МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 05 01-01 Информационные системы и технологии (в проектировании и производстве)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
по дисциплине «Программирование сетевых приложений»

на тему: «КЛИЕНТ-СЕРВЕРНОЕ ПРИЛОЖЕНИЕ, РЕАЛИЗУЮЩЕЕ МНОГОПОЛЬЗОВАТЕЛЬСКИЙ ЧАТ НА ОСНОВЕ ПРОТОКОЛА HTTP»

Исполнитель: студент гр. ИТП-41
Солодков М.А
Руководитель: доцент
Курочка К.С.

Дата проверки: _____
Дата допуска к защите: _____
Дата защиты: _____
Оценка работы: _____

Подписи членов комиссии
по защите курсового проекта: _____

Гомель 2020

Установа адукацыі «Гомельскі дзяржаўны
тэхнічны ўніверсітэт імя П.В. Сухога»
Факультэт аўтаматызаваных і інфармацыйных сістэм

Кафедра _____

РЭЦЭНЗІЯ
на курсавы праект (работу)

па дысцыпліне _____ ,

выканана студэнтам _____

групы _____

I. Пералік заўваг па тэксту курсавога праекта (работы)

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

II. Агульная характарыстыка работы

_____
_____
_____
_____
_____
_____
_____
_____

III. Складаючыя агульнай адзнакі па праекту (работе)

1. Своечасовасць выканання

+	–

2. Адпаведнасць заданню

нізкі ўзровень									высокі ўзровень
1	2	3	4	5	6	7	8	9	10

3. Рацыянальнасць праектных рашэнняў

нізкі ўзровень									высокі ўзровень
1	2	3	4	5	6	7	8	9	10

4. Правільнасць выканання разлікаў

нізкі ўзровень									высокі ўзровень
1	2	3	4	5	6	7	8	9	10

5. Поўнасць выканання і якасць афармлення тлумачальнай запіскі

нізкі ўзровень									высокі ўзровень
1	2	3	4	5	6	7	8	9	10

6. Поўнасць выканання і якасць афармлення графічнай часткі (пры наяўнасці)

нізкі ўзровень									высокі ўзровень
1	2	3	4	5	6	7	8	9	10

7. Поўнасць і абгрунтаванасць высноў па праекту (рабоце)

нізкі ўзровень									высокі ўзровень
1	2	3	4	5	6	7	8	9	10

Адзнака пра допуск
праекта (работы) да абароны _____
(дата допуску, подпіс аднаго з чальцоў камісіі па праверцы)

8. Абарона курсавога праекта (работы)

нізкі ўзровень									высокі ўзровень
1	2	3	4	5	6	7	8	9	10

Агульная адзнака праекта (работы) _____
(выстаўляецца з улікам усіх складнікаў па п.п. 1…8)

« ____ » _____ 20___.
(дата)

_____

_____

_____
(подпісі, ініцыялы, П., І., І.п.б. чальцоў камісіі па праверцы)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Вопросы общения интересовали людей всегда. Для того что бы иметь возможность обмениваться какой-либо информацией не только при личной встрече, но и на очень больших расстояниях люди изобретали все новые и новые средства, такие как различные почтовые системы, протягивали кабели через континенты и океаны, запускали спутники связи. Однако это все было не совсем рационально, так как было достаточно дорого для создания и обслуживания.

С развитием информационных технологий стали возможными еще более глобальные коммуникации. Историческим «докомпьютерным» предшественником чата, несомненно, был телефон. Ни почта, ни телеграф не позволяли общаться в режиме реального времени и не были доступны в домашней обстановке. Изобретение телефона и его распространение как средства связи вызвало настоящую революцию в средствах и способах общения. Возможность поговорить с собеседником на другой стороне Земли тогда казалось настоящим чудом.

В настоящее время с люди не всегда могут пользоваться сотовой связью или общаться вживую, чтобы рассказать о чём-то важном или просто поговорить на свободную тему. Использование сетевой коммуникации позволяет решить эти проблемы. С помощью локальных и глобальных сетей у людей имеется возможность общаться практически на любом расстоянии друг от друга и в любое время.

Актуальность темы связана с необходимостью обмена информацией практически во всех сферах человеческой деятельности. С помощью чатов можно передавать одну и ту же информацию сразу нескольким людям. А главной особенностью является то, что всю информацию можно хранить в базах данных, соответственно, получить доступ к информации можно в любое время.

1 ТЕХНОЛОГИИ ДЛЯ РЕАЛИЗАЦИИ СЕТЕВОГО ВЗАИМОДЕЙСТВИЯ МЕЖДУ ПРИЛОЖЕНИЯМИ

1.1 Архитектура клиент-сервер

Архитектура клиент-сервер – это вычислительная модель, в которой сервер размещает, доставляет и управляет большей частью ресурсов и служб, которые будут использоваться клиентом. В архитектуре этого типа один или несколько клиентских компьютеров подключены к центральному серверу через сеть. Такая система разделяет вычислительные ресурсы. Архитектура клиент-сервер также известна как сетевая вычислительная модель или сеть клиент-сервер, поскольку все запросы и данные передаются по сети [1].

Клиентские компьютеры предоставляют интерфейс, позволяющий пользователю компьютера запрашивать сервисы сервера и отображать результаты, возвращаемые сервером.

Серверы ждут запросов от клиентов, а обрабатывают их, выполняют некоторую задачу, а затем формируют ответ и отправляют клиенту. В идеале сервер предоставляет клиентам стандартизированный прозрачный интерфейс, чтобы клиенты не знали о специфике системы (т.е. об аппаратном и программном обеспечении), которая предоставляет услугу. Клиенты часто находятся на рабочих станциях или на персональных компьютерах, в то время как серверы располагаются в другом месте сети, обычно на более мощных компьютерах.

Эта вычислительная модель особенно эффективна, когда у каждого клиента и сервера есть разные задачи, которые они обычно выполняют. При обработке данных больницы, например, клиентский компьютер может запускать прикладную программу для ввода информации о пациенте, в то время как серверный компьютер запускает другую программу, которая управляет базой данных, в которой хранится необходимая информация. Многие клиенты могут одновременно получать доступ к информации на сервере, и в то же время клиентский компьютер может выполнять другие задачи, такие как отправка электронной почты. Поскольку и клиентские, и серверные компьютеры считаются интеллектуальными устройствами, модель клиент-сервер полностью отличается от старой модели «мэйнфрейма», в которой централизованный мэйнфрейм выполнял все задачи для связанных с ним «немых» терминалов [2].

Характеристики и особенности клиент-серверной архитектуры:

– клиентские и серверные компьютеры могут принадлежать разным производителям;

– клиентским и серверным компьютерам требуется разное количество аппаратных и программных ресурсов;

– обладает горизонтальной масштабируемостью – увеличение количества клиентских машин, и вертикальной масштабируемостью – миграция на более мощный сервер или на мультисерверное решение;

– клиентское или серверное приложение напрямую взаимодействует с протоколом транспортного уровня для установления связи и отправки или получения информации;

– один компьютер серверного типа может одновременно предлагать несколько услуг, для каждой из которых требуется отдельная программа [3].

Стандартная клиент-серверная архитектура включает в себя два уровня: уровень клиента и уровень сервера.

Другой распространенный дизайн клиент-серверных систем использует три уровня:

– клиент, который взаимодействует с пользователем;

– сервер приложений, содержащий бизнес-логику приложения.

– менеджер ресурсов, который хранит данные [4].

Трёхуровневая клиент-серверная архитектура приведена на рисунке 1.1.

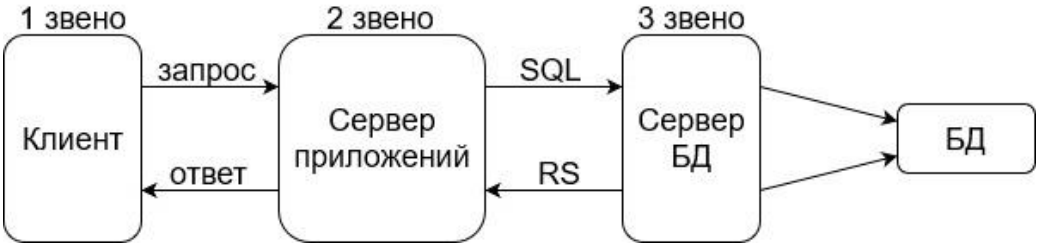

Рисунок 1.1 – Трехуровневая клиент-серверная архитектура

Клиент-серверная архитектура имеет множество преимуществ:

– развертывание клиент-серверных систем в организациях положительно увеличивает производительность за счет использования экономичных пользовательских интерфейсов, улучшенного хранилища данных, широких возможностей подключения и надежных сервисов приложений;

– улучшенный обмен данными – данные сохраняются обычными бизнес-процессами и обрабатываются на сервере и доступны для определенных пользователей (клиентов) через авторизованный доступ;

– интеграция сервисов – каждому клиенту предоставляется возможность доступа к корпоративной информации через клиентское приложение;

– общие ресурсы между различными платформами – приложения, используемые для модели клиент-сервер, создаются независимо от аппаратной платформы или технической подготовки соответствующего программного обеспечения, обеспечивая открытую вычислительную среду;

– взаимодействие с данными – все инструменты разработки, используемые для клиент-серверных приложений, обращаются к внутреннему серверу базы данных через *SQL* – стандартный язык запросов к базе данных;

– возможность обработки данных, несмотря на местонахождение – через клиент-серверную архитектуру пользователи могут напрямую входить в систему независимо от местоположения или технологии процессоров.

– простота обслуживания – поскольку архитектура клиент-сервер представляет собой распределенную модель, представляющую распределенные вычисления между независимыми компьютерами, то это предоставляет возможность обслуживать сервера, не затрагивая клиентов;

– безопасность – серверы имеют лучший контроль доступа и ресурсов, чтобы гарантировать, что только авторизованные клиенты могут получать доступ к данным или управлять ими.

Однако, несмотря на многочисленные преимущества, клиент-серверная архитектура имеет несколько недостатков:

– нагрузка на серверы – при частых одновременных запросах клиентов серверы сильно перегружаются;

– влияние централизованной архитектуры – поскольку сеть централизована, в случае отказа критически важного сервера запросы клиентов не будут обрабатываться [5].

1.2 Протоколы прикладного уровня

1.2.1 Протокол – это набор соглашений, который определяет обмен данными между различными программами. Протоколы задают способы передачи сообщений и обработки ошибок в сети, а также позволяют разрабатывать стандарты, не привязанные к конкретной аппаратной платформе [6, с. 50].

1.2.2 *HTTP (Hyper Text Transfer Protocol)* – один из прикладных протоколов *TCP/IP*, поддерживающих Интернет-соединение.

HTTP – это протокол, который позволяет получать различные ресурсы, например страницы *HTML*. Это является основой для любого обмена данными в интернете, кроме того это клиент-серверный протокол, что означает, что запросы инициируются клиентом, в качестве которого обычно выступает веб-браузер.

На рисунке 1.2 приведена схема взаимодействия клиентов (веб-браузер) с различными серверами по протоколу *HTTP*:

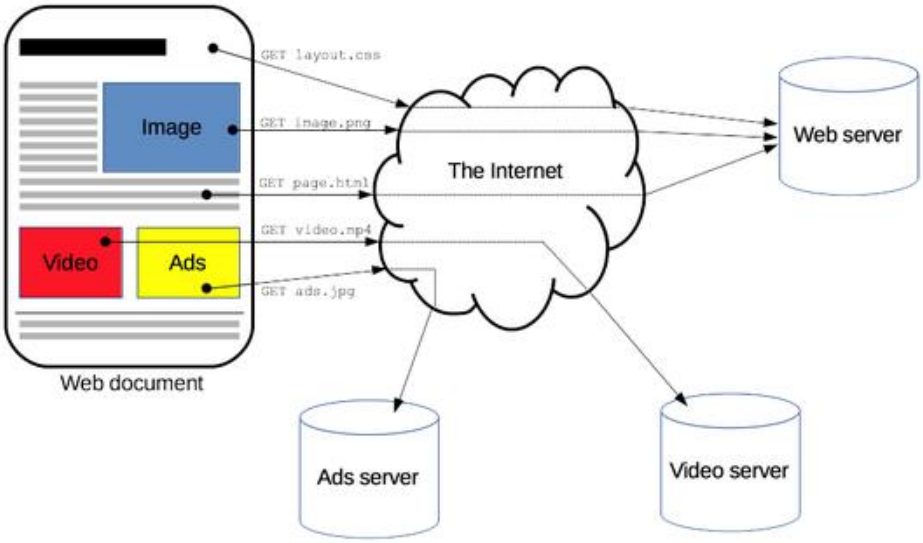

Рисунок 1.2 – Схема взаимодействия клиентов с сервером посредством протокола *HTTP*

Клиент и сервер общаются посредством обмена отдельными сообщениями. Сообщения, отправленные клиентом, в качестве которого чаще всего выступает веб-браузер, называются запросами, а сообщения, отправленные сервером в качестве ответа клиенту (веб-браузеру) называются ответами.

HTTP – протокол, который был разработан в начале 1990-х годов, представляет собой расширяемый протокол, который со временем развивался. *HTTP* протокол прикладного уровня, который пересылается через *TCP* соединение. Благодаря своей расширяемости он используется не только для получения гипертекстовых документов, но также изображений и видео или для публикации содержимого на серверах, например, данные *HTML* формы. Кроме того, *HTTP* можно использовать для получения документов для обновления веб-страниц по запросу клиента.

1.2.3 Клиент –это любой инструмент, который действует от имени пользователя. Эту роль, в основном, выполняет веб-браузер. Веб-браузер всегда является инициатором запроса. Это никогда не сервер, однако существуют различные механизмы для имитации инициируемых сервером запросов.

Что бы представить веб-страницу, браузер отправляет исходный запрос на получение *HTML*-документа, представляющего страницу. Затем он (браузер) анализирует этот файл, делая дополнительные запросы, соответствующие сценариям выполнения, информация о макете (*CSS)* для отображения и другим под

ресурсам, содержащимся на странице. Затем веб-браузер смешивает эти ресурсы, чтобы предоставить пользователю полный документ.

1.2.4 *HTTP* не имеет состояния: нет связи между двумя запросами, которые последовательно выполняются в одном и том же соединении. Это сразу может создать проблемы для пользователей, пытающихся связно взаимодействовать с определенными страницами, например, используя корзины покупок электронной коммерции. Но хотя ядро самого *HTTP* не имеет состояния, файлы *cookie HTTP* позволяют использовать сеансы с отслеживанием состояния. Используя расширяемость заголовка, *HTTP*-файлы *cookie* добавляются в рабочий процесс, что позволяет создавать сеанс для каждого *HTTP*-запроса, чтобы использовать один и тот же контекст или одно и то же состояние.

Соединение контролируется на транспортном уровне и поэтому принципиально выходит за рамки *HTTP*. Хотя *HTTP* не требует, чтобы базовый транспортный протокол был основан на соединении; только требуя, чтобы он был надежным, или не терял сообщения (чтобы как минимум представить ошибку). Среди двух наиболее распространенных транспортных протоколов в Интернете *TCP* является надежным, а *UDP* - нет. Поэтому *HTTP* полагается на стандарт *TCP*, основанный на соединении. Прежде чем клиент и сервер смогут обмениваться парой *HTTP*-запрос/ответ, они должны установить *TCP*-соединение, процесс, который требует нескольких циклов приема-передачи. По умолчанию *HTTP*/1.0 открывает отдельное *TCP*-соединение для каждой пары *HTTP*-запрос/ответ. Это менее эффективно, чем совместное использование одного *TCP*-соединения, когда несколько запросов отправляются в близкой последовательности.

На рисунке 1.3 приведена схемы запроса *HTTP*:

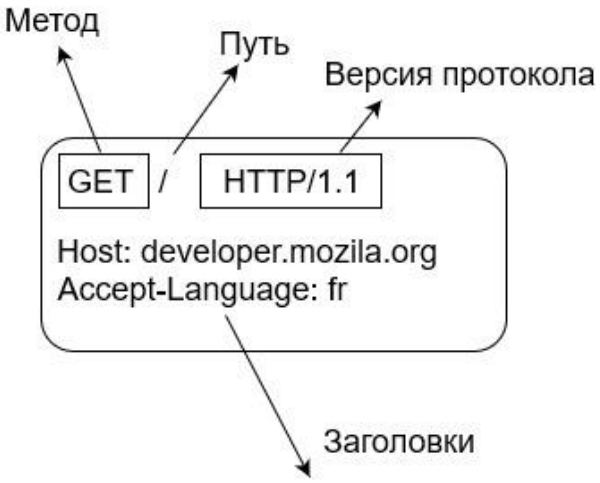

Рисунок 1.3 – Схема запроса *HTTP*

Запросы состоят из следующих элементов:

– метода *HTTP,* обычно это глагол, например, *GET, POST,* или существительное, например, *OPTIONS* или *HEAD,* который определяет, которую хочет выполнить клиент. Обычно клиент хочет получить ресурс (используется *GET)* или опубликовать значение *HTML*-формы (используется *POST)*;

– пусть к извлекаемому ресурсу. *URL*-адрес ресурса;

– версия протокола *HTTP;*

– Необязательные заголовки, передающие дополнительную информацию для сервера или серверов или тело для некоторых методов, таких как *POST,* аналогично тем, которые содержатся в ответах, которые содержат отправленный русурс.

На рисунке 1.4 приведена схема *HTTP* ответа сервера:

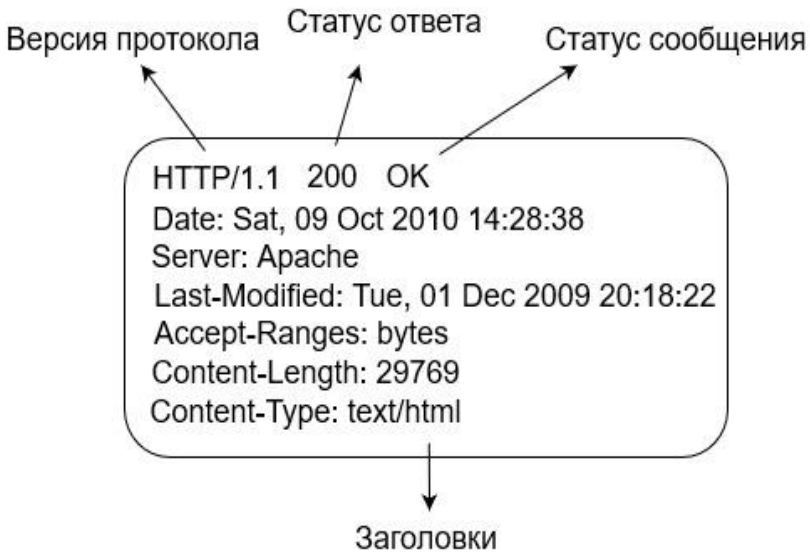

Рисунок 1.4 – Схема *HTTP* ответа

Ответы *HTTP* состоят из следующих элементов:

– версия протокола *HTTP;*

– код состояния, указывающий, был ли запрос успешным и почему;

– сообщение о состоянии, неавторизованное краткое описание состояния;

– заголовки *HTTP,* например, для запросов;

– необязательное тело, содержащее извлеченный ресурс.

HTTP – это расширяемый протокол, которым лего пользоваться. Клиент-серверная архитектура протокола в сочетании с возможностью добавления заголовков позволяет *HTTP* продвигаться вместе с расширенными возможностями интернета.

2 ЭТАПЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО КОМПЛЕКСА

2.1 Архитектура программного комплекса

Разрабатываемый программный комплекс должен функционировать с помощью сетевого взаимодействия двух приложений – сервера и клиента. Поэтому в качестве архитектуры программного комплекса выбрана клиент-серверная архитектура. Выбор такой архитектуры обусловлен наличием в ней следующих преимуществ:

– гибкость и масштабируемость системы;

– защищённая работа с данными;

– все трудоёмкие операции выполняются на стороне сервера;

– многопользовательское использование системы.

Архитектура клиент-сервер также называется «сеть клиент/сервер» или «модель сетевых вычислений», потому что в этой архитектуре все службы и запросы распространяются по сети. Его функциональность похожа на распределенную вычислительную систему, потому что все компоненты выполняют свои задачи независимо друг от друга.

Клиентская машина предоставляет удобный интерфейс, который помогает пользователям запускать службы запросов на серверном компьютере и, наконец, отображать ваши результаты в клиентской системе.

Архитектура клиент-сервер состоит из трех частей:

– *front-end* – это часть программного обеспечения, которая взаимодействует с пользователями, даже если они находятся на разных платформах с разными технологиями, любой интерфейсный модуль в архитектуре клиент-сервер предназначен для взаимодействия со всеми существующими устройствами на рынке, этот уровень содержит экраны входа в систему, меню, экраны данных и отчеты, которые предоставляют и принимают информацию от пользователей, например, большинство инструментов и фреймворков разработки позволяют создавать одну версию программы, которая работает для ПК, планшетов и телефонов;

– сервер приложений – это сервер, на котором установлены программные модули приложения, он подключается к базе данных (если она необходима и существует) и взаимодействует с пользователями (называемой клиентской частью);

– сервер базы данных (опционален) – этот сервер содержит таблицы, индексы и данные, которыми управляет приложение, здесь выполняются операции поиска и вставки / удаления / обновления.

Вот несколько преимуществ использования клиент-серверной архитектуры:

– архитектура клиент-сервер разделяет оборудование, программное обеспечение и функциональные возможности системы, например, если требуется адаптация программного обеспечения в конкретной стране, то есть необходимо изменение функциональности, его можно адаптировать в системе без необходимости разрабатывать версию для телефонов, планшетов или ноутбуков.

– поскольку архитектура разделяет аппаратное обеспечение, программное обеспечение и функциональные возможности системы, только интерфейс должен быть адаптирован для взаимодействия с различными устройствами.

Также есть несколько ключевых недостатков:

– если все клиенты одновременно запрашивают данные с сервера, он может быть перегружен.

– если сервер выйдет из строя по какой-либо причине, пользователь не сможет использовать систему.

В качестве архитектуры серверной части выбрана модульная архитектура. Её суть заключается в том, что приложение разбивается на модули, которые взаимодействуют между собой, и каждый из них решает свою поставленную задачу.

В серверном приложении выделено три модуля.

Первый модуль – модуль для принятия и отправки запросов от клиентов. Этот модуль принимает запрос от клиента, передаёт его в модуль обработки запроса и возвращает сформированный ответ клиенту.

Второй модуль – модуль для обработки запросов и формирования ответов. Он извлекает из запроса необходимые данные, выполняет операции, которые указаны в запросе, и формирует ответ для клиента. Взаимодействует с модулем для обмена данными с базой данных.

Третий модуль – модуль для обмена данными с базой данных. Реализует слой доступа к базе данных для выполнения операций чтения, сохранения, изменения и удаления записей из базы данных.

Схема модульной архитектуры для серверного приложения представлена на рисунке 2.1.

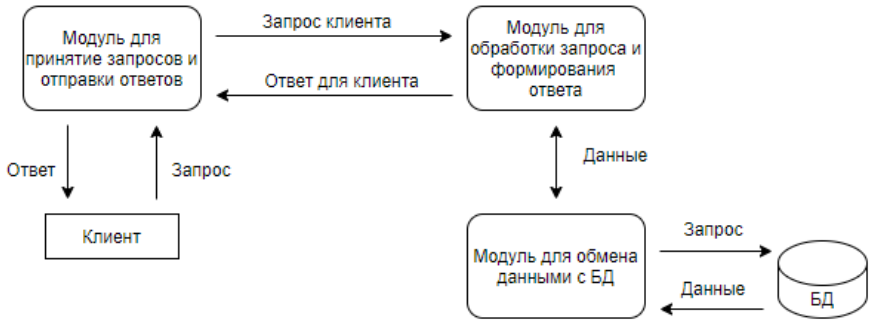

Рисунок 2.1 – Модульная архитектура серверного приложения

Такая архитектура обеспечивает простоту реализации и расширяемости приложения.

Для реализации клиентской части использовались средства *HTML* и *CSS*. Кроме того, для более удобной сборки и отладки клиентской части приложения использовался сборщик *Webpack*.

Для реализации сетевого взаимодействия между клиентом и сервером к клиентской части был добавлен модуль, который отвечат за формирование и отправку запросов и принятие ответов от сервера.

Так как данное приложение использует *HTTP*-протокол для взаимодействия между клиентом и сервером, оно является в перспективе масштабируемым и расширяемым.

2.2 Информационная модель програмнмого комплекса

Для формирования информационной модели программного комплекса первым этапом необходимо провести анализ исходных данных для выделения основных сущностей – классов однотипных объектов, информация о которых используется в модели предметной области.

В результате анализа исходных данных выделены следующие сущности:

– пользователь;

– сообщение.

Информация о пользователе содержит необходимые данные для идентификации пользователя в приложении. В приложении доступны две роли клиентов: администратор и зарегестрированный пользователь. Администратор имеет право просматривать историю сообщений любого клиента. Зарегестрированный пользователь, в отличие от администратора, имеет право просматривать только свою историю сообщений.

Информация о пользователе содержит следующие атрибуты:

– уникальный идентификатор;

– фамилия, имя отчество;

– адрес электронной почты;

– прозвище;

– роль;

– пароль.

Адрес электронной почты и пароль используются для аутентификации пользователя в приложении. При этом адрес электронной почты уникален для каждого пользователя.

Сущность «Сообщение» содержит информацию о пользователе, который его составил, а также непосредственно сам текст сообщения.

Информация о сообщении содержит следующие атрибуты:

– уникальный идентификатор;

– пользователь, отправивший сообщение в чат;

– текст сообщения.

После выделения основных сущностей необходимо установить вид связей между ними, т.е. сформировать информационно-логическую модель. В результате, исходя из выделенных сущностей, составлена схема данных, изображённая на рисунке 2.2.

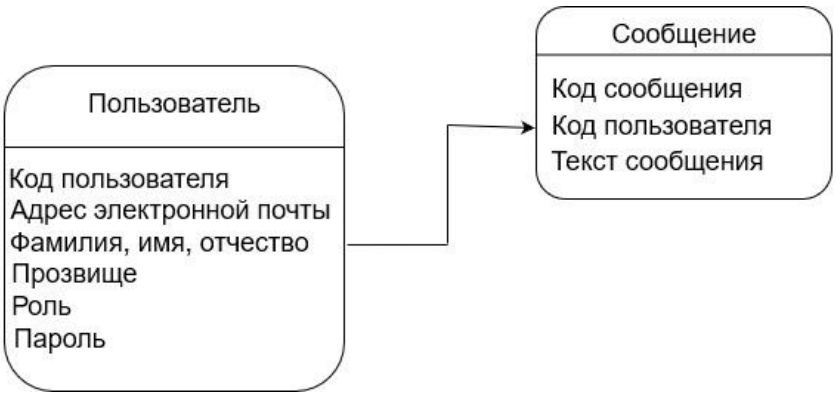

Рисунок 2.2 – Схема данных

При этом в каждой сущности для моделирования связей выделены уникальные идентификаторы. Сущность «Пользователь» связана с сущностью «Сообщение» типом связи «один ко многим», т.е. у одного пользователя может быть множество сообщений, а у одного сообщения может быть только один пользователь. Сущность «Пользователь» и сущность «Сообщение» связаны типом связи «один ко многим» с использованием внешнего ключа «Код пользователя».

После создания информационно-логической модели выполняется этап проектирования физической базы данных. Для этого информационно-логическая модель переводится в набор операторов *SQL*. Поскольку *MySQL* является реляционной базой данных, то преобразование логической модели в физическую базу данных является относительно несложной операцией. Для данного процесса преобразования существуют следующие правила:

– сущности становятся таблицами в физической базе данных;

– атрибуты становятся столбцами в физической базе данных. Также для каждого столбца необходимо определить подходящий тип данных;

– уникальные идентификаторы становятся столбцами, не допускающими значение *NULL*, т.е. первичными ключами. Также значение идентификатора делается автоинкрементным для обеспечения уникальности;

– все отношения моделируются в виде внешних ключей.

Для формирования физической модели базы данных использовались средства СУБД *MySQL Workbench*.

Диаграмма созданной базы данных представлена на рисунке 2.3.

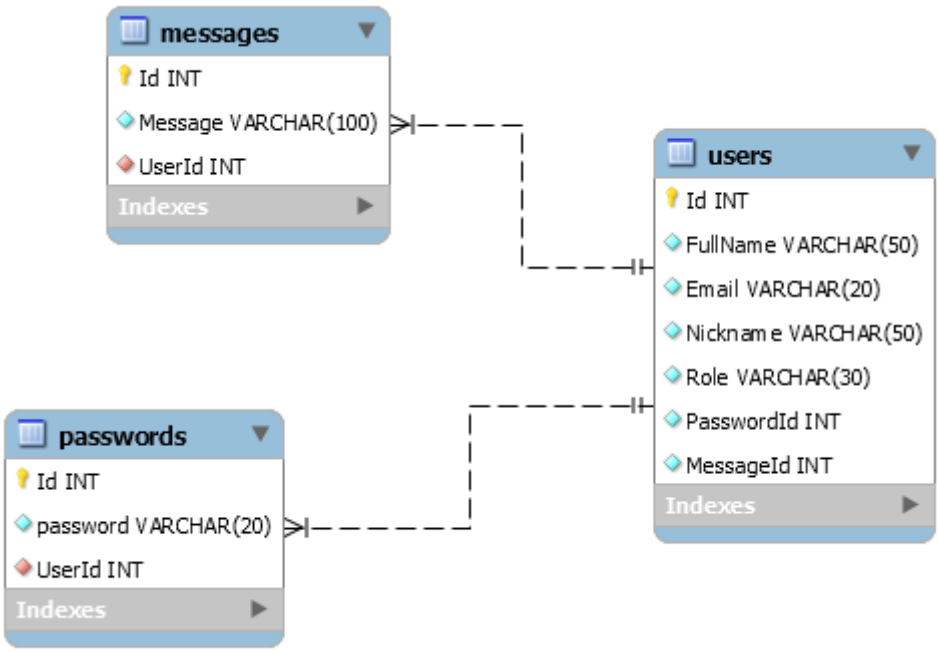

Рисунок 2.3 – Диаграмма базы данных

В разработанной физической модели базы данных выделенные на предыдущем этапе сущности моделируют следующие таблицы:

– таблица *users* моделирует сущность «Пользователь»;

– таблица *passwords* содержит атрибут «Пароль» из сущности «Пользователь»;

– таблица *messages* моделирует сущность «Сообщение»;

Решение выделить пароль пользователя в отдельную таблицу связано с безопасностью. При выборке из базы данных записи с чатом также необходимо произвести выборку пользователей, которые участвуют в этом чате. Соответственно, в данном случае информация о пароле является лишней, и, помимо этого, засекреченной. Поэтому такое разделение данных позволяет производить выборку только общедоступных данных о пользователях, а информацию о пароле – только при авторизации.

2.3 Сетевое взаимодействие между сервером и клиентом

Для реализации сетевого взаимодействия выбран протокол *HTTP*. Стандартная форма *HTTP* - это протокол передачи гипертекста. Веб-страницы создаются на языке гипертекстовой разметки, и эти веб-страницы передаются по протоколу *HTTP*. Он также использует протокол *DCP-IP* для передачи веб-страниц и другую форму *HTTP*, известную как безопасный протокол передачи гипертекста, который обеспечивает передачу данных в зашифрованном виде для предотвращения утечки конфиденциальных данных.

Данные, которыми обмениваются клиент и сервер, имеют формат *JSON*. Выбор формата обусловлен тем, что данные в этом формате очень просто формировать и обрабатывать. Сообщения, которыми обмениваются клиент и сервер, имеют три обязательных ключа:

– *action* – используется сервером и определяет действие, которое сервер должен выполнить;

– *body* – содержит тело сообщения;

– *status* – результат выполнения запроса.

При формировании запроса на сервер клиент указывает *action*, чтобы сервер понял, что необходимо сделать. В зависимости от необходимых действий клиент может поместить необходимые данные в поле *body*.

При формировании ответа клиенту сервер указывает *status*, который зависит от результата обработки и выполнения запроса. Также если клиент запрашивает данные, то сервер помещает их в поле *body*.

Если запрос клиента имеет неправильный формат, то сервер возвращает ответ с соответствующим значением в поле *status*.

Обмен данными происходит с помощью *websocket.* Для обмена ифнформацией на сервер был создан специальный класс, который называется конечной точкой приложения. Для настройки конечной точки использовались специальные аннотации на уровне методов. Для обмена данными между пользователем и сервером на сервере были созданы кодеры и декодеры, которые необходимы для преобразования объектов *Java* в *JSON* и наоборот.

3 СТРУКТУРА ПРОГРАММНОГО КОМПЛЕКСА

Для реализации клинет-серверного приложения был использован такой язык программирования как *Java*. Выбор обусловлен наличием в нём удобных классов для реализации сетевого взаимодействия с помощью протокола *Web-Socket.* Также этот язык является объектно-ориентированным, что позволяет представлять программу в виде набора классов и связей между ними.

Для реализации слоя доступа к данным использовался фреймворк *Hibernate,* который выполняет объектно-реляционное отображение, тем самым сокращая время на разработку слоя доступа к данным.

Для моделирования сущностей предметной области были реализованы следующие классы:

– *Message* – моделирует сущность «Сообщение»;

– *Password* – хранит пароль из сущности «Пользователь»;

– *User* – моделирует сущность «Пользователь».

Для корректной работы с фреймворком *Hibernate* классы, поля и методы помечаются специальными аннотациями:

– *Entity* – указывает на то, что класс, с такой аннотацией является сущностью;

– *Table* – хранит информацию о таблице, которую моделирует класс;

– *Id* – указывает на то, что проаннотированное поле является уникальным идентификатором таблицы;

– *Column* – хранит информацию о столбце в таблице базы данных, которому соответствует поле класса;

– *OneToMany* – хранит информацию о том, с какой сущностью имеется связь один ко многим;

– *OneToOne* – хранит информацию о том, с какой сущностью имеется связь один к одному.

Для передачи объектов этих классов по сети реализованы два класса: кодер и декодер. Эти два класса конвертирует передаваемые данные в формат *JSON* и обратно соответственно. Список реализованных классов для передачи объектов по сети соответствующий:

– класс *MessageEncoder,* который используется для кодирования объекта в формат *JSON.* Кодирование в формат *JSON* происходит с помощью *GSON.*

– класс *MessageDecoder,* который используется для декодирования объекта *JSON* в объект *Java*.

Диаграмма разработанных классов сущностей представлена на рисунке 3.1:

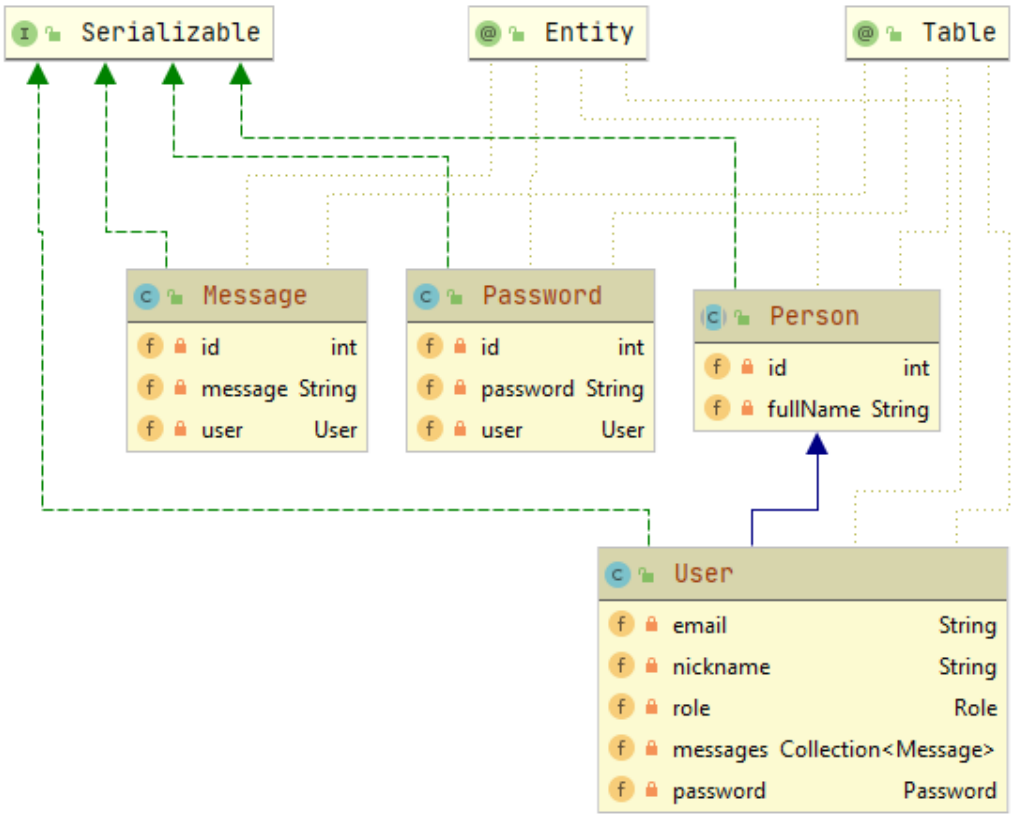

Рисунок 3.1 – Диаграмма классов сущностей.

Для выполнения операций с базой данных реализованы соответствующие классы с помощью шаблона проектирования *DAO (Data Access Object)*.

Для выполнения *CRUD* операций с базой данных реализован класс *HibernateSessionFactoryUtil* с помощью паттерна *Singleton*. Данный класс необходим для создания сессий, которые необходимы для осуществления запросов к базе данных.

Далее для определения методов, которые необходимы для выполнения *CRUD* операций были реализованы следуюище интерфейсы:

– *MessageService* – данный интерфейс необходим для определения методов, необходимых для *CRUD* операций к таблице *Messages;*

– *PasswordService* – данный интерфейс необходим для определения методов, нобходимых для *CRUD* операций к таблице *Passwords;*

– *UserService* – интерфейс, определяющий методы для *CRUD* операций к таблице *Users.*

Для выполнения *CRUD* операций к необходимым таблицам в базе данных созданы классы которые имплементируют вышеописанные интерфейсы:

– *MySqlMessageServiceImpl* – класс, который реализует методы, описанные в интерфейсе *MessageService;*

– *MySqlPasswordServiceImpl* – класс, который реализует методы, описанные в интерфейсе *PasswordService;*

– *MySqlUserServiceImpl* – класс, который реализует методы, описанные в интерфейсе *UserService.*

Для доступа к таблице *Messages* в классе *MySqlMessageServiceImpl* были реализованы следующие методы:

– метод *add* – используется для сохранения отправленного сообщения в базе данных;

– метод *delete* – используется для удаления сообщения из базы данных;

– метод *getMessages* – используется для просмотра сообщений конкретного пользователя.

Для доступа к таблице *Password* в классе *MySqlPasswordServiceImpl* были реализованы следуюище методы:

– метод *add* – используется для сохранения пароля в базе данных;

– метод *delete* – используется для удаления пароля из базы данных;

– метод *getPassword*– используется для получения пароля из базы данных.

Для доступа к таблице *Users* в классе *MySqlUsersServiceImpl* были реализованы следуюище методы:

– метод *add* – используется для сохранения пользователя в базе данных;

– метод *delete* – используется для удаления пользователя из базы данных;

– метод *getUsers*– используется для получения всех пользователей из базы данных.

Диаграмма разработанных классов для доступа к данным представлена на рисунке 3.2:

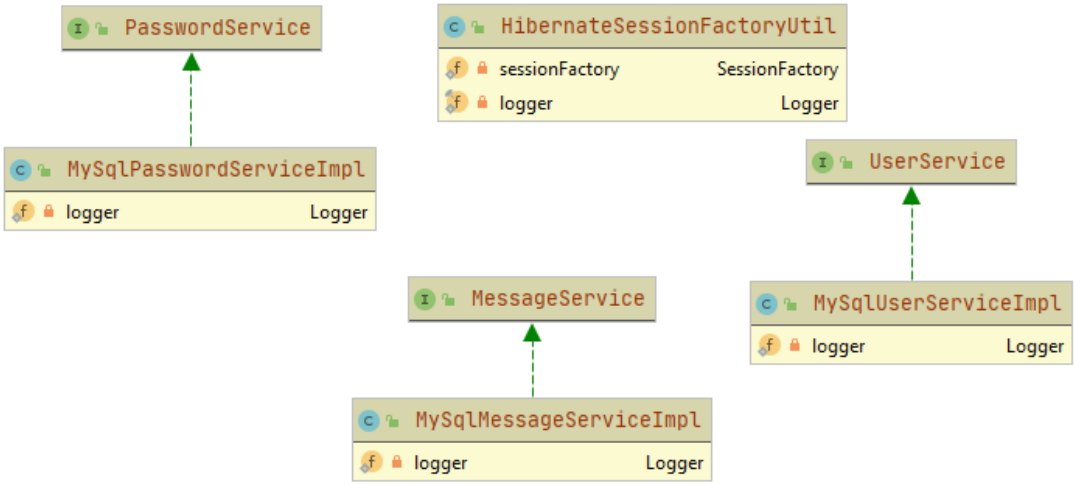

Рисунок 3.2 – Диаграмма классов для доступа к данным

Для реализации сетевого соединения в языке *Java* были использованы *WebSockets*. Их особенность заключается в том, что после отправки *HTTP*-ответа клиенту установленное соединение не закрывается, а остается открытым, что позволяет передавать и получать сообщения в режиме реального времени.

Листинг разработанных классов представлен в приложении Б.

4 РЕЗУЛЬТАТЫ ВЕРИФИКАЦИИ И ЭКСПЛУАТАЦИИ ПРОГРАММНОГО КОМПЛЕКСА

4.1 Результаты тестирования программного комплекса

Для проверки корректной работы сетевого приложения реализованы тесты для тестирования слоя доступа к данным.

Для тестирования созданных классов используются модульные тесты. Модульный тест представляет собой специальный класс, содержащий методы, в которых описана логика проведения тестов над классами и их методами. Для сравнения ожидаемых и фактических результатов используется класс *Assertions* и его статические методы.

Для проведения тестирования были реализованы следующие классы:

– *MySqlMessageServiceImplTest* – создан для тестирования класса *MySqlMessageServiceImpl;*

– *MySqlPasswordServiceImplTest* – создан для тестирования класса *MySqlPasswordServiceImpl;*

– *MySqlUserServiceImplTest* – создан для тестирования класса *MySqlUserServiceImpl;*

Тесты проводятся в том порядке, в котором они перечислены выше.

Тестированию подвергаются все публичные методы. Тестирование проводится на корректных и некорректных входных данных.

В классе *MySqlMessageServiceImpl* реализованы следующие методы:

– *add* – тестирует метод добавления в базу данных отправленного сообщения;

– *delete* – тестирует метод удаления из базы данных сообщения;

– *getMessages* – тестирует метод получения из базы данных сообщений конкретного пользователя;

Аналогичные методы были реализованы и в других классах тестов. Результаты проведения тестирования приведены на рисунке 4.1

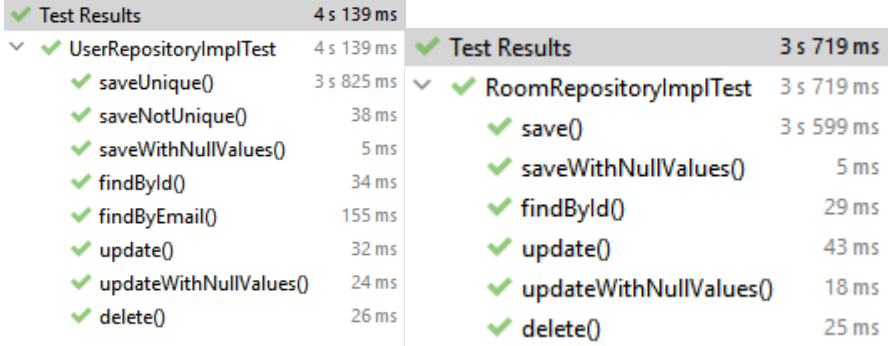

Рисунок 4.1 – Результаты выполнения тестов

В результате проведения тестирования все тесты были пройдены успешно.

4.2 Результаты эксплуатации программного комплекса

Для начала эксплуатации клиентского приложения необходимо запустить серверное приложение. Для запуска серверного приложения используется *Apache Tomcat.*

После запуска сетевого приложения в веб-браузере открывается клиентское приложение, которое представлено на рисунке 4.2

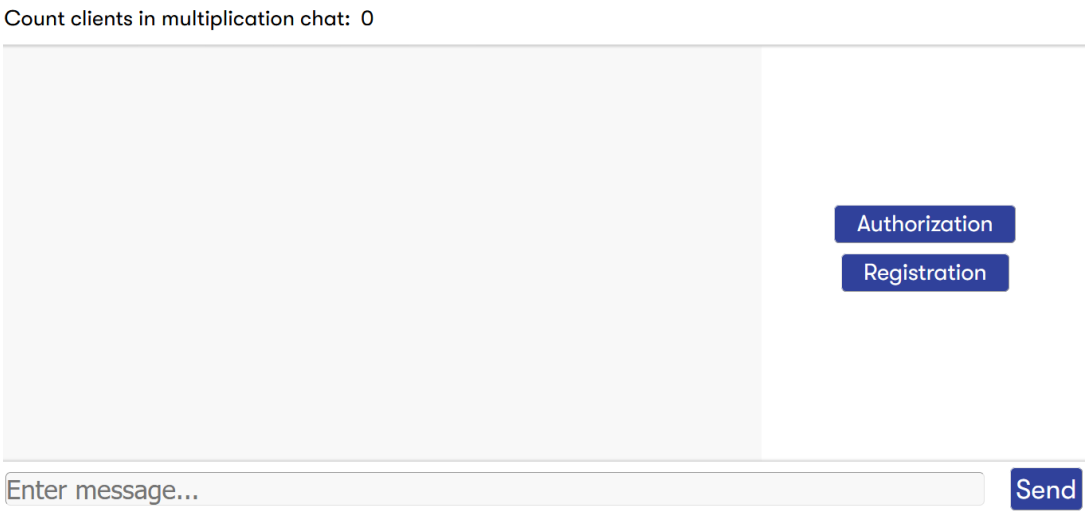

Рисунок 4.2 – Клиентское приложение

Для авторизации в приложении необходимо нажать на кнопку «Authorization» и ввести нужные данные. На рисунке 4.3 показан результат нажатия кнопки «Authorization».

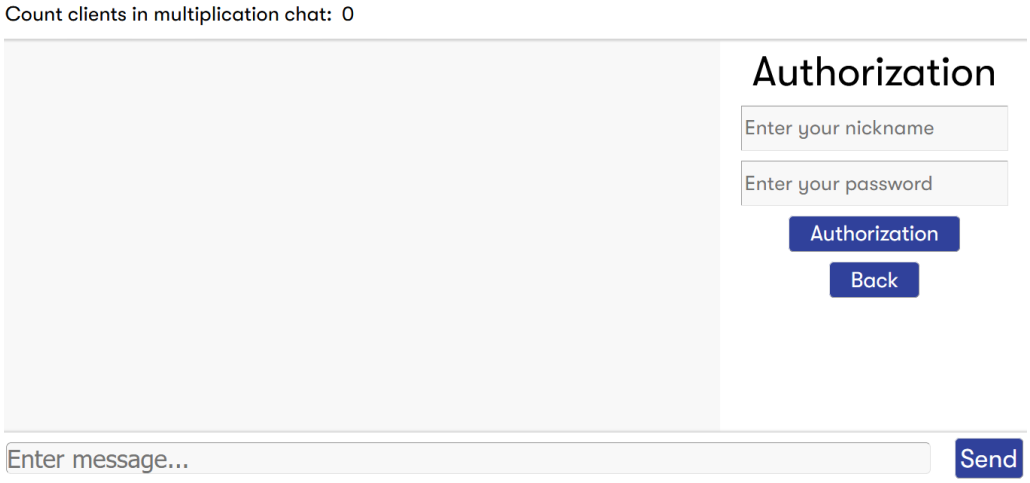

Рисунок 4.3 – Результат нажатия на кнопку «Authorization»

Для авторизации пользователя в системе необходимо заполнить поля и нажать на кнопку «Authorization». Если данные были введены корректно, то пользователь успешно авторизовывается в системе, получает все написанные сообщения в чате и также может отправлять свои сообщения в чат. Пример авторизованного пользователя и отправки сообщений приведен на рисунке 4.4

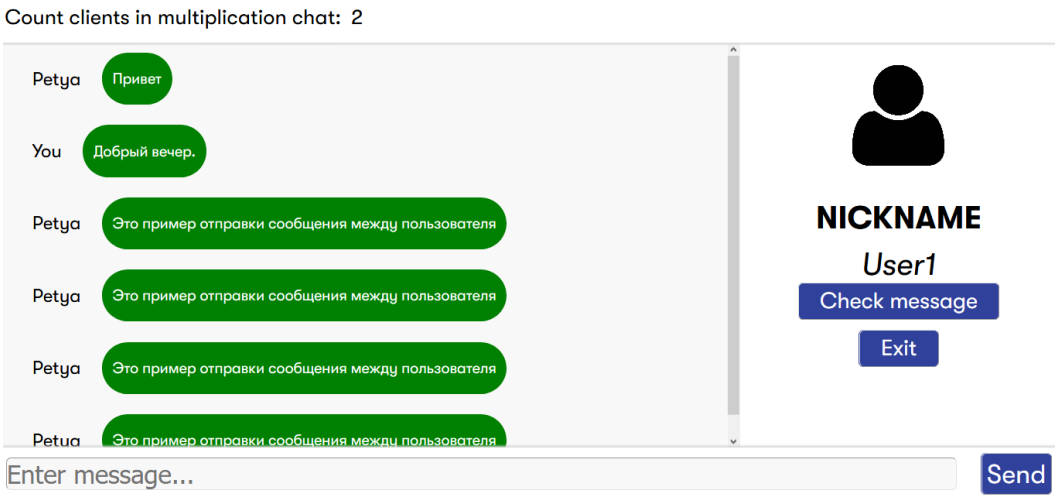

Рисунок 4.4 – Результат аутентификации пользователя и отправки сообщений

Для перехода в окно регистрации необходимо нажать на строку «Registration». В результате появится форма, представленная на рисунке 4.5.

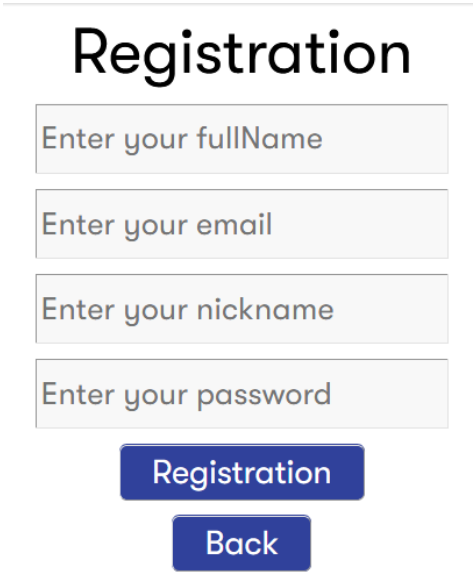

Рисунок 4.5 – Форма регистрации

Для успешной регистрации необходимо корректно заполнить все поля. Если поля заполнены некорректно, то при нажатии на кнопку «Зарегистрироваться» появится предупреждающее сообщение.

Если при регистрации был введён адрес электронной почты, на который уже зарегистрирован пользователь, то будет выведено соответствующее предупреждение и регистрация не будет произведена.

При успешной регистрации появится соответствующее сообщение и форма входа, в которую необходимо ввести данные для авторизации. При вводе некорректных данных будет выведено соответствующее предупреждение. При вводе данных о пользователе, которого не существует, также будет выведено соответствующее сообщение.

При успешном входе в аккаунт окно приложения будет выглядеть, как показано на рисунке 4.4

В левой части окна приложения содержится список чатов. В правой части – блок с сообщениями чата и строка для ввода и отправки сообщения. Для просмотра сообщений необходимо нажать на кнопку «*Check Message*». При выборе в блоке сообщений появится вся история сообщений. Если по каким-то причинам не удалось получить либо данные о чатах, либо данные о истории сообщений чата, то будет выведено соответствующее сообщение.

Для отправки сообщения необходимо ввести его в поле ввода и нажать на кнопку «Отправить». В случае удачной отправки сообщения оно отобразится в блоке сообщений. Результат представлен на рисунке 4.7.

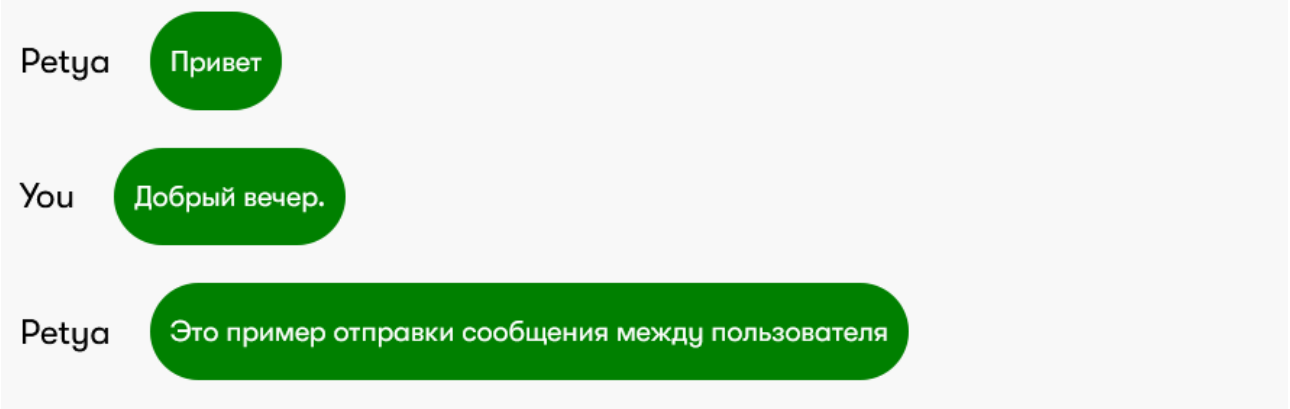

Рисунок 4.11 – Результат отправки сообщения

Для отключения от чата необходимо нажать на кнопку «Exit», которая расположена в правой части клиентского приложения.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта изучены основные технологии для реализации сетевого взаимодействия между приложениями.

Разработан программный комплекс, который состоит из серверного и клиентского приложения. Разработанный программный комплекс позволяет общаться с пользователями с помощью текстовых сообщений.

Серверное и клиентское приложения разработаны с помощью языка *Java* и с применением протокола *HTTPP* и технологии *WebSocket*. Использование языка *Java* обеспечивает кроссплатформенность реализованных приложений.

Приложение имеет систему регистрации и авторизации, что позволяет отображать определённую информацию в зависимости от ролевой политики пользователя. Данный функционал позволяет сохранять целостность данных, а также ограничить определённые группы пользователей от важной и недоступной для них информации.

Серверное приложение поддерживает одновременную обработку запросов от нескольких клиентов, что позволяет отображать чаты и обмениваться сообщениями с несколькими пользователями в реальном времени.

Выбранная архитектура приложений позволяет беспроблемно расширять их функционал, например, обмениваться не только текстовыми, но и графическими и голосовыми сообщениями.

На основании проведённой верификации и модульном тестировании можно утверждать о полной работоспособности всех модулей разработанного программного комплекса.

Список используемых источников

1. Практическое руководство к курсовому проектированию по курсу «Информатика» для студентов технических специальностей дневной и заочной форм обучения – Гомель: ГГТУ им. П.О. Сухого, 2019. – 32 с.

2. *Technopedia*: Свободная энциклопедия. – Электрон. данные. – Режим доступа:https://www.techopedia.com/definition/438/clientserverarchitecture. – Дата доступа: 19.12.2020.

3. *Britannica*: Свободная энциклопедия. – Электрон. данные. – Режим доступа: https://www.britannica.com/technology/client-server-architecture. – Дата доступа: 19.12.2020.

4. *Lia*: Свободная энциклопедия. – Электрон. данные. – Режим доступа: http://lia.deis.unibo.it/Courses/PMA4DS1112/materiale/9.client-server. – Дата доступа: 19.12.2020.

5. *IBM Knowledge Center*: Свободная энциклопедия. – Электрон. данные. – Режим доступа: https://www.ibm.com/support/knowledgecenter. – Дата доступа: 19.12.2020.

6. *Client/Server Architecture*: Свободная энциклопедия. – Электрон. данные. – Режим доступа: https://clientserverarch.blogspot.com/2013/03/advantages-and-disadvantages-of-client.html. – Дата доступа: 19.12.2020.

7. Олифер, В. Компьютерные сети. Принципы, технологии, протоколы: учебное пособие / В. Олифер, Н. Олифер. – СПб. : Питер, 2016. – 842 с.

8. Таненбаум, Э. Компьютерные сети / Э. Таненбаум. – СПб. : Питер, 2019. – 960 с.

ПРИЛОЖЕНИЕ А
(обязательное)

Функциональная схема приложения

ПРИЛОЖЕНИЕ Б
(обязательное)

Код программы

Message.java

```
package by.gstu.beans;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Objects;

@Entity
@Table(name = "messages")
public class Message implements Serializable {
   @Id
   @GeneratedValue
   @Column(name = "Id")
   private int id;

   @Column(name = "Message")
   private String message;

   @ManyToOne(optional = false, cascade = CascadeType.ALL)
   @JoinColumn(name = "UserId")
   private User user;

   public Message(){}
   public Message(int id, String message, User user){
      this.id = id;
      this.message = Objects.requireNonNull(message);
      this.user = user;
   }

   public void setId(int id){
      this.id = id;
   }
   public void setMessage(String message){
      this.message = Objects.requireNonNull(message);
   }
   public void setUser(User user){
      this.user = user;
   }

   public User getUser(){return user;}
   public int getId(){return id;}
   public String getMessage(){return message;}

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof Message)) return false;
      Message message1 = (Message) o;
      return getId() == message1.getId() &&
            Objects.equals(getMessage(), message1.getMessage()) &&
            Objects.equals(getUser(), message1.getUser());
   }

   @Override
```

```
    public int hashCode() {
        return Objects.hash(getId(), getMessage(), getUser());
    }

    @Override
    public String toString() {
        return "{id: " + id + ", message: " + message + "}";
    }
}
```

Password.java

```
package by.gstu.beans;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Objects;

@Entity
@Table(name = "passwords")
public class Password implements Serializable{
    @Id
    @GeneratedValue
    @Column(name = "Id")
    private int id;

    @Column(name = "password")
    private String password;

    @OneToOne(mappedBy = "password")
    private User user;

    public Password(){}
    public Password(int id, String password, User user){
        this.id = id;
        this.password = Objects.requireNonNull(password);
        this.user = user;
    }
    public Password(String password){
        this.password = Objects.requireNonNull(password);
    }

    public void setId(int id){
        this.id = id;
    }
    public void setPassword(String password){
        this.password = Objects.requireNonNull(password);
    }
    public void setUser(User user){
        this.user = user;
    }

    public int getId() {return id;}
    public String getPassword(){return password;}
    public User getUser(){return user;}
}
```

Persion.java

```
package by.gstu.beans;

import javax.persistence.*;
```

```
import java.io.Serializable;
import java.util.Objects;

@Entity
@Table(name = "users")
public abstract class Person implements Serializable {
   @Id
   @Column(name = "Id")
   @GeneratedValue
   private int id;
   private String fullName;

   public Person(){}
   public Person(int id, String fullName){
      this.id = id;
      this.fullName = Objects.requireNonNull(fullName);
   }

   public void setId(int id){this.id = id;}
   public void setFullName(String fullName){
      this.fullName = Objects.requireNonNull(fullName);
   }

   public int getId(){return id;}
   public String getFullName(){return fullName;}

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof Person)) return false;
      Person person = (Person) o;
      return getId() == person.getId() &&
            Objects.equals(getFullName(), person.getFullName());
   }

   @Override
   public int hashCode() {
      return Objects.hash(getId(), getFullName());
   }

   @Override
   public String toString() {
      return "{id: " +id+ ", fullName: " +fullName+"";
   }
}
```

User.java

```
package by.gstu.beans;

import by.gstu.beans.interfaces.Role;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Collection;
import java.util.Objects;
import java.util.Set;

@Entity
@Table(name = "users")
public class User extends Person implements Serializable{
   @Column(name = "Email")
```

```
    private String email;

    @Column(name = "Nickname")
    private String nickname;

    @Column(name = "Role")
    private Role role;

    @OneToMany(mappedBy = "user", fetch = FetchType.EAGER)
    private Collection<Message> messages;

    @OneToOne(optional = false, cascade = CascadeType.ALL)
    @JoinColumn(name = "PasswordId")
    private Password password;

    public User(){}
    public User(int id, String fullName, String email, String nickname, Role role){
        super(id, fullName);
        this.email = email;
        this.nickname = nickname;
        this.role = Objects.requireNonNull(role);
    }

    public void setEmail(String email){
        this.email = Objects.requireNonNull(email);
    }
    public void setNickname(String nickname){
        this.nickname = Objects.requireNonNull(nickname);
    }
    public void setRole(Role role){
        this.role = Objects.requireNonNull(role);
    }
    public void setMessages(Set<Message> messages){
        this.messages = messages;
    }

    public String getEmail(){return email;}
    public String getNickname(){return nickname;}
    public Role getRole(){return role;}
    public Collection<Message> getMessages(){return messages;}

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof User)) return false;
        if (!super.equals(o)) return false;
        User user = (User) o;
        return Objects.equals(getEmail(), user.getEmail()) &&
                Objects.equals(getNickname(), user.getNickname()) &&
                getRole() == user.getRole();
    }

    @Override
    public int hashCode() {
        return Objects.hash(super.hashCode(), getEmail(), getNickname(), getRole());
    }

    @Override
    public String toString() {
        return super.toString() + ", email: " +email+ ", nickname: " +nickname+ ", role: " +role+ "}";
    }
}
```

HibernateSessionFactoryUtil.java

```
package by.gstu.beans.dao.sessionFactory;

import by.gstu.beans.Message;
import by.gstu.beans.Password;
import by.gstu.beans.User;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class HibernateSessionFactoryUtil {
    private static SessionFactory sessionFactory;

    private static final Logger logger = LogManager.getLogger();

    private HibernateSessionFactoryUtil(){}

    public static SessionFactory getSessionFactory(){
        if (sessionFactory == null){
            try{
                Configuration configuration = new Configuration().configure();
                configuration.addAnnotatedClass(Message.class);
                configuration.addAnnotatedClass(Password.class);
                configuration.addAnnotatedClass(User.class);

                StandardServiceRegistryBuilder builder = new StandardServiceRegistryBuilder().applySettings(configura-
tion.getProperties());
                sessionFactory = configuration.buildSessionFactory();
            }catch (Exception e){
                logger.error(e.getMessage());
            }
        }
        return sessionFactory;
    }
}
```

MessageService.java

```
package by.gstu.beans.dao.interfaces;

import by.gstu.beans.Message;
import by.gstu.beans.User;

import java.util.List;

public interface MessageService {
    boolean add(Message message);
    boolean delete(Message mesage);
    boolean delete(int id);
    boolean update(Message message);
    List<Message> getMessages(User user);
    List<Message> getMessage(int userId);
}
```

PasswordService.java

```
package by.gstu.beans.dao.interfaces;

import by.gstu.beans.Message;
```

```
import by.gstu.beans.Password;
import by.gstu.beans.User;

import java.util.List;

public interface PasswordService {
    boolean add(Password password);
    boolean delete(Password password);
    boolean delete(int id);
    boolean update(Password password);
    Password getPassword(User user);
    Password getPassword(int userId);
}
```

UserService.java

```
package by.gstu.beans.dao.interfaces;

import by.gstu.beans.User;

import java.util.List;

public interface UserService {
    boolean add(User user);
    boolean delete(User user);
    boolean delete(int id);
    boolean update(User user);
    User getUser(int userId);
    List<User> getUsers();
}
```

MySqlMEssageServiceImpl.java

```
package by.gstu.beans.dao.mysql;

import by.gstu.beans.Message;
import by.gstu.beans.User;
import by.gstu.beans.dao.interfaces.MessageService;
import by.gstu.beans.dao.sessionFactory.HibernateSessionFactoryUtil;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.hibernate.Session;
import java.util.List;
import java.util.stream.Collectors;

public class MySqlMessageServiceImpl implements MessageService {

    private static Logger logger = LogManager.getLogger();

    @Override
    public boolean add(Message message) {
        try (Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            session.save(message);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
            return false;
        }
    }
```

```
@Override
public boolean delete(Message mesage) {
    try (Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
        session.beginTransaction();
        session.delete(mesage);
        session.getTransaction().commit();
        return true;
    }
    catch (Exception ex){
        logger.error(ex.getMessage());
        return false;
    }
}

@Override
public boolean delete(int id) {
    try (Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
        session.beginTransaction();
        Message message = session.get(Message.class, id);
        session.delete(message);
        session.getTransaction().commit();
        return true;
    }
    catch (Exception ex){
        logger.error(ex.getMessage());
        return false;
    }
}

@Override
public boolean update(Message message) {
    try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
        session.beginTransaction();
        session.update(message);
        session.getTransaction().commit();
        return true;
    }
    catch (Exception ex){
        logger.error(ex.getMessage());
        return false;
    }
}

@Override
public List<Message> getMessages(User user) {
    try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
        return user.getMessages()
                .stream()
                .filter(o -> o.getId() == user.getId())
                .collect(Collectors.toList());

    }
    catch (Exception ex){
        logger.error(ex.getMessage());
    }
    return null;
}

@Override
public List<Message> getMessage(int userId) {
    try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
```

```
            return session.createQuery("from Message", Message.class)
                    .list()
                    .stream()
                    .filter(o -> o.getId() == userId)
                    .collect(Collectors.toList());

        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return null;
    }
}
```

MySqlPasswordServiceImpl.java

```
package by.gstu.beans.dao.mysql;

import by.gstu.beans.Message;
import by.gstu.beans.Password;
import by.gstu.beans.User;
import by.gstu.beans.dao.interfaces.PasswordService;
import by.gstu.beans.dao.sessionFactory.HibernateSessionFactoryUtil;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.hibernate.Session;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class MySqlPasswordServiceImpl implements PasswordService {

    private static Logger logger = LogManager.getLogger();

    @Override
    public boolean add(Password password) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            session.save(password);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }

    @Override
    public boolean delete(Password password) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            session.delete(password);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }
```

```
    @Override
    public boolean delete(int id) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            Password findMsg = session.createQuery("from Password ", Password.class)
                    .stream()
                    .filter(o -> o.getId() == id)
                    .findAny().orElse(new Password());
            session.delete(findMsg);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }

    @Override
    public boolean update(Password password) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            session.update(password);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }

    @Override
    public Password getPassword(User user) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            return session.createQuery("from Password ", Password.class)
                    .stream()
                    .filter(o -> o.getId() == user.getId())
                    .findAny().orElse(new Password());
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return null;
    }

    @Override
    public Password getPassword(int userId) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            return session.createQuery("from Password ", Password.class)
                    .stream()
                    .filter(o -> o.getId() == userId)
                    .findAny().orElse(new Password());
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return null;
    }
}
```

MySqlUserServiceImpl.java

```
package by.gstu.beans.dao.mysql;

import by.gstu.beans.User;
import by.gstu.beans.dao.interfaces.UserService;
import by.gstu.beans.dao.sessionFactory.HibernateSessionFactoryUtil;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.hibernate.Session;

import java.util.List;
import java.util.stream.Collectors;

public class MySqlUserServiceImpl implements UserService {

    private static Logger logger = LogManager.getLogger();

    @Override
    public boolean add(User user) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            session.save(user);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }

    @Override
    public boolean delete(User user) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            session.delete(user);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }

    @Override
    public boolean delete(int id) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            User findUser = session.createQuery("from User ", User.class)
                    .stream()
                    .filter(o -> o.getId() == id)
                    .findAny().orElse(new User());
            session.delete(findUser);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }
```

```
    @Override
    public boolean update(User user) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            session.beginTransaction();
            session.update(user);
            session.getTransaction().commit();
            return true;
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return false;
    }

    @Override
    public User getUser(int userId) {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            return session.createQuery("from User ", User.class)
                    .stream()
                    .filter(o -> o.getId() == userId)
                    .findAny().orElse(new User());
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return null;
    }

    @Override
    public List<User> getUsers() {
        try(Session session = HibernateSessionFactoryUtil.getSessionFactory().openSession()){
            return session.createQuery("from User ", User.class)
                    .stream()
                    .collect(Collectors.toList());
        }
        catch (Exception ex){
            logger.error(ex.getMessage());
        }
        return null;
    }
}
```

WebSocketServer.java

```
package by.gstu.itp.server.api;

import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import java.util.Vector;

@ServerEndpoint(value = "/websocket")
public class WebSocketServer {

    private static final Set<Session> clients = new HashSet<>();
    private static int nextId;

    public static void sendingMessagesToClients(String message) {
        synchronized (clients) {
```

```
        for (Session client : clients) {
            try {
                client.getBasicRemote().sendText(message);
            } catch (IOException e) {
                System.out.println("client error");
                clients.remove(client);
            }
        }
    }
}

@OnMessage
public void onMessage(String message, Session session) {
    String[] commandContent = message.split(":");
    try {
        switch (commandContent[0]) {
            case "check":
                String password = commandContent[1].trim();
                boolean resCheckPaass = new Parser(password).checkPassword();
                new Thread(() -> sendingMessagesToClients("" + resCheckPaass)).start();
                break;
            case "quit":
                synchronized (clients) {
                    clients.remove(session);
                }
                break;
            default:
                session.getBasicRemote().sendText("undefined command");
        }
    } catch (IOException e) {
        System.out.println("client error");
        synchronized (clients) {
            clients.remove(session);
        }
    } catch (Exception e) {
        System.out.println("msg error");
        System.out.println(e.toString());
    }
}

@OnOpen
public void onOpen(Session session) {
    session.getUserProperties().put("username", "user_" + nextId++);
    String username = session.getUserProperties().get("username").toString();
    System.out.println(username + " connected to server");
    sendingMessagesToClients(username + " connected to server");

    synchronized (clients) {
        clients.add(session);
    }
}

@OnClose
public void onClose(Session session) {
    synchronized (session) {
        clients.remove(session);
    }

    System.out.println("client disconnected: " + session.getUserProperties().get("username"));
}

@OnError
```

```
    public void onError(Session session, Throwable throwable ) {
        System.out.println("Error socket, maybe someone disconnected.");
    }
}
```

index.js

```
import './css/layout.css'
import './css/header.css'
import './css/main.css'
import './css/footer.css'
import UserLogo from '../public/img/logo2.jpg'

/**
 * Roles users
 * @type {{}}
 */
const roles = {
    'admin': 1,
    'user': 0
}
const defaultURL = 'ws://localhost:3443/websocket'

/**
 * Displays the desired block
 * @param visibleBlock{String} block to be displayed
 * @param otherBlocks{Array}
 */
function exposeBlock(visibleBlock, otherBlocks) {
    document.querySelector(visibleBlock).style.display = 'flex'
    otherBlocks.forEach(classNameBlock => {
        document.querySelector(classNameBlock).style.display = 'none'
    })
}
/**
 * Authorization user
 */
function authorization() {
    exposeBlock('.user-info', ['.main-window', '.authorization', '.registration'])
    const data = {
        nickname: document.getElementById('input-ncknm').value.toString(),
        password: document.getElementById('input-pswrd').value.toString(),
    }
    fetch(defaultURL, {
        method: 'POST',
        body: data
    })
        .then(response => response.json())
        .then(response => {
            if (response.check === true){
                console.log(`User with nickname ${data.nickname} successfully authorization`)
                console.log(`You can start communication`)
                exposeBlock('.user-info', ['.main-window', '.authorization', '.registration'])
                document.querySelectorAll('.nickname').value = data.nickname

                if (response.role === roles['admin']){
                    document.querySelector('.btn-check-your-msg').style.display = 'flex'
                    document.querySelector('.check-user-messages').style.display = 'flex'
                }
                else{
                    document.querySelector('.btn-check-your-msg').style.display = 'flex'
                    document.querySelector('.check-user-messages').style.display = 'none'
```

```
            }
        }
        else{
            console.log(`Error authorization, incorrect nickname or password`)
        }
    })
}
/**
 * Function for registration user
 */
function registration() {
    const data = {
        fullName: document.getElementById('input-fullName-r').value.toString(),
        email: document.getElementById('input-email-r').value.toString(),
        nickname: document.getElementById('input-ncknm-r').value.toString(),
        password: document.getElementById('input-pswrd-r').value.toString()
    }
    fetch(defaultURL, {
        method: 'POST',
        body: data
    })
        .then(response => response.json())
        .then(response => {
            if (response.check === true){
                console.log(`User with nickname ${data.nickname} successfully authorization`)
                console.log(`You can start communication`)
                exposeBlock('.user-info', ['.main-window', '.authorization', '.registration'])
                document.querySelectorAll('.nickname').value = data.nickname

                if (response.role === roles['admin']){
                    document.querySelector('.btn-check-your-msg').style.display = 'flex'
                    document.querySelector('.check-user-messages').style.display = 'flex'
                } else{
                    document.querySelector('.btn-check-your-msg').style.display = 'flex'
                    document.querySelector('.check-user-messages').style.display = 'none'
                }
            } else{
                console.log(`Error authorization, incorrect nickname or password`)
            }
        })
}

function exit() {
    fetch(defaultURL)
        .then(response => console.log('Connection with server successfully closed'))
    exposeBlock('.main-window', ['.registration','.authorization','.user-info'])
}

const socket = new WebSocket(defaultURL)

socket.onopen = () => {
    console.log("Connected with server successfully")
}
socket.onmessage = (event) => {
    const data = JSON.stringify(event.data)
    const message = data.message
    const nickname = data.nickname
    document.querySelector('.show-msg').appendChild(createMessage(message, nickname))
}
socket.onerror = () => {
    console.error(`Error`)
}
```

```
socket.onclose = () => {
    console.log('Connection with server is closed')
}
/**
 * Send message to server
 */
function sendMessage(){
    const message = document.getElementById('message').value.toString()
    socket.send(message)
}
```

Util.js

```
/**
 * Create content in the DOM-element
 * @type {{string: (function(*=): Text), function: (function(*=): HTMLElementTagNameMap[K]), object: (function(*): *)}}
 */
const elementSelector = {
    'string' : child => document.createTextNode(child),
    'function' : child => document.createElement(child),
    'object' : child => child
}
/**
 *
 * @param message{string} user
 * @param nickname{string} user
 * @returns {object}
 */
function createMessage(message, nickname) {
    return node({
        type: 'div',
        classList: ['message'],
        children: [
            node({
                type: 'div',
                classList: ['label-msg'],
                children: nickname
            }),
            node({
                type: 'div',
                classList: ['content-msg'],
                children: message
            })
        ]
    })
    //console.log(root_node)
}
/**
 * Return node DOM
 * @param option{object}
 */
function node(option) {
    const type = option.type? option.type : 'div'
    const element = document.createElement(type)

    if (option.children){
        if (option.children instanceof Array){
            option.children.forEach(child => element.appendChild(elementSelector[typeof child](child)))
        }
        else{
            element.appendChild(elementSelector[typeof option.children](option.children))
```

```
        }
    }

    if (option.href){
        element.href = option.href
    }
    if (option.value){
        element.value = option.value
    }
    if (option.selected){
        element.selected = option.selected
    }
    if (option.inputType){
        element.type = option.inputType
    }
    element.onclick = option.onclick ? (e) => option.onclick(e) : element.onclick;
    element.onload = option.onload ? option.onload : element.onload;

    if (typeof element.onload === "function") {
        element.onload();
    }

    if (option.classList) {
        option.classList.forEach(c => element.classList.add(c));
    }

    if (option.id) {
        element.id = option.id;
    }

    if (option.name) {
        element.name = option.name;
    }

    if (option.for) {
        element.htmlFor = option.for;
    }

    if (option.placeholder) {
        element.placeholder = option.placeholder;
    }

    if (option.disabled) {
        element.disabled = option.disabled;
    }

    if (option.styles) {
        const optionStyles = option.styles;
        const styles = element.style;
        if (optionStyles.background) {
            styles.background = optionStyles.background;
        }
        if (optionStyles.color) {
            styles.color = optionStyles.color;
        }
    }

    return element;
}
```