

Cleaning up your act with the Microsoft SOAP Toolkit

by Craig Murphy

The Simple Object Access Protocol (SOAP) is a consistent means of transporting data and method calls between potentially distributed applications.

The Microsoft SOAP Toolkit is a developer tool that helps create SOAP “requests” (client calls to a server) and can unscramble SOAP “responses” (server responses to the client).

Over the course of this article, I’ll demonstrate the use of the Microsoft SOAP Toolkit. The example will be implemented using Delphi.

Introduction

SOAP is the collective work of Microsoft, IBM, Lotus Development Corp., UserLand Software, Inc. and Sun Microsystems. This is an impressive line-up, bringing together former arch enemies.

Microsoft has been feverishly busy this year. The .NET framework was announced in July 2000 and during June the BizTalk Framework 2.0 (draft) was issued – both take advantage of SOAP and naturally XML. Thus, it’s fair to say, that there’s going to be some mileage in SOAP and XML and if we’re not already building applications to exploit SOAP and XML, the next best thing that we can do is to learn about how we can.

Microsoft and others have been busy promoting the notion of “Web Services”. They’re keen to use the Internet as the vehicle for exposing function calls via URLs. SOAP (over HTTP) realises this notion – at the end of a URL there are methods that can be executed, i.e. it realises the idea of “the programmable web”. If we are able to implement the programmable web, then the benefits we could reap are enormous. Business-to-Business (B2B) integration will be encouraged, supply-chain integration will become a reality, and alliances and partnering will be promoted.

Prerequisites

If you plan to use SOAP I would urge you to take a look at the web sites I mention at the end of this article. The rest of this article will assume you are reasonably familiar with XML, and that you understand the basics of HTTP.

Some knowledge of SOAP, COM objects and web page scripting would also be useful. As part of the download associated with this article, you’ll find three documents to help fill any gaps:

1. A description of what a SOAP Message looks like.
2. An step-by-step guide that explains how to create the COM object that is used in this article.
3. An example of a web page executing Delphi methods via SOAP.

I’ve built and tested my SOAP applications using Windows 2000 Server. Microsoft has tested The Toolkit using Windows 2000 Server and Professional. The SOAP Toolkit has had basic testing with Windows NT 4.0 SP6a and Internet

Explorer 5.0. At the moment, there is no Windows 9x release – although if you look through the newsgroups mentioned at the end of this article, there are a couple of third parties who have ported a version to Windows 9x.

Unfortunately, the current version of the SOAP Toolkit requires that you have Visual Studio 6 SP3 installed - this is the case even if you’re not doing any development! There are a few OCXs that are installed along with Visual Studio. You don’t need Visual Studio installed on a server acting as a SOAP listener. It seems that Visual Studio is only required if you’re going to use the wizard that is provided with the SOAP Toolkit.

The SOAP Toolkit Explained

The Toolkit is available for download from the Microsoft web site (given at the end of this article).

The Toolkit consists of a compiled HTML Help file, The Service Description Language (SDL) specification (v0.9.4.3), the source code for “ROPE”, source code for ASP and ISAPI SOAP Listeners, a server-side web service, and a client application (not surprisingly, it’s implemented using Visual Basic). There’s also an SDL wizard, which is one of the big time savers, offered by the Toolkit. I’ll explain SDL and ROPE later in this article

The SDL Wizard is a server-side tool. The wizard asks us to provide the path to a COM object that supplies the methods that we wish to expose to our client applications. This is where Delphi fits in – we’re going to create a COM object that exposes some methods that our client applications can use. Our client applications will be a Delphi executable and a web page in Internet Explorer – thus proving that SOAP offers us the ability to execute remote methods regardless of the client capabilities.

The wizard creates an SDL file. SDL stands for Service Description Language. The SDL file is an XML file that describes the methods that are available in the COM object, and thus those methods that are available (over the web) to SOAP consumers.

The fact that the wizard builds an SDL file from a COM object is good news – it means we can leverage our existing investment in Delphi and C++Builder – both are great tools for creating COM objects. Equally, we’re now able to extract data from a legacy database and publish it via the Internet (and Intranet). But that’s not all – the SDL file effectively provides a platform independent interface to our COM object, thus we can use the COM object from whatever platform supports HTTP (and XML).

The wizard also creates an Active Server Page (ASP). The ASP contains what is effectively an interface to the COM object. It comprises a number of [VBScript] functions that instantiate the COM object and calls the method required. Here’s a sample:

```

Public Function Method1 ()
    Dim objMethod1
    Set objMethod1 =
        Server.CreateObject("Project1.abcd")

    Method1 = objMethod1.Method1()
    'Insert additional code here

    Set objMethod1 = NOTHING
End Function

```

The sample assumes a COM object called Project1.abcd has already been created. A method called Method1 is available and it takes no parameters. Thankfully, we don't need to worry ourselves with the actual implementation provided by the SOAP Toolkit – after all, it's simply created an abstraction for our COM object. The ASP is, in essence, nothing more than a wrapper for the COM object.

As an aside...

Interestingly, the wizard doesn't recognise the pseudo-COM object that is a Windows Script Component (WSC). This is a shame really, I find using WSCs useful for prototyping COM objects using JavaScript/VBScript prior to developing them using Delphi. WSCs are useful because it's possible to change the source code on the fly without the need to recompile or unload DLLs, etc.

Client-side support comes in the form of a Remote Object Proxy Engine (ROPE). Unfortunately the ROPE is supplied as a DLL, thus we are immediately imposing client-side operating system dependencies. This is disappointing because SOAP is meant to alleviate platform dependencies – which it does; it is the SOAP Toolkit that imposes the dependency. ROPE.DLL exposes objects that will allow us to create applications (using Delphi or C++Builder) that act as SOAP consumers. We'll learn more about ROPE later in this article.

Implementing SOAP

We could use a brute force approach to implementing SOAP – i.e. we could go ahead and create customised strings that represent SOAP requests and responses. An HTTP component would be needed to effect the “wire transfer” from the client to the server. We'd also need to use an XML parser to give us the ability to “get at” the XML elements. This approach is great for learning about SOAP (and XML, and HTTP), however it is very time consuming (to implement a truly generic and extensible SOAP library).

The SOAP Toolkit takes away much of the effort required by the brute force approach. The Toolkit does this by means of the SDL Wizard.

As part of the download you'll find a Richplum COM object. I'll be using this component to demonstrate The Toolkit.

Thus, these are the steps that I plan to go through:

1. Use the SDL Wizard to create Richplum.xml and Richplum.asp.
2. Create a SOAP consumer application using Delphi.

Step 1 – The SDL Wizard

As with all wizards, there is a series of steps to go through.

Running **SDLWizard.exe** starts the wizard; Figure 1 shows the initial welcome screen.



Figure 1



Figure 2

Clicking on Next reveals Figure 2.

By default, the wizard offers to “Generate source code for an existing COM object”. This is the option that suits us – we've already got a COM object that has been registered and tested. Click on the **Select COM object** button and locate the Richplum.dll that forms part of the download for this article. Once you've selected the COM object, click on the Next button to move on to the next step.



Figure 3

Figure 3 presents us with a list of the interfaces and methods that are provided by the Richplum object. The SOAP wizard allows us to specify which methods we would like to expose to our SOAP consumers. In this case, put a tick against the methods `getCustomerList` and `getCompanyDetail`. Click on the Next button when you've done that.



Figure 4

Figure 4 requests information about the SOAP Listener. The SOAP Listener is a file/process that listens for SOAP Requests. Set the URI to point to `http://localhost/delphi/`. Also, make sure that the “delphi” directory exists as a virtual directory in IIS or PWS.

The Listener type can be either ASP or ISAPI. An ASP listener provides “wrapper” functions around the methods in our COM object – these are implemented using VBScript. The ASP listener, whilst being a little slower than the ISAPI listener, brings with it the advantage of customisation – we can simply edit the VBScript in the ASP page to suit our needs. We'll create an ASP listener.



Figure 5

Figure 5 requests a location for storing source files. The two files that will be created are `Richplum.asp` and `Richplum.xml`. The ASP file is an abstraction of our COM object; the XML file is an SDL file representing the methods that we chose in Figure 3.

Click on Next, then Finish – we've completed all the steps required to create the SDL file and ASP wrapper for the Richplum object.

Listing 1 presents the complete SDL for the Richplum COM object; Listing 2 presents the ASP wrappers.

```
<?xml version='1.0' ?>
<!-- Generated 01/10/2000 17:26:25 by Microsoft
SOAP Toolkit Wizard, Version 205.0.3 -->
<serviceDescription name='SOAPToolkit'
xmlns='urn:schemas-xmlsoap-org:sdl.2000-01-25'
xmlns:dt='http://www.w3.org/1999/XMLSchema'
xmlns:Richplum='Richplum'
>
<import namespace=
'Richplum' location='#Richplum' />
<soap xmlns=
'urn:schemas-xmlsoap-org:soap-sdl-2000-01-25'>
<interface name='Richplum'>
<requestResponse name='getCustomerList'>
<request ref=
'Richplum:getCustomerList' />
<response ref=
'Richplum:getCustomerListResponse' />
</requestResponse>
<requestResponse name='getCompanyDetail'>
<request ref=
'Richplum: getCompanyDetail' />
<response ref=
'Richplum:getCompanyDetailResponse' />
<parameterorder>CustNo</parameterorder>
</requestResponse>
</interface>
<service>
<addresses>
<address uri=
'http://localhost/delphi/Richplum.asp' />
</addresses>
<implements name='Richplum' />
</service>
</soap>

<Richplum:schema id='Richplum'
targetNamespace='Richplum' xmlns='http://
www.w3.org/1999/XMLSchema'>
<element name='getCustomerList'>
</element>
<element name='getCustomerListResponse'>
<type>
<element name='return'
type='dt:string' />
</type>
</element>
<element name='getCompanyDetail'>
<type>
<element name='CustNo' type='dt:integer' />
</type>
</element>
<element name='getCompanyDetailResponse'>
<type>
<element name='return' type='dt:string' />
</type>
</element>
</Richplum:schema>
</serviceDescription>
```

Listing 1 – The Richplum SDL

```

<%@ Language=VBScript %>

<% Option Explicit

Response.Expires = 0

' _____
' SOAP ASP Interface file Richplum.asp
' Generated 01/10/2000 17:26:26
' By Microsoft SOAP Toolkit Wizard, Version
205.0.3
' _____

Const SOAP_SDLURI = "http://localhost/delphi/
Richplum.xml" 'URI of service description file
%>
<!--#include file="listener.asp"-->

<%
' _____

Public Function getCustomerList ()
    Dim objgetCustomerList
    Set objgetCustomerList =
        Server.CreateObject("SOAPToolkit.Richplum")

    getCustomerList =
        objgetCustomerList.getCustomerList()
    'Insert additional code here

    Set objgetCustomerList = NOTHING
End Function

' _____

Public Function getCompanyDetail (ByVal CustNo)
    Dim objgetCompanyDetail
    Set objgetCompanyDetail =
        Server.CreateObject("SOAPToolkit.Richplum")

    getCompanyDetail =
        objgetCompanyDetail.getCompanyDetail(CustNo)
    'Insert additional code here

    Set objgetCompanyDetail = NOTHING
End Function

' _____

%>

```

Listing 2 – The Richplum ASP wrapper functions



SOAP Toolkit ASP Listener – Architecture

Figure 6 provides a graphical view of our implementation. The SOAP Toolkit provides a generic Listener.asp. Listener.asp is able to analyse SOAP Requests and pass them through to the ASP Wrapper function, which in turn executes a method in the Richplum COM object.

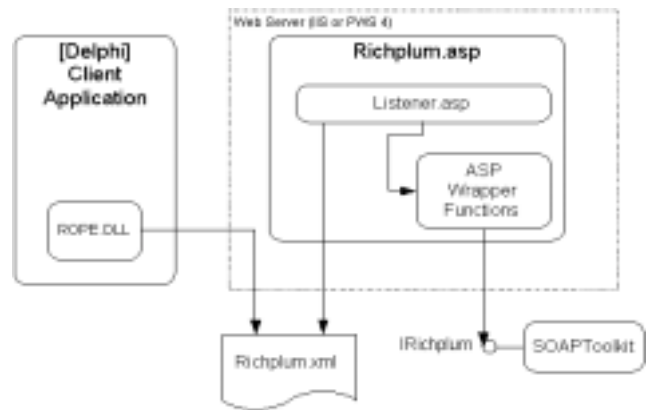


Figure 6

Step 2 – A Delphi client

We’re going to implement a Delphi client that acts as a SOAP consumer; however there will be two distinct versions.

The first version will require the ROPE.DLL Type Library – choose the Project menu followed by Import Type Library, then select the Rope 1.0 Type Library as shown in Figure 7.

The second version will use **late binding** to execute the remote SOAP methods. We’ll still use ROPE.DLL, but this time everything will be handled through an instance of the **ROPE.Proxy** object.

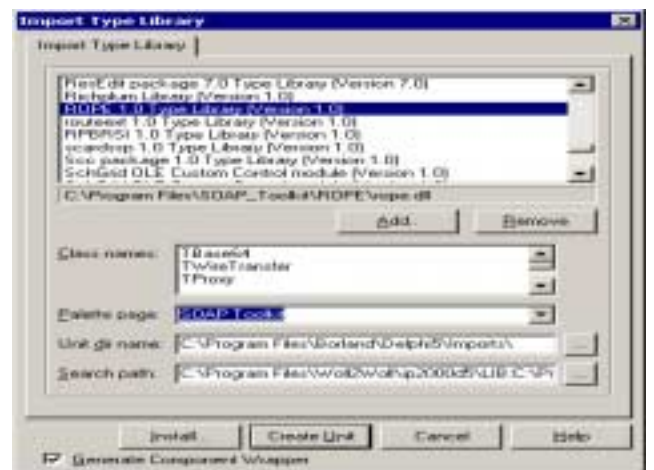


Figure 7

The ROPE Type Library exposes seven objects (those in bold will be used in the example application):

- **TProxy** is considered by Microsoft to be the primary client-side interface for access SOAP endpoints. We’ll see an example of the Proxy object later in this article.
- **TSOAPPackager** is able to package SOAP requests; it’s also able to “unpack” SOAP Responses.
- TServiceDescriptors
- TSDMethodInfo
- TSDParameterInfo
- TSDEndPointInfo
- **TWireTransfer** is responsible for sending the SOAP Request over “the wire”. Essentially, it provides all the HTTP GET and POST interactions that are required.

Figure 8 depicts a sample SOAP consumer. Clicking on the **Get Customer List** button sends a SOAP Request that returns a comma-separated list of the customer numbers found in the DBDEMOS Customer.db table – which then populates the **Customer List** list box with those values. Whenever the user clicks on a customer number, the **Customer Detail** is populated, but only after another SOAP Request is sent. At each stage, the SOAP Request and Response memo fields display the current SOAP transaction messages.

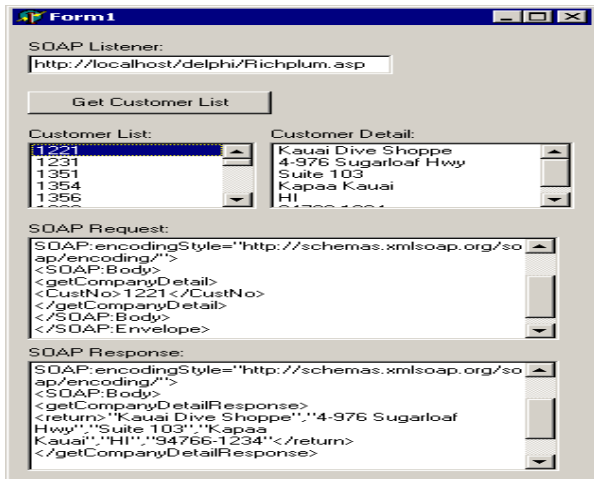


Figure 8

```

procedure TForm1.btnGetCustListClick(Sender:
                                TObject);

var sRequestStruct,
    sRequestPayload,
    sResponsePayload,
    sTemp : String;
begin
    SOAPPackager.LoadServicesDescription
    (icURI,
     'http://localhost/delphi/Richplum.xml', '');

    If SOAPPackager.
        IsMethodAvailable('getCustomerList') = 1
    Then Begin
        // Prepare the SOAP Request for the
        // getCustomerList method...
        SOAPPackager.SetPayloadData(icREQUEST, '',
        'getCustomerList', sRequestStruct);
        sRequestPayload :=
            SOAPPackager.GetPayload(icREQUEST);

        memoRequest.Text := sRequestPayload;

        // Send the SOAP Request over "the wire"
        // using HTTP...
        WireTransfer1.AddStdSOAPHeaders(edtSOAPListener.Text,
        'getCustomerList', Length(sRequestPayload));

        sResponsePayload:=WireTransfer1.PostDataToURI
        (edtSOAPListener.Text, sRequestPayload);

        SOAPPackager.SetPayload(icRESPONSE,
        sResponsePayload);

        memoResponse.Text := sResponsePayload;

        sTemp:=SOAPPackager.GetParameter(icRESPONSE,
        'return');

    End;

    Listbox1.items.commatext := sTemp;
end;

```

Listing 3

Listing 3 provides the code behind the Get Customer List button.

The single instance of the SOAPPackager object is worthless until a Service Description file – in this case the Richplum.xml SDL file - is loaded. We are now able to interrogate the SOAPPackager object and determine which methods are available. SOAPPackager.IsMethodAvailable(...) does just this.

SOAPPackager.SetPayloadData(...) is used to create the **Body** element of the SOAP message.

The SOAPPackager.GetPayload(...) method can be used to extract the **Envelope** element, i.e. the entire SOAP message. For example, calling GetPayload(icREQUEST) for the getCustomerList method returns the following XML:

```

<?xml version="1.0"?>
<SOAP:Envelope xmlns:SOAP="http://
    schemas.xmlsoap.org/soap/envelope/"
SOAP:encodingStyle="http://schemas.xmlsoap.org/
soap/encoding/">
<SOAP:Body>
<getCustomerList>
</getCustomerList>
</SOAP:Body>
</SOAP:Envelope>

```

Whereas, after the SOAP Request has been sent and a Response has arrived, a call to GetPayloadData(icRESPONSE) returns the following XML:

```

<?xml version="1.0"?>
<SOAP:Envelope xmlns:SOAP="http://
    schemas.xmlsoap.org/soap/envelope/"
SOAP:encodingStyle="http://schemas.xmlsoap.org/
soap/encoding/">
<SOAP:Body>
<getCustomerListResponse>
<return>1221,1231,1351,1354,1356,1380,1384,1510,1513,1551,1560,1563,
1624,1645,1651,1680,1984,2118,2135,2156,2163,2165,2315,2354,
2975,2984,3041,3042,3051,3052,3053,3054,3055,3151,3158,
3615,3984,4312,4531,4652,4684,5132,5151,5163,5165,5384,5412,
5432,5515,6215,6312,6516,6582,6812,9841
</return>
</getCustomerListResponse>
</SOAP:Body>
</SOAP:Envelope>

```

The call to Wiretransfer1.AddStdSOAPHeaders(...) adds an extra HTTP header: SOAPAction. The HTTP header for getCustomerList is shown below:

```

POST /delphi/Richplum.asp HTTP/1.0
User-Agent: MSDNWS
Content-Type: text/xml
Content-Length: 242
SOAPAction: http://localhost/delphi/
Richplum.asp#getCustomerList

```

By examining the SOAPAction header, Richplum.asp can determine which method is to be executed.

Calling WireTransfer1.PostDataToURI(...) generates an HTTP POST operation (synchronous), sending the SOAP Request to the URI specified. PostDataToURI(...) then waits until a SOAP Response is generated or a timeout occurs.

The SOAPPackager object allows us to extract information from the SOAP Response. If you examine the SOAP Response above, you'll notice a <return> element. The GetParameter(...) method allows us to extract the data held in a specified element:

```
// Extract the 'return' element
sTemp := SOAPPackager.GetParameter(icRESPONSE,
'return');
```

Using the ROPE objects has proved to involve a reasonable amount of work, however it does give us the fine granularity that some applications require. We'll now move on to the ROPE Proxy object – which is considerably easier to implement!

Using ROPE.Proxy

The code required to execute our SOAP Requests via a proxy is shown in Listing 4, as you can see, there's not much to it. This code creates an instance of **ROPE.Proxy**. It then loads the Richplum SDL file, after which our client application (SOAP consumer) can then treat the remote methods (web services) as if they were provided by a local COM object.

```
procedure
TForm2.btnGetCustListProxyClick(Sender:
                                TObject);
var sBuffer : string;
var objProxy : OleVariant;
begin
    objProxy := CreateOleObject('Rope.Proxy');

    // icURI = $00000001;
    objProxy.LoadServicesDescription(1,
'http://localhost/delphi/richplum.xml', '');

    sBuffer := objProxy.getCustomerList;

    Listbox1.items.commatext := sBuffer;
end;
```

Listing 4

Summary

Over the course of this article I've demonstrated that it's possible to write code once, and to share that code with other, possibly non-Win32, platforms. It's perfectly possible to have a Linux client application execute methods against a Win32 server process running Internet Information Server.

However, using the SOAP Toolkit as part of your Delphi client solution forces the use of ROPE.DLL. This immediately limits your client platform to Windows 2000. Not all is lost; we can still use the SOAP Toolkit on our Win32 server. Implementing a SOAP client that can communicate with a SOAP Toolkit Listener is a fairly straightforward task – thanks largely to the fact that SOAP is currently implemented using HTTP. SOAP libraries are popping up for most of the non-Win32 platforms.

Ultimately true vendor-interoperability will be the key to the success of SOAP. IBM and Microsoft, despite working on the SOAP specification together, both have their own implementations of SOAP itself. As you can imagine, the first releases on their respective implementations "didn't talk to each other". Over the coming months we can expect to see greater convergence between SOAP implementations – it won't be long before we see a Linux application built from methods being served up by a Win32 server.

All of the source code for this article is available from my web site-site: <http://www.isleofjura.demon.co.uk/bug>.

Resources

The SOAP Toolkit: <http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/000/242/msdncompositedoc.xml>

The SOAP Specification:
<http://msdn.microsoft.com/xml/general/soaptemplate.asp>

The Microsoft XML Developer Center:
<http://msdn.microsoft.com/xml/>

The .NET framework: <http://msdn.microsoft.com/net/>

The BizTalk web-site: <http://www.biztalk.org>

Additional reading:
<http://www.asptoday.com/articles/20000718.htm>
<http://www.asptoday.com/articles/20000728.htm>
<http://www.asptoday.com/articles/20000821.htm>

Newsgroups worth visiting:
microsoft.public.xml.soap
microsoft.public.msdn.soaptoolkit



Craig works as an Enterprise Developer (and Dilbert Evangelist!) for Currie & Brown (<http://www.currieb.com>) – their primary business is quantity surveying, cost management and project management. He can be reached via e-mail at: Craig.Murphy@currieb.co.uk or Craig@isleofjura.demon.co.uk



Angus' Nifty Tips

If you look at the background of Internet Explorer's toolbar, you'll see a very low profile bitmap. You can actually change this bitmap just like you can change Window's background (well, maybe not so easily!).

Run RegEdit Select:

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Toolbar

Run "Internet Explorer"

Select "Tools | Internet Options | General | Colors"

Uncheck "Use Windows colors" checkbox

Switch back to the Registry Editor

Add a new string value called BackBitmap as a "REG_SZ" type:

The value of the "BackBitmap" parameter should be set to the path to your bitmap (C:\WINNT\SETUP.BMP for example)

Restart Internet Explorer

Note that the colours you select for your bitmap can easily mask out the existing icons, so light pastel shades are recommended.