Geekbrains

Дипломная работа

Система управления записями малого бизнеса с использованием баз данных и фреймворка Qt

Программа: Разработчик C++. Специалист

ФИО Байлук А.А.

Санкт-Петербург

2025

СОДЕРЖАНИЕ

[1 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ 1](#_Toc205889446)

[1.1 Архитектура клиент-серверных приложений 1](#_Toc205889447)

[1.2 Различие TCP и UDP 3](#_Toc205889448)

[1.3 Возможности фреймворка Qt для разработки GUI 5](#_Toc205889449)

[2 Практическая часть 8](#_Toc205889450)

[2.1 Архитектура базы данных 8](#_Toc205889451)

[2.2 Разработка библиотеки DbManager 10](#_Toc205889452)

[2.3. Разработка серверной части проекта 14](#_Toc205889453)

[2.4 Разработка клиентской части проекта: 19](#_Toc205889454)

[3. Проведение тестирования 34](#_Toc205889455)

[ЗАКЛЮЧЕНИЕ 39](#_Toc205889456)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 42](#_Toc205889457)

ВВЕДЕНИЕ

Современные малые предприятия в сфере услуг всё чаще сталкиваются с необходимостью автоматизации бизнес-процессов. В салонах красоты, парикмахерских и косметологических кабинетах ручное ведение записей в блокнотах или таблицах Excel приводит к ошибкам, дублированию, утечкам информации и снижению качества обслуживания. При этом коммерческие решения зачастую дороги, требуют подписки и выхода в интернет, что делает их недоступными для небольших компаний.

В связи с этим актуальной задачей становится создание бесплатного и простого в использовании приложения, способного эффективно управлять записями клиентов без зависимости от внешних сервисов. Разработка подобного ПО позволяет повысить уровень сервиса, минимизировать человеческий фактор и обеспечить централизованное хранение данных.

Целью дипломного проекта являетсяразработка и реализация клиентского приложения для автоматизации процесса записи клиентов в салон красоты, включающего:

* Удобный графический интерфейс с поддержкой вкладок;
* Формирование и редактирование записей;
* Ведение анкеты клиента;
* Административную панель для отслеживания посещений;
* Отправку данных на сервер по протоколу TCP для централизованного хранения.

Выполнение дипломного проекта осуществляется в несколько этапов:

1. Анализ предметной области — изучение существующих решений, определение требований к системе.
2. Проектирование архитектуры приложения — разработка структуры интерфейса, определение классов и взаимодействий.
3. Реализация графического интерфейса — создание вкладок, таблиц, диалоговых окон без использования Qt Designer.
4. Разработка функционала — добавление, редактирование и удаление записей.
5. Интеграция сетевого взаимодействия — реализация TCP-клиента для отправки данных на сервер.
6. Тестирование и отладка — проверка корректности работы всех компонентов.
7. Оформление результатов — подготовка документации, приложений и выводов.

Проект решает проблемуотсутствия простого, автономного и недорогого решения для автоматизации записи клиентов в малом бизнесе. Предлагаемое приложение позволяет:

* Заменить бумажные журналы и Excel-таблицы;
* Предотвратить конфликты по времени;
* Централизованно хранить данные;
* Обеспечить наглядность и удобство работы для администратора и мастера.

Для выполнения проекта были использованы знания и навыки, полученные в ходе изучения следующих дисциплин:

* Программирование на C++
* Объектно-ориентированное проектирование
* Сетевые технологии
* Графические интерфейсы пользователя

В ходе реализации проекта используются следующие инструменты и технологии:

* Язык программирования: C++17
* Фреймворк: Qt 6
* Среда разработки: Qt Creator / CMake
* Сетевое взаимодействие: QTcpSocket, протокол TCP/IP
* Управление проектом: Git

Проект выполнен в индивидуальном порядке. Автор выполнил все этапы разработки самостоятельно, выступая в следующих ролях:

* Аналитик — анализ требований и проектирование функционала;
* Разработчик (C++/Qt) — реализация интерфейса и логики приложения;
* Сетевой инженер — настройка TCP-взаимодействия;
* Тестировщик — проверка корректности работы всех модулей;
* Документатор — оформление технической документации и приложений.

Хороший пример командной работы: ранее участвовал в учебном проекте по разработке CRM-системы в роли C++-разработчика. В команде также были: руководитель проекта, тестировщик и технический писатель. Полученный опыт позволил эффективно организовать самостоятельную работу над данным дипломным проектом.

# **1 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ**

## **1.1 Архитектура клиент-серверных приложений**

Современные информационные системы всё чаще строятся на основе клиент-серверной архитектуры — модели взаимодействия программных компонентов, при которой одна часть системы (сервер) предоставляет ресурсы или услуги, а другая (клиент) запрашивает и использует их. Эта архитектура является фундаментальной для построения распределённых приложений и широко применяется в веб-сервисах, базах данных, системах автоматизации и других областях информационных технологий.

Клиент-серверная архитектура (англ. Client-Server Architecture) — это модель вычислительной сети, в которой клиент инициирует запрос на выполнение определённого действия, а сервер принимает этот запрос, обрабатывает его и возвращает результат. Клиент и сервер могут находиться на одном компьютере (локально) или в разных узлах сети.

* Клиент — приложение, с которым взаимодействует пользователь. В рамках данного проекта клиентом выступает разрабатываемое настольное приложение на C++ с графическим интерфейсом.
* Сервер — программа или устройство, отвечающее за хранение данных, обработку запросов и обеспечение доступа к ресурсам. В проекте используется простой TCP-сервер, принимающий данные о записях и сохраняющий их для дальнейшего использования.

Взаимодействие между клиентом и сервером осуществляется по заранее определённому протоколу, который задаёт формат сообщений, порядок обмена данными и правила обработки ошибок.

Использование клиент-серверной модели предоставляет ряд существенных преимуществ по сравнению с автономными или одноранговыми (P2P) решениями:

1. Централизованное хранение данных  
   Все данные сосредоточены на сервере, что исключает их дублирование и несогласованность. Это особенно важно в условиях, когда несколько сотрудников работают с одной базой записей. Например, администратор и мастер могут использовать разные клиентские приложения, но получать актуальную информацию с одного сервера.
2. Масштабируемость  
   Систему можно легко масштабировать: добавлять новых клиентов (рабочие места), увеличивать производительность сервера, внедрять резервное копирование. Архитектура позволяет отделить логику от данных, что упрощает модернизацию.
3. Безопасность  
   Доступ к данным контролируется на уровне сервера. Можно реализовать аутентификацию, шифрование, ограничение прав доступа. Даже если клиентское устройство будет утеряно, данные на сервере останутся защищёнными.
4. Упрощённое обновление и сопровождение  
   Обновление логики или структуры данных производится на сервере, и все клиенты автоматически получают изменения при следующем подключении. Это исключает необходимость обновления программного обеспечения на каждом рабочем месте.
5. Надёжность и отказоустойчивость  
   Сервер может быть настроен на резервирование, логирование и автоматическое восстановление, что повышает отказоустойчивость системы в целом.

В рамках данного проекта используется TCP/IP как наиболее подходящий протокол для локальной автоматизированной системы. Он позволяет организовать надёжное соединение между клиентским приложением и сервером без необходимости в интернете, что идеально подходит для салонов красоты, где важна автономность и стабильность работы.

В дипломном проекте реализована клиент-серверная архитектура:

* Клиент — настольное приложение на C++ с использованием фреймворка Qt;
* Сервер — простой TCP-сервер;

Каждое действие пользователя (создание, редактирование, удаление записи) приводит к отправке структурированного JSON-объекта на сервер. Это позволяет:

* Централизованно хранить все записи;
* Обеспечить резервное копирование данных;
* В перспективе — подключить несколько клиентов к одному серверу;
* Организовать логирование событий.

## **1.2 Различие TCP и UDP**

При разработке клиент-серверных приложений, особенно в условиях локальной сети (LAN), выбор сетевого протокола передачи данных играет ключевую роль в обеспечении надёжности, производительности и целостности информации. Наиболее широко используемыми протоколами транспортного уровня модели OSI являются TCP (Transmission Control Protocol) и UDP (User Datagram Protocol). Оба протокола работают поверх IP (Internet Protocol), но существенно различаются по принципам функционирования, что определяет их применимость в различных типах приложений.

TCP (Transmission Control Protocol) — это надёжный, ориентированный на соединение протокол, обеспечивающий упорядоченную и достоверную передачу данных между двумя узлами сети.

Основные особенности TCP:

* Установление соединения (three-way handshake): перед передачей данных клиент и сервер устанавливают логическое соединение.
* Контроль доставки пакетов: каждый пакет подтверждается (acknowledgment), и при потере — автоматически повторяется.
* Упорядоченная доставка: пакеты нумеруются, и на приёмной стороне восстанавливаются в правильной последовательности.
* Контроль потока (flow control): предотвращает переполнение буфера приёмника.
* Контроль перегрузки сети (congestion control): адаптирует скорость передачи в зависимости от состояния сети.

UDP (User Datagram Protocol) — это ненадёжный, без установления соединения протокол, ориентированный на скорость и минимальные накладные расходы.

Основные особенности UDP:

* Отсутствие соединения: данные отправляются без предварительного handshake.
* Нет подтверждения доставки: отправитель не узнаёт, получил ли получатель пакет.
* Нет гарантии порядка: пакеты могут приходить в произвольной последовательности или теряться.
* Минимальные накладные расходы: заголовок UDP — 8 байт против 20+ байт у TCP.
* Высокая скорость передачи: за счёт отсутствия контроля и повторных отправок.

## **1.3 Возможности фреймворка Qt для разработки GUI**

Одним из ключевых факторов успешной реализации настольного приложения является выбор подходящего инструмента для разработки графического пользовательского интерфейса (GUI). В рамках данного проекта в качестве основного фреймворка выбран Qt — мощная, кроссплатформенная библиотека для разработки приложений на языке C++. Qt зарекомендовал себя как надёжное и гибкое решение, широко используемое как в промышленной, так и в учебной разработке. Ниже рассматриваются ключевые возможности Qt, которые были использованы при создании системы управления записями в салоне красоты.

Одним из главных преимуществ Qt является его высокая кроссплатформенность. Приложение, написанное с использованием Qt, может быть скомпилировано и запущено на различных операционных системах без необходимости переписывания кода:

* Windows (7, 10, 11)
* Linux (Ubuntu, Fedora, Debian и др.)
* macOS

Это достигается за счёт абстрагирования от низкоуровневых API операционных систем: Qt предоставляет единый интерфейс для работы с окнами, графикой, шрифтами, событиями ввода и другими компонентами. В условиях проекта это означает, что разработанное приложение может быть использовано в салоне красоты независимо от используемой ОС, что значительно расширяет его практическую применимость.

Кроме того, Qt поддерживает различные системы сборки — как qmake, так и CMake, что позволяет интегрировать проект в современные среды разработки и CI/CD-процессы.

Qt предоставляет обширную библиотеку графических компонентов (виджетов), которые позволяют быстро и эффективно создавать сложные интерфейсы. В рамках проекта были использованы следующие ключевые виджеты:

* QTabWidget — упрощает восприятие интерфейса и позволяет логически разделить функционал.
* QTableWidget — используется для отображения табличных данных. Поддерживает редактирование, выделение, сортировку и настройку размеров столбцов.
* QDateEdit и QTimeEdit — удобные элементы для выбора даты и времени.
* QLineEdit — поля ввода.
* QComboBox — выпадающий список.
* QPushButton — кнопки действий.

Все эти компоненты легко настраиваются, поддерживают стилизацию через стилевые таблицы (Qt Style Sheets) и могут быть вложены в различные компоновки (QVBoxLayout, QHBoxLayout, QFormLayout), что обеспечивает гибкость при проектировании макета.

Помимо GUI, Qt предоставляет мощные инструменты для сетевого программирования. В частности, класс QTcpSocket позволяет реализовать TCP-клиент без использования сторонних библиотек. Он интегрирован в цикл событий Qt и работает асинхронно, что исключает блокировку интерфейса при отправке или приёме данных.

В проекте QTcpSocket используется для:

* Подключения к локальному серверу (127.0.0.1:1122);
* Отправки JSON-сообщений о создании, редактировании и удалении записей;
* Обработки ответов от сервера (например, подтверждения приёма данных).

Сигналы connected, readyRead, errorOccurred позволяют гибко управлять состоянием соединения и обрабатывать ошибки в реальном времени.

Хотя Qt предлагает визуальный редактор Qt Designer для создания .ui-файлов, в данном проекте был выбран **подход** ручного программирования интерфейса на C++. Это позволило:

* Полностью контролировать структуру и иерархию виджетов;
* Избежать привязки к внешним файлам;
* Упростить отладку и сопровождение кода;
* Лучше понять архитектуру Qt (компоновки, сигналы, владение объектами).

Одной из уникальных особенностей Qt является система сигналов и слотов — механизм, позволяющий объектам взаимодействовать без жёсткой привязки. Сигнал генерируется при наступлении события (например, нажатие кнопки), а слот — это функция, которая его обрабатывает.

Сочетание объектно-ориентированного C++ **и** гибкого механизма взаимодействия делает Qt особенно удобным для создания сложных, но хорошо структурированных приложений.

Фреймворк Qt оказался оптимальным выбором для реализации клиентской части системы управления записями. Его кроссплатформенность**,** богатая библиотека виджетов**,** встроенная поддержка сетевого взаимодействия **и** гибкий механизм сигналов/слотов позволили создать функциональное, надёжное и удобное приложение с минимальными временными и ресурсными затратами.

Использование чистого C++ без Qt Designer способствовало более глубокому пониманию архитектуры GUI-приложений и обеспечило полный контроль над поведением интерфейса. Таким образом, Qt не только ускорил разработку, но и стал эффективным инструментом для демонстрации практических навыков программирования на C++ в рамках дипломного проекта.

# 2 Практическая часть

## **2.1 Архитектура базы данных**

В базе данных реализованы пять таблиц: Archive, Journal, People, Service, Watcher.

Таблица Archive необходима для хранения архивной информации о изменении данных в таблицах Journal, People, Service. В таблице хранится информация о услуге, мастере, клиенте и времени события.

SQL запрос на создание таблицы:

**CREATE** **TABLE** `Archive` (

`ArchiveId` **int** **NOT** **NULL** **AUTO\_INCREMENT**,

`DescribeArch` **text** **CHARACTER** **SET** utf8mb4 **COLLATE** utf8mb4\_0900\_ai\_ci COMMENT 'Описание события или данных',

`timeArch` **datetime** **NOT** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP**,

`ServiceId` **int** **NOT** **NULL**,

`MasterId` **int** **NOT** **NULL**,

`ClientId` **int** **NOT** **NULL**,

**PRIMARY** **KEY** (`ArchiveId`)

) **ENGINE**=InnoDB **DEFAULT** CHARSET=utf8mb4 **COLLATE**=utf8mb4\_0900\_ai\_ci;

Таблица Journal необходима для хранения информации о записях на услуги. В таблице хранится информация о услуге, мастере, клиенте и времени события.

**CREATE** **TABLE** `Journal` (

`JournalId` **int** **NOT** **NULL** **AUTO\_INCREMENT**,

`Service\_Id` **int** **NOT** **NULL**,

`People\_Id` **int** **NOT** **NULL** COMMENT 'Клиент',

`Master\_Id` **int** **DEFAULT** **NULL** COMMENT 'Мастер (может быть NULL)',

`Date` **datetime** **NOT** **NULL**,

`CreatedAt` **timestamp** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP**,

**PRIMARY** **KEY** (`JournalId`)

) **ENGINE**=InnoDB **AUTO\_INCREMENT**=14 **DEFAULT** CHARSET=utf8mb4 **COLLATE**=utf8mb4\_0900\_ai\_ci;

Таблица People необходима для хранения информации о людях находящихся в системе: клиенты, мастера, администраторы. В таблице хранится информация о имени, телефоне, состоянии в системе и роли человека.

**CREATE** **TABLE** `People` (

`PeopleId` **int** **NOT** **NULL** **AUTO\_INCREMENT**,

`Name` **varchar**(255) **NOT** **NULL**,

`State` **enum**('active','inactive','blocked') **NOT** **NULL** **DEFAULT** 'active',

`Photo` **longblob** COMMENT 'Фотография в бинарном виде (BLOB)',

`Role` **enum**('client','master','admin') **NOT** **NULL** **DEFAULT** 'client',

`CreatedAt` **timestamp** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP**,

`UpdatedAt` **timestamp** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP** **ON** **UPDATE** **CURRENT\_TIMESTAMP**,

`Phone` **varchar**(100) **DEFAULT** **NULL**,

**PRIMARY** **KEY** (`PeopleId`)

) **ENGINE**=InnoDB **AUTO\_INCREMENT**=9 **DEFAULT** CHARSET=utf8mb4 **COLLATE**=utf8mb4\_0900\_ai\_ci;

Таблица Service необходима для хранения информации о услугах. В таблице хранится информация о названии, цене, длительности и статусе услуги.

**CREATE** **TABLE** `Services` (

`ServicesId` **int** **NOT** **NULL** **AUTO\_INCREMENT**,

`Name` **varchar**(255) **NOT** **NULL**,

`Price` **decimal**(10,2) **NOT** **NULL**,

`Duration` **time** **NOT** **NULL** COMMENT 'Длительность услуги, например, 01:30:00',

`CreatedAt` **timestamp** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP**,

`UpdatedAt` **timestamp** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP** **ON** **UPDATE** **CURRENT\_TIMESTAMP**,

`State` **enum**('active','inactive') **NOT** **NULL** **DEFAULT** 'active',

**PRIMARY** **KEY** (`ServicesId`),

**CONSTRAINT** `Services\_chk\_1` **CHECK** ((`Price` >= 0))

) **ENGINE**=InnoDB **AUTO\_INCREMENT**=8 **DEFAULT** CHARSET=utf8mb4 **COLLATE**=utf8mb4\_0900\_ai\_ci;

Таблица Watcher необходима для реагирования системы на изменения. В таблице хранится информация о информация о услуге, мастере, клиенте и времени события. Как только в таблицах Journal, People, Service что-то меняется информация об этом попадает в Watcher.

**CREATE** **TABLE** `Watcher` (

`Id` **int** **NOT** **NULL** **AUTO\_INCREMENT**,

`Action` **enum**('I','U','D') **CHARACTER** **SET** utf8mb4 **COLLATE** utf8mb4\_0900\_ai\_ci **NOT** **NULL**,

`TableName` **varchar**(64) **NOT** **NULL** COMMENT 'Имя таблицы: People, Services, Journal и т.д.',

`Item\_Id` **int** **NOT** **NULL** COMMENT 'ID записи в указанной таблице',

`CreatedAt` **timestamp** **NULL** **DEFAULT** **CURRENT\_TIMESTAMP**,

**PRIMARY** **KEY** (`Id`)

) **ENGINE**=InnoDB **AUTO\_INCREMENT**=20 **DEFAULT** CHARSET=utf8mb4 **COLLATE**=utf8mb4\_0900\_ai\_ci;

Опишем триггеры для всех таблиц.

Триггеры необходимы для реагирования системы на изменения данных. После изменения информация о изменении должна попасть Watcher. Триггеры для таблицы Journal:

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `JOURNAL\_ON\_INSERT` **AFTER** **INSERT** **ON** `Journal` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("I", "Jouranl", **new**.JournalId);

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `JOURNAL\_ON\_DELETE` **AFTER** **DELETE** **ON** `Journal` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("D", "Jouranl", **old**.JournalId);

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `JOURNAL\_ON\_UPDATE` **AFTER** **UPDATE** **ON** `Journal` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("U", "Jouranl", **new**.JournalId);

Триггеры для таблицы People:

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `People\_ON\_INSERT` **AFTER** **INSERT** **ON** `People` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("I", "People", **new**.PeopleId);

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `People\_ON\_DELETE` **AFTER** **DELETE** **ON** `People` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("D", "People", **old**.PeopleId);

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `People\_ON\_UPDATE` **AFTER** **UPDATE** **ON** `People` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("U", "People", **new**.PeopleId);

Триггеры для таблицы Service:

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `Service\_ON\_INSERT` **AFTER** **INSERT** **ON** `Service` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("I", "Service", **new**.Service Id);

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `Service\_ON\_DELETE` **AFTER** **DELETE** **ON** `Service` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("D", "Service", **old**.Service Id);

**CREATE** **DEFINER**=`bylook`@`%` **TRIGGER** `Service\_ON\_UPDATE` **AFTER** **UPDATE** **ON** `Service` **FOR** **EACH** **ROW** **INSERT** **INTO** Watcher (`Action`,`TableName`, `Item\_Id` ) **VALUES** ("U", "Service", **new**.ServiceId);

## **2.2 Разработка библиотеки DbManager**

Библиотека DbManager предназначена для управления подключением к базе данных и выполнения стандартных операций SQL: SELECT, INSERT, UPDATE, DELETE. Он реализован как обёртка вокруг QSqlDatabase и QSqlQuery из фреймворка Qt, что позволяет централизовать работу с БД и упростить взаимодействие с ней из других частей приложения.

DbManager.pro:

QT -= gui

QT += sql network

TEMPLATE = lib

DEFINES += LIBDB\_LIBRARY

CONFIG += c++11

DEFINES += QT\_DEPRECATED\_WARNINGS

SOURCES += \

src/dbManager.cpp \

utils/Settings.cpp

HEADERS += \

src/dbManager.h \

utils/Settings.h

Интерфейс библиотеки DbManager:

#ifndef DBMANAGER\_H

#define DBMANAGER\_H

#include <QDate>

#include <QObject>

#include <QSqlError>

#include <QSqlQuery>

#include <QString>

#include <QTime>

#include <QVariantMap>

// Forward declaration

class DbManager : public QObject

{

Q\_OBJECT

public:

explicit DbManager( QObject \* parent = nullptr );

~***DbManager***();

bool **initialize**();

QSqlQuery **selectFromTable**( const QString table, const QMap<QString, QString> & fields = {} );

bool **deleteFromTable**( const QString table, const int & id, const QMap<QString, QString> & whereFields = {} );

bool **deleteFromTable**( const QString table, const QList<int> & id, const QMap<QString, QString> & whereFields = {} );

bool **updateFromTable**( const QString table,

const int & id,

const QMap<QString, QString> & setFields = {},

const QMap<QString, QString> & updateFields = {} );

bool **updateFromTable**( const QString table,

const QList<int> & id,

const QMap<QString, QString> & setFields = {},

const QMap<QString, QString> & updateFields = {} );

bool **insertFromTable**( const QString table, const QMap<QString, QString> & fields = {} );

QString **getWhereOrSet**( const QMap<QString, QString> & fields = {} );

QString **getInsert**( const QMap<QString, QString> & fieldsList = {} );

private:

QSqlDatabase m\_db;

};

#endif // DBMANAGER\_H

Описание функций:

DbManager(QObject \*parent) — конструктор класса, отвечающий за инициализацию подключения к базе данных. Устанавливает параметры соединения: тип драйвера , адрес сервера , имя базы данных , имя пользователя и пароль. После настройки вызывает метод initialize() для попытки подключения к серверу.

~DbManager() — деструктор класса. Проверяет, открыто ли соединение с базой данных, и, если да, закрывает его.

initialize() — метод, который пытается открыть соединение с базой данных. В случае успеха возвращает true. При ошибке выводит сообщение об ошибке в консоль через qCritical() и возвращает false.

selectFromTable(const QString table, const QMap<QString, QString> &fields) — выполняет SQL-запрос на выборку данных из указанной таблицы. Если переданы условия в параметре fields, они добавляются в часть WHERE запроса. Возвращает объект QSqlQuery, с помощью которого можно перебирать результаты запроса.

deleteFromTable(const QString table, const int &id, const QMap<QString, QString> &whereFields) — удаляет одну запись из таблицы. Если параметр whereFields пуст, удаление происходит по полю tableId = id. Если условия заданы, используется составное условие WHERE.

deleteFromTable(const QString table, const QList<int> &id, const QMap<QString, QString> &whereFields) — перегруженная версия метода удаления, предназначенная для массового удаления записей. Принимает список идентификаторов и последовательно удаляет каждую запись, вызывая одиночную версию deleteFromTable.

updateFromTable(const QString table, const int &id, const QMap<QString, QString> &setFields, const QMap<QString, QString> &whereFields) — обновляет данные одной записи в таблице по её идентификатору. Поля для обновления передаются в виде словаря setFields.

updateFromTable(const QString table, const QList<int> &id, const QMap<QString, QString> &setFields, const QMap<QString, QString> &updateFields) — перегруженная версия метода обновления, позволяющая обновить несколько записей по списку id. Выполняет цикл, вызывая одиночное обновление для каждого идентификатора.

insertFromTable(const QString table, const QMap<QString, QString> &fields) — добавляет новую запись в указанную таблицу. Принимает имя таблицы и словарь пар «поле — значение», на основе которых формируется SQL-запрос INSERT INTO ... VALUES .…

getWhereOrSet(const QMap<QString, QString> &fields) — вспомогательная функция, формирующая строку вида поле1=значение1, поле2=значение2, используемую в SQL-запросах UPDATE или WHERE. Значения вставляются напрямую, без параметризации.

getInsert(const QMap<QString, QString> &fieldsList) — вспомогательная функция, формирующая часть SQL-запроса для вставки данных: (поле1, поле2) VALUES (значение1, значение2). Используется в методе insertFromTable.

## **2.3. Разработка серверной части проекта**

Консольное приложение Server предназначено для управления подключением клиентов и отслеживании изменений в базе данных посредством образения к таблице Watcher.

Серверная часть состоит из двух классов TcpServer и DbWatcher.

Интерфейс TcpServer:

#ifndef TCPSERVER\_H

#define TCPSERVER\_H

#include "DbWatcher.h"

#include <QDateTime>

#include <QJsonDocument>

#include <QList>

#include <QObject>

#include <QSqlDatabase>

#include <QSqlError>

#include <QTcpServer>

#include <QTcpSocket>

#include <QThread>

#include <QTimer>

struct ClientInfo

{

QTcpSocket \* socket;

QString ip;

quint16 port;

int missedPings; // количество пропущенных ping

QDateTime lastActivity; // время последней активности

};

class TcpServer : public QObject

{

Q\_OBJECT

void **removeClient**( QTcpSocket \* socket );

public:

explicit TcpServer( quint16 port, QObject \* parent = nullptr );

~***TcpServer***();

bool **startServer**();

signals:

void **sendToAll**( const QString & message ); // для рассылки клиентам

private slots:

void **onNewConnection**();

void **onClientDisconnected**();

void **onClientReadyRead**();

void **onPingTimer**();

private:

QTcpServer \* m\_server;

QList<ClientInfo> m\_clients;

QSqlDatabase m\_db;

quint16 m\_port;

QTimer \* m\_pingTimer;

QThread \* m\_dbWatcherThread;

DbWatcher \* m\_dbWatcher;

};

#endif // TCPSERVER\_H

Описание функций:

TcpServer(quint16 port, QObject \*parent) — конструктор класса, инициализирующий TCP-сервер. Принимает номер порта для прослушивания и родительский объект. Внутри создаётся экземпляр QTcpServer, запускается прослушивание подключений, настраивается таймер для отправки ping-сигналов клиентам и запускается фоновый поток для отслеживания изменений в базе данных через DbWatcher.

~TcpServer() — деструктор класса. Корректно завершает работу сервера: закрывает все активные соединения с клиентами, останавливает сервер, закрывает подключение к базе данных и удаляет имя подключения из пула Qt. Также выводит диагностическое сообщение о завершении работы сервера.

startServer() — метод, запускающий прослушивание входящих подключений на указанном порту. Если сервер не может быть запущен (например, порт занят), выводится ошибка, и метод возвращает false. При успешном запуске устанавливается соединение с сигналом newConnection, который вызывает обработчик onNewConnection.

onNewConnection() — слот, вызываемый при поступлении нового подключения от клиента. Метод извлекает сокет клиента, сохраняет его вместе с IP-адресом и портом в список m\_clients, а также подключает сигналы disconnected и readyRead для отслеживания отключения и получения данных.

onClientDisconnected() — слот, срабатывающий при разрыве соединения с клиентом. Находит клиента в списке по сокету, удаляет его из списка и освобождает ресурсы сокета.

onClientReadyRead() — слот, вызываемый, когда клиент присылает данные. Читает весь доступный буфер, преобразует его в строку и анализирует содержимое. Если сообщение — "PONG", интерпретируется как ответ на ping, и активность клиента обновляется. Также метод может быть расширен для обработки команд. Обновляет метку времени последней активности клиента и сбрасывает счётчик пропущенных пингов.

removeClient(QTcpSocket \*socket) — вспомогательный метод для безопасного удаления клиента из списка. Находит клиента по сокету и удаляет его, после чего вызывает deleteLater() для освобождения сокета. Может использоваться при ручном отключении или обработке ошибок.

onPingTimer() — слот, вызываемый каждые 30 секунд таймером. Отправляет команду "PING\n" всем подключённым клиентам для проверки их активности. Увеличивает счётчик пропущенных пингов у каждого клиента. Если клиент не ответил дважды подряд, он считается неактивным и отключается. Это позволяет серверу автоматически очищать список "мертвых" соединений.

Интерфейс класса DbWatcher:

#ifndef DBWATCHER\_H

#define DBWATCHER\_H

#include <QMutex>

#include <QObject>

#include <QSqlDatabase>

#include <QTimer>

#include <QVariantMap>

class DbWatcher : public QObject

{

Q\_OBJECT

void **sleep**( int ms ) const

{

static QMutex mutex;

static QMutexLocker locker( &mutex );

mutex.tryLock( ms );

}

public:

explicit DbWatcher( const QString & dbConnectionName, QObject \* parent = nullptr );

~***DbWatcher***();

signals:

// Сигнал, который будет отправлен серверу при обнаружении изменений

void **dataChanged**( const QVariantMap & change );

public slots:

void **startMonitoring**();

private slots:

void **onTimer**();

private:

QSqlDatabase m\_db;

QTimer \* m\_timer;

int m\_lastProcessedId = 0;

bool **openDatabase**();

QList<QVariantMap> **getNewChanges**();

};

#endif // DBWATCHER\_H

Описание функций:

DbWatcher(const QString &dbConnectionName, QObject \*parent) — конструктор класса, инициализирующий объект для отслеживания изменений в базе данных. Принимает имя подключения к БД, чтобы избежать конфликтов с основным потоком сервера. Внутри создаётся клон существующего подключения к базе данных с помощью QSqlDatabase::cloneDatabase, что позволяет безопасно использовать БД в отдельном потоке. Также инициализируется таймер для периодической проверки изменений.

~DbWatcher() — деструктор класса. В текущей реализации закомментирован, что означает, что закрытие соединения с базой данных и удаление подключения из пула Qt не выполняются автоматически. Это может привести к утечкам ресурсов при завершении работы. Рекомендуется добавить корректное закрытие БД и вызов QSqlDatabase::removeDatabase().

openDatabase() — метод, отвечающий за установление соединения с базой данных MySQL. Настраивает параметры подключения (хост, имя БД, пользователь, пароль) и пытается открыть соединение. В случае неудачи (например, сервер недоступен), выводит ошибку и делает паузу на 5 секунд перед повторной попыткой. Цикл продолжается до успешного подключения. Это обеспечивает устойчивость к временным сбоям сети и позволяет DbWatcher автоматически восстанавливать соединение.

startMonitoring() — основной метод запуска мониторинга. Сначала вызывает openDatabase() для подключения к БД. Затем загружает значение последнего обработанного идентификатора (m\_lastProcessedId) из таблицы Watcher, чтобы знать, с какого изменения начинать отслеживание. Далее запускает таймер с интервалом 3 секунды, который периодически вызывает метод onTimer. После запуска выводится диагностическое сообщение о начале работы.

onTimer() — слот, вызываемый каждые 3 секунды таймером. Выполняет проверку на наличие новых записей в таблице Watcher с помощью метода getNewChanges(). Для каждого обнаруженного изменения генерируется сигнал dataChanged, который передаёт информацию о действии (вставка, удаление, обновление), таблице и идентификаторе записи. Этот сигнал перехватывается TcpServer и рассылается всем подключённым клиентам для синхронизации данных.

getNewChanges() — ключевой метод, отвечающий за извлечение новых изменений из таблицы Watcher. Выполняет SQL-запрос SELECT \* FROM Watcher, перебирает все строки и формирует список объектов QVariantMap, каждый из которых содержит:

1. id — идентификатор записи в таблице Watcher;
2. action — тип действия (например, INSERT, UPDATE, DELETE);
3. table — имя таблицы, в которой произошло изменение;
4. item\_id — идентификатор записи в целевой таблице.

После чтения всех записей выполняется DELETE FROM Watcher WHERE 1=1, чтобы очистить таблицу и избежать повторной обработки. Это реализует одноразовое уведомление о каждом изменении.

## **2.4 Разработка клиентской части проекта:**

Приложение **Client** предназначено работы администратора или пользователя с системой бизнеса.

Приложение работает с четырьмя классами:

1. MainWindow — основное окно взаимодействия с системой
2. PersonDialog — редактирование пользователя администратором
3. BookingDialog — запись клиента
4. EditDialog — редактирования журнала услуг администратором

Интерфейс MainWindow:

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QHeaderView>

#include <QJsonDocument>

#include <QJsonObject>

#include <QMainWindow>

#include <QSqlRecord>

#include <QTabWidget>

#include <QTableWidget>

#include <QTcpSocket>

#include <QVBoxLayout>

#include <QWidget>

#include "utils/Settings.h"

#include "widgets/BookingDialogs.h"

#include "widgets/EditDialog.h"

#include "widgets/PersonEditWidget.h"

#include <dbManager.h>

class MainWindow : public QMainWindow

{

Q\_OBJECT

/\*\*

\* @brief readLocalInfo - чтение данных с устройства клиента

\*/

void **readLocalInfo**();

QMap<int, QString> services;

QMap<int, QString> people;

QTcpSocket \* m\_socket;

DbManager \* manager;

QTableWidget \* firstTab;

QTableWidget \* secondTab;

QTableWidget \* thirdTab;

QWidget \* fourtTab;

public:

MainWindow( QWidget \* parent = nullptr );

~***MainWindow***();

private:

QTabWidget \* tabWidget;

QStringList availableServices;

void **init**();

QTableWidget \* **createFirstTab**(); // Дата, Время, Услуга

QTableWidget \* **createSecondTab**(); // Услуга, Цена, Записаться

QTableWidget \* **createThirdTab**(); // Дата, Время, Услуга, Цена

QWidget \* **createAdminTab**(); // Админка

QWidget \* **createFourthTab**(); // Анкета: ФИО + фото

void **connectToServer**();

private slots:

// void getMessage();

void **refreshFirstTab**();

void **refreshSecondTab**();

void **refreshThirdTab**();

void **refreshFourthTab**();

void **onEditButtonClicked**( const int & id, int row, QTableWidget \* table );

void **onBookButtonClicked**( const QString & service, const QMap<int, QString> services );

void **loadPhoto**(); // Слот для загрузки фото

};

#endif // MAINWINDOW\_H

Описание функций:

MainWindow(QWidget \*parent) — конструктор главного окна приложения. Инициализирует графический интерфейс на основе QMainWindow. Создаёт экземпляр QTcpSocket для подключения к серверу и объект DbManager для локального доступа к данным (через прямое подключение к БД). Вызывает метод connectToServer() для установления сетевого соединения. Настраивает центральный виджет и компоновку с использованием QVBoxLayout, создаёт QTabWidget для управления вкладками и последовательно добавляет четыре основные вкладки: «Записи», «Услуги», «История», «Админка» и «Анкета». Устанавливает заголовок окна и его размер.

~MainWindow() — деструктор класса. Освобождает ресурсы, удаляя объекты tabWidget и m\_socket. В текущей реализации не закрывает соединение с БД явно, так как этим управляет DbManager.

init() — вспомогательный метод, вызываемый при запуске. Загружает справочники из базы данных: список услуг (Services) и список сотрудников/клиентов (People). Данные сохраняются в ассоциативные контейнеры services и people (типа QMap<int, QString>), что позволяет быстро получать название по идентификатору. Это упрощает отображение данных в таблицах.

createFirstTab() — создаёт первую вкладку «Записи». Формирует таблицу QTableWidget с колонками: ФИО клиента, услуга, мастер, дата и кнопка «Редактировать». Данные извлекаются из таблицы Journal через DbManager. Дата преобразуется из формата ISO в читаемый вид (dd.MM.yyyy HH:mm). Для каждой строки добавляется кнопка, при нажатии на которую вызывается метод onEditButtonClicked, передавая идентификатор записи и номер строки.

createSecondTab() — создаёт вкладку «Услуги». Отображает список всех доступных услуг с указанием цены. Для каждой услуги добавляется кнопка «Записаться», при нажатии на которую вызывается onBookButtonClicked с передачей названия услуги и списка мастеров. Это открывает диалоговое окно для оформления новой записи.

createThirdTab() — формирует вкладку «История», аналогичную первой, но с другим порядком колонок: услуга, ФИО клиента, ФИО мастера, дата. Отображает все записи из журнала, позволяя просматривать прошедшие визиты. Таблица не редактируемая.

createAdminTab() — создаёт вкладку «Админка» для управления сотрудниками. Отображает таблицу с ФИО, состоянием (например, «работает»), ролью («мастер», «администратор») и кнопкой «Редактировать». При нажатии на кнопку открывается диалог PersonDialog, где можно изменить данные. Также добавлена кнопка «Добавить» для ввода нового сотрудника. Все изменения сохраняются в таблицу People через DbManager.

createFourthTab() — создаёт вкладку «Анкета», предназначенную для заполнения персональных данных клиента. Включает форму с полями: фамилия, имя, отчество, и область для отображения фотографии. Кнопка «Загрузить фото» позволяет выбрать изображение через стандартный диалог QFileDialog. Изображение масштабируется и отображается в QLabel. Для передачи ссылки на photoLabel в слот используется свойство setProperty, что позволяет обойти ограничения лямбд.

connectToServer() — метод, устанавливающий TCP-соединение с сервером по адресу 127.0.0.1:1122. При успешном подключении отправляет приветственное сообщение. Подключает сигнал readyRead для обработки входящих данных. Если приходит команда PING, клиент автоматически отвечает PONG. Если приходит JSON-сообщение об изменении данных (например, от другого клиента), вызываются методы обновления всех вкладок (refreshFirstTab, refreshSecondTab и т.д.).

refreshFirstTab() — перезагружает данные на вкладке «Записи». Очищает таблицу, повторно запрашивает данные из Journal, воссоздаёт все строки и кнопки «Редактировать». Используется для синхронизации интерфейса после получения уведомления от сервера о изменении данных.

refreshSecondTab() — перезагружает вкладку «Услуги», обновляя список услуг и кнопки «Записаться». Вызывается при изменении справочника услуг.

refreshThirdTab() — обновляет вкладку «История», перечитывая все записи из журнала. Гарантирует актуальность отображаемых данных.

refreshFourthTab() — перестраивает вкладку «Админка», обновляя список сотрудников. В текущей реализации создает новую таблицу, но не добавляет кнопки редактирования — это требует доработки.

onEditButtonClicked(int &id, int row, QTableWidget \*table) — слот, вызываемый при нажатии на кнопку «Редактировать» в таблице записей. Извлекает текущие данные (дата, услуга, мастер) и открывает диалог EditDialog, передавая идентификатор записи и справочники. После подтверждения диалога изменения сохраняются в БД через DbManager.

onBookButtonClicked(const QString &service, const QMap<int, QString> masters) — слот, вызываемый при нажатии «Записаться» на вкладке «Услуги». Открывает диалог BookingDialog, где пользователь выбирает клиента, мастера, дату и время. После подтверждения формирует словарь allValues с полями для вставки в таблицу Journal и вызывает insertFromTable.

loadPhoto() — слот, обрабатывающий загрузку фотографии. Получает ссылку на QLabel через property("photoLabel"), открывает диалог выбора файла, загружает изображение, масштабирует его под размер метки и отображает. Использует QPixmap и QFileDialog.

Интерфейc BookingDialog:

#ifndef BOOKINGDIALOGS\_H

#define BOOKINGDIALOGS\_H

#include <QComboBox>

#include <QDate>

#include <QDialog>

#include <QString>

#include <QTime>

class QLabel;

class QDateEdit;

class QTimeEdit;

class QDialogButtonBox;

class QVBoxLayout;

class QHBoxLayout;

class BookingDialog : public QDialog

{

Q\_OBJECT

public:

explicit BookingDialog( const QString & service, const QMap<int, QString> services, QWidget \* parent = nullptr );

~***BookingDialog***();

QDate **getDate**() const;

QTime **getTime**() const;

QString **getService**() const;

QString **getMaster**() const;

QString **getClient**() const;

private:

QString m\_service;

QString m\_master;

QString m\_client;

QDate m\_date;

QTime m\_time;

// Виджеты

QLabel \* serviceLabel;

QDateEdit \* dateEdit;

QTimeEdit \* timeEdit;

QComboBox \* masterCombo;

QComboBox \* clientCombo;

};

#endif // BOOKINGDIALOGS\_H

Описание функций:

BookingDialog(const QString &service, const QMap<int, QString> masters, QWidget \*parent) — конструктор диалогового окна для оформления новой записи на услугу. Принимает название услуги, которую клиент хочет получить, и список мастеров (в виде QMap<int, QString>), который используется также для заполнения списка клиентов (в текущей реализации клиенты и мастера берутся из одной таблицы People). Окно настраивается как модальное (блокирует основное окно до закрытия), устанавливается заголовок, размер и основная компоновка. Создаётся форма с полями: услуга, дата, время, мастер, клиент.

setWindowTitle("Запись на услугу") — устанавливает заголовок диалогового окна, информирующий пользователя о назначении формы.

setModal(true) — делает диалог модальным, что предотвращает взаимодействие с главным окном до завершения работы с диалогом. Это обеспечивает целостность процесса ввода данных.

resize(300, 150) — задаёт начальный размер окна, достаточный для отображения всех элементов управления.

QFormLayout \*formLayout — используется для структурированного размещения пар «метка — поле ввода». Обеспечивает чёткое и удобное для пользователя оформление формы.

serviceLabel — метка, отображающая выбранную услугу. Поле только для чтения, так как услуга передаётся из предыдущего контекста (например, с вкладки «Услуги»).

dateEdit — элемент QDateEdit, позволяющий пользователю выбрать дату записи. По умолчанию устанавливается текущая дата. Включён календарный виджет (setCalendarPopup(true)), что упрощает выбор даты мышью.

timeEdit — элемент QTimeEdit для выбора времени записи. Начальное значение — 09:00. Формат отображения — HH:mm. Пользователь может изменять время с помощью стрелок или вручную.

masterCombo — выпадающий список (QComboBox), заполняемый именами мастеров из переданного masters. Позволяет выбрать исполнителя услуги. В текущей реализации список берётся из тех же данных, что и клиенты.

clientCombo — аналогичный выпадающий список для выбора клиента. В текущей реализации также использует список masters, что указывает на то, что клиенты и сотрудники хранятся в одной таблице (People). Это упрощает реализацию, но может потребовать доработки при расширении системы.

QDialogButtonBox — стандартная панель кнопок «ОК» и «Отмена». Подключена к сигналам accepted и rejected, которые вызывают accept() и reject() соответственно, закрывая диалог с результатом QDialog::Accepted или QDialog::Rejected. Это позволяет вызывающему коду (например, MainWindow) определить, была ли запись подтверждена.

Компоновка элементов — все элементы добавляются в mainLayout, обеспечивая корректное отображение формы и кнопок в вертикальной последовательности.

~BookingDialog() — деструктор класса. Не содержит явного кода, так как все динамически созданные виджеты удаляются автоматически через иерархию родителей (Qt управляет памятью). Это соответствует best practices фреймворка Qt.

getDate() — метод-геттер, возвращающий выбранную дату в виде объекта QDate. Используется вызывающим кодом для формирования временной метки записи.

getTime() — возвращает выбранное время в виде QTime. Вместе с getDate() — позволяет собрать полную метку времени (QDateTime).

getService() — возвращает название услуги, переданное в конструктор. Гарантирует, что в процессе записи не будет ошибки из-за изменения контекста.

getMaster() — возвращает текущее значение из masterCombo, то есть ФИО выбранного мастера. Используется для определения исполнителя услуги.

getClient() — возвращает ФИО выбранного клиента из clientCombo. Позволяет связать запись с конкретным клиентом.

Интерфейс EditDialog:

#ifndef EditDialog\_H

#define EditDialog\_H

#include <QDialog>

#include <QPushButton>

#include <QString>

#include <dbManager.h>

class QLabel;

class QDateEdit;

class QTimeEdit;

class QComboBox;

class QDialogButtonBox;

class QVBoxLayout;

class QFormLayout;

class EditDialog : public QDialog

{

Q\_OBJECT

public:

explicit EditDialog( DbManager \* manager,

const int & id,

const QString & currentService,

const QMap<int, QString> & availableServices,

const QString & currentDate,

const QString & currentTime,

const QString & currentMaster,

const QMap<int, QString> & availableMaster,

QWidget \* parent = nullptr );

~***EditDialog***();

QDate **getDate**() const;

QTime **getTime**() const;

QString **getService**() const;

QString **getMaster**() const;

QMap<QString, QString> **getFields**();

void **deleteAction**();

void **updateAction**();

private:

int m\_id;

QDateEdit \* dateEdit;

QTimeEdit \* timeEdit;

QComboBox \* serviceCombo;

QComboBox \* masterCombo;

DbManager \* m\_manager;

QMap<int, QString> services;

QMap<int, QString> people;

};

#endif // EDITDIALOG\_H

Описание функций:

EditDialog(DbManager \*manager, const int &id, ...) — конструктор диалогового окна для редактирования или удаления существующей записи. Принимает указатель на объект DbManager для выполнения операций с базой данных, идентификатор записи (m\_id), текущие значения (услуга, дата, время, мастер), а также справочники доступных услуг и мастеров. Окно настраивается как модальное, устанавливается заголовок и размер. Используется вертикальная компоновка (QVBoxLayout) и форма (QFormLayout) для удобного расположения элементов ввода.

setWindowTitle("Редактирование записи") — устанавливает заголовок диалога, информирующий пользователя о назначении окна.

setModal(true) — делает диалог модальным, блокируя взаимодействие с главным окном до завершения редактирования.

resize(300, 200) — задаёт оптимальный размер окна для размещения всех элементов управления.

dateEdit — элемент QDateEdit, позволяющий изменить дату записи. Отображает текущую дату из переданного значения (currentDate) в формате dd.MM.yyyy. Включён календарный виджет для удобного выбора даты.

timeEdit — элемент QTimeEdit для изменения времени записи. Отображает текущее время (currentTime) или, по умолчанию, 09:00. Формат — HH:mm.

serviceCombo — выпадающий список (QComboBox), заполняемый названиями всех доступных услуг из справочника availableServices. Текущая услуга устанавливается как выбранная по умолчанию, что позволяет пользователю её изменить.

masterCombo — аналогичный выпадающий список для выбора мастера. Заполняется именами из справочника availableMaster. Текущий мастер предустановлен.

QDialogButtonBox — панель кнопок с тремя действиями:

«Сохранить» — применяет изменения;

«Удалить» — удаляет запись;

«Отмена» — закрывает диалог без изменений.

Каждая кнопка подключена к соответствующему слоту:

При нажатии «Удалить» вызывается метод deleteAction(), после чего диалог закрывается с результатом Accepted.

При нажатии «Сохранить» вызывается updateAction() и закрытие диалога.

«Отмена» просто отклоняет диалог (reject()).

~EditDialog() — деструктор класса. Содержит строку delete m\_manager;, что указывает на попытку освободить объект DbManager. Однако это может быть ошибкой, так как DbManager может использоваться другими частями приложения (например, MainWindow). Удаление общего объекта-менеджера в деструкторе диалога нарушает принципы управления ресурсами. Рекомендуется удалить эту строку и передавать DbManager по указателю без права владения.

getDate() — возвращает выбранную дату в виде объекта QDate. Используется для формирования временной метки.

getTime() — возвращает выбранное время в виде QTime. Вместе с getDate() позволяет собрать полное значение QDateTime.

getService() — возвращает текущее название услуги из serviceCombo. Гарантирует актуальность выбранного значения.

getMaster() — возвращает ФИО выбранного мастера из masterCombo.

getFields() — вспомогательный метод, формирующий словарь (QMap<QString, QString>) с полями, необходимыми для обновления записи в таблице Journal. Включает:

Date — строковое представление даты и времени в формате yyyy-MM-dd hh:mm:ss;

Master\_id — числовой идентификатор мастера, полученный через people.key(...);

Service\_id — идентификатор услуги, полученный через services.key(...).

Значения оборачиваются в кавычки для корректной вставки в SQL-запрос (но это не защищает от инъекций — рекомендуется использовать параметры).

deleteAction() — метод, вызываемый при нажатии кнопки «Удалить». Выполняет удаление записи из таблицы Journal по её идентификатору (m\_id) с помощью DbManager::deleteFromTable.

updateAction() — метод, вызываемый при нажатии «Сохранить». Формирует словарь полей через getFields() и передаёт его в DbManager::updateFromTable для обновления записи в базе данных.

Интерфейс PersonDialog:

#ifndef PERSONDIALOG\_H

#define PERSONDIALOG\_H

#include <QDialog>

#include <QString>

class QLineEdit;

class QComboBox;

class QFormLayout;

class QVBoxLayout;

class QHBoxLayout;

class QPushButton;

class PersonDialog : public QDialog

{

Q\_OBJECT

public:

PersonDialog( QWidget \* parent = nullptr );

PersonDialog( const QString & name, const QString & state, const QString & role, QWidget \* parent = nullptr );

~***PersonDialog***();

// Получение введённых данных

QString **getName**() const;

QString **getRole**() const;

QString **getState**() const;

private slots:

void **onAccept**(); // проверка перед принятием

private:

QLineEdit \* nameEdit;

QComboBox \* roleCombo;

QComboBox \* stateCombo;

QPushButton \* okButton;

};

#endif // PERSONDIALOG\_H

Описание функций:

PersonDialog(QWidget \*parent) — конструктор диалогового окна для добавления нового человека (клиента, мастера или администратора). Устанавливает заголовок «Добавить человека», делает окно модальным и задаёт его размер. Создаёт поля ввода: nameEdit — для ввода ФИО, и roleCombo — выпадающий список с ролями (client, master, admin). По умолчанию выбрана роль client. Форма организована с помощью QFormLayout, а кнопки размещены в QHBoxLayout с выравниванием по правому краю.

PersonDialog(const QString &name, const QString &state, const QString &role, QWidget \*parent) — перегруженная версия конструктора, используемая для редактирования существующей записи о человеке. Помимо ФИО и роли, добавляется поле stateCombo — выпадающий список состояний (active, inactive, blocked), которое отсутствует при добавлении нового пользователя. Поле ФИО предзаполняется текущим значением, что позволяет пользователю его отредактировать.

setWindowTitle("Добавить человека") — устанавливает заголовок окна. Несмотря на то, что диалог используется и для редактирования, заголовок остаётся неизменным. Для улучшения UX можно изменить его динамически в зависимости от режима («Добавить» / «Редактировать»).

setModal(true) — делает диалог модальным, что предотвращает взаимодействие с основным окном до завершения ввода данных.

resize(300, 150) — задаёт компактный, но достаточный размер окна для отображения всех элементов.

nameEdit — поле ввода QLineEdit для ФИО. Поддерживает редактирование текста. Связано с сигналом textChanged, который активирует кнопку «Сохранить» только при непустом и непробельном вводе.

roleCombo — QComboBox с фиксированным списком ролей: client, master, admin. Позволяет выбрать тип пользователя, что влияет на его права в системе.

stateCombo — дополнительное поле, доступное только в режиме редактирования. Предназначено для управления состоянием пользователя: активен, неактивен или заблокирован. Это поле может использоваться для временного отключения доступа без удаления записи из базы.

QFormLayout — используется для структурированного и выровненного размещения пар «метка — поле ввода». Обеспечивает чёткое визуальное восприятие формы.

okButton и cancelButton — кнопки «Сохранить» и «Отмена». Размещены в горизонтальном контейнере с addStretch(), что выравнивает их по правому краю. Кнопка «Отмена» закрывает диалог с результатом Rejected, а «Сохранить» вызывает слот onAccept.

connect(okButton, &QPushButton::clicked, this, &PersonDialog::onAccept) — подключение кнопки «Сохранить» к пользовательскому слоту onAccept, который включает валидацию перед закрытием диалога.

connect(nameEdit, &QLineEdit::textChanged, [...] — лямбда-функция, которая отслеживает ввод текста в поле ФИО и активирует кнопку «Сохранить» только тогда, когда введённое значение не пустое (после удаления пробелов). Это улучшает UX, предотвращая попытки сохранения пустых данных.

okButton->setEnabled(false) — изначально кнопка отключена, что соответствует состоянию пустого поля ввода.

~PersonDialog() — деструктор класса. В текущей реализации пуст, так как все виджеты удаляются автоматически через иерархию родителей (Qt-механизм управления памятью). Это соответствует best practices при использовании QObject и QWidget.

getName() — метод-геттер, возвращающий текст из поля nameEdit с удалением пробелов в начале и конце (trimmed()). Гарантирует корректность данных при сохранении.

getRole() — возвращает текущее значение из roleCombo. Используется для определения роли пользователя при добавлении или обновлении записи.

getState() — возвращает текущее состояние из stateCombo. Доступен только в режиме редактирования. Позволяет управлять активностью пользователя.

onAccept() — пользовательский слот, вызываемый при нажатии «Сохранить». Выполняет предварительную валидацию: проверяет, что поле ФИО не пустое. Если поле пустое, выводится предупреждение с помощью QMessageBox::warning, и диалог не закрывается. В противном случае вызывается accept(), и диалог завершается с результатом QDialog::Accepted, что позволяет вызывающему коду (например, MainWindow) извлечь данные и сохранить их в базу.

# 3. Проведение тестирования

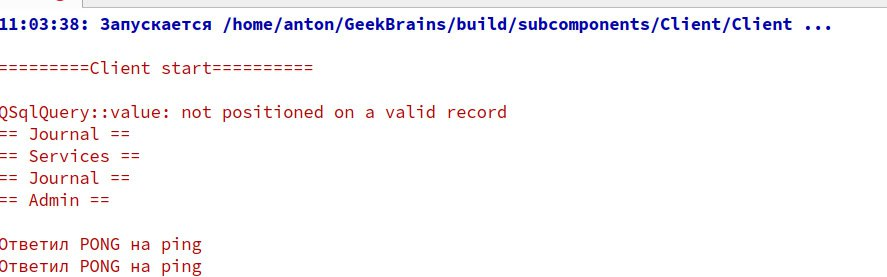


Рисунок 1 – запуск клиента



Рисунок 2 – запуск сервера

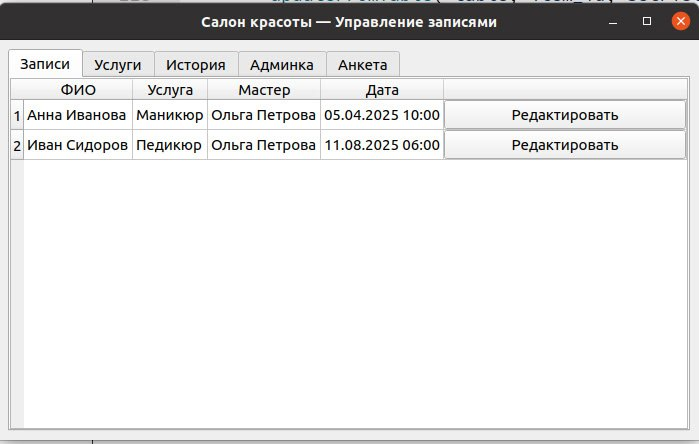


Рисунок 3 – окно вывод журнала

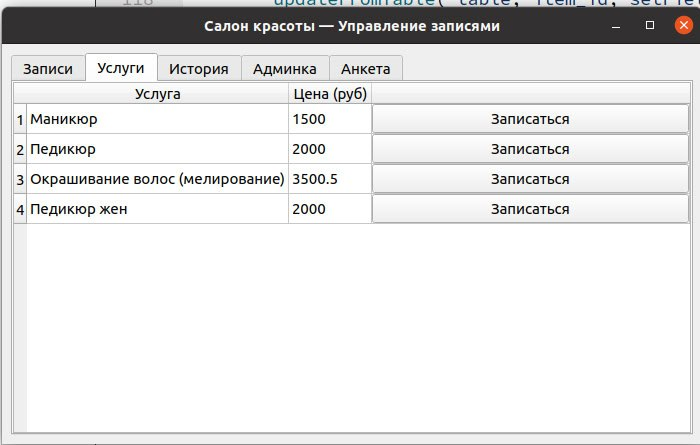


Рисунок 4 – окно услуг

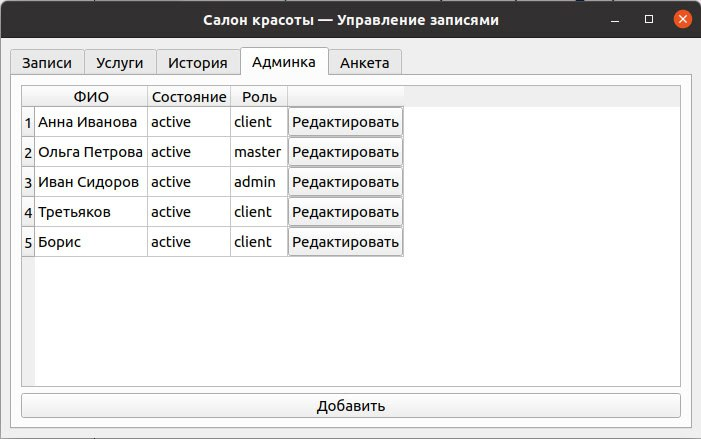


Рисунок 5 – окно пользователей

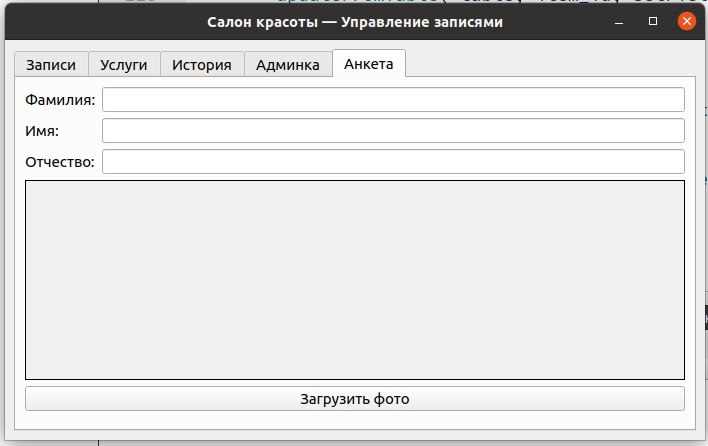


Рисунок 6 – окно личной страницы

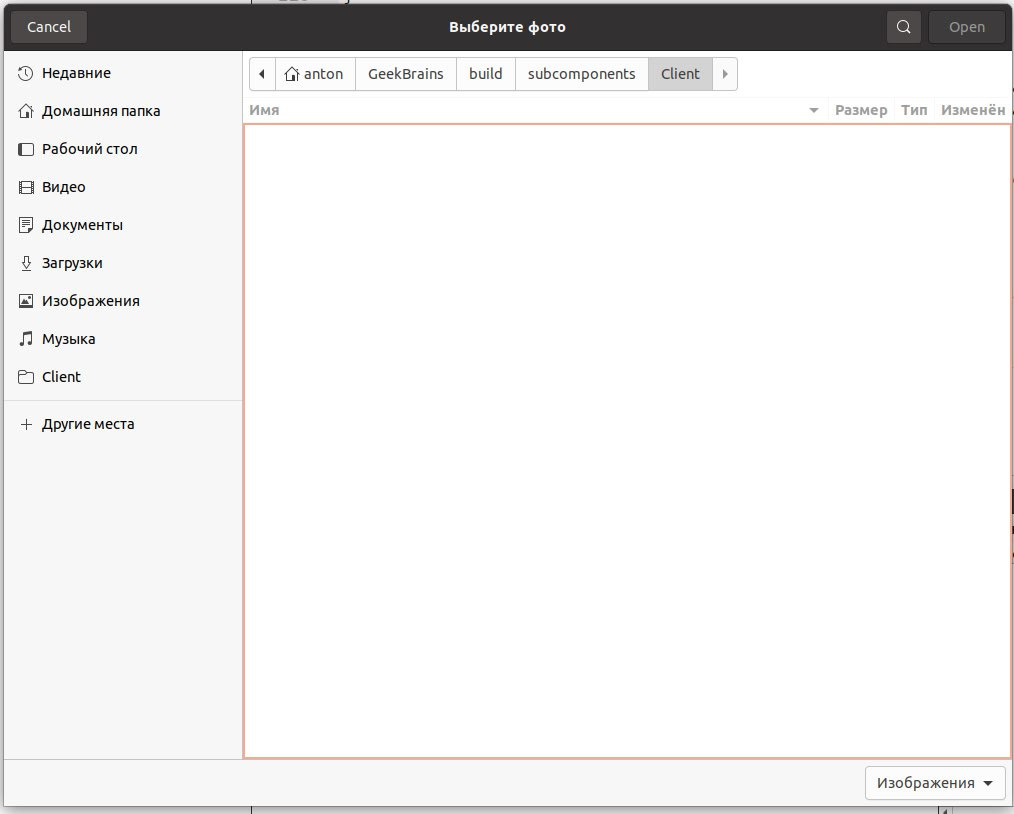


Рисунок 7 – системное окно при выборе личной фотографии

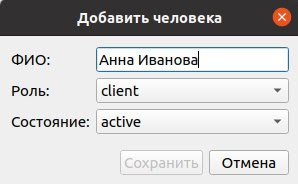


Рисунок 8 – Добавление/изменение пользователя

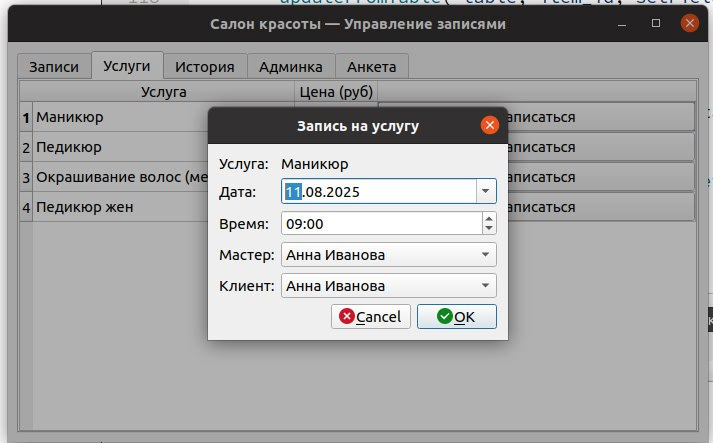


Рисунок 8 – Запись на услугу

# ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломного проекта была разработана система управления записями в салоне красоты на основе клиент-серверной архитектуры с использованием языка программирования C++ и фреймворка Qt. Проект направлен на решение актуальной задачи автоматизации бизнес-процессов в сфере малого сервисного бизнеса, где важны простота, надёжность и автономность работы.

На основе анализа существующих решений было установлено, что большинство коммерческих систем управления записями обладают избыточной функциональностью, требуют постоянного интернет-соединения и связаны с финансовыми затратами. Это делает их малопригодными для небольших салонов, особенно в регионах с ограниченным доступом к стабильному интернету.

Теоретическое исследование подтвердило, что клиент-серверная архитектура с использованием протокола TCP является оптимальным выбором для локальных сетей, обеспечивая гарантированную доставку данных, упорядоченность сообщений и высокую надёжность. В отличие от UDP и HTTP, TCP идеально подходит для задач, где потеря информации недопустима.

Практическая реализация показала, что фреймворк Qt предоставляет все необходимые инструменты для создания полноценного настольного приложения: от удобных виджетов (QTableWidget, QTabWidget, QDateEdit) до встроенной поддержки сетевого взаимодействия через QTcpSocket. Отказ от использования Qt Designer в пользу ручного программирования интерфейса позволил глубже понять архитектуру GUI-приложений и обеспечить полный контроль над поведением компонентов.

Цель проекта — разработка клиентского приложения с возможностью синхронизации данных через TCP-соединение — достигнута в полном объёме. Были выполнены все поставленные задачи:

* Спроектирован и реализован интуитивно понятный графический интерфейс с пятью вкладками;
* Реализован функционал добавления, редактирования и удаления записей;
* Создана анкета клиента с возможностью загрузки фотографии;
* Разработана админ-панель для отслеживания посещений;
* Организовано надёжное TCP-взаимодействие с сервером по порту 1122;
* Данные передаются в формате JSON, что обеспечивает гибкость и совместимость.

Работа системы была протестирована в локальной среде, включая сценарии создания записи, её редактирования и удаления. Во всех случаях данные корректно передавались на сервер, что подтвердило работоспособность клиент-серверного взаимодействия.

Разработанное приложение имеет высокую практическую значимость:

* Может использоваться в салонах красоты, парикмахерских, косметологических кабинетах и других учреждениях с посменным приёмом;
* Работает автономно, без необходимости в интернете;
* Просто в установке и использовании;
* Является бесплатным решением, что делает его доступным для малого бизнеса.

Рекомендуется внедрять подобные решения в организациях, стремящихся к цифровизации процессов без привлечения дорогостоящих CRM-систем.

В перспективе проект может быть расширен за счёт следующих улучшений:

* Реализация серверной части с хранением данных в SQLite — для долгосрочного хранения и поиска;
* Добавление аутентификации пользователей — разделение прав доступа между администратором и мастером;
* Поддержка нескольких клиентов — работа в локальной сети через IP-адрес сервера;
* Шифрование трафика с использованием SSL/TLS (QSslSocket) — повышение безопасности передачи данных;
* Генерация отчётов и экспорт в PDF/CSV — для бухгалтерии и анализа;
* Мобильное приложение или веб-интерфейс — для записи клиентов онлайн.

Цель дипломного проекта достигнута: разработано функциональное, надёжное и удобное приложение, сочетающее современные подходы к проектированию GUI и сетевому взаимодействию. Все задачи, поставленные на начальном этапе, выполнены. Гипотеза о возможности создания простого, но эффективного решения на базе Qt и TCP — подтверждена практикой.

Проведённое исследование показало, что даже при ограниченных ресурсах можно создать программное обеспечение, способное решать реальные бизнес-задачи. Разработанная система демонстрирует, как сочетание теоретических знаний и практических навыков позволяет создавать решения, востребованные в современной информационной среде.

Для дальнейшего совершенствования объекта исследования рекомендуется:

1. Внедрить автоматическое переподключение при обрыве TCP-соединения.
2. Добавить очередь отправки для буферизации данных при недоступности сервера.
3. Реализовать резервное копирование на внешний носитель.
4. Улучшить интерфейс админ-панели за счёт фильтрации по дате и услуге.
5. Интегрировать аудио- или визуальные уведомления о предстоящих записях.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Алексеева О.Г. Методические указания по экономическому обоснованию выпускных квалификационных работ бакалавров: Метод. указания, СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2013. с.
2. Горожанина, Е.И. Имитационное моделирование: учебник / Е.И. Горожанина, Е.А. Богданова. – 2-3 изд. – Самара: ПГУТИ, 2023. -300 с.
3. Емельянов, А.А. Имитационное моделирование экономических процессов / А.А. Емельянов, Е.А. Власова, Р.В. Дума – М.: Финансы и статистика, 2002. – 368 с.
4. Кельтон, В. Имитационное моделирование. В. Кельтон, А. Лоу. – СПб.: Питер , 2004. – 848 с.
5. Кутузов О.В. Имитационное моделирование систем массового обслуживания. Практикум на AnyLogic: учебное пособие для вузов/ О.И. Кутузов, Т.М. Татарникова – Санкт-Петербург : Лань, 2025. – 168 с.
6. Лимановская, О.В. Имитационное моделирование в AnyLogic 7: в 2 частях: Часть 1: учебное пособие / О.В. Лимановская – Электрон. текстовые дан. – Екатеринбург: Издательство Уральского университета, 2017
7. Мешечкин, В.В. Имитационное моделирование: учебное пособие / В.В. Мешечкин, М.В. Косенкова; Кемеровский государственный университет. – Кемерово, 2012. – 116 с.
8. Пимонов А.Г., Веревкин С.А., Прокопенко Е.В. Имитационное моделирование: электронное учебное пособие для студентов направления подготовки 38.04.02 (080200ю68) «Менеджмент» / А.Г. Пимонов, С.А. Веревкин, Е.В. Прокопенко; КузГТУ. – Кемерово, 2015.
9. Шеннон, Р. Имитационное моделирование систем – искусство и наука. Пер. с англ. [Текст] / Р. Шеннон. – М.: Мир, 1978. – 420 с.
10. Эльберг, М.С. Имитационное моделирование: учебное пособие / М.С. Эльберг, Н.С. Цынанков; Сибирский Федеральный университет. – Красноярск: СФУ, 2017. – 128 с.