The Late Session

by John O'Connell

Those of you using Delphi 2.0 might have noticed the `UpdateSQL` and `Session` components on the `Data Access` page of the component palette. The `UpdateSQL` component is new but the `Session` object is part of Delphi 1.0 and has existed since the days when Borland's database connectivity was known as the Open Database API (ODAPI), before the Borland Database Engine (BDE) arrived on the scene. Only now has `Session` been promoted to the component palette, but for what good reason? Well it's all to do with another new feature of Delphi 2.0 (and the Win32 architecture): the thread or `TThread` as known by Delphi.

BDE Objects

So what exactly is a session? The `TSession` object which encapsulates a BDE session provides a number of methods which return the names of aliases and databases, but there's more to sessions than just that. Let's look at the BDE architecture in a little depth. The BDE is object-based in design and each object is defined by its properties which are set when the object is created. BDE objects are created by applications using IDAPI (the Integrated Database API, used to call BDE services). A BDE application will make use of the following BDE objects:

- System
- Clients
- Sessions
- Database drivers
- Databases
- Cursors
- Query statements

A single `System` object controls all IDAPI resources used by applications running on the same machine and is created by the first application to initialise the BDE. All subsequently loaded BDE client applications will then use the same IDAPI `System` object. A single `Client` per application is created when that application initialises IDAPI. The `Client` controls all IDAPI resources used by that `Client`. The language used by the `Client` for error messages can be specified. The session basically provides the means to isolate database access operations without having to start another instance of the application. A default session is always created by the `Client` which can handle multiple sessions. Each session acts as a container for other database access objects (created at run-time) such as databases, cursors (tables) and query statements. Database drivers provide the connectivity layer between IDAPI and a physical database table. There are drivers for Paradox tables, dBase tables and the various SQL databases. Drivers provide information about their capabilities, such as whether the driver supports transactions or soft deletes, for example. A driver may also provide international language support used for the display of records in a database table. Databases are a collection of related tables and an application must successfully open a database before its tables can be accessed. Aliases can be defined to identify a particular database. For Paradox and dBase databases (which are essentially just a directory) no alias is required, but SQL databases must be identified by an alias. Aliases and the properties for IDAPI objects are set interactively using the BDE Configuration Utility which stores these properties in the IDAPI configuration file. The `System`'s properties in particular get a whole page to themselves!

IDAPI Objects And The VCL

As you'd expect, the data-access parts of the VCL encapsulate IDAPI objects, most of which are quite obvious (`TDatabase`, `TTable` etc), but some are not so obvious. I've already said that the `TSession` object encapsulates an IDAPI session, but where do the `System`, `Client` and drivers come into it? The `System` is never encapsulated and doesn't need to be, because it's just there and controls the IDAPI clients on one system. The same goes for the `Client` which is created (and may in turn create the `System` if no other clients exist) when the `Session` object (declared in DB.PAS) of type `TSession` is created. Drivers are not encapsulated within the VCL.

Sessions And Locks

I've already mentioned that a session is a container for databases and tables – therefore a session controls the locks on the databases and tables contained within that session. Let's investigate the relationship between sessions and locks with a very simple example.

The Delphi 2.0 application SESSDEM1 (included on the disk) contains two separate `TTable` objects, `Table1` and `Table2`, both opened on the same database and table and both positioned on the same record. Now `Table1` edits the record and a moment later `Table2` tries to deletes the same record which raises an exception saying that the record is locked in the same session. `Table2` then tries to delete the record and succeeds! That wasn't expected! Now `Table1` tries to post the changes to the now non-existent record, obviously fails and has to cancel the changes. But how come `Table2` was allowed to delete a locked record? Well, because `Table1` and `Table2` were opened in the same session, `Table1`'s record lock didn't prevent the record from being deleted as expected, because the default session owned the lock and could ignore it.

In a multi-threaded application the above situation could cause data-integrity problems when more than one thread accesses the same table. It's here that using the `Session` component to create another session saves the day. If

Table1 or Table2 had been opened in a separate session, Table2 would not have been able to delete the record locked by Table1: the attempt to delete the record would have raised an exception saying that the record is locked by another session. Try it for yourself with SESSDEM2 which uses a separate TSession called SessOne with which Table2 is associated.

With TSession comes a new Session property for the TTable, TQuery and TStoredProc which gives the wrong impression that tables and queries are directly associated with a session which is not the case – only databases are directly associated with the session and so TDatabase gets a Session property. The Session property for TTable and similar components actually refers to the session that owns the temporary TDatabase created at runtime for a TTable, TQuery or TStoredProc not connected to a persistent TDatabase object. However, from now on I'll be referring to TTables as if they were directly associated with sessions.

Delphi 2's TSession

So now we know how important the TSession can be, especially in a multi-threaded database application, let's examine just a few of its properties and methods. The Active property, when set to True, opens the TSession which allows any associated TDatabase (and hence any TDataset descendants owned by that TDatabase) to be opened. Setting the Active property to False closes down all owned TDatabases which in turn closes down all TDatasets – a convenient way of closing down many open tables in your application with a single line of code. An alternative to setting the Active property is to use the TSession Open and Close methods. Interestingly, opening a dataset associated with an inactive TSession opens that TSession.

The SessionName property is a unique identifier for the TSession and is used by the Session property of the other data access components. The name of the default session (the TSession named Session, declared in DB.PAS) created when the application initialises is called, strangely enough, Default, and should never be explicitly closed. The Databases property is an array of TDatabase components owned by the TSession. The function of the DatabaseCount property is pretty obvious.

The NetFileDir and PrivateDir properties relate to table and record locking. NetFileDir specifies the directory containing the shared network control file PDOXUSRS.NET which tracks all BDE client applications sharing Paradox tables on a single network. The network control file directory must be the same for all clients accessing the same tables otherwise table/record locking conflicts will occur and data integrity might be compromised. The file PDOXUSRS.LCK is created in any directory containing Paradox tables that are being accessed, and contains information about locks for each table in that directory. Every BDE client examines this file before attempting to place a lock on a table or record.

PrivateDir specifies the private directory for the session which contains all temporary files and tables created by the BDE (when performing local queries, restructuring tables, etc) for that session. The private directory for a session must be just that, private: any attempt to set a session's private directory to that of another session is not permitted by the BDE and will cause an error. The file PARADOX.LCK marks a directory as private. If another BDE client application tries to access Paradox tables in that directory it will fail. A PARADOX.LCK file is also created (alongside PDOXUSRS.LCK) in a shared directory containing tables being accessed by BDE clients. This stops a session making a shared directory its private directory. The file PDOXUSRS.LCK is also created in the private directory and contains information about locks owned by the session that uses that private directory.

The KeepConnections property specifies the default value of the KeepConnections property for a temporary TDatabase created at run-time and owned by that session. If KeepConnections is False and the temporary TDatabase has no tables open, the database connection will be dropped and the database closed, but if KeepConnections is True then the connection will persist regardless of whether any tables are open. This reduces network traffic between client and server when tables in a remote database are frequently opened and closed. The KeepConnections property of a persistent TDatabase overrides the KeepConnections property of the owning TSession, but making a call to TSession.DropConnections will close all active database connections regardless of the value of TDatabase.KeepConnections.

The TSession also provides two methods (OpenDatabase and CloseDatabase) to open and close each of its TDatabases and also provides methods (AddPassword, RemovePassword and RemoveAllPasswords) for controlling a password list used by password-protected Paradox tables. Adding the correct password to the password list will prevent a password dialog from popping up when a password protected Paradox table is opened. Other TSession methods retrieve information about: BDE aliases and database names (GetAliasNames and GetDatabaseNames), alias drivers and parameters (GetAliasDriverName and GetAliasParams), BDE driver names and their parameters (GetDriverNames and GetDriverParams), table names and stored procedure names for a specified database (GetTableNames and GetStoredProcNames). All potentially useful information which is interactively configured for each alias and database driver by using the BDE Configuration Utility.

But what's the difference between the information retrieved by GetAliasNames and GetDatabaseNames? The difference is that GetAliasNames returns all permanent BDE aliases (created with the BDE Configuration Utility) whereas GetDatabaseNames returns all BDE aliases plus any application specific aliases which are defined by the DatabaseName property of a

persistent TDatabase existing only for the running application's lifetime.

It's all very well having multiple sessions floating about in a Delphi 2.0 application but how can they be controlled? How can the application know what sessions have been created and how many of them are there? That's where the SessionList object comes in.

A TSessionList is a list of sessions owned by the IDAPI Client and is maintained by the TSession class. You can use TSessionList properties and methods to find out how many sessions exist (using the Count property) and what their names are (using GetSessionNames). You can obtain a pointer to a specific session in the list using the List or Sessions properties. To open a particular session use the OpenSession method which takes a session name as its only parameter. If the session specified by this parameter exists then it is opened, otherwise a new TSession instance is created and added to SessionList. Never use TSession.Create to create a new session, always use Sessions.OpenSession. Using TSession.Create will create another default session, which will in turn attempt to create a new client object, which will fail.

I've stated that Delphi 2.0 provides multiple BDE session handling to get around the potential data integrity problems that could arise in a multi-threaded database application. Delphi 1.0 doesn't support multi-threading and so doesn't need support for multiple sessions, which is the reason why Borland didn't provide such support. The Session object in Delphi 1.0 encapsulates the default BDE session. Attempting to create a new instance of TSession... well, you know the story. There is no such thing as SessionLists in Delphi 1.0, although the TSession properties and methods are more or less the same (apart from SessionName and Active) in both versions of Delphi.

It is possible to write a pseudo-multi-threading database application with Delphi 1.0 by judicious use of the Application.ProcessMessages method. In such cases support for multiple sessions is desirable in Delphi 1.0, so what can be done to remedy the situation?

Multiple Sessions In Delphi 1

The only way to implement multiple sessions support for Delphi 1.0 is to make direct calls to the BDE. The BDE function calls relating to session creation and control are DbiStartSession which starts a new session, DbiSetCurrSession which sets the active session and DbiCloseSession which closes the session.

DbiStartSession takes three parameters: a pointer to the session name, a pointer to the session handle used to identify the session and a pointer to the network control file directory. In addition to creating the new session, DbiOpenSession makes the new session the current session. The only parameter that we must specify is the session handle, the others can be passed as nil.

DbiSetCurrSession takes a session handle as its one parameter and activates the specified session. Usefully, if the session handle parameter is passed as nil the default session is made current.

DbiCloseSession takes a session handle as a parameter and closes the session. If the session being closed is the current non-default session then the default session becomes current. Closing a session will free all BDE objects owned by that session so make sure you close all tables and databases owned by the session before closing it otherwise you'll be asking for trouble.

Armed with these function calls we can open new sessions, switch between them and close them – everything we need.

But before continuing, it's important to clarify the terminology used when talking about opening and closing a session. In IDAPI-speak, opening a session actually creates a new session every time, closing a session will free that session and all objects owned by it (databases and hence tables). If you fail to close all TDatabases and TTables before closing a session created using DbiOpenSession, you'll end up with a data-aware component whose IDAPI handle has been freed, thus effectively disconnecting the component from its underlying IDAPI object, be it a database handle or table cursor handle. Needless to say, you'll encounter lots of problems if this occurs, so be careful.

So, after we've opened a new session, how can we associate a TDatabase or TTable with it? Very simple really, by creating and opening a TDatabase or TTable with the new session active, that TDatabase will be owned by the new session. Any other TDatabases or TTables created and opened with the new session active will also belong to that session. We can then switch to the default session knowing that any TDatabases and TTables associated with our new session be isolated from any TDatabase or TTable created and opened during the default session. I've set up a simple little Delphi demo application called SESSDEM3 to try out what I've outlined here (the code is on the disk of course).

Listing 1 shows the all important code snippet which creates the TDatabase at run-time that's owned by the new session.

➤ *Listing 1*

```
procedure TForm1.FormCreate(Sender: TObject);
var NetFileDir: array[0..255] of char;
begin
  Check(DbiStartSession(nil, HSession, StrPCopy(NetFileDir,
Session.NetFileDir)));
  ADatabase := TDatabase.Create(Self); {now owned by HSess}
  ADatabase.AliasName    := 'DBDEMOS';
  ADatabase.Databasename := 'AltSess';
  ADatabase.Open;
  ATable.Open;
  Check(DbiSetCurrSession(nil)); {make the default session active}
end;
```

I should mention the Check procedure which is defined in the DB.PAS unit and simply raises a database engine exception if the return code of the BDE function passed as a parameter indicates a failure. Borland use it in the VCL so it's good enough for me. The HSession variable is of type HDBISes which is defined in the unit DbiTypes which you'll need to include in the uses statement of the form's unit. You'll also need to use the DbiProcs unit which contains definitions for the various Dbi... functions used in the form's unit. Try out the application for yourself by opening the new session/database, edit any record in the lower grid and then try to delete the same record from the upper grid. An exception is raised and the delete fails. Back in the Delphi IDE, edit the FormCreate handler and comment out the call to DbiStartSession, run the application again and repeat the edit and delete as before. The locked record gets deleted.

So now we can create and open a TDatabase that's owned by a session other than the default session and therefore any tables opened from that TDatabase will have their locks isolated by the non-default session. If we wanted to add another TDatabase that is owned by the non-default session, we'd just call DbiSetCurrSession(HSess), create and open the TDatabase and then call DbiSetCurSession(nil) to switch back the default session.

However, if you have a look at the TDatabase.Open method in DB.PAS, you'll notice that the lifetime of the database IDAPI object (encapsulated in a TDatabase) starts when the TDatabase is opened and ends when the TDatabase is closed. Therefore we're only required to open a TDatabase or TTable in the context of a certain session in order for that session to own the database or table and so the required code reduces to that shown in Listing 2. Also, we're not forced to create the TDatabase at run-time. Don't forget to close any new sessions (by using DbiCloseSession(HSession)) when the form is destroyed.

How about creating a descendant of TDatabase to implement multiple sessions? We could do this, but it would mean that each instance of this descendant would have its own individual private session, although if the sole purpose of using multiple sessions is to isolate table and record locks from the default session then there's no problem. Our TDatabase descendant, called TDatabaseEx, will need to override the Open and Close methods, as is shown in Listing 3.

This is all well and good but in fact we can't subclass TDatabase in the way we want simply because its Open and Close methods are static and so can't be overridden. You might think it's a simple matter of just making TDatabase.Open and TDatabase.Close virtual in \DELPHI\SOURCE\VCL\DB.PAS, rebuilding and copying the unit to \DELPHI\LIB. But when you come to rebuild COMPLIB.DCL, Delphi will complain that a particular unit is out of date, which basically means that some units in \DELPHI\LIB which have not been included in \DELPHI\SOURCE\VCL depend on DB.DCU and the source code for those units is not supplied. So we're stuck.

Summing Up

You may wonder why I've presented a solution that cannot work and of course can never have been tested! Well if Borland ever decide to issue an updated version of the VCL source and \DELPHI\LIB files where TDatabase.Open and TDatabase.Close are declared as virtual then the solution presented will be tested and should work. However, all isn't lost because we can still fall back on the first method (in SESSDEM3) of implementing multiple sessions in Delphi 1.0 which is quite adequate.

In my next article I'll be looking at calling the BDE and how to use it to extend the capabilities of Delphi's data-access.

John O'Connell is a freelance software consultant/developer specialising in Delphi and database application development. He can be reached via email on 73064.74@compuserve.com

➤ *Listing 2*

```
procedure TForm1.FormCreate(Sender: TObject);
var
  NetFileDir: array[0..255] of char;
begin
  Check(DbiStartSession(
    nil, HSession, StrPCopy(NetFileDir, Session.NetFileDir)));
  ADatabase.Open;
  Check(DbiSetCurrSession(nil));
  ATable.DatabaseName := 'AltSession';
  ATable.Open;
end;
```

➤ *Listing 3*

```
procedure TDatabaseEx.Open;
var
  NetFileDir: array[0..255] of char;
begin
  if Handle = nil then begin
    Check(DbiSetCurrSession(FSession));
    inherited Open;
    Check(DbiSetCurrSession(nil));
  end;
end;
procedure TDatabaseEx.Close;
begin
  if Handle <> nil then begin
    Check(DbiSetCurrSession(FSession));
    inherited Close;
  end
end;
```