

# **Лабораторные работы по курсу Базы данных**

**Работа с программой ORIOKS simulator**

**Москва, 2025**

## Оглавление

1. Теоретическая часть .....	3
1.1. Подключение к серверу PostgreSQL .....	3
1.2. SQLQuery .....	3
1.3. Prepare и bind value.....	4
1.4. Обработка результата запроса .....	4
1.5. Таблицы, таблицы и еще раз таблицы .....	5
1.6. Создание виджета.....	5
1.7. Делегаты.....	6
1.8. Добавление новой записи в таблицу .....	7
1.9. Очистка таблицы .....	7
1.10. Базовые структуры.....	7
1.10.1. User.....	7
1.10.2. Subject .....	7
1.11. Основные возможности приложения .....	8
1.11.1. Форма логина .....	8
1.11.2. Интерфейс студента.....	9
1.11.3. Интерфейс преподавателя.....	10
1.11.4. Интерфейс методиста .....	13
1.12. SQL-инъекции .....	14
1.13. Защита от SQL-инъекций .....	15
2. Практическая часть.....	16
2.1. Задание 1. ....	16
2.2. Задание 2. ....	16
2.3. Задание 3. ....	16
Список литературы .....	16

## 1. Теоретическая часть

Эмулятор ОРИОКСа является демонстрационным примером взаимодействия базы данных с программным обеспечением. Данное ПО было написано при помощи фреймворка Qt. Архитектура приложения выстраивалась с использованием объектно-ориентированного подхода (ООП).

Для установки необходимых библиотек воспользуйтесь командами:

```
sudo apt install -y libpq-dev
sudo apt install -y qtcreator qtbase5-dev qt5-qmake cmake
sudo apt install -y libqt5sql5-psql
```

### 1.1. Подключение к серверу PostgreSQL

Для того чтобы работать с PostgreSQL из под Qt, возможно использовать библиотеку `psql`. Рассмотрим, как реализован "мост" между сервером PostgreSQL и Qt. Реализация описана в файлах `sqlservice.h` и `sqlservice.cpp`. Для того чтобы была возможность работать с базой данных, подключается следующий заголовочный файл:

```
#include <QSqlDatabase>
```

Подключение к серверу PostgreSQL и, соответственно, к базе данных, производится в теле данной функции:

```
void databaseConnect(); // Функция подключения к базе данных
```

Внутри данной функции производится инициализация объекта с именем **db** типа `QSqlDatabase`:

```
db = QSqlDatabase::addDatabase("QPSQL"); // Инициализирую базу данных при помощи "QPSQL"
```

Через данный объект и будет производиться установка соединения с базой данных на сервере. Далее у данного объекта производится установка параметров подключения через методы `setHostName`, `setDatabaseName`, после чего при помощи метода `open()` производится попытка установить соединение. Данный метод имеет тип `bool` и возвращает `true` или `false`. Именно этим фактом я воспользовался, для того чтобы проверить, установилось ли соединение с БД:

```
if (!db.open()) {
    qDebug() << "Failed to connect to database.";
    qDebug() << "Error: ";
    qDebug() << db.lastError().text();
} else {
    qDebug() << "Connected to database succesfully!";
}
```

### 1.2. SQLQuery

Работа с базой данных подразумевает выполнение SQL-запросов. Такая возможность реализована в данном примере. Следующий заголовочный файл является частью библиотеки `psql` и позволяет выполнять SQL-запросы:

```
#include <QSqlQuery> // query (от англ.) - очередь, запрос
```

В файле `sqlservice.h` объявлена функция `runQuery`, внутри которой описан пример логики выполнения SQL-запроса. Ниже приведен ее код с комментариями, подробно описывающими суть происходящего:

```
QSqlQuery SqlService::runQuery(QString content) {

    // Пример выполнения SQL-запроса

    QSqlQuery query;

    // Объект класса QSqlQuery позволяет выполнять SQL-запросы,
    // хранит в себе "таблицу", полученную после sql-запроса

    query.exec(content); // Метод exec() принимает на вход текст запроса
```

```

// Смотрим, выполнен ли запрос успешно:

if (query.lastError().isValid()) {
    qDebug() << "Ошибка выполнения запроса:" << query.lastError().text();
} else {
    // Запрос выполнен успешно
}

return query; // Возвращает объект очереди для дальнейшей обработки результата запроса
}

```

Конкретно данная функция в ходе работы приложения использоваться не будет и приведена лишь в ознакомительных целях. Однако, концепция выполнения запроса будет именно такая. О том, как именно будут выполняться запросы и как будет производиться **обработка** результата запроса рассказано ниже.

### 1.3. Prepare и bind value

Для выполнения запроса необходимо выполнить команду:

```
query.exec(content); // Метод exec() принимает на вход текст запроса
```

При этом текст запроса может иметь примерно такой вид:

```

SELECT password, access_level
FROM users
WHERE login = '123456'

```

Здесь четко указано, что необходимо найти все записи, где значение логина — 123456. Однако, в большинстве случаев заранее неизвестно, какое именно значение параметра нам нужно. Такой запрос не является гибким. Соответственно, возникает проблема, связанная с необходимостью формировать запрос таким образом, чтобы в него можно было подставить нужное значение параметра поиска?

На помощь приходит метод *prepare()* класса *QSqlQuery*. Рассмотрим пример его применения (взят из *loginform.cpp*):

```

QSqlQuery login_query;
login_query.prepare("SELECT password, access_level FROM users "
                  "WHERE login = :login");
// Подготавливаем запрос

```

В данном случае, вместо **:login** возможно подставить требуемое значение при помощи метода *bindValue()*:

```

QString requested_login = ... // Какая-то строка с данными
login_query.bindValue(":login", requested_login); // Биндим

```

По такой же логике переменных может быть несколько и для каждой устанавливается значение при помощи *bindValue()*.

### 1.4. Обработка результата запроса

В прошлом разделе был указан способ передать значения в качестве параметра SQL запроса. Кроме ввода данных необходимо получать результат запроса и корректно его обрабатывать в программе

Напомним, для того чтобы сделать запрос в простейшем случае необходимо выполнить следующую последовательность команд:

```

QString content = '...';
QSqlQuery query;
query.exec(content);

```

После запуска запроса внутри объекта *query* будет храниться **результат выполнения запроса**. Возможно 3 сценария:

- Пустой ответ из-за ошибки
- Пустой ответ, так как нет ни одной удовлетворяющей запросу записи в таблице
- Получен ответ в виде специально сформатированной таблицы.

Для того, чтобы привести результат выполнения запроса к более простому виду, обычно используется следующая конструкция (см. *'subjectselection.cpp'*)

```

QSqlQuery groups_query;
QString subject;
groups_query.prepare("Текст запроса с переменной :subject");

```

```
groups_query.bindValue(":subject", subject);
groups_query.exec();

// Парсим результат
// Получаем строки из таблицы, которая вернулась после запроса
// Делаем это до тех пор, пока не дойдем до конца таблицы

while (groups_query.next()) {
    QString group_name = groups_query.value(0).toString();
}
```

При помощи метода *next()* внутри объекта *groups\_query* происходит смещение внутреннего итератора на следующую строку возвращенной после запроса таблицы. Метод *value()* нужен для того чтобы получить конкретное значение из текущей строки. Например, в примере выше ***groups\_query.value(0)*** вернет значение из первого столбца.

### 1.5. Таблицы, таблицы и еще раз таблицы

Для того чтобы представлять результат SQL запроса в графическом интерфейсе, необходим механизм, который будет представлять данные в табличном формате. В Qt представление данных в виде таблицы можно реализовать при помощи *QTableWidget*. Ниже приведен пример отображения списка дисциплин, полученного из SQL-запроса при помощи виджета класса *QTableWidget*:

Имя	Дисциплина	Код	Журнал
Широков	Контроль и диагностика	ИВТ-21В	Журнал
Широков	Контроль и диагностика	ИВТ-41	Журнал
Широков	Контроль и диагностика	ИВТ-42	Журнал
Широков	Контроль и диагностика	ИВТ-43	Журнал
Хисамов	Электротехника	ИВТ-41	Журнал
Хисамов	Электротехника	ИВТ-42	Журнал
Хисамов	Электротехника	ИВТ-43	Журнал
Козин	Программируемые логические интегральные схемы	ИВТ-41	Журнал
Козин	Программируемые логические интегральные схемы	ИВТ-42	Журнал

В процессе работы с данной программой Вы не раз встретите подобные таблицы (очень много раз), и необходимо разобраться, как с ними работать.

### 1.6. Создание виджета

Поскольку это программное обеспечение создавалось в Qt Creator, в котором есть встроенный Qt Designer, размещение необходимых виджетов сильно упрощено. Достаточно просто найти виджет *QTableWidget* на панели слева и перетянуть его на форму, а не создавать данный виджет вручную.

После того, как виджет размещен на форму, необходимо дать ему имя, поскольку виджет — это тоже объект, также как и любой другой объект класса или переменная. Для

этого необходимо выбрать виджет таблицы в списке виджетов формы справа или нажать на него в форме:

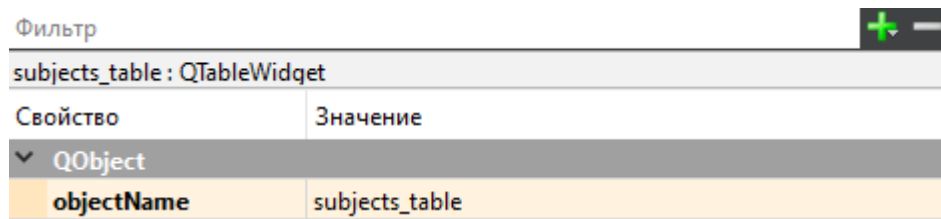


Рисунок 1 Назначение имени виджету

В данном примере атрибут **objectName** хранит в себе имя таблицы (объекта).

После этого, когда был создан виджет таблицы, необходимо заполнить его данными. Однако, перед этим требуется настроить его поведение – указать определенные параметры, например количество столбцов, имена столбцов, высоту ячейки, равномерное растяжение таблицы по ширине и др. На примере таблицы выше рассмотрим, как это сделать. Данная таблица взята из файла класса **subjectmoderation.h**. Внутри класса описан следующий метод:

```
void SubjectsModeration::configureTableParameters() {
    // Устанавливаю количество столбцов
    ui->subjects_table->setColumnCount(4);

    // Отключаю отображение названий столбцов
    ui->subjects_table->horizontalHeader()->setVisible(false);
    ui->subjects_table->verticalHeader()->setVisible(false);

    // Отключаю отображение сетки
    ui->subjects_table->setShowGrid(false);

    ... // Часть функции временно опущена
}
```

Для того чтобы получить доступ к объекту таблицы, который был размещен на форме, используется приватный атрибут класса `ui`.

В приведенном выше фрагменте кода таблицы устанавливается 4 столбца, название столбцов было отключено за ненадобностью, также была отключена сетка по умолчанию.

Для того чтобы содержимое таблицы автоматически подгонялось под размер окна по горизонтали, используется следующая команда:

```
ui->subjects_table->horizontalHeader()->setSectionResizeMode(номер_столбца,
QHeaderView::Stretch); // Выставляем QHeaderView::Stretch для каждого столбца
```

## 1.7. Делегаты

Для того, чтобы настроить гибкое поведение таблицы, иногда недостаточно вызвать какой-либо метод таблицы и что-либо туда передать. Например, при разработке данного приложения, возникла следующая проблема: нужно было сделать так, чтобы текст в каждой ячейке данного столбца выравнивался по вертикали. Оказалось, что такая тривиальная задача не имеет тривиального решения. Один из способов - пройти по всем ячейкам столбца в цикле и установить выравнивание ячейки по центру, однако такой подход с некоторой точки зрения является “костыльным”. Другим способом является использование так называемых **делегатов**.

Делегат — это специальный класс, позволяющий кастомизировать поведение таблицы, ячеек, групп ячеек и т.д

При установке делегата для столбца для каждой его ячейки применяются правила, описанные внутри делегата. Например:

```
ui->subjects_table->setItemDelegateForColumn(0, new CenterAlignmentDelegate());
```

Здесь для первого столбца таблицы (нулевой индекс - первый столбец) устанавливается делегат класса `CenterAlignmentDelegate`. Как уже можно догадаться из названия, данный делегат выравнивает содержимое ячеек таблицы по центру.

Ознакомиться со всеми имеющимися делегатами можно ознакомиться в папке *tables\_stuff*.

### 1.8. Добавление новой записи в таблицу

Для того чтобы добавить новые записи в таблицу, необходимо выполнить следующие действия:

1. При помощи *insertRow()* вставить новую строку в таблицу;
2. Заполнить содержимое строки, обратившись к каждой ячейке в строке при помощи метода *setItem()* или *setCellWidget()* (если нужно установить виджет в ячейку, например, кнопку).

Пример:

```
int rows = 0; // Счетчик количества строк в таблице

while (subjects_query.next()) {
    ui->subjects_table->insertRow(rows);
    ui->subjects_table->setRowHeight(rows, 50); // Высота

    ...

    ui->subjects_table->setItem(rows, 0, new QTableWidgetItem(teacher_surname));
    ui->subjects_table->setItem(rows, 1, new QTableWidgetItem(subject_name));
    ui->subjects_table->setItem(rows, 2, new QTableWidgetItem(group_name));
    ui->subjects_table->setCellWidget(rows, 3, button_container);
}
```

Поскольку данные в ячейках таблицы должны иметь тип *QTableWidgetItem* или тип *Widget* (кнопки и т.д), при вставке нового элемента создается объект типа *QTableWidgetItem*, в конструктор класса передается, как правило, строка *QString* с текстом, который мы хотим видеть в данной ячейке.

### 1.9. Очистка таблицы

У таблицы *QTableWidget* нет метода *clear()*. Одним из способов удалить все записи из таблицы - установить количество строк равное нулю:

```
ui->subjects_table->setRowCount(0);
```

### 1.10. Базовые структуры

Для удобства и структуризации некоторые сущности были выделены в отдельные структуры.

#### 1.10.1. User

Данная структура создана для того, чтобы удобно хранить информацию о текущем пользователе системы (информацию о ФИО, номере студенческого билета (для студентов), ID преподавателя (для преподавателей), уровне учетной записи и т.д). Описание данной структуры можно найти внутри файла **user.h**.

#### 1.10.2. Subject

Данная структура создана для того, чтобы удобно хранить информацию о каком-либо предмете из базы данных (название предмета, его id). Для корректной работы, внутри класса переопределен оператор <:

```
bool operator< (const Subject& other) const {
    if (subject_id < other.subject_id)
        return true;
    return false;
}
```

Это сделано для того, чтобы можно было использовать сортировку по отношению к предметам по номеру предмета в системе.

## 1.11. Основные возможности приложения

Теперь рассмотрим основные возможности симулятора ОРИОКСа. На рисунке 2 представлена общая схема программы:

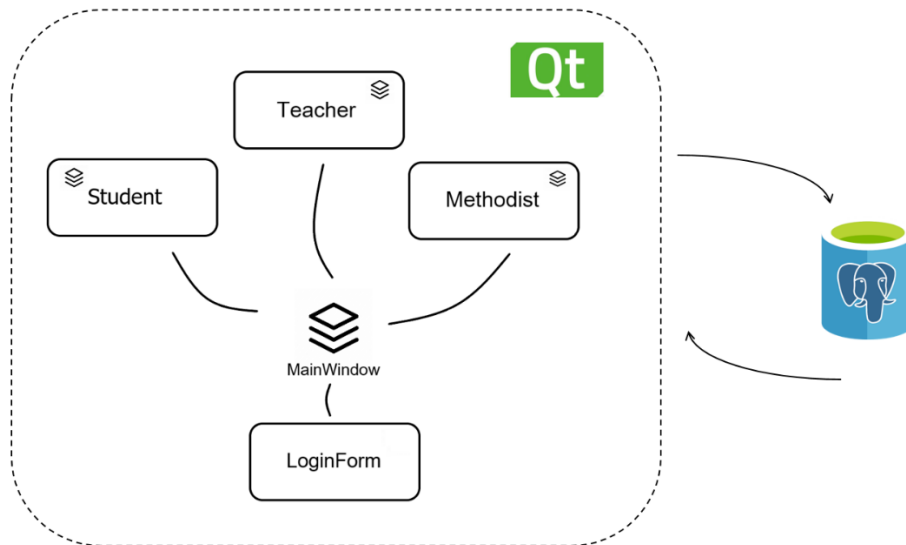


Рисунок 2 Схема программы

ПО представляет из себя следующее: в основе интерфейса лежит класс *MainWindow*, который отображает в текущий момент времени один элемент из списка:

- форму логина;
- интерфейс студента;
- интерфейс преподавателя;
- интерфейс методиста.

Это реализуется в следующем фрагменте кода из `mainwindow.cpp`:

```
stackedWidget = new QStackedWidget();

// Добавляю все виджеты в контейнер QStackedWidget:
stackedWidget->addWidget(login_form_);
stackedWidget->addWidget(teacher_interface_);
stackedWidget->addWidget(student_interface_);
stackedWidget->addWidget(methodist_interface);

// Устанавливаю текущий виджет
stackedWidget->setCurrentWidget(login_form_);
```

Для того чтобы приблизить программу к форме веб-страницы, возможно использовать специальный виджет класса *QStackedWidget*. На структурной схеме выше его использование указывается при помощи значка стека. Все, что делает данный виджет - хранит в себе другие виджеты и показывает выбранный пользователем. Таким образом, *QStackedWidget* является контейнером для других виджетов. На таком принципе так же построен и класс *Student*, класс *Teacher*, класс *Methodist*. Например, в классе *Teacher* так же, как и в *MainWindow* есть глобальный виджет *QStackedWidget*, который отображает один из следующих виджетов:

- виджет выбора дисциплины
- виджет выбора группы
- виджет выбора журнала
- виджет должников.

### 1.11.1. Форма логина

Как и в оригинальном, первым делом нас встречает форма логина, где необходимо ввести логин и пароль от учетной записи.



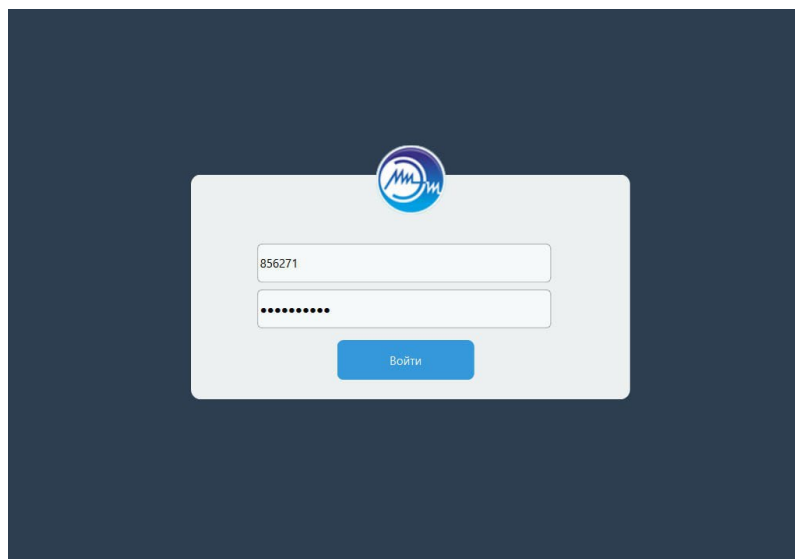


Рисунок 3 Форма ввода логина и пароля

Реализация формы логина описана в следующих классах:

- *loginform.h*;
- *loginform.cpp*;
- *loginform.ui*.

Рассмотрим её чуть более подробно.

Для того чтобы соотнести уровень доступа учетной записи с ее привилегиями используется следующее перечисление:

```
enum AccessLevel {
    student, teacher, methodist
};
```

Также внутри формы логина хранится информация о текущем пользователе в атрибуте типа *User*:

```
User current_user_;
```

Внутри конструктора класса происходит связь кнопки авторизации и запуска функции обработки запроса на авторизацию при помощи функции *connect*. Также внутри конструктора в секции кода «*debug purposes only*» содержатся данные от 3 учетных записей с разными уровнями доступа: студента, преподавателя и методиста.

Внутри функции *authoriseRequested()* происходит обработка запроса аутентификации при помощи SQL-запроса. Принцип работы следующий:

- Из полей ввода логина и пароля производится сбор данных;
- Выполняется SQL запрос для получения настоящего пароля по данному логину;
- Полученный посредством запроса пароль сравнивается с введенным.

#### 1.11.2. Интерфейс студента

Интерфейс студента состоит, по сути, из единственной вкладки - **обучение**. Выглядит он следующим образом:

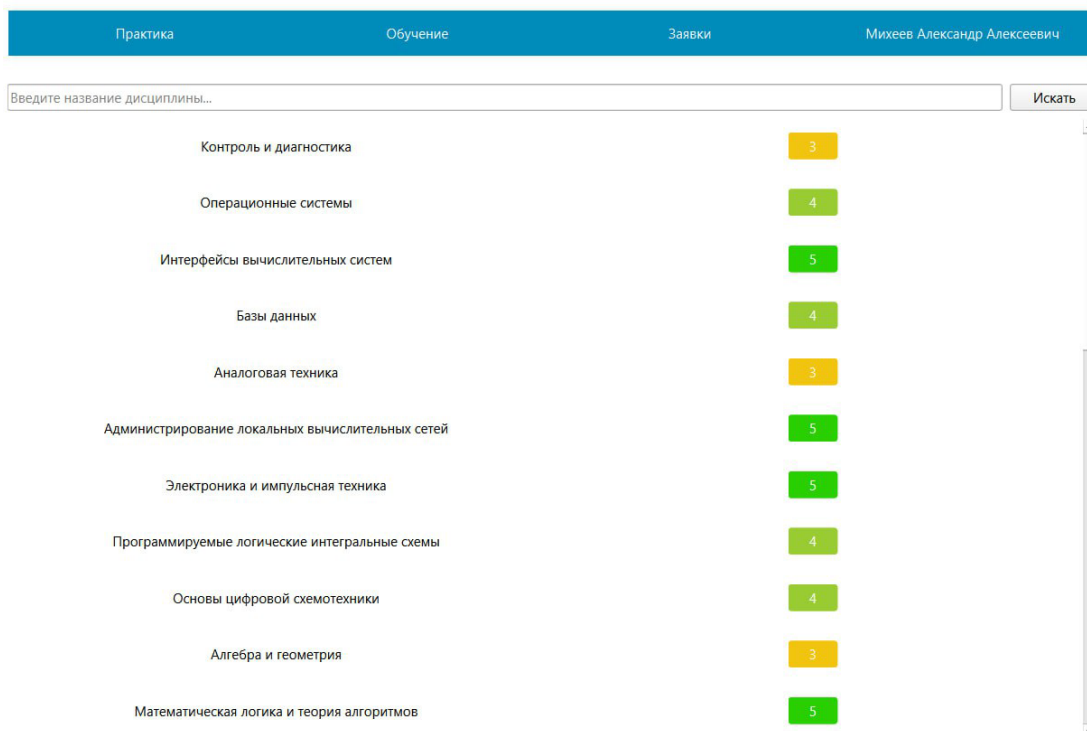


Рисунок 4 Интерфейс студента

Более подробно рассмотрим его устройство. Виджет, который отвечает за отображение дисциплин и оценок представлен классом *Diary*. По нажатию на кнопку “Обучение” данный виджет устанавливается текущим виджетом контейнера *QStackedWidget*:

```
void Student::openDiary()
{
    diary_ -> loadMarks();
    ui -> stackedWidget -> setCurrentWidget(diary_);
}
```

Внутри объекта класса *Diary* при помощи метода *setStudent()* устанавливается текущий ученик. Далее при помощи метода *loadMarks()* выполняется SQL-запрос на оценки данного студента из базы данных, после чего результат запроса передается в функцию *fillDiaryMarks()*, где результат обрабатывается и выводится на экран.

Для того чтобы оценка отрисовывалась в цветном прямоугольнике, используется метод *createMarkBorder()*, создающий такой прямоугольник через *QLabel*, после чего при помощи хитрости с использованием *QLayout* содержимое оцентровывается и отрисовывается.

### 1.11.3. Интерфейс преподавателя

Интерфейс преподавателя можно разбить на 4 части:

- выбор дисциплины;
- выбор группы по заданной дисциплине;
- журнал;
- должники (вам предстоит доработать данный раздел самостоятельно).

Рассмотрим их подробнее.

#### ▪ Выбор дисциплины

Ниже представлен раздел с выбором дисциплины:

Дисциплина:

Электроника и импульсная техника

ИБТ-41, ИБТ-42, ИБТ-43

Электротехника

ИБТ-41, ИБТ-42, ИБТ-43

Рисунок 5 Интерфейс выбора дисциплины

Рассмотрим бэкенд данной процедуры. Он описан внутри файла **bjectselection.cpp** функцией **fillSubjectsTable()**:

```
void SubjectSelection::fillSubjectsTable()
{
    QSqlQuery subjects_query;
    subjects_query.prepare("SELECT field_name, field_id "
                          "FROM field "
                          "WHERE professor_id = :id");

    subjects_query.bindValue(":id", current_user_.getUserId());

    // заполнение таблицы данными
    subjects_query.exec();

    ... // Далее идет парсинг полученного ответа на запрос и заполнение таблицы
}
```

Для того чтобы собрать все группы относящиеся к данной дисциплине воедино, используется словарь:

```
std::map<QString, vector<QString>> subject_links;
```

Следующая задача - сделать так, чтобы по нажатию на кнопку с названием соответствующей дисциплины открывался следующий по уровню вложенности виджет класса `GroupSelection` с соответствующими группами для выбора.

Делается это при помощи следующего "трюка":

```
// Создаем кнопку
QPushButton* group_list_link = new QPushButton(groups, ui->tableWidget);

// Устанавливаем прозрачный фон для кнопки через CSS
group_list_link->setStyleSheet("background-color: transparent; border: none; "
                              "color: black; text-align: left;");

// Связываю клик по созданной кнопке с функцией обработки клика через лямбда - функцию:

connect(group_list_link, &QPushButton::clicked, this, [i, this]() {
    handleGroupListRequest(i);
});

...

// Функция, которая вызывается после клика на кнопку благодаря connectу ранее:

void SubjectSelection::handleGroupListRequest(int row)
{
    QString subject_name = ui->tableWidget->item(row, 0)->text();
    QString subject_id = subjects_id_[subject_name];
    Subject current_subject{subject_name, subject_id};

    ...

    emit subjectSelected(current_subject);
    emit groupListRequested(groups);
}
```

## ▪ Выбор группы

Раздел с выбором группы выглядит следующим образом:

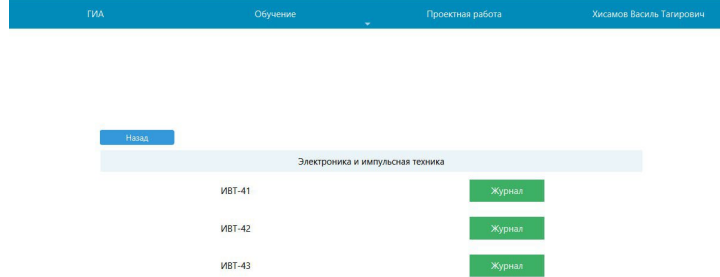


Рисунок 6 Интерфейс выбора группы

Заполнение таблички с группами происходит внутри следующей функции:

```
void GroupSelection::fillGroups(QStringList groups)
{
    ...
}
```

Семантика бэкенда похожа на бэкенд выбора дисциплины, за исключением того, что на этапе выбора группы уже не производится sql-запрос, а просто берется полученный ранее список групп, переданный параметром через QStringList и отображается на экране.

## ▪ Журнал

Так выглядит журнал с перспективы учителя (впрочем, у методиста он выглядит точно также):

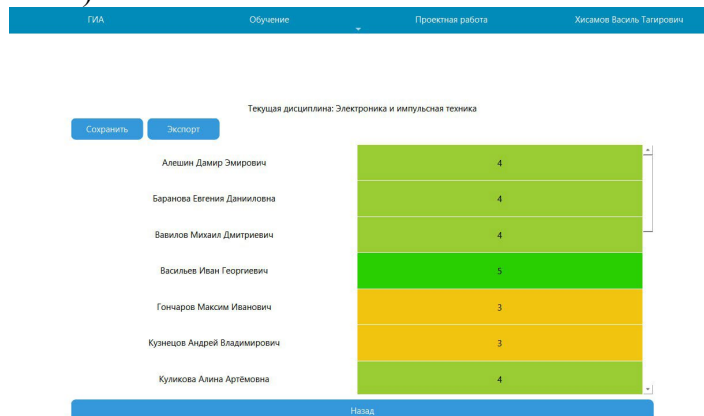


Рисунок 7 Интерфейс журнала

Реализация журнала представлена в классе 'Journal'.

Его логика работы следующая:

1. Фиксируется группа, для которой будет открыт журнал:

```
void Journal::openGroup(QString group)
{
    ...
    // Переходим ко второму этапу, заполняем журнал в графическом интерфейсе:
    fillMarks(std::move(query));
}
```

2. Открывается журнал (заполняется таблица студентов и оценок посредством парсинга результата SQL-запроса) с возможностью редактировать оценки:

```
void Journal::fillMarks(QSqlQuery marks_record)
{
    // Берем полученный объект QSqlQuery и получаем из него
    // информацию, как и обычно:
```

```

int rows = 0;

ui->journal->setRowCount(0);

// Заполняю таблицу с оценками
while (marks_record.next()) {
    ... // Здесь происходит заполнение таблицы

    ++ rows;
}
}

```

3. После внесения изменений по кнопке сохранить производится обновление оценок на сервере через SQL-запрос:

```

void Journal::updateDatabaseGrades()
{
    // Обновление данных об оценках на сервере
    // В цикле прохожу по всем оценкам и перезаписываю их
    // на сервере исходя из айди студента:

    QSqlQuery update_query;
    for (int row = 0; row < ui->journal->rowCount(); row++) {
        int mark = ui->journal->item(row, 1)->text().toInt();
        int student_id = students_id_[row];
        ...
    }
    qDebug() << "Marks list updated!";
}

```

По аналогичной концепции можно реализовать и раздел с должниками.

#### 1.11.4. Интерфейс методиста

Учетная запись методиста отличается от учетной записи преподавателя тем, что методист имеет доступ к любому предмету с возможностью править оценки.

##### ▪ Выбор дисциплины

Интерфейс выбора дисциплины выглядит следующим образом:

Рисунок 8 Интерфейс выбора дисциплины у методиста

Реализация такого интерфейса несколько сложнее реализации аналогичного интерфейса преподавателя по понятным причинам. Рассмотрим, как именно формируется табличка с предметами по заданным фильтрам.

Реализация данного функционала описана внутри класса **SubjectModeration**.

Прежде всего, при помощи SQL-запросов заполняются опции института, группы и преподавателя. За это ответственны следующие функции:

```

void fillTeachers();
void fillStructuralUnits();
void fillGroups();

```

По нажатию на кнопку “искать” производится SQL-запрос на дисциплины по заданным фильтрам, после чего найденные дисциплины представляются в виде таблицы. Реализация данного функционала описана внутри функции *searchSubjects()*. Ее суть

заключается в том, что формируется один большой запрос, который формируется исходя из выбранных условий фильтрации, условия добавляются в конец запроса при помощи оператора **AND**.

#### ▪ Назначение преподавателя

Еще одной особенностью методиста является возможность назначить преподавателя для какой-либо дисциплины:

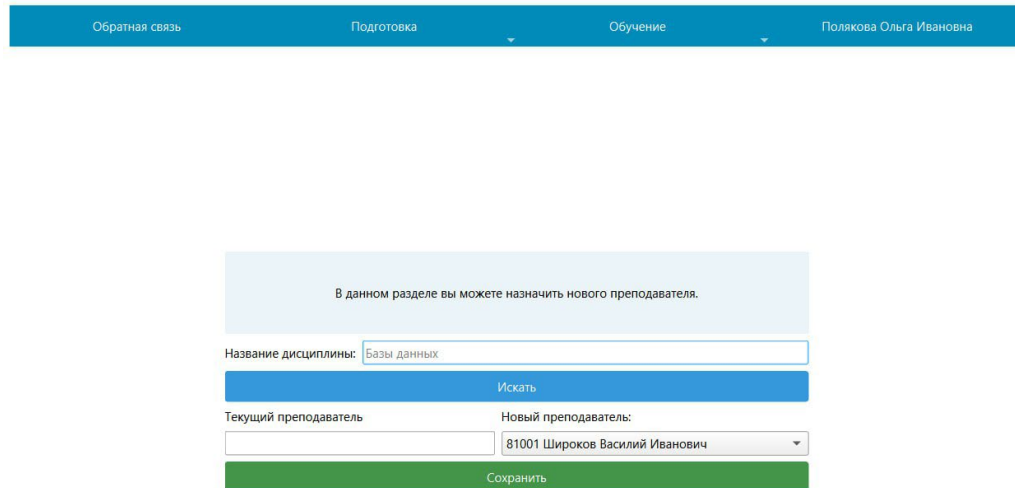


Рисунок 9 Интерфейс назначения преподавателя

Реализация данного функционала описана внутри класса `TeachersModeration`.

Функция *searchByDiscipline()* запускается по нажатию на кнопку **“Искать”**. Суть ее работы заключается в том, что из *lineedita* берется название введенной дисциплины и формируется SQL-запрос, который ищет преподавателя данной дисциплины.

После того, как выбран новый преподаватель и нажата кнопка **“Сохранить”**, необходимо обновить 2 параметра:

1. Таблицу предметов: установить нового преподавателя
2. Таблицу найма преподавателей: установить новый институт преподавателю при необходимости в зависимости от дисциплины.

Реализация данного функционала описана здесь:

```
void updateFieldTable();  
void updateEmploymentTable();
```

## 1.12. SQL-инъекции

Рассмотрим функцию, позволяющую производить поиск по названию предмета. Обратите внимание, что в данном случае вызов запроса идет напрямую, без использования методов *prepare* и *bindValue*.

```
void Diary::searchBySubject()  
{  
    QSqlQuery search_query;  
    QString query = "select field_name, mark FROM field_comprehension "  
                   "LEFT OUTER JOIN Field "  
                   "ON Field_comprehension.field = Field.field_id "  
                   "where student_id =" + QString::number(current_student_.getUserId()) + " AND  
field_name = '" + QString(ui->search_line->text()) + "'";  
    search_query.exec(query);  
  
    if (search_query.size() == 0) {  
        ui->search_line->clear();  
        ui->search_line->setPlaceholderText("Предметы не найдены!");  
    } else {
```

```

    fillDiaryMarks(std::move(search_query));
}
}

```

При корректном вызове функции программа выберет из базы данных необходимую дисциплину и выведет ее на экран.

Однако, злоумышленники могут вместо требуемых значений могут заведомо вводить ложные, чтобы получить доступ к закрытым данным. Для этого в поле «Поиск предмета» возможно внести строку, следующего содержания: «*1' OR 1=1; update field\_comprehension set mark = 5 where student\_id=856271 ;--*». Введите данную строку и нажмите на кнопку поиска. Постарайтесь самостоятельно понять, что произошло и объяснить полученный результат. Если не удастся – в следующем пункте находится объяснение.

### 1.13. Защита от SQL-инъекций

Защититься от SQL-инъекции возможно несколькими способами. Правильным является использование ORM библиотек или методов из семейства *prepare* и *bindValue*. Однако произвести защиту можно и другими способами. Например, если вводимое значение является числовым, то возможно сделать простейшую на это проверку.

```

bool isNumeric(std::string const &str)
{
    return !str.empty() && std::all_of(str.begin(), str.end(), ::isdigit);
}

```

В случае, если введенное ID является строкой, функция *isNumeric* вернет значение 0, что возможно будет отследить и вывести сообщение о попытке взлома программы.

Более интересным является случай защиты при вводе строки. Легко заметить, что предыдущий способ в случае примера выше не сработает. Чтобы совершить атаку, вводимая строка должна содержать одинарные кавычки. Первая кавычка закроет вводимую строку, далее произойдет ввод вредоносного кода, который также должен завершиться выражением, требующим закрывающую кавычку. В примере ниже **синим** цветом выделен код функции, а **красным** – код, вводимый злоумышленником.

```

select field_name, mark FROM field_comprehension
    LEFT OUTER JOIN Field
    ON Field_comprehension.field = Field.field_id
    where student_id = ID AND field_name = '1' OR 1=1; update
field_comprehension set mark = 5 where student_id=856271 ;--'

```

Защититься от такой инъекции можно двумя способами:

#### 1. Изменение кавычек на апострофы.

Если для пользователя не так важен вводимый символ, то возможно изменить символ ' на `. Таким образом вредоносный код будет принят как одна длинная строка и добавлен в базу данных.

Приведем функцию, изменяющую символ апострофа.

```

void CorrectApostrof(std::string &query)
{
    size_t pos;
    while ((pos = query.find('\')) != std::string::npos) {
        query.replace(pos, 1, "`");
    }
}

```

#### 2. Экранирование кавычек

Если для пользователя важен именно символ кавычки, то его для ввода в базу данных необходимо экранировать. Напомним, что кавычки в PostgreSQL экранируются повторением данного символа. Например, если мы хотим внести в таблицу строку имя John O'Brien, то запрос на добавление будет выглядеть следующим образом:

```

INSERT INTO ApostropheTest (Name, Surname)
VALUES ('John', 'O''Brien')

```

	name character varying (20) 🔒	surname character varying (20) 🔒
1	John	O'Brien

## 2. Практическая часть

### 2.1. Задание 1.

Запустите программу «ORIOKS Simulator». Обратите внимание на сообщение, которое было выведено в лог файл. Объясните полученную ошибку. Внесите необходимые изменения, чтобы подключиться к созданной базе данных

Войдите в учебную базу данных из-под разных пользователей – студент, преподаватель, методист. Сравните интерфейсы данных пользователей.

Реализуйте защиту от SQL-инъекции для строки поиска предмета. В случае попытки SQL - инъекции программа должна работать в штатном режиме с возможностью продолжать поиск предметов.

### 2.2.Задание 2.

Ознакомьтесь с функцией *searchBySubject()* класса *Diary*, реализующей поиск введенной дисциплины в списке предметов студента. Модифицируйте функцию поиска так, чтобы можно было искать сразу несколько предметов (предметы вводятся в строку поиска через запятую). В случае, если один или несколько из введенных предметов не существует в базе данных, вывести на экран только те предметы, которые есть в дневнике студента.

### 2.3. Задание 3.

Добавьте в программу автоматическую блокировку текущей учетной записи в случае попытки выполнить SQL –инъекцию, сопровождаемую сообщением о том, что учетная запись заблокирована (для того, чтобы определить, заблокирован пользователь или нет, добавляется еще одно поле в базе данных пользователя *is\_blocked*. Окно с информацией о блокировке должно отображаться поверх всех других окон и содержать кнопку выхода из учетной записи. При попытке зайти в данную учетную запись снова окно о блокировке вновь должно отображаться перед пользователем.

Зайдите под любым пользователем и выполните попытку SQL –инъекции. Продемонстрируйте результат.

## Список литературы

- [1] «Исходный код СУБД postgres,» [В Интернете]. Available: <https://github.com/postgres/postgres>. [Дата обращения: 30 01 2023].
- [2] Документация к PostgreSQL 15.1, 2022.
- [3] Е. Рогов, PostgreSQL изнутри, 1-е ред., Москва: ДМК Пресс, 2023, р. 662.
- [4] Б. А. Новиков, Е. А. Горшкова и Н. Г. Графеева, Основы технологии баз данных, 2-е ред., Москва: ДМК пресс, 2020, р. 582.
- [5] Е. П. Моргунов, PostgreSQL. Основы языка SQL, 1-е ред., Санкт-Петербург: БХВ-Петербург, 2018, р. 336.