

SQLdb Tutorial3

From Free Pascal wiki

| [English \(en\)](#) | [français \(fr\)](#) |

Contents

- 1 Overview
- 2 Multi database support
- 3 Login form
 - 3.1 Connection test callback function
 - 3.2 Additions/modifications
- 4 Getting database data into normal controls
- 5 Adapting SQL for various databases
 - 5.1 Revisiting our lowest/highest salary
- 6 Getting data out of normal controls into the database
 - 6.1 Parameterized queries
- 7 Summary
- 8 Code
- 9 See also

Overview

In this tutorial, you will learn how to

- make your application suitable for multiple databases, including using a login form
- get database data into normal controls (instead of database controls)
- get data out of the controls and back into the database
- execute parametrized queries.

Multi database support

Using any database and a login form, you can support multiple different database servers/embedded libraries that SQLDB supports.

Advantages:

- the user/programmer can dynamically use any sqldb t*connection, so he can choose between dbs

Disadvantages:

- More complicated SQL will likely need to still be adapted. Each database has its own dialect; it is of course possible to call db-specific SQL, this can grow into a maintenance problem.

Database 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

- You can't use T*connection specific properties (like *TIBConnection.Dialect* to set Firebird dialect).

To use multi-database support, instead of your specific T*Connection such as *TIBConnection*, use *TSQLConnector* (not *TSQLConnection*), which dynamically (when the program is running) chooses the specific T*Connection to use based on its *ConnectorType* property:

```
uses
...
var
  Conn: TSQLConnector;
begin
  Conn:=TSQLConnector.Create(nil);
  try
    // ...actual connector type is determined by this property.
    // Make sure the ChosenConfig.DBType string matches
    // the connectortype (e.g. see the string in the
    // T*ConnectionDef.TypeName for that connector .
    Conn.ConnectorType:=ChosenConfig.DBType;
    // the rest is as usual:
    Conn.HostName:='DBSERVER';
    Conn.DatabaseName:='bigdatabase.fdb';
    Conn.UserName:='SYSDBA';
    Conn.Password:='masterkey';
  try
    Conn.Open;
```

Login form

As mentioned in SQLdb Tutorial1, a user should login to the database using a form (or perhaps a configuration file that is securely stored), not via hardcoded credentials in the application. Besides security considerations, having to recompile the application whenever the database server information changes is not a very good idea.

In *dbconfiggui.pas*, we will set up a login form that pulls in default values from an ini file, if it exists. This allows you to set up a default connection with some details (database server, database name) filled out for e.g. enterprise deployments. In the form, the user can add/edit his username/password, and test the connection before going further.

The screenshot shows a standard Windows-style dialog box titled "Database credentials". It has a light gray background and a blue title bar with standard window controls. The dialog contains five labeled input fields: "Database type" (a dropdown menu showing "PostgreSQL"), "Host" (text box with "mypgserver"), "Database" (text box with "employee"), "User" (text box with "employee"), and "Password" (text box with "*****"). Below these fields is a "Test" button. At the bottom right are "Cancel" and "OK" buttons.

We use a separate `dbconfig.pas` unit with a `TDBConnectionConfig` class to store our chosen connection details. This class has support for reading default settings from an ini file.

This allows use without GUI login forms (e.g. when running unattended/batch operations), and allows reuse in e.g. web applications.

This `TDBConnectionConfig` class is surfaced in the login form as the `Config` property, so that the main program can show the config form modally, detect an OK click by the user and retrieve the selected configuration before closing the config form.

Connection test callback function

To keep the login form flexible (it may be used with other database layers like Zeos), we implement the test section as a callback function and let the main program deal with it.

The definition in the login form in `dbconfiggui.pas`:

```
type
  TConnectionTestFunction = function(ChosenConfig: TDBConnectionConfig): boolean of object;
```

The main form must implement a function that matches this definition to handle the test request from the config form.

The callback function takes the config object passed on by the config form, and uses that to construct a connection with the chosen database type. It then simply tries to connect with the server; if it is succesful, it sets the function result to true, otherwise the result stays false.

Because database connection attempts to non-existing servers can have a long timeout, we indicate to the user that he must wait by setting the cursor to the hourglass icon.

```
uses
...
dbconfig, dbconfiggui
...
procedure TForm1.FormCreate(Sender: TObject);
  LoginForm:=TDBConfigForm.Create(self);
  try
    // The test button on dbconfiggui will link to this procedure:
    ... this links the callback in 'dbconfiggui.pas' to the ConnectionTest function here...
    LoginForm.ConnectionTestCallback:=@ConnectionTest;
  ...
function TForm1.ConnectionTest(ChosenConfig: TDBConnectionConfig): boolean;
// Callback function that uses the info in dbconfiggui to test a connection
// and return the result of the test to dbconfiggui
var
  // Generic database connector...
  Conn: TSQLConnector;
begin
  result:=false;
  Conn:=TSQLConnector.Create(nil);
  Screen.Cursor:=crHourglass;
  try
    // ...actual connector type is determined by this property.
    // Make sure the ChosenConfig.DBType string matches
    // the connectortype (e.g. see the string in the
    // T*ConnectionDef.TypeName for that connector .
    Conn.ConnectorType:=ChosenConfig.DBType;
    Conn.HostName:=ChosenConfig.DBHost;
    Conn.DatabaseName:=ChosenConfig.DBPath;
    Conn.UserName:=ChosenConfig.DBUser;
    Conn.Password:=ChosenConfig.DBPassword;
    try
      Conn.Open;
      result:=Conn.Connected;
    except
      // Result is already false
    end;
    Conn.Close;
  finally
    Screen.Cursor:=crDefault;
    Conn.Free;
  end;
end;
```

Finally, the code in *dbconfiggui.pas* that actually calls the callback is linked to the Test button. It tests if the callback function is assigned (to avoid crashes), for completeness also checks if there is a valid configuration object and then simply calls the callback function:

```
...
TDBConfigForm = class(TForm)
...
  private
    FConnectionTestFunction: TConnectionTestFunction;
  public
    property ConnectionTestCallback: TConnectionTestFunction write FConnectionTestFunction;
  ...
procedure TDBConfigForm.TestButtonClick(Sender: TObject);
begin
  // Call callback with settings, let it figure out if connection succeeded and
  // get test result back
  if assigned(FConnectionTestFunction) and assigned(FConnectionConfig) then
    if FConnectionTestFunction(FConnectionConfig) then
      showmessage('Connection test succeeded.')
    else
      showmessage('Connection test failed.')
  else
    showmessage('Error: connection test code has not been implemented.');
```

Additions/modifications

Possible additions/modifications for the login form:

- Add command line arguments handling for dbconfig to preload suitable defaults, so the program can be used in batch scripts, shortcuts etc
- Add a "Select profile" combobox in the login form; use multiple profiles in the ini file that specify database type and connection details.
- Hide database type combobox when only one database type supported.
- Hide username/password when you're sure an embedded database is selected.
- Add support for specifying port number, or instance name with MS SQL Server connector
- Add support for trusted authentication for dbs that support it (Firebird, MS SQL): disable username/password controls
- If an embedded database is selected but the file does not exist: show a confirmation request and create the database
- Create a command-line/TUI version of the login form (e.g. using the curses library) for command-line applications

Updates to this article/the code warmly welcomed.

Getting database data into normal controls



Note: Before starting this section, please make sure you have set up the sample employee database as specified in SQLdb Tutorial0#Requirements

In previous tutorials, data-bound controls were covered: special controls such as the TDBGrid that can bind its contents to a TDataSource, get updates from that source and send user edits back.

It is also possible to programmatically retrieve database content and fill any kind of control (or variable) with that content. As an example, we will look at filling a stringgrid with some salary details for the sample employee database table.

On the main form, let's add a TStringGrid and retrieve the data (e.g. via a procedure *LoadSalaryGrid* called in the *OnCreate* event):

```
// Load from DB
try
  if not FConn.Connected then
    FConn.Open;
  if not FConn.Connected then
  begin
    ShowMessage('Error connecting to the database. Aborting data loading.');
```

```
    exit;
  end;

  // Lowest salary
  // Note: we would like to only retrieve 1 row, but unfortunately the SQL
  // used differs for various dbs. As we'll deal with db dependent SQL later
  // in the tutorial, we leave this for now.
  // MS SQL: 'select top 1 '...
  FQuery.SQL.Text:='select ' +
    '    e.first_name, ' +
    '    e.last_name, ' +
    '    e.salary ' +
    'from employee e ' +
    'order by e.salary asc ';
```

```

        // ISO SQL+Firebird SQL: add
        //'rows 1 '; here and below... won't work on e.g. PostgreSQL though
FTran.StartTransaction;
FQuery.Open;
SalaryGrid.Cells[1,1]:=FQuery.Fields[0].AsString; // i.e. Cells[Col,Row]
SalaryGrid.Cells[2,1]:=FQuery.Fields[1].AsString;
SalaryGrid.Cells[3,1]:=FQuery.Fields[2].AsString;
FQuery.Close;
// Always commit(retain) an opened transaction, even if only reading
// this will allow updates by others to be seen when reading again
FTran.Commit;
...
end;
except
on D: EDatabaseError do
begin
    MessageDlg('Error', 'A database error has occurred. Technical error message: ' +
        D.Message, mtError, [mbOK], 0);
end;
end;
end;

```

Things to note: we catch database errors using `try..except`. You'll notice we forgot to roll back the transaction in case of errors - which is left as an exercise to the reader.

We *Open* the query object, thereby asking *FQuery* to query the database via its *SQL* statement. Once this is done, we're on the first row of data. We simply assume there is data now; this is actually a programming error: it would be tidier to check for *FQuery.EOF* being true (or *FQuery.RecordCount* being >0).

Next, we retrieve the data from the first row of results. If we wanted to move to the next row, we'd use *FQuery.Next*, but that is not necessary here. We put the results in the stringgrid, giving the lowest salary in the list. A similar approach can be taken for the highest salary.

Adapting SQL for various databases

As we noticed above, various databases support various versions of SQL (either in addition to or in contradiction to the official ISO SQL standards). Fortunately, you can customize your application based on which DB it ends up using, which will be demonstrated by getting the standard deviation of the employees' salaries - built into e.g. PostgreSQL SQL but not available by default in e.g. Firebird.

In our `LoadSalaryGrid` procedure, we'll use the SQL for PostgreSQL and build a code solution for all other databases. First detect which database is loaded, below the other lines add:

```

...
SalaryGrid.Cells[3,2]:=FQuery.Fields[2].AsString;
FQuery.Close;
// Always commit(retain) an opened transaction, even if only reading
FTran.Commit;
//end of existing code

if FConn.ConnectorType = 'PostgreSQL' then
begin
    // For PostgreSQL, use a native SQL solution:
    FQuery.SQL.Text:='select stddev_pop(salary) from employee ';
    FTran.StartTransaction;
    FQuery.Open;
    if not FQuery.EOF then
        SalaryGrid.Cells[3,3]:=FQuery.Fields[0].AsString;
    FQuery.Close;
    // Always commit(retain) an opened transaction, even if only reading
    FTran.Commit;
end
else
begin
    // For other database, use the code approach:

```

```

    ....see below...
end;

```

Notice the use of *ConnectorType*; the string used must match exactly. We also properly check for empty results from the query (which might happen if the employee table is empty).

... now let's implement a code-based solution for other databases that do not support standard deviation:

```

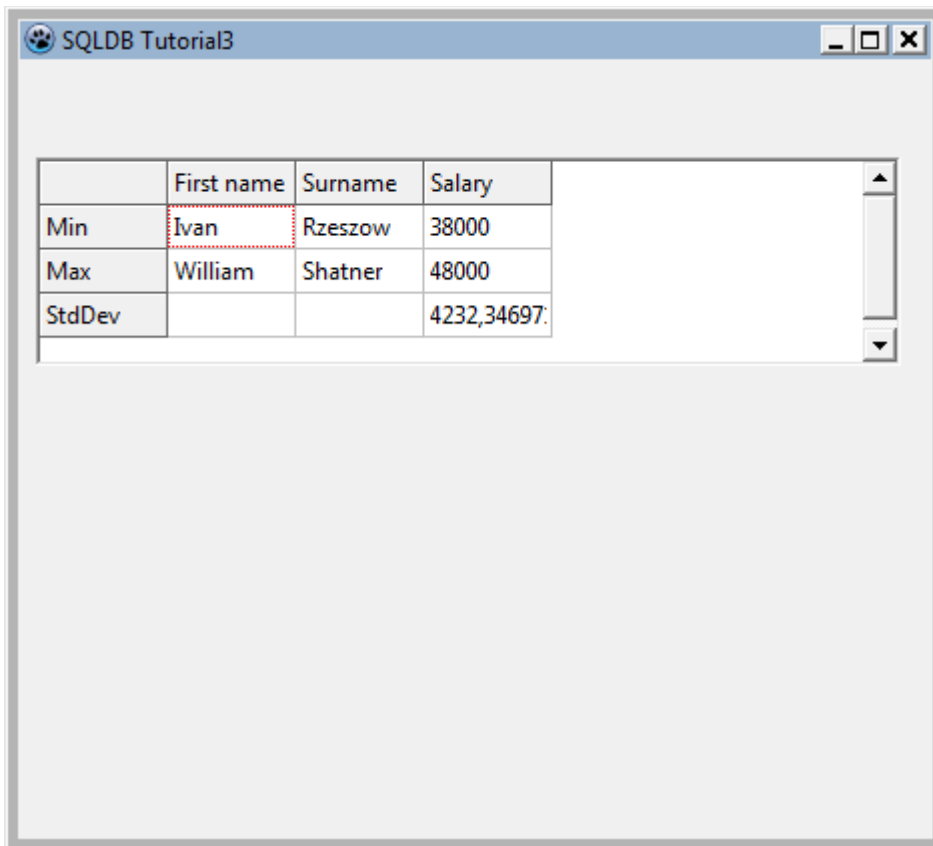
// For other databases, use the code approach:
// 1. Get average of values
FQuery.SQL.Text:='select avg(salary) from employee ';
FQuery.Open;
if FQuery.EOF then
    SalaryGrid.Cells[3,3]:='No data'
else
    begin
        Average:=FQuery.Fields[0].AsFloat;
        FQuery.Close;
        // 2. For each value, calculate the square of (value-average), and add it up
        FQuery.SQL.Text:='select salary from employee where salary is not null ';
        FQuery.Open;
        while not FQuery.EOF do
            begin
                DifferencesSquared:=DifferencesSquared+Sqr(FQuery.Fields[0].AsFloat-Average);
                Count:=Count+1;
                FQuery.Next;
            end;
        // 3. Now calculate the average "squared difference" and take the square root
        if Count>0 then //avoid division by 0
            SalaryGrid.Cells[3,3]:=FloatToStr(Sqrt(DifferencesSquared/Count))
        else
            SalaryGrid.Cells[3,3]:='No data';
        end;
        FQuery.Close;
    end;

```

Note that we use *FQuery.EOF* to check for empty data (and avoid division by zero errors etc). The loop shows how to:

- retrieve a database value into a variable
- use *FQuery.Next* to move to the next record
- properly check if the query dataset has hit the last record, then stop retrieving data.

The resulting screen should show something like this - note the use of a decimal comma - while your computer may show a decimal point depending on your locale:



The screenshot shows a window titled "SQLDB Tutorial3" containing a table with the following data:

	First name	Surname	Salary
Min	Ivan	Rzeszow	38000
Max	William	Shatner	48000
StdDev			4232,34697

Revisiting our lowest/highest salary

This section gives some more useful details on SQL but is not required to work through for the rest of the tutorial

Now we know how to deal with detecting various database connections, we can adjust the SQL that gets the lowest and highest salary as well to make use of db specific functionality.

An example: this would work for MS SQL Server by limiting the number of returned rows to just the first:

```
select top 1
e.first_name, e.last_name, e.salary
from employee e
order by e.salary asc
```

to get the lowest salary.

This efficiently returns one record. Other databases use other syntax, such as the ISO ROWS 1. The diligent SQL student will soon learn not to miss out that important part and request entire large recordsets just for one required record!

Let's briefly examine other ways to achieve the same thing, that are worth knowing.

Another way to retrieve the record(s) with the minimum salary would be :

```
SELECT e.first_name, e.last_name, e.salary FROM employee e WHERE e.salary=(SELECT min(salary) FROM employee)
```


SQL students would greatly benefit from researching Common Table Expressions.

A CTE allows a virtual temporary table to be used in a following expression, allowing you to clearly code some very complex queries that otherwise may not be possible. Knowing about CTEs will catapult you ahead of colleagues who have never heard of them! For example the above may be rewritten (example in Microsoft SQL Server syntax) as :

```
WITH TheMinimum as
(
  SELECT min(salary) as MinimumPay FROM Employee
)
SELECT e.first_name, e.last_name, e.salary FROM Employee e WHERE e.salary=(SELECT MinimumPay FROM TheMinimum)
```

Several such temporary tables may be chained together, each using the results from the previous tables. You can treat these virtual tables as though they were real tables in the database, using JOINS to link recordsets together. And it can be very useful for quick tests using hardcoded data - this can be run without any database connection :

```
WITH TestEmployee as
(
  SELECT 'Fred' as first_name, 'Bloggs' as last_name, 10500 as salary
  UNION
  SELECT 'Joe' as first_name, 'Public' as last_name, 10000 as salary
  UNION
  SELECT 'Mike' as first_name, 'Mouse' as last_name, 11000 as salary
),
TheMinimum as
(
  SELECT min(salary) as MinimumPay FROM TestEmployee
)
SELECT e.first_name, e.last_name, e.salary FROM TestEmployee e WHERE e.salary=(SELECT MinimumPay FROM TheMinimum)
```

You can end up with quite long strings for the code of such SQL queries, but it is only *one* query and may be called from anywhere where you are limited to a simple single expression - it can be useful to answer complex queries without resorting to functions or stored procedures.

Getting data out of normal controls into the database

Previously, we have seen:

- how to let SQLDB update the database with data-bound controls (earlier tutorials)
- how to get data out of the database using queries (the section above)

You can also execute SQL to get arbitrary data back into the database via code. This allows you to use variables or controls that have no db aware equivalent such as sliders or custom controls to enter data into the database, at the expense of a bit more coding.

As an example, we are going to allow the user to change the lowest and highest salary in the stringgrid.

For ease of editing, set the grid's *Options/goEditing* to true; then assign the procedure below to the OnValidate event for the grid, which will be called every time a user has finished updating the grid.

Parameterized queries

The following code also demonstrates how to use parameterized queries to avoid SQL injection, fiddling with quoting for string values, date formatting, etc.

As you can see in the code, you can name your parameters whatever you wish and prefix them with `:` in the SQL. In code, you can set/get their values by
`<somequery>.Params.ParamByName('<thename>').As'<variabletype>';` the code demonstrates `.AsFloat` and `.AsString`.

Parameterized queries are especially useful (and can be much faster) if you run the same query, only with different parameters, in a loop (think e.g. bulk loading of data).

Continuing with our example: after having set up the query SQL and parameters, the transaction is started (and later on committed) as usual, then the query is run by calling `ExecSQL` (which does not return a result set; if the SQL statement were e.g. a SELECT or INSERT...RETURNING that does return data, you would use `Open` as in the examples above):

```
procedure TForm1.SalaryGridValidateEntry(sender: TObject; aCol, aRow: Integer;
const OldValue: string; var NewValue: String);
begin
    // Only these cells have min and max salary:
    if (aCol=3) and ((aRow=1) or (aRow=2)) then
    begin
        // Allow updates to min and max salary if positive numerical data is entered
        if StrToFloatDef(NewValue,-1)>0 then
        begin
            // Storing the primary key in e.g. a hidden cell in the grid and using that in our
            // update query would be cleaner, but we can do it the hard way as well:
            FQuery.SQL.Text:='update employee set salary=:newsalary '+
                ' where first_name=:firstname and last_name=:lastname and salary=:salary ';
            FQuery.Params.ParamByName('newsalary').AsFloat:=StrToFloatDef(NewValue,0);
            FQuery.Params.ParamByName('firstname').AsString:=SalaryGrid.Cells[1,aRow];
            FQuery.Params.ParamByName('lastname').AsString:=SalaryGrid.Cells[2,aRow];
            FQuery.Params.ParamByName('salary').AsFloat:=StrToFloatDef(OldValue,0);
            FTran.StartTransaction;
            FQuery.ExecSQL;
            FTran.Commit;
            LoadSalaryGrid; //reload standard deviation
        end
        else
        begin
            // Notify user that his input was wrong... he'll be wondering otherwise:
            Showmessage('Invalid salary entered.');
```

Note how we forgot to add a `try..except` block to this code to nicely catch database errors and display a sensible error message. If you are running the Firebird sample EMPLOYEE database for this tutorial, try to change the salary to a very low value (say 1) and see what happens.

Finally, while this example showed an UPDATE SQL query, you could just as well run INSERT queries to insert new data programmatically. Also, you can use parameters in any kind of SQL query (SELECT, UPDATE, etc) as long as you use them for fields, not for table/view/procedure names.

Summary

This tutorial explained:

- how to code for multiple database types
- how to use a login form to decouple db access configuration from your program
- how to retrieve and update data programmatically

Code

Since November 2012, the code can be found in `$(lazarusdir)examples/database/sqldbtutorial3`

If you have an older version (e.g. Lazarus 1.0.2), you can also download the code via the Lazarus SVN website (<http://svn.freepascal.org/svn/lazarus/trunk/examples/database/sqldbtutorial3/>)

See also

- SQLdb Tutorial0: Instructions for setting up sample tables/sample data for the tutorial series.
- SQLdb Tutorial1: First part of the DB tutorial series, showing how to set up a grid that shows database data
- SQLdb Tutorial2: Second part of the DB tutorial series, showing editing, inserting etc.
- SQLdb Tutorial4: Fourth part of the DB tutorial series, showing how to use data modules
- Lazarus Database Overview: Information about the databases that Lazarus supports. Links to database-specific notes.
- SQLdb Package: information about the SQLdb package
- SQLdb Programming Reference: an overview of the interaction of the SQLdb database components
- using parameters (<http://www.freepascal.org/docs-html/fcl/sqlldb/usingparams.html>)
- SqlDBHowto: information about using the SQLdb package
- Working With TSQLQuery: information about TSQLQuery

Retrieved from "https://wiki.freepascal.org/index.php?title=SQLdb_Tutorial3&oldid=112394"

-
- This page was last edited on 5 October 2017, at 10:26.
 - Content is available under unless otherwise noted.