Writing SOAP applications in Delphi

Michaël Van Canneyt

August 15, 2001

1 What is SOAP ?

With the release of Delphi 6, Borland introduced support for Web-Services. Web-services is - arguably - one of the hottest topics of the moment in the internet community; It is also one of the cornerstones of Microsoft's .NET initiative. Web-services are small server programs that provide well-defined information to clients over a network such as the internet.

SOAP stands for *Simple Object Access Protocol*, and is a protocol to exchange information in a distributed environment. In other words, it is a new, platform independent way to handle remote object instantiation and remote procedure calls. It describes a communication protocol between a calling application (the client) and the called object (the server). The protocol uses XML to describe the request from the client and the response from the server. As such, the SOAP specification is equivalent to the CORBA (maintained by the OMG (Object Management Group) and DCOM standards (From Microsoft) for remote object instantiation.

SOAP describes just a protocol to handle (encode) requests and answers between the client and server; It does not specify how these requests and responses should be communicated; This is handled by so-called bindings: SOAP itself only specifies a HTTP binding. (HTTP is the communication protocol used between a web-server and a web-browser such as Netscape). There are also other bindings, but these are not yet supported by Delphi.

Typically, a request from a client to a server using SOAP would look as follows:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
<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:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The first four lines of this request are part of the HTTP binding; they are part of the HTTP protocol. The actual request starts with the `<SOAP-ENV:Envelope>` tag. It is a (fictional) request for the trade price of the `DIS` stock option (a method called GetLastTradePrice, which accepts an argument called `Symbol`, representing the ticker symbol for the stock option.

Once this request has been received by the HTTP server, it is processed, and an answer is sent back:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<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:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The first 3 lines are part of the HTTP protocol, and indicate that the request was handled succesfully. The rest is the answer of the web-service, encoded as prescribed by the SOAP protocol. It indicates that the current price for the stock option is 34.5 (no currency is specified)

Fortunately, it is not necessary to understand how this protocol works to write SOAP applications; these details are handled completely by Delphi. The above examples serve to show what SOAP actually does.

2 What is WSDL ?

SOAP only describes how the requests and answers between client and server should be encoded. It does not say how to describe the capabilities of the server, i.e. it does not give a protocol for the description of the calls that the server supports. This is handled by WSDL: *Web Services Description Language*. This is a protocol that uses XML to describe the requests that a server supports. This includes the actual request names, but also describes the parameters for the request, and the types of these parameters.

Part of the WSDL description of the stock-quote server application could look as follows:

```
<types>
<schema targetNamespace="http://example.com/stockquote.xsd"
        xmlns="http://www.w3.org/2000/10/XMLSchema">
  <element name="TradePriceRequest">
    <complexType>
      <all>
        <element name="tickerSymbol" type="string"/>
      </all>
    </complexType>
  </element>
  <element name="TradePrice">
    <complexType>
      <all>
        <element name="price" type="float"/>
      </all>
    </complexType>
  </element>
```

```
</schema>
</types>
<message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice"/>
</message>
<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>
```

The fist part of this XML document introduces some types (the `<types>` tag), and the second part describes the actual call (the `<operation>` tag).

People that create a web-service, should create and publish a WDSL document to describe the capabilities of their web-service to make it publicly accessible. The documents are often published on Internet. Delphi can use this document to create an interface to a web-service. There exist lists of web-services descriptions, which can be used to look up a service if one is needed (see the resources section). When a web-service is created using Delphi, the generated application can also generate the WDSL document for the developer, and for anyone wishing to use the service.

Delphi support for SOAP: Web-services

Delphi (Enterprise edition) supports creation of SOAP clients and of SOAP servers as part of the so-called Web-Services. In order for this to work, however, Microsoft's XML libraries (MSXML) must be installed. If it is not installed, Delphi will generate an error when any of the Webservices wizards are used. The XML libraries can be downloaded for free from Microsoft's site (see the resources section). In order for the Microsoft XML libraries to work, Internet Explorer 4.1 or higher must be installed on your windows system (which was unfortunately not the case for the Author's system).

When developing a SOAP server, a web-server should be installed on the development machine; Apache is a freely available web-server, which can be used for this purpose.

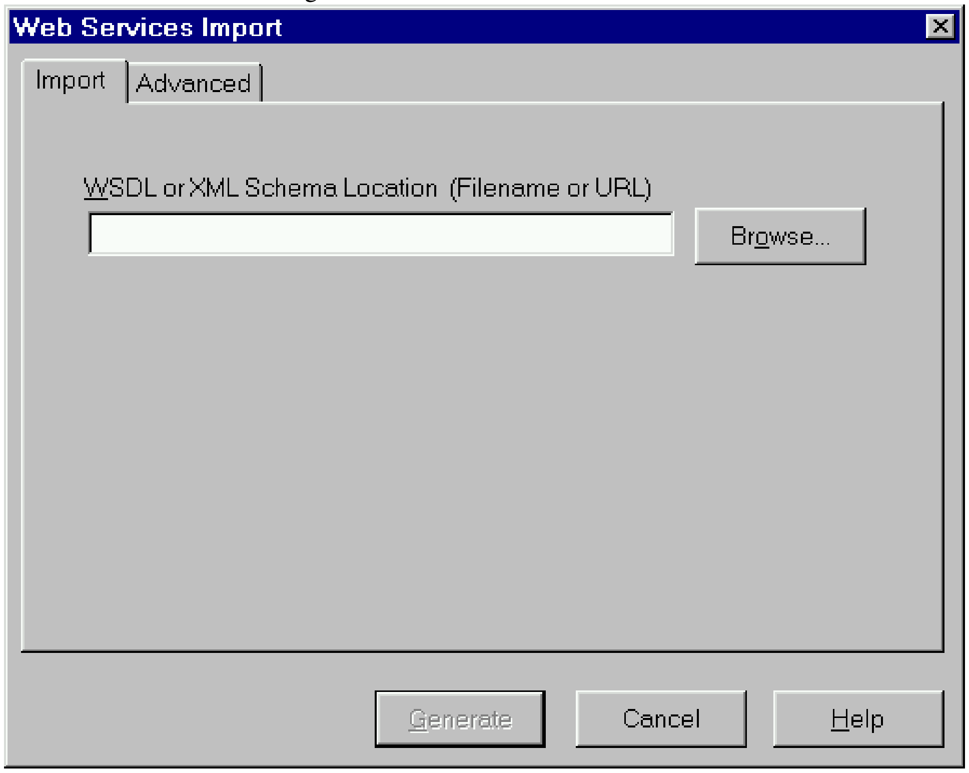
Once everything is installed correctly, creation of SOAP clients and servers is deceptively easy. All the gore details of SOAP and WSDL are completely hidden by Delphi. Creating SOAP client and Servers mainly involves the creation of a few interface definitions, and even this process an be handled by the IDE.

The key to the simplicity of all this is the use of a new interface: `IInvokable`. This interface is almost empty, but has been compiled in the `{$M+}` state, meaning that Run-Time Type Information will be generated for `IInvokable` and all its descendents. This RTTI information can then be used to construct SOAP messages and WSDL documents, without any efforts by the programmer.

3 Creating a SOAP client

When creating a SOAP client, essentially an interface definition must be created for the methods offered by the server application. Depending on whether the server was created

Figure 1: The babel fish service client



using Delphi or not, the declaration of this interface can be obtained in one of three ways:

- If the server was created using Delphi, then the unit that was used to create the interface of the server can be included in the client source, and that is it.
- If the server was not created using Delphi, or the unit is not available (e.g. when the server was developed by someone else), then the WSDL document describing the capabilities of the server can be used to create the interface.
- The interface can be hand-coded if the server capabilities are known.

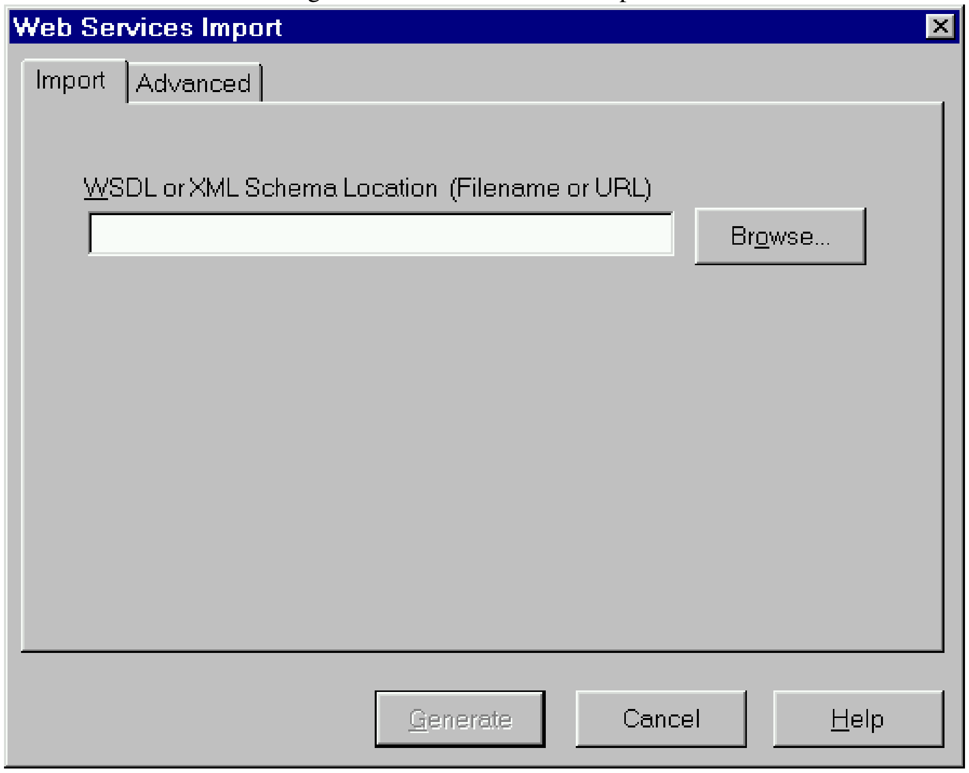
In all three cases, the interface must be descendent from `IInvokable`. IInvokable is an empty interface, whose sole purpose is the generation of run-time type information.

For the purpose of this section, the second method will be used. The web-service that will be accessed is the SOAP interace to the *BabelFish* service of AltaVista. This interface allows to translate a string from english to a set of non-english languages, or vice-versa. The WSDL for the BabelFish can be found e.g. on:
`http://www.xmethods.net/sd/2001/BabelFishService.wsdl`

Armed with this URI, the development of the client can be started. A form is created with two memos and two combo-boxes; the memos will contain the text to be translated, and the translated text. The comboboxes will contain the names of the source and target language. Adding an additional button to translate the text finishes the creation of the form. The form might look as in figure 1 on page 4.

When this is done, the interface for the web-service must be created. This can be done by selecting the 'Web Services Importer' wizard from the 'Web services' tab under the 'File|New|Other' menu. The wizard looks as in figure 2 on page 5.

Figure 2: The Web Services Importer



When this wizard opens, the URI for the BabelFish WSDL document can be filled in. Pressing the OK button will fetch this document from the internet, and will generate an `Interface` declaration.

The interface declaration for the `BabelFish` interface is quite simple:

```
Unit babelintf;

interface

uses Types, XSBuiltIns;

type

BabelFishPortType = interface(IInvokable)
  ['{2EB2E9D7-917A-11D5-8BC8-00409522C25A}']
  function BabelFish(const translationmode: WideString;
                     const sourcedata: WideString): WideString;  stdcall;
end;

implementation

uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(BabelFishPortType), '', '');
end.
```

The call to `InvRegistry.RegisterInterface` is needed so the VCL knows how to create an interface of the `BabelFishPortType` type.

Now the program 'knows' the interface of the BabelFish service. To call the service, it is necessary to drop a `THTTPRio` component (from the WebServices tab in the component palette) on the form. This component will do the actual call to the BabelFish service; This component is the entry to the SOAP mechanism of a SOAP client. The component itself does nothing special by itself, it just provides a mechanism so it can be typecasted to a known IInvokable interface, which will allow to call the methods of the web-service. To do this, it needs to know about the web-service that will be called, and some properties are provided for this.

When the component has been dropped on the form (call it `HPBabel`), It is necessary to specify which service should be used for this component. This can be done in one of two ways:

1. For services that are not written in Delphi, the `WSDLLocation`, `Service` and `Port` properties can be filled in.
2. For services that are written in Delphi, the URL of the service can be filled in. Delphi will then call the application to get the needed interface information from the service itself.

These methods are mutually exclusive. For the BabelFish service, this means that the `WSDLLocation` property must be filled in using the same value as was used in the Web-Service Import Wizard. Once this property is filled in, the `Service` property can be selected from a drop-down, and the `Port` type as well. (These values are retrieved from the WSDL document; This means that the document must be on a local disk or must be accessible through the internet at design-time)

Now the `THTTPRio` component can be used to access the web-service. In the `OnClick` handler of the `BTranslate` button, the following code is inserted:

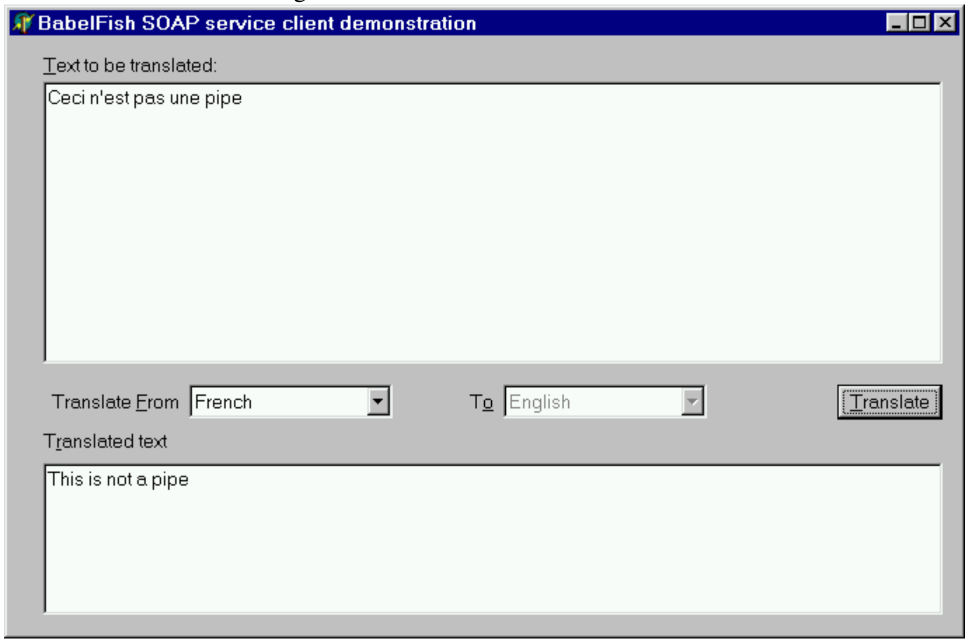
```
Var
  FromTo : String;

begin
  Screen.Cursor:=crHourGlass;
  Try
    Mto.Lines.Clear;
    Application.ProcessMessages;
    FromTo:=LanguageCodes[TBabelLanguage(CBFrom.ItemIndex)]+'_'
            +LanguageCodes[TBabelLanguage(CBTo.ItemIndex)];
    MTo.Text:=(HPBabel as BabelFishPortType).BabelFish(FromTo,MFrom.Text)
  finally
    Screen.Cursor:=crDefault;
  end;
end;
```

Since the operation goes over the web, and may take some time to complete, the cursor is switched to the hourglass cursor for the duration of the action; After that the string that describes the translation mode (from language - to language) is constructed based on the values set in the combo boxes. The TBabelLanguage is an enumerated type that enumerates all supported languages, and `LanguageCodes` is an array of strings that contain the codes for each language. All this is used to combine a translation code which could look for instance like `'en_fr'`, which means a translation from english to french.

Afer that is done, the web-service is actually called:

Figure 3: The BabelFish client in action


BabelFish SOAP service client demonstration

Text to be translated:

Ceci n'est pas une pipe

Translate From: French

To: English

Translated text

This is not a pipe

```
MTo.Text:=(HPBabel as BabelFishPortType).BabelFish(FromTo,MFrom.Text)
```

The `BabelFishPortType` is the interface that was generated by the Web Service Import Wizard. When this line is executed, a lot of work is done behind the scenes: Delphi will convert the `BabelFish` method call to a SOAP request (based on the RTTI information that it has generated for the `BabelFishPortType` interface), make a connection to the web-service, send the request, retrieve the SOAP answer, and decode it to determine any output parameters, and the result is then returned to the application. An example can be seen in figure 3 on page 7.

As far as the SOAP part of the application is concerned, this is the *only* line of code which has to be typed by the programmer; The rest is concerned with the GUI: There is some code for filling out the items in the comboboxes, and some code which prevents the user from selecting 2 non-english languages for target and source language; The code that pertains to SOAP is limited to the declaration of an interface (generated by the Delphi IDE) and the single line of code to retrieve the translation. No more detailed knowledge is required to create a SOAP web-service client.

4 Developing A SOAP server

A SOAP server written in Delphi is always a web-application (a webmodule); This can be a stand-alone cgi application, or a server module (for Apache, IIS or Netscape Server).

This has as a consequence that the SOAP server module is instantiated when a request comes in via the web-server, and that it is destroyed when the request has been handled.

If two requests come in from the same client, then it will be handled by 2 different instances of the SOAP server module, i.e. a SOAP server module is stateless: Any information that was gathered during the treatment of the first request is lost when the second request is handled; Unless measures are taken by the programmer, and a special API is constructed to foresee and take care of this kind of situation; In other words, some kind of session management must be provided by the programmer if some client information must be stored on the server in between calls.

Consider the following example: A SOAP server offers an interface to order tickets for a concert. Since ordering tickets requires the gathering of quite some information, the API is split over different calls. As the client gathers information, the server should keep track of data that was entered at earlier requests from this particular client. To this end, the client starts a session on the server, and receives a unique session ID from the server. At each following request to the server, this session ID is passed on to uniquely identify the client; the server can use this session ID to store data about the client as the process of ordering a ticket continues. When the client is done, the session is destroyed, and the server removes all information gathered in the session. For a web browser and web server, this is done using cookies.

For a SOAP service, the programmer should foresee some kind of session management. In this section, a SOAP server will be developed that manages a user session. This will show how a SOAP server works, and will at the same time tackle an important problem with SOAP servers, namely the session management.

The Borland community website offers an article that shows how to do this, using a Client-Dataset; Here an alternative approach will be presented, which needs no datasets, but uses .ini files to store session data. This will result in a more leightweight application, since the whole database management and dependence on Midas will drop out.

The soap server will offer a `ISessionManager` service. This service will allow to

1. Start a session by asking a user to log in with username and password. If the user is correctly logged in, a session ID is returned. This is a GUID identifier generated by windows.
2. End a session. The session to end must be identified with the session ID.
3. Set a key to a certain value. The session ID must be passed to the server, as well as the name and value of the key.
4. Retrieve the value of a certain key. The session ID must be passed to the server, as well as the name of the key whose value should be retrieved.
5. As a control mechanism, a client can ask whether a specific Session ID is valid.

The server will maintain a .ini file per session in a specific directory; The name of the session .ini file will be simply the Session ID. When the session is started, the Session ID is written to the file under the key name `SessionID`. When the session is ended, the session .ini file will be deleted. Checking whether a specific session is still valid will be done by checking whether the .ini file exists.

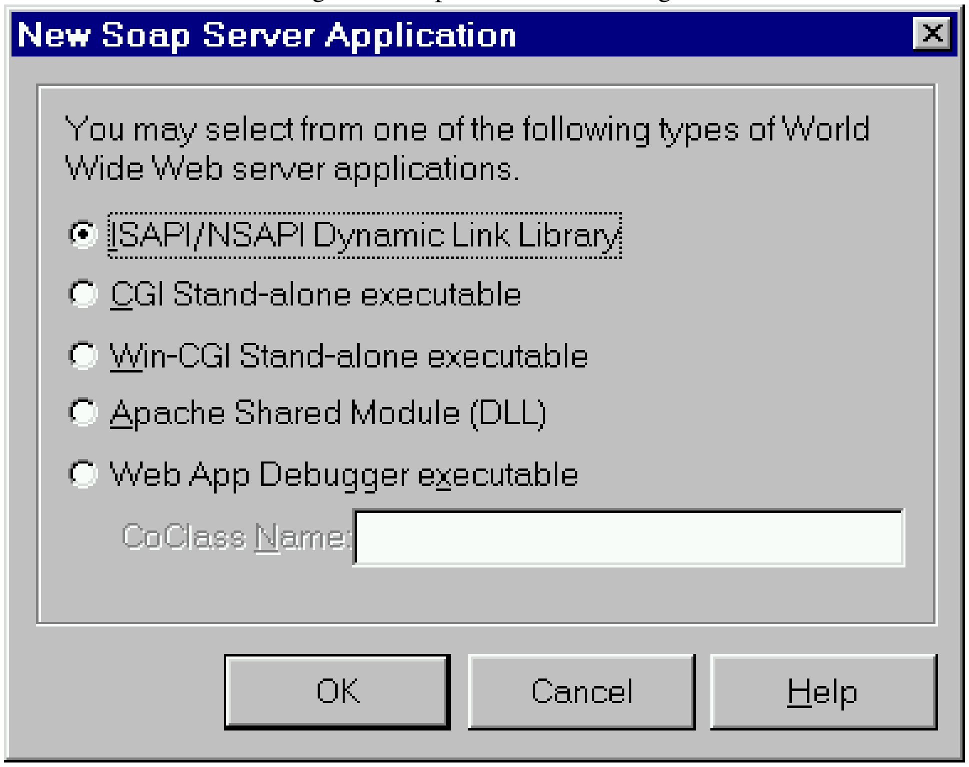
Getting and setting key values will then simply be the getting and setting of key values in the .ini file that corresponds to the Session ID.

The SOAP server here will be developed as a web-server CGI executable; This executable should run under most web servers on Windows (indeed, on Linux as well when compiled with Kylix.)

5 The SOAP server interface

To start a SOAP server, select the 'Soap Server Application' wizard from the 'Web services' tab under the 'File|New|Other' menu. The wizard will present a dialog that allows to choose the type of SOAP server, which should look more or less as in figure 4 on page 9.

Figure 4: Soap server wizard dialog.


New Soap Server Application

You may select from one of the following types of World Wide Web server applications.

[✓ ISAPI/NSAPI Dynamic Link Library]

CGI Stand-alone executable

Win-CGI Stand-alone executable

Apache Shared Module (DLL)

Web App Debugger executable

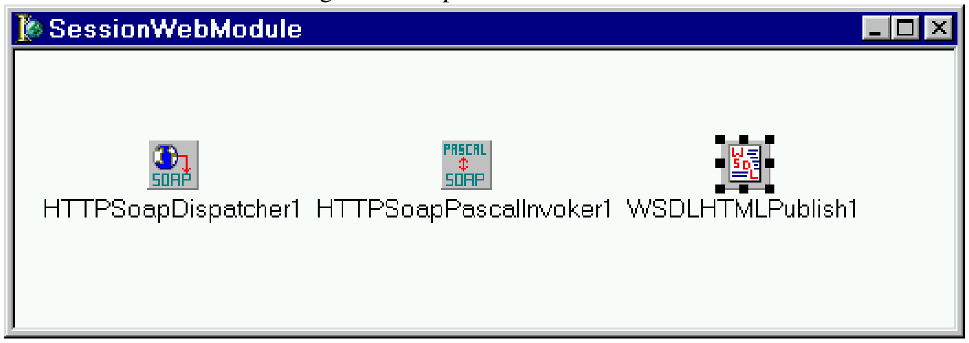
CoClass Name:

This is simply the kind of web-server application that will be created. After choosing the type of web-server application, the IDE will create a new web module and project (a console application in this case, which has been called smws). The web module looks as in figure 5 on page 9 On this web-module, the IDE has placed three components:

THTTPSoapDispatcher This component registers itself with the web module; it is responsible for translating the web-server's CGI request to SOAP requests, It also provides some additional functionality to allow publishing the WSDL document.

THTTPSOAPPascalInvoker This component is used by the `THTTPSOAPDispatcher` component to translate the SOAP request to a call to pascal code: It translates the SOAP request, looks up the interface that corresponds to the request, creates the object that implements the interface, calls then the required method, and returns the

Figure 5: Soap server web module.



result as a SOAP answer. This component does the actual work for the SOAP server.

TWSDLHtmlPublish This component creates the WSDL document for the services offered by the web-server. It also registers itself with the web-module. This component allows the web-service to publish its own WSDL document, and is used by the Delphi IDE to retrieve the WSDL documents from Delphi-created web-services.

To create the SOAP service, one must define and register one or more interfaces that descend from `IInvokable`. The registration process makes ensures that the `THTTPSoapPascalInvoker` knows the various interfaces or services that can be called when the SOAP service is called upon. Likewise, the registration process allows the `TWSDLHtmlPublish` component to produce the WSDL pages that correspond to the various interfaces or services known to the SOAP server.

Definition of the interface for the session manager service is quite easy, and is implemented in the ISMintf unit:

```
unit ISMintf;

interface

Type
  ISessionManager = Interface(IInvokable)
    ['{F10672DD-9254-11D5-8BCB-00409522C25A}']
    Function SessionStart(UserName,Password : String;
                          Out SessionID : String ): Boolean; stdcall;
    Function SessionEnd(SessionID : String): Boolean; StdCall;
    Function SetValue(SessionID,ValueName,Value : String): Boolean; StdCall;
    Function GetValue(SessionID,ValueName : String;
                      Out Value : String): Boolean; stdcall;
    Function SessionActive(SessionID : string) : Boolean; StdCall;
  end;

implementation

Uses
  InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(ISessionManager));
end.
```

The five calls defined in the `ISessionManager` interface correspond to the five methods supported by the session manager service as described above. The GUID that appears in the interface declaration was inserted by hitting the `CRTL-SHIFT-G` combination in the Delphi Code editor; this generates a unique GUID. and places it at the cursor location.

The call to `InvRegistry.RegisterInterface` is similar to the call to the same function in the BabelFish client interface definition; Indeed, this unit will be used in the client application as well. In the server it makes sure that the `THTTPSOAPPascalInvoker` component on the web-module knows about the `ISessionManager` service.

Now that the interface has been defined, a class must be created that supports this interface. (Remember that interfaces have no code associated with them) This will be done by creating a `TSessionManager` class, which is created in the `ISMimpl` unit:

```
unit ISMimpl;
```

```
interface

uses
  ISMintf,InvokeRegistry;

Type
  TSessionManager = Class(TInvokableClass,ISessionManager)
  public
    function SessionStart(UserName: String; Password: String;
      out SessionID: String): Boolean; stdcall;
    function SessionEnd(SessionID: String): Boolean; stdcall;
    function GetValue(SessionID: String; ValueName: String;
      out Value: String): Boolean; stdcall;
    function SessionActive(SessionID: String): Boolean; stdcall;
    function SetValue(SessionID: String; ValueName: String;
      Value: String): Boolean; stdcall;
  end;

implementation
```

By making the `TSessionManager` descendent from `TInvokableClass`, and registering it (this will happen in the initialization section of the unit) the `THTTPSOAPPascalInvoker` will know about this class and will instantiate it whenever it encounters a request for the `ISessionManager` service.

The implementation of the `TSessionManager` class is quite straightforward.

```
implementation

Uses FileCtrl,Windows,SysUtils,Classes,IniFiles,ActiveX;

{ TSessionManager }

Var
  BaseDir : String;
  LUsers  : TStringList;

Const
  FUserNames = 'username.ini';

Function GetSessionFileName(SessionID : String) : String;

begin
  // Cut braces.
  SessionID:=Copy(SessionID,2,Length(SessionID)-2);
  Result:=BaseDir+SessionId+'.ini';
end;
```

The `GetSessionFileName` converts a Session ID to a .ini filename. It does this by simply taking the base directory for the session data, and appending the session ID; Other mechanisms can be invented; choosing another mechanism means just adapting this function a bit, and the whole application will function as if nothing changed.

The `LUsers` stringlist object contains a list of `UserName=Password` pairs, and will be used to validate the users; This stringlist is initialized at the start of the application by reading the list contents from the file with the name `FUserNames`

The first method starts a new session. It searches the `LUsers` list and sees whether the username and password match a known user and password. If a matching entry is found, then a new session ID is created, and it is written to the session .ini file.

```
function TSessionManager.SessionStart(UserName, Password: String;
  out SessionID: String): Boolean;

Var
  I,J : Integer;
  N,P : String;
  GUID : TGUID;

begin
  I:=LUsers.Count-1;
  Result:=False;
  While (I>=0) and Not Result do
    begin
    P:=LUsers[i];
    J:=Pos('=',P);
    If J>0 then
      begin
      N:=Copy(P,1,J-1);
      Delete(P,1,J);
      end;
    Result:=(CompareText(N,UserName)=0) and (Password=P);
    dec(i);
    end;
  if Result then
    begin
    CoCreateGUID(GUID);
    SessionID:=GUIDToString(GUID);
    With TIniFile.Create(GetSessionFileName(SessionID)) do
      Try
        WriteString('Values','SessionID',SessionID);
      finally
        Free;
      end;
    end
  else
   SessionID:='';
end;
```

The `SessionActive` method simply checks whether the .ini file associated with a session ID exists:

```
function TSessionManager.SessionActive(SessionID: String): Boolean;
begin
  Result:=FileExists(GetSessionFileName(SessionID));
end;
```

The `SessionEnd` checks whether the session is still active, and if so, deletes the session's .ini file.

```
function TSessionManager.SessionEnd(SessionID: String): Boolean;
begin
  Result:=SessionActive(SessionID);
  If Result then
    Result:=DeleteFile(GetSessionFileName(SessionID));
end;
```

The `GetValue` and `SetValue` methods simply create a `TIniFile` object with the filename associated to the Session ID, and use then the read and write methods to get or set the desired keys. All values are written in the `'Values'` section of the .ini file.

```
function TSessionManager.GetValue(SessionID, ValueName: String;
  out Value: String): Boolean;

begin
  Result:=SessionActive(SessionID);
  If Result then
    begin
    With TIniFile.Create(GetSessionFileName(SessionID)) do
      Try
        Result:=ValueExists('Values',ValueName);
        If Result then
          Value:=ReadString('Values',ValueName,'');
      finally
        Free;
      end;
    end;
end;

function TSessionManager.SetValue(SessionID, ValueName, Value: String): Boolean;
begin
  Result:=SessionActive(SessionID);
  If Result then
    begin
    With TIniFile.Create(GetSessioNFileName(SessionID)) do
      Try
        WriteString('Values',ValueName,Value);
      finally
        Free;
      end;
    end;
end;
```

Finally, the initialization section registers the `TSessionManager` class, and initializes the `BaseDir` directory; This directory will contain all session .ini files. Some checks are done to ensure the base directory actually exists. After that, the list of users is created and filled from the appropriate file.

```
Initialization
  InvRegistry.RegisterInvokableClass(TSessionManager);
  BaseDir:=ExtractFilePath(Paramstr(0))+'SessionData';
  If Not DirectoryExists(BaseDir) Then
    begin
    If Not CreateDir(BaseDir) then
```

```
      BaseDir:=ExtractFilePath(Paramstr(0))
    end;
  If BaseDir[Length(BaseDir)]<>'\' then
    BaseDir:=BaseDir+'\';
  LUsers:=TStringList.Create;
  If FileExists(BaseDir+FUSerNames) then
    LUsers.LoadFromFile(BaseDir+FUserNames);

Finalization
  LUsers.Free;
end.
```

The finalization section frees the `LUsers` stringlist.

This concludes the creation of the SOAP server. After compiling the application, it is sufficient to copy it to a directory where the web-server can find it and execute it. In the case of Apache, this could be the cgi-bin directory under the Apache install directory.

6 A SOAP Session Manager client

Now that the server has been created, a client will be created to test the server.

To create the client, again the web-service interface definition must be created for the client. However, the interface is available, since it was already created for the server. The `ISMintf` unit can be used directly in the client to create the service definition.

The correct working of the SOAP server can be checked by calling the 'Web Services Import' wizard from the menu, and specifying the URL for the WSDL page of the created service. Since the SOAP server is capable of producing its own WSDL page, the following URL was entered in the edit box for the WSDL location:

```
http://localhost/cgi-bin/smws.exe/wsdl/ISessionManager
```

After hitting the OK button, the Delphi IDE should generate an interface definition that is an exact copy of the interface definition in the ISMintf unit.

For the client, the existing `ISMintf` unit will be used. So a new project is created, and the unit is added to it. The main form of the unit gets two edits to enter a username and password, and a button to log in or log out. A label to hold and display the session ID key is also added.

Separately, 2 edits are provided to enter a key name and key value, as well as 2 buttons that will get or set the key value.

Finally, a `THTTPRio` component is dropped to represent the connection to the client. The `WSDLLocation` property can be set to the above URL, and then the values of the `Service` and `Port` properties can be selected.

When all this is dropped on the form, it could look more or less like figure 6 on page 15.

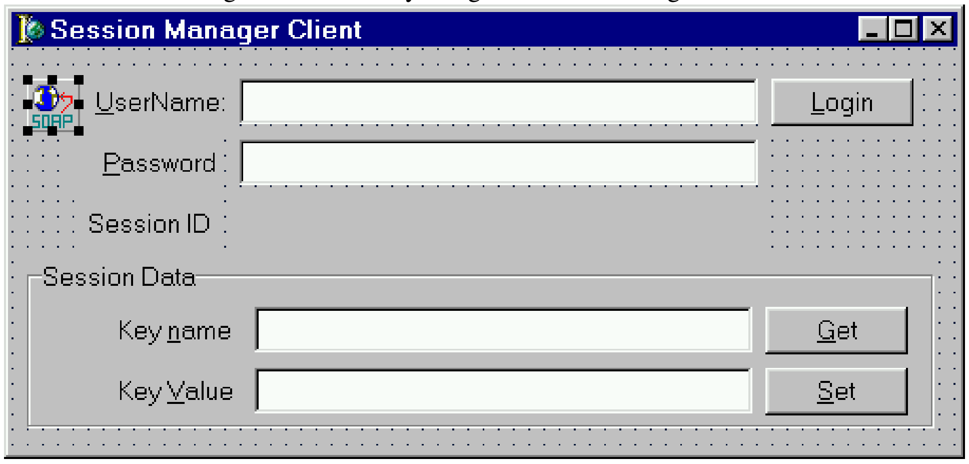
After this, the `OnClick` method of the login button is filled with the following code:

```
procedure TSessionManagerForm.BloginClick(Sender: TObject);

Var
  S : string;

begin
```

Figure 6: The newly designed session manager client



```
  With (HPSession as ISessionManager) do
    If (LSessionID.Caption='') then
      begin
      If SessionStart(EUserName.Text,EPassword.Text,S) then
        begin
        LSessionID.Caption:=S;
        BLogin.Caption:='&Logout';
        GBData.Enabled:=True;
        end
      else
        MessageDlg(SFailedToStartSession,mtError,[mbOk],0);
      end
    else
      begin
      If SessionEnd(LSessionID.Caption) Then
        begin
        GBData.Enabled:=False;
        LSessionID.Caption:='';
        BLogin.Caption:='&Login';
        end
      else
        MessageDlg(Format(SFailedToEndSession,[LSessionID.Caption]),
                   mtError,[mbOk],0);
      end;
end;
```

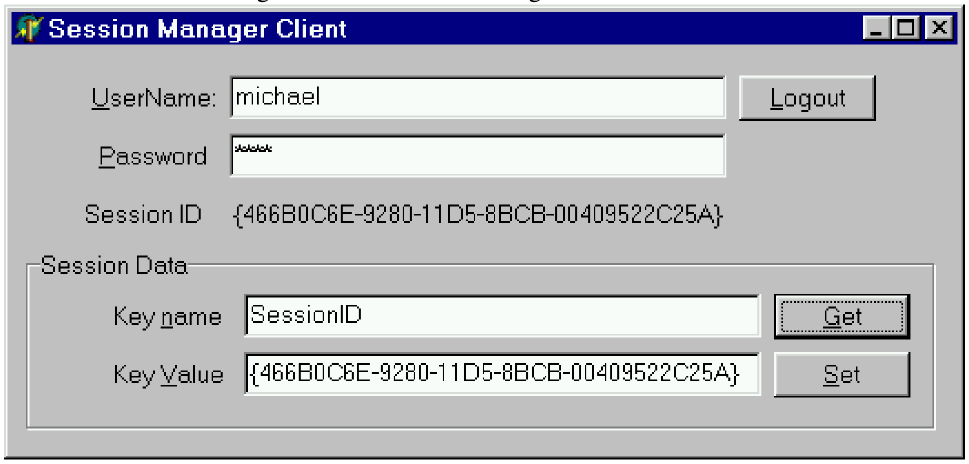
As can be seen, the code uses the `LSessionID` label's `Caption` property to determine whether it should log in or log out. Depending on what needs to be done, it uses the `SessionStart` or `SessionEnd` methods to log in or out, saving the Session ID when needed.

The button to set a value on the server gets the following onclick handler:

```
procedure TSessionManagerForm.BSetValueClick(Sender: TObject);

begin
  With (HPSession as ISessionManager) do
```

Figure 7: The session manager client in action.



UserName: michael

Password ****

Session ID {466B0C6E-9280-11D5-8BCB-00409522C25A}

Key name SessionID

Key Value {466B0C6E-9280-11D5-8BCB-00409522C25A}

```
    begin
    If Not SetValue(LSessionID.Caption,EKeyName.Text,EKeyValue.Text) then
      MessageDlg(Format(SFailedToSetValue,[EKeyName.Text]),mtError,[mbOK],0);
    end;
end;
```

Which is quite straightforward. The code for the retrieval of a value is similar:

```
procedure TSessionManagerForm.BGetValueClick(Sender: TObject);

Var
  S : String;

begin
  With (HPSession as ISessionManager) do
    begin
    If GetValue(LSessionID.Caption,EKeyName.Text,S) then
      EKeyValue.Text:=S
    else
      MessageDlg(Format(SFailedToGetValue,[EKeyName.Text]),mtError,[mbOK],0);
    end;
end;
```

And with this code, the client is ready to run.

When the client is run, the first thing that should be attempted is to log in. When the login succeeded, values can be set and retrieved, till the session is closed by logging out. After logging in, only one value is available; the Session ID, with Key Name `SessionID`. Note that to be able to log in, the name and password must match one in the list of known users on the server (an example file is provided)

In figure 7 on page 16 the result of a login and the retrieval of the SessionID is shown; Note that the value displayed in the edit box and the caption of the `LSessionID` label are the same.

While the client has a session running, the session data can be viewed by opening the session .ini file, which resides in the directory referenced by the `BaseDir` variable in the server.

7 Conclusion

The SOAP clients and servers introduced here may not be the most useful ones, but they are only examples, showing how easy the development of SOAP clients and servers is under Delphi. The SOAP server presented here tackles a problem of the SOAP support of Delphi, namely the fact that the server is stateless and does not preserve any information between invocations; The solution presented is conceptually simple, but is robust can be used as a starting point for a more functional SOAP web service; The idea of session management can be enhanced in several ways, but the essence of what it should do is present in the current implementation. Some ideas for enhancement could be

1. Another mapping between the Session ID and the used .ini file.
2. Management of users and passwords, support for multi-line values for stored keys.
3. Since a client can open a session but for some reason forget to close it, at regular intervals, a check could be done to remove 'old' session ini files, based on some timeout value. This check could be done at the start and/or end of each session. This would avoid a proliferation of session .ini files.
4. Retrieve a list of known key names for the session.

These are trivial improvements, which should not take much work to implement.

8 Resources

The following URI's can be used to find more information about SOAP, CORBA, DCOM and XML:

- The soap specification can be found at: `http://www.w3.org/TR/SOAP/`
- The WSDL specification can be found at: `http://www.w3.org/TR/wsdl`
- CORBA is described at `http://www.omg.org/`
- DCOM is described at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/`
- More information about Mircosofts XML support can be found at `http://msdn.microsoft.com/library/de`
- the XML libraries can be downloaded from `http://msdn.microsoft.com/downloads/default.asp?URL`

To find public web-services, the following places can be used:

- XMethods Web Services Directory: `http://www.xmethods.net/`
- IBM XML Registry: `http://www.alphaworks.ibm.com/tech/xrr`
- Collection of SOAP Implementations: `http://www.soapware.org/directory/4/implementations`
- SalCentral WSDL / SOAP Web Services Search Engine: `http://www.salcentral.com/salnet/webservic`

The Babelfish interface description can be found in e.g. `http://www.xmethods.com/detail.html?id=14`