НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Вебзастосунок для проведення онлайн-занять»

Виконав:
студент _IV_ курсу, групи_ІВ-72__

___Обух Василь Іванович_____ _____
(прізвище, ім'я, по батькові) (підпис)

Керівник _____асистент Сергієнко А.А._____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Консультант __н. контроль_ проф. д.т.н. Сімоненко В. П. _____
(назва розділу) (вчені ступінь та звання, прізвище, ініціали) (підпис)

Рецензент _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2021 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Освітньо-професійна програма

«Комп'ютерні системи та мережі» спеціальність

123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ ***Стіренко С.Г.***
(підпис)
"___"_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту

_____Обуха Василя Івановича_____

1. Тема проєкту «*Вебзастосунок для проведення онлайн-занять»*

керівник проєкту _Сергієнко Анастасія Анатоліївна_, асистент.
Затверджені наказом по університету від *« 11 »* *05* 2021р. №1139-с

2. Термін здачі студентом закінченої роботи_____*2021р.*

3. Вихідні дані до проєкту *технічне завдання, теоретичні дані.*

4. Зміст пояснювальної записки: *опис предметної області, дослідження та створення вебзастосунку, функціональна програма з її об'єктивним зображенням та використаний програмний код при її створенні.*

5. Консультант роботи, з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	д.т.н., проф. Сімоненко В.П.		

6. Дата видачі завдання _____ *року*

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту(роботи)	Примітки
1.	*Затвердження теми роботи*	10.12.2021-15.12.2021	
2.	*Вивчення та аналіз завдання*	15.12.2021-15.03.2021	
3.	*Розробка архітектури та загальної структури систем*	15.03.2021-25.03.2021	
4.	*Розробка структур окремих підсистем*	25.03.2021-5.04.2021	
5.	*Програмна реалізація системи*	5.04.2021-15.04.2021	
6.	*Оформлення пояснювальної записки*	15.04.2021-20.05.2021	
7.	*Передзахист*	23.05.2021	
8.	*Захист*	16.06.2021	

Студент _____

(підпис)

Керівник _____

(підпис)

Анотація

В бакалаврському дипломному проєкті розроблено вебзастосунок для проведення онлайн занять.

Запропонований вебзастосунок може бути використаний для фактичного зв’язку користувачів в форматі відео-зв’язку, з використанням обміну звуків та текстового поля. Продукт був створений з використанням React, Redux-saga, TypeScript у фронт-енді та Nestjs, MongoDB у бек-енді, який до того ж створений на основі мікросервісів та Doker-контейнерів.

Крім цього, пристосована технологія WebSocket і RTC Peer Connection зі збереженням фото додатків у сервісі Cloudinary.

Annotation

The bachelor's degree project has developed a web application for online classes.

The proposed web application can be used to actually communicate with users in video format, using audio sharing and a text box. The product was created using React, Redux-saga, TypeScript in front-end and Nestjs, Mongodb in back-end, which is also created on the basis of microservices and Doker-containers.

In addition, custom technology WebSocket and RTC Peer Connection with saving photo applications in the service Cloudinary.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4	ІАЛЦ.467100.001 ОА	Опис альбому	1	
2	А4	ІАЛЦ.467100.002 ТЗ	Технічне завдання	4	
3	А4	ІАЛЦ.467100.003 ПЗ	Пояснювальна записка	55	
4	А4	ІАЛЦ.467100.004 Д1	Схема внутрішнього взаємозв'язку користувачів	1	
5	А4	ІАЛЦ.467100.004 Д2	Схема алгоритму підготовки клієнтських частин перед формуванням сигнальних даних	1	
6	А4	ІАЛЦ.467100.004 Д3	Структурна схема сервісів	1	
7	А4	ІАЛЦ.467100.004 Д4	Лістинг програми	17	

					ІАЛЦ.467100.001 ОА
Розробив	*Обух В.І.*			**Вебзастосунок для проведення онлайн-занять** Технічне завдання	Літ: — / Аркушів: 1 / 1
Перевірив	*Сергієнко А.А.*				
Н. Контр.	Сімоненко В. П.				*НТУУ «КПІ» ФІОТ*
Затвердив					*Група ІВ-72*

ТЕХНІЧНЕ ЗАВДАННЯ

до дипломної роботи
освітньо-кваліфікаційного рівня___бакалавр___

на тему: “Вебзастосунок для проведення онлайн-занять”.

Київ – 2021 року

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: вебзастосунок для проведення онлайн-занять.

Область застосування: практичне використання людьми при користуванні мережею інтернет.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», затверджене кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проєкту є розробка вебзастосунку для проведення онлайн-занять

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література з теорії і практики програмування, бакалаврські роботи інших студентів, публікації в Інтернеті з даних питань.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розроблюваного продукту

- Самостійність курсу – чіткі контури предмету вивчення
- Самодостатність – курс повинен містити необхідну інформацію та дані, які дозволяють у повній мірі розкрити мету курсу
- Коректність та актуальність інформації, які охоплює курс

5.2. Вимоги до програмного забезпечення

- Unix-подібна ОС або Windows 7/8/10
- Docker

5.2. Вимоги до апаратної частини

- Комп'ютер на базі процесора Intel Core i5 та вище
- Оперативної пам'яті не менше 8 Гбайт
- Вільний простір жорсткого диску не менше 20 Гбайт

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення літератури	_____
Складання і узгодження технічного завдання	_____
Створення модулів системи, що розробляється	_____
Тестування окремих модулів системи	_____
Допрацювання, налагодження і виправлення помилок	_____
Оформлення документації дипломної роботи	_____

ПОЯСНЮВАЛЬНА ЗАПИСКА
до дипломного проєкту
на тему: «Вебзастосунок для проведення онлайн-занять»

Київ – 2021 року

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

DOM – (англ. Document Object Model) – об'єктна модель документа.

JSX – (англ JavaScript XML) це розширення синтаксису JavaScript.

СУБД − (англ. Database Management System, сокр. DBMS) − система керування (управління) базами даних.

MVC − (англ. Model-View-Controller) − Модель-Подання-Контролер.

VoIP − (англ. voice over IP — голос через IP) — технологія передачі медіа-даних у реальному часі за допомогою сімейства протоколів TCP/IP.

P2P – (англ. Peer-to-peer) — варіант архітектури системи, в основі якої стоїть мережа рівноправних вузлів.

CPU – (англ. central processing unit) − центральное обрабатывающее устройство.

UDP − (англ. User Datagram Protocol) − протокол пользовательских дейтаграмм.

TCP – (англ. Transmission Control Protocol) − протокол управління передачею даних.

TLS − (англ. transport layer security) — протокол захисту транспортного рівня.

HTTPS – (англ. HyperText Transfer Protocol Secure) — розширення протоколу HTTP для підтримки шифрування з метою підвищення безпеки.

HTTP − протокол для передачі гіпертекстових документів.

GUI − (англ. graphical user interface) – система засобів для взаємодії користувача з комп'ютером.

ВСТУП

Розвиток технологій безпосередньо впливає на розвиток всіх інших галузей. Не становить виняток і сфера освіти. Останнім часом світ переживає нелегкі часи. Найпоширенішою темою обговорення є пандемія, яка, до того ж започаткувала масовий перехід від офлайн до онлайн діяльності, у тому числі і освітнього процесу. **Актуальним** постало питання щодо обрання відповідної платформи для проведення таких занять.

Проблематика даної теми зводиться до обрання відповідної програми, додатку, вебзастосунку, тощо, яка б відповідала сучасним вимогам та безпосередньо підходила для відповідного навчального процесу.

Кінцева **мета** обраної теми – вирішити нагальні питання, щодо якості програм зв'язку, які використовуються під час проведення онлайн-занять.

Для досягнення поставленої мети до уваги **пропонується** вироблений вебзастосунок для проведення онлайн-занять, який акумулює в собі позитивні якості обраних до огляду програм.

За **результатами** був створений відповідний вебзастосунок, який відповідає сучасним вимогам ринку та до того ж його можливо використовувати практично.

РОЗДІЛ 1
ОГЛЯД НАЯВНИХ РІШЕНЬ МІГРАЦІЙ ДАНИХ

1.1 Історія виникнення перших спроб дистанційного навчання.

Для більшого розуміння концепції побудови вебзастосунку, спочатку слід проаналізувати його історію розвитку (становлення). Ймовірно, що аналіз проблематики та перешкод, які виникали протягом такого становлення наддасть змогу більш точно та якісно відобразити функціонал, який так чи інакше полегшить його використання.

Під вебзастосунком більшою мірою розуміють застосунок, який має дуальну систему (архітектуру), яка побудована на двох важливих компонентах – Клієнт (Користувач) та Сервер. Безпосередній взаємозв'язок між компонентами обслуговує веб-браузер.

Однак, така, звичайна модель, існувала не завжди. Свій початок вебзастосунок бере ще близько з XIX сторіччя. Саме тоді бере свій початок розвитку найперші так званні «вебзастосунки». Все розпочалось з реклами в шведській газеті в 1833 році, яка радувала можливість вивчити «Композицію через поштову службу» [1].

У 1840 році недавно створена пенні в Англії дозволила Ісааку Пітману дати коротку інструкцію по листуванню. Три роки по тому навчання було формалізовано з підставою Товариства фонографічних переписів, попередником коледжів сера Ісаака Пітмана. У Німеччині його створили Чарльз Туссен і Густав Лангеншайдт, які викладали мову в Берліні.

Дослідження листування перетнуло Атлантику в 1873 році, коли Анна Еліот Тікнор заснувала суспільство в Бостоні, щоб заохочувати навчання вдома, тобто дистанційно.

Студенти класичної навчальної програми (в основному жінки) щомісяця листувалися з учителями, які надавали орієнтири для подальшого читання. З 1883 по 1891 рік академічні ступені були дозволені штатом Нью-

Йорк через Коледж вільних мистецтв Чаутаукі для студентів, які завершили необхідні літні інститути та заочні курси [1].

В Університеті Вісконсіна розвиток «короткого курсу» і фермерських інститутів в 1885 році стало основою для розширення університету. Шість років по тому університет оголосив програму заочного навчання під керівництвом видатного історика Фредеріка Джексона Тернера.

Однак, як і в Чиказькому університеті, інтерес викладачів зовсім зник. Вільям Рейні Харпер, професор Єльського університету, який очолював програму, був розлючений в свою підтримку заочного навчання і впевнений у майбутній життєздатності нової освітньої форми: студент, який підготував певні уроки в заочній школі, знає більше про предмет і знає, що це краще, ніж учень, який перебував під час очного навчання в класі.

Кореспондентське дослідження продовжувало розвиватися в Великобританії зі створенням ряду заочних закладів, таких як Коледж Скеррі в Единбурзі в 1878 році і Університетський коледж кореспонденції в Лондоні в 1887 році.

У той же час розширення університетського руху в Сполучених Штатах і Англії сприяло спосіб відповідності. Серед піонерів в цій області були Іллінойс Весліан в 1877 році і Університетський відділ університету Чикаго в 1892 році. Іллінойс Весліан запропонував ступінь бакалавра і ступінь доктора наук в рамках програми, що моделюється за моделлю Оксфорда, Кембриджа і Лондона.

Між 1881 і 1890 роками навчалося 750 студентів. Однак стурбованість з приводу якості програми викликала рекомендацію про її припинення. Відділ заочного навчання Чиказького університету був успішним, принаймні, з точки зору кількості. Щороку 125 викладачів навчали 3000 студентів, що навчаються на 350 курсах.

Проте, ентузіазм в університеті для програми ослаб, частково через фінансові причини. Інститут Біблії Муді, заснований в 1886 році, сформував

заочне відділення в 1901 році, яке триває досі, з рекордною кількістю понад 1 мільйон учнів з усього світу.

Таким чином, на цьому етапі розвитку сформувалась система дистанційної освіти з використанням первинного способу – пошти, яка у свою чергу надала змогу розвинути освітянські послуги у бік дистанційної та заочної направленості.

Наступним етапом стало стійке розширення дистанційної освіти без радикальних змін в структурі, але з використанням більш досконалих методів і засобів масової інформації [1].

Аудіозапис використовувалася в інструкціях для сліпих і викладання мови для всіх учнів. Лабораторні комплекти використовувалися в таких областях, як електроніка і радіотехніка.

Практично всі великомасштабні організації дистанційного навчання були приватними заочними школами. Наприклад, у Сполучених Штатах успіхи в області технологій електронних комунікацій допомогли визначити домінуючу середу дистанційного навчання.

У 1920-х роках в навчальних закладах було побудовано щонайменше 176 радіостанцій, хоча більшість з них зникло до кінця десятиліття.

Ті, що залишились були в основному в коледжах, які надають гранти.

Західний резервний університет першим запропонував безперервну серію курсів, починаючи з 1951 року. Sunrise Semester був широко відомої телевізійної серією курсів в коледжі, пропонованих Нью-Йоркським університетом на CBS з 1957 до 1982 року.

Технологія супутників, розроблена в 1960-х роках і зроблена рентабельною в 1980-х роках, дозволила швидко поширити навчальний телебачення. Федеративно фінансуються експерименти в Сполучених Штатах і Канаді, такі як Аппалачский освітній супутник (1974-1975 роки), продемонстрували здійсненність інструкцій по супутниковому передачі. Однак ці ранні експерименти були голосно розкритиковані за те, що вони погано сплановані. Пізніші спроби дистанційного навчання пройшли

успішніше. Перша державна освітня супутникова система «Learn / Alaska» була створена в 1980 році. Вона щодня пропонувала 6 годин навчального телебачення в 100 селах, деякі з яких були доступні тільки по повітрю. Приватна мережа TI-IN, Сан-Антоніо, Техас, з 1985 року надає широкий спектр курсів по супутникових каналах в середніх школах Сполучених Штатів.

На цьому етапі розвитку дистанційної освіти вже простежуються відповідні технології, які покращують всю концепцію освіти. Такі технології пропонують нові шляхи вирішення проблем, які виникають під час навчання.

Скоріш за все, сучасні умови – глобальні катаклізми, стихійні явища, врешті-решт пандемія, яка триває досить довго – ці і багато інших чинників зумовили частого, а в деяких випадках і повністю перейти на дистанційну освіту [1].

Слід також вказати, дистанційна освіта − це загальний термін, який включає в себе ряд стратегій навчання і навчання, що використовуються коледжами листування, відкритими університетами, відділами дистанційного навчання звичайних університетів і навчальними підрозділами дистанційного навчання організацій приватного сектора.

Одне з найбільш повних визначень дистанційного навчання зробив Десмонд Кіган, британський воєнний історик у своїй праці «Основи дистанційної освіти» 1996-го року, запропонував шість основних визначальних елементів дистанційного навчання. Він зазначив, що дистанційна освіта характеризується поділом учня і наставника в протилежність особистого навчання, впливу освітньої організації, яка відрізняє дистанційна освіта від приватного навчання, використання технічних засобів інформації, наприклад, друк, аудіо або вебсайт, щоб об'єднати викладача і учня, забезпечити двосторонній зв'язок, щоб учень міг вести діалог з викладачем, можливість випадкових зустрічей для цілей взаємодії та самовизначенності характеру учня [2].

До речі, фактор дистанційності простежується не тільки в освітянських процесах, інший приклад − бізнес. Бізнес повинен сам себе окупати та ще й приносити прибуток. І звісно, ті бізнес моделі які мали змогу в теперішніх умовах перейти на дистанційну, або краще вже казати онлайн-роботу – безпосередньо перейшли на неї.

З огляду на наведені історичні факти, помилковим буде додаткове висвітлення інформації з приводу розробки та/або започаткування певних програм, або будь-якого іншого інструментарію, задля проведення онлайн занять.

Надалі, пропонується розглянути деякі відомі бренди (програми), які стали в нагоді, у тому числі, під час онлайн-навчання.

1.2 Skype

Створений ще в далекому 2003 році Skype дає змогу робити дзвінки з хорошою чутністю, не зважаючи на геолокацію користувача, миттєво надсилати повідомлення, вільно обирати текстовий чи голосовий зв'язок, робити дзвінки із відео-зв’язком, створювати відео-конференції, надсилати файли і дзвонити на мобільні та міські номери, що становить досить малу частину тієї суми, в порівнянні з використанням звичайного телефону [6].

Слід вказати, що Skype – це окрема прикладна програма. Тобто, для її використання слід завантажити відповідну копію програми на свій девайс для подальшого використання.

Skype − це різновид клієнта для голосового зв'язку через Інтернет (VoIP), заснованого на піринговій (P2P) технології.

VoIP телефонія (від англ. Voice over IP, скорочено VoIP) – технологія для організації голосового спілкування, що досягається завдяки використанню Інтернету, або будь-якої іншої мережі, що будується на основі протоколу IP. Тобто, іншими словами, маючи комп'ютер, що підключений до Інтернету, можна виступати в якості телефону, значно підвищуючи економію при дзвінках, особливо міжнародних.

Принцип Peer to Peer, в свою чергу забезпечує обмін даними в мережі на пряму між комп'ютерами, без необхідності наявності будь-якої централізованої серверної структури. Системою Skype використовується лише один центральний сервер, що містить інформацію про всі облікові записи користувачів. Даний сервер потрібний лише для пошуку необхідного абонента та встановлення зв'язку з ним, надалі обмін даними між комп'ютерами абонентів здійснюється безпосередньо ними самими (рис. 1.1) [3;5].

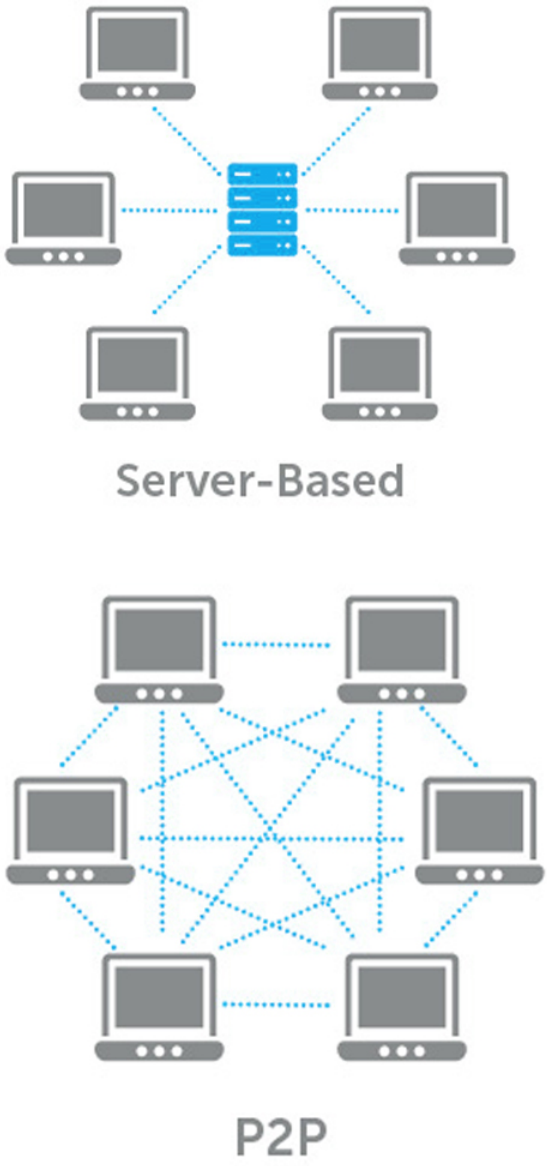


Рис. 1.1 Приклад взаємозв’язку в системі P2P.

Мовний інструментарій програми містить в собі Embarcadero Delphi, Objective-C, C++, Object Pascal и JavaScript.

Архітектура Skype, як зауважив Mark Gillett, CVP Skype Product Engineering & Operations побудована на supernodes'сах. Це супер-вузли. [4; 5]

У мережі Skype існує два типи машин: звичайний хост (Skype-клієнт) і супер-вузол (Super Node, SN). Супер-вузли являється кінцевим адресатом для звичайних хостів. Тобто звичайні хости підключаються до супер-вузлів. Супер-вузлом може стати будь-який комп'ютер із належною конфігурацією обладнання та з публічним IP. Звичайний хост − це комп'ютер звичайного користувача, який підключений до мережі і на який встановлено застосунок Skype.

Поняття супер-вузлів було введено для піринговых мереж. Вони дозволили знизити затримку передачі файлів, покращити показники пошуку, підвищити масштабованість мережі, а також розширили можливість відновлення перерваних закачувань і одночасного завантаження сегментів одного файлу з декількох джерел, так званих бенкетів. Супер-вузли відповідають також за «глобальну індексацію». Це дає змогу виконувати пошук інших користувачів в мережі. В основному супер-вузли допомагають звичайним хостам з'єднуватися один з одним і служать ефективними провідниками зашифрованого мережевого трафіку.

«Супер-вузли» допомагають клієнтам Skype знайти один одного в мережі, щоб вони могли зв'язатися для дзвінка по Skype. Простіше кажучи, супер-вузли виступають в ролі розподіленого довідника користувачів Skype. Дзвінки зі Skype на Skype не проходять через центри обробки даних і «супер-вузли» не беруть участь в передачі аудіо або відео між клієнтами Skype.

Дзвінки або зв'язок між учасниками, що беруть участь в розмові здійснюється за допомогою цих вузлів напряму навіть під час їх розмови. У деяких випадках Skype використовує сервери для надання допомоги в здійсненні дзвінків: наприклад, сервер використовується для повідомлення клієнта про те, що ініціюється новий виклик з ним, і там, де не запущено

повноцінне додаток Skype (наприклад, пристрій призупинено, "спить" або вимагає повідомлення про вхідний дзвінок для активації), а також при групових відеодзвінках, де сервер агрегує медіа-потоки (відео) від декількох клієнтів і направляє їх до клієнтів, у яких може не бути достатньої пропускної спроможності для встановлення з'єднання з усіма учасниками.

На початку становлення програми супер-вузли містились на серверах, однак, наразі всі супер-вузли містяться у хмарних сховищах. Що у свою чергу забезпечує швидкий доступ до виникаючих проблеми, а також дає можливість швидко провадити оновлення програмного забезпечення.

Безпеку та шифрування інформації програми Skype наразі забезпечує протокол «Signal Protocol» (алгоритм наскрізного шифрування, розроблений компанією Open Whisper Systems) [5].

1.3 Discord

Наступна програма запропонована для огляду – Discord.

Створений в 2015 році, Discord – являє собою найпростіший спосіб щоденного спілкування для будь-яких компаній: шкільних гуртків, ігрових груп, міжнародних співтовариств художників або просто друзів [7].

Слід вказати, що Discord – існує як відокремленим додатком, а також, і вебзастосунком, на відміну від вже згаданого Skype. Тобто, для її використання можливо як завантажити відповідну копію програми на свій девайс для подальшого використання, або використовуючи браузер перейти на відповідний сайт та використовувати функціонал.

Як зауважив у своєму блозі Jozsef Vass, Senior Software Engineer at Discord − аудіо та відео в Discord працює на WebRTC. Таким чином, браузерний додаток покладається на реалізацію WebRTC в браузері. Однак додатки для десктопів, iOS і Android використовують єдиний мультимедійний движок C ++, побудований поверх власної бібліотеки WebRTC, спеціально адаптованої до потреб наших користувачів. Це означає, що деякі функції в додатку працюють краще, ніж в браузері. [8;9].

Мовний інструментарій програми містить в собі JavaScript, React, Elixir, Rust.

Архітектура побудована на моделі «Сервер − клієнти», тобто кожен клієнт підключається безпосередньо до сервера. Сервер включає в собі 3 (три) сервіси: Discord Gateway, Discord Guilds и Discord Voice. Кожний елемент виконує свою роль, тобто є розподіленим.

Коли ви в мережі, ваш клієнт підтримує з'єднання WebSocket до шлюзу Discord Gateway (називають його шлюзовим підключенням WebSocket). Через це з'єднання ваш клієнт отримує події, пов'язані з групами і каналами, текстові повідомлення, пакети присутності і т. д.

При підключенні до голосового каналу статус з'єднання відображається об'єктом стану голосового зв'язку. Клієнт оновлює цей об'єкт по шлюзового підключення.

При підключенні до голосового каналу вам призначають один з серверів Discord Voice. Він відповідає за передачу звуку кожному учаснику каналу. Всі голосові канали в групі призначаються одного сервера. Якщо ви перший в чаті, сервер Discord Guilds відповідає за призначення сервера Discord Voice всій групі за допомогою описаного нижче процесу.

Сервер Discord Guilds стежить за системою пошуку послуг і призначає групі найменш використовуваний сервер Discord Voice в даному регіоні. Коли він обраний, всі об'єкти стану голосового зв'язку (також підтримувані сервером Discord Guilds) передаються на сервер Discord Voice, щоб той міг встановити переадресацію аудіо/відео. Клієнти повідомляються про обраний сервері Discord Voice. Тоді клієнт відкриває друге з'єднання WebSocket з голосовим сервером (називають його голосовим з'єднанням WebSocket), яке використовується для настройки переадресації мультимедіа та індикації мови [8].

Також слід звернути увагу на використання СУБД. Як зазначає Stanislav Vishnevskiy, CTO at Discord «Початкову версію Discord написали швидше ніж за два місяці на початку 2015 року» [10].

Звісно що однією з проблем ставало вмісте сховище або його аналог. Використання СУБД – MongoDB.

З розвитком структури Discord почали траплятись деякі проблеми, СУБД не витримувала навантаження.

Наступною використаною СУБД стала – Cassandra, вона повністю відповідала вимогам сучасності.

До речі, у розробці та майбутніх оновленнях стоїть Scylla, − це СУБД, сумісна з Cassandra і написана на C ++. В нормальній роботі Ноди Cassandra в реальності споживають небагато ресурсів CPU, проте в непікові годинник під час лагодження Cassandra (протівоентропійний процес) вони досить сильно залежать від CPU, а час лагодження зростає в залежності від кількості даних, записаних з моменту минулого лагодження. Scylla обіцяє значно збільшити швидкість лагодження.

Шифрування Discord здійснюється швидким інструментом − Salsa20.

1.4 Zoom

Ще одна програма запропонована для огляду – Zoom.

Створений в 2013 році, Zoom об’єднує відеоконференції, сучасне рішення для хмарних телефонів, групові обміни повідомленнями та програмну відеоконференцію кімнати в одну зручну платформу.

Zoom, як і попередньо розглядуваний Discord, − існує як відокремленим додатком, а також, і вебзастосунком.

Zoom побудована та має хмарну архітектуру. Архітектура починається з основного «рівня» − інтелектуального транспорту [11].

Це рівень, який вирішує, чи UDP, TCP, TLS або HTTPS на клієнтському рівні є найкращим досвідом для підключення на основі різних налаштування проксі і необхідність проходження брандмауерів.

Вбудований алгоритм виявить клієнта, його місцезнаходження та призначить закриті канали для подальшого зв’язку між клієнтами.

Наступний рівень − рівень реактивної якості обслуговування.

Тобто, допомога клієнтам з різних питань надається миттєво та реагує в реальному часі.

Цей рівень не тільки контролює пропускну здатність клієнта, втрату пакетів, затримку та тремтіння, але також збирає використання клієнтом центрального процесора, пам'яті та мережі. Він повідомляє верхній рівень, щоб зробити найкращі дії всередині.

В Zoom реалізована «адаптивна технологія». Адаптивний кодек Zoom на рівні сеансу, на відміну від інших постачальників, створюється із власним кодуванням. Множинні рівні навколо цього кодека оптимізують частоту кадрів відео та роздільну здатність, що забезпечують вищу якість та надійність для різних мережевих середовищ, а також різних пристроїв [11].

Додатково кодек використовує кілька потоків, дозволяючи програмі перемикатись між певними потоками, щоб забезпечити отримання відео найкращої якості.

Як і у багатьох програмах – реалізація багатопотоковість зв’язку [11].

Клієнти підключаються до серверу. Один сервер може обробляти 2000 клієнтів. Zoom використовує глобально розподілену технологію, засновану на геолокація користувача та оптимізований мережевий шлях. Зустріч учасники завжди підключені до сусіднього центру обробки даних і присвоюється найменш завантаженому серверу.

З іншої сторони, учасники зустрічі будуть об'єднані на один і той же сервер, якщо вони знаходяться в одному місці. Ця архітектура також задовольняє гнучкому приміщенню та гібридні моделі розгортання та забезпечує каскадне забезпечення транспорту для підприємств.

Додатково в Zoom реалізований інструментарій – «Сервер зустрічей». Цей сервер є MMR (мультимедійний маршрутизатор), такий MMR згрупований в «Зони зустрічей» [11].

Відповідні контролери зон керують усіма MMR та повідомляють про їх статус Глобальному хмарному контролеру для кожної Зони Зустрічі. Зони Зустрічі дублюються для кожного центру обробки даних з точно такою ж

архітектурою. Тобто, загально є три окремі рівні (MMR, Zoom Controller та Global Cloud Controller), які використовуються для збалансування ресурсів у різних місця.

Якщо на зустрічі є лише два учасники, сеанс використовуватиме однорангові з'єднання для чудової швидкості та надійності.

Пропозиція локального розгортання забезпечує вищий ступінь безпеки та продуктивності. Ця можливість дає можливість сегментувати користувачів як у локальних, так і у хмарних середовищах.

Пропозиція гібридного варіанту доставки забезпечує вищий ступінь гнучкості та продуктивності. Внутрішні користувачі підключаються до гібридних серверів, тоді як зовнішні користувачі автоматично підключаються до загальнодоступної хмари.

Як хмарні, так і локальні рішення розроблені з відмовними механізмами та механізмами балансування навантаження при розгортанні. Збільшити моніторний рівень зони з декількома віртуальними машинами, і якщо зона наближається до порогового значення або не вдається, вона перейде до наступної зони.

Подібним чином, на рівень віртуальної машини, якщо віртуальна машина виходить з ладу або наближається до порогового значення, підключення переходить до наступної віртуальної машини.

У Zoom є брокери та сервери зв'язку, розподілені між безліччю взаємопов’язаних центрів обробки даних по всьому світу, включаючи супер-вузли в Китаї та Індії.

Під час пошуку інформації про більш детально описаний програмний код, така інформація була дуже поверхово описана. Конкретного язику програмування не було відображено на офіційних сайтах Zoom [12].

За цим, до проведеного аналізу його архутектури звернулись до технічної документації, яка була описана вище.

Однин із відкритих форумів містив інформацію про використаний мовний інструментарій для розробки Zoom.

Однак, джерело не є офіційним, і людина, яка надала цю інформацію не є співробітником компанії.

За цим, беремо таку інформацію як оглядову.

Мовний інструментарій містить в собі C++, JavaScript, Java, Objective-C.

Шифрування використане в Zoom є наскрізним (E2EE) [13].

1.5 Порівняння наведених програм

Таблиця 1.1 – Порівняння наведених програм

Критерій / Назв. програм	Функціонал	Популярність	Модель використання	На чому написаний
Skype	Середній	Середня	Окрема програма	Embarcadero Delphi, Objective-C, C++, Object Pascal и JavaScript.
Discord	Великий	Популярна	І програма і вебзастосунок	JavaScript, React, Elixir, Rust, Cassandra.
Zoom	Малий	Популярна	І програма і вебзастосунок	*Невідомо, (C++, JavaScript, Java, Objective-C.)*

Таблиця 1.1 демонструє порівняльні характеристики для кожної з наведених програм, а саме: функціонал, популярність, модель використання, а також мовний інструментарій використаний при написання відповідної програми..

Спираючись на такий огляд, можливо перейняти відповідні якості з цих програм, та можливо викоренити недоліки під час написання власного вебзастосунку.

ВИСНОВКИ ДО РОЗДІЛУ 1:

В першому розділі для огляду було наведені найперші можливі шляхи становлення дистанційного навчання, яке зумовило розвиток усієї індустрії в цілому.

До того ж, розвиток технологій зв’язку надав певний розвиток для використання відповідних програм для дистанційного, або навіть онлайн навчання.

Додатково, було розглянуто програми, які використовують у тому числі, в навчальних процесах.

Наведені їх архітектури, їх певна взаємодія внутрішніх процесів, а також мовний інструментарій.

Таким чином, спираючись на наведену інформацію, можливо уявити яким повинен бути вебзастосунок, аби відповідати сучасним вимогам, та зайняти високу ланку під час його перенесення у загальний доступ.

РОЗДІЛ 2
РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ ПРОВЕДЕННЯ ОНЛАЙН-ЗАНЯТЬ

Під час безпосереднього написання вебзастосунку для проведення онлайн занять, був використаний відповідний мовний інструментарій.

Нижче за текстом пропонується стисло розглянути кожний використаний інструмент. Виокремлення його позитивних якостей, а також те, як він зв’язаний із іншими інструментами.

2.1 Front-end

2.1.1 Використання React

React – це бібліотека, створена компанією Facebook для мови програмування JavaScript, має відкритий вихідний код. Дана бібліотека була, в першу чергу, створена для однієї мети – ефективно та із простим інтерфейсом взаємодії виконувати задачі пов’язані із розробкою користувацького інтерфейсу. Якщо брати до уваги архітектурний шаблон Model View Controller, то React виконує функції View, тобто відповідає за візуальне представлення даних.

React було створено із метою плавного переходу на нову платформу, тому розробники надали інструментарій для використання вже існуючих компонентів, без необхідності переписувати існуючий функціонал. Завдяки цьому багато проектів перейшли на React платформу, що значно підвищило швидкість відгуку та плавності інтерфейсу.

Зовнішній вигляд, який написаний на React’і з боку програмного коду складається ніби з окремих елементів, їх ще називають компонентами.

Створення компонентів в React відбувається значно простіше, ніж в альтернативних фреймворках, оскільки для відображення розмітки

використовується JSX, що являє собою комбінацію HTML та JavaScript, що таким чином значно розширює синтаксис JavaScript.

На основі інтерфейсів, які були надані бібліотекою, стало можливим створення компонентів вищого порядку, що дає змогу виносити логіку по взаємодії із користувачем та DOM деревом, тобто сторонні ефекти, в окремі компоненти, які виступають обгортками для цільових компонентів, без необхідності зміни вихідного коду, міняється лише спосіб експорту даного компонента.

React ефективно оновлює процес DOM (об'єктна модель документа). Інструмент дає можливість створювати віртуальні DOM і розміщувати їх в пам'яті. В результаті кожен раз, коли відбувається зміна в реальному DOM, віртуальний змінюється негайно. Ця система буде перешкоджати тому, щоб фактичний DOM не форсувати постійні оновлення. Як результат, не страждатиме швидкість вашої програми. Початкова складність обновлення DOM дерева було надто високою $O(n^3)$, що на великих проєктах із сотнею компонентів привело б до постійних підвисань інтерфейсу. Тому задля того, щоб кардинально покращити складність даного процесу до показників O(n) було прийнято два припущення, першим з яких було те, що два елементи різного типу будуть створювати різні дерева. Друге припущення вимагає безпосередньої підтримки зі сторони самого розробника, в місцях де дочірній елемент непотрібно обновляти, достатньо добавити спеціальний атрибут key. Окрім цього, бібліотека ще надає доступ до методів життєвого циклу, де програмно, під час виконання коду, можна виставити умови при яких необхідно міняти стан компонента.

Іншим важливим етапом в розвитку бібліотеки було представлення React Hooks, де на противагу класовим компонентам було представлено інтерфейс для створення компонентів на базі функцій. Такий підхід ще більше пришвидшив роботу застосунків, що використовували цю технологію, оскільки класи в JavaScript з’явились досить пізно, і їхня реалізація відрізняється від імплементації в інших мовах програмування, що

неочікувано привело до проблеми при побудові вкладених об’єктів, що були створені на основі цих класів. Тому використання функцій стало вирішенням цієї проблеми, оскільки раніше функціональні компоненти могли використовуватись лише для візуального представлення даних, для певної взаємодії з користувачем, були необхідні класові компоненти. Але використання функціональних компонентів вимагає слідування певних правил, а саме використання хуків для мемоізації даних, оскільки без цього всі переваги в швидкості будуть недієвими.

До того ж, React підтримує багато платформ (кросплатформовий за своєю суттю) [15].

2.1.2 Використання Redux

Redux – це бібілотека для управління станом вебзастосунку, вона була побудована на основі шаблону Flux, який було створено компанією Facebook для створення клієнтської частини вебзастосунків. Саме цей шаблон визначає правила за якими відбувається процес зміни стану вебзастосунку, та місце, де зберігається сам стану. Детальніший вигляд структури та потоку даних на рис. 2.1:

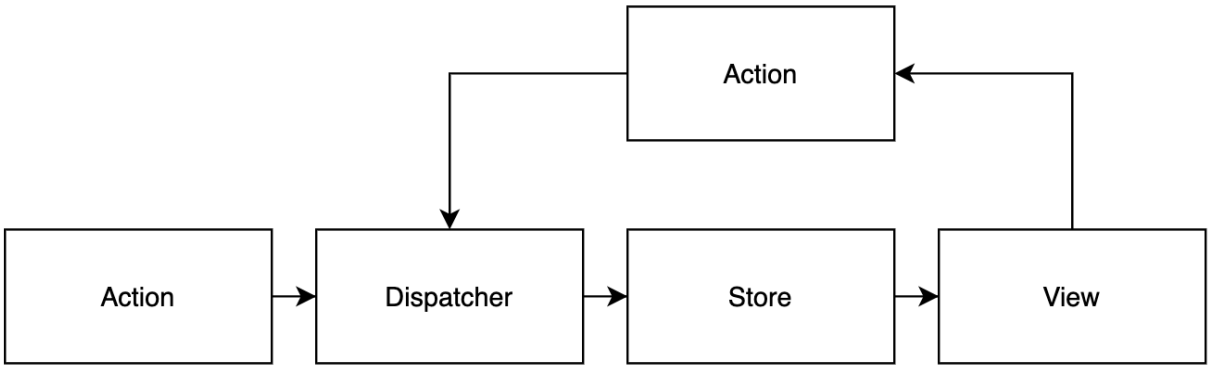


Рис. 2.1 Структура та потік даних Flux

На основі цієї структури будуються три основні принципи Redux. Єдине джерело правди – глобальний стан застосунку знаходиться в

деревовидному об’єкті в єдиному сховищі. Стан доступний тільки на читання – задля зміни стану вебзастосунку єдиним способом є відправлення action, об’єкту, який описує, як змінився стан. Зміни повинні бути зроблені лише чистими функціями – щоб вказати, як дерево стану трансформується за рахунок actions, потрібно писати чисті reducers.

Даний підхід керування станом став стандартом для вебзастосунків, і на його основі було створено бібліотеки для всіх основних front-end фреймворків, що дає можливість використовувати однаковий інтерфейс не залежно від обраної платформи.

2.1.3 Використання Redux-saga

Redux-saga − це бібліотека, що була створена задля вирішення проблеми роботи із сторонніми ефектами, такими як звернення до серверу та взаємодії із сховищами браузера. Основне її використання в React/Redux додатках, тому, з огляду на те, що в нашому випадку був обраний саме React – було вірним рішенням. Redux-saga спроможна викоренити асинхронні операції, наприклад: завантаження даних, або доступ до браузерного кешу.

Можна зобразити сагу, як окремий шар вебзастосунку, який є місцем для написання бізнес логіки та інших сторонніх ефектів. Redux-saga являє собою проміжним програмним забезпеченням для redux, що означає, що керування цим шаром відбувається тими ж механізмами, що і для redux, тобто за допомогою стандартних action об’єктів, окрім цього дані події можуть перехоплюватися, що дозволяє детальніше прописувати логіку вебзастосунку в одному місці. Також саги мають прямий доступ до стану всього вебзастосунку.

Redux-saga використовує функції генератори, представлені в останній версії EcmaScript, для того, щоб полегшити роботу із асинхронними діями. Такий підхід значно спрощує написання та тестування бізнес логіки. Тим самим, ці асинхронні потоки виглядають, як стандартний синхронний JavaScript код [16].

2.1.4 Використання Typescript

TypeScript − це мова з відкритим вихідним кодом, який заснований на JavaScript, одному з найбільш часто використовуваних в світі інструментів, шляхом додавання визначень статичних типів.

Типи надають спосіб опису форми об'єкта, забезпечуючи кращу документацію і дозволяючи TypeScript перевіряти правильність роботи коду.

До основних плюсів можливо віднести такі: 1) TypeScript забезпечує просту навігацію по коду, запобігання помилок і підтримку коду. 2) Він підтримує статичну типізацію. 3) Він також підтримує інтерфейси, підлеглі інтерфейси, класи, підкласи і ECMAScript 6 або ES6 (до того ж обрана Redux-saga також підтримує цей підклас). 4) Він поставляється з можливостями об'єктно-орієнтованого програмування.

2.2. Back-end

В основі написання бек-енду лежить мікросервіси та Docker-контейнери. На них вже буде будуватись інше – nestjs, mongodb.

З’єднання користувачів буде здійснюватись за допомогою − websocket та rtc peer connection.

2.2.1 Використання Мікросервісів

Як зазначив Мартін Фаулер, британський лектор з розробки програмного забезпечення, інженер-програміст, у своєму блозі, під мікросервісами мають на увазі певний архітектурний стиль. Тобто, це підхід, при якому певна множина невеликих сервісів складають єдиний додаток, при цьому кожен з них запускається окремо на іншому процесі, або ж віддалено. Взаємодія між цими сервісами будується за допомогою різних механізмів зв’язку, таких як HTTP, TCP, Kafka або Redis. Дані сервіси будуються згідно задач поставлених бізнесом, щоб задовольнити їхні потреби. Вони розгортаються незалежно одне від одного, використовуючи середовища, що

повністю автоматизовані. Централізоване управління даними сервісами або цілком відсутнє, або ж має обмежений функціонал, задля незалежності та повторного використання самих сервісів. Ці сервіси можуть бути написані різними мовами програмування і зберігати дані в різноманітних базах даних або їх аналогах[19].

2.2.2 Використання Docker-контейнерів

Контейнери Docker − це запущені екземпляри образів Docker. Запуск образу створює Docker-контейнер. Образи є якимись шаблонами, які можна використовувати для створення контейнерів. Вони містять інформацію про те, що потрібно для створення контейнера. Образи можуть зберігатися локально або віддалено [21].

Для простого розуміння інструментарію Doker-контейнерів, як вказують деякі програмісти, можливо уявити загальну «проблему». Для того аби вирішити цю «проблему» слід розбити її на біль менші, які можливо вирішити набагато швидше, що за наслідком безпосередньо вплине на сукупний результат.

Додатково Doker-контейнери порівнюють з звичайними контейнерами, які використовують в мореплавстві. Практична діяльність – перевозка великогабаритних предметів, товару, тощо. Розподіливши такий вантаж на певні контейнери, їх пересилка на відстані буде легшою.

Тобто, Doker-контейнери розподіляють наявний інструментарій в окремі контейнери для швидкого вивантаження та обробку програмного коду, збільшуючи при цьому пропускну здатність в цілому.

2.2.3 Використання NestJS

NestJS − це серверна прикладна середа. NestJs побудований на основі Typescript і Express.js. Він також об'єднує деякі елементи об'єктно-орієнтованого програмування і функціонального програмування.

Тобто, він буде яскравим доповненням обраному та описаному вище – Typescript'у.

NestJS, фреймворк, який повністю написаний на TypeScript (він також підтримує JS), він легко тестуємо і містить все необхідне [18].

Nest використовує надійний фреймворки HTTP-серверів.

Додатковий плюс Nest − спрощує рендеринг додатків React (якій був обраний у якості інструментарію Front-end`у та описаний вище за текстом) на стороні сервера. В свою чергу, це допомагає використовувати одну і ту ж логіку рендеринга як на стороні клієнта, так і на стороні сервера.

Nest.js заснований на концепціях захисту, каналів і перехоплювачів і поставляється з вбудованою підтримкою WebSockets і gRPC. Nest.js використовує Express і повністю сумісний з великою кількістю популярних бібліотек. Він в основному використовує класи, декоратори, відображення метаданих для більшості своїх абстракцій [20].

Зрештою, приписка «JS» не є звичайною. Nest ніби компанує та дає можливість вписувати мову JavaScript («Фронт-енду») безпосередньо в «бек-енд».

2.2.4 Використання MongoDB

MongoDB − це розподілена база даних загального призначення на основі документів, створена для сучасних розробників додатків і для хмарної епохи [22].

Додатково слід зазначити, що MongoDB − це документоорієнтована база даних типу NoSQL. На відміну від реляційних баз даних бази NoSQL використовують не таблиці з рядками і стовпцями, а колекції і JSON-подібні документи. Документи складаються з так званих пар «ключ-значення». Пара «ключ-значення» є ім'я поля (ключ документа) і його значення. Колекції ж складаються з груп документів.

Крім того, MongoDB можливо підключити до Node JS.

До того ж, під час розгляду вже створених програм (додатків) в Розділі 1, Discord на початку своєї розробки використовував базу даних саме – MongoDB.

До переваг MongoDB можливо віднести такі:

- Гнучкість − MongoDB зберігає дані в документах формату JSON, а не в таблицях. Це дає можливість зберігати інформацію зі складною структурою. При цьому зміст і розмір документів може бути різним, і не потрібно створювати певну схему бази даних.
- Кросплатформеність − MongoDB можна використовувати на операційних системах Windows, Linux (Ubuntu, Debian, CentOS), MacOS.
- Динамічні запити до документів.
- Реплікація − MongoDB може працювати на декількох серверах.
- Проста масштабованість [23].

З огляду на те, що вебзастосунок потребує безпосереднього зв’язку користувачів, то обмін даними буде підтримуватись через websocket і rtc peer connection.

2.2.5 Використання WebSocket

WebSocket – це протокол зв’язку. Вебсторінок використовують протокол WebSockets для двостороннього зв'язку з віддаленим хостом. Він являє інтерфейс WebSocket і визначає повнодуплексний канал зв'язку, який працює через один сокет, через Інтернет. WebSockets забезпечує величезну скорочення непотрібного мережевого трафіку і затримки в порівнянні з іншими рішеннями для опитування та тривалого опитування, які використовувалися для імітації полнодуплексного з'єднання шляхом підтримки двох з'єднань [24].

Протокол WebSocket був розроблений для хорошої роботи з наявною вебінфраструктурою. Як частина цього принципу проектування, специфікація протоколу визначає, що з'єднання WebSocket починає своє життя як з'єднання HTTP, гарантуючи повну зворотну сумісність з

WebSocket. Перемикання протоколу з HTTP на WebSocket згадується як «рукостискання» WebSocket.

Браузер відправляє запит на сервер, вказуючи, що він хоче переключити протоколи з HTTP на WebSocket. Клієнт висловлює своє бажання через відповідний заголовок [24].

2.2.6 Використання RTC Peer Connection

Інтерфейс RTCPeerConnection представляє з'єднання WebRTC (згаданий під час огляду програми – Discord) між локальним бенкетом (учасником з'єднання) на локальному комп'ютері і віддаленим бенкетом на віддаленому комп'ютері. Він надає методи для з'єднання з віддаленим учасником з'єднання, обслуговування, моніторингу та закриття з'єднання.

Припис «peer connection», схожий на «Peer-to-Peer» (P2P, згаданий під час огляду програми Skype).

Та має приблизно таку складову модель:

1. Перший користувач направляє Другому запрошення на зв’язок через відповідний сервер.

2. Другий користувач отримує запрошення та відправляє відповідну відповідь Першому.

3. Кінцевий результат – налаштування з’єднання між користувачами [25].

2.2.7 Використання Nginx

Nginx – це вебсервер, який також зарекомендував себе як зворотний проксі, HTTP кеш та load balancer.

Nginx був створений задля малого використання пам’яті, але високої паралельності. Nginx використовує асинхронний підхід, що базується на подіях, де всі запити обробляються єдиним потоком.

Зворотний проксі-сервер - це тип проксі-сервера, який зазвичай знаходиться за брандмауером у приватній мережі та спрямовує запити

клієнтів на відповідний серверний сервер. Зворотний проксі забезпечує додатковий рівень абстракції та контролю для забезпечення плавного потоку мережевого трафіку між клієнтами та серверами.

Однією із переваг зворотного проксі-сервера є безпека та анонімність - перехоплюючи запити, спрямовані на ваші серверні сервери, зворотний проксі-сервер захищає їх ідентифікаційні дані та діє як додатковий захист від атак безпеки. Це також забезпечує доступ до декількох серверів з одного локатора записів або URL-адреси незалежно від структури вашої локальної мережі.

2.3 Реєстраційний інструментарій запропонованого вебзастосунку

JWT, або JSON WEB TOKEN – токен, який слугує для ініціалізації користувачів, їх реєстрації з подальшим збереженням введеної інформації та звернення до неї з проміжком часу.

Принцип роботи JWT побудований на відповідних токенах та являє собою зашифрований формат упаковки даних, який використовується для безпечної передачі інформації між двома сторонами..

Дуже простий у використання для створення системи реєстрації користувачів.

Після того, як користувач пройшов авторизацію в системі, вказавши свій логін і пароль, система видає йому 2 токена: access token і refresh token.

Перший містить відповідну інформацію за певним посиланням і такий token існує невеликий проміжок часу. Слугує здебільшого для створення відповідного літеро-циферного коду.

Другий token також містить схожу конструкцію, однак він вже виконує роль підмінника, щоб під час зупинення роботи першого, користувач продовжував звертатись до збереженої інформації, міг віднайти інформацію, чи провести відповідний enter-вхід до сервісу [26].

2.4. Хмарне сховище мультимедії для вебзастосунку

Cloudinary – універсальна система для завантаження, зберігання, маніпуляції і поширення зображень і відео.

Cloudinary покриває все: від завантаження зображень, зберігання, маніпуляцій, оптимізації та до доставки.

З ним можна легко завантажувати зображення в хмару, автоматично виконувати інтелектуальні маніпуляції із зображеннями без установки будь-якого складного програмного забезпечення.

До речі зображення потім легко доставляються через швидкий CDN, оптимізуються і використовують кращі в галузі методи.

Cloudinary пропонує комплексні API-інтерфейси і можливості адміністрування і легко інтегрується з новими і наявними веб- і мобільними додатками.

Сервіс допомагає зберігати зображення в окремому хмарному сховищі, що у свою чергу допомагає зосередити свою увагу на інших процесах не відволікаючись.

Співпраця проходить програмно: Від сервісу Cloudinary до запиту клієнта-користувача і надалі до сайту з яким пов’язане сховище [27].

ВИСНОВКИ ДО РОЗДІЛУ 2

В цьому розділі до уваги було наведено використаний інструментарій під час створення власного вебзастосунку для проведення онлайн-занять.

Стисло наведено функціональні можливості кожного обраного інструментарію.

Зрештою, можливо простежити тенденцію, що один мовний інструмент доповнює інший. Або один інструмент безпосередньо базується на другому, що беззаперечно є позитивною характеристикою програмного коду.

Насамкінець, обрані мови, фреймворки, та супутні програми – відповідають сучасним вимогам: актуальність, підтримка, оновлення, та цей список є завеликим.

РОЗДІЛ 3
ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ

3.1 Внутрішня будова

Внутрішня будова запропонованого вебзастосунку являє собою конструкцію з багатьма змінними (рис. 3.1):

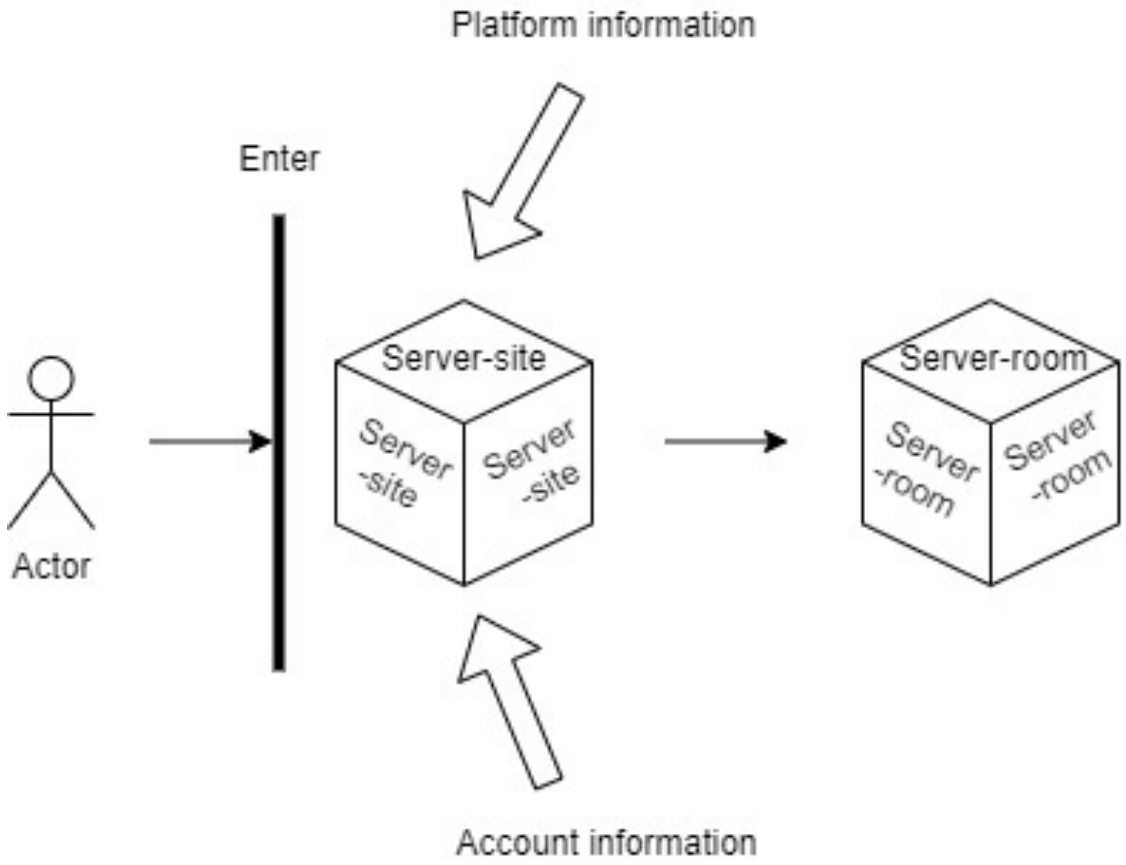


Рис. 3.1 Загальний вид внутрішньої будови запропонованого додатку

На наведеному вище рисунку можемо спостерігати, що користувач, перед використанням вебзастосунку, повинен пройти авторизаційний процес.

Після його успішного проходження він потрапляє на головний сервер вебзастосунку, там вже може користуватись певним функціоналом (налаштування, пошук відповідної кімнати, перегляд відправлених фотоматеріалів, тощо).

Під час входу в обрану кімнату, клієнт потрапляє на окремий сервер. Такий же сервер створюється і при створенні самої кімнати самим клієнтом.

В Додатку 1 висвітлена структура зв’язку користувачів у окремій кімнаті.

Встановлений взаємозв’язок між клієнтами (жовтий колір) вказує на стабільне з’єднання, що дещо нагадує P2P, однак, виходячи з того, що ми використовуємо RTCPeerConnection, то у нашій моделі присутній і відповідний сервер.

Цей сервер бере участь під час наявних у користувачів проблем зв’язку (червоний колір).

У свою чергу, користувач який має поганий зв’язок, звертається до серверу, який направляє зв’язок цьому користувачу за новими каналами (або навіть іншими серверами), таким чином налагоджує зв’язок між клієнтами у кімнаті (зелений колір).

Окремо розглянемо алгоритми, що дозволяють створити з'єднання по протоколу WebRTC, контролювати обробку «сигнальних» даних, здійснювати буферизацію «сигнальних» даних і групувати сокети клієнтів в окремі «кімнати».

Протокол WebRTC має особливості при з'єднанні користувачів: для створення з'єднання між клієнтами потрібно виконати операцію «handshake», яка складається з обміну різним типом «сигнальних» даних між браузерами, але одночасно клієнт може встановлювати тільки одне з'єднання по протоколу WebRTC.

Щоб вирішити проблему, що склалася, було запропоновано кілька додаткових підходів до побудови архітектури обміну даними в додатку: буферизація «сигнальних» даних протоколу WebRTC на клієнті і сервері, об'єднання сокетів, що з'єднують клієнтів в «кімнату» на сервері. «Кімната» - це масив, що знаходиться на сервері, що складається з декількох сокетів, що дає змогу обмінюватися даними тільки з сокетами, що знаходяться в даному масиві. Таким чином, «кімнати» ізолюють групи сокетів один від одного,

сприяючи поширенню даних тільки всередині певних груп. Такі підходи допомагають виключити втрату даних, дозволяють створити всі необхідні з'єднання між клієнтами і управляти процесами з'єднання клієнтів на різних етапах роботи програми.

Розглянемо алгоритм, що описує етап з'єднання клієнтів до формування «сигнальних» даних. Спочатку «викликаючий» клієнт подає запит на сервер для з'єднання з «відповідаючим» клієнтом по протоколу WebSocket. Далі сервер здійснює пошук «відповідаючого» клієнта серед підключених. Якщо клієнт відсутній, то сервер завершить дзвінок «викликаючого» клієнта, якщо «відповідаючий» клієнт знайдений, то йому відправляється запит на з'єднання по протоколу WebSocket. «Відповідаючий» клієнт формує і відправляє відповідь на запит. Якщо відповідь негативна, то сервер завершує дзвінок «викликає» клієнта, в разі позитивної відповіді сервер отримає id сокетів «викликаючого» і «відповідаючого» клієнтів, по id сокетів знайде кімнату, в яких сокети знаходяться в даний момент.

Після завершення даного алгоритму відбувається створення і обробка буферів сокетів на сервері за допомогою алгоритму, зображеного на рис. 3.4:

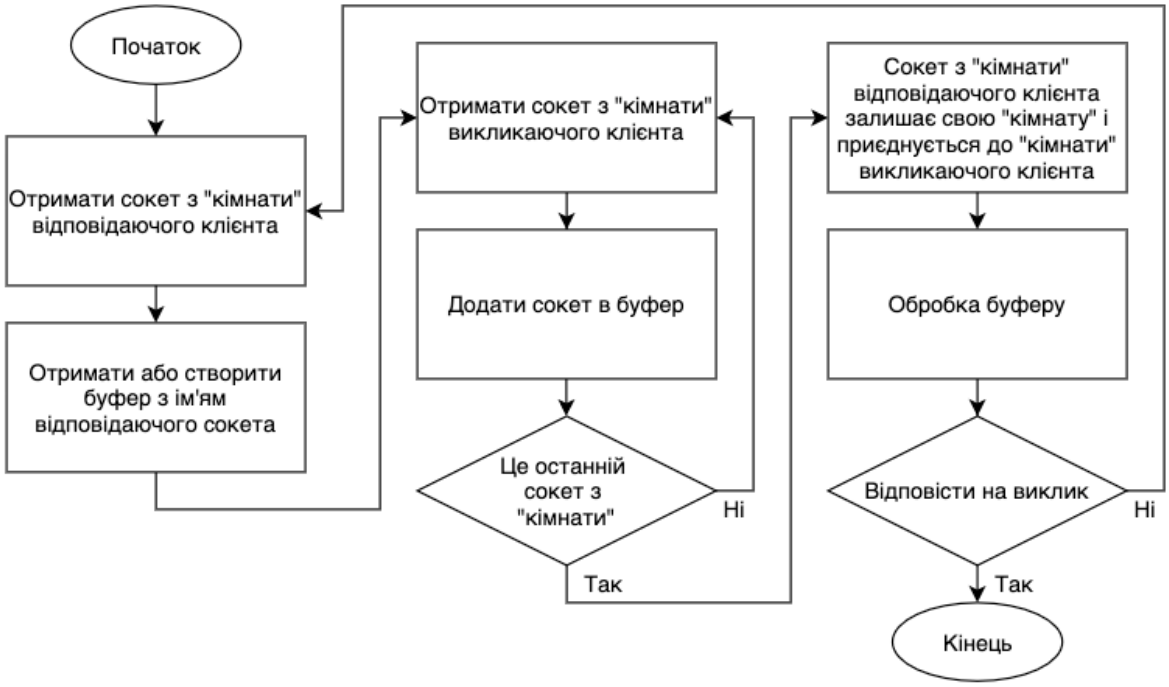


Рис. 3.4. Алгоритм розподілу гнізд і обробки їх буферів на сервері

Далі відбувається виклик функції обробки буфера сокета, яка виконує запит на формування сигнальних даних для всіх клієнтів, які перебувають в черзі даного буфера. В кінці алгоритму виконується перевірка на порожнечу кімнати «відповідаючого» клієнта, якщо «кімната» не порожня, то алгоритм повторить всі дії з самого початку, в іншому випадку алгоритм вважається завершеним. Як можна помітити, даний алгоритм додає в «кімнату» «викликаючого» клієнта клієнтів з кімнати «відповідаючого» клієнта, і при кожному зверненні до кімнаті «викликаючого» клієнта в даному алгоритмі кімната повинна збільшуватися. Але це не так через те, що перед початком алгоритму відбувається дублювання всіх користувачів з кімнати «викликаючого» клієнта в окремий масив, таким чином, алгоритм працює правильно, і кожен раз він використовує не основну кімнату, а заздалегідь підготовлений масив.

3.2 Структура серверної частини вебзастосунку

Для зручності підтримки, швидкості розгортання вебзастосунку на будь-якому середовищі, та кращої можливості паралельної обробки запитів було обрано мікросервісну архітектуру. Даний вебзастосунок було розділено на декілька мікросервісів, вказаних нище.

User-service – сервіс, який відповідає за автентифікацію і авторизацію користувачів, а також інші маніпуляції по обновленню та агрегації їх даних. Цей сервіс виконує найважливішу роль для всієї системи, оскільки більшість ендпоінтів системи доступні лише для авторизованих користувачів, то кожен запит супроводжується внутрішнім викликом на даний сервіс. Тому беручи до уваги критичність постійної роботи цього сервісу, при подальшому розгортанні системи на клауд платформі, його потрібно розмістити на більшій частині кластерів у порівнянні із іншими сервісами.

Вхідною точкою для роботи із вебзастосунком є реєстрація, тому розглянемо механізм її роботи. Для створення нового запису користувача в базі даних, сам користувач повинен надати інформацію про своє ім’я,

електронну пошту та пароль. Ці дані пройдуть валідацію і лише тоді сервіс відкриє канал для роботи із базою даних. В разі не унікальності електронної пошти, користувача буде повідомлено про це відповідним статус кодом. В іншому випадку наданий пароль буде захешовано, новий запис в таблиці users буде створено, а користувачу повернеться JWT токен, використання якого надасть можливість надалі взаємодіяти із системою. На наступних рисунках показано приклад запиту (рис. 3.5) та відповіді (рис. 3.6) при реєстрації:

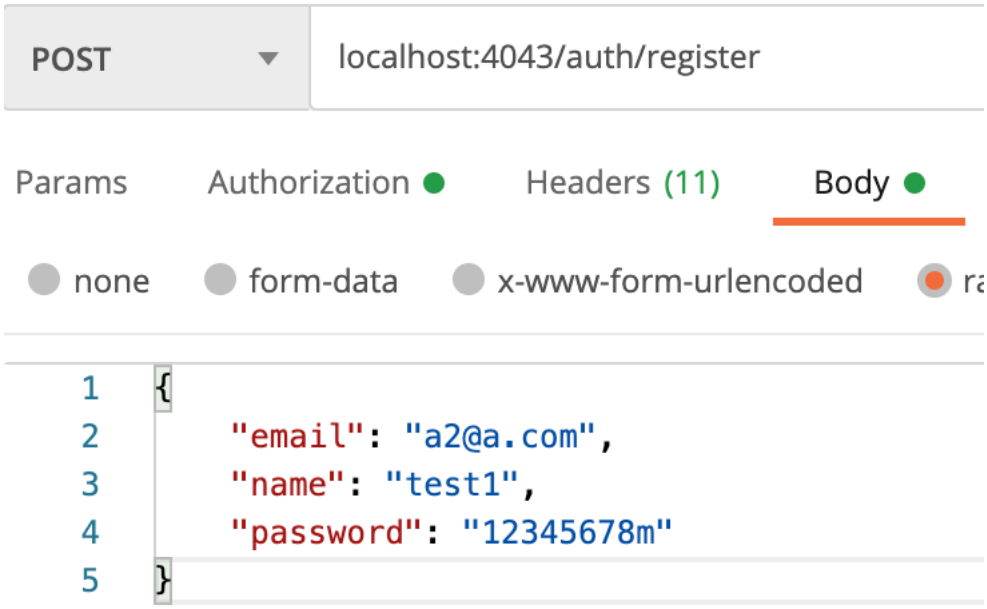


Рис. 3.5 Приклад POST запиту для реєстрації

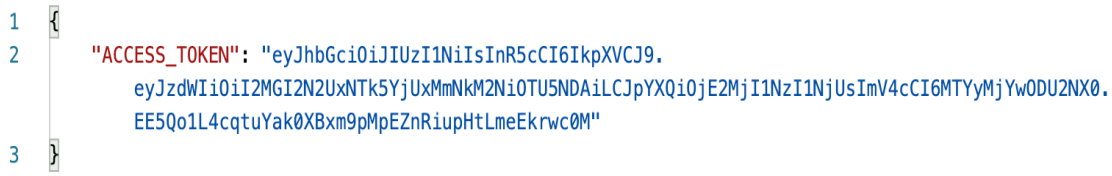


Рис. 3.6 Приклад отриманого JWT токена при успішній реєстрації

Іншим важливим механізмом даного сервісу є авторизація користувачів при логіні. Користувач повинен надати електронну пошту та пароль, що були

вказані при успішній реєстрації, надані дані будуть перевірені на відповідність типів.

Наступним кроком є валідація користувача, вона необхідна не лише при авторизації, але й для інших захищених контролерів. Першим кроком є пошук у базі даних запису із наданою електронною поштою, якщо такого запису не було знайдено, буде віддано відповідь із відповідним статус кодом. В іншому випадку за допомогою допоміжної бібліотеки буде виконано перевірку на відповідність наданого паролю із його хешованою версією на основі секретного ключа. При успішному сценарію користувач буде допущений до запрошуваного контролера, або ж йому буде надано JWT токен. На наступних рисунках показано приклад запиту (рис. 3.7) та відповіді, JWT токен при якій згенерувався новий (рис. 3.8) при логіні:

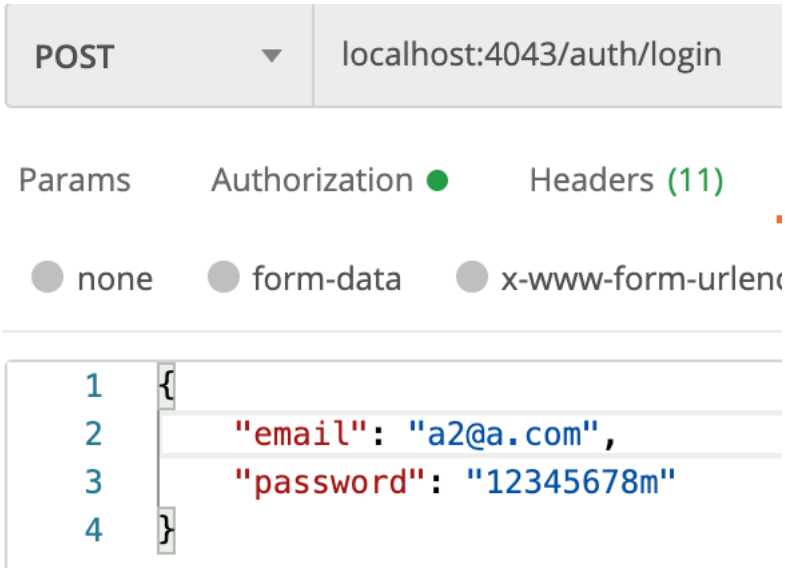


Рис. 3.7 Приклад POST запиту при логіні


```
{
    "ACCESS_TOKEN": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJzdWIiOiI2MGI2N2UxNTk5YjUxMmNkM2NiOTU5NDAiLCJpYXQiOjE2MjI1NzM3MzgsImV4cCI6MTYyMjYwOTczOH0.
        al08A_SOLblO0vzkHKXqPhn9gIhaGfKzrvjUt0J3Shw"
}
```


Рис. 3.8 Приклад отриманого JWT токена при успішному логіні

Room-service – сервіс, який відповідає за створення та обновлення кімнат, і окрім цього за допомогою нього інціалізується та підтримується відео та аудіо з'єднання між користувачами. Даний сервіс унікальний тим, що поєднує комунікацію через HTTP та через WebSockets, при цьому вхідна точка, тобто порт, залишається спільним для обох способів, що спрощує конфігурацію при взаємодії із клієнтською частиною або ж іншими мікросервісами.

Для взаємодії між мікросервісами обраний фреймворк надає кілька способів передачі даних, серед яких є TCP, Kafka та Redis. Така гнучкість дозволить для певних реплік системи вибрати інший спосіб передачі даних, для цього необхідно лише створити відповідний контейнер та передати шляхи та інші конфігураційні дані до сервісів даної системи-репліки.

Config-service – сервіс, який відповідає за всі конфігураційні файли, що необхідні на клієнтській стороні. Серед них можна виділити файли для локалізації контенту, що дозволить у майбутньому, без надлишкових зусиль, добавити підтримку кількох мов, оскільки всі записи знаходяться всередині звичайного JavaScript об'єкта (рис. 3.9), тому добавлення проксі до цього об'єкта ніяк не вплине на структуру решти коду.

```
'copyright-line1': 'Copyright © 2020 by Vasyl Obukh',
'copyright-line2': 'All rights reserved',
'nav-menu-rooms': 'Rooms',
'nav-menu-create-room': 'Create room',
'user-welcome-label': 'Welcome,',
'user-menu-rooms-label': 'My rooms',
```

Рис. 3.9 Структура текстових позначок

Proxy – окремий контейнер на базі nginx, що є вхідною точкою при взаємодії клієнта із сервером, тобто виступає в ролі API Gateway. Такий підхід дозволить збільшувати кількість або ж розділяти логіку уже наявних мікоросервісів в майбутньому, при цьому для клієнтської частини не потрібно буде вказувати нових способів з'єднання. Також цей підхід

забезпечить сильнішу безпеку, оскільки архітектурні рішення заховані під однією сутністю. На рис. 3.10 можна побачити дані конфігурації API Gateway із змінними середовища, що дає можливість налаштовувати які репліки сервісів використовувати на поточному середовищі:

```
server {
    listen $PORT default_server;

    location /config/ {
        proxy_pass $CONFIG_SERVICE_URL/;
    }

    location /room/ {
        proxy_pass $ROOM_SERVICE_URL/;
    }

    location /user/ {
        proxy_pass $USER_SERVICE_URL/;
    }
}
```

Рис. 3.10 Конфігурація API Gateway через nginx

3.3 Розробка бази даних

Однією із важливих причин для вибору бази даних були наявність чи відсутність великої кількості зв'язків між сутностями, необхідними для вебзастосунку. Оскільки такі дані як користувацькі файли та зображення зберігаються на серверах сторонніх сервісів, єдиним зв'язком являється зв'язок між користувачем та кімнати, що були створені ним. Із цих причин було обрано NoSQL базу даних MongoDB. Під час проектування задля досягнення поставлених задач, було створено декілька схем, основні з яких розберемо детальніше.

User – схема, що містить інформацію щодо користувача вебзастосунку, вона складається із наступних полів:

- _id – унікальний ідентифікатор;
- name – ім'я користувача, валідація якого складається із того, що поле приймає лише рядки із символами латинського та кириличного алфавітів;

- email – електронна адреса користувача, валідація якого складається із того, що поле приймає лише рядкові значення, що задовольняють стандартний email-шаблон;
- password – пароль користувача у вигляді хешу;

MongoDB надає пітримку різноманітних хуків, тобто можливість добавляти обробники на події, які відбуваються в процесі роботи бази даних, тому у даному вебзастосунку при кожному збереженні змін для екземпляра User схеми, перевіряється чи було внесено зміни в поле password, якщо так, то відбувається повторне хешування паролю.

На рис. 3.11 наведено приклад запису користувачів в базі даних:

Рис. 3.11 Записи користувачів в MongoDB

Іншою важливою схемою є room, що описує інформацію про кімнати, складається із таких полів:

- _id – унікальний ідентифікатор;
- title – заголовок кімнати, валідація якого складається лише із того, що поле повинно бути непорожнім рядком;

- image – шлях до зображення кімнати;
- private – булевий атрибут, який інформує чи потрібен пароль для того, щоб увійти в кімнату;
- ownerId – ідентифікатор користувача, що створив поточну кімнату, завдяки цьому полю MongoDB може робити агрегацію даних користувача при отриманні інформації про кімнату;

На рис. 3.12 наведено приклад запису кімнат в базі даних:


```
_id: ObjectId("602d7b145c191244456a8eab")
title: "title"
image: "https://res.cloudinary.com/eterry/image/upload/v1613593363/kshsbypemkv..."
private: true
ownerId: "601afff09c4d350859052c48"
__v: 0

_id: ObjectId("602d7b165c191244456a8eac")
title: "title"
image: "https://res.cloudinary.com/eterry/image/upload/v1613593365/z2qeef7b7ma..."
private: true
ownerId: "601afff09c4d350859052c48"
__v: 0

_id: ObjectId("602d7b185c191244456a8ead")
title: "title"
image: "https://res.cloudinary.com/eterry/image/upload/v1613593367/uexdnydhh3a..."
private: true
ownerId: "601afff09c4d350859052c48"
__v: 0
```


Рис. 3.11 Записи кімнат в MongoDB

ВИСНОВКИ ДО РОЗДІЛУ 3:

В цьому розділі були наведені та розглянуті функціональні особливості вебзастосунку. Також виділена зовнішня будова та окремо звернено увагу саме на відповідну модель з’єднання користувачів.

Наведені відповідні алгоритми роботи вебзастосунку.

Завдяки обраному архітектурному рішенню, описаному в даному розділі, вдалось досягти досить великої швидкості розгортання вебзастосунку на будь-якому середовищі та покращення обробки паралельних запитів.

Не менш важливо було розглянуто основні схеми бази даних, що дало краще розуміння роботи всієї системи.

РОЗДІЛ 4
ІНТЕРФЕЙС ЗАПРОПОНОВАНОГО ВЕБЗАСТОСУНКУ ДЛЯ ПРОВЕДЕННЯ ОНЛАЙН-ЗАНЯТЬ

Перш за все, запропонований вебзастосунок для проведення онлайн-занять покликаний вирішити ряд проблем з безпосереднім зв'язком у системі «студент-викладач». А також, поліпшити проведення викладачами семінарських, практичних та інших занять.

В цьому розділі до уваги буде презентований інтерфейс, зосередимо увагу на його UX дизайні, тобто на простоті взаємодії з ним, та проаналізуємо його фактичний функціонал.

Після потрапляння на відповідний вебсайт для використання вебзастосунку слід пройти звичайну авторизацію або реєстрацію у разі відсутності облікового запису в системі (рис 3.1 та рис 3.2):

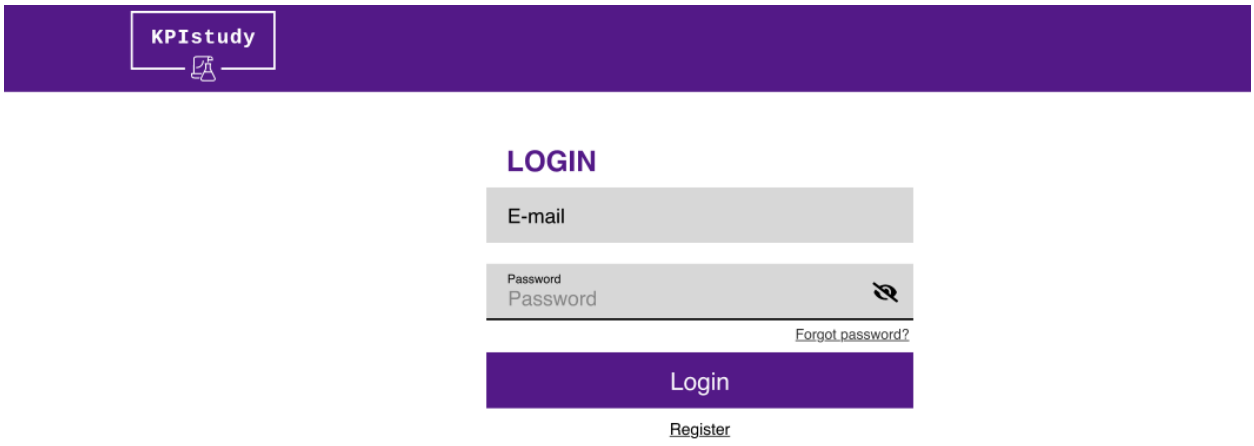


Рис 3.1 Діалогове вікно проходження авторизації

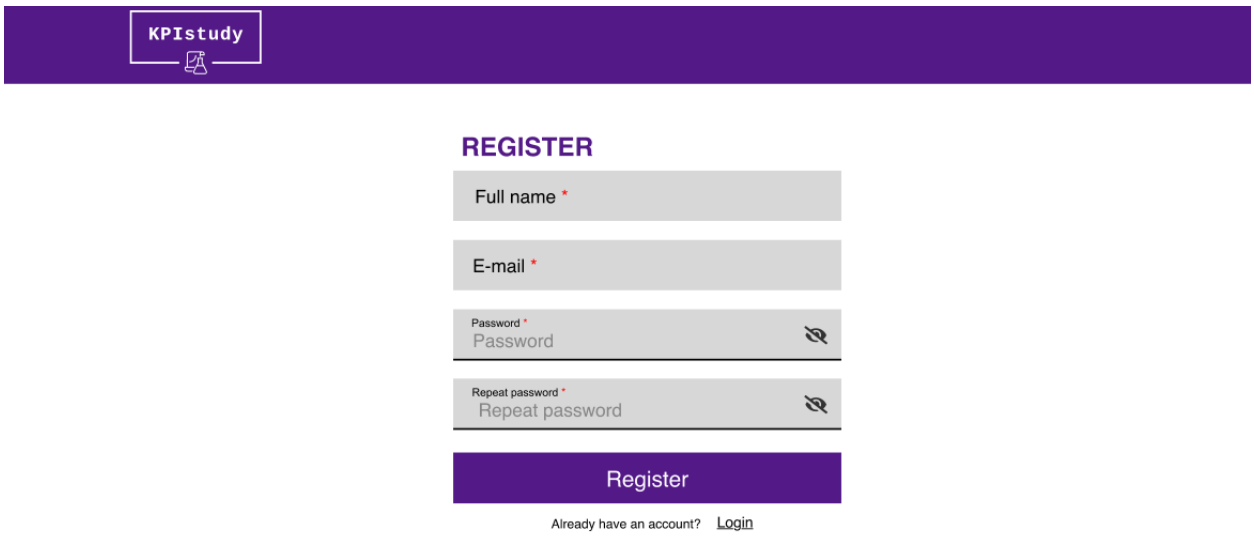


Рис 3.2 Діалогове вікно проходження реєстрації

Після того як користувач увійшов до системи він потрапляє на відповідну сторінку (рис. 3.3):

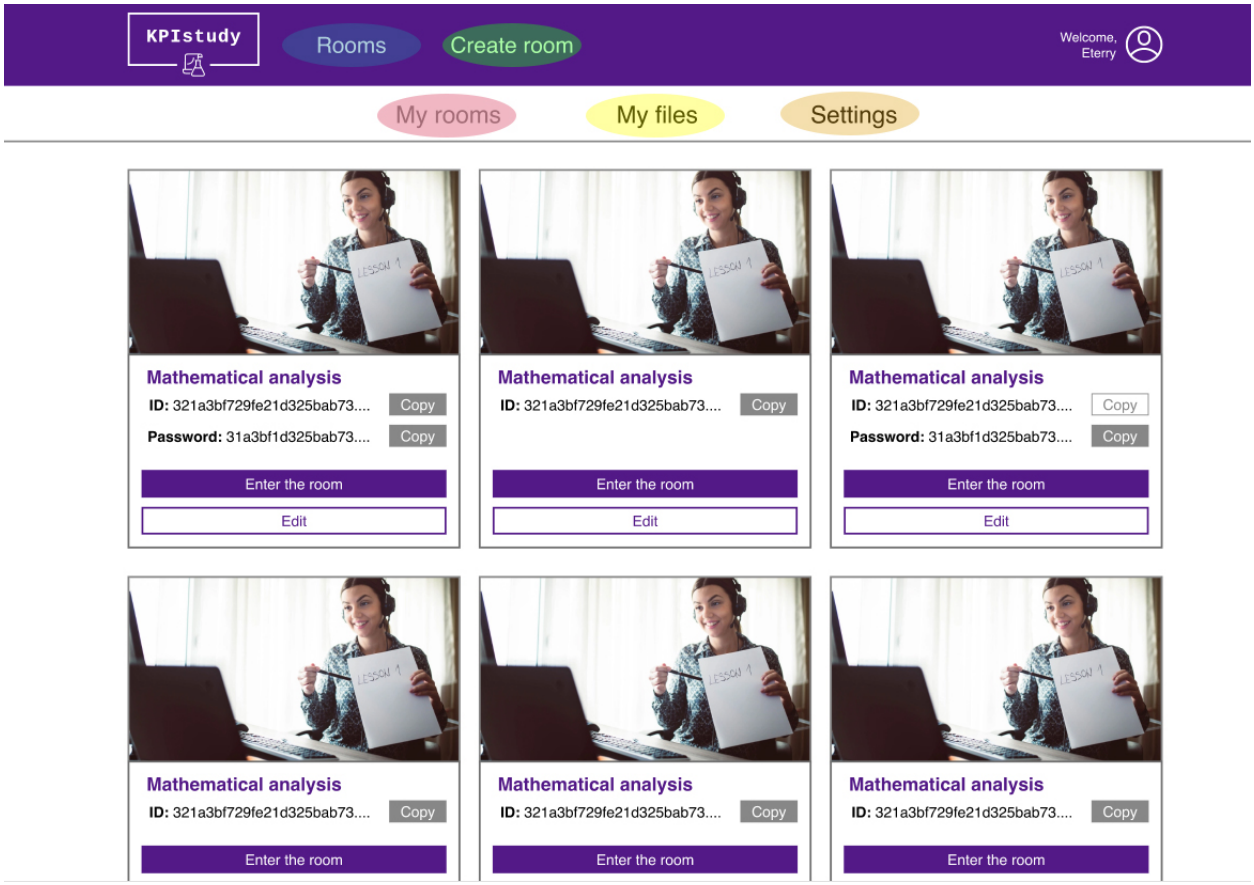


Рис. 3.3 My account сторінка вебзастосунку

На наведеному вище рис. 3.3 додатково виділені зони різними кольорами («My rooms» – червоним, «My files» – жовтим, «Settings» – помаранчевим, «Rooms» – синім, «Create room» – зеленим).

Слід коротко навести функціональні можливості кожної зазначеної зони.

«My rooms» – мої кімнати. Розділ, який відповідає за вже відвідані та/або додані канали (кімнати) зв'язку для швидкого віднаходження потрібного користувачу. Також тут можна відредагувати дані кімнати, що були створені поточним користувачем (рис. 3.4):

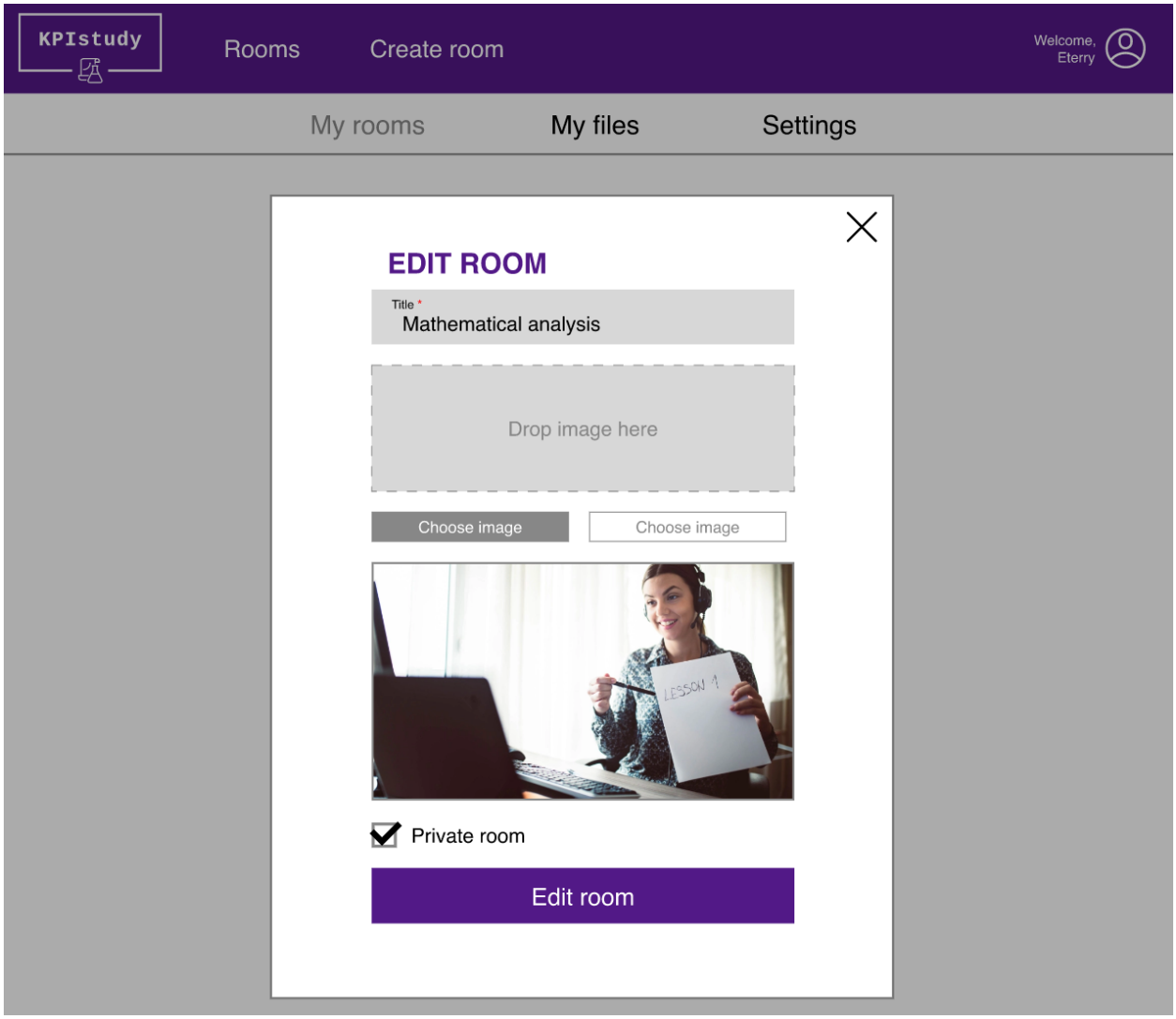


Рис. 3.4 Вікно для зміни інформації про кімнату

«My files» – мої файли. Дає можливість користувачу звертатись до надісланих, відправлених, отриманих та інших файлах у відповідних кімнатах.

«Settings» − налаштування. За допомогою цієї функції користувач може змінювати відомості внесені під час реєстрації облікового запису (рис. 3.5 та рис. 3.6):

Рис. 3.5 Вікно для зміни ім’я в обліковому записі користувача

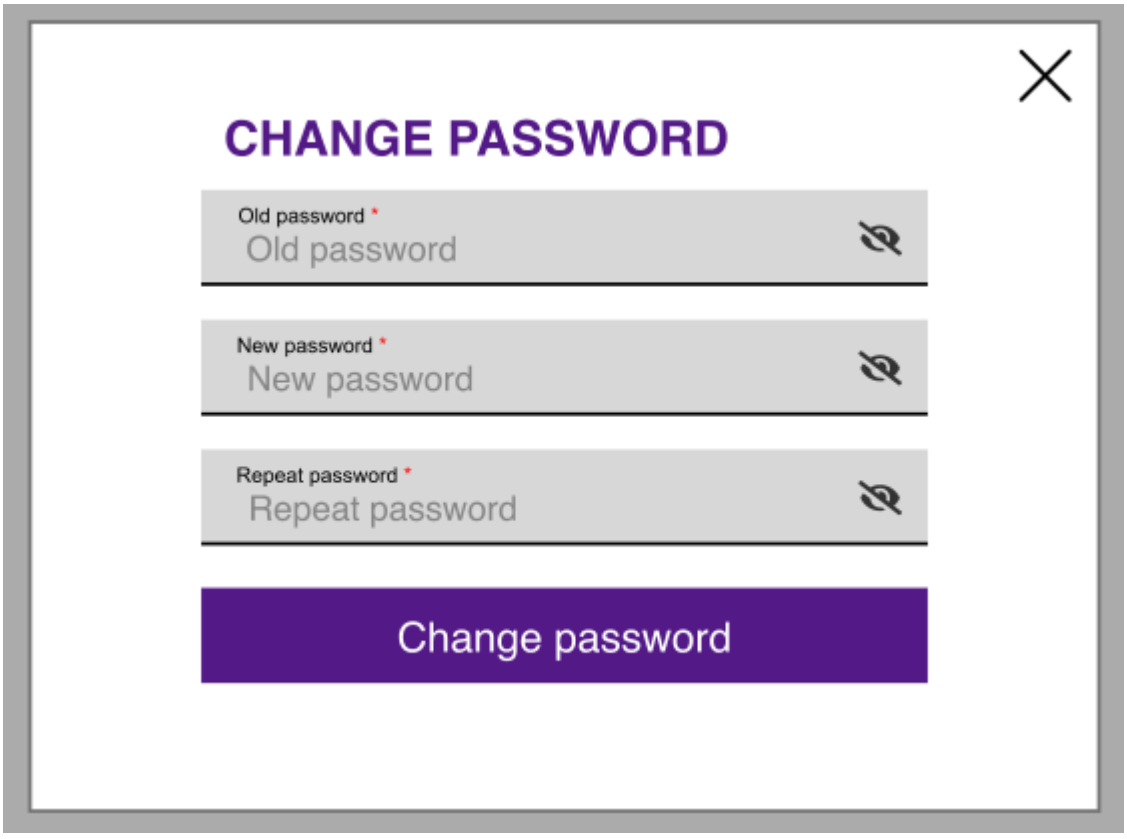


Рис. 3.6 Вікно для зміни паролю в обліковому записі користувача

«Rooms» − кімнати. Наявні в системі та включені наразі кімнати (рис. 3.7):

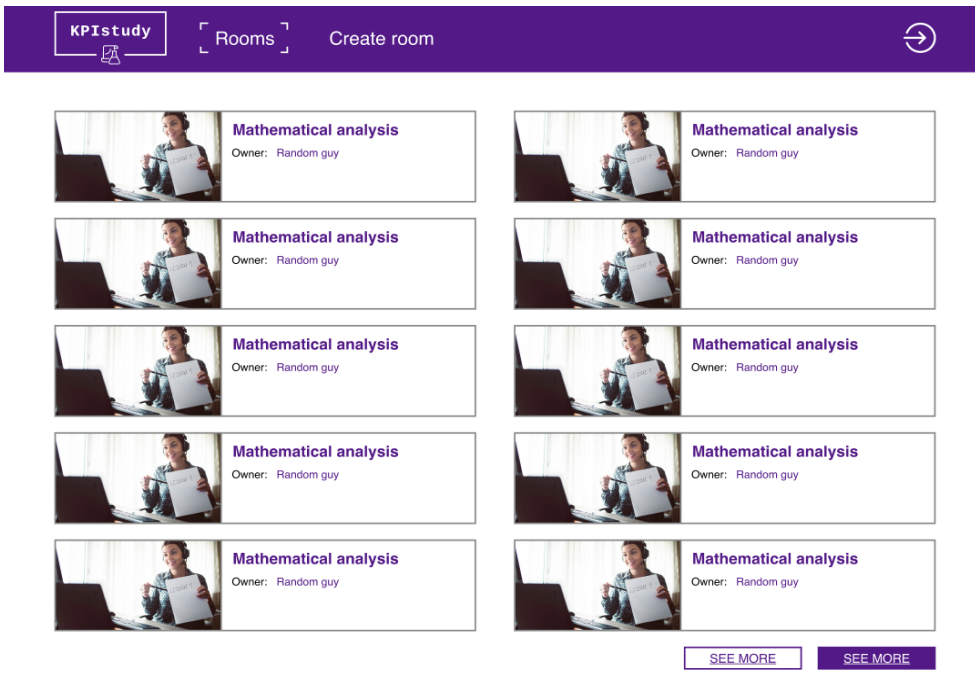


Рис. 3.7 Вікно обрання доступних кімнат

Обравши потрібну кімнату, користувачу (за наявності) слід буде вписати відповідний пароль, встановлений адміністратором кімнати (рис. 3.8):

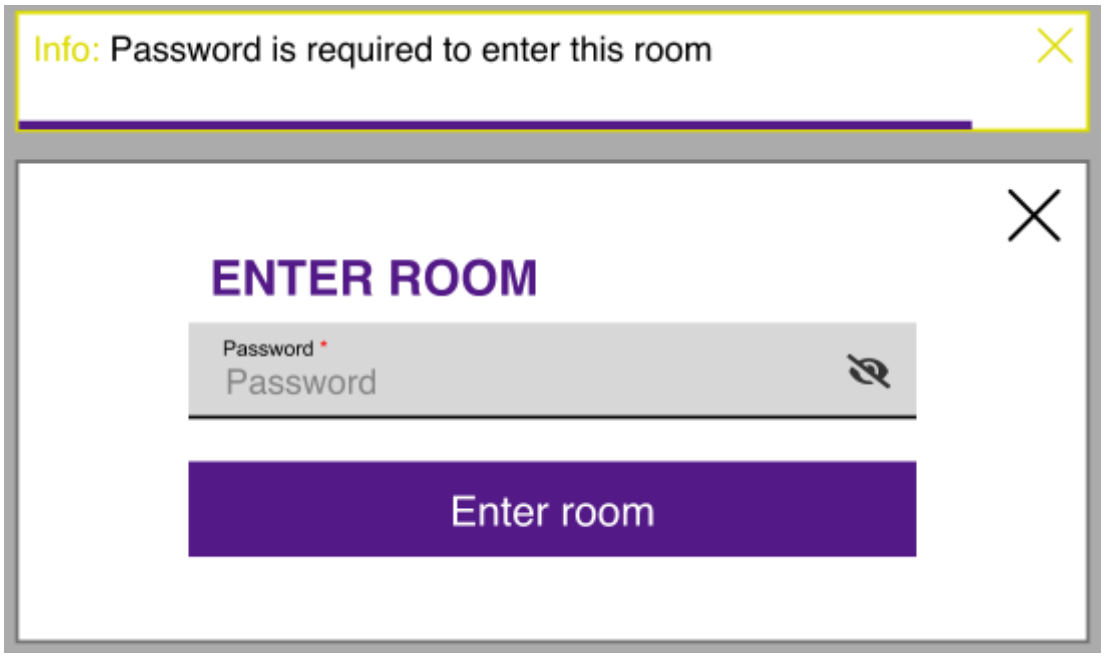


Рис. 3.8 Вікно для введення користувачем паролю

Після того як пароль (за наявності) був введений правильно, користувачу пропонується обрати аудіо та відео налаштування (рис. 3.9):

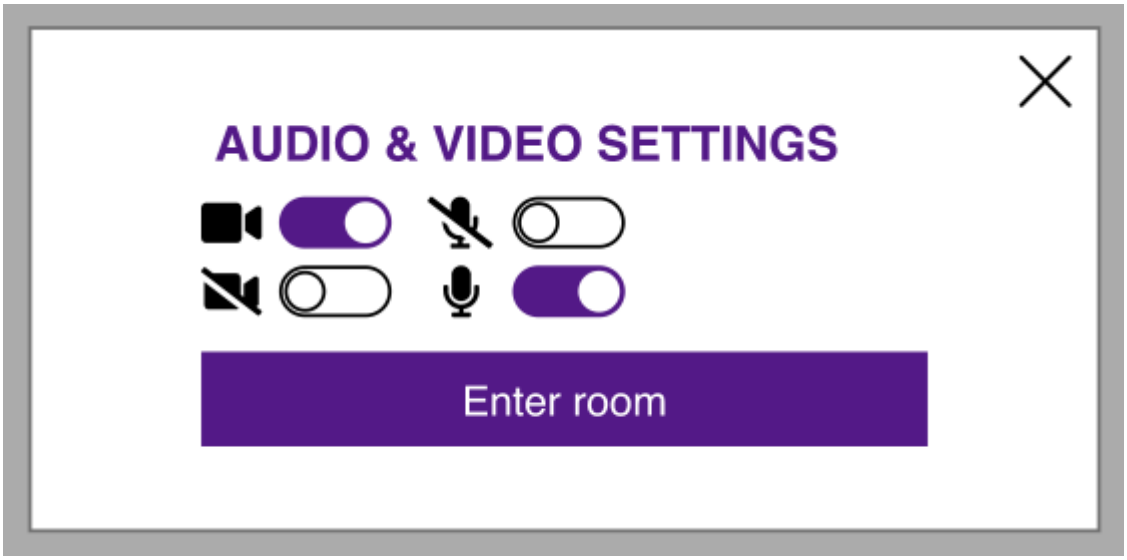


Рис. 3.9 Аудіо та відео налаштування

За наслідками цих всіх простих дій користувач потрапляє до відповідної обраної кімнати (рис. 3.10):

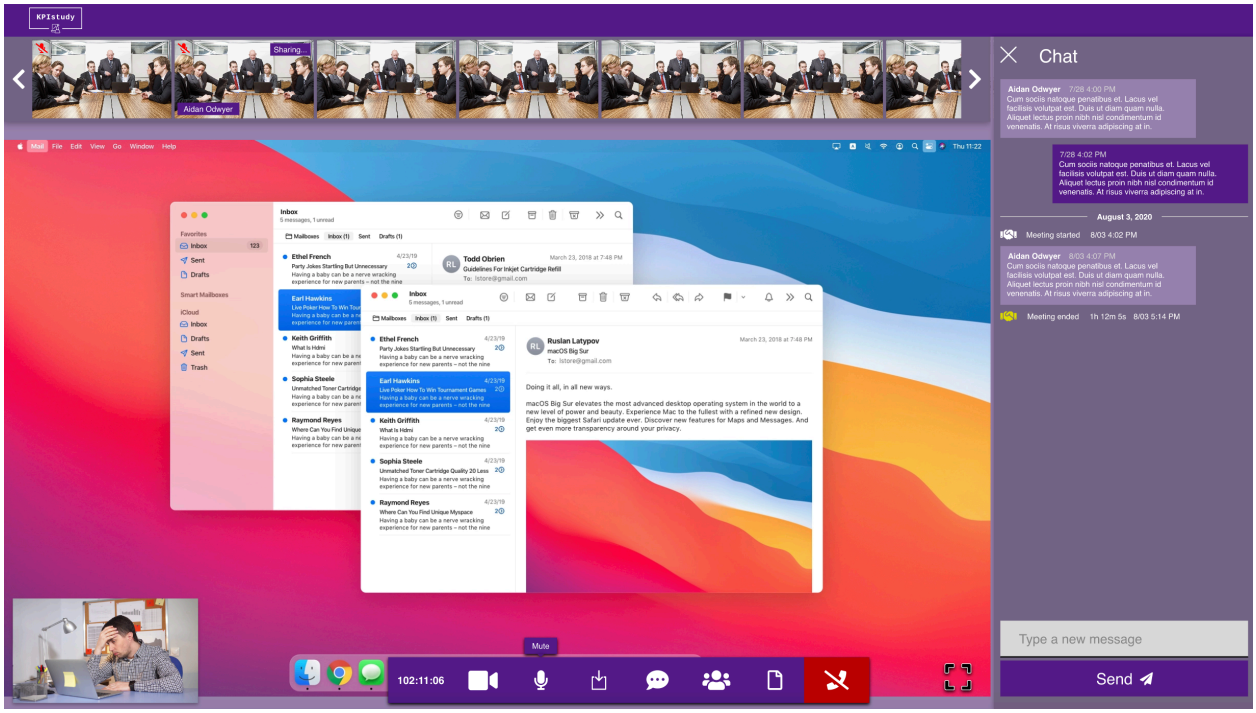


Рис. 3.10 Обрана користувачем кімната

В обраній кімнаті користувач може використовувати увесь функціонал: сприймати та відправляти звуки, відео, картинки, тощо.

«Create room» − створення кімнати. Функціонал який дає можливість стати адміністратором, обрати ім’я кімнати, додати пароль, змінити фонове зображення, тощо (рис 3.11):

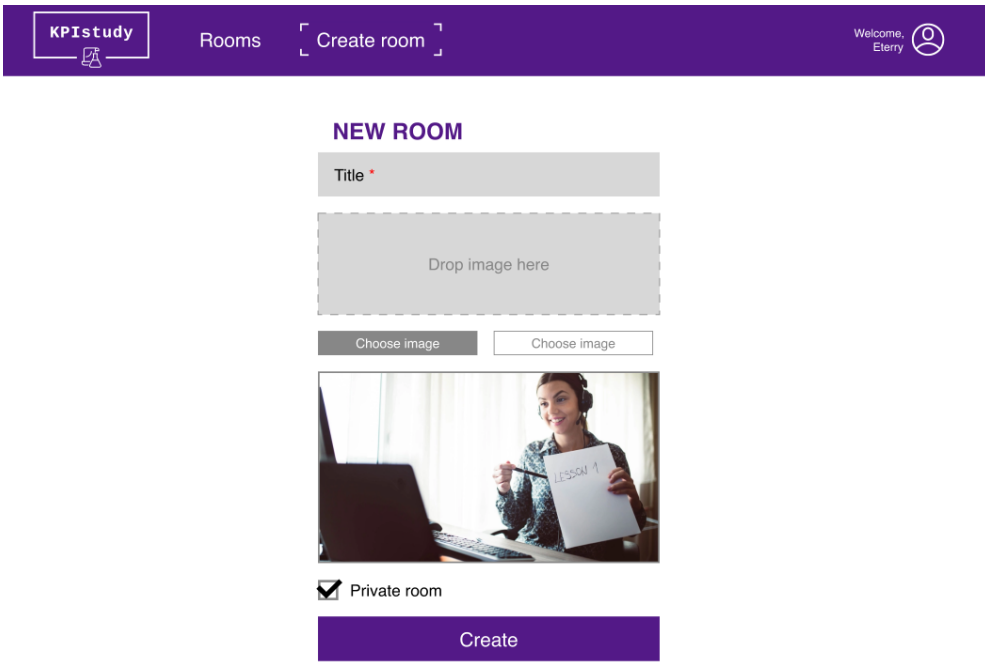


Рис. 3.11 Вікно створення нової кімнати

Розглянемо детальніше інтерфейс взаємодії із вебзастосунком всередині кімнати. В центрі екрану розміщений блок із відеопотоком із камери користувача, що наразі розмовляє, або ж екрану, що транслюється (рис. 3.12):

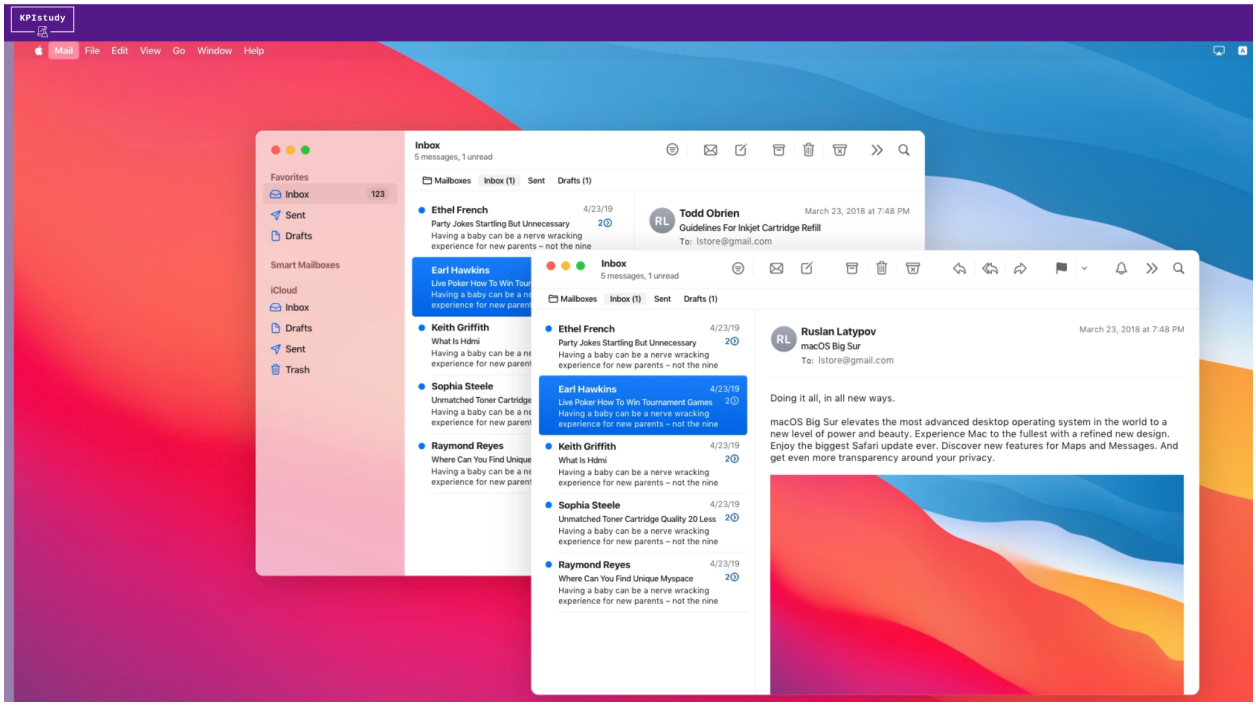


Рис. 3.12 Екран робочого стола, що транслюється

Вгорі сторінки знаходиться карусель із учасниками поточного дзвінка, де можна побачити статуси кожного користувача (рис. 3.13):

Рис. 3.13 Панель учасників поточного дзвінка

Внизу сторінки знаходиться панель для керування аудіо та відео виводом, трансляції екрану робочого столу, відкриття допоміжних панелей таких як чат та список учасників, панель для керування збереженими файлами, та кнопка щоб вийти із поточного дзвінка (рис. 3.14):

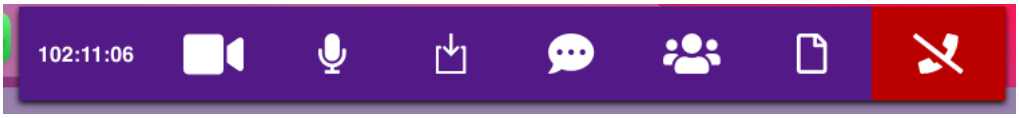


Рис. 3.14 Панель керування

При взаємодії із панеллю керування можна відкрити бокову панель чату, де можна побачити історії повідомлень та дзвінків, скільки часу вони тривали,

які користувачі долучались до дзвінка та виходили з нього, час даних дій, а також відправити нове повідомлення (рис. 3.15). При відкритті панелі учасників дзвінку можна отримати інформацію про кожного із учасників, чи увімкнуті у них мікрофон та камера. Для користувача, що створив дану кімнату, також доступні дії, щоб примусово виключити аудіо та відео потрібного користувача, або ж виключити його із поточного дзвінка (рис. 3.16):

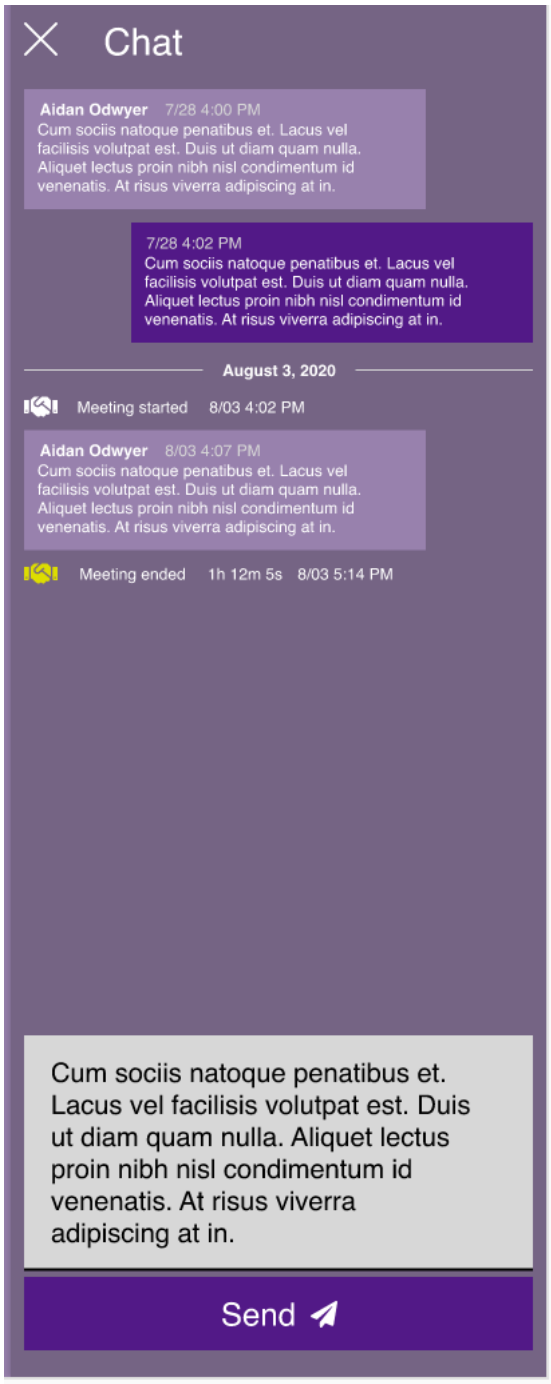


Рис. 3.14 Бокова панель для чату

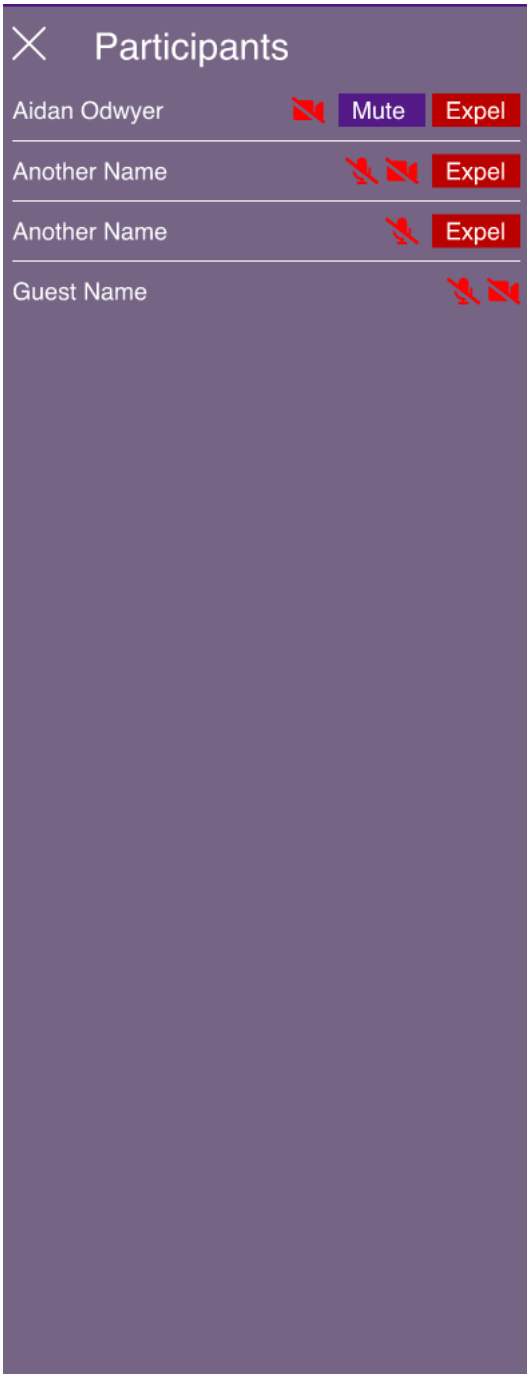


Рис. 3.16 Панель учасників дзвінка

ВИСНОВКИ ДО РОЗДІЛУ 4:

В цьому розділі було презентований зовнішній вигляд запропонованого до розгляду вебзастосунку для проведення онлайн занять.

Також виділена зовнішня будова та окремо звернено увагу саме на відповідну модель з’єднання користувачів.

Насамкінець проведений детальний огляд запропонованого вебзастосунку.

Отже, у даного вебзастосунка наявний весь основний функціонал, необхідний для проведення онлайн-занять, має досить простий для користування інтерфейс, що дозволить потенційним користувачам швидко звикнути до інтерфейсу взаємодії.

ВИСНОВКИ

В процесі роботи над дипломним проєктом наступні цілі було успішно досягнуто:

- Зроблено аналіз аналогів даного вебзастосунку;
- В процесі опрацювання даних, що були отримані, були створенні вимоги за допомогою яких було розроблено систему;
- На основі сформульованих критеріїв було підібрано оптимальні технології, які б змогли ефективно виконати поставлені задачі;
- Імплементована система;

Під час реалізації вебзастосунку вся система була декілька раз протестована щоб перевірити її коректну роботу.

Можна зробити висновок, що маючи певний набір універсальних програмних модулів та компонентів, та навичок роботи з ними, можна створювати будь-які програмні продукти та системи.

Як результат виконання проєкту було отримано продукт, який вже має весь необхідний для кінцевого користувача функціонал, що дозволить в короткі терміни почати бета тестування системи із цільовими користувачами. В процесі виконання було обрано такий підхід, щоб система була гнучкою для розширення, що дозволить покращувати вебзастосунок в майбутньому без надлишкових зусиль.

На основі отриманої системи, її функціоналу та роботи, можна вважати, що всі поставлені задачі було виконано, а мета дипломної роботи була досягнута.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ:

1. Ashish P., «Distance Learning: History, Problems and Solutions», 2014.

2. Keegan, D., «The Foundations of Distance Education. London: Croom Helm», 1986.

3. Спілкування по Скайпу: необхідне ПО і як воно працює, 2009 // [Режим доступу – електронне джерело: https://cutt.ly/hbS1sQq]

4. Mark G., CVP Skype Product Engineering & Operations, «How the Skype architecture works», 2012.

5. Как это работает: Skype //[Режим доступу – електронне джерело: https://aif.ru/society/web/20405].

6. Офіційний сайт Skype // [Режим доступу – електронне джерело: https://www.skype.com/].

7. Офіційний сайт Discord [Режим доступу – електронне джерело: https://discord.com/].

8. Jozsef V., «How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC», 2018.

9. WebRTC 1.0: Real-Time Communication Between Browsers // [Режим доступу – електронне джерело: https://www.w3.org/TR/webrtc/].

10. Stanislav V., «How Discord Stores Billions of Messages», 2017.

11. Zoom. Architected for Reliability Режим доступу – електронне джерело: https://zoom.us/docs/doc/Zoom_Global_Infrastructure.pdf].

12. Офіційний сайт Zoom [Режим доступу – електронне джерело: https://zoom.us/].

13. Наскрізне шифрування (E2EE) конференцій // [Режим доступу – електронне джерело: https://cutt.ly/IbS0vEs].

14. Офіційний сайт React // [Режим доступу – електронне джерело: https://ru.reactjs.org/].

15. Що таке React і як він працює насправді? // [Режим доступу – електронне джерело: https://www.hostinger.com.ua/rukovodstva/chto-takoe-react].

16. Офіційний сайт redux-saga // [Режим доступу – електронне джерело: https://ru.redux-saga.js.org/].
17. Офіційний сайт TypeScript // [Режим доступу – електронне джерело: https://www.typescriptlang.org/].
18. Introduction NestJS // [Режим доступу – електронне джерело: https://docs.nestjs.com/].
19. Martin F., «Microservices a definition of this new architectural term» // [Режим доступу – електронне джерело: https://martinfowler.com/articles/microservices.html].
20. Stack Overflow // [Режим доступу – електронне джерело: https://cutt.ly/ibS06RA].
21. Learning Docker // [Режим доступу – електронне джерело: https://www.hostinger.com.ua/rukovodstva/zapuskaem-docker-konteiner].
22. Офіційний сайт mongodb // [Режим доступу – електронне джерело: https://www.mongodb.com/].
23. Керівництво по MongoDB // [Режим доступу – електронне джерело: https://cutt.ly/vbS2f07].
24. About HTML5 WebSocket // [Режим доступу – електронне джерело: https://www.websocket.org/aboutwebsocket.html] .
25. Interface RTCPeerConnection // [Режим доступу – електронне джерело: https://developer.mozilla.org/ru/docs/Web/API/RTCPeerConnection].
26. JWT − як безпечний спосіб аутентифікації і передачі даних // [Режим доступу – електронне джерело: https://vc.ru/dev/106534-jwt-kak-bezopasnyy-sposob-autentifikacii-i-peredachi-dannyh].
27. Офіціальний сайт Cloudinary // [Режим доступу – електронне джерело: https://cloudinary.com/faq].

ДОДАТОК 1

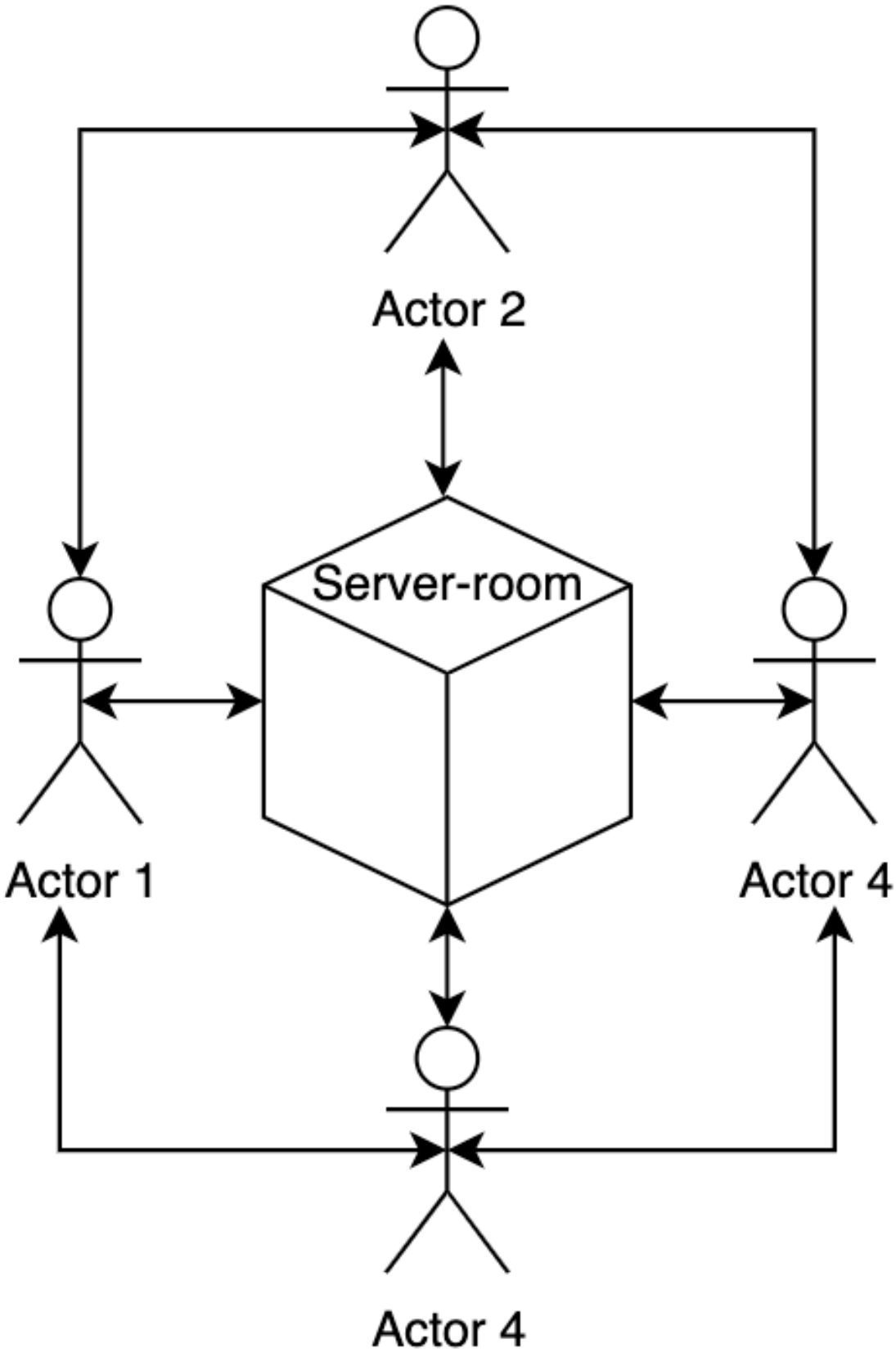
Вебзастосунок для проведення онлайн-занять

Схема внутрішнього взаємозв’язку користувачів

ІАЛЦ.467100 Д1

Листів 1

2021



					ІАЛЦ.467100.004 Д1	
Зм.		*№ документа*	*Підпис*	*Дата*		
Розробив		*Обух В.І.*			**Вебзастосунок для проведення онлайн-занять** Додаток 1	*Літ.* | *Аркуш* | *Аркушів*
Перевірив		*Сергієнко А.А.*				1 | 1
						НТУУ «КПІ» ФІОТ
Н. Контр.		Сімоненко В. П.				*Група ІВ-72*
Затвердив						

ДОДАТОК 2

Вебзастосунок для проведення онлайн-занять

Схема алгоритму підготовки клієнтських частин перед формуванням сигнальних даних

ІАЛЦ.467100 Д2

Листів 1

2021

					ІАЛЦ.467100.005 Д2	
Зм.		*№ документа*	*Підпис*	*Дата*		
Розробив		*Обух В.І.*			**Вебзастосунок для проведення онлайн-занять** Додаток 2	*Літ.* / *Аркуш* / *Аркушів*: 1 / 1
Перевірив		*Сергієнко А.А.*				
Н. Контр.		Сімоненко В. П.				*НТУУ «КПІ» ФІОТ*
Затвердив						*Група ІВ-72*

ДОДАТОК 3

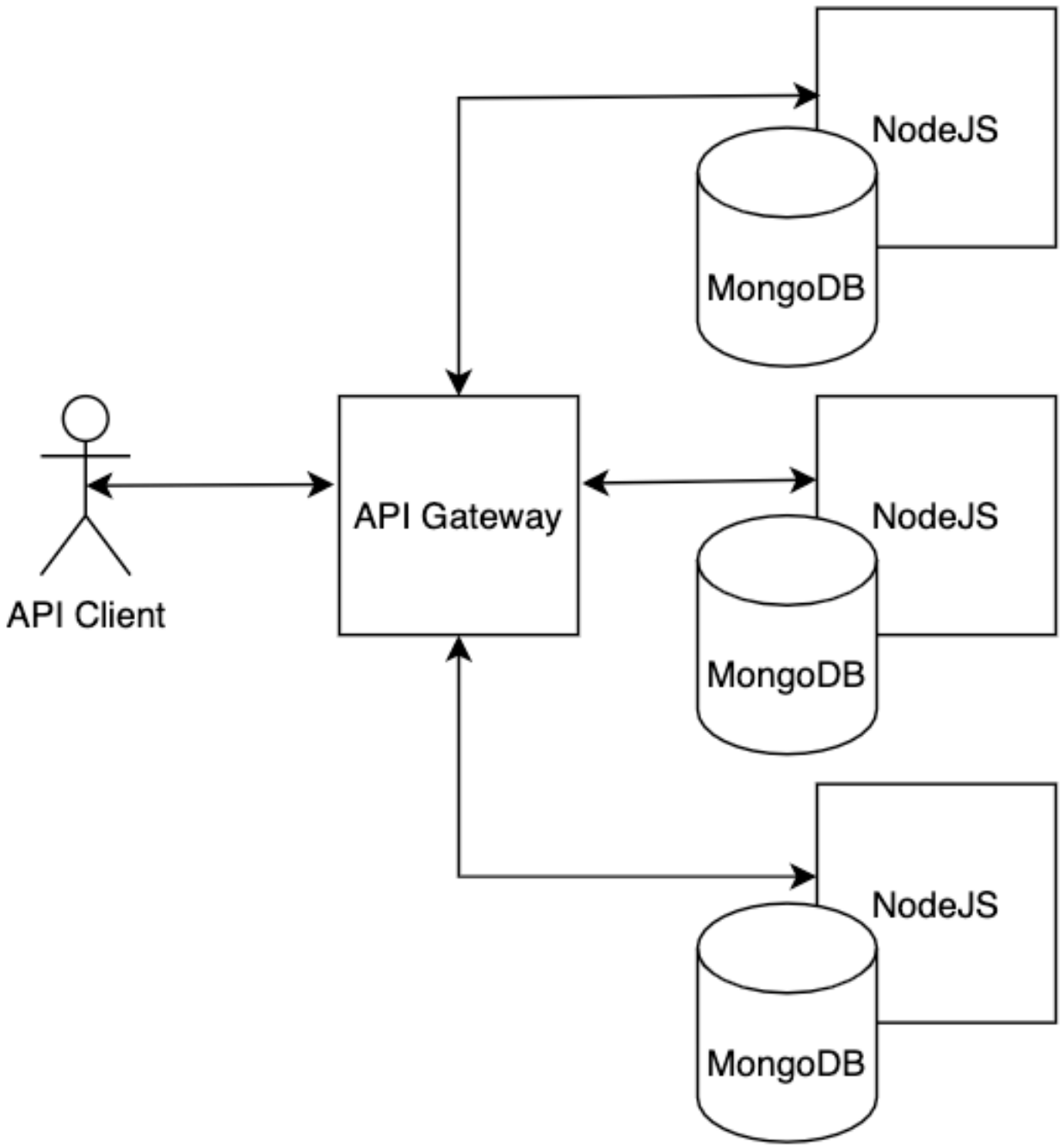
Вебзастосунок для проведення онлайн-занять

Структурна схема сервісів

ІАЛЦ.467100 Д3

Листів 1

2021



					ІАЛЦ.467100.006 Д3	
Зм.		*№ документа*	*Підпис*	*Дата*		
Розробив		*Обух В.І.*			**Вебзастосунок для проведення онлайн-занять** Додаток 3	*Літ.* / *Аркуш* / *Аркушів*: 1 / 1
Перевірив		*Сергієнко А.А.*				*НТУУ «КПІ» ФІОТ*
Н. Контр.		Сімоненко В. П.				*Група ІВ-72*
Затвердив						

ДОДАТОК 4

Вебзастосунок для проведення онлайн-занять

Лістинг програми

ІАЛЦ.467100 Д4

Листів 17

2021

```
export class UserDto {
  @IsNotEmpty({ message: ErrorMessages.EMPTY_NAME })
  @IsString({ message: ErrorMessages.NAME_TYPE })
  name: string;

  @IsNotEmpty({ message: ErrorMessages.EMPTY_EMAIL })
  @IsString({ message: ErrorMessages.EMAIL_TYPE })
  @IsEmail({}, { message: ErrorMessages.EMAIL_INVALID })
  email: string;

  @IsNotEmpty({ message: ErrorMessages.EMPTY_PASSWORD })
  @IsString({ message: ErrorMessages.PASSWORD_TYPE })
  @MinLength(8, { message: ErrorMessages.PASSWORD_INVALID_LENGTH })
  password: string;
}

@Schema()
export class User {
  _id: string;

  @Prop()
  name: string;

  @Prop()
  email: string;

  @Prop()
  password: string;
}

export type UserDocument = User & Document;

export type UserWithSafeFields = Omit<User, 'password'>;

export const UserSchema = SchemaFactory.createForClass(User);

UserSchema.pre<UserDocument>('save', async function(next) {
  const user: UserDocument = this;
  if (user.isModified('password')) {
    user.password = await bcrypt.hash(user.password, 8);
  }

  next();
});

@Injectable()
export class UserService {
  constructor(@InjectModel(User.name) private userModel: Model<UserDocument>)
{}

  static getUserSafeFields({ _id, name, email }: User): UserWithSafeFields {
    return { _id, name, email };
  }

  async createUser(userDto: UserDto): Promise<User> {
    return new this.userModel(userDto).save();
  }

  async findUserByEmail(email: string): Promise<User> {
    return await this.userModel.findOne({ email }).exec();
  }

  async findUserById(id: string): Promise<UserWithSafeFields> {
    return UserService.getUserSafeFields(
```

```
      await this.userModel.findById(id).exec(),
    );
  }

  async findUsersByIds(ids: string[]): Promise<UserWithSafeFields[]> {
    const users: User[] = await Promise.all(
      ids.map(id => this.userModel.findById(id).exec()),
    );

    return users.map(user => UserService
      .getUserSafeFields(user),
    );
  }
}

export const DtoMismatchValidationPipe = new ValidationPipe({
  exceptionFactory: (errors: ValidationError[]): BadRequestException => {
    return new BadRequestException(
      errors.map(error => Object.values(error.constraints)).flat(),
    );
  },
});

@Catch(UnauthorizedException)
export class UnauthorizedExceptionFilter implements ExceptionFilter {
  catch(exception: UnauthorizedException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const status = exception.getStatus();

    response.status(status).json({
      statusCode: status,
      error: exception.message,
      message: 'Credentials are not correct',
    });
  }
}

@Controller('auth')
export class AuthController {
  constructor(private authService: AuthService) {}

  @Post('register')
  async register(@Body() userDto: UserDto): Promise<AccessToken> {
    return await this.authService.register(userDto);
  }

  @UseGuards(LocalAuthGuard)
  @Post('login')
  async login(@Request() req): Promise<AccessToken> {
    return this.authService.getToken(req.user);
  }
}

@Module({
  imports: [
    UserModule,
    PassportModule,
    JwtModule.register({
      secret: process.env.JWT_SECRET,
      signOptions: { expiresIn: '600m' },
    }),
  ],
  providers: [AuthService, LocalStrategy, JwtStrategy],
```

```
  controllers: [AuthController, AuthRpcController],
  exports: [AuthService, JwtModule],
})
export class AuthModule {}

@Controller()
export class AuthRpcController {
  constructor(
    private readonly authService: AuthService,
    private readonly userService: UserService,
  ) {}

  @MessagePattern({ cmd: 'validate-jwt' })
  async validateToken(token: string) {
    try {
      const { sub: userId } = this.authService.validateToken(token);
      return this.userService.findUserById(userId);
    } catch (error) {
      Logger.error(error);
      return false;
    }
  }
}

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly userService: UserService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: process.env.JWT_SECRET,
    });
  }

  async validate(payload: any): Promise<UserWithSafeFields | null> {
    return this.userService.findUserById(payload.sub);
  }
}

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super({
      usernameField: 'email',
      passwordField: 'password',
    });
  }

  async validate(
    email: string,
    password: string,
  ): Promise<UserWithSafeFields> | never {
    const user: UserWithSafeFields = await this.authService.validateUser(
      email,
      password,
    );
    if (!user) {
      throw new UnauthorizedException();
    }
```

```
    return user;
  }
}
```

```
#!/bin/sh
unset RUN_DOCKERIZED

while getopts 'd' opt
do
  case $opt in
    d) RUN_DOCKERIZED=true;;
    *) echo "Error: Invalid flag";;
  esac
done

if [ "$RUN_DOCKERIZED" = true ]
then
  docker-compose up
else
  . ./.env
  export IP_ADDRESS=$(ipconfig getifaddr en0)
  export API_ROOT_URL=http://localhost
  export WS_URL=http://localhost:$ROOM_SERVICE_PORT
  export JWT_SECRET=$JWT_SECRET
  export CLOUDINARY_CLOUD_NAME=$CLOUDINARY_CLOUD_NAME
  export CLOUDINARY_API_KEY=$CLOUDINARY_API_KEY
  export CLOUDINARY_API_SECRET=$CLOUDINARY_API_SECRET
  export MONGODB_URI=mongodb://localhost:$MONGO_DB_PORT/$DB_NAME
  export FAST_REFRESH=false
  export USER_SERVICE_PORT=$USER_SERVICE_PORT

  export PORT=$CONFIG_SERVICE_PORT && cd ./config-service && npm run
start:dev &
  export PORT=$ROOM_SERVICE_PORT && cd ../room-service && npm run start:dev &
  export PORT=$USER_SERVICE_PORT && cd ../user-service && npm run start:dev &
  export PORT=$CLIENT_PORT && cd ../client && npm run start &

  cd .. && docker-compose -f docker-compose-nginx.yml up
fi
```

```
version: "3.8"

services:
  nginx:
    container_name: kpi-study-proxy
    build: ./proxy
    env_file:
      - .env
    environment:
      PORT: ${PROXY_PORT}
      CONFIG_SERVICE_URL: "http://${IP_ADDRESS}:${CONFIG_SERVICE_PORT}"
      ROOM_SERVICE_URL: "http://${IP_ADDRESS}:${ROOM_SERVICE_PORT}"
      USER_SERVICE_URL: "http://${IP_ADDRESS}:${USER_SERVICE_PORT}"
    ports:
      - ${PROXY_PORT}:${PROXY_PORT}
```

```
interface Peer extends User {
  peerId: string;
}

type Peers = Record<string, Peer>;
```

```
type Rooms = Record<string, Peers>;

@WebSocketGateway()
export class AppGateway implements OnGatewayDisconnect {
  @WebSocketServer()
  server: Server;

  peers: Peers = {};
  rooms: Rooms = {};

  constructor(
    @Inject('USER_SERVICE')
    private readonly userClient: ClientProxy,
    private readonly roomService: RoomService,
  ) {}

  getRoomDataByPeerId(id: string) {
    return Object.entries(this.rooms).find((room) => id in room[1]) || [];
  }

  handleDisconnect(client) {
    const [roomId, peers] = this.getRoomDataByPeerId(client.id);
    if (!peers) return;
    delete peers[client.id];
    client.in(roomId).emit('disconnectPeer', client.id);
  }

  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('enter-room')
  async handleRoomEntering(
    @MessageBody() roomId: string,
    @ConnectedSocket() client: Socket,
  ) {
    const room: Room = await this.roomService.getRoomById(roomId);
    if (!room) throw new WsException('room was not found');

    client.join(room._id);
    if (!this.rooms[room._id]) {
      this.rooms[room._id] = {};
    }

    if (!this.rooms[roomId][client.id]) {
      this.rooms[roomId][client.id] = {
        ...client.user,
        peerId: client.id,
      };
    }

    client.emit(
      'sendPeerList',
      this.getPeersWithoutTargetClient(this.rooms[roomId], client),
    );
  }

  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('signal')
  handleSignal(
    @MessageBody() data: { to: string; signal: string },
    @ConnectedSocket() client: Socket,
  ) {
    this.server.to(data.to).emit('signal', {
      signal: data.signal,
      callerId: client.id,
    });
```

```
  }

  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('call')
  handleCall(@MessageBody() callee: string, @ConnectedSocket() client:
Socket) {
    const [roomId] = this.getRoomDataByPeerId(client.id);
    this.server.to(callee).emit('call', this.rooms[roomId][client.id]);
  }

  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('sendMessage')
  handleIncomingMessage(
    @MessageBody() message: string,
    @ConnectedSocket() client: Socket,
  ) {
    const [roomId] = this.getRoomDataByPeerId(client.id);
    this.server.in(roomId).emit('sendMessage', {
      id: generateId(),
      text: message,
      date: new Date(),
      author: client.user,
    });
  }

  getPeersWithoutTargetClient(peers: Peers, client: Socket): Peers {
    return Object.fromEntries(
      Object.entries(peers).filter(([id]) => id !== client.id),
    );
  }
}

const PORT = parseInt(process.env.PORT) || 4042;

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.connectMicroservice({
    transport: Transport.TCP,
    options: {
      port: PORT,
    },
  });

  app.use(json({ limit: '50mb' }));
  app.use(urlencoded({ extended: true, limit: '50mb' }));
  app.useGlobalFilters(new UnauthorizedExceptionFilter());
  app.useGlobalPipes(DtoMismatchValidationPipe);

  await app.startAllMicroservicesAsync();
  await app.listen(PORT);
}

bootstrap();

@Injectable()
export class CloudinaryService {
  constructor() {
    cloudinary.config({
      cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
      api_key: process.env.CLOUDINARY_API_KEY,
      api_secret: process.env.CLOUDINARY_API_SECRET,
    });
  }
```

```
  uploadImage(image: string): Promise<{ secure_url: string }> {
    return cloudinary.uploader.upload(image);
  }
}

interface Peer extends User {
  peerId: string;
}

type Peers = Record<string, Peer>;

type Rooms = Record<string, Peers>;

@WebSocketGateway()
export class AppGateway implements OnGatewayDisconnect {
  @WebSocketServer()
  server: Server;

  peers: Peers = {};
  rooms: Rooms = {};

  constructor(
    @Inject('USER_SERVICE')
    private readonly userClient: ClientProxy,
    private readonly roomService: RoomService,
  ) {}

  getRoomDataByPeerId(id: string) {
    return Object.entries(this.rooms).find((room) => id in room[1]) || [];
  }

  handleDisconnect(client) {
    const [roomId, peers] = this.getRoomDataByPeerId(client.id);
    if (!peers) return;
    delete peers[client.id];
    client.in(roomId).emit('disconnectPeer', client.id);
  }

  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('enter-room')
  async handleRoomEntering(
    @MessageBody() roomId: string,
    @ConnectedSocket() client: Socket,
  ) {
    const room: Room = await this.roomService.getRoomById(roomId);
    if (!room) throw new WsException('room was not found');

    client.join(room._id);
    if (!this.rooms[room._id]) {
      this.rooms[room._id] = {};
    }

    if (!this.rooms[roomId][client.id]) {
      this.rooms[roomId][client.id] = {
        ...client.user,
        peerId: client.id,
      };
    }

    client.emit(
      'sendPeerList',
      this.getPeersWithoutTargetClient(this.rooms[roomId], client),
    );
  }
```

```
  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('signal')
  handleSignal(
    @MessageBody() data: { to: string; signal: string },
    @ConnectedSocket() client: Socket,
  ) {
    this.server.to(data.to).emit('signal', {
      signal: data.signal,
      callerId: client.id,
    });
  }

  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('call')
  handleCall(@MessageBody() callee: string, @ConnectedSocket() client:
Socket) {
    const [roomId] = this.getRoomDataByPeerId(client.id);
    this.server.to(callee).emit('call', this.rooms[roomId][client.id]);
  }

  @UseGuards(JwtWSAuthGuard)
  @SubscribeMessage('sendMessage')
  handleIncomingMessage(
    @MessageBody() message: string,
    @ConnectedSocket() client: Socket,
  ) {
    const [roomId] = this.getRoomDataByPeerId(client.id);
    this.server.in(roomId).emit('sendMessage', {
      id: generateId(),
      text: message,
      date: new Date(),
      author: client.user,
    });
  }

  getPeersWithoutTargetClient(peers: Peers, client: Socket): Peers {
    return Object.fromEntries(
      Object.entries(peers).filter(([id]) => id !== client.id),
    );
  }
}
import {
  Body,
  Controller,
  Get,
  ParseIntPipe,
  Post,
  Query,
  UseGuards,
} from '@nestjs/common';

@Controller('data')
export class RoomController {
  constructor(private readonly roomService: RoomService) {}

  @UseGuards(JwtAuthGuard)
  @Post()
  async createRoom(
    @Body() roomDto: RoomDto,
    @UserData() user: User,
  ): Promise<void> {
    await this.roomService.createRoom({
      ...roomDto,
```

```
      ownerId: user._id,
    });
  }

  @Get()
  async getPublicRooms(
    @Query('amount', ParseIntPipe) amount = 0,
  ): Promise<PopulatedRoom[]> {
    return this.roomService.getPublicPopulatedRooms(amount);
  }
}

@Schema()
export class Room {
  _id: string;

  @Prop()
  title: string;

  @Prop()
  image: string;

  @Prop()
  private: boolean;

  @Prop()
  ownerId: string;
}

export type RoomDocument = Room & Document;
export type PopulatedRoom = Omit<Room, 'ownerId'> & {
  owner: User;
};

export const RoomSchema = SchemaFactory.createForClass(Room);

type RoomCreationData = RoomDto & {
  ownerId: string;
};

@Injectable()
export class RoomService {
  constructor(
    @Inject('USER_SERVICE')
    private readonly client: ClientProxy,
    @InjectModel(Room.name)
    private readonly roomModel: Model<RoomDocument>,
    private readonly cloudinaryService: CloudinaryService,
  ) {}

  private static getUniqueOwnerIds(rooms: Room[]): string[] {
    return [...new Set(rooms.map((room) => room.ownerId))];
  }

  private static mergeOwnersIntoRooms(
    rooms: Room[],
    users: User[],
  ): PopulatedRoom[] {
    return rooms.map(({ ownerId, ...room }) => ({
      ...room,
      owner: users.find(({ _id }) => _id === ownerId),
    }));
  }
```

```
  async createRoom(roomCreationData: RoomCreationData): Promise<Room> {
    const { secure_url: image } = await this.cloudinaryService.uploadImage(
      roomCreationData.image,
    );

    return new this.roomModel({ ...roomCreationData, image }).save();
  }

  async getPublicRooms(amount = 0): Promise<Room[]> {
    const rooms: RoomDocument[] = await this.roomModel
      .find({ private: false }, null, {
        limit: amount,
      })
      .exec();

    return rooms.map((room) => room.toObject());
  }

  async getPublicPopulatedRooms(amount: number): Promise<PopulatedRoom[]> {
    const rooms: Room[] = await this.getPublicRooms(amount);
    const users: User[] = await this.client
      .send({ cmd: 'find-users-by-ids' },
RoomService.getUniqueOwnerIds(rooms))
      .toPromise();

    return RoomService.mergeOwnersIntoRooms(rooms, users);
  }

  async getRoomById(roomId: string): Promise<Room> {
    if (!Types.ObjectId.isValid(roomId)) return null;
    return this.roomModel.findById(roomId).exec();
  }
}

export class RoomDto {
  @IsNotEmpty({ message: ErrorMessages.EMPTY_TITLE })
  @IsString({ message: ErrorMessages.TITLE_TYPE })
  title: string;

  @IsNotEmpty({ message: ErrorMessages.EMPTY_IMAGE })
  @IsString({ message: ErrorMessages.IMAGE_TYPE })
  image: string;

  @IsNotEmpty({ message: ErrorMessages.EMPTY_PRIVACY })
  @IsBoolean({ message: ErrorMessages.PRIVACY_TYPE })
  private: boolean;
}

@Injectable()
export class JwtWSAuthGuard implements CanActivate {
  constructor(
    @Inject('USER_SERVICE')
    private readonly userClient: ClientProxy,
  ) {}

  static extractJwtFromClient(client: Socket): string {
    return client.handshake.headers.authorization.split(' ')[1];
  }

  static getUserData(client: Socket, userClient: ClientProxy): Promise<User>
{
    const jwt = JwtWSAuthGuard.extractJwtFromClient(client);

    return userClient.send({ cmd: 'validate-jwt' }, jwt).toPromise();
```

```
  }

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const client: Socket = context.switchToWs().getClient();
    const user: User = await JwtWSAuthGuard.getUserData(
      client,
      this.userClient,
    );

    if (user) {
      client.user = user;
      return true;
    }

    throw new WsException('Unauthorized');
  }
}

import { PayloadAction } from '@reduxjs/toolkit';
import { EventChannel } from 'redux-saga';
import { all, call, delay, fork, put, take, takeLatest, select, spawn } from
'redux-saga/effects';
import Peer, { Instance as PeerInstance, SignalData } from 'simple-peer';

import * as globalActions from 'store/global';
import { MessageType } from 'store/global/types';

import * as actions from '.';
import * as sagaActions from './actions';
import { selectCurrentPeerSharingState, selectCurrentPeerStream,
selectPeerById, selectPeers } from './selectors';
import * as Api from './api';
import { Message, Peer as PeerInterface } from './types';
import { createEventChannel, createEventListenerChannel,
createSocketEventChannel } from './utils/eventChannel';
import * as StreamService from './utils/streamService';
import { PeerService } from './services/peer';

function* initRoom({ payload: roomId }: PayloadAction<string>) {
  try {
    yield call(setupMedia);

    Api.connect();
    Api.emit('enter-room', roomId);
    yield spawn(watchPeerListReceive);
    yield spawn(watchSocketSignal);
    yield spawn(watchIncomingCall);
    yield spawn(watchSocketDisconnectPeer);
    yield spawn(watchMessagesReceive);
  } catch (error) {
    yield put(globalActions.addMessage({
      type: MessageType.Error,
      text: error.message
    }));
  }
}

function* setupMedia() {
  const stream: MediaStream = yield navigator.mediaDevices.getUserMedia({
    video: true,
    audio: true,
  });

  StreamService.addStream(stream);
```

```
  yield put(actions.setCurrentPeerMediaStreamId(stream.id));
  yield put(actions.setCurrentPeerMutedState(false));
  yield put(actions.setCurrentPeerCameraState(true));
}

function* watchPeerListReceive() {
  const socketEventChannel: EventChannel<PeerInterface[]> = yield
call(createSocketEventChannel, 'sendPeerList');

  while (true) {
    try {
      const peerList: PeerInterface[] = yield take(socketEventChannel);
      yield put(actions.setPeers(Object.values(peerList)));

      yield call(callPeers);
    } catch (error) {
      socketEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

function* callPeers() {
  const peersToCall: PeerInterface[] = yield select(selectPeers);

  yield all(
    Object.values(peersToCall)
      .map(peer => fork(callPeer, peer))
  );
}

function* callPeer(peer: PeerInterface) {
  yield call(addPeer, peer.peerId, true);
  Api.emit('call', peer.peerId);
}

function* addPeer(peerId: string, isInitiator: boolean = false) {
  const currentPeerStream: MediaStream = yield
select(selectCurrentPeerStream);

  PeerService.addPeer(peerId, new Peer({
    initiator: isInitiator,
    trickle: false,
    stream: currentPeerStream,
  }));

  yield spawn(watchPeerSignal, peerId);
  yield spawn(watchPeerStream, peerId);
  yield spawn(watchPeerData, peerId);
}

function* watchIncomingCall() {
  const socketEventChannel: EventChannel<string> = yield
call(createSocketEventChannel, 'call');

  while (true) {
    try {
      const caller: PeerInterface = yield take(socketEventChannel);
      yield put(actions.addPeer(caller));
      yield call(addPeer, caller.peerId);
    } catch (error) {
```

```
      socketEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

function* watchSocketSignal() {
  const socketEventChannel: EventChannel<string> = yield
call(createSocketEventChannel, 'signal');

  while (true) {
    try {
      const signalData: { callerId: string; signal: SignalData } = yield
take(socketEventChannel);
      const peer: PeerInstance | undefined =
PeerService.getPeerById(signalData.callerId);

      peer?.signal(signalData.signal);
    } catch (error) {
      socketEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

function* watchPeerSignal(peerId: string) {
  const peer: PeerInstance | undefined = PeerService.getPeerById(peerId);
  // @ts-ignore
  const peerEventChannel: EventChannel<string> = yield
call(createEventChannel, peer, 'signal');

  while (true) {
    try {
      const signal: SignalData = yield take(peerEventChannel);

      Api.emit('signal', {
        signal,
        to: peerId,
      });
    } catch (error) {
      peerEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

function* watchPeerStream(peerId: string) {
  const peer: PeerInstance | undefined = PeerService.getPeerById(peerId);
  // @ts-ignore
  const peerEventChannel: EventChannel<string> = yield
call(createEventChannel, peer, 'stream');

  while (true) {
    try {
      const stream: MediaStream = yield take(peerEventChannel);
```

```
      const audioTracks: MediaStreamTrack[] = stream.getAudioTracks();
      const videoTracks: MediaStreamTrack[] = stream.getVideoTracks();

      StreamService.addStream(stream);
      yield put(actions.setPeerMediaStreamId({ peerId, streamId: stream.id
}));
      yield put(actions.setPeerMutedState({
        peerId,
        mute: audioTracks.length ? !audioTracks[0].enabled : true,
      }));
      yield put(actions.setPeerCameraState({
        peerId,
        isCameraOn: videoTracks.length ? videoTracks[0].enabled : false,
      }));
    } catch (error) {
      peerEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

enum PeerDataTypes {
  TOGGLE_AUDIO = 'TOGGLE_AUDIO',
  TOGGLE_CAMERA = 'TOGGLE_CAMERA',
  TOGGLE_SCREEN_SHARE = 'TOGGLE_SCREEN_SHARE',
}

type PeerDataHandlers = Record<PeerDataTypes, (peerId: string) => void>;

const peerDataHandlers: PeerDataHandlers = {
  *[PeerDataTypes.TOGGLE_AUDIO](peerId: string) {
    const peer: PeerInterface = yield select(state =>
selectPeerById(state)(peerId));
    yield put(actions.setPeerMutedState({ peerId, mute: !peer.isMuted }));
  },
  *[PeerDataTypes.TOGGLE_CAMERA](peerId: string) {
    const peer: PeerInterface = yield select(state =>
selectPeerById(state)(peerId));
    yield put(actions.setPeerCameraState({ peerId, isCameraOn:
!peer.isCameraOn }));
  },
  *[PeerDataTypes.TOGGLE_SCREEN_SHARE](peerId: string) {
    const peer: PeerInterface = yield select(state =>
selectPeerById(state)(peerId));
    yield delay(500);
    yield put(actions.setPeerSharingState({ peerId, isSharing:
!peer.isSharing }));
  }
};

function* watchPeerData(peerId: string) {
  const peer: PeerInstance | undefined = PeerService.getPeerById(peerId);
  // @ts-ignore
  const peerEventChannel: EventChannel<string> = yield
call(createEventChannel, peer, 'data');

  while (true) {
    try {
      const peerDataBuffer: Uint8Array = yield take(peerEventChannel);
      const peerDataType: PeerDataTypes = peerDataBuffer.toString() as
PeerDataTypes;
```

```
      yield call(peerDataHandlers[peerDataType], peerId);
    } catch (error) {
      peerEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

function* watchSocketDisconnectPeer() {
  const socketEventChannel: EventChannel<string> = yield
call(createSocketEventChannel, 'disconnectPeer');

  while (true) {
    try {
      const peerId: string = yield take(socketEventChannel);
      const peer: PeerInterface = yield select(state =>
selectPeerById(state)(peerId));

      if (peer.streamId) {
        StreamService.deleteStreamById(peer.streamId);
      }
      PeerService.deletePeerById(peerId);
      yield put(actions.deletePeerById(peer.peerId));
    } catch (error) {
      socketEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

function disconnectSocket() {
  Api.disconnect();
}

function* sendMessage({ payload: message }: PayloadAction<string>) {
  try {
    Api.emit('sendMessage', message);
  } catch (error) {
    yield put(globalActions.addMessage({
      type: MessageType.Error,
      text: error.message,
    }));
  }
}

function* watchMessagesReceive() {
  const socketEventChannel: EventChannel<PeerInterface[]> = yield
call(createSocketEventChannel, 'sendMessage');

  while (true) {
    try {
      const message: Message = yield take(socketEventChannel);
      yield put(actions.addMessage(message));
    } catch (error) {
      socketEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
```

```
        text: error.message
      }));
    }
  }
}

function* toggleAudio() {
  const currentPeerStream: MediaStream = yield
select(selectCurrentPeerStream);
  const audioTrack = currentPeerStream.getAudioTracks()[0];

  audioTrack.enabled = !audioTrack.enabled;
  yield put(actions.setCurrentPeerMutedState(!audioTrack.enabled));

  PeerService.getPeers().forEach(peer => {
    peer.send(PeerDataTypes.TOGGLE_AUDIO);
  });
}

function* toggleCamera() {
  const currentPeerStream: MediaStream = yield
select(selectCurrentPeerStream);
  const videoTrack = currentPeerStream.getVideoTracks()[0];

  videoTrack.enabled = !videoTrack.enabled;
  yield put(actions.setCurrentPeerCameraState(videoTrack.enabled));

  PeerService.getPeers().forEach(peer => {
    peer.send(PeerDataTypes.TOGGLE_CAMERA);
  });
}

function* watchScreenSharingEnded(track: MediaStreamTrack) {
  const trackEventChannel: EventChannel<EventTarget> = yield
call(createEventListenerChannel, track, 'ended');

  while (true) {
    try {
      yield take(trackEventChannel);
      const currentPeerStream: MediaStream = yield
select(selectCurrentPeerStream);
      currentPeerStream.removeTrack(track);
      yield put(actions.setCurrentPeerSharingState(false));
      PeerService.getPeers().forEach(peer => {
        peer.removeTrack(track, currentPeerStream);
        peer.send(PeerDataTypes.TOGGLE_SCREEN_SHARE);
      });
    } catch (error) {
      trackEventChannel.close();
      yield put(globalActions.addMessage({
        type: MessageType.Error,
        text: error.message
      }));
    }
  }
}

function* shareScreen() {
  const currentPeerStream: MediaStream = yield
select(selectCurrentPeerStream);
  // @ts-ignore
  const stream = yield navigator.mediaDevices.getDisplayMedia();
  const screenSharingTrack = stream.getVideoTracks()[0];
  currentPeerStream.addTrack(screenSharingTrack);
```

```
  yield put(actions.setCurrentPeerSharingState(true));
  yield spawn(watchScreenSharingEnded, screenSharingTrack);

  PeerService.getPeers().forEach(peer => {
    peer.addTrack(screenSharingTrack, currentPeerStream);
    peer.send(PeerDataTypes.TOGGLE_SCREEN_SHARE);
  });
}

function* unshareScreen() {
  const currentPeerStream: MediaStream = yield
select(selectCurrentPeerStream);
  const screenSharingTrack: MediaStreamTrack =
currentPeerStream.getVideoTracks()[1];
  screenSharingTrack.stop();
  screenSharingTrack.dispatchEvent(new Event('ended'));
}

function* toggleScreenShare() {
  const isCurrentPeerSharingScreen: boolean = yield
select(selectCurrentPeerSharingState);
  if (isCurrentPeerSharingScreen) {
    yield call(unshareScreen);
  } else {
    yield call(shareScreen);
  }
}

export function* watchRoomInit() {
  yield takeLatest(sagaActions.initMeeting.toString(), initRoom);
}

export function* watchSocketDisconnectRequest() {
  yield takeLatest(sagaActions.disconnectSocket.toString(),
disconnectSocket);
}

export function* watchMessageSend() {
  yield takeLatest(sagaActions.sendMessage.toString(), sendMessage);
}

export function* watchAudioToggle() {
  yield takeLatest(sagaActions.toggleAudio.toString(), toggleAudio);
}

export function* watchCameraToggle() {
  yield takeLatest(sagaActions.toggleCamera.toString(), toggleCamera);
}

export function* watchScreenShareToggle() {
  yield takeLatest(sagaActions.toggleScreenShare.toString(),
toggleScreenShare);
}

export default [
  watchRoomInit(),
  watchSocketDisconnectRequest(),
  watchMessageSend(),
  watchAudioToggle(),
  watchCameraToggle(),
  watchScreenShareToggle(),
];
```