НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

“До захисту допущено”
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА

“___”_____2024 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
“Цифрові технології в енергетиці”
зі спеціальності *122 “Комп’ютерні науки”*

на тему: **Мобільний застосунок з функціями ділового органайзера**

Виконав (-ла): Лизогуб Анастасія
студент (-ка) IV курсу, групи ТР-03

ЛИЗОГУБ Анастасія Олександрівна _____
(прізвище, ім’я, по батькові) (підпис)

Керівник: *доцент каф. цифрових технологій в енергетиці*
доцент, к.т.н.,Володимир ТИХОХОД _____
(посада, вчене звання ,науковий ступінь, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент: *доцент каф. інженерії програмного забезпечення в енергетиці,*
доцент, к.т.н., _____
(посада, вчене звання, науковий ступінь, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль: *асистент Микита ГОЛОВАКІН* _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____
(підпис)

Київ – 2024

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА

«__»_____2024 р.

ЗАВДАННЯ
на дипломну роботу студенту

ЛИЗОГУБ Анастасії Олександрівні
(прізвище, ім’я, по батькові)

1. Тема роботи **Мобільний застосунок з функціями ділового органайзера**

керівник роботи ***Тихоход Володимир Олександрович, к.т.н., доцент***
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «___»_____ 202__ р. №_____

2. Термін подання роботи студентом **07.06.2024 року**

3. Вихідні дані до роботи мова програмування JavaScript, фреймворки React Native та NestJs, СКБД PostgreSQL, середовище розробки WebStorm інструмент для тестування Postman

4. Перелік питань, які потрібно розробити
1) провести аналіз серед наявних аналогів
2) веб-систем для надання та пошуку наставницьких послуг в сфері ІТ
3) аргументувати вибрані інструменти, технології та алгоритми для реалізації програмного забезпечення
4) побудувати архітектури програмного забезпечення, баз даних та сховища файлів

5) створити прикладний програмний веб-інтерфейс
6) представити процес застосування веб-інтерфейсу

5. Орієнтований перелік ілюстративного матеріалу діаграми, що демонструють роботу алгоритмів веб-інтерфейсу, архітектуру системи, бази даних та сховища файлів, взаємодію користувачів із системою, класи програмного забезпечення; приклади роботи з програмним продуктом.

6. Дата видачі завдання «15» вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми роботи		
2.	Вивчення та аналіз задачі	17-20.04.24	
3.	Розробка архітектури та загальної структури системи	21-25.04.24	
4.	Розробка частин окремих підсистем	25.04-02.05.24	
5.	Програмна реалізація системи	03-07.05.24	
6.	Оформлення пояснювальної записки	08-12.05.24	
7.	Захист програмного продукту	13-17.05.24	
8.	Передзахист	03-06.06.24	
9.	Подання готової роботи на кафедру	07.06.2024	
10.	Захист	17-21.06.2024	

Студент _____ **Анастасія ЛИЗОГУБ**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ **Володимир ТИХОХОД**
(підпис) (ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана на 65 сторінках, містить 12 ілюстрацій, 16 джерел в переліку посилань та 2 таблиці.

Мета роботи – створення мобільного застосунку з функціями ділового органайзера, який допомагає користувачам ефективно керувати своїми завданнями, подіями та часом.

Методи та засоби: робота виконується з використанням мови програмування JavaScript та допоміжних фреймворків NestJs та React Native. База даних PostgreSQL використовується для зберігання інформації про користувачів, завдання та події..

Результат – мобільний застосунок з функціями ділового органайзера, що дозволяє користувачам ефективно керувати своїми справами, підвищувати продуктивність та забезпечувати зручний доступ до інформації будь-де та будь-коли.

Ключові слова: МОБІЛЬНИЙ ЗАСТОСУНОК, МОБІЛЬНИЙ ЗАСТОСУНОК, ДІЛОВИЙ ОРГАНАЙЗЕР, ТРЕКЕР СПРАВ, JAVASCRIPT, REACT NATIVE, NESTJS, POSTGRESQL.

ABSTRACT

The thesis is completed on 65 pages, contains 12 illustrations, and includes 16 references in the bibliography and 2 tables.

The aim of the work is to create a mobile application with business organizer functions, which helps users efficiently manage their tasks, events, and time.

Methods and tools: the work is performed using the JavaScript programming language and auxiliary frameworks such as NestJs and React Native. The PostgreSQL database is used to store information about users, tasks, and events.

The result is a mobile application with business organizer functions that allows users to effectively manage their tasks, increase productivity, and ensure convenient access to information anytime and anywhere.

Keywords: MOBILE APPLICATION, BUSINESS ORGANIZER, TASK TRACKER, JAVASCRIPT, REACT NATIVE, NESTJS, POSTGRESQL.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ - програмне забезпечення

БД – база даних

СУБД - система управління базами даних

ВСТУП

Сучасний бізнес вимагає високої мобільності та ефективності у керуванні щоденними завданнями та процесами. Особливо це стосується підприємств, які прагнуть досягти високої продуктивності та оптимізувати робочий час своїх працівників. У цьому контексті мобільні застосунки з функціями ділового органайзера стають важливим інструментом, що дозволяє підприємцям та працівникам ефективніше планувати свій робочий час, відслідковувати виконання завдань і координувати дії.

Основною метою розробки мобільного застосунку з функціями ділового органайзера стало створення інструменту, який би значно спрощував і оптимізував управління бізнес-процесами. Застосунок покликаний забезпечити зручний та інтуїтивно зрозумілий інтерфейс, який дозволить користувачам легко взаємодіяти з ним, витрачаючи мінімум часу на освоєння. Завдяки гнучким налаштуванням та можливостям персоналізації, застосунок може бути адаптований під специфічні потреби різних підприємств, що робить його універсальним рішенням для широкого кола користувачів.

Є ряд програмних інструментів, що забезпечують таку підтримку, як зокрема, Trello, Asana, Microsoft To-Do, але їх суттєвим недоліком є обмежена мобільна функціональність або відсутність специфічних функцій, необхідних для певних бізнес-процесів. Мобільний застосунок з функціями ділового органайзера надає можливість завжди мати під рукою необхідні інструменти для управління завданнями, зустрічами та проєктами. Такий застосунок дозволяє користувачам створювати, редагувати та відслідковувати завдання, встановлювати пріоритети та дедлайни, а також отримувати нагадування про важливі події. Це значно підвищує продуктивність і допомагає уникнути хаосу в роботі.

Таким чином, мобільний застосунок з функціями ділового органайзера не лише підвищує ефективність управління бізнесом, але й сприяє зростанню продуктивності працівників, допомагаючи їм краще організувати свій робочий час та зосередитись на виконанні найважливіших завдань. В результаті, підприємства можуть досягти більш високих результатів, зменшити кількість помилок і забезпечити своєчасне виконання завдань, що є ключовим фактором для успішного ведення бізнесу в сучасних умовах конкуренції.

1 РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ З ФУНКЦІЯМИ ДІЛОВОГО ОРГАНАЙЗЕРА

У сучасному світі, де інформаційні технології відіграють ключову роль у повсякденному житті, зростає потреба у засобах, які допомагають людям організовувати та оптимізувати свої завдання та час. Мобільні застосунки з функціями ділових органайзерів є важливим інструментом для підвищення продуктивності та ефективного управління справами. Вони дозволяють користувачам легко планувати свої робочі та особисті завдання, встановлювати нагадування та отримувати сповіщення про важливі події.

Цей розділ присвячений розробці мобільного застосунку з функціями ділового органайзера, який забезпечить користувачам зручний та функціональний інструмент для управління своїм часом та завданнями. У розділі будуть розглянуті основні аспекти проектування та реалізації застосунку, включаючи визначення функціональних вимог, обрання технологій та інструментів, архітектурні рішення, а також методи забезпечення безпеки та продуктивності системи.

Зокрема, розділ охоплює постановку задачі, аналіз існуючих рішень, деталі розробки серверної та клієнтської частин, а також алгоритми, що забезпечують коректну роботу застосунку. Особлива увага приділяється питанням інтеграції та тестування, які є критично важливими для забезпечення надійної та безперебійної роботи програмного продукту.

У результаті розробки буде створено мобільний застосунок, що допоможе користувачам ефективно керувати своїми справами, підвищити продуктивність та забезпечити зручний доступ до необхідної інформації у будь-який час та з будь-якого місця.

1.1 Загальні відомості з предметної галузі

У наш час мобільними додатками кожен день користуються мільйони користувачів. Першим мобільним застосунком можна вважати прості програми, що постачалися з ранніми мобільними телефонами, такі як телефонна книга, календар та будильник. Однак, якщо розглядати більш складні та розширені мобільні застосунки, то першим офіційним і загальновизнаним мобільним застосунком вважається гра "Snake" (Змійка), яка була включена до мобільних телефонів Nokia у 1997 році. Важко повірити але лише за 27 років мобільні додатки сильно вкоренились у наше життя.

З появою смартфонів на початку 2000-х років, а зокрема з виходом Apple iPhone у 2007 році та запуском App Store у 2008 році, світ мобільних застосунків суттєво змінився. App Store став першою платформою, де користувачі могли легко завантажувати та встановлювати різні мобільні застосунки, що дало поштовх для масового розвитку індустрії мобільних застосунків.

1.2 Постановка задачі

Метою даної дипломної роботи є розробка мобільного застосунку з функціями ділового органайзера, який забезпечить ефективне управління завданнями, подіями та контактами для користувачів. Для досягнення цієї мети необхідно вирішити такі завдання:

1.2.1 Визначення функціональних вимог

Для досягнення мети розробки мобільного застосунку з функціями ділового органайзера, необхідно розробити функціонал для створення, редагування та видалення завдань і подій. Важливо забезпечити можливість налаштування нагадувань для важливих подій, що дозволить користувачам бути в курсі

запланованих заходів. Інтеграція календаря з можливістю перегляду щоденних, тижневих та місячних планів забезпечить зручність планування та відстеження завдань. Крім того, додаток має підтримувати функцію пошуку по всім планам, що дозволить швидко знаходити необхідну інформацію, а також сортування планів, що сприятиме ефективній організації робочого часу.

1.2.2 Користувацькі вимоги

Інтерфейс користувача повинен бути інтуїтивно зрозумілим та зручним, щоб забезпечити легкість у використанні застосунку. Необхідно також передбачити можливість індивідуалізації налаштувань відповідно до потреб кожного користувача, що дозволить адаптувати застосунок під особисті вподобання та робочі процеси. Реалізація адаптивного дизайну є обов'язковою умовою, оскільки це забезпечить коректну роботу застосунку на різних мобільних пристроях з різними розмірами екрану, що в свою чергу підвищить зручність і доступність використання застосунку для широкого кола користувачів.

1.2.3 Технічні вимоги

Необхідно визначити технології та інструменти, які будуть використані для розробки застосунку, зокрема мову програмування, фреймворки та базу даних. Забезпечення високої продуктивності та стабільності роботи застосунку є ключовим аспектом, що передбачає оптимізацію коду та використання ефективних алгоритмів. Також критично важливо гарантувати безпеку даних користувачів шляхом впровадження сучасних методів шифрування та захисту даних, що забезпечить конфіденційність та цілісність інформації, яка зберігається і обробляється застосунком.

1.2.4 Порівняльний аналіз

Проведення аналізу існуючих на ринку мобільних органайзерів є важливим етапом, що дозволяє оцінити сучасні тенденції та визначити найкращі практики в цій сфері та уникнути вже існуючих недоліків з інших програм. Необхідно визначити переваги та недоліки конкурентних продуктів, що допоможе зрозуміти, які аспекти потребують покращення та які функції слід додати. На основі цього аналізу буде розроблено унікальні функціональні можливості, які відрізнятимуть наш застосунок від інших і забезпечать його конкурентоспроможність.

1.2.5 Інтеграція та тестування

Інтеграція всіх функціональних компонентів застосунку є важливим етапом розробки, що забезпечує їх узгоджену роботу та взаємодію. Для цього необхідно розробити детальний тестовий план, який включатиме перевірку всіх функціональних можливостей застосунку. Після цього потрібно виконати тестування застосунку на різних пристроях та в різних умовах, щоб гарантувати його стабільну роботу та відповідність заданим вимогам.

1.2.6 Документація та підтримка

Для забезпечення ефективного використання мобільного застосунку необхідно створити детальну документацію, яка буде корисною як для користувачів, так і для розробників. Крім того, важливо передбачити механізми підтримки та оновлення застосунку, щоб забезпечити його тривалу функціональність і відповідність сучасним вимогам та стандартам.

Таким чином, постановка задачі охоплює весь процес розробки мобільного застосунку з функціями ділового органайзера, починаючи від визначення вимог та закінчуючи тестуванням та документуванням. Це забезпечить створення якісного та конкурентоспроможного продукту, який задовольнить потреби користувачів у ефективному управлінні своїми діловими справами.

1.3 Огляд методів

Мобільний додаток складається з двох основних частин: frontend та backend частин. Розглянемо їх детальніше.

1.3.1 Frontend частина мобільного застосунку

Frontend або клієнтська частина відповідає за взаємодію користувача з програмою. Для розробки було обрано мову програмування JavaScript, а саме фреймворк бібліотеки React – React Native з використанням Expo CLI.

React Native дозволяє розробникам створювати мобільні застосунки для iOS та Android, використовуючи той самий кодовий базис. Це значно скорочує час та ресурси на розробку, а також полегшує підтримку застосунку в майбутньому.[13]

У наступних підпунктах будуть наведені компоненти Frontend частини.

1.3.1.1 Інтерфейс користувача

Для забезпечення зручної навігації в мобільному застосунку використовується бібліотека React Navigation, яка дозволяє створювати багаторівневу навігацію між екранами. Це включає такі види навігації, як stack-навігація, таб-навігація та інші, що забезпечують легкість і інтуїтивність користування. Для вводу даних застосунок містить різноманітні компоненти, такі як поля для вводу тексту, вибору дат, перемикачі та інші форми вводу. Використання бібліотек Formik та Yup дозволяє ефективно управляти формами та здійснювати їх валідацію.[14]

Для відображення списку завдань та подій у застосунку використовуються спеціальні компоненти, які забезпечують можливість додавання, редагування та видалення елементів. Бібліотека FlatList дозволяє ефективно рендерити великі списки, що покращує продуктивність та зручність використання. Крім того, для реалізації

календаря з можливістю перегляду щоденних, тижневих та місячних планів інтегровано бібліотеку React Native Calendars, яка забезпечує необхідний функціонал та зручність у користуванні календарем.[13]

1.3.1.2 Дизайн та стилізація

Для забезпечення адаптивного дизайну використовується Flexbox, що дозволяє створювати макет, який буде коректно відображатися на різних пристроях із різними розмірами екранів. Це забезпечує універсальність і зручність використання застосунку незалежно від типу пристрою. Для створення консистентного вигляду застосунку використовуються Styled Components або StyleSheet, які дозволяють легко керувати стилями та підтримувати єдиний стильовий підхід.

1.3.1.3 Робота з API

У мобільних додатках, особливо якщо вони взаємодіють з сервером для обміну даними, використання бібліотеки Axios є стандартним підходом для здійснення HTTP-запитів. Axios надає зручний і простий інтерфейс для відправлення різних видів запитів (GET, POST, PUT, DELETE тощо) до сервера.

У застосунку реалізовано функції, що використовують Axios для взаємодії з сервером. Ці функції відповідають за різні операції з даними, такі як отримання списків, додавання нових записів, оновлення існуючих та видалення записів. Наприклад, функція для отримання списків може викликати HTTP-запит типу GET, який поверне перелік елементів з сервера.

Однак у процесі взаємодії з сервером можуть виникати помилки, такі як втрата з'єднання, недоступність сервера або некоректні дані. Для обробки цих помилок у застосунку впроваджено відповідні механізми. При виникненні помилки Axios автоматично генерує виняток, який можна перехопити і обробити. В результаті,

користувач отримує інформативне повідомлення про помилку, що дозволяє йому зрозуміти, що сталося, і при необхідності вжити відповідних заходів для вирішення проблеми.

1.3.1.4 Управління станом

Управління станом є ключовою частиною будь-якого мобільного додатка, оскільки дозволяє ефективно керувати даними, які використовуються у різних частинах програми і взаємодіють між собою. У застосунку використовується підхід до управління станом, який базується на використанні Context API або бібліотеки Redux. Це дозволяє зручно та ефективно керувати глобальним станом додатка і забезпечити консистентність даних між різними компонентами. Наприклад, коли дані змінюються в одному компоненті, ці зміни автоматично відображаються в інших компонентах, які використовують ці дані.

Крім того, для збереження налаштувань користувача та інших тимчасових даних використовується механізм локального збереження, такий як AsyncStorage. Це дозволяє зберігати дані на пристрої користувача і забезпечити доступ до них при наступних запусках додатка, що зручно для користувачів.

Для взаємодії зі станом компонентів використовуються глибокі знання функціональних хуків React, таких як useState та useEffect. useState використовується для зберігання та оновлення стану в компонентах, тоді як useEffect використовується для виконання певних дій в компоненті при зміні стану або після його оновлення. Такий підхід дозволяє ефективно реагувати на зміни стану та виконувати потрібні дії у відповідь на них.[13]

1.3.2.5 Інтеграція з Expo

Інтеграція з Expo спрощує процес розробки мобільних додатків, надаючи зручний інструментарій для створення та управління проектом. Використання Expo CLI дозволяє швидко налаштувати середовище для розробки і тестування застосунку. Бібліотека Expo також забезпечує доступ до різноманітних модулів, які дозволяють взаємодіяти з нативними функціями пристрою, такими як камера, локація, push-повідомлення та інші.

За допомогою Expo, frontend частина мобільного застосунку створена на основі React Native, що забезпечує інтуїтивно зрозумілий та зручний інтерфейс для користувачів. Крім того, додаток підтримує всі необхідні функціональні можливості для управління діловими завданнями та подіями, що робить його ефективним і зручним для використання.

1.3.2 Backend частина мобільного застосунку

Backend частина мобільного застосунку відповідає за обробку даних, логіку додатку та взаємодію з базою даних. Для розробки backend частини було обрано фреймворк NestJS, базу даних PostgreSQL та контейнеризацію за допомогою Docker.

У наступних підпунктах будуть наведені компоненти Backend частини.

1.3.2.1 NestJS

NestJS використовує архітектуру модулів, що дозволяє легко організувати програму на логічні блоки. Це забезпечує чітку та зрозумілу структуру проекту, використовуючи модулі, контролери та сервіси.[5]

Контролери відповідають за обробку HTTP-запитів від фронтенд частини. Вони приймають та обробляють запити на створення, читання, оновлення та видалення (CRUD) даних, забезпечуючи взаємодію між клієнтом та сервером.[5]

Основна бізнес-логіка реалізована у сервісах. Вони взаємодіють з базою даних та іншими модулями, виконуючи основні операції з даними та повертаючи результати до контролерів. Це дозволяє окремо визначати та керувати бізнес-процесами, що полегшує розробку та підтримку проекту.[5]

1.3.2.2 PostgreSQL

Для зберігання даних користувачів, завдань, подій та контактів використовується PostgreSQL як основна реляційна база даних. PostgreSQL відома своєю високою надійністю та продуктивністю при роботі з великими обсягами даних, що робить її відмінним вибором для потреб нашого застосунку.

Для взаємодії з базою даних використовується бібліотека TypeORM, яка забезпечує зручний спосіб роботи з даними за допомогою об'єктно-реляційного відображення (ORM). Це дозволяє нам працювати з базою даних без прямого використання SQL-запитів, що спрощує розробку та підтримку коду, а також підвищує його зручність.

Крім того, реалізована система міграцій, яка дозволяє керувати схемою бази даних та внесенням змін у структуру таблиць. Це забезпечує зручний та контрольований процес внесення змін до бази даних, що сприяє її стабільності та зручності управління.

1.3.2.3 Docker

Для забезпечення ефективного розгортання та масштабування backend частини застосунку використовується Docker і Docker Compose. Docker дозволяє контейнеризувати backend додаток, створюючи ізольоване середовище для його виконання. Це спрощує процес розгортання та забезпечує консистентність середовища між різними стадіями розробки та виробництва.

Docker Compose використовується для управління багатоконтейнерними додатками, що дозволяє одночасно запускати backend сервер, базу даних та інші сервіси у вигляді окремих контейнерів. Це забезпечує зручний та контрольований спосіб запуску всіх компонентів застосунку.

Однією з ключових переваг Docker є безпека та ізоляція. Docker забезпечує ізольоване середовище виконання додатку, що підвищує безпеку та стабільність роботи, оскільки кожен контейнер має своє власне середовище та ресурси, і не впливає на інші контейнери.

1.3.2.4 API та взаємодія з frontend

Backend частина застосунку надає RESTful API для взаємодії з frontend. REST API дозволяє frontend частині звертатися до різних функціональних можливостей застосунку шляхом стандартних HTTP-запитів, таких як GET, POST, PUT та DELETE.

Для забезпечення надійності та безпеки взаємодії, використовується валідація даних та обробка помилок. NestJS Pipes використовується для валідації даних, що надходять від клієнта, що дозволяє перевіряти їх на відповідність встановленим критеріям. Це допомагає захистити додаток від некоректних даних та забезпечує загальну стабільність та надійність в роботі. Також, обробка помилок дозволяє відстежувати і обробляти помилки, що виникають під час взаємодії з API, забезпечуючи користувачам зрозумілі повідомлення про помилки та відповідні коди статусу HTTP.[6]

1.3.2.5 Безпека

У backend частині мобільного застосунку, розробленій за допомогою NestJS, PostgreSQL та Docker, враховано важливі аспекти безпеки для забезпечення надійності та захищеності додатку.

Одним із ключових аспектів безпеки є аутентифікація та авторизація користувачів. Для цього використовується механізм JWT (JSON Web Tokens), який забезпечує захист доступу до приватних даних та функціональності додатку шляхом генерації та перевірки токенів.

Крім того, забезпечується захист конфіденційної інформації шляхом використання сучасних методів шифрування під час зберігання та передачі даних. Це дозволяє уникнути несанкціонованого доступу до даних та зберегти їх конфіденційність.

Таким чином, використання сучасних технологій та підходів в backend частині застосунку гарантує високу продуктивність, безпеку та масштабованість додатку, що важливо для забезпечення задоволення користувачів та захисту їх даних.

1.4 Огляд програмних засобів

Для розробки мобільного застосунку з функціями ділового органайзера було використано декілька ключових програмних засобів, які забезпечили ефективність та зручність у процесі розробки, тестування та симуляції. Ось огляд основних з них:

1.4.1 IDE для JavaScript та TypeScript – WebStorm

WebStorm від JetBrains - це інтегроване середовище розробки (IDE), спеціалізоване на JavaScript, TypeScript та інших веб-технологіях. Воно пропонує розширені можливості для розробки, такі як автодоповнення коду, рефакторинг, інтеграція з системами контролю версій, дебагінг та тестування.[1]

Основні переваги WebStorm включають:

- Інтелектуальне автодоповнення та аналіз коду, що спрощує процес написання коду та допомагає у виявленні помилок.

- Підтримка TypeScript, що забезпечує додаткову безпеку та контроль над типами даних, що особливо важливо для великих проектів.
- Інтеграція з популярними фреймворками та бібліотеками, такими як React та React Native, що полегшує розробку веб-додатків та мобільних застосунків.
- Зручний інтерфейс для роботи з системами контролю версій (Git), що дозволяє зручно взаємодіяти з кодом та зберігати його історію.

Це середовище розробки підходить для командних та індивідуальних проектів, де вимагається висока продуктивність та розширені можливості для розробки веб-додатків.[1]

1.4.2 Симуляція мобільного телефону – Xcode Simulator та ExpoGo

Для симуляції мобільних застосунків використовуються два основних інструменти.

Xcode Simulator, що входить до складу Xcode IDE від Apple, використовується для точної імітації роботи iPhone та iPad. Він дозволяє розробникам тестувати застосунки в різних умовах, не витрачаючи час на реальне підключення пристроїв.

ExpoGo, мобільний додаток, спрощує процес тестування застосунків, створених за допомогою платформи Expo. Завдяки ньому можна запускати та тестувати додатки без компіляції їх у бінарний формат. Однією з переваг є те, що ExpoGo підтримує як iOS, так і Android, що робить його універсальним інструментом для тестування кросплатформених мобільних застосунків.

1.4.3 Тестування застосунку на реальних пристроях – ExpoGo

Для тестування мобільного застосунку на реальних пристроях використовується ExpoGo. Цей інструмент дозволяє швидко завантажувати та запускати застосунок на будь-якому пристрої, який підтримує Expo, просто скануючи

QR-код. Це значно спрощує процес тестування та дозволяє миттєво перевіряти зміни, внесені в код.

Основні переваги використання ExpoGo включають можливість миттєвого перегляду змін в реальному часі, підтримку гарячого перезавантаження (Hot Reloading) та зручний спосіб тестування застосунку на різних пристроях без необхідності повторної збірки.

Використання таких сучасних програмних засобів, як WebStorm, Xcode Simulator та ExpoGo, сприяло зручності та ефективності у процесі розробки, тестування та симуляції мобільного застосунку. Це дозволило досягти високої якості коду, швидкому циклу розробки та тестування, а також забезпечити стабільність та продуктивність кінцевого продукту.

2 МЕТОДИ ТА АЛГОРИТМИ

У цьому розділі детально розглядаються методи та підходи, які будуть використані для розробки мобільного застосунку з функціями ділового органайзера. Враховуючи складність завдань, які необхідно вирішити, обґрунтування вибору відповідних методів і алгоритмів є важливим кроком у процесі розробки.

Мобільний застосунок повинен забезпечувати користувачам ефективне управління своїми завданнями, подіями та контактами, а також надавати інструменти для підвищення продуктивності та оптимізації використання часу. Для досягнення цієї мети, ми будемо використовувати сучасні технології та методології, що включають мікросервісну архітектуру, кросплатформенну розробку, забезпечення безпеки даних та інтеграцію різних сервісів.

2.1 Методи розв'язання задачі

2.1.1 Використання мікросервісної архітектури у бекенді

Мікросервісна архітектура буде використана для реалізації бекенду застосунку, що дозволяє розбити монолітну систему на окремі сервіси, кожен з яких відповідає за конкретний функціонал. Такий підхід має кілька переваг:[2]

Масштабованість: Кожен сервіс може масштабуватися незалежно, що дозволяє ефективніше використовувати ресурси та забезпечувати стабільну роботу системи при зростанні навантаження.[2]

Гнучкість: Розділення на сервіси дозволяє легше оновлювати та додавати новий функціонал, не впливаючи на роботу інших частин системи.[2]

Незалежність: Кожен мікросервіс може бути розроблений, розгорнутий та підтримуватися незалежно, що спрощує розробку та тестування.[2]

У контексті даного проекту, мікросервісна архітектура забезпечить ефективне управління даними користувачів, їхніми завданнями та подіями. Для цього було розроблено два мікросервіси: plans та users.

2.1.2 Застосування принципів модульності та масштабованості

У процесі розробки мобільного застосунку з функціями ділового органайзера будуть використовуватися принципи модульності та масштабованості. Це забезпечить ефективне управління кодом, полегшить процес тестування, а також сприятиме легкому розширенню та підтримці системи в майбутньому.

2.1.2.1 Модульний підхід

Модульність передбачає розділення програми на логічно відокремлені частини (модулі), кожен з яких відповідає за конкретну функціональність. У контексті даного проекту модульний підхід буде реалізовано наступним чином:

Розбиття на модулі: Кодова база буде розділена на окремі модулі, кожен з яких буде відповідати за конкретну область функціональності. Наприклад, модулі для роботи з користувачами, управління завданнями, обробки подій, аутентифікації, тощо.

Чітко визначені інтерфейси: Кожен модуль буде мати чітко визначені інтерфейси для взаємодії з іншими модулями. Це забезпечить незалежність модулів один від одного і спростить їх заміну або оновлення.

Ізольоване тестування: Модульність дозволяє тестувати кожен компонент системи ізольовано, що значно полегшує процес виявлення та виправлення помилок. Кожен модуль можна перевірити на правильність роботи окремо від інших частин системи.

Легка підтримка та розширення: Завдяки модульному підходу, додавання нових функцій або зміна існуючих не потребує значних змін у кодовій базі. Нові

модулі можуть бути легко інтегровані до системи без порушення роботи інших компонентів.

2.1.2.2 Масштабованість

Масштабованість забезпечує здатність системи справлятися зі збільшенням навантаження, додаючи більше ресурсів або модулів. Для досягнення масштабованості в даному проекті будуть застосовані наступні підходи:

Горизонтальне масштабування передбачає додавання нових екземплярів сервісів для розподілу навантаження. Наприклад, якщо сервіс, що обробляє плани користувачів, починає отримувати занадто багато запитів, можна додати нові екземпляри цього сервісу для розподілу навантаження.

Автоматичне масштабування передбачає використання хмарних платформ та оркестраторів контейнерів (наприклад, Kubernetes) дозволить автоматично масштабувати сервіси в залежності від поточного навантаження. Це забезпечить стабільну роботу системи навіть при пікових навантаженнях.

Розподілені обчислення передбачає застосування підходів розподілених обчислень які дозволить розподіляти задачі між різними сервісами та серверами, що підвищить продуктивність і надійність системи.

Балансування навантаження передбачає використання технологій балансування навантаження (load balancing) забезпечить рівномірний розподіл запитів між доступними сервісами, що підвищить ефективність використання ресурсів та покращить час відгуку системи.

Застосування принципів модульності та масштабованості у розробці мобільного застосунку з функціями ділового органайзера забезпечить гнучкість, надійність та ефективність системи. Це дозволить легко підтримувати та розширювати функціонал, адаптуючись до зростаючих вимог користувачів та змін у бізнес-середовищі.

2.1.3 Реалізація кросплатформенності з використанням React Native

Фронтенд мобільного застосунку буде реалізований з використанням сучасних технологій, таких як React Native, що дозволяє розробляти кросплатформні застосунки. Це дозволяє зекономити час і ресурси, розробляючи єдину кодову базу для різних платформ (iOS та Android). [13]

2.1.3.1 Переваги використання React Native

React Native відомий своєю кросплатформенністю, що спрощує розробку мобільних додатків для різних платформ. Замість витрат на розробку окремих кодових баз для iOS та Android, розробники можуть використовувати один спільний код, що зменшує витрати та спрощує підтримку.[14]

Гаряче оновлення (Hot Reloading) дозволяє розробникам швидко бачити зміни у застосунку без повного перезавантаження. Це прискорює процес розробки та полегшує пошук та виправлення помилок.[14]

Широка екосистема React Native включає велику кількість доступних бібліотек та компонентів, що значно полегшує розробку, оскільки можна використовувати готові рішення для різних задач.[14]

Адаптивність інтерфейсу та високий рівень кастомізації дозволяють створювати інтерфейси, що легко адаптуються під різні розміри екранів та орієнтації пристроїв, максимально відповідаючи потребам користувачів.

2.1.3.2 Впровадження React Native у проекті

Впровадження React Native у проекті включає кілька важливих аспектів, що спрямовані на полегшення розробки та підвищення якості застосунку.

Один з ключових аспектів - це розбиття застосунку на незалежні компоненти, що дозволяє зберігати код організовано та спрощує його обслуговування. Крім того, використання менеджменту стану, такого як Redux або Context API, робить керування даними більш ефективним.

У процесі тестування та відлагодження використовуються інструменти, що автоматизують процес, такі як Jest та Enzyme, що допомагає забезпечити високу якість коду та мінімізує кількість помилок у виробничому середовищі. Також використовуються вбудовані інструменти відлагодження, такі як Reactotron, для полегшення аналізу продуктивності застосунку.

Оптимізація продуктивності включає в себе використання оптимізованих компонентів та технік, таких як мемоізація, для зменшення навантаження на рендеринг та підвищення продуктивності. Також важливо зменшувати розмір бандлів, щоб зменшити час завантаження застосунку на пристроях користувачів.

Разом ці підходи, спрямовані на кросплатформенність, швидкий розвиток та гнучкість інтерфейсу, допомагають створити мобільний застосунок високої якості з мінімальними витратами часу та ресурсів, який легко підтримується та розширюється у майбутньому.

2.1.4 Використання RESTful API для взаємодії між фронтендом і бекендом

RESTful API (Representational State Transfer Application Programming Interface) є архітектурним стилем для створення веб-сервісів, що дозволяє клієнтам та серверам обмінюватися даними через HTTP-запити. У рамках даного проекту використання RESTful API буде ключовим елементом для забезпечення ефективної взаємодії між фронтендом і бекендом. Це дозволить створити добре структуровану, масштабовану та підтримувану систему.

2.1.4.1 Основні принципи RESTful API

Основні принципи RESTful API полягають у створенні ресурсів, що відображають об'єкти, які обробляються, та наданні доступу до них через стандартні HTTP методи. Ресурси ідентифікуються унікальними URI, такими як `/users` або `/plans`, які структуровані логічно.

HTTP методи використовуються для різних дій з ресурсами: GET для отримання даних, POST для створення нових ресурсів, PUT для оновлення існуючих та DELETE для видалення.

Коди статусу HTTP повертаються у відповідь на запити для інформування клієнта про результати, такі як успішність операції чи помилки.

Важливим принципом RESTful є безстанова взаємодія, що означає, що кожен запит повинен містити всю необхідну інформацію для обробки, і сервер не зберігає стану між запитами. Це робить систему масштабованою і спрощує підтримку.

2.1.4.2 Впровадження RESTful API у проекті

Під час впровадження RESTful API у проекті важливо спочатку створити структуру API, визначивши основні ресурси та їх ендпоінти, наприклад, для користувачів або планів. Потім потрібно забезпечити безпеку API через аутентифікацію та авторизацію з використанням JWT або OAuth, а також шифрування даних через HTTPS.

Документування API також важливий аспект, і для цього можна використовувати інструменти, які допоможуть створити зрозумілу документацію, наприклад, Swagger або OpenAPI. Також корисно надати приклади запитів та відповідей для кожного ендпоінту.

Нарешті, тестування API є ключовим етапом. Рекомендується писати юніт-тести для кожного ендпоінту для перевірки коректності їх роботи, а також використовувати інтеграційне тестування з інструментами, такими як Postman або Insomnia, для перевірки робочого циклу запитів та відповідей.

2.1.4.3 Переваги використання RESTful API

Використання RESTful API має кілька ключових переваг такі як: масштабованість, гнучкість та адаптивність та відповідність стандартам. Нижче розглянемо кожну перевагу.

Масштабованість передбачає легкість горизонтального масштабування сервісів завдяки безстанній природі RESTful запитів та можливість розподілу навантаження між декількома серверами.

Гнучкість та адаптивність передбачають легкість у додаванні нових ендпоінтів або зміні існуючих без порушення роботи клієнтських застосунків та можливість використання API різними клієнтами (мобільні застосунки, веб-клієнти, інші сервіси).

Відповідність стандартам передбачає дотримання стандартів REST забезпечує високу сумісність з іншими сервісами та інструментами, а також використання загальноприйнятих методів та практик, що робить API зрозумілим та легко підтримуваним.

Використання RESTful API для взаємодії між фронтендом і бекендом забезпечить ефективну, масштабовану та безпечну систему обміну даними, що дозволить швидко реагувати на зміни та вимоги користувачів.

2.1.5 Застосування баз даних для зберігання даних користувачів

У світі розробки програмного забезпечення вибір правильної бази даних є важливим рішенням, яке може значно вплинути на продуктивність, масштабованість і простоту використання вашого додатка. З численними доступними варіантами, може бути складно визначити найкращу систему управління базами даних (DBMS), яка ідеально підійде для ваших потреб.[3]

Для зберігання даних користувачів, їхніх завдань та подій буде використовуватися реляційна база даних PostgreSQL. Ця база даних забезпечує високу надійність, продуктивність та підтримку складних запитів, що дозволяє ефективно працювати з великими обсягами даних.[3]

PostgreSQL має кілька важливих переваг, які роблять її привабливим вибором для зберігання даних користувачів та їхніх завдань. Насамперед, PostgreSQL відома своєю стабільністю та надійністю, що критично важливо для зберігання важливих

даних. Вона забезпечує високу доступність і захист від збоїв завдяки використанню механізмів резервного копіювання та відновлення даних.

Крім того, потужні можливості SQL дозволяють виконувати складні запити та аналіз даних, що робить PostgreSQL ідеальним вибором для завдань, які потребують складної обробки даних. Масштабованість також є сильною стороною PostgreSQL, оскільки вона підтримує як вертикальне масштабування на рівні окремого сервера, так і горизонтальне масштабування в кластері, що дозволяє збільшувати продуктивність системи без значних змін у її структурі.[3]

Проте PostgreSQL має і свої недоліки. Незважаючи на велику спільноту та підтримку, документація все ще може бути непослідовною та неповною, що може викликати труднощі при використанні нових або менш поширених функцій. Також відсутність інструментів для звітності та аудиту є значним недоліком, оскільки це вимагає постійного ручного контролю за станом системи.

Нижче на графіку наведено популярність БД, як бачимо PostgreSQL є найпопулярнішою (Рисунок 2.1.5.1)[3]

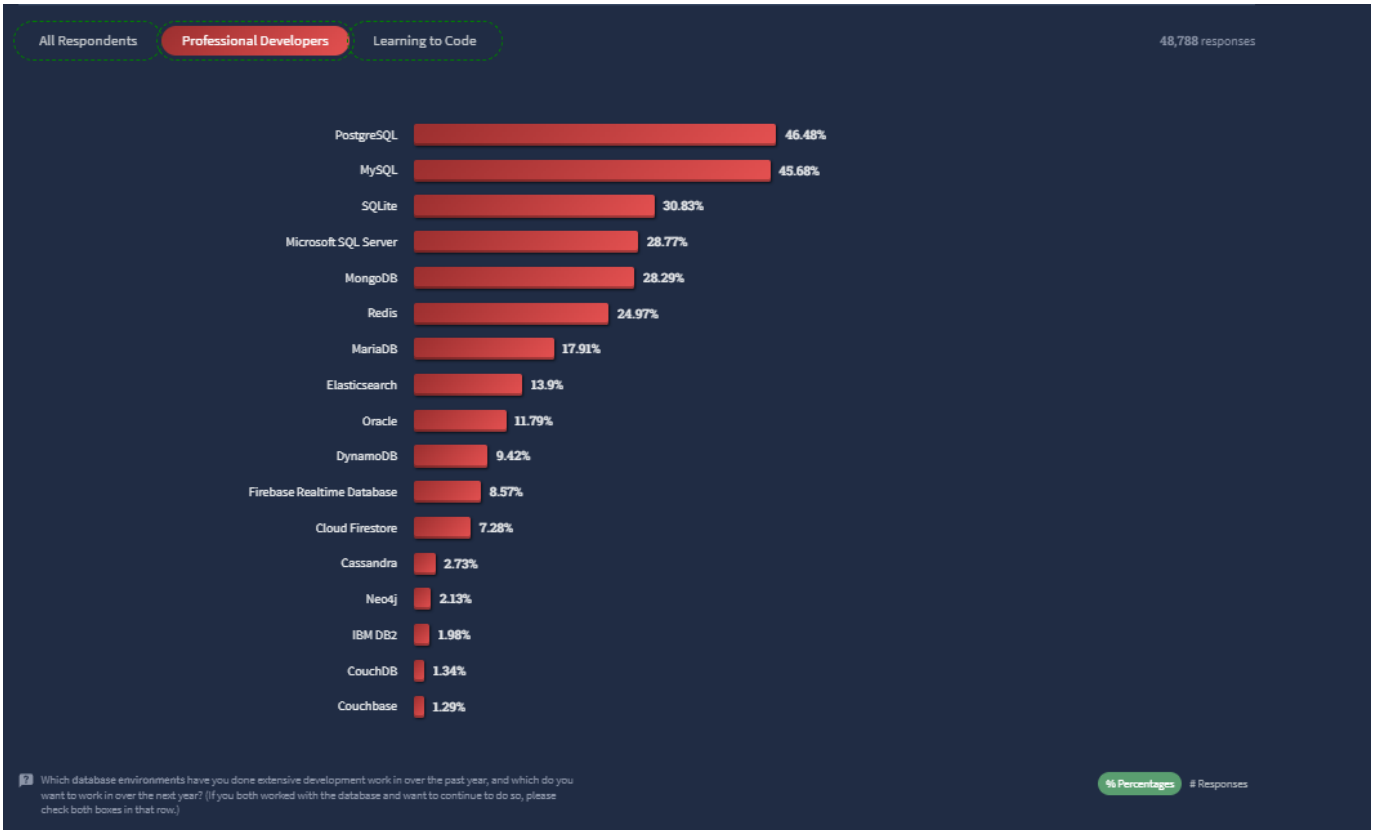

Рисунок 2.1.5.1 – Найпопулярніші системи баз даних [3]

Завдяки своїм можливостям обробки складних запитів і широкому вибору інтерфейсів, PostgreSQL ідеально підходить для аналізу даних і зберігання інформації. Якщо ви створюєте інструмент для автоматизації баз даних, PostgreSQL буде найкращим вибором завдяки своїм аналітичним можливостям, відповідності ACID та потужному SQL-двигуну. Вона популярна серед фінансових установ і телекомунікаційних систем завдяки здатності швидко обробляти великі обсяги даних. [3]

Загалом, вибір PostgreSQL як основної бази даних для зберігання даних користувачів та їхніх завдань у мобільному застосунку з функціями ділового органайзера обґрунтований її високою надійністю, продуктивністю та гнучкістю, що дозволяє задовольнити потреби як малого бізнесу, так і великих підприємств.[3]

2.1.6 Забезпечення безпеки даних і автентифікація користувачів

В рамках розробки мобільного застосунку необхідно вдосконалити методи забезпечення безпеки даних та здійснювати автентифікацію користувачів для захисту конфіденційності та цілісності інформації. Нижче наведено опис ключових методів безпеки та автентифікації, які будуть використовуватися у проекті:

2.1.6.1 Автентифікація користувачів

Автентифікація - це процес підтвердження ідентичності користувача перед наданням доступу до захищених ресурсів. Для цього можна використовувати такий метод як JWT (JSON Web Tokens). JWT є відкритим стандартом (RFC 7519), який використовується для створення токенів доступу, які можна використовувати для автентифікації та авторизації користувачів. Після успішної автентифікації користувача сервер генерує JWT, який містить певну інформацію про користувача та його права доступу. Цей токен передається клієнту, який використовує його для авторизації в майбутніх запитах. JWT забезпечує безпеку, оскільки він може бути підписаний сервером, інформація в ньому може бути зашифрована та перевірена на цілісність.

2.1.6.2 Шифрування даних

Шифрування даних - це процес перетворення звичайного тексту в нечитабельний формат (шифрований текст) за допомогою певного алгоритму шифрування. Шифрування даних допомагає захистити конфіденційні дані від несанкціонованого доступу. Для забезпечення безпеки даних в мобільному застосунку можна використовувати такі методи:

– Шифрування на рівні бази даних: Для зберігання конфіденційних даних, таких як паролі користувачів, можна використовувати методи шифрування, які надаються базою даних. Наприклад, можна застосувати хешування паролів з використанням солі для підвищення безпеки.

– Шифрування трафіку: Для захисту конфіденційних даних, які передаються між клієнтом та сервером через мережу, можна використовувати протокол HTTPS (HTTP over SSL або HTTP Secure), який шифрує дані за допомогою SSL (Secure Sockets Layer) або TLS (Transport Layer Security) протоколу.

Впровадження вищезазначених методів безпеки дозволить мобільному застосунку забезпечити високий рівень захисту конфіденційності та безпеки даних, а також надати користувачам впевненість у захищеності їхньої інформації.

2.2 Характеристики та параметри

2.3.1 Параметри налаштування середовища розробки

У процесі розробки мобільних додатків на React Native важливо налаштувати середовище розробки для зручності та ефективності. Використовуючи такі інструменти, як WebStorm, Expo Go та Xcode Simulator, розробники можуть спростити свою роботу та забезпечити високу якість своїх додатків.[14]

WebStorm - це потужне інтегроване середовище розробки, яке надає розробникам ряд інструментів для зручної та ефективної роботи над проектами. Він дозволяє налаштувати редактор коду, встановлювати різноманітні плагіни для підтримки розробки на різних технологіях, підключатися до системи контролю версій Git та використовувати шаблони коду для прискорення процесу написання коду.

Expo Go - це мобільний додаток, що дозволяє розробникам тестувати свої проекти безпосередньо на мобільних пристроях. Він забезпечує можливість налаштування Expo аккаунту для зберігання та спільного доступу до проектів, підключення до робочого середовища для тестування додатків в реальному часі та налаштування сповіщень для отримання повідомлень про стан розробки.

Xcode Simulator - це інструмент, який дозволяє тестувати мобільні додатки на пристроях з операційною системою iOS. Він надає можливість вибрати пристрій та

його версію для тестування, налаштувати обмеження для перевірки адаптивності додатку та управляти ресурсами для тестування в умовах обмеженого середовища.

Параметри налаштування середовища розробки допомагають забезпечити зручність та ефективність у процесі розробки мобільних додатків на React Native.

2.3.2 Налаштування сервера та розгортання на Docker

2.3.2.1 Налаштування сервера:

При налаштуванні сервера важливо враховувати кілька ключових аспектів. Спочатку необхідно вибрати сервер, що відповідає вашим потребам і вимогам проекту. Рекомендовані варіанти включають такі послуги, як AWS, DigitalOcean, Google Cloud або використання власних серверів. Далі слід встановити та налаштувати операційну систему на сервері. Для більшості ситуацій підходять Linux-дистрибутиви, такі як Ubuntu, CentOS або Debian.

Після цього необхідно встановити всі необхідні залежності для роботи серверного додатку, такі як Node.js, PostgreSQL, Nginx та інші. Також важливо налаштувати безпеку сервера, включаючи налаштування брандмауерів, обмеження доступу SSH, налаштування SSL-сертифікатів та інші заходи безпеки.

Окремо слід звернути увагу на конфігурацію веб-сервера, такого як Nginx або Apache. Це дозволить ефективно обробляти запити та маршрутизувати трафік до серверного додатку, забезпечуючи стабільну та надійну роботу системи.

2.3.2.2 Розгортання на Docker

Для розгортання на Docker необхідно виконати кілька ключових кроків. Почнемо зі створення Dockerfile, в якому описані всі кроки для створення контейнера з серверним додатком, включаючи вибір базового образу, копіювання файлів та встановлення залежностей.

Далі, для налаштування середовища розгортання створюється файл docker-compose.yml. Цей файл дозволяє налаштувати контейнер для серверного додатку, бази даних, а також можливо, веб-сервера та інших складових частин системи.

Після підготовки конфігураційних файлів потрібно виконати будівництво та запуск контейнера за допомогою команд Docker. Команда `docker build` використовується для створення образу, а `docker run` - для запуску контейнера з серверним додатком.

Нарешті, для ефективного моніторингу та керування контейнерами використовуються інструменти, такі як Docker Compose, Docker Swarm або Kubernetes. Вони дозволяють керувати багатьма контейнерами, масштабувати їх та моніторити їх роботу.

Розгортання на Docker забезпечує створення ізольованого середовища для серверного додатку, що полегшує процес розгортання та керування ним, забезпечуючи при цьому надійність та стабільність роботи системи.

2.3.3 Тестування та відлагодження

Налаштування тестування та відлагодження коду – це важливий етап у розробці програмного продукту, який забезпечує його якість та надійність. Методи тестування та відлагодження можуть бути різноманітні, включаючи юніт-тести, які спрямовані на перевірку окремих компонентів програми без залежності від інших, інтеграційні тести, що перевіряють взаємодію між різними частинами системи, та E2E тести, які симулюють поведінку реального користувача.

Ручне тестування використовується для перевірки випадків, які складно або неможливо автоматизувати, в той час як відлагодження коду допомагає виявити та виправити помилки. Профілювання та оптимізація коду використовуються для аналізу продуктивності програми та оптимізації її швидкодії.

Моніторинг віддалених запитів дозволяє виявити проблеми та оптимізувати використання мережевих ресурсів. Ці методи разом допомагають забезпечити якість, надійність та оптимальну продуктивність програмного продукту.

2.3.4 Забезпечення сумісності з різними версіями мобільних операційних систем

Забезпечення сумісності з різними версіями мобільних операційних систем (iOS та Android) є важливою частиною розробки мобільного додатку. Ось деякі заходи, які допомагають забезпечити цю сумісність:

Використання сучасних API: Важливо використовувати API, які підтримуються на різних версіях операційних систем. При виборі API слід уникати застарілих методів та функцій, які можуть бути вилучені у нових версіях.

Тестування на різних пристроях і версіях ОС: Перед випуском додатку важливо провести тестування на різних пристроях та версіях операційних систем. Це допоможе виявити можливі проблеми зі сумісністю та адаптувати додаток для різних умов.

Адаптивний дизайн: Використання адаптивного дизайну дозволяє автоматично адаптувати інтерфейс додатку до різних розмірів екрану та орієнтацій пристрою. Це полегшує користування додатком на різних пристроях.

Використання сумісних бібліотек та інструментів: Під час розробки слід використовувати бібліотеки та інструменти, які підтримуються на різних версіях операційних систем. Це дозволяє уникнути проблем зі сумісністю та забезпечити плавну роботу додатку.

Постійне оновлення: Важливо постійно вносити оновлення до додатку, враховуючи зміни у специфікаціях операційних систем та вимоги ринку. Це дозволяє забезпечити сумісність з новими версіями ОС та зберегти актуальність додатку для користувачів.

Моніторинг і зворотний зв'язок від користувачів: Слід постійно отримувати зворотний зв'язок від користувачів щодо роботи додатку на різних пристроях та версіях ОС. Це допоможе виявити проблеми зі сумісністю та вчасно їх виправити.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

Перш ніж переходити до реалізації програмного забезпечення, важливо ретельно спроектувати його структуру та функціональність. Ефективне проектування є основою успішного виконання завдання, оскільки воно забезпечує ясність і чіткість у всіх аспектах розробки, від визначення вимог до створення архітектури і вибору відповідних технологій.

У цьому розділі детально розглядаються всі етапи програмної реалізації мобільного застосунку з функціями ділового органайзера. Метою є створення надійного, масштабованого та зручного у використанні застосунку, який відповідає всім функціональним вимогам і потребам користувачів.

3.1 Функціональна модель

Функціональна модель є важливим етапом у розробці програмного забезпечення, оскільки вона визначає основні функції та взаємодії системи, що дозволяє краще зрозуміти її можливості та поведінку. У цьому підрозділі буде розглянута функціональна модель мобільного застосунку з функціями ділового органайзера на основі діаграми варіантів використання (usecase diagram).

Діаграма варіантів використання ілюструє взаємодію користувача із застосунком, показуючи основні функції, які він може виконувати. Нижче описано основні сценарії використання системи.

На діаграмі варіантів використання користувач (User) зображений як актор, що взаємодіє з різними функціями застосунку (Plan App). Основні взаємодії включають створення, видалення та редагування планів, пошук планів за назвою, а також управління профілем користувача і сесіями. Рисунок 3.1.1

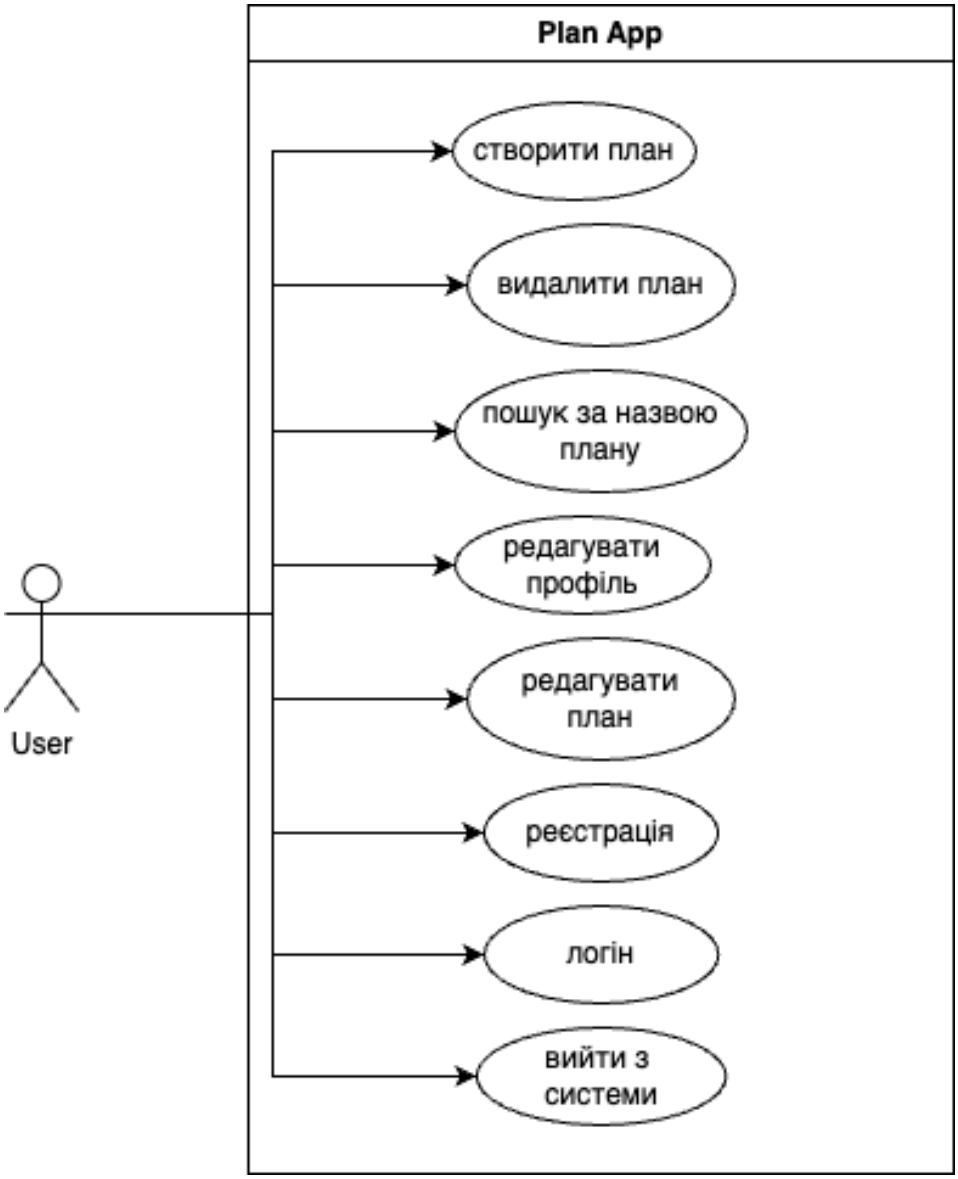

Рисунок 3.1.1 діаграма варіантів використання

На діаграмі варіантів використання, представленій на Рисунку 3.1.1, користувач (User) зображений як актор, який взаємодіє з різними функціями застосунку (Plan App). Основні взаємодії включають: створення плану, видалення плану, пошук за назвою плану, редагування профілю, редагування плану, реєстрацію, логін, вихід з системи. Ці взаємодії відображають основні сценарії використання мобільного

застосунку та допомагають зрозуміти, як користувачі будуть взаємодіяти з системою для досягнення своїх цілей.

3.2 Архітектура системи

Архітектура мобільного застосунку з функціями ділового органайзера є важливим аспектом розробки, оскільки вона визначає структуру та організацію всього додатку з урахуванням його функціональних вимог та потреб користувачів. Основні компоненти архітектури включають: клієнтську частину (мобільний інтерфейс), серверну частину (бекенд), базу даних, протоколи комунікації, засобів забезпечення безпеки.

Клієнтська частина це фронтендова частина застосунку, яка відповідає за відображення інтерфейсу користувача та обробку його дій. У випадку мобільного додатку, це мобільний інтерфейс, реалізований за допомогою React Native або аналогічного фреймворку. Клієнтська частина забезпечує зручність взаємодії користувача з додатком та відповідає за передачу запитів до серверної частини.

Серверна частина це бекенд, який є центральною складовою системи, яка відповідає за обробку запитів від клієнтської частини та управління базою даних. Вона може бути реалізована за допомогою фреймворків, таких як NestJS (Node.js), які забезпечують можливості для створення надійних та масштабованих серверних додатків.

База даних використовується для зберігання інформації про користувачів, завдання, нагадування та інші дані, необхідні для роботи застосунку. Вона може бути реалізована за допомогою реляційних баз даних, таких як PostgreSQL, з використанням ORM TypeORM для забезпечення доступу до даних.

Для забезпечення взаємодії між клієнтською та серверною частинами, а

також з базою даних, використовуються відповідні протоколи комунікації. Наприклад, для взаємодії між клієнтом та сервером може використовуватися протокол HTTP або WebSocket.

Забезпечення безпеки даних та захист від зловживань є важливим аспектом розробки мобільного застосунку. Для цього можуть використовуватися різноманітні засоби, такі як аутентифікація та авторизація користувачів, шифрування даних та захист від атак на витік інформації.

Архітектура системи мобільного застосунку з функціями ділового органайзера повинна бути ретельно спроектована з урахуванням всіх вищезазначених аспектів для забезпечення високої продуктивності, безпеки та зручності користування.

Архітектура системи мобільного застосунку з функціями ділового органайзера побудована на основі мікросервісної архітектури, що забезпечує розділення функціональних компонентів на окремі, незалежні служби. Це дозволяє ефективніше управляти кодовою базою, підвищує масштабованість системи та забезпечує надійне управління різними аспектами застосунку.

Plans Service відповідає за все, що пов'язане з планами, включаючи створення, редагування, видалення та отримання планів користувачів. Цей мікросервіс обробляє всі запити, що стосуються управління завданнями та подіями. Основні функції Plans Service включають створення нових планів, редагування існуючих планів, видалення планів, отримання списку планів для конкретного користувача та пошук планів за назвою.

Users Service відповідає за реєстрацію, автентифікацію та управління профілями користувачів. Цей мікросервіс обробляє всі запити, що стосуються користувачів, включаючи створення нових акаунтів, вхід до системи, оновлення інформації про користувачів та інші операції з профілем. Основні функції Users Service включають реєстрацію нових користувачів, автентифікацію користувачів, оновлення профілю користувачів, видалення акаунтів та управління сесіями користувачів. Рисунок 3.2.1 ілюструє архітектуру системи.

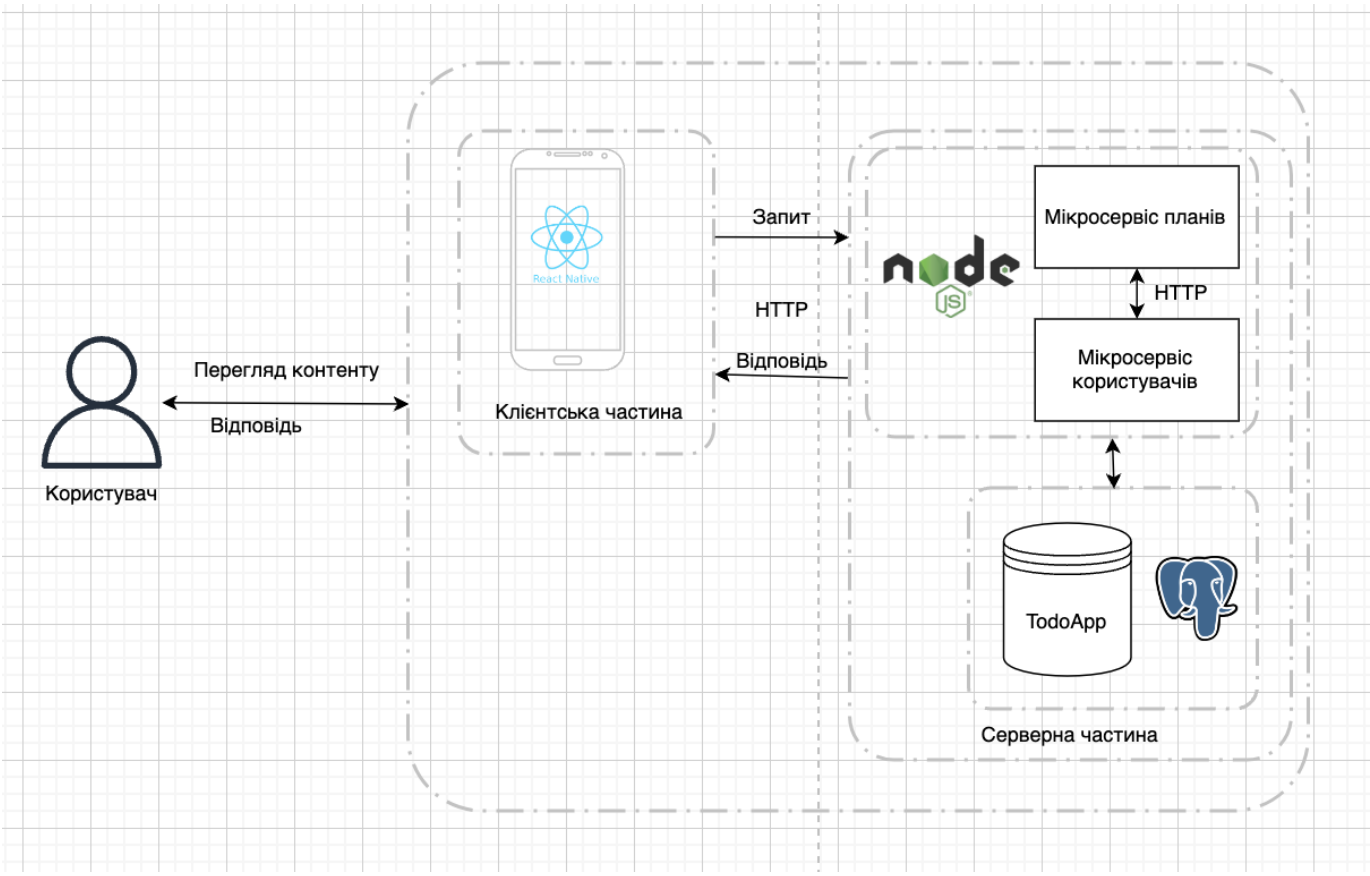

Рисунок 3.2.1 Архітектурна діаграма застосунку

Як бачимо з рисунку 3.2.1 на серверній частині є два мікросервіси: користувачів та планів які взаємодіють між собою через HTTP-запити. Розглянемо їх детальніше в наступних пунктах.

3.2.1 Мікросервіс користувачів

Мікросервіс користувачів відповідає за всі операції, пов'язані з управлінням користувачами, включаючи реєстрацію, автентифікацію та управління профілями користувачів. Цей мікросервіс забезпечує роботу з користувачами через набір HTTP-методів, які дозволяють створювати нові акаунти, входити в систему, оновлювати дані

профілю та отримувати інформацію про користувачів. Розглянемо діаграму класів мікросервісу користувачів, що зображена на рисунку 3.2.1.1

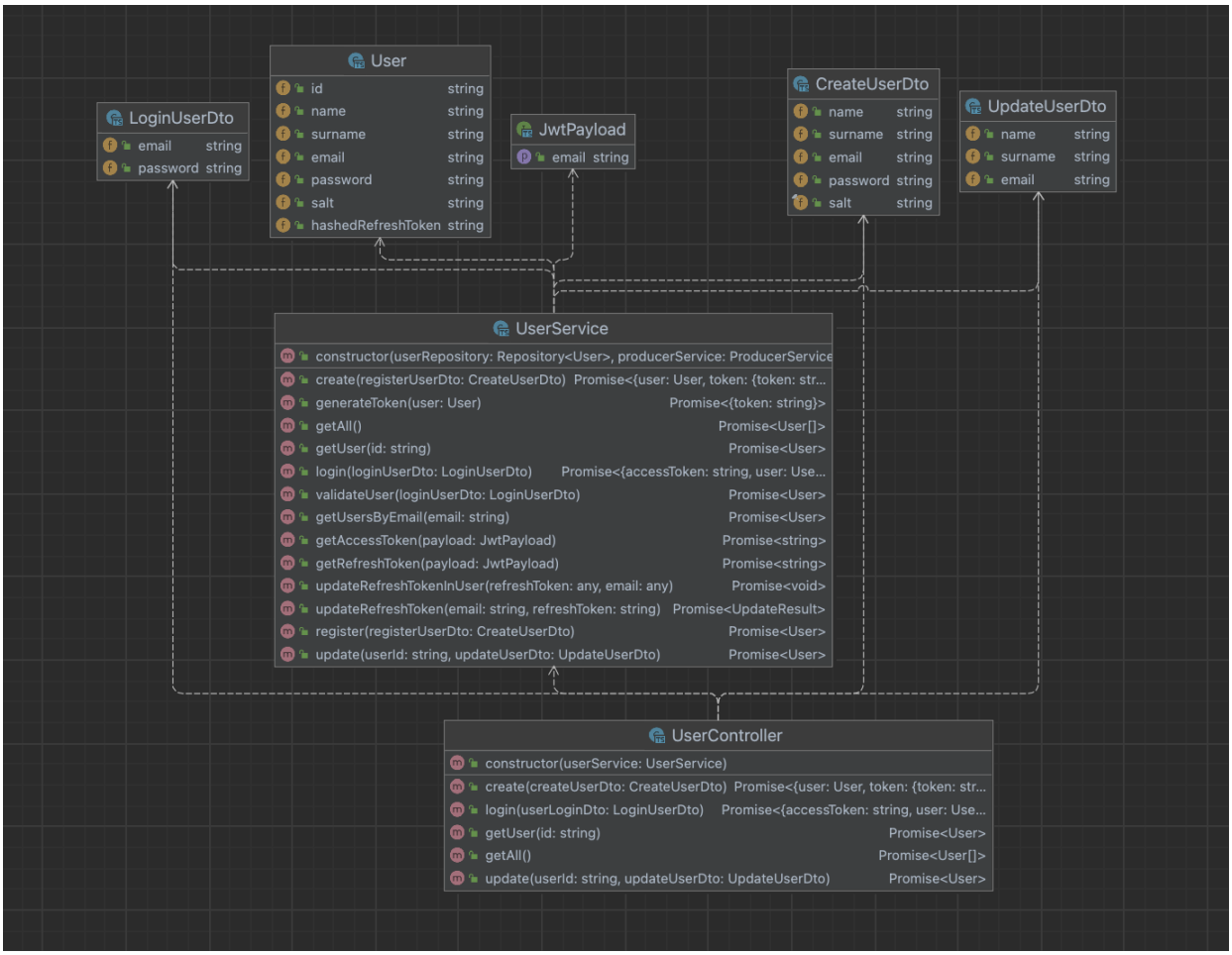

Рисунок 3.2.1.1 Діаграма класів мікросервісу користувача

Діаграма класів мікросервісу користувача (рисунок 3.2.1.1) відображає основні класи та їх зв'язки, які використовуються для реалізації функціональності цього мікросервісу. Нижче наведено детальний розгляд діаграми класів.

Клас UserController відповідає за обробку HTTP-запитів, пов'язаних з користувачами. Він містить методи для створення нового користувача, входу в систему, отримання інформації про користувачів та інші операції. Також в ньому

знаходяться всі маршрути цього мікросервісу, про які детальніше буде написано нижче (Таблиця 3.2.1.1 – Основні маршрути мікросервісу користувачів).

Клас UserService забезпечує логіку для виконання операцій з користувачами, таких як реєстрація, автентифікація та управління профілями. Він взаємодіє з базою даних для збереження та отримання інформації про користувачів.

Клас User представляє модель користувача. Він містить поля для зберігання основних даних про користувача, таких як ім'я, email, пароль тощо.

Клас CreateUserDto використовується для передачі даних при створенні нового користувача. Він містить поля, такі як: name, surname, email, password, salt.

Клас UpdateUserDto використовується для передачі даних при оновленні профілю користувача. Він містить поля, які можуть бути змінені користувачем, наприклад, ім'я або пароль. Він містить поля, такі як: name, surname, email.

Клас LoginUserDto використовується для передачі даних під час процедури автентифікації. Він містить email та пароль користувача.

Клас JwtPayload представляє об'єкт, який містить дані, які будуть закодовані у токені JWT (JSON Web Token). Це може включати ідентифікатор користувача, роль або будь-яку іншу інформацію, яка потрібна для автентифікації та авторизації користувача. У нашому випадку він містить лише одне поле – email, тобто пошту користувача.

Для взаємодії клієнтської частини з серверною використовуюсь HTTP-запроси, нижче наведена таблиці маршрутів мікросервісу користувачів (‘Users’) (Таблиця 3.2.1.1).

Таблиця 3.2.1.1 – Основні маршрути мікросервісу користувачів

Назва інтерфейсу	Назва методу	HTTP-метод	Опис
/user	/	POST	Створення нового користувача
	/login	POST	Вхід у систему

	/:userId	GET	Дістаємо дані користувача по його id
	/	GET	Отримання всіх користувачів
	/:userId	PATCH	Редагування даних користувача

Розглянемо детальніше кожен маршрут.

Маршрут /user/ з методом POST приймає тіло CreateUserDto і у відповідь отримуємо нового створеного користувача, який вже є у БД.

Маршрут /user/ login з методом POST приймає тіло LoginUserDto і у відповідь отримуємо дані користувача, ‘accessToken’ та ‘refreshToken’користувача, які надають доступ до сторінок мобільного застосунку.

Маршрут /user/ userId з методом GET приймає тіло лише один парамерт – id користувача і у відповідь отримуємо всі дані користувача. Цей маршрут є важливим, тому що до нього часто зсисається мікросервіс планів за допомоги HTTP-запиту, для перевірки існування користувача який робить запроси пов’язані з планами.

Маршрут /user/ з методом GET нічого не приймає і у відповідь отримуємо всіх користувачів з БД.

Маршрут /user/:userId з методом PATCH приймає тіло UpdateUserDto і у відповідь отримуємо редагованого користувача.

3.2.2 Мікросервіс планів

У цьому підрозділі розглядається мікросервіс, що відповідає за управління планами користувачів. Він забезпечує можливість створення, редагування, видалення та отримання планів для конкретного користувача. Мікросервіс також має можливість фільтрувати плани за різними критеріями, для зручного перегляду. Розглянемо діаграму класів мікросервісу планів, що зображена на рисунку 3.2.2.1

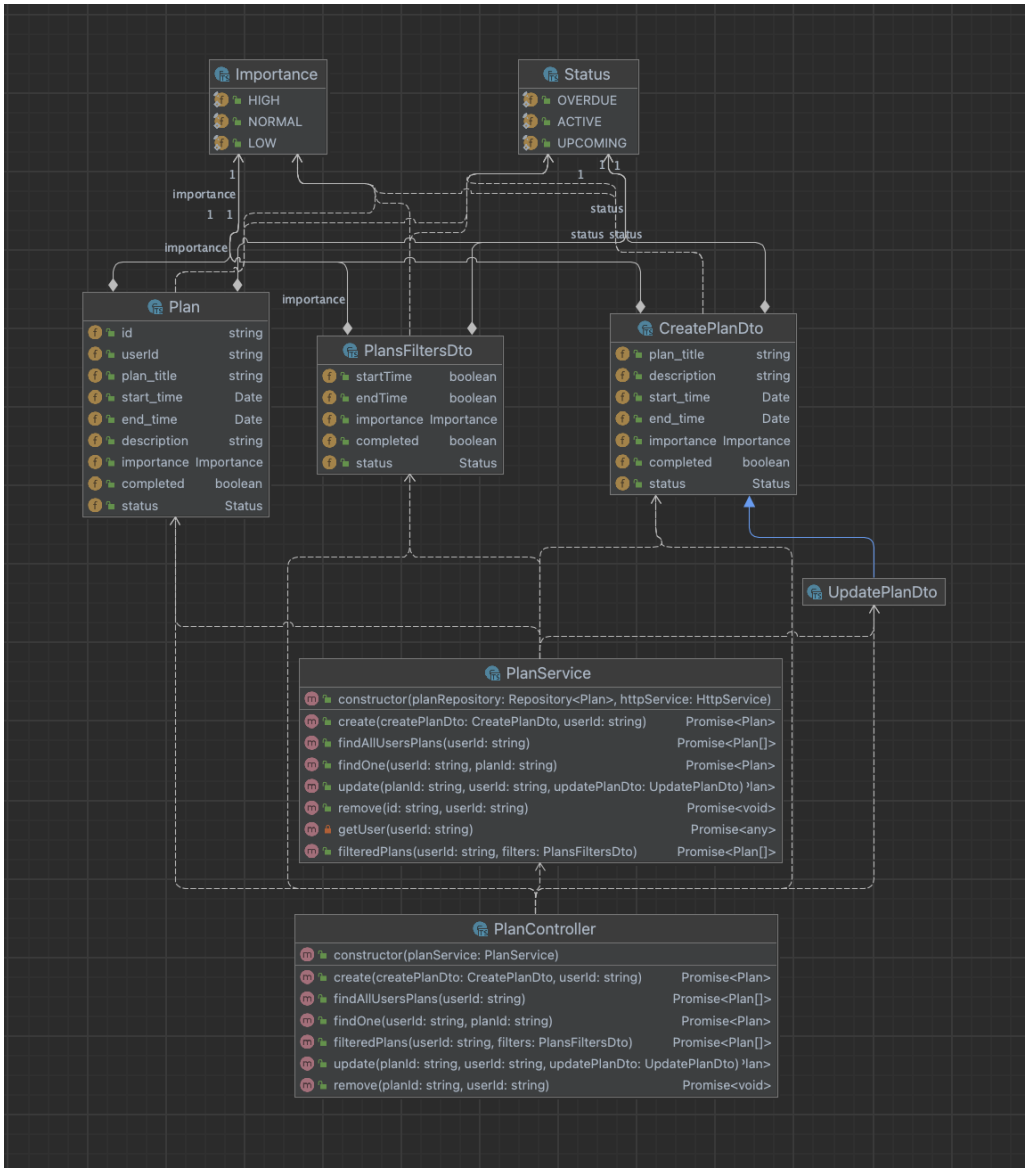

Рисунок 3.2.1 Діаграма класів мікросервісу планів

На діаграмі зображені основні класи, які використовуються у мікросервісі "Плани". Основні класи включають:

Клас PlanController це контролер, який обробляє HTTP-запити для роботи з планами.

Клас PlanService це сервіс, який забезпечує логіку створення, редагування, видалення та отримання планів.

Клас Plan представляє окремий план та зберігається вся необхідна інформація про нього.

Клас CreatePlanDto об'єкт передачі даних (DTO) для створення нового плану.

Клас UpdatePlanDto об'єкт передачі даних (DTO) для редагування існуючого плану.

Клас PlansFiltersDto об'єкт передачі даних (DTO) для фільтрації списку планів за різними критеріями.

Клас Importance (hight, normal, low) це тип перерахунку, який визначає важливість плану (висока, звичайна, низька).

Клас Status (overdue, active, upcoming) це тип перерахунку, який визначає статус плану (просрочений, активний, майбутній).

Ці класи взаємодіють між собою для забезпечення функціональності мікросервісу "Плани".

Для взаємодії клієнтської частини з серверною використовуюсь HTTP-запроси, нижче наведена таблиці маршрутів мікросервісу планів (‘Plans) (Таблиця 3.2.2.1).

Таблиця 3.2.2.1 – Основні маршрути мікросервісу планів

Назва інтерфейсу	Назва методу	HTTP-метод	Опис
/plans	/:userId	POST	Створення плану користувачем
	/:userId	GET	Отримання планів користувача
	/:userId/:planId	GET	Отримання конкретного плану користувача
	/filtered-plans/:userId	GET	Отримання відфільтровані дані користувача
	/:userId/:planId	PATCH	Редагування плану

	/:userId/:planId	DELETE	Видалення плану

Розглянемо детальніше кожен маршрут.

Маршрут /plans/:userId з методом POST приймає тіло CreatePlanDto і параметр userId, цей параметр повинен для перевірки користувача, який робить цей запрос. У відповідь отримуємо новий створений план, який вже є у БД.

Маршрут /plans/:userId з методом GET приймає лише параметр userId і у відповідь отримуємо всі плани цього користувача.

Маршрут /plans/:userId/:planId з методом GET приймає тіло параметри userId та planId і у відповідь отримуємо план ід якого є planId, користувача ід якого є userId.

Маршрут /plans/filtered-plans/:userId з методом GET приймає тіло PlansFiltersDto та параметр userId. У відповідь отримуємо відфільтровані плани відповідно до обраних користувачем.

Маршрут /plans/:userId/:planId з методом PATCH приймає тіло UpdatePlanDto і параметри userId та planId. У відповідь отримуємо відредагований план відповідно до нових значень з UpdatePlanDto.

Маршрут /plans/:userId/:planId з методом DELETE приймає параметри userId та planId. У відповідь нічого не отримуємо окрім коду статусу відповіді, тому що цей маршрут сотворений для видалення конкретного плану за його ідентифікатором, тобто його id.

3.2.3 Алгоритми управління завданнями та подіями

Управління завданнями та подіями є однією з ключових функцій мобільного застосунку. Для ефективного виконання цих завдань необхідно мати відповідні алгоритми, які забезпечують правильну реалізацію функціоналу. Створення завдання або події починається з введення користувачем необхідної інформації, після чого запит на створення надсилається на сервер, який зберігає інформацію в базі даних. Редагування завдання або події включає вибір користувачем елемента для

редагування, внесення змін та відправлення запиту на сервер для оновлення даних у базі. Видалення завдання або події здійснюється через вибір користувачем елемента для видалення та відправлення запиту на сервер, який видаляє відповідний запис з бази даних. Відстеження завдань та подій передбачає автоматичний моніторинг дат та часу подій сервером, який надсилає сповіщення користувачеві у разі наближення важливих подій.

3.2.4 Алгоритми синхронізації даних між клієнтом і сервером

Синхронізація даних між клієнтом та сервером є критичним аспектом для забезпечення коректної та надійної роботи мобільного застосунку. При створенні нових записів клієнтська частина надсилає запит на сервер, який зберігає дані у базі та повертає підтвердження, після чого клієнтська частина оновлює локальну базу даних. При оновленні записів зміни надсилаються на сервер, який оновлює дані та повертає підтвердження для оновлення локальної бази. Видалення записів включає відправлення запиту на сервер для видалення запису з бази даних та оновлення локальної бази після підтвердження. Автоматична синхронізація забезпечується періодичними запитами клієнта до серверу для оновлення локальних даних користувача.

3.2.5 Алгоритми обробки нагадувань та сповіщень

Нагадування та сповіщення про важливі події та завдання важливі для забезпечення вчасного виконання користувачем своїх обов'язків. Користувач налаштовує параметри нагадувань для завдань або подій, які зберігаються у локальній базі даних або на сервері. Система автоматично генерує та планує сповіщення на основі цих налаштувань, перевіряючи, чи настав час для сповіщення перед його відправленням. Користувач отримує сповіщення та може взаємодіяти з ними, наприклад, відкладати або відміняти нагадування. Синхронізація нагадувань між

пристроями забезпечується зберіганням даних у централізованому джерелі та періодичною синхронізацією клієнтських пристроїв.

3.2.6 Алгоритми оптимізації продуктивності застосунку

Оптимізація продуктивності застосунку важлива для забезпечення ефективної роботи, зменшення витрат ресурсів та покращення користувацького досвіду. Кешування даних дозволяє зменшити час відповіді, зберігаючи часто використовувані дані локально. Оптимізація запитів до бази даних включає використання індексів та оптимізацію структури запитів. Асинхронність та паралелізм дозволяють одночасно обробляти багато запитів без блокування потоку виконання. Мінімізація обсягу передаваних даних та оптимізація завантаження ресурсів зменшують кількість запитів до сервера та обсяг передаваних даних. Моніторинг та аналіз продуктивності дозволяють виявляти проблеми та оптимізувати систему.

3.2.7 Алгоритми роботи з базою даних PostgreSQL

Алгоритми роботи з базою даних PostgreSQL включають операції створення, читання, оновлення та видалення даних (CRUD) та оптимізацію запитів для забезпечення ефективності та швидкодії. Створення записів передбачає формування SQL-запиту для вставки нового запису та виконання запиту методом INSERT. Отримання записів здійснюється через формування та виконання SQL-запиту SELECT для вибору даних з бази. Оновлення записів включає формування та виконання SQL-запиту UPDATE для зміни даних. Видалення записів відбувається через формування та виконання SQL-запиту DELETE. Оптимізація запитів включає використання індексів, оптимізацію структури запитів та коректне використання транзакцій для забезпечення цілісності та безпеки даних.

3.3 Послідовність роботи системи

Першим кроком у розробці програмного забезпечення є розробка Business Process Model and Notation (BPMN) діаграми, яка відображає бізнес-процеси та взаємозв'язки між ними. У цьому підпункті ми розробляємо детальну діаграму бізнес-процесів, яка охоплює всі основні функції та взаємодії у межах нашого мобільного застосунку з функціями ділового органайзера.

BPMN діаграма допомагає зрозуміти послідовність дій, взаємозв'язки між різними модулями та взаємодію користувача з додатком. Вона є важливим інструментом для визначення потреб бізнесу та визначення ключових функціональних вимог до програмного забезпечення. Розробка BPMN діаграми допомагає уникнути недорозумінь та недоліків у сприйнятті вимог до програми та дозволяє забезпечити більшу ясність у процесі розробки. Розглянемо BPMN діаграму яка відображає

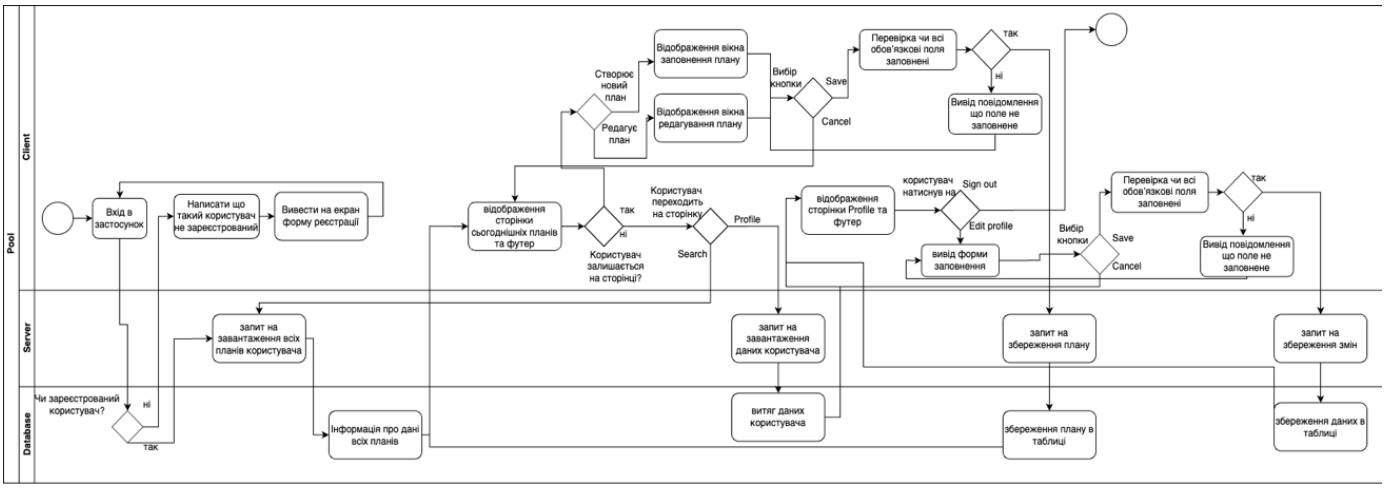

Рисунок 3.3.1 – BPMN діаграма

3.3 Структура бази даних

У цьому підпункті ми докладно описуємо структуру бази даних нашого мобільного застосунку з функціями ділового органайзера. Це включає в себе всі таблиці, поля, зв'язки між ними та основні сутності даних.

Структура бази даних є ключовим елементом програмного забезпечення, оскільки вона визначає, як дані будуть організовані та зберігатися. Це важливо для забезпечення ефективності та надійності додатку, а також для забезпечення легкості розширення та обслуговування в майбутньому.

База даних містить 3 таблиці Plans, Users та migrations. Структуру таблиць представлено на малюнку 3.3.1

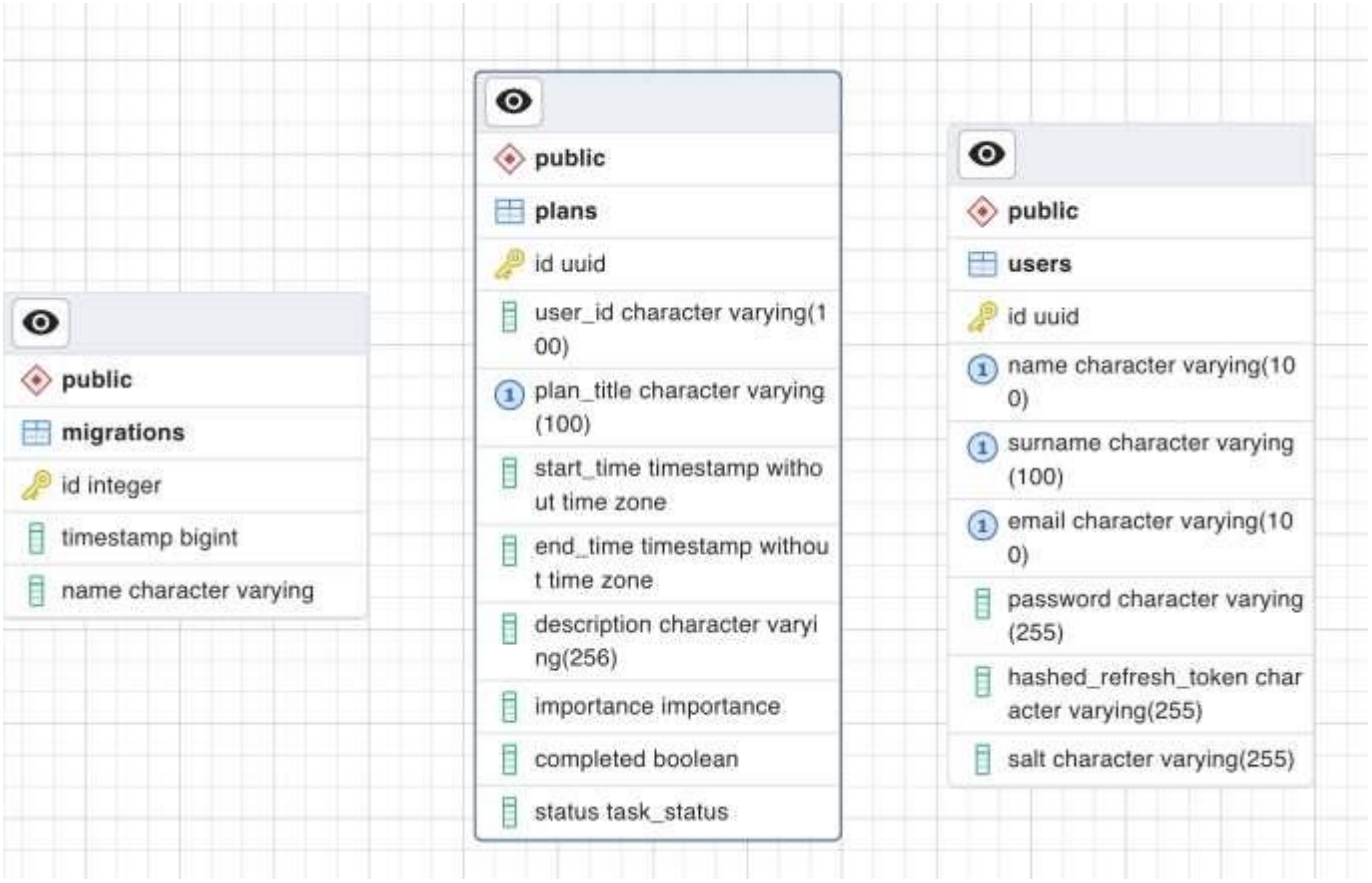

Рисунок 3.3.1 – Структура бази даних

Нижче наведено опис таблиць баз даних, та пояснено які дані зберігаються в кожній таблиці:

1. Таблиця Users містить дані про всіх зараєстрованих користувачів у додатку. У таблиці 7 полів: id, name, surname, email, password,

hashed_refresh_token, salt. Id – первинний ключ.

2. Таблиця Plans містить 9 полів: id, user_id, plan_title, start_time, end_time, description, importance, completed, status. Id – первинний ключ. Для поля importance було створено власний тип даних який може мати значення тільки low, high та normal. Для поля status було створено власний тип даних який може мати значення тільки: overdue, active, upcoming.

3. Таблиця migrations містить 3 полів: id, timestamp, name. Ця таблиця генерувалась автоматично за допомоги TypeORM, при генерації міграцій, timestamp – це час коли міграція для таблиці була створена, name – назва таблиці.

4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

У цьому розділі розглядається взаємодія користувача з мобільним застосунком з функціями ділового органайзера. Успіх будь-якого програмного продукту значною мірою залежить від того, наскільки зручною та інтуїтивно зрозумілою є робота з ним для кінцевого користувача. Тому важливо не лише реалізувати функціональні можливості, але й забезпечити високий рівень зручності та задоволеності користувачів.

4.1 Системні вимоги

Перш ніж користувачі почнуть використовувати мобільний застосунок з функціями ділового органайзера, вони повинні впевнитися, що їхні пристрої відповідають системним вимогам для безперебійної роботи програми. Нижче наведені основні системні вимоги:

1. Платформа: Мобільний застосунок розробляється для операційних систем iOS та Android. Тому користувачі повинні мати пристрої на базі цих операційних систем.
2. Версія операційної системи: Для iOS застосунок підтримується на версіях iOS 11 і вище, а для Android - на версіях Android 6.0 (Marshmallow) і вище.
3. Ресурси пристрою: Пристрої користувачів повинні мати достатньо ресурсів, таких як процесор, оперативна пам'ять і внутрішній сховище, для встановлення та виконання застосунку з комфортом.
4. Інтернет-з'єднання: Деякі функції застосунку можуть вимагати доступу до Інтернету, тому користувачам потрібно мати можливість підключитися до мережі Wi-Fi або мобільного Інтернету.

Перед встановленням застосунку користувачам рекомендується перевірити, чи відповідають їхні пристрої вказаним системним вимогам для оптимальної роботи програми.

4.2 Інсталяція проєкту

Інсталяція мобільного застосунку з функціями ділового органайзера є простим процесом, що дозволяє розробникам швидко запустити проєкт у середовищі розробки та підготувати його до тестування. Нижче наведені кроки для інсталяції проєкту:

1. Встановлення необхідних інструментів.

Переконайтеся, що у вас встановлено Xcode (якщо використовуєте macOS) та Expo CLI. Xcode потрібен для запуску симулятора iOS. - Для встановлення Expo CLI, виконайте команду – npm install -g expo-cli

2. Клонування репозиторію з GitHub.

Спочатку клонувати репозиторій вашого проєкту з GitHub. Виконайте команду – git clone https://github.com/lilevg/PlanApp. Потім перейдіть до директорії проєкту – cd your-project

3. Встановлення залежностей.

Виконайте команду для встановлення всіх необхідних залежностей – npm install

4. Запуск Expo.

Для запуску проєкту в режимі розробки за допомогою Expo, виконайте команду – npx expo start.

5. Запуск симулятора iOS.

В Expo Developer Tools, які відкриються у вашому браузері, натисніть "Run on iOS simulator". Для цього переконайтеся, що у вас встановлено Xcode і запущений iOS Simulator.

6. Запуск на Android.

Для запуску проєкту на Android, натисніть "Run on Android device/emulator" в Expo Developer Tools. Переконайтеся, що у вас встановлено Android Studio та запущений емулятор.

7. Налаштування середовища.

Для коректної роботи бекенду переконайтеся, що ваш сервер, створений за допомогою NestJS, запущений та доступний для мобільного застосунку. Переконайтеся, що підключення до бази даних PostgreSQL також налаштоване та працює.

8. Тестування та розробка.

Після запуску застосунку на симуляторі або фізичному пристрої, ви можете тестувати функціональність, відстежувати помилки та проводити подальшу розробку. Всі зміни в коді будуть автоматично відображатися на пристрої завдяки функції hot reloading, яку забезпечує Expo.

Ці кроки допоможуть вам налаштувати середовище розробки та розпочати роботу з мобільним застосунком з функціями ділового органайзера. Після завершення налаштування, ви будете готові до тестування та подальшої розробки вашого проєкту.

4.3 Сценарій взаємодії користувача з інтерфейсом

Взаємодія з інтерфейсом мобільного застосунку з функціями ділового органайзера відбувається через інтуїтивно зрозумілі елементи керування, які надають користувачам зручний доступ до всіх функціональних можливостей програми. Нижче наведений загальний сценарій взаємодії користувача з інтерфейсом застосунку:

1. Авторизація або реєстрація: У разі першого запуску або в разі виходу з облікового запису, користувачу може бути запропоновано авторизуватися або зареєструватися, введучи відповідні дані, Рисунок 4.3.1

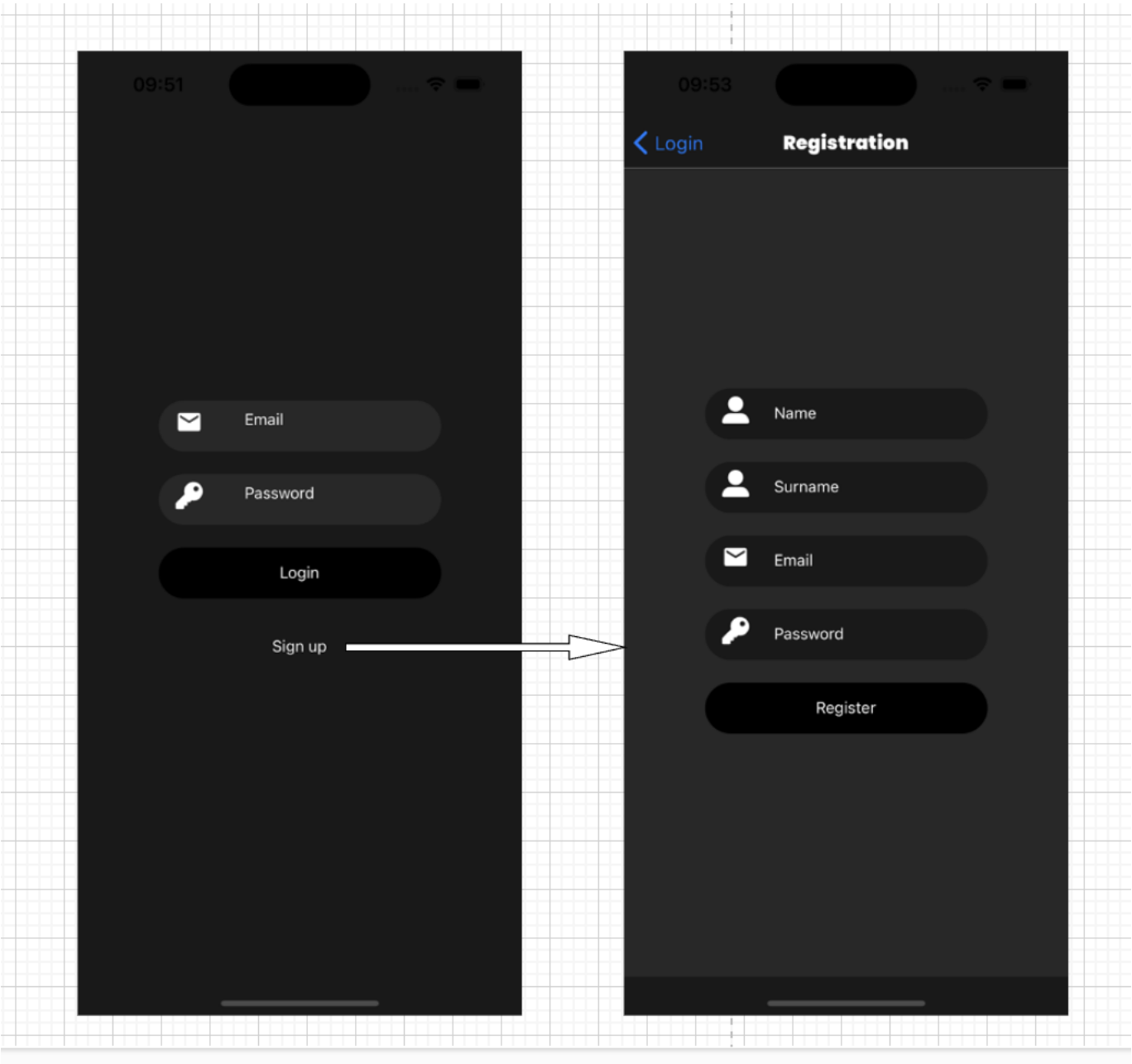

Рисунок 4.3.1 – Сторінка входу та реєстрації

3. Основний екран: Після авторизації користувач потрапляє на основний екран застосунку, де відображаються основні функціональні блоки, такі як список завдань, календар, нагадування тощо, Рисунок 4.3.3

Рисунок 4.3.2 – Сторінка екрану з сьогоднішніми планати

4. Взаємодія з функціями: Користувач може взаємодіяти з різними функціями застосунку шляхом натискання на відповідні елементи інтерфейсу. Наприклад, відкривши список завдань, він може створювати нові завдання, написнувши на сам план може редагувати чи видаляти існуючі. А написнувши на ‘+’ користувач може створити новий план.

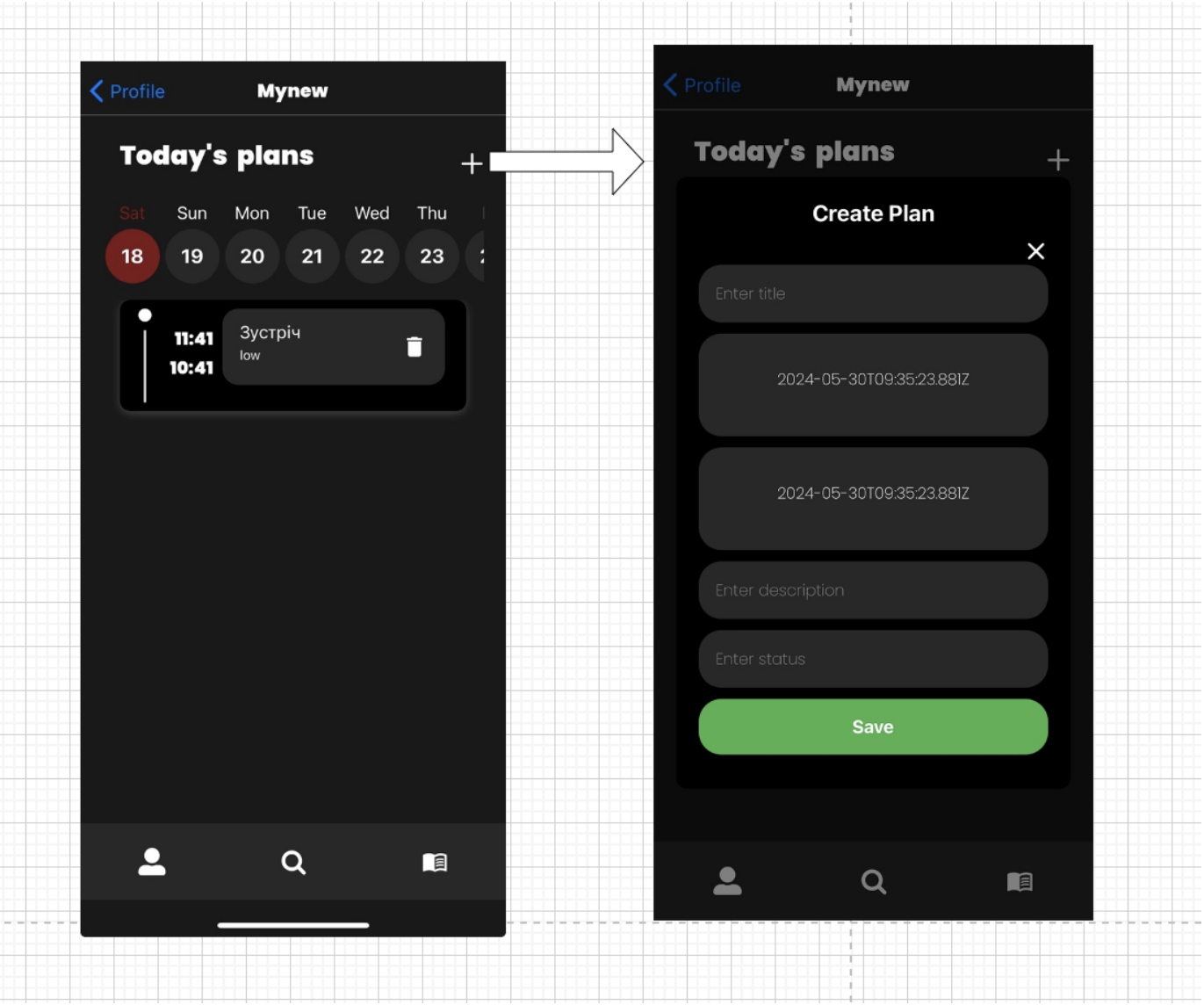

Рисунок 4.3.3 – Форма створення нового плану

5. Навігація: Для переходу між різними розділами та функціями застосунку користувач може використовувати навігаційне меню, яке може знаходиться знизу екрана, що можна побачити на попередніх скріншотах застосунку. В меню можна перейти на сторінки профілю та пошуку.

6. Сторінка пошуку надає можливість пошуку всіх планів, створення нового плану, видалення та редагування елементу, а також пошуку за фільтрами.

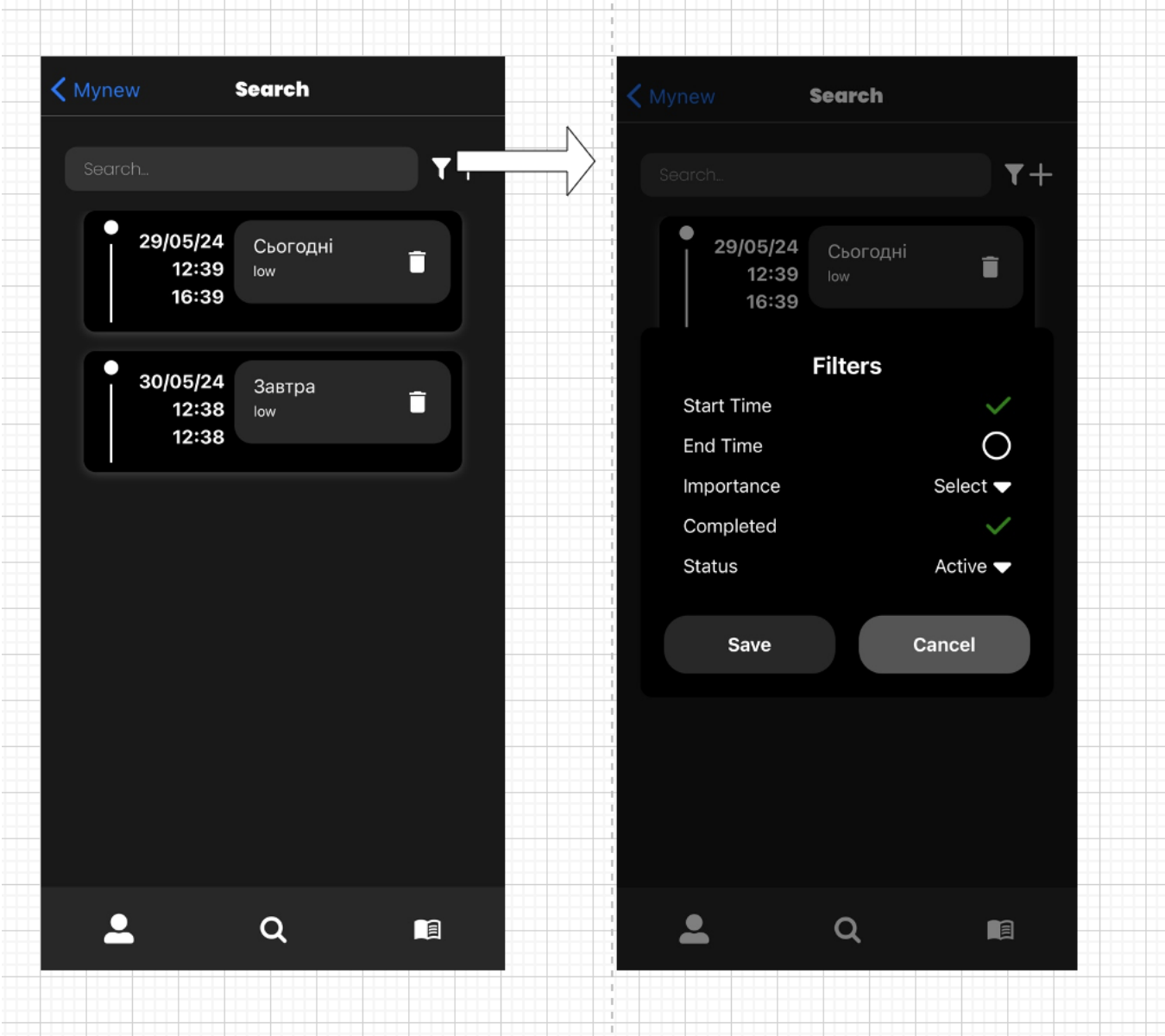

Рисунок 4.3.4 – Вікно вибору фільтрів

7. На сторінці профілю можна переглянути власні далі, редагувати їх та вийти з застосунку.

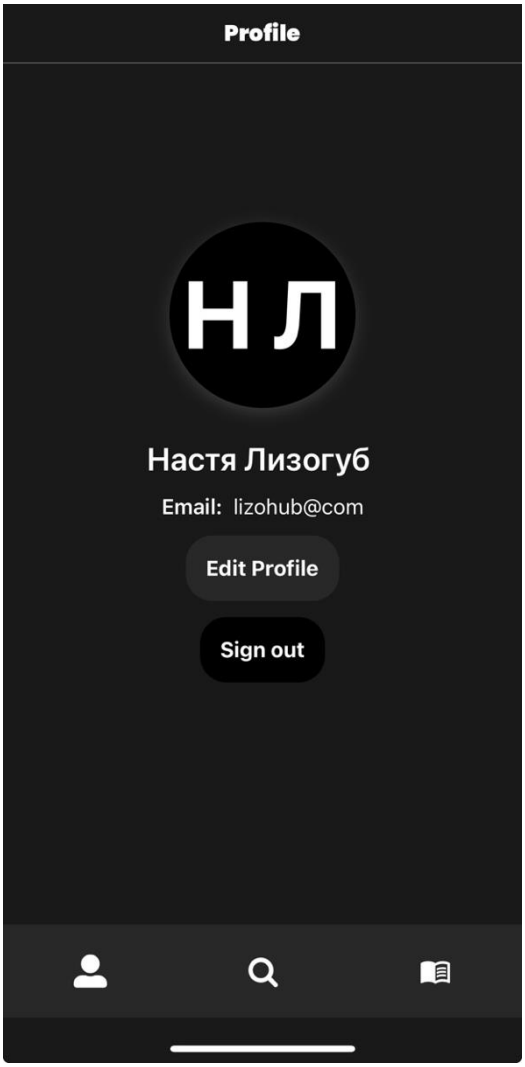

Рисунок 4.3.5 – Сторінка профілю

На сторінці профілю користувач може переглядати свої особисті дані, такі як ім'я, електронну пошту та інші контактні відомості. Інтерфейс також надає можливість редагувати ці дані, щоб підтримувати їх в актуальному стані. Користувачі можуть змінювати свої налаштування та оновлювати інформацію в реальному часі. Крім того, на сторінці профілю передбачено кнопку для виходу з облікового запису, що забезпечує безпеку та конфіденційність особистих даних. Інтуїтивно зрозумілий дизайн сторінки профілю дозволяє користувачам легко орієнтуватися та виконувати необхідні дії без зайвих зусиль.

ВИСНОВКИ

На основі аналізу підходів, методів та алгоритмів розв’язання поставленої задачі обґрунтовано використання JavaScript для реалізації мобільного застосунку з функціями ділового органайзера. JavaScript забезпечує широкі можливості розробки мобільних додатків та підтримується відомими фреймворками, такими як React Native, що сприяє швидкій та ефективній розробці з високим рівнем переносимості коду.

На основі аналізу програмних засобів обґрунтовано використання фреймворків NestJS та React Native для розробки мобільного застосунку. NestJS забезпечує потужний інструментарій для побудови надійних та масштабованих серверних додатків, тоді як React Native дозволяє створювати мобільні додатки для iOS та Android з використанням одного коду на JavaScript.

Розроблено програмну систему "Мобільний застосунок з функціями ділового органайзера". Результатом роботи є мобільний застосунок з функціями ділового органайзера, який дозволяє користувачам ефективно керувати своїми завданнями та подіями.

На основі тестування доведено коректність роботи розробленого програмного забезпечення. Виконано ретельне тестування застосунку на різних пристроях та в різних умовах для забезпечення його стабільної та надійної роботи.

Отже, розроблений застосунок забезпечує користувачам зручний інструмент для управління щоденними завданнями та бізнес-процесами, підвищуючи їхню продуктивність та організованість.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. WebStorm The JavaScript and TypeScript IDE URL: https://www.jetbrains.com/webstorm/
2. Microservice architecture. Website URL: https://www.atlassian.com/microservices/microservices-architecture
3. Comparing Database Management Systems: MySQL, PostgreSQL, MSSQL Server, MongoDB, Elasticsearch, and others URL: https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/
4. Microservices. Website URL: https://martinfowler.com/articles/microservices.html
5. Nest JS. Introduction URL: https://docs.nestjs.com/
6. Node JS. Run JavaScript Everywhere URL: https://nodejs.org/en
7. Node JS. Node.js Tutorial URL: https://www.geeksforgeeks.org/nodejs/
8. Mobile App URL: https://medium.com/@vitalka057/mobile-app-61c3acb3a093
9. Database migrations with Type ORM: https://wanago.io/2022/07/25/api-nestjs-database-migrations-typeorm/
10. Type ORM. Website URL:https://docs.nestjs.com/recipes/sql-typeorm
11. Type ORM. Website URL: https://typeorm.io/
12. Postgres. Website URL: https://www.postgresql.org/
13. React native. Website URL: https://react.dev/
14. React Native push notifications: A complete how-to guide URL: https://blog.logrocket.com/react-native-push-notifications-complete-guide/
15. Expo Documentation URL: https://docs.expo.dev/
16. Docker Documentation URL: https://docs.docker.com/

Додаток А

Програмні засоби:

– мова програмування – JavaScript;

– сервіс по роботі з планами

```
import {
  ConflictException,
  Injectable,
  InternalServerErrorException,
  NotFoundException,
} from '@nestjs/common';
import { CreatePlanDto } from './dto/create-plan.dto';
import { UpdatePlanDto } from './dto/update-plan.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository, Like } from 'typeorm';
import { HttpService } from '@nestjs/axios';
import { Plan } from './entities/plan.entity';
import { PlansFiltersDto } from './dto/plans-filters.dto';

@Injectable()
export class PlanService {
  constructor(
    @InjectRepository(Plan) private readonly planRepository: Repository<Plan>,
    private readonly httpService: HttpService,
  ) {}
  async create(createPlanDto: CreatePlanDto, userId: string): Promise<Plan> {
    const user = this.getUser(userId);
```

```
if (!user) {
  throw new ConflictException('Username already exists');
}
const title = createPlanDto.plan_title;
const existThisPlanTitle = await this.planRepository.findOne({
  where: { plan_title: title },
});
if (existThisPlanTitle) {
  const similarTitles = await this.planRepository.find({
    where: [
      { plan_title: Like(`${title} %`) },
      { plan_title: Like(`${title}%`) },
    ],
  });

  similarTitles.sort((a, b) => {
    const matchA = a.plan_title.match(/\((\d+)\)$/);
    const matchB = b.plan_title.match(/\((\d+)\)$/);

    const numA = matchA ? +matchA[1] : 0;
    const numB = matchB ? +matchB[1] : 0;

    return numB - numA;
  });

  const lastTitle = similarTitles[0];
  const regex = /.*\((\d+)\)$/;
  const match = lastTitle.plan_title.match(regex);
```

```
    let number = 0;

    if (match && match.length > 1) {
      number = +match[1];
    }

    const editTitle = `${title} (${++number})`;
    const newPlan1 = this.planRepository.create({
      plan_title: editTitle,
      description: createPlanDto.description,
      userId: userId,
    });
    const plan1 = await this.planRepository.save(newPlan1);
    return plan1;
  }
  const newPlan = this.planRepository.create({
    ...createPlanDto,
    userId: userId,
  });
  const plan = await this.planRepository.save(newPlan);
  return plan;
}

async findAllUsersPlans(userId: string) {
  const user = await this.getUser(userId);
  if (!user) {
    throw new ConflictException('Username already exists');
  }
```

```
  const plans = await this.planRepository.find({
    where: { userId },
    order: { plan_title: 'DESC' },
  });
  return plans;
}

async findOne(userId: string, planId: string): Promise<Plan> {
  const user = this.getUser(userId);
  if (!user) {
    throw new ConflictException('Username already exists');
  }
  return await this.planRepository.findOne({
    where: { userId: userId, id: planId },
  });
}

async update(planId: string, userId: string, updatePlanDto: UpdatePlanDto) {
  const plan = await this.planRepository.findOne({
    where: { id: planId, userId },
  });

  if (!plan) {
    throw new NotFoundException(`Plan with ID ${planId} not found`);
  }

  if (updatePlanDto.plan_title) {
    plan.plan_title = updatePlanDto.plan_title;
```

```
  }
  if (updatePlanDto.start_time) {
    plan.start_time = updatePlanDto.start_time;
  }
  if (updatePlanDto.end_time) {
    plan.end_time = updatePlanDto.end_time;
  }
  if (updatePlanDto.description) {
    plan.description = updatePlanDto.description;
  }
  if (updatePlanDto.importance) {
    plan.importance = updatePlanDto.importance;
  }
  if (updatePlanDto.completed !== undefined) {
    plan.completed = updatePlanDto.completed;
  }

  return this.planRepository.save(plan);
}

async remove(id: string, userId: string): Promise<void> {
  const user = this.getUser(userId);
  if (user) {
    await this.planRepository.delete(id);
  } else {
    throw new ConflictException('You can`t delete not yours plans');
  }
}
```

```
private async getUser(userId: string): Promise<any> {
  try {
    const response = await this.httpService
      .get(`http://localhost:8000/user/${userId}`)
      .toPromise();
    return response.data;
  } catch (error) {
    console.error('Error fetching user data:', error.response.data);
    throw new InternalServerErrorException();
  }
}

async filteredPlans(userId: string, filters: PlansFiltersDto) {
  const { importance, completed, status } = filters;

  const queryBuilder = this.planRepository.createQueryBuilder('plan');

  queryBuilder.where('plan.userId = :userId', { userId });

  if (importance) {
    queryBuilder.andWhere('plan.importance = :importance', { importance });
  }

  if (completed !== undefined) {
    queryBuilder.andWhere('plan.completed = :completed', { completed });
  }
```

```
    if (status) {
      queryBuilder.andWhere('plan.status = :status', { status });
    }

    return await queryBuilder.getMany();
  }
}
```