

Лекция №5. Qt and SQL

Автор: Набиев Марат

В 3-ей лекции мы пробовали сделать MVP проект, но данные хранили просто в списках, однако в 4-ой лекции мы поработали с SQL и это показалось довольно удобным. Таким образом, было бы удобно хранить данные в базе данных и обращаться к ним через наше приложение. Также после закрытия программы, эти данные сохраняются, т. к. база данных хранится на жестком диске. Итак, попробуем сделать это! (Здесь нужны исходники с 3 лекции!)

Qt может работать со следующими БД:

- QDB2 — IBM DB2 (версия 7.1 и выше)
- QIBASE — Borland InterBase
- QMYSQL — MySQL
- QOCI — Драйвер Oracle Call Interface
- QODBC — Open Database Connectivity (ODBC) — Microsoft SQL Server и другие ODBC-совместимые базы данных
- QPSQL — PostgreSQL (версия 7.3 и выше)
- QSQLITE2 — SQLite версии 2
- QSQLITE — SQLite версии 3
- QTDS — Драйвер Sybase Adaptive Server

Подключение QtSQL

Что же нужно, чтобы смочь работать с sql? Для начала откроем наш MVP проект. Наши изменения коснутся только модуля Model, т. к. Presenter и View никак не влияют на то, каком образом хранятся данные. Откроем `Model.pro` и допишем, что будем работать с sql.

```

1 #-----
2 #
3 # Project created by QtCreator 2018-04-
4 #
5 #-----
6
7 QT      -= gui
8 QT += sql
9 TARGET = Model
10 TEMPLATE = lib
11 CONFIG += staticlib
12

```

Эту строчку также надо добавить в `Presenter.pro`, иначе не будет работать.

Далее создадим класс, который будет работать с БД и назовем его `DataBase`.

(создаем как просто класс C++ и ни от чего не будем наследовать)

The screenshot shows the 'Define Class' dialog in Qt Creator for a C++ class. The dialog has a sidebar with 'Подробнее' (More) and 'Итог' (Summary) tabs. The 'Итог' tab is selected, showing the following configuration:

- Имя класса:** `DataBase`
- Базовый класс:** `<Особый>`
- Подключить:**
 - ☐ Подключить QObject
 - ☐ Подключить QWidget
 - ☐ Подключить QMainWindow
 - ☐ Подключить QDeclarativeItem - Qt Quick 1
 - ☐ Подключить QQuickItem - Qt Quick 2
 - ☐ Подключить QSharedData
- Заголовочный файл:** `database.h`
- Файл исходных текстов:** `database.cpp`
- Путь:** `/home/marat/files/qt/less3/Model`

At the bottom right, there are buttons for 'Далее >' (Next) and 'Отмена' (Cancel).

Сделаем класс DataBase классом-одиночкой. Т.е. можно только создать 1 экземпляр этого класса.

Шаблон проектирования «Одиночка»

Одиночка (Singleton) — шаблон проектирования, гарантирующий, что в приложении будет только 1 экземпляр этого класса и предоставляет глобальную точку доступа к этому экземпляру.

Зачем это нужно? Чтобы в нашем приложении был создано только 1 соединение с базой данных. Т.к. если все будут создавать экземпляр DataBase, только у 1 будет соединение, а остальные, кому еще нужен доступ, его не смогут получить, а так, все объекты приложения будут иметь доступ.

Как обеспечивается такой доступ?

В классе есть *статическая* ссылка на этот же класс, и все конструкторы закрыты. Для получения экземпляра класса реализуется *статический* метод getInstance, который дает экземпляр класса.

Сделаем это!

Так будет выглядеть database.h

```
1  #ifndef DATABASE_H
2  #define DATABASE_H
3
4  //это для соединения с бд
5  #include <QtSql>
6  class DataBase
7  {
8  private:
9      //для доступа к бд
10     QSqlDatabase base;
11     //статическая ссылка на себя
12     static DataBase *instance;
13     //закрытый конструктор
14     DataBase();
15 public:
16     //статический метод getInstance
17     static DataBase* getInstance();
18     //для добавления в базу данных
19     void addPerson(QString name);
20     //для извлечения из базы данных
21     QList<QString> getPersons();
22 };
23
24 #endif // DATABASE_H
25
```

Посмотрим как выглядит database.cpp

```
1  #include "database.h"
2  #include <QList>
3  #include <QString>
4  //в с++ статические поля инициализируются вне класса
5  DataBase *DataBase::instance = NULL;
6  DataBase::DataBase()
7  {
8  }
9  }
10
11 DataBase* DataBase::getInstance()
12 {
13     //если нет ни одного экземпляра, то создаем
14     if(instance==NULL)
15     {
16         instance = new DataBase;
17     }
18     //возвращаем ссылку на экземпляр
19     return instance;
20 }
21
22 void DataBase::addPerson(QString name)
23 {
24     //пока без реализации
25 }
26 QList<QString> DataBase::getPersons()
27 {
28     //TODO
29     return QList<QString>();
30 }
31
```

Теперь перейдем к работе с базой данных. Просто смотрите код.

```
6  DataBase::DataBase()
7  {
8      //говорим, что хотим добавить базу данных SQLite v3
9      base = QSqlDatabase::addDatabase("QSQLITE");
10     //называем нашу db
11     base.setDatabaseName("my_db.sqlite");
12     //открываем
13     base.open();
14 }
```

Вроде все понятно. Если непонятно, то пишите.

Теперь надо Model научить работать с DataBase. Так выглядит model.h

```

4  #include "imodel.h"
5  #include "database.h"
6  class Model: public IModel
7  {
8  private:
9      //наше хранилище имен
10     //QList<QString> names;
11     //вместо хранилища имен
12     DataBase *base;
13 public:
14     Model();
15     void addPerson(QString name) override;
16     QList<QString> getPersons() const;
17 };
18
19 #endif // MODEL_H
20

```

Следующим образом меняется model.cpp

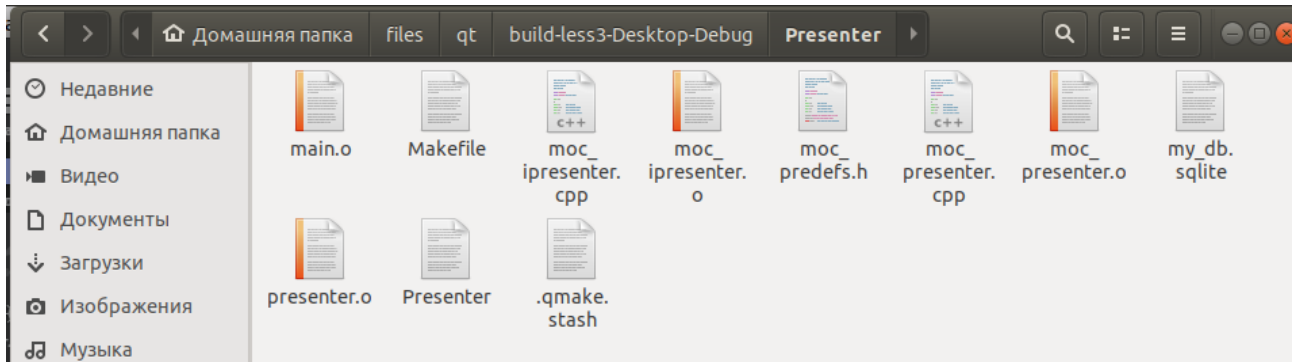
```

1  #include "model.h"
2
3
4  Model::Model()
5  {
6      //получаем экземпляр
7      base = DataBase::getInstance();
8  }
9  void Model::addPerson(QString name)
10 {
11     //добавляем в конец имя
12     //names.append(name);
13     //вместо добавления в список
14     base->addPerson(name);
15 }
16 QList<QString> Model::getPersons() const
17 {
18     //просто вернем список
19     //return names;
20     //вместо возвращения списка
21     return base->getPersons();
22 }
23

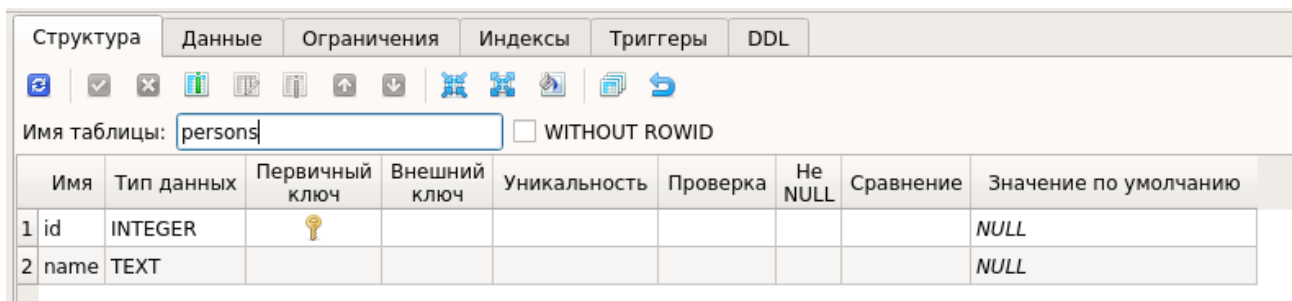
```

Попробуем запустить наш проект (если получится конечно же :D)

Если все хорошо, то в папке, где лежит собранный проект, а именно где Presenter, там должен появиться файл нашей БД `my_db.sqlite`.



Откроем его при помощи SqliteStudio и создадим там 1 таблицу `persons` с 2 столбцами: `id` и `name`. (Вы же умеете это делать)



Теперь надо научить базу данных добавлять в нашу таблицу данные.

Для этого откроем `DataBase::addPerson`

Посмотрим первый способ выполнения запросов.

Напишем строку-- `insert` - запрос.

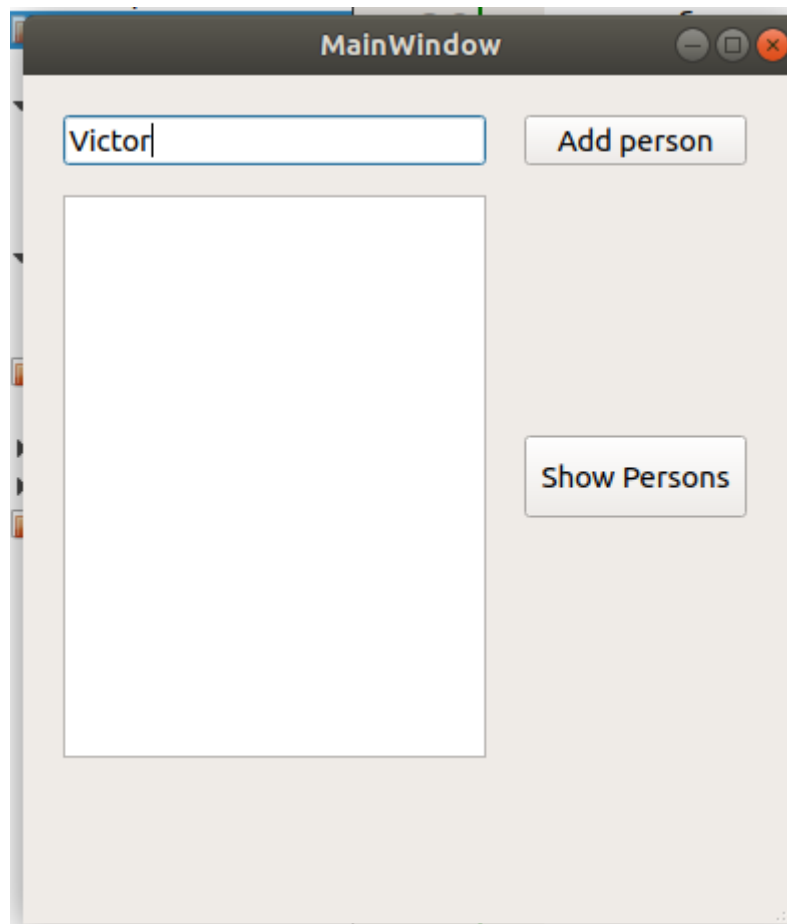
```

27 void DataBase::addPerson(QString name)
28 {
29     //запрос
30     QString insertStr = "INSERT INTO persons(name)"
31                         "VALUES('%1')";
32     //заменяем %1 на имя
33     insertStr = insertStr.arg(name);
34     //создаем QSqlQuery
35     QSqlQuery query;
36     //подготавливаем запрос
37     query.prepare(insertStr);
38     //выполняем его
39     query.exec();
40 }

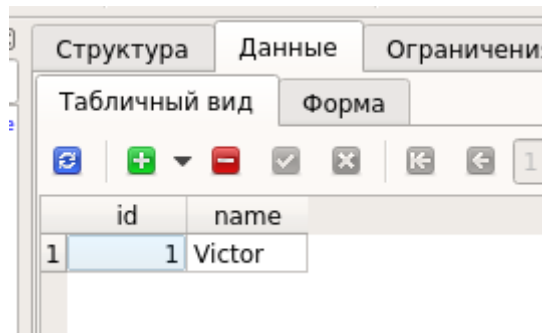
```

Здесь `QSqlQuery` — это особый класс Qt, который предоставляет интерфейс для работы с базами данных. А именно: выполнение запросов и навигацию по результату запроса.

Теперь попробуем собрать наш проект и добавить что-нибудь.



И вот в нашей БД лежат данные!



Теперь рассмотрим другой способ добавления.

Как по мне, так этот способ более наглядный нежели первый

```
27 void DataBase::addPerson(QString name)
28 {
29     //запрос
30     //QString insertStr = "INSERT INTO persons(name)"
31     //                                "VALUES('%1')";
32     //заменяем %1 на имя
33     //insertStr = insertStr.arg(name);
34     //создаем SqlQuery
35     QSqlQuery query;
36     //подготавливаем запрос
37     query.prepare("INSERT INTO persons(name)"
38                  "VALUES(:name)");
39     //пишем что заменяем и на что заменяем
40     query.bindValue(":name", name);
41     //выполняем его
42     query.exec();
43 }
```

При помощи `bindValue` мы заменяем наши метки на настоящие значения (обратите внимания, что когда мы писали `%1` там мы добавили одинарные кавычки для строки, а здесь это уже не нужно)

И если попробуем добавить что-нибудь, то увидим, что это работает

Структура		
Данные		
Оформление		
Табличный вид		
Форма		
id		
name		
1	1	Victor
2	2	Marat

Теперь надо получать данные.

Напишем SELECT в DataBase::getPersons

```

44  ▾ QList<QString> DataBase::getPersons()
45  {
46      QList<QString> names;
47
48      QSqlQuery selectQuery;
49      //нам нужен только столбец имен
50      selectQuery.prepare("SELECT name FROM persons;");
51      //выполнили
52      selectQuery.exec();
53      //пока есть данные
54  ▾ while(selectQuery.next())
55      {
56          // получаем имя с нужного столбца и приводим к строке
57          QString name = selectQuery.value("name").toString();
58          names.append(name);
59      }
60  return names;

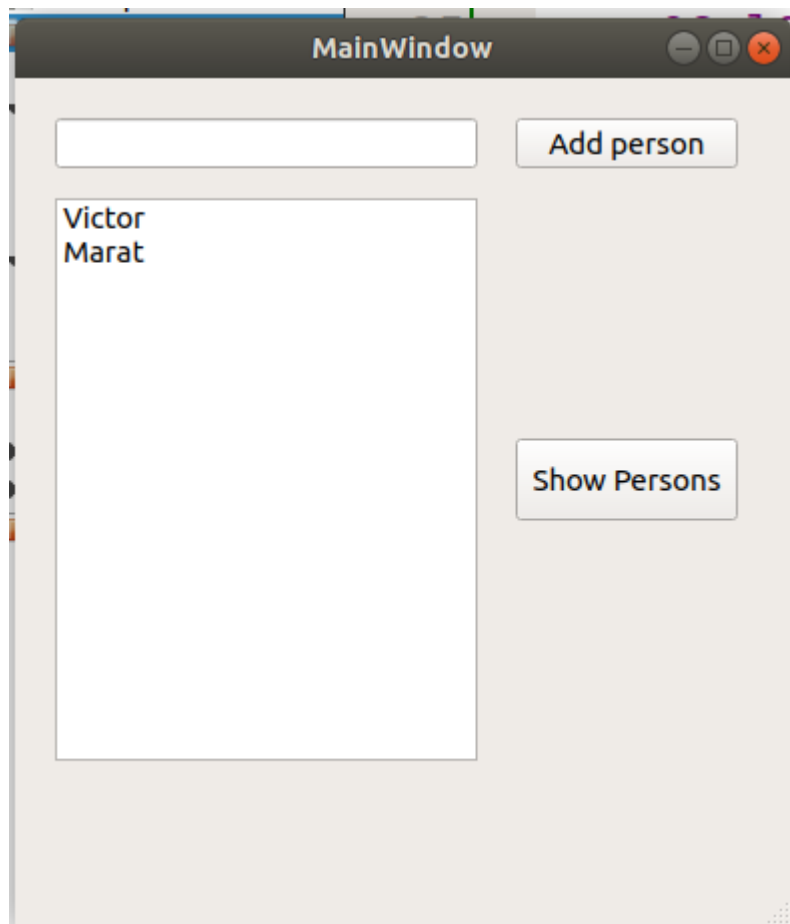
```

Таким образом у нас будет выглядеть SELECT .

По сути, selectQuery указывает нам на строку, и при помощи метода next мы пробегаемся по всем строкам и берем данные из столбца name, который имеет тип QVariant (особый тип Qt, т. к. C++ строго типизирован) и приводим его к строке.

И все данные мы кладем в наш список, который вернем в качестве результата.

Если мы запустим, и нажмем Show Persons, то увидим наши данные.

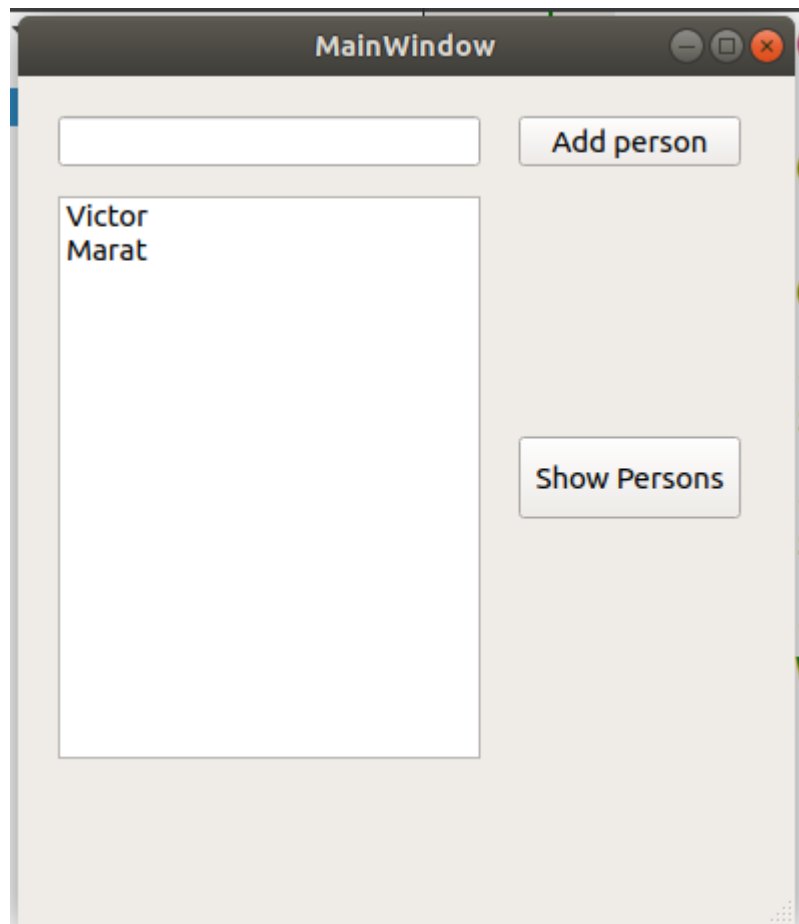


Еще один способ — это через модели.

```
61     QList<QString> names;  
62     QSqlQueryModel model;  
63     model.setQuery("SELECT name FROM persons;");  
64     for(int i=0; i<model.rowCount(); i++)  
65     {  
66         QString name = model.record(i).value("name").toString();  
67         names.append(name);  
68     }  
69     return names;  
70
```

Думаю все понятно. Выполняем запрос, пробегаемся по всем строкам и добавляем в список данные.

Если запустим наше приложение, то может добавлять и получать данные.



Здесь мы никак не обработали то, если запрос не выполнялся. Метод `QStringQuery::exec` возвращает тип `bool` и мы можем отследить, выполнялся или не выполнялся наш запрос. А при помощи метода `lastError` или `getLastError` можно получить более подробную информацию об ошибке.

Ссылка на исходники: <https://github.com/nma2207/qt-lessons/tree/master/less3>

Прошу прощения, что нечаянно испортил исходники позапрошлой лекции.