HOW TO IMPLEMENT WEB SERVICES FOR INDY

Editor:
Tomáš Mandys, tomas.mandys@2p.cz (2p plus)

Home site:
http://www.2p.cz

Document status:
Version 1.0 First release

Table of Contents

Disclaimer

1. Introduction

Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications.

Web Services are designed to allow a loose coupling between client and server. That is, server implementations do not require clients to use a specific platform or programming language. In addition to defining interfaces in a language-neutral fashion, they are designed to allow multiple communications mechanisms as well.

Support for Web Services is designed to work using SOAP (Simple Object Access Protocol). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. It uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol. For more information about SOAP, see the SOAP specification available at http://www.w3.org/TR/SOAP/

Web Service applications publish information on what interfaces are available and how to call them using a WSDL (Web Service Definition Language) document. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard or command-line utility can import a published WSDL document, providing you with the interface definitions and connection information you need. If you already have a WSDL document that describes the Web service you want to implement, you can generate the server-side code as well when importing the WSDL document.

The components that support Web Services are available in all Delphi Professional bundles since version 6 and Kylix 3. Hence both Windows and Linux platforms are supported.

But Delphi SOAP server side libraries are developed for interaction with Microsoft Internet Information Server (ISAPI), Netscape Information Server (NSAPI) or Apache. They are based on *TWebApplication* class that in addition does not support more instances in one executable. Hence it is not supported creating SOAP server application based on small custom HTTP server implemented using Indy components *TIdHttpServer*.

Second disadvantage of Delphi SOAP libraries that is not supported secure HTTP communication (HTTPS protocol = HTTP + SSL) neither at server nor client side.

Let's demonstrate how to develop *TWebApplication* compliant class using Indy HTTP server and how to add HTTPS support.

2. SOAP

2.1. Server side

Since Delphi SOAP components are based on *TWebApplication* class it's necessary to write an ancestor of *TWebApplication* that will serve HTTP requests from *TIdHttpServer*. Such class named *TIdWebApplication* is implemented in *IdWebBrooker* unit that cooperates with *TIdWebModule* (ancestor of *TWebModule*). A real *TIdWebModule* defining published SOAP

interface method and assigned to *TIdWebApplication.WebModuleClass* automatically generates WSDL document describing supported interfaces. When a client calls a method *TWebModule* and its *SOAPInvoker* property calls automatically method implementation or a *TWebActionItem*. *TSOAPInvoker* class is defined in *SOAPPasInv2* unit.

Example:

The example code defines one published *IMyInterface.published_soap_method* method. *TMyWebModule* server does automatic invoking using *OnAction_published_soap_method* or *published_soap_method*.

```
type
  { define published interface }
  IMyInterface = interface(IInvokable) ['{CE958F31-9770-34B6-1177-51D0779D1891}']
    function published_soap_method(const I: Integer): Integer; stdcall;
  end;

type
  { implement interface methods }
  TMyWebModule = class(TIdWebModule, IMyInterface)
  private
    { method called by TWebActionItem – optional }
    function OnAction_published_soap_method(
                Sender: TObject;
                Request: TWebRequest;
                Response: TWebResponse;
                var Handled: Boolean);
    { method called by invoker – optional }
    function  published_soap_method(const I: Integer): Integer; stdcall;
  protected
    procedure InitModule; override;
  public
    constructor Create(aOwner: TComponent); override;
  end;

  TMyWebApplication = class(TIdWebApplication)
  private
  end;


{ TMyWebModule }

constructor TMyWebModule.Create(aOwner: TComponent);
var
  A: TWebActionItem;
begin
  inherited;
  A:= Actions.Add;
  A.Default:= True;
  A.Enabled:= True;
  A.PathInfo:= '/IMyInterface/published_soap_method';
  A.OnAction:= OnAction_published_soap_method;
end;

procedure TMyWebModule.InitModule;
begin
  inherited;
// do a custom initialization here
...
//  WSDLPublish.TargetNamespace:=
```

```
end;

procedure TMyWebModule.OnAction_published_soap_method(
            Sender: TObject;
            Request: TWebRequest;
            Response: TWebResponse;
            var Handled: Boolean);
begin
  showmessage('OnSendAction');
//  Invoke(InvClassType, IntfInfo, ActionMeth, Request, Response, BindingType);
//  Handled:= True;
end;

function TMyWebModule.published_soap_method (I: Integer): Integer;
begin
  showmessage('INVOKED send');
end;

var
  fMySOAPServer: TMyWebApplication;
begin
  fMySOAPServer:= TMyWebApplication.Create(Self);
  fMySOAPServer.HttpServer.DefaultPort:= 80;
  fMySOAPServer.WebModuleClass := TMyWebModule;
  fMySOAPServer.Run;
end;
```

2.2. Server side – SSL

TIdWebApplication provides public *HttpServer* property. Developer can customize it to accept HTTPS connection so that assigns an instance of *TIdServerInterceptOpenSSL* to the *HttpServer.Intercept* property to allow use of the HTTPS protocol. For the Windows platform, you must install the Indy OpenSSL support .DLL's available at http://www.intelicom.si to enable Secure Socket Layer support.

2.3. Client side – SSL

Delphi supports SOAP clients using *SOAPHttpClient* and *SOAPHttpTrans* units. These libraries use *WinInit* unit that is based on *wininet.dll* library on Windows or on internal instance of *TIdHttp* on Linux. HTTPS is not supported.

We implemented native Indy support into *IdSOAPHttpClient* and *IdSOAPHttpTrans* units. *TIdHTTPReqResp* provides *HttpClient* property of *TIdCustomHttp* class.

Developer creates *TIdHTTPRIO* instance and initializes *HTTPWebNode.HttpClient* property. Using of *TIdHTTPRIO* is absolutely the same as *THTTPRIO*. For more information see *Using Web Services* Delphi documentation.

Example

The example code demonstrates *TIdHTTPRio* initialization for SSL connection. Note that code slightly differs for Indy v.9 or v.10.

```
var
  SoapClient: TIdHTTPRIO;
  SSLRequired: Boolean;
begin
  SoapClient:= TIdHTTPRIO.Create(Self);
```

```
  with SoapClient.HTTPWebNode.HttpClient do
  begin
    AllowCookies:= True;
    HandleRedirects:= False;
    ProtocolVersion:= pv1_1;
  {$IFNDEF INDY10}
    MaxLineLength:= 16384;
    RecvBufferSize:= 32768;
  {$ENDIF}
    if not SSLRequired then
      begin  // HTTP
        {$IFDEF INDY10}
        CreateIOHandler(nil);
        {$ELSE}
        IOHandler.Free;
        IOHandler:= nil;
        {$ENDIF}
      end
    else
      begin  // HTTPS
        {$IFDEF INDY10}
        IOHandler:= TIdSSLIOHandlerSocketOpenSSL.Create(
                      SoapClient.HTTPWebNode.HttpClient);
        {$ELSE}
        IOHandler:= TIdSSLIOHandlerSocket.Create(SoapClient.HTTPWebNode.HttpClient);
        {$ENDIF}
        with {$IFDEF INDY10}
             TIdSSLIOHandlerSocketOpenSSL
             {$ELSE}
             TIdSSLIOHandlerSocket
             {$ENDIF}(IOHandler), SSLOptions do
        begin
          Method:= sslvSSLv2;   // select SSL method
          CertFile:= 'my.crt';  // assign certificate
          KeyFile:= 'my.key';   // assign private key
          RootCertFile:= 'root.crt';

       // select verification method for server certificate
          VerifyMode:= [sslvrfPeer, sslvrfFailIfNoPeerCert, sslvrfClientOnce];
          VerifyDepth:= 1;
          OnVerifyPeer:= DoOnVerifyPeer;
        end;
      end;
    end;
  end;
  SoapClient.HTTPWebNode.OnLog:= SoapClientOnLog;  // custom logging
// set correct ReadTimeout
  TIdTCPClient(SoapClient.HTTPWebNode.HttpClient).OnConnected:=
    HttpClientOnConnected;
end;

procedure TMyClient.HttpClientOnConnected(Sender: TObject);
var
  S: string;
begin
  with (Sender as TIdCustomHTTP)
       {$IFDEF INDY10}, TIdIOHandlerStack(IOHandler){$ENDIF} do
    ReadTimeout:= 1000;
end;
```

3. Source code

3.1. IdWebBroker.pas

```
unit IdWebBrooker;

interface
uses
  Classes, WebBroker, WebBrokerSOAP, IdHTTPServer, IdCustomHTTPServer, IdTCPServer,
HTTPApp, SOAPHTTPPasInv, WSDLPub
  {$IFDEF INDY10}, IdContext{$ENDIF};

type
  TIdSoapHTTPServer = class(TIdHTTPServer)
  protected
    procedure DoCommandGet({$IFDEF INDY10}AContext: TIdContext{$ELSE}AThread:
TIdPeerThread{$ENDIF}; ARequestInfo: TIdHTTPRequestInfo; AResponseInfo:
TIdHTTPResponseInfo); override;
  public
    constructor Create(aOwner: TComponent); override;
  end;

  TIdWebRequest = class(TWebRequest)
  private
    FIdHttpServer: TIdCustomHTTPServer;
    FIdRequest: TIdHTTPRequestInfo;
    FIdResponse: TIdHTTPResponseInfo;
    FPort: Integer;
    FReadClientIndex: Integer;
  protected
    { Abstract methods overridden }
    function GetStringVariable(Index: Integer): string; override;
    function GetIntegerVariable(Index: Integer): Integer; override;
    function GetDateVariable(Index: Integer): TDateTime; override;
  public
    constructor Create(IdHttpServer: TIdCustomHTTPServer; IdRequest:
TIdHTTPRequestInfo; IdResponse: TIdHTTPResponseInfo);
    destructor Destroy; override;
    { Abstract methods overridden }
    // Read count bytes from client
    function ReadClient(var Buffer; Count: Integer): Integer; override;
    // Read count characters as a string from client
    function ReadString(Count: Integer): string; override;
    // Translate a relative URI to a local absolute path
    function TranslateURI(const URI: string): string; override;
    // Write count bytes back to client
    function WriteClient(var Buffer; Count: Integer): Integer; override;
    // Write string contents back to client
    function WriteString(const AString: string): Boolean; override;
    // Write HTTP header string
    function WriteHeaders(StatusCode: Integer; const ReasonString, Headers: string):
Boolean; override;
    function GetFieldByName(const Name: string): string; override;
    property IdRequestInfo: TIdHTTPRequestInfo  read FIdRequest;
  end;

  TIdWebResponse = class(TWebResponse)
  private
```

```
    FIdResponse: TIdHTTPResponseInfo;
    FSent: boolean;
  protected
    { Abstract methods overridden }
    function GetStringVariable(Index: Integer): string; override;
    procedure SetStringVariable(Index: Integer; const Value: string); override;
    function GetDateVariable(Index: Integer): TDateTime; override;
    procedure SetDateVariable(Index: Integer; const Value: TDateTime); override;
    function GetIntegerVariable(Index: Integer): Integer; override;
    procedure SetIntegerVariable(Index: Integer; Value: Integer); override;
    function GetContent: string; override;
    procedure SetContent(const Value: string); override;
    procedure SetContentStream(Value: TStream); override;
    function GetStatusCode: Integer; override;
    procedure SetStatusCode(Value: Integer); override;
    function GetLogMessage: string; override;
    procedure SetLogMessage(const Value: string); override;
  public
    { Abstract methods overridden }
    procedure SendResponse; override;
    procedure SendRedirect(const URI: string); override;
    procedure SendStream(AStream: TStream); override;
  public
    constructor Create(Request: TWebRequest; Response: TIdHTTPResponseInfo);
    destructor Destroy; override;
    function Sent: Boolean; override;
    procedure SetCookieField(Values: TStrings; const ADomain, APath: string;
AExpires: TDateTime; ASecure: Boolean);
    property IdResponse: TIdHTTPResponseInfo  read FIdResponse;
  end;

  TIdWebApplication = class;

  TIdWebModule = class(TCustomWebDispatcher, IWebRequestHandler)
  private
    fSOAPDispatcher: THTTPSoapDispatcher;
    fSOAPInvoker: THTTPSoapPascalInvoker;
    fWSDLPublish: TWSDLHTMLPublish;
    fWebApplication: TIdWebApplication;
  protected
    function HandleRequest(Request: TWebRequest; Response: TWebResponse): Boolean;
    procedure InitModule; virtual;
  public
    property WebApplication: TIdWebApplication read fWebApplication;
    property SOAPDispatcher: THTTPSoapDispatcher read fSOAPDispatcher;
    property SOAPInvoker: THTTPSoapPascalInvoker read fSOAPInvoker;
    property WSDLPublish: TWSDLHTMLPublish read fWSDLPublish;
    constructor Create(aOwner: TComponent); override;
  end;

  TIdWebApplication = class(TWebApplication)
  private
    FHTTPServer: TIdHTTPServer;
  protected
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Run; override;
    function HandleRequest(Request: TWebRequest; Response: TWebResponse): Boolean; //
to be public
    property HTTPServer: TIdHTTPServer read FHTTPServer;
```

```
  end;

implementation
uses
  SysUtils, BrkrConst, IdHTTPHeaderInfo, IdHeaderList, Math, SOAPPasInv2;

{ TIdSoapHTTPServer }

constructor TIdSoapHTTPServer.Create(aOwner: TComponent);
begin
  inherited;
  FOkToProcessCommand:= True;
end;

procedure TIdSoapHTTPServer.DoCommandGet({$IFDEF INDY10}AContext:
TIdContext{$ELSE}AThread: TIdPeerThread{$ENDIF}; ARequestInfo: TIdHTTPRequestInfo;
AResponseInfo: TIdHTTPResponseInfo);
var
  Request: TIdWebRequest;
  Response: TIdWebResponse;
begin
  Request:= TIdWebRequest.Create(Self, ARequestInfo, aResponseInfo);
  try
    Response := TIdWebResponse.Create(Request, AResponseInfo);
    try
      TIdWebApplication(Owner).HandleRequest(Request, Response);
    finally
      Response.Free;
    end;
  finally
    Request.Free;
  end;
  inherited;
end;

{ TIdWebApplication }

constructor TIdWebApplication.Create(AOwner: TComponent);
begin
  inherited;
  FHTTPServer:= TIdSoapHTTPServer.Create(Self);
end;

destructor TIdWebApplication.Destroy;
begin

  inherited;
end;

procedure TIdWebApplication.Run;
begin
  inherited;
  fHTTPServer.Active:= True;
end;

function TIdWebApplication.HandleRequest(Request: TWebRequest; Response:
TWebResponse): Boolean;
begin
  Result:= inherited HandleRequest(Request, Response);
end;
```

```
{ TIdWebRequest }

constructor TIdWebRequest.Create(IdHttpServer: TIdCustomHTTPServer; IdRequest:
TIdHTTPRequestInfo; IdResponse: TIdHTTPResponseInfo);
begin
  FIdHttpServer:= IdHttpServer;
  FIdRequest := IdRequest;
  FIdResponse:= IdResponse;
  FPort:= IdHTTPServer.Bindings[0].Port; // IdHttpServer.DefaultPort;
  inherited Create;
end;

destructor TIdWebRequest.Destroy;
begin
  inherited;
end;

function StripHTTP(const Name: string): string;
begin
  if Pos('HTTP_', Name) = 1 then
    Result := Copy(Name, Length('HTTP_')+1, MaxInt)
  else
    Result := Name;
end;


function TIdWebRequest.GetFieldByName(const Name: string): string;
begin
{$IF gsIdVersion = '8.0.25'} // D6, K2 compatible
  Result := FIdRequest.Headers.Values[StripHTTP(Name)];
{$ELSE}
  Result := FIdRequest.RawHeaders.Values[StripHTTP(Name)];
{$IFEND}
end;

const
  viMethod = 0;
  viProtocolVersion = 1;
  viURL = 2;
  viQuery = 3;
  viPathInfo = 4;
  viPathTranslated = 5;
  viCacheControl = 6;
  viDate = 7;
  viAccept = 8;
  viFrom = 9;
  viHost = 10;
  viIfModifiedSince = 11;
  viReferer = 12;
  viUserAgent = 13;
  viContentEncoding = 14;
  viContentType = 15;
  viContentLength = 16;
  viContentVersion = 17;
  viDerivedFrom = 18;
  viExpires = 19;
  viTitle = 20;
  viRemoteAddr = 21;
  viRemoteHost = 22;
  viScriptName = 23;
  viServerPort = 24;
```

```
  viContent = 25;
  viConnection = 26;
  viCookie = 27;
  viAuthorization = 28;

function TIdWebRequest.GetDateVariable(Index: Integer): TDateTime;
begin
  case Index of
    viDate: Result:= FIdRequest.Date;
    viIfModifiedSince: Result:= FIdRequest.LastModified;
    viExpires: Result:= FIdRequest.Expires;
  else
    Result:= 0;
  end;
end;

function TIdWebRequest.GetIntegerVariable(Index: Integer): Integer;
begin
  case Index of
    viContentLength: Result:= FIdRequest.ContentLength;
    viServerPort: Result:= fPort;
  else
    Result:= 0;
  end;
end;

function TIdWebRequest.GetStringVariable(Index: Integer): string;
  function HeaderValue(S: string): string;
  begin
    {$IF gsIdVersion = '8.0.25'} // D6, K2 compatible
    Result := FIdRequest.Headers.Values[S];
    {$ELSE}
    Result := FIdRequest.RawHeaders.Values[S];
    {$IFEND}
  end;

  function GetScriptName: string;
  var
    SlashPos: Integer;
  begin
    Result := FIdRequest.Document;
    if Length(Result) > 0 then
    begin
      Delete(Result, 1, 1); // delete the first /
      SlashPos := Pos('/', Result);
      if SlashPos <> 0 then
        Delete(Result, SlashPos, MaxInt); // delete everything after the next /
      // Add back in the starting slash
          Result := '/' + Result;
    end;
  end;

begin
  case Index of
    viMethod: Result := FIdRequest.Command;  // ExtractFileName(FIdRequest.Command)
    viProtocolVersion: Result := FIdRequest.Version;
    viURL: Result := '';  // Not implemented
    viQuery:
      if FIdRequest.ContentLength > 0 then
        Result := FIdRequest.UnparsedParams
      else
```

```
        Result := '';
    viPathInfo: Result:= FIdRequest.Document;
    viPathTranslated: Result := FIdRequest.Document;  // Not implemented
    viCacheControl: Result := FIdRequest.CacheControl;
    viAccept: Result := FIdRequest.Accept;
    viFrom: Result := FIdRequest.From;
    viHost: Result := FIdRequest.Host;
    viReferer: Result := FIdRequest.Referer;
    viUserAgent: Result := FIdRequest.UserAgent;
    viContentEncoding: Result := FIdRequest.ContentEncoding;
    viContentType: Result := FIdRequest.ContentType;
    viContentVersion: Result := FIdRequest.ContentVersion;
    viDerivedFrom: Result := ''; // Not implemented
    viTitle: Result := ''; // Not implemented
    viRemoteAddr,
    viRemoteHost: Result := FIdRequest.RemoteIP;
    viScriptName: Result := ''; // GetScriptName;
    viContent: Result := FIdRequest.UnparsedParams;
    viConnection: Result := FIdRequest.Connection;
    viCookie: Result:= HeaderValue('Cookie');
    viAuthorization: Result := FIdRequest.Authentication.Authentication;
  else
    Result := '';
  end;
end;

function TIdWebRequest.ReadClient(var Buffer; Count: Integer): Integer;
begin
  Count := Max(Length(FIdRequest.UnparsedParams) - FReadClientIndex, Count);
  if Count > 0 then
    begin
      Move(FIdRequest.UnparsedParams[FReadClientIndex+1], Buffer, Count);
      Inc(FReadClientIndex, Count);
      Result := Count;
    end
  else
    Result := 0;
end;

function TIdWebRequest.ReadString(Count: Integer): string;
var
  Len: Integer;
begin
  SetLength(Result, Count);
  Len := ReadClient(Pointer(Result)^, Count);
  if Len > 0 then
    SetLength(Result, Len)
  else
    Result := '';
end;

function TIdWebRequest.TranslateURI(const URI: string): string;
begin
  Result := URI;
end;

function TIdWebRequest.WriteClient(var Buffer; Count: Integer): Integer;
var
  S: string;
begin
  Result := Count;
```

```
  try
    SetString(S, PChar(@Buffer), Count);
    FIdResponse.ContentText := S;
    FIdResponse.WriteContent;
  except
    Result := 0;
  end;
end;

type
  TIdHTTPResponseInfoCracker = class(TIdHTTPResponseInfo)
  end;

function TIdWebRequest.WriteHeaders(StatusCode: Integer;
  const ReasonString, Headers: string): Boolean;
begin
//  Result := True;
  TIdHTTPResponseInfoCracker(FIdResponse).FHeaderHasBeenWritten := True;
  Result := WriteString(Format('HTTP/1.1 %s'#13#10'%s', [ReasonString, Headers]));
end;

function TIdWebRequest.WriteString(const AString: string): Boolean;
begin
  Result := WriteClient(Pointer(AString)^, Length(AString)) = Length(AString);
end;

{ TIdWebResponse }

constructor TIdWebResponse.Create(Request: TWebRequest;
  Response: TIdHTTPResponseInfo);
begin
  inherited Create(Request);
  FIdResponse := Response;
  FIdResponse.ContentType:= 'text/xml';
end;

destructor TIdWebResponse.Destroy;
begin
  FIdResponse.ContentStream.Free;
  FIdResponse.ContentStream := nil;
  inherited;
end;

function TIdWebResponse.GetContent: string;
begin
  Result := FIdResponse.ContentText;
end;

procedure TIdWebResponse.SetContent(const Value: string);
begin
  FIdResponse.ContentText := Value;
  FIdResponse.ContentLength := Length(Value);
end;

function TIdWebResponse.GetLogMessage: string;
begin
  Result := '';  // N/A
end;

procedure TIdWebResponse.SetLogMessage(const Value: string);
begin
```

```
  // N/A
end;

function TIdWebResponse.GetStatusCode: Integer;
begin
  Result := FIdResponse.ResponseNo;
end;

procedure TIdWebResponse.SetStatusCode(Value: Integer);
begin
  FIdResponse.ResponseNo := Value;
end;

procedure TIdWebResponse.SendRedirect(const URI: string);
begin
  FIdResponse.Redirect(URI);
  SendResponse;
end;

procedure TIdWebResponse.SendResponse;
begin
  FIdResponse.WriteHeader;
  FIdResponse.WriteContent;
{
  fIdResponse.FreeContentStream:= True;
  ContentStream.Position:= 0;
  fIdResponse.ContentStream:= TMemoryStream.Create;
  fIdResponse.ContentStream.CopyFrom(ContentStream, ContentStream.Size);}
  FSent := True;
end;

procedure TIdWebResponse.SendStream(AStream: TStream);
begin
  FIdResponse.ContentStream := AStream;
  try
    FIdResponse.WriteContent;
  finally
    FIdResponse.ContentStream := nil;
  end;
end;

function TIdWebResponse.Sent: Boolean;
begin
  Result := FSent;
end;

procedure TIdWebResponse.SetCookieField(Values: TStrings; const ADomain,
  APath: string; AExpires: TDateTime; ASecure: Boolean);
begin

end;

const
  vjDate = 0;
  vjExpires = 1;
  vjLastModified = 2;

  vjContentLength = 0;

  vjVersion = 0;
  vjReasonString = 1;
```

```
  vjServer = 2;
  vjWWWAuthenticate = 3;
  vjRealm = 4;
  vjAllow = 5;
  vjLocation = 6;
  vjContentEncoding = 7;
  vjContentType = 8;
  vjContentVersion = 9;
  vjDerivedFrom = 10;
  vjTitle = 11;

function TIdWebResponse.GetDateVariable(Index: Integer): TDateTime;
begin
  case Index of
    vjDate: Result:= fIdResponse.Date;
    vjExpires: Result:= fIdResponse.Expires;
    vjLastModified: Result:= fIdResponse.LastModified;
  else
    Result:= 0;
  end;
end;

procedure TIdWebResponse.SetDateVariable(Index: Integer;
  const Value: TDateTime);
begin
  inherited;
  case Index of
    vjDate: fIdResponse.Date:= Value;
    vjExpires: fIdResponse.Expires:= Value;
    vjLastModified: fIdResponse.LastModified:= Value;
  end;
end;

function TIdWebResponse.GetIntegerVariable(Index: Integer): Integer;
begin
  case Index of
    vjContentLength: Result:= fIdResponse.ContentLength;
  else
    Result := 0;
  end;
end;

procedure TIdWebResponse.SetIntegerVariable(Index, Value: Integer);
begin
  inherited;
  case Index of
    vjContentLength: fIdResponse.ContentLength:= Value;
  else
  end;
end;

const
  HTTPResponseNames: array[0..11] of string = (
    '',
    '',
    '',
    '',
    'WWW-Authenticate',
    '',
    'Allow',
    '',
    '',
```

```
    '',
    '',
    'Derived-From',
    'Title'
  );

function TIdWebResponse.GetStringVariable(Index: Integer): string;
begin
  Result := '';
  case Index of
    vjVersion: Result:= '1.1';
    vjReasonString: Result:= fIdResponse.ResponseText;
    vjServer: Result:= fIdResponse.ServerSoftware;
    vjRealm: Result:= fIdResponse.AuthRealm;
    vjLocation: Result:= fIdResponse.Location;
    vjContentEncoding: Result:= fIdResponse.ContentEncoding;
    vjContentType: Result:= fIdResponse.ContentType;
    vjContentVersion: Result:= fIdResponse.ContentVersion;
  else
    if Index in [Low(HTTPResponseNames)..High(HTTPResponseNames)] then
      Result:= FIdResponse.RawHeaders.Values[HTTPResponseNames[Index]];
  end;
end;

procedure TIdWebResponse.SetStringVariable(Index: Integer;
  const Value: string);
begin
  case Index of
    vjVersion: ;
    vjReasonString: fIdResponse.ResponseText:= Value;
    vjServer: fIdResponse.ServerSoftware:= Value;
    vjRealm: fIdResponse.AuthRealm:= Value;
    vjLocation: fIdResponse.Location:= Value;
    vjContentEncoding: fIdResponse.ContentEncoding:= Value;
    vjContentType: fIdResponse.ContentType:= Value;
    vjContentVersion: fIdResponse.ContentVersion:= Value;
  else
    if Index in [Low(HTTPResponseNames)..High(HTTPResponseNames)] then
      FIdResponse.RawHeaders.Values[HTTPResponseNames[Index]]:= Value;
  end;
end;

procedure TIdWebResponse.SetContentStream(Value: TStream);
begin
  FIdResponse.ContentStream := Value;
  FIdResponse.ContentLength := Value.Size;
end;

{ TIdWebModule }

constructor TIdWebModule.Create(aOwner: TComponent);
begin
  inherited;
  fSOAPDispatcher:= THTTPSoapDispatcher.Create(Self);
  fSOAPInvoker:= THTTPSoapPascalInvoker.Create(Self);
  TSoapPascalInvokerProvidingInvoker.AdjustSOAPHeaders(fSOAPInvoker);
  fWSDLPublish:= TWSDLHTMLPublish.Create(Self);
  fSOAPDispatcher.Dispatcher:= fSOAPInvoker;
end;

function TIdWebModule.HandleRequest(Request: TWebRequest;
```

```
  Response: TWebResponse): Boolean;
begin
  if Request is TIdWebRequest then
    fWebApplication:= TIdWebRequest(Request).FIdHttpServer.Owner as TIdWebApplication
  else
    fWebApplication:= nil;
  InitModule;
  Result:= inherited HandleRequest(Request, Response);
end;

procedure TIdWebModule.InitModule;
begin

end;

end.
```

3.2. SOAPPasInv2.pas

```
unit SOAPPasInv2;

interface
uses
  Classes, InvokeRegistry, SOAPPasInv;

type
  TSoapPascalInvokerProvidingInvoker = class(TSoapPascalInvoker)
  public
    class procedure AdjustSOAPHeaders(aSelf: TSoapPascalInvoker);
    class function GetInvoker(aClass: TInvokableClass): TSoapPascalInvoker;
  end;

  THeaderListWithInvoker = class(THeaderList)
  protected
    fInvoker: TSoapPascalInvoker;
  end;

implementation

type
  TInvokableClassCrack = class(TInvokableClass)
  end;

  TSOAPHeadersBaseCrack = class(TSOAPHeadersBase)
  end;

{ TSoapPascalInvokerProvidingInvoker }

class function TSoapPascalInvokerProvidingInvoker.GetInvoker(
  aClass: TInvokableClass): TSoapPascalInvoker;
begin
  with TInvokableClassCrack(aClass), TSOAPHeadersBaseCrack(fSOAPHeaders) do
    if FHeadersInbound is THeaderListWithInvoker then
      Result:= THeaderListWithInvoker(FHeadersInbound).fInvoker
  else
    Result:= nil;

end;

class procedure TSoapPascalInvokerProvidingInvoker.AdjustSOAPHeaders(
  aSelf: TSoapPascalInvoker);
```

```
begin
  if not (TSoapPascalInvokerProvidingInvoker(aSelf).fHeadersIn is
THeaderListWithInvoker) then
  begin
    TSoapPascalInvokerProvidingInvoker(aSelf).FHeadersIn.Free;
    TSoapPascalInvokerProvidingInvoker(aSelf).FHeadersIn:=
THeaderListWithInvoker.Create;

THeaderListWithInvoker(TSoapPascalInvokerProvidingInvoker(aSelf).FHeadersIn).fInvoker
:= aSelf;
    TSoapPascalInvokerProvidingInvoker(aSelf).FHeadersIn.OwnsObjects := False;
  end;
end;

end.
```

3.3. IdSOAPHTTPClient.pas

```
unit IdSOAPHTTPClient;

interface

uses Classes, Rio, WSDLNode, WSDLItems, OPConvert, OPToSOAPDomConv, IdSOAPHTTPTrans,
WebNode, XMLIntf;

type
  TIdHTTPRIO = class(TRIO)
  private
    FWSDLItems: TWSDLItems;
    WSDLItemDoc: IXMLDocument;
    FWSDLView: TWSDLView;
    FWSDLLocation: string;
    FDOMConverter:  TOPToSoapDomConvert;
    FHTTPWebNode: TIdHTTPReqResp;
    FDefaultConverter: TOPToSoapDomConvert;
    FDefaultWebNode: TIdHTTPReqResp;
    procedure ClearDependentWSDLView;
    procedure SetWSDLLocation(Value: string);
    function  GetPort: string;
    procedure SetPortValue(Value: string);
    function  GetService: string;
    procedure SetService(Value: string);
    procedure CheckWSDLView;
    procedure SetURL(Value: string);

    function  GetDomConverter: TOpToSoapDomConvert;
    procedure SetDomConverter(Value: TOPToSoapDomConvert);

    function  GetHTTPWebNode: TIdHTTPReqResp;
    procedure SetHTTPWebNode(Value: TIdHTTPReqResp);

    function  GetURL: string;
    function  GetDefaultWebNode: TIdHTTPReqResp;
    function  GetDefaultConverter: TOPToSoapDomConvert;
  protected
    procedure Notification(AComponent: TComponent; Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function QueryInterface(const IID: TGUID; out Obj): HResult; override; stdcall;
    property WSDLItems: TWSDLItems read FWSDLItems;
```

```
  published
    property WSDLLocation: string read FWSDLLocation  write SetWSDLLocation;
    property Service: string read GetService write SetService;
    property Port: string read GetPort write SetPortValue;
    property URL: string read GetURL write SetURL;
    property HTTPWebNode: TIdHTTPReqResp read GetHTTPWebNode write SetHTTPWebNode;
    property Converter: TOPToSoapDomConvert read GetDomConverter write
SetDOMConverter;
  end;

implementation

uses SysUtils, InvokeRegistry;

constructor TIdHTTPRIO.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Converter }
  FDomConverter := GetDefaultConverter;
  FConverter := FDomConverter as IOPConvert;
  { WebNode }
  FHTTPWebNode := GetDefaultWebNode;
  FWebNode := FHTTPWebNode as IWebNode;
end;

destructor TIdHTTPRIO.Destroy;
begin
  if Assigned(FConverter) then
    FConverter := nil;
  if Assigned(FWebNode) then
    FWebNode := nil;
  if Assigned(FWSDLView) then
    FWSDLView.Free;

  { All components we own are automatically cleaned up }
  inherited;
end;

function TIdHTTPRIO.GetDefaultWebNode: TIdHTTPReqResp;
begin
  if (FDefaultWebNode = nil) then
  begin
    FDefaultWebNode := TIdHTTPReqResp.Create(Self);
    FDefaultWebNode.Name := 'HTTPWebNode1';               { do not localize }
    FDefaultWebNode.SetSubComponent(True);
  end;
  Result := FDefaultWebNode;
end;

function TIdHTTPRIO.GetDefaultConverter: TOPToSoapDomConvert;
begin
  if (FDefaultConverter = nil) then
  begin
    FDefaultConverter := TOPToSoapDomConvert.Create(Self);
    FDefaultConverter.Name := 'Converter1';                { do not localize }
    FDefaultConverter.SetSubComponent(True);
  end;
  Result := FDefaultConverter;
end;

procedure TIdHTTPRIO.ClearDependentWSDLView;
```

```
begin
  if Assigned(FDomConverter) and Assigned(FDOMConverter.WSDLView) then
    FDOMConverter.WSDLView := nil;
  if Assigned(FHTTPWebNode) and Assigned(FHTTPWebNode.WSDLView) then
    FHTTPWebNode.WSDLView := FWSDLView;
end;

procedure TIdHTTPRIO.CheckWSDLView;
begin
  if not Assigned(FWSDLItems) then
  begin
    if not Assigned(FWSDLItems) then
    begin
      FWSDLItems := TWSDLItems.Create(nil);
      WSDLItemDoc := FWSDLItems;
    end;
    if not Assigned(FWSDLView) then
    begin
      FWSDLView := TWSDLView.Create(nil);
      FWSDLView.SetDesignState(csDesigning in ComponentState);
    end;
    FWSDLView.WSDL := FWSDLItems;
    if Assigned(FDomConverter) then
      FDOMConverter.WSDLView := FWSDLView;
    if Assigned(FHTTPWebNode) then
      FHTTPWebNode.WSDLView := FWSDLView;
  end;
end;

function TIdHTTPRIO.GetPort: string;
begin
  if Assigned(FWSDLView) then
    Result := FWSDLView.Port
  else
    Result := '';
end;

function TIdHTTPRIO.GetService: string;
begin
  if Assigned(FWSDLView) then
    Result := FWSDLView.Service
  else
    Result := '';
end;

procedure TIdHTTPRIO.SetPortValue(Value: string);
begin
  if Assigned(FWSDLView) then
    FWSDLView.Port := Value;
end;

procedure TIdHTTPRIO.SetService(Value: string);
begin
  if Assigned(FWSDLView) then
    FWSDLView.Service := Value;
end;

procedure TIdHTTPRIO.SetURL(Value: string);
begin
  if Assigned(FHTTPWebNode) then
  begin
```

```
    FHTTPWebNode.URL := Value;
    if Value <> '' then
    begin
      WSDLLocation := '';
      ClearDependentWSDLView;
    end;
  end;
end;

procedure TIdHTTPRIO.SetWSDLLocation(Value: string);
begin
  { WSDLLocation and URL are currently mutually exclusive }
  { So clear out URL if we're setting a WSDLLocation }
  if (Value <> '') and (URL <> '') then
    FHTTPWebNode.URL := '';
  { Clear any currently cached WSDLs.
    NOTE: A RIO can only be bound to one given interface.
          Therefore switching WSDL will be a rather rare
          scenario. However, it's possible to have multiple
          Services that implement the same portype but
          expose different WSDLs. Case in point is the
          Interop Service that are exposed by various
          SOAP vendors. So to that end, we'll clear
          the WSDL Cache }
  if Assigned(FWSDLItems) and (WSDLItemDoc <> nil) then
  begin
    WSDLItemDoc := nil;
    FWSDLItems := nil;
  end;
  { This will recreate the WSDLView/Items }
  CheckWSDLView;
  { Deactivate }
  if FWSDLItems.Active then
    FWSDLItems.Active  := False;
  FWSDLLocation := Value;
  { Store the WSDLLocation as the FileName of the TWSDLItems }
  FWSDLItems.FileName := Value;
  FWSDLView.Port := '';
  FWSDLView.Service := '';
end;

function TIdHTTPRIO.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  Result := inherited QueryInterface(IID, Obj);
  { Here we check if we just bounded to an interface - and if yes, we retrieve
    & update items that are HTTP/transport specific }
  if Result = 0 then
  begin
    if IsEqualGUID(IID, FIID) then
    begin
      FHTTPWebNode.SoapAction := InvRegistry.GetActionURIOfIID(IID);
    end;
  end;
end;

function TIdHTTPRIO.GetDomConverter: TOPToSoapDomConvert;
begin
  if not Assigned(FDomConverter) then
  begin
    FDomConverter := GetDefaultConverter;
    FConverter := FDomConverter as IOPConvert;
```

```
  end;
  Result := FDomConverter;
end;

procedure TIdHTTPRIO.SetDomConverter(Value: TOPToSoapDomConvert);
begin
  if Assigned(FDOMConverter) and (FDomConverter.Owner = Self) then
  begin
    FConverter := nil;
    if FDomConverter <> FDefaultConverter then
      FDomConverter.Free;
  end;
  FDomConverter := Value;
  if Value <> nil then
  begin
    FConverter := Value;
    FDomConverter.FreeNotification(Self);
    FDomConverter.WSDLView := FWSDLView;
  end;
end;

function TIdHTTPRIO.GetHTTPWebNode: TIdHTTPReqResp;
begin
  if not Assigned(FHTTPWebNode) then
  begin
    FHTTPWebNode := GetDefaultWebNode;
    FWebNode := FHTTPWebNode as IWebNode;
  end;
  Result := FHTTPWebNode;
end;

procedure TIdHTTPRIO.SetHTTPWebNode(Value: TIdHTTPReqResp);
var
  URL, UDDIOperator, UDDIBindingKey: string;
begin
  if Assigned(FHTTPWebNode) then
  begin
    { Save previous endpoint configuration }
    URL := FHTTPWebNode.URL;

    { Cleanup if we're owner and it's not out default one }
    if (FHTTPWebNode.Owner = Self) and (FHTTPWebNode <> FDefaultWebNode) then
    begin
      FWebNode := nil;
      FHTTPWebNode.Free;
    end
  end
  else
  begin
    URL := '';
    UDDIOperator := '';
    UDDIBindingKey := '';
  end;

  FHTTPWebNode := Value;

  if Value <> nil then
  begin
    FWebNode := Value;
    { Make sure we get notified so we may cleanup properly }
    FHTTPWebNode.FreeNotification(Self);
```

```
    { WSDLView }
    FHTTPWebNode.WSDLView := FWSDLView;
  end
  else
  begin
    FHTTPWebNode := FDefaultWebNode;
    FWebNode := FHTTPWebNode as IWebNode;
  end;

  { Transfer previous endpoint configuration }
  if FHTTPWebNode <> nil then
  begin
    if (URL <> '') and (FHTTPWebNode.URL = '') then
      FHTTPWebNode.URL := URL;
  end;
end;

function TIdHTTPRIO.GetURL: string;
begin
  if Assigned(FHTTPWebNode) then
    Result := FHTTPWebNode.URL
  else
    Result := '';
end;

procedure TIdHTTPRIO.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited;
  if (Operation = opRemove) and (AComponent = FHTTPWebNode) then
  begin
    FWebNode := nil;
    FHTTPWebNode := nil;
  end;
  if (Operation = opRemove) and (AComponent = FDomConverter) then
  begin
    FConverter := nil;
    FDomConverter := nil;
  end;
end;

end.
```

3.4. IdSOAPHTTPTrans.pas

```
unit IdSOAPHTTPTrans;

interface

uses
  SysUtils, Classes, WebNode, WSDLNode, Types, IntfInfo, WSDLIntf, SOAPAttachIntf,
  IdHTTP, IdIOHandlerSocket, IdSSLOpenSSL;

type

  EIdSOAPHTTPException = class(Exception)
  private
    FStatusCode: Integer;
  public
    constructor Create(const Msg: string; SCode: Integer = 0);
```

```
    constructor CreateFmt(const Msg: string; const Args: array of const; SCode:
Integer = 0);

    property StatusCode: Integer read FStatusCode write FStatusCode;
  end;

  IdSOAPInvokeOptions = (soNoValueForEmptySOAPAction,   { Send "" or absolutely no
value for empty SOAPAction }
                         soIgnoreInvalidCerts             { xxx Handle Invalid Server
Cert and ask HTTP runtime to ignore }
                        );
  TIdSOAPInvokeOptions= set of IdSOAPInvokeOptions;

  TIdHTTPReqRespOnLog = procedure (Sender: TComponent; aOutbound, aHeader: Boolean;
St: TStream) of object;

  TIdHTTPReqResp = class;

  { Provides access to HTTPReqResp component }
  IIdHTTPReqResp = interface
  ['{5FA6A197-32DE-4225-BC85-216CB80D1561}']
    function GetHTTPReqResp: TIdHTTPReqResp;
  end;

  TIdHTTPReqResp = class(TComponent, IInterface, IWebNode, IIdHTTPReqResp)
  private
    FUserSetURL: Boolean;
    FRefCount: Integer;
    FOwnerIsComponent: Boolean;
    FURL: string;
    FBindingType: TWebServiceBindingType;
    FMimeBoundary: string;
    FWSDLView: TWSDLView;
    FSoapAction: string;
    FUseUTF8InHeader: Boolean;
    FInvokeOptions: TIdSOAPInvokeOptions;
    fIdHttp: TIdCustomHttp;
    fOnLog: TIdHTTPReqRespOnLog;
    procedure SetURL(const Value: string);
    function GetSOAPAction: string;
    procedure SetSOAPAction(const SOAPAction: string);
    procedure SetWSDLView(const WSDLVIew: TWSDLView);
    function GetSOAPActionHeader: string;
  protected
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    function GetMimeBoundary: string;
    procedure SetMimeBoundary(Value: string);
    procedure Log(aOutbound, aHeader: Boolean; St: TStream); virtual;
    procedure LogString(aOutbound, aHeader: Boolean; S: string);
  public
    constructor Create(Owner: TComponent); override;
    class function NewInstance: TObject; override;
    procedure AfterConstruction; override;
    destructor Destroy; override;
    function GetHTTPReqResp: TIdHTTPReqResp;
    procedure Get(Response: TStream); virtual;
    {IWebNode}
    procedure BeforeExecute(const IntfMD: TIntfMetaData;
                            const MethMD: TIntfMethEntry;
                            MethodIndex: Integer;
```

```
                              AttachHandler: IMimeAttachmentHandler);
    procedure Execute(const DataMsg: String; Response: TStream); overload; virtual;
    procedure Execute(const Request: TStream; Response: TStream); overload; virtual;
    function Execute(const Request: TStream): TStream; overload; virtual;
    property URL: string read FURL write SetURL;
    property SoapAction: string read GetSOAPAction write SetSOAPAction;
  published
    property HttpClient: TIdCustomHttp read fIdHttp;
    property WSDLView: TWSDLView read FWSDLView write SetWSDLView;
    property UseUTF8InHeader: Boolean read FUseUTF8InHeader write FUseUTF8InHeader;
    property InvokeOptions: TIdSOAPInvokeOptions read FInvokeOptions write
FInvokeOptions;
    property OnLog: TIdHTTPReqRespOnLog read fOnLog write fOnLog;
  end;

implementation
uses
{$IFDEF MSWINDOWS}
  Windows,
{$ENDIF}
  Variants, SOAPConst, XMLDoc, XMLIntf, InvokeRegistry, WSDLItems,
  SOAPAttach, UDDIHelper, IdIntercept, IdException, IdURI, IdGlobal, IdHeaderList,
IdHTTPHeaderInfo;

constructor EIdSOAPHTTPException.Create(const Msg: string; SCode: Integer = 0);
begin
  inherited Create(Msg);
  FStatusCode := SCode;
end;

constructor EIdSOAPHTTPException.CreateFmt(const Msg: string; const Args: array of
const; SCode: Integer = 0);
begin
  inherited CreateFmt(Msg, Args);
  FStatusCode := SCode;
end;

constructor TIdHTTPReqResp.Create(Owner: TComponent);
begin
  inherited;
  FIdHTTP:= TIdCustomHTTP.Create(Self);
  FIdHttp.Request.AcceptCharSet:= 'utf-8';
  FIdHttp.Request.UserAgent := 'Borland SOAP 1.2'; { Do not localize }
  FInvokeOptions := [soIgnoreInvalidCerts];
end;

destructor TIdHTTPReqResp.Destroy;
begin
  inherited;
end;

class function TIdHTTPReqResp.NewInstance: TObject;
begin
  Result := inherited NewInstance;
  TIdHTTPReqResp(Result).FRefCount := 1;
end;

procedure TIdHTTPReqResp.AfterConstruction;
begin
  inherited;
  FOwnerIsComponent := Assigned(Owner) and (Owner is TComponent);
```

```
  InterlockedDecrement(FRefCount);
end;

{ IInterface }

function TIdHTTPReqResp._AddRef: Integer;
begin
  Result := InterlockedIncrement(FRefCount)
end;

function TIdHTTPReqResp._Release: Integer;
begin
  Result := InterlockedDecrement(FRefCount);
  { If we are not being used as a TComponent, then use refcount to manage our
    lifetime as with TInterfacedObject. }
  if (Result = 0) and not FOwnerIsComponent then
    Destroy;
end;

function TIdHTTPReqResp.GetHTTPReqResp: TIdHTTPReqResp;
begin
  Result := Self;
end;

function TIdHTTPReqResp.GetSOAPAction: string;
begin
  if (FSoapAction = '') and not (soNoValueForEmptySOAPAction in FInvokeOptions) then
    Result := '""'
  else
    Result := FSoapAction;
end;

procedure TIdHTTPReqResp.SetSOAPAction(const SOAPAction: string);
begin
  FSoapAction := SOAPAction;
end;

procedure TIdHTTPReqResp.SetWSDLView(const WSDLVIew: TWSDLView);
begin
  FWSDLView := WSDLView;
end;

procedure TIdHTTPReqResp.SetURL(const Value: string);
begin
  FUserSetURL := Value <> '';
  FURL := Value;
end;

procedure TIdHTTPReqResp.SetMimeBoundary(Value: string);
begin
  FMimeBoundary := Value;
end;

function TIdHTTPReqResp.GetMimeBoundary: string;
begin
  Result := FMimeBoundary;
end;

function TIdHTTPReqResp.GetSOAPActionHeader: string;
begin
  if (SoapAction = '') then
```

```
    Result := SHTTPSoapAction + ':'
  else if (SoapAction = '""') then
    Result := SHTTPSoapAction + ': ""'
  else
    Result := SHTTPSoapAction + ': ' + '"' + SoapAction + '"';
end;

{ Here the RIO can perform any transports specific setup before call - XML
serialization is done }
procedure TIdHTTPReqResp.BeforeExecute(const IntfMD: TIntfMetaData;
                                       const MethMD: TIntfMethEntry;
                                       MethodIndex: Integer;
                                       AttachHandler: IMimeAttachmentHandler);
var
  MethName: InvString;
  Binding: InvString;
  QBinding: IQualifiedName;
begin
  if FUserSetURL then
  begin
    MethName := InvRegistry.GetMethExternalName(IntfMD.Info, MethMD.Name);
    FSoapAction := InvRegistry.GetActionURIOfInfo(IntfMD.Info, MethName,
MethodIndex);
  end
  else
  begin
    { User did *NOT* set a URL }
    if WSDLView <> nil then
      begin
      { Make sure WSDL is active }
        if fIdHttp.ProxyParams.ProxyServer <> '' then
        begin
          WSDLView.Proxy:= fIdHttp.ProxyParams.ProxyServer +
':'+IntToStr(fIdHttp.ProxyParams.ProxyPort);
          WSDLView.UserName:= fIdHttp.ProxyParams.ProxyUsername;
          WSDLView.Password:= fIdHttp.ProxyParams.ProxyPassword;
        end else
        begin
          { no proxy with Username/Password implies basic authentication }
          WSDLView.UserName:= fIdHttp.Request.Username;
          WSDLView.Password:= fIdHttp.Request.Password;
        end;

        WSDLView.Activate;
        QBinding := WSDLView.WSDL.GetBindingForServicePort(WSDLView.Service,
WSDLView.Port);
        if QBinding <> nil then
        begin
          Binding := QBinding.Name;
          MethName:= InvRegistry.GetMethExternalName(WSDLView.IntfInfo,
WSDLView.Operation);
          { TODO: Better to Pass in QBinding here to avoid tricky confusion due to
lack of namespace }
          FSoapAction := WSDLView.WSDL.GetSoapAction(Binding, MethName, 0);
        end;
        {NOTE: In case we can't get the SOAPAction - see if we have something in the
registry }
        {      It can't hurt:) }
        if FSoapAction = '' then
          InvRegistry.GetActionURIOfInfo(IntfMD.Info, MethName, MethodIndex);
        { Retrieve URL }
```

```
        FURL := WSDLView.WSDL.GetSoapAddressForServicePort(WSDLView.Service,
WSDLView.Port);
        if (FURL = '') then
          raise EIdSOAPHTTPException.CreateFmt(sCantGetURL, [WSDLView.Service,
WSDLView.Port, WSDLView.WSDL.FileName]);
      end
    else
      raise EIdSOAPHTTPException.Create(sNoWSDLURL);
  end;

  { Are we sending attachments?? }
  if AttachHandler <> nil then
  begin
    FBindingType := btMIME;
    { If yes, ask MIME handler what MIME boundary it's using to build the Multipart
      packet }
    FMimeBoundary := AttachHandler.MIMEBoundary;

    { Also customize the MIME packet for transport specific items }
    if UseUTF8InHeader then
      AttachHandler.AddSoapHeader(Format(ContentTypeTemplate, [ContentTypeUTF8]))
    else
      AttachHandler.AddSoapHeader(Format(ContentTypeTemplate, [ContentTypeNoUTF8]));
    AttachHandler.AddSoapHeader(GetSOAPActionHeader);
  end else
    FBindingType := btSOAP;
end;

function DateTimemsToStr(aDT: TDateTime): string;
begin
  Result:= FormatDateTime(ShortDateFormat+' hh:nn:ss.zzz', aDT);
end;

const
  ProtocolVersionString: array[TIdHTTPProtocolVersion] of string = ('1.0', '1.1');

procedure TIdHTTPReqResp.Execute(const Request: TStream; Response: TStream);
var
  URI: TIdURI;
  ContentType: string;
begin
  fIdHttp.Request.CustomHeaders.Clear;
  if FBindingType = btMIME then
  begin
    fIdHttp.Request.ContentType:= Format(ContentHeaderMIME, [FMimeBoundary]);
    fIdHttp.Request.CustomHeaders.Add(MimeVersion);
  end else { Assume btSOAP }
  begin
    fIdHttp.Request.ContentType := sTextXML;
    fIdHttp.Request.CustomHeaders.Add(GetSOAPActionHeader);
  end;

  URI := TIdURI.Create(fURL);
  try
    if URI.Port <> '' then
      {$IFDEF INDY10}
      fIdHttp.URL.Port := URI.Port;
      {$ELSE}
      fIdHttp.URL.Port := URI.Port;
      {$ENDIF}
    if URI.Host <> '' then
```

```
      fIdHttp.{$IFDEF INDY10}URL.{$ENDIF}Host := URI.Host
    else
      fIdHttp.{$IFDEF INDY10}URL.{$ENDIF}Host := Copy(fURL, Length('http://')+1,
                Pos(':' + URI.Port, fURL) - (Length('http://')+1));
  finally
    URI.Free;
  end;

  {$IFNDEF INDY10}
  fIdHttp.InputBuffer.Clear;
  {$ENDIF}

  try   // log here to log correct stamp
    LogString(False, True, Format('POST %s HTTP/%s  [%s @ %s]', [fURL,
ProtocolVersionString[fIdHttp.ProtocolVersion], Name, DateTimeMsToStr(Now)])+#13#10);
    //fIdHttp.Request.CustomHeaders.Values['X-Debug']:= Format('%s %s', [UrlS,
DateTimeToXMLDateTime(Now)]);
    fIdHttp.Request.SetHeaders;  // postponed to log correct RawHeaders
    LogString(False, True, fIdHttp.Request.RawHeaders.Text);
    Log(False, False, Request);
    Request.Position:= 0;
  except
  end;
  try
    try
      fIdHttp.Post(fURL, Request, Response);
    except
      on E: Exception do
      begin
        try
          LogString(True, True, E.Message+#13#10);
        except
        end;
        raise;
      end;
    end;
  finally
    if Response.Size > 0 then
    begin
      try
        LogString(True, True, fIdHttp.ResponseText+Format('  [%s @ %s]', [Name,
DateTimeMsToStr(Now)])+#13#10);
        LogString(True, True, fIdHttp.Response.RawHeaders.Text);
        Log(True, False, Response);
        Response.Position:= 0;
      except
      end;
    end;
  end;
  ContentType := fIdHttp.Response.RawHeaders.Values[SContentType];
  FMimeBoundary := GetMimeBoundaryFromType(ContentType);
  if Response.Size = 0 then
    raise EIdSOAPHTTPException.Create(SInvalidHTTPResponse);

  if SameText(ContentType, ContentTypeTextPlain) or
     SameText(ContentType, STextHtml) then
    raise EIdSOAPHTTPException.CreateFmt(SInvalidContentType, [ContentType]);
end;

procedure TIdHTTPReqResp.Execute(const DataMsg: String; Response: TStream);
var
```

```
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    Stream.SetSize(Length(DataMsg));
    Stream.Write(DataMsg[1], Length(DataMsg));
    Execute(Stream, Response);
  finally
    Stream.Free;
  end;
end;

function TIdHTTPReqResp.Execute(const Request: TStream): TStream;
begin
  Result := TMemoryStream.Create;
  Execute(Request, Result);
end;

procedure TIdHTTPReqResp.Get(Response: TStream);
begin
  if URL = '' then
    raise EIdSOAPHTTPException.Create(SEmptyURL);

  fIdHttp.Request.Accept := '*/*';
  fIdHttp.Request.ContentType := sTextXml;
  fIdHttp.Request.CustomHeaders.Clear;
    try
      LogString(False, True, Format('GET %s HTTP/%s  [%s @ %s]', [fURL,
ProtocolVersionString[fIdHttp.ProtocolVersion], Name, DateTimeMsToStr(Now)])+#13#10);
      //fIdHttp.Request.CustomHeaders.Values['X-Debug']:= Format('%s %s', [UrlS,
DateTimeToXMLDateTime(Now)]);
      fIdHttp.Request.SetHeaders;  // postponed to log correct RawHeaders
      LogString(False, True, fIdHttp.Request.RawHeaders.Text);
    except
    end;
    try
      fIdHttp.Get(URL, Response);
    finally
      if Response.Size > 0 then
      begin
        try
          LogString(True, True, fIdHttp.ResponseText+Format('  [%s @ %s]', [Name,
DateTimeMsToStr(Now)])+#13#10);
          LogString(True, True, fIdHttp.Response.RawHeaders.Text);
          Log(True, False, Response);
          Response.Position:= 0;
        except
        end;
      end;
    end;
end;

procedure TIdHTTPReqResp.Log(aOutbound, aHeader: Boolean; St: TStream);
begin
  if Assigned(fOnLog) then
  begin
    St.Position:= 0;
    fOnLog(Self, aOutbound, aHeader, St);
  end;
end;
```

```
procedure TIdHTTPReqResp.LogString(aOutbound, aHeader: Boolean; S: string);
var
  St: TStringStream;
begin
  St:= TStringStream.Create(S);
  try
    Log(aOutbound, aHeader, St);
  finally
    St.Free;
  end;
end;

end.
```