kleiner kommunikation
max@kleiner.com

1 Espresso bitte, aber stark

1.1 Einen WebService als Remote Data Storage mit dbExpress realisiert

Angenommen, ein weltweites Netz von Leitrechnern sendet periodisch Diagnosedaten an einen zentralen InterBase-Server, der die ausgewerteten Daten dann per Browser für die Techniker bereithält. Das war mal die Vision, welche mit komplizierten und inkompatiblen RPC's (Remote Procedure Call) vor Jahren bei uns realisiert wurde. Mittlerweile gibt es SOAP, demzufolge die künftigen Applikationsarchitekturen wie J2EE oder CLX wieder miteinander sprechen können. In der folgenden Fallstudie möchte ich die Grundzüge dieser Technik näherbringen und vor allem das entfernte Speichern von Daten mit dbExpress erklären. Im Gegensatz zu den RPC's oder einer Web-Skriptsprache, lassen sich WebServices voll und ganz objektorientiert in den Code integrieren und dann sauber kompilieren. Erweitert wird im Grunde der Funktionsumfang eines Web-Servers, der meistens eine Ergebnismenge zurückliefert. Das Beispiel zeigt ein Funktionsmuster, das via Interfaces gesammelte Daten in eine Datenbank oder ein File speichert.

1.2 Der Pilot in OP

Das Fallbeispiel lief auch schon unter Delphi 6.0, zu empfehlen ist aber Delphi 6.02, das genauer mit WSDL und den entsprechenden Experten umgehen kann. Auch im Bereich dbExpress hat das SevicePack 2 einiges gebracht, zumal die Geschichte auch mit Kylix 2 läuft. Zum Thema Einrichten eines WebService mit ObjectPascal (OP) verweise ich auf die bisher erschienenen Artikel, wir werden direkt in die Implementation des Piloten mit den folgenden Themen einsteigen.

- Technische Infrastruktur aufbauen
- Einrichten einer dynamischen dbExpress Verbindung
- Generieren von HTML als Folge einer Abfrage
- Aufruf von OP, C# und Java aus

Das Delphi-Projekt, das für eine CGI-Web-Extension konfiguriert wird, besteht aus den vier Modulen, Projektdatei, Webmodul, der Interface-Unit und der Implementierungs-Unit, welche den eigentlichen Zugriff auf den InterBase Server bewerkstelligt. Unsere Web-Extension wird dann nach dem Speichern der Daten zusätzlich als Antwort auf eine HTTP-Anforderung ein Liste der eingetragenen „Techniker oder Leitrechner" im Browser auflisten.

Als Laborumgebung benötigen Sie nebst einem „Webstuhl" folgendes Inventar:

- Ein Webserver mit eingerichteten Benutzerrechten, in unserem Fall Apache 1.3 im Shell-Modus.
- dbExpress Native Treiber auf dem Server (wird mit Delphi installiert)
- InterBase mit der Standard Vendor-Library *GDS32.DLL* und die beigefügte Datei *umlbank.gdb*

Bei den Vorarbeiten in der Delphi IDE unter OPTIONS empfehle ich als output-directory das Script-Verzeichnis des Web-Servers einzutragen, z.B. *d:\apache\webshare\scripts*, da dort die EXE jeweils vom Server bei jedem Aufruf geladen wird und sich auch wieder schliesst. Aktivieren Sie auch die Option einer Map-Datei, die stabileres Debuggen erlaubt und deaktivieren Sie vorläufig den Code-Optimizer.

Und hier kommt das nächste Problem beim Testen, denn eine CGI-Extension lässt sich, ausser man simuliert den Webserver mit einer komplizierten Konfigurationsdatei, nicht so einfach debuggen. Hier fährt man am besten, das Projekt als DLL zu kompilieren und dann mit den entsprechenden Startparametern als Host zu starten. Dies erfordert in der Regel ein kleine Änderung in der Projektdatei.

Als erstes werfen wir einen Blick auf Implementierungsklasse, welche die Schnittstelle `IVCLScanner` unterstützt. Die Registerkarte WebServices im Object Repository baut für Sie ja das Grundgerüst zusammen. Uns interessieren die Methoden `PostData` und `PostUser`, welche die Kernfunktionalität anbieten. Die Funktion `PostData` speichert die gescannten Daten in ein File auf dem Server und die Prozedur `PostUser` verewigt die zugehörigen Techniker in der Datenbank:

```
uses InvokeRegistry, Types, XSBuiltIns, VCLScannerIntf;

type
  TVCLScanner = class (TInvokableClass, IVCLScanner)
  public
    function PostData(const UserData: WideString; const CheckSum: DWORD):
                        Boolean; stdcall;
    procedure PostUser(const Email, FirstName, LastName: WideString);
                                stdcall;
  end;
```

Auf dem Webmodul haben sich mittlerweile die drei Standardkomponenten Dispatcher, Invoker und Publisher eines jeden WebServices plaziert, wir benötigen noch einen TableProducer, welcher die gespeicherten Daten auch auf dem entfernten Browser sichtbar macht. Welche Aufgaben aber erfüllen diese Komponenten:

- Die Komponente `THTTPSoapDispatcher` empfängt und beantwortet hereinkommende SOAP-Nachrichten und entscheidet, welches Interface man verwendet.
- Das korrekte Objekt, welches das Interface realisiert, wird an den PascalInvoker weitergeleitet, der die richtig ausgepackte Methode mit den formatierten Argumenten einschliesslich Errorcodes in den eigentlichen OP-Aufruf inklusive Parametern transformiert.
- `TWSDLHTMLPublish` ist verantwortlich für das Veröffentlichen von WSDL auf dem Browser, zusätzlich wird die WSDLADMIN-Datei im Webverzeichnis erzeugt, also aufgepasst, die EXE muss Schreibrechte besitzen.

Die Struktur des folgenden Klassendiagramms zeigt nun das ganze Framework in seiner Pracht. Deutlich zu sehen, weil fast im Mittelpunkt, das Interface IVCLScanner, das wiederum von IInvokable abstammt. Ein kurzer Blick in die Quellen zeigt erstaunliches. Mit der Compiler-Direktive wird allen von IInvokable abgeleiteten Interfaces die Run Time Type Information (RTTI) ermöglicht, somit lässt sich zur Laufzeit der Typ von allen Methoden und Eigenschaften bestimmen, welcher nach dem XML-Transport mit SOAP wieder hergestellt werden muss. Normalerweise haben nur Published-Sektionen diese RTTI.

```
{$M+}
  IInvokable = interface(IInterface)
  end;
{$M-}
```

TInvokableClass selbst besitzt nur einen Konstruktor, der mit Hilfe einer Klassenreferenz auch zur Laufzeit den Typ bestimmen muss. Virtuelle Konstruktoren machen auch nur dann Sinn, wenn sie über eine Klassenreferenz aufgerufen werden. In den anderen Fällen steht ja schon zur Designzeit fest, welcher Konstruktor zum Zuge kommt, weshalb dann eine statische Bindung genügt!

```
TInvokableClass = class(TInterfacedObject)
  public
    constructor Create; virtual;
  end;
 TInvokableClassClass = class of TInvokableClass;
```

Weiter zeigt das Klassendiagramm, sowohl der Client als `TfrmMain` wie das Webmodule (vor allem der Dispatcher) kommunizieren strikte nur über das Interface (Borland sei Dank). Da unser Framework noch eine kleine Konfigurationsdatei einliest, gibt es auch eine Assoziation zwischen dem VCLScanner

und dem WebModule. In dieser Konfigurationsdatei *pathinfo.txt* sind Angaben betreffend Datenbank- und Dateipfad zu finden, ansonsten der Datenbankort hart codiert wäre.

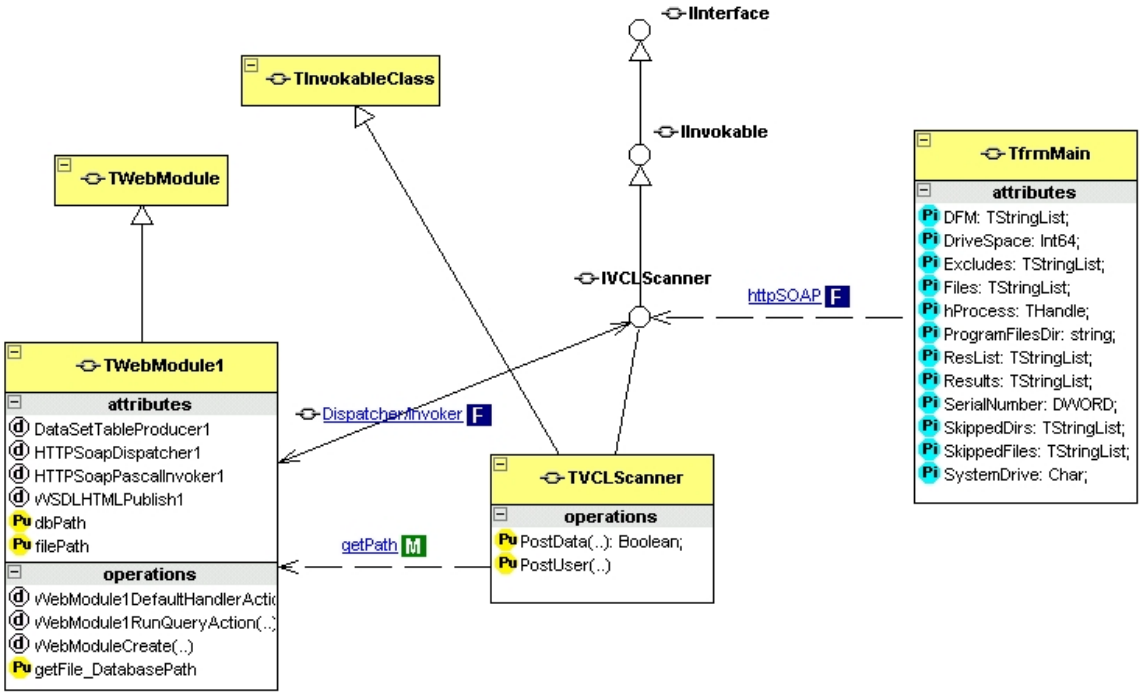

Abb 1: Hongkong sendet via PostData() an Frankfurt

// bild cd_wsstorage_web.jpg

Im weiteren benötigen wir einen Action Event-Handler, der aus der Anfrage eine HTML-Seite produziert. Das SQL-Statement für die Abfrage folgt weiter unten. Mittels eines Doppelklicks auf das OnAction-Event des Webmoduls entsteht der gewünschte Methodenrumpf. Mit dem Index-Property `Action` können Sie alle Aktionen der Kollektion `Actions` durchlaufen. 0 ist der Index der ersten Aktion, 1 der Index der zweiten Aktion usw. In unserem Fall gibt es zwei Aktionen, den DefaultHandler für die WSDL-Publikation und die hausgemachte Aktion `runselect`, für den TableProducer. Mit der Methode `ActionByName` können Sie auch auf einzelne Aktionen über ihren Namen zugreifen.

1.3 DbExpress ganz dynamisch

dbExpress ist eine Cross-Platform, welche ein gemeinsames Interface für mehrere SQL-Server definiert. Für jeden unterstützten Server bietet dbExpress einen nativen Treiber bezüglich Queries und Stored Procedures für Linux und Windows an.

Nebst der ausserordentlichen Performance ist es vor allem die Konfiguration und Verteilung, die enorme Effizienz aufweist, (dbExpress was designed to be fast, lightweight, and easy to deploy). Wie sie gleich sehen, genügen zwei Dateien (im Falle von zusätzlichem ClientDataSet drei Dateien), die auf dem Server residieren müssen. Das Ziel ist es, den Verbindungsaufbau wie die Konfiguration ohne nichtvisuelle Komponenten oder Aliase à la BDE zu ermöglichen. Alle Parameter sollen transparent und direkt im Code erscheinen.

Als wichtigen Parameter, lässt sich die Datenbank auch explizit im Code hinzufügen:

```
Params.Add('Database=milo2:D:\franktech\webservices\umlbank.gdb');
```

Wir aber ersetzen diesen Teil durch das Einlesen besagter Datei. Die folgende Routine PostUser erstellt nun eine TSQLConnection mit den benötigten Parametern, welche wiederum mit einem DataSet verknüpft wird. Nach wie vor besteht die Möglichkeit, mit der Methode

`LoadParamsOnConnect` die Parameter aus einer Datei wie z.B. unter Linux *dbxConnections* zu beziehen. Aber eben, wir wollen so schnell und autonom wie möglich sein. Anschliessend speichert das DataSet mit einer Insert-Operation die gewünschten Daten. Sicherheitsaspekte lassen sich hier der Einfachheit halber nicht berücksichtigen. Unter Kylix heisst der LibraryName *libsqlib.so* und die VendorLib *libgds.so*.:

```
procedure TVCLScanner.PostUser(const Email, FirstName, LastName :
WideString);
var
  Connection: TSQLConnection;
  DataSet: TSQLDataSet;
  logdate: String[15];
begin
  Connection:= TSQLConnection.Create(nil);
  with Connection do begin
    ConnectionName:= 'VCLScanner';
    DriverName:= 'INTERBASE';
    LibraryName:= 'dbexpint.dll';
    VendorLib:= 'GDS32.DLL';
    GetDriverFunc:= 'getSQLDriverINTERBASE';
    Params.Add('User_Name=SYSDBA');
    Params.Add('Password=masterkey');
    with TWebModule1.create(NIL) do begin
        getFile_DataBasePath;
        Params.Add(dbPath);
        free;
    end;
    LoginPrompt:= False;
    Open;
  end;
  logdate:=dateTimetostr(now);
  DataSet:= TSQLDataSet.Create(nil);
  with DataSet do begin
    SQLConnection:= Connection;
    CommandText:=
          Format('insert into KINGS values("%d","%s", "%s", "%s","%s")',
                     [10, Email,FirstName,LastName, logdate]);
    try
      ExecSQL;
    except
      raise
    end;
  end; //dataSet
   Connection.Close;
   DataSet.Free;
  Connection.Free;
end;
```

1.4 Feedback auf den Browser

Nachdem der WebService die Daten entfernt auf dem InterBase-Server gespeichert hat, ist nun die Darstellung auf dem Browser an der Reihe. Das Webmodul sucht die Instanz mit dem zugehörigen Aktionselement und löst die passende Ereignisbehandlungsroutine für `OnAction` aus, um eine Antwort auf die Anforderung zu generieren und zu senden. Die Hauptaufgabe von `Request` ist also, aus dem URL-String verschiedene Informationen, vor allem die `Pathinfo` und den Abfrage-Parameter, auszufiltern, in unserem Fall das runselect:

`http://milo2/scripts/vclscannerserver.exe/runselect`

Unser Event-Handler (auch Action-Handler genannt) läßt sich also mit der `Pathinfo runselect` verbinden. Im Action-Editor entspricht deshalb `runselect` dem `Pathinfo` (Pfadenteil).

Das zweite Objekt `Response` des Action-Handlers ist eine TWebResponse-Instanz, die von der Ereignisbehandlungsroutine ausgefüllt wird, d.h. der generierte HTML-Inhalt wird durch `Response` an den Web-Server übergeben. Der Parameter `Handled` zeigt schlussendlich an, ob das TWebActionItem-Objekt die Bearbeitung der Anforderung abgeschlossen hat.

Wenn man die VCL-Sourcen durchstöbert, kommt man zum schlichten Schluss: Es gibt keine statischen HTML-Vorlagen. Das Ergebnis wird durchwegs dynamisch mittels `response.content` dem Web-Server übergeben. Das folgende Sequenzdiagramm zeigt die beiden Szenarios als UseCases „Speichern der gesammelten Daten“ und „Abfragen der Technikerliste“ auf einen Blick:

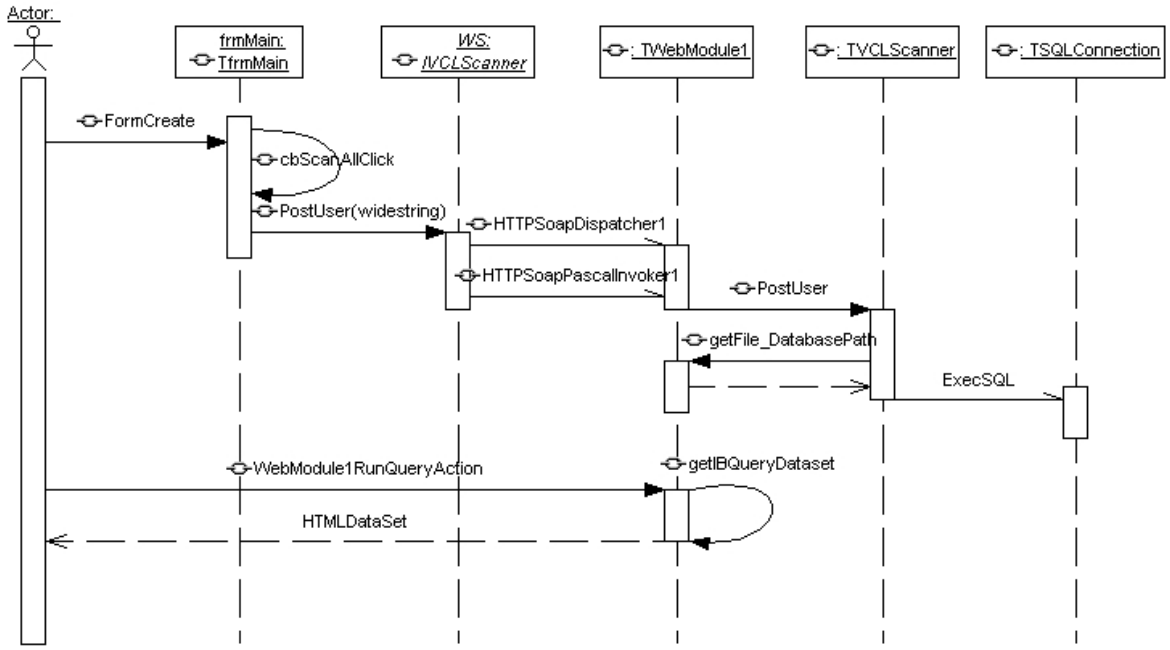

Abb. 2: Der Speicherfluss und die Abfrage im SEQ

// bild seq_rdstorage.jpg

Sobald die GET-Botschaft vom Browser kommt, generieren die TDataSetTableProducer-Objekte die Datenquelle intern. Die Eigenschaft `Query` des QueryTableProducer ist mit der Eigenschaft `DataSet` identisch, allerdings verwendet `Query` nicht die Basisklasse `TDataSet`, sondern eine eigene. Auch hier hat man wieder konsequent auf dbExpress gesetzt, so dass im WebModul keine weiteren Komponenten mehr ersichtlich sind (siehe Abb. 3).

Abb 3: Das Gerüst im Framework

// bild server_soap.tif

Kommen wir zur eigentlichen Abfrage von `TSQLDataSet`, die ich zur besseren Strukturierung in eine eigene Funktion auslagere, welche ein DataSet zurückliefert, dass sogar kompatibel zum TableProducer DataSet ist. Ein `TSQLDataSet` ist ressourcenschonend und schnell, hat aber seinen Preis. Da die Komponente keine Kopien der Daten gepuffert im Speicher verwaltet, ist die Menge unidirektional und nicht direkt beschreibbar, hat also in einem DBGrid nichts zu verlieren oder zu suchen ;).

```
procedure TWebModule1.WebModule1RunQueryAction(Sender: TObject; Request:
        TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  getFile_DatabasePath;
  with DataSetTableProducer1 do begin
    try
      dataSet:= getIBQueryDataset;
      response.Content:= content;
    except
      Response.Content:= Format('<html><body><b>database "%s" or query
                 not found!' + '</b></body></html>',[dbpath]);
    end;
  end;
    Connection.Close;
    myDataSet.Free;
    Connection.Free;
end;
```

1.5 Alle rufen OP

Wenn schlussendlich ein Client den Dienst nutzt, hat man durch die sprachübergreifende SOAP-Spezifikation freie Wahl. Bevor ich am Schluss noch exemplarisch den Aufruf in den drei Sprachen OP, Java und C# zeige, möchte ich kurz ein Geheimnis des hexadezimalen Hexenwerkes lüften. Wer verpackt eigentlich die ganze Datenfülle im XML-Envelope? Es ist die Komponente THTTPRIO, die bei genauem Debuggen (Abb. 4) offenbart, dass die CLX mit Hilfe von TOPToSoapDomConvert als IOPConvert das eigentliche Marshaling, d.h. Verpacken und Auspacken von Parametern mit den zugehörigen Funktionsaufrufen umsetzt. Als Parser setzt man DOM in Abhängigkeit der Library ein.

Klassen die IOPConvert implementieren benötigen zwingend die RTTI des Invokable Interface, womit die Compiler Direktive `{$M+}` wieder ins Spiel kommt. Das eigentliche Konvertieren und Interpretieren dieser „transportfähigen Strings“, welche ja codierte Methodenaufrufe mit Schnittstellen darstellen, genau das ist die Hexerei.

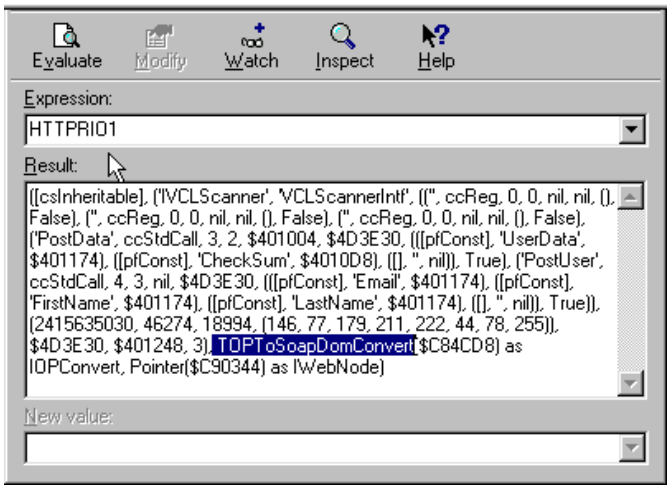

Abb. 4: Der Client-Aufruf vor dem Verpacken

// bild debug_dom.tif

Der Aufruf aus Delphi basiert auf der Schnittstellen Instanz, die nach erfolgreichem Abfragen des unterstützten Interfaces den wohlverdienten Zeiger erhält:

```
var
  WS: IVCLScanner;
  WS:= HTTPRIO1 as IVCLScanner;
  WS.PostData(reFinalResults.Text,CRC);

HTTPRIO1:URL
http://milo2/scripts/vclscannerserver.exe/soap/IVCLScanner
```

In Java und C# sind die Verhältnisse nicht unähnlich, bei Java wird der URL im Sinne des Schnittstellenzugriffs gleich mit dem Konstruktor übergeben, was bei OP in der Komponente als Eigenschaft erfolgt, womit wieder mal gesagt ist, die Mutter aller wohlerzogenen Sprachen Pascal ist ;).

```
  IVCLScannerStub stub = new IVCLScannerStub(new URL "http://milo2/
                         scripts/vclscanner.exe/soap/IVCLScanner"));
    stub.PostData(reFinalResults.Text,CRC);

  IVCLScanner.IVCLScannerservice objIVCLScanner = new
                              IVCLScanner.IVCLScannerservice();
  bool booScanner= objIVCLScanner.PostData(reFinalResults.Text,CRC);
```

Max Kleiner