Содержание

**Введение**

В современном мире сетевые мессенджеры стали неотъемлемой частью повседневной жизни, обеспечивая мгновенную связь между людьми независимо от их географического положения. Развитие технологий и широкое распространение интернета способствовали появлению множества приложений для обмена сообщениями, таких как WhatsApp, Telegram, Viber и другие. Однако, несмотря на обилие существующих решений, разработка собственного мессенджера остается актуальной задачей, позволяющей глубже понять принципы работы сетевых технологий и протоколов обмена данными.

Целью данной курсовой работы является разработка простейшего сетевого мессенджера, который будет обеспечивать базовые функции обмена сообщениями между пользователями.

Для достижения поставленной цели необходимо решить следующие задачи:

1) Изучить современные технологии и протоколы, используемые в разработке мессенджеров.

2) Спроектировать архитектуру простейшего мессенджера, включающую серверную и клиентскую части.

3) Реализовать серверную часть мессенджера, обеспечивающую прием и передачу сообщений между клиентами.

4) Разработать клиентское приложение, позволяющее пользователям обмениваться сообщениями в реальном времени.

5) Провести тестирование и отладку разработанного мессенджера для обеспечения его корректной работы.

Методы и средства решения поставленных задач включают использование языка программирования Python, а также библиотек и фреймворков, таких как Flask для серверной части и Tkinter для клиентской. Для обмена данными между клиентами и сервером будет использоваться протокол HTTP и WebSocket обеспечивающий двустороннюю связь в реальном времени.

Актуальность данной работы обусловлена потребностью в создании новых и улучшении существующих инструментов для обмена сообщениями, что способствует развитию коммуникационных технологий и повышению удобства взаимодействия между пользователями. Новизна работы заключается в создании мессенджера, который не только демонстрирует базовые принципы работы подобных приложений, но и предоставляет возможность для дальнейшего расширения и модификации.

Таким образом, данная курсовая работа направлена на разработку простейшего сетевого мессенджера, который будет служить эффективным инструментом для обмена сообщениями и позволит углубить понимание принципов работы сетевых приложений.

**1 Обоснование выбора методов реализации**

Я выбрал Python в качестве основного языка программирования для разработки данного программного продукта по нескольким причинам. Во-первых, у меня есть значительный опыт работы с Python, что позволяет мне эффективно и быстро решать задачи, возникающие в процессе разработки. Python известен своей простотой и читаемостью кода, что ускоряет процесс написания и отладки программ. Кроме того, богатая экосистема библиотек и фреймворков Python, таких как Tkinter и другие, предоставляет мощные инструменты для создания функциональных и удобных пользовательских интерфейсов.

Выбор формата настольного приложения был обусловлен рядом факторов, среди которых ключевым является наличие значительного опыта в разработке именно настольных приложений, что существенно превосходит мой опыт в создании web-приложений. Настольные приложения предоставляют более глубокий контроль над системными ресурсами и позволяют реализовать более сложные и производительные функции. Этот формат также обеспечивает более тесное взаимодействие с пользователем, что способствует улучшению пользовательского опыта.

Для реализации серверной части моего приложения я остановили свой выбор на библиотеке Flask. Этот выбор был сделан по нескольким причинам. Во-первых, Flask отличается простотой и удобством в использовании, что позволяет значительно ускорить процесс разработки и сосредоточиться на функциональности приложения. Во-вторых, Flask обладает обширным и активным сообществом разработчиков, что обеспечивает доступ к большому количеству ресурсов, таких как документация, учебные материалы и форумы, где можно найти ответы на возникающие вопросы и решения для различных проблем. Это сообщество также активно поддерживает и развивает библиотеку, что гарантирует её актуальность и надёжность.

Для создания интерфейса пользователя клиентской части приложения был выбран Tkinter, так как он изначально включен в установочный пакет Python. Tkinter достаточно удобен для создания простых графических интерфейсов. Он позволяет создавать все стандартные элементы интерфейсов, такие как надписи, кнопки, поля ввода текста и т.д. Поскольку Tkinter существует достаточно давно, в сети имеется большое количество различных обучающих материалов и возможностей решения тех или иных проблем, возникающих в ходе разработки приложения.

**2 Проектирование**

**2.1 Функциональные требования к приложению**

В качестве основных функций приложения были выбраны базовые функции простейшего мессенджера:

1. Регистрация новых пользователей в системе
2. Авторизация существующих пользователей в систему
3. Поиск пользователей другими пользователями и создание диалогов
4. Отправка и прием текстовых сообщений
5. Обновление окна диалога в реальном времени

**2.2 Структура приложения**

Так как мессенджер не может работать без базы данных нужно реализовать управление этой базой данных. Предоставлять доступ к базе данных напрямую клиенту было бы более простым решением, но такой подход является не безопасным, так как клиент может получить доступ ко всему содержимому базы или потенциально удалить все данные от туда. Решением проблемы может стать разработка серверного приложения управляющего базой данных и предоставляющего программный интерфейс (API) клиенту. Таким образом получается промежуточное звено между клиентом и базой данных ограничевающее возможные действия клиента.

**2.3 База данных**

В качестве базы данных принято решение использовать готовый условно бесплатный сервис Google Firebase который предоставляет расширяемую базу данных Firestore и API для работы с ней.

Google Firestore - это гибкая и масштабируемая NoSQL база данных, предоставляемая Google Cloud Platform. Она относится к категории документо-ориентированных баз данных и хранит данные в виде коллекций документов. Каждый документ содержит набор полей и значений, а также уникальный идентификатор. Одной из ключевых особенностей Firestore является возможность автоматической синхронизации данных между клиентскими устройствами и сервером, что обеспечивает быстрый доступ к актуальной информации. Кроме того, Firestore обладает масштабируемой инфраструктурой, позволяющей обрабатывать большие объемы данных и высокие нагрузки. Ее преимущества включают гибкую модель данных, высокую доступность и надежность.

В данном проекте база данных содержит 3 коллекции: «Users» для хранения данных пользователей, «Conversations» для хранения всех диалогов и «Messages» для хранения всех сообщений.

Каждый документ в коллекции «Users» в качестве уникального идентификатора имеет имя пользователя. Каждый документ содержит два поля:

1. «password» - содержащее хэш пароля пользователя.
2. «display\_name» - содержащее отображаемое имя пользователя. В текущем состоянии приложения оно не используется.

Пароли хранятся в виде хэшей для обеспечения безопасности. Подробнее про хэширование паролей рассказывается в пункте ???.

Документ из коллекции «Conversations» имеет уникальный идентификатор в виде строки отражающей имена участников формата:

<пользователь1>-<пользователь2>

Каждый документ из коллекции «Conversations» имеет поле «members» содержащее массив имен участников.

Документы из коллекции «Messages» имеют уникальный идентификатор в виде случайной уникальной последовательности символов, сгенерированной самим сервисом Google Firestore при создании документа. Структура документа:

1. «content» - содержание сообщения, строка.
2. «conversation\_id» - идентификатор документа диалога к которому принадлежит сообщение.
3. «sender» - идентификатор документа пользователя, оно же имя, который отправил сообщение.
4. «timestamp» - временная метка, записаная в формате API Google Firestore.

На рисунке 1 представлен снимок экрана с открытой консолью управления базой данных Google Firestore.

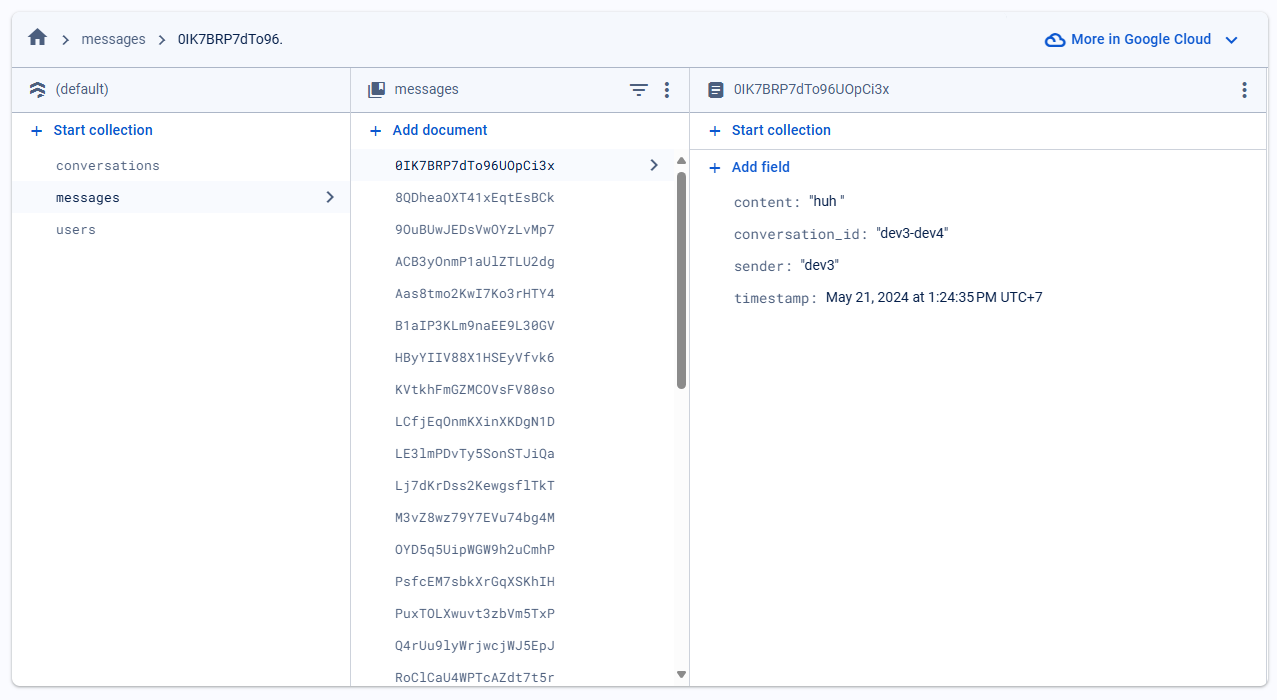


Рисунок 1 – снимок экрана консоли управления базой данных Google Firestore.

**2.2 Сервер**

Серверная часть приложения выполняет две задачи: управление базой данных и предоставление API клиентской части приложения по средствам протоколов HTTP и WebSocket.

Управление базой данных происходит с использованием библиотеки Firebase Admin SDK, оффициальной библиотеки разработаной Google для работы со всеми сервисами Firebase на языке программирования Python. В данной работе используется только база данных Firestore.

Клиент отправляет HTTP-запрос на сервер и он выполняет нужные действия для записи или поиска данных в базе. Обработка HTTP-запросов выполнена с помощью микрофреймворка Flask.

**2.2.1 Фреймворк Flask**

Flask – это микрофреймворк для создания веб-приложений на языке программирования Python. Он отличается простотой и легковесностью, что делает его идеальным выбором для разработки небольших и средних проектов. Flask имеет модульную структуру, что позволяет разработчику выбирать только необходимые компоненты и расширения для своего приложения. Одним из главных достоинств Flask является его гибкость и расширяемость, благодаря чему можно легко настраивать и расширять функциональность приложений.

Все HTTP-запросы в приложении выполнены методом POST. Обычно, в случаях, когда нужно получить данные с сервера, используется метод GET, но в этом методе все параметры запроса находятся в URL. Таким образом можно сформировать запрос даже в любом браузере и получить любые данные с сервера. Метод POST, в свою очередь, содержит всю полезную нагрузку в теле запроса, не видимую пользователю. А значит, сформировать такой запрос уже сложнее без каких-либо дополнительных инструментов.

Все функции на сервере реализованы на разных роутах. Роуты в Flask - это URL-адреса, которые определяют, какие функции должны быть вызваны при обращении к определенному URL-адресу. Роуты в Flask определяются с помощью декораторов, которые указывают на URL-адрес и метод запроса (GET, POST и т.д.), при котором должна быть вызвана соответствующая функция. Каждый роут в Flask связывает URL-адрес с определенным обработчиком, который выполняет определенные действия при обращении к указанному URL-адресу. Роуты в Flask позволяют организовать структуру приложения и определить, какие действия должны быть выполнены при запросе к определенному URL-адресу.

После выполнения определенных действий обработчик возвращает JSON объект с результатом выполнения и статус код выполнения операции.

Из основных роутов приложения:

1. /register
2. /login
3. /send
4. /find\_user
5. /create\_convo
6. /get\_chats
7. /get\_msgs

Функции login и register выполняют авторизацию и регистрацию пользователя соответственно.

Регистрация происходит следующим образом:

1. Клиент отправляет POST запрос, который содержит имя пользователя и текст пароля в полезной нагрузке запроса на соответсвующий роут.
2. Обработчик распаковывает данные из тела запроса и ищет в базе данных пользователя с таким именем и если он существует, то отправляет отрицательный результат с кодом 418.
3. Если пользователя нет, то обработчик создает документ с данными пользователя и отправляет положительный результат с кодом 201.

Хэширование пароля происходит с помощью библиотеки bcrypt.

Авторизация происходит следующим образом:

1. Клиент отправляет POST запрос, который содержит имя пользователя и текст пароля в полезной нагрузке запроса на соответсвующий роут.
2. Обработчик распаковывает данные из тела запроса и ищет в базе данных пользователя с таким именем.
3. Если пользователь существует, то обработчик сравнивает хэши паролей и отправляет положительный или отрицательный ответ в зависимости от результата.
4. Если пользователя не существует, отаправляется отрицательный ответ.

Отпрвка сообщения происходит следующим образом:

1. Обработчик распаковывает запрос, и проверяет на наличие всех необходимы данных, контента, отправителя и диалога.
2. Создает словарь из этих данных и добавляет новый документ в коллекцию «messages».
3. Далее происходит отправка сообщения на клиент получателя по средствам протокола WebSocket.

**2.2.2 Протокол WebSocket**

WebSocket - это протокол, который обеспечивает двустороннюю связь между клиентом и сервером через одно соединение TCP. Он позволяет установить постоянное соединение между клиентом и сервером, что позволяет обмениваться данными в режиме реального времени. WebSocket использует HTTP для установки соединения, после чего переключается на более эффективный протокол для обмена данными.

Работа WebSocket начинается с установки соединения через HTTP. После успешного установления соединения, сервер и клиент могут обмениваться данными, отправляя сообщения друг другу. Каждое сообщение содержит определенный тип данных, такой как текст или бинарные данные. Клиент и сервер могут отправлять сообщения в любое время, без необходимости повторного установления соединения.

В данной работе протокол WebSocket используется для отправки сообщения на клиент получателя, если тот находится в сети.

При подключении клиента на серверу, сервер заносит его имя, которое он передал в запросе на подключение, и его идентификатор сессии (SID) в словарь. Далее при работе обработчика отправки сообщений он получает имя получателя и ищет его в словаре, если оно там есть, он получет его SID и отправляет сообщение клиенту с этим SID’ом.