Under Construction: More Local ClientDataSets

by Bob Swart

I was one of the about 200 people attending the Kylix Preview at the Inprise UK Conference in London on September 25th, where David Intersimone and John Kaster talked about and showed the Kylix IDE, accompanied by their CEO Dale Fuller, who had come over to give us one important message: Kylix will ship when it's ready (and not sooner), and the aim for 'before the end of the year' is a mere aim, not a goal in itself.

During that presentation, David received some cheers when he publicly announced that MIDAS (and specifically `TClientDataSet`) would be moved down from the Enterprise level to the Professional level, and Kylix (specifically, Delphi Professional for Linux) would contain MIDAS and a `TClientDataSet` component. The cheers quickly became boos (from me, that is) when David also made it clear that only the thin-client side of MIDAS would be in the Pro boxes, and not the `TxxxConnection` components. Too bad, but not entirely unexpected, since the `Connection` components are the ones that make MIDAS applications turn into n-tier applications. You could, of course, decide to roll your own (at least when it comes to sending data from one tier to another), which is what we will do near the end of this column, so don't go away!

Last Time

Last time, I explained some of the benefits of using local `ClientDataSets` (speed, easy installation, etc) without actually using them, because we spend most of our time feeding the `ClientDataSets` with data. In fact, one could say that the majority of last month's column was focused on XML and the XML format produced (and recognised) by the `ClientDataSet` component. We even ended up with a table definition and generation tool that could generate table metadata in XML format.

This time, we'll use that result to create thin-client applications that require no additional configuration (other than deploying MIDAS.DLL) and will be blazing fast. We'll then use a socket component to make two tiers (but separate applications) that communicate `ClientDataSet` data in XML format between each other.

Simple CGI

Assuming that we have an XML data file available (either with data or just with metadata, as we generated last time), then the easiest way to demonstrate the ease of use of `TClientDataSet` components is to create a web server console application (in other words a CGI application) that just dumps the content of a `ClientDataSet` to an HTML table. I'm deliberately using a console application here, and not a WebBroker application, since that would only distract us from the use of the `ClientDataSet` component.

Once you've dynamically created a `TClientDataSet` instance, all you need to do is perform a `LoadFromFile` (or `LoadFromStream`) and call `Open` (or set `Active` to `True`) and the data is available. No BDE `DatabaseName` and `TableName`, nor an `ADOConnectionString` and `TableName` (or `CommandText`), but just a `LoadFromFile` to fill the `ClientDataSet` with data.

In fact, if you look at Listing 1, you will notice that this is a CGI application already. And with a little effort, you can make it search for specific records, like all the `BIOLIFE` fish that are smaller than a certain size (using the `Length (cm)` field).

Since `ClientDataSet` cannot perform SQL queries, you need to perform this search either by using a filter, or by checking the value of the `Length_In` field inside

➤ *Listing 1*

```
program CDSxmlC;
{$APPTYPE CONSOLE}
uses
  DBClient;
var
  i: Integer;
begin
  writeln('content-type: text/html');
  writeln;
  writeln('<HTML>');
  writeln('<BODY BGCOLOR=FFFFCC>');
  with TClientDataSet.Create(nil) do
  try
    LoadFromFile('table.xml');
    Open;
    First;
    writeln('<TABLE BORDER=1>');
    writeln('<TR>');
    for i:=0 to Pred(FieldCount) do
      writeln('<TD BGCOLOR=FFFFFF><B>',Fields[i].FieldName,'</B></TD>');
    writeln('</TR>');
    while not Eof do begin
      writeln('<TR>');
      for i:=0 to Pred(FieldCount) do
        writeln('<TD VALIGN=TOP>',Fields[i].AsString,'</TD>');
      writeln('</TR>');
      Next
    end;
    writeln('</TABLE>')
  finally
    Free
  end;
  writeln('</BODY>');
  writeln('</HTML>')
end.
```

the `while not Eof` loop. The latter solution would result in the code shown in Listing 2 (with 42 hard coded, but you get the idea).

The other solution, using a `Filter`, would result in the following additional code (right before you open the `ClientDataSet` to get the data):

```
Filter := 'Length_In > 42';
Filtered := True;
```

You can also combine these two techniques: using a filter for a rough filter and a narrow selection inside the `while not Eof` loop to pick only the records you really want to display (also useful for conditions that are hard to express as filter expressions).

Multi-User Read-Only

There's one important thing you should have noticed from the example in Listing 1 and that's the fact that I'm only using the `ClientDataSet` to load data, search it and present the selected records. I do not make any modifications (insert, append or edit) to the data, nor do I save the data back to disk. *Why would that be?*

Well, the main reason is that whilst a `ClientDataSet` *can* save itself to an external file, it can only save *everything* at once. All the records can be written to an external file with one `SaveToFile`, or using the `Filename` property, as we saw last time. This is no big deal for a small `ClientDataSet`, but with bigger `ClientDataSets` (say, a few Mb in size), we find a slight performance hit when loading the `ClientDataSet`, but performance falls especially when saving the `ClientDataSet` to file or stream again. And the worst thing is that we could potentially overwrite someone else's changes, because the `ClientDataSet` using `SaveToFile` is saving itself on a table-level and not on a record-level, compared to using a normal dataset.

Even when using the `ClientDataSet` inside an ISAPI DLL (instead of a CGI executable), we'd still need to perform a `SaveToFile` if the content changes, potentially overwriting changes that have happened in another ISAPI thread (unless you want to make the `ClientDataSet` available in a single thread only, but that would limit the usability of your web server application).

In short: using a `ClientDataSet` component as a standalone datasource (with `LoadFromFile` and `SaveToFile`) is only a sensible solution if you want to present read-only data, not when adding or updating data.

Standalone GUI

When using a standalone Windows GUI, we're playing a different ballgame, of course. This time, there's only one user, and rarely a chance of overwriting a `ClientDataSet` from another user (unless you decide to share a `ClientDataSet` .CDS or .XML file on a network, but in that case the data packets are actually moving from one machine to another and a MIDAS licence might be necessary, which is what I'd like to avoid when using standalone `ClientDataSets` this way).

In short: when writing standalone Windows GUI applications, there is little chance of running into the aforementioned update problems with `ClientDataSets` using `LoadFromFile` and `SaveToFile`. And you do have the benefit of a zero-configuration DBMS all in one MIDAS.DLL. When I'm talking about a Windows GUI application, I include ActiveForms as well. And this opens up a whole new way of deploying thin (only the ActiveForm with MIDAS.DLL) rich (all visual Delphi components) internet clients that's too good to ignore.

Communicating XML

Apart from having a read-only web server application or a fully read-write Windows GUI application, I now want to show you how

```
while not Eof do begin
  if FieldByName('Length_In').AsInteger > 42 then begin
    writeln('<TR>');
    for i:=0 to Pred(FieldCount) do
      writeln('<TD VALIGN=TOP>',Fields[i].AsString,'</TD>');
    writeln('</TR>')
  end;
  Next
end;
```

➤ *Listing 2*

we can write a Delphi application to produce XML compatible with `ClientDataSet` (using the code from last month) and use socket components to send it over a wire to another Delphi application that can use this XML to feed the `ClientDataSet` and show the data inside a `DBGrid`. Think of this as 2-tier the 'cheap' way, but also as 2-tier the 'professional' way, meaning that it's the 2-tier way we will have when Delphi Professional for Linux ships, including a `ClientDataSet` but without `Connection` components. The same will be true for the upcoming Delphi 6 Professional: MIDAS without the multi-tier communication connection components.

As an example, I now want to write the skeleton of a 2-tier ordering system, where multiple client applications (for example, used by customers) are using regular tables to produce XML data that they send to a single server application (for example, an ordering database). Since the DataSetXML code from last month can in fact operate on any dataset component (including a `TClientDataSet`), we can in practice decide to turn either the clients or the server or even both into `ClientDataSet` applications. The only thing we need to do by ourselves now is produce the data in XML format and send it over the wire using two socket components.

XML Server

In order to send a certain string (containing XML in our case) over a socket connection, we must use the `TServerSocket` and `TClientSockets` of Delphi 5 Professional. We start with the XML Server application, the one that receives the initial XML string. The Client application has to send the XML string

```
unit SUnit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ScktComp;
type
  TForm1 = class(TForm)
    ServerSocket1: TServerSocket;
    procedure FormActivate(Sender: TObject);
    procedure ServerSocket1ClientConnect(Sender: TObject;
      Socket: TCustomWinSocket);
  private
    XMLString: String;
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormActivate(Sender: TObject);
var
  F: File;
begin
  System.Assign(F,'table.xml');
  Reset(F,1);
  SetLength(XMLString,FileSize(F));
  BlockRead(F,XMLString[1],FileSize(F));
  System.Close(F)
end;
procedure TForm1.ServerSocket1ClientConnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Caption := Caption + '!';
  Socket.SendText(XMLString)
end;
end.
```

➤ *Listing 3*

to the server using a specific port. A port can be compared to a radio frequency: only if the radio is set to 'listen' to a specific frequency, will the radio play the music sent by a specific radio station. With sockets, we can determine the port number ourselves. Well, almost, that is, since port numbers smaller than 1024 are reserved for internal use and services such as FTP and HTTP connections (just about everything on the net boils down to a socket connection in the end).

Start a new application with Delphi 5 and drop a `TServerSocket` component from the `Internet` tab on the main form. Make sure to give the `Port` property a unique value for your own communication purpose (something like 4242, although that might not be unique when I'm around). Once we set the property `Active` to `True`, the server will be available (when the application runs) to receive client connections.

XML Client

Start a new application for the XML Client, but this time place a `TClientSocket` component on the main form. The `Port` number must be the same as the one you used in the `TServerSocket` component (otherwise they won't find each other), so I've used 4242 again here. Apart from the `Port` number, the client must specify which server it wants to connect to (and send its XML data to). There are two properties that we can use for this purpose, namely `Address` and `Host`. The property `Address` can only contain an IP address (like 192.168.92.201 for my laptop), while the `Host` property can contain either an IP address or a valid DNS name (like Voyager, which is the IP-address for the same laptop, as defined in the `HOSTS` file on my NT machine). Only when both the XML Server and Client are running on the same machine, it is possible to specify `LocalHost` as `Host`. In our case, I've specified the `Address`, since that feels like the fastest (and most maintenance needing) way to specify which server machine to connect to.

Once we set the `Active` property of the `ClientSocket` to `True`, it will directly try to find the `ServerSocket` component on the target machine (specified by the `Address` or `Host` property) by sending a connection string to the specified `Port` of that IP address. It should be obvious, I hope, that the server application must be started before you try to start the client application!

XML Connection

When the client application makes a connection to the server, then the `OnClientConnection` event is fired (at the server side). This is the moment where the server can send an XML string with the table definitions to the client. Listing 3 contains the server unit. Note that I'm sharing the `XMLString` here,

```
procedure TForm2.ClientSocket1Read(Sender: TObject; Socket: TCustomWinSocket);
var
  F: System.Text;
begin
  Caption := Caption + '.';
  SocketText := SocketText + Socket.ReceiveText;
  if Pos('</DATAPACKET>',SocketText) > 0 then begin
    System.Assign(F,'client.xml');
    Rewrite(F);
    writeln(F,SocketText);
    System.Close(F);
    Caption := Caption + '!';
    SocketText := '';
    ClientDataSet1.LoadFromFile('client.xml');
  end
end;
```

➤ *Listing 4*

because multiple clients may want to connect to the server to obtain the XML data. When the form is activated, I load an external file that holds the XML into the `XMLString` using a fast `BlockRead`.

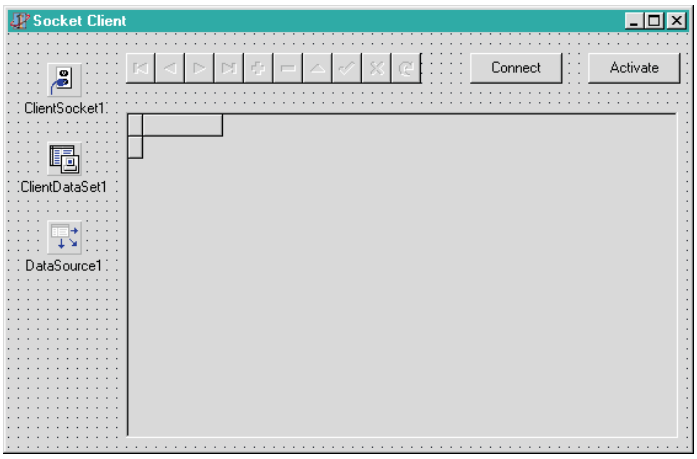
The content of `XMLString` could be generated by the Table-2-XML definition code we wrote last time (and in fact the file table.xml contains the Table-2-XML generated XML data). As soon as the `XMLString` is sent by the server and received by the client, the `OnRead` event of the `ClientSocket` on the client side is called. In this event handler, we can retrieve the `XMLString` data sent by the server, and use it to feed (initialise) our local `ClientDataSet`, see Listing 4.

Note that we cannot be sure that all data will be received in this single event. Hence the code to append all incoming data (from the `SocketServer`) to a single `SocketText` variable. As soon as the closing XML tag is received (in this case the `</DATAPACKET>` tag, not the `</XML>` tag), we know that the last bit of data has been transferred, and only then can we truly dump the `SocketText` content to a file (called client.xml) and load it inside the `ClientDataSet`.

You can now even decide to set the FileName property of the local ClientDataSet to client.xml, and thus benefit from the briefcase model. When the FileName property has a value, the content of ClientDataSet will automatically be saved to this particular file when you close the application.

One result of sending the XML as type XMLString over the socket connection (using port 4242) is that the client is truly a thin client. We only need the ClientSocket component, which receives the XML data that is fed to the ClientDataSet, and then of course a DataSource connecting the ClientDataSet to the DBNavigator and DBGrid. Apart from that, we only need the MIDAS.DLL on the client side.

Note, by the way, that the client application is totally unaware of the data definition (tables, fields) that it receives from the server. At the server side, we could decide to load and send the XML versions of BIOLIFE or CUSTOMER or whatever other database table we have converted to XML, including an empty definition only XML file generated with the final tool from last time. As long as we send ClientDataSet-compatible XML over the socket connection, the client will be able to display everything correctly. Ideal for thin web clients, if you ask me.



➤ *Figure 1*

XML String

The Server code in Listing 3 currently loads an XML file table.xml. This means that the XMLString was loaded with the

➤ *Listing 5*

```
unit XMLStr;
interface
uses
  DB;
function DataSetXMLString(DataSet: TDataSet): String;
implementation
uses
  SysUtils, TypInfo;
function DataSetXMLString(DataSet: TDataSet): String;
var
  Str: String;
  i: Integer;
  function Print(Str: String): String;
  { Convert a fieldname to a printable name }
  var
    i: Integer;
  begin
    for i:=Length(Str) downto 1 do
      if not (UpCase(Str[i]) in ['A'..'Z','1'..'9']) then
        Str[i] := '_';
    Result := Str
  end {Print};
  function EnCode(Str: String): String;
  { Convert memo contents to single line XML }
  var
    i: Integer;
  begin
    for i:=Length(Str) downto 1 do begin
      if (Ord(Str[i]) in [1..31]) or
        (Str[i] = '"') then begin
        Insert('&#'+IntToStr(Ord(Str[i]))+';',Str,i+1);
        Delete(Str,i,1)
      end else if Str[i] = #0 then
        Delete(Str,i,1)
    end;
    Result := Str
  end {EnCode};
begin
  ShortDateFormat := 'YYYYMMDD';
  try
    Str := '<?xml version="1.0" standalone="yes"?>';
    Str := Str + '<DATAPACKET Version="2.0">';
    with DataSet do begin
      Str := Str + '<METADATA>';
      Str := Str + '<FIELDS>';
      if not Active then
        { get info without opening the database }
        FieldDefs.Update;
      for i:=0 to Pred(FieldDefs.Count) do begin
        Str := Str + '<FIELD ';
        if Print(FieldDefs[i].Name) <> FieldDefs[i].Name
          then { fieldname }
          Str := Str + 'fieldname="' +
            FieldDefs[i].Name + '" ';
        Str := Str + 'attrname="' +
          Print(FieldDefs[i].Name) + '" fieldtype="';
        case FieldDefs[i].DataType of
          ftString,
        ftFixedChar,
      ftWideString: Str := Str + 'string';
         ftBoolean: Str := Str + 'boolean';
       ftSmallint: Str := Str + 'i2';
        ftInteger: Str := Str + 'i4';
        ftAutoInc: Str := Str + 'i4" readonly="true"
          SUBTYPE="Autoinc';
          ftWord, // why not i4 ??
        ftFloat: Str := Str + 'r8';
      ftCurrency: Str := Str + 'r8" SUBTYPE="Money';
          ftBCD: Str := Str + 'fixed';
         ftDate: Str := Str + 'date';
         ftTime: Str := Str + 'time';
     ftDateTime: Str := Str + 'datetime';
        ftBytes: Str := Str + 'bin.hex';
      ftVarBytes,
          ftBlob: Str := Str + 'bin.hex" SUBTYPE="Binary';
          ftMemo: Str := Str + 'bin.hex" SUBTYPE="Text';
       ftGraphic,
  ftTypedBinary: Str := Str + 'bin.hex"
                       SUBTYPE="Graphics';
      ftFmtMemo: Str := Str + 'bin.hex"
                       SUBTYPE="Formatted';
       ftParadoxOle,
       ftDBaseOle: Str := Str + 'bin.hex" SUBTYPE="Ole'
        end;
        if FieldDefs[i].Required then
          Str := Str + '"required="true';
        if FieldDefs[i].Size > 0 then
          Str := Str + '" WIDTH="' +
            IntToStr(FieldDefs[i].Size);
        Str := Str + '"/>'
      end;
      Str := Str + '</FIELDS>';
      Str := Str + '</METADATA>';
      if not Active then
        Open;
      Str := Str + '<ROWDATA>';
      while not Eof do begin
        Str := Str + '<ROW ';
        for i:=0 to Pred(Fields.Count) do
          if (Fields[i].AsString <> '') and
            ((Fields[i].DisplayText = Fields[i].AsString) or
             (Fields[i].DisplayText = '(MEMO)')) then
            Str := Str + Print(Fields[i].FieldName) + '="' +
              EnCode(Fields[i].AsString) + '" ';
        Str := Str + '/>';
        Next
      end;
      Str := Str + '</ROWDATA>'
    end;
    Str := Str + '</DATAPACKET>'
  finally
    Result := Str
  end
end;
end.
```

contents of an external XML file generated by the `DataSetXML` conversion routine from last month. To make things a little easier, I've modified the `DataSetXML` function into a `DataSetXMLString` function for the occasion (see Listing 5 for details), so it can return an instant XML `LongString` with the entire table. Quite handy, in fact, as it saves us the trouble of using external files to store and load the XML from.

Given a `DataSet` on the server side, the response to a client connection can be modified using the `DataSetXMLString`. And this dataset on the server can be used to apply the updates (if you decide to send those from the client back to the server again), but I'll leave that as an exercise for the reader.

What's important is that this technique will be very useful once Delphi Professional for Linux ships and as soon as this is available I will follow up with an article that shows how to connect a Delphi for Windows and a Delphi for Linux application to each other using the `Socket` component and sending XML as data back and forth. This should be very interesting, to say the least!

TableBob

Remember that last time I claimed that the `DataSetXML` converter and Table-2-XML definition utilities were in need of a name? Well, last week I remembered that I already had a similar freeware tool, called TableBob, which can convert Tables and records to Object Pascal source code (to recreate the table) and HTML. So I've decided to extend TableBob with the abilities to produce XML compatible with `ClientDataSet` as well. By the time you read this, the new version of TableBob will be available on my website at www.drbob42.com/tools (and rest assured that a Linux edition will ship shortly after Delphi for Linux is available).

Next Time

We have seen that producing XML is in fact quite similar to producing HTML. And from there, it's only a small step to producing WML, the Wireless Markup Language that can be used in WAP (Wireless Application Protocol) applications. The hype (and truth) about WML and WAP, and techniques to produce WAP WML applications using Delphi 5, are the topics of next month's column, *so stay tuned...*

Acknowledgements

I would like to thank my TAS-AT DOC colleague Arnim Mulder for his welcome assistance and suggestions while working with the socket server and client components.

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is an @-consultant for TAS Advanced Technologies and co-founder of the Delphi OplossingsCentrum (at www.tas-at.com/doc), as well as a freelance author and speaker.