

С++: разработка программ с графическим интерфейсом на Qt

Работа с базами данных в Qt

Использование БД SQLite в проектах

[Краткие сведения о СУБД SQLite](#)

[Основные операции при работе с БД в Qt](#)

[Инструменты Qt для работы с SQLite](#)

[Реализация работы с БД через _____ концепцию «модель — представление»](#)

[Сборка и подключение других библиотек баз данных \(PostgreSQL/MySQL/ETS\)](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Краткие сведения о СУБД SQLite

SQLite — это внутрипроцессная библиотека на языке C, которая реализует небольшой, быстрый, автономный, высоконадежный, полнофункциональный механизм базы данных SQL. Есть серверные БД, к которым можно подключаться с разных устройств, а сами данные хранятся на специально выделенном сервере (MySQL, MS SQL, PostgreSQL), и есть автономные БД, к которым и относится SQLite. SQLite реализует автономный (бессерверный) транзакционный механизм базы данных SQL без необходимости установки, настройки и конфигурации. Ее код открыт, и она бесплатна для использования в любых целях, коммерческих или личных. SQLite читает и пишет напрямую в обычные файлы на диске: полная база данных с несколькими таблицами, индексами, триггерами и представлениями содержится в одном файле. Формат файла базы данных кроссплатформенный — вы можете свободно копировать БД между 32-разрядными и 64-разрядными системами или между архитектурами с прямым и обратным порядком байтов. Поэтому SQLite хорошо подходит для несложных приложений. Это замена скорее не для Oracle, а для **fopen()**. Файлы базы данных SQLite — это рекомендуемый формат хранения Библиотеки Конгресса США.

Следующие два объекта и восемь методов — основные элементы интерфейса SQLite:

1. **Sqlite** — объект подключения к базе данных. Создается при помощи метода **sqlite3_open()** и уничтожается методом **sqlite3_close()**.
2. **Sqlite_stmt** — подготовленный объект заявления (запроса в БД). Создается методом **sqlite3_prepare()** и уничтожается с помощью **sqlite3_finalize()**.
3. **sqlite_open()** — открывает соединение с новой или существующей базой данных SQLite. Конструктор для **sqlite3**.
4. **sqlite_prepare()** — компилирует текст SQL в байт-код, который будет выполнять запросы или обновлять базу данных. Конструктор для **sqlite3_stmt**.
5. **sqlite_bind()** — хранит данные приложения в параметрах исходного SQL.
6. **sqlite_step()** — переход **sqlite3_stmt** к следующей строке результатов или до завершения.
7. **sqlite_column()** — значения столбцов в текущей строке результатов для **sqlite3_stmt**.
8. **sqlite_finalize()** — деструктор для **sqlite3_stmt**.
9. **sqlite_close()** — деструктор для **sqlite3**.
10. **sqlite_exec()** — функция-оболочка, которая выполняет **sqlite3_prepare()**, **sqlite3_step()**, **sqlite3_column()** и **sqlite3_finalize()** для строки из одного или нескольких операторов SQL.

Работа с БД в фреймворке Qt осуществляется с помощью модуля **sql**. Его нужно добавить в файл **qmake** (с расширением **.pro**): **QT += sql**.

```
QT           += core gui sql

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

```
TARGET = Testmysql
TEMPLATE = app
DEFINES += QT_DEPRECATED_WARNINGS

CONFIG += c++11

SOURCES += \
    main.cpp \
    mainwindow.cpp

HEADERS += \
    mainwindow.h

FORMS += \
    mainwindow.ui

qnx: target.path = /tmp/${TARGET}/bin
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Основные операции при работе с БД в Qt

Основные операции:

- создание БД;
- создание таблицы;
- поиск по ключу в таблице;
- удаление элементов в таблице по ключу;
- удаление таблицы;
- удаление базы данных.

Первый и последний пункты относятся к серверным БД, когда по одному сетевому адресу располагаются несколько различных БД. Фреймворк Qt содержит универсальный интерфейс по работе с различными БД. БД в представлении Qt являются драйверами к модулю **QtSql**. По умолчанию при установке фреймворка доступна БД SQLite, для остальных БД необходима установка и сборка драйверов под Qt.

Для работы с БД необходимо:

1. Создать БД с подключением необходимого драйвера.
2. Настроить HOST (если это подключение к серверной БД).
3. Ввести порт, имя пользователя и пароль (если это подключение к серверной БД).
4. Ввести название БД (для SQLite — путь к файлу).

```

database = QSqlDatabase::addDatabase("QSQLITE"); // SQLite-driver
database.setDatabaseName("./database.db");      // Файл БД
if (!database.open())
{
    // Обработка ошибки
}

QSqlQuery query(database);

```

5. Открыть БД и проверить, успешно ли была выполнена операция.
6. Создать объект для обмена сообщениями с драйвером БД.

Для создание таблиц в SQL-запросе используется команда CREATE.

Создадим таблицу **MyFirstTable** с тремя столбцами: порядковый номер, название города и краткое описание. SQL-команда:

```

CREATE TABLE `MyFirestTable` (`id` INT(11) NOT NULL,
`townname` VARCHAR(25) NOT NULL,
`country` VARCHAR(2048) NOT NULL)
//*****Qt*****
bool res = query.exec("create table MyFirstTable(id int(11)not null, townname
varchar(25)not null, country varchar(2048)"); // true - если команда выполнена
// успешно

```

Запрос для создания таблицы (при условии, что она еще не была создана в БД) будет выглядеть следующим образом:

```

CREATE TABLE if not exists `MyFirestTable` (`id` INT(11) NOT NULL,
`townname` VARCHAR(25) NOT NULL,
`country` VARCHAR(2048) NOT NULL)
//*****Qt*****
bool res = query.exec("create if not exists table MyFirstTable(id int(11)not
null, townname varchar(25)not null, country varchar(2048)");

```

Добавим описание Ярославля. SQL-код:

```

INSERT INTO MyFirestTable
VALUES(1, 'Ярославль', 'Российская Федерация.')
//*****Qt*****
bool res = query.exec("insert into MyFirstTable values(1,'Ярославль','Российская
Федерация')");

```

Для считывания данных из БД используется следующий SQL-запрос (считаем названия городов и их ID, но только четные). С помощью ключевого слова WHERE установим условие, по которому будет фильтроваться результат:

```
SELECT townname, id FROM MyFirstTable WHERE id % 2 > 0
// *****Qt*****
bool res = query.exec("select townname,id from MyFirstTable where id % 2>0");
// Теперь считываем ответ
while(query.next())
{
    int id = query.value("id").toInt(); // Считываем id; можно указать номер
                                         // столбца query.value(0).toInt()
    QString town = query.value("townname").toString();
}
```

Замена текущей записи на новую. Заменяем надпись «Москва» на «Ярославль»:

```
UPDATE MyFirstTable SET townname='Ярославль' WHERE townname='Москва'
//*****Qt*****
bool res = query.exec("update MyFirstTable set townname='Ярославль' WHERE
townname='Москва'");
```

Получение количества записей:

```
SELECT count (*) FROM MyFirstTable
// *****Qt*****
bool res = query.exec("select count(*) from MyFirstTable");
// Теперь считываем ответ
int amount = 0;
if (query.next())
{
    amount = query.value(0).toInt();
}
```

Если же метод **QSqlQuery::exec(QString)** возвращает false, то можно вывести ошибку, которая возникла при попытке отправить запрос (как правило, это синтаксические ошибки):

```
#include <QSqlError>
...
void ...() {
    QSqlQuery query(database);
    if (!query.exec("select count(*) into MyFirstTable")){
        qDebug << query.lastError().text(); // Будет выведена ошибка. В данном
                                             // случае - неправильное
                                             // использование ключевого слова
                                             // into

        return;
    }
}
```

```
}  
}
```

Инструменты Qt для работы с SQLite

Qt обеспечивает обширную совместимость с базами данных, с поддержкой как открытых, так и проприетарных продуктов. Поддержка SQL интегрирована с архитектурой «модель — представление» Qt, что упрощает интеграцию GUI приложений с базами данных.

Создадим нашу БД. Для этого понадобится класс **QSqlDatabase**: он нужен для создания объекта подключения к БД. Этапы:

1. Создаем объект с драйвером БД (для SQLite: QSQLITE).
2. Устанавливаем имя БД.
3. Открываем/устанавливаем соединение с БД.

Запишем город и описание. После чего считаем содержимое БД:

```
#include "mainwindow.h"  
#include <QApplication>  
#include <QSqlQuery>  
#include <QDebug>  
  
MainWindow::MainWindow(QWidget *parent)  
    : QMainWindow(parent)  
{  
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");           // Установка  
                                                                    // драйвера БД  
    db.setDatabaseName("TownList");                                   // Имя БД (если  
                                                                    // SQLite - имя  
                                                                    // файла)  
    if (!db.open())                                                  // Попытка  
                                                                    // установить  
                                                                    // соединение  
    {  
        qApp->quit();  
    }  
    else {  
        QSqlQuery query;  
        auto a = query.exec("create table townlist(id int(11)," // Создаем  
                                                                    // таблицу  
                               "town varchar(50),"                 
                               "Describ varchar(1024))"  
                               );  
        query.exec("insert into townlist values(1, 'Москва', 'Столица Российской  
Федерации')"); // Добавляем  
                                                                    // город и его
```

```

a =query.exec("select id, town from townlist");

while (query.next())

{
    int id = query.value(0).toInt();
    QString town = query.value(1).toString();
    QString d = QString::number(id) + ": " + town;
    qDebug << d;
}

}

MainWindow::~MainWindow()
{
    database.close();
}

```

// описание
// SQL-запрос
// на
// считывание
// содержимого
// БД
// Цикл
// считывания
// данных

В Qt для работы с базами данных есть несколько классов: **QDataBase**, **QSqlQuery**, **QSqlError**.

Реализация работы с БД через концепцию «модель — представление»

Для работы с данными базы данных можно использовать виджеты **QTreeView**, **QTableView**. Для использования БД нужно воспользоваться моделью **QSqlTableModel**.

QSqlTableModel позволяет работать с конкретной таблицей базы данных, имея при этом возможность легко редактировать значения ее ячеек. Давайте для начала создадим базу данных и получим доступ к данным, используя представление **QTableView**.

SQL коды можно выполнить как в СУБД (вкладка SQL в DB Browser for SQLite), так и из кода, но они должны быть вызваны последовательно.

Таблица с грейдами

```

CREATE TABLE IF NOT EXISTS grades (
    id INTEGER PRIMARY KEY,
    position TEXT NOT NULL,

```

```
        min_salary REAL NOT NULL,  
        max_salary REAL NOT NULL  
    );
```

Таблица с отделами

```
CREATE TABLE IF NOT EXISTS departments (  
    id INTEGER PRIMARY KEY,  
    department_name TEXT NOT NULL  
);
```

Таблица со списком сотрудников

```
CREATE TABLE IF NOT EXISTS employee (  
    id INTEGER PRIMARY KEY,  
    name TEXT NOT NULL,  
    phone TEXT NOT NULL UNIQUE,  
    department INTEGER NOT NULL,  
    date_of_birth TEXT NOT NULL,  
    grade INTEGER NOT NULL,  
    salary REAL NOT NULL,  
    FOREIGN KEY(department) REFERENCES departments(id),  
    FOREIGN KEY(grade) REFERENCES grades(id)  
);
```

Добавим значения в таблицы:

```
INSERT INTO grades  
VALUES (0, "Стажер", 25000, 45000),  
(1, "Младший сотрудник", 35000, 75000),  
(2, "Сотрудник", 75000, 140000),  
(3, "Старший сотрудник", 140000, 220000),  
(4, "Главный специалист", 210000, 270000),  
(5, "Руководитель подразделения", 250000, 350000),  
(6, "Руководитель департамента", 280000, 400000);
```

```
INSERT INTO departments  
VALUES (0, "Департамент веб-разработки"),  
(1, "HR департамент"),  
(2, "Казначейство"),  
(3, "Департамент маркетинга и стратегического планирования"),  
(4, "Департамент DevOps"),  
(5, "Отдел тестирования"),  
(6, "Отдел разработки мобильных и десктоп приложений");
```

```
INSERT INTO employee  
VALUES  
(0, "Сергеев Олег Вадимович", "+7 (321) 331 11 22", 0, "1992-03-03", 4, 220000),  
(1, "Алиева Сабрина Адамовна", "+7 (321) 331 11 24", 1, "1998-01-03", 1, 63000),  
(2, "Матвейчук Екатерина Олеговна", "+7 (321) 331 11 23", 1, "1968-12-28", 6, 370000),  
(3, "Семенова Елена Сергеевна", "+7 (321) 331 11 25", 0, "1994-05-21", 4, 200000),  
(4, "Агаев Ринат Ахметович", "+7 (321) 331 11 27", 2, "1982-03-01", 5, 320000),  
(5, "Серков Дмитрий Павлович", "+7 (321) 331 11 32", 5, "1992-11-12", 2, 85000),  
(6, "Ладогин Олег Петрович", "+7 (321) 331 11 33", 6, "1991-12-03", 6, 270000),  
(7, "Ладогин Иван Петрович", "+7 (321) 331 11 34", 6, "1991-12-03", 3, 150000),
```



```
(8, "Гант Игорь Сергеевич", "+7 (321) 331 11 37", 2, "1961-06-03", 6, 410000),
(9, "Игонов Леонид Александрович", "+7 (321) 331 11 83", 0, "1996-12-03", 4, 270000),
(10, "Ермаков Лев Леонидович", "+7 (321) 331 11 82", 0, "1983-09-11", 6, 310000),
(11, "Дубцова Евгения Александровна", "+7 (321) 331 11 92", 3, "1978-08-04", 6, 370000),
(12, "Щербакова Алина Алексеевна", "+7 (321) 331 11 99", 3, "1993-09-11", 4, 210000),
(13, "Зудина Ольга Дмитриевна", "+7 (321) 331 11 98", 3, "1996-11-11", 3, 180000),
(14, "Горин Глеб Александрович", "+7 (321) 331 11 97", 4, "1989-01-18", 6, 310000),
(15, "Долгов Иван Викторович", "+7 (321) 331 11 96", 0, "1991-12-14", 6, 400000);
```

Теперь у нас есть три заполненные таблицы. Настало время отобразить их, используя QSqlTableModel.

Добавим на форму ui QTableView, после чего добавим следующий код в mainwindow.cpp

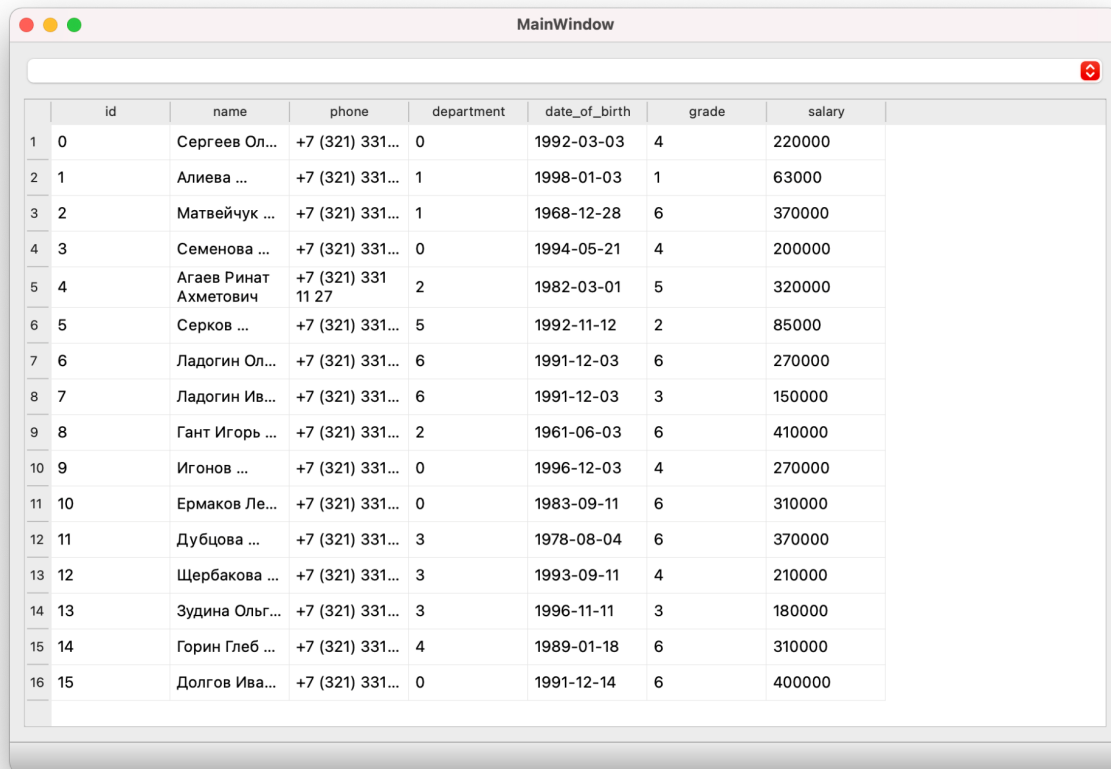
```
// Выше не забудьте добавить необходимые инклюды
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QSqlTableModel* model = new QSqlTableModel{ui->tableView,
    QSqlDatabase::addDatabase("SQLITE")};
    auto db = model->database();
    db.setDatabaseName("/Users/val/Downloads/dataBase/db_1.db");
    if (!db.open())
        qDebug() << "Бд не открыта" << db.lastError();

    ui->tableView->setModel(model);

    model->setTable("employee");
    model->select();
}
```

Результат выполнения программы



The screenshot shows a Qt application window titled "MainWindow". Inside the window is a table with 8 columns: an index column, "id", "name", "phone", "department", "date_of_birth", "grade", and "salary". The table contains 16 rows of employee data. The first column of the table (the index) contains numbers from 1 to 16. The "id" column contains numbers from 0 to 15. The "name" column contains names like "Сергеев Ол...", "Алиева ...", etc. The "phone" column contains phone numbers like "+7 (321) 331...". The "department" column contains numbers from 0 to 6. The "date_of_birth" column contains dates in "YYYY-MM-DD" format. The "grade" column contains numbers from 1 to 6. The "salary" column contains salary values like 220000, 63000, etc.

	id	name	phone	department	date_of_birth	grade	salary
1	0	Сергеев Ол...	+7 (321) 331...	0	1992-03-03	4	220000
2	1	Алиева ...	+7 (321) 331...	1	1998-01-03	1	63000
3	2	Матвейчук ...	+7 (321) 331...	1	1968-12-28	6	370000
4	3	Семенова ...	+7 (321) 331...	0	1994-05-21	4	200000
5	4	Агаев Ринат Ахметович	+7 (321) 331 11 27	2	1982-03-01	5	320000
6	5	Серков ...	+7 (321) 331...	5	1992-11-12	2	85000
7	6	Ладогин Ол...	+7 (321) 331...	6	1991-12-03	6	270000
8	7	Ладогин Ив...	+7 (321) 331...	6	1991-12-03	3	150000
9	8	Гант Игорь ...	+7 (321) 331...	2	1961-06-03	6	410000
10	9	Игонов ...	+7 (321) 331...	0	1996-12-03	4	270000
11	10	Ермаков Ле...	+7 (321) 331...	0	1983-09-11	6	310000
12	11	Дубцова ...	+7 (321) 331...	3	1978-08-04	6	370000
13	12	Щербакова ...	+7 (321) 331...	3	1993-09-11	4	210000
14	13	Зудина Ольг...	+7 (321) 331...	3	1996-11-11	3	180000
15	14	Горин Глеб ...	+7 (321) 331...	4	1989-01-18	6	310000
16	15	Долгов Ива...	+7 (321) 331...	0	1991-12-14	6	400000

Мы отобрали данные, которые есть в таблице, теперь мы также можем их редактировать. Все изменения, которые мы будем вносить через таблицу руками, будут тут же занесены в бд, если она не открыта в СУБД в режиме записи (частая проблема, когда базу данных временно блокирует открытая параллельно СУБД, будьте внимательны).

Для редактирования таблицы мы можем выбрать одну из нескольких стратегий, вызвав метод `void setEditStrategy(QSqlTableModel::EditStrategy strategy)`, где `QSqlTableModel::EditStrategy` может быть одним из трех вариантов:

`QSqlTableModel::OnFieldChange` – изменения будут приняты, когда будет изменена любая из ячеек (по умолчанию).

`QSqlTableModel::OnRowChange` – изменения будут приняты, когда пользователь выберет другую строку. Такой вариант работает оптимальнее, но может потерять данные для строки, если программа была закрыта принудительно или из-за ошибки.

QSqlTableModel::OnManualSubmit – изменения будут закешированы и приняты, когда будет вызван метод `submitAll()`, либо модель будет возвращена к прежнему значению при вызове `revertAll()`.

Теперь давайте попробуем сделать переключатель между таблицами нашей бд. Добавим над `QTableView` `QComboBox` и занесем туда названия таблиц.

```
MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent)
, ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QSqlTableModel* model = new QSqlTableModel{ui->tableView,
    QSqlDatabase::addDatabase("SQLITE")};
    auto db = model->database();
    db.setDatabaseName("/Users/val/Downloads/dataBase/db_1.db");
    if (!db.open())
        qDebug() << "Бд не открыта" << db.lastError();

    ui->tableView->setModel(model);

    model->setTable("employee");
    model->select();

    // Инициализация QComboBox
    ui->comboBox->addItem({"employee", "departments", "grades"});
    connect(ui->comboBox, &QComboBox::currentTextChanged, this, [model](const QString&
currentText) {
        model->setTable(currentText);
        model->select();
    });
}
```

Кроме задачи простого отображения данных перед нами нередко стоит задача фильтрации данных. Для этого в классе `QSqlTableModel` есть метод `filter(const QString&)`, который позволяет задать значение для поля `WHERE` в SQL запросе. Давайте усложним нашу программу и добавим возможность выбрать различные критерии для наших таблиц. Для начала добавим еще два `QComboBox` под уже существующим и один `QLineEdit`. Первый `QComboBox` будет выбирать таблицу, второй название колонки, а третий условие (`>`, `<`, `<=`, `=`, `LIKE`, `BEWTEEN` и тд). `QLineEdit` будет принимать значение для нашего условия. Также добавим кнопку “выбрать”.

```
#include <QHash>

namespace {
    const QHash<QString, QStringList> hshTableNameColumns {
        {"employee", {"id", "name", "phone", "department", "date_of_birth", "grade", "salary"}},
        {"departments", {"id", "department_name"}},
        {"grades", {"id", "position", "min_salary", "max_salary"}},
    };
}
```

```

}

MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent)
, ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QSqlTableModel* model = new QSqlTableModel{ui->tableView,
    QSqlDatabase::addDatabase("QSQLITE")};
    auto db = model->database();
    db.setDatabaseName("/Users/val/Downloads/dataBase/db_1.db");
    if (!db.open())
        qDebug() << "Бд не открыта" << db.lastError();

    ui->tableView->setModel(model);

    model->setTable("employee");
    model->select();

    // Инициализация QComboBox
    ui->comboBox->addItems({"employee", "departments", "grades"});
    ui->comboBox_3->addItems({"<", "<=", "=", ">", ">=", "BETWEEN", "NOT BETWEEN", "CUSTOM",
    "LIKE"});
    ui->comboBox_3->setCurrentIndex(-1);
    ui->comboBox_2->addItems(hshTableNameColumns["employee"]);
    ui->comboBox_2->setCurrentIndex(-1);
    connect(ui->comboBox, &QComboBox::currentTextChanged, this, [model, this](const QString&
currentText) {
        model->setTable(currentText);
        model->select();
        ui->comboBox_2->clear();
        ui->comboBox_2->addItems(hshTableNameColumns[currentText]);
        ui->comboBox_2->setCurrentIndex(-1); // Не выбирать первое значение
    });

    connect(ui->pushButton, &QPushButton::clicked, this, [model, this](){
        const auto columnName = ui->comboBox_2->currentText();
        if (columnName.isEmpty()) {
            qDebug() << "Колонка, по которой осуществляется выборка, не выбрана";
            return;
        }

        const auto operation = ui->comboBox_3->currentText();
        if (operation.isEmpty()) {
            qDebug() << "Селектор для выборки не выбран";
            return;
        }

        // Если выборка кастомная
        if (operation == QStringLiteral("CUSTOM")) {
            model->setFilter(ui->lineEdit->text());
            model->select();
        } else {
            const auto filter{ columnName + " " + operation + " " + ui->lineEdit->text() };
            model->setFilter(filter);
            model->select();
        }
    });
}

```

MainWindow

employee

name

LIKE

"%Ладогин%"

Выбрать

	id	name	phone	department	date_of_birth	grade	salary
1	6	Ладогин Ол...	+7 (321) 331...	6	1991-12-03	6	270000
2	7	Ладогин Ив...	+7 (321) 331...	6	1991-12-03	3	150000

MainWindow

employee

salary

BETWEEN

100000 and 250000

Выбрать

	id	name	phone	department	date_of_birth	grade	salary
1	0	Сергеев Ол...	+7 (321) 331...	0	1992-03-03	4	220000
2	3	Семенова ...	+7 (321) 331...	0	1994-05-21	4	200000
3	7	Ладогин Ив...	+7 (321) 331...	6	1991-12-03	3	150000
4	12	Щербакова ...	+7 (321) 331...	3	1993-09-11	4	210000
5	13	Зудина Ольг...	+7 (321) 331...	3	1996-11-11	3	180000

Обратите внимание, что некоторые запросы требуют наличия кавычек, как в первом примере.

В нашем примере с таблицей есть существенный недостаток — тяжелые для восприятия человеком поля id вместо удобных для глаза названий отделов и грейдов. Его можно решить, используя

QSqlRelationalTableModel. Обычный **QSqlTableModel** не может обрабатывать внешние ключи, об этом сказано в документации. Для этой задачи был сделан класс-наследник

QSqlRelationalTableModel. Этот класс позволяет автоматически подставлять значения из внешних таблиц в колонки, где проставлены foreign keys.

Давайте заменим класс **QSqlTableModel** на **QSqlRelationalTableModel** и убедимся в том, что программа продолжает работать корректно. Однако графические и логические элементы прошлой программы, связанные с выборкой данных, для удобства можно убрать.

Добавим код между выбором таблицы и select() в стеке конструктора главного окна и для корректной работы программы добавим его же в лямбду, присоединенную к сигналу первого комбобокса.

Синтаксис добавления связи:

```
model->setRelation(Номер колонки Foreign Key, QSqlRelation("Название таблицы с первичным ключом, на который ссылается foreign key", "название колонки первичного ключа", "название колонки, откуда нужно брать человекочитаемое описание"));
```

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QSqlRelationalTableModel* model = new QSqlRelationalTableModel{ui->tableView,
    QSqlDatabase::addDatabase("QSQLITE")};
    auto db = model->database();
    db.setDatabaseName("/Users/val/Downloads/dataBase/db_1.db");
    if (!db.open())
        qDebug() << "Бд не открыта" << db.lastError();

    ui->tableView->setModel(model);

    model->setTable("employee");
    model->setRelation(3, QSqlRelation("departments", "id", "department_name"));
    model->setRelation(5, QSqlRelation("grades", "id", "position"));
    model->select();

    // Инициализация QComboBox
    ui->comboBox->addItem({"employee", "departments", "grades"});
```

```

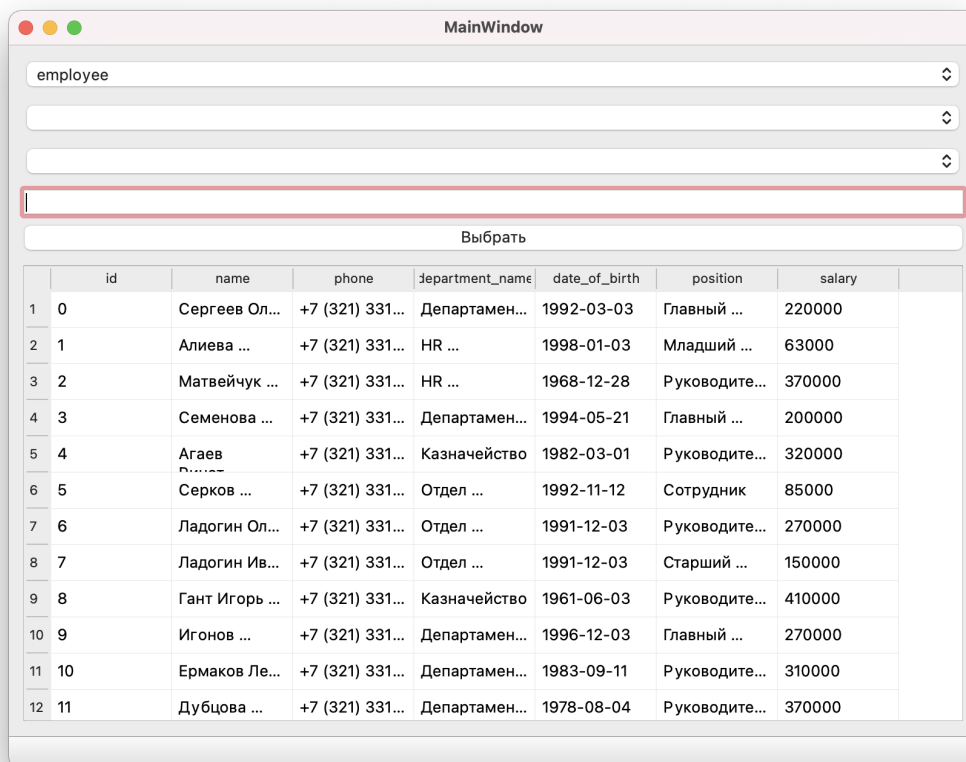
    ui->comboBox_3->addItem({ "<", "<=", "=", ">", ">=", "BETWEEN", "NOT BETWEEN", "CUSTOM",
    "LIKE"});

    ui->comboBox_3->setCurrentIndex(-1);
    ui->comboBox_2->addItem(hshTableNameColumns["employee"]);
    ui->comboBox_2->setCurrentIndex(-1);

    connect(ui->comboBox, &QComboBox::currentTextChanged, this, [model, this](const QString&
currentText) {
        model->setTable(currentText);
        if (currentText == QLatin1String("employee")) {
            model->setRelation(3, QSqlRelation("departments", "id", "department_name"));
            model->setRelation(5, QSqlRelation("grades", "id", "position"));
        }
        model->select();
        ui->comboBox_2->clear();
        ui->comboBox_2->addItem(hshTableNameColumns[currentText]);
        ui->comboBox_2->setCurrentIndex(-1); // Не выбирать первое значение
    });
    ...
}

```

Посмотрим на результат работы программы:



Теперь названия отделов и грейдов стали читаемыми, но при редактировании этих колонок мы не можем редактировать значения, для этого нам понадобится делегат, который, к тому же, существенно упростит работу пользователя.

Добавим заголовочный файл и следующую строку после выбора таблицы.

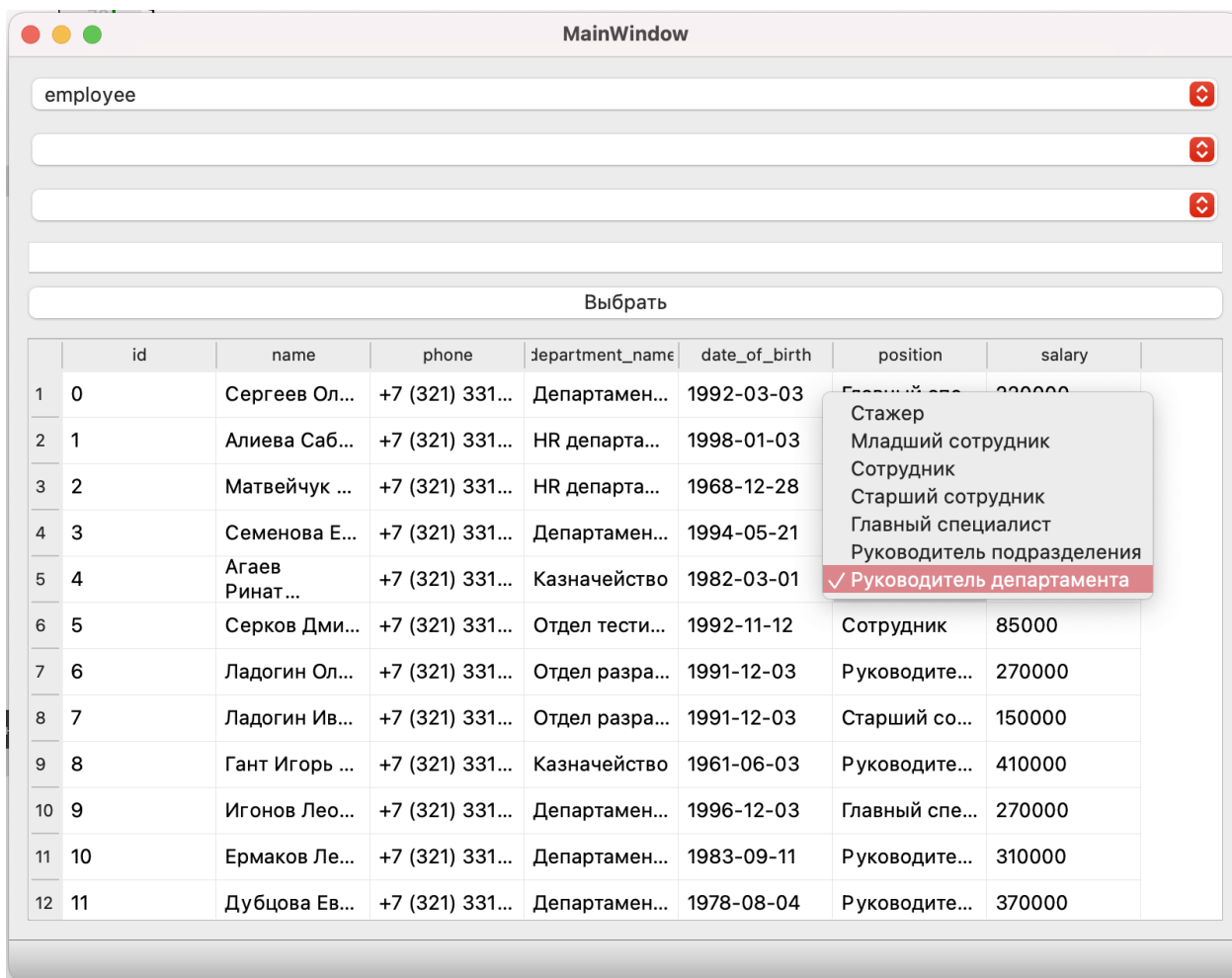
```
ui->tableView->setItemDelegate(new QSqlRelationalDelegate(ui->tableView));
```

Теперь посмотрим на результат:

employee

Выбрать

	id	name	phone	department_name	date_of_birth	position	salary
1	0	Сергеев Ол...	+7 (321) 331...	✓ Департамент веб-разработки			
2	1	Алиева Саб...	+7 (321) 331...	HR департамент			
3	2	Матвейчук ...	+7 (321) 331...	Казначейство			
4	3	Семенова Е...	+7 (321) 331...	Департамент маркетинга и стратегического планирования			
5	4	Агаев Ринат ...	+7 (321) 331...	Департамент DevOps			
6	5	Серков Дми...	+7 (321) 331...	Отдел тестирования			
7	6	Ладогин Ол...	+7 (321) 331...	Отдел разработки мобильных и десктоп приложений			
8	7	Ладогин Ив...	+7 (321) 331...	Отдел тести...	1992-11-12	Сотрудник	85000
9	8	Гант Игорь ...	+7 (321) 331...	Отдел разра...	1991-12-03	Руководите...	270000
10	9	Игонов Лео...	+7 (321) 331...	Отдел разра...	1991-12-03	Старший со...	150000
11	10	Ермаков Ле...	+7 (321) 331...	Казначейство	1961-06-03	Руководите...	410000
12	11	Дубцова Ев...	+7 (321) 331...	Департамен...	1996-12-03	Главный спе...	270000
				Департамен...	1983-09-11	Руководите...	310000
				Департамен...	1978-08-04	Руководите...	370000



Теперь при попытке редактировать эти значения, у нас будут появляться удобные комбобоксы.

Сборка и подключение других библиотек баз данных (PostgreSQL/MySQL/ETS)

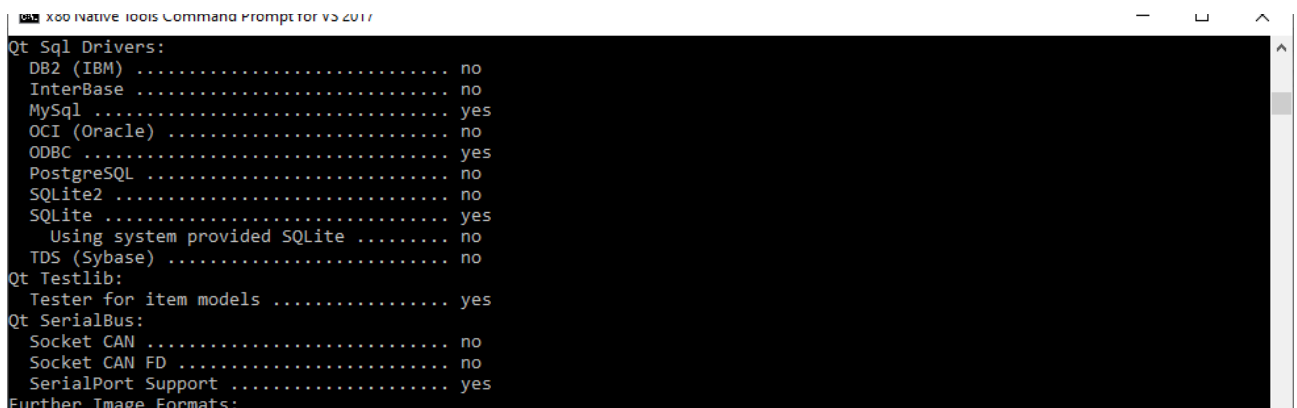
Для подключения БД необходимо с драйвером по работе с выбранной БД. Драйвер БД указывается так: `var = QSqlDatabase::addDatabase("<Драйвер БД>");`

Строка названия драйвера	Используемая БД
QDB2	IBM DB2 (версия 7.1 и новее)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Драйвер интерфейса Oracle

QODBC	MS SQL
QPSQL	Postgresql с версии 7.3
QSQLITE2	SQLite 2.0
QSQLITE	SQLite 3.0

Драйверы БД можно добавлять в фреймворк Qt или удалять из него путем пересборки библиотек.

1. Запустим консоль (под ОС Windows запускать консоль через меню Пуск -> Qt -> Qt (<необходимый компилятор>)).
2. Скачать SDK БД для компиляции драйвера ([MySQL](#), [PostgreSQL](#)).
3. Переходим в каталог с исходным кодом Qt.
4. Выполняем переконфигурацию, если используется собранная версия Qt (для статической версии), или конфигурируем под нужную платформу: **configure -prefix <путь к каталогу Qt для нужной платформы> -platform <платформа/компилятор> -I <путь к заголовочным файлам БД> -L <путь к статическим библиотекам БД> -sql-mysql (и/или -sql-psql для PostgreSQL)**. Если нужна переконфигурация (проект уже был сконфигурирован ранее), необходимо добавить параметр **-recheck-all**.
5. Если в качестве БД использовалась MySQL, то в терминале напротив MySQL должно высвечиваться **yes**:



```

x86 Native Tools Command Prompt for VS 2017

Qt Sql Drivers:
DB2 (IBM) ..... no
InterBase ..... no
MySQL ..... yes
OCI (Oracle) ..... no
ODBC ..... yes
PostgreSQL ..... no
SQLite2 ..... no
SQLite ..... yes
Using system provided SQLite ..... no
TDS (Sybase) ..... no
Qt Testlib:
Tester for item models ..... yes
Qt SerialBus:
Socket CAN ..... no
Socket CAN FD ..... no
SerialPort Support ..... yes
Further Image Formats:

```

6. Переходим в консоли к каталогу <исходный код Qt>/qtbase/src/plugins/sqldrivers/mysql (psql).
7. Запускаем утилиту: **qmake "INCLUDEPATH += <путь к заголовочным файлам БД>" "LIBS += <путь к статическим библиотекам БД>/<основная библиотека>" mysql.pro -o Makefile**. Основной библиотекой для БД MySQL является **libmysql**: для Windows — **libmysql.lib**, для UNIX-подобных — **libmysql.a**.
8. Выполняем сборку: **make -f Makefile.Release**. Для Windows в зависимости от компилятора — **nmake** или **mingw32-make**.
9. Устанавливаем драйвер: **make (nmake, mingw32-make) install**.

Практическое задание

Продолжаем работать с проектом Планировщик задач.

1. Сохранение задач реализовать в базе данных, а не в файле.
2. На основное окно добавить кнопку для просмотра задач. При нажатии на эту кнопку должно открыться новое окно, в котором в виде таблицы будут отображены все задачи. Окно может быть создано при помощи стандартных средств qt (не qml).

Дополнительные материалы

1. [SQL-запросы](https://tproger.ru/translations/sql-recap/). <https://tproger.ru/translations/sql-recap/>
2. [Официальная документация по классу QSqlDatabase](https://doc.qt.io/qt-5/qsqldatabase.html). <https://doc.qt.io/qt-5/qsqldatabase.html>
3. [Официальная документация по классу QSqlQuery](https://doc.qt.io/qt-5/qsqldbquery.html). <https://doc.qt.io/qt-5/qsqldbquery.html>
4. [Установка драйвера SQL](https://doc.qt.io/qt-5/sql-driver.html). <https://doc.qt.io/qt-5/sql-driver.html>

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [SQL-запросы](#).
2. [Официальная документация по классу QSqlDatabase](#).
3. [Официальная документация по классу QSqlQuery](#).
4. [Установка драйвера SQL](#).