SQLite

From Free Pascal wiki

| **English (en)** | **español (es)** | **français (fr)** | **日本語 (ja)** |

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

SQLite and FPC/Lazarus support

SQLite is an embedded (non-server) single-user database that can be used in FPC and Lazarus applications. Various drivers can be used to access SQLite from FPC/Lazarus programs. All drivers do need the SQLite library/dll in the executable directory (which can be your project directory or e.g. (projectdir)/lib/architecture/ depending on your Lazarus project settings) (and distributed with your executable) in order to work.

This might also need to be included in you Lazarus IDE directory. See [1] (https://forum.lazarus.freepascal.org/index.php?topic=46118.msg327940#msg327940) And read on especially for TSQLite3Dataset and TSQLiteDataset See below

Most Linux distributions have sqlite3 (eg libsqlite3.so.0) installed by default but Ubuntu distros, at least, also require the matching Dev package. Both should be install via system package manager and marked as a dependency rather than being distributed with your application.

Win64: please see warning here on not using certain FPC/Lazarus Win64 versions.

Direct access to SQLite

You can use an easy way to connect SQLite with Lazarus. Components you are named LiteDAC. SQLite Data Access Components (LiteDAC) is a library of components that provides native connectivity to SQLite from Lazarus (and Free Pascal) on Windows, macOS, iOS, Android, Linux, and FreeBSD for both 32-bit and 64-bit platforms. LiteDAC is designed for programmers to develop truly cross-platform desktop and mobile SQLite database applications with no need to deploy any additional libraries. You can download a trial version of this commercial product at Lazarus components (https://www.devart.com/litedac/download.html).

Built-in SQLDB

FPC/Lazarus offers the built-in SQLDB components that include support for SQLite databases (TSQLite3Connection) from the SQLdb tab of the Component Palette, which allow you to e.g. create GUIs with database components such as TDBGrids. The advantage of using SQLDB is that it is fairly easy to change to a different database such as Firebird or PostgreSQL without changing your program too much. See below for details.

Spatialite support

Spatialite are GIS extensions to SQLite which you can use from within SQLDB. See Spatialite.

Support for SQLite encryption

In recent FPC versions (implemented March 2012), SQLDB included support for some extended versions of SQLite3 which encrypt the SQLite database file using the AES algorithm. Use the password property to set the encryption key.

Examples:

- SQLCipher (http://sqlcipher.net/): open source, e.g. Windows binaries not for free (you have to compile them yourself)
- System.Data.SQLite (http://system.data.sqlite.org/): open source, Windows (32, 64, CE) binaries available, download e.g one of the Precompiled Binaries and rename SQLite.Interop.dll to sqlite3.dll (if you're using the Statically Linked ones, presumably you need to rename System.Data.SQLite.DLL to sqlite3.dll)
- wxSQLite3 (http://wxcode.sourceforge.net/docs/wxsqlite3/): open source, some binaries for Linux available (ex: https://launchpad.net/ubuntu/oneiric/+package/libwxsqlite3-2.8-0)

sqlite3backup

sqlite3backup is a unit provided with FPC (not in Lazarus but can be used programmatically) that provides backup/restore functionality for SQLite3. It uses SQLDB's sqlite3conn.

Zeos

Zeos (https://sourceforge.net/projects/zeoslib/)

SQLitePass

SqlitePass (http://source.online.free.fr/) components. Status: unknown.

TSQLite3Dataset and TSQLiteDataset

There are also separate TSQLiteDataset (unit sqliteds) and TSQLite3Dataset (unit sqlite3ds) packages; see below for a description on how to use them. Visit the sqlite4fpc homepage (http://sqlite4fpc.yolasite.com/) to find the API reference and more tutorials.

TSqliteDataset and TSqlite3Dataset are TDataset descendants that access, respectively, 2.8.x and 3.x.x sqlite databases. For new projects, you would presumably use TSQlite3Dataset as SQLite 3.x is the current version.

Below is a list of the principal advantages and disadvantages compared to other FPC/Lazarus SQLite drivers/access methods:

Advantages:

- Flexible: programmers can choose to use or not to use the SQL language, allowing them to work with simple table layouts or any complex layout that SQL/sqlite allows

Disadvantages:

- Changing to other databases is more difficult than if you use the SQLDB or Zeos components

Note: Given the above, many users will use SQLDB or Zeos due to the advantages unless they need lower-level access to the SQLite library

Using the SQLdb components with SQLite

These instructions are focused on SQLDB (the TSQLite3Connection) specifics for SQLite. For a general overview, have a look at SqlDBHowto which has some useful information about the SQLdb components.

See SQLdb_Tutorial1 for a tutorial on creating a GUI database-enabled program that is written for SQLite/SQLDB (as well as for Firebird/SQLDB, PostgreSQL/SQLDB, basically any RDBMS SQLDB supports).

We will use a combination of three components from the Lazarus SQLdb tab: TSQLite3Connection, TSQLTransaction and TSQLQuery. The TSQLQuery acts as our TDataset; in the simplest case it just represents one of our tables. For the sake of simplicity: make sure you already have an existing SQLite database file and don't need to create a new one now. TSQLite3Connection can be found in the *sqlite3conn* unit, if you want to declare it yourself or are working in FreePascal.

The three components are connected with each other as usual: In the TSQLQuery set the properties Database and Transaction, in the TSQLTransaction set the property Database. There is not much to do in the Transaction and Connection components, most of the interesting things will be done in the TSQLQuery. Configure the components as follows:

TSQLite3Connection:

- DatabaseName: Set this property to the file name (absolute path!) of your SQLite file. Unfortunately, you cannot simply use a relative path that works unchanged at designtime and at runtime ***is this still true? Can't you just copy the db file in a post-build shell script or symlink it?***. You should make sure that at application start the correct path to the file is always set programmatically, no matter what it contained at designtime.

Note: To set the full library path (if you place your sqlite dll/so/dylib in a place where the OS won't find it, like the application directory on Linux/OSX), you can set the *SQLiteLibraryName* property (BEFORE any connection is established e.g. in the OnCreate event of the main form), like this:

```
SQLiteLibraryName:='./sqlite3.so';
```

TSQLQuery:

- SQL: Set it to some simple select query on one of your tables. For example, if you have a table 'foo' and want this dataset to represent this table then just use the following:

```
SELECT * FROM foo
```

- Active: Set this to True from within the IDE to test whether it is all set up correctly. This will also automatically activate the transaction and the connection objects. If you receive an error then either the DatabaseName of the connection is not correct or the SQL query is wrong. Later, when we are done adding the fields (see below) set them all to inactive again, we don't want the IDE to lock the SQLite database (single user!) when testing the application.

- *Probably not necessary for proper operation - will need to be checked (June 2012)* Now we can add Fields to our TSQLQuery. While the components are still set to active do a right click and "edit fields...". Click the "+" button and add fields. It will list all fields your SQL query returned. Add every field you will need, you can also add lookup fields here; in this case just make sure you have already defined all needed fields in the other datasets before you start adding lookup fields that refer to them. If your table has many columns and you don't need them all you can just leave them out, you can also make your SQL a bit more specific.

- In your code you need to call SQLQuery.ApplyUpdates and SQLTransaction.Commit, TSQLQuery.AfterPost and AfterInsert events are a good place for this when using it with data aware controls but of course you can also postpone these calls to a later time. If you don't call them, the database will not be updated.

- "Database is locked": The IDE might still be locking the database (SQLite is a single user database), you probably forgot to set the components to inactive and disconnected again after you were done defining all the fields of your TSQLQuery objects. Use the Form's OnCreate event to set the path and activate the objects at runtime only. Most of the things you set in the TSQLQuery from within the IDE don't require (and some don't even allow) them to be active at design time, the only exception is defining the fields where it wants to read the table design, so inactive at design time should be the normal state.

- Your tables should all have a primary key and you must make sure that the corresponding field has pfInKey and nothing else in its PoviderFlags (these flags control how and where the field is used when automatically constructing the update and delete queries).

- If you are using lookup fields
  - make sure the ProviderFlags for the lookup field is completely empty so it won't attempt to use its name in an update query. The lookup field itself is not a data field, it only acts on the value of another field, the corresponding key field, and only this key field will later be used in the update queries. You can set the key field to hidden because usually you don't want to see it in your DBGrid but it needs to be defined.
  - LookupCache must be set to True. At the time of this writing for some reason the lookup field will not display anything otherwise (but still work) and strangely the exact opposite is the case when working with the TSQLite3Dataset or other TXXXDataset components, here it must be set to False. I'm not yet sure whether this is intended behavior or a bug.

- Usually with simple tables you won't need to set any of the InsertSQL, UpdateSQL and DeleteSQL properties, just leave them empty. If you have the ProviderFlags of all your fields set correctly it should be able to create the needed SQL on the fly. For more details on InsertSQL, UpdateSQL and DeleteSQL, see Working_With_TSQLQuery#TSQLQuery.InsertSQL.2C_TSQLQuery.UpdateSQL_and_TSQLQuery.Dele

After the above is all set up correctly, you should now be able to use the TSQLQuery like any other TDataset, either by manipulating its data programmatically or by placing a TDatasouce on the Form, connecting it to the TSQLQuery and then using data contols like TDBGrid etc.

Creating a Database

The TSQLite3Connection.CreateDB (http://www.freepascal.org/docs-html/fcl/sqldb/tsqlconnection.createdb.html) method inherited from the parent class actually does nothing; to create a database if no file exists yet, you simply have to write table data as in the following example:

(Code extracted from sqlite_encryption_pragma example that ships with Lazarus 1.3 onwards)

```
var
  newFile : Boolean;
begin

  SQLite3Connection1.Close; // Ensure the connection is closed when we start

  try
    // Since we're making this database for the first time,
    // check whether the file already exists
    newFile := not FileExists(SQLite3Connection1.DatabaseName);

    if newFile then
    begin
      // Create the database and the tables
      try
        SQLite3Connection1.Open;
        SQLTransaction1.Active := true;

        // Here we're setting up a table named "DATA" in the new database
        SQLite3Connection1.ExecuteDirect('CREATE TABLE "DATA"('+
                    ' "id" Integer NOT NULL PRIMARY KEY AUTOINCREMENT,'+
                    ' "Current_Time" DateTime NOT NULL,'+
                    ' "User_Name" Char(128) NOT NULL,'+
                    ' "Info" Char(128) NOT NULL);');

        // Creating an index based upon id in the DATA Table
        SQLite3Connection1.ExecuteDirect('CREATE UNIQUE INDEX "Data_id_idx" ON "DATA"( "id" );');
```

```
      SQLTransaction1.Commit;

      ShowMessage('Succesfully created database.');
    except
      ShowMessage('Unable to Create new Database');
    end;
  end;
 except
  ShowMessage('Unable to check if database file exists');
 end;
end;
```

Creating user defined collations

```
// utf8 case-sensitive compare callback function
function UTF8xCompare(user: pointer; len1: longint; data1: pointer; len2: longint; data2: pointer): longint;
cdecl;
var S1, S2: AnsiString;
begin
  SetString(S1, data1, len1);
  SetString(S2, data2, len2);
  Result := UnicodeCompareStr(UTF8Decode(S1), UTF8Decode(S2));
end;

// utf8 case-insensitive compare callback function
function UTF8xCompare_CI(user: pointer; len1: longint; data1: pointer; len2: longint; data2: pointer): longint;
cdecl;
var S1, S2: AnsiString;
begin
  SetString(S1, data1, len1);
  SetString(S2, data2, len2);
  Result := UnicodeCompareText(UTF8Decode(S1), UTF8Decode(S2));
end;

// register collation using SQLite3 API (requires sqlite3dyn unit):
sqlite3_create_collation(SQLite3.Handle, 'UTF8_CI', SQLITE_UTF8, nil, @UTF8xCompare_CI);
// or using method of TSQLite3Connection:
CreateCollation('UTF8_CI',1,nil,@UTF8xCompare_CI);

// now we can use case-insensitive comparison in SQL like:
// SELECT * FORM table1 WHERE column1 COLLATE UTF8_CI = 'á'

// but this does not work for LIKE operator
// in order to support also LIKE operator we must overload default LIKE function using sqlite3_create_function()
// http://www.sqlite.org/lang_corefunc.html#like
```

Creating user defined functions

```
// example overloading default LOWER() function with user supplied function
// to run this demo, you must add units 'sqlite3dyn' and 'ctypes' to your uses-clause
// and add a const 'SQLITE_DETERMINISTIC' with value $800

procedure UTF8xLower(ctx: psqlite3_context; N: cint; V: ppsqlite3_value); cdecl;
var S: AnsiString;
begin
  SetString(S, sqlite3_value_text(V[0]), sqlite3_value_bytes(V[0]));
  S := UTF8Encode(AnsiLowerCase(UTF8Decode(S)));
  sqlite3_result_text(ctx, PAnsiChar(S), Length(S), sqlite3_destructor_type(SQLITE_TRANSIENT));
end;

// register function LOWER() using SQLite3 API (requires sqlite3dyn unit):
sqlite3_create_function(SQLite3.Handle, 'lower', 1, SQLITE_UTF8 or SQLITE_DETERMINISTIC, nil, @UTF8xLower, nil,
nil);
```

SQLite3 and Dates

- SQLite 3 doesn't store dates as a special DateTime value. It can stores them as strings, doubles or integers

- see http://www.sqlite.org/datatype3.html#datetime.
- In strings, the date separator is '-' as per SQL standard/ISO 8601. Thus, if you do an INSERT using the built-in DATE function, it will store it as something like 'YYYY-MM-DD'.
- Reading a DateTime value can cause problems for DataSets if they are stored as strings: the .AsDateTime qualifier can stall on an SQLite 'string date' but this can be overcome by using something like `strftime(%d/%m/%Y,recdate) AS sqlite3recdate` in your SQL SELECT statement, which forces SQLite3 to return the date record in a specified format. (the format string %d/%m/%d corresponds to your locale date format which .AsDateTime will understand) **==> Please open a bug report with an example application demonstrating the problemif this is the case**
- When comparing dates stored as strings (using for example the BETWEEN function) remember that the comparison will always be a **string** comparison, and will therefore depend on how you have stored the date value.

Default values in local time instead of UTC

CURRENT_TIME, CURRENT_DATE and CURRENT_TIMESTAMP return current UTC date and/or time. For local date and/or times we can use:

```
DEFAULT (datetime('now','localtime')) for datetime values formated YYYY-MM-DD HH:MM:SS
DEFAULT (date('now','localtime')) for date value formated YYYY-MM-DD
DEFAULT (time('now','localtime')) for time value formated HH:MM:SS
```

SQLDB And SQLite troubleshooting

- Keep in mind that for designtime support to work (fields etc) Lazarus must find sqlite3.dll too.
- The same goes for the database filename. Always use absolute path if you use components to extract e.g. fieldnames at designtime. Otherwise the IDE will create an empty file in its directory. In case of trouble, check if the lazarus/ directory doesn't hold a zero byte copy of the database file.
- If you have master/detail relationship, you need to refresh master dataset after each insert, in order to get value for slave dataset foreign key field. You can do that in AfterPost event of the master dataset, by calling one of the following overloaded procedures:

```
interface
    procedure RefreshADatasetAfterInsert(pDataSet: TSQLQuery);overload;
    procedure RefreshADatasetAfterInsert(pDataSet: TSQLQuery; pKeyField: string);overload;

implementation

procedure RefreshADatasetAfterInsert(pDataSet: TSQLQuery; pKeyField: string);
//This procedure refreshes a dataset and positions cursor to last record
//To be used if Dataset is not guaranteed to be sorted by an autoincrement primary key
var
  vLastID: Integer;
  vUpdateStatus : TUpdateStatus;
begin
  vUpdateStatus := pDataset.UpdateStatus;
  //Get last inserted ID in the database
  pDataset.ApplyUpdates;
  vLastID:=(pDataSet.DataBase as TSQLite3Connection).GetInsertID;
  //Now come back to respective row
  if vUpdateStatus = usInserted then begin
    pDataset.Refresh;
    //Refresh and go back to respective row
    pDataset.Locate(pKeyField,vLastID,[]);
  end;
end;

procedure RefreshADatasetAfterInsert(pDataSet: TSQLQuery);
//This procedure refreshes a dataset and positions cursor to last record
```

```
//To be used only if DataSet is guaranteed to be sorted by an autoincrement primary key
var
  vLastID: Integer;
  vUpdateStatus : TUpdateStatus;
begin
  vUpdateStatus := pDataset.UpdateStatus;
  pDataset.ApplyUpdates;
  vLastID:=(pDataSet.DataBase as TSQLite3Connection).GetInsertID;
  if vUpdateStatus = usInserted then begin
    pDataset.Refresh;
    //Dangerous!
    pDataSet.Last;
  end;
end;

procedure TDataModule1.SQLQuery1AfterPost(DataSet: TDataSet);
begin
  RefreshADatasetAfterInsert(Dataset as TSQLQuery); //If your dataset is sorted by primary key
end;

procedure TDataModule1.SQLQuery2AfterPost(DataSet: TDataSet);
begin
  RefreshADatasetAfterInsert(Dataset as TSQLQuery, 'ID'); //if you are not sure that the dataset is always
sorted by primary key
end;
```

Vacuum and other operations that must be done outside a transaction

SQLDB seems to always require a connection, but some operations like Pragma and Vacuum must be done outside a transaction. The trick is to end transaction, execute what you must and start transaction again (so that sqldb doesn't get confused:)

```
  // commit any pending operations or use a "fresh" sqlconnection
  Conn.ExecuteDirect('End Transaction');  // End the transaction started by SQLdb
  Conn.ExecuteDirect('Vacuum');
  Conn.ExecuteDirect('Begin Transaction'); //Start a transaction for SQLdb to use
```

Using TSQLite3Dataset

This section details how to use the TSQLite2Dataset and TSQLite3Dataset components to access SQlite databases. by Luiz Américo luizmed(at)oi(dot)com(dot)br

Requirements

- For sqlite2 databases (legacy):
  - FPC 2.0.0 or higher
  - Lazarus 0.9.10 or higher
  - SQLite runtime library 2.8.15 or above*
- Sqlite2 is not maintained anymore and the binary file cannot be found in the sqlite site
- For sqlite3 databases:
  - FPC 2.0.2 or higher
  - Lazarus 0.9.11 (svn revision 8443) or higher
  - sqlite runtime library 3.2.1 or higer (get it from www.sqlite.org (http://www.sqlite.org))

Before initiating a lazarus project, ensure that:

- the sqlite library is either
  - in the system PATH or
  - in the executable output directory and Lazarus (or current project) directories - this option might work on Windows only
- under Linux, put cmem as the first unit in uses clause of the main program
  - In Debian, Ubuntu and other Debian-like distros, in order to build Lazarus IDE you must install the packages libsqlite-dev/libsqlite3-dev, not only sqlite/sqlite3 (Also applies to OpenSuSe)

How To Use (Basic Usage)

Install the package found at /components/sqlite directory (see instructions here)

At design time, set the following properties:

- FileName: path of the sqlite file [required]
- TableName: name of the table used in the sql statement [required]
- SQL: a SQL select statement [optional]
- SaveOnClose: The default value is false, which means that changes are not saved. One can change it to true. [optional]
- Active: Needs to be set at design time or at program startup. [required]

Creating a Table (Dataset)

Double-click the component icon or use the 'Create Table' item of the popup menu that appears when clicking the right mouse button. A simple self-explaining table editor will be shown.

Here are all field types supported by TSqliteDataset and TSqlite3Dataset:

- Integer
- AutoInc
- String
- Memo
- Bool
- Float
- Word
- DateTime
- Date
- Time
- LargeInt
- Currency

Retrieving the data

After creating the table or with a previously created Table, open the dataset with the Open method. If the SQL property was not set then all records from all fields will be retrieved, the same if you set the SQL to:

```
SQL := 'Select * from TABLENAME';
```

Applying changes to the underlying datafile

To use the ApplyUpdates function, the dataset must contain at least one field that fulfills the requirements for a

Primary Key (values must be UNIQUE and not NULL)

It's possible to do that in two ways:

- Set PrimaryKey property to the name of a Primary Key field
- Add an AutoInc field (This is easier since the TSqliteDataSet automatically handles it as a Primary Key)

If one of the two conditions is set, just call

```
ApplyUpdates;
```

Note: If both conditions are set, the field corresponding to PrimaryKey is used to apply the updates.

Note: Setting PrimaryKey to a field that is not a Primary Key will lead to loss of data if ApplyUpdates is called, so ensure that the chosen field contains not Null and Unique values before using it.

Master/detail example

Various examples of master/detail relations (e.g. the relation between customer and orders):

- the generic SQLDB way of doing this using the `DataSource` property: MasterDetail
- TSQLite3 specific example using locate: TSqlite3 Master Detail Example.

Remarks

- Although it has been tested with 10,000 records and worked fine, TSqliteDataset keeps all the data in memory, so remember to retrieve only the necessary data (especially with Memo Fields).
- The same datafile (Filename property) can host several tables/datasets
- Several datasets (different combinations of fields) can be created using the same table simultaneously
- It's possible to filter the data using WHERE statements in the sql, closing and reopening the dataset (or calling RefetchData method). But in this case, the order and number of fields must remain the same
- It's also possible to use complex SQL statements using aliases, joins, views in multiple tables (remember that they must reside in the same datafile), but in this case ApplyUpdates won't work. If someone wants to use complex queries and to apply the updates to the datafile, mail me and i will give some hints how to do that
- Setting filename to a sqlite datafile not created by TSqliteDataset and opening it is allowed but some fields won't have the correct field type detected. These will be treated as string fields.

Generic examples can be found at fpc/fcl-db/src/sqlite SVN directory

See also

- SQLdb_Tutorial1 Tutorial for Sqlite
- SqlDBHowto General information on SQLDB
- Working With TSQLQuery Information on SQLDB's TSQLQuery
- SQLite site (http://www.sqlite.org)

Retrieved from "https://wiki.freepascal.org/index.php?title=SQLite&oldid=129813"
