the `http://example.com/objectURI` endpoint, the client sends the following HTTP request:

```
POST /objectURI HTTP/1.1
Host: example.com
SOAPMethodName: interfaceURI#f
Content-Type: text/xml
Content-Length: nnnn

<SOAP:Envelope
  xmlns:SOAP='urn:schemas-xmlsoap-org:soap.v1'>
 <SOAP:Body>
  <i:f
    xmlns:i='interfaceURI'>
   <a1>24</a1>
   <a2>87</a2>
  </i:f>
 </SOAP:Body>
</SOAP:Envelope>
```

After servicing the request, the server sends back the following response:

```
200 OK
Content-Type: text/xml
Content-Length: nnnn

<SOAP:Envelope
  xmlns:SOAP='urn:schemas-xmlsoap-org:soap.v1'>
 <SOAP:Body>
  <i:fResponse
    xmlns:i='interfaceURI'>
   <a2>87.5</a2>
   <result>2.4</result>
  </i:fResponse>
 </SOAP:Body>
</SOAP:Envelope>
```

What clients do with this response is outside the scope of the SOAP specification.

## Conclusion

A few details of the protocol were glossed over in this article, including the syntax for arrays, fault reporting, the use of the HTTP extension framework, and support for alternative encoding styles. These issues are discussed in detail in the SOAP specification.

SOAP is simply an application of XML (and XML Schemas) to HTTP. It invents no new technology. Rather, SOAP leverages the engineering effort already invested in HTTP and XML technologies by codifying the application of the two in the context of remote method invocation.