МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Інститут післядипломної освіти

Кваліфікаційна робота бакалавра

на тему: «Розробка веб-застосунку для оптимізації функціонування міської інфраструктури екологічного транспорту»

Виконала студентка групи КН-5
спеціальності 122 Комп'ютерні науки
Козлова Анастасія Олексіївна

Керівник Фразе-Фразенко О.О.,
канд. техн. наук, доцент

Консультант

Рецензент Т.в.о.директора КП
"Обласний інформаційно-аналітичний
центр", Попов В.Л.

Одеса 2023

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

IT − інформаційні технології,
БД − база даних,
ПК − персональний комп’ютер,
UML − Unified Modeling Language, уніфікована мова моделювання,
HTML − HyperText Markup Language, це мова розмітки гіпертексту на сторінках у браузері,
CSS − Cascading Style Sheets, каскадні таблиці стилів, це спеціальна мова, за допомогою якої описують вигляд HTML-документів
TS − мова програмування TypeScript,
JS − мова програмування JavaScript,
ООП − об'єктно-орієнтоване програмування,
UI − user interface, інтерфейс користувача,
UX − user experience, досвід користувача,
JSX − JavaScript XML, HTML-подібний синтаксис для опису структури інтерфейсу,
XML − eXtensible Markup Language, мова розмітки,
DOM − Document Object Model, об'єктна модель документа,
HTTP − HyperText Transfer Protocol, протокол передачи гипертекста,
ORM − Object Relational Mapping, об'єктно-реляційна проєкція,
SQL − Structured query language, мова структурованих запитів,
СУБД − система управління базами даних,
SQL − Structured Query Language, мова структурованих запитів,
FE − Front-end,
BE − Back-end,
ER − Entity relational, реляційна модель сутностей,
JWT − JSON Web Token, стандарт для створення токенів
JSON − JavaScript Object Notation, текстовий формат обміну даними

ВСТУП

Останні досягнення в галузі інформаційних технологій відкривають багато можливостей для розвитку та вдосконалення різних сфер нашого життя: завдяки IT ми можемо швидко та ефективно обмінюватися інформацією, знаходити потрібні нам ресурси та розробляти нові ідеї.

Активний спосіб життя стає невід'ємною частиною життя багатьох людей, і їзда на велосипеді займає в ньому особливе місце. Велосипед не тільки дозволяє людям насолоджуватися природою, але й є екологічно чистим та економічно вигідним видом транспорту. Його популярність постійно зростає, а разом з нею і потреба у відповідній інформації для безпечного та комфортного пересування, для чого потрібен єдиний інформаційний простір у цій сфері.

Темою дипломної роботи є «Розробка веб-застосунку для оптимізації функціонування міської інфраструктури екологічного транспорту». Тема актуальна та має значний потенціал у покращенні екологічної ситуації в містах та сприянні здоровому способу життя громадян які використовують велосипеди.

Основною метою проекту є створення зручного та інтуїтивно зрозумілого додатку з функціоналом, який надає велосипедистам доступ до актуальної інформації про велосипедний світ та надає корисні дані для планування велосипедних маршрутів. Веб-додаток, повинен мати інтуїтивно зрозумілий та привабливий інтерфейс, що дозволяє велосипедистам швидко знаходити потрібну інформацію та взаємодіяти з потрібними даними.

Розробка цього додатку має великі можливості для масштабування та впровадження нових ідей для поширення велосипедного транспорту. Таким чином, розробка веб-додатку для оптимізації функціонування екологічної транспортної інфраструктури міста є важливим кроком у розвитку велосипедного руху та велосипедної культури.

У цілому, основними цілями проекту є:

- створення карти з велодоріжками міста Одеса;
- функціонал для надання користувачам інформації про велосипедні маршрути та потрібні місця на карті;
- функціонал надання інформації про новини та події, пов'язані з велосипедами;
- покращення навичок розробки набутих у університеті;
- вивчення та опанування затребуваних технологій на сучасному IT ринку.
- створення сучасного продукту який легко масштабувати у майбутньому

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Проблематика велосипедної інфраструктури та потреби користувачів

Катання на велосипеді є популярним видом активного відпочинку та транспорту для багатьох людей. Однак, існують певні проблеми і недоліки, з якими стикаються велосипедисти, особливо в містах.

Використання велосипеду може вирішити багато питань. Проблеми, на вирішення яких спрямований розвиток молодіжного спорту та створення велосипедної інфраструктури, зокрема у м. Одесі [11]:

1. Екологічна проблема.

2. Проблема мобільності громадян та гостей міста.

3. Проблема підвищення туристичної привабливості міста.

4. Проблема популяризації активного, здорового способу життя громадян міста і занять спортом

Нижче наведено основні проблеми та потреби, які виникають у спільноті велосипедистів:

1. Відсутність централізованої інформації про велодоріжки: У багатьох містах, включаючи Одесу, немає достатньої інформації щодо всіх наявних велосипедних шляхів. Це створює проблеми для велосипедистів, які бажають планувати свої маршрути та використовувати безпечні велосипедні доріжки.

2. Відсутність інформаційного порталу: Велосипедистам важлива актуальна інформація про новини, події та розвиток велосипедного спорту, будівництво нових велодоріжок та законодавчі ініціативи, спрямовані на поліпшення інфраструктури. Але на сьогоднішній день не існує централізованого ресурсу, де велосипедисти можуть знайти всю цю інформацію.

3. Розрізненість велосипедних шляхів: В багатьох містах велосипедні шляхи будувалися різними організаціями та в різний час, що призвело до їх розрізненості та відсутності єдиного плану розвитку велосипедної

інфраструктури. Це ускладнює переміщення велосипедистів та зменшує їх безпеку на дорозі.

4. Відсутність даних про сервіси та парковки для велосипедів, про інші корисні місця: Для велосипедистів дуже важливо мати інформацію про наявність сервісних центрів та місць для паркування велосипедів. Однак, в багатьох містах, зокрема в Одесі, немає централізованої бази даних, яка б містила інформацію про такі сервіси та парковки. Це ускладнює планування маршрутів та забезпечення зручності для велосипедистів.

Ураховуючи вищезазначені проблеми та потреби велосипедистів, веб-застосунок, який розробляється в рамках даного дипломного проекту, повинен бути спрямований на вирішення деяких з цих проблем. Для цього додаток повинен мати потрібний функціонал, який буде сприяти поліпшенню вирішення цих питань.

1.2 Визначення вимог до роботи веб-застосунку

Узагальнюючи проблеми у користувачів треба представити функціонал для вирішення цих проблем. Як і для будь-якого проекту, важливим етапом розробки є визначення вимог (функціональних та не функціональних), яке дозволяє за планом реалізувати весь зазначений на цьому етапі функціонал для додатку.

Функціональні вимоги визначають можливості програмного забезпечення, що будуть створені розробниками для вирішення задач користувачів, які визначені в бізнес вимогах до інформаційної системи [10].

На додаток до функціональних вимог виявляються нефункціональні вимоги, які описують зовнішні взаємодії між системою і зовнішнім світом, а також обмеження дизайну та реалізації [10].

Основні кроки які потрібно реалізувати: авторизація, права доступу, новини та взаємодія з ними, карта та взаємодія з нею, налаштування. Розробити веб-додаток з таким функціоналом можливо за допомогою клієнт-

серверної архітектури. Серверна частина (БД) повинна зберігати інформацію щодо користувачів, новин, велодоріжок та маркерів на карті, а клієнтська реалізувати зручний інтерфейс.

Розробка веб-застосунку для велосипедистів включає наступні функціональні вимоги:

1. Авторизація та права доступу:

− Веб-застосунок повинен забезпечувати авторизацію користувачів, що дозволяє їм виконувати дії в системі під своїм обліковим записом.

− Система повинна підтримувати два типи авторизації: звичайну (з використанням електронної пошти та пароля) і авторизацію через обліковий запис Google для швидкої авторизації.

− Визначення рівнів доступу користувачів, що дозволяє обмежити їх можливості в системі залежно від їх ролі.

2. Сторінка з новинами:

− Розроблений веб-застосунок повинен мати сторінку, на якій користувачі можуть переглядати новини.

− Користувачі з правами редагування можуть додавати нові новини, видаляти існуючі або редагувати інформацію в новинах.

− Користувач може знайти потрібні новини у пошуку

− При збільшенні кількості новин потрібна навігація за сторінками

3. Сторінка з картою:

− Веб-застосунок повинен мати сторінку з інтерактивною картою, на якій відображаються велодоріжки та маркери з місцями.

− Користувачі з відповідними правами можуть додавати нові велодоріжки та маркери на карту, редагувати їх та видаляти.

− Користувачі без спеціальних прав доступу можуть переглядати карту та інформацію на ній.

4. Налаштування користувача:

− Веб-застосунок повинен мати сторінку налаштувань, де користувачі можуть змінювати свій пароль для авторизації.

− Користувачі повинні мати можливість змінити свій обліковий запис та особисту інформацію (зміна паролю, фамілії тощо).

Не функціональні вимоги:

− Веб-застосунок повинен бути зручним у використанні з зрозумілим інтерфейсом.

− Веб-застосунок повинен мати ефективну роботу з базою даних для зберігання та керування інформацією, забезпечуючи швидкий доступ до даних що прискорює роботу додатка.

− Мати можливість до масштабування у майбутньому.

− Веб-застосунок повинен забезпечувати захист від несанкціонованого доступу або зловживання. Для цього можуть використовуватися механізми аутентифікації, авторизації, шифрування та інші заходи безпеки даних.

− Веб-застосунок повинен бути здатним працювати з ростом обсягу даних та навантаження. Він повинен бути легко масштабованим і гнучким, щоб відповідати потребам зростаючої користувацької бази.

− Веб-додаток повинен відповідати відповідним стандартам розробки, які сприяють якості, зручності та стабільності програмного забезпечення.

Для того щоб чітко розуміти що може і не може робити користувач треба зазначити які типи користувачів є у системі додатку. Можна передбачити що будуть потрібні ролі власника, редактора новин, редактора карти та звичайного користувача. Це можна представити у вигляді UML-діаграми.

UML (Unified Modeling Language) — уніфікована мова моделювання, що використовується розробниками програмного забезпечення для візуалізації процесів та роботи систем. Він допомагає в описі та проектуванні програмних систем, в особливості систем, побудованих з використанням об'єктно-орієнтованих технологій [7].

Прецеденти – це технологія визначення функціональних вимог до системи. Робота прецедентів полягає в описі типових взаємодій між

користувачами системи та самою системою та надання опису процесу її функціонування [7].

Діаграма варіантів використання — (діаграма прецедентів, сценарій використання, use-case) — дозволяє уявити типи ролей та їх взаємодію із системою. Проте не показує порядок виконання кроків. Зображує функціональні вимоги (те, що система може зробити) з точки зору користувача.

Порядок побудови діаграми наступний [10]:

1. Виділити групи осіб, які використовують систему.

2. Ідентифікувати варіанти використання системи.

3. Для кожного прецеденту розробити сценарій з описом.

Діаграму прецедентів для системи веб-додатку можем розглянути на Рис. 1.1.

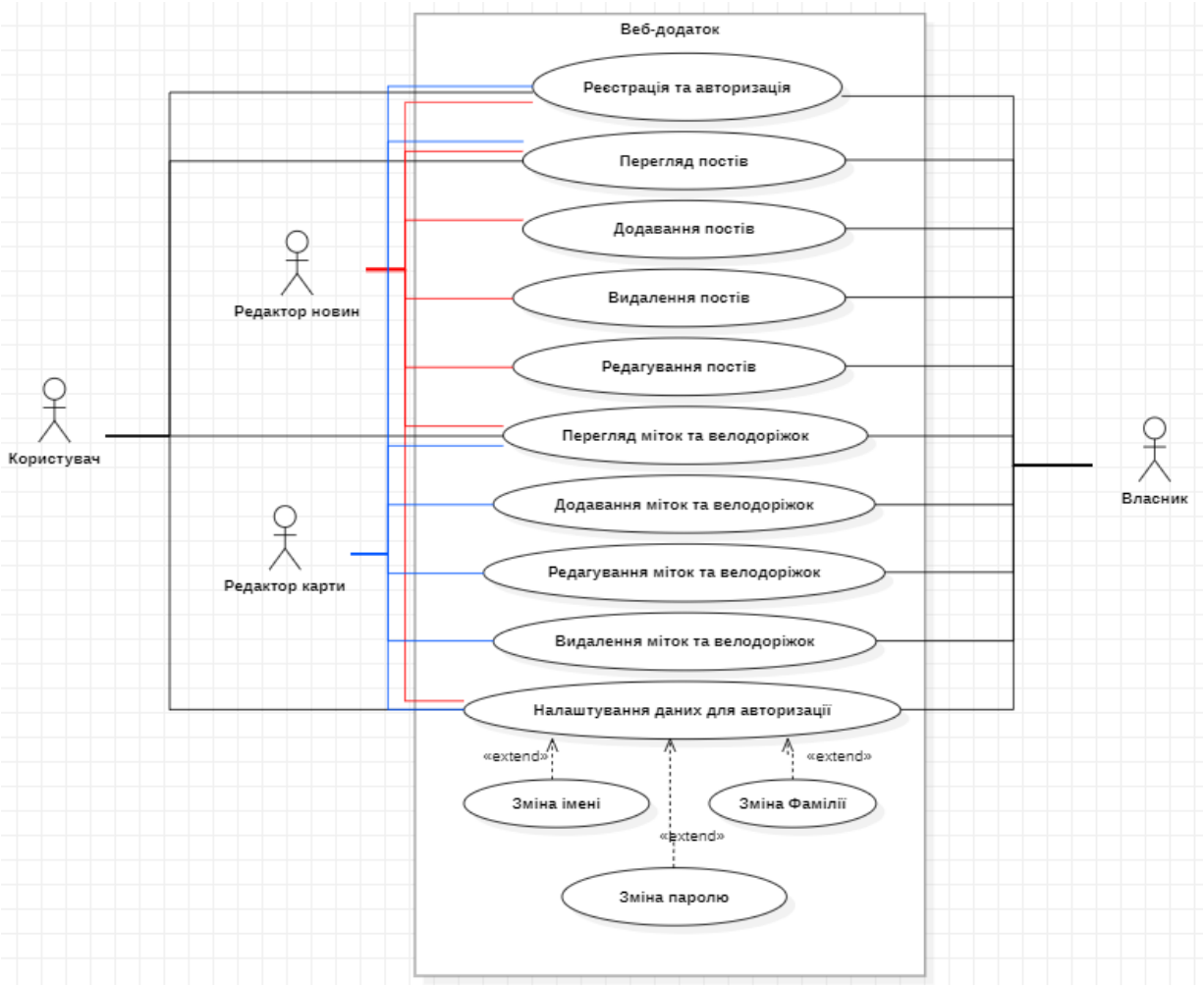

Рисунок 1.1 – Діаграма прецедентів

Виходячи з зазначених вимог для узагальнення створимо список необхідних до реалізації функцій (Табл. 1.1).

Таблиця 1.1 – Список функцій додатку до реалізації

Номер	**Частина функціоналу**	**Фунції до реалізації**
1.	Головна сторінка	Звичайна авторизація
		Авторизація через Google
2.	Загальне меню, навігація	Меню з новинами, мапою та налаштуваннями
3.	Новини	Перегляд новин
		Створення новин
		Редагування новин
		Видалення новин
		Пошук новин
4.	Мапа	Загальна мапа з переглядом
		Створення маркерів та велодоріжок
		Редагування маркерів та велодоріжок
		Видалення маркерів та велодоріжок
		Можливість скривати маркери та велодоріжки
5.	Загальне за проектом	Налаштування доступу до функцій додатку в залежності від ролі користувача
		Зміна мови у додатку (українська/англійська)

1.3 Важливість інтерфейсу користувача. Концепція Material Design

Інтерфейс користувача (UI) є ключовим аспектом будь-якого програмного забезпечення або веб-додатку, оскільки це є прямим способом взаємодії користувача з системою. Важливість UI полягає в тому, що він визначає, як користувачі будуть сприймати та взаємодіяти з додатком. Добре розроблений інтерфейс користувача допомагає забезпечити зручність, ефективність, задоволення та задоволення користувачів при використанні продукту.

UI (User Interface, інтерфейс користувача) - це те, як виглядає та працює інтерфейс користувача. Це включає в себе елементи дизайну, такі як кольори, типографіка, макети, кнопки та інші елементи, які взаємодіють з користувачем. Завдання UI-дизайну — створення не просто ефектних іконок та красивих кнопок, а інтуїтивно зрозумілих та логічно пов’язаних з іншими елементами інтерфейсу [1].

Основні характеристики UI-дизайну [1]:

1. Використовується тільки для роботи з цифровими продуктами.

2. Основна увага — користувацьким елементам.

3. Націлений на створення ідеальної комбінації шрифтів, кольорової палітри, форм, анімації, зображень та інших візуальних елементів.

4. Головне завдання — отримати продукт, який здатний задовольнити естетичні уподобання користувачів.

UX (User Experience, користувацький досвід) - це загальне враження користувача від взаємодії з продуктом або послугою. UX охоплює всі аспекти взаємодії користувача, включаючи дизайн інтерфейсу, навігацію, швидкість роботи, зручність використання та задоволення, яке відчуває користувач. UX-дизайн працює над тим, що клієнту було зручно й легко взаємодіяти з сайтом, сервісом, додатком тощо [1].

Основні характеристики UX-дизайну [1]:

1. Використовується в цифрових та в аналогових продуктах.

2. Націлений на вивчення й аналіз користувацького досвіду клієнта (від початку знайомства з продуктом до фінальної взаємодії з ним).

3. Працює над структурою майбутнього продукту (сайту/сервісу/додатку) та над спрощенням взаємодії клієнта з ним;

4. Головне завдання — отримати цифровий продукт, який розв’язує проблеми користувачів.

Material Design - це концепція дизайну інтерфейсу, розроблена компанією Google. Вона пропонує вказівки та рекомендації щодо створення сучасних, естетичних та функціональних інтерфейсів користувача. Концепція

Material Design базується на реальних об'єктах та їх фізичних властивостях, використовує тіні, анімацію, кольорові палітри та інші елементи.

Розповсюдженість дизайну від Google зробила цей дизайн зрозумілим та звичним для будь-якого користувача. Дизайн від гугл має свої бібліотеки для різних платформ. Його можна знайти і легко використовувати як у розробці сайтів так і у розробці мобільних або десктопних додатків. Це дає потенційну можливість легко масштабувати та адаптувати дизайн сайту на інші платформи у майбутньому. Також дизайн від гугл спрощує розробку та не потребує від розробника поглиблених знань у дизайні, що дозволяє зосередитись на розробці самого додатку.

2 ВИБІР ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ ТА АРХІТЕКТУРА ПРОЕКТУ

2.1 Клієнт-серверна архітектура

Клієнт-серверна архітектура є однією з фундаментальных концепцій у сучасному світі програмного забезпечення. Ця архітектура розподіляє функціональність та завдання між двома основними компонентами системи - клієнтом та сервером - для забезпечення ефективної комунікації та обробки даних.

Клієнт-серверна архітектура заснована на розділенні обов'язків між клієнтськими додатками(клієнтами) та серверами. Клієнт - це програмне забезпечення або пристрій, що звертається до сервера для отримання певної послуги або інформації. Сервер - це програмне забезпечення або пристрій, який надає запитані послуги або інформацію клієнту. Зв'язок між клієнтом і сервером відбувається через мережу, таку як інтернет. Тому для розробки веб-додатку нам потрібно розуміти взаємодію клієнт-сервер.

Основна ідея клієнт-серверної архітектури полягає у тому, що клієнтська програма виконує лише ту частину роботи, яка стосується користувацького інтерфейсу, взаємодії з користувачем та обробки простих обчислень. Серверна частина відповідає за обробку складних обчислень, доступ до бази даних, зберігання інформації та іншу функціональну логіку [9].

Основні переваги клієнт-серверної архітектури включають:

1. Розділення обов'язків: Клієнт і сервер мають чітко визначені обов'язки, що дозволяє розробникам зосереджуватись на своїх конкретних областях розробки програмного продукту та виконувати роботу окремо та частинами.

2. Масштабованість: Клієнтські додатки можуть бути розроблені для різних платформ та пристроїв що створює простір для розгортання системи та створення кросплатформних додатків.

3. Гнучкість розробки: Клієнт-серверна архітектура дозволяє розробникам змінювати та оновлювати клієнтські та серверні компоненти незалежно один від одного. Це полегшує розгортання нових функцій та виправлення помилок без необхідності змінювати всю систему одночасно. Що дуже добре сприяє на швидкість та зручність розробки.

4. Безпека: Можливість чіткішого розмежування повноважень доступу до різних рівнів інформаційної системи. Кожному клієнту – свій рівень доступу. Сервер може встановлювати політики безпеки, аутентифікувати користувачів, давати різні права і забезпечувати конфіденційність даних шляхом шифрування та інших заходів безпеки. Зазвичай сервер краще захищений від різного виду загроз, ніж звичайний клієнтський ПК або мобільний пристрій.

5. Швидкість та продуктивність: Клієнт-серверна архітектура дозволяє розподілити завдання між клієнтом та сервером, що сприяє ефективній роботі та зменшенню навантаження на окремі компоненти системи. Потужна серверна частина може виконувати більшу частину роботи, при мінімумі навантаження на клієнта. Це може покращити швидкість та продуктивність додатку.

6. Можливість розширення: Клієнт-серверна архітектура дозволяє легко розширювати функціональність системи шляхом додавання нових клієнтів або серверів. Це дає можливість адаптувати систему до зростаючих потреб користувачів або бізнесу. На прикладі проекту, може бути ще один клієнт який використовує базу велодоріжок. Або при популярності ресурсу можна добавляти безліч серверів для швидкості обробки запитів.

Основні недоліки клієнт-серверної архітектури включають:

1. Вихід з ладу сервера може призвести до непрацездатності всієї системи, яка його використовує.

2. Висока вартість серверного обладнання та його обслуговування (зокрема, може знадобитися окремий спеціаліст для обслуговування).

3. Високе навантаження на серверне обладнання та канал зв'язку до нього.

У сучасному світі клієнт-серверна архітектура широко застосовується в багатьох сферах, таких як веб-розробка, мобільні додатки, хмарні сервіси та багато інших. Вона дозволяє створювати складні та масштабовані системи, що задовольняють потреби користувачів і забезпечують ефективну роботу програмного забезпечення. Зрозуміння принципів та переваг клієнт-серверної архітектури дозволяє розробникам створювати високоякісні та надійні додатки.

Одним з найпоширеніших прикладів клієнт-серверної архітектури є веб-розробка. Веб-додатки використовуються мільйонами користувачів щодня і зазвичай складаються з клієнтської частини, яка відображається в браузері користувача, та серверної частини, яка обробляє запити та надсилає відповіді.

У веб-розробці клієнтська частина (Front-end, зовнішня частина), яка використовується в браузері, може бути реалізована за допомогою мов розмітки, таких як HTML, CSS та мов програмування таких як JavaScript, вони використовуються для створення структури, оформлення та логіки веб-сторінок. Ця частина відповідає за відображення інтерфейсу користувача (реалізацію привабливого дизайну для користувача), обробку подій та запитів користувача, та взаємодію з користувачем.

Серверна частина (Back-end, внутрішня частина), зазвичай реалізована за допомогою певного серверного фреймворку(Django, Express.js, Laravel та інші), вона обробляє запити від клієнтів, взаємодіє з базою даних та виконує необхідні обчислення. Серверна частина забезпечує обробку запитів, зберігання даних та здійснення всіх необхідних операцій, які не є видимими для користувача. Вона включає в себе сервери, бази даних, програмне

забезпечення та інші компоненти, необхідні для забезпечення функціональності програмного продукту.

Застосування клієнт-серверної архітектури у веб-розробці дозволяє розподілити завдання між клієнтом і сервером, забезпечуючи більшу швидкість та продуктивність системи. Клієнтська частина може бути розгорнута на різних пристроях та платформах, таких як комп'ютери, мобільні телефони, планшети, тоді як серверна частина забезпечує однаковий функціонал для всіх клієнтів. Тому з цим підходом до веб-додатку, наприклад, можна також створити мобільний додаток, який буде використовувати серверний функціонал поточного додатку.

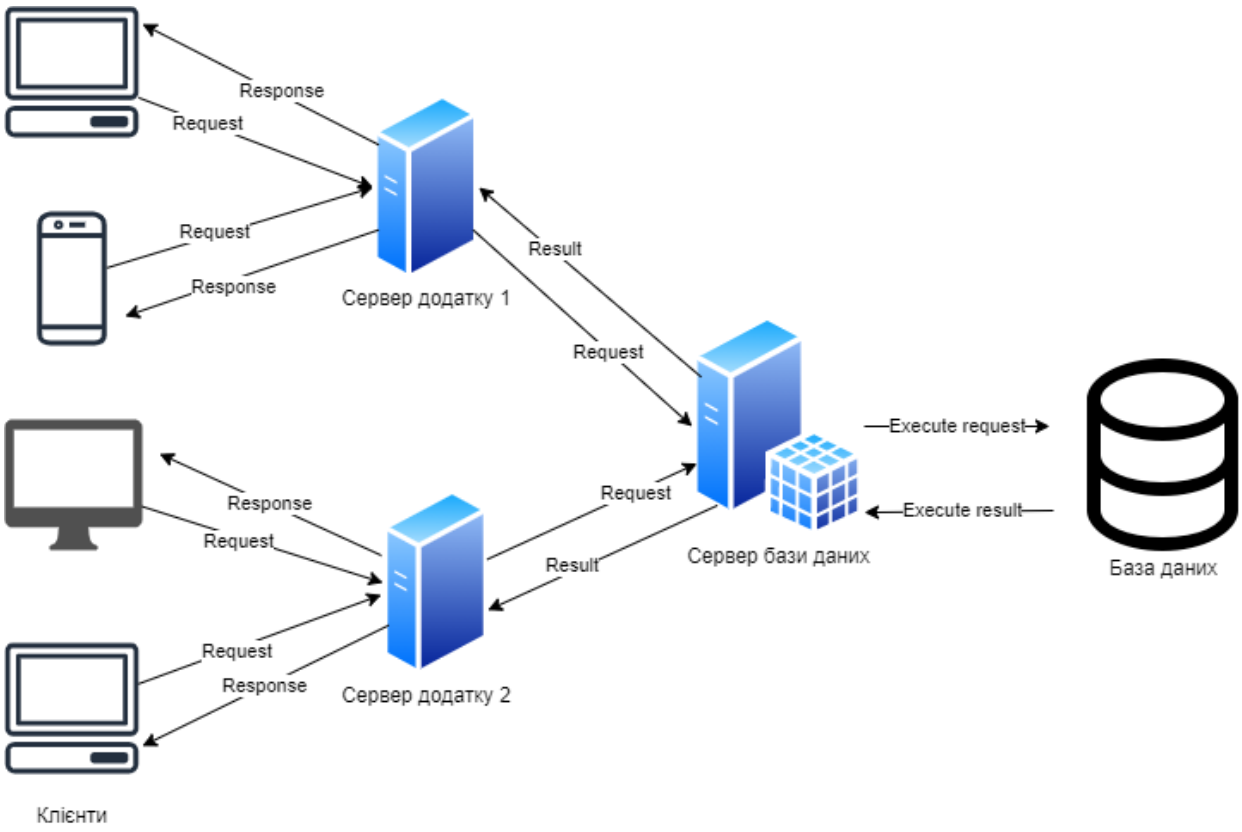

Рисунок 2.1 – Схема-приклад клієнт-серверної архітектури

Одна з переваг клієнт-серверної архітектури полягає в тому, що вона дозволяє досягти більшої масштабованості та ефективності. Завдяки розділенню функціональності між клієнтом і сервером, можна використовувати різні ресурси і клієнту і серверу для оптимального

виконання завдань. Наприклад, сервер може мати потужнішу обчислювальну машину та доступ до бази даних, що дозволяє ефективно обробляти запити та зберігати дані. Клієнтська частина може працювати на менш потужних пристроях, таких як мобільні телефони або планшети, забезпечуючи зручний інтерфейс для користувача.

Тому у клієнт-серверній архітектурі виділяють два типу клієнтів: «товстий клієнт» та «тонкий клієнт».

«Тонкий клієнт» - цей термін визначає клієнта, обчислювальних ресурсів якого достатньо лише запуску необхідного мережного докладання через інтерфейс. Інтерфейс користувача такого додатка формується засобами статичного HTML (виконання JavaScript не передбачається), вся прикладна логіка виконується на сервері. Для роботи тонкого клієнта достатньо забезпечити можливість запуску веб-браузера, у вікні якого і здійснюються всі дії. Тому веб-браузер часто називають "універсальним клієнтом".

«Товстий клієнт» - таким є робоча станція чи персональний комп'ютер, які працюють під управлінням власної операційної системи та мають необхідний набір програмного забезпечення. До мережних серверів «товсті» клієнти звертаються, переважно, за додатковими послугами (наприклад, доступ до сервера або корпоративної бази даних).

Також під «товстим» клієнтом мається на увазі і клієнтський мережевий додаток, запущений під управлінням локальної ОС. Така програма поєднує компонент представлення даних (графічний інтерфейс користувача ОС) і прикладний компонент (обчислювальні потужності клієнтського комп'ютера).

У цілому, з розвитком технологій розробки веб-додатків, виникла необхідність у створенні нового узагальненого або середнього типу клієнтів, які були більш складними та мають багатофункціональні можливості. Цей тип клієнтів називається «rich»-клієнт.

«Rich»-клієнт - це клієнтська програма або додаток, який має розширені можливості та здатність до виконання складних операцій без

необхідності постійного звернення до сервера. Завдяки цьому, «rich»-клієнт може виконувати обробку даних та візуалізацію локально, що покращує швидкодію та взаємодію з користувачем.

Загалом, клієнт-серверна архітектура є потужним та ефективним підходом у розробці програмного забезпечення, зокрема у веб-розробці.

2.2 Вибір середовища розробки

Для дипломного проекту було обрано середовище розробки Visual Studio Code.

Visual Studio Code – потужний редактор коду, призначений для різних мов розробки. Гнучкий і зручний, має велику кількість функцій. Є редактором коду від Microsoft, який виступає «полегшеною» інтерпретацією Visual Studio. За допомогою нього можна не лише займатися написанным додатків. Visual Studio Code підтримує велику кількість плагінів, які дозволять "розігнати" редактор до повноцінного середовища програмування. В основному з його допомогою розробляють сайти та веб-додатки.

Основні переваги цього середовища розробки [12]:

1. Це безкоштовний, з відкритим вихідним кодом і крос-платформний редактор, який працює на Windows, Linux і MacOS, так що ви можете працювати незалежно від платформи, на якій засновано пристрій.

2. Підтримує багато мов програмування

3. Легкий та інтуїтивний для використання та має велику документацію

4. Швидкість. Використовує не так багато ресурсів комп’ютера, та дозволяє швидко працювати навіть на девайсах які не мають потужне апаратне забезпечення.

5. Має внутрішню інтеграцію з системою контролю версій

6. Дозволяє удосконалювати систему за допомогою великої кількості плагінів для облегшення роботи з розробкою та редагуванням коду,

підтримки інших мов програмування, додавання нових модулів у середовище тощо.

2.3 Вибір основної мови програмування

Для дипломного проекту було обрано мову програмування TypeScript.

TypeScript — мова програмування, представлена Microsoft у 2012 році і позиціонована як засіб розробки веб-додатків, що розширює можливості JavaScript. Розробником мови TypeScript є Андерс Хейлсберг, який раніше створив такі мови як Turbo Pascal, Delphi і C#. З моменту свого анонсування компанією Microsoft у 2012 році, TypeScript не перестає розвиватися та схиляє все більше професійних розробників писати свої програми на ньому. Тому на даний момент практично неможливо знайти бібліотеку, яка б не була портована на TypeScript. Чимало проектів, написаних на JavaScript, стали переноситися на TypeScript. Популярність і актуальність ідей нової мови призвела до того, що низка цих ідей у подальшому стали частиною нового стандарту JavaScript. Переваги TypeScript були підхоплені розробниками ряду поширених і широко використовуваних фреймворків.

Основні переваги TypeScript:

1. TypeScript - це строго типізована та компільована мова, чим, можливо, буде ближче до програмістів Java, C# та інших строго типізованих мов. При запуску на виході компілятор генерує той самий JavaScript, який потім виконується браузером. TypeScript є типізованою надмножиною JavaScript, а це означає, що будь-яка програма на JS є програмою на TypeScript. У TS можна використовувати всі ті конструкції, які застосовуються в JS – ті ж оператори, умовні, циклічні конструкції. Також сувора типізація мови зменшує кількість потенційних помилок, які могли б виникнути під час розробки на JavaScript та поліпшує безпеку розробки [8].

2. TypeScript дозволяє писати більш зрозумілий код, що читається, максимально описує предметну область, за рахунок чого архітектура стає

більш вираженою, а розробка неявно збільшує професійний рівень програміста.

3. Потенціал TypeScript дозволяє швидше та простіше писати великі складні комплексні програми, відповідно їх легше підтримувати, розвивати, масштабувати та тестувати, ніж на стандартному JavaScript.

4. Підтримка багатьма популярними середовищами розробки.

5. TypeScript реалізує повноцінне об'єктно-орієнтоване програмування (ООП), підтримуючи різні концепції: класи, інтерфейси, успадкування та багато іншого. Що дозволяє використовувати різні підходи до створення архітектури та різні паттерни проектування. Це полегшує створення добре організованого масштабованого коду, роблячи TypeScript відмінним вибором для проектів, що розвиваються [8].

6. На TypeScript можна реалізувати як клієнтську (Front-end) так і серверну (Back-end) частину додатку.

В цілому ця мова програмування є зручним та популярним у сучасній розробці інструментом, що дає переваги у розвитку як розробляємого додатку так і професійного розвитку самого програміста.

2.4 Основні технології для розробки клієнтської частини

2.4.1 React

Для розробки дипломного проекту була обрана популярна бібліотека для створення інтерфейсу користувача.

React - бібліотека для веб-інтерфейсів та нативних інтерфейсів користувача. ReactJS - одна з найпопулярніших бібліотек JavaScript для розробки мобільних та веб-додатків. Створений Facebook, React містить набір багаторазово використовуваних фрагментів коду JavaScript, званих компонентами, які використовуються для створення інтерфейсу користувача (UI). React використовується для створення модульних інтерфейсів і сприяє

розробці багаторазово використовуваних компонентів інтерфейсу, які відображають динамічні дані [2].

Важливо, що ReactJS не є фреймворком JavaScript. Це тому, що він відповідає тільки за рендеринг компонентів шару подання програми. React – це альтернатива таким фреймворкам, як Angular та Vue, які дозволяють створювати складні функції.

React використовує декларативну парадигму, яка дозволяє програмам бути ефективними та гнучкими. Він створює прості уявлення для кожного стану у вашому додатку та ефективно оновлює та відображає лише потрібний компонент при зміні даних. Декларативне подання робить код більш передбачуваним і полегшує налагодження. Кожен компонент у додатку React відповідає за рендеринг окремого фрагменту HTML-коду, що багаторазово використовується. Можливість вкладення компонентів в інші компоненти дозволяє створювати складні програми із простих будівельних блоків. Компонент також може відстежувати свій внутрішній стан, наприклад компонент TabList може зберігати в пам'яті змінну для відкритої вкладки.

При створенні клієнтських додатків розробники усвідомили, що DOM працює повільно (об'єктна модель документа (DOM) - це інтерфейс прикладного програмування (API) для документів HTML і XML. Він визначає логічну структуру). документів та спосіб доступу до документа та маніпулювання їм.). Отже, щоб прискорити процес, React реалізує віртуальний DOM, який по суті є уявленням дерева DOM JavaScript. Тому, коли йому потрібно прочитати або записати в DOM, він використовуватиме його віртуальне уявлення. Потім віртуальний DOM спробує знайти найефективніший спосіб оновлення DOM браузера. На відміну від елементів DOM браузера, елементи React є простими об'єктами. React DOM дбає про оновлення DOM відповідно до елементів React.

На Рис. 2.2 можемо побачити процес порівняння віртуального та браузерного DOM.

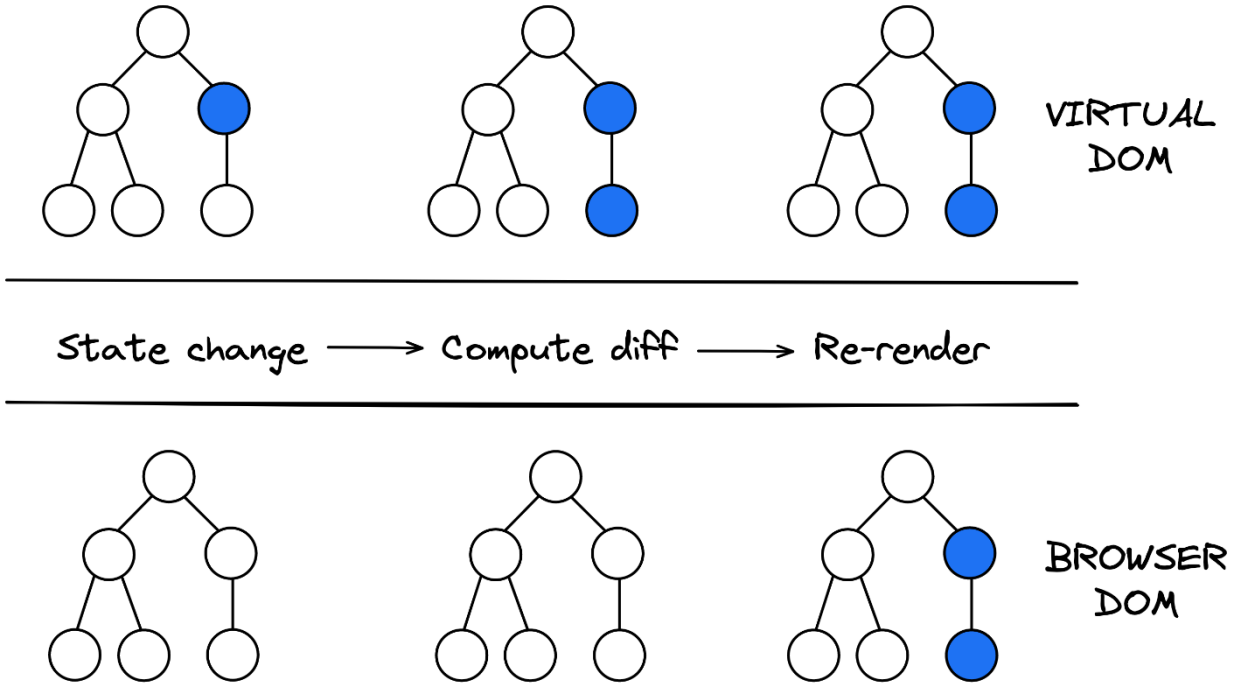

Рисунок 2.2 – Віртуальне дерево DOM та процес порівняння

Блакитні кружки представляють вузли, що змінилися. Ці вузли представляють елементи інтерфейсу користувача, стан яких змінилося. Потім розраховується різниця між попередньою версією віртуального дерева DOM і поточним деревом віртуального DOM. Потім все батьківське під-дерево повторно відображається для надання оновленого інтерфейсу користувача. Це оновлене дерево потім оновлюється до реального DOM.

React також ввів таке поняття як JSX. JSX - розширення мови JavaScript. Його використовують, коли потрібно пояснити React, як має виглядати UI. JSX нагадує мову шаблонів, наділену логікою JavaScript. А саме це просто синтаксичний прийом для створення дуже специфічних об'єктів JavaScript. Замість того, щоб штучно розділити технології, поміщаючи розмітку та логіку в різні файли, React поділяє відповідальність за допомогою слабко пов'язаних одиниць, які називаються «компонентами», які містять і розмітку, і логіку. Після компіляції кожен JSX-вираз стає звичайним викликом JavaScript-функції, результат якого об'єкт JavaScript [6].

Наступні частини коду на Рис 2.3 еквівалентні між собою.

```jsx
const element = (
  <h1 className="greeting">
    Привет, мир!
  </h1>
);
```

```jsx
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Привет, мир!'
);
```

Рисунок 2.3 – Приклад представлення JSX у вигляді об’єкту JS

Узагальнюючи, React має таки функціональні переваги:

1. Простота використання. Розробники зі знанням JS або TS можуть швидко опанувати бібліотеку та почати розробляти веб-додатки та інше.

2. Підтримує багаторазове використання різних компонентів у різних проектах. Оскільки ReactJS має відкритий вихідний код, можна створювати компоненти, скорочуючи час розробки складних веб-додатків. Більше того, React дозволяє вкладати компоненти між іншими для створення складних функцій без роздування коду. Оскільки кожен компонент має власні елементи управління, їх легко підтримувати.

3. Інтеграція JSX полегшує написання React-компонентів. Користувачі можуть створювати JavaScript-об'єкти у поєднанні з типографікою та HTML-тегами. JSX також спрощує багатофункціональний рендеринг, зменшуючи кількість коду без шкоди для функціональності програми. Хоча JSX не є найпопулярнішим синтаксичним розширенням, воно довело свою ефективність у розробці спеціальних компонентів та динамічних додатків.

4. Висока продуктивність. Віртуальний DOM дозволяє ReactJS оновлювати дерево DOM найефективнішим способом. Зберігаючи Virtual DOM у пам'яті, React усуває надмірний повторний рендеринг, який може зменшити продуктивність. Крім того, одностороння прив'язка даних React між елементами полегшує процес налагодження. Будь-які зміни, внесені до

дочірніх компонентів, не вплинуть на батьківську структуру, що знижує ризик помилок.

В цілому, ReactJS — це надійна бібліотека JavaScript, яка використовується в динамічній розробці веб-додатків, вона покращує продуктивність веб-додатка та дозволяє розробляти зручні компоненти для повторного використання у розробці.

2.4.2 MobX

Одним із ключових компонентів програм є управління станом. Управління станом відноситься до управління елементами інтерфейсу користувача (UI), такими як текстові поля і кнопки, - воно включає в себе вилучення, збереження і візуалізацію даних з елементів управління інтерфейсу користувача.

Для керування станом у JavaScript є бібліотека під назвою MobX. MobX - це проста масштабована бібліотека, яка добре працює разом з React. MobX надає механізм, необхідний для зберігання та оновлення стану програми, яку React використовує для рендерингу компонентів.

MobX є автономним і не залежить від будь-якої зовнішньої бібліотеки або фреймворку для роботи. Існують реалізації MobX у популярних інтерфейсних фреймворках або бібліотеках, таких як React, Vue та Angular.

На Рис. 2.4 ми можемо детально розглянути процесс роботи MobX.

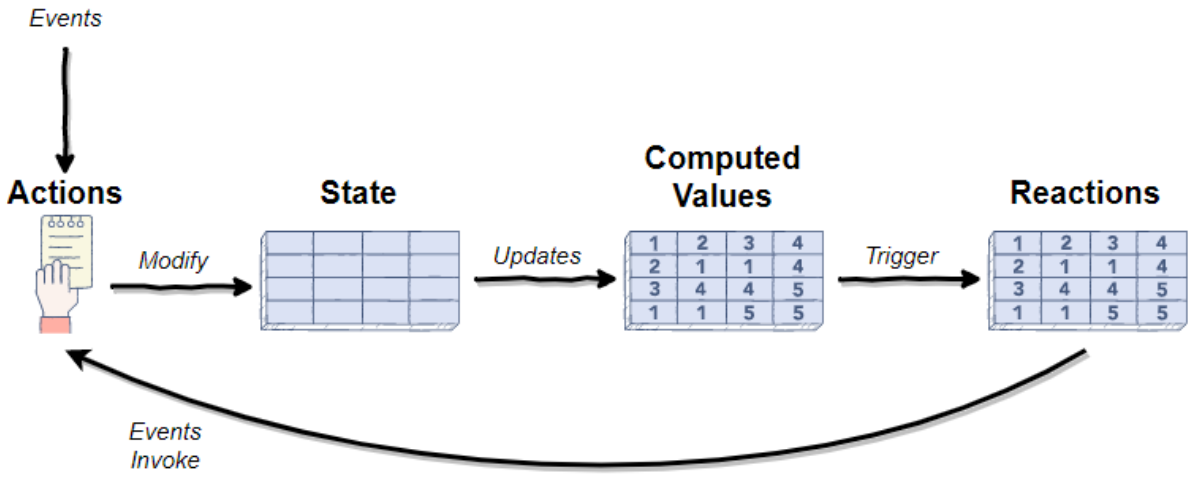

Рисунок 2.4 – Схема роботи MobX

MobX вводить кілька понять: стан, дії та похідні (включаючи реакції та обчислювані значення) [3]. Розглянемо детально поняття які представлені на цій схемі (Рис. 2.4).

Подія викликає дію , а дія — це поштовх , необхідний для зміни спостережуваного стану. Дії - це в основному функції, які змінюють стан програми. MobX підтримує односпрямований потік даних і забезпечує автоматичне оновлення всіх змін, на які впливає дія . Ці автоматичні оновлення є побічними ефектами дії. MobX дії – це методи, які маніпулюють та оновлюють стан. Ці методи можна пов'язати з обробником подій JavaScript, щоб забезпечити їх запуск подією інтерфейсу користувача.

Стан програми відноситься до всієї моделі програми та може містити різні типи даних, включаючи масив, числа та об'єкти. Кожна програма має такі структури даних. Стан цих структур даних підтримується шляхом додавання до них можливостей спостереження. Коли об’єкти доступні для спостереження, вони можуть транслювати нові зміни, на які спостерігач може реагувати.

Все (не лише значення), отримане зі стану програми без подальшої взаємодії, називається похідними. Похідні будуть прослуховувати будь-який

конкретний стан, а потім виконувати деякі обчислення для отримання окремого значення цього стану. Спадкування може повертати дані будь-якого типу, включаючи об'єкти. У MobX два типи похідних - це реакції та обчислювані значення.

Значення, що обчислюються - це значення, отримані зі стану програми . Ці значення обчислюються з уже певних спостережуваних. Ці обчислювані значення можуть змінюватись від простих значень, таких як кількість незавершених завдань, до складних речей, таких як візуальне HTML-подання завдань.

Реакції чимось схожі на значення, що обчислюються , за винятком того, що реакції є побічним ефектом зміни стану програми , і нове значення не створюється. Реакції можуть бути різних типів і оброблятися по різному. Іноді при зміні стану можуть виникати автоматичні побічні ефекти, необхідні для оновлення стану. MobX називає це реакцією та відрізняє реакції від обробників подій у DOM. Реакції можуть робити віддалений мережний запит, викликати локальне сховище або навіть додавати новий елемент DOM на льоту.

В цілому, MobX - це бібліотека, яка спрощує роботу з управлінням станом програми. Вона добре поєднується з React та іншими фреймворками і надає зручні механізми для збереження та оновлення стану, який використовується для рендерингу компонентів. MobX використовує поняття стану, дій та похідних, що дозволяє просто взаємодіяти зі станом програми та автоматично оновлювати залежні від нього елементи інтерфейсу користувача. Завдяки простоті використання та гнучкості MobX став популярним інструментом для управління станом в JavaScript додатках.

2.4.3 Material-UI

Зазвичай для реалізації UI ми використовуємо HTML, котрий потім стилізуємо за допомогою CSS, але для цього давно створені технології для

полегшення життя розробника - не один раз, не два, а кілька разів за допомогою таких інструментів, як bootstrap, Foundation, Tailwind, Bulma та Semantic UI. Але є одна структура інтерфейсу користувача, яка виділяється серед інших: Material-UI.

Material-UI призначений для швидкого прототипування, підвищення загальної швидкості розробки програмного забезпечення та швидкого створення інтерфейсів користувача. Це допомагає розробникам витрачати більше часу на логічні шари своїх програм, а не довго розробляти елементи інтерфейсу, пов'язаними з CSS. Перевага Material-UI у тому, що він створений для React.

Material-UI - це інфраструктура інтерфейсу користувача для React, яка реалізує концепцію Material Design від Google [13]. Одна з чудових особливостей Material-UI полягає в тому, що він пропонує широкий спектр компонентів, які можуть створювати практично будь-який мислимий тип інтерфейсу користувача. Наприклад, ми можемо знайти важливі компоненти, такі як кнопки, картки, повзунки, і більш складні компоненти, такі як діалоги, сітки та підказки. Крім того, Material-UI поставляється з вбудованою системою тем, яка спрощує створення тем користувача для програми.

Material-UI має безліч цікавих функцій та переваг, які можуть допомогти при розробці веб-сайтів та програм:

1. Сумісність. Однією з основних переваг Material-UI є його сумісність із різними системами стилів. Ця сумісність означає, що розробники можуть використовувати бібліотеку з різними структурами CSS або навіть із звичайним CSS.

2. Готові компоненти. Існує безліч доступних компонентів Material-UI, тому ви обов'язково знайдете той, який відповідає вашим потребам. Ось кілька прикладів:

— Діалоги використовуються для відображення важливої інформації користувача. Їх можна використовувати для підтвердження дії або надання інформації про процес.

— Значки. Іконки — чудовий спосіб додати візуальний інтерес до вашого проекту. Ви можете використовувати їх для представлення різних дій або просто для того, щоб додати яскравості дизайну.

— Сітки. Сітки – це потужний інструмент компонування, який можна використовувати для створення складних адаптивних макетів.

— Типографіка. Типографіка — важлива частина будь-якого проекту. Material-UI надає безліч опцій, які допоможуть вам створити красивий текст, що читається.

3. Теми з можливістю налаштування. Можливість легко налаштувати зовнішній вигляд Material-UI є однією з його ключових переваг. Широкий вибір готових тем та стилів компонентів означає, що зазвичай є щось, що задовольнить потреби кожного, але іноді потрібно щось своє. За допомогою кількох рядків коду можна створити власну тему або навіть налаштувати окремі компоненти, щоб усе виглядало так, як хоче розробник.

4. Регулярні поновлення. З популярністю Material-UI, що постійно зростає, його розробники регулярно випускають нові оновлення. Нові функції та компоненти доступні розробникам для використання у кожному випуску. Бібліотека також регулярно оновлюється для запобігання та усунення будь-яких виявлених помилок.

У результаті, Material-UI ідеально підходить для підприємств, стартапів та розробників додатків, які хочуть розпочати роботу якнайшвидше, з найвищим рівнем гнучкості та надійності, укладеним в одну просту у використанні структуру.

2.4.4 i18next

Бібліотека i18next є потужним інструментом для локалізації (перекладу) веб-додатків. Вона надає зручні та ефективні засоби для управління міжнародними версіями додатків, дозволяючи легко перекладати текстові ресурси на різні мови і забезпечувати мультиязичну підтримку.

Основні переваги бібліотеки i18next включають:

1. Простота використання: i18next пропонує простий та зрозумілий API, який дозволяє легко впровадити локалізацію у вашому додатку.

2. Підтримка багатомовності: i18next дозволяє створювати і керувати перекладами для різних мов. Ви можете додавати нові мови, оновлювати існуючі переклади та динамічно змінювати мову відображення в додатку.

3. Гнучкість: бібліотека надає різні можливості для роботи з перекладами, такі як підтримка змінних, форматування дати та часу, множини і числових форматів. Ви можете використовувати ці можливості для зручного та точного відображення текстових ресурсів у різних мовах.

4. Кешування перекладів: i18next використовує кешування для покращення продуктивності та ефективного використання перекладів. Це дозволяє знизити затрати на мережеві запити та прискорити завантаження додатку.

5. Розширюваність: бібліотека надає можливості для розширення та налаштування. Ви можете використовувати різні плагіни та інструменти, щоб додовати функціональність i18next. Наприклад, ви можете використовувати плагіни для інтеграції зі зовнішніми перекладачами або для автоматичного виявлення мови користувача.

6. Підтримка різних платформ: i18next може використовуватися не лише для веб-додатків, але й для мобільних додатків та інших платформ. Це дозволяє забезпечити єдиний механізм локалізації для всіх ваших проектів.

7. Активна спільнота та документація: i18next має широку та активну спільноту користувачів, яка надає підтримку, допомогу та розв'язання проблем. Крім того, бібліотека має детальну документацію з прикладами використання, що допомагає вам швидко освоїти її функціонал.

Загалом, бібліотека i18next є потужним інструментом для локалізації веб-додатків, що пропонує простоту використання, гнучкість, підтримку багатомовності та розширення. Вона допомагає зробити ваш додаток

доступним для користувачів з різних країн та культур, сприяючи його успіху на міжнародному ринку.

2.4.5 Google Maps API

React-Google-Maps API є бібліотекою, яка надає можливості інтеграції Google Maps у веб-додатки, розроблені з використанням фреймворку React. Ця бібліотека дозволяє створювати і взаємодіяти з картами Google у зручний і простий спосіб, надаючи розширені можливості для відображення мап, маркерів, ліній, полігонів, інфо-вікон та іншого вмісту на карті.[5]

Основні переваги використання React-Google-Maps API включають:

1. Простота інтеграції: Бібліотека надає простий та зрозумілий API, який легко інтегрується з React-додатками. Вона пропонує компоненти React, які дозволяють зручно керувати відображенням та взаємодією з картами Google.

2. Розширені можливості візуалізації: React-Google-Maps API надає багатий набір функціональності для відображення різних елементів на карті, таких як маркери, лінії, полігони, кірки, інфо-вікна тощо. Це дозволяє створювати різноманітні та виразні візуальні представлення на карті. Які можна використовувати для реалізації велодоріг.

3. Інтерактивність і взаємодія: Завдяки React-Google-Maps API ви можете легко додавати взаємодію з картою, таку як обробка кліків, перетягування маркерів, зміна масштабування тощо. Ви можете реагувати на події на карті та виконувати відповідні дії з урахуванням дій користувача.

4. Гнучкість та налаштування: React-Google-Maps API дозволяє налаштовувати вигляд і поведінку карт, використовуючи різні параметри та опції. Ви можете контролювати розміщення, стиль карт, масштабування, типи карти, стилізацію та інші аспекти відображення. Це дозволяє вам адаптувати карту до ваших потреб та стилістики додатку.

5. Інтеграція з іншими React-компонентами: React-Google-Maps API добре поєднується з іншими компонентами та функціональністю, що пропонує React-екосистема. Ви можете використовувати його разом з роутерами, формами та іншими розширеннями React, що спрощує розробку веб-додатків.

6. Підтримка мобільних пристроїв: React-Google-Maps API добре працює на мобільних пристроях, що дозволяє створювати мобільні додатки з використанням карт Google. Це відкриває широкі можливості для розробки локаційних сервісів, навігації та інших мобільних додатків, або для росширення веб-додатку у майбутньому.

7. Підтримка кешування та оптимізація продуктивності: React-Google-Maps API надає механізми кешування, які забезпечують ефективну роботу з картами та оптимізацію продуктивності. Це важливо, особливо при використанні багатоелементних карт або при обробці великої кількості даних на карті.

Загалом, React-Google-Maps API є потужним інструментом для інтеграції карт Google у веб-додатки, розроблені з використанням React. Вона надає зручність, гнучкість та багатий набір функціональності для відображення та взаємодії з картами Google, що допомагає створювати привабливі та функціональні локаційні додатки.

2.5 Основні технології для розробки серверної частини

2.5.1 Nest.js

Nest.js — одна з найшвидших платформ Node.js для створення ефективних, масштабованих серверних програм корпоративного рівня з використанням Node.js (Node. js — це платформа з відкритим вихідним кодом для роботи з мовою JavaScript, побудована на движку Chrome, вона дозволяє писати серверний код для веб-застосунків і динамічних веб-

сторінок, а також програм командного рядка). Він відомий створенням програм, що легко тестуються, підтримуються і масштабуються, з використанням сучасного JavaScript і TypeScript. [4]

Основною метою Nest.js є полегшення створення складних серверних додатків із застосуванням модульної та масштабованої архітектури. Він надає розробникам зручні інструменти для побудови додатків, що використовують різні шари і розділення відповідальностей.

Основні концепції, на яких ґрунтується Nest.js, включають модулі, провайдери, контролери та сервіси.

Nest.js додатки будуються з різних модулів, які організовують код в логічні групи. Кожен модуль може мати свої провайдери, контролери та інші компоненти.

Провайдери використовуються для створення та керування залежностями всередині додатку. Вони можуть бути сервісами, репозиторіями або будь-якими іншими компонентами, які надаються всередині модуля.

Контролери відповідають за обробку HTTP-запитів (протокол передачі інформації в інтернеті) та визначення маршрутів. Вони приймають запити, обробляють їх і повертають відповіді.

Сервіси містять бізнес-логіку додатку та виконують операції над даними. Вони можуть взаємодіяти з базою даних, іншими сервісами або зовнішніми API.

Декілька ключових рис та переваг Nest.js:

1. Модульність: Nest.js пропонує модульну архітектуру, яка дозволяє розділити функціональність додатку на невеликі, самодостатні модулі. Це сприяє збереженню чистоти коду, полегшує тестування та підтримку додатку.

2. Підтримка TypeScript: Nest.js може використовувати TypeScript, що дозволяє використовувати сильну типізацію та покращує розробку завдяки підказкам, перевірці типів та автодоповненню коду.

3. Обробка маршрутів та HTTP-запитів: Nest.js надає легкий та потужний механізм для обробки маршрутів та HTTP-запитів. За допомогою декораторів, ви можете визначати маршрути, контролери та обробники запитів, що спрощує роботу з маршрутизацією.

4. Масштабованість: Nest.js підтримує розширюваність та масштабованість додатків. За допомогою модульної архітектури ви можете легко додавати нові функціональність до свого додатку, розширюючи його з інших модулів. Це полегшує розробку та підтримку великих проектів.

5.Широкий екосистема та активна спільнота: Nest.js має активну спільноту розробників, яка надає підтримку, допомогу та створює розширення та плагіни для фреймворку. Крім того, ви можете використовувати пакети з екосистеми Node.js та TypeScript, що розширюють можливості вашого додатку.

6. Nest.js не залежить від бази даних, що дозволяє легко інтегруватися з будь-якою базою даних SQL або NoSQL. Можна напряму використовувати будь-яку бібліотеку або ORM для інтеграції даних загального призначення Node.js, наприклад, MikroORM, Sequelize, Knex. js, TypeORM і Prisma, щоб працювати на більш високому рівні абстракції. Nest.js забезпечує інтеграцію з TypeORM шляхом надання власного модуля TypeORM. Це дозволяє зручно використовувати TypeORM у Nest.js додатках і створювати розширені серверні додатки з базою даних.

В цілому, Nest.js є потужним фреймворком для розробки серверних додатків, який надає зручні інструменти для побудови масштабованих та модульних додатків. Nest.js допомагає зробити архітектуру проекту більш організованою та забезпечує його розширюваність та підтримку.

2.5.2 TypeORM

TypeORM — це бібліотека, що працює на node.js і написана на TypeScript.

Платформа TypeORM є структурою об'єктно-реляційної проекції (ORM). Загалом об'єктна частина відноситься до моделі у вашому додатку, реляційна частина відноситься до взаємозв'язку між таблицями в системі управління реляційними базами даних (наприклад, Oracle, MySQL, MS-SQL, PostgreSQL і т. д.) і, нарешті, частина проекції відноситься до акту з’єднання моделі та наших таблиць.

ORM – це тип інструмента, який зіставляє сутності з таблицями бази даних. ORM забезпечує спрощений процес розробки за рахунок автоматизації перетворення об'єкта на таблицю та таблиці на об'єкт. Як тільки буде написана своя модель даних в одному місці, стане простіше оновлювати, підтримувати та повторно використовувати код.

Оскільки модель слабко пов'язана з іншою частиною програми, то можна змінити її без будь-якої жорсткої залежності з іншою частиною програми і можена легко використовувати її будь-де всередині програми. TypeORM дуже гнучкий, абстрагує систему БД від програми.

Основні концепції та функції TypeORM включають:

1. Сутності (Entities): Ви визначаєте сутності як класи, які відображають таблиці у базі даних. Кожен екземпляр класу представляє рядок у таблиці. Ви можете використовувати анотації або декоратори, щоб визначити взаємозв'язки між сутностями, типи даних тощо.

2. Міграції (Migrations): TypeORM дозволяє вам створювати міграції, які представляють зміни у структурі бази даних, такі як створення таблиць, додавання або вилучення стовпців тощо. Міграції дозволяють зберігати версіоновану історію змін бази даних та легко виконувати їх.

3. Запити (Queries): TypeORM надає різноманітні методи та мову запитів для виконання операцій з базою даних, таких як вибірка даних, додавання, оновлення, вилучення тощо. Ви можете використовувати ланцюжки методів для визначення умов, сортування, обмежень та інших параметрів запиту. Приклади методів включають find, insert, update, delete та багато інших.

4. Відносини (Relations): TypeORM дозволяє встановлювати зв'язки між сутностями, такі як один до одного, один до багатьох або багато до багатьох. Ви можете використовувати анотації або декоратори для визначення відносин між сутностями. TypeORM автоматично створює SQL-запити(мова структурованих запитів) для зв'язків та пов'язує пов'язані об'єкти.

5. Транзакції (Transactions): TypeORM підтримує транзакції для забезпечення консистентності та цілісності даних. Ви можете використовувати транзакції для групування кількох запитів у єдину операцію, яка виконується атомарно.

TypeORM має такі переваги:

1. Масштабованість. Функції TypeORM надає механізм міграцій, що дозволяє зберігати історію змін схеми бази даних та легко розгортати ці зміни на різних середовищах.

2. Легко інтегрується з іншими модулями.

3. Ідеально підходить для будь-якої архітектури від малих до корпоративних програм.

4. Підтримує багато баз даних, що дає вам вибір у використанні підходящої бази для будь-якого проекту.

5. Оскільки TypeORM побудований на основі TypeScript, ми отримуємо переваги строгої типізації, підказок типів та перевірки на етапі компіляції, що полегшує виявлення помилок та покращує розробку.

6. Автоматична генерація SQL-запитів. TypeORM автоматично генерує SQL-запити на основі моделей та взаємозв'язків. Не потрібно писати складні SQL-запити вручну; TypeORM робить це за нас, що зменшує кількість витраченого часу та зусиль.

7. Інтеграція з Nest.js. TypeORM є одним з основних модулів, які підтримуються та рекомендуються використовувати в архітектурі проектів на основі Nest.js. Це забезпечує гладку і безпроблемну інтеграцію між Nest.js та базою даних за допомогою TypeORM. Можна легко налаштувати та використовувати TypeORM в рамках Nest.js модулів та сервісів.

8. Активна спільнота. TypeORM має велику та активну спільноту розробників, яка постійно вносить внески до проекту. Це означає, що ви можете швидко знайти підтримку, документацію та приклади використання.

Загалом, TypeORM є потужним та зручним інструментом для роботи з реляційними базами даних у JavaScript та TypeScript. Він надає ORM-підхід, що спрощує взаємодію з базою даних шляхом використання об'єктно-орієнтованої моделі програмування. Використання TypeORM дозволяє розробникам швидко і ефективно працювати з базою даних, зменшуючи необхідність у написанні складних SQL-запитів вручну та полегшуючи управління структурою бази даних.

2.5.3 MySQL

Для ефективної організації та збереження великого обсягу даних у багатьох проектах необхідне використання баз даних. База даних - це організована колекція даних, яка зберігається та управляється за допомогою системи управління базами даних (СУБД). СУБД надає інтерфейс для створення, збереження, оновлення та видалення даних з бази даних.

MySQL — це система управління реляційними базами даних. Термін «реляційна» (від латинського relatio — відношення) вказує, перш за все, на те, що така модель зберігання даних побудована на взаємовідношенні складових її частин. MySQL є однією з найпопулярніших систем керування базами даних. Вона була створена у 1995 році шведською компанією MySQL AB. Згодом компанія була придбана компанією Sun Microsystems, а потім Oracle Corporation стала власником MySQL. MySQL написана на мові програмування C та C++ і доступна для різних операційних систем, включаючи Windows, Linux та macOS. [14]

MySQL працює на основі клієнт-серверної архітектури, де СУБД виконується на сервері, а клієнти взаємодіють з базою даних за допомогою

мови запитів SQL (Structured Query Language). Клієнти можуть бути різних типів, таких як веб-сайти, додатки або інші сервери.

MySQL зберігає дані в таблицях, які складаються з рядків та стовпців. Вона підтримує широкий спектр типів даних, включаючи числа, рядки, дати, час, булеві значення та багато інших. Запити SQL використовуються для створення, зміни, видалення та запитування даних з бази даних. [14]

MySQL має декілька переваг, які роблять її популярним вибором для багатьох проектів. Ось детальніше про переваги MySQL:

1. Надійність та стабільність. MySQL відома своєю високою надійністю та стабільністю. Вона вже довгий час на ринку та має велику спільноту користувачів, яка активно сприяє у виявленні та виправленні помилок. Багато великих компаній та організацій використовують MySQL для критичних застосунків, що свідчить про її надійність.

2. Швидкодія. MySQL є відомим своєю високою продуктивністю та швидкодією. Вона ефективно оптимізує запити та має оптимізований двигун бази даних, що дозволяє швидко виконувати операції над даними. Багатопоточна архітектура MySQL дозволяє обробляти багато запитів одночасно, що покращує її продуктивність.

3. Масштабованість: MySQL може легко масштабуватись як вертикально (шляхом додавання більш потужного обладнання) так і горизонтально (шляхом розподілу даних між кількома серверами). Це дозволяє розширювати систему зростанням обсягу даних та навантаження. MySQL також підтримує реплікацію, що дозволяє створювати резервні копії та розподіляти навантаження між кількома серверами.

4. Гнучкість: MySQL підтримує багато функцій та розширень, що дозволяє розробникам створювати потужні та складні бази даних залежно від вимог проекту. Вона підтримує транзакції, індексування, зовнішні ключі, генерацію унікальних ідентифікаторів, процедури та багато іншого. Це дозволяє розробникам гнучко налаштовувати та оптимізувати базу даних під конкретні потреби проекту.

5. Безпека: MySQL надає різні механізми безпеки, що дозволяють забезпечити конфіденційність, цілісність та доступ до даних. Вона підтримує автентифікацію користувачів, рівні доступу, шифрування даних та інші заходи безпеки для захисту від несанкціонованого доступу та зловживань.

6. Простота використання: MySQL має зрозумілий та простий у використанні синтаксис SQL, що робить його доступним для початківців та досвідчених розробників. Вона надає зручний інтерфейс командного рядка та графічний інтерфейс для адміністрування та управління базами даних.

7. Підтримка: MySQL має велику та активну спільноту користувачів, яка надає підтримку, документацію, плагіни та різноманітні ресурси. Це робить розвиток, пошук відповідей на запитання та вирішення проблем швидкими та ефективними завдяки наявності багатьох ресурсів та експертної допомоги.

8. Відкритість та безкоштовність: MySQL базується на відкритій ліцензії, що дозволяє використовувати, модифікувати та розповсюджувати її безкоштовно. Це робить MySQL доступним для широкого кола користувачів та дозволяє економити витрати на ліцензування дорогих СУБД.

MySQL є потужною та надійною системою управління базами даних, яка пропонує широкі можливості для зберігання, керування та оптимізації даних. Вона володіє швидкодією, масштабованістю, безпекою та простотою використання, що робить її популярним вибором для розробників усього світу.

2.5.4 Passport-google-oauth20

Passport-google-oauth20 - це пакет аутентифікації стороннього розробника для Node.js, який надає можливість аутентифікувати користувачів за допомогою сервісу Google OAuth 2.0.

Google OAuth 2.0 є протоколом авторизації, який дозволяє користувачам надавати дозволи стороннім додаткам для доступу до їхнього

облікового запису Google. Passport.js - це фреймворк аутентифікації для Node.js, який спрощує процес автентифікації в веб-додатках.

passport-google-oauth20 реалізує автентифікацію з використанням Google OAuth 2.0. Він надає набір функцій і стратегій, що дозволяють легко інтегрувати аутентифікацію Google з вашим Node.js додатком.

Переваги використання passport-google-oauth20 включають:

1. Простота використання. Пакет passport-google-oauth20 дозволяє швидко налаштувати аутентифікацію Google OAuth 2.0 з декількома рядками коду. Він надає зручний інтерфейс для реалізації автентифікації.

2. Підтримка OAuth 2.0. Google OAuth 2.0 є відкритим та безпечним протоколом авторизації, який забезпечує безпеку та захист даних користувачів. Використовуючи passport-google-oauth20, ви отримуєте можливість використовувати цей протокол для аутентифікації користувачів.

3. Можливість налаштування. Passport-google-oauth20 надає гнучкі налаштування, що дозволяє вам контролювати процес автентифікації, включаючи доступ до різних полів інформації про користувача.

4. Інтеграція з Passport.js: passport-google-oauth20 є однією з багатьох стратегій аутентифікації, які працюють з Passport.js. Це дозволяє вам поєднувати аутентифікацію з Google з іншими стратегіями, такими як аутентифікація з використанням соціальних мереж або локальна аутентифікація (за поштою та паролем на сайті).

5. Безпека: passport-google-oauth20 забезпечує безпеку під час процесу автентифікації. Він використовує токени доступу і оновлювання, щоб забезпечити безпеку передачі даних між вашим додатком і сервісом Google.

6. Розширені можливості: passport-google-oauth20 дозволяє отримувати доступ до різних ресурсів інформації про користувача, таких як ім'я, електронна пошта, фотографія профілю та інші. Це дозволяє вам використовувати ці дані для персоналізації додатку або взаємодії з іншими сервісами.

Загалом, passport-google-oauth20 є потужним і надійним інструментом, який допоможе вам легко і безпечно реалізувати аутентифікацію користувачів з використанням Google OAuth 2.0 в вашому Node.js додатку. Він дозволяє вам сконцентруватися на розробці вашого додатку, забезпечуючи високий рівень безпеки та зручності для ваших користувачів.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОЕКТУ

3.1 План реалізації

Для успішного виконання будь-якого проекту треба спланувати етапи його розробки, використаємо для цього естімацію.

Естімація проекту - це процес визначення приблизного обсягу робіт, часу та ресурсів, необхідних для успішного виконання проекту. Вона дозволяє оцінити трудомісткість проекту та його вартість. Естімація є важливим етапом в плануванні та управлінні проектами, оскільки допомагає розробнику (або команді розробників) проекту визначити реалістичні цілі, зрозуміти, які ресурси та зусилля будуть необхідними, і підготувати план роботи.

Можемо побачити естімацію для нашого проекту на Таблиці 3.1. В таблиці ми можемо бачити стовпець «Частина» в якому є три види позначок: FE, BE, Сервіс. FE (Front-end) позначає що завдання треба виконати у клієнтській частині додатку. BE (Back-end) позначає що завдання треба виконати у серверній частині додатку. Сервіс відповідає за будь які налаштування.

Таблиця 3.1 – Естимація практичної частини дипломного проекту

Частина	**Завдання**	**Час виконання**	**Складність виконання**
	Частина 1: Налаштування проекту [FE]		
FE	Налаштування нового проекту React	6 ч	Середня
FE	Налаштування менеджеру стану (MobX)	4 ч	Легко
FE	Налаштування локалізації (en, uk)	4 ч	Легко
FE	Налаштування MaterialUI	4 ч	Легко
FE	Налаштування маршрутизації (react-router)	5 ч	Легко
	Частина 2: Налаштування проекту		

Частина	Завдання	Час виконання	Складність виконання
	[BE]		
BE	Налаштування нового проекту Nest.js	5 ч	Середня
BE	Налаштування сервера бази даних (MySQL)	8 ч	Складно
BE	Налаштування ORM	5 ч	Середня
Частина	**Завдання**	**Час виконання**	**Складність виконання**
	Частина 3: Авторизація		
BE	Створення стратегії авторизації	5 ч	Середня
BE	Увійти/Зареєструватися ендпоінт	3 ч	Легко
FE	Створити захищені маршрути	8 ч	Середня
FE	Створити форми для вхіду та реєстрації	5 ч	Середня
Сервіс	Створити google додаток для OAuth	5 ч	Середня
BE	Створити Google стратегію	5 ч	Середня
BE	Ендпоінти для вхіду та реєстрації через Google	3 ч	Середня
FE	Кнопка входу через гугл	3 ч	Легко
	Частина 4: Головна сторінка та навігація		
FE	Створити базовий макет інтерфейсу (Меню навігації, домашня сторінка)	8 ч	Середня
	Частина 5: Сторінка налаштування користувача		
FE	Створення сторінки налаштувань	8 ч	Середня
FE	Створення форми для зміни пароля	6 ч	Середня
FE	Створення форми для зміни особистих даних	6 ч	Середня
BE	Ендпоінт для зміни пароля	3 ч	Середня
BE	Ендпоінт для зміни особистих даних	3 ч	Середня
	Частина 6: Сторінка новин		
FE	Створення сторінки з новинами	8 ч	Середня
FE	Створення зберігання новин	7 ч	Середня
FE	Відображення новин	6 ч	Середня
BE	Створення моделі новин	2 ч	Середня
BE	Створення module, controller, service для новин	5 ч	Середня
BE	Створити ендпоінт для новин	3 ч	Легко

Частина	Завдання	Час виконання	Складність виконання
BE	Створення ендпоінту для створення нових новин	3 ч	Легко
BE	Створення ендпоінту для редагування новин	3 ч	Легко
BE	Створення ендпоінту для видалення новин	3 ч	Легко
FE	Створення редактору для створення новини	8 ч	Середня
FE	Створення редактору для редагування новини	5 ч	Середня
FE	Додати кнопки редагування/видалення/створення новини	4 ч	Легко
Частина	**Завдання**	**Час виконання**	**Складність виконання**
	Частина 7: Сторінка карти		
FE	Створення сторінки з картою	8 ч	Середня
Сервіс	Ініціалізвання google maps api	2 ч	Легко
FE	Створити компонент карти	3 ч	Середня
BE	Створити модель маркерів для карти	2 ч	Легко
BE	Створення для маркерів module, controller, service	5 ч	Середня
BE	Створення модель для маршрутів та маркерів	2 ч	Легко
BE	Створення для маршрутів module, controller, service	5 ч	Середня
FE	Додати можливість додавання/редагування/видалення маршрутів	8 ч	Складно
FE	Додати можливість додавання/редагування/видалення точок	8 ч	Складно

3.2 Серверна частина. Робота з базою даних

У серверній частині проекту використовується Nest.js в поєднанні з TypeORM для забезпечення взаємодії з базою даних MySQL. Наступні кроки включають:

– Встановлення залежностей та налаштування серверної частини Nest.js

– Підключення до бази даних MySQL за допомогою TypeORM

– Створення моделей для таблиць бази даних

– Створення модулів, контролерів та сервісів для роботи з базою даних

Для створення моделей бази даних необхідно спланувати які дані необхідні у базі даних, для цього можна використати ER модель.

ER (entity relational) модель – це концептуальна модель даних високого рівня. ER-моделювання допомагає систематично аналізувати вимоги до даних для створення добре спроектованої бази даних. Модель сутності-відносини представляє сутності реального світу та відносини між ними. Найкраще завершити моделювання ER перед впровадженням бази даних.

Можемо бачити модель для наших даних на Рис. 3.1.

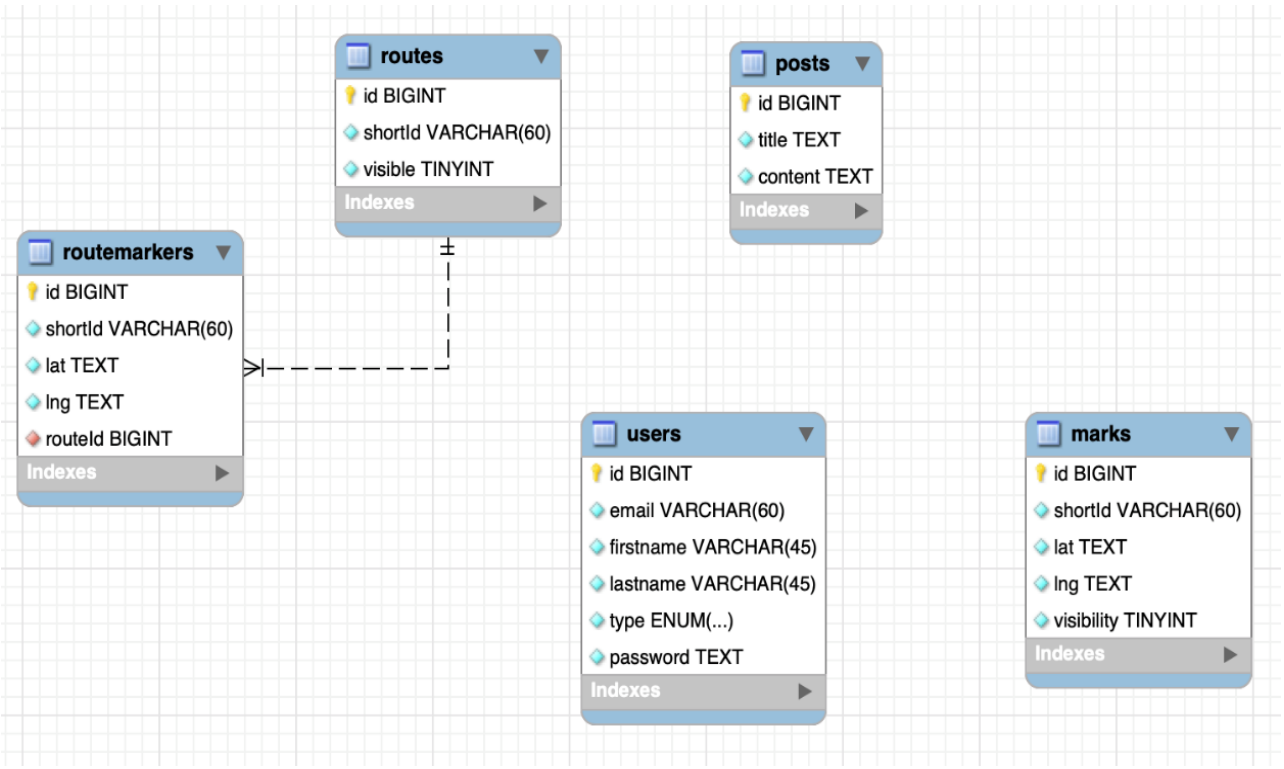

Рисунок 3.1 – Модель бази даних

Розглянемо створені TypeORM моделі. Модель для маркера на карті з широтою та довготою, і позначкою для видимості маркеру:

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm'

@Entity('marks')
export class Mark {
    @PrimaryGeneratedColumn({ type: 'bigint' })
    id: number

    @Column('varchar', {
        length: 60,
    })
    shortId: string

    @Column('text')
    lat: string

    @Column('text')
    lng: string

    @Column('tinyint')
    visibility: 1 | 0
}
```

Модель для новин яка має заголовок та зміст новини:

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm'

@Entity('posts')
export class Post {
    @PrimaryGeneratedColumn({ type: 'bigint' })
    id: number

    @Column('longtext')
    title: string

    @Column('longtext')
    content: string
}
```

Дві моделі для доріг. У MapRoute властивість markers у представляє зв'язок один-до-багатьох з моделлю RouteMarker, оскільки один маршрут може мати багато маркерів.

Перша модель - велодоріжки:

```
import { Entity, Column, PrimaryGeneratedColumn, OneToMany }
from 'typeorm'
import { RouteMarker } from './routemarker.entitie'

@Entity('routes')
export class MapRoute {
    @PrimaryGeneratedColumn({ type: 'bigint' })
    id: number

    @Column('varchar', {
        length: 60,
    })
    shortId: string
```

```
    @Column('tinyint')
    visible: 1 | 0

    @OneToMany(() => RouteMarker, (marker) => marker.route)
    markers: RouteMarker[]
}
```

Друга модель - маркеру велодоріжки:

```
import { Entity, Column, PrimaryGeneratedColumn, ManyToOne,
JoinColumn } from 'typeorm'
import { MapRoute } from './route.entitie'

@Entity('routemarkers')
export class RouteMarker {
    @PrimaryGeneratedColumn({ type: 'bigint' })
    id: number

    @Column('varchar', {
        length: 60,
    })
    shortId: string

    @Column('text')
    lat: string

    @Column('text')
    lng: string

    @ManyToOne(() => MapRoute, (maproute) => maproute.markers, {
        onDelete: 'CASCADE',
        nullable: false,
    })
    @JoinColumn({ name: 'routeId' })
    route: MapRoute
}
```

Також модель для користувача веб-додатку. Вона має електронну пошту, ім’я та фамілію, роль користувача (адміністратор, редактор та т.д) та пароль.

```
import { Roles, roles } from 'src/common/roles'
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm'

@Entity('users')
export class User {
    @PrimaryGeneratedColumn({ type: 'bigint' })
    id: number

    @Column('varchar', {
        length: 60,
    })
    email: string

    @Column('varchar', {
        length: 45,
    })
    firstname: string
```

```
    @Column('varchar', {
        length: 45,
    })
    lastname: string

    @Column({
        type: 'enum',
        enum: roles,
    })
    type: Roles

    @Column('text')
    password: string
}
```

Наступний етап − це контролери та сервіси, що є ключовими компонентами архітектури серверної частини додатка. Вони відіграють різні ролі і мають свої функціональні обов'язки.

Контролери (Controllers):

− Контролери відповідають за обробку вхідних HTTP-запитів та відправку відповідей клієнту.

− Вони приймають дані від клієнта і передають їх відповідним сервісам для обробки.

− Контролери містять методи, які кріпляться на конкретні маршрути (URL-шляхи) і визначають, як обробляти запити для цих маршрутів.

Сервіси (Services):

− Сервіси виконують бізнес-логіку додатка і містять специфічні для них операції.

− Вони забезпечують взаємодію з базою даних, виконують обчислення, роблять запити до зовнішніх сервісів та інші дії, необхідні для виконання конкретних функціональних завдань.

− Сервіси можуть містити більш складну логіку, яка може включати кілька операцій бази даних або інших взаємодій.

Контролери та сервіси працюють разом для обробки запитів та забезпечення необхідного функціоналу серверної частини. Зазвичай, контролер отримує дані з HTTP-запиту, викликає відповідний метод сервісу

для обробки цих даних, отримує результат від сервісу та повертає відповідь клієнту.

Розділення логіки додатка на контролери та сервіси дозволяє досягти розділення відповідальностей, більшої модульності та повторного використання коду. Контролери відповідають за комунікацію з клієнтом, тоді як сервіси відповідають за бізнес-логіку та обробку даних. Це дозволяє забезпечити чітку розділення обов'язків та зберігання логічної структури додатка.

Розглянемо контролер до карти. Даний код представляє контролер MarksController (Додаток А, п. 1), який відповідає за обробку HTTP-запитів, пов'язаних з маркерами та велодоріжками на карті.

У контролері використовується шлях /marks, що вказує на те, що всі ендпоінти цього контролера будуть доступні за URL /marks.

В конструкторі контролера відбувається ініціалізація сервісу MarksService, який відповідає за бізнес-логіку пов'язану з маркерами.

У контролері оголошені різні HTTP-методи (Get, Post, Put, Delete) для обробки різних типів запитів.

Кожен метод має свій URL-шлях (наприклад, routes, routes/:id) та використовує певні декоратори (@Get, @Post, @Put, @Delete), які вказують на тип запиту.

Кожен метод також має використовувати різні UseGuards декоратори для перевірки прав доступу до цих ендпоінтів. Наприклад, @UseGuards(JwtAuthGuard,CheckUser,checkPermissions([Permissions.can_view_map])), вказує, що ендпоінт доступний тільки автентифікованим користувачам з необхідними правами доступу до огляду карти.

Кожен метод викликає відповідний метод сервісу MarksService для виконання потрібної бізнес-логіки. Наприклад, return this.marksService.createMark(body) викликає метод createMark з сервісу MarksService.

Контролер MarksController відповідає за обробку різних запитів, пов'язаних з маркерами, та викликає відповідні методи сервісу MarksService, який виконує бізнес-логіку пов'язану з маркерами та велодоріжками.

Перейдемо для розгляду його сервісу. Цей сервіс, MarksService (Додаток А, п. 2), відповідає за бізнес-логіку пов'язану з маркерами та маршрутами на карті.

Основні функції сервісу:

1. getRoutes(): Повертає список маршрутів разом з їх маркерами. Використовує метод find() репозиторію routeRepository для отримання даних з таблиці MapRoute.

2. createRoute(data: CreateRouteDto): Створює новий маршрут на основі отриманих даних з об'єкта CreateRouteDto. Створює новий об'єкт MapRoute, встановлює значення його властивостей на основі переданих даних та зберігає його в таблиці MapRoute за допомогою методу save() репозиторію routeRepository.

3. editRoute(id: number): Редагує наявний маршрут з вказаним ідентифікатором. Перевіряє наявність маршруту за його ідентифікатором за допомогою методу findOne() репозиторію routeRepository. Якщо маршрут не знайдений, викидається виключення. Встановлює значення властивості visible маршруту на протилежне значення (0 або 1) та оновлює запис маршруту в таблиці MapRoute за допомогою методу update() репозиторію routeRepository.

4. deleteRoute(id: number): Видаляє наявний маршрут з вказаним ідентифікатором. Перевіряє наявність маршруту за його ідентифікатором за допомогою методу findOne() репозиторію routeRepository. Якщо маршрут не знайдений, викидається виключення HttpException з кодом HttpStatus.BAD_REQUEST. Видаляє запис маршруту з таблиці MapRoute за допомогою методу delete() репозиторію routeRepository.

5. createRouteMark(id: number, data: CreateRouteMarkerDto): Створює новий маркер для вказаного маршруту. Перевіряє наявність маршруту за його

ідентифікатором за допомогою методу findOne() репозиторію routeRepository. Якщо маршрут не знайдений, викидається виключення HttpException з кодом HttpStatus.BAD_REQUEST. Створює новий об'єкт RouteMarker, встановлює значення його властивостей на основі переданих даних та пов'язує його зі знайденим маршрутом. Зберігає новий маркер в таблиці RouteMarker за допомогою методу save() репозиторію routeMarkerRepository.

6. deleteRouteMark(id: number): Видаляє наявний маркер з вказаним ідентифікатором. Видаляє запис маркера з таблиці RouteMarker за допомогою методу delete() репозиторію routeMarkerRepository.

7. getMarks(): Повертає список маркерів. Використовує метод find() репозиторію marksRepository для отримання даних з таблиці Mark.

8. createMark(data: CreateMarkDto): Створює новий маркер на основі отриманих даних з об'єкта CreateMarkDto. Створює новий об'єкт Mark, встановлює значення його властивостей на основі переданих даних та зберігає його в таблиці Mark за допомогою методу save() репозиторію marksRepository.

9. setShow(id: number): Змінює статус видимості маркера з вказаним ідентифікатором. Перевіряє наявність маркера за його ідентифікатором за допомогою методу findOne() репозиторію marksRepository. Якщо маркер не знайдений, викидається виключення HttpException з кодом HttpStatus.BAD_REQUEST. Встановлює значення властивості visibility маркера на протилежне значення (0 або 1) та оновлює запис маркера в таблиці Mark за допомогою методу update() репозиторію marksRepository.

10. setDelete(id: number): Видаляє наявний маркер з вказаним ідентифікатором. Видаляє запис маркера з таблиці Mark за допомогою методу delete() репозиторію marksRepository.

Сервіс використовує репозиторії marksRepository, routeRepository та routeMarkerRepository. Кожен репозиторій відповідає за взаємодію з відповідною таблицею бази даних (Mark, MapRoute, RouteMarker).

Цей сервіс відповідає за логіку створення, редагування та видалення маркерів та маршрутів на карті, а також отримання списків маркерів та маршрутів з бази даних. Він використовує репозиторії для доступу до бази даних та виконання операцій з таблицями Mark, MapRoute та RouteMarker.

У коді контролера MarksController методи контролера викликають відповідні методи сервісу MarksService, передаючи їм необхідні дані. Таким чином, контролер взаємодіє з сервісом, який в свою чергу забезпечує обробку бізнес-логіки та виконання операцій з базою даних.

Ця структура дозволяє розділити логіку контролера (взаємодія з клієнтом) та логіку сервісу (бізнес-логіка та робота з базою даних), що сприяє покращенню модульності, читабельності коду.

3.3 Розгляд аутентифікації та прав доступу користувачів

Аутентифікація є невід'ємною частиною більшості програм. Існує безліч різних підходів та стратегій для обробки аутентифікації. Підхід, який використовується для будь-якого проекту, залежить від конкретних вимог програми.

У нашому проекті веб-додатку використовується комбінований підхід щодо аутентифікації, який включає кілька стратегій які ми розглянемо.

Локальна аутентифікація (LocalStrategy). Цей підхід дозволяє користувачам аутентифікуватися за допомогою локальних облікових записів, зокрема, за допомогою електронної пошти та паролю. Користувачі надсилають свої облікові дані (електронна пошта та пароль) на сервер, де вони перевіряються на валідність. Якщо дані є правильними, користувач отримує доступ до ресурсів.

Аутентифікація через JWT (JwtStrategy та JwtAuthGuard): Цей підхід використовує JSON Web Tokens (JWT) для аутентифікації користувачів. Користувач підтверджує свою ідентичність, надаючи правильний JWT-токен, який підписаний сервером. Цей токен перевіряється і розшифровується, щоб

переконатися, що він є дійсним і містить необхідні дані авторизації. Якщо токен дійсний, користувач отримує доступ до захищених ресурсів.

Аутентифікація через Google (GoogleStrategy): Цей підхід дозволяє користувачам авторизуватися за допомогою облікового запису Google. Користувачі переадресовуються на сторінку авторизації Google, де вони вводять свої облікові дані Google. Після успішної авторизації користувача Google повертається назад до додатку з відповідними даними про авторизацію, такими як електронна пошта та інформація про профіль. За допомогою цих даних створюється або оновлюється обліковий запис користувача в системі.

Розглянемо основний код який відповідає за реєстрацію.

Перший код у local.strategy.ts відповідає за стратегію локальної автентифікації. У цьому файлі використовується паспортна стратегія Strategy з модуля passport-local. Ця стратегія дозволяє аутентифікувати користувачів на основі їх локальних облікових даних, таких як електронна пошта та пароль.

Клас LocalStrategy є підкласом PassportStrategy і використовується для реалізації локальної стратегії. У конструкторі цього класу передається екземпляр AuthService, який використовується для перевірки користувача та проведення його аутентифікації.

Метод validate викликається під час спроби аутентифікації користувача. У цьому методі передаються електронна пошта та пароль користувача. За допомогою AuthService перевіряється існування користувача за вказаною електронною поштою. Якщо користувача не знайдено, виникає помилка HttpException з повідомленням "USER_NOT_FOUND" та статусом "BAD_REQUEST".

Після перевірки існування користувача проводиться порівняння хешу пароля, збереженого в базі даних, з хешем введеного пароля. Якщо хеші не збігаються, генерується помилка HttpException з повідомленням "INCORRECT_PASSWORD" та статусом "BAD_REQUEST".

Якщо користувач пройшов всі перевірки, метод validate повертає об'єкт користувача.

Файл local.strategy.ts:

```
import { Strategy } from 'passport-local'
import { PassportStrategy } from '@nestjs/passport'
import { HttpException, HttpStatus, Injectable } from
'@nestjs/common'
import * as sha256 from 'crypto-js/sha256'
import * as Base64 from 'crypto-js/enc-base64'
import { AuthService } from './auth.service'
import * as constants from '../constants'
import { User } from 'src/models/users.entitie'

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
    constructor(private authService: AuthService) {
        super({ usernameField: 'email' })
    }

    async validate(email: string, password: string):
Promise<any> {
        const user: User | null = await
this.authService.validateUser(email)
        if (!user) {
            throw new HttpException(constants.USER_NOT_FOUND,
HttpStatus.BAD_REQUEST)
        }
        if (user.password !==
Base64.stringify(sha256(password))) {
            throw new
HttpException(constants.INCORRECT_PASSWORD,
HttpStatus.BAD_REQUEST)
        }
        return user
    }
}
```

Код у local-auth.guard.ts. Коли класс LocalAuthGuard використовується як гвард для маршруту, він перехоплює запити перед тим, як вони дістаються до контролера. Гвард передає управління до локальної стратегії, яку ми описали у файлі local.strategy.ts. Локальна стратегія виконує аутентифікацію користувача на основі наданих облікових даних.

Якщо аутентифікація успішна, гвард дозволяє запиту продовжити до відповідного маршруту або методу контролера. У противному випадку, якщо аутентифікація не вдалася, гвард відправляє помилку автентифікації.

Завдяки такому підходу ми можемо застосувати гвард до відповідних маршрутів або методів контролера, щоб переконатися, що лише

аутентифіковані користувачі мають доступ до захищених ресурсів або функцій.

Файл local-auth.guard.ts:

```
import { Injectable } from '@nestjs/common'
import { AuthGuard } from '@nestjs/passport'

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

Файл jwt.strategy.ts визначає стратегію аутентифікації на основі JWT. Він розширює клас PassportStrategy з пакету @nestjs/passport і налаштовує стратегію аутентифікації JWT з використанням паспорту (Passport).

У конструкторі JwtStrategy визначені параметри стратегії, такі як витягування JWT з запиту, врахування терміну дії токену та секретний ключ, яким підписуються та перевіряються токени.

Стратегія визначає метод validate, який приймає розкодований JWT і повертає об'єкт, який представляє аутентифікованого користувача.

Файл: jwt.strategy.ts

```
import { ExtractJwt, Strategy } from 'passport-jwt'
import { PassportStrategy } from '@nestjs/passport'
import { Injectable } from '@nestjs/common'
import { jwtConstants } from './constants'

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
    constructor() {
        super({
            jwtFromRequest:
ExtractJwt.fromAuthHeaderAsBearerToken(),
            ignoreExpiration: false,
            secretOrKey: jwtConstants.secret,
        })
    }

    async validate(payload: any) {
        return { id: payload.id, email: payload.email }
    }
}
```

Файл jwt-auth.guard.ts представляє гвард для захисту маршрутів за допомогою JWT-аутентифікації. Він реалізує клас AuthGuard з пакету @nestjs/passport і використовує стратегію JWT для перевірки валідності токену та аутентифікації користувача.

Коли JwtAuthGuard використовується як гвард для маршруту, він перевіряє наявність та валідність JWT-токену у запиті. Якщо токен є валідним, гвард дозволяє запиту продовжити до відповідного маршруту або методу контролера, інакше він відправляє помилку автентифікації.

Файл jwt-auth.guard.ts

```
import { Injectable } from '@nestjs/common'
import { AuthGuard } from '@nestjs/passport'

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

Файл google.strategy.ts використовується для інтеграції аутентифікації через Google OAuth 2.0 у застосунку. У файлі google.strategy.ts визначена стратегія аутентифікації з використанням паспорту (Passport) і пакету passport-google-oauth20. Цей пакет надає стратегію для аутентифікації користувачів за допомогою сервісу Google.

Клас GoogleStrategy розширює PassportStrategy з пакету @nestjs/passport і Strategy з пакету passport-google-oauth20. У конструкторі виконується конфігурація стратегії, така як clientID (ідентифікатор клієнта), clientSecret (секретний ключ клієнта), callbackURL (URL зворотнього виклику) та scope (обсяг запиту).

Стратегія визначає метод validate, який викликається після успішної аутентифікації через Google. Цей метод отримує доступ до інформації про користувача, яку надає сервіс Google після аутентифікації. Метод validate отримує дані про користувача, такі як ім'я, електронна адреса, фото тощо. Він створює об'єкт користувача на основі цих даних та викликає функцію done для завершення процесу аутентифікації.

Файл google.strategy.ts:

```
import { Injectable } from '@nestjs/common'
import { PassportStrategy } from '@nestjs/passport'
import { Strategy, VerifyCallback } from 'passport-google-
oauth20'

@Injectable()
export class GoogleStrategy extends PassportStrategy(Strategy,
'google') {
    constructor() {
```

```
        super({
            clientID: process.env.GOOGLE_OAUTH_ID,
            clientSecret: process.env.GOOGLE_OAUTH_SECRET,
            callbackURL:
`${process.env.GOOGLE_REDIRECT_URL}/auth/google/callback`,
            scope: ['email', 'profile'],
        })
    }

    async validate(accessToken: string, refreshToken: string,
profile: any, done: VerifyCallback): Promise<any> {
        const { name, emails, photos } = profile
        const user = {
            email: emails[0].value,
            firstname: name.givenName,
            lastname: name.familyName,
            picture: photos[0].value,
            accessToken,
        }

        done(null, user)
    }
}
```

Файл auth.service.ts (Додаток А, п. 3) містить реалізацію сервісу автентифікації (AuthService) у застосунку. Цей сервіс відповідає за роботу з користувачами, валідацію даних, створення і перевірку токенів, а також виконання інших операцій, пов'язаних з автентифікацією. В сервісі AuthService визначено декілька методів, розглянемо їх.

Метод validateUser(username: string): Promise<User | null> - цей метод отримує електронну адресу користувача (ім'я користувача) і перевіряє, чи існує користувач з такою адресою в базі даних. Якщо користувач знайдений, повертається його об'єкт, в іншому випадку повертається значення null.

Метод login(user: any) - цей метод приймає об'єкт user, що містить дані про користувача після автентифікації, і генерує JWT-токен на основі цих даних за допомогою JwtService. Повертається об'єкт з access_token, який може бути використаний для подальшої автентифікації.

Метод signUp(user: CreateUserDto): Цей метод приймає об'єкт user з даними нового користувача, який хоче зареєструватися. Метод перевіряє, чи користувач з такою електронною адресою вже існує, перевіряє правильність електронної адреси, створює новий об'єкт користувача, хешує його пароль та

зберігає його в базі даних. Після цього генерується JWT-токен для нового користувача.

Метод googleLogin(req, res) - цей метод використовується для обробки процесу автентифікації через Google. Він отримує дані, отримані під час автентифікації через Google, створює об'єкт користувача на основі цих даних та зберігає його в базі даних. Якщо користувач вже існує, генерується JWT-токен для цього користувача і перенаправляється на вказану URL. Якщо користувач ще не існує, генерується випадковий пароль, створюється новий користувач з даними, отриманими від Google, і генерується JWT-токен для нового користувача. Після цього також перенаправляється на вказану URL.

Метод restorePassword(email: string): Promise<{ result: boolean }>: Цей метод використовується для відновлення пароля користувача. Він отримує електронну адресу користувача, перевіряє, чи існує користувач з такою адресою в базі даних. Якщо користувач знайдений, генерується новий випадковий пароль, оновлюється пароль користувача в базі даних та надсилається електронний лист з новим паролем. Повертається об'єкт { result: true } для підтвердження успішного відновлення пароля.

Файл app.controller.ts є одним з контролерів в вашому застосунку. Контролери виконують обробку запитів, визначають маршрутизацію та обробляють логіку, пов'язану з конкретними ендпоінтами. У контролері AppController визначено функціонал, який обробляє різні запити.

@Get('auth/google/login') та @Get('auth/google/callback'): ці декоратори вказують, що контролер обробляє GET-запити на шляхах /auth/google/login та /auth/google/callback. Ці маршрути використовуються для аутентифікації через Google. Обидва методи використовують декоратор @UseGuards(AuthGuard('google')), який вказує, що для цих маршрутів використовується стратегія аутентифікації Google.

@UseGuards(LocalAuthGuard): Цей декоратор вказує, що метод login обробляє POST-запити на шляху /auth і вимагає аутентифікації з використанням локальної стратегії.

@Post('signup'): Цей декоратор вказує, що метод signUp обробляє POST-запити на шляху /signup. Цей метод викликає функцію signUp з сервісу authService для реєстрації нового користувача. Об'єкт CreateUserDto, який передається в якості параметру методу, містить дані нового користувача, такі як електронна адреса, ім'я та пароль.

@Put('restore'): Цей декоратор вказує, що метод restorePassword обробляє PUT-запити на шляху /restore. Цей метод викликає функцію restorePassword з сервісу authService для відновлення пароля користувача. Об'єкт body, який передається в якості параметру методу, містить електронну адресу користувача, для якого потрібно відновити пароль.

Файл app.controller.ts:

```
import { Controller, Post, UseGuards, Req, Body, Put, Response,
Get } from '@nestjs/common'
import { AppService } from './app.service'
import { LocalAuthGuard } from './auth/local-auth.guard'
import { AuthService } from './auth/auth.service'
import { CreateUserDto } from './dto/createUser.dto'
import { AuthGuard } from '@nestjs/passport'

@Controller()
export class AppController {
    constructor(private readonly appService: AppService, private
authService: AuthService) {}

    @Get('auth/google/login')
    @UseGuards(AuthGuard('google'))
    googleAuth() {
        return
    }

    @Get('auth/google/callback')
    @UseGuards(AuthGuard('google'))
    googleAuthRedurect(@Req() req, @Response() res) {
        return this.authService.googleLogin(req, res)
    }

    @UseGuards(LocalAuthGuard)
    @Post('auth')
    async login(@Req() req) {
        return this.authService.login(req.user)
    }
```

```
    @Post('signup')
    async signUp(@Body() body: CreateUserDto) {
        return this.authService.signUp(body)
    }

    @Put('restore')
    restorePassword(@Body() body: { email: string }) {
        return this.authService.restorePassword(body.email)
    }
}
```

Для визначення доступу до будь-якої частини додатку треба розписати ролі(Roles) та їх дозволи(Permissions) які є у додатку. Потім у будь якому контролері можна буде перевіряти дозволи, як це було у прикладі контролеру для маркерів на карті у пункті 3.2 Серверна частина. Робота з базою даних.

Розглянемо файл roles.ts (Додаток А, п. 4).

Масив roles - це масив, в якому перераховані всі доступні ролі в додатку: 'owner', 'admin', 'mapeditor' та 'viewer'. Цей масив використовується для визначення можливих значень ролі при роботі з користувачами.

Roles - це перерахування (enum), яке визначає ті ж самі ролі, але в якості значень використовуються рядки ('owner', 'admin', 'mapeditor', 'viewer'). Використання enum дозволяє зручно використовувати ці значення в коді, замість безпосереднього використання рядків.

Permissions - це перерахування (enum), яке визначає різні дозволи, пов'язані з функціональністю додатку. Кожен дозвіл має унікальний рядковий ідентифікатор, наприклад, 'can_view_posts', 'can_create_posts' тощо. Ці дозволи використовуються для контролю доступу до певних операцій або ресурсів.

PermissionsByRole - це тип, який визначає об'єкт, де ключами є ролі, а значеннями - масиви дозволів, що відповідають цій ролі. Наприклад, роль 'owner' має дозволи can_view_posts, can_create_posts тощо. Цей тип використовується для збереження відповідних дозволів для кожної ролі.

permissionsByRole - це об'єкт, де ключами є ролі, а значеннями - масиви дозволів, які дозволені для цієї ролі. У цьому об'єкті визначені дозволи для кожної ролі, відповідно до вимог додатку. Наприклад, роль 'owner' має

доступ до всіх можливих дозволів, тоді як роль 'admin' має обмежений набір дозволів.

Загалом аутентифікація та права доступу користувачів є невід’ємною частиною безпеки будь якого додатку. В цілому, розуміння аутентифікації та прав доступу користувачів допомагає створити безпечне середовище для додатку, обмежуючи доступ до функціональності та ресурсів лише авторизованим користувачам.

3.4 Клієнтська частина. Опис функцій та інтерфейсу

Для роботи клієнтської частини треба створити сторінки для нашого додатку та store для роботи зі станом додатку.

Ми використовуємо підхід на основі бібліотеки MobX для управління станом додатку. Нам необхідні файли posts.store.ts, map.store.ts і profile.store.ts які представляють окремі "стори" (stores), які зберігають та керують даними, пов'язаними з відповідними функціональними областями додатку (робота с постами, картою, та даними профіля).

Store - це об'єкт, що містить стан додатку, а також функції, які змінюють цей стан. Наші Store (PostsStore, MapStore, ProfileStore) використовують декоратори @observable для позначення властивостей, які повинні бути спостережуваними та реактивними. Це означає, що будь-які зміни в цих властивостях будуть автоматично відслідковуватися та спричинять перерендеринг компонентів на сторінках, які залежать від цих даних.

Декоратор @action використовується для позначення функцій, які змінюють стан. Це дозволяє MobX відслідковувати зміни та автоматично оновлювати компоненти, що залежать від цих функцій.

Також треба створити index.tsx, де створити екземпляри кожного стору (ProfileStore, PostsStore, MapStore), після чого їх можна буде експортувати для використання в інших частинах додатку. Це дозволяє компонентам

отримувати доступ до стану та функцій, що його змінюють, через імпорт цих змінних.

Файл index.tsx:

```
import ProfileStore from './profile/profile.store'
import PostsStore from './posts/posts.store'
import MapStore from './map/maps.store'

export const profileStore = new ProfileStore()
export const postsStore = new PostsStore()
export const mapStore = new MapStore()
```

Розглянемо докладніше один з них. Наприклад profile.store.ts (Додаток А, п.5). Файл profile.store.ts дозволяє зберігати та керувати даними профілю користувача. Можна використовувати його властивості та дії для відображення та зміни профільних даних, а також управління правами доступу до функціональності додатку, що пов'язана з профілем користувача.

@observable є декоратором, який застосовується до властивостей класу ProfileStore, які потрібно зробити спостережуваними. У нашому випадку, ми використовуємо @observable для наступних властивостей:

− loadingAuth: вказує, чи відбувається завантаження авторизації.

− token: зберігає токен користувача.

− firstname: зберігає ім'я користувача.

− lastname: зберігає прізвище користувача.

− type: зберігає тип користувача.

− permissions: масив дозволів користувача.

Завдяки декоратору @observable, MobX відстежує зміни цих властивостей. Коли будь-яка з них змінюється, MobX автоматично оновлює компоненти, які залежать від цих даних.

@action є декоратором, який застосовується до методів класу ProfileStore, які змінюють стан. Ми використовуємо @action для наступних методів:

− setPermissions(data: Permissions[]): встановлює дозволи користувача.

− setLoadingUpdateProfile(value: boolean): встановлює значення, що вказує, чи відбувається оновлення профілю.

− setLoadingUpdatePassword(value: boolean): встановлює значення, що вказує, чи відбувається оновлення пароля.

− setLoadingAuth(value: boolean): встановлює значення, що вказує, чи відбувається завантаження авторизації.

− setToken(value: string | null): встановлює токен користувача.

− setFirstname(value: string): встановлює ім'я користувача.

− setLastname(value: string): встановлює прізвище користувача.

− setType(value: string): встановлює тип користувача.

− resetStore(): скидає значення всіх властивостей стору ProfileStore на їхні початкові значення.

Декоратор @action вказує MobX, що ці методи є дійсними діями, які змінюють стан. Коли будь-який з цих методів викликається та змінює стан, MobX автоматично оновлює всі компоненти, які залежать від цих даних.

Наприклад, якщо викликати метод setPermissions, який встановлює дозволи користувача, потім MobX спостережує за змінами цієї властивості та автоматично оновлює компоненти, які використовують ці дозволи. Таким чином, зміни відображаються безпосередньо в інтерфейсі користувача.

Загалом, використання @observable та @action у сторах дозволяє зручно вести управління станом вашого додатку. Можна спостерігати за змінами в спостережуваних властивостях та автоматично оновлювати компоненти, що залежать від цих даних, без необхідності вручну керувати оновленнями.

Далі розглянемо одну з реалізацій сторінок, а саме головну сторінку.

Файл Home.tsx (Додаток А, п. 6) містить компонент React, який представляє домашню сторінку додатку. Основний функціонал компонента полягає в рендерингу навігаційної панелі, бічного меню, та відображенні вмісту сторінок в залежності від маршруту.

Коли компонент рендериться, він спочатку ініціалізує стан за допомогою useState. Змінна open відповідає за стан бічного меню (відкрите або закрите).

Далі, з контейнера profileStore, компонент отримує ім'я, прізвище та тип користувача (firstname, lastname, type).

У компоненті використовуються різні компоненти та елементи з бібліотеки Material-UI, такі як AppBar, Drawer, Toolbar, Typography, IconButton, Badge тощо. Вони відповідають за розміщення елементів на сторінці, стилізацію і реакцію на події користувача.

Для роботи з маршрутизацією використовується компонент Routes та Route з пакету react-router. Залежно від поточного маршруту, відображається відповідний компонент сторінки. Наприклад, ViewPost, CreatePost, Posts, GoogleMapWrapper, Settings.

Також у компоненті використовуються компоненти ProtectPermission та Permissions з папки components та store/profile/profile.store відповідно, які виконують функцію контролю доступу користувача до певних сторінок за допомогою перевірки дозволів.

На останок, компонент Home експортується з додатковою обгорткою observer з пакету mobx-react, що дозволяє компоненту реагувати на зміни стану profileStore та автоматично оновлювати відображення при їх зміні.

На наступних рисунках можна розглянути реалізований інтерфейс користувача від профиля власника. На рисунку 3.2 можемо бачити сторінку для входу у додаток з логотипом та формою для футентифікації користувача, а також можливістю зайти до додатку через гугл та одразу виставити потрібну мову.

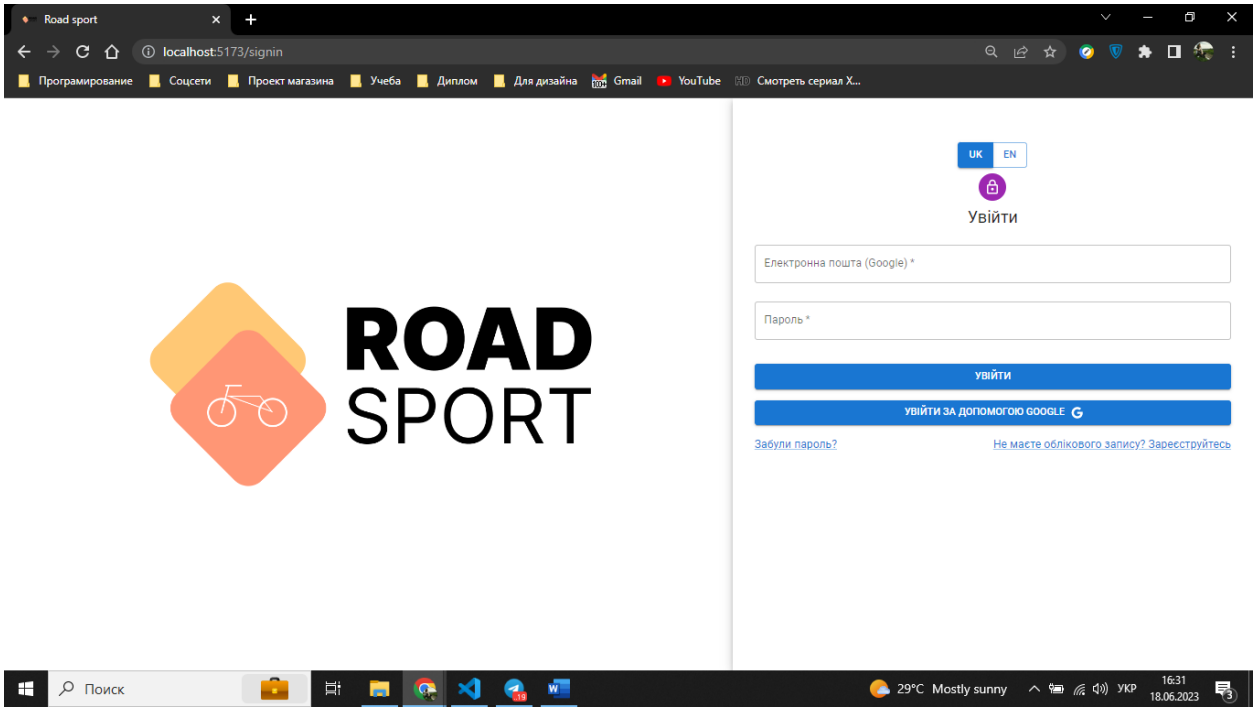

Рисунок 3.2 – Сторінка для входу у додаток

На рисунку 3.3 можемо бачити вкладку новин, де є список новин, навігація по сторінкам та можливість створити новину від профіля власника.

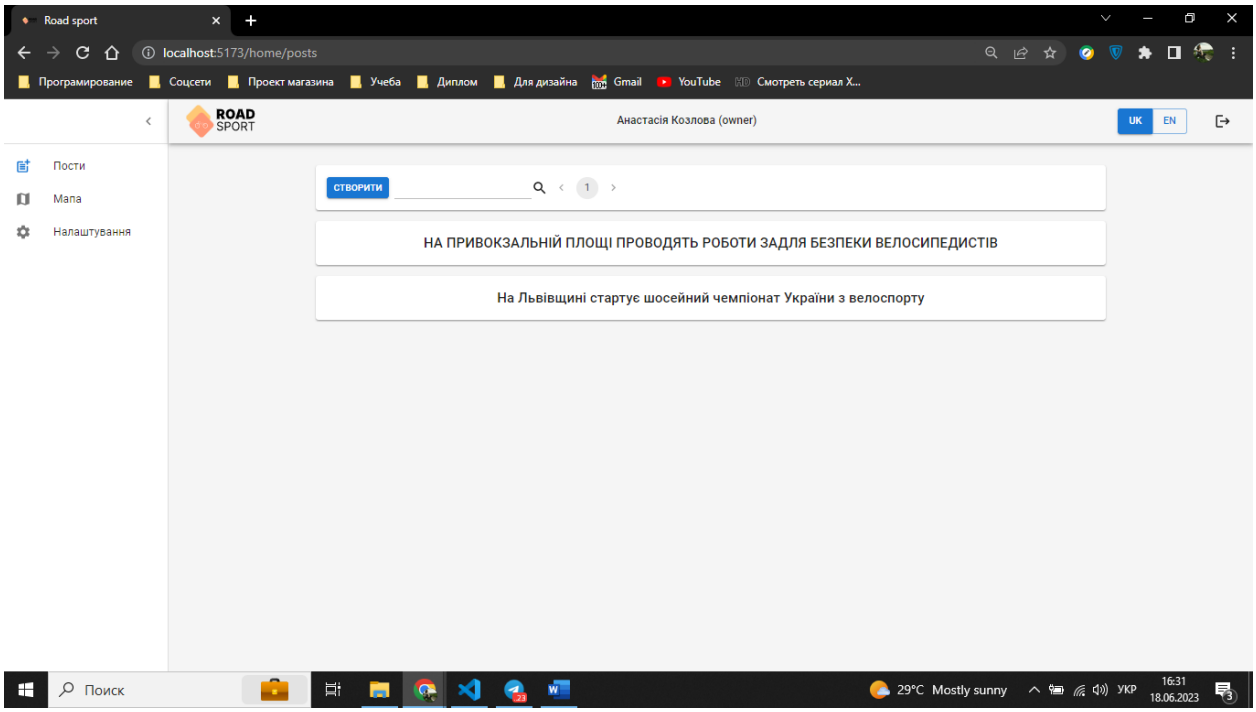

Рисунок 3.3 – Сторінка з новинами

На наступному рисунку 3.4 можемо бачити готову новину, яка показує можливості редагування для створення новин.

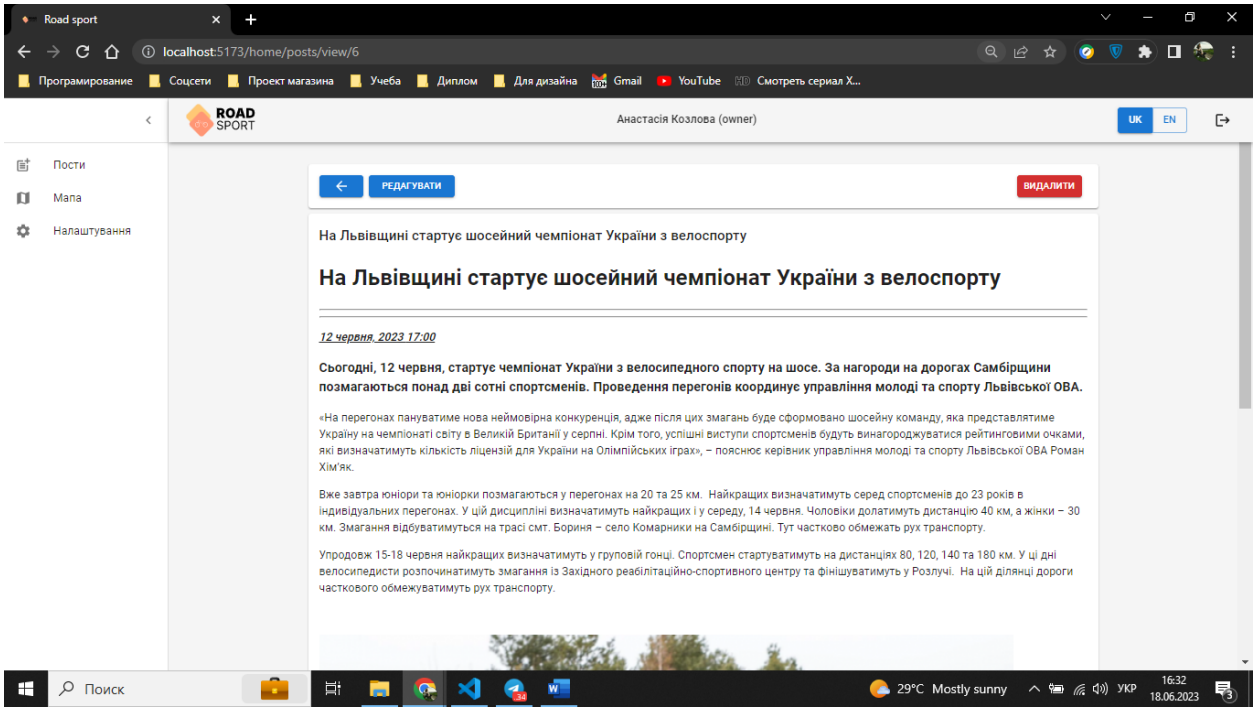

Рисунок 3.4 – Відкрита новина

На наступному рисунку 3.5 є можливість оглянути зібрані дані з велодоріжок на вкладці мапа, де в нас є можливість добавляти, редагувати, видаляти та скривати маркери та велодоріжки.

На рисунку 3.6 можемо бачити налаштування щодо аккаунту користувача де він може поновити свій пароль або змінити ім’я та прізвище.

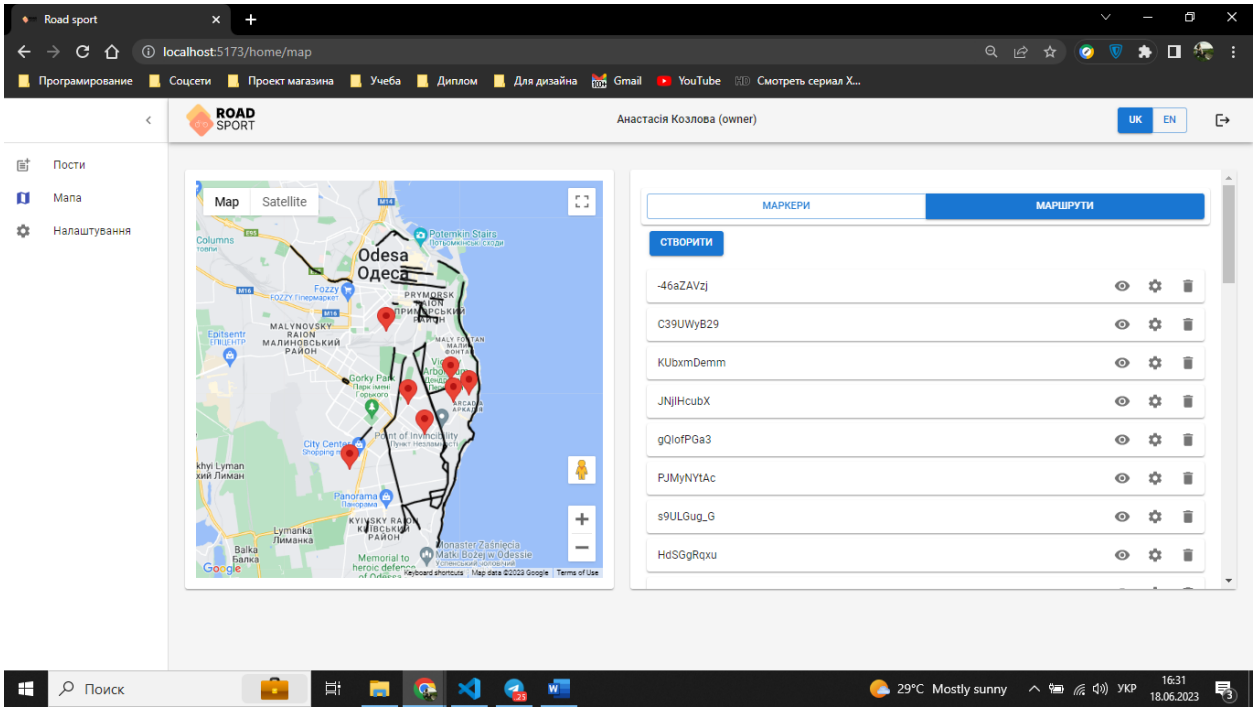

Рисунок 3.5 – Сторінка з мапою

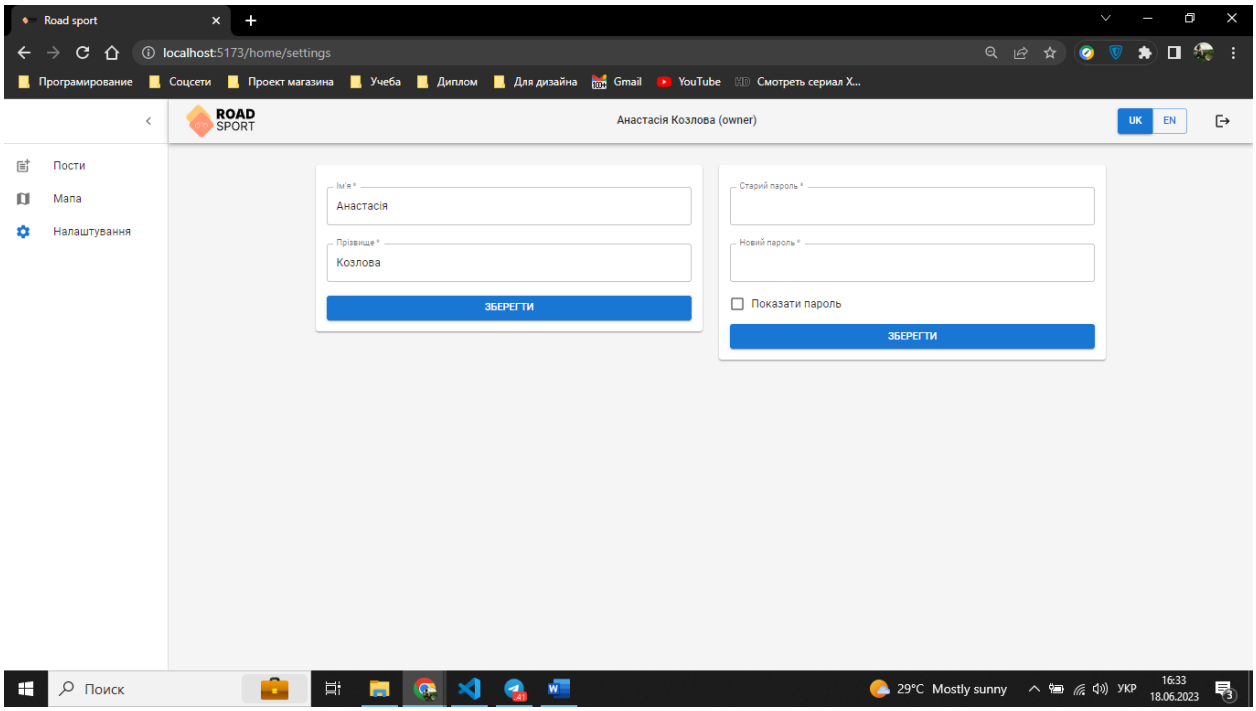

Рисунок 3.6 – Сторінка з налаштуваннями профіля

3.5 Тестування розробленого додатку

Тестування - це процес перевірки програмного забезпечення з метою виявлення помилок, дефектів або неправильного функціонування. Його основна мета полягає в перевірці, чи працює програма вірно, відповідає вимогам і очікуванням, і має задану якість.

Тестування допомагає виявити помилки, дефекти або недоліки в програмному забезпеченні, які можуть призвести до неправильної роботи або збоїв. Це дозволяє виправити проблеми ще до випуску продукту та забезпечити високу якість і надійність програми. Також тестування допомагає перевірити, чи виконує програма всі вимоги, встановлені для неї. Це важливо для забезпечення відповідності програмного забезпечення очікуванням замовника та користувачів.

Тестування розробленого веб-додатку можна розглянути у Таблиці 3.2.

Таблиця 3.2 – Тестування функцій додатку

Відмітка на виконання	Види тестів	Примітка
✓	Аутентифікація користувача	Тест пройдено успішно
✓	Перевірка прав доступу кожного типу користувача	Тест пройдено успішно
✓	Можливість переглядати новини	Тест пройдено успішно
✓	Можливість додавати новини	Тест пройдено успішно
✓	Можливість редагувати новини	Тест пройдено успішно
Відмітка на виконання	Види тестів	Примітка
✓	Можливість видаляти новини	Тест пройдено успішно

✓	Можливість переглядати велодоріжки та маркери на карті	Тест пройдено успішно
✓	Можливість додавати велодоріжки та маркери на карті	Тест пройдено успішно
✓	Можливість редагувати велодоріжки та маркери на карті	Тест пройдено успішно
✓	Можливість видаляти велодоріжки та маркери на карті	Тест пройдено успішно
✓	Робота інструкції користувача	Тест пройдено успішно
✓	Можливість переглядати, додавати, редагувати та видаляти велодоріжки на карті	Тест пройдено успішно
✓	Перевірка зміни інформації у налаштуваннях користувача	Тест пройдено успішно
✓	Перевірка локалізації	Тест пройдено успішно

Результат тестування успішний. Кожний функціонал веб-додатку який планувався на етапі визначення вимог до проекту протестований, та виповняє свої функції.

ВИСНОВКИ

Оскільки велосипедизм стає все популярнішим видом транспорту, веб-додаток має великий потенціал у поліпшенні екологічної ситуації в містах та сприянні здоровому способу життя громадян.

У ході розробки даного дипломного проекту було розроблено веб-застосунок для оптимізації функціонування міської інфраструктури екологічного транспорту, зокрема, велосипедистів. Основною метою проекту було створення зручного та інтуїтивно зрозумілого додатку, який надає велосипедистам доступ до актуальної інформації про велосипедний світ та надає корисні дані для планування велосипедних маршрутів.

В ході виконання дипломного проекту мною була розроблена серверна та клієнтська частини які взаємодіють один з одним.

За час написання дипломного проекту було поглиблено загальні базові знання в галузі веб-розробки та опановані нові технології, які мають актуальність на ринку праці. Отримано теоретичні та практичні знання з питань проектування, розробки та тестування веб-додатків.

Розроблений додаток задовольняє всім вимогам, поставленим на етапі постановки завдання та має великий модифікаційний потенціал у майбутньому.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. https://info.nic.ua/uk/blog-uk/ui-ux-design-2/ - Стаття «У чому відмінності між UX та UI дизайном»

2. https://react.dev/ – Документація з бібліотеки React.

3. https://mobx.js.org/ – Документація з MobX.

4. https://docs.nestjs.com/ – Документація з Nest.js.

5. https://developers.google.com/maps/documentation?hl=ru – Документація з використання API Google Maps.

6. React.js. Швидкий старт, Стефанов С., 2016 – 304 с.

7. UML. Основи. 3-е видання. Фаулер М. Символ, 2018 – 192 с.

8. Професійний TypeScript. Розроблення масштабованих JavaScript-додатків, Чорний, 2021 – 352 с.

9. Шаблони проектування Node.JS. - Маріо Каскіаро, Лучано Мамміно, 2017 – 396 с.

10. Розробка та аналіз вимог до програмного забезпечення. Компоненти програмної інженерії. Курсове проєктування, Л.А. Люшенко Я.В. Хіцко, Київ, КПІ ім. Ігоря Сікорського 2020 – 64 с.

11. Концепція розвитку молодіжного спорту, велосипедного руху і облаштування велосипедної інфраструктури у м. Одесі на 2013-2018 роки, Додаток до рішення Одеської міської ради № 18.07.2013р. від 3654-VI – 26 с.

12. https://code.visualstudio.com/ - Офіційний сайт середовища розробки Visual Studio Code

13. https://mui.com/material-ui/getting-started/overview/ - Material UI огляд

14. Бази даних MySQL: Навчальний посібник. — Тернопіль: Навчальна книга – Богдан, 2010.— 160 с.

ДОДАТОК А

ЛІСТИНГИ КОДУ ПРОГРАМНОГО ПРОДУКТУ

1 Контролер MarksController

```
import { Body, Controller, Delete, Get, Param, ParseIntPipe,
Post, Put, UseGuards } from '@nestjs/common'
import { JwtAuthGuard } from 'src/auth/jwt-auth.guard'
import { Permissions } from 'src/common/roles'
import { checkPermissions, CheckUser } from 'src/utils'
import { CreateMarkDto } from './dto/CreateMarkDto'
import { CreateRouteDto } from './dto/CreateRouteDto'
import { CreateRouteMarkerDto } from
'./dto/CreateRouteMarkerDto'
import { MarksService } from './marks.service'

@Controller('marks')
export class MarksController {
    constructor(private marksService: MarksService) {}

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map]))
    @Get('routes')
    getRoutes() {
        return this.marksService.getRoutes()
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_create_route]))
    @Post('routes')
    createRoute(@Body() body: CreateRouteDto) {
        return this.marksService.createRoute(body)
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_edit_route]))
    @Put('routes/:id')
    editRoute(@Param('id', ParseIntPipe) id: number) {
        return this.marksService.editRoute(id)
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_delete_route]))
    @Delete('routes/:id')
    deleteRoute(@Param('id', ParseIntPipe) id: number) {
        return this.marksService.deleteRoute(id)
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_edit_route]))
    @Post('routes/:id')
    createRouteMark(@Param('id', ParseIntPipe) id: number,
@Body() body: CreateRouteMarkerDto) {
        return this.marksService.createRouteMark(id, body)
```

```
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_edit_route]))
    @Delete('routes/deletemarker/:id')
    deleteRouteMark(@Param('id', ParseIntPipe) id: number) {
        return this.marksService.deleteRouteMark(id)
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map]))
    @Get()
    getMarks() {
        return this.marksService.getMarks()
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_edit_mark]))
    @Post()
    createMark(@Body() body: CreateMarkDto) {
        return this.marksService.createMark(body)
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_edit_mark]))
    @Put(':id')
    updateMark(@Param('id', ParseIntPipe) id: number) {
        return this.marksService.setShow(id)
    }

    @UseGuards(JwtAuthGuard, CheckUser,
checkPermissions([Permissions.can_view_map,
Permissions.can_delete_mark]))
    @Delete(':id')
    deleteMark(@Param('id', ParseIntPipe) id: number) {
        return this.marksService.setDelete(id)
    }
}
```

2 Сервіс MarksService

```
import { HttpException, HttpStatus, Injectable } from
'@nestjs/common'
import { InjectRepository } from '@nestjs/typeorm'
import { MARK_NOT_FOUND, ROUTE_NOT_FOUND } from 'src/constants'
import { Mark } from 'src/models/mark.entitie'
import { MapRoute } from 'src/models/route.entitie'
import { RouteMarker } from 'src/models/routemarker.entitie'
import { Repository } from 'typeorm'
import { CreateMarkDto } from './dto/CreateMarkDto'
import { CreateRouteDto } from './dto/CreateRouteDto'
import { CreateRouteMarkerDto } from
'./dto/CreateRouteMarkerDto'

@Injectable()
export class MarksService {
    constructor(
```

```
        @InjectRepository(Mark)
        private marksRepository: Repository<Mark>,

        @InjectRepository(MapRoute)
        private routeRepository: Repository<MapRoute>,

        @InjectRepository(RouteMarker)
        private routeMarkerRepository: Repository<RouteMarker>
    ) {}

    async getRoutes(): Promise<MapRoute[]> {
        return await this.routeRepository.find({
            relations: ['markers'],
            select: ['markers'],
        })
    }

    async createRoute(data: CreateRouteDto): Promise<MapRoute> {
        const newRoute = new MapRoute()
        newRoute.shortId = data.shortId
        newRoute.visible = 1
        return await this.routeRepository.save(data)
    }

    async editRoute(id: number): Promise<any> {
        const findRoute = await this.routeRepository.findOne({
            where: {
                id,
            },
        })

        if (!findRoute) throw new HttpException(ROUTE_NOT_FOUND,
HttpStatus.BAD_REQUEST)

        await this.routeRepository.update(
            {
                id,
            },
            {
                visible: !!findRoute.visible ? 0 : 1,
            }
        )

        return { completed: true }
    }

    async deleteRoute(id: number): Promise<any> {
        const findRoute = await this.routeRepository.findOne({
            where: {
                id,
            },
        })

        if (!findRoute) throw new HttpException(ROUTE_NOT_FOUND,
HttpStatus.BAD_REQUEST)

        await this.routeRepository.delete({
            id,
        })
    }
```

```
    async createRouteMark(id: number, data:
CreateRouteMarkerDto): Promise<RouteMarker> {
        const findRoute = await this.routeRepository.findOne({
            where: {
                id,
            },
        })

        if (!findRoute) throw new HttpException(ROUTE_NOT_FOUND,
HttpStatus.BAD_REQUEST)

        const newRouteMarker = new RouteMarker()
        newRouteMarker.lat = data.lat.toString()
        newRouteMarker.lng = data.lng.toString()
        newRouteMarker.shortId = data.shortId
        newRouteMarker.route = findRoute
        return await
this.routeMarkerRepository.save(newRouteMarker)
    }

    async deleteRouteMark(id: number): Promise<any> {
        await this.routeMarkerRepository.delete({
            id,
        })
        return { completed: true }
    }

    // markers

    async getMarks(): Promise<Mark[]> {
        const marks = await this.marksRepository.find({})
        return marks
    }

    async createMark(data: CreateMarkDto): Promise<Mark> {
        const newMark = new Mark()
        newMark.lat = data.lat.toString()
        newMark.lng = data.lng.toString()
        newMark.shortId = data.shortId
        newMark.visibility = 1
        return await this.marksRepository.save(newMark)
    }

    async setShow(id: number) {
        const findMark = await this.marksRepository.findOne({
            where: {
                id,
            },
        })

        if (!findMark) throw new HttpException(MARK_NOT_FOUND,
HttpStatus.BAD_REQUEST)

        await this.marksRepository.update(
            {
                id,
            },
            {
                visibility: !!findMark.visibility ? 0 : 1,
```

```
            }
        )

        return { completed: true }
    }

    async setDelete(id: number) {
        await this.marksRepository.delete({
            id,
        })

        return { completed: true }
    }
}
```

3 Файл auth.service.ts

```
import { HttpException, HttpStatus, Injectable } from
'@nestjs/common'
import { UsersService } from '../users/users.service'
import { JwtService } from '@nestjs/jwt'
import { User } from 'src/models/users.entitie'
import { CreateUserDto } from 'src/dto/createUser.dto'
import { InjectRepository } from '@nestjs/typeorm'
import * as sha256 from 'crypto-js/sha256'
import * as Base64 from 'crypto-js/enc-base64'
import { Repository } from 'typeorm'
import * as constants from '../constants'
import * as nodemailer from 'nodemailer'
import * as emailCheck from 'email-check'
import * as shortid from 'shortid'
import { Roles } from 'src/common/roles'

@Injectable()
export class AuthService {
    constructor(
        private usersService: UsersService,
        private jwtService: JwtService,

        @InjectRepository(User)
        private usersRepository: Repository<User>
    ) {}

    async validateUser(username: string): Promise<User | null> {
        try {
            const user: User | undefined = await
this.usersService.findOne(username)
            if (user) return user
            return null
        } catch (e) {
            throw new HttpException(e, HttpStatus.BAD_REQUEST)
        }
    }

    async login(user: any) {
        const payload = { email: user.email, id: user.id }
        return {
            access_token: this.jwtService.sign(payload),
        }
    }
```

```
    async signUp(user: CreateUserDto) {
        try {
            const findUser: User | undefined = await
this.usersRepository.findOne({
                where: {
                    email: user.email,
                },
            })

            if (findUser) {
                throw new HttpException(constants.USER_EXISTS,
HttpStatus.BAD_REQUEST)
            }

            const result = await emailCheck(user.email)

            if (!result) throw new
HttpException(constants.INCORRECT_DATA, HttpStatus.BAD_REQUEST)

            const newUser = new User()

            newUser.email = user.email
            newUser.firstname = user.firstname
            newUser.lastname = user.lastname
            newUser.password =
Base64.stringify(sha256(user.password))
            newUser.type = Roles.viewer

            const createUser = await
this.usersRepository.save(newUser)

            const payload = { email: createUser.email, id:
createUser.id }

            return {
                access_token: this.jwtService.sign(payload),
            }
        } catch (e) {
            throw new HttpException(e, HttpStatus.BAD_REQUEST)
        }
    }

    async googleLogin(req, res) {
        if (!req.user) {
            res.redirect(`${process.env.UI_PATH}`)
        }

        const findUser = await this.usersRepository.findOne({
            where: {
                email: req.user.email,
            },
        })

        if (findUser) {
            const payload = { email: findUser.email, id:
+findUser.id }

            const access_token = this.jwtService.sign(payload)
```

```
res.redirect(`${process.env.UI_PATH}/googleauth?access_token=${a
ccess_token}`)
        } else {
            const newpassword = shortid.generate()
            const newUser = new User()
            newUser.email = req.user.email
            newUser.firstname = req.user.firstname
            newUser.lastname = req.user.lastname
            newUser.type = Roles.viewer
            newUser.password =
Base64.stringify(sha256(newpassword))
            const user = await
this.usersRepository.save(newUser)

            const payload = { email: user.email, id: +user.id }

            const access_token = this.jwtService.sign(payload)


res.redirect(`${process.env.UI_PATH}/googleauth?access_token=${a
ccess_token}`)

            const transporter = nodemailer.createTransport({
                service: 'gmail',
                port: 587,
                auth: {
                    user: process.env.NODEMAILER_EMAIL,
                    pass: process.env.MODEMAILER_PASSWORD,
                },
            })

            const mailOptions = {
                from: process.env.NODEMAILER_EMAIL,
                to: req.user.email,
                subject: 'Password creation',
                html: newpassword,
            }

            await transporter.sendMail(mailOptions)
        }
    }

    async restorePassword(email: string): Promise<{ result:
boolean }> {
        try {
            const findUser = await
this.usersRepository.findOne({
                where: {
                    email,
                },
            })

            if (!findUser) throw new
HttpException(constants.USER_NOT_FOUND, HttpStatus.BAD_REQUEST)

            const newpassword = shortid.generate()

            await this.usersRepository.update(
                {
```

```
                    id: findUser.id,
                },
                {
                    password:
Base64.stringify(sha256(newpassword)),
                }
            )

            const transporter = nodemailer.createTransport({
                service: 'gmail',
                port: 587,
                auth: {
                    user: process.env.NODEMAILER_EMAIL,
                    pass: process.env.MODEMAILER_PASSWORD,
                },
            })

            const mailOptions = {
                from: process.env.NODEMAILER_EMAIL,
                to: email,
                subject: 'Password changing',
                html: newpassword,
            }

            await transporter.sendMail(mailOptions)

            return { result: true }
        } catch (e) {
            console.log(e)
            throw new HttpException(e, HttpStatus.BAD_REQUEST)
        }
    }
}
```

4 Файл roles.ts

```
export const roles = ['owner', 'admin', 'mapeditor', 'viewer']

export enum Roles {
    owner = 'owner',
    admin = 'admin',
    mapeditor = 'mapeditor',
    viewer = 'viewer',
}

export enum Permissions {
    // post permissions
    can_view_posts = 'can_view_posts',
    can_create_posts = 'can_create_posts',
    can_edit_posts = 'can_edit_posts',
    can_delete_posts = 'can_delete_posts',

    // marker permissions
    can_view_map = 'can_view_map',
    can_create_mark = 'can_create_mark',
    can_edit_mark = 'can_edit_mark',
    can_delete_mark = 'can_delete_mark',
```

```
    // route permissions
    can_create_route = 'can_create_route',
    can_edit_route = 'can_edit_route',
    can_delete_route = 'can_delete_route',
}

type PermissionsByRole = {
    [key in keyof typeof Roles]: Permissions[]
}

export const permissionsByRole: PermissionsByRole = {
    [Roles.owner]: [
        Permissions.can_view_posts,
        Permissions.can_create_posts,
        Permissions.can_edit_posts,
        Permissions.can_delete_posts,
        Permissions.can_view_map,
        Permissions.can_create_mark,
        Permissions.can_edit_mark,
        Permissions.can_delete_mark,
        Permissions.can_create_route,
        Permissions.can_edit_route,
        Permissions.can_delete_route,
    ],
    [Roles.admin]: [
        Permissions.can_view_posts,
        Permissions.can_create_posts,
        Permissions.can_edit_posts,
        Permissions.can_delete_posts,
        Permissions.can_view_map,
    ],
    [Roles.mapeditor]: [
        Permissions.can_view_posts,
        Permissions.can_view_map,
        Permissions.can_create_mark,
        Permissions.can_edit_mark,
        Permissions.can_delete_mark,
    ],
    [Roles.viewer]: [
        Permissions.can_view_posts,
        Permissions.can_view_map,
    ],
}
```

5 Файл profile.store.ts

```
import { getCookie } from 'cookie'
import { action, makeObservable, observable } from 'mobx'

export enum Roles {
    owner = 'owner',
    admin = 'admin',
    mapeditor = 'mapeditor',
    viewer = 'viewer',
}

export enum Permissions {
    // post permissions
    can_view_posts = 'can_view_posts',
    can_create_posts = 'can_create_posts',
```

```
    can_edit_posts = 'can_edit_posts',
    can_delete_posts = 'can_delete_posts',

    // marker permissions
    can_view_map = 'can_view_map',
    can_create_mark = 'can_create_mark',
    can_edit_mark = 'can_edit_mark',
    can_delete_mark = 'can_delete_mark',

    // route permissions
    can_create_route = 'can_create_route',
    can_edit_route = 'can_edit_route',
    can_delete_route = 'can_delete_route',
}

export default class ProfileStore {
    constructor() {
        makeObservable(this)
    }

    @observable loadingAuth: boolean = false
    @observable token: string | null = getCookie('token') ||
null

    @observable firstname: string = ''
    @observable lastname: string = ''
    @observable type: string = ''
    @observable permissions: Permissions[] = []

    @observable loadingUpdateProfile: boolean = false
    @observable loadingUpdatePassword: boolean = false

    @action setPermissions(data: Permissions[]) {
        this.permissions = data
    }

    @action setLoadingUpdateProfile(value: boolean): void {
        this.loadingUpdateProfile = value
    }

    @action setLoadingUpdatePassword(value: boolean): void {
        this.loadingUpdatePassword = value
    }

    @action setLoadingAuth(value: boolean): void {
        this.loadingAuth = value
    }

    @action setToken(value: string | null): void {
        this.token = value
    }

    @action setFirstname(value: string): void {
        this.firstname = value
    }

    @action setLastname(value: string): void {
        this.lastname = value
    }
```

```
    @action setType(value: string): void {
        this.type = value
    }

    @action resetStore(): void {
        this.loadingAuth = false
        this.token = null
        this.firstname = ''
        this.lastname = ''
        this.loadingUpdateProfile = false
        this.loadingUpdatePassword = false
        this.type = ''
        this.permissions = []
    }
}
```

6 Файл Home.tsx

Файл Home.tsx:

```
import { useEffect, useState } from 'react'
import { observer } from 'mobx-react'
import { Route, Routes } from 'react-router'
import CssBaseline from '@mui/material/CssBaseline'
import Box from '@mui/material/Box'
import Toolbar from '@mui/material/Toolbar'
import List from '@mui/material/List'
import Typography from '@mui/material/Typography'
import Divider from '@mui/material/Divider'
import IconButton from '@mui/material/IconButton'
import MenuIcon from '@mui/icons-material/Menu'
import ChevronLeftIcon from '@mui/icons-material/ChevronLeft'
import LogoutIcon from '@mui/icons-material/Logout'
import { MainListItems } from './listitems'
import { Badge } from '@mui/material'
import Logo from 'media/logo.svg'
import { getProfile, signOut } from 'api/auth.api'
import { AppBar, Drawer } from './components/Drawer'
import Settings from 'Pages/Settings/Settings'
import { profileStore } from 'store'
import ChangeLang from 'components/ChangeLang/ChangeLang'
import CreatePost from 'Pages/CreatePost/CreatePost'
import Posts from 'Pages/Posts/Posts'
import ViewPost from 'Pages/ViewPost/ViewPost'
import './styles.scss'
import ProtectPermission from
'components/Protect/ProtectPermission'
import { Permissions } from 'store/profile/profile.store'
import GoogleMapWrapper from 'Pages/Map/MapWrapper'

const Home: React.FC = () => {
    const [open, setOpen] = useState(true)

    const { firstname, lastname, type } = profileStore

    const toggleDrawer = () => {
        setOpen(!open)
    }

    useEffect(() => {
```

```
        getProfile()
    }, [])

    return (
        <Box sx={{ display: 'flex' }}>
            <CssBaseline />
            <AppBar position="absolute" color="default"
open={open}>
                <Toolbar
                    sx={{
                        pr: '24px',
                    }}
                >
                    <IconButton
                        edge="start"
                        color="inherit"
                        aria-label="open drawer"
                        onClick={toggleDrawer}
                        sx={{
                            marginRight: '36px',
                            ...(open && { display: 'none' }),
                        }}
                    >
                        <MenuIcon />
                    </IconButton>
                    <Typography
                        component="h6"
                        variant="h6"
                        color="inherit"
                        noWrap
                        sx={{ flexGrow: 1, display: 'flex',
alignItems: 'center' }}
                    >
                        <img className="logo" src={Logo} />
                    </Typography>
                    <Typography
                        component="h6"
                        variant="h6"
                        color="inherit"
                        noWrap
                        sx={{ flexGrow: 1, display: 'flex',
alignItems: 'center', fontSize: '16px' }}
                    >
                        {`${firstname} ${lastname} (${type})`}
                    </Typography>
                    <ChangeLang />
                    <IconButton color="inherit"
onClick={signOut} sx={{ ml: 4 }}>
                        <Badge color="secondary">
                            <LogoutIcon />
                        </Badge>
                    </IconButton>
                </Toolbar>
            </AppBar>
            <Drawer variant="permanent" open={open}>
                <Toolbar
                    sx={{
                        display: 'flex',
                        alignItems: 'center',
                        justifyContent: 'flex-end',
```

```
                        px: [1],
                    }}
                >
                    <IconButton onClick={toggleDrawer}>
                        <ChevronLeftIcon />
                    </IconButton>
                </Toolbar>
                <Divider />
                <List component="nav">
                    <MainListItems />
                </List>
            </Drawer>
            <Box
                component="main"
                sx={{
                    backgroundColor: (theme) =>
                        theme.palette.mode === 'light' ?
theme.palette.grey[100] : theme.palette.grey[900],
                    flexGrow: 1,
                    height: '100vh',
                    overflow: 'auto',
                    p: 0,
                }}
            >
                <Toolbar />
                <Routes>
                    <Route path="posts/view/:postId"
element={<ViewPost />} />
                    <Route path="posts/create"
element={<CreatePost />} />
                    <Route
                        path="posts"
                        element={
                            <ProtectPermission
permission={Permissions.can_view_posts}>
                                <Posts />
                            </ProtectPermission>
                        }
                    />
                    <Route
                        path="map"
                        element={
                            <ProtectPermission
permission={Permissions.can_view_map}>
                                <GoogleMapWrapper />
                            </ProtectPermission>
                        }
                    />
                    <Route path="settings" element={<Settings
/>} />
                </Routes>
            </Box>
        </Box>
    )
}

export default observer(Home)
```