МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Ігор ШЕЛЕХОВ
_____
(підпис)
червня 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавр

зі спеціальності 122 – Комп'ютерних наук,
освітньо-професійної програми «Інформатика»
на тему: «Веб-додаток для організації відеоконференцій»
здобувача групи ІН-94-1 Фоменка Ростислава Олександровича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Ростислав ФОМЕНКО
_____
(підпис)

Керівник,
асистент кафедри комп'ютерних наук,
кандидат фізико-математичних наук Олександр ВЛАСЕНКО _____
(підпис)

Суми – 2023

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 – Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН-94-1 Фоменка Ростислава Олександровича

1. Тема роботи: «Веб-додаток для організації відеоконференцій»
затверджую наказом по СумДУ від *«01» червня 2023 р. № 0475-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 09 червня 2023 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Огляд технологій, що використовуються для організації відеоконференцій. 3) Розробка веб-додатку для організації відеоконференцій у режимі реального часу. 4) Аналіз результатів.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 2023 р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
2	*Огляд технологій, що використовуються для розробки веб-додатків*		
3	*Розробка веб-додатку для відеоконференцій*		
4	*Тестування та аналіз отриманих результатів*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ Керівник _____

(підпис) (підпис)

АНОТАЦІЯ

Записка: 82 стр., 28 рис., 3 додатка, 15 використаних джерел.

Обґрунтування актуальності теми роботи – Тема кваліфікаційної роботи є актуальною, оскільки розв'язує завдання дистанційної роботи, навчання та загалом відео спілкування як такого. Ідея подібних програмних продуктів у наш час і в майбутньому ще дуже довго буде затребувана через світову глобалізацію та нові можливості в різних галузях, де необхідне спілкування на відстані.

Об’єкт дослідження – Передача медіаданих у режимі реального часу.

Мета роботи – розробка веб-додатку для організації відеоконференції шляхом передачі медіа даних, отриманих із зовнішніх пристроїв клієнта, в режимі реального часу.

Методи дослідження – розбір документацій до низькорівневих протоколів та побудова тестових спрощених моделей для передачі медіа даних у режимі реального часу.

Результати – розроблено веб-додаток для організації відеоконференцій в режимі реального часу. Веб-додаток має такі частини, як адаптивний дизайн, сигнальний протокол у вигляді сервера для встановлення з'єднання та взаємодії з функціями медіа-сервера, а також масштабоване та швидке API на стороні сервера для взаємодії з базою даних та логікою додатка.

ВЕБ-ДОДАТОК, API, МЕДІА-СЕРВЕР, WebRTC

ЗМІСТ

ВСТУП

Актуальність. У сучасному світі зростає популярність відеоконференцій як засобу комунікації, особливо в контексті роботи з віддаленими командами, дистанційного навчання та зв'язку між віддаленими користувачами. Велике значення набуває створення ефективних інструментів для організації відеоконференцій, що забезпечують зручність, надійність та якісне відтворення аудіо- та відеоінформації.

Об’єкт дослідження. Процес дослідження є розробка веб-додатку для організації відеоконференцій. Даний додаток буде забезпечувати можливість проведення відеоконференцій у режимі реального часу через інтернет, з використанням веб-технологій.

Предмет дослідження. Аналіз процесу організації відеоконференцій у реальному часі та забезпечення стабільного та ефективного з'єднання між учасниками. Дослідження різних аспектів, пов'язаних з встановленням та підтримкою стабільного з'єднання між учасниками відеоконференції. Предметом дослідження є технології, протоколи, алгоритми та інфраструктура, що використовуються для забезпечення відеозв'язку між учасниками.

Гіпотеза. Припускаємо, що розроблений веб-додаток зможе забезпечити ефективну організацію відеоконференцій, забезпечуючи високу якість передачі мультимедійних даних та зручний інтерфейс для користувачів.

Новизна. Розробка веб-додатку для організації відеоконференцій має актуальну новизну, оскільки поєднує в собі сучасні веб-технології та вимоги до забезпечення якості мультимедійної передачі. Представлена демонстраційна версія додатку орієнтована на потреби віддалених команд, освітніх закладів та інших сфер діяльності, де відеоконференції є важливим засобом комунікації.

Структура. Дана робота складається зі вступу, аналітичного огляду, постановки задачі, вибору методу розв'язання поставленої задачі, опису

програмного забезпечення інформаційної системи, висновків, списку використаних джерел та додатків.

1 АНАЛІТИЧНИЙ ОГЛЯД

1.1 Сучасний стан

Кваліфікаційна робота присвячена темі розробки відеоконференцій. Відеоконференція – це телекомунікаційна технологія передачі даних у форматах відео та аудіо. За допомогою відеоконференцій учасники бачать та чують один одного в режимі реального часу, а також можуть переписуватись та обмінюватись файлами під час сеансу відеозв'язку. Технологія відеоконференц-зв'язку стирає обмеження, зумовлені відстанню. Системи відеоконференцій дозволяють повноцінно спілкуватися з колегами, віддаленими співробітниками та друзями з будь-якої точки світу. Тому сьогодні групові відеоконференції, семінари та онлайн-наради з відео стали гарною альтернативою традиційним живим зустрічам.

Справжній "бум" у потребі продовжувати будь-яку діяльність, перебуваючи на безпечній відстані від партнера, стався у розпал пандемії коронавірусу COVID-19. У цей період багато корпорацій, які лідують у сфері надання послуг дистанційного спілкування, відчули на собі небувалий потік клієнтських запитів.

Якщо поринути в історію, то можна помітити, що ще в першій половині минулого століття, а саме в 1927 інженери дослідницького центру AT&T Bell Telephone Laboratories (згодом Bell Labs) зробили прорив у створенні першої в історії телевізійного зв'язку. Президент США Герберт Гувер, який випробував це рішення, майбутній на той час, звернувся до журналістів, передавши зображення і мову на відстані понад 320 кілометрів. Нюанс був у тому, що зв'язок був абсолютно одностороннім. Перший повноцінний варіант двосторонньої аудіовізуальної телекомунікаційної системи був представлений компанією AT&T лише через три роки — саме 1930-й можна вважати роком народження відеозв'язку у його сучасному розумінні.

З появою комп'ютерів та широкого розповсюдження Інтернету з'явилися перші спроби створення систем відеоконференцій. У 1990-х роках були

розроблені програмні засоби, які дозволяли передавати відео- та аудіо сигнали через мережу Інтернет. Проте на той час таке з'єднання було досить повільними, а якість відео і звуку була недостатньою для задоволення потреб користувачів. З появою швидкого Інтернету і вдосконаленням технологій стиснення відео та аудіо, видеоконференції стали більш доступними та популярними. У 2000-х роках з'явилися перші веб-камери та програми для відеодзвінків, які дозволяли користувачам спілкуватися за допомогою відеозв'язку. Однак ці системи мали свої недоліки, адже можливості передачі великої кількості даних були обмеженими.

Сучасні програми для відеоконференцій — це інтерактивні рішення для роботи онлайн. Сервіси організації відеоконференцій паралельно проводять прийом та передачу різних потоків інформації. Нинішній стан розвитку інтернет-технологій та комунікацій дозволяють робити та розгортати багато процесів для якісних та зручних дзвінків. Від клієнта потрібно мати лише пристрій або набір пристроїв для отримання і передачі необхідної фізичної інформації, а також щіпка часу на вивчення інструкцій для взаємодії з програмним інтерфейсом.

Розглянемо деякі з основних причин, чому відеоконференції є важливим інструментом і як вони продовжують розвиватися.

- Зв'язок та спілкування: Відеоконференції забезпечують зв'язок між людьми, незалежно від їх географічного розташування. Вони дозволяють проводити зустрічі, засідання, презентації та спілкуватися з колегами, партнерами, клієнтами та іншими учасниками на відстані. Це зберігає час і кошти, які були б витрачені на подорожі, та робить комунікацію більш ефективною.

- Віддалена робота та телекомутація: Відеоконференції стали необхідним інструментом для віддаленої роботи та телекомунікації. Вони дозволяють співробітникам працювати з будь-якого місця, зберігаючи зв'язок

з командою та клієнтами. Це підвищує продуктивність, забезпечує гнучкість та збільшує рівень задоволення працівників.

- Освіта та навчання на відстані: Відеоконференції змінили парадигму освіти, дозволяючи студентам та викладачам зв'язуватися та проводити уроки на відстані. Вони розширюють доступ до навчання, дозволяючи отримувати знання незалежно від місця проживання. Крім того, відеоконференції допомагають організовувати вебінари, тренінги та інші форми професійного розвитку.

- Медицина: Відеоконференції забезпечують доступ до медичних консультацій та діагностики на відстані. Це особливо корисно для пацієнтів, які проживають у віддалених районах або не можуть фізично відвідати лікаря. Телемедицина здатна зберегти життя та покращити доступ до якісної медичної допомоги.

- Подальший розвиток: Відеоконференції продовжують розвиватися та вдосконалюватися. З'являються нові функції, які поліпшують користувальницький досвід, такі як віртуальна реальність, додаткові можливості спілкування та інтеграція з іншими інструментами. Технології штучного інтелекту, розпізнавання голосу та обробки зображень також впливають на подальший розвиток відеоконференцій.Подальший розвиток: Відеоконференції продовжують розвиватися та вдосконалюватися. З'являються нові функції, які поліпшують користувальницький досвід, такі як віртуальна реальність, додаткові можливості спілкування та інтеграція з іншими інструментами. Технології штучного інтелекту, розпізнавання голосу та обробки зображень також впливають на подальший розвиток відеоконференцій.

Усе це робить відеоконференції надзвичайно важливими для нашого сучасного світу. Вони забезпечують зв'язок, спілкування та співпрацю на відстані, полегшують наші робочі та особисті життя, та відкривають нові можливості для розвитку в різних галузях людської діяльності.

1.2 Аналіз аналогічних проєктів

Будь-які сучасні програми зараз йдуть в одну ногу не тільки з апаратними, але й програмними технологіями для досягнення найбільшої вигоди в бізнесі та залучення таким чином більшої уваги клієнтів. Сфера відеоконференцій має дуже великий попит у наш час через що вимагала до себе увагу великих компаній, що володіють колосальними ресурсами для покриття потреб практично всіх зацікавлених у спілкуванні на відстані клієнтів. Такими провідними додатками зараз є Google Meet та Zoom. Кожен перелічений продукт має ряд особливостей як позитивних так і негативних. Це дає можливість проаналізувати і оцінити їх для того, щоб впровадити кращі практики при реалізації своєї відеоконференції.

Google Meet базується на клієнт-серверній архітектурі. Основні компоненти системи включають клієнтські програми, якими є веб-додатки, мобільні додатки для Android і iOS, які в свою чергу інтегруються з екосистемою Google та серверну інфраструктуру, яка обробляє відео- і аудіо потоки, керує мережевими з'єднаннями та забезпечує функціональність відеоконференцій. Він використовує різні кодеки для стиснення та передачі медіа даних. Зазвичай використовуються кодеки, такі як VP9 та H.264 для відео та Opus для аудіо. Потоки передаються через мережеве з'єднання між клієнтами та серверами. За допомогою технологій WebRTC та механізмів розрахованих на багато користувачів у конференціях (MCU). Google Meet підтримує одночасний зв'язок між декількома клієнтами. Варто зазначити, що Google Meet використовує алгоритми управління мережевими ресурсами, які динамічно адаптуються до швидкості мережі, затримки та пропускної спроможності. Він також використовує шифрування даних у дорозі та спокої, а також включає функції, такі як захист від несанкціонованого доступу, автентифікація користувачів та контроль доступу до конференцій. Клієнтський інтерфейс надає низку додаткових функцій, таких як можливість обміну екраном, використання чату, спільна робота над документами та

презентаціями, запис сесій. Однак при всьому цьому варто пам'ятати, що функціонал обмежується на апаратних можливостях клієнта і пропускних здібностях інтернет-з'єднання.

Google Meet має обмеження на деяких платформах та браузерах. Не всі клієнти можуть підтримувати такі технології, як WebRTC. Варто зазначити, що з технічної точки зору різні додатки для організації відеоконференцій намагаються будувати свої відмінні протоколи та механізми для цього.

Zoom має практично такі самі технічні переваги і недоліки як і попередній продукт. Але деякі з них варто виділити детальніше. Zoom не такий гігант як Google Meet, але при цьому він відмінно справляється з великою кількістю навантаження, тому працює досить стабільно. Zoom медіа-сервер дозволяє змінювати потоки під час пересилання , наприклад, змінюючи фон зображення, що передається. Права доступу в конференції дають можливість призначати адміністраторів, тим самим оперуючи різними рівнями привілеїв, змінювати доступність медіа потоків учасників, а саме включати та вимикати мікрофон, а також вимикати відео або вимагати включення відео у всіх учасників сесії. Демонстрацію екрану можна ставити на паузу, крім того є можливість, як і в Google Meet, ділитися не всім екраном, а лише окремими програмами, наприклад, включати демонстрацію браузера. У налаштуваннях можна дати всім учасникам можливість ділитися екранами або включити обмеження, щоб робити це міг лише організатор. Обидва продукти, оскільки використовують медіасервер між учасниками, можуть робити записи відеоконференцій, оскільки отримують абсолютно кожен пакет даних від клієнтів. Однак варто відзначити, що більшість користувачів скаржаться на незручний інтерфейс Zoom, що так само варто враховувати під час розробки власної організації потокового спілкування.

Враховуючи все проаналізоване можна відзначити, що сучасний додаток для відеоконференцій має бути доступним для всіх і при цьому мати якнайменше обмежень. Він має забезпечувати безпечний і конфіденційний зв'язок з безліччю клієнтів і уникати споживання зайвої кількості клієнтських

апаратних ресурсів.

1.3 Постановка задачі

Метою для розробки веб-додатків є з'єднати двох клієнтів потоковим відеозв'язком у режимі реального часу та надати можливість користувачеві оперувати логічно пов'язаними даними додатку.

Саме тому для досягнення поставленої мети необхідно вирішити наступні задачі:

1) Створити сигнальний протокол під потреби додатка та медіа-сервера;

2) Зібрати клієнтську програму з розробленим адаптивним дизайном для демонстрації можливостей відеоконференц зв'язку;

3) Написати масштабовану серверну частину для взаємодії з сутностями бази даних;

2 ВИБІР МЕТОДУ РОЗВ’ЯЗАННЯ ЗАДАЧІ

2.1 Інформаційна модель

2.1.1 Передача потоку медіаданих

Передача медіаданих – це процес передачі аудіо-, відео- або зображення з одного пристрою на інший за допомогою комунікаційного каналу, такого як мережа Інтернет, супутниковий зв'язок або локальна мережа.

Як і вся комп'ютерна інформація медіа передаються у вигляді бітів – нулів та одиниць. Нуль чи один є абстракцією над станом, де є струм або де його немає. Для передачі таких бітів, що б один пристрій обмінювався інформацією точно так само, як і інший, необхідний певний стандарт або як його ще називають – комунікаційний протокол, що являє собою обумовлені наперед правила передачі даних між двома пристроями. До основних параметрів, які описує протокол, відносяться: тип перевірки помилок, що використовується метод компресії інформації спосіб визначення передаючим пристроєм завершення передачі. У моделі OSI (абстрактна мережева модель для комунікацій і розроблення мережевих протоколів) на транспортному рівні для того самого процесу використовуються два основні протоколи: TCP та UDP.

Протокол TCP (Transmission Control Protocol) – це один з основних протоколів передачі даних в комп'ютерних мережах. Він забезпечує надійну та упорядковану передачу даних між вузлами (комп'ютерами) в мережі Інтернет. Використовується у випадках, коли потрібна надійна доставка повідомлень. Він звільняє прикладні процеси від необхідності використовувати таймаути та повторні передачі для забезпечення надійності. Це забезпечується шляхом використання механізмів підтвердження отримання пакетів (відформатований блок бітів), виявлення та повторної передачі в разі втрати або пошкодження пакетів. TCP також зберігає послідовність передачі даних, тобто дані надходять до приймача в тому ж порядку, в якому вони були надіслані від відправника і регулює передачу

даних, щоб уникнути переповнення буферів. Для встановлення з'єднання протокол TCP проводить процес "рукостискання", очікуючи від сторони-приймача явне підтвердження.

У свою чергу UDP (User Datagram Protocol) забезпечує безз'єднані передачі даних між вузлами, тобто не встановлює постійного з'єднання між відправником і приймачем. Кожен пакет (у цьому випадку датаграмами – простий блок інформації) даних відправляється окремо та незалежно від інших пакетів. Цей протокол не надає гарантії доставки пакетів або впорядкованості їх надходження. Це означає, що пакети можуть бути втрачені, дублюватися або надходити в іншому порядку. Через те, що UDP не має механізмів контролю та перевірки, він має швидкість у використанні.

Різниця помітна, через що не важко зробити висновок, що передачу медіаконтенту краще заводити через UDP, оскільки приріст у швидкості буде суттєвим і втрачені кадри тут не відіграють великої ролі.

Треба зауважити, що термін "потік даних" відноситься до послідовного потоку бітів або байтів, які передаються або отримуються в процесі комунікації або обробки даних. Він представляє собою неперервний потік інформації, що переміщується з одного місця в інше. Потік даних може мати різні джерела та призначення, і він може бути переданий через різні канали зв'язку, такі як мережі передачі даних або фізичні з'єднання. Він може представляти собою текст, звук, відео, графіку або будь-який інший тип даних.

З самого початку процеса передачі даних треба встановити з'єднання, якщо потрібна їх цілісність. Перед відправкою бітів дані розбиваються на пакети для значного зменшення їхньої ваги та подальшого легкого транспортування. Один такий пакет загортається у сегмент транспортного протоколу. Він використовує номера портів для ідентифікації відправника та отримувача. Сегменти передаються через IP-протокол, який використовує IP-адреси для маршрутизації пакетів по мережі. Під час передачі пакети маршрутизуються по мережі від відправника до одержувача. Це відбувається за допомогою маршрутизаторів, які приймають рішення про найкоротший

шлях для доставки пакетів на основі IP-адреси призначення. Залежно від транспортного протоколу, що використовується, може знадобитися закрити з'єднання.

У самому фундаменті моделі передачі даних по мережі Інтернет знаходиться фізична вистава нашого неідеального світу, що може стати причиною втрати кадрів. Починаючи від матеріальних властивостей передавача, закінчуючи спотворенням сигналу. На рівнях вище також чимало причин, з яких можуть губитися кадри. По-перше, перевантаження мережі. Пакети можуть бути втрачені, якщо маршрутизатори неправильно маршрутизують їх або вибирають неоптимальний шлях для доставки. Це може статися через несправність маршрутизатора або помилки в маршрутизаційних таблицях. По-друге, у деяких випадках кадри можуть бути збережені в буфері пристрою, наприклад, маршрутизатора, якщо буфер переповнений або його потужності не вистачає (зв’язку вже забитий). В результаті, деякі кадри можуть бути втрачені або відкинуті з буфера. Вищерозглянутий транспортний протокол TCP за рахунок своїх властивостей допомагає уникнути багатьох таких проблем.

Такий ефект як "джиттер" [8] буває частою небажаною подією під час передачі сигналу. Він виникає через непрогнозовану затримку під час передачі даних по мережі. Наприклад, пакети можуть перебувати в черзі для передачі або зазнавати затримок через недоліки у мережі. Це може призвести до нерівномірного потоку даних при отриманні. Для компенсації джиттеру використовується джиттер-буфер. Приймач збирає прийняті пакети і тимчасово затримує їх перед відтворенням. Інформація в буфері використовується для вирівнювання затримки і регулювання швидкості відтворення, щоб уникнути перебоїв або перекосів у відтворенні. Він також часто використовується для того, щоб повторно запросити interframe

2.1.2 Вигляд медіаданих

Медіа дані мають свій певний вид по дорозі до одержувача. Отримана інформація, яка повинна бути передана як відео або аудіо в режимі реального часу, має свій вигляд як RTP протокол прикладного рівня. Протокол RTP забезпечує послідовну і передбачувану доставку мультимедійних даних з важливістю на збереження плинності та мінімізацію затримок. Він додає до мультимедійних даних додатковий RTP-заголовок, який містить інформацію про час, послідовність та синхронізацію. Протокол RTP переносить у своєму заголовку дані, необхідні для відновлення голосу та відео на приймальному вузлі, а також дані про тип кодування інформації. В заголовку цього протоколу, зокрема, передаються мітка і номер пакету. Ці параметри дозволяють при мінімальних затримках визначити порядок і час декодування кожного пакета, а також інтерполювати втрачені пакети.

Компоненти протоколу:

- Сам протокол RTP.
- Так як RTP базується на UDP, він часто використовується разом з протоколом RTCP (Real-Time Control Protocol), який відповідає за передачу контрольної інформації про якість обслуговування, статистику, синхронізацію та інші параметри мультимедійного потоку.
- Керуючий сигнальний протокол. Сигнальні протоколи керують відкриттям, модифікацією та закриттям RTP-сесій між пристроями та додатками реального часу.
- Керуючий протоколом опису медіа, такий як Session Description Protocol.

Біти вмісту пакета (рис. 2.1) мають наступний діапазон і значення [4]:

0-1 – Ver. (2 біти) вказує версію протоколу. Поточна версія – 2.

2 – P (один біт) використовується у випадках, коли RTP-пакет доповнюється порожніми байтами на кінці.

3 – X (один біт) використовується для вказівки розширень протоколу, задіяних у пакеті.

4-7 – CC (4 біта) містить кількість CSRC-ідентифікаторів, наступних за постійним заголовком.

8 – M (один біт) використовується на рівні програми та визначається профілем. Якщо це поле встановлено, дані пакета мають якесь особливе значення для програми.

9-15 – PT (7 біт) вказує формат корисного навантаження та визначає її інтерпретацію додатком.

64-95 – SSRC вказує джерело синхронізації.

EHL (Extension Header Length) – кількість 32-бітних слів у блоці даних розширення заголовка.

L – останній байт у пакеті, що визначає довжину області заповнення в байтах (використовується для вирівнювання в останньому пакеті).

+ Біти	0-1	2	3	4-7	8	9-15	16-31
0	Бачити.	П	X	CC	M	PT	Порядковий номер
32	Мітка часу						
64	SSRC-ідентифікатор						
96, якщо CC>0	[CSRC-ID]						
96+(CC×32), якщо X=1	[Заголовок розширення – певне профілем значення]						[Заголовок розширення – кількість блоків даних по 32 біти (EHL)]
96+(CC×32)+32	[Заголовок розширення – блоки даних]						
96+(CC×32)+X*(32+32×EHL)	Дані						
якщо P=1	Заповнення (Padding data)						Л

Рисунок 2.1 – Вигляд RTP пакета

RTP не має стандартного зарезервованого порту. Єдине обмеження полягає в тому, що з'єднання проходить з використанням парного номера, а наступний непарний номер використовується для зв'язку RTCP. RTP має

сесію, яка встановлюється кожного потоку мультимедіа. Наприклад, аудіо- та відеопотоки будуть мати різні RTP-сесії, що дозволяють приймачеві для цього виділити конкретний потік.

Протокол RTCP є супутнім протоколом до протоколу RTP і використовується для передачі контрольної інформації в системах реального часу. Його головна мета – забезпечити моніторинг та керування якістю обслуговування мультимедійних потоків. Саме завдяки цьому протоколу можна контролювати пропускну здатність мережі, встановлювати швидкість передачі даних та адаптувати її до змінюючихся умов мережі, збирати інформацію про показники якості обслуговування, такі як затримка, джиттер, пакетні втрати та інші параметри, що дозволяють оцінити якість передачі мультимедійних даних. Щоб забезпечити правильне відтворення мультимедійних потоків у реальному часі RTCP використовує таймстампи для синхронізації.

Перед розподілом медіа даних на RTP пакети їх кодують спеціально зазначеними кодеками на стороні відправника і декодують отриману інформацію з даних RTP пакета на стороні одержувача. Кодеки відіграють важливу роль у збереженні та передачі медіаданих, зокрема аудіо та відео. Основна мета використання кодеків – зменшення обсягу даних, не втрачаючи при цьому суттєвої якості сигналу та підтримуючи різні формати. Вони виконують компресію медіаданих, що дозволяє зменшити їх обсяг для більш ефективної передачі та збереження. Це особливо важливо у випадку відео, оскільки великий обсяг відеоданих може вимагати значних пропускних здатностей мережі та зберігання на пристроях, через що кодеки допомагають економити пропускну здатність мережі. Вони використовують, в більшості випадків, стиск із втратами, що призводить до погіршення якості та меншого обсягу. Варто зауважити, що кодеки забезпечують сумісність між різними пристроями та платформами. Завдяки цьому, медіадані можуть бути відтворені на різних пристроях без проблем.

У відеоконференціях використовуються різні кодеки. Багато клієнтів в одній сесії можуть використовувати різні кодеки при передачі медіапотоку один одному. Розглянемо деякі популярні з них.

H.264 – один з найбільш популярних стандартів стиснення відео, який забезпечує високу якість зображення при розумному обсязі даних. H.264 є ефективним кодеком і використовується в багатьох системах відеоконференцій, включаючи популярні платформи Zoom, Skype, Microsoft Teams тощо.

VP8 та VP9 – розроблені компанією Google і використовуються у їх відеоконференційній платформі Google Meet. VP8 та VP9 забезпечують високу якість відео при зменшеному обсязі даних, що дозволяє ефективно передавати відеосигнал через мережу.

Opus – аудіокодек, розроблений для високоякісного стиснення аудіо даних. Opus використовується у багатьох відео конференційних платформах для передачі голосу з високою якістю та низькою затримкою. Підтримує широкий діапазон бітрейтів.

Advanced Audio Coding (AAC) – є популярним аудіокодеком, який використовується для стиснення аудіоданих у відеоконференціях та інших мультимедійних додатках. AAC надає високу якість звуку при невеликому обсязі даних.

SILK – є аудіокодеком, спеціально розробленим для передачі голосу в режимі реального часу через мережу з високою якістю та низькою затримкою. Його часто використовують у відеоконференційних системах та програмах VoIP.

Якщо клієнти не мають спільного набору кодеків, то передача медіаданих може бути проблематичною або навіть неможливою. Тому важливо, щоб всі клієнти у сесії мали можливість підтримувати однакові кодеки та декодеки або мати механізм перекодування (конвертації) медіа даних з одного формату в інший для забезпечення сумісності. Це може збільшити використання ресурсів і вплинути на якість передачі даних.

Коли отримувач отримує стиснений потік даних, він використовує відповідний декодер (той же кодек або сумісний) для розпакування та відтворення оригінальних медіаданих.

Більшість кодеків, замість того, щоб передавати, наприклад, відеозображення як 30 випадкових кадрів на секунду, роблять ставку на те, що кадри між собою особливо не відрізняються. Тому кодеки формують якийсь головний кадр – keyframe, з якого все починається, який зберігає повну інформацію про зображення, після чого всі наступні кадри передають як різниця або відмінність від цього головного кадру і попереднього стану – interframe. Якщо в середині буде втрачено якийсь interframe, то будь-які наступні вже не будуть представляти жодної цінності, поки попередній не буде відновлений. Це відбувається тому, що стан було втрачено і неможливо від нічого зробити якусь різницю і загалом порівняння як таке.

2.1.3 Знайомство через опис та кандидатів

Перед відправкою будь-якого виду медіаданих два клієнти, під час встановлення з'єднання обмінюються між собою кількома видами інформації, одними з яких є SDP та ICE.

ICE (Interactive Connectivity Establishment) – це протокол і техніка, яка використовується для встановлення прямих мережевих з'єднань між клієнтами в мережі, яка може включати NAT (Network Address Translation) і фаєрволи. У процесі встановлення з'єднання за допомогою ICE клієнти обмінюються інформацією про свої IP-адреси, порти та протоколи. Вони можуть використовувати STUN для визначення своєї зовнішньої IP-адреси та порта, а також типу NAT, який вони використовують. У разі, якщо пряме з'єднання неможливе, ICE може використовувати TURN, щоб маршрутизувати трафік через реле-сервер, який дозволяє передавати медіадані.

SDP (Session Description Protocol) – це протокол опису сесії, який потрібен для обміну інформацією про параметри мультимедійної сесії між учасниками комунікації. SDP містить інформацію про медіа-потоки, такі як

тип кодеку, бітрейт, розмір зображення, параметри звуку та інші характеристики. Він дозволяє встановлювати з'єднання між клієнтами і налаштовувати параметри передачі мультимедійних даних. Опис сеансу містить нуль або більше медіа-описів. Медіа-опис зазвичай відображається в одному медіа-потоку. Отже, якщо потрібно описати дзвінок трьома відео потоками та двома звуковими доріжками, то повинно бути 5 медіа-описів.

Кожен рядок в SDP починатиметься з одного символу, який є ключем. Після цього йде знак рівності. Усе, що стоїть після знака рівності, є значенням. Після завершення значення на наступному рядку повинна бути нова пара ключ-значення. Протокол опису сеансу визначає всі дійсні ключі. Дозволяється використовувати лише літери для ключів, як визначено в протоколі. Усі ці ключі мають важливе значення.

Приклад того, як виглядає інформація найпростішого SDP повідомлення:

```
v=0
m=audio 4000 RTP/AVP 111
a=rtpmap:111 OPUS/48000/2
m=video 4000 RTP/AVP 96
a=rtpmap:96 VP8/90000
a=my-sdp-value
```

Є два медіа-описи: один типу аудіо з fmt 111 і один типу відео з форматом 96. Перший медіа-опис має лише один атрибут. Цей атрибут відображає тип корисного навантаження 111 на Opus. Другий медіа-опис має два атрибути. Перший атрибут відображає тип корисного навантаження 96 як VP8, а другий атрибут – просто my-sdp-value.

Деякі з позначень ключей:

v – версія, має дорівнювати 0.

o – походження, містить унікальний ідентифікатор, корисний для повторного узгодження.

s – назва сеансу має дорівнювати -.

t – час, має дорівнювати 0 0.

m – Опис носія (m=<media> <port> <proto> <fmt> ...), докладно описаний нижче.

a – атрибут, вільне текстове поле.

c – Дані підключення мають дорівнювати IN IP4 0.0.0.0.

SDP може містити необмежену кількість медіа-описів. Визначення медіа-опису містить список форматів. Ці формати відповідають типам корисного навантаження RTP. Фактичний кодек потім визначається атрибутом зі значенням rtpmap в описі носія.

Що таке вже згаданий NAT і чому з ним мають виникнути проблеми у відеоконференції та встановлення зв'язку двох клієнтів? NAT (Network Address Translation) – це технологія, яка використовується в комп'ютерних мережах для перетворення IP-адрес. Вона дозволяє групі пристроїв використовувати одну зовнішню IP-адресу для з'єднання з Інтернетом. На даний момент в інтернеті найбільш поширені IP адреси версії IPv4. Це дозволяє створити приблизно лише 4.3 млрд. IP-адресів. Однак цієї, начебто, великої кількості критично не вистачає. Щоб вирішити цю проблему, і був створений механізм NAT. Він дозволяє внутрішнім приватним/сірим адресам, що не маршрутизуються, виходити у світ під одним або декількома білими зовнішніми адресами.

Існує декілька видів NAT:

- Статичний: переводить одну внутрішню адресу в одну зовнішню.
- Динамічний: має декілька зовнішніх адрес для перекладу внутрішніх.
- Порт-переклад: дозволяє внутрішнім адресам використовувати одну зовнішню зіставляючи внутрішні та зовнішні унікальні порти запитів.
- Симетричний: створює унікальне відображення адреси та порту для кожного з'єднання.

Проблема під час встановлення з'єднання у відеоконференціях виникає через те, що кожен новий пакет, що проходить через NAT, отримує новий порт у таблиці сесій NAT. Це робить неможливим для зовнішнього джерела ідентифікувати та встановити з'єднання з конкретним пристроєм, що знаходиться за NAT.

2.2 Метод сервісного рішення

2.2.1 WebRTC

Існує безліч програм та сайтів для спілкування по відеозв'язку, включаючи такі популярні платформи, як Zoom, Microsoft Teams, Google Meet, Skype, FaceTime, WhatsApp, Discord тощо. Кожен з цих продуктів має власну реалізацію оркестрування різними технологіями та протоколами для створення медіа спілкування у реальному часі. Для організації відеоконференцій компанія Google створила особливий браузерний протокол – WebRTC, який має відкритий вихідний код і вимагає не великих зусиль для того, щоб зібрати весь необхідний сервіс.

WebRTC (Web Real-Time Communication) – це набір правил для двох клієнтів (далі агент або пір) для узгодження двонаправленого безпечного зв’язку в реальному часі. Можна сміливо сказати, що організація сесій через цей протокол це набір послідовних дій які залежить один від одного. Наприклад такі кроки як: сигналізація, підключення, захист, спілкування є послідовними, що означає, що попередній крок має бути успішним на 100%, щоб почати наступний [1].

WebRTC так влаштований, що кожен такий крок насправді складається з інших протоколів та різних технологій. Це робить його тим самим так званим оркестратором. Залежність компонентів протоколу будується таким способом (рис. 2.2).

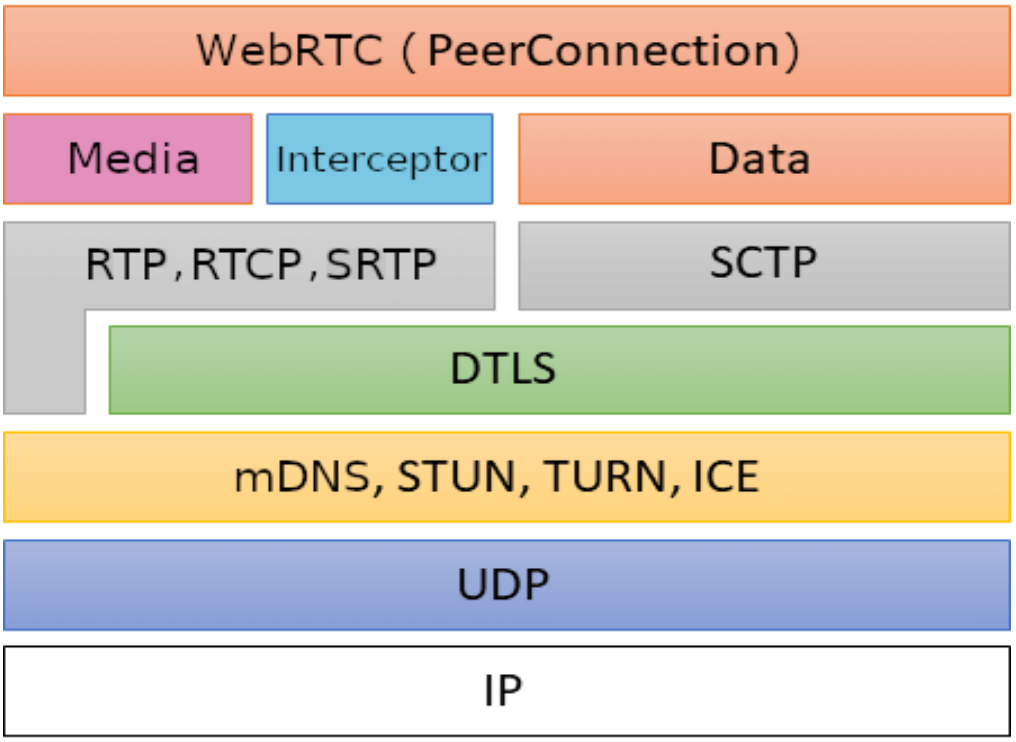

Рисунок 2.2 – Компоненти WebRTC протоколу

Щоб переконатися у надійності та ефективності будь-якої технології необхідно проаналізувати її явні переваги та недоліки [6]:

- Інтегрований безпосередньо у веб-браузери, тому для участі у відеоконференціях не потрібне встановлення додаткових програм або плагінів.
- Забезпечує низьку затримку та високу швидкість передачі даних, що робить комунікацію по відео більш плавною та натуральною.
- Використовуються механізми автоматичного відновлення з'єднання та адаптивної буферизації, щоб мінімізувати втрати даних та забезпечити стабільний зв'язок.
- Працює виключно із зашифрованими даними, забезпечуючи конфіденційність та захист особистої інформації.

При всьому цьому пір може зіткнутися з проблемами мережі, такими як обмеження смуги пропускання, затримка або пакетні втрати, які можуть вплинути на якість відеозв'язку. Незважаючи на широку підтримку WebRTC у

багатьох сучасних браузерах, можливі деякі відмінності у реалізації та підтримці стандартів, що може вимагати додаткового налаштування для забезпечення сумісності.

Для того, щоб пір став агентом WebRTC йому необхідно зрозуміти з ким і про що йому починати спілкування.

2.2.2 Сигналізація

Сигналізація в контексті відеоконференцій – це процес обміну управляючою інформацією, і сигналами між учасниками конференції. Вона відіграє важливу роль у встановленні та управлінні з'єднанням, дозволяє пірам взаємодіяти один з одним, передавати інформацію про стан та контролювати різні аспекти відеоконференції. Процес сигналізації проходить до кінця сесії. Важливо зауважити, що реалізація сигналізації зазвичай відбувається поза темою протоколу, що означає, що сервіси зазвичай не використовують сам WebRTC для обміну сигнальними повідомленнями. Будь-яка архітектура придатна для їх надсилання. Тобто в цьому випадку потрібно побудувати окремий сервер з будь-якою зручною інфраструктурою, наприклад, через вебсокети [2].

Клієнт надсилає запит на сигнальний сервер про створення/приєднання до сесії. Створювати конференцію можна як завгодно, у нашому разі клієнт передаватиме ідентифікатор сигнальному серверу. Будь-який інший клієнт відправляє такий самий запит, після чого у відповідь може отримати пул сокетів учасників сесії. Кожен пір генерує свій власний SDP і передає його на сигналізацію решті пірів, що чекають на підключення в одній конференції. Це дозволяє клієнтам встановити відповідні параметри зв'язку та обмінятися інформацією про свої можливості для успішного проведення відеоконференції [5]. Разом з цим вони використовують ICE протокол обмінюючись ICE кандидатами для того, щоб виявити доступні мережеві адреси, подолати проблеми з NAT, фаерволами та встановити оптимальний шлях зв'язку між собою. В кінці, коли всі встановили з'єднання та налаштували свої параметри

зв'язку, вони готові розпочати передачу медіаданих у рамках відеоконференції. Весь час активного сеансу його учасники продовжують обмін необхідної їм інформації, таких як SDP та ICE кандидати.

2.2.3 STUN та TURN

Той факт, що RTP використовує адреси портів що присвоюються динамічно, створює йому труднощі з проходженням міжмережевих екранів або NAT, для обходу цієї проблеми, як правило, використовується STUN-сервер. STUN (Session Traversal Utilities for NAT) дозволяє клієнтам визначити свою зовнішню IP-адресу та порт, а також тип NAT, через який вони підключені до Інтернету. Клієнти надсилають запити STUN-серверу, який повертає інформацію про адресу та порт клієнта, видимих ззовні. Це дозволяє клієнтам дізнатися про свої загальнодоступні адреси та використовувати їх у процесі встановлення пірингового з'єднання [3].

У процесі роботи клієнт відправляє запит (рис. 2.3) на STUN-сервер, який відповідає з інформацією про зовнішню IP-адресу та порт клієнта. Ця інформація дозволяє клієнту дізнатися, які зовнішні адреси та порти він повинен використовувати для встановлення зв'язку з іншими пристроями в мережі сеансу. STUN сервер є дуже простим сервісом і його завдання – "подивитися" на клієнта з Інтернету.

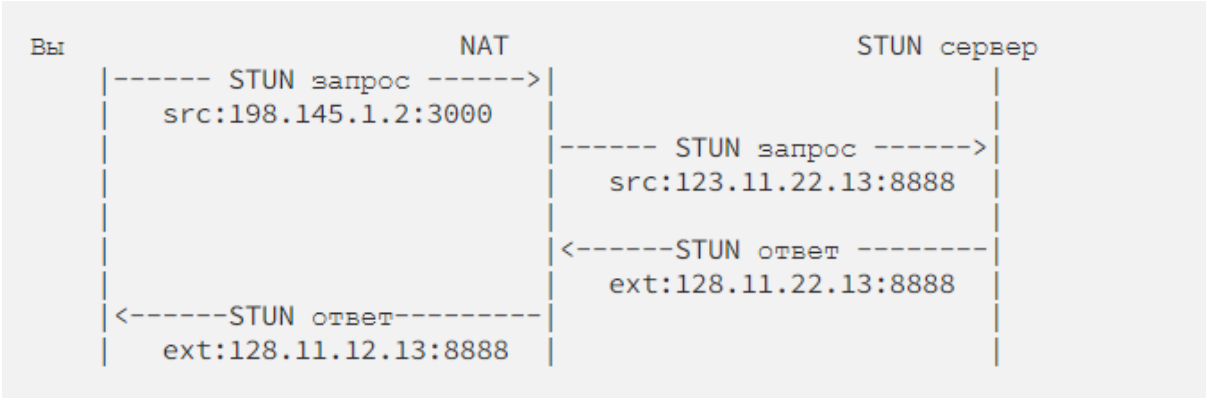

Рисунок 2.3 – Схема STUN запитів

Якщо пряме пірингове з'єднання не може бути встановлено, клієнт може використовувати сервер TURN як посередник (рис. 2.4). Клієнти, які не можуть встановити пряме з'єднання, відправляють свої дані на сервер TURN (Traversal Using Relays around NAT), а потім отримують їх назад. TURN діє як проміжна точка обміну даними між клієнтами. Він відповідає з IP-адресою та портом, які клієнт може використовувати для встановлення з'єднання. Клієнт потім надсилає свій трафік через TURN-сервер, який пересилає його іншому клієнту, який також використовує TURN. Це дозволяє уникнути проблем, пов'язаних з обмеженнями NAT, такі як блокування портів або використання симетричного NAT. Однак використання такого рішення може додати затримку та навантаження на мережу.

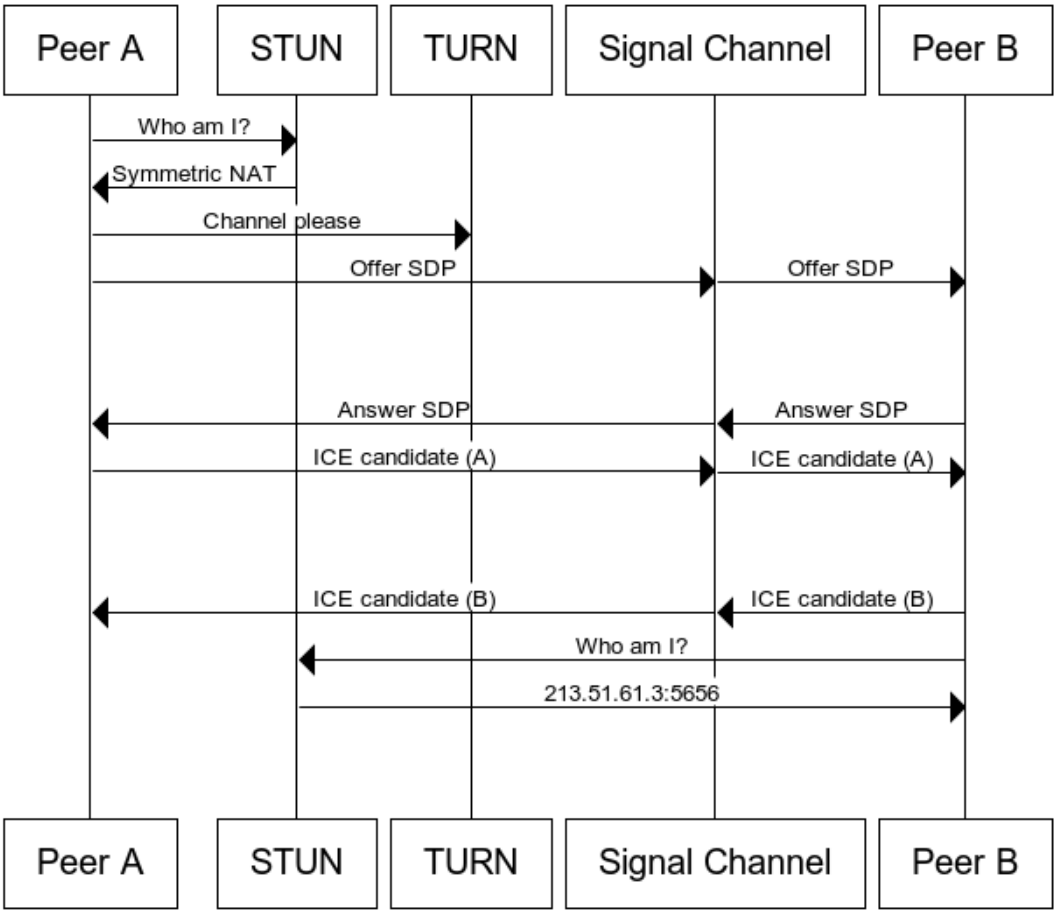

Рисунок 2.4 – Загальна схема процесу сигналізації

2.2.4 Медіа-сервер

Яким чином можна поєднати між собою велику кількість людей у мережу? На це питання, як правило, бере на себе обов'язки відповісти топологія. Є кілька способів зробити сеансову мережу і всі вони мають свої переваги та недоліки. Ці рішення загалом діляться на дві категорії; однорангові або клієнт/сервер. Гнучкість WebRTC дозволяє створювати обидва.

Один-на-один, або peer-2-peer (рис. 2.5). При такому з'єднанні всі пакети медіа передаються безпосередньо від агента до агента.

Рисунок 2.5 – Схема P2P топології

Повна сітка або Full Mesh (рис 2.6). У цій топології кожен користувач встановлює з’єднання з кожним іншим безпосередньо. Це означає, що треба кодувати та завантажувати відео незалежно для кожного учасника дзвінка. Умови мережі між кожним з’єднанням відрізнятимуться, тому не можливо повторно використовувати ті самі відео- та аудіо- потоки. Такий варіант також може дати велике та небажане навантаження на пристрій клієнта.

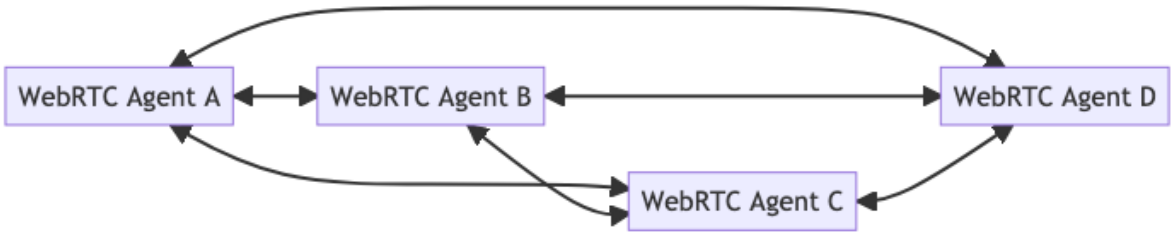

Рисунок 2.6 – Схема Full Mesh топології

Гібридна сітка або Hybrid Mesh (рис. 2.7). Такий підхід не визначає з'єднання між кожним учасником. Натомість медіа ретранслюється через

однорангові пристрої в мережі. Це означає, що творцю медіа не потрібно використовувати таку велику пропускну здатність для його розповсюдження. Таке рішення не дасть клієнту дізнатися кому пересилаються дані та успішність їх доходу.

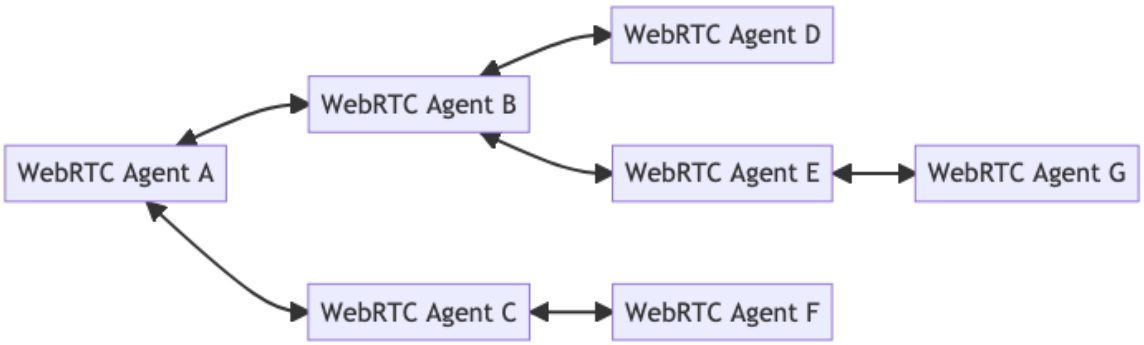

Рисунок 2.7 – Схема Hybrid Mesh топології

SFU (Selective Forwarding Unit) (рис. 2.8). Цей варіант реалізує топологію клієнт/сервер замість P2P. Кожен вузол WebRTC підключається до SFU та завантажує свої медіадані. Потім SFU пересилає цей поток до кожного підключеного клієнта. За допомогою SFU кожен агент WebRTC має закодувати та завантажити своє відео лише один раз. Тягар розповсюдження його серед усіх пірів лежить на SFU. Підключення за допомогою SFU набагато легше, ніж P2P.

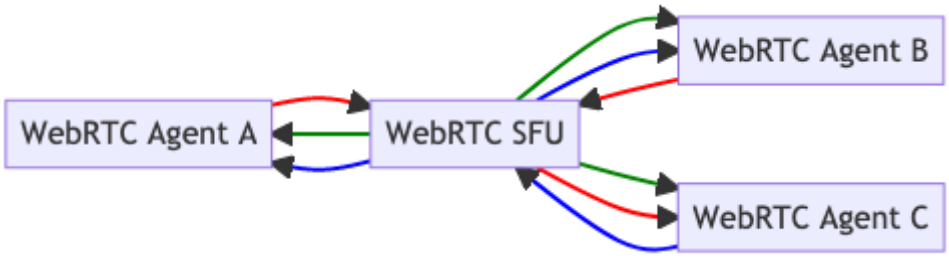

Рисунок 2.8 – Схема SFU топології

MCU (Multipoint Conferencing Unit) (рис. 2.9). Це топологія клієнт/сервер, подібна до SFU, але об’єднує вихідні потоки. Замість того, щоб

розповсюджувати вихідні медіа потоки в незмінному вигляді, він повторно кодує їх як один канал.

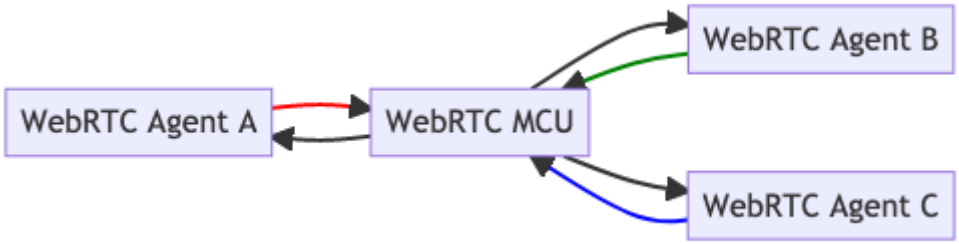

Рисунок 2.9 – Схема MCU топології

Таким чином, останні дві варіації з'єднань клієнтів між собою створюють необхідну в ефективності та безпеці топологію "зірочка".

У центрі цих топологій знаходиться централізований медіасервер. Медіа-сервер обробляє та керує потоком медіаданих у режимі реального часу. Він виконує функції кодування, декодування, обробки та передачі медіа-вмісту між учасниками в комунікаційній мережі. Медіа сервер приймає пакети від клієнтів, що означає можливість повної взаємодії з ними. Наприклад, він здатний змінювати кодування на формат не доступний іншим клієнтам. Потік пакетів можна зберігати або перенаправляти копії на інші сервери, зберігаючи запис конференції. Розкодування медіа-сервером медіаданих дає можливість їх видозмінити, наприклад, додавши до відеотрансляції візуальні ефекти або фон.

2.2.5 Криптографічний захист

WebRTC влаштований таким чином, що нав'язує і не дозволяє використовувати не захищену інформацію. Можна бути впевненим, що третя сторона не зможе побачити, що надсилається.

WebRTC для цього використовує два існуючі протоколи: DTLS (Datagram Transport Layer Security) та SRTP (Secure Real-time Transport Protocol). Це вже знайомі терміни та словосполучення, але обгорнуті деякою безпекою.

DTLS дозволяє узгодити сеанс, а потім безпечно обмінюватися даними між двома вузлами. Це як TLS, але DTLS використовує UDP замість TCP як транспортний рівень. Це означає, що протокол має обробляти ненадійну доставку. Першим використовується DTLS. Він здійснює “рукостискання” через з’єднання, яке забезпечує ICE. DTLS – це клієнт/серверний протокол, тому одна сторона має розпочати “рукостискання”. Ролі клієнт/сервер вибираються під час сигналізації. У цей момент обидві сторони пропонують сертифікат. Після завершення процесу цей сертифікат порівнюється з хешем сертифіката в описі сеансу.

SRTP – це протокол, розроблений спеціально для шифрування RTP-пакетів. Створення сеансу SRTP ініціалізується за допомогою ключів, згенерованих DTLS. SRTP не має механізму “рукостискання”, тому його потрібно завантажувати за допомогою зовнішніх ключів. Після цього можна обмінюватися медіафайлами, зашифрованими за допомогою SRTP.

В цілому, криптографія WebRTC забезпечує захищену передачу даних і забезпечує конфіденційність, цілісність і аутентифікацію в рамках комунікацій між клієнтами.

2.3 Архітектура серверної частини додатку

Архітектура веб-додатку в основному представляє відносини та взаємодії між такими компонентами, як інтерфейси користувача, монітори обробки транзакцій, бази даних та інші. Основна мета – переконатися, що всі елементи правильно працюють разом. Правильно обрана архітектура спрощує тестування, підтримку, модифікацію, розробку та розгортання, а також забезпечує незалежність між компонентами програми. Для розробки серверної частини була обрана так звана чиста архітектура (рис. 2.10).

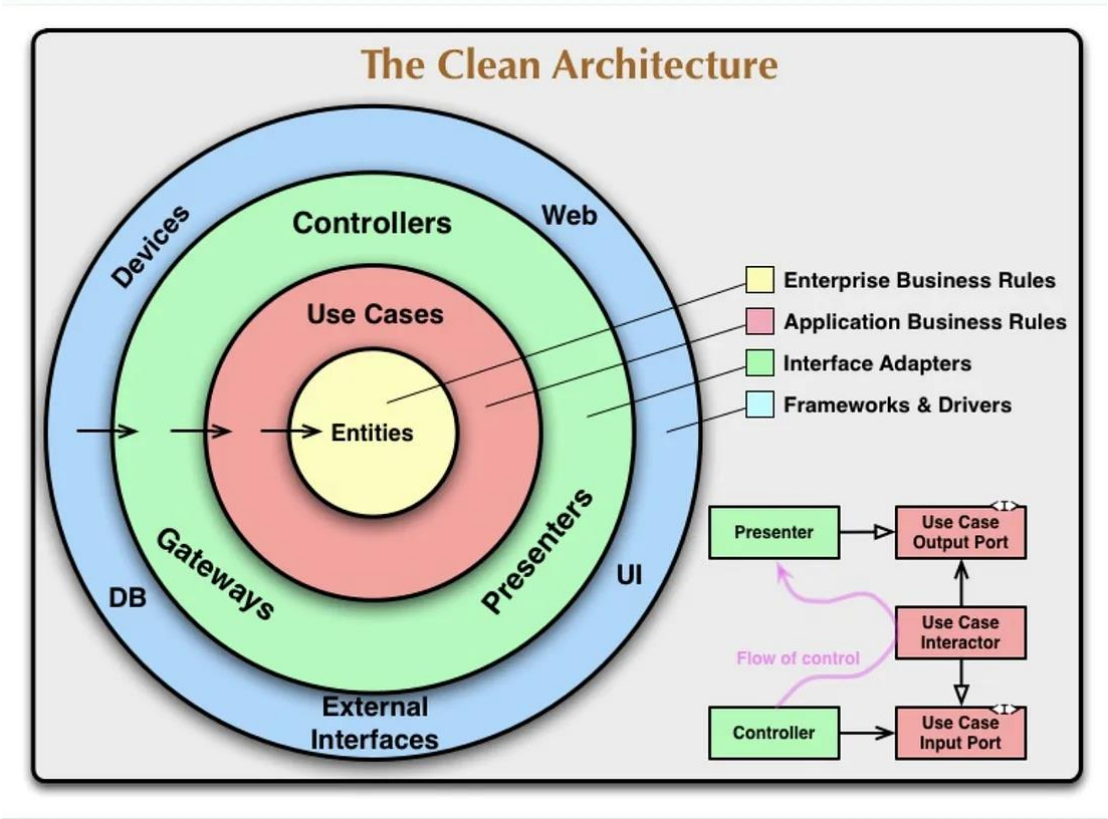

Рисунок 2.10 – Схема чистої архітектури

Відповідно до цієї схеми будь-які залежності можуть бути спрямовані лише всередину кола. Це означає, що компоненти із внутрішнього кола можуть взагалі нічого не знати про компоненти із зовнішнього, тобто внутрішнє коло ніяк не залежить від зовнішнього. Напрямок чорних стрілок на схемі відповідає цьому принципу. Більш детальний огляд кожного з представлених шарів:

Frameworks and Drivers (синій колір). Тут розміщені програмні компоненти:

- User Interface – інтерфейс користувача.
- Database – бази даних.
- External Interfaces – зовнішні інтерфейси, такі як API.
- Інші мережеві запити.

Interface Adapters (зелений колір). Шар інтерфейсних адаптерів включає:

- Presenters – логіка та стан інтерфейсу користувача, наприклад вид даних відповіді.

- Controllers – інтерфейс із необхідними додатком методами, які реалізовані через інтернет, пристрої або зовнішні інтерфейси.
- Gateways – інтерфейс, що включає всі операції типу CRUD (create, read, update, delete), що виконуються додатком з базою даних.

Application Business Rules (червоний колір). Це правила, які є хоч і не основними, але необхідними для логіки функціонування конкретної програми. У цей рівень входять Use Cases (сценарії використання), тобто він містить всі функції, що надаються додатком. З іншого боку, цей рівень визначає контролер/шлюз , викликані конкретного варіанта використання. Іноді потрібні контролери із різних модулів. Саме Use Cases в цій моделі архітектури додатку покриваються якомога більшою кількістю тестів.

Enterprise Business Rules (жовтий колір). Цей рівень містить правила основного чи предметного рівня логіки функціонування. Крім того, це найменш схильний до змін рівень. На нього не впливають зміни на будь-якому зовнішньому рівні. Оскільки логіка функціонування буде змінюватися не часто, зміни на цьому рівні відбуваються дуже рідко. Цей рівень включає Entities. Entity може бути або основною структурою даних, необхідною для правил логіки функціонування, або об'єктом з методами, що містять логіку функціонування.

2.4 Генерація сторінок

Генерація клієнтських сторінок (інтерфейс, який бачитиме візуально і буде зручно взаємодіяти користувач) визначає те, яким буде підхід до організації навігації та оновлення всієї frontend частини програми. Для роз'єднання залежності клієнта від сервера було обрано так званий варіант SPA (Single Page Application). Односторінкова програма буквально складається з однієї HTML сторінки [14]. Вона постійно взаємодіє з користувачем, динамічно переписуючи свій стан, а не завантажуючи цілі нові сторінки з сервера. Це збільшує швидкість відповіді від сервера і скорочує обсяг даних,

що передається між браузером і сервером. Все це досягається за допомогою мови програмування JavaScript, яка покликана у браузері взаємодіяти з деревом елементів сторінки. Це його відрізняє від Multi Page Application, де кожен запит формує нові HTML дані і знову завантажуються браузером. Варто зазначити, що SPA програми будуть чуйно працювати тільки якщо у користувача достатньо ресурсів пристрою, оскільки будь-які обчислення та ре-рендеринг лежить на плечах його браузера.

Таким чином основний недолік клієнтського рендерингу полягає в тому, що кількість необхідного JavaScript зазвичай збільшується разом із зростанням програми. Ситуація погіршується з підключенням нових JavaScript-бібліотек, поліфілів та іншого стороннього коду, який змагається між собою за обчислювальні потужності та часто потребує обробки, перш ніж вміст сторінки можна буде відобразити. Підходом до рішення є поділ основного коду на бандли, які ліниво завантажуються залежно від потреб у даному проміжку часу.

3 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

3.1 Формування вхідних даних

Основні дані для організації відеоконференції, якими є медіа потоки користувача, надаватимуть механізми WebRTC. Цими потоками буде вся інформація із зовнішніх медіа пристроїв користувача, таких як мікрофон і камера, у вигляді безперервного потоку байтів.

Для основного розгортання WebRTC у веб-додатку протокол надає зручний браузерний інтерфейс (API), який бере на себе багато низькорівневих функцій надаючи лише архітектурну абстракцію над всіма процесами, що відбуваються за “лаштунками”.

Під час сигналізації на стороні клієнта буде використано такі деякі інтерфейси. RTCPeerConnection відповідає за встановлення та керування з'єднанням між пірами. Він обробляє сигнальну обробку, управління ICE кандидатами та встановлення сесії для передачі медіаданих. RTCSessionDescription описує метадані сесії, такі як тип та характеристики медіапотоку. Він використовується для обміну SDP між пірами.

Метод .getUserMedia() дозволяє отримати доступ до камери та мікрофону користувача. Він використовується для захоплення відео та аудіо сигналу з пристроїв та передачі його самому веб-додатку. .addTrack() створює новий потік RTP. .createOffer() генерує опис сеансу, який буде надано спільно з віддаленим одноранговим вузлом. .setLocalDescription() фіксує будь-які зміни та викликається зі значенням, яке генерує .createOffer(). Метод .setRemoteDescription() встановлює вказаний опис сеансу як поточну пропозицію або відповідь віддаленого вузла. .addIceCandidate() дозволяє агенту додавати більше віддалених кандидатів ICE у будь-який час. Цей API надсилає ICE Candidate прямо в підсистему ICE і не має іншого впливу на ширше з’єднання WebRTC [7].

MediaStream представляє потік медіаданих, що містить відео, аудіо або обидва типи даних. Він використовується для передачі медіа-потоків між пірами. MediaStreamTrack представляє окремі треки всередині медіа-потоку, такі як відео або аудіо. Він дозволяє контролювати активацію та деактивацію треків під час комунікації.

3.2 Вибір засобів програмної реалізації

Для написання всієї серверної частини програма була обрана мова програмування Rust. Rust – це мова програмування, заточена під розробку високонавантажених систем. На ньому пишуть веб-застосунки, браузерні движки, блокчейни та інші складні платформи, які обробляють запити мільйонів користувачів. Перед запуском код на Rust відразу перекладається машинною мовою, тобто перетворюється на набір нулів та одиниць. Цим Rust відрізняється, наприклад, від JavaScript, де код спочатку обробляється інтерпретатором, рядок за рядком виконуючись процесором. Навпаки ж скомпільовані програми запускаються швидше та споживають менше пам'яті. Головна перевага Rust полягає у поєднанні швидкості та надійності. З одного боку, мова дає програмісту повний доступом до пам'яті, з другого страхує його від багатьох помилок. Тому на Rust пишуть додатки, для яких важлива стабільна робота у важких умовах: багатопотокові системи, програмне забезпечення для складних обчислень тощо.

Деякі основні концепції Rust, які дають всі ці переваги полягають у:

- Запобігання помилкам безпеки пам'яті, таким як нульові покажчики, вихід за межі масивів та гонки даних. Він досягає цього за допомогою системи володіння, системи типів та перевірок часу компіляції.
- Використання системи володіння управління пам'яттю. Володіння визначає, які частини коду відповідальні за звільнення виділеної пам'яті. Це дозволяє уникнути проблем подвійного звільнення чи витоків пам'яті.
- Керування часом життя посилань та гарантування, що посилання

залишаються дійсними під час використання. Це допомагає запобігти помилкам використання недійсних посилань.

- За допомогою механізмів типів та перевірок часу компіляції Rust забезпечує безпеку доступу до даних із різних потоків.

Завдяки цим особливостям програма буде написана без технічних помилок, дозволяючи приділити більше уваги при написанні лише логічних тестів.

Як об'єктно-реляційна система управління базами даних була обрана PostgreSQL, найбільш розвинена з відкритих СКБД у світі. За допомогою СКБД (система керування базами даних) можна створювати, модифікувати або видаляти записи, відправляти транзакцію (набір з декількох послідовних запитів) спеціальною процедурною мовою запитів SQL. Ця система ще має негласну назву безкоштовний Oracle Database, оскільки обидві системи адаптовані під великі проекти та високе навантаження. PostgreSQL працює зі складними, складовими запитами. Система справляється із завданнями розбору та виконання трудомістких операцій, які мають на увазі і читання, і запис, і валідацію одночасно. Ця СКБД має модифікацію до SQL, яка називається PL/pgSQL. Це процедурне розширення, яке підтримує складні обчислення та доповнює класичний SQL новими можливостями. PostgreSQL є проектом, який відомий високою якістю налагодження. Кожна версія системи з'являється у доступі лише після повної перевірки, тому СКБД дуже стабільна. Часта проблема безкоштовних проектів полягає у нових версіях з великою кількістю багів, що в цьому випадку нівелюється цією СКБД. Згідно з незалежними автоматизованими дослідженнями, у вихідному коді PostgreSQL є одна помилка на 39000 рядків коду. Це у п'ять разів менше, ніж у MySQL, та у п'ятдесят разів менше, ніж у ядрі операційної системи Linux.

Як центральна частина розподілу медіа потоків від клієнтів буде обрано медіа-сервер з відкритим вихідним кодом від компанії MeetEcho [9]. Їх продукт Janus є скоріше простим медіа-сервером, тому його найкраще використовувати для проектів, які потрібно швидко запускати та не

потребують багато налаштувань. Це робить його цікавою платформою, яку варто дослідити. З цікавих моментів можна виділити його здатність спілкуватися з плагінами, які можуть надати йому різних додаткових можливостей. У моєму випадку плагіни дозволяють налаштувати зручне спілкування між Janus gateway та власним сигнальним сервером.

3.3 Проектування бази даних

У процесі проектування бази даних було визначено її структуру, а саме склад таблиць, їхній вигляд і логічні зв'язки. Реляційна база даних вимагає, щоб кожна таблиця мала стовпці зі своїми розміром, типом даних, особливими обмеженнями (ключі, індекси). Спочатку необхідно було визначити основні завдання для розв'язання яких створюється база, та потреби цих задач у даних.

У процесі розробки було виділено сутності майбутньої бази даних. Сутність є типом об'єктів, які зберігатимуться у сховищі. Кожна таблиця у базі даних має представляти одну сутність. Було взято правило визначити кожну сутність як об'єкт реалізованої сфери дипломного проекту. Кожна сутність має набір атрибутів. Атрибут є властивістю, яка описує деяку характеристику об'єкта. Кожен стовпець зберігає один атрибут сутності. А кожен рядок представляє окремий об'єкт чи екземпляр сутності.

Повний огляд таблиць, визначених під час проектування бази даних:

1) users – таблиця, що вміщує інформацію про користувача (рис. 3.1) та має такі поля:

- id – ідентифікаційний номер;
- username – ім'я користувача;
- nickname – ім'я облікового запису;
- email – електронна пошта;
- password – пароль від облікового запису;
- picture – аватар користувача;
- is_verif_confirmed – значення верифікації;

- created_at – дата створення;
- updated_at – дата останнього оновлення;

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	bigint			●	●	nextval('us
username	character varying	70		●	○	
nickname	character varying	50		●	○	
email	text			●	○	
password	text			●	○	
picture	text			○	○	
is_verif_confirmed	boolean			●	○	true
is_banned	boolean			●	○	false
created_at	timestamp without tim...			●	○	now()
updated_at	timestamp without tim...			●	○	now()

Рисунок 3.1 – Таблиця users з додатку pgAdmin 4

2) user_tokens – таблиця, що вміщує інформацію про токени користувача (рис. 3.2) та має такі поля:

- id – ідентифікаційний номер;
- verification_token – токен верифікації;
- confirmation_token – токен підтвердження;
- user_id – ідентифікаційний номер користувача;
- created_at – дата створення;
- updated_at – дата останнього оновлення;

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	integer			●	●	nextval('us
is_video_muted	boolean			●	○	true
is_audio_muted	boolean			●	○	false
user_id	bigint			●	○	
created_at	timestamp without time...			●	○	now()
updated_at	timestamp without time...			●	○	now()

Рисунок 3.2 – Таблиця user_settings з додатку pgAdmin 4

3) user_settings – таблиця, що вміщує інформацію про налаштування користувача (рис. 3.3) та має такі поля:

- id – ідентифікаційний номер;
- is_video_muted – значення стану відеокамери;
- is_audio_muted – значення стану мікрофону;
- user_id – ідентифікаційний номер користувача;
- created_at – дата створення;
- updated_at – дата останнього оновлення.

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	integer			●	●	nextval('us
verification_token	text			●	○	
confirmation_token	text			○	○	
user_id	bigint			●	○	
created_at	timestamp without time...			●	○	now()
updated_at	timestamp without time...			●	○	now()

Рисунок 3.3 – Таблиця user_tokens з додатку pgAdmin 4

4) user_sessions – таблиця, що вміщує інформацію про активні сесії користувача (рис. 3.4) та має такі поля:

- id – ідентифікаційний номер;
- token – токен оновлення;

- ip_address – IP адреса, з якої була створена сесія;
- operating_system – операційна система, з якої була створена сесія;
- device – пристрій, з якого була створена сесія;
- user_id – ідентифікаційний номер користувача;
- created_at – дата створення;
- updated_at – дата останнього оновлення;

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	integer			●	●	
token	text			●	○	
ip_address	text			●	○	
operating_system	text			○	○	
device	text			○	○	
user_id	bigint			●	○	
created_at	timestamp without tim...			●	○	now()
updated_at	timestamp without tim...			●	○	now()

Рисунок 3.4 – Таблиця user_sessions з додатку pgAdmin 4

5) rooms – таблиця, що вміщує інформацію про кімнату та (рис. 3.5) має такі поля:

- id – ідентифікаційний номер;
- tag – унікальний тег кімнати;
- key – секретний ключ для приєднання;
- name – назва кімнати;
- picture – аватар кімнати;
- created_at – дата створення;
- updated_at – дата останнього оновлення;

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	integer			✓	✓	nextval('roo
tag	text			✓		
key	character varying	150		✓		
name	character varying	150		✓		
picture	text					
created_at	timestamp without time...			✓		now()
updated_at	timestamp without time...			✓		now()

Рисунок 3.5 – Таблиця rooms з додатку pgAdmin 4

6) **room_settings** – таблиця, що вміщує інформацію про налаштування кімнати (рис. 3.6) та має такі поля:

- id – ідентифікаційний номер;
- is_private – значення приватності кімнати;
- is_video_muted – значення стану відеокамер;
- is_audio_muted – значення стану мікрофонів;
- is_chat_muted – значення активності чату;
- room_id – ідентифікаційний номер кімнати;
- created_at – дата створення;
- updated_at – дата останнього оновлення;

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	integer			✓	✓	nextval('ro
is_private	boolean			✓		true
is_video_muted	boolean			✓		false
is_audio_muted	boolean			✓		false
is_chat_muted	boolean			✓		false
room_id	integer			✓		
created_at	timestamp without tim...			✓		now()
updated_at	timestamp without tim...			✓		now()

Рисунок 3.6 – Таблиця room_settings з додатку pgAdmin 4

7) **members** – таблиця, що вміщує інформацію про учасників кімнати (рис. 3.7) та має такі поля:

- is_video_muted – значення стану відеокамери у кімнаті;
- is_audio_muted – значення стану мікрофона у кімнаті;
- is_chat_muted – значення доступу до чату у кімнаті;
- user_id – ідентифікаційний номер користувача;
- room_id – ідентифікаційний номер кімнати;
- created_at – дата створення;
- updated_at – дата останнього оновлення;

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
is_video_muted	boolean			●	○	false
is_audio_muted	boolean			●	○	false
is_chat_muted	boolean			●	○	false
user_id	bigint			●	●	
room_id	integer			●	●	
role_id	integer			●	○	
created_at	timestamp without ti...			●	○	now()
updated_at	timestamp without ti...			●	○	now()

Рисунок 3.7 – Таблиця members з додатку pgAdmin 4

8) **black_lists** – таблиця, що вміщує інформацію про заблокованих учасників у кімнаті (рис. 3.8) та має такі поля:

- user_id – ідентифікаційний номер користувача;
- room_id – ідентифікаційний номер кімнати;
- created_at – дата створення;
- updated_at – дата останнього оновлення;

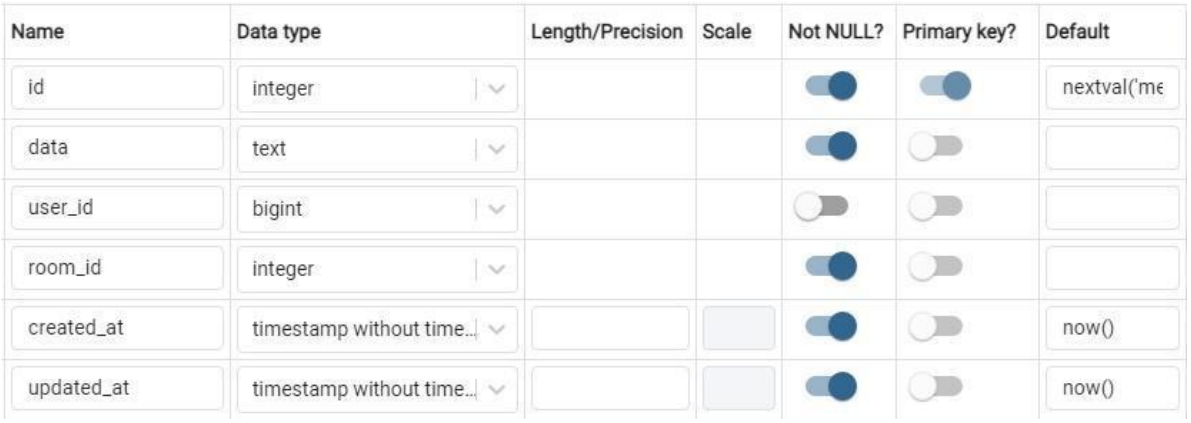

Рисунок 3.8 – Таблиця messages з додатку pgAdmin 4

9) **messages** – таблиця, що вміщує інформацію про повідомлення з чату кімнати (рис. 3.9) та має такі поля:

- id – ідентифікаційний номер;
- data – дані повідомлення;
- user_id – ідентифікаційний номер користувача;
- room_id – ідентифікаційний номер кімнати;
- created_at – дата створення;
- updated_at – дата останнього оновлення;

Рисунок 3.9 – Таблиця black_lists з додатку pgAdmin 4

10) **roles** – таблиця, що вміщує інформацію про ролі (рис. 3.10) та має такі поля:

- id – ідентифікаційний номер;
- alias – псевдонім або назва ролі;
- created_at – дата створення;

- updated_at – дата останнього оновлення;

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	integer			●	●	nextval('rol
alias	text			●	○	
created_at	timestamp without time...			●	○	now()
updated_at	timestamp without time...			●	○	now()

Рисунок 3.10 – Таблиця roles з додатку pgAdmin 4

На рисунку 3.11 представлено концептуальну модель бази даних веб-додатку для організації відеоконференцій. Концептуальна модель бази даних є абстрактне подання даних, їх структури та зв'язків між ними. Вона описує основні сутності та його атрибути, а також зв'язок між сутностями. Концептуальна модель є основою для проектування фізичної моделі бази даних.

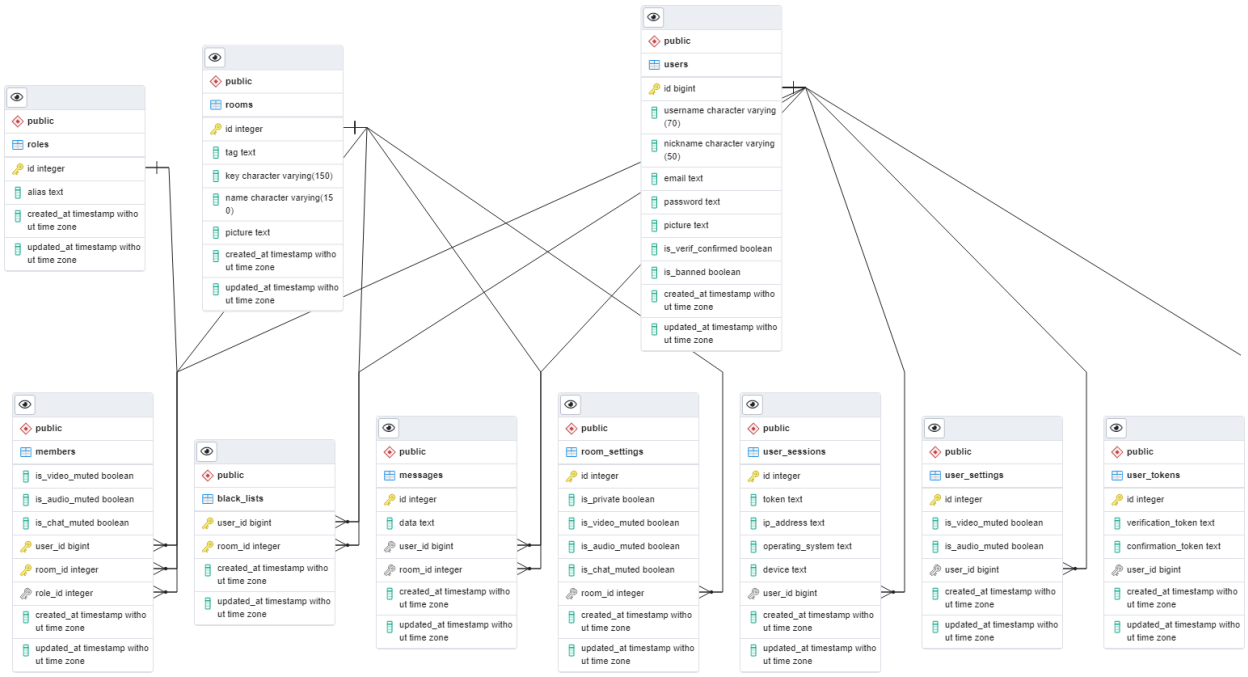

Рисунок 3.11 – Концептуальна модель бази даних з додатку pgAdmin 4

Для гнучкості та ефективності роботи з даними у сховищі бази даних

було проведено операцію нормалізації. Це процес організації та структурування бази даних з метою скорочення надмірності даних. Це спосіб переконатися в тому, що кожне поле та запис організовані логічно таким чином, щоб не лише уникнути надмірності, але й зробити використання будь-якої реляційної бази даних ефективнішим. У результаті вийшло, що кожен стовпець у таблиці містить значення, які не можуть бути розділені на більш дрібні частини й мають унікальні імена. Всі дані поділені на групи (зв'язані таблиці), які залежать лише від свого первинного ключа. Наприклад, сутність користувача має окремо таблиці під налаштування, токени та сесії. А інформація про учасників кімнати винесена в окрему сутність, яка пов'язує між собою користувача та кімнати, в яких він знаходиться.

3.4 Опис програмної реалізації

Додаток складається з кількох частин, кожна з яких виконує свої клієнт-серверні функції. Перше, що бачить користувач, це безпосередньо дизайн (верстка) клієнтської частини, яка має приємний візуальний стиль і адаптивність. Демонстраційний додаток використовує макет у побудові frontend частини за допомогою фреймворку ReactJS. На цьому рівні використовуючи WebRTC API для зчитування медіа даних користувача, обміну деякими подіями з сигнальним сервером та встановлення з'єднання з медіа-сервером, через STUN/TURN системи, приймаючи на один RTCPeerConnection усі потоки інших учасників, а з другого публікуючи свої власні. Хоча Janus і може виступати як сигнальний сервер, деякі його гнучкі можливості необхідно було обернути в іншу реалізацію.

Janus Gateway за допомогою плагіна [10] дозволяє створювати постійні кімнати, даючи можливість розпочати чи зупинити передачу потоків. За логікою програми необхідно, щоб сесії були тільки в рамках передачі та отримання медіа даних [11], а весь функціонал взаємодії з кімнатою повинен був лягти на плечі backend частини. Також Janus не дозволяє мікрофону

перебувати в активному стані, поки відеокамера користувача не передає потоки байтів медіа-серверу. Крім того, завданням сигнального сервера в цій схемі є контролювання знаходження користувача в даний момент тільки в одній конференції, обриваючи зв'язок з попередньою.

Паралельно з цим існує вже згаданий backend, який відповідає за серверний функціонал збереження та обробки даних користувача у рамках логіки самого проекту. З'єднання з ним здійснюється за допомогою REST API, протоколом, який ефективно використовує HTTP. Детальний опис кожного маршруту (роуту), за якими можна здійснювати CRUD операції з сервером та базою даних:

POST /v1/user/registration – створення нового облікового запису користувача разом зі стандартними значеннями налаштувань.

GET /v1/user/verification/:token – верифікація облікового запису користувача за допомогою токена, раніше надісланого на його електронну пошту після реєстрації.

POST /v1/user/authorization – авторизація користувача у системі. На цьому етапі створюються два закодованих токена, перший з яких необхідний для приватного доступу до деяких маршрутів, а другий для перестворення токенів у разі, якщо час життя першого закінчиться.

GET /v1/user/regenerate – маршрут, який робить оновлення двох токенів.

PATCH /v1/user/edit(/, /password, /picture, /settings) – набір маршрутів для редагування даних користувача, його налаштувань, аватара та пароля.

GET /v1/user/destroy/confirmation – запит на надсилання повідомлення з токеном для підтвердження видалення облікового запису.

DELETE /v1/user/destroy/:token – підтвердження дії видалення облікового запису користувача.

GET /v1/user/me – отримання детальної інформації про обліковий запис разом із налаштуваннями.

GET /v1/user/session/s – відповідає перерахуванням активних сесій використовуючи пагінацію.

DELETE /v1/user/session/destroy/:id – завершення вказаної сесії користувача.

GET /v1/user/unauthorization – завершення поточної сесії та її видалення з бази даних.

POST /v1/room/create – створення нової кімнати та встановлення поточного користувача в ній як адміністратора.

GET /v1/room/join/:room_id – приєднання до вказаної кімнати в ролі користувача.

GET /v1/room/leave/:room_id – маршрут дозволяє залишити вказану кімнату, видаляючи всі дані про свою участь, за винятком повідомлень у чаті.

PATCH /v1/room/edit(/:room_id, /:room_id/picture, /:room_id/settings) – набір маршрутів для редагування інформації про кімнату, її налаштувань та аватара відповідно.

DELETE /v1/room/destroy/:room_id – видалення кімнати та будь-якої пов'язаної з нею інформації.

GET /v1/room/s – відповідає перерахуванням всіх кімнат, у яких перебуває поточний учасник, використовуючи пагінацію.

GET /v1/room/search – пошук кімнат використовуючи надані частини назв із параметрів запиту.

GET /v1/room/c – отримання числа кімнат, у яких перебуває поточний учасник.

PATCH /v1/room/:room_id/member/edit(/:user_id, /:user_id/role) – зміна налаштувань та роль вказаного учасника в кімнаті.

GET /v1/room/:room_id/member(/eject/:user_id, /block/:user_id, /unblock/:user_id) – запит на виконання ряду дій над вказаним учасником, таких як його виключення з кімнати, блокування та розблокування в кімнаті.

GET /v1/room/:room_id/member/:user_id – отримання детальної інформації про вказаного учасника у кімнаті.

GET /v1/room/:room_id/member(/s/:user_id, /m/:user_id) – ряд дій на отримання інформації про учасників у кімнаті, використовуючи або пагінацію,

або передаючи набір їх ідентифікаторів.

GET /v1/room/:room_id/member/c – запит на отримання числа учасників у вказаній кімнаті.

GET /v1/room/:room_id/member/blocked – перегляд інформації з чорного списку кімнати використовуючи пагінацію для вибірки.

GET /v1/room/chat/:room_id/message/s – отримання списку повідомлень із чату вказаної кімнати використовуючи пагінацію.

WS /v1/room/chat/:room_id – приєднання та активна участь у чаті зазначеної кімнати за WebSocket протоколом.

Спілкування із сигнальним сервером весь час сесії здійснюється виключно за допомогою WebSocket протоколу. Повідомлення має складатися з назви запиту та об'єкта певних очікуваних даних. Ряд дій та подій якими можна обмінюватися із сервером:

make-contract – приймає ідентифікатор учасника для створення двох сесій, якими оперуватиме сервер під час спілкування з Janus.

relay-description – це одночасно і дія і подія, яка покликана пересилати всі SDP.

relay-candidate – пересилає всіх ICE кандидатів весь час сеансу.

join-room – дія для створення, у разі відсутності, або приєднання до активної конференції.

leave-room, leaved-room – залишити активну сесію та закінчити з'єднання з сервером.

take-media, retake-media – дія на отримання (або зміну одержування) потоків за їх зазначеною інформацією

modify-media, modified-media – зміна власних потоків. Після цієї дії всі учасники сесії отримають повідомлення про цю подію.

modify-media, pin-me, modified-media, he-pinned – прості події на повідомлення учасників про підняту руку або закріплення себе на головному екрані інтерфейсу конференції.

sanction-media, eject-member, sanctioned-media, ejected-member – ряд дій

для керування учасником сесії. Включає блокування чи розблокування активного потоку учасника або його повне виключення з конференції.

gived-media – подія, що дає потоки, які варто запитати, щоб активно отримувати їх від медіа-сервера.

gived-members – подія, дані якої складаються із детальної інформації учасників активної сесії.

3.5 Аналіз результатів

Система має перевірку доступу для авторизованих користувачів, надаючи можливість створення сесії засобами JWT. Для того, щоб використовувати всі можливості веб-програми клієнту необхідно мати обліковий запис та його позитивний статус верифікації.

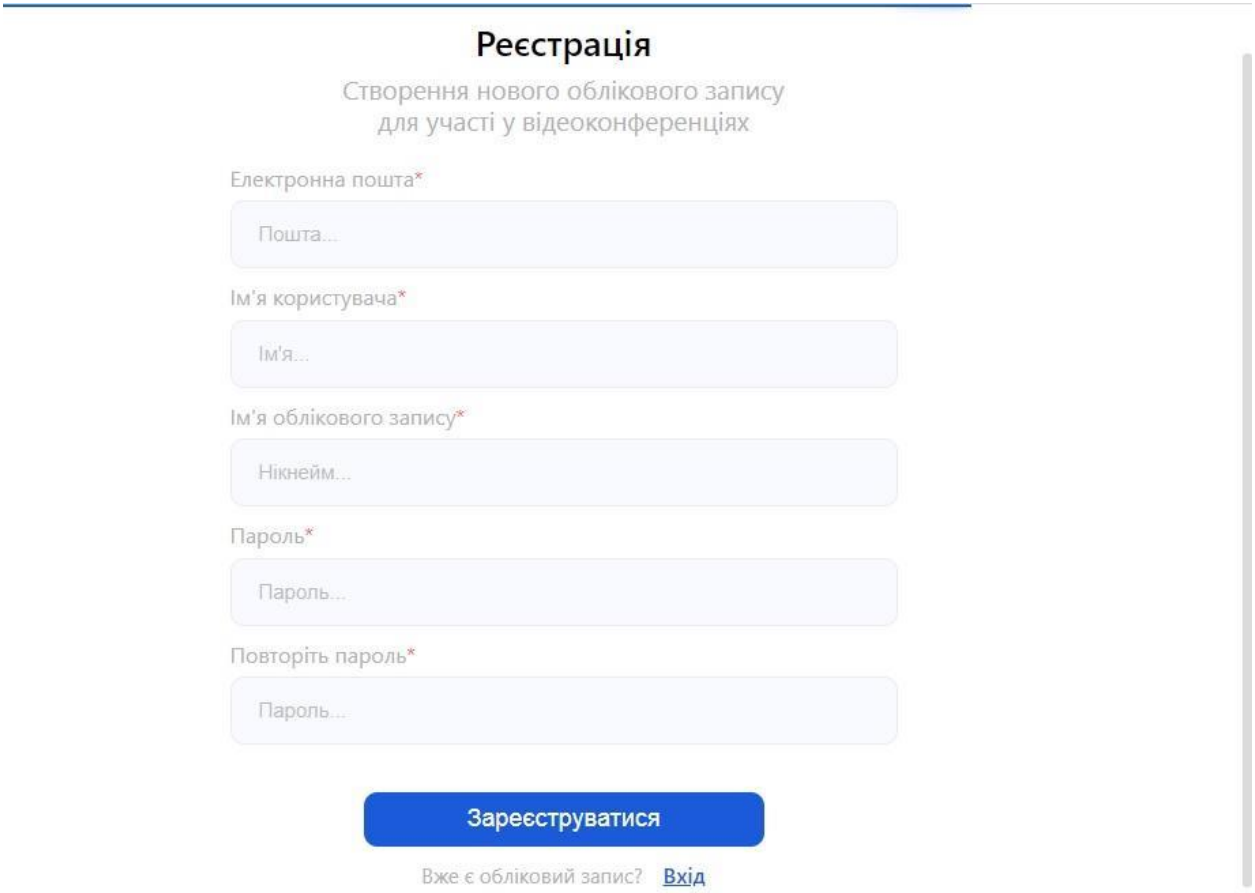

Рисунок 3.12 – Сторінка реєстрації

Рисунок 3.13 – Сторінка авторизації

Користувач може за бажанням регулювати власні налаштування у відповідному місці (рис. 3.14).

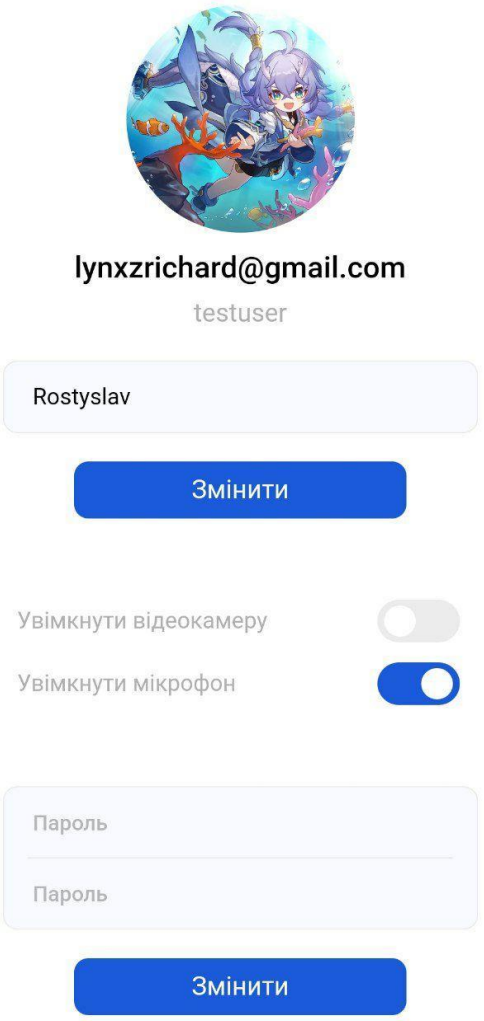

Рисунок 3.14 – Мобільний вигляд сторінки налаштувань

У другій частині цієї сторінки можна знайти список активних сесій і завершити їх за необхідності використаа відповідну, для цієї дії, частину інтерфейсу (рис. 3.15).

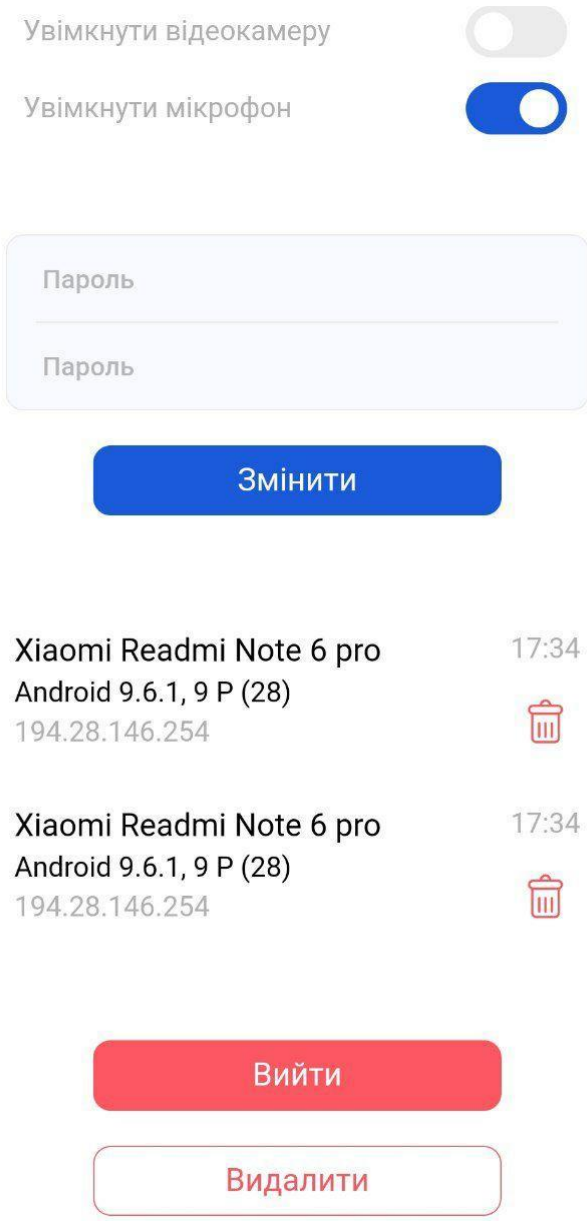

Рисунок 3.15 – Мобільний вигляд другої частини сторінки налаштувань

Головна сторінка містить список кімнат, учасником яких є поточний користувач. Елемент інтерфейсу у вигляді кнопки "Створити" відкриває додаткове меню, в якому необхідно заповнити дані для створення нової кімнати (рис. 3.16).

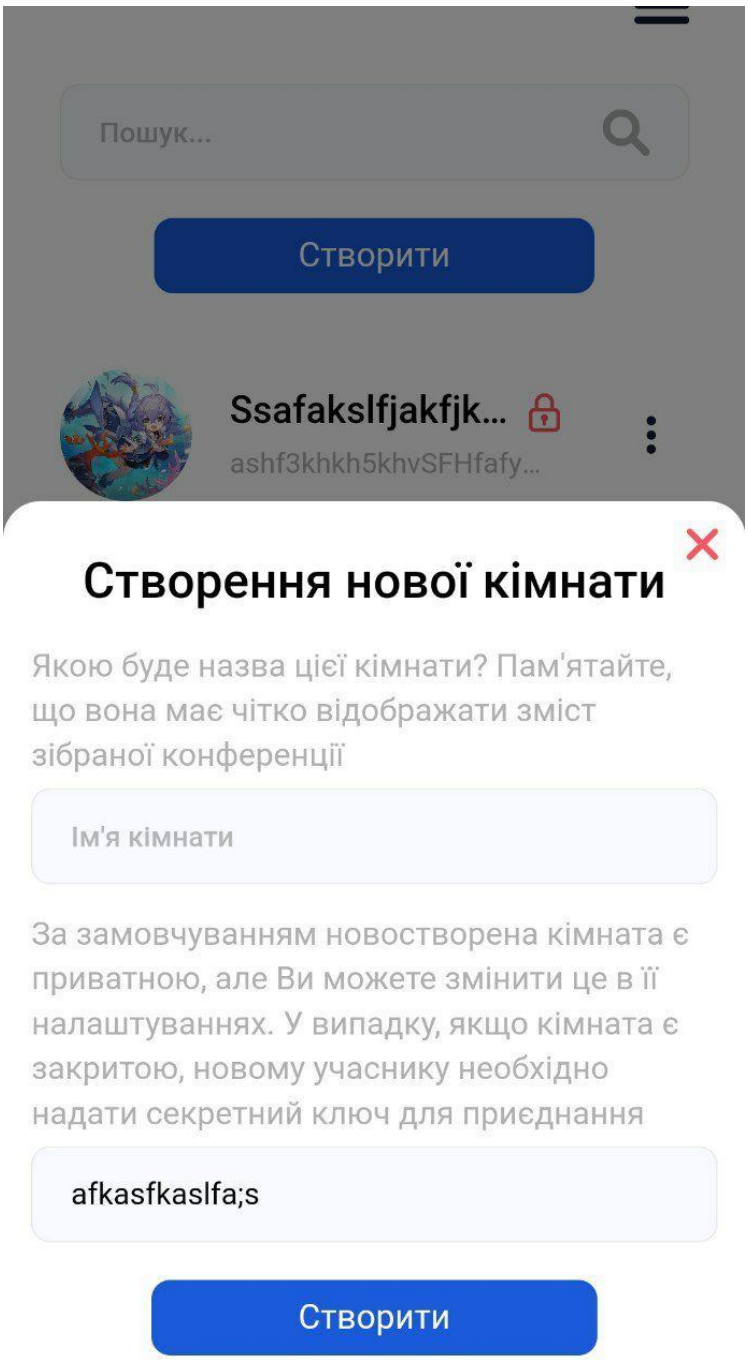

Рисунок 3.16 – Мобільний вигляд меню створення нової кімнати

Під час відеоконференції учасник може взаємодіяти з нею, використовуючи спеціальні функції в інтерфейсі, такі як "Підняти руку", "Включити відеокамеру" тощо (рис. 3.17). У нижній частині є зручний чат для текстового спілкування між учасниками кімнати (рис. 3.18). Це також є невід'ємною частиною сучасних програм для конференцій в режимі реального часу.

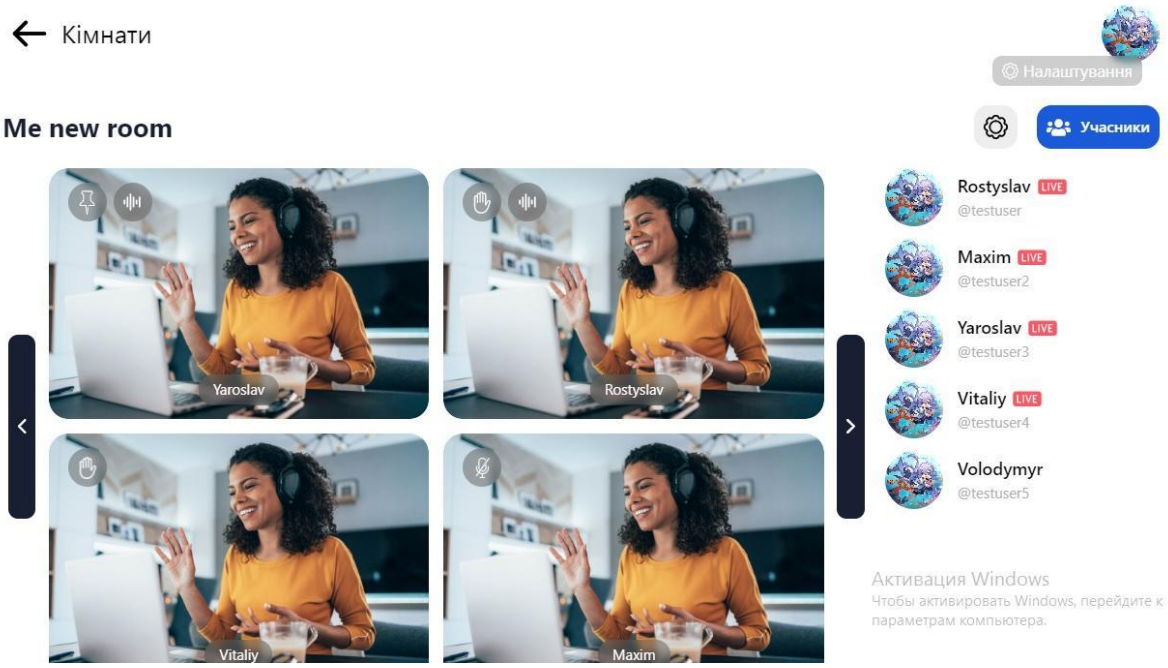

Рисунок 3.17 – Відеоконференція

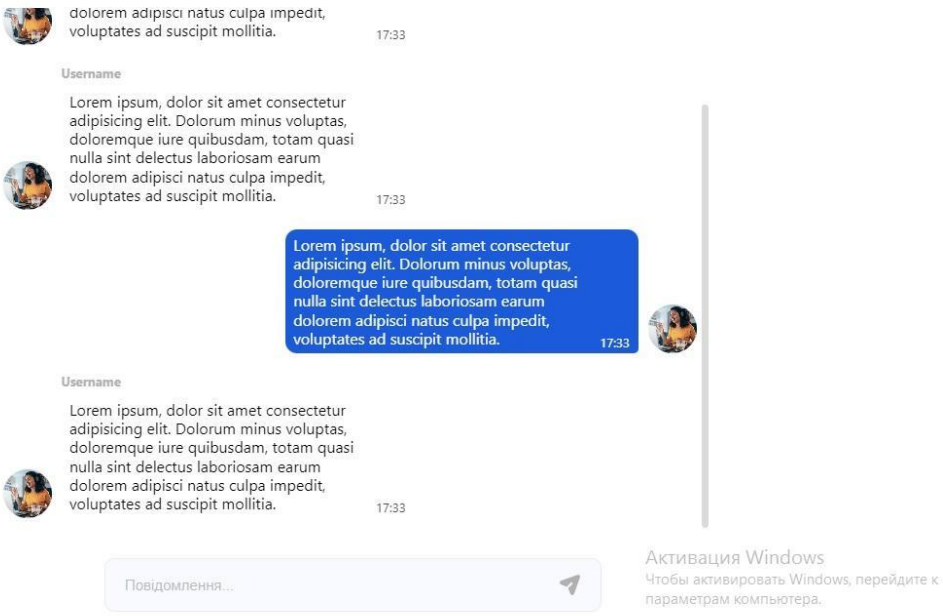

Рисунок 3.18 – Чат

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було виконано такі завдання:

1. Спроектовано систему сутностей бази даних на основі якої було нормалізовано схему реляційного сховища. У результаті вдалося досягти необхідної ефективності, а також масштабованості для майбутніх оновлень програми.

2. Розроблено незалежну backend частину, яка має чисту архітектуру, яка розділила додатки на шари абстракції, що зробило програмний код незалежним, масштабованим без серйозних наслідків. Правильно побудована архітектура, а також обрана мова програмування дали можливість загострити увагу та приділити якнайбільше часу тестуванню логіки серверної частини.

3. З урахуванням сучасних вимог був відмальований і зверстаний візуально приємний дизайн, який у поєднанні з майбутнім інтерфейсом дає інтуїтивність і зрозумілість у його використанні відображаючи гарний досвід користувача.

4. Був налаштований Janus медіа-сервер з відкритим вихідним кодом для потреб організації відеоконференцій. Для того, щоб з'єднати його з іншою логікою серверної частини програми був створений сигнальний сервер який інкапсулює будь-які звернення до Janus у свій невеликий протокол, тим самим закриваючи його деякі недоліки та функції, які не змогли підійти в процесі розробки до додатку.

У ході дослідження було розглянуто також основні особливості WebRTC, його переваги та причини, за якими ця технологія була обрана для реалізації відеоконференцій у проекті. Він має низку переваг. Одне з таких – це його інтеграція з веб-браузерами без встановлення додаткового програмного забезпечення. Це спрощує процес використання та розповсюдження відеоконференцій, оскільки користувачі можуть підключатися до сесія з будь-якого пристрою з веб-браузером, включаючи комп'ютери, смартфони та планшети.

WebRTC надає розробникам широкий набір API та інструментів для створення налаштованих рішень, включаючи можливість додавання додаткових функцій. Можна зробити висновок, що WebRTC є ефективною та зручною технологією для організації відеоконференцій. Він забезпечує високу якість передачі аудіо та відео даних, має низьку затримку та хорошу сумісність з різними пристроями та операційними системами. Гнучкість та розширюваність WebRTC дозволяють налаштовувати відеоконференції відповідно до необхідних потреб. Цей протокол, що дуже важливо, диктує безпечні та правильні серверно-архітектурні правила.

Надалі планується розробка повноцінного frontend додатку, який може використовувати всі доступні функції вже існуючих сервісів та за основу якого буде взято вже зверстаний інтерфейсний дизайн.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. WebRTC для допитливих. URL: https://webrtcforthecurious.com (дата звернення: 13.04.2023).

2. Сигналізація та відео виклик. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling (дата звернення: 16.04.2023).

3. Перетворення мережевих адрес. URL: https://en.wikipedia.org/wiki/Network_address_translation (дата звернення: 18.04.2023).

4. RTP. URL: https://uk.wikipedia.org/wiki/RTP (дата звернення: 13.04.2023).

5. Як побудувати відеочат на RUST+WASM за допомогою WebRTC. URL: https://charles-schleich.medium.com/webrtc-video-chat-tutorial-using-rust-wasm-fa340f7aeef9 (дата звернення: 22.04.2023).

6. Введення у протоколи. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols (дата звернення: 22.04.2023).

7. WebRTC API. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API (дата звернення: 22.04.2023).

8. Jitter Buffer. URL: https://getstream.io/glossary/jitter-buffer (дата звернення: 22.04.2023).

9. Поради щодо вибору медіасервера WebRTC для своїх потреб. URL: https://www.softermii.com/blog/best-webrtc-media-server-tips-how-to-choose-one-for-your-needs (дата звернення: 23.04.2023).

10. Janus VideoRoom документація до плагіна. URL: https://janus.conf.meetecho.com/docs/videoroom (дата звернення: 03.05.2023).

11. Janus RESTful, WebSockets, RabbitMQ, MQTT, Nanomsg та UnixSockets API. URL: https://janus.conf.meetecho.com/docs/rest.html (дата звернення: 03.05.2023).

12. Чистий код з Rust та Axum. URL: https://www.propelauth.com/post/clean-code-with-rust-and-axum (дата звернення: 12.05.2023).

13. Модульне тестування Rust Web API за допомогою Axum 0.6. URL: https://www.youtube.com/watch?v=_cYIhG_3qSo&ab_channel=RainerStropek (дата звернення: 15.05.2023).

14. Односторінковий додаток (SPA). URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA (дата звернення: 18.05.2023).

15. Випуск React 18: глибоке занурення в нові функції та оновлення. UR: https://www.scalablepath.com/react/react-18-release-features (дата звернення: 19.05.2023).

ДОДАТОК А

```
#[derive(serde::Serialize, serde::Deserialize)]
#[serde(rename_all = "kebab-case")]
pub enum Action {
    MakeContract {
        #[serde(rename(deserialize = "id"))]
        member: Member
    },
    RelayDescription {
        sdp: Sdp,
        priority: u8
    },
    RelayCandidate {
        ice: Ice,
        priority: u8
    },

    JoinRoom {
        room: Room,
        #[serde(default)]
        name: Option<String>,
        #[serde(rename(deserialize = "streams"))]
        #[serde(default)]
        stream_datas: Option<Vec<StreamData>>
    },
    LeaveRoom,
    TakeMedia {
        #[serde(rename(deserialize = "streams"))]
        stream_datas: Vec<StreamData>
    },
    RetakeMedia {
        #[serde(rename(deserialize = "newStreams"))]
        take_stream_datas: Vec<StreamData>,
        #[serde(rename(deserialize = "oldStreams"))]
        untake_stream_datas: Vec<StreamData>
    },

    ModifyMedia {
        room: Room,
        #[serde(rename(deserialize = "configs"))]
        stream_configs: Vec<StreamConfig>
    },
    RelayMedia {
        #[serde(rename(deserialize = "id"))]
        member: Member,
        #[serde(rename(deserialize = "configs"))]
        stream_configs: Vec<StreamConfig>
    },
    ModifyHand {
        room: Room
    },
    PinMe {
        room: Room
    },
    ViewMembers {
        room: Room
    },

    SanctionMedia {
        room: Room,
        #[serde(rename(deserialize = "id"))]
        member: Member,
```

```
        #[serde(rename(deserialize = "config"))]
        stream_config: StreamConfig
    },
    EjectMember {
        room: Room,
        #[serde(rename(deserialize = "id"))]
        member: Member
    },

    #[serde(rename_all(serialize = "camelCase"))]
    GivedMedia {
        #[serde(rename(serialize = "streams"))]
        stream_datas: Vec<StreamData>,
        is_lead: bool
    },
    #[serde(skip_deserializing)]
    GivedMembers {
        #[serde(rename(serialize = "members"))]
        member_datas: Vec<MemberData>
    },
    ModifiedMedia {
        #[serde(rename(serialize = "id"))]
        member: Member,
        #[serde(rename(serialize = "configs"))]
        stream_configs: Vec<StreamConfig>
    },
    ModifiedHand {
        #[serde(rename(serialize = "id"))]
        member: Member
    },
    HePinned {
        #[serde(rename(serialize = "id"))]
        member: Member
    },
    ViewedMembers {
        members: Vec<Member>
    },
    SanctionedMedia {
        #[serde(rename(serialize = "config"))]
        stream_config: StreamConfig
    },
    LeavedRoom {
        #[serde(rename(serialize = "id"))]
        member: Member
    },
    EjectedMember
}

impl Action {
    pub fn parse(data: &str) -> Result<Self> {
        Ok(
            serde_json::from_str(data).or_else(|_|
                serde_json::from_str(&format!(r#""{}""#, data))
            )?
        )
    }

    pub fn escape(&self) -> Result<String> {
        Ok(serde_json::to_string(self)?)
    }
}


fn handle(action: Action, socket_addr: SocketAddr, transaction_list:
```

```
TransactionList, (s_a_m_index, m_s_a_index, m_h_index): Indicies, relay:
Relay) -> Result<()> {
    match action {
        Action::MakeContract { member } => take_handle(socket_addr, member,
transaction_list, s_a_m_index, relay)?,
        Action::RelayDescription { sdp, priority } =>
relay_session_description(socket_addr, sdp, priority, s_a_m_index, m_h_index,
relay)?,
        Action::RelayCandidate { ice, priority } =>
relay_interactive_candidate(socket_addr, ice, priority, s_a_m_index,
m_h_index, relay)?,

        Action::JoinRoom { room, name, stream_datas: None } =>
run_check_before_join_the_room(socket_addr, room, transaction_list,
s_a_m_index, m_h_index, relay)?,
        Action::JoinRoom { room, stream_datas: Some(stream_datas), .. } =>
observe_the_conference_room(socket_addr, room, stream_datas, s_a_m_index,
m_h_index, relay)?,
        Action::LeaveRoom =>
leave_and_unobserve_the_conference_room(socket_addr, s_a_m_index, m_h_index,
relay)?,
        Action::TakeMedia { stream_datas } => take_members_media(socket_addr,
stream_datas, s_a_m_index, m_h_index, relay)?,
        Action::RetakeMedia { take_stream_datas, untake_stream_datas } =>
take_or_untake_members_media(socket_addr, take_stream_datas,
untake_stream_datas, s_a_m_index, m_h_index, relay)?,

        Action::ModifyMedia { room, stream_configs } =>
modify_member_media(socket_addr, room, stream_configs, transaction_list,
s_a_m_index, relay)?,
        Action::RelayMedia { member, stream_configs } =>
relay_member_media(socket_addr, member, stream_configs, s_a_m_index,
m_s_a_index, relay)?,
        Action::ModifyHand { room } => modify_member_hand(socket_addr, room,
transaction_list, s_a_m_index, m_h_index, relay)?,
        Action::PinMe { room } => pin_member(socket_addr, room,
transaction_list, s_a_m_index, m_h_index, relay)?,
        Action::ViewMembers { room } => view_members_in_room(socket_addr,
room, transaction_list, relay)?,

        Action::SanctionMedia { room, member, stream_config } =>
sanction_member_media(room, member, stream_config, transaction_list, relay)?,
        Action::EjectMember { room, member } => eject_member_from_room(room,
member, relay)?,

        _ => {}
    }

    Ok(())
}

pub struct Plugin<'a> {
    media_tx: &'a Tx,
    pub media_transaction: Transaction,
    media_session: Reference,
    media_handle: Reference
}


impl<'a> Plugin<'a> {
    pub fn new(media_tx: &'a Tx, media_transaction: Transaction,
```

```
media_session: Reference, media_handle: Reference) -> Self {
        Self {
            media_tx,
            media_transaction,
            media_session,
            media_handle
        }
    }

    fn send(&self, message: Message) -> Result<()> {
        self.media_tx.unbounded_send(message)?;

        Ok(())
    }

    fn send_text(&self, data: String) -> Result<()> {
        self.send(Message::Text(data))
    }

    pub fn emit_attach(&self) -> Result<()> {
        self.send_text(gen_attach(&self.media_transaction,
self.media_session))
    }

    pub fn emit_offer(&self, handle: Reference, r#type: String, sdp: String) -
> Result<()> {
        self.send_text(gen_message_with_jsep(&self.media_transaction,
self.media_session, handle, body::as_publish(), jsep::as_(&r#type, &sdp)))
    }

    pub fn emit_answer(&self, handle: Reference, r#type: String, sdp: String)
-> Result<()> {
        self.send_text(gen_message_with_jsep(&self.media_transaction,
self.media_session, handle, body::as_start(), jsep::as_(&r#type, &sdp)))
    }

    pub fn emit_candidate(&self, handle: Reference, sdp_mid: String,
sdp_m_line_index: u8, candidate: String) -> Result<()> {
        self.send_text(get_trickle(&self.media_transaction,
self.media_session, handle, candidate::as_(&sdp_mid, sdp_m_line_index,
&candidate)))
    }
```

```
    pub fn emit_exists(&self, room: Room) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, self.media_handle, body::as_exists(room)))
    }

    pub fn emit_listparticipants(&self, room: Room) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, self.media_handle, body::as_listparticipants(room)))
    }

    fn emit_create(&self, room: Room) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, self.media_handle, body::as_create(room)))
    }

    fn emit_publisher(&self, handle: Reference, room: Room, member: Member,
name: String) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, handle, body::as_publisher(room, member, &name)))
    }

    pub fn emit_subscriber(&self, handle: Reference, room: Room, members:
Vec<(Member, Stream)>) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, handle, body::as_subscriber(room, &members)))
    }

    pub fn emit_subsribe(&self, handle: Reference, members: Vec<(Member,
Stream)>) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, handle, body::as_subscribe(&members)))
    }

    pub fn emit_update(&self, handle: Reference, members1: Vec<(Member,
Stream)>, members2: Vec<(Member, Stream)>) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, handle, body::as_update(&members1, &members2)))
    }

    pub fn emit_kick(&self, room: Room, member: Member) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, self.media_handle, body::as_kick(room, member)))
    }
```

```
    pub fn emit_leave(&self, handle: Reference) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, handle, body::as_leave()))
    }

    fn emit_detach(&self, handle: Reference) -> Result<()> {
        self.send_text(gen_detach(self.media_session, handle))
    }

    fn emit_destroy(&self, room: Room) -> Result<()> {
        self.send_text(gen_message(&self.media_transaction,
self.media_session, self.media_handle, body::as_destroy(room)))
    }
}

#[derive(Deserialize)]
pub struct OuterAnswer {
    #[serde(rename(deserialize = "sender"))]
    #[serde(default)]
    handle: Reference,
    transaction: Transaction,
    #[serde(rename(deserialize = "data"))]
    pub sign_data: SignData
}

impl OuterAnswer {
    pub fn parse(data: &str) -> Result<Self> {
        Ok(from_str(data)?)
    }
}

#[derive(Deserialize)]
pub struct SignData {
    pub id: u64
}

#[derive(Debug, PartialEq, Deserialize)]
struct InnerAnswer {
    #[serde(rename(deserialize = "sender"))]
    #[serde(default)]
    handle: Reference,
    #[serde(default)]
```

```
    transaction: Transaction,
    #[serde(rename(deserialize = "plugindata"))]
    #[serde(deserialize_with = "parse_event")]
    #[serde(default)]
    event: Event,
    #[serde(rename(deserialize = "jsep"))]
    #[serde(default)]
    expression: Option<Sdp>,
    #[serde(rename(deserialize = "candidate"))]
    #[serde(default)]
    variant: Option<Ice>
}

impl InnerAnswer {
    pub fn parse(data: &str) -> Result<Self> {
        Ok(from_str(data)?)
    }
}

#[derive(Debug, Default, PartialEq)]
enum Event {
    Checked {
        room: Room,
        is_exists: bool
    },
    Created {
        room: Room
    },
    Joined {
        member: Member,
        stream_datas: Vec<StreamData>
    },
    Leaved {
        room: Option<Room>,
        member: Option<Member>
    },
    Lefted,
    Published {
        stream_datas: Vec<StreamData>
    },
    Generated {
        stream_datas: Vec<StreamData>
    },
```

```
    Transmitted {
        track_datas: Vec<TrackData>
    },
    Kicked,
    Finded {
        room: Room,
        members: Vec<Member>
    },
    #[default]
    Unknown
}
fn handle(data: String, handle_list: HandleList, transaction_list:
TransactionList, (s_a_m_index, m_s_a_index, m_h_index): Indicies, relay:
Relay) -> Result<()> {
    let inner_answer = InnerAnswer::parse(&data)?;

    if let Some(expression) = inner_answer.expression {
        let track_datas = match inner_answer.event {
            Event::Generated { stream_datas } => Some(
                stream_datas
                    .into_iter()
                    .map(|s_a| s_a.into())
                    .collect()
            ),
            Event::Transmitted { track_datas } => Some(track_datas),
            _ => None
        };

        relay_session_description(inner_answer.handle, track_datas,
expression, handle_list, m_s_a_index, m_h_index, relay)?;

        return Ok(());
    }
    if let Some(variant) = inner_answer.variant {
        relay_interactive_candidate(inner_answer.handle, variant, handle_list,
m_s_a_index, m_h_index, relay)?;

        return Ok(());
    }
    match inner_answer.event {
        Event::Checked { room, is_exists: false } =>
create_conference_room(inner_answer.transaction, room, transaction_list,
relay)?,
```

```
        Event::Created { room } | Event::Checked { room, .. } =>
join_the_conference_room(inner_answer.transaction, room, handle_list,
transaction_list, relay)?,
        Event::Joined { member, stream_datas } =>
notify_about_media_of_members(inner_answer.transaction, member, stream_datas,
transaction_list, m_s_a_index, relay)?,
        Event::Leaved { room: Some(room), member: None } =>
run_series_actions_after_leaving_the_room(inner_answer.transaction,
inner_answer.handle, room, handle_list, transaction_list, m_s_a_index,
m_h_index, relay)?,
        Event::Leaved { room: None, member: Some(member) } =>
notify_about_leaving_the_room(inner_answer.handle, member, handle_list,
m_s_a_index, relay)?,
        Event::Lefted =>
untake_handle_after_unobserve_the_room(inner_answer.handle, relay)?,
        Event::Published { stream_datas } =>
notify_about_new_media_of_members(inner_answer.handle, stream_datas,
handle_list, m_s_a_index, relay)?,
        // Event::Configured { room } =>
run_collection_after_regulate_media(inner_answer.transaction, room,
transaction_list, relay)?,
        // Event::Permitted { member, stream_config } =>
notify_about_accessibility_media_of_member(inner_answer.handle, member,
stream_config, handle_list, m_s_a_index, relay)?,
        Event::Kicked =>
run_series_actions_after_eject_from_room(inner_answer.handle, handle_list,
s_a_m_index, m_s_a_index, m_h_index, relay)?,
        Event::Finded { room, members } => {
            let (is_tk_room_empty, is_tk_change_publisher, is_tk_change_hand,
is_tk_make_pin) = {
                let transaction_list = transaction_list
                    .lock()
                    .unwrap();

                (

transaction_list.contains_key(&TransactionKind::RoomEmpty(inner_answer.transa
ction.clone())),

transaction_list.contains_key(&TransactionKind::ChangePublisher(inner_answer.
transaction.clone())),

transaction_list.contains_key(&TransactionKind::ChangeHand(inner_answer.trans
```

```
action.clone())),

transaction_list.contains_key(&TransactionKind::MakePin(inner_answer.transact
ion.clone()))
                )
            };

            if is_tk_room_empty && members.is_empty() {
                destroy_the_confrerence_room(inner_answer.transaction, room,
transaction_list, relay)?
            } else if is_tk_change_publisher {

notify_about_regulated_media_of_member(inner_answer.transaction, members,
transaction_list, m_s_a_index, relay)?
            } else if is_tk_change_hand {
                notify_about_member_hand_status(inner_answer.transaction,
members, handle_list, transaction_list, m_s_a_index, relay)?
            } else if is_tk_make_pin {
                notify_about_new_pinned_member(inner_answer.transaction,
members, handle_list, transaction_list, m_s_a_index, relay)?
            } else {
                show_conference_room_members(inner_answer.transaction,
members, transaction_list, relay)?
            }
        },
        _ => {}
    }

    Ok(())
}
```

ДОДАТОК Б

```
pub fn router(app_state: AppState) -> Router<AppState, axum::body::Body> {
    Router::new()
        .route("/registration", post(registration))
        .route("/verification/:token",
            get(verification)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .route("/authorization", post(authorization))
        .route("/regenerate", get(regenerate))
        .nest("/edit",
            Router::new()
                .route("/",
                    patch(edit)
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
                .route("/password",
                    patch(edit_password)
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
                .route("/picture",
                    patch(edit_picture)
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
                .route("/settings",
                    patch(edit_settings)
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
        )
        .nest("/destroy",
            Router::new()
                .route("/:token",
                    delete(destroy)
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
                .route("/confirmation",
                    get(destroy_confirmation)
```

```
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
        )
        .route("/me",
            get(me)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .nest("/session",
            Router::new()
                .route("/s",
                    get(sessions)
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
                .route("/destroy/:user_session_id",
                    delete(session_destroy)
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
        )
        .route("/unauthorization",
            get(unauthorization)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
}

pub fn router(app_state: AppState) -> Router<AppState, axum::body::Body> {
    Router::new()
        .route("/create",
            post(create)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .route("/join/:room_id",
            get(join)
                .layer(
                    ServiceBuilder::new()
                        .layer(from_fn_with_state(app_state.clone(),
black_list_guard))
                        .layer(from_fn_with_state(app_state.clone(),
room_guard))
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
```

```
                )
        )
        .route("/leave/:room_id",
            get(leave)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .nest("/edit",
            Router::new()
                .route("/:room_id",
                    patch(edit)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), role_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/:room_id/picture",
                    patch(edit_picture)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), role_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/:room_id/settings",
                    patch(edit_settings)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), role_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
        )
        .route("/destroy/:room_id",
            delete(destroy)
                .layer(
                    ServiceBuilder::new()
```

```
                        .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), role_guard))
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
        )
        .route("/me/:tag",
            get(me)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .route("/:room_id",
            get(settings)
                .layer(
                    ServiceBuilder::new()
                        .layer(from_fn_with_state(app_state.clone(),
black_list_guard))
                        .layer(from_fn_with_state(app_state.clone(),
room_guard))
                        .layer(from_fn_with_state(app_state.clone(),
access_guard))
                )
        )
        .route("/s",
            get(rooms)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .route("/search",
            get(room_search)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .route("/c",
            get(room_c)
                .layer(from_fn_with_state(app_state.clone(), access_guard))
        )
        .nest("/:room_id/member",
            Router::new()
                .nest("/edit",
                    Router::new()
                        .route("/:user_id",
                            patch(member_edit)
                                .layer(
                                    ServiceBuilder::new()
```

```
.layer(from_fn_with_state((app_state.clone(), vec!["admin".to_string()]),
exclude_guard))

.layer(from_fn_with_state((app_state.clone(), vec!["admin".to_string(),
"moder".to_string()]), role_guard))

.layer(from_fn_with_state(app_state.clone(), access_guard))
                                )
                        )
                        .route("/:user_id/role",
                            patch(member_edit_role)
                                .layer(
                                    ServiceBuilder::new()

.layer(from_fn_with_state((app_state.clone(), vec!["admin".to_string()]),
role_guard))

.layer(from_fn_with_state(app_state.clone(), access_guard))
                                )
                        )
                )
                .route("/eject/:user_id",
                    get(member_eject)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), exclude_guard))
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string(), "moder".to_string()]), role_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/block/:user_id",
                    get(member_block)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), exclude_guard))
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string(), "moder".to_string()]), role_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
```

```
                    )
                )
                .route("/unblock/:user_id",
                    get(member_unblock)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), role_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/:user_id",
                    get(member)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state(app_state.clone(),
black_list_guard))
                                .layer(from_fn_with_state(app_state.clone(),
room_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/s/:user_id",
                    get(members)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state(app_state.clone(),
black_list_guard))
                                .layer(from_fn_with_state(app_state.clone(),
room_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/m/:user_id",
                    get(member_m)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state(app_state.clone(),
black_list_guard))
                                .layer(from_fn_with_state(app_state.clone(),
```

```
room_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/c/:user_id",
                    get(member_c)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state(app_state.clone(),
black_list_guard))
                                .layer(from_fn_with_state(app_state.clone(),
room_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
                .route("/blocked",
                    get(member_blocked)
                        .layer(
                            ServiceBuilder::new()
                                .layer(from_fn_with_state((app_state.clone(),
vec!["admin".to_string()]), role_guard))
                                .layer(from_fn_with_state(app_state.clone(),
access_guard))
                        )
                )
        )
}

pub fn router(app_state: AppState) -> Router<AppState, axum::body::Body> {
    Router::new()
        .route("/:room_id", get(chat))
        .nest("/:room_id/message",
            Router::new()
                .route("/s",
                    get(messages)
                        .layer(
                            tower::ServiceBuilder::new()
                                .layer(from_fn_with_state(app_state.clone(),
http::black_list_guard))
                                .layer(from_fn_with_state(app_state.clone(),
http::room_guard))
```

```
                                .layer(from_fn_with_state(app_state.clone(),
http::access_guard))
                        )
                )
        )
}

axum::Server::bind(&SocketAddr::from(([0, 0, 0, 0], config.app_port)))
        .serve(
            Router::new()
                .nest("/v1",
                    Router::new()
                        .nest("/user",
controller::user::router(app_state.clone()))
                        .nest("/room",
                            Router::new()

.merge(controller::room::router(app_state.clone()))
                                .merge(
                                    Router::new()
                                        .nest("/chat",
controller::chat::router(app_state.clone()))
                                )
                        )
                )
                .layer(
                    tower_http::cors::CorsLayer::new()

.allow_origin(["http://127.0.0.1:5173".parse().unwrap()])
                )
                .with_state(app_state)
                .into_make_service_with_connect_info::<SocketAddr>()
        )
        .await
        .unwrap();
```

ДОДАТОК В

```
ART TRANSACTION;

CREATE TABLE users (
    id BIGSERIAL,
    username VARCHAR(70) NOT NULL,
    nickname VARCHAR(50) UNIQUE NOT NULL,
    email TEXT NOT NULL,
    password TEXT NOT NULL,
    picture TEXT,
    is_verif_confirmed BOOLEAN NOT NULL DEFAULT true,
    is_banned BOOLEAN NOT NULL DEFAULT false,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE user_tokens (
    id SERIAL,
    verification_token TEXT NOT NULL,
    confirmation_token TEXT,
    user_id BIGINT UNIQUE NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE user_settings (
    id SERIAL,
    is_video_muted BOOLEAN NOT NULL DEFAULT true,
    is_audio_muted BOOLEAN NOT NULL DEFAULT false,
    user_id BIGINT UNIQUE NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE user_sessions (
    id INT,
    token TEXT NOT NULL,
    ip_address TEXT NOT NULL,
    operating_system TEXT,
    device TEXT,
    user_id BIGINT NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
```

```
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE rooms (
    id SERIAL,
    tag TEXT UNIQUE NOT NULL,
    key VARCHAR(150) NOT NULL,
    name VARCHAR(150) NOT NULL,
    picture TEXT,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE room_settings (
    id SERIAL,
    is_private BOOLEAN NOT NULL DEFAULT true,
    is_video_muted BOOLEAN NOT NULL DEFAULT false,
    is_audio_muted BOOLEAN NOT NULL DEFAULT false,
    is_chat_muted BOOLEAN NOT NULL DEFAULT false,
    room_id INT UNIQUE NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE members (
    is_video_muted BOOLEAN NOT NULL DEFAULT false,
    is_audio_muted BOOLEAN NOT NULL DEFAULT false,
    is_chat_muted BOOLEAN NOT NULL DEFAULT false,
    user_id BIGINT,
    room_id INT,
    role_id INT NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE black_lists (
    user_id BIGINT,
    room_id INT,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE messages (
```

```
    id SERIAL,
    data TEXT NOT NULL,
    user_id BIGINT,
    room_id INT NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE roles (
    id SERIAL,
    alias TEXT UNIQUE NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

ALTER TABLE users ADD CONSTRAINT pk_users_id PRIMARY KEY (id);
ALTER TABLE user_settings ADD CONSTRAINT pk_user_settings_id PRIMARY KEY (id);
ALTER TABLE user_sessions ADD CONSTRAINT pk_user_sessions_id PRIMARY KEY (id);
ALTER TABLE user_tokens ADD CONSTRAINT pk_user_tokens_id PRIMARY KEY (id);
ALTER TABLE rooms ADD CONSTRAINT pk_rooms_id PRIMARY KEY (id);
ALTER TABLE room_settings ADD CONSTRAINT pk_room_settings_id PRIMARY KEY (id);
ALTER TABLE members ADD CONSTRAINT pk_members_user_id_room_id PRIMARY KEY
 (user_id, room_id);
ALTER TABLE black_lists ADD CONSTRAINT pk_black_lists_user_id_room_id PRIMARY
 KEY (user_id, room_id);
ALTER TABLE roles ADD CONSTRAINT pk_roles_id PRIMARY KEY (id);
ALTER TABLE messages ADD CONSTRAINT pk_messages_id PRIMARY KEY (id);

ALTER TABLE user_settings ADD CONSTRAINT fk_user_settings_user_id FOREIGN KEY
 (user_id) REFERENCES users (id) ON DELETE CASCADE;
ALTER TABLE user_sessions ADD CONSTRAINT fk_user_sessions_user_id FOREIGN KEY
 (user_id) REFERENCES users (id) ON DELETE CASCADE;
ALTER TABLE user_tokens ADD CONSTRAINT fk_user_tokens_user_id FOREIGN KEY
 (user_id) REFERENCES users (id) ON DELETE CASCADE;
ALTER TABLE room_settings ADD CONSTRAINT fk_room_settings_room_id FOREIGN KEY
 (room_id) REFERENCES rooms (id) ON DELETE CASCADE;
ALTER TABLE members ADD CONSTRAINT fk_members_user_id FOREIGN KEY (user_id)
 REFERENCES users (id) ON DELETE CASCADE;
ALTER TABLE members ADD CONSTRAINT fk_members_room_id FOREIGN KEY (room_id)
 REFERENCES rooms (id) ON DELETE CASCADE;
ALTER TABLE members ADD CONSTRAINT fk_members_role_id FOREIGN KEY (role_id)
 REFERENCES roles (id);
ALTER TABLE black_lists ADD CONSTRAINT fk_black_lists_user_id FOREIGN KEY
```

```
(user_id) REFERENCES users (id) ON DELETE CASCADE;
ALTER TABLE black_lists ADD CONSTRAINT fk_black_lists_room_id FOREIGN KEY
(room_id) REFERENCES rooms (id) ON DELETE CASCADE;
ALTER TABLE messages ADD CONSTRAINT fk_messages_user_id FOREIGN KEY (user_id)
REFERENCES users (id) ON DELETE SET NULL;
ALTER TABLE messages ADD CONSTRAINT fk_messages_room_id FOREIGN KEY (room_id)
REFERENCES rooms (id) ON DELETE CASCADE;

CREATE FUNCTION trigger_an_update_on_the_updated_at_column()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at := NOW();
    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION trigger_an_insert_on_the_user_settings_table()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO user_settings (user_id) VALUES (NEW.id);
    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION trigger_an_insert_on_the_room_settings_table()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO room_settings (room_id) VALUES (NEW.id);
    RETURN NEW;
END
$$ LANGUAGE plpgsql;


DO $$
DECLARE
    t TEXT;
BEGIN
    FOR t IN SELECT table_name FROM information_schema.columns WHERE
column_name = 'updated_at'
    LOOP
        EXECUTE FORMAT('CREATE TRIGGER update_updated_at_column BEFORE UPDATE
ON %I FOR EACH ROW EXECUTE PROCEDURE
trigger_an_update_on_the_updated_at_column();', t, t);
```

```
    END LOOP;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER insert_in_user_settings_table AFTER INSERT ON users FOR EACH
ROW EXECUTE PROCEDURE trigger_an_insert_on_the_user_settings_table();

CREATE TRIGGER insert_in_room_settings_table AFTER INSERT ON rooms FOR EACH
ROW EXECUTE PROCEDURE trigger_an_insert_on_the_room_settings_table();

CREATE VIEW detailed_members AS SELECT users.username, users.nickname,
users.picture, roles.alias as role, CASE WHEN members.is_video_muted THEN
true WHEN user_settings.is_video_muted THEN true ELSE false END as
is_video_muted, CASE WHEN members.is_audio_muted THEN true WHEN
user_settings.is_audio_muted THEN true ELSE false END as is_audio_muted,
members.is_chat_muted, members.created_at, members.user_id, members.room_id
FROM members JOIN users ON members.user_id = users.id JOIN roles ON
members.role_id = roles.id JOIN user_settings ON users.id =
user_settings.user_id;

CREATE VIEW detailed_rooms AS SELECT rooms.id, rooms.tag, rooms.key,
rooms.name, rooms.picture, rooms.created_at, room_settings.is_private FROM
rooms JOIN room_settings ON rooms.id = room_settings.room_id;

CREATE VIEW detailed_messages AS SELECT messages.id, messages.data,
messages.user_id, messages.room_id, messages.created_at, users.username,
users.nickname, users.picture FROM messages LEFT JOIN users ON
messages.user_id = users.id;

COMMIT;
```