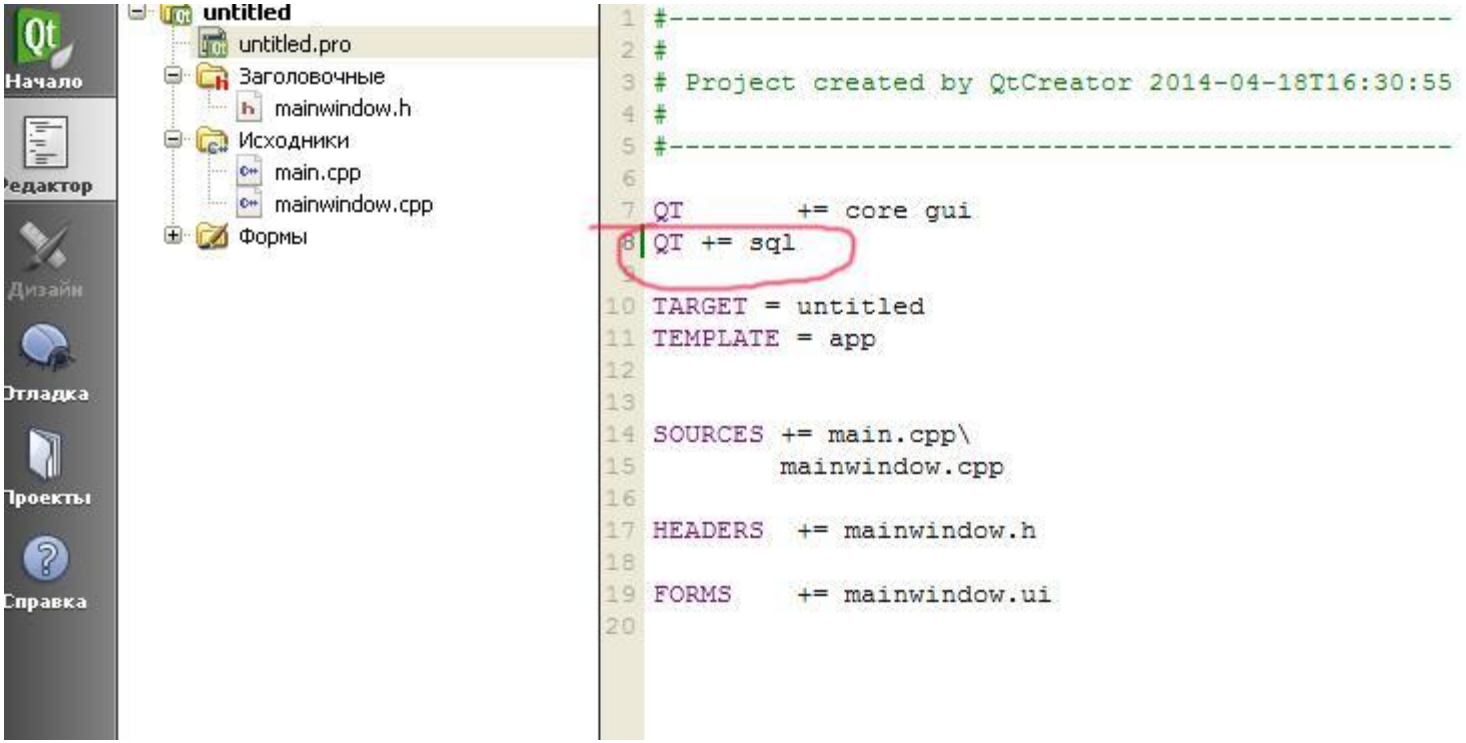
Qt и SQL. Программирование баз данных

Для использования баз данных, Qt предоставляет отдельный модуль **QtSql**. Для его использования необходимо сообщить об этом — просто добавьте в проектный файл следующую строку:



- А для того, чтобы работать с классами этого модуля, необходимо включить заголовочный метафайл **QtSql**.

- Для высылки запросов используется класс **QSqlQuery.**

- Для создания таблицы используется оператор CREATE TABLE, в котором указываются имена столбцов таблицы, их тип, а также задается первичный ключ:

- CREATE TABLE addressbook (
- number INTEGER PRIMARY KEY NOT NULL,
- name VARCHAR(15),
- phone VARCHAR(12),
- email VARCHAR(15)
-);

- После создания таблицы можно добавлять данные. Для этого SQL предоставляет оператор вставки **insert into**. Сразу после названия таблицы нужно указать в скобках имена столбцов, в которые будут заноситься данные.

- Сами данные указываются после ключевого слова **values**.

- INSERT INTO addressbook (number, name, phone, email)
- VALUES(1, 'Piggy', '+49 631322187', 'piggy@mega.de');
- INSERT INTO addressbook (number, name, phone, email)
- VALUES(2, 'Kermit', '+49 631322181', 'kermit@mega.de');

- Составной оператор **select ... from ... where** осуществляет операции выборки и проекции. Выборка соответствует выбору строк, а проекция — выбору столбцов. Этот оператор возвращает таблицу, созданную согласно заданным критериям.

- Ключевое слово **SELECT** является оператором для проведения проекции, то есть в нем указываются столбцы, которые должны стать ответом на запрос. Если указать после **SELECT** знак *, то результирующая таблица будет содержать все столбцы таблицы, к которой был адресован запрос. Указание конкретных имен столбцов устраняет в ответе все остальные.

- Ключевое слово **FROM** задает таблицу, к которой адресован запрос.
- Ключевое слово **WHERE** является оператором выборки. Выборка осуществляется согласно условиям, указанным сразу после оператора.

- Например, для получения адреса электронной почты мисс Piggy нужно сделать следующее:
- SELECT email
- FROM addressbook
- WHERE name = 'Piggy';

- Для изменения данных таблицы используется составной оператор **UPDATE ... SET**. После названия таблицы в операторе **SET** указывается название столбца (или столбцов, через запятую), в который будет заноситься нужное значение. Изменение данных производится в строках, удовлетворяющих условию, поставленному в ключевом слове **WHERE**.

- В показанном ниже примере производится замена адреса электронной почты мисс Piggy с piggy@mega.de на piggy@supermega.de:
- UPDATE addressbook
- SET email = 'piggy@supermega.de'
- WHERE name = 'Piggy';

- Удаление строк из таблицы производится при помощи оператора **DELETE ... FROM**. После ключевого слова WHERE следует критерий, согласно которому производится удаление строк. Например, удалить адрес мисс Piggy из таблицы можно следующим образом:
- DELETE FROM addressbook
- WHERE name = 'Piggy';

Классы модуля QtSql разделяются на три уровня:

- Уровень драйверов.
- Программный уровень.
- Уровень пользовательского интерфейса.

- К первому уровню относятся классы для получения данных на физическом уровне. Это такие классы, как: QSqlDriver, QSqlDriverCreator<T*>, QSqlDriverCreatorBase, QSqlDriverPlugin и QSqlResult
- Они необходимы для создания собственного драйвера для менеджера базы данных. Но в большинстве случаев все ограничивается использованием конкретной СУБД , поддерживаемой Qt.

- Классы второго уровня предоставляют программный интерфейс для обращения к базе данных. К классам этого уровня относятся следующие классы: QSqlDatabase, QSqlQuery, QSqlError, QSqlField, QSqlIndex и QSqlRecord.

- Третий уровень предоставляет модели для отображения результатов запросов в представлениях интервью. К этим классам относятся: QSqlQueryModel, QSqlTableModel и QSqlRelationalTableModel.

- Для соединения с базой данных прежде всего нужно активизировать драйвер. для этого вызывается статический метод **QSqlDatabase::addDatabase().**
- В него нужно передать строку, обозначающую идентификатор драйвера СУБД.

Для того чтобы подключиться к базе данных, потребуется четыре следующих параметра:
имя базы данных — передается в метод QSqlDatabase:: setDatabaseName();
имя пользователя, желающего к ней подключиться, — передается в метод QSqlDatabase::setUserName();
имя компьютера, на котором размещена база данных, — передается в метод QSqlDatabase::setHostName();
пароль — передается в метод QSqlDatabase::setPassword().

```
#include <QtDebug>
#include <QtGui>
#include <QSqlDatabase>
#include <QSqlQueryModel>
#include <QSqlError>
#include <QTableView>
#include <QtSql>


int main(int argc, char *argv[])
{
QApplication app(argc, argv);

QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("MyBase1");
    db.setUserName("user");
    db.setHostName("localhost");
    db.setPassword("password");
```

- Методы должны вызываться из объекта, созданного с помощью статического метода **QSqlDatabase::addDatabase().**

- Само соединение осуществляется методом **QSqlDatabase::open()**. Значение, возвращаемое им, рекомендуется проверять. В случае возникновения ошибки, информацию о ней можно получить с помощью метода **QSqlDatabase::lastError()**, который возвращает объект класса **QSqlError**.

- Его содержимое можно вывести на экран с помощью **qDebug()**. Если у вас возникла необходимость получить строку с ошибкой, то нужно вызвать из объекта класса **QSqlError** метод **text().**

```
19      if (!db.open()) {
20              qDebug() << "Cannot open database:" << db.lastError();
21              return false; }
22      else qDebug() << "The Base.";
```

- Для исполнения команд SQL, после установления соединения, можно использовать класс **QSqlQuery**.
- Запросы (команды) оформляются в виде обычной строки, которая передается в конструктор или в метод **QSqlQuery::exec()**.
- В случае конструктора, запуск команды будет производиться автоматически, при создании объекта.

```
24      QSqlQuery query;
25         QString   str  = "CREATE TABLE Mybase ( "
26                             "number INTEGER PRIMARY KEY NOT NULL, "
27                             "name   VARCHAR(15), "
28                             "phone  VARCHAR(12), "
29                             "email  VARCHAR(15) "
30                           ");";
31
32         if (!query.exec(str)) {
33             qDebug() << "Unable to create a table";
34         }
```

- Класс **QSqlQuery** предоставляет возможность навигации. Например, после выполнения запроса **SELECT** можно:

- С помощью метода **next()** перемещаться на следующую строку данных,

- Методом **previous()** перемещаться на предыдущую строку данных.

- При помощи методов **first()** и **last()** можно установить первую и последнюю строку данных соответственно.

- Метод **seek()** устанавливает строку данных по указанному целочисленному индексу в его параметре.

- Количество строк данных можно получить вызовом метода **size().**

- Дополнительные сложности возникают с запросом **INSERT**. Дело в том, что в запрос нужно внедрять данные. Для достижения этого можно воспользоваться двумя методами: **prepare()** и **bindValue()**. В методе **prepare()** мы задаем шаблон, данные в который подставляются методами **bindValue().** Например:

```
query.prepare("INSERT INTO addressbook
   (number, name, phone, email)
   VALUES(:number, :name, :phone, :email);");
query.bindValue (" :number", "1");
query.bindValue(":name", "Piggy");
query.bindValue(":phone", " + 49 631322187");
query.bindValue(":email", "piggy@mega.de");
```

- Также можно воспользоваться вариантом использования безымянных параметров:

```
query.prepare("INSERT INTO addressbook (number, name, phone, email) VALUES(?, ?, ?, ?);");
query.bindValue("1");
query.bindValue("Piggy");
query.bindValue("+ 49 631322187");
query.bindValue("piggy@mega.de");
```

- В качестве третьего варианта — можно воспользоваться классом **QString**, в частности методом **QString::arg()**, с помощью которого можно произвести подстановку значений данных.

```
36          //Adding some information
37              QString strF =
38                      "INSERT INTO Mybase  (number, name, phone, email) "
39                      "VALUES(%1, '%2', '%3', '%4');";
40
41              str = strF.arg("1")
42                        .arg("Andrew")
43                        .arg("+49 631322187")
44                        .arg("andrew@mega.de");
45              if (!query.exec(str)) {
46                  qDebug() << "Unable to do insert opeation";
47              }
48
49              str = strF.arg("2")
50                        .arg("Alex")
51                        .arg("+49 631322181")
52                        .arg("alex@mega.de");
53              if (!query.exec(str)) {
54                  qDebug() << "Unable to do insert operation";
55              }
```

```
56
57          if (!query.exec("SELECT * FROM MyBase;")) {
58              qDebug() << "Unable to execute query - exiting";
59              return 1;
60          }
61
62          //Reading of the data
63          QSqlRecord rec     = query.record();
64          int        nNumber = 0;
65          QString    strName;
66          QString    strPhone;
67          QString    strEmail;
68
69          while (query.next()) {
70              nNumber  = query.value(rec.indexOf("number")).toInt();
71              strName  = query.value(rec.indexOf("name")).toString();
72              strPhone = query.value(rec.indexOf("phone")).toString();
73              strEmail = query.value(rec.indexOf("email")).toString();
74
75              qDebug() << nNumber << " " << strName << ";\t"
76                       << strPhone << ";\t" << strEmail;
77          }
78
79      return app.exec();
80  }
81
```

```
#include <QTableView>
#include <QtSql>


int main(int argc, char *argv[])
{
QApplication app(argc, argv);

QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("MyBase1");
    db.setUserName("user");
    db.setHostName("localhost");
    db.setPassword("password");
    if (!db.open()) {
            qDebug() << "Cannot open database:" << db.lastError();
            return false; }
    else qDebug() << "The Base.";

    QSqlQuery query;
       QString   str  = "CREATE TABLE Mybase ( "
                        "number INTEGER PRIMARY KEY NOT NULL, "
```

Консоль приложения

untitled

```
Запускается D:\Kat\basa\untitled-build-desktop\debug\untitled.exe...
The Base.
1   "Andrew" ;        "+49 631322187" ;   "andrew@mega.de"
2   "Alex" ;          "+49 631322181" ;   "alex@mega.de"
```

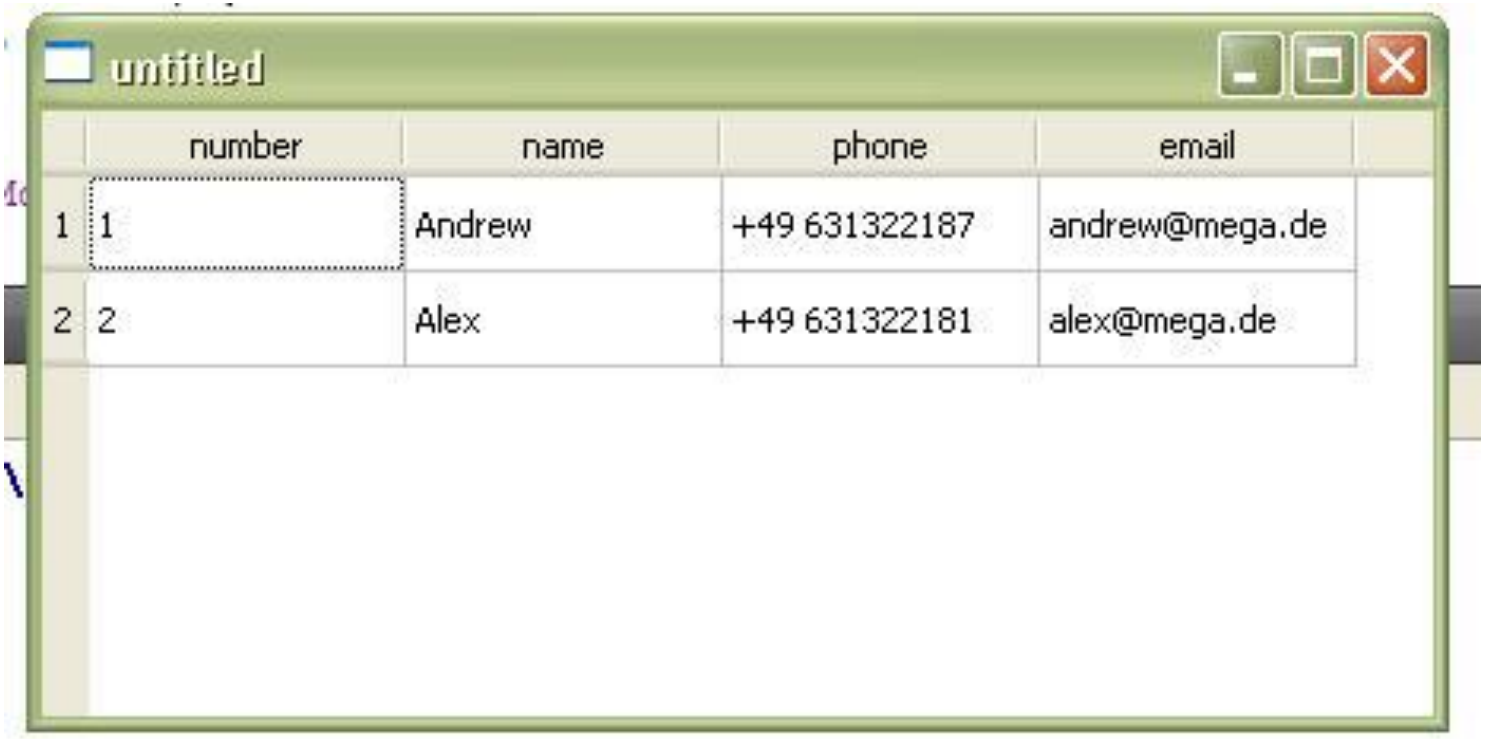
- В случае удачного соединения с базой данных с помощью **createConnection()** создается строка, содержащая команду SQL для создания таблицы.
- Эта строка передается в метод **exec()** объекта класса **QSqlQuery**. Если создать таблицу не удается, то на консоль будет выведено предупреждающее сообщение.
- Ввиду того, что в таблицу будет внесена не одна строка, в строковой переменной **strF** при помощи символов спецификации определяется шаблон для команды **INSERT**. Вызовы методов **arg()** класса **QString** подставляют нужные значения используя шаблон.

- Затем, когда база данных создана и все данные были внесены в таблицу, выполняется запрос **SELECT**, помещающий строки и столбцы таблицы в объект **query**.
- Вывод значений таблицы на консоль производится в цикле. При первом вызове метода **next()** этот объект будет указывать на самую первую строку таблицы.
- Последующие вызовы приведут к перемещению указателя на следующие строки. В том случае, если записей больше нет, метод **next()** вернет false, что приведет к выходу из цикла.

- Для получения результата запроса следует вызвать метод **QSqlQuery::value()**, в котором необходимо передать номер столбца. Для этого мы воспользуемся методом **record().**
- Этот метод возвращает объект класса **QSqlRecord**, который содержит информацию, относящуюся к запросу **SELECT.**
- С его помощью, вызовом метода **QSqlRecord::indexOf()**, мы получаем индекс столбца.

- Метод **value()** возвращает значения типа **QVariant. QVariant** — это специальный класс, объекты которого могут содержать в себе значения разных типов.
- Поэтому, в нашем примере, полученное значение нужно преобразовать к требуемому типу, воспользовавшись методами **QVariant::toInt()** и **QVariant::toString().**

- Модуль QtSql поддерживает концепцию Интервью, самый простой способ отобразить данные таблицы. Здесь не потребуется цикла для прохождения по строкам таблицы.



```
24          QTableView      view;
25          QSqlTableModel model;
26
27          model.setTable("Mybase");
28          model.select();
29          model.setEditStrategy(QSqlTableModel::OnFieldChange);
30
31          view.setModel(&model);
32          view.show();
```

- После соединения с базой данных, проводимого с помощью функции **createConnection()**,

- создается объект табличного представления **QTableView** и объект табличной модели **QSqlTableModel**. Вызовом метода **setTable()** мы устанавливаем актуальную базу в модели. Вызов метода **select()** производит заполнение данными.

- вызовом метода **setEditStrategy()** устанавливается стратегия редактирования **SqlTableModel::OnFieldChange**. Теперь данные нашей модели можно изменять после двойного щелчка на ячейке. В завершение мы устанавливаем модель в представлении вызовом метода **setModel().**

- Класс **QSqlTableModel** предоставляет три стратегии редактирования, которые устанавливаются с помощью метода **setEditStrategy():**
- **onRowChange** — производит запись данных, как только пользователь перейдет к другой строке таблицы;
- **onFieldChange** — производит запись данных после того, как пользователь перейдет к другой ячейке таблицы;
- **OnManualSubmit** —записывает данные по вызову слота submitAl(). Если вызывается слот revertAll(), то данные возвращаются в исходное состояние.

- Если вам понадобится произвести отображение данных какого-либо конкретного опроса **SELECT**, то для этого целесообразнее будет воспользоваться другим классом SQL-моделей — классом **QSqlQueryModel**.
- Листинг ниже иллюстрирует отображение только электронных адресов и телефонных номеров всех контактов с именем Piggy. В нашем случае он будет всего лишь один.

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    if (!createConnection()) {
        return -1;
    }

    QTableView     view;
    QSqlQueryModel model;
    model.setQuery("SELECT phone, email "
                   "FROM addressbook "
                   "WHERE name = 'Piggy';"
                  );

    if (model.lastError().isValid()) {
        qDebug() << model.lastError();
    }

    view.setModel(&model);
    view.show();

    return app.exec();
}
```

Здесь мы создаем табличное представление **QTableView** и модель опроса **QSqlQueryModel**. Строка запроса передается в метод **setQuery()**, после чего результат выполнения запроса проверяется в операторе **if** на наличие проблем исполнения, с помощью метода **lastError().** Неверный объект класса **QSqlError**, возвращаемый этим методом, означает, что ошибок нет и никаких проблем не произошло. Это проверяется методом **isValid()**. Возникновение проблем повлечет их отображение в **qDebug().** В завершение, модель устанавливается в представлении вызовом метода **setModel().**