Оглавление

[**ВВЕДЕНИЕ** 3](#_Toc72423160)

[**ГЛАВА I. ТЕОРЕТИЧЕСКИЕ И МЕТОДОЛОГИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ КЛИЕНТ – СЕРВЕРНЫХ ПРИЛОЖЕНИЙ** 5](#_Toc72423161)

[**1.1.** **Основные понятия** 5](#_Toc72423162)

[**1.2. Кроссплатформенный фреймворк Qt** 6](#_Toc72423163)

[**1.3. Объектная модель. Сигналы и слоты** 10](#_Toc72423164)

[**1.4. Структура проекта в Qt** 11](#_Toc72423165)

[**1.5. Клиент – серверная (распределённая) архитектура.** 12](#_Toc72423166)

[**ГЛАВА II. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА КРОССПЛАТФОРМЕННОГО РАСПРЕДЕЛЁННОГО ПРИЛОЖЕНИЯ «ЧАТ»** 14](#_Toc72423167)

[**2.1. Техническое задание** 14](#_Toc72423168)

[**2.2. Структура приложения** 16](#_Toc72423169)

[**2.3.1. Архитектура клиентского приложения** 17](#_Toc72423170)

[**2.3.2. Архитектура серверного приложения** 19](#_Toc72423171)

[**2.4. База данных** 20](#_Toc72423172)

[**2.5. Этапы создания приложения** 21](#_Toc72423173)

[**ЗАКЛЮЧЕНИЕ** 41](#_Toc72423174)

[**СПИСОК ЛИТЕРАТУРЫ** 42](#_Toc72423175)

# **ВВЕДЕНИЕ**

**Актуальность.** В текущее время трудно себе представить повседневную жизнь без обмена всевозможными видами информации. Большую часть из них составляет обмен мгновенными сообщениями через различные типы мессенджеров.

При разработке таких приложений используются всевозможные подходы для организации обмена данными. Довольно простым и в то же время эффективным способом построения мессенджера является клиент – серверная архитектура, где сообщения и прочие данные от конечных пользователей, то есть клиентов, отправляются сначала на сервер, которых может быть несколько. Там эти сообщения обрабатываются и уходят дальше одному или нескольким получателям.

Так же используются различные инструменты для построения качественного пользовательского интерфейса – UI, и эффективной и быстрой «начинки» – логики приложения.

Использование различных инструментов для разработки того или иного продукта занимает довольно длительное время. А если пытаться использовать сторонние библиотеки, которые, с первого взгляда, должны облегчать процесс разработки, не дают той простоты, удобства и эффективности, которую хотелось бы иметь.

Согласно прогнозам аналитиков, кроссплатформенная разработка программного обеспечения является будущим индустрии информационных технологий. Выпуская свое приложение на разные платформы и операционные системы, можно увеличить количество пользователей. Преимущество кроссплатформенной разработки заключается в том, время и сложность разработки значительно сокращаются. В первую очередь это связано с тем, что разработчику не нужно делать разный исходный код программы под каждую платформу и операционную систему, а достаточно работать внутри конкретного фреймворка. Также отпадает необходимость знать тонкости и нюансы каждой из платформ. Учитывая всё это, затрачиваемые на разработку трудовые ресурсы будут уменьшены и, следовательно, уменьшаться финансовые затраты на разработку.

**Цель** **работы –** реализация и изучение процесса разработки клиентского приложения и сервера, которые будут являться кроссплатформенными.

Для достижения заданной цели выделим следующие **задачи**:

* Изучение предметной области;
* Анализ методов проектирования кроссплатформенных приложений;
* Реализация.

**Структура работы.** Работа будет состоять из двух глав. В первой затронем теоретический фундамент и методы подхода. Во второй главе будут описаны этапы проектирования и разработки кроссплатформенного клиент – серверного приложения.

# **ГЛАВА I. ТЕОРЕТИЧЕСКИЕ И МЕТОДОЛОГИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ КЛИЕНТ – СЕРВЕРНЫХ ПРИЛОЖЕНИЙ**

## **Основные понятия**

**Клиентское приложение** – программа, работающая на устройстве пользователя и обеспечивающее его интерактивное взаимодействие с удалённой системой.

**Сервер** – программный компонент вычислительной системы, выполняющий сервисные и обслуживающие функции по запросу одного или нескольких клиентов, предоставляя им доступ к определённым ресурсам и услугам.

**Кроссплатформенность** – возможность работы программного обеспечения на двух или более платформах. Эта возможность обеспечивается благодаря использованию в процессе разработки высокоуровневых языков программирования и специальных сред разработки, которые, в свою очередь, поддерживают условную компиляцию, компоновку и выполнение кода для двух или более платформ. Самым простым примером является программное обеспечение, которое может работать как на *ОС Windows*, так и на *ОС Linux*, причем для сборки программного обеспечения используется один и тот же программный код.

**Библиотеки и среды выполнения** – множество библиотек, которые включаются определенными средами разработки. Входят в состав компиляторов, операционных систем или сред разработки, как сопутствующие элементы. Главной задачей является поддержка функций во время выполнения программы, от запуска до завершения ее работы.

Яркими представителями кроссплатформенных библиотек являются: *Qt*, *Boost*, *STL* (Standard Template Library), *OpenGL* и другие.

## **1.2. Кроссплатформенный фреймворк Qt**

Данный фреймворк включает в себя около большое количество всевозможных классов. Эти классы имеют строгую иерархию, которая представлена в строгой внутренней структуре, которая в свою очередь охватывает основную часть функциональных возможностей. Она не является монолитной и имеет множество модулей, которые могут взаимодействовать между собой.

Ниже представлен список модулей, доступных в Qt.

* QtCore — классы ядра библиотеки, используемые другими модулями;
* QtGui — компоненты графического интерфейса;
* QtWidgets — содержит классы для классических приложений на основе виджетов, модуль выделен из QtGui в Qt 5;
* Qt QML — модуль для поддержки QML;
* QtNetwork — набор классов для сетевого программирования. Поддержка различных высокоуровневых протоколов может меняться от версии к версии. В версии 4.2.x присутствуют классы для работы с протоколами FTP и HTTP. Для работы с протоколами TCP/IP предназначены такие классы, как QTcpServer, QTcpSocket для TCP и QUdpSocket для UDP;
* QtOpenGL — набор классов для работы с OpenGL;
* QtSql — набор классов для работы с базами данных с использованием SQL. Основные классы данного модуля в версии 4.2.х: QSqlDatabase — класс для предоставления соединения с базой, для работы с какой-нибудь конкретной базой данных требует объект, унаследованный от класса QSqlDriver — абстрактного класса, который реализуется для конкретной базы данных и может требовать для компиляции SDK базы данных. Например, для сборки драйвера под СУБД Firebird или InterBase требуются .h-файлы и библиотеки статической компоновки, входящие в комплект поставки данной СУБД;
* QtScript — классы для работы с Qt Scripts;
* QtSvg — классы для отображения и работы с данными Scalable Vector Graphics (SVG);
* QtXml — модуль для работы с XML, поддерживаются модели SAX и DOM;
* QtDesigner — классы создания расширений для своих собственных виджетов;
* QtUiTools — классы для обработки в приложении форм Qt Designer;
* QtAssistant — справочная система;
* Qt3Support — модуль с классами, необходимыми для совместимости с библиотекой Qt версии 3.х.х;
* QtTest — классы для поддержки модульного тестирования;
* QtWebKit — модуль WebKit, интегрированный в Qt и доступный через её классы. (Начиная с Qt 5.6 признан устаревшим);
* QtWebEngine — модуль Chromium, интегрированный в Qt и доступный через её классы.
* QtXmlPatterns — модуль для поддержки XQuery 1.0 и XPath 2.0;
* Phonon — модуль для поддержки воспроизведения и записи видео и аудио, как локально, так и с устройств и по сети (Начиная с Qt 5 заменён на QtMultimedia);
* QtMultimedia — модуль для поддержки воспроизведения и записи видео и аудио, как локально, так и с устройств и по сети;
* QtCLucene — модуль для поддержки полнотекстового поиска, применяется в новой версии Assistant в Qt 4.4;
* ActiveQt — модуль для работы с ActiveX и COM технологиями для Qt-разработчиков под Windows.
* QtDeclarative — модуль, предоставляющий декларативный фреймворк для создания динамичных, настраиваемых пользовательских интерфейсов.

В таблице 1.1 ниже показаны платформы, поддерживаемые данным фреймворком.

Таблица 1. Основные модули

|  |  |
| --- | --- |
| Платформа | Описание |
| **Linux/Unix** | |
| X11 | Qt для оконного менеджера X (Linux, FreeBSD, HP-UX, Solaris, AIX, и т. д.). |
|  |  |
| Wayland | Qt для Wayland. Приложения на Qt могут переключаться между графическими бэкэндами вроде X и Wayland во время загрузки, если добавить опцию командной строки – platform. Это позволяет приложениям незаметно переходить с X11 на Wayland. |
| Встраиваемые Linux-системы | Qt для встраиваемых систем: КПК, смартфонов, и т. д. Существует в виде нескольких платформ, в зависимости от технологии отрисовки. DirectFB, LinuxFB и EGLFS (EGL Full Screen). |
|  | |
| Android | Qt для Android, ранее известный как Necessitas. |
|  |  |
| **Платформы Microsoft** | |
| Windows | Qt для Microsoft Windows XP, Vista, 7, 8 и 10. |
|  |  |
| Windows CE | Qt для Windows CE 6 и Windows Embedded Compact 7. |
|  | |
| Windows RT | Поддержка для основанных на WinRT приложениях для Windows 8 и Windows Phone 8. Начиная с версии 5.4: Windows Phone 8.1. |
|  |  |

## **1.3. Объектная модель. Сигналы и слоты**

Объектная модель подразумевает то, что все построено на объектах. Класс *QObject* является базовым и практически все классы библиотеки Qt являются его наследниками. Если планируется использовать механизм сигналов и слотов, то класс должен быть наследником класса *QObject*.

*Сигналы и слоты -* средство, с помощью которых можно построить эффективное, простое для понимания и в то же время абстрактное взаимодействие между объектами путем вызова событий, вырабатываемых объектами.

Механизм сигналов и слотов заменяет и улучшает старую концепцию *callback* функций, при этом являясь объектно – ориентированным подходом.

Старая концепция *callback* функций использует процедурный подход и обычные функции, которые вызываются в результате некоторых действий. Использование данного подхода сильно усложняет исходный код программы и делает его трудно читаемым. Так же отсутствует возможность проверки типа возвращаемых значений. Это связано с тем, что во всех случаях возвращается указатель на *void* и надо собственноручно приводить типы.

Возможность соединять объекты становится одной из основных концепций написания программ с использованием Qt. Каждый класс, который унаследован от *QObject* имеет возможность как отправлять, так и принимать сигналы и иметь неограниченное количество сигналов и слотов. Сигналы могут вызывать другие сигналы и слоты, привязанные к этому сигналу, в том числе, если объекты исполняются в разных потоках. Это очень сильно упрощает доступ к общим ресурсам. Также сообщения, отправляемые с помощью сигналов, могут иметь множество аргументов любого типа. Определить соединение сигналов и слотов можно в абсолютно любой части программы.

## **1.4. Структура проекта в Qt**

Структура Qt проекта достаточно проста. Помимо исходных файлов в проекте находится файл проекта с расширением *pro*. Он необходим для вызова утилиты *qmake* и последующего создания *make*-файла. Он хранит в себе заранее предусмотренные инструкции, при помощи которых создается исполняемый модуль.

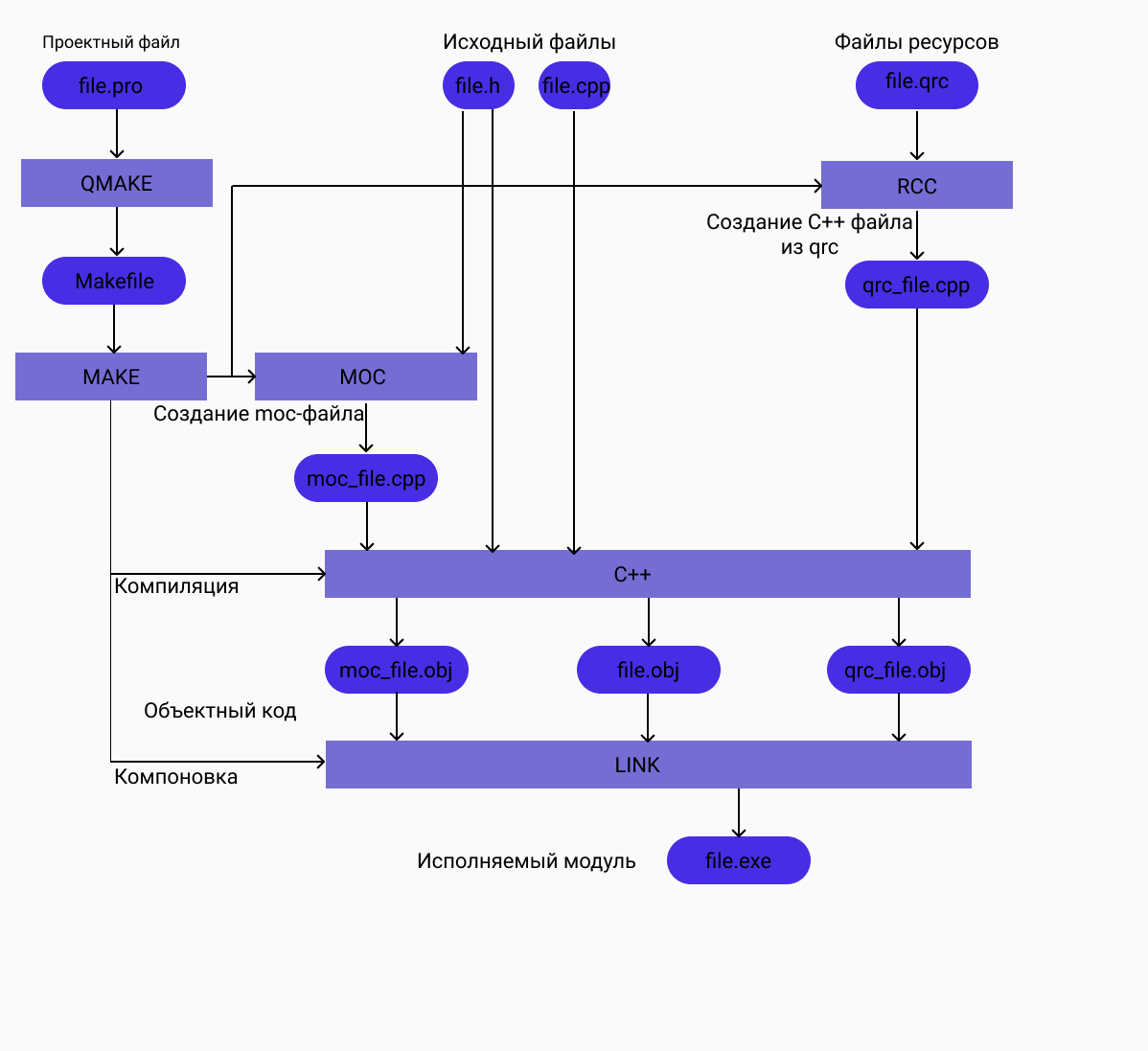


Рисунок 1. Структура Qt проекта

В случае, если в проекте имеются дополнительные ресурсы, такие как, например, картинки и шрифты, то также будет создан и файл ресурсов. После этих процедур происходит процесс компиляции в файлы объектного кода. Они, в свою очередь, объединяются линкером, в результате чего мы и получаем исполняемый файл.

## **1.5. Клиент – серверная (распределённая) архитектура.**

При использовании данной архитектуры сетевая нагрузка распределяется между поставщиками услуг или серверами, и получателями услуг – клиентами. Фактически и клиент, и сервер являются программным обеспечением. В большинстве случаев они располагаются разных устройствах и общаются между собой средствами сетевых протоколов.

Сервер напрямую влияет на общую эффективность работы клиентов, так как на сервере должны обрабатываться запросы и отправляться ответы всем подключённым клиентам. По этой причине при наличии большого количества клиентов программа сервера должна обладать большим быстродействием и, желательно, быть установленной на высокопроизводительной машине. Данные на стороне сервера обычно хранятся в базе данных. Ниже приведу простейший пример клиент – серверного взаимодействия.

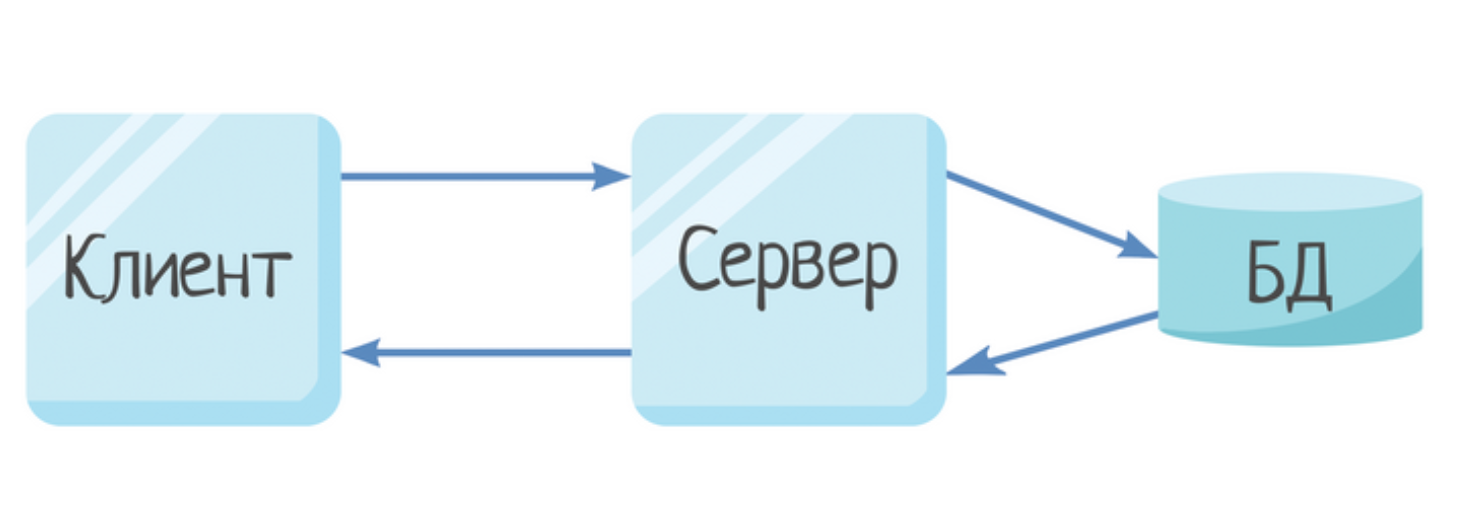


Рисунок 2. Клиент – серверная архитектура.

Клиентское приложение, однако, может быть установлено и на малопроизводительных системах. Но благодаря тому, что вычисления и обработка данных происходит на удалённом сервере, приложение будет обладать приемлемым быстродействием.

В качестве примера можно привести любой сервис онлайн – карт. Если бы они не обладали клиент – серверной архитектурой, то весь внушительный объём карт приходилось бы хранить на стороне пользователя. Так же появилась бы проблема синхронизации и обновления данных карт.

Клиент – серверная архитектура решает эти проблемы. Клиент отправляет на сервер запрос необходимой зоны карты, а сервер в ответ отправляет ему данные этой зоны.

# **ГЛАВА II. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА КРОССПЛАТФОРМЕННОГО РАСПРЕДЕЛЁННОГО ПРИЛОЖЕНИЯ «ЧАТ»**

## **2.1. Техническое задание**

Процесс разработки приложения включает в себя следующие основные этапы:

* Формирование технического задания;
* Разработка;
* Тестирование;

Мною был сформировано техническое задание.

*Клиентская часть* должна требовать минимальное количество вычислительных ресурсов, иметь понятный и минималистичный интерфейс, понятный любому пользователю. Должны иметься возможность нахождения в разных каналах, отправка как личных сообщений, так и общих всем участникам канала. Так же должна быть возможность создать канал. Форма аутентификации и регистрации, создания и присоединения к каналам. Возможность использования криптографии при передаче данных.

*Серверная часть* должна состоять из консольной программы и файла локальной базы данных, в которой хранятся аутентификационные данные пользователя и данные созданных групп. Сервер должен поддерживать одновременное подключение нескольких клиентов и работу с ними без конфликтов. В консоль приложения программа должна отображать лог действий, происходящих на сервере в реальном времени, таких как подключение, авторизация, регистрация, создание групп, подключение к ним, отправитель, тип сообщения, получатель. Возможность использования криптографии при передаче данных.

**Общие сведения о приложении**: распределённое приложение «чат». Основной задачей приложения является удобный обмен мгновенными сообщениями с возможностью выбора получателя. Так же должна быть предусмотрена возможность разделения пользователей на группы или комнаты. Интерфейс должен быть интуитивно понятен и лаконичен.

Серверная часть должна быть максимально производительной и иметь информативные логи. Так же должен быть предусмотрен способ защищённой передачи данных от клиента серверу и обратно.

**Цели и задачи приложения**:

* Возможность обмена мгновенными сообщениями;
* Защищённость данных от перехвата;
* Кроссплатформенность клиентской и серверной части;

**Целевая аудитория**: пользователи персональных компьютеров и носимой электроники.

Структура серверной части проекта:

* Консольное приложение сервера;
* Файл с базой данных;

Структура клиентской части проекта:

* Страница с формой авторизации и регистрации;
* Страница с формой создания комнаты;
* Страница с формой подключения к комнате;
* Страница с формой обмена мгновенными сообщениями;
* Страница, на которой отображаются пользователи, находящиеся в текущей комнате;

## **2.2. Структура приложения**

Файловые структуры не собранных в исполняемые файлы проектов представлена на рисунках ниже.

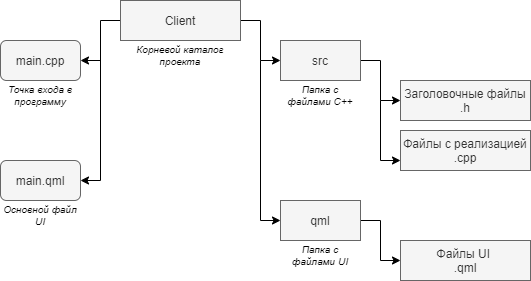


Рисунок 3. Структура проекта клиентского приложения

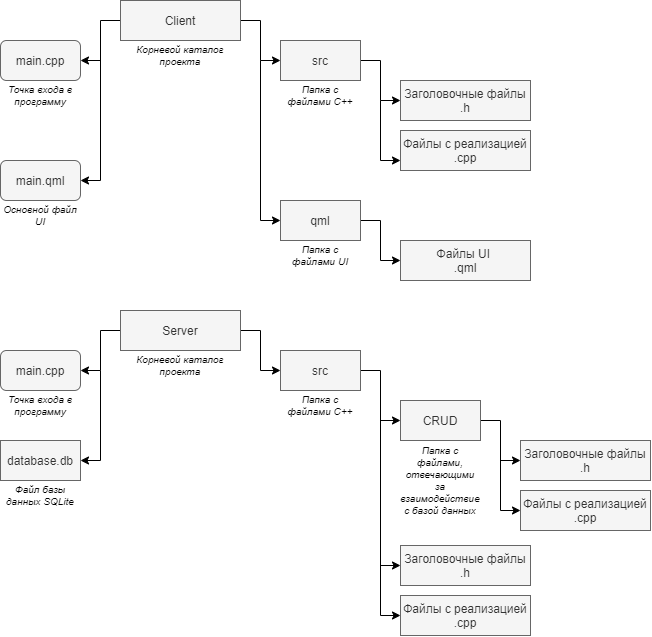


Рисунок 4. Структура проекта серверного приложения

## **2.3.1. Архитектура клиентского приложения**

Во время проектирования архитектуры клиентского приложения было учтено то, что основной задачей является обеспечить быстрое взаимодействие пользователя с данными и их отображение. Исходя из этого был выбран паттерн проектирования *Model View Controller (MVC).* Основной идеей этого шаблона является разделение хранения данных, логики взаимодействия пользователя с ними и их отображения. Со схематическим представлением этого шаблона можно ознакомиться ниже на рисунке.

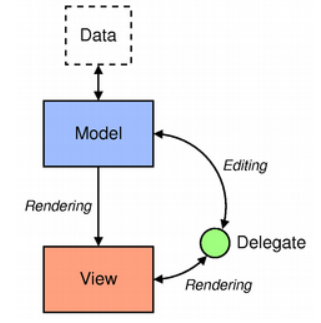


Рисунок 5. Model View Controller

Рассмотрим компоненты данного паттерна.

* Model (модель) отвечает за хранение и правильный доступ к данным.
* View (представление) отображает данные в пользовательском интерфейсе и обеспечивает правильное взаимодействие с моделью.
* Delegate (делегат) является элементом представления и определяет то, как именно будут отображаться данные.

Паттерн *Model-View* является основополагающим при проектировании графического приложения в фреймворке Qt. За доступ, хранение и управление данными отвечает ядро программы, реализованное на C++. За представление этих данных же отвечает часть, написанная на QML.

Приложение состоит из трёх основных модулей.

Модуль, отвечающий за хранение данных о текущей сессии чата. Там находятся отправленные и полученные сообщения. Так же оно отвечает за хранение данных о текущем пользователе. Реализовано в классе *ChatModel*.

Модуль, который отображает подключённых в данный момент пользователей к текущей комнате – класс *UserListModel*.

Модуль, в котором происходит защищённое взаимодействие с сервером по протоколу TCP. Реализовано классом *TCPClient*. Для того, чтобы в программе существовал только один объект данного класса, при его реализации использовался паттерн Singleton (одиночка).

Обе модели – *ChatModel* и *UserListModel* содержат в себе указатель на объект TCPClient, чтобы независимо друг от друга отправлять и получать запросы к серверу.

## **2.3.2. Архитектура серверного приложения**

Сервер должен взаимодействовать с клиентами и базой данных. Для работы с ней было принято решение использовать принцип *CRUD* – взаимодействие с данными основывается на четырёх операциях: Create, Read, Update, Delete. В языке SQL эти операции являются операторами INSERT, SELECT, UPDATE и DELETE. Этот функционал был вынесен в отдельный модуль, к которому модуль взаимодействия с клиентами имеет доступ через указатель.

Класс, отвечающий за взаимодействие других модулей с базой данных реализован с помощью паттерна Singleton (одиночка), чтобы гарантировать одну точку подключения и доступа к данным.

## **2.4. База данных**

База данных имеет очень простую структуру. Имеются три таблицы: Group\_, User\_ и Message\_. В первой хранится информация о созданных комнатах. Во второй – о зарегистрированных клиентах. Таблица Message\_ хранит информацию о всех сообщениях, прошедших через сервер: отправитель, получатель, комната и содержимое сообщения.

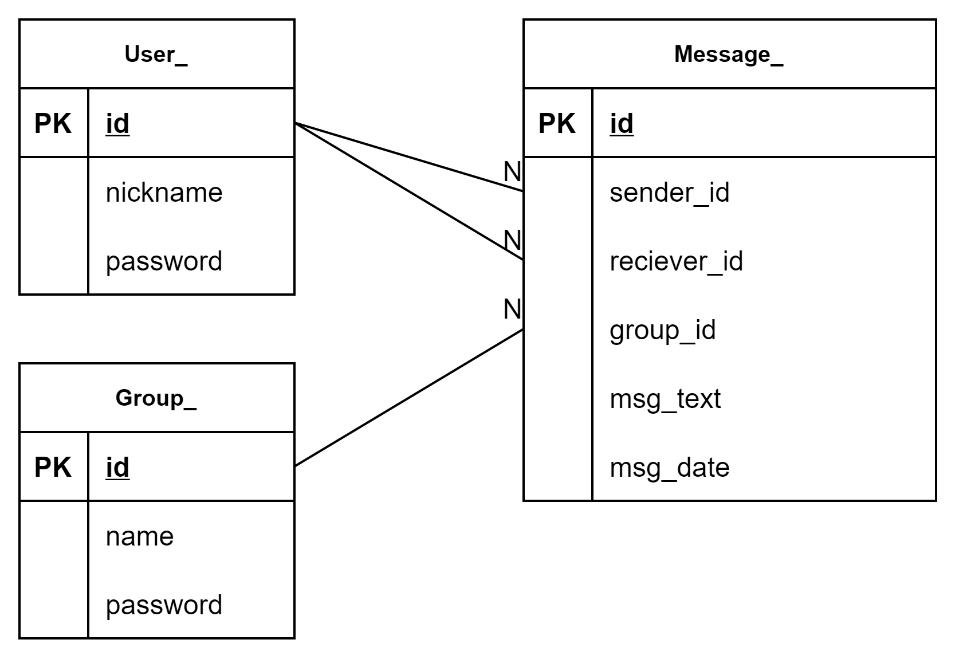


Рисунок 6. Структура базы данных

## **2.5. Этапы создания приложения**

**Выбор формата сообщений между клиентом и сервером**

Прежде всего стоит определиться с тем, каким образом будут формироваться сообщения перед отправкой по протоколу TCP. Для того, чтобы выделять отдельное сообщение из всех данных, которые поступили в сокет, будем добавлять в начало отправляемого пакета его размер. После него будем указывать тип данного сообщения. В качестве типа будем использовать восьмибитное число. В проекте используется перечисление (*enum*) для удобства. Ниже привожу все доступные типы сообщений:

enum MessageType : quint8

{

USER\_JOIN = 1,

USER\_LEFT,

USERS\_LIST,

USERS\_LIST\_REQUEST,

PUBLIC\_MESSAGE,

PRIVATE\_MESSAGE,

PRIVATE\_MESSAGE\_FAIL,

AUTH\_REQUEST,

AUTH\_SUCCESS,

AUTH\_FAIL,

REGISTER\_REQUEST,

REGISTER\_SUCCESS,

REGISTER\_FAIL,

JOIN\_GROUP\_REQUEST,

JOIN\_GROUP\_SUCCESS,

JOIN\_GROUP\_FAIL,

LEAVE\_GROUP\_REQUEST,

LEAVE\_GROUP\_SUCCESS,

LEAVE\_GROUP\_FAIL,

CREATE\_GROUP\_REQUEST,

CREATE\_GROUP\_SUCCESS,

CREATE\_GROUP\_FAIL

};

После типа сообщения будет указываться получатель, если сообщение адресовано конкретному пользователю. После будет записан текст сообщения. Ниже приведу реализацию функции, формирующей сообщения.

QByteArray TCPServer::makeByteArray(const quint8 &msg\_type, const QStringList &params)

{

QByteArray data;

QDataStream data\_stream(&data, QIODevice::WriteOnly);

data\_stream << quint16(0) << quint8(msg\_type);

for (const auto &i : params)

data\_stream << i;

data\_stream.device()->seek(0);

data\_stream << static\_cast<quint16>(data.size() - sizeof(quint16));

return data;

}

На вход функция получает константные ссылки на переменные, содержащие в себе тип сообщения и список строк с параметрами, которые будут записаны по порядку после типа сообщения. В начало вставляется вычисляемый размер сообщения. Т.к. данные требуется передаются в виде набора байт, используется встроенный класс *QByteArray* и класс для облегчения работы с ним *QDataStream*.

Для передачи таких сообщений используется класс *QTcpSocket*, который реализует абстрактный интерфейс сетевого взаимодействия по протоколу TCP/IP. Для защиты передаваемой информации использовалась модифицированная версия этого класса, обеспечивающая защиту данных при передаче. Для отправки сообщения необходимо сначала сформировать массив байт по ранее рассмотренному формату и передать его в метод *QTcpSocket::write(const QByteArray &data)* сокета, который подключён к сокету на принимающей стороне.

Принимающий сокет испускает сигнал *QTcpSocket::readyRead()*, к которому подключён слот класса – обёртки *TCPClient::onReadyRead()*. Внутри этого слота происходит разбор полученных данных. После этого, в зависимости от его типа, вызываются другие методы. Ниже приведена реализация этого слота.

void TCPClient::onReadyRead()

{

QByteArray data {socket->readAll()};

QDataStream data\_stream {&data, QIODevice::ReadOnly};

if (block\_size == 0)

{

if (quint64(data.size()) < sizeof(quint16))

{

return;

}

data\_stream >> block\_size;

}

if (quint64(data.size()) < block\_size)

{

return;

}

block\_size = 0;

quint8 type {};

data\_stream >> type;

switch (type)

{

case MessageType::REGISTER\_REQUEST:

{

QString n\_name {};

QString password {};

data\_stream >> n\_name;

data\_stream >> password;

qDebug() << "REGISTER\_REQUEST from " + n\_name;

emit registerRequest(handle, n\_name, password);

break;

}

case MessageType::AUTH\_REQUEST:

{

QString n\_name {};

QString password {};

data\_stream >> n\_name;

data\_stream >> password;

qDebug() << "AUTH\_REQUEST from " + n\_name;

emit authRequest(handle, n\_name, password);

break;

}

case MessageType::JOIN\_GROUP\_REQUEST:

{

QString group\_name {};

QString group\_password {};

data\_stream >> group\_name;

data\_stream >> group\_password;

qDebug() << "JOIN\_GROUP\_REQUEST from " + name + " to group " + group\_name;

emit joinGroupRequest(name, group\_name, group\_password);

break;

}

case MessageType::CREATE\_GROUP\_REQUEST:

{

QString group\_name {};

QString group\_password {};

data\_stream >> group\_name;

data\_stream >> group\_password;

qDebug() << "CREATE\_GROUP\_REQUEST: [" + group\_name + " : " + group\_password + "]";

emit createGroupRequest(name, group\_name, group\_password);

break;

}

case MessageType::LEAVE\_GROUP\_REQUEST:

{

emit leaveGroupRequest(name, current\_group);

break;

}

case MessageType::USERS\_LIST\_REQUEST:

{

qDebug() << "USERS\_LIST\_REQUEST from " + name + " in group: " + current\_group;

emit usersListRequest(name, current\_group);

break;

}

case MessageType::PUBLIC\_MESSAGE:

{

QString msg {};

data\_stream >> msg;

qDebug() << "PUBLIC\_MESSAGE: [" + msg + "] from " + name;

emit publicMessage(name, current\_group, msg);

break;

}

case MessageType::PRIVATE\_MESSAGE:

{

QString reciever {};

QString msg {};

qDebug() << "PUBLIC\_MESSAGE: [" + msg + "] from " + name + " to " + reciever;

emit privateMessage(name, reciever, current\_group, msg);

break;

}

}

emit socket->readyRead();

}

Реализация этого метода на сервере и клиенте не отличается ничем, кроме набора кейсов в операторе *switch()*.

**Реализация серверной части**

В качестве основного модуля реализован класс *TCPServer*, являющийся наследником встроенного класса *QTcpServer*. Используется перегрузка метода *virtual void incomingConnection(qintptr handle)* для того, чтобы корректно обрабатывать входящие подключения. Заголовочный файл класса *TCPServer:*

#pragma once

#include <QObject>

#include <QTcpServer>

#include <QTcpSocket>

#include <QDataStream>

#include <QByteArray>

#include <QHash>

#include <QCryptographicHash>

#include <QDebug>

#include "tcpclient.h"

#include "group.h"

#include "CRUD/processor.h"

namespace Server

{

class TCPServer : public QTcpServer

{

Q\_OBJECT

public:

explicit TCPServer(QObject \*parent = nullptr);

virtual void incomingConnection(qintptr handle) override;

bool start();

void stop();

private slots:

void onRegisterRequest(quintptr handle, QString name, QString password);

void onAuthRequest(quintptr handle, QString name, QString password);

void onClientDisconnected(QString name);

void onCreateGroupRequest(QString client\_name, QString group\_name, QString group\_password);

void onJoinGroupRequest(QString client\_name, QString group\_name, QString group\_password);

void onLeaveGroupRequest(QString client\_name, QString group\_name);

void onPublicMessage(QString sender, QString group\_name, QString msg);

void onPrivateMessage(QString sender, QString receiver, QString group\_name, QString msg);

void onUsersListRequest(QString client\_name, QString group\_name);

private:

static QByteArray makeByteArray(const quint8 &msg\_type, const QStringList &params);

static QByteArray makeByteArray(const quint8 &msg\_type, const QString &param = {});

QHash<quintptr, TCPClient\*> not\_auth\_clients;

QHash<QString, Group> groups;

CRUD::Processor \*crud\_processor;

};

}

Класс содержит приватные свойства:

* *QHash<quintptr, TCPClient\*> not\_auth\_clients* – хеш-таблица, содержащая сокеты не авторизированных пользователей;
* *QHash<QString, Group> groups* – хеш-таблица, содержащая объекты чат-комнат;
* *CRUD::Processor \*crud\_processor* – указатель на объект, через который происходит взаимодействие с базой данных;

Заголовочный файл класса *TCPClient:*

#pragma once

#include <QTcpSocket>

#include <QObject>

#include "types.h"

#include "tcpserver.h"

namespace Server

{

class TCPClient : public QObject

{

Q\_OBJECT

friend class TCPServer;

public:

TCPClient(quintptr handle);

private slots:

void onReadyRead();

void onDisconnected();

signals:

void authRequest(quintptr handle, QString name, QString password);

void registerRequest(quintptr handle, QString name, QString password);

void clientDisconnected(QString name);

void createGroupRequest(QString client\_name, QString group\_name, QString group\_password);

void joinGroupRequest(QString client\_name, QString group\_name, QString group\_password);

void leaveGroupRequest(QString client\_name, QString group\_name);

void publicMessage(QString sender, QString group\_name, QString msg);

void privateMessage(QString sender, QString receiver, QString group\_name, QString msg);

void usersListRequest(QString name, QString group);

private:

QTcpSocket \*socket;

quint16 block\_size;

quintptr handle;

QString name;

QString current\_group;

};

}

Класс *TCPClient* служит для обёртки стандартного класса *QTcpSocket*. Не отличается от реализации на клиентской стороне. Содержит в себе служебные приватные поля, динамический объект *QTcpSocket*, имя пользователя и название группы, в которой он состоит.

При входящем подключении к *TCPServer,* т.е. при попытке некого сокета подключиться к серверу,вызывается слот   
*void TCPServer::incomingConnection(qintptr handle).* Рассмотрим его реализацию.

void TCPServer::incomingConnection(qintptr handle)

{

TCPClient \*new\_client = new TCPClient(handle);

connect(new\_client, &TCPClient::authRequest, this, &TCPServer::onAuthRequest);

connect(new\_client, &TCPClient::registerRequest, this, &TCPServer::onRegisterRequest);

connect(new\_client, &TCPClient::clientDisconnected, this, &TCPServer::onClientDisconnected);

connect(new\_client, &TCPClient::createGroupRequest, this, &TCPServer::onCreateGroupRequest);

connect(new\_client, &TCPClient::joinGroupRequest, this, &TCPServer::onJoinGroupRequest);

connect(new\_client, &TCPClient::leaveGroupRequest, this, &TCPServer::onLeaveGroupRequest);

connect(new\_client, &TCPClient::publicMessage, this, &TCPServer::onPublicMessage);

connect(new\_client, &TCPClient::privateMessage, this, &TCPServer::onPrivateMessage);

connect(new\_client, &TCPClient::usersListRequest, this, &TCPServer::onUsersListRequest);

not\_auth\_clients[handle] = new\_client;

qDebug() << handle << "connected";

}

В метод в качестве параметра передаётся дескриптор подключающегося сокета. С помощью него создаётся ответный сокет, находящийся в классе *TCPClient*, производятся нужные соединения сигналов и слотов, далее объект *new\_client* добавляется в хеш-таблицу не авторизированных клиентов.

После этого от клиента ожидается авторизация. При получении сообщения с запросом авторизации из него берутся никнейм пользователя и захешированный пароль и вызывается метод попытки авторизации. Ниже его реализация.

void TCPServer::onAuthRequest(quintptr handle, QString name, QString password)

{

if (!crud\_processor->userExist(name))

{

QByteArray data = makeByteArray(MessageType::AUTH\_FAIL, "This user not exist");

not\_auth\_clients[handle]->socket->write(data);

qDebug() << name << "not exist";

return;

}

auto isAlreadyJoined = [this](const QString &client\_name)

{

for (const auto &i : qAsConst(groups))

{

for (const auto &j : i.clients)

{

if (j->name == client\_name)

{

return true;

}

}

}

return false;

};

if (isAlreadyJoined(name))

{

QByteArray data = makeByteArray(MessageType::AUTH\_FAIL,

"User with this nickname already on server");

not\_auth\_clients[handle]->socket->write(data);

qDebug() << "User with \"" + name + "\" nickname already on server";

return;

}

if (!crud\_processor->checkUserPassword(name, password))

{

QByteArray data = makeByteArray(MessageType::AUTH\_FAIL, "Wrong nickname or password");

not\_auth\_clients[handle]->socket->write(data);

qDebug() << name << "wrong password";

return;

}

qDebug() << name << "authorized";

not\_auth\_clients[handle]->current\_group = "None";

not\_auth\_clients[handle]->name = name;

groups["None"].clients[name] = std::move(not\_auth\_clients[handle]);

not\_auth\_clients.remove(handle);

QByteArray data = makeByteArray(MessageType::AUTH\_SUCCESS, name);

groups["None"].clients[name]->socket->write(data);

}

В нём первым делом производится поиск полученного никнейма по всем группам чтобы один пользователь не мог зайти одновременно с двух разных устройств. Далее производится запрос к базе данных, проверяющий существование записи с соответствующими данными. При успехе клиент, отправивший запрос, перемещается в хеш-таблицу клиентов группы *“None”.* При отрицательном результате поиска по базе данных клиенту отправляется сообщение о неудачной авторизации, и он остаётся в контейнере не авторизированных пользователей. Так же в консоль приложения выводятся соответствующие сообщения.

Так же от клиента может прийти запрос на вход в комнату, содержащий в себе её название и хешированный пароль. Для реализации групп объекты *TCPClient* раскладываются по соответствующим объектам *Group* и сообщения между ними происходят только в рамках этой комнаты.

struct Group

{

Group() : name {"None"}, clients {}

{ }

Group(const QString &name) :

name {name}

{ }

QString name;

QHash<QString, TCPClient\*> clients;

};

Структура *Group* содержит в себе своё название и хеш-таблицу с подключёнными к ней клиентами.

При получении запроса на вход в комнату вызывается слот *TCPServer::onJoinGroupRequest*. Ниже приведена его реализация.

void TCPServer::onJoinGroupRequest(QString client\_name, QString group\_name, QString group\_password)

{

if (!groups.contains(group\_name))

{

QByteArray data = makeByteArray(MessageType::JOIN\_GROUP\_FAIL, "Group not exist");

groups["None"].clients[client\_name]->socket->write(data);

return;

}

if (!crud\_processor->checkGroupPassword(group\_name, group\_password))

{

QByteArray data = makeByteArray(MessageType::JOIN\_GROUP\_FAIL, "Wrong password");

groups["None"].clients[client\_name]->socket->write(data);

return;

}

groups["None"].clients[client\_name]->current\_group = group\_name;

groups[group\_name].clients[client\_name] = std::move(groups["None"].clients[client\_name]);

groups["None"].clients.remove(client\_name);

qDebug() << client\_name + " [ None -> " + group\_name + " ]";

QByteArray data = makeByteArray(MessageType::JOIN\_GROUP\_SUCCESS, group\_name);

groups[group\_name].clients[client\_name]->socket->write(data);

data = makeByteArray(MessageType::USER\_JOIN, client\_name);

for (const auto &i : qAsConst(groups[group\_name].clients))

{

i->socket->write(data);

}

}

Реализация похожа на метод авторизации клиента. Сначала проверяется существование такой группы, потом к базе отправляется запрос для проверки соответствия названия и пароля. В зависимости от результата, клиент либо помещается в нужную комнату и ему отправляется сообщение об успехе, либо остаётся в группе *«None»* и получает сообщение о неудаче.

Далее рассмотрим принцип работы сервера с базой данных. Главный объект в работе с ней – *ConnectionManager.* В этом классе происходит конфигурация, настройка и слежение за валидным состоянием подключения. Ниже приведена реализация метода *ConnectionManagerPrivate::setup*.

bool ConnectionManagerPrivate::setup()

{

const QString driver {"QSQLITE"};

if(!QSqlDatabase::isDriverAvailable(driver))

{

m\_state = STATE::ERROR\_NO\_DRIVER;

qWarning() << driver << " is not available";

return false;

}

if(!setupWorkspace())

{

m\_state = STATE::ERROR\_WORKSPACE;

qWarning() << "Workspace setup failde";

return false;

}

QSqlDatabase\* database {new QSqlDatabase(QSqlDatabase::addDatabase(driver))};

m\_database.reset(database);

m\_database->setDatabaseName(m\_databasePath);

if(!m\_database->open())

{

m\_state = STATE::ERROR\_OPENING;

qWarning() << "Opening database" << m\_database->databaseName() << "failed"

<< " reason:" << m\_database->lastError().text();

return false;

}

qDebug() << "Openning database success";

return true;

}

За работу с базой отвечает объект стандартного класса *QSqlDatabase*. Ему необходимо указать название драйвера, который работает с базой и параметры подключения. В случае СУБД SQLite нужно указать только путь к файлу базы. В данном методе сначала проверяется валидность драйвера, производятся некоторые сервисные действия. Далее происходит попытка подключиться к базе. Метод возвращает результат успешности этой попытки.

Все операции, входящие в CRUD, вызываются классом *Processor.* Далее, в зависимости типа операции (SELECT, INSERT и т.д.), вызываются методы классов *Selector* или *Manipulator*. Они отвечают за формирование запросов к базе. За *SELECT* отвечает *Selector*, за *INSERT* – *Manipulator*. Эти запросы обрабатывает объект класса *Executor*. Ниже рассмотрим реализацию метода *Executor::execute.*

std::pair<RESULT, QSqlQuery> Executor::execute(const QString &queryText, const QVariantList &args)

{

if(!m\_connectionManager.isValid())

{

qCritical() << "Database isn't valid";

return std::make\_pair(RESULT::FAIL, QSqlQuery());

}

QSqlQuery query {queryText};

for(int i = 0; i < args.size(); ++i)

{

query.bindValue(i, args[i]);

}

RESULT result {RESULT::SUCCESS};

if(!query.exec() && query.lastError().isValid())

{

qCritical() << query.lastError().text() << " " << query.lastQuery();

result = RESULT::FAIL;

}

return { result, query };

}

Сначала происходит проверка состояния подключения к базе. Далее обрабатываются параметры функции. *queryText* содержит текст запроса, список *args* хранит в себе значения, которые нужно подставить в запрос. Для исполнения запроса его необходимо обернуть в класс *QSqlQuery.* В объект этого класса нужно передать текст запроса, после методом *QSqlQuery::bindValue* в запрос подставляются значения. При вызове метода *QSqlQuery::exec* запрос выполняется в текущей подключённой базе данных, и в этом же объекте оказывается возвращаемое значение запроса – некоторая таблица. Доступ к нему можно получить, поочерёдно перебирая записи таблицы методом *QSqlQuery::next* и обрабатывая их методом *QSqlQuery::value*. Метод имеет две перегрузки. В первой значение поля записи можно получить по его индексу по порядку. Во второй – по названию.

При работе серверная программа выводит в консоль логи всех происходящих событий. Пример приведён ниже.

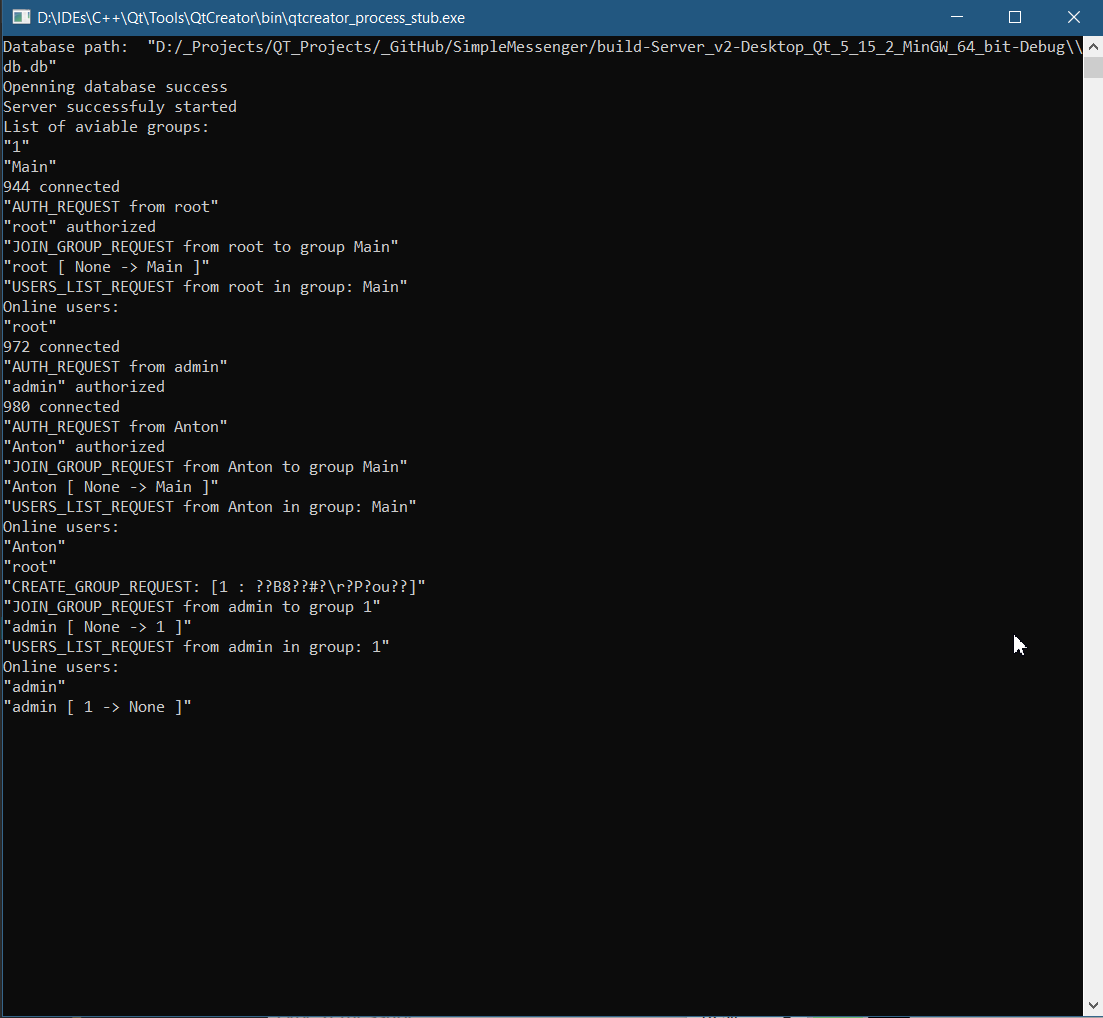


Рисунок 7. Логи сервера

**Реализация клиентской части**

Главным классом клиентского приложения является модель данных чата *ChatModel*. Объект данного класса отвечает за взаимодействие с пользователем. Класс имеет довольно много различных методов, но самое основное – приватные поля данного класса. Заголовочный файл ниже.

#pragma once

#include "tcpclient.h"

#include <QAbstractListModel>

#include <QTime>

#include <QCryptographicHash>

#include <QByteArray>

#include <QHostAddress>

#include "messageitem.h"

class ChatModel : public QAbstractListModel

{

Q\_OBJECT

Q\_PROPERTY(bool isAuth READ isAuth NOTIFY isAuthChanged)

Q\_PROPERTY(bool isJoined READ isJoined NOTIFY isJoinedChanged)

Q\_PROPERTY(QString group READ group NOTIFY groupChanged)

Q\_PROPERTY(QString nickname READ nickname NOTIFY nicknameChanged)

public:

explicit ChatModel(QObject \*parent = nullptr);

enum Roles

{

SenderRole = Qt::UserRole + 1,

MessageRole,

TimeRole,

IsMyRole,

FontColorRole,

BackColorRole

};

int rowCount(const QModelIndex &parent = QModelIndex()) const override;

QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const override;

QHash<int, QByteArray> roleNames() const override;

bool isAuth() const { return m\_isAuth; }

bool isJoined() const { return m\_isJoined; }

QString nickname() const { return m\_nickname; }

QString group() const { return m\_group; }

public slots:

void sendPrivateMsg(const QString &reciever, const QString &message);

void sendPublicMsg(const QString &message);

void signUp(const QString &nickname, const QString &password);

void signIn(const QString &nickname, const QString &password);

void signOut();

void joinGroup(const QString &group\_name, const QString &password);

void leaveGroup();

void createGroup(const QString &group\_name, const QString &password);

signals:

void isAuthChanged(bool isAuth);

void isJoinedChanged(bool isJoined);

void nicknameChanged(QString nickname);

void groupChanged(QString group);

void gs();

private slots:

void onPublicMessageRecieved (QString sender, QString message);

void onPrivateMessageRecieved(QString sender, QString reciever, QString message);

void onUserJoinRecieved(QString user);

void onUserLeftRecieved(QString user);

void onAuthSuccess(QString nickname);

void onAuthFail(QString error);

void onRegisterSuccess(QString nickname);

void onRegisterFail(QString error);

void onJoinGroupSuccess(QString group);

void onJoinGroupFail(QString error);

void onLeaveGroupSuccess();

private:

void addMsgToList(const MessageItem &msg\_item);

QByteArray hashPassword(const QString &password) const;

QList <MessageItem> m\_messages\_list;

TCPClient \*client;

QString m\_nickname;

QString m\_group;

bool m\_isAuth;

bool m\_isJoined;

};

* *QList <MessageItem> m\_messages\_list –* список сообщений, полученных и отправленных в текущую сессию и отображаемых в интерфейсе.
* *TCPClient \*client* – объект, содержащий в себе клиентский сокет, через который происходит клиент – серверное взаимодействие. Реализация совпадает с серверной частью.
* *QString m\_nickname* – никнейм текущего пользователя. Получается при успешной авторизации от сервера.
* *QString m\_group* – название комнаты, в которой в данный момент находится клиент. Так же получается от сервера при успешном подключении к комнате.
* *bool m\_isAuth* и *bool m\_isJoined* – флаги, указывающие на то, авторизован ли пользователь и подключён ли он к некоторой комнате.

Далее будет рассмотрим реализацию пользовательского интерфейса – UI. Основным его элементом является объект *ApplicationWindow* – главное окно, в котором будут отображаться элементы интерфейса. Находится этот объект в основном файле *main.qml.* Ниже приведён код.

import QtQuick 2.13

import QtQuick.Controls 2.12

import QtQuick.Layouts 1.12

import ChatModel 1.0

import UserListModel 1.0

import "qml/"

ApplicationWindow {

id: root

visible: true

width: 600

height: 800

ChatModel {

id: chat\_model

onIsAuthChanged: {

if (isAuth && isJoined) {

stack\_view.clear()

stack\_view.push("qml/ChatPage.qml")

}

else if (isAuth && !isJoined) {

stack\_view.clear()

stack\_view.push("qml/GroupJoinPage.qml")

}

else {

stack\_view.clear()

stack\_view.push("qml/LoginPage.qml")

}

}

onIsJoinedChanged: {

if (isAuth && isJoined) {

stack\_view.clear()

stack\_view.push("qml/ChatPage.qml")

}

else if (isAuth && !isJoined) {

stack\_view.clear()

stack\_view.push("qml/GroupJoinPage.qml")

}

else {

stack\_view.clear()

stack\_view.push("qml/LoginPage.qml")

}

}

}

UserListModel {

id: users\_model

}

StackView {

id: stack\_view

anchors.fill: parent

clip: false

initialItem: LoginPage {}

}

}

У *ApplicationWindow* указываются начальные значения свойств ширины и высоты. Внутри создаётся объект *ChatModel*, с будет происходить взаимодействие по схеме *Model – View.* Определяются необходимые действия, происходящие при изменении значения флагов. Далее создаётся объект *UserListModel*, который отвечает за отображение списка подключённых пользователей. После чего создаётся элемент графического интерфейса *StackView,* который будет контролировать отображение одной из выбранных страниц приложения. При разработке UI использовался стиль *Material*, часто используемый в приложениях компании Google. Рассмотрим страницы приложения.



Рисунок 8. Форма авторизации

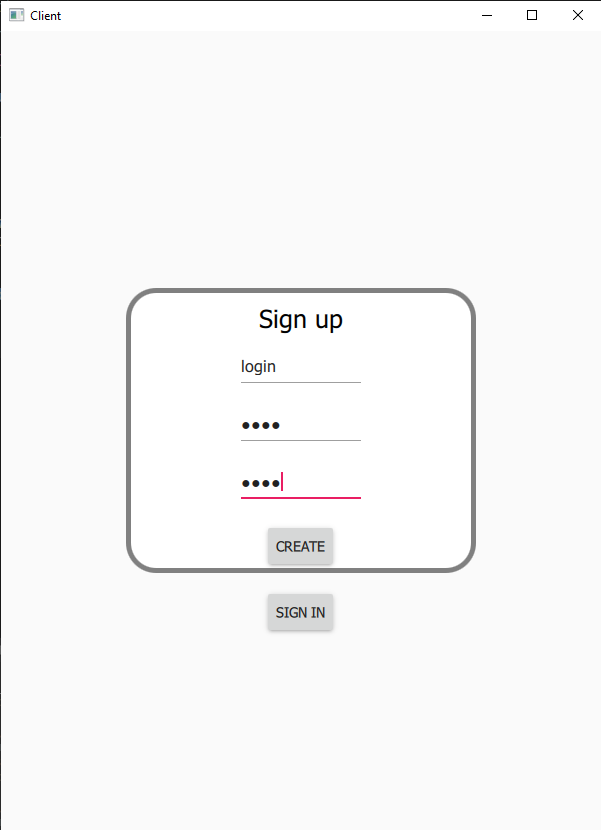


Рисунок 9. Форма регистрации

При успешной авторизации пользователь попадает на форму присоединения и создания комнаты.

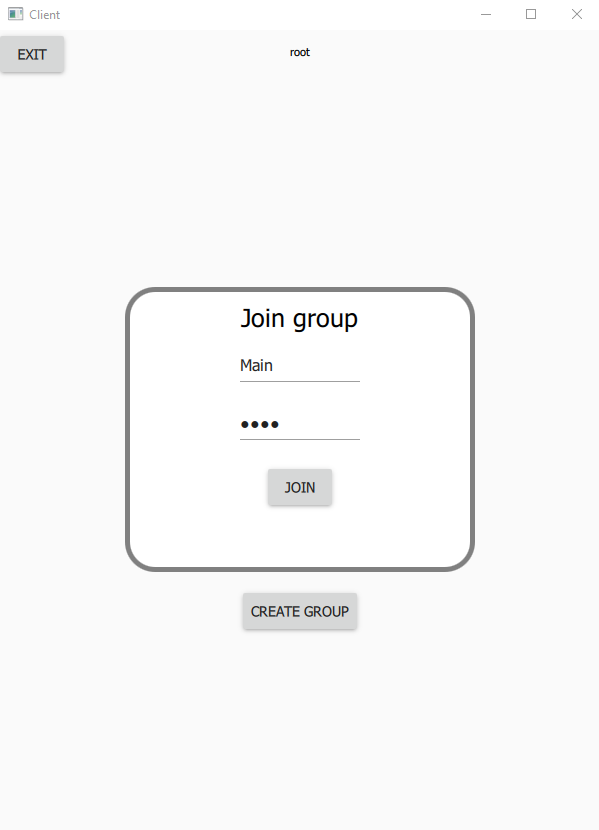


Рисунок 10. Форма присоединения к комнате

После успешного подключения к комнате пользователь переходит на страницу с чатом. При подключении или отключении его от комнаты пользователя всем остальным приходит соответствующее сообщение.

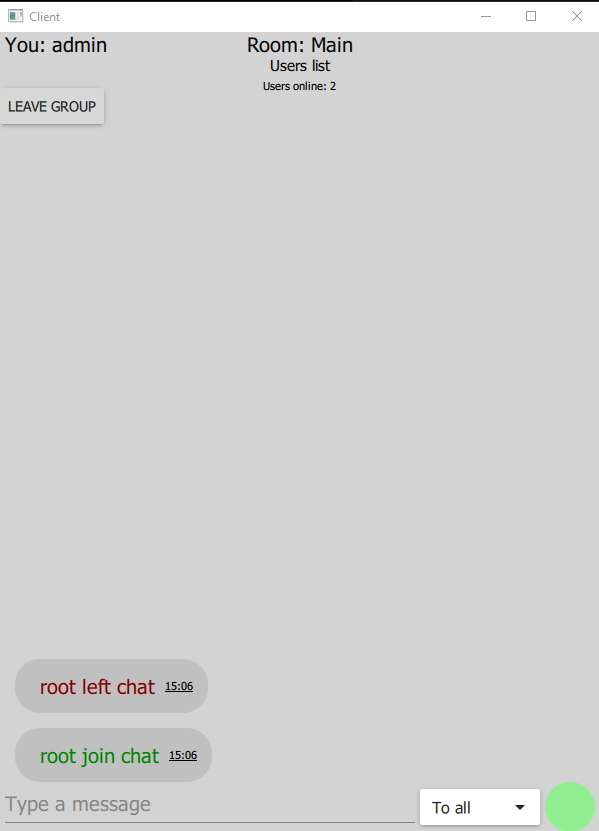


Рисунок 11. Страница чата

В верхней части формы отображается никнейм, название текущей комнаты, количество подключённых к ней пользователей. При нажатии на надпись «Users list» откроется окно со списком подключённых пользователей, выполненный в простейшем дизайне. Так же на странице чата имеется combobox с выбором типа сообщения. При выборе приватного появляется второй combobox с выбором адресата.

Рисунок 12. Виды сообщений

Сообщения отображаются следующим образом.

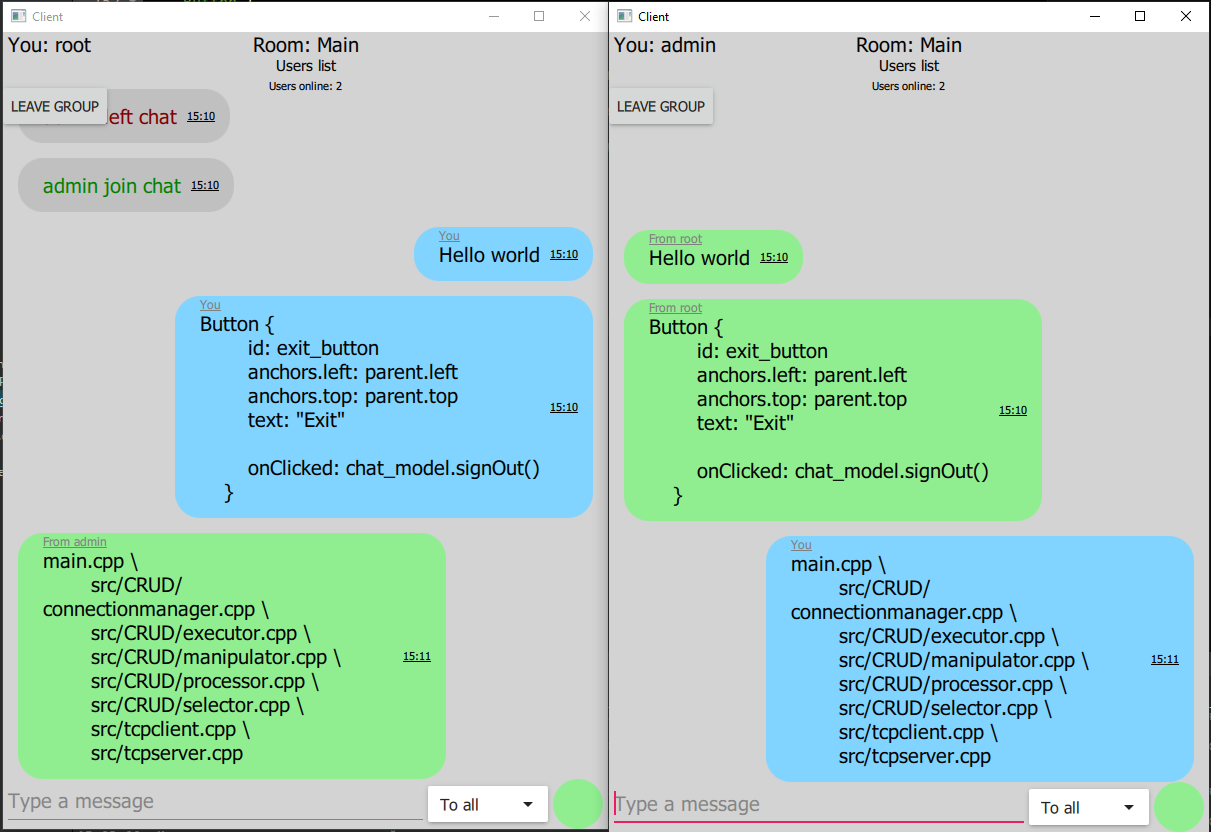


Рисунок 13. Отображение сообщений

Для входящих и исходящих сообщений предусмотрен разный цвет и положение на форме. В сообщении указывается его текст, отправитель и дата.

# **ЗАКЛЮЧЕНИЕ**

Благодаря гибкости и разнообразию фреймворка Qt в частности, и эффективности C++ стандарта 2017 года в целом, даже без использования многопоточного программирования можно быстро и качественно разрабатывать достаточно комплексные и сложные проекты. Изучив клиент – серверную архитектуру и сетевое взаимодействие, разработчику не составит труда средствами протокола TCP/IP и встроенных классов Qt реализовать распределённое приложение любой сложности. Благодаря наличию огромного количества встроенных в фреймворк классов появляется возможность без особых трудностей начинить разрабатываемую программу любым необходимым функционалом, который, благодаря скорости языка C++ и кроссплатформенности Qt, будет работать даже на самых слабых машинах и почти любой операционной системе и платформе. Так же, изучив базовые принципы криптографии, симметричных и ассиметричных шифров можно без труда организовать защищённый обмен данными между клиентом и сервером.

# **СПИСОК ЛИТЕРАТУРЫ**

1. Qt Documentation [Электронный ресурс]. – Режим доступа: <https://doc.qt.io/>, свободный (Дата обртащения 01.05.2020)
2. Книга: Qt 5.10. Профессиональное программирование на С++. Автор: Шлее Макс.
3. Книга: Effective Modern C++. Автор: Scott Meyers.
4. Habr [Электронный ресурс]. – Режим доступа: https://habr.com/ru/.
5. GitHub [Электронный ресурс]. – Режим доступа: <https://github.com/>.
6. Сервер GitLab компании «Инфоматика». – Режим доступа: <https://git.infomatika.ru/>, закрытый (только сотрудникам).
7. Сервис Draw.IO [Электронный ресурс]. – Режим доступа: <https://app.diagrams.net/>
8. Электронное учебное пособие: Математические основы защиты информации. Автор: Ш.Т. Ишмухаметов, Р.Г. Рубцова.