МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ЗАСТОСУНОК ДЛЯ ЕФЕКТИВНОЇ КОМУНІКАЦІЇ ТА ОРГАНІЗАЦІЇ РОБОТИ В КОЛЕКТИВІ

Спеціальність 122 «Комп'ютерні науки»

122 – КРБ – 402.2010319

Виконав студент 4-го курсу, групи 402
_____ ***С. В. Стипаненко***
«21» червня 2024 р.

Керівник: **д-р пед. наук, професор**
_____***О. П. Мєщанінов***
«21» червня 2024 р.

Миколаїв – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет ім. Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

Рівень вищої освіти **бакалавр**
Спеціальність **122 «Комп'ютерні науки»**
(шифр і назва)
Галузь знань **12 «Інформаційні технології»**
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____ Ю. П. Кондратенко
«____» _____2024 р.

З А В Д А Н Н Я
на виконання кваліфікаційної роботи

Видано студенту групи 402 факультету комп'ютерних наук Стипаненку Сергію Валентиновичу.

1. Тема кваліфікаційної роботи «Застосунок для ефективної комунікації та організації роботи в колективі».

Керівник роботи Мєщанінов Олександр Павлович, д-р пед. наук, професор.
Затв. наказом Ректора ЧНУ ім. Петра Могили від «28» грудня 2023 р. № 271

2. Строк представлення кваліфікаційної роботи студентом «21» червня 2024 р.

3. Вхідні (початкові) дані до роботи: аналіз сучасних інструментів для комунікації та організації роботи у колективі; оцінка технологій та критеріїв для визначення оптимальних засобів розробки вебзастосунку.

Очікуваний результат: застосунок для ефективної комунікації та організації роботи у колективі.

4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

– аналіз сучасних додатків для комунікації та управління роботою у колективі;

– розробка інтерфейсу користувача;

– реалізація функціональних можливостей.

5. Перелік графічного матеріалу: таблиці, рисунки, презентація.

6. Завдання до спеціальної частини: «Забезпечення вимог охорони праці у робочому приміщенні та організація безпечної роботи за комп’ютером»

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	Алексєєва А. О., доцент кафедри екології	

Керівник роботи __д-р пед. наук, професор Мєщанінов О. П._____
(наук. ступінь, вчене звання, прізвище та ініціали)

_____
(підпис)

Завдання прийнято до виконання ___Стипаненко С. В._____
(прізвище та ініціали)

_____
(підпис)

Дата видачі завдання «_14_» _____січня____ 2024 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Застосунок для ефективної комунікації та організації роботи в колективі

	Найменування роботи	Початок	Закінчення	Примітки
1	Подання заяви на затвердження теми та керівників КРБ	10.11.2023	15.11.2023	Виконано
2	Отримання завдання на виконання КРБ	10.01.2024	15.01.2024	Виконано
3	Складання календарного плану роботи на весь період виконання КРБ	16.01.2024	30.01.2024	Виконано
4	Отримання завдання на переддипломну практику	15.04.2024	29.04.2024	Виконано
5	Проходження переддипломної практики, збір та аналіз матеріалів до КРБ	29.04.2024	11.05.2024	Виконано
6	Розробка звіту з переддипломної практики	12.05.2024	15.05.2024	Виконано
7	Виконання КРБ: аналіз предметної області, огляд методів та технологій розробки застосунку, розробка ПЗ	13.05.2024	22.06.2024	Виконано
8	Перший попередній захист КРБ на засіданні комісії кафедри	27.05.2024	27.05.2024	Виконано
9	Доробка та остаточне оформлення КРБ	28.05.2024	09.06.2024	Виконано
10	Другий попередній захист КРБ на засіданні комісії кафедри	10.06.2024	10.06.2024	Виконано
11	Подання КРБ рецензенту	19.06.2024	20.06.2024	Виконано
12	Подання КРБ, її електронної копії та інших документів (відгуку, рецензії) до захисту	21.06.2024	21.06.2024	Виконано
13	Захист БКР перед екзаменаційною комісією (ЕК)	28.06.2024	28.06.2024	Виконано

Розробив студент __Стипаненко С. В._____ _____
(прізвище, ім'я, по батькові студента) (*підпис*)

Керівник роботи __Мєщанінов О. П._____ _____
(прізвище, ім'я, по батькові студента) (*підпис*)

«_29_» _____01_____ 2024 р.

АНОТАЦІЯ

кваліфікаційної роботи студента групи 402 ЧНУ ім. Петра Могили

Стипаненка Сергія Валентиновича

Тема: «Застосунок для ефективної комунікації та організації роботи в колективі»

Актуальність дослідження полягає в необхідності забезпечення ефективної комунікації та організації роботи в сучасному колективі. Сучасні виклики професійної діяльності вимагають швидкого і доступного обміну інформацією, а також ефективного управління робочими процесами, що є критичними факторами для досягнення успіху.

Об'єкт дослідження – процес розробки вебзастосунку для комунікації та організації роботи у колективі.

Предмет дослідження – технології та підходи до розробки вебзастосунку, зокрема функціональність, швидкість продукту та зручність користування, необхідні для забезпечення платформи з комунікаційними інструментами.

Мета дослідження – реалізація застосунку, призначеного для ефективної комунікації та організації роботи в колективі з використанням сучасних технологій.

Пояснювальна записка складається зі вступу, трьох розділів та висновків. У першому розділі розглядається аналіз предметної області з описом теоретичних основ розробки вебзастосунків, аналіз існуючих рішень та постановка задачі. Другий розділ присвячено методам та технологіям розробки, включаючи вибір редактора коду, вибір технологій для створення інтерфейсу та реалізації серверної частини, вибір бази даних та огляд технологій для реалізації аутентифікації. У третьому розділі детально описано програмну реалізацію застосунку.

Кваліфікаційна робота містить 109 сторінок, 62 рисунки, 1 таблицю, 25 літературних джерел.

Ключові слова: вебзастосунок, організація роботи, комунікація, React, Next.js, Typescript.

ABSTRACT

of the qualification work of the student of group 402 of the Petro Mohyla Black Sea National University

Stypanenko Serhii

«Application for effective communication and work organization in a team»

The relevance of this research lies in the need to ensure effective communication and organization within a modern team. Contemporary professional challenges require rapid and accessible information exchange, as well as efficient management of work processes, which are critical factors for achieving success.

The object of the research is the process of developing a web application for communication and organization within a team.

The subject of the research is the technologies and approaches to developing a web application, specifically the functionality, product speed, and user convenience necessary to provide a platform with communication tools.

The aim of the research is to implement an application designed for effective communication and organization within a team using modern technologies.

The explanatory note consists of an introduction, three chapters, and conclusions. The first section deals with the analysis of the subject area with a description of the theoretical foundations of web application development, analysis of existing solutions and problem statement. The second section is devoted to development methods and technologies, including the choice of a code editor, the choice of technologies for creating the interface and implementing the server side, the choice of a database, and an overview of technologies for implementing authentication. The third section describes the software implementation of the application in detail.

The qualification work contains 109 pages, 62 images, 1 table, and 25 literature sources.

Keywords: Web application, work organization, communication, React, Next.js, TypeScript.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

CSS – Cascading Style Sheets

DOM – Document Object Model

HTML – HyperText Markup Language

JSON – JavaScript Object Notation

ORM – Object-relational mapping

REST – Representational State Transfer

SASS – Syntactically Awesome Stylesheets

SQL – Structured Query Language

SSG – Static Site Generation

SSR – Server-Side Rendering

XML – EXtensible Markup Language

ВСТУП

Сучасні умови роботи вимагають ефективних засобів комунікації та організації роботи у колективі, особливо в контексті зростаючої кількості віддалених робочих процесів. Це пов'язано з розвитком технологій, які дозволяють працювати з будь-якої точки світу, а також зі змінами в культурі роботи, де фокус з переміщення до організації робочих процесів стає все більш значущим.

Дослідження показують, що ефективна комунікація впливає на результативність роботи команди та загальну задоволеність роботою. Застосунок, який поєднує в собі можливості обміну повідомленнями в реальному часі, голосові та відеовиклики, сприяє зменшенню затримок у спілкуванні та сприяє швидкому прийняттю рішень.

Організації також потребують ефективного інструменту для організації роботи. Управління завданнями, проектами та ресурсами може стати важливим аспектом успішної роботи команди. Застосунок, який включає у себе можливості створення та організації дошок, списків завдань та карток, забезпечує зручний інструмент для планування та виконання роботи.

Застосунок створюється з урахуванням потреб організацій, які активно використовують віддалену роботу та співпрацюють у розподілених командах. Основною метою є створення зручного та ефективного застосунку для комунікації, спільного планування та виконання завдань, що дозволить зменшити час, необхідний для обміну інформацією, підвищити продуктивність та організованість робочих процесів.

Для досягнення поставленої мети необхідно:

1) проаналізувати вже існуючі рішення;
2) обрати засоби та інструменти розробки програмного продукту;
3) здійснити реалізацію застосунку.

Завдання проекту включають:

1) розробку та імплементацію основних функцій, таких як обмін

повідомленнями в реальному часі, можливість створення голосових та відеовикликів, організація дошок, списків та завдань;

2) забезпечення надійності та безпеки застосунку, включаючи аутентифікацію користувачів та управління правами доступу;

3) розробку зручного та інтуїтивно зрозумілого інтерфейсу користувача, який би дозволив легко орієнтуватися в застосунку та швидко виконувати необхідні дії;

4) проведення тестування застосунку з метою забезпечення його стабільної та ефективної роботи.

Цей проект спрямований на створення інноваційного інструменту, який сприятиме покращенню співпраці та продуктивності в командних проектах, незалежно від географічного розташування їх учасників.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Аналіз предметної області є ключовим етапом у будь-якому проєкті, оскільки він дозволяє глибше зрозуміти ринок і потреби користувачів. У випадку розробки застосунку для ефективної комунікації та організації роботи в колективі, проведення аналізу допомагає ідентифікувати популярні рішення на ринку, їх функціональність, а також визначити конкурентні переваги та недоліки. Крім того, вивчення потреб та очікувань користувачів дає змогу визначити, які функції та можливості повинні бути реалізовані у застосунку, щоб відповідати вимогам різних категорій користувачів і забезпечити їх задоволення від використання продукту. Це дозволить зробити проєкт більш ефективним і конкурентоспроможним.

1.1 Опис предметної сфери

Сучасний динамічний світ вимагає від колективів ефективної комунікації та чіткої організації роботи для досягнення високих результатів. Особливо це актуально для команд, що працюють у сфері інформаційних технологій. У сучасному бізнес-середовищі інформаційні технології відіграють ключову роль у забезпеченні конкурентоспроможності та ефективності компаній. Однією з найважливіших складових є вебзастосунки, які надають можливості для покращення комунікації, управління проектами, аналітики даних та багато іншого. Вибір інструментів для комунікації та управління проектами суттєво впливає на продуктивність та ефективність роботи. Тому розробка застосунку для комунікації та організації роботи в колективі є надзвичайно важливим завданням.

Аналізуючи предметну галузь застосунків для ефективної комунікації, можна відзначити їх швидкий розвиток та адаптацію до потреб користувачів у сферах бізнесу, науки та освіти. Вони використовуються для забезпечення взаємодії між співробітниками, організації онлайн-зборів, спільного планування та координації дій у реальному часі. Крім того, застосунки можуть допомагати в

управлінні ресурсами та виконанні завдань, що сприяє підвищенню продуктивності та ефективності роботи команди.

Однією з ключових переваг застосунків для ефективної комунікації є можливість створення віртуальних робочих просторів, де члени команди можуть спілкуватися незалежно від географічного розташування, обмінюватися документами та зберігати історію спілкування. Такі інструменти сприяють підвищенню комунікаційної культури в організації та сприяють зменшенню часу на узгодження планів і рішень.

1.2 Теоретичні основи розробки вебзастосунків

Вебзастосунки стали невід'ємною частиною сучасного інтернет-простору, надаючи користувачам широкий спектр можливостей для взаємодії та співпраці через мережу. Цей розділ надає загальну інформацію про вебзастосунки та розглядає їх переваги.

1.2.1 Загальна інформація

Вебзастосунок – це програмне забезпечення, яке користувачі можуть використовувати через веб-браузер. Вони не потребують встановлення на пристрої користувача і можуть бути доступні з будь-якого пристрою з підключенням до Інтернету. Веб-застосунки використовуються для різних цілей, від спілкування та соціальних мереж до бізнес-інструментів та ігор.

Вебзастосунки можуть бути статичними, динамічними або інтерактивними. Статичні вебзастосунки складаються з фіксованих сторінок, які не змінюються без змін вихідного коду. Динамічні вебзастосунки використовують бази даних або API для генерації контенту на льоту. Інтерактивні вебзастосунки включають в себе можливості взаємодії з користувачем без перезавантаження сторінки, такі як форми, анімація та інше.

Розробка вебзастосунків включає в себе створення клієнтської та серверної частин програми. Клієнтська частина – це фронтенд, який відповідає за

відображення інтерфейсу користувача та взаємодію з ним через веббраузер. Серверна частина – це бекенд, який відповідає за обробку запитів користувачів, доступ до бази даних, бізнес-логіку та інші серверні операції.

1.2.2 Переваги вебзастосунків

1. Кросплатформенність: Вебзастосунки можуть бути запущені на будь-якому пристрої з веб-браузером, незалежно від операційної системи (Windows, macOS, Linux, Android, iOS тощо). Це робить їх універсальними та доступними для широкого кола користувачів.

2. Немає необхідності встановлення: Користувачам не потрібно встановлювати додаткове програмне забезпечення на свої пристрої, оскільки вебзастосунки працюють у веббраузері. Це полегшує розгортання та оновлення застосунків і зменшує витрати на обслуговування.

3. Легка оновлення: Оновлення вебзастосунків зазвичай відбуваються на сервері, що дозволяє автоматично оновлювати всі версії застосунків одночасно. Користувачам не потрібно вручну завантажувати та встановлювати оновлення, це відбувається автоматично під час перезавантаження сторінки.

4. Зручний доступ до даних: Вебзастосунки можуть легко інтегруватися з різними сервісами та базами даних, що дозволяє забезпечити користувачам доступ до їхніх даних з будь-якого пристрою за умови підключення до Інтернету.

5. Складність обробки даних: Більшість обробки даних відбувається на сервері, що дозволяє зменшити навантаження на пристрої користувача і забезпечити більшу швидкість та ефективність роботи застосунків, особливо на мобільних пристроях з обмеженими ресурсами.

6. Широкі можливості розширення: Вебзастосунки мають великий потенціал для розширення функціональності за рахунок використання різних веб-технологій та інтеграції з різними сервісами та API.

Таким чином, вебзастосунки мають велику кількість плюсів за відсутності видимих мінусів, найбільшим і очевидним з яких є неможливість використання

програм за відсутності доступу до Всесвітнього павутиння.

1.3 Аналіз існуючих програмних рішень

У сучасному світі існує велика кількість програмних рішень, спрямованих на поліпшення комунікації та організацію роботи у колективі. Деякі з них вже стали стандартом для багатьох організацій та команд, в той час як інші продовжують активно розвиватися та пропонувати нові функціональні можливості.

1.3.1 Discord

Discord є популярною платформою для текстового, голосового та відеозв'язку, спроектована для спілкування у спільнотах та групах. Вона поєднує в собі можливості текстового чату, групових голосових та відеодзвінків, а також функціонал для спільної гри (див. рис. 1.1).

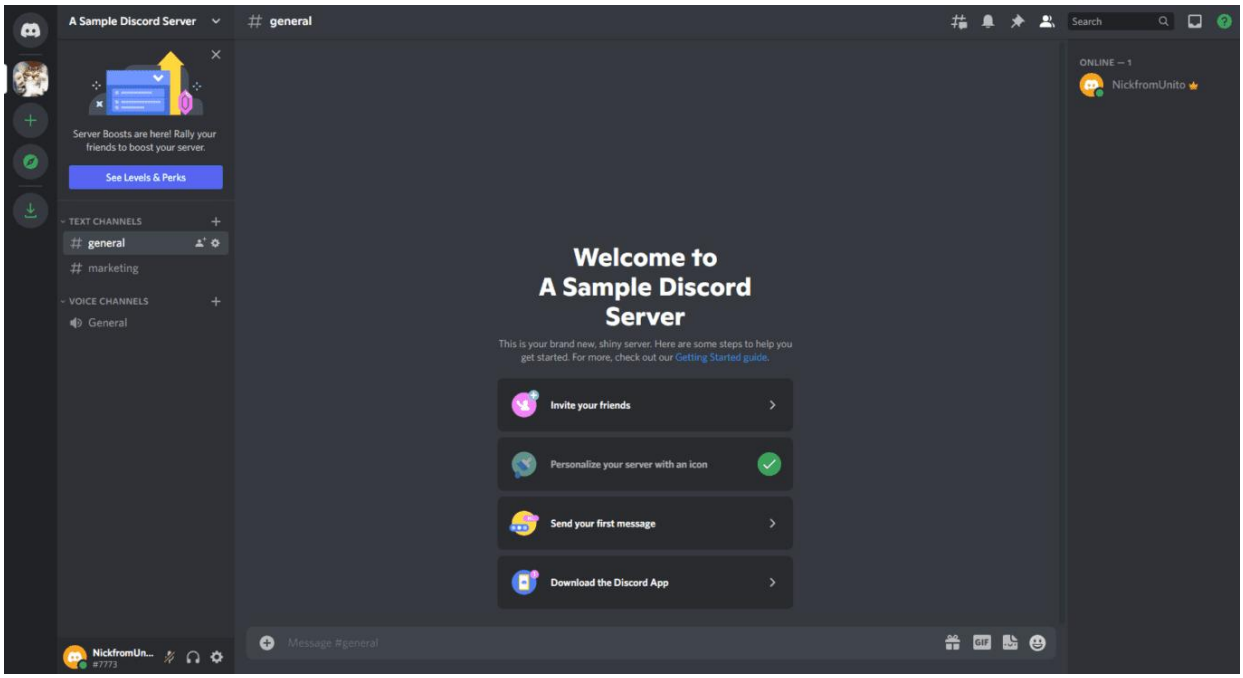

Рисунок 1.1 – Інтерфейс Discord

Функціональні можливості:

1) текстові канали для обговорень та спілкування;
2) голосові та відео-чати, демонстрація екрану під час зустрічей;

3) можливість створення серверів для організації спільнот та проектів;
4) інтеграція з різними ігровими платформами та сервісами.

Серед переваг можна виділити наступні:

1) простий та зрозумілий інтерфейс;
2) широкі можливості для голосового та відео спілкування;
3) велика кількість налаштувань для керування серверами та каналами.

Серед недоліків можна виділити те, що головною метою Discord є геймери, тому корпоративні функції менш розвинені порівняно з іншими рішеннями.

1.3.2 Trello

Trello – це онлайн-інструмент для управління проектами, який базується на системі дошок та карток. Користувачі можуть створювати завдання, призначати їм терміни виконання, додавати коментарі та відстежувати прогрес (див. рис. 1.2).

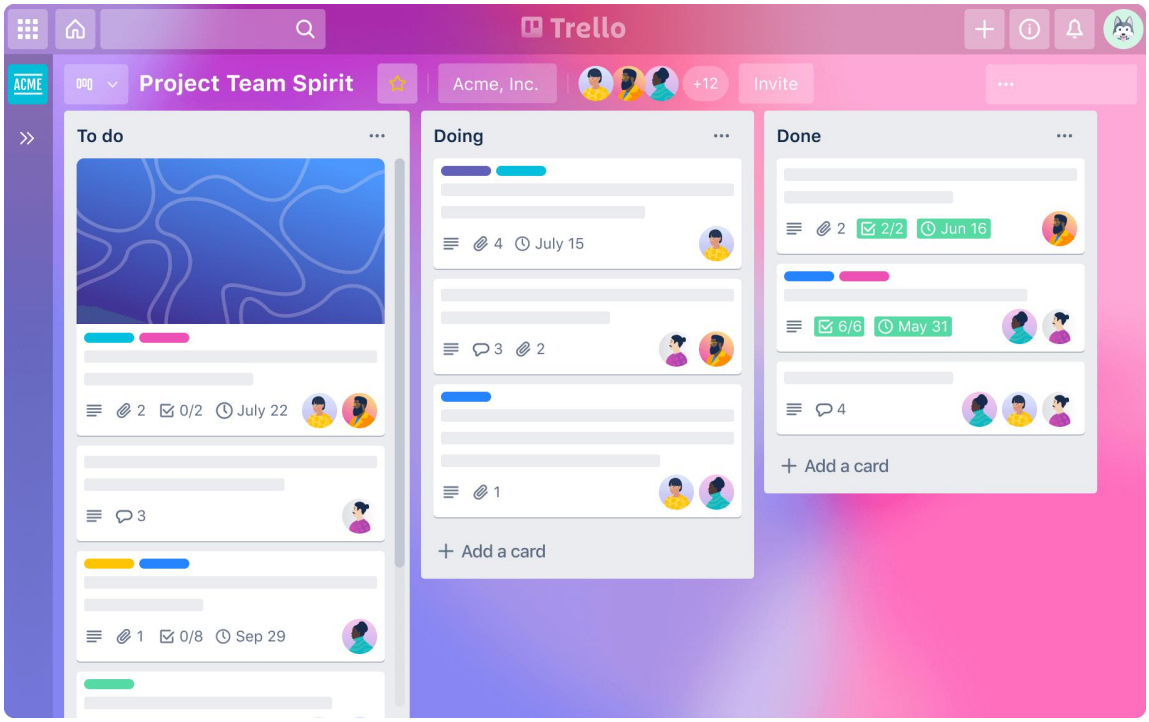

Рисунок 1.2 – Інтерфейс Trello

Функціональні можливості:

1) створення проектів та дошок для організації завдань;
2) додавання завдань у вигляді карточок, які можна переміщувати між

колонками;

3) можливість додавати до карточок коментарі, прикріплювати файли, встановлювати терміни виконання та інші деталі.

Переваги:

1) простий та інтуїтивно зрозумілий інтерфейс;
2) гнучка система організації завдань;
3) можливість розширення функціоналу за допомогою інтеграцій.

Недоліком є обмежена можливість налаштувань у безкоштовній версії.

1.3.3 Microsoft Teams

Microsoft Teams – це комунікаційний та колаборативний інструмент, який поєднує в собі текстову, голосову та відео комунікацію, а також можливості спільної роботи над файлами та завданнями (див. рис. 1.3).

Рисунок 1.3 – Логотип Microsoft Teams

Функціональні можливості:

1) інтеграція з іншими сервісами microsoft, такими як office 365, sharepoint, onedrive тощо;
2) можливість створення команд та каналів для спілкування та спільної роботи;
3) відеоконференції та демонстрація екрану під час зустрічей;
4) спільна робота над документами в режимі реального часу.

Перевагою є глибока інтеграція з іншими продуктами Microsoft, широкий

функціонал для комунікації та спільної роботи. Серед недоліків можна виділити складність використання для користувачів, які не знайомі з екосистемою Microsoft та потреба у використанні інших продуктів Microsoft для повного функціоналу.

1.3.4 Skype

Skype – це популярний сервіс для голосового та відео спілкування, який також має можливості обміну повідомленнями (див. рис. 1.4).

Рисунок 1.4 – Логотип Skype

Функціональні можливості:

1) голосове та відео спілкування в одиночному та груповому режимах;
2) обмін текстовими повідомленнями;
3) можливість відправки вкладень;
4) інтеграція з microsoft-акаунтами;
5) використання на різних платформах: комп'ютери, мобільні пристрої, планшети.

Серед переваг виділяють простоту у використанні, популярність та можливість використання на різних платформах. Недоліками є обмежена функціональність у порівнянні з іншими сервісами і не глибока інтеграція з іншими продуктами.

1.3.5 Zoom

Zoom – це сервіс для відеоконференцій, який також має можливості голосового та текстового спілкування (див. рис. 1.5).

Рисунок 1.5 – Логотип Zoom

Функціональні можливості:

1) голосове, відео та текстове спілкування;
2) відеоконференції з можливістю додавання багатьох учасників;
3) ділитися екраном та файли під час відеоконференцій;
4) реєстрація та запис відеоконференцій;
5) мобільні додатки для спілкування на ходу.

Перевагами є простий та зручний інтерфейс, велика кількість учасників у відеоконференції та можливість запису та архівування зустрічей. Недоліками є деякі обмеження у безкоштовній версії та питання щодо безпеки даних та конфіденційності.

В додатку А представлено порівняльну таблицю всіх вище згаданих програмних рішень.

1.4 Ключові особливості застосунку

Застосунок представляє собою інтерактивну платформу для спілкування та співпраці у колективі. Він поєднує в собі можливості текстової, аудіо та відео комунікації, а також організації робочих процесів через створення, організацію та візуалізацію завдань та проектів.

Ключові особливості:

1) спілкування за допомогою Socket.io: дозволяє користувачам обмінюватися повідомленнями миттєво, забезпечуючи оперативну комунікацію;

2) надсилання вкладень як повідомлень: користувачі можуть надсилати файли як повідомлення для зручності спілкування;

3) редагування та видалення повідомлень в реальному часі для всіх користувачів: забезпечує можливість виправлення помилок та актуалізації інформації безпосередньо під час спілкування;

4) створення каналів для текстових, аудіо та відео дзвінків: забезпечує різноманітні способи комунікації, включаючи голосові та відеодзвінки;

5) спілкування один з одним та відеодзвінки між користувачами: забезпечує можливість приватного спілкування та зв'язку між користувачами;

6) керування учасниками (виключення, зміна ролей);

7) генерація унікальних посилань для запрошення та система запрошень: забезпечує зручний спосіб долучення нових користувачів до колективу;

8) інтерфейс з використанням Tailwind: забезпечує користувачам естетичний та зручний інтерфейс;

9) повна адаптивність та мобільний інтерфейс: гарантує зручний доступ до застосунку з будь-якого пристрою;

10) підтримка Websocket з резервним запасом: забезпечує надійну комунікацію навіть при обмежених умовах мережі;

11) ORM з використанням Prisma та база даних MySQL з використанням платформи Aiven: забезпечує надійне зберігання та управління даними;

12) аутентифікація: забезпечує безпечний доступ до системи для користувачів;

13) робочі простори: дозволяє створювати та керувати різними групами проектів та завдань;

14) створення дошок: надає користувачам засіб для організації завдань та проектів;

15) журнал активності для карток: дозволяє відстежувати історію змін та дій в межах організації;

1.5 Постановка задачі

При постановці задачі необхідно чітко визначити мету проекту та завдання,

які необхідно виконати для досягнення цієї мети. Головною метою нашого проекту є розробка застосунку для ефективної комунікації та організації роботи в колективі. Основною метою додатку є забезпечення зручного, швидкого та безпечного способу комунікації між членами команди, а також ефективного управління завданнями та ресурсами в колективі.

Завдання проекту включають:

1) розробку інтерфейсу застосунку: створення інтуїтивно зрозумілого інтерфейсу з урахуванням сучасних тенденцій дизайну та потреб користувачів, що забезпечить легке освоєння і зручність використання;

2) розробку та імплементацію основних функцій, таких як обмін повідомленнями в реальному часі, можливість створення голосових та відеовикликів, організація дошок, списків та завдань;

3) забезпечення надійності та безпеки застосунку, включаючи аутентифікацію користувачів, управління правами доступу та захист персональних даних;

4) проведення тестування застосунку з метою забезпечення його стабільної та ефективної роботи.

1.5.1 Прогноз майбутнього використання застосунку

Застосунок має можливість стати надійним засобом для організації робочих процесів, незалежно від галузі діяльності. Його функціонал, такий як створення та налаштування серверів, робота з дошками та картками, а також можливості комунікації в реальному часі, робить його універсальним інструментом для вирішення різноманітних завдань.

У бізнес-середовищі застосунок може стати важливим інструментом для організації комунікації та керування завданнями. Він дозволить командам ефективно спілкуватися, відстежувати прогрес проектів, делегувати завдання та виконувати їх у реальному часі. Зокрема, функції створення серверів, керування користувачами та каналами, а також редагування повідомлень можуть бути

використані для покращення комунікації та організації процесів у великих компаніях.

У навчальних установах застосунок може слугувати засобом спілкування між викладачами та студентами, організації уроків та відстеження навчальних досягнень. Можливості відео та аудіодзвінків, а також інструменти для керування завданнями можуть сприяти ефективному навчанню та співпраці в освітньому процесі.

Прогнозується, що з розвитком та розширенням функціоналу застосунка, він стане невід'ємною частиною робочого процесу для багатьох компаній та команд. Застосунок може стати основою для побудови різноманітних інструментів та сервісів, спрямованих на підвищення продуктивності та спрощення спільної роботи.

Висновки до розділу 1

У розділі «Аналіз предметної області» було проведено глибокий огляд потреб та вимог користувачів у сфері застосунків для ефективної комунікації та організації роботи в колективі. Виокремлено основні аспекти та методології розробки таких застосунків, що включають вебзастосунки та їх теоретичні основи.

Було виявлено, що сучасні ринкові умови характеризуються високим рівнем конкуренції серед застосунків для комунікації. Такі популярні інструменти, як Discord, Trello, Microsoft Teams, Skype та Zoom, демонструють різні підходи до організації комунікації та співпраці в колективі. Кожен з них має свої переваги та недоліки, що варто враховувати при розробці нового застосунку.

Аналіз показав, що існуючі рішення відповідають певним потребам ринку, проте є потреба у розвитку нових застосунків, які можуть враховувати зростаючі вимоги до зручності, швидкодії та функціональності. Наприклад, деякі інструменти не завжди задовольняють потреби в високому рівні безпеки або інтеграції з іншими платформами.

У майбутньому використання застосунків для ефективної комунікації та

організації роботи в колективі буде тісно пов'язане з розвитком технологій штучного інтелекту, аналізом даних та збільшенням зручності користування. Прогнозується збільшення попиту на застосунки з високим рівнем персоналізації та можливостями для адаптації до індивідуальних потреб користувачів.

Отже, успішна розробка нових застосунків для ефективної комунікації та організації роботи в колективі передбачає не лише технічну реалізацію, але й глибоке розуміння потреб користувачів та динаміку змін на ринку інформаційних технологій.

2 МЕТОДИ ТА ТЕХНОЛОГІЇ РОЗРОБКИ ЗАСТОСУНКУ

2.1 Вибір редактора коду

Редактор коду може також надавати додаткові функції, такі як вбудований термінал, інтеграція з системами контролю версій, підтримка різних плагінів та інші корисні інструменти. Ці можливості дозволяють програмістам працювати більш ефективно, зменшуючи кількість помилок та прискорюючи процес розробки. Наприклад, функція автозавершення дозволяє швидко вставляти стандартні фрагменти коду, а підсвічування синтаксису допомагає уникнути синтаксичних помилок.

Редактор коду – це програмне забезпечення, що використовується розробниками для написання, редагування та управління кодом програм. Він є критично важливим інструментом у процесі розробки, оскільки дозволяє розробникам ефективно створювати та підтримувати програмний код, а також розуміти його структуру.

Одна з головних переваг використання редактора коду полягає у значному полегшенні процесу розробки. Він забезпечує кольорове виділення синтаксису, що дозволяє швидко орієнтуватися в коді і виявляти синтаксичні помилки. Автоматичне форматування коду покращує читабельність та підтримуваність проекту. Крім того, редактори коду надають можливості для автоматизації багатьох рутинних завдань, що дозволяє розробникам зосередитися на вирішенні складніших проблем.

Редактори коду можуть пропонувати додаткові функції, такі як вбудований термінал для виконання команд безпосередньо з редактора, інтеграція з системами контролю версій (наприклад, Git), підтримка різноманітних плагінів і розширень, які додають нові можливості та інструменти для розробки. Ці функції роблять редактор коду універсальним інструментом, який задовольняє потреби як початківців, так і досвідчених розробників.

2.1.1 Visual Studio Code

Для розробки застосунку був обраний редактор коду Visual Studio Code. Visual Studio Code – засіб для створення, редагування та зневадження сучасних вебзастосунків і програм для хмарних систем. Visual Studio Code є безкоштовним та відкритим для використання. Це означає, що розробники можуть використовувати його безкоштовно та навіть вносити свої власні зміни для вдосконалення редактора (див. рис. 2.1).

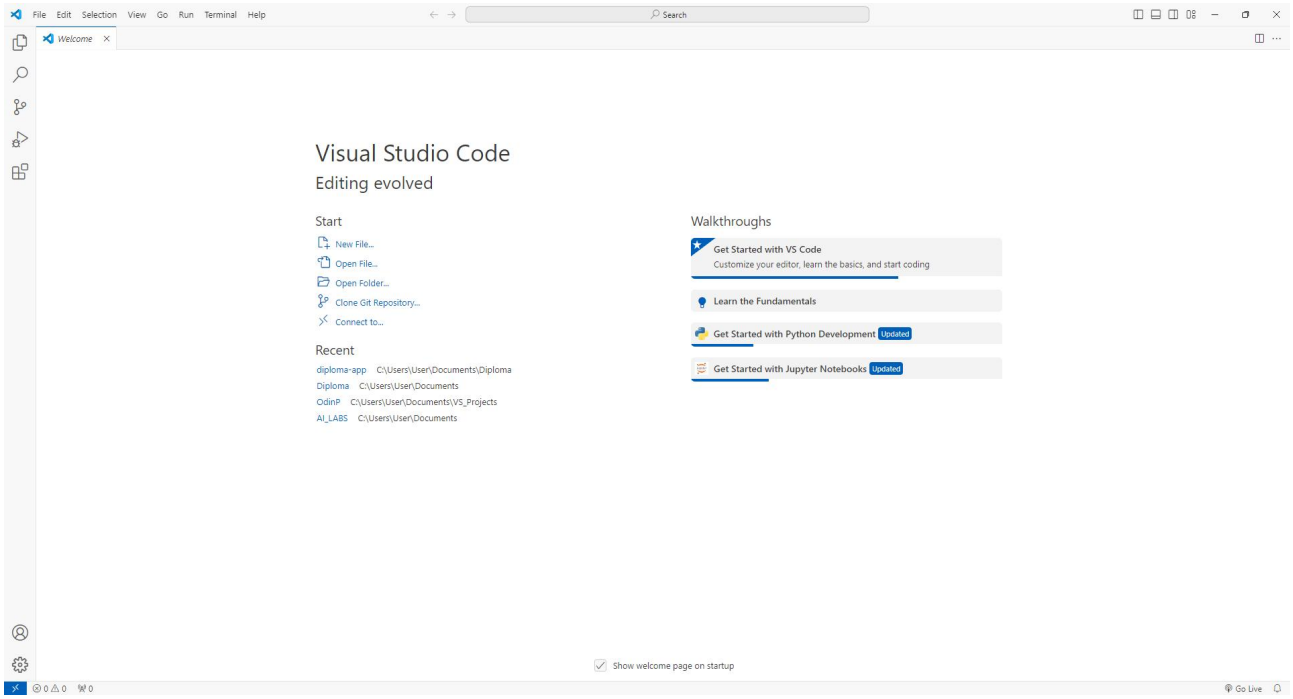

Рисунок 2.1 – Інтерфейс Visual Studio Code

Visual Studio Code має інтуїтивно зрозумілий інтерфейс користувача, що дозволяє легко орієнтуватися у редакторі навіть початківцям. Він має зручну систему навігації, можливість швидкого пошуку файлів та функцій, а також підтримує різні теми оформлення, що дозволяє налаштувати вигляд редактора відповідно до власних вподобань.

Редактор підтримує розробку для платформ ASP.NET і Node.js, і позиціюється як легковагове рішення, що дозволяє обійтися без повного інтегрованого середовища розробки. Серед підтримуваних мов і технологій: JavaScript, C++, C#, TypeScript, jade, PHP, Python, XML, Batch, F#, DockerFile,

Coffee Script, Java, HandleBars, R, Objective-C, PowerShell, Luna, Visual Basic, Markdown, JSON, HTML, CSS, LESS і SASS, Haxe.

Також Visual Studio Code є високо розширюваним редактором, що дозволяє використовувати різноманітні розширення для покращення його функціоналу. Це дозволяє розробникам налаштовувати редактор відповідно до їх потреб та використовувати різні інструменти та сервіси безпосередньо в середовищі Visual Studio Code.

Visual Studio Code також забезпечує кросплатформеність, що дозволяє використовувати його на різних операційних системах, таких як Windows, macOS та Linux. Вбудований термінал дозволяє виконувати командний рядок безпосередньо з редактора, що спрощує робочий процес і робить його більш безперервним. Підтримка IntelliSense забезпечує інтелектуальне автозавершення коду, яка робить написання коду швидшим та зручнішим.

Завдяки усім цим функціям, Visual Studio Code стає незамінним інструментом для розробників будь-якого рівня, від початківців до професіоналів. Його гнучкість, потужність і зручність у використанні сприяють підвищенню продуктивності та якості написаного коду, що є надзвичайно важливим у сучасній розробці програмного забезпечення.

2.2 Інтерфейс застосунку

Інтерфейс вебзастосунків є ключовою складовою їхньої успішності та користувацького досвіду. Це візуальна частина застосунку, яка взаємодіє з користувачем через різноманітні елементи, такі як кнопки, поля введення, таблиці, меню та інші компоненти. Головною метою інтерфейсу є забезпечення зручного та ефективного способу взаємодії користувача з функціональністю застосунку.

Інтерфейс вебзастосунків має бути інтуїтивно зрозумілим та легким у використанні, щоб користувачі могли швидко орієнтуватися та використовувати всі доступні функції без особливих труднощів. Це досягається за допомогою правильного розташування елементів на сторінці, зрозумілого навігаційного меню,

консистентного дизайну та оптимального використання кольорів і шрифтів.

Для досягнення високоякісного інтерфейсу використовуються сучасні технології та методи дизайну, такі як зрозумілий дизайн, що забезпечує адаптацію інтерфейсу до різних пристроїв та розмірів екранів, анімації для покращення взаємодії користувача з застосунком та зручний вибір кольорової палітри для покращення сприйняття інформації.

Успішний інтерфейс вебзастосунків не лише сприяє збільшенню задоволеності користувачів, але й впливає на їхню продуктивність та відчуття комфорту під час використання застосунку. Таким чином, розробка інтерфейсу є важливим етапом у створенні успішних вебзастосунків, який вимагає уваги до деталей, тестування та постійного вдосконалення з метою досягнення найвищих стандартів якості і зручності для кінцевих користувачів.

Розробниками широко використовуються фреймворки з метою спрощення та прискорення процесу створення вебзастосунків. Вони дозволяють підтримувати високий рівень якості інтерфейсу, забезпечуючи швидкість розробки і стандартизацію в проекті.

Фреймворки, такі як React, Vue.js, Angular і Svelte, відомі своєю популярністю серед розробників завдяки своїм універсальним можливостям та широкій підтримці спільноти. Вони дозволяють створювати не лише статичні сторінки, а й динамічні інтерфейси з багатьма взаємодіючими елементами.

При виборі фреймворку для побудови інтерфейсу вебзастосунку важливо враховувати вимоги проекту, рівень зручності використання та підтримку спільноти розробників. Кожен фреймворк має свої особливості та переваги, які можуть відповідати різним потребам та завданням веброзробки. Вони надають розробникам інструменти та компоненти, які дозволяють створювати ефективні, красиві та інтерактивні користувацькі інтерфейси без значного напруження у розробці з нуля.

Серед фреймворки найпопулярніші наступні:

1) React: бібліотека JavaScript для створення інтерактивних інтерфейсів.

React використовує компонентний підхід, що дозволяє легко структурувати інтерфейс та забезпечувати його повторне використання;

2) Angular: фреймворк JavaScript, який пропонує повноцінну платформу для розробки вебзастосунків. Він включає в себе велику кількість інструментів для розробки, таких як модулі, сервіси, компоненти та інші;

3) Vue.js: прогресивний JavaScript-фреймворк для створення користувацьких інтерфейсів. Vue.js пропонує простоту використання разом із сильними можливостями для розширення.

Кожен з цих фреймворків має свої особливості та підходи до розробки, що дозволяє розробникам вибирати той, який найкращим чином відповідає потребам їхнього проекту. Використання таких інструментів дозволяє зосередитися на створенні відмінного користувацького досвіду та забезпеченні ефективної взаємодії з користувачем.

Основні функції та переваги фреймворків для побудови інтерфейсу включають:

1) компонентний підхід: фреймворки дозволяють розбити інтерфейс на невеликі, самостійні компоненти, які можна перевикористовувати та комбінувати для створення складних інтерфейсів;

2) лекларативний синтаксис: вони часто використовують декларативний підхід до опису інтерфейсу, що спрощує розуміння та модифікацію коду;

3) шаблонізація: фреймворки часто надають можливості для шаблонізації, що спрощує створення структурних шаблонів інтерфейсу;

4) підтримка мобільних платформ: багато фреймворків надають інструменти для створення адаптивних і мобільних інтерфейсів, що підтримують різні розміри екранів та пристрої.

2.2.1 React

Для побудови інтерфейсу користувача було обрано фреймворк React. React – це відкрита JavaScript бібліотека з кодом для розробки інтерфейсів користувача.

Вона розроблена компанією Facebook і використовується використовується для побудови великих масштабованих вебзастосунків з використанням компонентного підходу.

Основні принципи React включають в себе компонентний підхід, віртуальний DOM (Document Object Model), односторінкову архітектуру, що сприяє швидкодії та ефективності, а також можливість використання JSX – розширення JavaScript, яке дозволяє використовувати HTML-подібний синтаксис для опису структури інтерфейсу.

Одним із ключових переваг React є його компонентний підхід. Кожен елемент інтерфейсу у React може бути представлений як компонент, який зазвичай містить у собі відображення (UI) та логіку. Це дозволяє створювати більш складні інтерфейси, розділяючи їх на невеликі, повторно використовувані компоненти.

Віртуальний DOM – це концепція, яка дозволяє React оптимізувати оновлення елементів інтерфейсу шляхом маніпулювання віртуальним представленням DOM, замість безпосередньої маніпуляції реального DOM. Це дозволяє підвищити продуктивність застосунку, оскільки оновлення відбуваються тільки у випадках, коли вони є необхідними.

React також підтримує розширення JSX, яке дозволяє використовувати HTML-подібний синтаксис безпосередньо в JavaScript коді. Це полегшує створення та розуміння структури інтерфейсу, а також сприяє підвищенню швидкості розробки.

2.2.2 Tailwind

Tailwind – це інструмент для створення веб-інтерфейсів, який базується на концепції «utility-first» CSS. Замість написання власних CSS-стилів, як це робиться в традиційному підході, у Tailwind використовуються готові класи, які надають необхідний вигляд та функціональність елементам інтерфейсу. Основні переваги Tailwind включають прискорення процесу розробки, спрощення спільної

роботи над проектом, а також покращення структури та підтримки коду.

Tailwind надає широкий набір вбудованих класів, що охоплюють різноманітні аспекти веб-дизайну, від типографії та розмітки до кольорів та просторових відносин. Ці класи можуть бути використані для швидкого створення різноманітних компонентів та макетів без необхідності написання CSS-стилів з нуля. Крім того, Tailwind дозволяє легко налаштовувати вигляд та поведінку елементів інтерфейсу за допомогою конфігураційного файлу, що спрощує інтеграцію з іншими інструментами та бібліотеками.

Однією з ключових особливостей Tailwind є його гнучкість та розширюваність. Крім вбудованих класів, користувачі можуть створювати власні utility-класи для вирішення специфічних завдань в їхньому проекті. Це дозволяє підтримувати чистоту коду та забезпечує більшу гнучкість у використанні бібліотеки.

Однією з переваг Tailwind є його спосіб структурування CSS, що сприяє покращенню читабельності та обслуговуваності коду. Використання готових класів забезпечує однорідність у стилізації елементів інтерфейсу, що допомагає уникнути дублювання коду та спрощує роботу з CSS.

2.2.3 Shadcn/ui

ShadcnUI – це набір готових компонентів і стилів, який можна використовувати для швидкої розробки сучасного та стильного інтерфейсу вебзастосунків. Цей набір розроблений з урахуванням сучасних тенденцій у дизайні, що дозволяє створювати привабливі і ергономічні користувацькі інтерфейси.

Основні особливості ShadcnUI включають:

1) готові компоненти: Набір містить різноманітні готові компоненти, такі як кнопки, форми, картки, навігаційні панелі, які можна легко інтегрувати в проект;

2) стилізація і теми: ShadcnUI має вбудовану підтримку для налаштування

тем і стилів, що дозволяє з легкістю адаптувати вигляд компонентів до корпоративного стилю або індивідуальних вимог проекту;

3) адаптивний дизайн: компоненти ShadcnUI розроблені з урахуванням принципів адаптивного дизайну, що дозволяє їм пристосовуватися до різних розмірів екранів і пристроїв;

4) простота використання: інтерфейс ShadcnUI створений з акцентом на зручність використання. Він має чітку документацію і простий API, що спрощує процес інтеграції та налаштування компонентів;

5) сумісність з іншими технологіями: ShadcnUI може інтегруватися з різними фреймворками, такими як React, Vue.js, Angular та іншими, що робить його універсальним вибором для різноманітних проектів.

ShadcnUI дозволяє розробникам швидко створювати сучасні та інтуїтивно зрозумілі інтерфейси з мінімальними зусиллями, що робить його цінним інструментом для будь-якого проекту веброзробки.

2.3 Серверна частина застосунку

Серверна частина вебзастосунку відповідає за обробку запитів від клієнтської сторони і забезпечення необхідної логіки для коректної роботи програми. Це сервер, який працює в хмарному середовищі, віддаленої машині або в локальній мережі, і виконує важливі завдання, такі як зберігання даних, обробка бізнес-логіки, автентифікація та авторизація користувачів, інтеграція з іншими сервісами та технологіями.

Основні функції серверної частини:

1) обробка запитів: серверна частина отримує запити від клієнтської частини (наприклад, через HTTP або WebSocket протоколи), обробляє їх і повертає відповіді. Це може включати авторизацію, обробку бізнес-логіки та взаємодію з базою даних;

2) управління даними: серверна частина відповідає за зберігання, оновлення та видалення даних у базі даних або інших сховищах, що

використовуються додатком. Вона забезпечує консистентність та цілісність даних, а також їх захист від несанкціонованого доступу;

3) взаємодія з іншими сервісами: багато сучасних вебзастосунків взаємодіють з зовнішніми сервісами, такими як платіжні системи, сервіси аутентифікації, сервіси для роботи з медіафайлами тощо. Серверна частина відповідає за інтеграцію з такими сервісами та обробку їхніх відповідей;

4) безпека: одним з важливих аспектів серверної частини є забезпечення безпеки додатку. Це включає в себе захист від атак, обробку валідації даних, контроль доступу до ресурсів і застосування найкращих практик забезпечення даних;

5) масштабованість: серверна частина повинна бути готовою до масштабування, щоб забезпечити високу доступність і продуктивність додатку при збільшенні обсягів обробки даних і кількості користувачів.

2.3.1 Next.js

Next.js, універсальний фреймворк, який здобув значну популярність у спільноті веброзробників, знаходиться на перетині фронтенд та бекенд розробки. Цей фреймворк, розроблений компанією Vercel, поєднує можливості обох частин вебзастосунків, забезпечуючи комплексне рішення для створення сучасних вебдодатків.

Next.js – це фреймворк на основі React, створений для більш ефективної та гнучкої розробки вебзастосунків. Він розширює можливості React, вводячи такі функції, як серверна рендеринг (SSR) та статична генерація сайтів (SSG), що робить його потужним інструментом у арсеналі розробника.

Основні переваги Next.js включають:

1) статичний та серверний рендеринг: Next.js підтримує як статичний, так і серверний рендеринг, що дозволяє оптимізувати швидкодію застосунку та покращити його індексацію пошуковими системами;

2) клієнтська та серверна частина: Next.js дозволяє легко розділити код між

клієнтською та серверною частинами застосунку, що забезпечує високу продуктивність та масштабованість;

3) маршрутизація на основі файлів: спрощена маршрутизація на основі файлової системи, що полегшує управління сторінками та посиланнями;

4) автоматична оптимізація зображень: вбудована підтримка оптимізації зображень дозволяє автоматично стискати та оптимізувати зображення для швидкості завантаження вебсторінок;

5) API маршрути: Дозволяє створювати API ендпоінти в межах проекту Next.js, що забезпечує безперебійну роботу з даними і функціональністю бекенду;

6) автоматичне розділення коду: оптимізує продуктивність, завантажуючи тільки необхідний JavaScript для поточної сторінки;

7) розширені можливості маршрутизації: Next.js надає розширені можливості маршрутизації, включаючи динамічні маршрути та передачу параметрів;

8) HMR (hot module replacement): Next.js підтримує HMR, що дозволяє автоматично оновлювати зміни в коді без перезавантаження сторінки в режимі розробки;

9) розширена підтримка CSS: Next.js має розширену підтримку css, включаючи можливість використання CSS модулів та популярних препроцесорів CSS, таких як Sass та Less;

10) статичний експорт: можливість статичного експорту дозволяє генерувати статичні файли вебсайту, що можна розгорнути на будь-якому хостингу, що підтримує статичні файли;

11) підтримка TypeScript: Next.js надає підтримку TypeScript, що дозволяє розробникам використовувати сучасні інструменти для типізації коду та покращення його надійності.

Next.js відзначається у фронтенд розробці завдяки своїм оптимізованим методам рендерингу та легким у використанні функціям. SSR динамічно генерує сторінки для кожного запиту, забезпечуючи актуальний контент і переваги для

SEO. SSG попередньо рендерить сторінки під час збірки, що значно покращує час завантаження та продуктивність статичного контенту.

Створення інтерфейсів за допомогою Next.js спрощено завдяки інтеграції з React. Розробники можуть використовувати модель компонентів та управління станом React у поєднанні з розширеннями Next.js для ефективного створення інтерактивних та динамічних інтерфейсів.

Next.js також пропонує потужні клієнтські функції, такі як маршрутизація та динамічне рендеринг сторінок на клієнтській стороні, що забезпечує плавний користувацький досвід, подібний до односторінкових застосунків (SPA). Він ефективно керує клієнтським JavaScript, забезпечуючи швидкі взаємодії та переходи між сторінками.

Бекенд можливості Next.js часто залишаються у тіні його фронтенд особливостей, але він також добре справляється з серверною логікою. Next.js дозволяє створювати API маршрути в межах проекту, що дозволяє розробникам будувати бекенд функціональність, таку як отримання даних, аутентифікація та інше. Ці API маршрути є безсерверними функціями, що робить їх масштабованими та ефективними.

Отримання та обробка даних на сервері є невід'ємною частиною Next.js. Це дозволяє розробникам отримувати дані на серверному рівні, обробляти їх та рендерити разом з React компонентами, забезпечуючи безперебійну інтеграцію бекенд та фронтенд операцій.

Next.js відомий своєю здатністю легко інтегруватися з різними базами даних та зовнішніми API. Це дозволяє безперешкодно підключатися до різних баз даних, таких як SQL (наприклад, PostgreSQL) або NoSQL (наприклад, MongoDB), безпосередньо в API маршрутах Next.js.

Next.js часто розглядається як fullstack фреймворк, здатний обробляти як клієнтські, так і серверні операції. Ця унікальна риса полягає в його здатності обслуговувати статичні сторінки, генерувати серверні рендерингу сторінки та керувати API маршрутами, все це в межах одного фреймворку. Повностекова

природа Next.js означає, що розробники можуть створювати повноцінний вебзастосунок – від інтерфейсу користувача до серверної логіки та управління базою даних – без необхідності використовувати додаткові фреймворки або технології.

Однією з ключових переваг Next.js є його здатність об'єднувати фронтенд та бекенд розробку в єдиний робочий процес. Це досягається завдяки його директорії сторінок, де кожен файл стає маршрутом, та функції API маршрутів, яка дозволяє розробникам писати серверний код безпосередньо в додатку Next.js. Ця інтеграція спрощує процес розробки та знижує складність, що зазвичай пов'язана з розділенням фронтенд та бекенд завдань.

Реальні приклади використання Next.js у full stack розробці різноманітні та широкі. E-commerce платформи виграють від його SEO-дружніх можливостей рендерингу та високої продуктивності, тоді як контент-орієнтовані сайти використовують його функції статичної генерації для оптимальних часів завантаження. Next.js також використовується в корпоративних додатках, де його масштабованість та надійність є вирішальними.

2.3.2 Socket.io

Socket.io є бібліотекою JavaScript для реалізації вебсокетів, які забезпечують двосторонню зв'язок між клієнтом і сервером у реальному часі. Це дозволяє створювати інтерактивні вебзастосунки, які можуть надсилати та отримувати дані без перезавантаження сторінки. Однією з ключових переваг Socket.io є підтримка різних транспортних протоколів, таких як WebSocket, AJAX та інші, що дозволяє автоматично вибирати найефективніший спосіб зв'язку для кожного клієнта.

Основні характеристики Socket.IO включають:

1) реалізація вебсокетів: Socket.IO дозволяє легко створювати та управляти веб-сокетами, що забезпечує надійний двосторонній зв'язок між клієнтом і сервером;

2) підтримка різних транспортних протоколів: Під час встановлення

з'єднання Socket.IO спробує використати найбільш ефективний транспортний протокол, такий як WebSocket, а в разі його недоступності автоматично впаде у покращений спосіб передачі даних, такий як HTTP long-polling або AJAX;

3) легке використання: API Socket.IO простий у використанні і дозволяє легко створювати події та обробники для обміну даними між клієнтом і сервером;

4) підтримка різних типів даних: Socket.IO дозволяє передавати різні типи даних, такі як текст, JSON, бінарні дані тощо;

5) автоматичне перепідключення: Socket.IO автоматично намагається встановити з'єднання в разі втрати зв'язку або відмови сервера, що забезпечує стабільну роботу застосунку навіть при нестабільних умовах мережі.

Socket.IO є потужним інструментом для створення вебзастосунків, які вимагають реального часу обміну даними, таких як чати, графіки в реальному часі, співпрацючі редактори тощо. Його використання дозволяє створювати ефективні та масштабовані застосунки з великою швидкістю та надійністю.

2.3.3 Typescript

TypeScript, створений компанією Microsoft, є строго типізованим надмножиною JavaScript, яке компілюється в чистий JavaScript. TypeScript додає типізацію до мови JavaScript, що дозволяє розробникам визначати типи для змінних, функцій, об'єктів та іншого. Це забезпечує кілька суттєвих переваг для розробки серверної частини веб-додатків.

По-перше, статична типізація дозволяє виявляти помилки ще на етапі розробки, до виконання коду. Це зменшує кількість помилок, які можуть виникнути під час виконання, що сприяє більш стабільному і передбачуваному коду. Наприклад, TypeScript вказує на невідповідності типів, відсутні властивості в об'єктах та інші помилки, які можуть залишитися непоміченими в JavaScript.

По-друге, TypeScript покращує автодоповнення та навігацію коду в редакторах коду, таких як Visual Studio Code. Завдяки чіткій типізації, редактори можуть надавати більш точні підказки та рекомендації, що значно полегшує

процес написання коду, особливо у великих проєктах з багатьма залежностями та складними структурами.

Крім того, TypeScript підтримує сучасні функції JavaScript, зокрема асинхронні операції, класи, модулі та інші, що робить його сумісним з будь-якою JavaScript-бібліотекою або фреймворком. Це означає, що розробники можуть використовувати всі переваги нових можливостей JavaScript, одночасно маючи підтримку типів і додаткових перевірок.

У контексті серверної частини, TypeScript часто використовується разом з фреймворками, такими як Node.js та Next.js. Завдяки своїй підтримці модулів CommonJS та ECMAScript, TypeScript легко інтегрується з існуючими проектами Node.js, забезпечуючи при цьому більш безпечний і надійний код. У Next.js, TypeScript допомагає визначати типи для пропсів компонентів, сторінок, API маршрутів та інших частин додатка, що покращує взаємодію між фронтенд і бекенд частинами проекту.

Також важливо відзначити, що TypeScript активно розвивається і підтримується спільнотою, що забезпечує регулярні оновлення та нові можливості. Це робить його надійним вибором для довгострокових проектів, де стабільність і підтримка коду мають велике значення.

Впровадження TypeScript у серверну частину веб-додатка приносить численні переваги, включаючи покращену якість коду, зменшення кількості помилок на етапі розробки, полегшення налагодження та підтримки, а також кращу інтеграцію з сучасними фреймворками і бібліотеками. Завдяки цьому TypeScript стає все більш популярним серед розробників, які прагнуть створювати надійні і масштабовані веб-додатки.

2.4 Вибір бази даних

Вибір бази даних є одним з ключових рішень при розробці веб-застосунків, оскільки база даних відповідає за зберігання, організацію та управління даними, що використовуються застосунком. Існує кілька різних типів баз даних, кожен з

яких має свої переваги та недоліки в залежності від конкретних вимог проекту.

Реляційні бази даних (SQL) є одними з найбільш поширених і використовуються для структурованого зберігання даних у вигляді таблиць. Найпопулярніші реляційні бази даних включають MySQL, PostgreSQL, SQLite, Oracle та Microsoft SQL Server. Вони добре підходять для проектів, де дані мають чітку структуру і підтримують складні запити та транзакції. Реляційні бази даних використовують мову SQL (Structured Query Language) для управління та запиту даних.

NoSQL бази даних стали популярними завдяки їх гнучкості та здатності масштабуватися. Вони не використовують традиційні таблиці і можуть зберігати дані у вигляді документів, графів, ключ-значення або широких колонок. Прикладами NoSQL баз даних є MongoDB, Cassandra, Redis та Amazon DynamoDB. Вони особливо корисні для проектів з неструктурованими даними або з потребою у високій продуктивності та масштабованості.

MySQL – це популярна реляційна база даних, яка є відкритим програмним забезпеченням і підтримується великою спільнотою. Вона широко використовується в веб-розробці завдяки своїй надійності, простоті у використанні та підтримці складних запитів. MySQL добре інтегрується з різними фреймворками та мовами програмування, такими як PHP, Python, Java і Node.js.

PostgreSQL є ще однією потужною реляційною базою даних з відкритим вихідним кодом, яка славиться своєю надійністю та розширюваністю. Вона підтримує складні типи даних, розширені запити, транзакції та зберігання процедур. PostgreSQL відома своєю відповідністю стандартам SQL та широким набором функцій, що робить її ідеальним вибором для складних та масштабних додатків.

MongoDB – це документо-орієнтована NoSQL база даних, яка зберігає дані у форматі BSON (бінарне представлення JSON). Вона забезпечує високу продуктивність, масштабованість та гнучкість, що дозволяє швидко розробляти і розширювати додатки. MongoDB часто використовується в проектах, де

структура даних може змінюватися, або де важлива висока швидкість обробки даних.

Вибір бази даних залежить від конкретних вимог проекту, таких як тип даних, обсяг даних, необхідна продуктивність, масштабованість та підтримка транзакцій. Для більшості веб-застосунків реляційні бази даних є надійним вибором завдяки їх структурованій природі та підтримці складних запитів. Проте NoSQL бази даних можуть бути кращими для проектів з великими обсягами неструктурованих даних або де потрібно забезпечити високу масштабованість та продуктивність.

2.4.1 MySQL

Для зберігання даних застосунку використовується база даних MySQL, розгорнута на платформі Aiven. MySQL – це відкрита реляційна система керування базами даних (RDBMS), яка базується на мові SQL (Structured Query Language). Вона використовується для зберігання та управління даними вебзастосунків, забезпечуючи швидкий доступ до інформації, надійність та ефективність роботи застосунків.

MySQL має багато переваг, які роблять її популярним вибором для розробки вебзастосунків. Вона відома своєю високою продуктивністю, масштабованістю та надійністю. MySQL підтримує велику кількість операцій з базами даних, таких як додавання, видалення, оновлення та пошук даних, що робить її ідеальним рішенням для розробки вебзастосунків з великим обсягом даних та високою навантаженістю.

Однією з ключових переваг MySQL є його висока продуктивність. Вона забезпечує швидкий доступ до даних завдяки своїй оптимізованій архітектурі та ефективному використанню ресурсів сервера. MySQL також має широкий набір функцій, таких як індексація даних, оптимізація запитів та кешування, які дозволяють підтримувати високу продуктивність навіть при роботі з великим обсягом даних.

Ще однією перевагою MySQL є його масштабованість. Він може легко масштабуватися вгору або вниз в залежності від потреб користувачів, дозволяючи підтримувати ефективність та продуктивність вебзастосунків навіть при збільшенні обсягу даних або кількості користувачів.

Крім того, MySQL відомий своєю надійністю. Вона має вбудовані механізми резервного копіювання та відновлення даних, які забезпечують захист від втрати даних в разі виникнення непередбачених ситуацій. MySQL також підтримує реплікацію даних, що дозволяє створювати дублікати даних для забезпечення високої доступності та надійності системи.

2.4.2 Prisma

Prisma є сучасним ORM (Object-Relational Mapping) інструментом для TypeScript і JavaScript, який значно спрощує взаємодію з базами даних. Він надає розробникам зручний спосіб описувати та керувати базами даних за допомогою зручного і зрозумілого синтаксису. Основні переваги Prisma полягають у його простоті використання, швидкості розробки та забезпеченні безпеки та надійності роботи з базою даних.

Основні функціональні можливості Prisma наведені нижче.

1. Декларативні моделі даних: Розробники можуть визначати моделі даних у вигляді звичайних класів або інтерфейсів TypeScript, що спрощує процес створення та управління схемою бази даних.

2. Автоматичне мігрування схеми: Prisma автоматично генерує та застосовує міграції схеми бази даних на основі змін у моделях даних, що дозволяє зосередитися на розробці функціоналу замість управління структурою бази даних.

3. Запити з виразною типізацією: Запити до бази даних виконуються за допомогою Prisma Client, який надає виразну типізацію та автодоповнення, що спрощує взаємодію з даними та уникнення помилок.

4. Зручне управління зв'язками між об'єктами: Prisma надає зручні засоби для визначення та управління зв'язками між об'єктами в базі даних, включаючи

один-до-одного, один-до-багатьох та багато-до-багатьох зв'язки.

5. Підтримка різних баз даних: Prisma підтримує різні типи баз даних, такі як PostgreSQL, MySQL та SQLite, що дозволяє розробникам вибрати найбільш підходящий тип бази даних для свого проекту.

6. Інтеграція з GraphQL: Prisma може використовуватися в якості GraphQL-резольвера, що дозволяє зручно взаємодіяти з базою даних через GraphQL API.

Prisma також має інтеграцію з популярними фреймворками та інструментами для розробки, такими як Next.js, GraphQL та REST API. Це забезпечує безшовну інтеграцію з іншими частинами додатка і дозволяє використовувати всі переваги сучасного стека технологій.

Безпека даних також є важливим аспектом Prisma. Він забезпечує автоматичну перевірку даних на відповідність типам, що знижує ризик помилок та вразливостей у коді. Крім того, Prisma підтримує транзакції, що дозволяє забезпечити цілісність даних навіть у випадку складних операцій.

2.4.3 Aiven

Aiven було обрано для розгортання бази даних MySQL. Aiven є керованою хмарною платформою, яка надає інфраструктурні сервіси для різних баз даних та інших інструментів з відкритим кодом. Вона пропонує простий у використанні інтерфейс для розгортання, управління та моніторингу баз даних у різних хмарних середовищах, таких як AWS, Google Cloud Platform, Microsoft Azure та інших.

Основною перевагою використання Aiven є автоматизація рутинних завдань з адміністрування баз даних. Це включає автоматичне масштабування, резервне копіювання, оновлення, моніторинг продуктивності та безпеки. Завдяки цьому розробники можуть зосередитися на розробці додатків, а не на управлінні інфраструктурою.

Aiven підтримує широкий спектр баз даних, включаючи реляційні бази даних, такі як PostgreSQL, MySQL та MariaDB, а також NoSQL бази даних, такі як

Cassandra та Redis. Крім того, платформа надає підтримку для інших популярних інструментів з відкритим кодом, таких як Apache Kafka, Elasticsearch, InfluxDB та Grafana.

Однією з ключових особливостей Aiven є висока доступність і надійність. Платформа забезпечує автоматичне резервне копіювання і відновлення, а також багатозонне розгортання для забезпечення безперервної роботи додатків. Крім того, Aiven надає можливості для швидкого масштабування ресурсів, що дозволяє адаптувати інфраструктуру до змінних потреб бізнесу.

Безпека є ще одним важливим аспектом платформи Aiven. Вона забезпечує шифрування даних як у стані спокою, так і під час передачі, підтримує аутентифікацію за допомогою SSL/TLS і інтеграцію з системами управління доступом. Це гарантує, що дані клієнтів залишаються захищеними у будь-який момент.

Інтеграція з DevOps процесами також є сильною стороною Aiven. Платформа підтримує API та інструменти командного рядка, що дозволяє автоматизувати розгортання та управління інфраструктурою. Це сприяє більш швидкому та ефективному циклу розробки, розгортання та оновлення додатків.

Aiven надає також докладні аналітичні звіти та інструменти для моніторингу, що дозволяють відстежувати продуктивність баз даних і виявляти можливі проблеми на ранніх етапах. Це допомагає підтримувати високу продуктивність і доступність додатків.

Загалом, Aiven є потужною платформою для управління базами даних та іншими інструментами з відкритим кодом у хмарному середовищі. Вона забезпечує високу доступність, безпеку, масштабованість і інтеграцію з DevOps процесами, що робить її привабливим вибором для сучасних вебзастосунків.

2.5 Аутентифікація

Для реалізації аутентифікації та управління користувачами використовується сервіс Clerk. Clerk є платформою, що забезпечує просту та

надійну систему аутентифікації для веб та мобільних додатків. Вона дозволяє розробникам легко додавати функції реєстрації, входу та управління користувачами, забезпечуючи безпечний і зручний процес аутентифікації для кінцевих користувачів. Clerk підтримує різні методи аутентифікації, такі як електронна пошта, пароль, одноразові паролі (OTP), а також соціальні логіни через Google, Facebook, GitHub та інші.

Однією з головних переваг Clerk є її простота інтеграції. Платформа надає готові компоненти для React та інших популярних фреймворків, що дозволяє швидко впроваджувати функції аутентифікації без необхідності розробки з нуля. Це значно скорочує час розробки та знижує ймовірність помилок, забезпечуючи при цьому високу безпеку.

Clerk також пропонує потужний API для управління користувачами та їх сесіями. Розробники можуть легко додавати, видаляти та оновлювати інформацію про користувачів, керувати сесіями та контролювати доступ до різних частин додатка. Це забезпечує гнучкість і контроль над процесом аутентифікації, дозволяючи адаптувати його до специфічних потреб проекту.

Безпека є пріоритетом для Clerk. Платформа використовує сучасні методи шифрування для захисту даних користувачів, а також забезпечує багатофакторну аутентифікацію (MFA) для підвищення рівня безпеки. Крім того, Clerk підтримує автоматичне виявлення та запобігання підозрілим активностям, що допомагає захистити додаток від несанкціонованого доступу та зловмисних дій.

Крім аутентифікації, Clerk надає функції для управління профілями користувачів. Користувачі можуть оновлювати свою особисту інформацію, змінювати паролі та налаштовувати параметри безпеки через зручний інтерфейс. Це забезпечує позитивний досвід користувача і спрощує управління обліковими записами.

Clerk також інтегрується з різними сервісами та інструментами для розробки, такими як Next.js, Vercel, Firebase та інші. Це забезпечує безшовну інтеграцію з існуючою інфраструктурою додатка та дозволяє використовувати всі

переваги сучасного стеку технологій.

Clerk надає прості та зручні інструменти для додавання реєстрації, входу та управління профілями користувачів у вебзастосунку. Це дозволяє забезпечити високий рівень безпеки та зручності для користувачів.

Висновки до розділу 2

У цьому розділі було розглянуто ключові аспекти розробки застосунку, включаючи вибір редактора коду, розробку інтерфейсу, серверну частину, базу даних та аутентифікацію.

У розділі «Вибір редактора коду» виявлено, що використання Visual Studio Code є оптимальним вибором завдяки його розширеному набору функцій, що сприяють підвищенню продуктивності розробників.

Інтерфейс застосунку, розглянутий у підрозділі «Інтерфейс застосунку», визначено як сучасний, зручний та естетичний завдяки використанню TailwindCSS та ShadcnUI, що забезпечує якісну візуалізацію і взаємодію з користувачем.

Серверна частина застосунку, описана у розділі «Серверна частина застосунку», базується на Next.js, Socket.io та Typescript, що забезпечує швидку та надійну обробку запитів, а також інтеграцію з MySQL базою даних через Prisma ORM.

Вибір бази даних, викладений у розділі «Вибір бази даних», визначено як MySQL на платформі Aiven, що забезпечує високу надійність, швидкість та безпеку для зберігання та обробки даних.

У розділі «Аутентифікація» розглянуто інтеграцію з Clerk, що забезпечує безпечну та зручну систему аутентифікації користувачів, включаючи підтримку різних методів входу та управління профілями.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ДЛЯ КОМУНІКАЦІЇ ТА ОРГАНІЗАЦІЇ РОБОТИ В КОЛЕКТИВІ

3.1 Налаштування середовища та структури проєкту

Щоб розпочати розробку, перш за все необхідно налаштувати середовище для нашого застосунку. Як зазначалось в минулому розділі, в якості редактора було обрано Visual Studio Code. Головне вікно містить команди, поради та підказки, які можуть стати в нагоді при першому запускі проекту (див. рис. 3.1).

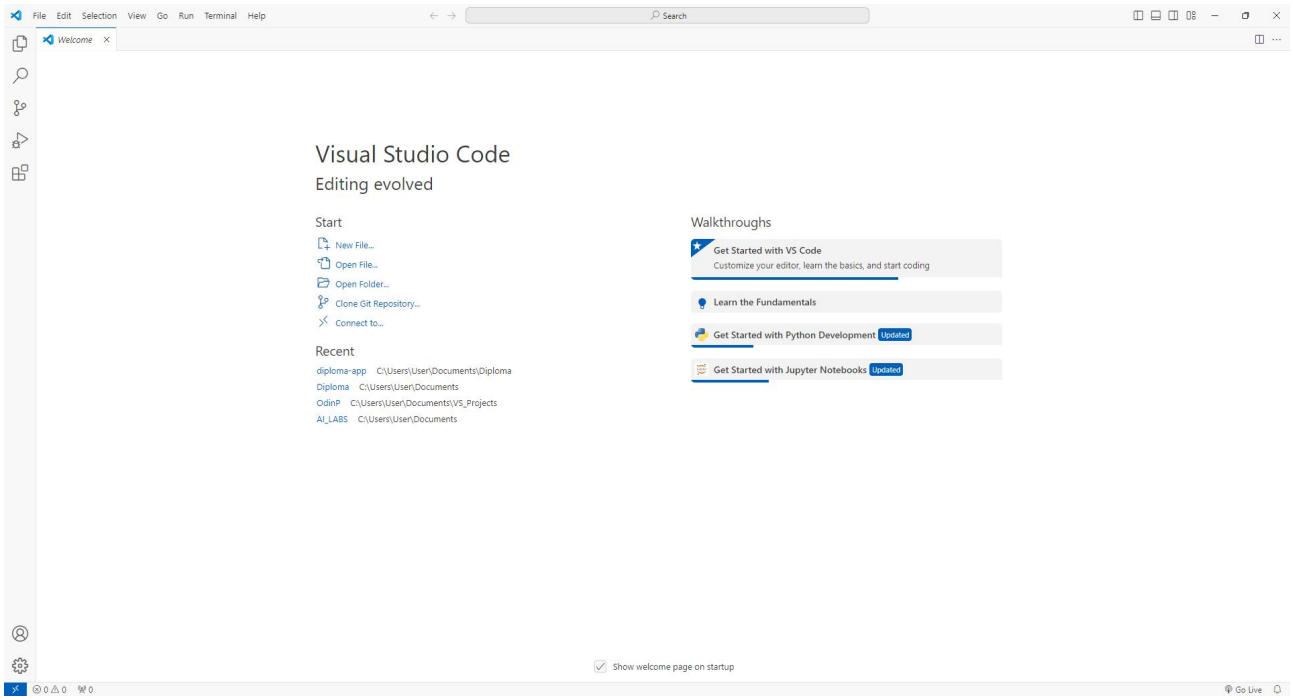

Рисунок 3.1 – Головне вікно Visual Studio Code

Фронтенд буде побудований використовуючи Next.js, який являється потужним фреймворком для React-додатків. Тож наступним кроком було проініціалізовано проєкт за допомогою команди «npx create-next-app@latest diplioma-app --typescript --tailwind --eslint». Ця команда генерує новий проєкт Next.js з назвою «diploma-app», використовуючи TypeScript для забезпечення типової безпеки, Tailwind CSS для стилізації та ESLint для лінтингу коду (див. рис. 3.2).

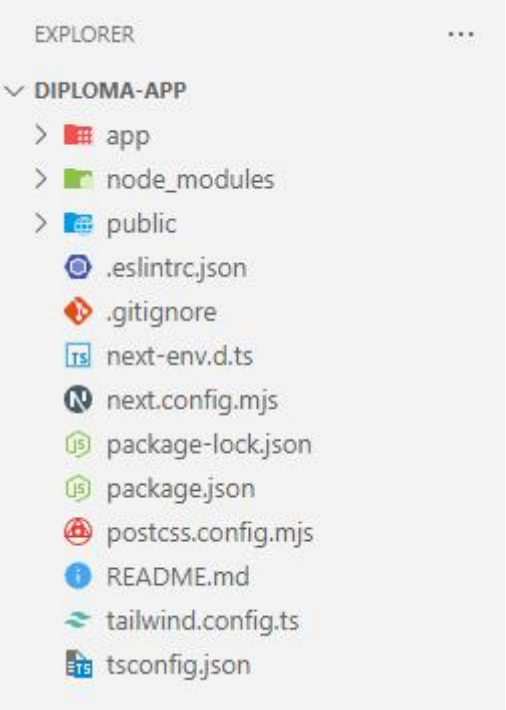

Рисунок 3.2 – Структура згенерованого проєкту

Далі було встановлено бібліотеку компонентів Shadcn UI за допомогою команди «npx shadcn-ui@latest init». Ця бібліотека використовується разом з Next.js для створення стилів застосунку.

Після виконання команд структура проекту включає декілька основних директорій, які представлені нижче.

1. /app: У цій директорії знаходиться код застосунку, включаючи сторінки та глобальні конфігурації CSS. Вона підтримує такі особливості, як визначення макетів, помилок і завантаження файлів для окремих маршрутів, груп маршрутів, компонентів сервера React тощо.

2. /components: Тут ми керуємо компонентами інтерфейсу користувача. Компоненти Shadcn UI автоматично додаються сюди для легкої інтеграції.

3. /lib: В цій папці розміщуються допоміжні функції або утиліти, які використовуються в різних частинах застосунку. Наприклад, функції для обробки дат, роботи з API, функції форматування тексту, функція CN для управління динамічними класами Tailwind CSS тощо.

Наступним кроком було додано фрагмент коду до файлу globals.css для коректного відображення стилів у застосунку (див. рис. 3.3).

```
globals.css 5, M
app > globals.css > ...
1   @tailwind base;
2     @tailwind components;
3     @tailwind utilities;
4
5   html,
6   body,
7   :root {
8     height: 100%;
9   }
```

Рисунок 3.3 – Доданий фрагмент до файлу globals.css

Після налаштування робочого середовища та організації структури проекту ми готові приступити до розробки основних функцій нашого застосунку. Наступні кроки включають впровадження аутентифікації, налаштування інтеграції з базамою даних використовуючи Prisma, а також створення інтуїтивно зрозумілих інтерфейсів користувача.

3.2 Реалізація аутентифікації та авторизації

В цьому розділі буде розглянуто процес інтеграції системи аутентифікації та авторизації в застосунок за допомогою платформи Clerk (див. рис. 3.4).

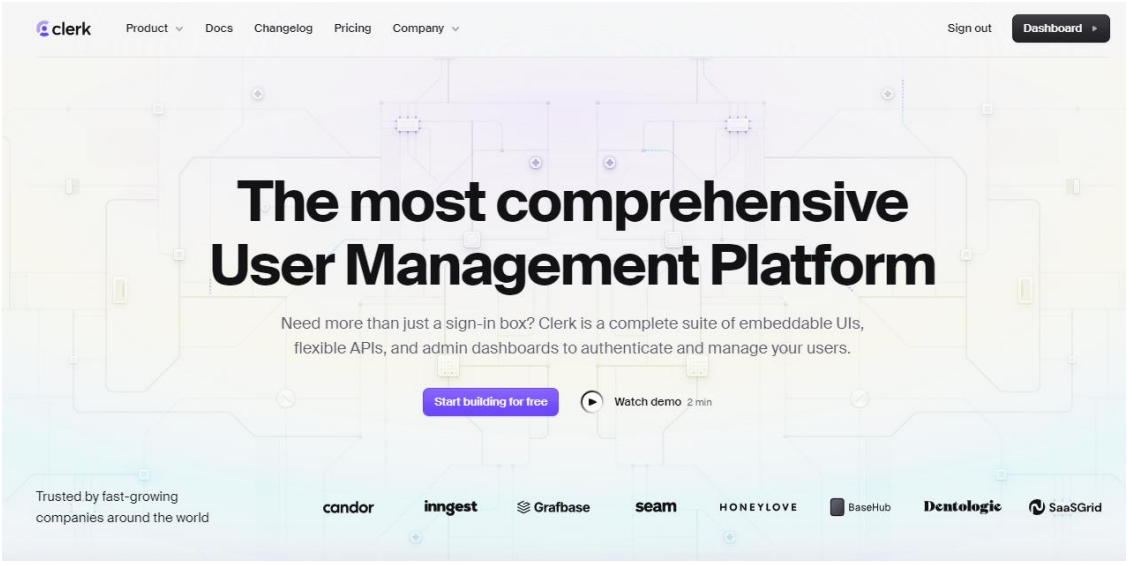

Рисунок 3.4 – Головна сторінка платформи Clerk

Перш за все було створено обліковий запис на платформі Clerk, створено

новий застосунок та отримано необхідні ключі доступу. Ці ключі доступу необхідно помістити у файл .env, щоб мати змогу отримати доступ до них з будь-якої точки застосунку (див. рис. 3.5).

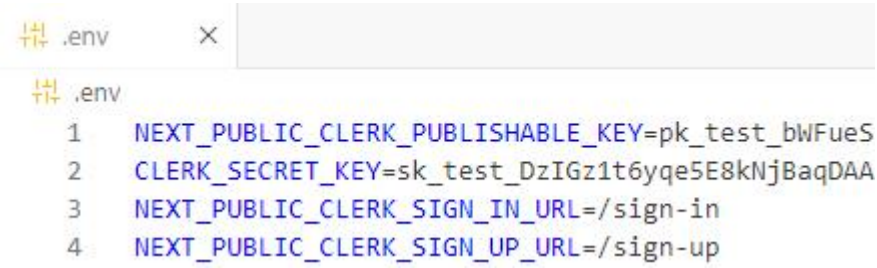

Рисунок 3.5 – Частина доданих ключів у файлі .env

Clerk SDK надає можливість обрати декілька сервісів для входу. Тож було вирішено додати можливість входу через популярні соціальні мережі, такі як Google та Facebook, а також GitHub, що дозволяє забезпечити швидкий і зручний процес авторизації. Clerk SDK автоматично обробляє потоки аутентифікації, включаючи підтвердження через електронну пошту та повідомлення для підвищення безпеки.

Для того щоб інтегрувати хуки та компоненти платформи Clerk в застосунок необхідно в терміналі прописати команду «npm install @clerk/nextjs» та додати middleware (див. рис. 3.6).

Рисунок 3.6 – Створений middleware

Функція clerkMiddleware() надає доступ до статусу аутентифікації користувача в усьому застосунку, на будь-якому маршруті чи сторінці. Вона також дозволяє захистити певні маршрути від неаутентифікованих користувачів.

Усі хуки та компоненти Clerk мають бути нащадками компонента <ClerkProvider>, який надає активний сеанс і контекст користувача. Через це маємо обгорнути наші компоненти у файлі layout.tsx (див. рис. 3.7).

```
13  export default function RootLayout({
14    children,
15  }: Readonly<{
16    children: React.ReactNode;
17  }>) {
18    return (
19      <ClerkProvider>
20        <html lang='en'>
21          <body>
22            <SignedOut>
23              <SignInButton />
24            </SignedOut>
25            <SignedIn>
26
27            </SignedIn>
28            {children}
29          </body>
30        </html>
31    </ClerkProvider>
32    );
33  }
```

Рисунок 3.7 – Обгорнуті компоненти в у файлі app/layout.tsx

Далі було створено сторінки реєстрації та авторизації використовуючи компоненти Clerk SDK (див. рис. 3.8).

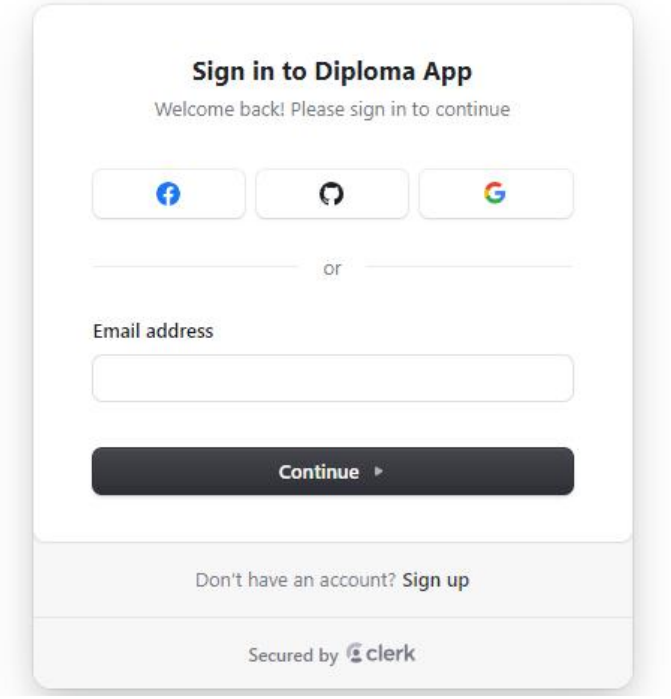

Рисунок 3.8 – Форма авторизації

Як видно на рис. 3.8, в застосунок можна авторизуватись використовуючи пошту, обліковий запис Google, Facebook та GitHub. На рис. 3.9 представлено форму реєстрації.

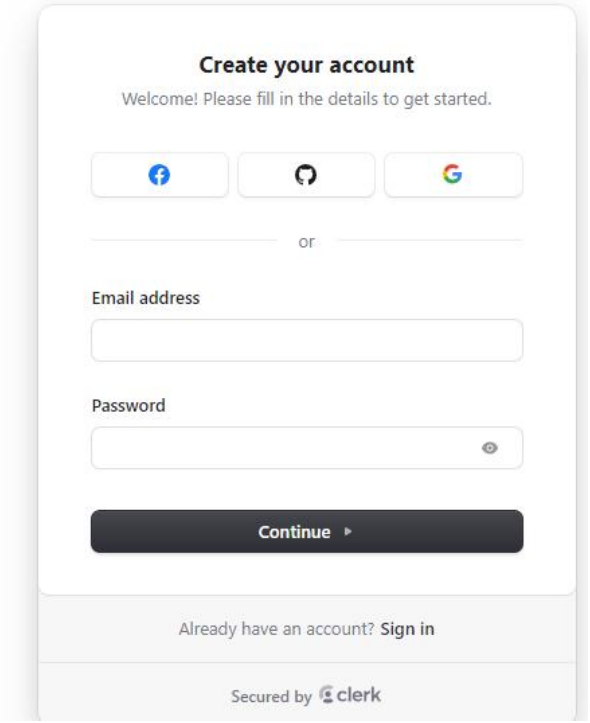

Рисунок 3.9 – Форма реєстрації

Реєстрація можлива використовуючи пошту або облікові записи Google, Facebook та GitHub.

3.3 Реалізація взаємодії з БД

Для взаємодії з базою даних було використано Prisma, що спростило взаємодію з базою даних. Для встановлення Prisma необхідно виконати команду «npm install -D prisma». Ця команда встановить і додасть Prisma до devDependencies.

Тепер необхідно створити початкове налаштування Prisma за допомогою команди init інтерфейсу командного рядка Prisma, а саме командою «npx prisma init». Ця команда створює нову директорію Prisma з файлом schema.prisma, який визначає підключення до бази даних і містить схему бази даних. Команда також модифікує файл .env, додаючи змінну DATABASE_URL.

Підключення до бази даних налаштовано в блоці джерела даних у файлі schema.prisma. За замовчуванням встановлено postgresql, але оскільки ми використовуєте базу даних MySQL, потрібно змінити provider на MySQL (див. рис. 3.10).

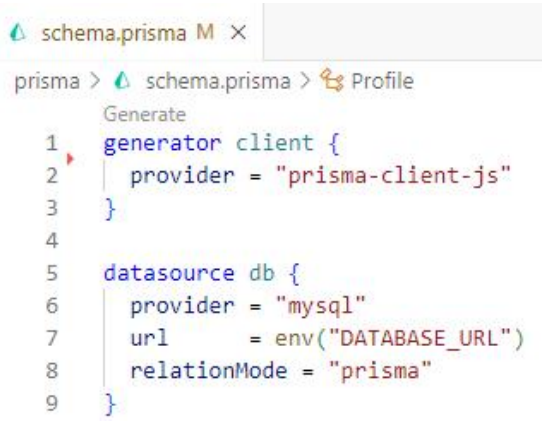

Рисунок 3.10 – Зміна з PostgreSQL на MySQL у файлі schema.prisma

Платформу Aiven було обрано для розгортання бази даних MySQL. Aiven для MySQL дозволяє отримати повністю керовану базу даних MySQL, розгорнуту в хмарі за кілька хвилин. При створені служби можна обрати хмарного постачальника, де працюватиме наша база даних. В обраній безкоштовній версії доступний лише DigitalOcean, тож цю компанію й було обрано. В платній версії можна обрати між AWS, GCP, Azure, DigitalOcean і UpCloud (див. рис. 3.11).

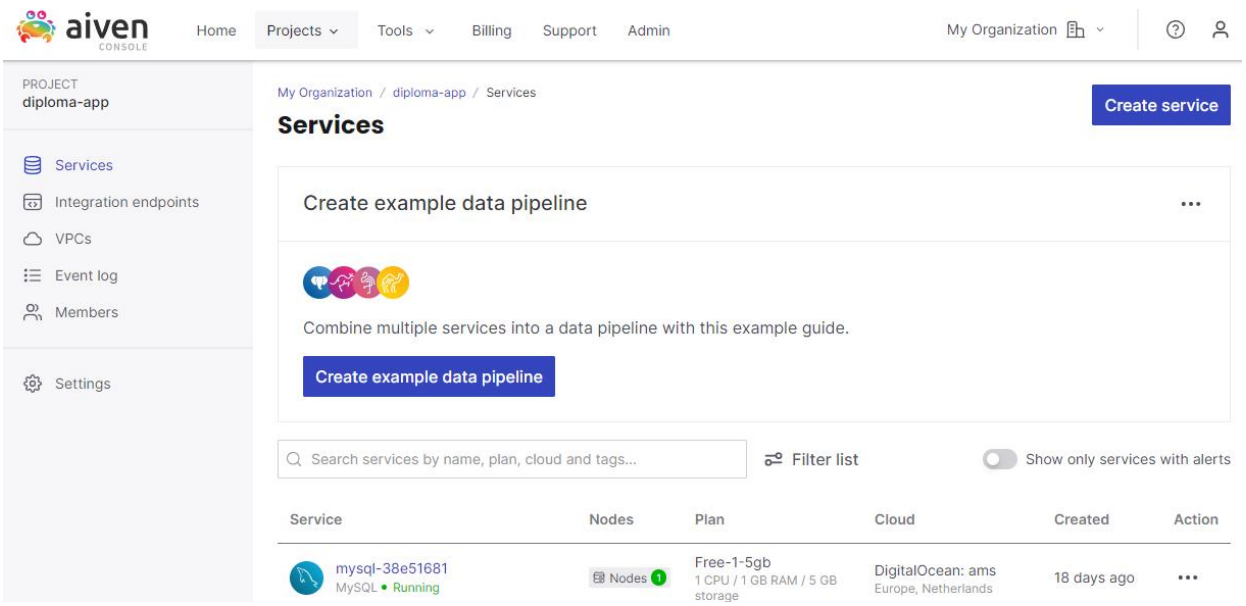

Рисунок 3.11 – Розгорнута база даних MySQL на платформі Aiven

На рисунку 3.11 можна бачити створену службу, її статус та інформацію про неї.

Далі необхідно скопіювати рядок підключення до Aiven MySQL. Для цього на сторінці служби Aiven MySQL необхідно скопіювати Service URI. Рядок підключення показано на рис. 3.12.

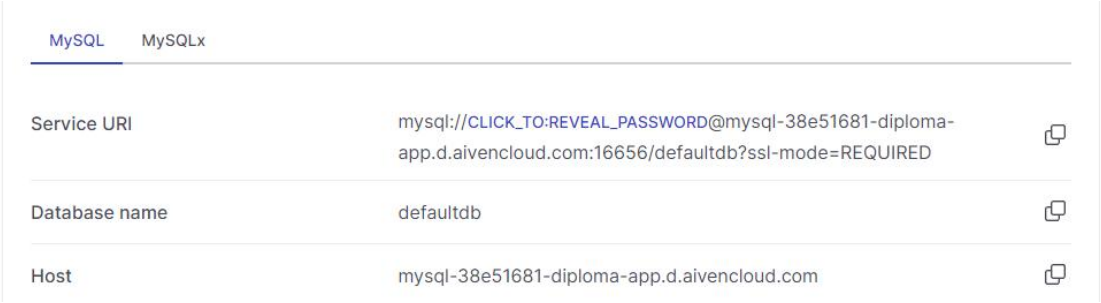

Рисунок 3.12 – Інформація для підключення до служби

Після копіювання рядок Service URI потрібно додати до створеної у файлі .env змінної DATABASE_URL. Після цього можна приступати до створення необхідних моделей.

Моделі додаються у файл schema.prisma. Модель Profile була створена для зберігання основної інформації про користувача. Вона містить унікальний ідентифікатор, ім'я користувача, URL зображення та електронну пошту. Модель також включає списки серверів, членів і каналів, які асоціюються з профілем користувача. Крім того, модель має поля для відстеження часу створення та оновлення запису (див. рис. 3.13).

```
11  model Profile {
12    id String @id @default(uuid())
13    userId String @unique
14    name String
15    imageUrl String @db.Text
16    email String @db.Text
17
18    servers Server[]
19    members Member[]
20    channels Channel[]
21
22    createdAt DateTime @default(now())
23    updatedAt DateTime @updatedAt
24  }
```

Рисунок 3.13 – Модель Profile

Модель Server призначена для представлення серверів, які будуть створюватись користувачами та до яких можуть приєднуватися інші користувачі. Вона містить унікальний ідентифікатор, назву сервера, URL зображення та унікальний код запрошення. Модель також включає зв'язок з профілем власника сервера, список членів і каналів, що належать до цього сервера. Поля createdAt і updatedAt відстежують час створення та оновлення сервера (див. рис. 3.14).

```
model Server {
  id String @id @default(uuid())
  name String
  imageUrl String @db.Text
  inviteCode String @unique

  profileId String
  profile Profile @relation(fields: [profileId], references: [id], onDelete: Cascade)

  members Member[]
  channels Channel[]

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  @@index([profileId])
}
```

Рисунок 3.14 – Модель Server

На рис. 3.15 представлено модель Member.

```
50  model Member {
51    id String @id @default(uuid())
52    role MemberRole @default(GUEST)
53
54    profileId String
55    profile Profile @relation(fields: [profileId], references: [id], onDelete: Cascade)
56
57    serverId String
58    server Server @relation(fields: [serverId], references: [id], onDelete: Cascade)
59
60    messages Message[]
61    directMessages DirectMessage[]
62
63    conversationsInitiated Conversation[] @relation("MemberOne")
64    conversationsReceived Conversation[] @relation("MemberTwo")
65
66
67    createdAt DateTime @default(now())
68    updatedAt DateTime @updatedAt
69
70    @@index([profileId])
71    @@index([serverId])
72  }
```

Рисунок 3.15 – Модель Member

Модель Member використовується для зберігання інформації про членів сервера. Вона містить унікальний ідентифікатор, роль члена (ADMIN, MODERATOR, GUEST), а також зв'язки з профілем користувача та сервером. Крім того, модель включає списки повідомлень, прямих повідомлень і ініційованих або отриманих розмов. Поля createdAt і updatedAt відстежують час створення та оновлення члена.

Модель Channel призначена для представлення каналів на сервері. Вона містить унікальний ідентифікатор, назву каналу та тип каналу (TEXT, AUDIO, VIDEO). Модель також включає зв'язки з профілем користувача та сервером, до якого належить канал, а також список повідомлень на цьому каналі. Поля createdAt і updatedAt відстежують час створення та оновлення каналу (див. рис. 3.16).

```
80  model Channel {
81    id String @id @default(uuid())
82    name String
83    type ChannelType @default(TEXT)
84
85    profileId String
86    profile Profile @relation(fields: [profileId], references: [id], onDelete: Cascade)
87
88    serverId String
89    server Server @relation(fields: [serverId], references: [id], onDelete: Cascade)
90
91    messages Message[]
92
93    createdAt DateTime @default(now())
94    updatedAt DateTime @updatedAt
95
96    @@index([profileId])
97    @@index([serverId])
98  }
```

Рисунок 3.16 – Модель Channel

Відношення між всіма моделями продемонстровано у додатку Б.

Також було створено два enum (див. рис. 3.17), які використовуються для визначення ролей користувачів та типу каналів. Текстові канали призначені для обміну текстовими повідомленнями. Вони є основним типом каналів для письмового спілкування між учасниками.

Аудіоканали використовуються для голосового спілкування. Вони дозволяють користувачам спілкуватися за допомогою мікрофона в реальному часі.

Відеоканали призначені для відеозв'язку. Вони дозволяють учасникам спілкуватися за допомогою відеокамер та мікрофонів, забезпечуючи більш інтерактивне та особисте спілкування.

Роль ADMIN призначена для адміністраторів сервера. Адміністратори мають повні права на управління сервером, включаючи додавання або видалення користувачів, зміну налаштувань сервера та модерацію контенту.

Модератори(MODERATOR) мають обмеженіші права порівняно з адміністраторами. Вони можуть модерувати контент, видаляти повідомлення, банити користувачів тощо, але не мають повного доступу до всіх налаштувань сервера.

Гості(GUEST) мають мінімальні права на сервері. Вони можуть переглядати канали та спілкуватися, але їхні можливості обмежені у порівнянні з адміністраторами та модераторами.

```
enum MemberRole {
  ADMIN
  MODERATOR
  GUEST
}
enum ChannelType {
  TEXT
  AUDIO
  VIDEO
}
```

Рисунок 3.17 – Створені enum

Далі в директорії lib було створено файл initial-profile.ts для забезпечення створення та обробки профілю користувача у застосунку. Він має забезпечити перевірку автентичності користувача та забезпечити створення нового профілю, якщо його ще не існує. На рис. 3.18 предстaлено вміст цього файлу. В коді було імпортовано currentUser, RedirectToSignIn, auth з пакету @clerk/nextjs для роботи з

аутентифікацією та авторизацією через Clerk та db з локального модуля @/lib/db для взаємодії з базою даних.

lib > initial-profile.ts > ...

```ts
import { currentUser } from "@clerk/nextjs/server";
import { auth } from "@clerk/nextjs/server";
import { db } from "@/lib/db";

export const initialProfile = async () => {
    const user = await currentUser();

    if(!user){
        return auth().redirectToSignIn()
    }

    const profile = await db.profile.findUnique({
        where: {
            userId:user.id
        }
    })

    if (profile) {
        return profile;
    }

    const newProfile = await db.profile.create({
        data: {
            userId:user.id,
            name: `${user.firstName} ${user.lastName}`,
            imageUrl: user.imageUrl,
            email: user.emailAddresses[0].emailAddress
        }
    })

    return newProfile;
}
```

Рисунок 3.18 – Вміст initial-profile.ts

Спочатку функція викликає currentUser для отримання поточного аутентифікованого користувача. Якщо користувач не знайдений, то функція викликає метод redirectToSignIn для перенаправлення неавторизованого користувача на сторінку входу. Наступним кроком функція шукає профіль у базі даних за допомогою db.profile.findUnique, використовуючи userId поточного користувача. Якщо профіль знайдений (profile не null), функція повертає

знайдений профіль. Якщо профіль не знайдений (profile дорівнює null), функція створює новий профіль за допомогою db.profile.create. Під час створення нового профілю використовуються дані поточного користувача.

Далі було створено файл app/(setup)/page.tsx. Вміст файлу наведено нижче на рис. 3.19. Цей файл забезпечує процес налаштування користувача при першому вході у додаток. Він обробляє створення та перевірку профілю користувача використовуючи створений раніше initialProfile(), а також перевіряє, чи користувач вже є учасником будь-якого сервера за допомогою команди db.server.findFirst, і відповідне перенаправляє на цей сервер. В інакшому випадку відображає модальне вікно InitialModal, яке призначене для початкового налаштування користувача.

app > (setup) > page.tsx > default

```tsx
import { initialProfile } from "@/lib/initial-profile";
import { db } from "@/lib/db";
import { redirect } from "next/navigation";
import { InitialModal } from "@/components/modals/initial-modal";

const SetupPage = async () => {
    const profile = await initialProfile();

    const server = await db.server.findFirst({
        where: {
            members: {
                some: {
                    profileId: profile.id
                }
            }
        }
    })
    if(server){
        return redirect(`/servers/${server.id}`);
    }
    return (
        <InitialModal/>
    );
}

export default SetupPage;
```

Рисунок 3.19 – Вміст app/(setup)/page.tsx

Тепер якщо зареєструватись в застосунку, то дані збережуться в базі даних.

Це можна перевірити відкривши Prisma Studio командою npx prisma studio. Як видно на рис. 3.20 в базі даних на даний момент існує два користувачі.

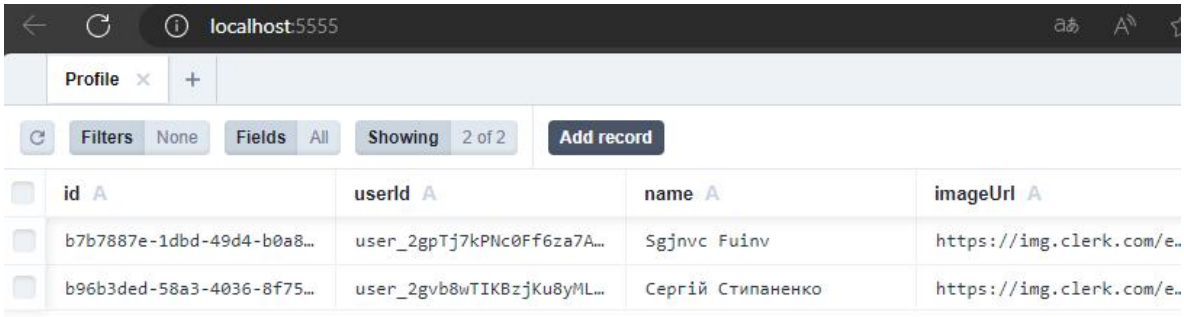

Рисунок 3.20 – Список користувачів

3.4 Створення серверної частини та інтерфейсу користувача

3.4.1 Реалізація вікна першого входу

При першому вході або під час налаштування нової кімнати користувач зустріне модальне вікно initialModal (див. рис. 3.21), яке було згадано раніше. Це вікно дозволяє користувачам налаштувати свою кімнату при першому вході або під час налаштування нової кімнати.

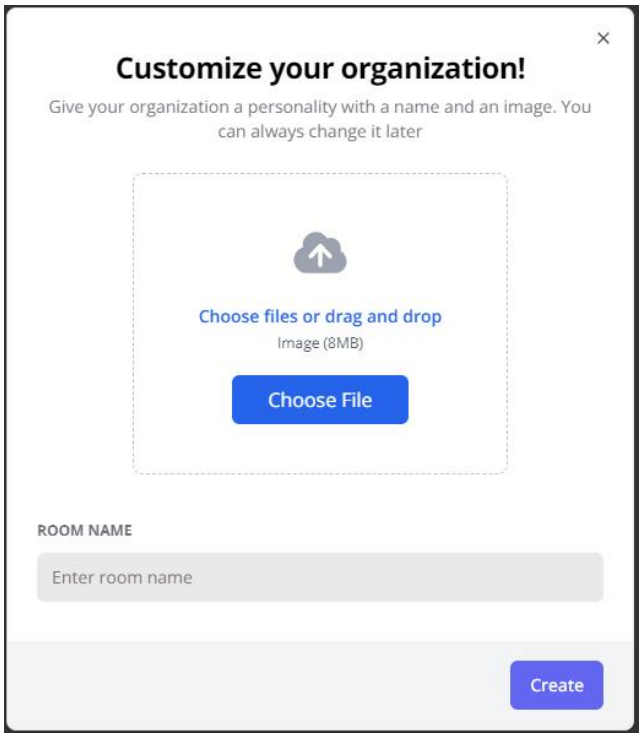

Рисунок 3.21 – Вигляд модального вікна initialModal

Це модальне вікно надає користувачам можливість ввести ім'я для організації та завантажити для неї зображення. Завантаження зображення реалізовано за допомогою платформи UploadThing. Для цього було створено файл app/api/uploadthing/core.ts (див. рис. 3.22). Він використовує бібліотеку uploadthing для керування завантаженням файлів і інтегрує аутентифікацію за допомогою платформи Clerk.

app > api > uploadthing > core.ts > ...

```ts
import { createUploadthing, type FileRouter } from "uploadthing/next";
import { auth } from "@clerk/nextjs/server";
const f = createUploadthing();

const handleAuth = () => {
    const {userId} = auth();
    if(!userId) throw new Error("Unauthorized");
    return {userId:userId}
}
export const ourFileRouter = {
    serverImage: f({image:{maxFileSize:"8MB", maxFileCount:1}})
        .middleware(()=>handleAuth())
        .onUploadComplete(()=>{}),
    messageFile: f(["image", "pdf", "video","application/vnd.ms-word.document.macroenabled.12"])
        .middleware(()=>handleAuth())
        .onUploadComplete(()=>{})
} satisfies FileRouter;

export type OurFileRouter = typeof ourFileRouter;
```

Рисунок 3.22 – Вміст app/api/uploadthing/core.ts

Функція handleAuth використовується для аутентифікації користувачів. Вона викликає функцію auth, щоб отримати інформацію про поточного користувача. Якщо userId не існує, вона кидає помилку «Unauthorized». Якщо userId існує, вона повертає об'єкт з userId. serverImage визначає маршрут для завантаження зображень не сервер. Він дозволяє завантажувати тільки зображення з максимальним розміром файлу 8MB і можливістю завантажити лише 1 файл. Цей маршрут використовує middleware, щоб перевірити аутентифікацію користувача через handleAuth, і викликає onUploadComplete після завершення завантаження.

3.4.2 Реалізація бічної панелі навігації

Бічна панель навігації за дизайном схожа на бічну панель застосунку

Discord, адже забезпею зручну навігацію між каналами та організаціями та гарантує швидку взаємодію (див. рис. 3.23).

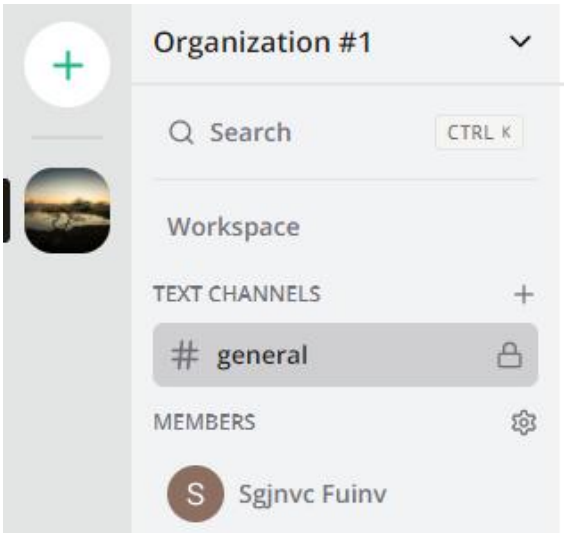

Рисунок 3.23 – Вигляд бічної панелі

На бічній панелі бачимо кнопку організації з її фотографією, при натисканні на яку відкривається бічна панель цієї організації з її каналами, учасниками, полем для пошуку інформації, меню для налаштування та кнопкою Workspace. Також наявна велика кнопка зі знаком плюс, яка забезпечує можливість створення нової організації. При її натисканні відкриється вікно яке було представлено на рис. 3.21. Для того щоб побачити меню для налаштування організації необхідно натиснути на назву створеної організації або на направлену вниз стрілочку праворуч від назви.

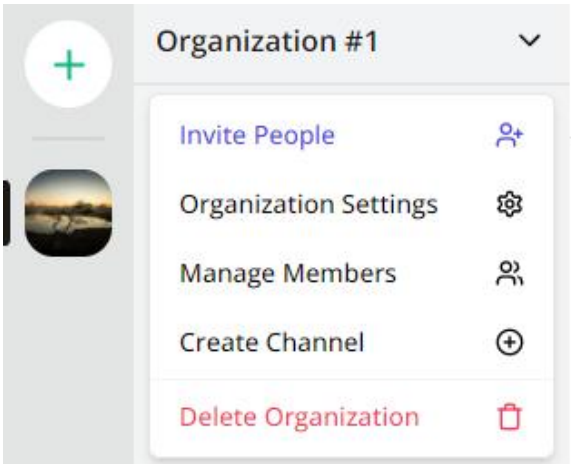

Рисунок 3.24 – Вигляд меню для управління організацією

Меню містить декілька важливих опцій. Опція Invite People відкриває модальне вікно для запрошення людей до організації. Це вікно представлено на рис. 3.25. Опція Organization Settings відкриє вікно налаштувань представлене на рис. 3.26, в якому можна змінити назву та фотографію організації. Опція Manage Members відкриє вікно керування учасниками (рис. 3.27), в якому можна видалити користувача з організації та змінити його роль. Опція Create Channel відкриє вікно створення каналу (рис. 3.28), в якому можна вказати назву та тип каналу. Опція Delete Organization видались організацію без можливості повернення.

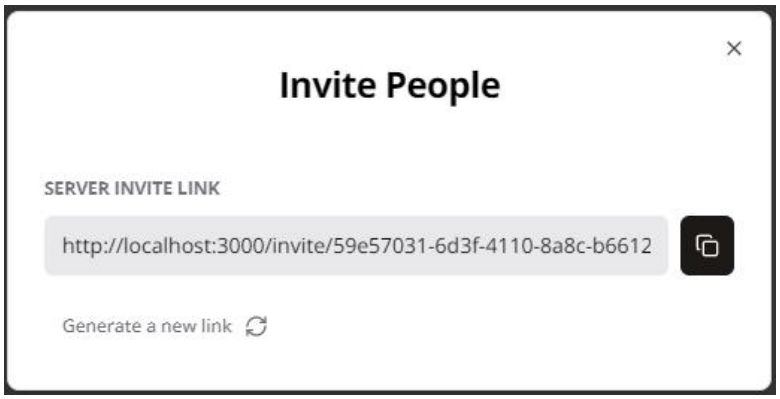

Рисунок 3.25 – Вигляд вікна запрошення

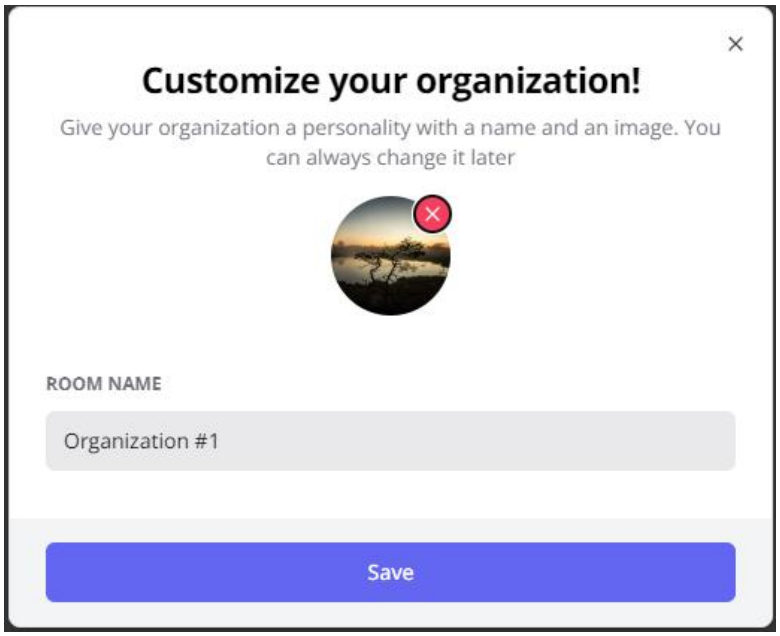

Рисунок 3.26 – Вигляд вікна Organization Settings

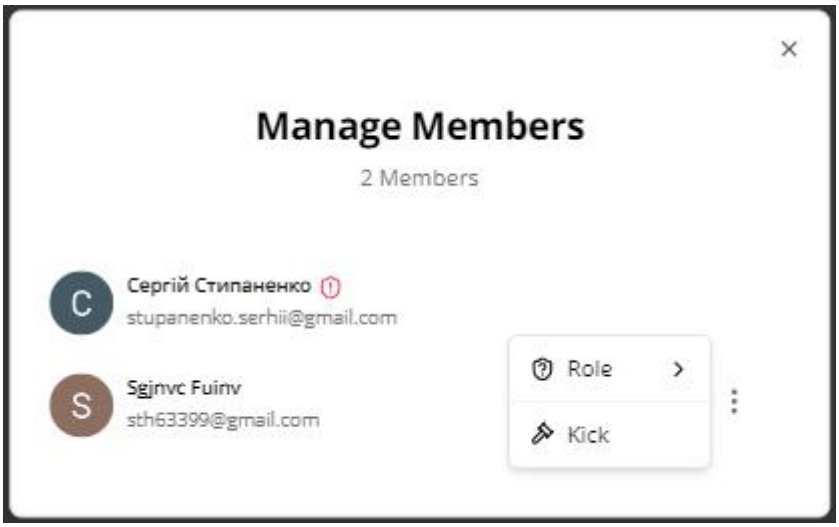

Рисунок 3.27 – Вигляд вікна Manage Members

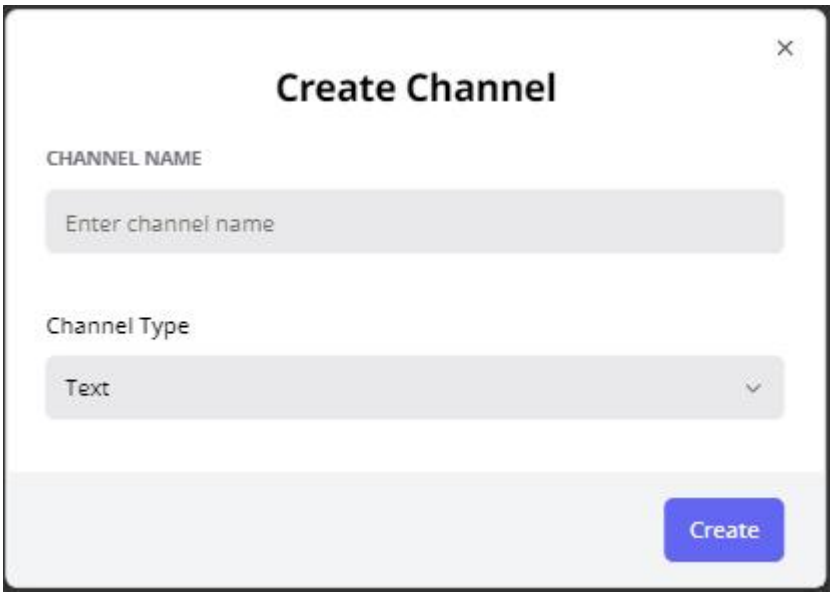

Рисунок 3.28 – Вигляд вікна Manage Members

Код на рис. 3.29 забезпечує відображення серверів, до яких належить користувач, а також надає елементи управління, такі як перемикач режимів (темний/світлий) і кнопку користувача. Спочатку викликається функція currentProfile для отримання профілю поточного користувача. Якщо його не знайденом, то користувач перенаправляється на головну сторінку. Після цього виконується запит до бази даних для отримання списку серверів, в яких користувач є учасником.

```
export const NavigationSidebar = async () => {
    const profile = await currentProfile();

    if(!profile) return redirect("/");

    const servers = db.server.findMany({
        where:{
            members:{
                some:{
                    profileId:profile.id
                }
            }
        }
    });
    return (
        <div className="space-y-4 flex flex-col items-center h-full text-primary w-full dark:bg-[#1E1F22] bg-[#E3E5E5] py-3">
            <NavigationAction/>
            <Separator className="h-[2px] bg-zinc-300 dark:bg-zinc-700 rounded-md w-10 mx-auto"/>
            <ScrollArea className="flex-1 w-full">
                {(await servers).map((server)=>(
                    <div key={server.id} className="mb-4">
                        <NavigationItem id={server.id} name={server.name} imageUrl={server.imageUrl} />
                    </div>
                ))}

            </ScrollArea>
            <div className="pb-3 mt-auto flex items-center flex-col gap-y-4">
                <ModeToggle/>
                <UserButton afterSignOutUrl="/" appearance={{elements:{avatarBox:"h-[48px] w-[48px]"}}}/>
            </div>
        </div>
    )
}
```

Рисунок 3.29 – Вміст файлу navigation-sidebar.tsx

Функція PATCH (див. рис. 3.30) використовуються для оновлення коду запрошення сервера. Функція перевіряє автентифікацію користувача і оновлює код запрошення організації в базі даних.

```
export async function PATCH(req:Request, {params}:{params:{serverId:string}}) {
    try {
        const profile = await currentProfile();
        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!params.serverId) return new NextResponse("Server ID missing", {status:400});

        const server = await db.server.update({
            where:{
                id: params.serverId,
                profileId: profile.id
            },
            data: {
                inviteCode: uuidv4()
            }
        })
        return NextResponse.json(server);
    } catch (error) {
        console.log("[SERVER_ID]", error);
        return new NextResponse("Internal Error", {status:500});
    }
}
```

Рисунок 3.30 – Функція для оновлення коду запрошення

Функція на рис. 3.31 забезпечує можливість створення нових каналів для певної організації, гарантуючи, що тільки авторизовані користувачі з відповідними ролями можуть виконувати цю операцію. На рис. 3.32 наведено вигляд бічної панелі після додавання каналів.

```
export async function POST(req:Request) {
    try {
        const profile = await currentProfile();
        const {name, type} = await req.json();
        const {searchParams} = new URL(req.url);

        const serverId = searchParams.get("serverId");
        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!serverId) return new NextResponse("Server ID missing", {status:400});

        if(name === "general") return new NextResponse("Name cannot be 'general'", {status:400});

        const server = await db.server.update({
            where:{
                id: serverId,
                members: {
                    some: { profileId: profile.id, role: {in: [MemberRole.ADMIN, MemberRole.MODERATOR]}}
                }
            },
            data: {channels: { create: { profileId: profile.id, name, type}}}
        });
        return NextResponse.json(server);
    } catch (error) {
        console.log("[CHANNEL_POST]", error);
        return new NextResponse("Internal Error", {status:500});
    }
}
```

Рисунок 3.31 – Функція для створення нового каналу

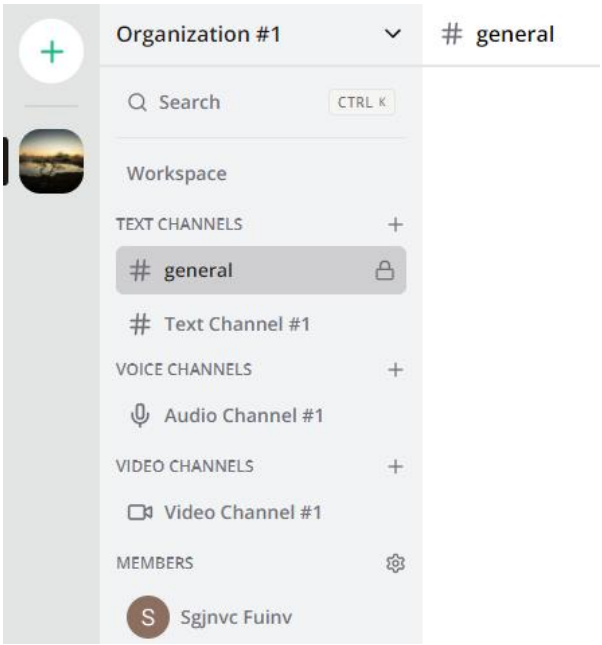

Рисунок 3.32 – Вигляд бічної панелі після додавання каналів

На рис. 3.33 представлено вікно пошуку всередині організації. Пошук може здійснюватися як по наявним каналам, так і по учасникам. При натисканні на канал чи учасника у вікні пошуку, користувач буде перенаправлений на відповідний канал або на сторінку чату з учасником.

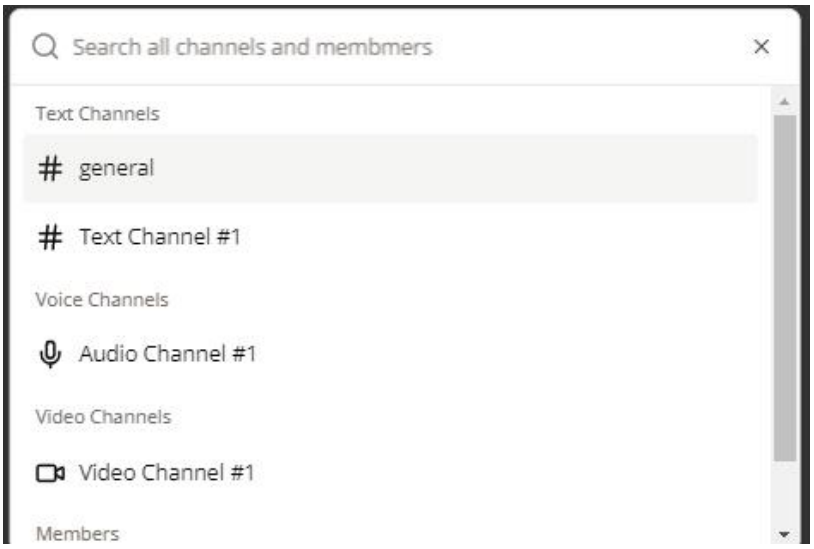

Рисунок 3.33 – Вигляд бічної панелі після додавання каналів

Таким чином навігація по каналам і учасникам організації є досить зручною.

3.4.3 Реалізація робочого простору організації

При натисканні на кнопку Workspace на бічній панелі організації (див. рис. 3.23) перед користувачем з’явиться вікно з дошками організації (див. рис. 3.34).

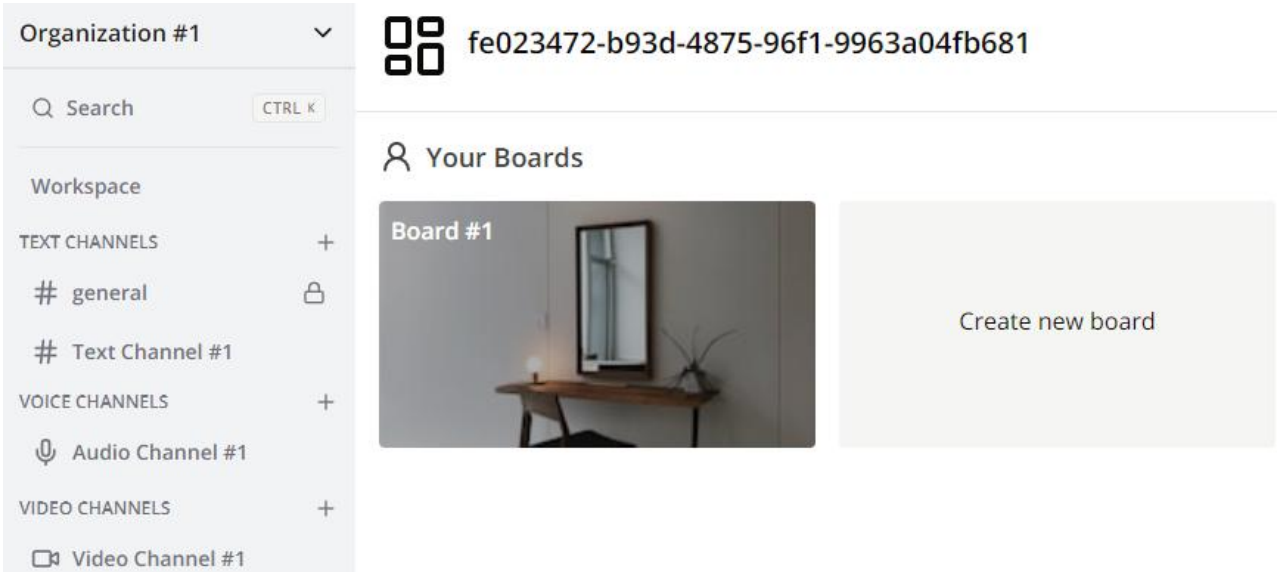

Рисунок 3.34 – Вигляд робочого простору організації

У цьому вікні користувачі зможуть бачити уже створені дошки та створювати нові (див. рис. 3.35).

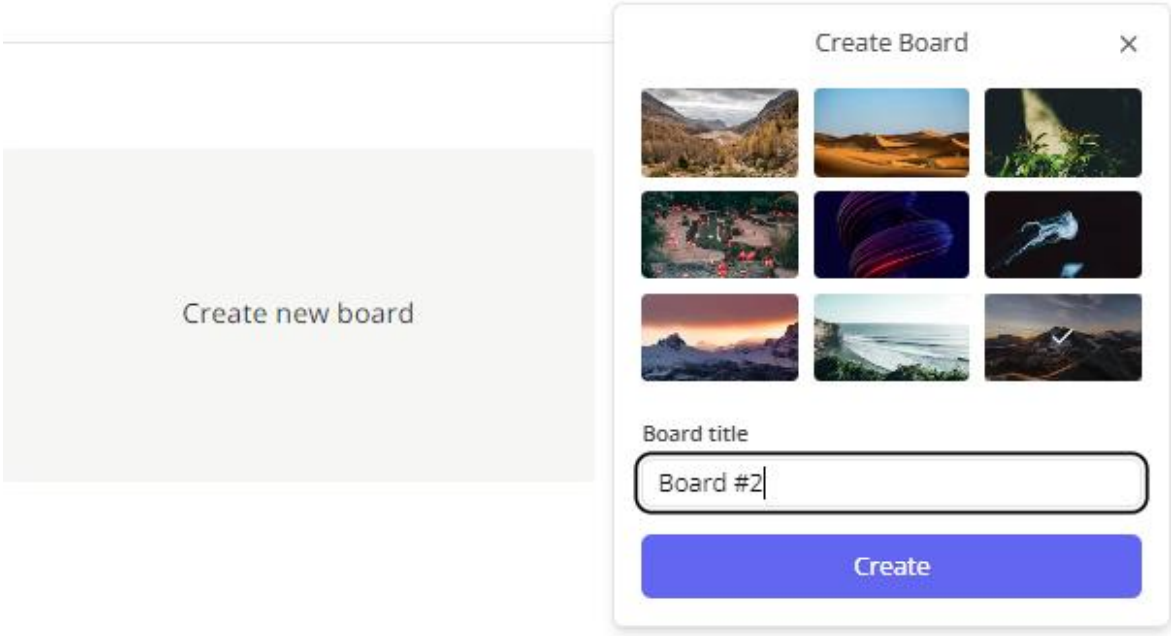

Рисунок 3.35 – Процес створення нової дошки

При створенні нової дошки користувачі мають змогу придумати назву та обрати випадкову фотографію для дошки. Фотографії генеруються випадковим чином використовуючи Unsplash API та бібліотеку unsplash-js. В наведеному на рис. 3.36 коді useEffect використовується для завантаження випадкових зображень.

```
useEffect(() => {
    const fetchImages = async () => {
      try {
        const result = await unsplash.photos.getRandom({collectionIds: ["305011"], count: 9});
        if (result && result.response) {
          const responseImages = result.response as Array<Record<string, any>>;
          setImages(responseImages);
        } else {
          setImages(defaultImages);
          console.error("Failed to get images from Unsplash.");
        }
      } catch (error) {
        console.error(error);
        setImages(defaultImages);
      } finally {
        setIsloading(false);
      }
    };
    fetchImages();
}, []);
```

Рисунок 3.36 – Фрагмент коду для завантаження випадкових фотографій

Якщо через якісь причини виникне помилка, то будуть використані зображення за замовчуванням. Таким чином користувачі завжди матимуть доступні фотографії.

Далі користувачі можуть натиснути на будь-яку зі створених дошок і перейти до управління списками дошки (див. рис. 3.37).

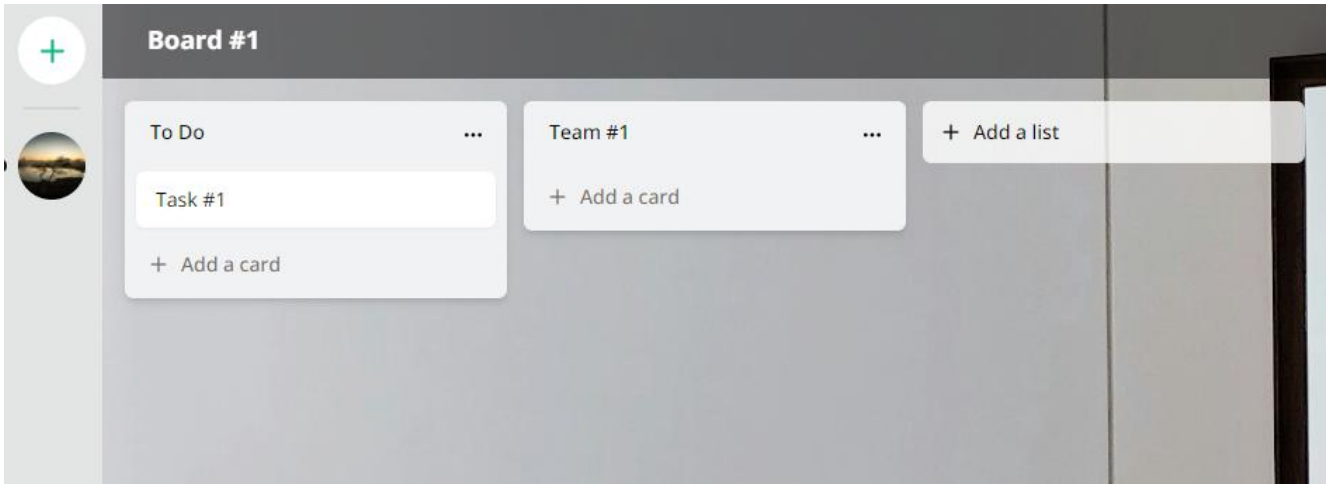

Рисунок 3.37 – Додавання списків завдань до обраної дошки

Користувачі можуть створювати необхідну кількість списків, копіювати та видаляти їх та додавати до списків картки із завданнями. Картки із завданнями можна переміщати між списками, копіювати, видяляти та редагувати (див. рис. 3.38).

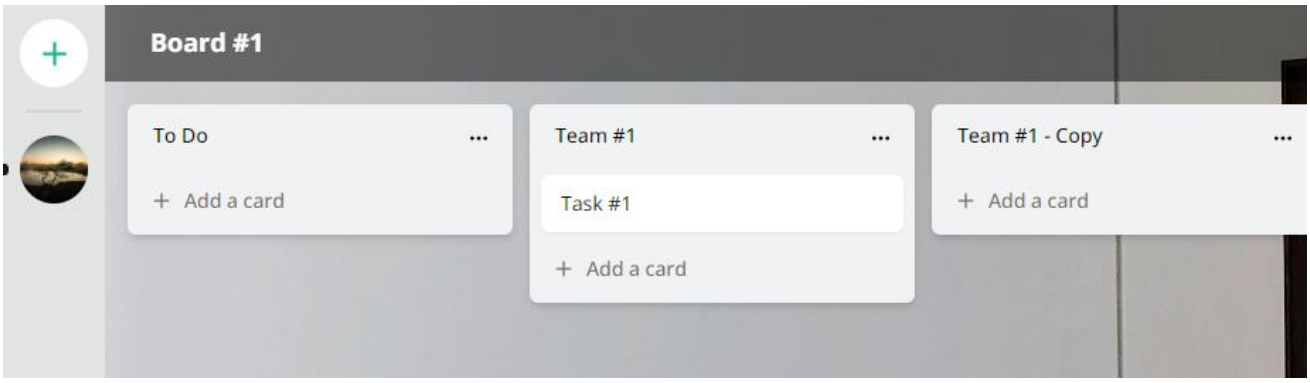

Рисунок 3.38 – Демонстрація копіювання списку та переміщення картки з завданням

Код на рис. 3.39 є серверним обробником для створення нового списку на вказаній дошці. Код перевіряє чи авторизований користувач. Якщо так, то

отримує title та boardId з вхідних даних. Потім проводить перевірку на існування дошки у базі даних і знаходить останній список у дошці та визначає новий порядок для нового списку. Потім створює новий список у базі даних із заданими даними і оновлює кеш для шляху /board/${boardId} за допомогою функції revalidatePath.

```
try {
  const board = await db.board.findUnique({
    where: { id: boardId},
  });

  if (!board) return { error: "Board not found"};

  const lastList = await db.list.findFirst({
    where: { boardId: boardId },
    orderBy: { order: "desc" },
    select: { order: true },
  });

  const newOrder = lastList ? lastList.order + 1 : 1;
  list = await db.list.create({ data: { title, boardId, order: newOrder} });

} catch (error) {
  return { error: "Failed to create." }
}

revalidatePath(`/board/${boardId}`);
return { data: list };
```

Рисунок 3.39 – Частина коду для створення нового списку

Фрагмент коду на рис. 3.40 призначений для створення копії існуючого списку разом із всіма його картками в базі даних.

```
list = await db.list.create({
  data: {
    boardId: listToCopy.boardId,
    title: `${listToCopy.title} - Copy`,
    order: newOrder,
    cards: {
      createMany: {
        data: listToCopy.cards.map((card) => ({
          title: card.title,
          description: card.description,
          order: card.order,
        })),
      },
    },
  },
  include: {
    cards: true,
  },
});
```

Рисунок 3.40 – Частина коду для копіювання списку

Новий список матиме таку саму дошку, як і оригінал, а також буде містити картки з такими самими заголовками, описами та порядковими номерами, як у оригіналі. Назва нового списку буде відрізнятися від оригіналу лише додаванням суфіксу «Copy».

Код на рис. 3.41 реалізує серверну функцію для видалення списку з певної дошки. Він перевіряє аутентифікацію користувача та видаляє відповідний список з бази даних.

```
const handler = async (data: InputType): Promise<ReturnType> => {
  const { userId } = auth();
  if (!userId) return { error: "Unauthorized." };
  const { id, boardId } = data;
  let list;

  try {
    list = await db.list.delete({
      where: { id, boardId,},
    });

  } catch (error) {
    return {
      error: "Failed to delete.",
    };
  }
  revalidatePath(`/board/${boardId}`);
  return { data: list };
};
```

Рисунок 3.41 – Частина коду для видалення списку

На рис. 3.42 наведено код для оновлення карток.

```
const handler = async (data: InputType): Promise<ReturnType> => {
    const { userId } = auth();
    if (!userId) return { error: "Unauthorized." };

    const { id, boardId, ...values } = data;
    let card;
    try {
      card = await db.card.update({
        where: { id, list: { board: {} }},
        data: {...values},
      });
    } catch (error) {
      return {
        error: "Failed to update."
      }
    }
    revalidatePath(`/board/${boardId}`);
    return { data: card };
  };
```

Рисунок 3.42 – Код для оновлення карток

Код для створення, копіювання та видалення карток дуже схожий на відповідний код для створення, копіювання та видалення списків. Відмінність лише в тому, що працюємо з моделю карток в базі даних.

На рис. 3.43 представлено вікно з інформацією про картку. В ньому користувачі можуть додавати опис, змінювати назву, копіювати та видаляти картку. Всі зміни пов’язані з карткою відображаються в секції Activity де видно інформацію про внесену зміну, ким та коли вона була зроблена.

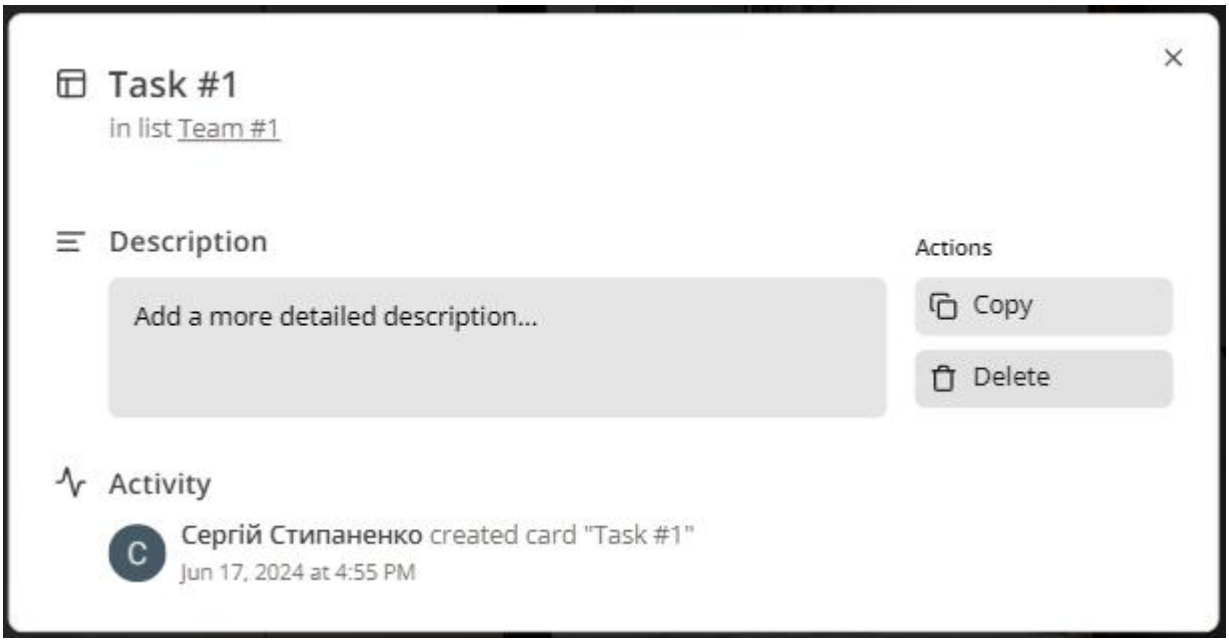

Рисунок 3.43 – Вікно з інформацією про картку

Файл schema.prisma було доповнено моделями Board (див. рис. 3.44), List (див. рис. 3.45) та Card (див. рис. 3.46).

```
model Board {
  id String @id @default(uuid())
  orgId String
  title String
  imageId String
  imageThumbUrl String @db.Text
  imageFullUrl String @db.Text
  imageUsername String @db.Text
  imageLinkHTML String @db.Text
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  lists List[]
}
```

Рисунок 3.44 – Модель Board

Модель Board призначена для представлення дошок (boards) в системі. Кожна дошка відноситься до конкретної організації (orgId) і має заголовок (title), id зображення (imageId), посилання на мініатюри (imageThumbUrl) та посилання на повне зображення (imageFullUrl). Кожна дошка має масив списків (lists), які включають картки (cards), що відносяться до цієї дошки. Поля createdAt і updatedAt відстежують час створення та оновлення дошки.

```
model List {
  id String @id @default(uuid())
  title String
  order Int
  boardId String
  board Board @relation(fields: [boardId], references: [id], onDelete: Cascade)

  cards Card[]

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  @@index([boardId])
}
```

Рисунок 3.45 – Модель List

Модель List використовується для представлення списків на дошці. Кожен список має унікальний ідентифікатор, назву, порядок відображення, а також посилання на дошку, до якої він відноситься. Список також включає масив карток (cards), які відносяться до цього списку. Поля createdAt і updatedAt відстежують час створення та оновлення списку.

```
model Card {
  id String @id @default(uuid())
  title String
  order Int
  description String? @db.Text
  listId String
  list List @relation(fields: [listId], references: [id], onDelete: Cascade)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  @@index([listId])
}
```

Рисунок 3.46 – Модель Card

Модель представляє окреме завдання або картку в списку. Кожна картка має

унікальний ідентифікатор, назву, опис, порядок відображення, а також посилання на список, до якого вона відноситься. Поля createdAt і updatedAt відстежують час створення та оновлення картки.

3.4.4 Реалізація комунікації між користувачами

Користувачі мають змогу спілкуватись різними способами. На рис. 3.47 предсталено спілкування у текстовому каналі. В цьому каналі всі учасники можуть відсилати повідомлення, фото, відео та файли. Користувачі також мають можливість редагувати та видаляти свої повідомлення. В модератора є можливість видаляти повідомлення інших учасників.

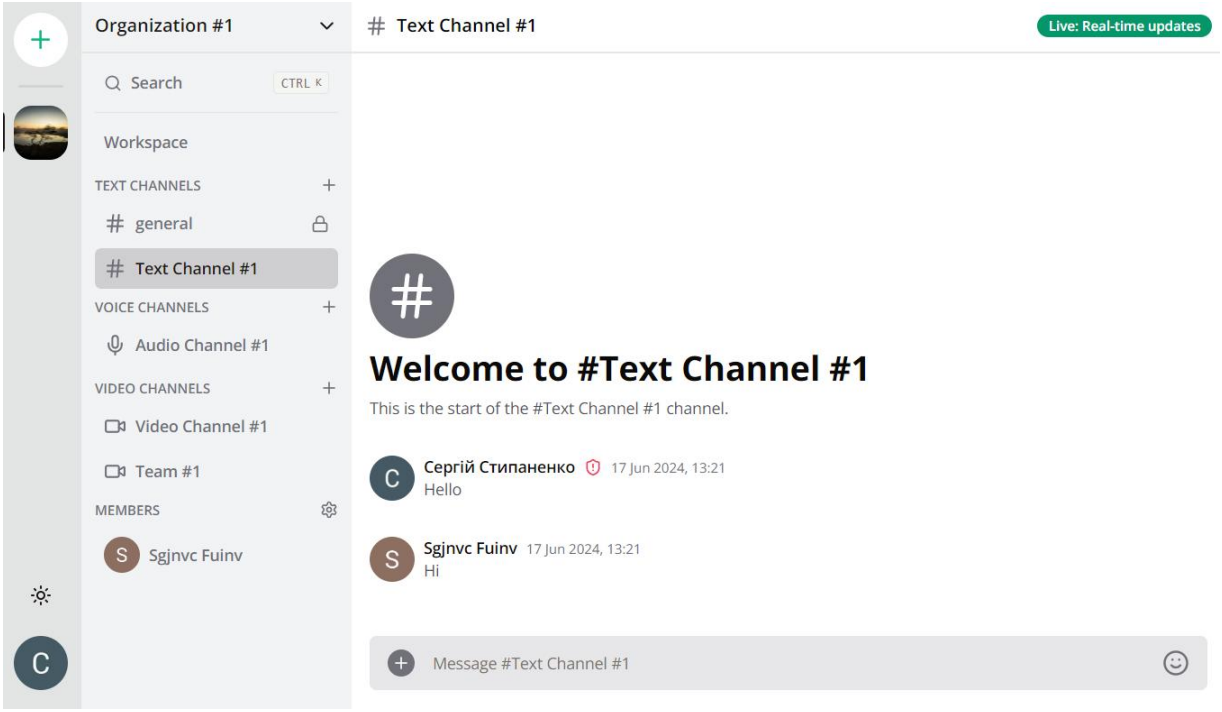

Рисунок 3.47 – Спілкування у текстовому каналі

Якщо повідомлення було відредаговано, то воно позначається відповідним підписом (див. рис. 3.48).

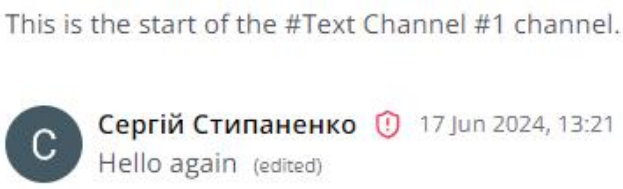

Рисунок 3.48 – Вигляд відредагованого повідомлення

На рис. 3.49 представлено спілкування один на один з іншим учасником. Користувачі можуть обмінюватись повідомленнями, файлами, фотографіями та відео так само як і в текстовому каналі. В приватному чаті з іншим учасником додатково є можливість відеодзвінка.

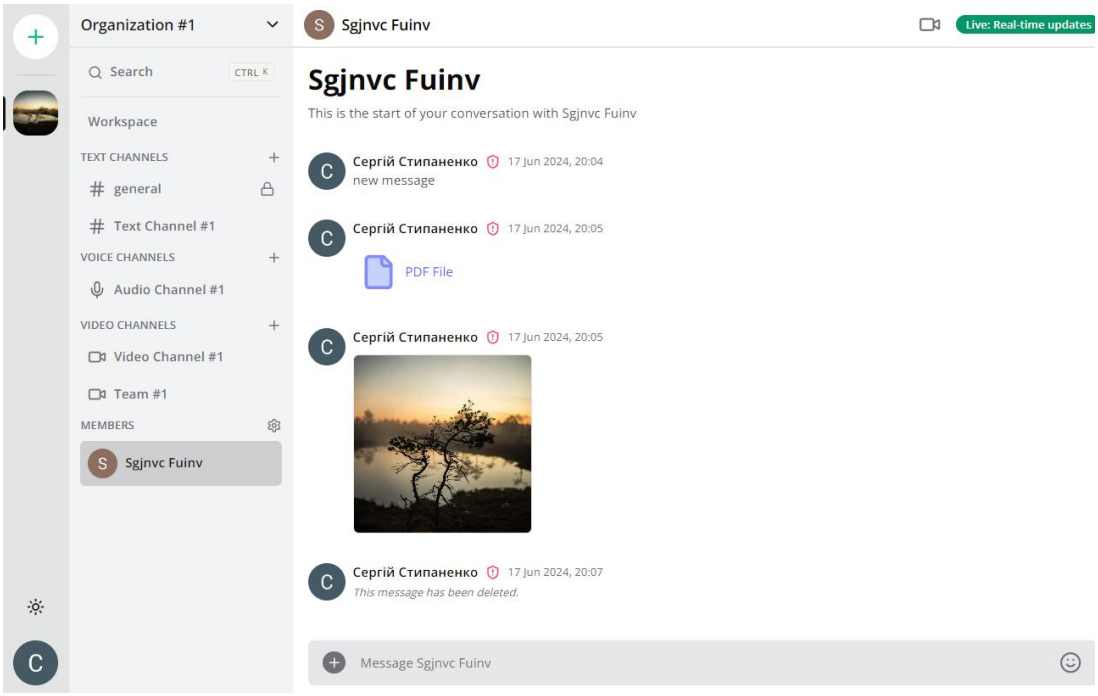

Рисунок 3.49 – Спілкування з іншим учасником один на один

На рис. 3.50 продемостровано зовнішній вигляд спілкування у відео каналі.

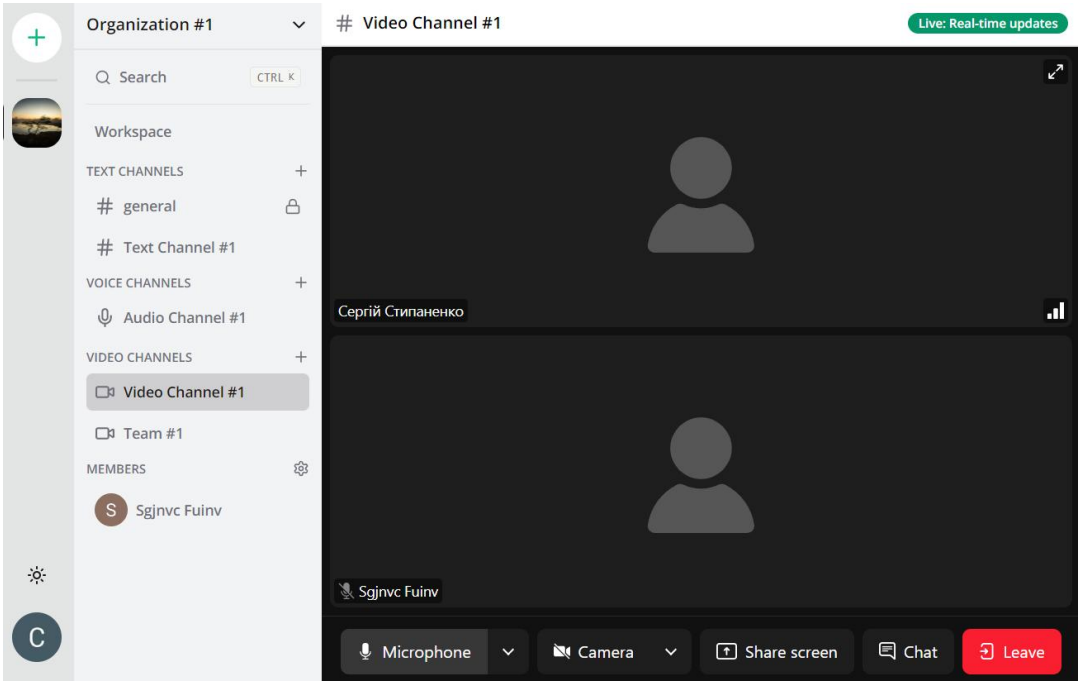

Рисунок 3.50 – Спілкування з іншим учасником у відео каналі

В ньому користувачі можуть спілкуватись використовуючи камери та мікрофон, демонструвати екран та спілкуватись у вбудованому чаті.

В аудіо каналі (див. рис. 3.51) користувачі можуть спілкуватись використовуючи лише мікрофон.

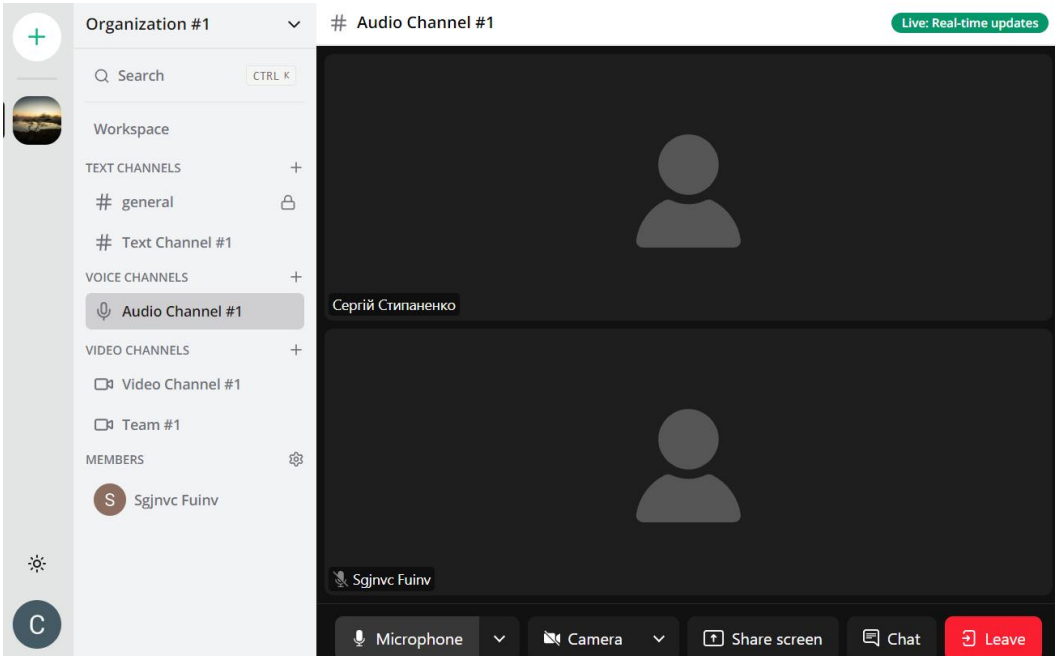

Рисунок 3.51 – Спілкування з іншим учасником у аудіо каналі

На рис. 3.51 бачимо функцію getOrCreateConversation, яка призначена для отримання існуючої розмови між двома користувачами або створення нової, якщо розмова не знайдена. Вона перевіряє наявність розмови за двома ідентифікаторами користувачів. Якщо розмова не знайдена, вона створює новий запис розмови у базі даних за допомогою функції createNewConversation. Як результат, функція повертає об'єкт розмови.

```
export const getOrCreateConversation = async (memberOneId:string, memberTwoId:string) =>{
    let conversation = await findConversation(memberOneId,memberTwoId)
                    || await findConversation(memberTwoId,memberOneId)

    if(!conversation) conversation = await createNewConversation(memberOneId,memberTwoId)
    return conversation;
}
```

Рисунок 3.51 – Функція getOrCreateConversation

Функція findConversation (див. рис 3.52) шукає розмову у базі даних за

двома ідентифікаторами користувачів. Вона використовує метод findFirst з об'єктом db.conversation, щоб знайти перший запис Якщо розмова знайдена, функція включає інформацію про обох учасників розмови та їх профілі, використовуючи опцію include. Якщо розмова не знайдена або виникає помилка, функція повертає null.

```
const findConversation = async (memberOneId:string, memberTwoId:string) => {
    try {
        return await db.conversation.findFirst({
            where:{
                AND:[{memberOneId:memberOneId},{memberTwoId:memberTwoId}]
            },
            include:{
                memberOne:{
                    include:{ profile:true }
                },
                memberTwo:{
                    include:{ profile:true }
                }
            }
        })
    } catch (error) {
        return null;
    }
}
```

Рисунок 3.52 – Функція findConversation

Функція createNewConversation на рис. 3.53 створює новий запис розмови у базі даних з заданими memberOneId і memberTwoId. Фунція створює новий запис у базі даних включаючи інформацію про обох учасників через опцію include. Якщо створення нової розмови вдається, функція повертає створений об'єкт розмови.

```
const createNewConversation = async (memberOneId:string, memberTwoId:string) => {
    try {
        return await db.conversation.create({
            data:{ memberOneId, memberTwoId},
            include:{
                memberOne:{
                    include:{ profile:true}
                },
                memberTwo:{
                    include:{ profile:true }
                }
            }
        })
    } catch (error) {
        return null;
    }
}
```

Рисунок 3.53 – Функція createNewConversation

Наведений на рис. 3.54 код обробляє GET запити щоб отримати повідомлення з бази даних відповідно до обраного каналу. Він забезпечує перевірку авторизації користувача, сортування повідомлень за датою та включає інформацію про користувачів, що створили повідомлення.

```
export async function GET(req:Request) {
    try {
        const profile = await currentProfile();
        const {searchParams} = new URL(req.url);
        const cursor = searchParams.get("cursor");
        const channelId = searchParams.get("channelId");

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!channelId) return new NextResponse("Channel ID Missing", {status:400});

        let messages: Message[] = []
        if(cursor) {
            messages = await db.message.findMany({
                take:MESSAGES_BATCH,
                skip:1,
                cursor: {id: cursor},
                where: {channelId},
                include:{ member: {include: {profile:true}}},
                orderBy: {createdAt: "desc"}
            })
        } else {
            messages = await db.message.findMany({
                take: MESSAGES_BATCH,
                where: {channelId},
                include: {member: {include: {profile: true}}},
                orderBy: {createdAt: "desc"}
            })
        }
        let nextCursor = null;
        if(messages.length === MESSAGES_BATCH) nextCursor = messages[MESSAGES_BATCH - 1].id

        return NextResponse.json({items:messages,nextCursor})
    } catch (error) {
        return new NextResponse("Internal Error", {status:500});
    }
}
```

Рисунок 3.54 – Функція для обробки GET запитів

Код на рис. 3.55 об'єднує різні аспекти для ефективної обробки та відправки повідомлень у реальному часі через вебсокети. Він забезпечує обробку POST запитів для створення нових повідомлень в чаті. При цьому перевіряючи права доступу та валідацію вхідних даних та використання веб-сокетів для миттєвої передачі повідомлень всім учасникам чату.

```
export default async function handler (req:NextApiRequest, res:NextApiResponseServerIo) {
    if(req.method !== "POST") return res.status(405).json({error: "Method not allowed"});

    try {
        const profile = await currentProfilePages(req);
        const {content, fileUrl} = req.body;
        const {serverId, channelId} = req.query;

        if(!profile) return res.status(401).json({error: "Unauthorized"});
        if(!serverId) return res.status(400).json({error: "Server ID Missing"});
        if(!channelId) return res.status(400).json({error: "Channel ID Missing"});
        if(!content) return res.status(400).json({error: "Content Missing"});

        const server = await db.server.findFirst({
            where:{id: serverId as string, members: { some: {profileId: profile.id}}},
            include: { members: true}
        })
        if(!server) return res.status(404).json({error: "Server not found"});

        const channel = await db.channel.findFirst({
            where:{ id: channelId as string, serverId: serverId as string}
        })
        if(!channel) return res.status(404).json({error: "Channel not found"});

        const member = server.members.find((member) => member.profileId === profile.id);
        if(!member) return res.status(404).json({error: "Member not found"});

        const message = await db.message.create({
            data: {
                content,
                fileUrl,
                channelId: channelId as string,
                memberId: member.id
            },
            include: { member: { include: { profile: true }}}
        })

        const channelKey = `chat:${channelId}:messages`;
        res?.socket?.server?.io?.emit(channelKey, message);

        return res.status(200).json(message);
    } catch (error) {
        return res.status(500).json({message: "Internal Error"})
    }
}
```

Рисунок 3.55 – Функція для обробки POST запитів повідомлень

Предаставлений на рис. 3.56 код є призначений для обробки DELETE і PATCH запитів зі сторони клієнта. Він дозволяє користувачам застосунку видаляти або редагувати їх власні повідомлення в каналах чату та забезпечує оновлення повідомлень в реальному часі використовуючи вебсокет.

```
const isMessageOwner = message.memberId === member.id;
const isAdmin = member.role === MemberRole.ADMIN;
const isModerator = member.role === MemberRole.MODERATOR;
const canModify = isMessageOwner || isAdmin || isModerator;

if (!canModify) return res.status(401).json({ error: "Unauthorized" });

if (req.method === "DELETE") {
    message = await db.message.update({
      where: { id: messageId as string },
      data: { fileUrl: null, content: "This message has been deleted.", deleted: true },
      include: { member: { include: { profile: true }}}
    });
}

if (req.method === "PATCH") {
    if (!isMessageOwner) return res.status(401).json({ error: "Unauthorized" });
    message = await db.message.update({
      where: { id: messageId as string },
      data: { content },
      include: { member: { include: { profile: true }}}
    });
}

const updateKey = `chat:${channelId}:messages:update`;
res?.socket?.server?.io?.emit(updateKey, message);
return res.status(200).json(message);
```

Рисунок 3.56 – Функція для обробки POST запитів повідомлень

Після видалення або оновлення повідомлення воно відправляється користувачам використовуючи функцію res?.socket?.server?.io?.emit(updateKey, message). Це відправить повідомлення до всіх підключених клієнтів, які перебувають у відповідному каналі.

Висновки до розділу 3

У цьому розділі було детально розглянуто та виконано ключові аспекти розробки та реалізації функціональності застосунку. Було налаштовано необхідне середовище для розробки, визначено структуру проекту та налаштовано необхідні інструменти. Реалізація авторизації забезпечила безпечний доступ користувачів до системи, що є критичним для такого роду застосунку.

Також було реалізовано взаємодію з базою даних, що включало зберігання та управління даними користувачів, а також іншими даними, необхідними для функціонування застосунку. Використання MySQL у поєднанні з Prisma та Aiven

дозволило забезпечити надійність та ефективність роботи з даними.

Створення серверної частини та інтерфейсу користувача включало реалізацію ключових компонентів, таких як вікно першого входу, бічна панель навігації, робочий простір організації та комунікація між користувачами. Розроблений за допомогою React, Tailwind та Shadcn/ui інтерфейс, забезпечив зручність та інтуїтивну зрозумілість для користувачів.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено застосунок для ефективної комунікації та організації роботи в колективі. Для досягнення поставленої мети було виконано наступні пункти:

1) проаналізовано вже існуючі рішення;
2) обрано засоби та інструменти розробки програмного продукту;
3) здійснено реалізацію застосунку.

Також було виконано поставлені завдання, які включали в себе:

1) розробку та імплементацію основних функцій, таких як обмін повідомленнями в реальному часі, можливість створення голосових та відеовикликів, організація дошок, списків та завдань;

2) забезпечення надійності та безпеки застосунку, включаючи аутентифікацію користувачів та управління правами доступу;

3) розробку зручного та інтуїтивно зрозумілого інтерфейсу користувача, який би дозволив легко орієнтуватися в застосунку та швидко виконувати необхідні дії.

У першому розділі було здійснено огляд існуючих програмних рішень для комунікації та організації роботи, включаючи такі популярні інструменти, як Discord, Trello, Microsoft Teams, Skype та Zoom. Було визначено, що ринок таких застосунків є висококонкурентним, проте існує потреба у нових рішеннях, які враховують зростаючі вимоги до зручності та функціональності. Було зроблено висновок, що успішна розробка нового застосунку повинна базуватися на детальному розумінні потреб користувачів та тенденцій ринку.

У другому розділі було обґрунтовано вибір технологій для розробки застосунку. Було вирішено використовувати Visual Studio Code як основний редактор коду завдяки його функціональності та зручності. Для розробки інтерфейсу було обрано React у поєднанні з TailwindCSS, що дозволило створити сучасний і зручний інтерфейс. Серверна частина була реалізована на базі Next.js, та Typescript, що забезпечило швидку та надійну обробку запитів. Базою даних

обрано MySQL у поєднанні з Prisma та платформою Aiven, що гарантує високу надійність і продуктивність. Було також інтегровано систему аутентифікації Clerk, яка забезпечує безпеку та зручність користувачів.

У третьому розділі було виконано детальну реалізацію основних функцій застосунку. Було налаштовано середовище для розробки, визначено структуру проєкту та реалізовано основні функціональні можливості. Реалізовано систему авторизації, що забезпечила безпечний доступ користувачів до системи. Здійснено інтеграцію з базою даних для надійного зберігання та управління даними. Також було створено інтерфейс користувача, включаючи вікно першого входу, бічну панель навігації, робочий простір організації та засоби комунікації між користувачами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Alex Banks, Eve Porcello, Learning React: Functional Web Development with React and Redux. 1st Edition. O'Reilly Media, 2017. 243 p.

2. Boris Cherny, Programming TypeScript: Making Your JavaScript Applications Scale. 1st Edition. O'Reilly Media, 2019. 322 p.

3. Jiho Seok, TypeScript Mastery: A Comprehensive Guide to Robust and Maintainable Web Applications, Independently published, 2023. 155 p.

4. Greg Lim, NextJS 13 and React Crash Course: Build a Full Stack NextJS 13 App with React, Tailwind and Prisma backend, Independently published, 2023. 135 p.

5. Franco Nicolás Coronel. Building the Future: Next.js in Action: Mastering Next.js with Tailwind CSS, Prisma, and Socket.io, 2023. 356 p.

6. Robin Wieruch, The Road to React: The React.js with Hooks in JavaScript Book, 2017. 412 p.

7. Kartik Bhat, Ultimate Tailwind CSS Handbook: Build sleek and modern websites with immersive UIs using Tailwind CSS, 2023. 389 p.

8. Grace Huang, Dynamic Trio: Building Web Applications with React, Next.js & Tailwind, 2023. 232 p.

9. Noel Rappin, Modern CSS with Tailwind. Pragmatic Bookshelf, 2022. 168 p.

10. Tejas Kumar, Fluent React. O'Reilly Media, 2024. 541 p.

11. Zac Gordon, React Explained: Your Step-by-Step Guide to React. OS Training, LLC, 2019. 368 p.

12. Tyson Cadenhead, Socket.IO Cookbook. Packt Publishing, 2015. 186 p.

13. Franco Nicolás Coronel, Building the Future: Next.js in Action: Mastering Next.js with Tailwind CSS, Prisma, and Socket.io, 2023. 356 p.

14. Carl Rippon, Learn React with TypeScript: A beginner's guide to reactive web development with React 18 and TypeScript. Packt Publishing, 2023. 474 p.

15. Maximilian Schwarzmüller, React Key Concepts: Consolidate your knowledge of React’s core features. Packt Publishing, 2022. 590 p.

16. Greg Lim, Beginning React (incl. Redux and React Hooks), 2017. 166 p.

17. React Get Started: вебсайт. URL: https://react.dev/learn (дата звернення: 28.02.2024)

18. Get started with Tailwind CSS: вебсайт. URL: https://tailwindcss.com/docs/installation (дата звернення: 02.03.2024)

19. Shadcn/ui Docs: вебсайт. URL: https://ui.shadcn.com/docs (дата звернення: 05.03.2024).

20. Next.js Docs: вебсайт. URL: https://nextjs.org/docs (дата звернення: 15.03.2024).

21. Typescript Docs: вебсайт. URL: https://www.typescriptlang.org/docs/ (дата звернення: 26.03.2024).

22. Clerk Documentation: вебсайт. URL: https://clerk.com/docs (дата звернення: 04.04.2024).

23. Socket.IO Docs: вебсайт. URL: https://socket.io/docs/v4/ (дата звернення: 08.04.2024).

24. Prisma Documentation: вебсайт. URL: https://www.prisma.io/docs (дата звернення: 09.04.2024).

25. Aiven Documentation: вебсайт. URL: https://aiven.io/docs (дата звернення: 13.04.2024).

ДОДАТОК А

Порівняльна таблиця існуючих програмних рішень

Таблиця А.1 – Порівняльна таблиця існуючих програмних рішень

Назва	Функціональні можливості	Переваги	Недоліки
Discord	1. Текстові канали для обговорень та спілкування. 2. Голосові та відео-чати, демонстрація екрану під час зустрічей. 3. Можливість створення серверів для організації спільнот та проектів. 4. Інтеграція з різними ігровими платформами та сервісами.	1. Простий та зрозумілий інтерфейс. 2. Можливості голосового та відео спілкування. 3. Велика кількість налаштувань для керування серверами та каналами.	Головним направленням Discord є геймери, тому корпоративні функції менш розвинені.
Trello	1. Створення проектів та дошок для організації завдань. 2. Додавання завдань у вигляді карточок, які можна переміщувати між колонками. 3. Можливість додавати до карточок коментарі, прикріплювати файли, встановлювати терміни виконання та інші деталі.	1. Простий та інтуїтивно зрозумілий інтерфейс. 2. Гнучка система організації завдань.	1. Обмежена можливість налаштувань у безкоштовній версії.
Microsoft Teams	1. Інтеграція з іншими сервісами microsoft. 2. Можливість створення команд та каналів для спілкування та спільної роботи. 3. Відеоконференції та демонстрація екрану під час зустрічей.	1. Глибока інтеграція з іншими продуктами Microsoft 2. Широкий функціонал для комунікації та спільної роботи.	1. Складність використання для користувачів, які не знайомі з екосистемою Microsoft 2. Потреба у використанні інших продуктів Microsoft для повного функціоналу
Skype	1. Голосове та відео спілкування в одиночному та груповому режимах. 2. Обмін текстовими повідомленнями. 3. Можливість відправки вкладень. 4. Інтеграція з microsoft-акаунтами. 5. Використання на різних платформах: комп'ютери, мобільні пристрої, планшети.	1. Простота у використанні. 2. Популярність. 3. Можливість використання на різних платформах.	1. Обмежена функціональність у порівнянні з іншими сервісами; 2. Не глибока інтеграція з іншими продуктами
Zoom	1. Голосове, відео та текстове спілкування. 2. Відеоконференції з можливістю додавання багатьох учасників; 3. Ділитися екраном та файли під час відеоконференцій; 4. Запис відеоконференцій. 5. Мобільні додатки.	1. Простий та зручний інтерфейс. 2. Велика кількість учасників у відеоконференції. 3. Можливість запису та архівування зустрічей.	1. Обмеження у безкоштовній версії. 2. Питання щодо безпеки даних та конфіденційності

ДОДАТОК Б

Відношення між таблицями бази даних

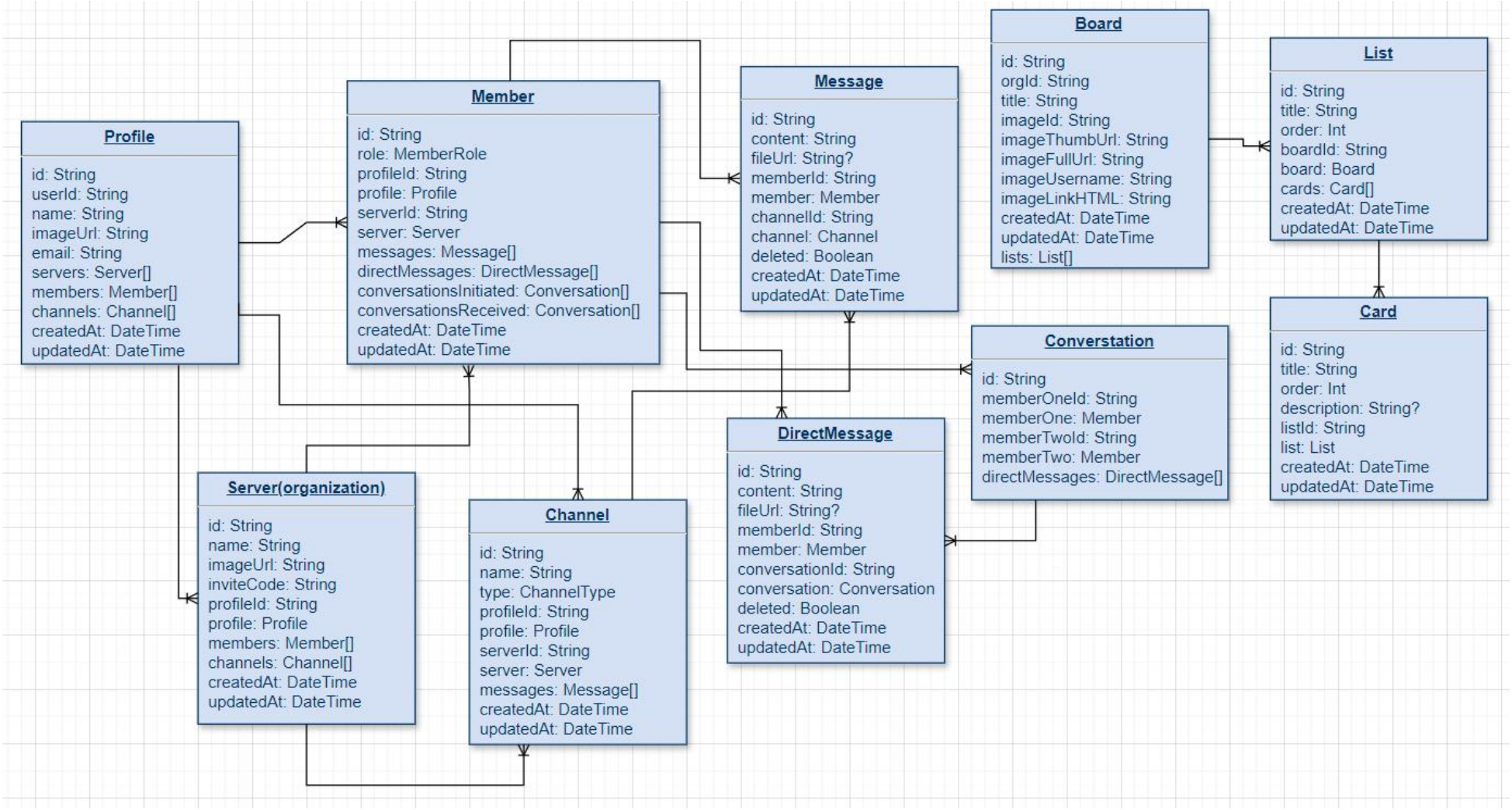

Рисунок Б.1 – Відношення між моделями (таблицями) бази даних

ДОДАТОК В

Програмний код застосунку

conversation.ts

```
import { db } from "./db";

export const getOrCreateConversation = async (memberOneId:string, memberTwoId:string) =>{
    let conversation = await findConversation(memberOneId,memberTwoId)
                    || await findConversation(memberTwoId,memberOneId)

    if(!conversation) conversation = await createNewConversation(memberOneId,memberTwoId)
    return conversation;
}

const findConversation = async (memberOneId:string, memberTwoId:string) => {
    try {
        return await db.conversation.findFirst({
            where:{
                AND:[{memberOneId:memberOneId},{memberTwoId:memberTwoId}]
            },
            include:{
                memberOne:{
                    include:{ profile:true }
                },
                memberTwo:{
                    include:{ profile:true }
                }
            }
        })
    } catch (error) {
        return null;
    }
}

const createNewConversation = async (memberOneId:string, memberTwoId:string) => {
    try {
        return await db.conversation.create({
            data:{ memberOneId, memberTwoId},
            include:{
                memberOne:{
                    include:{ profile:true}
                },
                memberTwo:{
                    include:{ profile:true }
                }
            }
        })
    } catch (error) {
        return null;
    }
}
```

create-audit-log.ts

```
import { auth, currentUser } from "@clerk/nextjs/server";
import { ACTION, ENTITY_TYPE } from "@prisma/client";

import { db } from "@/lib/db";

interface Props {
  entityId: string;
  entityType: ENTITY_TYPE,
```

```
  entityTitle: string;
  action: ACTION;
};

export const createAuditLog = async (props: Props) => {
  try {
    const user = await currentUser();
    if (!user) throw new Error("User not found!");
    const { entityId, entityType, entityTitle, action } = props;

    await db.auditLog.create({
      data: { entityId, entityType, entityTitle, action,
        userId: user.id,
        userImage: user?.imageUrl,
        userName: user?.firstName + " " + user?.lastName,
      }
    });
  } catch (error) {
    console.log("[AUDIT_LOG_ERROR]", error);
  }
}
```

current-profile.ts

```
import { db } from "./db";
import { auth } from "@clerk/nextjs/server";

export const currentProfile = async () => {
    const {userId} = auth();
    if(!userId){
        return null;
    }

    const profile = await db.profile.findUnique({
        where:{
            userId
        }
    })

    return profile;
}
```

initial-profile.ts

```
import { currentUser } from "@clerk/nextjs/server";
import { auth } from "@clerk/nextjs/server";
import { db } from "@/lib/db";

export const initialProfile = async () => {
    const user = await currentUser();

    if(!user) return auth().redirectToSignIn()

    const profile = await db.profile.findUnique({
        where: {
            userId:user.id
        }
    })

    if (profile) return profile;

    const newProfile = await db.profile.create({
        data: {
            userId:user.id,
```

```
            name: `${user.firstName} ${user.lastName}`,
            imageUrl: user.imageUrl,
            email: user.emailAddresses[0].emailAddress
        }
    })

    return newProfile;
}
```

generate-log-messages.ts

```
import { ACTION, AuditLog } from "@prisma/client";

export const generateLogMessage = (log: AuditLog) => {
  const { action, entityTitle, entityType } = log;

  switch (action) {
    case ACTION.CREATE:
      return `created ${entityType.toLowerCase()} "${entityTitle}"`;
    case ACTION.UPDATE:
      return `updated ${entityType.toLowerCase()} "${entityTitle}"`;
    case ACTION.DELETE:
      return `deleted ${entityType.toLowerCase()} "${entityTitle}"`;
    default:
      return `unknown action ${entityType.toLowerCase()} "${entityTitle}"`;
  };
};
```

upload-thing.ts

```
import {
    generateUploadButton,
    generateUploadDropzone,
  } from "@uploadthing/react";

  import type { OurFileRouter } from "@/app/api/uploadthing/core";

  export const UploadButton = generateUploadButton<OurFileRouter>();
  export const UploadDropzone = generateUploadDropzone<OurFileRouter>();
```

api/messages/route.ts

```
import { currentProfile } from "@/lib/current-profile";
import { db } from "@/lib/db";
import { Message } from "@prisma/client";
import { NextResponse } from "next/server";

const MESSAGES_BATCH = 10;

export async function GET(req:Request) {
    try {
        const profile = await currentProfile();
        const {searchParams} = new URL(req.url);
        const cursor = searchParams.get("cursor");
        const channelId = searchParams.get("channelId");

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!channelId) return new NextResponse("Channel ID Missing", {status:400});

        let messages: Message[] = []
        if(cursor) {
            messages = await db.message.findMany({
                take:MESSAGES_BATCH,
                skip:1,
                cursor: {id: cursor},
```

```
                where: {channelId},
                include:{ member: {include: {profile:true}}},
                orderBy: {createdAt: "desc"}
            })
        } else {
            messages = await db.message.findMany({
                take: MESSAGES_BATCH,
                where: {channelId},
                include: {member: {include: {profile: true}}},
                orderBy: {createdAt: "desc"}
            })
        }
        let nextCursor = null;
        if(messages.length === MESSAGES_BATCH) nextCursor = messages[MESSAGES_BATCH - 1].id

        return NextResponse.json({items:messages,nextCursor})
    } catch (error) {
        return new NextResponse("Internal Error", {status:500});
    }
}
```

api/servers/route.ts

```
import { db } from "@/lib/db";
import { currentProfile } from "@/lib/current-profile";
import { NextResponse } from "next/server";

import {v4 as uuidv4} from "uuid";
import { MemberRole } from "@prisma/client";

export async function POST(req:Request) {
    try {
        const {name, imageUrl} = await req.json();
        const profile = await currentProfile();

        if(!profile){
            return new NextResponse("Unauthorized", {status:401});
        }

        const room = await db.server.create({
            data:{
                profileId:profile.id,
                name,
                imageUrl,
                inviteCode: uuidv4(),
                channels: {
                    create: [
                        {name: "general", profileId:profile.id}
                    ]
                },
                members: {
                    create: [
                        {profileId:profile.id, role: MemberRole.ADMIN}
                    ]
                }
            }
        })
        return NextResponse.json(room);
    } catch (error) {
        console.log("[ROOM_POST    ]", error);
        return new NextResponse("Internal Error", {status:500});
    }
}
```

api/servers/leave/route.ts

```
import { currentProfile } from "@/lib/current-profile";
import { db } from "@/lib/db";
import { NextResponse } from "next/server";

export async function PATCH(req:Request,{params}:{params:{serverId:string}}){
    try {
        const profile = await currentProfile();

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!params.serverId) return new NextResponse("Server ID missing", {status:400});

        const server = await db.server.update({
            where: {
                id:params.serverId,
                profileId: {
                    not: profile.id
                },
                members:{
                    some: {
                        profileId: profile.id
                    }
                }
            },
            data:{
                members:{
                    deleteMany: {
                        profileId:profile.id
                    }
                }
            }
        })
        return NextResponse.json(server);

    } catch (error) {
        console.log("[SERVER_ID_LEAVE]", error);
        return new NextResponse("Internal Error", {status:500})
    }
}
```

api/uploadthing/core.ts

```
import { createUploadthing, type FileRouter } from "uploadthing/next";
import { auth } from "@clerk/nextjs/server";
const f = createUploadthing();

const handleAuth = () => {
      const {userId} = auth();
      if(!userId) throw new Error("Unauthorized");
      return {userId:userId}
}
export const ourFileRouter = {
      serverImage: f({image:{maxFileSize:"8MB", maxFileCount:1}})
            .middleware(()=>handleAuth())
            .onUploadComplete(()=>{}),
      messageFile: f(["image", "pdf", "video","application/vnd.ms-word.document.macroenabled.12"])
            .middleware(()=>handleAuth())
            .onUploadComplete(()=>{})
} satisfies FileRouter;

export type OurFileRouter = typeof ourFileRouter;
```

api/messages/route.ts

```
import { currentProfile } from "@/lib/current-profile";
import { db } from "@/lib/db";
import { Message } from "@prisma/client";
import { NextResponse } from "next/server";

const MESSAGES_BATCH = 10;

export async function GET(req:Request) {
    try {
        const profile = await currentProfile();
        const {searchParams} = new URL(req.url);
        const cursor = searchParams.get("cursor");
        const channelId = searchParams.get("channelId");

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!channelId) return new NextResponse("Channel ID Missing", {status:400});

        let messages: Message[] = []
        if(cursor) {
            messages = await db.message.findMany({
                take:MESSAGES_BATCH,
                skip:1,
                cursor: {id: cursor},
                where: {channelId},
                include:{ member: {include: {profile:true}}},
                orderBy: {createdAt: "desc"}
            })
        } else {
            messages = await db.message.findMany({
                take: MESSAGES_BATCH,
                where: {channelId},
                include: {member: {include: {profile: true}}},
                orderBy: {createdAt: "desc"}
            })
        }
        let nextCursor = null;
        if(messages.length === MESSAGES_BATCH) nextCursor = messages[MESSAGES_BATCH - 1].id

        return NextResponse.json({items:messages,nextCursor})
    } catch (error) {
        return new NextResponse("Internal Error", {status:500});
    }
}
```

api/members/route.ts

```
import { currentProfile } from "@/lib/current-profile";
import { db } from "@/lib/db";
import { NextResponse } from "next/server";

export async function PATCH(req:Request, {params}: {params:{memberId:string}}) {
    try {
        const profile = await currentProfile();
        const {searchParams} = new URL(req.url);
        const {role} = await req.json();
        const serverId = searchParams.get("serverId");

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!serverId) return new NextResponse("Server ID missing", {status:400});
        if(!params.memberId) return new NextResponse("Member ID missing", {status:400});
```

```
        const server = await db.server.update({
            where:{
                id: serverId,
                profileId: profile.id
            },
            data: {
                members:{
                    update: {
                        where: {
                            id:params.memberId,
                            profileId:{ not: profile.id }
                        },
                        data:{ role }
                    }
                }
            },
            include: {
                members: {
                    include: {   profile: true },
                    orderBy: { role: "asc" }
                }
            }
        })

        return NextResponse.json(server);

    } catch (error) {
        console.log("[MEMBER_ID_PATCH]",error);
        return new NextResponse("Internal Error", {status:500})
    }
}

export async function DELETE(req:Request, {params}: {params:{memberId:string}}) {
    try {
        const profile = await currentProfile();
        const {searchParams} = new URL(req.url);
        const serverId = searchParams.get("serverId");

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!serverId) return new NextResponse("Server ID missing", {status:400});
        if(!params.memberId) return new NextResponse("Member ID missing", {status:400});

        const server = await db.server.update({
            where: {
                id: serverId,
                profileId: profile.id,
            },
            data: {
                members: {
                    deleteMany: {
                        id: params.memberId,
                        profileId: { not: profile.id }
                    }
                }
            },
            include: {
                members: {
                    include: { profile: true },
                    orderBy: { role: "asc" }
                }
            }
        })
```

```
        return NextResponse.json(server);
    } catch (error) {
        console.log("[MEMBER_ID_DELETE]",error);
        return new NextResponse("Internal Error", {status:500})
    }
}
```

api/channels/route.ts

```
import { currentProfile } from "@/lib/current-profile";
import { db } from "@/lib/db";
import { MemberRole } from "@prisma/client";
import { NextResponse } from "next/server";

export async function POST(req:Request) {
    try {
        const profile = await currentProfile();
        const {name, type} = await req.json();
        const {searchParams} = new URL(req.url);

        const serverId = searchParams.get("serverId");
        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!serverId) return new NextResponse("Server ID missing", {status:400});

        if(name === "general") return new NextResponse("Name cannot be 'general'", {status:400});

        const server = await db.server.update({
            where:{
                id: serverId,
                members: {
                    some: { profileId: profile.id, role: {in: [MemberRole.ADMIN, MemberRole.MODERATOR]}}
                }
            },
            data: {channels: { create: { profileId: profile.id, name, type}}}
        });
        return NextResponse.json(server);
    } catch (error) {
        return new NextResponse("Internal Error", {status:500});
    }
}
```

api/channels/[channelId]/route.ts

```
import { currentProfile } from "@/lib/current-profile";
import { NextResponse } from "next/server";
import { db } from "@/lib/db";
import { MemberRole } from "@prisma/client";

export async function DELETE(req:Request, {params}:{params:{channelId:string}}) {
    try {
        const profile = await currentProfile();
        const {searchParams} = new URL(req.url);
        const serverId = searchParams.get("serverId");

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!serverId) return new NextResponse("Server ID missing", {status:400});
        if(!params.channelId) return new NextResponse("Channel ID missing", {status:400});

        const server = await db.server.update({
            where:{
                id: serverId,
                members: {
                    some: {
                        profileId: profile.id,
```

```
                        role: {
                            in: [MemberRole.ADMIN, MemberRole.MODERATOR]
                        }
                    }
                }
            },
            data: {
                channels: {
                    delete: {
                        id: params.channelId,
                        name: {
                            not: "general"
                        }
                    }
                }
            }
        });

        return NextResponse.json(server);

    } catch (error) {
        console.log("[CHANNEL_ID_DELETE]", error);
        return new NextResponse("Internal Error", {status:500});
    }
}

export async function PATCH(req:Request, {params}:{params:{channelId:string}}) {
    try {
        const profile = await currentProfile();
        const {name,type} = await req.json();
        const {searchParams} = new URL(req.url);
        const serverId = searchParams.get("serverId");

        if(!profile) return new NextResponse("Unauthorized", {status:401});
        if(!serverId) return new NextResponse("Server ID missing", {status:400});
        if(!params.channelId) return new NextResponse("Channel ID missing", {status:400});
        if(name === "general") return new NextResponse("Name cannot be 'general'", {status:400})

        const server = await db.server.update({
            where:{
                id: serverId,
                members: {
                    some: {
                        profileId: profile.id,
                        role: {
                            in: [MemberRole.ADMIN, MemberRole.MODERATOR]
                        }
                    }
                }
            },
            data: {
                channels: {
                    update: {
                        where: {
                            id: params.channelId,
                            NOT: {
                                name: "general"
                            }
                        },
                        data:{
                            name,type
                        }
```

```
                    }
                }
            }
        });

        return NextResponse.json(server);

    } catch (error) {
        console.log("[CHANNEL_ID_PATCH]", error);
        return new NextResponse("Internal Error", {status:500});
    }
}
```

api/cards/route.ts

```
import { auth } from "@clerk/nextjs/server";
import { NextResponse } from "next/server";

import { db } from "@/lib/db";

export async function GET(
  req: Request,
  { params }: { params: { cardId: string } }
) {
  try {
    const { userId} = auth();

    if (!userId) {
      return new NextResponse("Unauthorized", { status: 401 });
    }

    const card = await db.card.findUnique({
      where: {
        id: params.cardId,
        list: {

        },
      },
      include: {
        list: {
          select: {
            title: true,
          },
        },
      },
    });

    return NextResponse.json(card);
  } catch (error) {
    return new NextResponse("Internal Error", { status: 500 });
  }
}
```

api/cards/[card]/route.ts

```
import { auth } from "@clerk/nextjs/server";
import { NextResponse } from "next/server";
import { ENTITY_TYPE } from "@prisma/client";

import { db } from "@/lib/db";

export async function GET(
  request: Request,
```

```
  { params }: { params: { cardId: string } }
) {
  try {
    const { userId } = auth();

    if (!userId) {
      return new NextResponse("Unauthorized", { status: 401 });
    }

    const auditLogs = await db.auditLog.findMany({
      where: {
        entityId: params.cardId,
        entityType: ENTITY_TYPE.CARD,
      },
      orderBy: {
        createdAt: "desc",
      },
      take: 3,
    });

    return NextResponse.json(auditLogs);
  } catch (error) {
    return new NextResponse("Internal Error", { status: 500 });
  }
};
```

create-board/index.ts

```
"use server";
import { auth } from "@clerk/nextjs/server";
import { InputType, ReturnType } from "./types";
import { db } from "@/lib/db";
import { revalidatePath } from "next/cache";
import { createSafeAction } from "@/lib/create-safe-action";
import { CreateBoardSchema } from "./schema";
import { createAuditLog } from "@/lib/create-audit-log";
import { ACTION, ENTITY_TYPE } from "@prisma/client";

interface CombinedInput {
  data: {
    title: string;
    image: string;
  };
  serverId: string;
}
const handler = async (data: InputType): Promise<ReturnType> => {
  const { userId, orgId } = auth();

  if (!userId) {
    return {
      error: "Unauthorized.",
    };
  }

  const { title, image } = data;
  const [imageId, imageThumbUrl, imageFullUrl, imageLinkHTML, imageUsername] = image.split("|");
  console.log({
    imageId, imageThumbUrl, imageFullUrl, imageLinkHTML, imageUsername
  });

  if (!imageId || !imageThumbUrl || !imageFullUrl || !imageLinkHTML || !imageUsername) {
    return {
      error: "Missing fields. Failed to create board.",
```

```
    };
  }

  let board;
  try {
    board = await db.board.create({
      data: {
        orgId: "Test3",
        title,
        imageId,
        imageThumbUrl,
        imageFullUrl,
        imageUsername,
        imageLinkHTML
      },
    });
    await createAuditLog({
      entityTitle: board.title,
      entityId: board.id,
      entityType: ENTITY_TYPE.BOARD,
      action: ACTION.CREATE,
    })
  } catch (error) {
    return {
      error: "Failed to create board.",
    };
  }

  revalidatePath(`/board/${board.id}`);
  return {
    data: board,
  };
}

export const createBoard = createSafeAction(CreateBoardSchema, handler);
```

create-list/index.ts

```
"use server";

import { auth } from "@clerk/nextjs/server";
import { revalidatePath } from "next/cache";

import { db } from "@/lib/db";
import { createSafeAction } from "@/lib/create-safe-action";

import { CreateList } from "./schema";
import { InputType, ReturnType } from "./types";
import { ACTION, ENTITY_TYPE } from "@prisma/client";
import { createAuditLog } from "@/lib/create-audit-log";

const handler = async (data: InputType): Promise<ReturnType> => {
  const { userId} = auth();

  if (!userId) {
    return {
      error: "Unauthorized",
    };
  }

  const { title, boardId } = data;
  let list;
```

```
  try {
    const board = await db.board.findUnique({
      where: {
        id: boardId,
      },
    });

    if (!board) {
      return {
        error: "Board not found",
      };
    }

    const lastList = await db.list.findFirst({
      where: { boardId: boardId },
      orderBy: { order: "desc" },
      select: { order: true },
    });

    const newOrder = lastList ? lastList.order + 1 : 1;

    list = await db.list.create({
      data: {
        title,
        boardId,
        order: newOrder,
      },
    });

    await createAuditLog({
      entityTitle: list.title,
      entityId: list.id,
      entityType: ENTITY_TYPE.LIST,
      action: ACTION.CREATE,
    })
  } catch (error) {
    return {
      error: "Failed to create."
    }
  }

  revalidatePath(`/board/${boardId}`);
  return { data: list };
};

export const createList = createSafeAction(CreateList, handler);
```

create-card/index.ts

```
"use server";

import { auth } from "@clerk/nextjs/server";
import { revalidatePath } from "next/cache";

import { db } from "@/lib/db";
import { createSafeAction } from "@/lib/create-safe-action";

import { CreateCard } from "./schema";
import { InputType, ReturnType } from "./types";
import { createAuditLog } from "@/lib/create-audit-log";
import { ACTION, ENTITY_TYPE } from "@prisma/client";

const handler = async (data: InputType): Promise<ReturnType> => {
```

```
  const { userId} = auth();

  if (!userId) {
    return {
      error: "Unauthorized",
    };
  }

  const { title, boardId, listId } = data;
  let card;

  try {
    const list = await db.list.findUnique({
      where: {
        id: listId,
      },
    });

    if (!list) {
      return {
        error: "List not found",
      };
    }

    const lastCard = await db.card.findFirst({
      where: { listId },
      orderBy: { order: "desc" },
      select: { order: true },
    });

    const newOrder = lastCard ? lastCard.order + 1 : 1;

    card = await db.card.create({
      data: {
        title,
        listId,
        order: newOrder,
      },
    });

    await createAuditLog({
      entityId: card.id,
      entityTitle: card.title,
      entityType: ENTITY_TYPE.CARD,
      action: ACTION.CREATE,
    });
  } catch (error) {
    return {
      error: "Failed to create."
    }
  }

  revalidatePath(`/board/${boardId}`);
  return { data: card };
};

export const createCard = createSafeAction(CreateCard, handler);
```

create-channel-modal.tsx

```
"use client"
import { Dialog, DialogContent, DialogDescription, DialogFooter, DialogHeader, DialogTitle} from
```

```
"@/components/ui/dialog"

import {Form, FormControl, FormField, FormItem, FormLabel, FormMessage} from "@/components/ui/form"

import { Select, SelectContent, SelectItem, SelectTrigger, SelectValue } from "@/components/ui/select"

import { Input } from "../ui/input";
import { Button } from "../ui/button";

import { useForm } from "react-hook-form";
import * as z from "zod";
import {zodResolver} from "@hookform/resolvers/zod";
import { useEffect, useState } from "react";
import { FileUpload } from "../file-upload";

import axios from "axios";
import { useParams, useRouter } from "next/navigation";
import { useModal } from "@/hooks/use-modal-store";
import { ChannelType } from "@prisma/client";

import qs from "query-string";

const formSchema = z.object({
    name: z.string().min(1,{
        message: "Channel name is required."
    }).refine(
        name => name !== "general",{message:"Channel name cannot be 'general'"}
    ),
    type: z.nativeEnum(ChannelType)
})

export const CreateChannelModal = () => {
    const {isOpen, onClose, type, data} = useModal();
    const router = useRouter();
    const params = useParams();

    const isModalOpen = isOpen && type === "createChannel";
    const {channelType} = data;

    const form = useForm({
        resolver: zodResolver(formSchema),
        defaultValues: {
            name:"",
            type: channelType || ChannelType.TEXT
        }
    });

    useEffect(()=>{
        if(channelType){
            form.setValue("type", channelType)
        }else {
            form.setValue("type", ChannelType.TEXT)
        }
    },[channelType, form])
    const isLoading = form.formState.isSubmitting;

    const onSubmit = async (values: z.infer<typeof formSchema>) => {
        try {
            const url = qs.stringifyUrl({
                url: "/api/channels",
                query: {
                    serverId: params?.serverId
```

```
            }
        });

        await axios.post(url, values);
        form.reset();
        router.refresh();
        onClose();
    } catch (error) {
        console.log(error);
    }
}

const handleClose = () => {
    form.reset();
    onClose();
}

return (
    <Dialog open={isModalOpen} onOpenChange={handleClose}>
        <DialogContent className="bg-white text-black p-0 overflow-hidden">
            <DialogHeader className="pt-8 px-6">
                <DialogTitle className="text-2xl text-center font-bold">
                    Create Channel
                </DialogTitle>
            </DialogHeader>
            <Form {...form}>
                <form onSubmit={form.handleSubmit(onSubmit)} className="space-y-8">
                    <div className="space-y-8 px-6">
                        <FormField control={form.control} name="name" render={({field}) => (
                            <FormItem>
                                    <FormLabel className="uppercase text-xs font-bold text-zinc-500 dark:text-
secondary/70">
                                    Channel name
                                </FormLabel>
                                <FormControl>
                                    <Input
                                        disabled={isLoading}
                                            className="bg-zinc-300/50 border-0 focus-visible:ring-0 text-black focus-
visible:ring-offset-0"
                                        placeholder="Enter channel name"
                                        {...field}
                                    />
                                </FormControl>
                                <FormMessage />
                            </FormItem>
                        )} />
                        <FormField control={form.control} name="type" render={({field})=>(
                            <FormItem>
                                <FormLabel>Channel Type</FormLabel>
                                                <Select disabled={isLoading} onValueChange={field.onChange}
defaultValue={field.value}>
                                    <FormControl>
                                        <SelectTrigger className="bg-zinc-300/50 border-0 focus:ring-0 text-black ring-
offset-0 focus:ring-offset-0 capitalize outline-none" >
                                            <SelectValue placeholder="Select a channel type" />
                                        </SelectTrigger>
                                    </FormControl>
                                    <SelectContent>
                                        {Object.values(ChannelType).map((type)=> (
                                            <SelectItem key={type} value={type} className="capitalize">
                                                {type.toLowerCase()}
                                            </SelectItem>
```

```
                                    ))}
                                </SelectContent>
                            </Select>
                            <FormMessage/>
                        </FormItem>
                    )} />
                </div>
                <DialogFooter classNamе="bg-gray-100 px-6 py-4">
                    <Button disabled={isLoading} variant={"primary"}>
                        Create
                    </Button>
                </DialogFooter>
            </form>
        </Form>
    </DialogContent>
</Dialog>
);
}
```

navigation-sidebar.tsx

```
import { currentProfile } from "@/lib/current-profile"
import { redirect } from "next/navigation";
import { db } from "@/lib/db";
import { NavigationAction } from "./navigation-action";
import { Separator } from "../ui/separator";
import { ScrollArea } from "../ui/scroll-area";
import { NavigationItem } from "./navigation-item";
import { ModeToggle } from "../mode-toggle";
import { UserButton } from "@clerk/nextjs";

export const NavigationSidebar = async () => {
    const profile = await currentProfile();

    if(!profile) return redirect("/");

    const servers = db.server.findMany({
        where:{
            members:{
                some:{
                    profileId:profile.id
                }
            }
        }
    });
    return (
        <div classNamе="space-y-4 flex flex-col items-center h-full text-primary w-full dark:bg-[#1E1F22] bg-[#E3E5E5]
py-3">
            <NavigationAction/>
            <Separator classNamе="h-[2px] bg-zinc-300 dark:bg-zinc-700 rounded-md w-10 mx-auto"/>
            <ScrollArea classNamе="flex-1 w-full">
                {(await servers).map((server)=>(
                    <div key={server.id} classNamе="mb-4">
                        <NavigationItem id={server.id} name={server.name} imageUrl={server.imageUrl} />
                    </div>
                ))}

            </ScrollArea>
            <div classNamе="pb-3 mt-auto flex items-center flex-col gap-y-4">
                <ModeToggle/>
                <UserButton afterSignOutUrl="/" appearance={{elements:{avatarBox:"h-[48px] w-[48px]"}}}/>
            </div>
        </div>
```

```
    )
}
```

socket-provider.tsx

```
"use client";
import {createContext, useContext, useEffect, useState} from "react";
import {io as ClientIo} from "socket.io-client";

type SocketContextType = {
    socket: any | null;
    isConnected: boolean;
}

const SocketContext = createContext<SocketContextType>({
    socket: null,
    isConnected: false,
})

export const useSocket = () => {
    return useContext(SocketContext);
}

export const SocketProvider = ({children}:{children:React.ReactNode}) => {
    const [socket, setSocket] = useState(null);
    const [isConnected, setIsConnected] = useState(false);

    useEffect(()=>{
        const socketInstance = new (ClientIo as any)(process.env.NEXT_PUBLIC_SITE_URL!, {
            path:"/api/socket/io",
            addTrailingSlash: false
        });

        socketInstance.on("connect", ()=> {
            setIsConnected(true);
        })

        socketInstance.on("disconnect", ()=> {
            setIsConnected(false);
        });

        setSocket(socketInstance);
```

```
        return () => {
            socketInstance.disconnect();
        }

    },[])

    return (
        <SocketContext.Provider value={{socket, isConnected}}>
            {children}
        </SocketContext.Provider>
    )
}
```

server-channel.tsx

```
"use client";

import { cn } from "@/lib/utils";
import { Channel, ChannelType, MemberRole, Server } from "@prisma/client";
import { Edit, Hash, Lock, Mic, Trash, Video } from "lucide-react";
import { useParams, useRouter } from "next/navigation";
import { ActionTooltip } from "../action-tooltip";
import { ModalType, useModal } from "@/hooks/use-modal-store";

interface ServerChannelProps {
    channel: Channel;
    server: Server;
    role?:MemberRole
}

const iconMap = {
    [ChannelType.TEXT]: Hash,
    [ChannelType.AUDIO]: Mic,
    [ChannelType.VIDEO]: Video
}

export const ServerChannel = ({channel, server, role}: ServerChannelProps) => {
    const params = useParams();
    const router = useRouter();

    const {onOpen} = useModal()
```

```
    const Icon = iconMap[channel.type]

    const onClick = () => {
        router.push(`/servers/${params?.serverId}/channels/${channel.id}`)
    }

    const onAction = (e:React.MouseEvent, action: ModalType) => {
        e.stopPropagation();
        onOpen(action, {channel,server});
    }
    return (
        <button
        onClick={onClick}
                className={cn("group px-2 py-2 rounded-md flex items-center gap-x-2 w-full hover:bg-zinc-700/10 dark:hover:bg-zinc-700/50 transition mb-1",
                        params?.channelId === channel.id && "bg-zinc-700/20 dark:bg-zinc-700"
        )}>
            <Icon className="flex-shrink-0 w-5 h-5 text-zinc-500 dark:text-zinc-400" />
                <p className={cn("line-clamp-1 font-semibold text-sm text-zinc-500 group-hover:text-zinc-600 dark:text-zinc-400 dark:group-hover:text-zinc-300 dark:group-hover:max-w-[7.5rem] overflow-hidden transition",
                                params?.channelId === channel.id && "text-primary dark:text-zinc-200 dark:group-hover:text-white"
            )}>
                {channel.name}
            </p>
            {channel.name !== "general" && role !== MemberRole.GUEST && (
                <div className="ml-auto flex items-center gap-x-2" >
                    <ActionTooltip label="Edit">
                        <Edit onClick={(e)=>onAction(e, "editChannel")} className="hidden group-hover:block w-4 h-4 text-zinc-500 hover:text-zinc-600 dark:text-zinc-400 dark:hover:text-zinc-300 transition" />
                    </ActionTooltip>
                    <ActionTooltip label="Delete">
                        <Trash onClick={(e)=>onAction(e, "deleteChannel")} className="hidden group-hover:block w-4 h-4 text-zinc-500 hover:text-zinc-600 dark:text-zinc-400 dark:hover:text-zinc-300 transition" />
                    </ActionTooltip>
                </div>
            )}
            {channel.name === "general" && (
                <Lock className="ml-auto w-4 h-4 text-zinc-500 dark:text-zinc-400" />
            )}
```

```
        </button>
    )
}
```

server-search.tsx

```
"use client"
import { Search } from "lucide-react";
import { useEffect, useState } from "react";
import { CommandDialog, CommandEmpty, CommandGroup, CommandInput, CommandItem, CommandList } from "../ui/command";
import { useParams, useRouter } from "next/navigation";

interface ServerSearchProps {
    data:{
        label:string;
        type: "channel" | "member";
        data: {
            icon: React.ReactNode;
            name: string;
            id: string;
        }[] | undefined
    }[]
}

export const ServerSearch = ({data} : ServerSearchProps) => {
    const [open, setOpen] = useState(false);
    const router = useRouter();
    const params = useParams();

    useEffect(()=>{
        const searchKeyDown = (e:KeyboardEvent) => {
            if(e.key === "k" && (e.metaKey || e.ctrlKey)){
                e.preventDefault();
                setOpen((open) => !open)
            }
        }

        document.addEventListener("keydown", searchKeyDown);
        return () => document.removeEventListener("keydown", searchKeyDown);
    },[])
```

```
    const onClick = ({id,type}:{id: string, type: "channel" | "member"}) => {
        setOpen(false);

        if(type==='channel') return router.push(`/servers/${params?.serverId}/channels/${id}`)
        if(type==='member') return router.push(`/servers/${params?.serverId}/conversations/${id}`)
    }

    return (
        <>
            <button
                onClick={()=>setOpen(true)}
                        className="group px-2 py-2 rounded-md flex items-center gap-x-2 w-full hover:bg-zinc-700/10 dark:hover:bg-zinc-700/50 transition" >
                <Search className="w-4 h-4 text-zinc-500 dark:text-zinc-400" />
                        <p className="font-semibold text-sm text-zinc-500 dark:text-zinc-400 group-hover:text-zinc-600 dark:group-hover:text-zinc-300 transition" >
                    Search
                </p>
                  <kbd className="pointer-events-none inline-flex h-5 select-none items-center gap-1 rounded border bg-muted px-1.5 font-mono text-[10px] font-medium text-muted-foreground ml-auto" >
                    <span className="text-xs" >CTRL</span>K
                </kbd>
            </button>
            <CommandDialog open={open} onOpenChange={setOpen}>
                <CommandInput placeholder="Search all channels and membmers" />
                <CommandList>
                    <CommandEmpty>
                        No Results found
                    </CommandEmpty>
                    {data.map(({label, type, data}) => {
                        if(!data?.length) return null;

                        return (
                            <CommandGroup key={label} heading={label}>
                                {data?.map(({id, icon, name}) => {
                                    return (
                                        <CommandItem onSelect={()=>onClick({id,type})} key={id}>
                                            {icon}
                                            <span>{name}</span>
                                        </CommandItem>
                                    )
```

```
                                    })}
                                </CommandGroup>
                            )
                        })}
                    </CommandList>
                </CommandDialog>
            </>
        )
}
```

server-sidebar.tsx

```
import { currentProfile } from "@/lib/current-profile";
import { redirect } from "next/navigation";
import { db } from "@/lib/db";
import { ChannelType, MemberRole } from "@prisma/client";
import { ServerHeader } from "./server-header";
import { ScrollArea } from "../ui/scroll-area";
import { ServerSearch } from "./server-search";
import { Hash, Mic, ShieldAlert, ShieldCheck, Video } from "lucide-react";
import { Separator } from "../ui/separator";
import { ServerSection } from "./server-section";
import { ServerChannel } from "./server-channel";
import { ServerMember } from "./server-member";
import { Button } from "../ui/button";
import { ServerWorkspace } from "./server-workspace";

interface ServerSidebarProps {
    serverId:string;
}

const iconMap = {
    [ChannelType.TEXT]: <Hash className="mr-2 h-4 w-4" />,
    [ChannelType.AUDIO]: <Mic className="mr-2 h-4 w-4" />,
    [ChannelType.VIDEO]: <Video className="mr-2 h-4 w-4" />
}

const roleIconMap = {
    [MemberRole.GUEST]: null,
    [MemberRole.MODERATOR]: <ShieldCheck className="w-4 h-4 text-indigo-500 mr-2"/>,
    [MemberRole.ADMIN]: <ShieldAlert className="w-4 h-4 text-rose-500 mr-2"/>
}
```

```
export const ServerSidebar = async ({serverId}: ServerSidebarProps) => {
    const profile = await currentProfile();
    if(!profile) redirect("/");
    const server = await db.server.findUnique({
        where:{
            id: serverId
        },
        include:{
            channels:{
                orderBy:{
                    createdAt:"asc"
                },
            },
            members:{
                include:{
                    profile: true
                },
                orderBy:{
                    role: "asc"
                }
            }
        }
    })

    const textChannels = server?.channels.filter((channel)=>channel.type === ChannelType.TEXT);
    const videoChannels = server?.channels.filter((channel)=>channel.type === ChannelType.VIDEO);
    const audioChannels = server?.channels.filter((channel)=>channel.type === ChannelType.AUDIO);
    const members = server?.members.filter((member) => member.profileId !== profile.id);

    if(!server) redirect("/");

    const role = server.members.find((member) => member.profileId === profile.id)?.role;

    return (
        <div className="flex flex-col h-full text-primary w-full dark:bg-[#2B2D31] bg-[#F2F3F5]">
            <ServerHeader server={server} role={role} />
            <ScrollArea className="flex-1 px-3">
                <div className="mt-2">

                    <ServerSearch data={[
```

```
            {
                label: "Text Channels",
                type: "channel",
                data: textChannels?.map((channel) => ({
                    id: channel.id,
                    name: channel.name,
                    icon: iconMap[channel.type],
                }))
            },
            {
                label: "Voice Channels",
                type: "channel",
                data: audioChannels?.map((channel) => ({
                    id: channel.id,
                    name: channel.name,
                    icon: iconMap[channel.type],
                }))
            },
            {
                label: "Video Channels",
                type: "channel",
                data: videoChannels?.map((channel) => ({
                    id: channel.id,
                    name: channel.name,
                    icon: iconMap[channel.type],
                }))
            },
            {
                label: "Members",
                type: "member",
                data: members?.map((member) => ({
                    id: member.id,
                    name: member.profile.name,
                    icon: roleIconMap[member.role],
                }))
            }
        ]} />
    </div>
    <Separator className="bg-zinc-200 dark:bg-zinc-700 rounded-md my-2" />
    <div className="space-y-[2px]">
        <ServerWorkspace serverId={serverId} />
```

```
            </div>
            {!!textChannels?.length && (
                <div>
                    <ServerSection sectionType="channels" channelType={ChannelType.TEXT} role={role}
label="Text Channels" />
                    <div className="space-y-[2px]">
                        {textChannels.map((channel) => (
                            <ServerChannel key={channel.id} channel={channel} role={role} server={server} />
                        ))}
                    </div>
                </div>
            )}
            {!!audioChannels?.length && (
                <div>
                    <ServerSection sectionType="channels" channelType={ChannelType.AUDIO} role={role}
label="Voice Channels" />
                    <div className="space-y-[2px]">
                        {audioChannels.map((channel) => (
                            <ServerChannel key={channel.id} channel={channel} role={role} server={server} />
                        ))}
                    </div>
                </div>
            )}
            {!!videoChannels?.length && (
                <div>
                    <ServerSection sectionType="channels" channelType={ChannelType.VIDEO} role={role}
label="Video Channels" />
                    <div className="space-y-[2px]">
                        {videoChannels.map((channel) => (
                            <ServerChannel key={channel.id} channel={channel} role={role} server={server} />
                        ))}
                    </div>
                </div>
            )}
            {!!members?.length && (
                <div>
                    <ServerSection sectionType="members" role={role} label="Members" server={server} />
                    <div className="space-y-[2px]">
                        {members.map((member) => (
                            <ServerMember key={member.id} member={member} server={server}/>
                        ))}
```

```
                        </div>
                    </div>
                )}
            </ScrollArea>
        </div>
    )
}
```

board/[boardId]/page.tsx

```
import { db } from "@/lib/db";
import { auth } from "@clerk/nextjs/server";
import { redirect } from "next/navigation";
import { ListContainer } from "./_components/list-container";

interface IBoardIdPageProps {
    params: {
      boardId: string;
    };
}
interface IBoardIdPageProps {
    params: {
      boardId: string;
    };
}
const BoardIdPage = async ({ params }: IBoardIdPageProps) => {

    const { boardId } = params;
    const lists = await db.list.findMany({
        where: {
          boardId: boardId,
        },
        include: {
          cards: {
            orderBy: {
              order: "asc",
            },
          },
        },
        orderBy: {
          order: "asc",
        },
```

```
        });
    return (
        <div className="p-4 h-full overflow-x-auto">
            <ListContainer data={lists} boardId={params.boardId} />
        </div>
    )
}

export default BoardIdPage;
```

list-form.tsx

```
"use client";
import { Plus, X } from "lucide-react";
import { ListWrapper } from "./list-wrapper";
import { useState, useRef, ElementRef } from "react";
import { useEventListener,useOnClickOutside   } from 'usehooks-ts'
import { FormInput } from "@/components/form/form-input";
import { useParams, useRouter } from "next/navigation";
import { FormSubmit } from "@/components/form/form-button";
import { Button } from "@/components/ui/button";
import { useAction } from "@/hooks/use-action";
import { createList } from "@/actions/create-list";
import { toast } from "sonner";
export const ListForm = () => {
  const router = useRouter();
  const params = useParams();
  const [isEditing, setIsEditing] = useState(false);

  const formRef = useRef<ElementRef<"form">>(null);
  const inputRef = useRef<ElementRef<"input">>(null);

  const enableEditing = () => {
      setIsEditing(true);
      setTimeout(() => {
        inputRef.current?.focus();
      });
  };

  const disableEditing = () => {
      setIsEditing(false);
  };
```

```
const { execute, fieldErrors } = useAction(createList, {
  onSuccess: (data) => {
    toast.success(`List "${data.title}" created`);
    disableEditing();
    router.refresh();
  },
  onError: (error) => {
    toast.error(error);
  },
});

const onKeyDown = (e: KeyboardEvent) => {
    if (e.key === "Escape") {
      disableEditing();
    };
};

const onSubmit = (formData: FormData) => {
  const title = formData.get("title") as string;
  const boardId = formData.get("boardId") as string;

  execute({
    title,
    boardId
  });
}

useEventListener("keydown", onKeyDown);
useOnClickOutside(formRef, disableEditing);

if (isEditing) {
  return (
    <ListWrapper>
      <form
        action={onSubmit}
        ref={formRef}
        className="w-full p-3 rounded-md bg-white space-y-4 shadow-md"
      >
        <FormInput
          ref={inputRef}
```

```
            errors={fieldErrors}
            id="title"
                 className="text-sm px-2 py-1 h-7 font-medium border-transparent hover:border-input focus:border-input transition"
            placeholder="Enter list title..."
          />
          <input
            hidden
            value={params?.boardId}
            name="boardId"
          />
          <div className="flex items-center gap-x-1">
            <FormSubmit>
              Add list
            </FormSubmit>
            <Button
              onClick={disableEditing}
              size="sm"
              variant="ghost"
            >
              <X className="h-5 w-5" />
            </Button>
          </div>
        </form>
      </ListWrapper>
    );
  };
  return (
    <ListWrapper>
      <button
        onClick={enableEditing}
         className="w-full rounded-md bg-white/80 hover:bg-white/50 transition p-3 flex items-center font-medium text-sm"
      >
        <Plus className="h-4 w-4 mr-2" />
        Add a list
      </button>
    </ListWrapper>
  );
}
```