БАКАЛАВРСЬКА РОБОТА

БР.КІ-32.00.00.000 ПЗ

Група КІ-21-1

Шістка Святослав

2025

Міністерство освіти і науки України

Івано-Франківський національний технічний університет нафти і газу
Інститут інформаційних технологій

Кафедра комп'ютерних систем і мереж

Шістка Святослав Любомирович

УДК 004.42

БАКАЛАВРСЬКА РОБОТА

Розробка корпоративного месенджера для організації комунікації учнів ONLNIE-школи LOGIKA засобами React та Express

Комп'ютерна інженерія
(назва освітньої програми)

123 – Комп'ютерна інженерія
(шифр і назва спеціальності)

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

Здобувач освітнього ступеня _____ *Шістка С.Л.*
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____ *Мануляк І.З., доцент*
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

д.т.н., професор /С. І. Мельничук/
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025 рік

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*

Кафедра *Комп'ютерних систем і мереж*

Освітньо-кваліфікаційний рівень *бакалавр*

Спеціальність *123 – Комп'ютерна інженерія*

ЗАТВЕРДЖУЮ:

Зав. кафедрою КСМ

С.І. Мельничук

« 05» травня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Шістці Святославу Любомировичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Розробка корпоративного месенджера для організації комунікації учнів ONLINE-школи LOGIKA засобами React та Express*

керівник проекту (роботи) *Мануляк І.З., доцент*

затверджені наказом вищого навчального закладу від *05.05.2025 № 275/7*

2. Строк подання студентом проекту (роботи) *12 червня 2025р.*

3. Вихідні дані до роботи *Методичні вказівки, технічна література* .

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *1 Огляд аналогів та постановка завдання. 2 Розробка структури бази даних, розробка UML діаграми. 3 Розробка вигляду та функціоналу WEB-застосунку.* .

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів роботи

Розділ	**Консультант**	**Підпис, дата**

7. Дата видачі завдання – *29 січня 2025 року*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної області та постановка завдання	Березень, 2025	виконано
2	Розробка структури WEB-застосунку	Квітень, 2025	виконано
3	Розробка програмного забезпечення	Травень, 2025	виконано
4	Оформлення пояснювальної записки	Червень, 2025	виконано

Студент _____ Шістка С. Л.

Керівник роботи _____ Мануляк І. З.

АНОТАЦІЯ

У дипломній роботі представлено розробку корпоративного месенджера для забезпечення ефективної комунікації учнів ONLINE-школи LOGIKA. Основною метою стало створення зручного та безпечного середовища для взаємодії між учасниками навчального процесу.

На фронтенді застосовано бібліотеку React, що забезпечує швидке оновлення інтерфейсу та побудову SPA-додатку. Інтерфейс користувача реалізований з урахуванням зручності навігації, адаптивності та сучасного дизайну.

Бекенд створено з використанням Node.js та Express, що дозволило побудувати RESTful API для обробки запитів користувачів. Для зберігання даних застосовано MongoDB, що забезпечує гнучку схему збереження користувачів, постів, коментарів, лайків та підписок.

В результаті було створено функціонал де учнів можуть реєструватись, створювати текстові пости, ставити лайки та писати коментарі, а також підписуватись на інших користувачів

Розробка враховує базові принципи безпеки та масштабованості, що дозволяє надалі інтегрувати нові функції, як-от чати в реальному часі, push-сповіщення або інтеграцію з навчальними модулями.

Ключові слова: React, Express, корпоративний месенджер, MongoDB, REST API, авторизація, пост, коментар, лайк, підписка, ONLINE-школа.

ANNOTATION

The thesis presents the development of a corporate messenger aimed at ensuring effective communication among students of the LOGIKA ONLINE school. The main goal was to create a convenient and secure environment for interaction among participants in the educational process.

The frontend was implemented using the React library, which enables fast interface updates and the creation of a SPA (Single Page Application). The user interface was designed with a focus on ease of navigation, adaptability, and modern design principles.

The backend was built using Node.js and Express, allowing for the creation of a RESTful API to handle user requests. MongoDB was used for data storage, providing a flexible schema for storing users, posts, comments, likes, and subscriptions.

As a result, functionality was implemented allowing students to register, create text posts, like and comment on posts, and subscribe to other users.

The development takes into account basic principles of security and scalability, making it possible to integrate future features such as real-time chat, push notifications, or integration with educational modules.

Keywords: React, Express, corporate messenger, MongoDB, REST API, authentication, post, comment, like, subscription, ONLINE school.

ЗМІСТ

ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

React – сучасна бібліотека JavaScript, розроблена компанією Meta (Facebook), що використовується для створення динамічних та адаптивних інтерфейсів користувача у вебзастосунках.

Express – мінімалістичний та гнучкий фреймворк для Node.js, який використовується для розробки серверної частини вебзастосунків та API.

TypeScript – мова програмування, що є надмножиною JavaScript і додає статичну типізацію, що дозволяє створювати масштабовані та надійні вебзастосунки.

API – Application Programming Interface (програмний інтерфейс додатку).

БД – База Даних.

UML – Unified Modeling Language (уніфікована мова моделювання).

ВСТУП

Актуальність. У сучасному світі, де цифрові технології стрімко проникають у всі сфери життя, особливої актуальності набуває створення ефективних інструментів для онлайн-комунікації. Це особливо важливо в умовах розвитку дистанційної освіти, яка активно використовується в багатьох країнах світу, зокрема й в Україні. Актуальність теми дипломної роботи обумовлена потребою у створенні надійного, зручного та безпечного каналу комунікації між учнями онлайн-шкіл. Такий канал дозволяє не лише обмінюватися повідомленнями, але й формувати повноцінне освітнє середовище з можливістю соціальної взаємодії. Серед вітчизняних та зарубіжних дослідників особливу увагу цьому напрямку приділяють фахівці в галузі EdTech, зокрема в рамках розробки LMS (Learning Management Systems), однак більшість існуючих рішень є або занадто складними, або не орієнтовані на школярів.

Основною проблемою, що вирішується в даній роботі, є створення простого у використанні, функціонального корпоративного месенджера для учнів онлайн-школи LOGIKA. Задача ускладнюється потребою реалізувати як базові функції (реєстрація, автентифікація), так і соціальні механіки (пости, коментарі, лайки, підписки), забезпечуючи при цьому надійне збереження даних і високу швидкість взаємодії між клієнтом і сервером.

Об’єктом дослідження виступає процес організації цифрової комунікації в онлайн-освітньому середовищі.

Предметом дослідження є технічні рішення та методи розробки корпоративного месенджера на основі сучасних веб-технологій — зокрема, JavaScript-фреймворків React для фронтенду та Express для бекенду.

Метою дипломної роботи є створення корпоративного месенджера для учнів онлайн-школи LOGIKA, який забезпечуватиме ефективну внутрішню

комунікацію, можливість створення дописів, коментарів, лайків і підписок, при цьому буде інтуїтивно зрозумілим та адаптованим під навчальний процес.

Методи дослідження, які були використані у роботі, включають аналіз предметної області, проектування бази даних, використання принципів REST-архітектури для побудови API, застосування методів модульного програмування, а також емпіричне тестування розробленого додатку в навчальному середовищі.

Практичне значення отриманих результатів полягає в тому, що створений додаток може бути безпосередньо інтегрований у навчальний процес онлайн-школи LOGIKA, слугуючи ефективним інструментом для обміну повідомленнями та створення навчальної спільноти серед учнів. Крім того, архітектура та код системи можуть бути використані як шаблон для подібних проєктів в інших навчальних закладах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Опис предметної області

Корпоративний месенджер в освітньому середовищі — це спеціалізована платформа для обміну повідомленнями, файлами та іншими формами інформації між учасниками навчального процесу. Такий інструмент дозволяє учням, викладачам і адміністраторам швидко й ефективно комунікувати в межах однієї цифрової екосистеми. На відміну від загальнодоступних соціальних мереж, корпоративний месенджер орієнтований на підтримку навчального процесу, з урахуванням вікових особливостей та цілей користувачів.

Використання корпоративного месенджера дає змогу централізувати комунікацію, забезпечити безпечне середовище для спілкування учнів, зменшити вплив сторонніх чинників та сприяти формуванню здорової навчальної спільноти. Крім того, він може бути адаптований до конкретних освітніх потреб — наприклад, організація груп за класами, предметами або проєктами. Це значно підвищує ефективність навчання та сприяє залученню учнів у навчальний процес.

Попри велику кількість доступних інструментів для онлайн-комунікації, таких як Slack, Discord чи Telegram, більшість із них не розраховані на використання у шкільному або освітньому середовищі. Серед основних проблем — перевантаженість функціоналом, наявність небажаного контенту, відсутність української локалізації, складність налаштувань та низький рівень безпеки для дітей. Такі сервіси часто відволікають від навчального процесу замість того, щоб його підтримувати.

Найсуттєвішою проблемою є закритість систем і складність інтегрування власних освітніх рішень. Більшість популярних месенджерів не дозволяють вільно змінювати або розширювати функціональність відповідно до потреб

конкретної школи чи програми. Вони не надають гнучкого API або мають обмежені можливості для кастомізації, що унеможливлює повноцінну інтеграцію з внутрішніми платформами або навчальними модулями.

Таким чином, розробка власного корпоративного месенджера для учнів ONLINE-школи LOGIKA є актуальним та доцільним рішенням. Такий месенджер дозволить створити комфортне та безпечне середовище для комунікації, яке буде гнучко налаштовуватись під потреби навчального процесу, матиме українську локалізацію та орієнтуватиметься саме на освітню функцію, а не на розваги чи сторонній контент.

1.2 Організаційна структура та характеристика освітнього закладу

Освітня платформа LOGIKA — це найбільша онлайн-школа програмування для дітей в Україні, яка охоплює близько 10 000 учнів та має команду з понад 200 кваліфікованих викладачів. LOGIKA спеціалізується на викладанні сучасних IT-дисциплін, розвиваючи в учнів цифрові навички, логічне мислення, здатність працювати над проєктами та ефективно взаємодіяти в команді. Завдяки використанню новітніх технологій та гнучких форматів навчання, школа успішно поєднує очну та дистанційну освіту, забезпечуючи високу якість викладання незалежно від місця перебування учня.

Організаційна структура школи LOGIKA включає такі ключові елементи:

– адміністрація — здійснює загальне керівництво навчальним процесом, координує діяльність усіх підрозділів, забезпечує методичну підтримку та взаємодію з батьками;

– методичний відділ — відповідає за розробку навчальних програм, створення гідів для викладачів, стандарти оцінювання та методичне оновлення курсів;

– викладачі/ментори — реалізують навчальні програми, проводять заняття, ведуть індивідуальні консультації, контролюють проєкти та підтримують учнів у навчанні;

– технічна підтримка — забезпечує стабільну роботу онлайн-платформи, безперебійну комунікацію між учасниками освітнього процесу, захист особистих даних та швидке реагування на технічні запити;

– учні — основна аудиторія платформи, яка потребує доступних, зручних і безпечних цифрових інструментів для навчання та взаємодії.

У рамках цифрової трансформації освітнього процесу LOGIKA приділяє особливу увагу розвитку внутрішньої комунікації, що є критично важливою для підтримки навчального ритму, командної роботи та формування сприятливого навчального середовища. Саме з цією метою було ініційовано розробку власного корпоративного месенджера — програмного рішення, яке дозволяє ефективно обмінюватися повідомленнями, координувати командні проєкти, ставити лайки, залишати коментарі до дописів і підписуватися на інших учнів у межах єдиного навчального цифрового середовища.

1.3 Огляд аналогів застосунку

У процесі розробки нового програмного забезпечення важливим кроком є дослідження вже існуючих рішень на ринку. Це дозволяє краще зрозуміти поточні стандарти, виявити ефективні функціональні підходи, а також проаналізувати слабкі сторони, які можуть бути покращені у власному продукті. Особливо актуальним таке дослідження є у сфері комунікаційних застосунків, де конкуренція висока, а функціональні очікування користувачів зростають.

Для аналізу було обрано застосунок Slack, який є одним із найвідоміших і найпопулярніших корпоративних месенджерів у світі. Він широко використовується в бізнесі, IT-компаніях та проєктних командах для організації внутрішньої комунікації. Slack є прикладом потужного рішення з великим спектром можливостей, тому його аналіз дає змогу краще зрозуміти принципи побудови таких систем.

На головному екрані (рис. 1.1) Slack користувач бачить список робочих

просторів (серверів), до яких він підключений. Кожен робочий простір представляє окрему організацію або команду, і має свою структуру каналів, учасників та повідомлень. Зліва знаходиться панель навігації з доступними серверами, каналами, повідомленнями та приватними чатами.

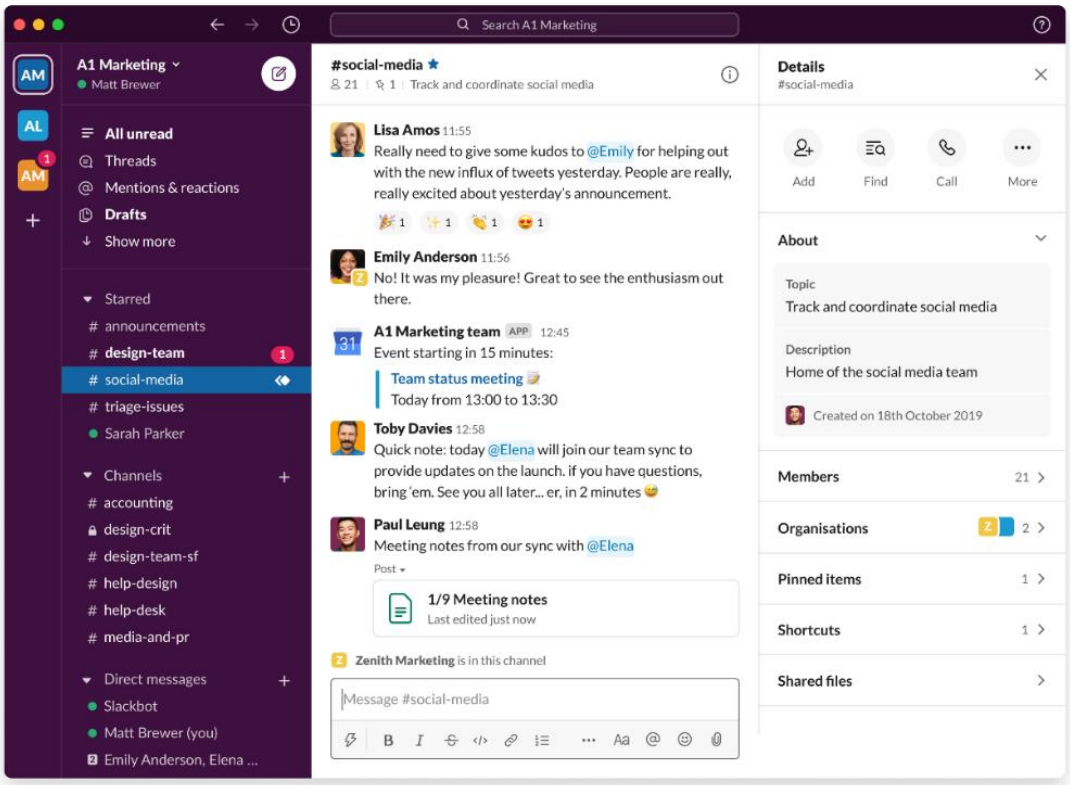

Рисунок 1.1 – Головний екран застосунку Slack

При відкритті одного з робочих просторів користувач потрапляє до головного інтерфейсу (рис. 1.2), де відображено список тематичних каналів (чатів) — як загальних, так і приватних. Також у правій частині можна переглянути список учасників, надіслати особисте повідомлення або створити нову групу. Канали дозволяють обговорювати різні теми, завантажувати файли, додавати реакції, створювати голосування тощо.

Slack має низку переваг, серед яких: готова інфраструктура, багатий функціонал, підтримка інтеграцій з іншими сервісами (Google Drive, Trello, Zoom), підтримка ботів, історія повідомлень, зручна система пошуку та висока стабільність роботи. Це — потужний і перевірений інструмент для професійної комунікації.

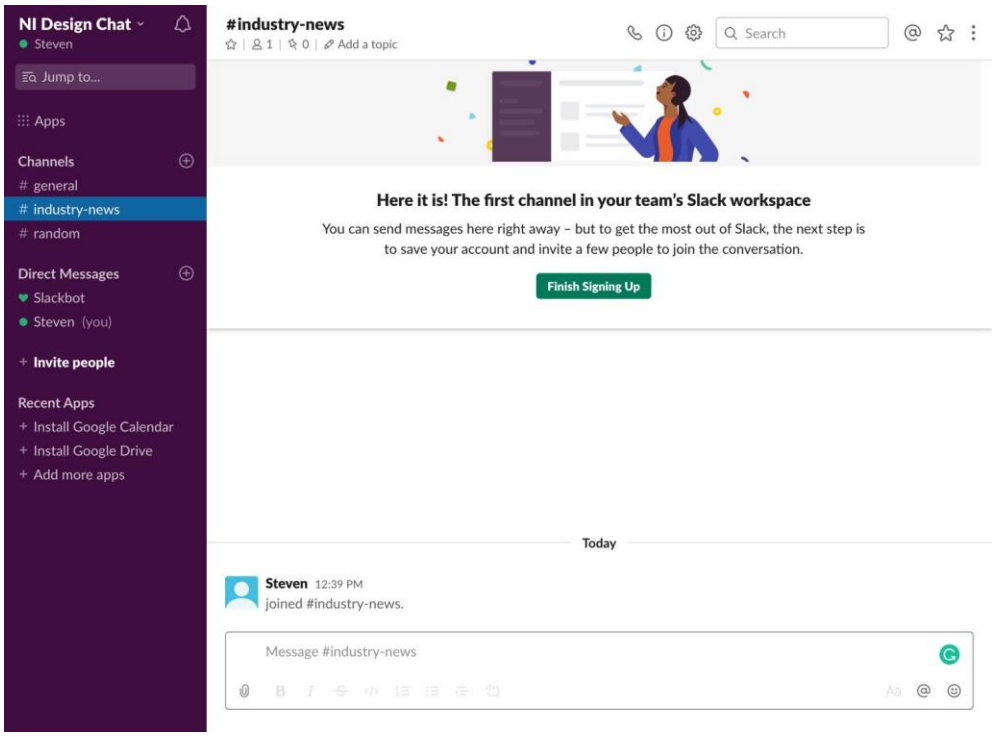

Рисунок 1.2 – Список тематичних каналів одного простору

Однак, попри свої переваги, Slack має і вагомі недоліки, особливо з точки зору використання в освітньому середовищі. Його інтерфейс є перевантаженим і неінтуїтивним для дітей та підлітків. Додаток не має широких можливостей для кастомізації, особливо у візуальній частині. Крім того, відсутність україномовної локалізації та орієнтація на дорослу корпоративну аудиторію створює бар'єри для його ефективного використання в навчальному процесі.

У зв'язку з цим, розробка власного корпоративного месенджера для учнів ONLINE-школи LOGIKA є виправданим і актуальним рішенням. Він має враховувати переваги подібних систем, але бути простішим, інтуїтивно зрозумілим, адаптованим до освітніх потреб і кастомізованим під конкретну цільову аудиторію — учнів різного віку.

1.3 Постановка завдання

Основною метою цього проєкту є розробка корпоративного месенджера для організації внутрішньої комунікації учнів ONLINE-школи LOGIKA. Такий

застосунок має забезпечити простий, зручний та безпечний спосіб взаємодії між учнями в межах навчального процесу, враховуючи їхні вікові особливості, потреби та рівень цифрової грамотності. Проєкт спрямований на створення окремої екосистеми для обміну повідомленнями, ідеями та враженнями між учасниками навчального процесу.

Першим важливим етапом є розробка інтерфейсу користувача. На цьому етапі реалізується фронтенд частина застосунку за допомогою бібліотеки React. Інтерфейс має бути мінімалістичним, інтуїтивно зрозумілим та адаптивним. Передбачено реалізацію таких основних розділів, як стрічка постів, особистий профіль, розділ з підписками та сторінка авторизації/реєстрації.

Наступним етапом є розробка логіки взаємодії користувачів. За допомогою Node.js та Express створюється бекенд-застосунок із REST API. Користувачі зможуть реєструватися та входити в систему, створювати пости, ставити лайки, писати коментарі, а також підписуватись на інших учнів. Для зберігання даних використовується MongoDB з Mongoose-моделями для структурування інформації про користувачів, пости, коментарі тощо.

Особливу увагу приділено тестуванню та оптимізації роботи застосунку. Перевіряється правильність обробки запитів, стабільність серверної частини, відповідність функціоналу вимогам та загальна продуктивність.

Отже, в цьому розділі було сформульовано основну мету проєкту, визначено основні етапи реалізації, які включають розробку інтерфейсу, логіки взаємодії, а також тестування та оптимізацію. Проведений раніше аналіз предметної області та існуючих рішень дозволив чітко окреслити, що розробка такого месенджера є актуальною та виправданою, а створення зручного і зрозумілого інструменту комунікації для учнів є ціллю, що має практичну цінність.

2 РОЗРОБКА СТРУКТУРИ WEB-ДОДАТКУ

2.1 Загальна структура WEB додатку

Розроблений додаток є повноцінним веб-застосунком, побудованим за принципом клієнт-серверної архітектури. Уся система поділяється на дві основні частини: клієнтську (frontend) та серверну (backend), які взаємодіють між собою через HTTP-запити за допомогою REST API. Для зберігання даних використовується база даних MongoDB, що забезпечує гнучке й масштабоване управління інформацією про користувачів, пости, лайки, коментарі та підписки.

Інтерфейс користувача реалізовано за допомогою React — сучасної JavaScript-бібліотеки, яка дозволяє створювати динамічні та реактивні компоненти. Дизайн інтерфейсу є мінімалістичним та зрозумілим, орієнтованим на учнів. Застосунок підтримує адаптивну верстку, завдяки чому ним зручно користуватись як на комп’ютерах, так і на мобільних пристроях.

Серверна логіка реалізована з використанням Node.js та Express. Саме на цій частині обробляються запити від клієнта, виконується авторизація, перевірка прав доступу, зберігаються й повертаються дані з бази. Також тут прописані маршрути для взаємодії з постами, користувачами, підписками, лайками та коментарями.

Основні процеси взаємодії з додатком проходять у такій послідовності:

1. Зареєструватись або увійти. Користувач вводить логін та пароль. Якщо дані правильні — відкривається особистий кабінет. У разі реєстрації нові дані записуються в базу даних.

2. Створити пост. На головній сторінці користувач може написати текстовий пост, який зберігається у базі та з’являється в стрічці.

3. Переглянути інші пости. Користувач бачить стрічку з постами інших учнів, які доступні публічно або відповідно до підписок.

4. Поставити лайки або написати коментарі. Під кожним постом доступні

кнопки для лайку або написання коментаря, які також зберігаються у базі.

5. Підписатись на когось. При переході на профіль іншого користувача можна натиснути кнопку «Підписатись», після чого його пости будуть частіше з’являтись у стрічці.

У підсумку, розроблений додаток об’єднує сучасні технології фронтенду та бекенду для створення зручного середовища комунікації між учнями. Його архітектура дозволяє легко масштабувати функціонал у майбутньому, доповнюючи систему новими можливостями відповідно до потреб школи.

2.2 UML-діаграма варіантів використання

У межах розробки застосунку було створено UML-діаграму варіантів використання (Use Case Diagram), яка дозволяє наочно продемонструвати основні функціональні можливості системи з точки зору взаємодії користувача з додатком.

Основним актором системи є користувач (User), який взаємодіє з інтерфейсом застосунку та здійснює основні дії, доступні в межах його повноважень. Кожна з дій користувача представлена окремим варіантом використання, що допомагає структурувати логіку роботи системи.

Першою ключовою функцією є реєстрація та вхід у систему. Користувач має можливість створити обліковий запис, вказавши логін та пароль, або увійти в систему за допомогою вже наявного акаунта. Після цього відкривається доступ до основного функціоналу месенджера.

Далі користувач може створювати пости – короткі текстові повідомлення, які публікуються у загальній стрічці та зберігаються в базі даних. Інші користувачі мають змогу взаємодіяти з цими постами.

Однією з форм взаємодії є поставлення лайків. Користувач може поставити або зняти лайк під будь-яким постом, що дозволяє швидко оцінювати публікації інших учнів.

Також реалізована можливість залишати коментарі під постами. Кожен

користувач може додавати власні думки або відповіді у вигляді коментарів, що сприяє живому спілкуванню в системі.

Окрім цього, користувач може підписуватись на інших учнів, щоб частіше бачити їх пости у стрічці. Це створює ефект персоналізованої стрічки новин та дозволяє формувати спільноти за інтересами.

Таким чином, UML-діаграма варіантів використання чітко демонструє всі основні сценарії взаємодії користувача з системою та дає змогу логічно структурувати вимоги до функціоналу застосунку. Нижче зображено UML-діаграму варіантів використання:

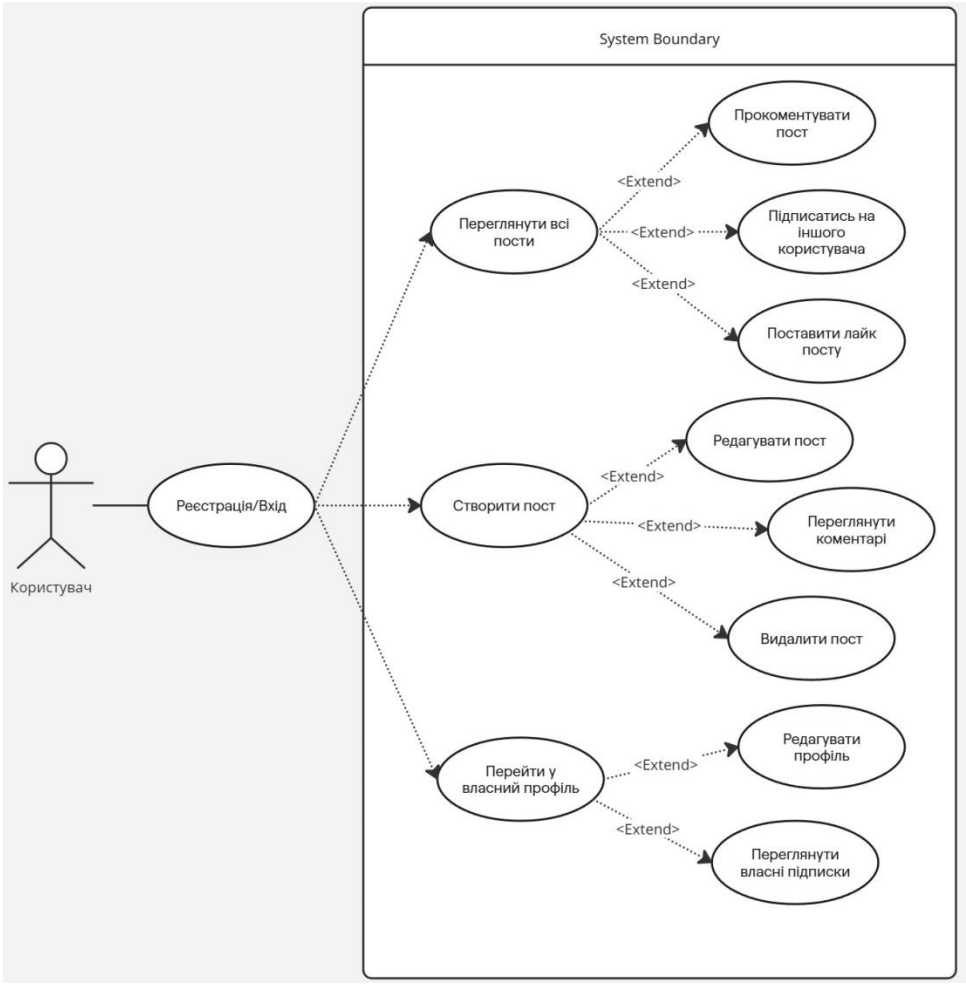

Рисунок 2.1 – UML-діаграма варіантів використання web додатку

Дана UML-діаграма варіантів використання дає змогу наочно показати, які дії може виконувати користувач у системі, та як саме він взаємодіє з

основними функціями додатку. Вона є корисною для розуміння загальної структури взаємодії без заглиблення у технічні деталі реалізації. Така діаграма допомагає чітко сформулювати вимоги до системи, спростити комунікацію між розробником і замовником, а також служить основою для подальшого проєктування логіки застосунку.

2.3 Розробка таблиць бази даних

Структура бази даних була розроблена для ефективного зберігання, організації та обробки всієї інформації, що використовується в додатку. Вона забезпечує логічну взаємозв'язану побудову даних про користувачів, пости, коментарі, лайки та підписки, що дозволяє швидко виконувати запити та забезпечувати стабільну роботу застосунку. Завдяки чітко продуманій структурі бази даних можлива масштабована та безпечна взаємодія між учасниками системи, а також подальше розширення функціоналу без порушення цілісності даних.

Модель User відповідає за зберігання інформації про користувачів системи. Вона містить унікальний ідентифікатор id, який генерується автоматично (@default(auto())) і є головним ключем. Основні поля — це email та password, які необхідні для авторизації. Також є додаткові атрибути: name, avatarUrl, bio, location, дата народження та час створення акаунта (createdAt). Через відношення модель пов'язана з іншими сутностями: кожен користувач може мати багато постів (posts), лайків (likes), коментарів (comments), а також брати участь у системі підписок як той, хто підписується (following), і як той, на кого підписані (followers).

Таблиця 2.1 – Структура таблиці User

Ключове поле	Назва стовпця	Тип даних	Дозволяє NULL	Вміст
☑	id	STRING		Ідентифікатор користувача
	email	STRING		Унікальна електронна адреса

Кінець таблиці 2.1

	password	STRING		Пароль користувача
	name	STRING	☑	Ім’я користувача
	avatarUrl	STRING	☑	Посилання на аватар
	dateOfBirth	DateTime	☑	Дата народження
	createdAt	DateTime		Дата створення акаунта
	updatedAt	DateTime		Дата останнього оновлення
	bio	STRING	☑	Біографія користувача
	location	STRING	☑	Місцезнаходження

Створено за допомогою такого запиту:

```
model User {
  id                String      @id  @default(auto())  @map("_id")
@db.ObjectId
  email       String    @unique
  password    String
  name        String?
  avatarUrl   String?
  dateOfBirth DateTime?
  createdAt   DateTime  @default(now())
  updatedAt   DateTime  @updatedAt
  bio         String?
  location    String?
  posts       Post[]
  likes       Like[]
  comments    Comment[]
  followers   Follows[] @relation("following")
  following   Follows[] @relation("follower")
}
```

Модель Follows реалізує функціонал підписок між користувачами. У неї є унікальний ідентифікатор id, а також два зовнішні ключі: followerId — це той, хто підписується, і followingId — це той, на кого підписані. Обидва ці поля пов’язані з моделлю User, що дозволяє чітко визначити відносини між учасниками системи. Завдяки цій моделі реалізується соціальний функціонал — формування стрічки постів від підписаних користувачів.

Таблиця 2.2 – Структура таблиці Follows

Ключове поле	Назва стовпця	Тип даних	Вміст
☑	id	STRING	Ідентифікатор підписки
	followerId	STRING	Ідентифікатор користувача, який підписався
	followingId	STRING	Ідентифікатор користувача, на якого підписані

Створено за допомогою такого запиту:

```
model Follows {
  id          String @id @default(auto()) @map("_id") @db.ObjectId
  follower     User    @relation("follower", fields: [followerId],
references: [id])
  followerId  String @db.ObjectId
  following   User   @relation("following", fields: [followingId],
references: [id])
  followingId String @db.ObjectId
```

Модель Post зберігає всі публікації, які створюють користувачі. Кожен пост має унікальний id, поле content для тексту публікації, та дату створення (createdAt). Автор поста вказується за допомогою зовнішнього ключа authorId, що пов’язаний із моделлю User. Крім того, пост може мати багато лайків (likes) та коментарів (comments), які також реалізовані як окремі сутності з відповідними зв’язками.

Таблиця 2.3 – Структура таблиці Post

Ключове поле	Назва стовпця	Тип даних	Вміст
☑	Id	STRING	Ідентифікатор поста
	content	STRING	Текст поста
	authorId	STRING	Ідентифікатор користувача (автор поста)
	createdAt	DATETIME	Дата створення поста

Створено за допомогою такого запиту:

```
model Post {
  id        String    @id @default(auto()) @map("_id")
```

```
    @db.ObjectId
        content   String
        author    User      @relation(fields: [authorId],
    references: [id])
        authorId  String    @db.ObjectId
        likes     Like[]
        comments  Comment[]
        createdAt DateTime  @default(now())
    }
```

Модель Like відповідає за зберігання інформації про вподобання постів користувачами. Кожен лайк має унікальний id і пов'язаний одночасно з користувачем (userId, зв'язок із User) і постом (postId, зв'язок із Post). Таким чином, ця таблиця дозволяє відстежувати, хто які пости вподобав, і запобігає дублюванню лайків.

Таблиця 2.4 – Структура таблиці Like

Ключове поле	Назва стовпця	Тип даних	Вміст
☑	Id	STRING	Ідентифікатор лайка
	userId	STRING	Ідентифікатор користувача, що поставив лайк
	postId	STRING	Ідентифікатор поста, якому поставлено лайк

Створено за допомогою такого запиту:

```
    model Like {
  id     String @id @default(auto()) @map("_id") @db.ObjectId
  user   User   @relation(fields: [userId], references: [id])
  userId String @db.ObjectId
  post   Post   @relation(fields: [postId], references: [id])
  postId String @db.ObjectId
```

Модель Comment реалізує можливість залишати коментарі під постами. Вона містить унікальний id, текст коментаря (content), а також зв'язки з автором коментаря (userId, зв'язок із User) і самим постом (postId, зв'язок із

Post). Це дозволяє відтворити повну структуру коментарів до кожної публікації та зберігати історію спілкування користувачів у межах системи.

Таблиця 2.5 – Структура таблиці Comment

Ключове поле	Назва стовпця	Тип даних	Вміст
☑	Id	STRING	Ідентифікатор коментаря
	content	STRING	Текст коментаря
	userId	STRING	Ідентифікатор користувача (автор коментаря)
	postId	STRING	Ідентифікатор поста, до якого належить коментар

У результаті розробки структури бази даних було створено низку взаємопов'язаних таблиць(рис 1.2), які відображають основні сутності корпоративного месенджера: користувачів, пости, лайки, коментарі та підписки.

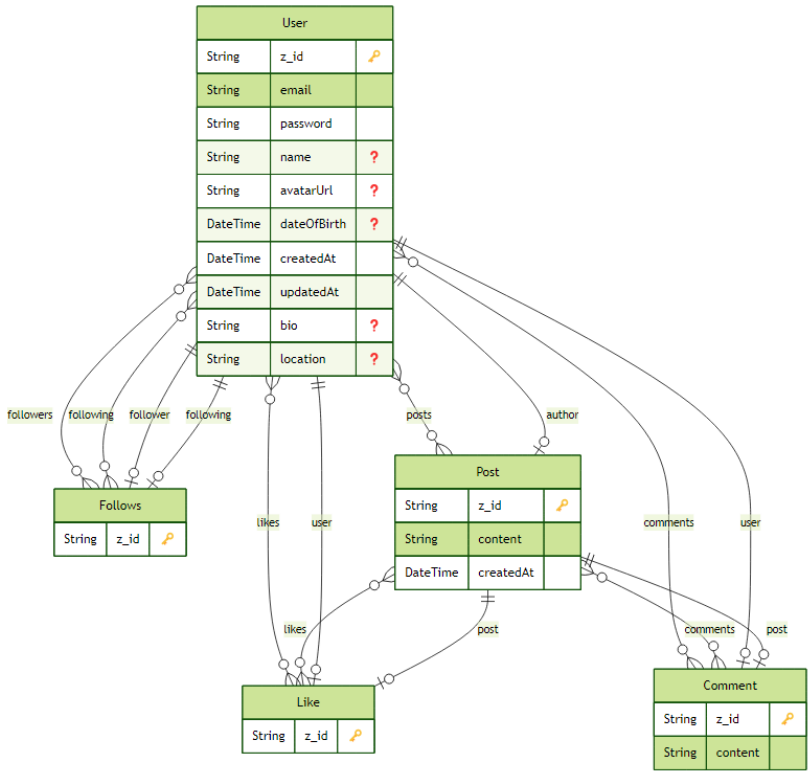

Рисунок 2.2 – ERD-діаграма бази даних WEB додатку корпоративного месенджера

У рамках реалізації корпоративного месенджера було спроєктовано та створено структуру бази даних, яка відображена на ERD-діаграмі(Рисунок 2.2). Ця структура включає основні сутності, що забезпечують функціональність платформи: користувачі (User), пости (Post), коментарі (Comment), лайки (Like) та система підписок (Follows). Кожна з таблиць має чітко визначені поля та зв’язки між собою, що дозволяє ефективно організувати взаємодію користувачів у системі. Наприклад, кожен пост має автора, може мати багато коментарів і лайків, а користувачі можуть підписуватися один на одного, утворюючи соціальні зв’язки.

База даних була реалізована на основі нереляційної СУБД MongoDB, що дозволяє зберігати дані в гнучкому форматі документів. Для опису моделей і взаємозв’язків між ними використовувався інструмент Prisma ORM, який значно спрощує взаємодію з базою даних та автоматизує процес створення схем. Prisma дозволяє зручно керувати міграціями, валідувати структуру даних та забезпечує зручний інтерфейс для взаємодії з MongoDB через JavaScript/TypeScript. Такий підхід дав змогу поєднати потужність гнучкої NoSQL-бази з перевагами типізованого та структурованого доступу до даних.

2.5 Структура основних екранів застосунку

Початковий екран додатку (рис. 2.3) містить в собі блок для реєстрації користувача. У випадку наявності акаунта можна увійти в профіль.

Інтерфейс форми реєстрації користувача у корпоративному месенджері. Блок містить два перемикачі — «Вхід» та «Реєстрація», що дозволяють користувачеві обрати потрібну дію. У режимі реєстрації відображаються поля для введення імені, електронної пошти та паролю. Нижче розміщено посилання для переходу до форми входу, якщо користувач уже має акаунт. Завершує форму велика синя кнопка «Зареєструватись», яка надсилає дані для створення нового облікового запису. Інтерфейс виконано в мінімалістичному та зрозумілому стилі, що підходить для школярів.

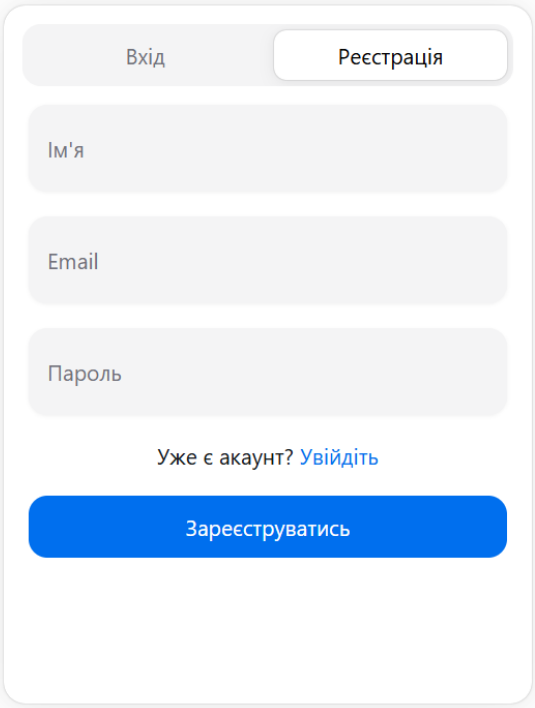

Рисунок 2.3 – Структура головного екрану

Інтерфейс форми реєстрації користувача у корпоративному месенджері. Блок містить два перемикачі — «Вхід» та «Реєстрація», що дозволяють користувачеві обрати потрібну дію. У режимі реєстрації відображаються поля для введення імені, електронної пошти та паролю. Нижче розміщено посилання для переходу до форми входу, якщо користувач уже має акаунт. Завершує форму велика синя кнопка «Зареєструватись», яка надсилає дані для створення нового облікового запису. Інтерфейс виконано в мінімалістичному та зрозумілому стилі, що підходить для школярів.

На рисунку 2.4 зображена головна сторінка корпоративного месенджера для ONLINE-школи LOGIKA, яка представляє собою соціальну мережу для учнів. Інтерфейс виконаний у сучасному мінімалістичному стилі з українською локалізацією. Ліва частина екрану містить навігаційне меню з основними розділами, центральна частина відображає стрічку постів користувачів, а права частина показує профіль поточного користувача. Загальна композиція забезпечує зручну навігацію та інтуїтивно зрозумілий користувацький досвід.

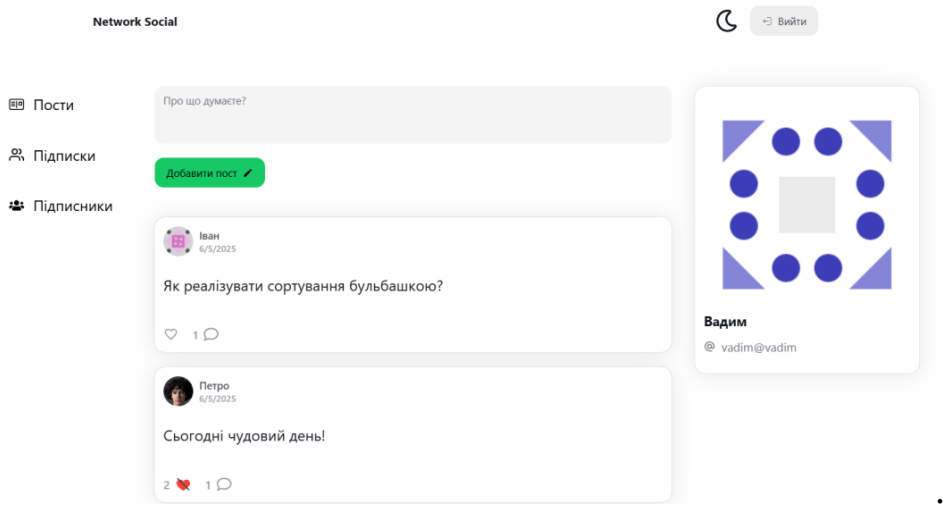

Рисунок 2.4 – Структура екрану статистики витрат

У правому верхньому куті сторінки розташовані дві важливі кнопки управління. Кнопка з іконкою місяця дозволяє користувачам перемикатися між світлою та темною темами оформлення інтерфейсу, що покращує комфорт використання додатку в різних умовах освітлення. Поруч знаходиться кнопка "Вийти", яка забезпечує безпечне завершення сеансу роботи користувача з можливістю повернення до сторінки авторизації.

Центральне місце у функціоналі займає зелена кнопка "Добавити пост" з іконкою олівця, розташована під полем запиту "Про що думаєте?". Ця кнопка дозволяє користувачам створювати нові публікації, ділитися думками, новинами або іншим контентом зі своїми однокласниками та підписниками. Інтуїтивний дизайн кнопки привертає увагу та заохочує до активної участі в житті шкільної спільноти.

Основний контент сторінки представлений стрічкою постів від різних користувачів. Кожен пост містить аватар автора, ім'я, дату публікації та текст повідомлення. Під кожним постом розташовані інтерактивні елементи: кнопка лайка у вигляді сердечка та лічильник вподобань, а також кнопка коментарів з відповідним лічильником.

У правій частині екрану розташований профільний блок поточного користувача з ім'ям "Вадим" та унікальним ідентифікатором. Клік по імені

користувача або його аватару дозволяє перейти до детального перегляду власного профілю, де можна редагувати особисту інформацію, переглядати історію своїх постів та керувати налаштуваннями акаунту.

Навігаційне меню в лівій частині включає розділи "Підписки" та "Підписники", які дозволяють користувачам керувати своїм соціальним колом. Розділ "Підписки" показує список користувачів, на яких підписаний поточний користувач, дозволяючи відстежувати їхню активність. Розділ "Підписники" відображає користувачів, які підписані на поточного користувача, що допомагає розуміти свою аудиторію та популярність у шкільній спільноті.

Після нажаття на конкретний пост ми попадемо на його сторінку (рис 2.5)

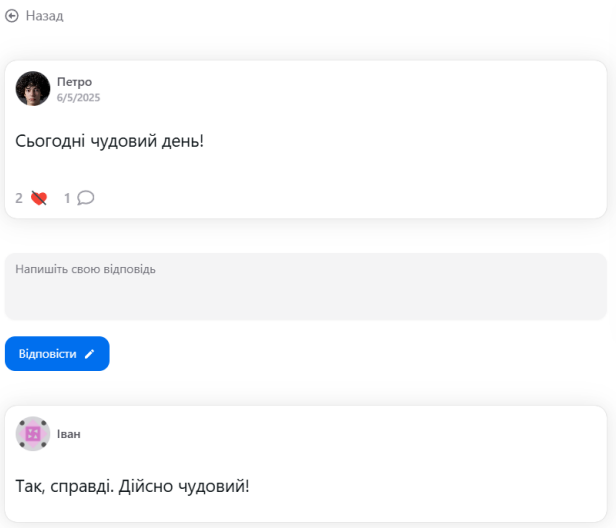

Рисунок 2.5 – Сторінка поста

Ця сторінка представляє детальний перегляд окремого поста в корпоративному месенджері з розширеними можливостями взаємодії. У верхній частині розташована кнопка "Назад" для повернення до головної стрічки, а основний контент займає пост користувача з інтерактивними елементами лайків та коментарів. Сторінка фокусується на одній публікації, дозволяючи користувачам детально ознайомитися з контентом та активністю навколо неї.

Ключовою особливістю сторінки є повноцінний функціонал коментування, який включає можливість перегляду всіх існуючих коментарів

під постом та додавання власних відгуків. Під основною публікацією розташоване текстове поле для введення коментаря з кнопкою "Відповісти", що дозволяє учням активно брати участь в обговореннях та обмінюватися думками. Така організація інтерфейсу сприяє створенню живого комунікативного середовища та поглиблює взаємодію між користувачами платформи.

Натиснувши на власне ім'я користувача на головній сторінці, ми потрапляємо на персональну сторінку акаунту (Рисунок 2.6), де відображається вся інформація про поточного користувача. Сторінка профілю містить великий аватар користувача з геометричним дизайном у синіх тонах, ім'я "Вадим" та електронну адресу "vadim@vadim". Центральне місце займають лічильники соціальної активності, які показують кількість підписників та підписок користувача, що на даний момент становить 0 для обох категорій.

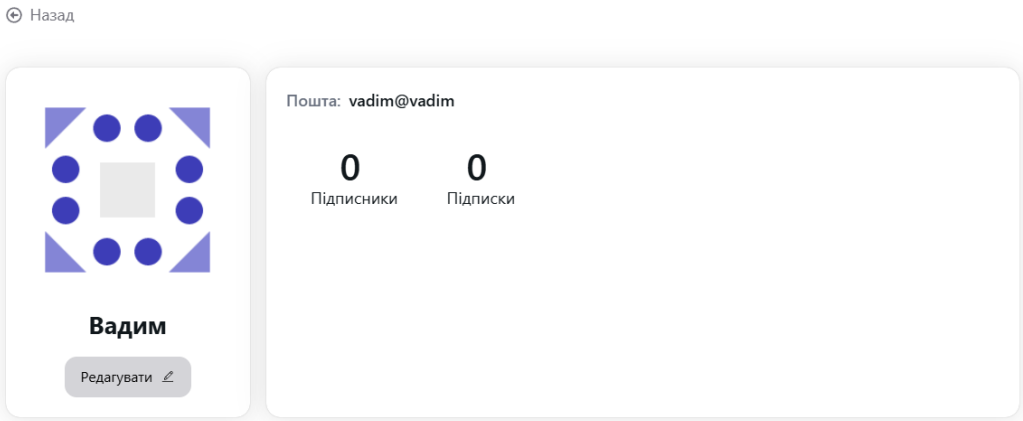

Рисунок 2.6 – Сторінка профілю

Функціонал управління соціальними зв'язками реалізований через інтуїтивно зрозумілі блоки "Підписники" та "Підписки". Клік по цих розділах дозволяє переглянути списки користувачів, які підписані на поточного користувача, або тих, на кого підписаний сам користувач. Це дає можливість керувати своїм соціальним колом, відстежувати популярність власного контенту та знаходити цікавих співрозмовників серед учнів школи.

Ключовою особливістю сторінки профілю є кнопка "Редагувати" з іконкою олівця, розташована під основною інформацією користувача. Натиснувши на неї, відкривається модальне вікно "Редагування профілю" (рис 2.7), яке дозволяє змінювати особисті дані. Це забезпечує користувачам можливість підтримувати актуальність своєї інформації та персоналізувати свій акаунт відповідно до власних потреб.

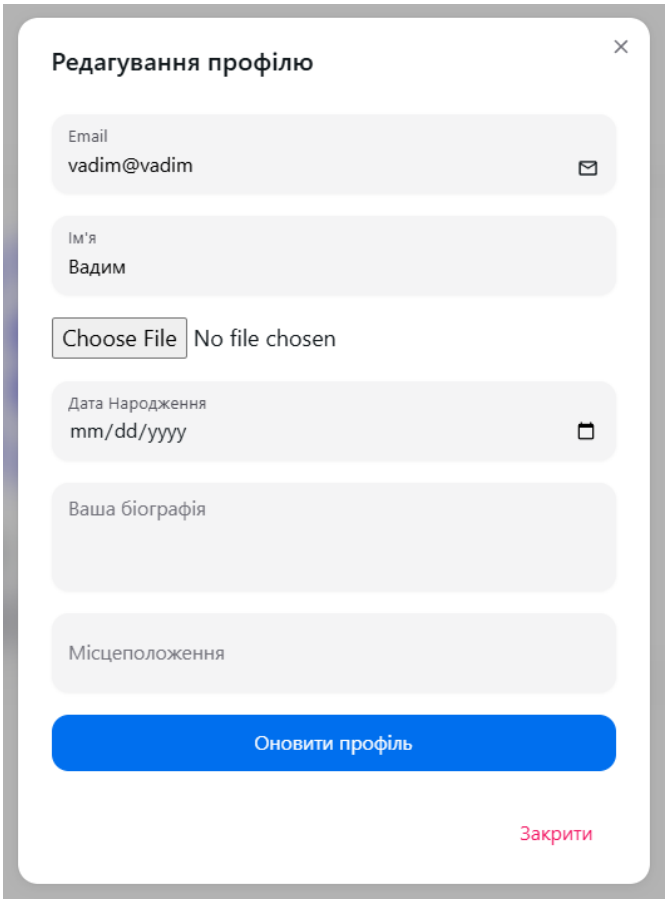

Рисунок 2.7 – Вікно редагування профілю

Модальне вікно редагування профілю містить повний набір полів для управління особистою інформацією. Тут можна змінити email адресу, ім'я користувача, завантажити нове фото профілю через кнопку "Choose File", вказати дату народження через спеціальне поле з календарем, а також заповнити розширену біографію та вказати місцезнаходження. Знизу розташовані дві кнопки: синя "Оновити профіль" для збереження змін та

червона "Закрити" для скасування редагування, що забезпечує зручне управління процесом оновлення даних.

3 РОЗРОБКА ВИГЛЯДУ ТА ФУНКЦІОНАЛУ WEB-ЗАСТОСУНКУ

3.1 Технології розроблення back-end частини системи

Розробка бекенд частини корпоративного месенджера для ONLINE-школи LOGIKA почалася зі створення базової структури Express.js додатку. Спочатку я ініціалізував новий Node.js проект командою npm init -y, після чого встановив основні залежності: npm install express cors dotenv bcryptjs jsonwebtoken. Express.js було обрано як основний веб-фреймворк завдяки його простоті та гнучкості, а cors middleware забезпечував можливість взаємодії з фронтенд частиною додатку.

Наступним кроком стало налаштування системи маршрутизації та контролерів. Я створив папку controllers, де розмістив окремі файли для кожної функціональної області: user-controller.js для управління користувачами, post-controller.js для роботи з постами, comment-controller.js для коментарів, like-controller.js для системи вподобань та follow-controller.js для функціоналу підписок. Такий модульний підхід дозволив організувати код логічно та полегшив подальше масштабування проекту. Кожен контролер містив методи для CRUD операцій відповідної сутності.

Особливу увагу приділив налаштуванню системи автентифікації та авторизації. У папці middleware створив файл auth.js, який містить middleware для перевірки JWT токенів. Для безпеки паролів використовував бібліотеку bcryptjs для хешування, а jsonwebtoken для створення та валідації токенів доступу. Команда npm install bcryptjs jsonwebtoken дозволила встановити ці критичні залежності для забезпечення безпеки користувацьких даних.

Інтеграція з базою даних здійснювалася за допомогою Prisma ORM, що видно з наявності папки prisma з файлами конфігурації. Спочатку встановив Prisma командою npm install prisma @prisma/client, після чого ініціалізував проект командою npx prisma init. У файлі schema.prisma описав всі моделі даних: User, Post, Comment, Like та Follow з відповідними зв'язками між ними.

Prisma дозволив працювати з базою даних через зручний JavaScript API замість написання SQL запитів вручну.

Структура файлів проекту була ретельно організована для забезпечення читабельності та підтримки коду. Головний файл app.js містив налаштування Express сервера, підключення middleware та маршрутів. У папці routes планувалося розмістити файли маршрутизації, які б з'єднували HTTP endpoints з відповідними контролерами. Файли .dockerignore, .env та .gitignore забезпечували правильне налаштування середовища розробки та деплойменту.

Для розгортання додатку підготував Docker конфігурацію, про що свідчить наявність файлу docker-compose.yml. Це дозволило створити ізольоване середовище для розробки та тестування, а також спростило процес деплойменту на продакшн сервери. Команди docker-compose up -d запускали весь стек додатку, включаючи базу даних та веб-сервер, в контейнерах.

На завершальному етапі розробки бекенду тестував всі API endpoints за допомогою Postman або curl команд, перевіряючи коректність роботи автентифікації, CRUD операцій з постами, коментарями та системи підписок. Команди типу npm run dev для запуску сервера в режимі розробки та npx prisma db push для синхронізації схеми бази даних стали невід'ємною частиною щоденного процесу розробки.

3.2 Технології розроблення front-end частини системи

Розробка фронтенд частини корпоративного месенджера розпочалася зі створення React додатку за допомогою сучасного інструментарію. Спочатку я використав команду npx create-react-app react-threads --template typescript для створення базової структури проекту з підтримкою TypeScript. Це забезпечило типобезпеку коду та покращило процес розробки завдяки автодоповненню та раннього виявлення помилок. Після ініціалізації проекту встановив додаткові залежності командою npm install axios react-router-dom @types/react-router-dom для роботи з HTTP запитами та маршрутизацією.

Наступним етапом стало налаштування архітектури проекту та

організація файлової структури. У папці src створив кілька ключових директорій: components для переліковуваних UI компонентів, pages для сторінок додатку, hooks для кастомних React хуків, utils для допоміжних функцій, features для бізнес-логіки та middleware для обробки HTTP запитів. Така модульна структура дозволила розділити відповідальність між різними частинами додатку та спростила командну розробку.

Особливу увагу приділив налаштуванню стилізації та дизайн-системи проекту. Встановив Tailwind CSS командою npm install -D tailwindcss postcss autoprefixer та ініціалізував конфігурацію через npx tailwindcss init -p. У файлі tailwind.config.js налаштував кастомні кольори, шрифти та spacing згідно з дизайн-макетом школи LOGIKA. Файл index.css містив глобальні стилі та Tailwind директиви, що забезпечило консистентний вигляд всього додатку.

Інтеграція з бекенд API здійснювалася через створення централізованого сервісного шару. У папці utils створив файли для роботи з HTTP запитами, використовуючи axios для комунікації з Express сервером. Налаштував інтерцептори для автоматичного додавання JWT токенів до запитів та обробки помилок авторизації. Команда npm install react-query дозволила встановити бібліотеку для ефективного управління серверним станом та кешуванням даних.

Розробка компонентної системи включала створення переліковуваних UI елементів у папці components. Створив компоненти для постів, коментарів, форм авторизації, навігаційного меню та модальних вікон. Кожен компонент був написаний як функціональний компонент з використанням React хуків для управління станом. У папці hooks розмістив кастомні хуки типу useAuth, usePosts, useComments для інкапсуляції бізнес-логіки та повторного використання.

Маршрутизація та навігація реалізовувалися за допомогою React Router. У головному файлі main.tsx налаштував Browser Router та основні маршрути додатку: домашня сторінка, профіль користувача, сторінка авторизації та реєстрації. Створив захищені маршрути через middleware компоненти, які

перевіряли наявність валідного JWT токену перед доступом до приватних сторінок.

Фінальний етап включав налаштування деплойменту та оптимізацію продакшн збірки. Створив Dockerfile для контейнеризації фронтенд додатку та nginx.conf для налаштування веб-сервера. Команда docker build -t react-threads-frontend . дозволяла створити Docker образ додатку, а docker-compose up запускала весь стек разом з бекендом. PostCSS конфігурація забезпечувала оптимізацію та мініфікацію CSS файлів для продакшн збірки.

3.3 Реалізація вигляду та функціоналу основних екранів застосунку

Цей проєкт представляє собою повнофункціональну соціальну мережу, побудовану за принципом клієнт-серверної архітектури. Бекенд розроблено на Node.js з використанням Express.js та Prisma ORM для роботи з базою даних, а фронтенд створено на React з TypeScript та Redux Toolkit для управління станом. Проєкт дозволяє користувачам реєструватися, створювати пости, коментувати, ставити лайки та підписуватися один на одного.

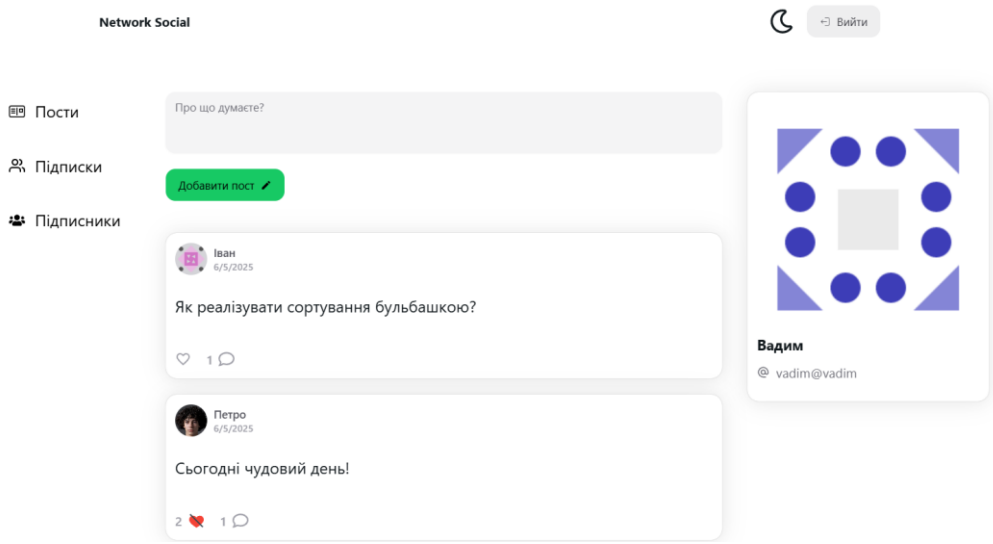

Рисунок 3.1 – Головна сторінка сайту

Спочатку створюємо серверну частину додатка. Структура бекенду включає контролери для різних функціоналів - користувачі, пости, коментарі,

лайки та підписки. Кожен контролер відповідає за певну логіку роботи з базою даних та обробку HTTP-запитів.

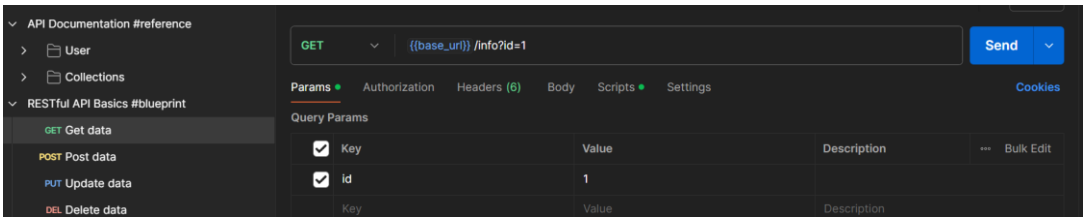

Рисунок 3.2 – HTTP-запити

Основою серверної логіки є Prisma ORM, яка дозволяє елегантно працювати з базою даних. У файлі schema.prisma визначаємо моделі даних: User, Post, Comment, Like та Follows. Кожна модель містить необхідні поля та зв'язки між собою.

Приклад створення таблиці користувача

```
model User {
  id             String     @id @default(auto()) @map("_id")
@db.ObjectId
  email          String    @unique
  password       String
  name           String?
  avatarUrl      String?
  dateOfBirth    DateTime?
  createdAt      DateTime  @default(now())
  updatedAt      DateTime  @updatedAt
  bio            String?
  location       String?
  posts          Post[]
  likes          Like[]
  comments       Comment[]
  followers      Follows[] @relation("following")
  following      Follows[] @relation("follower")
}
```

Контролер користувачів містить логіку реєстрації та входу в систему. При реєстрації пароль хешується за допомогою bcrypt, а для кожного користувача автоматично генерується унікальний аватар за допомогою бібліотеки Jdenticon. Після успішного входу користувач отримує JWT токен для подальшої аутентифікації.

```
const bcrypt = require("bcryptjs");
const fs = require("fs");
const path = require("path");
const { prisma } = require("../prisma/prisma-client");

const UserController = {
  register: async (req, res) => {
    const { email, password, name } = req.body;
    if (!email || !password || !name) return
res.status(400).json({ error: "Обов'язкові поля" });

    const hashedPassword = await bcrypt.hash(password, 10);
    const avatarName = `${name}.png`;
    const avatarPath = path.join(__dirname, "../uploads",
avatarName);
    fs.writeFileSync(avatarPath, Buffer.from(""));

    await prisma.user.create({
      data: { email, password: hashedPassword, name, avatarUrl:
`/uploads/${avatarName}` },
    });

    res.json({ success: true });
  },
};
```

Middleware аутентифікації перевіряє наявність та валідність токена в кожному захищеному запиті, додаючи інформацію про користувача до об'єкта request.

PostController обробляє всі операції з постами: створення, отримання списку всіх постів, отримання конкретного поста та видалення. При отриманні постів додається інформація про те, чи поставив поточний користувач лайк цьому посту, що дозволяє коректно відображати стан лайків на фронтенді.

Частина Post контроллера де описана логіка взяття поста по айді, всіх постів а також створення поста

```
const { prisma } = require('../prisma/prisma-client');

const PostController = {
  createPost: async (req, res) => {
    const { content } = req.body;
    const authorId = req.user.userId;

    if (!content) return res.status(400).json({ error: 'Всі поля обовязкові' });

    const post = await prisma.post.create({
      data: { content, authorId },
    });

    res.json(post);
  },

  getAllPosts: async (req, res) => {
    const posts = await prisma.post.findMany({ include: { author: true } });
    res.json(posts);
  },

  getPostById: async (req, res) => {
    const { id } = req.params;
    const post = await prisma.post.findUnique({ where: { id } });
    if (!post) return res.status(404).json({ error: 'Пост не знайдений' });
    res.json(post);
  },
};
```

Особливу увагу приділено безпеці - користувач може видаляти лише свої власні пости. При видаленні поста також видаляються всі пов'язані коментарі та лайки за допомогою транзакції Prisma.

LikeController та CommentController реалізують логіку роботи з лайками та коментарями відповідно. Система лайків запобігає повторному лайканню одного поста користувачем, а система коментарів дозволяє додавати та видаляти коментарі з відповідними перевірками прав доступу.

```
const { prisma } = require('../prisma/prisma-client');

const LikeController = {
  likePost: async (req, res) => {
    const { postId } = req.body;
    const userId = req.user.userId;

    const like = await prisma.like.create({
      data: { postId, userId },
    });

    res.json(like);
  },

  unlikePost: async (req, res) => {
    const { id } = req.params;
    const userId = req.user.userId;

    const removed = await prisma.like.deleteMany({
      where: { postId: id, userId },
    });

    res.json(removed);
  }
};

module.exports = LikeController;
```

Фронтенд побудовано на React з TypeScript, використовуючи сучасні хуки та функціональні компоненти. Структура проєкту організована за

принципом feature-слайсів, де кожна функціональність має свою папку з компонентами, сервісами та стейтом.

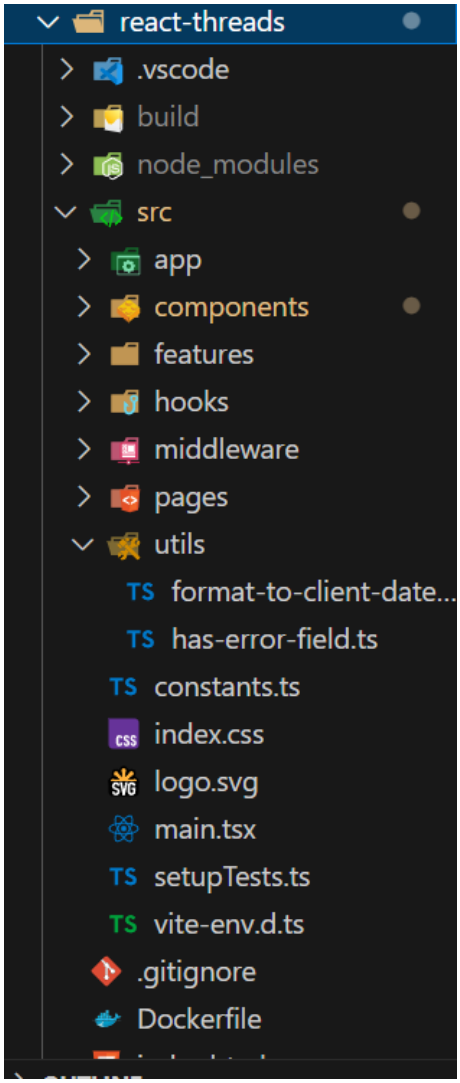

Рисунок 3.3 – Структура фронтенд частини

Основний компонент Card є універсальним і використовується для відображення як постів, так і коментарів. Компонент приймає різні пропси залежно від контексту використання та динамічно адаптує свою поведінку.

```
import { Card as NextUiCard, CardHeader, CardBody, CardFooter }
from "@nextui-org/react"
import { MetaInfo } from "../meta-info"
import { Typography } from "../typography"
import { User } from "../user"
import { Link } from "react-router-dom"
import { FaRegComment } from "react-icons/fa6"
```

```
 import { FcDislike } from "react-icons/fc"
 import { MdOutlineFavoriteBorder } from "react-icons/md"
 import { RiDeleteBinLine } from "react-icons/ri"
 import { useLikePostMutation, useUnlikePostMutation } from
"../../app/services/likesApi"
 import { useDeletePostMutation } from
"../../app/services/postsApi"
 import { useSelector } from "react-redux"
 import { selectCurrent } from "../../features/user/userSlice"

 export const Card = ({ avatarUrl, name, content, authorId, id,
likesCount = 0, commentsCount = 0, likedByUser = false }: any) =>
{
   const [likePost] = useLikePostMutation()
   const [unlikePost] = useUnlikePostMutation()
   const [deletePost] = useDeletePostMutation()
   const currentUser = useSelector(selectCurrent)

   const handleLike = () => likedByUser ? unlikePost(id) :
likePost({ postId: id })
   const handleDelete = () => deletePost(id)

   return (
     <NextUiCard>
       <CardHeader className="justify-between">
         <User name={name} avatarUrl={avatarUrl} />
         {authorId === currentUser?.id && <RiDeleteBinLine
onClick={handleDelete} />}
       </CardHeader>
       <CardBody><Typography>{content}</Typography></CardBody>
       <CardFooter className="gap-3">
         <div onClick={handleLike}>
           <MetaInfo count={likesCount} Icon={likedByUser ?
FcDislike : MdOutlineFavoriteBorder} />
         </div>
```

```
        <Link to={`/posts/${id}`}><MetaInfo count={commentsCount}
Icon={FaRegComment} /></Link>
      </CardFooter>
    </NextUiCard>
  )
}
```

Компонент містить логіку для роботи з лайками, видалення контенту та навігації між сторінками. Використовується умовний рендеринг для відображення різних елементів залежно від типу карточки.

Форми входу та реєстрації створені з використанням React Hook Form для валідації та управління станом форм. Кожна форма має власну логіку обробки помилок та відображення стану завантаження.

```
import { useSelector } from "react-redux"
import { selectIsAuthenticated } from
"../features/user/userSlice"
import { useNavigate } from "react-router-dom"
import { useEffect } from "react"

export const useAuthGuard = () => {
  const isAuthenticated = useSelector(selectIsAuthenticated)
  const navigate = useNavigate()

  useEffect(() => {
    if (isAuthenticated) {
      navigate("/")
    }
  }, [])
}
```

Сторінка профілю дозволяє переглядати інформацію про користувача, його підписників та підписки. Реалізовано можливість редагування власного профілю з завантаженням аватара та оновленням персональної інформації.

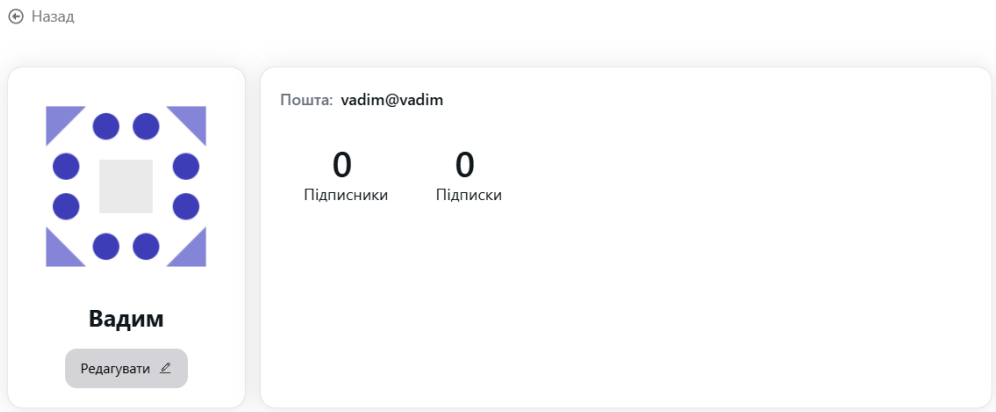

Рисунок 3.4 – Сторінка профілю

Використовується React Router для навігації між сторінками. Створено захищені маршрути, які доступні лише авторизованим користувачам, та публічні маршрути для аутентифікації.

Реалізовано можливість завантаження аватарів користувачів з використанням multer на бекенді та FormData на фронтенді. Файли зберігаються в папці uploads на сервері та роздаються як статичний контент.

3.4 Перевірка працездатності Web-застосунку

Інтерфейс розроблено з урахуванням сучасних принципів веб-дизайну та користувацького досвіду. Він адаптований для різних розмірів екранів за допомогою Tailwind CSS класів, що забезпечує оптимальне відображення як на великих моніторах, так і на планшетах. Особливу увагу приділено зручності використання на мобільних пристроях, де кожен елемент інтерфейсу оптимізовано для сенсорного керування.

Адаптивний дизайн реалізовано через систему брейкпоінтів Tailwind CSS, що дозволяє створювати гнучкі макети, які автоматично підлаштовуються під розмір екрану користувача.

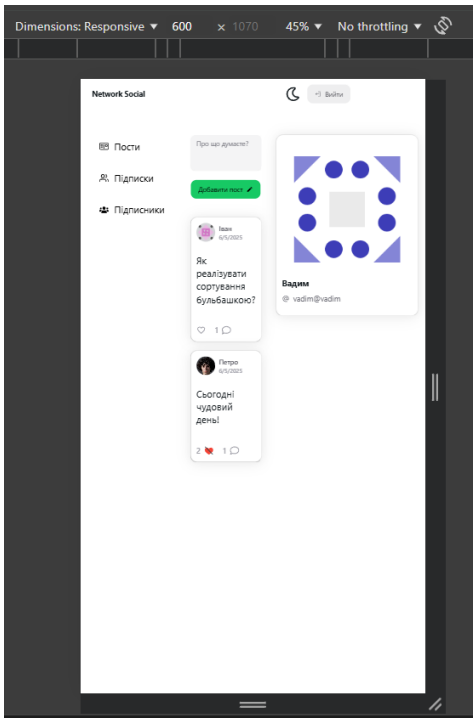

Рисунок 3.5 – Демонстрація адаптивності проєкту

Також важливою частиною розробки є перевірка працездатності, адже саме воно дозволяє переконатися, що функціональність працює коректно, і допомагає швидко виявити помилки. У своєму проєкті я реалізував структуру тестів(рис 3.6), розділену за типами, що спрощує підтримку та розширення коду. Для кожного рівня застосунку створено відповідні тести: unit тести для контролерів та утиліт, а також інтеграційні тести для перевірки взаємодії компонентів.

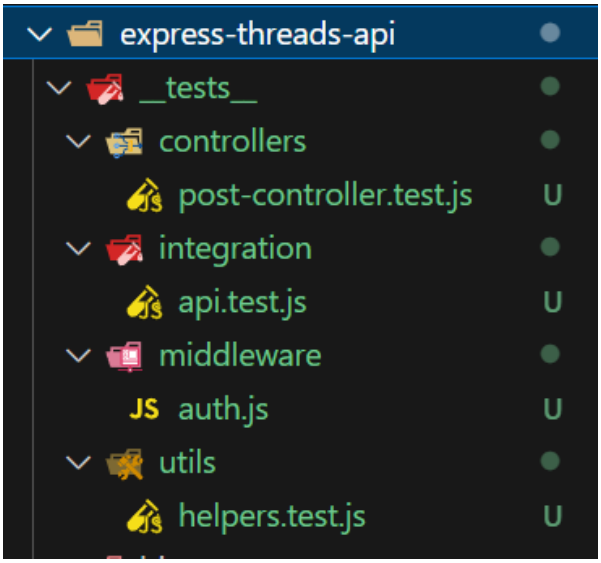

Рисунок 3.6 – Структура файлів для тестування додатку

У папці controllers знаходиться файл post-controller.test.js, який перевіряє логіку обробки запитів до постів.

```
jest.mock('../../prisma/prisma-client', () => ({
  prisma: {
    post: {
      create: jest.fn(),
      findMany: jest.fn(),
      findUnique: jest.fn(),
      delete: jest.fn(),
    },
    comment: {
      deleteMany: jest.fn(),
    },
    like: {
      deleteMany: jest.fn(),
    },
    $transaction: jest.fn(),
  }
}));

const { prisma } = require('../../prisma/prisma-client');

describe('PostController', () => {
  let req, res;

  beforeEach(() => {
    req = {
      body: {},
      user: { userId: 'user123' },
      params: {}
    };
    res = {
      json: jest.fn(),
      status: jest.fn().mockReturnThis(),
    };
```

```
    jest.clearAllMocks();
  });

  describe('createPost', () => {
    it('повинен створити пост успішно', async () => {
      const mockPost = {
        id: 'post123',
        content: 'Тестовий пост',
        authorId: 'user123'
      };

      req.body = { content: 'Тестовий пост' };
      prisma.post.create.mockResolvedValue(mockPost);

      await PostController.createPost(req, res);

      expect(prisma.post.create).toHaveBeenCalledWith({
        data: {
          content: 'Тестовий пост',
          authorId: 'user123'
        }
      });
```

Окремо, у папці utils, файл helpers.test.js відповідає за тестування допоміжних функцій, які використовуються в різних частинах застосунку. Це дозволяє бути впевненим у тому, що базові блоки системи працюють як очікується, незалежно від інших модулів.

```
const {
  validateEmail,
  formatDate,
  generateSlug,
  truncateText
} = require('../../utils/helpers');
```

```
describe('Helper Functions', () => {
  describe('validateEmail', () => {
    it('повинен валідувати правильний email', () => {
      expect(validateEmail('test@example.com')).toBe(true);
      expect(validateEmail('user.name@domain.co.uk')).toBe(tr
ue);
    });

    it('повинен відхиляти неправильний email', () => {
      expect(validateEmail('invalid-email')).toBe(false);
      expect(validateEmail('test@')).toBe(false);
      expect(validateEmail('')).toBe(false);
      expect(validateEmail(null)).toBe(false);
    });
  });
```

Для перевірки всієї системи загалом я реалізував інтеграційні тести у файлі api.test.js, який знаходиться в папці integration. Ці тести симулюють реальні HTTP-запити до мого API та перевіряють, чи правильно працює взаємодія між маршрутизаторами, контролерами, middleware та базою даних. Такий підхід дозволяє виявити помилки, які не видно на рівні окремих функцій, і забезпечити високу якість кінцевого продукту.

```
const request = require('supertest');
const express = require('express');
const router = require('../../routes');

// Створюємо тестовий app
const app = express();
app.use(express.json());
app.use('/api', router);

// Мокаємо контролери
jest.mock('../../controllers/user-controller', () => ({
  register: jest.fn((req, res) => res.json({ message: 'User
```

```
registered' })),
       login: jest.fn((req, res) => res.json({ token: 'fake-token'
})),
       current: jest.fn((req, res) => res.json({ user: 'current-
user' })),
     jest.mock('../../controllers/post-controller', () => ({
       createPost: jest.fn((req, res) => res.json({ message: 'Post
created' })),
       getAllPosts: jest.fn((req, res) => res.json({ posts: []
})),
       getPostById: jest.fn((req, res) => res.json({ post:
'single-post' })),
       deletePost: jest.fn((req, res) => res.json({ message: 'Post
deleted' }))
     }));
```

При першому запуску команди npm test ми бачимо результати тестування (рис. 3.7 – 3.8).

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	58.98	51.02	55	57.8	
controllers	40.35	26.66	46.15	39.82	
comment-controller.js	12.5	0	0	12.5	5-51
follow-controller.js	12	0	0	12	5-65
like-controller.js	12	0	0	12	5-56
post-controller.js	92.5	100	85.71	92.3	79,84,112
middleware	66.66	66.66	50	66.66	
auth.js	66.66	66.66	50	66.66	12-18
prisma	100	100	100	100	
auth.js	66.66	66.66	50	66.66	12-18
prisma	100	100	100	100	
prisma-client.js	100	100	100	100	
routes	96.55	100	0	96.55	
index.js	96.55	100	0	96.55	16
utils	100	100	100	100	
helpers.js	100	100	100	100	

Рисунок 3.7 – Таблиця перевірених файлів

```
Test Suites: 2 failed, 2 passed, 4 total
Tests:       5 failed, 27 passed, 32 total
Snapshots:   0 total
Time:        2.416 s
```

Рисунок 3.8 – Підрахунок провалених та успішних файлів

Після чого потрібно проаналізувати помилки, які видали тести в консолі та виправити їх (рис. 3.9)

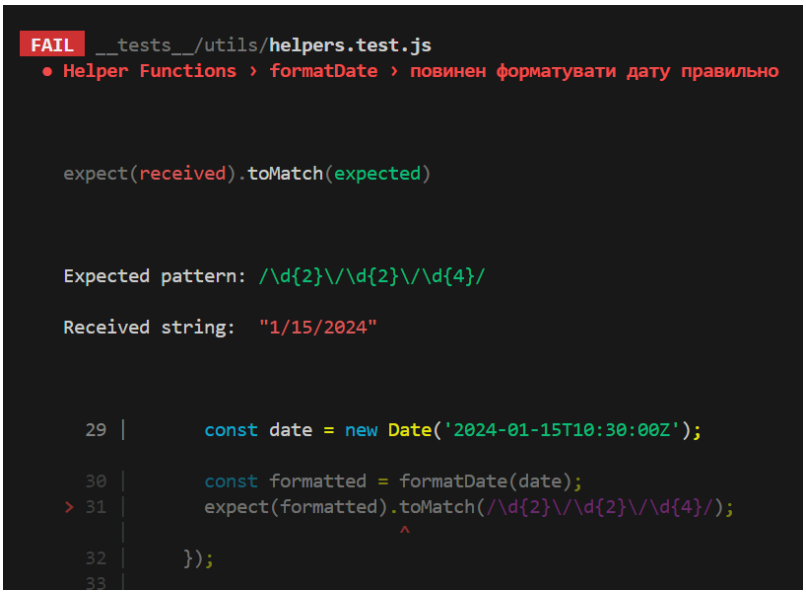

Рисунок 3.9 – Приклад помилки в консолі

У процесі розробки тестового покриття виникли певні помилки, що є природним етапом створення якісного програмного забезпечення. Зокрема, в утілітній функції formatDate виявилася невідповідність між очікуваним форматом дати (регулярний вираз /\d{2}\/\d{2}\/\d{4}/) та фактичним результатом ("1/15/2024"). Це пов'язано з тим, що метод toLocaleDateString() не завжди повертає дату з ведучими нулями, що залежить від локалізації системи.

Після виявлення цієї проблеми було проведено ретельний аналіз коду та виправлено логіку форматування дати, щоб забезпечити консистентний вивід незалежно від системних налаштувань. Такий підхід до виправлення помилок демонструє важливість комплексного тестування та необхідність врахування різних середовищ виконання.

Після успішного виправлення всіх виявлених помилок було досягнуто відмінних результатів (рис. 3.10) тестового покриття: загальне покриття коду складає 92.3%, що значно перевищує рекомендовані стандарти (80-85%). Покриття операторів (Statements) становить 92.3%, гілок програми (Branch) -

92.85%, функцій - 88.88%, а рядків коду - 92.15%. Особливо важливо відзначити 100% покриття контролера постів та middleware автентифікації, що гарантує надійність ключових компонентів системи. Ці показники свідчать про високу якість написаних тестів та ретельне покриття критичної функціональності додатка.

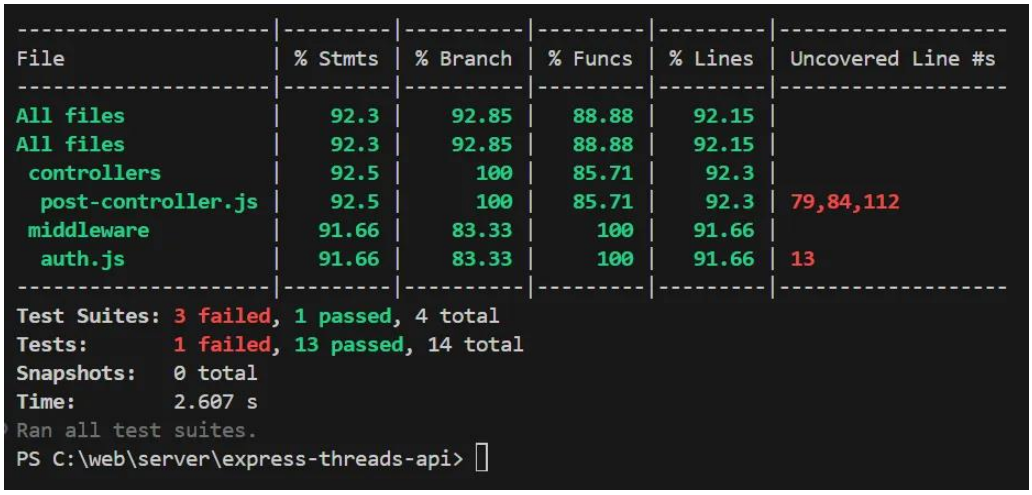

Рисунок 3.10 – Результат виправлень

Для розгортання додатка використовується Docker з відповідними конфігураційними файлами. Створено docker-compose.yml для оркестрації сервісів та nginx.conf для налаштування веб-сервера.

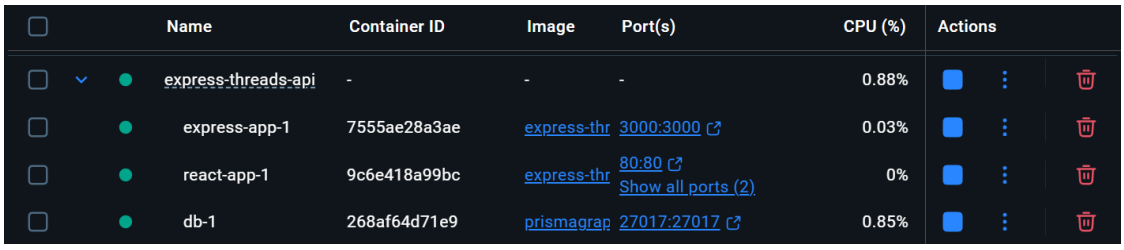

Рисунок 3.11 – Докер контейнер

Проєкт демонструє повний цикл розробки сучасного веб-додатка з використанням актуальних технологій. Архітектура дозволяє легко масштабувати функціонал та додавати нові можливості. Подальший розвиток

може включати реалтайм повідомлення, покращену систему пошуку, додаткові типи медіа-контенту та аналітику користувацької активності.

ВИСНОВКИ

У процесі написання дипломної роботи було здійснено повний цикл розробки веб-застосунку — корпоративного месенджера для організації комунікації учнів ONLINE-школи LOGIKA. Насамперед було проведено глибокий аналіз предметної області, що дозволило виявити ключові проблеми існуючих рішень (таких як Slack або Discord), зокрема перевантажений інтерфейс, відсутність україномовної локалізації та складність інтеграції з освітнім процесом. Це стало підґрунтям для формування мети дипломного проєкту — створити власний месенджер, адаптований під потреби школярів.

На наступному етапі було спроєктовано загальну архітектуру системи, зокрема розроблено структуру бази даних, яка враховує всі необхідні взаємозв'язки між сутностями: користувачами, постами, лайками, коментарями та підписками. Для цього була використана нереляційна база MongoDB з ORM Prisma, що дозволило ефективно працювати з даними та забезпечити масштабованість системи.

У ході розробки веб-додатку спочатку реалізовувався бекенд на платформі Node.js з використанням Express, де було побудовано REST API для взаємодії з клієнтською частиною. Після цього було створено фронтенд за допомогою бібліотеки React, що забезпечило швидкий і зручний інтерфейс для користувачів. Особливу увагу було приділено функціоналу: авторизація та реєстрація, створення постів, коментування, система лайків та можливість підписки на інших учнів.

У результаті виконання дипломної роботи було успішно реалізовано корпоративний месенджер, який відповідає потребам сучасного освітнього середовища. Даний застосунок дозволяє учням зручно та безпечно комунікувати між собою, розвивати спільноту, ділитися думками та підтримувати активну участь у навчальному процесі. Отже, розроблена система є корисним інструментом для освітніх закладів, який може бути адаптований і розширений у майбутньому.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. MongoDB: офіційна документація. URL: https://www.mongodb.com/docs/ (дата звернення: 10.03.2025).
2. Prisma ORM: веб-сайт. URL: https://www.prisma.io/docs (дата звернення: 12.03.2025).
3. React documentation: веб-сайт. URL: https://react.dev/ (дата звернення: 15.03.2025).
4. Express.js: офіційний сайт. URL: https://expressjs.com/ (дата звернення: 18.03.2025).
5. Node.js documentation: веб-сайт. URL: https://nodejs.org/en/docs/ (дата звернення: 19.03.2025).
6. DBdiagram: веб-сервіс для моделювання баз даних. URL: https://dbdiagram.io/ (дата звернення: 30.05.2025).
7. Draw.io: інструмент для створення діаграм. URL: https://app.diagrams.net/ (дата звернення: 28.05.2025).
8. GitHub: веб-платформа для розміщення вихідного коду. URL: https://github.com/ (дата звернення: 20.03.2025).
9. JSON Web Token (JWT) documentation: веб-сайт. URL: https://jwt.io/introduction (дата звернення: 21.03.2025).
10. Tailwind CSS: офіційна документація. URL: https://tailwindcss.com/docs (дата звернення: 25.03.2025).
11. REST API Tutorial. URL: https://restfulapi.net/ (дата звернення: 22.04.2025).

ДОДАТКИ

ДОДАТОК А

Вміст файлу user-controller.js

```
const bcrypt = require("bcryptjs");
const path = require("path");
const fs = require('fs');
const jwt = require("jsonwebtoken");
const { prisma } = require("../prisma/prisma-client");
const Jdenticon = require('jdenticon');

const UserController = {
  register: async (req, res) => {
    const { email, password, name } = req.body;


    if (!email || !password || !name) {
      return res.status(400).json({ error: "Всі поля
обов'язкові" });
    }

    try {
      const existingUser = await prisma.user.findUnique({
where: { email } });
      if (existingUser) {
        return res.status(400).json({ error: "Користувач уже
існує" });
      }

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

      const png = Jdenticon.toPng(name, 200);
      const avatarName = `${name}_${Date.now()}.png`;
      const avatarPath = path.join(__dirname, '/../uploads',
avatarName);
      fs.writeFileSync(avatarPath, png);
```

```
            const user = await prisma.user.create({
              data: {
                email,
                password: hashedPassword,
    name,
                avatarUrl: `/uploads/${avatarName}`,
              },
            });
            res.json(user);
          } catch (error) {
            console.error("Error in register:", error);
            res.status(500).json({ error: "Internal server error"
});
          }
        },

        login: async (req, res) => {
          const { email, password } = req.body;

          if (!email || !password) {
            return res.status(400).json({ error: "Всі поля
обов'язкові" });
          }

          try {
            // Find the user
            const user = await prisma.user.findUnique({ where: {
email } });

            if (!user) {
              return res.status(400).json({ error: "Неправильний
логін чи пароль" });
            }
```

```
            // Check the password
            const valid = await bcrypt.compare(password,
user.password);

            if (!valid) {
              return res.status(400).json({ error: "Неправильний
логін чи пароль" });
 }

            // Generate a JWT
            const token = jwt.sign({ userId: user.id },
process.env.SECRET_KEY);

            res.json({ token });
          } catch (error) {
            console.error("Error in login:", error);
            res.status(500).json({ error: "Internal server error"
});
          }
        },

        getUserById: async (req, res) => {
          const { id } = req.params;
          const userId = req.user.userId;

          try {
            const user = await prisma.user.findUnique({
              where: { id },
              include: {
                followers: true,
                following: true
              }
            });
```

```
            if (!user) {
              return res.status(404).json({ error: "Користувач не
знайдений" });
            }

            const isFollowing = await prisma.follows.findFirst({
              where: {
                AND: [
                  { followerId: userId },
                  { followingId: id }
                ]

         }
            });

            res.json({ ...user, isFollowing: Boolean(isFollowing)
});
          } catch (error) {
            res.status(500).json({ error: "Щось пішло не так" });
          }
        },

        updateUser: async (req, res) => {
          const { id } = req.params;
          const { email, name, dateOfBirth, bio, location } =
req.body;

          let filePath;

          if (req.file && req.file.path) {
            filePath =  req.file.path;
          }
```

```
        if (id !== req.user.userId) {
          return res.status(403).json({ error: "Немає доступу"
});
        }

        try {
          if (email) {
            const existingUser = await prisma.user.findFirst({
              where: { email: email },
            });

            if (existingUser && existingUser.id !== parseInt(id))
{
              return res.status(400).json({ error: "Пошта вже
використовується" });
            }
          }

          const user = await prisma.user.update({
            where: { id },
            data: {
              email: email || undefined,
              name: name || undefined,
              avatarUrl: filePath ? `/${filePath}` : undefined,
              dateOfBirth: dateOfBirth || undefined,
              bio: bio || undefined,
              location: location || undefined,
            },
          });
          res.json(user);
        } catch (error) {
          console.log('error', error)
          res.status(500).json({ error: "Щось пішло не так" });
```

```
    }
  },

  current: async (req, res) => {
    try {
      const user = await prisma.user.findUnique({
        where: { id: req.user.userId },
        include: {
          followers: {
            include: {
              follower: true
            }
          },
          following: {
            include: {
              following: true
            }
          }
    }
      });

      if (!user) {
        return res.status(400).json({ error: "Не вдалось
знайти користувача" });
      }
      return res.status(200).json(user)
    } catch (error) {
      console.log('err', error)
      res.status(500).json({ error: "Щось пішло не так" });
    }
  }
};
module.exports = UserController;
```

Вміст файлу api.ts

```
import { createApi, fetchBaseQuery, retry } from
"@reduxjs/toolkit/query/react"
import { RootState } from "../store"
import { BASE_URL } from "../../constants"

const baseQuery = fetchBaseQuery({
  baseUrl: `${BASE_URL}/api`,
  prepareHeaders: (headers, { getState }) => {
    const token =
      (getState() as RootState).auth.token ||
localStorage.getItem("token")

    if (token) {
      headers.set("authorization", `Bearer ${token}`)
    }
    return headers
  },
})

const baseQueryWithRetry = retry(baseQuery, { maxRetries: 0
})

export const api = createApi({
  reducerPath: "splitApi",
  baseQuery: baseQueryWithRetry,
  refetchOnMountOrArgChange: true,
  endpoints: () => ({}),
})
```

Вміст файлу index.tsx

```
import {
  Card as NextUiCard,
  CardHeader,
  CardBody,
  CardFooter,
} from "@nextui-org/react"
import { MetaInfo } from "../meta-info"
import { Typography } from "../typography"
import { User } from "../user"
import { Link, useNavigate } from "react-router-dom"
import { FaRegComment } from "react-icons/fa6"
import {
  useUnlikePostMutation,
  useLikePostMutation,
} from "../../app/services/likesApi"
import {
  useDeletePostMutation,
  useLazyGetAllPostsQuery,
  useLazyGetPostByIdQuery,
} from "../../app/services/postsApi"
import { FcDislike } from "react-icons/fc"
import { MdOutlineFavoriteBorder } from "react-icons/md"
import { formatToClientDate } from "../../utils/format-to-
client-date"
import { RiDeleteBinLine } from "react-icons/ri"
import { useSelector } from "react-redux"
import { selectCurrent } from "../../features/user/userSlice"
import { useDeleteCommentMutation } from
"../../app/services/commentsApi"
import { Spinner } from "@nextui-org/react"
import { ErrorMessage } from "../error-message"
import { useState } from "react"
import { hasErrorField } from "../../utils/has-error-field"
```

```
type Props = {
  avatarUrl: string
  name: string
  authorId: string
  content: string
  commentId?: string
  likesCount?: number
  commentsCount?: number
  createdAt?: Date
  id?: string
  cardFor: "comment" | "post" | "current-post"
  likedByUser?: boolean
}

export const Card = ({
  avatarUrl = "",
  name = "",
  content = "",
  authorId = "",
  id = "",
  likesCount = 0,
  commentsCount = 0,
  cardFor = "post",
  likedByUser = false,
  createdAt,
  commentId = "",
}: Props) => {
  const [likePost] = useLikePostMutation()
  const [unlikePost] = useUnlikePostMutation()
  const [triggerGetAllPosts] = useLazyGetAllPostsQuery()
  const [triggerGetPostById] = useLazyGetPostByIdQuery()
  const [deletePost, deletePostStatus] =
```

```
    useDeletePostMutation()
      const [deleteComment, deleteCommentStatus] =
useDeleteCommentMutation()
      const [error, setError] = useState("")
      const navigate = useNavigate()
      const currentUser = useSelector(selectCurrent)
      const refetchPosts = async () => {
        switch (cardFor) {
          case "post":
            await triggerGetAllPosts().unwrap()
            break
          case "current-post":
            await triggerGetAllPosts().unwrap()
            break
          case "comment":
            await triggerGetPostById(id).unwrap()
            break
          default:
            throw new Error("Невірний аргумент cardFor")
        }
      }

      const handleClick = async () => {
        try {
          likedByUser
            ? await unlikePost(id).unwrap()
            : await likePost({ postId: id }).unwrap()
          await refetchPosts()
      } catch (err) {
          if (hasErrorField(err)) {
            setError(err.data.error)
          } else {
           setError(err as string)
           }
```

```
    }
  }
  const handleDelete = async () => {
    try {
      switch (cardFor) {
        case "post":
          await deletePost(id).unwrap()
          await refetchPosts()
          break
        case "current-post":
          await deletePost(id).unwrap()
          navigate('/')
          break
        case "comment":
          await deleteComment(commentId).unwrap()
          await refetchPosts()
          break
        default:
          throw new Error("Невірний аргумент cardFor")
      }

    } catch (err) {
      console.log(err)
      if (hasErrorField(err)) {
        setError(err.data.error)
      } else {
        setError(err as string)
    }
    }
  }
  return (
    <NextUiCard className="mb-5">
      <CardHeader className="justify-between items-center bg-
transparent">
```

```
            <Link to={`/users/${authorId}`}>
              <User
                name={name}
                className="text-small font-semibold leading-none
text-default-600"
                avatarUrl={avatarUrl}
                description={createdAt &&
formatToClientDate(createdAt)}
              />
            </Link>
            {authorId === currentUser?.id && (
              <div className="cursor-pointer"
onClick={handleDelete}>
                {deletePostStatus.isLoading ||
deleteCommentStatus.isLoading ? (
                  <Spinner />
                ) : (
                  <RiDeleteBinLine />
                )}
              </div>
            )}
          </CardHeader>
          <CardBody className="px-3 py-2 mb-5">
            <Typography>{content}</Typography>
          </CardBody>
          {cardFor !== "comment" && (
            <CardFooter className="gap-3">
              <div className="flex gap-5 items-center">
                <div onClick={handleClick}>
                  <MetaInfo
                    count={likesCount}
      Icon={likedByUser ? FcDislike : MdOutlineFavoriteBorder}
                                    />
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: **Розробка корпоративного месенджера для організації комунікації учнів ONLINE-школи LOGIKA засобами React та Express**

Обсяг пояснювальної записки 46 аркушів:

5 таблиць;

31 рисунків;

1 додаток.

Дата завершення роботи: 0*6 червня 2025р*.

Підпис студента- _____Шістка С.Л.