SQLdb Tutorial2

From Free Pascal wiki

| **English (en)** | **français (fr)** |

Contents

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

Overview

If you have followed SQLdb Tutorial1, you have a basic grid showing database information. While this application works, we can add some refinements on this.

Dynamic database connection

Up to now, we used a fixed database server name, database location, username and password for simplicity. As mentioned, "real" applications normally let users specify their own username and password.

Let's change the form so we can specify them: add two TEdits from the standard menu. Set their name properties to Username and Password. Set the Password's PasswordChar property to * (the asterisk) for some security against people looking over your shoulder.

If you want to make it easier (and less secure, of course) to connect, you can set the UserName Text property to a valid database user, such as SYSDBA. You could even set the Password Text property to a default value like masterkey, easy for testing on developer machines if security doesn't matter...

Cosmetically, adding some labels so people know what they're supposed to type is useful.

Also, to make it easier to connect to any employee sample database on any Firebird/Interbase server, we add two textboxes for server name and database path. Add another two TEdits, and name them ServerName and DatabaseName.

If you want, you can set the 'Text' property to default sensible values for your situation, e.g. localhost and `C:\Program Files\Firebird\Firebird_2_5\examples\empbuild\EMPLOYEE.FDB`

Labels to explain what users need to enter would help here, too.

For clarity, we're going to remove the connection info from our design time components: on the TSQLConnector component, remove all text from the UserName, Password, DatabaseName and HostName properties.

Now, finally, we need to tell our database connection component how to connect. This should normally only be necessary at the beginning of an application run. In our case the existing 'Button1' code is a good way to set up the connection.

Add code until you get:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SQLQuery1.Close;
  //Connection settings for Firebird/Interbase database
  //only needed when we have not yet connected:
  if not DBConnection.Connected then
  begin
    DBConnection.HostName := ServerName.Text;
    DBConnection.DatabaseName := DatabaseName.Text;
    DBConnection.Username := UserName.Text;
    DBConnection.Password := Password.Text;
    // Now we've set up our connection, visually show that
    // changes are not possibly any more
    ServerName.ReadOnly:=true;
    DatabaseName.ReadOnly:=true;
    UserName.ReadOnly:=true;
    Password.ReadOnly:=true;
  end;
  SQLQuery1.SQL.Text:= 'select * from CUSTOMER';
  DBConnection.Connected:= True;
  SQLTransaction1.Active:= True;
  SQLQuery1.Open;
end;
```

Now run and test if you can connect.

SQLite, other databases

Adjust the Text property in the DatabaseName TEdit as needed; e.g. `employee.sqlite` for SQLite.

For sqlite, specifying HostName, Username and Password doesn't make sense, so you can omit these TEdits. Obviously, leave out/comment out assigning the corresponding values to DBConnection in the code above. For Firebird embedded, please hardcode the Username to SYSDBA; specifying this when sqlite is used won't hurt.

The code will look something like:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  SQLQuery1.Close;
  //Connection settings for embedded databases
  //only needed when we have not yet connected:
  if not DBConnection.Connected then
  begin
    DBConnection.DatabaseName := DatabaseName.Text;
    DBConnection.UserName := 'SYSDBA'; //Firebird embedded needs this; doesn't harm if using SQLite
    // Now we've set up our connection, visually show that
    // changes are not possibly any more
    DatabaseName.ReadOnly:=true;
  end;
  SQLQuery1.SQL.Text:= 'select * from CUSTOMER';
  DBConnection.Connected:= True;
  SQLTransaction1.Active:= True;
  SQLQuery1.Open;
end;
```

Filtering data

Often, tables contain a huge amount of data that the user doesn't want to see (and that might take a long time to query from the database and travel over the network). Let's assume that only the customers from the USA should be displayed. Therefore the SQL instruction in 'SQLQuery1' would look like:

```
select * from CUSTOMER where COUNTRY = 'USA'
```

... which would translate to something like this in our code:

```
SQLQuery1.SQL.Text := 'select * from CUSTOMER where COUNTRY = 'USA'';
```

There are two reasons why we will not use this instruction for our example application:

First there is a problem with the usage of the single quote. The compiler would interpret the quote before USA as a closing quote (the first quote is before the select from...) and so the SQL instruction would become invalid. Solution: double the inside quotes:

```
SQLQuery1.SQL.Text := 'select * from CUSTOMER where COUNTRY = ''USA''';
```

The second, more important reason is the fact, that we probably don't know what constraints the user will want to filter on. We don't want to limit the flexibility of the user.

To get this flexibility, first we change our SQL query statement and replace 'USA' by a placeholder (a parameter in SQL speak): change the Button1click procedure and replace

```
SQLQuery1.SQL.Text := 'select * from CUSTOMER';
```

with

```
SQLQuery1.SQL.Text:= 'select * from CUSTOMER where COUNTRY = :COUNTRY';
```

In FPC SQLDB, the SQL parameter is marked by the leading colon (other languages/environments use other conventions like ?). To allow the user to enter a value for the filter, we place a *TEdit* component on our form. Delete the value of its 'Text' property.

We can now take the text entered in the TEdit and fill the SQL COUNTRY parameter by using the 'Params' property of TSQLQuery. Add this below the previous statement:

```
SQLQuery1.Params.ParamByName('COUNTRY').AsString := Edit1.Text;
```

The parameter can be specified by its position or name. Using the name should improve the readability of the source code, and obviously helps if you insert more parameters in the middle of existing parameters.

We use .AsString to assign a string value to the parameter; there are equivalent property assignments for integer parameters, boolean parameters etc.

The code up to now forces us to use a filter. If a user specifies an empty value in the edit box, no record will be displayed. This is probably not what we want. Let's test for an empty value and build our query accordingly. We should end up with a procedure like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SQLQuery1.Close;
  //Connection settings for Firebird/Interbase database
  //only needed when we have not yet connected:
  if not DBConnection.Connected then
  begin
    DBConnection.HostName := ServerName.Text;
    DBConnection.DatabaseName := DatabaseName.Text;
    DBConnection.Username := UserName.Text;
    DBConnection.Password := Password.Text;
    // Now we've set up our connection, visually show that
    // changes are not possibly any more
    ServerName.ReadOnly:=true;
    DatabaseName.ReadOnly:=true;
    UserName.ReadOnly:=true;
    Password.ReadOnly:=true;
  end;
  // Show all records, or filter if user specified a filter criterium
  if Edit1.Text='' then
    SQLQuery1.SQL.Text := 'select * from CUSTOMER'
  else
  begin
    SQLQuery1.SQL.Text := 'select * from CUSTOMER where COUNTRY = :COUNTRY';
    SQLQuery1.Params.ParamByName('COUNTRY').AsString := Edit1.Text;
  end;
  DBConnection.Connected:= True;
  SQLTransaction1.Active:= True;
  SQLQuery1.Open;
end;
```

Now you can play around a bit with filtering using Edit1. If you enter a country that's not present in the database, an empty grid is shown.

Error handling

The application should run, but sometimes problems can occur. Databases, even embedded databases can crash (e.g. when the database server crashes, the disk is full, or just due to a bug), leaving the application hanging.

Access to a database (any external process, really) should therefore **always** be integrated in a try ... except and/or try ... finally construct. This ensures that database errors are handled and the user isn't left out in the cold. A rudimentary routine for our example application could look like this:

```
begin
```

```
  try
    SQLQuery1.Close;
    ...
    SQLQuery1.Open;
  except
    //We could use EDatabaseError which is a general database error, but we're dealing with Firebird/Interbase,
so:
    on E: EDatabaseError do
    begin
      MessageDlg('Error','A database error has occurred. Technical error message: ' + E.Message,mtError,
[mbOK],0);
      Edit1.Text:='';
    end;
  end;
end;
```

SQLite, PostgreSQL, other databases

You can either use the more generic EDatabaseError, or - if available - your own specialized databaseerror, if you need more details. E.g. SQLite and the PostgreSQL driver in FPC 2.6.1 and lower doesn't have a specialized E*DatabaseError; you'd have to use EDatabaseError. PostgreSQL on FPC trunk (development version) has EPQDatabaseError.

Editing data using the grid

Editing

Up to now, if you tried to edit data in the grid, the changes would not be saved. This is because the *SQLQuery1* is not instructed to send the changes to the database transaction at the right moment. We need to fix this, and then commit the transaction in the database, so all changes get written. For this, you would use code like this:

```
SQLQuery1.ApplyUpdates; //Pass user-generated changes back to database...
SQLTransaction1.Commit; //... and commit them using the transaction.
//SQLTransaction1.Active now is false
```

We want to make sure any edits (inserts, updates, deletes) are written to the database:

- when the users changes the filtering criteria and presses the button to query the database
- when the form is closed

It makes sense to make a separate procedure for this that is called in those two instances. Go to the code, and add an empty line here:

```
  TForm1 = class(TForm)
    Button1: TButton;
    Datasource1: TDatasource;
    DBGrid1: TDBGrid;
    Edit1: TEdit;
    DBConnection: TIBConnection;
    SQLQuery1: TSQLQuery;
    SQLTransaction1: TSQLTransaction;
*****insert the empty line here****
    procedure Button1click(Sender: TObject);
    procedure Formclose(Sender: TObject; var Closeaction: Tcloseaction);
  private
```

then type

```
    procedure SaveChanges;
```

press shift-ctrl-c (default combination) to let code completion automatically create the corresponding procedure body.

We need to add error handling and check that the transaction is active - remember, this code also gets called when pressing the button the first time, when the transaction is not active yet. We get:

```
procedure Tform1.SaveChanges;
// Saves edits done by user, if any.
begin
  try
    if SQLTransaction1.Active then
    // Only if we are within a started transaction;
    // otherwise you get "Operation cannot be performed on an inactive dataset"
    begin
      SQLQuery1.ApplyUpdates; //Pass user-generated changes back to database...
      SQLTransaction1.Commit; //... and commit them using the transaction.
      //SQLTransaction1.Active now is false
    end;
  except
  on E: EDatabaseError do
    begin
      MessageDlg('Error', 'A database error has occurred. Technical error message: ' +
        E.Message, mtError, [mbOK], 0);
      Edit1.Text := '';
    end;
  end;
end;
```

Now we need to call this procedure at the appropriate moments:

```
procedure Tform1.Button1click(Sender: TObject);
begin
  SaveChanges; //Saves changes and commits transaction
  try
    SQLQuery1.Close;
....
```

and

```
procedure Tform1.Formclose(Sender: TObject; var Closeaction: Tcloseaction);
begin
  SaveChanges; //Saves changes and commits transaction
  SQLQuery1.Close;
....
```

Now test and see if edits made in the dbgrid are saved to the database.

Hiding primary key column

Often, you don't want your users to see autonumber/generated primary keys as they are only meant to maintain referential integrity. If users do see them, they might want to try the edit the numbers, get upset that the numbers change, that there are gaps in the numbers, etc.

In our example, CUST_NO is the primary key, with content auto-generated by Firebird using triggers and a sequence/generator. This means that you can insert a new record without specifying the CUST_NO; Firebird will create one automatically.

We could simply change our SQLQuery1.SQL.Text property to not include CUST_NO, but this would lead to problems when editing data - a primary key is needed in those circumstances for uniquely identifying the row/record in question.

Therefore, let's use a trick to query for all columns/fields in the table, but keep the grid from showing the first field, CUST_NO: in the Button1Click procedure, add code so it looks like:

```
procedure Tform1.Button1click(Sender: TObject);
begin
...
    SQLQuery1.Open;
    // Hide the primary key column which is the first column in our queries.
    // We can only do this once the DBGrid has created the columns
    DBGrid1.Columns[0].Visible:=false;
```

Recompile, and check to see if the primary key column is really hidden.

SQLite, other databases

- Other databases: a lot of other databases use an 'autonumber' or 'autoinc' type of field to provide auto-generated field content. Try changing your table definition and see if it works.
- Sqlite: the example above works for SQLite as is because we're using an integer primary key. See the documentation (http://www.sqlite.org/autoinc.html) for details.

Inserting new data

If you insert new rows/records without any CUST_NO information you may have noticed that you get an error message: `Field CUST_NO is required, but not supplied`. This also happens if you hid the CUST_NO column, as in the previous section.

The reason: Lazarus thinks that CUST_NO is required. That's not so strange, because it is a primary key and the underlying table definition in the database does say it is required.

If we can instruct Lazarus that this field is not actually required, we can pass empty values (=NULL values) to the database. Fortunately, a query's field object has a *Required* property that does exactly that.

Change the code to something like:

```
    SQLQuery1.Open;
    {
    Make sure we don't get problems with inserting blank (=NULL) CUST_NO values, e.g.:
    Field CUST_NO is required, but not supplied
    We need to tell Lazarus that, while CUST_NO is a primary key, it is not required
    when inserting new records.
    }
    SQLQuery1.FieldByName('CUST_NO').Required:=false;
    // Hide the primary key column which is the first column in our queries.
    // We can only do this once the DBGrid has created the columns
    DBGrid1.Columns[0].Visible:=false;
```

Deleting data

You can let your users use the mouse to do this. You don't even need to code a single line for this functionality...

On the 'Data Controls' tab, select a *TDBNavigator* component and drop it on the form, above the grid.

To indicate what the navigator should be linked to, set its *DataSource* property to your existing datasource ('DataSource1') using the Object Inspector. Now you can use the button on the *DBNavigator* to delete records, but also insert them, and move around the records. Also, when editing cells/fields, you can use the *Cancel* button to cancel your edits.

To allow users to delete the row they're in on the grid using the Delete key, add *LCLType* (this contains defintions for key codes) to your uses clause:

```
uses
  Classes, SysUtils, sqldb, pqconnection, DB, FileUtil, Forms,
  Controls, Graphics, Dialogs, DBGrids, StdCtrls, DbCtrls, LCLType;
```

... then handle the KeyUp event for the grid, which occurs when a key is released if in the grid. However, we do need to check that the user is not editing a field - as he'll probably use the Delete key to delete letters rather than the record he's working on.

Select the grid, then go to events and create an OnKeyUp event like this:

```
procedure TForm1.DBGrid1KeyUp(Sender: TObject; var Key: Word; Shift: TShiftState
  );
begin
  // Check for del key being hit and delete the current record in response
  // as long as we're not editing data
  if (key=VK_DELETE) and (not(DBGrid1.EditorMode)) then
  begin
    //... delete current record and apply updates to db:
    SQLQuery1.Delete;
    SQLQuery1.ApplyUpdates;
  end;
end;
```

Note: By default TDBGrid property Options / dgDisableDelete is set to false, this means a user can delete any record with the ctrl - delete key combo. You may not want this behaviour.

Summary

If you followed along up to now, you can retrieve data from the database, filter it, and edit and delete data in the grid. Your code should look something like this:

```
unit sqldbtutorial1unit;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, sqldb, pqconnection, DB, FileUtil, Forms,
  Controls, Graphics, Dialogs, DBGrids, StdCtrls, DbCtrls, LCLType;
type

  { TForm1 }

  TForm1 = class(TForm)
    Button1: TButton;
    DatabaseName: TEdit;
    Datasource1: TDatasource;
    DBGrid1: TDBGrid;
    Dbnavigator1: Tdbnavigator;
```

```
    Edit1: TEdit;
    Label2: Tlabel;
    Label3: Tlabel;
    Label4: Tlabel;
    Label5: Tlabel;
    Password: TEdit;
    UserName: TEdit;
    ServerName: TEdit;
    DBConnection: TIBConnection;
    Label1: TLabel;
    SQLQuery1: TSQLQuery;
    SQLTransaction1: TSQLTransaction;
    procedure SaveChanges;
    procedure Button1click(Sender: TObject);
    procedure Formclose(Sender: TObject; var Closeaction: Tcloseaction);
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

{ TForm1 }

procedure Tform1.Savechanges;
// Saves edits done by user, if any.
begin
  try
    if SQLTransaction1.Active then
    // Only if we are within a started transaction
    // otherwise you get "Operation cannot be performed on an inactive dataset"
    begin
      SQLQuery1.ApplyUpdates; //Pass user-generated changes back to database...
      SQLTransaction1.Commit; //... and commit them using the transaction.
      //SQLTransaction1.Active now is false
    end;
  except
  on E: EDatabaseError do
    begin
      MessageDlg('Error', 'A database error has occurred. Technical error message: ' +
        E.Message, mtError, [mbOK], 0);
      Edit1.Text := '';
    end;
  end;
end;

procedure TForm1.DBGrid1KeyUp(Sender: TObject; var Key: Word; Shift: TShiftState
  );
begin
  // Check for del key being hit and delete the current record in response
  // as long as we're not editing data
  if (key=VK_DELETE) and (not(DBGrid1.EditorMode)) then
  begin
    //... delete current record and apply updates to db:
    SQLQuery1.Delete;
    SQLQuery1.ApplyUpdates;
  end;
end;

procedure Tform1.Button1click(Sender: TObject);
begin
  SaveChanges; //Saves changes and commits transaction
  try
    SQLQuery1.Close;
    //Connection settings for Firebird/Interbase database
    //only needed when we have not yet connected:
    if not DBConnection.Connected then
    begin
      DBConnection.HostName := ServerName.Text;
      DBConnection.DatabaseName := DatabaseName.Text;
      DBConnection.Username := UserName.Text;
      DBConnection.Password := Password.Text;
      // Now we've set up our connection, visually show that
```

```
      // changes are not possibly any more
      ServerName.ReadOnly:=true;
      DatabaseName.ReadOnly:=true;
      UserName.ReadOnly:=true;
      Password.ReadOnly:=true;
    end;
    // Show all records, or filter if user specified a filter criterium
    if Edit1.Text='' then
      SQLQuery1.SQL.Text := 'select * from CUSTOMER'
    else
    begin
      SQLQuery1.SQL.Text := 'select * from CUSTOMER where COUNTRY = :COUNTRY';
      SQLQuery1.Params.ParamByName('COUNTRY').AsString := Edit1.Text;
    end;
    DBConnection.Connected := True;
    SQLTransaction1.Active := True; //Starts a new transaction
    SQLQuery1.Open;
    {
    Make sure we don't get problems with inserting blank (=NULL) CUST_NO values, i.e. error message:
    "Field CUST_NO is required, but not supplied"
    We need to tell Lazarus that, while CUST_NO is a primary key, it is not required
    when inserting new records.
    }
    SQLQuery1.FieldByName('CUST_NO').Required:=false;
    {
    Hide the primary key column which is the first column in our queries.
    We can only do this once the DBGrid has created the columns
    }
    DBGrid1.Columns[0].Visible:=false;
  except
    // EDatabaseError is a general error;
    // you could also use one for your specific db, e.g.
    // use EIBDatabaseError for Firebird/Interbase
    on E: EDatabaseError do
    begin
      MessageDlg('Error', 'A database error has occurred. Technical error message: ' +
        E.Message, mtError, [mbOK], 0);
      Edit1.Text := '';
    end;
  end;
end;

procedure Tform1.Formclose(Sender: TObject; var Closeaction: Tcloseaction);
begin
  SaveChanges; //Saves changes and commits transaction
  SQLQuery1.Close;
  SQLTransaction1.Active := False;
  DBConnection.Connected := False;
end;

end.
```

Embedded database without code changes

Firebird on Windows

A bonus for Firebird users on Windows: if you have been following this tutorial (even if you only did the basic example), you renamed the `fbembed.dll` embedded Firebird library to `fbclient.dll`. With this, Lazarus could connect to regular Firebird servers (either on another machine or on your local machine). However, you can also copy the `employee.fdb` database to your application directory, run the application, clear the *Server name* TEdit and use Firebird embedded to directly connect to the database file, without any servers set up.

This is great if you want to deploy database applications to end users, but don't want the hassle of installing servers (checking if a server is already installed, if it's the right version, having users check firewalls, etc).

See Firebird embedded for more details.

September 2011: in recent development (SVN) versions of Free Pascal, FPC tries to first load `fbembed.dll`, so

you need not rename `fbclient.dll` anymore for this to work.

Firebird on Linux/OSX/Unix

There must be a way to get this to work on Linux/OSX. See Firebird for hints and links. Updates to the wiki are welcome.

SQLite

SQLite certainly offers embedded functionality - it does not allow a client/server setup on the other hand. By following the tutorial above, you can see that switching between databases (e.g. SQLite and Firebird) is not so much work at all.

Other databases

Your database might offer similar functionality. Updates of this wiki for other database systems are welcome.

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 Tutorial3: Third part of the DB tutorial series, showing how to program for multiple databases and use a login form
- 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
- SqlDBHowto: information about using the SQLdb package
- Working With TSQLQuery: information about TSQLQuery