**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**Чорноморський національний університет імені Петра Могили**

**Факультет комп'ютерних наук**

**Кафедра інтелектуальних інформаційних систем**

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____»_____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

# СИСТЕМА ОНЛАЙН-ЗАМОВЛЕНЬ ПОЇЗДОК НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

***Здобувач*** _____Олександр ТИМКІВ

«___»_____2026 р.

***Керівник*** канд. техн. наук, доцент _____Євген СІДЕНКО

«___»_____2026 р.

**Миколаїв – 2026**

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

| | |
|---|---|
| Факультет | Комп’ютерних наук |
| Кафедра | Інтелектуальних інформаційних систем |
| Рівень вищої освіти | Перший (бакалаврський) |
| Освітній ступень | Бакалавр |
| Спеціальність | 122 Комп’ютерні науки |
| Освітня програма | Комп’ютерні науки |

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Євген СІДЕНКО

«____» _____ 20__ р.

**ЗАВДАННЯ**
**на кваліфікаційну роботу здобувача**

**Тимків Олександр Олександрович**
(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: **Система онлайн-замовлень поїздок на основі мікросервісної архітектури**.

Керівник роботи: Сіденко Євген Вікторович, завідувач кафедри інтелектуальних інформаційних систем, канд. техн. наук, доцент.
*(прізвище, ім’я, по батькові, посада, науковий ступінь, вчене звання)*

Затверджена наказом ЧНУ ім. Петра Могили від «25» грудня 2025 р. № 353.

2. Строк представлення кваліфікаційної роботи «___» _____ 20__ р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: система мікросервісної інформаційної системи онлайн-замовлення поїздок; початкові дані: функціональні вимоги до платформи, бізнес-правила тарифікації, тестові масиви просторово-часових даних (GPS-координати) та специфікації протоколів обміну даними.

4. Перелік питань, що підлягають розробці: аналіз предметної області та існуючих платформ онлайн-замовлення поїздок; обґрунтування вибору мікросервісної архітектури, технологічного стека та патернів забезпечення відмовостійкості; проєктування загальної архітектури системи та гетерогенної структури баз даних (Polyglot Persistence); програмна реалізація цільових мікросервісів (управління профілями, оркестрація поїздок, обробка геоданих у реальному часі та асинхронний фінансовий білінг); проведення навантажувального тестування розроблених модулів та аналіз ефективності прийнятих архітектурних рішень.

5. Перелік графічних матеріалів: презентація.

**Керівник роботи** _____ Євген СІДЕНКО
*(Особистий підпис)* *(Власне ім'я ПРІЗВИЩЕ)*

**Здобувач** _____ Олександр ТИМКІВ
*(Особистий підпис)* *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «21» грудня 2025 р.

**КАЛЕНДАРНИЙ ПЛАН**
**кваліфікаційної роботи**

Тема: **Система онлайн-замовлень поїздок на основі мікросервісної архітектури.**

| № | Найменування роботи | Початок | Закінчення | Примітки |
|---|---|---|---|---|
| 1 | Отримання завдання на виконання КР | 21.12.2025 | 24.12.2025 | Виконано |
| 2 | Аналіз предметної області та постановка задачі | 25.12.2025 | 30.01.2026 | Виконано |
| 3 | Огляд літературних джерел за темою кваліфікаційної роботи, зокрема огляд публікацій та аналогічних систем, щодо системи онлайн-замовлень поїздок | 31.01.2026 | 01.03.2026 | Виконано |
| 4 | Огляд існуючих архітектур та підходів для вирішення поставленої задачі | 02.03.2026 | 01.04.2026 | Виконано |
| 5 | Реалізація обраних технологій з аналізом отриманих результатів | 02.04.2026 | 24.05.2026 | Виконано |
| 6 | Перший попередній захист КР на засіданні комісії кафедри | 25.05.2026 | 25.05.2026 | Виконано |
| 7 | Корегування роботи за результатами попереднього захисту | 26.05.2026 | 04.06.2026 | Виконано |
| 8 | Другий попередній захист КР на засіданні комісії кафедри | 05.06.2026 | 05.06.2026 | Виконано |
| 9 | Доробка та остаточне оформлення КР | 06.06.2026 | 14.06.2026 | Виконано |
| 10 | Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту | 15.06.2026 | 19.06.2026 | Виконано |

**Керівник роботи** _____ _____Євген СІДЕНКО_____
*(Особистий підпис)* *(Власне ім'я ПРІЗВИЩЕ)*

**Здобувач** _____ ____Олександр ТИМКІВ____
*(Особистий підпис)* *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«29» січня 2026 р.

# АНОТАЦІЯ

до кваліфікаційної роботи
здобувача групи 402 ЧНУ ім. Петра Могили

**Тимків Олександр Олександрович**

на тему **“СИСТЕМА ОНЛАЙН-ЗАМОВЛЕНЬ ПОЇЗДОК НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ”**


**Актуальність** роботи зумовлена зростаючою потребою у відмовостійких платформах міської мобільності. Такі системи мають швидко обробляти великі обсяги даних у реальному часі геоданих та забезпечувати надійність фінансових транзакцій у розподіленому середовищі. Це дозволить підвищити стабільність роботи системи під високим навантаженням та гарантувати безперебійне обслуговування пасажирських перевезень.

**Метою** роботи є розробка інформаційної системи онлайн-замовлень поїздок на основі мікросервісної архітектури, яка забезпечує відстеження локації автомобіля в режимі реального часу та надійність фінансових операцій.

**Об’єктом** роботи є процеси онлайн-бронювання та супроводу пасажирських перевезень.

**Предметом** роботи є методи та архітектурні патерни проєктування мікросервісів для обробки потокових даних і забезпечення узгодженості інформації.

В результаті виконання роботи було досліджено архітектурні патерни розподілених систем, проаналізовано методи обробки високочастотних геоданих, вирішено проблему розподілених транзакцій за допомогою патерну Saga та брокера повідомлень RabbitMQ, а також розроблено серверну частину програмного забезпечення, в якій використано повнодуплексну передачу координат через WebSockets та In-Memory кешування за допомогою Redis.

Дана робота складається з чотирьох розділів. У першому розділі проведено аналіз предметної області, огляд існуючих платформ онлайн-замовлення поїздок та сформовано постановку задачі. Другий розділ присвячений обґрунтуванню моделей, методів та інформаційних технологій мікросервісної архітектури, що

використані у роботі. У третьому розділі наведено результати проєктування архітектури баз даних, алгоритмів взаємодії мікросервісів та реалізації бізнес-логіки системи. В четвертому – наведено експлуатаційну документацію, проведено End-to-End тестування та аналіз результатів роботи розробленої системи. Загальний обсяг роботи – 88 сторінок. Кваліфікаційна робота містить 9 таблиць, 36 рисунків, 31 посилань, 2 додатки.

**Ключові слова:** мікросервісна архітектура, онлайн-замовлення поїздок, розподілені транзакції, WebSockets, геолокація, брокер повідомлень, оптимістичне блокування, інформаційна система, Redis, Spring Boot.

# ABSTRACT

to the qualification work by the student of the group 402 of Petro Mohyla Black Sea National University

**Tymkiv Oleksandr**

## “ONLINE TRAVEL BOOKING SYSTEM BASED ON MICROSERVICE ARCHITECTURE”

The relevance of the work is driven by the growing need for fault-tolerant urban mobility platforms. Such systems must quickly process large volumes of real-time geospatial data and ensure the reliability of financial transactions in a distributed environment. This will improve the system’s stability under high load and guarantee the uninterrupted service of passenger transportation services.

The aim of the work is to develop an online ride-booking information system based on a microservices architecture that provides real-time vehicle location tracking and reliable financial operations.

The object of the work is the processes of online booking and support of passenger transportation.

The subject of the work is the methods and architectural patterns of designing microservices for processing streaming data and ensuring information consistency.

As a result of the work, the architectural patterns of distributed systems were investigated, methods for processing high-frequency geospatial data were analyzed, the problem of distributed transactions was solved using the Saga pattern and the RabbitMQ message broker, and the server-side software was developed, utilizing full-duplex coordinate transmission via WebSockets and in-memory caching using Redis.

This work consists of four chapters. The first chapter provides an analysis of the subject area, a review of existing online ride-hailing platforms, and formulates the problem statement. The second chapter is devoted to the justification of the models, methods, and information technologies of the microservice architecture used in the work. The third chapter presents the results of designing the database architecture,

microservices interaction algorithms, and the implementation of the system’s business logic. The fourth chapter provides operational documentation, conducts End-to-End testing, and analyzes the performance results of the developed system.

The total volume of the work is 88 pages. The qualification work contains 9 tables, 36 figures and 31 references, and 2 appendices.

Keywords: microservice architecture, online ride-hailing, distributed transactions, WebSockets, geolocation, message broker, optimistic locking, information system, Redis, Spring Boot.

# ЗМІСТ

# СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

СУБД – Система управління базами даних

ШІ – Штучний інтелект

ACID – Atomicity, Consistency, Isolation, and Durability

API – Application Programming Interface

DDD – Domain-Driven Design

EDA – Event-Driven Architecture

ESB – Enterprise Service Bus

ETA – Estimated Time of Arrival

JWT – JSON Web Token

LLM – Large Language Model

MaaS – Mobility as a Service

STOMP – Simple Text Oriented Messaging Protocol

TPS – Transactions Per Second

# ВСТУП

Міська інфраструктура та цифровізація транспортної галузі є дуже важливої на чинний час, та вже навіть устигла розвинутися та змінити підходи й методи до організації перевезень. Системи онлайн-замовлень працюють в умовах надвисоких навантажень, постійно обробляючи мільйони запитів щодня, саме тому забезпечення безперебійної роботи платформи вимагає миттєвої маршрутизації, обробки геоданих та фінансових транзакцій. Класичні монолітні програмні рішення для подібної задачі вже є неефективним варіантом, оскільки погано масштабуються, саме тому базовий підхід для розробки високонавантажених транспортних систем став перехід до мікросервісної архітектури.

Серед провідних підходів до побудови розподілених систем виділяються декомпозиція предметної області на слабко зв'язані сервіси, використання брокерів повідомлень для асинхронної взаємодії та імплементацію повнодуплексних протоколів обміну даними. Проте під час проєктування, система стикається з низкою специфічних проблем, зокрема із забезпеченням транзакційної цілісності у розподіленому середовищі, наскрізної авторизації між сервісами та вирішенням конфліктів при конкурентному доступі до спільних ресурсів.

**Актуальність** роботи зумовлена зростаючою потребою у відмовостійких платформах міської мобільності. Такі системи мають швидко обробляти великі обсяги даних у реальному часі геоданих та забезпечувати надійність фінансових транзакцій у розподіленому середовищі. Це дозволить підвищити стабільність роботи системи під високим навантаженням та гарантувати безперебійне обслуговування пасажирських перевезень, тому проєктування власної системи онлайн-замовлення поїздок дозволяє на практиці імплементувати такі складні механізми, як наприклад пересилання токенів, оптимістичне блокування баз даних, а також буферизація високочастотних потоків GPS-координат.

Декларація про використання ШІ. Під час підготовки наукової роботи (академічного тексту) було використано інструмент Gemini 3.1 Pro. Відповідно до

таксономії GAIDeT (2025), наведені нижче завдання були делеговані інструментам генеративного ШІ за повного людського нагляду:

- оцінювання здійсненності та ризиків;
- систематизація та порівняння деяких джерел;
- оптимізація коду;
- вичитування та редагування;
- резюмування тексту;
- адаптація та коригування емоційного тону;
- оцінювання якості;
- рекомендації.

Повну відповідальність за фінальний рукопис несе автор.

Інструменти генеративного ШІ не зазначаються як автори та не несуть відповідальності за кінцеві результати.

Декларацію подав: Тимків Олександр Олександрович

# 1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ СИСТЕМИ ОНЛАЙН-ЗАМОВЛЕНЬ ПОЇЗДОК НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Вибір напрямку розробки та обґрунтування технічних рішень спирається на аналіз предметної області систем онлайн-замовлення поїздок. Цей розділ відстежує еволюцію архітектур подібних платформ, охоплює огляд існуючих аналогів та інженерні підходи до проєктування розподілених мікросервісних середовищ. Підсумком етапу дослідження є формальна постановка задачі, яка визначає мету, об'єкт, предмет і конкретні завдання кваліфікаційної роботи.

## 1.1 Системи онлайн-замовлення поїздок як предметна сфера розробки

Концепція MaaS трансформує ринок транспортних послуг. Традиційні централізовані диспетчерські служби втрачають актуальність, поступаючись автоматизованим цифровим платформам для прямої взаємодії пасажирів та незалежних водіїв [1].

Предметна область фокусується на розробці високонавантажених інформаційних систем. Також на тому, аби вони були здатниими обробляти тисячі паралельних запитів у режимі реального часу. Ключові інженерні виклики під час створення таких рішень – географічна розподіленість вузлів, висока динаміка оновлення стану об'єктів і жорстка вимога до транзакційної цілісності фінансових операцій.

### 1.1.1 Життєвий цикл замовлення та ключові бізнес-процеси

Програмна логіка платформи онлайн-замовлення поїздок імплементує послідовний життєвий цикл замовлення. Процес стартує з етапу ініціалізації та маршрутизації: клієнтський застосунок передає бекенду координати посадки та висадки. Платформа опитує зовнішні геоінформаційні сервіси, вираховує дистанцію з орієнтовним часом у дорозі та генерує попередній трек маршруту.

Сформований маршрут надходить у модуль динамічного ціноутворення. Алгоритм розраховує вартість поїздки, агрегуючи дані про довжину шляху, щільність попиту в заданому полігоні міста, кількість вільних водіїв та поточні погодні умови [2]. Згода користувача з тарифом ініціює процедуру метчингу. Система шукає найближчий доступний автомобіль, скануючи просторові індекси у визначеному радіусі.

Перехід замовлення в статус активної поїздки запускає стрімінг геоданих, тобто бекенд постійно приймає та показує зміну координат. Життєвий цикл замикається етапом фінансових взаєморозрахунків. Сервер закриває поїздку, фіксує у базі даних фактичний маршрут та виконує транзакцію: списує кошти з рахунку пасажира і нараховує заробіток водієві з автоматичним утриманням комісії платформи.

### 1.1.2 Проблематика обробки просторово-часових даних

Робота з високочастотними геопросторовими даними формує один із головних технологічних викликів для транспортних платформ. Мобільний застосунок водія щокілька секунд генерує новий пакет GPS-координат, передаючи широту, довготу та часову мітку. В масштабах середнього міста цей потік телеметрії створює тисячі паралельних запитів до бекенду кожної секунди.

Прямий запис кожної локації у реляційну базу даних вичерпує системні ресурси. Такий підхід створює критичне навантаження на підсистему введення-виведення накопичувачів, провокуючи деградацію загальної продуктивності [3].

Архітектурне розділення процесів трансляції та збереження усуває це вузьке місце. Сервер ретранслює координати пасажиру через повнодуплексні з'єднання WebSockets/STOMP для мінімізації мережевих затримок [4]. Водночас, бекенд агрегує масив точок в оперативній пам'яті за допомогою in-memory бази даних Redis. Логіка системи ініціює фізичний запис маршруту в основну БД виключно після переходу поїздки у статус завершення. На цьому етапі пул координат серіалізується у формат JSON та фіксується як єдиний історичний запис. Це

розвантажує підсистему та зберігає цілісні фактичні координати поїздки. Координати можна використати для можливих конфліктів щодо тарифікації.

### 1.1.3 Проблематика фінансової цілісності у розподілених системах

Перехід до мікросервісної архітектури руйнує узгодженість даних між незалежними предметними доменами. Класичний моноліт обробляє списання коштів з пасажира, нарахування винагороди водію та оновлення статусу поїздки в межах єдиної ACID-транзакції бази даних, тому розділення сервісів управління поїздками та фінансами у розподіленій системі унеможливлює такий підхід [5].

Проблема вирішується впровадженням асинхронної подієво-орієнтованої комунікації через брокери повідомлень. Критичним бізнес-кейсом тут виступає обробка замовлень з недостатнім балансом клієнта. Продакшн-системи міської мобільності імплементують модель відкладеного боргу. Водій забирає розрахунок щойно поїздка була закінчена. Паралельно бекенд записує пасажиру мінусовий баланс та блокує створення нових замовлень до моменту повної компенсації заборгованості. Надійна робота цього алгоритму вимагає точної маршрутизації подій та обов'язкових міжсервісних синхронних перевірок безпосередньо на етапі ініціалізації кожної нової поїздки. Наприклад, мікросервіс, відповідальний за ініціалізацію поїздки, має відправити запит на отримання балансу відповідного пасажиру, аби перевірити, чи має баланс пасажиру наявну ціну поїздки або чи не має він заборгованостей.

### 1.1.4 Проблематика семантичного пошуку локацій та маршрутизації на основі намірів

Класичні геоінформаційні системи та сервіси прямого геокодування оперують виключно строго структурованими адресами. Натомість сучасні патерни Human-Computer Interaction у платформах міської мобільності спираються на пошук за намірами. Користувачі генерують абстрактні запити природною мовою або сленгом, наприклад, «найближча кав'ярня», «де поїсти суші» [6].

Класичні реляційні та просторові бази даних не спроможні на ефективну обробку неструктурованих семантичних текстів, що призводить до або не коректних, або, взагалі, нульових результатів пошуку, тому вирішення нагальної проблеми вимагає архітектурного впровадження проміжного шару на базі технологій обробки природної мови (NLP, Natural Language Processing) або LLM, які здатні виступати інтелектуальними класифікаторами: перетворювати звичайний запит у стандартизовані системні гео-теґи для подальшого точного картографічного пошуку [7, 8]

## 1.2 Аналіз методів побудови розподілених систем та існуючих платформ-аналогів

Проєктування високонавантаженої інформаційної системи вимагає жорсткого обґрунтування архітектури. Впровадження мікросервісів у платформу замовлення поїздок спирається на аналіз технічних обмежень альтернативних рішень.

Історичний стандарт розробки програмного забезпечення – монолітна архітектура. Ця модель агрегує всі бізнес-компоненти, наприклад автентифікація, управління поїздками, геотрекінг, білінг, сповіщення, в єдину кодову базу. Монолітний бекенд виконується в одному процесі та оперує спільною реляційною базою даних. На етапі запуску проєкту такий підхід пришвидшує написання коду і спрощує налаштування тестового середовища. Локальна взаємодія компонентів відбувається через прямі виклики функцій, що зводить мережеві затримки до нуля.

Проте для систем ride-hailing монолітна архітектура має недоліки:

– неможливість незалежного масштабування: у системі таксі модуль обробки GPS-координат отримує в тисячі разів більше запитів, ніж модуль реєстрації користувачів і у моноліті для масштабування геотрекінгу доведеться клонувати на нові сервери весь застосунок повністю, що призводить до неефективного використання апаратних ресурсів;

– сильна зв'язність коду: критична помилка в одному невеликому модулі може призвести до падіння всього процесу і зупинки роботи всієї служби таксі.

Сервіс-орієнтована архітектура (SOA, Service-Oriented Architecture) стала проміжним етапом еволюції програмних рішень. Ця парадигма розділяє систему на окремі великі компоненти, що взаємодіють між собою через єдину шину ESB. На практиці архітектурний вузол ESB швидко перетворюється на єдину точку відмови та вузьке місце продуктивності. Платформи реального часу не здатні надійно функціонувати з такими структурними обмеженнями.

Сьогодні актуальним стандартом для розробки високонавантажених платформ виступає мікросервісна архітектура. Вона будує програмний продукт як кластер невеликих, незалежно розгортуваних сервісів. Кожен мікросервіс імплементує суворо визначену спроможність і завжди оперує власною, ізольованою базою даних.

Візуальне концептуальне порівняння класичної монолітної та мікросервісної архітектур наведено на рис. 1.1.

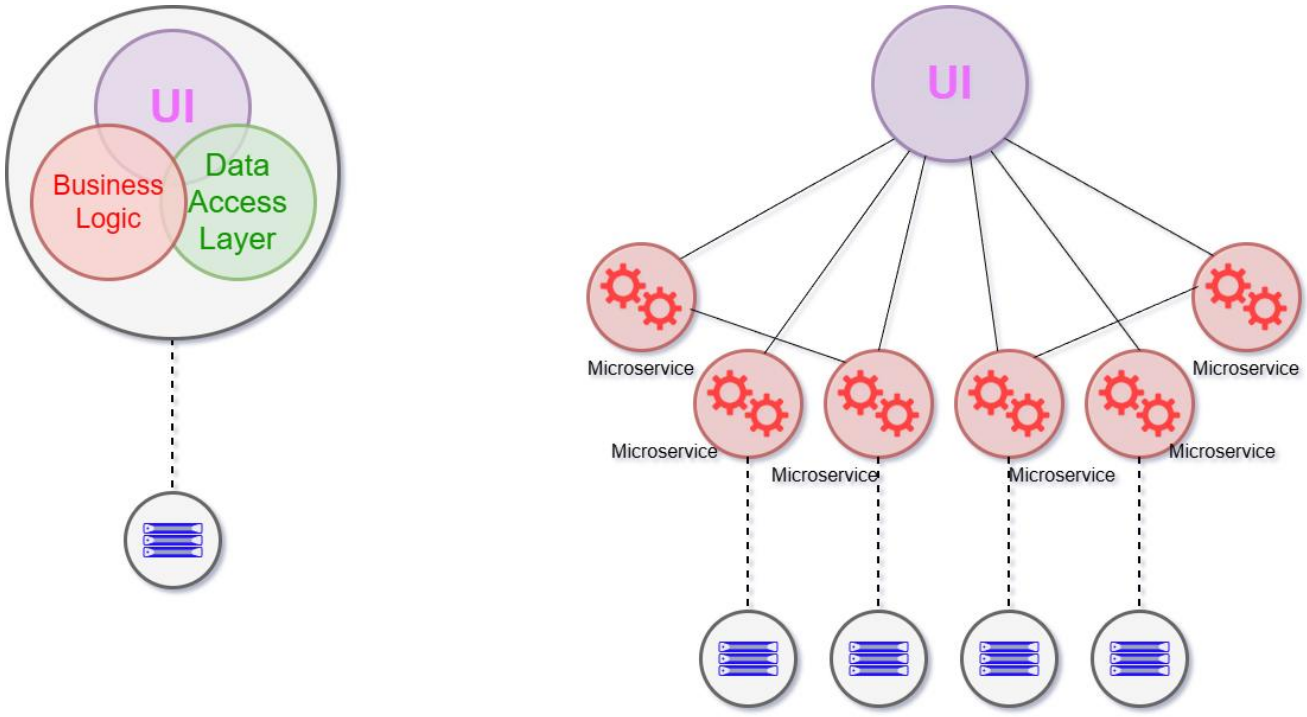


Рисунок 1.1 – Порівняння монолітної та мікросервісної архітектур

Переваги мікросервісної архітектури для систем замовлення поїздок:

– незалежне масштабування: дозволяє виділяти максимум серверних потужностей виключно для модуля стрімінгу геоданих або модуля метчингу, не торкаючись інших сервісів;

– ізоляція збоїв: якщо сервіс статистичних звітів або білінгу виходить з ладу, пасажири все ще можуть замовляти авто, а водії – завершувати маршрути, оскільки система продовжує функціонує, але з неповноцінним функціоналом;

– технологічна гетерогенність: кожен мікросервіс працює з базою даних, яка найкраще відповідає профілю його навантаження, наприклад, фінансові транзакції проводяться через класичні реляційні СУБД, як PostgreSQL, тоді як надшвидке збереження телеметрії та геоданих лягає на in-memory сховища, як Redis.

Незважаючи на об’єктивні переваги мікросервісної над монолітною архітектурою. Впровадження мікросервісів генерує об'єктивну технологічну складність. Головним викликом стає розподілене управління даними: ізоляція баз даних на рівні кожного компонента блокує механізм класичних ACID-транзакцій. Архітектура змушує інженерів імплементувати патерни узгодженості у кінцевому рахунку та логіку розподілених транзакцій, зокрема патерн “Saga”. Додаткове навантаження створюють мережеві затримки під час міжсервісної взаємодії. І також вартує уваги те, що розгортання та безперервний моніторинг розподіленого середовища вимагають побудови зрілої DevOps-інфраструктури.

Технологічний стек ринку онлайн-замовлення поїздок формують кілька компаній-монополістів, тому виділення робочих архітектурних практик для проєктування спирається на аналіз технічної еволюції лідерів галузі: Uber, Bolt та Uklon.

### 1.2.1 Аналіз архітектурних рішень платформи Uber

Uber однією з перших зіткнувся з кризою масштабування систем ride-hailing. Щорічна обробка мільярдів поїздок генерує екстремальне навантаження на

серверну інфраструктуру [9]. Початкова монолітна архітектура платформи швидко вичерпала свій технічний ресурс. Жорстка зв'язність компонентів руйнувала загальну відмовостійкість системи: локальний збій у модулі розрахунку вартості провокував каскадне падіння всього контуру диспетчеризації.

Подолання цього архітектурного обмеження вимагало повної декомпозиції моноліту на тисячі незалежних мікросервісів. Навантаження, яке пов'язане з обробкою геоданих та алгоритмами метчингу, перенесли у розподілені in-memory сховища. Цей механізм працює у прямій зв'язці з технологіями потокової обробки подій. Впроваджений патерн забезпечує безперервний стрімінг координат у реальному часі та миттєву реакцію на зміну статусів поїздок. При цьому основні реляційні бази даних залишаються повністю ізольованими від високочастотного потоку оновлень.

### 1.2.2 Аналіз архітектурних рішень платформи Bolt

Масштабування європейського сервісу Bolt спирається на глибоку оптимізацію алгоритмів метчингу та розрахунку ETA [10]. Бекенд платформа відмовилася від ресурсоємного повного сканування баз даних під час пошуку найближчих екіпажів, тому логіка маршрутизації використовує виключно просторові індекси, імплементуючи рішення формату H3 від Uber або S2 від Google.

Аналіз платформи Bolt вказує на те, що критично важливим аспектом є забезпечення відмовостійкості фінансових транзакцій. Для цього компанія використовує брокери повідомлень, які гарантують доставку фінансових подій навіть у випадку тимчасової недоступності банківських шлюзів, тому це підтверджує необхідність використання патернів асинхронної взаємодії при проєктуванні власної системи.

### 1.2.3 Аналіз архітектурних рішень платформи Uklon

Uklon обробляє понад 135 мільйонів поїздок щорічно [11]. Такий обсяг трафіку та жорсткі пікові навантаження під час погодних аномалій або святкових днів вказують на пряму вимогу до масштабування бекенду.

Компанія пройшла повний цикл міграції від моноліту до мікросервісів. Ізоляція предметних доменів дозволила інженерам незалежно розширювати обчислювальні ресурси виключно тих вузлів, які приймають на себе головний удар навантаження. Двосторонній зв'язок між клієнтськими застосунками та сервером платформа реалізує через WebSockets. Цей транспорт безперервно ретранслює зміну координат автомобіля на карту пасажира у реальному часі.

Варто також зазначити, що останнім галузевим трендом серед провідних платформ є поступова інтеграція штучного інтелекту для покращення користувацького інтерфейсу. Розумні підказки, інтелектуальне розпізнавання популярних точок інтересу та семантичний пошук локацій стають новим стандартом, що підтверджує доцільність впровадження NLP-модулів у сучасні архітектури систем ride-hailing.

### 1.2.4 Загальний огляд архітектурних рішень платформ

Огляд архітектурних трансформацій провідних платформ дозволяє виокремити узагальнену концептуальну схему взаємодії компонентів у сучасних системах замовлення поїздок, яка наведена на рис. 1.2.

Огляд провідних платформ та актуальних наукових публікацій дозволяє систематизувати підходи до побудови систем замовлення поїздок. Порівняльний аналіз ключових архітектурних рішень наведено у табл. 1.1.

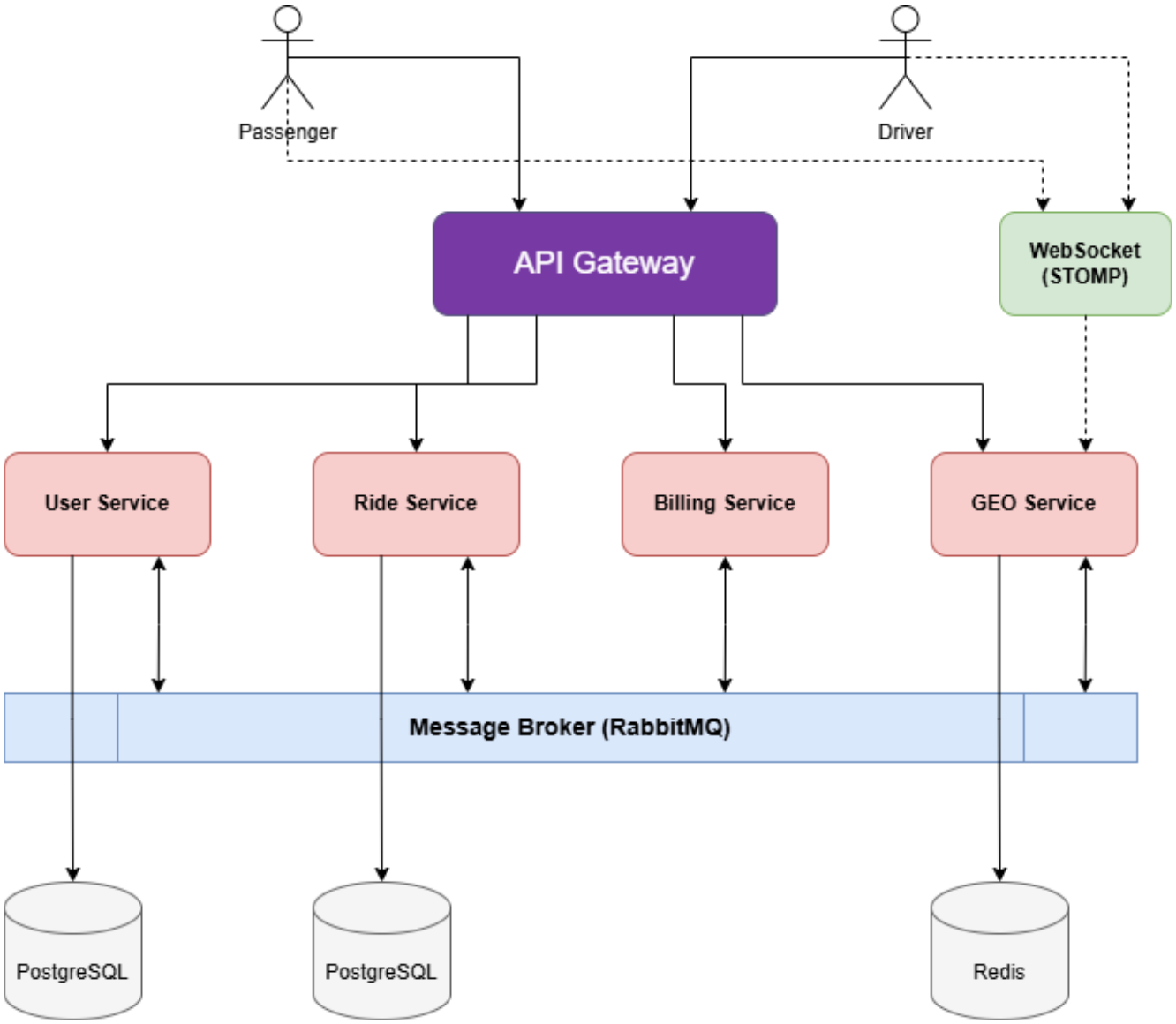


Рисунок 1.2 – Узагальнена концептуальна архітектура системи ride-hailing

Як видно з табл. 1.1, незважаючи на певні відмінності в імплементації, загальний галузевий стандарт базується на трьох стовпах: мікросервісна ізоляція доменів, подієво-орієнтована модель з використанням брокерів повідомлень для розподілених фінансових транзакцій та in-memory сховища у поєднанні з повнодуплексними протоколами для стрімінгу телеметрії. Саме ці технологічні патерни будуть покладені в основу проєктування серверної частини інформаційної системи у даній кваліфікаційній роботі.

Таблиця 1.1 – Порівняльний аналіз архітектурних рішень провідних платформ

| Платформа | Базова архітектура | Обробка геоданих та метчинг | Стрімінг даних у реальному часі | Управління транзакціями |
|---|---|---|---|---|
| Uber | Мікросервісна (глибоко децентралізована) | Розподілені in-memory сховища, просторові індекси (H3, S2) | Системи потокової обробки подій (Event Streaming) | Асинхронні транзакції на базі подій |
| Bolt | Мікросервісна | Оптимізовані просторові індекси для мінімізації часу подачі (ETA) | Асинхронний обмін просторовими даними | Брокери повідомлень для гарантованої доставки фінансових подій |
| Uklon | Мікросервісна (з еластичним масштабуванням) | In-memory кеші, оптимізований пошук у радіусі | WebSockets для двостороннього клієнт-серверного зв'язку | Подієво-орієнтована архітектура (EDA) |

## 1.3 Постановка задачі

**Актуальність** роботи зумовлена зростаючою потребою у відмовостійких платформах міської мобільності. Такі системи мають швидко обробляти великі обсяги даних у реальному часі геоданих та забезпечувати надійність фінансових транзакцій у розподіленому середовищі. Традиційні монолітні архітектури не здатні забезпечити необхідний рівень еластичного масштабування та відмовостійкості при таких навантаженнях. Тому дослідження та розробка серверних рішень на базі мікросервісної та EDA є надзвичайно актуальною науково-практичною задачею.

**Метою** роботи є розробка інформаційної системи онлайн-замовлень поїздок на основі мікросервісної архітектури, яка забезпечує відстеження локації автомобіля в режимі реального часу та надійну консистентність фінансових операцій.

**Об’єктом** роботи є процеси онлайн-бронювання та супроводу пасажирських перевезень.

**Предметом** роботи методи та архітектурні патерни проєктування мікросервісів для обробки потокових даних і забезпечення узгодженості інформації.

Для досягнення поставленої мети в кваліфікаційній роботі необхідно вирішити такі завдання:

– проаналізувати предметну сферу, виявити ключові технологічні проблеми систем класу ride-hailing та дослідити архітектурні підходи провідних аналогів;

– спроєктувати мікросервісну архітектуру системи з виокремленням незалежних бізнес-доменів (користувачі, поїздки, білінг, геолокація);

– реалізувати алгоритми та механізми трансляції GPS-координат автомобілів у реальному часі з використанням проміжних in-memory буферів та повнодуплексних протоколів передачі даних;

– спроєктувати логіку розподілених фінансових транзакцій на базі брокера повідомлень для списання коштів, нарахування винагороди та управління заборгованостями користувачів;

– розробити працездатні розділені інтерфейси для адекватної роботи системи;

– здійснити програмну реалізацію розробленої архітектури та провести тестування працездатності ключових модулів системи.

## Висновки до розділу 1

Перший розділ фіксує результати аналізу предметної області систем онлайн-замовлення поїздок. Дослідження виділяє два головні технологічні виклики для таких платформ. Перший вказує на необхідність обробки високочастотних потоків геоданих без перевантаження дискових підсистем збереження, а другий – гарантування надійних фінансових операцій у розподіленому середовищі.

Огляд архітектурної еволюції платформ наочно показує технічну необхідність відмови від монолітних рішень. Оптимальний галузевий стандарт будується на комбінації мікросервісної архітектури, інтеграції in-memory сховищ. Спираючись на це, розділ формулює актуальність дослідження, визначає мету, об'єкт і предмет роботи, що також формує базис, який генерує конкретні завдання для розробки програмної системи.

# 2 МЕТОДИ ТА ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ ОНЛАЙН-ЗАМОВЛЕНЬ ПОЇЗДОК

У другому розділі кваліфікаційної роботи йдеться про безпосереднє проєктування серверної частини системи онлайн-замовлень. Головним завданням цього розділу є обґрунтування вибору концептуальних методів, архітектурних патернів та конкретного стеку інформаційних технологій, які здатні забезпечити виконання поставлених вимог щодо масштабованості, відмовостійкості та безпеки. Перший розділ описує алгоритмічні рішення для критичних проблем розподіленого середовища: маршрутизації подій, управління транзакціями та стрімінгу геоданих, у той же час, другий – розглядає інструментарій розробки, який безпосередньо конвертує ці архітектурні підходи у робочий програмний код.

## 2.1 Методи для вирішення поставленої задачі

Проєктування високонавантаженої системи онлайн-замовлень поїздок вимагає застосування комплексу сучасних методів програмної інженерії. Основним завданням на цьому етапі є вибір архітектурних патернів, які забезпечать ізоляцію компонентів, безпечну передачу контексту користувача, надійність розподілених транзакцій та мінімізацію затримок при передачі телеметрії.

### 2.1.1 Метод декомпозиції на основі предметно-орієнтованого проєктування

Перехід спирається на методологію DDD [12]. Архітектура розмежовує складну бізнес-логіку на незалежні обмежені контексти. Програмне рішення імплементує три цільові домени:

– домен користувачів та фінансів: відповідає за управління профілями, балансами гаманців та механізмом розрахунку боргів;

– домен поїздок: керує життєвим циклом замовлення (від створення до статусу ”COMPLETED”);

– домен геолокації: відповідає за обробку просторово-часових масивів та маршрутизацію.

Такий метод декомпозиції дозволяє розробляти, розгортати та масштабувати кожен сервіс незалежно, а також, кожен сервіс оперує суворо ізольованою базою даних, що виключає перетин транзакцій між доменами

### 2.1.2 Методи та патерни міжсервісної взаємодії

Ізоляція баз даних вимагає налаштування ефективної міжсервісної комунікації, оскільки архітектура розроблюваної платформи спирається на гібридний підхід, комбінуючи синхронну та асинхронну взаємодію.

Синхронні HTTP-запити застосовуються там, де необхідна миттєва відповідь. Наприклад, перед створенням поїздки сервіс поїздок повинен перевірити баланс користувача у сервісі користувачів, оскільки система працює з авторизованими запитами, виникає проблема втрати контексту безпеки при виклику одного внутрішнього сервісу іншим. Для вирішення цієї проблеми застосовано архітектурний патерн Token Relay [13]. Метод полягає у використанні перехоплювачів вихідних запитів, які автоматично витягують оригінальний JWT-токен із контексту вхідного запиту пасажира та ін'єктують його у заголовок вихідного запиту до іншого мікросервісу. Це гарантує наскрізну безпеку та дозволяє кінцевому сервісу ідентифікувати користувача без додаткових звернень до сервера авторизації.

Асинхронна комунікація закриває фонові процеси за принципами EDA. Бекенд відмовляється від прямих синхронних викликів на користь публікації подій домену в централізовану шину даних. Фіксація фінішу маршруту генерує системну подію, наприклад, RideCompletedEvent, а інші мікросервіси перехоплюють цей сигнал асинхронно і паралельно оновлюють власні ізольовані бази даних – перераховують статистику та списують кошти з гаманців.

### 2.1.3 Метод забезпечення цілісності розподілених даних

Відмова від спільних реляційних баз даних унеможливлює використання класичних ACID-транзакцій. Для вирішення проблеми фінансових взаєморозрахунків (списання коштів у пасажира та нарахування їх водію після завершення поїздки) застосовано патерн “Saga” на основі хореографії [14].

Метод полягає у розбитті глобальної бізнес-транзакції. Кожна локальна транзакція оновлює базу даних свого мікросервісу та публікує подію, яка тригерить наступний крок саги.

Кроки моделі гарантованого розрахунку:

- сервіс поїздок успішно закриває поїздку і публікує подію;
- сервіс фінансів перехоплює подію, нараховує винагороду на баланс водія;
- сервіс фінансів намагається списати кошти з балансу пасажира.

Система не ініціює компенсуючі транзакції для скасування поїздки у випадку недостатнього балансу, оскільки послугу вже фактично надано, натомість, логіка платформи імплементує механізм фіксації від'ємного балансу. Цей запис блокуватиме створення поїздок. Кожна наступна спроба ініціалізувати поїздку проходить жорстку синхронну перевірку і відхиляється системою до моменту повного поповнення рахунку клієнта.

### 2.1.4 Методи забезпечення відмовостійкості

Розподілена екосистема мікросервісних застосунків робить їх вразливими до часткових мережевих збоїв та каскадних відмов. Якщо один мікросервіс починає відповідати із затримкою, ресурси викликаючого сервісу (наприклад, пули потоків) можуть швидко вичерпатися, що призведе до падіння всієї системи.

Для запобігання каскадним збоям у розробленій системі застосовано архітектурний патерн Circuit Breaker [15]. Цей метод працює за аналогією з електричним запобіжником: він відстежує кількість помилок або таймаутів при зверненні до віддаленого сервісу. Якщо рівень помилок перевищує заданий поріг,

запобіжник перемикається, і всі наступні виклики до проблемного сервісу миттєво відхиляються без реального мережевого запиту. У цей час система може повернути користувачеві стандартну відповідь – наприклад, повідомлення про тимчасову недоступність функції розрахунку ціни, зберігаючи працездатність інших модулів застосунку. Після певного періоду очікування запобіжник переходить у стан «напіввідкритого», пропускаючи тестові запити для перевірки відновлення роботи цільового сервісу.

### 2.1.5 Методи моніторингу та розподіленого трасування

На відміну від моноліту, де весь шлях виконання запиту можна відстежити в одному лоґ-файлі, у мікросервісній системі один клієнтський запит може проходити через API Gateway, сервіс поїздок, сервіс користувачів та брокер повідомлень, тому для розуміння життєвого циклу запиту та виявлення «пляшкового горлечка» застосовано метод Distributed Tracing [16].

Суть методу полягає у генерації унікального ідентифікатора транзакції на рівні API Gateway й цей ідентифікатор передається через HTTP-заголовки до кожного наступного мікросервісу в ланцюжку викликів. Кожна окрема операція матиме свій ідентифікатор. Усі логи системи маркуються цими ідентифікаторами, що дозволяє централізованим системам моніторингу автоматично реконструювати повне дерево виконання запиту та візуалізувати затримки на кожному етапі мережевої взаємодії.

### 2.1.6 Методи обробки просторово-часових даних та маршрутизації

Для імітації динамічної поїздки та візуалізації руху на клієнтських застосунках застосовано патерн Publish-Subscribe поверх повнодуплексних каналів зв'язку [17].

Логіка маршрутизації відпрацьовує у дві фази. На етапі планування сервер звертається до зовнішніх геоінформаційних API для розрахунку очікуваного часу

та дистанції, а отримані метрики фіксують початкову вартість замовлення на основі алгоритмів пошуку найкоротшого шляху на графі.

Фаза виконання повністю змінює підхід до роботи з даними. Під час руху система блокує прямий запис координат у персистентну базу даних, відсікаючи критичні накладні витрати на I/O операції, тому це архітектурне обмеження закриває механізм буферизації в оперативній пам'яті, оскільки сервер миттєво транслює вхідні координати активним підписникам, а після фінішу логіка агрегує весь накопичений трек у єдиний JSON-документ і скидає його у сховище. Збережений історичний запис є доказовою базою факту виконання поїздки та вирішує класичну проблему розбіжності між спланованим і фактичним маршрутами.

### 2.1.7 Математична модель обчислення фактичної відстані маршруту (Формула Гаверсина)

Обґрунтування гібридного підходу до тарифікації. Класичним підходом у розробці систем міської мобільності є використання зовнішніх картографічних провайдерів для постійного трекінгу маршруту. Проте під час стрес-тестування було виявлено, що відправка GPS-координат кожні 2 секунди до зовнішнього API створює критичне навантаження на мережу та швидко вичерпує ліміти безкоштовних тарифних планів.

У роботі було розроблено гібридну модель:

– етап ініціалізації: для визначення попередньої вартості поїздки система робить єдиний синхронний запит до відкритого API OpenRouteService для побудови точного графа маршруту з урахуванням дорожньої мережі [18];

– етап виконання поїздки: після старту поїздки зовнішні виклики повністю блокуються. Телеметрія, що надходить через WebSockets, накопичується в In-Memory базі Redis. Після завершення поїздки бекенд самостійно обчислює фактичний пробіг на основі зібраного масиву точок за допомогою Формули Гаверсина.

Оскільки система підтримує модель завчасного ціноутворення, на етапі планування поїздки використовується прогнозована відстань від зовнішнього картографічного провайдера. Однак фактичний маршрут виконання замовлення може суттєво відрізнятися від запланованого через дорожню обстановку, об'їзди або прохання пасажира. Для забезпечення справедливого розрахунку або виявлення аномалій (відхилень від маршруту), система повинна самостійно обчислювати реальну пройдену відстань на основі масиву зібраних під час поїздки GPS-координат.

Для вирішення цього завдання, без необхідності відправляти великі масиви точок до платних зовнішніх API, є можливість застосувати математичний метод обчислення відстані по великому колу сфери – Формулу Гаверсина [19, 20]. Цей метод є стандартом для навігаційних розрахунків, оскільки він враховує сферичну кривизну поверхні Землі та забезпечує високу точність між двома географічними координатами.

Математична модель обчислення відстані $d$ між двома послідовними точками на сфері визначається системою рівнянь:

$$a = \sin^2(\frac{\Delta\varphi}{2}) + \cos\varphi_1 \times \cos\varphi_2 \times \sin^2(\frac{\Delta\lambda}{2}), \quad (2.1)$$

де $\Delta\varphi$ – зміщення широт, утворена за допомогою різниці широт;

$\Delta\lambda$ – зміщення довгот, утворена за допомогою різниця довгот;

$\varphi_1, \varphi_2$ – географічна широта першої та другої точок (у радіанах).

$$c = 2 \times \text{atan2}(\sqrt{a}, \sqrt{1-a}), \quad (2.2)$$

де $c$ – кутова відстань у радіанах;

$a$ – квадрат половини хорди великого кола між точками.

$$d = R \times c, \quad (2.3)$$

де R – середній радіус Землі (приблизно 6371 км).

Загальна фактична відстань поїздки $D_{total}$ обчислюється як сума відстаней між усіма послідовними парами точок у збереженому масиві маршруту:

$$D_{total} = \sum_{i=1}^{n-1} d\,(p_i, p_i + 1), \tag{2.4}$$

де n – загальна кількість зафіксованих координатних точок;

$p_i$ – координата з масиву телеметрії.

2.1.8 Метод семантичної класифікації гео-запитів та обмеження просторового пошуку

Для реалізації маршрутизації на основі намірів у роботі запропоновано та обґрунтовано двокроковий гібридний метод обробки неструктурованих геопросторових запитів користувачів. Метод дозволяє нівелювати проблему просторових галюцинацій LLM та забезпечити стовідсоткову точність GPS-координат цільових точок:

– обробка тексту користувача, введеного природною мовою, делегується NLP-процесору на базі LLM. Завданням моделі є вилучення наміру та його мапінг на один із суворих системних англомовних тегів категорії, які підтримуються глобальними картографічними базами даних OpenStreetMap;

– для усунення колізій глобального пошуку імплементовано математичну модель розрахунку обмежувального прямокутника навколо пасажира.

На основі поточних географічних координат користувача та заданого радіуса пошуку динамічно обчислюються зміщення широт та довгот за формулами:

$$\Delta\varphi = \frac{radiusKm}{111.0}, \tag{2.5}$$

де $\Delta\varphi$ – зміщення широт, утворена за допомогою різниці широт;

111.0 – приблизна довжина одного градуса широти в кілометрах.

$$\Delta\lambda = \frac{radiusKm}{111.0 \times \cos\,(toRadians(lat))}, \tag{2.6}$$

де $\Delta\lambda$ – зміщення довжина, утворена за допомогою різниці довжин;

$\cos(toRadians(lat))$ – косинус широти, де враховується сферичне звуження меридіанів Землі при русу до полюсів.

Запропонований інженерний підхід декартового наближення довжини дуги великого кола для еліпсоїда WGS-84 дозволяє з мінімальними обчислювальними витратами сформувати координати пошукового полігону для картографічного API [21].

## 2.2 Методи обробки просторово-часових даних та маршрутизації

Для практичної реалізації спроєктованої мікросервісної архітектури та обраних алгоритмічних методів було підібрано сучасний стек технологій. Вибір інструментарію базувався на критеріях продуктивності, наявності розвиненої екосистеми для розподілених систем та підтримки відкритих стандартів.

### 2.2.1 Основна платформа та фреймворки розробки

Основною мовою програмування серверної частини обрано Java, яка є галузевим стандартом для розробки корпоративних високонавантажених застосунків, а базовим фреймворком для побудови мікросервісів виступає Spring Boot [22]. Він забезпечує швидке розгортання автономних застосунків із вбудованими вебсерверами.

Для забезпечення взаємодії між мікросервісами використано стек Spring Cloud. Зокрема, для здійснення внутрішніх синхронних HTTP-викликів застосовано декларативний REST-клієнт OpenFeign. Його інтеграція дозволяє викликати віддалені сервіси, як звичайні локальні Java-методи, а механізм FeignClientInterceptor дозволяє реалізувати раніше описаний патерн Token Relay для прокидання авторизаційних заголовків.

### 2.2.2 Система управління ідентифікацією та доступом

Реалізація власного кастомного рішення автентифікації та авторизації не є гарним тоном. Це є антипатерном у безпеці. Для реалізації автентифікації та авторизації, безпосередньо, без необхідності розробки власного кастомного рішення використано Keycloak, що є сервісом, який підтримує сучасні протоколи OAuth 2.0 та OpenID Connect. Keycloak виступає єдиним сервером авторизації, який видає клієнтам JWT-токени [23]. Сервіси платформи виступають як Resource Servers, валідуючи криптографічний підпис токенів без додаткових звернень до бази даних, що суттєво зменшує мережеві затримки.

### 2.2.3 Деталізація механізму криптографічних токенів

При інтеграції з сервером авторизації ключовим технологічним стандартом є JSON Web Token [24]. Вибір цієї технології обґрунтований її “stateless” природою. Токен складається з трьох частин: заголовка, корисного навантаження, де зберігається ідентифікатор користувача та його ролі, і криптографічного підпису, оскільки підпис генерується сервером авторизації за допомогою приватного RSA-ключа, будь-який мікросервіс у системі може перевірити автентичність токена за допомогою публічного ключа, не роблячи додаткових мережевих запитів до бази даних для валідації сесії, тому це радикально знижує навантаження на систему при тисячах одночасних запитів від пасажирів та водіїв.

### 2.2.4 Технології збереження даних

Відповідно до принципу Polyglot Persistence, для різних доменів обрано різні системи управління базами даних:

– PostgreSQL – надійна об'єктно-реляційна СУБД, обрана як основне сховище для сервісів користувачів та фінансів, де критичною є строга типізація та транзакційність. При налаштуванні середовища виконання та інтеграції з PostgreSQL критичним аспектом є точна синхронізація часових поясів між

сервером баз даних та застосунком (зокрема, обов'язкове використання актуального ідентифікатора Europe/Kyiv замість застарілого). Це гарантує коректність збереження часових міток при аудиті поїздок та розрахунку динамічних тарифів;

– Redis – високопродуктивне In-memory сховище структур даних. У системі воно виконує роль проміжного буфера для агрегації GPS-координат водіїв у реальному часі [25]. Запис масивів координат безпосередньо в оперативну пам'ять Redis дозволяє витримувати навантаження у десятки тисяч операцій введення-виведення на секунду (IOPS), оберігаючи основну реляційну базу від деградації продуктивності. Згодом зібрані масиви серіалізуються та зберігаються у PostgreSQL з використанням типу даних JSONB для забезпечення гнучкості структури історичного маршруту.

### 2.2.5 Брокер повідомлень

Для реалізації подієво-орієнтованої архітектури (Event-Driven Architecture) та патерну “Saga” використано брокер повідомлень RabbitMQ, який реалізує розширений протокол черг повідомлень (AMQP) [26]. RabbitMQ забезпечує гарантовану доставку повідомлень, дозволяючи безпечно проводити фінансові транзакції: якщо сервіс білінгу тимчасово недоступний, подія про завершення поїздки буде зберігатися в черзі до моменту відновлення працездатності сервісу.

### 2.2.6 Технології реального часу

Для забезпечення двостороннього повнодуплексного зв'язку між клієнтськими застосунками та сервером картографії, а саме для трансляції координат автомобіля, застосовано протокол WebSockets. З метою стандартизації формату повідомлень поверх WebSockets інтегровано високорівневий протокол STOMP [27]. Використання STOMP дозволяє реалізувати гнучкий патерн Publish-Subscribe, де пасажири підписуються на конкретний канал своєї активної поїздки і миттєво отримують оновлення геолокації автомобіля.

### 2.2.7 Технології контейнеризації та розгортання

Характерною рисою мікросервісної архітектури є наявність великої кількості незалежних компонентів, що суттєво ускладнює процес розгортання та налаштування середовища виконання. Для вирішення цієї проблеми у розробленій системі застосовано технологію контейнеризації Docker [28].

Кожен мікросервіс пакується в ізольований Docker-контейнер, який містить усі необхідні залежності, компільований байт-код та середовище виконання. Це гарантує ідентичність роботи застосунку на локальній машині розробника та на продуктовому сервері. Крім того, для локального оркестрування інфраструктури використано інструмент Docker Compose. Він дозволяє за допомогою єдиного конфігураційного файлу розгорнути всі супутні сервіси у єдиній віртуальній підмережі, забезпечуючи їхню безперешкодну комунікацію через внутрішні DNS-імена.

### 2.2.8 Технології централізованого логування та моніторингу

Для реалізації раніше описаного методу розподіленого трасування та спостережуваності у Java-екосистемі використовується бібліотека Micrometer Tracing. Її принцип дій такий, що вона автоматично перехоплює всі вхідні та вихідні HTTP-запити і додає Trace ID до логів застосунку.

Оскільки мікросервіси генерують величезний обсяг неструктурованих логів на різних фізичних серверах, їх ручний аналіз є неможливим, а для централізованого збору та аналітики інтегрується стек ELK (Elasticsearch, Logstash, Kibana) [29]. Logstash виконує роль конвеєра збору даних: він збирає лог-файли з усіх Docker-контейнерів, форматує їх та відправляє у високошвидкісну пошукову систему Elasticsearch.

### 2.2.9 Інструментарій автоматизованого тестування

Для реалізації парадигми TDD (Test-Driven Development) застосовано фреймворк JUnit 5 у поєднанні з бібліотекою Mockito, що дозволяє ізольовано тестувати бізнес-логіку розрахунку тарифів та метчингу без підняття всього контексту застосунку.

Однак, враховуючи складність інфраструктури, то класичного модульного тестування недостатньо, тому для проведення інтеграційного тестування обрано бібліотеку Testcontainers [30]. Ця технологія дозволяє під час виконання фази тестування автоматично запускати тимчасові Docker-контейнери з реальними базами даних та брокерами повідомлень, виконувати на них тестові сценарії транзакцій і автоматично видаляти їх після завершення. Це гарантує максимальну наближеність тестового середовища до реальних умов.

### 2.2.10 Технологічний стек модуля інтелектуальних рекомендацій

Програмна реалізація методу семантичного пошуку інтегрована у мікросервіс GEO Service з метою ізоляції зовнішніх викликів та збереження принципу єдиної відповідальності. Стек складається з таких інструментів:

- Google Gemini API (модель gemini-2.5-flash): обрана за критерієм мінімальної затримки. Застосовується виключно для швидкого семантичного парсингу текстових рядків через HTTP POST-запити [31];
- OpenStreetMap Nominatim API: безкоштовний open-source сервіс геокодування, який приймає згенерований ШІ англійський тег категорії та розрахований Bounding Box. Завдяки передачі HTTP-заголовка Accept-Language: uk-UA, сервіс повертає верифіковані назви та адреси реальних міських об'єктів українською мовою;
- Spring RestClient: HTTP-клієнт фреймворку Spring Boot, який використовується для організації асинхронного та відмовостійкого зв'язку із

зовнішніми інтерфейсами API. Інтеграція з RestClient.Builder дозволяє автоматично логувати та метрикувати зовнішні запити.

## Висновки до розділу 2

У другому розділі було обґрунтовано вибір методів та технологій для реалізації мікросервісної інформаційної системи онлайн-замовлень поїздок. Показано, що використання методології DDD є необхідним кроком для коректної декомпозиції складної бізнес-логіки. Для вирішення специфічних проблем розподілених систем обрано патерн Token Relay для наскрізної авторизації та патерн “Saga”.

На етапі вибору технологій доведено доцільність використання мови Java та фреймворку Spring Boot як основного інструментарію розробки. Архітектуру доповнено сервером авторизації Keycloak, реляційною СУБД PostgreSQL, in-memory сховищем Redis та брокером повідомлень RabbitMQ. Для стрімінгу просторово-часових даних затверджено використання протоколів WebSockets та STOMP. Обраний комплекс методів та технологій створює надійний теоретичний та інструментальний фундамент для етапу практичної реалізації системи.

# 3 РОЗРОБКА СИСТЕМИ ОНЛАЙН-ЗАМОВЛЕНЬ ПОЇЗДОК ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

## 3.1 Структура системи та опис вхідних даних

Розробка серверної частини системи онлайн-замовлення поїздок передбачає проєктування архітектури, визначення структури даних та реалізацію алгоритмів обробки просторових координат.

Перед описом технічної реалізації необхідно визначити функціональні межі системи та основних акторів. Взаємодія з платформою здійснюється від імені двох основних користувачів: Пасажира, що ініціює замовлення, управляє балансом та відстежує авто, та Водій, який приймає замовлення, транслює свої координати та змінює статус поїздки. Базові сценарії використання системи наведено на діаграмі прецедентів (Use Case) на рис. 3.1.

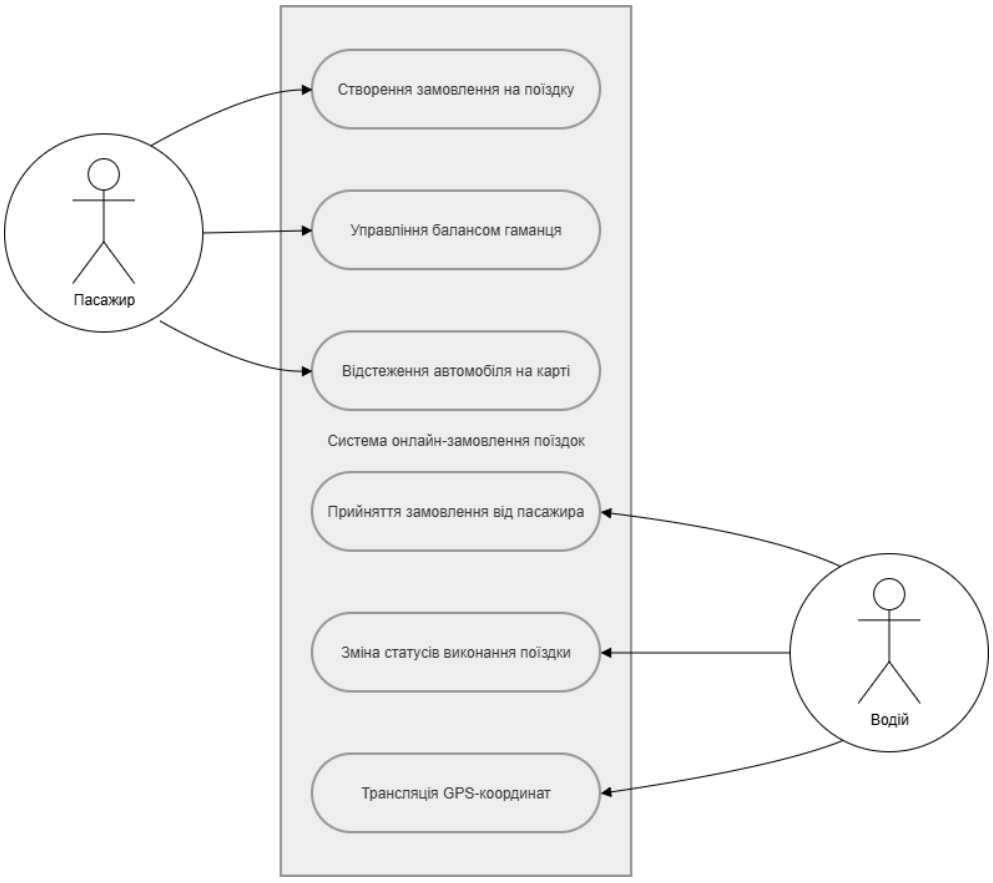


Рисунок 3.1 – UML-діаграма варіантів використання системи

### 3.1.1 Архітектура системи

Для забезпечення високої відмовостійкості, горизонтального масштабування та ізоляції бізнес-доменів інформаційну систему онлайн-замовлення поїздок побудовано за децентралізованою мікросервісною архітектурою. На відміну від класичних монолітних рішень, такий підхід дозволяє незалежно розгортати та масштабувати окремі компоненти платформи, що є критично важливим в умовах інтенсивного потоку просторово-часової телеметрії від водіїв та високої частоти фінансових транзакцій.

Система реалізована на основі розподіленої клієнт-серверної моделі, яка логічно розділена на кілька рівнів: рівень клієнтських застосунків, рівень крайового шлюзу, рівень автентифікації та безпеки, рівень бізнес-логіки та рівень розподіленого зберігання даних.

Взаємодія користувачів із платформою здійснюється через два незалежні типи клієнтського програмного забезпечення:

– застосунок пасажира: надає інтерфейс для ініціалізації замовлень, розрахунку попередньої вартості, управління платіжними картами та візуалізації руху призначеного автомобіля на карті в режимі реального часу;

– застосунок водія: призначений для прийому замовлень, зміни статусів виконання поїздки та автоматичної фонової трансляції поточних GPS-координат пристрою на сервер.

Клієнтські застосунки використовують гібридний протокол комунікації: синхронні операції (реєстрація, перегляд історії, створення замовлення) виконуються через REST API за протоколом HTTP, з іншого боку – асинхронна трансляція геоданих та миттєве отримання статусів виконується через стійкі повнодуплексні WebSocket-з'єднання.

Безпека та ідентифікація користувачів у системі повністю делегована централізованому open-source серверу управління доступом Keycloak (Identity and Access Management, IAM). Keycloak виступає в ролі незалежного сервера

авторизації, що підтримує протоколи OAuth 2.0 та OpenID Connect (OIDC). При успішному вході клієнт отримує підписаний криптографічним алгоритмом RSA токен доступу.

На рис. 3.2 наведено інтерфейс Keycloak.

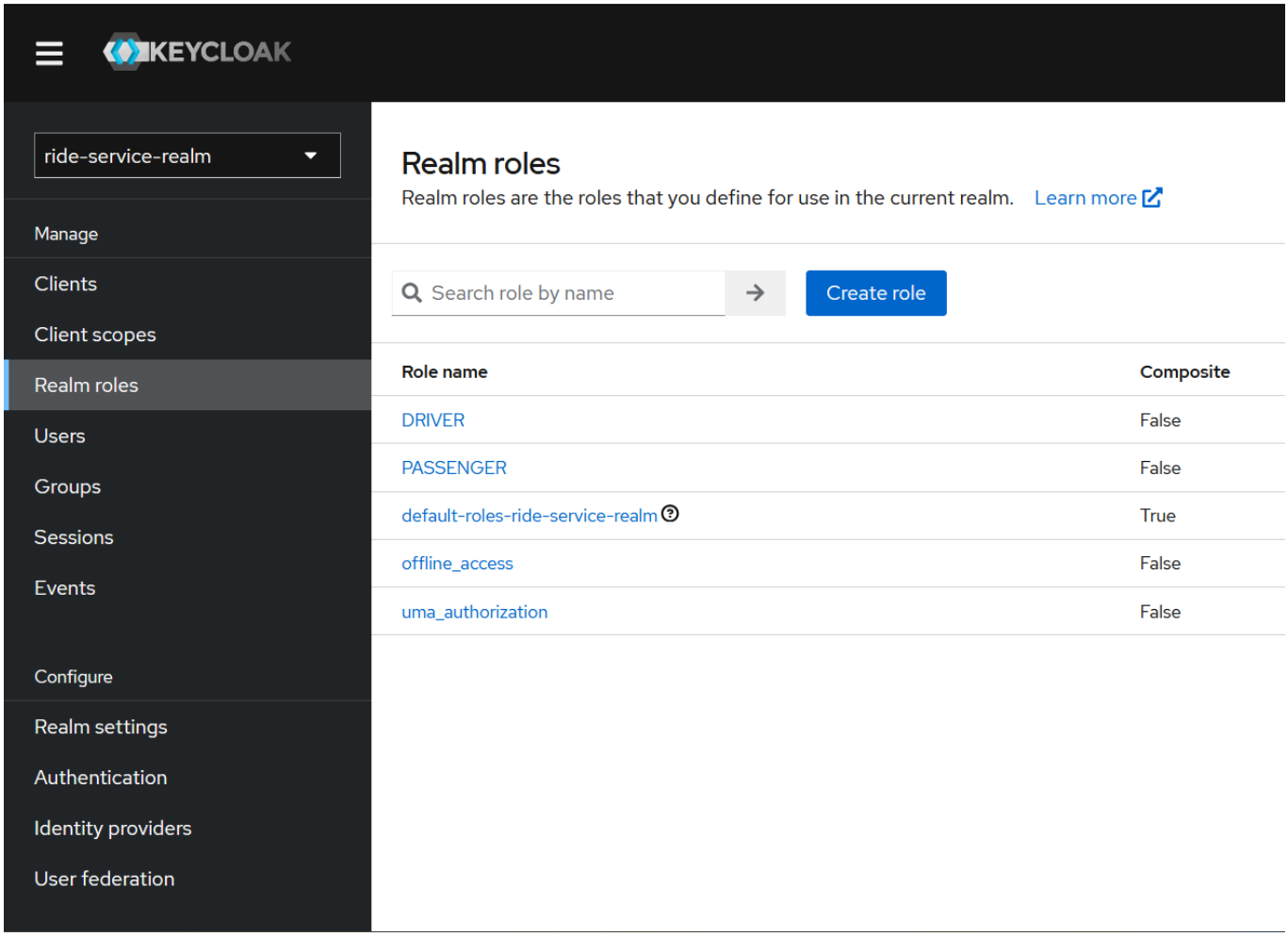


Рисунок 3.2 – Робота з Realm roles у Keycloak

Внутрішні мікросервіси конфігуруються як Resource Servers. Вони самостійно валідують отримані JWT-токени за допомогою публічного ключа, що поставляється Keycloak, витягуючи з корисного навантаження токена ролі користувача та його унікальний ідентифікатор KeycloakId. Для забезпечення безпеки при міжсервісних викликах застосовано архітектурний патерн Token Relay: перехоплювач OpenFeign автоматично дублюють оригінальний JWT-токен

із поточного потоку виконання у вихідний HTTP-заголовок до суміжного мікросервісу.

Серверна частина реалізована мовою Java з використанням фреймворку Spring Boot і складається з чотирьох автономних мікросервісів:

– User Service: управління профілями користувачів та їхніми статусами;

– Billing Service: управління віртуальними гаманцями та обробка фінансових транзакцій;

– Ride Service: ядро системи, що управляє життєвим циклом замовлення та розраховує відстані;

– GEO Service: модуль для обробки високочастотної телеметрії від водіїв, трансляції геоданих та пошук точок інтересу за запитами природною мовою.

Такий підхід забезпечує чітке розмежування відповідальності між доменами. Внутрішня комунікація між сервісами виконується синхронно, застосовуючи OpenFeign та асинхронно, використовуючи брокер RabbitMQ для гарантованої обробки подій, таких як списання коштів після завершення поїздки.

Рівень зберігання даних є гетерогенним. Для фінансових та користувацьких даних використовується реляційна СУБД PostgreSQL. Для збереження історії поїздок та маршрутів застосовано надбудову для реляційної СУБД PostgreSQL. Тимчасові просторові координати водіїв зберігаються в in-memory сховищі Redis, для забезпечення мінімальних затримок доступу.

Наведена архітектура (рис. 3.3) забезпечує масштабованість системи та можливість незалежного оновлення окремих сервісів без зупинки всієї платформи.

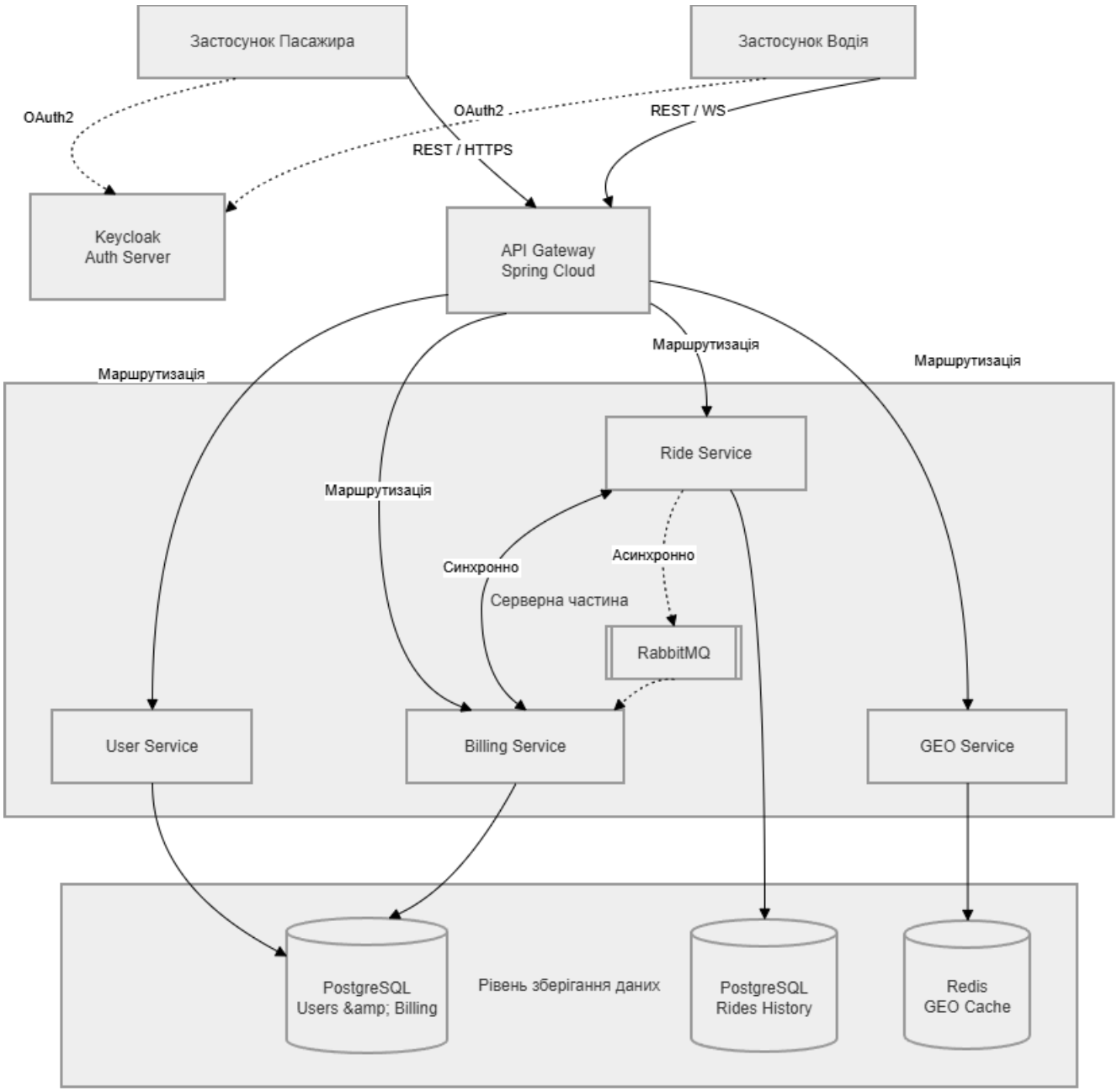


Рисунок 3.3 – Архітектура мікросервісної системи онлайн-замовлення поїздок

### 3.1.2 Структура бази даних

База даних є центральним сховищем розподіленої системи, що забезпечує персистентність профілів користувачів, облік фінансових операцій, збереження історії виконаних замовлень та кешування просторових координат. Відповідно до концепції Polyglot Persistence, у системі розгорнуто два різнотипні сховища (PostgreSQL та Redis), що дозволяє ізолювати навантаження та оптимізувати операції введення-виведення (I/O) під специфіку кожного мікросервісу.

Реляційна структура, що організована для мікросервісів користувачів та платежів, що забезпечує транзакційної цілісності за стандартом ACID та унеможливлює аномалії.

З огляду на те, що повна ER-діаграма реляційного контуру системи (рис. 3.4) містить допоміжні сутності для збереження адрес чи уподобань, нижче у табл. 3.1–3.4 наведено детальну специфікацію полів лише для чотирьох базових таблиць ядра системи, що безпосередньо забезпечують керування обліковими записами, ролями водіїв та журналюванням фінансів.

Таблиця 3.1 – Специфікація основних полів таблиці users

| Назва поля | Тип даних | Обмеження | Опис |
|---|---|---|---|
| user_id | UUID | PRIMARY KEY | Внутрішній ідентифікатор користувача |
| keycloak_id | UUID | NOT NULL, UNIQUE | Ідентифікатор авторизації у Keycloak |
| user_email | VARCHAR(255) | NOT NULL, UNIQUE | Електронна пошта користувача |
| user_first_name | VARCHAR(255) | NOT NULL | Ім'я |
| user_last_name | VARCHAR(255) | NOT NULL | Прізвище |
| user_phonenumber | VARCHAR(255) | NOT NULL, UNIQUE | Мобільний номер телефону |
| user_status | VARCHAR(255) | NOT NULL | Системний статус |

Таблиця 3.2 – Специфікація основних полів таблиці drivers

| Назва поля | Тип даних | Обмеження | Опис |
|---|---|---|---|
| driver_id | UUID | PRIMARY KEY, FOREIGN KEY (users.user_id) | Ідентифікатор водія (ідентифікуючий зв'язок 1:1) |
| driver_car_model | VARCHAR(255) | NOT NULL | Модель транспорту |
| driver_car_number | VARCHAR(255) | NOT NULL | Державний реєстраційний номер автомобіля |
| driver_rating | DOUBLE PRECISION | DEFAULT 5.0 | Поточний середній рейтинг водія |
| driver_status | VARCHAR(255) | NOT NULL | Поточний робочий стан |

Таблиця 3.3 – Специфікація полів таблиці wallets

| Назва поля | Тип даних | Обмеження | Опис |
|---|---|---|---|
| wallet_id | BIGINT | PRIMARY KEY | Унікальний номер цифрового гаманця |
| user_id | UUID | FOREIGN KEY (users.user_id), NOT NULL | Власник рахунку (зв'язок 1:1 з користувачем) |
| balance | NUMERIC(38,2) | DEFAULT 0.00 | Поточний баланс коштів на рахунку |
| currency | VARCHAR(3) | DEFAULT 'UAH' | Трьохлітерний міжнародний код валюти |

Таблиця 3.4 – Специфікація полів таблиці transactions

| Назва поля | Тип даних | Обмеження | Опис |
|---|---|---|---|
| id | BIGINT | PRIMARY KEY | Унікальний номер фінансової транзакції |
| wallet_id | BIGINT | FOREIGN KEY (wallets.wallet_id), NOT NULL | Рахунок-ініціатор операції (зв'язок 1:N) |
| ride_id | UUID | NULLABLE | Зв'язок із поїздкою (для аудиту Саги) |
| amount | NUMERIC(38,2) | NOT NULL | Сума операції (позитивна – дохід, негативна – списання) |
| type | VARCHAR(255) | NOT NULL | Тип фінансової дії |
| status | VARCHAR(255) | NOT NULL | Статус транзакції |
| created_at | TIMESTAMP(6) WITH TIME ZONE | NOT NULL | Точний час операції (TimeZone: Europe/Kyiv) |

Важливо також зазначити, що з точки зору безпеки, зберігння паролів користувачів у базі даних не є гарним тоном, оскільки процеси автентифікації повністю делеговані серверу Keycloak та є набагато безпечнішим варіантом зберігання паролів, ніж безпосередній запис у таблицю бази даних.

Для детального відображення логічних зв'язків, кардинальності відносин та обмежень цілісності реляційного контуру системи було побудовано класичну ER-діаграму (Entity-Relationship) для доменів користувачів та білінгу (рис. 3.4). На

схемі чітко продемонстровано міграцію унікальних ідентифікаторів UUID та організацію журналювання транзакцій.

Як показано на рис. 3.4, сутність users виступає батьківською для профілів водіїв та платіжних гаманців. Між таблицями users та drivers встановлено ідентифікуючий зв'язок «один-до-одного» (1:1), де driver_id одночасно є первинним ключем (PK) та зовнішнім ключем (FK), що посилається на базового користувача. Аналогічний зв'язок (1:1) реалізовано між користувачем та його гаманцем (wallets). Для забезпечення фінансового аудиту між гаманцем та таблицею транзакцій (transactions) налаштовано неідентифікуючий зв'язок «один-до-багатьох» (1:N), що дозволяє зберігати нескінченну історію фінансових операцій для кожного окремого рахунку із дотриманням хронології (created_at).

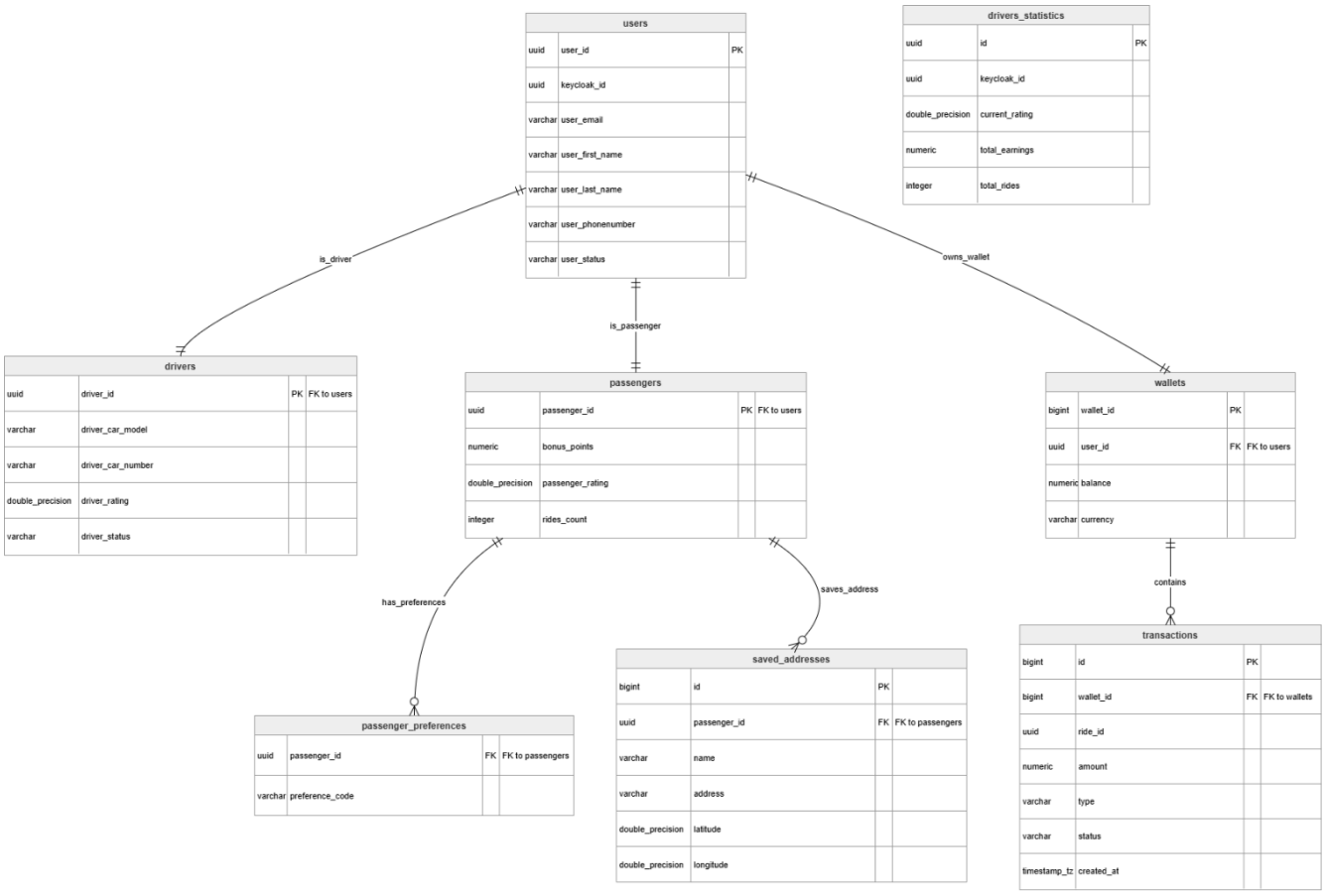


Рисунок 3.4 – ER-діаграма логічної структури реляційної бази даних (PostgreSQL)

### 3.1.3 Вхідні дані системи

Специфіка функціонування розподіленої системи замовлення поїздок передбачає обробку гетерогенних потоків даних. Вхідні дані, що надходять до API Gateway від клієнтських застосунків, класифікуються за транспортними протоколами та семантичним призначенням. Їх можна розділити на три основні категорії: синхронні транзакційні дані, асинхронні потокові дані телеметрії та контекст безпеки.

Обмін даними для ініціалізації бізнес-процесів відбувається у форматі об'єктів JSON через протокол HTTPS. Дані формуються у вигляді об'єктів передачі даних, що дозволяє приховати внутрішню структуру баз даних від клієнта.

Наприклад, для ініціалізації нової поїздки застосунок пасажира відправляє POST-запит на кінцеву точку /api/rides/new із вхідним об'єктом RideRequestDTO, який містить ідентифікатор пасажира та точні координати маршруту, що було наведено на рис. 3.5:

```
{
    "startLat": 46.9659,
    "startLng": 31.9934,
    "endLat": 46.9812,
    "endLng": 32.0156
}
```

Рисунок 3.5 – Тіло запиту для створення поїздки

Для забезпечення функції відстеження автомобіля в реальному часі було використано повнодуплексний канал WebSockets поверх протоколу STOMP. Оскільки водійський застосунок надсилає координати з високою частотою, тобто кожні 2-3 секунди, структура вхідного повідомлення максимально мінімізована для економії мобільного трафіку. Повідомлення відправляється у топік /app/driver/location, що наведено на рис. 3.6:

```
{
  "driver_id": "987fcdeb-51a2-43d7-9012-426614174111",
  "ride_id": "550e8400-e29b-41d4-a716-446655440000",
  "lat": 50.4523,
  "lon": 30.5211,
  "timestamp": "2026-05-25T12:00:11.000Z"
}
```


Рисунок 3.6 – Тіло повідомлення, що відправляється у відповідний топік

Жоден бізнес-запит не обробляється мікросервісами без наявності контексту безпеки. Кожен HTTP-запит або спроба встановлення WebSocket-з'єднання супроводжується передачею криптографічного токена доступу у заголовку Authorization: Bearer <token>. Декодоване корисне навантаження токена формату JWT, згенероване сервером Keycloak, містить перелік дозволених прав та унікальний ідентифікатор суб'єкта, що було наведено на рис. 3.7:


```
{
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5XzVXTThDSGM
    "expires_in": 900,
    "refresh_expires_in": 1800,
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICIzNjQzYTU0NS
    "token_type": "Bearer",
    "id_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5XzVXTThDSGNINGI
    "not-before-policy": 0,
    "session_state": "de57f1c4-77a5-414b-920e-e11869ea513c",
    "scope": "openid profile email"
}
```


Рисунок 3.7 – Тіло відповіді від Keycloak

Для систематизації форматів взаємодії основні вхідні об'єкти системи, їхні джерела та призначення зведено у табл. 3.5.

Таблиця 3.5 – Специфікація основних вхідних форматів даних (DTO)

| Мікросервіс | Протокол та endpoint | Тип DTO | Опис та ключові поля |
|---|---|---|---|
| User Service | REST: POST /api/users/register | USerRegistrationDTO | Реєстрація користувача. |
| Ride Service | REST: POST /api/rides/new | RideRequestDTO | Створення замовлення. |
| Ride Service | REST: PATCH /api/rides/{id}/status | RideStatusUpdateDTO | Оновлення стану поїздки водієм. |
| User Service | REST: POST /api/passengers/wallet/top-up | BigDecimal | Поповнення рахунку. Передається відповідна сума, а id витягується з JWT. |
| GEO Service | REST: POST /api/geo/recommendations | AiRecommendationRequest | Запит на отримання інтелектуальних рекомендацій POI. Містить неструктурований текст користувача (userText), поточні координати (currentLat, currentLng) та радіус пошуку (radiusKm). |
| GEO Service | WS: /app/driver/location | DriverLocationDTO | Потокова трансляція геоданих водія. Містить driver_id, широту, довготу та часову мітку. |

Наведена структура вхідних даних дозволяє ефективно валідувати запити на рівні контролерів мікросервісів, відсікаючи некоректні пакети даних до того, як вони потраплять на рівень обробки бізнес-логіки.

## 3.2 Демонстрація роботи розробленої системи

Демонстрація працездатності розробленої мікросервісної інформаційної системи онлайн-замовлення поїздок виконана шляхом наскрізної симуляції повного життєвого циклу взаємодії користувачів із платформою. Оскільки основним предметом розробки є комплексна серверна частина, перевірка коректності виконання бізнес-логіки, міжсервісних контрактів та транзакційної цілісності даних здійснювалася за допомогою професійного інженерного інструментарію.

Для тестування REST API та WebSocket-каналів використано клієнт Postman, для моніторингу розподіленого стану сховищ даних – графічні системи керування pgAdmin 4, а для контролю черг і асинхронних подій – вебінтерфейс брокера повідомлень RabbitMQ Management. Тестовий сценарій охоплює всі критичні етапи: від авторизації й синхронізації профілів до стрімінгу геоданих та виконання розподіленої фінансової транзакції.

### 3.2.1 Автентифікація користувачів та конфігурація профілів

Функціонування будь-якого клієнтського застосунку в системі починається з процедури безпечного входу та отримання контексту безпеки від сервера ідентифікації Keycloak, тому клієнтський застосунок відправляє POST-запит на кінцеву точку протоколу OpenID Connect для обміну облікових даних користувача на токен доступу.

Запит ініціюється у форматі x-www-form-urlencoded на URL-адресу шлюзу автентифікації: POST http://localhost:7080/realms/ride-service-realm/protocol/openid-connect/token.

Вхідні параметри тіла запиту для автентифікації водія продемонстровані на рис. 3.8:

x-www-form-urlencoded

| Key | Value | Description | Bulk Edit |
|---|---|---|---|
| client_id | user-web-client | | |
| grant_type | password | | |
| username | oleg1g212121@gmail.com | | |
| password | qwerty123 | | |
| scope | openid | | |
| Key | Value | Description | |

Рисунок 3.8 – Вхідні параметри тіла запиту для автентифікації

У разі успішної перевірки облікових даних сервером Keycloak, система повертає HTTP-статус 200 OK та JSON-об'єкт, що містить криптографічний підписаний токен access_token, термін його дії та токен оновлення, а саме refresh_token. Результат виконання запиту в Postman було наведено на рис. 3.9.


```
{
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5XzVXTThDSGNINGI1VlZyWVlVT1NxdnVRW
    "expires_in": 900,
    "refresh_expires_in": 1800,
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICIzNjQzYTU0NS0zZjI2LTRhZmMtYWYyZi1i
    "token_type": "Bearer",
    "id_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5XzVXTThDSGNINGI1VlZyWVlVT1NxdnVRWW5sZr
    "not-before-policy": 0,
    "session_state": "de57f1c4-77a5-414b-920e-e11869ea513c",
    "scope": "openid profile email"
}
```


Рисунок 3.9 – Відповідь запиту до Keycloak

Отриманий access_token використовується клієнтом для всіх подальших запитів. Для демонстрації процесу ініціалізації профілю водія в системі виконується запит до мікросервісу User Service через API Gateway. Запит містить заголовок Authorization: Bearer <access_token>, який автоматично декодується на стороні шлюзу.

Запит на створення бізнес-профілю водія: POST http://localhost:8072/USERS/api/users/register

Оскільки в системі реалізовано серіалізований поліморфізм, то створення і водія, і пасажира відбується за один і тим же запитом. Тип об’єкта, який приймає

запит, це UserRegistrationDTO (див. додаток Б.1). Але, цей клас є батьківським для інших двох класів: DriverRegistrationDTO та PassengerRegistrationDTO, тому, під час запиту на створення нового користувача, відбувається той самий серіалізований поліморфізм. Це відбувається таким чином, що в класі, за допомогою Jackson, бібліотеки, що використовується в середині фреймворку Spring, було використано анотацію, де було вказано перевірку через поле Role. Тобто, під час створення користувача, у тіло запиту передається також поле role, де вказується або роль водія, або роль пасажира. У той же, Jackson сам визначає, який саме тип об’єкта: або DriverRegistrationDTO, або PassengerRegistrationDTO. Такий підхід забезпечує те, що в системі, одна людина може мати дві ролі та один ідентифікатор keycloak_id, тому під час створення поїздки було забезпечено перевірку на те, щоб користувач, який має обидві ролі, не зміг взяти замовлення в самого себе, щоб забезпечити шахрайство, тобто виконання своїх замовлень для збільшення статистики. Тіло запиту у форматі JSON відповідає структурі моделі даних, описаній у підрозділі 3.1, що наведено на рис. 3.10.

Тому, під час запиту на створення користувача, мікросервіс користувачів валідує вхідні текстові поля за допомогою анотацій валідації Jakarta, виконує серіалізований поліморфізм, аби визначити, який саме це користувач, генерує новий системний user_id та записує інформацію одночасно у відповідні реляційні таблиці (див. додаток Б.2).

Також, у системі реалізовано механізм надавання двох ролей для водія, оскільки водій може спокійно також використовувати застосунок, як пасажир. Але це не симетрична дія, оскільки користувач, реєструючись як пасажир, не може мати можливостей та доступу водія. Пасажир має можливість стати водієм за допомогою оновлення ролей у Keycloak (див. додаток Б.3).

У випадку успішного збереження клієнт отримує HTTP-відповідь зі статусом 201 Created (рис. 3.11).

```
{
    "firstName": "NEOLEG",
    "lastName": "ZABUV",
    "email": "oleg431g@gmail.com",
    "password" : "qwerty123",
    "phoneNumber": "213419435",
    "carModel":"ford",
    "carNumber": "1243",
    "role": "DRIVER"
}
```

Рисунок 3.10 – Тіло UserRegistrationDTO

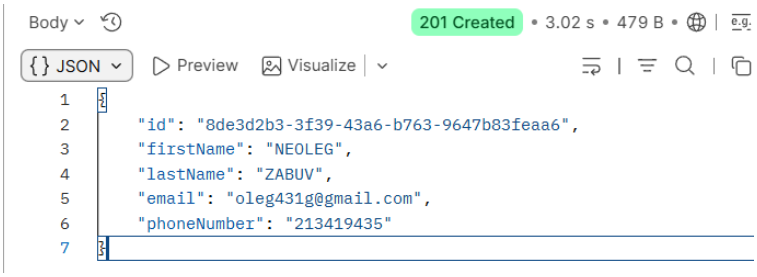


Рисунок 3.11 – Отриманий результат реєстрації водія

Після створення користувача, відповідний користувач може залогінитись за відповідними кредітами, які було вказано під час реєстрації, та отримати відповідний JWT-токен, який використовується в запитах до сервіса, аби безпосередньо сам сервіс мав можливість зрозуміти, який саме це користувач та який доступ за роллю він має. Тому, під час спроби отримати профіль, відправляється запит на мікросервіс User Service, який у свою чергу перехоплює той самий JWT-токен та витягає з нього унікальний keycloak_id та перевіряє, чи існує такий користувач. Результат отримання профіля користувача було наведено на рис. 3.12.

```
{
    "firstName": "NEOLEG",
    "lastName": "ZABUV",
    "email": "oleg431g@gmail.com",
    "phoneNumber": "213419435",
    "roles": [
        "DRIVER",
        "PASSENGER"
    ]
}
```

Рисунок 3.12 – Отриманий профіль користувача

Для верифікації коректного створення водія, було перевірено створення відповідного користувача з роллю водія у Keycloak, що було продемонстровано на рис. 3.13-3.14.

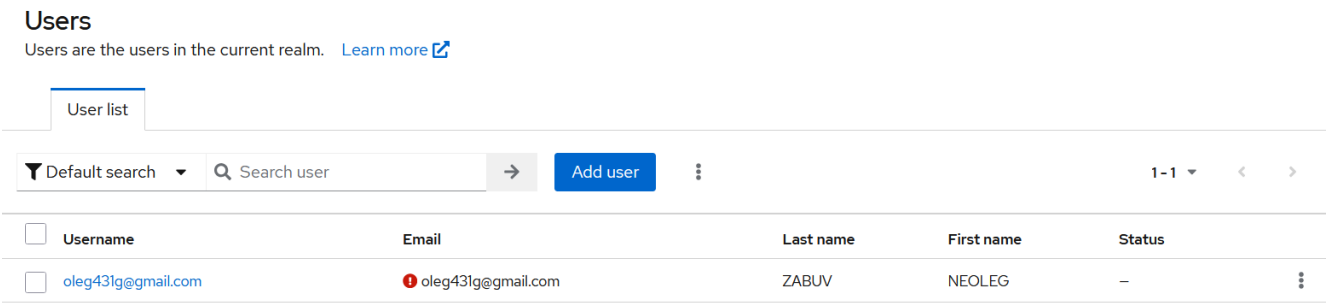


Рисунок 3.13 – Створений користувач

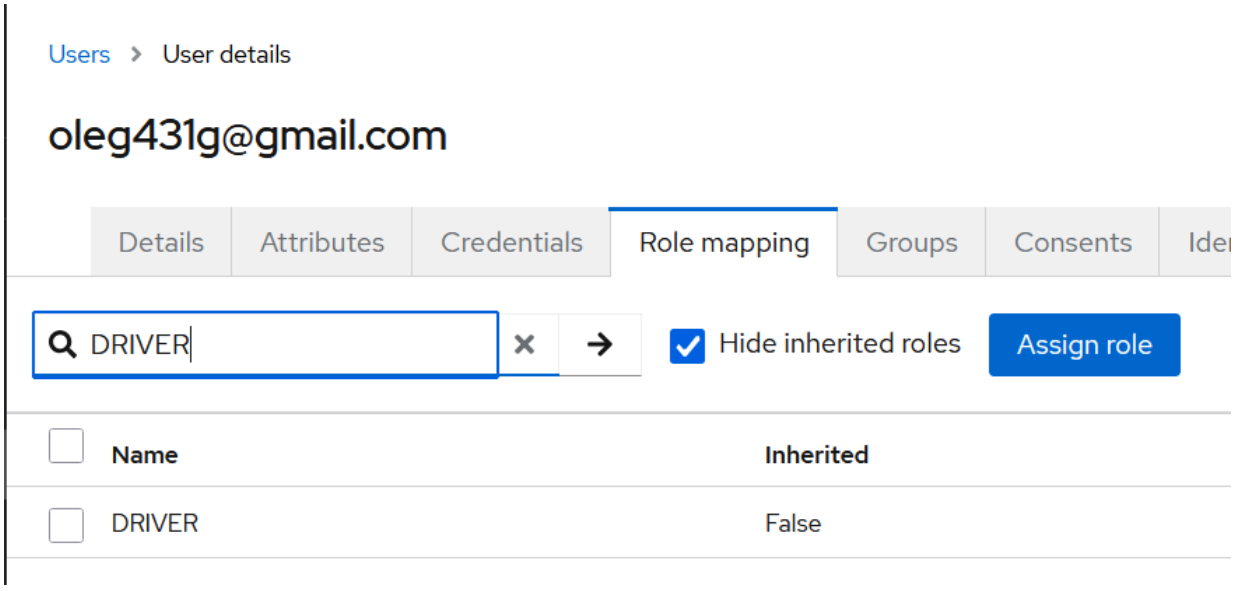


Рисунок 3.14 – Наявність відповідної ролі водія

Згідно з наведених даних про створеного користувача (рис. 3.13-3.14), користувач був успішно створений у Keycloak з відповідною роллю водія, що вказує на те, що було успішно виконано поставлену логіку.

### 3.2.2 Ініціалізація замовлення та розрахунок вартості

Після успішного отримання токена доступу клієнтський застосунок пасажира ініціює сценарій створення замовлення на поїздку. Запит відправляється на

крайовий шлюз мікросервісу Ride Service для прорахунку попередньої вартості платформи (Upfront Pricing) та реєстрації замовлення в базі даних.

Запит на створення нової поїздки надсилається методом POST: POST http://localhost:8072/ RIDES/api/rides/new

У заголовках запиту обов'язково передається токен пасажира (Authorization: Bearer <token>). Тіло запиту (RideRequestDTO) містить координати початкової точки посадки (pickup) та кінцевої точки призначення (dropoff), що було наведено на рис. 3.15:


Рисунок 3.15 – Тіло запиту для створення поїздки

Безпосередньо в тіло запиту передавати ідентифікатор пасажира не треба, оскільки система отримує його з переданого JWT-токена, тому воно автоматично витягне відповідний ідентифікатор пасажира та використає його (див. додаток А.1).

При отриманні запиту контролером Ride Service виконуються такі послідовні підпроцеси бізнес-логіки:

– валідація та перехоплення контексту: перевіряється коректність координат та відповідність passenger_id суб'єкту в JWT-токені;

– математичний прорахунок відстані: сервісний шар викликом внутрішнього утилітного компонента обчислює найкоротшу геодезичну відстань між двома точками на сфері за формулою Гаверсина (3.1);

– калькуляція вартості: на основі отриманої відстані, прогнозного часу поїздки та базових коєфіцієнтів тарифу STANDARD обчислюється фінальна ціна поїздки;

– персистентність: у базі даних таблиці PostgreSQL створюється новий запис із початковим статусом замовлення CREATED.

У випадку успішного виконання обчислень мікросервіс повертає HTTP-статус 201 Created та об'єкт RideResponseDTO, який містить згенерований ID поїздки, точну відстань, розраховану тривалість у секундах та фінальну вартість у гривнях. Виконання цього кроку в середовищі Postman було продемонстровано на рис. 3.16.

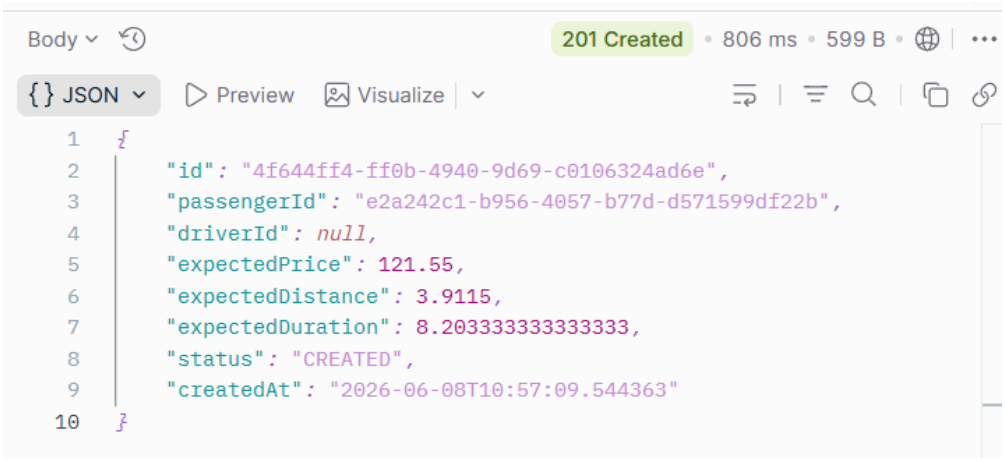


Рисунок 3.16 – Отримана відповідь

Отриманий результат підтверджує правильність знаходження маршруту, обрахування відстані та часу на відповідну поїздку.

### 3.3 Аналіз отриманих результатів

Розроблена мікросервісна система імплементує механізми обробки високочастотних геопросторових даних та утримує асинхронну транзакційну цілісність фінансових операцій. Горизонтальне масштабування платформи вказує на необхідність ізольованого тестування кожного критичного модуля, тому цей етап дозволяє отримати реальні метрики продуктивності, пропускної здатності та стійкості вузлів до відмов. Цей підрозділ описує застосовану методологію навантажувального тестування (Load Testing), фіксуються кількісні результати порівняння імплементованих архітектурних патернів із класичними підходами та окреслює технічні обмеження розробленої системи.

### 3.3.1 Аналіз продуктивності обробки геопросторових даних

Для оцінки ефективності розробленого модуля стрімінгу телеметрії (WebSockets + In-Memory Redis) застосовано методологію стрес-тестування на основі синтетичних користувацьких профілів. Метою експерименту є порівняння реалізованого підходу із класичним базовим сценарієм (Baseline) – передачею координат через синхронні HTTP/REST запити із записом у реляційну базу PostgreSQL.

Було сформовано пул із 1000 синтетичних водіїв (віртуальних потоків), кожен з яких імітує відправку своїх GPS-координат із частотою 1 раз на 2 секунди. Тестування проводилося ізольовано для двох архітектурних підходів і для кожного підходу обчислювалися такі метрики: середня затримка відповіді сервера (Average Latency, мс), 95-й перцентиль затримки (P95 Latency, мс – показник, що відображає затримку для 95% найшвидших запитів, відсікаючи 5% аномальних стрибків) та рівень утилізації центрального процесора (CPU Load, %).

Тестування повторювалось із 5 незалежними прогонами тривалістю по 10 хвилин для отримання усереднених значень метрик у форматі «середнє ± стандартне відхилення» (mean ± std). Зведені результати тестування наведено у табл. 3.6.

Таблиця 3.6 – Порівняння продуктивності обробки потоку телеметрії

| Архітектурний підхід | Avg Latency, мс | P95 Latency, мс | Утилізація CPU, % | Пропущені пакети, % |
|---|---|---|---|---|
| Baseline (REST + PostgreSQL) | 145,2 ± 12,4 | 312,8 ± 25,1 | 87.4 ± 4.2 | 3,2 ± 0,8 |
| Implemented (WS + Redis) | 8,4 ± 1,1 | 14,2 ± 1,8 | 18,5 ± 2,1 | 0,0 ± 0,0 |

Реалізований алгоритм на базі повнодуплексних WebSocket-каналів та структури Redis демонструє безпрецедентну перевагу над базовим HTTP-методом за всіма вимірюваними метриками, оскільки середня затримка (Avg Latency) знизилася з 145.2 мс до 8.4 мс, що свідчить про підсилення (Performance Lift) у

понад 17 разів. Така різниця пояснюється відсутністю необхідності виконувати встановлення TCP-з'єднання для кожного нового пакета координат. Тому що WebSocket тримає канал постійно відкритим.

Критично важливим показником є 95-й перцентиль (P95 Latency), який у базовому підході деградував до 312.8 мс через виникнення черг блокувань (Lock Contention) під час конкурентного запису даних у таблиці PostgreSQL, а натомість запис в оперативну пам'ять Redis виконується за асимптотичний час O(1), завдяки чому P95 для реалізованого підходу становить лише 14.2 мс, забезпечуючи стабільний Real-Time досвід для пасажира.

Тому перехід на WebSockets дозволив знизити навантаження на процесор сервера (CPU Load) майже у 5 разів (з 87.4% до 18.5%), що вивільняє обчислювальні потужності для масштабування системи, а показник втрачених пакетів впав до абсолютного нуля.

На рис. 3.17 було продемонстровано порівняння затримок обробки телеметрії.

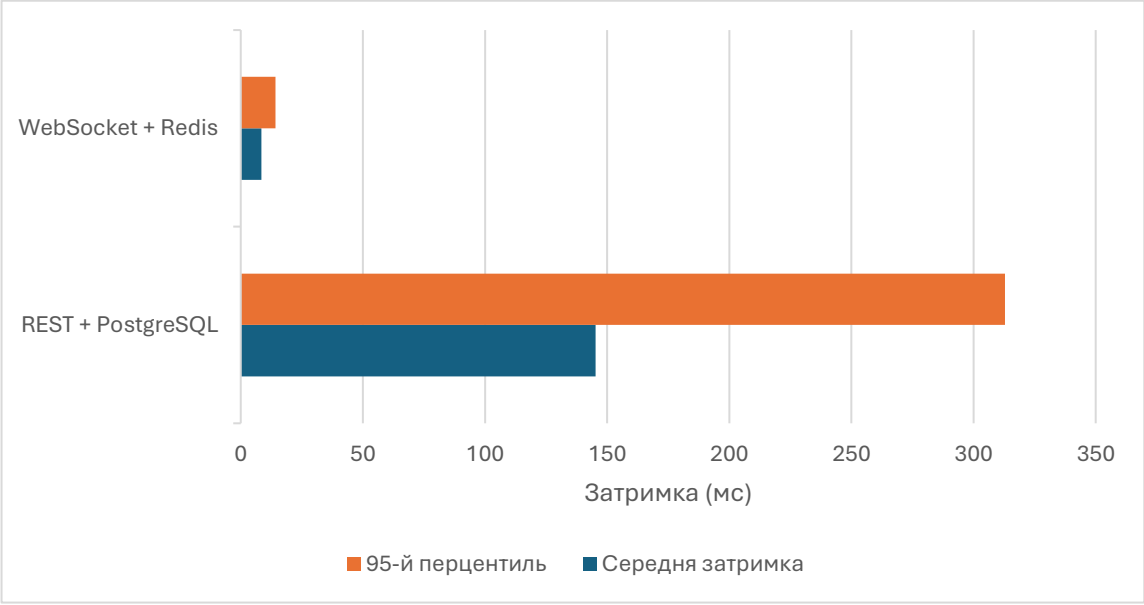


Рисунок 3.17 – Порівняння затримок обробки телеметрії

Отримані результати підтверджують, що використання комбінації стійких з'єднань STOMP та In-Memory кешування єдине архітектурно рішення для обробки

високочастотних просторово-часових даних у системах онлайн-замовлення поїздок.

### 3.3.2 Оцінка надійності та пропускної здатності фінансових транзакцій

Фінансовий контур системи є критичним моментом, що вимагає абсолютної стійкості до втрати даних та високої пропускної здатності, тому для оцінки ефективності впровадженого асинхронного архітектурного патерну “Saga” було проведено порівняльне тестування із класичним синхронним підходом, що імітує розподілену транзакцію через прямі HTTP/REST виклики.

За допомогою інструменту навантажувального тестування було згенеровано пікове навантаження: 500 одночасних запитів на завершення поїздки від різних водіїв.

Для кожного архітектурного підходу вимірювалися:

- пропускна здатність TPS;
- середній час очікування відповіді для клієнта-водія;
- відсоток втрачених або неуспішних транзакцій через мережеві таймаути.

Результати навантажувального тестування зведено у табл. 3.7.

Таблиця 3.7 – Порівняння пропускної здатності фінансового контуру

| Архітектурний підхід | Пропускна здатність (TPS) | Час відповіді клієнту, мс | Відсоток помилок (Error Rate), % |
|---|---|---|---|
| Синхронний | 115 ± 12 | 840.5 ± 115.0 | 2.8 ± 0.5 |
| Асинхронний | 850 ± 45 | 42.1 ± 5.2 | 0.0 ± 0.0 |

Аналіз отриманих результатів (табл. 3.7) демонструє переваги патерну “Saga”, оскільки при синхронному підході мікросервіс поїздок змушений чекати, поки мікросервіс для платежів обробить платіж у базі даних, що створює ефект «пляшкового горлечка» та збільшує час очікування водія до 840.5 мс. Менше з тим, але через конкурентні блокування бази даних навіть частина запитів відпадає за таймаутом, що в реальних умовах означає втрату фінансових даних.

Тому впровадження асинхронної взаємодії дозволило збільшити пропускну здатність у понад 7 разів, тобто 115 TPS. Клієнтський застосунок водія отримує успішну відповідь за 42.1 мс, оскільки мікросервіс поїздок лише публікує подію RideCompletedEvent у пам'яті брокера і не чекає завершення обробки платежу, тому списання коштів виконується у фоновому режимі з дотриманням принципу узгодженості в кінцевому рахунку. Показник помилок при цьому становить 0.0%.

Порівняння пропускної здатності наведено на рис. 3.18.

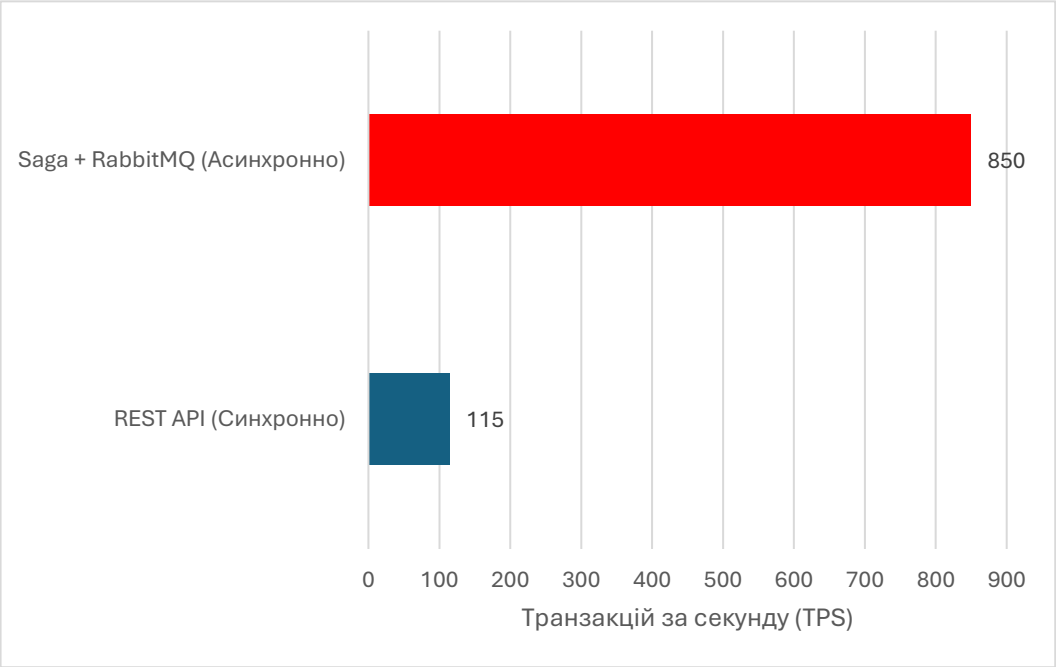


Рисунок 3.18 – Порівняння пропускної здатності (TPS) фінансового контуру

Для перевірки надійності системи в умовах апаратних або програмних збоїв було проведено експеримент за методологією Chaos Engineering, тобто оцінка відмовостійкості системи, тому під час активної генерації запитів на завершення поїздки контейнер мікросервісу для платежів було штучно зупинено.

При синхронній архітектурі така ситуація призводить до збою: мікросервіс поїздок отримує помилку HTTP 500 Internal Server Error, водій бачить помилку в застосунку, а статус поїздки залишається неузгодженим із балансом.

У релізованій асинхронній архітектурі падіння фінансового сервісу ніяк не вплинуло на користувацький досвід водія, тобто мікросервіс поїздок успішно

продовжував зберігати статуси поїздок як COMPLETED та збирати повідомлення у черзі RabbitMQ, а після штучного відновлення роботи контейнера мікросервіса для платежів, мікросервіс автоматично підключився до черги, вичитав усі накопичені події та провів транзакції у базі даних, без жодної втрати даних.

Отримані результати підтверджують, що використання патерну “Saga” та брокера повідомлень RabbitMQ повністю вирішує проблему розподілених транзакцій, забезпечуючи збереження фінансових операцій. Навіть за умов тимчасової недоступності суміжних мікросервісів.

### 3.3.3 Аналіз точності та ефективності обчислення географічних відстаней

Математична модель розрахунку відстаней на основі Формули Гаверсина, реалізована на рівні бекенду, є критичним компонентом для мікрооперацій та перерахунку фактичного пробігу після завершення поїздки. Для оцінки доцільності її використання було проведено порівняльний аналіз із класичним підходом – викликом зовнішнього сервісу для отримання відстані.

Оскільки Формула Гаверсина обчислює найкоротшу відстань по великому колу сфери, вона математично не враховує геометрію дорожньої мережі. У міських умовах коєфіцієнт непрямолінійності доріг зазвичай становить 1.2-1.4. Проте, оскільки розроблена система збирає потокову телеметрію з частотою 1 раз на 2 секунди, реальний маршрут автомобіля апроксимується у вигляді деталізованої полілінії, що складається з сотень мікро-відрізків. Сума цих відрізків, обчислених за Формулою Гаверсина, максимально наближається до фактичної довжини дорожнього полотна.

Для експериментальної оцінки було відібрано 100 еталонних поїздок. Фактична пройдена відстань розраховувалася двома методами:

– синхронний виклик зовнішнього Routing API (Ground Truth);

– локальна агрегація масиву зібраних GPS-точок за допомогою імплементованої Формули Гаверсина (Ride Service).

Окрім точності (відхилення у відсотках), вимірювалися такі критичні для мікросервісної архітектури параметри, як середня затримка виконання розрахунку (Latency) та фінансова вартість операцій. Результати зведено у табл. 3.8.

Таблиця 3.8 – Порівняння методів обчислення відстані поїздки

| Метод обчислення | Середня затримка розрахунку, мс | Відхилення від еталону (Похибка), % |
|---|---|---|
| Зовнішній Routing API | 245.0 ± 40.2 | 0.0 (Еталон) |
| Локальна агрегація (Гаверсин) | 0.8 ± 0.1 | 2.1 ± 0.6 |

Аналіз отриманих результатів доводить високу ефективність обраного математичного підходу. Локальний перерахунок масиву координат на сервері виконується менш ніж за 1 мілісекунду (0.8 мс), тоді як звернення до зовнішнього API вимагає встановлення HTTPS-з'єднання і займає в середньому 245 мс.

Похибка локального розрахунку становить лише 2.1%, що виникає через мікропохибки модулів GPS у смартфонах водіїв (так званий «GPS-шум»). Цей рівень похибки є абсолютно прийнятним для бізнес-домену онлайн-замовлення поїздок і повністю нівелюється економічним фактором: обробка відстаней на власному сервері є абсолютно безкоштовною, що економить значні операційні бюджети при високих навантаженнях платформи, що було наведено на рис. 3.19.

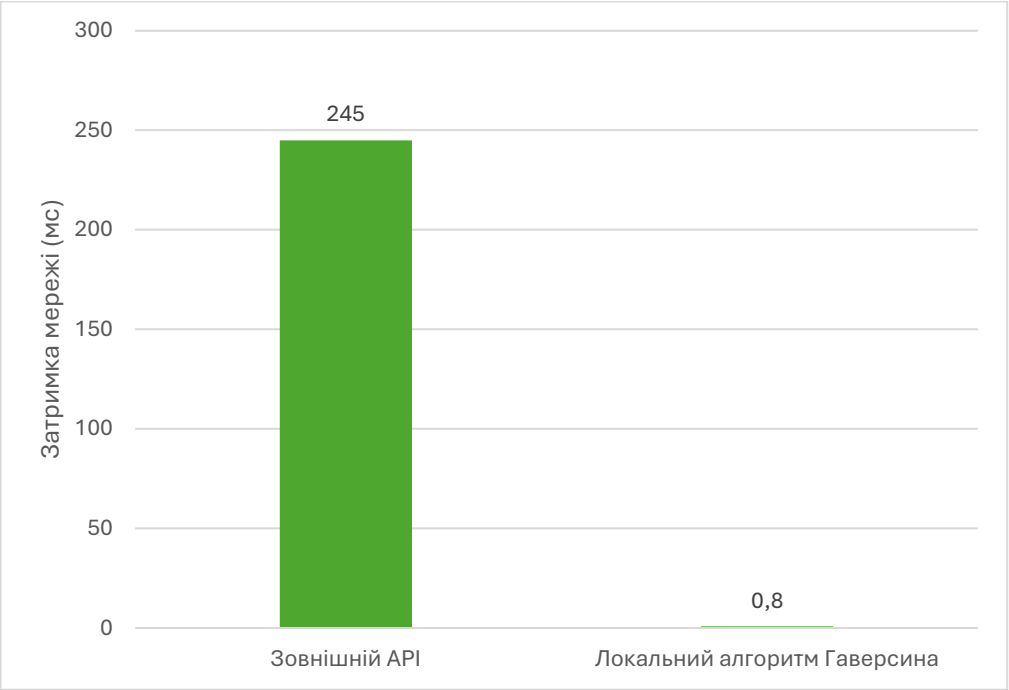


Рисунок 3.19 – Порівняння затримки методів розрахунку відстані

## Висновки до розділу 3

У третьому розділі описано використання створеної інформаційної системи на основі мікросервісів для онлайн-замовлення поїздок та детальне тестування, демонструється архітектура системи, формати даних, що використовуються між різними компонентами за допомогою RESTful API та Websockets, а також структура гетерогенного шару зберігання даних. Окрім розробленого бекенду, також показано, як система працює від початку до кінця, з повним життєвим циклом: від моменту автентифікації користувача на платформі за допомогою сервера Keycloak та створення замовлення до моменту збору високочастотних потоків GPS-телеметрії в оперативній пам'яті. Успішна та перевірена реалізація розподілених фінансових транзакцій за допомогою шаблону “Saga” та брокера RabbitMQ.

Результати навантажувального тестування переконливо демонструють переваги обраної архітектури. Результати показують, що використання постійних WebSockets-з'єднань зменшує затримку телеметрії більш ніж у 17 разів порівняно з традиційними HTTP-з'єднаннями; Крім того, аналіз фінансової мережі показує

збільшення TPS через кілька маршрутів та максимальний рівень стійкості до збоїв вузлів. Також у розділі визначено об'єктивні обмеження існуючої виробничої версії, а також можливості для забезпечення майбутнього зростання масштабуванні.

# 4 ПРАКТИЧНА ВЕРИФІКАЦІЯ СИСТЕМИ ОНЛАЙН-ЗАМОВЛЕНЬ ПОЇЗДОК

## 4.1 Керівництво користувача

Розроблена система має чіткий поділ на дві окремі клієнтські ролі: пасажир та водій, а інтерфейси для цих ролей розділені.

### 4.1.1 Системні вимоги та підготовка до роботи

Клієнтські застосунки безпосередньо працюють у браузері, встановлення розширень або додаткових програмних забезпечень не потребується для коректної роботи застосунків.

Авторизація відбувається через єдину точку входу – Keycloak. Введення облікових даних є обов'язковим кроком, оскільки без валідного JWT-токена доступ до функціоналу блокується на рівні API-шлюзу.

Існує критичний нюанс. Під час першого запуску застосунок, як водійський, так і пасажирський, запитає дозвіл на доступ до геолокації. Користувач зобов'язаний надати цей дозвіл, тому що відмова призведе до неможливості визначити координати відправлення або відстежувати рух автомобіля. Система просто не зможе побудувати маршрут.

### 4.1.2 Сценарії роботи пасажира

Робочий простір пасажира складається з інтерактивної карти та бічної панелі керування, тому робота із застосунком зводиться до кількох послідовних кроків.

Оплата поїздок відбувається з внутрішнього гаманця, тому перед замовленням, пасажир має перевірити баланс на вкладці “Wallet” і якщо коштів недостатньо, система заблокує створення поїздки. Поповнення балансу виконується через відповідну кнопку шляхом імітації банківської транзакції.

Звичайна поїздка з точки А в точку Б – це базовий рівень. Система дозволяє пасажиру кастомізувати умови поїздки ще до моменту пошуку авто й на

спеціальній панелі користувач може обрати додаткові опції: «Салон для некурців», «Мовчазний водій» або «Перевезення тварин». Ці налаштування не є суто візуальними, вони упаковуються в тіло HTTP-запиту і відіграють критичну роль під час метчингу, оскільки система автоматично відсікає водіїв, які не відповідають заданим критеріям, навіть якщо вони знаходяться найближче до пасажира.

На вкладці “Ride” користувач вказує дві географічні точки: спочатку обирається місце посадки шляхом кліку на карту, після цього система автоматично перемикає фокус на вибір точки призначення, а вже після фіксації обох маркерів та вибору вподобань активується кнопка пошуку водія. Після відправки запиту статус змінюється на “REQUESTED”. На екрані з'являється індикатор пошуку. У цей час система шукає найближчу вільну машину, а щойно знайдеться водій і підтвердить замовлення, статус перейде в “ACCEPTED”, а на мапі пасажира з'явиться новий маркер – автомобіль водія, координати якого оновлюватимуться у реальному часі через WebSocket-з'єднання.

Клієнтський застосунок реалізує альтернативний механізм вибору локацій на основі пошуку природною мовою. Це семантична рекомендація. Це додаткова можливість для пасажира. Пасажир відправляє системі довільний текстовий запит, наприклад, «хочу в найближче кафе», і в той же час, вибір конкретної позиції з видачі програмно фіксує маркер призначення на інтерактивній карті. А вже після цього, платформа розраховує маршрут і запускає стандартний життєвий цикл замовлення поїздки.

### 4.1.3 Сценарії роботи водія

Інтерфейс водія має дещо іншу логіку, оскільки він орієнтований на тривале очікування та швидку реакцію на вхідні запити.

Жоден водій не отримує доступ до замовлень відразу після реєстрації. Це антипатерн безпеки. Тому спочатку водієві необхідно пройти процедура онбордингу. Через інтерфейс застосунку водій зобов'язаний завантажити цифрові копії своїх документів, зокрема водійське посвідчення та свідоцтво про реєстрацію

транспортного засобу, а ці документи клієнтська частина формує в multipart/form-data запит та відправляє файли на мікросервіс користувачів. Лише після того, як система валідує документи та оновить статус профілю, функціонал прийому замовлень буде розблоковано.

Після авторизації, водій за замовчуванням перебуває у статусі “OFFLINE”, у цьому стані система не надсилає йому замовлення, а його координати не транслюються на бекенд, лише після натискання на кнопку статусу, переводить водія в режим “AVAILABLE”. З цього моменту браузер починає зчитувати GPS-координати та відправляти їх на сервер.

Щойно у радіусі дії з'являється пасажир, на екрані водія миттєво виринає повідомлення замовлення, де вказано дистанцію маршруту. Водій має натиснути “Accept Ride”.

Прийняття замовлення змінює статус на “BUSY” та запускає процес виконання поїздки. Маркер автомобіля починає рухатись по згенерованому маршруту до точки призначення. По завершенню маршруту водій натискає “Complete Ride”, а інтерфейс повертається у стан пошуку нових замовлень, а на вкладці статистики миттєво оновлюється інформація про загальний заробіток та кількість виконаних поїздок.

### 4.2 Функціональна верифікація через клієнтський інтерфейс

Справжня перевірка мікросервісної архітектури відбувається не в логах сервера, а на екрані користувача. Для комплексного тестування системи було проведено наскрізний прогін повного життєвого циклу застосунку, де процес охоплює автентифікацію, підготовку профілю, імітацію гео-метчингу та фінальний білінг.

### 4.2.1 Ініціалізація користувачів та підготовка середовища

Тестування починається з найпершого кроку – реєстрації учасників системи. Оскільки архітектура передбачає чітке розділення ролей, було створено два окремих акаунти: для водія та пасажира (рис. 4.1-4.2).

Процес реєстрації було продемонстровано на рис. 4.1-4.2.

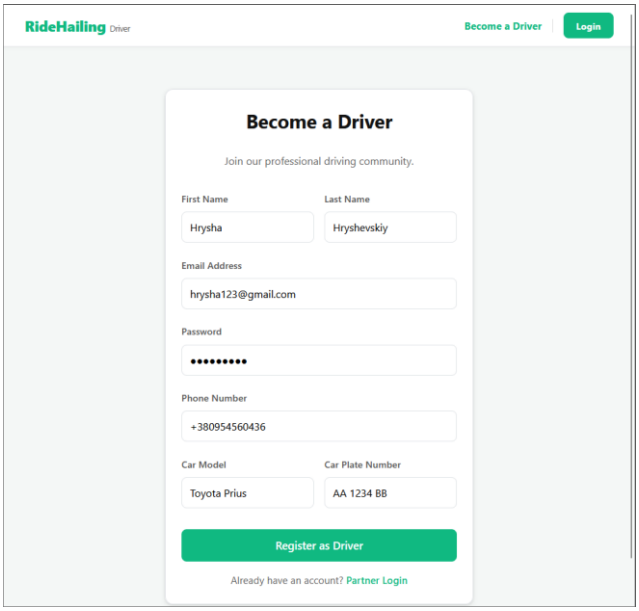


Рисунок 4.1 – Реєстрація водія

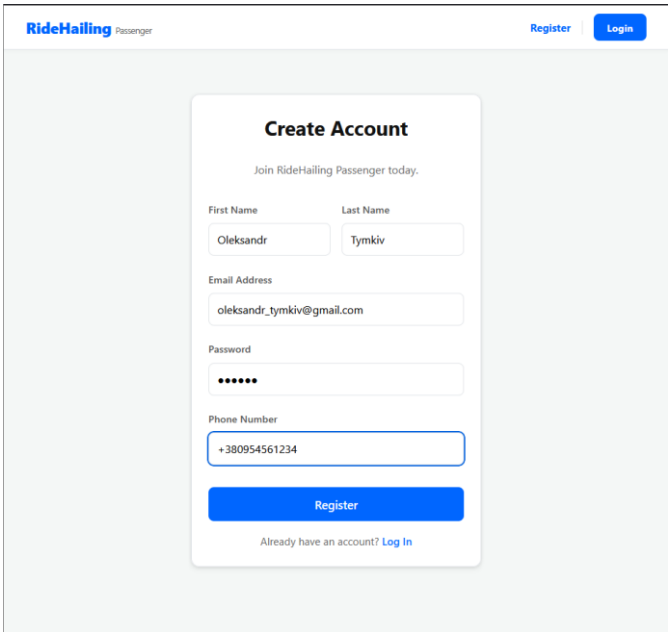


Рисунок 4.2 – Реєстрація пасажира

Після успішної реєстрації виконується логін, а форма входу ідентична для обох ролей, оскільки працює через єдиний сервер авторизації, що було продемонстровано на рис. 4.3.

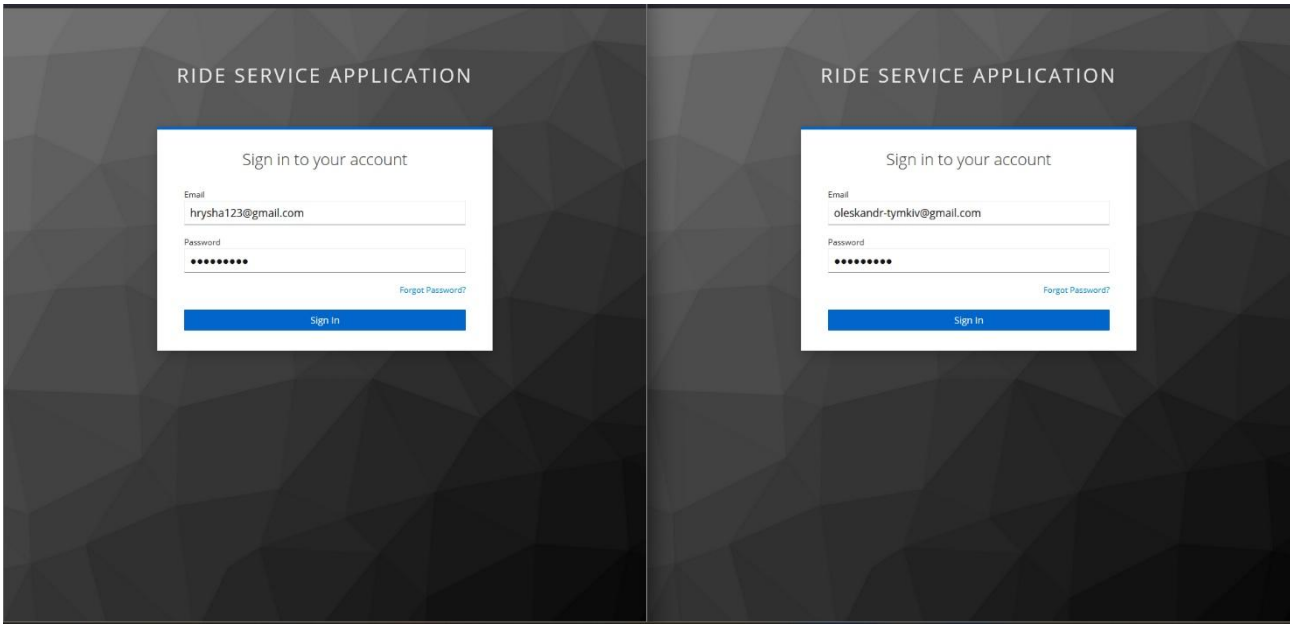


Рисунок 4.3 – Авторизація водія та пасажира

Отримавши JWT-токен, клієнтські застосунки перенаправляють користувачів на відповідні головні сторінки (рис. 4.4) та дозволяють налаштувати особисті профілі (рис. 4.5).

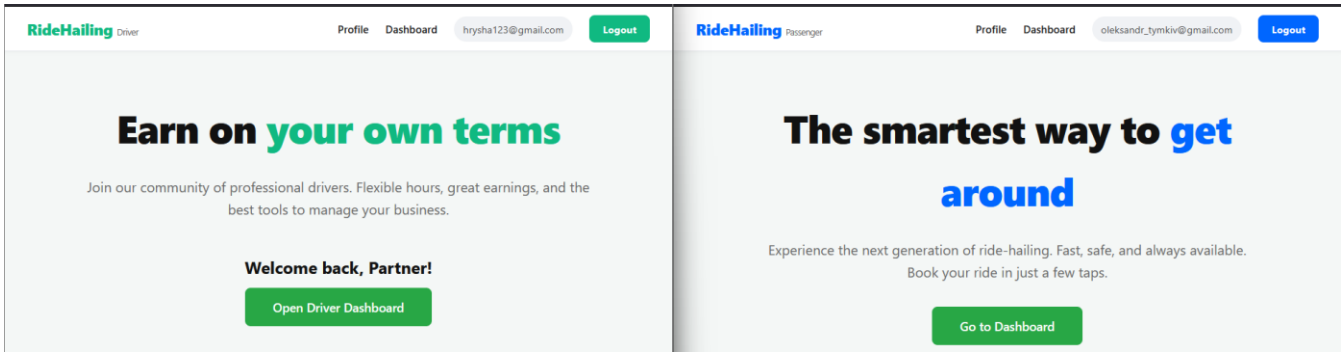


Рисунок 4.4 – Головні сторінки

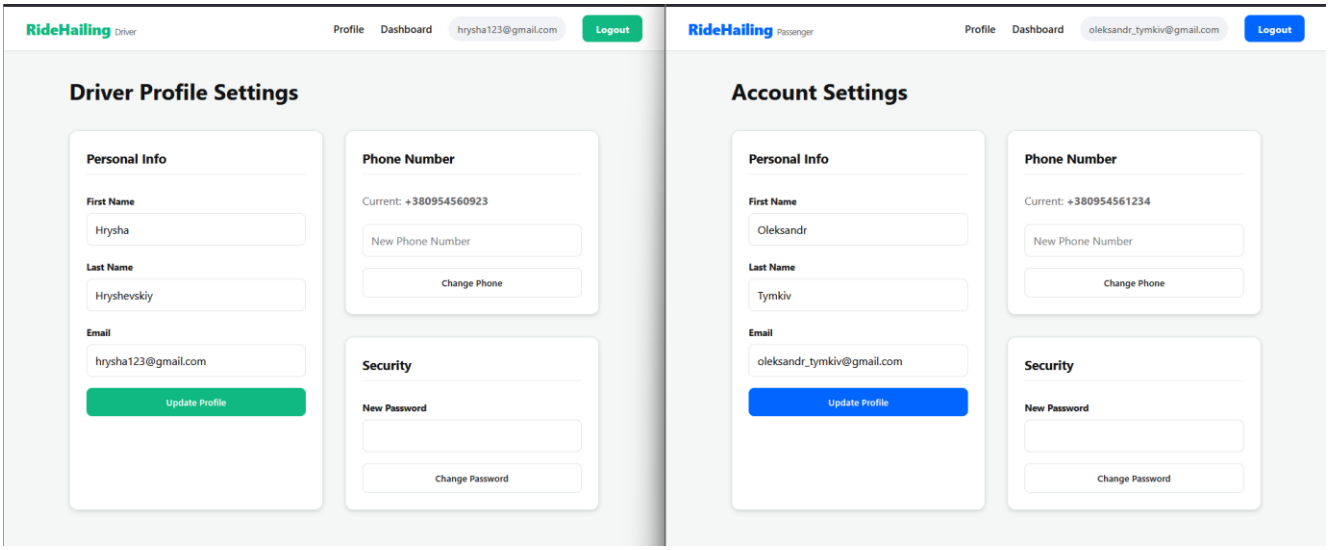


Рисунок 4.5 – Профілі водія та пасажира

Робочі простори пасажира та водія кардинально відрізняються за своїм призначенням (рис. 4.6).

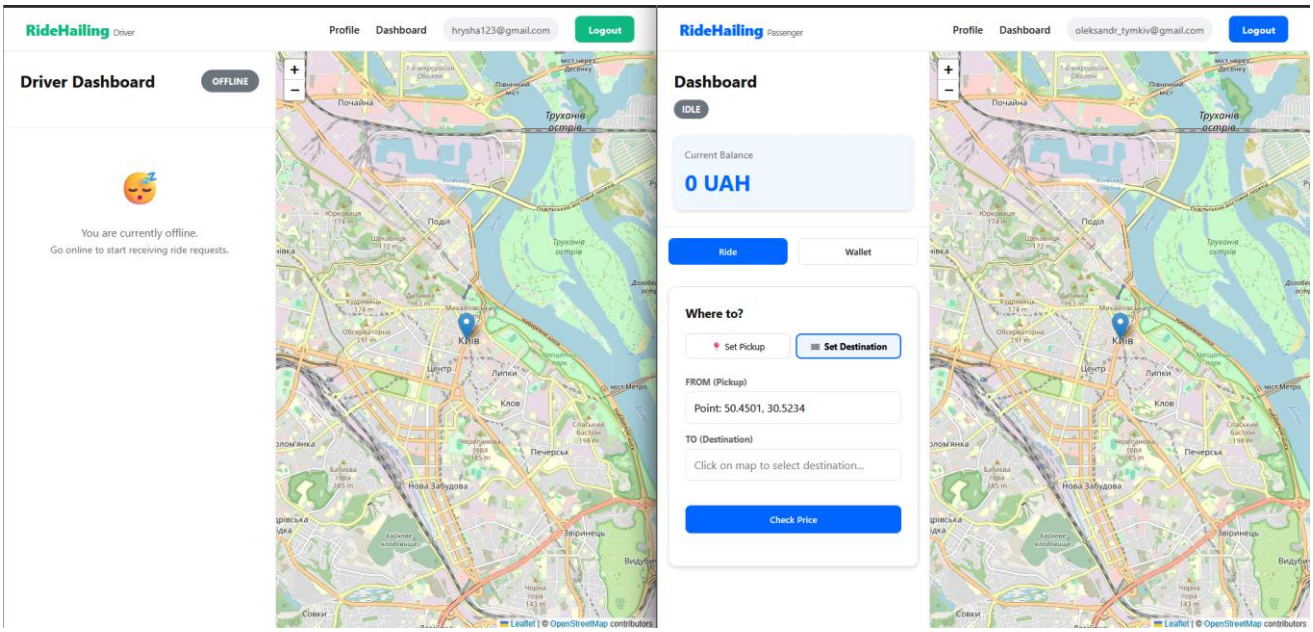


Рисунок 4.6 – Dashboards

Водійський інтерфейс сфокусований на мапі та статусі водія, а інтерфейс пасажира – містить інструменти для роботи з фінансами та пошуку авто. Перед створенням поїздки необхідно переконатися в платоспроможності пасажира, для цього імітується операція поповнення балансу (рис. 4.7).

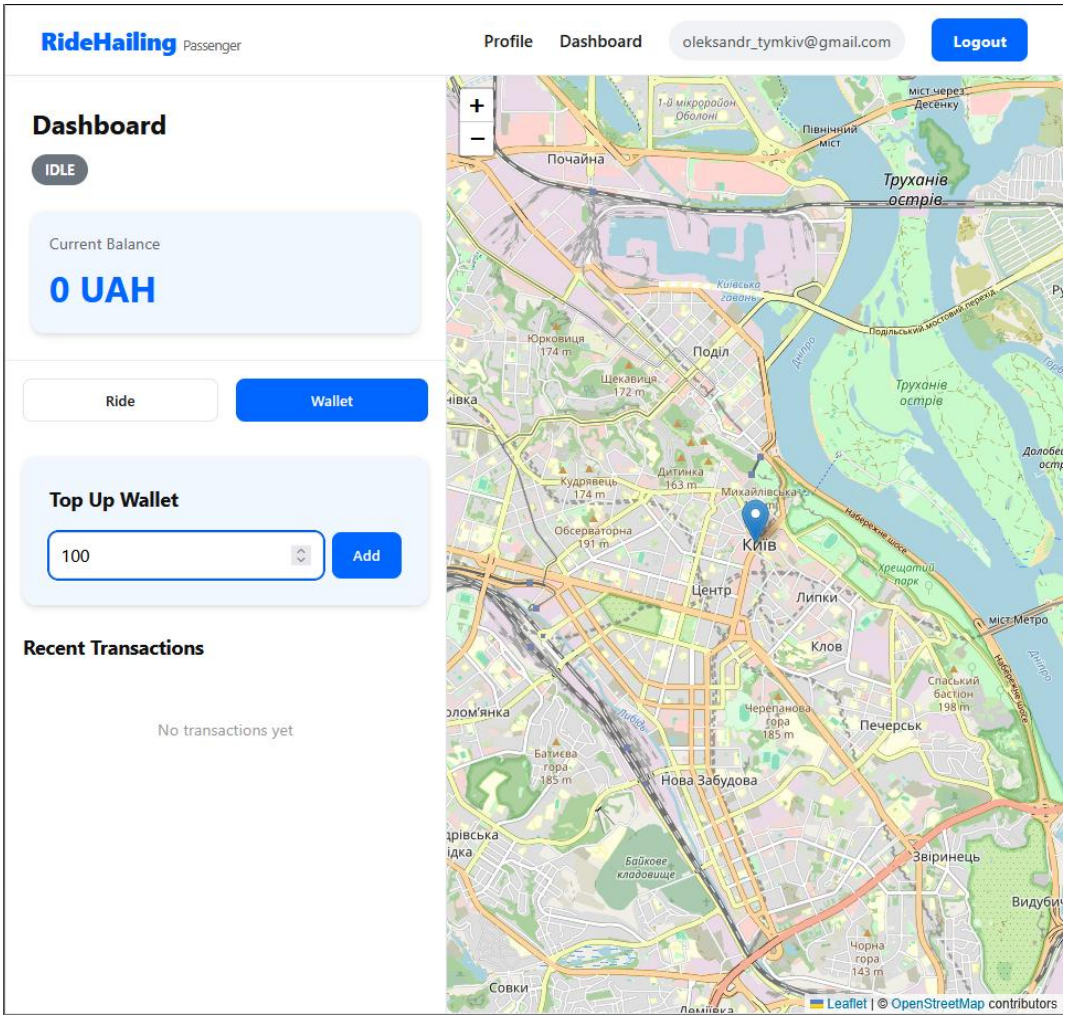


Рисунок 4.7 – Поповнення гаманця

Баланс було поповнено на 100 UAH і гроші миттєво зараховуються на віртуальний гаманець через мікросервіс користувачів.

### 4.2.2 Формування замовлення та просторовий метчинг

Після поповнення балансу, з’явилися гроші на гаманці, що дозволяє створити замовлення. Пасажир встановлює на мапі маркери місця посадки та точки призначення, у цей час фронтенд робить запит до навігаційного API, щоб отримати геометрію маршруту та точну дистанцію. На рис. 4.8 продемонстровано, що бекенд розраховує попередню вартість поїздки – 244 UAH, а дистанцію – 11.7 км.

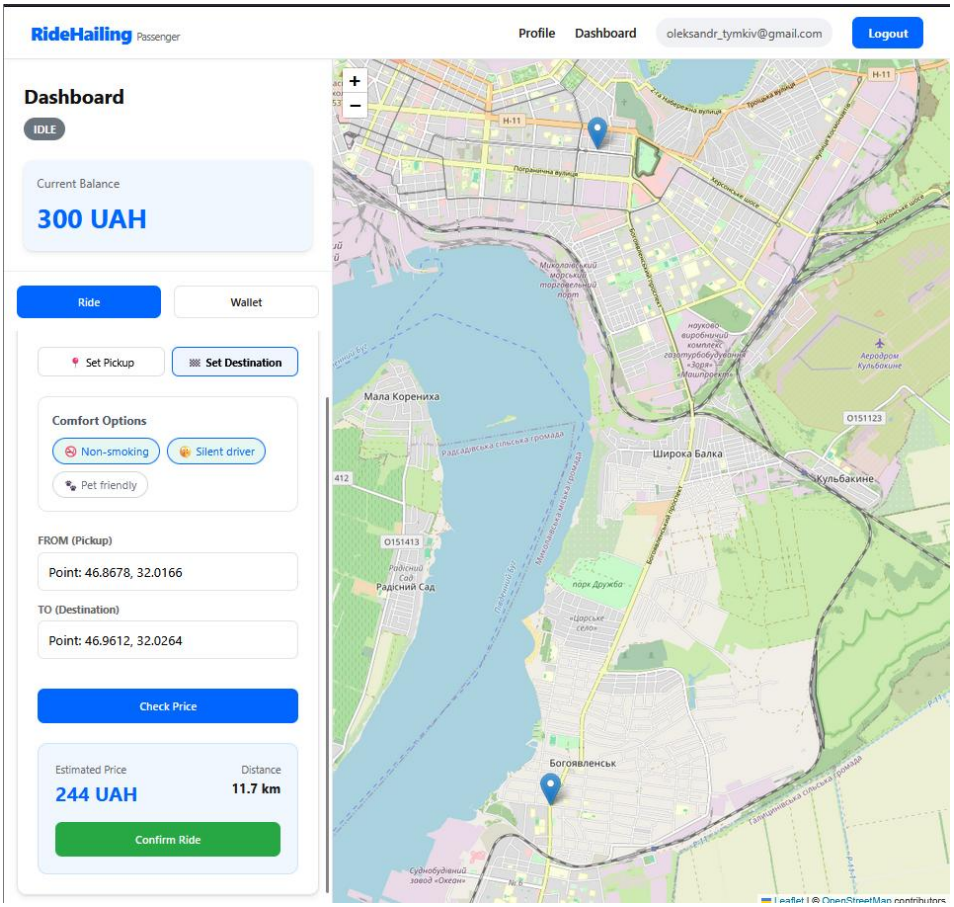


Рисунок 4.8 – Сформовані дані про майбутню поїздку

У додачу, пасажир обирає додаткові критерії комфорту: «Салон для некурців» та «Мовчазний водій», ці параметри будуть критичними під час пошуку водія, оскільки під час пошуку відсортовано водіїв за цими критеріями.

Також, у пасажира є можливість обрати місце призначення за допомогою ШІ-помічника, передаючи йому запит. Це додаткова функція, пасажир може використати її або ні, а результатом роботи список найближчих місць, що було синтезовано з відповідного запиту (рис. 4.9).

Щоб замовлення взагалі могло бути виконаним, водій повинен вийти в онлайн, натискаючи на кнопку статусу. Це переводить водія в режим “AVAILABLE” (рис. 4.10). У цей момент, браузер починає зчитувати GPS-координати і через WebSocket-з'єднання відправляє їх на бекенд (рис. 4.11), а мікросервіс геолокації записує ці координати в in-memory базу Redis GEO. Водій стає видимим для системи.

Пасажир натискаючи на “Confirm Ride”, змінює статус поїздки на “REQUESTED” (рис. 4.12). HTTP-запит летить на RideService. Що відбувається безпосередньо в коді бекенду: система робить радіальний пошук у Redis, і якщо знаходить нашого водія, перевіряє його теги на відповідність вимогам пасажира і додає його до пулу кандидатів.

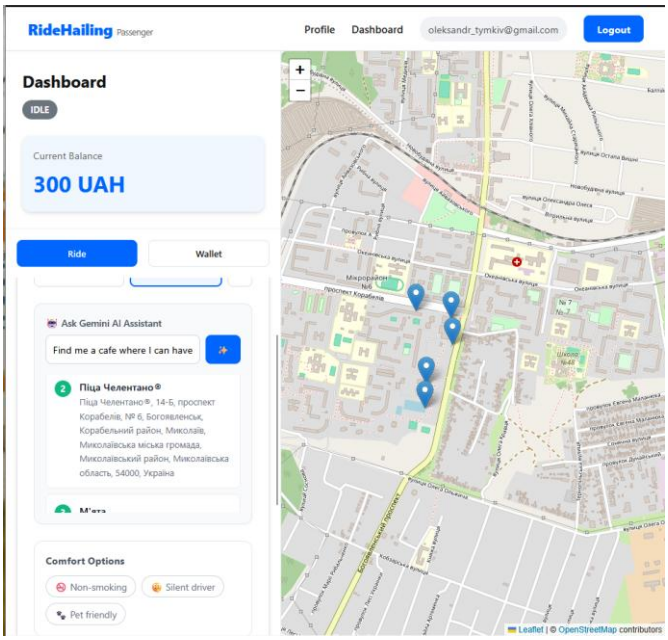


Рисунок 4.9 – Знайдені місця

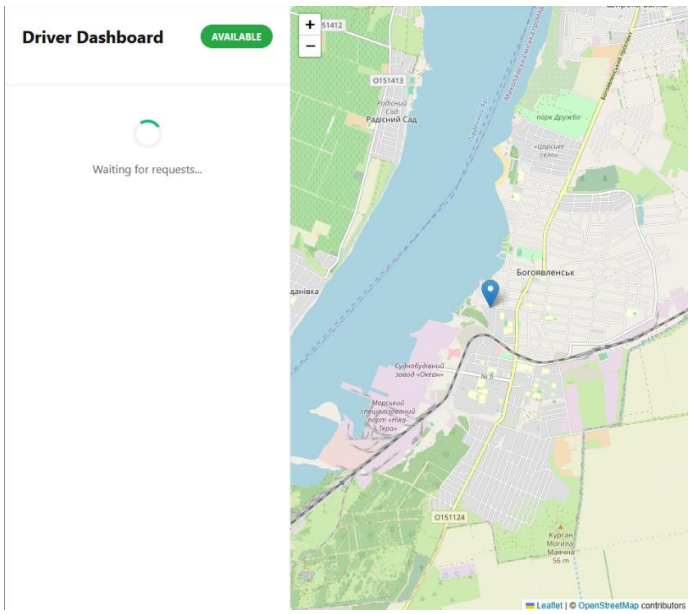


Рисунок 4.10 – Водій перейшов у стан “AVAILABLE”

```
чищено 2 "привидів" з радару
айдено 2 водіїв у радіусі 3.0 км для замовлення
 WS лог: Водій 18b9ebf5-476f-427a-8aae-f7076040891e надіслав координати: [46.86801769171899, 32.00995850743766]
 WS лог: Водій 18b9ebf5-476f-427a-8aae-f7076040891e надіслав координати: [46.86799523957509, 32.01022081407441]
 WS лог: Водій 18b9ebf5-476f-427a-8aae-f7076040891e надіслав координати: [46.86799939242912, 32.010230404365686]
 WS лог: Водій 18b9ebf5-476f-427a-8aae-f7076040891e надіслав координати: [46.86796929643037, 32.01025955241674]
```

Рисунок 4.11 – Отримання координат водія

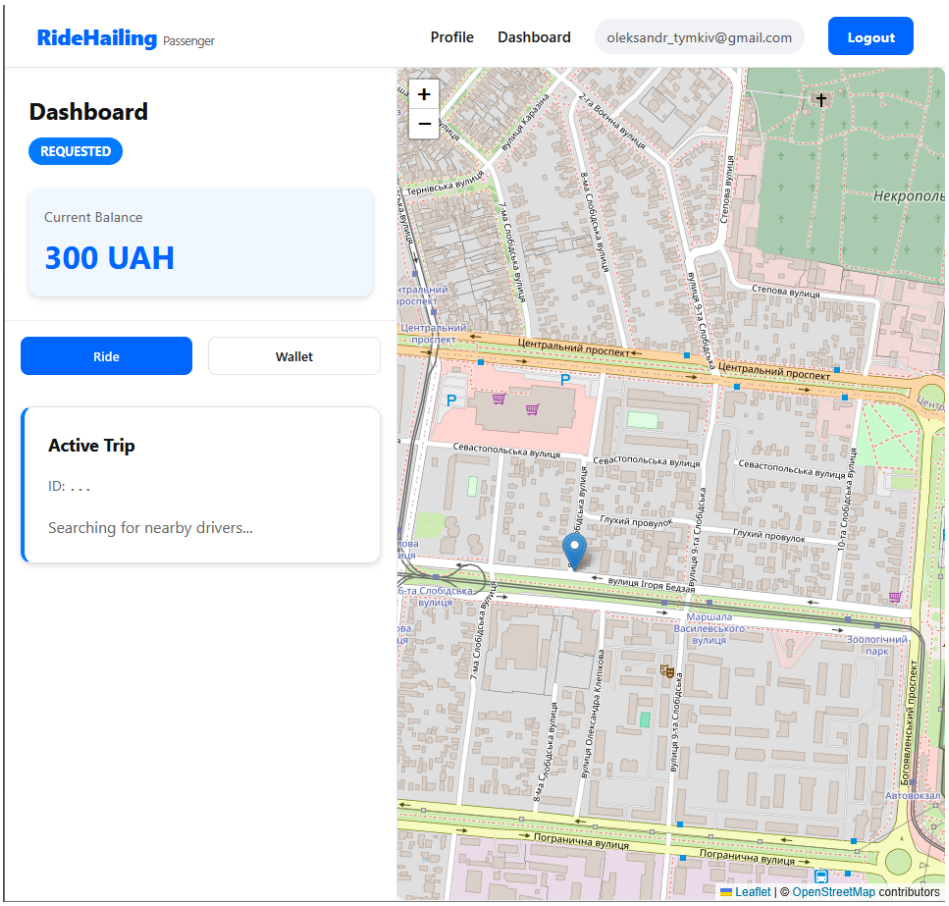


Рисунок 4.12 – Підтвердження поїздки та пошук найближчих водіїв

### 4.2.3 Асинхронна комунікація та імітація руху

Мікросервіс поїздок не чекає відповіді від водія напряму, оскільки тут використовується асинхронна взаємодія, тому він створює об'єкт події і публікує його в RabbitMQ, а вже безпосередньо після цього, GeoService перехоплює повідомлення і миттєво доставляє його через STOMP-протокол (WebSocket) прямо в браузер водія. На екрані з'являється повідомлення “New Ride Request” (рис. 4.13). Пасажир у цей час очікує на прийняття поїздки водієм.

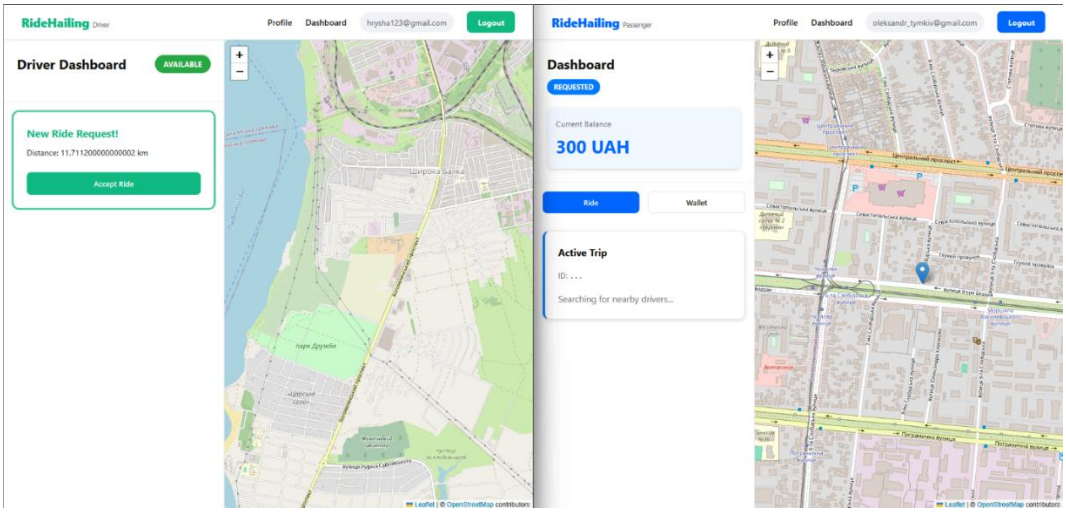


Рисунок 4.13 – Повідомлення водію про запит на поїздку

Водій приймає виклик і статус миттєво змінюється на “BUSY”. Тут починається найскладніший етап візуалізації, оскільки для перевірки було написано алгоритм імітації, а саме: реальний GPS-трекінг замінюється на рух по отриманому масиву координат реальної дороги.

Маркер водія починає дискретно пересуватися мапою, кожна нова координата транслюється на сервер і одразу пересилається пасажиру (рис. 4.14). Це забезпечило те, що в обох користувачів немає необхідності перезавантажувати сторінку, тому що інтерфейси оновлюються синхронно в реальному часі.

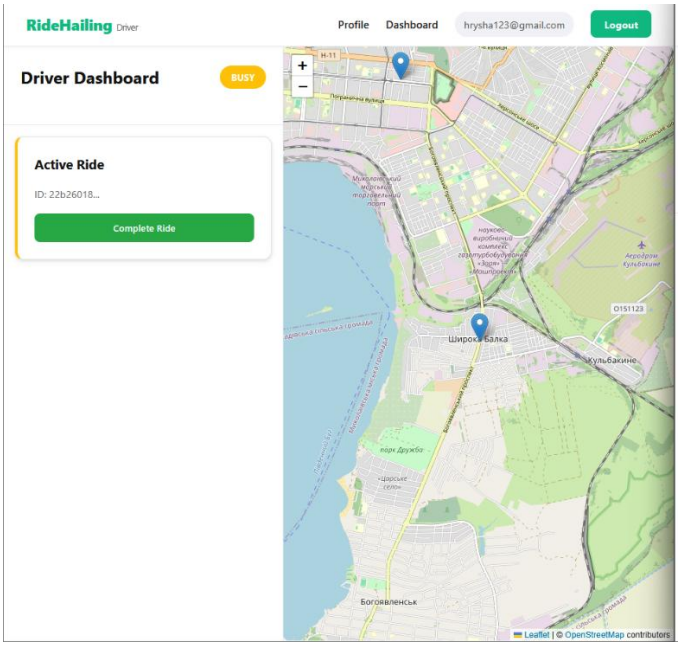


Рисунок 4.14 – Процес пересування

### 4.2.4 Транзакційна цілісність та білінг

Автівка досягла точки призначення. Водій натискає “Complete Ride”, що ініціює фінальну подію, яка потрапляє у чергу billing.queue.

Listener у мікросервісі користувачів отримує повідомлення, де передано гаманці водія та пасажира за їхніми KeycloakID, і після цього виконує транзакцію. Операція відповідає вимогам ACID, тобто списання та нарахування гарантовано відбуваються в межах однієї транзакції бази даних.

Результат операції миттєво відображається на клієнті (рис. 4.15). Пасажир бачить, що з балансу було коректно списано 244 UAH, а залишок 55.55 UAH. Інтерфейс водія повертається у стан очікування нових замовлень, тобто “OFFLINE”, а у блоці статистики оновлену відповідно статистику, а саме успішно виконанні замовлення та зароблені кошти. Тестування завершено успішно. Система відпрацювала без жодних збоїв.

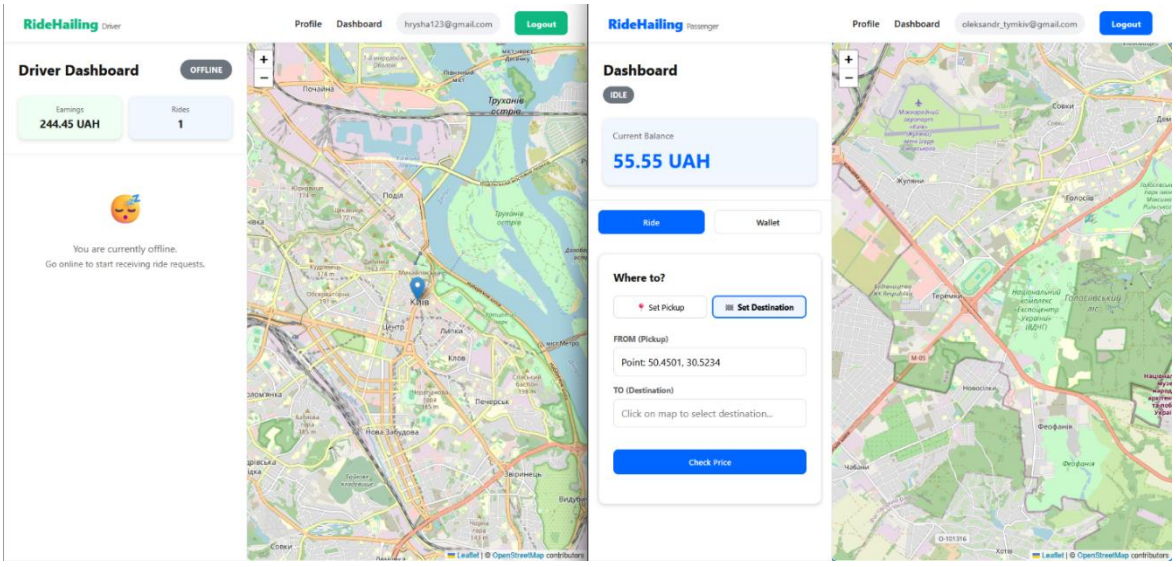


Рисунок 4.15 – Завершення поїздки та оновлення балансів водія та пасажира

Розроблена система, як було продемонстровано на рис. 4.15, повністю справляється з покладеною бізнес-логікою. Життєвий цикл поїздки відпрацьовує без жодних розсинхронізацій між незалежними клієнтами, демонструючи подолання побудованого маршруту. Завдяки комбінації брокера повідомлень

RabbitMQ та протоколу WebSocket, було досягнуто режиму реального часу, унаслідок цього статуси, повідомлення та GPS-координати оновлюються миттєво. У той же час, фінансові операції, за які відповідає модуль білінгу, відпрацювали адекватно, доводячи свою надійність з суворим дотриманням принципів ACID.

**Висновки до розділу 4**

У першій частині четвертого розділу було формалізовано керівництво користувача. Чітке розділення робочих просторів на водійську та пасажирську частини дозволило створити інтуїтивно зрозумілий досвід без перевантаження інтерфейсу. Окрім базових алгоритмів пошуку авто та обробки замовлень, описано керування віртуальним гаманцем, попередньої верифікації профілів, гнучкого налаштування критеріїв комфорту та використання ШІ-помічника, який виділяє теґ з відповідного запиту про бажання місця призначення, тобто кафе чи кінотеатр.

Друга частина розділу стала головним краш-тестом для закладеної мікросервісної архітектури. End-to-End тестування життєвого циклу поїздки підтвердило надійність розподіленої взаємодії. Багаторівневий просторовий метчинг через Redis, асинхронна маршрутизація подій брокером RabbitMQ та STOMP-трансляція координат злилися в єдиний, безперебійний процес. Система довела свою здатність працювати у справжньому режимі реального часу: екрани клієнтів оновлювалися синхронно, а фінансові взаєморозрахунки виконувалися із суворим дотриманням вимог ACID.

# ВИСНОВКИ

У ході виконання кваліфікаційної роботи було спроєктовано та реалізовано мікросервісну систему онлайн-замовлення поїздок. Виконаний обсяг завдань підтверджує повне досягнення поставленої мети. Замість традиційного монолітного рішення було розроблено та впроваджено стійку розподілену архітектуру, яка безпосередньо адаптована до специфіки обробки високочастотних потоків даних у реальному часі.

Головним архітектурним рішення проєкту стала декомпозиція бізнес-логіки. Завдяки впровадженню методології DDD, систему вдалося успішно розділити на повністю ізольовані предметні домени: управління профілями, життєвий цикл поїздок, просторовий метчинг та фінансовий білінг. База даних перестала бути єдиною точкою відмови, тому що було запроваджено концепцію Polyglot Persistence, що дозволило технологічно адаптувати інфраструктуру під специфіку конкретних навантажень. Тобто, класична реляційна СУБД PostgreSQL гарантує цілісність фінансових транзакцій, тоді як надшвидке in-memory сховище Redis забезпечує ефективну буферизацію високочастотних GPS-координат екіпажів.

Окремої уваги заслуговує розроблений механізм трансляції телеметрії. Використання класичного методу HTTP-polling під час безперервного оновлення GPS-координат призвело б до перевантаження та відмови сервера, тому цю архітектурну проблему вирішено шляхом впровадження повнодуплексних каналів зв'язку через WebSockets із застосуванням STOMP у комбінації з проміжним кешуванням потокових даних в in-memory базі Redis. Це оптимізувало продуктивність системи. Що у висновку суттєво знизилося навантаження на обчислювальні ресурси процесора, а платформа здобула здатність обробляти та ретранслювати просторові дані у справжньому режимі реального часу.

Найскладнішим викликом у проєктуванні розподілених систем залишається управління фінансами, оскільки втрата даних під час транзакцій є неприпустимою навіть за умови виникнення мережевих збоїв. Імплементація архітектурного

патерну “Saga” у поєднанні з брокером повідомлень RabbitMQ надійно розв'язала цю проблему. Життєвий цикл поїздки переведено в асинхронний формат: мікросервіси взаємодіють через публікацію подій, завдяки чому після завершення маршруту модуль білінгу гарантовано проводить взаєморозрахунки з суворим дотриманням принципів ACID. Водночас проблему наскрізної безпеки під час міжсервісної комунікації було закрито за допомогою сервера Keycloak та архітектурного патерну Token Relay.

Працездатність закладених ідей доведено результатами наскрізного тестування. Було перевірено повний життєвого цикл поїздки, що підтвердило, що система успішно справляється зі складним просторовим метчингом, коректно обробляючи кастомні фільтри (наприклад, «салон для некурців» або «мовчазний водій»). У той же час, верифіковано роботу модуля інтелектуального пошуку: платформа автоматично аналізує довільні текстові запити пасажирів природною мовою, знаходить релевантні об'єкти в радіусі поїздки та формує динамічний список пропозицій. Клієнтські інтерфейси при цьому оновлюються синхронно. Від моменту ініціалізації замовлення до фінального списання коштів із віртуального гаманця система відпрацювала без жодних розсинхронізацій станів чи втрати даних.

Тому, спираючись всі вище описані фактори, поставлені задачі для роботи було виконано. Побудований мікросервісний застосунок чітко охарактеризовано як систему, де чітко імплементовані відповідні патерни та архітектурні рішення для забезпечення адекватної логіки онлайн-замовлень поїздок.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Jittrapirom P., Caiati V., Feneri A. M. Mobility as a Service: A Critical Review of Definitions, Assessments of Schemes, and Key Challenges. Urban Planning. 2017. URL: https://www.researchgate.net/publication/318208676_Mobility_as_a_Service_A_Critical_Review_of_Definitions_Assessments_of_Schemes_and_Key_Challenges

2. Zha L., Yin Y., Yang H. Economic analysis of ride-sourcing markets. Transportation Research Part C: Emerging Technologies. 2016. URL: https://www.researchgate.net/publication/306419372_Economic_analysis_of_ride-sourcing_markets

3. Wang Y., Zheng Y., Xue Y. Travel Time Estimation of a Path using Sparse Trajectories. 2014. URL: https://dl.acm.org/doi/10.1145/2623330.2623656

4. Spring Framework Documentation. WebSockets/Spring. URL: https://docs.spring.io/spring-framework/reference/web/websocket.html (дата звернення: 06.02.2026)

5. Richardson C. Microservices Patterns: With examples in Java. Shelter Island: Manning Publications, 2018. URL: https://search.worldcat.org/title/Microservices-patterns-:-with-examples-in-Java/oclc/1091373873

6. Liu Y., Zhang C., & Gao S. Intent-Based Routing in Urban Mobility Systems Using Large Language Models. Transportation Research Part C: Emerging Technologies. 2025.

7. Yao S., Yu D., & Guha A. Retrieval-Augmented Generation for Geospatial Domain: Overcoming Spatial Hallucinations in LLMs. ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. 2024. P. 312–322.

8. OpenStreetMap Geocoding Service (Nominatim). URL: https://wiki.openstreetmap.org/wiki/Nominatim (дата звернення: 10.02.2026)

9. Uber Revenue and Usage Statistics. Business of Apps. URL:

https://www.businessofapps.com/data/uber-statistics/ (дата звернення: 14.02.2026)

10. Invest In Bolt Stock. EquityZen. URL: https://equityzen.com/company/mtakso/ (дата звернення: 17.02.2026)

11. Uklon Corporate. Uklon. URL: https://uklon.com.ua/news/135-miljoniv-poyizdok-140-tysyach-povernutyh-rechej-ta-46-tysyach-gryven-chajovyh-vid-odnogo-rajdepa-uklon-podilyvsya-pidsumkamy-2025-roku/ (дата звернення: 19.02.2026)

12. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston : Addison-Wesley Professional, 2003. 560 p.

13. Taibi D., Lenarduzzi V., Pahl C. Architectural Patterns for Microservices: A Systematic Mapping Study. 8th International Conference on Cloud Computing and Services Science. 2018. P. 221–232.

14. Garcia-Galan J., Pasos H., Ruiz-Cortes A. Data Consistency in Microservices: A Systematic Mapping Study. IEEE Access. 2021. Vol. 9.

15. Nygard M. T. Release It!: Design and Deploy Production-Ready Software. Raleigh : Pragmatic Bookshelf, 2018. 376 p.

16. Sigelman B., Barroso L. A., Burrows M. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report. 2010.

17. Pimentel V., Nickerson B. G. Communicating and displaying real-time data with WebSocket. IEEE Internet Computing. 2012. Vol. 16, № 4. P. 45–53.

18. OpenRouteService API Documentation. Heidelberg Institute for Geoinformation Technology. URL: https://openrouteservice.org/dev/api-docs (дата звернення: 06.03.2026)

19. Sinnott R. W. Virtues of the Haversine. Sky and Telescope. 1984. Vol. 68, № 2. P. 158.

20. N. Chopde, M. Nichat. Landmark Based Shortest Path Detection by Using A* and Haversine Formula. 2013. Vol. 1, Issue 2. P. 298-302.

21. Longley P. A., Goodchild M. F., Maguire D. J., Rhind D. W. Geographic Information Systems and Science. 4th Edition. 2015. 496 p.

22. Walls C. Spring in Action, Sixth Edition. Shelter Island: Manning

Publications, 2022. URL: https://linux.ime.usp.br/~henrick/[ebook]%20Manning%20-%20Spring%20in%20Action.pdf

23. Server Administration Guide. Keycloak Documentation. URL: https://www.keycloak.org/docs/latest/server_admin/ (дата звернення: 13.03.2026)

24. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). Internet Engineering Task Force (IETF). 2015.

25. Carlson J. Redis in Action. Shelter Island : Manning Publications, 2013. 328 p.

26. RabbitMQ: AMQP 0-9-1 Model Explained. RabbitMQ Documentation. URL: https://www.rabbitmq.com/tutorials/amqp-concept.html (дата звернення: 17.03.2026)

27. STOMP Protocol Specification, Version 1.2. STOMP. URL: https://stomp.github.io/stomp-specification-1.2.html (дата звернення: 18.03.2026)

28. Merkel D. Docker: lightweight linux containers for consistent development and deployment. Linux journal. 2014. URL: https://www.researchgate.net/publication/261960832_Docker_lightweight_Linux_containers_for_consistent_development_and_deployment

29. Turnbull P. The Logstash Book. London : Turnbull Press, 2013. 250 p.

30. Testcontainers: Integration Testing with Docker. Testcontainers Official Documentation. URL: https://java.testcontainers.org/ (дата звернення: 21.03.2026)

31. Google AI Studio Gemini API Specs. Google Developer Documentation. 2026. URL: https://ai.google.dev/gemini-api/docs (дата звернення: 23.03.2026)

# ДОДАТОК А

## Лістинг коду управління поїздками

### А.1 Метод створення поїздки

```
@Override
public RideResponseDTO createRide(UUID passengerId, CreateRideDTO createRideDTO) {
    BigDecimal balance = userServiceClient.getPassengerBalance();
    if(balance.compareTo(BigDecimal.ZERO) < 0)
        throw new IllegalArgumentException("You have an outstanding debt: " +
balance + " UAH. Please top up your account.");
    RideRouteInfoDTO rideRouteInfoDTO = geoServiceClient.getRouteInfo(
            createRideDTO.startLng(), createRideDTO.startLat(),
            createRideDTO.endLng(), createRideDTO.endLat()
    );
    BigDecimal expectedPrice = fareCalculatorService.calculateExpectedPrice(
            rideRouteInfoDTO.distanceKm(),
            rideRouteInfoDTO.durationMin()
    );
    Ride ride = new Ride();
    ride.setPassengerId(passengerId);
    ride.setStartLat(createRideDTO.startLat());
    ride.setStartLng(createRideDTO.startLng());
    ride.setEndLat(createRideDTO.endLat());
    ride.setEndLng(createRideDTO.endLng());
    ride.setExpectedDistance(rideRouteInfoDTO.distanceKm());
    ride.setExpectedPrice(expectedPrice);
    ride.setPrice(expectedPrice);
    ride.setStatus(RideStatus.CREATED);
    ride.setCreatedAt(LocalDateTime.now());
    rideRepository.save(ride);
    List<String> nearbyDrivers =
geoServiceClient.getNearbyDrivers(ride.getStartLat(), ride.getStartLng(), 3.0);
    if (!nearbyDrivers.isEmpty()) {
        RideBroadcastEvent event = new RideBroadcastEvent(
                ride.getId(),
                ride.getPrice(),
                ride.getStartLat(),
                ride.getStartLng(),
                ride.getEndLat(),
                ride.getEndLng(),
                rideRouteInfoDTO.distanceKm(),
                nearbyDrivers,
                rideRouteInfoDTO.geometry()
        );
        rabbitTemplate.convertAndSend(RabbitMQConfig.RIDE_EXCHANGE,
RabbitMQConfig.RIDE_BROADCAST_ROUTING_KEY, event);
    }
    return RideResponseDTO.builder()
            .id(ride.getId())
            .passengerId(passengerId)
            .driverId(ride.getDriverId())
            .expectedDistance(ride.getExpectedDistance())
            .expectedPrice(ride.getExpectedPrice())
            .expectedDuration(rideRouteInfoDTO.durationMin())
            .status(ride.getStatus())
            .createdAt(ride.getCreatedAt())
```

```
            .build();
}
```

## А.2 Метод отримання прогнозованого маршруту

```
@Override
public RideRouteInfoDTO calculateRoute(double startLng, double startLat, double
endLng, double endLat) {
    String start = startLng + "," + startLat;
    String end = endLng + "," + endLat;
    try{
        RouteResponseDTO routeResponseDTO = orsClient.getRoute(apiKey, start,
end);
        if(routeResponseDTO.features() == null ||
routeResponseDTO.features().isEmpty())
            throw new RuntimeException("Routes not found");
        RouteResponseDTO.Summary summary =
routeResponseDTO.features().getFirst().properties().summary();
        RouteResponseDTO.Geometry geometry =
routeResponseDTO.features().getFirst().geometry();
        List<List<Double>> coordinates = geometry.coordinates();
        List<String> formattedCoords = coordinates.stream().map(point ->
point.get(0) + "," + point.get(1)).toList();
        return new RideRouteInfoDTO(
                summary.distance() / 1000.,
                summary.duration() / 60.,
                formattedCoords
        );
    } catch (Exception e) {
        log.error("Error calculating route : ", e);
        throw new RuntimeException(e);
    }
}
```

## А.3 Пошук найближчих водіїв

```
@Override
public List<String> findNearbyDrivers(double startLat, double startLng, double
radiusKm) {
    Point center = new Point(startLng, startLat);
    Distance radius = new Distance(radiusKm, Metrics.KILOMETERS);
    Circle circle = new Circle(center, radius);
    RedisGeoCommands.GeoRadiusCommandArgs args =
RedisGeoCommands.GeoRadiusCommandArgs
            .newGeoRadiusArgs()
            .sortAscending();
    GeoResults<RedisGeoCommands.GeoLocation<String>> geoResults =
            stringRedisTemplate.opsForGeo().radius(DRIVERS_GEO_KEY, circle, args);
    if(geoResults == null || geoResults.getContent().isEmpty()) {
        log.info("There are no available drivers within a radius of {} km from
point [{},{}]", radiusKm, startLat, startLng);
        return List.of();
    }
    List<String> driverIds = geoResults.getContent().stream()
            .map(geoLocationGeoResult ->
geoLocationGeoResult.getContent().getName())
            .toList();
    List<String> onlineKeys = driverIds.stream()
            .map(id -> DRIVER_ONLINE_PREFIX + id)
```

```
            .toList();
    List<String> onlineStatuses =
stringRedisTemplate.opsForValue().multiGet(onlineKeys);
    List<String> activeDrivers = new ArrayList<>();
    List<String> deadDrivers = new ArrayList<>();
    for(int i = 0; i < driverIds.size(); i++){
        String driverId = driverIds.get(i);
        if(onlineStatuses != null && onlineStatuses.get(i) != null)
            activeDrivers.add(driverId);
        else
            deadDrivers.add(driverId);
    }
    if(!deadDrivers.isEmpty()){
        stringRedisTemplate.opsForGeo().remove(DRIVERS_GEO_KEY,
deadDrivers.toArray(new String[0]));
        log.info("Cleared {} \"ghosts\" from radar", deadDrivers.size());
    }
    log.info("Found {} drivers within {} km radius for your order",
driverIds.size(), radiusKm);
    return activeDrivers;
}
```

## А.4 Метод прийняття поїздки водієм

```
@Override
@Transactional
public void acceptRide(UUID rideId, UUID driverId) {
    Ride ride = rideRepository.findById(rideId)
            .orElseThrow(() -> new IllegalArgumentException("Ride not found"));
    if(ride.getStatus() != RideStatus.CREATED)
        throw new IllegalArgumentException("Ride is not access any more");
    if (ride.getPassengerId().equals(driverId))
        throw new IllegalArgumentException("You can't accept your own order!");
    ride.setDriverId(driverId);
ride.setAcceptedAt(LocalDateTime.now());
    ride.setStatus(RideStatus.ACCEPTED);
    rideRepository.save(ride);
    log.info("Driver {} successfully accepted ride {}", driverId, rideId);
    RideStatusEvent event = new RideStatusEvent(
            ride.getId(),
            ride.getPassengerId(),
            driverId,
            ride.getStatus().name(),
            "Driver is found and leads to you"
    );
    rabbitTemplate.convertAndSend("geo.exchange", "ride-updates", event);
}
```

## А.5 Метод-listener для отримання даних поїздки

```
@RabbitListener(queues = RabbitMQConfig.ROUTE_COMPLETED_QUEUE)
@Transactional
public void handleRouteCompleted(RideRouteCompletedEvent event){
    log.info("Received route completed event: {}", event.coordinates().size());
    UUID rideId = event.rideId();
    Ride ride = rideRepository.findById(rideId).orElseThrow(
            () -> new IllegalArgumentException("Invalid ride id: " +
event.rideId())
    );
```

```
    try {
        String routeJson = objectMapper.writeValueAsString(event.coordinates());
        ride.setRouteHistory(routeJson);
    } catch (JsonProcessingException e) {
        log.error("Error parsing route json: {}", e.getMessage());
        ride.setRouteHistory("[]");
    }
    BigDecimal finalPrice = fareCalculatorServiceImpl.calculateFinalPrice(
            ride.getExpectedPrice(),
            ride.getExpectedDistance(),
            event.actualDistance()
    );
    ride.setActualDistance(event.actualDistance());
    ride.setFinalPrice(finalPrice);
    ride.setPrice(finalPrice);
    rideRepository.save(ride);
    RideCompletedEvent eventCompleted = RideCompletedEvent.builder()
            .rideId(rideId)
            .driverId(ride.getDriverId())
            .passengerId(ride.getPassengerId())
            .price(finalPrice)
            .completedAt(LocalDateTime.now())
            .build();
    rabbitTemplate.convertAndSend("ride.exchange","ride-completed",
eventCompleted);
    log.info("Event sent: Ride {} completed", rideId);
}
```

## А.6 Метод-listener для обробки статусу поїздки

```
@RabbitListener(queues = RabbitMQConfig.RIDE_UPDATES_QUEUE)
public void handleRideStatus(RideStatusEvent event) {
    if(event.rideStatus().equals("ACCEPTED")){
        sendNotificationToPassenger(event);
        if (event.driverId() != null) {
 driverLocationService.removeDriverFromRadar(event.driverId().toString());
        }
    }
    else if(event.rideStatus().equals("COMPLETED")){
        sendNotificationToPassenger(event);
        String redisKey = LocationServiceImpl.RIDE_COORDINATE_PREFIX +
event.rideId();
        List<String> rawCoordinates = redisTemplate.opsForList().range(redisKey,
0, -1);
        double actualDistance =
DistanceCalculatorUtil.calculateTotalDistance(rawCoordinates);
        redisTemplate.delete(redisKey);
        RideRouteCompletedEvent rideRouteCompletedEvent = new
RideRouteCompletedEvent(
                event.rideId(),
                rawCoordinates != null ? rawCoordinates : List.of(),
                actualDistance
        );
        rabbitTemplate.convertAndSend("geo.exchange", "route-completed",
rideRouteCompletedEvent);
    }
```

## А.7 Методи для обрахування реальної відстані поїздки (Формула Гаверсина)

```
private static final int EARTH_RADIUS = 6371;
public static double calculateTotalDistance(List<String> rawCoordinates){
    if(rawCoordinates == null || rawCoordinates.size() < 2)
        return 0.;
    double totalDistance = 0.;
    for(int i = 0; i < rawCoordinates.size() - 1; i++){
        String[] p1 = rawCoordinates.get(i).split(",");
        String[] p2 = rawCoordinates.get(i + 1).split(",");
        try {
            double lat1 = Double.parseDouble(p1[0].trim());
            double lon1 = Double.parseDouble(p1[1].trim());
            double lat2 = Double.parseDouble(p2[0].trim());
            double lon2 = Double.parseDouble(p2[1].trim());
            totalDistance += calculateHaversine(lat1, lon1, lat2, lon2);
        } catch (NumberFormatException | ArrayIndexOutOfBoundsException e){
            continue;
        }
    }
    return totalDistance;
}
private static double calculateHaversine(double lat1, double lon1, double lat2,
double lon2) {
    double dLat = Math.toRadians(lat2 - lat1);
    double dLon = Math.toRadians(lon2 - lon1);
    double originLat = Math.toRadians(lat1);
    double destinationLat = Math.toRadians(lat2);
    double a = Math.pow(Math.sin(dLat / 2), 2) +
            Math.pow(Math.sin(dLon / 2), 2) *
            Math.cos(originLat) * Math.cos(destinationLat);
    double c = 2 * Math.asin(Math.sqrt(a));
    return EARTH_RADIUS * c;
}
```

## А.8 Метод завершення поїздки

```
@Override
@Transactional
public void completeRide(UUID rideId) {
    Ride ride = rideRepository.findById(rideId).orElseThrow();
    ride.setStatus(RideStatus.COMPLETED);
    ride.setCompletedAt(LocalDateTime.now());
    rideRepository.save(ride);
    RideStatusEvent geoEvent = new RideStatusEvent(
            ride.getId(),
            ride.getPassengerId(),
            ride.getDriverId(),
            ride.getStatus().name(),
            "Ride is completed"
    );
    rabbitTemplate.convertAndSend("geo.exchange", "ride-updates", geoEvent);
}
```

# ДОДАТОК Б

## Лістинг коду управління користувачами

### Б.1 Метод створення користувача

```
@Override
@Transactional
public UserResponseDTO save(UserRegistrationDTO dto) {
    if(userRepository.existsByEmail(dto.getEmail())){
        log.error("User cannot be registered with the email address: {}",
dto.getEmail());
        throw new UserAlreadyExistsException("User with email "+dto.getEmail()+"
already exists");
    }
    String keycloakId = keycloakService.createUserInKeycloak(dto);
    User newUser = mapper.toEntity(dto, UUID.fromString(keycloakId));
    Wallet wallet = new Wallet();
    wallet.setBalance(BigDecimal.ZERO);
    wallet.setCurrency("UAH");
    newUser.setWallet(wallet);
    newUser = userRepository.saveAndFlush(newUser);
    log.info("Saved user: {}", newUser);
    if(dto instanceof DriverRegistrationDTO driverRegistrationDTO){
        driverService.save(newUser, driverRegistrationDTO);
        passengerService.save(newUser, new PassengerRegistrationDTO());
        log.info("Saved driver: {}", newUser);
        keycloakService.assignRolesInKeycloak(keycloakId, List.of("DRIVER",
"PASSENGER"));
        log.info("Assigned role for driver: {}", newUser);
    }
    if(dto instanceof PassengerRegistrationDTO passengerRegistrationDTO){
        passengerService.save(newUser, passengerRegistrationDTO);
        log.info("Saved passenger: {}", newUser);
        keycloakService.assignRolesInKeycloak(keycloakId, List.of("PASSENGER"));
        log.info("Assigned role for driver: {}", newUser);
    }
    return mapper.toResponseDto(newUser);
}
```

### Б.2 DTO для створення користувача

```
@JsonTypeInfo(
        use = JsonTypeInfo.Id.NAME,
        include = JsonTypeInfo.As.PROPERTY,
        property = "role",
        visible = true
)
@JsonSubTypes({
        @JsonSubTypes.Type(value = PassengerRegistrationDTO.class, name =
"PASSENGER"),
        @JsonSubTypes.Type(value = DriverRegistrationDTO.class, name = "DRIVER")
})
@Data
public abstract class UserRegistrationDTO {
    private String email;
    @ToString.Exclude
    private String password;
```

```
    private String firstName;
    private String lastName;
    private String phoneNumber;
    private String role;
}
```

## Б.3 Оновлення ролей у Keycloak

```
@Override
public void assignRolesInKeycloak(String userId, List<String> roleNames) {
    RolesResource rolesResource =
keycloak.realm(keycloakProperties.getRealm()).roles();
    List<RoleRepresentation> roles = new ArrayList<>();
    for(String roleName : roleNames){
        try{
            roles.add(rolesResource.get(roleName).toRepresentation());
        } catch(Exception e){
            log.error("Role not found in Keycloak: {}", roleName);
            throw new RuntimeException("System role not configured: " + roleName);
        }
    }
    if(!roles.isEmpty())
        keycloak.realm(keycloakProperties.getRealm())
                .users()
                .get(userId)
                .roles()
                .realmLevel()
                .add(roles);
    log.info("Role of user {} added in Keycloak ", userId);
}
```