МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія (кафедра) комп'ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ WEB API ДЛЯ ПЛАТФОРМИ ПАСАЖИРОПЕРЕВЕЗЕНЬ

Виконав: студент групи 1П-21
Спеціальності
121 Інженерія програмного забезпечення
Олексій ГУРЕЄВ
Керівник:
Станіслав МАРЧЕНКО

Черкаси 2025

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Кафедра комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри КІ та ІТ

____________ Владислав ХОТУНОВ
(підпис)

«_____»______________2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Гуреєву Олексію Сергійовичу
(прізвище, ім'я, по батькові студента)

1. Тема випускної роботи Проєктування та реалізація Web API для платформи пасажироперевезень

Керівник роботи Марченко Станіслав Віталійович, спеціаліст І категорії
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від «07» жовтня 2024 року № 68у.

2. Строк подання студентом випускної роботи 03.06.2025

3. Вихідні дані до випускної роботи мова програмування C#, фреймворк для створення API ASP.NET Core, фреймворк для роботи з базами даних EntityFramework Core, об'єктно-реляційна система керування базами даних (СКБД) PostgreSQL, універсальна мова моделювання (UML), платформа переглядання та тестування API Postman.

4. Зміст випускної роботи (перелік питань, які потрібно розробити) огляд предметної області (базові поняття в контексті пасажироперевезень, сучасний стан ринку додатків для пасажироперевезень, особливості побудови додатків для пасажирських перевезень, постановка задачі на розробку), проєктування та реалізація програмного забезпечення (аналіз вимог до Web API, високорівнева архітектура програмного рішення, детальне проєктування та подробиці реалізації програмного забезпечення), тестування програмного продукту (вибір методів та інструментів тестування, побудова тестового плану, проведення та результати тестування розробленого програмного рішення)

5. Дата видачі завдання 15.09.2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання з підписами керівника і студента
1	Вступ	14.10.2024	
2	Розділ 1. Огляд предметної області	09.12.2024	
3	Розділ 2. Проєктування та реалізація Web API	10.03.2025	
4	Розділ 3. Тестування програмного продукту	28.04.2025	
5	Висновки	12.05.2025	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2025	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2025	
8	Подання кваліфікаційної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	10.06.2025	

Студент ______________________________ Олексій ГУРЕЄВ
(підпис)

Керівник роботи ______________________________ Станіслав МАРЧЕНКО
(підпис)

АНОТАЦІЯ

Кваліфікаційна робота присвячена аналізу та розробленню сервісу, пов'язаного з пасажирськими перевезеннями із використанням сучасних технологій. У межах роботи було здійснено створення програмного рішення – API для застосунку, який забезпечує організацію автобусних перевезень з урахуванням кореляції маршрутів на різні відстані з застосуванням принципів ride-sharing.

У роботі розглянуто сучасний ринок додатків пасажирських перевезень, частинами якого є як і класичні таксі, так і сервіси, подібні до Uber. Проведено порівняльний аналіз особливостей статистики використання сервісів сумісних поїздок. Також були розглянуті питання навігації, оптимального розподілу поїздок та технологій, що сприяють зменшенню навантаження на API.

Визначено технічні вимоги до API, розроблено архітектуру та реалізовано її програмний прототип. Описано інструменти, використані під час розробки, зокрема мови програмування, бібліотеки та засоби тестування. API підтримує динамічне планування маршрутів, враховує точки посадки та висадки пасажирів і забезпечує інтеграцію з клієнтськими застосунками.

Результати роботи можуть бути корисними для розвитку систем організації пасажирських перевезень і стати основою стартапу сумісних автобусних перевезень.

Ключові слова: API, АРХІТЕКТУРА, РОЗРОБЛЕННЯ, ПАСАЖИРОПЕРЕВЕЗЕННЯ.

ABSTRACT

The qualification work is devoted to the analysis and development of a service related to passenger transport using modern technologies. As part of the work, a software solution was created – an API for an application that ensures the organization of bus transportation, considering the correlation of routes over different distances using the principles of ride-sharing.

The paper considers the current market of passenger transportation applications, which includes both classic taxis and Uber-like services. A comparative analysis of the features of statistics on the use of shared travel services was conducted. The article also considers the issues of navigation, optimal distribution of trips, and technologies that help reduce the load on the API.

The technical requirements for the API were determined, the architecture was developed, and its software prototype was implemented. The tools used in the development were described, including programming languages, libraries, and testing tools. The API supports dynamic route planning, takes into account passenger boarding and disembarkation points, and provides integration with client applications.

The results of the work can be useful for the development of passenger transportation organization systems and can become the basis for a shared bus transportation startup.

Keywords: API, ARCHITECTURE, DEVELOPMENT, PASSENGER TRANSPORTATION

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ... 3
ВСТУП ... 4
РОЗДІЛ 1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ... 6
1.1 Базові поняття в контексті пасажироперевезень ... 6
1.2 Сучасний стан ринку додатків для пасажироперевезень ... 10
1.3 Особливості побудови додатків для пасажирських перевезень ... 14
1.4 Постановка задачі на розробку ... 23
Висновки до першого розділу ... 24
РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ WEB API ... 25
2.1 Аналіз вимог до Web API ... 25
2.2 Високорівнева архітектура програмного рішення ... 33
2.3 Детальне проєктування та подробиці реалізації програмного забезпечення ... 38
Висновки до другого розділу ... 48
РОЗДІЛ 3 ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ ... 51
3.1 Вибір методів та інструментів тестування ... 51
3.2 Побудова тестового плану ... 52
3.3 Проведення та результати тестування розробленого програмного рішення ... 57
Висновки до третього розділу ... 59
ВИСНОВКИ ... 60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 61
ДОДАТКИ
Додаток А – Базовий код реалізації CRUD-операцій
Додаток Б – Перетворення моделі
Додаток В – Посилання на репозиторій

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

API	Application Programming Interface, інтерфейс прикладного програмування
ETA	Estimated Time of Arrival, передбачений час прибуття
SSE	Server-Sent Events
DTO	Data Transfer Object
СКБД	Система керування базами даних
JWT	JSON Web Token
DRY	Don’t Repeat Yourself, не повторюйся
SOLID	single responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion
ER	Entity-Relationship, модель «сутність-зв’язок»
CRUD	Create, Read, Update, Delete, створення, читання, оновлення і вилучення даних

ВСТУП

На сьогодні пасажирські перевезення є однією з найпопулярніших послуг, що має різноманітні формати: від традиційного таксі до сервісів на зразок Uber, де компанія виступає лише посередником між водієм і пасажиром. Ця галузь активно автоматизується, проте класичне таксі досі зберігає свої позиції. За даними 2024 року, у США 18% населення користуються звичайним таксі, тоді як 29% – сервісами на кшталт Uber [1]. В інших країнах ситуація відрізняється: наприклад, у Південній Кореї лише 9% людей користуються сервісами спільних поїздок, тоді як 58% хоча б інколи викликають таксі [1]. Ринок пасажирських перевезень постійно еволюціонує під впливом новітніх технологій, урбанізації та зміни поведінки споживачів. Популярність подібних сервісів спільного користування транспорту, поширення платформ замовлення поїздок, та інтеграція з громадським транспортом змушують змінюватись традиційні моделі перевезення. У цьому контексті особливої актуальності набуває пошук ефективних технічних рішень, що дають змогу оптимізувати маршрути, зменшити час очікування, а також підвищити рівень комфорту й безпеки пасажирів.

Об’єктом дослідження є функціональність і структура додатків пасажирських перевезень, а також особливості побудови масштабованого API, що виконувало б функції додатку пасажирських перевезень.

Предметом дослідження виступають технологічні підходи до організації пасажирських перевезень, зокрема методи побудови маршрутів, розробки API для оптимізації замовлень, а також моделі інтеграції послуг сумісного транспорту з уже наявною інфраструктурою.

Мета дослідження полягає в створенні прототипу прикладного програмного прото в формі Web API для інтеграції в якості бекенд-частини у власні платформи пасажироперевезень.

У межах кваліфікаційної роботи обрано до розгляду та реалізації такі завдання:

1) дослідити ринок додатків пасажирських перевезень, проаналізувати сучасні додатки сумісних поїздок, їхніх переваг і недоліків;

2) розробити архітектуру API для сервісу сумісного автобусного перевезення з використанням практик аналогічних додатків пасажирських перевезень;

3) виконати програмну реалізацію прототипу API в відповідності до розробленої архітектури;

4) відтестувати програмну систему відповідно до визначених вимог та тестових сценаріїв.

РОЗДІЛ 1
ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Базові поняття в контексті пасажироперевезень

Згідно з Федеральною адміністрацією автомобільного транспорту США, під терміном «сумісні поїздки» розуміють «…організовані поїздки, під час яких кілька пасажирів разом подорожують на одному транспортному засобі, маючи спільні або схожі маршрути. Це включає карпулінг (carpooling) та ванпулінг (vanpooling) як форми спільного використання автомобілів, де кілька осіб ділять транспортний засіб для спільних поїздок, зазвичай на роботу або навчання» [2].

Залежно від країни, правила, які стосуються перевезення людей, можуть суттєво відрізнятися. Це не лише технічні обмеження, що можуть бути виправлені технічними правками. Законодавство може впливати навіть на те, які можливості взагалі має перевізник або сервіс. Наприклад, у Південній Кореї не можна створювати застосунки, які дають змогу людям об’єднуватися для спільних поїздок. Якщо водій не має спеціальної ліцензії, а просто їздить на своїй машині й бере гроші – це вже порушення [3].

Популярність додатків для пасажироперевезень може залежати й від культурного контексту. Наприклад, у Великобританії, де типові жовті таксі стали символом країни, 48% людей обирають звичайне таксі, на відмінну від 19% для сумісних поїздок. Повною протилежністю є Індія, де глобалізація наступила пізніше, й велика кількість людей обирають достатньо прибуткову роботу в регіоні – таксування. У такому суспільстві як Uber, так і звичайне таксі користуються подібним рівнем попиту. Звідси, можна зробити висновок, що є країни, де подібні додатки було б набагато складніше розвивати за різними причинами, зокрема, історичний контекст, обумовлений розвитком звичайного таксі, та політичний контекст, як у Південній Кореї або Китаї [1].

На відміну від звичних моделей перевезення пасажирів, де всі учасники заздалегідь погоджують умови (пасажир знає точку прибуття, водій розуміє, кого везе, а оператор координує загальний хід виконання замовлення), у випадку з сумісними поїздками ситуація дещо інша. Учасники не завжди мають повну картину заздалегідь, тому доводиться бути більш гнучким. Усе – від часу до складу маршруту – формується динамічно, залежно від того, як складаються запити, місця знаходження учасників та умов, які динамічно з’являються. Через це система повинна бути не просто стабільною, а такою, що встигає адаптуватись у реальному часі до нових змін.

У центрі такої системи є три сторони: пасажир, водій і сервіс, що обслуговує ці поїздки. Усе починається з того, що пасажир формує запит: обирає, звідки й куди їхати. Далі цей запит потрапляє до водія, якщо той перебуває неподалік. Водій бачить інформацію й вирішує, чи брати замовлення. За позитивного рішення відбувається з’єднання. Сама система при цьому паралельно підбирає маршрут, обчислює приблизну вартість і зв’язує обидві сторони. Така логіка має працювати без затримок, оскільки саме від цього залежить, чи поїздка взагалі відбудеться. Для цього потрібна точна робота механізмів маршрутизації й ціноутворення [4, с. 27].

Через свою динамічну природу такі виклики включатимуть розробку ефективних методів аналізу та обчислення даних для моделей подорожей і мобільності. Програмна система має характеризуватись високою масштабованістю, наприклад, працювати на рівні міста або в кількох районах одночасно, щоб фіксувати, як змінюються рух і поведінка водіїв та пасажирів, не зупиняючи сам процес. Додатково існує потреба в обробці гетерогенних даних, оскільки інформація приходить від багатьох пристроїв, часто в неструктурованому чи частково структурованому вигляді. Проте потреба систематизації даних залишається, щоб маршрути були більш точні й справді оптимальні.

Систему сумісних поїздок треба розглядати як широку сукупність понять [7]:

– сумісне використання таксі – послуга, за якої кілька пасажирів, що рухаються в одному напрямку, ділять одне таксі, щоб зменшити вартість поїздки;

– сумісне використання автомобілів – практика, коли приватний автомобіль використовують кілька людей, що дає змогу не тільки зекономити на паливі, а й зменшити трафік та шкоду для довкілля;

– слагінг – менш офіційна модель, де водій і пасажири домовляються про поїздку самостійно; часто безкоштовно або просто заради спільної вигоди, наприклад, щоб мати доступ до спеціальних смуг для громадського транспорту;

– об'єднання автобусних перевезень, коли різні маршрути або навіть окремі перевізники координують свої графіки й транспорт, щоб уникати дублювань і використовувати ресурси ефективніше.

Виділяючи транспорт, яким можуть користуватися пасажири, зручно поділити його на кілька основних типів: (1) наземний транспорт; (2) залізничний транспорт; (3) авіаційний транспорт; (4) водний транспорт. Наземний транспорт включає автобуси, трамваї, тролейбуси, а також приватні машини і навіть вантажівки, якщо ними дозволено возити людей. Залізничний транспорт передусім охоплює швидкі потяги, якими користуються на міжміських напрямках. Авіаційний транспорт передбачає звичайні рейси (як за графіком, так і чартерні), які забезпечують зв'язок між містами і країнами. До водного транспорту можна віднести пороми, що переправляють пасажирів, а також кораблі для довших поїздок, наприклад, круїзів.

Хоч можна уявити собі спільні перельоти або поїздки водним транспортом, насправді таких моделей поки що майже немає [5]. Крім невеликої цільової аудиторії, складно зробити такі поїздки зручними. Тому увага переважно фокусується на наземному транспорті. Такі перевезення мають більший потенціал у цьому контексті: багато транспортних засобів та водіїв, і кожен може, теоретично, підключитися до системи й почати возити пасажирів.

При класифікуванні перевезень орієнтуються на кілька критеріїв: тип маршруту, регулярність поїздок та зручність подорожей. За маршрутом поїздка може бути пряма або з пересадкою. За регулярністю вона або постійна, або час

від часу, або разова. Існують також спеціальні рейси, які виконуються тільки на замовлення. Щодо комфорту поїздки зазвичай поділяють на економ, бізнес і преміум класи. Але ці категорії не завжди чіткі, оскільки фірми самостійно обирають ступінь якості сервісного обслуговування.

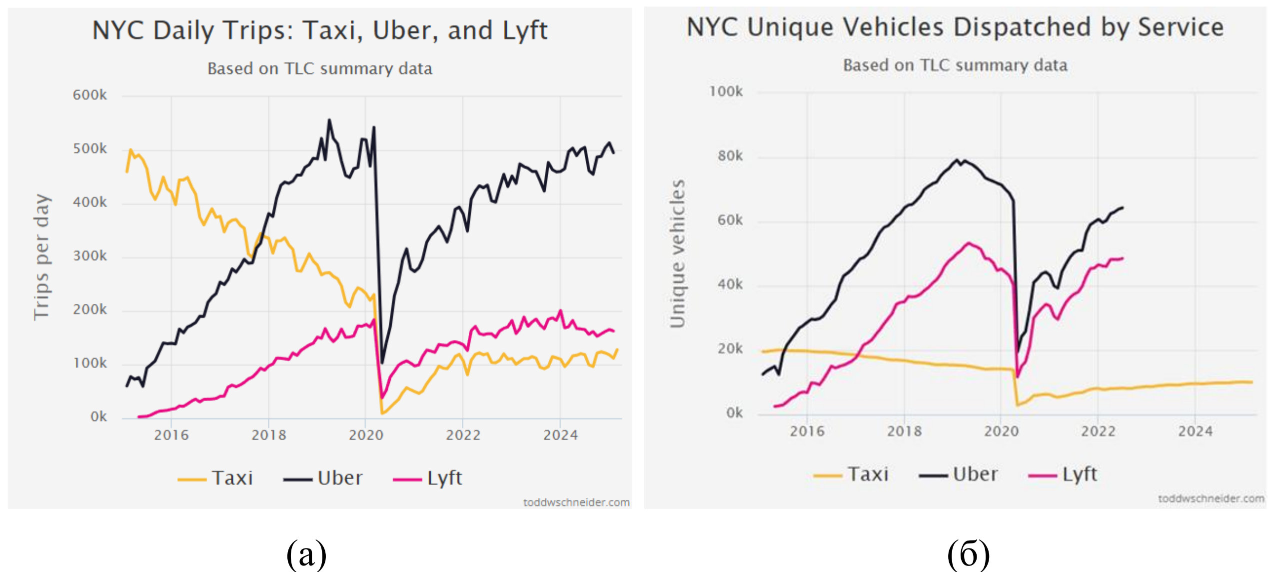
Індивідуальні перевезення характеризуються тим, що пасажир самостійно формує запит, без фіксованого маршруту чи часу. Ціна залежить від багатьох факторів: відстань, час, кількості учасників. При регулярних поїздках заздалегідь відомі маршрут, зупинки та час відправлення, у той же час, нерегулярні не мають чіткого графіка. Чартерні подорожі передбачають замовлення транспорту під конкретні події чи для обмеженого списку учасників [6].

Правове регулювання теж визначає коло можливостей, доступних користувачам додатків для сумісних пасажироперевезень. Особливо важливо, щоб були прописані права пасажирів, зокрема, право знати розклад та ціни, отримати компенсацію, якщо рейс скасували або він запізнився. Перевізник, у свою чергу, повинен дотримуватись правил, не порушувати домовленостей, відповідати за речі та людей, які перевозяться. Особливо складна ситуація з сервісами типу Uber або Bolt через різне ставлення до них у різних країнах. У результаті формуються нерівні умови, які створюють напругу на ринку, тому саме закон має зробити правила зрозумілими й однаковими для всіх [7].

При оцінюванні ефективності перевезень розглядаються показники кількості перевезених людей (пасажиропотік), наповненості транспорту (завантаженість), комфортності та собівартості поїздки [6]. Звідси, програмна система має включати в формули розрахунків ціну поїздки, беручи до уваги такі зміні, як пасажиропотік, завантаженість транспортних засобів, собівартість перевезень, рівень комфорту та безпеки, відстань, тип автомобіля, тариф і завантаженість дороги.

1.2 Сучасний стан ринку додатків для пасажироперевезень

Прогнози до 2027 року показують стабільне зростання ринку пасажироперевезень, десь на рівні 16% щорічно. Це пов'язано з широким розповсюдженням комунікаційних технологій, зокрема, смартфонів та доступу до мережі Інтернет [8]. Для наочності можна згадати Нью-Йорк. Ще у 2015 році Uber там здійснив понад 36 мільйонів поїздок. За статистикою (рис. 1.3) видно, що звичайні таксі домінували протягом 2015-2018 років, проте відтоді зберігається стійка тенденція до переважання сумісних поїздок, хоч і з видимим впливом пандемії коронавірусу в 2020 році. [8] Аналізуючи статистику, можна дійти висновків, що традиційні таксі вийшли на стійкий показник перевезень після поступового зниження, а кількість автомобілів Uber і Lyft має майже прямо пропорційний характер кількості щоденних поїздок цих самих сервісів.



(а) (б)

Рисунок 1.1 – Порівняння статистики для Нью-Йорку (а) щоденних поїздок; (б) унікальних машин різних сервісів [8]

Функціональність будь-яких додатків пасажироперевезень багато в чому збігається. Наприклад, Uber має приблизно такі функціональні можливості: (1) запит на поїздку; (2) відстеження в реальному часі; (3) розділення платежу; (4) поширення подробиць поїздки; (5) прозорість ціни; (6) оплата різними

способами; (7) розділення точок висадки; (8) рейтинг та оцінки; (9) інформація щодо поїздок у минулому; (10) можливість тримати зв'язок з водієм в додатку; (11) відкладена поїздка. [9]

У контексті формування запиту на перевезення, щоб викликати авто, достатньо відкрити додаток, вказати точку доставки й підтвердити. Додаток самостійно знаходить водія поблизу. Далі користувач здійснює перегляд маршруту в реальному часі. Одразу після підтвердження на карті стає видно, де перебуває автомобіль, як він рухається в напрямку до пасажира, а потім і власне маршрут поїздки.

При запиті на поїздку можливо розділення суми оплати при використанні одного й того ж авто декількома людьми. При роботі з іншими людьми також можливе поширення інформації щодо поїздки, що може бути корисним в контексті особистісної безпеки при користуванням таксі. За таких умов інформація завжди передається іншим людям і відхилення від маршруту неможливе.

Особливо важливою функцією будь-якої системи перевезення пасажирів є автоматичне обрахування ціни поїздки до запуску цієї поїздки. Подібна система не дає можливості шахраювати водіям і планувати поїздку або фінанси користувачу. У подібних системах оплата має різноманітний характер: через додаток, готівкою або карткою. Це заздалегідь має налаштовуватися перед поїздкою.

У випадках, коли поїздку ділять одразу декілька людей, можливе розділення поїздки на декілька точок висадки. Система будує один маршрут таким чином, щоб висадити кожного з пасажирів окремо, не перебудовуючи повний маршрут. Після маршруту користувачу надається можливість оцінити поїздку і залишити коментар щодо водія. Така ж функціональність надається й водію.

Усі поїздки зберігаються в особистому кабінеті: з адресами, цінами, іменами водіїв. При потребі можна переглянути деталі або повторити маршрут у майбутньому. В додатку також є можливість спілкуватись користувачу з водієм,

що прибирає потребу у дзвінку за особистим номером водія, дозволяючи сфокусуватись на поїздці.

Планування поїздок – функціональна можливість, що допомагає користувачам запланувати поїздку заздалегідь. Користувач задає час і місце, система же відправляє запити різним водіям, які можуть погодитись на заплановану поїздку.

Підтримка користувачів реалізується всередині застосунку у вигляді чату або форми. Звернення обробляються через бекенд-систему тікетів, часто з інтеграцією сервісів типу Zendesk чи Intercom для масштабованості [10]. У таблиці 1.1 наведено порівняння подібної функціональності у різних райд-шейрингових компаній.

Таблиця 1.1 – Порівняння функціональності для користувачів у різних компаній з пасажироперевезень

№	Функція	Uber	Bolt	Uklon	Lyft
1	Запит на поїздку	Так	Так	Так	Так
2	Відстеження авто в реальному часі	Так	Так	Так	Так
3	Розділення оплати між пасажирами	Так	Ні	Ні	Так
4	Надсилання маршруту друзям	Так	Так	Так	Так
5	Прозорий розрахунок вартості	Так	Так	Так	Так
6	Кілька способів оплати	Так	Так	Так	Так
7	Декілька точок висадки	Так	Так, до 4 зупинок включно з кінцевою	Так	Так
8	Система рейтингу водіїв і пасажирів	Так	Так	Так	Так
9	Історія поїздок	Так	Так	Так	Так
10	Зв’язок із водієм через додаток	Так	Так	Так	Так
11	Відкладене бронювання поїздки	Так	Так, до 90 днів наперед	Так	Так, з гарантією вчасного прибуття або компенсацією

Функціональні можливості додатків пасажироперевезень для водіїв також здебільшого збігаються. Прикладом такого є: (1) прийом замовлень; (2) відстеження заробітку; (3) планування графіка; (4) destination mode; (5) автоматичне прийняття замовлень; (6) інформація про зони з високим попитом; (7) перегляд історії поїздок; (8) система рейтингів та відгуків; (9) виведення коштів; (10) підтримка; (11) режим «Останній рейс»; (12) доступ до навчальних матеріалів; (13) заплановані замовлення; (14) інтеграція з навігацією; (15) інтеграція з музичними сервісами; (16) бонусна система; (17) асистент з оптимізації доходу [11].

Водій завдяки додатку отримує інформацію щодо активних поїздок і інформації щодо цих маршрутів, після чого приймають їх. Відображення доходу відбувається також в реальному часі і реалізовано всередині додатку. Це дає змогу краще планувати день та контролювати свій заробіток.

Перевізник також може всередині додатку налаштовувати години своєї роботи, що дозволяє системі і водію надавати найкращу інформацію щодо заказів, або планувати свій час. Якщо водій завершує роботу, він може встановити режим «Їду додому», що дає змогу приймати лише ті замовлення, що ведуть у напрямку заданої адреси. Функціональність замовлень також може бути автоматизованою, за таких умов система автоматично знаходить замовлення і опрацьовує їх, не чекаючи на підтвердження водія.

У таких системах також реалізовано підказки щодо зон з підвищеним попитом, а також журнал поїздок. Такі можливості забезпечують аудит усіх виконаних замовлень, і за потреби водій переглядає усі виконані поїздки.

Система відгуків реалізована також і для водіїв. Після виконання замовлення перевізник залишає оцінку пасажиру і може написати коментар. Зв’язок із підтримкою реалізовано в додатку через чат або дзвінок.

Додаток реалізує виведення коштів на будь яку картку із наявних в системі. Комісія щодо виплат відрізняється в різноманітних сервісах, але зазвичай в проміжку між 20% і 25% [12].

Функція «Останній рейс» існує для завершення зміни водія. Система дає останнє замовлення і більше не турбує. Також для перевізників наявні навчальні матеріали з порадами щодо безпеки, взаємодії з пасажирами і збільшенню доходу.

Опрацювання попередньо запланованих поїздок відбувається також в додатку, водії переглядають попередньо заброньовані маршрути і погоджуються на них. Така система працює в тандемі з системою планування дня і може пропонувати маршрути з вечору.

Інтегрована навігаційна система працює всередині додатку і не потребує використання користувачем сторонніх систем. Така система має вбудовану систему GPS-навігації.

Інтеграція з музикою й асистент оптимізації надаються обмежено в різних сервісах. Застосунок Uber підтримує інтеграцію з Pandora в США, завдяки якій водій керує музикою і налаштовує її напряму в додатку. В більшості країн ця функція недоступна. Асистент з оптимізації доходу надається Lyft. Він аналізує активність водія і завдяки інтеграції з ШІ, підказує де краще вийти на зміну з метою більшого заробітку. Порівняння загальної функціональності між платформами пасажироперевезень наведено в таблиці 1.2.

1.3 Особливості побудови додатків для пасажирських перевезень

Користувач взаємодіє з системою таким чином: створює акаунт, бронює поїздку, система знаходить водія, водій підбирає користувача, відвозить до кінцевої точки, користувач платить за поїздку, користувач і водій оцінюють один одного. Таким чином система охоплює основні потреби користувача. Типовий сценарій використання додатка пасажирами продемонстровано на рис. 1.2а.

Водій налаштовує профіль, перевіряє заробіток, автоматично переходить в офлайн-режим за потреби, має навігацію, може відмовлятись від поїздки, вибірково налаштовує кращі точки підбору пасажирів. Також перевізники можуть підбирати пасажирів, що чекають на таксі біля узбіччя, обмежувати

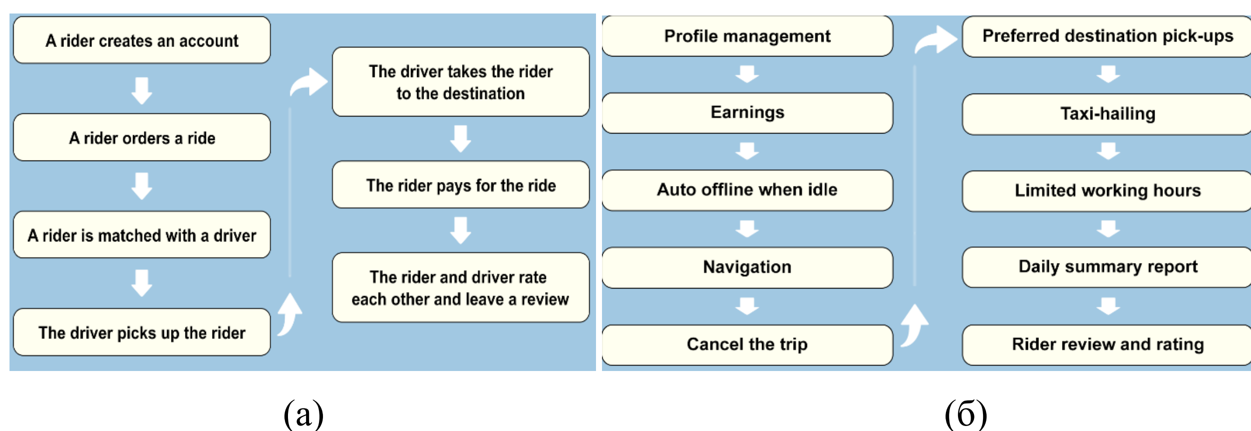
годити роботи, отримувати звіт за день роботи, залишати оцінки пасажирам. Загальний сценарій користування додатком водіями представлено на рис. 1.2б.

Таблиця 1.2 – Порівняння функціональності для водіїв у різних компаній з пасажироперевезень

№	Функція	Uber	Bolt	Uklon	Lyft
1	Прийом замовлень через додаток	Так	Так	Так	Так
2	Відстеження заробітку в реальному часі	Так	Так	Так	Так
3	Планування робочого дня	Так	Частково, залежить від регіону	Так	Так
4	Режим "Їду додому" (Destination Mode)	Так	Так	Так	Так
5	Автоматичне прийняття замовлень	Так	Так	Так	Так
6	Інформація про зони з високим попитом	Так	Так	Так	Так
7	Можливість перегляду історії поїздок	Так	Так	Так	Так
8	Система рейтингів та відгуків	Так	Так	Так	Так
9	Миттєве виведення коштів	Так	Так	Так	Так
10	Підтримка в додатку (чат або дзвінок)	Так	Так	Так	Так
11	Режим "Останній рейс"	Так	Так	Так	Так
12	Доступ до навчальних матеріалів	Так	Так	Так	Так
13	Можливість планування попередніх поїздок	Так	Так	Так	Так
14	Вбудовані навігаційні інструменти	Так	Так	Так	Так
15	Інтеграція з музичними сервісами	Так, через Pandora (у США)	Ні	Ні	Ні
16	Нагороди та бонуси за досягнення	Так	Так	Так	Так
17	AI-помічник для оптимізації заробітку	Ні	Ні	Ні	Так, "Earnings Assistant"

Більш розширена функціональність також включає бронювання поїздок на потім, динамічне відслідковування водіїв, поширення відслідковування поїздки, розподіл оплати, декілька точок висадки, декілька опцій для оплати, вибір кращого водія, перегляд оцінок і коментарів для водія, історія поїздок, збереження точок висадки, чат в додатку, синхронізація з додатком календарю, кнопка «SOS». Візуалізація роботи з розширеними можливостями показана на рис. 1.3.

Для адміністраторів системи функціональність включає відкладені запити, отримання статусу водія, оцінки користувача, відстеження водія, меню партнерів, типи сервісу, наявні країни і заробіток водіїв.



(а) (б)

Рисунок 1.2 – Базові можливості додатка пасажироперевезень (а) для користувача; (б) для водія [13]

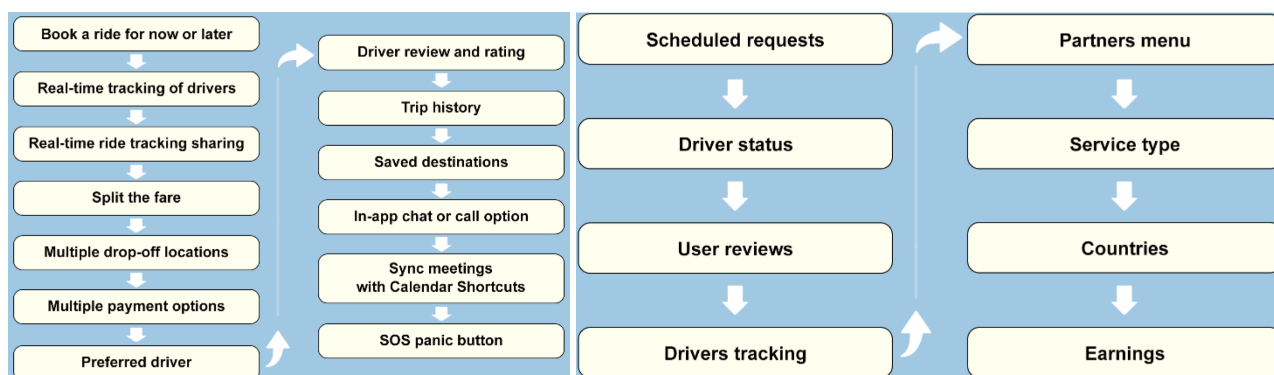


Рисунок 1.3 – (а) розширені можливості додатка пасажироперевезень для користувача; (б) базові можливості додатка пасажироперевезень для водія [13]

За діаграмою прецедентів з рис. 1.4 стає зрозуміло, як з системою працюють різні групи користувачів і відбувається розділення функціональних можливостей та відповідальності в системі. Користувач має доступ лише до потрібної йому функціональності для замовлення поїздок та інформації щодо них. У свою чергу, водій отримує доступ до опрацювання поїздок і потрібної йому роботи.

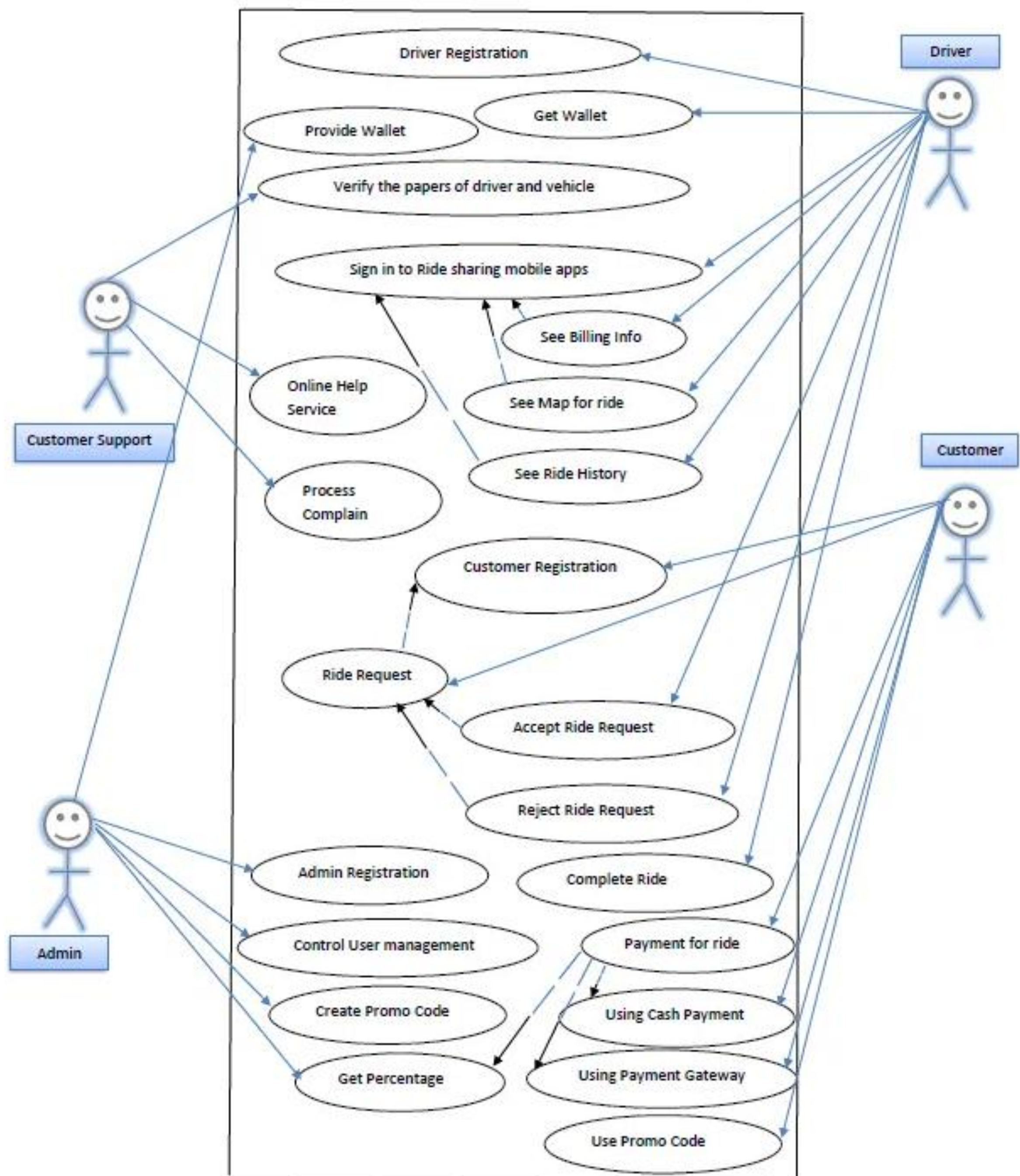


Рисунок 1.4 – Діаграма прецедентів роботи системи пасажироперевезень [14]

Однією з основних задач програмного додатка для сумісних поїздок є з'єднання між собою водія і пасажира. Зберігання інформації та пошук таксі обов'язковий фактор для працездатності подібних додатків. Для таких цілей використовується база даних. [4, с. 11] На рисунку 1.5 наведено приклад схеми бази даних, проте варто зауважити залежність між архітектурою даних та архітектурою програмної системи в цілому.

Реляційна база даних будується на таблицях і зв'язках між ними. Прототип схеми бази даних з рис. 1.5 передбачає такі сутності, як drivers (водії); cabs (автомобілі); raitings (рейтинг); customers (користувачі); payments (оплата) й trips (поїздки). Відповідно до розглянутих раніше типових можливостей додатків для пасажироперевезень, таблиця drivers з даними водіїв пов'язана зв'язком «1:1» з таблицями cabs, ratings та trips, оскільки кожного моменту один водій може керувати не більше, ніж однією поїздкою, залишати єдину рейтингову оцінку за неї та керувати лише одним транспортним засобом.

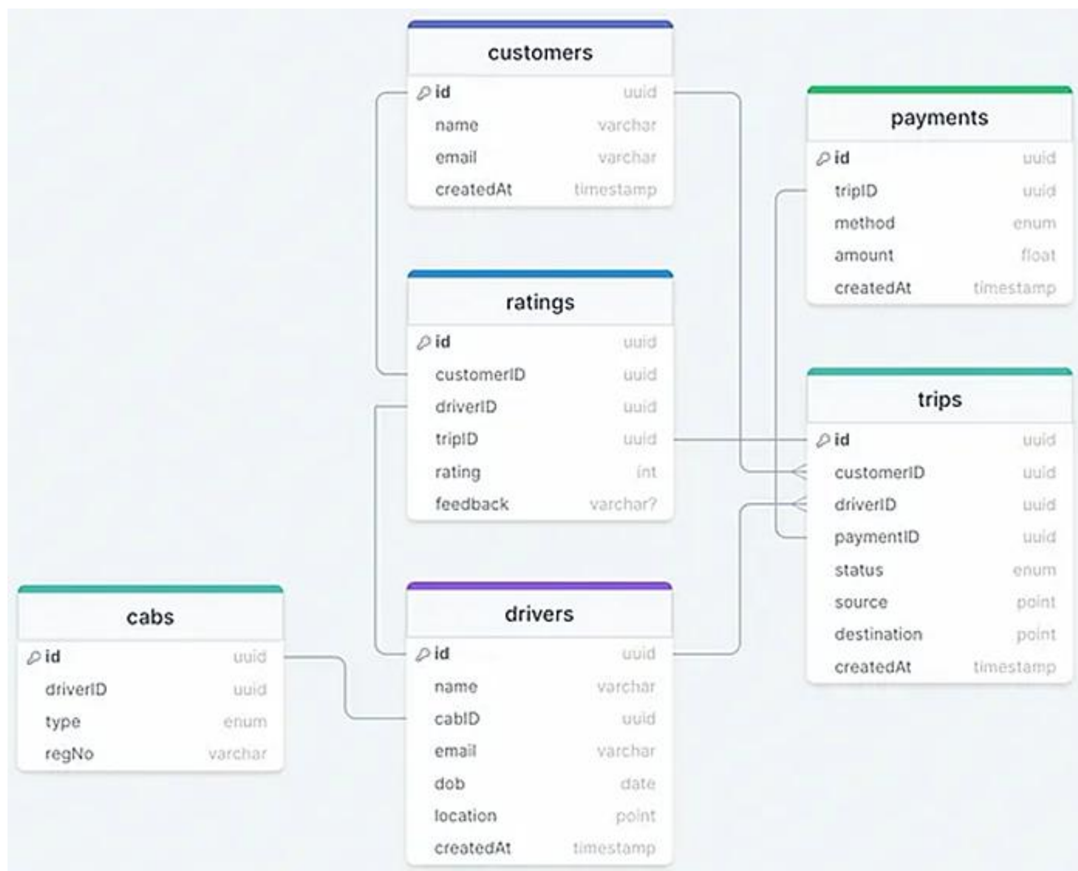


Рисунок 1.5 – Зразок схеми бази даних для пасажирських перевезень [15]

У той же час, таблиця customers пов’язана з таблицею drivers також типом зв’язку «один до одного», оскільки одномоментно клієнти здійснюють поїздку лише з одним водієм. Проте загалом збереження історії поїздок передбачає зв’язок «1:М» з таблицею trips. Враховуючи те, що платежі стосуються конкретних поїздок, таблиця payments має зв’язок лише з таблицею trips, в якій через зовнішні ключі можна отримувати й інформацію щодо учасників подорожі.

Відстеження маршруту в реальному часі, обчислення вартості поїздки, система для керування водіями, зручна форма підтримки та обробка платежів повинні бути інтегрованими в екосистему. Без відстеження пасажиру та водію буде значно складніше синхронізувати свій рух назустріч одне одному, оцінювати, скільки часу залишилось до приїзду. При роботі з тарифами важлива прозорість. Тариф має рахуватись без ручного втручання за відомими обом сторонам показниками [16].

Сервіс таксі використовує GPS та WebSocket-з’єднання, щоб передавати координати водія і пасажира кожні кілька секунд [17]. Це дозволяє обом учасникам бачити рух у реальному часі. Такі картографічні сервіси, як Google Maps або Mapbox, будують маршрут і розраховують ETA (час прибуття) через API, а стрімінг забезпечує миттєве оновлення на карті.

Система GPS має велике навантаження через постійне оновлення даних перевізників, що змінюють своє розташування й відправляють дані по деяких, обраних розробниками додатка каналах зв’язку. Загалом існує два основних підходи: (1) Push model; (2) Pull model [18, с. 3].

Pull-модель (модель витягування) передбачає, що клієнт сам надсилає запити до сервера. Здійснюється це не постійно, а з певним інтервалом. Кожного разу він може передати своє місцезнаходження, а також отримати щось у відповідь, наприклад, інформацію про те, скільки часу залишилось до прибуття або яка наразі вартість. Така модель не завжди миттєва, тому що оновлення не приходять автоматично. Один зі способів реалізувати такий підхід – використати long polling. У такому сценарії клієнт звертається до сервера, але не отримує відповідь одразу. Якщо є якісь нові дані, сервер надішле їх, проте якщо нічого

нового не з’явилось, то запит очікує на зміни або тайм-аут. Як тільки відповідь прийшла, клієнт знову запускає такий же запит, і все повторюється. [18, с. 10]

Push-модель (модель надсилання) організована так, щоб клієнт спочатку встановлював стабільне з’єднання з сервером, а потім уже власне сервер надсилає оновлення, як тільки вони з’являються. Дані приходять одразу, без повторних запитів. Це зручно, коли важлива швидкість реакції. Технічно це можна реалізувати через WebSocket або Server-Sent Events (SSE), якщо не потрібна двостороння передача. Загальне порівняння технологій здійснено в таблиці 1.3.

Таблиця 1.3 – Порівняння технологій реалізації відстеження учасників пасажироперевезень

Параметр	Long polling	WebSockets	Server-Sent Events (SSE)
Тип зв’язку	Запит-відповідь	Двосторонній	Односторонній (сервер → клієнт)
Затримка	Висока	Низька	Низька
Масштабованість	Низька	Висока	Середня
Ресурсомісткість	Висока	Низька	Низька
Підтримка браузерами	Висока	Висока (але потребує підтримки WebSockets)	Висока (крім IE)

Вартість поїздки розраховується автоматично за формулою: базовий тариф плюс плата за відстань, час і коефіцієнт попиту (surge). GPS-дані зберігають реальний маршрут і тривалість, без втручання людини. Після завершення поїздки пасажиру надсилається детальний електронний чек [19].

У сучасній фінансовій екосистемі співіснують різноманітні форми оплати: готівка, банківські картки, сервіси третіх сторін і криптовалюта, причому кожна з них має свої функціональні переваги й обмеження. Наприклад, готівкові розрахунки чи банківські перекази вирізняються високим ступенем надійності через правову визначеність і є простими в використанні, однак втрачають ефективність у цифровому середовищі через низьку швидкість та відсутність гнучких механізмів захисту прав споживача [20].

Системи третіх сторін (наприклад, Alipay, PayPal або WeChat Pay) забезпечують зручність, швидкість, а також захист обох сторін транзакції завдяки посередництву. Вони сприяють зменшенню ризиків недобросовісної поведінки учасників ринку, водночас зіштовхуючись із проблемами регуляторного контролю, надмірної ринкової концентрації та ризиками витоку персональних даних. Крім того, активне зростання подібних платформ створює конкуренцію банкам, впливаючи на традиційні моделі прибутку фінансових установ [21].

Криптовалюта як форма децентралізованого цифрового активу відкриває нові перспективи у сфері міжнародних платежів, знижуючи транзакційні витрати та посилюючи фінансову автономію користувачів. Утім, її широке застосування стримується високою волатильністю, нормативною невизначеністю та обмеженою довірою з боку інституційних учасників. Отже, жодна з платіжних систем не є універсальною, і ефективна фінансова інфраструктура має передбачати їхнє паралельне використання, оптимізуючи співвідношення між зручністю, безпекою та регуляторною відповідністю.

З огляду на все вищезгадане, стає зрозумілим, що подібне програмне забезпечення повинно поєднувати різні елементи: від логіки маршруту до підтримки, від API до платіжних шлюзів. Загальна архітектура таких проєктів має витримувати високі навантаження, легко масштабуватись та бути відмовостійкою. Крім того, додаток повинен захищати персональні дані учасників в іншому випадку можлива проблема довіри до додатку з сторони користувачів і вибір конкурентів.

Останнім часом найбільш поширеним архітектурним стилем є мікросервісна архітектура [22]. Центральним компонентом за такого підходу може стати API Gateway, через який проходять усі запити (рис. 1.6). З ним працюють окремі сервіси, що реалізовані незалежно один від одного, навіть на різних мовах програмування. Це зручно для масштабування, оскільки можна збільшити виділені ресурси лише для того сервісу, якому бракує ефективності під навантаженням.

За таких умов реалізацією більш складної бізнес-логіки займається окремий API, комунікація з яким може здійснюватись за цілою низкою протоколів та архітектурних стилів: (1) REST API; (2) GraphQL; (3) gRPC; (4) SOAP; (5) WebSockets API; (6) Graph API тощо. Зазвичай додатки для організації пасажироперевезень використовують кілька підходів і зв'язують їх між собою. Порівняння відповідних технологій наведено в таблиці 1.4.

Для реалізації відкритості в взаємодії з даними у додатків пасажироперевезень виникає необхідність працювати одразу з декількома підходами при реалізації функціональності. Для імплементації CRUD-запитів можливе використання REST API, що також опрацьовує дані і має власну бізнес-логіку в реалізованих внутрішніх сервісах. У свою чергу, реалізація GPS-трекінгу можлива завдяки WebSockets API, що допомагає тримати з'єднання в реальному часі. [23]

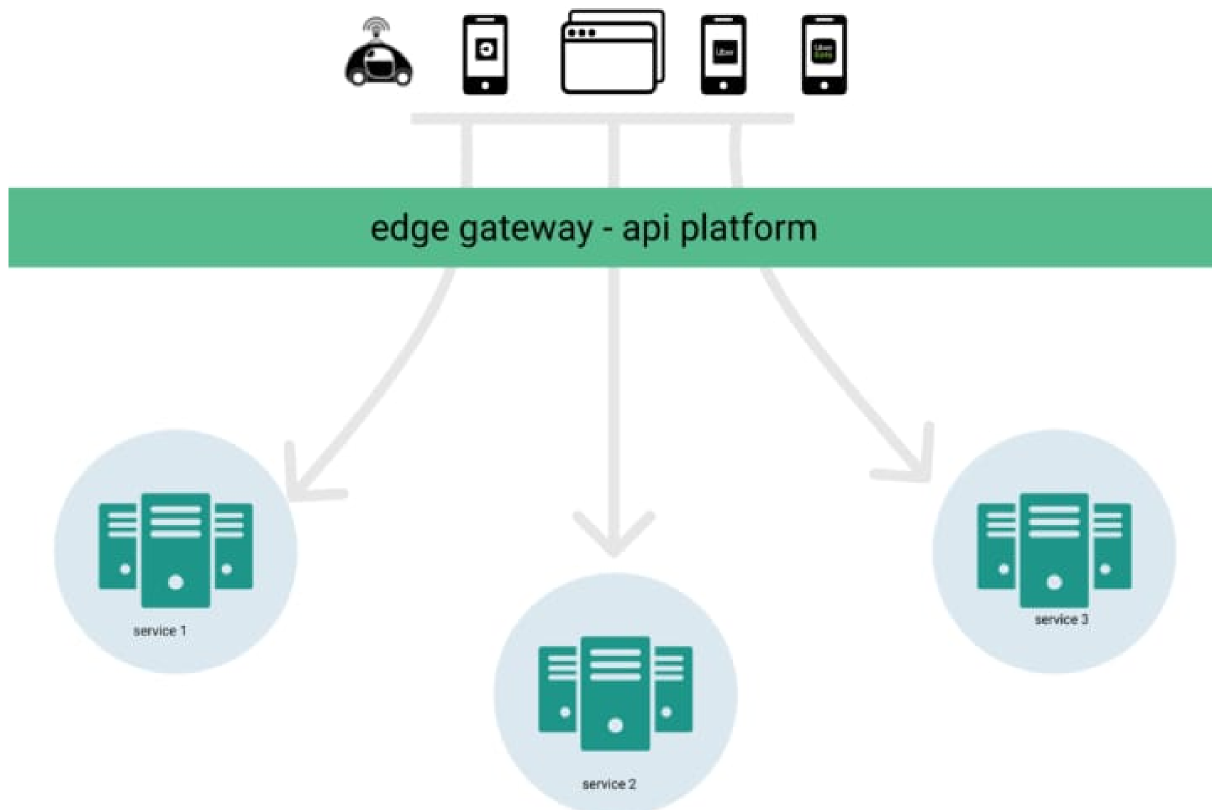


Рисунок 1.6 – Візуалізація взаємодії через API Gateway на платформі Uber [24]

Таблиця 1.4 – Порівняння технологій побудови API

Критерій	REST	GraphQL	gRPC	SOAP	Websockets	Graph API
Простота у використанні	так	так	ні	ні	так	так
Відкритість / Виявлення (discoverable)	так	ні	ні	так	ні	ні
Надійність	так	так	так	так	ні	ні
Продуктивність	так	ні	так	ні	так	так
Масштабованість	так	ні	так	ні	так	так
Спостережуваність	так	ні	так	так	ні	ні
Гнучкість запитів	ні	так	ні	ні	ні	так
Стрімінг / Постійне з'єднання	ні	ні	так	ні	так	ні
Формальна структура	ні	ні	так	так	ні	ні
Підтримка складних зв'язків	ні	ні	ні	ні	ні	так

За результатами огляду, було визначено, що найважливішою в розробленні подібних систем є можливість інтеграції сервісів між собою через шини зв'язку. Кращим прикладом такого підходу є API, що легко інтегруються в готові системи.

1.4 Постановка задачі на розробку

Слабкі місцями сучасних моделей автобусних перевезень: або повна відсутність гнучкості, або закритість до інтеграцій. Тому в межах кваліфікаційної роботи здійснюється спроба реалізації новітнього сервісу сумісних автобусних перевезень. Маршрути формуються з окремих відрізків – субпоїздок, які логічно поєднуються одна з одною. Завдяки такому підходу створюється осередок для гнучкості системи: один маршрут матиме багато можливих точок посадки і висадки. Інші сервіси надають таку гнучкість лише в системах таксі чи карпулінгу.

За результатами попереднього огляду було вирішено сфокусуватись на розробленні REST API для додатку сумісних автобусних поїздок. Створення API відбуватиметься на базі мови програмування C#, технологій ASP.NET Core REST API та об'єктно-реляційного відображення засобами Entity Framework Core, використовуючи метод Code First. Імплементація бази даних здійснюватиметься

засобами СКБД PostgreSQL. Також буде застосована технологія DTO для зручнішого отримання даних на фронтенді додатку [25, с. 7].

API повинен не лише обробляти запити до бази даних, а й здійснювати обчислення, що стосуються бізнес-логіки, реалізованої сервісами. До важливої функціональності включено розрахунок вартості, часу та маршруту поїздки; опрацювання push-повідомлень; інтегрована система оплати; відстеження водія в реальному часі.

Висновки до першого розділу

У межах першого розділу було детально проаналізовано актуальні функціональні можливості додатків для пасажироперевезень, їх структура, можливе використання, технології та архітектурні рішення, які використовуються при розробці. За результатами аналізу було визначено базові функціональні та нефункціональні рішення, що мають лягти в основу розроблення Web API для організації сумісних поїздок на автобусному транспорті.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ WEB API

2.1 Аналіз вимог до Web API

Загальна ідея додатка полягає в тому, що водій створює маршрут з кінцевою точкою прибуття, який реєструється системою. Користувачі створюють запит зі своєю точкою прибуття, система підбирає найкращі маршрути (найменший відсоток відхилення і задовольняють усі фільтри, що виставленні пасажиром). Ціна обчислюється залежно від відстані, тарифу водія і відсотка відхилення маршруту. Користувач вибирає найзручніший із варіантів і відправляє запит водію, водій вправі відмовитись, або погодитись на поїздку.

Коли водій розуміє, що пасажирів достатньо, він виставляє дату від'їзду, якщо вона не була встановлена заздалегідь. Усі пасажири отримують повідомлення з оновленою інформацією і змушенні або відмовитися від поїздки, або заплатити не менше як за двадцять чотири години до відправлення. Вони також отримують ще одне повідомлення ближче до дати відправки.

Водій починає поїздку, система опрацьовує її як список точок висадки, з яких будується повний маршрут. Досягаючи кожної точки з списку, водій залишає одного з пасажирів, які, в свою чергу, пишуть коментар та виставляють оцінку поїздці. Досягаючи останньої точки в списку маршрутів, водій отримує виплату і залишає свою оцінку пасажирам.

У межах кваліфікаційної роботи було спроєктовано три бізнес-кейси з використанням API: (1) бронювання поїздки пасажиром; (2) використання API приватним перевізником; (3) інтеграція API в систему пасажирської компанії.

Метою першого бізнес-кейсу є надання пасажирам простої можливості знайти зручний маршрут та забронювати поїздку без потреби взаємодіяти з операторами. Пасажир відкриває додаток, вводить пункт призначення, після чого система автоматично підбирає наявні маршрути. Користувач обирає відповідний маршрут та надсилає запит на поїздку. Перевізник розглядає запит

та може прийняти або відхилити його. У випадку прийняття, пасажир оплачує поїздку через додаток. По завершенню поїздки пасажир має можливість залишити відгук, що сприяє покращенню якості послуг. Діаграму прецедентів для такого сценарію наведено на рис. 2.1.

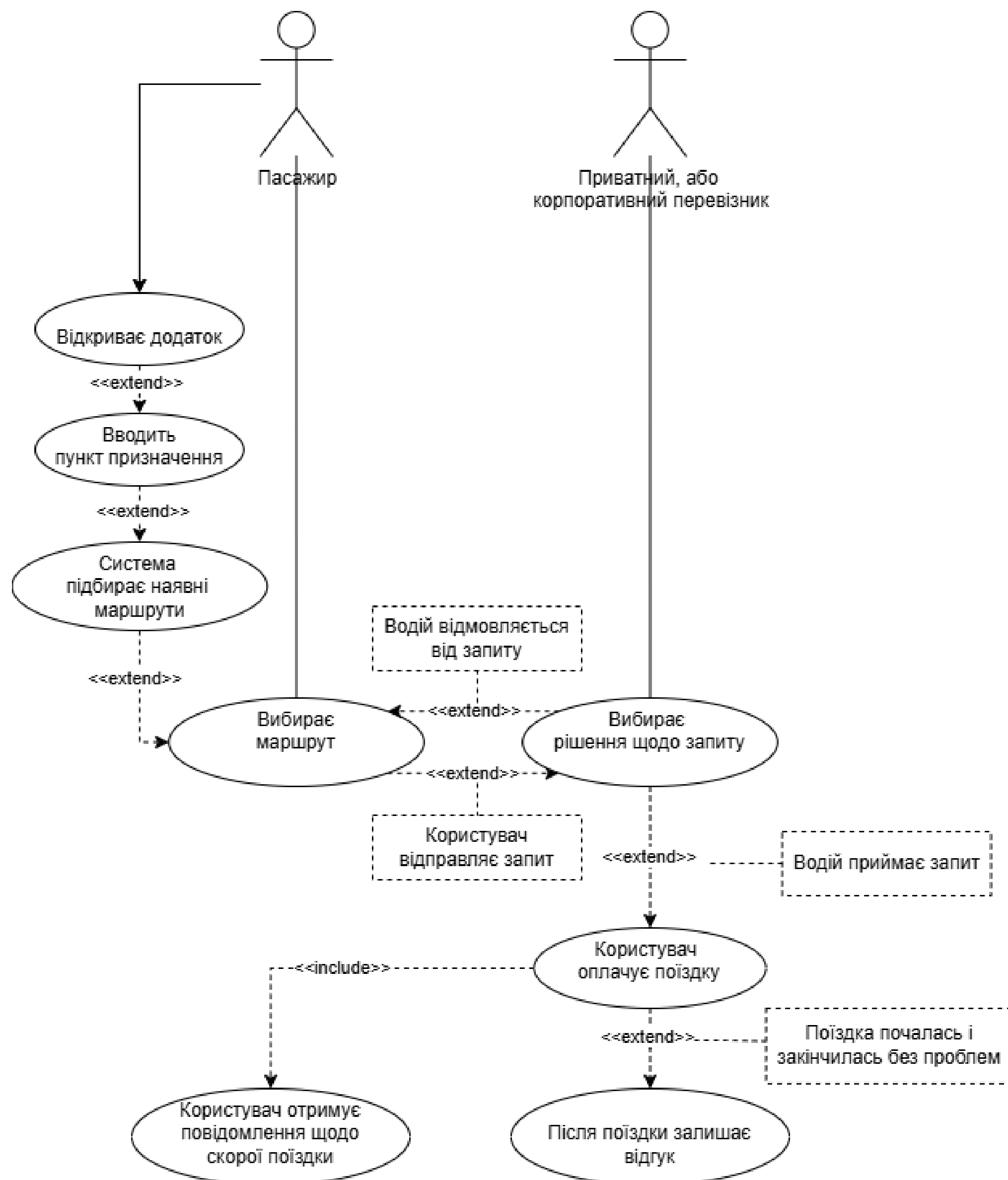


Рисунок 2.1 – Діаграма прецедентів бізнес-кейсу організації поїздки через мобільний додаток

У другому бізнес-кейсі розглядається організація поїздки водієм за допомогою програмної системи з API. Він має на меті здійснити автоматизований підбір пасажирів для певного маршруту, спростити

комунікацію між усіма сторонами та забезпечити контроль над оплатою й виконанням поїздки. Приватний перевізник створює маршрут у системі, після чого система API автоматично підбирає потенційних пасажирів. Пасажири надсилають запити на бронювання місць. Водій може приймати або відхиляти ці запити. Коли система фіксує достатню кількість підтверджених пасажирів, поїздка запускається у встановлений час. Після завершення маршруту перевізник отримує оплату. Діаграма прецедентів для подібного бізнес-кейсу зображена на рис. 2.2.

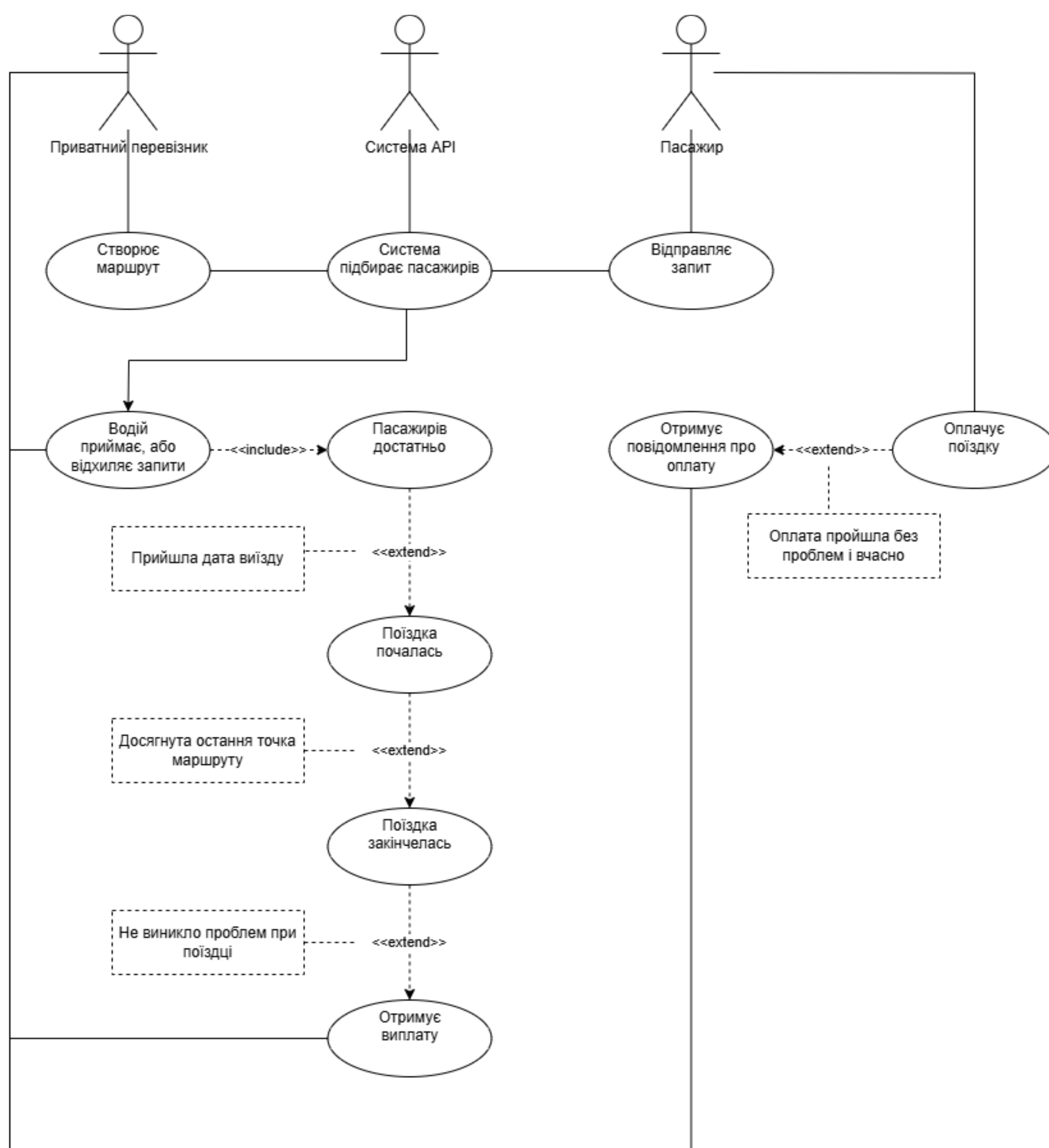


Рисунок 2.2 – Діаграма прецедентів бізнес-кейсу організації поїздки водієм за допомогою API-системи

Третій бізнес-кейс описує сценарій інтеграції компанії-перевізника з системою маршрутів через Web API, що відображено за допомогою діаграми прецедентів на рис. 2.3. Він ставить за мету забезпечити технічну інтеграцію стороннього додатку компанії з централізованою маршрутною системою через Web API для доступу до актуальних даних і бронювання поїздок. Компанія-перевізник впроваджує Web API у власне програмне забезпечення. Персонал компанії надає актуальну інформацію щодо водіїв та маршрутів, а користувачі отримують змогу здійснювати пошук і бронювання маршрутів. Усі запити обробляються на стороні API-сервера, після чого інформація про виконані поїздки повертається до системи компанії.

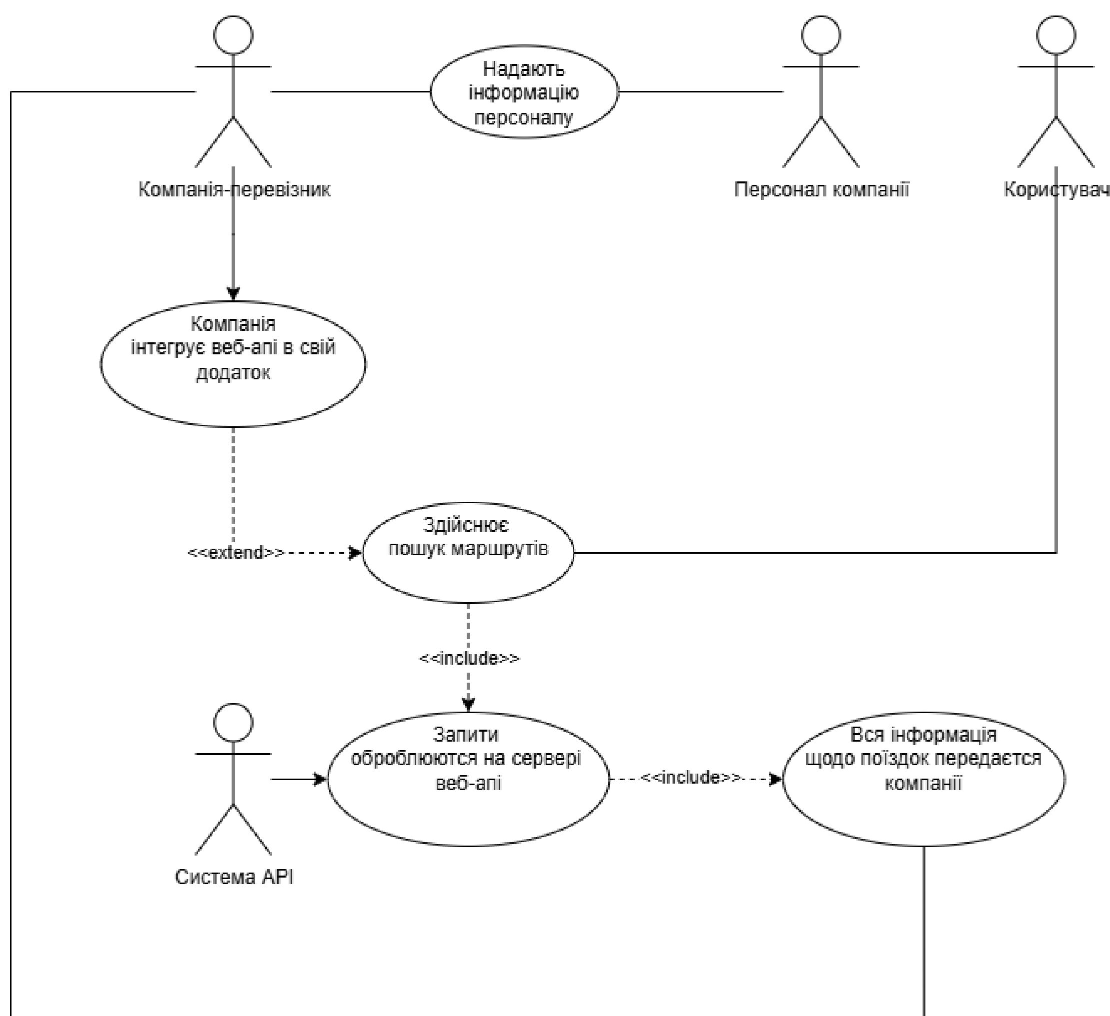


Рисунок 2.3 – Діаграма прецедентів бізнес-кейсу інтеграції компанії-перевізника з системою маршрутів через Web API

Узагальнюючи наведені бізнес-кейси, користувачі можуть виконувати такі дії:

– зареєструватися в системі та увійти в обліковий запис;

– переглядати список доступних маршрутів та деталі кожного маршруту;

– створювати бронь на спеціальну точку;

– переглядати та скасовувати власні бронювання;

– отримувати загальну інформацію про автобуси та водіїв.

Разом з тим, адміністратори можуть додавати, редагувати та видаляти маршрути.

На рисунку 2.4 наведено типовий сценарій використання програмного продукту, який розробляється в межах кваліфікаційної роботи. Програмна система за введеною пасажиром бажаною локацією шукає найкращі маршрути з найменшим відсотком відхилення від неї. Після вибору доречного маршруту, програма надсилає запит на погодження водію маршруту. При узгодженні поїздки система може відредагувати маршрут, щоб зробити його більш зручним для пасажира. При відмові водія користувач може обрати інший доступний маршрут.

Всім, хто записаний на автобус, система шле повідомлення одразу як вони записались, ближче до дати виїзду і за 24 години до початку поїздки. Користувач повинен оплатити поїздку за 24 годин до дати виїзду. Якщо оплата не поступила, бронювання спадає.

Поїздка відбувається за наступним сценарієм: Автобус під’їжджає до точки висадки пасажиру, пасажир виходить, користувач залишає оцінку від 1 до 5 з можливістю написати коментар.

Коли автобус приїхав до останньої точки маршруту, водій отримує свою оплату. Після цього йому теж доступна опція оцінити поїздку або пасажирів, якщо вважає за потрібне.

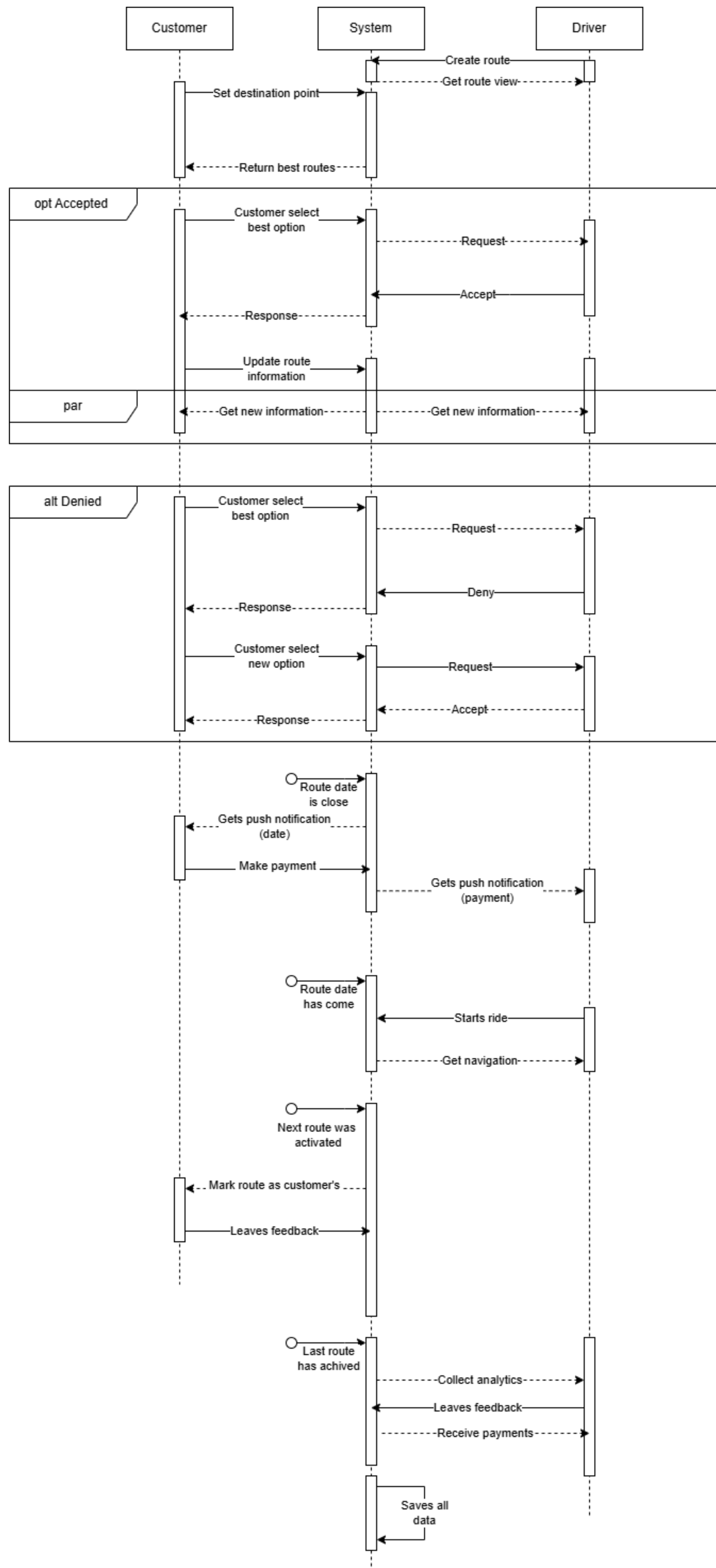


Рисунок 2.4 – Типовий сценарій використання додатку

Автоматизована калькуляція вартості поїздки виконується ще до відбуття і враховує такі зміни: відхилення основного маршруту, спричинене новим пунктом прибуття або проміжною зупинкою; відстань, що займає повний відрізок від відбуття до прибуття до пункту призначення користувача; тариф водія; додаткові затрати.

GPS-трекінг у режимі реального часу відбувається завдяки протоколу WebSockets: встановлюється постійне з’єднання з сервісом динамічного оновлення даних через HTTP (handshake), сервер приймає запит і встановлює двостороннє з’єднання.

Інструменти менеджменту реалізовані завдяки Swagger, що дозволяє переглядати усі можливі таблиці в базі даних та гнучко маніпулювати ними завдяки запитам до API. Загальні функціональні вимоги до розроблюваного програмного забезпечення продемонстровано в зведеній таблиці 2.1.

Таблиця 2.1 – Функціональні вимоги до додатку

Категорія	Функціональна вимога
1. Робота API	API використовує REST-архітектуру; підтримує HTTP-методи (GET, POST, PUT, DELETE); передача даних у форматі JSON.
2. Дії користувачів	Користувачі: реєстрація, логін, перегляд розкладів, бронювання квитків, скасування поїздок, перегляд GPS-локації. Водії: керування рейсами, тарифами, підтвердження поїздок. Адміністратори: управління маршрутами, автобусами, ролями, перегляд та редагування баз даних.
3. Сценарій використання API	Користувач реєструється → обирає рейс → API обчислює вартість → користувач бронює поїздку → отримує статус у реальному часі → поїздка відслідковується через GPS.
4. Калькуляція вартості поїздки	Автоматично до відбуття враховуються: – відхилення маршруту через нову точку прибуття; – повна відстань між відправленням і пунктом призначення; – тариф, встановлений водієм; – додаткові витрати (напр. комісії, сервісні збори).
5. GPS-трекінг	Реалізовано через WebSockets. Використовується HTTP handshake → постійне двостороннє з’єднання. Сервер надсилає динамічні оновлення про локацію автобусів у реальному часі.
6. Інструменти менеджменту	Swagger використовується як інтерфейс управління API. Можливість перегляду і модифікації таблиць у базі даних. Повноцінна взаємодія з ендпоінтами API без потреби сторонніх інструментів.

Розроблення ведеться з використанням мови програмування C# і таких технологій, як ASP.NET Core REST API, Entity Framework Core, PostgreSQL, Google API для побудови маршрутів. Нефункціональні вимоги до Web API автобусних пасажироперевезень зібрано в таблиці 2.2.

Таблиця 2.2 – Нефункціональні вимоги до додатку

Категорія	Вимоги
Продуктивність	Обробка запитів < 1 секунда; підтримка ≥ 50 одночасних з'єднань для стабільної роботи при навантаженні.
Безпека	Аутентифікація через JWT; реалізація ролей доступу (користувач, водій, адмін); захист від SQL Injection.
Документація	Ведення API-документації відповідно до OpenAPI (Swagger); для зменшення витрат часу на розробку і покращення комунікації між розробниками.
Архітектура API	REST-дизайн; правильна реалізація HTTP-методів (GET, POST, PUT, DELETE тощо) та структура ендпоінтів.

Для забезпечення високої продуктивності Web API використовуються асинхронні контролери, що дозволяє обробляти численні запити одночасно без блокування потоків.

Аутентифікація реалізована за допомогою JWT (JSON Web Token), що підтримується стандартними засобами ASP.NET Core (Microsoft.AspNetCore.Authentication.JwtBearer). Ролі (користувач, водій, адміністратор) реалізовані за допомогою політик авторизації. Для запобігання SQL Injection використовується Entity Framework Core, який формує параметризовані запити до бази даних.

Уся документація до API була створена з використанням OpenAPI (Swagger). У ASP.NET Core це реалізується через бібліотеку Swashbuckle, яка автоматично генерує Swagger UI на основі анотацій у контролерах. API розроблено за принципами REST-дизайну: кожен ендпоінт відповідає певному ресурсу, а методи HTTP (GET, POST, PUT, DELETE) використовуються згідно з їх призначенням.

2.2 Високорівнева архітектура програмного рішення

Зважаючи на бізнес кейси, Web API забезпечує зв'язок між бізнес-логікою на сервері та фронтенд-частиною програми, а також може реалізовувати просту бізнес-логіку, обробляти різні типи запитів, взаємодіяти з базою даних та повертати DTO у форматі JSON, який легко зчитуєтся фронтенд-мовами.

API працює через HTTPS-запити до системи REST API, побудованої мовою програмування C# і технологією ASP.NET Core [26 с. 4-5]. Усі запити і відповіді відбуваються через JSON об'єкти, що на бекенді серіалізуються в Entity-класи, дозволяючи гнучко працювати з даними через Entity Framework Core, і мають простий формат для абстрактної роботи на фронтенд-частині додатка [25, с. 205]. Помилки повертаються з 4xx і 5xx кодами для простого налагодження помилок і роботи з документацією, виявлення проблемних місць та їх усунення. Аутентифікація відбувається через JWT (JSON Web Token), який користувач отримує після авторизації. Всі запити до захищених ресурсів повинні включати токен в заголовку Authorization: Bearer <token> [27].

Завдяки описаній вище моделі вдається зменшити кількість ручної роботи операторів зв'язку чи кількість ведення аудиту поїздок. У контексті роботи приватного перевізника, API допомагає уникнути залежності від сторонніх сервісів, рекламних витрат і ручного пошуку клієнтів. Пасажир, зі свого боку, має змогу обрати місце висадки, що скорегує основний маршрут і вплине на ціну оплату.

Щоб користуватись API, потрібен клієнтський інтерфейс, саме він надсилає запити на сервер [26 с. 41-46]. При інтеграції з сервісами іншої компанії усі запити мають можливість бути відправленими через спеціальній інтерфейс на сервері компанії, або через реалізований фронтенд компанії. У випадку реалізації застосунку для користувача: потрібен телефон, контактні дані (пошта чи номер), і сам застосунок. Після запуску вказується локація, система показує всі актуальні маршрути, користувач знайомиться з деталями й обирає перевізника.

На сервері розгорнуто Web API, який приймає й обробляє HTTP(S)-запити. Оброблені запити передаються до різноманітних сервісів і повертають дані користувачам.

API має кілька основних маршрутів: /api/Buses; /api/Customers; /api/Drivers; /api/Trips; /api/Additions; /api/Auth. Усі вони підтримують повний цикл CRUD-операцій. Крім базової функціональності, реалізовані специфічні сценарії, які покривають потреби реального використання: фільтрація поїздок, динамічна побудова маршрутів, підрахунок цін тощо.

З метою швидкого тестування при розробці, було інтегровано Swagger (OpenAPI). Завдяки цій технології можливе тестування напряму в браузері з метою перевірки усіх запитів і доступності усіх маршрутів з параметрами, які вони приймають, у якому форматі надсилають запити та що приходить у відповідь. Для більшої наочності наведено першу версію документації API через Swagger (рисунок 2.5) [26, с. 48].

Передача даних між клієнтом і сервером відбувається в форматі JSON. На стороні сервера дані автоматично перетворюються в об’єкти Entity. За збереження й читання з бази відповідає PostgreSQL у зв’язці з Entity Framework Core, що дає змогу зручно працювати з LINQ і швидко адаптувати модель під нові вимоги [25, с. 39].

За C4-діаграмою контексту (рисунок 2.6) видно як актори співпрацюють з системою API завдяки HTTP-запитам і отримують відповідь. Після отримання запиту API оброблює його і працює зі сторонніми системами, такими як база даних, чи API повідомлень.

Далі відображено діаграму контейнерів (рисунок 2.7), де можна побачити як і сам API розділено на контейнери для контролерів, сервісів, які реалізують бізнес-логіку і моделей для роботи з базою даних через Entity Framework Core. Самі сервіси імплементуються через механізм впровадження залежностей і працюють також з зовнішніми системами, такими як Google Maps API і Messages API. А також діаграму компонентів (рисунок 2.8), де більш докладно показана взаємодія між контейнерами, їх вміст і деталі реалізації програмного продукту,

як наприклад робота з базою даних, що була реалізована через Database Context, який автоматично формує SQL запити до СКБД PostgreSQL завдяки LINQ запитам, побудованими в тандемі з моделями сутностей бази даних – Entity.

Additions

GET /api/Additions/{table}
POST /api/Additions/{table}
GET /api/Additions
GET /api/Additions/{table}/{id}
PUT /api/Additions/{table}/{id}
DELETE /api/Additions/{table}/{id}

Buses

GET /api/Buses
POST /api/Buses
GET /api/Buses/{id}
PUT /api/Buses/{id}
DELETE /api/Buses/{id}

Customers

GET /api/Customers
POST /api/Customers
GET /api/Customers/{id}
PUT /api/Customers/{id}
DELETE /api/Customers/{id}

Drivers

GET /api/Drivers
POST /api/Drivers
GET /api/Drivers/{id}
PUT /api/Drivers/{id}
DELETE /api/Drivers/{id}
PATCH /api/Drivers/UpdateLocations/{id}

Trips

GET /api/Trips
GET /api/Trips/{id}
DELETE /api/Trips/{id}
GET /api/Trips/{id}/SubtripsOnly
POST /api/Trips/AddCustomer/{customerId}
DELETE /api/Trips/DeleteCustomer/{customerId}
DELETE /api/Trips/Subtrips/{id}

Рисунок 2.5 – Перша версія документації API

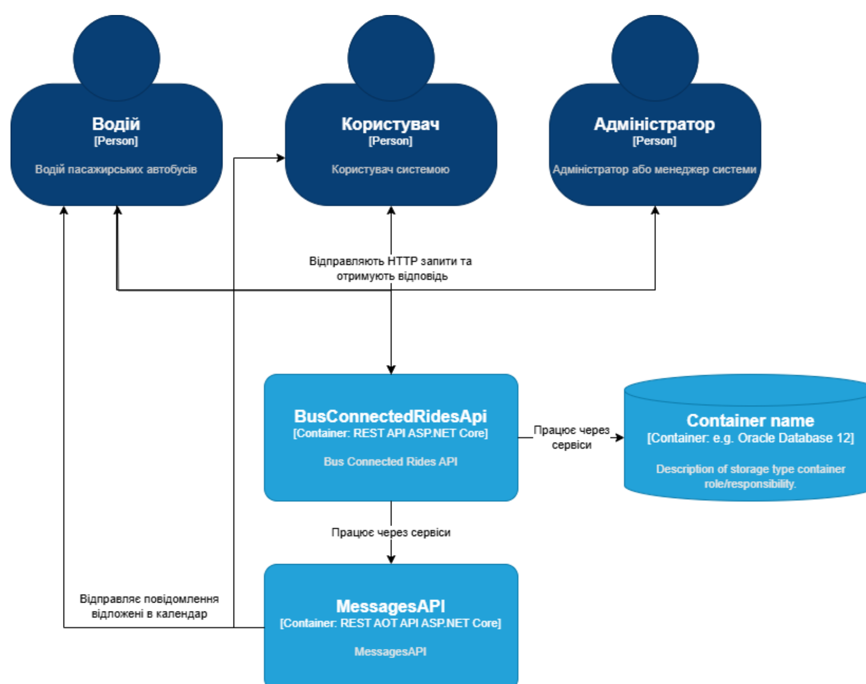


Рисунок 2.6 – Попередня архітектура, описана за допомогою діаграми контексту (C4-модель)

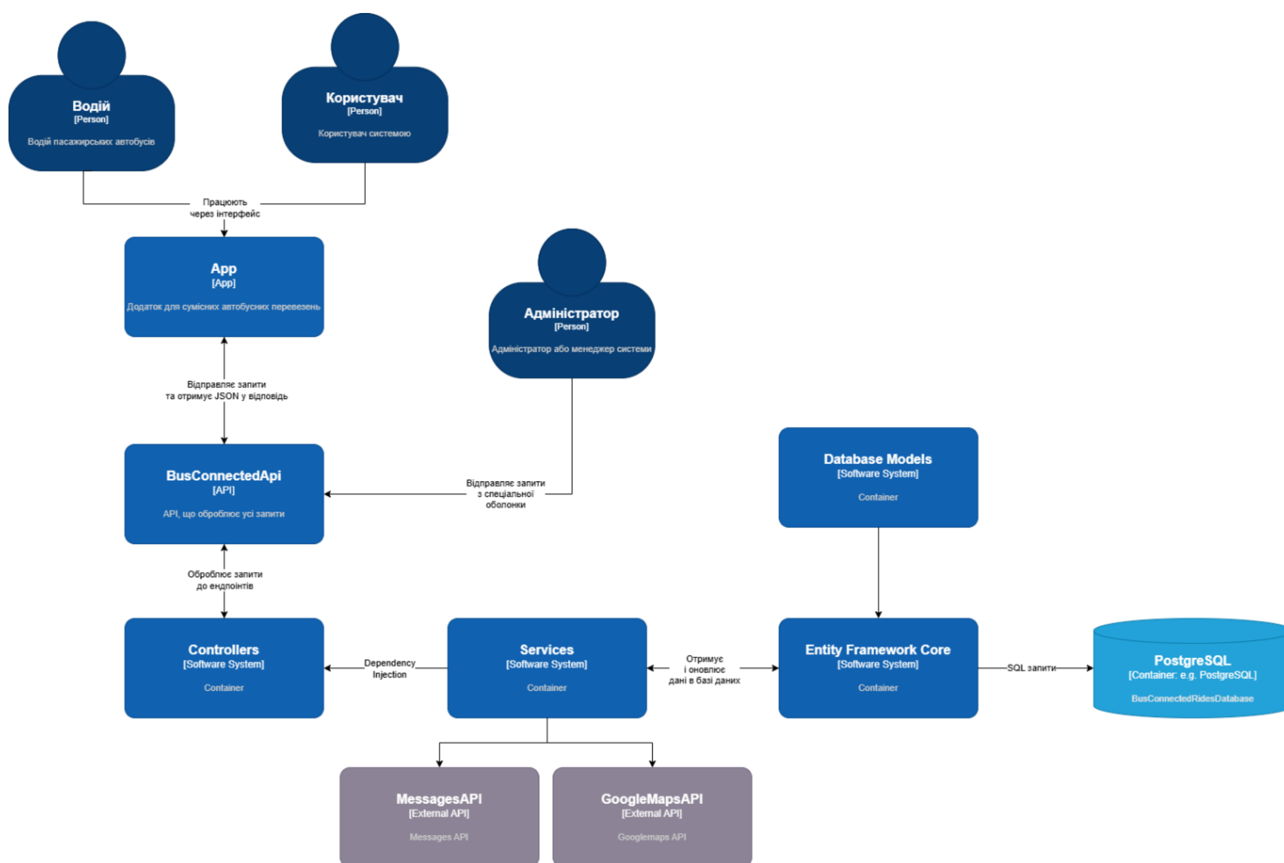


Рисунок 2.7 – Попередня архітектура, описана за допомогою діаграми контейнерів (C4-модель)

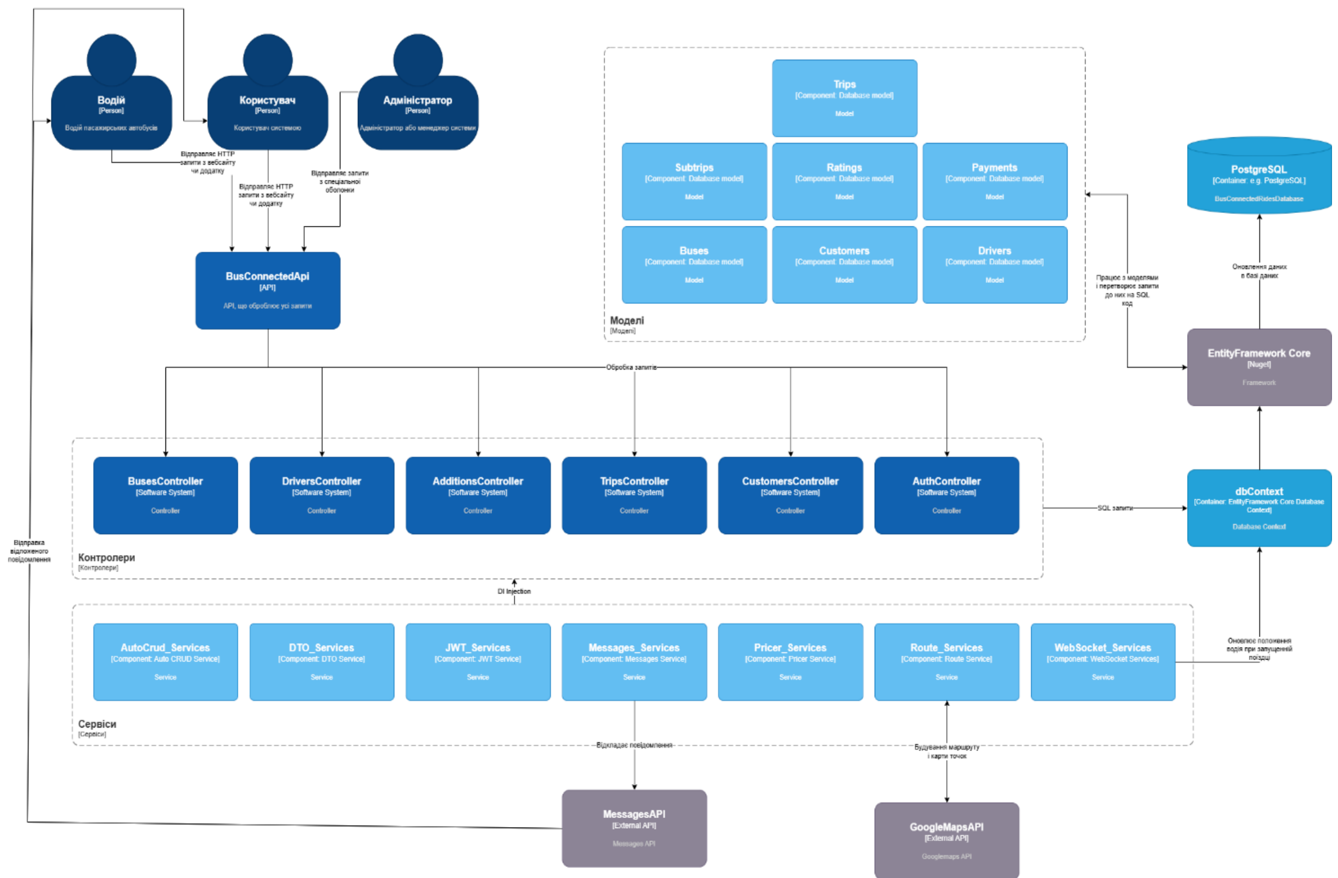


Рисунок 2.8 – Попередня архітектура, описана за допомогою діаграми компонентів (С4-модель)

Наступна діаграма – це діаграма класів (рис. 2.9) для сервісів, що опрацьовують бізнес-логіку додатку та під'єднуються через Dependency Injection до контролерів. Прикладом реалізації є AutoCrud_Service, що автоматично працює з простими CRUD запитами динамічно будуючи їх за типом об'єкту що передається до методів через параметри. Маємо п'ять відкритих методів для видалення, оновлення, вставки, повного отримання даних та отримання окремих об'єктів за ідентифікатором відповідно. Один приватний метод займається отриманням залежностей об'єкта за типом. У зв'язку зі специфікою сервісу, методи працюють з базою даних завдяки DbContext. Методи отримання об'єктів з бази даних автоматично підтягують залежності сутності з бази даних через приватний метод GetIncludedQuerry().

Рисунок 2.9 – Попередня архітектура сервісів, описана за допомогою діаграми класів (С4-модель)

Загалом, система спроєктована з розрахунком на масштабування та реальну експлуатацію в промислових умовах. Її можна інтегрувати з іншими сервісами: оплатою, картами, повідомленнями. При цьому вона залишається зрозумілою, підтримуваною та гнучкою.

2.3 Детальне проєктування та подробиці реалізації програмного забезпечення

Архітектура застосунку побудована за клієнт-серверним принципом, також всередині програмної системи використовується велика кількість зв’язків та підсистем. Запити оброблюють контролери, а бізнес-логіку реалізують сервіси. Замість складної логіки усі дані передаються далі в сервіси, в яких відбувається вся основна логіка, обробка даних, звернення до бази, фільтрація, перевірки, формування відповідей. Назад із сервісу повертається готовий результат у вигляді DTO або іншої форми, яка потрібна користувачу [28].

Сервіси, які вирішують основні задачі мають більш складну структуру. У той же час, присутні й інші, маленькі та допоміжні. Вони конвертують дані,

формують параметри, роблять обчислення. Цей розподіл дозволяє не втрачати логіку в ході розроблення додатку та розподіляти відповідальності між сервісами [28].

У результаті виходить модульна, розділена за ролями система з чітко інкапсульованою логікою, яку легко підтримувати, змінювати й розширювати без ризику отримати критичну помилку при використані. Реєстрація сервісів у DI-контейнері відбувається за допомогою уривку коду з лістингу 2.1.

Лістинг 2.1 – Додавання сервісів у DI-контейнер

```
builder.Services.AddTransient<IDTO_Service, DTO_Service>();
builder.Services.AddScoped<IAutoCrud_Service, AutoCrud_Service>();
builder.Services.AddScoped<IRouteBuilder_Service, RouteBuilder_Service>();
builder.Services.AddScoped<IMessages_Service, Messages_Service>();
builder.Services.AddScoped<IJwtToken_Service, JwtToken_Service>();

// Depricated services
builder.Services.AddTransient<ITo_JSON_Service, ToJSON_Service>();
builder.Services.AddAutoMapper(typeof(MappingProfile));
var app = builder.Build();
```

Щоб не повторювати подібні дії в кожному контролері, було створено універсальний CRUD-сервіс. Він реалізує основну роботу простих CRUD-запитів: створення, читання, оновлення й видалення даних. Відповідна програмна реалізація подана в додатку А. Усі контролери працюють із цим сервісом через загальний інтерфейс, що не лише скорочує кількість коду, а й дає змогу швидко вносити зміни. У результаті прості запити не перевантажують систему, а складні сценарії можна легко надбудовувати зверху, не чіпаючи програмну основу. Завдяки цьому підходу зменшується кількість помилок, легше писати тести, і вся система стає стабільнішою.

Окремо було реалізовано MessagesAPI – це окремий мікросервіс, який відповідає за роботу з повідомленнями. Його завдання полягає в збереженні повідомлень, розсиланні їх користувачам і прив’язуванні до календарних подій. Замість того, щоб розробляти цю логіку в основній API, було винесено її в окремий компонент. Це дозволяє зберігати чисту архітектуру, де кожна частина системи відповідає лише за свою сферу.

Щоб зв’язатися з цим API, основна система використовує окремий сервіс, який відправляє HTTP-запити. Такий підхід дає змогу взаємодіяти з повідомленнями як із зовнішнім сервісом, без прямого втручання в його внутрішню логіку. Завдяки цьому забезпечується гнучкість і незалежність модулів: якщо в майбутньому з’явиться нова система повідомлень, її можна буде підключити без перебудови всієї платформи. Файлова структура сервісу MessagesAPI наведена на рис. 2.10, а базова імплементація відправки запитів до цього сервісу продемонстрована в лістингу 2.2.

Лістинг 2.2 – Відправка запитів до MessagesAPI

```
public class Messages_Service : IMessages_Service
{
    private readonly ILogger<Messages_Service> _logger;
    private readonly HttpClient _httpClient;

    public Messages_Service(ILogger<Messages_Service> logger, IHttpClientFactory
httpClientFactory)
    {
        _logger = logger;
        _httpClient = httpClientFactory.CreateClient("MessagesApi");
    }

    public async Task ScheduleMessage(string message, string reciver, int
messageType, DateTime schedule)
    {
        _httpClient.DefaultRequestHeaders.Accept.Clear();
        _httpClient.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));

        var messageData = new
        {
            MessageContent = message,
            MessageReciver = reciver,
            MessageType = messageType,
            DelieverTime = schedule
        };

        var response = await
_httpClient.PostAsJsonAsync("/api/message/QueueMessage", message);
        response.EnsureSuccessStatusCode();
    }
}
```

У межах розробленої системи створено компонент RouteBuilderService – це окремий сервіс, який відповідає за формування маршрутів для спільних поїздок. Його головна задача полягає в прокладанні шляху з урахуванням потреб як пасажирів, так і водіїв. Сервіс приймає на вхід точки відправлення і

призначення, поточне місцезнаходження транспорту, і намагається побудувати маршрут так, щоб нові пасажири могли приєднатися до вже існуючої поїздки без істотного відхилення від курсу.

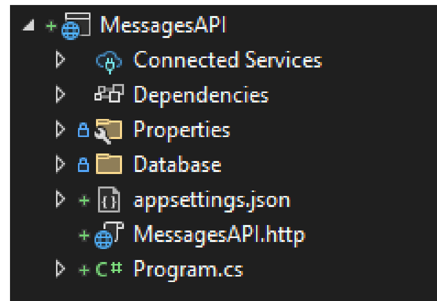


Рисунок 2.10 – Структура MessagesAPI

В основі логіки лежить аналіз координат та оцінка допустимого відхилення. Якщо новий запит вкладається в ці межі, він додається у вигляді проміжного сегмента – субшляху. Після цього маршрут перебудовується з урахуванням нових зупинок, і оптимізується черговість, щоб уникнути зайвих затримок та кілометражу. Для обчислень використовується інтеграція з Google Maps API. Завдяки ній враховуються реальні умови на дорогах: затори, час у дорозі, стан трафіку.

Сервіс розрахунку вартості поїздки працює за власним алгоритмом, що показано в лістингу 2.3 за допомогою псевдокоду. В ціну закладаються параметри: тариф за кілометр, тариф за хвилину, відстань подорожі, приблизна тривалість, відхилення від основного маршруту через нову точку, відсоток комісії.

Тарифи виставляє водій, комісію – компанія, що надає послуги завдяки API. Приблизна тривалість поїздки обчислюється ще до початку дороги завдяки відстані подорожі і основним характеристикам, що надає Google Maps API. Відхилення також рахується через порівняння попереднього й нового маршруту.

Схема бази даних представлена за допомогою ER-діаграми на рис. 2.11 та має такі таблиці: (1) Customers; (2) Payments; (3) Rating; (4) Trips; (5) Subtrips; (6) Drivers; (7) Buses; (8) Auth.

Лістинг 2.3 – Псевдокод алгоритму

```
ввід:
      ТарифЗаКілометр
      ТарифЗаХвилину
      ВідстаньКм
      ТривалістьХв
      ВідхиленняМаршруту
      ВідсотокКомісії
Обчислення:
      БазоваЦіна = (тарифЗаКілометр * відстаньКм) + (тарифЗаХвилину *
      тривалістьХв)
      КоефВідхилення = 1 + (відхиленняМаршруту / 100)
      КоефКомісії = 1 + (відсотокКомісіїі / 100)
      ЦінаБезКомісії = базоваЦіна * коефВідхилення
      Комісія = цінаБезКомісії * (відсотокКомісії / 100)
      ОстаточнаЦіна = цінаБезКомісії + комісія
Вивід:
      ОстаточнаЦіна
```

Таблиця Customers (Користувачі) включає поля: ідентифікатор користувача, ім’я, електрону пошту та час створення. Таблиця Payments (Оплата) включає поля з ідентифікатором платежу, ідентифікатором маршруту, методом оплати, обсягом платежу та часом здійснення. Таблиця Rating (Рейтинг) включає поля: ідентифікатор рейтингового запису, ідентифікатор користувача, ідентифікатор водія, ідентифікатор маршруту, рейтинг від користувача, рейтинг від водія та час створення. Таблиця Trips (Маршрути) містить поля: ідентифікатор, ідентифікатори користувачів, ідентифікатор водія, ідентифікатори підмаршрутів, статус поїздки і час створення. Таблиця Subtrips (Підмаршрути) пропонує такі поля: ідентифікатор, ідентифікатор користувача, ідентифікатор маршруту, ідентифікатор оплати, статус підмаршруту, точка відправки, та прибуття. Таблиця Drivers (Водії) передбачає такі поля: ідентифікатор, ім’я, ідентифікатори зареєстрованих автобусів, електрону пошту, дату народження, поточну локацію, час створення. Таблиця Buses (Автобуси) вводить поля: ідентифікатор, ідентифікатор водія, тип автобуса

та реєстраційний номер. Таблиця Auth містить ідентифікатор, логін, пароль та прапорець «Чи онлайн?».

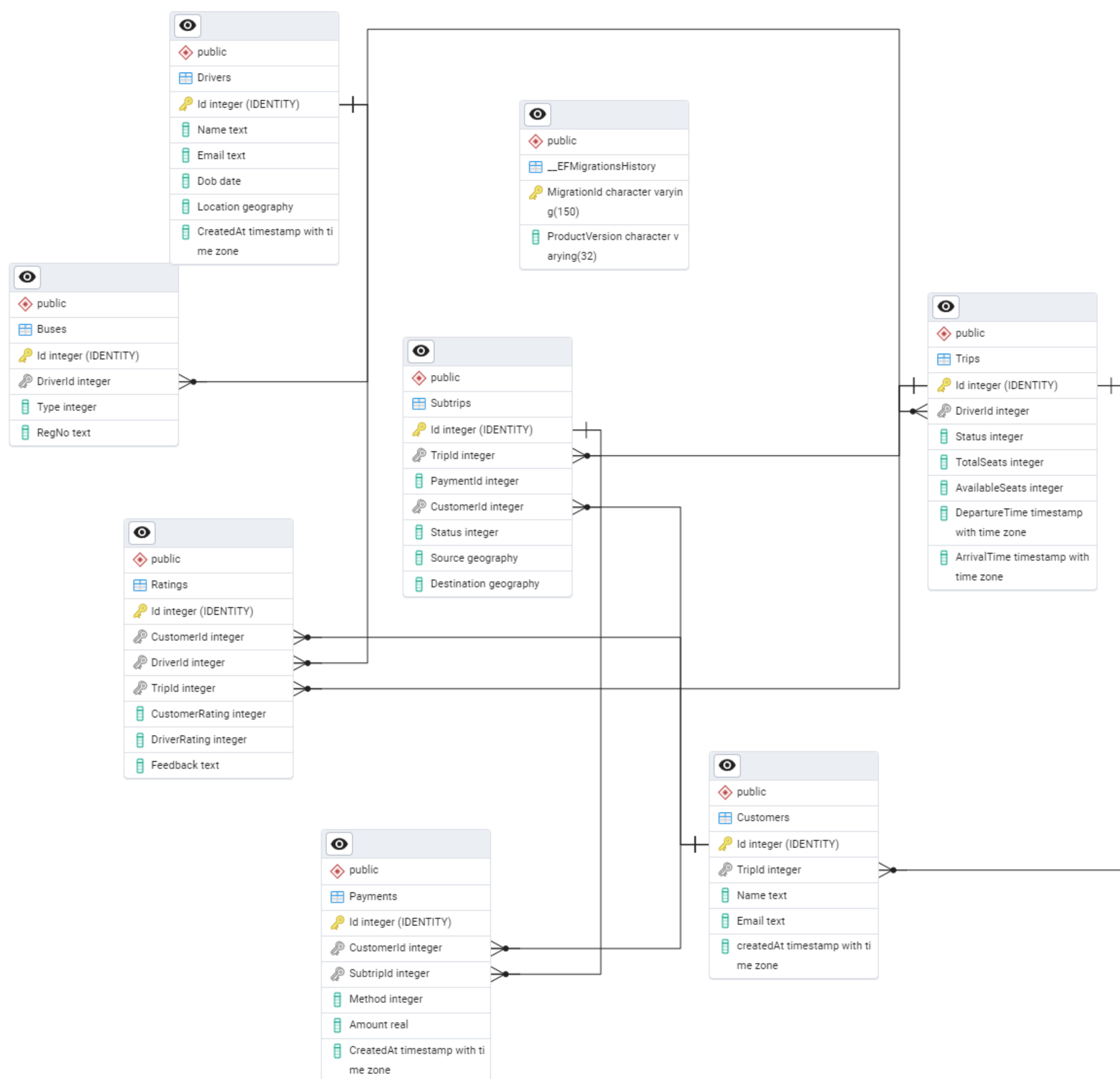


Рисунок 2.11 – ER-діаграма побудованої бази даних

У базі даних, крім основних таблиць, що відповідають за користувачів, поїздки, автобуси тощо, зберігаються також службові таблиці, які створює фреймворк Entity Framework Core. Наприклад, там є таблиця з історією міграцій в контексті бази даних, які вносились у структуру бази під час розробки. Ще є таблиця spatial_ref_sys, яка використовується для роботи з геолокацією.

База даних побудована за підходом Code-First. Спочатку були створені класи у C#, які описують усю інформацію щодо сутностей у базі даних, а вже потім відбувалась її автоматичне генерування на основі цього коду. Загальний клас DbContext оперує структурою бази та дозволяє працювати з нею через код за допомогою LINQ-запитів.

База даних була перевірена на відповідність третій нормальній формі, тобто відповідає і першій, і другій НФ включно. Усі значення атомарні, відсутні повторювані стовбці, всі неключові атрибути повністю функціонально залежні від первинного ключа, і всі неключові атрибути не залежать транзитивно від первинного ключа.

У застосунку є п’ять основних контролерів, кожен з яких відповідає за свою частину системи та відображає бізнес-логіку роботи з відповідними сутностями (таблицями або групами таблиць у базі даних). Така структура дає змогу розмежовувати відповідальності за допомогою контролерів:

1) AdditionsController працює з допоміжними таблицями на кшталт Payments і Ratings. Базова логіка в поточній редакції досить прямолінійна, проте відкрита для розширення, наприклад, для обчислення бонусів тощо;

2) BusesController передбачає, що кожен запис інформації до таблиці Buses включає інформацію, за ким із водіїв він закріплений, а власне контролер дає змогу додавати, редагувати, видаляти чи переглядати ці записи;

3) CustomersController відповідає за оновлення профілю, перегляд поїздок; через нього проходить усе, що стосується взаємодії звичайного користувача із системою;

4) у DriversController зберігається уся інформація про профілі, статуси, а також відображається зв’язок водіїв з автобусами й іншими таблицями;

5) TripsController відповідає за поїздки: як основні маршрути, так і проміжні субшляхи, які додаються під час планування. Саме тут логіка побудови маршруту, оптимізації зупинок і фіксації поїздки в базі виконується через сервіси бізнес логіки.

Для того, щоб не витягувати зайву інформацію в кожному запиті, були реалізовані окремі класи для обміну даними між клієнтом і сервером у вигляді DTO та ShortDTO. DTO-об’єкти несуть повну інформацію про сутність в базі даних, наприклад, всю інформацію про поїздку чи користувача, включаючи усі зв’язки вглиб. ShortDTO – це скорочені версії тих же об’єктів, їх зручно використовувати, коли потрібно просто показати перелік без зайвих деталей.

Перетворення з однієї моделі в іншу реалізовано через AutoMapper (програмна реалізація наведена в додатку Б). Це бібліотека, яка дає змогу автоматично перетворювати об’єкти одного типу в інші. Для цього було створено спеціальний клас з прописаними правилами, як моделі мають перетворюватися в DTO, й навпаки [29]. Типова реалізація DTO-об’єкта відображена в лістингу 2.4. Файлова структура з відповідними класами представлена на рис. 2.12.

Лістинг 2.4 – Типова реалізація DTO в проєкті

```
public class DriversDTO
{
    public string? Name { get; set; }
    public string? Email { get; set; }
    public DateOnly Dob { get; set; }

    public float Latitude { get; set; }
    public float Longitude { get; set; }

    public BusesShortDTO? BusDTO { get; set; }
    public IEnumerable<TripsShortDTO>? Trip { get; set; }
}

public class DriversShortDTO
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public string? Email { get; set; }
    public DateOnly Dob { get; set; }

    public float Latitude { get; set; }
    public float Longitude { get; set; }
}
```

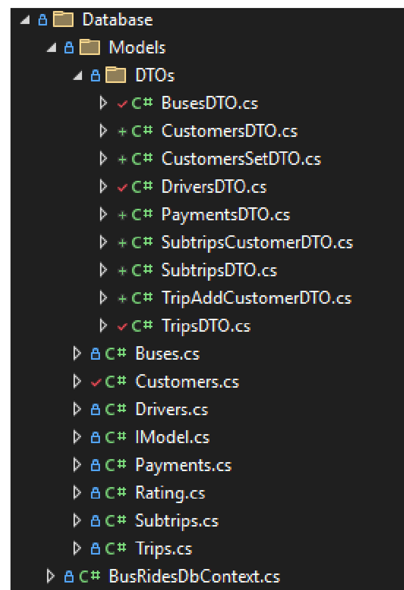


Рисунок 2.12 – Структура DTO і класів для побудови бази даних та їх опрацювання

Лістинг 2.5 відображає уривок коду, що стосується системи парсингу більш складних DTO-моделей. Вони потребують додаткових параметрів при створені.

Лістинг 2.5 – Система парсингу більш складних DTO-моделей

```
public IModel? ParseDTO(object dto, Type modelType)
{
    var model = Activator.CreateInstance(modelType);

    if (model == null || dto == null ||
!modelType.GetInterfaces().Contains(typeof(IModel)))
    {
        return null;
    }

    foreach (var property in modelType.GetProperties())
    {
        var modelProperty = dto.GetType().GetProperty(property.Name);
        if (modelProperty == null) continue;

        if (modelProperty.Name != "Id")
        {
            property.SetValue(model, modelProperty.GetValue(dto));
        }
        else if(modelProperty.Name == "createdAt")
        {
            property.SetValue(model, DateTime.Now);
        }
    }

    return model as IModel;
}
```

Відстеження водія під час поїздки відбувається завдяки WebSocket, що дає змогу без великого навантаження обробляти поточну геолокацію водія й передавати її користувачам для відстеження маршруту в реальному часі. Відповідний уривок коду продемонстровано в лістингу 2.6.

Лістинг 2.6 – Обробка WebSocket

```
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            var webSocket = await context.WebSockets.AcceptWebSocketAsync();

            var buffer = new byte[1024];
            while (true)
            {
                var result = await webSocket.ReceiveAsync(new
ArraySegment<byte>(buffer), CancellationToken.None);
                if (result.CloseStatus.HasValue)
                {
                    await webSocket.CloseAsync(result.CloseStatus.Value,
result.CloseStatusDescription, CancellationToken.None);
                    break;
                }

                // Process the received message

            }
        }
        else
        {
            context.Response.StatusCode = 400;
        }
    }
    else
    {
        await next();
    }
});
```

У системі також реалізовано авторизацію через JWT (JSON Web Token). Спрощено процес виглядає так: користувач надсилає свої облікові дані, вони потрапляють до спеціального контролера, який перевіряє, чи є дані в базі даних. Якщо все збігається, система формує токен і повертає його клієнту. Кожен новий

запит містить цей токен у заголовку (в полі Authorization), і сервер, отримавши його, перевіряє, чи дійсний він. Відповідні уривки коду продемонстровано в лістингах 2.7 і 2.8.

Лістинг 2.7 – Налаштування JWT-токенів і бар'єру

```
builder.Services.Configure<JwtSettings>(builder.Configuration
                                               .GetSection("JwtSettings"));

var jwtSettings =
builder.Configuration.GetSection("JwtSettings").Get<JwtSettings>();
var key = Encoding.ASCII.GetBytes(jwtSettings.Secret);

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,

        ValidIssuer = jwtSettings.Issuer,
        ValidAudience = jwtSettings.Audience,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ClockSkew = TimeSpan.Zero
    };
});
```

Висновки до другого розділу

Отже, в ході реалізації було дотримано принципів SOLID і DRY, компоненти чітко розмежовані: окремі сервіси, окремі контролери, окремі DTO. Залежності впроваджені через DI-контейнер, таким чином завантажується лише те, що виконується в контексті виконання логіки запиту.

Проєкт побудований, за стандартними принципами, які використовують у вебархітектурі. Код розділено по рівнях (папках проєктів, рис. 2.12) за ролями. Такий поділ дає змогу легше розбиратися в коді, підтримувати його в майбутньому, і, загалом, система виходить більш масштабованою.

Лістинг 2.8 – JWT-сервіс

```
public class JwtToken_Service : IJwtToken_Service
{
    private readonly JwtSettings _jwtSettings;

    public JwtToken_Service(IOptions<JwtSettings> jwtSettings)
    {
        _jwtSettings = jwtSettings.Value;
    }

    public string GenerateToken(string userId, string userEmail)
    {
        var claims = new[] {
        new Claim(JwtRegisteredClaimNames.Sub, userId),
        new Claim(JwtRegisteredClaimNames.Email, userEmail),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

        var key = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(_jwtSettings.Secret));
        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken(
            issuer: _jwtSettings.Issuer,
            audience: _jwtSettings.Audience,
            claims: claims,
            expires: DateTime.UtcNow.AddMinutes(_jwtSettings.ExpiryMinutes),
            signingCredentials: creds);

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}
```

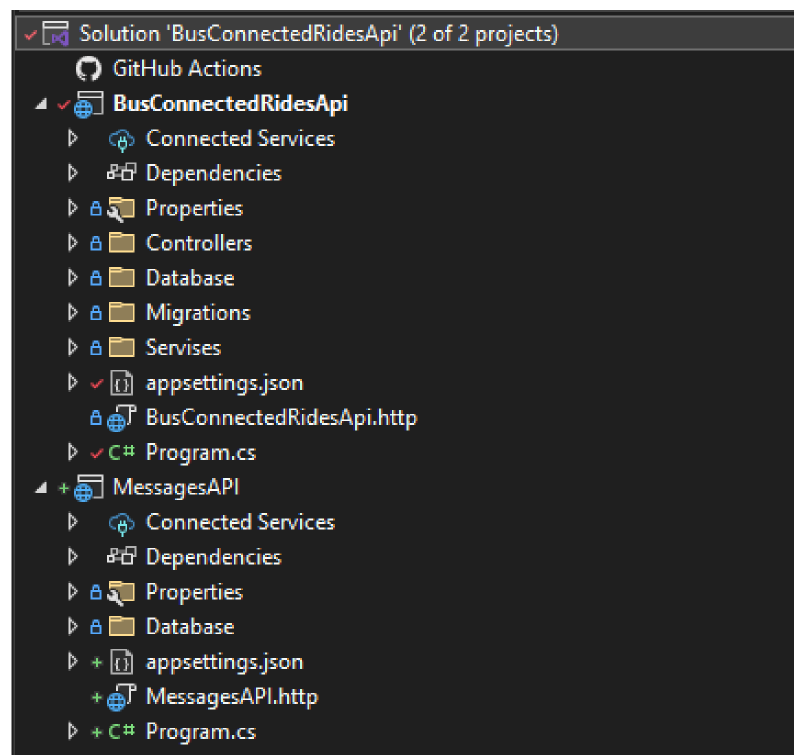


Рисунок 2.12 – Повна файлова структура проєкту

Крім базової логіки, додано кілька важливих речей:

– Перевірка даних. Валідація відбувається як у DTO, так і в контролерах. Для цього використовуються стандартні атрибути типу [Required], [MaxLength] тощо. Це дозволяє відловлювати неправильні значення ще до обробки.

– Логування. Для відстеження подій задіяно вбудований механізм ILogger. Через нього журналюються основні дії й помилки під час виконання, що зручно при налагодженні.

– Асинхронність. Робота з базою даних та зовнішніми API зроблена засобами async/await. Як відомо, це дає змогу не блокувати потік і працювати з ресурсами ефективніше, особливо під навантаженням.

– Документація. Підключено Swagger UI, через який можна дивитись опис ендпоінтів і тестувати запити прямо з браузера, що пришвидшує перевірку.

РОЗДІЛ 3
ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Вибір методів та інструментів тестування

З метою забезпечення стабільної роботи програмного рішення на різних етапах і рівнях у ході розроблення було застосовано як ручне, так і автоматизоване тестування. Ним охоплено реалізацію як і бізнес логіки, так і залежностей між частинами системи. Підхід був орієнтований на типову класифікацію, яка включає кілька важливих типів перевірок: (1) модульне тестування; (2) інтеграційне тестування; (3) функціональне тестування; (4) навантажувальне тестування [30]. Відповідність між використаними програмними інструментами та методами тестування відображена в таблиці 3.1.

Таблиця 3.1 – Використані інструменти для різних типів тестування

Рівень тестування	Інструменти
Модульне тестування	XUnit, EF Core InMemory
Інтеграційне тестування	EF Core InMemory
Функціональне тестування та тестування API	Postman, Swagger

Для модульного тестування було обрано фреймворк xUnit. У контексті платформи .NET це один із найзручніших варіантів, особливо з урахуванням інтеграції з ASP.NET Core. Фреймворк не потребує значних зусиль для конфігурування та підтримує інверсію залежностей, що спрощує роботу при ізоляції окремих компонентів. Тести можна запускати паралельно, що помітно економить час у великих проєктах. Це дало змогу перевіряти окремі фрагменти логіки (без бази даних, сторонніх API тощо) і зосередитись саме на поведінці ізольованих уривків коду [31].

Інтеграційне тестування реалізоване через EF Core InMemory. Цей підхід дає змогу перевіряти роботу з даними без реальної БД, що помітно спрощує

роботу, оскільки відсутня потреба піднімати окремий екземпляр PostgreSQL з тестовими значеннями. Такий варіант доречний для перевірки складних LINQ-запитів, правильності міграцій та взаємодії сервісів з контролерами [32].

Тестування REST API відбуватиметься за допомогою програмного інструменту Postman. У ньому можна зберігати запити в колекції, легко запускати їх повторно, працювати з JWT-токенами, що використовуються в системі для авторизації, а також перевіряти, як API поводиться в різних ситуаціях. Крім того, використовується сервіс Swagger, який автоматично формує документацію. За його допомоги теж можна здійснювати тестування прямо з браузера, без зайвого коду чи середовищ [33].

Оскільки стабільність роботи при навантаженні є критично важливим аспектом у сучасних умовах, була передбачена можливість запуску навантажувальних тестів. Для цього використовуються програмні інструменти k6 та Grafana, що даватиме змогу симулювати високі обсяги трафіку та зрозуміти, як поводиться API під навантаженням запитами.

Покриття тестами стосується основних бізнесових сценаріїв. Зокрема, тестувалася маршрутизація, перевірка введених даних, обчислення вартості поїздки, а також обробка запитів загалом. У процесі були враховані як позитивні кейси, так і ситуації, де щось може піти не так, проте система має правильно зреагувати. Загалом, кожен інструмент було підібрано з урахуванням технічних аспектів реалізації (платформа ASP.NET Core) та необхідності тестувати REST API.

3.2 Побудова тестового плану

Тестовий план охоплює обсяги, підходи, інструменти й загальну логіку перевірки, щоб забезпечити повну, а головне, стабільну роботу додатку. У проєкті процес тестування будувався з урахуванням того, що система має багаторівневу архітектуру на базі технологічного стеку ASP.NET Core, а також працює з REST API, маршрутизацією, обробкою повідомлень.

Передусім, здійснювалось тестування роботи сервісів бізнес-логіки та їх елементів: AutoCrud_Service, DTO_Service, Messages_Service, Routes_Service, Pricer_Service. Крім того, тестуванню підлягають контролери REST API (наприклад, TripsController, DriversController, CustomersController), компоненти з доступом до даних через Entity Framework, структури DTO та окремо WebSocket-модуль, який відповідає за передачу геолокації у режимі реального часу.

Тестування проводилося поетапно, як відображено в таблиці 3.2. Приймальними критеріями, в цілому, стали такі:

– всі критичні функції повинні проходити тести без помилок;

– система повинна обробляти 90% запитів зі стабільним часом відповіді < 1 секунди під навантаженням;

– усі негативні сценарії мають коректно оброблятися без збоїв.

Завдяки xUnit вдалося відтестувати, обчислення вартості поїздки, створення бронювань, маршрути та збереження повідомлень. Особливу увагу приділено різним вхідним даним, включно з помилковими, щоб побачити, як система поводиться в нестандартних ситуаціях. Відповідні тестові сценарії в контексті модульного тестування зібрано в таблиці 3.3.

Таблиця 3.2 – План тестування за етапами

Етап	Опис робіт	Очікуваний результат
1. Модульне тестування	Тестування логіки сервісів ізольовано	Перевірка функціональності бізнес-логіки
2. Інтеграційне тестування	Перевірка взаємодії сервісів із БД та контролерами	Коректна робота зв’язків та моделі даних
3. Функціональне тестування	Перевірка API за основними сценаріями користувача	Очікувана поведінка при типових запитах
4. Навантажувальне тестування	Перевірка поведінки системи під трафіком	Підтвердження стабільної роботи під навантаженням

Таблиця 3.3 – Тестові сценарії для модульних тестів

№	Назва тесту	Мета тестування	Вхідні дані / Умови	Очікуваний результат	Короткий опис процедури тестування
1	GetByIdAsync_ReturnsEntity_WhenExists	Перевірити, що метод повертає існуючий об’єкт за ID	Об’єкт Buses з Id=1, доданий у базу	Метод повертає об’єкт з полем RegNo="ABC123"	Додати об’єкт у БД, викликати GetByIdAsync(1), перевірити, що результат не null і має правильне значення RegNo
2	GetByIdAsync_ReturnsNull_WhenNotExists	Перевірити, що метод повертає null для неіснуючого ID	ID = 99, в базі немає відповідного об’єкта	Метод повертає null	Викликати GetByIdAsync(99) на порожній базі, переконатися, що результат null
3	AddAsync_AddsEntity	Перевірити додавання нового об’єкта у базу	Новий об’єкт Buses з Id=1, RegNo="XYZ"	Об’єкт додано, в таблиці 1 запис, RegNo="XYZ"	Викликати AddAsync(bus), перевірити наявність об’єкта у таблиці
4	UpdateAsync_UpdatesEntity	Перевірити оновлення існуючого об’єкта	Існуючий об’єкт Buses з RegNo="OLD" змінено на RegNo="NEW"	Поле RegNo оновлено до "NEW"	Додати об’єкт, змінити поле, викликати UpdateAsync(bus), перевірити оновлення
5	DeleteAsync_RemovesEntity_WhenExists	Перевірити видалення існуючого об’єкта	Існуючий об’єкт Buses з Id=1	Об’єкт видалено, таблиця порожня	Додати об’єкт, викликати DeleteAsync<Buses>(1), перевірити, що таблиця порожня
6	DeleteAsync_DoesNothing_WhenNotExists	Перевірити, що метод не змінює базу при спробі видалити неіснуючий об’єкт	ID = 99, об’єкт відсутній	Таблиця залишається порожньою	Викликати DeleteAsync<Buses>(99) на порожній базі, переконатися у відсутності змін
7	ParseDTO_ReturnsNull_IfNotIModel	Перевірити, що ParseDTO повертає null для типів, які не реалізують інтерфейс IModel	DTO — анонімний об’єкт, тип — string (не імплементує IModel)	Метод повертає null	Викликати ParseDTO(dto, typeof(string)), перевірити, що повертається null
8	GenerateToken_ReturnsTokenString	Перевірити генерацію JWT токена з правильним форматом	Налаштування JWT, ім’я користувача, емейл	Метод повертає непорожній JWT токен із крапками	Ініціалізувати JwtToken_Service, викликати GenerateToken, перевірити формат та наявність токена
9	ScheduleMessage_SendsPostRequest	Перевірити відправлення POST-запиту для планування повідомлення	Повідомлення, емейл, кількість, дата	Відправлено POST-запит на правильний URL	Мокування HTTP, виклик ScheduleMessage, перевірка виклику SendAsync із POST
10	CountPrice_ReturnsExpectedValue	Перевірити правильність обчислення вартості	Параметри ціни)	Повертається очікуване числове значення	Викликати countPrice із параметрами, перевірити правильність результату
11	RouteBuild_ReturnsTrue	Перевірити, що метод побудови маршруту повертає успіх	Координати початку і кінця маршруту	Метод повертає true	Викликати RouteBuild із координатами, перевірити, що результат — true
12	AddCustomerToTrip_CompletesSuccessfully	Перевірити, що додавання клієнта до поїздки проходить без винятків	Об’єкти Trips і Customers з ідентифікаторами, координати, ціна	Метод успішно завершує роботу	Викликати AddCustomerToTrip асинхронно, перевірити відсутність винятків

У лістингу 3.1 наведено приклад побудови тесту завдяки фреймворку xUnit та провайдеру EF Core InMemory db. Додатково в лістингу 3.2 показано приклад побудови параметризованого тесту на базі теореми xUnit і пакету AspNetCore.Mvc.Testing.

Лістинг 3.1 – Приклад тесту на базі xUnit та EF Core InMemory db

```
    [Fact]
    public async Task GetByIdAsync_ReturnsEntity_WhenExists()
    {
        var dbName = nameof(GetByIdAsync_ReturnsEntity_WhenExists);
        using var context = GetDbContext(dbName);
        var bus = new Buses { Id = 1, DriverId = 1, Type = BusType.Mini, RegNo =
"ABC123" };
        context.Buses.Add(bus);
        context.SaveChanges();

        var service = new AutoCrud_Service(context);
        var result = await service.GetByIdAsync<Buses>(1);

        Assert.NotNull(result);
        Assert.Equal("ABC123", result.RegNo);
    }
```

Лістинг 3.2 – Приклад теореми xUnit та використання пакету AspNetCore.Mvc.Testing

```
    [Theory]
    [InlineData("Buses")]
    [InlineData("Drivers")]
    [InlineData("Customers")]
    public async Task Get_Endpoint_ReturnsSuccess(string endpoint)
    {
        var client = _factory.CreateClient();
        var response = await client.GetAsync($"/api/{endpoint}");
        Assert.Equal(HttpStatusCode.OK, response.StatusCode);
    }
```

З інтеграційними тестами завданням було перевірити взаємодію між компонентами: чи правильно обробляються запити, чи коректно працює відображення моделей у DTO та чи коректно відбуваються міграції. Для цього використовувався інструмент EF Core InMemory, який надав можливість обійтися без справжньої бази даних і водночас створити тестове середовище, близьке до продакшну.

У контексті функціонального тестування здійснювалось тестування REST API засобами Postman. Було створено набір запитів, які імітують роботу

кінцевого користувача: логін, створення поїздки, пошук маршруту, бронювання, надсилання повідомлень. Окремо перевірялися кейси з некоректними запитами, відсутніми параметрами або спробами доступу без авторизації. Відповідні тестові сценарії відображено в таблиці 3.4. Разом з тим, як допоміжний інструмент під час розроблення використовувався Swagger: він був зручний для ручних перевірок, коли налагоджуються контролери.

Таблиця 3.4 – Тестові сценарії функціонального тестування

№	Назва тесту	Мета тестування	Вхідні дані / Умови	Очікуваний результат	Короткий опис процедури тестування
1	AuthController_Login_ReturnsJwtToken	Перевірити успішну авторизацію користувача та отримання JWT токена	Email: admin@example.com, Password: password	HTTP 200 OK, відповідь містить дійсний JWT токен	Викликати POST /api/auth/login з тіла з email/паролем, перевірити статус, токен, тип контенту
2	AuthController_Login_TokenStoredInEnv	Перевірити, що токен збережено у змінну середовища Postman	Відповідь з полем token	Значення токена збігається із збереженим у JWTtoken	Після логіну зчитати токен з відповіді, зберегти у змінну середовища і порівняти
3	AuthController_Login_ResponseFormatValid	Перевірити коректний формат відповіді авторизації	Запит на логін	Content-Type: application/json	Перевірити заголовок Content-Type у відповіді
4	AuthController_Login_ResponseTimeFast	Перевірити швидкодію авторизації	Запит на логін	Час відповіді < 200 мс	Перевірити, що час відповіді не перевищує 200 мілісекунд
5	BusesController_GetAll_ReturnsJson	Перевірити отримання переліку автобусів у форматі JSON	JWT токен у заголовку авторизації	Content-Type: application/json	Викликати GET /api/Buses з токеном, перевірити заголовок відповіді

За допомогою інструменту k6 відбулося моделювання багатьох одночасних запитів до API. Це дало змогу побачити, як сервер поводиться під навантаженням: час відповіді, де починаються просідання, як поводиться пам’ять тощо. Так виявлялися вузькі місця й перевірялася стійкість системи до пікового трафіку.

3.3 Проведення та результати тестування розробленого програмного рішення

Тестування проводилось локально, для кожного з наведених у тестовому планів кейсів. Усі тести, окрім Postman і k6, були побудовані на основі xUnit і бібліотек, що були згадані в минулому пункті. На рисунку 3.1 відображено файлову структуру реалізованих модульних тестів.

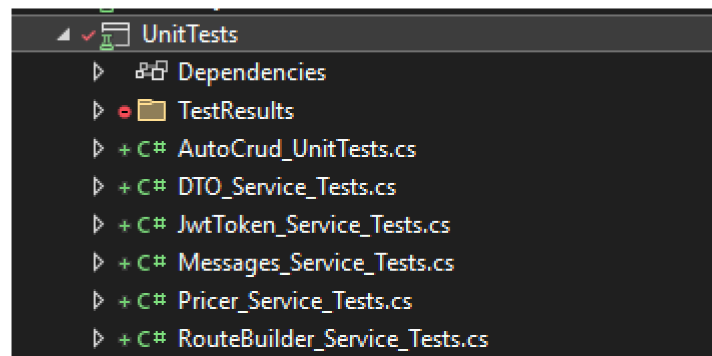


Рисунок 3.1 – Файлова структура модульних тестів

Під час реалізації тестів було створено окремі проєкти у складі загального рішення:

– UnitTests – для модульного тестування використовувався xUnit. Тут імітувались залежності, а також перевірялась логіка без фактичного доступу до бази даних або зовнішніх ресурсів завдяки EntityFramework InMemory Db.

– IntegrationTests – перевірка взаємодії сервісів, контролерів і InMemory Database.

– PostmanTests – JSON-колекція запускалась за допомогою newman (CLI-версія Postman). Таким чином вдалось автоматизувати HTTP-запити до REST API у реальному середовищі і перевірити можливість авторизації.

– LoadTests – створено скрипти для k6, що імітували багатопоточні навантаження на REST API.

Результати проходження вище описаних тестів продемонстровано на рис. 3.2-3.5. Звідси, автоматизоване тестування охопило як технічні аспекти, так і бізнес-сценарії, що дозволяє досягнути високого рівня впевненості в працездатності та надійності розробленого рішення.

```
PS C:\Users\dimon\Documents\GitHub\BusConnectedRidesApi> dotnet test ./UnitTests --no-build --logger trx
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v2.5.3.1+6b60a9e56a (64-bit .NET 8.0.14)
[xUnit.net 00:00:00.04]   Discovering: UnitTests
[xUnit.net 00:00:00.06]   Discovered:  UnitTests
[xUnit.net 00:00:00.06]   Starting:    UnitTests
Secret byte length: 44
[xUnit.net 00:00:00.45]   Finished:    UnitTests
Results File: C:\Users\dimon\Documents\GitHub\BusConnectedRidesApi\UnitTests\TestResults\dimon_LUMPI_2025-05-31_21_23_45.trx
  UnitTests test succeeded (0,9s)

Test summary: total: 12, failed: 0, succeeded: 12, skipped: 0, duration: 0,9s
Build succeeded in 1,1s
```

Рисунок 3.2 – Проходження юніт-тестів

```
Results File: C:\Users\dimon\Documents\GitHub\BusConnectedRidesApi\IntegrationTests\TestResults\dimon_LUMPI_2025-06-01_15_29_44.trx
  IntegrationTests test succeeded (2,8s)

Test summary: total: 12, failed: 0, succeeded: 12, skipped: 0, duration: 2,8s
Build succeeded in 3,1s
```

Рисунок 3.3 – Проходження інтеграційних-тестів

```
→ Auth
  POST https://localhost:7102/api/auth/login [200 OK, 437B, 65ms]
  √  Response status code is 200
  √  Response time is less than 200ms
  √  Content-Type header is application/json
  √  Token field exists in the response
  √  Token is a non-empty string
  √  Token is extracted and stored in environment variable

→ Buses
  GET https://localhost:7102/api/Buses [200 OK, 542B, 72ms]
  √  Response content type is JSON
```

	executed	failed
iterations	1	0
requests	2	0
test-scripts	4	0
prerequest-scripts	2	0
assertions	7	0
total run duration: 313ms		
total data received: 697B (approx)		
average response time: 68ms [min: 65ms, max: 72ms, s.d.: 3ms]		

Рисунок 3.4 – Проходження колекції тестів Postman

```
PS C:\Users\dimon\Documents\GitHub\BusConnectedRidesApi\LoadTest> k6 run rest-load-test.js

         /\      Grafana   /‾‾/
    /\  /  \     |\  __   /  /
   /  \/    \    | |/ /  /   ‾‾\
  /          \   |   (  |  (‾)  |
 / __________ \  |_|\_\  \_____/

     execution: local
        script: rest-load-test.js
        output: -

     scenarios: (100.00%) 1 scenario, 50 max VUs, 1m20s max duration (incl. graceful stop):
              * default: Up to 50 looping VUs for 50s over 3 stages (gracefulRampDown: 30s, gracefulStop: 30s)


  █ TOTAL RESULTS

    checks_total.......................: 2416    47.852922/s
    checks_succeeded...................: 100.00% 2416 out of 2416
    checks_failed......................: 0.00%   0 out of 2416

    ✓ status is 200
    ✓ body is not empty

    HTTP
    http_req_duration.......................................................: avg=6.23ms min=1.5ms med=5.1ms max=1s    p(90)=6.44ms p(95)=7.99ms
      { expected_response:true }............................................: avg=6.23ms min=1.5ms med=5.1ms max=1s    p(90)=6.44ms p(95)=7.99ms
    http_req_failed.........................................................: 0.00%  0 out of 1208
    http_reqs...............................................................: 1208   23.926461/s

    EXECUTION
    iteration_duration......................................................: avg=1s     min=1s    med=1s    max=2.03s p(90)=1s     p(95)=1s
    iterations..............................................................: 1208   23.926461/s
    vus.....................................................................: 3      min=1         max=50
    vus_max.................................................................: 50     min=50        max=50

    NETWORK
    data_received...........................................................: 690 kB 14 kB/s
    data_sent...............................................................: 72 kB  1.4 kB/s




running (0m50.5s), 00/50 VUs, 1208 complete and 0 interrupted iterations
default ✓ [======================================] 00/50 VUs  50s
```

Рисунок 3.5 – Тестування навантаження через k6

Висновки до третього розділу

У межах розділу було розроблено тестові випадки, що покрили переважну частину бізнес-логіки проєкту. Завдяки таким інструментам тестування, як xUnit, Entity Framework Core InMemory Database, Postman і k6, проведено верифікацію роботи всіх важливих частин додатку і налагоджено всі підсистеми відповідно до функціональних і нефункціональних вимог до програмного забезпечення. Додатково було проведено дослідницьке тестування завдяки Swagger і ручним запитам для перевірки повної працездатності додатка.

ВИСНОВКИ

У результаті виконання усіх етапів кваліфікаційної роботи було здійснено всебічний аналіз ринку та функціональних можливостей сучасних додатків для пасажирських перевезень. Особливу увагу приділено вивченню наявних рішень, їхніх переваг та недоліків, що дозволило визначити основні потреби кінцевого користувача та виявити ключові вимоги до майбутнього програмного продукту.

На основі дослідження було детально розглянуто сучасні інформаційні технології, які застосовуються у сфері розроблення подібних систем. З урахуванням технічних можливостей та вимог проєкту, були обрані найбільш ефективні інструменти та технології для реалізації додатку сумісних автобусних поїздок. Зокрема, було побудовано повну архітектуру системи з урахуванням принципів моделювання архітектури API, що забезпечує гнучкість, масштабованість і легкість інтеграції.

У ході реалізації проєкту було створено повноцінний програмний продукт, який має широкі перспективи застосування в майбутньому. Розроблена система може бути використана як технологічна основа для комерційного продажу або ж як база для подальшого розвитку власного стартапу з метою виходу на ринок і конкуренції з наявними сервісами пасажирських перевезень.

Під час проведення тестування програмного забезпечення було налагоджено всі ключові компоненти системи. Проведено перевірку працездатності, правильності реалізації бізнес-логіки та загальної придатності до експлуатації. Це дало змогу переконатися у стабільності роботи додатку та відповідності поставленим вимогам.

У підсумку, створена система є масштабованою, зручною у використанні та має значний потенціал для подальшого розвитку й удосконалення. Результати кваліфікаційної роботи підтверджують доцільність розробки подібних програмних продуктів та їхню актуальність у сучасних умовах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Zandt F. Infographic: Taking Taxis Trumps Sharing Rides // *Statista Daily Data*. URL: https://www.statista.com/chart/31348/share-of-respondents-using-taxis-and-ride-sharing-ride-hailing-services/ (дата звернення: 01.06.2025).
2. Dictionary.com | Meanings & Definitions of English Words // *Dictionary.com*. URL: https://www.dictionary.com/browse/ridesharing (дата звернення: 01.06.2025).
3. Kim S., Lee H., Son S.-W. Emerging Diffusion Barriers of Shared Mobility Services in Korea // *Sustainability*. 2021. Vol. 13, No. 14. P. 7707. URL: https://doi.org/10.3390/su13147707 (дата звернення: 02.06.2025).
4. Ghoseiri K. et al. Real-time rideshare matching problem. – Mid-Atlantic Universities Transportation Center, 2010. – № UMD-2009-04. URL: https://rosap.ntl.bts.gov/view/dot/25988#tabs-2 (дата звернення: 01.06.2025).
5. Beirigo B. A., Atasoy B. To wait or not to wait? A learning-based approach for on-demand ride-pooling water transport systems // *INFORMS Transportation Science and Logistics Workshop*. 2022.
6. White P. *Public Transport*. Ed. 6. – New York: Routledge, 2016. – 292 с. URL: https://doi.org/10.4324/9781315675770 (дата звернення: 01.06.2025).
7. Shaheen S. Shared mobility: The potential of ridehailing and pooling. – Island Press / Center for Resource Economics, 2018. – С. 55–76. URL: https://link.springer.com/chapter/10.5822/978-1-61091-906-7_3 (дата звернення: 01.06.2025).
8. Schneider T. Taxi, Uber, and Lyft Usage in New York City // *toddwschneider.com*. URL: https://toddwschneider.com/posts/taxi-uber-lyft-usage-new-york-city/ (дата звернення: 01.06.2025).
9. Uber App Features: Experience Seamless Rides and Safety // *Codeflash Infotech*. URL: https://codeflashinfotech.com/top-uber-app-features/ (дата звернення: 01.06.2025).

10. Krawczyk R., Ziółkowski M. How to build an app like Uber in 6 steps: a complete guide // *RST Software*. URL: https://www.rst.software/blog/how-to-build-an-app-like-uber-in-6-steps-a-complete-guide (дата звернення: 02.06.2025).
11. Sharma R. Uber's App for Drivers Has These 6 Features You Didn't Know About // *Gadgets 360*. URL: https://www.gadgets360.com/apps/features/6-features-of-the-uber-app-for-drivers-you-didnt-know-about-1474144 (дата звернення: 02.06.2025).
12. Uber vs. Lyft Driver: Who Pays Better? // *Life Asset*. URL: https://lifeasset.org/uber-vs-lyft/ (дата звернення: 08.06.2025).
13. Must-Have Uber App Features: Key Features to Include // *Addevice*. URL: https://www.addevice.io/blog/uber-app-features (дата звернення: 02.06.2025).
14. Islam T. System Analysis of Ride Sharing Application API and Practical Experience // *Medium*. URL: https://tariqul-islam-rony.medium.com/system-analysis-of-ride-sharing-application-api-and-practical-experience-2d2d0a626263 (дата звернення: 02.06.2025).
15. Singh K. P. System Design: Uber // *Medium*. URL: https://medium.com/@karan99/system-design-uber-33593137a4fe (дата звернення: 02.06.2025).
16. Asghari M. et al. Price-aware real-time ride-sharing at scale // *SIGSPATIAL'16: 24th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*. Burlingame, CA. New York, 2016. URL: https://doi.org/10.1145/2996913.2996974 (дата звернення: 01.06.2025).
17. System Design Uber, Lyft | System Design Interview Question. URL: https://systemdesignschool.io/problems/uber/solution (дата звернення: 08.06.2025).
18. Appelqvist R., Örnmyr O. Performance Comparison of XHR Polling, Long Polling, Server Sent Events and Websockets: thesis. – 2017. – 47 с. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:bth-14497 (дата звернення: 01.06.2025).

19. Uber Cost Algorithm // *Why Boobo?*. URL: https://whyboobo.com/research/uber-costs/ (дата звернення: 01.06.2025).
20. Hancock D., Humphrey D. B. Payment Transactions, Instruments, and Systems: A Survey // *Journal of Banking & Finance*. 1997. Vol. 21, No. 11–12. P. 1573–1624. URL: https://doi.org/10.1016/s0378-4266(97)00046-0 (дата звернення: 01.06.2025).
21. Zhang Z., Chen M. Advantages and Disadvantages of Third Party Payment Method and Traditional Payment Method // *Proc. of the 10th Int. Conf.*, Beijing, China, 15–17 July 2019. New York, 2019. URL: https://doi.org/10.1145/3345035.3345091 (дата звернення: 01.06.2025).
22. De Lauretis L. From Monolithic Architecture to Microservices Architecture // *2019 IEEE Int. Symp. on Software Reliability Engineering Workshops (ISSREW)*, Berlin, Germany, 27–30 October 2019. – C. 93–96. URL: https://doi.org/10.1109/issrew.2019.00050 (дата звернення: 01.06.2025).
23. Nugraha K. A. Real-Time Bus Arrival Time Estimation API using WebSocket in Microservices Architecture // *Int. J. on Advanced Science, Engineering and Information Technology*. 2023. Vol. 13, No. 3. P. 1018. URL: https://doi.org/10.18517/ijaseit.13.3.18116 (дата звернення: 01.06.2025).
24. The Architecture of Uber's API Gateway // *Uber Blog*. URL: https://www.uber.com/en-FI/blog/architecture-api-gateway/ (дата звернення: 01.06.2025).
25. Smith J. *Entity Framework Core in Action*. 2nd ed. – Manning Publications Co. LLC, 2021. – 520 с.
26. Auburn M., Bryant D., Gough J. *Mastering API Architecture: Defining, Connecting, and Securing Distributed Systems and Microservices*. – O'Reilly Media, 2022. – 286 с.
27. Bucko A. et al. Enhancing JWT Authentication and Authorization in Web Applications Based on User Behavior History // *Computers*. 2023. Vol. 12, No. 4. P. 78. URL: https://doi.org/10.3390/computers12040078 (дата звернення: 01.06.2025).

28. Rajam S. et al. Enterprise Service Bus Dependency Injection on MVC Design Patterns // *2010 IEEE Region 10 Conf. (TENCON 2010)*, Fukuoka, 21–24 November 2010. URL: https://doi.org/10.1109/tencon.2010.5686452 (дата звернення: 01.06.2025).
29. Bogard J. *AutoMapper Documentation. Release.* – 16 November 2017. – 50 с. URL: https://app.readthedocs.org/projects/automapper/downloads/pdf/v6.2.1/ (дата звернення: 01.06.2025).
30. Nanayakkara S. et al. Software for IT Project Quality Management // *Domain-Specific Bodies of Knowledge in Project Management*. 2023. P. 411–450. URL: https://doi.org/10.1142/9789811240584_0015 (дата звернення: 01.06.2025).
31. Metzgar D. *NET Core in Action*. – Manning Publications Co. LLC, 2018. – 288 с.
32. Bozo L., Dedej O. In-Memory Database Testing Performance Measurements in Azure // *RTA-CSIT*. 2021. – С. 61–66.
33. Kore P. P. et al. API Testing Using Postman Tool // *International Journal for Research in Applied Science and Engineering Technology*. 2022. Vol. 10, No. 12. С. 841–843.
34. Alvarez F., Argente D. Consumer Surplus of Alternative Payment Methods // *Review of Economic Studies*. 2024. URL: https://doi.org/10.1093/restud/rdae112 (дата звернення: 01.06.2025).

ДОДАТКИ

Додаток А – Базовий код реалізації CRUD-операцій

```
using System.Linq; using BusConnectedRidesApi.Data;
using Microsoft.EntityFrameworkCore;

namespace BusConnectedRidesApi.Servises.AutoCrud_Service
{
  public class AutoCrud_Service : IAutoCrud_Service
  {
    private readonly BusRidesDbContext _context;
    public AutoCrud_Service(BusRidesDbContext context) {
      _context = context;
    }
    private IQueryable<T> GetIncludedQuerry<T>(IQueryable<T> query) where T :
class
    {
        var entityTypes = _context.Model.GetEntityTypes();

        foreach (var property in typeof(T).GetProperties())
        {
            if (entityTypes.Any(e => e.ClrType == property.PropertyType))
            {
                query = query.Include(property.Name);
            }
        }

        return query;
    }

    public async Task<T?> GetByIdAsync<T>(int id) where T : class {
        IQueryable<T> query = _context.Set<T>();
        query = GetIncludedQuerry(query);

        return await _context.Set<T>().FindAsync(id);
    }

    public async Task<List<T>> GetAllAsync<T>() where T : class {
        IQueryable<T> query = _context.Set<T>();
        query = GetIncludedQuerry(query);

        return await query.ToListAsync();
    }
    public async Task AddAsync<T>(T entity) where T : class {
        await _context.Set<T>().AddAsync(entity);
        await _context.SaveChangesAsync();
    }
    public async Task UpdateAsync<T>(T entity) where T : class {
        _context.Set<T>().Update(entity);
        await _context.SaveChangesAsync();
    }
    public async Task DeleteAsync<T>(int id) where T : class {
        var entity = await GetByIdAsync<T>(id);
        if (entity != null)
        {
            _context.Set<T>().Remove(entity);
            await _context.SaveChangesAsync();
        }
    }
  }
}
```

Додаток Б – Перетворення моделі

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        // Short DTOs
        CreateMap<Drivers, DriversShortDTO>();
        CreateMap<Buses, BusesShortDTO>();
        CreateMap<Customers, CustomersShortDTO>();
        CreateMap<Payments, PaymentsShortDTO>();
        CreateMap<Trips, TripsShortDTO>();
        CreateMap<Subtrips, SubtripsShortDTO>();

        // Full BusesDTO (with DriverShortDTO)
        CreateMap<Buses, BusesDTO>()
            .ForMember(dest => dest.DriverDTO, opt => opt.MapFrom(src =>
src.Driver))
            .ReverseMap();

        // Full DriversDTO (with BusShortDTO)
        CreateMap<Drivers, DriversDTO>()
            .ForMember(dest => dest.Latitude, opt => opt.MapFrom(src =>
src.Location.Y))
            .ForMember(dest => dest.Longitude, opt => opt.MapFrom(src =>
src.Location.X))
            .ForMember(dest => dest.BusDTO, opt => opt.MapFrom(src => src.Bus))
            .ReverseMap()
            .ForMember(dest => dest.Location, opt => opt.MapFrom(src =>
                new NetTopologySuite.Geometries.Point(src.Longitude,
src.Latitude) { SRID = 4326 }))
            .ForMember(dest => dest.Bus, opt => opt.MapFrom(src => src.BusDTO));

        // Customers
        CreateMap<Customers, CustomersDTO>()
            .ForMember(dest => dest.Payment, opt => opt.MapFrom(src =>
src.Payment))
            .ForMember(dest => dest.Subtrip, opt => opt.MapFrom(src =>
src.Subtrip))
            .ForMember(dest => dest.TripId, opt => opt.MapFrom(src =>
src.TripId))
            .ReverseMap();

        // Payments
        CreateMap<Payments, PaymentsDTO>()
            .ForMember(dest => dest.Subtrip, opt => opt.MapFrom(src =>
src.Subtrip))
            .ForMember(dest => dest.Customer, opt => opt.MapFrom(src =>
src.Customer))
            .ReverseMap();

        // Trips
        CreateMap<Trips, TripsDTO>()
            .ForMember(dest => dest.Customers, opt => opt.MapFrom(src =>
src.Customers))
```

```
            .ForMember(dest => dest.Subtrips, opt => opt.MapFrom(src =>
src.Subtrips))
            .ForMember(dest => dest.Driver, opt => opt.MapFrom(src =>
src.Driver))
            .ReverseMap();

        // Subtrips
        CreateMap<Subtrips, SubtripsDTO>()
            .ForMember(dest => dest.LatitudeSource, opt => opt.MapFrom(src =>
src.Source.Y))
            .ForMember(dest => dest.LongitudeSource, opt => opt.MapFrom(src =>
src.Source.X))
            .ForMember(dest => dest.LatitudeDestination, opt => opt.MapFrom(src
=> src.Destination.Y))
            .ForMember(dest => dest.LongitudeDestination, opt => opt.MapFrom(src
=> src.Destination.X))
            .ReverseMap()
            .ForMember(dest => dest.Source, opt => opt.MapFrom(src =>
                new NetTopologySuite.Geometries.Point(src.LongitudeSource,
src.LatitudeSource) { SRID = 4326 }))
            .ForMember(dest => dest.Destination, opt => opt.MapFrom(src =>
                new NetTopologySuite.Geometries.Point(src.LongitudeDestination,
src.LatitudeDestination) { SRID = 4326 }));
    }
}
```

Додаток В – Посилання на репозиторій

Репозиторій з програмною реалізацією, виконаною в межах кваліфікаційної роботи, розташовано за адресою https://github.com/Lump1/BusConnectedRidesApi