

Get Your Hands Dirty with SOAP

by Rob Bracken

In the last issue (November/December 2000), I gave a brief overview of SOAP (Simple Object Access Protocol), and Craig Murphy described how you can use the Microsoft SOAP toolkit to implement SOAP clients and servers. This issue, I'll describe a simple Delphi SOAP client and server, which show you what's happening "behind the scenes".

SOAPing up

The aim of this article is to create a client program that uses the SOAP protocol to execute a method on a server object and get a result. I want to keep things as simple as possible, so the server will just add two integers together and return an Int64 sum. I want to send the SOAP call with HTTP, so I'll host the server inside a CGI program, which is called by a Web Server.

This is the sequence of actions:

- 1 The user enters two integer values and clicks the Add button.
- 2 The integer values are packaged into a SOAP method call and sent to the Web Server with an HTTP POST request.
- 3 The Web Server passes the content of the POST request (i.e. the SOAP method call) to a CGI program.
- 4 The CGI program instantiates a TAdder server object, unpacks the method call and calls the method on the server.
- 5 The CGI program packages the result into a SOAP response and returns it to the Web Server.
- 6 The Web Server returns the response to the client program as the content of the HTTP response.
- 7 The client program unpacks the result from the SOAP response and displays it.

Prewash

The client and server programs use the Open Source, native Delphi XML parser components created by Dieter Kohler. You need to download them from <http://www.philo.de/homepage.htm> and install them on the component palette. You can find out more about them in Charlie Calvert's Tech Voyage articles – *The Basics of Parsing XML in Delphi 5* and *Intermediate Level XML parsing in Delphi 5* – both available at <http://community.borland.com>.

The SOAP client

The client program is a form containing two text boxes for input, one text box to display the result, an Add button and a Close button. It also contains two memos, to display the SOAP request and response. Clicking the Add button makes a call to a TAdder object, which is actually a proxy for the real TAdder. You can see the code for the Add button's Click event in Listing 1.

```
procedure TfrmClient.cmdAddClick(Sender:
                                TObject);
{ Convert the edit strings to integers, add
them together and display the result }
var
  taTemp: TAdder;
begin
    // Clear the HTTP memos
  mmoRequest.Clear;
  mmoResponse.Clear;

  // Add the two numbers together and display
  // the results
  taTemp := TAdder.Create;
  try
    edtResult.Text :=
      taTemp.Add(StrToInt(edtFirst.Text),
                StrToInt(edtSecond.Text));
  finally
    taTemp.Free;
  end;
end;
```

Listing 1 – the Click event handler for the Add button

The TAdder proxy packages the method call into a SOAP request and uses a TNMHTTP component to POST it to a web server.

```
function TAdder.Add(Int1, Int2: integer):
                    String;
{ Package the call to the Add method in a SOAP
request and use HTTP to send it to a CGI
program which will unpackage it and call the
method on the real object }
begin
    // Build the XML SOAP request
  FXML := BuildXML(Int1, Int2);
  try
    // Send it via HTTP
    FhttpSOAP.OutputFileMode := False;
    FhttpSOAP.InputFileMode := False;
    FhttpSOAP.Post('http://localhost/scripts/
SOAPServer.exe', FXML.Text);
    // Display the HTTP header
    frmClient.mmoRequest.Lines.AddStrings(FHTTPHeader);
    // Display the SOAP request
    frmClient.mmoRequest.Lines.Add('- XML
request: -');
    frmClient.mmoRequest.Lines.Add(FXML.Text);
    // Extract the return value from the HTTP
    // response
    Result := GetResult(FhttpSOAP.Body, 'Add');
    // Display the returned XML
    frmClient.mmoResponse.Lines.Add('- HTTP
Header: -');
    frmClient.mmoResponse.Lines.Add(FhttpSOAP.Header);
    frmClient.mmoResponse.Lines.Add('- XML
Response: -');
    frmClient.mmoResponse.Lines.Add(FhttpSOAP.Body);
  finally
    FXML.Free;
  end;
end;
```

```
function TAdder.BuildXML(Int1, Int2: integer):
    TStringList;
{ Build the XML for the SOAP request }
begin
    // The XML doesn't have to be on separate
    // lines
    Result := TStringList.Create;
    Result.Add( '<SOAP-ENV:Envelope' );
    Result.Add( 'xmlns:SOAP-ENV="http://
schemas.xmlsoap.org/soap/envelope/" );
    Result.Add( 'SOAP-ENV:encodingStyle="http://
schemas.xmlsoap.org/soap/encoding/" );
    Result.Add( '<SOAP-ENV:Body>' );
    Result.Add( '    <m:Add
xmlns:m="SOAPAdder">' );
    Result.Add( '        <Int1>' +
IntToStr(Int1) + '</Int1>' );
    Result.Add( '        <Int2>' +
IntToStr(Int2) + '</Int2>' );
    Result.Add( '    </m:Add>' );
    Result.Add( '</SOAP-ENV:Body>' );
    Result.Add( '</SOAP-ENV:Envelope>' );
end;
```

Listing 2 – Doing the SOAP method call

When the TNMHTTP component gets its response, it displays a message to tell you if it was successful or not, uses a TXmlToDomParser component to parse the XML into a document tree and extracts the returned result. The proxy passes the result back to the form, which displays it in a text box. You can see the code for these actions in Listing 2.

I've made the proxy put the HTTP for the request and response into memo boxes, so you can see what was sent.

Packaging a SOAP request

A SOAP request is a block of XML, which adheres to XML syntax rules. It consists of a SOAP envelope, containing an optional SOAP header and a mandatory SOAP body. The SOAP XML for our method call looks like this:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
soap/envelope/"
  SOAP-ENV:encodingStyle="http://
schemas.xmlsoap.org/soap/encoding/" >
  <SOAP-ENV:Body>
    <m:Add xmlns:m="SOAPAdder">
      <Int1>10</Int1>
      <Int2>100</Int2>
    </m:Add>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

It calls the Add method of a SOAPAdder object, passing the values 10 and 100 as Int1 and Int2.

The SOAP Server

The SOAP server is a Web Module, containing one WebAction item, which responds to the mtPost method type.

The TNMHTTP component in the client program modifies the content of the POST request into a form that won't upset a web browser. It replaces HTML syntax characters (such as "<" and ">") with a 2 digit Hex number, prefixed by "%". So a "<" will be replaced with "%3C". The DecodePost routine converts the content string back to its original form. The WebAction item passes the converted string to a TAdderStub object, which descends from TCustomSOAPStub. The stub object uses a

TXmlToDomParser component to parse the XML into a document tree and extracts the method call, with its parameters. It calls the method on an associated TAdder object, which adds the two numbers and returns an Int64 result.

The stub packages the result into a SOAP XML response and returns it to the WebAction item, which passes it back to the web server as the content of the HTTP POST response. You can see the code for the WebAction item in Listing 3. The code which parses the SOAP XML is in Listing 4 and the code that executes it is in Listing 5.

```
procedure
TWebModule1.WebModule1WebActionItem1Action(Sender:
TObject; Request: TWebRequest; Response:
TWebResponse; var Handled: Boolean);
var
  sTemp: string;
begin
  sTemp := DecodePost(Request.Content);
  Response.Content :=
    FAdderStub.Execute(sTemp);
end;
```

Listing 3 – the WebAction item

Install the program by copying it into the Scripts directory of your web server. I've assumed that you will be using PWS, so the address is hard-coded as localhost. If you want to change it, it's in the Add method of the untAdderProxy unit in the SOAP client.

The SOAP XML response looks like:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
soap/envelope/"
  SOAP-ENV:encodingStyle="http://
schemas.xmlsoap.org/soap/encoding/" >
  <SOAP-ENV:Body>
    <m:Add xmlns:m="SOAPAdder">
      <Sum>110</Sum>
    </m:Add>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Rinsing off

I've taken a simplistic approach to these programs, so that the mechanism is clear. They're not meant to be a definitive demonstration of how to write SOAP proxies and stubs.

The SOAP specification contains a lot more detail on how a SOAP request should be formatted. It also contains details of a SOAP Fault element for error information, which could be used for exception handling.

I used HTTP for the transport. You could also use SMTP, straight TCP/IP or a text file to pass the SOAP request from the client to the server.

Delphi 6 and Kylix will contain support for SOAP interfaces, so all this will be done for us behind the scenes.

The source for this article is available on the BUG website.



Further Information

- SOAP: Simple Object Access Protocol:
- <http://msdn.microsoft.com/xml/general/soapspec.asp>
- Charlie Calvert's XML parsing articles:
- <http://homepages.borland.com/ccalvert/techPapersDelphi/XMLSimple/XMLSimple.html>
- <http://homepages.borland.com/ccalvert/TechPapers/Delphi/XMLBrowse/index.htm>
- A Delphi XML Parser:
- <http://www.philo.de/homepage.htm>
- Everything you wanted to know about XML but were afraid to ask: <http://www.xml.org>

```
function TCustomSOAPStub.Execute(Request:
    string): string;
{ Parse the XML request and execute method
  calls }

function GetNextXMLElement(XMLNode:
    TDOMNode): TDOMNode;
{ Find the _next_ XML element node at this
  level (NodeType = ntElement_Node) }
begin
    Result := XMLNode.NextSibling;
    while (Result.NodeType <> ntElement_Node)
    and (Result <> nil) do
        Result := Result.NextSibling;
end;

function FindNode(XMLNode: TDOMNode;
    sValueToFind: string): TDOMNode;
{ Find the first XML element at this level
  that has the supplied value, starting
  with the supplied node. (An XML element
  is a node with a NodeType of
  tElement_Node)
  Note that we actually look at the
  NodeName property, rather than the
  NodeValue. }

begin
    if XMLNode.NodeType <> ntElement_Node
    then
        begin
            Result := GetNextXMLElement(XMLNode);
        end
    else
        begin
            Result := XMLNode;
        end;
    // Find the first child that has a
        matching value
    while (Result.NodeName <> sValueToFind)
    and (Result <> nil) do
        Result := GetNextXMLElement(XMLNode);
    end;

function GetMethodName(SOAPMethodNode:
    TDOMNode): string;
{ Extract the method name from the node
  name. The node name is in the form
  "ns:method" where ns = the XML namespace
  and method = the method name,
  so we need to extract everything after
  the colon. }
var
    nColonPos: integer;
begin
    nColonPos :=
        Pos(':', SOAPMethodNode.NodeName);
    Result := Copy(SOAPMethodNode.NodeName,
        nColonPos+1, length(SOAPMethodNode.NodeName)
        - nColonPos);
end;
```

```
var
    SOAPEnvelopeNode, SOAPBodyNode,
    SOAPMethodNode: TDOMNode;
begin
    // Parse the XML and load it into a DOM
        document tree
    FDomDoc := FXMLParser.stringToDom(Request);

    // Find the SOAP envelope node (will be a
        child of the DOM document node)
    SOAPEnvelopeNode :=
        FindNode(FDomDoc.FirstChild,
            'SOAP-ENV:Envelope');

    // Find the SOAP Body node (will be a child
        of the envelope node)
    SOAPBodyNode :=
        FindNode(SOAPEnvelopeNode.FirstChild,
            'SOAP-ENV:Body');

    { Method calls will be children of the SOAP
      Body node.
      For simplicity, we've assumed that all the
      method calls are for us, so we don't check
      the namespace. To check the namespace, find
      the "xmlns" attribute of the method node
      and compare its value. }

    SOAPMethodNode := SOAPBodyNode.FirstChild;
    while Assigned(SOAPMethodNode) do
        begin
            { ProcessRequest takes a SOAP method node,
              performs the action and returns the result
              as XML. Actions which don't return
              anything will return a blank string. If
              more than one method is called, the return
              strings are concatenated together. }

            Result := Result +
                ProcessRequest(SOAPMethodNode,
                    GetMethodName(SOAPMethodNode));
            SOAPMethodNode :=
                SOAPMethodNode.NextSibling;

            while Assigned(SOAPMethodNode)
            and (SOAPMethodNode.NodeType <>
                ntElement_Node) do
                // Skip any text or comment nodes
                SOAPMethodNode :=
                    SOAPMethodNode.NextSibling;
            end;
            // Wrap the result in a SOAP envelope.
            Result := '<SOAP-ENV:Envelope' +
                ' xmlns:SOAP-ENV='
                + "http://schemas.xmlsoap.org/soap/envelope/" +
                ' SOAP-ENV:encodingStyle='
                + "http://schemas.xmlsoap.org/soap/encoding/" +
                '>' + Result +
                '</SOAP-ENV:Body>' +
                '</SOAP-ENV:Envelope>';
        end;
end;
```

Listing 4 – Parsing a SOAP request

```
function TAdderStub.ProcessRequest(XMLNode:
    TDOMNode; MethodName: string): string;
{ Class-specific processing of a SOAP request }
function CallAddMethod(XMLNode: TDOMNode):
    string;
{ Perform the add method and return the
  result as XML }
var
    Int1, Int2: integer;
    Sum: Int64;
    tmpNode: TDOMNode;
```

```

const
  SyntaxStuff = [WideChar(#9), WideChar(#10),
                 WideChar(#13)];
begin
  Result := '<m:Add xmlns:m="SOAPAdder">';
  tmpNode := XMLNode.FirstChild;

  while tmpNode.NodeName <> 'Int1' do
    tmpNode := tmpNode.NextSibling;

  tmpNode := tmpNode.FirstChild;
  while tmpNode.NodeValue[1] in SyntaxStuff do
    tmpNode := tmpNode.NextSibling;

  Int1 := StrToInt(tmpNode.NodeValue);
  tmpNode := tmpNode.ParentNode;
  while tmpNode.NodeName <> 'Int2' do
    tmpNode := tmpNode.NextSibling;

  tmpNode := tmpNode.FirstChild;
  while tmpNode.NodeValue[1] in SyntaxStuff do
    tmpNode := tmpNode.NextSibling;

  Int2 := StrToInt(tmpNode.NodeValue);
  Result := Result + '<Sum>' +
             IntToStr(FAdder.Add(Int1, Int2)) +
             '</Sum>';
  Result := Result + '</m:Add>';
end;

```

```

begin
  if Assigned(FAdder) then
  begin
    if UpperCase(MethodName) = 'ADD' then
    begin
      Result := CallAddMethod(XMLNode);
    end;
  end;
end;

```

Listing 5 – Performing the method call



Rob Bracken is Managing Director of Bracken Software Limited and has over 10 years' experience of trying to make different programs talk to each other. You can contact him on rob_bracken@branway.freeserve.co.uk.