Tutorial: XML messaging with SOAP

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial introduction

Who should take this tutorial?

This tutorial gives a hands-on introduction to using the Simple Object Access Protocol (SOAP) for communication between components. SOAP is quickly emerging as a very popular protocol for XML messaging. It is relatively simple, and it's designed to work with HTTP, SMTP and other such native Internet protocols. It also has broad support from application vendors and Web-based programming projects. If you are working on dynamic Web applications, Web Services or just distributed programming in general, or if you are contemplating ways of communicating between components using Web protocols, this tutorial will be useful.

Navigation

Navigating through the tutorial is easy:

* Use the Next and Previous buttons to move forward and backward through the tutorial.
* When you're finished with a section, select Next section for the next section. Within a section, use the Section menu button to see the contents of that section. You can return to the main menu at any time by clicking the Main menu button.
* If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Prerequisites

You should be familiar with CORBA IDL, the HTTP protocol and XML (including XML namespaces). If need be, the previous tutorials in this series provide the necessary background. See Resources for a link to those tutorials.

Getting help and finding out more

For technical questions about the content of this tutorial, contact the author, *Uche Ogbuji* .

Uche Ogbuji is a computer engineer, co-founder and principal consultant at *Fourthought, Inc* . He has worked with XML for several years, co-developing *4Suite* , a library of open-source tools for XML development in *Python* , and *4Suite Server* , an open-source, cross-platform XML data server providing standards-based XML solutions. He writes articles on XML for IBM developerWorks, LinuxWorld, SunWorld and XML.com. Mr. Ogbuji is a Nigerian immigrant living in Boulder, CO.

Section 2. Introducing the component

HTML calendar

We'll examine an example component for this tutorial: an event calendar widget.

The calendar widget can receive event updates, and return an HTML-formatted display of events for any given month. This might be used in a company intranet where you want different people to be able to submit event listings using a form, and where you want to have the calendar of upcoming events displayed dynamically on the intranet.

This very simple example will illustrate making requests to a persistent component with XML messaging.

The calendar component IDL

We specify the interface for accessing the calendar component using CORBA IDL. Remember that CORBA IDL is a handy means of specifying formal component APIs even if the communication with the component does not use CORBA directly.

The outline of the IDL is as follows:

```
module Calendar {
  struct Date {
    //...
  };

  exception InvalidDate {
    //...
  };

  interface CalendarWidget {
    //...
  };
};
```

The date structure

We'll be using a date structure in the calendar widget communications.

```
struct Date {
  unsigned short day;
  unsigned short month;
  unsigned short year;
};
```

There are many ways to structure a date, including ISO-8601. I chose to break it all up in this way to illustrate structure types in XML messaging.

The InvalidDate exception

There is always the possibility that a user sends an invalid date such as February 30, 2001. In this case the server has to signal an error, which is represented by the `InvalidDate` exception.

```
exception InvalidDate {
  string field;
};
```

The `field` member indicates which of the date fields was wrong. It is either "month", "date" or "year". Note that in IDL there is a better way to express such a collection of named possibilities: an enumeration. We'll stick to a string in this case for simplicity.

Adding a listing

The first method of the `CalendarWidget` interface allows us to add an event for listing.

```
  void addListing(in Date when, in string what)
      raises (InvalidDate);
```

The `when` argument represents the date on which the event we're listing occurs. The `what` argument represents an arbitrary string describing the event. If an invalid date is provided, an appropriate exception will be raised.

Get the listings

The second method retrieves the listing as an HTML string.

```
    string getListings(in date monthOf)
        raises (InvalidDate);
```

The `monthOf` argument specifies the month and year for which we want to see the listings. The `day` member of this structure is ignored by the widget. The return value is an HTML table with the listings.

The entire IDL

I've deliberately kept the interface simple.

```
module Calendar {
  struct Date {
    unsigned short day;
    unsigned short month;
    unsigned short year;
  };

  exception InvalidDate {
    string field;
  };

  interface CalendarWidget {
    void addListing(in Date when, in string what)
        raises (InvalidDate);
    string getListings(in Date monthOf)
        raises (InvalidDate);
  };
};
```

Section 3. Adding a listing by SOAP request

What is SOAP anyway?

We'll be implementing the interface defined in the last section as XML messages.

In the last tutorial in this series we looked at generic XML messaging. There have been many initiatives to provide a standard mechanism for XML messaging. One of these is Simple Object Access protocol (SOAP).

SOAP is a method of accessing remote objects by sending XML messages, which provides platform and language independence. It works over various low-level communications protocols, but the most common is HTTP, as covered in an earlier tutorial in this series.

The Resources section points to some other material on SOAP, but for purpose of this hands-on tutorial, I'll present SOAP by brief example.

The HTTP request header

The first part of the SOAP HTTP request header is familiar territory.

```
POST /calendar-request HTTP/1.1
Host: uche.ogbuji.net
Content-Type: text/xml; charset="utf-8"
Content-Length: 507
```

The SOAPAction header

The only addition to the HTTP header for SOAP is the SOAPAction.

```
SOAPAction: "http://uche.ogbuji.net/soap-example"
```

This header is intended for firewalls and other network infrastructure that are aware of SOAP, especially for filtering and routing purposes. The value is a URI which identifies the action requested by this message.

Note that there hasn't been a great deal of work on specifying such issues as security, firewall processing and routing of SOAP requests. Unfortunately, SOAP's popularity has to some extent outrun the development of its infrastructure, but these issues are currently being discussed in layers above SOAP.

The SOAP envelope

The HTTP request body contains the SOAP request itself. This is wrapped in a *SOAP envelope*, which contains metadata important for understanding the request. The structure is as follows:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
>
  <!-- SOAP body goes here -->
</SOAP-ENV:Envelope>
```

The SOAP envelope is in the XML namespace prescribed by the specification: `http://schemas.xmlsoap.org/soap/envelope/`.

The envelope must specify how to interpret the SOAP body. This is accomplished with the `SOAP-ENV:encodingStyle` attribute. The value is a URI indicating a specification for the structure of the body.

SOAP encodings

The envelope specified the encoding for the body as `http://schemas.xmlsoap.org/soap/encoding/`. This is a method for structuring the request that is suggested within the SOAP spec itself, known as the *SOAP serialization*.

It's worth noting that one of the biggest technical complaints against SOAP is that it mixes a specification for message transport with a specification for message structure. You needn't feel constrained to use the SOAP serialization encoding style.

Other possible encoding styles include Web Distributed Data Exchange (WDDX), XML Remote Procedure Call (XML-RPC), Resource Description Framework (RDF), or just a custom XML structure. The latter is probably good enough if you're not worried about whether an attribute value of "1" is supposed to be interpreted as a string or an integer, for example. See Resources for information on these encoding styles.

The SOAP body

There is one more layer of SOAP element enclosing the actual elements of the specialized event calendar requests.

```
<SOAP-ENV:Body>
  <!-- User request code here -->
</SOAP-ENV:Body>
```

The SOAP body clearly marks the separation of SOAP *metadata* and data.

SOAP data versus metadata

SOAP is like a set of Russian dolls in its layers of encapsulation. There are the HTTP headers, then the SOAP envelope, and then the SOAP body. Each layer provides *metadata* for the immediately enclosed layer.

Metadata is information that helps an application make sense of the important data being transported. For instance, the HTTP header specifies the character encoding as metadata for the HTTP body. This makes sure that the application properly translates the stream of bits into a series of characters.

The outermost portion of the HTTP body is the SOAP envelope. This contains the `SOAP-ENV:encodingStyle` attribute which specifies the structure of the actual request. This metadata tells the application how to make sense of the XML elements and attributes within the body, which make up the actual request.

SOAP metadata examples

There are several examples of metadata such as you might find in a SOAP envelope. They would typically be implemented as separate elements within a separate XML namespace outside the SOAP envelope but within the body. Some of these are still the subject of ongoing standardization.

* *Transactions*: headers that bind multiple SOAP requests (and other sorts of actions) together to ensure consistent results even in the case of errors.
* *Authentication*: headers to present a user's authorization to make the given request.
* *Tracing*: headers to mark each request distinctly for auditing or troubleshooting purposes.
* *Versioning*: headers to indicate any application-specific versions related to the request.

The request element

But back to our actual SOAP example.

The top-level element within the body is in a namespace particular to the calendar widget.

```
<c:AddListing xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
  <!-- Request arguments go here ->
</c:AddListing>
```

The element name indicates which particular request we're making of the calendar widget: adding a listing.

The request arguments, part 1

You will remember from the IDL that the first request argument is a date structure.

```
<c:when>
  <c:Date>
    <day>21</day><month>6</month><year>2001</year>
  </c:Date>
</c:when>
```

The `when` element gives the argument name. The `Date` element is the structure type.

The `Date` structure contains three elements representing the members of the structure. Remember that the IDL specified these as type short.

Note that the SOAP encoding doesn't require the structure member elements to be in a namespace.

The request arguments, part 2

The second argument is a string describing the listing to be added.

```
<c:what>A total eclipse will be visible across Central Africa.</c:what>
```

Notice how much simpler this argument is to present than `when`, the date argument. Such single values are known as *simple types* in the SOAP encoding. The date argument is what is known as a *compound type* or, more specifically a *struct*.

The entire SOAP request

As you can see, SOAP is pretty straightforward.

```
POST /calendar-request HTTP/1.1
Host: uche.ogbuji.net
Content-Type: text/plain; charset="utf-8"
Content-Length: 507

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
>
  <SOAP-ENV:Body>
    <c:AddListing xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
      <c:when>
        <c:Date>
          <day>21</day><month>6</month><year>2001</year>
        </c:Date>
      </c:when>
      <c:what>A total eclipse will be visible across Central Africa.</c:what>
    </c:AddListing>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Section 4. The Add listing response

HTTP Response header

There are no special SOAP headers for the response in our case.

```
HTTP/1.1 200 OK
Server: PythonSimpleHTTP/2.0
Date: Tue, 28 Nov 2000 04:23:03 GMT
Content-type: text/xml; charset="utf-8"
Content-length: 296
```

SOAP envelope and body

The HTTP response body also has a SOAP envelope and body at the top level.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
>
  <SOAP-ENV:Body>
    <!-- response details here -->
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The return result

In the IDL the return value for the request is `void`, meaning, in effect, no response. We'll represent this with an empty response element.

```
<c:AddListingResponse xmlns:c="http://uche.ogbuji.net/soap-example/calendar"/>
```

Note that the response element name is merely the request element with "Response" appended. This is a useful convention.

The entire response

Even simpler than the request.

```
HTTP/1.1 200 OK
Server: PythonSimpleHTTP/2.0
Date: Tue, 28 Nov 2000 04:23:03 GMT
Content-type: text/xml; charset="utf-8"
Content-length: 296

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
>
  <SOAP-ENV:Body>
    <c:AddListingResponse xmlns:c="http://uche.ogbuji.net/soap-example/calendar"/>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Section 5. Get listings request/response

The request body

The HTTP request headers and SOAP envelopes for the `getListings` request are just as in the `addListing` request. The only difference is in the body.

```
<c:GetListings xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
  <c:monthOf>
    <c:Date>
      <day/><month>6</month><year>2001</year>
    </c:Date>
  </c:monthOf>
</c:GetListings>
```

Since the `day` field is ignored by the widget, we just use an empty element. The above encodes a request for the event listings for June 2001.

The entire request

Here is the entire HTTP request

```
POST /calendar-request HTTP/1.1
Host: uche.ogbuji.net
Content-Type: text/plain; charset="utf-8"
Content-Length: 424

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
>
  <SOAP-ENV:Body>
    <c:GetListings xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
      <c:when>
        <c:Date>
          <day/><month>6</month><year>2001</year>
        </c:Date>
      </c:when>
    </c:GetListings>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The response

Again the response headers and SOAP envelopes are similar to the `addListing` request, but the body is quite different.

The IDL specifies a string return value, and the prose description indicates that this value is a chunk of HTML representing a table of the calendar listings.

The IDL return value is represented by a value in the body of the SOAP response. Since the value in this case is HTML, and SOAP is XML, we have to be careful of confusing the HTML tags with XML structure. In general, this is something to be mindful of when sending strings containing HTML or XML in SOAP messages.

Response body

The `CDATA` construct is special XML markup that allows us to encode tag names and other special XML characters without confusing the XML processor. This is one way to encode HTML and XML that must be sent as arguments to SOAP requests or responses. Other approaches include Base64 encoding.

```
    <c:GetListingsResponse xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
      <![CDATA[
<TABLE>
  <TR>
    <TD>2001-06-21</TD><TD>A total eclipse will be visible across Central Africa.</TD>
  </TR>
</TABLE>
      ]]>
    </c:GetListingsResponse>
```

The HTML returned in the response is a simple table with one row for each listing that has been added.

The full response

Notice how the HTML is embedded in the XML.

```
HTTP/1.1 200 OK
Server: PythonSimpleHTTP/2.0
Date: Tue, 28 Nov 2000 04:23:03 GMT
Content-type: text/xml; charset="utf-8"
Content-length: 473

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
>
  <SOAP-ENV:Body>
    <c:GetListingsResponse xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
      <![CDATA[
<TABLE>
  <TR>
    <TD>2001-06-21</TD><TD>A total eclipse will be visible across Central Africa.</TD>
  </TR>
</TABLE>
      ]]>
    </c:GetListingsResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Section 6. SOAP faults and other notes

Error handling

In our IDL we specified the `invalidDate` exception in case, for example, the client passes a date of year 2002, month 14, day 6. In SOAP, exceptions are signalled by a special SOAP message known as a fault.

A SOAP fault is considered a server-side error by the HTTP protocol, so its HTTP header is as follows:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 489
```

SOAP envelope

No surprises in the SOAP envelope again.

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

The SOAP fault element

The outer element of every SOAP fault body is a fixed element.

```
<SOAP-ENV:Fault>
  <!-- general fault details here -->
</SOAP-ENV:Fault>
```

A mandatory field in a SOAP fault is the `faultcode` element.

```
<faultcode>SOAP-ENV:Client</faultcode>
```

The `SOAP-ENV:Client` code indicates that the fault is a result of the request contents.

SOAP-specific fault elements

Another mandatory field is the `faultstring` element.

```
<faultstring>Client Error</faultstring>
```

`faultstring` is a human-readable statement of the cause of error.

```
<detail>
  <!-- application-specific details here -->
</detail>
```

Any information specific to the application is provided in the `detail` field.

Application-specific details

We encode the exception in an element that is also in the Calendar namespace (which is analogous to the "Calendar" module in the IDL).

```
<c:invalidDate xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
   <!-- Representation of the invalidDate exception -->
</c:invalidDate>
```

There is only one data member of the `invalidDate`, which is rendered in XML in a straightforward manner.

```
<c:field>month</c:field>
```

A full SOAP fault

SOAP faults have some standard fields and some application-specific fields.

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 489

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
           <faultcode>SOAP-ENV:Client</faultcode>
           <faultstring>Client Error</faultstring>
           <detail>
              <c:invalidDate xmlns:c="http://uche.ogbuji.net/soap-example/calendar">
                 <c:field>month</c:field>
              </c:invalidDate>
           </detail>
       </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Forcing consideration of the headers

We have discussed various SOAP metadata headers. SOAP specifies an optional attribute, `mustUnderstand`, which indicates that if a SOAP server does not understand the header in question, it must signal a fault and abort the request.

This allows the requestor to ensure that the server follows requirements encoded in the headers.

Relationship of SOAP to other protocols

In our example, we use SOAP send within an HTTP body, but SOAP messages can also be sent using other transports such as SMTP (Internet e-mail) or even Jabber.

In general, SOAP is considered a transport protocol for XML messaging, but it in turn relies on other transport protocols (such as HTTP), just as those protocols in turn rely on others (such as TCP/IP).

Structuring SOAP bodies

It is worth noting again that although we have been using the SOAP encoding in this example, we are free to use other encoding styles, such as WDDX and XML-RPC. The SOAP serialization and the messaging protocol should be considered separately, even though they are bundled into the same specification.

The SOAP encoding makes heavy reference to XML Schemas as a suggested way to define the structure of the message bodies. I use IDL instead because it is more established and more widely known. In fact, XML Schemas are not even yet a complete specification.

Opaque SOAP

Some implementations of SOAP do not focus on the XML structure, but allow users to make SOAP requests transparently from the language of their choice.

In other words, rather than explicitly constructing the XML as we have examined here, they allow the programmer to simply call:

```
calendar_widget.addListing(when, where)
```

This call gets automatically translated to a SOAP request to a remote object. Note that such implementations often use the SOAP encoding behind the scenes, so they might be unsuitable if you need to interoperate with a system that uses a different encoding.

Apache SOAP and Python's soaplib are examples of this approach.

Section 7. Resources and feedback

Try it yourself

There is sample *code for download* which implements the SOAP-based calendar widget described in this tutorial.

You can examine the code for details of how to implement the SOAP messaging in Python. It uses all the techniques we have already introduced in this tutorial series.

Also, you can run the code to see the messaging in action. See the enclosed README for details.

Resources

* *The SOAP 1.1 specification*
* *The xml.org Cover Pages on SOAP*
* *Developmentor's SOAP pages, including a SOAP FAQ*
* *soaplib: a friendly SOAP library for Python*
* *Apache SOAP: a Java library*
* *SOAP news and resources*
* *More SOAP information and news*
* *A log of Web pages related to SOAP*
* *Home page for XML Remote Procedure Call (XML-RPC)*

Giving feedback and finding out more

For technical questions about the content of this tutorial, contact the author, *Uche Ogbuji* .

Uche Ogbuji is a computer engineer, co-founder and principal consultant at *Fourthought, Inc* . He has worked with XML for several years, co-developing *4Suite* , a library of open-source tools for XML development in *Python* , and *4Suite Server* , an open-source, cross-platform XML data server providing standards-based XML solutions. He writes articles on XML for IBM developerWorks, LinuxWorld, SunWorld, and XML.com. Mr. Ogbuji is a Nigerian immigrant living in Boulder, CO.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source

file illustrates the power and flexibility of XML.