Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Програмно-технічний засіб моніторингу трафіку в кіберфізичній системі «Розумне місто»
Назва теми

КвРКІ. 180105.18.01.05 ПЗ
Шифр

Галузь знань 12 «Інформаційні технології»
Шифр, назва

Спеціальність 123 «Комп'ютерна інженерія»
Шифр, назва

Освітня програма «Комп'ютерна інженерія»
Назва

Виконав: студент IV курсу, група КІ–18–1 _____ В.О. Дідух
Підпис Ініціали, прізвище

Керівник _____ Т.М. Кисіль
Підпис, дата Ініціали, прізвище

Нормоконтролер _____ С.М. Лисенко
Підпис, дата Ініціали, прізвище

До захисту допускаю:
Зав. кафедри комп'ютерної інженерії та інформаційних систем _____ Т.О. Говорущенко
Підпис Ініціали, прізвище

«15» червня 2022 р.

Хмельницький 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Освітній рівень бакалавр

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма Освітня програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

"11" 01 2022 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Дідуху Владиславу Олександровичу
Прізвище, ім'я, по батькові

Тема проекту (роботи) Програмно-технічний засіб моніторингу трафіку в кіберфізичній системі «Розумне місто»

Керівник проекту (роботи) Кисіль Т.М., к.ф-м.н., доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 16.11.2021 р. № 11

2. Строк подання студентом проекту (роботи) на кафедру 17.06.2022р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Дослідження предметної області та постановка задачі

Методологія та програмно-апаратне забезпечення

Програмно-апаратна реалізація системи моніторингу

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Інтерфейс застосунку

Блок-схема роботи застосунку

Блок-схема вичислення позиції маршруту зі сторони серверу

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Лисенко С.М., професор кафедри КІІС		
Антиплагіат	Нічепорук А.О., доцент кафедри КІІС		

7. Дата видачі завдання «_16_»_____11_____2021 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	11.01.2022	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2022	виконано
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2022	виконано
4	Робота над розділом 2 – методологія та програмно апаратне забезпечення	01.04.2022	виконано
5	Робота над розділом 3 - програмно-апаратна реалізація системи моніторингу	01.05.2022	виконано
6	Оформлення пояснювальної записки згідно вимог	20.05.2022	виконано
7	Попередній захист ВКР	24.05.2022	виконано
8	Захист ВКР на засіданні ЕК	Червень 2022	

Студент _____ В.О. Дідух
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ Т.М. Кисіль
Підпис Ініціали, прізвище

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 180105.18.01.05 ПЗ	Пояснювальна записка	58		
			Графічні матеріали			
2		КвРКІ 180105.18.01.05 Е8	Інтерфейс застосунку	1		
3		КвРКІ 180105.18.01.05 Е8	Блок-схема роботи	1		
			застосунку			
4		КвРКІ 180105.18.01.05 Е8	Блок-схема вичислення	1		
			позиції маршруту зі			
			сторони серверу			

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 180105.18.01.05 ВП		
Розробив		Ділух			Відомість проекту	Літера	Аркуш	Аркушів
Перевір.		Кисіль				У	1	1
Н. контр.		Лисенко				ХНУ, КІ-18-1		
Затв.		Говорушенко						

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмно-технічний засіб моніторингу трафіку в кіберфізичній системі «Розумне місто»».

Автор роботи: Дідух Владислав Олександрович.

Керівник роботи: Кисіль Тетяна Миколаївна.

Пояснювальна записка: 58 с., 58 рис., 1 табл., 4 дод., 26 джерела.

Графічна частина: 7 презентаційних слайдів.

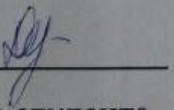
МОНІТОРИНГ ТРАФІКУ, РОЗУМНЕ МІСТО, СЕРВЕР, КЛІЄНТ, БАЗА ДАНИХ.

Мета роботи – створення засобу для моніторингу трафіку в кіберфізичній системі «Розумне місто».

Об'єктом дослідження є комплекс програмних засобів: сервер для реалізації моніторингу та клієнтська частина для візуалізації даних.

Предметом дослідження є формалізований опис, архітектура та технології для реалізації моніторингу.

Практичне значення має спроєктований та розроблений сервер, котрий емулює трафік та передає інформацію клієнтові, підхід до реалізації серверу з використанням нових технологій.



Підпис студента

01.06.2022

Дата

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

JS – JavaScript

CSS – Cascading Style Sheets

SPA – Single Page Application

JSS – JavaScript Style Sheets

SASS – Syntactically Awesome Style Sheets

SCSS – Sassy Cascading Style Sheets

VSN – Vehicular Sensor Network

API – Application Programming Interface

REST – Representational State Transfer

БД – База Даних

JSON – JavaScript Object Notation

HOC – High Order Component

UI – User Interface

UX – User Experience

ВСТУП

Ріст населення в будь-якому населеному пункті породжує нові потреби, котрі потрібно задовольнити. Зі сторони представників місцевої влади запроваджуються нові проекти, мета яких покращити інфраструктуру міста: відкриваються нові навчальні заклади, лікарні, будується нова нерухомість для проживання людей. Покращується урбаністика за рахунок розширення велосипедних доріжок, поліпшення стану доріг. Бізнес також грає свою роль в цьому циклі розвитку рівня життя людей й розвиває класичні галузі споживання та обслуговування (торгові центри, супермаркети, ресторани), намагається пропонувати все більш різноманітний спектр сервісів. Чи достатньо цього всього для дійсно комфортного життя в абстрактному «місті майбутнього»? Ні. Розвиток інформаційних технологій відкрив неймовірний потенціал, котрий поліпшив та пришвидшив процеси ринку послуг, зміг тісно інтегрувати людину в динамічний навколишній світ: здається, ніби абсолютно все у тебе під рукою, а точніше, весь світ зміг вміститися в твоєму ґаджеті.

Міська інфраструктура не повинна стояти осторонь й також користуватись благами ІТ індустрії. Що ж можна покращити діджиталізацією? Перше, що приходить – це громадський транспорт.

Громадський транспорт – це важливий стовп міського життя. Завдяки ньому відбувається переміщення громадян від точки А в точку Б. Це можна порівняти з кровоносними судинами: робітники переміщаються по вузлам міського транспорту від домівок до свого робочого місця, де вони рухають економічні процеси та грошові обороти.

Враховуючи сказане вище, оптимізація та покращення «інституту» громадського транспорту це необхідність будь-якого міста. А «розумне місто» робитиме це зі залученням ІТ.

З сучасними можливостями уже недостатньо щоб громадський транспорт приїзжав в точку призначення секунда в секунду. Завдяки розвитку ІТ, «розумне місто» може (й повинне) надавати повну інформацію про конкретний рейс:

можливість відслідковувати в реальному часі на карті, мати змогу дивитись яким він маршрутом їде. Подібні послуги значно покращать зручність громадського транспорту міста.

Отже, моніторинг трафіку це фундаментальний атрибут «Розумного міста» майбутнього. Подібний механізм потребує ефективного контролю динамічного потоку даних стосовно трафіку, а значить і ретельного планування. Подібна задача містить два виклики: з однієї сторони, потрібно розглянути всі можливі варіанти для збору та аналізу даних, вибрати найбільш оптимальний з точки зору коефіцієнту ціна-ефективність. З другої сторони – потрібно врахувати, що цільова аудиторія подібного сервісу це користувачі смартфонів, а значить, користувачі мобільної мережі інтернет. Для хорошої швидкодії та стабільності потрібно побудувати потік даних між сервером та клієнтом з мінімальним використанням об’єму трафіку.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз задачі, обґрунтування вибору моделі життєвого циклу для реалізації проекту

Моніторинг трафіку в системі «розумне місто» передбачає можливість відслідковування в реальному часі місцеположення потрібного для користувача рейсу, перегляд маршруту в цілому.

Традиційні шляхи та підходи моніторингу трафіку опираються на міську інфраструктуру та її ресурси для отримання інформації про топографію міста. Наприклад, дані про потік трафіку доволі часто вираховуються датчиками вбудованими прямо в тротуар. Альтернативно, інформація збирається камерами, котрі розміщені на важливих, «стратегічних» точках міста. Тим не менш, подібні методи можуть виявитись зовсім нерентабельними й стати справжньою катастрофою з фінансової точки зору, хоч і здаються найбільш надійними із-за свого високотехнологічного підґрунтя. Тому вартує розглянути й інші варіанти.

Можливим варіантом є свого роду співпраця між представниками приватних перевізників та владою міста: залучити весь міський транспорт у використанні датчиків, котрі передаватимуть дані в реальному часі про своє місцеположення через GPS сенсори, завантаженість (кількість вільних місць). Завдяки цьому можна досягти гнучкості в моніторингу трафіку, так як така система легко розширюватиметься і в майбутньому можна передавати інформацію не тільки про сам транспорт, а й про те що його оточує: світлофори, затори і т.д.

Схожою альтернативою є використання більш просунутих технологій та методів, як Vehicular Sensor Networks (VSN). Суть такого підходу заключається в тому, що роль мобільних датчиків виконують транспортні засоби, оснащені бортовими блоками. Такий транспорт пересувається по місту та збирає інформацію для визначення умов руху і, в решті решт, формують централізовану

матрицю руху. Планування шляху для таких транспортних засобів, котрі містять зондування, відповідно до кореляції вибірки та похибки, вирішується проблема нерівномірної вибірки, котра була спричинена використанням громадського транспортного засобу. Для відновлення даних без вибірки, метод оцінки, який основується на заповненні матриці, можна застосувати до однорідної матриці трафіку з випадковою вибіркою [21].

Найбільш екстраординарним методом можна назвати метод з просунутою технологією використання БПЛА для моніторингу в реальному часі. Оскільки БПЛА орудує в небі, то цей механізм ніяк не залежить від будь яких умов трафіку чи інцидентів на дорозі. Але такий підхід вимагає великих інвестицій і, в решті решт, має всі шанси виявитись занадто дорогим та нерентабельним.

Доволі сучасним та прогресним можна назвати метод Network Tomography. Цей метод можна вважати ідеальним, якщо виключаються будь-які види колаборації між представниками влади міста та представниками бізнесу (підприємців), які надають послуги з перевезення. Причини для відмови подібної співпраці можуть бути абсолютно різні. В тому числі, й сугубо організаційні: кожен транспорт повинен містити датчик та виступати пасивним зондом для збору інформації. Сам метод мережевої томографії повністю покладається на себе, збір інформації самодостатній та не потребує додаткових домовленостей з третіми обличчями. Мережева томографія покладається на визначенні константних (постійних) зв’язків [21]. Це можна легко реалізувати в будь-якій мережі з кількома (а саме, з трьома і більше) межами підключення за допомогою одного монітора чи датчика, який циклічно оновлюється для збору інформації. Задіяний алгоритм може виглядати наступним чином: обчислюється контрольний набір крайніх точок для кожного вузла за можливими протоколами маршрутизації. Далі алгоритм відбирає вузли для утворення підмножини вузлів зондування, котрі були розглянуті як перспективний варіант для отримання покриття краю в даній мережі. Замість дослідження вибору вузлів, актуальним є вибір набору шляхів, які були позитивно оцінені відповідно до рангу лінійної системи при наявності

проблем та збоїв у системі. Мережева томографія, в своїх перших концептах, була створена для комунікаційних мереж як Інтернет. Основна задача – збір інформації, по якій можна зробити висновок про стан внутрішньої мережі за допомогою різних наскрізних обчислень та вимірювань датчиками, які розміщуються на краю мережі. Сама мережа представляє із себе неорієнтований граф

$$G = (V, E), \tag{1.1}$$

де V – це множина вузлів, а E – множина з'єднань. Певні вузли вибираються в якості датчиків і обмінюються інформацією про зондування та збирають наскрізні вимірювання метриків, які необхідні для використання. У прикладі з мережею Інтернет, це може бути загальна затримка чи звіт про втрати пакетів між двома датчиками. На рисунку 1.1 вузли датчиків позначені колами, напівжирні лінії підкреслюють три з можливих шляхів між парами аналогічних датчиків. Конкретніше, $p_1(m_1, m_3) = \{ l_1, l_2 \}$, $p_2(m_1, m_9) = \{ l_1, l_{15} \}$, $p_3(m_9, m_3) = \{ l_{15}, l_7 \}$.

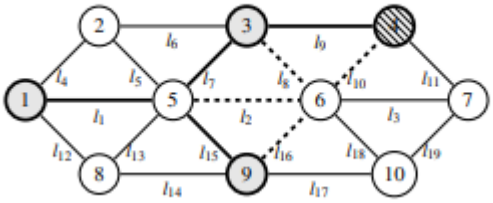


Рисунок 1.1 – Зразок мережі з 10 вузлами та 19 зв'язками

$$\begin{cases} l_1 + l_7 + l_9 = b_1 \\ l_1 + l_2 + l_8 + l_9 = b_2 \\ l_1 + l_{15} = b_3 \\ l_{15} + l_7 + l_9 = b_4 \\ l_{16} + l_8 + l_9 = b_5 \end{cases}, \tag{1.2}$$

$$R = \begin{bmatrix} 1000001010000000000 \\ 1100000110000000000 \\ 1000000000000010000 \\ 0000001010000010000 \\ 0000000110000001000 \end{bmatrix}. \quad (1.3)$$

Основним тезисом є залежна природа метриків, яку потрібно обгрунтувати [21]. Наприклад, затримка наскрізного зовнішнього шляху є сумою затримок його зв'язків. Якщо надати можливість b_i розпізнавати вимірювання, яке було отримане при зондуванні шляху i та x_j, та при затримці на зв'язку j^{th}, всі вимірювання на шляху між датчиками m_1 та m_9 на рисунку 1.1 можна виразити як

$$x_1 + x_{15} = b_1. \quad (1.4)$$

В цілому, допустимо, що P = { p_1, p_2, …, $p_{|P|}$ } являє собою набір шляхів зондування між датчиками. Відношення між шляхами та зв'язками представлено бінарною матрицею R. Розмірами цієї матриці є |P| × |E|, де кожен ряд відноситься до певного шляху зондування. Конкретніше, елемент (i, j) в матриці R прирівняний до 1, якщо зв'язок l_j відноситься до шляху pi, або, в інакшому випадку, прирівнюється до 0. Наскрізні вимірювання зберігаються в 1 × |P| вектор b, елементи b_i представляють наскрізні вимірювання на шляхові pi. Якщо x_j представляє затримку по шляху l_j, то загальна затримка на шляху pi виражена як

$$\sum_{j=1}^{|E|} r_{ij} x_j = b_i, \quad (1.5)$$

Ця формула легко розширюється по всім шляхам, тому можна сформулювати лінійну систему

$$Rx = b, \quad (1.6)$$

де x являє собою вимірюванням індивідуальних зв'язків.

На рисунку 1.1, вузли 1 та 9 є датчиками. Додаючи вузол 3, рішення відповідної лінійної системи повністю визначають час поїздки для всіх залучених зв'язків. Вузол 5 не є датчиком, тому для нього не потрібно розміщати камеру на відповідному перехресті доріг. Важливо пам'ятати, що інакший вибір датчиків, котрі охоплюють більшу частину мережі, не завжди може покращити ідентифікацію. Наприклад, з датчиками { 1, 4, 9 } та шляхами $p_1(m_1, m_4) = \{ l_1, l_7, l_9 \}$, $p_2(m_1, m_9) = \{ l_1, l_{15} \}$, й $p_3(m_9, m_4) = \{ l_{15}, l_7, l_9 \}$, система не може використати унікальне рішення для зв'язків l_7 та l_9. Розгляд додаткових шляхів також не є гарантом покращення ідентифікації [21].

Нарешті, останнє рішення, котре може підійти для вирішення поставленої задачі – це використання технологій націлених на використання скупчень людей для збору даних для аналізу трафіку. Наприклад, використання смартфонів як датчиків для збору потрібної інформації. В такому випадку, моніторинг трафіку залежить від уже існуючої інфраструктури підтримки мобільного зв'язку. Такий підхід вимагає збір даних про геолокацію користувача. З технічної сторони, основною проблемою такого підходу виступатиме поганий зв'язок та лаги. З побічних факторів, проблемою буде можливий ріст цін на мобільний зв'язок (а він обов'язково настане, бо на якісь гроші потрібно проводити подібні нововведення), та питання кондиційності інформації для користувача. В обов'язковому порядку, така ініціатива повинна бути на добровільній основі серед користувачів, тому потрібна хороша реклама та агітаційна кампанія для залучення якомога більше людей в цій програмі.

Проаналізувавши всі можливі варіанти та підходи, для реалізації дипломної роботи буде використано саме метод мобільних датчиків на громадському транспорті, котрий при пінгуванні передаватиме інформацію про цей транспорт. Це традиційний метод, котрий розраховує та спирається на отриманій інформації «з перших рук», тому й вважається найбільш надійним та ефективним. Метод моніторингу трафіку зі залученням БПЛА потребує багато фінансів, навіть для етапу тестування, котрий не факт, що покаже себе добре, та, в решті решт,

окупиться в перспективі. Метод з застосування користувачів-добровольців здається також малоефективним та також затратним: потрібна агітація та постійна мотивація для користувачів у подібній співпраці. Підхід VSN добре підходить для моніторингу заторів та збору геолокаційних даних (як онлайн мапи, на кшталт Google Maps), але цей метод не зовсім доречний для збору інформації стосовно трафіку, а саме, координати положення в даний момент громадського транспорту. Мобільні датчики на громадському транспорті здається найбільш виграшною позицією.

По перше, з приватними підприємствами набагато простіше домовитись ніж з кожним окремим користувачем, якщо порівнювати цей метод з методом залучення смартфонів жителів міста.

По друге, це найбільш виграшний варіант з точки зору ефективність-ціна. Варіант з VSN та БПЛА програють, так як вартує також враховувати ціну на утримання такої апаратури. Згадуючи метод мобільних датчиків, можна провести з парадигмою Об’єктно Орієнтованим Програмуванням, в якому принцип побудови ієрархії класів базується на максимальній незалежності класів та концентрації на власній ролі і ні на чому більше (Принцип Єдиної Відповідальності).

Навіщо потрібні сторонні пристрої, як БПЛА, смартфони, чи VSN-транспорт, які віддаватимуть інформацію про громадський транспорт, якщо можна напряму «звернутись» до цього громадського транспорту з такою самою метою.

1.2 План реалізації проекту

Оскільки данні про місцеположення громадського транспорту ніде не надаються у відкритому доступі, то доведеться власноруч написати моки й передавати їх через власноруч написаний backend, ніби це реальні дані. Іншими словами, доведеться емулювати дані.

Важливою приміткою є те, що фактично застосунок являється емуляцією, й для того, щоб максимально приблизитись до реальності буде використовуватись база даних. При ініціалізації сервера, моки будуть записуватись в базу даних. Альтернативно, можна було би обійтися й без БД, просто зберігаючи моки в окремому модулі й звернення до них резолверами, котрі приймають GET запити. Тим не менш, ціль проекту приблизити емуляцію максимально до реальної архітектури.

Тобто, ініціалізація серверу виглядатиме наступним чином:

1) запуск сервера, підключення до БД;
2) при успішній ініціалізації, моки записуються в БД;
3) при успішному записі, сервер готовий обробляти запити. Данні братиме з БД.

Тому, завдання можна поділити на дві частини: розробка серверу та бази даних (backend), та розробка інтерфейсу користувача (frontend).

Зі сторони backend частини, план реалізації включає:

1) розробка моків;
2) організація бази даних;
3) написання резолверів, котрі оброблятимуть GET запити зі сторони фронт-частини;
4) тестування та debug.

Що стосується frontend частини, якщо розглядати REST, то застосунок включатиме тільки GET запити так як кінцевий користувач ніяк не може модифікувати данні.

Ціль проекту – тільки моніторинг даних, ніяких мутацій.

Весь інтерфейс представлятиме із себе приблизно наступну схему як на рисунку 1.2:

1) полотно на якому буде розміщена карта;
2) кнопка для відкриття модального вікна;

3) модальне вікно, в якому можна вибрати, або навпаки, приховати відображення певного маршруту;

4) кнопка, котра відкриє маню налаштувань застосунку.

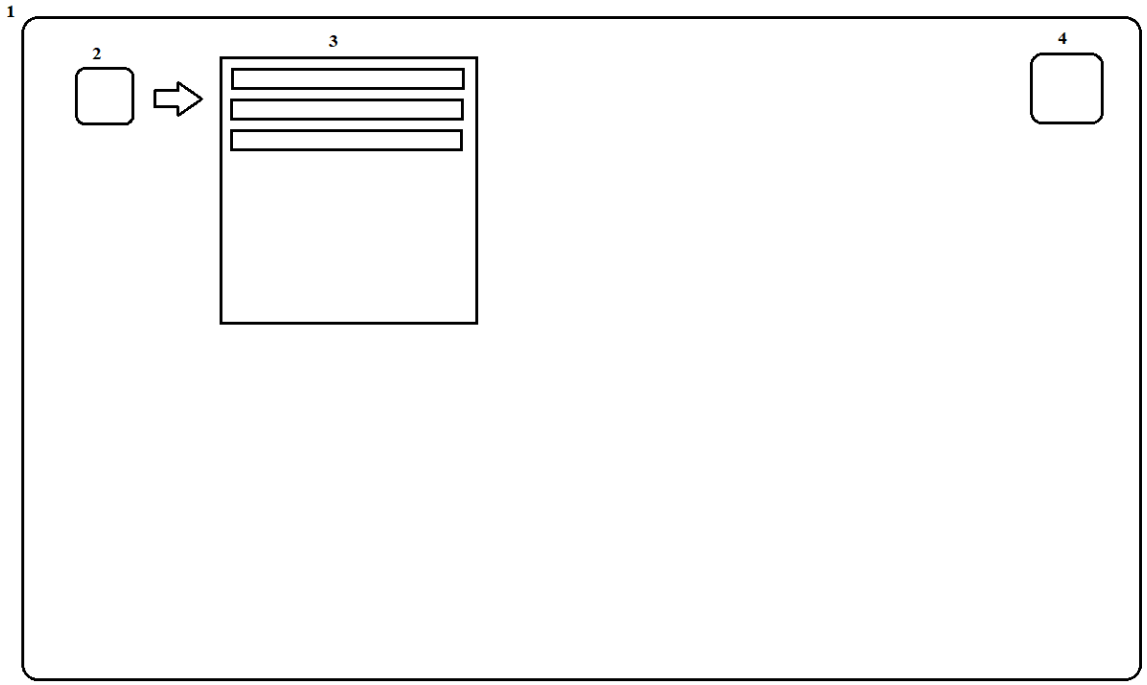


Рисунок 1.2 – Схема застосунку

Кроки реалізації проекту зі сторони фронт-частини:

1) розробка frontend частини згідно запланованого макету;
2) тестування та debug.

1.3 Висновки

В цьому розділі було розглянуто різні підходи до реалізації системи моніторингу трафіку, описано базові принципи функціонування проекту та продумано цикл розробки.

2 МЕТОДОЛОГІЯ ТА ПРОГРАМНО-АПАРАТНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Огляд технологій

Проект для дипломної роботи використовуватиме різні сучасні технології для розробки, котрі повинні покращити якість сервісу та полегшити процес розробки. В цьому розділі буде обгрунтовано їхнє використання.

Для написання backend частини, буде використовуватись платформа Node.js разом з фреймворком Express. База даних – MongoDB. Замість звичайного REST використаємо GraphQL.

Для написання frontend частини, буде використовуватись JavaScript бібліотека для створення користувацьких інтерфейсів – React. Також, в якості додаткового інструменту, буде використано бібліотеку Material UI для стилізації інтерфейсу.

2.2 Node.js

Node.js – це платформа, котра дозволяє мові програмування JavaScript взаємодіяти з пристроями вводу-виводу через власне API. Node.js вирішує проблему «ізольованості» мови програмування JavaScript, так як вона могла взаємодіяти тільки з браузерним оточенням. Завдяки цій платформі, за допомогою JavaScript тепер можна писати не тільки інтерфейси користувача в браузері, а й десктопні застосунки й, як у нашому випадку, сервери [18]. Серед плюсів Node.js – це кросплатформність, величезна спільнота JavaScript програмістів, які підтримують цю технологію та вже створили безліч безкоштовних інструментів, котрі вирішують різні проблеми та задачі [5]. Головна перевага Node.js – це простота використання. Також, це дуже легкий поріг входу – frontend програмісти можуть легко навчитись роботі з технологією Node.js оскільки вона використовує таку самоу мову програмування, а саме, JavaScript . Проект не потребує рішення

для гігантських ентерпрайз проектів, так як він невеликих розмірів. Не вартує перевантажувати застосунок недоречними та тяжкими рішеннями.

Для того щоб дізнатись як працює Node.js, вартує розглянути принципи роботи JavaScript. Суть в тому, що сам браузер компілює код JavaScript в машинний код й таким чином виконує сценарії та скріпти, написані на цій мові програмування. Тим не менш, браузер не мав змогу якось взаємодіяти з файлами на ПК користувачів, тому можливості JavaScript не виходили за межі самого браузера. З плином часу рушії браузерів вдосконалювались й 2 вересня 2008-го року данське відділення Google випустило перший реліз рушія V8. Рушій пропонував компіляцію JavaScript в машинний код, уникаючи етап проміжного байт коду; ефективне керування пам’яттю рушія (сам JavaScript не має можливості прямого керування пам’яттю, як це уміє, скажімо, мова програмування C) та оптимізація прибирання сміття (визволення пам’яті); підвищення оптимізації та швидкодії браузера [18]. Випуск V8 – це знакова подія в Web розробці, бо V8 став частиною Chromium, на основі якого, засновані безліч веб-оглядачів. Іншими словами, це передумова стандартизації, усуненню хаосу в можливостях JavaScript: різні браузери по різному слідували стандартам EcmaScript, що значило, що певні мови програмування могли працювати в одних браузерах, а в інших ні. В такому випадку були потрібні поліфіли. Сам V8 завжди підтримував самі останні новинки стандартів EcmaScript.

У випадку з Node.js, випуск V8 – це також передумова створення вище цієї платформи.

Історія Node.js починається в 2009 році. Перша версія була написана програмним інженером Раяном Далом, котрий критикував можливості популярного на той час вебсервера Apache HTTP Server. Критиці підлягав факт того, що код цього застосунку блокував весь процес при обробці асинхронних з'єднань, або залучав декілька стеків виконання у разі одночасного з’єднання. Node.js був представлений на європейській конференції JSConf, котра відбулася 8 листопада 2009 року [18]. Ключові характеристики Node.js – поєднання рушія V8

для JavaScript коду, цикл обробки подій на основі C-бібліотек і низькорівневий API для команд вводу-виводу. Серед важливих подій у життєвому циклі Node.js можна згадати створення менеджеру пакетів npm для Node.js в 2010 році. Нині, цей менеджер є одним із головних та невід'ємних стовпів розробки на мові JavaScript. npm полегшує публікацію, поширення та обмін бібліотек і спрощує встановлення, оновлення й видалення цих же бібліотек.

2.3 Express

Express – це фремфорк для серверних застосунків Node.js. Дуже популярний інструмент, який серед спільноти програмістів уже став стандартом, один із китів різних стеків технологій. Мінімалістичний, легковажний та простий. Цей фреймворк являється, де-факто, каркасом для Node.js, який спрощує написання CRUD (Create, Read, Update, Delete) операцій [8].

Головна ціль фреймворку Express – полегшення роботи з обробкою запитів на сервер. Наприклад, на рисунку 1.2 виглядає «нативна» обробка запитів на Node.js.

Спершу ініціюється сервер на порту 3000. Всі запити, які приходитимуть на цей порт оброблятимуться колбек-функцією, яка приходить як аргумент для createServer нативного пакету http.

Тепер потрібно перевіряти який метод прийшов на сервер (GET/POST) й, відповідно, виконувати логіку, котра запланована для цих запитів, враховуючи параметри, які прийшли зі запитом. Наприклад, для GET запитів потрібно отримати ці параметри в request.url.

З POST запитами все складніше: данні приходять чанками асинхронно, тому доводитись збирати їх поетапно в якусь глобальну змінну [2].

На рисунку 2.1, всі чанки записуються в змінну body.

```
const http = require('http');

http.createServer((request, response) => {
    switch(request.method) {
        case 'GET': {
            //! Обробляємо GET запит
            const urlRequest = parse(request.url);

            if (urlRequest.query.test) {
              //! GET параметр "test"
            }
        }
        case 'POST': {
            //! Обробляємо POST запит
            let body = '';

            request.on('data', chunk => body += chunk.toString())
            request.on('end', () => {
              const params = parse(body);

              if (params.test) {
                //! POST параметр "test"
              }
            })
        }
        default: {
          console.log('unexpected method')
        }
    }
}).listen(3000)
```

Рисунок 2.1 – Нативна обробка запитів в Node.js

На рисунку 2.2 зображено подібні операції значно простіші завдяки Express:

Завдяки зручному API фреймворку Express все виглядає куди очевидніше та простіше.

Завдяки функціям get та post, котрі слухають задані ендпойнти, код стає читаємим та зрозумілим. Тепер не потрібно парсити данні або збирати їх з чанків. Тепер все доступно з потрібних полів об’єкту request.

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

app.listen(3000, function() {
  console.log('listening on 3000')
})

app.get('/products/:productId', (request, response) => {
    const testValue = request.params.productId;
    //! GET параметр "products"
})

app.post("/products", (request, response) => {
    const body = request.body;
    //! GET параметр для ендпойнту '/products'
});
```

Рисунок 2.2 – GET/POST запити в Express

2.4 MongoDB

MongoDB – класична NoSQL система управління базою даних. Дуже популярна система, аналогічно фреймворку Express, є одним із китів стеків технологій. Вибір для застосунку дипломної роботи також обумовлений простотою та легкістю рішення для малих та середніх проектів [9].

Основні можливості MongoDB:

1) повна підтримка індексів;
2) динамічні запити;
3) документо-орієнтоване сховище;
4) гнучка мова для формування запитів;
5) трекінг операцій, що змінюють данні в БД;
6) підтримка відмовостійкості й масштабованості: асинхронна реплікація, набір реплік;
7) профілювання запитів;

8) ефективне зберігання бінарних даних великих розмірів як фото і відео;
9) JSON-подібна схема даних;
10) кросплатформеність.

Величезним плюсом MongoDB, який вартує зазначити, є дуже зручний графічний клієнт MongoDB Compass. Він інтуїтивно зрозумілий та простий, дозволяє створювати або видаляти колекції, додання, перегляд, зміну й видалення документів. Набагато зручніше рішення ніж стандартний моніторинг через консоль.

2.5 REST vs GraphQL

REST – це підхід до створення архітектури для мережевих протоколів. Фундаментом цього підходу є принципи функціонування Всесвітньої павутини, принципи роботи протоколу HTTP. Вартує пам’ятати, REST описаний та розповсюджений програмістом Роєм Філдінгом – одним із творців HTTP [22]. REST охарактеризовується наступними словами: як клієнт повинен спілкуватись з сервером. За цим тезисом слідують підпункти:

- яким способом клієнт звернеться до серверу;
- що клієнт передає на сервер;
- що хоче бачити клієнт в якості відповіді;

Опираючись на ці пункти й будується RESTful архітектура.

«Яким способом клієнт звернеться до серверу?» - можна перефразувати для більшої точності на «Куди?» та «Як?». Мова йдеться про HTTP-методи та ендпойнти. Концепція ендпойнтів відповідає на питання «Куди?», а методи – «Як?». Ендпойнт – це URL, до якого звертається клієнт. Посмію порівняти з адресом вашого будинку, куди листоноша повинен доставити пакунок.

```
await fetch('https://api.example.com/home');
```

Рисунок 2.3 – Звернення зі сторони клієнту на певний ендпойнт

На рисунку 2.3 продемонстрований самий звичайне та банальне звернення до ендпойнту [7]. Звертаючись зі сторони клієнта на такий ендпойнт, ми очікуємо, що сервер прослуховує такий ендпойнт і в разі отримання запиту, відправить нам щось у відповідь. Наприклад, результат, який можна побачити на рисунку 2.4.

```
app.get('/home', (request, response) => {
    response.send('Запит отримано та оброблено')
})
```

Рисунок 2.4 – Приклад резолверу на сервер, котрий реагує на запит на конкретний ендпойнт

При зверненні на продемонстрований ендпойнт, сервер відповість рядком «Запит отримано та оброблено». Результат звернення зі сторони клієнту можна тепер прочитати. Це зображено на рисунку 2.5.

```
const response = await fetch('https://api.example.com/home');

console.log(response) //* Запит отримано та оброблено
```

Рисунок 2.5 – Вивід результату запиту

Підбиваючи підсумки, резолвер – це механізм, який очікує на звернення на якийсь ендпойнт та реагує на нього.

Але ендпойнти це не тільки КУДИ слід звернутись, але й можливість передати якісь «легкі» данні за допомогою квері-параметрів. Це зображено на рисунку 2.6.

Тепер резолвер на ендпойнт /home очікує, що в запиті містимуться квері-параметри page та article. Формуються квері-параметри наступним чином: після ендпойнту /home ставиться символ «?», який символізує те, що далі йдуть квері-параметри.

```
const response = await fetch('https://api.example.com/home?page=1&article=1');

app.get('/home', (request, response) => {
    const page = request.query.page;
    const article = request.query.article;

    response.send(`Клієнт звернувся на ендпойнт home з параметрами page ${page} та article ${article}`)
})

console.log(response) //* Клієнт звернувся на ендпойнт home з параметрами page 1 article 1
```

Рисунок 2.6 – Обробка запиту з квері-параметрами

Самі параметри будуються в форматі параметр=значення, відповідно, page=1. Але параметрів всього два, тому між ними, як роздільний символ, ставить «&». В свою чергу, резолвер дістає ці параметри з об’єкту request та його поля query [2].

Питання «Як?» характеризується методами повідомлень протоколу HTTP. Протокол HTTP має аж 8 різних типів повідомлень на сервер [22]! Але для побудови CRUD (Create, Read, Update, Delete) використовують тільки 4 із них:

1) GET;
2) POST;
3) DELETE;
4) PUT.

Зі сторони клієнта ці запити формуються так, як показано на рисунку 2.7.

```
await fetch('https://api.example.com/home', {
    method: 'POST' //* GET/PUT/DELTE
});
```

Рисунок 2.7 – Методи запиту зі сторони клієнта

Другим аргументом до нативного методу fetch передається об’єкт з конфігурацією запиту. В попередніх прикладах такий об’єкт не був вказаний, тому за замовчуванням, був задіяний GET метод. Зі сторони серверу, Express аналогічно передбачає резолвери для відповідних методів [8]. Вони зображені на рисунку 2.8.

Рисунок 2.8 – Резолвери для кожного із HTTP-методів

Всі ці методи відрізняються своїм призначенням. Метод GET (Read в абревіатурі CRUD) призначений ТІЛЬКИ для отримання специфічних даних.

Написати на стороні серверу резолвер, який при GET запиті на основі ендпойнту чи квері-параметрів модифікуватиме Базу Даних вважається найгіршою із поганих практик програмування.

Це порушення правил REST архітектури та й можлива причина виникнення проблем, усунення яких коштуватиме великих грошей. Як мінімум, тому що при потенційних проблемах з Базою Даних, ніхто не очікуватиме, що проблема ховається в резолвері для GET запиту. Цей рзолвер призначений тільки для видачі потрібних даних клієнтові.

Стосовно інших методів, а саме POST/PUT/DELETE – їхнє призначення це модифікація Бази Даних.

Де-факто, вони між собою відрізняються лиш семантично, бо формується кожен з них однаково (рисунок 2.9).

Суть в тому який із резолверів спрацює на конкретний ендпойнт з конкретним методом. POST використовується у внесення нових даних в Базу Даних.

PUT – зміну уже існуючих даних в Базі Даних (вибірка по унікальному ідентифікатору, наприклад). DELETE, відповідно, видалити існуючі дані із Бази Даних.

```
const endpoint = 'https://api.expample.com/'

//* POST (CREATE) - додання нового студента
app.post(endpoint, (request, response) => {});

//* PUT (UPDATE) - оновлення студента
app.put(endpoint, (request, response) => {});

//* DELETE (DELETE) - видалення студента
app.delete(endpoint, (request, response) => {});
```

Рисунок 2.9 – POST/PUT/DELETE методи

Для того, щоб сервер розпізнав які дані клієнт хоче розмітити/оновити/видалити із Бази Даних, клієнт повинен їх передати. Це зображено на рисунку 2.10.

```
await fetch(endpoint, {
    method: 'POST', //* PUT/DELETE
    body: JSON.stringify(data)
});
```

Рисунок 2.10 – Передача даних на сервер зі сторони клієнту

В конфігурацію запиту в поле body передається об’єкт даних, котрі сервер отримає й відповідно, обробить належним чином.

Важлива примітка, що дані повинні передаватись винятково в JSON форматі, який є простим рядком [24]. А точніше, рядковим представленням будь-яких нерядкових типів даних (рисунок 2.11).

```
JSON.stringify({});                  // '{}'
JSON.stringify(true);                // 'true'
JSON.stringify('foo');               // '"foo"'
JSON.stringify([1, 'false', false]); // '[1,"false",false]'
JSON.stringify({ x: 5 });            // '{"x":5}'
```

Рисунок 2.11 – Типі даних та їх представлення в JSON

З точки зору використання методів, можна підмітити, що методи PUT та DELETE це просто інакша обгортка методу POST, що вони нічим не відрізняються один від одного. І це правда, їхня роль – семантична. Доволі багато систем написано з використанням одних лиш GET та POST методів, нехтуючи іншими. Це справді можливо, але тим не менш, якщо звернутися до рисунку 2.9 можна побачити переваги використання семантичних методів: один й той самий ендпойнт може обслуговуватися купою різних методів.

Величезна кількість сервісів та застосунків в інтернеті написані з використанням REST API. При вивченні frontend та backend програмування, в першу чергу, береться на увагу й вивчення REST API. Тим не менш, у 2015 році стався перший випуск GraphQL – мови запитів та маніпуляції з даними, розробленої компанією Facebook (нині уже відома як Meta). GraphQL надає інакший підхід до розробки Web API, тому цей інструмент це пряме протиставлення уже звичного та відомого всім REST API [4]. В одній із презентацій новоствореного GraphQL в 2016 році розробники розповіли про свої амбіціозні плани про поширення цієї графової мови запитів на всі великі компанії та технологічні гіганти. Ціль – зробити GraphQL всюди сутнім.

Перевагами GraphQL перед звичного REST API можна назвати економію: економія часу на розробку, економія трафіку на стороні клієнта.

Враховуючи, що цільова аудиторія сервісу буде використовувати застосунок саме через мобільний інтернет, то кожен зекономлений байт пам’яті має велике значення.

Підпункт про економію часу має на увазі оптимізацію трафіку зі сторони клієнта та оптимізацію часу на розробку.

Припустим, в Базі Даних існує сутність Vehicle, що зображена на рисунку 2.12, котра містить різноманітні поля, котра описує кожнен автобус: назву маршруту, номер, тип та дані про підприємство.

```
const vehicle = {
  route: '21',
  number: 12345,
  id: '180105',
  kind: 'bus',
  enterprise: {
    name: 'Електротранс',
    id: 67890,
  }
}
```

Рисунок 2.12 – Сутність Vehicle

Здається все доволі прозоро: клієнт відправляє GET запит на ендпойнт /vehicles, резолвер на бекенді відреагує, дістане весь транспорт й поверне ці дані у відповідь. Резолвер, приблизно виглядатиме так, як зображено на рисунку 2.13.

```
app.get('/vehicles', async (_, response) => {
  //* псевдо-ініціалізація БД
  const db = new Database();
  try {
    const vehicles = await db.getAll({ key: 'vehicle' })
    response.send(vehicles)
  } catch (error) {
    response.send(error)
  }
})
```

Рисунок 2.13 – Резолвер для БД

Але з плином часу проектові можливо буде потрібен інший ендпойнт, котрий поверне тільки підприємство із сутності Vehicle. Можливі рішення для подібної задачі:

1) написати окремий резолвер, котрий дістане підприємство, звертаючись в іншу таблицю Бази Даних;

2) якщо бекенд був погано спроектований, та не був готовий до подібних можливих нововведень, доведеться діставати сутність enterprise із сутності vehicle (найгірший із можливих сценаріїв) та повертати в новому резолвері.

Якщо говорити про перший пункт, тут також все прозоро: створюємо ще один резолвер поряд, котрий поверне підприємство.

Якщо подивитись на рисунок 2.14, можна побачити невтішну тенденцію росту кількості резолверів.

Здається, ніби все добре, проект масштабується та росте, тому це цілком природньо. Тим не менш, проблема в тому, що проект «росте на місці» - не були додані нові сутності, які потрібно обробити, це старі сутності обростають новими резолверами. Такий малий приклад, а уже великий обсяг коду.

Подібні тенденції можна уникнути завдяки GraphQL.

```
app.get('/vehicles', async (_, response) => {
  const db = new Database();
  try {
    const vehicles = await db.getAll({ key: 'vehicle' })
    response.send(vehicles)
  } catch (error) {
    response.send(error)
  }
})

app.get('/enterprises', async (_, response) => {
  const db = new Database();
  try {
    const enterprise = await db.getAll({ key: 'enterprise' })
    response.send(enterprise)
  } catch (error) {
    response.send(error)
  }
})
```

Рисунок 2.14 – Негативна тенденція росту кількості резолверів

У випадку якщо потрібен буде ендпойнт на кшталт /vehicle-ids (рисунок 2.15), в якому резолвер повинен повернути тільки унікальні ідентифікатори із сутності Vehicle здійсниться план дій, в якому потрібно розширювати резолвери.

```
app.get('/vehicle-ids', async (_, response) => {
  const db = new Database();
  try {
    const vehicles = await db.getAll({ key: 'vehicle' })
    const ids = vehicles.map(vehicle => vehicle.id)

    response.send(ids)
  } catch (error) {
    response.send(error)
  }
})
```

Рисунок 2.15 – Резолвер для ендпойнту /vehicle-ids

В такому випадку існує 2 виходи: написати новий резолвер, котрий дістає сутність Vehicle, дістає із цієї сутності тільки потрібні поля (в нашому випадку, це поле id) та повернути оброблені дані або використовувати ендпойнт /vehicles і уже на стороні клієнта діставати із поверненої сутності потрібні поля. Обидва варіанти погані.

В першому випадку росте кількість резолверів, які «топчуться на місці» та обслуговують одну й ту ж сутність.

В другому – передаються клієнтові непотрібні дані. Із-за цього, час коли клієнт отримає ці дані зростає заради нічого. Вигідним рішенням цієї проблеми буде використання GraphQL.

Перш за все, потрібно об’явити типи сутностей GraphQL (рисунок 2.16): Vehicle, Enterprise та кореневу сутність Query, яка із себе являє список всіх можливих GraphQL резолверів, які обробляють query запити (GET запити в середовищі GraphQL) [4].

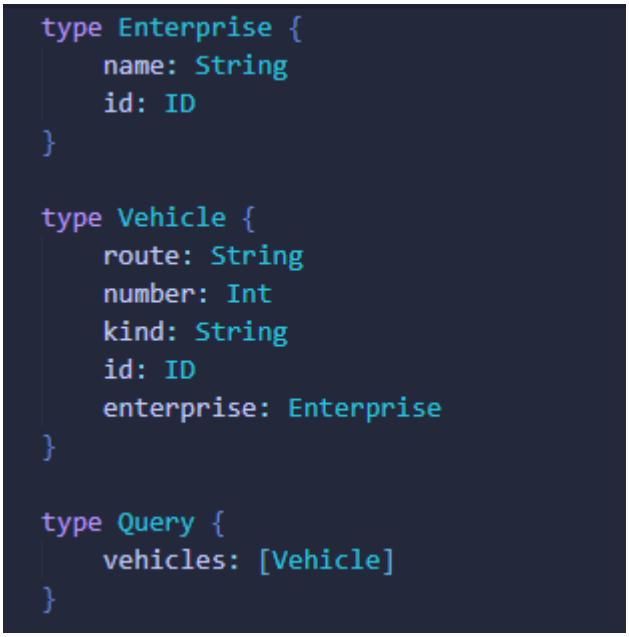


Рисунок 2.16 – Типи GraphQL

В тип Query як поле указується ім’я GraphQL резолверу, котрий обробить запит. Як значення цього поля, указується тип даних, котрі повертаються цих резолвером. В нашому випадку, резолвер vehicles поверне масив даних із типом Vehicle.

Тепер залишається лише написати резолвер для vehicles. Об’єкт resolvers отримує резолвер для корінного типу Query, котрий був об’явлений на рисунку 2.16.

Та по описаній аналогії була створена функція-резолвер vehicles, яка зображена на рисунку 2.17, котра повертає всього транспорту із БД.

Тепер клієнт повинен сам повідомити які йому потрібні поля. Для цього формується GraphQL-схема запиту.

Рисунок 2.18 демонструє як формуються уже згадані схеми запиту й використання відповідної схеми має аналогічний ефект порівняно з раніше демонстрованими резолверами по ендпойнтам /vehicles, /enterprises та /vehicle-ids. Кардинально змінився підхід до розробки: бекенд віддає всі дані, а клієнт сам вирішує, що із них потрібно.

```
const resolvers = {
  Query: {
    vehicles: () => {
      const db = new Database();

      return await db.getAll({ key: 'vehicle' })
    },
  }
};

const server = new ApolloServer({
  resolvers
});
```

Рисунок 2.17 – Приклад резолверу для vehicles

Не потрібно писати додаткові резолвер, щоб витягнути якесь конкретне поле із БД. Максимум – написати додатковий специфічний резолвер для виконання уже якоїсь специфічної логіки.

Це значна економія часу для написання таких специфічних запитів, а також й економія трафіку.

```
export const VEHICLES = gql`
    query vehicles {
      route
      number
      kind
      id
      enterprise {
        name
        id
      }
    }
`;
```

```
export const ENTERPRISES = gql`
    query vehicles {
      id
      vehicles {
        name
        id
      }
    }
`;

export const VEHICLE_IDS = gql`
    query vehicles {
      id
    }
`;
```

Рисунок 2.18 – Схеми для запитів

2.6 Стилізація та оформлення

Стилізація застосунку – це дуже кропітка та чутлива робота. Красивий User Interface приваблює користувачів, але одночасно з цим, помилки в оформлені, допущені навіть на рівні 1-2 пікселів, можуть стати фатальними та залишити відлякати користувачів. З точки зору розробки також вартує бути обережним. Навіть маленький лендінг може запросто розростись до 500 і більше рядків коду CSS. Тому не вартує нехтувати та бути легковажним у виборі інструментів для стилізації. Стилізація – це величезний пласт роботи в розробці будь-якого frontend застосунку. Якщо вибрати незручний або недоречний інструмент – страждання протягом усього життєвого циклу розробки застосунку гарантовані.

Сучасна веб-розробка може похвалитись різноманіттям інструментів для роботи зі стилями. Років 3-5 назад, основним кістяком був CSS, а інші інструменти як SCSS тільки набували своєї сьогоднішньої популярності, то зараз технологія SCSS стала мастодонтом та уже своєрідним стандартом у розробці, а такі неординарні технології як JSS стають звичною практикою.

JSS (JavaScript in stylesheet, або як ще називають CSS-in-JS) – підхід стилізації, коли стилі CSS описуються в JavaScript об’єктах. Плюси використання JSS:

1) динамічні стилі. Оскільки це звичайні JavaScript об’єкти, ними можна легко маніпулювати локальними змінними, тернарними операторами, або функціями, які в залежності від аргументів, повернуть потрібні стилі;

2) масштабованість. З ростом застосунку, збільшується й кількість стилів CSS, тому часто виникають колізії – конфлікти в назвах класів. Тяжко регулювати файл стилів, якщо він займає 1000 і більше рядків коду. Завдяки модульності JSS, програміст застрахований від таких проблем.

Такий підхід імплементують купа різних інструментів, найбільш відомими є бібліотека Styled Components та Material-UI. Вартує зазначити, що React також реалізовує базовий JSS: кожен блок має атрибут style, який приймає об’єкт з описаними стилями CSS (рисунок 2.19).

```jsx
const Component = () => {
    return (
        <div
            style={{
                border: '1px solid yellow',
                width: 100,
                height: 100,
                background: 'blue',
                color: 'yellow',
            }}
        >
            Styled div
        </div>
    )
}
```

Рисунок 2.19 – «Нативний» JSS в React

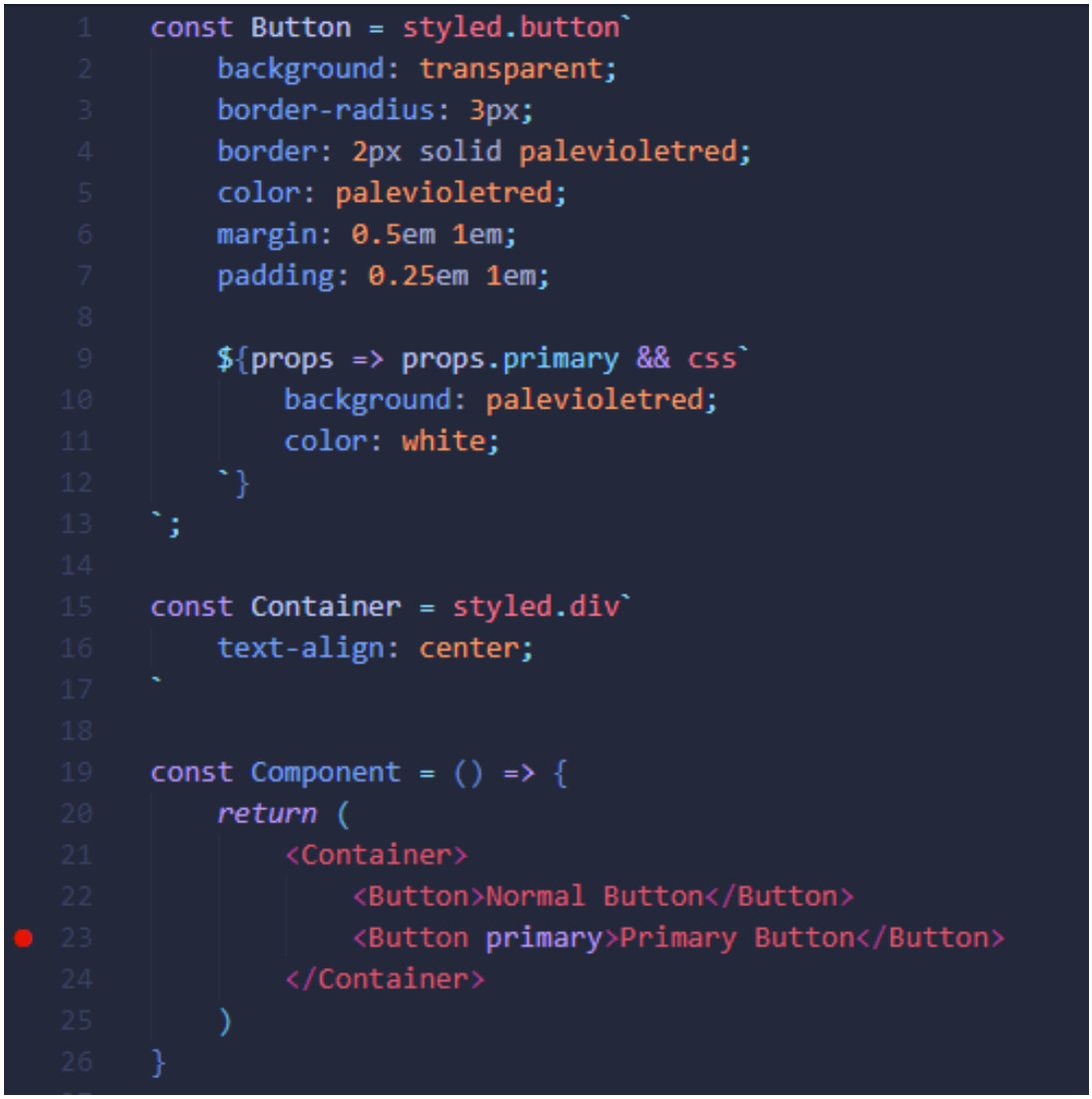
Проблема «нативного» JSS в React в тому, що такі стилі при компіляції JSX розмітки в атрибут style HTML розмітки [11]. А це значить, що такі стилі апріорі мають найвищий пріоритет, тому ними важко маніпулювати зовні – потрібне обов'язкове втручання в конкретний блок напряму. Рисунок 2.20 демонструє цю проблему. Якщо в CSS якийсь елемент має клас child, то його можна переписати зовні, «переваживши» за допомогою комплексного селектору. Код на рисунку 2.20 означає що кожен елемент з класом child матиме жовтий колір, але такі елементи, які є дітьми батьківського елемента з класом container матимуть синій колір. «Нативний» JSS в React позбавлений такої можливості.

```css
.child {
    color: yellow
}

.container .child {
    color: blue
}
```

Рисунок 2.20 – переписування стилів за допомогою комплексних селекторів

«Використовуйте найкращі версії ES6 та CSS, щоб стилізувати свої програми без стресу» - з таким слоганом зустрічає Styled Components [26]. Синтаксис зображено на рисунку 2.21.



```
const Button = styled.button`
    background: transparent;
    border-radius: 3px;
    border: 2px solid palevioletred;
    color: palevioletred;
    margin: 0.5em 1em;
    padding: 0.25em 1em;

    ${props => props.primary && css`
        background: palevioletred;
        color: white;
    `}
`;

const Container = styled.div`
    text-align: center;
`

const Component = () => {
    return (
        <Container>
            <Button>Normal Button</Button>
            <Button primary>Primary Button</Button>
        </Container>
    )
}
```

Рисунок 2.21 – Синтаксис Styled Components

Styled Components пропонує створення стилів у вигляді компонентів, які використовуються в самому React.

На рисунку 2.21 продемонстровано що створюється компонент Button і завдяки директиві styled.button бібліотека розуміє, що потрібно створити компонент на основі HTML елемента <button />. Далі, описуються стилі для цього компонента.

Також, продемонстровано динамічні стилі, які регулюються завдяки аргументам, переданим в компонент Button на рядку 23. Компонент аргумент primary, завдяки чому ця кнопка отримає додаткові стилі, написані в рядкові 9.

Плюси Styled Components:

1) як і будь-який JSS, Styled Components уникає колізій імен класів;

2) простота в створені динамічних стилів;

3) автоматичні префікси CSS. Деякі CSS правила потребують спеціальних префіксів, щоб різні браузери могли їх розпізнати. Styled Components автоматично генерує ці префікси.

Мінуси Styled Components:

1) гігантизм. Оголошення компонентів Styled Components не може похвалитися компактністю;

2) колізії на локальному рівні. Не дивлячись на модульність Styled Components, завдяки якій програміст може об’явити компоненти з однаковими іменами в різних файлів, проблема з’являється коли ці компоненти з однаковими іменами треба використати в одному файлі разом.

Ще одною популярною імплементацією JSS є бібліотека Material-UI. Синтаксис принципу JSS в реалізації цієї бібліотеки зображено на рисунку 2.22.

Основна відмінність від Styled Components – це те що всі стилі описуються в рамках єдиного об’єкту, а не окремими незалежними компонентами [26].

Приклад на рисунку 2.22 реалізує аналогічне оформлення, що і приклад на рисунку 2.21.

Плюси Material-UI:

1) відсутність колізій класів та імен компонентів на локальному рівні. Всі стилі зберігаються в межах одного об’єкту;

2) бібліотека містить в собі додаткові корисні інструменти, як брейкпойнти для адаптивного дизайну [17];

3) оскільки стилі не роздроблені на незалежні компоненти, як Styled Components, дуже легко маніпулювати стилями через CSS селектори.

```
const useStyles = makeStyles({
    button: {
        background: 'transparent',
        borderRadius: 3,
        border: '2px solid palevioletred',
        color: 'palevioletred',
        margin: '0.5em 1em',
        padding: '0.25em 1em',
    },
    primaryButton: {
        background: 'palevioletred',
        color: 'white',
    },
    container: {
        textAlign: 'center',
    }
  });

const Component = () => {
    const classes = useStyles();

    return (
        <div className={classes.container}>
            <button className={classes.button}>Normal Button</button>
            <button className={`${classes.button} ${classes.primaryButton}`}>Primary Button</button>
        </div>
    )
}
```

Рисунок 2.22 – Синтаксис Material-UI

Мінуси Material-UI:

1) порівняно зі Styled Components, складніше створювати динамічні стилі;

2) Styled Components автоматично генерує префікси стилів, в той час як Material-UI не може цим похвалитись.

Останній інструмент, котрий хотілось би розглянути, це – SCSS. Перш за все, важливо уточнити, що SCSS, це похідна, «діалект» від SASS. SASS (Syntactically Awesome Style Sheets) – метамова, яка інтерпретується в звичайний CSS. Суть SASS – додаткові можливості, котрих немає в CSS [15].

Фактично, SASS – це синтаксичний цукор над CSS. В принципі, на це натякає абревіатура SASS, котру можна перевести як «Синтаксично класні каскадні таблиці».

Мова SASS має два синтаксиси: SASS (основний) та SCSS. Перш за все, в чому різниця між ними? Все доволі просто: в SCSS передбачено наявність фігурних дужок. Це показано на рисунку 2.23.

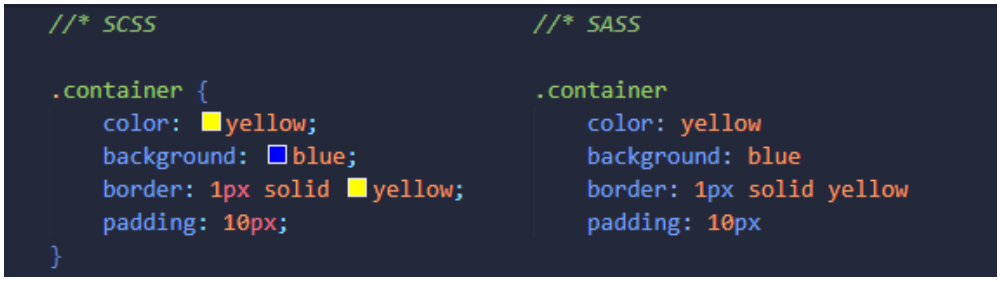


Рисунок 2.23 – SCSS та SASS

В SASS границі блоків визначаються табами, в той час SCSS ці границі визначає по фігурним дужкам. А також SCSS визначає кінець правила стилів за допомогою крапки з комою. Наче між SCSS та SASS немає різниці, але ці відмінності в лиці наявності фігурних дужок та крапки з комою грають велику роль [15]. Із-за наявності цих двох атрибутів SCSS стає дуже схожим на нативний CSS, цей інструмент асоціюється як звичайне доповнення існуючого функціоналу, яким по факту, він і є та має бути. В той час SASS із-за синтаксичних відмінностей від CSS сприймається як щось чужорідне, ніби це зовсім інша мова. Та і відокремлення блоків за допомогою табів, а не класичних фігурних дужок, не дуже зручно. Саме тому SCSS, хоч і являється діалектом від SASS, має куди більший попит.

SCSS це доповнення CSS, яке додає різні нововведення. Серед них, це набагато зручніший синтаксис об’явлення змінних, вкладеність правил в ієрархію, створення міксинів та зручне розширення/наслідування наборів правил стилів.

Плюси SCSS:

1) інструмент має багатий функціонал, котрий спрощує та робить зручним написання стилів;

2) простота синтаксису та максимальна схожість до нативного CSS. Освоєння такого інструменту займе максимум 1 вечір й не викликає ніяких труднощів у використанні.

Мінуси SCSS:

1) SCSS хоч і намагається покращити CSS, але це не врятувало інструмент від наслідування стандартних проблем CSS, як колізії імен класів та погану масштабованість;

2) має найбільший розмір пакету (4.49 МБ). До прикладу, Material UI styles важить 754 КБ, а Styled Components 3.42 МБ. Для великих проектів, в котрих задіяно багато інструментів та бібліотек це вагомий аргумент.

Розглянувши всі варіанти, як висновок, вибір інструменту для реалізації проекту дипломної роботи, ліг на Material UI styles, оскільки здається оптимальним варіантом:

1) лаконічний синтаксис;
2) легке масштабування;
3) наявність всіх плюсів підходу JSS;
4) найменший розмір пакету, що є найбільш виграшним вибором у критерії функціонал-розмір.

2.7 Висновки

В цьому розділі було здійснено планування проекту та вибір можливих підходів й методів. Також було здійснено аналіз допоміжних інструментів, огляд їх плюсів та мінусів, проведено паралелі з їх конкурентами та була дана аргументована відповідь стосовно їхнього вибору для реалізації проекту дипломної роботи.

ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ СИСТЕМИ МОНІТОРИНГУ

3.1 Принципи роботи системи моніторингу та план реалізації проекту

Будь-яка розробка проекту починається саме з розробки backend частини, з чітко продуманою архітектурою, ендпойнтами та резолверам. Лише після цього виконується розробка frontend частини за уже продуманим дизайном. Backend працюватиме за наступною схемою:

1) ініціалізація застосунку. В Базу Даних відразу записуються всі доступні маршрути транспорту, щоб потім можна було використовувати ці данні для резолверів в якості відповіді на запити зі сторони клієнта;

2) квері routes повертатиме всі доступні маршрути;

3) квері route повертатиме данні про конкретний маршрут, в тому числі й координати положення транспорту в даний момент:

a) клієнт робить запит на сервер (Рисунок 3.1, пункт 1);

b) сервер обробляє запит за допомогою GraphQL резолвера (Рисунок 2.1, пункт 2);

c) GraphQL резолвер відправляє запит на Express резолвер, котрий являє собою симуляцію датчику геолокації транспорту, щоб дізнатись про координати положення транспорту в даний момент (Рисунок 3.1, пункт 3);

d) оскільки відкритих API не передбачено, буде здійснена емуляція руху транспорту: при кожному новому запиті на Express резолвер, координати положення транспорту будуть автоматично змінюватись, тим самим здійсниться симуляція руху транспорту. Ці данні повертаються у відповідь на запит резолверу GraphQL (рисунок 3.1, пункт 4);

e) при отримані відповіді від Express резолвера, GraphQL резолвер поверне у якості відповіді данні клієнтові, які він отримав.

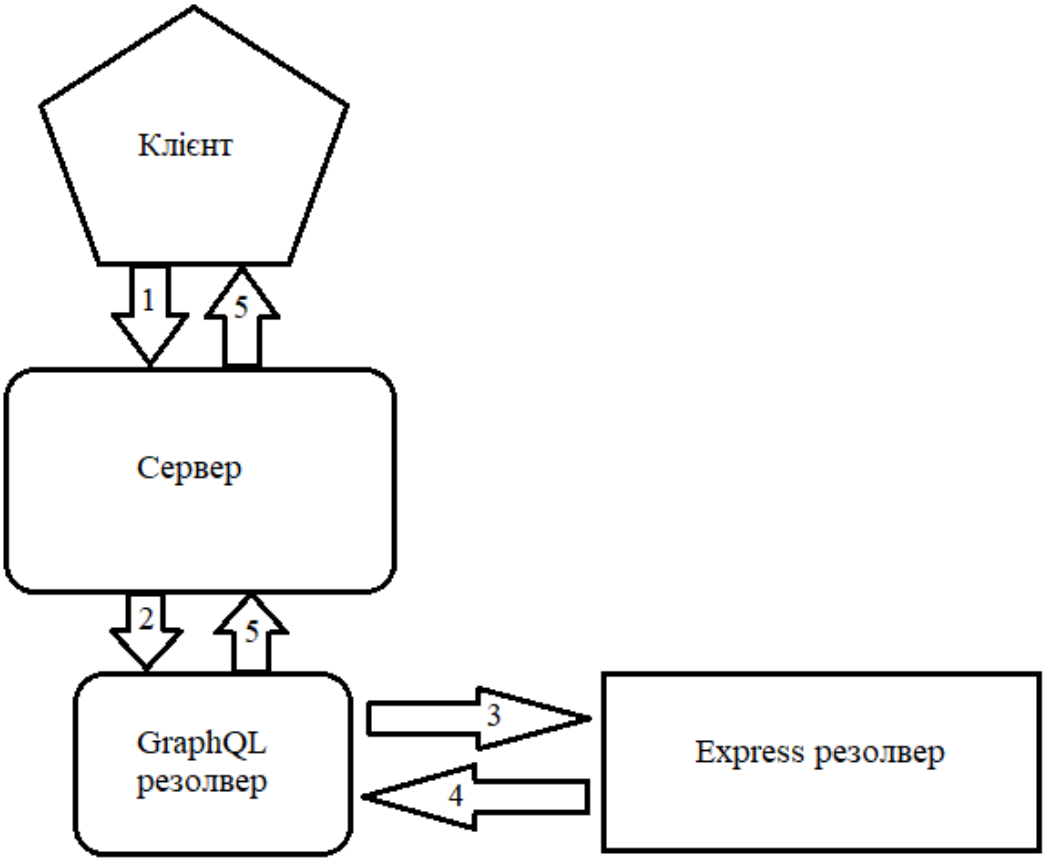


Рисунок 3.1 – Схема роботи сервера та резолвера для квері route

В свою чергу, для frontend частини стосовно взаємодії з сервером вартує прояснити принцип роботи запиту на квері route. Вибір маршруту серед списку доступних маршрутів записуватиме його в локальний стан React. Для кожного збереженого маршруту в локальному стані спрацьовуватиме квері-запит route автоматично кожні 5-10 секунд (інтервал запитів можна буде налаштувати в конфігурації застосунку). Таким чином, користувач кожні 5-10 секунд отримуватиме оновленні дані для маршрутів, які його цікавлять. Розробка з'єднання через веб-сокети, для отримання даних в реальному часі, не планується. Такий спосіб отримання актуальні данні в рамках проекту здається марно затратним, бо користувачеві зовсім ні до чого отримувати дані кожну мілісекунду по веб-сокетам. Це затратно з точки зору економії трафіку, дані «в реальному часі» не будуть інформативними в рамках проекту, а також цілком достатньо

оновлювати дані через невеликі інтервали, тому нерозривне з'єднання не в пріоритеті розробки.

3.2 Розробка backend частини

Першим кроком розробки серверу є налаштування середовища розробки. За допомогою команди термінала npm init –y (середовище Node.js попередньо уже встановлене на комп'ютер). Ця команда генерує package.json файл, який є конфігурацією проекту, а також, списком доданих залежностей. Флаг –y означає, що потрібно створити стандартну конфігурацію. Єдина частина конфігурації, котра підлягає модифікації – це список скриптів. Було додано скрипти start та serve (рисунок 3.2).

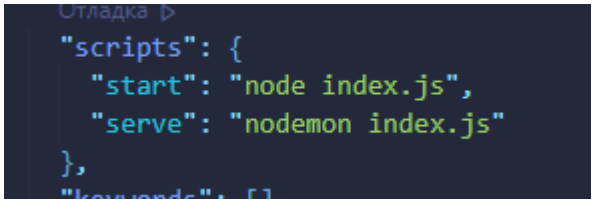


Рисунок 3.2 – Список скриптів для backend

Скрипт start запускає корінний файл index.js, котрий являється корінним файлом всього проекту. Скрипт serve запускає моніторинг змін в файлі index.js за допомогою пакету nodemon. Діло в тому, що середовище Node.js не реагує на зміни в файлах при звичайному запуску проекту. Щоб побачити результат внесених змін, потрібно кожного разу перезапускати проект, що доволі не зручно для розробки. Пакет nodemon запускає проект та реагує на кожну зміну у файлах, автоматично перезапускаючи проект. Цей пакет встановлюється в якості «залежності розробника» (в оригіналі – developer dependency) – npm install -D. Це значить, що при білді проекту (перебудови всього вихідного коду в мінімізований та оптимізований source код, який використовується в production) цей пакет не буде залучено в production збірку проекту.

Оскільки план проекту передбачає створення двох окремих серверів, потрібно об’явити вільні порти для кожного із них. Для серверу, котрий обслуговує клієнт напряму, - це порт 4001, а для серверу, який імітує датчики в громадському транспорту для моніторингу, - це порт 4000. Такі дані, зазвичай, записуються в спеціальний файл .env (від слова environment), який відповідає за всі критично важливі в архітектурі проекту змінні (рисунок 3.3).

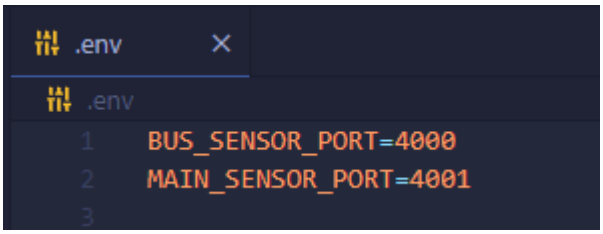


Рисунок 3.3 – Файл .env

Тепер ці змінні будуть доступні у всіх файлах та директоріях проекту через глобальний процес process.env. Хорошим тоном розробки, є створення файлу .env.dist, який копіює список всіх змінних записаний в .env, але при цьому не містить в собі значень для цих змінних. Тобто, це список порожніх змінних. Це робиться для того, тому що, зазвичай, файл .env приховується в системах контролю версій, тому розробник, котрий стягує репозиторій проекту не матиме цього файлу. Данні, котрі містяться в .env конфіденційні й повинні бути захищені. Вони повинні отримуватись від власника проекту, а не через відкритті репозиторії. Тому, створюється .env.dist файл, який є «формою» для заповнення цих даних після їх отримання.

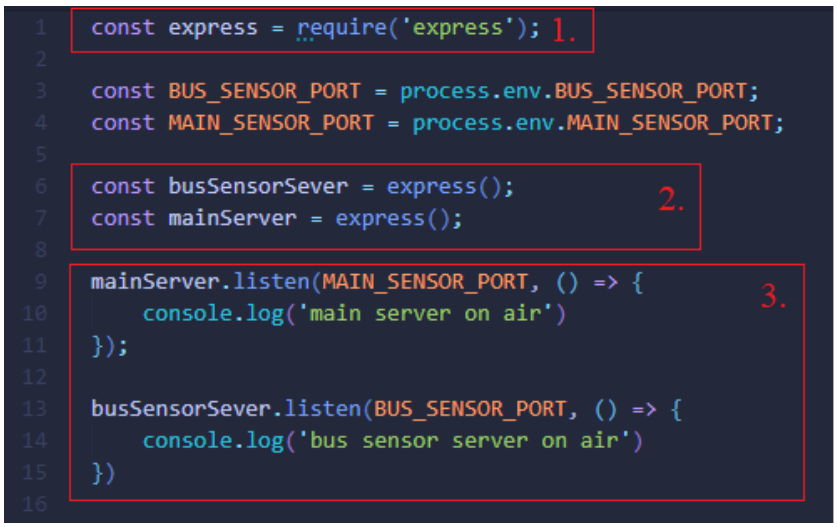
В файл .gitignore, традиційно, записана директорія node_modules та файл .env. Це робиться для того, щоб система контролю версій git не індексувала ці файли та папки, і, як результат, вони не потрапили в репозиторій.

На даному етапі розробки, було ініціалізовано два сервери. Як говорилось уже раніше, перший – для обслуговування запитів клієнта, а другий – імітація датчика транспорту. Для цього були зроблені наступні кроки (рисунок 3.4):

1) встановлено та імпортовано Express.js (npm install express) [8];

2) ініціалізовано сервера;

3) запуск серверів.




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

const BUS_SENSOR_PORT = process.env.BUS_SENSOR_PORT;
const MAIN_SENSOR_PORT = process.env.MAIN_SENSOR_PORT;

const busSensorSever = express();
const mainServer = express();

mainServer.listen(MAIN_SENSOR_PORT, () => {
    console.log('main server on air')
});

busSensorSever.listen(BUS_SENSOR_PORT, () => {
    console.log('bus sensor server on air')
})
```

Рисунок 3.4 – Базова ініціалізація серверів

Тепер запуск проекту за допомогою команди npm run serve сповіститься нас про те, що обидва сервери успішно запущені (рисунок 3.5).

```
[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
bus sensor server on air
main server on air
```

Рисунок 3.5 – Консоль серверу, котра сповіщає про успішний запуск серверів

Оскільки сервери готові до експлуатації, потрібно ініціалізувати базу даних:

1) перейти на сайт MongoDB та зареєструватись;

2) створити новий проект (рисунок 3.6);

3) створити нову базу даних: вибрати регіон та провайдер БД (в даному випадку, було вибрано Франкфурт, AWS), встановити логін та пароль для з’єднання з БД.

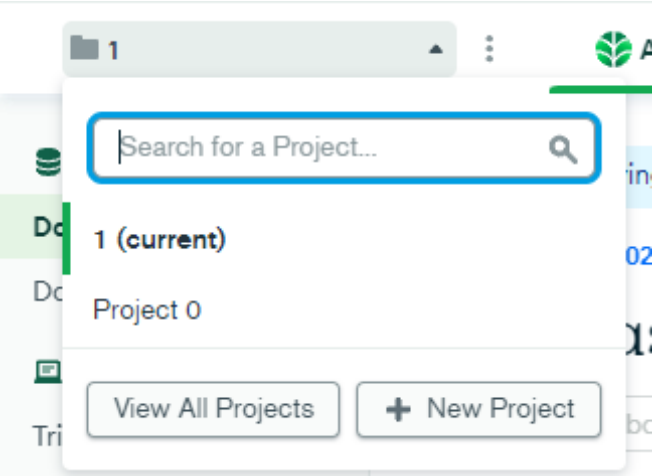


Рисунок 3.6 – Створення нового проекту

Після всіх цих маніпуляцій, створено кластер в Базі Даних, за допомогою якого, можна моніторити активність БД, моніторити наявні дані та колекції в БД та змінювати конфігурацію БД (рисунок 3.7).

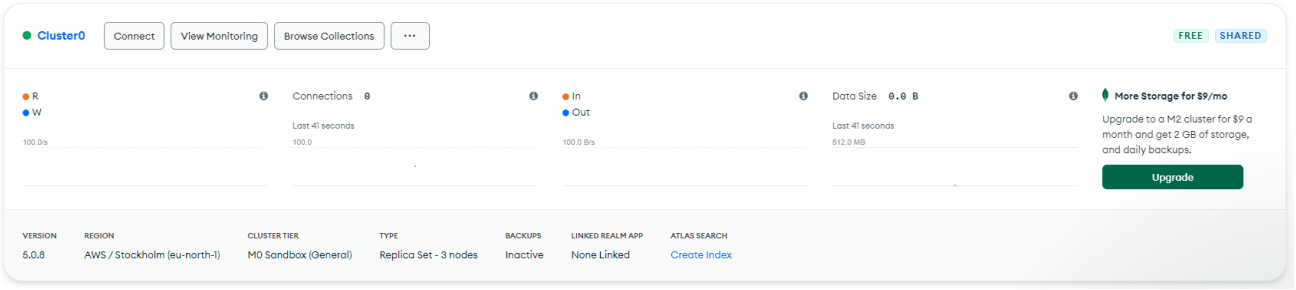


Рисунок 3.7 – Кластер MongoDB

Тепер БД готова до експлуатації. На сервері потрібно встановити пакет mongoose (npm install mongoose) [14], котрий дозволяє створювати з'єднання з БД та проводити маніпуляції як запис, пошук, видалення даних.

Якщо натиснути в кластері кнопку Connect, ми отримаємо URL, по якому сервер зможе з'єднатись з БД. Цей URL в містить в собі плейсхолдери, котрі потрібно замінити вказаними при створенні БД логіном та паролем. Це зображено на рисунку 3.8 в пункті 1. Логін та пароль вартує також записати в файл .env, що зображено на рисунку 3.8 в пункті 2. Для з'єднання з БД, було створено

асинхронну функцію, в яку також перемістили запуск головного сервера (рисунок 3.8, пункт 3).

```
const mongoose = require('mongoose');  807.2k (gzipped: 220.2k)

const password = process.env.MONGO_PASSWORD
const username = process.env.MONGO_USERNAME;   2.

const db = `mongodb+srv://${username}:${password}@cluster0.umhoc.mongodb.net/?retryWrites=true&w=majority`;   1.

const init = async () => {
    await mongoose.connect(db, { useNewUrlParser: true, useUnifiedTopology: true });

    mainServer.listen(MAIN_SENSOR_PORT, () => {
        console.log('main server on air')          3.
    });
}

init();
```

Рисунок 3.8 – З’єднання з БД

Тепер час об’явити моделі БД. Mongoose дозволяє створювати абстракції, котрі називаються моделями для зручної маніпуляції з даними. Модель містить в собі схему даних, котра описує всі структури, котрі містить ця модель. Іншими словами, модель – це інтерфейс певної колекції, котрий дозволяє зчитування, запис, видалення та пошук даних цієї колекції [14]. Моделі, зазвичай, зберігають в папці models, тому була створена така папка та файл bus.js, в якому описана модель Bus (рисунок 3.9).

```
const { default: mongoose, Schema } = require('mongoose');

const busSchema = Schema({
    route: String,
    capacity: String,
    transportNumber: Number,
    kind: String,
    position: [Number],
    routeId: String
});

const Bus = mongoose.model('Bus', busSchema);

module.exports = Bus;
```

Рисунок 3.9 – Модель Bus

Модель описує структуру даних, яка буде зберігатись в БД. В моделі Bus описані наступні поля:

1) route – маршрут транспорту;
2) capacity – наповненість транспорту;
3) transportNumber – державний номер транспорту;
4) kind – вид транспорту;
5) position – масив можливих координат положення транспорту (до цього повернемось значно пізніше);
6) routeId – унікальний ідентифікатор транспорту.

Настав час описати API для busSensorServer. Цей сервер слухатиме ендпойнт /route/:routeId (рисунок 3.10), де routeId – ідентифікатор транспорту. Ендпойнт дістає з БД дані про цей транспорт та автоматично змінить позицію цього транспорту (до цього також повернемось значно пізніше). А поки, цей ендпойнт просто поверне дані про цей транспорт.

```
busSensorSever.get('/route/:routeId', async (request, response) => {
    const routeId = request.params.routeId;
    const bus = await Bus.findOne({ routeId });

    response.json(bus);
})
```

Рисунок 3.10 – Ендпойнт, який обслуговує сервер-імітатор датчику транспорту

Тепер опишемо сервер, котрий обслуговує клієнт напряму.

Цей сервер використовуватиме GraphQL, тому перший крок – це опис GraphQL схем (рисунок 3.11).

```
const schema = buildSchema(`
    type Bus {
        id: ID
        route: String
        capacity: String
        transportNumber: Int
        kind: String
        position: [Int]
        routeId: String
    }
    type Query {
        routes: [Bus]
        route(routeId: ID): Bus
    }
`);
```

Рисунок 3.11 – Схеми GraphQL

Тип Bus дуже схожий на модель Bus, єдина відмінність – наявність поля id, котра генерується автоматично GraphQL та необхідна для всіх схем та типів. Також, слід звернути увагу на тип Query, в якому описуються уже всі функції-обробники в GraphQL резолвері [4]. Як і було заплановано, GraphQL матиме такі резолвери, як route – для специфічного транспорту та routes – щоб дістати список наявного транспорту.

Опис резолверів зображено на рисунку 3.12. Функція routes просто дістає весь список транспорту за допомогою mongoose моделі Bus.

Конструкція find({}) означає що потрібно дістати усі дані із колекції моделей Bus (рисунок 3.12, пункт 1). Функція route приймає як аргумент унікальний ідентифікатор routeId та використовує як квері-параметр в запиті на сервер busSensorServer та просто повертає значення, котре прийшло у відповідь (рисунок 3.12, пунк 2).

Одним із останніх кроків на даному етапі створенні backend частини є ініціалізація GraphQL middleware [13]. Вона необхідна, щоб GraphQL резолвери реагували на всі квері, котрі приходять зі сторони клієнта (Рисунок 3.13).

```
const resolvers = {
    routes: async () => {
        const buses = await Bus.find({}); 1.

        return buses;
    },
    route: async ({ routeId }) => {
        const busResponse = await axios.get(`http://localhost:${BUS_SENSOR_PORT}/route/${routeId}`); 2.

        return busResponse.data;
    },
}
```

Рисунок 3.12 – GraphQL резолвер

```
const { graphqlHTTP } = require('express-graphql');

mainServer.use('/graphql', graphqlHTTP({
    graphiql: true,
    schema,
    rootValue: resolvers,
}))
```

Рисунок 3.13 – GraphQL middleware

Проблема в тому, що БД порожня. Щоб не вносити данні вручну на даному етапі розробки, використаємо наступну хитрість (рисунок 3.14):

1) створимо файл зі статичними даними про транспорт (масив data, елементи якої, відповідні до моделі Bus);

2) перевірка чи порожня БД. Якщо маршрут з route=21 існує, значить БД уже проініціалізована і не потребує в записі даних;

3) в іншому випадку, вносимо константу data в БД.

Настав час етапу тестування серверу. Завдяки параметру graphiql=true при ініціалізації GraphQL middleware, ми маємо доступ до інтерактивної пісочниці від GraphQL, в якій ми можемо запускати квері та тестувати результат.

Пісочниця доступна за ендпойнтом /graphql (рисунок 3.15). Запуск квері route з аргументом routeId=1 поверне дані про цей транспорт. При чому, ми можемо розширювати список потрібних полів, котрі нам потрібні і відповідають GraphQL

типу Bus. В цьому сенс GraphQL – ми дістаємо тільки ті поля і дані, які нам потрібні.

```
const data = [
    {
        routeId: '1',
        route: '21',
        position: [1, 2, 3, 4, 5],
        capacity: 'full',
        transportNumber: 1111,
        kind: 'bus',
    },
    {
        routeId: '2',
        route: '22',
        position: [6, 7, 8, 9, 10],
        capacity: 'full',
        transportNumber: 2222,
        kind: 'bus',
    },
]

const init = async () => {
    await mongoose.connect(db, { useNewUrlParser: true, useUnifiedTopology: true });

    const bus = await Bus.findOne({ route: '21' });

    if (!bus) await Bus.insertMany(data)

    mainServer.listen(MAIN_SENSOR_PORT, () => {
        console.log('main server on air')
    });
}
```

Рисунок 3.14 – Запис даних при ініціалізації БД

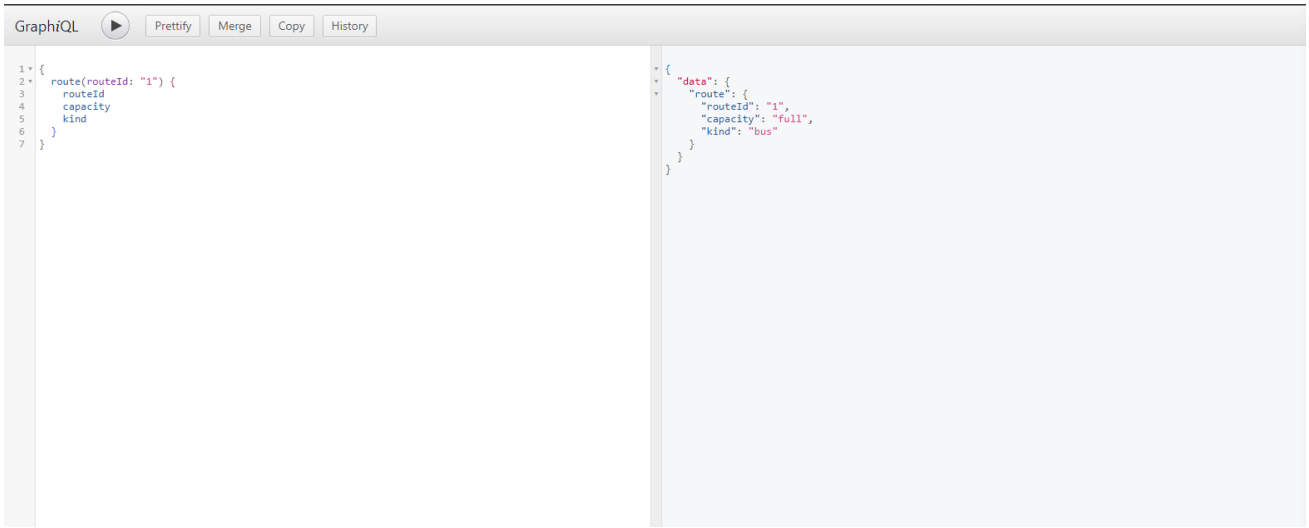


Рисунок 3.15 – Пісочниця GraphQL

Тим не менш, можливості пісочниці GraphQL не обмежуються одним лиш тестування запитів, але й можливістю перегляду доступних запитів (квері,

мутацій, підписок) GraphQL з повним описом структур даних, котрі використовуються для, або повертаються цим запитом (рисунок 3.16).

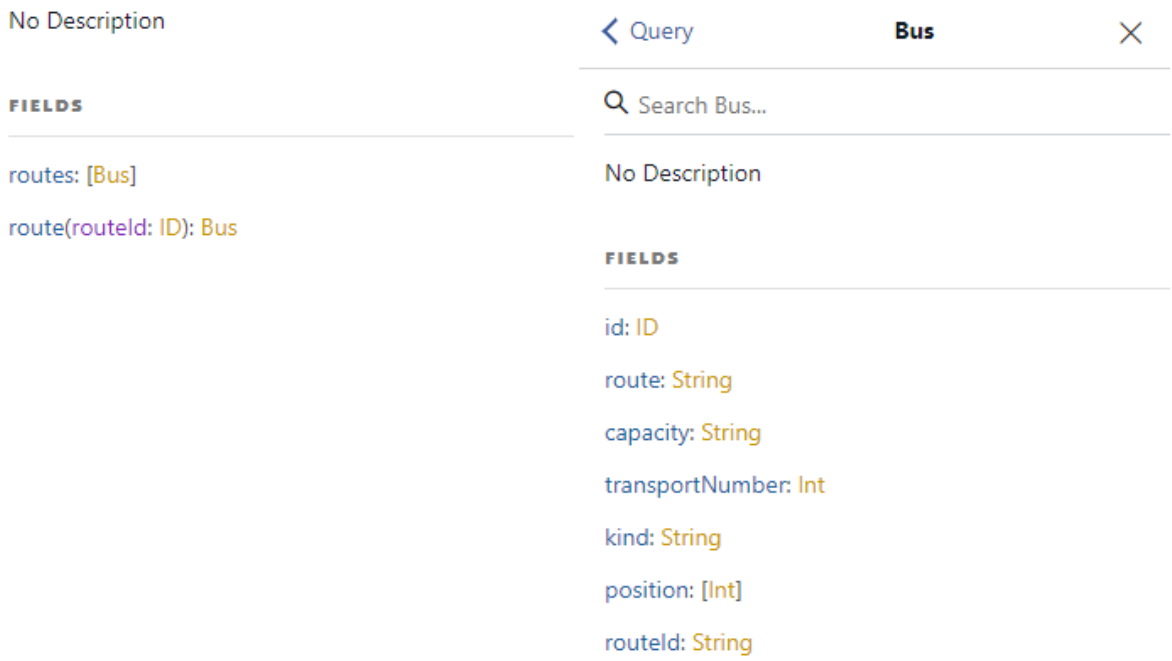


Рисунок 3.16 – Документація запитів GraphQL

Як висновок, GraphQL показав себе не тільки як надзвичайно ефективна та надійна технологія в противагу класичному REST API, але і дуже зручний інструмент з купою утиліт «з під коробки», котрі значно облегшують розробку, що тільки підтверджує тезис про надзвичайну оптимізацію процесів протягом розробки проектів.

3.2 Розробка frontend частини

Перший крок для розробки клієнт застосунку є ініціалізація React проекту за допомогою утиліти Create React App: npx create-react-app diploma-project, де diploma-project назва проекту, а команда npx означає запуск CLI утиліти. Create React App створить конфігурацію проекту, необхідну для стабільної роботи React, та стандартну організацію файлів та директив (рисунок 3.17).

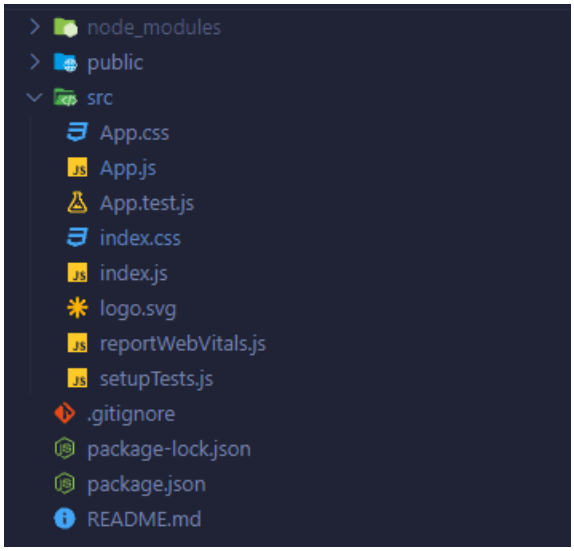


Рисунок 3.17 – Create React App

Create React App це зручна CLI утиліта для швидкої збірки проектів, розроблена командою Meta, котра дозволяє пропустити етап конфігурації збірників проектів як Webpack, менеджменту скріптів та налаштування компіляції та збірок проекту в Single Page Application (йдеться мова про налаштування плагінів, як Babel).

Наступним кроком буде організація Google Maps API на Платформі Google Карт, де слід зареєструвати обліковий запис, завести billing-акаунт та отримати ключ від API [12].

Хоч і Google розповсюджує своє API безкоштовно, але тим не менш, використання платформи Google Maps супроводжується попередженням, що в такому випадку це API може використовуватись тільки в некомерційних цілях, та й досвід роботи з подібним обмеженням у вигляді повідомлення-попередження (рисунок 3.18) не є задовільним в рамках сервісу.

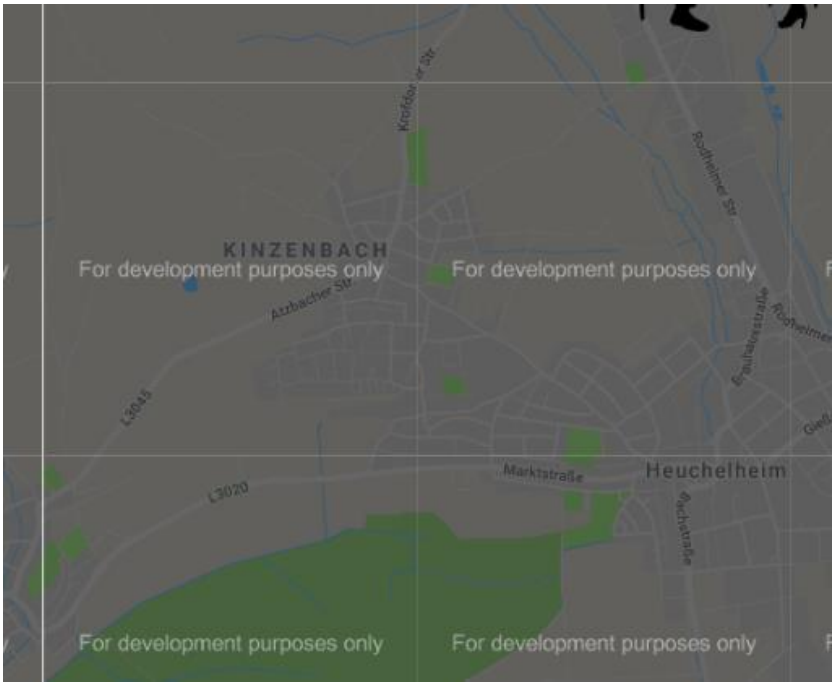


Рисунок 3.18 – Повідомлення-попередження Google API

Після отримання ключа для API, слід заповнити файл середовища .env необхідними системними змінними, як адрес backend серверу та вище згаданого ключа API (рисунок 3.19).

```
.env
1   REACT_APP_BACKEND_URL='http://localhost:4001'
2   REACT_APP_GOOGLE_API_KEY='AIzaSyAw71vCz3jwTOsNwiH3u2kYzXh6N8XbYXQ'
```

Рисунок 3.19 – Файл середовища для клієнта

Також, тепер настав час і для установки необхідних залежностей для використання Google API, а саме, пакет google-maps-react. Цей пакет включає в себе такі компоненти як карта, маркери для візуалізації даних, обробники подій й купу іншого функціоналу, необхідного для взаємодії з API. Первинне налаштування API зображеному на рисунку 3.20. Корінний компонент всього проекту App огортається High Order Component (HOC) GoogleApiWrapper для того, щоб, по перше, компонент App мав доступ до інструментів та локальних утиліт Google Maps у вигляді аргументів переданих компонентові (в React

термінології подібні аргументи називаються пропсами). По друге, через HOC GoogleApiWrapper передається API ключ.

```
import { Map, GoogleApiWrapper } from 'google-maps-react';  112.6k (gzipped: 30.4k)

function App(props) {
  const { google } = props;

  return (
    <Map
      google={google}
      zoom={14}
      initialCenter={{
        lat: 49.409358507364864,
        lng: 26.960622647423083
      }}
      disableDoubleClickZoom
      draggable={false}
      scrollwheel={false}
      disableDefaultUI
    >
      {
        //* Any other stuff wrapped by map
      }
    </Map>
  );
}

export default GoogleApiWrapper({
  apiKey: (process.env.REACT_APP_GOOGLE_API_KEY)
})(App);
```

Рисунок 3.20 – Первинне налаштування Google Maps API

Компонент Map із пакету google-maps-react відображає карту. Для більш коректного результату використовується наступна конфігурація (таблиця 3.1). Також зображено результат конфігурації Google API на рисунку 3.21.

Таблиця 3.1 – Параметри конфігурації Google API

Параметр	Значення	Довідка
initialCenter	lat: 49.409, lng: 26.960	Початкова точка яка відображається на карті. Приведені параметри вказують на центр міста Хмельницький

Кінець таблиці 3.1 – Параметри конфігурації Google API

Zoom	14	Рівень приближення
disableDoubleClickZoom	true	При true, цей параметр вимикає можливість приближення шляхом подвійного кліку по карті
draggable	false	При false, цей параметр вимикає можливість перетягувати карту (тимчасово вимкнено)
scrollWheel	false	При false, цей параметр вимикає можливість прокрутки карти колесом миші (тимчасово вимкнено)
disableDefaultUI	true	Вимикає стандартний UI Google Maps

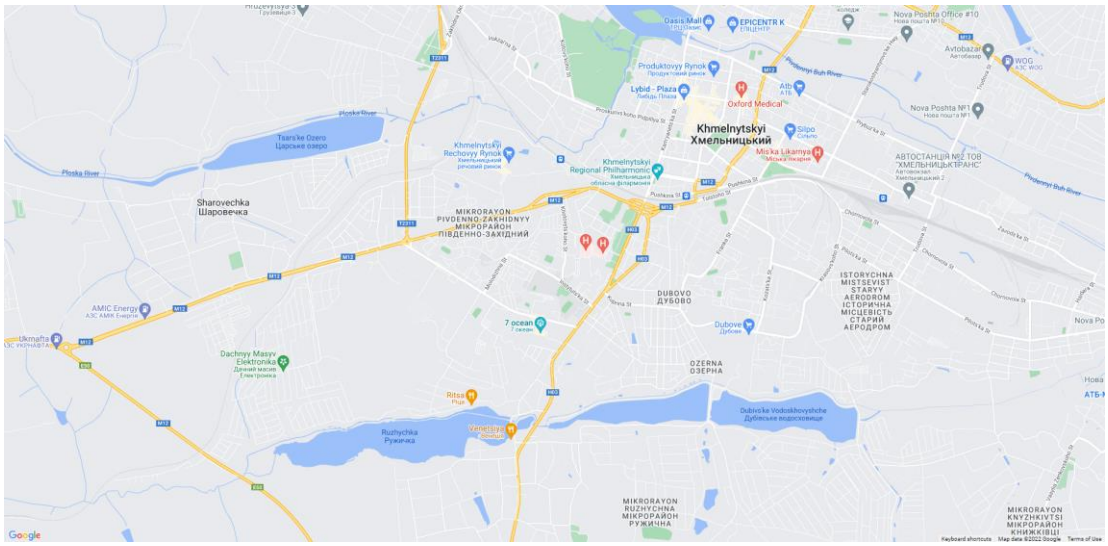


Рисунок 3.21 – Результат конфігурації Google API

Тепер настав час конфігурації GraphQL для клієнту. Перш за все, потрібна установка необхідних пакетів для GraphQL: npm install @apollo/client graphql. Пакет @apollo/client включає в себе все необхідне для конфігурації GraphQL, утиліти для виконання запитів та контекст даних скрізь весь застосунок. Пакет

graphql потрібен лиш для парсингу логіки GraphQL квері [10]. Конфігурація GraphQL дивовижно мінімалістична (рисунок 3.22): перший крок – це оголошення клієнту. Клієнт – це екземпляр класу ApolloClient. Все що потребує цей клас, це адрес backend серверу та конфігурацію кешування даних. Після чого цей клієнт передається в ApolloProvider, котрий забезпечує доступ до цього клієнту з будь-якої точки та компоненту проекту.

```
import { ApolloClient, InMemoryCache, ApolloProvider } from "@apollo/client";

const client = new ApolloClient({
    uri: process.env.REACT_APP_BACKEND_URL,
    cache: new InMemoryCache()
});

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </React.StrictMode>
);
```

Рисунок 3.22 – Конфігурація GraphQL

Стосовно кешування, це також один із надзвичайних козирів GraphQL, котре допомагає значно економити трафік. Конфігурація кешу складається лиш з ініціалізації класу InMemoryCache і все. Принцип роботи кешування продемонстрований на прикладі потоку роботи з квері запитами (рисунок 3.23). Продемонстрована ситуація коли різні сторінки на різних маршрутах проекту використовують одинакові квері запити:

1) клієнт робить запит на сервер;
2) сервер повертає відповідь клієнтові;
3) GraphQL автоматично вносить відповідь у кеш;
4) відбувається маршрутизація зі сторінки /home на сторінку /about;
5) замість того, щоб знову при квері запиті звертатись до серверу, GraphQL звернеться до кешу спершу. Якщо в кеші є необхідні дані, він використає їх, замість непотрібного додаткового запиту на сервер. Слід враховувати, що життєвий цикл

кешування припиняється при перезавантаженні сторінки, кеш очиститься й надалі будуть виконуватись квері запити на сервер.

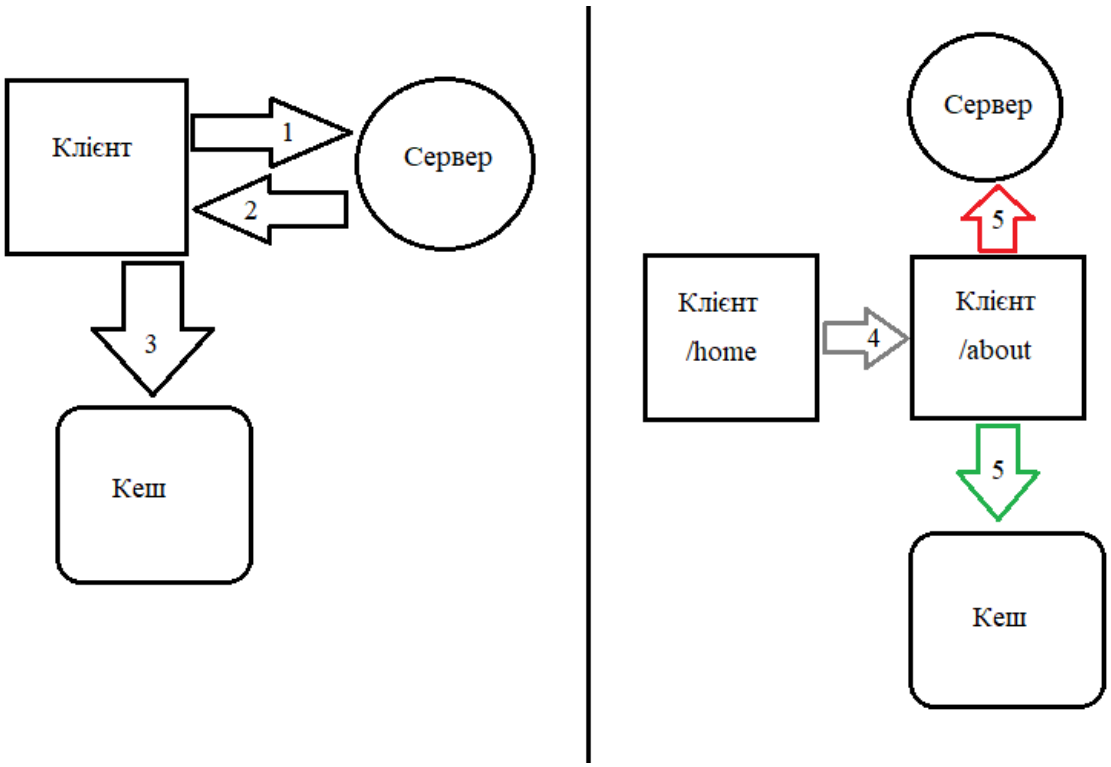


Рисунок 3.23 – Потік роботи квері запитів з кешуванням

При мутаціях відбувається схожий потік роботи. Вся відмінність в тому, що при мутаціях, котрі так чи інакше модифікують дані, кеш дозволяє вручну також модифікувати збережені квері-дані. Іншими словами, замість того, щоб чекати відповідь від серверу й тримати всі дані в кеші оновленими, GraphQL надає доступ до ручної модифікації кешу.

Після конфігурації GraphQL можна приступити до розробки перших інтерактивних елементів, а саме – компоненту, в якому демонструються всі доступні транспортні маршрути. Першим кроком, буде опис GraphQL квері (рисунок 3.24).

На рисунку 3.24 продемонстрований опис двох основних квері для потоку роботи проекту. Константа ALL_ROUTES описує звернення до квері routes, вона використовуватиметься для збору даних про наявні маршрути та відображення їх.

Константа GET_ROUTE описує квері route, котра уже витягує специфічну інформацію про конкретний маршрут.

```
import { gql } from '@apollo/client';

export const ALL_ROUTES = gql`
    query {
        routes {
            id
            routeId
            route
        }
    }
`;

export const GET_ROUTE = gql`
    query getRoute($routeId: ID) {
        route (routeId: $routeId) {
            id
            routeId
            route
            capacity
            transportNumber
            kind
            position
        }
    }
`;
```

Рисунок 3.24 – Опис GraphQL квері

Використовуватиметься цей запит при виборі певного маршруту. Вартує звернути увагу, що константа GET_ROUTE створила обгортку-псевдонім getRoute, яка слугує винятково для того, щоб прийняти як аргумент параметр routeId зі скалярним типом ID. Наступним фундаментальним кроком є створення контексту AppState (рисунок 3.25).

Контекст API в React дозволяє передавати дані крізь весь застосунок. Будь-який компонент, який виявився в області, яку огорнув цей контекст, зможе отримати напряму від нього дані. Контекст дозволяє запобігти анти-патерну props-drilling – тобто ситуації, коли програміст вимушений передавати аргумент із компонента в компонент по дереву компонентів, щоб певні дані могли дістатись якийсь вкладений компонент. Також було створено React-хук useAppContext для зручного звернення до даних контексту.

```
const AppContext = createContext({});

const AppProvider = ({ children }) => {
    const routes = useQuery(ALL_ROUTES)

    return (
        <AppContext.Provider value={{ routes }}>
            { children }
        </AppContext.Provider>
    )
}

const useAppContext = () => useContext(AppContext);
```

Рисунок 3.25 – Контекст AppState

Використання функції useQuery з аргументом ALL_ROUTES (константа, яка містить в собі тіло запиту квері routes) означає, що запит на відповідну квері буде виконано автоматично. Не потрібно ніяким чином ініціювати запит, про все подбає GraphQL API. Якщо відкрити Developer Tools в браузері на вкладці Network, можна побачити, що запит на квері route успішно виконався автоматично. Навіть можна побачити які дані повернув цей запит (рисунок 3.26).

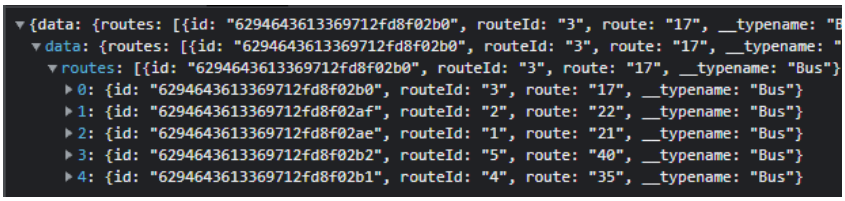


Рисунок 3.26 – Інформація про запит у вкладці Network

Згідно з дизайном, проект повинен мати дві кнопки: одна для того, щоб відкрити список доступних маршрутів, друга – для відкриття налаштувань застосунку. Для цього вартує створити батьківський компонент Navbar, котрий займеться організацією цих кнопок та відповідних спливаючих вікон (рисунок 3.27).

Рисунок 3.27 – Структура компоненту Navbar

Компоненти Material UI допоможуть з верстkою [16]: використані бібліотечні компоненти іконок для кнопок, та сам компонент кнопки. Компонент Navbar являється всього обгорткою для кнопок відкриття модальних вікон. Логіка також доволі проста: компонент містить лише стани, які відповідають поточному статусу для відповідних модальних вікон. Тепер розглянемо компонент SelectBusPopup.

Цей компонент являється модальним вікном вибору потрібного маршруту, тому його логіка наступна (рисунок 3.28):

1) створити модальне вікно;
2) взяти необхідні дані про маршрути із контексту AppState;
3) якщо дані уже завантажились та готові для використання, за допомогою методу map ітерувати масив маршрутів і на кожній ітерації відобразити кнопку відповідного маршруту;
4) при натисканні на певну кнопку робити запит на сервер для того, щоб дістати уже повну інформацію про цей маршрут;
5) отримані дані зберегти в глобальний стан контексту AppState.

Рисунок 3.28 – Потік даних SelectBusPopup

Принцип роботи SettingsPopup подібний до SelectBusPopup. Компонент являється модальним вікном, в якому вже знаходиться меню налаштувань для запитів та функціоналу проекту в цілому (рисунок 3.29):

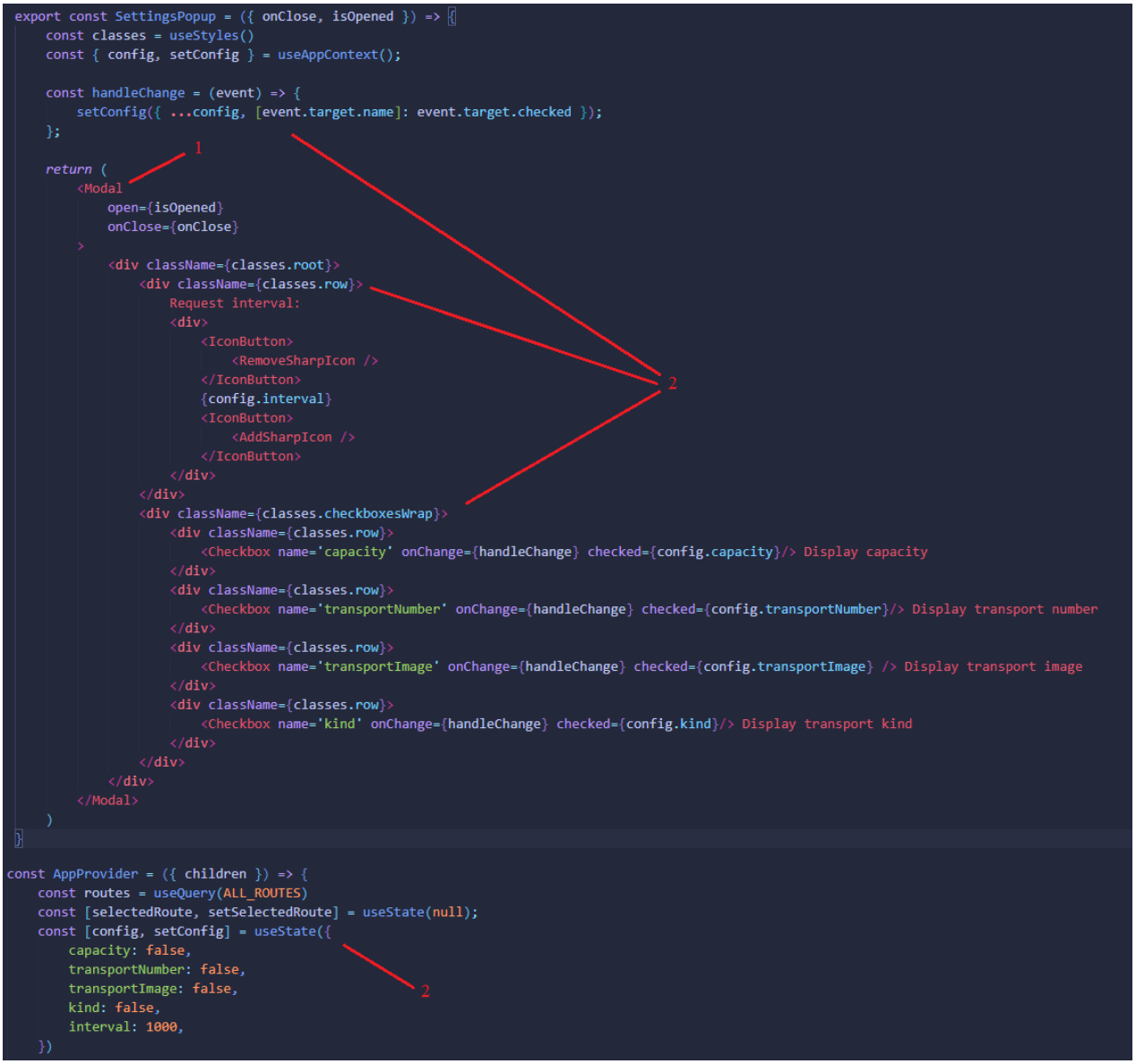


Рисунок 3.29 – Потік роботи SettingsPopup

1) створюється модальне вікно;
2) робота з контекстом:
   a) створюється та передається в контексті стан config;
   b) кожна опція config регулюється обробником handleChange, котрий змінює стан кожного із параметрів на true або false. Іншими словами, вмикає або вимикає той чи інший параметр конфігурації.

Реалізація динамічного тіла запиту квері GraphQL з врахуванням локального стану AppState зображена на рисунку 3.30:

1) оголошення стану queryBody, котре буде містити тіло запиту в собі;

2) на кожну зміну стану config React хук useEffect реагує та згідно цього стану будує тіло запиту й записує в стан queryBody;

3) в SelectBusPopup локально оголошуємо основну частину квері – обгортку, котра приймає routeId та деякі фундаментальні поля, як позиція, унікальний ідентифікатор та поєднуємо з динамічною частиною запиту – queryBody;

4) використання створеної квері.

Рисунок 3.30 – Реалізація динамічного тіла запиту

Все що залишилось реалізувати на стороні клієнта:

1) візуалізація маршруту транспорта;

2) візуалізацію поточної позиції транспорту;

3) відображення повної інформації про транспорт.

Реалізація візуалізації маршруту транспорта зображено на рисунку 2.31.

```
  const drawPath = async (google) => {
      if (!selectedRoute?.position) return;

      const directionsService = new google.maps.DirectionsService();
      const directionsRenderer = new google.maps.DirectionsRenderer();

      const directions = await directionsService.route({
          waypoints: selectedRoute.position,
          optimizeWaypoints: true,
      });
      directionsRenderer.setDirections(directions);
  };

useEffect(() => {
  drawPath(google);

  // eslint-disable-next-line react-hooks/exhaustive-deps
}, [config, selectedRoute]);
```

Рисунок 3.31 – Функція візуалізації маршруту

Функція drawPath ініціалізує екземпляри класів DirectionsService та DirectionsRenderer. DirectionsService відповідає за підготовку даних для візуалізації. Параметром waypoints для обробки екземпляром DirectionsRenderer передається поле position, яке містить в собі дані про можливі позиції, вибраного маршруту. Результат обробки передається в метод setDirections екземпляру DirectionsRenderer, який зрештою візуалізує на карті маршрут по заданим точкам. Функція drawPath буде виконувати цю візуалізацію в React хукові useEffect при кожній зміні стану config або selectedRoute. Іншими словами, при кожній зміні конфігурацію користувачем, або при виборі іншого маршруту.

Візуалізація поточного положення транспорту відбувається за допомогою компоненту Marker (Рисунок 3.32), який міститься в пакеті google-maps-react.

```
{selectedRoute && <Marker
  title={selectedRoute?.route || ''}
  name={selectedRoute?.route || ''}
  position={selectedRoute.position[selectedRoute.positionIndex] || null}
  onClick={onMarkerClick}
  icon={{
    url: DirectionsBusIcon,
  }}
/>}
```

Рисунок 3.32 – Використання Marker

Компонент Marker буде відображатись тільки якщо наразі є вибраний маршрут (selectedRoute не рівний null). Цей компонент приймає title та name які рівні назві маршруту транспорту. Ці параметри необхідні для підсказки, котра появляється при наведенні миші на цей маркер. Важливим моментом є те, що при клікові на Marker спрацює функція onMarkerClick, яка відкриє модальне вікно з інформацією про цей маршрут. Модальне вікно з інформацією про маршрут реалізовано в компоненті CurrentRoutePopup (рисунок 3.33).

```
export const CurrentRoutePopup = ({ isOpened, onClose}) => {
    const classes = useStyles()
    const { selectedRoute } = useAppContext();

    return (
        <Modal
            open={isOpened}
            onClose={onClose}
        >
            <div className={classes.root}>
                <div className={classes.body}>
                    {selectedRoute?.transportImage && (
                        <div className={classes.dataSection}>
                            <img src={selectedRoute.transportImage} alt={selectedRoute.route}/>
                            <p>{selectedRoute.route}</p>
                        </div>
                    )}
                    <div className={classes.dataSection}>
                        {selectedRoute?.capacity && <p>Capacity {selectedRoute.capacity}</p>}
                        {selectedRoute?.transportNumber && <p>Transport number {selectedRoute.transportNumber}</p>}
                        {selectedRoute?.kind && <p>Kind {selectedRoute.kind}</p>}
                        {selectedRoute?.route && <p>Route {selectedRoute.route}</p>}
                    </div>
                </div>
            </div>
        </Modal>
    )
```

Рисунок 3.33 – Компонент CurrentRoutePopup

Компонент CurrentRoutePopup дуже примітивний, та слугує лиш для візуалізацію даних, які зберігаються в глобальному стані selectedRoute. Єдина логіка, яка виконується в цьому компоненті – це перевірка наявності певних полів. Оскільки користувач може вимикати поля для запитів на сервер, заради економії трафіку, то є ризик, що те чи інше поле було вимкнуто і просто не збережене в цьому стані.

Тому перед відображенням даних, проводяться перевірки чи ці дані взагалі наявні на даний момент.

3.3 Висновки

В цьому розділі було описано процес розробки серверу, продемонстровано плюси та мінуси різних підходів, обговорено найкращі практики. Проведено інтеграцію двух різних серверів, а саме Express та GraphQL, налаштовано БД MongoDB. Також, було розроблено клієнтську частину, детально описані UI та UX, проведена робота з Google API, побудовано такі концепції, як динамічні тіла запитів GraphQL.

ВИСНОВКИ

Результатом виконання дипломної роботи є розробка сервісу для моніторингу трафіку в системі «Розумне місто». Була виконана величезна робота: від дослідження варіантів та принципів моніторингу трафіку до планування та розробки архітектури застосунку. На основі власного та також уже здобутого протягом досліджень досвіду був спроектований робочий процес.

Основа проекту – це сервер та клієнт. Зі сторони серверу, було налаштовано кілька резолверів, які працювали паралельно та взаємодіяли один з одним по мірі необхідності. Було налаштовано Базу Даних та спроектовано потік даних. Зі сторони клієнту, було розроблено зовнішній каркас застосунку, розроблено механізми реалізації динамічних тіл запитів GraphQL.

Сервіс моніторингу трафіку призначений для покращення та полегшення життя в абстрактному «Розумному місті», це незамінний інструмент в міському потокові життя. Оптимізація процесів, пов’язаних з громадським транспортом, – це необхідна умова в будь-якому місті.

Сервіс використовує передові технології, як GraphQL, для оптимізації роботи та економії трафіку, що значно покращує користувацький досвід застосунку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Бенкс А., Порселло Є. GraphQL. Мова запитів для сучасних веб-застосунків. США: O'Reilly, 2019. С. 100 – 150.

2. Браун І. Веб-розробка з використанням Node та Express. Повноцінне використання стеку JavaScript. США: O'Reilly, 2021. С. 15 – 90.

3. Васильїв О. JavaScript в прикладах та задачах. Харків: Ранок, 2021. С. 180 – 183.

4. GraphQL documentation. URL: https://graphql.org/code/#javascript (дата звернення: 03.04.2022).

5. NodeJS 14.x documentation. URL: https://nodejs.org/docs/latest-v14.x/api/ (дата звернення: 04.04.2022).

6. Сучасник підручник JavaScript. URL: https://uk.javascript.info/ (дата звернення 14.03.2022).

7. MDN. URL: https://developer.mozilla.org/en-US/ (дата звернення 12.03.2022).

8. Express documentation. URL: https://expressjs.com/ru/starter/installing.html (дата звернення 12.03.2022).

9. MongoDB documentation. URL: https://www.mongodb.com/docs/manual/tutorial/getting-started/ (дата звернення 14.04.2022).

10. Apollo Client documentation. URL: https://www.apollographql.com/docs/react/ (дата звернення 13.04.2022).

11. Документація React. URL: https://uk.reactjs.org/docs/getting-started.html (дата звернення 15.05.2022).

12. Google Maps API. URL: https://developers.google.com/maps/documentation/javascript (дата звернення 17.04.2022).

13. Running an Express and GraphQL server. URL: https://graphql.org/graphql-js/running-an-express-graphql-server/ (дата звернення 09.03.2022).

14. Mongoose documentation. URL: https://mongoosejs.com/docs/guide.html (дата звернення 08.03.2022).

15. SASS documentation. URL: https://sass-lang.com/documentation/ (дата звернення 29.05.2022).

16. Material UI documentation. URL: https://v4.mui.com/getting-started/usage/ (дата звернення 30.05.2022).

17. Material UI JSS. URL: https://v4.mui.com/styles/advanced/ (дата звернення 30.05.2022).

18. The history of NodeJS. URL: https://www.section.io/engineering-education/history-of-nodejs/ (дата звернення 09.03.2022).

19. Банкер К. MongoDB в дії. США: O'Reilly, 2018. С. 99 – 105.

20. GraphQL vs Rest APIs. URL: https://graphcms.com/blog/graphql-vs-rest-apis (дата звернення 29.05.2022).

21. Zhang R. A network tomography approach for traffic monitoring in smart cities. США: MUST, 2018. С. 4 – 48.

22. Поллард Б. HTTP в дії. США: Manning, 2021. С. 10 – 28.

23. Меджуї М., Уайлд Е., Мітра Р., Амундсен М. Безперервний розвиток API. Правильні рішення в динамічному технологічному ландшафті. США: O'Reilly, 2020. С. 5 – 100.

24. Хаверберке М. Виразний JavaScript. Сучасне програмування. Україна: Київ, 2021. С. 10 – 190.

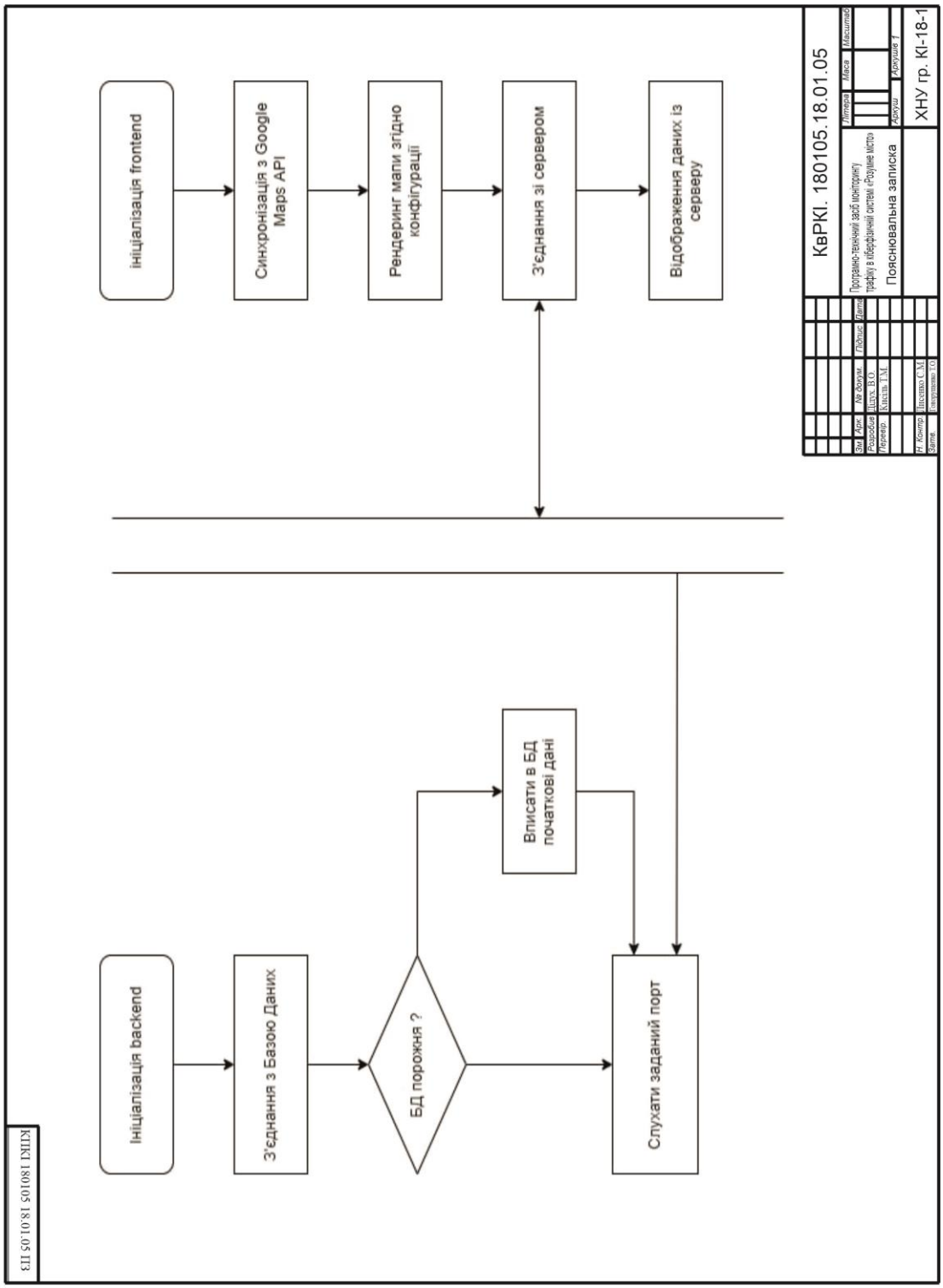
25. Бредшоу Ш., Брезіл Й., Ходоров К. MongoDB, повна інструкція. США: O'Reilly, 2020. С. 50 – 69.

26. Styled Components documentation. URL: https://styled-components.com/docs (дата звернення 20.05.2022).

Додаток А

(обов’язковий)

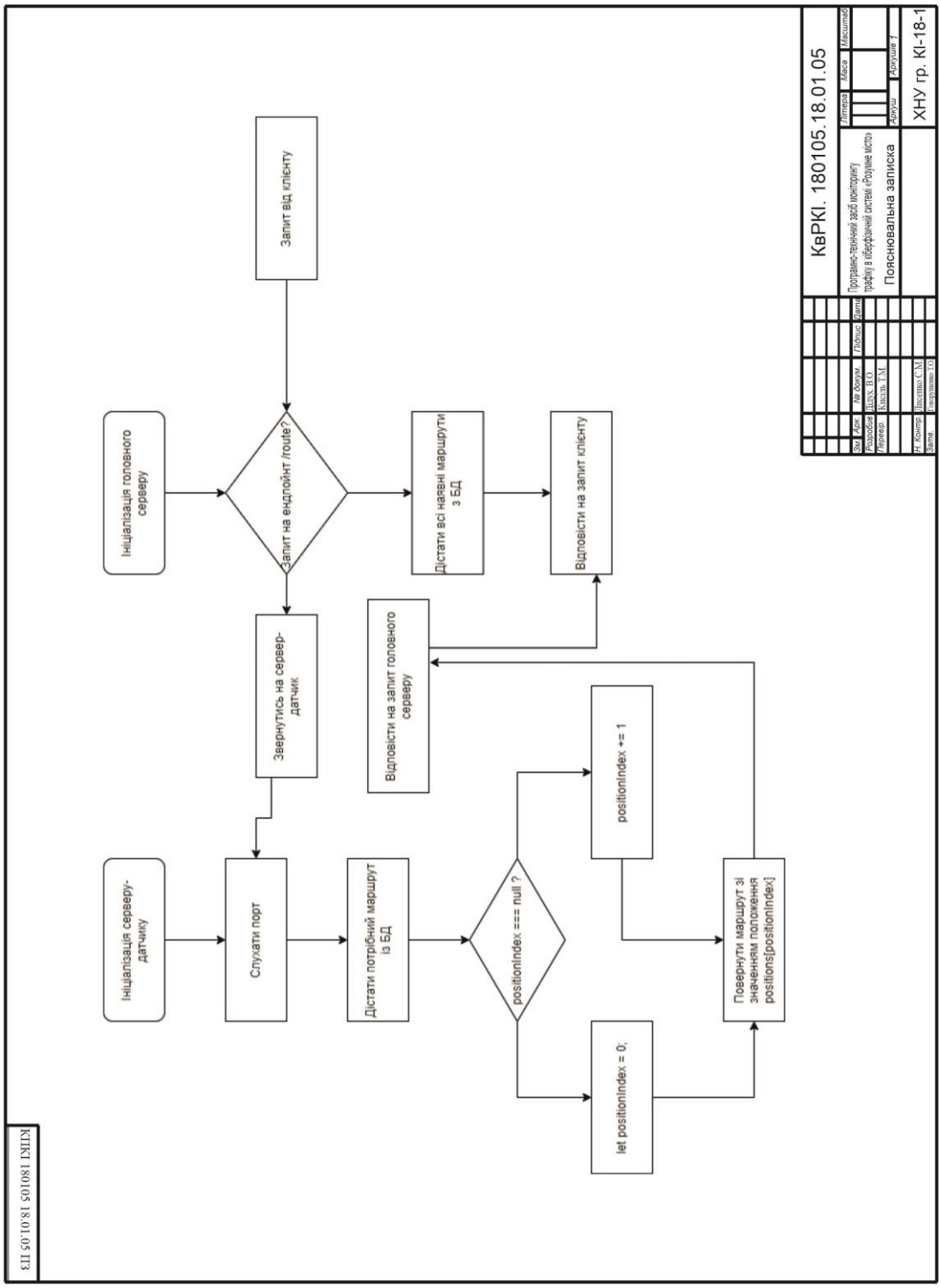
Копія креслення «Блок-схема роботи застосунку»



Додаток Б

(обов’язковий)

Копія креслення «Блок-схема вичислення позиції маршруту зі сторони серверу»



Додаток В

(обов’язковий)

Копія креслення «Інтерфейс застосунку»

Додаток Г

(обов’язковий)

Лістинг коду

backend/index.js

```
require('dotenv').config();
const express = require('express');
const { data } = require('./assets/data');
const { graphqlHTTP } = require('express-graphql');
const { schema } = require('./schema');
const { resolvers } = require('./resolvers');
const mongoose = require('mongoose');
const Bus  = require('./models/bus');
const cors = require('cors');

const password = process.env.MONGO_PASSWORD
const username = process.env.MONGO_USERNAME;

const db =
`mongodb+srv://${username}:${password}@cluster0.umhoc.mongodb.net/?retryWrites=true&w=majority`;

const busSensorSever = express();
const mainServer = express();

const BUS_SENSOR_PORT = process.env.BUS_SENSOR_PORT || 4000;
const MAIN_SENSOR_PORT = process.env.MAIN_SENSOR_PORT || 4001;

const init = async () => {
   await mongoose.connect(db, { useNewUrlParser: true, useUnifiedTopology: true });

   const bus = await Bus.findOne({ route: '21' });

   if (!bus) {
      await Bus.insertMany(data);
   }

   mainServer.listen(MAIN_SENSOR_PORT, () => {
      console.log('main server on air')
   });
}

busSensorSever.get('/route/:routeId', async (request, response) => {
   const routeId = request.params.routeId;
   const bus = await Bus.findOne({ routeId });

   response.json(bus);
```

```
})

busSensorSever.listen(BUS_SENSOR_PORT, () => {
   console.log('bus sensor server on air')
})

mainServer.use('/graphql', cors(), graphqlHTTP({
   graphiql: true,
   schema,
   rootValue: resolvers,
   cors: {
      origin: ["http://localhost:3000"]
   }
}))

init();
```

backend/models/bus.js

```
const { default: mongoose, Schema } = require('mongoose');

const busSchema = Schema({
   route: String,
   capacity: String,
   transportNumber: Number,
   kind: String,
   position: [Number],
   routeId: String
});

const Bus = mongoose.model('Bus', busSchema);

module.exports = Bus;
```

backend/resolvers.js

```
const axios = require('axios').default;
const Bus = require('./models/bus');

const BUS_SENSOR_PORT = process.env.BUS_SENSOR_PORT || 4000;

const resolvers = {
   routes: async () => {
      const buses = await Bus.find({});

      return buses;
   },
   route: async ({ routeId }) => {
      const busResponse = await
axios.get(`http://localhost:${BUS_SENSOR_PORT}/route/${routeId}`);
```

```
        return busResponse.data;
    },
}

module.exports.resolvers = resolvers;
```

backend/schema.js

```
const { buildSchema } = require('graphql');

const schema = buildSchema(`
    type Bus {
        id: ID
        route: String
        capacity: String
        transportNumber: Int
        kind: String
        position: [Int]
        routeId: String
    }
    type Query {
        routes: [Bus]
        route(routeId: ID): Bus
    }
`);

module.exports.schema = schema;
```

frontend/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { AppProvider } from './context/AppState';
import { ApolloClient, InMemoryCache, ApolloProvider } from "@apollo/client";

const client = new ApolloClient({
    uri: process.env.REACT_APP_BACKEND_URL + '/graphql',
    cache: new InMemoryCache()
});

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <ApolloProvider client={client}>
      <AppProvider>
        <App />
      </AppProvider>
    </ApolloProvider>
  </React.StrictMode>
);
```

frontend/queries.js

```
import { gql } from '@apollo/client';

export const ALL_ROUTES = gql`
   query {
      routes {
         id
         routeId
         route
      }
   }
`;

export const GET_ROUTE = gql`
   query getRoute($routeId: ID) {
      route (routeId: $routeId) {
         id
         routeId
         route
         capacity
         transportNumber
         kind
         position
      }
   }
`;
```

frontend/app.js

```
import { useEffect, useState } from 'react';
import { Map, GoogleApiWrapper, Marker } from 'google-maps-react';
import { makeStyles } from '@material-ui/core/styles';
import { Navbar } from './components/Navbar';
import { useAppContext } from './context/AppState';
import { CurrentRoutePopup } from './components/CurrentRoutePopup';
import DirectionsBusIcon from '@material-ui/icons/DirectionsBus';

const useStyles = makeStyles({
  '@global': {
    '*': {
      boxSizing: 'border-box',
      margin: 0,
    },
    '#root': {
      height: '100vh',
      width: '100vw',
    }
  }
});
```

```
function App(props) {
  const { google } = props;
  const { config, drawPath, selectedRoute } = useAppContext();
  const [isCurrentRoutePopup, setIsCurrentRoutePopup] = useState(false);

  useStyles();

  useEffect(() => {
    drawPath(google);

    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [config, selectedRoute]);

  const onMarkerClick = () => setIsCurrentRoutePopup(true)

  return (
    <div>
      <Navbar />
      <CurrentRoutePopup
        onClose={() => setIsCurrentRoutePopup(false)}
        isOpened={isCurrentRoutePopup}
      />
      <Map
        google={google}
        zoom={14}
        initialCenter={{
          lat: 49.409358507364864,
          lng: 26.960622647423083
        }}
        disableDoubleClickZoom
        draggable={false}
        scrollwheel={false}
        disableDefaultUI
      >
        {selectedRoute && <Marker
          title={selectedRoute?.route || ''}
          name={selectedRoute?.route || ''}
          position={selectedRoute.position[selectedRoute.positionIndex] || null}
          onClick={onMarkerClick}
          icon={{
            url: DirectionsBusIcon,
            scaledSize: new google.maps.Size(15,15)
          }}
        />}
      </Map>
    </div>
  );
}

export default GoogleApiWrapper({
  apiKey: (process.env.REACT_APP_GOOGLE_API_KEY),
```

```
})(App);
```

frontend/context/AppState.js

```
import { createContext, useContext, useState, useEffect } from 'react';
import { useQuery } from '@apollo/client';
import { ALL_ROUTES } from '../queries';

const AppContext = createContext({});

const AppProvider = ({ children }) => {
  const routes = useQuery(ALL_ROUTES)
  const [selectedRoute, setSelectedRoute] = useState(null);
  const [config, setConfig] = useState({
    capacity: true,
    transportNumber: true,
    // transportImage: true,
    kind: true,
    interval: 1000,
  })
  const [queryBody, setQueryBody] = useState('');

  useEffect(() => {
    let newQuery = '';

    for (const param in config) {
      if (!config[param]) {
        newQuery += `
          ${param}
        `
      }
    }

    setQueryBody(newQuery)
  }, [config])

  const drawPath = async (google) => {
    if (!selectedRoute?.position) return;

    const directionsService = new google.maps.DirectionsService();
    const directionsRenderer = new google.maps.DirectionsRenderer();

    const directions = await directionsService.route({
      waypoints: selectedRoute.position,
      optimizeWaypoints: true,
    });
    directionsRenderer.setDirections(directions);
  };

  return (
    <AppContext.Provider value={{
```

```
            routes,
            setSelectedRoute,
            selectedRoute,
            config,
            setConfig,
            queryBody,
            drawPath,
        }}>
            { children }
        </AppContext.Provider>
    )
}

const useAppContext = () => useContext(AppContext);

export { AppContext, AppProvider, useAppContext }
```

frontend/components/CurrentRoutePopup.js

```
import { makeStyles } from '@material-ui/core/styles';
import { Modal } from '@material-ui/core';
import { useAppContext } from '../context/AppState';

const useStyles = makeStyles({
    root: {
        width: '50vw',
        height: '50vh',
        borderRadius: 10,
        display: 'flex',
        justifyContent: 'center',
        alignItems: 'center',
        background: '#fff',
        flexDirection: 'column',
    },
    body: {
        display: 'flex',
    },
    dataSection: {
        display: 'flex',
        flexDirection: 'column',
    }
});

export const CurrentRoutePopup = ({ isOpened, onClose}) => {
    const classes = useStyles()
    const { selectedRoute } = useAppContext();

    return (
        <Modal
            open={isOpened}
            onClose={onClose}
```

```
      >
        <div className={classes.root}>
          <div className={classes.body}>
            {selectedRoute?.transportImage && (
              <div className={classes.dataSection}>
                <img src={selectedRoute.transportImage} alt={selectedRoute.route}/>
                <p>{selectedRoute.route}</p>
              </div>
            )}
            <div className={classes.dataSection}>
              {selectedRoute?.capacity && <p>Capacity {selectedRoute.capacity}</p>}
              {selectedRoute?.transportNumber && <p>Transport number
{selectedRoute.transportNumber}</p>}
              {selectedRoute?.kind && <p>Kind {selectedRoute.kind}</p>}
              {selectedRoute?.route && <p>Route {selectedRoute.route}</p>}
            </div>
          </div>
        </div>
      </Modal>
    )
}
```

frontend/components/Navbar.js

```
import React, { useState } from 'react';
import { makeStyles } from '@material-ui/core/styles';
import { IconButton } from '@material-ui/core';
import DirectionsBusIcon from '@material-ui/icons/DirectionsBus';
import SettingsIcon from '@material-ui/icons/Settings';
import { SelectBusPopup } from './SelectBusPopup';
import { SettingsPopup } from './SettingsPopup';

const useStyles = makeStyles({
  root: {
    width: '100%',
    display: 'flex',
    justifyContent: 'space-between',
    position: 'absolute',
    zIndex: 100,
    top: 0,
    padding: '20px'
  },
});

export const Navbar = () => {
    const classes = useStyles();
    const [selectBusPopup, setSelectBusPopup] = useState(false);
    const [settingsPopup, setSettingPopup] = useState(false);

    return (
      <div className={classes.root}>
```

```
            <IconButton onClick={() => setSelectBusPopup(true)}>
               <DirectionsBusIcon />
            </IconButton>
            <SelectBusPopup
               isOpened={selectBusPopup}
               onClose={() => setSelectBusPopup(false)}
            />
            <IconButton onClick={() => setSettingPopup(true)}>
               <SettingsIcon />
            </IconButton>
            <SettingsPopup
               isOpened={settingsPopup}
               onClose={() => setSettingPopup(false)}
            />
         </div>
      )
}
```

frontend/components/SelectBusPopup.js

```
import { Modal, Button  } from '@material-ui/core';
import { useAppContext } from '../context/AppState';
import { makeStyles } from '@material-ui/core/styles';
import { useLazyQuery } from "@apollo/client";
import { gql } from '@apollo/client';

const useStyles = makeStyles({
   root: {
      width: '50vw',
      height: '50vh',
      borderRadius: 10,
      display: 'flex',
      justifyContent: 'center',
      alignItems: 'center',
      background: '#fff',
   },
   buttonsRow: {
      display: 'flex',
      padding: 10,
      width: '100%',
      justifyContent: 'center',

      '& button': {
         marginRight: 10
      }
   }
 });

export const SelectBusPopup = ({ isOpened, onClose }) => {
   const {
      routes,
```

```
    setSelectedRoute,
    queryBody
  } = useAppContext();
  const classes = useStyles()
  const query = gql`
    query getRoute($routeId: ID) {
      route (routeId: $routeId) {
        id
        routeId
        route
        position
        ${queryBody}
      }
    }
  `
  const [loadBus] = useLazyQuery(query, {
    onCompleted: ({ route }) => {
      setSelectedRoute(route);
      onClose();
    }
  });

  return (
    <Modal
      open={isOpened}
      onClose={onClose}
    >
      <div className={classes.root}>
        {
          routes?.loading ? 'Loading...' : (
            <div className={classes.buttonsRow}>
              {routes?.data?.routes.map(({ route, routeId }) => (
                <Button
                  color='primary'
                  variant='contained'
                  key={routeId}
                  onClick={() => {
                    loadBus({ variables: { routeId }})
                  }}
                >
                  { route }
                </Button>
              ))}
            </div>
          )
        }
      </div>
    </Modal>
  )
}
```

frontend/components/SettingsPopup.js

```
import { makeStyles } from '@material-ui/core/styles';
import { Modal, IconButton, Checkbox   } from '@material-ui/core';
import AddSharpIcon from '@material-ui/icons/AddSharp';
import RemoveSharpIcon from '@material-ui/icons/RemoveSharp';
import { useAppContext } from '../context/AppState';

const useStyles = makeStyles({
   root: {
      width: '50vw',
      height: '50vh',
      borderRadius: 10,
      display: 'flex',
      justifyContent: 'center',
      alignItems: 'center',
      background: '#fff',
      flexDirection: 'column',
   },
   row: {
      display: 'flex',
      justifyContent: 'center',
      alignItems: 'center',
   },
   checkboxesWrap: {
      display: 'flex',
      flexDirection: 'column',
      alignItems: 'baseline',
   }
 });

export const SettingsPopup = ({ onClose, isOpened }) => {
   const classes = useStyles()
   const { config, setConfig } = useAppContext();

   const handleChange = (event) => {
      setConfig({ ...config, [event.target.name]: event.target.checked });
   };

   return (
      <Modal
         open={isOpened}
         onClose={onClose}
      >
         <div className={classes.root}>
            <div className={classes.row}>
               Request interval:
               <div>
                  <IconButton>
                     <RemoveSharpIcon />
                  </IconButton>
```

```
                {config.interval}
                <IconButton>
                  <AddSharpIcon />
                </IconButton>
              </div>
            </div>
            <div className={classes.checkboxesWrap}>
              <div className={classes.row}>
                <Checkbox name='capacity' onChange={handleChange} checked={config.capacity}/>
Display capacity
              </div>
              <div className={classes.row}>
                <Checkbox name='transportNumber' onChange={handleChange}
checked={config.transportNumber}/> Display transport number
              </div>
              {/* <div className={classes.row}>
                <Checkbox name='transportImage' onChange={handleChange}
checked={config.transportImage} /> Display transport image
              </div> */}
              <div className={classes.row}>
                <Checkbox name='kind' onChange={handleChange} checked={config.kind}/> Display
transport kind
              </div>
            </div>
          </div>
        </Modal>
    )
}
```

UNICHECK

Ім'я користувача:
Кафедра КІ

ID перевірки:
1011567245

Дата перевірки:
13.06.2022 20:14:41 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
13.06.2022 20:14:59 EEST

ID користувача:
100005591

Назва документа: **Дідух_Програмно-технічний засіб моніторингу трафіку в кіберфізичній системі «Розумне м...**

Кількість сторінок: **63** Кількість слів: **9334** Кількість символів: **66555** Розмір файлу: **2.10 MB** ID файлу: **1011438111**

1.14%
Схожість

Найбільша схожість: **0.71%** з джерелом з Бібліотеки (ID файлу: **1011396088**)

0.34% Джерела з Інтернету	6	Сторінка 65
1.03% Джерела з Бібліотеки	86	Сторінка 65

0.06% Цитат

Цитати	1	Сторінка 66

Не знайдено жодних посилань

0%
Вилучень

Немає вилучених джерел

Tue Jun 14 12:04:36 EEST 2022, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальное совпадение с одним документом 1.0%

Словари проверки: en_US, ru_RU, ua_UA. **Ошибок в документах: 13%**

ID: 105214 Название: Програмно-технічний засіб моніторингу трафіку в кіберфізичній системі «Розумне місто» Добавлено в БД: 2022-06-14 Авторы: В.О. Дідух Руководители: Т.М. Кисіль Консультанты: Опоненты:	Документ		Суммарное совпадение по Базе Данных	
	Символы	Лексемы	Символы	Лексемы
	59632	561	499 (1%)	7 (1%)

Источник плагиата

ID	Описание	Наличие плагиата в документе	
		Символы	Лексемы

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Дідух Владислав Олександрович

Тема: Програмно-технічний засіб моніторингу трафіку в кіберфізичній системі «Розумне місто»

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 Кількість сторінок записки 61

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є створення засобу для моніторингу трафіку в кіберфізичній системі «Розумне місто».

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки, і техніки, і передових методів роботи: В першому розділі кваліфікаційної роботи проведено дослідження предметної області (проаналізовано всі можливі варіанти та підходи до реалізації бажаного результату кваліфікаційної роботи) та виконано постановку задачі дослідження. Також, був описаний життєвий цикл розробки проекту. В другому розділі кваліфікаційної роботи проведено дослідження на предмет використання технологій: їхній опис; обґрунтування доцільності використання того чи іншого інструменту; порівняння технологій. В третьому розділі кваліфікаційної роботи було описано процес розробки проекту. Було детально об'яснено як працюють різні інструменти; продемонстровано їхню ефективність та найкращі практики розробки подібних застосунків.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: відсутні.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. Інші зауваження: _____

9. Оцінка дипломної роботи: добре

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) Гурман І.В. к.т.н., доцент кафедри інженерія програмного забезпечення

"14" червня 2022 р. _____ (підпис)

Завідувачу кафедри КІІС
д-ру техн.наук, проф. Говорущенко Т. О.

Дідух В.О.

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ-18-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність плагіату ознайомлений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

_____ _____

дата підпис

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ
КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Програмно-технічний засіб моніторингу трафіку в кіберфізичній системі «Розумне місто»

Автор: Дідух Владислав Олександрович

Спеціальність: 123-Комп'ютерна інженерія

Освітня програма: освітньо-професійна

Науковий керівник: Кисіль Тетяна Миколаївна, к.ф-м.н., доцент

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, не є плагіатом, вони законні, тому що:

1) усі запозичені фрагментальні, або мають оформленні посилання;
2) всі зафіксовані зміни тексту відносяться до комбінування латинських символів з україномовними скороченнями і не є модифікацією тексту;
3) запозичення, які мають місце в розділах аналізу існуючих аналогів та прототипів, не описують безпосередньо авторське дослідження і не стосуються результатів роботи.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості, складає 1.14% і адресується до 92 першоджерел, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ Т.М. Кисіль

Гарант ОП _____ С.М. Лисенко

Завідувач кафедри КІІС _____ Т.О. Говорущенко