

# Database Development with dbExpress

CHAPTER

**8**

## IN THIS CHAPTER

- Using dbExpress 350
- dbExpress Components 351
- Designing Editable dbExpress Applications 359
- Deploying dbExpress Applications 360

dbExpress is Borland's new technology that provides lightweight database development to Delphi 6 developers.

dbExpress is important for three reasons. First, it is much lighter from a deployment standpoint than its predecessor, the BDE. Second, it is the cross-platform technology that you should use if developing applications intended for the Linux platform using Kylix. Third, it is extensible. To develop dbExpress drivers, one simply implements the required interfaces and provides the resulting database access library.

dbExpress's underlying architecture consists of drivers for supported databases, each of which implement a set of interfaces enabling access to server specific data. These drivers interact with applications through DataCLX connection components in much the same way a TDatabase component interacts with the BDE—minus the extra overhead.

## Using dbExpress

dbExpress is designed to efficiently access data and to carry little overhead. To accomplish this, dbExpress uses *unidirectional* datasets.

### Unidirectional, Read-Only Datasets

The nature of unidirectional datasets means that they don't buffer records for navigation or modification. This is where the efficiency is gained against the bi-directional BDE datasets that do buffer data in memory. Some limitations that result are

- Unidirectional datasets only support the `First()` and `Next()` navigational methods. Attempts to call other methods—such as `Last()` or `Prior()`—will result in an exception.
- Unidirectional dataset records aren't editable because there is no buffer support for editing. Note, however, that you would use other components (`TClientDataset`, `TSQLClientDataset`) for editing, which we'll discuss later.
- Unidirectional datasets don't support filtering because this is a multirecord feature and unidirectional datasets don't buffer multiple records.
- Unidirectional datasets don't support lookup fields.

### dbExpress Versus the Borland Database Engine (BDE)

dbExpress offers several advantages over the BDE, which we'll briefly go over.

Unlike the BDE, dbExpress doesn't consume server resources with metadata queries or other extraneous requests when user-defined queries are executed against the database server.

dbExpress doesn't consume as many client resources as the BDE. Because of the unidirectional cursor, no caching is done. dbExpress doesn't cache metadata on the client either. Metadata definition is handled through the data-access interface DLLs.

Unlike the BDE, dbExpress doesn't generate internal queries for things like navigation and BLOB retrieval. This makes dbExpress much more efficient at runtime in that only those queries specified by the user are executed against the database server. dbExpress is far simpler than the BDE.

## dbExpress for Cross-Platform Development

A key advantage to dbExpress is that it is cross-platform between Windows (using Delphi 6) and Linux (using Kylix). By using the CLX components for dbExpress, you can compile your application with Kylix and have the same application running in Linux. In fact, dbExpress can use a cross-platform database such as MySQL or InterBase.

### NOTE

At the time of this writing, support for the latest version of MySQL was limited to an earlier version (3.22). However, Delphi 6 can work with the latest version of the database (3.23) by using the shipping version of the dbExpress DLL. Borland is working on an update of the library.

## dbExpress Components

All the dbExpress components appear on the dbExpress tab of the Component Palette.

### TSQLConnection

For those who have done BDE development, the TSQLConnection will appear very similar to the TDatabase component. In fact, the purpose is the same in that they both encapsulate the database connection. It is through the TSQLConnection that dbExpress datasets access server data.

TSQLConnection relies on two configuration files, `dbxdrivers.ini` and `dbxconnections.ini`. These files are installed to the “\Program Files\Common Files\Borland Shared\DbExpress” directory. `dbxdrivers.ini` contains a listing of all dbExpress supported drivers and driver specific settings. `dbxconnections.ini` contains a listing of “*named connections*”—which can be considered similar in nature to a BDE alias—and any specific settings for these connections. It is possible not to use the default `dbxconnections.ini` file at runtime by setting the `TSQLConnection.LoadParamsOnConnect` property to true. We'll show an example of doing this momentarily.

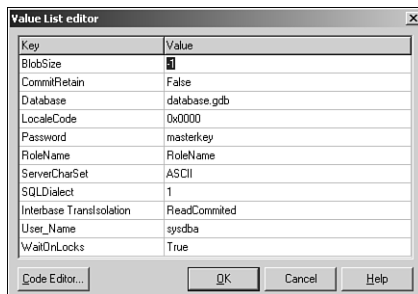
A `TSQLConnection` component must use a `dbExpress` driver specific to the type of database that you are using. This driver is specified in the `dbxdrivers.ini` file.

The `TSQLConnection`'s methods and properties are adequately covered in the online help. As always, we direct you to the online help for detailed information. In this book, we will walk you through establishing a database connection and in creating a new connection.

## Establishing a Database Connection

To establish a connection with an existing database, simply drop a `TSQLConnection` on a form and specify a `ConnectionName` by selecting one from the drop-down list in the Object Inspector. When doing so, you should see at least four different connections: `IBLocal`, `DB2Connection`, `MSCConnection`, and `Oracle`. If you didn't install a version of `InterBase` when you installed Delphi, do so now. You'll need one for this example. Once you have one installed, select the `IBLocal` connection because `Local InterBase` should have been installed with your Delphi 6 installation.

Upon selecting a `ConnectionName`, you'll see that other properties such as `DriverName`, `GetDriverFunc`, `LibraryName`, and `VendorLib` are automatically filled in. These default values are specified in the `dbxdrivers.ini` file. You can examine and modify other driver specific properties from the `Params` property's editor, shown in Figure 8.1.



**FIGURE 8.1**

*TSQLConnection.Params property editor.*

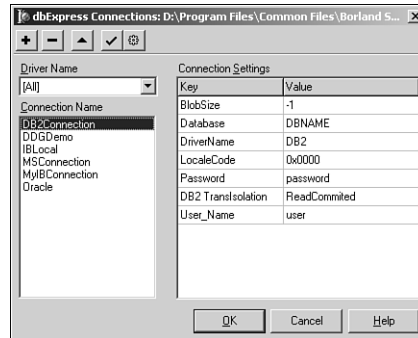
### NOTE

The default value in the "Database" key in the Params property editor is simply "database.gdb". This refers to a nonexistent database. You can change this value to the "Employee.gdb" example database that should exist in a subdirectory of your `InterBase` installation. On our machine, this is "...\\Program Files\\Borland\\InterBase6\\examples\\Database\\Employee.gdb".

Once you have the `TSQLConnection` component referring to a valid database, you can change the `Connected` property value to `True`. You'll be prompted for a username and password, which are **"sysdba"** and **"masterkey"**, respectively. This should connect you to the database. It would be a good idea to refer to the help files for each of the `TSQLConnection` properties at this point.

## Creating a New Database Connection

You can create additional "named" connections that refer to databases that you specify. For instance, this would be helpful if you were creating an application that used two separate databases such as a live and a test database. To create a new connection, simply double-click on the `TSQLConnection` component to bring up the Connection Editor (see Figure 8.2). You can also right-click and select "Edit Connection Properties" from the `TSQLConnection` local menu to invoke this editor.



**FIGURE 8.2**

*The TSQLConnection Connection Editor.*

You'll see that there are five speed buttons on this editor. We'll examine the "Add" button now. When pressed, you are asked to provide a Driver Name and a Connection Name. The Driver Name drop-down will be one of the four supported database drivers. You can select InterBase in this example. You can specify any name for the Connection Name such as **"MyIBConnection"**. When you select "OK", you'll see the Connection Settings grid display the driver settings for your specific connection. These are the same as the `TSQLConnection.Params` property values. Again, you'll need to change the "Database" setting to a valid InterBase database. At this point, you should be able to close the editor and set the `Connected` property to `True` by specifying the proper username and password.

## Bypassing/Replacing the Login Prompt

Bypassing the login prompt is easy. Simply set the `LoginPrompt` property to `False`. You'll have to make sure that the `UserName` and `Password` settings in the `Params` property have a valid user name and password, respectively.

To replace the login prompt with your own login dialog, the `LoginPrompt` property must be set to `True`. Then, you must add an event handler to the `OnLogin` event. For instance, the following code illustrates how this might look:

```
procedure TMainForm.SQLConnection1Login(Database: TSQLConnection;
    LoginParams: TStrings);
var
    UserName: String;
    Password: String;
begin
    if InputQuery('Get UserName', 'Enter UserName', UserName) then
    if InputQuery('Get Password', 'Enter Password', Password) then
    begin
        LoginParams.Values['UserName'] := UserName;
        LoginParams.Values['Password'] := Password;
    end;
end;
```

In this example, we're using a call to the `InputQuery()` function to retrieve the values needed. You would be able to use your own dialog for the same purpose. You'll find this example on the CD that also demonstrates the use of the `AfterConnect` and `AfterDisconnect` events.

## Loading Connection Settings at Runtime

The connection settings that you see from the Connection Editor or the Params property editor are defaults that get loaded at design time from the `dbxconnections.ini` file. It is possible for you to load these at runtime. You might do this, for example, if you needed to provide a separate `dbxconnections.ini` file than that provided with Delphi. Of course, you must remember to deploy this new file with your application installation.

To enable your application to load these settings at runtime, you must set the `LoadParamsOnConnect` property to `True`. When your application launches, the `TSQLConnection` component will look to the registry for the "Connection Registry File" key in "HKEY\_CURRENT\_USER\Software\Borland\DBExpress". You must modify this value to point to the location of your own `dbxconnections.ini` file. This is something that you would probably do in the installation of your application.

## TSQLDataset

`TSQLDataset` is the unidirectional dataset used for retrieving data from a `dbExpress` supported server. This dataset can be used to represent data in a database table, a selection query, or the results of a stored procedure. It can also execute a stored procedure.

TSQDataset's key properties are `CommandType` and `CommandText`. The value selected for `CommandType` determines how the content of `CommandText` will be used. Possible values for `CommandType` are listed in Table 8.1 and in the Delphi help file.

**TABLE 8.1** `CommandType` Values (from Delphi Online Help)

<code>CommandType</code>	<i>Corresponding</i> <code>CommandText</code>
<code>ctQuery</code>	An SQL statement that the dataset executes.
<code>ctStoredProc</code>	The name of a stored procedure.
<code>ctTable</code>	The name of a table on the database server. The SQL dataset automatically generates a <code>SELECT</code> statement to fetch all the records of all the fields in this table.

When the `CommandType` property contains the `ctQuery` value, `CommandText` is an SQL statement. This statement might be a `SELECT` statement that returns a resultset such as the following SQL statement: `"SELECT * FROM CUSTOMER"`.

If `CommandType` is `ctTable`, `CommandText` refers to a table name on the database server. The `CommandText` property will change to a drop down. If this is an SQL database, any SQL statements needed to retrieve data are automatically generated.

If `CommandType` has the value `ctStoredProc`, `CommandText` will then contain the name of a stored procedure to execute. This would be executed by calling the `TSQDataSet.ExecSQL()` method rather than by setting the `Active` property to `True`. Note, that `ExecSQL()` should be used if `CommandType` is `ctQuery` and the SQL statement doesn't result in a resultset.

## Retrieving Table Data

To extract table data using the `TSQDataset`, you simply set the `TSQDataSet.CommandType` property to `ctTable`. The `CommandText` property will change to a drop down from which you can select the table name. You can look at an example on the CD in the "TableData" directory.

## Displaying Query Results

To extract data from a query select statement, simply set the `TSQDataSet.CommandType` property to `ctQuery`. In the `CommandText` property, you can enter a query select statement such as `"Select * from Country"`. This is demonstrated in the example on the CD under the "QueryData" directory.

## Displaying Stored Procedure Results

Given a stored procedure that returns a resultset such as the InterBase procedure that follows, you can extract the resultset using a TSQLDataset component:

```
CREATE PROCEDURE SELECT_COUNTRIES RETURNS (
    RCOUNTRY VARCHAR(15),
    RCURRENCY VARCHAR(10)
) AS
BEGIN
    FOR SELECT
        COUNTRY, CURRENCY FROM COUNTRY
    INTO
        :rCOUNTRY, :rCURRENCY
    DO
        SUSPEND;
END
```

To do this, set the TSQLDataset.CommandType property to ctQuery and add the following to its CommandText property: `Select * from SELECT_COUNTRIES`. Note that we use the stored procedure name as though it were a table.

## Executing a Stored Procedure

Using the TSQLDataset component, you can execute a stored procedure that does not return a resultset. To do this, set the TSQLDataSet.CommandType property to ctStoredProc. The TSQLDataset.CommandText property will become a drop down that displays a list of stored procedures on the database. You must select one of the stored procedures that doesn't return a resultset. For example, the example on the CD under the directory "ExecSProc" executes the following stored procedure:

```
CREATE PROCEDURE ADD_COUNTRY (
    ICOUNTRY VARCHAR(15),
    ICURRENCY VARCHAR(10)
) AS
BEGIN
    INSERT INTO COUNTRY(COUNTRY, CURRENCY)
    VALUES (:iCOUNTRY, :iCURRENCY);
    SUSPEND;
END
```

This procedure is a simple insert statement into the country table. To execute the procedure, you must call the TSQLDataset.ExecSQL() method as shown in the following code:

```
procedure TForm1.btnAddCurrencyClick(Sender: TObject);
begin
    sqlDSAddCountry.ParamByName('ICountry').AsString := edtCountry.Text;
```



```

sqlDSAddCountry.ParamByName('ICURRENCY').AsString := edtCurrency.Text;
sqlDSAddCountry.ExecSQL(False);
end;

```

The first thing you must do is to set the parameter values. Then, by calling `ExecSQL()`, the specified procedure will be executed with the values you've added. Note that `ExecSQL()` takes a Boolean parameter. This parameter is used to determine whether any parameters need to be prepared. By default, this parameter should be true.

## Metadata Representation

You can retrieve information about a database using the `TSQLDataset` component. To do this, you use the `TSQLDataset.SetSchemaInfo()` procedure to specify the type of schema information you desire. `SetSchemaInfo` is defined as

```

procedure SetSchemaInfo( SchemaType: TSchemaType;
  SchemaObjectName, SchemaPattern: string );

```

The `SchemaType` parameter specifies the type of schema information that you are requesting. `SchemaObjectName` holds the name of a table or procedure in the case of a request for parameter, column, or index information. `SchemaPattern` is an SQL pattern mask used for filtering the resultset.

Table 8.2 is taken from the Delphi online help for the `SetSchemaInfo()` procedure and describes the types of schema information that you can retrieve.

**TABLE 8.2** SchemaType Values (from Delphi Online Help)

SchemaType Value	Description
stNoSchema	No schema information. The SQL dataset is populated with the results of its query or stored procedure rather than metadata from the server.
stables	Information about all the data tables on the database server that match the criteria specified by the SQL connection's <code>TableScope</code> property.
stSysTables	Information about all the system tables on the database server. Not all servers use system tables to store metadata. Requesting a list of system tables from a server that doesn't use them results in an empty dataset.
stProcedures	Information about all the stored procedures on the database server.
stColumns	Information about all the columns (fields) in a specified table.
stProcedureParams	Information about all the parameters of a specified stored procedure.
stIndexes	Information about all the indexes defined for a specified table.

We've provided an example of using the `SetSchemaInfo()` procedure on the CD under the directory "SchemaInfo". Listing 8.1 shows some of the code for this procedure from this example.

---

**LISTING 8.1** Example of `TSQLDataset.SetSchemaInfo()`

---

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
    sqldsSchemaInfo.Close;
    cdsSchemaInfo.Close;

    case RadioGroup1.ItemIndex of
    0: sqldsSchemaInfo.SetSchemaInfo(stSysTables, '', '');
    1: sqldsSchemaInfo.SetSchemaInfo(stTables, '', '');
    2: sqldsSchemaInfo.SetSchemaInfo(stProcedures, '', '');
    3: sqldsSchemaInfo.SetSchemaInfo(stColumns, 'COUNTRY', '');
    4: sqldsSchemaInfo.SetSchemaInfo(stProcedureParams, 'ADD_COUNTRY', '');
    5: sqldsSchemaInfo.SetSchemaInfo(stIndexes, 'COUNTRY', '');
    end; // case

    sqldsSchemaInfo.Open;
    cdsSchemaInfo.Open;
end;
```

---

In the example, we use the selection in `TRadioButton` component to determine which type of schema information we want. We then call the `SetSchemaInfo()` procedure using the proper `SchemaType` parameter before opening the dataset. The values are stored in a `TDBGrid` in the example.

## Backward Compatibility Components

You'll find three components on the `dbExpress` tab in the Component Palette that are synonymous with the BDE dataset components. These are `TSQLTable`, `TSQLQuery`, and `TSQLStoredProc`. These components are used very much in the same manner as their BDE counterparts except that they cannot be used in a bidirectional manner. For the most part, you will be using the `TSQLDataset` components.

## TSQLMonitor

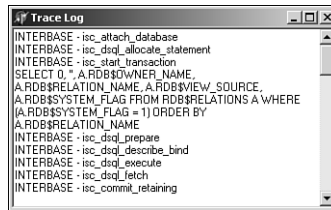
The `TSQLMonitor` component is useful for debugging SQL applications. `TSQLMonitor` logs the SQL commands being communicated through a `TSQLConnection` component. To use this, you simply set the `TSQLMonitor.SQLConnection` parameter to a valid `TSQLConnection` component.

The `TSQLMonitor.TraceList` property will then log the commands being passed between the client and the database server. `TraceList` is a simple `TStrings` descendant, so you can save this information to a file or add it to a memo component for viewing the information.

**NOTE**

You can use the `FileName` and `AutoSave` properties to automatically store the `TraceList` contents.

The example code provided on the CD in the `SQLMon` directory shows how to add the contents of the `TraceList` to a memo control. The resulting SQL tracelist is shown in Figure 8.3.



```
Trace Log
INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT 0 ,A.RDB$OWNER_NAME
,A.RDB$RELATION_NAME,A.RDB$VIEW_SOURCE,
A.RDB$SYSTEM_FLAG FROM RDB$RELATIONS A WHERE
(A.RDB$SYSTEM_FLAG = 1) ORDER BY
A.RDB$RELATION_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
```

**FIGURE 8.3**

Results of the `TSQLMonitor` component.

## Designing Editable dbExpress Applications

Up to now, we have discussed dbExpress in the context of unidirectional/read-only datasets. The only exception is the example using a `TSQLDataset` component to execute a stored procedure that adds data to a table. Another method to make datasets editable as with a bidirectional dataset is to use cached updates. To do so, this requires the use of another component, `TSQLClientDataset`.

### TSQLClientDataset

`TSQLClientDataset` is a component that contains an internal `TSQLDataset` and `TProvider` component. The internal `TSQLDataset` gives the `TSQLClientDataset` the fast data access benefits of dbExpress. The internal `TSQLProvider` gives the `TSQLClientDataset` the bidirectional navigation and ability to edit data.

Using the `TSQLClientDataset` is very much the same as using the standard `TClientDataset`. This information is covered in Chapter 21, “DataSnap Development.”

Setting up an application using `TSQLClientDataset` is relatively simple. You'll need a `TSQLConnection`, a `TSQLClientDataset`, and a `TDataSource` component if you intend to display the data. An example is provided on the CD under the directory "Editable".

The `TSQLClientDataset.DBConnection` property must be set to the `TSQLConnection` component. Use the `CommandType` and `CommandText` properties as previously discussed for the `TSQLDataset` component.

Now, when running this application, you will note that it is navigable in both directions and it is possible to add, edit, and delete records from the dataset. However, when you close the dataset, none of your changes will persist because you are actually editing the in-memory buffer held by the `TSQLClientDataset` component. Any changes you make are cached in memory. To save your changes to the database server, you must call the `TSQLClientDataset.ApplyUpdates()` method. In the sample provided on the CD, we've added the `ApplyUpdates()` call to the `AfterDelete` and `AfterPost` events of the `TSQLClientDataset` component. This gives us a row-by-row update of server data. For further information on using `TSQLClientDataset`, refer to Chapter 21, or Chapters 32 and 34 in *Delphi 5 Developer's Guide*, which is provided on the CD.

#### NOTE

The `TSQLClientDataset` contains a `TSQLDataSet` and `TProvider` component. However, it doesn't expose all the properties and events of these two components. If access to these events are needed, you can use the regular `TClientDataset` and `TDataSetProvider` components in lieu of the `TSQLClientDataset` component.

## Deploying dbExpress Applications

You can deploy dbExpress applications as a standalone executable or by providing the required dbExpress driver DLLs. To compile as a standalone, you'll need to add the units listed in Table 8.3 to the uses clause of your application as described in the Delphi online help.

**TABLE 8.3** Units Required for dbExpress Standalone Application

<i>Database unit</i>	<i>When to Include</i>
<code>dbExpInt</code>	Applications connecting to InterBase databases
<code>dbExpOra</code>	Applications connecting to Oracle databases
<code>dbExpDb2</code>	Applications connecting to DB2 databases
<code>dbExpMy</code>	Applications connecting to MySQL databases
<code>Crt1</code> , <code>MidasLib</code>	Required by dbExpress executables that use client datasets such as <code>TSQLClientDataSet</code>

If you want to deploy the DLLs along with your application, you will have to deploy the DLLs specified in Table 8.4.

**TABLE 8.4** DLLs to Deploy with a dbExpress Application

<i>Database DLL</i>	<i>When to Deploy</i>
dbexpint.dll	Applications connecting to InterBase databases
dbexpora.dll	Applications connecting to Oracle databases
dbexpdb2.dll	Applications connecting to DB2 databases
dbexpmy.dll	Applications connecting to MySQL databases
Midas.dll	Required by database applications that use client datasets

## Summary

With dbExpress, it will be possible to develop robust and lightweight applications not otherwise possible using the BDE. Combined with the caching mechanisms built into `TSQLClientDataset` and `TClientDataset`, developers can develop complete cross-platform database applications.

