Qt basic SQLite functions

CPP 11 & Qt 4.11.1

IN THE NAME OF GOD

With this, you will no longer need to use basic SQL code for your applications. SQL codes have been translated to Qt C++ code and are now available here

Amirreza Nadi

Specialized in SQLite

Contents

1. Introduction

This is a Qt C++ based module that gives you access to SQL (specialized in SQLite). SQL codes are translated to C++ and you can use them to store and read data in SQL format. Functions are all written in C++, allowing you to use them without knowing how to code in SQL.

The module contains a class consisting of only static functions, allowing you to use them without need for an object of that class.

If you take a look at the contents of the library file (sqlhandle.h), you will spot the word "Command" or "CommandList" in every function prototype[1]. The ones that have "Command" at the end will return a QString value and the ones with "CommandList", will return a QStringList. Functions with QString return type need to be executed through a query[2] and functions with QStringList return type need a loop to execute each member of the list separately.

CAUTION: datatypes of SQL are different from C++. Here, we use TEXT for strings, VCHAR(number) for char arrays, INTEGER for integer numbers, FLOAT for float and DOUBLE for double.

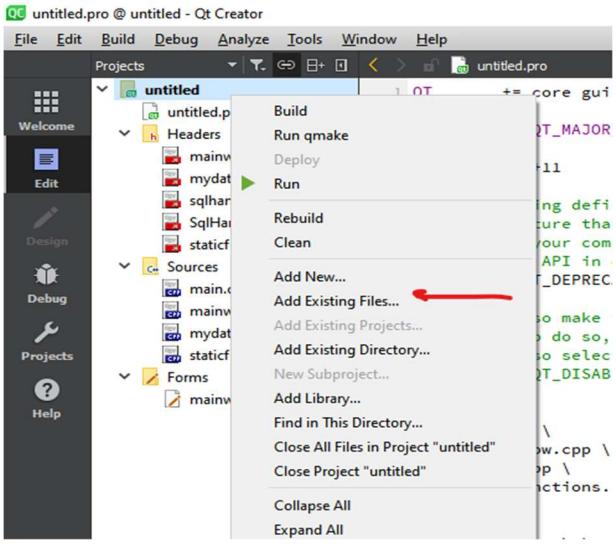
In this document, there is an explanation for every function, along with an example of how to use them. If you still need help with the module, you can contact me through this E-Mail: amirrezanadichaghadari@gmail.com

[1] https://en.wikipedia.org/wiki/Function_prototype

[2] Explained in part 5: How to execute functions

2. How to Attach Modules to Your Project

1. Copy header files (sqlhandle.h and SqlHandle_global.h) and paste them in your project folder (same folder that contains your .pro file)
2. Add header files to your project by either these ways:
   2.1. Right click on the folder of your project inside Qt and choose “Add Existing Files…”



   2.2. Go to your .pro file and add their names to list of headers

```
SOURCES += \
    main.cpp \
    staticfunctions.cpp

HEADERS += \
    sqlhandle.h \
    SqlHandle_global.h
```

3. Extract SqlHandle.zip and open the folder that suits your chosen building kit best.
4. Copy .dll file, .o file, and .a file and paste them all in your build directory, which is the parent directory of the folder containing your built .exe file.
5. Return to your .pro file and write LIBS += SqlHandle.dll

You’re all set now.

3. List of Functions

name	usage
CreateTableCommand	Creating a new table
AddRecordCommand	Adding a row to a table
DeleteRecordCommand	Deleting a row from a table
DeleteRecordCommandLikeOperator	Deleting rows containing a similar content in one field
EditRecordCommand	Editing a row
EditRecordCommandLikeOperator	Editing rows containing a similar content in one field
DropColumnCommandList	Removing a column
RenameColumnCommandList	Renaming a column
AddColumnCommand	Adding a column
RenameTableCommand	Renaming a table
DropTableCommand	Deleting a whole table
ReorderTableCommandList	Changing order of a table's columns

4. About “like” Operator in SQL

In SQL, like is an operator used for comparison. Here’s an example of how it works:

Imagine we have a list of names like this:

James
Robert
John
Michael
David
William
Charles

We can use “like” operator for 3 purposes.

1. Finding all elements that start with a specific phrase. For example, if we use it to find all names starting with “j”, we will get “James” and “John”
2. Finding all elements that end with a specific phrase. For example, if we use it to find all names ending with “es”, we will get “Charles” and “James”
3. Finding all element that contain a special phrase. For example, if we use it to find all names containing “ch”, we will get “Michael” and “Charles”.

You can find more explanation on the web if you’re interested.

5. How to Execute Functions

In order to execute and use functions of this module, you'll need to connect to a SQL file or server first. Here's how to do it in SQLite:

1. Create an object with type "QSqlDatabase"
2. Set it equal to this phrase: "QSqlDatabase::addDatabase("QSQLITE")"
3. Try opening it by using " .open " function.

Here's an example of a function doing all that:

```
bool MainWindow::CreateConnection()
{
    data = QSqlDatabase::addDatabase("QSQLITE");
    data.setDatabaseName("test.qsqlite");
    if(!data.open())
    {
        qDebug()<<"Failed to open file ...   "<<data.lastError();
        return false;
    }
    return true;
}
```

Where data is an object of that type.

After that, you'll need a query. I recommend using dynamic memory allocation in this case. I've defined my object like this: QSqlQuery* qu;
and allocated it this way: qu = new QSqlQuery(data);
and data has already been defined and connected.

"qu" is a variable of QSqlQuery type. This type has a function called "exec" which can take a QString as an argument and execute that string as a SQL command. In order to use this module, you can use its functions as the argument for "exec" function of "qu" like this:

qu->exec(SqlHandle::function_name(arguments…))

the exec function has Boolean return type. It's set true if the operation is done successfully. Therefore, you can use it in collaboration with an if clause like this:

if(qu->exec(SqlHandle::function_name(arguments…)))

or:

if(!qu->exec(SqlHandle::function_name(arguments…)))

6. How to Use Each Function

6.1. CreateTableCommand:

```
bool MainWindow::CreateTable()
{
    if(!qu->exec(SqlHandle::CreateTableCommand("testing",SeNames,SeTypes)))
    {
        qDebug()<<"Failed to create table...    "<<qu->lastError();
        return false;
    }
    return true;
}
```

Where SeNames is a QStringList containing lable of each column and SeTypes is a QStringList containing each column's datatype. "testing" is the name of the table. You can choose any name you want, just no spaces.

6.2. AddRecordCommand:

```
bool MainWindow::addrec(QStringList vls)
{
    if(!qu->exec(SqlHandle::AddRecordCommand("testing",SeNames,SeTypes,vls)))
    {
        qDebug()<<"couldn't add ...  "<<qu->lastError();
        return false;
    }
    return true;
}
```

vls is the list of values you want to fill fields with. It has to be in the same order as the SeNames and SeTypes.

6.3. DeleteRecordCommand:

```
bool MainWindow::deleterec(QString primaryName, QString primaryType, QString primaryValue)
{
    if(!qu->exec(SqlHandle::DeleteRecordCommand("testing",primaryName,primaryType,primaryValue)))
    {
        qDebug()<<"falied to delete ...  "<<qu->lastError();
        return false;
    }
    return true;
}
```

When you want to delete a field, you'll need to find it first based on the value of one of its records. primaryName is the lable of the column that the mentioned

record is in. primaryType is the same column's datatype. Finally, primaryValue is the value of the mentioned record. "testing" is the name of the table as mentioned before.

6.4. DeleteRecordCommandLikeOperator:

```
bool MainWindow::deleterec_likeop(QString primaryName, QString primaryType, QString primaryValue
                                  , SqlHandle::OperatorLikePlace place)
{
    if(!qu->exec(SqlHandle::DeleteRecordCommandLikeOperator("testing",primaryName,primaryType,primaryValue,place)))
    {
        qDebug()<<"falied to delete likeOP ...  "<<qu->lastError();
        return false;
    }
    return true;
}
```

This function does the same job as the one before, but with a different method of finding records. Instead of searching for a record by having their exact values, you can search for them by a part of them as explained in part 4 of this document.

"place" is an enum variable that can have 3 values: Begining, Middle, and End. By giving Beginning as the value, the search will be like the first condition explained in part 4. End will be like the second one and Middle will be like the third one.

6.5. EditRecordCommand:

```
bool MainWindow::edit(QString pk, QStringList vls)
{
    if(!qu->exec(SqlHandle::EditRecordCommand("testing",SeNames[0],SeTypes[0],pk,SeNames,SeTypes,vls)))
    {
        qDebug()<<"falied to edit ...  "<<qu->lastError();
        return false;
    }
    return true;
}
```

This edits the record which has the same value as argument pk in the column labeled as SeNames[0].

In this example, I've put the primary key in the list of columns as the first value. That's why I've used SeNames[0] as argument for primary name. same goes for primary type.

6.6. EditRecordCommandLikeOperator:

```
bool MainWindow::editlikeOP(QString primaryName, QString primaryType, QString primaryValue,
                            QStringList vls, SqlHandle::OperatorLikePlace place)
{
    if(!qu->exec(SqlHandle::EditRecordCommandLikeOperator("testing",primaryName,primaryType,primaryValue,SeNames
                                                          ,SeTypes,vls,place)))
    {
        qDebug()<<"falied to edit ...  "<<qu->lastError();
        return false;
    }
    return true;
}
```

6.7. DropColumnCommandList:

```
QStringList commandList = SqlHandle::DropColumnCommandList("testing",SeNames,SeTypes,3);
qDebug()<<commandList;
for(int i=0;i<commandList.length();i++)
{
    qu->exec(commandList[i]);
}
```

As mentioned before, return type of this function is QStringList. So, you'll need a loop to execute each using the query. It's recommended to code a function that sends strings to the query and can handle errors instead of what I've done, which is directly using qu->exec in the loop.

6.8. RenameColumnCommandList:

```
QStringList commandList = SqlHandle::DropColumnCommandList("testing",SeNames,SeTypes,3,"newName");
qDebug()<<commandList;
for(int i=0;i<commandList.length();i++)
{
    qu->exec(commandList[i]);
}
```

6.9. AddColumnCommand:

```
qu->exec(SqlHandle::AddColumnCommand("testing","Hi","TEXT"));
```

6.10. **RenameTableCommand:** It is a very simple function with simple arguments and output. No need for examples.

6.11. **DropTableCommand:** It is a very simple function with simple arguments and output. No need for examples.

6.12. ReorderTableCommandList:

```
QStringList l1 = SeNames,l2 = SeTypes;

l1.swapItemsAt(2,3);
l2.swapItemsAt(2,3);

QStringList commandList = SqlHandle::ReorderTableCommandList("testing",SeNames,SeTypes,l1,l2);

for(int i=0;i<commandList.length();i++)
{
    qu->exec(commandList[i]);
}
```

Here, I've first set l1 and l2 to be the same as SeNames, SeTypes. Then I've changed its order and, in the end, I've saved list of commands in commandList. It's recommended to do the same: setting 2 lists equal to originals and then changing their order however you need. This way, you're less likely to make a mistake typing the column labels and types.

The End …