Лекция №5. Qt and SQL

Автор: Набиев Марат

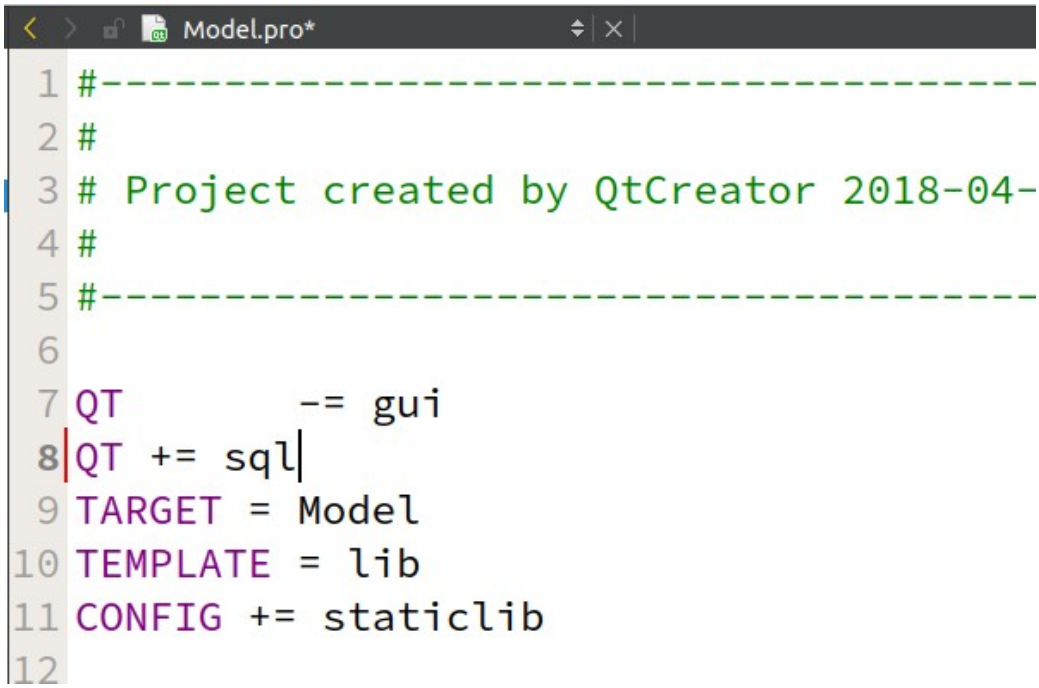
В 3-ей лекции мы пробовали сделать MVP проект, но данные хранили просто в списках, однако в 4-ой лекции мы поработали с SQL и это показалось довольно удобным. Таким образом, было бы удобно хранить данные в базе данных и обращаться к ним через наше приложение. Также после закрытии программы, эти данные 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

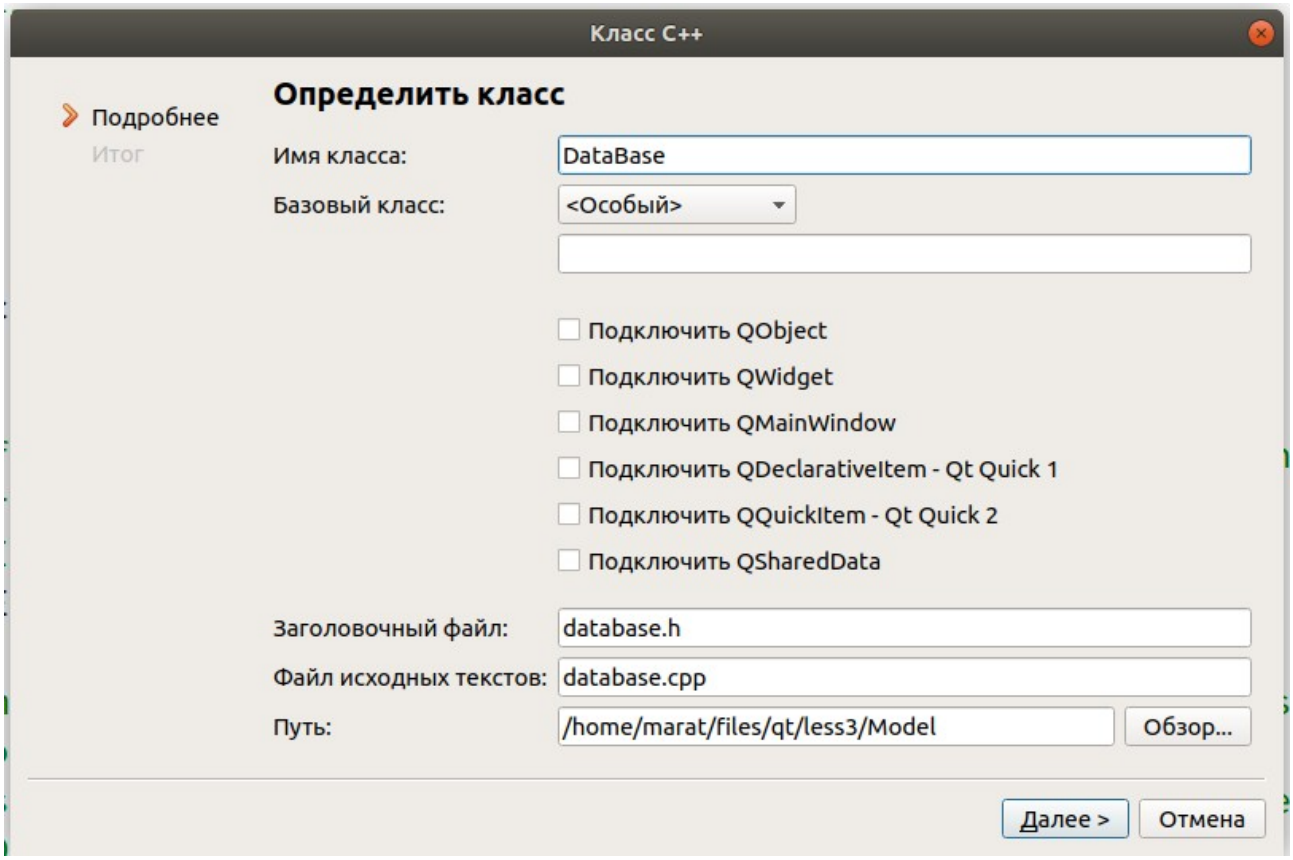
Что же нужно, чтобы смочь работать с sql? Для начала откроем наш MVP проект. Наши изменения коснуться только модуля Model, т. к. Presenter и View никак не влияют на то, каком образом хранятся данные. Откроем `Model.pro` и допишем, что будем работать с sql.



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)



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 закрытые. Для получения экземпляра класса реализуется *статический* метод `getInstance`, который дает экземпляр класса.

Сделаем это!

Так будет выглядеть `database.h`

```
#ifndef DATABASE_H
#define DATABASE_H

//это для соединения с бд
#include <QtSql>
class DataBase
{
private:
    //для доступа к бд
    QSqlDatabase base;
    //статическая ссылка на себя
    static DataBase *instance;
    //закрытый констуктор
    DataBase();
public:
    //статический метод getInstance
    static DataBase* getInstance();
    //для добавления в базу данных
    void addPerson(QString name);
    //для извлечения из базы данных
    QList<QString> getPersons();
};

#endif // DATABASE_H
```

Посмотрим как выглядит `database.cpp`

```cpp
#include "database.h"
#include <QList>
#include <QString>
//в с++ статические поля инициализируются вне класса
DataBase *DataBase::instance = NULL;
DataBase::DataBase()
{

}

DataBase* DataBase::getInstance()
{
    //если нет ни одного экземпляра, то создаем
    if(instance==NULL)
    {
        instance = new DataBase;
    }
    //возвращаем ссылку на экземпляр
    return instance;
}

void DataBase::addPerson(QString name)
{
    //пока без реализации
}
QList<QString> DataBase::getPersons()
{
    //TODO
    return QList<QString>();
}

```

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

```cpp
DataBase::DataBase()
{
    //говорим, что хотим добавить базу данных  SQLite v3
    base =  QSqlDatabase::addDatabase("QSQLITE");
    //называем нашу db
    base.setDatabaseName("my_db.sqlite");
    //открываем
    base.open();
}
```

Вроде все понятно. Если непонятно, то пишите.

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

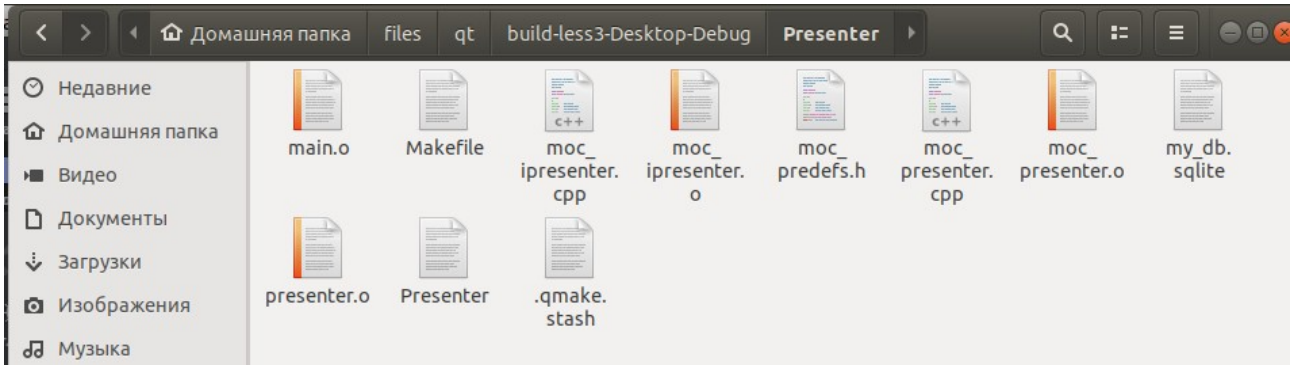
```
#include "imodel.h"
#include "database.h"
class Model: public IModel
{
private:
    //наше хранилище имен
    //QList<QString> names;
    //вместо хранилища имен
    DataBase *base;
public:
    Model();
    void addPerson(QString name) override;
    QList<QString> getPersons() const;
};

#endif // MODEL_H

```

Следующим образом меняется `model.cpp`

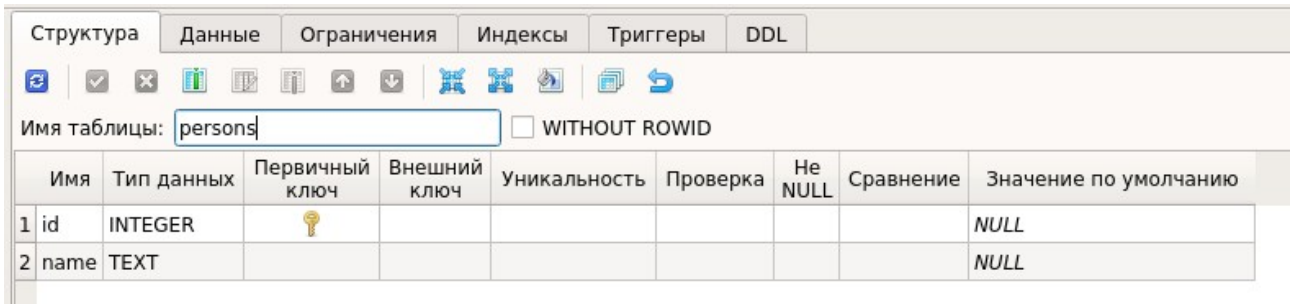
```
#include "model.h"


Model::Model()
{
    //получаем экземпляр
    base = DataBase::getInstance();
}
void Model::addPerson(QString name)
{
    //добавляем в конец имя
    //names.append(name);
    //вместо добавления в список
    base->addPerson(name);
}
QList<QString> Model::getPersons() const
{
    //просто вернем список
    //return names;
    //вместо возвращения списка
    return base->getPersons();
}

```

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 появится файлик 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`

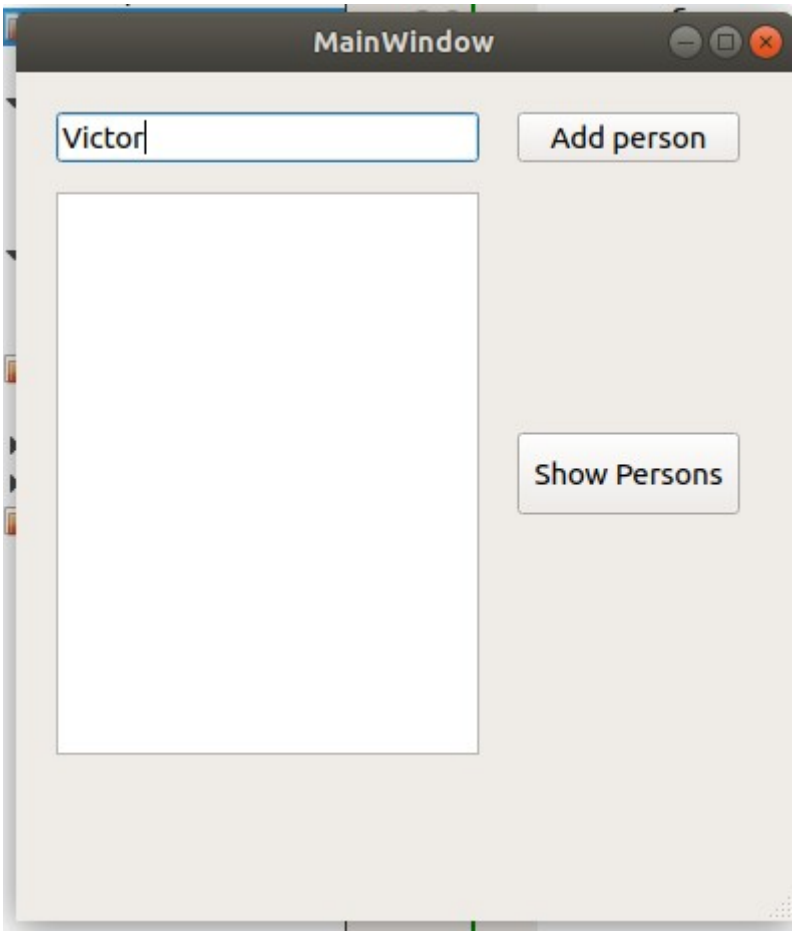
Посмотрим первый способ выполнения запросов.

Напишем строку-- insert - запрос.

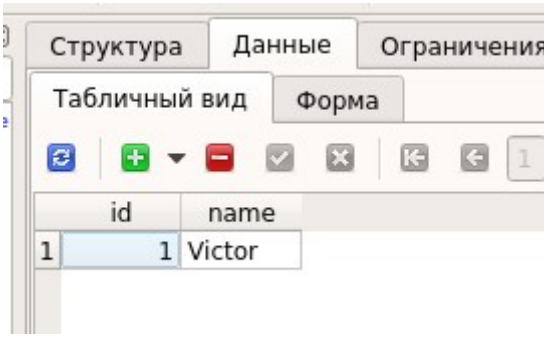
```
27  void DataBase::addPerson(QString name)
28  {
29      //запрос
30      QString insertStr = "INSERT INTO persons(name)"
31                          "VALUES('%1');";
32      //заменяем %1 на имя
33      insertStr = insertStr.arg(name);
34      //создаем SqlQuery
35      QSqlQuery query;
36      //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!



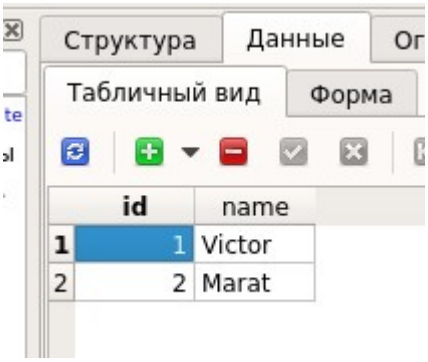
Теперь рассмотрим другой способ добавления.

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      //пишешм 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 ковычки для строки, а здесь это уже не нужно)

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



Теперь надо получать данные.

Напишем SELECT в `DataBase::getPersons`

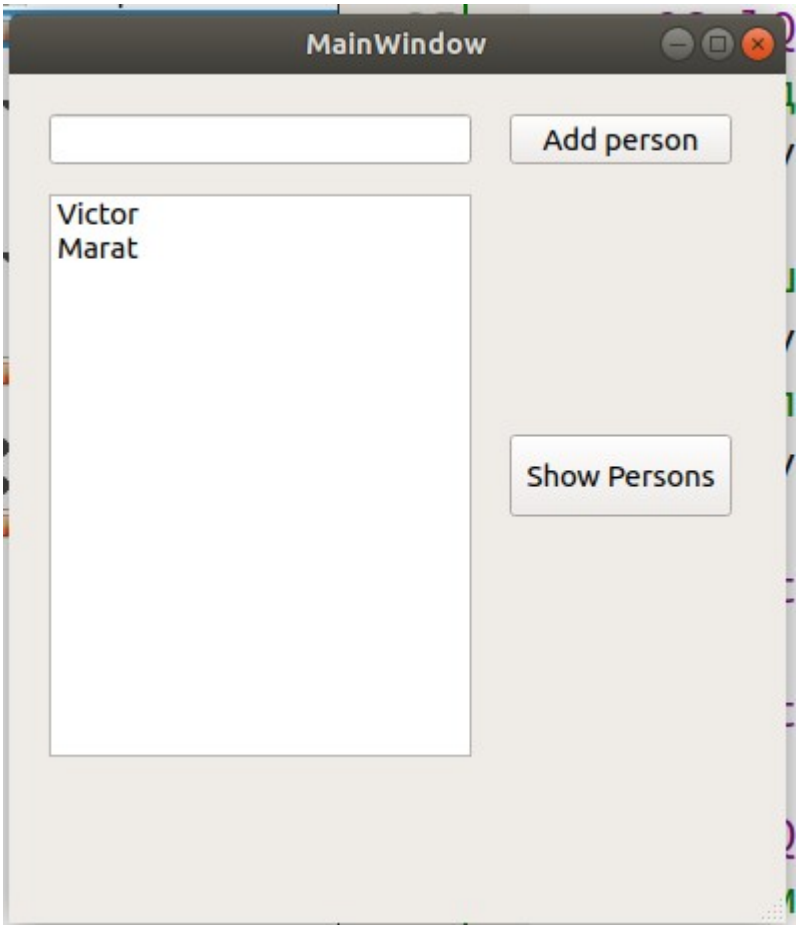
```
44  QList<QString> DataBase::getPersons()
45  {
46      QList<QString> names;
47
48      QSqlQuery selectQuery;
49      //нам нужен только столбец имен
50      selectQuery.prepare("SELECT name FROM persons;");
51      //выполнили
52      selectQuery.exec();
53      //пока есть данные
54      while(selectQuery.next())
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.

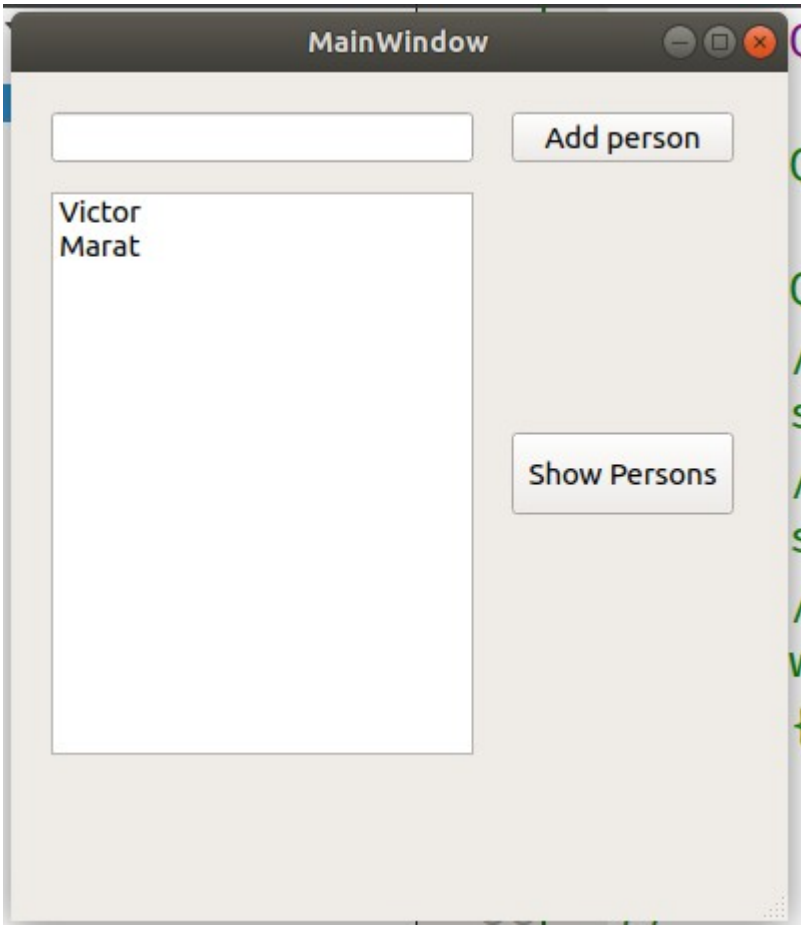


Еще один способ — это через модели.

```
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      }
70      return names;
```

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 выполнился. Метод `QSqlQuery::exec` возвращает тип `bool` и мы можем отследить, выполнился или не выполнился наш запрос. А при помощи метода `lastError` или `getLastError` можно получить более подробную информацию об ошибке.