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: «Мобільний додаток для онлайн-публікацій»

здобувача групи ІН – 92 Лисянського Захара Володимировича

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

_____ Захар ЛИСЯНСЬКИЙ
(підпис)

Керівник,
доцент, кандидат технічних наук

Сергій ПЕТРОВ _____
(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-92 Лисянського Захара Володимировича

1. Тема роботи: «Мобільний додаток для онлайн-публікацій»
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) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Огляд технологій, що використовуються для розробки мобільних додатків. 3) Розробка мобільного додатку для онлайн-публікацій. 4) Аналіз результатів.
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	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
	Огляд технологій, що використовуються для розробки		

2	*мобільних додатків*		
3	*Розробка мобільного додатку для онлайн-публікацій*		
4	*Аналіз отриманих результатів*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ Керівник _____

(підпис) (підпис)

АНОТАЦІЯ

Записка: 58 стр., 30 рис., 4 додаток, 15 використаних джерел.

Обґрунтування актуальності теми роботи – за останні роки мобільні пристрої стали невід'ємною частиною нашого повсякденного життя. Люди використовують їх для доступу до інформації, розваг, соціальних мереж та інших сервісів. Застосунок для онлайн-публікацій задовольнить ці потреби. Він дозволить користувачам створювати та споживати вміст, який відповідає їхнім індивідуальним інтересам.

Об’єкт дослідження — процес розробки мобільного крос-платформного додатку.

Мета роботи — створення мобільного застосунку з використанням технологій React Native, express та PostgreSQL, що дозволить користувачам ділитися власними думками та досвідом, знаходити та переглядами корисну інформацію згідно своїх інтересів.

Методи дослідження — методи розробки крос-платформних застосунків.

Результати — розроблено мобільний додаток, що дозволяє користувачам створювати та переглядати публікації, здійснювати пошук інформації відповідно своїх інтересів, залишати коментарі та оцінки.

МОБІЛЬНИЙ ДОДАТОК ДЛЯ ОНЛАЙН-ПУБЛІКАЦІЙ, REACT NATIVE, TYPESCRIPT, EXPRESS, POSTGRESQL

ЗМІСТ

ВСТУП

Люди, які живуть у сучасному цифровому суспільстві, постійно прагнуть полегшити своє життя. Інтернет став невід'ємною складовою частиною їхнього повсякденного існування. Це призвело до значного попиту на розробку мобільних додатків. У наш час, завдяки мобільному інтернету, розробка програм для мобільних пристроїв стала надзвичайно актуальною. Навіть у випадку відсутності домашньої Wi-Fi мережі, люди можуть завжди підключитися до Інтернету за допомогою свого смартфона, планшета або навіть годинника. Але без мобільних додатків, які працюють з цим підключенням, мобільний Інтернет мав би обмежене застосування. Ці додатки виконують безліч завдань, таких як робота з користувацькими даними, відправлення їх на сервери та обробка.

Сьогодні багато програмістів займаються розробкою різноманітних програм для платформ Android та iOS. Ці програми можуть мати різне застосування: комунікація, навчання, продаж товарів, ігри, читання та пошук корисної інформації. Мобільні додатки можуть використовуватися як для роботи, так і для розваг.

Об'єктом дослідження є процес розробки мобільного крос-платформного додатку.

В даній роботі буде описано створення мобільного застосунку для онлайн-публікацій, який дозволить користувачам писати публікації на різні теми, знаходити цікаву інформацію, відповідно своїх інтересів, коментувати та оцінювати публікації.

Дана робота складається зі вступу, літературного огляду, методики вирішення поставлених задач, програмної реалізації, висновку та додатків.

1. ЛІТЕРАТУРНИЙ ОГЛЯД

1.1. Огляд аналогів додатку для онлайн-публікацій

У цьому розділі будуть розглянуті деякі аналоги додатків для онлайн-публікацій, які вже існують на ринку. Аналіз таких додатків надає цінну інформацію для розробки та вдосконалення власного продукту. Нижче представлено кілька прикладів:

Medium є однією з провідних платформ для онлайн-публікацій, що надає можливість авторам ділитися своїми статтями та історіями. Додаток Medium пропонує зручну інтерфейс для читання та пошуку корисної інформації. Користувачі можуть залишати коментарі та публікувати власні матеріали.

Wattpad - це популярна платформа для онлайн-читання та публікації літератури. Мобільний додаток Wattpad забезпечує доступ до великої колекції безкоштовних електронних книг у різних жанрах. Користувачі можуть взаємодіяти зі змістом, залишати відгуки та навіть писати свої власні історії.

WordPress - це популярна платформа для веб-публікацій і керування контентом. Мобільний додаток WordPress надає зручні можливості адміністрування та редагування блогів або веб-сайтів. Користувачі можуть створювати, редагувати та опубліковувати публікації, а також взаємодіяти зі своєю аудиторією.

Flipboard - це персоналізована платформа новин та публікацій. Мобільний додаток Flipboard збирає вміст з різних джерел і презентує його у зручному форматі для читання. Користувачі можуть фільтрувати контент відповідно своїх інтересів, створювати власні журнали та ділитися змістом з іншими.

Аналізуючи ці аналоги, можна зрозуміти, які функції та можливості слід розглядати при розробці та покращенні власного мобільного застосунку

у сфері онлайн-публікацій. Також слід звернути увагу на інтерфейс та зручність у використанні.

1.2. Етапи створення мобільних додатків

Процес розробки мобільного додатку може бути розділений на наступні етапи:

1. **Аналіз продукту**. Спочатку визначається мета продукту. Проводиться дослідження ринку та конкурентів, включаючи непрямих. Формується відповідь на запитання "Як цей продукт допоможе користувачам?".
2. **Специфікація**. Створюється документ, який містить детальний опис мобільного додатку для розробників.
3. **Оцінка та планування**. На основі специфікації складається оцінка вартості та термінів реалізації проекту. Враховується обсяг робіт, витрати, ризики та запобіжні заходи.
4. **Дизайн**. Розробляється зовнішній вигляд додатку, включаючи дизайн-концепцію, компоненти інтерфейсу, макети та інтерактивні прототипи.
5. **Програмування**. Розробники реалізують функціонал додатку.
6. **Тестування**. Проводиться перевірка додатку на стійкість, надійність та безпеку, забезпечуючи високу якість продукту.
7. **Реліз**. Збірки проекту завантажуються в магазин додатків, такі як App Store для iOS та Google Play для Android, для доступу користувачів.

Ці етапи допомагають забезпечити надійний та ефективний процес розробки мобільного додатку.

1.3. Технології створення мобільних додатків

Існує кілька технологій і інструментів, які використовуються для створення мобільних додатків. Ось деякі з них:

Нативна розробка - це розробка додатків для конкретної платформи, такої як iOS (з використанням мови програмування Swift або Objective-C) або Android (з використанням Java або Kotlin). Цей підхід дозволяє отримати

найвищу продуктивність та можливості, але вимагає розробки окремого коду для кожної платформи.

Крос-платформна розробка - це підхід, при якому використовується одна кодова база для створення додатку, який може працювати на різних платформах. Популярні фреймворки для крос-платформної розробки: React Native, Xamarin і Flutter. Вони дозволяють ефективно використовувати загальний код і прискорюють процес розробки, але можуть мати обмеження щодо доступних можливостей платформи.

Веб-додатки можуть бути запущені через веб-браузер на мобільних пристроях і не вимагають встановлення окремого застосунку. Вони розробляються з використанням веб-технологій, таких як HTML, CSS і JavaScript. Фреймворки, такі як React, Angular і Vue.js, можуть бути використані для розробки мобільних веб-додатків.

Гібридна розробка - це поєднання нативного і крос-платформного підходів. Гібридні додатки використовують веб-технології для розробки користувацького інтерфейсу, який запускається у вбудованому браузері в мобільному додатку. Apache Cordova і Ionic є прикладами фреймворків для гібридної розробки.

Кожен з цих підходів має свої переваги та обмеження. У табл. 1.1 можна наглядно оцінити переваги та недоліки різних технологій створення мобільних додатків.

Таблиця 1.1 Порівняння технологій розробки мобільних застосунків

Технологія	**Переваги**	**Недоліки**
Нативна розробка	1. Нативні додатки показують найвищий рівень продуктивності та функціональності. 2. Має повний доступ до функцій та можливостей платформи. 3. Нативні додатки зазвичай мають кращу інтеграцію з апаратним забезпеченням пристрою.	1. Розробка для кожної платформи (iOS та Android) вимагає написання окремого коду. 2. Вимагає спеціалізованого знання мов програмування для кожної платформи.

Крос-платформна розробка	1. Один код може бути використаний для створення додатків на різних платформах. 2. Забезпечує швидке розгортання проекту і підтримку на різних платформах. 3. Можливість використання загальних ресурсів та бібліотек.	1. Обмеження в продуктивності та доступності функцій, порівняно з нативною розробкою. 2. Залежність від фреймворків, які можуть мати обмежені можливості або не враховувати останні можливості платформ.
Веб-додатки	1. Веб-технології широко поширені і знайомі для розробників. 2. Один код може працювати на різних платформах без необхідності розробки окремих версій для кожної платформи. 3. Простота оновлення і розгортання додатку.	1. Обмежені можливості доступу до апаратного забезпечення пристрою і операційної системи. 2. Не може використовувати всі можливості платформи, такі як повний доступ до API і функцій.
Гібридна розробка	1. Використання веб-технологій для розробки інтерфейсу, що забезпечує швидшу розробку. 2. Можливість запуску на різних платформах з використанням одного коду. 3. Швидкий цикл розробки і оновлення.	1. Обмеження в доступності функцій і можливостей платформи, порівняно з нативною розробкою. 2. Залежність від фреймворків і засобів розробки.

Важливо відзначити, що ключова відмінність між нативними та крос-платформними додатками полягає в їх продуктивності та здатності вирішувати низькорівневі задачі. Однак цей аспект стосується в основному великих проектів, тоді як малий і середній бізнес, а також IT-стартапи можуть вибрати крос-платформну розробку як вигідну альтернативу.

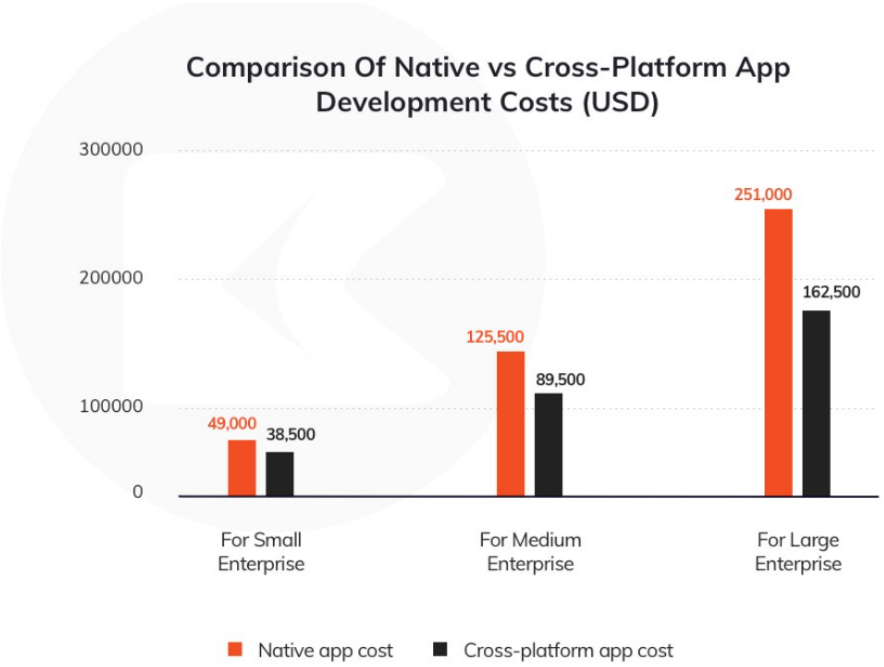

Рисунок 1.1 — Порівняння ціни нативної та крос-платформної [11]

На рисунку 1.1 можна побачити детальне порівняння затрат при виборі конкретної технології. Для малого бізнесу та стартапів швидкість виходу на ринок, вартість і легкість розробки має високий пріоритет, що є значущим аргументом при виборі технології на користь крос-платформної розробки.

1.4. Постановка задачі

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. провести аналіз технологій, їх особливостей, переваг та недоліків;
2. виконати проектування бази даних майбутнього додатку;
3. виконати проектування архітектури мобільного додатку та серверної частини;
4. виконати програмну реалізацію додатку згідно заданих умов.

У разі успішної реалізації ми отримаємо повноцінний мобільний додаток з можливістю створювати та переглядати публікації, здійснювати пошук інформації відповідно своїх інтересів, залишати коментарі та оцінки.

2. МЕТОДИКА ВИРІШЕННЯ ПОСТАВЛЕНИХ ЗАДАЧ

2.1 Проектування бази даних та вибір СУБД

2.1.1 Проектування бази даних

Процес проектування бази даних розпочинається з формулювання завдання та ідентифікації об'єктів, процесів та сутностей предметної області. Один з перших та основних етапів проектування бази даних - це концептуальне проектування, що включає збір, аналіз та редагування вимог до даних. Для поставленої мети виконуються наступні кроки:

- Проведення дослідження предметної галузі та вивчення її інформаційної структури.
- Виявлення всіх компонентів, кожен з яких має своє користувальницьке представлення, інформаційні об'єкти, зв'язки між ними та процеси.
- Моделювання та інтеграція всіх концепцій.

Результат цього етапу - концептуальна модель, інваріантна до структури бази даних, часто представляється у вигляді ERD-діаграми.

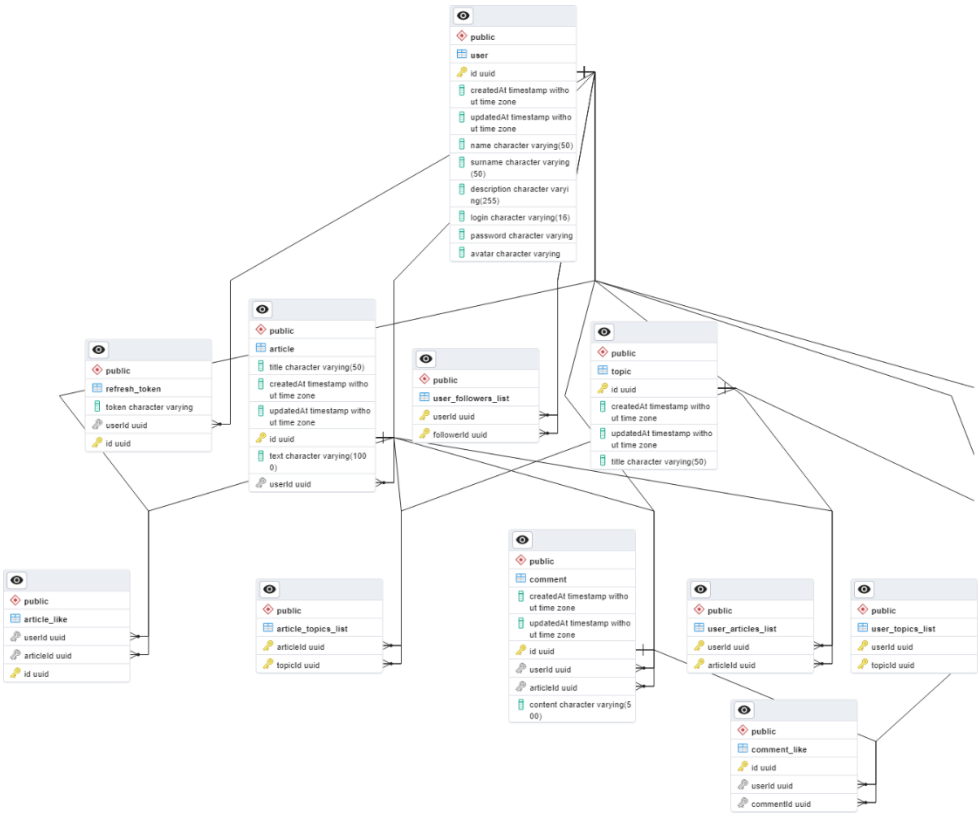

Рисунок 1.2 – ERD-діаграма бази даних

На рисунку 1.2 зображена ERD-діаграма на якій можна побачити всі необхідні сутності та зв'язки між ними для коректного функціонування додатку з усім необхідним функціоналом.

2.1.2 СУБД PostgreSQL

PostgreSQL є потужною та розширюваною відкритою СУБД, яка має широкі можливості, включаючи підтримку складних запитів, широкий вибір типів даних, безпеку транзакцій та реплікацію. Якщо потрібна СУБД з великим різноманіттям функції та великою гнучкістю, PostgreSQL є хорошим вибором.

PostgreSQL використовує реляційну модель та підтримує стандартну мову запитів SQL. Основні переваги PostgreSQL полягають у широкому спектрі можливостей, які майже повністю відповідають функціоналу інших систем управління базами даних, а також має в наявності додаткові функцій. Також PostgreSQL працює на більшості UNIX-платформ, включаючи UNIX-подібні системи, такі як FreeBSD та Linux.

Основні переваги та функціональні можливості PostgreSQL [9] такі:

- **Надійність**. PostgreSQL гарантує дотримання принципів ACID (атомарність, узгодженість, ізольованість, довговічність), використовує багатоверсійність та механізм протоколювання Write Ahead Logging (WAL) для реєстрації всіх транзакцій. Варто відзначити можливість відновлення бази даних до певного моменту в часі (Point in Time Recovery - PITR), реплікацію, підтримку цілісності даних на рівні схеми.
- **Розширені можливості запитів**. PostgreSQL підтримує ряд розширених можливостей запитів, таких як JOIN, підзапити, пов'язані запити, агрегаційні функції, віконні функції та інші. Це дозволяє виконувати складні та потужні запити до бази даних.

- **Типи даних**. PostgreSQL має широкий спектр вбудованих та користувацьких типів даних. Крім стандартних типів, таких як цілі числа, рядки та дата/час, PostgreSQL підтримує географічні типи даних (наприклад, точки, лінії, полігони), JSON-об'єкти, XML та багато інших. Це дозволяє зберігати та опрацьовувати різноманітні дані в базі даних.

- **Безпека**. PostgreSQL забезпечує механізм транзакцій для забезпечення цілісності даних. Він підтримує команди COMMIT та ROLLBACK для підтвердження або скасування транзакцій. Це дозволяє виконувати операції з базою даних в безпечному та надійному режимі.
- **Реплікація**. PostgreSQL має можливості для реплікації бази даних. Він підтримує різні методи реплікації, такі як Master-Slave реплікація та реплікація за допомогою кластерів. Це дозволяє створювати резервні копії даних, підвищувати доступність та розподіляти навантаження між серверами.
- **Розширюваність**. PostgreSQL дозволяє створювати користувацькі функції та розширення. Можна написати свої власні функції та типи даних на мовах програмування, таких як C, Python, Perl та інші, та використовувати їх у проектах. Це дозволяє розширювати функціональність PostgreSQL згідно зі своїми потребами.

2.2 Клієнтська частина мобільного додатку

React Native - це інноваційний фреймворк, який дозволяє створювати крос-платформні мобільні додатки швидко і ефективно. Його головна перевага полягає в тому, що потрібно розробити лише один проект, який працюватиме як на платформі IOS, так і на Android. Великі компанії, такі як Facebook, Instagram, Skype, Pinterest, Uber, Tesla, SoundCloud та інші, успішно використовують React Native у своїх проектах.

Основні переваги використання React Native:

- Підтримка розвитку з боку Facebook та широкої спільноти розробників.
- Ліцензія відкритого програмного забезпечення.
- Крос-платформність дозволяє створювати мобільні додатки, які працюють на платформах iOS та Android. Це дозволяє значно зекономити час і зусилля, оскільки не потрібно писати окремий код для кожної платформи.
- Завдяки архітектурі React Native, яка використовує нативні компоненти, додатки, створені з використанням цього фреймворку, мають високу продуктивність та швидкість роботи. Вони наближаються до поведінки нативних додатків.
- React Native має велику спільноту розробників, яка активно працює над його розвитком і підтримкою. На просторах інтернету можна знайти багато корисних ресурсів, документацій, бібліотек та інструментів, які допоможуть у розробці.

На жаль, є кілька недоліків, з якими варто ознайомитись:

- Проблеми з керуванням пам'яттю, які можуть вплинути на продуктивність додатків.
- Питання безпеки, пов'язані з використанням JavaScript.
- Деякі модулі та API можуть бути відсутніми, що потребуватиме додаткових зусиль для розробки певного функціоналу.

Загалом, враховуючи простоту використання, швидкість розробки та мінімальні втрати продуктивності, React Native є чудовим вибором для реалізації мобільних додатків.

В основі React Native краще використовувати синтаксис TypeScript, що дозволяє виразити рішення завдання у вигляді коду в файлі .ts або .tsx. Він є еволюцією синтаксису JavaScript, тому будь-яка програма JS синтаксично коректна на TypeScript. Компілятор tsc допоможе виявити безліч дефектів перед етапом випуску додатку. Він перетворює код TypeScript на JavaScript і аналізує програму, намагаючись знайти проблемні місця. TypeScript може створювати js-файли для будь-якої версії JavaScript, починаючи з ES3. Також

можна диктувати правила TypeScript, вмикаючи одні та вимикаючи інші [10]. На рисунку 1.3 можна побачити відмінності в написанні коду на TypeScript та JavaScript.

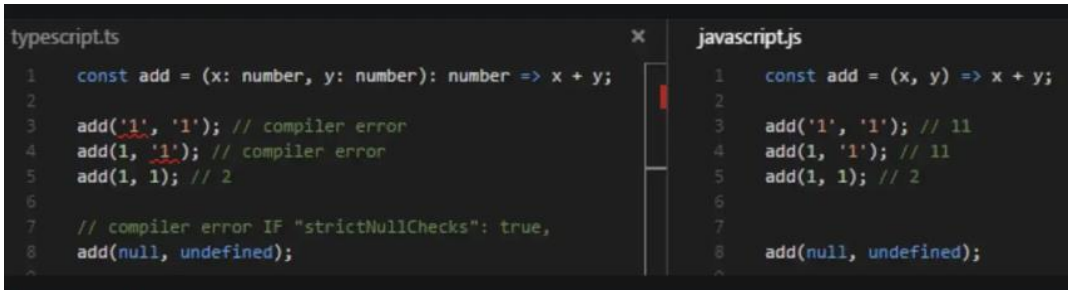

Рисунок 1.3 – TypeScript vs JavaScript [7]

Як бачимо TypeScript дає змогу запобігти появі помилок, що супроводжуються динамічною типізацією та неуважністю розробника.

2.3 Серверна частина мобільного додатку

Express - це мінімалістичний і гнучкий фреймворк для розробки API з використанням мови програмування JavaScript. Він побудований на основі Node.js і дозволяє створювати швидкі та масштабовані веб-сервери з дуже малою кількістю коду [3].

Також до переваг можна віднести:

- Можливість застосовувати одну мову на клієнті та сервері, що зробить можливим перевикористання функціоналу.
- Велике та бадьоре ком'юніті. Оскільки це відкрите програмне забезпечення, веб-розробники можуть писати різні модулі та пакети і ділитися ними. Найчастіше модулі добре комбінуються.
- Технологія швидко покращується, над цим працюють тисячі програмістів по всьому світу.

Спираючись на ці фактори фреймворк Express буде чудовим рішенням для написання серверної частини додатку. Гарним

доповненням є використання синтаксису TypeScript як описано в главі “Клієнтська частина мобільного додатку”.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1. Реалізація серверної частини

3.1.1. Структура серверної частини

У цьому розділі буде розглянута структура серверної частини додатку, реалізованого за допомогою Express.js з використанням TypeScript.

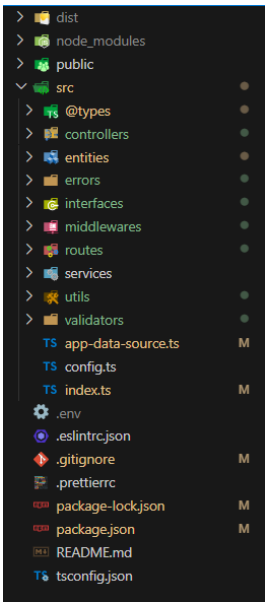

Рисунок 3.1 – Структура серверної частини

На рисунку 3.1 можна побачити загальну файлову структуру серверної частини. Розглянемо основні складові цієї структури:

1. Основні папки та файли проекту:

   src/ - ця папка містить весь вихідний код додатку.

   dist/ - папка, в яку компілюється TypeScript - код після збирання проекту.

   public/ - папка для зберігання публічних ресурсів в проекті.

   .env - це файл, що використовується для зберігання конфіденційної інформації, такої як секретні ключі API, паролі бази даних та ін.

   .eslintrc.json – конфігураційний файл для ESLint, який є популярним інструментом для статичного аналізу коду в JavaScript проектах.

   .prettierrc – це конфігураційний файл для Prettier, інструмента для автоматичного форматування коду в проектах.

.gitignore – це файл, який використовується в системі контролю версій Git для виключення певних файлів і папок з відстежування.

node_modules/ - це папка, що містить усі залежності (пакети) проекту Node.js.

2. Конфігурація проекту:

   tsconfig.json - файл конфігурації TypeScript, в якому вказуються налаштування компіляції та шляхи до вихідного та компільованого коду.

   package.json - файл залежностей та скриптів проекту.

3. Підключення до бази даних:

   src/app-data-source.ts - файл підключення до бази даних за допомогою TypeORM.

4. Основні функціональні частини:

   index.ts – головний файл проекту, в ньому виконується запуск серверу.

   src/routes/ - папка, в якій знаходяться файли, що відповідають за роутинг та обробку HTTP-запитів. Кожен файл представляє окремий шлях, або групу шляхів API.

   src/controllers/ - папка, в якій знаходяться контролери, що обробляють запити з роутів. Контролери виконують бізнес-логіку та взаємодіють з сервісами та базою даних.

   src/services/ - папка, в якій знаходяться сервіси, що виконують специфічні операції.

   src/entities/ - папка, в якій знаходяться моделі даних, які використовуються для взаємодії з базою даних та передачі даних між різними компонентами додатку.

   src/middlewares/ - папка, в якій знаходяться файли, що містять middleware - функції, які обробляють запити перед тим, як вони досягнуть роутів.

   @types/ – папка з глобальними типами додатку.

errors/ – папка в якій знаходяться обробники помилок.

interfaces/ - папка з інтерфейсами, які призначені для типізації даних надісланих з клієнтської частини.

utils/ – папка, яка використовуватися для зберігання корисних утиліт та допоміжних функцій.

validators/ – папка в якій зберігаються схеми валідації даних.

config.ts – це файл, який використовується для зберігання конфігураційних параметрів та налаштувань.

3.1.2. Основні використані бібліотеки

TypeORM (версія 0.3.15). Ця бібліотека дозволяє працювати з базами даних за допомогою TypeScript. Вона надає ORM (Object-Relational Mapping) для спрощення взаємодії з базою даних та забезпечує механізми міграцій, створення моделей даних та виконання запитів.

bcrypt (версія 5.1.0). Ця бібліотека надає функції хешування паролів. Вона дозволяє захищати паролі користувачів, шифруючи їх перед збереженням у базі даних.

cors (версія 2.8.5). Ця бібліотека дозволяє налаштовувати Cross-Origin Resource Sharing (CORS). Вона забезпечує контроль доступу до ресурсів сервера з інших доменів або портів.

helmet (версія 6.1.5). Ця бібліотека допомагає захищати додаток від різних видів атак, додавши різні заголовки безпеки до вихідних HTTP-відповідей.

jsonwebtoken (версія 9.0.0). Ця бібліотека надає інструменти для створення та перевірки JSON Web Tokens (JWT). JWT використовуються для аутентифікації та авторизації користувачів у веб-додатках.

joi (версія 17.9.2). Ця бібліотека дозволяє валідувати та перевіряти вхідні дані, такі як параметри запитів або дані форм.

multer (версія 1.4.5-lts.1). Ця бібліотека дозволяє завантажувати файли на сервер. Вона спрощує процес завантаження файлів в додатку.

dotenv (версія 16.0.3). Ця бібліотека дозволяє отримувати конфігураційні змінні з файлу ".env". Вона корисна для збереження конфіденційної інформації, такої як ключі API або налаштування бази даних, в окремому файлі, який не входить до системи контролю версій.

Ці бібліотеки використовуються для розширення функціональності серверної частини.

3.1.3. Опис моделей та взаємодія з ними

Функціонал бібліотеки TypeORM надає можливість автоматичного створення та виконання міграцій бази даних, що дозволяє контролювати структуру бази даних і зберігати її синхронізованою з моделями.

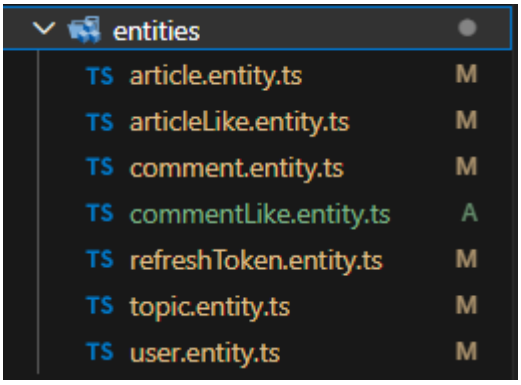

Рисунок 3.2 – Основні моделі сутностей

На рисунку 3.2 можна побачити основні моделі, які необхідні для коректного функціонування серверної частини відповідно до визначених вимог, для ознайомлення з кодом моделей див. Додаток А.

Варто відзначити, що проміжні таблиці: “user_followers_list”, “article_topics_list”, “user_articles_list” та “user_topics_list” – генеруються автоматично при зазначенні зв’язку "багато-до-багатьох".

Розглянемо клас “Article”, який використовується для моделювання відповідної таблиці в базі даних:

```
@Entity()
export class Article {
  @PrimaryGeneratedColumn('uuid')
  id: string
  @Column({ length: 50 })
```

```
  title: string
  @Column({ length: 1000 })
  text: string
  @CreateDateColumn()
  createdAt: Date
  @UpdateDateColumn()
  updatedAt: Date
  @OneToMany(() => Comment, (comment) => comment.article, { cascade: true })
  comments: Comment[]
  @OneToMany(() => ArticleLike, (articleLike) => articleLike.article, { cascade: true })
  articleLikes: ArticleLike[]
  // many - one
  @ManyToOne(() => User, (user) => user.articles, { onDelete: 'CASCADE' })
  @JoinColumn({ name: 'userId' })
  user: User
  // many - many
  @ManyToMany(() => Topic, (topic) => topic.articles, { onDelete: 'CASCADE' })
  @JoinTable({ name: 'article_topics_list' })
  topics: Topic[]
  @ManyToMany(() => User, (user) => user.savedArticles)
 savedByUsers: User[]
 }
```

Цей код описує клас “Article”, який представляє сутність "стаття" у системі. Варто відзначити деякі ключові елементи цього коду:

- Клас має декоратор “@Entity()”, що вказує, що це сутність бази даних, яка буде відображена у вигляді таблиці.
- Використовується декоратор “@PrimaryGeneratedColumn('uuid')” для створення унікального ідентифікатора статті, що генерується автоматично.
- За допомогою декораторів “@Column” визначаються колонки таблиці для полів “title” та “text” з відповідними обмеженнями довжини тексту.
- Декоратори “@CreateDateColumn()” та “@UpdateDateColumn()” встановлюють поля “createdAt” та “updatedAt”, які автоматично оновлюються при створенні та оновленні статті.

- За допомогою декоратора “@OneToMany” встановлюється зв'язок "один-до-багатьох" між статтею та коментарями, а також між статтею та лайками статей.
- За допомогою декоратора “@ManyToOne” встановлюється зв'язок "багато-до-одного" між статтею та користувачем, який її створив.
- За допомогою декоратора “@ManyToMany” встановлюється зв'язок "багато-до-багатьох" між статтею та темами статей, а також між статтею та користувачами, які її зберегли.

Цей код відображає структуру моделі “Article” і її взаємозв'язки з іншими класами. Він використовується для автоматичного створення та оновлення таблиці статей в базі даних та взаємодії з нею через ORM (Object-Relational Mapping) функціонал TypeORM.

Розглянемо приклад взаємодії з моделлю “Article” на прикладі функціоналу отримання списку статей відповідно заданих фільтрів:

- const { skip, take, sortBy, order, searchTerm, userId, topicId } = req.query as GetArticlesFilterQuery

В цьому рядку отримуємо параметри запиту, такі як “skip”, “take”, “sortBy”, “order”, “searchTerm”, “userId” та “topicId”, які будуть використовуватись для фільтрації та пагінації результатів.

- ```
  const queryBuilder = AppDataSource.getRepository(Article)
      .createQueryBuilder('article')
      .select(['article', 'user.id', 'user.name', 'user.surname', 'user.avatar', 'topic'])
      .leftJoin('article.user', 'user')
      .leftJoin('article.topics', 'topic')
      .leftJoin('article.articleLikes', 'articleLike')
      .loadRelationCountAndMap('article.likesCount', 'article.articleLikes')
  ```

В наведеному вище коді ми створюємо “QueryBuilder” для моделі “Article”, використовуючи “AppDataSource.getRepository(Article)”. За допомогою методу “leftJoin”, встановлюємо зв'язки з моделями “user” і “topic”, та виконуємо підрахунок кількості лайків для кожної статті, використовуючи “loadRelationCountAndMap”.

- ```
  if (searchTerm) {
      queryBuilder.andWhere(`article.title LIKE :searchTerm`, { searchTerm: `%${searchTerm}%` })
  }
  ```

```
if (userId) {
  queryBuilder.andWhere(`article.user.id = :userId`, { userId })
}
if (topicId) {
  queryBuilder.andWhere(`topic.id = :topicId`, { topicId })
}
```

В цій частині ми перевіряємо наявність параметрів фільтрації, таких як “searchTerm”, “userId” та “topicId”, і додаємо відповідні умови до “QueryBuilder”.

- const paginationQueryBuilder = queryBuilder.orderBy(`article.${sortBy}`, order).skip(skip).take(take)

Створюємо новий “QueryBuilder” з врахуванням параметрів пагінації, таких як skip, take та використовуємо “orderBy” для сортування результатів за вказаним полем (“sortBy”) та напрямком (“order”).

- const articles = await paginationQueryBuilder.getMany()

Виконуємо запит до бази даних, використовуючи “getMany()”, щоб отримати статті, які відповідають умовам та параметрам пагінації.

- const totalCount = await queryBuilder.getCount()

Отримуємо загальну кількість статей, які відповідають фільтрам, використовуючи “getCount()”.

- res.status(200).json({ models: articles, totalCount: totalCount })

Відправляємо відповідь зі статтями (“articles”) та загальною кількістю (“totalCount”) у форматі JSON.

В даному прикладі було продемонстровано функціонал взаємодії з моделлю “Article”, що актуальний і для інших моделей також.

3.1.4. Авторизація та реєстрація

Додаток використовує метод авторизації на основі токенів, що є одним із популярних способів забезпечення безпеки та доступу до ресурсів. Основні принципи авторизації такі:

- **Видача токенів.** Після успішної аутентифікації користувача (за допомогою логіна та паролю), сервер видає два типи токенів:
  - *Access Token (токен доступу)* - це короткочасний токен, який містить обмежену інформацію про користувача і

використовується для авторизації та доступу до обмежених ресурсів. Час життя цього токену дуже короткий (30 хвилин).

- **Refresh Token (оновлювальний токен)** - це довготривалий токен, який використовується для оновлення Access Token після закінчення його терміну дії. Час життя цього токену може бути значно довшим.

- **Аутентифікація та авторизація.** Після отримання токенів, клієнт передає Access Token до сервера при кожному запиті на доступ до обмежених ресурсів. Сервер перевіряє дійсність Access Token, декодує його та перевіряє права доступу користувача до ресурсу. Якщо Access Token є дійсним і має необхідні права доступу, сервер надає користувачеві доступ до запитаного ресурсу. Якщо життя токену вичерпано, або він недійсний, сервер відхиляє запит та повертає статус 401 або 403.
- **Оновлення токенів.** Коли у Access Token закінчується термін дії, клієнт використовує Refresh Token для отримання нового Access Token без необхідності повторної аутентифікації. Запит на оновлення токена відправляється до спеціального маршруту сервера (/auth/refresh-token). Сервер перевіряє дійсність Refresh Token, і якщо він є дійсним, видає новий Access Token та оновлює Refresh Token. Нові токени повертаються клієнту для подальшого використання.

Використання Refresh Token допомагає підтримувати безперервний доступ користувача до захищених ресурсів після закінчення терміну дії Access Token та зменшити шкоду у разі його викрадення.

3.1.5. Валідація даних та обробка помилок

Важливою частиною коректної роботи серверної частини є правильна обробка помилок та валідація вхідних даних.

Розглянемо систему обробки реалізовану на основі класів:

- Клас “CustomError” - це абстрактний клас, який розширює стандартний клас “Error”. Він має абстрактну властивість “statusCode”, яка представляє HTTP статус код помилки. Також він має абстрактний метод “serializeErrors()”, який повертає серіалізовані дані про помилку.
- Клас “NotFoundError” - це підклас “CustomError”, який представляє помилку "Не знайдено" (HTTP статус код 404). Він приймає об'єкт помилки у конструкторі і має реалізований метод “serializeErrors()”, який повертає серіалізовані дані про помилку.
- Клас “RequestValidationError” - це підклас “CustomError”, який представляє помилку "Невірні параметри запиту" (HTTP статус код 400). Він приймає об'єкт помилки у конструкторі і має реалізований метод “serializeErrors()”, який повертає серіалізовані дані про помилку.
- Клас “NotAuthorizedError” - це підклас “CustomError”, який представляє помилку "Не авторизовано" (HTTP статус код 401). Він має реалізовані властивості “statusCode” і метод “serializeErrors()”.
- Клас “AccessDeniedError” - це підклас “CustomError”, який представляє помилку "Відмовлено в доступі" (HTTP статус код 403). Він має реалізовані властивості “statusCode” і метод “serializeErrors()”.

```
export const handleErrors = (error: Error, _req: Request, res: Response, next: NextFunction) => {
  console.log(error)
  if (error instanceof CustomError) {
    return res.status(error.statusCode).json(error.serializeErrors())
  }
  res.status(500).json('Internal Server Error')
}
```

Функція “handleErrors” використовується як middleware (проміжне програмне забезпечення) для обробки помилок у маршрутах Express. Вона отримує помилку першим аргументом, перевіряє, чи є вона екземпляром “CustomError”, і повертає відповідь з відповідним HTTP статус кодом та серіалізованими даними про помилку.

```
export const handleNotFound = (_req: Request, res: Response, next: NextFunction) => {
  next(new NotFoundError())
}
```

Функція “handleNotFound” використовується як “middleware” для обробки ситуацій, коли маршрут не знайдений. Вона викликає помилку “NotFoundError” і передає у наступний обробник помилок.

Ця система обробки помилок дозволяє централізовано обробляти помилки і повертати їх у відповідь з відповідним статус кодом. Крім того, вона дозволяє створювати спеціалізовані помилки зі своїми властивостями та поведінкою, що полегшує розробку та налагодження програми.

Тепер розглянемо валідацію вхідних даних на основі схеми створення статті:

```
export const createArticleSchema = Joi.object<CreateArticleRequestBody>({
  title: Joi.string().min(1).max(50).required(),
  text: Joi.string().min(1).max(1000).required(),
  topics: Joi.array().items(Joi.string().uuid()).min(1).max(10).required(),
})
```

В наведеному вище коді створюється схема валідації для об'єкта “CreateArticleRequestBody”, що являє собою вхідні дані запиту. Цей об'єкт має властивості “title”, “text” і “topics”, які підлягають таким правилам валідації:

- “title” має бути рядком (Joi.string()), мінімальною довжиною 1 символ (min(1)), максимальною довжиною 50 символів (max(50)), і є обов'язковим (required()).
- “text” має бути рядком (Joi.string()), мінімальною довжиною 1 символ (min(1)), максимальною довжиною 1000 символів (max(1000)), і є обов'язковим (required()).
- “topics” має бути масивом (Joi.array()), елементи якого є рядками, що відповідають формату UUID (Joi.string().uuid()), мінімальною довжиною 1 елемента (min(1)), максимальною довжиною 10 елементів (max(10)), і є обов'язковим (required()).

Така схема валідації дозволяє перевірити, чи відповідають дані об'єкта “CreateArticleRequestBody” визначеним правилам. Якщо дані не валідні, Joi поверне об'єкт зі списком помилок валідації. Далі помилка надійде до проміжного програмного забезпечення “validation”:

```
export const validation = (schema: Joi.ObjectSchema, entity: 'body' | 'query' | 'params' | undefined = 'body') => {
  return (_req: Request, res: Response, next: NextFunction) => {
    try {
      const { err, value } = schema.validation(_req[entity])
      if (err) {
        throw new RequestValidationError(err)
      }
      _req[entity] = value
      next()
    } catch (error) {
      next(error)
    }
  }
}
```

В свою чергу проміжне програмне забезпечення “validation” створить повідомлення про помилку, використовуючи об’єкт класу “RequestValidationError”.

3.2. Реалізація клієнтської частини

3.2.1. Структура клієнтської частини

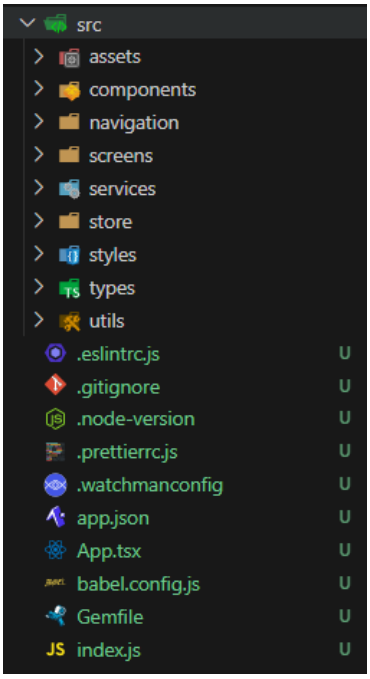

Рисунок 3.3 – Структура клієнтської частини

Розглянемо основні модулі клієнтської частини додатку (рис. 3.3):

- *src/* - це основна директорія проекту, де розміщуються основні функціональні частини програми.
- *components/* - містить повторно використовувані компоненти додатку, такі як кнопки, форми, заголовки тощо.
- *screens/* - вміщує компоненти, які представляють окремі екрани додатку.
- *navigation/* - містить файли, пов'язані з навігацією між екранами.
- *services/* - розташовані файли пов’язані з запитами до серверної частини.
- *utils/* - вміщує допоміжні функції та утиліти, які використані у різних частинах додатку.
- *styles/* - містить файли стилів, які використовуються для оформлення компонентів і екранів.
- *assets/* - вміщує зображення та шрифти які використовуються у додатку.

- *types/* - вміщує глобальні типи додатку.
- *store/* - містить файл створення Redux store та редуктори.
- *App.tsx* - кореневий компонент додатку.
- *index.js* - основний файл додатку, де відбувається ініціалізація та рендер кореневого компонента додатку.

3.2.2. Основні використані бібліотеки

Короткий опис використаних бібліотек в React Native проекті:

- **React Navigation**. Бібліотека надає функцій для навігації між екранами. Вона підтримує різні типи навігації, такі як стек, вкладки, бокове меню тощо. В проекті використано два типи навігації: стек та вкладки.
- **Axios**. Популярна бібліотека для здійснення мережевих запитів в React Native. Вона надає простий та зрозумілий інтерфейс для виконання HTTP-запитів, таких як отримання даних з API та відправка даних на сервер. Axios також підтримує інтерцептори, що дозволяє легко обробляти помилки та додавати заголовки до запитів.
- **date-fns**: Бібліотека для роботи з датами та часом. Вона є альтернативою вбудованому об'єкту Date в JavaScript та надає зручний API для роботи з датами.
- **Redux**. Бібліотека для організації стейт-менеджменту в додатку. Вона пропонує централізований підхід збереження даних, що дозволяє зберігати стан додатку в одному місці і легко керувати ним за допомогою дій (actions) та редукторів (reducers).
- **React Hook Form**. Легка та потужна бібліотека для керування формами, що заснована на використанні хуків. Вона дозволяє ефективно керувати станом форми та зберігати дані.
- **Zod**. Бібліотека для валідації даних в JavaScript та TypeScript. Вона надає декларативний спосіб визначення схем даних і використовується для перевірки правильності даних.

Ці бібліотеки спрощують і пришвидшують розробку та надають готові рішення для навігації, мережевих запитів, роботи з датами та часом, керування станом, валідації та роботи з формами.

3.2.3. Аналіз результатів.

Після завантаження програми, на телефоні з’явиться іконка, що зображена на рисунку 3.4.

Рисунок 3.4 – Іконка програми

Відкривши програму першим нас зустрічає екран авторизації зображений на рисунку 3.5. Екран містить логотип програми, поле для вводу логіну, поле вводу паролю з можливістю побачити введені данні при натисканні на відповідну іконку, кнопку для відправки форми на серверну частину та посилання на екран реєстрації зображеного на рисунку 3.6. Екран реєстрації містить поля для вводу логіну, імені прізвища, паролю та поле для повторного введення паролю.

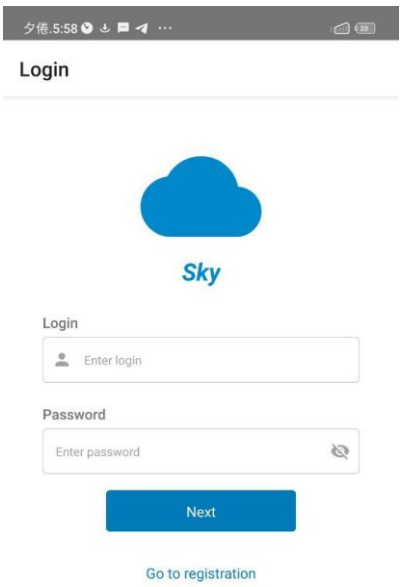

Рисунок 3.5 – Екран авторизації

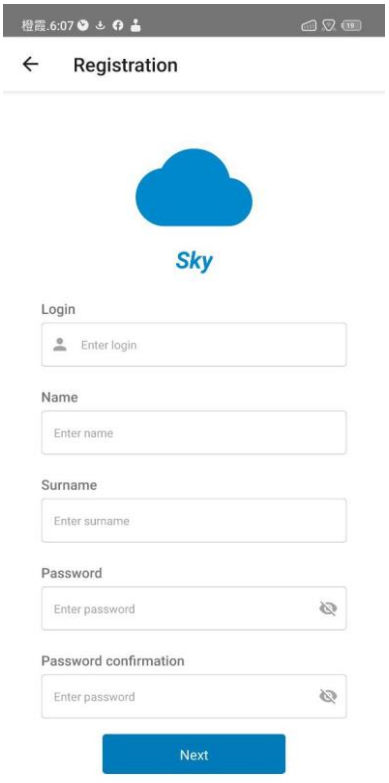

Рисунок 3.6 – Екран реєстрації

Після заповнення полів та натисканні кнопки “Next” дані перевіряються на валідність та користувач має змогу побачити повідомлення про помилку знизу кожного з полів, рисунок – 3.7. У разі виникнення помилки на серверній частині, відповідне повідомлення буде відображено у верхній частині екрану, рисунок – 3.8.

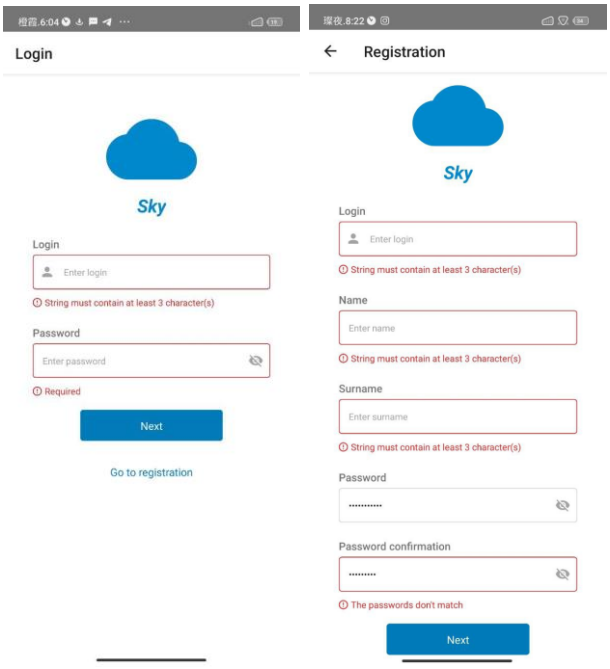

Рисунок 3.7 – Екран авторизації з повідомленнями про помилку

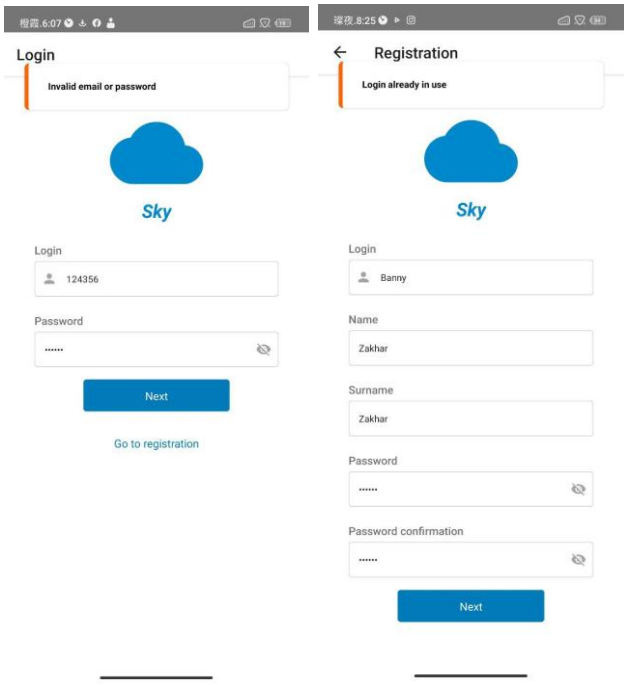

Рисунок 3.8 – Екран авторизації з повідомленням від серверної частини

Після успішної авторизації користувач переходить до головного екрану програми, що зображений на рисунку 3.9, на якому можна побачити наступні елементи:

- нижнє меню програми, що містить три основні розділи: головна сторінка “Home”, розділ з пошуком “Search” та розділ профілю “Profile”;
- списки з публікаціями які розділені верхнім навігаційним меню, що відповідає інтересам користувача;
- кнопку створення нової публікації у блоці із заголовком поточного екрану.

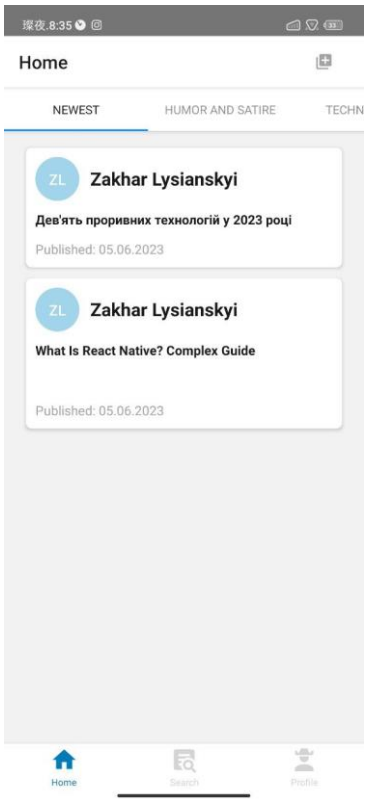

Рисунок 3.9 – Головний екран додатку

На рисунку 3.10 зображений екран створення публікації, що містить наступні елементи:

- поле вводу заголовку;
- кнопка для вибору категорій;
- поле вводу основного тексту публікації;
- панель форматування тексту.

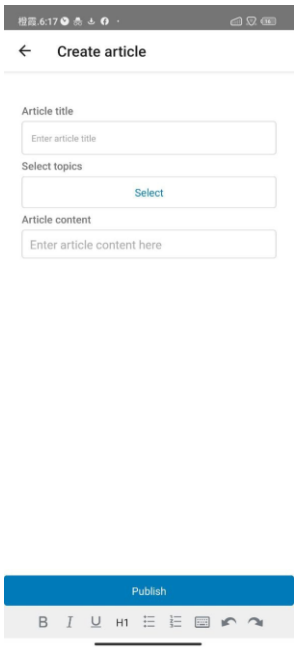

Рисунок 3.10 – екран створення публікації

Після натискання на кнопку “Select” головного екрану створення публікації, відкривається модальне вікно, що містить список доступних категорій для прив’язки до створюваної публікації. Також є можливість пошуку потрібної категорії через поле вводу, рисунок 3.11.

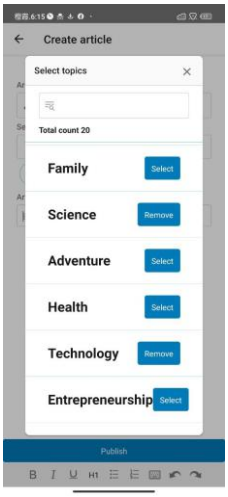

Рисунок 3.11 – Модальне вікно вибору категорій

Після вибору необхідних категорій вони будуть відображені в списку, як показано на рисунку 3.12. Також є можливість видалити обрану категорію натиснувши на іконку праворуч.

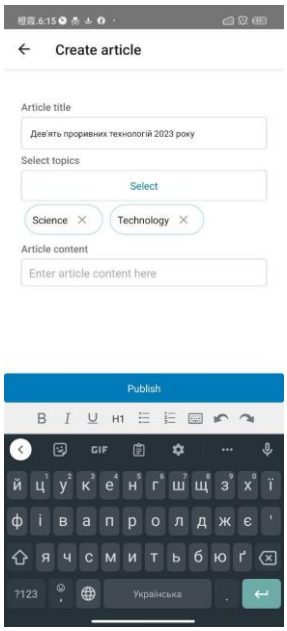

Рисунок 3.12 – Екран створення публікації після вибору категорій

Варто зазначити, що існує можливість форматувати основний текст публікації за допомогою панелі знизу екрана, рисунок - 3.13.

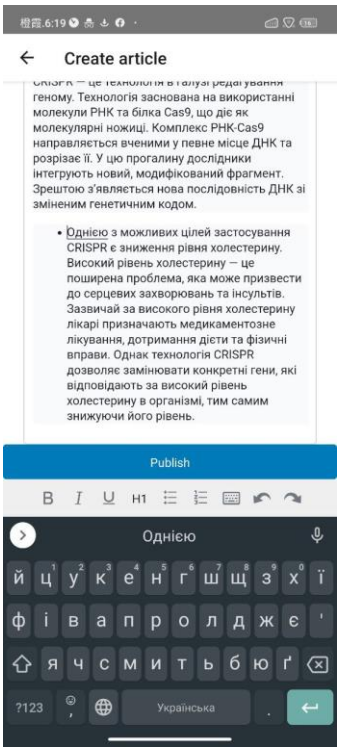

Рисунок 3.13 – Екран створення публікації з відформатованим текстом

Далі розглянемо екран пошуку. Він містить два основних розділи:

- розділ “ARTICLES”, що містить список усіх публікацій з можливістю пошуку по заголовку, рисунок – 3.14;

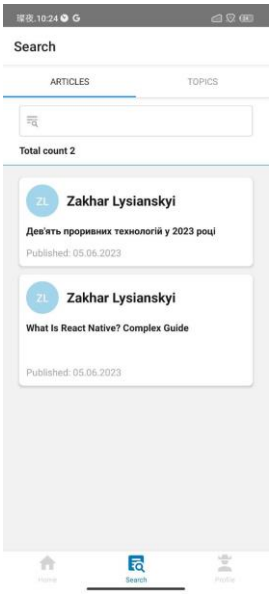

Рисунок 3.14 – Розділ “ARTICLES”

- розділ “TOPICS”, що містить список усіх категорій з можливістю пошуку, вибору конкретної категорії, кнопка “Add”, та видалення категорій з обраних, кнопка “Remove”, рисунок – 3.15.

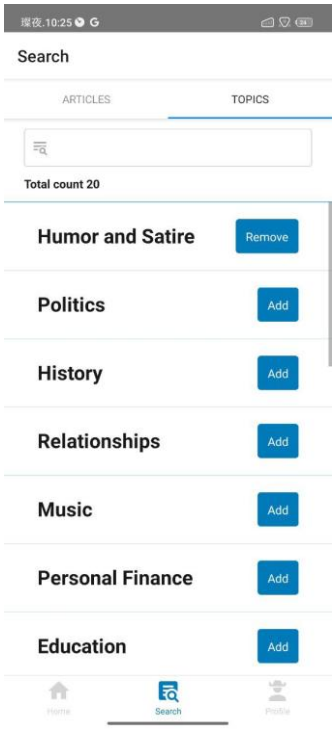

Рисунок 3.15 – Розділ “TOPICS”

Після того як користувач обрав категорію, на головному екрані додатку з’явиться новий розділ, що відповідає назві категорії та відповідні публікації в списку до неї, рисунок - 3.9.

Розглянемо екрани розділу “Profile”.

На рисунку 3.16 зображено екран профілю який містить інформацію про ім’я та прізвище користувача, кнопки “Edit” та “Logout”, розділи: “MY ARTICLES”, “MY TOPICS” та “DESCRIPTION”.

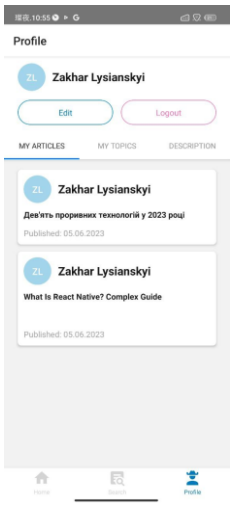

Рисунок 3.16 – Розділ “MY ARTICLES”

В розділі ”MY ARTICLES” користувач може швидко знайти всі свої публікації, які з’являються одразу після створення.

На рисунку 3.17 можна побачити розділ зі списком категорій, що дозволяє користувачу переглядати та видаляти обрані категорії.

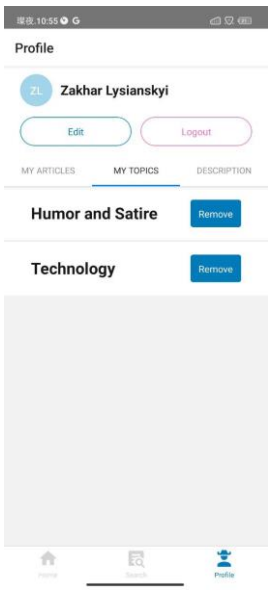

Рисунок 3.17 – Розділ “MY TOPICS”

На рисунку 3.18 зображено розділ з описом користувача, який він може змінювати при редагуванні профілю.

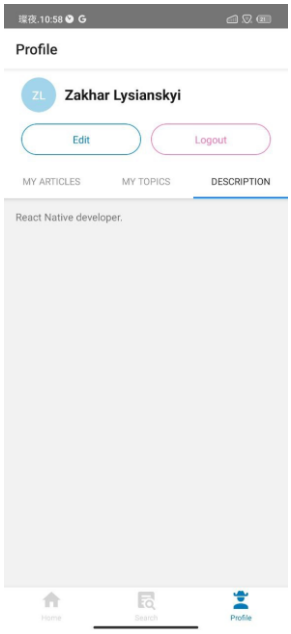

Рисунок 3.18 – Розділ “DESCRIPTION”

Після натискання кнопки “Logout” з’являється вікно підтвердження операції, що допомагає запобігти небажаній операції через випадкове натискання, рисунок – 3.19.

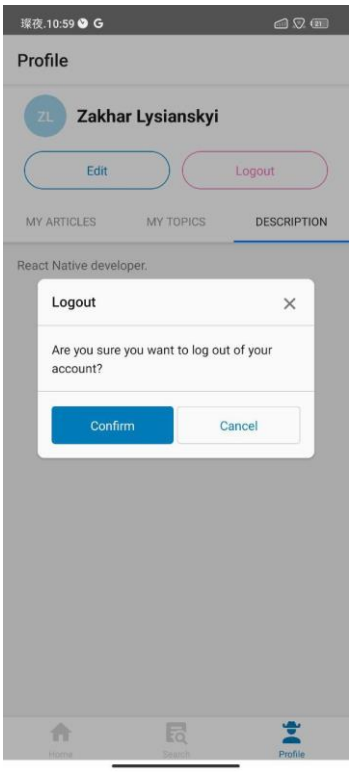

Рисунок 3.19 – Модальне вікно для підтвердження операції виходу

На рисунку 3.20 зображено екран редагування профілю, який дозволяє змінювати відомості про ім’я, прізвище, логін та опис користувача, після успішної зміни даних додаток поверне користувача на екран профілю.

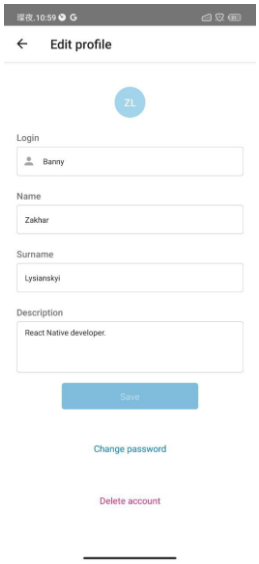

Рисунок 3.20 – Екран редагування профілю.

Також на екрані редагування присутня кнопка видалення профілю “Delete account” при натисканні на яку з’явиться модальне вікно підтвердження операції після чого користувача поверне на екран авторизації, рисунок – 3.21.

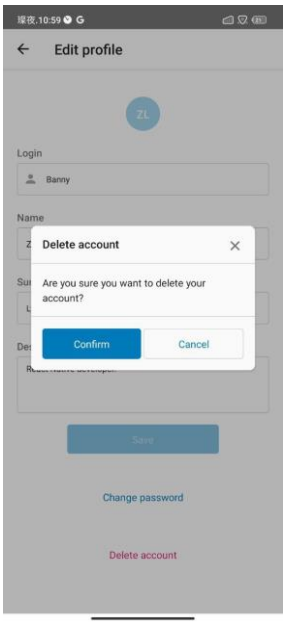

Рисунок 3.21 – Модальне вікно для підтвердження операції видалення

На рисунку 3.22 зображено екран зміни паролю, перехід на який здійснюється через кнопку “Change password” на екрані редагування.

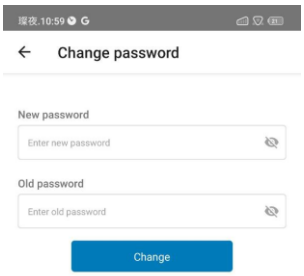

Рисунок 3.22 –Екран зміни паролю

Користувач повинен ввести новий пароль валідний пароль та старий пароль, при зазначенні неправильного старого паролю користувач отримає відповідну помилку, рисунок – 3.23.

Рисунок 3.23 –Екран зміни паролю з відображенням помилки

Розглянемо екран публікації, що зображено на рисунку 3.24. Він містить таку інформацію:

- заголовок;
- основний текст у форматі html, що дає змогу переходити за посиланнями;
- кількість оцінок від користувачів, при натисканні на іконку число змінюється;
- список з категоріями, що прив’язані до публікації;
- 10 останніх коментарів від користувачів;
- кнопку “Write comment”, що відкриває модальне вікно з полем вводу, рисунок – 3.25;
- кнопку “Show more”, яка переносить користувача на екран зі списком всіх коментарів до поточної публікації, рисунок – 3.26.

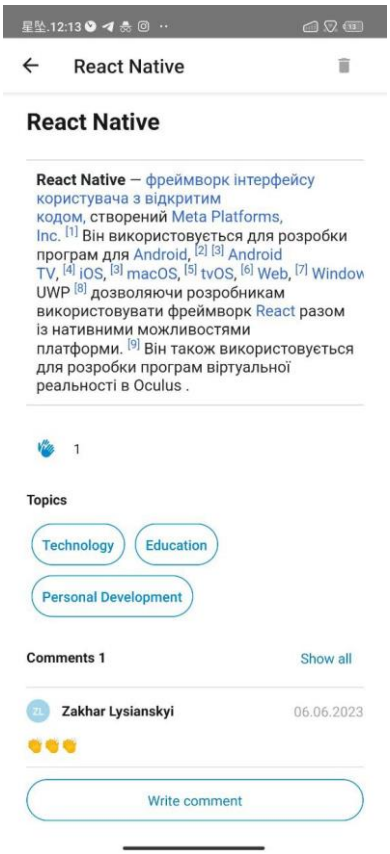

Рисунок 3.24 –Екран публікації

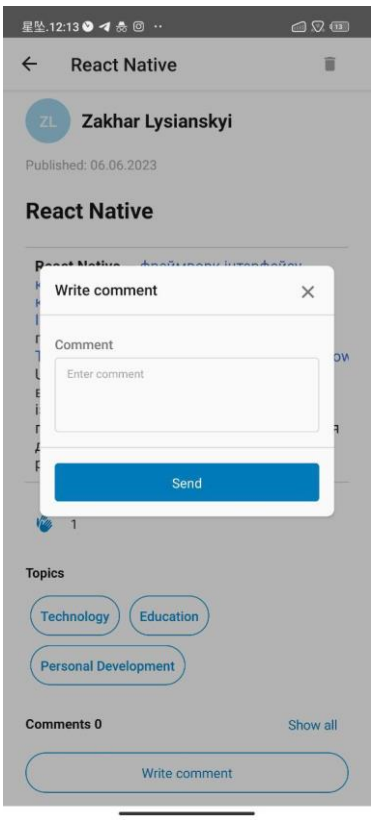

Рисунок 3.25 – Модальне вікно написання коментаря

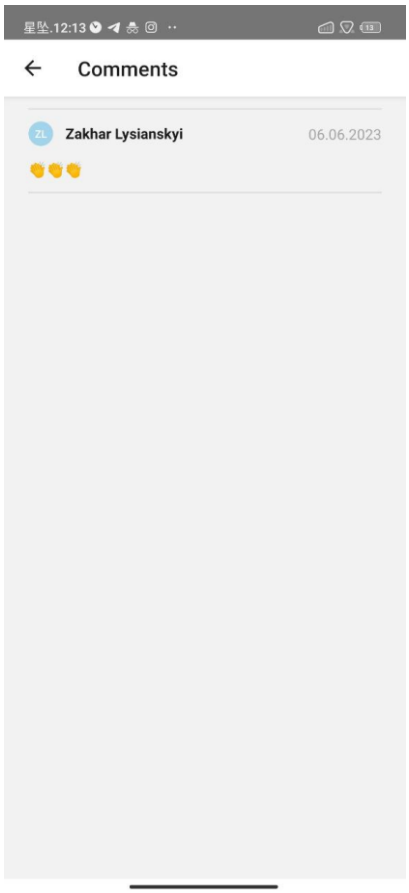

Рисунок 3.26 – Екран з коментарями

Якщо обрана публікація є власністю поточного користувача на екрані публікації буде доступна кнопка видалення в верхньому правому кутку, після натискання буде відкрито модальне вікно з підтвердженням операції, рисунок 3.27.

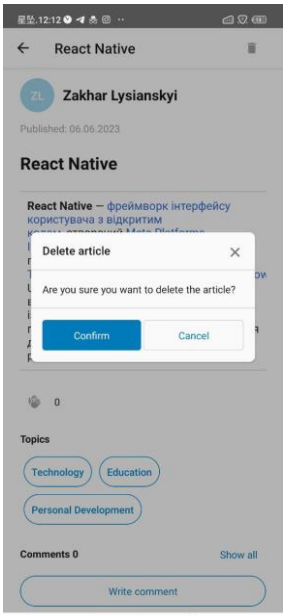

Рисунок 3.27 – Екран з коментарями

Після видалення публікації вона зникне зі списку без можливості відновлення.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи було виконано наступні завдання:

1) виконано огляд аналогів додатку, проаналізовано їх можливості та функціонал;

2) детально описано етапи створення мобільних додатків, які допомагають в розробці власного продукту;

3) визначено існуючі технології для розроби мобільних додатків, розглянуто їх переваги та недоліки;

4) проаналізовано поточний ринок технологій та наведено аргументи щодо обрання крос-платформної розробки;

5) розглянуто концептуальний етап проектування бази даних, визначено сутності зі зв’язками та створено ERD-діаграму;

6) описано СУБД PostgreSQL, що використана для фізичної реалізації бази даних;

7) спираючись на аналіз технологій створення мобільних додатків було обрано найактуальніші інструменти: React Native та Express. Описано їх особливості, переваги та недоліки;

8) описано процес реалізації серверної та клієнтської частин додатків, наведено списки використаних бібліотек, описана структура проектів та деталі реалізації важливих функціональних частин, таких як: обробка помилок, взаємодія з базою даних, процес аутентифікації з використанням токенів, обробка помилок та валідація даних;

9) розглянуто принцип роботи та всі функціональні частини мобільного додатку.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Katruk M. Mobile App Development: Native vs Cross-Platform. URL: https://trm.solutions/archive/2018/09/10/mobile-app-development-native-vs-cross-platform (дата звернення: 10.04.2023).
2. React Native Introduction. URL: .https://reactnative.dev/docs/getting-started
3. Express. Fast, unopinionated, minimalist web framework for Node.js. URL: https://expressjs.com/ (дата звернення: 15.04.2023).
4. NodeJS Documentation. URL: https://nodejs.org/uk/docs/ (дата звернення: 17.04.2023).
5. Pang A. TypeScript vs. JavaScript. URL: https://medium.com/geekculture/typescript-vs-javascript-e5af7ab5a331 (дата звернення: 19.04.2023).
6. Cordenne B. 15 Examples of Successful Companies Using React Native in 2022. URL: https://www.trio.dev/blog/companies-use-react-native (дата звернення: 23.04.2023).
7. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 28.04.2023).
8. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 28.04.2023).
9. Joi Documentation. URL: https://joi.dev/api/?v=17.9.1 (дата звернення: 05.05.2023).
10. TypeORM Documentation. URL: https://typeorm.io/ (дата звернення: 12.05.2023).
11. Native vs Hybrid vs Cross Platform - Best option for Mobile App. URL: https://citrusbug.com/blog/native-vs-hybrid-vs-cross-platform-mobile-application (дата звернення: 12.04.2023).
12. TypeORM Documentation. URL: https://typeorm.io/ (дата звернення: 13.05.2023).

13. React Navigation Documentation. URL: https://reactnavigation.org/docs/getting-started (дата звернення: 15.05.2023).
14. Material Design Icons. URL: https://pictogrammers.com/library/mdi/ (дата звернення: 15.05.2023).
15. NativeBase Documentation. URL: https://docs.nativebase.io/?utm_source=HomePage&utm_medium=header&utm_campaign=NativeBase_3 (дата звернення: 16.05.2023).

ДОДАТОК А

Моделі сутностей TypeORM

article.entity.ts

```
@Entity()
export class Article {
  @PrimaryGeneratedColumn('uuid')
  id: string

  @Column({ length: 50 })
  title: string

  @Column({ length: 100000 })
  text: string

  @CreateDateColumn()
  createdAt: Date

  @UpdateDateColumn()
  updatedAt: Date

  @OneToMany(() => Comment, (comment) => comment.article, { cascade: true })
  comments: Comment[]

  @OneToMany(() => ArticleLike, (articleLike) => articleLike.article, { cascade: true })
  articleLikes: ArticleLike[]

  // many - one
  @ManyToOne(() => User, (user) => user.articles, { onDelete: 'CASCADE' })
  @JoinColumn({ name: 'userId' })
  user: User

  // many - many
  @ManyToMany(() => Topic, (topic) => topic.articles, { onDelete: 'CASCADE' })
  @JoinTable({ name: 'article_topics_list' })
  topics: Topic[]

  @ManyToMany(() => User, (user) => user.savedArticles)
  savedByUsers: User[]
}
```

articleLike.entity.ts

```
@Entity()
export class ArticleLike {
  @PrimaryGeneratedColumn('uuid')
  id: string

  @ManyToOne(() => User, (user) => user.articleLikes)
  @JoinColumn({ name: 'userId' })
  user: User

  @ManyToOne(() => Article, (article) => article.articleLikes, { onDelete: 'CASCADE' })
  @JoinColumn({ name: 'articleId' })
  article: Article
}
```

comment.entity.ts

```
@Entity()
export class Comment {
  @PrimaryGeneratedColumn('uuid')
  id: string

  @Column({ length: 500 })
  content: string

  @ManyToOne(() => Article, (article) => article.comments)
  @JoinColumn({ name: 'articleId' })
  article: Article

  @ManyToOne(() => User, (user) => user.comments)
  @JoinColumn({ name: 'userId' })
  user: User

  @CreateDateColumn()
  createdAt: Date

  @UpdateDateColumn()
  updatedAt: Date

  @OneToMany(() => CommentLike, (commentLike) => commentLike.comment)
  commentLikes: CommentLike[]
}
```

refreshToken.entity.ts

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn } from 'typeorm'
import { User } from './user.entity'

@Entity()
export class RefreshToken {
  @PrimaryGeneratedColumn('uuid')
  id: string

  @Column()
  token: string

  @ManyToOne(() => User, (user) => user.refreshTokens, { onDelete: 'CASCADE' })
  @JoinColumn({ name: 'userId' })
  user: User
}
```

topic.entity.ts

```
@Entity()
export class Topic {
  @PrimaryGeneratedColumn('uuid')
  id: string

  @Column({ length: 50 })
  title: string

  @CreateDateColumn()
  createdAt: Date

  @UpdateDateColumn()
  updatedAt: Date

  @ManyToMany(() => Article, (article) => article.topics, { onDelete: 'CASCADE' })
  articles: Article[]

  @ManyToMany(() => User, (user) => user.topics, { onDelete: 'CASCADE' })
  users: User[]
}
```

user.entity.ts

```
@Entity()
export class User {
 @PrimaryGeneratedColumn('uuid')
 id: string

 @Column({ length: 16 })
 login: string

 @Column()
 password: string

 @Column({ length: 50 })
 name: string

 @Column({ length: 50 })
 surname: string

 @Column({ nullable: true })
 avatar: string

 @Column({ nullable: true, length: 255 })
 description: string

 @CreateDateColumn()
 createdAt: Date

 @UpdateDateColumn()
 updatedAt: Date

 @OneToMany(() => RefreshToken, (refreshToken) => refreshToken.user, { cascade: true })
 refreshTokens: RefreshToken[]

 @OneToMany(() => Article, (article) => article.user, { cascade: true })
 articles: Article[]

 @OneToMany(() => ArticleLike, (articleLike) => articleLike.user)
 articleLikes: ArticleLike[]

 @OneToMany(() => CommentLike, (commentLike) => commentLike.user)
 commentLikes: CommentLike[]
```

```
    @OneToMany(() => Comment, (comment) => comment.user)
    comments: Comment[]

    @ManyToMany(() => Topic, (topic) => topic.users, { onDelete: 'CASCADE' })
    @JoinTable({ name: 'user_topics_list' })
    topics: Topic[]

    @ManyToMany(() => User, (user) => user.followers)
     @JoinTable({ name: 'user_followers_list', joinColumn: { name: 'userId' }, inverseJoinColumn: { name:
'followerId' } })
    following: User[]

    @ManyToMany(() => User, (user) => user.following)
    followers: User[]

    @ManyToMany(() => Article, (article) => article.savedByUsers)
    @JoinTable({ name: 'user_articles_list' })
    savedArticles: Article[]
  }
```

ДОДАТОК Б

Обробка помилок та авторизація

```
import axios from 'axios'
import { DEFAULT_API } from '@env'
import { AuthService } from '@app/services/auth'

const privateInstance = axios.create({
  baseURL: DEFAULT_API,
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
})

// Request interceptor for API calls
privateInstance.interceptors.request.use(
  async (config) => {
    const accessToken = await AuthService.getAccessToken()

    if (accessToken && config.headers) {
      config.headers.Authorization = 'Bearer ' + accessToken
    }

    return config
  },
  (error) => {
    Promise.reject(error)
  },
)

// Response interceptor for API calls
privateInstance.interceptors.response.use(
  (response) => {
    return response
  },
  async function (error) {
    const originalRequest = error.config

    if (error?.response?.status === 401 && !originalRequest._retry) {
      originalRequest._retry = true
```

```
      const accessToken = await AuthService.refreshToken()

      if (!accessToken) {
        return Promise.reject(error)
      }

      originalRequest.headers.Authorization = 'Bearer ' + accessToken

      return privateInstance(originalRequest)
    } else if (originalRequest._retry) {
      await AuthService.logout()
    }

    return Promise.reject(error)
  },
)

export const apiPrivate = privateInstance
export const apiPublic = axios.create({ baseURL: DEFAULT_API })
```

ДОДАТОК В

Сервіс авторизації

```
class AuthService {
  private static refreshKey = 'refreshToken'
  private static accessKey = 'accessToken'

  public async getAccessToken(): Promise<string | null> {
    try {
      const accessToken = await Keychain.getGenericPassword({
        service: AuthService.accessKey,
      })

      if (!accessToken) {
        return null
      }

      return accessToken.password
    } catch {
      await this.logout()
      return null
    }
  }

  public async refreshToken(): Promise<string | null> {
    try {
      const refreshToken = await Keychain.getGenericPassword({
        service: AuthService.refreshKey,
      })

      if (!refreshToken) {
        return null
      }

      const { data }: AxiosResponse<TTokens> = await apiPublic.post(
        `/auth/refresh-token`,
        {
          refreshToken: refreshToken.password,
        },
      )

      await this.setTokens(data)
```

```
      return data.accessToken
    } catch {
      await this.logout()
      return null
    }
  }

  public async setTokens({ refreshToken, accessToken }: TTokens) {
    try {
      await Keychain.setGenericPassword(AuthService.refreshKey, refreshToken, {
        service: AuthService.refreshKey,
      })
      await Keychain.setGenericPassword(AuthService.accessKey, accessToken, {
        service: AuthService.accessKey,
      })
    } catch {
      await this.logout()
    }
  }

  public async logout(): Promise<boolean> {
    try {
      await Keychain.resetGenericPassword({
        service: AuthService.refreshKey,
      })
      await Keychain.resetGenericPassword({
        service: AuthService.accessKey,
      })
      store.dispatch(userActions.logoutAction())
      return true
    } catch {
      return false
    }
  }
}

export default new AuthService()
```

ДОДАТОК Г

Головний компонент навігації

```
export const RootNavigation = () => {
  const logined = !!useTypedSelector((store) => store.userState.user?.id)
  const { isLoading, isFetching } = useGetProfileQuery(null)

  if (isLoading || isFetching) {
    return <LoadingLayout />
  }

  return (
    <NavigationContainer>
      {logined ? <MainBottomTabs /> : <AuthStack />}
    </NavigationContainer>
  )
```