Under Construction: Local ClientDataSets

by Bob Swart

You might think of MIDAS just as a multi-tier solution, but there are parts of MIDAS that can be used for local solutions too. Specifically, I'm talking about the `ClientDataSet` component here: a DBMS in itself, which is also contained within the single MIDAS.DLL. The news about Kylix and dbExpress emphasises the future importance of MIDAS and related technologies for future releases of Delphi (Windows, Linux, ...?). So this month I'll focus on using local `ClientDataSets`.

The `ClientDataSet` component can be found on the `Midas` tab of Delphi 5 Enterprise. Yes, I'm afraid that even though we're only using it for local database access, the `ClientDataSet` is only part of Delphi 5 Enterprise. However, some of the supporting utilities work with other versions and editions of Delphi as well, including even Delphi 1. But before we get onto the supporting act, let's first start with some explaining.

Why Local ClientDataSets?

The benefits of using `ClientDataSet` as a local `DataSet` over the BDE, for example, are numerous. First of all, the `ClientDataSet` is contained within a single DLL called MIDAS.DLL. Just drop it into the Windows\System directory (or leave it in the same directory as your executable) and you're in business. Compare this to installing the BDE on your client machine. And even if you decide to use ADO, which will probably be on your user's machine already, you need to make sure the actual database backend (Access, SQL Server, etc) is also present and accounted for. In short, the MIDAS.DLL is probably one of the easiest to install database engines I've seen so far.

The `ClientDataSet` component, implemented by this MIDAS.DLL, is also one of the fastest `DataSet` implementations I've seen. Sorting, filtering, all with blazing speed (we'll see some examples of this next month, when we actually start using local `ClientDataSet` components in applications).

How come it's so fast? Well, that's actually one of the potential downsides of the `ClientDataSet`. You see, everything is managed in memory. And every operation (sorting, filtering, searching and so on) is also done in memory. This explains the speed, but it also means that for a really large `ClientDataSet`, you also need a large amount of memory.

Next month, we'll look at the possibilities of using standalone `ClientDataSet` components (inside Windows applications, internet applications or simple console applications). This time, however, we'll focus on loading the `ClientDataSets` with data.

Feeding Local ClientDataSets

The easiest way to load a `ClientDataSet` at design-time is to right click on the `ClientDataSet` component and select the `Assign Local Data` pop-up menu option. This will show a dialog that lists all available `DataSets` (tables, queries, etc) that are available on the form or data module itself. If you pick one, then all data from that `DataSet` will be assigned to the `ClientDataSet`. You can then remove the source `DataSet` from the form or data module, and be left with a standalone `ClientDataSet` only. Note that you still need to remove the `DBTables` unit if you used a BDE table or query and want to make your new application BDE-free.

A `ClientDataSet` can also load and store its information on disk. At design-time you can use the `Load From File` and `Save To File` pop-up menu options to accomplish this. The `ClientDataSet` component also contains methods to do this, as well as `LoadFromStream` and `SaveToStream` methods, which you can redirect to different kinds of streams (like a `TMemoryStream`, right before sending this stream over a socket connection, for example, but we'll look at that next time).

`ClientDataSet` can load and store two kinds of data formats. The first one is typically called 'cds' format, and is the internal (and undocumented) binary format. Small, native and almost impossible to share (except with other `ClientDataSet` components from Delphi 5 and C++Builder 5). The second data format that `ClientDataSets` support is XML. And we've all heard that XML means eXtended Markup Language, and is often used in the same sentence with words like 'open', 'cross-platform' and 'portable'. Unfortunately, once you start working with XML, you'll quickly realise that it is just a language and nothing more (not a magic potion).

For the remainder of this article, we'll focus on the XML data format used by `ClientDataSet` and ways to generate and use it beyond the `ClientDataSet`.

DataSet To XML

Once a `ClientDataSet` is loaded, we can save the contents to a file. At design-time, this means a right click with the mouse on the `ClientDataSet` component and, after you've selected the `Save To File` pop-up menu choice, you can specify a filename.

To save some work, we can use the `FileName` property: give this property a value and the `ClientDataSet` will load itself (if the file is available) when activated, and save itself when deactivated (when the application closes). The file will always be in 'cds' format, except when you give it an .XML file extension. This obviously gives us an XML file and is the easiest way to get a first look at the XML generated by `ClientDataSet`.

ClientDataSet XML

The ClientDataSet XML format consists of a header (first line) and

a data packet, which in its turn contains two blocks: meta data and row data. The meta data contains the field definitions, whilst the row data contains the individual records. Note that the row data can be empty (when you only have an empty table definition), but the meta data should not be empty (unless you have a table with no fields).

Note that this XML format differs from the XML format that is generated by ADO, such as by an `ADODataSet` component, for example. The latter contains more detail and uses some different tagnames. So, unfortunately, a `ClientDataSet` cannot load an XML file generated by an `ADODataSet`, or vice versa: you will get an access violation while loading, although a more descriptive error message might be helpful). *[What's that about XML being the new ascii? Hmmm. Ed]* The `ClientDataSet` XML file is just one line, by the way, and Listing 1 shows the XML file with some editorial formatting applied, so that humans can read it and actually make sense of it!

Why is it important to examine the `ClientDataSet` XML format in more detail? Well, mainly because we need a way to feed the `ClientDataSet` with data. And XML seems to be the only way to feed it from an external source. Although the `ClientDataSet` itself can generate XML (once it is loaded by a local 'source' `DataSet`), we do not always have a 'source' `DataSet` available. Besides, there's a licence fee rule that specifies that a MIDAS deployment licence is required if a MIDAS data packet is transferred from one machine to another. And although I'm still planning to write this article about *local* `ClientDataSets`, it would be nice to be able to generate XML 'input' for `ClientDataSets` without needing MIDAS (or a MIDAS licence) to do so, even when generating them from another machine.

➤ *Listing 1: ClientDataSet XML.*

```
<?xml version="1.0" standalone="yes"?>
<DATAPACKET Version="2.0">
<METADATA>
<FIELDS>
 <FIELD attrname="FieldName" fieldtype="string" WIDTH="100"/>
 <FIELD attrname="FieldType" fieldtype="string" WIDTH="24"/>
 <FIELD attrname="Size" fieldtype="string" WIDTH="4"/>
</FIELDS>
</METADATA>
<ROWDATA>
</ROWDATA>
</DATAPACKET>
```

➤ *Listing 2: Unit TableXML with DataSetXML.*

```
unit TableXML;
interface
uses DB;
function DataSetXML(DataSet: TDataSet; const FileName:
  String): Integer;
implementation
uses SysUtils, TypInfo;
function DataSetXML(DataSet: TDataSet; const FileName:
  String): Integer;
var
  F: System.Text;
  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]) < 32) or (Str[i] = '"') then begin
        Insert('&#'+IntToStr(Ord(Str[i]))+';',Str,i+1);
        Delete(Str,i,1)
      end
    end;
    Result := Str
  end {EnCode};
begin
  Result := -1;
  ShortDateFormat := 'YYYYMMDD';
  System.Assign(F,FileName);
  try
    System.Rewrite(F);
    writeln(F,'<?xml version="1.0" standalone="yes"?>');
    writeln(F,'<DATAPACKET Version="2.0">');
    with DataSet do begin
      writeln(F,'<METADATA>');
      writeln(F,'<FIELDS>');
      if not Active then
        FieldDefs.Update;
      for i:=0 to Pred(FieldDefs.Count) do begin
        write(F,'<FIELD ');
        if Print(FieldDefs[i].Name) <> FieldDefs[i].Name
          then { fieldname }
          write(F,'fieldname="',FieldDefs[i].Name,'" ');
        write(F,'attrname="',Print(FieldDefs[i].Name),
          '" fieldtype="');
        case FieldDefs[i].DataType of
          ftString, ftFixedChar, ftWideString :
            write(F,'string');
          ftBoolean  : write('boolean');
          ftSmallint : write(F,'i2');
          ftInteger  : write(F,'i4');
          ftAutoInc  : write(F,
            'i4" readonly="true" SUBTYPE="Autoinc');
          ftWord, ftFloat : write(F,'r8');
          ftCurrency : write(F,'r8" SUBTYPE="Money');
          ftBCD      : write(F,'fixed');
          ftDate     : write(F,'date');
          ftTime     : write(F,'time');
          ftDateTime : write(F,'datetime');
          ftBytes    : write(F,'bin.hex');
          ftVarBytes, ftBlob : write(F,
            'bin.hex" SUBTYPE="Binary');
          ftMemo     : write(F,'bin.hex" SUBTYPE="Text');
          ftGraphic, ftTypedBinary : write(F,
            'bin.hex" SUBTYPE="Graphics');
          ftFmtMemo  : write(F,
            'bin.hex" SUBTYPE="Formatted');
          ftParadoxOle, ftDBaseOle : write(F,
            'bin.hex" SUBTYPE="Ole')
        end;
        if FieldDefs[i].Required then
          write(F,'"required="true');
        if FieldDefs[i].Size > 0 then
          write(F,'"WIDTH="',FieldDefs[i].Size);
        writeln(F,'"/>')
      end;
      writeln(F,'</FIELDS>');
      writeln(F,'</METADATA>');
      if not Active then
        Open;
      writeln(F,'<ROWDATA>');
      Result := 0;
      while not Eof do begin
        Result := Result + 1;
        write(F,'<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
            write(F,Print(Fields[i].FieldName),'="',
              EnCode(Fields[i].AsString),'" ');
        writeln(F,'/>');
        Next
      end;
      writeln(F,'</ROWDATA>')
    end;
    writeln(F,'</DATAPACKET>')
  finally
    System.Close(F)
  end
end;
end.
```

Table To XML

This is the point where it gets interesting for Delphi Professional users too. Apart from using a `ClientDataSet` to convert a `DataSet` to an XML file, you can just take a normal `DataSet` and write some code to turn it into XML data yourself. All the information that we need (field information such as field name, field type and size) is stored in a `DataSet`. And the same holds for the field name and values to produce the row data. The fact that this isn't that hard to do is proved by Listing 2: less than 100 lines of code for the `TableXML` unit with the `DataSetXML` function.

When I wrote this `DataSetXML` function, I tried to mimic the functionality of the output that the original `ClientDataSet` generates. I did add some line breaks to make output more readable, but I tried to remove some spaces between fields to make it smaller. I then used another `ClientDataSet` component to verify that the data represented by the XML file was indeed the same. When I used Internet Explorer to show the XML inside a browser, however, it revealed that you can't remove the spaces between field-value pairs. There must be a space after each ending double-quote and before the following field name. In HTML and the *official* XML definition such whitespace has no meaning and can be removed, thereby reducing the size of the XML file itself. Using both `ClientDataSet` and Internet Explorer as test cases made sure the XML file was correct both syntactically and semantically (Internet Explorer will show syntax errors and `ClientDataSet` will just produce an Access Violation when the XML is broken somehow).

Note that this function can be extended in several areas. First of all, I'm not converting all possible field types, but only the 23 most commonly used in my personal situation (I've left out some of the Oracle Blob fields, for example). Also note that of all the binary fields (`Blob`, `Memo`, `Graphic`, etc) I only display the `Memo` field value inside the XML file: the others are just ignored, as `ClientDataSet` itself does when generating XML data. Next time, I'll be looking at a way to extend this functionality in the `DataSetXML` function, but for now it'll do just fine.

With this unit, you can turn any `DataSet` (BDE, ODBC, ADO, or even `ClientDataSet` or a custom `DataSet`) into an XML file dump, which in turn can be used to 'feed' a `ClientDataSet` again.

➤ *Listing 3: Biolife to XML.*

```
program BIOLIFE;
{$APPTYPE CONSOLE}
uses DBTables, TableXML;
var Table: TTable;
begin
  Table := TTable.Create(nil);
  try
    Table.DatabaseName :=
      'DBDEMOS';
    Table.TableName :=
      'BIOLIFE.DB';
    DataSetXML(Table,
      'C:\BIOLIFE.XML',True);
  finally
    Table.Free;
  end;
end.
```

XML To Table

Now that we have a means of converting a `DataSet` to XML, what about the other way around? Can we go from XML to a `DataSet` again? Yes, of course we can, because a `ClientDataSet` can use the generated XML to 'load' itself with a `DataSet`. And this also means that we can 'create' a new table by defining the fields in XML format.

For example, the BIOLIFE.DB table can be turned into XML using the program in Listing 3. This time we're not using a `ClientDataSet`, so it compiles with Delphi 5 Professional and Delphi 4, and with a little work you could make it compile with other versions of Delphi.

The resulting XML file, generated by our `DataSetXML` function, is somewhat more readable than the single-line XML generated by the `ClientDataSet` component. The interesting part is the `<METADATA>` section of the XML file, which contains the field definitions, and is defined in Listing 4.

➤ *Listing 4: Biolife XML metadata.*

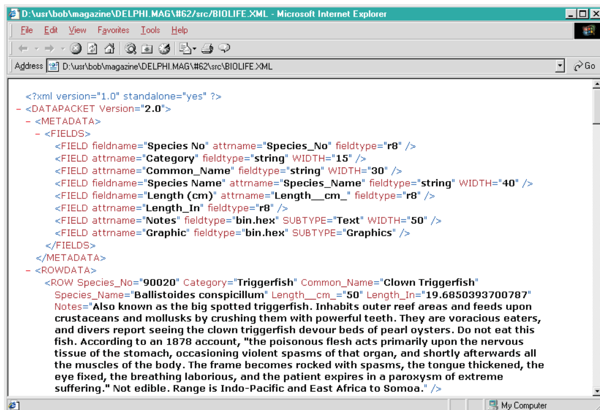
```
<METADATA>
<FIELDS>
 <FIELD fieldname="Species No" attrname="Species_No" fieldtype="r8"/>
 <FIELD attrname="Category" fieldtype="string" WIDTH="15"/>
 <FIELD attrname="Common_Name" fieldtype="string" WIDTH="30"/>
 <FIELD fieldname="Species Name" attrname="Species_Name"
   fieldtype="string" WIDTH="40"/>
 <FIELD fieldname="Length (cm)" attrname="Length__cm_" fieldtype="r8"/>
 <FIELD attrname="Length_In" fieldtype="r8"/>
 <FIELD attrname="Notes" fieldtype="bin.hex" SUBTYPE="Text" WIDTH="50"/>
 <FIELD attrname="Graphic" fieldtype="bin.hex" SUBTYPE="Graphics"/>
</FIELDS>
</METADATA>
```

Note that when the field name contains characters that cannot be part of an identifier (such as spaces or brackets), then these characters are replaced by an underscore, and both the original `fieldname` and (fixed) `attrname` are present in the field definition.

Internet Explorer version 5 and higher contain some XML support. Basically, this means that any XML structure (like the `FIELD` inside `FIELDS` and `ROW` inside `ROWDATA`) is shown as a tree structure, which can be seen in Figure 1. Syntax highlighting also helps to make this XML file more readable.

The table definition of Listing 4 is actually pretty easy to read, with the exception of the `fieldtype` values, for which I've made the translation table shown in Table 1 between Delphi and XML field types (note that I'm still using `r8` instead of `i4` for Word, based on what the original `ClientDataSet` seems to generate, but I may have to get back on that next month).

Using this translation table and the knowledge gathered so far, I decided to make another XML utility. This time a TableXML tool that collects a set of `FieldName`, `FieldType` and (optionally) `Size` information, the basic information for a field inside a `DataSet`, and produces an XML table definition for it. Such an XML definition would consist of the `FIELDS` information only, and the `ROWDATA` would be empty, of course. Having such empty table definition files can be very useful, as these correspond to the empty tables that you sometimes need to ship with your applications. `ClientDataSets` can read



➤ *Figure 1: Biolife XML in Internet Explorer.*

these XML table definitions and use those to connect to data-aware controls. Like a `DBGrid` (see Figure 2) that is connected to a `ClientDataSet` named `CDS` which just happens to load an XML table definition like the one we saw in Listing 1.

The idea of using a `ClientDataSet` loaded with an XML file (generated by an earlier version of this application) is indeed a nice demonstration of bootstrapping. The current edition of this application shows a grid where you can enter values for `FieldNames`, select values from a drop-down list for `FieldTypes` and enter a value for `Size` (note that at this time you need to know that `Size` is only relevant for `String` or `Memo/Blob` fields).

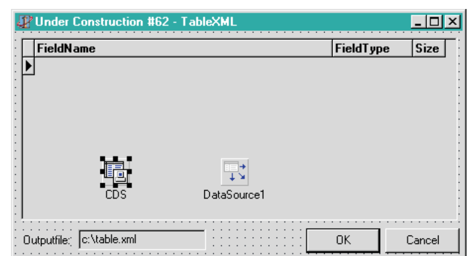
The code to traverse the records of the `ClientDataSet` to produce an XML table definition (inside the specified file), can be seen in Listing 5. Note that this code is a bit similar to that of Listing 2, but different in a number of places. We cannot inspect the actual `FieldDefs` this time, for example, so need to look at the actual string for the `FieldType` (which results in a long `if..then..else` sequence). And of course this time we don't have to produce `ROWDATA` itself, because we are only producing a table definition file.

Using the unit from Listing 5, we can actually go back and produce the output from Listing 1 again.

Conclusion

So, what have we got at the end of this article? First of all, a handy routine to convert a `DataSet` to an

➤ *Figure 2: TableXML using ClientDataSet.*



➤ *Listing 5: Producing XML Table Definitions.*

```
unit Unit62;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Db, DBClient, Grids, DBGrids, StdCtrls;
type
  TForm1 = class(TForm)
    CDS: TClientDataSet;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    ButtonCancel: TButton;
    ButtonOK: TButton;
    Label1: TLabel;
    EditFileName: TEdit;
    CDSFieldName: TStringField;
    CDSFieldType: TStringField;
    CDSSize: TStringField;
    procedure ButtonOKClick(Sender: TObject);
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.ButtonOKClick(Sender: TObject);
var
  F: System.Text;
  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 {Printable};
begin
  System.Assign(F,EditFileName.Text);
  try
    System.Rewrite(F);
    writeln(F,'<?xml version="1.0" standalone="yes"?>');
    writeln(F,'<DATAPACKET Version="2.0">');
    writeln(F,'<METADATA>');
    writeln(F,'<FIELDS>');
    CDS.First;
    while not CDS.Eof do begin
      write(F,' <FIELD ');
      if Print(CDSFieldName.AsString) <>
        CDSFieldName.AsString then { fieldname }
        write(F,'fieldname="'+CDSFieldName.AsString+'" ');
      write(F,'attrname="' + Print(CDSFieldName.AsString)+
        '" fieldtype="');
      if CDSFieldType.AsString = 'AutoInc' then
        write(F,'i4" readonly="true" SUBTYPE="Autoinc')
      else if CDSFieldType.AsString = 'Integer' then
        write(F,'i4') // Integer
      else if CDSFieldType.AsString = 'Smallint' then
        write(F,'i2') // Smallint
      else if (CDSFieldType.AsString = 'String') or
        (CDSFieldType.AsString = 'WideString') or
        (CDSFieldType.AsString = 'FixedChar') then
        write(F,'string') // FixedChar, String, WideString
      else if CDSFieldType.AsString = 'Memo' then
        write(F,'bin.hex" SUBTYPE="Text') // Memo
      else if CDSFieldType.AsString = 'BCD' then
        write(F,'fixed') // BCD
      else if (CDSFieldType.AsString = 'Blob') or
        (CDSFieldType.AsString = 'VarBytes') then
        write(F,'bin.hex" SUBTYPE="Binary')
      else if CDSFieldType.AsString = 'Boolean' then
        write(F,'boolean') // Boolean
      else if CDSFieldType.AsString = 'Bytes' then
        write(F,'bin.hex') // Bytes
      else if CDSFieldType.AsString = 'Currency' then
        write(F,'r8" SUBTYPE="Money') // Currency
      else if CDSFieldType.AsString = 'Date' then
        write(F,'date') // Date
      else if CDSFieldType.AsString = 'DateTime' then
        write(F,'datetime') // DateTime
      else if (CDSFieldType.AsString = 'dBaseOle') or
        (CDSFieldType.AsString = 'ParadoxOle') then
        write(F,'bin.hex" SUBTYPE="Ole')
      else if (CDSFieldType.AsString = 'Float') or
        (CDSFieldType.AsString = 'Word') then
        write(F,'r8') // Float, Word
      else if CDSFieldType.AsString = 'FmtMemo' then
        write(F,'bin.hex" SUBTYPE="Formatted') // FmtMemo
      else if (CDSFieldType.AsString = 'Graphic') or
        (CDSFieldType.AsString = 'TypedBinary') then
        write(F,'bin.hex" SUBTYPE="Graphics')
      else if CDSFieldType.AsString = 'Time' then
        write(F,'time'); // Time
      if Length(CDSSize.AsString) > 0 then
        write(F,'" WIDTH="',CDSSize.AsString);
      writeln(F,'"/>');
      CDS.Next
    end;
    writeln(F,'</FIELDS>');
    writeln(F,'</METADATA>');
    writeln(F,'<ROWDATA>');
    writeln(F,'</ROWDATA>');
    writeln(F,'</DATAPACKET>');
  finally
    System.Close(F)
  end;
  Close
end;
end.
```

Delphi Field Type	XML Field Type
String, WideString, FixedChar	string
Boolean	boolean
Smallint	i2
Integer	i4
AutoInc	i4, readonly=true subtype=Autoinc
Word, Float	r8
Currency	r8, subtype=Money
BCD	fixed
Date	date
Time	time
DateTime	datetime
Bytes	bin.hex
VarBytes, Blob	bin.hex subtype=Binary
Memo	bin.hex subtype=Text
Graphic, TypedBinary	bin.hex subtype=Graphics
FmtMemo	bin.hex subtype=Formatted
ParadoxOle, dBaseOle	bin.hex subtype=Ole

➤ *Table 1: Delphi field types mapped to XML field types.*

XML file (complete with data), and secondly, a little tool (which is in need of a good name) to enter `DataSet` field type specifications and produce an XML file for a new table definition. Both types of XML file can be used as input for `ClientDataSet` components in a standalone environment, which is what we will be looking into next month...

Next Time

We'll continue with some examples of local `ClientDataSets` in real world applications (using XML input), which will focus on some of the major benefits and a few potential problems of `ClientDataSets`. We'll also experiment with combining 'foreign' datasources (no pun intended) with `ClientDataSets`. And lastly we'll keep our ears open for any Kylix news that may be forthcoming.

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