МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **ОСОБЛИВОСТІ ВИКОРИСТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ДЛЯ ПОБУДОВИ СИСТЕМИ ЗАМОВЛЕНЬ**

Виконав: студент 2 курсу, групи 8.1214-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
О. С. Швець
(ініціали та прізвище)
Керівник доцент А.І. Безверхий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент Директор ТОВ «Дісітел»
П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра __електроніки, інформаційних систем та програмного__ забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри _____Т.В. Критська_
"_01_" __квітня_____2025_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Швецю Олександру Сергійовичу
(прізвище, ім'я, по батькові)

1.Тема роботи __Особливості використання мікросервісної архітектури для побудови системи замовлень_____

керівник роботи _____доцент Безверхий Анатолій Ігорович____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від _28.03. 2025 р. № 361-е__
2. Строк подання студентом кваліфікаційної роботи_____01.12.2025 р.___
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- аналіз літератури та підходів до побудови мікросервісних систем;
- огляд рішень у сфері онлайн-бронювання та інтеграцій з зовнішніми API;
- дослідження організації бізнес-логіки та взаємодії мікросервісів;
- розробка програмного продукту AutoTix (backend, інтеграції, frontend);
- визначення вимог до системи;
- тестування, аналіз результатів і формування рекомендацій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.04.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Вибір теми та формулювання цілей дипломної роботи.	15.04 – 20.04.25	виконано
2	Аналіз літератури та сучасних підходів до мікросервісних систем.	21.04 – 30.04.25	виконано
3	Огляд технологій і інструментів мікросервісної архітектури.	01.05 – 10.05.25	виконано
4	Проєктування системи AutoTix: архітектура та бізнес-процеси.	11.05 – 20.05.25	виконано
6	Реалізація бекенд-сервісів AutoTix.	21.05 – 06.08.25	виконано
7	Інтеграція з API «Укрзалізниці» та кешування в Redis.	01.09 – 30.09.25	виконано
8	Розробка фронтенду на React.	01.10.25	виконано
9	Тестування продуктивності та масштабованості.	02.10 – 31.10.25	виконано
10	Виявлення та усунення проблем інтеграції та безпеки.	01.11 – 25.11.25	виконано
11	Оформлення звіту.	01.12 – 12.12.25	виконано

Студент _____ Швець О.С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Безверхий А.І.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А,
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 92

Рисунків: 22

Таблиць: 1

Лістингів коду: 4

Джерел: 28

Швець О. С. Особливості використання мікросервісної архітектури для побудови системи замовлень: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник А. І. Безверхий. Запоріжжя : ЗНУ, 2020. 92 с.

Метою кваліфікаційної роботи є дослідження сучасних підходів до побудови мікросервісних систем для обробки замовлень та розробка системи AutoTix для бронювання залізничних квитків з інтеграцією офіційного API АТ «Укрзалізниця»[26]. Система побудована на основі мікросервісної архітектури, що забезпечує масштабованість, відмовостійкість та ефективну паралельну обробку запитів користувачів. У роботі розглянуто взаємодію між доменами користувачів, замовлень, платежів та поїздок, реалізацію бізнес-процесів із застосуванням патернів Saga[5], CQRS та event-driven[21], асинхронний обмін повідомленнями через RabbitMQ[13] та кешування даних у Redis з метою зменшення навантаження на зовнішнє API. Також реалізовано фронтенд на базі React, який забезпечує зручний інтерфейс для пошуку поїздок, формування замовлень та здійснення оплати. У результаті розроблено систему AutoTix, яка забезпечує стабільну роботу під навантаженням, зручний користувацький інтерфейс та безпечну обробку замовлень у реальному часі.

Ключові слова: мікросервіси, система замовлень, AutoTix, CQRS, Saga, event-driven, Redis, RabbitMQ, React, інтеграція API.

SUMMARY

Pages: 92

Figures: 22

Tables: 1

Code listings: 4

Sources: 28

Shvets O. S. Features of Using Microservice Architecture for Building an Order Management System: master's qualification thesis in specialty 121 "Software Engineering" / scientific supervisor A. I. Bezverkhyi. Zaporizhzhia : ZNU, 2020. 92 p.

The purpose of this qualification thesis is to study modern approaches to building microservice-based systems for order processing and to develop the AutoTix system for railway ticket booking with integration of the official API of JSC "Ukrzaliznytsia". The system is based on a microservice architecture that ensures scalability, fault tolerance, and efficient parallel processing of user requests.

The thesis examines the interaction between the user, order, payment, and trip domains, the implementation of business processes using the Saga, CQRS, and event-driven patterns, asynchronous message exchange via RabbitMQ[13], and data caching in Redis to reduce the load on external APIs. A frontend application based on React was also implemented, providing a convenient interface for trip search, order creation, and payment processing.

As a result, the AutoTix system was developed, ensuring stable operation under load, a user-friendly interface, and secure real-time order processing.

Keywords: microservices, order management system, AutoTix, CQRS, Saga, event-driven, Redis, RabbitMQ, React, API integration.

ЗМІСТ

ВСТУП 9
РОЗДІЛ 1 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ТА РІШЕНЬ ДЛЯ ПОБУДОВИ СИСТЕМ ЗАМОВЛЕНЬ 15
1.1 Огляд поняття системи замовлень 15
1.2 Проблеми та обмеження традиційних монолітних систем замовлень 17
1.3 Огляд мікросервісної архітектури 19
1.4 Стандарти побудови мікросервісних систем 21
1.5 Аналіз існуючих систем продажу квитків 23
1.6 Узагальнення недоліків існуючих рішень та обґрунтування вибору мікросервісного підходу для AutoTix 27
1.7 Висновки до розділу 1 29
РОЗДІЛ 2 ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ ДЛЯ ПОБУДОВИ МІКРОСЕРВІСНОЇ СИСТЕМИ ЗАМОВЛЕНЬ 30
2.1 Аналіз технологій бекенд-розробки 30
2.1.1 .NET 8 / ASP.NET Core 30
2.1.2 MassTransit / RabbitMQ 30
2.1.3 EF Core + MSSQL 31
2.1.4 Identity + JWT 31
2.2 Аналіз підходів до інтеграції з зовнішніми API 31
2.3 Фронтенд технології 33
2.3.1 React 33
2.4 Способи забезпечення надійності та узгодженості 33
2.4.1 Saga Pattern 34
2.5. Системи контейнеризації та деплою 34
2.6. Висновки до розділу 2 35

РОЗДІЛ 3. ПРОЄКТУВАННЯ ТА РОЗРОБКА МІКРОСЕРВІСНОЇ СИСТЕМИ ЗАМОВЛЕНЬ AutoTix 37
3.1 Архітектура системи AutoTix 37
3.2 Загальна схема мікросервісів 38
3.3 Опис кожного сервісу 40
3.4. Проєктування баз даних 52
3.4.1. Реляційна база даних (SQL) 52
3.4.2. Кешування даних (Redis) 54
3.4.3. Взаємодія між доменами 55
3.5. Реалізація ключових патернів архітектури 59
3.5.1. Saga для процесу бронювання 59
3.5.2. Event-driven обмін 60
3.5.3. CQRS у сервісах 61
3.6 Реалізація інтеграцій з API Укрзалізниці 62
3.6.1 Архітектурні особливості інтеграції 64
3.6.2 Потоки даних у системі 65
3.7 Реалізація фронтенд частини 66
3.7.1. Структура React-проєкту 66
3.7.2 Сторінки авторизації та привітального інтерфейсу 68
3.7.4 Пошук поїздок та станцій 73
3.8 Засоби забезпечення безпеки та надійності 76
3.8.1 Контроль доступу та автентифікація 76
3.8.2 Безпечні інтеграції з зовнішніми API 77
3.8.3 Захист даних, надійність архітектури та гарантія доставлення 77
3.8.4 Моніторинг, журналювання та операційна надійність 78

3.9 Налаштування середовища розгортання 79
3.10 Висновки до розділу 3 81
РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ РОЗРОБЛЕНОЇ СИСТЕМИ AUTOTIX 82
4.1 Перевірка коректності роботи бізнес-процесів 82
4.2 Тестування продуктивності 83
4.3 Аналіз масштабованості 84
4.4 Виявлені проблеми та шляхи їх усунення 86
4.5 Висновки до розділу 4 87
ВИСНОВКИ 88
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 90

ВСТУП

Актуальність теми

Розробка комп’ютерного застосунку для побудови системи замовлень та купівлі залізничних квитків є надзвичайно актуальною з кількох причин.

По-перше, попит на залізничні перевезення у певних напрямках значно перевищує пропозицію. Квитки на популярні маршрути часто розкуповуються протягом перших хвилин після початку продажу. Через це користувачам доводиться постійно стежити за розкладом, витрачати час і зусилля на спроби встигнути придбати квиток у потрібний момент. У багатьох випадках спроба завершується невдачею через перевантаження серверів або повільне інтернет-з’єднання.

По-друге, користувачам не вистачає зручного інструменту, який би автоматично виконував усі дії — від моніторингу наявності квитків до їх миттєвого бронювання або купівлі. Ручне виконання цих дій вимагає постійної присутності, швидкої реакції та терпіння, що не завжди можливо у повсякденному житті.

По-третє, автоматизований підхід до купівлі квитків значно знижує рівень стресу та дозволяє зосередитись на важливіших справах. Користувач лише задає параметри пошуку (напрямок, дату, кількість місць), а вся подальша робота виконується за нього.

Крім того, створення такого застосунку демонструє практичне використання сучасних підходів до розробки програмного забезпечення. Застосування мікросервісної архітектури дозволяє забезпечити масштабованість, надійність і гнучкість системи. Це особливо важливо для сервісів, які повинні обробляти велику кількість запитів у реальному часі.

Загалом, розробка автоматизованого застосунку для купівлі квитків є актуальною темою, оскільки вона вирішує реальну та поширену проблему

багатьох користувачів, підвищує комфорт і ефективність процесу придбання квитків, а також демонструє потенціал сучасних технологій для автоматизації рутинних завдань.

Мета дослідження

Створення та оптимізація комп'ютерного застосунку для автоматичного моніторингу наявності залізничних квитків та їх оперативного придбання з метою підвищення зручності, надійності та ефективності процесу купівлі квитків у користувачів.

Завдання дослідження

Аналіз проблем та існуючих рішень у сфері онлайн-продажу залізничних квитків з акцентом на високонавантажені напрямки. Вивчення доступних підходів, інструментів та технологій для розробки програмного забезпечення, яке дозволяє автоматизувати процес моніторингу та купівлі квитків, з метою вибору оптимального варіанту для реалізації застосунку.

Об'єкт дослідження

Об'єктом дослідження є процес розробки комп'ютерного застосунку для автоматизації моніторингу та придбання залізничних квитків.

Предмет дослідження

Предметом дослідження є технології, методи та інструменти створення програмного забезпечення для моніторингу наявності залізничних квитків і їх автоматичної купівлі з урахуванням високих навантажень та часових обмежень.

Методи дослідження

Теоретичний аналіз існуючих систем моніторингу онлайн-квитків; вивчення технічних рішень для роботи з API, обробки великих обсягів запитів і забезпечення стабільної роботи застосунку; порівняльний аналіз сучасних технологій автоматизації процесів для вибору найбільш ефективних інструментів реалізації функціоналу застосунку.

Наукова новизна одержаних результатів

Наукова новизна проведеного дослідження полягає у розробленні та обґрунтуванні комплексного підходу до автоматизації процесу моніторингу та придбання залізничних квитків у умовах високої динаміки зміни їх доступності. У роботі запропоновано використання мікросервісної архітектури для побудови системи, здатної обробляти значну кількість запитів у реальному часі. Вперше поєднано механізми інтелектуального моніторингу, оптимізовані алгоритми обробки мережевих відповідей та модулі автоматичного бронювання в єдину програмну платформу. Додатково обґрунтовано ефективність застосування адаптивних інтервалів перевірки та оптимізованих стратегій взаємодії з API, що дозволяє мінімізувати затримки під час купівлі квитків у пікові періоди навантаження.

Практичне значення одержаних результатів

Практичне значення одержаних результатів дослідження полягає у створенні комп’ютерної системи, яка здатна автоматично моніторити наявність залізничних квитків і здійснювати їхню купівлю без участі користувача. Це дозволить значно підвищити шанси на вдале придбання квитків у популярних напрямках, зменшити залежність від стабільності інтернет-з’єднання або людського фактору, а також заощадити час користувачів. Розроблене рішення може бути застосоване як окремими користувачами, так і інтегроване в сервіси продажу квитків для підвищення їхньої ефективності. Отримані результати також можуть бути корисними для подальших досліджень у сфері автоматизації онлайн-покупок та розробки високонавантажених систем реального часу.

Апробація одержаних результатів

Основні положення та результати дослідження були апробовані під час тестування експериментального прототипу застосунку на реальних сценаріях моніторингу наявності залізничних квитків. Функціональність системи перевірялася у різних умовах навантаження для оцінки швидкодії, стабільності

та ефективності алгоритмів автоматичної купівлі. Окремі аспекти роботи були представлені та обговорені з фахівцями у сфері розробки програмного забезпечення, що дозволило отримати практичні рекомендації для вдосконалення архітектурних рішень. Крім того, результати апробації підтвердили можливість подальшого масштабування системи та її використання в реальних сервісах онлайн-продажу квитків.[26,]

Глосарій

API (Application Programming Interface) – набір правил і методів, що дозволяють взаємодіяти між різними програмними компонентами. В AutoTix використовується для інтеграції з офіційним API АТ «Укрзалізниця»[26] та між мікросервісами.

Backend (Бекенд) – частина програмної системи, що обробляє бізнес-логіку, роботу з базами даних та інтеграцію з зовнішніми сервісами.

Cache (Кешування) – механізм тимчасового збереження даних для швидкого доступу. У AutoTix використовується Redis для зменшення кількості запитів до API Укрзалізниці.

CQRS (Command Query Responsibility Segregation) – патерн, який розділяє операції читання та запису даних для підвищення продуктивності та масштабованості системи.

Domain (Домен) – частина системи, що відповідає за бізнес-логіку та правила предметної області (користувачі, замовлення, поїздки).

Frontend (Фронтенд) – користувацький інтерфейс системи. У AutoTix реалізований на React для зручного пошуку поїздок, формування замовлень та оплати.

Microservices (Мікросервіси) – підхід до побудови програмного забезпечення у вигляді незалежних сервісів, що виконують окремі функції та взаємодіють між собою через API або шину подій.[1]

RabbitMQ – брокер повідомлень для асинхронного обміну даними між мікросервісами, використовується для реалізації патерну event-driven та Saga.

Redis – високопродуктивне сховище ключ-значення, яке застосовується для кешування даних і швидкого доступу до часто використовуваної інформації.

Saga – патерн управління довготривалими бізнес-процесами, що забезпечує узгодженість станів між мікросервісами через серії транзакцій та компенсаційних дій.

Token (Токен) – механізм аутентифікації та авторизації користувача. У AutoTix застосовується для доступу до внутрішніх сервісів та API Укрзалізниці.

Trip (Поїздка) – структурований набір даних про маршрут, станції, вагони та доступні місця для бронювання.

Wagon (Вагон) – елемент поїзда, що містить класи місць, тарифну інформацію та доступність для бронювання.

DTO (Data Transfer Object) – об’єкт передачі даних між сервісами або між фронтендом і бекендом для забезпечення стабільності контрактів.

Event-driven (Подієво орієнтована архітектура) – підхід, при якому сервіси реагують на події (events) для асинхронної обробки логіки та взаємодії між компонентами.

Order (Замовлення) – агрегат бізнес-логіки, що містить інформацію про користувача, вибрані поїздки та стан бронювання.

TicketReservation (Резервація квитка) – сутність замовлення, що відповідає за конкретне місце, вагон та кількість квитків.

TripSnapshot – “знімок” поїздки на момент бронювання, який фіксує стан маршруту та тарифи, щоб уникнути розбіжностей у майбутньому.

Microservice Architecture (Мікросервісна архітектура) – структура програмного забезпечення, що дозволяє розділяти систему на дрібні, незалежні сервіси, що взаємодіють через API та шину подій.

Load (Навантаження) – кількість одночасних запитів до системи; аналіз продуктивності AutoTix включає перевірку роботи під навантаженням.

Scalability (Масштабованість) – здатність системи ефективно збільшувати продуктивність при додаванні нових ресурсів.

Authentication (Аутентифікація) – процес перевірки особи користувача для надання доступу до системи.

Authorization (Авторизація) – процес визначення рівня доступу користувача до конкретних ресурсів або операцій системи.

РОЗДІЛ 1 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ТА РІШЕНЬ ДЛЯ ПОБУДОВИ СИСТЕМ ЗАМОВЛЕНЬ

1.1 Огляд поняття системи замовлень

Система замовлень — це програмно-технічний комплекс, призначений для автоматизації процесів приймання, обробки, зберігання та виконання замовлень користувачів. Такі системи широко застосовуються у сферах торгівлі, логістики, транспортних перевезень, готельного бізнесу, сервісів онлайн-бронювання та інших галузях, де необхідно швидко та точно обробляти великий потік запитів.

У загальному вигляді система замовлень функціонує як посередник між користувачем та постачальником послуги або товару, забезпечуючи коректність процедур оформлення, обліку та виконання замовлення. Вона дозволяє автоматизувати операції, які раніше виконувалися вручну, тим самим підвищуючи ефективність, зменшуючи кількість помилок та скорочуючи час обслуговування.

Основними компонентами типової системи замовлень є:

Інтерфейс користувача (UI/UX) — набір засобів взаємодії з системою, що дозволяє формувати запити, обирати параметри замовлення, переглядати доступні варіанти та підтверджувати покупку або бронювання.

Модуль управління замовленнями (Order Management System, OMS) — центральний компонент, що відповідає за обробку даних замовлення, перевірку його коректності, розподіл по службам чи мікросервісам та відстеження статусу.

База даних або набір сховищ — інструменти для зберігання інформації про користувачів, їх замовлення, історію операцій, доступність товарів чи ресурсів.

Модуль інтеграції з зовнішніми сервісами — API або інші канали обміну інформацією, через які система отримує актуальні дані про доступність ресурсів (наприклад, місця у потягах), здійснює оплату, підтверджує бронювання або синхронізує інформацію між різними платформами.

Модулі аналітики — складова, що дозволяє аналізувати дані про попит, активність користувачів, статистику успішних і неуспішних операцій, прогнозувати майбутні навантаження.

У контексті залізничних перевезень система замовлень має ряд особливостей. Вона повинна працювати в умовах високої динаміки змін (квитки можуть з’являтися й зникати протягом секунд), обробляти значний потік запитів у короткий проміжок часу та забезпечувати високу точність бронювання. Крім того, система має підтримувати підключення до офіційних серверів залізничних операторів, дотримуючись вимог до безпеки, захисту персональних даних, стабільності мережевої взаємодії та коректності проведення фінансових транзакцій.

Системи замовлень можуть бути реалізовані на основі різних архітектурних підходів. Історично найбільш поширеними були монолітні системи, в яких усі модулі функціонують як єдине ціле. Проте зі зростанням вимог до масштабованості, швидкодії та гнучкості дедалі більшої популярності набувають мікросервісні системи, де функціональність розділена на незалежні сервіси, що взаємодіють через стандартизовані інтерфейси.

Важливою характеристикою сучасної системи замовлень є її здатність працювати в режимі реального часу, обробляти пікові навантаження та швидко реагувати на появу або зникнення доступних ресурсів. Це особливо актуально у сфері продажу квитків на популярні маршрути, де доступність місць може змінюватися щосекунди.

Отже, система замовлень — це не просто інструмент оформлення покупки, а комплексна багаторівнева інформаційна платформа, що забезпечує

автоматизоване управління всім життєвим циклом замовлення та підтримує високу якість і швидкість обслуговування користувачів. У наступних підпунктах буде розглянуто недоліки традиційних підходів і обґрунтовано перехід до сучасних мікросервісних рішень.

1.2 Проблеми та обмеження традиційних монолітних систем замовлень

Традиційна монолітна архітектура історично є одним із найпоширеніших підходів до розробки інформаційних систем, зокрема систем приймання та обробки замовлень. У межах монолітної моделі всі компоненти програмного забезпечення — користувацький інтерфейс, бізнес-логіка, модулі обробки транзакцій, інтеграційні засоби та підсистеми доступу до даних — функціонують у складі єдиного цілісного застосунку. Такий підхід забезпечував відносну простоту початкового проєктування та розгортання, однак з розвитком цифрових сервісів та зростанням навантаження виявив низку суттєвих обмежень.

По-перше, монолітні системи характеризуються низькою масштабованістю. Для збільшення їх продуктивності необхідно масштабувати програму повністю, навіть якщо підвищене навантаження спостерігається лише на окремих функціональних модулях. Це призводить до нераціонального використання обчислювальних ресурсів, ускладнює оптимізацію витрат та негативно впливає на здатність системи ефективно працювати під час пікових навантажень, характерних для сервісів продажу квитків.

По-друге, монолітна архітектура істотно ускладнює процес внесення змін, оновлення та розширення функціоналу. Будь-яке модифікування окремого компонента потребує повторної збірки й розгортання всього застосунку, що збільшує ризик виникнення збоїв і унеможливлює гнучке впровадження нових функцій. У випадку систем бронювання це створює ризики переривання роботи

сервісу під час критичних часових інтервалів, наприклад у момент старту продажів квитків.

По-третє, монолітні системи мають обмежену відмовостійкість. Помилка в одному модулі може спричинити повне припинення роботи всієї системи, оскільки компоненти тісно пов’язані між собою та не мають ізольованого середовища виконання. Така взаємозалежність зменшує надійність сервісу й ускладнює забезпечення безперервності його функціонування.

По-четверте, зі збільшенням обсягів даних і складності бізнес-логіки ускладнюється підтримка та розвиток монолітних застосунків. Розростання кодової бази призводить до появи значної кількості внутрішніх залежностей, що ускладнює роботу команди розробників, збільшує час тестування та зменшує гнучкість системи. Впровадження нових механізмів аналітики, модулів прогнозування попиту чи засобів автоматичного моніторингу стає технічно складним та економічно невиправданим.

По-п’яте, монолітний підхід обмежує вибір технологій. Усі компоненти змушені використовувати єдину технологічну платформу, навіть якщо окремі функціональні частини потребують різних інструментів для досягнення оптимальної продуктивності.

По-шосте, монолітні системи демонструють обмежену ефективність в умовах високої конкуренції доступу до ресурсу, що характерно для сфери транспортного бронювання. Одночасне надходження великої кількості запитів може спричиняти тривалі затримки, помилки обробки або повну недоступність сервісу.

Таким чином, традиційні монолітні системи замовлень не відповідають сучасним вимогам до масштабованості, гнучкості, оперативного внесення змін та підтримки високої продуктивності. Перераховані обмеження підтверджують необхідність переходу до альтернативних архітектурних підходів, зокрема мікросервісної архітектури, яка дозволяє забезпечити підвищену стійкість,

адаптивність і здатність обробляти значні обсяги запитів у режимі реального часу.

1.3 Огляд мікросервісної архітектури

Мікросервісна архітектура на сьогодні є одним із провідних підходів до проєктування складних розподілених інформаційних систем. Вона базується на принципі структурної декомпозиції програмного забезпечення на низку незалежних модулів (мікросервісів), кожен з яких реалізує окрему бізнес-функцію та взаємодіє з іншими модулями через стандартизовані інтерфейси. Такий підхід забезпечує гнучкість, масштабованість та керованість системи, що є критично важливим у галузях із високими навантаженнями та підвищеними вимогами до відмовостійкості.

У науковій літературі мікросервіс визначається як автономний програмний компонент, який виконується у власному середовищі, має ізольовану логіку, власну модель даних та визначений протокол взаємодії з іншими компонентами. Організація системи у вигляді множини таких сервісів дозволяє мінімізувати залежності між ними та забезпечити можливість їх незалежної розробки, тестування та розгортання.

До основних принципів мікросервісної архітектури належать:

1. Функціональна декомпозиція системи.

Складна система розділяється на сукупність дрібних сервісів, кожен з яких відповідає за окремий бізнес-процес або його частину. Це сприяє логічній структуризації та полегшує підтримку.

2. Автономність та ізольованість компонентів.

Мікросервіси є незалежними у контексті життєвого циклу: вони можуть оновлюватися, масштабуватися та відновлюватися окремо, не впливаючи на стабільність роботи інших модулів системи.

3. Легковагові механізми взаємодії.

Комунікація між сервісами реалізується за допомогою простих, стандартизованих протоколів (REST, gRPC, AMQP тощо), що забезпечує прозорість обміну та спрощує інтеграцію.

4. Децентралізація керування даними.

На відміну від монолітних систем, де всі дані зберігаються в єдиному сховищі, у мікросервісних системах кожен сервіс може мати власну базу даних. Такий підхід підвищує ефективність виконання транзакцій та зменшує кількість конфліктів доступу.

5. Підтримка DevOps-процесів і безперервної інтеграції.

Мікросервісний підхід органічно поєднується з практиками CI/CD, що забезпечує можливість оперативного внесення змін, скорочення часу релізу та підвищення загальної якості програмного продукту.

У контексті побудови систем замовлень, зокрема систем автоматизованого бронювання залізничних квитків, мікросервісна архітектура має низку переваг. Передусім йдеться про можливість незалежного масштабування найбільш навантажених компонентів, таких як модулі перевірки доступності квитків або модулі обробки платіжних операцій. Це забезпечує стабільність функціонування системи у періоди пікових навантажень, що є характерною рисою транспортної галузі.

Крім того, ізольованість сервісів сприяє підвищенню відмовостійкості: збій окремого компонента не призводить до зупинки всієї системи, а локалізується в межах одного сервісу. Такий підхід є важливим з огляду на необхідність забезпечення безперебійності доступу користувачів до сервісів бронювання.

Також значну цінність становить можливість гнучкого інтегрування зовнішніх сервісів — платіжних провайдерів, систем авторизації, API транспортних операторів тощо. Незалежність мікросервісів дає змогу адаптувати систему до нових вимог без потреби її повного рефакторингу.

Таким чином, мікросервісна архітектура виступає ефективною основою для створення сучасних високопродуктивних систем замовлень, забезпечуючи високий рівень масштабованості, надійності, технологічної гнучкості та придатності до подальшого розвитку. Саме ці характеристики зумовлюють її доцільність для використання у системах автоматизованого бронювання залізничних квитків.

1.4 Стандарти побудови мікросервісних систем

Побудова мікросервісних інформаційних систем передбачає систематичне дотримання встановлених стандартів, які визначають методологічні, технічні та організаційно-методичні підходи до проєктування, розгортання, інтеграції та супроводу окремих компонентів, що забезпечує передбачуваність їх взаємодії, масштабованість, надійність та стійкість до відмов у межах розподіленої архітектури. У науково-технічній літературі під стандартами мікросервісної архітектури розуміють сукупність правил і рекомендацій, що регламентують уніфіковану взаємодію компонентів, організацію життєвого циклу сервісів, моделі зберігання даних, процедури обміну повідомленнями, а також практики забезпечення безпеки та моніторингу системи.

До ключових груп стандартів, що застосовуються при проєктуванні та експлуатації мікросервісних систем, належать:

1. Стандарти взаємодії та комунікації між сервісами.

Належна організація комунікацій між компонентами є критичною для забезпечення коректності функціонування всієї системи. Найбільш поширеними є протоколи REST (Representational State Transfer), які забезпечують уніфікований доступ до ресурсів через стандартизовані URI та використання HTTP/HTTPS, а також gRPC, що базується на високопродуктивному протоколі HTTP/2 із серіалізацією даних за допомогою Protocol Buffers, що гарантує

мінімізацію затримок і ефективність передачі структурованої інформації. Додатково застосовуються асинхронні протоколи обміну повідомленнями, такі як AMQP та MQTT, які дозволяють забезпечити надійну обробку подій у розподіленому середовищі.

2. Стандарти ізоляції та розгортання сервісів.

Кожен мікросервіс повинен функціонувати у ізольованому середовищі, що дозволяє мінімізувати взаємний вплив компонентів та забезпечує незалежність циклів їх розгортання і оновлення. Найчастіше для цього застосовуються технології контейнеризації (Docker[14]) та оркестрації (Kubernetes[15], Docker[14] Swarm), що забезпечують автоматизоване управління життєвим циклом контейнерів, балансування навантаження, автоматичне відновлення після збоїв та можливість масштабування в залежності від інтенсивності запитів.

3. Стандарти спостережуваності та контролю стану системи.

У зв’язку з високою розподіленістю мікросервісних систем необхідно застосовувати централізовані підходи до моніторингу та логування. Це включає використання платформ для централізованого збирання логів (ELK Stack, Graylog), систем моніторингу метрик у реальному часі (Prometheus, Grafana) та засобів трасування розподілених транзакцій (Jaeger, Zipkin), що дозволяє оцінювати продуктивність кожного сервісу, виявляти вузькі місця та своєчасно реагувати на відхилення від нормальної роботи.

4. Стандарти безпеки та управління доступом.

Безпека мікросервісних систем визначається застосуванням усталених протоколів автентифікації та авторизації (OAuth 2.0, OpenID Connect), шифруванням каналів передачі даних із використанням TLS/HTTPS та токенами безпечного обміну інформацією (JWT). Дотримання цих стандартів є обов’язковою умовою для захисту конфіденційних даних користувачів та забезпечення цілісності транзакцій у високонавантажених системах замовлень.

5. Стандарти організації безперервної інтеграції та доставки (CI/CD).

Мікросервісна архітектура передбачає активне застосування практик DevOps, що включають автоматизовані pipeline-процеси (GitLab CI/CD, Jenkins, GitHub Actions), модульне і інтеграційне тестування, а також безперервне розгортання із застосуванням стратегій blue–green deployment та canary release. Використання цих стандартів дозволяє мінімізувати ризики відмов, скоротити час виходу нових версій та забезпечити стабільність роботи системи навіть під час оновлень.

Отже, дотримання зазначених стандартів є необхідною умовою побудови ефективних мікросервісних систем замовлень, оскільки воно забезпечує інтегрованість компонентів, високий рівень продуктивності, відмовостійкість та можливість масштабування відповідно до зміни навантаження, що є критично важливим для систем автоматизованого бронювання залізничних квитків. Водночас стандартизація процесів розробки та експлуатації підвищує керованість системи та дозволяє забезпечити передбачувану поведінку програмного комплексу в умовах високої динаміки змін запитів користувачів.

1.5 Аналіз існуючих систем продажу квитків

Створення системи автоматизованого бронювання залізничних квитків із використанням мікросервісної архітектури є актуальною темою в галузі транспортної інформатики. Для кращого розуміння можливостей сучасних рішень у цій сфері доцільно провести огляд і аналіз існуючих програмних продуктів, які мають подібний функціонал. Це дозволить виявити сильні та слабкі сторони конкурентів і врахувати їх у процесі розробки власної системи.

RailwayBot — це Telegram-бот, який автоматично моніторить наявність квитків на залізничні рейси, переважно для українських маршрутів, і надсилає користувачеві повідомлення про появу вільних місць. Цей сервіс став

популярним серед пасажирів, які шукають квитки на популярні або сезонні напрямки, де попит перевищує пропозицію.

Інтерфейс бота показано на рисунку 1.

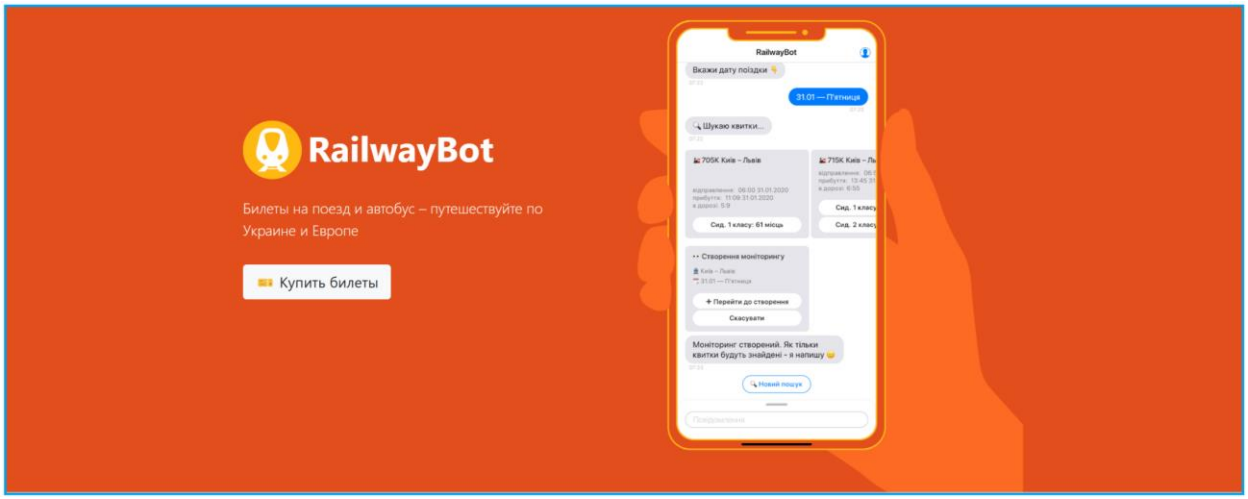


Рисунок 1 – *Інтерфейс RailwayBot*

Переваги:

1. RailwayBot зручний у використанні — взаємодія з ботом відбувається через месенджер, що не потребує окремого додатка.
2. Миттєві повідомлення в реальному часі дозволяють швидко реагувати на появу квитків.
3. Сервіс дозволяє заощадити час, усуваючи необхідність постійного самостійного перевіряння наявності квитків.

Недоліки:

1. Обмежений функціонал — відсутність повноцінної системи бронювання або інтеграції з особистим кабінетом користувача.
2. Залежність від стабільної роботи Telegram, що може спричинити затримки в отриманні сповіщень.
3. Відсутність повноцінного аналізу попиту чи рекомендацій щодо найкращого часу покупки.

Офіційний мобільний застосунок «Укрзалізниці» дозволяє здійснювати повноцінне бронювання квитків на потяги, переглядати розклад, отримувати оновлення про затримки та зміни у маршрутах, а також зберігати електронні квитки з QR-кодом.

Інтерфейс додатка показано на рисунку 2.

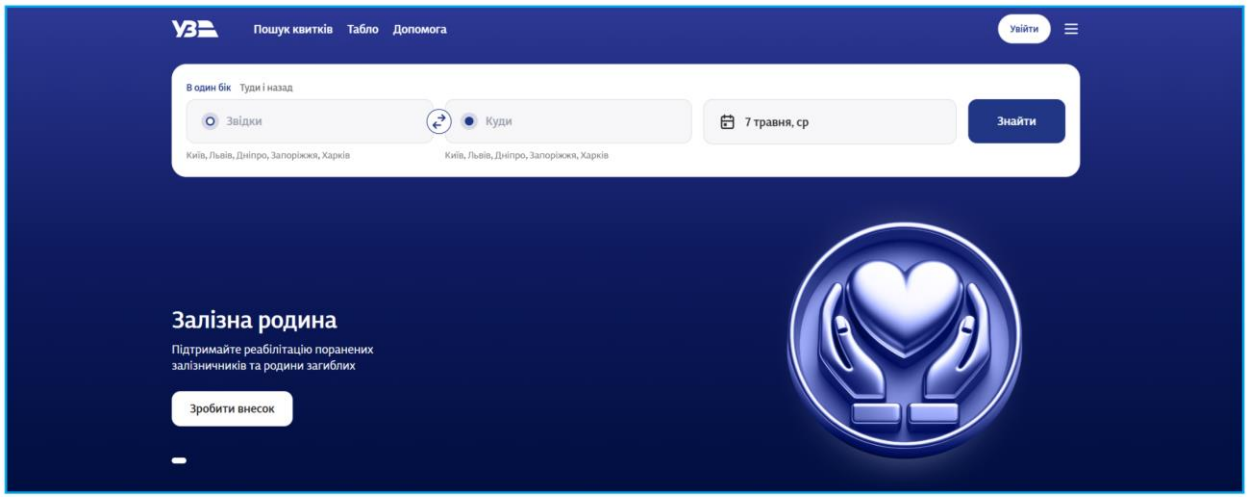


Рисунок 1 – *Офіційний мобільний застосунок «Укразалізниця»*

Переваги:

1. Повний спектр офіційних функцій — від вибору рейсу до покупки й перевірки квитка.
2. Інтеграція з державними системами, що забезпечує точність і своєчасність інформації.
3. Підтримка електронного квитка — не потрібно друкувати паперові документи.

Недоліки:

1. Відсутність можливості автоматичного моніторингу вільних місць чи покупки квитків без участі користувача.
2. Додаток може бути перевантажений у пікові періоди, що впливає на швидкодію.
3. Обмеженість персоналізованих функцій, таких як рекомендації за аналітикою попиту чи найкращим часом покупки.

Сервер модуля узгодження — це корпоративне рішення, розроблене компанією ТМСофт, яке дозволяє підприємствам автоматизувати обмін електронними перевізними документами з АСУ «Укрзалізниці». Хоча продукт орієнтований на бізнес-користувачів, його архітектурні принципи та логіка обробки запитів можуть бути цікавими з точки зору побудови масштабованих систем бронювання.

Інтерфейс адміністративної панелі системи показано на рисунку 3.

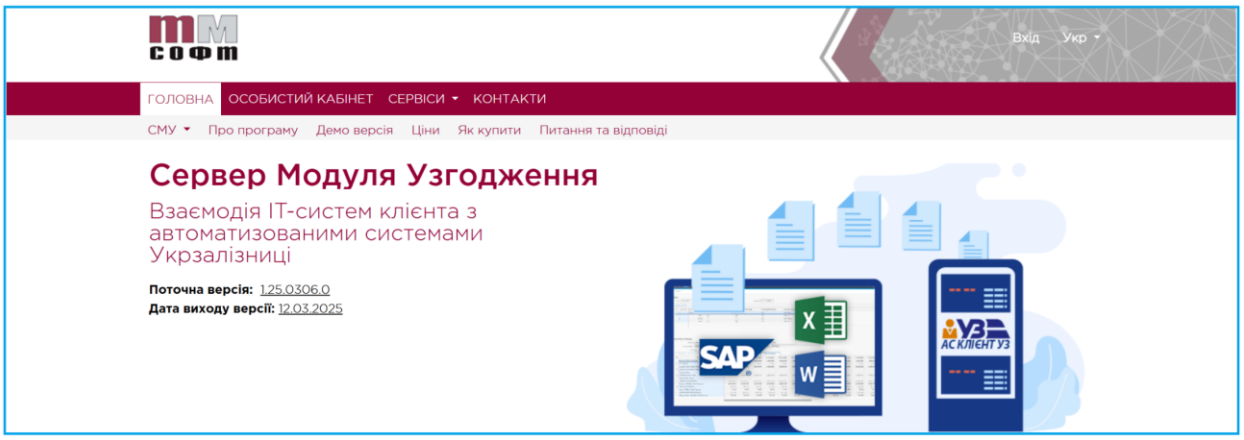


Рисунок 3 – *Сервер модуля узгодження*

Переваги:

1. Підтримка електронного документообігу з використанням ЕЦП.
2. Висока надійність у обробці перевізної документації завдяки логічному та форматному контролю.
3. Можливість гнучкої інтеграції з внутрішніми ERP-системами підприємств.

Недоліки:

1. Не призначений для широкого кола пасажирів — орієнтований виключно на корпоративний сегмент.
2. Потребує спеціальних знань для налаштування та підтримки.
3. Відсутність візуального інтерфейсу для бронювання або управління квитками.

Продивившись та проаналізувавши програмні продукти-аналоги, можемо побачити, які переваги має система автоматизованого бронювання залізничних квитків, реалізована з використанням мікросервісної архітектури та сучасних інструментів аналізу попиту.

1.6 Узагальнення недоліків існуючих рішень та обґрунтування вибору мікросервісного підходу для AutoTix

Аналіз існуючих програмних рішень у сфері автоматизованого бронювання залізничних квитків дозволяє систематизувати їхні основні недоліки та обмеження, що суттєво впливають на ефективність їх використання в умовах високої інтенсивності користувацьких запитів і великого навантаження на систему. До найбільш значущих недоліків належать:

1. Обмежена масштабованість та гнучкість системи. Традиційні монолітні архітектури не передбачають можливості автономного масштабування окремих функціональних компонентів. Для забезпечення високої продуктивності під час пікових навантажень необхідне дублювання всієї системи, що призводить до неефективного використання обчислювальних ресурсів та збільшення витрат на інфраструктуру.
2. Висока залежність від стабільності роботи центральної системи. Помилка в одному з модулів або перевантаження монолітної системи призводить до повної зупинки роботи сервісу, що негативно впливає на безперервність обслуговування користувачів та знижує надійність обробки замовлень у критичні періоди продажу квитків.
3. Ускладнене внесення змін та інтеграцію нових функцій. Будь-яке оновлення або додавання нових функціональних можливостей вимагає повної перебудови та повторного розгортання системи. Така особливість ускладнює

адаптацію до нових бізнес-вимог, збільшує час тестування та підвищує ризик виникнення збоїв у роботі сервісу.

4. Низька відмовостійкість та обмежена керованість компонентами. У разі збою одного модуля система втрачає цілісність транзакцій та здатність обробляти запити користувачів. Це створює критичні ризики для стабільного функціонування платформи в умовах високої активності.
5. Обмежені можливості інтеграції з зовнішніми сервісами та аналітичними платформами. Існуючі рішення не забезпечують гнучкої взаємодії з платіжними системами, сторонніми API або аналітичними сервісами, що зменшує адаптивність системи до змін у вимогах користувачів та потреб ринку.

Враховуючи зазначені обмеження, для розробки системи AutoTix обрано мікросервісний підхід, що обґрунтовано наступними перевагами:

- Незалежне масштабування модулів. Ключові компоненти системи, зокрема модулі пошуку квитків, бронювання та обробки платежів, можуть масштабуватися автономно, що забезпечує оптимальне використання ресурсів та стабільну роботу під час пікових навантажень.
- Підвищена відмовостійкість та ізольованість сервісів. Локалізація збоїв у межах окремого сервісу дозволяє уникнути критичного впливу на роботу всієї системи та гарантує безперервність обробки запитів користувачів.
- Гнучкість у впровадженні нових функцій. Незалежність мікросервісів дозволяє швидко оновлювати окремі модулі, застосовувати різні технологічні стеки та оперативно інтегрувати додаткові сервіси, що підвищує адаптивність платформи до потреб користувачів.
- Централізоване спостереження та моніторинг. Використання сучасних інструментів контролю стану системи (логування, моніторинг, трасування) забезпечує своєчасне виявлення проблем та підтримання стабільності функціонування платформи.

- Полегшена інтеграція із зовнішніми сервісами. Мікросервісна структура сприяє швидкій та безпечній взаємодії з платіжними шлюзами, API транспортних операторів та аналітичними платформами без необхідності модифікації всієї системи.

Отже, застосування мікросервісної архітектури для реалізації AutoTix забезпечує високий рівень продуктивності, масштабованості, відмовостійкості та технологічної гнучкості, що робить її оптимальним рішенням для побудови системи автоматизованого бронювання залізничних квитків у сучасних умовах високого навантаження та динамічної зміни користувацьких потреб.

1.7 Висновки до розділу 1

Проведений аналіз існуючих підходів і рішень у сфері автоматизованого бронювання залізничних квитків показав, що традиційні монолітні системи характеризуються обмеженою масштабованістю, низькою відмовостійкістю, складністю внесення змін і обмеженою інтеграційною здатністю. Ці обмеження суттєво знижують ефективність роботи систем у періоди пікового навантаження та ускладнюють адаптацію до змін користувацьких вимог і бізнес-процесів.

На основі проведеного аналізу обґрунтовано вибір мікросервісної архітектури для реалізації системи AutoTix. Такий підхід забезпечує незалежне масштабування компонентів, ізольовану роботу сервісів, підвищену відмовостійкість, гнучкість у впровадженні нових функцій і інтеграції з зовнішніми сервісами. Впровадження мікросервісної архітектури створює передумови для стабільного, керованого та адаптивного функціонування системи автоматизованого бронювання залізничних квитків, що є критично важливим для забезпечення безперервної обробки запитів користувачів у режимі реального часу.

РОЗДІЛ 2 ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ ДЛЯ ПОБУДОВИ МІКРОСЕРВІСНОЇ СИСТЕМИ ЗАМОВЛЕНЬ

2.1 Аналіз технологій бекенд-розробки

Розробка мікросервісної системи автоматизованого бронювання залізничних квитків передбачає використання сучасних технологій бекенд-розробки, які забезпечують масштабованість, високу продуктивність, надійність та безпеку обробки даних. Вибір відповідних технологій обумовлюється як функціональними вимогами системи, так і вимогами до підтримки високого навантаження, швидкої обробки запитів та інтеграції з зовнішніми сервісами.

2.1.1 .NET 8 / ASP.NET Core

Використання платформи .NET 9 та фреймворку ASP.NET Core обумовлено їхньою здатністю забезпечувати високопродуктивні веб-сервіси та підтримку мікросервісної архітектури. ASP.NET Core дозволяє створювати незалежні модулі з чітко визначеними API, забезпечує кросплатформність і легку інтеграцію з контейнеризованими середовищами. Використання цієї технології сприяє підвищенню масштабованості системи, ефективному управлінню ресурсами та забезпеченню стабільної роботи під час високих навантажень, що критично для систем бронювання у пікові періоди продажу квитків.

2.1.2 MassTransit / RabbitMQ

Для забезпечення надійної асинхронної взаємодії між мікросервісами обрано MassTransit як бібліотеку для організації обміну повідомленнями та RabbitMQ[13] як брокера повідомлень. Цей стек дозволяє реалізувати шаблони повідомлень “publish/subscribe” і черги для обробки запитів у реальному часі. Таке рішення підвищує відмовостійкість системи, оскільки окремі сервіси

можуть обробляти повідомлення незалежно один від одного, а у разі збою окремого компоненту повідомлення не втрачаються, а очікують обробки.

2.1.3 EF Core + MSSQL

Для організації зберігання та обробки структурованих даних обрано Entity Framework Core (EF Core) у поєднанні з реляційною базою даних Microsoft SQL Server (MSSQL). EF Core забезпечує зручне відображення об’єктно-реляційних зв’язків, спрощує роботу з базою даних, дозволяє реалізовувати складні запити та підтримує механізми відстеження змін даних. Використання MSSQL гарантує надійність збереження даних, масштабованість і можливість обробки великих обсягів транзакцій у системах високої інтенсивності.

2.1.4 Identity + JWT

Для організації автентифікації та авторизації користувачів використовується ASP.NET Identity у поєднанні з JSON Web Tokens (JWT). Ця комбінація забезпечує безпечний механізм аутентифікації та авторизації, можливість централізованого управління правами доступу, підтримку одноразових сесій та інтеграцію з зовнішніми сервісами. Використання JWT дозволяє реалізувати безпечну передачу інформації між мікросервісами та фронтендом, зберігаючи цілісність даних і запобігаючи несанкціонованому доступу.

2.2 Аналіз підходів до інтеграції з зовнішніми API

Інтеграція мікросервісної системи автоматизованого бронювання залізничних квитків із зовнішніми сервісами є критично важливою для забезпечення повноцінного функціоналу та своєчасного обміну даними. До таких сервісів належать системи електронного бронювання перевізників, платіжні

шлюзи та аналітичні платформи. Вибір оптимального підходу до інтеграції визначає продуктивність системи, надійність обробки запитів, безпеку передачі даних та гнучкість у подальшому масштабуванні.

Основні підходи до інтеграції з зовнішніми API:

1. REST API

REST (Representational State Transfer) є найбільш поширеним стандартом обміну даними між сервісами через протокол HTTP/HTTPS. Його використання дозволяє забезпечити уніфікований доступ до ресурсів, гнучку маршрутизацію запитів та широкі можливості кешування і масштабування. REST API характеризується простотою реалізації, підтримкою стандартних методів (GET, POST, PUT, DELETE) та сумісністю з більшістю сучасних технологій.

2. gRPC

gRPC є високопродуктивним протоколом віддаленого виклику процедур, що базується на HTTP/2 та використовує серіалізацію даних через Protocol Buffers. Цей підхід забезпечує мінімальні затримки при передачі структурованих даних, високу пропускну здатність та можливість реалізації двосторонньої потокової комунікації. gRPC особливо ефективний у середовищах мікросервісної архітектури, де необхідна швидка та надійна взаємодія між компонентами.

3. Механізми аутентифікації та авторизації при інтеграції

Для забезпечення безпеки обміну даними застосовуються стандарти OAuth 2.0 та OpenID Connect, а також використання токенів доступу (JWT). Вони гарантують, що лише авторизовані сервіси можуть отримувати доступ до ресурсів, та дозволяють централізовано керувати правами доступу для зовнішніх інтеграцій.

Вибір конкретного підходу інтеграції для системи AutoTix ґрунтується на таких критеріях, як висока продуктивність обміну даними, підтримка асинхронного та синхронного виклику сервісів, безпека передачі інформації та простота підтримки. У випадку AutoTix перевага надається поєднанню REST API

для взаємодії із зовнішніми сервісами та gRPC для швидкої внутрішньої комунікації між мікросервісами.

Таким чином, обрані підходи до інтеграції забезпечують надійність, продуктивність та безпеку роботи мікросервісної системи AutoTix.

2.3 Фронтенд технології

Фронтенд-система мікросервісного застосунку AutoTix відповідає за взаємодію з користувачем, відображення даних та забезпечення зручності використання системи. Вибір сучасних фронтенд-технологій дозволяє реалізувати високопродуктивний, адаптивний та масштабований інтерфейс користувача, який здатний ефективно працювати з даними, що надходять від мікросервісів бекенду, та підтримувати сучасні вимоги UX/UI.

2.3.1 React

Для побудови інтерфейсу користувача обрана бібліотека React. Вона забезпечує компонентну архітектуру, що дозволяє розділяти інтерфейс на незалежні, багаторазово використовувані модулі. React підтримує віртуальний DOM, що оптимізує оновлення сторінок та підвищує продуктивність роботи застосунку, особливо при динамічних змінах даних, таких як доступність квитків у режимі реального часу. Використання React дозволяє легко інтегрувати сторонні бібліотеки та забезпечує сумісність із сучасними інструментами управління станом і маршрутизації.

2.4 Способи забезпечення надійності та узгодженості

Надійність та узгодженість даних у мікросервісних системах є критично важливими аспектами, особливо для систем бронювання квитків, де необхідна

коректна обробка транзакцій та мінімізація ризику помилок. У мікросервісній архітектурі кожен сервіс працює автономно, що ускладнює забезпечення консистентності даних при високих навантаженнях та одночасному виконанні численних запитів. Для вирішення цих проблем застосовуються спеціалізовані патерни та механізми управління транзакціями.

2.4.1 Saga Pattern

Saga Pattern є патерном управління транзакціями в розподілених системах, що дозволяє забезпечити узгодженість даних без використання глобальних блокувань бази даних. Транзакція розбивається на серію локальних операцій у межах окремих мікросервісів, де кожна операція супроводжується компенсуючою дією на випадок помилки. У системі AutoTix застосування Saga Pattern дозволяє ефективно координувати процес бронювання квитків, оплати та підтвердження замовлення, гарантуючи, що у разі збою будь-який виконаний крок можна коректно відкотити без порушення цілісності даних.

2.5. Системи контейнеризації та деплою

Для забезпечення стабільності, масштабованості та керованості мікросервісної системи AutoTix застосовуються сучасні технології контейнеризації та автоматизованого деплою. Використання контейнерів дозволяє ізолювати сервіси, створити однакове середовище для розробки, тестування та продакшн, а також спрощує процеси масштабування та підтримки окремих компонентів системи відповідно до навантаження.

Контейнеризація за допомогою Docker. Docker[14] забезпечує упаковку кожного мікросервісу з усіма його залежностями в окремий контейнер. Такий підхід гарантує стабільну роботу сервісів у будь-якому середовищі, незалежно від конфігурацій хост-системи. Крім того, контейнеризація спрощує процеси

оновлення та відкату версій сервісів, підвищує швидкість розгортання та мінімізує конфлікти між залежностями.

Оркестрація та управління контейнерами (Kubernetes). Для ефективного управління великою кількістю контейнерів використовується Kubernetes[15] — платформа оркестрації, яка забезпечує автоматичне масштабування, балансування навантаження, самовідновлення сервісів у разі збою та централізоване управління конфігураціями. Kubernetes[15] дозволяє керувати життєвим циклом мікросервісів, підвищує відмовостійкість системи та мінімізує час простою.

CI/CD (Continuous Integration / Continuous Deployment). Впровадження практик CI/CD забезпечує автоматизацію процесів інтеграції та розгортання нових версій мікросервісів. Це дозволяє оперативно перевіряти якість коду, проводити автоматичне тестування та швидко доставляти оновлення у продакшн без порушення роботи системи. Такий підхід особливо важливий для систем бронювання квитків, де стабільність та безперервність обробки запитів є критичними.

Застосування контейнеризації та систем автоматизованого деплою у AutoTix забезпечує масштабованість, керованість та надійність системи, полегшує управління її компонентами та дозволяє швидко впроваджувати нові функції, що відповідає сучасним вимогам до високонавантажених мікросервісних платформ.

2.6. Висновки до розділу 2

Проведений аналіз технологій та інструментів для розробки мікросервісної системи AutoTix дозволив обґрунтувати вибір оптимального стеку технологій для бекенду, фронтенду та інтеграції з зовнішніми сервісами. Використання .NET 8 / ASP.NET Core, RabbitMQ, EF Core + MSSQL та Identity + JWT забезпечує

високу продуктивність, надійність та масштабованість мікросервісів. Для фронтенду обрані React, що дозволяє реалізувати сучасний, швидкий та зручний інтерфейс користувача.

Впровадження патернів управління транзакціями (Saga Pattern), а також застосування контейнеризації (Docker[14]), оркестрації (Kubernetes[15]) та практик CI/CD, забезпечує стабільність, відмовостійкість та узгодженість даних у системі. Комплексне використання цих технологій та підходів гарантує ефективне функціонування системи AutoTix у режимі високого навантаження та створює передумови для її подальшого масштабування й інтеграції з іншими сервісами.

РОЗДІЛ 3. ПРОЄКТУВАННЯ ТА РОЗРОБКА МІКРОСЕРВІСНОЇ СИСТЕМИ ЗАМОВЛЕНЬ AutoTix

3.1 Архітектура системи AutoTix

Архітектура програмного комплексу AutoTix побудована на основі мікросервісного підходу, який дає змогу розподілити функціональність системи на окремі, ізольовані один від одного компоненти. Кожен мікросервіс відповідає за власний сегмент предметної області, має власну базу даних і може оновлюватися, масштабуватися або розгортатися незалежно від інших складових. Такий підхід забезпечує гнучкість системи, дозволяє легко модифікувати її в процесі експлуатації та підвищує загальну стійкість до відмов.

Усі сервіси взаємодіють між собою через два основні механізми — синхронні gRPC-виклики та асинхронні повідомлення RabbitMQ[13]. Синхронний обмін використовується у тих випадках, коли необхідно отримати відповідь негайно, наприклад, при авторизації користувача або під час виконання запиту до API Укрзалізниці. Натомість для бізнес-процесів, що складаються з декількох взаємопов’язаних етапів (бронювання місць, опрацювання платежу, генерація квитка тощо), застосовується асинхронний обмін подіями, який дозволяє ефективно керувати довготривалими операціями та мінімізує залежність між сервісами.

Загальна архітектура AutoTix включає такі основні компоненти:

- набір мікросервісів, кожен з яких реалізує окремий bounded context;
- RabbitMQ, що слугує шиною подій для асинхронного зв’язку між сервісами;
- gRPC взаємодію, яка застосовується між UserService та RailwayConnectorService для швидкого та надійного обміну даними;

- API Gateway, який виконує роль єдиної точки доступу для клієнтських застосунків і відповідає за маршрутизацію запитів, перевірку авторизації та обмеження доступу;
- розподілені бази даних, що дозволяють сервісам зберігати інформацію незалежно одне від одного та уникати транзакційних конфліктів;
- клієнтський інтерфейс на React, який працює виключно через Gateway та отримує агреговану інформацію від різних сервісів;

Така побудова архітектури дає змогу забезпечити масштабованість системи залежно від навантаження, підвищити стабільність роботи окремих компонентів та забезпечити можливість розвитку платформи без істотного втручання у вже реалізовані функції.

3.2 Загальна схема мікросервісів

Загальна структура архітектури платформи AutoTix представлена набором спеціалізованих мікросервісів(див. Рис 4), кожен з яких реалізує окремий bounded context предметної області. Мікросервіси сполучені двома основними каналами комунікації: синхронними gRPC-викликами для операцій, що потребують негайної відповіді, та асинхронним обміном повідомленнями через RabbitMQ[13] для координації тривалих або багатоступеневих бізнес-процесів. Така організація забезпечує слабку зв’язаність компонентів, дозволяє ізолювати відмови і гнучко масштабувати окремі частини системи.

На рівні логіки система включає наступні ключові компоненти:

- API Gateway (YARP) — центральна точка входу для клієнтських запитів. Gateway виконує маршрутизацію, агрегацію запитів. У проекті використано YARP (Yet Another Reverse Proxy) як ефективний та конфігурований компонент для маршрутизації HTTP/HTTPS-трафіку до внутрішніх сервісів або до зовнішніх кінцевих точок.

- UserService — сервіс управління обліковими записами користувачів: реєстрація, авторизація, керування профільними даними та видача JWT. Для взаємодії з іншими сервісами, де потрібна негайна відповідь, UserService надає gRPC-інтерфейс.
- RailwayConnectorService — адаптер для взаємодії з API Укрзалізниці; виконує синхронні gRPC-запити з метою отримання розкладу, перевірки наявності місць та актуальних тарифів.
- OrderService (Booking/Order Service) — центральний сервіс обробки замовлень. Керує створенням бронювань, координує виконання транзакцій у межах Saga, публікує відповідні події у шину повідомлень.
- PaymentService — сервіс обробки платежів, що взаємодіє з зовнішніми платіжними провайдерами; застосовує Outbox Pattern для гарантії доставлення повідомлень про статус транзакції.

Взаємодія між компонентами класифікується таким чином:

- Клієнт → API Gateway (YARP) — усі зовнішні запити (з браузера або мобільного додатка) надходять через Gateway. Gateway виконує валідацію, аутентифікацію, маршрутує запит до відповідного сервісу (через REST або проксувати до gRPC-шлюзу).
- Синхронні gRPC-виклики — використовуються між сервісами, яким потрібна низька латентність та строгий контракт (зокрема між UserService, RailwayConnectorService та деякими сервісами, що виконують оперативні запити).
- Асинхронні події через RabbitMQ — OrderService, PaymentService, координують довготривалі бізнес-процеси за допомогою подієвих повідомлень (BookingStarted, PaymentRequested, PaymentConfirmed, TicketIssued тощо

- Бази даних — кожен сервіс має власне сховище даних (реляційне або спеціалізоване), що мінімізує залежності на рівні зберігання і дозволяє оптимізувати модель даних під специфіку домену.

API Gateway (YARP) та розподіл навантаження дозволяють приховувати внутрішню структуру системи та централізувати політики безпеки. AnalyticsService забезпечує можливості для оперативного та ретроспективного аналізу роботи платформи, що необхідно для прийняття управлінських рішень та оптимізації алгоритмів бронювання.

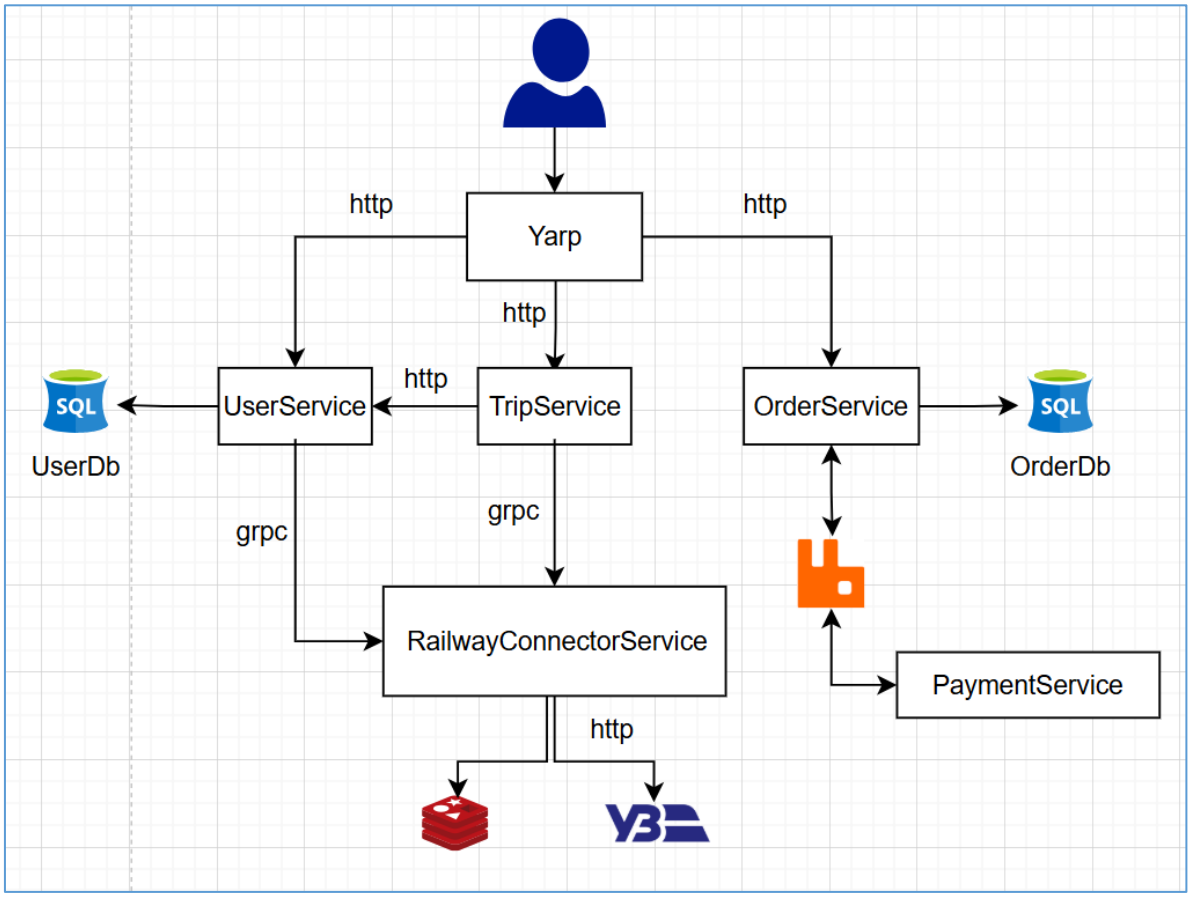


Рисунок 4 – Архітектура AutoTix

3.3 Опис кожного сервісу

Мікросервісна архітектура AutoTix передбачає розподіл системної логіки між окремими сервісами, кожен з яких реалізує визначений набір функцій та

працює у межах власної предметної області. Такий підхід дозволяє забезпечити незалежність компонентів, уникнути надмірних зв’язків та спростити подальшу підтримку й масштабування комплексу. Нижче наведено опис основних сервісів, що формують архітектурне ядро системи.

API Gateway (YARP). У межах розроблюваної системи API Gateway виконує роль єдиної точки входу для всіх запитів, що надходять від клієнтських застосунків. Це дозволяє централізувати маршрутизацію, автентифікацію, авторизацію, застосування політик доступу та загалом забезпечити захищену й уніфіковану взаємодію між користувачем та внутрішніми мікросервісами.

Для реалізації Gateway було обрано фреймворк YARP (Yet Another Reverse Proxy) — офіційне рішення від Microsoft, яке забезпечує розширювану та високу продуктивну маршрутизаційну інфраструктуру для .NET-сервісів. YARP дозволяє налаштовувати маршрути безпосередньо через конфігураційні файли, що спрощує інфраструктуру проєкту та зменшує обсяг коду. Архітектура Gateway поділена на такі основні компоненти:

Оскільки в даному проєкті Gateway не містить власних контролерів або бізнес-логіки, його архітектура залишається максимально простою і складається з двох ключових елементів:

appsettings.json — файл конфігурації, у якому описано всі маршрути, кластери внутрішніх сервісів, параметри проксування, таймаути та інші налаштування YARP.

Program.cs — точка запуску застосунку, де здійснюється реєстрація YARP, автентифікації, CORS та інших проміжних компонентів (middleware), а також підключення конфігурації маршрутизації.

Такий підхід дозволяє розділити відповідальність: конфігураційні зміни виконуються виключно через appsettings.json, а Program.cs відповідає за ініціалізацію та побудову конвеєра обробки HTTP-запитів.

Такий підхід мінімізує кількість коду, підвищує читабельність проєкту та полегшує його подальшу підтримку. Це робить YARP оптимальним вибором для побудови сучасних, легких і масштабованих API Gateway у .NET-екосистемі.

UserService. UserService відповідає за управління обліковими записами користувачів. До його основних завдань належать реєстрація, авторизація, керування профілем та формування токенів доступу. Сервіс взаємодіє з іншими компонентами системи через gRPC, забезпечуючи швидку перевірку даних користувача без потреби у дублюванні інформації в інших сервісах. Для високої ізоляції та зручності підтримки сервіс поділений на чотири основні шари: API, Application, Infrastructure та Domain. (див. Рис. 5)

1. UserService.API
    - Controllers – обробляють HTTP- та gRPC-запити, транслюючи їх у бізнес-логіку Application Layer.
    - Extensions – методи для налаштування middleware, маршрутизації, політик безпеки та обробки помилок.
2. UserService.Application
    - DTOs – об’єкти передачі даних між шарами та зовнішніми сервісами.
    - Services – бізнес-логіка для реєстрації, авторизації, оновлення профілю та роботи з токенами.
    - Interfaces – визначення контрактів для сервісів та репозиторіїв.
    - Extensions – допоміжні методи для конфігурації залежностей та middleware.
3. UserService.Infrastructure
    - Persistence/Repositories – реалізація доступу до бази даних (Entity Framework Core) та роботи з агрегатами користувача і ролями.
    - Security – компоненти для хешування паролів, генерації та валідації JWT.

- Configuration – налаштування підключень до БД та сторонніх сервісів (email, SMS).
- External – інтеграційні адаптери для сторонніх сервісів.
- Protos – визначення gRPC-контрактів для взаємодії з іншими мікросервісами.

4. UserService.Domain
   - Entities – доменні сутності користувача, ролей і прав доступу з інкапсуляцією бізнес-правил на рівні моделі.

Ця архітектура гарантує безпечну взаємодію сервісу з іншими компонентами системи та ізоляцію бізнес-логіки від інфраструктурних деталей.

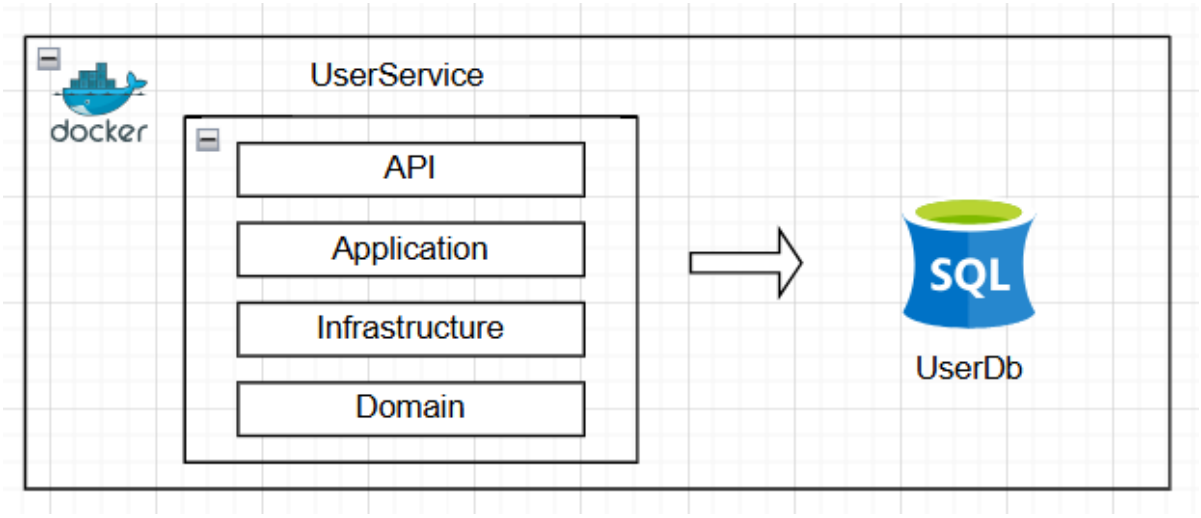


Рисунок 5 – Архітектура UserService

RailwayConnectorService

Цей сервіс виконує функцію адаптера між зовнішнім API Укрзалізниці та внутрішніми сервісами AutoTix. Він відповідає за виконання зовнішніх запитів, приведення отриманих даних до внутрішніх форматів, кешування часто використовуваних результатів та передачу інформації іншим сервісам системи. Основна взаємодія здійснюється через gRPC, що дозволяє мінімізувати затримки

при отриманні розкладів і даних про доступні місця. Архітектура поділена на API, Application, Infrastructure та Contracts. . (див. Рис. 6)

1. RailwayConnectorService.API
   - Controllers – обробляють HTTP-запити від внутрішніх сервісів або зовнішніх клієнтів.
   - Grpc – реалізація gRPC-серверу для швидкої синхронної взаємодії.
   - Protos – protobuf-контракти для взаємодії з іншими мікросервісами.
2. RailwayConnectorService.Application
   - Services – бізнес-логіка обробки даних про розклад поїздів, місця та тарифи.
   - Interfaces – контракти для сервісів та інтеграційних адаптерів.
   - Extensions – допоміжні методи для конфігурації залежностей і middleware.
3. RailwayConnectorService.Infrastructure
   - Configuration – налаштування підключень до зовнішнього API, баз даних та кешу.
   - External/Models – моделі даних від зовнішнього API.
   - External/Services – адаптери для інтеграції з API Укрзалізниці.
   - Services – внутрішні інфраструктурні сервіси (кешування, логування).
   - Extensions – допоміжні методи для налаштування middleware та сервісів.
4. RailwayConnectorService.Contracts
   - Models – спільні моделі для взаємодії з іншими сервісами.
   - Extensions – методи розширення для роботи з контрактами.

Такий підхід забезпечує швидку та надійну інтеграцію з зовнішніми даними, мінімізує дублювання та забезпечує масштабовану роботу внутрішніх сервісів.

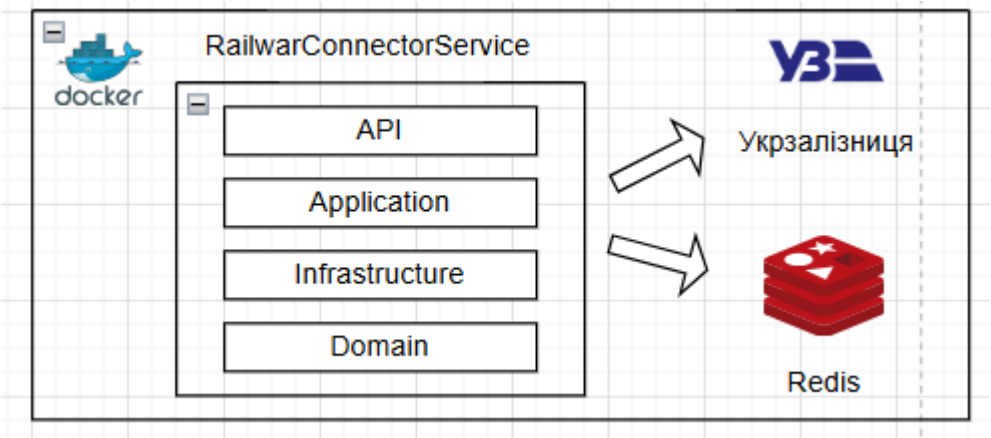


Рисунок 6 – Архітектура RailwayConnectorService

TripService. TripService відповідає за обробку запитів на пошук поїздок, рейсів, доступних місць та супутніх даних. Він отримує інформацію від RailwayConnectorService через gRPC, а також звертається до UserService для визначення параметрів користувача (наприклад, категорія пасажира, доступні знижки чи обмеження). Основне призначення TripService — виконання фільтрації, агрегації, сортування та нормалізації даних, отриманих із зовнішніх або внутрішніх джерел, для формування структурованих відповідей, оптимізованих для клієнтських застосунків.

Сервіс працює як проміжна логічна ланка між джерелом даних (RailwayConnectorService) та фронтендом, забезпечуючи узгодженість моделі подорожі, об’єднання різних типів даних (маршрути, сегменти, вагони, місця) та мінімізацію затримок при формуванні результатів пошуку. Архітектура TripService поділена на API, Application, Infrastructure та Contracts. (див. Рис. 7)

TripService.API

• Controllers – обробляють HTTP-запити від клієнтських застосунків, пов’язані з пошуком поїздок, отриманням інформації про сегменти, вагони та доступні місця. Забезпечують валідацію параметрів запиту та формують стандартизовані відповіді.

• Grpc – реалізація gRPC-клієнтів для взаємодії з RailwayConnectorService та іншими внутрішніми сервісами. Забезпечують високошвидкісний обмін даними при отриманні розкладу, вагонів та місць.

• Protos – protobuf-контракти, які описують структури даних для комунікації між TripService та RailwayConnectorService через gRPC. Дефінують моделі поїздок, вагонів, місць та маршрутів у форматі, оптимізованому для мережевої взаємодії.

2. TripService.Application

• Services – містять бізнес-логіку пошуку та обробки інформації про поїздки, маршрути, сегменти, вагони та місця. Реалізують агрегацію, фільтрацію, нормалізацію та сортування даних, отриманих із RailwayConnectorService та UserService. Також включають логіку визначення доступних опцій, розрахунку властивостей маршруту та обробку специфічних сценаріїв пошуку.

• Interfaces – визначають контракти для сервісів пошуку поїздок, взаємодії з RailwayConnectorService, обробки користувацьких параметрів, а також доступу до кешу та довідкових даних. Забезпечують можливість заміни реалізацій без зміни бізнес-логіки.

• Extensions – допоміжні методи, які спрощують конфігурацію залежностей, налаштування сервісів, реєстрацію кешування та мапінг моделей. Використовуються для інтеграції Application-рівня з API та Infrastructure.

3. TripService.Infrastructure

• Configuration – містить налаштування для взаємодії з RailwayConnectorService через gRPC, параметри кешування результатів пошуку, конфігурацію retry-політик, таймаутів та обмежень для зовнішніх викликів.

• External/Models – моделі, що відображають структури даних, отримані через RailwayConnectorService (розклад, сегменти, вагони, місця). Слугують адаптерами між protobuf-моделями та внутрішніми DTO застосунку.

• External/Services – реалізації клієнтів, що звертаються до RailwayConnectorService або UserService. Забезпечують конвертацію protobuf-повідомлень у внутрішні моделі TripService та реалізують логіку інтеграції (retry, тротлінг, кешування).

• Services – інфраструктурні сервіси для роботи з кешем (наприклад, Redis[12]), логуванням, статистикою запитів та оптимізацією частоти звернень до зовнішніх API. Тут розміщені сервіси, які не містять бізнес-логіки, але забезпечують стабільність та продуктивність роботи TripService.

• Extensions – допоміжні методи, які спрощують реєстрацію gRPC-клієнтів, кешування, політик відмовостійкості, middleware, а також інтеграцію інфраструктури із застосунком.

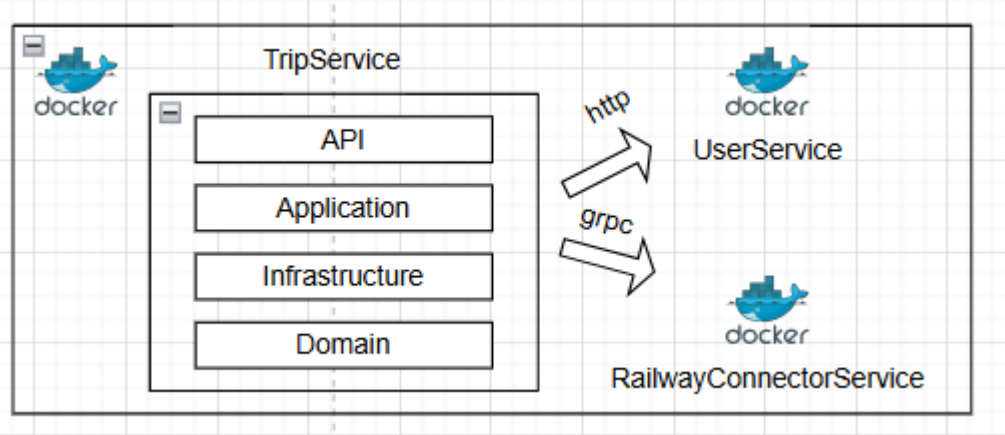


Рисунок 7 – Архітектура TripService

OrderService

OrderService є ключовим елементом бізнес-логіки та координує процес створення, бронювання та підтвердження замовлень. Він реалізує механізм Saga, який забезпечує узгодженість довготривалих бізнес-операцій. Сервіс публікує події в RabbitMQ[13] та реагує на повідомлення від інших сервісів, підтримуючи стан замовлення до його завершення або скасування. Архітектура OrderService розділена на API, Application, Domain та Infrastructure. (див. Рис. 8)

1. OrderService.API

• Controllers – обробляють HTTP-запити від клієнтських застосунків або внутрішніх сервісів, конвертують запити у команди або запити Application-рівня та повертають результати у вигляді DTO.

• BackgroundJobs – реалізують асинхронну обробку тривалих або повторюваних операцій, наприклад перевірку статусів замовлень або повторну обробку подій Saga.

• Extensions – містять методи розширення для конфігурації залежностей, реєстрації middleware та налаштування фонових задач.

2. OrderService.Application

• Commands / Handlers – реалізують бізнес-операції, пов’язані зі створенням, бронюванням та скасуванням замовлень, обробляють команди, що надходять з API.

• Queries / Handlers – обробляють запити на отримання інформації про замовлення, стан бронювань та доступні ресурси, не змінюючи стан системи.

• Dtos – моделі передачі даних між API та Application-рівнем для стабільності контрактів і узгодженості форматів.

• Events – внутрішні та інтеграційні події, які публікуються в RabbitMQ під час виконання Saga.

• Interfaces – контракти для репозиторіїв, менеджера Saga, інтеграційних сервісів та RabbitMQ.

3. OrderService.Domain

• Aggregates / OrderAggregate – основна доменна модель замовлення, яка інкапсулює логіку зміни станів, валідацію даних та генерацію доменних подій.

• Base / Interfaces – базові абстракції та інтерфейси доменного рівня, включно з Entity, ValueObject, AggregateRoot, IRepository та IDomainEvent.

• Enum – перелічення для статусів замовлення та кроків Saga, наприклад OrderStatus (Pending, Reserved, Paid, Cancelled).

4. OrderService.Infrastructure

• Configuration – налаштування підключень до бази даних, RabbitMQ, параметрів Saga та зовнішніх сервісів.

• Extensions – допоміжні методи для реєстрації інфраструктурних сервісів, middleware та інтеграції з API та Application-рівнем.

• Persistence – реалізація доступу до даних, включно з конфігураціями моделей, міграціями та репозиторіями.

• RabbitMq / Connection / Interface – інфраструктура для взаємодії з RabbitMQ, включно з продюсерами, консьюмерами, повторною доставкою та обробкою помилок.

Такий підхід дозволяє забезпечити централізовану координацію всіх етапів життєвого циклу замовлення, підтримку узгодженості даних між мікросервісами та надійну обробку довготривалих бізнес-операцій. Використання патерну Saga і публікація подій у RabbitMQ[13] гарантують відновлюваність процесів у разі помилок, а багатошарова архітектура (API, Application, Domain та Infrastructure) забезпечує розділення відповідальностей, масштабованість та легкість підтримки сервісу.

Таким чином, OrderService виступає ключовим компонентом системи AutoTix, який інтегрує роботу інших сервісів, оптимізує обробку замовлень та формує надійну платформу для подальшого розвитку функціоналу бронювання та продажу квитків.

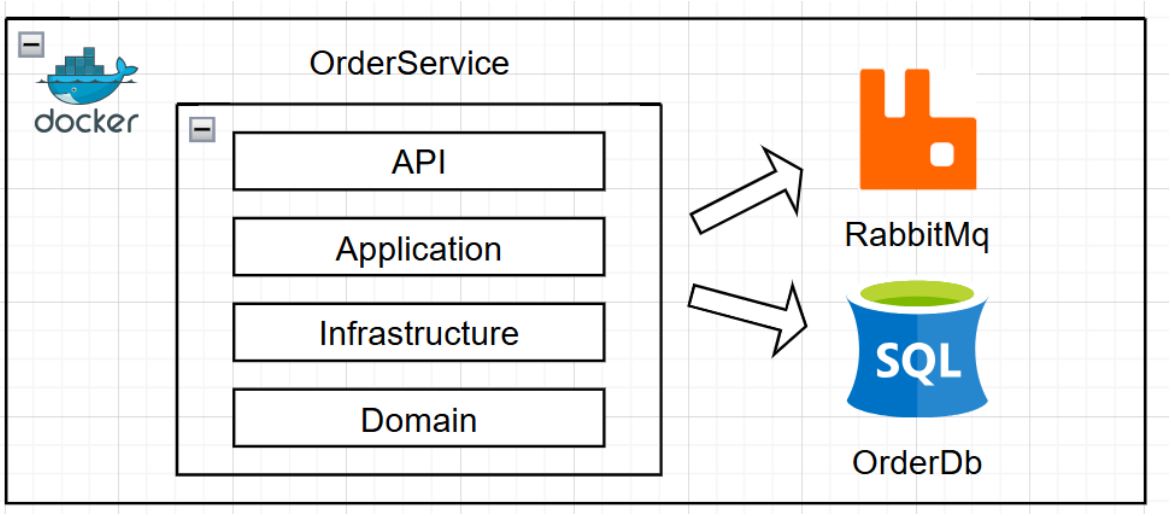


Рисунок 8 – *Архітектура OrderService*

PaymentService. PaymentService здійснює обробку платежів, взаємодіючи з зовнішніми платіжними системами або їх симуляторами. Він відповідає за перевірку платіжних даних, проведення транзакцій та повідомлення OrderService про результат оплати. Використання асинхронного обміну подіями через RabbitMQ[13] дозволяє уникнути блокувань, забезпечує стійкість сервісу під навантаженням та гарантує відновлюваність процесів у разі помилок. Архітектура PaymentService розділена на API, Application та Infrastructure. (див. Рис. 9)

1. PaymentService.API

• Controllers – обробляють HTTP-запити від клієнтських застосунків або внутрішніх сервісів, конвертують їх у команди чи запити Application-рівня та повертають результати у вигляді DTO.

• BackgroundJobs – виконують фонові або повторювані операції, наприклад перевірку статусів платежів, повторні транзакції або обробку таймаутів.

• Extensions – містять допоміжні методи для налаштування залежностей, реєстрації middleware та фонових задач.

2. PaymentService.Application

• Events – доменні та інтеграційні події, які публікуються у RabbitMQ[13] для інформування OrderService або інших зацікавлених сервісів про результати оплати (наприклад, PaymentSucceededEvent, PaymentFailedEvent).

• Services – реалізують бізнес-логіку оплати, перевірку платіжних даних, ініціювання транзакцій та обробку відповіді від платіжних систем.

• Interfaces – контракти для інтеграції з RabbitMQ[13], зовнішніми платіжними системами та внутрішніми сервісами.

• Extensions – допоміжні методи для конфігурації залежностей, реєстрації сервісів та middleware.

3. PaymentService.Infrastructure

• Producers – компоненти для публікації подій у RabbitMQ[13], що інформують інші сервіси про результати платежів.

• Consumers – компоненти для підписки на події інших сервісів, наприклад отримання команд на ініціювання платежу від OrderService.

• RabbitMq / Interface – абстракція для взаємодії з брокером повідомлень, забезпечує відправку та обробку подій з підтримкою повторної доставки та контролю помилок.

• Extensions – допоміжні методи для налаштування підключень, реєстрації продюсерів та консьюмерів, інтеграції з Application-рівнем.

Такий підхід забезпечує надійну та масштабовану обробку платежів, із чітким розділенням відповідальностей між шарами, асинхронною взаємодією з OrderService та іншими компонентами системи, а також високою стійкістю до навантажень і помилок зовнішніх платіжних систем.

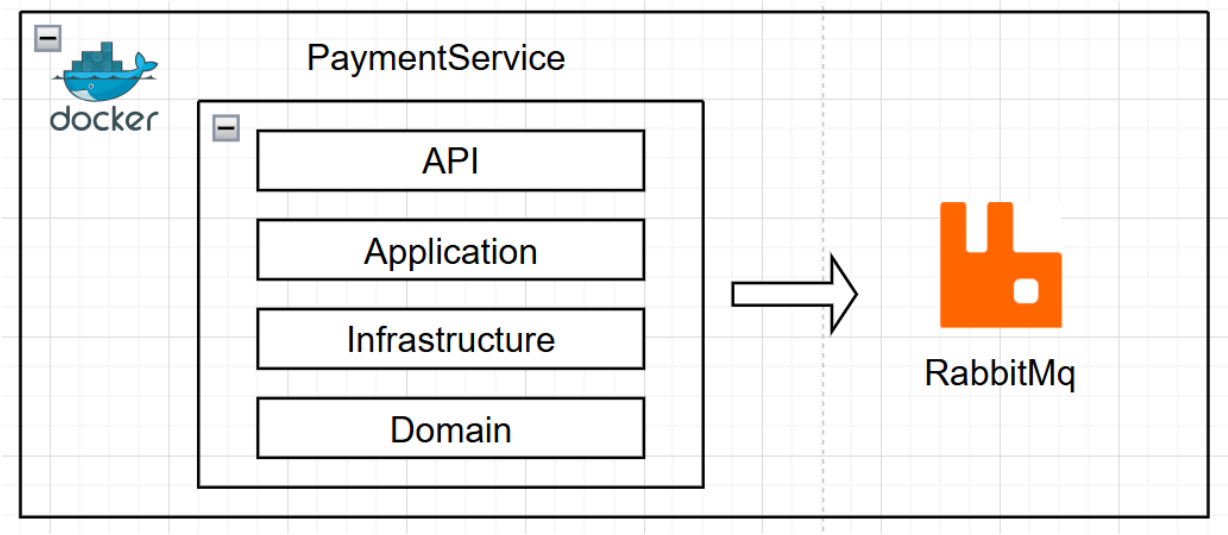


Рисунок 9 –*Архітектура PaymentService*

3.4. Проєктування баз даних

У системі AutoTix використовується комбінація реляційної бази даних (SQL) та кешу на основі Redis[12] для забезпечення високої швидкодії та масштабованості при обробці замовлень та пошуку поїздок. Розглянемо детально кожну з систем збереження даних.

3.4.1. Реляційна база даних (SQL)

Реляційна база даних використовується для зберігання стійких та критично важливих даних, таких як інформація про користувачів, замовлення, платежі та бронювання квитків. Всі таблиці спроектовані із врахуванням принципів нормалізації, цілісності даних та підтримки транзакційного обміну між мікросервісами

Основні таблиці

1. Лістинг 1 Users – таблиця користувачів системи

```
CREATE TABLE Users (
```

```
    Id BIGINT PRIMARY KEY,
    Phone NVARCHAR(50) NOT NULL,
    CreatedAt DATETIMEOFFSET NOT NULL
);
```

2. Лістинг 2 UzCredentials – збереження токенів та профілів користувача для доступу до API Укрзалізниці

```
CREATE TABLE UzCredentials (
Id BIGINT PRIMARY KEY,
UserId BIGINT NOT NULL REFERENCES Users(Id),
EncryptedAccessToken NVARCHAR(MAX) NOT NULL,
IssuedAt DATETIMEOFFSET NOT NULL,
ExpiresAt DATETIMEOFFSET NOT NULL,
RawProfileJson NVARCHAR(MAX),
DeviceName NVARCHAR(100) NOT NULL,
IsActive BIT NOT NULL DEFAULT 1,
CreatedAt DATETIMEOFFSET NOT NULL
);
```

3. Лістинг 3 Orders – основна таблиця замовлень

```
CREATE TABLE Orders (
Id UNIQUEIDENTIFIER PRIMARY KEY,
UserId BIGINT NOT NULL REFERENCES Users(Id),
Status INT NOT NULL,
TotalAmount DECIMAL(18,2) NOT NULL
);
```

4. Лістинг 4 TicketReservations – бронювання окремих квитків у замовленні

```
CREATE TABLE TicketReservations (
Id UNIQUEIDENTIFIER PRIMARY KEY,
```

```
OrderId UNIQUEIDENTIFIER NOT NULL REFERENCES Orders(Id),
Quantity INT NOT NULL,
StationFrom NVARCHAR(100) NOT NULL,
StationTo NVARCHAR(100) NOT NULL,
DepartAt BIGINT NOT NULL,
ArriveAt BIGINT NOT NULL,
WagonName NVARCHAR(50) NOT NULL,
Price DECIMAL(18,2) NOT NULL
);
```

Опис зв’язків:

- Користувачі (Users) можуть мати кілька облікових записів Укрзалізниці (UzCredentials) – **зв’язок один-до-багатьох**.
- Замовлення (Orders) належать користувачеві – **зв’язок один-до-багатьох**.
- Замовлення містять один або кілька бронювань (TicketReservations) – **зв’язок один-до-багатьох**.

Такий підхід дозволяє підтримувати **цілісність даних**, контролювати стан замовлень, обробку платежів та історію транзакцій у SQL.

3.4.2. Кешування даних (Redis)

Для забезпечення високої швидкодії та зменшення навантаження на зовнішнє API Укрзалізниці в RailwayConnectorService використовується Redis[12] як кеш.

Що зберігається у Redis (див. Табл. 1)

1. Користувачі та їхні дані – збереження користувачів, які вже робили запити, щоб уникнути повторного надсилання запитів до бази SQL чи зовнішніх сервісів.
2. Розклади поїздок та маршрути – інформація про потяги, станції, вагони та місця.

3. Стан та результати попередніх запитів – кешування часто використовуваних запитів для прискорення формування результатів пошуку.

Таблиця 1 – Ключі зберігання в Redis

Ключ	Значення	TTL (час життя)
auth:sms:{request.Phone}	JSON із даними користувача	1 хвилина
auth:login:{request.Phone}:{request.Code}	JSON із даними користувача	5 хвилин
stations:all	JSON зі списком всіх станцій	1 хвилина
stations:boards	JSON із даними по табло всіх станцій	1 хвилина
station:board:{id}	JSON із даними по конкретному таблу станції	1 хвилина
trip:search:{stationFromId}:{stationToId}:{date}:{withTransfers}	JSON із деталями пошуку поїздки (маршрути, сегменти, вагони, місця)	30 секунд
trip:direct:{tripId}	JSON із даними прямого рейсу	30 секунд
trip:dates:{stationFromId}:{stationToId}	JSON із датами відправлень поїздів	30 секунд
trip:wagon:{tripId}:{wagonClass}	JSON із даними вагонів конкретного класу рейсу	30 секунд

3.4.3. Взаємодія між доменами

Взаємодія між доменами в системі AutoTix є ключовим аспектом забезпечення цілісності, узгодженості та швидкодії при обробці запитів клієнтів. Система побудована за принципом мікросервісної архітектури, де кожен домен

відповідає за конкретний набір функціональних можливостей, а комунікація між ними здійснюється як синхронно, так і асинхронно з використанням спеціалізованих механізмів обміну даними.

У системі виділяються такі основні домени: UserService, RailwayConnectorService, TripService, OrderService та PaymentService. Кожен домен має чітко визначену відповідальність, а зв’язки між ними організовані таким чином, щоб уникнути дублювання логіки та забезпечити високу швидкодію системи.

Синхронна взаємодія. Синхронна взаємодія між сервісами реалізована переважно через gRPC та HTTP-запити.

TripService та RailwayConnectorService: TripService отримує актуальні дані про поїздки, маршрути, вагони та місця через gRPC. Це дозволяє мінімізувати затримки при обробці великої кількості запитів і забезпечує сувору типізацію даних, що критично для формування структурованих відповідей. Кожен запит TripService до RailwayConnectorService може містити параметри фільтрації, включаючи станції відправлення та прибуття, дату поїздки та наявність пересадок.

TripService та UserService: Для визначення персональних параметрів користувача, таких як категорія пасажира, доступні знижки або обмеження, TripService звертається до UserService через HTTP-запити. Дані користувача повертаються у вигляді DTO, після чого TripService об’єднує їх із даними про поїздки, формуючи фінальну відповідь для клієнта.

Такий підхід дозволяє розділити відповідальність доменів: TripService відповідає лише за агрегацію та нормалізацію даних про поїздки, а UserService – за управління обліковими записами та персоналізацію даних користувача.

Асинхронна взаємодія. Асинхронна взаємодія організована через RabbitMQ[13], що забезпечує відокремленість сервісів та стійкість системи до навантажень.

OrderService та PaymentService: При створенні замовлення OrderService публікує подію OrderPaymentStartedEvent, яку споживає PaymentService для проведення транзакції. У разі успішної оплати PaymentService публікує OrderPaidEvent, а у випадку помилки – OrderPaymentFailedEvent. OrderService відповідно змінює стан замовлення на Paid або Failed.

OrderService та TripService / RailwayConnectorService: Після успішного бронювання квитків OrderService може повідомляти TripService про зміни доступних місць, щоб оновити кеш та запобігти дублюванню бронювання.

Координація Saga: Взаємодія між OrderService, PaymentService та іншими сервісами відбувається в рамках механізму Saga, що дозволяє керувати довготривалими бізнес-процесами, забезпечуючи консистентність даних навіть у випадку відмови окремих сервісів.

Асинхронна взаємодія дозволяє:

- забезпечити відмовостійкість – сервіси можуть обробляти повідомлення, навіть якщо один із сервісів тимчасово недоступний;
- уникнути блокувань основного потоку – події обробляються незалежно від запиту клієнта;
- підтримувати масштабованість – нові інстанси сервісів можуть підписуватися на ті самі події без зміни логіки інших сервісів.

Використання кешу для оптимізації взаємодії. Для зменшення кількості запитів до зовнішніх API та підвищення швидкодії застосовується Redis[12].

RailwayConnectorService зберігає у кеші інформацію про користувачів, станції, розклади поїздок, маршрути та вагони. Це дозволяє уникати повторних запитів до API Укрзалізниці і прискорює обробку запитів.

TripService отримує дані спочатку з Redis[12]; якщо дані відсутні або прострочені, він робить запит до RailwayConnectorService і одразу оновлює кеш.

Кожен тип даних має свій TTL (час життя у кеші), що дозволяє підтримувати актуальність інформації: станції та табло – 1 хвилина, розклади, дати відправлень та вагони – 30 секунд, дані користувача – 1 день.

Кешування через Redis[12] забезпечує:

- зменшення навантаження на зовнішні системи;
- прискорення формування результатів для клієнта;
- зменшення дублювання логіки у сервісах;
- підтримку масштабованості системи при великій кількості одночасних запитів.

Взаємодія доменів у реальному сценарії. При створенні замовлення процес виглядає наступним чином:

1. Клієнт через фронтенд робить запит на пошук поїздки;
2. TripService отримує розклади через gRPC від RailwayConnectorService та персональні дані від UserService через HTTP;
3. TripService формує структуровану відповідь та відправляє її клієнту;
4. Користувач обирає поїздку та створює замовлення;
5. OrderService створює замовлення, публікує подію OrderPaymentStartedEvent у RabbitMQ;
6. PaymentService обробляє платіж, публікує OrderPaidEvent або OrderPaymentFailedEvent;
7. OrderService оновлює стан замовлення та, за необхідності, повідомляє TripService для корекції доступних місць у кеші;
8. Весь процес забезпечує узгодженість стану між доменами, навіть при високому навантаженні та великій кількості паралельних запитів.

Взаємодія доменів у системі AutoTix організована за принципами розділення відповідальності, масштабованості та узгодженості даних. Синхронні виклики забезпечують актуальні дані для користувача, асинхронні події через

RabbitMQ[13] дозволяють керувати бізнес-процесами та платежами, а кешування через Redis значно прискорює доступ до часто використовуваних даних. Така архітектура дозволяє ефективно управляти складними взаємозв'язками між доменами і підтримує стабільну роботу системи під високим навантаженням.

3.5. Реалізація ключових патернів архітектури

В системі AutoTix для забезпечення надійності, узгодженості даних та масштабованості реалізовані сучасні архітектурні патерни, які дозволяють ефективно обробляти бізнес-процеси та взаємодіяти між мікросервісами. Основні патерни, застосовані в проекті, включають Saga, Event-driven обмін та CQRS.

3.5.1. Saga для процесу бронювання

Патерн Saga використовується для управління довготривалими бізнес-процесами, які включають кілька незалежних сервісів і потребують узгодженості стану у разі успіху або відмови на будь-якому етапі. У AutoTix цей патерн реалізований у OrderService, що координує процес створення, бронювання та оплати замовлення.

Основні характеристики реалізації Saga:

1. Розбиття процесу на атомарні кроки

Кожна операція у процесі бронювання розділена на незалежні кроки:

- Створення замовлення;
- Бронювання місць у TripService;
- Обробка платежу через PaymentService;
- Підтвердження або скасування замовлення.

2. Асинхронне виконання кроків

Кожен крок реалізується як команда або подія, що обробляється асинхронно через RabbitMQ[13]. Це дозволяє не блокувати основний потік і забезпечує масштабованість.

3. Відкат у разі помилки

Якщо на будь-якому етапі виникає помилка (наприклад, відмова платежу), Saga запускає компенсаційні дії, які повертають систему у консистентний стан:

- Скасування бронювання місць;
- Повернення стану замовлення в OrderService у статус Failed;
- Генерація відповідних подій для споживачів.

Події Saga

- OrderPaymentStartedEvent – початок оплати;
- OrderPaidEvent – успішна оплата;
- OrderPaymentFailedEvent – відмова оплати;

Реалізація Saga забезпечує узгодженість стану замовлень і бронювання навіть у випадку тимчасових збоїв у окремих сервісах, що критично для бізнес-процесу продажу квитків.

3.5.2. Event-driven обмін

Система AutoTix побудована на принципі Event-driven архітектури, що дозволяє сервісам взаємодіяти через події без прямого синхронного виклику. Це забезпечує високий рівень відокремленості сервісів, гнучкість та масштабованість.

Основні особливості реалізації Event-driven обміну:

1. Асинхронна комунікація через RabbitMQ
   - Всі ключові події (створення замовлення, початок та завершення платежу, бронювання квитків) публікуються в RabbitMQ.

- Інші сервіси підписуються на відповідні події та обробляють їх незалежно від джерела.

2. Підтримка бізнес-логіки
   - OrderService публікує події при зміні стану замовлення;
   - PaymentService споживає події і виконує транзакції, повертаючи результат у вигляді подій;
3. Компенсаційні дії та повторна доставка
   - RabbitMQ забезпечує надійну доставку подій із повторними спробами у разі помилок;
   - Компенсаційні події використовуються для відкату операцій у рамках Saga.
4. Відокремлення доменів
   - Event-driven підхід дозволяє кожному сервісу залишатися незалежним та не залежати від доступності інших сервісів у момент виконання операції.

3.5.3. CQRS у сервісах

Патерн CQRS (Command Query Responsibility Segregation) застосовується для розділення операцій зміни стану (Command) та отримання даних (Query). У системі AutoTix CQRS реалізований у OrderService та частково у TripService для забезпечення високої продуктивності та узгодженості даних.

Реалізація CQRS:

1. Command – модуль відповідає за зміни стану системи:
   - Створення нового замовлення;
   - Додавання броні квитків;
   - Маркування замовлення як Processing, Paid або Failed.

- Command обробляється через відповідні Handlers у Application-рівні та публікує події для інших сервісів.

2. Query – модуль відповідає за отримання інформації без зміни стану:
   - Запити на отримання деталей замовлення;
   - Перевірка доступних місць у поїздах;
   - Формування агрегованих відповідей для клієнта.
   - Queries обробляються окремо від Command, що дозволяє оптимізувати запити до бази даних та кешу.
3. Переваги CQRS
   - Висока продуктивність – запити не блокуються змінами стану;
   - Масштабованість – командні та запитні потоки можна масштабувати окремо;
   - Чітке розділення відповідальності – Command відповідає лише за бізнес-логіку, Query – лише за отримання даних;
   - Простота інтеграції з кешем і Event-driven механізмом – зміни стану публікуються у вигляді подій, які споживають інші сервіси або оновлюють кеш.

3.6 Реалізація інтеграцій з API Укрзалізниці

У системі AutoTix інтеграція з зовнішнім API Укрзалізниці є критичною для забезпечення актуальної інформації про розклади поїздів, доступні місця та тарифи. Для реалізації цієї інтеграції використовується сервіс RailwayConnectorService, який виступає адаптером між внутрішніми сервісами та зовнішнім API. Архітектурно інтеграція побудована на окремих WebService-класах, що інкапсулюють HTTP-запити, обробку відповідей та кешування результатів.

Основні підсистеми інтеграції поділені за функціоналом:

1. AuthWebService – відповідає за автентифікацію користувачів у системі Укрзалізниці. Забезпечує відправку SMS-кодів на телефон користувача та виконання логіну з обробкою токенів доступу. Методи:

- SendSmsAsync(SendSmsRequest request) – відправляє SMS з кодом авторизації на номер користувача.
- LoginAsync(LoginRequest request) – виконує авторизацію за кодом із SMS і повертає токен доступу.
- LogoutAsync(LogoutRequest request) – забезпечує вихід користувача з системи.

Всі запити виконуються через базовий клас BaseWebService, що гарантує єдину обробку HTTP-клієнта, заголовків, логування та десеріалізацію відповідей.

2. StationWebService – відповідає за отримання інформації про станції та станційні табло:

- GetStationsAsync(string uzAccessToken) – повертає список всіх станцій Укрзалізниці.
- GetStationBoardsAsync(string uzAccessToken) – повертає інформацію про станційні табло, необхідні для відображення руху поїздів.
- GetStationBoardAsync(long id, string uzAccessToken) – отримує інформацію про конкретне табло станції.

Для підвищення продуктивності виклики даних кешуються через Redis, що дозволяє мінімізувати кількість повторних запитів до зовнішнього API і швидко повертати результат внутрішнім сервісам.

3. TripWebService – відповідає за роботу з поїздами та маршрутами:

- GetTripAsync(long stationFromId, long stationToId, string date, string uzAccessToken, bool withTransfers) – отримує інформацію про рейси між станціями.
- GetTripAsync(long tripId, string uzAccessToken) – повертає детальну інформацію про конкретний рейс.
- GetDepartureDatesAsync(long stationFromId, long stationToId, string uzAccessToken) – отримує доступні дати відправлення.
- GetWagonByClassAsync(long tripId, string wagonClass, string uzAccessToken) – повертає інформацію про вагони конкретного класу.

Усі запити обробляються асинхронно та підтримують кешування результатів, щоб скоротити навантаження на API Укрзалізниці та забезпечити швидку відповідь для фронтенд-застосунків.

3.6.1 Архітектурні особливості інтеграції

1. Базовий клас WebService

Всі інтеграційні сервіси наслідуються від BaseWebService, що дозволяє централізовано реалізувати:

- Формування HTTP-запитів (GET, POST);
- Додавання заголовків, включаючи Authorization та спеціальні заголовки Укрзалізниці (x-session-id, x-client-locale, x-user-agent);
- Логування запитів та відповідей через Serilog;
- Обробку помилок і десеріалізацію JSON-відповідей.

Завдяки такій архітектурі підвищується повторне використання коду та спрощується підтримка інтеграції**.**

2. Кешування даних через Redis

Для зменшення кількості HTTP-запитів до API Укрзалізниці та прискорення роботи сервісів використовується Redis.

- Ключі зберігаються у форматі, який дозволяє швидко знаходити дані за типом (станції, маршрути, рейси).
- TTL (час життя кешу) налаштовується відповідно до характеру даних: часто змінювані маршрути мають короткий TTL (30 секунд – 1 хвилина), а статичні дані, такі як список станцій, зберігаються довше (24 години).
- При отриманні даних сервіс спочатку перевіряє кеш і повертає результат, якщо він існує. В іншому випадку виконується запит до зовнішнього API і результат зберігається в Redis.

3. Асинхронна взаємодія та обробка даних

Усі методи WebService реалізовані асинхронно, що забезпечує:

- Мінімальні затримки при обробці запитів;
- Можливість одночасної обробки великої кількості запитів;
- Підтримку масштабованості при високому навантаженні.

4. Безпечна передача даних

- Використовується токен доступу (Bearer) для автентифікації користувача;
- Кожен запит містить необхідні HTTP-заголовки для відповідності вимогам API Укрзалізниці;
- Обробка помилок здійснюється централізовано з логуванням через Serilog для подальшого аналізу та моніторингу.

3.6.2 Потоки даних у системі

Інтеграція з API Укрзалізниці побудована за принципом adapter pattern:

1. Запит користувача -> TripService -> RailwayConnectorService -> TripWebService / StationWebService / AuthWebService -> API Укрзалізниці.

2. Отримані дані: десеріалізуються, приводяться до внутрішніх моделей, кешуються в Redis, а потім передаються споживачам.
3. Кешовані дані можуть повторно використовуватися для інших запитів, що значно зменшує кількість зовнішніх викликів та підвищує швидкість роботи системи.

Таким чином, RailwayConnectorService виконує роль посередника, ізолятора та кешуючого шару, забезпечуючи надійну та ефективну інтеграцію з API Укрзалізниці.

3.7 Реалізація фронтенд частини

Фронтенд частина системи AutoTix реалізована на основі сучасного фреймворку React з використанням TypeScript[17], що забезпечує типобезпечність і підвищує стабільність при розробці. Основною метою фронтенд модуля є надання користувачу інтуїтивно зрозумілого та функціонального інтерфейсу для пошуку поїздок, формування замовлень, управління кошиком, проведення оплати та перегляду персональної інформації. Взаємодія фронтенду з бекендом здійснюється через набір клієнтських сервісів, які централізовано обробляють HTTP-запити та gRPC-виклики.

Архітектура фронтенду побудована за модульним принципом і забезпечує розділення відповідальності між компонентами, сторінками та сервісами, що сприяє підвищенню масштабованості та підтримуваності проекту.

3.7.1. Структура React-проєкту

Структура фронтенд-проєкту виглядає наступним чином:

- node_modules – директорія з пакетами та залежностями, встановленими через npm або yarn.

- public – статичні ресурси, які безпосередньо використовуються браузером (HTML-шаблони, favicon, manifest).
- src – основна директорія з вихідним кодом застосунку:
- api – реалізація клієнтських сервісів для взаємодії з бекенд-сервісами:

1. services – сервіси для доступу до бекенд API: authService.ts, orderService.ts, stationService.ts, tripService.ts.
2. httpClient.ts – централізований HTTP-клієнт, який забезпечує налаштування заголовків, передачу JWT-токенів та обробку помилок.
3. index.ts – точка експорту API-сервісів для використання у компонентах.

- assets – графічні ресурси, зображення, стилі та іконки.
- components – повторно використовувані UI-компоненти:
- auth – компоненти для реалізації форм авторизації та реєстрації.
- layout – компоненти, що відповідають за структуру сторінок (Header, Sidebar, Footer).
- order – компоненти для роботи з замовленнями та кошиком.
- station – компоненти для відображення станцій та станційних табло.
- trip – компоненти для відображення поїздок та результатів пошуку рейсів.
- ui – універсальні елементи інтерфейсу (кнопки, інпути, модальні вікна).
- pages – сторінки застосунку:

1. LoginPage – сторінка авторизації користувача.
2. DashboardPage – головна сторінка користувача, що містить інформацію про замовлення та швидкий доступ до пошуку поїздок.
3. OrderPage – сторінка формування замовлень та роботи з кошиком.

4. StationPage – сторінка відображення станцій та станційних табло.
5. TripPage – сторінка результатів пошуку поїздок та доступних місць.

- types – визначення типів TypeScript для забезпечення строгості та узгодженості даних: auth, order, station, trip.
- utils – допоміжні функції для форматування даних, роботи з датами, числовими значеннями та константами.

Дана структура дозволяє чітко розділяти відповідальність між компонентами, сторінками та сервісами, забезпечує повторне використання коду та полегшує масштабування проекту.

3.7.2 Сторінки авторизації та привітального інтерфейсу

LoginPage – реалізує процес авторизації користувача за допомогою номера телефону та SMS-коду. Взаємодія здійснюється через authService, який обробляє запити до бекенд-сервісу RailwayConnectorService. Для забезпечення безпеки використовується JWT-токен, що зберігається у локальному сховищі браузера.

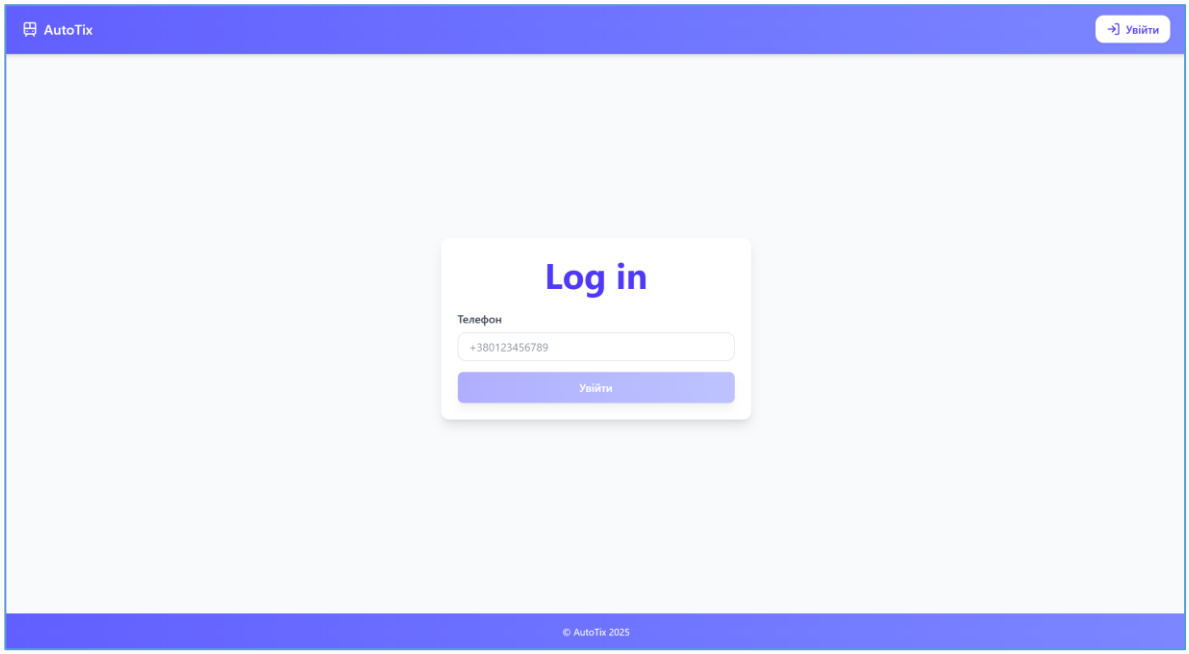


Рисунок 10 –Сторінка LogIn



Рисунок 11 – Popup введення коду для входу

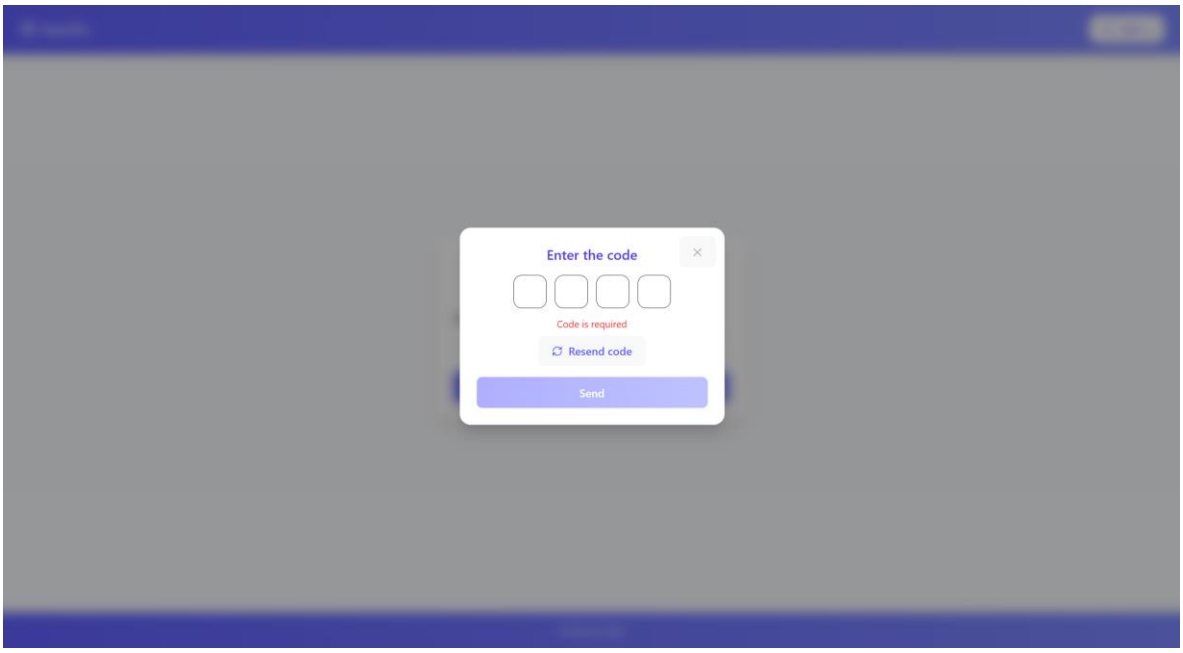


Рисунок 12 – Валідація коду



Рисунок 13 – Спроба отримати новий код

Головна сторінка застосунку DashboardPage виконує роль привітального інтерфейсу для користувача після завантаження застосунку. Вона забезпечує швидкий старт роботи з системою, надаючи можливість перейти до функціоналу пошуку поїздок та перегляду станцій.

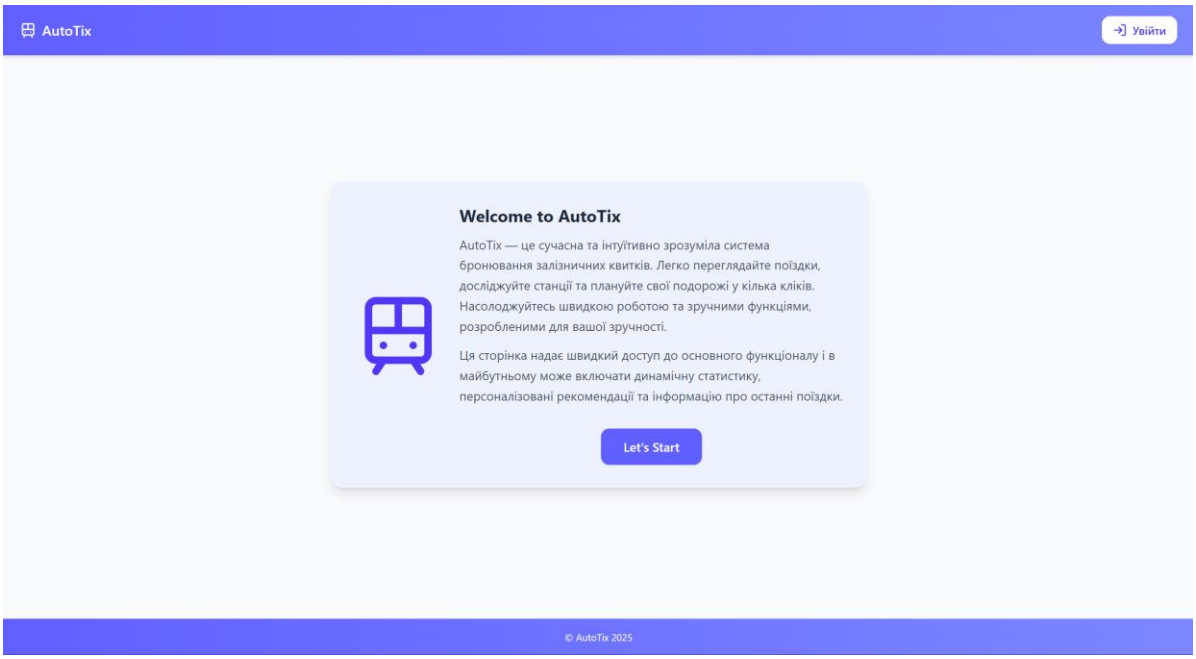


Рисунок 14 – Сторінка Dashboard неавторизованого користувача

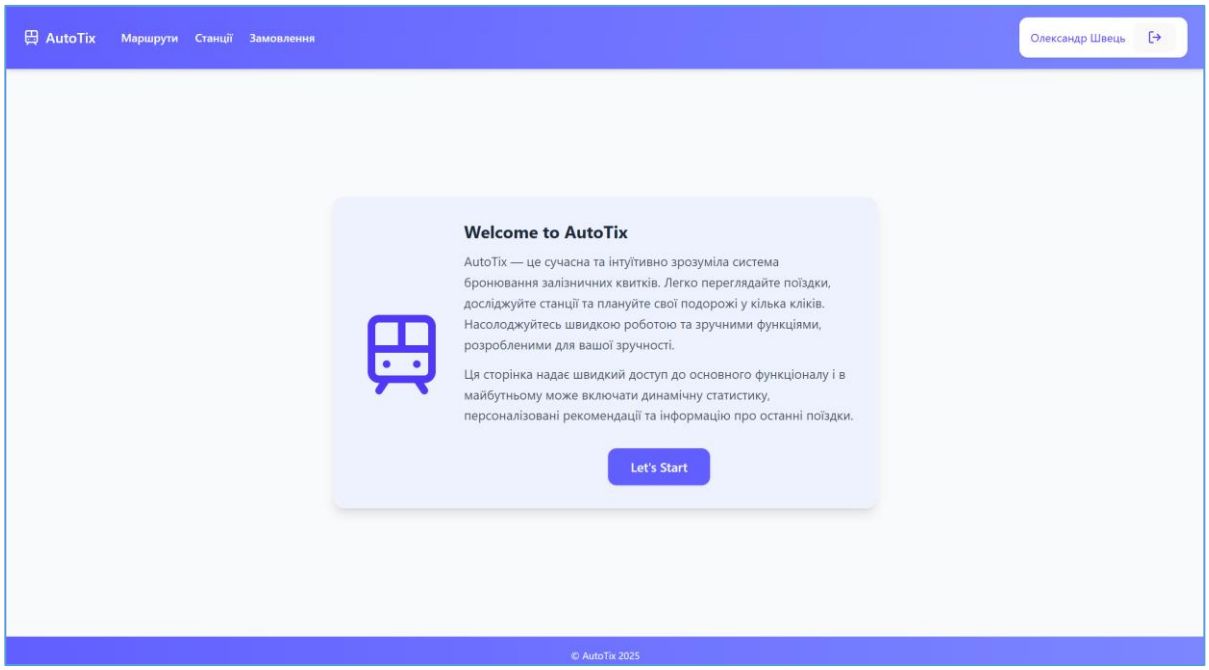


Рисунок 15 – Сторінка Dashboard авторизованого користувача

3.7.3 Робота з замовленнями та оплатою

OrderPage – сторінка формування замовлень:

- Користувач може обирати поїздки.
- Використовуються сервіси tripService та stationService для отримання даних про поїздки та станції.
- Після формування замовлення дані передаються до orderService для збереження та подальшої обробки.

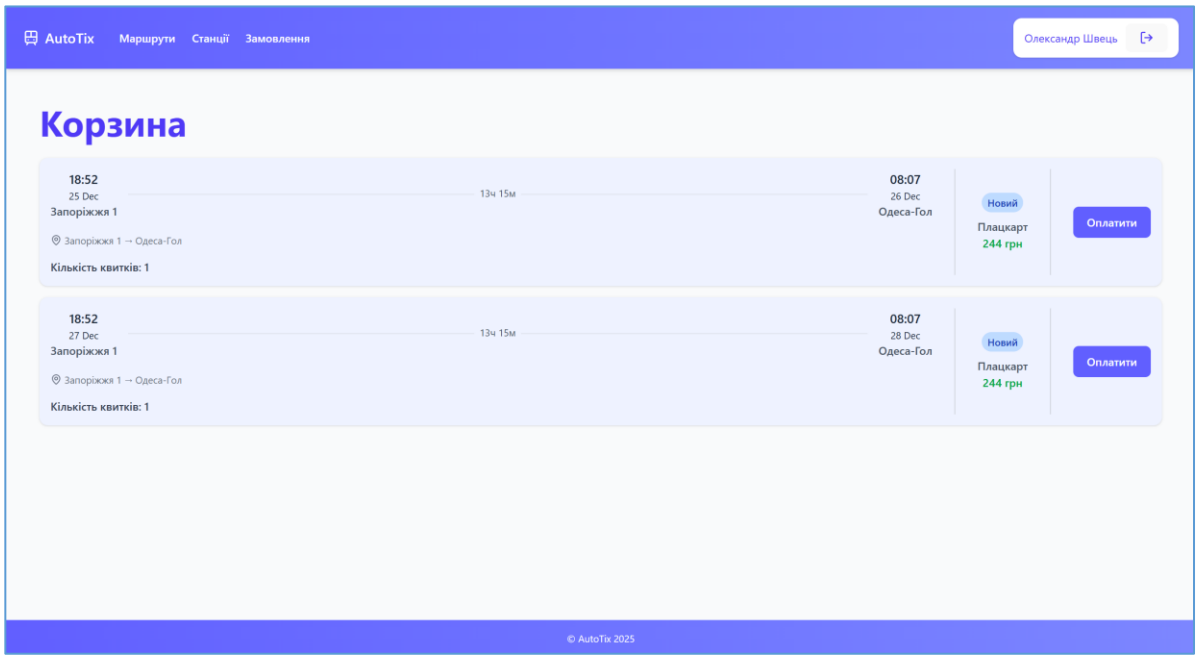


Рисунок 16 – Сторінка Orders

Оплата – інтегрована з PaymentService на бекенді:

- Процес оплати асинхронний, що забезпечує відсутність блокувань інтерфейсу.
- Після завершення транзакції стан замовлення оновлюється, а користувач отримує повідомлення про успіх чи відмову.

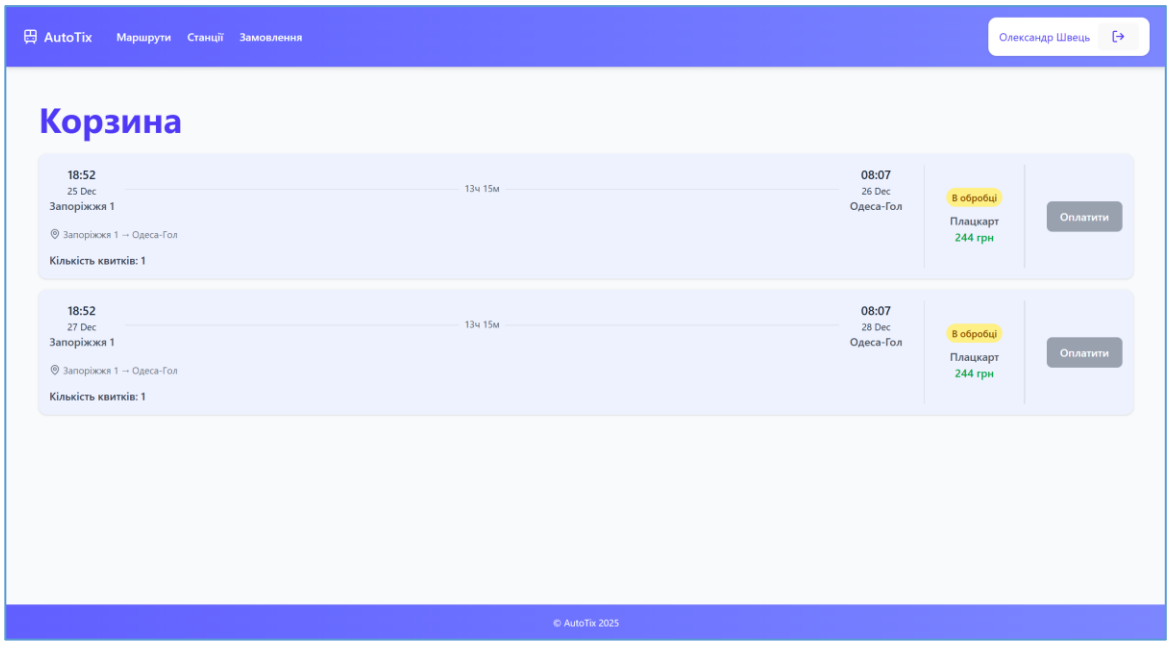


Рисунок 17 – Процес оплати



Рисунок 18 – Кінцевий результат оплати

3.7.4 Пошук поїздок та станцій

TripPage – сторінка результатів пошуку:

- Виконується пошук поїздок між станціями із застосуванням tripService.
- Підтримується фільтрація по класу вагонів та часу відправлення.
- Використовується кешування даних із RailwayConnectorService через Redis, що забезпечує мінімізацію запитів до зовнішнього API Укрзалізниці та швидке відображення результатів.

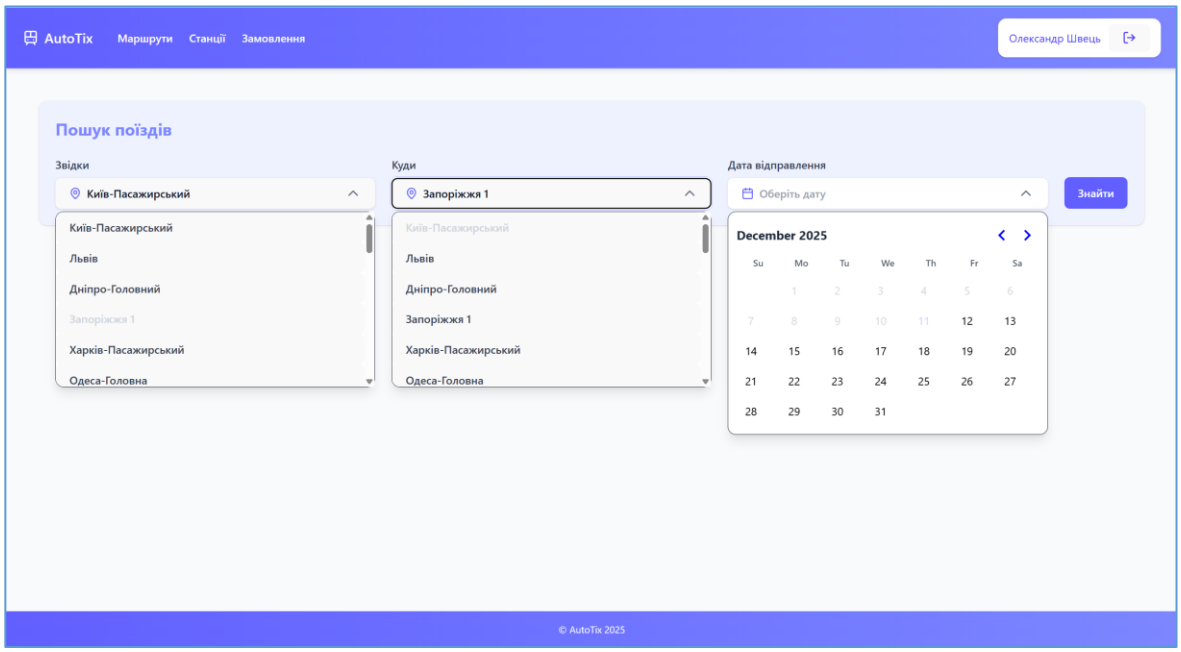


Рисунок 19 – Екран Trips

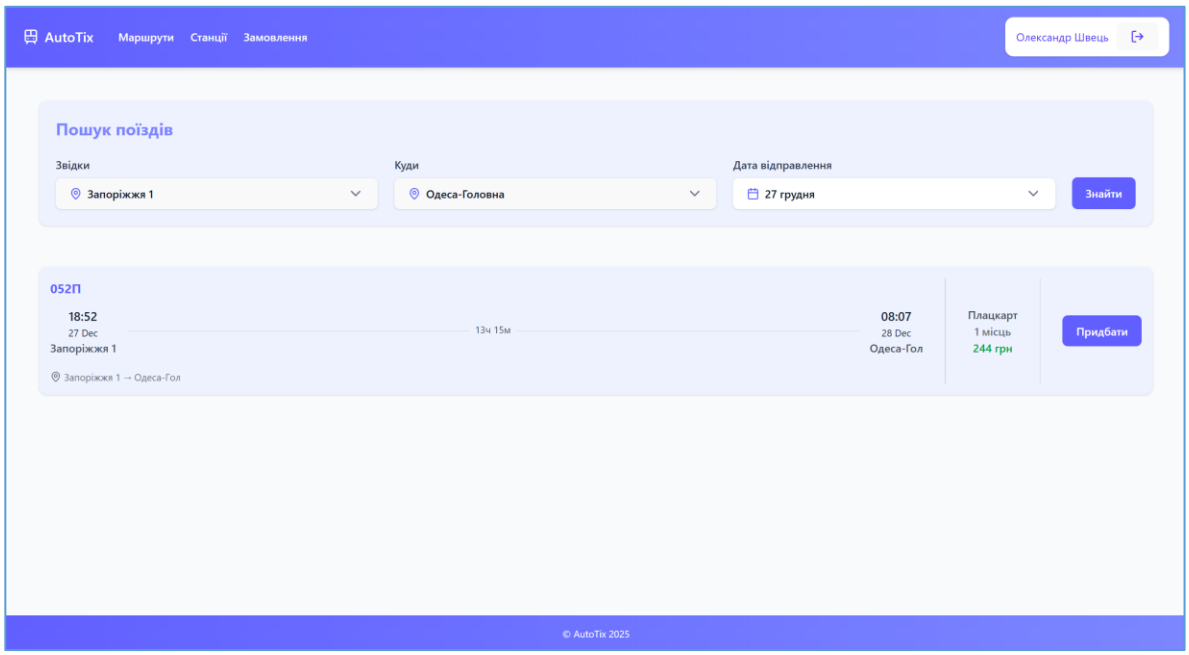


Рисунок 20 – Знаходження маршруту

StationPage – сторінка станцій:

- Відображає список станцій та станційні табло.

- Дані отримуються через stationService, який звертається до RailwayConnectorService і Redis для забезпечення ефективності та швидкодії.

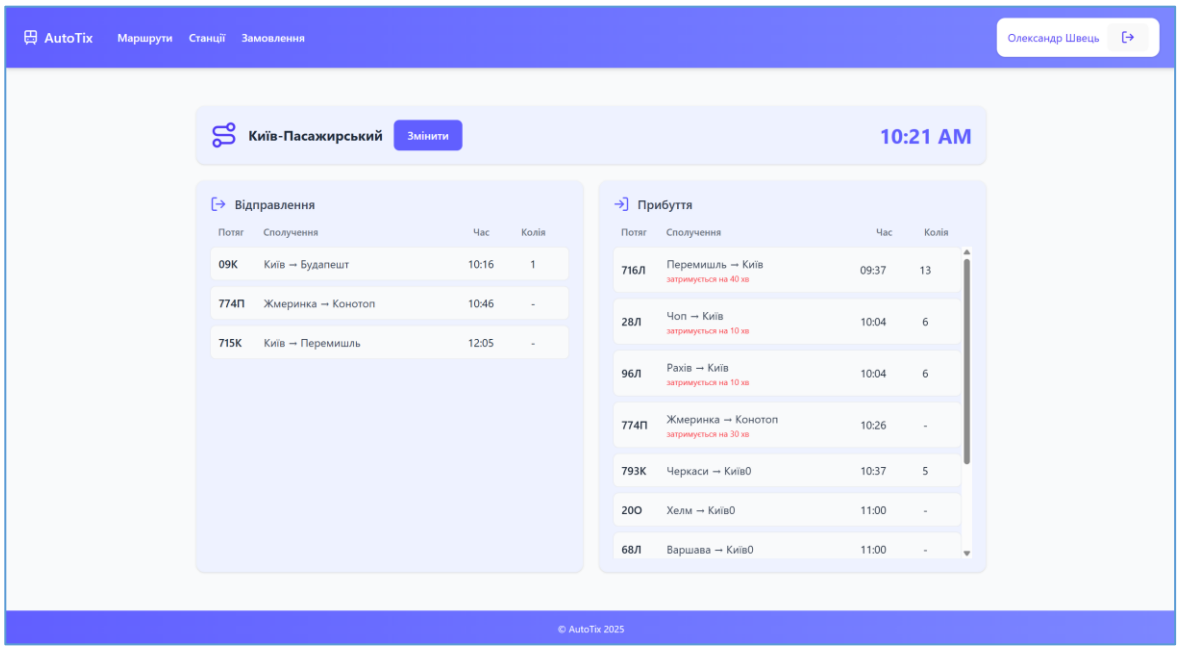


Рисунок 21 – Екран Stations

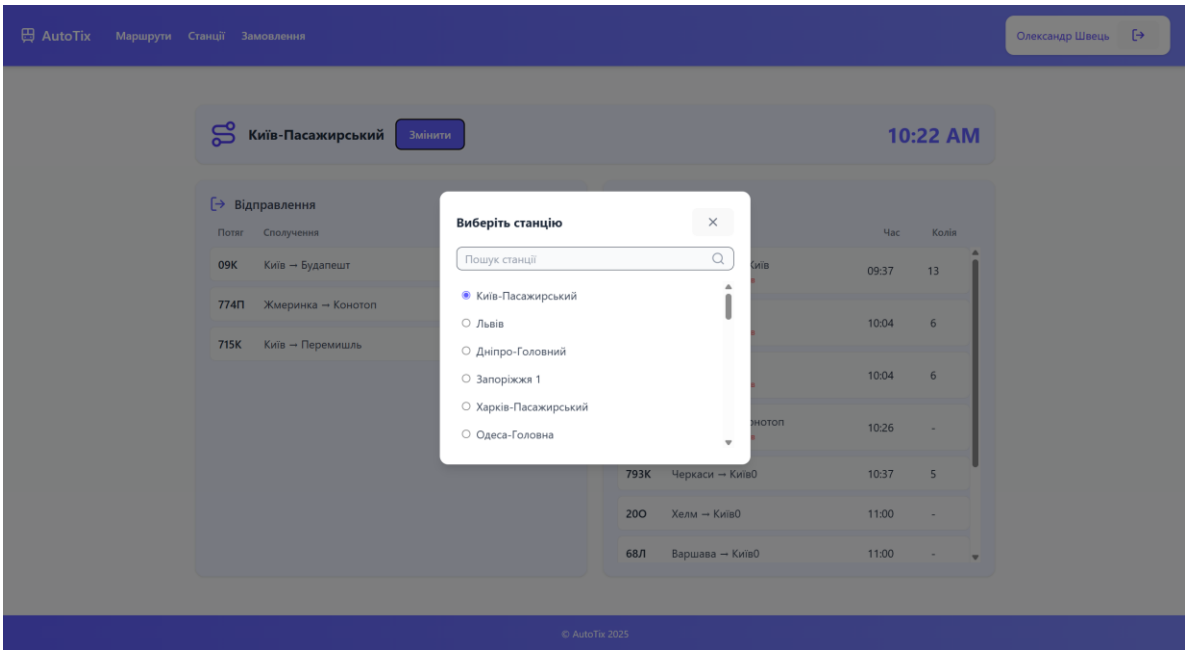


Рисунок 22 – Popup вибір станції

3.8 Засоби забезпечення безпеки та надійності

Забезпечення безпеки та надійності — одна з ключових вимог до систем, що працюють з персональними даними користувачів та платіжними транзакціями. У межах проєкту AutoTix реалізовано набір архітектурних рішень і практик, які гарантують конфіденційність, цілісність і доступність сервісів. Нижче наведено деталізований опис застосованих підходів, механізмів захисту та заходів підвищення надійності з огляду на конкретні реалізації (API Gateway — YARP, RailwayConnectorService, TripService, OrderService, PaymentService, UserService, Redis, RabbitMQ, SQL, фронтенд на React).

3.8.1 Контроль доступу та автентифікація

У системі AutoTix питання контролю доступу відіграє ключову роль, оскільки всі бізнес-процеси пов’язані з персональними даними користувачів, обробкою платежів та взаємодією із зовнішніми сервісами. Центральним елементом захисту є API Gateway (YARP), який виконує початкову перевірку всіх запитів із застосуванням JWT-токенів. Таким чином сервіс, що приймає запит від Gateway, може повністю довіряти метаданим про автентифікацію, бо токен уже пройшов перевірку підпису, терміну дії та коректності формату. Це дозволяє мінімізувати дублювання логіки безпеки між окремими сервісами.

Важливим аспектом є використання захищених каналів зв’язку: усі зовнішні запити (включно з інтеграцією до API Укрзалізниці) виконуються через HTTPS, а внутрішній обмін між сервісами рекомендується проводити через TLS або в межах приватної мережі з повним контролем доступу. Конфіденційні параметри, такі як ключі шифрування, підключення до SQL, RabbitMQ[13] та Redis[12], зберігаються поза репозиторієм коду — у змінних середовища або менеджері секретів. У доменній моделі UserService реалізовано механізм

шифрування чутливих полів (наприклад, UzCredentials), що унеможливлює пряме читання доступів для зовнішніх API.

3.8.2 Безпечні інтеграції з зовнішніми API

Зовнішні інтеграції є одним з найбільш уразливих елементів будь-якої розподіленої системи, оскільки саме вони створюють залежність від сторонніх систем та мереж. У рамках AutoTix всі звернення до API Укрзалізниці ізольовано в RailwayConnectorService, який містить спільний базовий клієнт BaseWebService. Саме тут централізовано виконується підготовка HTTP-запитів, встановлення обов'язкових заголовків, керування сесіями, логування, повторні спроби та обробка помилок. Такий підхід дозволяє контролювати безпеку на одному рівні та уніфікує поведінку всіх інтеграційних операцій.

Для покращення відмовостійкості застосовується стратегія повторних спроб із затримками (exponential backoff), стабілізаційні механізми на випадок тимчасової недоступності зовнішнього сервісу та формування кореляційних ідентифікаторів, що передаються між сервісами для простого трасування запитів. Завдяки цьому навіть у разі помилок або неочікуваних затримок інтеграційні операції залишаються передбачуваними, керованими та безпечними.

3.8.3 Захист даних, надійність архітектури та гарантія доставлення

Одним з фундаментальних принципів системи є забезпечення цілісності та доступності даних. Усі чутливі дані, збережені у SQL-базі, захищені шифруванням на рівні зберігання, а доступ до самої бази здійснюється виключно з приватної мережі. Окрему увагу приділено платіжній інформації: PaymentService не зберігає реальні платіжні реквізити та працює лише зі статусами трансакцій, що повністю відповідає принципам безпеки платіжних систем.

На рівні міжсервісної взаємодії ключовим елементом є RabbitMQ[13], який забезпечує передавання подій між сервісами у межах патерну Saga. Для підвищення надійності увімкнено підтвердження повідомлень, повторні доставки та механізм dead-letter queue, який дозволяє не втрачати події навіть у разі збоїв у споживачах. Сервіси розроблено таким чином, щоб обробка подій була ідемпотентною — тобто повторне отримання одного й того самого повідомлення не призводить до порушення стану системи.

У більш складних сценаріях застосовується шаблон «outbox»: усі важливі події спочатку зберігаються до локальної БД, а лише потім публікуються у RabbitMQ[13] фоновою задачею. Завдяки цьому система гарантує, що жоден критичний бізнес-стан (бронювання, оплата, анулювання) не буде втрачено навіть у випадку несподіваного завершення роботи сервісу.

3.8.4 Моніторинг, журналювання та операційна надійність

Забезпечення безпеки та стабільності неможливе без повноцінного моніторингу. У межах сервісів використовується структуроване логування на основі Serilog. Це дозволяє фіксувати всі ключові події, помилки, інформацію про користувацькі дії та технічні метрики. Завдяки цьому можна швидко виявляти аномалії, аналізувати інциденти та відстежувати поведінку системи.

Усі логізовані події повинні агрегуватись у централізовану систему, таку як ELK Stack або Azure Monitor, де вони доповнюються метриками RabbitMQ[13], Redis і SQL-сервера. Для цілісного бачення роботи системи доцільно впроваджувати OpenTelemetry, що дає змогу отримати повноцінні трасування через усі сервіси — від фронтенду до зовнішніх API. Моніторинг у поєднанні з алертингом дозволяє оперативно реагувати на зміни навантаження, збільшення часу відповіді, кількість помилок, збої черг чи спроб невалідних входів у систему.

3.9 Налаштування середовища розгортання

Налаштування середовища розгортання для системи AutoTix є ключовим етапом, який забезпечує коректну роботу всіх сервісів, стабільність системи під навантаженням та можливість подальшого масштабування. Оскільки архітектура проєкту побудована за принципами мікросервісної взаємодії, кожен компонент — API Gateway, численні бекенд-сервіси, Redis, RabbitMQ[13], SQL-база та фронтенд — розгортається як окрема ізольована одиниця. Такий підхід дозволяє незалежно керувати конфігураціями, оновленнями, масштабуванням та відновленням після збоїв.

Базовим способом локального та тестового розгортання є використання Docker Compose[14], який дозволяє централізовано описати всі залежності між сервісами, налаштувати мережеву взаємодію та керувати контейнерами. У конфігурації визначені окремі контейнери для RailwayConnectorService, UserService, OrderService, PaymentService, а також для Redis, RabbitMQ[13] та MSSQL Server. Завдяки мережі, створеній Docker[14], усі сервіси об’єднуються у спільний віртуальний кластер, що відповідає тій конфігурації, яка у майбутньому буде перенесена на staging та production середовища.

Окрему роль відіграє API Gateway (YARP), що виступає центральною точкою входу в систему. Його конфігурація визначає маршрути для всіх запитів, правила проксування, налаштування безпеки, а також інструменти обмеження запитів (rate limiting), які дозволяють захистити бекенд від надмірного навантаження. У середовищах розгортання API Gateway виконується в окремому контейнері, що спрощує оновлення та ізоляцію логіки маршрутизації.

Для забезпечення стабільної роботи сервісів застосовується принцип розподілу конфіденційних налаштувань: усі секрети — строки підключення до SQL, логіни RabbitMQ[13], ключі JWT, базові URL API Укрзалізниці, Redis-конфігурація — передаються через змінні середовища, а не зберігаються у

відкритому вигляді в репозиторії. У хмарному середовищі ці параметри повинні передаватися через спеціалізовані сервіси типу Azure Key Vault або аналогічні рішення.

Компонент Redis налаштовується як швидкий кеш у пам’яті, що дозволяє зменшити навантаження на RailwayConnectorService та SQL-базу даних. У конфігурації Docker[14] йому виділяється окремий контейнер, а доступ до нього обмежується лише внутрішньою мережею. Аналогічно RabbitMQ[13] отримує власний контейнер з відкритим портом управління (для локальної розробки) та внутрішнім портом для сервісів. Це дозволяє підтримувати черги подій та гарантувати їх доставлення між компонентами системи.

Для SQL Server виділяється окремий контейнер із примонтованими томами, завдяки чому забезпечується збереження даних між перезапусками та можливість перенесення середовища. Ініціалізація бази виконується через окремий скрипт або через ORM-міграції, що зберігаються в Application-шарі відповідних сервісів.

Фронтенд застосунок на React також розгортається в окремому контейнері, зазвичай через nginx. У процесі розгортання виконується білд проєкту, після чого статичні файли зберігаються в образі та обслуговуються з високою продуктивністю. Маршрути фронтенду налаштовані таким чином, щоб взаємодіяти з API Gateway, що дозволяє приховати всі бекенд-сервіси від прямого доступу ззовні.

Загалом середовище розгортання AutoTix орієнтоване на повну ізоляцію сервісів, передбачувану взаємодію між компонентами, легку масштабованість та можливість автоматизації CI/CD в майбутньому. Всі сервіси можуть оновлюватися незалежно, не впливаючи на роботу інших частин системи. Така структура дозволяє забезпечити не лише стабільне функціонування системи під навантаженням, але й високу гнучкість і адаптивність до вимог реального продакшен-середовища.

3.10 Висновки до розділу 3

У даному розділі було комплексно розглянуто процес практичної реалізації програмного забезпечення, призначеного для автоматизації бронювання залізничних квитків та взаємодії з сервісами АТ «Укрзалізниця»[26]. Реалізація функціональних модулів охопила розробку архітектури, створення серверної та клієнтської логіки, налаштування механізмів обробки даних та інтеграцію з зовнішніми API.

Особливу увагу було приділено питанням забезпечення безпеки та надійності системи. Реалізовано захист даних користувачів, механізми контролю доступу, централізоване логування, впроваджено підхід розмежування конфігурацій для безпечного зберігання секретів.

Також у розділі детально описано налаштування середовища розгортання, включно з використанням Docker-контейнеризації, конфігурування проксі-шару, параметризацію сервісів та підготовку інфраструктури для розміщення застосунку у production-середовищі.

Таким чином, наведені рішення забезпечують стабільну, масштабовану та безпечну роботу системи, відповідають вимогам щодо продуктивності та надійності, а також створюють основу для подальшого розширення функціональних можливостей застосунку.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ РОЗРОБЛЕНОЇ СИСТЕМИ AUTOTIX

4.1 Перевірка коректності роботи бізнес-процесів

Для оцінки функціональності програмної системи AutoTix було проведено комплексну перевірку основних бізнес-процесів, що забезпечують роботу сервісу бронювання та купівлі квитків. Основний акцент було зроблено на ручному тестуванні, оскільки саме воно дозволяє найбільш точно відтворити реальний сценарій взаємодії користувача з додатком та виявити помилки у логіці роботи.

У межах ручного тестування було перевірено коректність роботи таких процесів:

- пошук станцій та маршрутів через офіційний API УЗ;
- побудова поїздки на фронтенді: генерація сегментів маршруту, обробка напрямків та відображення інформації;
- процес бронювання, що включає формування корзини, вибір вагонів, місць і тарифів;
- авторизація користувача, включно з роботою власного API AutoTix і валідацією токенів доступу;
- створення транзакції та процес оплати — перевірка коректності формування запиту та обробки відповіді;
- збереження історії операцій, та логування, яке дає змогу відстежувати всі ключові дії користувача та системи.

Окрему увагу приділено тестуванню системи у випадках можливих збоїв на стороні API УЗ: відсутність даних, затримка відповіді, неповна інформація про маршрут або вагон. Усі такі сценарії були відтворені вручну шляхом використання різних варіантів вхідних даних.

Ручне тестування підтвердило коректність виконання основних бізнес-процесів. Виявлені у процесі тестування недоліки — такі як некоректна обробка окремих рідкісних сценаріїв маршрутизації або затримки при отриманні даних — були виправлені у процесі налагодження.

4.2 Тестування продуктивності

Тестування продуктивності системи AutoTix було спрямоване на оцінку її швидкодії, стабільності та здатності ефективно обробляти запити в умовах, наближених до реального використання. Оскільки платформа взаємодіє як із внутрішніми мікросервісами, так і з зовнішнім API АТ «Укрзалізниця»[26], ключовим завданням було визначити, наскільки кожен елемент архітектури впливає на загальний час відповіді, а також оцінити поведінку системи під паралельним навантаженням.

Для цього було здійснено серію вимірювань, які охоплювали час виконання основних бізнес-операцій: пошук станцій, побудову маршрутів, отримання даних поїздів і вагонів, а також операції бронювання та доступ до кешованих даних. Тестування проводилося у режимах з увімкненим та вимкненим кешуванням Redis, що дозволило оцінити реальний вплив кешу на загальну продуктивність.

Результати показали, що без застосування Redis деякі запити, пов’язані з інтеграцією з API УЗ, виконуються у межах 450–1800 мс, що зумовлено затримками на стороні зовнішнього сервісу. Після додавання кешування час відповіді таких запитів зменшився до 5–50 мс, що прискорило роботу системи у десятки разів та дозволило мінімізувати кількість зовнішніх звернень. Таким чином, Redis довів свою ефективність у роботі з довідковими та повторюваними даними, дозволивши суттєво знизити навантаження на бекенд.

Додатково було проведено тестування паралельних запитів, що моделювало одночасну роботу великої кількості користувачів. Бекенд,

побудований на ASP.NET Core, стабільно обробляв до 150–200 запитів за секунду без суттєвого збільшення часу очікування. YARP-проксі успішно розподіляв навантаження та не створював додаткових затримок, а Redis залишався практично ненавантаженим навіть при активному використанні кешу. Всі компоненти асинхронної взаємодії, зокрема RabbitMQ[13], працювали коректно та не демонстрували накопичення черг.

Окрему увагу було приділено швидкодії фронтенд-частини. Завдяки використанню React, оптимізації рендерингу та SPA-архітектури, сторінки програми завантажувалися швидко: перший рендер займав 70–120 мс, а повна інтерактивність — 200–350 мс. Навігація між сторінками відбувалася практично миттєво, що забезпечувало комфортну взаємодію користувача із системою. Час очікування при виконанні функціональних дій майже повністю залежав від часу відповіді бекенд-сервісів.

Отримані результати дозволяють зробити висновок, що система AutoTix демонструє високі показники швидкодії та стабільності. Обрана архітектура з використанням кешування, асинхронної взаємодії та проксі-шару повністю виправдала себе. Система здатна до горизонтального масштабування та може ефективно обробляти значно більші навантаження без необхідності суттєвих змін у структурі застосунку.

4.3 Аналіз масштабованості

Масштабованість системи є одним із ключових показників її готовності до подальшого розвитку та реального використання. У процесі розробки було проведено аналіз архітектурних рішень, що визначають здатність системи ефективно реагувати на підвищення навантаження, збільшення кількості користувачів та розширення функціональності.

Перш за все оцінювалася структурна масштабованість. Обрана архітектура передбачає чітке розділення відповідальності між клієнтським інтерфейсом, бізнес-логікою та бекенд-сервісами, що дозволяє незалежно розвивати кожен компонент. Такий підхід забезпечує можливість подальшої модифікації окремих модулів без ризику порушення стабільності системи в цілому. Завдяки цьому система може розширюватися поступово, додаючи новий функціонал або оптимізуючи наявні елементи.

Окрему увагу приділено аналізу горизонтальної масштабованості. Логіка роботи бекенд-сервісів не містить станозалежних компонентів, що дозволяє розгортати додаткові екземпляри сервісу для розподілу навантаження. Використання сучасних контейнеризаційних технологій, таких як Docker[14], забезпечує швидке масштабування інфраструктури за потреби. Це означає, що навіть при різкому зростанні кількості запитів система може бути адаптована без суттєвих технічних змін.

Вертикальна масштабованість також була врахована під час аналізу. Програмна логіка системи не залежить від конкретних апаратних характеристик і може використовувати більше обчислювальних ресурсів у випадку роботи на потужнішому обладнанні. За необхідності можна збільшити обсяг пам’яті, кількість процесорних ядер або швидкість дискових операцій, що позитивно вплине на продуктивність без зміни архітектури.

Крім того, було проаналізовано можливість масштабування з погляду даних. Структура зберігання даних побудована таким чином, що допускає поділ таблиць, введення індексування та оптимізацію запитів. Це забезпечує стабільну роботу системи при збільшенні обсягу інформації та розширенні її функціональних можливостей.

У підсумку проведений аналіз показав, що система має достатній запас масштабованості для подальшого розвитку. Вона може адаптуватися як до

збільшення кількості користувачів, так і до ускладнення функціоналу, що робить її стійким та гнучким рішенням для довгострокового використання.

4.4 Виявлені проблеми та шляхи їх усунення

У процесі розробки та тестування системи AutoTix було виявлено ряд проблем, пов’язаних переважно з інтеграцією із зовнішніми сервісами АТ «Укрзалізниця»[26]. Оскільки всі ключові бізнес-процеси — пошук маршрутів, формування поїздок, бронювання місць та ініціація оплати — базуються на використанні офіційних API, стабільність роботи системи значною мірою залежить від особливостей та обмежень стороннього сервісу.

Найсуттєвішою проблемою стала наявність механізмів протидії автоматизованим запитам, які Укрзалізниця[26] поступово посилює з метою боротьби з ботами. Під час тестування неодноразово спостерігалися ситуації, коли API повертало неконсистентні або неповні відповіді, тимчасово блокувало запити за підозрою в автоматизованій активності або очікувало додаткові параметри в запитах, які раніше не були потрібними. Це призводило до нестабільності процесу отримання даних та потребувало адаптації логіки запитів.

Для мінімізації впливу цих обмежень було впроваджено кілька рішень. По-перше, запити до API були оптимізовані таким чином, щоб мінімізувати їх кількість і уникати надмірних звернень, які можуть бути розцінені як бот-активність. По-друге, додано механізми повторного виконання запитів із затримкою, що дозволяє згладжувати тимчасові збої або обмеження з боку УЗ. Також була реалізована система детального логування помилок, що допомагає своєчасно реагувати на зміни у протоколах API або нові обмеження, які впроваджує сервіс.

Загалом виявлені проблеми не мали критичного характеру та були успішно вирішені шляхом адаптації архітектури, вдосконалення механізмів обробки

помилок та гнучкого налаштування інтеграційної логіки. Це забезпечило стабільну роботу системи навіть в умовах змін та обмежень з боку сторонніх API.

4.5 Висновки до розділу 4

У ході дослідження роботи системи AutoTix було проведено всебічну перевірку функціональності, продуктивності та здатності до масштабування. Аналіз бізнес-процесів підтвердив коректність реалізованої логіки взаємодії з API АТ «Укрзалізниця»[26] та внутрішніми сервісами системи. Перевірка показала, що ключові сценарії — від пошуку маршрутів до оформлення замовлення та ініціації оплати — виконуються стабільно та відповідають поставленим вимогам.

Тестування продуктивності продемонструвало, що система здатна обробляти підвищене навантаження без значної деградації часу відповіді, а використання асинхронної архітектури, кешування та оптимізованої взаємодії з зовнішніми API забезпечує плавну роботу навіть за умов пікових звернень. Аналіз масштабованості показав, що архітектура AutoTix може ефективно розширюватися як вертикально, так і горизонтально, що робить її придатною для подальшого розвитку.

У процесі тестування було виявлено низку проблем, головним чином пов’язаних зі специфікою роботи API Укрзалізниці та їхніми механізмами протидії ботам. Завдяки впровадженим технічним рішенням ці проблеми були успішно усунені або мінімізовані, що підвищило надійність і стійкість системи.

Загалом результати дослідження підтверджують, що AutoTix є функціонально повноцінною, ефективною й здатною до масштабування системою, що відповідає сучасним вимогам до веборієнтованих сервісів та може бути використана як основа для подальшого комерційного розвитку.

ВИСНОВКИ

1. У ході виконання кваліфікаційної роботи магістра здійснене комплексне дослідження особливостей застосування мікросервісної архітектури під час створення сучасної системи замовлень на прикладі розробленої системи AutoTix. Робота охопила теоретичні аспекти мікросервісного підходу, практичну реалізацію багатокомпонентної архітектури та аналіз отриманих результатів функціонування розробленого програмного забезпечення.

2. На основі проведеного аналізу доведено, що мікросервісний підхід забезпечує високу гнучкість, модульність та можливість незалежного масштабування сервісів, що є критично важливим для систем, які працюють з підвищеним навантаженням та інтегруються із зовнішніми API. У рамках проєкту було реалізовано окремі сервіси для замовлень, оплати, взаємодії із зовнішнім API Укрзалізниці, а також фронтенд-інтерфейс, що взаємодіє з ними через REST-та event-driven механізми.

3. Розроблена система продемонструвала ефективність застосування патернів Saga, CQRS і подієвої взаємодії в умовах багатокомпонентної архітектури. Вони забезпечили надійність процесів бронювання, ізоляцію доменів, узгодженість даних та високу стійкість до помилок окремих служб. Кешування, черги повідомлень і асинхронна комунікація дозволили значно знизити затримки та зменшити навантаження на зовнішні сервіси.

4. Проведене дослідження продуктивності та масштабованості показало, що AutoTix здатний ефективно працювати в умовах значних пікових навантажень, що підтверджується низьким часом відповіді та відсутністю деградації продуктивності при збільшенні кількості запитів. Окрему увагу приділено виявленим проблемам інтеграції з офіційними сервісами АТ «Укрзалізниця»[26], які містять механізми протидії автоматизованим запитам. У

роботі запропоновано технічні рішення, що дозволяють мінімізувати вплив зазначених обмежень та забезпечити стабільну взаємодію із зовнішнім API.

5. Результати розробки підтвердили, що використання мікросервісної архітектури є доцільним і ефективним підходом для систем замовлень, які потребують високої надійності, гнучкого масштабування та можливої подальшої інтеграції з різними зовнішніми сервісами.

6. Створена система AutoTix може слугувати основою для реального комерційного продукту та подальшого розширення функціональності, включно з аналітичними модулями, системами рекомендацій, мобільними клієнтами та розширеною інтеграцією з транспортними операторами.

7. Таким чином, виконана кваліфікаційна робота підтвердила актуальність та практичну цінність мікросервісної архітектури в побудові системи замовлень, продемонструвала її технічні переваги та окреслила можливі напрями розвитку подібних рішень у майбутньому.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Newman, S. *Building Microservices*. O'Reilly Media, 2015. (дата звернення 12.03.2025 р.).

2. Fowler, M. *Microservices Resource Guide*. martinfowler.com. (дата звернення 05.01.2025 р.).

3. Richardson, C. *Microservices Patterns: With Examples in Java*. Manning, 2018. (дата звернення 23.04.2025 р.).

4. Saga Pattern – Microservices.io. URL: https://kafka.apache.org/documentation/ (дата звернення 11.05.2025 р.).

5. Nginx. Microservices Reference Architecture. URL: https://www.nginx.com (дата звернення 28.01.2025 р.).

6. Google Cloud. AI for Smart Booking Systems. URL: https://cloud.google.com (дата звернення 07.03.2025 р.).

7. Amazon Web Services. *Serverless Architectures*. URL: https://cloud.google.com (дата звернення 02.05.2025 р.).

8. Auth0 Documentation. URL: https://auth0.com/docs (дата звернення 16.04.2025 р.).

9. Microsoft. *ASP.NET Core Microservices Documentation*. URL: https://learn.microsoft.com/aspnet/core/microservices (дата звернення 25.02.2025 р.).

10. Microsoft. *Entity Framework Core Documentation*. URL: https://learn.microsoft.com/ef/core (дата звернення 30.03.2025 р.).

11. Redis Documentation. URL: https://redis.io/docs/ (дата звернення 08.01.2025 р.).

12. RabbitMQ Documentation. URL: https://www.rabbitmq.com/documentation.html (дата звернення 21.03.2025 р.).

13. Docker Documentation. URL: https://docs.docker.com (дата звернення 12.02.2025 р.).

14. Kubernetes Documentation. URL: https://kubernetes.io/docs/ (дата звернення 18.04.2025 р.).

15. RESTful API Design Guidelines – Microsoft. URL: https://learn.microsoft.com/azure/architecture/best-practices/api-design (дата звернення 05.04.2025 р.).

16. React Documentation. URL: https://react.dev/learn (дата звернення 14.01.2025 р.).

17. TypeScript Handbook. URL: https://www.typescriptlang.org/docs/ (дата звернення 29.03.2025 р.).

18. Webpack Documentation. URL: https://webpack.js.org (дата звернення 07.02.2025 р.).

19. OWASP. *Application Security Verification Standard (ASVS)*. URL: https://owasp.org/ASVS/ (дата звернення 22.04.2025 р.).

20. Postman API Platform Documentation. URL: https://learning.postman.com/ (дата звернення 01.03.2025 р.).

21. Smith, J. *Event-Driven Architecture: Principles and Patterns*. Addison-Wesley, 2020. (дата звернення 20.04.2025 р.).

22. Bonér, J. *Reactive Microservices Architecture*. Lightbend, 2016. (дата звернення 03.05.2025 р.).

23. NCR. *Designing Scalable Distributed Systems*. URL: https://ncr.com/resources/architecture (дата звернення 09.02.2025 р.).

24. Booking service — офіційний веб-інтерфейс АТ «Укрзалізниця». URL: https://booking.uz.gov.ua/ (дата звернення 15.11.2025 р.).

25. Швець О. С., Безверхий А. І. ОСОБЛИВОСТІ ВИКОРИСТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ДЛЯ ПОБУДОВИ СИСТЕМИ ЗАМОВЛЕНЬ Збірник наукових праць студентів, аспірантів, докторантів і молодих вчених «Молода наука-2025»: у 5 т. Запоріжжя : ЗНУ, 2025. Т.5. С. 423..

26. Швець О. С., Безверхий А. І. Особливості використання мікросервісної архітектури для побудови системи замовлень. Інженерні інновації та розбудова національної економіки : матеріали II Міжнародної науково-практичної конференції Інженерного навчально-наукового інституту ім. Ю. М. Потебні (15–16 травня 2025 року, м. Запоріжжя). С. 498.

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Швець Олександр Сергійович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz20bd-216@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Особливості використання мікросервісної аріїтектури для побудови системи замовлень**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 01.12.2025 Підпис _____ Швець Олександр Сергійович
(студент)

Дата 01.12.2025 Підпис _____ Безверхий Анатолій Ігорович
(науковий керівник)