МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП’ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри

___________ Ігор Жуков
(підпис) (ПІБ)

« __ » _____________ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ “МАГІСТР”
ЗА СПЕЦІАЛЬНІСТЮ 123 «КОМП’ЮТЕРНА ІНЖЕНЕРІЯ»

Тема: “Комп'ютерна система для встановлення зв'язку між користувачами на платформі NodeJS”

Виконавець: студент групи КС-231М Бабій Максим Андрійович

Керівник: кандидат технічних наук, доцент Кудренко Станіслава Олексіївна

Нормоконтролер: _________________ Василь МАЛЯРЧУК

Засвідчую, що у магістерській роботі немає запозичень праць інших авторів без відповідних посилань

Студент ______________ Бабій М. А.

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук та технологій

Кафедра комп'ютерних систем та мереж

Спеціальність 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри КСМ

_______________ Ігор Жуков
(підпис) (ПІБ)

« ___» _____________ 2023 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Бабія Максима Андрійовича
(прізвище, ім’я, по батькові)

1. Тема роботи: “Комп'ютерна система для встановлення зв'язку між користувачами на платформі NodeJS” затверджена наказом ректора від «29» серпня 2023 року № 1521/ст.

2. Термін виконання проєкту (роботи): з 02.10.2023 до 31.12.2023

3. Вихідні дані до роботи: розробити комп'ютерну систему для встановлення зв'язку між користувачами на платформі NodeJS

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

1) Аналіз архітектури комп’ютерних систем для встановлення зв’язку

2) Методи встановлення взаємозв’язку

3) Описання розробленої комп'ютерної системи для встановлення зв'язку між користувачами на платформі NodeJS

5. Перелік обов’язкового графічного матеріалу:

Презентація *Power Point*

6. Календарний план-графік

№ п/п	Етапи виконання кваліфікаційної роботи	Термін виконання етапів	Примітка
1	Ознайомитись з постановкою задачі кваліфікаційної роботи	02.10.2023 - 03.10.2023	
2	Вивчити спеціальну літературу і технічну документацію	03.10.2023 - 05.10.2023	
3	Проаналізувати використані інструменти та їхні можливі замінники	06.10.2023 - 08.10.2023	
4	Написати розділ 1	09.10.2023 - 20.10.2023	
5	Проаналізувати та порівняти схожі альтернативні системи	25.10.2023 - 30.10.2023	
6	Написати розділ 2	31.10.2023 - 10.11.2023	
7	Розробити систему та підготувати базовий опис	11.11.2023 - 20.11.2023	
8	Написати розділ 3	21.11.2023 - 30.11.2023	
9	Оформити пояснювальну записку та пройти нормоконтроль	10.12.2023 - 22.12.2023	
10	Підготувати презентаційний матеріал та захистити роботу	23.12.2023 - 31.12.2023	

7. Дата видачі завдання «02» жовтня 2023 р.

Керівник кваліфікаційної роботи ______________________ Кудренко С. О.

(підпис)

Завдання прийняв до виконання ______________________ Бабій М. А.

(підпис студента)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи “Комп'ютерна система для встановлення зв'язку між користувачами на платформі NodeJS”: 93 с., 51 рис., 24 використаних джерел.

МЕРЕЖА, ІНТЕРНЕТ, ВЕБ ТЕХНОЛОГІЇ, МЕТОДИ ВЗАЄМОЗВ’ЯЗКУ ТА З’ЄДНАННЯ, З’ЄДНАННЯ У РЕАЛЬНОМУ ЧАСІ.

Об’єкт дослідження - процес встановлення зв’язку між користувачами на платформі NodeJS.

Предмет дослідження - комп'ютерна система для встановлення зв'язку між користувачами на платформі NodeJS.

Мета кваліфікаційної роботи - розробка системи встановлення зв’язку та розгляд основних методів для його встановлення

Прогнози припущення щодо розвитку об’єкта дослідження - створення робочого зразка комп’ютерної системи та її налаштування.

Результати кваліфікаційної роботи рекомендується використовувати при розробці нових програмних засобів, які можуть містити в собі функціонал зв’язку між користувачами.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

RTCP - Real-Time Transport Control Protocol

HTTP - HyperText Transfer Protocol

IPC - Inter-process communication

HTTPS - HyperText Transfer Protocol Secure

HTML - HyperText Markup Language

CSS - Cascading Style Sheets

JS - JavaScript

WS - WebSocket

WSS - WebSocket Secure

TLS - Transport Layer Security

TS - TypeScript

SQL - Structured Query Language

API - Application Programming Interface

RAM - Random Access Memory

AWS - Amazon Web Services

IT - Information Technologies

URL - Uniform Resource Locator

JSX - JavaScript XML

AJAX - Asynchronous JavaScript And XML

SPA - Single Page Application

REST - REpresentational State Transfer

DOM - Document Object Model

NPM - Node Package Manager

ООП - Об'єктно-орієнтоване програмування

ФП - Функціональне програмування

ФРП - Функціонально-реактивне програмування

ACID - Atomicity, Consistency, Isolation, and Durability

STUN - Session Traversal Utilities for NAT

TURN - Traversal Using Relays around NAT

IP - internet protocol

AMQP - Advanced Message Queuing Protocol

gRPC - gRPC Remote Procedure Calls

MVC - Model-View-Controller

DDoS - Distributed Denial-of-Service

SDP - Session Description Protocol

ICE - Information and Content Exchange

CRM - Customer Relationship Management

JWT - JSON web token

ВСТУП

Комп'ютерні системи стали невіддільною частиною повсякденного життя кожної людини на планеті. На зараз, практично будь-яку дію можливо виконати завдяки певній системі або застосунку. Декілька прикладів використання систем представлені нижче:

– Замовлення або купування продуктів, квитків, ліків, запчастин та іншого. Завдяки системам, люди мають можливість отримувати будь-який товар з доставленням до дверей оселі. Товари також можуть бути з іншого міста або навіть країни.

– Банківські операції. Валютні операції та взаємодія з банком стала надзвичайно простою. Перекази можуть бути виконані між рахунками різних банків та різних країн.

– Промисловість. У виробничому секторі комп'ютерні системи можуть бути використані для автоматизації процесів виробництва, контролю за якістю продукції та управління обладнанням.

– Навчання та наукові дослідження. У навчальних закладах і наукових установах комп'ютери використовуються для проведення досліджень, викладання, розв'язання математичних задач та інших завдань.

– Медицина. Комп'ютерні системи можуть допомагати в лікарській діагностиці, обробці медичної інформації, моделюванні та управлінні медичним обладнанням.

– Розваги. Комп'ютерні ігри, трансляція аудіо й відео контенту, мультимедійні розваги та багато чого іншого стало доступним завдяки комп’ютерним системам.

– Транспорт. У транспортному секторі комп'ютерні системи використовуються для навігації, управління транспортними засобами та розв'язання інших транспортних задач.

– Громадянська безпека та оборона. У сферах безпеки та оборони комп'ютерні системи використовуються для керування військовою технікою, обробки інформації та управління комунікаціями.

Комп’ютерні системи сильно проникли у життя людей, що послуговує збільшенням загальної продуктивності виконання будь-якої роботи. З розвитком мережі Інтернет, відкрилась ще одна сфера для використання комп’ютерних систем - дистанційне працевлаштування та комунікація. З’явились різні системи які відкривали людям шлях для розмови на невизначених відстанях.

Першим кроком до такої комунікації став процес еволюції звичайної пошти до електронної. Перші клієнти електронної пошти були досить простими порівняно з тим, які існують сьогодні. Вони виконували базовий функціонал, такий як:

– Відправлення та отримання текстових повідомлень.

– Зберігання повідомлень.

– Адресація повідомлень.

– Перегляд інформації про вхідну та вихідну пошту.

– Видалення повідомлень.

Цей невеликий функціонал революціонізував спосіб спілкування людей. Він відкрив можливість миттєвого доставлення повідомлення до людини у будь-якій точці світу.

З майбутнім розвитком технологій з’являлись все більш просунені методи зв’язку, які працювали краще, ефективніше та надавали позитивний досвід користувачу. На сьогодні, найпоширенішими методами зв’язку є застосунки для обміну текстовими повідомленнями у режимі реального часу та програми для відео/аудіо спілкування. Зазвичай, ці програми вирішують не тільки задачу зв’язку, а також задачі сфери, на яку цей застосунок орієнтований. Прикладами таких систем можуть стати:

– *Google Meet*. Ця система створена для проведення аудіо та відео зустрічей. Вона має такий базовий функціонал як: презентація екрана, вбудований чат. З додаткового функціонала, *Google Meet* має синхронізацію з *Google Calendar*, що дає можливість планувати майбутні зустрічі. Таким чином, Google Meet розв'язує проблему менеджменту всіх зустрічей та вільного часу.

– *Microsoft Teams*. *Teams* був створений для організування дистанційної роботи. Ця система, як і *Google Meet*, також має можливість проведення відео та аудіо дзвінків. Але у додаток до цього, Teams надає робочий простір, у якому можливо мати повноцінні приватні та групові чати, канали зв’язку, календар майбутніх зустрічей, чат-ботів для виконання тривіальних задач та багато іншого корисного функціоналу. Виходячи з цього, ця система розв’язує проблему існування повноцінного віртуального робочого простору, який буде зручним та забезпечувати швидку й продуктивну роботу користувачам.

– *Discord*. На початку свого шляху, *Discord* був націлений на людей, які грали у комп’ютерні ігри. Система пропонувала зручний інтерфейс та функції, які були б корисні для таких людей. Але з набиранням популярності, *Discord* змінив свій профіль і крім базового функціонала чатів та аудіо або відео дзвінків, почав пропонувати функції для більшої взаємодії різних користувачів. *Discord* ввів такі поняття як сервери та канали. Це групи, які орієнтовані на певний рід діяльності та об'єднують між собою різних людей. Родами діяльності можуть бути абсолютно все, починаючи з комп’ютерних ігор і закінчуючи складними науками, або вузькоспеціалізованими темами. Підсумовуючи, *Discord* намагається зробити зручний, ефективний інтерфейс та об’єднати мільйони людей.

– Електронна пошта. Як вже було сказано вище, цей метод являється прогресом звичайної пошти. Електронна пошта не має функціоналу аудіо й відео дзвінків та обміну повідомленнями у реальному часі. Але з розвитком технологій, цей метод також прогресував і, попри появу багатьох просунутих застосунків, залишається одним з основних методів офіційного спілкування. Користувачі мають можливість тонкого налаштування пошти, можуть отримувати маркетингові листи від різних компаній або магазинів і багато іншого.

Підбиваючи підсумки з вищесказаного, комп'ютерні системи для встановлення зв’язку між користувачами відкрили нові можливості для взаємодії людей з різних точок світу. Цими людьми можуть бути робочі команди якоїсь компанії, шкільний або студентський навчальний клас, або просто декілька незнайомців, яких об’єднує спільний інтерес до якоїсь теми. На зараз, такі системи є надзвичайно важливими, мають великий попит і розв’язують найрізноманітніші задачі.

Комп'ютерні системи для встановлення зв'язку є ключовим елементом сучасного інформаційного суспільства, надаючи надійний та швидкий обмін даними. Вони об'єднують у собі апаратне та програмне забезпечення для забезпечення ефективної передачі інформації через мережі, незалежно від їх розміщення чи фізичного розташування. Від традиційних локальних мереж до глобального Інтернету, ці системи стають дорогоцінним інструментом для спілкування, обміну даними та забезпечення неперервного доступу до інформації у високотехнологічному світі.

Метою цього проєкту є створення такої системи, яка буде мати базовий функціонал у вигляді можливості створення відеозустрічей та їхнім керуванням. Під час виконання роботи буде розібрано, який набір технологій було використано, а також проведено аналіз можливих альтернатив. У роботі будуть представлені альтернативні системи, які вже мають мільйони активних користувачів. Так само буде проведений аналіз архітектурних та технологічних рішень, які були запропоновані цими системами.

Сьогодні комп'ютерні системи для встановлення зв'язку стали невіддільною (частина/ознака) частиною практично усіх сфер життя, починаючи від корпоративного сектору і закінчуючи повсякденними комунікаційними потребами. Їх роль вирішальна в ефективному функціонуванні бізнесу, наукових досліджень, освіти, медицини та суспільства в цілому. Забезпечуючи швидку передачу інформації та сприяючи взаємодії між різними платформами, ці системи сприяють розвитку інновацій та підвищенню ефективності в усіх галузях. Таким чином, вивчення та вдосконалення комп'ютерних систем для встановлення зв'язку стає важливим завданням для подальшого розвитку інформаційного суспільства.

Головним завданням є побудова системи для комунікації між користувачами, де люди зможуть реєструвати облікові записи, створювати зустрічі та проводити їх у відеоформаті.

Система буде мати базовий функціонал, який буде слугувати основою застосунку. Подальший розвиток буде залежати від напрямку, де цей застосунок буде використовуватись. Новим функціоналом, який не залежить від сфери використання, може стати:

– Можливість створення приватних та групових чатів.

– Перегляд всіх зустрічей у вигляді календаря.

– Вбудований чат в середині зустрічі.

– Можливість презентації екрана.

РОЗДІЛ 1
ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ДЛЯ РЕАЛІЗАЦІЇ ПРОТОТИПУ СИСТЕМИ ДЛЯ КОМУНІКАЦІЇ

1.1. Комп’ютерна вебсистема для комунікації між користувачами

Комп'ютерні системи є складними агрегатами апаратних та програмних засобів, спроєктованих для вирішення конкретних завдань. Вони містять апаратне забезпечення (процесори, пам'ять, пристрої введення-виведення) та програмне забезпечення (операційні системи, додаткові програми). Комп'ютерні системи можуть бути вбудованими в пристрої чи існувати як окремі, автономні установки.

Комп’ютерні системи мають 3 основних елементи:

– Апаратне забезпечення. Таке забезпечення містить в собі всі фізичні компоненти комп’ютера. Це процесори, пам’ять, прилади введення-виведення, мережеві пристрої та інші.

– Програмне забезпечення. Програмне забезпечення є невіддільною частиною комп’ютерних систем. Операційні системи керують роботою апаратних засобів, а додаткові застосунки виконують певні користувацькі задачі.

– Мережева взаємодія. Комп’ютерні системи можуть взаємодіяти між собою через мережу. Мережа може бути локальною в офісі або глобальним Інтернетом. Це дозволяє проводити обмін даними.

Вебсистеми являють собою важливий аспект сучасних комп’ютерних систем. Завдяки цим системам, користувачі можуть використовувати певні функції через веббраузер, що робить взаємодію із системою більш доступною та універсальною. Під'єднатися до вебсистем можна через протокол *HTTP*, а взаємодіяти за допомогою багатьох різних [8].

Такі системи мають певні переваги перед класичними настільними програмами:

1. Вебзастосунок не потребує інсталяції. Все, що необхідно для роботи - це будь-який веббраузер та підключення до мережі Інтернет. Така робота забезпечує сумісність та легкість використання на великій кількості різноманітних пристроїв, незалежно від платформи.

2. Вебсистеми сумісні з більшістю браузерів і працюють узгоджено в усіх операційних системах, незалежно від оновлення чи версії. Це дає можливість використовувати застосунок на будь-якій платформі з будь-якого пристрою.

3. Менші витрати на розробку. Можливість запуску вебсистеми у будь-якому середовищі зменшує загальну вартість розробки. Зникає необхідність розробляти окремі настільні та мобільні застосунки для різних операційних систем.

Головною метою цього кваліфікаційного проєкту є розробка прототипу вебсистеми для ефективної комунікації між користувачами. Заснована на принципі забезпечення можливості створення та проведення відеозустрічей з обмеженим колом учасників, система розробляється з урахуванням сучасних тенденцій і вимог користувачів.

Розробка даної системи підкреслює важливість інновацій у сфері дистанційної комунікації. Прототип цієї системи є лише початком. Майбутній розвиток може визначити нові стандарти у галузі віртуальної комунікації.

Підбиваючи проміжні підсумки, комп'ютерна система для взаємозв'язку є надважливою сучасною темою, яка розв'язує проблему спілкування на великих дистанціях.

1.2. Інструменти для реалізації

Прототип системи цієї кваліфікаційної роботи буде працювати у вебсередовищі, тому базовими технологіями будуть *HTML5*, *CSS3* і *TypeScript* [9]. Оскільки це прототип і навантаження системи не очікується, буде використана монолітна архітектура. Такий вибір пришвидшить розробку і зменшить кількість роботи над конфігураційними файлами. Серверна частина буде використовувати *NodeJS* і *NestJS*, що дає можливість роботи парадигми “*JavaScript is Everywhere*”. *PostgreSQL* буде обраною базою даних. Також, для реалізації самих відео зустрічей, буде використано технологію *Vonage*, яка надає змогу створювати сесії та “*peer to peer*” зв'язок.

1.2.1. Системна архітектура

Монолітна архітектура (рис. 1.1) є традиційною уніфікованою моделлю для розробки програмного забезпечення. Монолітний, у цьому контексті, означає «скомбінований в одній частині». Відповідно до словників, прикметник монолітний означає «занадто великий» і «неможливий для змін» [19].

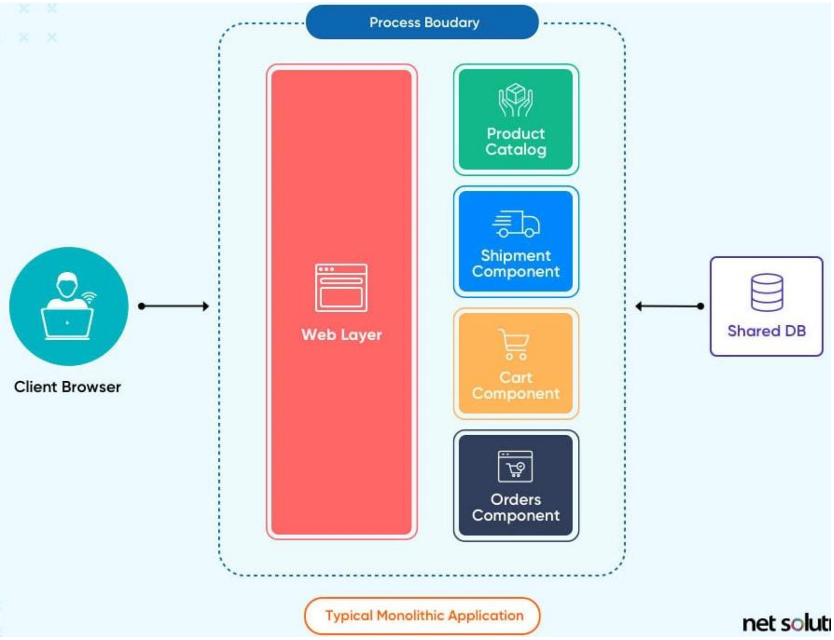


Рис. 1.1. Монолітна архітектура

Монолітне програмне забезпечення автономне. У ньому компоненти або функції програми тісно пов’язані. У монолітній архітектурі, кожен компонент і пов’язані з ним компоненти, повинні бути доступними для того, щоб виконання коду або запуск програмного забезпечення було можливим.

Монолітні застосунки є однорівневими, що означає, що кілька компонентів об'єднані в одному великому застосунку. Як наслідок, вони мають великі кодові бази, які з часом можуть бути надзвичайно важкими для управління.

Крім того, якщо один компонент програми потребує оновлення, інші елементи також можуть вимагати переписування, і вся програма повинна бути перекомпільована та протестована. Цей процес може забирати багато часу та обмежувати гнучкість та швидкість роботи команд розробників програмного забезпечення.

Попри ці проблеми, такий підхід все ще використовується, оскільки він має свої переваги. Крім того, існує багато старих програм, які були створені як монолітне програмне забезпечення і тому цей підхід не може бути повністю відкинутий, коли ці програми все ще використовуються.

На початку розробки, програми з монолітною архітектурою можуть мати кращий показник продуктивності, ніж модульні застосунки. Вони також можуть бути легше протестовані та налагоджені, оскільки в них менше компонентів, що спрощує ситуацію [19].

Використання однієї кодової бази полегшує журналювання, управління конфігурацією, моніторинг продуктивності програми та інші аспекти розробки. Розгортання також може бути спрощене шляхом копіювання готового додатку на сервер. Крім того, існує можливість розміщення кількох копій додатка за балансувальником навантаження для горизонтального масштабування.

Але слід зауважити, що монолітичний підхід зазвичай краще підходить для простих, невеликих застосунків. Для складніших застосунків, з частими змінами в коді або з розширенням потреб у масштабуванні, цей підхід може виявитися не підхожим. Для складніших задач, зазвичай, обирають мікросервісну архітектуру, але розробку самого прототипу пишуть з використанням монолітної архітектури, оскільки це заощаджує час.

1.2.2. *HTML5* і *CSS3*

HTML - це мова розмітки гіпертексту, яка в основному використовується для створення документів у мережі Інтернет. Історія *HTML* розпочалася в 1993 році та на той час це була примітивна мова для створення вебсторінок. Неможливо уявити сучасний веб без цієї мови. Поточною версією *HTML* є *HTML5*, яка була випущена у 2014 році. Протягом років *HTML* зазнав численних змін і отримав нові функції, але ключовий концепт залишився незмінним [21].

HTML-файли мають розширення *.html* або *.htm* і складаються з кількох основних компонентів: тегів, типів даних, посилань на символи та сутності. Структура *HTML*-файлу містить вкладені теги, кожен з яких має свої власні особливі атрибути. Нижче наведено приклад такої структури:

```
<!DOCTYPE html>
<html>
    <head>
        <title>This is a title</title>
```

```
        <meta charset="UTF-8">
    </head>
    <body>
        <div>HTML &#38 CSS</div>
        <h1>Hello World!</h1>
        <span><b>Bold Text</b></span>
        <h2>Heading 2</h2>
    </body>
</html>
```

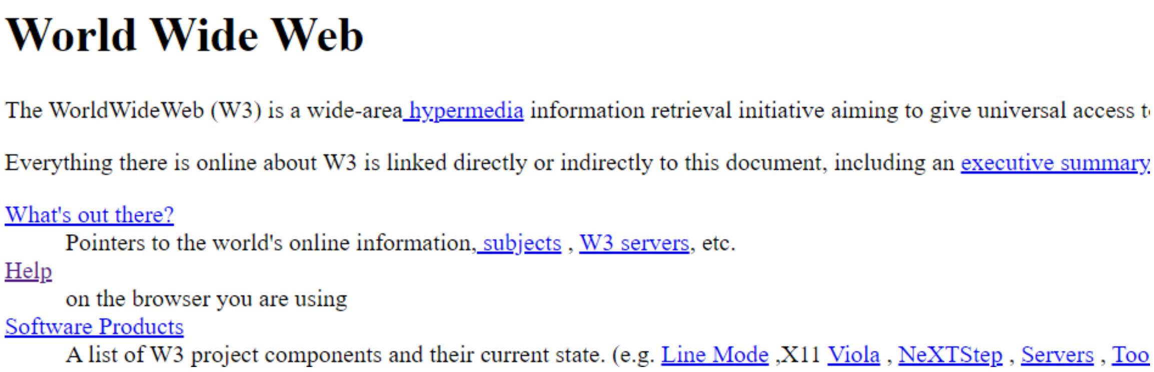
В цій структурі можна побачити декілька основних тегів, які мають бути у кожному *HTML* файлі. Починається файл з декларації версії *HTML*. У 5 версії це робиться таким чином:

```
<!DOCTYPE html>
```

Далі іде тег “*html*”. Цей тег є корінним і огортає всі інші теги. Наступними тегами є “*head*” та “*body*”. Елемент “*head*” є контейнером для метаданих. Метадані - це дані про документ і вони не відображаються. Зазвичай такі дані визначають назву документа, набір символів, стилі та іншу інформацію. Елемент “*body*” містить в собі зміст всього документу. Прикладом можуть стати параграфи, картинки, посилання та таблиці.

HTML - це потужна мова розмітки, яка має багато функцій і є простою для вивчення. Ця мова розмітки не є самостійною технологією і, зазвичай, використовується разом з *CSS*.

CSS означає *Cascading Style Sheets*, і його головною метою є опис і визначення вигляду всього документа чи окремих елементів. Перша версія *CSS* була випущена в 1996 році, і поточною версією є *CSS* 3. Без цієї мови веб був би колекцією сайтів з простим текстом [20]. Перша вебсторінка, яка використовувала лише *HTML*, показана на рис. 1.2.



World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access t

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Too

Рис. 1.2. Перша вебсторінка

Важко щось було б зрозуміти, якщо сторінка мала б лише текст та гіперпосилання. Для розв'язання цієї проблеми було винайдено *CSS*. За допомогою цієї мови можна змінити та стилізувати будь-що на вебсайті.

Структура *CSS*-файлу набагато простіша, ніж *HTML*-файлу. Ця мова працює на основі правил і має два основні концепти: селектори та властивості. Приклад коду *CSS* наведений нижче:

```
.container {
  display: flex;
  align-items: center;
  justify-content: center;
  background-color: aqua;
}
```

HTML і *CSS* є двома з трьох базових вебтехнологій. Без цих технологій неможливо уявити сучасний веб.

1.2.3. *React* і *Redux*

HTML та *CSS* є важливими вебтехнологіями, але писати велику програму або систему, використовуючи лише ці їх - довго і незручно. Для розробки щось більш масштабного, розробники використовують глобальні бібліотеки або фреймворки.

React був створений *Facebook* і перша його версія була випущена 29 травня 2013 року. Його ключовими функціями є *JSX* (*JavaScript XML*), компонента архітектура, віртуальний *DOM* та багато іншого [9].

JSX - це розширення для мови програмування *JavaScript*, яке робить код схожим на *HTML*. Завдяки *JSX*, розробники можуть використовувати елементи мови програмування всередині *HTML* розмітки. Такими елементами можуть стати цикли, змінні, властивості об’єктів чи масивів, тернарні оператори та інше. Приклад коду *JSX* зображений на рис. 1.3.

```
<div>
  <h1>Title goes here</h1>

  {rooms.length > 0 ? (
    rooms.map((room) => <Room key={room.id} room={room} />)
  ) : (
    <Box
      display="flex"
      justifyContent="center"
      alignItems="center"
      sx={{ mt: 4 }}
    >
      <Typography variant="h6">No meetings</Typography>
    </Box>
  )}
</div>;
```

Рис. 1.3. Розширення *JSX*

У прикладі (див. рис. 1.3) можна побачити використання властивостей й методів масиву, а також тернарних операторів. Такий підхід максимально спрощує розробку і робить код зрозумілим та читабельним.

Компонента архітектура надає можливість значного перевикористання коду. Це оптимізує процес розробки та прибирає проблему повторюваності. Компоненти можуть бути гнучкими та приймати різні параметри, як і звичайні функції будь-якої мови програмування. Завдяки цим параметрам можна змінювати логіку та поведінку компонента, а якщо до цього додати можливості *JSX*, то з’явиться можливість зміни зовнішнього вигляду.

Також компоненти *React* мають функцію стану. Стан компонента може зберігати будь-які дані. Якщо стан буде змінений, компонент буде відмальований з новими даними. Таким чином, користувач буде бачити миттєві зміни без повного перезавантаження сторінки.

Класичний *React* компонент зображений на рис. 1.4.

```
export const Room: React.FC<CardProps> = (props) => {
  const navigate = useNavigate();

  const handleJoinClick = () => {
    navigate(`/${props.room.id}`);
  };

  return (
    <Card sx={{ m: 2 }} variant="outlined">
      <CardContent>
        <Typography variant="h6">{props.room.title}</Typography>
        <Typography variant="body2" color="grey">
          {props.room.description}
        </Typography>
      </CardContent>
      <CardActions sx={{ p: 2 }}>
        <Button onClick={handleJoinClick} variant="outlined">
          Join
        </Button>
      </CardActions>
    </Card>
  );
};
```

Рис. 1.4. *React* компонент

Віртуальний *DOM* це технічне рішення розробників *React*, яке направлено на оптимізацію роботи з *DOM* деревом веббраузера. Основна ідея рішення - це створення віртуальної копії реального *DOM* дерева у пам’яті. Ця копія оновлюється набагато швидше ніж реальне дерево. Після змін, віртуальне дерево порівнюється з реальним і рушій перемальовує тільки ті частини, які дійсно змінились. Це робить *React* дуже швидким.

Концепція стану компонентів в *React* є ключовою. Ця функція є зручною і гнучкою, але вона має певні проблеми, які з’являються зі збільшенням масштабу системи. Чим більше стає різних компонентів, тим більше з’являється локальних станів, які стає все важче контролювати. Такий підхід може викликати багато неочікуваних проблем, які буде важко знайти та налагодити. Для розв'язання цієї проблеми, з’явились такі бібліотеки як *Redux*. Головна задача таких бібліотек - це менеджмент стану системи. *Redux* є однією з найпопулярніших бібліотек, оскільки вона була однією з перших.

Redux пропонує наступну реалізацію. Ціла система буде мати один великий стан, який буде називатись store. Він може бути розбитий на певні групи, на кшталт “*users*”, “*rooms*” та інше. Цей стан не може бути змінений ззовні та не може бути модифікованим. Стан керується за допомогою спеціальних функцій і тільки ними. Зазвичай, ці функції виносяться в окремі файли.

Таким чином, всі локальні стани компонентів стають непотрібними. Вся інформація в них береться з єдиного, головного стану. В компонентах можна залишити тільки суто компонентні речі, як, наприклад керування виглядом або індикатором завантажування даних. Хоча деякі розробники, переносять в стан *Redux* такі елементи також. Глобальний стан *Redux* зображений на рис. 1.5.

```
import user from "./user/reducer";
import room from "./room/reducer";

const store = configureStore({
  reducer: {
    user,
    room,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: false,
    }),
});

export type RootState = ReturnType<typeof store.getState>;
```

Рис. 1.5. Головний стан *Redux*

1.2.4. *Node.js* і *NestJS*

Node.js і *Nest.JS* є 2 головними технологіями серверної частини проєкту. *Node.js* - це відкрите середовище виконання *JavaScript*, яке працює на різних операційних системах та платформах. Перша версія *Node* була випущена 27 травня 2009 року. *NestJS* - це фреймворк для створення серверних застосунків на базі *Node.js*, який запроваджує свій погляд на те, як має бути побудований сучасний проєкт. *Nest* надає багато вбудованих функцій, які виконують тривіальні задачі.

Вибір *Node.js* як серверної платформи, дозволяє використовувати парадигму "*JavaScript is Everywhere*". Це значно спрощує розробку, оскільки весь проєкт використовує одну мову програмування - *JavaScript* [16].

Node.js використовує рушій *JavaScript V8*, який був розроблений *Google* для веббраузера *Google Chrome*. Ця технологія має певні властивості:

– Асинхронна однониткова модель виконання запитів.

– Система модулів *CommonJS*.

– Неблокуючий ввід/вивід.

NodeJS має певні вбудовані модулі, які допомагають у написанні серверних застосунків. Наприклад:

– Модуль “*http*”. Цей модуль має певні функції та методи, які допомагають використовувати *NodeJS* як *HTTP* сервер.

– Модуль “*fs*”. В цьому модулі находяться методи взаємодії з файловою системою.

– Модуль “*os*”. Цей модуль містить функції для роботи з операційною системою.

Для побудови простого сервера на *NodeJS* без використання зовнішніх бібліотек, необхідно використовувати вбудований http модуль. В цьому модулі присутні всі необхідні інструменти для створення сервера, обробки запитів, надсилання відповідей, роботи з файлами та інше [17].

Приклад простого *HTTP* серверу зображений на рис. 1.6.

```
const http = require("http");

const host = 'localhost';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, host, () => {
  console.log(`Server is running on http://${host}:${port}`);
});
```

Рис. 1.6. Простий сервер на платформі *Node*

Проаналізувавши предметну область можна виділити наступні переваги платформи *Node*:

1. Висока ефективність. Однією з найважливіших характеристик *Node.js* є можливість створення застосунків, які оперують швидко і видають результат виконання за лічені секунди.

2. Велика підтримка спільноти. Мільйони розробників активно роблять свій внесок у спільноту *Node.js*. Існує багато форумів і сайтів, де можна знайти рішення навіть самої незвичайної проблеми, або знайти людину, яка може допомогти з її вирішенням.

Більш того, існує велика кількість бібліотек і фреймворків, які можливо встановити за декілька секунд і які можуть спростити розв'язання типових задач. Ці бібліотеки знаходяться у *NPM*. *NPM* (*Node Package Manager*) - це один з найбільших реєстрів пакетів у світі. Він містить численні інструменти, які можна використовувати безпосередньо у вашому проєкті.

3. Масштабованість. *Node* надає можливість масштабувати застосунок як у горизонтальному, так і вертикальному напрямках. Горизонтальне розширення відбувається шляхом додавання вузлів до наявної системи, а вертикальне - шляхом додавання ресурсів до окремих вузлів.

Середовище Node також має певні недоліки, наприклад:

1. Нестабільний *API*. *API* може часто змінюватись і може бути доволі ненадійним. Також, нові функції можуть виходити з несумісними змінами. Це може викликати неочікувані помилки, які можуть бути важкими для налагоджування.

2. Модель асинхронного програмування.

NestJS використовує прогресивний *JavaScript*, який дає змогу використовувати останні нові та сучасні методи мови програмування. Він побудований з використанням *TypeScript* і повністю його підтримує (проте все ще дозволяє розробникам програмувати на чистому *JavaScript*). *NestJS* поєднує елементи ООП (об'єктно-орієнтованого програмування), ФП (функціонального програмування) і ФРП (функціонального реактивного програмування). *NestJS* побудований з використанням надійних *HTTP*-серверних фреймворків, таких як *Express*.

NestJS має свій власний підхід для побудови серверного застосунку. Застосунок поділяється на окремі модулі. Кожен модуль відповідає за певну логіку і складається з таких головних компонентів:

– Модуль “*controller*”. Даний модуль відповідає за збереження логіки *REST* методів.

– Модуль “*dto*”. Цей модуль зберігає об’єктні описи для всіх перевірок даних.

– Модуль “*service*”. Даний модуль зберігає головну логіку, яка буде виконуватися всередині контролерів.

– Модуль “*repository*”. Даний модуль відповідає за збереження конфігурацій, які пов’язані з базою даних та сутностями.

– Модуль “*entity*”. Даний модуль відповідає за збереження схеми сутності у базі даних.

Приклад типового модулю *Nest* зображений на рис. 1.7.

```
@Module({
  imports: [
    ConfigModule.forRoot({
      load: [configuration],
      isGlobal: true,
    }),
    TypeOrmModule.forRoot({
      ...configuration().database,
    } as TypeOrmModuleOptions),
    JwtModule.register({
      global: true,
      secret: configuration().jwt.secret,
      signOptions: { expiresIn: '30d' },
    }),
    UserModule,
    AuthModule,
    RoomModule,
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Рис. 1.7. Типовий модуль *Nest*

NestJS має багато модулів, які можуть розв'язувати тривіальні задачі замість розробника. *Nest* може бути легко налаштований для використання не тільки *HTTP*, а і *GraphQL* або *WebSocket*. Також він має чудові бібліотеки для налаштування валідації даних, які роблять її читабельною та зрозумілою. *Nest* надає можливість легкого налаштування захищених шляхів за допомогою *Guards*. В цілому, існує дуже багато високорівневого *API*, які максимально спрощують розробку. При цьому, він надає зручні можливості налаштування власноруч, якщо запроваджена базова логіка не влаштовує.

1.2.5. *PostgreSQL*

PostgreSQL є потужною, відкритою об'єктно-реляційною системою бази даних, яка використовує та розширює мову *SQL* численними корисними функціями для надійного зберігання інформації та масштабування найскладніших завдань обробки даних [10].

PostgreSQL завоювала велику репутацію завдяки доведеній архітектурі, надійності, цілісності даних, потужному набору функцій, розширюваності та відданості спільноти відкритого програмного забезпечення, яка постійно забезпечує продуктивні та інноваційні рішення. *PostgreSQL* підтримує всі основні операційні системи, вона відповідає вимогам *ACID* (вірогідність, консистентність, ізольованість та стійкість) з 2001 року і має потужні доповнення, такі як популярний розширювач геопросторової бази даних *PostGIS* [10].

Інтеграція *PostgreSQL* з *Nest* буде виконана за допомогою бібліотеки *Typeorm*. Ця бібліотека спрощує роботу з базою даних шляхом надавання простого *API* для побудування схем даних та їх перевірок. Також *Typeorm* надає зручні методи для створення запитів до бази даних. Ці методи можуть бути як простими функціями *GET*, так і складними агрегаціями.

Приклад побудованої схеми таблиці за допомогою *Typeorm* зображений на рис. 1.8.

```
import { Column, Entity, ManyToMany, OneToMany } from 'typeorm';

import { BaseEntity } from 'src/common/base.entity';
import { Room } from '../room/room.entity';

@Entity()
export class User extends BaseEntity {
  @Column({ nullable: false })
  name: string;

  @Column({ nullable: false })
  username: string;

  @Column({ nullable: false })
  password: string;

  @OneToMany(() => Room, (room) => room.author)
  createdRooms: Room[];

  @ManyToMany(() => Room, (room) => room.participants)
  rooms: Room[];
}
```

Рис. 1.8. Схема побудована із допомогою *Typeorm*

Нижче наведені ключові функції бібліотеки Typeorm та їхній короткий опис:

1. Схеми. Схеми таблиць пишуться мовою *JavaScript* о простій і зрозумілій формі - об’єкт з ключами та значеннями.

2. Високорівневі запити. *Typeorm* надає можливість використовувати простий і читабельний *API* для виконання запитів до бази.

3. Спрощена система реляційних зв’язків. Реляційні зв’язки між таблицями робляться за допомогою трьох простих декораторів.

4. Проста інтеграція з такими бібліотеками як NestJS.

5. Детальна конфігурація та інструменти відладки. Typeorm надає зручні інструменти відладки, які можуть бути додатково налаштовані.

1.2.6. *Vonage Video API*

Vonage - один з найпопулярніших постачальників різноманітних функцій для побудування комунікаційних систем. За допомогою *Vonage* можливо побудувати як простий чат-бот підтримки, так й складні системи зв’язку.

Для побудови свого *API, Vonage* використовує базові протоколи зв’язку, які необхідні для будь-якого базового зв’язку.

HTTP (*Hypertext Transfer Protocol*) є фундаментальним протоколом всесвітньої мережі та був розроблений для отримання ресурсів, таких як документи *HTML*. Це протокол рівня застосунків, який використовує модель клієнт-сервер. Модель клієнт-сервер означає, що отримувач інформації, зазвичай веббраузер, ініціює запити. *HTTP* дозволяє завантажувати не лише документи *HTML*, але й файли *JavaScript, CSS,* зображення та відео. *HTTP* - протокол, який дозволяє спілкування між сервером і клієнтом.

Приклад повідомлення *HTTP*:

GET / HTTP/1.1

Host: google.com

Accept-Language: en-US

Існує важлива версія *HTTP*, якою є *HTTPS* (*Hypertext Transfer Protocol Secured*). Шифрування *HTTPS* використовується для захисту даних, які надсилаються користувачем або сервером. Безпечна версія протоколу *HTTP* використовується майже всюди, оскільки це важливий стандарт і надає певну впевненість у ресурсі, яким користується користувач.

WebSocket, або *WS* - протокол, який надає можливість встановлення двостороннього зв'язку між сервером та клієнтом. Він належить до рівня застосунків моделі *OSI*. Протокол *WebSocket* є постійним, що означає, що після встановлення з'єднання, воно не буде закрите, поки сервер або клієнт його не закриють самостійно. Цей протокол також називають протоколом повного дуплекса. Повний дуплекс означає, що обидві системи (клієнт і сервер у цьому випадку) можуть спілкуватися одночасно. Протокол *WebSocket* може бути використаний в застосунках, де необхідно оновлювати дані в реальному часі. Наприклад, чати, ігри або будь-який вебзастосунок, який потребує постійного потоку даних.

Протокол *WebSocket* працює за допомогою *HTTP*. Для встановлення *WebSocket* з'єднання, необхідно здійснити спеціальний *HTTP*-запит на сервер. Цей запит називається запитом на вдосконалення підключення. Такий запит виглядає наступним чином:

GET /index.html HTTP/1.1

Host: www.example.com

Connection: upgrade

Upgrade: example/1, foo/2

Якщо сервер схвалює цей запит, він відправить клієнту статус 201, і *WebScoket* з'єднання буде встановлене.

WebScoket, як і *HTTP*, має окрему, захищену версію - *WSS*, або *WebScoket Secured*. Протокол *WSS* використовує *TLS*-з'єднання для встановлення каналу, тоді як звичайний *WebSocket* використовує незашифроване з'єднання.

RTP, або *Real-time Transport Protocol* - це мережевий протокол для передачі аудіо і відео через *IP* мережі. *RTP* використовується в комунікаційних і розважальних системах, які включають потокове медіа, наприклад телефонію, застосунки для відеотелеконференцій, телевізійні служби та вебфункції *push-to-talk*.

RTP, зазвичай, працює через *UDP*, разом з протоколом *RTCP*. Головними задачами *RTCP* є моніторинг статистики передачі даних і якості сервісу та допомога у синхронізації декількох потоків даних. *RTP* може часто використовуватись у поєднанні з протоколом сигналізації, таким як *SIP*, який встановлює з’єднання у мережі.

STUN (*Session Traversal Utilities for NAT*) та *TURN* (*Traversal Using Relays around NAT*) сервери грають важливу роль в обміні повідомленнями в реальному часі (відео, аудіо, спільне використання екрана та іншими даними). Вони використовуються в процесі сигналізації, але з різних причин.

Перш ніж встановлювати *peer-to-peer* з’єднання, важливо, щоб сервери ідентифікували *IP*-адресу для кожного учасника. Застосунки *WebRTC* використовують сервери *STUN* більшу частину часу. Вони прості, швидкі, а головне не створюють великого навантаження.

Щобільше, вони застосовуються лише під час налаштування з'єднання для виявлення та обміну зовнішніми парами хост:порт. Після встановлення сесії, обмін медіафайлами буде відбуватись безпосередньо між учасниками. Однак (одначе), *STUN* іноді обмежений брандмауером. Таким чином, розробникам потрібно зробити поворот (*turn*) і використовувати сервер *TURN*, коли *STUN* виходить з ладу.

Брандмауери та закриті корпоративні середовища можуть обмежувати доступ до *IP*-адреси. Без унікальних ідентифікаторів двом мережам стає неможливо знайти одна одну. Цю проблему можна вирішити, використовуючи сервери *TURN* як посередника. На відміну від *STUN*, сервер *TURN* залишається в медіашляху після встановлення з'єднання.

Однак використання цього методу, може додати деяку затримку вашій відеоконференції. Крім того, це також може збільшити експлуатаційні витрати вашої IT-інфраструктури. Що більше трафіку ви спрямовуєте на свій сервер *TURN*, то складніша інфраструктура вам знадобиться.

Vonage API спрощує взаємодію з усіма протоколами. Сервіс пропонує різні бібліотеки для популярних мов програмування з високорівневими функціями. Розробка застосунку стає комфортною і простішою.

1.3. Альтернативні інструменти

У наш час, існує безліч технологій, які можна використати для виконання однієї й тої самої задачі. Команди розробників можуть використати різні мови програмування, бібліотеки та фреймворки та мати однаковий результат. При виборі технічного стека необхідно враховувати багато змінних.

Наприклад: у якому середовищі система або застосунок буде працювати; яке можливе навантаження; які можливі порушення безпеки та інше. Від цих змін залежить вибір абсолютного всіх компонентів - починаючи з маленьких бібліотек і закінчуючи архітектурою, мовою програмування, базами даних і фреймворків.

1.3.1. Можливі заміни монолітній архітектурі

Як було сказано вище, в цьому проєкті використовується монолітна архітектура. У цього типа архітектури є дві основні альтернативи - безсерверні обчислення і мікросервіси.

Архітектура мікросервісів — це архітектура, де вся система розподілена на модулі, кожен з яких виконує своє завдання. Ці модулі спілкуються між собою за допомогою легких механізмів як то *HTTP*, *gRPC*, *AMQP*. Вони розгортаються незалежно один від одного. Такі сервіси можуть використовувати абсолютно різні технології й бути написаними різними мовами програмування. Все залежить від задач, які цей модуль має виконувати [24].

Оскільки складові сервіси невеликі, вони можуть створюватися однією або декількома невеликими командами з самого початку, розділених межами сервісів, що полегшує розширення зусиль розробки, якщо це необхідно.

Після розробки ці сервіси також можна розгортати незалежно один від одного, і, отже, легко визначити найзавантаженіші служби та масштабувати їх незалежно від цілої системи. Мікросервіси також пропонують покращену ізоляцію помилок, завдяки чому, в разі помилки в одній службі, не обов’язково припиняється функціонування всієї програми. Коли помилку виправлено, її можна буде розгорнути лише для відповідної служби замість повторного розгортання всієї програми.

Ще одна перевага, яку надає архітектура мікросервісів, полягає в тому, що полегшується вибір технологічного стека (мови програмування, бази даних тощо), який найкраще підходить для необхідної функціональності (сервісу), замість того, щоб обирати більш стандартизований та універсальний підхід. Розробники також можуть обирати різних хмарних постачальників [24].

Також, мікросервісна архітектура робить простішою інтеграцію нового коду в різнорідні та застарілі системи. Є звіти про досвід кількох компаній, які успішно замінили частини свого чинного програмного забезпечення мікросервісами або перебувають у процесі цього. Процес модернізації програмного забезпечення застарілих програм виконується з використанням поступового підходу, що забезпечує роботу системи без критичних перепадів.

Схема мікросервісної архітектури зображена на рис. 1.9.

Рис. 1.9. Архітектура мікросервісів

З переваг мікросервісної архітектури можна виділити наступне:

1. Масштабованість. Систему, яка побудована на мікросервісній архітектурі, легко масштабувати, тому що для додавання нової функції розробнику достатньо додати новий модуль. Нема потреби змінювати всю систему.

2. Надійність. Якщо один модуль вийшов з ладу, це не означає, що вся система стане недоступною.

3. Багаторазове використання та простота. Сервіси невеликі за розміром, тому їх легко читати. Щобільше, кожен з них відповідаю за певну задачу і це робить їх придатними для повторного використання в інших системах.

Недоліками мікросервісної архітектури є:

1. Складність. Усе в архітектурі мікросервісів стає важче розробляти та підтримувати.

2. Збільшена затримка. Для отримання результатів сервіси мають комунікувати один з одним і це збільшує час, який необхідний для видачі фінального результату користувачу.

Безсерверні обчислення – це архітектура, де вся система складається з численних функцій і розгортається в хмарному середовищі. Задача постачальника - надавати простір для розгортання коду, займатись підтримкою стабільності системи та надавати зручне і гнучке налаштування системи. Задача розробників - писати код у різних функціях та розгортати їх. Це максимально спрощує розробку серверної частини застосунку через “відсутність” сервера. Вся серверна робота виконується хмарою, а всі налаштування можуть бути зроблені через *UI* інтерфейс.

За допомогою безсерверної архітектури можна запускати всю систему одразу або лише її частину. Щойно код програми запускається, сервер виділяє йому ресурси та звільняє їх, коли програма більше не використовується.

Власник програми стягує плату лише протягом часу, коли програма використовується. Компанії з хмарних послуг надають *Backend-as-a-Service* (*BaaS*) і *Function-as-a-Service* (*FaaS*).

Існує 6 основних компонентів безсерверної архітектури:

1. Функція як послуга (*FaaS*). *FaaS* є основоположним будівельним блоком безсерверної системи, який відповідає за виконання логіки, яка визначає, як розподіляються ресурси в певному сценарії. Залежно від хмарного середовища, яке використовується, ви можете вибрати спеціальну пропозицію *FaaS*, як-от *AWS Lambda* для *Amazon Web Services* (*AWS*), *Microsoft Azure Functions* для *Azure*, *Google Cloud Functions* для *Google Cloud Platform* (*GCP*) і *IBM Cloud Functions* для приватних або гібридних середовищ. Ці функції читатимуть вашу серверну базу даних, коли користувач ініціює подію, оброблять результат та надішлють відповідь.

2. Клієнтський інтерфейс. Клієнтський інтерфейс відіграє важливу роль у безсерверній функціональності. Розробники не можуть примусово вставити безсерверну архітектуру в будь-яку програму, яку ви виберете. Інтерфейс має бути здатний підтримувати короткі стрибки запитів, взаємодії без стану та гнучку інтеграцію. Інтерфейс також має бути розроблений таким чином, щоб він був сумісний із передачею даних надзвичайно великого чи малого обсягу.

3. Хмарний вебсервер. Вебсервер – це місце, де буде ініційовано взаємодію без стану після того, як її ініціює користувач і до того, як служба *FaaS* завершить її. Вебсервер відрізняється від серверної бази даних, де зберігається інформація, що надається користувачам. Наприклад, припустімо, що ви є постачальником онлайн-відеовмісту. У цьому випадку на вебсервері розміщуються запити користувачів, сценарії та відповіді *FaaS*, перш ніж він буде вимкнений відповідно до природи

безсерверного режиму. З іншого боку, відеоконтент зберігатиметься у внутрішньому сховищі, очікуючи на запит користувача.

4. Служба безпеки. Безпека є ключовим елементом безсерверних операцій, оскільки:

– Програма обробляє тисячі одночасних запитів. Перед надсиланням відповіді кожен запит має бути автентифікований.

– Через свою природу без стану історія минулих взаємодій не зберігається. Програма не може повернутися до попередніх взаємодій, щоб перевірити майбутні.

– Безсерверна модель ускладнює прозорість і моніторинг. Ви повинні отримувати дані безпеки з мільйонів подій, які реєструються щодня.

– Розподілений характер безсерверної архітектури означає, що залучено кілька служб і постачальників.

Як правило, безсерверні програми використовують службу маркерів, де для користувачів генеруються тимчасові облікові дані, які можна використовувати для виклику функції. Також можливо інтегрувати безсерверні служби ідентифікації та керування доступом у систему. Наприклад, *AWS Cognito* працює з *AWS Lambda* для автентифікації користувача через *SSO* або соціальні мережі. Такі послуги є різними для кожного хмарного постачальника. Також, якісь певні функції можуть бути відсутніми в інших.

5. База даних сервера. Внутрішня база даних – це місце, де зберігається інформація, якою потрібно поділитися з користувачем. Це може варіюватися від репозиторіїв статичного вмісту до баз даних *SQL*, від зберігання медіа до режимів прямого мовлення. Зазвичай розробники використовують рішення *Backend* як послуга (*BaaS*), щоб ще більше скоротити зусилля з обслуговування та адміністратора. Крім того, більшість хмарних постачальників надають рішення *BaaS*, сумісні з їх пропозицією *FaaS*.

6. Шлюз *API*. Шлюз *API* з'єднує компоненти 1 і 2, тобто *FaaS* і клієнтський інтерфейс. Коли користувач ініціює дію, вона ретранслюється через шлюз *API*, щоб ініціювати подію через службу *FaaS*. Шлюз може підключати клієнтський інтерфейс до кількох сервісів *FaaS* і розширювати функціональні можливості програми.

Схема безсерверної архітектури зображена на рис. 1.10.

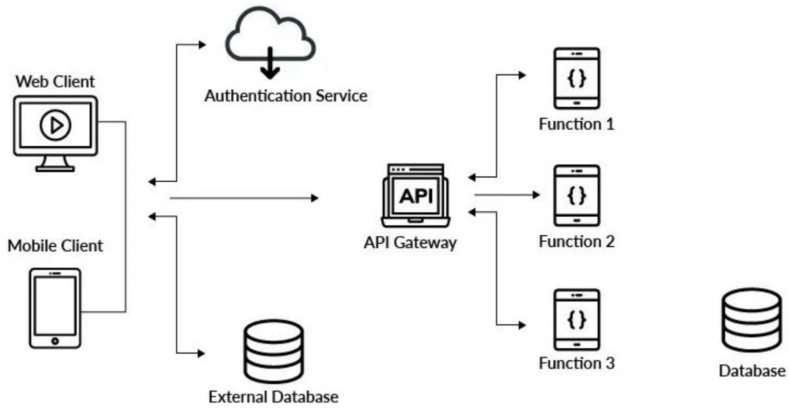


Рис. 1.10. Безсерверна архітектура

З переваг безсерверної архітектури можливо виділити наступні пункти:

1. Легке розгортання. Для розгортання нових функцій чи оновлення старих, розробник має написати один скрипт, в якому будуть необхідні налаштування для хмари та назви функцій, які будуть оновлені. Далі вся робота буде зроблена хмарою.

2. Покращена масштабованість. У безсерверній архітектурі масштабованість майже автоматична. Хмарне середовище автоматично регулює ресурси, які використовуються розгорнутими функціями.

Безсерверна архітектура має наступні недоліки:

1. Прив'язка до постачальника. При використанні якогось хмарного постачальника - команда автоматично блокує собі стек можливих технологій. Якщо команда вирішить використовувати *Firebase*, то вона зможе використовувати тільки технології, які пропонує *Firebase*. Використовувати різних хмарних постачальників разом важко, тому що вони не заточені один під одного і мають свої власні напрацювання для саме своїх платформ.

2. Холодний старт. Якщо певна функція не викликалася протягом тривалого часу, вона вивантажується хмарним середовищем і наступний виклик цієї функції буде повільним.

Підводячи висновки, архітектура мікросервісів є хорошим вибором для систем, які вже зрілі та мають постійне навантаження. Ця архітектура потребує більше ресурсів для підтримки, але вона забезпечує велику кількість переваг, таких як надійність і масштабованість. Безсерверні обчислення можуть бути корисною концепцією для систем, де навантаження не дуже велике. Крім того, ця архітектура є гарними вибором для проєктів, які не планують змінювати стек технологій у майбутньому

1.3.2. Альтернативи *Vonage*

Існує багато сервісів, які надають зручний *API* для побудови складних систем. Головною альтернативою *Vonage* є *WebRTC*.

WebRTC підтримує передачу відео, голосу та загальних даних між однoранговими вузлами, що дозволяє розробникам створювати потужні рішення для голосового та відеозв’язку [11].

Технологія доступна в усіх сучасних браузерах, а також у нативних клієнтах для всіх основних платформ.

Технології, що лежать в основі *WebRTC*, реалізовані як відкритий вебстандарт і доступні як звичайний *API JavaScript* у всіх основних браузерах. Для нативних клієнтів, таких як застосунки для систем *Android* та *iOS*, доступна бібліотека, яка надає ті самі функції.

Проєкт *WebRTC* є проєктом з відкритим кодом і підтримується *Apple*, *Google*, *Microsoft* і *Mozilla*.

Приклад використання *WebRTC API* зображений на рисунку 1.11.

```
// Listen for local ICE candidates on the local RTCPeerConnection
peerConnection.addEventListener('icecandidate', event => {
  if (event.candidate) {
      signalingChannel.send({'new-ice-candidate': event.candidate});
  }
});

// Listen for remote ICE candidates and add them to the local RTCPeerConnection
signalingChannel.addEventListener('message', async message => {
  if (message.iceCandidate) {
      try {
          await peerConnection.addIceCandidate(message.iceCandidate);
      } catch (e) {
          console.error('Error adding received ice candidate', e);
      }
  }
});
```

Рис. 1.11. Приклад використання *WebRTC API*

WebRTC API є більш низькорівневим, ніж *Vonage API*, що трохи ускладнює розробку, але надає більш гнучкі можливості. У випадку прототипу цієї кваліфікаційної роботи, не було необхідності у більшій гнучкості, тому обраним був саме *Vonage*.

1.3.3. Клієнтські технології та зовнішній вигляд

Для розробки клієнтської частини існує безліч різних бібліотек і фреймворків. Є технології, які повноцінно змінюють підхід до побудови застосунку, а є такі, які являються доповненням до попередніх. Прикладом технологій, які повноцінно змінюють підхід до розробки, можуть бути *React*, *Angular*, *Vue*. Ця трійця є найпопулярнішими варіантами на момент написання. Для спрощення розробки дизайну складових частин застосунку, існують бібліотеки, які містять збірку готових компонентів, які використовують певні стандарти. Наприклад, *MUI*, *Ant Design*, *Bootstrap*.

Angular - це фреймворк, який має багато функцій, які працюють прямо з коробки. Він побудований на *TypeScript*. Перша версія *Angular* була випущена 14 вересня 2016 року. Розробкою цього фреймворку займається компанія Google. Він використовується для побудови односторінкових застосунків (SPA) та надає розширені засоби для розробки вебсистем.

Як платформа, фреймворк *Angular* містить в собі наступний функціонал:

– Заснований на компонентах фреймворк для створення масштабованих вебзастосунків.

– Набір добре інтегрованих бібліотек, що охоплюють широкий спектр функцій: маршрутизація, керування формами, клієнт-серверна взаємодія тощо.

– Набір інструментів розробника, які допоможуть вам розробляти, будувати, тестувати й оновлювати код.

При роботі з *Angular*, розробники використовують переваги платформи, яка може масштабуватись від малих застосунків до систем корпоративного рівня. *Angular* розроблений таким чином, щоб процес оновлення був максимально простим і у команди розробників були можливості використовувати останні інновації у своїй роботі.

На відміну від *React, Angular* має багато вбудованого функціонала, який не потребує ніяких додаткових інсталяцій і працює одразу після створення проєкту. Деякий вбудований функціонал фреймворку Angular зображений на рис. 1.12.

Рис. 1.12. Екосистема *Angular*

Кожна іконка (див. рис. 1.12) представляє певний набір інструментів, які вбудовані всередину Angular. Деякі з цих вбудованих функцій описані нижче:

– *CLI*. На думку великих підприємств, *CLI* є однією з найкращих функій, якими оснащено *Angular*. Цей інструмент дозволяє створювати компоненти, служби та модулі за допомогою простої команди. Також *CLI* дозволяє автоматично перетворити застосунок на прогресивний або оновити його до останньої стабільної версії.

– *Schematics*. Це генератор коду, який дозволяє створювати та перетворювати код. Наприклад, замість того, щоб знову і знову створювати моделі *TypeScript*, можливо створити схему, яка генерує увесь клас із засобами доступу та іншим. Також його можна використати як інструмент для оптимізації великої програми шляхом використання зовнішніх шаблонів замість вбудованих.

– *Angular Material*. Бібліотека компонентів, розроблена *Google*, яка надає розробникам доступ до набору повністю протестованих компонентів, які працюють із коробки. За замовчуванням, компоненти тематично відповідають системі дизайну *Angular Material*, але є *CDK* (комплект розробки компонентів), що надає можливість використовувати будь-яку систему дизайну.

– *Http Module*. Виклики *HTTP* присутні в більшості систем. Саме тому *Angular* поставляється з модулем, який полегшує роботу з ними. Він заснований на *Observables*, тому *HTTP* виклики можна зробити будь-яким способом. Крім того, він має *HTTP*-перехоплювачі, які дозволяють легко трансформувати вхідні/вихідні запити.

– *i18n*. *Angular* надає вбудовані функції, для допомоги у створенні багатомовних вебсистем. *Angular* читає шаблони та створює файли *XLIFF*, які можна використовувати для перекладу текстів, чисел, дат і валют. Коли розробники налаштують переклади, *CLI* створить відповідні збірки без будь-яких додаткових зусиль.

Компоненти *Angular* дуже схожі на *React* компоненти, але використовують класи як основний метод побудови. Класичний компонент *Angular* зображений на рис. 1.13.

```
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent  {
  public counter: number = 0;

  constructor() { }

  public increment() {
    this.counter++;
    return this.counter;
  }

  public decrement() {
    this.counter--;
    return this.counter;
  }

  public reset() {
    this.counter = 0;
    return this.counter;
  }
}
```

Рис. 1.13. *Angular* компонент

Angular є гарним вибором для застосунків та систем, які мають велику кількість функцій вже на самому початку свого шляху. Розробникам не доведеться шукати та підбирати бібліотеки для виконання певних функцій, тому що *Angular* має рішення для більшості тривіальних речей. Але через цю велику кількість речей, *Angular* являється важчим у вивченні.

Vue досить схожий на *Angular* і *React*, тому що його розробники надихались саме цими технологіями. *Vue* так само пропонує компонентну основу, можливість написання *SPA* систем, підтримку стану компонент та багато інших функцій. За замовчуванням, *Vue* немає великої кількості вбудованих бібліотек. Бібліотека також має концепцію віртуального *DOM* дерева.

Vue доволі простий у вивчанні. Все, що необхідно для розуміння його роботи - базове знання *HTML, CSS* і *JavaScript*. Компоненти *Vue* пишуться в одному файлі, що робить їх легкими для розуміння і читання. Також *Vue* надає зручні інструменти розробника.

Проаналізувавши предметну область, можна виділити наступні переваги бібліотеки *Vue*:

– Спрощений процес розробки. *VueJS* пропонує простий та інтуїтивно зрозумілий синтаксис, що полегшує вивчення та написання коду. Наприклад, створення компонента *Vue* передбачає визначення шаблону, даних і методів в одному файлі, що спрощує процес розробки. Його компонентна архітектура сприяє повторному використанню коду, що призводить до швидшого циклу розробки. Наприклад, розробники можуть створити багаторазовий компонент, як-от спадне меню, і легко використовувати його в різних частинах програми.

– Відмінна продуктивність *VueJS* використовує віртуальну *DOM* і ефективні алгоритми візуалізації, забезпечуючи продуктивність, що вражає. Оновлюються лише необхідні компоненти, зменшуючи непотрібне повторне перемалювання та підвищуючи загальну швидкість. Його легка природа та оптимізовані механізми оновлення забезпечують швидший час завантаження та відтворення. Оптимізація продуктивності *Vue* забезпечує безперебійну та швидку роботу користувача навіть зі складними програмами, які потребують великих даних.

– Універсальність і гнучкість. *VueJS* — це адаптивний фреймворк, який можна легко інтегрувати в наявні проєкти. Якщо ви хочете додати *Vue* до невеликої частини своєї вебсистеми чи створити повноцінну, він може легко адаптується до потреб команди. Фреймворк може створювати прогресивні вебзастосунки (*PWA*), односторінкові програми (*SPA*) або навіть гібридні мобільні програми. Гнучкість *Vue* дозволяє вибрати правильний підхід на основі вимог вашого проєкту та масштабувати його за потреби.

– Комплексна екосистема. *VueJS* має квітучу екосистему з багатьма плагінами, бібліотеками та інструментами. Наприклад, *Vuex* надає рішення для керування станом, *Vue Router* пропонує потужну систему маршрутизації, а *Vuetify* надає багатий набір компонентів інтерфейсу користувача. Екосистема надає рішення для різних функцій, таких як маршрутизація, керування станом і компоненти інтерфейсу користувача. Ці ресурси підвищують продуктивність і дозволяють розробникам використовувати перевірені та оптимізовані рішення для типових випадків використання.

– Надійна документація та навчальні ресурси. *VueJS* має вичерпну та добре організовану документацію, що полегшує розробникам початок роботи. Офіційна документація *Vue* охоплює всі аспекти розробки на платформі *Vue*, від основ до складних концепцій. Офіційний посібник, посилання на *API* та навчальні статті пропонують покрокові інструкції та приклади, які допоможуть розробникам зрозуміти концепції та найкращі практики *Vue*. Наявність зрозумілої та доступної документації прискорює процес навчання.

– Читабельний код і одно файлові компоненти. *VueJS* сприяє написанню чистого та читабельного коду, покращуючи співпрацю та зручність обслуговування. Його декларативний синтаксис і проста структура полегшують розробникам програмного забезпечення розуміння та підтримку кодової бази. Його одно файлові компоненти інкапсулюють *HTML*, *CSS* і *JavaScript*, створюючи цілісну модульну структуру коду. Наприклад, одно файловий компонент може містити шаблон, сценарій і стилі, пов’язані з певним компонентом, забезпечуючи кращу організацію та розподіл завдань.

– Невеликий розмір програми. Головною причиною вибору *Vue* є його розмір, а саме 18–21 Кб. Бібліотека пропонує високу швидкість, попри свій невеликий розмір. Загалом, *Vue* заохочує розробників обирати його як для великих, так і для невеликих прикладних систем.

– Зручні умовні позначення. Написання шаблонного коду може зайняти багато часу. Фреймворк ухиляється від цього трудомісткого завдання, пропонуючи вбудовані рішення та підтримку створення компонентів, керування станом і анімацією.

Класичний компонент *Vue* зображений на рис. 1.14.

```
<script setup>
import { ref } from 'vue'
const count = ref(0)
</script>

<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>

<style scoped>
button {
  font-weight: bold;
}
</style>
```

Рис. 1.14. Одно файловий компонент *Vue*

Ураховуючи всі плюси та мінуси цих трьох технологій, для проєкту був обраний *React*, через його простоту, легкість і велику підтримку. У випадку, якби проєкт був більшим за масштабом, можливо було б обрати *Angular* або *Vue*, як основний інструмент для розробки клієнтської частини.

Ant Design, *MUI*, *Bootstrap* - це бібліотеки, які надають базові компоненти, з готовими стилями та певною логікою. Кожна бібліотека має свої стандарти стилю і свою логіку.

Ці бібліотеки дуже зручні, тому що розробникам не треба витрачати час на розробку базових компонентів. Але така зручність створює і певні проблеми, у випадку якщо стиль або логіка готового компоненту має бути змінена. Такі зміни можуть бути неможливими або доволі складними в імплементації.

Bootstrap є однією з найпопулярніших бібліотек для розробки користувацьких інтерфейсів вебсистем. Ця бібліотека була розроблена компанією *Twitter* і перша її версія була випущена 19 серпня 2011 року. *Bootstrap* пропонує прості компоненти, які зроблені виключно за допомогою *HTML*, *CSS* та *JavaScript*. Компоненти *Bootstrap* створені для початку швидкої та ефективної розробки. Також *Bootstrap* пропонує зручну систему сіток для організації структури сторінок та розміщення елементів [15]. Типові компоненти, які пропонуються бібліотекою, зображені на рис. 1.15.

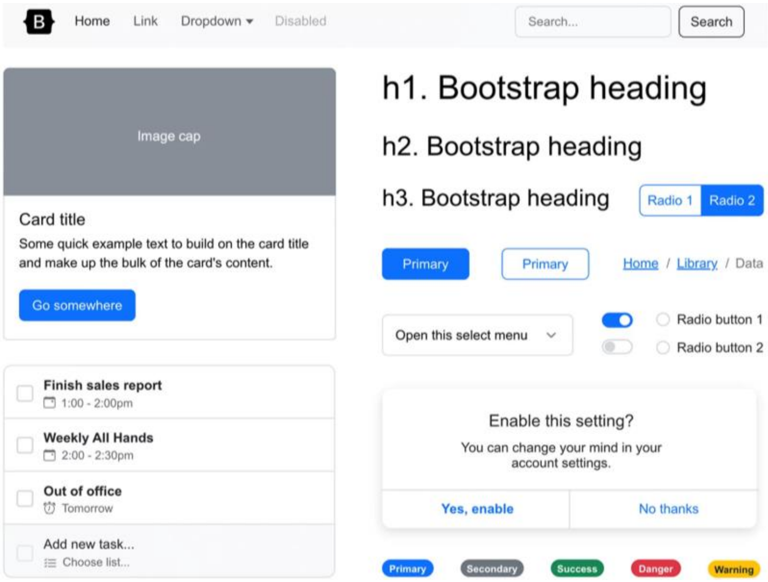


Рис. 1.15. Компоненти *Bootstrap*

MUI - бібліотека *React*, яка надає компоненти, засновані на концепціях дизайну *Google Material*. Перша версія цієї бібліотеки була випущена у 2014 році і на сьогодні, вона отримала 5 глобальних оновлень. *MUI* вирізняється чистим й консистентним дизайном та легкістю інтеграції в проєкти *React*. Основними її характеристиками є:

– Підтримка темизації для швидкої зміни кольорів та стилів у всій системі, а також окремо для локальних компонентів.

– Розширена можливість налаштування логіки компонентів.

– Компоненти, які засновані на дизайні *Google Material*.

– Підтримка адаптивного дизайну. Компоненти *Material-UI* підтримують адаптивний дизайн, що дозволяє їм гармонійно виглядати на різних пристроях та екранах.

– Інструменти для тестування та відладки. *Material-UI* надає інструменти для ефективного тестування та зневадження, що полегшує процес розробки та підтримки.

Саме бібліотека *MUI* була використана для побудови системи цього кваліфікаційного проєкту. Такий вибір був зроблений, оскільки клієнтська частина роботи використовує React та через легкість імплементування додаткових налаштувань. Приклад деяких готових компонентів бібліотеки *MUI* можна побачити на рис. 1.16.

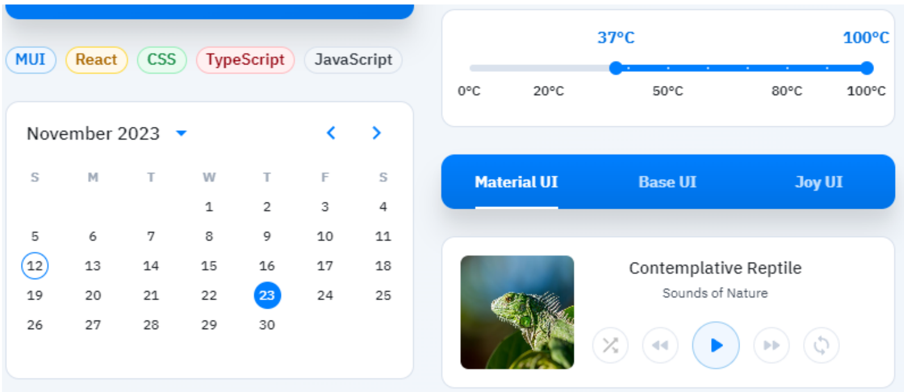


Рис. 1.16. *MUI* компоненти

Ant Design - це комплексна бібліотека дизайну, яка була розроблена компанією *Alibaba*. Головна відмінність *Ant Design* від *MUI* - це зовнішній вигляд. *Ant Design* так само пропонує готові універсальні компоненти, які мають багато корисного вбудованого функціонала. Приклад компонентів цієї бібліотеки зображений на рис. 1.17.

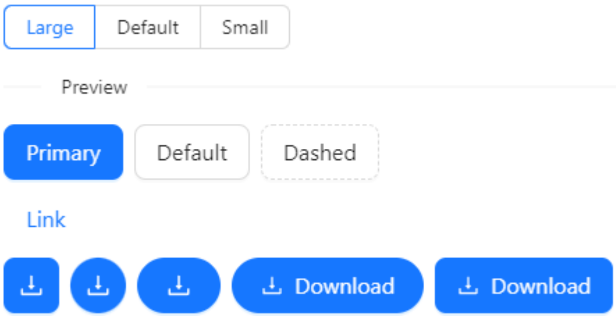


Рис. 1.17. Кнопки *Ant Design*

На сьогодні, такі бібліотеки є одними з ключових частин великих проєктів, оскільки економлять багато часу та надають зручні та перевірені компоненти.

Висновки за розділом

Комп'ютерні системи є складними агрегатами апаратних та програмних засобів, спроєктованих для вирішення конкретних завдань. Вони містять апаратне забезпечення (процесори, пам'ять, пристрої введення-виведення) та програмне забезпечення (операційні системи, додаткові програми). Комп'ютерні системи можуть бути вбудованими в пристрої чи існувати як окремі, автономні установки.

Вебсистеми являють собою важливий аспект сучасних комп’ютерних систем. Завдяки цим системам, користувачі можуть використовувати певні функції через веббраузер, що робить взаємодію із системою більш доступною та універсальною. У цій частині роботи були визначено, яким чином такі системи працюють і які вони мають переваги та недоліки.

Перший розділ роботи, містить в собі детальний опис технологій, які були обрані для розробки цієї системи. Були описані переваги та недоліки цих технологій. Отже, основний стек технологій можливо описати наступним чином:

– Клієнтська частина: *React* та *Redux*.

– Серверна частина: *NodeJS* та *NextJS*.

– Мова програмування: *TypeScript*.

– Допоміжний *API*: *Vonage TokBox*.

– База даних: *PostgreSQL*.

Ці технології були обрані внаслідок їхньої одночасної ефективності та гнучкості. Вибір платформи *NodeJS* для розробки серверної частини системи дозволив використовувати парадигму “*JavaScript is Everywhere*”, що значно спростило та прискорило розробку.

Також, у цьому розділі була приділена увага можливим альтернативам деяких технологій, а саме наданий невеликий загальний опис, наведені базові приклади використання та описані плюси й мінуси у порівнянні з обраними технологіями.

Головними альтернативами *React* є *Angular* та *Vue*. Ці бібліотеки пропонують схожі концепти, але мають іншу реалізацію та інші підходи. *Angular* має велику кількість вбудованого функціонала, який не потребує додаткових інсталяцій та налаштувань. *Vue* має зручні одно файлові компоненти та є максимально оптимізованим.

Монолітну архітектуру застосунку можливо замінити мікросервісною. Мікросервісна архітектура надає багато переваг у вигляді стабільності системи та гнучкості у виборі технологій для розробки певного функціонала. Але також, ця архітектура ускладнює процес розробки, що не підходить на самому початку проєкту. Зазвичай, перші версії застосунків будують саме з використанням монолітної архітектури, оскільки це спрощує розробку, але у майбутньому, ця архітектура є рекомендованою.

WebRTC може стати заміною *Vonage API* та надати більше гнучких налаштувань та можливостей, трохи ускладнивши розробку. *WebRTC* надає змогу використовувати низькорівневий *API*, що надає можливість тонкого налаштування.

В цілому, було проведено базове ознайомлення з проєктом, його технологіями та альтернативами. Це надає ширший погляд на підхід до розробки системи і дає розуміння, чому вибір початкового технологічного стека є надзвичайно важливим першим кроком.

РОЗДІЛ 2
ПОРІВНЯННЯ З ІНШИМИ СИСТЕМАМИ

2.1. *Discord*

Discord - це програма для спілкування між мільйонами користувачів по всьому світу. Перша версія *Discord* була випущена 13 травня 2015 року і була розрахована на гравців у комп’ютерні ігри. З часом, застосунок ставав все більш популярним і розробники почали робити зміни у сторону більш універсального клієнта, а ніж спеціалізованого для геймерів. Зовнішній вигляд настільного застосунку в операційній системі *Windows* зображений на рис. 2.1.

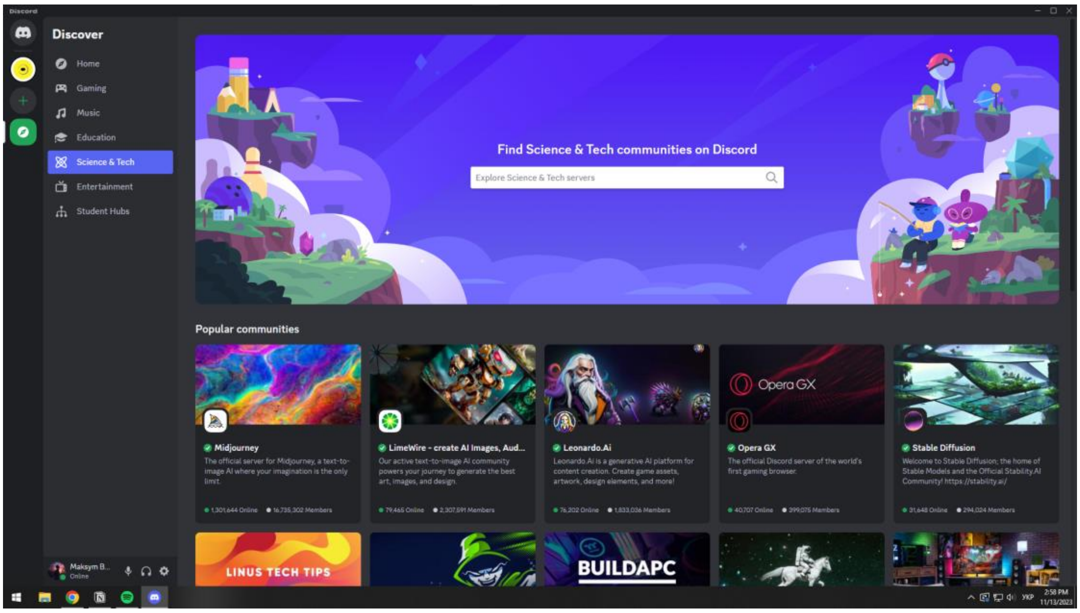


Рис. 2.1. *Discord* у системі *Windows* 10

З головних функцій *Discord* можна виділити:

– Миттєвий обмін повідомленнями та медіа.

– Канали, групи та приватні чати.

– *VoIP* функції. *Discord* надає можливість створювати голосові та відеодзвінки між необмеженою кількістю людей.

– Сервери. Завдяки серверам, люди можуть створювати тематичні місця, де користувачі зі схожими інтересами зможуть знайомитись та чимось ділитись.

На момент написання, *Discord* має приблизно 150 мільйонів активних користувачів кожний місяць і продовжує активно розвиватись. Застосунок підтримує всі сучасні платформи - *macOS*, *Linux*, *iOS*, *Android*, *Windows* і браузер.

Для написання клієнтської частини, розробники Discord використовують мови програмування *JavaScript* і *TypeScript*. Головною бібліотекою для мобільного застосунку є *React Native*, який дозволяє писати універсальні програми для *iOS* і *Android* одночасно. Головним фреймворком для настільних застосунків являється *Electron*, за допомогою якого також можливо будувати універсальні застосунки. Веббраузер використовує бібліотеку *React*. Таким чином, розробники *Discord* використовують єдину мову програмування для всіх своїх клієнтських частин. Такий підхід зменшує вартість розробки та спрощує розробку, роблячи її більш зрозумілою і швидшою. *Discord* не використовує парадигму “*JavaScript is everywhere*”, тому на серверній частині можна побачити мову програмування *Elixir* разом з фреймворком *Phoenix*. Мова програмування *Elixir* є функціональною і відома своєю масштабованістю та відмовостійкістю. Фреймворк *Phoenix* був розроблений для створення масштабованих програм у режимі реального часу. Для зберігання даних, *Discord* використовує базу даних *PostgreSQL*. Застосунок розгорнутий у таких хмарних середовищах як *Google Cloud Platform* і *AWS*. Також *Discord* використовує *webRTC* для побудови функцій аудіо і відеодзвінків [5].

З головних відмінностей технологічного стека *Discord* і мого прототипу можна відмітити серверні технології й архітектуру застосунку.

2.1.1. *Elixir* і *Phoenix* як серверне рішення

Elixir — це функціональна мова паралельного програмування, яка побудована на віртуальній машині *Erlang* (*BEAM*). Підкреслюючи принципи функціонального програмування, вона забезпечує легкі процеси для ефективного паралелізму та слідує моделі актора для передачі повідомлень. Мова *Elixir* була спроєктована для побудови надійних, відмовостійких систем з фокусом на підтримку та розширюваність. Ключовими функціями цієї мови програмування є парадигми функціонального програмування, спрощений паралелізм та відмовостійкість, успадковану від *Erlang OTP* (*Open Telecom Platform*) [22].

Після вивчення документації мови програмування *Elixir* можна виділити наступні плюси:

– Паралельність. *Elixir* має легкі процеси, які забезпечують ефективне паралельне програмування.

– Відмовостійкість. Мова програмування *Elixir* успадковує надійні механізми відмовостійкості *Erlang*.

– Масштабованість. *Elixir* є гарним вибором для програм із горизонтальним масштабуванням.

Мінуси використання мови програмування *Elixir*:

– Поріг входу. Для адаптації та вивчання цієї мови, необхідне знання функціонального програмування.

– Розмір екосистеми.

Phoenix — це фреймворк веброзробки для *Elixir*, який брав натхнення з фреймворку *Rails*. *Phoenix* дотримується шаблону *MVC*, наголошуючи на швидкості, надійності та масштабованості. Фреймворк особливо підходить для застосунків які працюють у реальному часі.

Ключовими функціями фреймворку є висока продуктивність, функціональність у реальному часі через канали та масштабованість для застосунків із високим трафіком [23].

Використовуючи фреймворк *Phoenix* можливо отримати наступні плюси:

– Продуктивність. *Phoenix* дотримується певних конвенцій та умов задля продуктивності розробників. Ці конвенції схожі на *Ruby on Rails*.

– Можливості роботи у реальному часі. Фреймворк має підтримку вбудованих функцій і механізмів для роботи у реальному часі, що спрощує розробку.

– Масштабованість. *Phoenix* легко масштабується горизонтально завдяки базовій моделі паралелізму *Elixir*.

З мінусів використання фреймворку *Phoenix* можна виділити:

– Поріг входу. Для роботи з цим фреймворком необхідне гарне знання мови програмування *Elixir* та певних концепцій, які можуть використовуватись всередині самого фреймворку.

– Розмір спільноти. *Phoenix* має доволі активну спільноту розробників, але ця вона може бути набагато меншою, ніж спільноти інших популярних і усталених фреймворків.

Приклад використання мови програмування *Elixir* і фреймворку *Phoenix* зображений на рис. 2.2.

```
defmodule RealWorldWeb do
  def controller do
    quote do
      use Phoenix.Controller, namespace: RealWorldWeb
      import Plug.Conn
      import RealWorldWeb.Router.Helpers
      import RealWorldWeb.Gettext
    end
  end

  def view do
    quote do
      use Phoenix.View,
        root: "lib/real_world_web/templates",
        namespace: RealWorldWeb

      import Phoenix.Controller, only: [get_flash: 2, view_module: 1]

      import RealWorldWeb.Router.Helpers
      import RealWorldWeb.ErrorHelpers
      import RealWorldWeb.Gettext
    end
  end
```

Рис. 2.2. Мова програмування *Elixir* та фреймворк *Phoenix*

Підводячи висновки, *Elixir* і *Phoenix* забезпечують потужне середовище для розробки масштабованих, стійких до збоїв вебдодатків із можливостями працювати у режимі реального часу. Поріг входу у технології, невелика спільнота й екосистема є можливими проблемами на розгляд для розробників, які хочуть використовувати дані технології.

2.1.2. Методи роботи з мільйонами активних користувачів

Кожне аудіо/відео спілкування в *Discord* є багатостороннім. Підтримка великих групових каналів вимагає мережевої архітектури клієнт-сервер, оскільки *peer-to-peer* мережа стає непомірно дорогою зі збільшенням кількості учасників.

Маршрутизація всього мережевого трафіку через сервери *Discord* також гарантує, що *IP*-адреса користувачів ніколи не буде розкрита, не дозволяючи будь-кому її дізнатися та розпочати *DDoS*-атаку. Маршрутизація аудіо/відео через медіасервери також пропонує інші переваги, такі як модерація. Наприклад, адміністратори можуть вимкнути аудіо/відео для учасників-порушників [4].

Як було сказано вище, *Discord* працює на великій кількості платформ. Єдиний спосіб, яким розробники можуть підтримувати всі платформи - повторно використовувати код та можливості *WebRTC*. *WebRTC* доступний у всіх сучасних веббраузерах, а також він доступний як бібліотека для нативних програм. Браузерна версія застосунку покладається саме на браузерну імплементацію *WebRTC*, але нативні версії програми використовують єдиний медіа рушій С++, який створений на основі нативної бібліотеки *WebRTC*. Цей рушій більше орієнтований на вимоги системи та її користувачів. Це означає, що певні функції будуть працювати набагато краще саме у нативних застосунках, а ніж у браузерній версії. Прикладом таких функцій можуть стати:

– Можливість обходу автоматичного пригнічення звуку у *Windows*. *Windows* автоматично зменшує гучність всіх застосунків, коли система використовує будь-який пристрій для зв'язку. Це дуже незручно, якщо користувач паралельно зі дзвінком виконує якусь роботу, де йому потрібен якісний звук.

– Можливість окремого керування гучності. *Discord* має можливість зміни гучності кожного користувача окремо, що забирає необхідність зміни гучності системи.

– Можливість доступу до необробленого аудіо. Це допомагає виявити голосову активність та передавати декілька різних потоків звуку.

– Можливість зменшення пропускної здатності та споживання ресурсів процесора під час періодів тиші.

– Можливість імплементації загальносистемної функції "*push to talk*".

– Можливість відправлення додаткової інформації разом з аудіо чи відеопакетами. Наприклад, дані про пріоритетного мовця.

– Можливість обходу автоматичного приглушення пристрою зв'язку за замовчуванням у *Windows*. Приглушення, або зменшення гучності, означає, що Windows автоматично зменшує гучність усіх програм, коли використовується комунікаційний пристрій.

У *Discord*, голосовий або відеозв'язок ініціюється входом до голосового каналу або початком виклику. Це означає, що зв'язок завжди розпочинає клієнт, що зменшує складність як клієнтської, так і серверної частини застосунку, а також підвищує стійкість до неочікуваних помилок. У разі збою інфраструктури, користувачі завжди можуть ініціювати повторне з'єднання до нового серверу.

Оскільки розробники *Discord* мають повний контроль над своїм рушієм, вони роблять певні речі не такими методами як у браузерній версії. По-перше, *WebRTC* покладається на протокол *SDP* для узгодження аудіо або відеоінформація між учасниками [4]. Ці дані можуть мати розмір близько десяти кілобайтів в обидві сторони. Рушій *Discord* дозволяє використовувати низькорівневий *WebRTC API* для створення як потоку надсилання даних, так і потоку їх отримання.

Такий підхід робить можливим зменшення кількості переданої інформації при ініціалізації зв'язку. Розробники *Discord* передають мінімум даних, в які входять: адреса та порт голосового сервера, ключі та метод шифрування, кодек, ідентифікатор потоку. Вся ця інформація виходить близько 1000 байтів.

Код створення такого аудіо потоку зображений на рис. 2.3.

```
webrtc::AudioSendStream* createAudioSendStream(
  uint32_t ssrc,
  uint8_t payloadType,
  webrtc::Transport* transport,
  rtc::scoped_refptr<webrtc::AudioEncoderFactory> audioEncoderFactory,
  webrtc::Call* call)
{
    webrtc::AudioSendStream::Config config{transport};
    config.rtp.ssrc = ssrc;
    config.rtp.extensions = {{"urn:ietf:params:rtp-hdrext:ssrc-audio-level", 1}};
    config.encoder_factory = audioEncoderFactory;
    const webrtc::SdpAudioFormat kOpusFormat = {"opus", 48000, 2};
    config.send_codec_spec =
      webrtc::AudioSendStream::Config::SendCodecSpec(payloadType, kOpusFormat);
    webrtc::AudioSendStream* audioStream = call->CreateAudioSendStream(config);
    audioStream->Start();
    return audioStream;
}
```

Рис. 2.3. Код створення аудіопотоку

WebRTC використовує *ICE* для визначення найкращого шляху зв'язку між учасниками. Оскільки кожен клієнт підключається до спеціального сервера, розробники *Discord* не використовують цю функцію на своїх нативних клієнтах. Це дозволяє забезпечувати надійніше з'єднання і тримати *IP* адресу користувача прихованою від сторонніх у каналі. Також, розробники *Discord* не використовують протокол *SRTP*, який використовує *WebRTC* для шифрування медіаресурсів. Їхній рушій дозволяє впровадити свій транспортний прошарок. Завдяки цьому, *Discord* може використати швидшу систему шифрування *Salsa*. У додаток до цього, не відбувається надсилання аудіоінформації під час тиші, що результує у значному збереженні пропускної здатності та ресурсів центрального процесора. Проте, не варто забувати, що браузерна версія використовує всі ці технології та однією з головних задач серверної частини *Discord* є подолання відмінностей браузерної та нативної імплементації.

Discord має декілька серверних сервісів, які роблять можливим голосовий зв'язок. У цій роботі будуть описані три з них, а саме - *Discord Gateway*, *Discord Guilds* та *Discord Voice*.

Всі сигнальні сервери написані мовою програмування Elixir. Клієнт підтримує *WebSocket* з'єднання з сервером *Discord Gateway* і саме через нього отримує події пов'язані з каналами, повідомленнями, сповіщеннями та іншим. Коли користувач підключений до голосового каналу, статус зв'язку представлений у вигляді об'єкта. Цей об'єкт зображений на рис 2.4.

```
defmodule VoiceStates.VoiceState do
  @type t :: %{
          session_id: String.t(),
          user_id: Number.t(),
          channel_id: Number.t() | nil,
          token: String.t() | nil,
          mute: boolean,
          deaf: boolean,
          self_mute: boolean,
          self_deaf: boolean,
          self_video: boolean,
          suppress: boolean
        }

  defstruct session_id: nil,
            user_id: nil,
            token: nil,
            channel_id: nil,
            mute: false,
            deaf: false,
            self_mute: false,
            self_deaf: false,
            self_video: false,
            suppress: false
end
```

Рис. 2.4. Об'єкт статусу зв'язку

Коли користувач доєднується до голосового каналу, він призначається до сервера *Discord Voice*. Задача цього серверу - передавати аудіо кожного користувача у канал. Усі голосові канали в гільдії призначені одному серверу *Discord Voice*. Якщо ви перший голосовий учасник гільдії, сервер *Discord Guilds* відповідає за призначення сервера *Discord Voice* гільдії.

Discord Gateway, Discord Guilds і *Discord Voice* є горизонтально масштабованими. *Discord Gateway* і *Discord Guilds* розгорнуті на *Google Cloud Platform*.

Discord використовує понад 850 голосових серверів у 13 регіонах по всьому світу. Така кількість сутностей потребує надмірної обробки всіх помилок, невиконань та можливих атак. Завдяки всім підходам і ідеям, які використали розробники, цей застосунок здатен обслуговувати більше ніж 2.6 мільйонів одночасних голосових дзвінків із вихідним трафіком понад 220 Гбіт/с (біт за секунду) і 120 Мпікс/с (пакетів за секунду).

2.2. *Microsoft Teams*

Microsoft Teams — це платформа для співпраці, яка об'єднує чат, відеоконференції, зберігання файлів та інтеграцію інших застосунків в одному центрі. Він є частиною набору інструментів *Microsoft* 365 і призначений для полегшення спілкування та співпраці в організаціях, незалежно від того, чи є вони малими або великими компаніями [6]. Перша версія *Microsoft Teams* була випущена 14 березня 2017 року.

З головних функцій *Microsoft Teams* можна виділити:

– Чати та обмін повідомленнями. *Microsoft Teams* дозволяє комунікувати за допомогою приватних чатів або груп. Користувачі можуть ділитися повідомленнями, файлами та іншими медіаресурсами.

– Відео конференції. *Microsoft Teams* надає надійне рішення для відеоконференцій, що дозволяє користувачам проводити віртуальні зустрічі з колегами або клієнтами. Він підтримує відео- та аудіодзвінки, показ екрана та можливість планувати зустрічі або приєднуватися до них безпосередньо з програми.

– Канали. *Microsoft Teams* організований в канали, які діють як спеціальні місця для обговорень і співпраці навколо певних тем або проєктів. Канали можуть бути загальнодоступними або приватними, що дозволяє створювати гнучкі командні структури.

– Інтеграція програм сторонніх розробників. *Microsoft Teams* підтримує широкий спектр інтеграцій програм сторонніх розробників, що дозволяє користувачам переносити зовнішні інструменти та служби безпосередньо в інтерфейс

Teams. Сюди входять програми для управління проєктами, управління взаємовідносинами з клієнтами (*CRM*) тощо.

– Мобільна та кросплатформна підтримка. *Microsft Teams* доступний на різних пристроях, включаючи настільні комп’ютери, ноутбуки, смартфони та планшети. Ця кросплатформна підтримка гарантує, що користувачі можуть залишатися на зв’язку та співпрацювати незалежно від свого місцеперебування чи пристрою.

Зовнішній вигляд *Microsoft Teams* зображений на рис. 2.5.

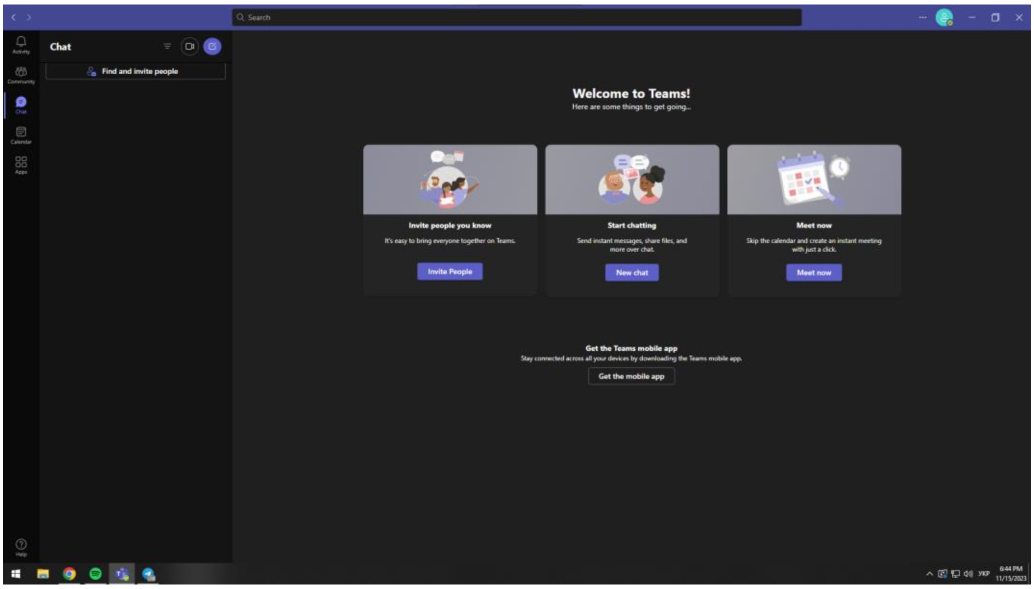


Рис. 2.5. *Microsoft Teams* у системі *Windows* 10

На момент написання цієї роботи, *Microsoft Teams* має приблизно 280 мільйонів активних користувачів кожен місяць. Застосунок підтримується у таких операційних системах: *Windows*, *Mac*, *iOS*, *Android*, and *Linux*.

2.1.1. Основні технології

У 2023 компанія Microsoft запустила проєкт *Microsoft Teams New*. У цьому клієнті вони змінили стек технологій клієнтської частини. Класичний застосунок *Microsoft Teams* використовував програмне забезпечення з відкритим кодом. Як хост застосунку виступав *Electron*, *AngularJS* використовувався як фреймворк для веброзробки, а різноманітні елементи керування були розроблені за допомогою *HTML* і *CSS*.

З початком розробки *Teams* у 2015 році, цей стек технологій дозволив швидко надати кросплатформенний вебклієнт і настільний застосунок для персональних комп’ютерів. Однак, з ростом можливостей та функціоналу застосунку, обрані технології сильно навантажували пристрої, що приводило до незручностей у користуванні. Усвідомлюючи це, команда *Microsoft Teams* почала свій аналіз доступних технологій, створення прототипів та визначення майбутню нову архітектуру [12].

Ключовими рішеннями стали перехід до колекції елементів керування *Fluent UI*, використання *React* замість *AngularJS* для побудови користувацьких інтерфейсів та використання *WebView2* як хост застосунку. Наочна схема технологічних стеків старого та нового клієнту Teams зображена на рис. 2.6.

Рис. 2.6. Клієнтські технології старого та нового *Microsoft Teams*

Fluent UI це бібліотека з готовими елементами користувацького інтерфейсу, яка може бути використана у будь-якому середовищі. *Fluent Design System* була розроблена компанією *Microsoft* та випущена у 2017 році. Ця система використовується у багатьох системах *Microsoft*.

Бібліотека *Fluent UI* дозволила розробникам *Teams* стандартизувати загальні компоненти та досягти узгоджених результатів на різних платформах. Використання цих адаптивних кросплатформних стилів і елементів керування, призвело до кращої продуктивності, порівняно з багатьма елементами з класичного клієнта *Microsoft Teams*, та забезпечило стабільну та однакову роботу на всіх пристроях [13].

Зовнішній вигляд компонентів *Fluent UI* зображений на рис. 2.7.

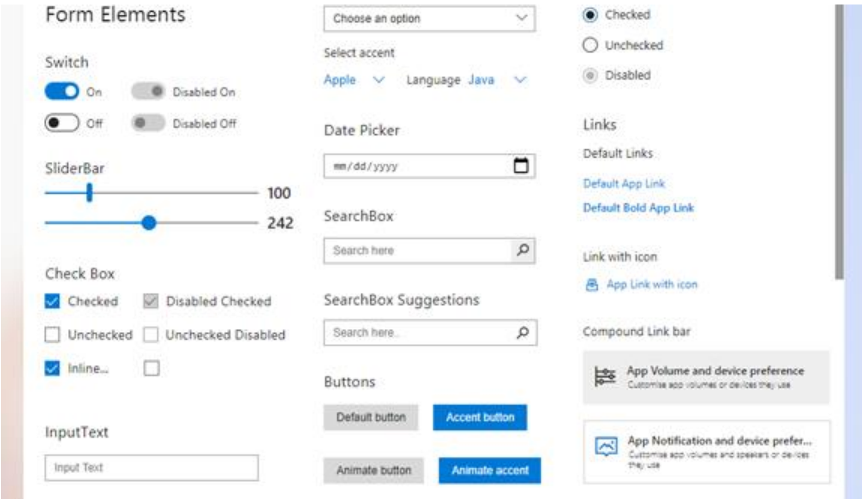


Рис 2.7. *Fluent UI*

Мова програмування *JavaScript* - однопотокова. Щоб подолати це обмеження, команда *Microsoft Teams* перемістила керування даними в окремий робочий процес, який називається *Client Data Layer*. Таке рішення дозволило працювати паралельно, не навантажуючи основний потік, таким функціям як вибірка й отримання даних, зберігання інформації, push-сповіщення та офлайн режим. *Client Data Layer* доступний через прошарок *GraphQL* з головного потоку програми. З'єднання між цими прошарками виконується завдяки набору засобів обміну повідомленнями між процесами.

Client Data Layer надав змогу прибрати певний функціонал з головного потоку програми, що відобразилося на її кращій продуктивності та роботі. Також, розділення обов'язків привело до більш оптимізованого, чистого та читабельного коду.

Microsoft Edge WebView2 дозволяє вбудовувати вебтехнології (*HTML*, *CSS* і *JavaScript*) у нативні застосунки. *WebView2* використовує *Microsoft Edge* як рушій для візуалізації вебкомпонентів. За допомогою цієї технології, розробники можуть вставляти веб код у різні частини нативної програми або повністю побудувати цілу систему.

WebView2 Runtime використовує ту саму модель процесу, що й браузер *Microsoft Edge*. Багато сучасних систем корпорації *Microsoft* використовують цю технологію, оскільки таким чином стандартизується кодова база.

Група процесів *WebView2* — це набір процесів *WebView2 Runtime*. Група процесів *WebView2* включає наступні процеси:

– Єдиний процес браузера.

– Один або кілька процесів візуалізації.

– Інші допоміжні процеси, наприклад процес графічного процесора та процес аудіослужби.

Групи процесів *WebView2 Runtime* зображені на рис. 2.8.

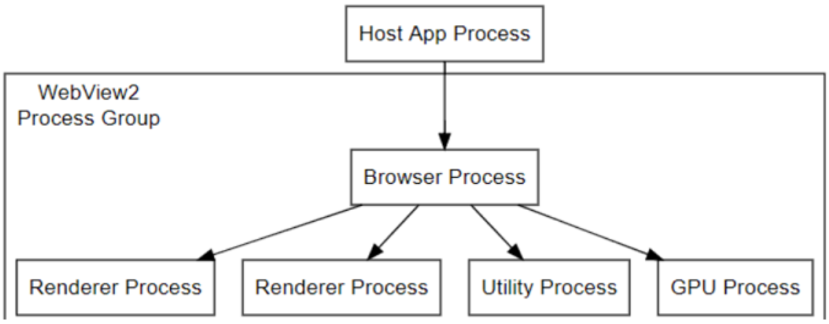


Рис. 2.8. Процеси *WebView2*

Кількість і наявність процесів у групі процесів *WebView2* може змінюватися, оскільки програма *WebView2* використовує різні функції.

Ключові переваги переходу від *Electron* до *WebView2* включають зменшення використання оперативної пам’яті та пам’яті диска.

WebView2 доступний як *SDK* для створення гібридних кросплатформенних програм з більш ефективним використанням ресурсів і кращою інтеграцією з операційною системою.

WebView2 забезпечує фінального користувача продуктивнішим та надійнішим застосунком. Цей перехід став позитивним кроком для *Microsoft Teams*.

2.1.2. Нова клієнтська архітектура та її переваги

Microsoft Teams з новою архітектурою працює вдвічі швидше, використовуючи половину системних ресурсів. Оновлений *Teams* забезпечує простіший і багатофункціональний досвід для різноманітної та наростальної бази користувачів.

Оновлена система надає гнучкіший досвід, наприклад, співпрацювати з людьми поза межами організації.

Хост системи використовує переваги технології *Edge WebView2*. *GraphQL* абстрагує прошарок *Client Data Layer*. Набір засобів обміну повідомленнями між процесами (*IPC*) діє як інструмент для встановлення з’єднання. Бібліотеки *React* і *Fluent* є стандартизованими технологіями, які використовуються для побудови гнучкого та багатофункціонального користувацького інтерфейсу. Мова програмування *TypeScript* робить код більш строгим та зрозумілим.

Діаграма, яка показує основні елементи та технології, з якої складається новий клієнт *Microsoft Teams*, зображена на рис. 2.9.

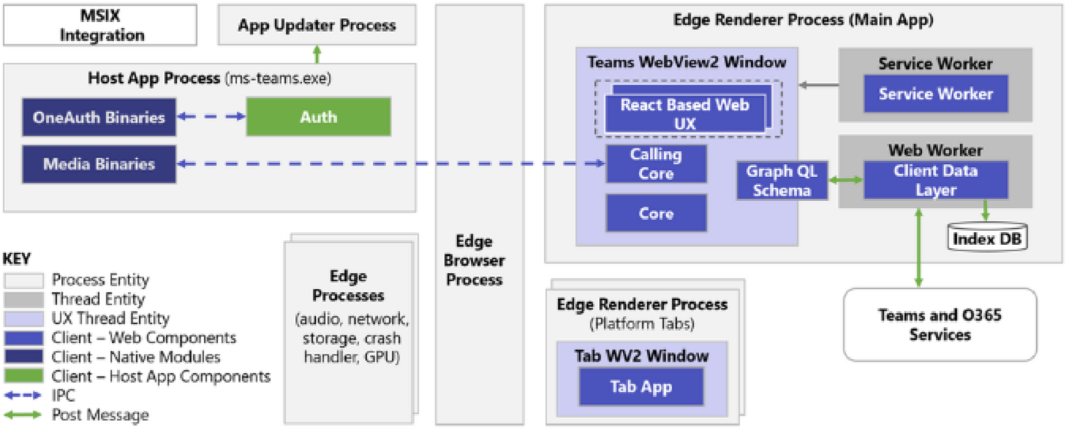


Рис. 2.9. Діаграма основних технологій *Teams*

Завдяки новій архітектурі, команда *Microsoft Teams* змогла імплементувати та покращити наступні функції:

– Перебудований конвеєр відтворення відео. Попит на більше використання відео під час зустрічей та інтенсивність його використання піднялись. З нуля був розроблений ефективніший процес керування відео, віртуалізований список для більш плавної навігації та був зроблений перехід до власного композитора для обробки медіа. Нова архітектура відео набагато ефективніша в обробці відео, що призводить до зниження енергоспоживання на 50% і дозволяє додатку *Teams* підтримувати складніші та розширені функції відео (наприклад, відеосітка 7x7, динамічні перегляди) на ширшому спектрі апаратного забезпечення.

– Масштабні зустрічі. Групові зустрічі широко використовуються для широкомасштабних комунікацій на рівні організації чи компанії, окрім прямої взаємодії невеликих груп. Були зроблені цілеспрямовані покращення продуктивності та масштабу великих зустрічей у *Teams*, де зазвичай є тисячі або більше одночасних учасників з інтенсивним використанням відео та чатів. Оптимізації варіювалися від групування та зменшення подій *IPC*, зменшення кількості візуалізацій інтерфейсу користувача та усунення шуму. Завдяки роботі з оптимізацією, є можливість забезпечити швидшу та послідовнішу затримку приєднання до наради.

– Можливість додавання кількох облікових записів. Був проведений комплексний перегляд підтримки для організацій із кількома клієнтами та обліковими записами. Це включає значні вдосконалення процесів автентифікації, синхронізації та сповіщень.

– Покращення безпеки. Були вжиті заходи для додавання розширеного захисту системи, застосувавши надійні типи та впровадивши більш сувору політику безпеки. Ці зусилля призвели до посилення захисту від атак типу *XSS*.

– Оптимізація пам'яті. Деякі стратегії, що використовуються для покращення продуктивності, іноді потребують витрат пам’яті (наприклад, кешування даних у пам’яті або попередня вибірка інформації чи попереднє завантаження коду). Була додана можливість динамічного вибору пам’яті на основі моделі використання. Крім того, кілька оптимізацій продуктивності, які включали зменшення обсягу даних, що збираються, обробляються та зберігаються локально, значно знизили споживання пам’яті.

Висновки за розділом

У другій частині, був зроблений докладний порівняльний аналіз схожих за функціональністю застосунків з прототипом системи цієї кваліфікаційної роботи. Порівняні були серверна частина програми *Discord* та клієнтська частина застосунку *Microsoft Teams* і *Microsoft Teams New*.

Досліджуючи систему *Discord*, було виявлено, що її сервера написані мовою програмування *Elixir* з використанням фреймворку *Phoenix*.

Elixir — це функціональна мова паралельного програмування, яка побудована на віртуальній машині *Erlang* (*BEAM*). Підкреслюючи принципи функціонального програмування, вона забезпечує легкі процеси для ефективного паралелізму та слідує моделі актора для передачі повідомлень. Мова *Elixir* була спроєктована для побудови надійних, відмовостійких систем з фокусом на підтримку та розширюваність.

Phoenix — це фреймворк веброзробки для *Elixir*, який брав натхнення з фреймворку *Rails*.

Discord використовує мікросервісну архітектуру з розділенням функціонала. Є 3 основних сервери - *Discord Gateway*, *Discord Guilds* і *Discord Voice*.

Для розробки *VoIP* функціоналу, розробники використовують *WebRTC* у браузері, а у нативних програмах, використовується свій власний рушій, який дозволяє додавання нового та унікального функціонала, який може бути корисним саме для користувачів *Discord*.

Застосунок *Discord* є потужним та надійним інструментом, який можуть використовувати як у великих компаніях для робочих справ, так і маленькому колі друзів.

У застосунку *Microsoft Teams* була розглянута стара та нова архітектура клієнтської частини.

Старий клієнт *Teams* використовував *Electron* як хост, *AngularJS* як веб фреймворк та *HTML* і *CSS* для побудови компонентів користувацького інтерфейсу. У новому клієнті ці технології були замінені на *WebView2, React* та *Fluent* відповідно, що дало значний приріст у продуктивності виконання програми, покращило користувацький досвід та зменшило кількість проблем старого клієнту.

Microsoft Teams та *Discord* є великими та складними застосунками, які надають свої послуги та функції мільйонам користувачам щодня. Розробники цих програм працюють над складними рішеннями, які забезпечують надійність та стабільність фінального продукту. Проєкт цієї кваліфікаційної роботи є простим прототипом і не має складних архітектурних рішень чи багатого функціоналу. Його задачею є показати базову ідею та як її можна реалізувати.

РОЗДІЛ 3
ІМПЛЕМЕНТАЦІЯ ПРОТОТИПУ СИСТЕМИ

3.1. Структура прототипу і базове налаштування

Кожен проєкт має свою файлову структуру. Файлова структура може залежати від обраних технологій, архітектури та вподобань команди розробників. Також, кожен проєкт може мати свої налаштування. Наприклад, настройки форматувальнику коду, мови програмування, середовища, де був розгорнутий проєкт та інші.

3.1.1. Файлова структура

Коренева тека проєкту має 3 головних елементи:

– Тека “*packages*”. Це тека, яка містить кожну частину проєкту. У цьому випадку, цими проєктами є client і server.

– Файл “*.gitignore*”. Цей файл має записи про елементи у файловій структурі, які не треба зберігати у *git*-репозиторії.

– Файл “*README.md*”. У цьому файлі зберігається загальний опис проєкту, процес його запуску, технологічний стек та інше.

Така файлова структура робить проєкт більш зрозумілим і легкокерованим у *git*-репозиторії. Також, під таку структуру буде простіше писати *Docker* конфігурацію для просто запуску системи у будь-якому середовищі за допомогою контейнерів.

Тека “*client*” - місце вихідного коду клієнтської частини. Вона містить в собі декілька конфігураційних файлів для npm і *TypeScript*, теку “*src*” з головним кодом та теку “*public*” зі статичними файлами та базовим *HTML* файлом. Така структура пропонується генератором *React* проєкту - “*create-react-app*”. Теку “*src*” розробники наповнюють самостійно. У випадку цього проєкту, було вирішено зробити 5 головних груп:

– “*components*”. В цій теці зберігаються всі *React*-компоненти, їхні типи та стилізація.

– “*config*”. Ця тека зберігає певні конфігураційні файли для окремих бібліотек.

– “*global*”. Місце для зберігання глобальних речей, які використовуються у різних компонентах по всьому проєкту. Це можуть бути певні типи, стилі або код, який часто повторюється.

– “*hooks*”. У *React*, з додаванням підтримки функціональних компонент, були введено термін *React Hooks*. Розробники також можуть створювати свої власні хуки, для перевикористання якоїсь певної логіки. Саме в цій теці зберігаються такі хуки.

– “*redux*”. Для ефективнішого управління станом компонентів, в цьому проєкті використовується бібліотека “*redux*” та “*redux/toolkit*”. В цій теці зберігається все, що пов’язано з цією бібліотекою, а також виклики *API*.

Також, на цьому рівні розташований файл “*index.tsx*” та “*index.css*”. “*index.tsx*” - це файл, який є початковою точкою для старту *React* проєкту. В ньому відбувається монтування головного *React*-компонента у *HTML* структуру. Файл “*index.css*” має глобальні стилі, які розповсюджуються на весь проєкт.

Поверхнева файлова структура клієнтської частини зображена на рис. 3.1.

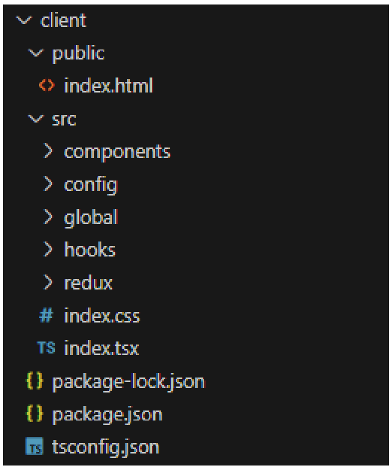


Рис. 3.1. Файлова структура клієнтської частини

Тека “*server*” - місце, де знаходиться вихідний код серверної частини проєкту. Так само як і клієнтська частина, на кореневому рівні тека *server* має конфігураційні файли та теку “*src*”. Серверна частина проєкту використовує бібліотеку *NestJS*, яка пропонує свою власну файлову структуру і концепт розробки. Виходячи з цього, тека src має 5 головних груп:

– “*common*”. Місце, в якому зберігаються речі загального призначення, які можуть бути використані у різних частинах проєкту.

– “*config*”. Ця тека має таке саме призначення, як і у клієнтській частині. Додатково, в ній знаходиться файл, який збирає всі секрети та ключі проєкту в один об’єкт.

– “*decorators*”. В цій теці зберігаються класові декоратори, які були написані розробниками для цілей проєкту.

– “*guards*”. “*guard*” - термін, який надається бібліотекою *NestJS*. З їхньою допомогою, можливо захищати певні частини застосунку від не авторизованого доступу.

– “*modules*”. Найголовніша частина проєкту. *NestJS* пропонує розбивати логіку застосунку на різні модулі.

Окрім цих 5 груп, корінь теки “*src*” має головний файл “*main*” та головний модуль “*app*”. Поверхнева структура зображена на рис. 3.2.

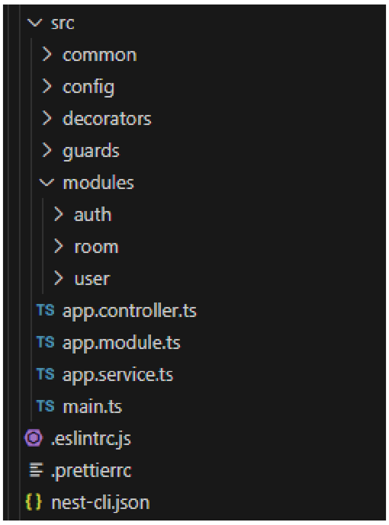


Рис. 3.2. Файлова структура серверної частини

3.1.2. Конфігураційні файли

Як було вказано вище, кожна частина має свої певні файли з конфігураціями та базовими налаштуваннями. Без цих файлів, проєкт буде працювати неправильно, або не буде працювати взагалі. У цьому проєкті є мінімальні та базові налаштування для локального запуску проєкту.

Обидві частини мають декілька схожих файлів конфігурації, а саме “*package.json*” та “*tsconfig.json*”. Файл “*package.json*” - це файл конфігурації для проєктів, написаних на платформі *Node.js*. Він містить загальні метадані про проєкт, такі як назва, версія, залежності скрипти та інше. Приклад файлу “*package.json*” серверної частини зображений на рис. 3.3.

```
{
  "name": "server",
  "version": "0.0.1",
  "description": "",
  "author": "",
  "private": true,
  "license": "UNLICENSED",
  ▷ Debug
  "scripts": {
    "build": "nest build",
    "format": "prettier --write \"src/**/*.ts\" \"test/**/*.ts\"",
    "start": "nest start",
    "start:dev": "nest start --watch",
    "start:debug": "nest start --debug --watch",
    "start:prod": "node dist/main",
    "lint": "eslint \"{src,apps,libs,test}/**/*.ts\" --fix",
    "test": "jest",
    "test:watch": "jest --watch",
    "test:cov": "jest --coverage",
    "test:debug": "node --inspect-brk -r tsconfig-paths/register -r ts-node
    "test:e2e": "jest --config ./test/jest-e2e.json"
  },
  "dependencies": {…
  },
  "devDependencies": {…
  },
```

Рис. 3.3. Приклад файлу “*package.json*” серверної частини

Файл поділяється на певні частини. Найголовніші з них описані нижче:

– “*dependencies*” та “*devDependencies*”. Тут містяться всі залежності проєкту, які необхідні для його правильної роботи. Залежності, які записані в “*devDependencies*”, використовуються тільки для розробки.

– “*scripts*”. Розділ “*scripts*” містить скрипти пов’язані з проєктом. Це можуть бути скрипти для роботи з базою даних, запуску проєкту, запуску тестів та інше.

– “*name*”, “*version*”, “*descripion*”, “*author*”, “*private*”, та “*license*”. Це загальні поля, які зберігають метаінформацію про проєкт.

Серверна частина має ще 3 додаткових файли - “*.eslintrc.js*”, “*.prettierrc*” і “*nest-cli.json*”. “*.eslintrc.js*” і “*.prettierrc*” налаштовують аналізатор та форматувальник коду. Файл “*.eslintrc.js*” зображений на рис. 3.4.

У файлі “*.eslintrc.js*” описуються правила для перевірки коду, а саме описання того, що буде вважатися помилкою, а що ні. “*.prettierrc*” - файл для налаштування *Prettier*. Він має дуже просту структуру - ключ та значення. *ESLint* та *Prettier* взаємопов’язані між собою. *ESLint* показує помилки, а *Prettier* виправляє помилки форматування. Додатково, *ESLint* може бути налаштований разом з *TypeScript*, що надасть змогу одразу бачити помилки, через які код *TypeScript* не буде скомпільований у код *JavaScript*.

Приклад конфігураційного файлу “*.eslintrc.js*” зображений на рисунку 3.4.

```
module.exports = {
  parser: '@typescript-eslint/parser',
  parserOptions: {
    project: 'tsconfig.json',
    tsconfigRootDir: __dirname,
    sourceType: 'module',
  },
  plugins: ['@typescript-eslint/eslint-plugin'],
  extends: [
    'plugin:@typescript-eslint/recommended',
    'plugin:prettier/recommended',
  ],
  root: true,
  env: {
    node: true,
    jest: true,
  },
  ignorePatterns: ['.eslintrc.js'],
  rules: {
    '@typescript-eslint/interface-name-prefix': 'off',
    '@typescript-eslint/explicit-function-return-type': 'off',
    '@typescript-eslint/explicit-module-boundary-types': 'off',
    '@typescript-eslint/no-explicit-any': 'off',
  },
};
```

Рис. 3.4. Структура файлу “*.eslintrc.js*”

Ключі та секрети проєкту зберігаються у файлах “*.env*”. Ці файли виключені з *git*-репозиторію, задля збереження приватності інформації. Файли “.env” зберігають в собі змінні середовища.

Ці змінні вже включено в екосистему *Node.js*, що дає їм значну перевагу порівняно з альтернативними варіантами конфігурації, такими як файл “*config.js*” або “*config.json*”. Змінні середовища, особливо коли вони використовуються в поєднанні з автоматизацією, як-от конвеєр побудови, дозволяють уникнути таких неприємних речей, як написання скриптів для конфігурацій.

Інші конфігурації можуть бути знайдені у теках “*config*”. На даному етапі, в цій теці можливо знайти 2 файли. Перший з них - це файл “*opentok.ts*”. В цьому файлі іде базова конфігурація бібліотеки OpenTok, а саме створення сутності класу з необхідними ключами та секретами. Ця сутність буде використовуватись у всьому коді. Її імплементація зображена на рис. 3.5.

```
import * as OpenTok from 'opentok';

import configuration from './configuration';

const opentok = new OpenTok(
  configuration().vonage.apiKey,
  configuration().vonage.apiSecret,
);

export { opentok };

```

Рис. 3.5. Файл “*opentok.ts*”

Другим файлом у теці “*config*” є “*configuration.ts*”. Цей файл збирає всі секрети та змінні середовища в один об’єкт, що спрощує їх використання по всьому коду. Файл “*configuration.ts*” зображений на рис. 3.6.

```
import { config } from 'dotenv';

config();

export default () => ({
  env: process.env.NODE_ENV || 'development',
  port: parseInt(process.env.PORT, 10) || 5000,
  database: {
    host: process.env.DB_HOST,
    type: 'postgres',
    port: process.env.DB_PORT,
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    autoLoadEntities: true,
    synchronize: true,
    entities: ['../modules/**/*.entity.ts'],
  },
  jwt: {
    secret: process.env.JWT_SECRET,
  },
  vonage: {
    apiKey: process.env.VONAGE_API_KEY,
    apiSecret: process.env.VONAGE_API_SECRET,
  },
});

```

Рис. 3.6. Файл “*configuration.ts*”

Файл “*configuration.ts*” експортує одну функцію, яка повертає об’єкт, в якому зберігаються всі необхідні секрети. Для зручності, об’єкт поділений на групи, такі як “*database*” та “*vonage*”. Таким чином їх простіше читати та розуміти.

Є різні стилі, методи та підходи для зберігання чутливих даних застосунків. Кожен розробник обирає такий підхід, який йому найбільше подобається та який буде зручним. Головною умовою різних методів є забезпечення безпеки інформації. Цією умовою не можна нехтувати у сучасному світі, оскільки це може поставити під загрозу вашу систему та дані ваших користувачів.

3.2. Логіка процесу авторизації та аутентифікації

Авторизація та аутентифікація це два різних процеси. Аутентифікація - це процес перевірки ким є користувач, а авторизація - процес перевірки доступів, які він має. Існує багато різних методів для побудови цих процесів. Наприклад, процес аутентифікації може бути виконаний такими основними шляхами:

– Базова аутентифікація. Такий метод пропонує реєстрацію та вхід за допомогою звичайного імені користувача та паролю.

– Аутентифікація за допомогою сторонніх сервісів. З цим методом, користувачам не треба заповнювати імена користувача та пароль. Вони можуть здійснити вхід за допомогою таких сервісів як *Google*, *Facebook* та інших.

Додатковим захисним прошарком може бути двофакторна перевірка. Цей процес потребує прив’язки номера телефону, пошти або спеціального застосунку до облікового запису користувача.

У прототипі системи цього кваліфікаційного проєкту використовується базова аутентифікація та авторизація за допомогою *JWT* токену.

3.2.1. Імплементація процесів авторизації та аутентифікації у *NestJS*

Для побудови системи авторизації та аутентифікації на стороні сервера, необхідно створити два окремих *NestJS* модуля, один “*guard*” та один “декоратор”.

Перший модуль *NestJS* буде називатись “*user*”. В цьому модулі буде зберігатись все необхідне для роботи з користувачами. Наприклад, його схема в базі даних, методи для створення або отримання та інше.

Почнемо розробку цього модуля, з побудування *Typeorm* схеми. Базова схема користувача є доволі простою. Вона буде містити в собі 3 поля - “*name*”, “*username*” та “*password*”. Ці поля будуть обов’язковими для створення нової сутності. З часом, ця схема буде ускладнена додатковими полями, а саме “*createdRooms*” та “*rooms*”. “*createdRooms*” буде зв’язком з таблицею “*room*” і буде тримати всі кімнати, які були створені цим користувачем. “*rooms*” буде зв’язком, який буде описувати всі кімнати, в яких користувач є учасником. “*createdRooms*” буде описаний зв’язком “*one to many*”, а “*rooms*” - “*many to many*”. Повна *Typeorm* схема користувача зображена на рис. 3.7.

```
@Entity()
export class User extends BaseEntity {
  @Column({ nullable: false })
  name: string;

  @Column({ nullable: false })
  username: string;

  @Column({ nullable: false })
  password: string;

  @OneToMany(() => Room, (room) => room.author)
  createdRooms: Room[];

  @ManyToMany(() => Room, (room) => room.participants)
  rooms: Room[];
}
```

Рис. 3.7. *Typeorm* схема користувача

Також модуль “*User*” буде мати деякі функції для роботи з моделлю, а саме метод “*create*” та метод “*getUserEntityById*”. Метод “*create*” перевіряє чи вільне введене користувачем ім’я, хешує пароль та зберігає користувача у базу даних. Хешування відбувається за допомогою бібліотеки “*bcrypt*”. Код методу “*create*” зображений на рис. 3.8.

```
async create(props: CreateUserDTO) {
  const doesUserExist = await this.getUserEntityByUsername(props.username);

  if (doesUserExist) {
    throw new ConflictException('username is already taken.');
  }

  const hashedPassword = await bcrypt.hash(props.password, 10);

  const user = this.userRepository.create({
    ...props,
    password: hashedPassword,
  });

  return this.userRepository.save(user);
}
```

Рис. 3.8. Код методу “*create*”

Метод “*getUserEntityById*” є допоміжнім і намагається знайти користувача в базі даних по ID. Ця функція, у більшості випадків, буде використовуватись як допоміжний в інших методах.

Модуль “*auth*” працює з усім, що пов’язано з авторизацією та автентифікацією користувачів. Таке розділення робить код більш читабельним та зрозумілим для наступних розробників системи. Модуль “*auth*” має контролер, в якому є 4 головних *HTTP*-маршруту, а саме:

– “*/auth/signup*”. Використовується для реєстрації нових користувачів шляхом створення нового облікового запису.

– “*/auth/signin*”. Використовується для аутентифікації зареєстрованих користувачів.

– “*/auth/me*”. Використовується для отримання інформації про поточного користувача, для її відображення на клієнті.

– “*/auth/logout*”. Використовується для виходу з системи.

У складніших системах, можуть бути додаткові методи, які використовуються для відновлення імені користувачу та паролю, генерації коду для двофакторного захисту та інші. Також, в модуль “*auth*” можна включити функціонал для авторизації за допомогою таких сторонніх ресурсів як *Google Account*, *Facebook*, *Twitter*, *GitHub* та інших.

Імплементація сервісного методу контролеру “*/auth/signup*” зображена на рис. 3.9.

Рис. 3.9. Контролер “*/auth/signup*”

Контролери “*/auth/signup*” та “*/auth/signin*” працюють схожим чином. Обидва методи генерують *JWT* токен та зберігають його в *HTTP* cookie. Різниця полягає у тому, що метод “*/auth/signup*” просто створює нового користувача, а метод “*/auth/signin*” перевіряє дані, які були введені користувачем.

Наступним кроком буде імплементація таких сутностей як “*guard*” та декоратор.

“*guard*” - це *NestJS* сутність, яка виступає за захист ресурсів. Завдяки їй, розробники мають можливість перевіряти токени користувачів, їхні сесії, декодовану інформацію, ролі користувачів та багато іншого.

Декоратор — це особливий вид оголошення, яке можна приєднати до оголошення класу, методу, засобу доступу, властивості або параметра. Завдяки цій сутності, розробники мають можливість розширювати можливості своїх класів та розробляти додатково логіку та функціонал.

Логіка сутності “*guard*” цієї системи доволі проста. З запиту забирається *cookie*, в якому є або нема *JWT* токену. Після цього цей токен дешифрується. Якщо токена не існує, або він не валідний, або інформація всередині його не валідна - запит буде відхилений. У іншому випадку, код продовжить своє виконання. Ця логіка зображена на рис. 3.10.

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<any> {
    const request = context.switchToHttp().getRequest();

    const token = request.cookies['access_token'];

    if (!token) {
      throw new UnauthorizedException();
    }

    try {
      const payload = await this.jwtService.verifyAsync(token, {
        secret: configuration().jwt.secret,
      });

      request['userId'] = payload.sub;
    } catch (error) {
      throw new UnauthorizedException();
    }

    return true;
  }
}
```

Рис. 3.10. Код “*guard*”

Декоратор “*GetCurrentUserId*” є спрощенням отримання *ID* поточного користувача у захищених *HTTP*-маршрутах. За допомогою *ID* є можливість отримати повну інформацію користувача та навіть його роль, якщо таке поле існує у базі даних.

3.2.2. Розробка екранів входу та реєстрації на стороні клієнта

Як було описано у розділі 3.1, цей проєкт має групу “*components*”. Всередині цієї групи також є група з назвою “*screens*”. Тека “*screens*” є місцем, де зібрані всі компоненти, які є кореневою обгорткою якогось місця системи. Прикладом таких файлів можуть бути “*Dashboard*”, “*Profile*”, “*Catalogue*” та інші. В цьому підрозділі роботи, будуть додані файли “*SignIn*” та “*SignUp*”.

Ці файли є *React* компонентами, в яких буде відбуватись вся головна логіка. Вони також можуть використовувати всередині себе інші *React* компоненти

Також, для коректної роботи, необхідно створити сутність у *Redux*. Вона буде мати назву “*user*” і буде працювати тільки з даними користувача. Стан сутності “*user*” бути зберігати три ключових поля, а саме “*id*”, “*name*” та “*username*”. Для роботи з цим станом вистачить одного методу, який буде заповнювати ці поля або робити їх пустими. Ця логіка зберігається у файлі “*reducer.ts*” і зображена на рис. 3.11.

```
export const initialUser: User = {
  id: '',
  name: '',
  username: '',
};

const initialState: UserState = {
  user: initialUser,
};

export const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {
    setUser: (state, action: PayloadAction<SetUserPayload>) => {
      state.user = action.payload.user;
    },
  },
});
```

Рис. 3.11. Файл “*reducer.ts*” сутності “*user*”

Запити до *API* серверу будуть зберігатись в окремому файлі - “*api.ts*”. Всі запити є окремими методами класу. Кожен метод приймає необхідні параметри для свого функціонування. *API* файл сутності “*user*” буде містити в собі запити до *HTTP* маршрутів модуля “*auth*”.

Оскільки в цьому проєкті буде використовуватись базова аутентифікація, форма входу та реєстрації буде простою.

Компонент “*SignIn*” має два основних методи, а саме “*handleSubmit*” та “*hndleInputChange*”. Метод “*handleSubmit*” буде спрацьовувати тоді, коли користувач буде підтверджувати форму. Тіло методу виконує декілька запитів. Перший запит - це запит на вхід. Параметрами передаються дані з форми - “*username*” та “*password*”. У випадку успіху, проводиться другий запит “*getMe*”, де іде спроба отримати дані користувача завдяки токену. Якщо це запит буде також успішним, то дані користувача будуть збережені в стан *Redux*. Імплементація методу “*handleSubmit*” зображена на рис. 3.12.

```
const handleSubmit = async (
  e: React.FormEvent<HTMLFormElement>
): Promise<void> => {
  try {
    e.preventDefault();

    await UserAPI.signin(data);
    const response = await UserAPI.getMe();

    dispatch(setUser({ user: response.data }));

    navigate('/');
  } catch (error) {
    console.error(error);
  }
};
```

Рис. 3.12. Метод “*handleSubmit*”

Дані з форми зберігаються в локальному стані компонента. Отримуються вони з текстових полів форми. Для слідкування за змінами даних у цих полях, використовується метод “*onChange*”. В цей метод треба передати функцію, яка буде виконуватись на кожну зміну значення всередині поля. Це буде функція “*handleInputChange*”. Вона буде приймати параметром об’єкт “*event*”, в якому буде вся інформація про текстове поле та його стан. Після отримання, функція буде оновлювати стан компонента і таким чином зберігати введені дані. Функція “*handleInputChange*” зображена на рис. 3.13.

```
const handleInputChange = (e: React.ChangeEvent<HTMLInputElement>): void => {
  set$data((prev) => ({
    ...prev,
    [e.target.name]: e.target.value,
  }));
};
```

Рис. 3.13. Метод “*handleInputChange*”

Для розробки зовнішнього вигляду використовувались готові *MUI* компоненти “*Input*”, “*Button*” та “*Typography*”. Вони мають декілька варіантів зовнішнього вигляду і багато вбудованих функцій. Зовнішній дизайн сторінки реєстрації зображений на рис. 3.14.

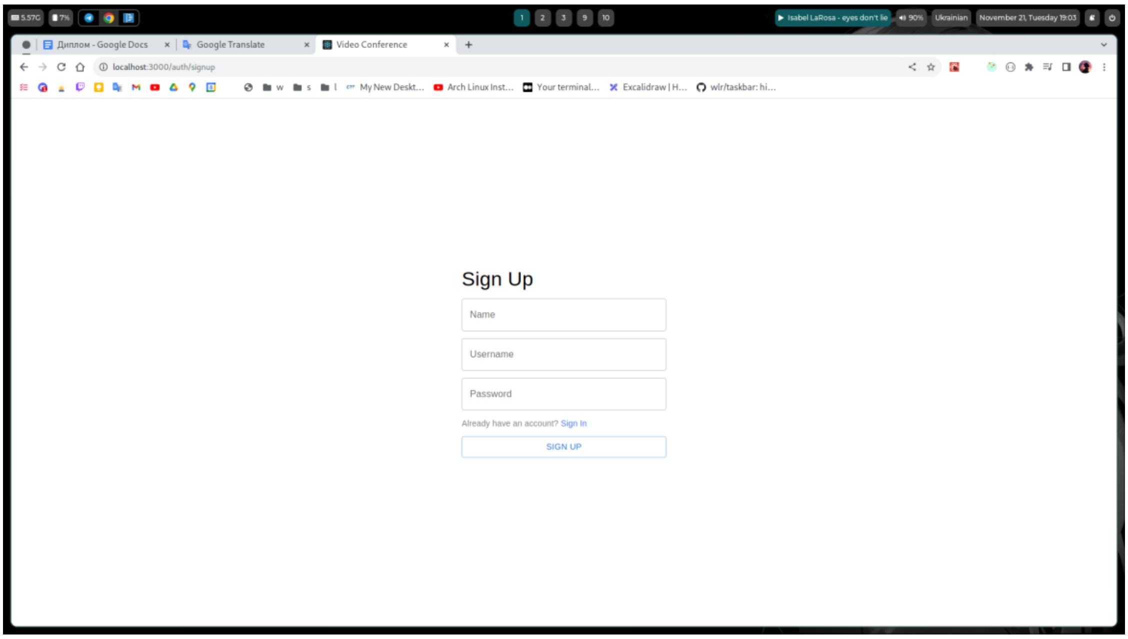


Рис. 3.14. Дизайн сторінки реєстрації

3.3. Взаємодія з відеокімнатами

У цьому підрозділі буде описані методи для взаємодії з кімнатами. Кімнати - головна сутність проєкту. Також, для роботи з кімнатами, необхідно налаштувати систему *OpenTok*, за допомогою якої будуть створюватись сесії. Сесія *OpenTok* - це також кімната, але вона створюється на серверах *OpenTok*. В кімнаті на стороні проєкту будуть збережені всі дані які пов’язані з системою, а також *ID* сесії *OpenTok*. Сутність “*room*” буде зберігати такі поля:

– “*title*”. Назва кімнати, обов’язкове поле.

– “*description*”. Опис кімнати, необов’язкове поле.

– “*startDate*”. Дата початку зустрічі, обов’язкове поле.

– “*opentokSessionId*”. *ID* сесії *OpenTok*, обов’язкове поле. За допомогою цього *ID* буде відбуватись підключення до сесії *OpenTok*.

– “*author*”. Дані про користувача, який створив зустріч. Це поле є “*many-to-one*” зв'язком з таблицею “*user*”.

– “*participants*”. Масив даних про всіх учасників зустрічі. Це поле є “*many-to-many*” зв’язком з таблицею “*user*”.

Ці дані можуть бути доповнені великою кількістю інформації, але для базового прототипу цього достатньо. Typeorm схема сутності “*room*” зображена на рис. 3.15.

```
@Entity()
export class Room extends BaseEntity {
  @Column({ nullable: false })
  title: string;

  @Column({ nullable: true })
  description: string;

  @Column({ nullable: false })
  startDate: Date;

  @Column({ nullable: false })
  opentokSessionId: string;

  @ManyToOne(() => User, (user) => user.createdRooms)
  @JoinColumn()
  author: User;

  @ManyToMany(() => User, (user) => user.rooms)
  @JoinTable()
  participants: User[];
}
```

Рис. 3.15. Typeorm схема сутності “*room*”

Метод “*create*” власне створює кімнату у системі. На вхід він приймає основні параметри, а саме “*currentUserId*”, “*title*”, “*description*”, “*participants*” та “*startDate*”. Поле “*currentUserId*” береться із токена користувача та необхідне для розуміння того, ким ця операція виконується. Валідація полів клієнтської частини зображена на рис. 3.16.

```
export class CreateRoomDTO {
  @IsString()
  title: string;

  @IsString()
  startDate: string;

  @IsString({ each: true })
  @IsNotEmpty()
  @IsArray()
  participants: string[];

  @IsOptional()
  @IsString()
  description: string;
}
```

Рис. 3.16. Валідація полів у *NestJS*

Якщо всі початкові пункти валідації були пройдені, починається виконання тіла методу.

Першим кроком є перевірка того, чи існує користувач, який виконує операцію. Якщо користувача не існує, виконання припиниться з помилкою 404.

Наступним кроком є пошук всіх користувачів, які є майбутніми учасниками зустрічі. Масив “*participants*” містить в собі тільки імена користувачів, тому що вони являються унікальним ідентифікатором, по якому їх можливо знайти у системі. Завдяки цьому масиву, робить пошук користувачів. Якщо якогось користувача не було знайдено, операція буде перервана. Ця логіка може бути змінена. Наприклад, якщо один користувач з п’яти не був знайдений, з зустріч буде створена тільки з чотирма учасниками, а про не знайденого буде відправлена помилка-сповіщення.

Наступна операція є взаємодією з *API OpenTok*, а саме, створення сесії. Для її створення нема потреби у передачі якихось особливих параметрів, тому це операція нескладна. При створенні сесії, можна вказати її режим. Існує два режими: “routed” та “relayed”. В цьому проєкті використовується режим “*routed*”, для того, щоб зв’язок відбувався через сервер *OpenTok*, а не шляхом “*peer-to-peer*”. Після створення сесії, метод *OpenTok API* повертає дані про неї. Ця операція зображена на рис. 3.17.

```
const session: Session = await new Promise((resolve, reject) => {
  opentok.createSession(
    { mediaMode: 'routed' },
    (error: Error | null, session: Session | undefined) => {
      if (error) {
        reject(error);
        return;
      }

      resolve(session);
    },
  );
});
```

Рис. 3.17. Створення сесії *OpenTok*

Останньою операцією є збереження створеної зустрічі до бази даних. Зберігаються дані, які були передані користувачем, системні дані, а також дані, які були повернені після створення сесії *OpenTok*.

Процес створення кімнати на клієнтській стороні відбувається за допомогою модального вікна. У цьому вікні присутні 4 поля для заповнення базової інформації. На цей момент, учасники запрошуються переліком їхніх імен через кому. У майбутньому, цю логіку можна покращити, додавши автоматичне заповнення імен, пропозиції, історію доданих людей у минулому та зробивши зручний дизайн. Зовнішній вигляд вікна створення кімнати з заповненими полями зображений на рис. 3.18.

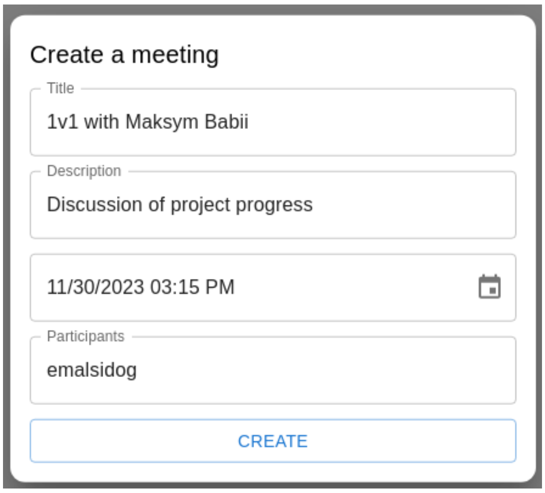


Рис. 3.18. Модальне вікно створення кімнати

Всі кімнати, які були створені поточним користувачем або в яких поточний користувач є учасником, можуть бути знайдені на домашній сторінці системи. Вони відображаються як картки з назвою, описом, датою та кнопкою "*Join*". Приклад зображений на рис. 3.19.

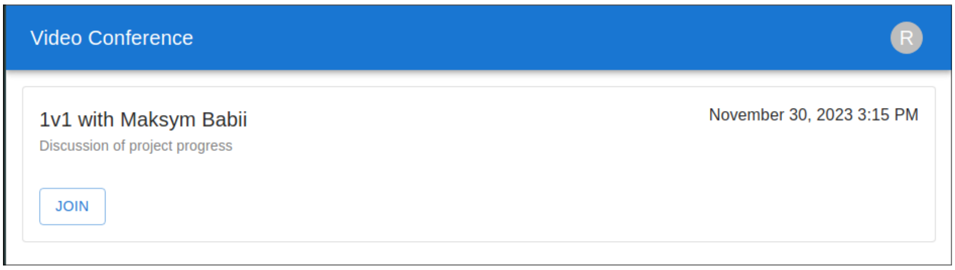


Рис. 3.19. Зовнішній вигляд створеної кімнати

У майбутньому, можна покращити цей функціонал наступним чином:

– Блокувати кнопку "*JOIN*" за певний час до початку зустрічі.

– Групувати картки в залежності від часу зустрічі. Прикладом можуть бути такі групи: "Минулі", "В процесі" та "Майбутні".

– Відображати список запрошених людей та людей, які вже доєднались. Такий список покращить користувацький досвід шляхом надавання корисної інформації.

Методи видалення та зміни кімнат набагато простіші в імплементації, оскільки не потребують взаємодії з *API OpenTok.*

3.4. Імплементація відеодзвінка

У цьому розділі буде описана логіка найважливішого та головного функціоналу застосунку - відеозв'язку.

3.4.1. Доєднання до кімнати

Щоб розпочати дзвінок, користувачу необхідно натиснути на кнопку “*Join*”. Після цієї дії, він буде автоматично підключений до кімнати. У випадку цього проєкту, проміжного лобі, в якому користувач міг би провести фінальні налаштування, не буде і тому користувач одразу опиниться у дзвінку.

У майбутніх версіях, додавання проміжного лобі буде важливою задачею, оскільки це надає користувачам важливу інформацію про зустріч, таку як кількість учасників, які вже доєднались, їхні імена, назву та опис зустрічі, її дату та час. Також проміжне лобі слугує місцем, де користувачі можуть додатково налаштувати свій мікрофон та камеру, перевірити, як їх видно в кадрі та як їх чути. Додатково, тут можуть знаходитись опції для фільтрів заднього фону та функцій, які присутні саме в цій системі (реакції, сповіщення тощо).

При натисканні кнопки “*Join*”, користувач буде перенаправлений на інший екран. *URL* адреса буде виглядати наступним чином - “*http://localhost:3000/5674*”, де 5674 - *ID* кімнати. *ID* кожної кімнати є унікальним. Перед доєднанням до зустрічі відбувається перевірка поточного користувача, а саме чи є він автором зустрічі, або чи є він запрошеним до неї. При заході на цю сторінку буде спрацьовувати функція “*createOpentokSession*”, яка є основним механізмом для встановлення з’єднання.

Першим кроком є запит до сервера з *ID* кімнати який передається як параметр. Це необхідно для того, щоб знайти необхідну кімнату у базі даних та перевірити чи існує вона взагалі. Це робиться за допомогою методу “*getOpentokTokenByRoomId*”. Цей метод також виконує важливу функцію, а генерація *OpenTok* токену, за допомогою якого буде відбуватись доєднання до раніше згенерованої сесії. Щоб згенерувати цей токен, необхідно використати метод “*generateToken*”, який надається бібліотекою *OpenTok*. Параметром необхідно передати *ID* сесії для якої генерується токен. Також, ця функція приймає параметр типу *string*. Інформація цього параметру буде закодована всередині згенерованого токена. У цьому випадку, там буде збережена інформація про поточного користувача у вигляді *JSON*. Метод “*getOpentokTokenByRoomId*” зображений на рис. 3.20.

```
async getOpentokTokenByRoomId(
  currentUserId: string,
  roomId: string,
): Promise<{ token: string; sessionId: string }> {
  const currentUser = await this.userService.getUserEntityById(currentUserId);

  if (!currentUser) {
    throw new NotFoundException(`User (ID: ${currentUserId}) was not found.`);
  }

  const room = await this.roomRepository.findOne({ where: { id: roomId } });

  if (!room) {
    throw new NotFoundException(`Room (ID: ${roomId}) was not found.`);
  }

  const token = opentok.generateToken(room.opentokSessionId, {
    data: JSON.stringify(currentUser),
  });

  return {
    token,
    sessionId: room.opentokSessionId,
  };
}
```

Рис. 3.20. Процес отримання токена та *OpenTok ID*

Наступним кроком є ініціалізація *OpenTok* сесії. Це робиться за допомогою *OpenTok API* метода “*initSession*”. Як параметри, ця функція приймає ключ та “*opentokSessionId*”. Після успішної ініціалізації, з’являється можливість створити обробники подій. *OpenTok* генерує певні події при деяких діях користувачів. Метод “createOpentokSession” зображений на рис. 3.21.

```
const createOpentokSession = async () => {
  try {
    const response = await RoomAPI.getOpentokTokenByRoomId({
      roomId: params.roomId,
    });

    const { data } = response;

    const OT = window.OT;

    const session = OT.initSession(
      process.env.REACT_APP_VONAGE_API_KEY,
      data.sessionId
    );

    handleSessionEvents(session);

    publisher.current.session = session;
    publisher.current.token = data.token;
    publisher.current.properties = {…
    };

    subscribers.current.session = session;
    subscribers.current.token = data.token;
    subscribers.current.properties = {…
    };
```

Рис. 3.21. Метод “*createOpentokSession*”

Всі обробники подій винесені в окрему функцію, яка називається “*handleSessionEvents*”. Як параметр, вона приймає новостворену сесію, завдяки чому всередині функції є доступ до всіх властивостей сесії. Існує багато різних подій, але в цьому проєкті, використовується дві, а саме - “*connectionCreated*” і “*connectionDestroyed*”. Ці дві події відбуваються тоді, коли будь-який користувач або підключається до сесії, або відключається. Функція “*handleSessionEvents*” зображена на рис. 3.20.

```
const handleSessionEvents = (session) => {
  session.on("connectionCreated", function (e) {
    set$participantsNumber((prev) => prev + 1);
  });

  session.on("connectionDestroyed", function (e) {
    set$participantsNumber((prev) => prev - 1);
  });
};
```

Рис. 3.20. Обробник подій “*handleSessionEvents*”

Як видно з рисунка, події використовуються для стеження за кількістю користувачів у дзвінку. Кількість осіб допоможе динамічно змінювати макет застосунку.

Останнім кроком виконання функції ініціалізації буде налаштування *OpenTok API*, а саме таких сутностей як “*publisher*” та “*subscriber*”. “*publisher*” - це є поточний користувач, а “*subscriber*” - це всі інші користувачі, які підключені до сесії. На цьому, етап встановлення з’єднання можна вважати завершеним.

3.4.2. Інтерфейс відео дзвінка

Інтерфейс дзвінка є класичним, але спрощеним. Елементом керування є тільки нижня частина застосунку. Вона має три основні кнопки. Перша - керування мікрофоном, друга - керування камерою і третя - покинути дзвінок. Зовнішній вигляд зображений на рис. 3.21.

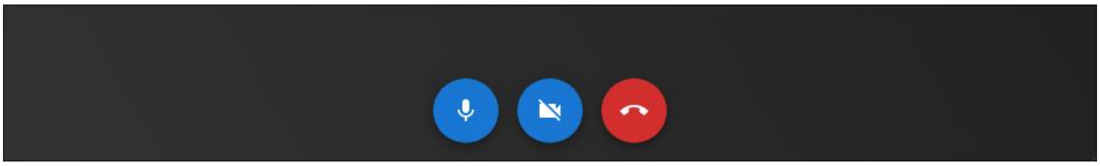


Рис. 3.21. Елементи керування відеодзвінка

Додатковими елементами керування можуть бути:

– Кнопка для відкриття меню з користувачами. Це меню буде відповідати за відображення списку поточних та запрошених учасників. Також в цьому меню може знаходитись кнопка для додавання нових учасників.

– Кнопка для відкриття чату. У відеодзвінку можна підтримувати функцію чату між учасниками. Цей чат буде внутрішнім і повідомлення будуть зберігатись тільки всередині дзвінку.

JSX код панелі зображений на рис. 3.22.

```
export const Footer: React.FC<FooterProps> = (props) => {
  const { isCamOn, isMicOn, handleCam, handleMic, handleLeaveRoom } = props;

  return (
    <Box sx={{ position: "fixed", bottom: 16, width: "100%", zIndex: 1000000 }}>
      <Stack direction="row" justifyContent="space-around">
        <Stack direction="row" spacing={2}>
          <Fab onClick={handleMic} color="primary">
            {isMicOn ? <MicIcon /> : <MicOffIcon />}
          </Fab>

          <Fab onClick={handleCam} color="primary">
            {isCamOn ? <VideocamIcon /> : <VideocamOffIcon />}
          </Fab>

          <Fab onClick={handleLeaveRoom} color="error">
            <CallEndIcon />
          </Fab>
        </Stack>
      </Stack>
    </Box>
  );
};
```

Рис. 3.22. *JSX* код панелі керування

Панель повністю побудована з допомогою готових компонентів *MUI*. Функція "*hanleLeaveRoom*" виконує 2 головні функції - відключення від сесії *OpenTok* та переадресація на домашню сторінку. Ця функція зображена на рис. 3.23.

```
const handleLeaveRoom = () => {
  publisher.current.session.disconnect();
  navigate("/");
};
```

Рис. 3.23. Функція "*handleLeaveRoom*"

Як було описано вище, макет дзвінку змінюється в залежності від кількості користувачів:

– Якщо користувач один, то його камера буде займати весь екран.елементи інтерфейсу не рухаються.

– Якщо користувачів два, то камера поточного користувача буде маленькою, в куті екрану, а учасника на весь екран.

– Якщо учасників більше ніж два, то камери всіх учасників будуть вирівняні по сітці.

Процес змінювання макета відбувається за допомогою *CSS* стилів та лічильнику учасників. Стилі зберігаються в окремому *JS* об'єкті. Вони представлені на рис. 3.24.

```
const connectionsStyles = {
  1: {
    publisher: {
      display: "flex",
      height: "100vh",
      minWidth: "100vw",
    },
  },
  2: {
    publisher: {
      display: "flex",
      position: "fixed",
      height: "100px",
      width: "200px",
      left: "10px",
      top: "10px",
      zIndex: 1000,
    },
```

Рис. 3.24. *CSS* стилі для зміни макета відеодзвінка

Ключі кореневого об'єкта репрезентують кількість учасників. А ключі, внутрішніх об’єктів репрезентують сутність, до якої стилі треба застосовувати - поточний користувач або інші учасники. Зовнішній вигляд макета дзвінку з 2 користувачами представлений на рис. 3.25.

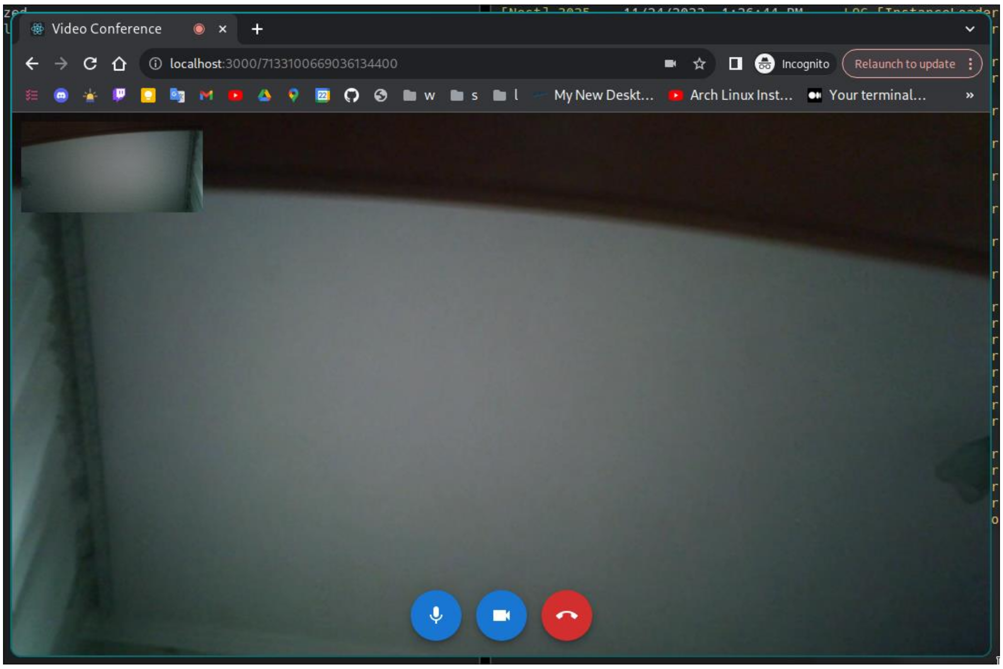


Рис. 3.25. Макет та дизайн дзвінка з 2 учасниками

Висновки за розділом

У третій частині роботи було висвітлено деталізований аналіз файлової архітектури проєкту та важливі етапи розробки ключових компонентів системи, таких як логіка авторизації й аутентифікації, API для взаємодії з кімнатами та відеодзвінок.

Файлова архітектура проєкту має теку "*packages*" у корні, в якій знаходяться частини проєкту, а саме "*client*" і "*server*".

Структура теки "*server*" зроблена бібліотекою *Nest*. Ця бібліотека пропонує свій повноцінний підхід до розробки серверних застосунків і файлова структура також входить в цей підхід. *Nest* додає таке поняття як "модуль", на які розбивається вся система. Це абстрагує певні частини функціонала одну від одної, що робить код більш зрозумілим та читабельним.

Коренева структура теки "*client*" була імплементована утилітою "*create-react-app*". Ця утиліта створює та налаштовує базовий проєкт. Сюди також входить базова файлова структура. На відміну від *Nest*, теку "*src*" розробник заповню так, як він вважає за потрібним.

У процесах авторизації та аутентифікації використовується базовий метод, за допомогою імені користувача, паролю та *JWT* токену. *JWT* токен зберігається в *HTTP cookie* і передається з кожним запитом на сервер. Захист сервера розроблений за допомогою *NestJS*, а саме такого функціонала як "*guards*". У майбутньому, цей функціонал потребуватиме доробки у вигляді ролей. Ролі - це важливий функціонал будь-якого застосунку, оскільки багато яких систем мають певні речі, які можуть використовувати вузьке коло людей. Прикладом такої ролі може бути "*admin*".

Було розроблене початкове *API* для взаємодії з кімнатами. Користувачі можуть створювати нові кімнати та редагувати й видаляти вже створені. *API* виконаний у стилі *REST*, що надає простоти в розумінні та легке керування захистом.

У цьому розділі також була описана детальна взаємодія з *API OpenTok*. Було продемонстровано яким чином створюються сесії та генеруються токени. Також був описаний процес доєднання до створеної сесії.

У підрозділі присвяченому розробці самого відеодзвінку, був проведений детальний опис методу підключення до кімнати та інтерфейс самої кімнати. Також була продемонстрована робота з *OpenTok* для ініціалізації сесії. Був проведений детальний огляд зміни макета в залежності від кількості підключених користувачів.

Головний функціонал цієї системи був повністю розроблений та детально розглянутий. Функціонал системи не має надскладних функцій, але він показує базові процеси які необхідно розробити для побудови системи такого роду.

Сьогодні, комп'ютерні системи для встановлення зв'язку стали невіддільною частиною практично усіх сфер життя, починаючи від корпоративного сектору і закінчуючи повсякденними комунікаційними потребами. Їх роль вирішальна в ефективному функціонуванні бізнесу, наукових досліджень, освіти, медицини та суспільства в цілому.

ВИСНОВКИ

В даній кваліфікаційній роботі, була досліджена та створена комп'ютерна система для комунікації між користувачами. Був проведений детальний розбір списку використаних технологій та аналіз наявних кращих альтернатив. Розроблене рішення було порівняно з іншими системами.

У висновку слід відзначити, що комп'ютерні системи для встановлення зв'язку є кількісною і якісною основою сучасного світу, сприяючи зростанню ефективності, гнучкості та глобальності в усіх сферах життя. Розвиток цих систем безперервно визначається високим темпом технологічного прогресу, включаючи новаторські рішення у галузі безпеки, швидкості передачі даних та масштабованості. Запровадження штучного інтелекту, розширеної реальності та інших передових технологій визначає новий рівень функціональності та можливостей для цих систем. Перспективи їхнього розвитку безмежні, відкриваючи двері для нових досягнень у галузі комунікацій та взаємодії, що допомагає суспільству не лише залишатися з'єднаним, а й розвиватися в напрямку інновацій та сталого прогресу.

Спрямованість на постійний розвиток комп'ютерних систем для встановлення зв'язку свідчить про їхню критичну роль у формуванні майбутнього технологічного ландшафту. Зростання об'ємів обміну даними, підвищення вимог до безпеки та вдосконалення інтерфейсів вимагають постійного удосконалення і адаптації цих систем.

У першій частині роботи, було розглянуте загальне поняття вебсистеми. Були розглянуті технології, які були використані для побудови прототипу системи. Основним стеком технологій прототипу є *HTML, CSS, React, TypeScript, NodeJS, NestJS, PostgreSQL* та *Vonage*. Ці технології є потужним універсальним набором інструментів для побудови будь-якої системи.

React є однією з найпопулярніших бібліотек для побудування користувацьких інтерфейсів, завдяки своїй легкості та гнучкості.

Фреймворк *NestJS* спрощує розробку серверного *API* та пропонує зручну файлову структуру проєкту.

Платформа *Vonage* надає чудовий *API* для побудови аудіо та відео зустрічей.

Також, у першому розділі, були надані певні альтернативи деяким обраним технологіям, наведені базові приклади використання, описані їхні переваги та недоліки.

Головними альтернативами *React* є *Angular* і *Vue*. Вони мають схожі концепти побудови системи, але відрізняються їхньою реалізацією. Монолітну архітектуру застосунку можна замінити мікросервісною, яка надасть проєкту більше надійності, але буде важча у підтримці. *WebRTC* може стати заміною *Vonage API* та надати більше гнучких налаштувань та можливостей, трохи ускладнивши розробку.

У другій частині роботи, були представлені застосунки які пропонують схожий функціонал, а саме *Discord* та *Microsoft Teams*. Ці системи мають мільйони активних користувачів, тому головною метою цього розділу був огляд реалізації певних частин цих систем, для розуміння того, яким чином вони оброблюють таке навантаження. Для застосунку *Discord* був проведений більш детальний аналіз серверної частини, а для *Microsoft Teams* - клієнтської.

Discord використовує доволі складну серверну структуру, яка розбита на різні частини, кожна з яких виконує свою певну роль. Основними серверами є *Discord Gateway*, *Discord Guilds* та *Discord Voice*. Така структура є надійною і забезпечує високий рівень стабільності. Навіть якщо сервер *Discord Voice* не буде активним, користувач все ще зможе отримувати повідомлення та сповіщення.

Microsoft Teams має окремий підхід до клієнтської частини. Розробники системи запропонували метод для обходу одно поточності мови програмування *JavaScript*. Вони представили "*Client Data Layer*", який надав змогу прибрати певний функціонал з головного потоку програми, що відобразилося на її кращій продуктивності та роботі. Також, розділення обов'язків привело до більш оптимізованого, чистого та читабельного коду.

Microsoft Teams та *Discord* є великими та складними застосунками, які надають свої послуги та функції мільйонам користувачам щодня. Розробники цих програм працюють над складними рішеннями, які забезпечують надійність та стабільність фінального продукту.

У третій частині, був описаний загальний процес розробки прототипу системи. Було повністю розглянуто файлову архітектуру проєкту та головні компоненти системи, а саме логіка авторизації й аутентифікації, *API* для взаємодії з кімнатами та сам відеодзвінок.

Система може бути використана як початковий фундамент для побудови більш потужного функціонала. Наступними функціями можуть стати:

– Покращений метод пошуку користувачів для додавання до зустрічі.

– Більша кількість потрібної метаінформації про кімнати та користувачів.

– Календар з можливістю огляду всіх запланованих зустрічей.

– Додавання проміжного лобі, перед підключенням до самої кімнати, з можливістю налаштування камери та мікрофона.

– Внутрішній чат.

– Взаємодія з учасниками під час дзвінку.

Майбутній розвиток буде залежати від того, в якій сфері застосунок буде використовуватись. Сьогодні існує багато різних систем, які розв'язувати задачу відео та аудіо зв’язку. Такий зв’язок надважливий і без нього багато речей зупиниться або з’являться певні труднощі у їхній роботі.

Системи відео та аудіо зв’язку вирішують не тільки проблему дистанційної комунікації, а і задачі, які пов’язані зі сферою використання системи. Наприклад, *Microsoft Teams* пропонує гнучкий віртуальний робочий простір, а *Discord* об’єднує людей зі спільними інтересами та надає зручний повсякденний метод комунікації з друзями та рідними.

У ході виконання даної кваліфікаційної роботи було розв'язано загальну проблему взаємодії користувачів через створення функціональної системи з можливістю створення кімнат для обговорення різних питань. Важливість простоти та легкості використання були враховані під час проєктування, забезпечуючи доступність інструментів навіть для менш досвідчених користувачів.

Хоча побудована система може виглядати простою, важливість забезпечення стабільної роботи та ефективного обміну інформацією в ній не може бути переоцінена. Дана розробка слугує важливим кроком у розвитку засобів електронної комунікації, особливо в контексті співпраці та обміну ідеями в обмеженому просторі віртуальних кімнат.

Хочемо наголосити, що навіть така базова система може мати значний вплив на поліпшення комунікації та співпраці, що, своєю чергою, може призвести до покращення результативності та результатів проєктів.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Комп’ютерна інженерія: методичні рекомендації до виконання дипломних проєктів для студентів освітньо-кваліфікаційного рівня “Бакалавр” напряму

підготовки 6.050102 “Комп’ютерна інженерія” / Уклад.: І.А. Жуков, М.М. Проценко – К.: НАУ, 2015. - 36 с.

2. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Національного авіаційного університету. – К.: НАУ, 2017. 63 с.

3. ДСТУ 2392-94 Інформація та документація. Базові поняття.

4. *How discord handles two and half million concurrent voice users using WebRTC* [Електронний ресурс] - Режим доступу до ресурсу: https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc

5. *Tech Stack of Discord* [Електронний ресурс] - Режим доступу до ресурсу: https://www.linkedin.com/pulse/tech-stack-discord-faysal-ahmed/

6. *Microsoft Teams: Advantages of the new architecture* [Електронний ресурс] - Режим доступу до ресурсу: https://techcommunity.microsoft.com/t5/microsoft-teams-blog/microsoft-teams-advantages-of-the-new-architecture/ba-p/3775704

7. *WebRTC* технологія для безпеки корпоративного спілкування [Електронний ресурс] - Режим доступу до ресурсу: https://corewin.ua/blog/webrtc_nextcloud_talk/

8. Що таке вебсервіс та їх види? [Електронний ресурс] - Режим доступу до ресурсу: https://2ip.ua/ua/blog/web-services

9. Вебтехнології. Їх різновиди та функції [Електронний ресурс] - Режим доступу до ресурсу: https://sites.znu.edu.ua/webprog/lect/1170.ukr.html

10. *SQL* чи *NoSQL* – ось в чому питання [Електронний ресурс] - Режим доступу до ресурсу: https://alternativescience.net/programming/242-sql-chy-nosql-os-v-chomu-pytannya/

11. *WebRTC* та Розробка Веб-Додатків для Реального Часу Комунікації [Електронний ресурс] - Режим доступу до ресурсу: https://it-rating.ua/webrtc-ta-rozrobka-veb-dodatkiv-dlya-realnogo-chasu-komunikatsii

12. *New Microsoft Teams Client Faster with Latest Architecture* [Електронний ресурс] - Режим доступу до ресурсу: https://www.anoopcnair.com/new-microsoft-teams-client-faster

13. *Fluent UI* [Електронний ресурс] - режим доступу до ресурсу: https://developer.microsoft.com/en-us/fluentui#

14. *Introduction to Microsoft Edge WebView2* [Електронний ресурс] - Режим доступу до ресурсу: https://learn.microsoft.com/en-us/microsoft-edge/webview2

15. *What is Bootstrap: A Beginner's Guide* [Електронний ресурс] - Режим доступу до ресурсу: https://careerfoundry.com/en/blog/web-development/what-is-bootstrap-a-beginners-guide

16. *Explain the working of Node.js* [Електронний ресурс] - Режим доступу до ресурсу: https://www.geeksforgeeks.org/explain-the-working-of-node-js

17. *How Does Node.js Work?* [Електронний ресурс] - Режим доступу до ресурсу: https://upstackhq.com/blog/software-development/how-does-node-js-work

18. *The Good and the Bad of Node.js Web App Programming* [Електронний ресурс] - Режим доступу до ресурсу: https://altexsoft.com/blog/the-good-and-the-bad-of-node-js-web-app-development

19. Архітектура програмного забезпечення: все що треба знати [Електронний ресурс] - Режим доступу до ресурсу: https://wezom.com.ua/ua/blog/arhitektura-programmnogo-obespecheniya

20. *HTML: HyperText Markup Language* [Електронний ресурс] - Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/HTML

21. *Web APIs* [Електронний ресурс] - Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API

22. *Elixir and Phoenix can do it all!* [Електронний ресурс] - Режим доступу до ресурсу: https://fly.io/phoenix-files/elixir-and-phoenix-can-do-it-all

23. *Meet Phoenix: A Rails-like Framework for Modern Web Apps on Elixir* [Електронний ресурс] - Режим доступу до ресурсу: https://www.toptal.com/phoenix/phoenix-rails-like-framework-web-apps

24. Мікросервісна архітектура [Електронний ресурс] - Режим доступу до ресурсу: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d