Under Construction: DataCLX And dbExpress

by Bob Swart

This month, I'm going to explore dbExpress, the new cross-platform database access layer currently available in Borland Kylix (on Linux) and which in the future will also be available in Borland Delphi 6 (on Windows).

Exit BDE?

Before I start with dbExpress, however, you should know that the Borland Database Engine (BDE) is not available for Kylix. In fact, there is no BDE for Linux, so this poses a potential problem for existing BDE tables in (say) Paradox format. I'll cover some BDE migration techniques at the end of this article, showing how you can move existing BDE data to dbExpress or MyBase (MyBase is the personal XML-based database engine which is included with Kylix).

What does this mean for the future of the Borland Database Engine? Is the BDE dead? Well, not officially. There has been no announcement regarding the dismissal of the BDE, but at the same time, I haven't heard any news of future BDE updates or enhancements either. Delphi 6 will ship with both the BDE and dbExpress, that much is sure. But for cross-platform database access, the BDE will be a no-no, and hence the focus this month on dbExpress (and moving from the BDE to dbExpress).

What Is dbExpress?

dbExpress is a cross-platform, lightweight, fast and open database access architecture. A dbExpress driver must implement a number of interfaces to get metadata, execute SQL queries or a stored procedure, and return a unidirectional cursor. We'll get back to this in a moment.

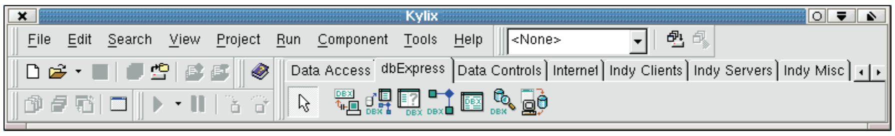
The Kylix Component Library is called CLX (Component Library for X-platform). And CLX is divided into four parts: BaseCLX, VisualCLX, NetCLX and DataCLX. A question that comes up often is where exactly dbExpress fits in. Obviously, dbExpress and DataCLX are connected. In fact, that's exactly what's happening: dbExpress is the low-level database access driver and DataCLX is a set of components that connect to this driver. Not the visual data-aware components, mind you (these are part of VisualCLX), but the data access components that we can use to specifically work with the data (regardless of the differences in the underlying data).

As you probably know by now, Kylix version 1.0 is released in two editions: Kylix Desktop Developer and Kylix Server Developer. Kylix Desktop Developer includes dbExpress drivers for InterBase and MySQL; Kylix Server Developer adds drivers for DB2 and Oracle 8i. Future native SQL drivers for dbExpress are expected, perhaps even before the next edition of Kylix is available (which should be the Enterprise Studio).

By the way, although the Kylix CD itself contains InterBase 5.6, you can find a copy of InterBase 6 on the companion CD. So you're free in your choice of InterBase version on Linux.

Custom dbExpress

And that's not all, because dbExpress was created as an open database architecture, meaning that anyone can write a dbExpress-compatible driver for use with Kylix and future versions of Delphi. An article about the dbExpress internals by Ramesh Theivendran, the architect of dbExpress, was published on the Borland Community website in July of last year. Although this was just a draft specification, it made it clear that anyone can write a driver.



➤ *Figure 1*

As a practical example, Easysoft has developed a dbExpress Gateway for ODBC, which can be used to connect to UNIX ODBC, and, via its ODBC-ODBC Bridge, even to a remote Microsoft SQL Server, Access or other Windows ODBC driver.

Components

If you start Kylix and take a look at the Component Palette, you'll notice a tab called `dbExpress` and no tab called `DataCLX`. This is a mistake in my view; the tab should have been called `DataCLX`, since the components are part of CLX, and only wrap the dbExpress functionality. Using Kylix Server Developer, the `dbExpress` tab contains seven components: `TSQLConnection`, `TSQLDataSet`, `TSQLQuery`, `TSQLStoredProcedure`, `TSQLTable`, `TSQLMonitor` and the last one is `TSQLClientDataSet`.

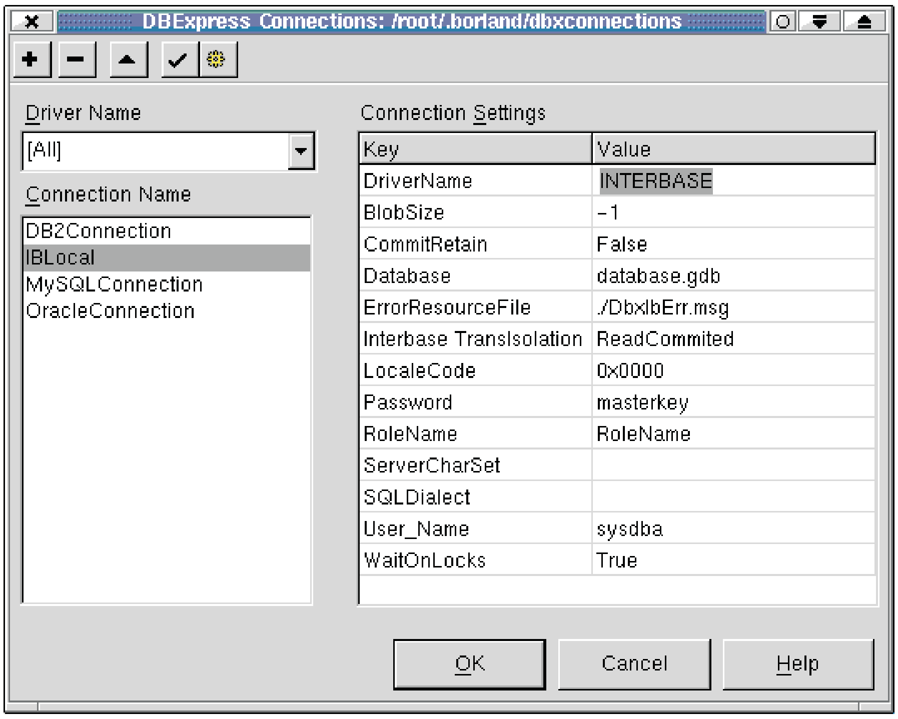
TSQLConnection

The `TSQLConnection` component is literally the connection between the dbExpress drivers and the other DataCLX components. If you drop this component onto a Kylix form or data module, you will see only 12 properties. The one that's probably most used is the `ConnectionName` property, which can be assigned with one of the values from the combobox. On my Kylix Server installation, I have the choice of DB2Connection, IBLocal, MySQLConnection and OracleConnection. If you select IBLocal, then the DriverName property gets the value INTERBASE, the GetDriverFunc property gets the value getSQLDriverINTERBASE, the LibraryName property gets the value libsqlib.so.1 and the Vendorlib property gets the value libgds.so.0: all automatically, based on the value IBLocal for the ConnectionName.

You can open the Params stringlist editor to edit the values of the parameters. These are also automatically filled in, by the way, when you select a value for the ConnectionName property. If you do not want this to happen, for example when you are writing some non-visual code to access databases and you want to provide your own parameter values, then you can set the LoadParamsOnConnection to False.

If you right-click on the TSQLConnection component, you will see the connection settings for the four connection names. As you can see in Figure 2, the Database is set to database.gdb (by default). You need to set that to an actual InterBase database, such as the employee.gbd which on my PC is at /usr/interbase/examples/employee.gdb.

➤ *Figure 2*



DB2Connection
IBLocal
MySQLConnection
OracleConnection

Key	Value
DriverName	INTERBASE
BlobSize	-1
CommitRetain	False
Database	database.gdb
ErrorResourceFile	./DbxIbErr.msg
Interbase TransIsolation	ReadCommited
LocaleCode	0x0000
Password	masterkey
RoleName	RoleName
ServerCharSet	
SQLDialect	
User_Name	sysdba
WaitOnLocks	True

Once you have everything set right, you can set the Connected property to True (and either get an error message if the database cannot be found, or see the property get the value True indeed for success).

TSQLDataSet

Once you have a connected TSQLConnection component, you can use any of the other DataCLX components, such as the TSQLDataSet, which is the most 'general' of these components. Always start by setting the SQLConnection property of this component to (one of) the available TSQLConnection component(s). The TSQLQuery, TSQLStoredProc and TSQLTable components can be seen as special instantiations of the TSQLDataSet component. In fact, this reminds me a lot of ADOExpress, in which the TADODataSet component is the 'mother' of the TADOQuery, TADODataSet and TADOTable components. And both the Delphi 5 ADOExpress and Kylix (and future Delphi) dbExpress TxxxDataSet 'core' components share the CommandType and CommandText properties, with which you can determine the sub-type of the component. If you set the value of the CommandType property to ctQuery, then the CommandText property is interpreted as an SQL query. If you set the CommandType to ctStoredProc, then the CommandText specifies the name of the stored procedure. Finally, if you set CommandType to ctTable, then CommandText contains the names of the individual tables.

By the way, although they are hardly necessary (the TSQLDataSet component is flexible enough), I see the main purpose of the TSQLQuery, TSQLStoredProc and TSQLTable components being to help with migrating existing BDE code to dbExpress.

In our case, using the general TSQLDataSet component, we can set the CommandType to ctTable, and the CommandText to customer to select the customer table. If you set the Active property to True, you get live data at design-time, just as we've been used to with Delphi (and if you set the LoginPrompt property of the SQLConnection component to False, you don't even see the login dialog). Nothing special, nothing different. Yet.

Unidirectional?

We can now move to the Data Controls tab of the component palette, and use some of these to display the data we receive from the active TSQLDataSet component. Note that we cannot use all of these components right now without some special considerations. This is the place where the biggest difference between the BDE and the dbExpress architecture is present. TSQLDataSet (and the derived TSQLQuery, TSQLStoredProc and TSQLTable) returns a unidirectional cursor. Meaning that you can move forwards, but not backwards. Which isn't useful with a TDBGrid (we can only see one record at a time!), and watch out when using a TDBNavigator too, as clicking on the Back or First button will raise an exception!

So why a unidirectional cursor? Well, the obvious answer is speed. The BDE has never been our best friend (let's call it a good friend, or a friendly relative), but it has helped us with our small and simple database needs. Unfortunately, the BDE footprint and overhead hasn't been small. And BDE

```
var
  Posts: Integer = 0;
procedure TForm1.SQLClientDataSet1AfterPost(DataSet: TDataSet);
const
  MaxPosts = 7;
begin
  Inc(Posts);
  if Posts > MaxPosts then begin
    (DataSet AS TSQLClientDataSet).ApplyUpdates(-1);
    Posts := 0
  end
end;
```

➤ *Listing 1*

tables have never been known for their amazing speed. And that's an area where Borland wanted to show some real improvements. The new dbExpress architecture is designed with speed in mind. And hence unidirectional cursors as a result set, with no overhead for buffering data or managing metadata.

A unidirectional cursor is especially useful when you really only need to see the results once, or need to walk through your result set from start to finish (again once), for example in a `while not eof` loop, processing the results of a query or stored procedure. Real-world situations where this is useful include reporting, and web server applications that produce dynamic web pages as output.

However, you will quickly realise that, in a GUI driven environment, using visual data-aware controls, the user will often want to go back one record. So you need to somehow cache these records in order to be able to show them in a grid and to browse backwards as well as forwards. That's where the `TClientDataSet` comes in, which you may remember from my past MIDAS-related articles (or the recent MIDAS MasterClasses I gave, organised by the UK Borland User Group). It's entirely possible to use a `TDataSetProvider` (from the `Data Access` tab of the Kylix component palette) to hook up with the `TSQLDataSet` component, and then use a `TClientDataSet` to obtain its records from this `TDataSetProvider`. The result is a `ClientDataSet` that gets its records (once) from a unidirectional source: the `SQLDataSet`. The `DataSetProvider` is only used as a local transportation means. This combination works very well, and in fact ended up as a single component in its own right: the `TSQLClientDataSet` component.

TSQLClientDataSet

The `TSQLClientDataSet` component combines the speed and light weight of the new dbExpress architecture with the caching and speed abilities of the well-known `TClientDataSet` component. And there is another reason why we want to use the `TSQLClientDataSet` at times: the unidirectional `TSQLDataSet` (and derived components) have an additional limitation in that they cannot be used to update the data in the dataset. For that, you have to use a `TClientDataSet` component (like the `TSQLClientDataSet`). How can this be done, you may ask. Well, as a regular `ClientDataSet`, all the changes that are made locally are cached inside the `TSQLClientDataSet` component. And all the changes are sent back (resolved) to the actual database (via the dbExpress driver in this case) by calling the `ApplyUpdates` method also inherited from `TClientDataSet`. The `ApplyUpdates` method call will use the provider to send so-called delta packets to the database server. Something a lone `TSQLDataSet` component isn't capable of.

Isn't it a nuisance to have to call `ApplyUpdates`? Suppose you forget to call it in your application. Or the end-user just changes a lot of data, but is surprised that other users don't see his changes because he never calls the `ApplyUpdates` method. At first sight, this `ClientDataSet` layer seems only to add potential confusion. But the confusion can be solved by making sure the `ApplyUpdates` method is called on a frequent basis. In fact, you can easily use the `AfterPost` event of the `TSQLClientDataSet` component to call the `ApplyUpdates` method, which will make sure that, after every (local) post to the `ClientDataSet`, the data is immediately also sent as an update packet to the database server. And in cases where you don't want to

➤ *Listing 2*

```
{$APPTYPE CONSOLE}
program dbAlias;
uses
  Classes, SysUtils, DB, DBTables, Provider, DBClient;
var
  i: Integer;
  TableNames: TStringList;
  Table: TTable;
  DataSetProvider: TDataSetProvider;
  ClientDataSet: TClientDataSet;
begin
  TableNames := TStringList.Create;
  with TSession.Create(nil) do
  try
    AutoSessionName := True;
    GetTableNames(ParamStr(1), '', True, False, TableNames);
  finally
    Free
  end {TSession};
  Table := TTable.Create(nil);
  DataSetProvider := TDataSetProvider.Create(nil);
  ClientDataSet := TClientDataSet.Create(nil);
  try
    Table.DatabaseName := ParamStr(1);
    for i:=0 to Pred(TableNames.Count) do begin
      writeln(Table.TableName);
      Table.TableName := TableNames[i];
      Table.Open;
      DataSetProvider.DataSet := Table;
      ClientDataSet.SetProvider(DataSetProvider);
      ClientDataSet.Open;
      ClientDataSet.SaveToFile(ChangeFileExt(Table.TableName,'.xml'));
      ClientDataSet.Close;
      Table.Close
    end
  finally
    Table.Free
  end
end.
```

do that, because it may take additional time to make that call, you can always 'save' your posts, increase an internal counter, and only call ApplyUpdates when the counter reaches a certain number of posts (after which you also need to reset the counter, of course). The latter can be implemented as shown in Listing 1 (which only compiles with Kylix, by the way, and not with Delphi 5).

Since the TSQLClientDataSet component is actually derived from the TClientDataSet, we could have cast the DataSet parameter to a TClientDataSet in order to make the code a bit more general, but I'm sure you get the idea. Note that the TSQLDataSet (or derived) component doesn't have the AfterPost event. Obviously, that's because these components cannot post, but return a unidirectional read-only cursor.

TClientDataSet

Let's return to the 'normal' TClientDataSet now, and see if it's still the same little powerhouse we've known from Delphi 5. Yes, it is, and its internal data format is compatible with its Windows counterpart (it would have surprised me if that wasn't the case, but it's good to confirm). This means that we can use a TClientDataSet on either Windows or Linux and save its contents to a binary .cds or readable .xml format, and then send that file to the other platform, and load it in another TClientDataSet component. This is the cornerstone of not only multi-tier but multi-tier over cross-platform applications! The only thing that's stopping us is the fact that the current TDataSet-Provider component in Kylix can only handle local datasets (ie local database servers) and cannot yet make any remote database connections using any of the TxxxConn-ection components we've used in Delphi 5. This will all have to wait until the Kylix Enterprise Studio ships, but why let that stop us? In the meantime, we can still use anything from FTP to a TCP/IP socket connection to send a saved ClientDataSet file from one platform to another, load it in again, and use it (or resolve the changes to a database server).

Migrating From BDE

In fact, the TClientDataSet component in Kylix, as well as Delphi 5, means a quick and easy way to migrate local database tables (such as, indeed, BDE tables in Paradox or dBASE format). This is the first way in which you can migrate from the BDE to dbExpress: migrating data. The second way is by migrating the application as well (a future topic).

To continue now with the way to migrate data from the BDE to a native ClientDataSet format, consider the code in Listing 2 for a new utility called dbAlias that I've written, which will convert all the tables from a given alias (passed on the command-line passed) into XML files.

This is a quick-and-dirty way to convert your existing BDE aliases containing Paradox and dBASE files to XML, after which you can put these files on a Linux box (using FTP, a network connection or even a floppy disk) and load them in Kylix using a TClient-DataSet component. Obviously, this code only compiles in Delphi 5, and not in Kylix.

Follow-Up

If you want to know more about dbExpress and the internals of dbExpress, then wait just another month for an article by Guy Smith-Ferrier in your favourite Delphi magazine (that's this one, obviously), where he will follow up with lots more interesting details about this new cross-platform database access layer called dbExpress.

Next Time...

In the meantime, I'll be working on an exploration of NetCLX, or rather: WebBroker support in Kylix. The Server Developer edition, that is, since the WebBroker components are not in the Desktop Developer edition.

Anyway, we will explore Web-Broker, and see how we can write Apache web server applications for Linux using Kylix (which might again show some glimpses of the future that Delphi 6 might bring us regarding the forthcoming support for Apache on Windows). And of course we'll be using some dbExpress unidirectional datasets (which, as I stated in this article, are perfectly suited for use in a web server application).

All this and more next month, *so stay tuned...*

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a professional IT-consultant for the Everest Kylix/Delphi OplossingsCentrum in Eindhoven, The Netherlands, and a freelance technical author.