МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

_____ Ігор ШЕЛЕХОВ
(підпис)

_____ червня 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавр

зі спеціальності 122 – Комп'ютерних наук,

освітньо-професійної програми «Інформатика»

на тему: «Інформаційна система накопичення, узагальнення і розповсюдження корисних порад»

здобувача групи ІН-91 Чупіки Артема Миколайовича

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

_____ Артем ЧУПІКА
(підпис)

Керівник,

асистент кафедри комп'ютерних наук

кандидат фізико-математичних наук

Олександр ВЛАСЕНКО _____
(підпис)

Суми - 2023

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

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

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН-91 Чупіки Артема Миколайовича

1. Тема роботи: «Інформаційна система накопичення, узагальнення і розповсюдження корисних порад»
затверджую наказом по СумДУ від _____
2. Термін здачі здобувачем кваліфікаційної роботи *до 09 червня 2023 року*
3. Вхідні дані до кваліфікаційної роботи _
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Інформаційний огляд. 2) Постановка задачі 3) Вибір методів розв'язання поставленої задачі 4) Програмна реалізація 5) Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)____
6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

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

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Інформаційний огляд*		
2	*Постановка задачі*		
3	*Вибір методів розв'язання поставленої задачі*		
4	*Програмна реалізація*		
5	*Висновки*		

Здобувач вищої освіти _____ (підпис) Керівник _____ (підпис)

АНОТАЦІЯ

Записка: 96 стр., 15 рис., 17 використаних джерел.

Обґрунтування актуальності теми роботи — тема кваліфікаційної роботи є актуальною, оскільки присвячена розробці системи, яка дозволяє людям поширювати корисні поради, тим самим полегшуючі їм життя.

Об’єкт дослідження — інформаційна система накопичення, узагальнення і розповсюдження корисних порад.

Мета роботи — проєктування та розробка інформаційної системи накопичення, узагальнення і розповсюдження корисних порад.

Методи дослідження — аналіз та оцінка ефективності інформаційної системи для накопичення, узагальнення і розповсюдження корисних порад.

Результати — розроблено інформаційну систему, що забезпечує збір, аналіз та узагальнення корисних порад. Система дозволяє користувачам зареєструватися, авторизуватися та створювати пости до відповідних категорій. Для розробки інформаційної системи були використані технології та інструменти, такі як PostgreSql для зберігання даних, Express для реалізації серверної частини та React Native для створення мобільного додатку.

ІНФОРМАЦІЙНА СИСТЕМА, NODEJS, EXPRESS, POSTGRESQL, JAVASCRIPT, REACT NATIVE.

ЗМІСТ

ВСТУП

Сучасні люди можуть не розбиратися в багатьох сферах або бути не достатньо досвідченим, але це не заважає нікому жити повним життям. Якщо буде потрібно то кожен зможе наприклад замінити лампочку або прибрати вдома. В разі коли людина не знає як зробити якусь роботу, їй на допомогу приходить Інтернет.

Більшість людей можуть використати свій комп’ютер або телефон, щоб знайти потрібну інформацію. За статистикою запит зі словами «how to make» має від 81 до 100 балів популярності [3]. Тобто з виконанням більшості завдань людина впорається навіть без допомоги спеціалістів.

Але залишається проблема відсіювання зайвої інформації. Наразі дуже важко відрізнити знання які допоможуть від тих які не підходять, застарілі або взагалі не працюють. Звісно є статті або експерти, які викривають неправдиву інформацію [17]. Все ж таки на це потрібен час, а людина яка шукає інформацію в інтернеті цього часу не має.

Зараз популярність дістається мобільним додаткам в яких є тільки корисна інформація або люди які діляться власним досвідом. На жаль все це розкидано по різним додаткам і все одно, щоб знайти потрібну інформацію треба витратити достатньо власного часу.

Мета даної роботи і виходить з цих проблем. Тобто створити мобільний додаток який буде давати можливість людям знаходити потрібну інформацію, яка буде базуватися на досвіді інших людей. Це допоможе швидше вирішити незнайому проблему та не буде займати багато часу.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Аналіз принципів побудови мобільних-додатків

Процес розробки мобільного додатку повинен бути продуманий з самого початку. Одразу треба будувати таку структуру проекту яка може розширятися в майбутньому та бути комфортною для розробки вже зараз [15].

На даний момент популярна модульна архітектура мобільних додатків (рис.1.1).

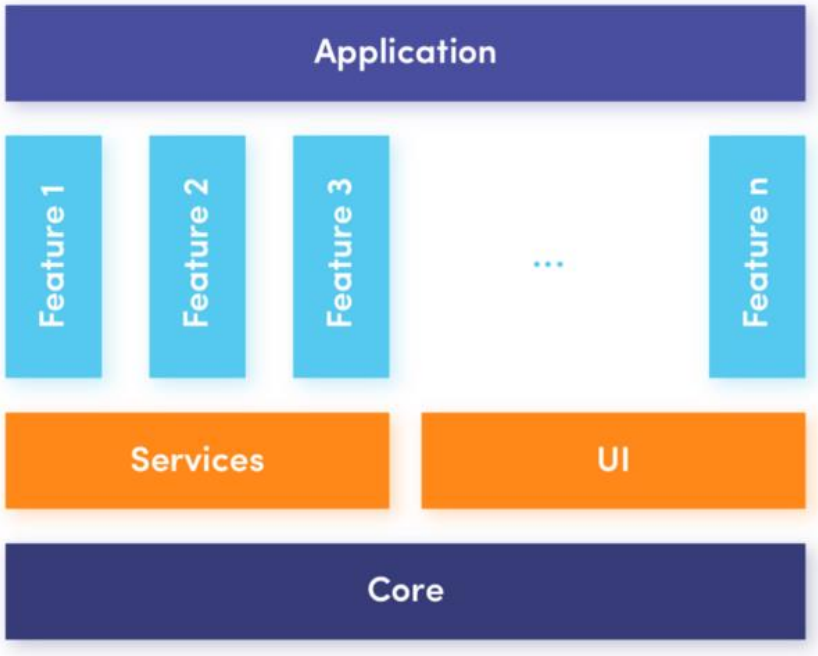

Рисунок 1.1 – Модульна архітектура [6]

Вона дає можливість додавати нові функціональні частини без змін в старих. Також при використанні модульної структури розробнику досить зручно використовувати модулі один в іншому.

Але потрібно побудувати чітку ієрархію та залежності між ними. Завдяки цьому можливо досягти функціональної правильності в роботі додатку та залишити можливість для подальшого розширення.

Також є принцип розподілення логіки, даних та функціональних частин. Це робиться для того щоб не було великих кусків неконтрольованої логіки роботи. Наприклад, одну сторінку в мобільному додатку можна розбити на багато частин та імпортувати їх в декілька сторінок де є повторення інтерфейсу або логіки. Так можна зробити код програми більш легким для розуміння та розширяє можливості зменшення коду взагалі.

Дуже важливе є асинхронне завантаження даних. Користувачу було б не приємно чекати мінімум по декілька секунд у відповідь на кожну дію в додатку. Треба робити так, щоб при відправленні та отриманні запитів користувач міг вільно користуватися іншими частинами додатку.

Важливу роль в принципах побудови мобільних додатків грає безпека даних користувача. Потрібно використовувати методи шифрування та дешифрування даних, а також забезпечити індивідуальну безпеку кожному аккаунту. Користувачі знаючі, що використання додатку не зробить їх особисті дані публічними, мають більшу довіру до додатку.

1.2 Інструменти для розробки мобільних-додатків

Способів розробки мобільних додатків досить багато. Починаючі з язиків програмування під одну конкретну платформу до кросплатформних. Переваги нативних язиків програмування це швидкодія, оптимізація та можливість використання нативних модулей. Але у випадку використання нативної мови є головний мінус – це неможливість використання додатку на іншій платформі. Ця проблема вирішується при використанні кросплатформної мови програмування.

Всім відомо що є декілька основних платформ, Android та IOS. Для того щоб написати мобільний додаток під Android потрібно використати одну з мов програмування Java або Kotlin. На даний момент більшість програмістів віддають перевагу Kotlin. Інструменти розробки для даних мов виступають Android Studio, Android IDE, IntellijIdea.

Для платформи IOS можна використати також дві мови – це Objective C та Swift. Наразі також програмісти віддають перевагу Swift. В такому разі інструментами розробки виступають Xcode, AppCode, Atom.

Але далеко не всі компанії, які розробляють мобільні додатки, дають перевагу нативним мовам програмування. Тому що це більш затратно в плані грошей та часу. Та звісно ніяк не вийде викласти оновлення одночасно під дві платформи.

У таких випадках на допомогу приходять косплатформені мови програмування або фреймворки, які мають такі ж можливості. До таких можна віднести JavaScript, C#, C, C++, Ruby, Python.

Сьогодні існує кілька JavaScript фреймворків, спеціально призначених для мобільних платформ, таких як Ionic 2 та React Native. За допомогою цих фреймворків і бібліотек дуже легко розробляти кросплатформні мобільні додатки. Це означає, що вам потрібно написати тільки одну версію програми, і вона буде працювати на iOS або Android [12].

1.3 Постановка задачі

Завданням даного дипломного проекту є проєктування та розробка інформаційної системи, спрямованої на накопичення та поширення корисних порад серед користувачів. Основною метою цієї системи є надання користувачам зручного та ефективного інструменту для отримання цінної інформації, взаємодії з іншими користувачами та збагачення своїх знань та навичок.

Для досягнення поставленої мети в рамках дипломного проекту необхідно виконати наступні завдання:

1. Визначити функціональні та нефункціональні вимоги до інформаційної системи, зокрема, створення механізму авторизації та реєстрації користувачів, можливість створення та категоризації порад, взаємодії та обміну порадами між користувачами.

2. Розробити архітектуру системи, включаючи вибір необхідних технологій та інструментів, таких як база даних, фреймворки для реалізації серверної та клієнтської частин, інтерфейс користувача тощо.

3. Реалізувати функціональність системи, забезпечуючи можливість створення та перегляду порад, коментування та поширення порад серед користувачів, а також інші необхідні функції, що сприяють зручності та корисності використання системи.

4. Документувати процес розробки, включаючи опис постановки задачі, аналіз вимог, архітектуру системи, реалізований функціонал, результати тестування та оцінку ефективності системи.

Виконання поставлених завдань дозволить розробити інформаційну систему, яка забезпечить користувачам зручну та доступну платформу для накопичення, узагальнення та розповсюдження корисних порад.

2. ВИБІР МЕТОДІВ РОЗВ'ЯЗАННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

2.1 Вибір мови програмування для розробки

За основну мову програмування для розробки цього мобільного додатку було віддано перевагу JavaScript. JavaScript є легкою, інтерпретованою мовою програмування з об'єктно-орієнтованими можливостями та першокласними функціями. Вона найбільш відома як мова сценаріїв для розробки веб-сторінок, але також знаходить широке застосування в різних середовищах, не обмежених браузером. JavaScript базується на прототипному підході до програмування, що дозволяє використовувати кілька парадигм програмування, зокрема об'єктно-орієнтований, імперативний та функціональний стиль.

Основний синтаксис мови JavaScript був спеціально розроблений з використанням конструкцій, подібних до мов Java і C++, з метою зменшення кількості нових концепцій, що потребуються для освоєння мови. Умовні оператори, такі як if, цикли for і while, а також блоки switch і try ... catch, функціонують так само, як в згаданих мовах програмування. Використання JavaScript не обмежується лише розробкою веб-сторінок. Вона знайшла широке застосування у різних областях програмування, де вимагається гнучкість та динамічність. Можливість працювати з об'єктами, функціями першого класу та підтримка різних стилів програмування роблять JavaScript потужним і універсальним інструментом для вирішення різноманітних завдань у програмуванні. [4].

2.2 Вибір бази даних

Для збереження даних мобільного додатку потрібно використати базу даних. Було обрано PostgreSQL, тому що PostgreSQL надійний, безпечний і розширюваний, а також має багату екосистему доступних інструментів,

розробники використовують PostgreSQL для різних випадків використання. Програмне забезпечення розроблено таким чином, щоб бути сумісним з усіма основними операційними системами, включаючи Linux, Windows і Macintosh, і воно підтримує текст, зображення, звуки та відео, що робить його популярною базою даних для людей і компаній з різними потребами. PostgreSQL широко вважається улюбленою технологією баз даних розробників, поступаючись лише MySQL.

PostgreSQL пропонує безліч варіантів для користувачів. Наприклад, можна вибрати такі функції, як відновлення на певний момент часу, ведення журналів із попереднім записом, деталізований контроль доступу, табличні простори, вкладені транзакції, онлайн-резервне копіювання та багатоверсійний контроль паралельності.

Надзвичайна масштабованість є відмінною рисою PostgreSQL. Програмне забезпечення може легко керувати величезними обсягами даних. Масштабованість PostgreSQL стосується не лише обсягу даних, якими він може керувати, але й кількості одночасно працюючих користувачів, якими він може керувати. [16]

2.3 Вибір інструменту для серверної частини

Для серверної частини віддано перевагу NodeJS. NodeJS – це кросплатформне середовище виконання з відкритим вихідним кодом, яке дозволяє розробникам створювати всілякі серверні інструменти та програми на JavaScript. Середовище виконання призначене для використання поза контекстом браузера. Таким чином, середовище пропускає специфічні API JavaScript для браузера та додає підтримку більш традиційних API ОС, включаючи HTTP і бібліотеки файлової системи.

Node має ряд переваг:

- Чудова продуктивність. Node був розроблений для оптимізації пропускної здатності та масштабованості додатків.

- Код на мові JavaScript як на стороні клієнта так і на стороні сервера.
- Менеджер пакетів вузлів (npm) надає доступ до сотень тисяч багаторазових пакетів. Він також має найкраще у своєму класі вирішення залежностей і також може використовуватися для автоматизації більшості інструментів збірки.
- Node.js є портативним. Він доступний у Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS і NonStop OS. Крім того, він добре підтримується багатьма провайдерами веб-хостингу, які часто надають спеціальну інфраструктуру та документацію для розміщення сайтів Node.

А також використано фреймворк ExpressJS для серверної частини. Express — це найпопулярніший фреймворк Node. Він забезпечує механізми для:

- Записує обробники для запитів з різними дієсловами HTTP на різних URL-шляхах (маршрутах).
- Інтеграція з механізмами візуалізації "view", щоб генерувати відповіді, вставляючи дані в шаблони.
- Встановлює загальні параметри веб-додатків, як-от порт для підключення та розташування шаблонів, які використовуються для відтворення відповіді.
- Додає додаткове «middleware» обробки запитів у будь-якій точці конвеєра обробки запитів.

Хоча Express сам по собі досить мінімалістичний, розробники створили сумісні пакети проміжного програмного забезпечення для вирішення майже будь-якої проблеми розробки. Існують бібліотеки для роботи з файлами cookie, сеансами, логінами користувачів, параметрами URL, даними POST, заголовками безпеки та багатьма іншими. [2]

2.4 Вибір інструменту для клієнтської частини

JavaScript є однією з найпопулярніших мов програмування, які використовують для розробки мобільних додатків. У результаті розвитку фреймворку React Native, ця мова стала основою для створення мобільних додатків. Фреймворк React Native використовує найкращі аспекти нативної розробки та поєднує їх з перевагами бібліотеки JavaScript – React, яка вважається передовою у своєму класі інструментів для створення інтерфейсів користувача. Одна з важливих переваг React Native полягає у тому, що він дає змогу розробникам використовувати спільну технологію – React, для підтримки двох основних мобільних платформ. Завдяки цьому, розробники можуть створювати мобільні додатки для Android та iOS, використовуючи одну кодову базу.

React Native надає базовий набір компонентів, які не залежать від конкретної платформи, таких як "View", "Text" та "Image". Ці компоненти безпосередньо відображаються на основних блоках інтерфейсу користувача, специфічних для кожної платформи. Крім того, компоненти React Native взаємодіють з нативними API через декларативну парадигму інтерфейсу React та JavaScript. Це дозволяє розробникам створювати мобільні додатки, які засновані на нативному коді і зберігають високу продуктивність та якість взаємодії з користувачем.

Фреймворк React Native розширює можливості розробників, надаючи їм зручний спосіб створення нативних мобільних додатків. Він дозволяє новим командам розробників швидко освоїти процес розробки мобільних додатків і забезпечує швидкість та ефективність роботи для вже наявних команд розробників, які вже мають досвід роботи з нативними технологіями. Загалом, React Native є потужним інструментом для розробки мобільних додатків, який поєднує переваги нативної розробки з перевагами JavaScript та фреймворку React. Він дозволяє розробникам створювати високоякісні мобільні додатки, які забезпечують зручний і приємний досвід користувачів. [9]

3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Реалізація бази даних

У ході розробки було створено діаграму бази даних (рис.3.1).

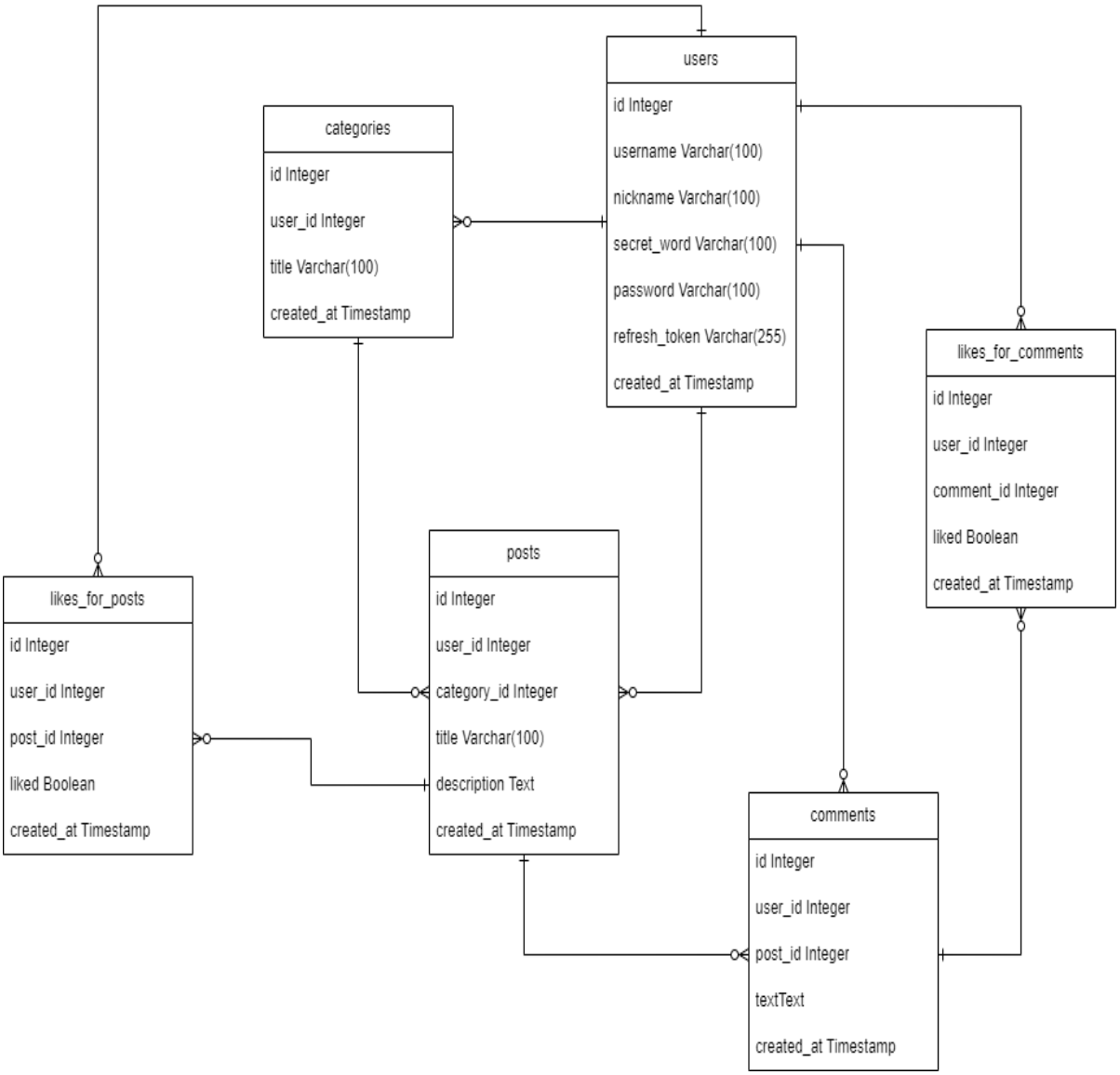

Рисунок 3.1 – Діаграма бази даних

Діаграма бази даних є важливим інструментом в розробці програмного забезпечення, який допомагає візуалізувати структуру та взаємозв'язки між таблицями, сутностями та атрибутами в базі даних. Вона представляє собою

графічне зображення, що ілюструє схему бази даних і відображає логічну архітектуру та організацію даних.

Оптимальна діаграма бази даних відображає структуру, яка повністю задовольняє поставлені вимоги та враховує потреби додатку або системи, для якої вона розробляється. Вона ретельно спроектована з урахуванням нормалізації даних, що забезпечує ефективне зберігання, маніпулювання та отримання інформації з бази даних. Така структура дозволяє забезпечити цілісність даних та уникнути зайвої дублювання інформації.

Гарна діаграма бази даних також враховує взаємозв'язки між таблицями та сутностями, використовуючи зв'язки такі як один до одного, один до багатьох або багато до багатьох. Це дозволяє встановлювати залежності між даними і забезпечує їх правильне збереження та доступ до них. Крім того, діаграма може включати в себе індекси та обмеження, які допомагають покращити продуктивність та надійність бази даних.

Така оптимальна та гарно структурована діаграма бази даних є незамінним інструментом для розробників і аналітиків даних. Вона допомагає зрозуміти логіку та взаємозв'язки між даними, спрощує розробку та підтримку системи, а також сприяє ефективному аналізу та використанню даних.

3.1.1 Опис таблиць

Таблиця "users" створена для збереження даних користувачів. Це головна таблиця в цій базі даних. Вона містить такі поля як:

- "id" – унікальне число, генерується автоматично
- "username" – обов’язкове поле, повинно містити унікальне слово, яке ввів користувач. Використовується для того щоб входити в систему.
- "nickname" – поле зроблене для того щоб користувач міг написати бажаний псевдонім і не використовувати "username"
- "secret_word" - обов’язкове поле, вводиться користувачем, використовується у подальшому для зміни паролю.

- "password" - обов’язкове поле, вводиться користувачем, використовується для входу в систему, хешується на сервері.
- "refresh_token" – значення генерується сервером, використовується для оновлення "access_token", який не зберігається в бд.
- "created_at" – дата та час створення сутності.

Таблиця "categories" створена для збереження категорій. Категорії потрібні щоб можна було організувати пости схожі за тематикою. Таблиця містить такі поля як:

- "id" – унікальне число, генерується автоматично
- "user_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "users".
- "title" – обов’язкове поле, назва категорії.
- "created_at" – дата та час створення сутності.

Таблиця "posts" створена для збереження даних постів. Пости створюються користувачами. Кожний пост пов’язаний з категорією. Таблиця містить такі поля як:

- "id" – унікальне число, генерується автоматично
- "user_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "users".
- "category_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "categories".
- "title" – обов’язкове поле, заголовок посту.
- "description" – обов’язкове поле, основний текст посту.
- "created_at" – дата та час створення сутності.

Таблиця "comments" створена для збереження коментарів до постів. До коментарів, як і до постів можна ставити лайки. Таблиця містить такі поля як:

- "id" – унікальне число, генерується автоматично
- "user_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "users".

- "post_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "posts".
- "text" – обов’язкове поле, текст коментарю.
- "created_at" – дата та час створення сутності.

Таблиця "likes_for_posts" створена для збереження лайків, які відносяться до постів. Таблиця містить такі поля як:

- "id" – унікальне число, генерується автоматично
- "user_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "users".
- "post_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "posts".
- "liked" – поле яке показує що поставив корисатувач, лайк або дізлайк.
- "created_at" – дата та час створення сутності.

Таблиця "likes_for_comments" створена для збереження лайків, які відносяться до коментарів. Таблиця містить такі поля як:

- "id" – унікальне число, генерується автоматично
- "user_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "users".
- "comment_id" – обов’язкове поле, зовнішній ключ який посилається на таблицю "comments".
- "liked" – поле яке показує що поставив корисатувач, лайк або дізлайк.
- "created_at" – дата та час створення сутності.

Було зроблено дві окремі таблиці для лайків - "likes_for_comments" та "likes_for_posts". Таке рішення зробило логіку на сервері більш логічною, а дані які зберігаються в самій базі даних стали кращими для сприйняття.

3.1.2 Опис типів даних

Нижче описані типи даних які були використані в базі даних (табл.3.1).

Таблиця 3.1 – Опис типів даних

Назва	**Опис**
Integer	Використовується для зберігання цілих чисел без десяткової частини. Він відповідає 32-бітному знаковому цілому числу і може приймати значення в діапазоні від -2147483648 до 2147483647.
Varchar	Використовується для зберігання рядків змінної довжини. "Varchar" може містити рядки з будь-яким набором символів, включаючи літери, цифри та спеціальні символи.
Boolean	Використовується для зберігання значень логічного типу, яке може приймати дві можливі вартості: "TRUE" (істинне) або "FALSE" (хибне).
Timestamp	Використовується для зберігання дати та часу з точністю до мілісекунди. Він представляє собою комбінацію дати та часу і має формат "YYYY-MM-DD HH:MI:SS".
Text	Використовується для зберігання великого обсягу текстових даних. Він може містити будь-який текст, включаючи лапки, спеціальні символи та роздільники рядків.

3.2 Реалізація серверної частини

На серверній частині свого проекту, було використано Node.js, Express та Typescript. Ці технології мають свої унікальні переваги та прекрасно підходять для розробки мобільного додатку, за допомогою якого можна накопичувати, узагальнювати та розповсюджувати корисні поради.

Node.js є платформою, побудованою на базі JavaScript, яка забезпечує ефективне виконання коду на стороні сервера. Вона відмінно підходить для асинхронного програмування, що є важливим аспектом розробки веб-додатків

з великою кількістю взаємодій. Завдяки широкій підтримці та активній спільноті, Node.js забезпечує багато розширень і модулів, що спрощують розробку та розширення функціональності додатку.

Express є легковаговим фреймворком для побудови веб-додатків на Node.js. Він надає простий та інтуїтивно зрозумілий інтерфейс для розробки серверних додатків. Express пропонує широкий набір функціональності, такий як маршрутизація, обробка запитів, підтримка шаблонів та багато іншого. Його легкість використання дозволяє прискорити процес розробки та підтримку додатку.

Typescript є мовою програмування, яка надає типізацію та розширену функціональність над JavaScript. Використання Typescript дозволяє підвищити надійність та легкість обслуговування коду, забезпечуючи перевірку типів на етапі розробки. Це особливо корисно при роботі з великими проектами, де точність типів та чіткість коду є важливими факторами. Крім того, Typescript забезпечує багато інструментів для поліпшення продуктивності розробників та зниження ризиків виникнення помилок.

Обрані технології мають кілька переваг для мого проекту. Використання Node.js дозволяє розробити потужний та масштабований сервер, що може ефективно обробляти багатокористувацькі запити. Express надає простоту у встановленні та роботі з серверними маршрутами, обробкою запитів та відправленням відповідей. Typescript допомагає збільшити надійність та стабільність проекту, забезпечуючи перевірку типів та більшу чіткість коду.

Комбінація Node.js, Express та Typescript дозволить розробити мобільний додаток на високому рівні якості з ефективним серверним API, надійною обробкою даних та зручними інструментами для розробників. Ці технології мають широку підтримку, активну спільноту розробників та безліч ресурсів для навчання та підтримки.

3.2.1 Огляд архітектури серверної частини

Для цього проекту було використано архітектуру з контролерами та роутами (рис. 3.2).

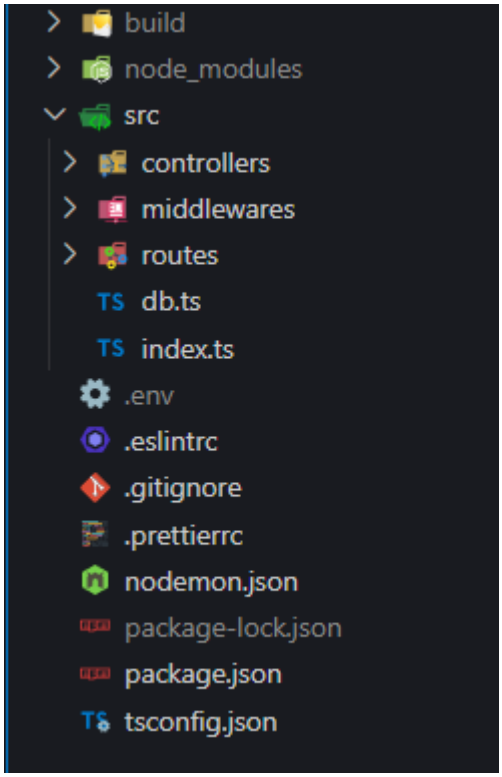

Рисунок 3.2 – Архітектура коду серверу

Використання архітектури з контролерами і роутами є оптимальним варіантом для реалізації серверної частини даного проекту. Ця архітектура базується на принципах модульності, розподіленості обов'язків та забезпеченні легкості обслуговування коду.

Контролери виконують роль посередників між маршрутами і бізнес-логікою додатку. Вони приймають HTTP-запити, виконують необхідні перевірки, обробляють дані та ініціюють відповідні дії або запити до сервісів. Це дозволяє відокремити логіку маршрутизації від основних функцій додатку, спрощує розподіл обов'язків та забезпечує більшу модульність.

Роути визначають шляхи доступу до різних функціональних блоків додатку. Вони визначають маршрутизацію запитів до відповідних контролерів, що дозволяє забезпечити логічну структуру додатку та легке

розширення функціональності. Роути також можуть мати параметри, що дозволяє передавати додаткові дані або ідентифікатори, необхідні для обробки запитів.

Переваги такої архітектури для цього проекту очевидні. Вона забезпечує логічне розділення функціональності, дозволяє легко масштабувати та розширювати додаток. За допомогою контролерів і роутів можна ефективно обробляти різні типи запитів, виконувати перевірку даних, аутентифікацію та авторизацію. Крім того, ця архітектура сприяє підтримці розподіленої командної розробки, де розробники можуть працювати над окремими модулями без взаємних перешкод.

Загалом, використання архітектури з контролерами і роутами допомагає побудувати структурований, гнучкий та легко розширюваний серверний шар для мобільного додатку. Вона забезпечує чітку організацію коду, спрощує роботу з HTTP-запитами та підвищує зручність розробки та обслуговування проекту.

3.2.2 Опис контролерів та роутів

1) "Auth"

Контролер **"auth"** виконує роль обробника запитів, пов'язаних з аутентифікацією користувачів. Цей контролер містить набір функцій, які забезпечують реєстрацію, вхід, скидання пароля, оновлення токенів та отримання профілю користувача. Кожна функція виконує конкретний функціонал і відповідає на відповідні HTTP-запити.

Функція **"register"** відповідає за обробку POST-запиту на шлях **"/register"**. Вона перевіряє наявність користувача з вказаним іменем у базі даних. Якщо користувач існує, повертається помилка. У протилежному випадку, пароль та секретне слово хешуються і зберігаються у базі даних, а також генеруються та повертаються токени доступу і оновлення. Інформація про користувача разом з токеном доступу відправляється у відповідь.

Функція **"login"** відповідає за обробку POST-запиту на шлях **"/login"**. Вона перевіряє наявність користувача з вказаним іменем у базі даних. Якщо користувач не існує або пароль невірний, повертається помилка. У протилежному випадку, генеруються токени доступу і оновлення, оновлюється оновлюючий токен користувача у базі даних, і інформація про користувача разом з токеном доступу відправляється у відповідь.

Функція **"profile"** відповідає за обробку GET-запиту на шлях **"/profile"**. Вона перевіряє коректність токену доступу, отримує інформацію про користувача з бази даних за його іменем, і повертає цю інформацію у відповідь.

Функція **"passwordReset"** відповідає за обробку POST-запиту на шлях **"/password-reset"**. Вона перевіряє наявність користувача з вказаним іменем у базі даних та правильність секретного слова. Якщо користувач не існує або секретне слово невірне, повертається помилка. У протилежному випадку, пароль хешується і оновлюється у базі даних, і повертається повідомлення про успішну зміну пароля.

Функція **"refreshToken"** відповідає за обробку POST-запиту на шлях **"/refresh-token"**. Вона перевіряє коректність оновлюючого токену, отримує інформацію про користувача з бази даних за його іменем, генерує нові токени доступу і оновлення, оновлює оновлюючий токен у базі даних, і повертає нові токени у відповідь.

Роутер до контролера **"auth"** встановлює відповідні маршрути для кожної функції контролера. Усі маршрути починаються з **"/auth"**. Наприклад, **"POST /auth/register"** викликає функцію **"register"** контролера **"auth"**. Таким чином, роутер встановлює зв'язок між URL-шляхами та функціями контролера, що дозволяє обробляти відповідні HTTP-запити.

2) "Category"

Контролер **"category"** виконує роль обробника запитів, пов'язаних з категоріями. Він містить набір функцій, які забезпечують створення,

отримання, оновлення та видалення категорій. Кожна функція виконує конкретний функціонал і обробляє відповідні HTTP-запити.

Функція **"createCategory"** відповідає за обробку POST-запиту на шлях **"/"**. Вона отримує назву категорії з тіла запиту і перевіряє, чи вже існує категорія з такою назвою у базі даних. Якщо така категорія вже існує, повертається помилка зі статусом 400. У протилежному випадку, отримується токен доступу з заголовка запиту, розшифровується для отримання ідентифікатора користувача, і створюється нова категорія у базі даних з вказаною назвою та ідентифікатором користувача. Нова категорія повертається у відповідь зі статусом 200.

Функція **"getCategories"** відповідає за обробку GET-запиту на шлях **"/"**. Вона отримує параметри limit (ліміт кількості категорій) та title (часткова назва категорії) з запиту і виконує відповідний запит до бази даних. Результати запиту містять список категорій разом з кількістю пов'язаних з ними постів. Категорії та загальна кількість повертаються у відповідь зі статусом 200.

Функція **"updateCategory"** відповідає за обробку PUT-запиту на шлях **"/:id"**, де **":id"** - ідентифікатор категорії. Вона отримує нову назву категорії з тіла запиту і перевіряє наявність категорії з вказаним ідентифікатором у базі даних. Якщо категорія не існує, повертається помилка зі статусом 400. У протилежному випадку, оновлюється назва категорії у базі даних і повертається оновлена категорія у відповідь зі статусом 200.

Функція **"deleteCategory"** відповідає за обробку DELETE-запиту на шлях **"/:id"**, де **":id"** - ідентифікатор категорії. Вона перевіряє наявність категорії з вказаним ідентифікатором у базі даних. Якщо категорія не існує, повертається помилка зі статусом 400. У протилежному випадку, видаляється категорія з бази даних і повертається повідомлення про успішне видалення у відповідь зі статусом 200.

Роутер до контролера **"category"** встановлює відповідні маршрути для кожної функції контролера. Усі маршрути починаються з **"/"**. Наприклад, **"POST /"** викликає функцію **"createCategory"** контролера **"category"**. Таким

чином, роутер встановлює зв'язок між URL-шляхами та функціями контролера, що дозволяє обробляти відповідні HTTP-запити

3) "Posts"

Контролер **"posts"** виконує обробку запитів, пов'язаних з постами. Він містить набір функцій, кожна з яких обробляє конкретну операцію пов'язану з постами, такі як створення, отримання, оновлення та видалення. Кожна функція виконує певні перевірки та операції з базою даних, і повертає відповідь зі статусом HTTP.

Функція **"createPost"** відповідає за обробку POST-запиту на шлях **"/"**. Вона отримує дані про заголовок, опис та ідентифікатор категорії з тіла запиту. Перевіряється, чи всі необхідні поля передані у запиті, і якщо хоча б одне з полів відсутнє, повертається помилка зі статусом 400. Далі, отримується токен доступу з заголовка запиту та розшифровується для отримання ідентифікатора користувача. Пост створюється у базі даних з отриманими даними та ідентифікатором користувача, і повертається створений пост у відповідь зі статусом 200.

Функція **"getPosts"** відповідає за обробку GET-запиту на шлях **"/"**. Вона отримує параметри **"limit"** (ліміт кількості постів), **"title"** (часткова назва поста), **"sort_variant"** (варіант сортування) та **"category_id"** (ідентифікатор категорії) з запиту. За допомогою отриманих параметрів формується SQL-запит до бази даних, який отримує всі пости, відповідні параметрам. Результати запиту повертаються у відповідь зі статусом 200.

Функція **"getPostById"** відповідає за обробку GET-запиту на шлях **"/:id"**, де **":id"** - ідентифікатор поста. Вона отримує ідентифікатор поста з URL-параметру та токен доступу з заголовка запиту. Перевіряється наявність поста з вказаним ідентифікатором у базі даних. Якщо пост не існує, повертається помилка зі статусом 400. У протилежному випадку, отримується інформація про пост з бази даних та повертається у відповідь зі статусом 200.

Функція **"updatePost"** відповідає за обробку PUT-запиту на шлях **"/:id"**, де **":id"** - ідентифікатор поста. Вона отримує дані про заголовок, опис та

ідентифікатор категорії з тіла запиту, а також ідентифікатор поста з URL-параметру. Перевіряється наявність поста з вказаним ідентифікатором у базі даних. Якщо пост не існує, повертається помилка зі статусом 400. У протилежному випадку, оновлюються дані поста у базі даних і повертається оновлений пост у відповідь зі статусом 200.

Функція **"deletePost"** відповідає за обробку DELETE-запиту на шлях **"/:id"**, де **":id"** - ідентифікатор поста. Вона перевіряє наявність поста з вказаним ідентифікатором у базі даних. Якщо пост не існує, повертається помилка зі статусом 400. У протилежному випадку, видаляється пост з бази даних і повертається повідомлення про успішне видалення у відповідь зі статусом 200.

Роутер до контролера **"posts"** встановлює відповідні маршрути для кожної функції контролера. Усі маршрути починаються з **"/"**. Наприклад, **"POST /"** викликає функцію **"createPost"** контролера **"posts"**. Таким чином, роутер встановлює зв'язок між URL-шляхами та функціями контролера, що дозволяє обробляти відповідні HTTP-запити.

4) "Comments"

Контролер **"comments"** виконує обробку запитів, пов'язаних з коментарями. Він містить набір функцій, кожна з яких відповідає за певну операцію з коментарями, такі як створення, отримання, оновлення та видалення коментарів. Цей контролер використовує модуль **"express"** для обробки HTTP-запитів та співпрацює з базою даних для збереження та отримання коментарів.

Функція **"createComment"** є обробником запиту типу POST і відповідає за створення нового коментаря. Вона отримує дані коментаря з тіла запиту, включаючи текст коментаря та ідентифікатор пов'язаного поста. Після цього, за допомогою автентифікаційного токену, отримується ідентифікатор користувача. Далі, за допомогою запиту до бази даних, новий коментар зберігається і повертається відповідь зі статусом 200, що містить створений коментар.

Функція **"getComments"** є обробником запиту типу GET і відповідає за отримання списку коментарів. Вона отримує параметри запиту, такі як ліміт кількості коментарів та варіант сортування. За допомогою автентифікаційного токену, отримується ідентифікатор користувача. Далі, виконується запит до бази даних, який повертає список коментарів з їх статистикою (кількість лайків, кількість дизлайків тощо). Результат повертається відповіддю зі статусом 200.

Функція **"getCommentById"** є обробником запиту типу GET і відповідає за отримання конкретного коментаря за його ідентифікатором. Вона отримує ідентифікатор коментаря з параметрів запиту. За допомогою автентифікаційного токену, отримується ідентифікатор користувача. Далі, виконується запит до бази даних, який повертає конкретний коментар зі статистикою (кількість лайків, кількість дизлайків тощо). Якщо коментар не знайдено, повертається відповідь зі статусом 400 та повідомленням про відсутність коментаря. В іншому випадку, коментар повертається відповіддю зі статусом 200.

Функція **"updateComment"** є обробником запиту типу PUT і відповідає за оновлення коментаря. Вона отримує новий текст коментаря з тіла запиту та ідентифікатор коментаря з параметрів запиту. Далі, виконується запит до бази даних, який оновлює коментар з новим текстом і повертає оновлений коментар відповіддю зі статусом 200. Якщо коментар не знайдено, повертається відповідь зі статусом 400 та повідомленням про відсутність коментаря.

Функція **"deleteComment"** є обробником запиту типу DELETE і відповідає за видалення коментаря. Вона отримує ідентифікатор коментаря з параметрів запиту. Далі, виконується запит до бази даних, який видаляє коментар за його ідентифікатором. Після видалення коментаря, повертається відповідь зі статусом 200 та повідомленням про успішне видалення коментаря. Якщо коментар не знайдено, повертається відповідь зі статусом 400 та повідомленням про відсутність коментаря.

Роутер **"comments"** визначає маршрутизацію для запитів, пов'язаних з коментарями. Він використовує модуль **"express"** і має п'ять основних маршрутів: **"POST /"** для створення коментаря, **"PUT /:id"** для оновлення коментаря, **"DELETE /:id"** для видалення коментаря, **"GET /"** для отримання списку коментарів та **"GET /:id"** для отримання конкретного коментаря за його ідентифікатором. Кожен маршрут викликає відповідний контролер для обробки запиту.

5) "PostLikes"

Контролер **"postLikes"** відповідає за операції, пов'язані з лайками до постів. Він містить чотири функції: **"createLike"**, **"getLikes"**, **"updateLike"** і **"deleteLike"**. Кожна з цих функцій обробляє відповідний запит і повертає результат у форматі JSON.

Функція **"createLike"** приймає запит типу POST і відповідає за створення нового лайку до поста. Вона отримує з тіла запиту параметри **"liked"** (булеве значення, чи сподобався пост) і **"post_id"** (ідентифікатор поста). Далі, виконується перевірка токену авторизації, отримання ідентифікатора користувача з токену і виконання запиту до бази даних, який додає новий запис в таблицю **"likes_for_posts"** з відповідними значеннями. Після успішного створення лайку, повертається відповідь зі статусом 200 та повідомленням про успішне створення лайку. У разі виникнення помилки, повертається відповідь зі статусом 500 та повідомленням про помилку.

Функція **"getLikes"** приймає запит типу GET і відповідає за отримання статистики лайків для конкретного поста. Вона отримує з тіла запиту параметр **"post_id"** (ідентифікатор поста). Далі, виконується перевірка токену авторизації, отримання ідентифікатора користувача з токену та виконання декількох запитів до бази даних для підрахунку загальної кількості лайків, загальної кількості дизлайків та перевірки, чи користувач вже поставив лайк до цього поста. Після цього, повертається відповідь зі статусом 200 і відповідними статистичними даними у форматі JSON. У разі виникнення

помилки, повертається відповідь зі статусом 500 та повідомленням про помилку.

Функція **"updateLike"** приймає запит типу PUT і відповідає за оновлення статусу лайку до поста. Вона отримує з тіла запиту параметри **"liked"** (нове значення лайку) і **"post_id"** (ідентифікатор поста). Далі, виконується перевірка токену авторизації, отримання ідентифікатора користувача з токену та перевірка наявності запису лайку користувача до цього поста в базі даних. Якщо запис не існує, повертається відповідь зі статусом 400 та повідомленням про помилку. У разі наявності запису, виконується запит до бази даних для оновлення значення лайку. Після успішного оновлення лайку, повертається відповідь зі статусом 200 та повідомленням про успішне оновлення лайку. У разі виникнення помилки, повертається відповідь зі статусом 500 та повідомленням про помилку.

Функція **"deleteLike"** приймає запит типу DELETE і відповідає за видалення лайку до поста. Вона отримує з тіла запиту параметр **"post_id"** (ідентифікатор поста). Далі, виконується перевірка токену авторизації, отримання ідентифікатора користувача з токену та перевірка наявності запису лайку користувача до цього поста в базі даних. Якщо запис не існує, повертається відповідь зі статусом 400 та повідомленням про помилку. У разі наявності запису, виконується запит до бази даних для видалення лайку. Після успішного видалення лайку, повертається відповідь зі статусом 200 та повідомленням про успішне видалення лайку. У разі виникнення помилки, повертається відповідь зі статусом 500 та повідомленням про помилку.

Контролер **"postLikes"** експортується з модуля і використовується в роутері. Роутер містить чотири маршрути: **"POST /"** для створення нового лайку, **"PUT /"** для оновлення лайку, **"DELETE /"** для видалення лайку і **"GET /"** для отримання статистики лайків. Кожен маршрут використовує відповідну функцію контролера для обробки запиту. Роутер експортується з модуля та може бути підключений до головного файлу додатка для обробки запитів, пов'язаних з лайками до постів.

6) "CommentLikes"

Функції такі самі як в **"PostLikes"** але тут вони пов’язані з таблицею лайків для коментарів.

Використання контролерів і роутерів дозволяє структурувати серверну частину додатку та відокремити логіку обробки запитів від маршрутизації. Це сприяє кращій організації коду, полегшує розширення функціоналу і забезпечує легкість управління маршрутами. Крім того, використання контролерів і роутерів сприяє збереженню принципів модульності та розділенню відповідальностей, що покращує читабельність, підтримку і тестування коду.

Лістинг коду див. Додаток А.

3.2.3 Утиліта для перевірки запитів на сервер

Для перевірки запитів на сервер я використав утиліту Postman [8] (рис.3.3).

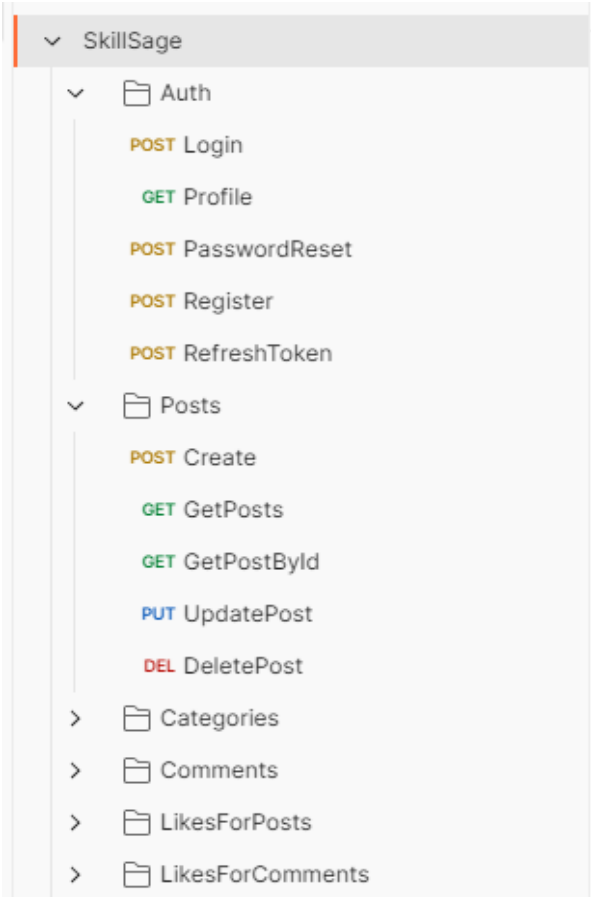

Рисунок 3. 3 – Архітектура коду серверу

Postman є популярним засобом для тестування та перевірки API. Він надає зручний та інтуїтивно зрозумілий інтерфейс, що дозволяє виконувати різні типи запитів до серверу і отримувати відповіді для перевірки правильності роботи серверної частини додатка.

Один з головних аргументів на користь використання Postman полягає в його універсальності. Він підтримує різні типи запитів, такі як GET, POST, PUT, DELETE, а також дозволяє передавати параметри запиту, заголовки, тіло запиту та отримувати відповіді сервера. Це дозволяє зручно тестувати різні функції серверу та перевіряти їх правильну реалізацію.

Крім того, Postman забезпечує можливість збереження та організації наборів тестів, що дозволяє створювати комплексні сценарії тестування та автоматизувати їх виконання. Це особливо корисно при розробці та тестуванні API з великою кількістю ендпоінтів та різноманітними параметрами.

Додатковою перевагою Postman є його здатність працювати як на різних операційних системах, так і у веб-браузері, що робить його доступним для розробників на будь-якій платформі.

У випадку використання Postman для перевірки запитів до серверу, розробник може швидко та зручно виконувати запити, передавати необхідні параметри, перевіряти статуси відповідей, перевіряти правильність обробки запитів та забезпечувати відповідну реакцію сервера на них.

Таким чином, використання Postman є раціональним вибором для перевірки запитів до серверу, оскільки він забезпечує зручність, універсальність та можливість організації тестів, що дозволяє ефективно перевіряти та валідувати функціональність серверного API.

3.3 Реалізація клієнтської частини

React Native для клієнтської частини мобільного додатку було обрано з кількох причин.

Перш за все, React Native базується на React, який є одним з найпопулярніших та потужних фреймворків для веб-розробки. Використання React Native дозволяє використовувати ті ж самі концепції та навички, що й при розробці веб-додатків, забезпечуючи більш простий перехід для веб-розробників до мобільного додатку.

Друга причина полягає у швидкості розробки. React Native дозволяє створювати мобільні додатки, використовуючи один код для обох платформ (Android та iOS). Це робить процес розробки більш ефективним та скорочує час, необхідний для випуску додатку на ринок.

Щодо структури проекту, я обрав структуру, яка сприяє легкості управління кодом та розширенню функціональності. Вона включає:

- components/: Директорія для зберігання реактивних компонентів додатку, таких як кнопки, поля вводу та інші.
- constants/: Директорія для зберігання статичних змінних.
- screens/: Директорія для зберігання екранів додатку. Кожен екран може мати свою власну папку та файли для кращої організації коду.
- navigation/: Директорія для зберігання файлів навігації, таких як конфігурація маршрутів та навігаційних компонентів.
- store/: Директорія для зберігання файлів управління станом за допомогою Redux.
- services/: Директорія для зберігання запитів на серверну частину.

Ця структура дозволяє групувати пов'язаний функціонал разом, полегшує розуміння коду та підтримку проекту.

Вибір React Native для розробки клієнтської частини мобільного додатку є обґрунтованим завдяки його швидкості розробки та доступності великої кількості розширень та бібліотек. Із застосуванням відповідної структури проекту та використанням відповідних бібліотек, розробка мобільного додатку на React Native стає ефективною та зручною для розробників.

3.3.1 Опис бібліотек в проекті

Основні бібліотеки в проекті:

1. React Navigation – це бібліотека для навігації в React Native додатках [12]. Вона надає розширені можливості для створення різних типів навігації, таких як стекова, табова, бічна панель тощо. Завдяки React Navigation, розробники можуть легко організувати та керувати рухом між екранами додатку. Основні переваги React Navigation:

- Простота використання: React Navigation має зрозумілий API та простий синтаксис, що дозволяє швидко налаштовувати навігацію в додатку.
- Гнучкість: Бібліотека надає широкі можливості налаштування навігації згідно з вимогами проекту. Вона підтримує різні типи анімацій переходів між екранами та дозволяє створювати власні компоненти навігації.
- Крос-платформеність: React Navigation підтримує як платформу Android, так і iOS, що дозволяє створювати переносні додатки для обох платформ з одним кодом.

2. Redux – це бібліотека для керування станом додатка у React та React Native [13]. Вона надає централізований зберігання стану додатка, що спрощує управління та оновлення даних у додатку. Основні переваги Redux:

- Однозначність стану: Redux сприяє встановленню однозначного та передбачуваного стану додатка.
- Спрощення управління станом: Redux дозволяє централізовано керувати станом додатка, що полегшує взаємодію між компонентами та забезпечує їхню консистентність.
- Зручність для великих додатків: Redux підходить для великих та складних додатків, де стан може бути розподіленим між багатьма компонентами. Він дозволяє ефективно керувати станом додатка незалежно від його розміру та складності.
- Розширюваність: Redux надає можливість розширення функціональності за допомогою middleware, які дозволяють перехоплювати та змінювати дії, що відбуваються у додатку.

- Тестування: Redux сприяє легкому тестуванню додатків, оскільки стан зберігається централізовано та може бути легко перевірений у тестах.

3. Axios – це бібліотека для виконання HTTP-запитів у браузері та на сервері [1]. Вона надає зручний та простий API для взаємодії з веб-серверами та отримання/відправлення даних. Основні переваги Axios:

- Простота використання: Axios має простий та зрозумілий API, що дозволяє легко виконувати HTTP-запити та обробляти отримані дані.
- Крос-платформеність: Бібліотека підтримує як браузерні середовища, так і Node.js, що дозволяє використовувати її як у клієнтській, так і серверній частині додатка.
- Підтримка інтерсепторів: Axios надає можливість встановлення інтерсепторів, що дозволяють перехоплювати та змінювати запити та відповіді перед їхнім відправленням або обробкою.
- Підтримка Promise: Axios використовує Promise для асинхронного отримання результатів запитів, що спрощує обробку асинхронних операцій та дозволяє уникнути callback-хеллу.

4. Lodash - це невелика, але потужна бібліотека JavaScript, яка надає функції високого рівня для роботи з масивами, об'єктами, рядками та іншими типами даних [5]. Основні переваги Lodash включають:

- Універсальність: Lodash підтримує різні середовища виконання, такі як браузер та Node.js, що дозволяє використовувати його як у клієнтських, так і серверних додатках.
- Зручний API: Бібліотека має добре організований API, який простий у використанні та допомагає зменшити кількість коду, необхідного для вирішення типових завдань.
- Функціональність: Lodash надає багато корисних функцій, таких як маніпуляція з масивами, робота з об'єктами та робота з рядками.
- Висока продуктивність: Lodash оптимізований для швидкого виконання та ефективного використання ресурсів, що дозволяє працювати з великими наборами даних без втрати продуктивності.

5. React-Native-Toast-Message – це бібліотека для React Native, яка дозволяє легко відображати сповіщення у мобільних додатках [10]. Основні переваги цієї бібліотеки включають:

- Простота використання: React-Native-Toast-Message надає простий та зрозумілий API для відображення та керування тостовими повідомленнями. Завдяки цьому, додавання сповіщень до додатка стає швидким та простим процесом.
- Налаштовуваність: Бібліотека дозволяє налаштовувати вигляд тостових повідомлень, такі як кольори, типи повідомлень, час відображення тощо. Це дозволяє адаптувати сповіщення до дизайну додатка.
- Підтримка анімацій: React-Native-Toast-Message підтримує анімації відображення та зникнення тостових повідомлень, що додає взаємодії та візуальну привабливість до сповіщень.
- Розширюваність: Бібліотека має гнучку архітектуру, яка дозволяє розширювати та додавати власний функціонал до тостових повідомлень, такий як додаткові кнопки або зображення.

6. NativeBase – це бібліотека компонентів для React Native, яка надає набір готових компонентів і стилів для швидкої розробки мобільних додатків [7]. Основні переваги NativeBase включають:

- Готові компоненти: Бібліотека містить велику кількість готових компонентів, таких як кнопки, списки, форми, заголовки тощо. Це дозволяє ефективно будувати інтерфейси додатків без необхідності використання власного коду для кожного елемента.
- Крос-платформеність: Компоненти NativeBase підтримуються на різних платформах, включаючи iOS та Android. Це дозволяє розробникам створювати мобільні додатки, які працюють на різних пристроях без зайвих зусиль.
- Теми: Бібліотека надає можливість легко налаштовувати вигляд компонентів шляхом зміни теми. За допомогою NativeBase можна швидко

змінювати кольори, шрифти, розміри тощо відповідно до дизайну вашого додатка.

- Розширюваність: NativeBase дозволяє розширювати та наслідувати компоненти, що дозволяє створювати власні компоненти на основі існуючих, що спрощує повторне використання коду та забезпечує консистентність в додатку.

7. React-Native-Vector-Icons – це бібліотека для React Native, яка надає набір векторних іконок для використання у мобільних додатках [11]. Основні переваги цієї бібліотеки включають:

- Багато наборів іконок: React-Native-Vector-Icons містить набір готових векторних іконок з різних наборів, таких як FontAwesome, Material Icons, Ionicons тощо. Це дозволяє використовувати широкий спектр іконок увашому додатку.
- Розширюваність та налаштовуваність: Бібліотека дозволяє легко налаштовувати вигляд іконок, змінюючи їхні кольори, розміри та стилі.
- Векторна графіка: Векторні іконки розширюються без втрати якості та чіткості навіть при зміні розміру. Це дозволяє легко змінювати розміри іконок у залежності від потреб вашого додатка, без необхідності використання різних наборів растрових зображень.
- Крос-платформеність: React-Native-Vector-Icons підтримується на різних платформах, включаючи iOS та Android. Це дозволяє використовувати однакові іконки в обох середовищах без необхідності використання різних ресурсів.

3.3.2 Опис модулів та екранів

1. Модуль авторизації

Модуль авторизації складається з трьох екранів: логін, реєстрація, відновлення паролю.

Екран логіну в мобільному додатку створений з метою забезпечення функціональності та зручності для користувача. Він складається з різних елементів, які допомагають здійснити вхід в обліковий запис (рис. 3.4).

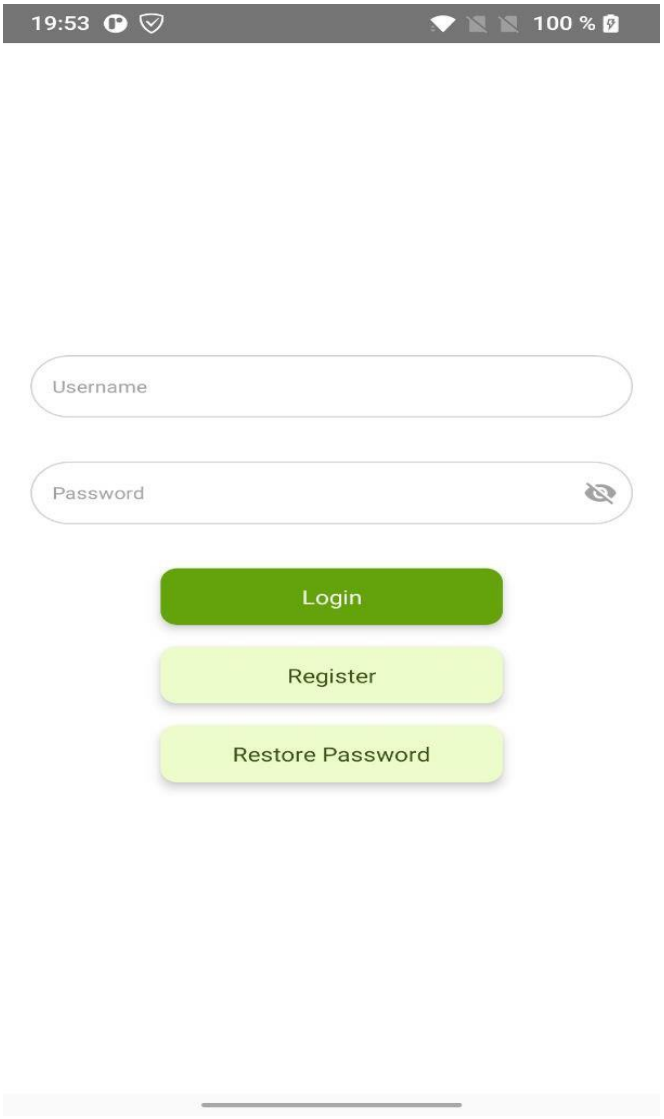

Рисунок 3.4 – Скріншот екрану логіну

У цьому екрані є два поля для введення даних: "Username" і "Password". Ці поля мають валідацію, яка перевіряє введені дані на коректність. У разі некоректного вводу відображаються повідомлення про помилку поруч з відповідним полем. Це допомагає користувачеві виправити помилки перед надсиланням даних.

Кнопка "Login" розташована після полів вводу і ініціює процес авторизації з введеними даними. Вона може бути візуально виділена, щоб привернути увагу користувача. Якщо користувач ще не має облікового запису, нижче розташована кнопка "Register", яка дозволяє перейти на екран реєстрації нового облікового запису. Це зручний спосіб для тих, хто бажає створити обліковий запис у додатку.

Також на екрані логіну є кнопка "Restore Password", яка надає можливість перейти на екран відновлення паролю. Це особливо корисно для користувачів, які забули свій пароль і потребують його відновлення.

При некоректних даних або невдалій спробі входу, на екрані можуть з'являтися повідомлення про помилку, які повідомляють користувача про необхідність виправити помилки або спробувати інший спосіб входу.

Крім того, були використані кольори, шрифти та розміщення елементів, щоб створити зручне і привабливе візуальне сприйняття. Це допоможе забезпечити легку навігацію та читабельність для користувача і покращить загальне враження від використання мобільного додатку.

Екран реєстрації у мобільному додатку призначений для створення нового облікового запису користувача. У верхній частині екрану розташовані три поля для введення даних: "Username", "Password" та "Secret Word" (рис.3.5).

Щоб забезпечити коректність введених даних, на кожному полі застосована валідація. Якщо користувач вводить неправильні дані, поруч з відповідним полем відображається повідомлення про помилку, яке допомагає користувачу виправити помилки перед надсиланням даних.

Після полів вводу розташована кнопка "Register", яка використовується для ініціювання процесу створення нового облікового запису з введеними даними. Ця кнопка візуально виділена, за допомогою кольору або стилю, щоб привернути увагу користувача.

Нижче знаходиться кнопка "Login", яка надає можливість перейти на екран входу в обліковий запис. Це зручний спосіб для користувачів, які вже мають обліковий запис і хочуть увійти в додаток.

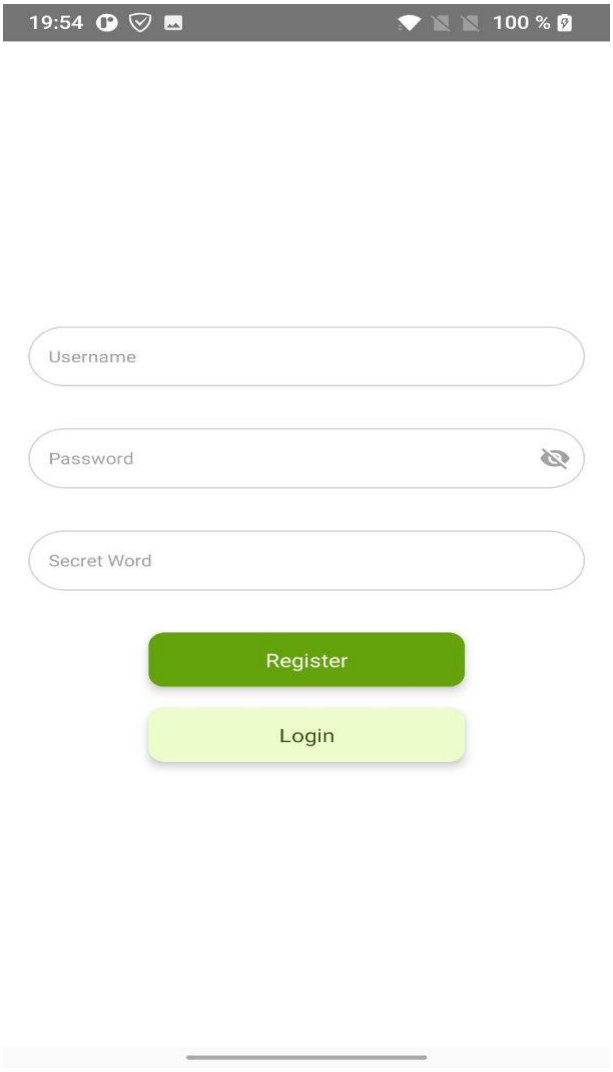

Рисунок 3.5 – Скріншот екрану реєстрації

Екран реєстрації також враховує можливість виникнення помилок або невдалої спроби реєстрації. В таких випадках на екрані можуть з'являтися повідомлення про помилку, які сповіщають користувача про необхідність виправити помилки або спробувати інший спосіб реєстрації.

Екран відновлення паролю у мобільному додатку призначений для користувачів, які забули свій пароль і потребують відновити доступ до свого облікового запису. На екрані розташовані поля для введення даних, які

ідентичні полям на екрані реєстрації. Користувач повинен ввести своє ім'я користувача та секретне слово, яке використовувалося при реєстрації облікового запису (рис. 3.6).

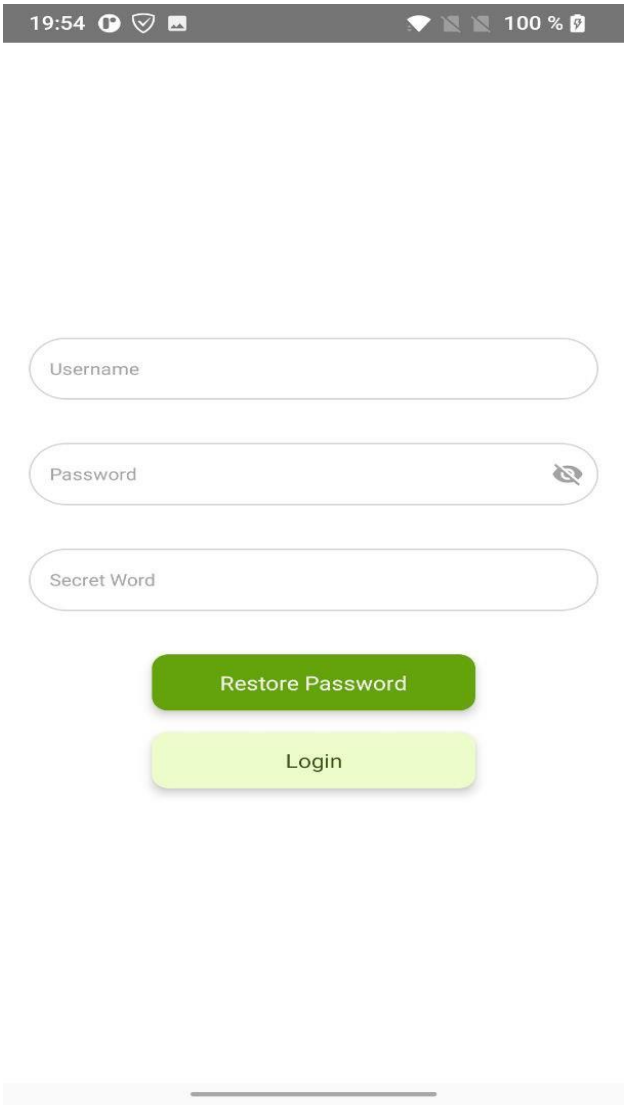

Рисунок 3.6 – Скріншот екрану відновлення паролю

Після введення відповідних даних користувач натискає кнопку "Restore Password". Ця кнопка запускає процес відновлення паролю, шляхом відправлення запиту на сервер для зміни паролю. Це дозволяє користувачеві змінити старий пароль.

Нижче знаходиться кнопка "Login", яка надає можливість користувачам перейти на екран логіну. Ця кнопка дозволяє користувачам увійти в свій обліковий запис після відновлення паролю.

На екрані також є валідація полів вводу, щоб забезпечити коректність введених даних. У разі некоректного вводу або помилок користувачу будуть відображені повідомлення про помилки поруч з відповідними полями вводу.

2. Модуль даних користувача

Екран профілю в мобільному додатку надає користувачам можливість переглядати та змінювати особисту інформацію, таку як ім'я користувача та псевдонім (рис 3.7).

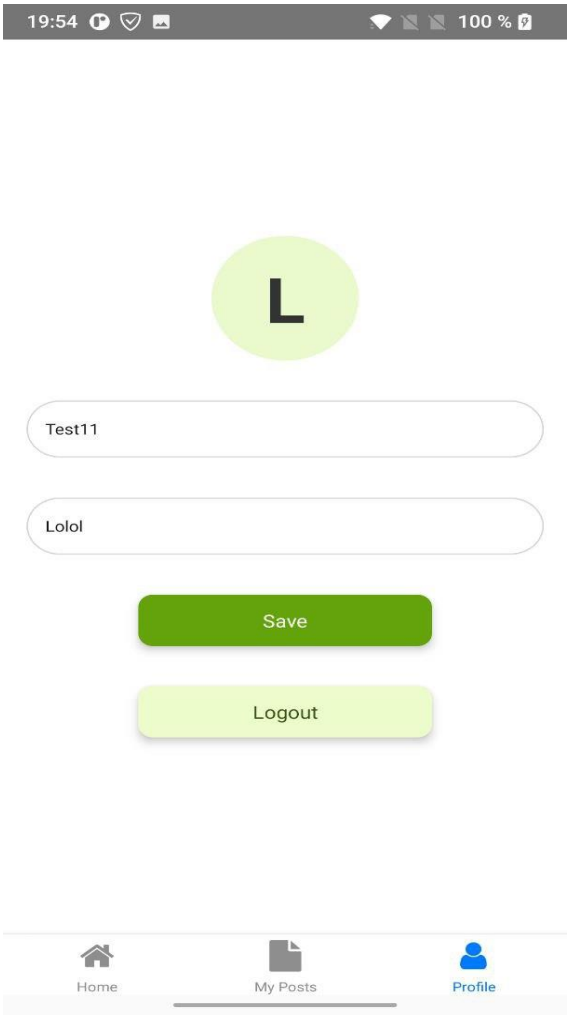

Рисунок 3.7 – Скріншот екрану профілю

У верхній частині екрана, над полями для вводу, розташований круглий елемент з першою літерою ім'я користувача. Це додає персоналізований акцент до профілю користувача.

Нижче знаходяться поля для введення інформації, зокрема поле для зміни ім'я користувача та поле для зміни псевдоніма (nickname). Користувач може вводити нові дані у ці поля залежно від своїх потреб.

Ще нижче розташована кнопка "Save", яка використовується для збереження змінених даних. Коли користувач натискає на цю кнопку, додаток зберігає нові дані, введені в поля для зміни, і оновлює профіль користувача з новою інформацією.

Також у нижній частині екрана розташована кнопка "Logout", яка надає можливість користувачеві розлогуватися з облікового запису. Після натискання на цю кнопку користувач повертається до екрану авторизації. Є валідація полів вводу, щоб переконатися, що користувач вводить коректні дані. При несумісних даних або помилках можуть з'являтися відповідні повідомлення поруч з відповідними полями вводу.

3. Модуль постів користувача

Екран з постами користувача у мобільному додатку надає можливість переглядати та шукати пости за їхнім заголовком. У верхній частині екрана розташоване поле для пошуку. Користувач може вводити заголовок поста у це поле для здійснення пошуку. Це поле дозволяє знайти пости, які містять введений користувачем текст у своєму заголовку (рис. 3.8).

Нижче розташований список постів користувача. Кожен пост відображається у списку разом зі своїм заголовком. Користувач може прокручувати список, щоб переглянути всі пости.

Коли користувач натискає на певний пост, він переходить до екрану з деталями поста. Цей екран відображає повну інформацію про пост, включаючи його заголовок, вміст та інші деталі. На цьому екрані користувач може подивитися коментарі написані іншими користувачами або написати свій.

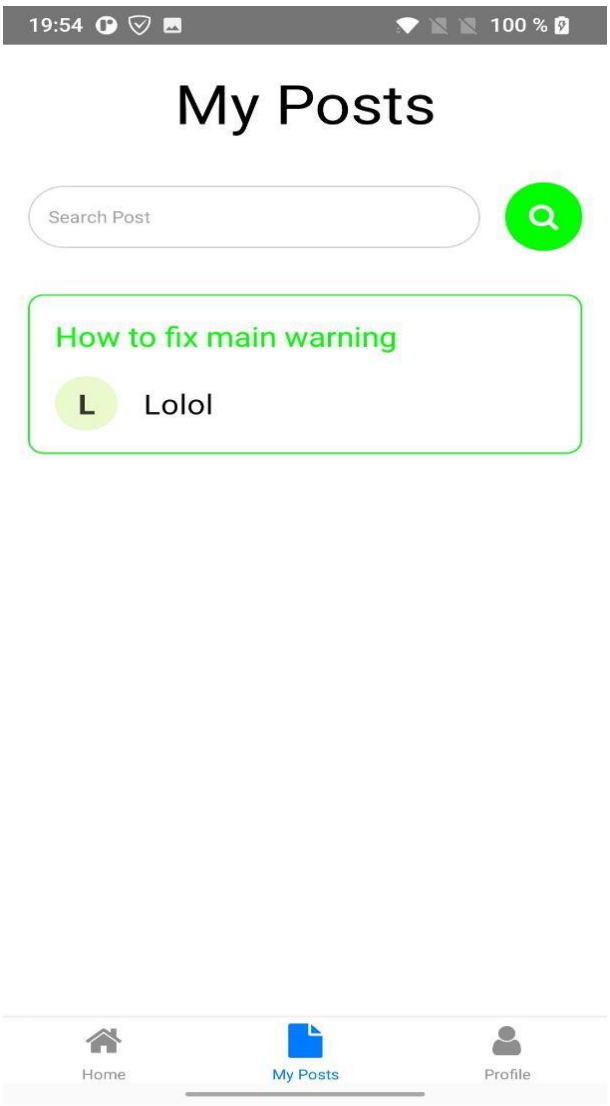

Рисунок 3.8 – Скріншот екрану постів користувача

4. Модуль основної частини

Екран з категоріями у мобільному додатку надає можливість користувачеві переглядати та шукати категорії за їхнім заголовком. Верхня частина екрана містить поле для пошуку. Користувач може ввести заголовок категорії у це поле для здійснення пошуку. Це поле дозволяє знайти категорії, які містять введений користувачем текст у своєму заголовку (рис. 3.9).

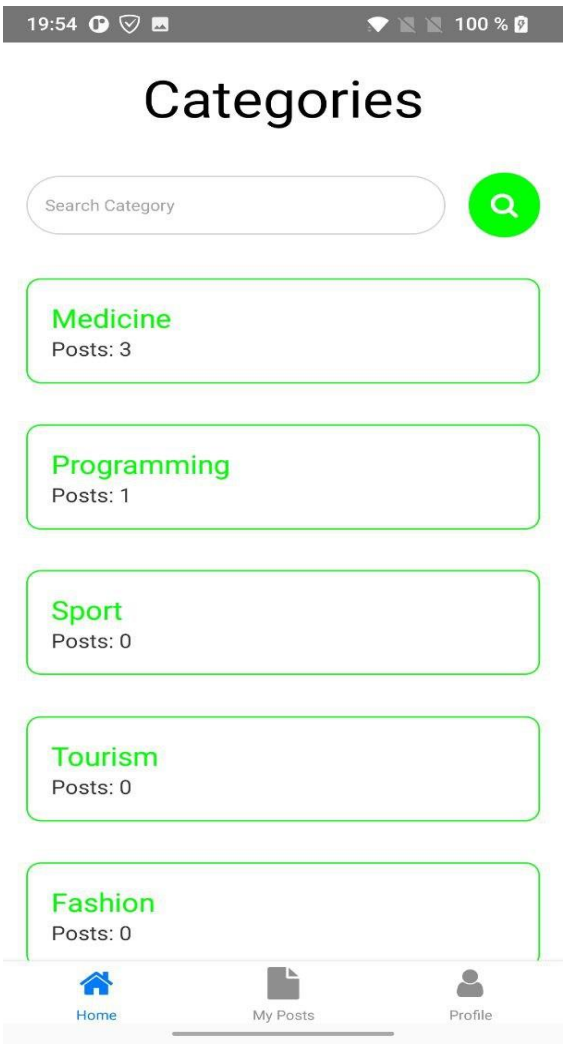

Рисунок 3.9 – Скріншот екрану категорій

Нижче розташований список категорій. Кожна категорія відображається у списку разом зі своїм заголовком та кількістю постів, що належать до цієї категорії. На екрані з категоріями також була реалізована пагінація, яка дозволяє користувачеві оптимізувати використання пам’яті мобільного телефону. Також це забезпечує зручну навігацію і дозволяє користувачеві переглядати більшу кількість категорій.

При натисканні на певну категорію, користувач переходить до екрану з постами цієї категорії. На цьому екрані відображаються пости, які належать до обраної категорії. Користувач може переглядати ці пости та додавати коментарі.

Екран з виводом постів у мобільному додатку забезпечує користувачам зручний спосіб перегляду постів та шукати потрібні за їхнім заголовком (рис.3.10).

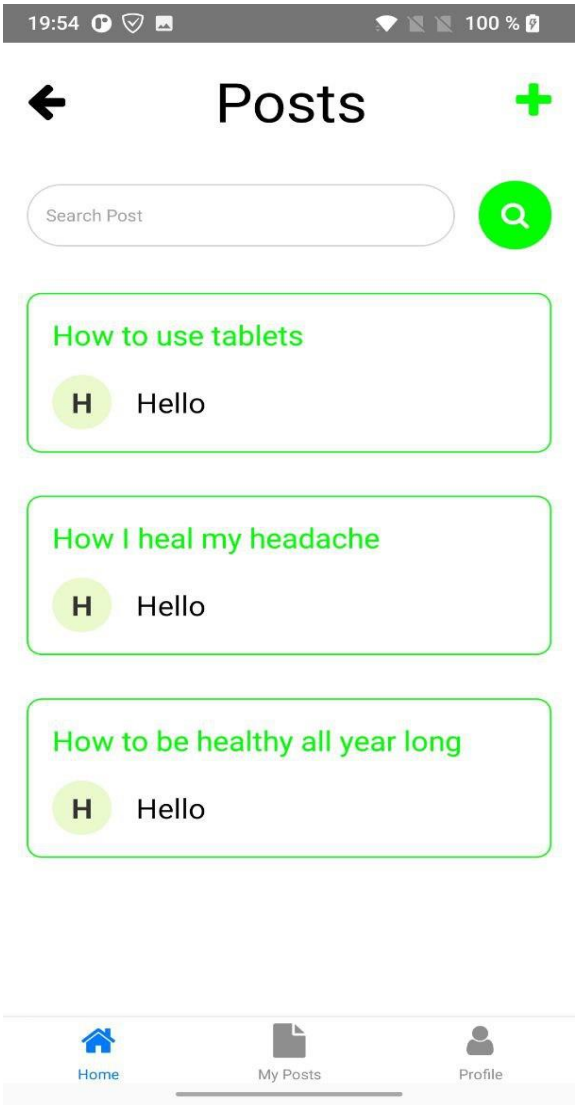

Рисунок 3.10 – Скріншот екрану постів

У верхній частині екрана розміщується поле для пошуку. Користувач може ввести заголовок посту у це поле для здійснення пошуку. Це поле дозволяє знайти пости, які містять введений користувачем текст у своєму заголовку.

Під полем для пошуку розміщений список постів. Кожен пост відображається у списку разом зі своїм заголовком та ім'ям автора. Ця

інформація дозволяє користувачеві швидко переглянути список постів та знайти потрібні йому матеріали.

На екрані з виводом постів також присутня кнопка "+" або "Створити пост", яка надає можливість користувачеві створити новий пост. При натисканні на цю кнопку, користувач переходить до екрану створення посту, де він може заповнити необхідну інформацію та опублікувати новий пост.

Екран додавання посту у мобільному додатку надає можливість користувачеві створити новий пост шляхом введення необхідних даних. На екрані додавання посту користувач може ввести два поля - "Title" (заголовок) та "Description" (опис). Введені дані дозволяють користувачеві створити пост зі зазначеними заголовком та описом (рис. 3.11).

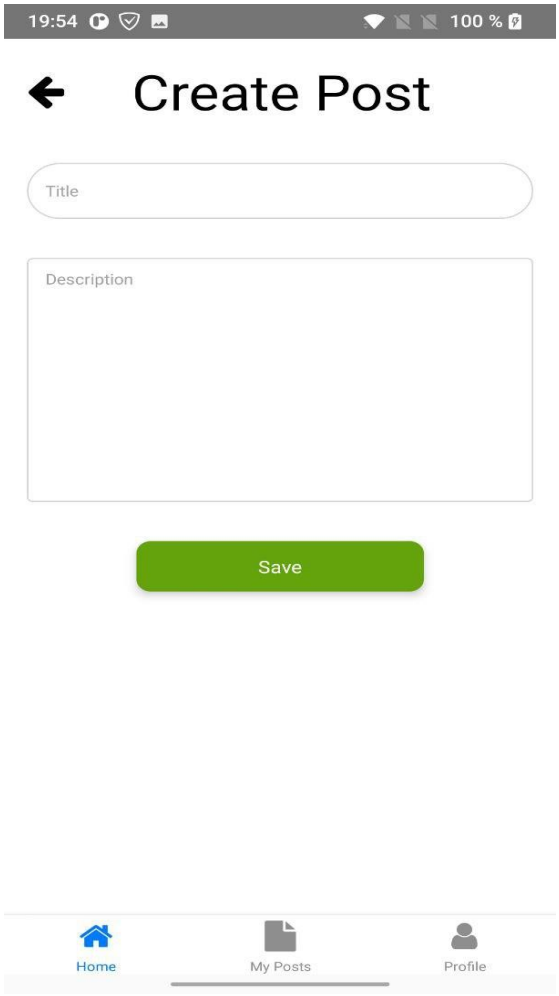

Рисунок 3.11 – Скріншот екрану додавання постів

У полі "Title" користувач вводить заголовок посту. Це поле має валідацію, щоб перевірити, чи введено коректні дані. Валідація допомагає уникнути створення постів з недостатньою або некоректною інформацією. У полі "Description" користувач вводить опис посту. Це поле також має валідацію для перевірки введених даних. Валідація допомагає забезпечити введення достатньої та коректної інформації у поле опису.

На екрані додавання посту присутня кнопка "Save" або "Зберегти", яка дозволяє користувачеві зберегти новий пост. При натисканні на цю кнопку система проводить валідацію введених даних. Якщо дані валідні, пост зберігається у системі, а користувач отримує повідомлення про успішне збереження посту. У випадку, якщо виникає помилка під час збереження, користувач отримує повідомлення про помилку, що допомагає розібратися з проблемою або повторити спробу пізніше.

Екран деталей посту у мобільному додатку призначений для виведення всієї докладної інформації про певний пост. Він має за мету забезпечити користувачам повний контекст та можливості для взаємодії з постом та коментарями (рис. 3.12).

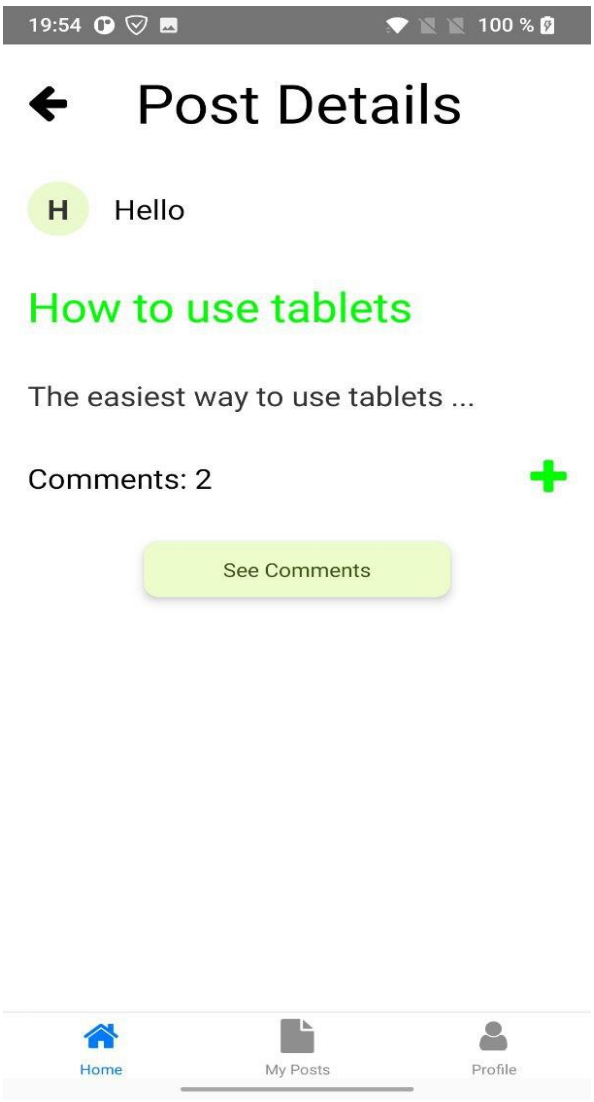

Рисунок 3.12 – Скріншот екрану деталей посту

При відкритті екрану деталей посту перше, що видно, це ім'я автора посту. Це допомагає користувачам встановити авторство та походження даного посту.

Наступним елементом є заголовок посту, який відображається вище контенту. Цей заголовок допомагає користувачам швидко зорієнтуватися в тематиці або змісті посту. Опис або контент посту відображається після заголовка. Це поле містить детальну інформацію, яку користувач хотів передати через пост.

Після контенту посту знаходиться інформація про кількість коментарів. Це число вказує на кількість коментарів, що були залишені до даного посту.

Воно дає користувачам уявлення про активність та обговорення, пов'язані з цим постом.

Нижче розміщені дві кнопки. Перша кнопка, позначена символом "+", дозволяє користувачеві перейти на екран створення нового коментаря до даного посту. При натисканні на цю кнопку відбувається перехід на відповідний екран, де користувач може ввести свій коментар. Друга кнопка є кнопкою переходу на екран коментарів до посту. При натисканні на неї відбувається перехід на сторінку зі списком коментарів, що були залишені до даного посту. Користувач може переглянути коментарі і, за бажанням, додати новий коментар або взаємодіяти з наявними коментарями.

Якщо пост, відображений на екрані деталей, належить самому користувачеві, то до розміщених кнопок може бути додана кнопка видалення посту. Ця кнопка надає можливість користувачеві видалити свій власний пост з додатку. Такий механізм надає користувачам контроль над своїми постами та можливість керувати ними за своїм розсудом.

Екран з виводом коментарів до посту надає користувачеві можливість переглянути коментарі, які були залишені до конкретного посту. На цьому екрані виводиться список коментарів з можливістю пагінації, щоб забезпечити зручну навігацію користувача (рис. 3.13).

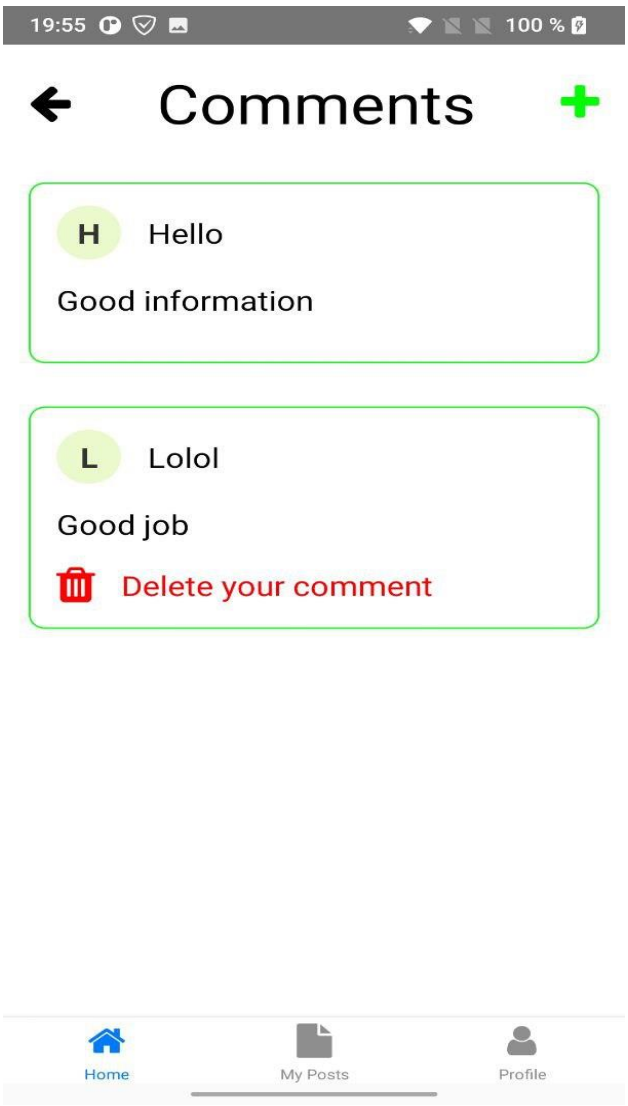

Рисунок 3.13 – Скріншот екрану коментарів до посту

Кожен коментар містить такі дані:

- Ім'я автора: відображається ім'я користувача, який залишив коментар.
- Текст коментаря: виводиться текст, що був написаний у коментарі.
- Кнопка "Видалити": якщо коментар належить користувачеві, який в даний момент користується додатком, то відображається кнопка "Видалити". Ця кнопка надає можливість користувачу видалити свій власний коментар з посту.

Зверху екрану розміщена кнопка "+", яка дозволяє користувачеві перейти на екран додавання нового коментаря. При натисканні на цю кнопку

користувач перенаправляється на відповідний екран, де він може ввести свій коментар та надіслати його.

Використання пагінації дозволяє показувати обмежену кількість коментарів на одному екрані щоб більш оптимізовано використовувати пам’ять телефону.

Екран створення коментаря надає користувачеві можливість написати свій коментар до посту. На цьому екрані присутнє поле вводу "Description", де користувач може ввести текст свого коментаря (рис. 3.14).

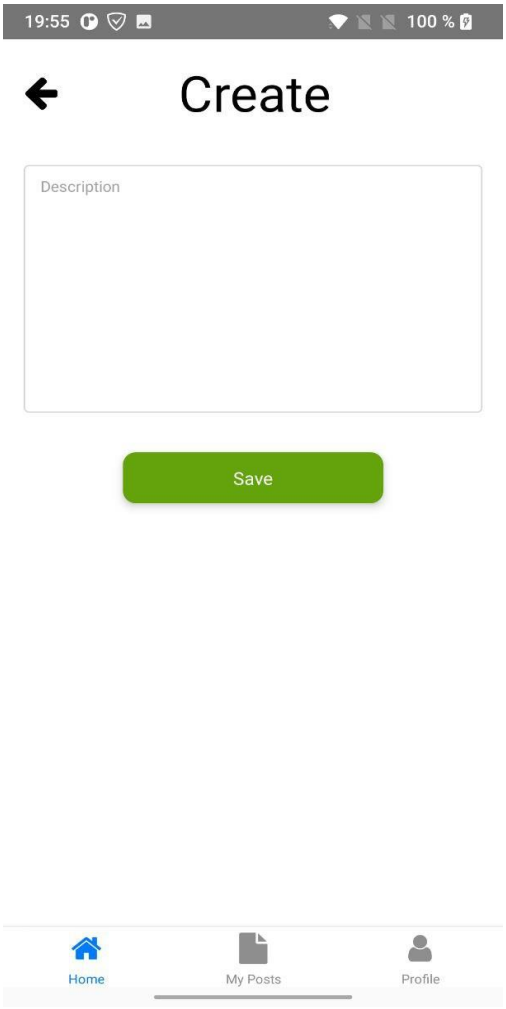

Рисунок 3.14 – Скріншот екрану коментарів до посту

Валідація поля "Description" забезпечує перевірку введених даних на валідність. Після введення тексту коментаря, користувач може натиснути

кнопку "Зберегти", щоб зберегти свій коментар. Після натискання кнопки виконується перевірка на валідність введених даних.

Якщо дані введені правильно, коментар зберігається успішно, і користувач отримує повідомлення про успішне створення коментаря. У випадку помилки, наприклад, якщо поле "Description" не пройшло валідацію, користувач отримує повідомлення про помилку, що нагадує про необхідність виправити введені дані перед збереженням коментаря.

Такий екран створення коментаря дозволяє користувачеві легко висловлювати свої думки та взаємодіяти з постами, створюючи нові коментарі і долучаючись до дискусій.

Лістинг коду див. Додаток Б.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було зроблено наступне:

1. Інформаційний огляд по даній темі.

У результаті дослідження було з’ясовано які є принципи побудови мобільних додатків. Були виділені моменти, які краще покажуть користувачу комфортність використання даного додатку.

Були перераховані інструменти для розробки мобільних додатків. А також їх переваги та недоліки. Досить чітким було пояснення різниці між кросплатформною мовою програмування та нативною.

2. Вибір методів розв’язання поставленої задачі

Було описано які інструменти розробки було обрано для виконання даної роботи. Виділено їх можливості та описані переваги. Докладно було написано про базу даних, мову програмування, інструменти для серверної частини та клієнтської.

3. Програмна реалізація

У цій програмній реалізації були використані база даних, серверна частина та клієнтська частина. База даних використовується для зберігання даних про користувачів, пости, коментарі та лайки. Серверна частина реалізує різні функції, такі як створення, оновлення, отримання та видалення даних через відповідні ендпоінти API. Клієнтська частина, реалізована за допомогою React Native, взаємодіє з сервером для отримання та відображення даних для користувачів мобільного додатку.

У результаті програмної реалізації було створено функціональну систему, яка дозволяє користувачам створювати пости, залишати коментарі та виражати свої вподобання за допомогою лайків. Крім того, було забезпечено можливість отримувати список постів, коментарів та статистику лайків. Така програмна реалізація відкриває широкі можливості для взаємодії користувачів з платформою та створення соціального взаємодії.

Комбінація бази даних, серверної та клієнтської частини дозволяє створити ефективну та зручну систему для управління та спілкування з великою кількістю даних. Програмна реалізація підтримує розширення та може бути пристосована до потреб проекту.

У цілому, ця програмна реалізація демонструє потужність та гнучкість використання бази даних, серверної та клієнтської частини для створення функціональних та інтерактивних мобільних додатків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Axios [Електронний ресурс] – Режим доступу до ресурсу: https://axios-http.com/docs/intro.

2. Express/Node introduction [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction.

3. Google trends [Електронний ресурс] – Режим доступу до ресурсу: https://trends.google.com/trends/explore?q=how%20to%20make.

4. JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

5. Lodash [Електронний ресурс] – Режим доступу до ресурсу: https://lodash.com/.

6. Modularised architecture [Електронний ресурс] – Режим доступу до ресурсу: https://miro.medium.com/max/720/0*Cn2q2B9ZtCizT6G7.

7. Native Base [Електронний ресурс] – Режим доступу до ресурсу: https://nativebase.io/.

8. Postman [Електронний ресурс] – Режим доступу до ресурсу: https://www.postman.com/.

9. React Native [Електронний ресурс] – Режим доступу до ресурсу: https://reactnative.dev/.

10. React-native-toast-message [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/calintamas/react-native-toast-message.

11. React native vector icons [Електронний ресурс] – Режим доступу до ресурсу: https://www.npmjs.com/package/react-native-vector-icons.

12. React Navigation [Електронний ресурс] – Режим доступу до ресурсу: https://reactnavigation.org/.

13. Redux [Електронний ресурс] – Режим доступу до ресурсу: https://redux.js.org/.

14. Siripathi S. Мови програмування для мобільної розробки [Електронний ресурс] / Sandamal Siripathi – Режим доступу до ресурсу: https://code.tutsplus.com/uk/articles/mobile-development-languages--cms-29138.

15. The principles of building modern, scalable mobile applications [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/miquido/the-principles-of-building-modern-scalable-mobile-applications-128ecd808d4e.

16. What is PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-postgresql/.

17. Вікіпедія:Видання, що публікують неправдиву інформацію [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%92%D1%96%D0%BA%D1%96%D0%BF%D0%B5%D0%B4%D1%96%D1%8F:%D0%92%D0%B8%D0%B4%D0%B0%D0%BD%D0%BD%D1%8F,_%D1%89%D0%BE_%D0%BF%D1%83%D0%B1%D0%BB%D1%96%D0%BA%D1%83%D1%8E%D1%82%D1%8C_%D0%BD%D0%B5%D0%BF%D1%80%D0%B0%D0%B2%D0%B4%D0%B8%D0%B2%D1%83_%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D1%96%D1%8E.

ДОДАТОК А

Лістинг серверної частини:

Головний файл

```
import express, { Request, Response, NextFunction } from 'express'
import router from './routes/index.js'
import cors from 'cors'
import dotenv from 'dotenv'

dotenv.config()

const PORT = process.env.PORT || 8000

const app = express()

const corsOptions = {
  origin: '*',
  optionsSuccessStatus: 200,
}

app.use((req: Request, res: Response, next: NextFunction) => {
  const origin = '*'
  res.setHeader('Access-Control-Allow-Origin', origin)
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT,
DELETE')
  res.setHeader('Access-Control-Allow-Headers', 'Origin, X-
Requested-With, Content-Type, Accept, Authorization')
  res.setHeader('Access-Control-Allow-Credentials', 'false')
  next()
})

app.use(cors(corsOptions))

app.use(express.json())
app.use('/api', router)
```

```
app.use((error: Error, req: Request, res: Response) => {
  res.status(500).json({
    message: 'Something went wrong',
    error,
    location: 'Main File',
  })
})

app.listen(PORT, () => console.log(`Server started on port -
${PORT}`))

export default app
```

Контролер авторизації:

```
import { Request, Response } from 'express'
import bcrypt from 'bcrypt'
import jwt from 'jsonwebtoken'
import db from '../db.js'

export type User = {
  id: number
  username: string
  password: string
  secret_word: string
  refresh_token: string
}

type UserRegisterBody = {
  username: string
  password: string
  secret_word: string
}

type UserAuthBody = {
```

```
  username: string
  password: string
}

export const register = async (req: Request, res: Response): Promise<void> => {
  try {
    const { username, password, secret_word } = req.body as UserRegisterBody

    const users = await db.query<User>('SELECT * from users where username = $1', [username])

    if (!!users?.rows[0]) {
      res.status(400).json({
        message: 'Name already in use',
      })

      return
    }

    const salt = bcrypt.genSaltSync(10)
    const hashedPassword = bcrypt.hashSync(password, salt)
    const hashedSecretWord = bcrypt.hashSync(secret_word, salt)

    const newPerson = await db.query<User>(
      'INSERT INTO users (username, password, secret_word) values ($1, $2, $3) returning *',
      [username, hashedPassword, hashedSecretWord],
    )
    const newUser = newPerson.rows[0]

    const accessToken = jwt.sign({ id: newUser.id, username }, process.env.ACCESS_TOKEN_KEY || '', {
      expiresIn: '1h',
    })
```

```
    const refreshToken = jwt.sign({ id: newUser.id, username },
process.env.REFRESH_TOKEN_KEY || '', {
      expiresIn: '30d',
    })

    const user = await db.query('UPDATE users set refresh_token =
$1 where username = $2 returning *', [
      refreshToken,
      username,
    ])

    res.status(200).json({ data: user.rows[0], accessToken })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Register method',
    })
  }
}

export const login = async (req: Request, res: Response):
Promise<void> => {
  try {
    const { username, password } = req.body as UserAuthBody
    const users = await db.query<User>('SELECT * from users where
username = $1', [username])

    if (!users?.rows[0]) {
      res.status(400).json({
        message: 'User does not exist',
      })

      return
    }
```

```
    const passwordCorrect = bcrypt.compareSync(password,
users?.rows[0]?.password)

    if (!passwordCorrect) {
      res.status(400).json({
        message: 'Password is not correct',
      })

      return
    }

    const newUser = users.rows[0]

    const accessToken = jwt.sign({ id: newUser.id, username },
process.env.ACCESS_TOKEN_KEY || '', {
      expiresIn: '1h',
    })
    const refreshToken = jwt.sign({ id: newUser.id, username },
process.env.REFRESH_TOKEN_KEY || '', {
      expiresIn: '30d',
    })

    const user = await db.query('UPDATE users set refresh_token =
$1 where username = $2 returning *', [
      refreshToken,
      username,
    ])

    res.status(200).json({ data: user.rows[0], accessToken })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Login method',
    })
  }
```

```
}

export const profile = async (req: Request, res: Response): Promise<void> => {
  try {
    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    if (!user) {
      res.status(400).json({
        message: 'Invalid token',
      })

      return
    }

    const dbUser = await db.query('SELECT * from users where username = $1', [user.username])

    if (!dbUser?.rows[0]) {
      res.status(400).json({
        message: 'User does not exist',
      })

      return
    }

    res.status(200).json(dbUser?.rows[0])
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Get Profile method',
    })
  }
}
```

```
export const passwordReset = async (req: Request, res: Response): Promise<void> => {
  try {
    const { username, password, secret_word } = req.body as UserRegisterBody

    const users = await db.query('SELECT * from users where username = $1', [username])

    if (!users?.rows[0]) {
      res.status(400).json({
        message: 'User does not exist',
      })

      return
    }

    const secretWordCorrect = bcrypt.compareSync(secret_word, users?.rows[0]?.secret_word)

    if (!secretWordCorrect) {
      res.status(400).json({
        message: 'Secret code is not correct',
      })

      return
    }

    const salt = bcrypt.genSaltSync(10)
    const hashedPassword = bcrypt.hashSync(password, salt)

    await db.query('UPDATE users set password = $1 where username = $2 returning *', [hashedPassword, username])

    res.status(200).json({
```

```
      message: 'Password changed successfully',
    })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Password reset method',
    })
  }
}

export const refreshToken = async (req: Request, res: Response):
Promise<void> => {
  const token = req.body.token

  if (!token) {
    res.status(403).send('A token is required for refreshing')
  }

  try {
    jwt.verify(token, process.env.REFRESH_TOKEN_KEY as string)

    const user = jwt.decode(token) as User

    const users = await db.query<User>('SELECT * from users where
username = $1', [user.username])

    if (!users?.rows[0]) {
      res.status(401).json({
        message: 'User does not exist',
      })

      return
    }

    if (users?.rows[0]?.refresh_token !== token) {
```

```
      res.status(401).json({
        message: 'Tokens do not match',
      })

      return
    }

    const accessToken = jwt.sign({ id: user.id, username: user.username }, process.env.ACCESS_TOKEN_KEY || '', {
      expiresIn: '1h',
    })
    const refreshToken = jwt.sign({ id: user.id, username: user.username }, process.env.REFRESH_TOKEN_KEY || '', {
      expiresIn: '30d',
    })

    await db.query('UPDATE users set refresh_token = $1 where username = $2 returning *', [refreshToken, user.username])

    res.status(200).json({ accessToken, refreshToken })
  } catch (error) {
    res.status(401).send('Invalid Token')
  }
}

export const updateUserData = async (req: Request, res: Response): Promise<void> => {
  try {
    const { username, nickname } = req.body

    let updateQuery = 'UPDATE users SET'
    const values: any[] = []

    if (username) {
      const users = await db.query<User>('SELECT * from users where username = $1', [username])
```

```
      if (!!users?.rows[0]) {
        res.status(400).json({
          message: 'Username already in use',
        })

        return
      }

      updateQuery += ' username = $1'
      values.push(username)
    }

    if (nickname) {
      if (username) {
        updateQuery += ','
      }
      updateQuery += ' nickname = $' + (values.length + 1)
      values.push(nickname)
    }

    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    updateQuery += ' WHERE username = $' + (values.length + 1) +
' RETURNING *'
    values.push(user?.username)

    const updatedUser = await db.query(updateQuery, values)

    res.status(200).json(updatedUser.rows[0])
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Update user data method',
```

```
    })
  }
}
```

Контролер категорій:

```
import { Request, Response } from 'express'
import jwt from 'jsonwebtoken'
import { User } from './auth.controller.js'
import db from '../db.js'

type CategoryRequestBody = {
  title: string
}

export const createCategory = async (req: Request, res: Response): Promise<void> => {
  try {
    const { title } = req.body as CategoryRequestBody

    const categories = await db.query<User>('SELECT * from categories where title = $1', [title])

    if (!!categories?.rows[0]) {
      res.status(400).json({
        message: 'The category has already been created',
      })

      return
    }

    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    const newCategory = await db.query('INSERT INTO categories (title, user_id) values ($1, $2) returning *', [
```

```
      title,
      user?.id,
    ])

    res.status(200).json(newCategory.rows[0])
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Create category method',
    })
  }
}

export const getCategories = async (req: Request, res: Response):
Promise<void> => {
  try {
    const { limit = 10, title = '' } = req.query

    const categories = await db.query(
      `
      SELECT categories.*, COUNT(posts.*) AS post_count
      FROM categories
      LEFT JOIN posts ON categories.id = posts.category_id
      WHERE categories.title ILIKE $1 || '%'
      GROUP BY categories.id
      LIMIT $2`,
      [title, limit],
    )

    const total = await db.query(`SELECT COUNT(*) from categories
WHERE title ILIKE $1 || '%'`, [title])

    const formattedCategories = categories.rows.map((row) => ({
      ...row,
      post_count: +row.post_count,
```

```
    }))

    res.status(200).json({ data: formattedCategories, total:
+total.rows[0].count })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Get categories method',
    })
  }
}

export const updateCategory = async (req: Request, res: Response):
Promise<void> => {
  try {
    const { title } = req.body as CategoryRequestBody
    const id = req.params.id

    const categories = await db.query('SELECT * from categories
where id = $1', [id])

    if (!categories?.rows[0]) {
      res.status(400).json({
        message: 'Category does not exist',
      })

      return
    }

    const category = await db.query('UPDATE categories set title
= $1 where id = $2 returning *', [title, id])

    res.status(200).json(category?.rows[0])
  } catch (error) {
    res.status(500).json({
```

```
      message: 'Something went wrong',
      error,
      location: 'Update category method',
    })
  }
}

export const deleteCategory = async (req: Request, res: Response):
Promise<void> => {
  try {
    const id = req.params.id

    const categories = await db.query('SELECT * from categories
where id = $1', [id])

    if (!categories?.rows[0]) {
      res.status(400).json({
        message: 'Category does not exist',
      })

      return
    }

    await db.query('DELETE from categories where id = $1', [id])

    res.status(200).json({
      message: 'Category successfully deleted',
    })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Delete category method',
    })
  }
}
```

Контролер постів:

```
import { Request, Response } from 'express'
import jwt from 'jsonwebtoken'
import { User } from './auth.controller.js'
import db from '../db.js'

type PostRequestBody = {
  title: string
  description: string
  category_id: number
}

export const createPost = async (req: Request, res: Response): Promise<void> => {
  try {
    const { title, description, category_id } = req.body as PostRequestBody

    if (!title || !description || !category_id) {
      res.status(400).json({
        message: 'Bad request',
      })

      return
    }

    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    const newPost = await db.query(
      'INSERT INTO posts (title, description, category_id, user_id) values ($1, $2, $3, $4) returning *',
      [title, description, category_id, user?.id],
    )
```

```
    res.status(200).json(newPost.rows[0])
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Create post method',
    })
  }
}

export const getPosts = async (req: Request, res: Response): Promise<void> => {
  try {
    const { limit = 10, title = '', sort_variant = 'ASC', category_id = null, user_id = null } = req.query

    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    let query = `SELECT p.id, p.user_id, p.category_id, p.title, p.description, p.created_at,
        COALESCE(SUM(CASE WHEN l.liked THEN 1 ELSE 0 END), 0) AS likes_count,
        COALESCE(SUM(CASE WHEN NOT l.liked THEN 1 ELSE 0 END), 0) AS dislikes_count,
        MAX(CASE WHEN l.user_id = $1 THEN CAST(l.liked AS INT) ELSE NULL END) AS liked,
        u.username, u.id AS user_id, u.nickname
      FROM posts p
      LEFT JOIN likes_for_posts l ON p.id = l.post_id
      JOIN users u ON p.user_id = u.id
      WHERE p.title ILIKE $2 || '%'`

    const params = [user?.id, title, limit]

    if (!!category_id) {
```

```
  query += ' AND p.category_id = $4'

  params.push(category_id)
}

if (!!user_id) {
  const num = !!category_id ? 5 : 4

  query += ` AND p.user_id = $${num}`

  params.push(user_id)
}

query += ` GROUP BY p.id, u.id
  ORDER BY p.created_at ${sort_variant}
  LIMIT $3`

const result = await db.query(query, params)

const posts = result.rows.map((row) => ({
  id: row.id,
  user_id: row.user_id,
  category_id: row.category_id,
  title: row.title,
  description: row.description,
  created_at: row.created_at,
  likes_count: row.likes_count,
  dislikes_count: row.dislikes_count,
  liked: row.liked,
  user: {
    id: row.user_id,
    username: row.username,
    nickname: row.nickname,
  },
}))
```

```
    const userPosition = category_id ? 3 : 2

    const totalResult = await db.query(
      `SELECT COUNT(*) from posts WHERE title ILIKE $1 || '%'
${category_id ? 'AND category_id = $2' : ''} ${
        user_id ? `AND user_id = $${userPosition}` : ''
      }`,
      user_id
        ? category_id
          ? [title, category_id, user_id]
          : [title, user_id]
        : category_id
        ? [title, category_id]
        : [title],
    )

    const total = +totalResult.rows[0].count

    res.status(200).json({ data: posts, total })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Get posts method',
    })
  }
}

export const getPostById = async (req: Request, res: Response):
Promise<void> => {
  try {
    const id = req.params.id
    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    const query = `
```

```
      SELECT p.*,
        COALESCE(SUM(CASE WHEN l.liked THEN 1 ELSE 0 END), 0) as
likes_count,
        COALESCE(SUM(CASE WHEN NOT l.liked THEN 1 ELSE 0 END), 0)
as dislikes_count,
        MAX(CASE WHEN l.user_id = $2 THEN CAST(l.liked AS INT)
ELSE NULL END) as liked,
        u.id AS user_id, u.username, u.nickname
      FROM posts p
      LEFT JOIN likes_for_posts l ON p.id = l.post_id
      JOIN users u ON p.user_id = u.id
      WHERE p.id = $1
      GROUP BY p.id, u.id
    `

    const params = [id, user?.id]

    const result = await db.query(query, params)
    const post = result.rows[0]

    if (!post) {
      res.status(400).json({
        message: 'Post does not exist',
      })
      return
    }

    const populatedPost = {
      id: post.id,
      user_id: post.user_id,
      category_id: post.category_id,
      title: post.title,
      description: post.description,
      created_at: post.created_at,
      likes_count: post.likes_count,
      dislikes_count: post.dislikes_count,
```

```
      liked: post.liked,
      user: {
        id: post.user_id,
        username: post.username,
        nickname: post.nickname,
      },
    }

    res.status(200).json(populatedPost)
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Get post by id method',
    })
  }
}

export const updatePost = async (req: Request, res: Response): Promise<void> => {
  try {
    const { title, description, category_id } = req.body as PostRequestBody
    const id = req.params.id

    const posts = await db.query('SELECT * from posts where id = $1', [id])

    if (!posts?.rows[0]) {
      res.status(400).json({
        message: 'Post does not exist',
      })

      return
    }
```

```
    const post = await db.query(
      'UPDATE posts set title = $1, description = $2, category_id
= $3 where id = $4 returning *',
      [title, description, category_id, id],
    )

    res.status(200).json(post?.rows[0])
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Update post method',
    })
  }
}

export const deletePost = async (req: Request, res: Response):
Promise<void> => {
  try {
    const id = req.params.id

    const posts = await db.query('SELECT * from posts where id =
$1', [id])

    if (!posts?.rows[0]) {
      res.status(400).json({
        message: 'Post does not exist',
      })

      return
    }

    await db.query('DELETE from posts where id = $1', [id])

    res.status(200).json({
      message: 'Post successfully deleted',
```

```
    })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Delete post method',
    })
  }
}
```

Контролер коментарів:

```
import { Request, Response } from 'express'
import jwt from 'jsonwebtoken'
import { User } from './auth.controller.js'
import db from '../db.js'

type CommentRequestBody = {
  text: string
  post_id: number
}

type CommentUpdateBody = {
  text: string
}

export const createComment = async (req: Request, res: Response):
Promise<void> => {
  try {
    const { text, post_id } = req.body as CommentRequestBody

    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    const newComment = await db.query('INSERT INTO comments (text,
post_id, user_id) values ($1, $2, $3) returning *', [
```

```
      text,
      post_id,
      user?.id,
    ])

    res.status(200).json(newComment.rows[0])
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Create comment method',
    })
  }
}

export const getComments = async (req: Request, res: Response):
Promise<void> => {
  try {
    const { limit = 10, sort_variant = 'ASC', post_id = null } =
req.query

    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    const query = `
      SELECT c.*,
        COALESCE(SUM(CASE WHEN l.liked THEN 1 ELSE 0 END), 0) AS
likes_count,
        COALESCE(SUM(CASE WHEN NOT l.liked THEN 1 ELSE 0 END), 0)
AS dislikes_count,
        MAX(CASE WHEN l.user_id = $1 THEN CAST(l.liked AS INT)
ELSE NULL END) AS liked,
        u.username, u.id AS user_id, u.nickname
      FROM comments c
      LEFT JOIN likes_for_comments l ON c.id = l.comment_id
      JOIN users u ON c.user_id = u.id
```

```
      WHERE c.post_id = $2 OR ($2 IS NULL AND c.post_id IS NULL)
      GROUP BY c.id, u.id
      ORDER BY c.created_at ${sort_variant}
      LIMIT $3
    `

    const commentsResult = await db.query(query, [user?.id,
post_id !== null ? post_id : undefined, limit])

    const comments = commentsResult.rows.map((row) => ({
      id: row.id,
      text: row.text,
      user_id: row.user_id,
      post_id: row.post_id,
      created_at: row.created_at,
      likes_count: row.likes_count,
      dislikes_count: row.dislikes_count,
      liked: row.liked,
      user: {
        id: row.user_id,
        username: row.username,
        nickname: row.nickname,
      },
    }))

    const totalResult = await db.query(
      `SELECT COUNT(*) from comments WHERE post_id = $1 OR ($1 IS
NULL AND post_id IS NULL)`,
      [post_id],
    )
    const total = +totalResult.rows[0].count

    res.status(200).json({ data: comments, total })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
```

```
      error,
      location: 'Get comments method',
    })
  }
}

export const getCommentById = async (req: Request, res: Response):
Promise<void> => {
  try {
    const id = req.params.id
    const token = req.headers['authorization'] as string
    const user = jwt.decode(token?.split(' ')[1]) as User | null

    const query = `
      SELECT c.*,
        COALESCE(SUM(CASE WHEN l.liked THEN 1 ELSE 0 END), 0) AS
likes_count,
        COALESCE(SUM(CASE WHEN NOT l.liked THEN 1 ELSE 0 END), 0)
AS dislikes_count,
        MAX(CASE WHEN l.user_id = $1 THEN CAST(l.liked AS INT)
ELSE NULL END) AS liked,
        u.username, u.id AS user_id, u.nickname
      FROM comments c
      LEFT JOIN likes_for_comments l ON c.id = l.comment_id
      JOIN users u ON c.user_id = u.id
      WHERE c.id = $2
      GROUP BY c.id, u.id
    `

    const commentsResult = await db.query(query, [user?.id, id])

    if (!commentsResult?.rows[0]) {
      res.status(400).json({
        message: 'Comment does not exist',
      })
      return
```

```
    }

    const comment = commentsResult.rows[0]

    const commentData = {
      id: comment.id,
      text: comment.text,
      user_id: comment.user_id,
      content: comment.content,
      created_at: comment.created_at,
      likes_count: comment.likes_count,
      dislikes_count: comment.dislikes_count,
      liked: comment.liked,
      user: {
        id: comment.user_id,
        username: comment.username,
        nickname: comment.nickname,
      },
    }

    res.status(200).json(commentData)
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Get comment by id method',
    })
  }
}

export const updateComment = async (req: Request, res: Response):
Promise<void> => {
  try {
    const { text } = req.body as CommentUpdateBody
    const id = req.params.id
```

```
    const comments = await db.query('SELECT * from comments where
id = $1', [id])

    if (!comments?.rows[0]) {
      res.status(400).json({
        message: 'Comment does not exist',
      })

      return
    }

    const comment = await db.query('UPDATE comments set text = $1
where id = $2 returning *', [text, id])

    res.status(200).json(comment?.rows[0])
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Update comment method',
    })
  }
}

export const deleteComment = async (req: Request, res: Response):
Promise<void> => {
  try {
    const id = req.params.id

    const comments = await db.query('SELECT * from comments where
id = $1', [id])

    if (!comments?.rows[0]) {
      res.status(400).json({
        message: 'Comment does not exist',
      })
```

```
      return
    }

    await db.query('DELETE from comments where id = $1', [id])

    res.status(200).json({
      message: 'Comment successfully deleted',
    })
  } catch (error) {
    res.status(500).json({
      message: 'Something went wrong',
      error,
      location: 'Delete comment method',
    })
  }
}
```

ДОДАТОК Б

Лістинг клієнтської частини:

Екран логіну:

```
import React, { useCallback, useState } from 'react'
import { SafeAreaView } from 'react-native-safe-area-context'
import { Input, Divider, Pressable, Icon, FormControl } from 'native-base'
import { useDispatch, useSelector } from 'react-redux'
import { useFocusEffect, useNavigation } from '@react-navigation/native'
import { NativeStackNavigationProp } from '@react-navigation/native-stack'
import MaterialIcons from 'react-native-vector-icons/MaterialIcons'

import styles from './index.styled'

import CustomButton from '../../components/CustomButton'
import { AuthStackParamList } from '../../navigation/AuthNavigator'
import {
  selectAuthUsername,
  selectAuthPassword,
  selectAuthLoading,
} from '../../features/auth/selectors'
import {
  setAuthUsername,
  setAuthPassword,
  clearAuthFields,
  loginRequest,
} from '../../features/auth/authSlice'

const LoginScreen = () => {
```

```
  const [showPassword, setShowPassword] = useState<boolean>(false)
  const [showError, setShowError] = useState<boolean>(false)

  const dispatch = useDispatch()

  const { navigate } =

useNavigation<NativeStackNavigationProp<AuthStackParamList>>()

  const loading = useSelector(selectAuthLoading)
  const username = useSelector(selectAuthUsername)
  const password = useSelector(selectAuthPassword)

  const onRegisterPress = () => {
    navigate('Register')
  }

  const onRestorePasswordPress = () => {
    navigate('RestorePassword')
  }

  const onUsernameChange = (text: string) =>
dispatch(setAuthUsername(text))

  const onPasswordChange = (text: string) =>
dispatch(setAuthPassword(text))

  const onLoginPress = () => {
    if (!showError) {
      setShowError(true)
    }

    if (!username.isValid || !password.isValid) {
      return
    }
```

```
    dispatch(loginRequest())
  }

  useFocusEffect(
    useCallback(() => {
      return () => {
        dispatch(clearAuthFields())

        setShowError(false)
      }
    }, [dispatch]),
  )

  return (
    <SafeAreaView style={styles.container}>
      <FormControl isInvalid={showError && !username.isValid}>
        <Input
          variant="rounded"
          placeholder="Username"
          value={username.value ?? ''}
          onChangeText={onUsernameChange}
        />

        <FormControl.ErrorMessage>
          Username is invalid. It must contain at least 5
characters
        </FormControl.ErrorMessage>
      </FormControl>

      <Divider orientation="vertical" size={8} bg="transparent"
/>

      <FormControl isInvalid={showError && !password.isValid}>
        <Input
          variant="rounded"
```

```
          placeholder="Password"
          value={password.value ?? ''}
          onChangeText={onPasswordChange}
          type={showPassword ? 'text' : 'password'}
          InputRightElement={
            <Pressable onPress={() => setShowPassword(prev => !prev)}>
              <Icon
                as={
                  <MaterialIcons
                    name={showPassword ? 'visibility' : 'visibility-off'}
                  />
                }
                size={5}
                mr="2"
                color="muted.400"
              />
            </Pressable>
          }
        />

        <FormControl.ErrorMessage>
          Password is invalid. It must contain at least 8 characters, one
          capital letter and one number
        </FormControl.ErrorMessage>
      </FormControl>

      <Divider orientation="vertical" size={8} bg="transparent" />

      <CustomButton title="Login" onPress={onLoginPress} isLoading={loading} />
```

```
      <Divider  orientation="vertical"  size={4}  bg="transparent"
/>

      <CustomButton
        title="Register"
        onPress={onRegisterPress}
        variant="subtle"
      />

      <Divider  orientation="vertical"  size={4}  bg="transparent"
/>

      <CustomButton
        title="Restore Password"
        onPress={onRestorePasswordPress}
        variant="subtle"
      />
    </SafeAreaView>
  )
}

export default LoginScreen
```

Екран категорій:

```
import React from 'react'
import { SafeAreaView } from 'react-native-safe-area-context'
import { Text, FlatList } from 'react-native'
import { Divider } from 'native-base'
import { useDispatch, useSelector } from 'react-redux'

import styles from './index.styled'
import CategoryItem from './components/CategoryItem'

import SearchInput from '../../components/SearchInput'
import {
```

```
  selectCategorySearch,
  selectCategoryCategories,
  selectCategoryTotal,
} from '../../features/category/selectors'
import {
  Category,
  searchCategoryRequest,
  setCategorySearch,
} from '../../features/category/categorySlice'

const CategoryScreen = () => {
  const dispatch = useDispatch()

  const total = useSelector(selectCategoryTotal)
  const search = useSelector(selectCategorySearch)
  const categories = useSelector(selectCategoryCategories)

  const onSearchChange = (text: string) =>
dispatch(setCategorySearch(text))

  const onSearchPress = () => dispatch(searchCategoryRequest())

  const keyExtractor = (item: Category) => item.id.toString()

  const onEndReached = () => {
    if (categories.length >= total) return

    dispatch(searchCategoryRequest({ loadMore: true } as any))
  }

  const renderItem = ({ item }: { item: Category }) => (
    <CategoryItem item={item} />
  )

  return (
    <SafeAreaView style={styles.container}>
```

```
      <Divider orientation="vertical" size={4} bg="transparent"
/>

      <Text style={styles.title}>Categories</Text>

      <Divider orientation="vertical" size={8} bg="transparent"
/>

      <SearchInput
        placeholder="Search Category"
        value={search}
        onChangeText={onSearchChange}
        onPress={onSearchPress}
      />

      <Divider orientation="vertical" size={8} bg="transparent"
/>

      <FlatList
        data={categories}
        keyExtractor={keyExtractor}
        renderItem={renderItem}
        onEndReached={onEndReached}
        showsVerticalScrollIndicator={false}
      />
    </SafeAreaView>
  )
}

export default CategoryScreen
```

Екран постів:

```
import React, { useCallback, useEffect } from 'react'
import { SafeAreaView } from 'react-native-safe-area-context'
```

```
import { Text, FlatList, View, TouchableOpacity } from 'react-native'
import { Divider } from 'native-base'
import { useDispatch, useSelector } from 'react-redux'
import Icon from 'react-native-vector-icons/FontAwesome'
import { useFocusEffect, useNavigation } from '@react-navigation/native'
import { NativeStackNavigationProp } from '@react-navigation/native-stack'

import styles from './index.styled'
import PostItem from './components/PostItem'

import SearchInput from '../../components/SearchInput'
import {
  selectPostSearch,
  selectPostPosts,
  selectPostTotal,
  selectPostCategoryId,
} from '../../features/post/selectors'
import { colors } from '../../theme'
import {
  Post,
  clearPostFields,
  searchPostRequest,
  setPostSearch,
} from '../../features/post/postSlice'
import { HomeStackParamList } from '../../navigation/HomeNavigator'

const PostScreen = () => {
  const dispatch = useDispatch()

  const { goBack, navigate } =

useNavigation<NativeStackNavigationProp<HomeStackParamList>>()
```

```
  const total = useSelector(selectPostTotal)
  const posts = useSelector(selectPostPosts)
  const search = useSelector(selectPostSearch)
  const categoryId = useSelector(selectPostCategoryId)

  const onSearchChange = (text: string) =>
dispatch(setPostSearch(text))

  const onSearchPress = () => dispatch(searchPostRequest())

  const keyExtractor = (item: Post) => item.id.toString()

  const onEndReached = () => {
    if (posts.length >= total) return

    dispatch(searchPostRequest({ loadMore: true } as any))
  }

  const renderItem = ({ item }: { item: Post }) => <PostItem
item={item} />

  const onPlusPress = () => navigate('PostCreateScreen')

  useEffect(() => {
    return () => {
      dispatch(clearPostFields())
    }
  }, [dispatch])

  useFocusEffect(
    useCallback(() => {
      if (categoryId !== null) {
        dispatch(searchPostRequest())
      }
    }, [dispatch, categoryId]),
```

```
  )

  return (
    <SafeAreaView style={styles.container}>
      <Divider  orientation="vertical"  size={4}  bg="transparent"
/>

      <View style={styles.header}>
        <TouchableOpacity onPress={goBack} hitSlop={20}>
          <Icon  name="arrow-left"  size={30}  color={colors.black}
/>
        </TouchableOpacity>

        <Text style={styles.title}>Posts</Text>

        <TouchableOpacity onPress={onPlusPress} hitSlop={20}>
          <Icon name="plus" size={30} color={colors.lime} />
        </TouchableOpacity>
      </View>

      <Divider  orientation="vertical"  size={8}  bg="transparent"
/>

      <SearchInput
        placeholder="Search Post"
        value={search}
        onChangeText={onSearchChange}
        onPress={onSearchPress}
      />

      <Divider  orientation="vertical"  size={8}  bg="transparent"
/>

      <FlatList
        data={posts}
        keyExtractor={keyExtractor}
```

```
        renderItem={renderItem}
        onEndReached={onEndReached}
        showsVerticalScrollIndicator={false}
      />
    </SafeAreaView>
  )
}

export default PostScreen
```

Екран коментарів:

```
import React from 'react'
import { SafeAreaView } from 'react-native-safe-area-context'
import { Text, FlatList, View, TouchableOpacity } from 'react-native'
import { Divider } from 'native-base'
import { useDispatch, useSelector } from 'react-redux'
import Icon from 'react-native-vector-icons/FontAwesome'
import { useNavigation } from '@react-navigation/native'
import { NativeStackNavigationProp } from '@react-navigation/native-stack'

import styles from './index.styled'
import {
  Comment,
  getCommentsRequest,
} from '../../features/comment/commentSlice'
import {
  selectComments,
  selectCommentTotal,
} from '../../features/comment/selectors'
import CommentItem from './components/CommentItem'
import { colors } from '../../theme'
import { HomeStackParamList } from '../../navigation/HomeNavigator'
```

```
const CommentScreen = () => {
  const dispatch = useDispatch()

  const { goBack, navigate } =

useNavigation<NativeStackNavigationProp<HomeStackParamList>>()

  const comments = useSelector(selectComments)
  const total = useSelector(selectCommentTotal)

  const keyExtractor = (item: Comment) => item.id.toString()

  const onEndReached = () => {
    if (comments.length >= total) return

    dispatch(getCommentsRequest({ loadMore: true } as any))
  }

  const renderItem = ({ item }: { item: Comment }) => (
    <CommentItem item={item} />
  )

  const onPlusPress = () => navigate('CommentCreateScreen')

  return (
    <SafeAreaView style={styles.container}>
      <Divider  orientation="vertical"  size={4}  bg="transparent"
/>

      <View style={styles.header}>
        <TouchableOpacity onPress={goBack} hitSlop={20}>
          <Icon  name="arrow-left"  size={30}  color={colors.black}
/>
        </TouchableOpacity>
```

```
        <Text style={styles.title}>Comments</Text>

        <TouchableOpacity onPress={onPlusPress} hitSlop={20}>
          <Icon name="plus" size={30} color={colors.lime} />
        </TouchableOpacity>
      </View>

      <Divider orientation="vertical" size={8} bg="transparent"
/>

      <FlatList
        data={comments}
        keyExtractor={keyExtractor}
        renderItem={renderItem}
        onEndReached={onEndReached}
        showsVerticalScrollIndicator={false}
        nestedScrollEnabled
      />
    </SafeAreaView>
  )
}

export default CommentScreen
```