Surviving Client/Server:
OLE Automation With SQL Server

by Steve Troxell

Last month we looked at MIDAS and Delphi 3's DCOM-based support for multi-tiered database development. Dave Jewell has been enlightening us about COM and OLE in general for several months. Since we are in a COM/OLE frame of mind, I thought we'd look at an OLE interface for a client/server RDBMS. Microsoft SQL Server ships with a library of OLE Automation objects collectively referred to as Distributed Management Objects (DMO). With these objects you can access any aspect of the SQL Server environment you care to mention. You can start or stop the SQL Server service, manage user logins and permissions, backup and restore databases, execute SQL queries, and a number of other capabilities.

OK, so you can do most of that anyway by issuing SQL statements. What's the advantage of using DMO? First, we can bypass the Borland Database Engine and eliminate the need to install and configure the BDE. Of course, by doing so we lose all those nifty data access components and data-aware controls. But we saw in Issues 25 and 26 how we can make custom `TDataset` descendants which can access the data through any API.

Second, the BDE by necessity must 'homogenize' its capabilities in order to perform equally well across a number of database platforms. By going straight to the proprietary database API, we can unlock a few features of the server that previously were difficult or impossible to take advantage of.

Third, the object-oriented architecture of DMO encapsulates many routine functions we might otherwise perform with SQL queries. Some aspects of development may be easier to work with in an object-oriented design rather than with unstructured SQL.

The fourth advantage is that it's neat!

The classes in the DMO architecture are arranged in a hierarchy as shown in Figure 1. The major classes are listed across the top with the ancillary classes listed beneath. For example, the `SQLServer` object contains collections of objects for manipulating the server configuration, user logins, the server registry, etc. The DMO classes generally map directly to an entity within the SQL Server environment. The application connects to one or more `SQLServer` objects, each with one or more database objects, each with one or more table objects, and so on.

➤ *Figure 1*

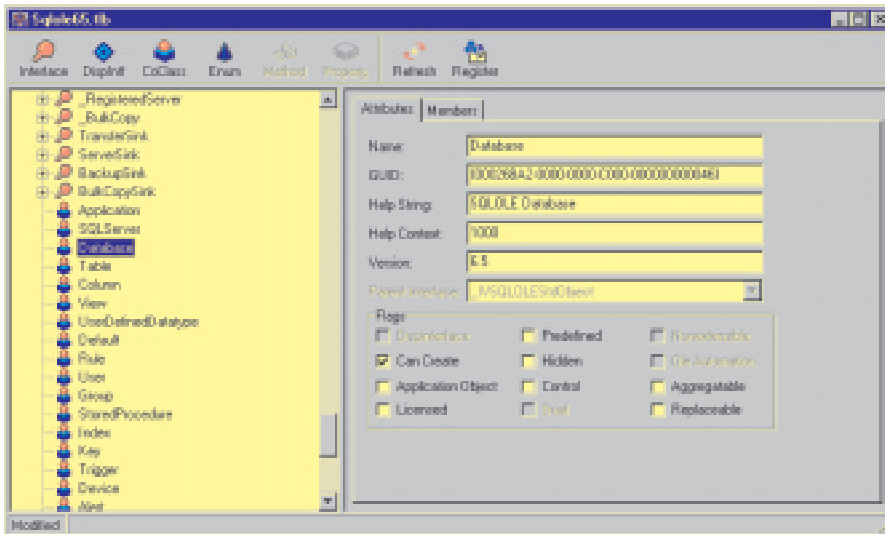
Application	SQLServer	Database	Table
Backup	Device	DBObject	Column
Permission	Login	StoredProcedure	Index QueryResults
Language	Rule	Trigger	
HistoryFiler	RemoteServer	Default	Key
Names	Configuration	User	Check
Backup	Executive	DBObject	Column
Property	Registry	Group	
	IntegratedSecurity	DBOption	
	Alert	TransactionLog	
	Operator	SystemDatatype	
		UserDefinedDatatype	
		Publication	

Getting Started With DMO

Once we get past the scary terminology like 'COM,' 'interfaces,' 'class factories' and their ilk, we realize that developing with a well-designed OLE Automation library is no more daunting than working with any other object-oriented architecture. We just have to keep a few more housekeeping chores in mind.

To build an application that uses DMO, we start with the DMO type library, which is a file called SQLOLE65.TLB that ships with SQL Server. If we copy this file into our project directory and open it in Delphi, we get the Type Library Editor as shown in Figure 2. Clicking the Refresh button will produce a Delphi interface file called SQLOLE_TLB.PAS. This gives us Delphi class definitions for all the DMO objects. Because some of the identifiers in DMO conflict with Delphi identifiers, several of them are renamed when the interface file is generated. A list of all changes is shown in the Type Library Editor and also stored as comments in the interface file. You may have to go into the Type Library Editor and manually change the name of DMO's `Application` class to avoid a conflict with Delphi's `Application` variable in the project file.

By including the `SQLOLE_TLB` unit in the uses clause, we can reference all the classes and interfaces defined in the type library. In most respects the DMO classes behave much as you've come to expect from working with the Delphi VCL.



➤ *Figure 2*

There are some key differences worth noting before we dive in.

Indexed properties of a class are normally used to access a collection of objects referenced by an index number. In the VCL, for example, the `TQuery.Fields` property accesses the collection of `TField` objects of the query result set. The `TStringList.Strings` property accesses the individual strings in the list (even though `String` is not strictly speaking an object).

DMO breaks up this type of reference into two categories, indexed properties and collections. Indexed properties always return a scalar datatype, while collections are used to reference a set of related objects. The significant difference between the two is merely the syntax used: square brackets versus parentheses. The code below shows how we reference the `ColumnName` property to return a simple string, and how we access a database object from the server's `Databases` collection:

```
Name := Results.ColumnName[3];
CurrentDatabase :=
  Server.Databases.Item(1);
```

It's important to note that the range of indexes in DMO runs from 1 to N rather than 0 to N - 1 as is the convention within Delphi's VCL.

Getting Our Feet Wet

Before we can really do anything with DMO, we have to connect to a server. For this we instantiate a `SQLServer` class. Figure 3 shows some of the things we can do with a `SQLServer`. For our first simple venture into DMO, we'll connect to a server and get a list of the databases available on the server. From Figure 3, it looks like iterating through the `Databases` collection will get us what we want. Looking at Figure 4 we see that, sure enough, the `Database` object will reveal its name. Listing 1 shows just how this is done.

Getting at the tables within the database is just as easy. Usually when we work with a database we will refer to it by name, so we have to search the server's list of databases by name until we find the `Database` object we want. Then we examine its `Tables` collection. Figure 5 shows us what we can expect from a `Table`.

Listing 2 shows how we can get a list of non-system tables in a database, assuming we have a connected `SQLServer` object to work with. Conveniently, the `Table` object includes a `boolean` property called `SystemObject`, which tells us whether or not the table is a system table.

➤ *Left: Figure 3, Right : Figure 4*

SQLServer Members (Abridged List)	
BeginTransaction	Start a transaction
CommitTransaction	Commit a transaction
Connect	Log into a server
Databases	Collection of Database objects
Disconnect	Log out of the server
ExecuteImmediate	Execute an SQL query
ExecuteWithResults	Execute an SQL query returning a result set
HostName	Name of the client computer
KillDatabase	Drop a database
KillProcess	Terminate a client connection
Logins	Collection of user logins
Name	Name of the SQL Server
RollbackTransaction	Rollback a transaction
SaveTransaction	Set a transaction savepoint
ShutDown	Shutdown the SQL Server service
Start	Startup the SQL Server service

Database Members (Abridged List)	
Checkpoint	Flushes the dirty pages cache
Defaults	Collection of Default objects
Dump	Performs a database backup
ExecuteImmediate	Execute an SQL query
ExecuteWithResults	Execute an SQL query returning a result set
Grant	Grant user permissions
Load	Restore a database from backup
Name	Name of the database
Parent	The SQLServer object over this database
Remove	Drop the database
Revoke	Revoke user permissions
Rules	Collection of Rule objects
StoredProcedures	Collection of StoredProcedure objects
SystemObject	Indicates a system database
Tables	Collection of Table objects
Views	Collection of View objects

```
procedure GetDatabaseNames(aList: TStrings);
var
  Server: SQLServer;
  I: Integer;
begin
  Server := CoSQLServer.Create;
  try
    with Server do begin
      Connect('MyServer', 'SteveT', 'abracadabra');
      aList.Clear;
      for I := 1 to Databases.Count do
        aList.Add(Databases.Item(I).Name);
      Disconnect;
    end;
  finally
    Server := nil;
  end;
end;
```

➤ *Listing 1*

```
procedure GetTableList(aServer: SQLServer; aDatabaseName: string;
  aList: TStrings);
var
  I: Integer;
  CurrentDatabase: Database;
begin
  { Search the server for the given database; no error checking if
    the databasename is invalid }
  with aServer do begin
    for I := 1 to Databases.Count do
      if CompareText(Databases.Item(I).Name, aDatabaseName) = 0 then begin
        CurrentDatabase := Databases.Item(I);
        Break;
      end;
  end;
  with CurrentDatabase.Tables do begin
    { the collection of Table objects }
    aList.Clear;
    for I := 1 to Count do
      if not Item(I).SystemObject then
        { omit system tables }
        aList.Add(Item(I).Name);
  end;
end;
```

➤ *Listing 2*

Table Members (Abridged List)	
Checks	Collection of Check objects (table-level check constraints)
Columns	Collection of Column objects (column definitions)
DataSpaceUsed	Disk space used by table rows
Indexes	Collection of Index objects
IndexSpaceUsed	Disk space used by table indexes
Keys	Collection of Key objects
Name	Name of the table
Parent	The Database object over this table
PrimaryKey	Primary key for the table
Refresh	Refreshes tables values from the server
Remove	Drops the table
Rows	Number of rows in the table
Script	Produces an SQL script to create the table
SystemObject	Indicates a system table
Triggers	Collection of Trigger objects
TruncateData	Deletes all rows in the table
UpdateStatistics	Updates the data distribution statistics for all indexes on the table

➤ *Figure 5*

OK, enough beating around the bush. How do we get at the actual data in the tables? All data manipulation must be done through SQL queries. Both the `SQLServer` and `Database` classes include methods to execute queries: `ExecuteImmediate` and `ExecuteWithResults`. `ExecuteImmediate` works just like `TQuery.ExecSQL` and is meant for queries that do not return a result set (`UPDATE`, `DELETE`, etc). `ExecuteWithResults` is like `TQuery.Open` and is meant for queries returning data (`SELECT`). In practice, `ExecuteWithResults` can be used for all queries since, unlike `TQuery`, no error will be generated if we do so, and we can easily detect the fact that no data has been returned.

The query to execute is passed into `ExecuteWithResults` as a string of characters. The query's results are handed back to us in the form of an instance of the `QueryResults` class. `QueryResults` is analogous to Delphi's `TDataSet` class and encapsulates a matrix of variable data (see Figure 6).

Listing 3 shows a code snippet that executes a query and displays the column names and data in a `TMemo` control called `memResults`. We pass in the database object of the database in which we want to run our query (which we get from `SQLServer.Databases`) and a string containing our query. It is then a simple matter to loop through the columns and rows and fetch the column names and cell data to display in a simple matrix.

We have to use the `GetColumnType` method to determine which method we need to retrieve a given cell's data. `GetColumnType` returns an integer that indicates the column's datatype. Fortunately, the type library defines a number of constants for us which we can use to symbolically represent the datatype. For example, `SQLOLE_DTypeInt4` means the column is a 32-bit integer datatype. We arbitrarily decided to display each column with a width of 20, but with a little ingenuity we could use `ColumnMaxLength` to dynamically change the width of each column.

In what seems to be the most glaring problem with the DMO

```
procedure GetQueryResults(aDatabase: Database;
  aQuery: string);
var
  Col, Row: Integer;
  S: string;
  Results: QueryResults;
begin
  Results := aDatabase.ExecuteWithResults(aQuery);
  with Results do begin
    { echo the column names }
    S := '';
    for Col := 1 to Columns do
      S := S + Format('%-20.20s ', [ColumnName[Col]]);
    memResults.Lines.Add(S);
    { echo the cell values }
    for Row := 1 to Rows do begin
      S := '';
      for Col := 1 to Columns do
        case ColumnType[Col] of
          SQLOLE_DTypeChar,
          SQLOLE_DTypeVarchar,
          SQLOLE_DTypeText,
          SQLOLE_DTypeDateTime,
          SQLOLE_DTypeDateTime4 :
            S := S + Format('%-20.20s ',
                 [GetColumnString(Row, Col)]);
          SQLOLE_DTypeInt1,
          SQLOLE_DTypeInt2,
          SQLOLE_DTypeInt4 :
            S := S + Format('%-20d ',
                 [GetColumnLong(Row, Col)]);
          SQLOLE_DTypeFloat4,
          SQLOLE_DTypeMoney4 :
            S := S + Format('%-20g ',
                 [GetColumnFloat(Row, Col)]);
          SQLOLE_DTypeFloat8,
          SQLOLE_DTypeMoney :
            S := S + Format('%-20g ',
                 [GetColumnDouble(Row, Col)]);
          SQLOLE_DTypeImage :
            S := S + Format('%-20.20s ', ['(image)']);
          SQLOLE_DTypeVarBinary,
          SQLOLE_DTypeBinary :
            S := S + Format('%-20.20s ', ['(binary)']);
          SQLOLE_DTypeBit :
            S := S + Format('%-20d ',
                 [Ord(GetColumnBool(Row, Col))]);
        else
          S := S + Format('%-20.20s ', ['(xxxxxx)']);
        end;
      memResults.Lines.Add(S);
    end;
  end;
end;
```

➤ *Listing 3*

design, there is no reliable way to detect the value 'null' within a result set column.

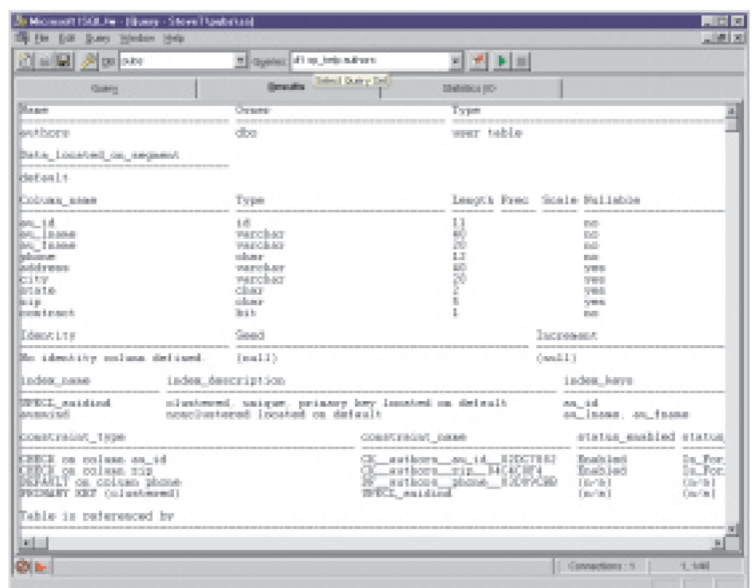
Queries Returning Multiple Result Sets

With SQL Server, it is possible for a query script or stored procedure to return more than one result set. While such a thing does not happen often in ad hoc queries, there are a number of built-in system stored procedures that do this. For example, Figure 7 shows the output from the `sp_help` system stored procedure, which is being used to display information about a table (`Authors`) in the database. This procedure uses several different `SELECT` statements to return information about the table. This means several different result sets are returned from one invocation of this procedure. Looking at Figure 7, the first result set contains one row showing the table name, owner, and type. The third result set contains one row for each column in the table. The fifth result set contains one row for each index on the table. And so on.

Delphi's `TDataSet` is not designed to deal with a single query execution that returns multiple result sets with differing column configurations. At best you'll only see the contents of the first result set. At worst, you'll get a BDE error. In general, this isn't a significant limitation of `TDataSet`. The need to create queries like this will only

QueryResults Members (Abridged List)	
Columns	Number of columns in the current result set
ColumnMaxLength	Returns the maximum length of data for a given column
ColumnName	Returns the name of a given column
ColumnType	Returns the datatype of a given column
CurrentResultSet	The current result set [1..ResultSets]
GetColumnBinary	Returns a cell's data as an array of Bytes
GetColumnBool	Returns a cell's data as a WordBool
GetColumnDate	Returns a cell's data as a TDateTime
GetColumnDouble	Returns a cell's data as a Double
GetColumnFloat	Returns a cell's data as a Single
GetColumnLong	Returns a cell's data as a LongInt
GetColumnString	Returns a cell's data as a WideString
Refresh	Reexecutes the query to return current values from the server
ResultSets	Number of result sets
Rows	Number of rows in the current result set

➤ *Figure 6*



➤ *Figure 7*

come up rarely, and they can always be broken into separate queries. It also seems clear that system procedures like sp_help are meant to be executed in an interactive environment like ISQL and their output read by human eyes, rather than processed by program logic.

However, it may be helpful to be able to run procedures like sp_help and capture their output within a program. For example, one tool we have here at Ultimate Software Group handles all our table structure definitions and rebuilds tables as necessary when we make changes. Each time we make a build of the project, any number of tables might be restructured as well. As an aid to our quality assurance process, it would be helpful to have an audit report showing the structure before and after the automated rebuild of each table. The output of sp_help is perfect for this job, so it would be handy to call it from within the rebuild tool and save its output to a text file for review. Unfortunately, because it returns more than one result set, we can't get the full output from sp_help using TQuery or TStoredProc. We would have to write our own series of queries to gather the same information.

The QueryResult class in DMO is specifically designed for more than one result set. The property ResultSets give us the number of different result sets returned by the query. Actually, if we've run a query that does not return any results, such as an UPDATE statement, then no result sets are returned and the property ResultSets equals zero. This should not be confused with a query that returns a result set with no rows, such as a SELECT statement with a WHERE clause which doesn't happen to match any rows. In this case one result set is returned (ResultSets = 1) with no rows (Rows = 0).

We set the property CurrentResultSet to switch between the result sets returned by the query. When CurrentResultSet = 1, then we are operating with the first result set. All the query properties and methods are relative to the current result set. For example, Rows gives us the number of rows in the current result set, not all result sets combined. To retrieve the full output of any query, we simply loop through the result sets by setting CurrentResultSet to values between 1 and ResultSets inclusive.

Conclusion

Microsoft SQL Server's Distributed Management Object library provides us with an OLE Automation interface to a SQL Server database backend. We can use this API to access the server's databases simply by referencing object properties and methods. It provides a well designed object-oriented alternative to tasks normally done through unstructured SQL.

Steve Troxell is a software engineer with Ultimate Software Group in the USA. He can be contacted via email at Steve_Troxell@USGroup.com