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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)

“__” _____2022 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система геолокації та маршрутизації

Виконав : студент __4__ курсу, групи ІП-84
(шифр групи)

_____ Шмалько Богдан Ігорович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____ асистент Молчанова А. А._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) професор, д.т.н., Сімоненко В. П._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2022 р.

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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Шмалька Богдана Ігоровича

1. Тема проєкту *Система геолокації та маршрутизації*
керівник проєкту Молчанова Анастасія Анатоліївна, асистент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11 травня 2022 року №1139-с

2. Термін здачі студентом закінченого проєкту *24 червня 2022 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні та статистичні дані, прикладний програмний інтерфейс.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Системи геолокації та маршрутизації.

Розділ 2. Вибір і обгрунтування засобів реалізації.

Розділ 3. Розробка системи.

Розділ 4. Тестування та аналіз розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи(діаграма розгортання застосунку) , функціональна схема (діаграма класів), принципова схема роботи.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В. П.		

7. Дата видачі завдання «26» жовтня 2021 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*15.10.2021-30.10.2021*	
2.	*Вивчення та аналіз завдання*	*17.11.2021-05.03.2022*	
3.	*Розробка архітектури та загальної структури системи*	*05.03.2022-15.03.2022*	
4.	*Програмна реалізація системи*	*15.03.2022-15.04.2022*	
5.	*Оформлення пояснювальної записки*	*15.04.2022-05.06.2022*	
6.	*Захист програмного продукту*	*27.05.2022*	
7.	*Передзахист*	*11.06.2022*	
8.	*Захист*	*24.06.2022*	

Студент-дипломник _____ Богдан ШМАЛЬКО
(підпис)

Керівник проєкту _____ Анастасія МОЛЧАНОВА
(підпис)

АНОТАЦІЯ

У бакалаврському дипломному проєкті реалізовано систему, яка дає можливість розрахувати оптимальний маршрут від однієї точки на карті світу до іншої. В даному проєкті реалізовані алгоритми Беллмана-Форда та Флойда–Уоршелла. Система передбачає перехід на будь-який алгоритм. Реалізований перехід на алгоритм системи Bing та Openroute. Програма надає інтерфейс для взаємодії різних систем одночасно.

Для швидкої та надійної взаємодії між користувачами, чи групою користувачів було реалізовано інтерфейс на базі протоколу WebSocket та Http. Для взаємодії даної системи з будь-якою іншою системою в режимі реального часу, було розроблено інтерфейс з використанням технології WebHook.

Програмний продукт реалізовано на TypeScript.

Ключові слова: геолокація, маршрутизація, Nest.js, Node.js, TypeScript.

ANNOTATION

In this project for a Bachelor's Degree implements a system that allows to calculate the optimal route from one point on the world map to another. Bellman-Ford and Floyd-Warshall algorithms are implemented in this project. The system has a transition to the existing algorithm. Implemented the transition to the algorithm of the Bing and Openroute system. The program provides an interface for interacting with different systems simultaneously

An interface based on WebSocket and Http was implemented for fast and reliable interaction between users or groups of users. To interact with this system with any other system in real time, an interface was developed using WebHook technology.

The software product is implemented in TypeScript.

Keywords: geolocation, routing, Nest.js, Node.js, TypeScript.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система геолокації та	3		
			маршрутизації			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система геолокації та	93		
			маршрутизації			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система геолокації та	1		
			маршрутизації			
			Структурна схема системи			
			(діаграма розгортання			
			застосунку)			
	А4	***ІАЛЦ.467200.005 Д2***	Система геолокації та	1		
			маршрутизації			
			Функціональна схема			
			(діаграма класів)			
	А4	***ІАЛЦ.467200.006 Д3***	Принципова схема роботи	1		
	А4	***ІАЛЦ.467200.007 Д4***	Система геолокації та	42		
			маршрутизації			
			Текст програмного коду			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	
Розроб		Шмалько Б.І.			
Перев		Молчанова А.А.			

ІАЛЦ.467200.001 ОА

Система геолокації та маршрутизації
Опис альбому

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського ФІОТ ІП-84

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система геолокації та маршрутизації*»

Київ – 2022

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: Система геолокації та маршрутизації.

Дане технічне завдання поширюється на розробку систем прокладання маршруту на карті та визначення геолокації, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є комерційні цілі, для окремих систем, користувацьких додатків, окремих пристроїв для розрахунку найкоротшого шляху на карті.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проєкту є розгляд наявних систем, які надають можливість швидко й ефективно розв’язати завдання побудови короткого шляху на карті, реалізація алгоритмів дискретної математики та теорії графів, розробка програмного продукту, тестування програмного продукту.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література з теорії графів та дискретної математики, науково-технічна література та практики об’єктно-орієнтованого та функціонального програмування, технічна документація, публікації в періодичних виданнях, довідники, публікації в Інтернеті з даних питань.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий, функціональний та зрозумілий інтерфейс програми.
- Можливість легкого впровадження інших алгоритмів та сервісів.
- Система повинна бути автономна і не залежати від встановленого програмного забезпечення, виключаючи платформу (операційну систему).
- Зрозуміла технічна документація системи.
- Основний код покрито тестами.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Node.js версії 16.13.2 і вище.
- Visual Studio Code IDE.
- Node package manager версії 8.1.2 і вище.
- СУБД (система управлінням базами даних) MongoDB 4.4 і вище.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	15.10.2021-30.10.2021
Вивчення та аналіз завдання	17.11.2021-05.03.2022
Розробка архітектури та загальної структури системи	05.03.2022-15.03.2022
Програмна реалізація системи	15.03.2022-10.04.2022
Тестування та документування системи	10.04.2022-15.04.2022
Оформлення пояснювальної записки	15.04.2022-05.06.2022

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система геолокації та маршрутизації*»

Київ – 2022

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API	(Application Programming Interface) Програмний інтерфейс застосунка.
IDE	(Integrated Development Environment) комплексне програмне рішення для розробки програмного забезпечення
СУБД	система управлінням базами даних.
HTTP	(HyperText Transfer Protocol) Протокол передачі гіпертексту.
REST	(Representational State Transfer) передача репрезентативного стану, підхід до архітектури мережевих протоколів.
GPS	(Global Positioning System) Система глобального позиціювання.
SDK	(Software development kit) набір інструментів розробки.
JSON	(Java Script Object Notation) текстовий формат для обміну даними.

ВСТУП

Протягом останніх 10-15 років, зросла кількість компаній пов’язаних з логістикою. Стрімкий ріст вартості ведення такого бізнесу спонукала власників шукати кращі рішення оптимізації власної справи, тому ефективне планування маршрутів стало дуже актуальним в останній час.

Раніше існувало безліч методів, як люди розв’язували проблему планування маршрутів. Один з них – ручка та папір. Цей спосіб малоефективний у сучасному світі, коли треба планувати та оптимізувати маршрути у режимі реального часу. Сьогодні існує безліч програмних рішень, які надають можливість розв’язати дані проблеми, але їх ефективність, швидкість та користь не завжди виправдовують їх ціну. Крім того, системам не вистачає багатьох ключових функцій, необхідних для безпосереднього розв’язання проблем користувачів.

Програмне рішення проблеми оптимізації та маршрутизації в сучасному світі дуже ефективне, через те, що воно допомагає знизити витрати, покращити використання ресурсів та забезпечити найшвидше рішення проблем оптимізації та маршрутизації.

Постійні витрати можуть бути прямо чи опосередковано пов’язані з пройденим шляхом тому не ефективне прокладання маршруту, може призводити до жахливих наслідків. Незаплановані маршрути призводять до збільшення кількості пройденого шляху, збільшення споживання палива та збільшення часу водіння, не кажучи вже про високі витрати на технічне обслуговування автомобіля через недостатню ефективність. Плануючи маршрути за допомогою програмного забезпечення для оптимізації маршрутів, можна різко скоротити витрати на паливо, заробітну плату водіїв і технічне обслуговування транспортних засобів.

Програмне забезпечення розроблене в дипломному проєкті, використовує ефективні алгоритми з теорії графів та дискретної математики, щоб побудувати оптимальний маршрут на карті. В проєкті використовуються найефективніші рішення для вирішення проблем пов’язаних зі швидкістю розрахунку маршруту. Система запам’ятовує попередні результати визначення маршруту, на основі цих даних сама аналізує вже виконану раніше роботу й тим самим пришвидшується.

РОЗДІЛ 1. СИСТЕМИ ГЕОЛОКАЦІЇ ТА МАРШРУТИЗАЦІЇ

1.1 Необхідність використання систем геолокації та маршрутизації в сучасному світі

Розрахунок оптимального й найшвидшого маршруту на карті в сучасному світі це дуже важлива й актуальна проблема. Власники малого чи середнього бізнесу, які контролюють вимоги доставлення, мають справу з витратами на доставлення та водночас намагаються надавати кращі послуги клієнтам. В цьому виді діяльності якнайкраще допомагають автоматичні системи геолокації та оптимізації маршруту на карті.

Послідовність маршрутів — це важливий процес, який забезпечує більше зручності всім водіям та пішоходам, підвищує продуктивність роботи бізнесу. Цей процес життєво важливий для будь-якої компанії, яка має потреби в логістиці, маршрутах або послугах на місцях. Якщо потрібне рішення, яке може допомогти заощадити час і усунути щоденний хаос доставлення, планування маршруту - це єдине рішення для всіх проблем, пов’язаних зі сферою доставлення. Таким чином, зростає попит на програмне забезпечення для оптимізації маршрутів, яке допомагає створювати оптимальний шлях і керувати ним ефективніше. Багато підприємців відчувають труднощі, коли справа доходить до виконання послуг доставлення. Можуть виникнути питання, як підвищити продуктивність бізнесу? А хто буде стежити за ефективністю розрахунку маршруту для доставлення? Відповідь на ці запитання – це інтелектуальне програмне забезпечення для оптимізації та планування маршрутів, тому що, програмне забезпечення для оптимізації маршрутів допомагає заощадити час, зусилля та гроші, знаходячи найкращий варіант маршруту. Це допомагає підприємствам різних галузей ефективно планувати

маршрути та оптимізувати операції доставлення. Також програмне забезпечення допомагає користувачам орієнтуватися у раніше не знайомому місці, швидко привести до бажаного місця й заощадити час та уникнути зайвих хвилювань при пошуку ефективного маршруту.

Водії проходять через неприємності, коли використовують неточні маршрути з незнайомими дорогами, дорогами з поганим покриттям, дорогами які не підходять для їхнього транспортного засобу, або маршрути з поганим трафіком. За даними, опублікованими на вебсайті Інституту страхової інформації [1], такі ситуації змушують водіїв відчувати стрес та втому – і ці водії частіше потрапляють в ДТП. Планувальник маршрутів надасть водіям найшвидший маршрут з точними покроковими вказівками від одного місця до іншого, допомагаючи їм залишатися спокійними та зосередженими на дорозі. Особливо це важливо в сучасному світі, на тлі того, що більшість закладів харчування, переходять на спосіб доставлення готових страв безпосередньо до житла клієнтів. Саме тому, потрібно планувати маршрут, щоб швидко та ефективно доставити страву.

1.2 Огляд систем геолокації та маршрутизації

1.2.1 Google maps API

Маючи понад мільярд активних користувачів щомісяця, Google Maps було запущено у 2005 році як рішення для персональних комп’ютерів (ПК), щоб допомогти людям дістатися з «точки А в точку Б». Після понад 15 років, Google maps стали найрозповсюдженішою службою, якою користуються як користувачі, так і розробники програмного забезпечення [2].

Google maps API (Application Programming Interface) зарекомендував себе як надійний постачальник інформації по визначенню маршрутизації та

геолокації. Незалежно від того чи користувач шукає найкращий ресторан чи найближчий банк Google maps зроблять всю важку роботу за нього. Google maps надають зрозумілий API для впровадження можливостей їхньої системи у будь-який вебсайт чи мобільний застосунок, незалежно від операційної системи приладу. Google має зрозумілу документацію API, яка надає велику кількість функцій.

Одна з особливостей – це карти (рис. 1.1). Карти Google чудово інтегровані з Google maps API, тож щоб створити візуальну частину системи, достатньо використати вже наявні можливості Google. Карти Google дуже гнучкі, тож налаштувати їх стилі дуже просто під дизайн вебсайту чи застосунку для кінцевого користувача.

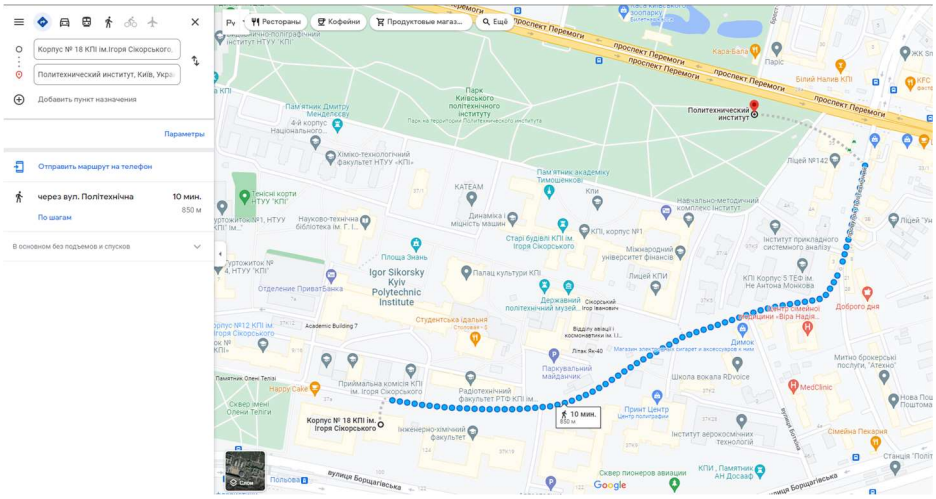


Рисунок 1.1 – Вебінтерфейс Google Maps

Понад 150 000 сайтів уже використовують API Карт Google, тож спираючись на цю статистику можна бути цілком впевненим, що API Карт Google довіряють багато користувачів та розробників [3].

Мови програмування, які використовуються для створення фреймворку Google Maps, це C++, JavaScript, та технології XML та Ajax. На початку дане програмне забезпечення діяло тільки для ПК. З ростом популярності мобільних застосунків, Google maps інтегрували програмне забезпечення на мобільні прилади. Кожен телефон на базі Android має попередньо встановлену програму Google maps. Google Maps використовує два алгоритми на графах – алгоритм

Дейкстри та алгоритм A* (алгоритм обходу графа та пошуку шляху, сформульованого спеціально для зважених графіків), щоб обчислити найкоротшу відстань від точки A (Джерело) до точки B (пункту призначення).

1.2.2 Bing

Bing надає інтуїтивно зрозумілий API для розробки програмного забезпечення. Список можливостей Bing API вичерпний, на основі якого можна створювати безмежну кількість застосунків. [4]

Microsoft Bing Maps пропонує інтелектуальну технологію показу логістики для створення карт на основі даних для служб управління автопарком і логістичних компаній.

Bing API надає інтерфейс карт (рис. 1.2), аерофотознімки, які можуть в собі поєднувати підприємства, навігаційні маршрути, схеми руху, погодні умови тощо. Також, API можна використовувати в комерційних цілях, зокрема, для логістичних компаній, для керування автопарком, розроблення комерційних маршрутів, оптимізація маршрутів із кількома зупинками, відстеження транспортних засобів тощо.

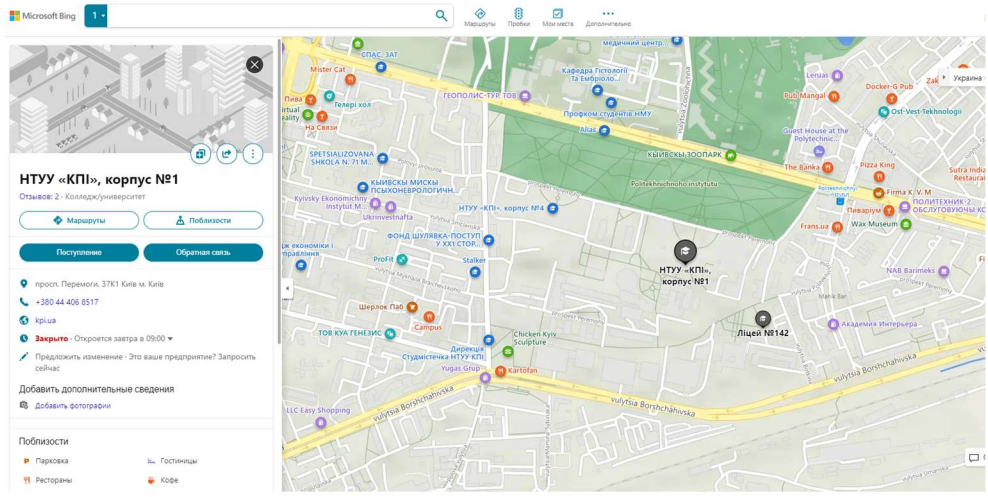


Рисунок 1.2 – Вебінтерфейс Bing Maps

API маршрутизації карт Bing дає можливість оптимізувати час у дорозі як для звичайних користувачів, так і для комерційних застосунків з функціями

аналізу місцезнаходження з використанням історичних даних. Можливості аналізу місцезнаходження надають можливість програмам взаємодіяти з фізичним світом, забезпечуючи такі функції, як AutoSuggest API і локальний пошук. Можна використовувати цей API на основі REST (Representational State Transfer) майже в кожному середовищі розробки. Підтримка орієнтованих на клієнта функцій, таких як зображення Streetside і змішана реальність, робить Bing Maps API найкращою платформою для розробки на iOS і Android. [5]

1.2.3 Open Source Routing Machine

Open Source Routing Machine (OSRM) це високопродуктивний механізм маршрутизації, написаний мовою програмування C++, розроблений для роботи з даними OpenStreetMap. Сервіс надає послуги доступні через HTTP (HyperText Transfer Protocol) API, інтерфейс бібліотеки C++ та обгортку Node.js [6].

Open Source Routing Machine надає багато можливостей для кінцевого користувача чи системи. Такі як:

- Пошук найближчого місця по координатах.
- Пошук найшвидшого маршруту між координатами.
- Обчислення тривалості або відстані найшвидшого маршруту між усіма парами наданих координат.
- Фіксувати GPS-координати (Global Positioning System) до дорожньої мережі найбільш доцільним способом.
- Розв’язання задачі комівояжера (знаходження найвигіднішого маршруту, що проходить через вказані міста) за допомогою жадібної евристики (алгоритм, спроможний вибрати прийнятне розв'язання серед багатьох рішень).
- Генерування векторної плитки Mapbox із метаданими маршрутизації [7].

Окрім хронологічної маршрутизації, OSRM також надає додаткові функції, такі як створення карти (рис. 1.3), розв'язання проблеми комівояжера та створення векторних фрагментів , які містять метадані маршрутизації.

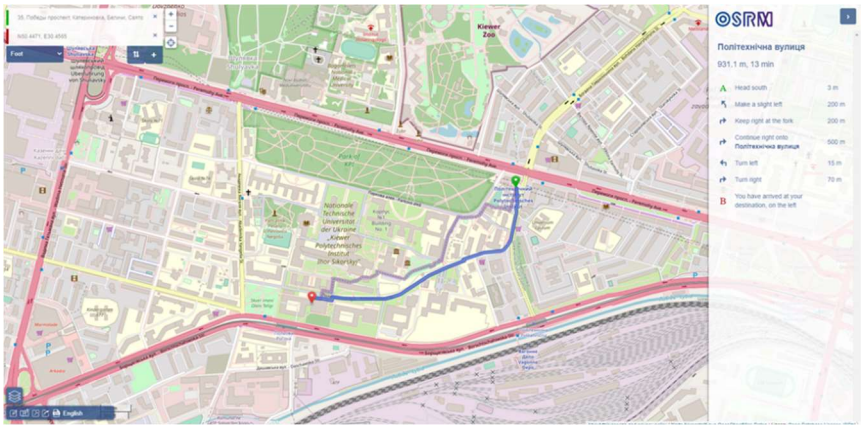


Рисунок 1.3 – Вебінтерфейс Open Source Routing Machine

На відміну від інших систем геолокації та маршрутизації OSRM не використовує алгоритм A* для обчислення найкоротшого шляху, а використовує багаторівневі ієрархії Дейкстри. Це забезпечує швидкодію, особливо для регіонів Європи. Через цю особливість, багато вебзастосунків, які потребують швидкого розрахунку в режимі реального часу використовують OSRM. OSRM відрізняється від інших систем простим форматом даних, що дає можливість замінити формат даних OpenStreetMap на дані власного формату. Більш того OSRM можна налаштовувати під різні види транспортних засобів. OSRM підтримує локалізацію покрокових текстових інструкцій.

1.2.4 YOURS navigation API

YOURS navigation API це одна з чудових альтернатив Google API. Основна ціль системи – надання маршрутів на карті. YOURS navigation API дуже гнучка у використанні й має ряд особливостей, які містять:

- Підтримка API у всіх регіонах світу
- Має різноманіття функцій

- Генерування найшвидших або найкоротших маршрутів в різних режимах
- Може використовувати всі дороги без виключення у побудові маршруту (велодороги, автомобільні, пішохідні)
- Виставлення меж побудови маршруту
- Вибір виду транспорту
- Обмеження за кількістю точок маршруту
- Оновлення даних у режимі реального часу
- Геокодування (відповідність місцю унікальний географічний ідентифікатор).

На рис. 1.3 можна переглянути демо інтерфейс та маршрут, який був побудований за допомогою даного сервісу.

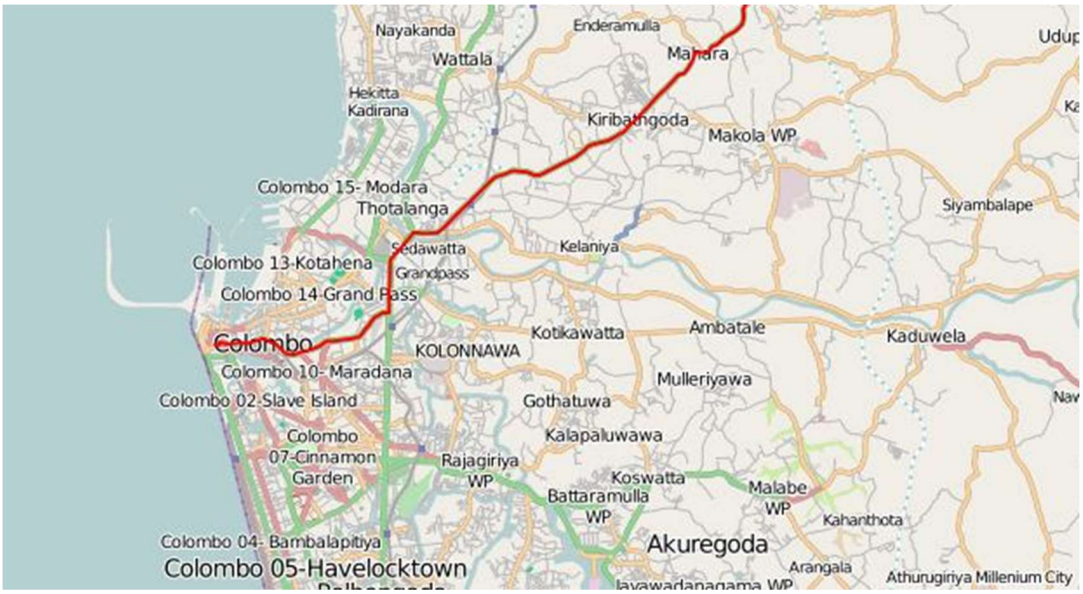


Рисунок 1.3 – YOURS API demo

Та багато інших можливостей, які можна переглянути у документації [8].

1.2.5 Graphhopper Direction Map API

GraphHopper Directions API надає можливості планування маршруту від пункту А до пункту Б, з покроковою навігацією, оптимізацією маршруту,

обчисленням ізохрон (лінії, маршрути чи явища, які мають однакове значення чи результат) та багато іншого. Програмний інтерфейс GraphHopper Directions (API) що дає можливість легко взаємодіяти різним системам, використовувати алгоритми й функції GraphHopper. API оптимізації маршрутів надає все необхідне для розв'язання різноманітних проблем маршрутизації транспортних засобів, включаючи класичну «проблему комівояжера». Алгоритми GraphHopper можуть вирішувати бізнес проблеми пов'язані з плануванням, логістикою та транспортом. Його неймовірно швидко та легко інтегрувати у кінцеву програму [9].

Можливості які надає GraphHopper Directions API:

- Мінімізація кількості зайнятих водіїв та транспортних засобів.
- Розподіл роботи для залучення максимальної кількості водіїв.
- Складання маршруту спираючись на графік та тривалість роботи водіїв.
- Підтримка великої кількості транспортних засобів, від велосипедів і скутерів до вантажівок.
- Врахування трафіку на дорозі, використовуючи мільйони зібраних анонімних даних.
- Швидке планування.
- Оптимізація маршрутів

GraphHopper Directions API створене для будь-якого відстеження та налаштування маршруту транспортних засобів. В більшості випадків застосовується в комерційних цілях. API може бути використане на будь-якій відомій карті. GraphHopper не має застосунків для кінцевих користувачів, але натомість допомагає іншим сервісам та розробникам створювати ці застосунки. API маршрутизації надає ще багато додаткової та унікальної інформації, таку як висота, клас, якість доріг, дані поверхні, точні оцінки часу та відстані. Завдяки підтримці багатьох профілів транспортних засобів, GraphHopper

ідеально підходить для прокладання маршруту для велотранспорту, бо API передбачає такі види транспорту як гірські велосипеди, електровелосипеди, звичайні велосипеди, тож це потужний інструмент для використання на природі [10].

1.2.6 ArcGIS Direction API

ArcGIS Direction API може знаходити маршрути, напрямки та виконувати розширений аналіз вуличних мереж. Система може розв’язувати проблеми, такі як створення оптимізованого маршруту з декількома пунктами зупинки, пошук найближчого транспортного засобу або об’єкта швидкої допомоги, визначення зони прокладання маршруту [11].

Також API підтримує безліч вже готових варіантів карт, та надає власний інтерфейс (рис. 1.4).

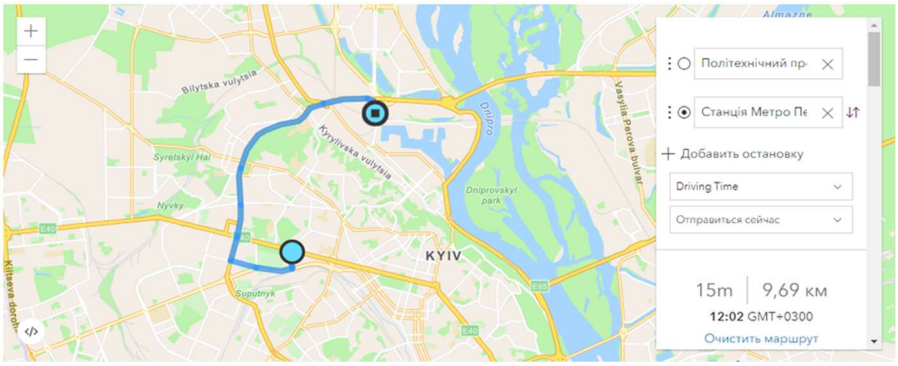


Рисунок 1.4 – Вебінтерфейс ESRI Map Direction API

На основі API можна створювати програми для вирішення таких задач:

- Знаходження найкоротшого шляху від пункту А до пункту Б
- Знаходити найшвидший шлях
- Знайти оптимальний маршрут для кількох точок призначення
- Локалізувати маршрутизацію

1.2.7 MapBox Direction Map API

Mapbox надає потужні механізми маршрутизації, точний час у дорозі та інтуїтивно зрозумілі вказівки, щоб допомогти розробникам створити навігацію для власних застосунків. Понад 600 млн активних користувачів на місяць у 45 000 застосунках Android та iOS використовують карти Mapbox. Це дуже диверсифіковане та розподілене джерело даних щомісяця генерує понад 400 мільярдів оновлень місцезнаходження в реальному часі. Повне покриття адрес у Північній Америці, Європі та інших регіонах регулярно оновлюється. Лише за останній рік додано 30 мільйонів адрес [12].

MapBox Direction Map API надає оптимізований маршрут з набором адресних точок. Дану систему можна використовувати в галузі таксі для розрахунку маршруту який включає початкове місцезнаходження транспорту, точку з місцезнаходженням клієнта та кінцеву точку маршруту. MapBox надає найсучасніші карти, які включають номери адрес, супутникові зображення, які дуже ефективно взаємодіють з їхнім API.

MapBox Direction Map API найчастіше використовується в таких випадках:

- Використання карт, які можна легко стилізувати під дизайн власного бренду.
- Оптимізація та планування маршрутів для логістичного бізнесу.
- Збереження даних місцезнаходження, що дає можливість зручно інтегруватися різним системам.
- Створення інтерфейсу для дисплея в сучасних автомобілях з підтримкою геолокації.

MapBox має власну SDK (Software development kit), яка містить велику кількість інструментів для різних типів програмних продуктів. MapBox Map SDK надає зручні й легко стилізуємі карти для галузей розробки frontend (приклад карти на рис. 1.5).

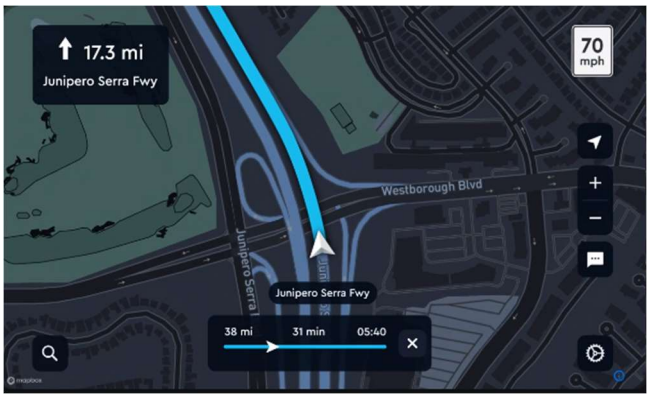


Рисунок 1.5 – Мобільний інтерфейс MapBox Direction Map API

MapBox SDK, водночас надає API для серверної розробки застосунків. На основі тільки MapBox SDK можна побудувати повноцінний мобільний чи вебзастосунок [13].

1.2.8 Here API

HERE Routing API це один з найпопулярніших постачальників API для вебзастосунків і не тільки, який пропонує легку та швидку маршрутизацію для кількох регіонів світу [14].

Наступні статичні дані, які впливають на оптимізацію та прокладання маршруту підтримуються системою HERE:

- форма шляху (наприклад, кільцеві перехрестя, пандуси, службові дороги)
- номери доріг
- напрямок транспортного потоку (наприклад, одностороннє, двостороннє, розділене шосе)
- стан дороги
- заблоковані проходи
- спеціальні обмеження
- платні дороги
- швидкісні дороги

- забудовані території
- смуги для руху автомобілів (смуги для транспортних засобів з великою кількістю пасажирів)
- сезонні закриття доріг
- Інформація про обмеження, що залежать від часу
- Інформація про пішоходів
- Інформація що до сходів
- Інформація що до ліфтів
- Доріжки через парки, площі, будівлі, мости та тунелі
- Інформація про атрибути вантажівки
- Юридичні обмеження для вантажівок (вантажівки заборонені, обмеження поворотів для вантажівок)
- Фізичні обмеження для вантажівок (вага, висота, ширина і довжина)
- Обмеження для небезпечних вантажівок [15]

HERE має також власний SDK, який підтримує інтерфейс власних карт (рис. 1.6), який легко імплементується у власні мобільні чи вебзастосунки. HERE Maps SDK чудово взаємодіє з Here API [16].

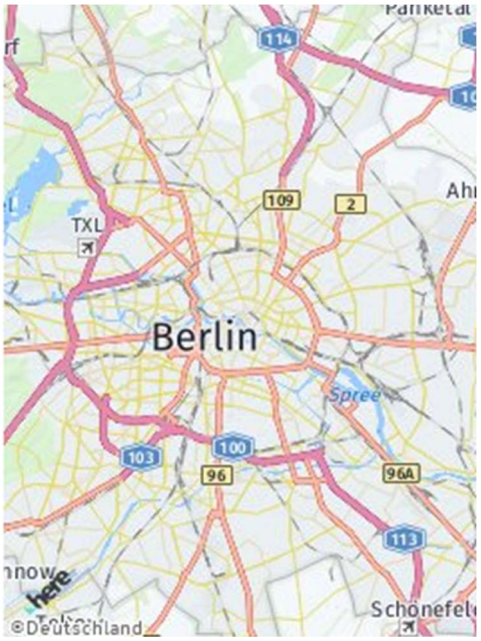


Рисунок 1.6 –Інтерфейс Here API карт

API маршрутизації надає такі можливості:

- Розрахунок маршруту через вказаний набір пунктів на карті
- Оновлення попередньо розрахованого маршруту
- Обчислення площі, яку можна пройти, подолавши задану відстань або час
- Обчислення площі, яку можна пройти за задану кількість палива чи електрики
- Розрахунок матриці маршрутів для багатьох початкових та кінцевих пунктів

1.2.9 TomTom Routing API and Extended Routing API

TomTom Routing API and Extended Routing API забезпечує кінцевих користувачів оптимізованими маршрутами та приблизним часом прибуття. API підтримує різні види та класи транспортних засобів, такі як автомобілі, електромобілі, вантажівки тощо. TomTom Routing підходить для логістичного бізнесу, оскільки має можливість обчислювати оптимальний маршрут спираючись на багато параметрів, таких як дані про трафік [17].

API дає можливість швидко й просто взаємодіяти з застосунками для кінцевих користувачів. TomTom Routing API також підтримує власний інтерфейс карт (рис. 1.7), який просто використовувати у власних вебзастосунках чи мобільних застосунках. Карти дуже ефективно взаємодіють з API TomTom Routing.

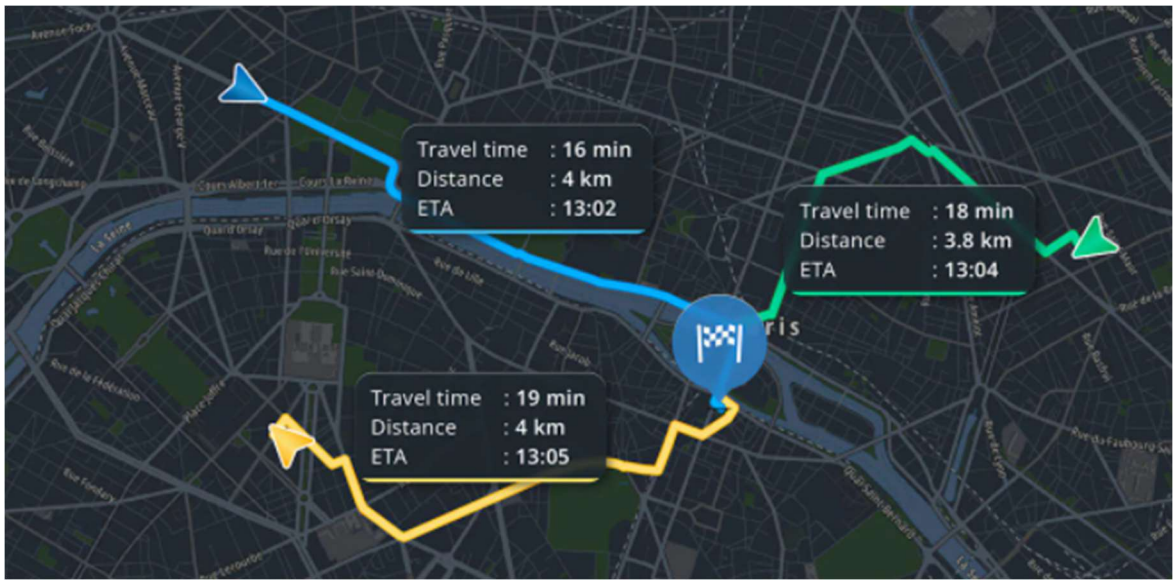


Рисунок 1.7 – Інтерфейс TomTom Routing API and Extended Routing API

TomTom Routing API має можливість прокладати маршрут, враховуючи при цьому додаткові пункти зупинок. Також, на оптимізацію та маршрутизацію впливають багато додаткових факторів таких як, врахування трафіку руху. Цей сервіс підтримує можливість ізохрон, щоб користувач сам міг вибрати оптимальний шлях з пункту А в пункт Б кількох наявних. Дане API можна використовувати у системах реального часу, так як система може враховувати положення користувача цієї миті і на основі цього повторно оптимізувати маршрут. Система допомагає водіям та пасажирам дуже зручно розраховувати власний час. Дані карт і трафіку TomTom, а також API маршрутизації розроблені для задоволення конкретних потреб у логістиці. Сервіс передбачає автоматичний та ефективний розрахунок використання транспорту з наявного в автопарку, враховуючи різні фактори транспорту, такі як: об’єм баку, габарити, допустима маса вантажу. TomTom Routing API що дає можливість збільшити точність своїх розрахунків за допомогою матричної маршрутизації. Результатом цього є матриця, яка дає можливість розрахувати відстань, час у дорозі та приблизний час прибуття, щоб відправити потрібного водія в потрібний час і в потрібне місце [18].

1.2.10 Openroute service

Openrouteservice використовується у всьому світі для прокладання й оптимізації маршруту на карті. Сервіс підтримує розширені інструкції щодо маршрутів у клієнтських програмах для автомобілів, вантажівок, різних профілів велосипедів, пішоходів чи інвалідних візків. Для оптимізації, система використовує налаштування такі як, врахування габаритів транспортного засобу чи розрахунок на основі різного роду дорожніх обмежень. В деяких випадках користувач сам хоче вирішувати, яким маршрутом рухатися, тож openrouteservice може створювати ізохрони, які показують декілька результатів для досягнення відстані з пункту А в пункт Б. Більш того, сервіс розраховує орієнтовний час в дорозі, час прибуття та пройдену відстань. Openrouteservice створений для розв'язання задачі комівояжера та оптимізації маршрутів. На основі проєкту Vroom openrouteservice надає оптимальні маршрути з урахуванням конкретного транспортного засобу та обмежень у часі [19].

Openrouteservice підтримує функції геокодування. Можна з'ясувати, які території або об'єкти є в певному місці, наприклад, школи, медичні установи, або навіть громадський транспорт. Основна технологія генерування ізохрон що дає можливість визначати величезні області з надзвичайно високою продуктивністю. Сервіс що дає можливість вказати кілька місць одночасно і по API знайти для перетинів ізохрон. Служба матриці часу й відстані Openrouteservice що дає можливість отримувати інформацію про час і відстань між набором місць (початковими пунктами і пунктами призначення) і повертає їх у структурованій відповіді JSON (Java Script Object Notation). Цей API надзвичайно зручний і масштабований для пакетних запитів, які визначають агреговані показники маршрутів. Openrouteservice передбачає додаткові параметри, які можуть вплинути на кінцеву оптимізацію маршруту, наприклад, уникнення певних типів доріг або характеристик об'єкта. Один із відомих варіантів використання, який можна побудувати на основі цієї послуги, — це

легко визначити найшвидшу або найкоротшу комбінацію набору місць призначення, до яких потрібно дістатися. Якщо вказати декілька транспортних засобів, результат буде оптимізовано для всього автопарку (рис. 1.8 демонструє приклад роботи Openrouteservice) [20].

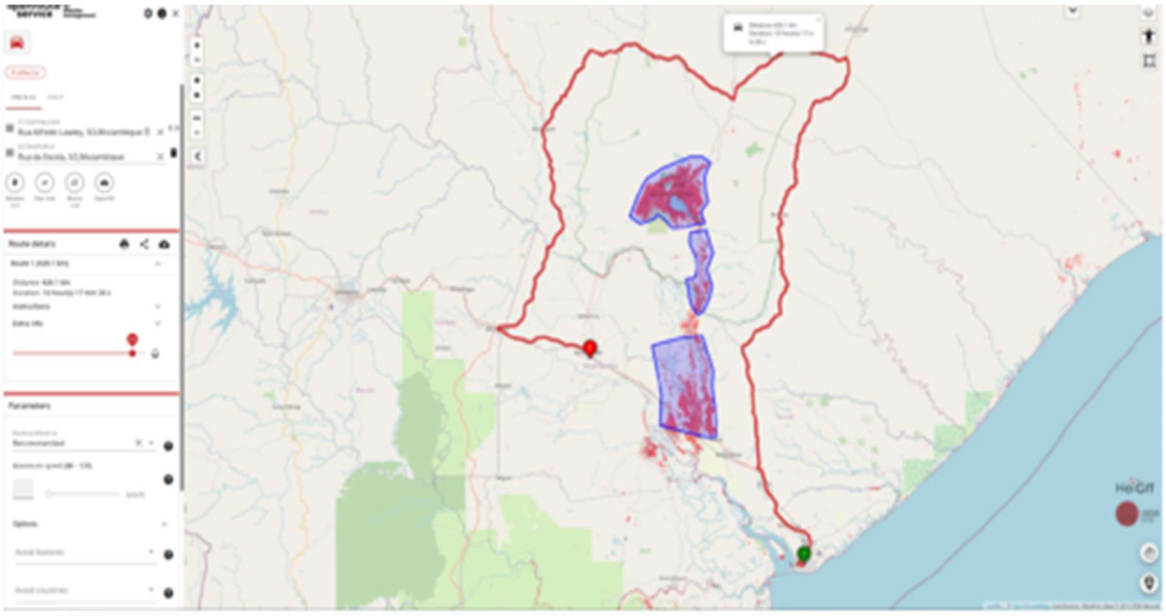


Рисунок 1.8 – Приклад роботи Openroute service

API оптимізації Openrouteservice можна використовувати для розв'язання різних проблем маршрутизації транспортних засобів. Ця універсальна послуга не тільки надасть швидкі відповіді, але й що дає можливість налаштувати параметри транспорту, завдання та часові обмеження відповідно до потреб користувача.

1.2.11 MapQuest API

MapQuest надає програмний інтерфейс для розрахунку напрямів руху. MapQuest API має багато функцій для кінцевих користувачів. Функція «Оптимізований маршрут» що дає можливість користувачам налаштовувати свій маршрут відповідно до своїх уподобань, включаючи найшвидший час у дорозі, найкоротшу відстань або навіть те, скільки часу займе пішохідна прогулянка замість водіння. Місце відправлення та призначення залишаються

фіксованими, але маршрут весь час оновлюється, щоб знайти оптимальний шлях в будь-який момент часу. Функція «Матриця маршрутів» дає можливість користувачеві бачити час і відстань між місцями. Функція «альтернативних маршрутів» дає можливість користувачам запитувати декілька потенційних маршрутів між двома місцями (ізохрони). Функція «Route Shape» забезпечує візуальний індикатор (форму) попередньо запитаного маршруту між будь-якою кількістю точок. Як правило, це лінія, що веде від місця до місця (приклад на рис. 1.9). Функція «Шлях від маршруту» надає час/відстань, необхідні для досягнення набору місць з наявного маршруту [21].

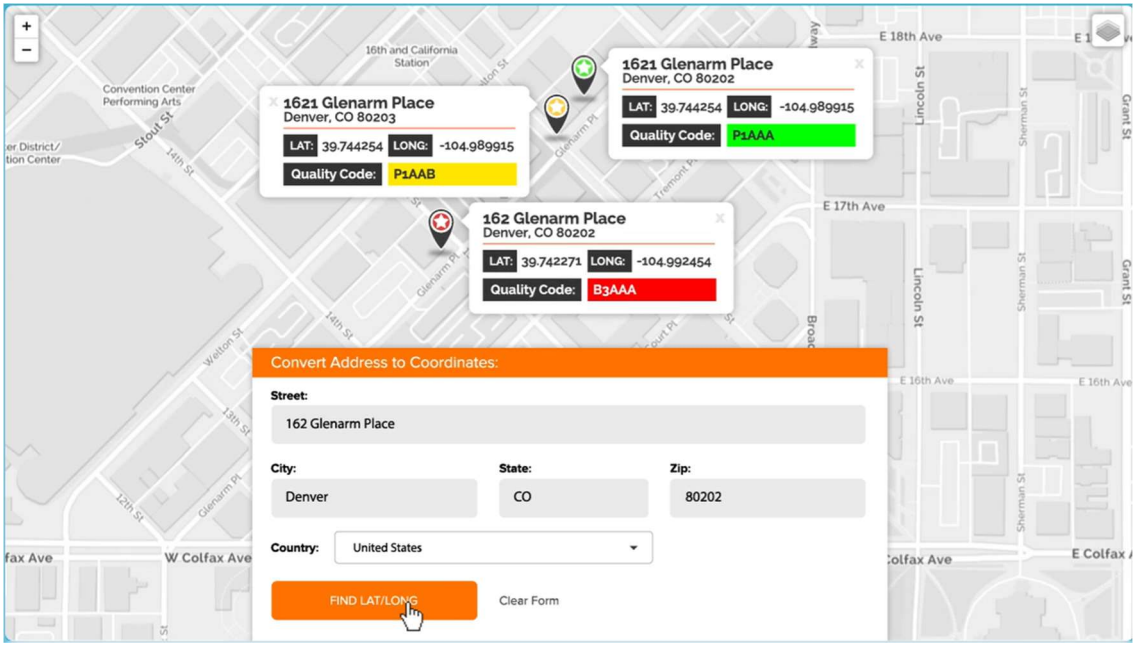


Рисунок 1.9 – Приклад Route Shape у MapQuest API

Система розв’язує проблему, як дістатися з точки A до точки B або точок C, D і E.

1.3 Порівняння розглянутих систем

Далі будуть описані у таблиці 1.1. такі переваги та недоліки систем геолокації та маршрутизації, що використовуються в сучасному програмному забезпеченні, як Google maps API, Bing, Open Source Routing Machine, YOURS

navigation API, Graphhopper Direction Map API, ArcGIS Direction API, MapBox Direction Map API, Here API, TomTom Routing API and Extended Routing API, Openroute service, MapQuest API.

Таблиця 1.1. – Переваги та недоліки систем геолокації та маршрутизації

Система	Переваги	Недоліки
Google maps API	Додатково визначає приблизний час проходження маршруту, гнучкі карти для стилізації, має багато відокремлених сервісів для маршрутизації, підтримка мобільного та веб API, підтримка різних транспортних засобів.	Не передбачає можливості створення проміжної точки маршруту стандартними методами, орієнтовано більше на користувача, а не бізнес, складність у використанні більше одного сервісу.
Bing	Підтримка мобільного та веб API, надається власний інтерфейс карт, безплатний для базового використання, надаються вже готові бізнес рішення, підтримка проміжної зупинки, підтримка різних транспортних засобів.	Великий обсяг технычної документації, надлишкова інформація, API не фільтрує інформацію.
Open Source Routing Machine	Взаємодія з GPS, надання даних в зручному вигляді, зрозуміла і доступна документація, унікальний алгоритм, швидкодія, гнучкий формат даних, підтримка локалізації.	Не взаємодіє з всіма операційними системами, не коректно працює для деяких регіонів, складне налаштування системи.

Продовження таблиці 1.1.

Система	Переваги	Недоліки
YOURS navigation API	Має унікальні можливості, підтримується у всьому світі, підтримка різних транспортних засобів.	Надлишкові можливості, створені для специфічних цілей, мала бізнес-цінність, складність у розгортанні, не підтримує інші платформи.
Graphhopper Direction Map API	Надання великої кількості додаткової інформації, легка взаємодія по API, зрозуміла й доступна документація, чудово підходить для бізнесу пов’язаний з логістикою, підтримка великої кількості видів транспорту, підтримка проміжної зупинки.	Не підходить пересічним користувачам, не надає власний інтерфейс карт, не має власного застосунка, не передбачає пішохідні маршрути.
ArcGIS Direction API	Підтримка власних карт, підтримка декількох пунктів зупинки, пошук найближчого транспортного засобу швидкої допомоги, локалізація.	Дуже прості можливості, мало орієнтований для бізнес потреб.
MapBox Direction Map API	Велика база користувачів, власний інтерфейс карт, інтегрований по всьому світу, актуальні дані, підходить для логістичного бізнесу.	Складне API, яке не підходить для маленьких та простих застосунків, об’ємна документація, надлишковість інформації, API не передбачає фільтрацію інформації.

Продовження таблиці 1.1.

Система	Переваги	Недоліки
Here API	Чудове рішення для логістичного бізнесу та вантажоперевезень, доступна та зрозуміла технічна документація, підтримка багатьох параметрів одночасно, які впливають на оптимізацію та прокладання маршруту, підтримка інтерфейсу власних карт.	Складне API, яке не підходить для маленьких та простих застосунків, підтримуються не всі регіони світу, орієнтований тільки на бізнес пов'язаний з логістикою та грузоперевезенням, існує тільки платна версія.
TomTom Routing API and Extended Routing API	Інтерфейс власних карт, різноманіття транспортних засобів, хороше рішення для бізнесу пов'язаним з логістикою та грузоперевезенням, підтримка декількох пунктів зупинки, підтримка додаткових параметрів, які впливають на оптимізацію, можливість використовувати в режимі реального часу.	Погане рішення для звичайного кінцевого користувача, дуже довга процедура отримання доступу до API.
MapQuest API	Розрахунок маршруту з врахуванням проміжних точок, підтримує унікальні функції порівняння.	Дуже прості можливості, не підтримується у всьому світі, не орієнтовано на бізнес.

Кінець таблиці 1.1.

Система	Переваги	Недоліки
Openroute service	Різноманіття транспортних засобів, підтримка додаткових параметрів, які впливають на оптимізацію, велика клієнтська база, підтримка ізохрон, розрахунок маршруту з врахуванням проміжних точок, чудове рішення як для звичайного кінцевого користувача, так і для бізнесу, безплатна, швидке отримання ключа API.	Використовує в деяких випадках допоміжне API, не використовується в режимі реального часу.

Як видно з таблиці 1.1 наявні системи в більшості випадків розроблені для потреб бізнесу і з огляду на це, вони мають дуже велике різноманіття можливостей пов'язаних з маршрутизацією та оптимізацією. В більшості випадків системи орієнтовані тільки для задоволення потреб кінцевих користувачів, або тільки для потреб бізнесу. Наприклад, MapQuest API чи Google API підходять для потреб користувачів, натомість TomTom Routing API чи HERE API це бізнес рішення. Є винятки, як от MapBox Direction Map API які орієнтовані на створення застосунків для мобільних приладів та дисплеїв автомобілів, або Bing API Openroute service, які орієнтовані як на користувача, так і на бізнес.

Більшість систем надають можливість користуватися їхніми картами, які взаємодіють з їхнім API. Наприклад Google API, Bing, HERE API надають інтерфейс карт для створення вебзастосунків та мобільних застосунків. MapBox

Direction Map API надає карти орієнтовані ще й для автомобільних дисплеїв. Майже всі системи вже мають можливість, яка передбачає прокладання маршруту враховуючи проміжний пункт. Це чудове рішення для бізнес потреб, таких як логістика чи таксі. Кожен з перелічених сервісів має свії унікальні можливості. Наприклад, MapQuest API має унікальну функцію порівняння часу, відстані для пішоходів та автомобілів, ArcGIS Direction API мє можливість знайти найближчий транспорт швидкої допомоги. Тож при виборі системи, кінцевий сервіс повинен спиратися на функції постачальника API, які задовольняють саме його потреби, тому що, не кожна система орієнтована під бізнес потреби чи користувацькі потреби. Не кожний бізнес рішення може підійти. І треба зважати на те, що деякі системи мають вже готові унікальні функції, які можуть розв’язувати основну проблему бізнесу.

У наступних розділах буде показано, як дані системи можна використовувати у системі-клієнті, на прикладі Bing API та Openroute Service.

ВИСНОВОК ДО РОЗДІЛУ 1

Задача розробки систем геолокації та маршрутизації в сучасному світі є дуже важливою. Вплив систем геолокації та маршрутизації на формування й розвиток логістичного бізнесу дуже великий. Такі системи є дуже актуальними для звичайного користувача.

В результаті порівняння найвідоміших систем, які розв’язують задачу визначення та оптимізації маршруту сьогодні, було досліджено їхні можливості та описано, які саме проблеми вони вирішують. Також було виділено основні переваги та недоліки описаних систем, обґрунтовано який вид клієнтських програм доцільно використовувати для тої чи іншої системи.

Таким чином задача розробки системи маршрутизації та геолокації сьогодні є дуже важливою для потреб кінцевих користувачів та бізнесу. Майже всі сучасні проєкти, які мають на меті використовувати карти з різних причин, мають використовувати схожі сервіси для більш швидкої та якісної роботи своєї програми.

РОЗДІЛ 2. ВИБІР І ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Вибір інструментів розробки серверної частини системи

Далі буде детально описано вибір інструментів для розробки серверної частини програми. Буде проаналізовано та вибрано мову програмування та інструменти розробки саме для технології Node.js.

2.1.1 Аналіз мов програмування для розробки сервера

Важливість вибору серверних мов програмування зростає з кожним днем. Однак вибір мови програмування є складним завданням для кожного проєкту, тому що мова програмування це основа, для створення програмного продукту. У даному розділі розглянуто і досліджено переваги та недоліки найпопулярніших та найефективніших мов програмування.

Нижче наведено найвідоміші мови програмування, для серверної розробки:

- PHP
- Ruby
- Python
- Java
- Rust
- Solidity
- Go
- Kotlin
- Node.js з JavaScript чи TypeScript

Серверні мови програмування зазвичай поділяються на два основні типи, це об’єктно-орієнтовані та функціональні.

Об'єктно-орієнтоване програмування (ООП) - це підхід до розробки сервера, який зосереджує увагу на формуванні об’єктів чи класів з полями та функціями. Техніка об’єктно-орієнтованого програмування є найкращим підходом, коли розробники працюють у команді над великими та складними проєктами.

PHP, Java, Ruby і Python це приклади серверного ООП. Хоч ООП має багато переваг, але експерти критикують ООП через його складний стиль написання коду.

Функціональне програмування (ФП) - це тип програмування, який більше зосереджується на оголошеннях і результатах, ніж на процедурі програмування. ФП підтримує паралельні незмінні дані та не має жодних побічних ефектів. Отже, ФП є хорошим вибором, якщо треба підвищити продуктивність і модульність. SQL, R, Haskell і F# — це відомі серверні мови програмування на базі ФП. Ці серверні мови програмування також мають деякі обмеження. Складне вивчення ФП для початківців, складне обслуговування та будування складної архітектури є деякими основними недоліками використання функціональних мов програмування [22].

В опитуванні розробників Stack Overflow за 2021 рік [23] йдеться про те, що JavaScript є найпопулярнішою мовою програмування (рис. 2.1). Дійсно, JavaScript отримав 64,96%, Python 48,24%, а Java 35,35% голосів від професійних розробників у цьому опитуванні. Треба ще враховувати той факт, що технологія Node.js (33,91%) використовує мови програмування JavaScript та TypeScript.

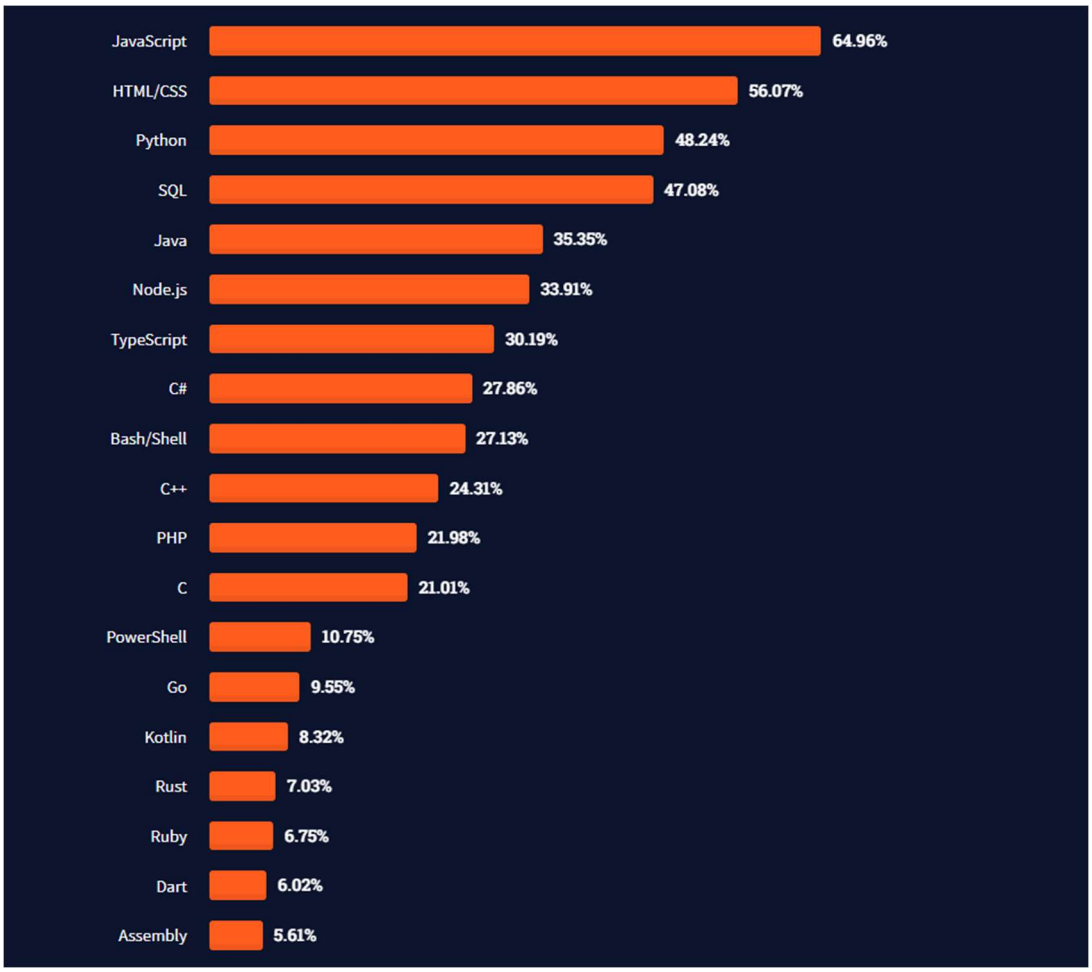


Рисунок 2.1 – Діаграма найкращих мов програмування у 2021 [23]

Зі списку мов програмування для серверної розробки можна виокремити найбільш популярні й ефективніші, це Python, Java, Node.js з JavaScript чи TypeScript. При виборі мови програмування треба враховувати всі переваги та недоліки, та розрахувати що буде корисним, а що не дуже для даного проєкту.

Python — це мова програмування високого рівня, яка має синтаксис, схожий на англійську. Це полегшує читання та розуміння коду. Python — дуже продуктивна мова. Завдяки простоті Python розробники можуть зосередитися на написанні коду, не зупиняючись на його оптимізації. Python є інтерпретованою мовою, що означає, що Python безпосередньо виконує код рядок за рядком. Python автоматично призначає тип даних під час виконання коду. Python є інтерпретованою мовою і мовою динамічного типу, порядкове

виконання коду часто призводить до повільного виконання. Щоб забезпечити простоту розробнику, Python має використовувати великий обсяг пам'яті [24].

Java була однією з найпоширеніших і найефективніших мов програмування протягом дуже тривалого часу. Причина в тому, що вона має багато переваг, які допомагають програмістам легко розв’язувати складні проблеми. Однією з головних переваг Java є те, що це об'єктно-орієнтована мова програмування. У таких мовах, як C і C++, використовуються покажчики, які надають можливість отримати доступ до місця в пам'яті. Це загрожує безпеці та може призвести до несанкціонованого доступу до пам’яті. Java використовує такі концепції ООП, як інкапсуляція, абстракція, успадкування, що підвищує безпеку та запобігає несанкціонованому доступу користувачів. Java займає більше пам'яті в порівнянні з іншими мовами програмування, такими як C і C++ [25].

Node.js - це платформа Java Script з відкритим вихідним кодом, яка використовується середовищем виконання для легкої розробки серверних програм і мережевих застосунків. Побудований на рушію Java Script V8 від Chrome, Node.js використовує неблокуючу модель виводу, керовану подіями, що робить його ефективним і легким. Node.js є середовищем виконання з відкритим кодом, тому його можна використовувати безплатно за ліцензією MIT. Node.js — це кросплатформна платформа, яка працює на Windows, Mac, Linux. Node.js також підтримує багато модулів з відкритим вихідним кодом. Мова Javascript (або її розширена версія - TypeScript) використовується для написання коду в Node.js. Node.js використовує V8, розроблений Google для Chrome. V8 безпосередньо компілює JavaScript у машинний код, що призводить до більшої швидкості та ефективного виконання коду. Node.js використовує асинхронне програмування. Кожна операція введення-виводу не блокується, тож можна виконувати кілька операцій одночасно. Платформа Node.js надає інструменти для дуже легкої та швидкої розробки REST API, але Node.js API має деякі проблеми з узгодженістю. У більшості випадків новий API містить

багато змін, тоді програмісти повинні внести зміни в код, щоб зробити його сумісним з попередньою версією програми. У більшості випадків Node.js не коректно взаємодіє з реляційними базами даних, тож доводиться використовувати noSQL бази даних [26].

Дослідивши сучасні мови програмування для серверної розробки, можна зробити висновок, що кожна мова має свої переваги та недоліки. В дипломному проєкті було вибрано технологію Node.js з мовою програмування TypeScript, через її легку взаємодію з http, вебсокетами та noSQL базами даних. Взаємодія з http, вебсокетами та noSQL базою даних це основна робота, в даному дипломному проєкті.

2.1.2 Інструменти розробки на базі Node.js

Найпопулярнішими технологіями для розробки серверної частини на базі Node.js є фреймворки Express.js, Koa.js та Nest.js.

Express.js найпопулярніший фреймворк Node.js для розробки серверних програм. За даними NPM (пакетний менеджер для Node.js), тільки за допомогою їхнього пакетного менеджера Express.js було завантажено більше ніж 24 мільйона разів на тиждень [27]. На офіційній сторінці github з відкритим кодом Express.js має 57 тисяч зірок [28]. Перевагою і водночас недоліком Express.js є те, що архітектуру проєкту на даній технології можна розробляти як забажає сам програміст. Це надає гнучкі можливості в розробці архітектури, але може дуже сильно заплутати проєкт, навіть привести до неможливості продовжувати його розробку. Фреймворк є швидким, надійним та продуктивним, Express.js дає змогу створювати надійний та безпечний програмний інтерфейс. З цих причин Express.js популярний у всьому світі та його вже використовують такі компанії як Twitter та Uber. Серверна розробка на даному фреймворку легка й інтуїтивно зрозуміла. На рис. 2.2 наведено весь код найпростішого застосунку на Express.js. Цей код вже що дає можливість

створити сервер, який працює на порту 3000 і обробляє головну сторінку застосунку.

```
const express = require('express')
const app = express()

app.get('/', function (req, res) {
  res.send('Hello World')
})

app.listen(3000)
```

Рисунок 2.2 – Найпростіший застосунок на Express.js [28]

Express.js це хороше рішення, якщо треба швидко й ефективно побудувати серверний застосунок. Для даного фреймворку створено багато бібліотек, без яких складно використовувати Express.js, такі як: cors для налаштування заголовків для браузерів, розширення для налаштування кінцевих точок (endpoint) і різні бібліотеки для проміжного рівня програмного забезпечення (middleware).

Koa.js це ще один популярний фреймворк для Node.js. Koa.js був розроблений тими ж розробниками, які створили Express.js з урахуванням багатьох проблем та помилок. Цей фреймворк зарекомендував себе, як чудове рішення для розробки вебзастосунків чи API на базі Node.js. Основна перевага Koa.js над Express.js, це підтримка асинхронних функцій. Також Koa.js що дає можливість дуже зручно обробляти помилки. Для роботи фреймворку не потрібні допоміжні модулі, тож розробка на Koa.js дуже швидка й не потребує допоміжного допрацювання чи пошуку вже готових рішень. По архітектурі та стилю написання коду Koa.js дуже схожий на Express.js тому перехід від одного до іншого фреймворку дуже легке [29]. На рис. 2.3 можна побачити найпростіший застосунок на Koa.js.

```
const Koa = require('koa');
const app = new Koa();

// response
app.use(ctx => {
  ctx.body = 'Hello Koa';
});

app.listen(3000);
```

Рисунок 2.2 – Найпростіший застосунок на Koa.js [30]

Для початківців Koa.js не найкраще рішення, тому що фреймворк передбачає знання по специфікації архітектури, та має меншу підтримку зі сторони інших розробників.

З найпопулярніших інструментів для серверної розробки для Node.js фреймворк Nest.js найновішим. Як пишуть розробники Nest.js “Nest.js це прогресивний Node.js фреймворк для створення ефективних, надійних та масштабованих серверних застосунків” [31]. Якщо потрібно створити сервер з фіксованою архітектурою, легкий у підтримці й інтеграції нових розробників, з можливістю постійного масштабування, тоді Nest.js – найкращий вибір. Nest.js це фреймворк, який використовує як основу exspess або fastify фреймворки з використанням мови програмування TypeScript. Наразі Nest.js дуже популярний та має більш ніж 47 тисяч зірок на github [32]. Спираючись на статистику NPM Nest.js щотижня завантажується близько 1.3 мільйона разів [33]. Nest.js сумісний з новітніми можливостями Node.js та TypeScript, такими як об’єктно-орієнтоване програмування та функціональне програмування. Nest.js має багато переваг над іншими фреймворками, а саме:

- Масштабування – легко розширювати новими функціями та можливостями, а також, легко підключати сторонні бібліотеки.

- В собі зразу ж має набір готових рішень та інструментів, для розв’язання будь-якої проблеми, тож завантажувати додаткові бібліотеки не завжди потрібно
- Використовує найновіші технології Node.js та TypeScript і завжди оновлюється з оновленням даних технологій

В бакалаврському проєкті було вибрано фреймворк Nest.js, оскільки хоча для Node.js (і серверного JavaScript) існує багато чудових бібліотек, допоміжних засобів та інструментів, жодна з них ефективно не розв’язує головну проблему архітектури. Фреймворк найсучасніший на час написання програмного продукту та ідеально підходить для виконання основного завдання проєкту.

2.2 Nest.js фреймворк для Node.js

Опис та переваги фреймворку Nest.js над іншими описано в розділі 2.1.2. Nest.js надає готову архітектуру застосунків, яка що дає можливість розробникам і командам створювати добре тестовані, масштабовані, слабо пов’язані та легко обслуговувані програми.

Фреймворк Nest.js запозичив підхід до архітектури та синтаксису в іншого фреймворку Angular, який використовується для створення клієнтських застосунків. Саме тому розробники, які працювали з Angular дуже легко можуть перейти з розробки клієнтських застосунків (frontend) на розробку серверних застосунків (backend), що значно пришвидшує процес створення програмного продукту. Через те, що Nest.js використовує об’єктивно-орієнтований підхід до створення застосунків, розробники, які працювали з фреймворком Java Spring boot (мовою програмування Java), зможуть легко зрозуміти код та архітектуру Nest.js, обидва фреймворки використовують програмний підхід «декоратори» для написання зручного та зрозумілого коду.

Nest.js складається з 3-х основних блоків – це контролери, провайдери та модулі.

Модуль це найпростіший клас, який обгортається декоратором @Module(), в який можна додати інші модулі, зареєструвати контролер та забезпечити передачу сервісів [34].

Контролери використовуються для обробки запитів, які приходять від кінцевих клієнтів, а також для створення автоматичної документації API та програмних модулів. Контролери часто використовують для валідації даних, що дуже спрощує розробку і не потребує написання додаткових методів. Щоб створити контролер, достатньо до класу дописати декоратор @Controller() та додати до модуля [35].

Провайдери використовуються для написання основної логіки проєкту. Провайдери можуть бути сервісом, репозиторієм, чи загальним класом-помічником. Щоб створити провайдер, треба до класу додати декоратор @Injectable() та зареєструвати його в модулі [36].

Провайдери-сервіси, це класи, які як правило додаються до контролера за допомогою паттерна програмування Dependency injection (коли клас можна використати з іншого класу) і виконують основну логіку по роботі з кінцевими запитами від контролера.

Провайдери-репозиторії, це класи, які, як правило, додаються до провайдерів-сервісів за допомогою шаблону проектування програмування Dependency injection. Цей вид провайдера створений тільки для взаємодії з базою даних.

Провайдери-помічники, це класи, як правило додаються до провайдерів-сервісів за допомогою паттерна програмування Dependency injection. Вони створені для багаторазового використання в багатьох місцях. Вони імплементують, якусь загальну логіку та поведінку програмного продукту.

2.3 Мова програмування Typescript

TypeScript (TS) – це сучасна мова програмування, розроблена за підтримки компанії Microsoft. TypeScript розширює мову програмування JavaScript (JS). Основне, що відрізняє TypeScript від JavaScript це статична типізація. Динамічна типізація – один з основних недоліків мови програмування JavaScript. При розробці великих програм динамічна типізація нерідко призводить до появи помилок.

TypeScript може бути використаний як для розробки клієнтської, так і серверної частини програми. TypeScript може бути використаним у будь-якому місці де можна використати JavaScript, тому що в результаті компіляції – весь код TypeScript перетвориться у код JavaScript. TypeScript має ліцензію Apache 2.0. і включена до редактора коду Microsoft Visual Studio 2013, поряд із C# та іншими мовами Microsoft.

За останні кілька років TypeScript стає все популярнішим. Один з найпопулярніших фреймворків для розробки користувацьких застосунків Angular використовує TypeScript. Близько 60% програмістів JS вже використовують TypeScript.

JavaScript це основна мова написання вебзастосунків, але сьогодні JS можна використовувати як на стороні клієнта, так і на стороні сервера, використовуючи такі технології як Node.js і Deno.

За статистикою Microsoft, TS вирішує близько 15% всіх помилок JavaScript. Динамічна типізація часто породжує помилки в програмі, що сильно знижує ефективність роботи програміста і зупиняє розробку через додавання нового коду. Таким чином, нездатність JavaScript включати такі речі, як типи та перевірку кода на помилки під час компіляції, робить його поганим вибором для серверної розробки на великих проєктах. Як говорить їх девіз, TypeScript – це JavaScript, який масштабується.

На відміну від інших мов, які додають статичну типізацію до JavaScript, таких як CoffeeScript (який додає синтаксичний цукор) або PureScript (який зовсім не схожий на JavaScript), TypeScript не треба вчити з початку для написання програм. Типи в TS є необов'язковими, і кожен файл JS є дійсним файлом TypeScript.

Таким чином, TS можна використовувати для тих самих цілей що й JS, але він значно кращий для розробки великих проєктів та для написання серверної частини програми.

Мови програмування діляться на дві категорії: статично типізовані або динамічно типізовані. У мовах зі статичною типізацією тип змінної повинен бути відомий під час компіляції. Якщо ми оголошуємо змінну, компілятор повинен знати чи це буде число, рядок чи логічне значення (приклад мови – Java). У мовах з динамічним типом це не обов'язково. Тип змінної відомий лише під час запуску програми (приклад мови – Python).

TypeScript може підтримувати статичну типізацію, тоді як JavaScript ні. TypeScript підтримує основні типи, як Boolean, Number, String, Array та інші [37].

2.4 Visual Studio Code

Visual Studio Code (VS Code) — безплатний текстовий редактор з відкритим вихідним кодом від Microsoft. VS Code доступний для Windows, Linux і macOS, тому користувач може почати роботу незалежно від платформи. Хоча редактор є відносно легким, він містить деякі потужні функції, які зробили VS Code одним із найпопулярніших інструментів розробки програмних продуктів.

Продуктивний процес редагування та налагодження коду що дає можливість зосередити більше уваги на написання коду, а не пошук чи редагування.

Visual Studio Code це чудовий текстовий редактор для написання коду, для повсякденного використання. Завдяки підтримці сотень мов VS Code допомагає миттєво працювати з підсвічуванням синтаксису, узгодженням дужок, автоматичним відступом, виділенням поля, фрагментами тощо. Інтуїтивно зрозумілі комбінації клавіш, легка настройка та зіставлення комбінацій клавіш, створені спільнотою, надають можливість легко переміщатися по коду. Visual Studio Code включає розширене розуміння семантичного коду та навігацію, а також рефакторинг коду.

Visual Studio Code включає інтерактивний налагоджувач, тож можна переходити через вихідний код, перевіряти змінні, переглядати стеки викликів та виконувати команди на консолі.

VS Code також інтегрується з інструментами збирання та написання сценаріїв для виконання звичайних завдань, що прискорює повсякденні робочі процеси. VS Code підтримує Git, тому можна працювати з системою керування кодом, не виходячи з редактора.

VS Code можна налаштовувати на свій смак, або завантажувати додаткові розширення для текстового редактора, якщо є така потреба. Хоча більшість сценаріїв працюють «з коробки» без конфігурації, VS Code також розвивається разом із розробником. VS Code — це проєкт з відкритим вихідним кодом, тому кожен може зробити свій внесок у зростаючу та активну спільноту на GitHub. Visual Studio Code включає загальнодоступну модель розширюваності, яка що дає можливість розробникам створювати та використовувати розширення.

VS Code включає вбудовану підтримку для розробників Node.js за допомогою JavaScript і TypeScript, що працює на основі тих самих базових технологій, які побудували Visual Studio.

Архітектурно Visual Studio Code поєднує в собі найкращі вебтехнології, нативні та мовні технології. Використовуючи Electron, VS Code поєднує вебтехнології, такі як JavaScript і Node.js. Крім того, VS Code використовує архітектуру служби інструментів, яка що дає можливість йому інтегруватися з багатьма з тих самих технологій, що й у Visual Studio, включаючи Roslyn для .NET, TypeScript [38].

2.5 Mongodb database

MongoDB — це NoSQL документно-орієнтована база даних, яка чудово підходить для збереження величезного обсягу даних. Якщо порівнювати з реляційними базами даних, де використовуються таблиці та рядки, то MongoDB використовує колекції та документи. Якщо детальніше розглядати структуру документа, то це набір даних які зберігаються у вигляді ключ-значення, які є основною одиницею даних у MongoDB. Колекції містять набори документів і функцій, що є еквівалентом таблиць реляційної бази даних.

Кожна база даних містить колекції, які своєю чергою містять документи. Кожен документ може відрізнятися різною кількістю полів. Розмір і зміст кожного документа можуть відрізнятися один від одного. Структура самого документу чимось нагадує, як самі розробники будують свої класи та об’єкти на різних мовах програмування. Тому можна порівнювати документ з класом, тому що, вони не є рядками чи стовпцями, а мають фіксовані пари ключ-значення. Одна з особливостей MongoDB, що документи не повинні мати чіткої визначеної схеми даних. Натомість поля для схеми можуть бути динамічно створюватися. Модель даних, доступна в MongoDB, що дає можливість легше представляти ієрархічні зв’язки, зберігати масиви та інші складніші структури.

Одна з основних особливостей MongoDB це масштабованість. Компанії по всьому світу використовують кластери MongoDB, а деякі з них мають понад 100 вузлів з близько мільйоном документів у базі даних.

Оскільки MongoDB є базою даних типу NoSQL, і зберігає дані у документах, тому це робить базу даних дуже гнучким і адаптованим до реальної ситуації та бізнес вимог. MongoDB має складну систему пошуку за полями, та пошук по регулярним виразам. Можна робити запити для повернення конкретних полів у документах. Одна з найбільших переваг MongoDB це індексування. Індексування даних дуже пришвидшує швидкість та якість пошуку даних. Будь-яке поле в документі MongoDB можна проіндексувати. Ще одна особливість MongoDB це реплікація. База даних може забезпечити високу доступність за допомогою наборів реплік. Набір реплік може складатися з більш ніж двох екземплярів MongoDB. Кожен з набору реплік може виконувати роль первинної або вторинної репліки в будь-який час. Основною реплікою є основний сервер, який взаємодіє з клієнтом і виконує всі операції читання/запису. Вторинні репліки зберігають копію даних первинної за допомогою вбудованої реплікації. Коли первинна репліка виходить з ладу, набір реплік автоматично перемикається на вторинний, а потім стає основним сервером.

В сучасному світі, коли програму, відповідно й базу даних, можуть використовувати одночасно мільйони користувачів, постає проблема горизонтального масштабування. Балансування навантаження у MongoDB чудово реалізовано, шляхом розподілу даних між кількома екземплярами бази даних. MongoDB може працювати на кількох серверах, балансуючи навантаження та/або дублюючи дані, щоб підтримувати роботу системи в разі збою обладнання.

Одна з особливостей реляційних баз даних це забезпечення цілісності даних. Для MongoDB це не явна вимога. Реляційні бази даних вимагають попередню нормалізацію даних перед записом, щоб уникнути дублюванню чи

породженню пустих полів. Нормалізація бази даних вимагає більшої кількості додаткових таблиць, що призводить до більшої кількості зв'язків між таблицями, що своєю чергою призводить до створення більшої кількості ключів та індексів. Оскільки бази даних починають рости, продуктивність може почати ставати проблемою. MongoDB є гнучким і не потребує попередньої нормалізації даних [39].

2.6 Heroku

Heroku — це платформа, на яку часто покладаються розробники серверного програмного забезпечення. Платформа дає змогу розробникам здійснювати безпроблемне розгортання застосунків, масштабування й керування ними. Heroku підтримує широкий спектр мов програмування, таких як Java, Ruby, PHP, Node.js, Python, Scala та Clojure. Heroku запускає програми через віртуальні контейнери, які називаються Dynos.

Heroku розгортає проєкти користувачів на основі кількох своїх віртуальних машин. Сама платформа разом з програмами користувачів використовує Amazon Web Services як базову інфраструктуру розгортання. Це що дає можливість розробникам зосередити увагу на розробці продукту, а не його розгортанні, що досить зручно.

Dyno — це контейнер на платформі Heroku, який використовується для запуску та масштабування програм Heroku. По суті, це віртуальні контейнери Linux, які використовуються для виконання коду на основі команд користувача. Програми можна масштабувати до певної кількості Dynos відповідно до вимог розробників. Heroku пропонує функції керування контейнерами, які допомагають користувачам легко масштабувати та керувати розміром, типом і числом Dynos на основі вимог програми. Тому розробники отримують свободу від необхідності виконувати завдання з управління інфраструктурою і замість

цього можуть зосередитися на важливих аспектах створення та запуску програм.

Одна з основних переваг Heroku це те, що розробники можуть використовувати платформу з безплатним рівнем, що для багатьох маленьких проєктів є цілком прийнятним рішенням.

Heroku має зрозумілий вебінтерфейс для розробників, що що дає можливість розгортати проєкти користувацькими методами, а не програмними. Розгортання одним клацанням миші є досить зручним для розробників для запуску програми. Функція автоматичного масштабування Heroku допомагає легко виявляти стрибки трафіку та відповідно створювати більше Dynos.

Heroku пропонує розробникам безпечний спосіб розробки програм завдяки набору функцій безпеки. Ця платформа зменшує вимоги розробників у створенні постійних виправлень безпеки, що може бути проблемою, особливо в складніших вебзастосунках. Heroku забезпечує оптимальний рівень безпеки для серверів, коду програми та запобігає будь-яким можливим проблемам з безпекою програми.

Інтерфейс командного рядка (CLI) Heroku — це функція, яка допомагає зручно розгортати програми на Heroku і керувати ними [40].

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі було обрано інструменти для розробки сервісу системи геолокації та маршрутизації. Було проведено аналіз серверних мов програмування та технічних рішень. Було детально описано принцип роботи фреймворку Nest.js, проаналізовано особливості мови програмування TypeScript. Описано інструменти для розробки програмного забезпечення, а саме Visual Studio Code. Обґрунтований вибір бази даних MongoDB, та платформи розгортання проєкту Heroku.

Спочатку було досліджено основні сучасні серверні мови програмування. Проведено аналіз, по категоріях цих мов. Описано основні недоліки та переваги об’єктно-орієнтованого та функціонального підходів у програмування. Було вибрано 3 основні мови програмування для аналізу, це Python, Java, Node.js з JavaScript чи TypeScript. Було досліджено ці мови з різних сторін. В результаті аналізу було обрано Node.js з JavaScript чи TypeScript для розробки програмного продукту проєкту.

Потім було проаналізовано основні інструменти на базі технології Node.js. Проаналізувавши серверні технології для розробки на Node.js, було обрано 3 найкращі, це Express.js, Koa.js та Nest.js. В результаті, було вибрано фреймворк Nest.js, оскільки на день написання дипломного проєкту це найновіша й найефективніша технологія для розробки складного й архітектурно структурованого проєкту.

В наступному розділі було описано мову програмування TypeScript, її особливості над JavaScript, та які проблеми вона вирішує при розробці програми.

Далі було описано програмне забезпечення для редагування коду - Visual Studio Code. Було розглянуто особливості програми, а також чим саме вона

підходить для розробки проєкту на базі фреймворку Nest.js та мовою програмування TypeScript.

Також було розглянуто базу даних MongoDB, її особливості, та переваги над реляційними базами даних.

В кінці розділу було обрано платформу Heroku для розгортання проєкту, оскільки вона безплатна та задовольняє всі особливості дипломного проєкту.

В результаті аналізу було обрано технологію, мову програмування, редактор коду, базу даних, та платформу для розгортання проєкту.

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ

Відповідно до обраних в розділі 1 сервісів, в розділі 2 технологій, мови програмування, бази даних та програмного забезпечення, можна приступити до розробки програмного продукту проєкту. Для того, щоб створити систему, спочатку необхідно описати архітектуру й побудувати програмні модулі. Після написаного основного коду проєкту треба створити модулі «заглушки» й покрити наявний код тестами. Після цього можна приступати до створення автоматичної документації системи.

3.1 Архітектура основної директорії

На рис. 3.1 продемонстрована архітектура файлів основної директорії програми.

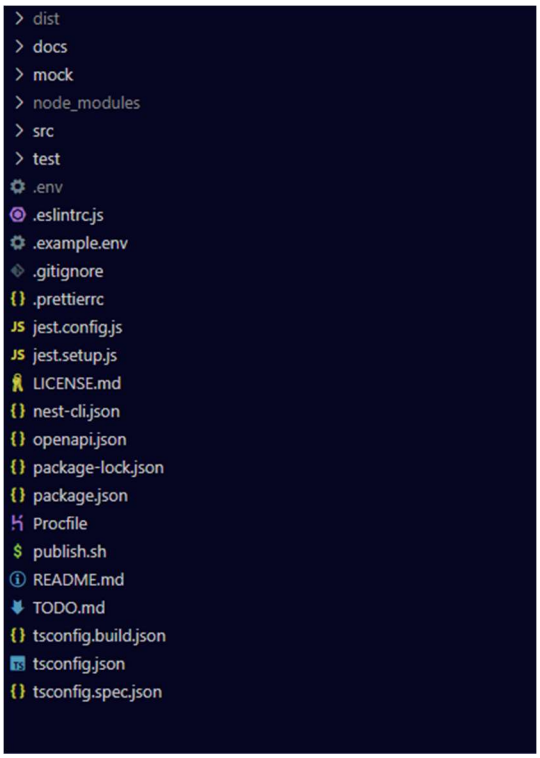


Рисунок 3.1 – Файлова архітектура основної директорії проєкту

В корені проєкту знаходяться основні файли потрібні для конфігурації, збору й тестування програми.

Файл «.env» - це файл конфігурації проєкту, який відповідає за збереження статичних даних, секретних ключів і елементів конфігурації для основної програми. На рис. 3.2 можна переглянути заповнений .env файл тестовими даними.

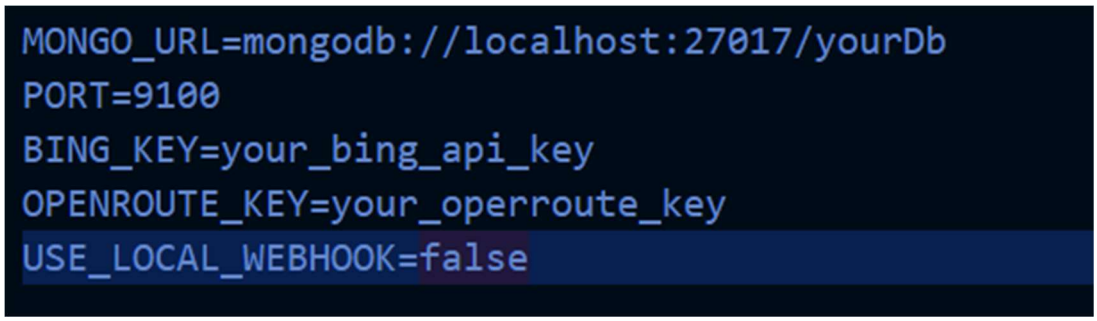


Рисунок 3.2 – Файл конфігурації проєкту з тестовими даними

- MONGO_URL - зберігає посилання на кластер бази даних MongoDB.
- PORT – поле, яке вказує, на якому порту буде запущений сервіс.
- BING_KEY – не обов'язкове поле, яке зберігає ключ API для сервісу Bing API.
- OPENROUTE_KEY - не обов'язкове поле, яке зберігає ключ API для сервісу Openroute service.
- USE_LOCAL_WEBHOOK – поле, яке визначає чи можна використовувати посилання для вебхуків на локальний адрес сервера. Створене в цілях безпеки міжмережевої взаємодії.

Файли «.eslintrc.js», «.prettierrc», «nest-cli.json» відповідають за зручність та полегшення процесу написання коду.

Файли «jest.config.js», «jest.setup.js», «tsconfig.spec.json» відповідають за конфігурацію тестування системи.

Файли «tsconfig.json», «tsconfig.build.json», «publish.sh», «Procfile», «package.json», «package-lock.json» відповідають за збірку кінцевого варіанту застосунку, а також за збірку документації проєкту.

Файли «TODO.md», «README.md», «LICENSE.md», «openapi.json» інформативні. Зберігають загальну інформацію про проєкт, а також ліцензію.

3.2 Основна директорія з програмним кодом проєкту src

На рис. 3.3 продемонстрована архітектура файлів директорії з програмним кодом проєкту «src».

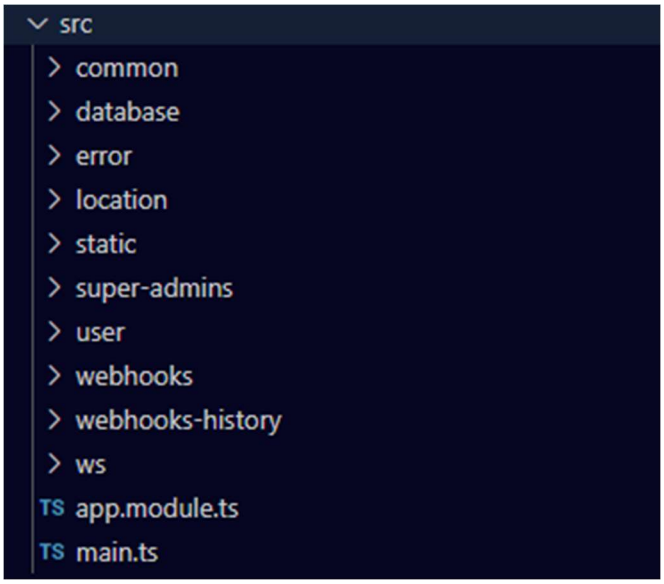


Рисунок 3.3 – Файлова архітектура директорії «src»

main.ts - основна програма сервісу, яка ініціює запуск та роботу модулів. На рис. 3.4 показано код цього файлу.

```
13  async function bootstrap() {
14    const app = await NestFactory.create(AppModule, { cors: true });
15    const port = process.env.PORT || 9100;
16    const config = new DocumentBuilder()
17      .setTitle('System for optimization of maprouting')
18      .setDescription('Documentation REST API')
19      .setVersion('1.0.0')
20      .build();
21    const document = SwaggerModule.createDocument(app, config);
22    SwaggerModule.setup('/api', app, document);
23    app.useGlobalPipes(
24      new ValidationPipe({
25        stopAtFirstError: true,
26        transform: true,
27        whitelist: true,
28      }),
29    );
30    await writeFile('./openapi.json', JSON.stringify(document, null, 2), { encoding: 'utf8'});
31    app.useGlobalFilters(new AllExceptionFilter());
32    if(process.env.docgen === 'true') await app.close();
33    else await app.listen(port, () => new Logger('MAIN').log(`Server started on port ${port}`));
34  }
35  bootstrap();
```

Рисунок 3.4 – Код файлу main.ts

Основна функція, яка запускає всю програму називається bootstrap. За допомогою вбудованих у Nest.js функції, налаштовується вебсервіс і підключається основний модуль проєкту AppModule. Відштовхуючись від змінних конфігурації визначається порт на якому буде працювати сервер. В рядках 16-30 створюється документація Swagger проєкту та генерується файл openapi. Рядок 31 підключає глобальний фільтр проєкту, який відповідає за обробку http помилок. Рядок 33 запускає програму. На рядку 35 функція bootstrap викликається.

Основний модуль програми, який підключає інші модулі, знаходиться у файлі app.module.ts. На рис. 3.5 можна переглянути підключення модулів програми. У Додатку 2 можна переглянути функціональну схему (діаграму класів) проєкту.

```
@Module({
  imports: [
    ConfigModule.forRoot({
      envFilePath: '.env',
      isGlobal: true,
    }),
    ServeStaticModule.forRoot({
      rootPath: path.join(__dirname, 'static'),
    }),
    MongooseModule.forRoot(process.env.MONGO_URL || ''),
    AuthModule,
    UserModule,
    LocationModule,
    ErrorModule,
    WsModule,
    SuperAdminsApiModule,
    WebhookModule,
    WebhookHistoryModule,
  ],
})
export class AppModule {}
```

Рисунок 3.5 – Основний модуль програми в файлі app.module.ts.

- ConfigModule – бібліотечний модуль, який відповідає за конфігурацію проєкту.
- ServeStaticModule – бібліотечний модуль, для передачі статичних даних.
- MongooseModule – бібліотечний модуль для взаємодії з базою даних MongoDB.
- AuthModule – програмний модуль, який відповідає за права доступу та ідентифікацію клієнтів.
- UserModule – програмний модуль для взаємодії з користувачами.
- LocationModule – програмний модуль для маршрутизації користувачів, підключення алгоритмів та сервісів.
- ErrorModule – програмний модуль для фіксації помилок користувачів.
- WsModule – програмний модуль для впровадження протоколу Websocket та взаємодії кінцевих користувачів в режимі реального часу.
- SuperAdminsApiModule – програмний модуль для адміністраторів системи, який генерує API ключі та налаштовує клієнтів.
- WebhookModule – програмний модуль для конфігурації та створення вебхуків.

- WebhookHistoryModule – програмний модуль для відправлення вебхуків на кінцеві сервери клієнтів, для взаємодії в режимі реального часу.

Кожен програмний модуль підключає додаткові модулі для роботи з базою даних, визначає контролери з кінцевими точками для клієнтів, а також підключає провайдер-сервіс, в якому знаходиться основна логіка модуля. На рисинку 3.6 можна побачити підключення на прикладі модуля UserModule.

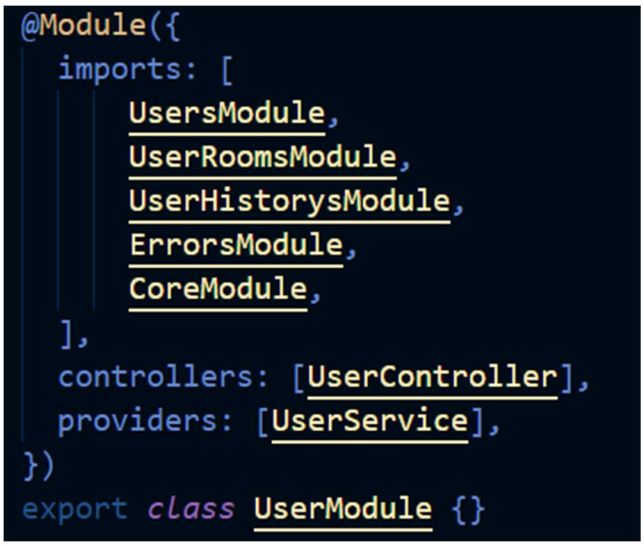


Рисунок 3.6 – Програмний модуль UserModule

Далі буде описано основну логіку програмних модулів, а саме класи провайдери-сервіси.

3.2.1 Клас SuperAdminsApiService

Клас SuperAdminsApiService створений тільки для адміністраторів системи для надання API ключів та конфігурацію клієнтів. Клас використовує допоміжні провайдери-репозиторії, для взаємодії з базою даних, такі як: SuperAdminsService, ClientsService, WebHooksService, WebHooksHistoryService, UsersService, UserRoomsService, UserHistorysService, ErrorsService, та провайдери-помічники: AuthService, CoreService. Клас має 4 метода: getClients, createNewClient, updateClient, deleteClient. Код SuperAdminsApiService можна переглянути в Додатку 4 у файлі super-admins.service.ts.

Функція getClients створена для вибірки клієнтів. Функція приймає набір стандартних інструкцій вибірки. Стандартний набір інструкцій вибірки включає: відсортовування даних зі вказанням порядку та поля для сортування, вибір по параметру поля, вибірка по даті «з/по», вказати ліміт вибірки, вказати скільки полей при вибірці слід пропустити. Результатом запиту функції є дані клієнтів, в які входять поля: ідентифікатор, ідентифікатор адміна, який створив клієнта, ідентифікатор адміна, який останній оновив клієнта, назва клієнта та ключ API клієнта.

Функція createNewClient створює нового клієнта. Якщо клієнт з такими параметрами вже існує – повертається помилка, про те, що вже такий клієнт існує. Результатом її виконання є створення клієнта й повернення інформації про нього.

Функція updateClient створена для оновлення інформації клієнта по його ідентифікатор. Якщо ідентифікатор не знайдено – повертається помилка. В результаті успіху, дані про клієнта успішно оновлюються і створюється вказівник на адміна.

Функція deleteClient створена для видалення клієнта по його ідентифікатор, а також видаляє всю інформацію пов’язану з ним в інших колекціях.

У Додатку 1 можна переглянути структурну схему (діаграма розгортання застосунку) системи, для адміністраторів, клієнтів та користувачі.

3.2.2 Клас UserService

Клас UserService створений для взаємодії з користувачами клієнтів. Клас використовує допоміжні провайдери-репозиторії по роботі з базою даних такі як: UsersService, UserRoomsService, UserHistorysService, ErrorsService, та провайдер-помічник: CoreService. Клас має 4 метода: getUsers, createNewUser,

updateUser, deleteUser. Код UserService можна переглянути в Додатку 4 у файлі user.service.ts.

Функція getUsers створена для вибірки користувачів клієнта. Функція приймає набір стандартних інструкцій вибірки. Результатом запиту функції є дані користувачів, в які входять поля: ідентифікатор, ідентифікатор вказане клієнтом, алгоритм обчислення маршрутизації, додаткові параметри вказані клієнтом.

Функція createNewUser створює нового користувача. Результатом її виконання є створення користувача й повернення інформації про нього.

Функція updateUser створена для оновлення інформації про користувача по його ідентифікатор. Якщо ідентифікатор не знайдено – повертається помилка. В результаті успіху, дані про користувача успішно оновлюються.

Функція deleteUser створена для видалення користувача по його ідентифікатор, а також видаляє всю інформацію пов’язану з ним в інших колекціях.

3.2.3 Клас ErrorService

Клас ErrorService створений для фіксації помилок користувачів. Клас використовує допоміжний провайдер-репозиторій по роботі з базою даних: ErrorsService, та провайдер-помічник: CoreService. Клас має 2 метода: getErrors, deleteError. Код ErrorService можна переглянути в Додатку 4 у файлі error.service.ts.

Функція getErrors створена для вибірки помилок користувачів. Функція приймає набір стандартних інструкцій вибірки. Результатом запиту функції є дані про помилки користувачів, в які входять поля: ідентифікатор, ідентифікатор користувача, повідомлення помилки.

Функція deleteError створена для видалення помилки користувача по ідентифікатор.

3.2.4 Клас LocationService

Клас LocationService створений для маршрутизації користувачів, підключення алгоритмів та сервісів. Клас використовує допоміжні провайдери-репозиторії для роботи з базою даних такі як: LocationsService, LocationStepsService, UserHistorysService, UsersService, та провайдери-помічники: CoreService, AlgorithmsService, LocationStrategyService. Клас має 5 методів: fixUserLocation, getUserLocation, getRoute, insertByApiAlgorithm, formByApiAlgorithms. Код LocationService можна переглянути в Додатку 4 у файлі location.service.ts.

Функція fixUserLocation фіксує місцеперебування користувача у вказаний момент часу.

Функція getUserLocation створена для вибірки місцеперебування користувача в якийсь момент часу. Функція приймає набір стандартних інструкцій вибірки. Результатом запиту функції є дані про місцеперебування, в які входять поля: ідентифікатор, ідентифікатор користувача, час, координати.

Функція getRoute це основна функція програми. Алгоритм функції такий: спочатку проводиться пошук користувача по його ідентифікатор та ідентифікатор клієнта, якщо користувач не був знайдений, тоді повертається помилка про те, що такого користувача не існує, в результаті успіху, програма знаходить вибраний користувачем алгоритм пошуку маршрут, далі, в базі даних шукаються історичні дані (з обмеженням по даті, для актуальності даних), по координатах та алгоритму пошуку, якщо такі дані було знайдено, тоді користувач моментально отримує свій результат. Перевага, того щоб зберігати історичні дані про маршрути є те, що система сама розвивається і коли одночасно будуть йти запити на маршрутизацію для одних і тих же координат, тоді відповідь від системи буде моментальною і при цьому не будуть використані допоміжні розрахунки чи звернення до сторонніх сервісів. Якщо такі дані не було знайдено, тоді здійснюється новий історичний запис, який

фіксує цей запит про маршрутизацію. Після запису викликається функція formByApiAlgorithms і в результаті користувачу повертається результат її виконання.

Функція insertByApiAlgorithm вибирає потрібний обробник у відношенні до алгоритму користувача і викликає функцію formByApiAlgorithms з потрібними налаштуваннями.

Функція formByApiAlgorithms викликає потрібний обробник з вказаними параметрами від користувача. Існує 4 обробники: floydWarshallAlgorithm, bellmanFordAlgorithm, bingStrategy, openrouteStrategy.

Обробник floydWarshallAlgorithm імплементує алгоритм Флойда–Уоршелла. Цей алгоритм знаходить найкоротший шлях між всіма парами вершин графа за O(n^3). Алгоритм використовує матрицю A розміру n*n, у якій обчислюються довжини найкоротших шляхів. A[i, j] = C[i, j] всім i != j. Якщо дуга i - j відсутня, то C [i, j] = infinity. Кожен діагональний елемент матриці A дорівнює 0. Над матрицею A виконується ітерації. Після k-ї ітерації A[i, j] містить значення найменшої довжини шляхів з вершини «i» у вершину, які не проходять через вершини з номером, більшим k. Між кінцевими вершинами шляху «i», можуть знаходитися тільки вершини, номери яких менші або рівні k. На k-й ітерації для обчислення матриці A застосовується така формула: $A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])$ [41]. Код детальніше можна переглянути в Додатку 4 у файлі floyd-warshall.service.ts.

Обробник bellmanFordAlgorithm імплементує алгоритм Беллмана-Форда. Цей алгоритм знаходить найкоротший шлях від однієї вершини до всіх інших у зваженому графі за час O(|V| × |E|). На відміну від алгоритму Дейкстри, алгоритм Беллмана-Форда припускає ребра з негативною вагою. Для заданого зваженого орієнтованого графа G = (V, E) з початком в «s» і ваговою функцією «w» : E -» R алгоритм Беллмана-Форда повертає логічне значення, що вказує на те, чи міститься у графі цикл з негативною вагою, що досягається з початку.

Якщо такий цикл існує, алгоритм вказує, що рішення не існує. Якщо ж таких циклів немає, алгоритм видає найкоротші шляхи та їхню вагу [42]. Код детальніше можна переглянути в Додатку 4 у файлі bellman-ford.service.ts.

Обробник bingStrategy імплементує алгоритм сервісу Bing API, враховує розрахунок маршрутів для пішоходів, водіїв та для переміщення громадським транспортом. Код детальніше можна переглянути в Додатку 4 у файлі bing-strategy.service.ts.

Обробник openrouteStrategy імплементує алгоритм сервісу Openroute service, враховує розрахунок маршрутів для водіїв авто, вантажівок та велотранспорту. Код детальніше можна переглянути в Додатку 4 у файлі openroute-strategy.service.ts.

Архітектура проєкту передбачає просте написання нових обробників і автоматичне їх використання в системі.

У Додатку 3 можна переглянути алгоритм роботи даної кінцевої точки.

3.2.5 Клас WebhookService

Клас WebhookService створений для конфігурації та створення вебхуків. Клас використовує допоміжні провайдери-репозиторії для роботи з базою даних такі як: WebHooksService, WebHooksHistoryService, та провайдер-помічник: CoreService. Клас має 5 методів: getWebhooks, createNewWebhook, updateWebhook, deleteWebhook, isValidUrl. Код WebhookService можна переглянути в Додатку 4 у файлі webhooks.service.ts.

Функція getWebhooks створена для вибірки вебхуків клієнта. Функція приймає набір стандартних інструкцій вибірки. Результатом запиту функції є інформація про вебхуки клієнта, в які входять поля: ідентифікатор, системна назва події, клієнтська назва події, посилання на яке відправляється вебхук.

Функція createNewWebhook створює новий вебхук клієнта. Результатом її виконання є створення вебхука й повернення інформації про нього.

Функція updateWebhook створена для оновлення інформації про вебхук по його ідентифікатор. Якщо ідентифікатор не знайдено – повертається помилка. В результаті успіху, дані про вебхук успішно оновлюються.

Функція deleteWebhook створена для видалення вебхука клієнта по його ідентифікатор.

3.2.6 Клас WebhookHistoryService

Клас WebhookHistoryService створений для відправлення вебхуків на кінцеві сервіси клієнтів, для взаємодії з серверами в режимі реального часу. Клас використовує допоміжні провайдери-репозиторії по роботі з базою даних такі як: WebHooksHistoryService, WebHooksService, ClientsService, UsersService, UserRoomsService, UserHistorysService, ErrorsService, та провайдери-помічники: CoreService, HttpService. Клас має 11 методів: getWebhooksHistory, resendWebhook, webhookHistoryWatcher, usersWatcher, webhookWatcher, usersHistoryWatcher, errorsWatcher, clientsWatcher, usersRoomsWatcher, sendWebhook, watcherHandler. Код WebhookHistoryService можна переглянути в Додатку 4 у файлі webhooks-history.service.ts.

Функція getWebhooksHistory створена для вибірки історії надсилань вебхуків до клієнта. Функція приймає набір стандартних інструкцій вибірки. Результатом запиту функції є дані історія вебхуків, в які входять поля: ідентифікатор, ідентифікатор вебхука, статус відправлення, відправлена інформація.

Функція resendWebhook створена для повторного відправлення вебхука по ідентифікатор на сервер клієнта.

Функція watcherHandler обробляє надсилання вебхуків. Вона приймає в якості параметра функцію-сервіс, назву системної події, назву вебхука і повертає іншу функцію, яка обробляє подію змін, видалення чи створення документа в базі даних. Дана функція є основою для методів

webhookHistoryWatcher, usersWatcher, webhookWatcher, usersHistoryWatcher, errorsWatcher, clientsWatcher, usersRoomsWatcher.

Функція sendWebhook створена для валідації посилань та відправлення вебхуків на сервери клієнтів.

3.2.7 Клас WsService

Клас WsService створений для використання протоколу Websocket та взаємодії кінцевих користувачів в режимі реального часу. Клас використовує допоміжні провайдери-репозиторії для роботи з базою даних такі як: UsersService, RoomsService, UserHistorysService, UserRoomsService, ErrorsService, та провайдер-помічник: AuthService. Клас має 9 метода: afterInit, handleConnection, handleDisconnect, handleErrorDevice, handleUpdateLocation, handleCreateRoom, handleJoinToRoom, handleLeaveFromRoom, userWsGuard. Код WsService можна переглянути в Додатку 4 у файлі ws.service.ts.

Функція afterInit – обробник події ініціалізації протоколу вебсокетів.

Функція handleConnection – обробник події підключення користувача до сервісу. Функція знаходить всі кімнати користувача й додає його до цих кімнат.

Функція handleDisconnect – обробник події, коли користувач закінчує підключення до сервісу. Від’єднує користувача від всіх його кімнат і сповіщає користувачів у кімнаті про те, що цей користувач від’єднався.

Функція handleErrorDevice – обробник події, коли користувач отримує помилку при роботі з сервісом.

Функція handleUpdateLocation – обробник події, коли місцеперебування користувача змінюється. Подія оповіщує всіх користувачів в кімнаті, що місцеперебування даного користувача змінилося, а також додає історичний запис про це.

Функція handleCreateRoom – обробник події, коли користувач створив кімнату.

Функція handleJoinToRoom – обробник події, коли користувач бажає приєднатися до кімнати. Додає користувача до кімнати, створюється запис у базі даних та сповіщає всіх користувачів, про те, що цей користувач приєднався до кімнати.

Функція handleLeaveFromRoom – обробник події, коли користувач від'єднується від кімнати. Від'єднує користувача від кімнати, видаляє запит з бази даних про користувача в даній кімнаті та сповіщає всіх користувачів у кімнаті про те, що цей користувач покинув кімнату.

Функція userWsGuard знаходить користувача по унікальному ідентифікатор встановлене клієнтом. Використовується у всіх інших функціях даного класу.

3.3 Обмеження в доступі та обробка помилок

В системі існує декілька ролей користувачів з обмеженнями та можливостями. На рис. 3.7 можна переглянути схему бази даних, яка демонструє залежності різних типів користувачів. Для ідентифікації користувачів, визначення їхніх ролей та прав створений провайдер-помічник AuthService. AuthService допомагає генерувати ключі API та ідентифікувати користувачів та їх ролі по цих ключах.

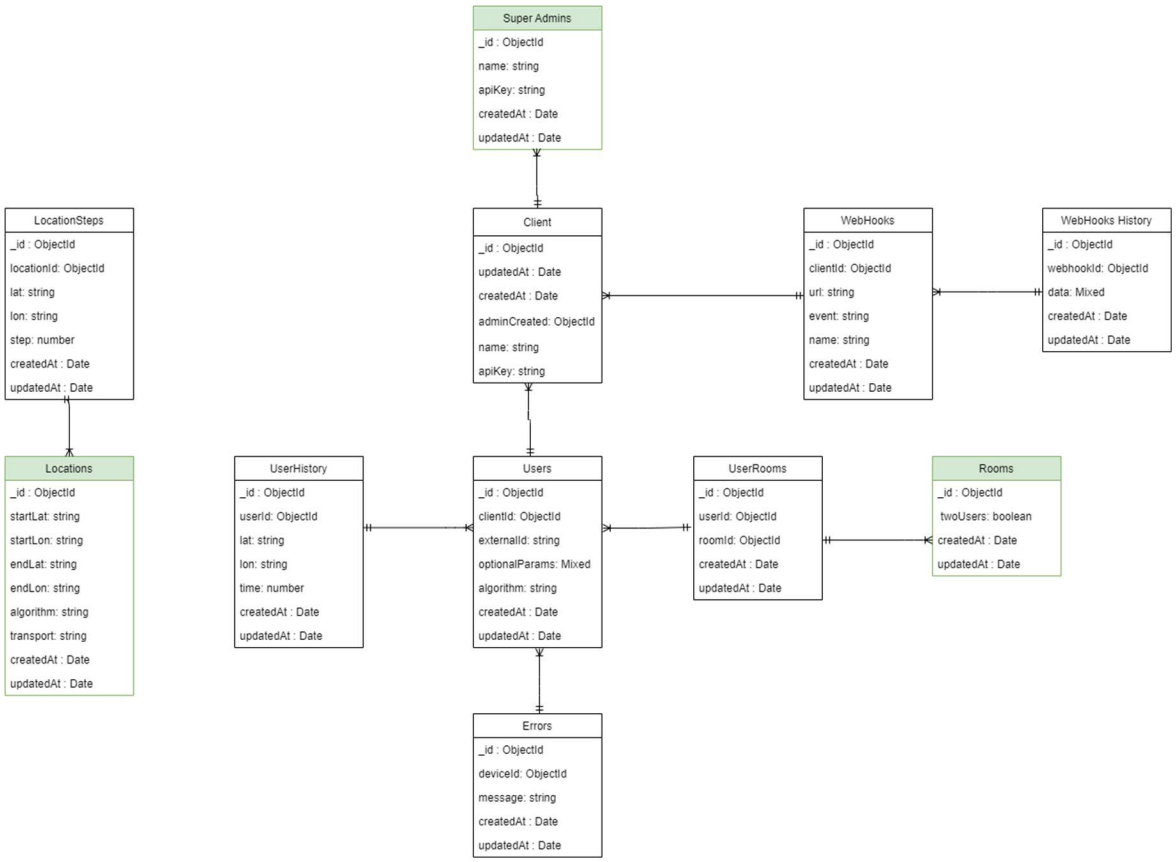


Рисунок 3.7 – Схема бази даних

Для того, щоб не викликати перевірку й взяття даних по користувачу на кожній новій створеній функції сервісу, був створений Guard [43]. Цей Guard по ключу API клієнта чи адміністратора визначає його роль та передає його дані на обробку для всіх функцій сервісу. Якщо в базі даних не було знайдено клієнта чи адміністратора, яким відповідає ключ, тоді повертається помилка користувачу, про те що в доступі відмовлено і наступна логіка функцій не виконується. Якщо клієнт успішно ідентифікувався й бажає отримати будь-яку інформацію (дані про своїх користувачів, помилки користувачів, вебхуки тощо), чи якось з нею взаємодіяти, тоді він буде мати доступ тільки до даних пов’язаних саме з ним і не буде мати доступу ні до чого іншого.

Система генерує помилки для клієнтів, які надсилають не коректні дані, чи хочуть звернутися до захищених даних. Для того, щоб дані відправлялися в фіксованому форматі й не спричиняли крах системи, було створено глобальний фільтр [44] AllExceptionFilter. Цей фільтр приводить всі помилки до

фіксованого вигляду й визначає їх HTTP статуси. Кожна система має свої внутрішні помилки та збої, для того, щоб клієнти розуміли це і система не припиняла роботу, цей фільтр обробляє такі помилки й повертає користувачу повідомлення про серверну помилку та HTTP статус 500.

3.4 Архітектура директорії з кодом-заглушками – mock

На рис. 3.8 продемонстрована архітектура файлів директорії з кодом «заглушками» - «mock».

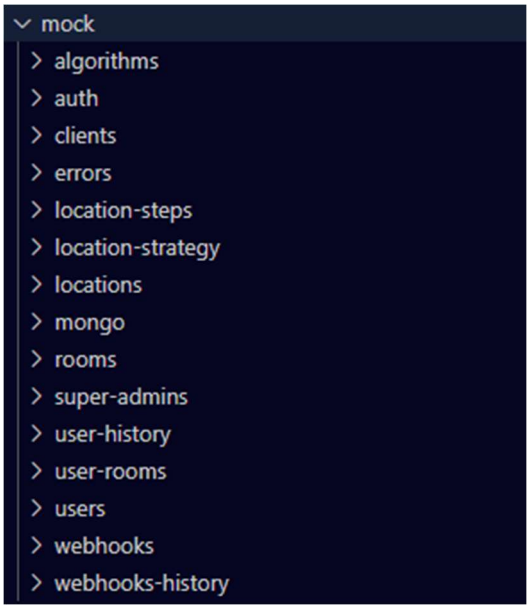


Рисунок 3.8 – Файлова архітектура директорії «mock»

Основна функція коду, який знаходиться в деректорії mock – заміна функцій пов'язаних з використанням зовнішніх сервісів, таких як база даних чи сервісів розрахунку маршруту. Ці функції використовуються в тестуванні програмного забезпечення, не взаємодіючи з зовнішніми ресурсами.

У Додатку 4, у файлі mongo.service.mock.ts можна переглянути основну заглушку бази даних MongoDB. Цей клас виділяє пам’ять під емуляцію бази даних методами самого ж TypeScript. Звернутися до такого місця в пам’яті можна за допомогою так званих «гетерів» та «сетерів», які відповідають кожній колекції в базі даних. Також, ця функція автоматично генерує статичні

унікальні Mongo ідентифікатор. Для правильної взаємодії та тестування, методом clearDb можна привести це місце в пам’яті до початкового стану.

Кожен з провайдерів-репозиторіїв по взаємодії з базою даних був переписаний на статичні методи TypeScript з використанням заглушки бази даних MongoDB. У Додатку 4, у файлі locations.service.mock.ts можна переглянути приклад такого провайдера-репозиторія.

3.5 Архітектура директорії з програмними тестами test

На рисунку 3.9 продемонстрована архітектура файлів директорії з програмними тестами «test».

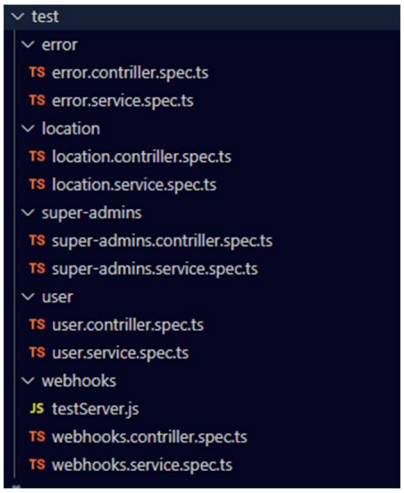


Рисунок 3.9 – Файлова архітектура директорії «test»

Коли вносяться будь-які зміни в програмний код програми, треба бути впевненим, що дані зміни виконують саме те, що було заплановано розробником. Найпростіший спосіб перевірки нових можливостей, це відкрити програму, й кожну зміну протестувати вручну. Це рутинне ручне тестування відбирає дуже багато часу й продуктивності програміста. На заміну цьому застарілому способу прийшло автоматичне тестування, яке робить всю ту ж роботу тільки займає в рази менше часу й сил.

Для написання автоматичних тестів, було використано фреймворк jest для TypeScript. Цей фреймворк надає багато можливостей, які більше ніж задовольняють потреби проєкту.

Для кожного класу було написано автоматичні тести, які покривають всі методи. Для емулювання бази даних, було використано заглушку бази даних MongoDB, і провайдери-репозиторії написані на синхронних функціях TypeScript. Приклад коду автоматичного тестування можна переглянути у Додатку 4 у файлі location.service.spec.ts. Для запуску автоматичних тестів достатньо ввести команду «npm test» в головній деректорії проєкту. Результатом проходження тестів, буде таблиця, яка показує статус відпрацювання тестів та помилки (якщо такі наявні). На рис. 3.10 показано результат проходження тестів, які показують, що програма на 100% правильно працює.

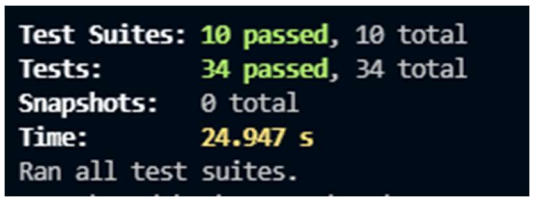


Рисунок 3.10 – Результат відпрацювання тестів

Виключенням при тестуванні системи являються вебхуки. Вебхуки створені для взаємодії між різними серверами. Одним сервером який віддає інформацію є сервер даної системи, а інший сервер – клієнтський сервер. Для емуляції клієнтського серверу було написано найпростіший http сервер, який виводить в консоль інформацію про дані які приходять. Приклад сервера можна переглянути у Додатку 4 у файлі testServer.js.

3.6 Архітектура директорії з документацією проєкту – docs

На рисунку 3.11 продемонстрована архітектура файлів директорії з документацією проєкту - «docs».

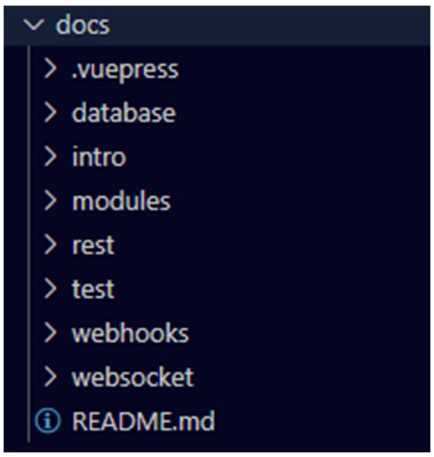


Рисунок 3.11 – Файлова архітектура директорії «docs»

Для створення швидкої та красивої документації коду, була створена директорія docs. На основі даної директорії бібліотека vuepress перетворює текст з формату markdown в красивий інтерфейс.

На рис. 3.12 можна переглянути як звичайний текст перетворився у красиву вебсторінку з документацією.

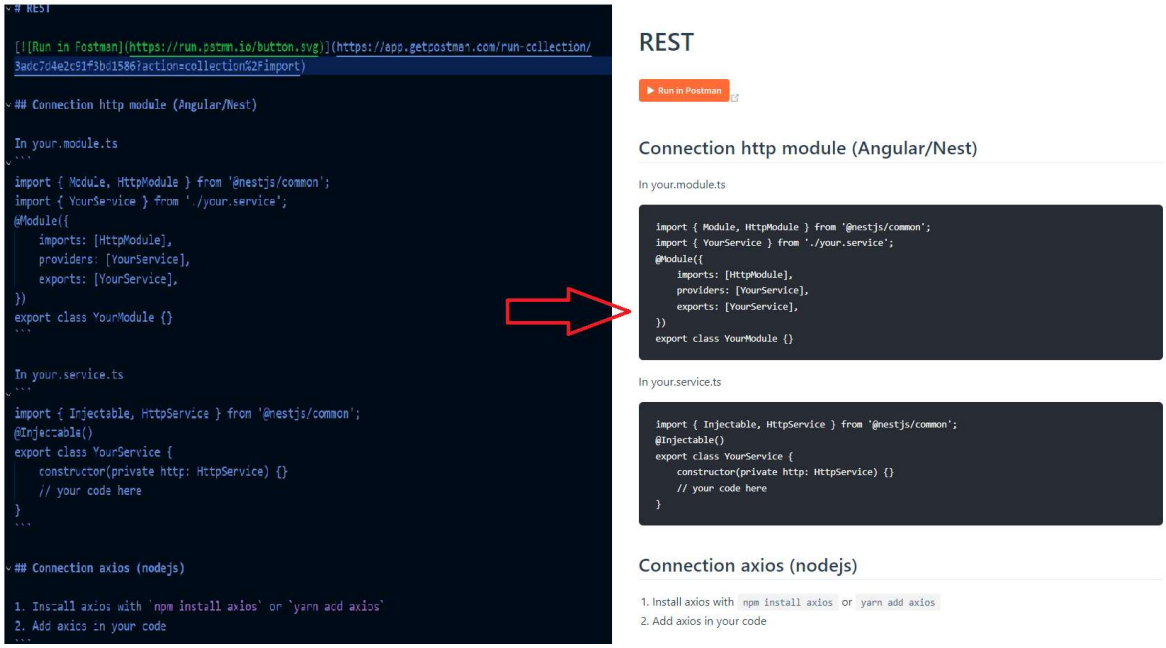


Рисунок 3.12 – Результат генерування вебсторінки з документацією

Директорія .vuepress створена для налаштування відображення та збереження статичних даних, таких як картинки та іконки. Інші директорії відповідають за кожну окрему сторінку документації, в кожній такій документації існує README.md файл, який містить контент сторінки.

3.7 Автоматичне документування проєкту

Документування створеної системи це дуже важливий етап розробки. Розробляти документацію можна багатьма способами. Якщо створювати документацію вручну з самого початку, без ніяких додаткових інструментів, тоді це забирає дуже багато ефективності й часу розробки, а також таку документацію дуже важко підтримувати. Сьогодні існує безліч інструментів, які автоматично генерують документацію проєкту відштовхуючись від вже написаного коду. В даному проєкті було використано технології Swagger та Vuepress в поєднанні з jsdoc2md.

3.7.1 Swagger для автоматичної генерації документації API

Специфікація OpenAPI, яка раніше називалася Swagger Specification — це формат опису REST API.

Бібліотека @nestjs/swagger що дає можливість автоматично генерувати файл зі специфікацією OpenApi та створювати вебінтерфейс (рис. 3.13).

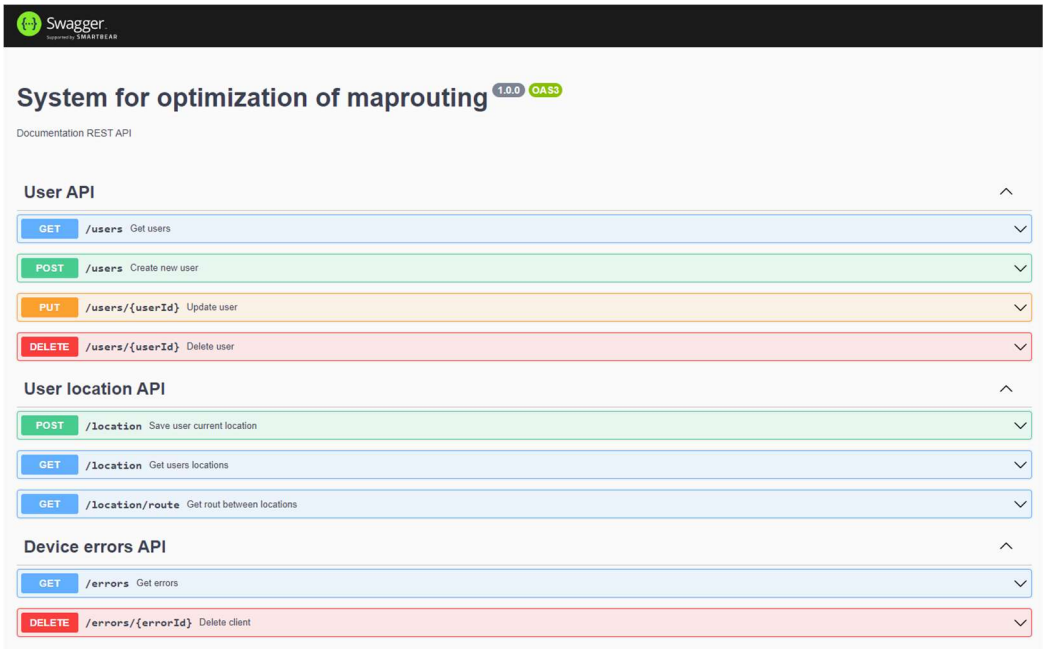


Рисунок 3.13 – Вебсторінка swagger

Файл OpenAPI що дає можливість описати API, а саме:

- Кінцеві точки (endpoints) системи, їх назви, шлях, та особливості.
- Вхідні та вихідні дані кожної кінцевої точки.
- Методи аутентифікації.
- Різного роду специфікацію проєкту, наприклад, ліцензію, мову програмування, умови використання тощо.

Для генерації розділу кінцевих точок, в контролері достатньо вибраний клас обгорнути декоратором @ApiTags, в який можна вказати шлях. Окремо кожну кінцеву точку можна налаштувати великою кількістю декораторів та їх налаштувань. Декоратори для кінцевих точок, які найбільше використовуються, це @ApiOperation для опису, @ApiResponse, для показу відповіді, @ApiHeader для опису заголовків запиту (в тому числі й заголовків авторизації). Приклад контролера з декораторами для автоматичної генерації документації можна переглянути в Додатку 4 у файлі location.controller.ts.

3.7.2 Vuepress для швидкого створення документації програмного продукту

Для створення вебсайту з документацією проєкту було використано бібліотеку Vuepress, яка генерує з файлів у форматі markdown повноцінний сайт. Для автоматичного опису та генерації файлу з модулями й класами проєкту було використано бібліотеку jsdoc2md, яка зі специфікації опису коду jsdoc генерує markdown, що своєю чергою перетворюється в вебсторінку з документацією. Приклад опису специфікації jsdoc можна переглянути у Додатку 4.

На рис. 3.14 можна побачити приклад сторінки вебсайту згенерованого за допомогою Vuepress.

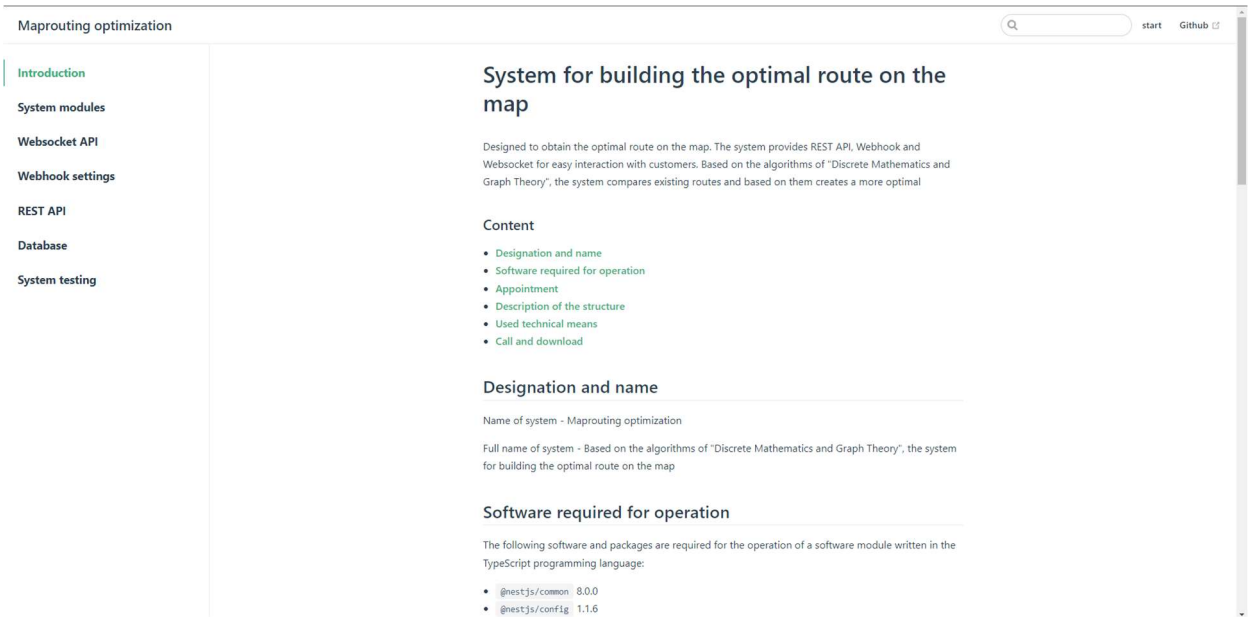


Рисунок 3.14 – Сторінка документації згенерована за допомогою Vuepress

Перевага Vuepress в тому, що кожна сторінка, створена Vuepress, має власний попередньо відтворений статичний HTML, що забезпечує високу продуктивність завантаження. Проте, як тільки сторінка завантажується, Vue бере на себе статичний вміст і перетворює його на повноцінну односторінкову програму (SPA). Додаткові сторінки завантажуються на вимогу, коли користувач переміщається по сайту.

ВИСНОВОК ДО РОЗДІЛУ 3

В розділі описано процес розробки системи. Було детально розглянуто архітектуру проєкту, кожен основний клас, та їх функції, алгоритм роботи з помилками та доступом. А також, було детально описано процес автоматичного тестування та документування системи.

Спочатку було описано структуру основної директорії проєкту. Детально розказано про призначення кожного файлу й директорії. Було розказано про файли, які відповідають за конфігурації проєкту і розписано кожне поле конфігурації, про файли, які полегшують процес написання коду, файли, які відповідають за конфігурацію тестування, файли які відповідають за збірку проєкту та документації.

Далі було описано процес та необхідність розробки класів-заглушок для автоматичного тестування. Після цього розписано важливість автоматичного тестування системи та показано процес покриття тестами проєкту.

В кінці було продемонстровано документацію проєкту у двох екземплярах – на основі swagger та vuepress. Також було детально описано процес створення та автоматичної генерації документації.

В результаті було створено програму, яка реалізовує всі заплановані можливості, покрито програму автоматичними тестами та створена документація до проєкту.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

Базуючись на розробленій системі геолокації та маршрутизації, в даному розділі буде показано результати її роботи. Також проаналізовано систему, та порівняно її переваги та недоліки над іншими системами.

4.1 Тестування модулів програми

Для тестування модулів системи було використано безплатну систему для тестування й перевірки API – Postman. Postman підтримує специфікацію OpenApi, яку генерує сама програма, тому для створення звернень достатньо всього лиш імпортувати згенеровану специфікацію до Postman. В документації до проєкту на сторінці REST API є кнопка під назвою «Run in Postman», натиснувши на яку можна відкрити специфікацію зразу ж у застосунку Postman. Але звертатися до API можна через будь-який зручний спосіб, та через будь-яку мову програмування, яка це підтримує.

4.1.1 Тестування Super admins модуля

Програмний модуль Super admins має 4 кінцеві точки, а саме, для взяття, створення, зміни та видалення клієнтів. Для взаємодії по API треба вказати ключ API адміністратора. При першому розгортанні проєкту створюється адміністратор з ключем fd9cb975-a09f-436f-8c00-3811b8f4197e (який слід зразу ж змінити), тож в цілях тестування буде використовуватися цей ключ.

На рис. 4.1 продемонстровано результат виклику кінцевої точки для створення клієнта. В результаті виклику, було створено нового клієнта та повернуто інформацію про нього.

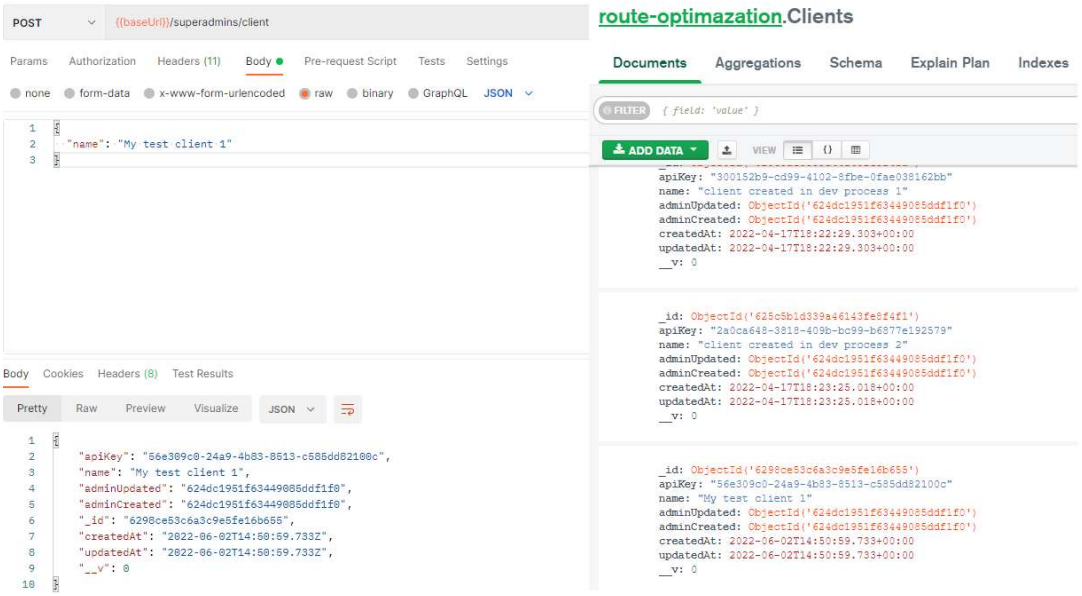


Рисунок 4.1 – Результат відпрацювання кінцевої точки для створення клієнта

На рис. 4.2 продемонстровано результат виклику кінцевої точки для взяття клієнтів. В результаті виклику, було повернуто користувача створеного на попередньому етапі. Кінцева точка підтримує фільтрацію даних по різних параметрах, а також що дає можливість сортувати вибірку даних.

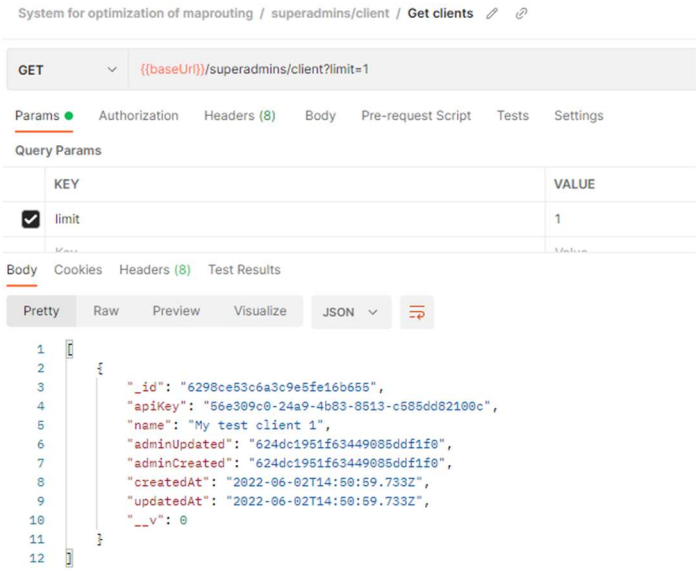


Рисунок 4.2 – Результат відпрацювання кінцевої точки для взяття клієнтів

На рис. 4.3 продемонстровано результат виклику кінцевої точки для зміни даних клієнта. В результаті виклику, було оновлено інформацію про клієнта в базі даних та повернуто інформацію зі зміненими даними.

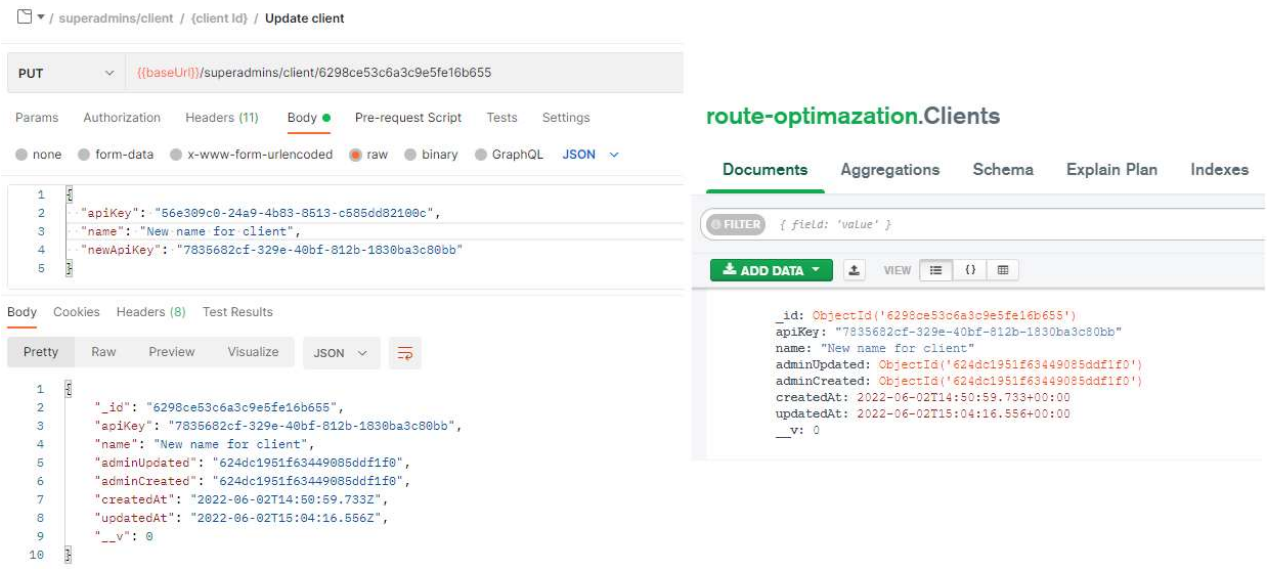


Рисунок 4.3 – Результат відпрацювання кінцевої точки для зміни клієнта

На рис. 4.4 продемонстровано результат виклику кінцевої точки для видалення клієнта.

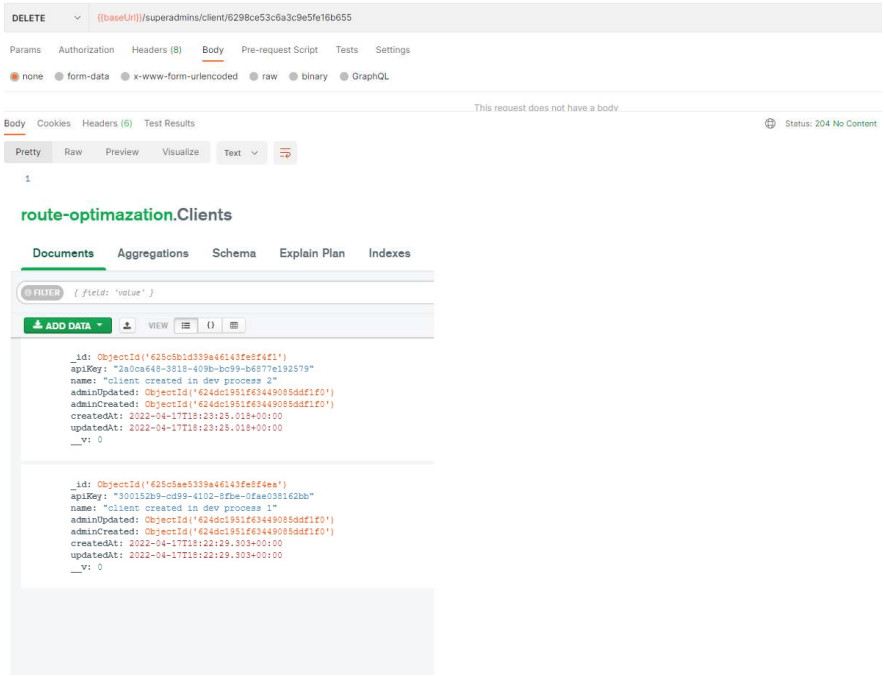


Рисунок 4.4 – Результат відпрацювання кінцевої точки видалення клієнта

В результаті виклику, було видалено всю інформацію про клієнта в базі даних, що включає інформацію про самого клієнта, всі вебхуки клієнта та історію надсилання вебхуків, всіх користувачів, їх кімнати, місцеперебування та помилки. В результаті успішного видалення буде повернуто порожнє тіло зі статусом 204.

4.1.2 Тестування Users модуля

Для тестування функцій клієнта, було створеного нового клієнта (як в минулому прикладі). Тестовий клієнт отримав ключ API d1682dc4-9c26-4283-840a-290a19244b2a, який буде далі використовуватися для ідентифікації клієнта.

На рис. 4.5 продемонстровано результат виклику кінцевої точки для створення користувача. В результаті виклику, було створено нового користувача для даного клієнта та повернуто інформацію про нього.

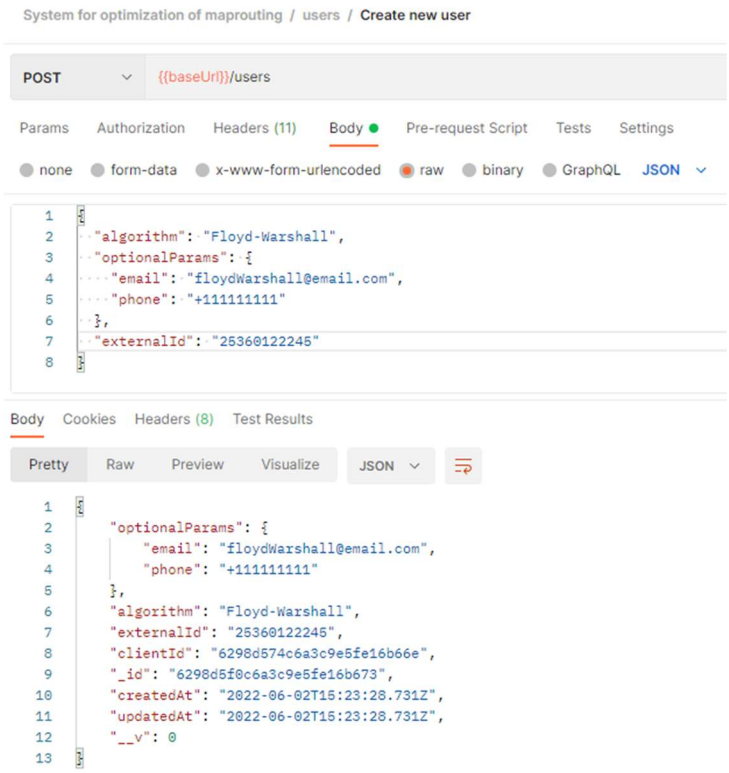


Рисунок 4.5 – Результат відпрацювання кінцевої точки створення користувача

На рис. 4.6 продемонстровано результат виклику кінцевої точки для взяття користувачів. В результаті виклику, було повернуто користувача створеного на попередньому етапі. Кінцева точка підтримує фільтрацію даних по різних параметрах, а також що дає можливість сортувати вибірку даних.

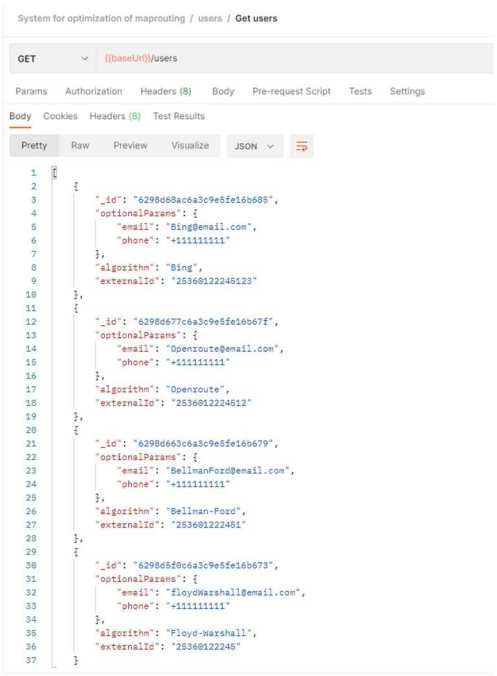


Рисунок 4.6 – Результат відпрацювання кінцевої точки взяття користувача

На рис. 4.7 продемонстровано результат виклику кінцевої точки для зміни даних клієнта. В результаті виклику, було оновлено інформацію про користувача в базі даних та повернуто інформацію зі зміненими даними.

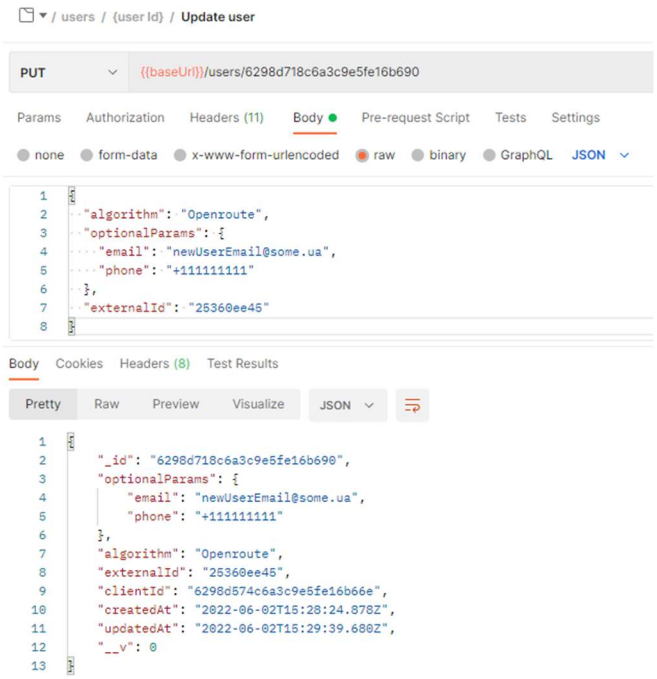


Рисунок 4.7 – Результат відпрацювання кінцевої точки зміни користувача

На рис. 4.8 продемонстровано результат виклику кінцевої точки для видалення користувача.

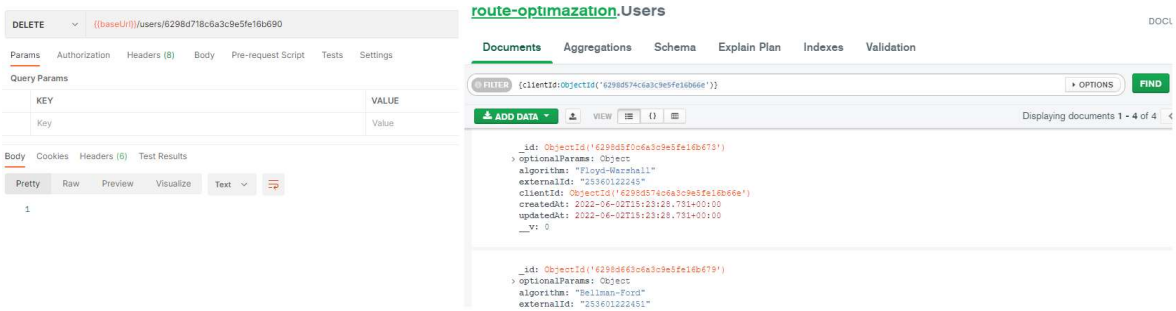


Рисунок 4.8 – Результат відпрацювання кінцевої точки видалення користувача

В результаті виклику, було видалено всю інформацію про користувача, його кімнати, місцеперебування та помилки. В результаті успішного видалення буде повернуто порожнє тіло зі статусом 204.

4.1.3 Тестування Errors модуля

На рис. 4.9 продемонстровано результат виклику кінцевої точки для взяття помилок користувачів. В результаті виклику, було повернуто помилки всіх користувачів для даного клієнта. Кінцева точка підтримує фільтрацію даних по різних параметрах, а також що дає можливість сортувати вибірку даних.

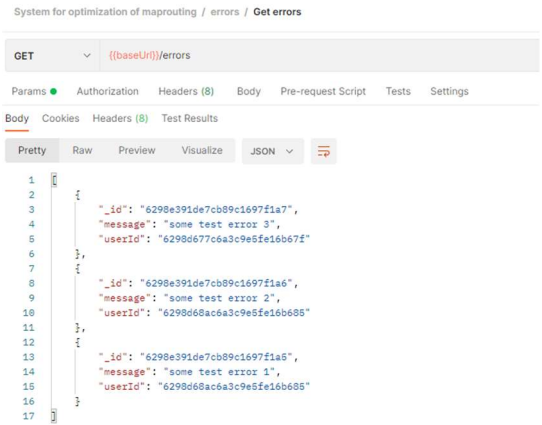


Рисунок 4.9 – Результат відпрацювання кінцевої точки взяття помилок користувачів

На рис. 4.10 продемонстровано результат виклику кінцевої точки для видалення помилки користувача.

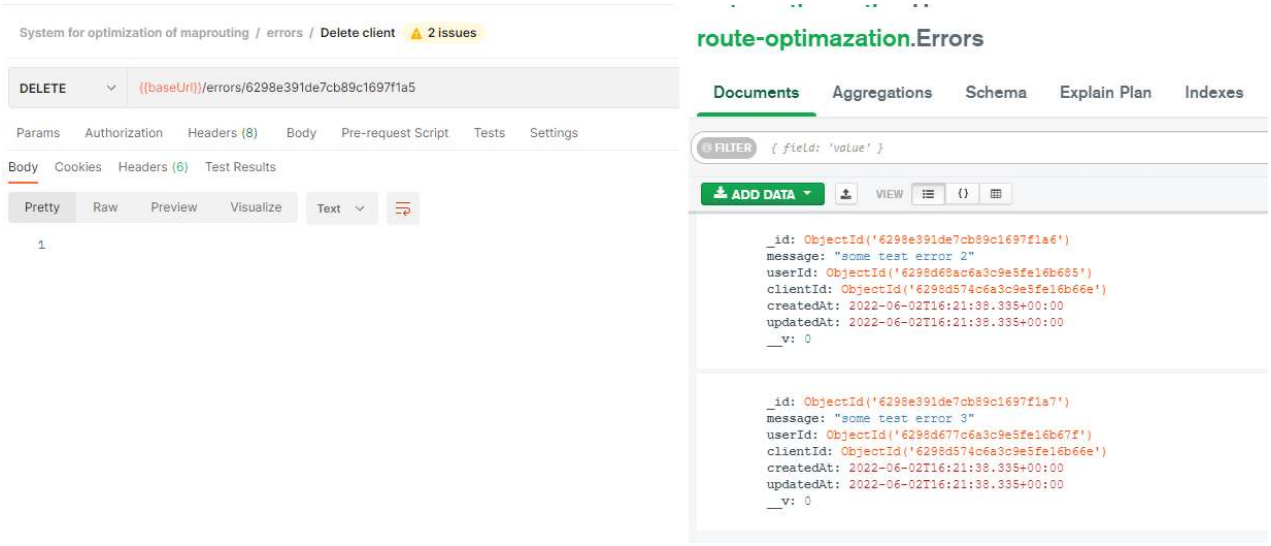


Рисунок 4.10 – Результат відпрацювання кінцевої точки видалення помилки користувача

В результаті виклику, було видалено помилку користувача. В результаті успішного видалення буде повернуто порожнє тіло зі статусом 204. Дана кінцева точка була створена з метою підтримки видалення помилок клієнтом, тож є можливість видаляти їх самим користувачем, оскільки помилки може створювати тільки сам користувач.

4.1.4 Тестування Locations модуля

На рис. 4.11 продемонстровано результат виклику кінцевої точки для фіксації місцеперебування користувача. В результаті виклику, було зафіксовано місцеперебування даного користувача та повернуто інформацію про це. Дана інформація може бути використана в цілях збереження історії переміщення користувача. Функція що дає можливість застосункам для кінцевих користувачів показувати місцеперебування користувача й для інших користувачів, які використовують систему, в режимі реального часу.

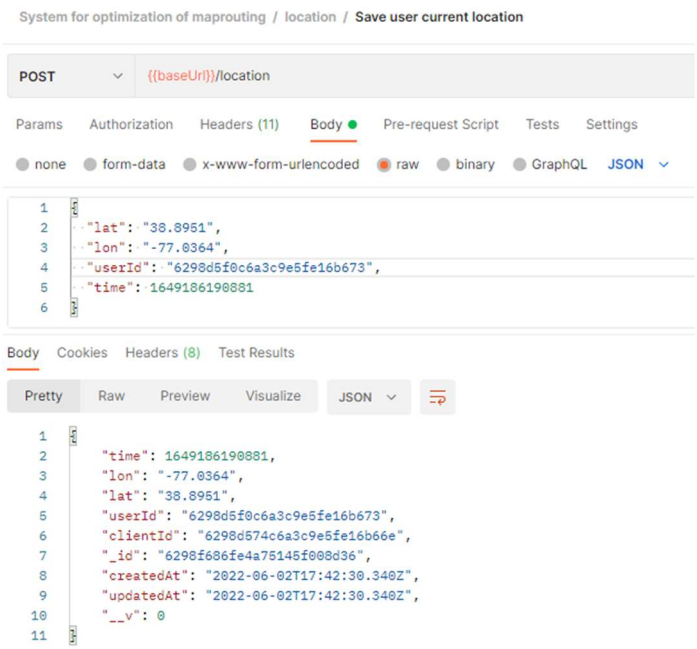


Рисунок 4.11 – Результат відпрацювання кінцевої точки для збереження координат користувача в момент часу

На рис. 4.12 продемонстровано результат виклику кінцевої точки для взяття місцеперебування користувачів у деякий момент часу. В результаті виклику, було повернуто місцеперебування всіх користувачів для даного клієнта. Кінцева точка підтримує фільтрацію даних по різних параметрах, а також що дає можливість сортувати вибірку даних.

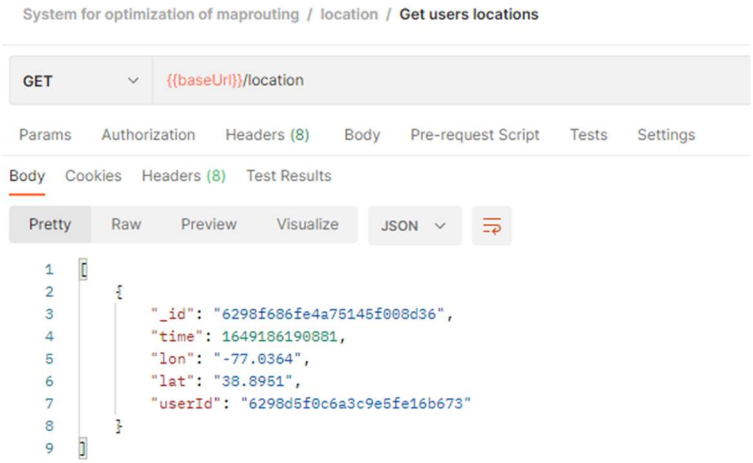


Рисунок 4.12 – Результат відпрацювання кінцевої точки для взяття координат користувача в момент часу

На рис. 4.13-4.16 продемонстровано результат виклику кінцевої точки для маршрутизації шляху з пункту А в пункт Б.

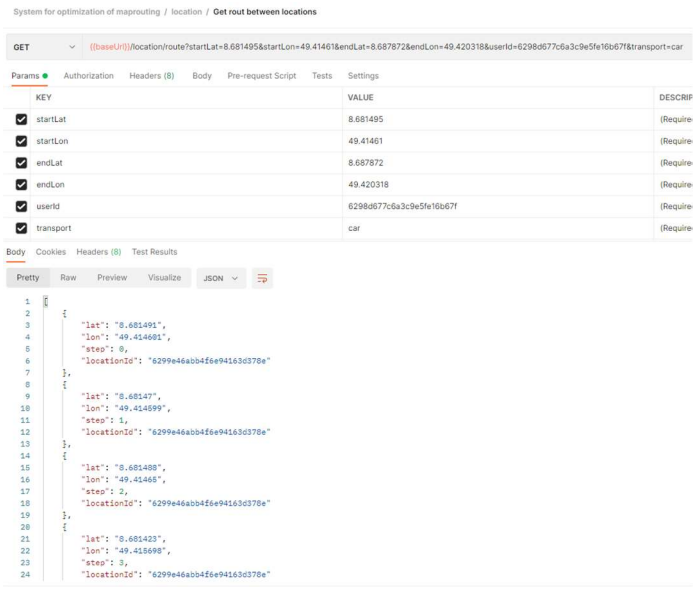


Рисунок 4.13 – Результат оптимізації та маршрутизації за допомогою Openroute

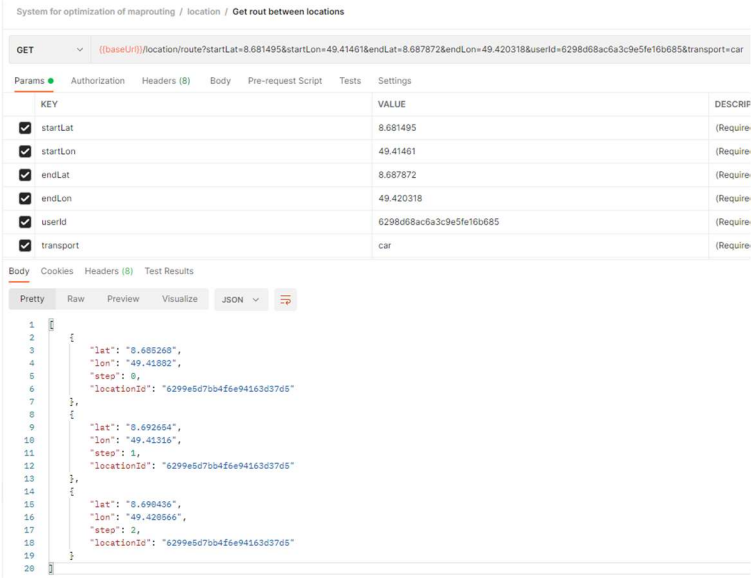


Рисунок 4.14 – Результат оптимізації та маршрутизації за допомогою Bing

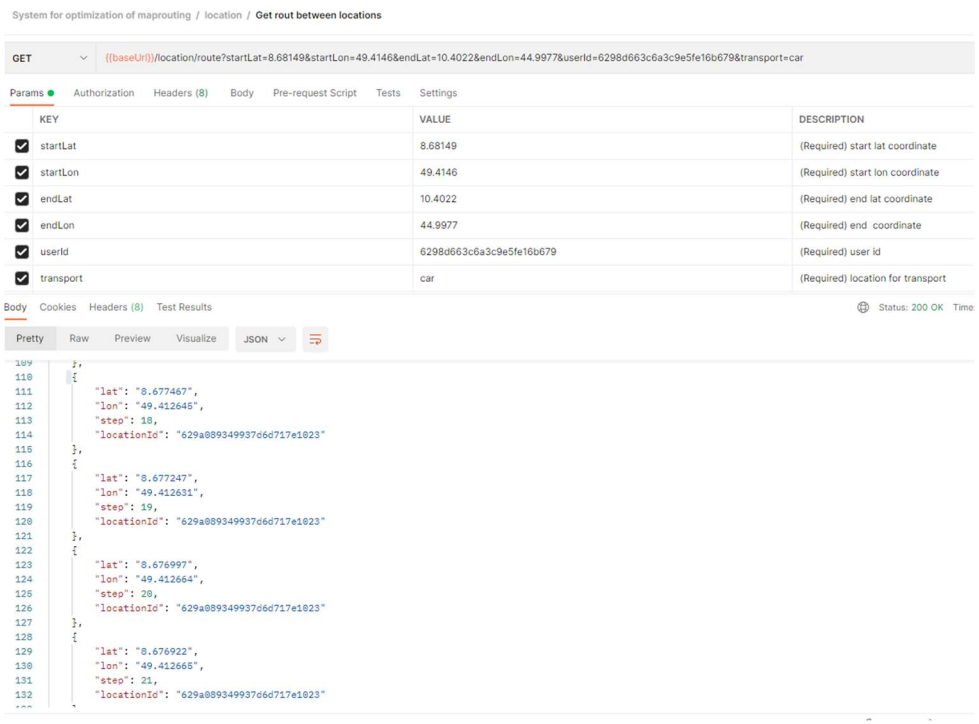


Рисунок 4.15 – Результат оптимізації та маршрутизації за допомогою алгоритму Беллмана-Форда

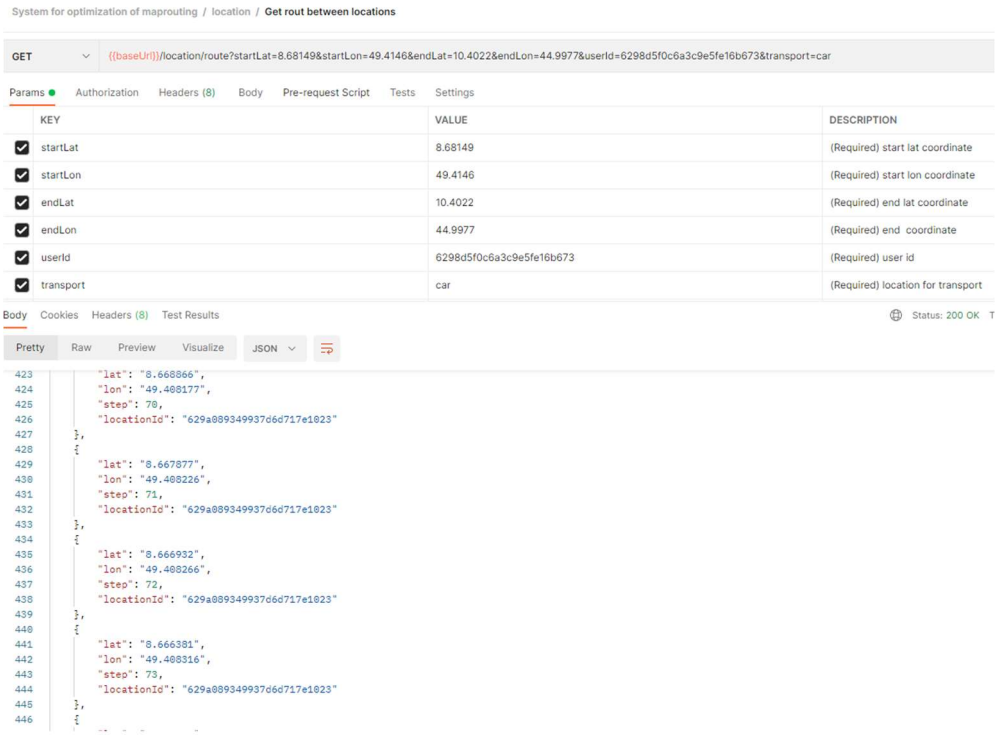


Рисунок 4.16 – Результат оптимізації та маршрутизації за допомогою алгоритму та Флойда–Уоршелла

В даних прикладах було вказано користувачів, які використовують алгоритми оптимізації Беллмана-Форда, Флойда–Уоршелла та сервісів Openroute, Bing. В результаті виклику, було повернуто оптимізований маршрут з покроковою інструкцією. Кінцева точка підтримує розрахунок для різних типів транспортних засобів.

4.1.5 Тестування Webhooks модуля

На рис. 4.17 продемонстровано результат виклику кінцевої точки для реєстрації нового вебхука. В результаті виклику, було зареєстровано новий вебхук та повернуто інформацію про нього.

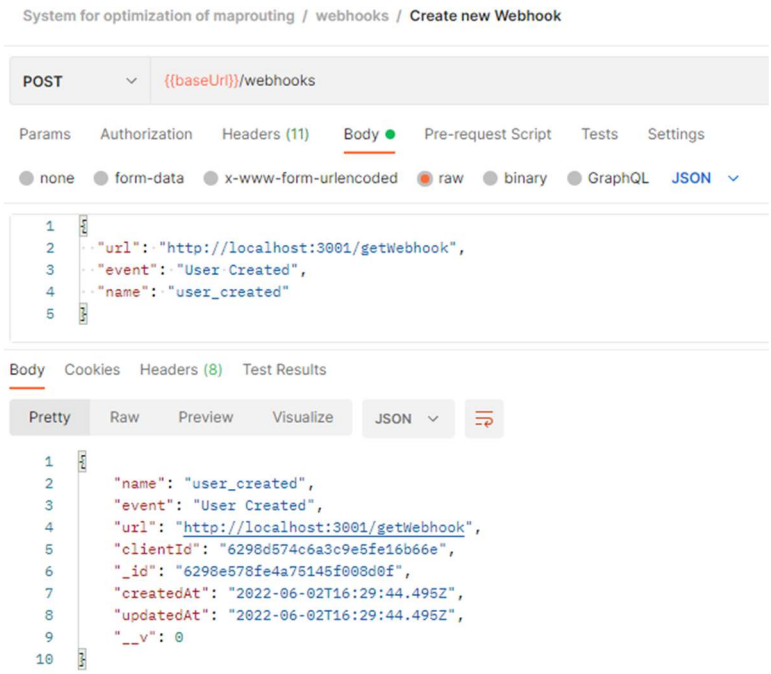


Рисунок 4.17 – Результат відпрацювання кінцевої точки для реєстрації вебхука

На рис. 4.18 продемонстровано результат виклику кінцевої точки для взяття вебхуків. В результаті виклику, було повернуто вебхук створений на попередньому етапі. Кінцева точка підтримує фільтрацію даних по різних параметрах, а також що дає можливість сортувати вибірку даних.

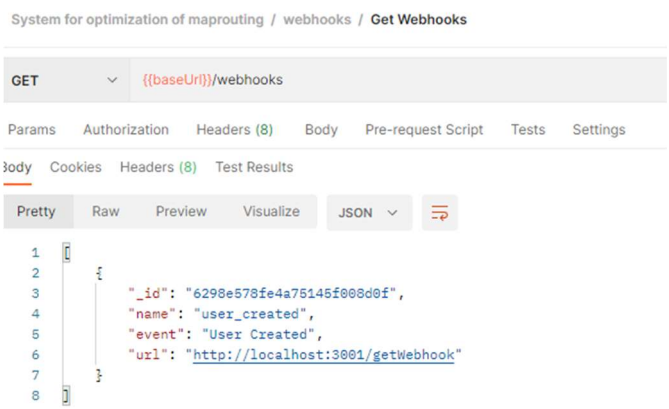


Рисунок 4.18 – Результат відпрацювання кінцевої точки для взяття вебхуків

На рис. 4.19 продемонстровано результат виклику кінцевої точки для зміни даних про вебхук. В результаті виклику, було оновлено інформацію про вебхук в базі даних та повернуто інформацію зі зміненими даними.

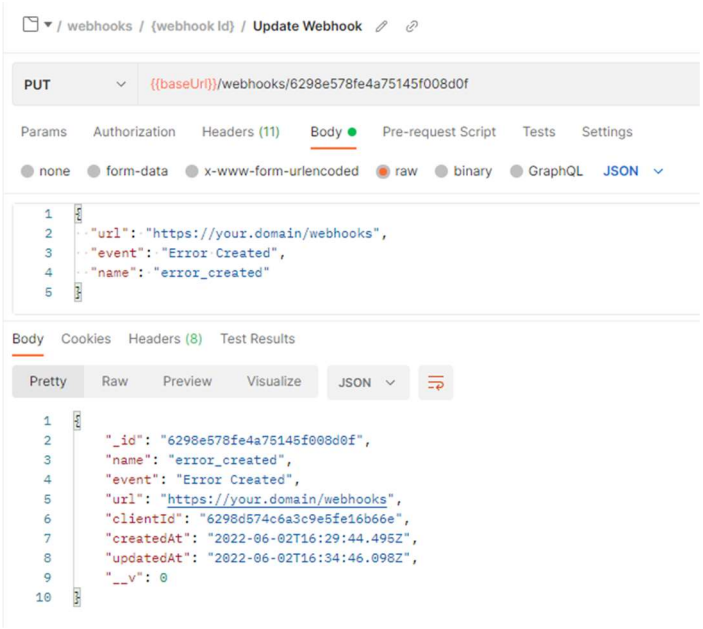


Рисунок 4.19 – Результат відпрацювання кінцевої точки для оновлення вебхука

На рис. 4.20 продемонстровано результат виклику кінцевої точки для видалення вебхука клієнта.

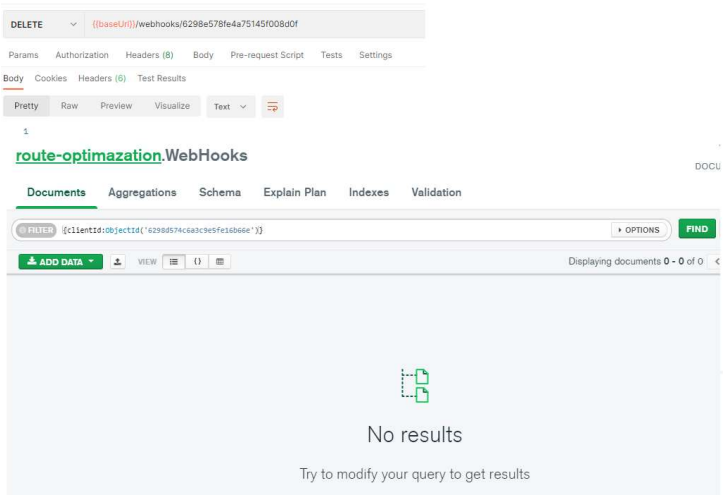


Рисунок 4.20 – Результат відпрацювання кінцевої точки для видалення вебхука

В результаті виклику, було видалено вебхук. В результаті успішного видалення буде повернуто порожнє тіло зі статусом 204.

4.2 Тестування міжмережевої взаємодії

Система використовує технологію вебхуків для взаємодії декількох систем в режимі реального часу. Тож для тестування міжмережевої передачі даних між різними системами, було створено тестовий сервер клієнта, який приймає дані від системи описаної в дипломному проєкті й виводить ці дані в термінал (найпростіше, наочне рішення для тестування міжмережевої взаємодії). Код сервера клієнта можна переглянути в Додатку 4 у файлі testServer.js.

На рис. 4.21 було успішно зареєстровано вебхук на створення користувача. Тож, коли буде створено нового користувача для даного клієнта, буде надісланий вебхук з даними про користувача та інформацією з метаданими. Після надсилання вебхука, історія про це буде записана у базі даних, тому клієнт в будь-який час може переглянути статус надсилання та за необхідності повторно надіслати цю інформацію.

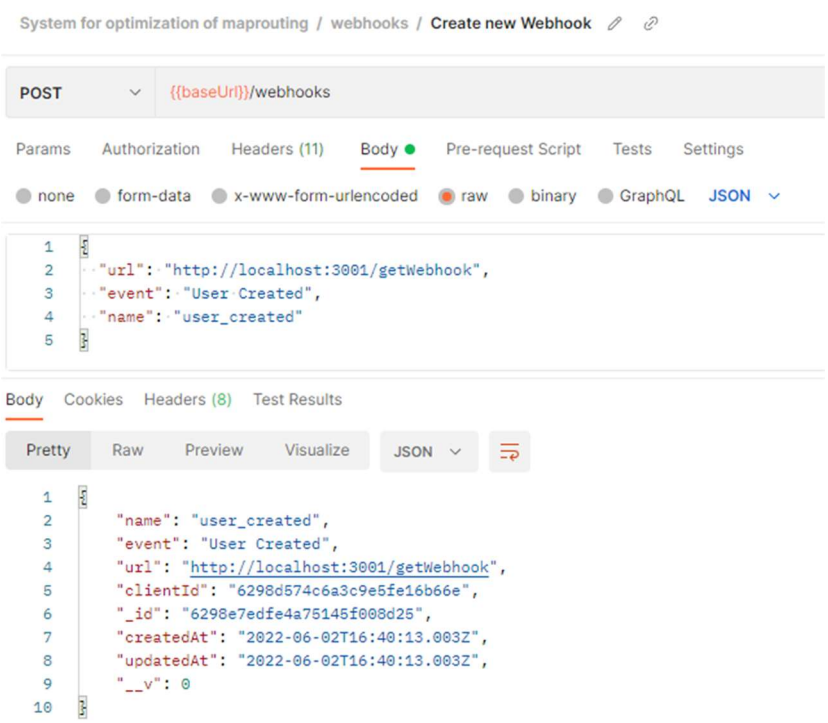


Рисунок 4.21 – Результат реєстрації вебхука який реагує на подію створення користувача

На рис. 4.22 продемонстровано результат створення нового користувача для даного клієнта.

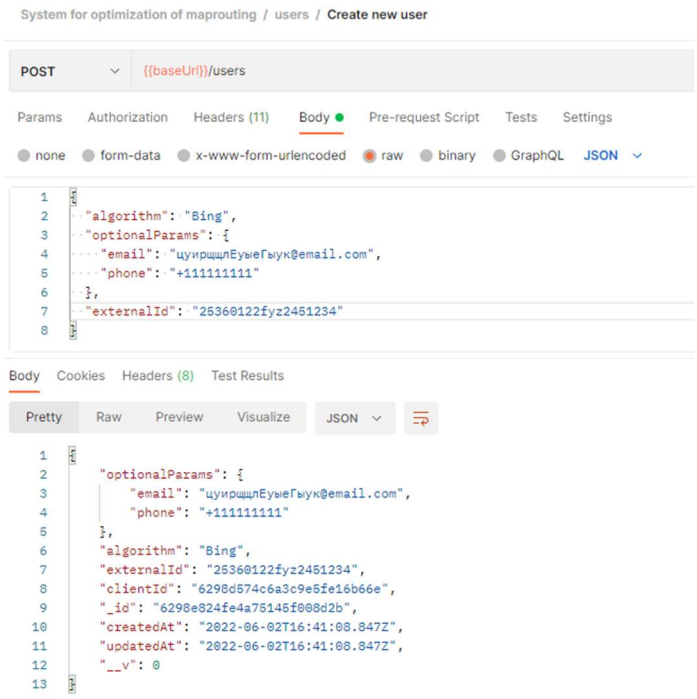


Рисунок 4.22 – Результат створення користувача з зареєстрованим вебхуком

На рис. 4.23 показано результат приймання вебхука на стороні сервера клієнта.

Рисунок 4.23 – Приклад прийняття даних сервером-клієнтом

Як видно, після створення нового користувача в системі, було викликано вебхук, який реагує на цю події, та дана інформація надіслана на сервер клієнта.

4.3 Аналіз розробленої системи

В результаті було створено систему геолокації та маршрутизації. Система надає велику кількість можливостей як для кінцевих користувачів, так і для розробників інших систем чи застосунків. Було створено API для адміністраторів та клієнтів. Систему легко адмініструвати за допомогою зрозумілого API. Клієнти мають велику кількість можливостей, які задовольняють потреби користувачів. Клієнти можуть використовувати REST API для взаємодії з базою даних, websocket API який надає інтерфейс для кінцевих користувачів, який працює в режимі реального часу, також можуть використовувати вебхуки, які створені для взаємодії між різними серверами в режимі реального часу.

Система призначена як для потреб кінцевих користувачів, так і для потреб бізнесу. Кінцеві користувачі можуть зберігати свою геолокацію, для подальшого аналізу та взаємодії, можуть розрахувати та оптимізувати власний маршрут від пункту А до пункту Б. Система покриває велику кількість бізнес

речень, тому що має можливість визначити та оптимізувати маршрут для різних видів транспортних засобів. Основна можливість системи – підключення різного роду сервісів маршрутизації та оптимізації, тож можна знайти окремий сервіс, який покриває специфічні потреби бізнесу та імплементувати його в розроблену систему дуже просто. У розділі 1 було описано багато систем зі специфічними можливостями, кожна з описаних систем може використовуватися у розробленому проєкті.

Аналізуючи розроблену систему в порівнянні з іншими, можна відокремити такі особливості:

Простий, функціональний та зрозумілий інтерфейс програми. За допомогою документації у Swagger зі специфікацією OpenAPI легко зрозуміти інтерфейс програми й самостійно його перевірити.

• Розроблена система має можливість взаємодіяти з кінцевими користувачами в режимі реального часу, з всіх розглянутих систем, таку можливість має тільки Google, але при підключених декількох їхніх сервісів.

• Розроблена система має можливість взаємодіяти з серверами клієнтів у режимі реального часу за допомогою технології вебхуків. З всіх розглянутих систем немає ні одної, яка б підтримувала дану можливість.

• В порівнянні з більшістю розглянутих систем, в даному проєкті використовуються ті ж алгоритми оптимізації, такі як Флойда–Уоршелла та Беллмана-Форда, так як ці алгоритми себе зарекомендували як найефективніші для даної задачі.

• Код системи розробленої в дипломному проєкті знаходиться в відкритому доступі на сторінці github [45], тож кожен охочий може її вдосконалювати й використовувати для своїх потреб, чого не можна сказати про більшість розглянутих систем.

• Через збереження історичних даних, які були оптимізовані для потреб одного користувача, ці дані можуть використовуватися іншими користувачами, що значно пришвидшує роботу системи. Деякі системи також використовують

такий підхід для швидкодії, але таким як, наприклад, HERE API цього не вистачає.

• Більшість розглянутих систем це комерційні, платні проєкти, які мають працівників для підтримки й постійного покращення системи. Цей проєкт не має таких можливостей, на противагу цьому його може вдосконалювати спільнота програмістів чи прихильників, тому що система знаходиться у відкритому доступі.

• Дана система не надає інтерфейс карт. Більшість розглянутих систем мали власний інтерфейс, для легкого поєднання з API. Карти не є ціллю даної роботи, тому вони не були розроблені, але створене API підтримує будь-які наявні карти.

• Дана система немає специфічних функцій для цілей окремого користувача чи бізнесу. На противагу цьому, система підтримує взаємодію з будь-якою іншою системою. Тож якщо буде потреба, наприклад, знайти найближчу карету швидкої допомоги - можна приєднати ArcGIS Direction API та розв’язати дану задачу методами зовнішнього API.

Проаналізувавши розроблену систему в дипломному проєкті, можна зробити висновок, що всі заплановані функції було реалізовано, а саме:

• Простий, функціональний та зрозумілий інтерфейс програми. За допомогою документації у Swagger зі специфікацією OpenAPI легко зрозуміти інтерфейс програми й самостійно його перевірити.

• Можливість легкого впровадження інших алгоритмів та сервісів. В результаті розробки було створено допоміжний програмний модуль, в який можна додати новий алгоритм чи сервіс і система автоматично зможе використовувати нові можливості.

• Система повинна бути автономна і не залежати від встановленого програмного забезпечення, виключаючи платформу (операційну систему). Система написана на Node.js, тому розроблена система не залежить від операційної системи.

• Зрозуміла технічна документація системи. Документація системи була розроблена на Vuepress та Swagger. Всі програмні модулі та кінцеві точки описані в документації.

• Основний код покрито тестами. Всі кінцеві точки програми покрито тестами.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі було продемонстровано вже готову систему геолокації та маршрутизації. Було показано використання кінцевих точок наявних в проєкті та надані додаткові відомості та вказівки щодо використання. Було розказано про метод ідентифікації користувача системи, показано результати оптимізації маршруту за різними алгоритмами. В кінці було проведено аналіз розробленої системи та вказано її основні переваги та недоліки, порівнюючи з іншими системами.

З початку було розказано про метод тестування системи, та наявні способи для швидкого початку впровадження та ручного тестування системи. В даному розділі було продемонстровано роботу кожного модуля системи. Розказано про особливості використання кінцевих точок, такі як використання ключа API та різні методи для вибірки, створення, зміни та видалення інформації.

Далі було детально показано процес міжмережевої взаємодії серверів у режимі реального часу.

В кінці було проаналізовано результат виконаної роботи. Було вказано на основні переваги та недоліки системи. До переваг було віднесено: можливість взаємодії з кінцевими користувачами в режимі реального часу, можливість синхронізації з серверами клієнтів в режимі реального часу, використання алгоритмів дискретної математики, код у відкритому доступі, швидкодія шляхом зберігання історичних даних. До недоліків було віднесено: підтримка тільки зі сторони спільноти програмістів, відсутній інтерфейс карт, немає специфічних функцій.

В результаті було проаналізовано та виконано всі заплановані цілі дипломного проєкту. Було створено простий, функціональний та зрозумілий інтерфейс програми, надана можливість легкого впровадження інших алгоритмів та сервісів, програма автономна та незалежна від операційної системи, створено зрозумілу технічну документацію на базі Vuepress та Swagger, та весь основний код покрито тестами.

ВИСНОВКИ

Під час виконання дипломного проєкту було створено систему геолокації та маршрутизації. Дана система надає можливість користувачам клієнтів швидко розраховувати маршрут та оптимізувати його. Система що дає можливість взаємодіяти різним серверам в режимі реального часу для моментальної синхронізації.

У першому розділі було розглянуто актуальність систем геолокації та маршрутизації в сучасному світі для потреб бізнесу та кінцевих користувачів. Було досліджено найвідоміші системи геолокації та маршрутизації, а саме Google maps API, Bing, Open Source Routing Machine, YOURS navigation API, Graphhopper Direction Map API, ArcGIS Direction API, MapBox Direction Map API, Here API, TomTom Routing API and Extended Routing API, Openroute service, MapQuest API. В результаті було порівняно всі системи, виокремлені їхні переваги та недоліки. В результаті аналізу наявних систем, було обрано найефективніші алгоритми з теорії графів, які використовують найвідоміші системи геолокації та маршрутизації. На основі аналізу було вибрано два сервіси, а саме Bing та Openroute service, які було використано в сервісі, для демонстрації того, що розроблена система може легко взаємодіяти з іншими сервісами.

У другому розділі було порівняно та обґрунтовано вибір засобів реалізації програми. Було досліджено основні, сучасні на момент написання дипломної роботи, технології та мови програмування для створення серверів. В результаті аналізу було обрано технологію Node.js на базі мови програмування TypeScript. Далі було проаналізовано засоби реалізації проєкту на базі Node.js, а саме фреймворки Express.js, Koa.js, Nest.js. Дослідивши всі переваги та недоліки фреймворків, було вибрано Nest.js, як сучасну й структуровану технологію. Далі було описано мову програмування TypeScript та її переваги над JavaScript.

Було вибрано IDE (Integrated Development Environment) Visual Studio Code, як зручне програмне забезпечення для написання коду. Далі було описано особливості бази даних MongoDB, обґрунтовано переваги бази даних. В кінці розділу описано платформу Heroku, яка була використана для розгортання проєкту. В результаті цього було розроблено систему на базі Node.js, що дозволило легко на швидко розробити HTTP сервер. За допомогою фреймворку Nest.js було побудовано зручну архітектуру проєкту, що дозволяє легко читати код та вдосконалювати систему. Мова програмування TypeScript дозволила швидко розробити систему і впевнитися в її надійності, через свою можливість статичної типізації. Розробляючи проєкт на Visual Studio Code, було дуже зручно переміщатися по проєкту, й запускати та тестувати систему зразу ж з IDE. База даних MongoDB, надала можливість ефективно взаємодіяти з Node.js, легко розширювати структуру даних та швидко обмінюватися інформацією. За допомогою платформи Heroku, розроблену систему вдалося безплатно розгорнути на віддаленому сервері, для спільного користування.

У третьому розділі було реалізовано систему геолокації та маршрутизації. У даному розділі описано архітектуру системи, та детально розказано про всі її файли та їх функції. Було описано кожен основний модуль програми, його функції та підключення. В наступному підрозділі було описано те, як система ідентифікує користувачів, обмежує клієнтів у правах доступу, та обробляє серверні чи клієнтські помилки. Для того, щоб бути точно впевненим у функціях програми, було створено автоматичні тести з заглушками. Далі описано про автоматичне документування проєкту за допомогою Swagger та Vuepress в поєднанні з jsdoc2md. В результаті цього, було розроблено систему геолокації та маршрутизації, яка має декілька ролей користувачів, надає можливості взаємодіяти користувачам з системою за допомогою HTTP, Websocket, та серверам за допомогою технології вебхуків. Система була покрита тестами, що гарантує її надійність та швидку перевірку при масштабуванні. В результаті було написано документацію, що дозволяє як

користувачам, так й іншим розробникам дізнатися інформацію про роботу самої системи.

У четвертому розділі було проведено детальний аналіз розробленої системи. У розділі показано готову систему, яка розроблялася протягом всієї дипломної роботи. Було показано використання кінцевих точок наявних в проєкті та надані додаткові відомості та вказівки щодо використання. Було розказано про метод ідентифікації користувача системи, показано результати оптимізації маршруту за різними алгоритмами. В кінці було проведено аналіз розробленої системи та вказано її основні переваги та недоліки, порівнюючи з іншими системами. В результаті тестування системи, було перевірено та показано її надійність та можливості. В результаті аналізу, було описано основні переваги та недоліки, обрано наступні кроки для вдосконалення розробленої системи.

В результаті була розроблена система геолокації та маршрутизації, яка задовольняє всі вимоги вказані на початку дипломної роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Insurance Information Institute, Inc. [Електронний ресурс] – Режим доступу до ресурсу: https://www.iii.org/fact-statistic/facts-statistics-drowsy-driving (дата звернення 20.05.2022).
2. Elizabeth Reid [Електронний ресурс] – Режим доступу до ресурсу: https://blog.google/products/maps/look-back-15-years-mapping-world/ (дата звернення 20.05.2022).
3. Google Cloud Google Maps platform [Електронний ресурс] – Режим доступу до ресурсу: https://mapsplatform.google.com/ (дата звернення 20.05.2022).
4. Bing Maps APIs [Електронний ресурс] – Режим доступу до ресурсу: https://www.microsoft.com/en-us/maps/choose-your-bing-maps-api (дата звернення 20.05.2022).
5. Bing Maps SDK for Android and iOS [Електронний ресурс] – Режим доступу до ресурсу: https://www.microsoft.com/en-us/maps/mobile/ (дата звернення 20.05.2022).
6. GitHub Project-OSRM [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Project-OSRM/osrm-backend (дата звернення 20.05.2022).
7. OSRM API Documentation [Електронний ресурс] – Режим доступу до ресурсу: http://project-osrm.org/docs/v5.5.1/api/#table-service (дата звернення 20.05.2022).
8. OpenSource Directions API Demo – YOURS Navigation API [Електронний ресурс] – Режим доступу до ресурсу: https://codezone4.wordpress.com/2013/10/07/opensource-directions-api-demo-yours-navigation-api/ (дата звернення 20.05.2022).
9. The GraphHopper Directions API Route Planning For Your Application [Електронний ресурс] – Режим доступу до ресурсу: https://www.graphhopper.com/ (дата звернення 20.05.2022).

10. GraphHopper Directions API documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.graphhopper.com/#tag/Routing-API (дата звернення 20.05.2022).
11. Documentation Mapping APIs and location services [Електронний ресурс] – Режим доступу до ресурсу: https://developers.arcgis.com/documentation/mapping-apis-and-services/routing (дата звернення 20.05.2022).
12. Mapbox maps and location for developers [Електронний ресурс] – Режим доступу до ресурсу: https://www.mapbox.com/ (дата звернення 20.05.2022).
13. Mapbox maps SDK for Android [Електронний ресурс] – Режим доступу до ресурсу: https://docs.mapbox.com/android/maps/guides/ (дата звернення 20.05.2022).
14. HERE API [Електронний ресурс] – Режим доступу до ресурсу: https://www.here.com/ (дата звернення 20.05.2022).
15. HERE API Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://developer.here.com/documentation/routing-api/dev_guide/index.html (дата звернення 20.05.2022).
16. Get started - HERE developer portal [Електронний ресурс] – Режим доступу до ресурсу: https://developer.here.com/documentation/map-image/dev_guide/topics/quick-start.html (дата звернення 20.05.2022).
17. Tomtom Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://developer.tomtom.com/routing-api/documentation/product-information/introduction (дата звернення 25.05.2022).
18. Tomtom Matrix Routing service [Електронний ресурс] – Режим доступу до ресурсу: https://developer.tomtom.com/routing-api/documentation/matrix-routing/matrix-routing-service (дата звернення 25.05.2022).
19. Openrouteservice [Електронний ресурс] – Режим доступу до ресурсу: https://openrouteservice.org/ (дата звернення 25.05.2022).

20. Openrouteservice documentation [Електронний ресурс] – Режим доступу до ресурсу: https://openrouteservice.org/dev/#/api-docs/v2/matrix (дата звернення 25.05.2022).
21. Mapquest [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mapquest.com/ (дата звернення 25.05.2022).
22. Jessica Clark The Best 10 Backend Programming Languages [Електронний ресурс] – Режим доступу до ресурсу: https://blog.back4app.com/backend-programming-languages-list/#:~:text=JavaScript%2C%20PHP%2C%20Python%2C%20Java,PHP%20as%20server%2Dside%20applications (дата звернення 25.05.2022).
23. stackoverflow.com statistyc [Електронний ресурс] – Режим доступу до ресурсу: https://insights.stackoverflow.com/survey/2021 (дата звернення 27.05.2022).
24. Python Advantages and Disadvantages [Електронний ресурс] – Режим доступу до ресурсу: https://techvidvan.com/tutorials/python-advantages-and-disadvantages/ (дата звернення 27.05.2022).
25. Advantages and Disadvantages of Java [Електронний ресурс] – Режим доступу до ресурсу: https://data-flair.training/blogs/pros-and-cons-of-java/ (дата звернення 27.05.2022).
26. Shailendra Sethiya Advantages & Disadvantages of Using @Nodejs [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@shailendrasethiya/advantages-disadvantages-of-using-nodejs-a2ae640fc141 (дата звернення 27.05.2022).
27. Npm express [Електронний ресурс] – Режим доступу до ресурсу: https://www.npmjs.com/package/express (дата звернення 28.05.2022).
28. Github express [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/expressjs/express (дата звернення 28.05.2022).
29. KoaJS [Електронний ресурс] – Режим доступу до ресурсу: https://koajs.com/ (дата звернення 28.05.2022).

30. Github Koa.js [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/koajs/koa (дата звернення 28.05.2022).
31. Nest.js [Електронний ресурс] – Режим доступу до ресурсу: https://nestjs.com/ (дата звернення 28.05.2022).
32. Github NestJS [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/nestjs/nest (дата звернення 28.05.2022).
33. Npm Nest.js [Електронний ресурс] – Режим доступу до ресурсу: https://www.npmjs.com/package/@nestjs/core (дата звернення 28.05.2022).
34. Nest.js documentation modules [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/modules (дата звернення 28.05.2022).
35. Nest.js documentation controllers [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/controllers (дата звернення 28.05.2022).
36. Nest.js documentation providers [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/providers (дата звернення 28.05.2022).
37. What is TypeScript [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org/ (дата звернення 29.05.2022).
38. Visual Studio Code introduction [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/docs/editor/whyvscode (дата звернення 29.05.2022).
39. MongoDB introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/ (дата звернення 29.05.2022).
40. Heroku introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.heroku.com/what (дата звернення 29.05.2022).
41. Алгоритм Флойда-Уоршелла [Електронний ресурс] – Режим доступу до ресурсу: https://intellect.icu/algoritm-flojda-uorshella-4135 (дата звернення 01.06.2022)
42. Алгоритм Беллмана — Форда [Електронний ресурс] – Режим доступу до ресурсу: https://intellect.icu/algoritm-bellmana-forda-bellman-ford-algorithm-4397 (дата звернення 02.06.2022)

43. Nest.js Guards [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/guards (дата звернення 02.06.2022)
44. Nest.js Exception filters [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/exception-filters (дата звернення 04.06.2022)
45. system-of-optimization-of-maprouting [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/BohdanShmalko/-system-of-optimization-of-maprouting

ДОДАТОК 1

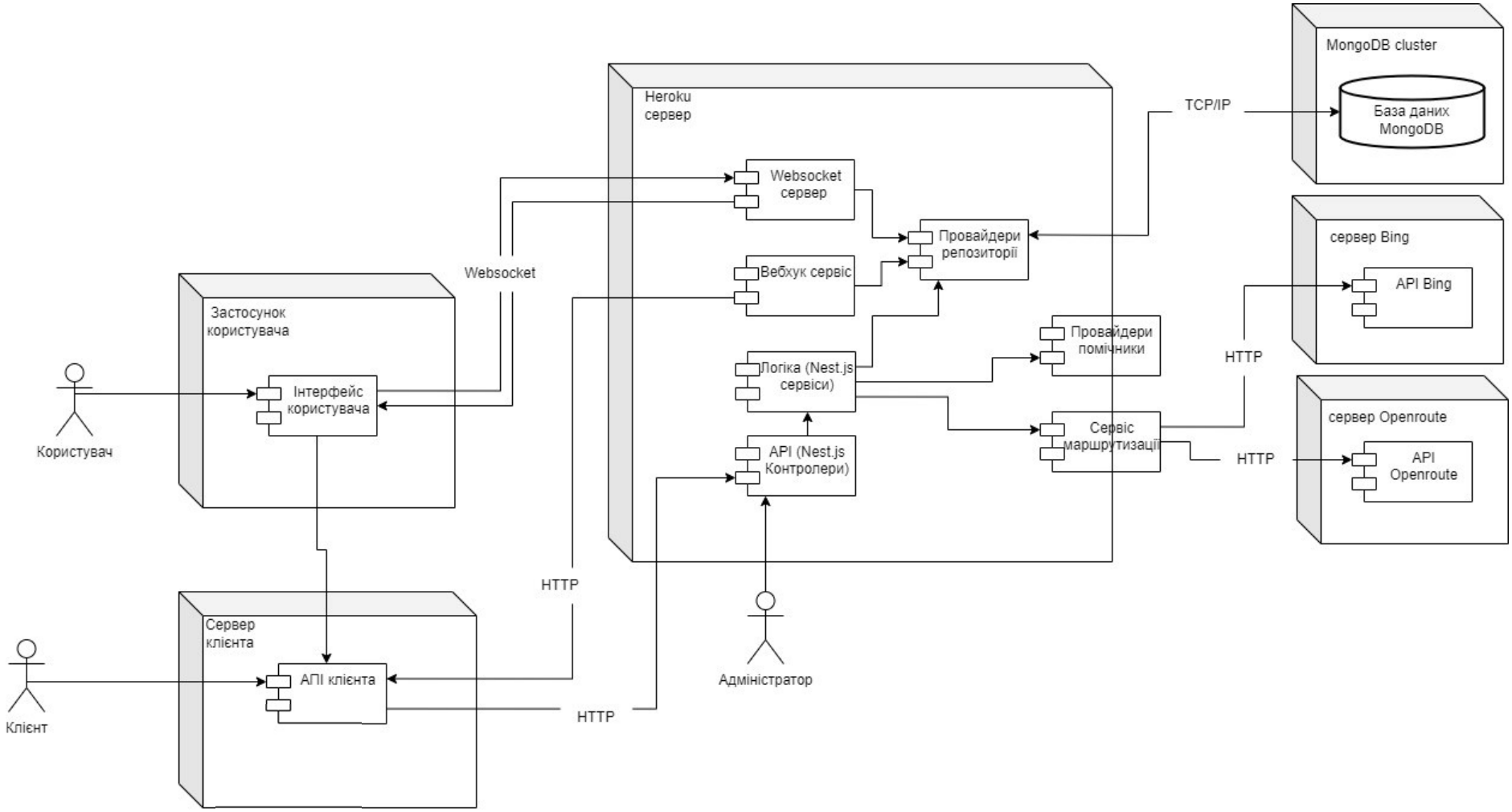
Система геолокації та маршрутизації

Структурна схема системи
(діаграма розгортання застосунку)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р



		№ докум.	Підпис	Дата
Розробив		Шмалько Б.І.		
Перевірив		Молчанова А.А.		
Реценз.				
Н. Контр.		Сімоненко В. П.		
Затвердив				

ІАЛЦ.467200.005 Д1

Система геолокації та маршрутизації

Структурна схема системи (діаграма розгортання застосунку)

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84

ДОДАТОК 2

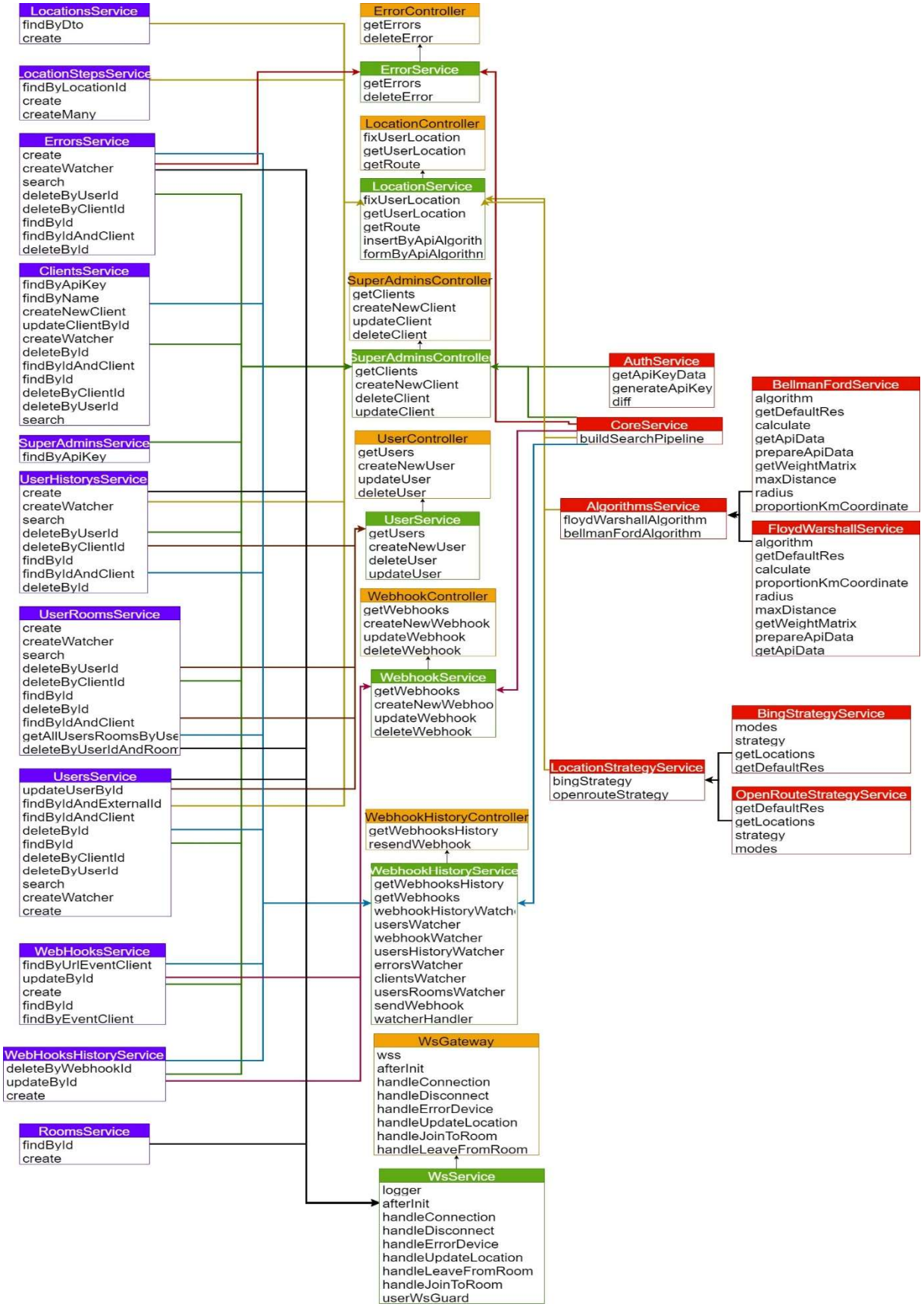
Система геолокації та маршрутизації

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.005 Д2
		№ докум.	Підпис	Дата	
Розробив		Шмалько Б.І.			**Система геолокації та маршрутизації** Функціональна схема (діаграма класів)
Перевірив		Молчанова А.А.			
Н. Контр.		Сімоненко В. П.			
Затвердив					

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84

ДОДАТОК 3

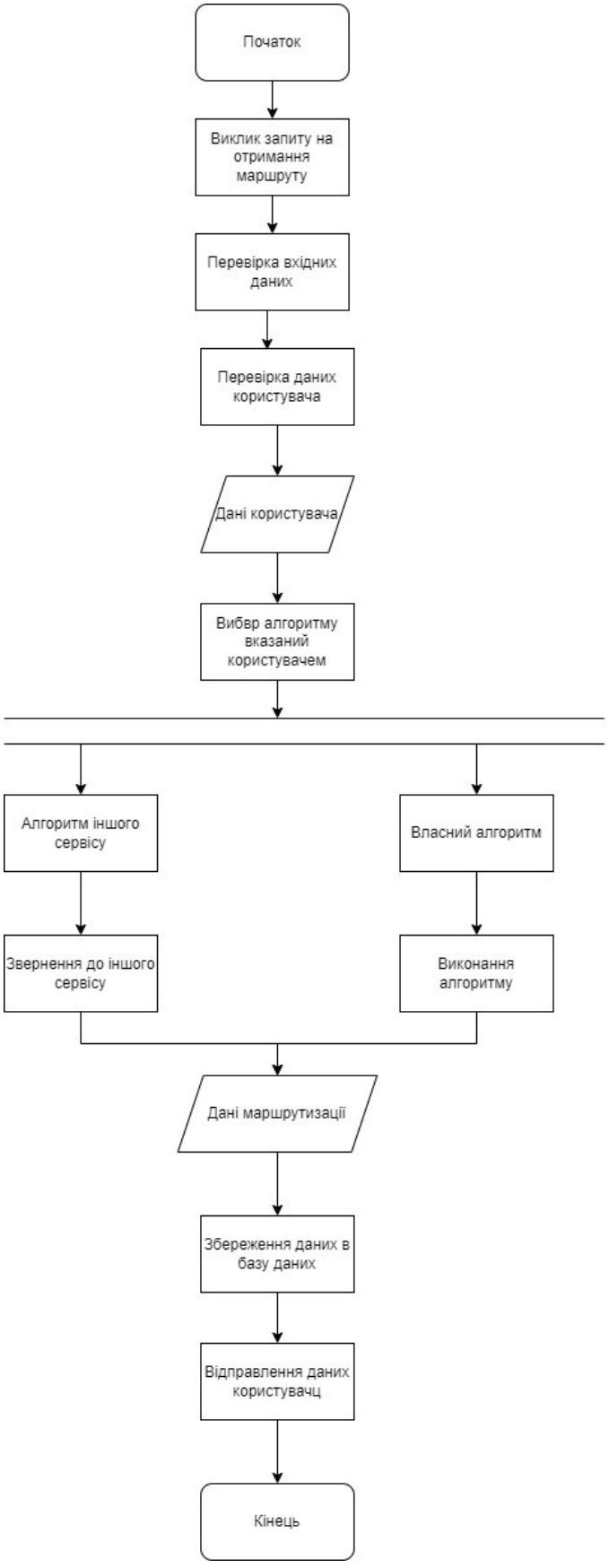
Система геолокації та маршрутизації

Принципова схема роботи

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.006 Д3
		№ докум.	Підпис	Дата	
Розробив		Шмалько Б.І.			**Система геолокації та маршрутизації** Принципова схема роботи
Перевірив		Молчанова А.А.			
Н. Контр.		Сімоненко В. П.			
Затвердив					

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84

ДОДАТОК 4

Система геолокації та маршрутизації

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 42

Київ 2022 р

main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { Logger, ValidationPipe } from '@nestjs/common';
import { writeFile } from "fs/promises";
import { AllExceptionFilter } from '@common';

/**
* Main app function
* @name bootstrap
* @kind function
*/
async function bootstrap() {
  const app = await NestFactory.create(AppModule, { cors: true });
  const port = process.env.PORT || 9100;
  const config = new DocumentBuilder()
    .setTitle('System for optimization of maprouting')
    .setDescription('Documentation REST API')
    .setVersion('1.0.0')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('/api', app, document);
  app.useGlobalPipes(
    new ValidationPipe({
      stopAtFirstError: true,
      transform: true,
      whitelist: true,
    }),

);
  await writeFile('./openapi.json', JSON.stringify(document, null, 2), { encoding: 'utf8'});
  app.useGlobalFilters(new AllExceptionFilter());
  if(process.env.docgen === 'true') await app.close();
  else await app.listen(port, () => new Logger('MAIN').log(`Server started on port ${port}`));
}
```

```
bootstrap();
```

bellman-ford.service.ts

```
import { HttpService } from '@nestjs/axios';
import { Injectable } from '@nestjs/common';
import { ILocationInput, ILocationsDots } from '../location-strategy/location-strategy.service';
import { BaseAlgorithmsService } from './base-algorithms.service';

/**
* BellmanFord Algorithm Service class
* @name BellmanFordService
* @kind class
*/
@Injectable()
export class BellmanFordService extends BaseAlgorithmsService {
  constructor(
    protected http: HttpService,
  ) { super(http) }

  public async algorithm(data: ILocationInput): Promise<ILocationsDots[]> {
    const { weightMatrix, allCoordinates } = await this.getWeightMatrix(data);
    if(isFinite(weightMatrix[0][weightMatrix.length - 1].distance)){
      return this.getDefaultRes(data);
    }
    const V = allCoordinates.length;
    const graph = [];
    weightMatrix.forEach((matrix, i) => {
        matrix.forEach((el, j) => {
            if(el => el.distance !== 0 && isFinite(el.distance))
                graph.push([i, j, el.distance]);
        })
    })
    const E = graph.length;
    const src = 0;
    const dis = Array(V).fill(Infinity);

    dis[src] = 0;
    const allDots = weightMatrix.map((_, i) => [i]);

    for (let i = 0; i < V - 1; i++) {
        for (let j = 0; j < E; j++) {
            const fromIndex = graph[j][0];
            const fromDistance = dis[fromIndex];
```

```
            const toIndex = graph[j][1];
            const toDistanceOther = graph[j][2];
            const toDistance = dis[toIndex];
            if ((fromDistance + toDistanceOther) < toDistance) {
                dis[toIndex] = dis[fromIndex] + toDistanceOther;
                allDots[toIndex] = [fromIndex, ...allDots[fromIndex]];
            }
        }
    }
    if(allDots[weightMatrix.length - 1].length < 2) return this.getDefaultRes(data);

    return allDots[weightMatrix.length - 1].map((index, step) => ({
        lat: allCoordinates[index].lat.toString(),
        lon: allCoordinates[index].lon.toString(),
        step: step + 1,
      })
    )
  }
}
```

floyd-warshall.service.ts

```
import { HttpService } from '@nestjs/axios';
import { Injectable } from '@nestjs/common';
import { ILocationInput, ILocationsDots } from '../location-strategy/location-strategy.service';
import { BaseAlgorithmsService } from './base-algorithms.service';

/**
* FloydWarshall Algorithm Service class
* @name FloydWarshallService
* @kind class
*/
@Injectable()
export class FloydWarshallService extends BaseAlgorithmsService {
  constructor(
    protected http: HttpService,
  ) { super(http) }

  public async algorithm(data: ILocationInput): Promise<ILocationsDots[]> {
    const { weightMatrix, allCoordinates } = await this.getWeightMatrix(data);
    if(isFinite(weightMatrix[0][weightMatrix.length - 1].distance)){
      return this.getDefaultRes(data);
    }
    weightMatrix.forEach((_, i) => {
```

```
      weightMatrix.forEach((_, j) => {
        weightMatrix.forEach((_, k) => {
          const ikDistance = weightMatrix[i][k].distance;
          const kjDistance = weightMatrix[k][j].distance;
          const ijDistance = weightMatrix[i][j].distance;
          if (ikDistance + kjDistance < ijDistance)
            weightMatrix[i][j] = {
              distance: ikDistance + kjDistance,
              longDistance: weightMatrix[i][j].longDistance,
              dots: [...weightMatrix[i][k].dots, ...weightMatrix[k][j].dots.slice(1)],
            };
          });
         });
      });
    return weightMatrix[0][weightMatrix.length - 1].dots.map((index, step) => ({
        lat: allCoordinates[index].lat.toString(),
        lon: allCoordinates[index].lon.toString(),
        step: step + 1,
      })
    )
  }
}
```

bing-strategy.service.ts

```
import { ETransports } from '@db';
import { HttpService } from '@nestjs/axios';
import { Injectable } from '@nestjs/common';
import { BaseLocationStrategyService } from './base-location-startegy.service';
import { ILocationInput, ILocationsDots } from './location-strategy.service';
import { BingResponseType } from './strategy.type';

/**
* Google Strategy Service class
* @name BingStrategyService
* @kind class
*/
@Injectable()
export class BingStrategyService extends BaseLocationStrategyService {
  constructor(
    private http: HttpService,
  ) { super() }

  public modes = {
    [ETransports.Walking]: 'walking',
```

```
    [ETransports.Car]: 'driving',
    [ETransports.Transit]: 'transit',
  };

  public async strategy(data: ILocationInput): Promise<ILocationsDots[]> {
    if(!this.modes[data.transport]) return this.getDefaultRes(data);
    const apiData = await this.getLocations(data);
    return apiData.resourceSets[0]?.resources[0]?.routeLegs[0]?.itineraryItems?.map(
      (el, step) => ({
        lat: el.maneuverPoint.coordinates[0].toString(),
        lon: el.maneuverPoint.coordinates[1].toString(),
        step,
      })
    ) || this.getDefaultRes(data);
  }
private async getLocations(data: ILocationInput): Promise<BingResponseType> {
    const mode = this.modes[data.transport];
    const result = await this.http.get(
      `http://dev.virtualearth.net/REST/V1/Routes/${mode}`,
      {
        params: {
          'wp.0': `${data.startLat},${data.startLon}`,
          'wp.1': `${data.endLat},${data.endLon}`,
          key: process.env.BING_KEY,
        }
      }
    ).toPromise();
    return result.data;
  }
}
```

openroute-strategy.service.ts

```
import { ETransports } from '@db';
import { HttpService } from '@nestjs/axios';
import { Injectable } from '@nestjs/common';
import { BaseLocationStrategyService } from './base-location-startegy.service';
import { ILocationInput, ILocationsDots } from './location-strategy.service';
import { OpenRouteResponseType } from './strategy.type';

/**
* Openroute Strategy Service class
* @name OpenRouteStrategyService
* @kind class
*/
```

```
@Injectable()
export class OpenRouteStrategyService extends BaseLocationStrategyService {
  constructor(
    private http: HttpService,
  ) { super() }

  public modes = {
    [ETransports.Car]: 'driving-car',
    [ETransports.Hgv]: 'driving-hgv',
    [ETransports.Bike]: 'cycling-regular',
  };

  public async strategy(data: ILocationInput): Promise<ILocationsDots[]> {
    if(!this.modes[data.transport]) return this.getDefaultRes(data);
    const apiData = await this.getLocations(data);
    return apiData?.features[0]?.geometry?.coordinates?.map(
      (el, step) => ({
        lat: el[0].toString(),
        lon: el[1].toString(),
        step,
      })
    ) || this.getDefaultRes(data);
  }

  private async getLocations(data: ILocationInput): Promise<OpenRouteResponseType> {
    const mode = this.modes[data.transport];
    const result = await this.http.get(
      `https://api.openrouteservice.org/v2/directions/${mode}`,
      {
        params: {
          start: `${data.startLat},${data.startLon}`,
          end: `${data.endLat},${data.endLon}`,
          api_key: process.env.OPENROUTE_KEY,
        }
      }
    ).toPromise();
    return result.data;
  }
}
```

super-admins.service.ts

```
import {
    AuthService,
    ClientIdDto,
```

```
    CoreService,
    CreateClientDto,
    EHttpExceptionMessage,
    GetClientDto,
    UpdateClientDto
} from '@common';
import {
    Clients,
    ClientsService,
    ErrorsService,
    SuperAdminsService,
    UserHistorysService,
    UserRoomsService,
    UsersService,
    WebHooksHistoryService,
    WebHooksService
} from '@db';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';

/**
* SuperAdmins service class
* @name SuperAdminsService
* @kind class
*/
@Injectable()
export class SuperAdminsApiService {
    constructor(
        private readonly superAdminsService: SuperAdminsService,
        private readonly clientsService: ClientsService,
        private readonly authService: AuthService,
        private readonly core: CoreService,
        private readonly webhooksService: WebHooksService,
        private readonly webhooksHistoryService: WebHooksHistoryService,
        private readonly usersService: UsersService,
        private readonly userRoomsService: UserRoomsService,
        private readonly userHistoryService: UserHistorysService,
        private readonly errorsService: ErrorsService,
    ) {}

    /**
    * Get clients
    * @name getClients
    * @kind function
```

```
    * @property {Object}  query  - query object dto
    * @property {Object}  req  - req object
    * @returns {Object} clients documents
    */
     async getClients(req, query: GetClientDto): Promise<Clients[]> {
        const searchObj = this.core.buildSearchPipeline(req.client, query);
        try {
            const result = await this.clientsService.search(searchObj);
            return result;
        }catch(e) {
            throw            new            HttpException(EHttpExceptionMessage.InvalidQuery,
HttpStatus.BAD_REQUEST);
        }
    }

    /**
    * Create new client
    * @name createNewClient
    * @kind function
    * @property {Object}  dto  - client dto
    * @property {Object}  req  - req object
    * @returns {Object} new client document
    */
    async createNewClient(req, dto: CreateClientDto): Promise<Clients> {
        const adminId = req.client.document._id;
        const existingClient = await this.clientsService.findByName(dto.name);
        if(existingClient) throw new HttpException(
            EHttpExceptionMessage.ClientWithNameExist,
            HttpStatus.CONFLICT
        );
        const newClient = await this.clientsService.createNewClient({
            ...dto,
            adminCreated: adminId,
            adminUpdated: adminId,
            apiKey: this.authService.generateApiKey(),
        });
        return newClient;
    }

    /**
    * Update client
    * @name updateClient
    * @kind function
```

```
* @property {Object}  dto  - update client dto
* @property {Object}  param  - client id dto
* @property {Object}  req  - req object
* @returns {Object} new client document
*/
 async updateClient(req, param: ClientIdDto, dto: UpdateClientDto): Promise<Clients> {
    const adminId = req.client.document._id;
    const existingClient = await this.clientsService.findById(param.clientId);
    if(!existingClient) throw new HttpException(
        EHttpExceptionMessage.ClientNotExistApiKey,
        HttpStatus.BAD_REQUEST
    );
    if(dto.name && existingClient.name !== dto.name) {
        const clientWithSameName = await this.clientsService.findByName(dto.name);
        if(clientWithSameName) throw new HttpException(
            EHttpExceptionMessage.ClientWithNameExist,
            HttpStatus.CONFLICT
        );
    }
    const newClientSet: any = {
        ...dto,
        adminUpdated: adminId,
    }
    if(dto.newApiKey) newClientSet.apiKey = dto.newApiKey;
    delete newClientSet.newApiKey;
    const updatedClient = await this.clientsService.updateClientById(
        existingClient._id,
        newClientSet,
    );
    return updatedClient;
}

/**
* Delete client
* @name deleteClient
* @kind function
* @property {Object}  req  - req object
* @property {Object}  param  - client id dto
* @returns {string} delete status
*/
async deleteClient(req, param: ClientIdDto): Promise<string> {
    const adminId = req.client.document._id;
    const clientId = param.clientId;
```

```
        const existingUser = await this.clientsService.findById(clientId);
        if(!existingUser) throw new HttpException(
            EHttpExceptionMessage.ClientNotExistApiKey,
            HttpStatus.BAD_REQUEST
        );
        await this.errorsService.deleteByClientId(clientId);
        await this.userHistoryService.deleteByClientId(clientId);
        await this.userRoomsService.deleteByClientId(clientId);
        await this.usersService.deleteByClientId(clientId);
        await this.webhooksHistoryService.deleteByClientId(clientId);
        await this.webhooksService.deleteByClientId(clientId);
        await this.clientsService.deleteById(clientId);
        return 'ok';
    }
}
```

user.service.ts

```
import {
  CoreService,
  CreateUserDto,
  EHttpExceptionMessage,
  GetUsersDto,
  UpdateUserDto,
  UserIdDto
} from '@common';
import {
  ErrorsService,
  UserHistorysService,
  UserRoomsService,
  Users,
  UsersService
} from '@db';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';

/**
* User service class
* @name UserService
* @kind class
*/
@Injectable()
export class UserService {
  constructor(
    private readonly usersService: UsersService,
    private readonly core: CoreService,
```

```
  private readonly userRoomsService: UserRoomsService,
  private readonly userHistoryService: UserHistorysService,
  private readonly errorsService: ErrorsService,
) {}

/**
* Get users
* @name getUsers
* @kind event
* @property {Object}  query  - query object dto
* @property {Object}  req  - req object
* @returns {Object} users documents
*/
async getUsers(req, query: GetUsersDto): Promise<Users[]> {
  const searchObj = this.core.buildSearchPipeline(req.client, query);
  try {
      const result = await this.usersService.search({...searchObj, select:
        {
          externalId: 1,
          algorithm: 1,
          optionalParams: 1,
        }
      });
      return result;
  }catch(e) {
      throw new HttpException(EHttpExceptionMessage.InvalidQuery, HttpStatus.BAD_REQUEST);
  }
}

/**
* Create new users
* @name createNewUser
* @kind event
* @property {Object}  dto  - create user dto
* @property {Object}  req  - req object
* @returns {Object} new user document
*/
async createNewUser(req, dto: CreateUserDto): Promise<Users> {
  const clientId = req.client.document._id;
  const newUser = await this.usersService.create({
    ...dto,
    clientId,
  });
```

```
  return newUser;
}

/**
* Update user
* @name updateUser
* @kind event
* @property {Object}  dto  - update user dto
* @property {Object}  param  - user id dto
* @property {Object}  req  - req object
* @returns {Object} new user document
*/
async updateUser(req, param: UserIdDto, dto: UpdateUserDto): Promise<Users> {
  const userId = param.userId;
  const clientId = req.client.document._id;
  const existingDocument = await this.usersService.findByIdAndClient(userId, clientId);
  if(!existingDocument) throw new HttpException(
    EHttpExceptionMessage.UserNotExistId,
    HttpStatus.BAD_REQUEST
  );
  const updatedUser = await this.usersService.updateUserById(userId, dto);
  return updatedUser;
}

/**
* Delete user
* @name deleteUser
* @kind function
* @property {Object}  req  - req object
* @property {Object}  param  - user id dto
* @returns {string} delete status
*/
 async deleteUser(req, param: UserIdDto): Promise<string> {
  const userId = param.userId;
  const clientId = req.client.document._id;
  const existingDocument = await this.usersService.findByIdAndClient(userId, clientId);
  if(!existingDocument) throw new HttpException(
    EHttpExceptionMessage.UserNotExistId,
    HttpStatus.BAD_REQUEST
  );
  await this.userHistoryService.deleteByUserId(userId);
  await this.userRoomsService.deleteByUserId(userId);
  await this.errorsService.deleteByUserId(userId);
```

```
    await this.usersService.deleteById(userId);
    return 'ok';
  }
}
```

error.service.ts

```
import { CoreService, EHttpExceptionMessage, ErrorIdDto, GetErrorsDto } from '@common';
import { Errors, ErrorsService } from '@db';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';

/**
* Error service class
* @name ErrorService
* @kind class
*/
@Injectable()
export class ErrorService {
    constructor(
      private readonly errorsService: ErrorsService,
      private readonly core: CoreService,
    ) {
    }
    /**
    * Get errors
    * @name getErrors
    * @kind function
    * @property {Object}  query  - query object dto
    * @property {Object}  req  - req object
    * @returns {Object} errors documents
    */
    async getErrors(req, query: GetErrorsDto): Promise<Errors[]> {
      const searchObj = this.core.buildSearchPipeline(req.client, query);
    try {
        const result = await this.errorsService.search({...searchObj, select: { userId: 1,
message: 1 }});
        return result;
    }catch(e) {
        throw new HttpException(EHttpExceptionMessage.InvalidQuery, HttpStatus.BAD_REQUEST);
    }
    }

   /**
   * Delete error
   * @name deleteError
```

```
   * @kind function
   * @property {Object}  req  - req object
   * @property {Object}  param  - error id dto
   * @returns {string} delete status
   */
    async deleteError(req, param: ErrorIdDto): Promise<string> {
      const errorId = param.errorId;
      const clientId = req.client.document._id;
      const existingDocument = await this.errorsService.findByIdAndClient(errorId, clientId);
      if(!existingDocument) throw new HttpException(
        EHttpExceptionMessage.ErrorNotExist,
        HttpStatus.BAD_REQUEST
      );
      await this.errorsService.deleteById(errorId);
      return 'ok';
    }
}
```

location.service.ts

```
import    {    EAlgorithms,    LocationsService,    LocationStepsService,    UserHistorys,
UserHistorysService, UsersService } from '@db';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';
import {
  algorithmSettingsDataset,
  AlgorithmsService,
  CoreService,
  CreateUserHistoryDto,
  EHttpExceptionMessage,
  ELocationServiceName,
  GetLocationDto,
  GetUserHistorysDto,
  LocationStepsDto,
  LocationStrategyService,
} from '@common';

/**
* Location service class
* @name LocationService
* @kind class
*/
@Injectable()
export class LocationService {

  constructor(
```

```
    private readonly locationsService: LocationsService,
    private readonly locationStepsService: LocationStepsService,
    private readonly userHistoryService: UserHistorysService,
    private readonly usersService: UsersService,
    private readonly core: CoreService,
    private readonly algorithmService: AlgorithmsService,
    private readonly locationStrategyService: LocationStrategyService,
  ){
    this[ELocationServiceName.AlgorithmService] = this.algorithmService;
    this[ELocationServiceName.LocationService] = this.locationStepsService;
  }

  /**
  * Save user location in database
  * @name fixUserLocation
  * @kind function
  * @property {Object}  req  - req object
  * @property {Object}  dto  - user history create dto
  * @returns {Object} UserHistory document
  */
  async fixUserLocation(req, dto: CreateUserHistoryDto): Promise<UserHistorys> {
    const clientId = req.client.document._id;
    const user = await this.usersService.findByIdAndClient(dto.userId, clientId);
    if(!user)       throw       new       HttpException(EHttpExceptionMessage.UserNotExistId,
HttpStatus.FORBIDDEN);
    const userHistory = await this.userHistoryService.create({
      ...dto as any,
      clientId,
    });
    return userHistory;
  }

  /**
  * Get user location history
  * @name getUserLocation
  * @kind function
  * @property {Object}  query  - query object dto
  * @property {Object}  req  - req object
  * @returns {Object} UserHistory documents
  */
  async getUserLocation(req, query: GetUserHistorysDto): Promise<UserHistorys[]> {
    const searchObj = this.core.buildSearchPipeline(req.client, query);
    try {
```

```
      const result = await this.userHistoryService.search({...searchObj, select:
        {
          lat: 1,
          lon:1,
          time: 1,
          userId: 1,
        }
      });
      return result;
    }catch(e) {
      throw new HttpException(EHttpExceptionMessage.InvalidQuery, HttpStatus.BAD_REQUEST);
    }
  }

  /**
  * Get rout between locations
  * @name getRoute
  * @kind function
  * @property {Object}  query  - query object dto
  * @property {Object}  req  - req object
  * @returns {Object} UserHistory documents
  */
  async getRoute(req, query: GetLocationDto): Promise<LocationStepsDto[]> {
    const clientId = req.client.document._id;
    const user = await this.usersService.findByIdAndClient(query.userId, clientId);
    const apiStrategys = Object.keys(algorithmSettingsDataset);
    if(!user)       throw       new       HttpException(EHttpExceptionMessage.UserNotExistId,
HttpStatus.FORBIDDEN);
    const algorithm = user.algorithm;
    const locationDto = {
      ...query,
      algorithm,
    }
    const location = await this.locationsService.findByDto(locationDto);
    if(location){
      const steps = await this.locationStepsService.findByLocationId(location._id);
      if(steps) return steps.map(step => ({
        locationId: location._id.toString(),
        lat: step.lat,
        lon: step.lon,
        step: step.step,
      }));
    }
```

```
    const newLocation = await this.locationsService.create(locationDto);
    return this.formByApiAlgorithms(
        apiStrategys,
        algorithm,
        newLocation,
        locationDto,
    )
  }

  private async insertByApiAlgorithm(func, locationId, locationDto, withId) {
    const allSteps = await func(locationDto);
          const result = allSteps.map(obj => ({
            ...obj,
            locationId,
          }));
    await this.locationStepsService.createMany(result);
    return withId ? result : allSteps;
  }

  private async formByApiAlgorithms(apiStrategys, algorithm, newLocation, locationDto, withId
= true) {
    for(const strategy of apiStrategys) {
      if(strategy === algorithm) {
        const { service, funcName } = algorithmSettingsDataset[strategy];
        return this.insertByApiAlgorithm(
          this[service][funcName].bind(this.locationStrategyService),
          newLocation._id.toString(),
          locationDto,
          withId,
        )
      }
    }
    return [];
  }
}
```

webhooks.service.ts

```
import { ClientIdDto, CoreService, CreateWebhookDto, EHttpExceptionMessage, GetWebhooksDto,
UpdateWebhookDto, WebhookIdDto } from '@common';
import { WebHooks, WebHooksHistoryService, WebHooksService } from '@db';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';
const URL = require("url").URL;

/**
```

```
* Webhook service class
* @name WebhookService
* @kind class
*/
@Injectable()
export class WebhookService {
  constructor(
    private readonly webhookService: WebHooksService,
    private readonly webhookHistoryService: WebHooksHistoryService,
    private readonly core: CoreService,
  ) {
  }

  /**
  * Get Webhooks
  * @name getWebhooks
  * @kind function
  * @property {Object}  query  - query object dto
  * @property {Object}  req  - req object
  * @returns {Object} Webhooks documents
  */
  async getWebhooks(req, query: GetWebhooksDto): Promise<WebHooks[]> {
    const searchObj = this.core.buildSearchPipeline(req.client, query);
    try {
        const result = await this.webhookService.search({...searchObj, select:
          {
            url: 1,
            event: 1,
            name: 1,
            userId: 1,
          }
        });
        return result;
    }catch(e) {
        throw new HttpException(EHttpExceptionMessage.InvalidQuery, HttpStatus.BAD_REQUEST);
    }
  }

  /**
  * Create new Webhooks
  * @name createNewWebhook
  * @kind function
  * @property {Object}  dto  - create Webhook dto
```

```
  * @property {Object}  req  - req object
  * @returns {Object} new Webhook document
  */
  async createNewWebhook(req, dto: CreateWebhookDto): Promise<WebHooks> {
    const clientId = req.client.document._id;
    const existWebhook = await this.webhookService.findByUrlEventClient({
      url: dto.url,
      event: dto.event,
      clientId,
    });
    if(existWebhook) throw new HttpException(
      EHttpExceptionMessage.WebHookExist,
      HttpStatus.BAD_REQUEST
    );
    if(!this.isValidUrl(dto.url)) throw new HttpException(
      EHttpExceptionMessage.InvalidUrl,
      HttpStatus.BAD_REQUEST
    );
    const newWebhook = await this.webhookService.create({
      ...dto,
      clientId,
    })
    return newWebhook;
  }

  /**
  * Update Webhook
  * @name updateWebhook
  * @kind function
  * @property {Object}  dto  - update Webhook dto
  * @property {Object}  param  - Webhook id dto
  * @property {Object}  req  - req object
  * @returns {Object} new Webhook document
  */
  async updateWebhook(req, param: WebhookIdDto, dto: UpdateWebhookDto): Promise<WebHooks> {
    const webhookId = param.webhookId;
    const clientId = req.client.document._id;
    const existingDocument = await this.webhookService.findByIdAndClient(webhookId,
clientId);
    if(!existingDocument) throw new HttpException(
      EHttpExceptionMessage.WebHookNotExistId,
      HttpStatus.BAD_REQUEST
    );
```

```
    if(dto.url && !this.isValidUrl(dto.url)) throw new HttpException(
      EHttpExceptionMessage.InvalidUrl,
      HttpStatus.BAD_REQUEST
    );
    const updatedUser = await this.webhookService.updateById(webhookId, dto);
    return updatedUser;
  }

  /**
  * Delete Webhook
  * @name deleteWebhook
  * @kind function
  * @property {Object}  req  - req object
  * @property {Object}  param  - Webhook id dto
  * @returns {string} delete status
  */
  async deleteWebhook(req, param: WebhookIdDto): Promise<string> {
    const webhookId = param.webhookId;
    const clientId = req.client.document._id;
    const   existingDocument   =   await   this.webhookService.findByIdAndClient(webhookId,
clientId);
    if(!existingDocument) throw new HttpException(
      EHttpExceptionMessage.WebHookNotExistId,
      HttpStatus.BAD_REQUEST
    );
    await this.webhookHistoryService.deleteByWebhookId(webhookId);
    await this.webhookService.deleteById(webhookId);
    return 'ok';
  }

  private isValidUrl(s) {
    try {
      new URL(s);
      return true;
    } catch (err) {
      return false;
    }
  };
}
```

webhooks-history.service.ts

```
import { CoreService, EHttpExceptionMessage, GetWebhooksHistoryDto, ResendWebhookDto } from
'@common';
```

```
import { ClientsService, ErrorsService, EWebhookEvents, UserHistorysService,
UserRoomsService, UsersService, WebHooksHistory, WebHooksHistoryService, WebHooksService }
from '@db';
import { HttpService } from '@nestjs/axios';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';
import { ObjectId } from 'mongoose';
const urlModule = require('url');

/**
* WebhookHistory service class
* @name WebhookHistoryService
* @kind class
*/
@Injectable()
export class WebhookHistoryService {
    constructor(
        private readonly core: CoreService,
        private readonly http: HttpService,
        private readonly webhookHistoryService: WebHooksHistoryService,
        private readonly webhookService: WebHooksService,
        private readonly clientsService: ClientsService,
        private readonly usersService: UsersService,
        private readonly userRoomsService: UserRoomsService,
        private readonly userHistorysService: UserHistorysService,
        private readonly errorsService: ErrorsService,
    ) {}

    /**
    * Get Webhooks history
    * @name getWebhooksHistory
    * @kind function
    * @property {Object}  query  - query object dto
    * @property {Object}  req  - req object
    * @returns {Object} Webhooks history documents
    */
    async getWebhooksHistory(req, query: GetWebhooksHistoryDto): Promise<WebHooksHistory[]> {
        const searchObj = this.core.buildSearchPipeline(req.client, query);
        try {
            const result = await this.webhookHistoryService.search({...searchObj, select:
                {
                    webhookId: 1,
                    data: 1,
                    success: 1,
```

```
                    userId: 1,
                }
            });
            return result;
        }catch(e) {
            throw            new            HttpException(EHttpExceptionMessage.InvalidQuery,
HttpStatus.BAD_REQUEST);
        }
    }

    /**
    * Resend webhook
    * @name resendWebhook
    * @kind function
    * @property {Object}  query  - query object dto
    * @property {Object}  req  - req object
    * @returns {Object} Webhooks history document
    */
    async resendWebhook(req, dto: ResendWebhookDto): Promise<WebHooksHistory> {
        const clientId = req.client.document._id;
        const existWebhookHistory = await this.webhookHistoryService.findByIdAndClient(
            dto.webhookHistoryId,
            clientId,
        );
        if(!existWebhookHistory) throw new HttpException(
            EHttpExceptionMessage.WebHookHistoryNotExistId,
            HttpStatus.BAD_REQUEST,
        );
        const webhook = await this.webhookService.findById(existWebhookHistory.webhookId);
        if(!webhook) throw new HttpException(
            EHttpExceptionMessage.WebHookNotExistId,
            HttpStatus.INTERNAL_SERVER_ERROR,
        );
        const    {    error,    message    }    =    await    this.sendWebhook(webhook.url,
existWebhookHistory.data);
        await this.webhookHistoryService.updateById(dto.webhookHistoryId, {
            success: !error,
        });
        if(error) throw new HttpException(
            message,
            HttpStatus.BAD_REQUEST,
        );
        existWebhookHistory.clientId = undefined;
```

```
        return existWebhookHistory;
    }

    /**
    * Function that send webhooks
    * @name webhookHistoryWatcher
    * @kind function
    */
    webhookHistoryWatcher() {
        this.webhookHistoryService.createWatcher().on('change', async (next) => {
            if (next.operationType === 'insert') {
                const webhookHistory = next.fullDocument;
                const webhook = await this.webhookService.findById(webhookHistory.webhookId);
                if(!webhook) return
                const { error, message } = await this.sendWebhook(webhook.url,
webhookHistory.data);
                await this.webhookHistoryService.updateById(webhookHistory._id, {
                    success: !error,
                });
            }
        })
    }

    /**
    * Function that send webhooks if user changed/created
    * @name usersWatcher
    * @kind function
    */
    usersWatcher() {
        this.usersService.createWatcher().on('change', this.watcherHandler(
            this.usersService,
            'user',
            {
                insert: EWebhookEvents.UserCreated,
                update: EWebhookEvents.UserUpdated,
                delete: EWebhookEvents.UserDeleted,
            }
        ))
    }

    /**
    * Function that send webhooks if webhook changed/created
    * @name webhookWatcher
```

```
* @kind function
*/
webhookWatcher() {
    this.webhookService.createWatcher().on('change', this.watcherHandler(
        this.webhookService,
        'webhook',
        {
            insert: EWebhookEvents.WebhookCreated,
            update: EWebhookEvents.WebhookUpdated,
            delete: EWebhookEvents.WebhookDeleted,
        }
    ))
}

/**
* Function that send webhooks if user history created or deleted
* @name usersHistoryWatcher
* @kind function
*/
usersHistoryWatcher() {
    this.userHistorysService.createWatcher().on('change', this.watcherHandler(
        this.userHistorysService,
        'userHistory',
        {
            insert: EWebhookEvents.UserHistoryCreated,
            delete: EWebhookEvents.UserHistoryDeleted,
        }
    ))
}

/**
* Function that send webhooks if error created or deleted
* @name errorsWatcher
* @kind function
*/
errorsWatcher() {
    this.errorsService.createWatcher().on('change', this.watcherHandler(
        this.errorsService,
        'error',
        {
            insert: EWebhookEvents.ErrorCreated,
            delete: EWebhookEvents.ErrorDeleted,
        }
```

```
    ))
}

/**
* Function that send webhooks if client updated
* @name clientsWatcher
* @kind function
*/
 clientsWatcher() {
    this.clientsService.createWatcher().on('change', this.watcherHandler(
        this.clientsService,
        'client',
        {
            update: EWebhookEvents.ClientUpdated,
        }
    ))
}

/**
* Function that send webhooks if usersRoom created or deleted
* @name usersRoomsWatcher
* @kind function
*/
 usersRoomsWatcher() {
    this.userRoomsService.createWatcher().on('change', this.watcherHandler(
        this.userRoomsService,
        'userRoom',
        {
            insert: EWebhookEvents.UserRoomsCreated,
            delete: EWebhookEvents.UserRoomsDeleted,
        }
    ))
}

private async sendWebhook(url, data) {
    const domain = urlModule.parse(url).hostname;
    const badLocal = domain === 'localhost' && process.env.USE_LOCAL_WEBHOOK !== 'true';
    if(!domain || badLocal)
        return { error: true, message: EHttpExceptionMessage.InvalidDomain };
    try{
        await this.http.post(url, data).toPromise();
        return { error: false };
    }catch(e) {
```

```
            return { error: true, message: EHttpExceptionMessage.InvalidWebhookResponse };
        }
    }

    private watcherHandler(service: any, name: string, events: any) {
        return async (next) => {
            const data = await service.findById(next.documentKey._id);
            let event;
            const eventsName = Object.keys(events);
            try{
                for(const eventName of eventsName) {
                    if(next.operationType === eventName){
                        event = events[eventName];
                        break;
                    }
                }
                if(event) {
                    const webhooks: any[] = await this.webhookService.findByEventClient(
                        data.clientId,
                        event,
                    )
                    for(const webhook of webhooks) {
                        await this.webhookHistoryService.create({
                            clientId: data.clientId,
                            success: false,
                            webhookId: webhook._id,
                            data: {
                                event,
                                name: webhook.name,
                                [name]: data,
                            }
                        });
                    }
                }
            }catch(e) {
                console.log(e);
            }
        }
    }
}
```

ws.service.ts

```
import {Injectable, Logger} from '@nestjs/common';
import {WsResponse} from "@nestjs/websockets";
```

```
import {Socket, Server} from "socket.io";
import {
  AuthService,
  CreateConnectionDto,
  DeviceErrorDto,
  EWsExceptionMessage,
  JoinRoomDto,
  UserLocationTimeDto,
  WsResponseDto
} from "@common";
import {
  ErrorsService,
  RoomsService,
  UserHistorysService,
  UserRoomsService,
  UsersService
} from '@db';
import { WsEnum } from './ws.enum';

/**
* Websocket service class
* @name WsService
* @kind class
*/
@Injectable()
export class WsService {
  constructor(
    private readonly usersService: UsersService,
    private readonly authService: AuthService,
    private readonly roomsService: RoomsService,
    private readonly userHistorysService: UserHistorysService,
    private readonly userRoomsService: UserRoomsService,
    private readonly userErrorsService: ErrorsService,
  ) {
  }

  private logger: Logger = new Logger('App Gateway');

  /**
  * Init websocket server
  * @name afterInit
  * @kind function
  * @property {Object}  server  - websocket server
```

```
  */
  public async afterInit(server: Server): Promise<any> {
    this.logger.log('Initialized');
  }

  /**
  * User connect
  * @name handleConnection
  * @kind function
  * @property {Object}  client  - ws client
  * @property {Object[]}  args  - data from user
  * @property {Object}  wss  - websocket server
  * @returns {Object} message for user
  */
  public async handleConnection(client: Socket, message: CreateConnectionDto, wss: Server) {
    const serverEvent = WsEnum.ServerConnect;
    const { error, user } = await this.userWsGuard(client, message, serverEvent);
    if(error) return;
    const allRooms = await this.userRoomsService.getAllUsersRoomsByUserId(user._id);
    const roomsId = allRooms.map(({ _id }) => _id.toString());
    for(const room of roomsId) {
      client.join(room);
      wss.to(room).emit(WsEnum.ClientConnect, {
        hasError: false,
        data: {
          user: user._id,
          serverEvent,
        }
      })
    }
  }

  /**
  * User disconnect
  * @name handleDisconnect
  * @kind function
  * @property {Object}  client  - ws client
  * @property {Object}  wss  - websocket server
  */
  public async handleDisconnect(client: Socket, message: CreateConnectionDto, wss: Server):
Promise<any> {
    const serverEvent = WsEnum.ServerDisconnect;
    const { error, user } = await this.userWsGuard(client, message, serverEvent);
```

```
  if(error) return;
  const allRooms = await this.userRoomsService.getAllUsersRoomsByUserId(user._id);
  const roomsId = allRooms.map(({ _id }) => _id.toString());
  for(const room of roomsId) {
    client.leave(room);
    wss.to(room).emit(WsEnum.ClientDisconnect, {
      hasError: false,
      data: {
        user: user._id,
        serverEvent,
      }
    });
  }
}

/**
* Error in on of the user device
* @name handleErrorDevice
* @kind function
* @property {Object}  client  - ws client
* @property {Object}  message  - data from user
* @property {Object}  wss  - websocket server
* @returns {Object} message for user
*/
public async handleErrorDevice(client: Socket, message: DeviceErrorDto, wss: Server) {
  const serverEvent = WsEnum.ServerErrorDevice;
  const { error, user } = await this.userWsGuard(client, message, serverEvent);
  if(error) return;
  const newError = await this.userErrorsService.create({
    userId: user._id,
    clientId: user.clientId,
    message: message.errorMessage,
  })
  client.emit(WsEnum.ClientErrorDevice, {
    hasError: false,
    data: {
      newError,
      serverEvent,
    }
  });
}

/**
```

```
  * Update user location for all users in the range
  * @name handleUpdateLocation
  * @kind function
  * @property {Object}  client  - ws client
  * @property {Object}  message  - data from user
  * @property {Object}  wss  - websocket server
  * @returns {Object} message for user
  */
  public async handleUpdateLocation(client: Socket, message: UserLocationTimeDto, wss: Server)
{
    const serverEvent = WsEnum.ServerUpdateLocation;
    const { error, user } = await this.userWsGuard(client, message, serverEvent);
    if(error) return;
    await this.userHistorysService.create({
      userId: user._id,
      clientId: user.clientId,
      lat: message.lat,
      lon: message.lon,
      time: message.time,
    })
    const allRooms = await this.userRoomsService.getAllUsersRoomsByUserId(user._id);
    const roomsId = allRooms.map(({ _id }) => _id.toString());
    for(const room of roomsId) {
      wss.to(room).emit(WsEnum.ClientUpdateLocation, {
        hasError: false,
        data: {
          user: user._id,
          location: {
            lat: message.lat,
            lon: message.lon,
          },
          serverEvent,
        }
      });
    }
  }

  /**
  * Create users room
  * @name handleCreateRoom
  * @kind function
  * @property {Object}  client  - ws client
  * @property {Object[]}  args  - data from user
```

```
  * @property {Object}  wss  - websocket server
  * @returns {Object} message for user
  */
    public async handleCreateRoom(client: Socket, message: CreateConnectionDto, wss: Server)
{
      const serverEvent = WsEnum.ServerCreateRoom;
      const { error, user } = await this.userWsGuard(client, message, serverEvent);
      if(error) return;
      const newRoom: any = await this.roomsService.create({ twoUsers: false });
      client.join(newRoom._id);
      client.emit(WsEnum.ClientCreateRoom, {
        hasError: false,
        data: {
          roomId: newRoom._id,
          serverEvent,
        }
      })
    }

  /**
  * Join for another room
  * @name handleJoinToRoom
  * @kind function
  * @property {Object}  client  - ws client
  * @property {Object[]}  args  - data from user
  * @property {Object}  wss  - websocket server
  * @returns {Object} message for user
  */
  public async handleJoinToRoom(client: Socket, message: JoinRoomDto, wss: Server) {
    const serverEvent = WsEnum.ServerJoinRoom;
    const { error, user } = await this.userWsGuard(client, message, serverEvent);
    if(error) return;
    const room: any = await this.roomsService.findById(message.roomId);
    if(!room) {
      client.emit(WsEnum.ClientError, {
        hasError: true,
        data: { message: EWsExceptionMessage.RoomNotFound, serverEvent }
      });
      return;
    }
    await this.userRoomsService.create({
      userId: user._id,
      roomId: room._id,
```

```
    clientId: user.clientId,
  })
  wss.to(message.roomId).emit(WsEnum.ClientConnect, {
    hasError: false,
    data: {
      user: user._id,
      serverEvent,
    }
  })
}

/**
* Leave from room
* @name handleLeaveFromRoom
* @kind function
* @property {Object}  client  - ws client
* @property {Object[]}  args  - data from user
* @property {Object}  wss  - websocket server
* @returns {Object} message for user
*/
public async handleLeaveFromRoom(client: Socket, message: JoinRoomDto, wss: Server) {
  const serverEvent = WsEnum.ServerJoinRoom;
  const { error, user } = await this.userWsGuard(client, message, serverEvent);
  if(error) return;
  const room: any = await this.roomsService.findById(message.roomId);
  if(!room) {
    client.emit(WsEnum.ClientError, {
      hasError: true,
      data: { message: EWsExceptionMessage.RoomNotFound, serverEvent }
    });
    return;
  }
  await this.userRoomsService.deleteByUserIdAndRoomId(
    user._id,
    room._id,
  )
  wss.to(message.roomId).emit(WsEnum.ClientDisconnect, {
    hasError: false,
    data: {
      user: user._id,
      serverEvent,
    }
  })
```

```
  }

  private async userWsGuard(client, message, serverEvent) {
    const existUser: any = await this.usersService.findByIdAndExternalId(message.userId,
message.externalId);
    if(!existUser) {
      client.emit(WsEnum.ClientError, {
        hasError: true,
        data: { message: EWsExceptionMessage.UserNotFound, serverEvent }
      });
      return { error: true };
    }
    return { error: false, user: existUser };
  }
}
```

mongo.service.mock.ts

```
import { Injectable } from '@nestjs/common';
import { ObjectId } from 'mongodb';

type DbType = { [key: string]: any[] };

@Injectable()
export class MongoService {
    mockDb: DbType = {
        clients: [
            {
                _id: '625aff2b0eb8b15a9867eb5d',
                adminCreated: '625afebe1e6f8eee5bf3528b',
                adminUpdated: '625afebe1e6f8eee5bf3528b',
                name: 'testClient',
                apiKey: '93704f8b-d536-4737-86ec-73f06631140c',
            }
        ],
        errors: [],
        location_steps: [],
        locations: [],
        rooms: [],
        super_admins: [
            {
                _id: '625afebe1e6f8eee5bf3528b',
                name: 'testAdmin',
                apiKey: '3b9cbc9a-56e5-4100-a359-eed7742be21a',
            }
```

```
    ],
    user_history: [],
    user_rooms: [],
    users: [],
    webhooks: [],
    webhooks_history: [],
};

startDb = JSON.parse(JSON.stringify(this.mockDb));

private count = 0;

public get clients() {
    return this.mockDb.clients;
}
public set clients(value) {
    this.mockDb.clients = value;
}

public get super_admins() {
    return this.mockDb.super_admins;
}
public set super_admins(value) {
    this.mockDb.super_admins = value;
}

public get errors() {
    return this.mockDb.errors;
}
public set errors(value) {
    this.mockDb.errors = value;
}

public get location_steps() {
    return this.mockDb.location_steps;
}
public set location_steps(value) {
    this.mockDb.location_steps = value;
}

public get locations() {
    return this.mockDb.locations;
}
```

```
public set locations(value) {
    this.mockDb.locations = value;
}

public get rooms() {
    return this.mockDb.rooms;
}
public set rooms(value) {
    this.mockDb.rooms = value;
}

public get user_history() {
    return this.mockDb.user_history;
}
public set user_history(value) {
    this.mockDb.user_history = value;
}

public get user_rooms() {
    return this.mockDb.user_rooms;
}
public set user_rooms(value) {
    this.mockDb.user_rooms = value;
}

public get webhooks() {
    return this.mockDb.webhooks;
}
public set webhooks(value) {
    this.mockDb.webhooks = value;
}

public get users() {
    return this.mockDb.users;
}
public set users(value) {
    this.mockDb.users = value;
}

public get webhooks_history() {
    return this.mockDb.webhooks_history;
}
public set webhooks_history(value) {
```

```
        this.mockDb.webhooks_history = value;
    }
    public clearDb() {
        this.mockDb = JSON.parse(JSON.stringify(this.startDb));
        this.count = 0;
    }
    public newId() {
        const start = '61ade95308cfdf9f43748fec';
        this.count++;
        return start.slice(0, start.length - this.count.toString().length) + this.count;
    }
}
```

locations.service.mock.ts

```
import { Injectable } from '@nestjs/common';
import { Locations, LocationsDocument, LocationsService as LocationsServiceReal } from '@db';
import { ObjectId } from 'mongodb';
import { MongoService } from '../mongo/mongo.service.mock';
import { Document } from 'mongoose';

@Injectable()
//@ts-ignore
export class LocationsService implements LocationsServiceReal {
    constructor(private mongo : MongoService){}
    findByDto(dto: any): Promise<Location & Document<any, any, any> & { _id: any; }> {
        return this.mongo.locations.find(doc => {
            const keys = Object.keys(dto);
            const condition = keys.reduce((acc, cur) => acc && dto[cur] === doc[cur], true);
            return condition;
        });
    }
    create(dto: Locations): Promise<LocationsDocument> {
        const collection = this.mongo.locations;
        const newDoc = {
            _id: this.mongo.newId(),
            ...dto,
        }
        collection.push(newDoc);
        this.mongo.locations = collection;
        return newDoc as any;
    }
}
```

location.service.spec.ts

```
 import { MongoService } from '@mock/mongo/mongo.service.mock';
import { CreateUserHistoryDto, GetLocationDto, GetUserHistorysDto, CoreService } from '@common';
import { clientReq } from '@mock/auth/auth-mocks';
import { LocationService } from '@location/location.service';
import { LocationsService as LocationsServiceMock } from '@mock/locations/locations.service.mock';
import { LocationStepsService } from '@mock/location-steps/location-steps.service.mock';
import { UserHistorysService } from '@mock/user-history/user-historys.service.mock';
import { UsersService } from '@mock/users/users.service.mock';
import { AlgorithmsService } from '@mock/algorithms/algorithms.service';
import { LocationStrategyService } from '@mock/location-strategy/location-strategy.service';
import { BipartiteSubsetsService } from '@mock/algorithms/bipartite-subset.service';
import { BingStrategyService } from '@mock/location-strategy/bing-strategy.service';
import { OpenRouteStrategyService } from '@mock/location-strategy/openroute-strategy.service';
import { ETransports } from '@db';

describe('LocationService', () => {
    const mongo: MongoService = new MongoService();
    let locationService: LocationService = new LocationService(
      new LocationsServiceMock(mongo) as any,
      new LocationStepsService(mongo) as any,
      new UserHistorysService(mongo) as any,
      new UsersService(mongo) as any,
      new CoreService(),
      new AlgorithmsService(
        new BipartiteSubsetsService()
      ) as any,
      new LocationStrategyService(
        new BingStrategyService(),
        new OpenRouteStrategyService(),
      ) as any,
   );

    describe('LocationService methods', () => {
      it('fixUserLocation', async () => {
        mongo.users = [
          {
            _id: 'uniqueUserId',
            clientId: clientReq.client.document._id,
            name: 'user1'
```

```
    }
  ]

  const newDto = new CreateUserHistoryDto();
  newDto.lat = '10.00';
  newDto.lon = '11';
  newDto.time = 111111111;
  newDto.userId = 'uniqueUserId';
  const data = await locationService.fixUserLocation(clientReq, newDto);
  expect(data).toEqual({
    _id: '61ade95308cfdf9f43748fe1',
    lat: '10.00',
    lon: '11',
    time: 111111111,
    userId: 'uniqueUserId',
    clientId: '625aff2b0eb8b15a9867eb5d'
  });
  mongo.clearDb();
})

it('getRoute', async () => {
  mongo.users = [
    {
      _id: 'uniqueUserId',
      clientId: clientReq.client.document._id,
      name: 'user1'
    }
  ]

  const query = new GetLocationDto();
  query.endLat = '11.11';
  query.endLon = '12.13';
  query.startLat = '13.14';
  query.startLon = '14.15';
  query.transport = ETransports.Walking;
  query.userId = 'uniqueUserId';
  const data = await locationService.getRoute(clientReq, query);
  expect(data).toEqual([]);
  mongo.clearDb();
})

it('getUserLocation', async () => {
  mongo.users = [
```

```
        {
          _id: 'uniqueUserId',
          clientId: clientReq.client.document._id,
          name: 'user1'
        }
      ]

      const newDto = new CreateUserHistoryDto();
      newDto.lat = '10.00';
      newDto.lon = '11';
      newDto.time = 111111111;
      newDto.userId = 'uniqueUserId';
      const newLocation = await locationService.fixUserLocation(clientReq, newDto);

      const query = new GetUserHistorysDto();
      const data = await locationService.getUserLocation(clientReq, query);
      expect(data).toEqual([
        {
          _id: '61ade95308cfdf9f43748fe1',
          lat: '10.00',
          lon: '11',
          time: 111111111,
          userId: 'uniqueUserId',
          clientId: '625aff2b0eb8b15a9867eb5d'
        }
      ]);
      mongo.clearDb();
    })
  })
})
```

testServer.js

```
const http = require('http');
const { parse } = require('querystring');

const hostname = 'localhost';
const port = 3001;

const server = http.createServer((req, res) => {
  if(req.method.toUpperCase() === 'POST' && req.url === '/getWebhook'){
      let body = '';
      req.on('data', function (data) {
        body += data;
      });
```

```
      req.on('end', function () {
        try {
          req.body = JSON.parse(body);
          console.log(req.body);
          res.statusCode = 200;
          res.setHeader('Content-Type', 'text/plain');
          res.end('ok');
        } catch (e) {
          req.body = parse(body);
        }
      });
  }else {
    res.statusCode = 404;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Not found');
  }
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

location.controller.ts

```
import { Controller, Post, UseGuards, HttpCode, Get, Req, Query, Body } from '@nestjs/common';
import { LocationService } from './location.service';
import {
  ApiKeyAuthGuard,
  Keys,
  CreateUserHistoryDto,
  GetLocationDto,
  GetUserHistorysDto,
  LocationStepsDto,
} from '@common';
import { ApiOperation, ApiResponse, ApiTags } from '@nestjs/swagger';
import { UserHistorys } from '@db';

/**
* Location controller
* @name LocationController
* @kind class
*/
@ApiTags('User location API')
@Keys()
@UseGuards(ApiKeyAuthGuard)
```

```
@Controller('location')
export class LocationController {
  constructor(private readonly service: LocationService) {}

  /**
  * Save user location in database
  * @name fixUserLocation
  * @kind event
  * @property {Object}  req  - req object
  * @property {Object}  dto  - user history create dto
  * @returns {Object} UserHistory document
  */
  @ApiOperation({ summary: 'Save user current location' })
  @ApiResponse({ status: 201, type: () => UserHistorys })
  @HttpCode(201)
  @Post()
  fixUserLocation(
    @Req() req,
    @Body() dto: CreateUserHistoryDto,
  ): Promise<UserHistorys> {
    return this.service.fixUserLocation(req, dto);
  }

  /**
  * Get user location history
  * @name getUserLocation
  * @kind event
  * @property {Object}  query  - query object dto
  * @property {Object}  req  - req object
  * @returns {Object} UserHistory documents
  */
   @ApiOperation({ summary: 'Get users locations' })
   @ApiResponse({ status: 200, type: () => UserHistorys, isArray: true })
   @HttpCode(200)
   @Get()
   getUserLocation(
      @Req() req,
      @Query() query: GetUserHistorysDto,
   ): Promise<UserHistorys[]> {
     return this.service.getUserLocation(req, query);
   }

  /**
```

```
  * Get rout between locations
  * @name getRoute
  * @kind event
  * @property {Object}  query  - query object dto
  * @property {Object}  req  - req object
  * @returns {Object} UserHistory documents
  */
   @ApiOperation({ summary: 'Get rout between locations' })
   @ApiResponse({ status: 200, type: () => LocationStepsDto, isArray: true })
   @HttpCode(200)
   @Get('route')
   getRoute(
      @Req() req,
      @Query() query: GetLocationDto,
   ): Promise<LocationStepsDto[]> {
     return this.service.getRoute(req, query);
   }
}
```