Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система планування маршрутів поїздок та продажу квитків з урахуванням пересадок на різні види транспорту. API на основі протоколу HTTP.
(тема)

Виконав:
здобувач 4 року навчання
групи ПЗПІ-21-4
Данило НАЗАРЬКО
(Власне ім’я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник проф. кафедри ПІ Володимир КОБЗЄВ
(посада, Власне ім’я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім’я, ПРІЗВИЩЕ)

2025

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіт перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"___" _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Назарьку Данилу Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Програмна система планування маршрутів поїздок та продажу квитків з урахуванням пересадок на різні види транспорту. API на основі протоколу HTTP.

Затверджена наказом по університету від 397Ст від 19.05.2025

2. Термін подання здобувачем роботи до екзаменаційної комісії 16.06.2025

3. Вихідні дані до роботи Розробити програмну систему для планування маршрутів із пересадками та продажу квитків, що інтегрує дані різних транспортних компаній та автоматизує управління ресурсами.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	20.05.2025	*виконано*
2	Створення специфікації ПЗ	22.05.2025	*виконано*
3	Проектування ПЗ	24.05.2025	*виконано*
4	Розробка ПЗ	28.05.2025	*виконано*
5	Тестування ПЗ	30.05.2025	*виконано*
6	Оформлення пояснювальної записки	07.06.2025	*виконано*
7	Підготовка презентації та доповіді	08.06.2025	*виконано*
8	Попередній захист	11.06.2025	*виконано*
9	Нормоконтроль, рецензування	08.06.2025	*виконано*
10	Здача роботи у електронний архів	12.06.2025	*виконано*
11	Допуск до захисту у зав. кафедри	13.06.2025	*виконано*

Дата видачі завдання 19 травня 2025р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. кафедри ПІ Володимир КОБЗЄВ
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра, 85 стор., 9 рис., 2 табл., 18 джерел.

АЛГОРИТМ ДЕЙКСТРИ, ВЕБ-СИСТЕМА, ПОШУК МАРШРУТІВ ІЗ ПЕРЕСАДКОЮ, ПРОДАЖ КВИТКІВ, ANSIBLE, APS.NET, C#, CONTINUOUS DELIVERY, CONTINUOUS INTEGRATION, HTTP API, LAYERED CLEAN ARCHITECTURE, POSTGRESQL, PROXMOX, SCALEABLE DEPLOYMENT, TERRAFORM

Робота присвячена розробці програмної системи для планування мультитранспортних поїздок із пересадками та продажу квитків. Метою є створення уніфікованого рішення, яке інтегрує дані різних транспортних компаній, автоматизує управління ресурсами. Актуальність дослідження обумовлена зростанням потреби в ефективному плануванні подорожей, що поєднують різні види транспорту, та відсутністю інтеграції між існуючими системами.

Система базується багатошаровій архітектурі з використанням технологій ASP.NET та PostgreSQL для розробки. Для пошуку оптимальних маршрутів застосовано алгоритм Дейкстри, який враховує час, вартість та кількість пересадок. Реалізовано централізовану базу даних, що об'єднує інформацію про станції, транспорт, розклади та квитки. Інтеграція з платіжними системами дозволяє миттєво оформлювати комбіновані квитки.

Proxmox, Terraform, Ansible та Docker забезпечують масштабоване розгортання. Стійкість гарантують модульне та інтеграційне тестування.

Результатом роботи стало створення функціональної платформи, що спрощує планування подорожей для пасажирів, оптимізує управління транспортом для перевізників та зменшує операційні витрати.

ABSTRACT

DIJKSTRA'S ALGORITHM, WEB SYSTEM, ROUTE SEARCH WITH TRANSFERS, TICKET SALES, ANSIBLE, APS.NET, C#, CONTINUOUS DELIVERY, CONTINUOUS INTEGRATION, HTTP API, LAYERED CLEAN ARCHITECTURE, POSTGRESQL, PROXMOX, SCALABLE DEPLOYMENT, TERRAFORM

The work is dedicated to the development of a software system for planning multimodal routes with transfers and ticket sales. The goal is to create a unified solution that integrates data from various transport companies and automates resource management. The relevance of the study stems from the growing need for efficient travel planning that combines different modes of transport and the lack of integration between existing systems.

The system is based on a multi-layered architecture using ASP.NET and PostgreSQL for development. Dijkstra's algorithm is applied to find optimal routes, considering time, cost, and the number of transfers. A centralized database integrates information about stations, transport, schedules, and tickets. Integration with payment systems enables instant booking of combined tickets.

The system supports scalable deployment through Proxmox, Terraform, Ansible, and Docker, ensuring flexibility and resilience under load. Testing included unit tests for modules and integration testing.

The result is a functional platform that simplifies travel planning for passengers, optimizes transport management for carriers, and reduces operational costs.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

СУБД (Система управління базами даних) – програмне забезпечення для створення, управління та взаємодії з базами даних (наприклад, MySQL, PostgreSQL).

API (Application Programming Interface) – інтерфейс програмування застосунків – набір правил, протоколів та інструментів, що дозволяють різним програмним компонентам взаємодіяти між собою.

CRM-система (Customer Relationship Management) – програмне забезпечення для управління взаємодією з клієнтами, яке використовує аналіз даних для покращення бізнес-відносин, автоматизації продажів, маркетингу та підтримки.

DDoS-атака (Distributed Denial of Service Attack) – кібератака, спрямована на перевантаження сервера або мережі величезним обсягом трафіку з багатьох джерел, що призводить до тимчасової недоступності ресурсу.

ERP-система (Enterprise Resource Planning) – інтегрована платформа для управління основними бізнес-процесами (наприклад, ланцюгом поставок, фінансами, HR) у реальному часі через централізовану базу даних.

GDS (Global Distribution System) – глобальна дистриб'юторська система, що об'єднує інформацію про послуги авіакомпаній, готелів, прокату авто тощо, дозволяючи турагентам та корпораціям бронювати їх у режимі онлайн.

GDPR (General Data Protection Regulation) – загальний регламент ЄС щодо захисту даних, що регулює обробку персональних даних громадян Європи, вимагаючи від компаній прозорості, згоди користувачів та захисту інформації.

HTTP (Hypertext Transfer Protocol) – протокол передачі гіпертексту, основа комунікації в інтернеті, який визначає формат запитів та відповідей між клієнтами (наприклад, браузерами) і серверами.

KVM (Kernel-based Virtual Machine) – модуль ядра Linux для апаратної віртуалізації.

QEMU – емулятор, який забезпечує віртуалізацію архітектур.

LXC (Linux Containers) – технологія контейнеризації на рівні ОС, що ізолює процеси та ресурси для запуску множини "контейнерів" на єдиному ядрі Linux.

PCI DSS (Payment Card Industry Data Security Standard) – стандарт безпеки даних платіжних карток, обов'язковий для організацій, що обробляють, зберігають або передають дані карток (наприклад, банки, мерчанти).

REST API (Representational State Transfer Application Programming Interface) – архітектурний стиль для розробки API, де клієнт і сервер обмінюються даними через HTTP-методи (GET, POST тощо).

SOLID – набір принципів об'єктно-орієнтованого програмування: Single Responsibility (єдина відповідальність класу), Open/Closed (відкритість для розширення, закритість для змін), Liskov Substitution (об'єкти підкласів повинні замінювати об'єкти батьківських класів), Interface Segregation (спеціалізовані інтерфейси замість універсальних), Dependency Inversion (залежність від абстракцій, а не реалізацій).

SPA (Single Page Application) – веб-застосунок, який завантажує єдину HTML-сторінку та динамічно оновлює її вміст без перезавантаження, використовуючи JavaScript-фреймворки (наприклад, React, Angular).

TLS (Transport Layer Security) – криптографічний протокол для захисту передачі даних через інтернет. Використовується в HTTPS, електронній пошті тощо.

TMS (Transportation Management System) – система управління транспортними операціями: планування маршрутів, моніторинг вантажів, оптимізація використання рухомого складу.

ВСТУП

Сучасний світ характеризується стрімким зростанням мобільності населення, розширенням транспортних мереж та збільшенням попиту на мультитранспортні подорожі, що поєднують різні види транспорту (автобуси, літаки, поїзди, тощо). Однак планування таких маршрутів залишається складним завданням як для користувачів, так і для транспортних компаній. Існуючі системи часто обмежені: вони не враховують динамічні зміни розкладів, ускладнюють пошук оптимальних варіантів із пересадками, а також не забезпечують повноцінної інтеграції з платіжними сервісами та інструментами управління ресурсами перевізників. Це призводить до неефективного використання транспортної інфраструктури, збільшення витрат часу пасажирів та зниження рівня їхнього комфорту.

Актуальність роботи обумовлена необхідністю створення уніфікованого програмного рішення, яке автоматизує процеси планування маршрутів, продажу квитків та управління транспортними ресурсами. Зростання конкуренції на ринку перевезень, вимоги до екологічної ефективності та очікування клієнтів щодо зручності оформлення поїздок підкреслюють потребу в інноваційних інструментах. Крім того, відсутність централізованих систем, що об'єднують дані різних перевізників, ускладнює координацію між ними та унеможливлює швидке реагування на збої, наприклад, скасування рейсів.

Метою роботи є розробка програмної системи для планування маршрутів із пересадками та продажу квитків, яка інтегрує функції управління транспортом, персоналом перевізників, станціями та платіжними операціями. Система має забезпечувати:

- формування оптимальних маршрутів з урахуванням часу, вартості та доступності різних видів транспорту;
- автоматизований облік транспортних засобів та розкладів руху;
- інтеграцію з платіжними системами для миттєвого оформлення квитків;

- групування станцій у маршрути, які можна прив'язувати до конкретних транспортних засобів;
- розмежування прав доступу для персоналу компаній-перевізників (водії, диспетчери, адміністратори).

Основні завдання дослідження:

- аналіз існуючих аналогічних систем та виявлення їхніх недоліків;
- проектування архітектури системи, включаючи модулі планування маршрутів, управління даними та інтеграції зі зовнішніми сервісами;
- розробка алгоритмів пошуку оптимальних шляхів із врахуванням пересадок між різними видами транспорту;
- реалізація механізму групування станцій в маршрути;
- налаштування інтеграції з платіжними системами (наприклад, Stripe, PayPal) для безпечного проведення транзакцій;
- створення прикладного інтерфейсу на основі протоколу HTTP для подальшої можливості набудови користувацького інтерфейсу;
- тестування працездатності системи та оцінка її ефективності.

Галузь застосування результатів охоплює:

- транспортні компанії – для автоматизації управління автопарком, розкладами та персоналом;
- агрегатори подорожей – як основа для платформ із продажу квитків на мультитранспортні подорожі;
- туристичні агенції – формування комплексних турів із використанням різних видів перевезень;

Реалізація такої системи сприятиме переходу до інтелектуальних транспортних мереж, де кожен елемент (від квитка до розкладу) взаємопов'язаний. Це не лише підвищить якість послуг для пасажирів, але й зробить роботу перевізників більш прозорою та економічно ефективною.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Сьогодні транспортна галузь охоплює три основні види перевезень: автомобільні (автобуси), залізничні (поїзди) та авіаційні. Кожен із цих видів має власну інфраструктуру, правила роботи та програмні інструменти для управління. Наприклад:

- авіакомпанії використовують системи типу GDS (Global Distribution Systems) для продажу квитків і управління рейсами;
- залізничні перевізники часто працюють через національні платформи (наприклад, Deutsche Bahn у Німеччині);
- автобусні перевезення керуються локальними розкладами та мобільними застосунками (на кшталт FlixBus).

Проте інтеграція між цими системами відсутня. Пасажирам, які планують подорож із пересадками, доводиться самостійно поєднувати дані з різних джерел, що збільшує ризик помилок і втрату часу.

Крім того, транспортні компанії стикаються з низкою внутрішніх проблем:

- розподіл транспорту, персоналу та розкладів часто здійснюється без задіяння автоматизації через електронні таблиці або застарілі ERP-системи;
- відсутність вчасної реакції на зміни в розкладах (затримки, скасування), вони не відображаються миттєво, що призводить до порушень логістики.

Пасажири, які використовують кілька видів транспорту, стикаються з такими труднощами:

- відсутність єдиного джерела даних про розклади, ціни та наявність квитків. Наприклад, поєднання авіарейсу з автобусним маршрутом вимагає перевірки кількох сайтів;
- існуючі планувальники (Google Maps, Rome2Rio) пропонують варіанти на основі загальних алгоритмів, без врахування актуального стану транспорту (наприклад, ремонт колій) або персональних уподобань (пріоритет вартості над часом);

- складність придбання квитків через необхідність взаємодії з різними сервісами, що може призвести до зниження безпеки та зручності процесу.

Сучасні платформи для продажу квитків або планування маршрутів (наприклад, Omio, Trainline) мають такі недоліки:

- більшість систем спеціалізуються на одному виді транспорту;
- розклади оновлюються вручну, що робить їх нечутливими до динамічних змін (погодні умови, страйки);
- відокремленість від управління перевізниками, системи не інтегруються з внутрішніми процесами компаній (наприклад, облік водіїв або технічного стану автобусів), що ускладнює оперативну корекцію маршрутів;

На основі аналізу ринку та потреб користувачів можна виділити ключові вимоги до програмної системи планування маршрутів:

- об'єднання інформації від різних перевізників у єдиному застосунку;
- урахування змін у реальному часі (наприклад, затримки рейсів) та автоматична корекція маршрутів;
- інтеграція з бекендом перевізників: Можливість синхронізації з системами управління транспортом, персоналом та станціями;
- інструменти для оптимізації маршрутів на основі історичних даних та прогнозування навантаження;
- підтримка веб-інтерфейсів, мобільних застосунків та API для зовнішніх сервісів.

1.2 Виявлення та вирішення проблем

1.2.1 Фрагментація транспортних систем та відсутність інтеграції

Як зазначено в роботі [1], вибір методології збору даних у інформаційних системах критично впливає на точність та ефективність інтеграції різних джерел, що особливо актуально для транспортних платформ, де дані про розклади, станції та квитки надходять від розрізнених перевізників.

Проблема. Основним бар'єром для мультитранспортних подорожей є технологічна роздробленість систем різних видів транспорту. Дані про розклади,

доступність квитків та станції зберігаються в ізольованих базах, що унеможливлює їх автоматичну синхронізацію. Наприклад, авіакомпанія може використовувати власну CRM-систему, тоді як залізничний перевізник – локальну платформу, несумісну з іншими.

Наслідки:

- пасажири отримують неповну або застарілу інформацію про маршрути;
- перевізники втрачають потенційних клієнтів через неможливість пропозиції комбінованих послуг;
- зростають операційні витрати на узгодження даних між компаніями.

Вирішення:

- розробка уніфікованого api для інтеграції з існуючими системами перевізників (наприклад, GDS для авіації, TMS для залізниць);
- створення централізованої бази даних зі стандартизованими форматами інформації (розклади, ціни, станції).

1.2.2 Неефективне управління ресурсами перевізників

Проблема. Більшість компаній керують транспортом, персоналом та розкладами за допомогою застарілих інструментів (Excel, ERP-системи 2-го покоління). Це призводить до:

- ручного введення даних, що сповільнює процеси;
- помилок у розподілі водіїв або технічного обслуговування транспорту;
- відсутності зв'язку між плануванням маршрутів та фактичним станом автопарку.

Приклад. Якщо автобус виходить з ладу, диспетчер не може оперативно переналаштувати маршрути інших транспортних засобів, що веде до скасування рейсів.

Вирішення:

- модуль для відстеження технічного стану транспорту (датчики GPS, IoT-пристрої);

- система управління персоналом з інтеграцією графіків роботи та кваліфікації співробітників;
- динамічне планування маршрутів за допомогою алгоритмів, що враховують доступність транспорту та персоналу в реальному часі.

1.2.3 Відсутність механізмів реального часу для користувачів

Проблема. Пасажири часто отримують інформацію про затримки або скасування рейсів із запізненням. Наприклад, дані про закриття аеропорту через погоду не відображаються в системах продажу квитків, що призводить до купівлі непотрібних квитків на автобуси чи поїзди.

Наслідки:

- зростання кількості скарг від клієнтів;
- втрата довіри до сервісів планування;
- фінансові збитки перевізників через невідповідність очікувань і реальності.

Вирішення:

- використання даних про погоду (OpenWeatherMap), дорожній трафік (Google Maps API);
- миттєве інформування пасажирів про зміни через мобільний застосунок або електронну пошту;
- автоматична генерація альтернативних варіантів при виявленні збоїв.

1.2.4 Обмеженість функціоналу продажу квитків

Проблема. Існуючі системи продажу квитків не підтримують мультитранспортність, що змушує пасажирів купувати квитки окремо для кожного виду транспорту. Додаткові труднощі:

- відсутність єдиного кошика для комбінованих маршрутів;
- обмежені способи оплати (наприклад, відсутність криптовалют);

Вирішення:

- мультивалютність та підтримка різних платіжних методів (Stripe, PayPal, Privat24, BTCPay);
- створення єдиного цифрового квитка, що охоплює етапи подорожі на різному транспорті.

1.2.5 Складність групування станцій та маршрутів

Проблема. Існуючі системи не дозволяють гнучко групувати станції в маршрути, які можна прив'язувати до конкретних транспортних засобів. Наприклад, автобусний маршрут між двома містами може включати різні проміжні зупинки залежно від типу автобуса або часу доби.

Наслідки:

- неефективне використання транспорту (напівпорожні рейси);
- незручність для пасажирів, які змушені вибирати між прямими та непрямими маршрутами.

Вирішення:

- створення можливості додаваня та видалення станцій з урахуванням їх геолокації та пропускної спроможності;
- візуалізація маршрутів на мапі з автоматичним розрахунком часу в дорозі;
- динамічне групування за допомогою алгоритми, які формують маршрути на основі попиту (наприклад, додаткові автобуси у зв'язку зі скасуванням поїзда).

1.3 Постановка задачі

1.3.1 Визначення архітектури системи

Потрібно розробити модульну архітектуру системи, яка забезпечить інтеграцію функцій планування маршрутів, управління ресурсами перевізників, обліку станцій та продажу квитків.

Розподіл на наступні модулі:

- модуль планування маршрутів, що відповідає за пошук оптимальних шляхів з урахуванням часу, вартості та пересадок;

- модуль управління транспортом, що включає облік транспортних засобів, їх технічного стану та розкладів;

- модуль персоналу для реєстрація працівників компаній-перевізників (водії, диспетчери) з розмежуванням прав доступу;

- модуль станцій і маршрутів для групування станцій у маршрути, прив'язка їх до конкретних транспортних засобів;

- платіжний модуль для інтеграції зі зовнішніми платіжними системами (Stripe, PayPal, BTCPay) та генерація квитків.

Технології для реалізації:

- бекенд на мові C# із фреймворком ASP.NET [2] для реалізації бізнес-логіки;

- підтримка декількох баз даних, спочатку PostgreSQL [3];

- REST API на основі протоколу HTTP для зовнішньої інтеграції (GDS авіакомпаній, TMS залізниць).

1.3.2 Розробка алгоритмів пошуку маршрутів

Завданням є створення алгоритмів, що генерують оптимальні мультитранспортні маршрути з урахуванням динамічних факторів (затримки, технічні збої).

Математична модель:

- використання теорії графів, де вершини – станції, ребра – транспортні зв'язки;

- ваги ребер – час, вартість, комфорт (наприклад, кількість пересадок).

Алгоритми пошуку маршрутів:

- алгоритм Дейкстри [4, с. 620] для пошуку найкоротшого шляху за часом або вартістю;

- машинне навчання для прогнозування затримок на основі історичних даних.

1.3.3 Побудова централізованої бази даних

Потрібно розробити базу даних, що об'єднує інформацію про станції, транспорт, персонал, розклади та квитки.

Структура даних:

- станції. назва, геокоординати, тип (аеропорт, вокзал), пропускна спроможність;
- транспортні засоби. id, тип (автобус, літак), місткість, технічні характеристики, графік то;
- персонал. дані працівників, графіки роботи, кваліфікація;
- розклади. час відправлення/прибуття, зв'язок із транспортними засобами та маршрутами.

Забезпечення цілісності даних:

- використання транзакційних запитів для уникнення конфліктів оновлення;
- реалізація тригерів для автоматичного перерахунку маршрутів при зміні розкладів та денормалізованих значень.

Масштабованість:

- індексація ключових полів (наприклад, геокоординати станцій) для прискорення пошуку.

1.3.4 Інтеграція з платіжними системами

Потрібно забезпечити безпечний процес оплати квитків із підтримкою мультивалютності та різних методів транзакцій.

Вибір платіжних шлюзів:

- Stripe для кредитних карток;
- BTCPay для криптовалют;
- Приват24 для українських користувачів.

Реалізація єдиного кошика:

- можливість додавати квитки на різні види транспорту в одне замовленняж;

- розрахунок загальної вартості з урахуванням знижок (наприклад, за ранньою покупкою).

1.3.5 Розробка інтерфейсів користувачів

Завданням є створення інтуїтивних інтерфейсів для різних категорій користувачів: пасажирів, диспетчерів, адміністраторів.

Пасажирський інтерфейс:

- пошук маршрутів за критеріями (час, ціна, екологічність);
- візуалізація маршруту на мапі (інтеграція з Google Maps API);
- особистий кабінет з історією поїздок та збереженими даними.

Інтерфейс диспетчера:

- панель управління транспортом із відображенням його стану в реальному часі;
- інструменти для корекції розкладів та повідомлення пасажирів.

Адміністративний інтерфейс:

- управління персоналом (додавання, видалення, налаштування прав);
- аналітика: завантаженість маршрутів, фінансові звіти, статистика використання.

1.3.6 Забезпечення безпеки системи

Потрібно реалізувати механізми захисту від кібератак та несанкціонованого доступу.

Шифрування даних:

- AES-256 для зберігання конфіденційної інформації (шифрування дисків серверів БД, data at rest encryption);
- TLS 1.3 для передачі даних між клієнтом і сервером.

Розмежування прав доступу:

- створення системи ролей: адміністратор (повний доступ), диспетчер (редагування розкладів), пасажир (перегляд);
- JWT-токени для аутентифікації користувачів.

Захист від DDoS-атак:

- використання CDN (Cloudflare) для розподілу навантаження;
- галаштування фаервола (Web Application Firewall) для блокування підозрілих запитів.

1.3.7 Тестування та валідація системи

Завданням є перевірка працездатності системи, її продуктивності та відповідності вимогам користувачів:

- юніт-тестування модулів пошуку маршрутів та іншої бізнес логіки на різних наборах даних;
- інтеграційне тестування системи цілком.

1.4 Порівняльний аналіз з існуючими рішеннями

1.4.1 Міжнародні аналоги

Таблиця 1 – Міжнародні аналоги

Назва сервісу	Переваги	Недоліки
Google Maps [5]	Інтеграція з різними видами транспорту (автобуси, поїзди, таксі)	Відсутність продажу квитків (лише посилання на сторонні сервіси)
	Реалізація алгоритмів пошуку маршрутів (на основі даних у реальному часі)	Обмежена підтримка мультитранспортних поїздок у деяких країнах
Rome2Rio [6]	Пошук комбінованих маршрутів (літаки + поїзди + автобуси)	Відсутність єдиного кошика для мультитранспортних квитків
	Підтримка бронювання квитків через партнерські сервіси	Обмежена інтеграція з локальними перевізниками
Omio [7]	Агрегація даних від різних перевізників (Європа, США)	Не враховує динамічні зміни розкладів (наприклад, затримки)
	Можливість купівлі квитків у застосунку	Відсутність інструментів для перевізників (управління транспортом)

1.4.2 Українські аналоги

Таблиця 2 – Українські аналоги

Назва сервісу	Переваги	Недоліки
Укрзалізниця [8]	Продаж квитків на поїзди з підтримкою електронного квитка	Обмеження лише залізничним транспортом
	Інтеграція з системою бронювання місць	Відсутність пошуку маршрутів з пересадками на інші види транспорту
FlixBus [9]	Швидке бронювання автобусних квитків	Відсутність інтеграції з іншими видами транспорту (наприклад, поїздами)
	Мобільний застосунок зі зручним інтерфейсом	Немає механізму динамічного оновлення розкладів

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Концепт-документ програмної системи

2.1.1 Мета та цільова аудиторія

Мета: створення інтегрованої платформи для планування мультитранспортних поїздок, автоматизації управління ресурсами перевізників та продажу квитків з урахуванням пересадок між різними видами транспорту.

Цільова аудиторія:

- користувачі, які потребують швидкого та зручного планування поїздок;
- автобусні оператори, залізничні та авіакомпанії, які керують транспортом і персоналом;
- організатори турів, що потребують інтеграції транспортних послуг.

2.1.2 Сценарії використання

Пошук маршруту з пересадками:

- пасажир вводить пункти відправлення та призначення, обирає критерії (час, вартість);
- система генерує варіанти з урахуванням доступних видів транспорту та пересадок.

Продаж квитка:

- користувач обирає маршрут, додає його до кошика, оплачує через інтегрований платіжний шлюз;
- система формує цифровий квиток із штрих-кодом.

Управління транспортом:

- диспетчер перевіряє технічний стан автобуса, коректує розклад при необхідності.

2.1.3 Ключові принципи розробки

- система складається з незалежних компонентів (планування, управління, оплата);

- всі операції (наприклад, придбання квитка) мають бути відкочуваними;
- відкритість API, інтеграція зі сторонніми сервісами (GDS, Google Maps).

2.2 Функціональні вимоги

2.2.1 Модуль планування маршрутів

Пошук оптимальних шляхів:

- врахування часу, вартості та кількості пересадок;
- підтримка динамічних оновлень (затримки, скасування рейсів);

Візуалізація маршруту:

- відображення на мапі з позначками станцій, видів транспорту та часу в дорозі.

2.2.2 Модуль управління ресурсами

Облік транспорту:

- реєстрація транспортних засобів, відстеження технічного стану;
- автоматичне формування розкладів на основі доступності.

Управління персоналом:

- розподіл працівників (водії, диспетчери) з урахуванням графіків та кваліфікації;
- розмежування прав доступу (ролі: адміністратор, диспетчер, водій).

2.2.3 Платіжний модуль

Інтеграція з платіжними системами Stripe, PayPal, Privat24 та BTCPay.

Генерація квитків:

- створення цифрових квитків у форматі PDF/QR-код;
- можливість повернення коштів у разі скасування.

2.2.4 Модуль станцій та маршрутів

Групування станцій:

- створення маршрутів через візуальний редактор з геолокацією;

- прив'язка маршрутів до конкретних транспортних засобів.

2.3 Нефункціональні вимоги

2.3.1 Продуктивність

- час відгуку не більше 2 секунд для пошуку маршрутів;
- підтримка до 10 000 одночасних користувачів;
- можливість розширення архітектури за рахунок додавання нових серверів.

2.3.2 Безпека

- шифрування використовуючи AES-256 для зберігання даних, TLS 1.3 для передачі;
- двофакторна аутентифікація (SMS, Time-Based One Time Password);
- відповідність стандартам PCI DSS [10] для платіжних операцій, GDPR [11] для обробки персональних даних.

2.3.4 Юзабіліті

- відповідність кращим практикам створення програмних інтерфейсів;
- підтримка Української та Англійської мов із можливістю перекладу іншими;
- підтримка локалізації відображуваних даних (дати, часу, чисел, валют);
- підтримка локального часу користувача враховуючи літній час.

2.4 Бізнес-вимоги

2.4.1 Економічні цілі

- зниження операційних витрат перевізників на 20-30% за рахунок автоматизації управління;
- збільшення доходів на 15% через продаж комбінованих квитків.

2.4.2 Організаційні цілі

- стандартизація процесів за рахунок уніфікації форматів даних для всіх учасників ринку;
- підвищення лояльності клієнтів за рахунок зручності та персоналізації.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проєктування ПЗ

На рисунку 1 показана діаграма прецедентів системи.

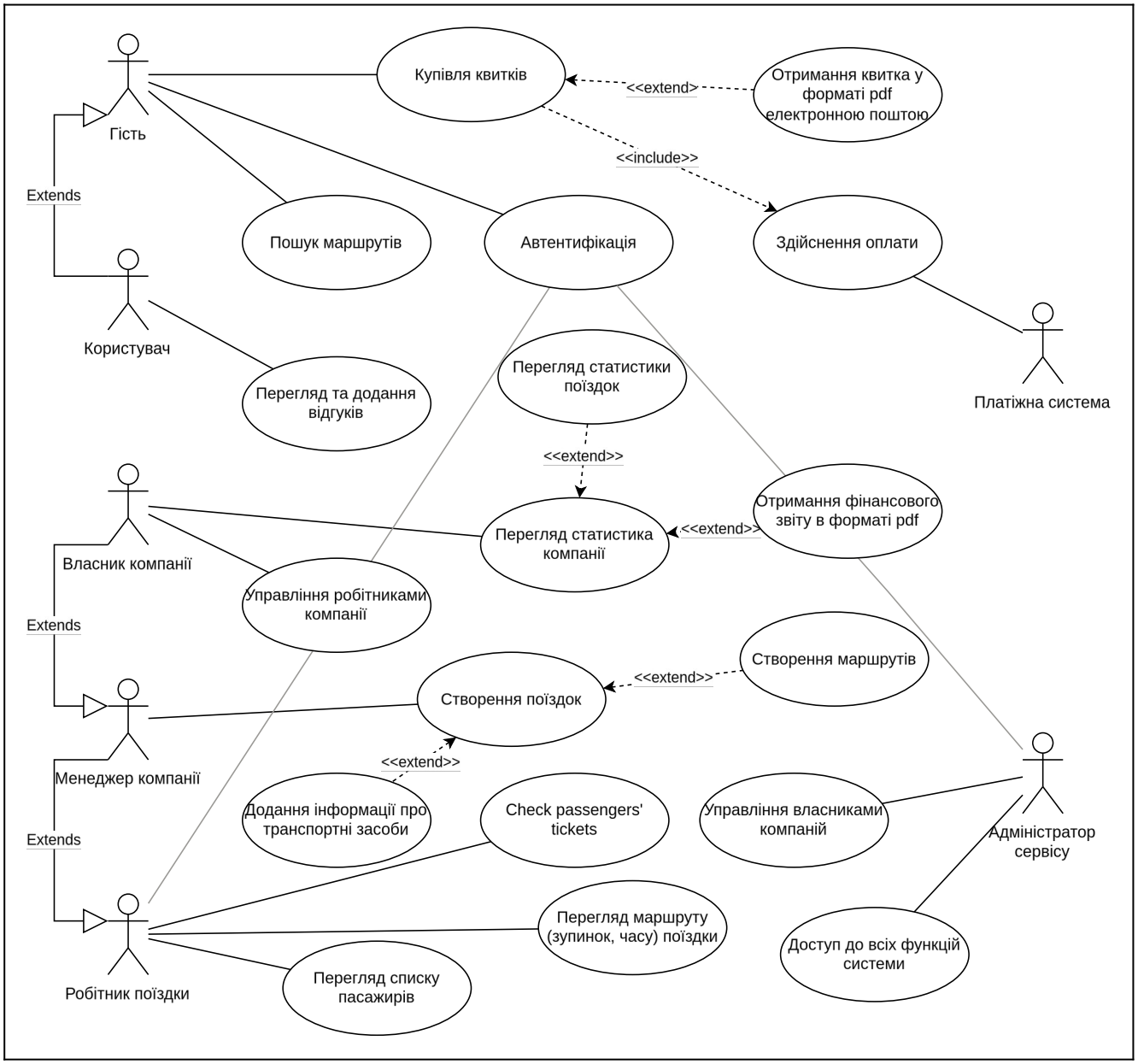

Рисунок 1 – Діаграма прецедентів системи

Система включає кілька акторів, кожен з яких взаємодіє з нею через певні варіанти використання. Користувач є основним актором, який може переглядати та додавати відгуки. Цей актор також має зв'язок із Гостем, який успадковує можливості користувача, але з обмеженим доступом. Гість може здійснювати пошук маршрутів, купівля квитків та автентифікацію. Купівля квитків включає

процес оплати через Платіжну систему та може бути розширена отриманням квитка у форматі PDF електронною поштою.

Власник компанії управляє робітниками компанії та переглядає статистику компанії. Ця роль також може отримувати фінансові звіти у форматі PDF. Менеджер компанії успадковує повноваження власника і додатково створює поїздки. Створення поїздок може бути розширене доданням інформації про транспортні засоби або створенням маршрутів.

Робітник поїздки взаємодіє з системою для перевірки квитків пасажирів, перегляду списку пасажирів та маршруту поїздки (зупинок та часу). Його дії також потребують автентифікації. Адміністратор сервісу має доступ до всіх функцій системи, включаючи управління власниками компаній та перегляд статистики. Він також використовує автентифікацію для забезпечення безпеки доступу.

Платіжна система виступає зовнішнім актором, який забезпечує проведення оплати за квитки. Цей процес тісно пов'язаний із варіантом використання "Здійснення оплати", який є частиною купівлі квитків для гостя або користувача. Усі актори, крім платіжної системи, взаємодіють із системою через автентифікацію, що підкреслює її ключову роль у забезпеченні контролю доступу.

3.2 Проєктування архітектури ПЗ

3.2.1 Архітектура коду системи

Код системи буде структуровано відповідно до правил "чистої" архітектури [12] у визначенні та розумінні Роберта Мартіна, що він виклав у однойменній книзі та блозі на своєму сайті. Синонімічними назвами є "цибулинна" або "гексагональна" архітектура, або більш технічно – багатошарова архітектура.

Ключовими принципом чистої багатошарової архітектури є розділення системи на шари, ізолюючи бізнес-логіку від інтерфейсів (наприклад, UI, баз даних).

Такі системи мають наступні переваги:

- незалежність від фреймворків, що дає змогу використання бібліотеки та інструменти, не обмежуючи можливість їх заміни в майбутньому;

- тестованість – бізнес-логіку можна тестувати без UI, баз даних чи зовнішніх сервісів;

- замінність UI, тобто можливість підтримки різних типів інтерфейсу, наприклад веб (HTTP API або SPA), десктопний, мобільний або консоль застосунки, не впливає на ядро системи;

- гнучкість СУБД, завдяки чому можна легко перейти від використання SQL до NoSQL або навіть гетерогенного рішення без змін у бізнес-правилах.

На концептуальній діаграмі на рисунку 2 показано інтеграцію цих ідей з акцентом на правило залежностей. Залежності коду мають спрямовуватися всередину, внутрішні шари (політики) не можуть посилатися на зовнішні (механізми). Формати даних із зовнішніх шарів (наприклад, згенеровані фреймворками) не повинні впливати на внутрішні.

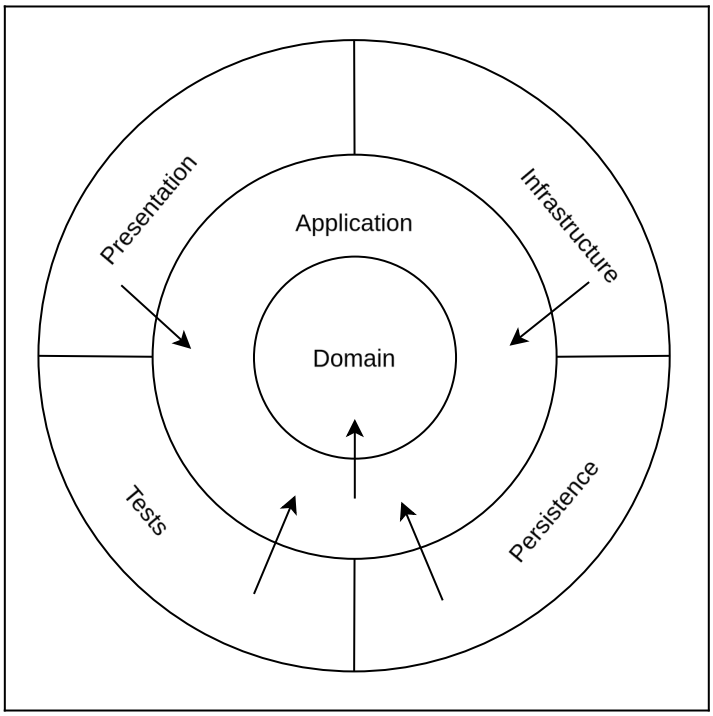

Рисунок 2 – Концептуальна діаграма архітектури програмного коду [12]

Шари системи:

- сутності (Domain) інкапсулюють універсальні бізнес-правила системи, не залежать від зовнішніх змін;

- сценарії використання (Application) реалізують бізнес-логіку системи, здійснюючи певні операції над сутностями та керуючи потоком даних між сутностями та інтерфейсами зовнішніми системами;

- зовнішні шари із інструментами (бази даних, веб-фреймворки) має містити мінімум коду, тільки інтеграції.

У місцях перетину меж використовуються інтерфейси (принцип інверсії залежностей із принципів SOLID), тобто внутрішні рівні використовують визначають інтерфейси, а зовнішні їх реалізують, що робить систему модульною. Дані між рівнями передаються в простих, не специфічних до СУБД або фреймворків форматах (інкапсулюються в об'єкти), щоб уникнути порушень правила залежностей.

Більш формально архітектура програмного коду системи показана на діаграмі пакетів на рисунку 3.

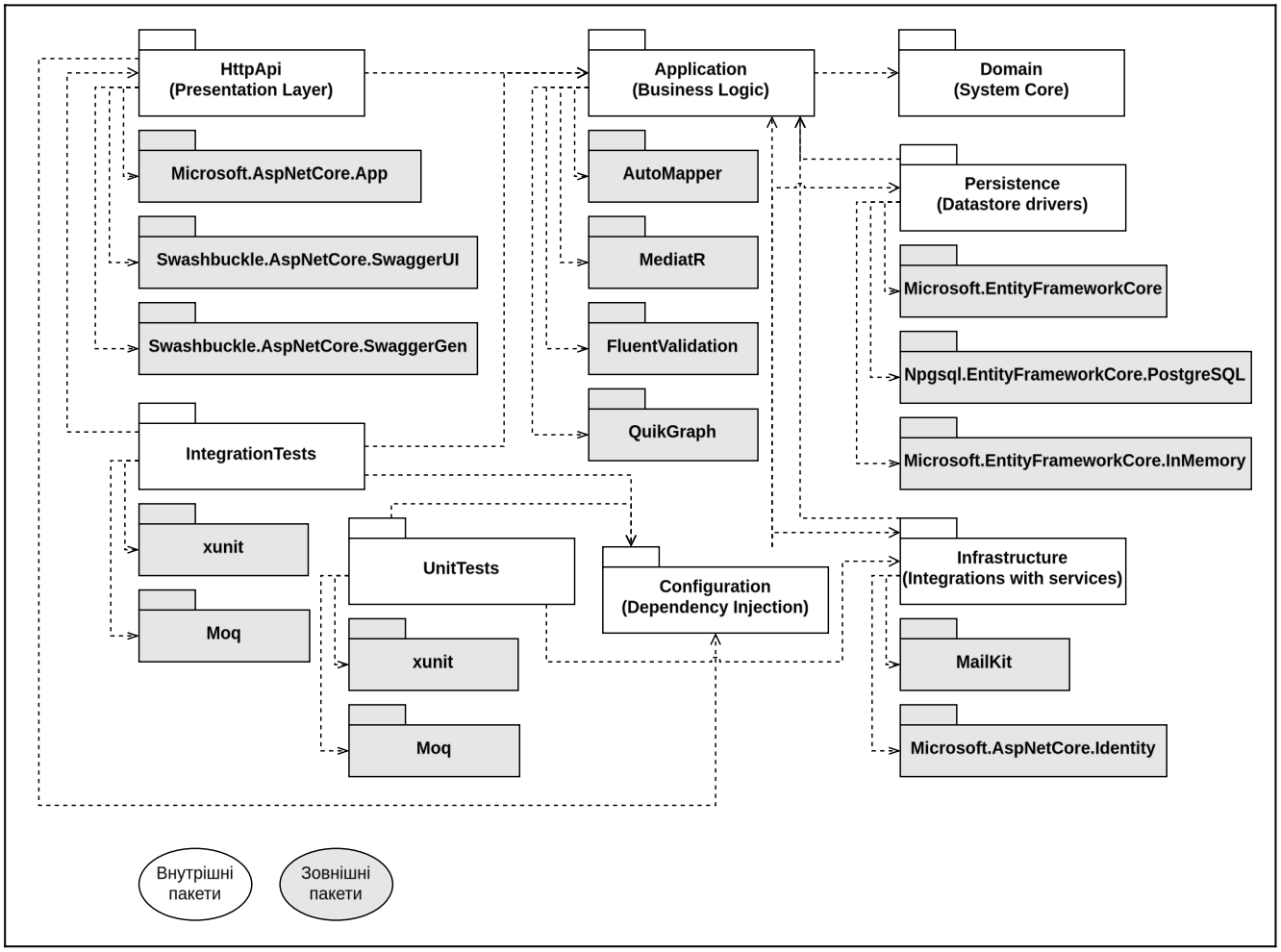

Рисунок 3 – Діаграма пакетів програмного коду застосунку

Для стислості даних з імені кожного внутрішнього пакету була прибрана частина, що вказує на організацію та назву проєкту, тобто наприклад повна назва пакету Application – ua.nure.pzpi-21-4.danylo-nazarko.TravelGuide.Application.

Зовнішні пакети є орієнтовними, на їх вибір вплинули вимоги системи, особистий досвід реалізації подібного функціоналу в минулому та порівняння результатів пошуку.

3.2.2 Архітектура розгортання

На рисунку 4 показана діаграма розгортання системи. Загалом таку структуру можна використовувати як при локальному (on-premise) розгортанні, так і в хмарі.

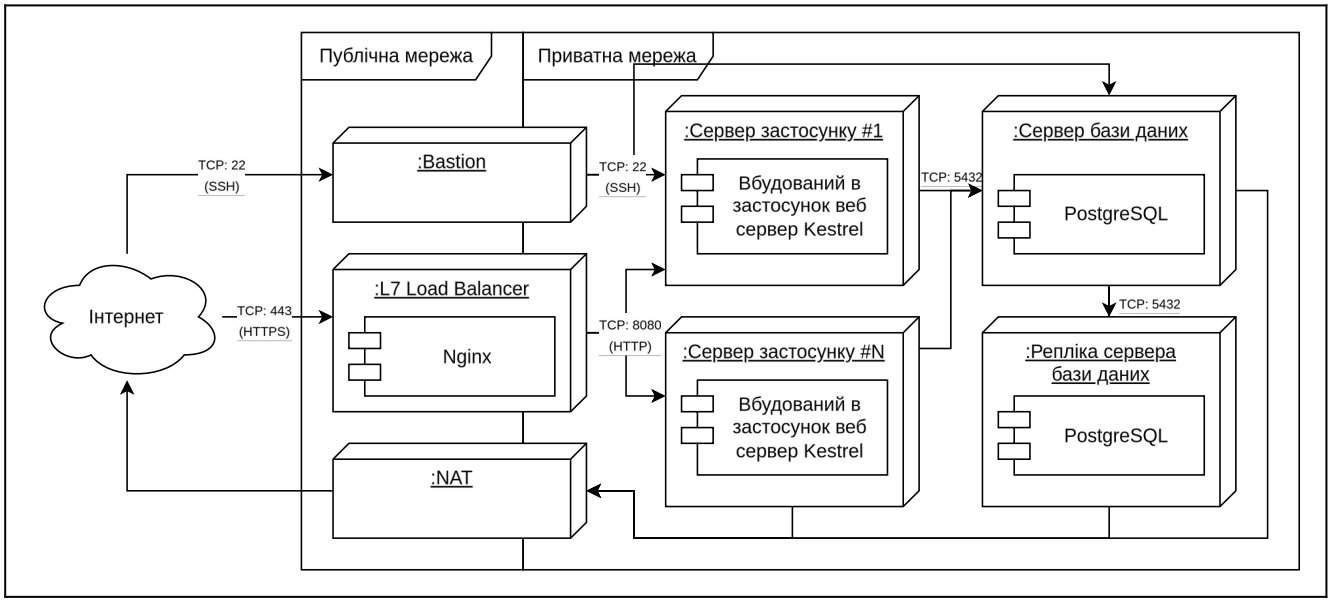

Рисунок 4 – Діаграма розгортання системи

Приватну мережу та вузли, що знаходяться в ній, можна склонувати, створивши декілька середовищ, наприклад по одному для розробки, тестування та експлуатації.

При розгортанні в хмарі вузли балансувальника навантаження та серверу бази даних можна замінити керованими сервісами (managed DB and LB) – сервіси від постачальника хмарних послуг, на підтримку яких витрачається менша кількість зусиль, ніж на локально розгорнені.

Для кожного вузла мають бути налаштовані групи безпеки або правила міжмережевого екрану. Вони мають приймати мінімальну кількість трафіку, що потрібен для функціонування системи, наприклад сервер застосунку приймає TCP трафік на порту 8080 від адреси балансувальника навантаження та TCP трафік на порту 22 від адреси бастіону. За таким же принципом відкрити доступ для програм та агентів нагляду за інфраструктурою та централізованого логування, ранерів систем неперервної інтеграції та розгортання, наприклад сервер для збирання метрик та експортери Prometheus, сервер для збирання логів Grafana Loki та панель для візуалізації Prometheus.

Шифрування трафіку при обміні даними всередині приватної мережі не є обов'язковим, якщо правила міжмережевих екранів налаштовані належним чином.

Для отримання та автоматичного оновлення TLS сертифікатів для балансувальника навантаження можна скористатися послугами безкоштовного сервісу LetsEncrypt. В хмарі це реалізується за допомогою managed сервісу, а локально за допомогою налаштування періодичного оновлення з використання скрипту certbot та системи планування cron.

В рамках цієї роботи використовуватиметься платформа локальної віртуалізації Proxmox Virtual Environment із задіюванням LXC контейнерів, бо вони використовують менше ресурсів гіпервізору порівняно з віртуальними машинами під керуванням KVM/QEMU, за рахунок використання ядра системи гіпервізору. Розгортання вузлів (host provisioning) відбуватиметься за допомогою інструменту "інфраструктура як код" Terraform, а розгортання програм – із використанням платформи автоматизації Ansible та контейнерів Docker. Завдяки такому підходу код систем Terraform та Ansible слугуватимуть документацією інфраструктури

3.3 Проєктування структури зберігання даних

Дані, що потрібні для функціонування системи, зберігатимуться в реляційні БД, ER-діаграма якої показана на рисунку 5.

Відношення Країна, Регіон, Місто, Адреса ієрархічно відображають місцевість. Адреси групуються в маршрут з указанням послідовності. Транспортні (vehicles) засоби прикріплюються до маршруту – створюється поїздка (vehicle_enrollments).

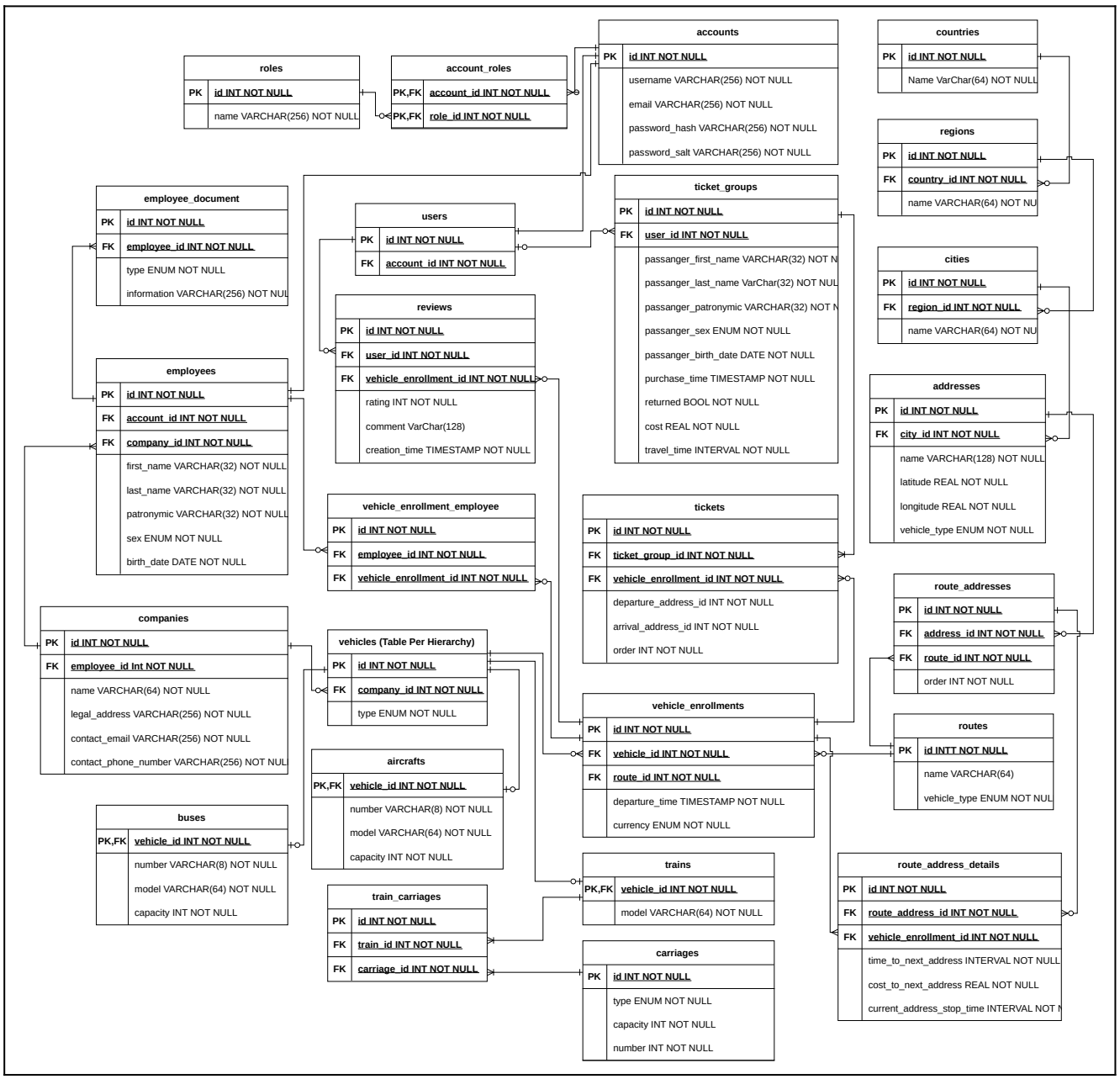

Рисунок 5 – ER-діаграма бази даних

Для можливості пошуку квитків не лише від початкової до кінцевої станції (адреси) поїздки, кожній адресі в поїздці час пересування та ціну від цієї до наступної станції, а також час зупинки на цій станції. При пошуку квитків ці дані сумуються і розраховується час та ціна поїздки. Для зменшення навантаження на систему, наприклад коли користувач зробить запит на отримання даних про квиток, була проведена денормалізація – в таблицю "ticket_groups" записуватимуться підраховані загальна ціна та час подорожі. Під час проведення експериментів із використанням PostgreSQL зі спрощеною схемою, що інкапсулює

процес пошуку маршрутів та показана на рисунку 6, було виявлено, що швидкість отримання раніше зазначених зросла в 30 разів. Доцільність введення такої денормалізації засноване на міркуваннях, що перегляд даних квитка користувачем є часто виконуваним запитом.

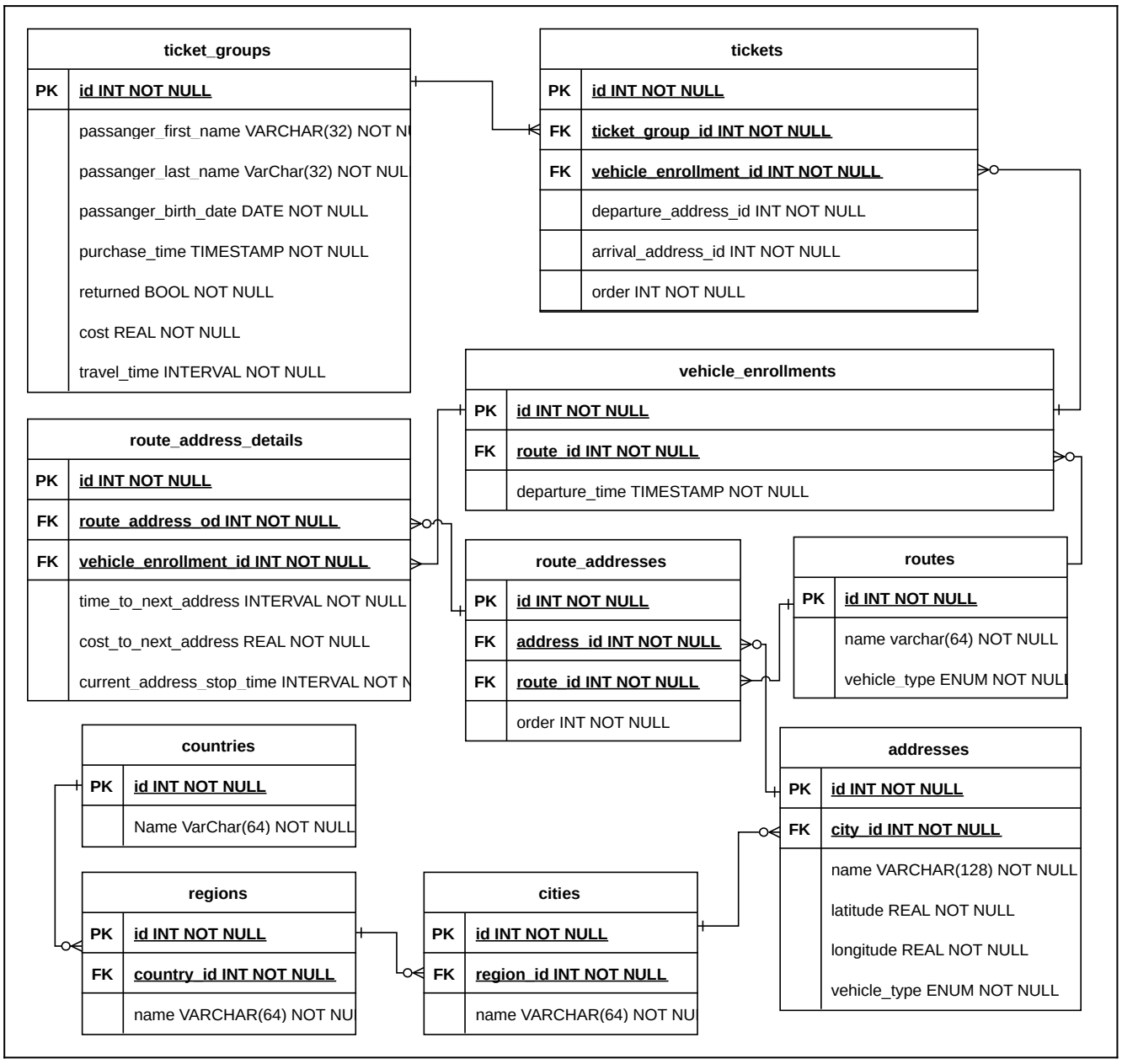

Рисунок 6 – Спрощена ER-діаграма БД, що містить потрібні сутності для логіки пошуку маршрутів та продажу квитків

Для підтримки обліку квитків на поїздки з пересадками будо створено два відношення: "tickets" та "ticket_groups". В першому зберігається інформація про

одну поїздку, друге згруповує декілька перших – це те, як будуть бачити квиток користувачі.

Для уніфікації даних під час пошуку маршруту з пересадками, облік транспортних засобів зроблений у виді ієрархії наслідування. Хоча в схемі показано по одному відношенню для кожного виду транспорту, насправді буде використовуватись підхід "таблиця на ієрархію", тобто поля з даними всіх транспортних засобів зберігатимуться в одній таблиці, а програма буде вибирати потрібні спираючись на значення дискримінатора type, що вказуватиме на тип транспортного засобу. Цей метод був обраний через простоту реалізації та швидкість отримання даних порівняно з "таблиця на тип" та "таблиця на конкретний тип".

Через специфіку розроблюваної системи, також була створена спрощена схема графової бази даних, що показана на рисунку 7.

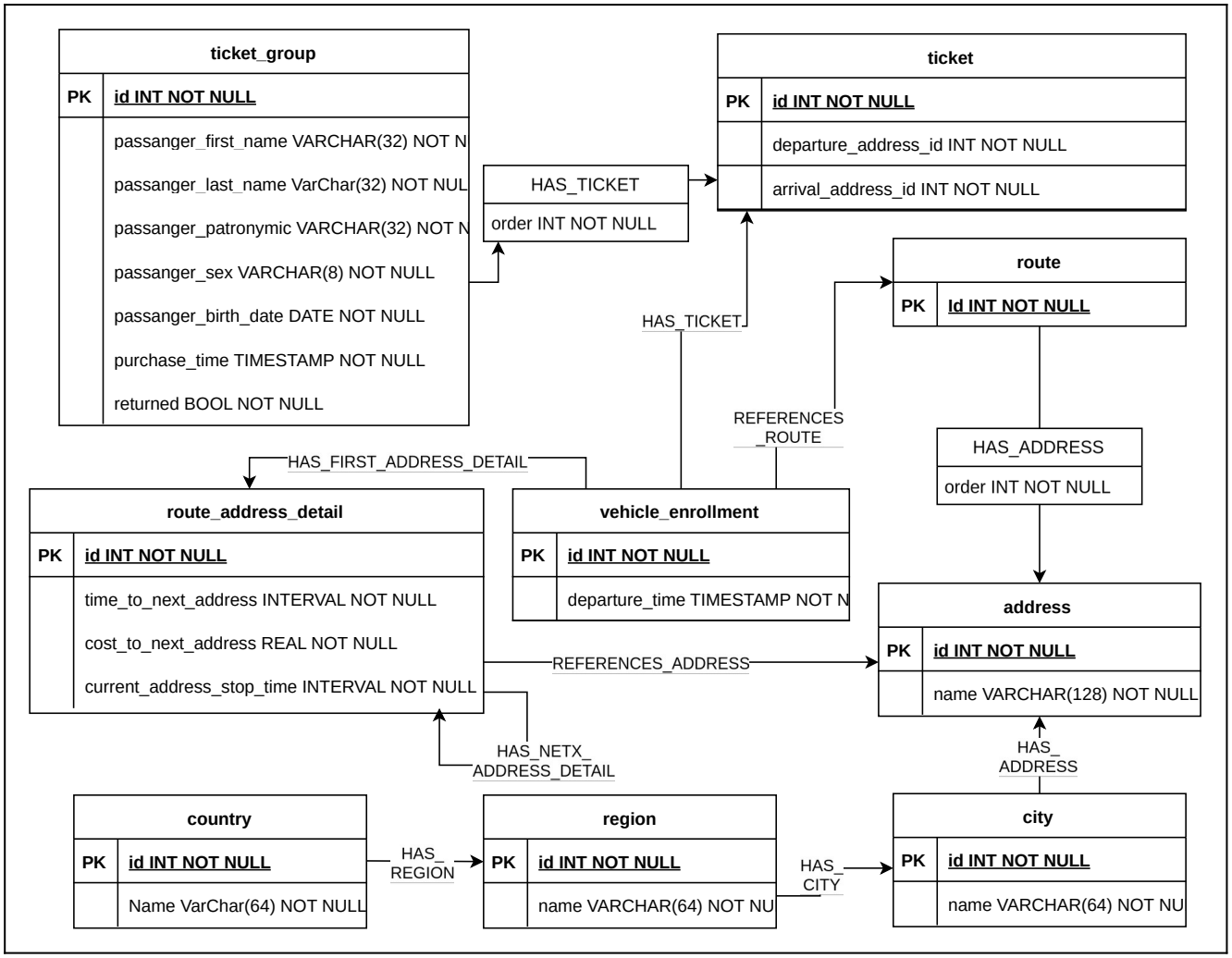

Рисунок 7 – Спрощена схема графової БД

Маючи схеми реляційної та графової баз даних, можна визначити області, що доречно буде реалізувати в тому чи іншому типі БД. Таким чином була отримана спрощена діаграма гетерогенної бази даних, що показана на рисунку 8. Таблиці, дані яких використовуються при пошуку маршруту були винесені в графову БД, що дозволить спростити алгоритм шляхом використання вбудованих в графову СУБД методів, інші відношення залишились в реляційній.

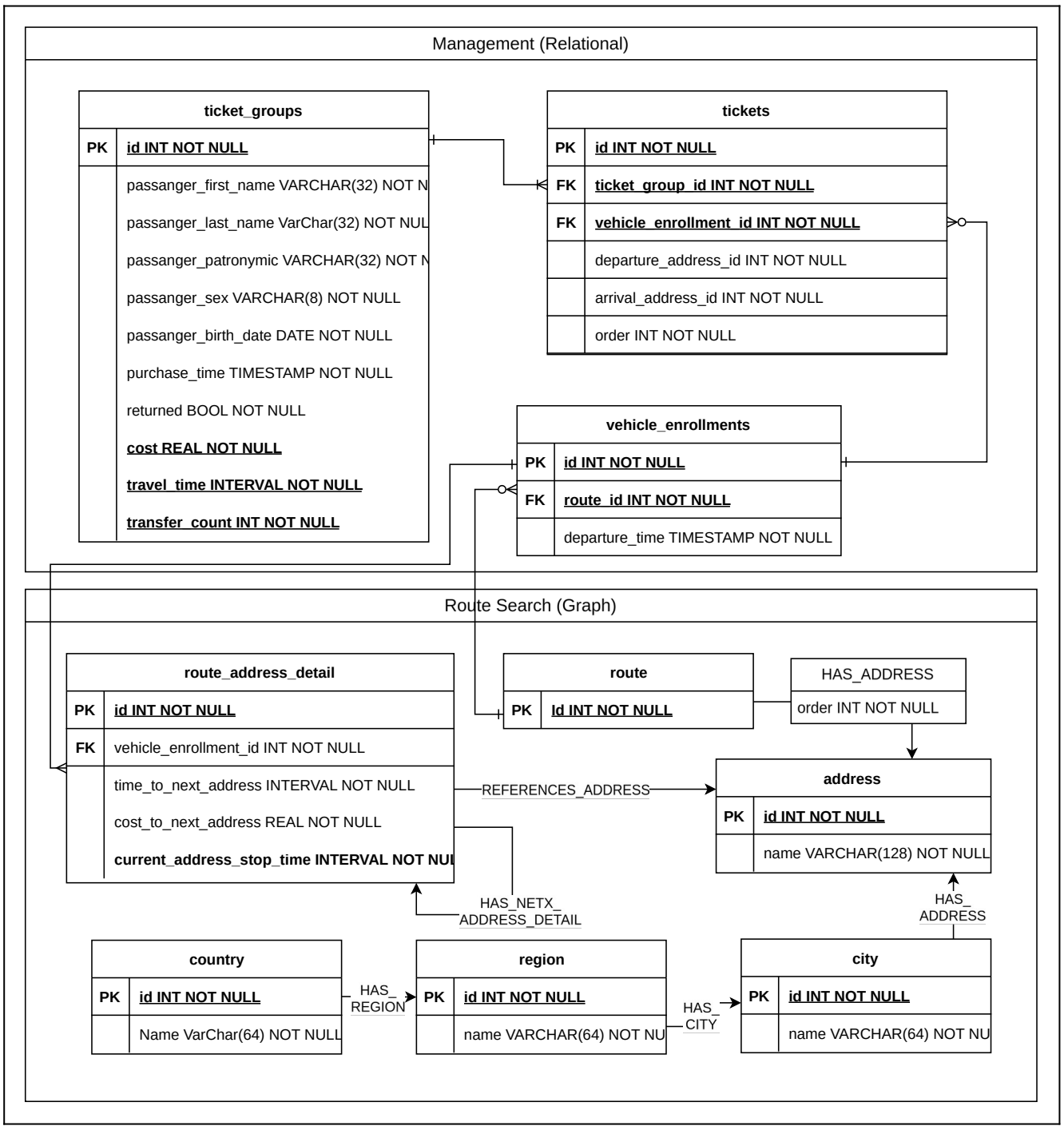

Рисунок 8 – Спрощена схема гетерогенної БД

3.4 Приклади найцікавіших алгоритмів та методів

Центральним алгоритмом системи стане процес пошуку маршруту з пересадками, тому він був описаний у вигляді діаграми потоку даних, що показана на рисунку 9.

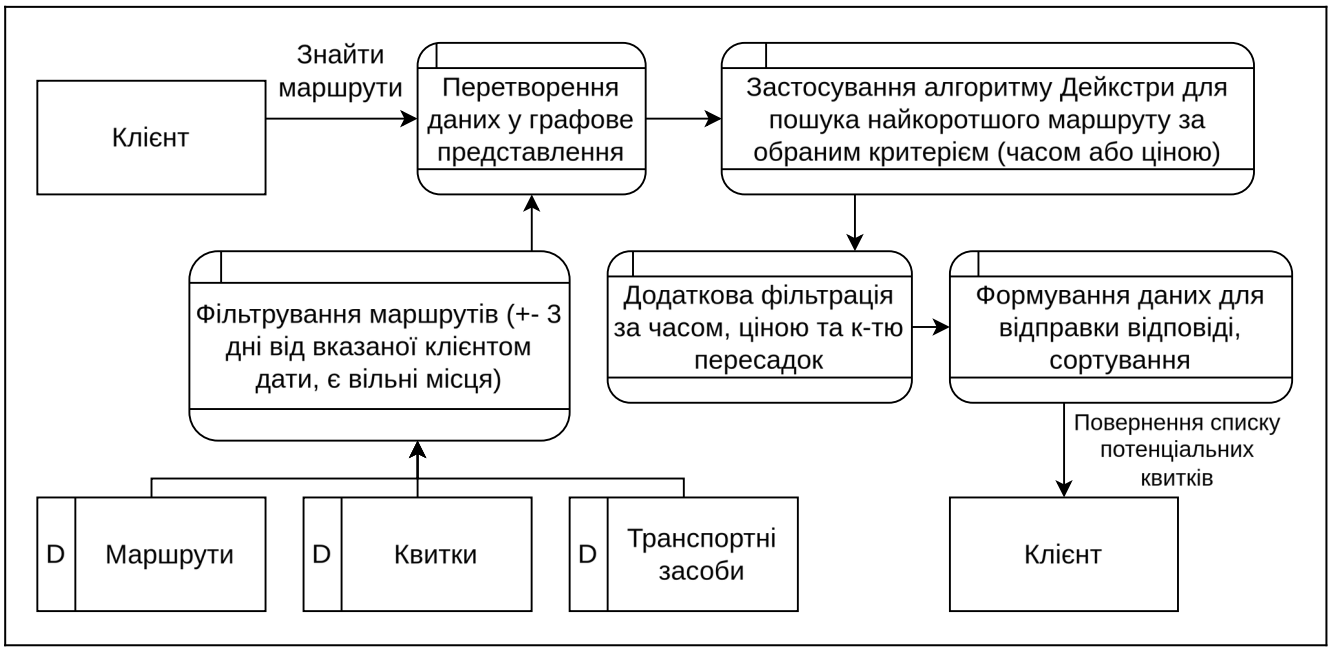

Рисунок 9 – Діаграма потоку даних процесу пошуку маршруту

Для збільшення вірогідності знаходження маршруту, в початкові дані включаються поїздки, дата відправлення яких знаходиться в проміжку трьох днів від заданої користувачем. Після перетворення в графову об'єктну модель застосовуватиметься алгоритм Дейкстри для пошуку найкоротшого шляху за обраним користувачем критерієм – часом (найшвидші) або ціною (найдешевші). Вершини графу (зупинки, адреси) вважатимуться рівними, коли відстань без врахування рельєфу місцевості між ними, що розраховуватиметься за географічними координатами, буде менша за вказану користувачем.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Мова програмування

Серед популярних мов, що використовуються для створення серверних частин застосунків, таких як: Python (Django/FastAPI), JavaScript/TypeScript (Node.js), Java (Spring), Go та C# (.NET), останній виділяється завдяки своїй збалансованості та інтеграції з сучасними технологіями.

Однією з ключових переваг .NET є висока продуктивність розробки завдяки строгій типізації та сучасному синтаксису. Це дозволяє уникнути багатьох помилок на етапі компіляції. На відміну від інтерпретованих мов, таких як Python або JavaScript, компільований код C# забезпечує кращу швидкодію, що робить його ідеальним для високонавантажених систем.

.NET також пропонує багату екосистему з великою кількістю бібліотек та фреймворків, які спрощують розробку складних функцій, таких як автентифікація, робота з базами даних або обробка запитів. Відповідно до вимог та тенденцій сучасності більшість інструментів є крос-платформенними, що дозволяє розгортати розроблені програми на різних операційних системах та архітектурах процесору.

Ще одна перевага .NET – це підтримка корпорації Microsoft, яка забезпечує довгострокову стабільність, регулярні оновлення та безпекові виправлення. Для продуктового проєкту, який має жити роками, це критично важливо. На відміну від деяких open-source рішень, де підтримка може бути непередбачуваною, .NET гарантує стабільність і сумісність між версіями.

Крім того, C# – це мова, яка постійно розвивається, інтегрує сучасні практики та технології, такі як: records, pattern matching, async/await та інші. Це робить код більш виразним і легшим для підтримки. У порівнянні з Java, синтаксис C# часто виходить більш лаконічним, а у порівнянні з Go – більш гнучким завдяки підтримці ООП та функціональних підходів.

4.2 Шаблони об'єктно-орієнтованого програмування

4.2.1 Посередник (Mediator)

Шаблон проектування "Посередник" спрощує взаємодію між об'єктами, усуваючи пряме посилання одних об'єктів на інші. Замість цього вони спілкуються через централізований об'єкт-посередник, який інкапсулює логіку взаємодії. У розробленому застосунку для реалізації цього патерну була використана бібліотека MediatR, що надає зручний спосіб організації команд, запитів та подій.

У MediatR основна концепція полягає у відокремленні запитів (або команд) від їх обробників. Команда – це об'єкт, що інкапсулює дані, необхідні для виконання певної дії. Обробник команди – це клас, що отримує команду та виконує відповідну логіку.

В додатку Б.1 показані команда AddAddressCommand, що містить дані про адресу, та її обробник AddAddressCommandHandler. MediatR автоматично знаходить відповідний обробник для команди та викликає його метод Handle.

Також бібліотека дозволяє додавати конвеєри обробки – додаткові кроки, які виконуються до або після основного обробника.

В додатку Б.2 наведений конвеєр LoggingBehavior, що журналює інформацію про те, яка команда обробляється. Він викликається перед основним обробником і після нього. Також він виводить попередження, якщо команда виконувалась довше 500 мілісекунд.

Також в застосунку присутній конвеєр для валідації параметрів об'єкта та викиду винятків, якщо вони не задовольняють вимогам.

4.2.2 Сховище (Repository)

Шаблон проектування "Сховище" використовується для інкапсуляції логіки доступу до даних, відокремлює бізнес-логіку від деталей роботи з джерелом даних. Він надає абстрактний інтерфейс для виконання CRUD-операцій (Create,

Read, Update, Delete), що спрощує тестування та заміну джерела даних без змін у бізнес-логіці.

Головна мета Repository – приховати складність роботи зі сховищем даних, наприклад, файловою системою, базою даних або зовнішнім API за простим інтерфейсом. Замість того, щоб розкидати запити до бази даних по всій програмі, ми зосереджуємо їх у спеціальних класах-репозиторіях. Це дозволяє легко змінювати джерело даних, наприклад, перейти з SQL Server на PostgreSQL або додавати кешування без змін у бізнес-логіці.

У додатках Б.3, Б.4, Б.5 та Б.6 показані узагальнений (Generic) інтерфейс репозиторію, абстрактна реалізація для взаємодії з СУБД PostgreSQL із використання об'єктно-реляційного мапера Entity Framework, конкретного інтерфейсу та класу репозиторію відповідно.

4.2.3 Одиниця роботи (Unit of Work)

Шаблон "Одиниця роботи" використовується для керування транзакціями та групування операцій з даними у єдиний атомарний блок. Він дозволяє відкладати збереження змін у сховище даних до моменту явного підтвердження, що спрощує управління цілісністю даних і підвищує ефективність.

Головна мета Unit of Work – інкапсулювання операцій зі сховищем даних (додавання, оновлення, видалення) у єдиний контекст, який можна зафіксувати однією транзакцією. Це особливо корисно у складних сценаріях, де кілька репозиторіїв мають оновлювати дані атомарно. У створеній системі прикладом таких операцій є створення, оновлення та видалення інформації про працівників компаній, де одночасно треба змінити дві сутності: робітник та документ, що у випадку з реляційними СУБД будуть знаходитись в різних відношеннях.

У додатках Б.7 та Б.8 показані інтерфейс та реалізація класу "Одиниці роботи". Для скорочення прикладів наведений лише репозиторій адрес. У програмній системі задіюються усі репозиторії сутностей бізнес логіки.

4.3 Асинхронність

Для підвищення продуктивності застосунку в багатьох модулях було використано шаблон асинхронного програмування TAP (Task-based Asynchronous Pattern). Він базується на типі Task та ключових словах async, await. Він дозволяє ефективно виконувати довготривалі операції, наприклад, операції запису та читання з диску, мережеві запити, без блокування основного потоку. На відміну від застарілих патернів APM (Asynchronous Programming Model) та EAP (Event-based Asynchronous Pattern), TAP є більш простим, гнучким і інтегрованим у сучасні фреймворки .NET.

Асинхронний метод, який не повертає значення, має тип Task, а якщо повертає – Task<T>, де T – тип значення, що повертає метод.

Ключове слово async позначає метод як асинхронний, а await перед викликом асинхронного методу – призупиняє його виконання до завершення задачі без блокування потоку.

TAP підтримує переривання операцій за допомогою токену скасування. Комплексна операцію реалізована у вигляді асинхронного методу між виконанням дій може перевіряти токен скасування та, якщо він буде сигналізувати, що операцію потрібно закінчити, зробити усі необхідні дії для коректного завчасного завершення виконання.

В додатку Б.9 показане створення колекції задач конвертації ціни квитків, що будуть виконуватись асинхронно. Спеціальний метод Task.WaitAll очікує, коли всі задачі виконаються перед продовженням виконання основного методу.

4.4 Передача інформації між шарами системи

Для передачі інформації між шаром представлення та бізнес логіки системи використовуються 3 типи POCO (Plain Old C# Object) класів, що інкапсулюють дані потрібні для відповідного шару. Шар представлення під час отримання інформації від користувача збирає їх в моделі представлення (View Models), з яких формуються команди та запити для шару бізнес логіки. Після обробки дані

повертаються від шару бізнес логіки до шару представлення у вигляді DTO (Data Transfer Object). Такий підхід дозволяє чітко розмежувати шари та підвищує захищеність системи, не дозволяючи внутрішнім даним, наприклад хешам паролів, бути показаними користувачу.

4.5 Інтернаціоналізація

4.5.1 Мови та форматування

З шару представлення до системи потрапляє тег мови в форматі зазначеному в rfc5646 [13]. В розробленій програмній системі це відбувається через встановлення HTTP заголовку "Accept-Language" до запиту. Тег мови складається з головного тегу – дво- або трьохзначного коду мови відповідно до ISO 639 [14] та підтегу – дво- або трьохзначного коду країни відповідно до ISO 3166 [15]. Головний тег вказує на мову, якою має бути показана інформація, а підтег – на стандарт форматування, що буде застосований до чисел, дат та часу. Наприклад, тег "en-US" вказує на англійську мову та форматування відповідно до стандартів США, а "uk-UA" – на українську мову та стандарт форматування.

Мова програмування C# підтримує зазначений формат тегів за допомогою класу System.Globalization.CultureInfo. Разом із рядком форматування значення CultureInfo передається в статичний метод Format класу String, що повертає рядок із відформатованими відповідно до вказаного регіону числами та датами.

Рядок форматування отримується з екземпляру класу, що реалізує інтерфейс IStringLocalizer. Він отримує значення мови з CultureInfo та за заданим ключем повертає рядок написаний відповідною мовою. IStringLocalizer отримує локалізовані рядки з файлів у форматі json. Їх назва збігається з тегом мови. Реалізація валідатора даних команд, що використовує зазначені методи для локалізації повідомлень про помилки та приклад файлу з локалізованими рядками показані в додатках Б.10 та Б.11 відповідно.

Розроблена система підтримує лише дві мови: англійську та українську, але може буди розширена для підтримки інших без змін в програмному коді. Якщо

запит не міститиме тегу або його значення не підтримуватиметься системою, у відповідь буде повернено локалізацію англійською мовою (en-US).

4.5.2 Часові пояси

З шару представлення до системи потрапляє тег часу, що має відповідати одному з перелічених в базі даних часових поясів IANA [16]. В розробленій програмній системі це відбувається через встановлення HTTP заголовку "Accept-TimeZone" до запиту. Зазвичай тег має наступний формат: "Регіон/Місто", наприклад, "America/New_York" або "Europe/Kyiv". Ці значення інкапсулюють в собі часовий пояс та дати початку та закінчення літнього часу.

Мова програмування C# підтримує зазначений формат тегів за допомогою класу System.TimeZoneInfo. Він дозволяє отримувати інформацію про доступні часові пояси, конвертувати час між ними та обробляти правила переходу на літній/зимовий час.

Клас System.DateTimeOffset представляє дату та час разом із зсувом (offset) від UTC (Coordinated Universal Time). Це дозволяє точно визначати момент часу, незалежно від часового поясу.

Поєднуючи ці два класів, розроблена програмна система перетворює значння часу на локальні для клієнта під час формування результуючих даних запиту. Якщо запит не міститиме тегу, система поверне час у нульовому часовому поясі (UTC +00:00).

4.5.3 Валюти

Усі об'єкти моделі даних, що зберігають ціну, мають поле, що вказує на валюту. В розробленій програмній системі таке поле є в сутностях vehicle_enrollment та tciket (рисунок ERD). Значення цих полів відповідають кодам валют відповідно до ISO 4217 [17].

Код валюти потрапляє до системи з шару представлення через встановлення HTTP заголовку "Accept-Currency". Він перетворюється на екземпляр

класу-перелічення, що інкапсулює валюти: трьохзначний код, номер та кількість дробових розрядів. Він наведений в додатку Б.12.

Для конвертації між валютами був створений клас-сервіс, що використовує безкоштовний HTTP API для отримання курсу. Під час формування результуючих даних запиту значення цін конвертується з валюти, в якій вони були збережені, у валюту, що запитав клієнт. Якщо запит не міститиме значення валюти, система поверне ціну в валюті, збереженій в сховищі даних.

4.6 Фреймворк шару представлення

Для створення шару представлення був застосований ASP.NET – сучасний фреймворк від Microsoft для розробки веб-застосунків та API. Він підтримує різні підходи до створення серверних застосунків, включаючи MVC (Model-View-Controller), Razor Pages та Web API. Однією з ключових переваг ASP.NET є його модульність, висока продуктивність та інтеграція з сучасними стандартами, такими як OpenAPI, Dependency Injection та асинхронне програмування.

4.6.1 HTTP контролери

HTTP контролери в ASP.NET – це класи, які обробляють HTTP-запити, як-от: GET, POST, PUT, DELETE, тощо. Вони містять методи (actions), що відповідають за конкретні маршрути (endpoints). Кожен метод контролера може повертати різні HTTP-статуси, наприклад, 200 OK або 404 Not Found. Атрибути [HttpGet], [HttpPost] та інші визначають тип запиту, який обробляє метод. Контролер для управління адресами в розробленій системі (скорочено для стислості, наведений лише маршрут для додання адрес) показаний в додатку Б.13.

4.6.2 Конвеєр обробки помилок

Конвеєр (middleware) ASP.NET – це послідовність middleware-компонентів, які обробляють HTTP-запити та відповіді. Кожен middleware може модифікувати запит або відповідь. У розробленій програмній системі використовується конвеєр для обробки виключень та формування змістовних відповідей, що показаний у

додатку Б.14 (наведена скорочена версія із обробкою виключень валідації та незапланованих виключень).

4.6.3 Періодичні задачі

Для виконання періодичних або фонових задач у ASP.NET використовуються спеціальні класи, такі як: BackgroundService або HostedService. У додатку Б.15 наведений сервіс, що видаляє квитки, які були зарезервовані більше 10-ти хвилин тому, тобто за які не заплатив користувач.

4.6.4 Документування за допомогою OpenAPI Swagger

Для документування було використано OpenAPI Swagger. Після запуску програми в браузері можна відкрити сторінку з інтерактивну документацію, де відображені усі ендпоїнти програмного інтерфейсу, схеми даних запитів та відповідей, трактування статус кодів, форми для надання параметрів автентифікації, встановлення HTTP заголовків та інше.

Візуалізація веб сторінки створюється на основі файлу формату json, в якому описані усі раніше зазначені деталі. Із цього файлу можливе створення клієнтських заглушок, наприклад для запиту "POST /addresses" інструменти різних мов програмування можуть створити моделі даних та код взаємодії з API автоматично. Це значно прискорює процес інтеграції та зменшує кількість ручної роботи.

4.7 Налаштування

Налаштування програми можна здійснити трьома методами:

- у директорії запуску створити файл в форматі json з параметрами;
- в середовищі запуску встановити змінні оточення;
- встановити значення параметрів в команді запуску.

Такий підхід дозволяє гнучко поставитися до процесу розгортання – не залежно від методу (самостійний запуск, створення сервісу системи ініціалізації,

використання контейнерів Docker) можна налаштувати всі параметри будь-яким найбільш зручним способом.

4.8 Логування

В застосунку присутні два формати виводу логів: json та plain text. В залежності від вимог до розгортання може буди використаний той чи інший метод.

Логування у форматі json надає структурований підхід до зберігання інформації, що значно полегшує її обробку та аналіз. Кожен запис логу містить чіткі поля, такі як час події, рівень логування та повідомлення, що дозволяє швидко фільтрувати та шукати потрібні дані.

Звичайні текстові логи, хоча й більш звичні для читання людиною, часто потребують додаткових зусиль для парсингу. Наприклад, якщо лог-рядок містить різноманітну інформацію, його доводиться розбивати за допомогою регулярних виразів або інших методів обробки текстів.

4.9 Взаємодія з СУБД

Для взаємодії з базою даних було використано Entity Framework – об'єктно-реляційний мапер (ORM) від Microsoft, що взаємодіє з реляційними базами даних, використовуючи об'єкти, без необхідності писати SQL-записи вручну. Для синхронізації моделей даних з базою даних був застосований підхід розробки на основі коду (Code First), де структура бази даних створюється на основі класів моделі.

Для збереження гнучкості в налаштуванні БД (встановлення типів полів, створення процедур, тригерів та ін.) була використано Fluent Configuration, тобто спосіб налаштування проєктування класів на структури БД за допомогою спеціального API в коді.

Для застосування налаштувань використовуються міграції – поступові оновлення схеми БД у відповідності до змін у моделях. Кожна міграція представляє собою окремий крок, який містить код для застосування та відкату

змін. Це дозволяє легко синхронізувати базу між різними середовищами (наприклад, розробки, тестування та продакшену) та відстежувати історію змін.

4.10 Пошук маршрутів з пересадками

Для пошуку маршруту з пересадками застосовувались два алгоритми: Дейкстри – для знаходження одного найкоротшого маршруту за заданим критерієм та пошуку в ширину – усіх маршрутів для подальшого сортування та фільтрації.

Код алгоритму пошуку всіх маршрутів з пересадками показаний у додатку Б.16. На початку алгоритму створюється список paths, який буде містити всі знайдені шляхи у вигляді послідовностей ребер. Також ініціалізується черга queue, що допомагає організувати обхід графа. Кожен елемент у черзі містить поточне ребро та шлях, який до нього привів.

Спочатку у чергу додаються всі ребра, що виходять з початкової вершини departureAddress. Для кожного такого ребра створюється новий шлях, який поки складається лише з цього ребра. Далі алгоритм входить у цикл, який продовжує роботу, поки черга не стане порожньою.

На кожній ітерації циклу з черги вилучається поточний елемент. Якщо ціль ребра (current.edge.Target) збігається з кінцевою вершиною arrivalAddress, знайдений шлях додається до списку paths, і алгоритм переходить до наступної ітерації. У протилежному випадку для всіх ребер, що виходять з поточної вершини, виконується перевірка: чи вже це ребро присутнє у поточному шляху. Якщо ні, створюється новий шлях, який включає всі попередні ребра та поточне ребро-сусід, і цей шлях додається у чергу для подальшого дослідження.

Таким чином, алгоритм ретельно досліджує всі можливі маршрути, уникаючи циклів (завдяки перевірці на наявність ребра у шляху), і збирає всі шляхи, що ведуть від початкової до кінцевої вершини.

5 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У розробленій програмній системі застосовуються тест-фреймворк xUnit.v3, що забезпечує зручний спосіб опису тестових випадків та Moq для ізоляції тестів від зовнішніх залежностей. Інтеграційні тести перевіряють взаємодію між різними компонентами, такими як сервіси автентифікації, обробники команд (MediatR) та доступ до даних. Для цього використовується базовий клас TestBase, який налаштовує тестовий середовище, імплементує заміни сервісів ("моки") та забезпечує ізоляцію тестів.

Клас TestBase ініціалізує колекцію сервісів IServiceCollection, де конфігуруються основні модулі застосунка такі, як: логування, робота з даними та бізнес-логіка. Для тестування використовується in-memory база даних, що дозволяє уникнути залежності від зовнішніх систем. Також тут налаштовуються культура (en-US) та часовий пояс (Europe/Kyiv), щоб забезпечити консистентність тестів.

Наприклад, метод SetAuthenticatedUserRoles (додаток Б.17) створює "мокований" сервіс SessionUserService, який імітує автентифікованого користувача з певними ролями. Це дозволяє тестувати сценарії, де доступ до певних операцій обмежений правами.

У класі CountriesTests перевіряються операції додавання, оновлення та видалення країн. Наприклад, тест AddCountry_WithAdminRole_CountryCreated (Б.18) перевіряє, що адміністратор може успішно додати нову країну. Після виконання команди AddCountryCommand відбувається запит до бази даних через GetCountryQuery, і перевіряється, що назва країни збереглася коректно.

Інший тест (додаток Б.19) перевіряє, що спроба додати країну з дубльованою назвою призводить до винятку DuplicateEntityException. Це демонструє, що бізнес-логіка коректно обробляє унікальність даних.

Система має суворі правила авторизації. Наприклад, звичайний користувач не може додавати нові країни. Це перевіряється в тесті

AddCountry_WithUserRole_ThrowsForbiddenException, де очікується виняток ForbiddenException. Аналогічно, неавтентифікований користувач не має доступу до операцій модифікації даних, що підтверджується тестом AddCountry_WithUnAuthenticatedUser_ThrowsUnAuthorizedException.

Теоретичні тести (із анотацією [Theory]) використовуються для перевірки валідації вхідних даних. Наприклад, метод AddCountry_WithInvalidName_WithAdminRole_ThrowsValidationException (додаток Б.20) перевіряє, що назва країни не може бути порожньою або довшою за 64 символи (граничні випадки). Якщо користувач з будь-якими правами доступу спробує ввести некоректні дані, система викине ValidationException.

Подібні методи застосовуються для тестування всіх функцій застосунку.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було проаналізовано предметну галузь за обраною темою, окреслено вимоги, розроблено документацію та створено програмну систему для планування мультитранспортних поїздок із пересадками та продажу квитків. Система інтегрує дані різних транспортних компаній, автоматизує управління ресурсами та забезпечує користувачам зручний інструмент для організації подорожей. Актуальність розробки обумовлена зростанням попиту на ефективні рішення, які поєднують різні види транспорту, та відсутністю інтеграції між існуючими платформами.

Важливим аспектом є підвищення ефективності роботи транспортних компаній. Система автоматизуватиме розподіл транспорту, персоналу та розкладів, зменшуючи операційні витрати. Для пасажирів буде створено інтуїтивний інтерфейс з можливістю пошуку маршрутів, візуалізації їх на мапі та отримання інформації про зміни в реальному часі. Тестування підтвердить стабільність роботи системи навіть за високого навантаження.

Розроблена платформа відкриє перспективи для впровадження в транспортній галузі, зокрема серед авіакомпаній, залізничних перевізників та туристичних агенцій. Вона посприяє переходу до інтелектуальних транспортних мереж, де кожен елемент взаємопов'язаний, що підвищить комфорт пасажирів та економічну ефективність перевізників.

Специфікація, вихідний код, відеозапис роботи розробленої програми та презентація завантажені на GitHub [18].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kilani, M. A. & Kobziev, V. Methodology of Data Collection in Information System (IS). In: Current Overview on Science and Technology Research. 2022, Vol. 9, 132–144. DOI: 10.9734/bpi/costr/v9/3712C.

2. ASP.NET documentation. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core (дата звернення: 06.05.2025).

3. PostgreSQL: documentation. PostgreSQL: The world's most advanced open source database. URL: https://www.postgresql.org/docs/ (дата звернення: 11.05.2025).

4. Cormen T. H., Stein C., Rivest R. L. Introduction to algorithms, fourth edition. MIT Press, 2022. 1332 с.

5. Google Maps. URL: https://maps.google.com (дата звернення: 06.06.2025).

6. Rome2Rio. URL: https://www.rome2rio.com (дата звернення: 06.06.2025).

7. Omio. URL: https://www.omio.com (дата звернення: 06.06.2025).

8. Укрзалізниця. URL: https://www.uz.gov.ua (дата звернення: 06.06.2025).

9. FlixBus Україна. URL: https://www.flixbus.ua (дата звернення: 06.06.2025).

10. Official PCI security standards council site. PCI Security Standards Council. URL: https://www.pcisecuritystandards.org/ (дата звернення: 08.05.2025).

11. General data protection regulation (GDPR) – legal text. General Data Protection Regulation (GDPR). URL: https://gdpr-info.eu/ (дата звернення: 12.05.2025).

12. Robert C. M. The clean architecture. Clean Coder Blog. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (дата звернення: 15.05.2025).

13. RFC 5646: tags for identifying languages. IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc5646 (дата звернення: 03.06.2025).

14. ISO 639 – International Organization for Standardization. URL: https://www.iso.org/obp/ui/#iso:std:iso:639:ed-2:v1:en (дата звернення: 03.06.2025).

15. ISO 3166-1 – International Organization for Standardization. URL: https://www.iso.org/obp/ui/#iso:std:iso:3166:-1:ed-4:v1:en (дата звернення: 03.06.2025).

16. Time zone and daylight saving time data. Internet Assigned Numbers Authority. URL: https://data.iana.org/time-zones/tz-link.html (дата звернення: 03.06.2025).

17. ISO 4217 – International Organization for Standardization. URL: https://www.iso.org/obp/ui/#iso:std:iso:4217:ed-8:v1:en (дата звернення: 03.06.2025).

18. GitHub – NureNazarkoDanylo/bachelor. GitHub. URL: https://github.com/NureNazarkoDanylo/bachelor (дата звернення: 03.06.2025).