# SqlDBHowto 编辑

---

## Databases portal

References:

- General info
- Libraries
- Field types
- Controls
- FAQ
- SQL how-to
- Working With TSQLQuery
- In-memory database applications

Tutorials/practical articles:

- Overview
- 0 - Database set-up
- 1 - Getting started
- 2 - Editing
- 3 - Queries
- 4 - Data modules
- SQLdb Programming Reference

Databases

> Advantage - MySQL - MSSQL - Postgres - Interbase - Firebird - Oracle - ODBC - Paradox - SQLite - dBASE - MS Access - Zeos

This text is setup as a 'how-to'. I want to answer a number of questions one by one, and explain how you can use the various classes. All those questions are put one after the other and form a sort of tutorial.

I will try to word it in such a way that the text can be used for Lazarus as well as Free Pascal. However, the examples are for FreePascal (i.e. they are console applications.)

## Where can I find official documentation?

Please see the official documentation at SQLDB documentation.

## How to connect to a database server?

SqlDB doesn't connect to a database server directly but uses a client that corresponds to the used database server. SqlDB sends the commands to the client library; the client library connects to the database and and transfers the commands. This means that a client library must be installed on the computer to make a connection to a database. Under Windows a client is usually a .dll, under Linux an .so and under OS/X a .dylib.

When the client library is installed properly you can connect to a database server using a TSQLConnection component. Various TSQLConnection components are available for different database servers (see SQLdb_Package):

- Firebird/Interbase: TIBConnection
- MS SQL Server: TMSSQLConnection (available since FPC 2.6.1)
- MySQL v4.0: TMySQL40Connection
- MySQL v4.1: TMySQL41Connection
- MySQL v5.0: TMySQL50Connection
- MySQL v5.1: TMySQL51Connection (available since FPC version 2.5.1
- MySQL v5.5: TMySQL55Connection (available since Lazarus 1.0.8/FPC version 2.6.2
- MySQL v5.6: TMySQL56Connection (available in Lazarus 1.2.4/FPC version 2.6.4
- ODBC: TODBCConnection (see ODBCConn#TODBCConnection)
- Oracle: TOracleConnection (see Oracle)
- PostgreSQL: TPQConnection (see postgresql#SQLDB)
- Sqlite3: TSQLite3Connection (available since FPC version 2.2.2, see SQLite#Built-in_SQLDB)
- Sybase ASE: TSybaseConnection (available since FPC 2.6.1, see Lazarus and MSSQL/Sybase)

Note for MySQL - There are many differences between the client versions to the extent that the clients and connections cannot be interchanged. If a MySQL client library version 4.1 is installed, you have to use a TMySQL41Connection. This is not related to the MySQL server; using the MySQL 4.1 client library you can probably connect to a MySQL 5.0 server (see MySQL documentation regarding what combinations are supported).

Although details differ for the various databases, in general you need to set four properties to connect to a database server:

- the server name or IP address
- the name of the database
- the username
- the password

When these properties are set, you can create a connection with the 'open' method. If the connection fails, a EDatabaseError exception is thrown. Use the property 'connected' to test if a connection has been made with the database server. Use the 'close' method to end the connection with the server.

```
Program ConnectDB;

{$mode objfpc}{$H+}

uses
  IBConnection;

function CreateConnection: TIBConnection;
begin
  result := TIBConnection.Create(nil);
```

```
  result.Hostname := 'localhost';
  result.DatabaseName := '/opt/firebird/examples/employee.fdb';
  result.UserName := 'sysdba';
  result.Password := 'masterkey';
end;

var
  AConnection : TIBConnection;

begin
  AConnection := CreateConnection;
  AConnection.Open;
  if Aconnection.Connected then
    writeln('Successful connect!')
  else
    writeln('This is not possible, because if the connection failed, ' +
            'an exception should be raised, so this code would not ' +
            'be executed');
  AConnection.Close;
  AConnection.Free;
end.
```

If an exception is thrown, read the error message carefully. It may be that the database server is not running, the user name or password are incorrect or the database name or IP address are typed incorrectly. If the error message states that the client library cannot be found, then check if the client is installed correctly. Often the error message states literally the name of the file looked for.

## How to execute direct queries/make a table?

SqlDB - the name says it all - only works with database server that make use of SQL. SQL stands for 'Structured Query Language' SQL is a language developed to allow working with relational databases. Virtually every database system has its own dialect, but a large number of SQL statements are the same for all database systems.

In FPC, there is a difference between:

- SQL statements that return information (a dataset). For this, you have to use the TSQLQuery component; see #How to read data from a table?.
- statements that do not return information but do something else, e.g. update data. For this, you may also use the 'ExecuteDirect' method of a TSQLConnection. (You can also use this if you get a dataset back but are not interested in the results, e.g. in a selectable stored procedure).

Most database system execute SQL statements within a transaction. If you want changes made within a transaction available in other transactions, or have those changes available even after closing the transaction(!), then you have to 'commit' the transaction.

To support transactions Sqldb contains the TSQLTransaction component. A SQL statement that is executed by Sqldb must always be executed within a transaction, even if the database system does not support transactions. Also, there are database systems that do support transaction for which TSQLConnection does not (yet) support transaction. Even then, you must use the TSQLTransaction component.

To use `TSQLConnection.ExecuteDirect` to execute a SQL statement you must specify which 'Transaction' must be used. In turn, to use TSQLTransaction you must specify which TSQLConnection component must be used.

The following example creates a table 'TBLNAMES' with fields 'NAME' and 'ID' and inserts two records. This time using SQLite. The used SQL statements are not explained. For more information about the SQL statements, their use and syntax, please refer to the database system documentation. Note that this example does not attempt to catch any errors, thats a bad thing! Look into Exceptions.

```
program CreateTable;
{$mode objfpc} {$ifdef mswindows}{$apptype console}{$endif}
uses
  sqldb, sqlite3conn;

var
  AConnection : TSQLite3Connection;
  ATransaction : TSQLTransaction;

begin
  AConnection := TSQLite3Connection.Create(nil);
  AConnection.DatabaseName := 'test_dbase';

  ATransaction := TSQLTransaction.Create(AConnection);
  AConnection.Transaction := ATransaction;
  AConnection.Open;
  ATransaction.StartTransaction;
  AConnection.ExecuteDirect('create table TBLNAMES (ID integer, NAME
varchar(40));');

  ATransaction.Commit;

  ATransaction.StartTransaction;
  AConnection.ExecuteDirect('insert into TBLNAMES (ID,NAME) values
(1,''Name1'');');
  AConnection.ExecuteDirect('insert into TBLNAMES (ID,NAME) values
(2,''Name2'');');
  ATransaction.Commit;
  AConnection.Close;
  ATransaction.Free;
```

```
  AConnection.Free;
end.
```

## How to read data from a table?

Use the TSQLQuery component to read data from a table. A TSQLQuery component must be connected to a TSQLConnection component and a TSQLTransaction component to do its work. Setting the TSQLConnection and TSQLTransaction is discussed in #How to connect to a database server? and #How to execute direct queries/make a table?.

When the TSQLConnection, TSQLTransaction and TSQLQuery are connected, then TSQLQuery needs to be further configured to work. TSQLQuery has a 'SQL' property containing a TStrings object. The 'SQL' property contains a SQL statement that must be executed. If all data from a table `tablename` must be read, then set the 'SQL' property to:

```
'SELECT * FROM tablename;'
```

.

Use 'open' to read the table from the server and put the data in the TSQLQuery dataset. The data can be accessed through TSQLQuery until the query is closed using 'close'.

TSQLQuery is a subclass of TDataset. TDataset has a 'Fields' collection that contains all columns of the table. The TDataset also keeps track of the current record. Use 'First', 'Next', 'Prior' and 'Last' to change the current record. 'Bof' returns 'True' if the first record is reached, and 'Eof' returns 'True' if the last record is reached. To read the value of a field in the current record, first find the right 'TField' object and then use 'AsString', 'AsInteger', etc.

### Example: reading data from a table

Below is an example that displays all values of the table as it was made in CreateTable example above. Copy the test_database file into the ShowData working directory. Note, again, no error checking has taken place !

```
Program ShowData;
 {$mode objfpc} {$ifdef mswindows}{$apptype console}{$endif}
uses
  DB, Sysutils, sqldb, sqlite3conn;

var
  AConnection  : TSQLConnection;
  ATransaction : TSQLTransaction;
  Query        : TSQLQuery;

begin
  AConnection := TSQLite3Connection.Create(nil);
  ATransaction := TSQLTransaction.Create(AConnection);
```

```
  AConnection.Transaction := ATransaction;
  AConnection.DatabaseName := 'test_dbase';
  Query := TSQLQuery.Create(nil);
  Query.SQL.Text := 'select * from tblNames';
  Query.Database := AConnection;
  Query.Open;
  while not Query.Eof do
  begin
    Writeln('ID: ', Query.FieldByName('ID').AsInteger, 'Name: ' +
                                  Query.FieldByName('Name').AsString);
    Query.Next;
  end;
  Query.Close;
  AConnection.Close;
  Query.Free;
  ATransaction.Free;
  AConnection.Free;
end.
```

(The code above of course is not quite finished, it misses 'try...finally' blocks. However, the above code intends to show the database code and thus the finishing touches are left out.) Please note that 'TSQLTransaction.StartTransaction' is not used. This is not necessary. When TSQLQuery is opened, the SQL statement is executed and if no transaction is available then a transaction is automatically started. The programmer does not need to start the transaction explicitly. The same applies for the connection maintained by TSQLConnection. The connection is opened as needed, the line 'Aconnection.Open' is not really required. If a TSQLTransaction is destroyed, an automatic 'rollback' will be executed. **Possible changes to data contained in the transaction will be lost.**

## Why does TSQLQuery.RecordCount always return 10?

To count the records in a dataset, use '.RecordCount'. However, notice that '.RecordCount' shows the number of records that is already loaded from the server. For performance reasons, SqlDB does not read all records when opening TSQLQuery by default, only the first 10. Only when the eleventh record is accessed will the next set of 10 records be loaded, etc. Using '.Last', all records will be loaded.

When you want to know the real number of records on the server you can first call '.Last' and then call '.RecordCount'.

An alternative is available. The number of records returned by the server is set by the '.PacketRecords' property. The default value is 10; if you make it -1 then all records will be loaded at once.

In current stable FPC, '.RecordCount' does not take filters into account, i.e. it shows the unfiltered total.

If you need the exact number of records, it often is a better idea to directly query the number of records in a query using another SQL query, but you would have to do that in the same transaction, as other transactions may have changed the number of records in the meanwhile.

### Lazarus

Lazarus has various components to show data from a TDataset on a form. Instead of a While-loop and Writeln statements as used above, you can use the components to show the data in a table. Place the right TSQLConnection, TSQLTransaction and TSQLQuery components on a form, then connect them and set them properly. In addition you will need a TDatasource; set the 'TDatasource.Dataset' property to the TSQLQuery component you used. (**Note** do not set the 'TSQLQuery.Datasource' property to the TDatasource compnent you used. The 'TSQLQuery.Datasource' property is used only in master-detail tables - see MasterDetail) Subsequently you may put a TDBGrid onto the form and set the 'Datasource' property of the grid to the TDatasource component you added before.

To see if it all works, set the 'Connected' property of the TSQLConnection to 'True' in the Lazarus IDE. The IDE will try to connect to the database server immediately. If this works you can set the 'TSQLQuery.Active' property to 'True'. If everything is right, you will see - within the IDE - all data from the table immediately on the screen.

## How to change data in a table?

To change the data in a record (or records), the general process is get TSQLQuery to search for the records you wish to change, make the changes there and then push them back to the database. The TDataSet (from which TSQLQuery is derived) must be set to edit mode. To enter edit mode call the '.Edit', '.Insert' or '.Append' methods. Use the '.Edit' method to change the current record. Use '.Insert' to insert a new record before the current record. Use '.Append' to insert a new record at the end of the table. In edit mode you can change field values through the 'Fields' property. Use 'Post' to validate the new data, if the data is valid then the edit mode is left. If you move to another record - for example by using '.Next' - and the dataset is in edit mode, then first '.Post' is called. Use '.Cancel' to discard all changes you made since the last '.Post' call and leave the edit mode.

```
Query.Edit;
Query.FieldByName('NAME').AsString := 'Edited name';
Query.Post;
```

The above is not the complete story yet. TSQLQuery is derived from TBufDataset which makes use of buffered updates. Buffered update means that after you called 'Post' the changes in the dataset are visible immediately, but they are not sent to the database server. What does happen is that the changes are maintained in a change log. When the '.ApplyUpdates' method is called, then all changes in the change log are sent to the database. Only then will database server know of the changes. The changes are sent to the server within a transaction of TSQLTransaction. Make sure to properly set the transaction before 'ApplyUpdates'. After applying the updates, a commit must be executed to save the changes on the database server.

The below is an example of changing the data in a table, sending the changes to the server and committing the transaction. Again, no error checking, again, thats bad!

```
Program EditData;
{$mode objfpc} {$ifdef mswindows}{$apptype console}{$endif}
uses
```

```
    db, sqldb, sqlite3conn;
var
  AConnection : TSQLConnection;
  ATransaction : TSQLTransaction;
  Query : TSQLQuery;

begin
  AConnection := TSQLite3Connection.Create(nil);
  ATransaction := TSQLTransaction.Create(AConnection);
  AConnection.Transaction := ATransaction;
  AConnection.DatabaseName := 'test_dbase';
  Query := TSQLQuery.Create(nil);
  Query.DataBase := AConnection;
  Query.SQL.Text := 'select * from tblNames where ID = 2';
  Query.Open;
  Query.Edit;
  Query.FieldByName('NAME').AsString := 'Name Number 2';
  Query.Post;
  Query.UpdateMode := upWhereAll;          // defined in db
  Query.ApplyUpdates;
  ATransaction.Commit;
  Query.Free;
  ATransaction.Free;
  AConnection.Free;
end.
```

The actual works starts with the SQL statement "select * from tblNames where ID = 2" identifying the record (or records) you wish to change. If you leave out the "where ID = 2" bit, the TSQLQuery apparently sets ID (and other integer fields?) to 1. And therefore will operate on lines where ID=1 only. For a discussion of 'UpdateMode' continue reading.

## How does SqlDB send the changes to the database server?

In the code example in #How to change data in a table?, you will find the line

```
Query.UpdateMode := upWhereAll;
```

without explanation of what it does. The best way to find out what that line does is to leave it out. If you leave out the statement and the followed this howto precisely, then you will receive the following error message:

```
No update query specified and failed to generate one. (No fields for
inclusion in where statement found)
```

To understand what went wrong, you must understand how changes are sent to the database server. The only way to get data in a SQL server is by executing SQL queries. SQL has three types of queries for

three different ways of manupulating a record. To create a new record, change or delete a record insert, update and delete statements are executed respectively. An update statement may be as follows:

```
update TBLNAMES set NAME='Edited name' where ID=1;
```

To send a change to the database server, Sqldb must assemble an update query. To assemble the query, three things are needed:

The name of the table
: The table name is retrieved from parsing the select query, although this doesn't always work.

`UPDATE` or `INSERT` clause
: These contain the fields that must be changed.

`WHERE` clause
: This contains the fields that determine which records should be changed.

Every field (each *TField* in *Fields*) has a ProviderFlags property. Only fields with **pfInUpdate** in *ProviderFlags* will be used in the update or insert cluase of a query. By default all fields have *pfInUpdate* set in their *ProviderFlags* property.

Which fields are used in the `WHERE` clause depends on the *UpdateMode* property of the query and the *ProviderFlags* property of the fields. Fields with *pfInkey* in their *ProviderFlags* are always used in the `WHERE` clause. A field will have the *pfInKey* flag set automatically if the field is part of the primary key of the table and 'TSQLQuery.UsePrimaryKeyAsKey' returns 'True'.

The default value for *UpdateMode* of the query is *upWhereKeyOnly*. In this update mode only fields with *pfInkey* in their *ProviderFlags* property are used in the `WHERE` clause. If none of the fields have their *pfInKey* flag set, then no fields are available for the `WHERE` clause and the error message from the beginning of this section will be returned. You can solve the issue by:

- Adding a primary key to the table and set *TSQLQuery.UsePrimaryKeyAsKey* to 'True', or
- Setting the *pfInkey* flag for one or more fields in code.

The **UpdateMode** property knows two more possible values. 'upWhereAll' can be used to add all fields with the 'pfInWhere' flag set to the `WHERE` clause. By default all fields have this flag set. 'upWhereChanged' can be used to add only those fields that have the 'pfInWhere' flag set **and** that are changed in the current record.

# How to handle Errors

Run time errors are unavoidable, disks may fill up, necessary libraries or helper apps may not be available, things go wrong and we need to allow for that. The FPC detects and handles run time errors quite well. It usually gives you a concise and reasonable explanation of what went wrong. However, you will want to monitor and handle errors yourself for a number of reasons -

- You probably don't want the programme to teminate at the first sign of trouble.
- If we do keep going, lets make sure any memory allocated in the problem area is recovered, we don't want any memory leaks.

- If we are going to go under however, lets give the user a context sensitive explanation.

The following bit of code is based on the above examples but this time it DOES check for errors in critical places. Key is the try...finally...end and try...except...end blocks. You can test it by doing things like uninstalling SQLite3, putting a dummy file in place of the test_dbase database and so on.

```
program DemoDBaseWithErrors;
{$mode objfpc} {$ifdef mswindows}{$apptype console}{$endif}
uses
  DB, Sysutils, sqldb, sqlite3conn;
var
    Connect : TSQLite3Connection;
    Trans : TSQLTransaction;


procedure WriteTable (Command : string);
begin
        Connect.ExecuteDirect(Command);
        Trans.Commit;
end;

procedure ReadTable ();
var
   Query : TSQLQuery;
   Count : smallint;
begin
    Count := 0;
    try
        Query := TSQLQuery.Create(nil);
        Query.DataBase := Connect;
        Query.SQL.Text:= 'select * from tblNames';
        Query.Open;          // This will also open Connect
        while not Query.EOF do begin
            writeln('ID: ', Query.FieldByName('ID').AsInteger, '  Name: ' +
                             Query.FieldByName('Name').AsString);
            Query.Next;
            Count := Count + 1;
        end;
    finally
        Query.Close;
        Query.Free;
    end;
    writeln('Found a total of ' + InttoStr(Count) + ' lines.');
end;
```

```
procedure FatalError(ClassName, Message, Suggestion : string);
begin
    writeln(ClassName);
    writeln(Message);
    writeln(Suggestion);
    Connect.Close;              // Its possibly silly worrying about freeing
    Trans.free;                 // if we are going to call halt() but its
    Connect.Free;               // a demo, alright ?
    halt();
end;

begin
    Connect := TSQLite3Connection.Create(nil);
    Trans := TSQLTransaction.Create(Connect);
        Connect.Transaction := Trans;
    Connect.DatabaseName := 'test_dbase';
    try
        if not fileexists(Connect.DatabaseName) then begin
            Connect.Open;   // give EInOutError if (eg) SQLite not installed
            Trans.StartTransaction;
            WriteTable('create table TBLNAMES (ID integer Primary Key, NAME
varchar(40));');
            Trans.Commit;
        end;
        Connect.open;
        Trans.StartTransaction;
        WriteTable('insert into TBLNAMES (NAME) values (''AName1'');');
        WriteTable('insert into TBLNAMES (NAME) values (''AName2'');');
    except
        on E : EDatabaseError do
            FatalError(E.ClassName, E.Message, 'Does the file contain the
correct database ?');
        on E : EInOutError do
            FatalError(E.ClassName, E.Message, 'Have you installed SQLite
(and dev package)?');
        on E : Exception do
            FatalError(E.ClassName, E.Message, 'Something really really bad
happened.');
     end;
    ReadTable();
    Connect.Close;
    Trans.Free;
    Connect.Free;
end.
```

## How to execute a query using TSQLQuery?

Next to statements that return a dataset (see #How to read data from a table?) SQL has statements that do not return data. For example `INSERT`, `UPDATE` and `DELETE` statements do not return data. These statements can be executed using *TSQLConnection.ExecuteDirect*, but TSQLQuery can also be used. If you do not expect return data use *TSQLQuery.ExecSQL* instead of *TSQLQuery.Open*. As mentioned earlier, use *TSQLQuery.Open* to open the dataset returned by the SQL statement.

The following procedure creates a table and inserts two records using TSQLQuery.

```
procedure CreateTable;

var
  Query : TSQLQuery;

begin
  Query := TSQLQuery.Create(nil);
  try
    Query.Database := AConnection;

    Query.SQL.Text := 'create table TBLNAMES (ID integer, NAME
varchar(40));';
    Query.ExecSQL;

    Query.SQL.Text := 'insert into TBLNAMES (ID,NAME) values (1,''Name1'');';
    Query.ExecSQL;

    Query.SQL.Text := 'insert into TBLNAMES (ID,NAME) values (2,''Name2'');';
    Query.ExecSQL;
  finally
    Query.Free;
  end;
end;
```

## How to use parameters in a query?

In the code example of #How to execute a query using TSQLQuery? the same query is used twice, only the values to be inserted differ. A better way to do this is by using parameters in the query.

The syntax of parameters in queries is different per database system, but the differences are handled by TSQLQuery. Replace the values in the query with a colon followed by the name of the parameter you want to use. For example:

```
Query.SQL.Text := 'insert into TBLNAMES (ID,NAME) values (:ID,:NAME);';
```

This query will create two parameters: 'ID' and 'NAME'. To determine the parameters, the query is parsed at the moment the text of *TSQLQuery.SQL* is assigned or changed. All existing parameters will be removed and the new parameters will be added to the 'TSQLQuery.Params' property. Assigning a value to a parameter is similar to assigning a value to a field in the dataset:

```
Query.Params.ParamByName('Name').AsString := 'Name1';
```

You can't tell from the query what kind of data must be stored in the parameter. The data type of the parameter is determined at the moment a value is first assigned to the parameter. By assigning a value using '.AsString', the parameter is assigned the data type 'ftString'. You can determine the data type directly by setting the 'DataType' property. If an incorrect datatype is assigned to the parameter, then problems will occur during opening or executing the query. See Database field type for more information on data types.

## Select query

An example of a select query with parameters would be to change something like this:

```
  Query.SQL.Text := 'select ID,NAME from TBLNAMES where NAME =
'''+Edit1.Text+''' ORDER BY NAME ';
```

to something like this:

```
  Query.SQL.Text := 'select ID,NAME from TBLNAMES where NAME = :NAMEPARAM
ORDER BY NAME ';
  Query.Params.ParamByName('NAMEPARAM').AsString := Edit1.Text;
```

## Example

The following example creates the same table as the previous example, but now parameters are used:

```
procedure CreateTableUsingParameters;

var
  Query : TSQLQuery;

begin
  Query := TSQLQuery.Create(nil);
  try
    Query.Database := AConnection;

    Query.SQL.Text := 'create table TBLNAMES (ID integer, NAME
varchar(40));';
    Query.ExecSQL;

    Query.SQL.Text := 'insert into TBLNAMES (ID,NAME) values (:ID,:NAME);';
    Query.Prepare;
```

```
    Query.Params.ParamByName('ID').AsInteger := 1;
    Query.Params.ParamByName('NAME').AsString := 'Name1';
    Query.ExecSQL;

    Query.Params.ParamByName('ID').AsInteger := 2;
    Query.Params.ParamByName('NAME').AsString := 'Name2';
    Query.ExecSQL;

    //Query.UnPrepare; // no need to call this; should be called by
Query.Close
    Query.Close;
  finally
    Query.Free;
  end;
end;
```

Notice that this example requires more code than the example without the parameters. Then what is the use of using parameters?

Speed is one of the reasons. The example with parameters is faster, because the database server parses the query only once (in the .Prepare statement or at first run).

Another reason to use prepared statements is prevention of SQL-injection (see also Secure programming.

Finally, in some cases it just simplifies coding.

## Troubleshooting: TSQLConnection logging

You can let a TSQLConnection log what it is doing. This can be handy to see what your Lazarus program sends to the database exactly, to debug the database components themselves and perhaps to optimize your queries. NB: if you use prepared statements/parametrized queries (see section above), the parameters are often sent in binary by the TSQLConnection descendent (e.g. TIBConnection), so you can't just copy/paste the logged SQL into a database query tool. Regardless, connection logging can give a lot of insight in what your program is doing.

Alternatives are:

1. you can use the debugger to step through the database code if you have built FPC (and Lazarus) with debugging enabled.
2. if you use ODBC drivers (at least on Windows) you could enable tracelog output in the ODBC control panel.
3. many databases allow you to monitor all statements sent to it from a certain IP address/connection.

If you use TSQLConnection logging, two things are required:

1. indicate which event types your TSQLConnection should log
2. point TSQLConnection at a function that receives the events and processes them (logs them to file, prints them to screen, etc.).

That function must be of type TDBLogNotifyEvent (see sqldb.pp), so it needs this signature:

```
TDBLogNotifyEvent = Procedure (Sender : TSQLConnection; EventType :
TDBEventType; Const Msg : String) of object;
```

## FPC (or: the manual way)

A code snippet can illustrate this:

```
uses
...
TSQLConnection, //or a child object like TIBConnection, TMSSQLConnection
...
var
type
  TMyApplication = class(TCustomApplication); //this is our application that
uses the connection
...
  private
    // This example stores the logged events in this stringlist:
    FConnectionLog: TStringList;
...
  protected
    // This procedure will receive the events that are logged by the
connection:
    procedure GetLogEvent(Sender: TSQLConnection; EventType: TDBEventType;
Const Msg : String);
...
  procedure TMyApplication.GetLogEvent(Sender: TSQLConnection;
    EventType: TDBEventType; const Msg: String);
  // The procedure is called by TSQLConnection and saves the received log
messages
  // in the FConnectionLog stringlist
  var
    Source: string;
  begin
    // Nicely right aligned...
    case EventType of
      detCustom:   Source:='Custom:  ';
      detPrepare:  Source:='Prepare: ';
      detExecute:  Source:='Execute: ';
      detFetch:    Source:='Fetch:   ';
```

```
      detCommit:   Source:='Commit:  ';
      detRollBack: Source:='Rollback:';
      else Source:='Unknown event. Please fix program code.';
    end;
    FConnectionLog.Add(Source + ' ' + Msg);
  end;

...
  // We do need to tell our TSQLConnection what to log:
    FConnection.LogEvents:=LogAllEvents; //= [detCustom, detPrepare,
detExecute, detFetch, detCommit, detRollBack]
    // ... and to which procedure the connection should send the events:
    FConnection.OnLog:=@Self.GetLogEvent;
...
  // now we can use the connection and the FConnectionLog stringlist will
fill with log messages.
```

You can also use TSQLConnection's GlobalDBLogHook instead to log everything from multiple connections.

### Lazarus (or: the quick way)

Finally, the description above is the FPC way of doing things as indicated in the introduction; if using Lazarus, a quicker way is to assign an event handler to the TSQLConnection's OnLog event.

## See also

- Working With TSQLQuery