

Лабораторная работа №1

Подключение разрабатываемого приложения к БД

1. Создание базы данных:

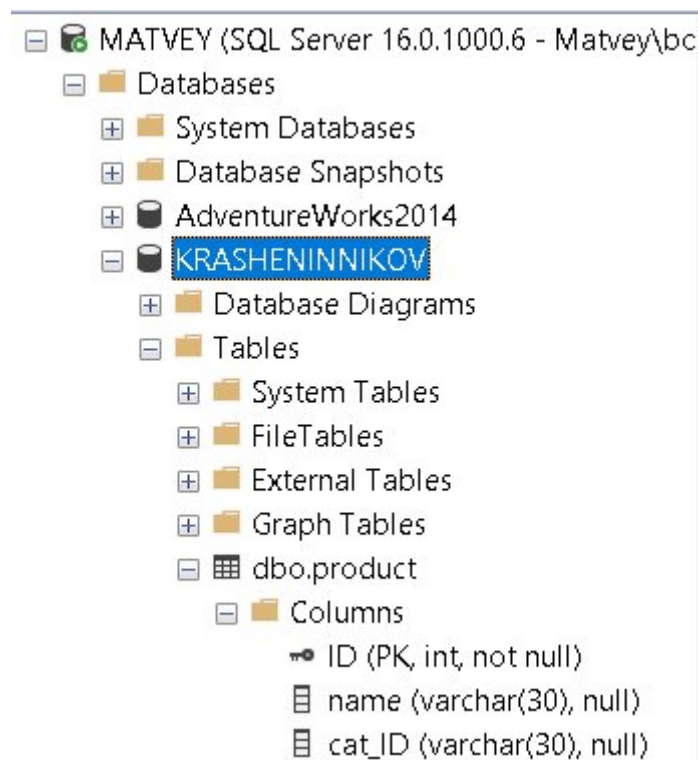


рис. 1

На рисунке 1 была создана база данных с моей фамилией, а в ней – таблица с 3 полями.

2. Добавление графических элементов:

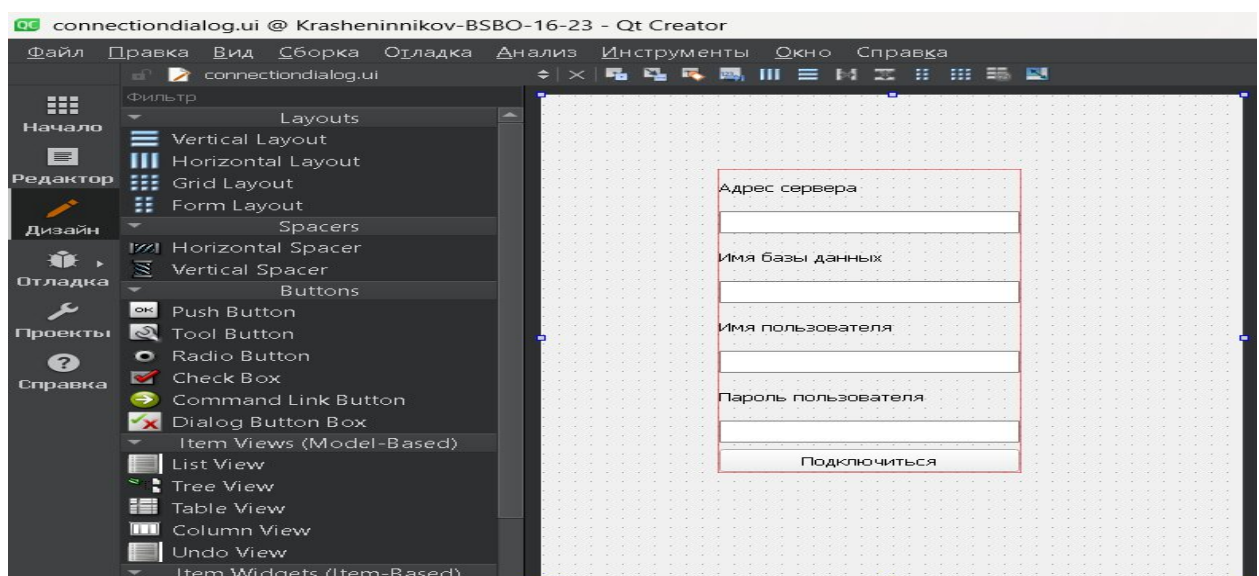


рис. 2

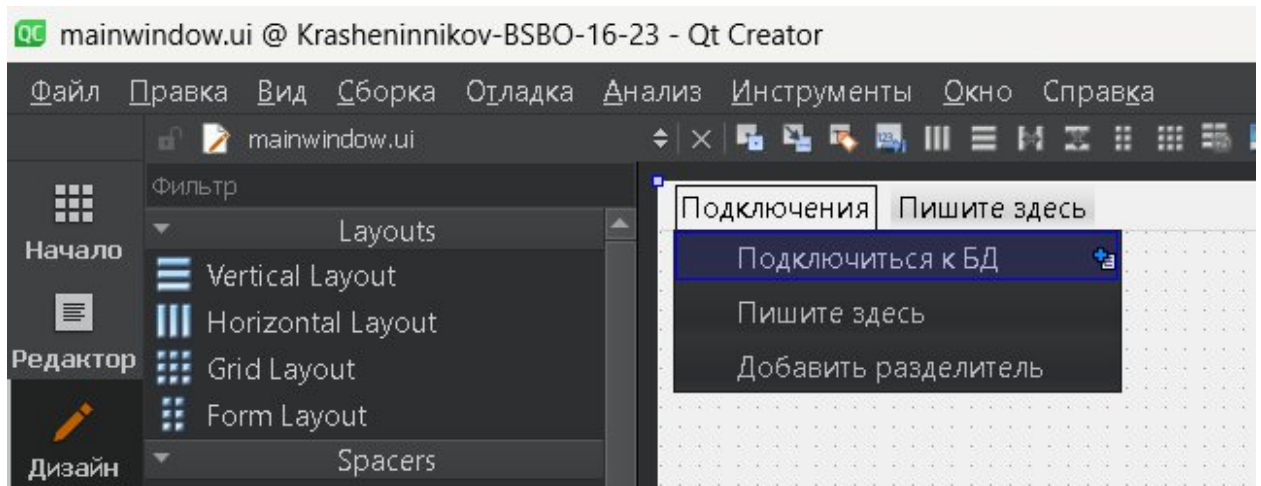


рис. 3

На рисунке 2 изображено несколько объектов типов label, lineEdit и один pushButton, которые были помещены на окно путём перетаскивания из списка существующих элементов.

На рисунке 3 была добавлена дополнительная кнопка «Подключиться к БД», которая будет открывать ConnectionDialog, а уже в нём будет происходить основная логика.

3. Создание нового класса ConnectionDialog, отсюда и соответствующие файлы - .cpp и .h, в котором описано определение обработки нажатия на кнопку «Подключиться», а в .cpp уже сама реализация. Также добавлены нужные классы для работы с БД. Новые приватные поля – указатель на ConnectionDialog и переменная типа QSqlDatabase, публичное – указатель на QMessageBox.

```
private slots:
```

```
void on_pushButton_clicked();
```

```
private:    Ui::ConnectionDialog *ui;    QSqlDatabase db;
```

```
public:    QMessageBox *msg;
```

Объяснение, почему db – это не указатель на QSqlDatabase: использование delete db нарушает логику работы с QSqlDatabase, поскольку этот объект управляется Qt и не предназначен для удаления с помощью delete.

4. Подключение соответствующего заголовка и реализация метода с предыдущего пункта:

```
void ConnectionDialog::on_pushButton_clicked() {
```

```

db = QSqlDatabase::addDatabase("QODBC"); // драйвер, который будем собирать
db.setDatabaseName("DRIVER={SQL Server}; SERVER="+ui->lineEdit->text()+"";
DATABASE="+ui->lineEdit_2->text()+"");
db.setUserName(ui->lineEdit_3->text());
db.setPassword(ui->lineEdit_4->text());

msg = new QMessageBox();

db.open() ? msg->setText("SUCCESS!") : msg->setText("FAIL");

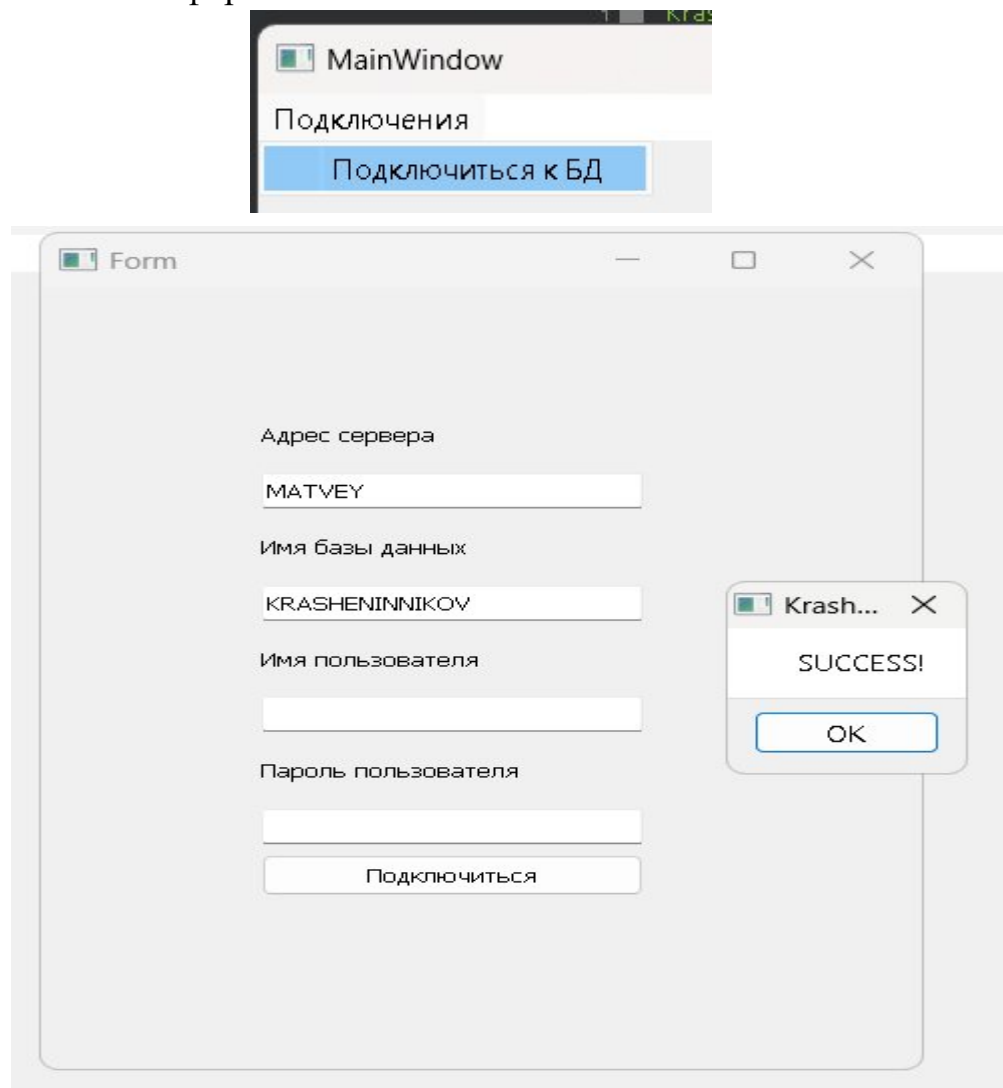
msg->show();

}

```

Изначально устанавливаем драйвер QODBC, предназначенный для подключения к базам данных через ODBC (Open Database Connectivity). В следующей строке устанавливаем DatabaseName на основе введенных пользователем данных (рис. 2). Далее выделяем память под сообщение, информирующее о подключении – успешно или нет, а затем отображаем его.

5. Имеем такие формы:



Контрольные вопросы

1. Какой класс необходим для установления соединения с базой данных?
2. На какие трёх слоях (уровнях) Qt осуществляет взаимодействие с базой данных?
3. Назовите назначение каждого слоя (уровня) взаимодействия.
4. Найдите в Интернете описание ODBC и кратко опишите его назначение и перечень поддерживаемых им источников данных.
5. Каким плагином следует воспользоваться, если нужно осуществить подключение к СУБД MySQL?

1. QSqlDatabase
2. Уровни: соединение, запросы, модели и представления.
3. Соединение: управление подключением (QSqlDatabase).

Запросы: выполнение SQL-команд (QSqlQuery).

Модели: работа с данными (QSqlTableModel, QSqlQueryModel).

4. ODBC — стандарт для доступа к базам через драйверы (MS SQL, MySQL, PostgreSQL, Oracle и др.).
5. QMYSQL

Лабораторная работа №2

Отображение содержимого таблицы на главной форме. Добавление новой строки.

Подключения Отчёт

Обновить Добавить

ID

Наименование

Категория

Изменить Удалить

Дата
01.01.2000

Печать в PDF

Рис 2.1

Добавляем кнопку «Обновить»:

```
void MainWindow::on_pushButton_clicked() {    flag = true;    model = new
 QSqlQueryModel();    model->setQuery("SELECT * FROM dbo.product;");
 model->setHeaderData(0, Qt::Horizontal, "Номер");    model->setHeaderData(1,
 Qt::Horizontal, "Название");    model->setHeaderData(2, Qt::Horizontal,
 "Категория");    model->setHeaderData(3, Qt::Horizontal, "Путь");
 model->setHeaderData(4, Qt::Horizontal, "Дата");
 ui->tableView->setModel(model);    ui->tableView->resizeColumnsToContents();
 // автоматическая установка нужных размеров    ui->tableView->show();}
```

Метод `on_pushButton_clicked()` выполняет SQL-запрос к базе данных и отображает данные в графическом интерфейсе пользователя, используя `QSqlTableModel` и `QTableView`.

Нажимаем:

	Номер	Название	Категория	Путь
1	1	Яблоко	Фрукты	
2	2	Груша	Фрукты	
3	3	Картофель	WHO	
4	4	Огурец	Овощи	
5	1007	КАРТИНКА	НЕ ЕДА	C:/Users/boltf/OneDrive/Рабо
6	1008	Колбаса	Вкусно!!!	
7	1009	Пельмени	Фрукты	

Контрольные вопросы

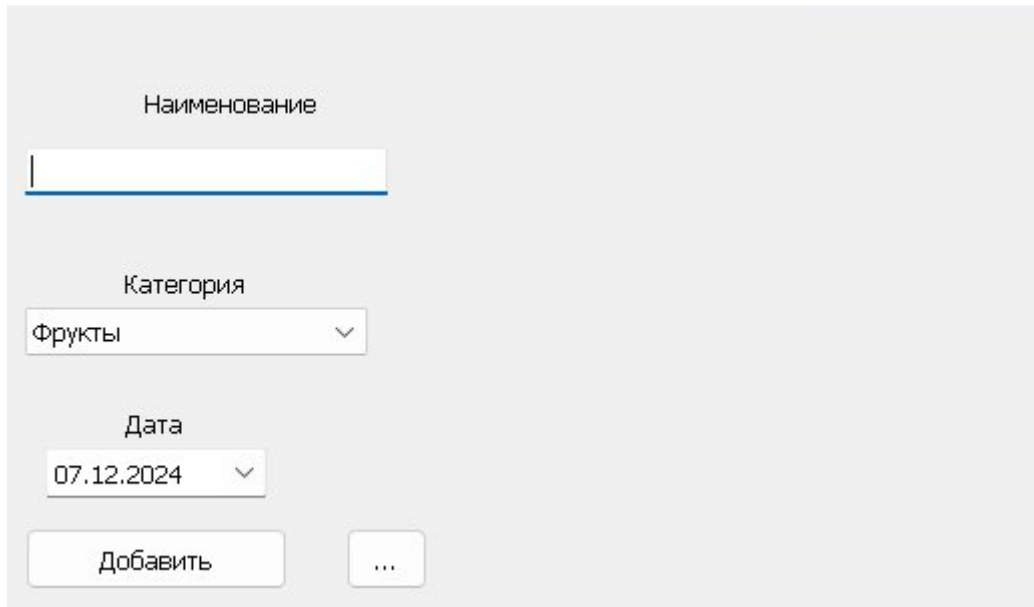
1. Какой шаблон проектирования лежит в основе архитектуры модель/представление?
2. Из каких объектов она состоит?
3. В чем отличие архитектуры модель/представление от шаблона MVC?
4. Найти в документации Qt способ создания модели для выборки данных из нескольких таблиц с возможностью редактирования?
5. Какие ещё классы моделей поддерживаются Qt?

1. MVC
2. Модель, представление, делегат
3. Контроллер заменён делегатом
4. `QSqlRelationalTableModel`
5. `QStandardItemModel`, `QSqlTableModel`, `QSqlQueryModel`

Лабораторная работа №3

Изменение и удаление строк в таблице БД

На рис. 2.1 установлена кнопка «Добавить», которая позволяет добавить новую строку в таблицу, а также отдельное диалоговое окно для удобства:



В конструкторе нового класса: `QSqlQuery* query = new QSqlQuery();`
`query->exec("SELECT cat FROM category;");`

Один из основных методов:

```
void AddDialog::on_pushButton_clicked() {    QSqlQuery* query = new
QSqlQuery();    query->prepare("INSERT INTO product VALUES
(:name, :cat_ID, :picadr, :dat)");    query->bindValue(":name",
ui->lineEdit->text());    query->bindValue(":cat_ID",
ui->comboBox->itemText(combo));    query->bindValue(":picadr", imageAdr);
query->bindValue(":dat", ui->dateEdit->text());    QMessageBox* mes = new
QMessageBox();    if (!query->exec()) {        mes->setText("Запрос составлен
неверно");        mes->show();    }    // либо добавить, что строка добавлена
успешно    close(); }
```

Выполняем запросик, подставляя в параметры значения из lineEdit'ов, а также проверяем, выполнен ли запрос

Удаление:

```
void MainWindow::on_pushButton_4_clicked() {    QSqlQuery* query = new
QSqlQuery();    query->prepare("DELETE FROM product WHERE ID=:ID"); //
сигнатура запроса    query->bindValue(":ID", ui->lineEdit->text());
query->exec(); // само выполнение запроса    ui->lineEdit->clear();
ui->lineEdit_2->clear();    ui->lineEdit_3->clear();
MainWindow::on_pushButton_clicked();}
```

Изменение:

```
void MainWindow::on_pushButton_3_clicked() {    QSqlQuery* query = new
QSqlQuery();    query->prepare("UPDATE product SET name=:name, cat_ID=:cat_ID
WHERE ID=:ID");    query->bindValue(":ID", ui->lineEdit->text());
query->bindValue(":name", ui->lineEdit_2->text());
query->bindValue(":cat_ID", ui->lineEdit_3->text());    query->exec();
ui->lineEdit->clear();    ui->lineEdit_2->clear();
ui->lineEdit_3->clear();    MainWindow::on_pushButton_clicked();}
```

Схожие методы, но разница именно в запросах – а так, взаимодействуем с введёнными значениями, биндим (подставляем) их в параметры запроса, выполняем, очищаем lineEdit'ы и вызываем обновление окошка, чтобы увидеть свежие данные.

Контрольные вопросы

1. В чём главное отличие объекта QSqlQuery от QSqlQueryModel? 2. Какие инструкции можно выполнить при помощи QSqlQuery? 3. Напишите метод, который бы считывал данные из результирующей выборки, количество записей в которой заранее не известно. 4. Какие сценарии работы с QSqlQuery вы знаете?

1. QSqlQuery: для выполнения запросов; QSqlQueryModel: для отображения данных.
2. Инструкции: SELECT, INSERT, UPDATE, DELETE, DDL.
3. Метод:

```
void method(QSqlQuery &query) {
```

```
    while (query.next()) {
```

```
        QString data = query.value(0).toString();
```

```

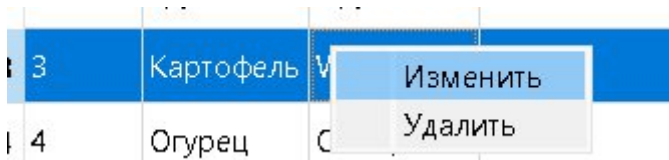
        // здесь уже обрабатываем данные
    }
}

```

4. Сценарии: Выполнение запросов с результатом (SELECT). Выполнение команд без результата (INSERT, UPDATE, DELETE). Использование параметризованных запросов.

Лабораторная работа №4

Использование контекстного меню для изменения и удаления строк таблицы



Добавили такое контекстное меню для реализации тех же функций, что и в предыдущей лабораторной работе. К слову, логика схожая, но используются немного другие методы.

```

void MainWindow::customMenuReq(QPoint position) {    if (flag)
{
    QModelIndex index = ui->tableView->indexAt(position);
    globalID = ui->tableView->model()->data
    (
        ui->tableView->model()->index(index.row(),
    0)
        )
        .toInt();
        QMenu* menu = new
    QMenu(this);
        QAction* change = new QAction("Изменить", this);
    connect(change, SIGNAL(triggered()), this, SLOT(changeRecord()));
    QAction* _delete = new QAction("Удалить", this);
    connect(_delete,
    SIGNAL(triggered()), this, SLOT(deleteRecord()));
    menu->addAction(change);
    menu->addAction(_delete);
    menu->popup(ui->tableView->viewport()->mapToGlobal(position));
}}void
MainWindow::deleteRecord() {    QSqlQuery* query = new QSqlQuery();
query->prepare("DELETE FROM product WHERE ID=:ID");
query->bindValue(":ID", globalID);    if (query->exec())
{
    MainWindow::on_pushButton_clicked();
}}void
MainWindow::changeRecord() {    ch = new changing();
connect(this,
    SIGNAL(sendID(int)), ch, SLOT(processSendID(int)));
    emit sendID(globalID);
ch->show();
disconnect(this, SIGNAL(sendID(int)), ch,
    SLOT(processSendID(int)));
}

```

customMenuReq: создаёт контекстное меню с действиями "Изменить" и "Удалить".

deleteRecord: удаляет запись из таблицы product по globalID и обновляет таблицу.

changeRecord: открывает окно changing, передавая в него globalID.

Контрольные вопросы

1. Что такое простой отчёт? Приведите пример.
2. Какие ещё виды отчётов существуют?
3. Какая технология использовалась при выполнении экспорта данных в MS Word?
4. Опишите фреймворк AxtiveX существующий в Qt.
 1. Простой отчёт: Сводка данных без сложных вычислений. Пример: список товаров с их ценами.
 2. Другие виды отчётов: Детализированные, сводные, графические, PDF-отчёты.
 3. Технология экспорта в MS Word: Использование библиотеки QAxObject для работы с COM-объектами Word.
 4. ActiveX в Qt: Поддержка через QAxWidget для встраивания ActiveX-компонентов в приложения.

Лабораторная работа №5

Формирование отчёта

На рис. 2.1 имеем кнопку «Печать в PDF». Реализация:

```
void MainWindow::on_pushButton_5_clicked() {    QString str;
str.append("<html><head></head><body><center>"+QString("Пример создания
отчета"));    str.append("<table border=1><tr>");
str.append("<td>"+QString("Номер")+"</td>");
str.append("<td>"+QString("Название")+"</td>");
str.append("<td>"+QString("Категория")+"</td></tr>");    QSqlQuery*
query=new QSqlQuery();    query->exec("SELECT id, name, cat_ID FROM
product");    while(query->next()) {        str.append("<tr><td>");
str.append(query->value(0).toString());        str.append("</td><td>");
str.append(query->value(1).toString());        str.append("</td><td>");
str.append(query->value(2).toString());
str.append("</td></tr>");    }
str.append("</table></center></body></html>");    QPrinter printer;
printer.setOrientation(QPrinter::Portrait);
printer.setOutputFormat(QPrinter::PdfFormat);
printer.setPaperSize(QPrinter::A4);    QString path =
QFileDialog::getSaveFileName(NULL, "Сохранить в
PDF", "Отчёт", "PDF (*.pdf)");    if (path.isEmpty()) {        return;    }
```

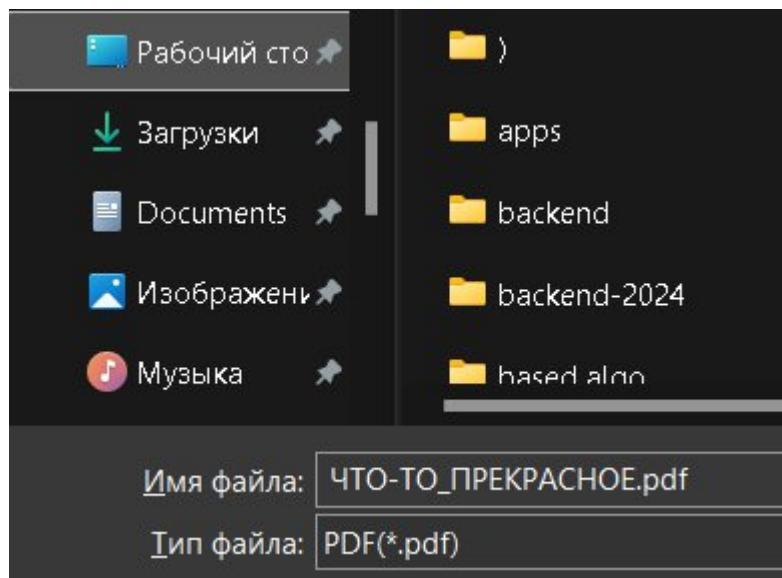
```
printer.setOutputFileName(path);    QTextDocument doc;
doc.setHtml(str);    doc.print(&printer);}
```

Формируется строка в формате HTML с заголовками таблицы и данными из базы (выборка из таблицы product).

QSqlQuery выполняет SQL-запрос для получения данных (ID, имя и категория товара) из базы данных.

Для каждого результата запроса добавляются строки в таблицу HTML. Используется QPrinter для создания PDF-документа, настроенного на формат A4 и ориентацию Portrait.

Через QFileDialog пользователь выбирает место для сохранения PDF-файла. Если путь выбран, создаётся и сохраняется PDF-документ с помощью QTextDocument, который выводит данные в формате HTML.



Сохраним и откроем результат:



Пример создания отчета

Номер	Название	Категория
1	Яблоко	Фрукты
2	Груша	Фрукты
3	Картофель	ВНО
4	Огурец	Овощи
1007	КАРТИНКА	НЕ ЕДА
1008	Колбаса	Вкусно!!!
1009	Пельмени	Фрукты

Контрольные вопросы

1. При помощи какой группы операторов таблица product была разделена на две таблицы? 2. Какой тип данных используется для хранения информации об изображении и почему? 3. Можно ли хранить изображение непосредственно в базе данных? 4. Где должно храниться изображение, чтобы оно было доступно всем пользователям информационной системы?

1. Join или Union
2. BLOB (Binary Large Object), но в нашем случае мы хранили локальный путь до изображения
3. Да, изображения можно хранить в базе данных в виде BLOB
4. В файловой системе на сервере

Лабораторная работа №6 и 7

Добавление изображений и даты к записи

Использование combobox для выбора значения ячейки

Усовершенствуем таблицу product:

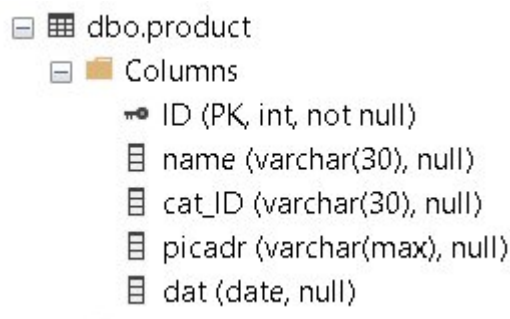


Рис. 2.1 – нажмём добавить:

Наименование

Вареники

Категория

Фрукты

Дата

07.12.2024

Добавить

...

Путь	Дата
OneDrive/Рабочий стол/ava.jpg	
	2024-12-06
	2024-12-07
OneDrive/Рабочий стол/погодите это реально.jpg	2024-12-07

Обновить

Добавить

ID

1010

Наименование

Вареники

Категория

Фрукты

Изменить

Удалить

Дата

01.01.2000

Печать в PDF

	Номер	Название	Категория	
2	2	Груша	Фрукты	
3	3	Картофель	WHO	
4	4	Огурец	Овощи	
5	1007	КАРТИНКА	НЕ ЕДА	C:/Users/boltf/OneDrive/Рабо
6	1008	Колбаса	Вкусно!!!	
7	1009	Пельмени	Фрукты	
8	1010	Вареники	Фрукты	C:/Users/boltf/OneDrive/Рабо

Реализация:

```

В конструкторе: ui->dateEdit->setDate(QDate::currentDate());    QSqlQuery*
query = new QSqlQuery();    query->exec("SELECT cat FROM category;");
while (query->next())
{
    ui->comboBox->addItem(query->value(0).toString());    }    combo =
0;
void AddDialog::on_toolButton_clicked() {    imageAdr =
QFileDialog::getOpenFileName(0, "Открыть изображение", "/", "*.jpg");

```

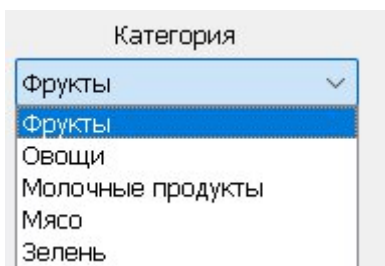
```

ui->label_3->setScaledContents(true);
ui->label_3->setPixmap(QPixmap(imageAdr));}void
AddDialog::on_comboBox_currentIndexChanged(int index) {    combo = index;}

```

В цикле while (query->next()) мы проходим по всем строкам результата запроса. Для каждой строки извлекается значение из первого столбца (в данном случае cat), и оно добавляется в комбинированный список (QComboBox) с помощью метода addItem. Это заполняет выпадающий список значениями категорий, полученными из базы данных. После завершения цикла, переменной combo присваивается значение 0.

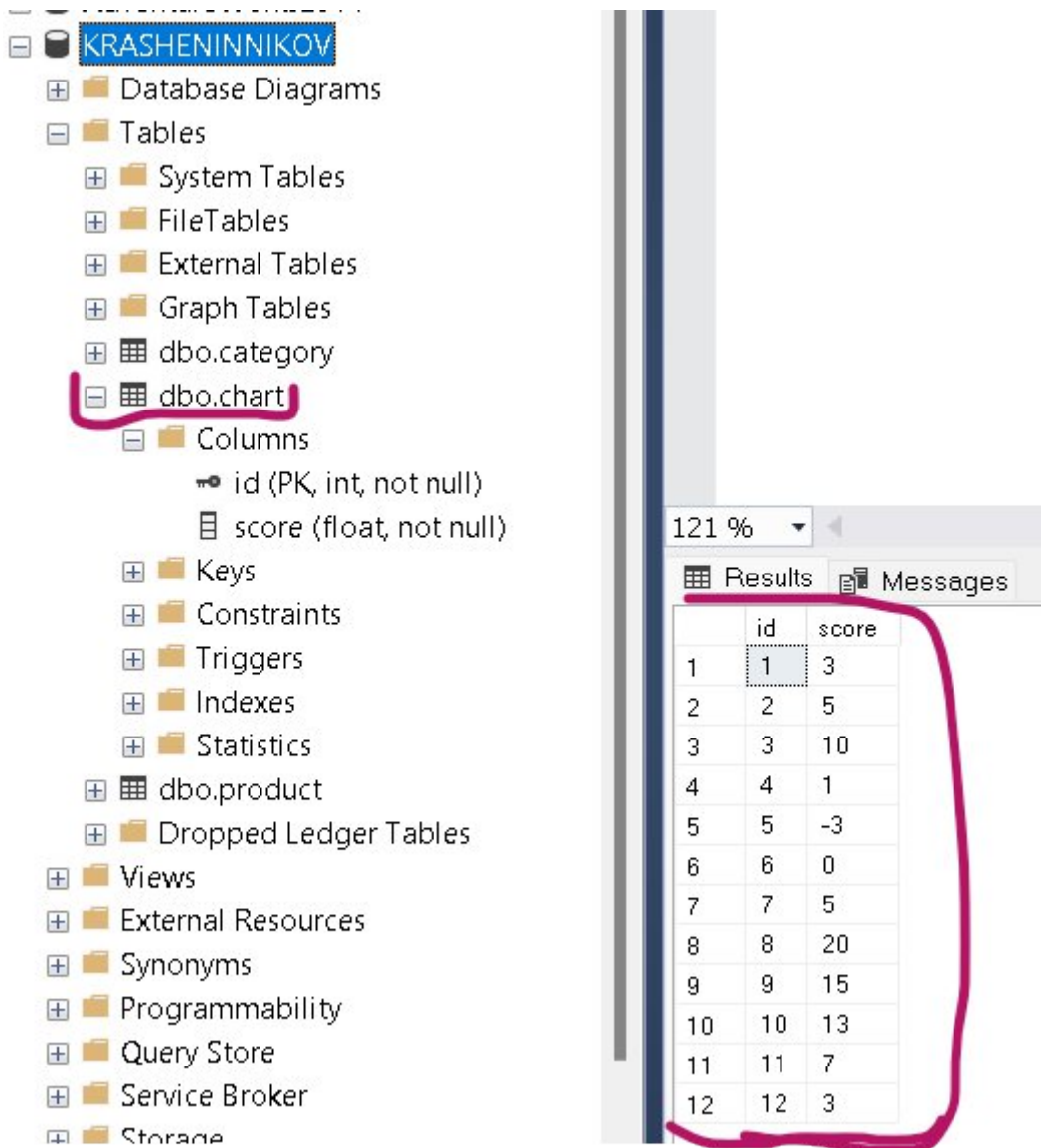
query->bindValue(":cat_ID", ui->comboBox->itemText(combo)): Привязывает выбранное значение из QComboBox (используется индекс postcombo, который, как предполагается, указывает на выбранный элемент) к параметру :cat_ID. Значение преобразуется в строку с помощью itemText().
void AddDialog::on_comboBox_currentIndexChanged(int index) { postcombo = index; } Присваиваем в combo index, который был передан



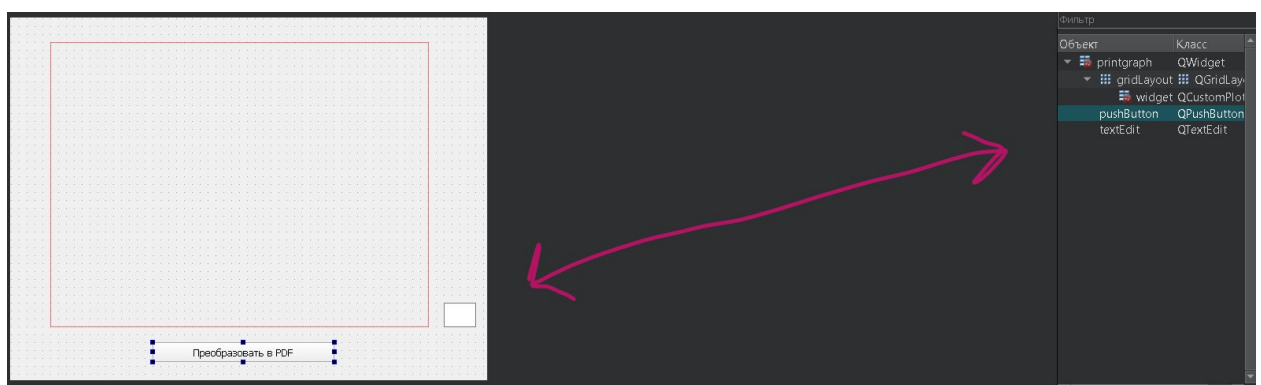
Лабораторная работа №8

Создание графика и сохранение его в PDF

Создали простеньку табличку с выручкой:



Добавили новую форму и нужные элементы:



Основная реализация:

```
void printgraph::on_pushButton_clicked() {    QString filename =
QFileDialog::getSaveFileName(0, "Сохранить PDF", "/", "*.pdf");    if
(!filename.isEmpty()) {        QPrinter print;
print.setFullPage(true);        print.setPaperSize(QPrinter::A4);
print.setOrientation(QPrinter::Portrait);
```

```
print.setOutputFormat(QPrinter::PdfFormat);
print.setOutputFileName(filename);
ui->textEdit->document()->print(&print);    }}
```

При клике на кнопку открывается диалог для сохранения файла в формате PDF. Если пользователь выбрал путь, создаётся объект QPrinter, настраиваются параметры печати (формат A4, ориентация Portrait, вывод в PDF), и содержимое textEdit печатается в PDF.

```
QCPDocumentObject* poHandler = new QCPDocumentObject(this);
ui->textEdit->document()->document
Layout()->registerHandler(QCPDocumentObject::PlotTextFormat,
poHandler);    ui->textEdit->setVisible(false);
ui->widget->plotLayout()->insertRow(0);
ui->widget->plotLayout()->addElement(0, 0, new QCPPlotTitle(ui->widget,
"График выручки"));    QVector<double> dx, dy;    double minX, minY, maxX,
maxY;    minX = minY = maxX = maxY = 0;    QSqlQuery* query = new
QSqlQuery();    if (query->exec("SELECT * FROM chart")) {        while
(query->next()) {            if (minX >= query->value(0).toDouble()) minX =
query->value(0).toDouble();            if (minY >=
query->value(1).toDouble()) minY = query->value(1).toDouble();            if
(maxX <= query->value(0).toDouble()) maxX = query->value(0).toDouble();
if (maxY <= query->value(1).toDouble()) maxY = query->value(1).toDouble();
dx << query->value(0).toDouble();            dy <<
query->value(1).toDouble();            QCPBars* bar = new
QCPBars(ui->widget->xAxis, ui->widget->yAxis);
bar->setName("Значение");            bar->setBrush(QColor(255, 0, 0, 255));
bar->setData(dx, dy);            bar->setWidth(0.20);
ui->widget->xAxis->setLabel("Месяц");
ui->widget->yAxis->setLabel("Выручка (млн)");
ui->widget->xAxis->setRange(minX, maxX + 0.20);
ui->widget->yAxis->setRange(minY, maxY + 1.0);
ui->widget->xAxis->setAutoTickStep(false);
ui->widget->yAxis->setAutoTickStep(false);
ui->widget->xAxis->setTickStep(1.0);
ui->widget->yAxis->setTickStep(1.0);
ui->widget->replot();        }    }    QTextCursor cur =
ui->textEdit->textCursor();
cur.insertText(QString(QChar::ObjectReplacementCharacter),
QCPDocumentObject::generatePlotFormat(ui->widget, 480, 340));}
```

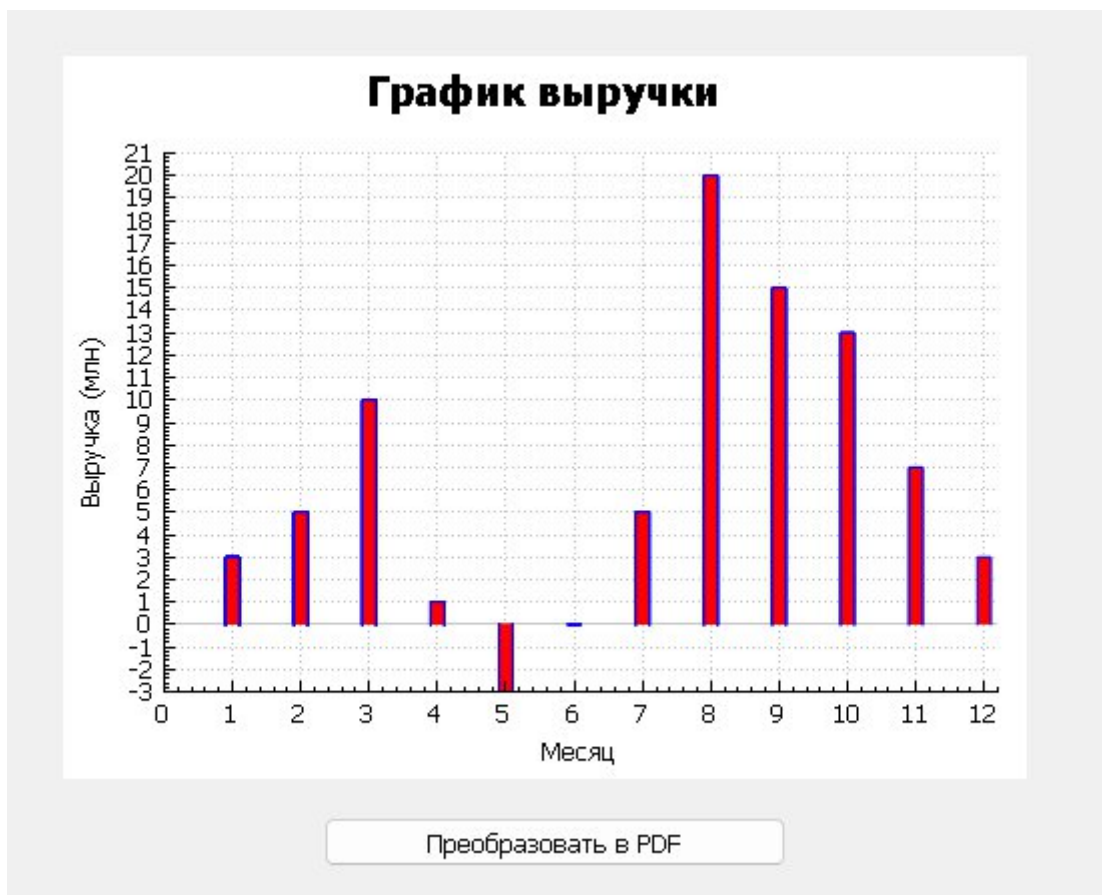
Создаётся объект QCPDocumentObject, который будет использоваться для вставки графика в текстовый документ.

Регистрируется обработчик для вставки графика в текст с использованием формата QCPDocumentObject::PlotTextFormat.

В widget добавляется строка, затем устанавливается заголовок графика "График выручки".

Выполняется SQL-запрос для получения данных из таблицы chart. Для каждого результата определяются минимальные и максимальные значения для осей X и Y.

Используется QSPBars для построения графика с данными о выручке. Устанавливаются параметры для осей (метки, диапазоны, шаги). В textEdit вставляется объект-заполнитель для графика (заменитель объекта) с использованием функции generatePlotFormat, которая генерирует изображение графика. После настройки всех параметров вызывается replot() для обновления отображения графика.



Преобразуем в PDF старым способом:

