МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет компьютерных технологий и прикладной математики
Кафедра информационных технологий

Допустить к защите
Заведующий кафедрой
канд. физ.-мат. наук, доц.
_____В.В. Подколзин
(подпись)
_____ 2024 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

МИКРОСЕРВИСЫ ОРГАНИЗАЦИИ ОНЛАЙН-КОНФЕРЕНЦИЙ

Работу выполнил _____Р.Г. Орлов
(подпись)

Направление подготовки 01.03.02 Прикладная математика и информатика
(код, наименование)
Направленность (профиль) Программирование и информационные технологии

Научный руководитель
канд. пед. наук, доц. _____ А.В. Харченко
(подпись)
Нормоконтролер
канд. пед. наук, доц. _____ А.В. Харченко
(подпись)

Краснодар
2024

РЕФЕРАТ

Выпускная квалификационная работа 68 с., 39 рис., 12 источников.

КОНФЕРЕНЦИИ, КОМАНДЫ, МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, КЛИЕНТ, СЕРВЕР, VUE JS, JAVA, PYTHON, GOLANG, WEBRTC, WEBSOCKET.

Цель работы: разработать веб-приложение организации онлайн-конференций, серверная часть которого имеет микросервисную архитектуру, а также для командной работы и общения, с помощью языков программирования Vue JS, Java, Python, Golang, а также технологии обмена медиапотоками между браузерами WebRTC.

В процессе работы были изучены концепции построения микросервисной архитектуры, технология обмена медиапотоками WebRTC, протокол Websocket, СУБД Redis для кеширования данных, Vue JS и Golang.

В практической части было разработано веб-приложение организации онлайн-конференций, а также для командной работы и общения. Серверная часть приложения имеет микросервисную архитектуру.

Инструментарий разработки: VueJS, Java, Spring, Python, FastAPI, Django, Golang, Gin, PostgreSQL, Redis, Websocket, WebRTC, Docker.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Множество независимых модулей отвечают за обработку личных данных пользователей, совершение платежей по подписке и предоставление персонализированных рекомендаций. Выпуская цифровые сервисы, даже крупный бизнес вынужден использовать стартап-подход. Ведь разрабатывать и выводить новые продукты на рынок нужно быстро, одновременно не забывая о качестве обслуживания клиентов, — только так можно выжить в высококонкурентной среде. Кроме того, компаниям важно сразу получать дополнительные IT-ресурсы, когда нагрузки на сервисы растут, и не переплачивать за них во время спадов активности пользователей. Именно для таких задач и используются микросервисы.

На сегодняшний день существует несколько подходов к проектированию и разработке серверной части приложений. Но подход к созданию современных приложений на основе микросервисной архитектуры является наиболее популярным. Одна из компаний, которая использует микросервисную архитектуру, — Netflix. Платформа построена на более чем 1000 микросервисов. Множество независимых модулей отвечают за обработку личных данных пользователей, совершение платежей по подписке и предоставление персонализированных рекомендаций.

Целью выпускной квалификационной работы является разработка веб-приложения для организации онлайн-конференций и командной работы. Серверная часть данного приложения имеет микросервисную архитектуру, и имеет следующие компоненты: единая точка входа в приложение Gateway, а также микросервисы для работы с пользователями, командами, конференциями, чатами и сообщениями. Используется такой инструментарий, как VueJS, Java, Spring, Python, FastAPI, Django, Golang, Gin, PostgreSQL, Redis, Websocket, WebRTC, Docker. Планируется реализация следующего функционала: работа с командами, конференциями, а также организация общения в режиме реального времени.

1 Анализ проблемы

Выбор архитектуры для сервера приложения организации онлайн-конференций и командной работы представляет собой непростую задачу. Но возможность разделения функционала на части и вынесение их в отдельные компоненты кажется наиболее разумной. Такой подход позволяет осуществить микросервисная архитектура, но в то же время это влечёт за собой свои особенности.

Во-первых, это коммуникация между микросервисами. Если запрос на сервер требует обращения к нескольким компонентам, то обеспечение надёжной коммуникации играет большую роль, прежде всего, в целостности данных. Выбор необходимых технологий и протоколов, решающих данную проблему, требует особого внимания при недопустимости ошибок согласованности данных.

Далее, это управление данными. Микросервисная архитектура требует обеспечения надёжности баз данных и кэширования, а также грамотной синхронизации данных между компонентами.

Наконец, система должна быть способна работать при высоких нагрузках без задержек и сбоев. Распределение нагрузки между микросервисами позволит обеспечить отказоустойчивость сервера.

2 Понятие о микросервисной архитектуре

Микросервисы - независимо существующие и совместно работающие компоненты серверной части веб-приложения, создающиеся на основе предметной области конкретного бизнеса [1].

По мере создания программного кода для решения конкретных задач в веб-приложении разрастаются и потребности бизнеса. Ставится всё больше задач, которые должно решать веб-приложение. Вследствие этого становится всё больше программного кода. С течением времени программа становится более сложной по части нахождения тех мест в коде, где требуется внести корректировки.

Существует множество подходов для разработки микросервисной архитектуры. Один из них гласит о том, что сервисы так должны формироваться, чтобы было достаточно просто найти необходимые части кода для конкретных задач. Для этого необходимо создавать некоторые границы в веб-приложении, которые регулируются бизнес-процессами. Данный подход помогает обходить негативные последствия большого разрастания кодовой базы веб-приложения.

Отдельный микросервис со стороны остальных рассматривается как чёрный ящик, с которым можно взаимодействовать по написанным разработчиками конечным точкам (эндпоинтам) и по заранее обговорённым сводам правил и технологиям передачи данных (например, REST API или gRPC).

Разберем преимущества микросервисной архитектуры.

2.1 Технологическая разнородность

При создании системы с совместно работающими компонентами для их создания могут использоваться различные технологии, включающие в себя языки программирования, библиотеки, фреймворки и инструменты [2]. Такой

подход позволяет делать выбор в пользу конкретного набора технологий при решении задач из предметной области бизнеса. Если для решения конкретной задачи при создании какой-то из частей системы необходимо, чтобы эта часть обладала той или иной характеристикой, можно без труда подобрать нужную технологию.

2.2 Устойчивость

При выборе архитектуры серверной части веб-приложения в пользу микросервисной намного увеличивается способность бэкенда быть более устойчивым к сбоям, доступным и производительным. Если случится отказ какой-либо части системы при условии, что данный отказ не вызовет по цепочке серию сбоев других частей системы, то в этом случае есть возможность сохранить работоспособность остальных частей приложения. Существуют специальные подходы и паттерны для реализации отказоустойчивости микросервисной архитектуры. Это позволяет обеспечивать стабильность в условиях больших нагрузок на распределенные системы. Используя это, программисты могут предотвратить многие негативные последствия, среди которых появление непредвиденных ошибок и большая нагрузка на приложение. Тем самым, серверная часть веб-приложения становится более производительной. В монолитной архитектуре, в отличие от микросервисной, в случае сбоя останавливает работу всё приложение. Конечно, разрабатывая монолитную архитектуру, можно развернуть несколько экземпляров веб-приложения на нескольких компьютерах, но микросервисная архитектура при таком же развертывании способна преодолевать огромное количество сбоев, так как нагрузка в значительном количестве случаев может идти лишь на некоторый функционал веб-приложения. Это важно, поскольку в сложных распределенных системах отказов не избежать. Отказы могут быть различного характера, например задержки в сети и неисправность оборудования и так далее.

2.3 Масштабирование

При создании кодовой базы веб-приложения появляется всё больше требований, что неизбежно приводит к масштабированию [3]. Масштабирование микросервисов – способность изменять их количество в соответствии с конкретными задачами бизнес-логики, что позволяет лучше реагировать на изменение нагрузки.

Если сравнивать монолитную и микросервисную архитектуры в разрезе масштабируемости, то можно прийти к следующим выводам. Если в монолитном приложении при наличии части системы, обладающей ограниченной производительностью, наблюдается затормаживание работы остальной части приложения, то масштабировать необходимо всё приложение целиком. В микросервисной архитектуре же можно масштабировать лишь конкретные сервисы, поскольку сами сервисы являются независимыми друг от друга по своей функциональности. То есть приложение масштабируется не как единое целое, а лишь по частям.

2.4 Независимое развёртывание

Выбор в пользу микросервисной архитектуры обеспечивает возможность независимого развёртывания отдельно взятого микросервиса. Это означает, что если необходимо в данный сервис внести какие-либо корректировки, то дальнейшее его развёртывание после правок можно осуществить без необходимости развёртывания каких-либо других сервисов. Но для достижения этого необходимо, чтобы сервисы обладали слабой связанностью между собой, то есть изменение одного микросервиса должно происходить без необходимости изменения остальных микросервисов.

3 Технология WebRTC

WebRTC – технология передачи потоковых данных (таких как аудио и видео) между двумя или несколькими браузерами в режиме реального времени с открытым исходным кодом, разработанная компанией Google. Поддерживается большинством браузеров, в частности Apple Safari, Google Chrome, Microsoft Edge, Mozilla Firefox и так далее [4]. На рисунке 1 представлена структура WebRTC.

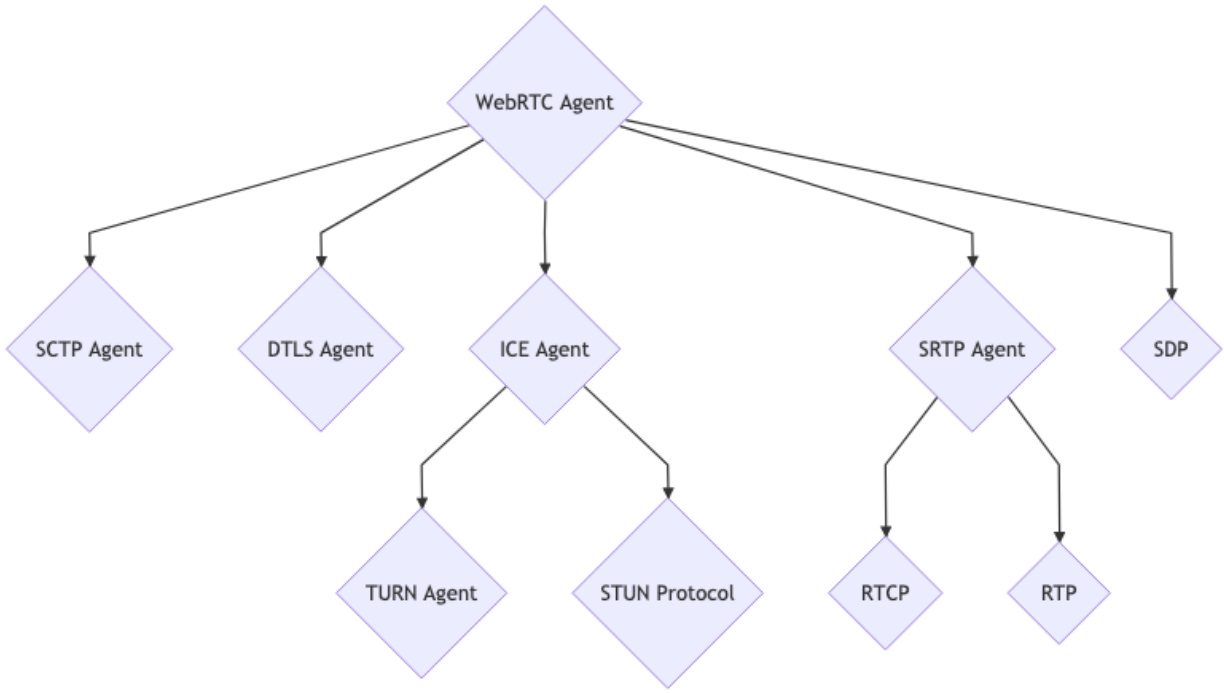

Рисунок 1 – Структура технологии WebRTC

Технология WebRTC предполагает использование peer-to-peer (P2P) соединения между браузерами, с помощью которого браузеры взаимодействуют между собой напрямую, не используя при этом каких-либо посредников (в частности, сервер). WebRTC была разработана на базе набора существующих к тому времени протоколов и кодеков, а код данной

технологии написан в библиотеке libwebrtc на базе языка программирования C++. Разберём основные части данной технологии.

При желании двух каких-либо браузеров установить между собой WebRTC-соединение, прежде всего, происходит этап сигнализации (signaling) [5]. На данной стадии целью является подготовка двух агентов WebRTC к передачи потоковых данных между ними. Для сигнализации необходим посредник между клиентами, и им может служить специальный компонент серверной части приложения, который бы по протоколу Websocket в режиме реального времени принимал запросы клиентов-браузеров на отправку так данных о сессии. Такую отправку осуществляют по протоколу SDP (Session Description Protocol), согласно которому передаётся множество информации, среди которой: данные о количестве треков, которые хочет отправить инициатор соединения; данные о кодеках, которые будут использоваться агентами WebRTC-соединения; данные, предназначенные для безопасности соединения и некоторые другие данные.

После этапа сигнализации искомые агенты WebRTC, имея при себе данные о потенциальном соединении, которые были сформированы на предыдущем шаге, наступает попытка создать искомое соединение на основе этих данных. Но для того, чтобы можно было говорить о следующем этапе, необходимо дать определение такому понятию из компьютерных сетей, как NAT.

В Интернете для идентификации компьютеров используются IP-адреса четвертой версии (IPv4). Но проблема в том, что таких IP-адресов (их еще называют внешними IP-адресами) не хватает для всех существующих в мире устройств. Поэтому необходимы способы экономии внешних IP-адресов. NAT (Network Address Translation) – технология, позволяющая преобразовывать внутренние IP-адреса (которые относятся к частным сетям и не видны в сети Интернет) к внешним. Таким образом, нескольким устройствам в одной частной сети может соответствовать один внешний IP-адрес благодаря данной технологии.

В контексте передачи потоковых данных из-за технологии NAT создание WebRTC-соединения не может начаться напрямую, поскольку по внешнему IP-адресу, как уже было объяснено выше, нельзя установить однозначно необходимое устройство. Для решения этой проблемы применяются STUN-сервера. Данные сервера необходимы для того, чтобы агенты WebRTC имели возможность узнать свой общедоступный IP-адрес и порт. Такие сервера являются публичными и доступными для всех браузеров и могут свободно использоваться при разработке веб-приложений, использующих WebRTC.

Но, как показывает практика, в некоторых случаях при получении общедоступных IP-адресов и портов могут возникать проблемы, и, как следствие, одним STUN-сервером не обойтись. Поэтому в таких случаях применяются еще один вид серверов: TURN-сервера. Они необходимы для того, чтобы через них отправлялись потоки аудио и видео искомых агентов. Но такие сервера уже не являются общедоступными.

Оба видов серверов, представленных выше, объединяет стандарт ICE (Interactive Connection Establishment). Данный стандарт необходим для того, чтобы организовать подключение между двумя агентами WebRTC. Такое подключение может быть порой установлено несколькими способами, ICE же находит наиболее эффективный из всех. Сначала идёт попытка установки прямого соединения между агентами WebRTC. Если это не удаётся (например, из-за NAT), то идёт попытка установки соединения с помощью STUN-сервера. Если же и данный шаг не удаётся осуществить, то задействуется TURN-сервер. Маршруты связи ICE-кандидатов (так называются искомые агенты WebRTC на данном этапе) записываются в текстовом формате.

После установки соединения наступает третий этап: обеспечение безопасности соединения. В отличие, например, от протокола HTTP, где по умолчанию данные передаются по незащищённому каналу связи и не формируется защищенность соединения, в технологию WebRTC по стандарту входит задача обезопасить передачу медиаданных. Причин появления каких-

либо рисков безопасности передачи аудиоданных и видеоданных как для отправителей, так и для получателей, может быть несколько. Но стандарт WebRTC даёт много возможностей для создания соответствующих приложений, при этом разработчикам не нужно беспокоиться обеспечении безопасности, так как это уже априори входит в рассматриваемую технологию.

В WebRTC защищенность соединения создаётся с помощью протоколов DTLS и SRTP. Первый из них отвечает за само шифрование данных при их передаче по двунаправленному каналу связи, а второй – за передачу этих данных по уже защищенному каналу. Стоит отметить, что для предоставления безопасности в WebRTC используется проверка совпадения отпечатка, создающегося на этапе сигнализации, с сертификатом, переданным по протоколу DTLS, в отличие от протокола HTTPS, где для той же самой задачи применяются центры доверенной сертификации.

После того, как соединение удалось сделать защищенным, наступает последний, четвертый этап: коммуникация между узлами соединения. Для данной задачи используются протоколы SRTP (для передачи медиаданных по защищенному каналу связи) и SCTP (для защищенной передачи сообщений по DataChannel – канал передачи текстовых и бинарных данных).

Таким образом, WebRTC объединяет множество существующих специальных протоколов в одно целое, что позволяет решать большое количество различных задач, связанных с передачей медиаданных.

4 Паттерны микросервисной архитектуры

Для проектирования микросервисной архитектуры используется множество паттернов. Они используются для решения проблем, которые могут возникать при создании такой архитектуры (например, проблема взаимодействия с базой данных, проблема организации взаимодействия микросервисов между собой, предоставление отказоустойчивости). Паттерны проектирования помогают разрабатывать надёжные распределенные системы, способствуют повышению производительности и безопасности. Серверные приложения, разработанные на основе микросервисной архитектуры с помощью этих паттернов, становятся более масштабируемыми и гибкими.

4.1 Паттерн Service Discovery

При развёртывании готового бэкенд-приложения обычно может создаваться по несколько экземпляров каждого микросервиса. Каждому экземпляру назначается свой IP-адрес, чаще всего – динамический. Но что, если в серверной части приложения на стадии развёртывания какой-либо из экземпляров необходимо убрать? Или, например, какой-либо из экземпляров внезапно перестал работать? Тогда пришлось бы переписывать код под работающие экземпляры, что неудобно. Вместо этого можно воспользоваться паттерном Service Discovery (рисунок 2), который мог бы автоматически пересылать запросы на доступные в данный момент экземпляры микросервисов в случае, если что-то случится с одним или несколькими узлами. Такой паттерн подразумевает введение нового компонента в микросервисную архитектуру в виде отдельной программы.

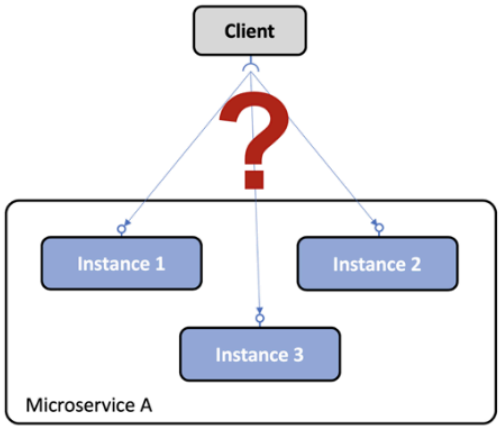

Рисунок 2 – Паттерн Service Discovery

4.2 Паттерн Gateway

В целях безопасности часто стараются скрывать динамические IP-адреса микросервисов от стороннего доступа, в том числе от клиентской части приложения. Для взаимодействия клиента и сервера применяется специальный паттерн Gateway (рисунок 3), который очередной запрос с клиента перенаправляет на другие микросервисы.

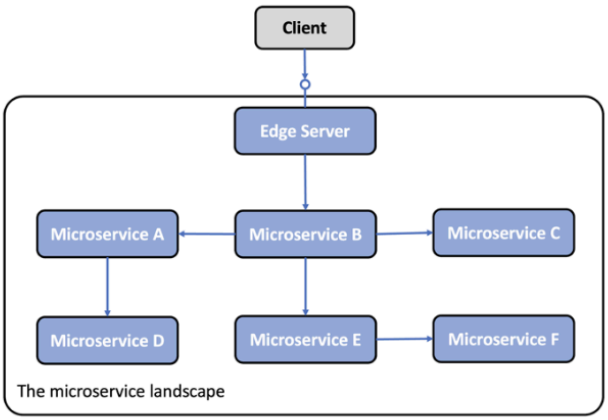

Рисунок 3 – Паттерн Gateway

Таким образом, вместо нескольких IP-адресов, связанных с серверной частью приложения, на клиенте необходимо хранить всего один IP-адрес – адрес компонента, реализующего данный паттерн, что гораздо удобнее. Этот

компонент может быть также интегрирован с компонентом, реализующим паттерн Service Discovery, описанного выше.

4.3 Паттерн Central Configuration

Бэкенд-приложение обычно развёртывается вместе с настройками конфигурации. Они могут представлять из себя IP-адреса компонентов сервера, переменные окружения операционной системы, конфигурационными файлами и так далее. Но возникает 2 вопроса. Во-первых, как можно получить полную картину конфигурации всех экземпляров для каждого из микросервисов? Во-вторых, как можно быстро в случае необходимости обновить конфигурацию в нужных местах? Для этого применяется паттерн Central Configuration (рисунок 4), хранящий в себе конфигурацию всего бэкенд-приложения.

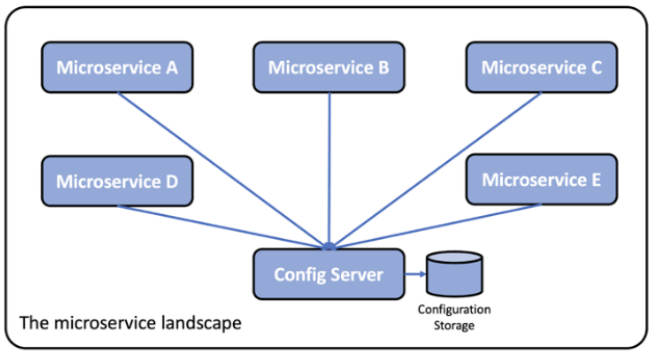

Рисунок 4 – Паттерн Central Configuration

4.4 Паттерн централизованного логирования

Для того, чтобы иметь полное представление о состоянии всей системы в каждый момент времени, разработчики внедряют логирование – сбор информации в виде событий о работе программы в хронологическом порядке, происходящих с каким-либо объектом в системе. Такая информация обычно

записывается в специальные файлы или базу данных, называемые журналом событий.

Поскольку микросервисная архитектура подразумевает одновременную работу нескольких таких программ, то возникает несколько проблем, связанных с логированием. Во-первых: если пользователи будут обращаться к разработчикам системы о какой-либо проблеме в приложении, то как можно обнаружить, какие экземпляры стали причиной возникновения этой проблемы? Во-вторых, как получить полное представление о том, что происходит во всей системе, когда один какой-либо экземпляр создает событие и записывает его в соответствующий журнал? В-третьих, как можно узнать, возникли ли проблемы у каких-либо экземпляров, начали ли они записывать информацию об ошибках в свой журнал событий? Для этого применяется паттерн централизованного логирования (рисунок 5), реализуемый так же, как и прошлые паттерны, в виде отдельной программы. Такой паттерн подразумевает использование централизованного журнала событий, в который и будет записываться информация о работе системы в хронологическом порядке для дальнейшего её просмотра и анализа.

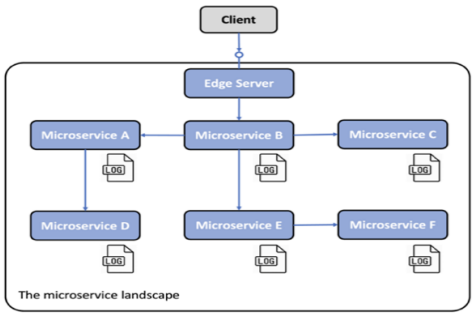

Рисунок 5 – Паттерн централизованного логирования

5 Технология gRPC

Недостатки HTTP/1.1 представлены ниже.

1) При необходимости отправки очередного запроса по данному протоколу нужно постоянно создавать новое TCP-соединение, что порождает накладные расходы [6].

2) Head-of-Line Blocking – проблема, заключающаяся в том, что в очереди на скачивание нескольких файлов, один из которых может иметь большой размер и стоять в её начале, может произойти её блокировка, поскольку необходимо будет дожидаться полного скачивания этого файла. Данная проблема объясняется тем, что при таком протоколе создаётся лишь одно TCP-соединение, и не происходит никакого распараллеливания при скачивании данного набора файлов. Например, для загрузки браузером обычной HTML-страницы с файлами JavaScript последние имеют гораздо больший размер по сравнению с файлами формата HTML и CSS. В качестве решения данной проблемы браузеры создают несколько TCP-соединений, но, как было отмечено выше, это создаёт дополнительные накладные расходы и, помимо этого, такой подход неэффективен.

3) При сжатии заголовков запроса используется алгоритм, который так или иначе поддаётся взлому. Это опасно, например, тем, что в случае присутствия в заголовках Cookie, которые могут содержать информацию о сессии пользователя, содержимое Cookie может быть легко перехвачено злоумышленниками.

4) Поскольку данный протокол предполагает, что взаимодействие между клиентом и сервером происходит в форме «запрос-ответ», причём HTTP – протокол без состояния, то невозможно отправить сервером по своей инициативе клиенту важной информации. Такая проблема решается, например, с помощью протокола двунаправленной передачи данных Websocket, работающего поверх HTTP, но сам такой протокол предполагает большие накладные расходы.

Спустя время была выпущена новая версия протокола HTTP, а именно HTTP/2, которая решала данные проблемы. И именно эта версия протокола стала основой для такой технологии, как gRPC.

RPC – форма взаимодействия между клиентом и сервером, при которой вместо обычного HTTP-запроса используется вызов функции на других, удалённых машинах. RPC расшифровывается как «удалённый вызов процедур» (Remote Procedure Call). Удалённый вызов процедур позволяет скрыть детали взаимодействия и обеспечить прозрачность доступа.

gRPC – технология удалённого вызова процедур с открытым исходным кодом, предоставляющая возможность многостороннего взаимодействия с приложениями в довольно понятном виде. Разработана компанией Google в 2015 году. Эта технология включает в себя 3 основных составляющих:

1) формат обмена данными Protocol Buffers (описан далее);

2) протокол передачи данных HTTP/2 (описан выше);

3) язык описания интерфейса IDL.

IDL – специальный язык описания интерфейса, позволяющий наиболее удобно представить сервисы удалённого вызова процедур. Синтаксис описания схож с описанием классов в любом объектно-ориентированном языке программирования, а само описание создаётся и хранится в файлах с расширением proto.

Технология gRPC позволяет организовывать взаимодействия клиента и сервера с помощью следующих режимов:

1) двунаправленный обмен – данные передаются в обоих направлениях;

2) потоковая передача сервера;

3) потоковая передача клиента;

4) однонаправленный обмен.

Рисунок 6 иллюстрирует вышесказанное.

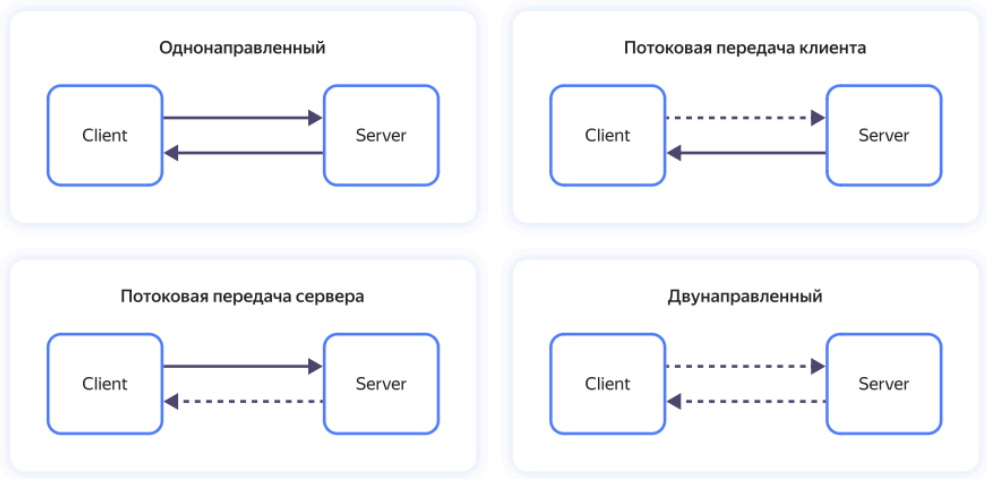

Рисунок 6 – Режимы обмена данными в gRPC

Технология gRPC, как и любая другая реализация RPC, имеет следующий принцип работы. У клиента и сервера есть такие программные интерфейсы, как заглушки. Они необходимы для предоставления клиентскому и серверному приложению соответственно интерфейса удалённого вызова функций, а также для преобразования байтов входящих сообщений в структуры конкретного языка программирования. Сначала клиентское приложение инициирует вызов определенной функции, создавая при этом параметры вызова. Такой процесс называется ещё маршаллингом, который реализуется с помощью бинарного протокола обмена данными Protobuf, речь о котором пойдёт ниже. Затем клиентская заглушка объединяет их в некоторое сообщение и организует его отправку gRPC-серверу. На стороне сервера, в свою очередь, идёт распаковка параметров вызова с помощью того же Protobuf, отработка необходимой функции, создание ответного сообщения и дальнейшей отправки его к клиенту. И наконец, обработка полученного сообщения и предоставление результата клиентской машине. В итоге создаётся иллюзия локального выполнения процедуры. На рисунке 7 представлена наглядно архитектура gRPC.

Рисунок 7 – Архитектура gRPC

5.1 Формат обмена данными Protobuf

В контексте проектирования микросервисной архитектуры вопрос передачи данных между компонентами сервера очень важен, поскольку данные между ними передаются по сети, что может отражаться на производительности приложения [7]. Долгое время стандартом построения микросервисной архитектуры являлся архитектурный стиль REST, реализацией которого является протокол HTTP, а также формат передачи данных JSON. Один из очевидных и весомых плюсов JSON – использование понятного человеку текстового формата. Но также имеются и недостатки, которые перечислены ниже.

1) Динамическая схема: набор полей, передающихся от одного микросервиса к другому при таком формате, строго не регламентирован. Это означает, что этот момент держится лишь на договорённости между коллегами по разработке системы, и ничто не препятствует случайно поменять структуру набора этих полей.

2) Отсутствие типизации: формат JSON представляет собой набор пар «ключ-значение» и не содержит информацию о типе данных значений. Поэтому если кто-то из команды разработчиков в какое-либо поле вставит

значение другого типа, то некоторые компоненты системы могут неправильно обработать такие данные, что может привести к ошибкам или непредвиденному поведению программы.

3) Большой размер данных: согласно протоколу HTTP 1.1, тело запроса передаётся как текст, и следовательно, оно может иметь большой размер и плохую оптимизацию.

Чтобы решить данные проблемы, был разработан новый формат передачи данных, называемый Protocol Buffers (сокращённо – Protobuf), который и стал использоваться в технологии gRPC. Помимо этого, данный формат имеет поддержку нескольких языков программирования, что позволяет определять свои gRPC-сервисы, а также взаимодействовать с существующими, не привязываясь к конкретному языку программирования.

Protobuf является буферным протоколом сериализации структурированных данных, преобразование которых происходит в двоичный формат. Такой формат даёт возможность наиболее эффективного сжатия пакетов сообщений.

Технология gRPC обладает большим количеством возможностей. Рассмотрим некоторые из них.

5.2 Крайние сроки и время ожидания

Для более разумного использования имеющихся ресурсов необходимо, чтобы при долгом формировании ответа сервера у клиента была возможность не ждать этот ответ, поскольку в некоторых случаях причиной большой задержки могут быть внутренние ошибки на стороне сервера, и следовательно, ответ может и не сформироваться вовсе [8]. Технология gRPC позволяет задать такие параметры, как время ожидания и крайний срок. Введение разработчиками данной технологии этих параметров обусловлено тем, что в сети могут присутствовать задержки. Теперь допустим, имеется некоторый

удалённый вызов, который влечёт за собой вызовы, связывающие несколько сервисов.

Время ожидания – параметр, который можно установить для каждого из вызовов при обращении к каждому следующему сервису.

Крайний срок – параметр, который устанавливается для всего жизненного цикла удалённого вызова. То есть это общее время ожидания всего вызова, который уже не учитывает отрезки времени промежуточных вызовов. Это тот срок, до завершения которого должна быть выполнена вся цепочка необходимых вызовов. Если не устанавливать никакое значение данному параметру, то клиент может ждать неограниченное время ответа от сервера, в результате чего могут быть исчерпаны все ресурсы gRPC-сервера, а в худшем случае – сервер может выйти из строя.

5.3 Перехватчики

При выполнении некоторого набора запросов, вне зависимости от того, какой это запрос, до или после него могут быть необходимы проделаны некоторые операции, отвечающие требованиям системы. Сюда входят такие рутинные задачи, как проверка аутентификации, сбор некоторых метрик, логирование. Для этого в gRPC разработан механизм перехватчиков, позволяющий организовывать выполнение потребительских требований. Перехватчиков может быть несколько, в зависимости от нужд приложения. Технология gRPC предоставляет 2 типа перехватчиков: унарные и потоковые, которые используются в унарных и потоковых удалённых вызовах соответственно.

5.4 Механизм отмены

Если рассматривать один какой-либо удалённый вызов, то он может быть успешно отработан и на клиенте, и на сервере. Но может быть, например,

так, что на сервере вызов отработал успешно, а на клиенте – завершился с ошибкой. В связи с этим gRPC поддерживает механизм отмены, который позволяет вызывающей стороне отменить выполнение данного вызова, после чего информация о его отмене передаётся противоположной стороне.

5.5 Обработка ошибок

При вызове определённой процедуры формируется ответ, который содержит код состояния и сообщение, будь то об успешном выполнении или об ошибке. Технология gRPC поддерживает наборы кодов состояния. Некоторые из них:

1) OK – успешное выполнение;
2) CANCELLED – операция отменена;
3) DEADLINE_EXCEEDED – истечение параметра крайнего срока;
4) INVALID_ARGUMENT – указан недопустимый аргумент при вызове.

5.6 Мультиплексирование

Технология gRPC позволяет использовать одно соединение для нескольких вызовов, что и называется мультиплексированием. На одном gRPC-сервере можно выполнять несколько вызовов разных сервисов одновременно.

5.7 Метаданные

В ряде случаев необходимо вызывать не только процедуры, связанные с пользовательскими требованиями, но и процедуры, которые бы возвращали информацию об RPC-вызовах, либо же наоборот передавали её. Для этого в gRPC предусмотрен механизм метаданных, которые может отправлять и

принимать как сервер, так и клиент. Например, наиболее распространённая такая информация – обмен данными о заголовках безопасности gRPC-вызовов.

6 Событийно-ориентированная архитектура

Как уже было сказано, выбор в пользу микросервисной архитектуры даёт возможность независимого развёртывания любого микросервиса при условии, что сервисы обладают слабой связанностью между собой [9]. Как показывает опыт, из всех существующих систем самую низкую степень взаимосвязи обеспечивает событийно-ориентированная архитектура.

Чтобы понять данную архитектуру, необходимо ввести понятие события в распределенной системе. Событие – существенное изменение состояния системы, факт совершения которого может представлять интерес для компонентов данной системы. Событийно-ориентированная архитектура – парадигма, которая предполагает различного рода взаимодействия с событиями: их создание, использование, обнаружение, реагирование на них.

В событийно-ориентированной архитектуре основными понятиями также являются понятия производителя событий, потребителя и брокера сообщений [10].

Производитель – сторона, заинтересованная в создании конкретных событий. Потребитель - сторона, заинтересованная в обнаружении и дальнейшей обработке конкретных событий, которые создаются производителями. Основные принципы событийно-ориентированной архитектуры представлены ниже.

1) Потребители реагируют на события производителя без его ведома. Также потребитель необязательно является конечным адресатом: он может быть промежуточным звеном во всей цепочке, и может в ней, например, делать какую-либо промежуточную обработку поступившего к нему события. Затем он способен отправлять это событие следующим потребителям через брокера сообщений, речь о котором пойдёт далее.

2) Производитель не знает, кто является потребителем событий, которые он порождает. Производитель также не знает, существует ли хотя бы один потребитель для его событий.

Таким образом, каждая передача события в событийно-ориентированной архитектуре является «слепой трансляцией». В контексте микросервисной архитектуры и производителями, и потребителями могут выступать какие-либо микросервисы, причём допускается, чтобы один компонент выполнял обе эти роли. Если получится каким-либо образом реализовать данные концепции, то обеспечится и слабая связанность между компонентами системы. И чтобы реализовать данные концепции, было принято ввести понятие брокера сообщений, который бы и занимался распределением поступающих событий. Вместо взаимодействия напрямую, производители и потребители взаимодействуют с этим брокером. И в случае, если кто-то из производителей или потребителей выйдет из строя, не придётся в этом случае менять код у зависящих от него сторон, ведь это всё останется на плечах у брокера сообщений. А это хорошо известная проблема: системы с высокой степенью взаимосвязи компонентов подвержены коррелированным отказам. Это означает, что при отказе одного из объектов могут выйти из строя и те, которые от него зависят. Поэтому введение понятия брокера сообщений – разумный подход.

7 Развёртывание и Kubernetes

Раньше, на заре становления цифровой эпохи, программисты разрабатывали сервера монолитной архитектуры [11]. Такие приложения обновлялись довольно редко. По окончании написания кода программного продукта, разработчики передавали его системным администраторам для того, чтобы они развёртывали продукт, а также следили за его состоянием с помощью различных метрик, логов и других необходимых инструментов.

Но с появлением микросервисной архитектуры и технологии контейнеризации начали по-другому развёртывать программное обеспечение. Концепция микросервисов, как было сказано ранее, позволяет уделять большее внимание конкретным частям сервера, чем монолит. Это позволяет быстро и часто создавать, менять и удалять необходимые компоненты системы.

Но с расширением бизнес-требований к приложению система становится всё более разрастающейся, а количество компонентов становится очень большим. Вследствие этого становится тяжелее системным администраторам развёртывать такую систему и следить за ней вручную. Становятся актуальными задачи такого размещения компонентов системы по имеющимся машинам, чтобы производительность и использование ресурсов достигали максимальных значений нужных показателей. Всё это также привело бы к снижению затрат на необходимое оборудование. В связи с этим появлялись потребности в программе, которая бы решала данные задачи.

Одна из таких программ – Kubernetes. Это платформа контейнерной оркестрации, предназначенная для управления и развёртывания контейнерных приложений. Kubernetes разработан компанией Google и со временем стал общепринятым стандартом контейнерной оркестрации.

Kubernetes является удобным инструментом как для системных администраторов, так и для разработчиков. Разработчики могут самостоятельно развёртывать свои приложения настолько часто, насколько

это необходимо, не прибегая к помощи системных администраторов. Для последних же эта платформа полезна тем, что она позволяет автоматически перемещать компоненты сервера при аварийных сбоях. Kubernetes также позволяет контролировать потребляемые приложением ресурсы, равномерно распределять нагрузку по всей инфраструктуре и многое другое.

Kubernetes обладает большим количеством возможностей, но полностью прочувствовать их все можно только в крупных дата-центрах. Данная платформа позволяет развёртывать приложения на тысячах, а также десятках и сотнях тысяч компьютеров. Процедура развёртывания никак не зависит от количества узлов системы.

На рисунке 8 представлена архитектура Kubernetes. Основной элемент Kubernetes – кластер [12]. На аппаратном уровне кластер состоит из нескольких узлов, которые делятся на два типа:

1) рабочие узлы, на которых хранятся один или несколько контейнеров и непосредственно выполняются приложения;

2) ведущий узел, который содержит плоскость управления, управляющую всей системой Kubernetes, а также взаимодействует с рабочими узлами, сообщая им, когда и какие контейнеры необходимо удалять и создавать; хранит данные о состоянии и конфигурации всего кластера.

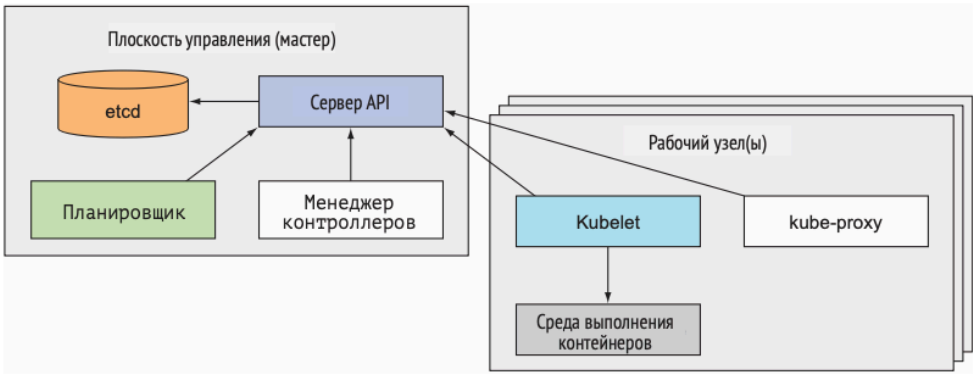

Рисунок 8 – Архитектура Kubernetes

Плоскость управления – это «сердце» кластера Kubernetes; то, что его заставляет функционировать. Она состоит из следующих компонентов:

1) сервер API, с которым взаимодействует программист и остальные компоненты плоскости управления;

2) планировщик, который распределяет компоненты системы по машинам, исходя из доступности; отслеживает нагрузку рабочих узлов в кластере;

3) менеджер контроллеров, который выполняет такие функции, как отслеживание рабочих узлов, обработка аварийных сбоев и так далее; управляет контроллерами реплик и узлов; взаимодействует с контурами регулирования, управляющими состоянием кластера;

4) etcd – распределённое хранилище данных «ключ-значение»; у каждого узла есть доступ к данному хранилищу и через него узлы узнают, как поддерживать конфигурацию своих контейнеров.

В узлах кластера Kubernetes приложения работают в контейнерах, и для этого, как правило, используется Docker. Помимо задачи выполнения, в узлах кластера происходит мониторинг и предоставление приложениям служб. За это отвечают следующие компоненты узлов кластера Kubernetes:

1) среда выполнения контейнеров (например, Docker);

2) kubelet - процесс-агент, обменивающийся с сервером API и отвечающий за управление состоянием узла;

3) kube-proxy – сетевой прокси, балансирующий нагрузку сетевого трафика для сервисов, запущенных на узле.

8 Практическая часть

В рамках данной выпускной квалификационной работы было разработано веб-приложение для организации онлайн-конференций. Это приложение представляет собой аналог популярного и широко используемого приложения Microsoft Teams.

В разработанном приложении представляются возможности работы с командами, чатами и конференциями. Пользователи могут создавать команды, вступать в них, добавлять новых участников. Далее, администраторы команд имеют возможность менять роли участников, что позволяет разграничивать права доступа для команд. Приложение позволяет создавать, изменять, удалять посты и получать их список, а также создавать, изменять и удалять комментарии к постам. Имеется возможность управлять участниками команд (получение списка участников, добавление и удаление), а также работать с ожидающими запросами. Ожидающие запросы представляют из себя запросы от пользователей, которые хотят вступить в определенную команду. Приложение обеспечивает следующие возможности для работы с чатами: получение списка чатов для пользователя, создание чата, добавление пользователя в чат по его электронной почте, отправка сообщений, получение списка сообщений чата, отслеживание статуса того, что сообщение прочитано собеседником, редактирование сообщение в режиме реального времени.

Рассматриваемое веб-приложение состоит из двух частей: клиентская (фронтенд) и серверная (бэкенд). Клиентская часть написана на Vue JS – фреймворке Java Script, а серверная представляет из себя приложение на основе микросервисной архитектуры. Бэкенд написан на трёх языках программирования: Java, Python и Golang. Это три языка программирования, которые очень часто используются для написания серверов. Каждый из этих языков имеет свои преимущества в написании серверной части.

Для серверной части рассматриваемого приложения в качестве архитектуры была выбрана микросервисная. Это было сделано вследствие

разделения большой предметной области на части: работа с пользователями, с чатами, командами и конференциями. Данный подход позволил вести разработку серверной части приложения более удобно, не хранив всю бизнес-логику в одной программе, а также упростить само написание кода в целом. Каждый микросервис отвечает за свою часть функциональности, что делает код более модульным, понятным и легко масштабируемым. Благодаря микросервисной архитектуре, серверная часть приложения становится более гибкой и легко масштабируемой. Если один из сервисов потребует некоторого обновления, то необходимые изменения можно внести независимо от других сервисов.

Микросервисная архитектура состоит из компонентов, приведённых ниже.

1) Gateway (единая точка входа в серверное приложение),

2) микросервис работы с пользователями,

3) микросервис работы с чатами и сообщениями,

4) микросервис работы с чатами и командами,

5) микросервис работы с конференциями.

Для написания серверной части рассматриваемого веб-приложения использовались не только возможности самих языков программирования, но также и широко известные в своих узких сферах инструменты, которые позволяют разрабатывать микросервисную архитектуру, наделяя её хорошими свойствами, такими как отказоустойчивость, гибкость и надёжность. К таким инструментам относятся системы управления базами данных, брокеры сообщений, технологии обеспечения кеширования и полнотекстового поиска, мониторинга приложения (анализ метрик и логов), а также технологии передачи данных.

В качестве СУБД в серверной части разработанного веб-приложения была выбрана PostgreSQL вследствие большой информационной базы для внедрения этой СУБД в свои приложения, а в качестве инструмента кеширования – Redis.

В результате разработки веб-приложения протоколами взаимодействия клиентской части с серверной стали HTTP и Websocket. Рассмотрим второй из этих протоколов подробнее.

Протокол HTTP является распространённым протоколом передачи данных. Однако он имеет несколько недостатков.

1) Клиент должен постоянно создавать новое TCP-соединение с сервером, так как соединение существует лишь во время обработки запроса сервером; это приводит к лишней сетевой нагрузке и относительно большим накладным расходам;

2) новые данные, полученные сервером и актуальные для данного клиента, не могут быть отправлены последнему моментально: для их получения необходим ещё один HTTP-запрос.

Все эти проблемы привели к созданию нового протокола двунаправленной передачи данных в реальном времени Websocket.

Протокол Websocket – протокол взаимодействия между клиентом и сервером, предоставляющий возможность двунаправленного обмена данными. Работает поверх протокола транспортного уровня TCP модели OSI. На рисунке 9 представлена схема работы протокола Websocket.

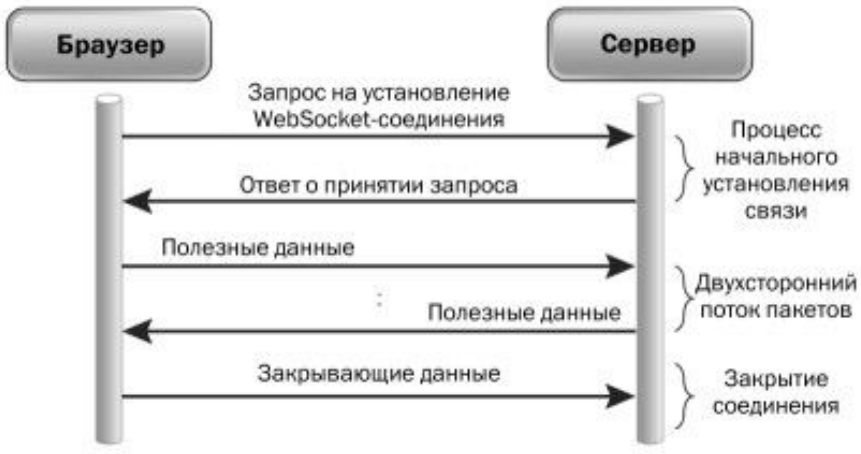

Рисунок 9 – Схема работы протокола Websocket

Принцип работы данного протокола следующий. Вначале клиент отправляет TCP-запрос серверу, который содержит заголовок Upgrade со значением websocket. Такой заголовок говорит о том, что мы желаем создать с сервером Websocket-соединение. После того, как сервер принимает такое предложение клиента (такой процесс ещё называют «рукопожатием»), устанавливается TCP-соединение на всё время до отключения одного из участников взаимодействия, и начинается двунаправленный обмен данными.

Но встаёт вопрос о том, как сервер может понимать, подключён ли ещё клиент к нему, ведь клиент может отключиться от соединения, не предупредив никак об этом сервер (например, компьютер неожиданно вышел из строя), а сервер бы всё равно считал клиент подключённым к нему. Для этого согласно протоколу Websocket необходимо, чтобы клиент посылал некоторые служебные сообщения с определённым тайм-аутом. Также в ответ таким сообщениям необходимо и серверу рассылать ответные сообщения похожего формата. И если некоторое служебное сообщение от клиента пришло позднее назначенного срока, то клиент считается отключённым от Websocket-соединения.

Для закрытия соединения необходимо отправить серверу соответствующее Websocket-сообщение о закрытии соединения. В ответ сервер так же должен отправить ответное сообщение.

Может показаться, что этот протокол лучше протокола HTTP во всём. Но у протокола Websocket также есть и свои недостатки. Один из основных заключается в следующем. В случае, если клиенту необходимо получать от сервера некоторые данные, не меняющиеся очень часто и не требующие их доставки клиенту в режиме реального времени (например, получение HTML-страниц от сервера), создание Websocket-соединения может лишь тратить ресурсы устройства, и это будет избыточным. Также, если нет надобности поддерживать постоянное соединение, протокол HTTP будет предпочтительнее.

8.1 Клиентская часть приложения

Как уже было сказано ранее, в качестве инструментов для написания клиентской части веб-приложения были использованы HTML, CSS и Vue JS - один из популярных веб-фреймворков для JavaScript. Веб-фреймворк Vue JS был выбран вследствие лёгкости его освоения, а также большой информационной базы по его изучению. Был использован стиль вёрстки flexbox, который позволяет создавать HTML-страницы на основе такого понятия, как «контейнер».

Как и любое другое веб-приложение, клиентская часть которого написана на любом фреймворке JavaScript, фронтенд разработанного в ходе работы продукта был написан на основе компонентов. Такой подход позволяет переиспользовать уже написанный код, не допуская повторений, тем самым удовлетворяя популярным и широко известным принципам написания чистого кода, таким как SOLID, DRY и так далее. Было создано 20 компонентов, каждый из которых представлял либо всю какую-либо HTML-страницу с JavaScript-кодом, либо её часть.

Компоненты клиентской части приложения приведены на рисунке 10.

1) MyAuthentication – компонент страницы аутентификации пользователя,

2) MyChat – компонент страницы текущего чата пользователя,

3) MyChatPage – компонент страницы списка чатов пользователя,

4) MyCommentContainer – компонент контейнера комментариев для конкретного поста определённой команды,

5) MyConfContainer – компонент контейнера списка конференций для определённой команды,

6) MyConference – компонент страницы определённой конференции для определённой команды,

7) MyCreateConference – компонент страницы создания конференции для определённой команды,

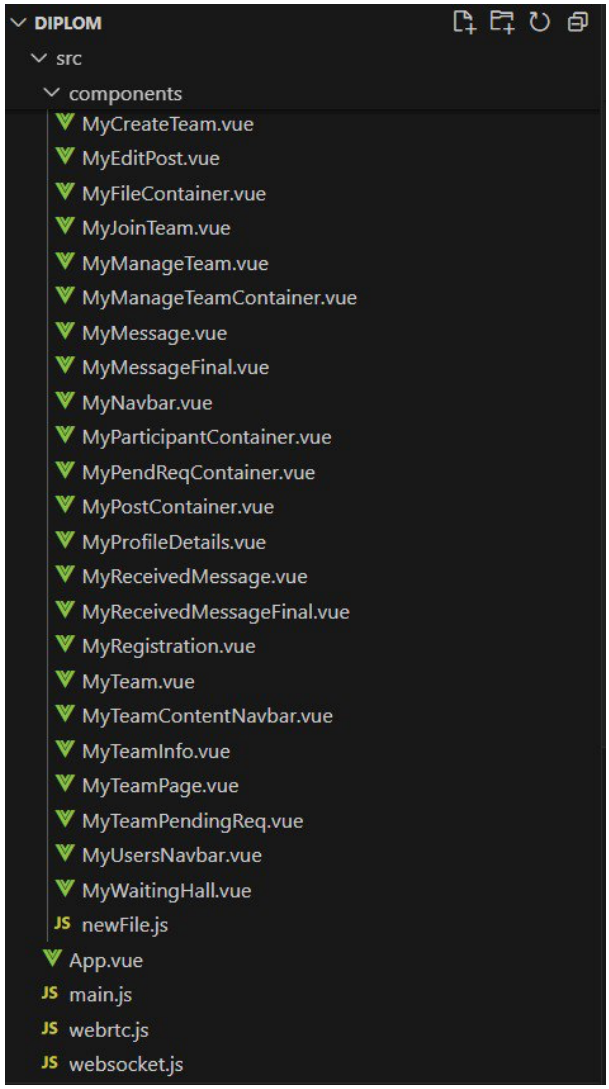

Рисунок 10 – Компоненты клиентской части приложения

8) MyCreatePost – компонент страницы создания поста определённой команды,

9) MyCreateTeam – компонент страницы создания команды,

10) MyEditPost – компонент страницы редактирования текущего поста определённой команды,

11) MyJoinTeam – компонент страницы вступления в определённую команду,

12) MyMessage – компонент сообщения определённого чата, отправленного текущим пользователем,

13) MyReceivedMessage - компонент сообщения определённого чата, отправленного каким-либо другим пользователем, являющимся участником чата (но не текущим),

14) MyNavbar – компонент навигационной панели приложения,

15) MyParticipantContainer – компонент контейнера списка пользователей для определённой команды,

16) MyPendReqContainer – компонент контейнера списка ожидающих запросов для определённой команды,

17) MyPostContainer - компонент контейнера списка постов для определённой команды,

18) MyTeam – компонент страницы текущей команды,

19) MyTeamPage – компонент страницы списка команд текущего пользователя,

20) MyWaitingHall – компонент страницы ожидающего зала для определённой конференции.

Помимо компонентов, также были созданы файлы websocket.js и webrtc.js, необходимые для конфигурации Websocket-соединений и WebRTC-соединений соответственно.

Взаимодействие с бэкендом происходило по протоколу HTTP и Websocket. Для общения по HTTP использовался метод fetch с конкретными параметрами запроса, такими как url, метод запроса (GET, POST), тело и некоторые заголовки запроса (при необходимости). Для взаимодействия по Websocket использовался класс Websocket, а также слушатели, срабатывающие при открытии соединения и при получении нового Websocket-сообщения.

Обмен медиапотоками происходит через WebRTC API, предоставляемый JavaScript. В качестве сигнального сервера выступал один из микросервисов, написанный на языке программирования Python. А в качестве STUN-сервера использовался общедоступный STUN-сервер от Google “stun.l.google.com:19302”.

Поскольку использовался компонентный подход к написанию клиентской части, было необходимо обеспечить взаимодействие компонентов между собой. В контексте двух каких-либо компонентов A и B, в веб-фреймворке Vue JS такое взаимодействие возможно лишь в том случае, когда один из этих компонентов является родительским для другого (соответственно оставшийся компонент будет дочерним). Будем считать, что компонент A – родительский для компонента B.

В случае, если поступающие на компонент A данные необходимо передать в компонент B, используются так называемые входные параметры (по-другому – пропсы). На рисунке 11 представлен пример взаимодействия компонентов через пропсы.

```
<template>
    <div class="post-container" v-if="replyNotClicked & createPostConst == false & editPostConst == false">
        <!-- <button @click="createPost">Создать пост</button> -->
                <div class="post" v-for="post in posts" :key="post.post_id" @click="editPost(post)"> <!--@click="deletePost(post)"-->
                    <div class="post-details">
                        <div class="member-img">
                            <img src="" alt="IMAGE">
                        </div>
                        <div class="member-name">{{ post.team_member_creator_username }}</div>
                        <div class="post-date-time">{{ post.postDate }}</div>
                </div>
                    <div class="post-content">{{ post.text }}</div>
                    <div class="post-comments">комментариев: {{ post.postCommentCount }}</div>
                    <div class="reply" @click="replyNotClicked = false">Ответить</div>

                </div>
        </div>
        <MyCreatePost v-if="createPostConst == true" @post="reloadPosts"/>
        <MyEditPost v-if="editPostConst == true" :text="currPostText" :post_id="currPostId" @editPost="editPostReload"/>
</template>
```

Рисунок 11 – Пример взаимодействия компонентов через пропсы

В данном примере на рисунке выше в родительском компоненте MyPostContainer есть дочерний компонент MyEditPost, которому в качестве пропсов передаются значения параметров text и post_id.

В случае, если поступающие на компонент B данные необходимо передать в компонент A, используются пользовательские события. На родительском компоненте создаётся слушатель предпочитаемого события (пусть оно для примера называется example) и с помощью директивы @example можно вызвать метод в родительском компоненте, который будет

выполняться при срабатывании данного события example. В дочернем компоненте же, при поступлении данных, можно вызвать метод $emit с обязательным параметром названия события и вторым параметром, указывающим на переменную с нужными данными (при необходимости).

В примере на рисунке 12 в родительском компоненте MyTeam есть дочерней компонент MyTeamContentNavbar, который содержит слушатели на несколько событий. В случае обновления данных на одном из дочерних компонентах на нём же вызывается метод $emit с необходимыми параметрами, и, следовательно, вызывается нужный слушатель в родительском компоненте, где также и реализован.

```
<template>
    <MyNavbar/>
    <div class="team-container">
        <MyTeamInfo :name="name"/>
        <div class="team-content">
            <MyTeamContentNavbar channelName="Общий" :createPost="createPost" :editPost="editPost" :deletePost="deletePost"
            @enablePosts="enablePosts()" @enableFiles="enableFiles()" @enableConfs="enableConfs()"
            @enableParticipants="enableParticipants()" @enablePendingRequests="enablePendingRequests()"/>
            <MyPostContainer v-if="postsEnabled"/>
            <MyFileContainer v-if="filesEnabled"/>
            <MyConfContainer v-if="confsEnabled"/>
            <MyParticipantContainer v-if="participantsEnabled"/>
            <MyPendReqContainer v-if="pendingRequestsEnabled"/>

            <!-- <MyCommentContainer :replyNotClicked="false"/> -->
        </div>
        </div>
</template>
```

Рисунок 12 – Пример взаимодействия компонентов через пользовательские события

В качестве роутера для маршрутизации используется библиотека vue-router, выбранная вследствие своей популярности и простоте.

На рисунке 13 показаны все маршруты, используемые в фронтенде приложения. Каждому маршруту ставится в соответствие компонент, который необходимо выводить.

Рисунок 13 – Маршруты клиентской части приложения

8.2 Серверная часть приложения

Ранее была представлена структура серверной части разработанного веб-приложения. Рассмотрим её подробнее.

8.2.1 Компонент Gateway

Данный компонент представляет собой единую точку входа в серверное приложение. Написан на Java с помощью фреймворка Spring.

Как уже было сказано ранее, это – один из паттернов микросервисной архитектуры, обеспечивающий возможность перенаправления запросов на нужные микросервисы. Это позволяет оставить за кадром для клиентской части IP-адреса остальных микросервисов, что является очень удобным, так как IP-адреса могут быть динамическими и постоянно меняющимися, из-за чего пришлось бы часто переписывать код клиентской части.

На рисунке 14 представлена файловая структура данного компонента:

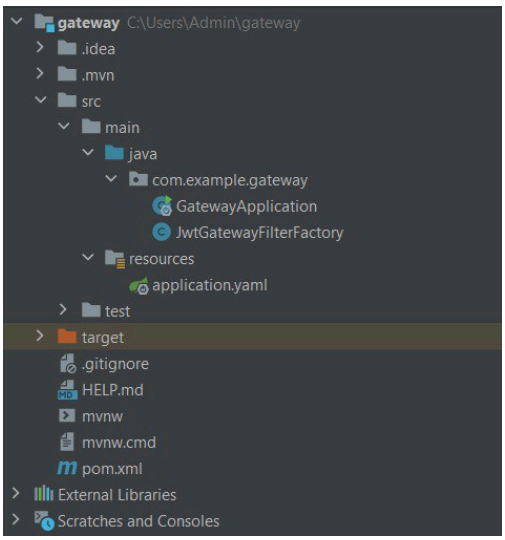

Рисунок 14 – Файловая структура компонента Gateway

Из основных файлов здесь стоит отметить следующие:

1) GatewayApplication – файл, который позволяет запустить приложение;

2) JwtGatewayFilterFactory – файл, который создаёт фильтр, проверяющий наличие JWT-токена в заголовке аутентификации запроса; запросы, содержащие валидный JWT-токен, идут дальше по цепочке к нужным микросервисам, а остальные – отклоняются;

3) application.properties – файл, содержащий настройки компонента (такие как порт, на котором он запускается, а также настройки gateway, такие как распределение по нужным маршрутам и настройки CORS);

4) pom.xml – файл с зависимостями компонента, в числе которых зависимости для подключения Spring и Spring Cloud Gateway.

CORS – стандарт, позволяющий браузеру пользователя получать разрешения на доступ к сторонним интернет-ресурсам. Сторонним считается такой интернет-ресурс, который отличается от запрашиваемого протоколом, доменом или портом. Для этого используются различные HTTP-заголовки и настройка CORS на стороне сервера. Те источники (origin), которые указаны на сервере, смогут получать доступ к ресурсам данного сервера. На рисунке 15 представлена схема работы CORS.

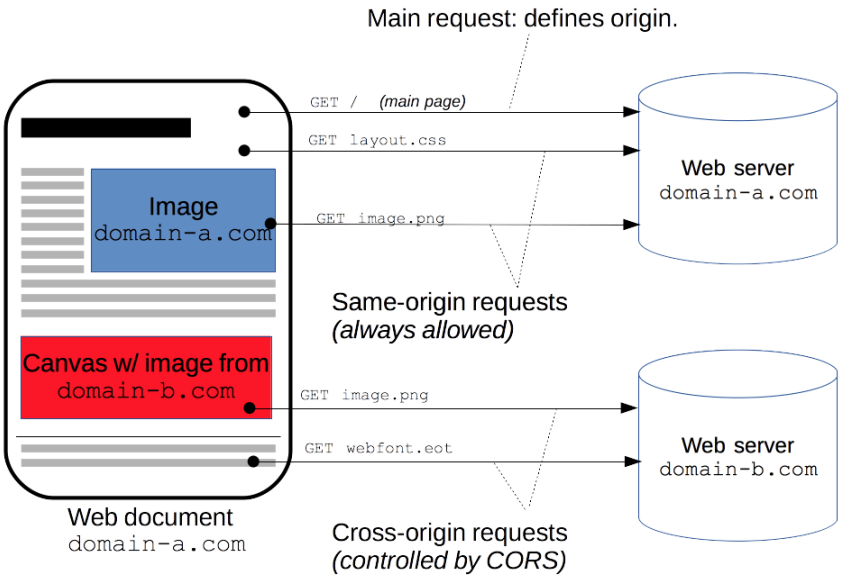

Рисунок 15 – Схема работы CORS

Проверка того, что запрос является авторизованным, происходит с помощью подхода, основанного на JWT-токенах. Рассмотрим подробнее данный подход.

JWT (JSON Web Token) – открытый стандарт аутентификации на основе токенов доступа.

Токен состоит из трёх частей: заголовок (header), полезные данные (payload) и подпись токена (signature). Сам токен имеет формат “заголовок.полезные_данные.подпись”.

Заголовок – строка, содержащая информацию о том, как должна выполняться подпись. Примером таких данных служит информация об алгоритме хеширования. Пример такого алгоритма: HS256 – используется при симметричном шифровании.

Полезные данные – такие данные, которые могут быть ассоциированы с текущим пользователем (его идентификатор, электронная почта и так далее).

Подпись – строка, вычисляющаяся на основе заголовка, полезных данных (и то, и то закодировано в формат base64) и ключа шифрования, определённого соответствующим алгоритмом.

Поскольку первые 2 части никак не шифруются, а лишь кодируются, то в случае перехвата JWT-токена злоумышленником он сможет увидеть их содержимое. Это означает, что в полезных данных не следует хранить секретные данные (например, пароль пользователя). Но наличие подписи не позволяет злоумышленнику подделать данные, поскольку на стороне сервера происходит проверка равенства подписи пришедшего JWT-токена вычисляемой подписи (вычисление происходит на основе тех же первых двух частей токена и ключа шифрования). Таким образом, к микросервисам могут поступать только авторизованные запросы, а если запрос не авторизован – этот компонент будет высылать ответ с соответствующей ошибкой. На рисунке 16 представлен пример JWT-токена.

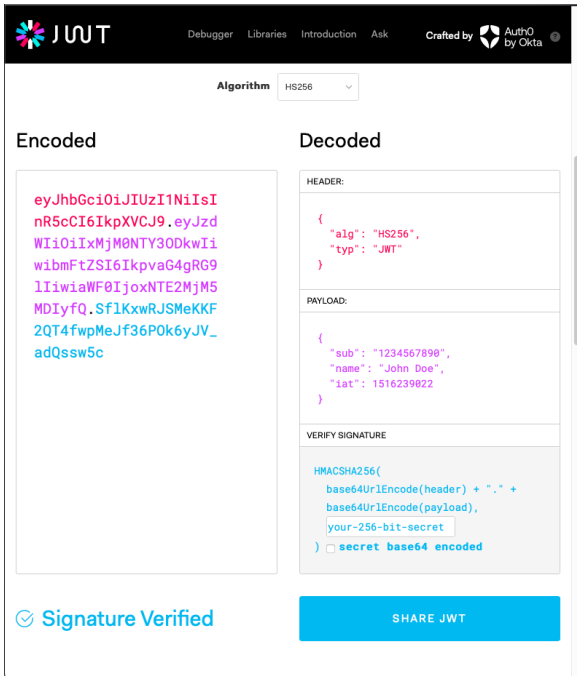

Рисунок 16 – Пример JWT-токена

8.2.2 Микросервис для работы с пользователями

В данном компоненте реализована такая логика, как регистрация и авторизация пользователей, а также получение пользователя по его почте. Написан на Java с помощью фреймворка Spring.

При реализации данного компонента использован паттерн Контроллер-Сервис-Репозиторий, который позволяет разграничивать ответственность за составляющие в коде.

Контроллеры – классы, отвечающие за обработку запросов. Это может включать в себя вызов методов сервисного слоя, обработка исключений, формирование ответов.

Сервисы – классы, отвечающие за бизнес-логику, которая включает в себя вызов методов репозитория, методов, отвечающих за кеширование, при необходимости остальных вспомогательных методов и другое.

Репозитории – классы, отвечающие за взаимодействие с базой данных (сохранение, извлечение, обновление данных).

Каждый из данных слоёв знает о существовании лишь одного нижестоящего для него слоя согласно порядку в названии данного паттерна.

Основные 2 эндпоинта данного компонента – регистрация и авторизация пользователя – обрабатываются с помощью паттерна Контроллер-Сервис-Репозиторий. Оба запроса на эти эндпоинты принимает контроллер AuthController, преобразует тело запроса в классы моделей, при их обработке вызываются методы сервиса UserDetailsServiceImpl, и затем в нём – методы репозитория UserRepository. Данный репозиторий же, в свою очередь, взаимодействует с базой данных этого компонента. После отработки методов сервиса идёт возвращение на слой контроллеров и формирование ответа.

8.2.3 Микросервис для работы с командами

Данный компонент предоставляет API для взаимодействия с такими сущностями, как команды, а также с тем, что к ним относится (посты, комментарии к постам, ожидающие запросы, участники). Написан на языке Python, фреймворк – FastAPI. На рисунке 17 представлена файловая структура данного компонента:

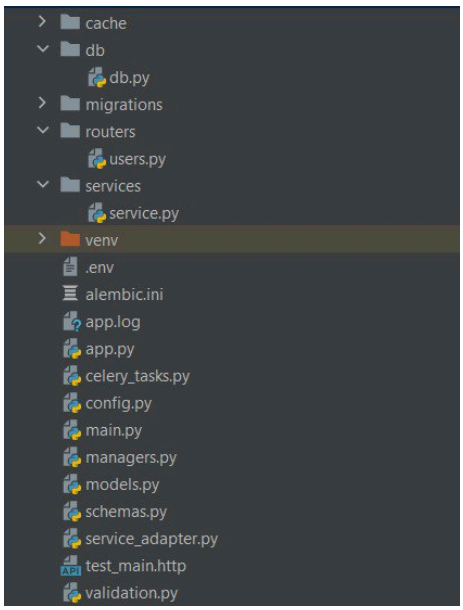

Рисунок 17 – Файловая структура компонента

Из основных файлов здесь стоит отметить следующие:

1) db.py – файл, хранящий методы для взаимодействия с базой данных;

2) service.py – файл, хранящий методы сервисного слоя;

3) app.py – файл, в котором реализован класс экземпляра приложения;

4) main.py – файл, в котором хранятся методы слоя контроллеров;

5) models.py – файл, хранящий описание классов-моделей;

6) service_adapter.py – вспомогательный файл, внедрённый по техническим причинам;

7) schemas.py, validation.py – файлы, хранящие классы тел запросов.

Помимо эндпоинтов, которые обрабатываются по протоколу HTTP, здесь также обрабатываются Websocket-сообщения, то есть клиенты держат с данным компонентом Websocket-соединение.

За работу с Websocket-соединениями отвечает класс WebsocketManager, который имеет методы, слушающие события добавления и удаления очередного Websocket-соединения, а также приём новых Websocket-сообщений. Для того, чтобы хранить какое-либо Websocket-соединение и

легко ассоциировать его с пользователем, существует список словарей, где очередной словарь представляет собой полную информацию о пользователе. Значениями этого словаря являются идентификатор пользователя и его Websocket-соединение. Процесс подключения клиента к серверу по протоколу Websocket и дальнейшего взаимодействия между ними происходит следующим образом:

1) клиент делает HTTP-запрос на эндпоинт компонента, который принимает запросы на установку Websocket-соединения и прослушивание сообщений от клиента; данный запрос содержит заголовок Upgrade со значением Websocket, что говорит о том, что клиент хочет установить с сервером Websocket-соединение;

2) после принятия запроса сервер разрешает дальнейшее взаимодействие по данному протоколу, поскольку серверный эндпоинт это позволяет сделать; после этого создаётся словарь для этого клиента, речь про который была выше;

3) затем начинается общение клиента и сервера по протоколу Websocket.

Данное Websocket-соединение проходит и через компонент Gateway, поэтому это не противоречит паттерну единой точки входа в приложение.

У данного эндпоинта под капотом работает бесконечный цикл while, который и слушает Websocket-сообщения от клиента. Поскольку протокол Websocket – асинхронный, то это позволяет компьютеру принимать не только Websocket-сообщения, но и простые HTTP-запросы, в одном потоке. Поэтому данный бесконечный цикл не мешает обработке простых HTTP-запросов.

Структура Websocket-сообщений представляет собой словарь, в котором по ключу payload хранится вложенный словарь. Этот вложенный словарь хранит в себе две пары «ключ-значение»: параметры необходимого запроса и название этого запроса (чтобы на стороне сервера распознавать, что необходимо сделать с данными параметрами). Для наглядности, на рисунке 18 приведено одно такое сообщение:

```
websocket.send(JSON.stringify(
{'payload':
    {'request': 'applyJoinTeam',
    'params': {
        'user_id': Number(localStorage.getItem('userId')),
        'username': localStorage.getItem('userName'),
        'email': localStorage.getItem('userEmail'),
        'team_id': this.teamId
        }
    }
}))
```

Рисунок 18 – Пример Websocket-сообщения

На стороне рассматриваемого компонента Websocket-эндпоинт принимает данное сообщение, отправляет в обработчик handle_message, и с помощью оператора switch-case определяет, какой запрос пришёл от клиента и как необходимо обработать этот запрос. На рисунке 19 представлена обработка данного сообщения:

```
elif payload['request'] == 'applyJoinTeam':
    req = ApplyJoinTeamRequest(
        user_id=payload['params']['user_id'],
        username=payload['params']['username'],
        email=payload['params']['email'],
        team_id=payload['params']['team_id'],
    )
    resp = await apply_join_team(req, db)
    await websocket.send_json(resp)
    resp2 = await get_users_for_team(req.team_id, db)
    for usr in resp2['result']['users']:
        ws = self.get_ws_by_user_id(usr['user_id'])
        await ws.send_json(resp)
```

Рисунок 19 – Код обработки Websocket-сообщения

Из рисунка видно, что при обработке происходят следующие действия:

1) извлечение параметров запроса из сообщения;
2) формирование из них экземпляров классов Python (о них речь пойдёт дальше);
3) вызов метода сервисного слоя;
4) получение результата из этого метода;
5) формирование ответа;
6) отправка ответного Websocket-сообщения клиенту.

Также сообщение от сервера может приходить и другим клиентам, которым необходимо получать данное сообщение. Для этого в классе WebsocketManager реализован метод get_ws_by_user_id, где по идентификатору пользователя возвращается его Websocket-соединение и записывается в переменную. В момент отработки данного метода у этой переменной как класса Websocket (стандартный класс данного протокола в FastAPI) вызывается метод отправки данного сообщения необходимым клиентам.

Данный компонент принимает и обрабатывает следующие запросы: создание команд, вступление в команды, добавление новых участников, работа с постами (чтение, создание, изменение, удаление), работа с пользователями (удаление, изменение их роли в команде), работа с ожидающими запросами (получение, принятие и отклонение запроса), получение списка конференций и их создание.

В данном компоненте также реализован паттерн Контроллер-Сервис-Репозиторий. Контроллер расположен в файле main.py, сервис – в service.py, репозиторий – в db.py. Также есть файл service_adapter.py, который был создан для избавления от ошибки циклических импортов, связанных со структурой веб-фреймворка FastAPI. В этом файле находятся методы, которые просто вызывают соответствующие методы из сервисного слоя, а сами методы из файла service_adapter.py используются для вызова из слоя контроллера.

Также в данном компоненте представлены методы, которые работают с Redis в целях кеширования необходимых данных. Эти методы описаны в файле cache.py. Redis работает в контейнере Docker на стандартном порту.

Для более удобного обращения к нужным инструментам был создан класс самого приложения компонента TeamsMicroservice, который имел лишь один экземпляр, объявленный в файле main.py. Поля этого класса составляют экземпляр веб-фреймворка FastAPI, переменная логирования приложения и экземпляр класса, отвечающего за кеширование в Redis.

8.2.4 Микросервисы для работы с чатами и сообщениями

Данные компоненты предоставляют такие возможности, как взаимодействие с сущностями чатов и сообщений в приложении. Написаны на языке Golang, веб-фреймворк – Gin. На рисунках 20 и 21 представлены файловые структуры данных компонентов:

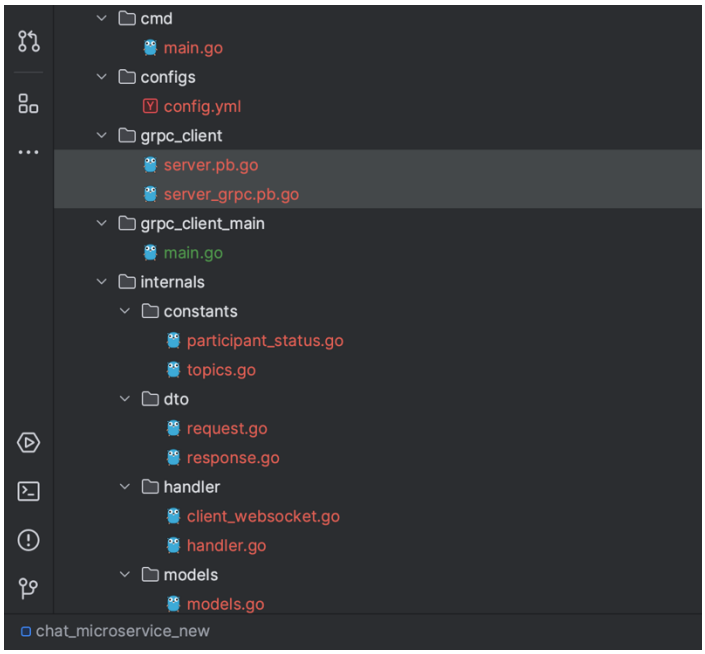

Рисунок 20 – Файловая структура микросервиса для работы с чатами

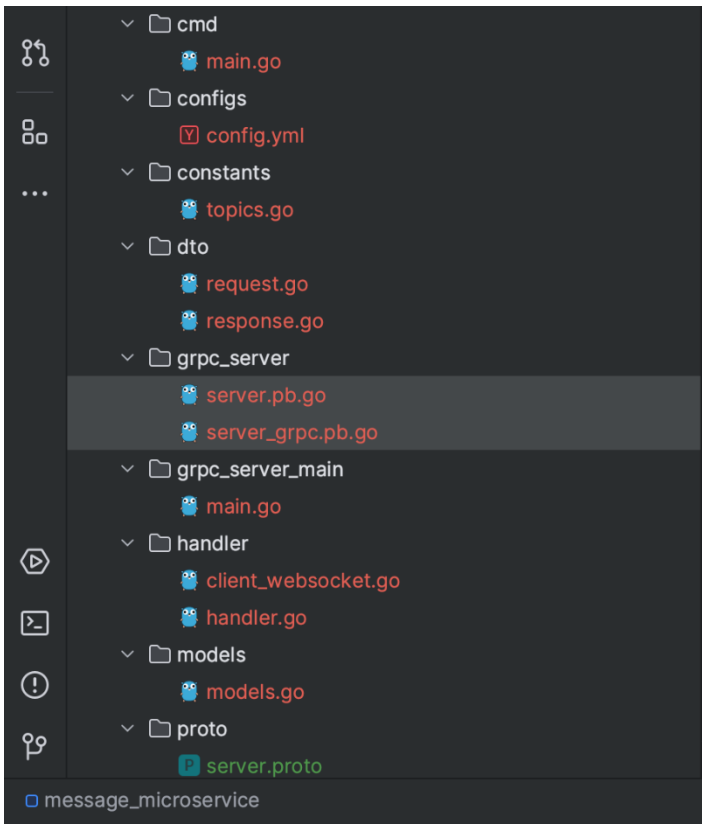

Рисунок 21 – Файловая структура микросервиса для работы с сообщениями

Предназначение данных файлов:

1) cmd/main.go – файл, запускающий текущий компонент;

2) config.yml – конфигурационный файл, хранящий некоторые настройки в виде пар «ключ-значение»;

3) requests.go, response.go – файлы, хранящие структуры тел запросов и ответов;

4) grpc_client.go, grpc_server.go – файлы, сгенерированные автоматически при подключении gRPC к данным компонентам на основе файла server.proto;

5) grpc_client_main.go, grpc_server_main.go – файлы, запускающие gRPC-клиент и gRPC-сервер соответственно;

6) client_websocket.go – файл, в котором хранятся методы чтения и записи Websocket-сообщений;

7) handler.go – файл, описывающий структуру основного обработчика запросов;

8) models.go – файл, описывающий таблицы в базе данных с помощью соответствующих структур.

Данные компоненты позволяют взаимодействовать с ними по протоколу Websocket, описанному ранее. Для создания Websocket-соединения была выстроена следующая архитектура. Изначально клиент посылает запрос на эндпоинт serveWS, который способен создавать Websocket-соединение. Затем после подтверждения со стороны сервера создания такого соединения создаётся переменная, хранящая данное соединение. После этого создаётся экземпляр структуры клиента, которая содержит в качестве полей само соединение и идентификатор пользователя, присущий этому соединению, и добавляется в переменную списка всех клиентов. Наконец, вызываются две горутины: горутина, в которой происходит чтение Websocket-сообщений от клиента, и горутина, в которой происходит отправка Websocket-сообщений клиенту. Такой порядок действий при создании соединения используется для всех клиентов, подключающихся к этим компонентам.

Между рассматриваемыми компонентами реализовано взаимодействие с помощью технологии gRPC. В разработанном проекте микросервис сообщений реализован как gRPC-сервер, а микросервис чатов – как gRPC-клиент.

Как уже было сказано ранее, технология gRPC позволяет организовывать взаимодействие между компонентами путём вызова удалённых процедур. Рассмотрим реализацию gRPC-сервера в разработанном приложении. Для того, чтобы его реализовать, необходимо написать специальный файл с расширением proto. Подобные файлы для реализации сложных gRPC-систем, могут быть устроены так же сложно, но в процессе разработки данных компонентов такой файл содержал реализацию простейшего gRPC-сервера. После их написания с применением некоторой команды, описанной ниже, произойдёт автоматическая генерация кода,

позволяющая работать с содержимым proto-файла. На рисунке 22 представлено содержимое файла server.proto, о котором и идёт речь:

```
syntax = "proto3";

package proto;

option go_package = "./proto";

service MessageGrpcService {
  rpc GetLastMessageOfChat(GetRequest) returns (GetResponse) {}
}

message GetRequest {
  int32 chatId = 1;
}

message GetResponse {
  int32 messageId = 1;
}
```

Рисунок 22 – Содержание файла server.proto

Структура данного файла:

1) syntax – ключевое слово, указывающее на формат файла и на версию;

2) package – ключевое слово, указывающее на рабочую директорию, в которой будет храниться автоматически сгенерированный код;

3) option go_package – опция, указывающая на полный путь до пакета, где будут лежать сгенерированные файлы;

4) service – ключевое слово, объявляющее интерфейс некоторого сервиса;

5) rpc – ключевое слово, объявляющее процедуру без её реализации;

6) message – ключевое слово, объявляющее некоторый класс с полями, описанными разработчиком в файле.

Как видно из рисунка выше, процедура GetLastMessageOfChat содержит класс GetRequest в качестве входных параметров для неё, а также возвращает некоторый результат в виде класса GetResponse. Объявления классов GetRequest и GetResponse описаны ниже сервиса. Сообщение GetRequest содержит поле chatId, указывающее на идентификатор нужного чата, а сообщение GetResponse – messageId, указывающее на идентификатор последнего сообщения данного чата. Также необходимо заметить, что полям chatId и messageId присвоены некоторые целочисленные значения. Это не значения, которые хранят эти поля. Это номера полей в сообщении. Их предназначение – сохранить логический смысл своих полей, названия которых могут меняться разработчиками со временем. Они используются при сериализации и десериализации сообщения, уникальны для каждого поля в пределах одного сообщения, и не меняются со временем. Таким образом, если необходимо поменять название какого-то поля в одной из копий proto-файла, то не придётся менять название того же поля в остальных копиях этого proto-файла, поскольку название будет ассоциировано с номером поля. Это полезно при командной разработке какого-либо проекта.

После написания proto-файла необходимо с помощью утилиты protoc создать нужный код gRPC-сервиса. Для этого была прописана команда, приведённая на рисунке 23:

```
$ protoc --go_out=. --go_opt=paths=source_relative \
    --go-grpc_out=. --go-grpc_opt=paths=source_relative \
    helloworld/helloworld.proto
```

Рисунок 23 – Команда для формирования кода gRPC

После этого создалось два файла: server.pb.go и server_grpc.pb.go. Первый из них содержит определения структур данных и методов, которые используются для сериализации и десериализации сообщений в формате

Protocol Buffers, а второй - код, который позволяет создавать gRPC-сервер и клиент на основе определённых сервисов. Используя данные файлы и библиотеки, работающие с gRPC и являющихся специфичными для каждого языка программирования, можно писать код, работающий с данной технологией.

8.2.5 Микросервис для работы с конференциями

Данный компонент предоставляет API для работы с конференциями. Написан на языке Python, веб-фреймворк – Django.

На рисунке 24 представлена файловая структура данного компонента.

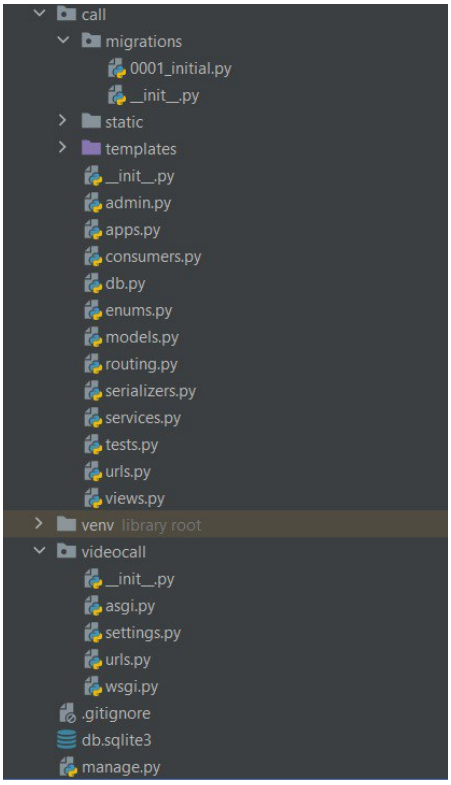

Рисунок 24 – Файловая структура компонента

Из основных файлов стоит отметить следующие:

1) consumers.py – файл, содержащий класс, который отвечает за работу с Websocket-соединениями;

2) db.py – файл для работы с базой данных;

3) models.py – файл, описывающий модели базы данных в виде классов Python;

4) routing.py – файл, описывающий роутинг для протокола Websocket;

5) services.py – файл, описывающий сервисный слой компонента;

6) views.py - файл, описывающий слой обработчиков компонента.

Данный компонент предоставляет возможность взаимодействовать с ним по протоколам HTTP и Websocket. Для реализации общения по HTTP используются обычные обработчики, зарегистрированные в файле urls.py, а для реализации общения по Websocket – библиотека django_channels, а конкретно – файл consumers.py. В этом файле описан класс CallConsumer, который схож с классом WebsocketManager микросервиса работы с командами. Имеются следующие методы:

1) метод connect, который регистрирует очередное подключение по протоколу Websocket;

2) метод disconnect, который является слушателем, вызывающимся при событии отключения определённого клиента;

3) метод receive, который вызывается на данном компоненте при вызове метода отправки Websocket-сообщения на стороне клиента, и который обрабатывает эти сообщения.

Отдельно необходимо выделить тот факт, что этот компонент является сигнальным сервером для передачи служебных сообщений согласно технологии WebRTC. Следующие 3 метода данного класса используются согласно технологии WebRTC:

1) метод call_received – вызывается в момент, когда какой-либо клиент A отправляет предложение (offer) другому клиенту B для дальнейшей передачи медиапотоков между собой;

2) метод call_answered – вызывается в момент, когда клиент B выразил согласие на передачу медиапотоков с клиентом A;

3) метод ICEcandidate – вызывается при появлении нового ICE-кандидата.

Схема установления WebRTC-соединения и передачи медиапотоков следующая. При переходе на страницу конференции текущий клиент делает предложение (offer) каждому клиенту, находящемуся в данной конференции. Для наглядности достаточно рассмотреть одну пару клиентов. У каждой пары клиентов A и B есть такие свойства, как localDescription и remoteDescription, которые хранят в себе значения, формируемые на этапе сигнализации технологии WebRTC. Для клиента A свойство localDescription хранит предложение, которое формируется изначально по протоколу SDP, а свойство remoteDescription – ответ от клиента B, который так же имеет некоторые значения и формируется аналогично предложению, отправленному клиентом A. Для клиента B аналогично, но наоборот. Алгоритм установки WebRTC-соединения следующий.

1) клиент A в момент формирования предложения устанавливает в свойство localDescription само предложение;

2) затем клиент A посылает это предложение клиенту B через сигнальный сервер по протоколу Websocket;

3) на стороне сигнального сервера идёт передача этого предложения нужному клиенту через его Websocket-соединение;

4) клиент B устанавливает в свойство remoteDescription предложение, отправленное клиентом A, а в свойство localDescription – ответ, который он формирует;

5) клиент B отправляет данный ответ клиенту A через сигнальный сервер;

6) клиент A устанавливает в своё свойство remoteDescription сам ответ от клиента B.

После этого возможна передача медиапотоков между клиентами A и B.

9 Результаты работы

В ходе работы было разработано веб-приложение, позволяющее организовывать онлайн-конференции. Рассмотрим весь реализованный функционал данного приложения.

В приложении предоставляется возможность регистрации и авторизации пользователей. Для регистрации необходимо ввести псевдоним пользователя (username), его электронную почту (email) и пароль (password). Идентификация пользователя происходит по псевдониму и по почте. Пароль должен содержать не менее 8 символов. На рисунках 25 и 26 представлены пользовательские интерфейсы для регистрации и авторизации:

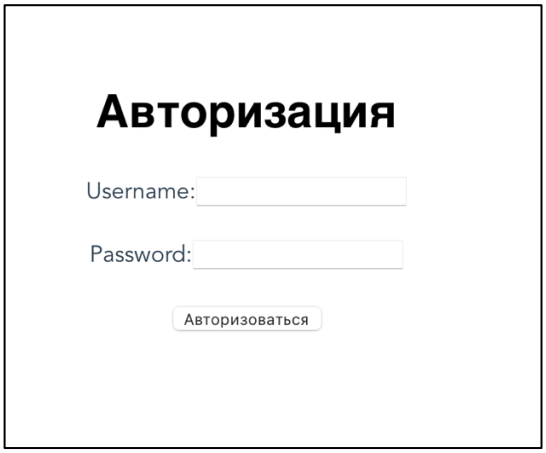

Рисунок 25 – Страница авторизации

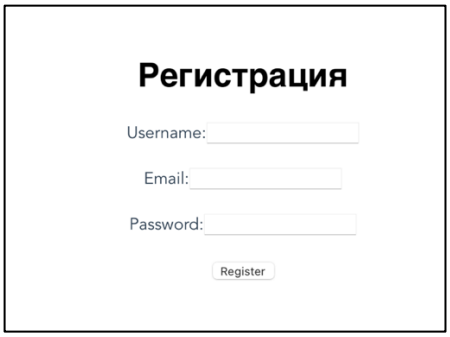

Рисунок 26 – Страница регистрации

При регистрации пользователя происходит запрос в базу данных на сохранение данных об этом пользователе, а при авторизации – извлечение из базы пароля пользователя для проверки правильности введённого пользователем пароля.

После того, как пользователь авторизовался, его перенаправляет на страницу с навигационной панелью, предоставляющей возможность переместиться к списку чатов и к списку команд пользователя. На странице списка команд расположена информация о пользователе в виде его псевдонима и почты, а также кнопки «Создать команду» и «Вступить в команду».

По нажатию на кнопку «Создать команду» пользователя перенаправляют на страницу создания команды, где можно ввести её название и её описание. На рисунке 27 представлена страница создания команды.

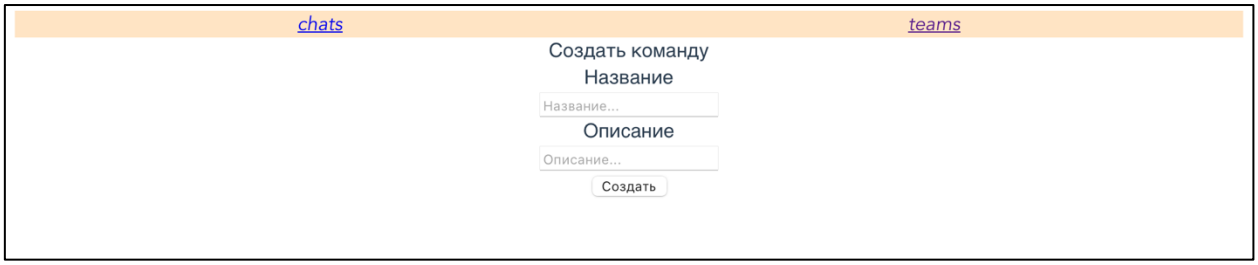

Рисунок 27 – Страница создания команды

По нажатию на кнопку «Вступить в команду» пользователя перенаправляют на страницу вступления в команду, где ему необходимо ввести так называемый код команды, который можно узнать у участников этой команды (рисунок 28). Таким образом, код команды играет роль ссылки на эту команду.

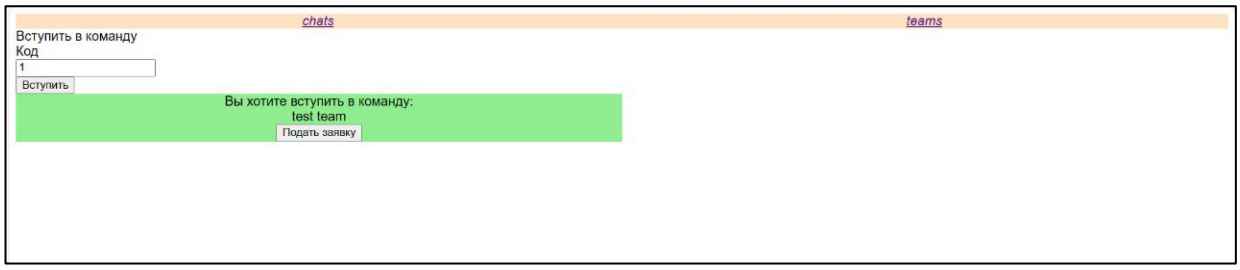

Рисунок 28 – Страница вступления в команду

После создания команды на странице списка команд пользователя появляется команда с введённым для неё названием при её создании. При нажатии на эту команду приложение перенаправляет пользователя на страницу самой команды. Данная страница представлена на рисунке 29:

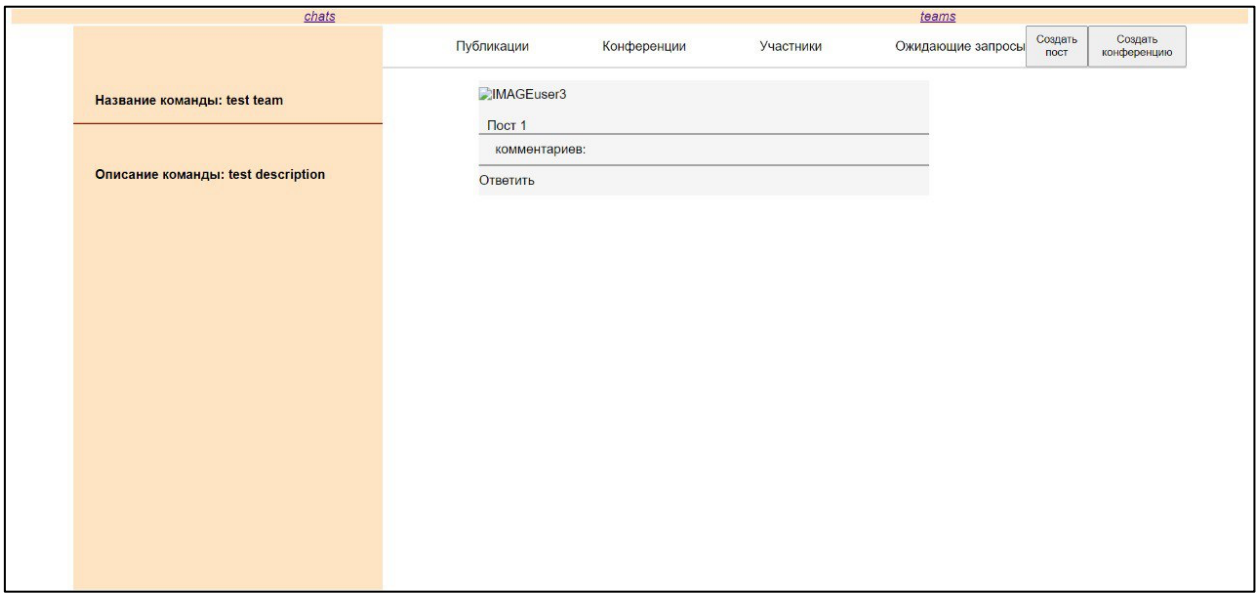

Рисунок 29 – Страница команды

Слева на странице расположено пространство для названия и описания команды, а справа – пространство для различных списков, в зависимости от перехода на ту или иную вкладку. При переходе на вкладку «Публикации» пользователю отобразятся все посты, написанные участниками данной команды. Пост содержит псевдоним его автора, текстовое содержание, количество комментариев к данному посту и кнопку «Ответить» для создания комментария к данному посту. Имеется возможность создания,

редактирования и удаления постов. На рисунках 30 и 31 представлены страницы создания и редактирования постов:

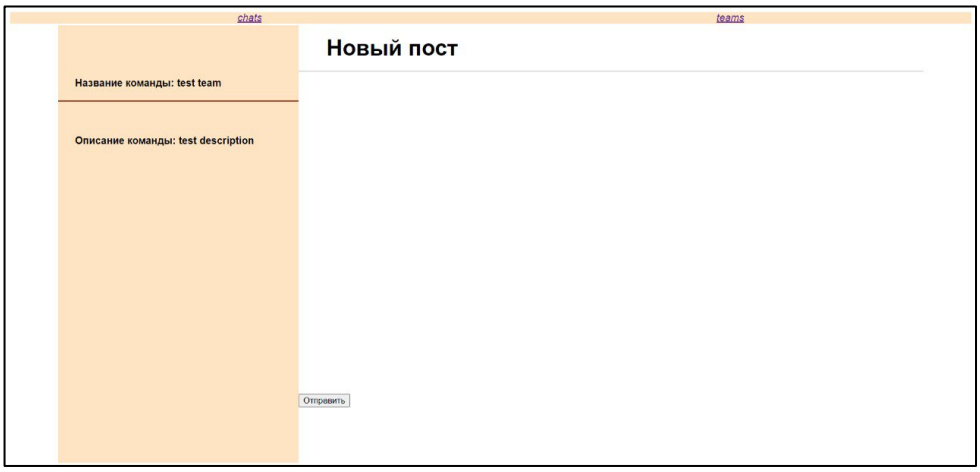

Рисунок 30 – Страница создания поста

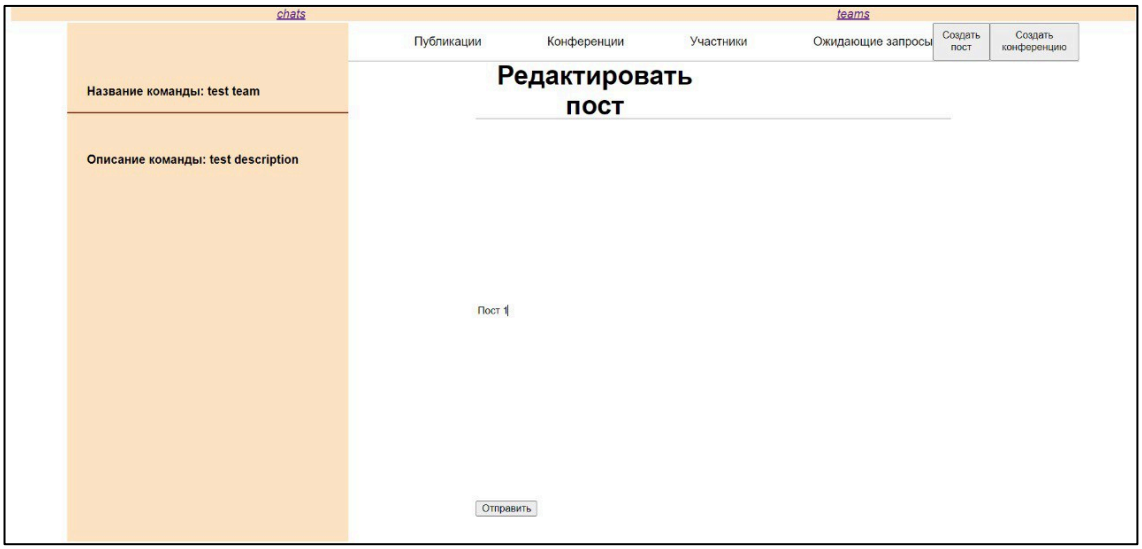

Рисунок 31 – Страница редактирования поста

Создание поста осуществляется по кнопке «Создать пост», а редактирование – по нажатию на данный пост.

По нажатию на вкладку «Участники» (рисунок 32) пользователю отобразятся участники данной команды с учётом их ролей. Ролей всего две: роль администратора и роль простого участника. Если пользователь является администратором данной команды, то ему также отображается возможность

удаления данного пользователя и изменения его роли. Также имеется возможность добавления участника по его электронной почте. После введения почты нужного пользователя и нажатия кнопки «Добавить в команду» этот пользователь будет добавлен в команду.

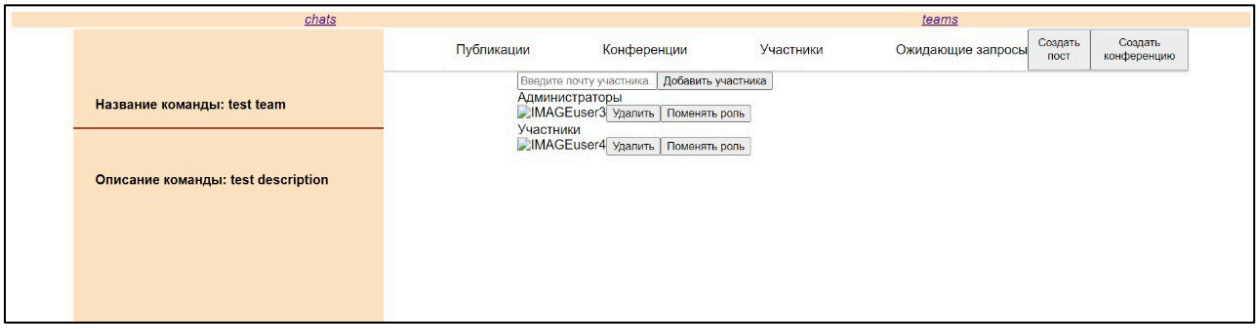

Рисунок 32 – Вкладка «Участники»

По нажатию на вкладку «Ожидающие запросы» (рисунок 33) пользователю отобразится список ожидающих запросов, где каждый элемент списка будет содержать псевдоним пользователя, его почту, а также при наличии у участника команды привилегии администратора – две кнопки: принять и отклонить. Первая кнопка принимает запрос на вступление в команду, вторая – отклоняет. Ожидающий запрос создаёт пользователь, желающий вступить в данную команду. Такой запрос можно создать по рассмотренной выше кнопке «Вступить в команду» на странице списка команд пользователя.

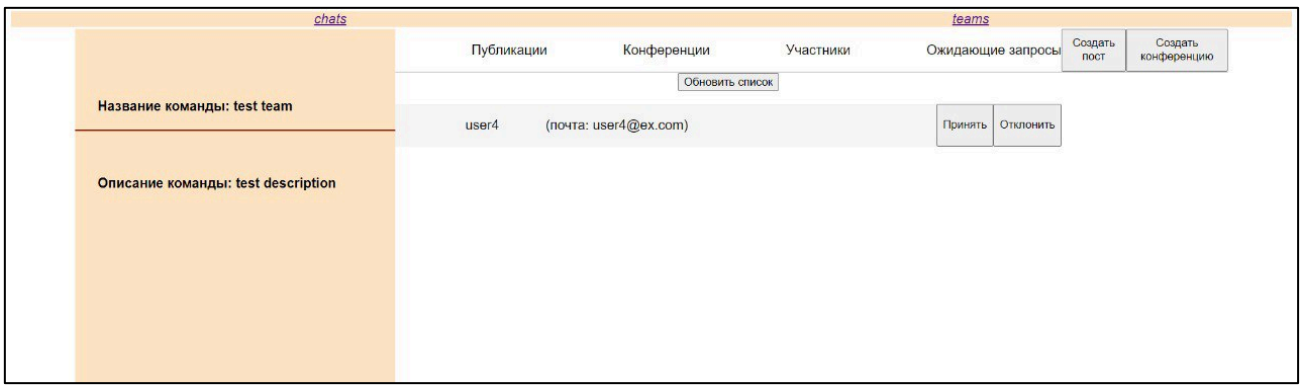

Рисунок 33 – Вкладка «Ожидающие запросы»

По нажатию на вкладку «Конференции» (рисунок 34) пользователю отобразится список конференций данной команды. Каждый элемент этого списка содержит информацию об очередной конференции в виде её названия.

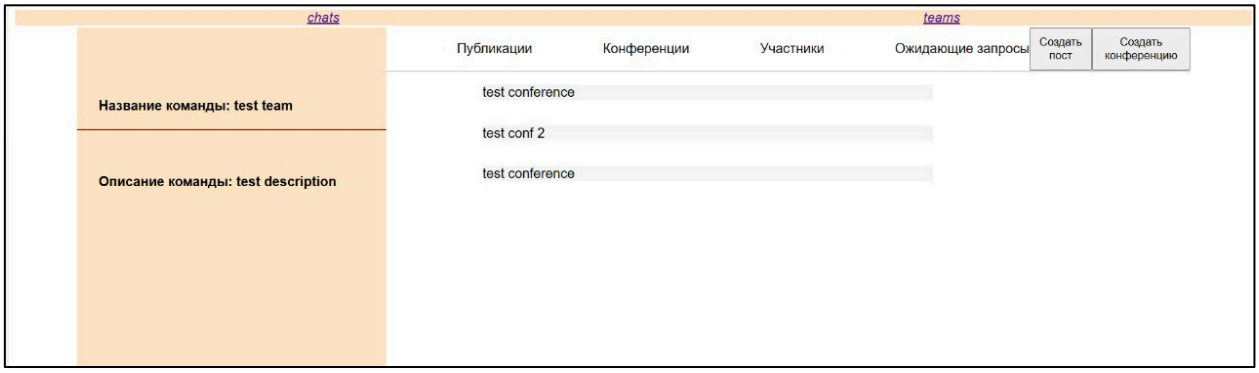

Рисунок 34 – Вкладка «Конференции»

Имеется возможность создания конференции по нажатию на кнопку «Создать конференцию». По нажатию на эту кнопку пользователя перенаправляет на страницу создания конференции. На рисунке 35 представлен пользовательский интерфейс для данного функционала:

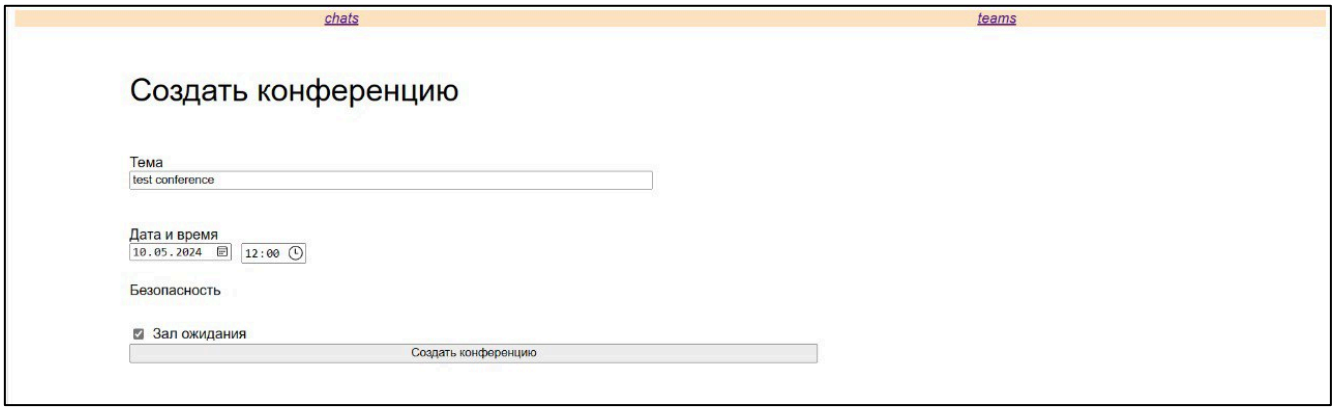

Рисунок 35 – Страница создания конференции

Пользователь может ввести тему конференции, дату и время начала, а также выбрать, нужен ли зал ожидания для данной конференции. По нажатию на кнопку «Создать» пользователь автоматически становится её

администратором, а также происходит перенаправление пользователя на страницу конференции.

После того, как конференция создана, любой участник может к ней присоединиться. Необходимо перейти на вкладку «Конференции» в странице команды и выбрать искомую конференцию. Пользователей будет перенаправлять в зал ожидания (рисунок 36), где им предоставляется возможность проверки аудио- и видеоустройств.

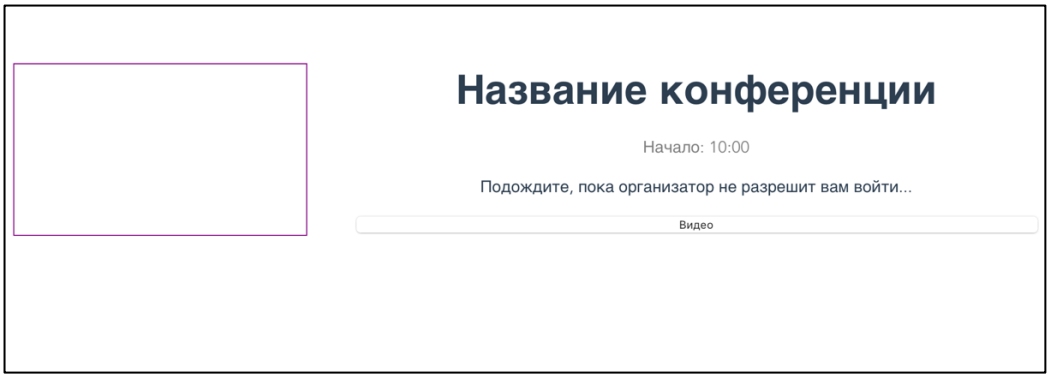

Рисунок 36 – Страница зала ожидания

На странице конференции (рисунок 37) есть возможность демонстрации экрана и просмотра списка участников данной конференции с учётом ролей участников. Ролей в конференции так же две: роль администратора и роль обычного участника. Также есть список пользователей, находящихся в зале ожидания.

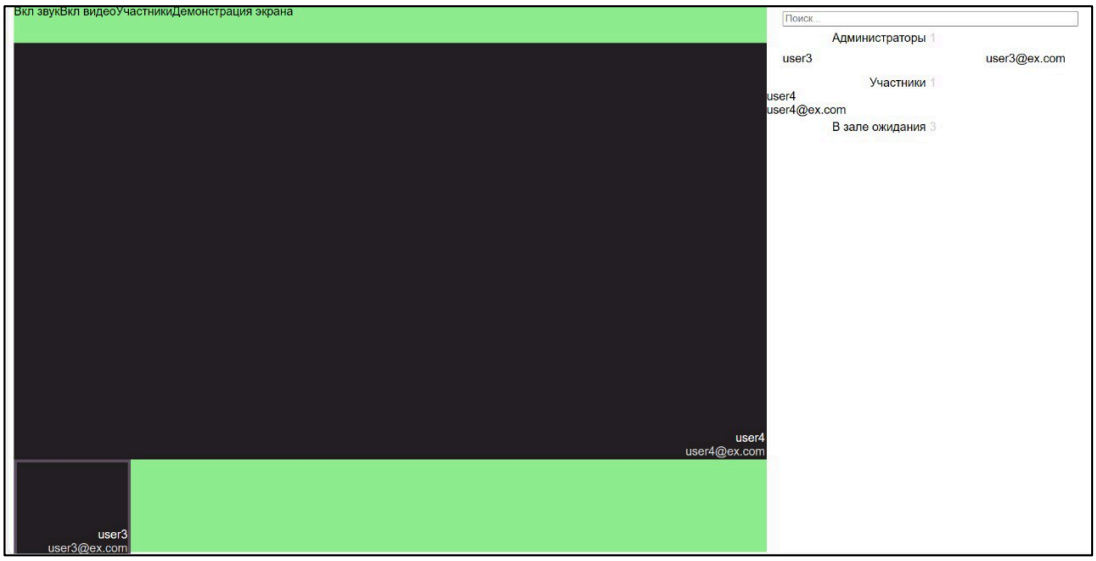

Рисунок 37 – Страница конференции

Цифры светло-серого цвета обозначают количество людей в данном списке. Также из рисунка видно, что для пользователей отведено место для потенциального отображения их видео, а также информация о них в виде псевдонима и электронной почты.

Помимо функционала, связанного с командами, пользователю предоставляется функционал, связанный с возможностью переписываться с нужными ему другими пользователями. При переходе на вкладку “chats” (рисунок 38) в навигационной панели пользователя перенаправляет на страницу списка чатов. На ней имеется возможность создания чата и получения списка чатов.

Рисунок 38 – Страница списка чатов пользователя

Каждый элемент списка чатов состоит из названия этого чата. При переходе на конкретный чат пользователя перенаправляет на страницу переписки (рисунок 39).

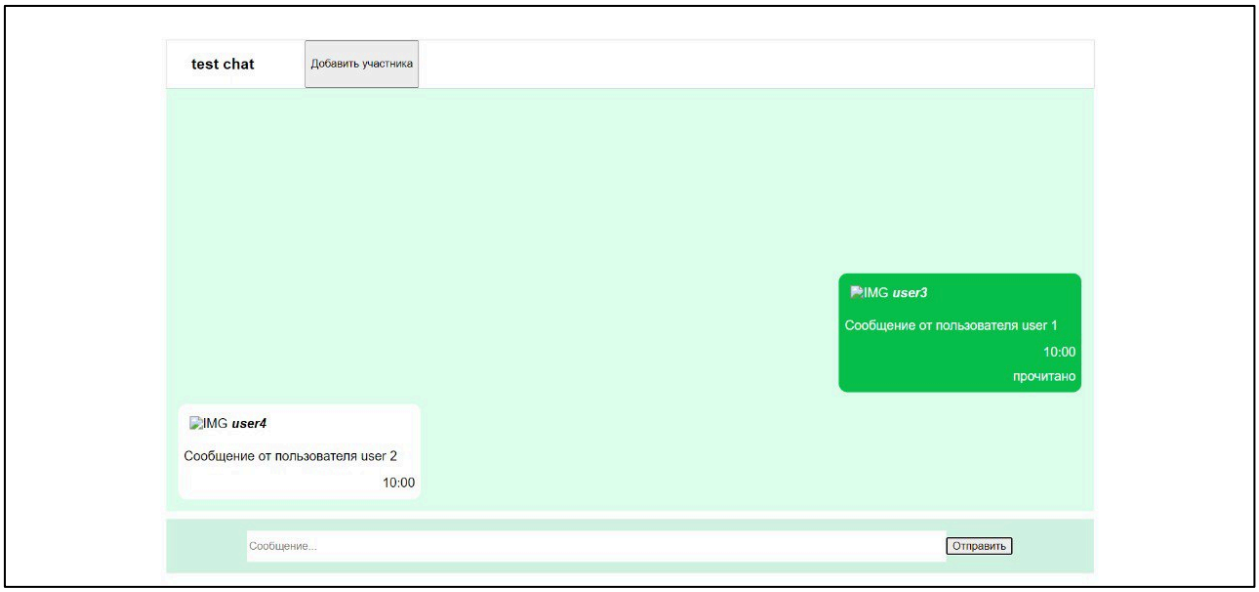

Рисунок 39 – Страница чата

На ней имеется возможность добавления участника по его почте, а также отправка самих сообщений в текстовом виде. После добавления другого пользователя в данный чат можно вести с ним переписку. Сообщения отправляются друг другу в режиме реального времени по протоколу Websocket. Также имеются возможность отслеживания статуса прочтения данного сообщения и редактирования сообщения пользователем при условии, что он создал это сообщение. При редактировании происходит перенос текущего содержимого сообщения в поле ввода, где можно изменить содержание данного сообщения.

ЗАКЛЮЧЕНИЕ

Цель работы достигнута, изучены концепции построения микросервисной архитектуры, технология обмена медиапотоками WebRTC, протокол Websocket, СУБД Redis для кеширования данных, Vue JS и Golang. Было спроектировано и разработано веб-приложение для организации онлайн-конференции и командной работы. Клиентская часть приложения была написана на Vue JS, серверная – на Java, Python, Golang. Серверная часть состоит из микросервисов для работы с пользователями, чатами, сообщениями, командами и конференциями, а также из единой точки входа в приложение Gateway.

В разработанном приложении представляются возможности работы с командами, чатами и конференциями. Пользователи могут создавать команды, вступать в них, добавлять новых участников. Администраторы команд имеют возможность менять роли участников, что позволяет разграничивать права доступа для команд. Приложение позволяет создавать, изменять, удалять посты и получать их список, а также создавать, изменять и удалять комментарии к постам. Имеется возможность управлять участниками команд (получение списка участников, добавление и удаление), а также работать с ожидающими запросами. Ожидающие запросы представляют из себя запросы от пользователей, которые хотят вступить в определенную команду. Кроме того, приложение обеспечивает следующие возможности для работы с чатами: получение списка чатов для пользователя, создание чата, добавление пользователя в чат по его электронной почте, отправка сообщений, получение списка сообщений чата, отслеживание статуса того, что сообщение прочитано собеседником, редактирование сообщение в режиме реального времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ньюмен, С. Создание микросервисов. / С. Ньюмен. – Санкт-Петербург: Питер, 2016. – 304 с. – ISBN 978-5-496-02011-4

2. Микросервисы, скалы и гигантские приложения // VK Cloud: [сайт]. – URL: https://cloud.vk.com/blog/mikroservisy-skaly-i-gigantskie-prilozheniya#:~:text=Преимущества%3A,сервиса%20с%20SLA%2099%2C95%25 (дата обращения: 22.03.2024)

3. Реализация паттернов отказоустойчивости в микросервисах // AppMaster: [сайт]. – URL: https://appmaster.io/ru/blog/modeli-otkazoustoichivosti-arkhitektury-mikroservisov#pattern-rate-limiter (дата обращения: 22.03.2024)

4. Что такое WebRTC? // Труконф: [сайт]. – URL: https://trueconf.ru/webrtc.html (дата обращения: 10.04.2024)

5. WebRTC для всех и каждого. Часть 1 // Хабр: [сайт]. – URL: https://habr.com/ru/companies/timeweb/articles/656947/ (дата обращения: 10.04.2024)

6. Касун, И. gRPC: запуск и эксплуатация облачных приложений. / И. Касун, У. Данеш – СПб.: Питер, 2021. – 224 с. – ISBN 978-5-4461-1737-6 (дата обращения: 10.04.2024)

7. Что такое GRPC? // Yandex Cloud: [сайт]. – URL: https://yandex.cloud/ru/docs/glossary/grpc?utm_referrer=https%3A%2F%2Fwww.google.com%2F (дата обращения: 10.04.2024)

8. gRPC // Хабр: [сайт]. – URL: https://habr.com/ru/companies/otus/articles/780720/ (дата обращения: 10.04.2024)

9. Как событийно-ориентированная архитектура решает проблемы современных веб-приложений // Хабр: [сайт]. – URL: https://habr.com/ru/companies/piter/articles/530514/ (дата обращения: 10.04.2024)

10. Событийно-ориентированная архитектура // Википедия: [сайт]. – URL: https://ru.wikipedia.org/wiki/Событийно-ориентированная_архитектура (дата обращения: 10.04.2024)

11. Что такое Kubernetes? Знакомимся с популярной платформой контейнерной оркестрации // CROC Cloud Services: [сайт]. – URL: https://cloud.croc.ru/blog/about-technologies/chto-takoe-kubernetes/ (дата обращения: 10.04.2024)

12. Основы Kubernetes // Хабр: [сайт]. – URL: https://habr.com/ru/articles/258443/ (дата обращения: 10.04.2024)