НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

До захисту допущено:

Завідувач кафедри

__________ Олександр РОЛІК

«___»_____________2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інформаційні управляючі системи та технології»
спеціальності 126 «Інформаційні системи та технології»
на тему: «Система управління замовленнями в електронній комерції на базі мікросервісної архітектури»

Виконав:
студент IV курсу, групи ІС-13
Хрущ Андрій Володимирович ___________

Керівник:
доцент кафедри ІСТ, к.ф.-м.н.
Рибачук Людмила Віталіївна ___________

Рецензент:
доцент кафедри ІПІ, к.т.н.
Олійник Юрій Олександрович ___________

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент ______________

Київ – 2025 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_______ Олександр РОЛІК

«___»____________2025 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Хрущу Андрію Володимировичу

1. Тема проєкту «Система управління замовленнями в електронній комерції на базі мікросервісної архітектури», керівник проєкту Рибачук Людмила Віталіївна, к.ф.-м.н., доцент, затверджені наказом по університету від «23» травня 2025р. №1705-с

2. Термін подання студентом проєкту: 09.05.2025.

3. Вихідні дані до проєкту: мікросервісна архітектура, сервіси REST API, база даних PostgreSQL, фреймворк Spring Boot.

4. Зміст пояснювальної записки: опис предметної області, аналіз і обґрунтування засобів вирішення поставленої задачі, програмна реалізація системи, математичне забезпечення та тестування системи.

5. Перелік графічного матеріалу (із зазначенням обов’язкових креслеників, плакатів, презентацій тощо): схема архітектури системи, взаємодії баз даних та DTO, моделювання процесу створення замовлення та діаграма прецедентів системи

6. Дата видачі завдання: 08.03.25.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Аналіз предметної області	13.03.25 – 19.03.25	
2	Аналіз існуючих рішень	20.03.25 – 27.03.25	
3	Формування вимог до системи мікросервісів	28.03.25 – 03.04.25	
4	Огляд структур існуючих систем для розробки	04.04.25 – 11.04.25	
5	Розроблення програмного продукту	12.04.25 – 12.05.25	
6	Розроблення алгоритмів та стратегій оптимізації	13.05.25 – 20.05.25	
7	Тестування системи	21.05.25 – 28.05.25	
8	Оформлення текстової документації	29.05.25 – 09.06.25	

Студент Андрій ХРУЩ

Керівник Людмила РИБАЧУК

АНОТАЦІЯ

Система управління замовленнями в електронній комерції на базі мікросервісної архітектури.

Проєкт містить 73 с. тексту, 30 рисунків, 11 таблиць, посилання на 16 літературних джерел, 2 додатки та 4 конструкторських документа.

KEYCLOAK, POSTGRESQL, RABBITMQ, SPRING BOOT, ЕЛЕКТРОННА КОМЕРЦІЯ, МІКРОСЕРВІСНА АРХІТЕКТУРА, СИСТЕМА УПРАВЛІННЯ ЗАМОВЛЕННЯМИ.

Об’єктом розробки є система управління замовленнями в електронній комерції для моделі B2C.

Мета розробки — створення масштабованого та гнучкого програмного рішення для автоматизації обробки замовлень, управління каталогом, складськими запасами, доставкою та підтримкою клієнтів.

У дипломному проєкті розроблено прототип системи, що складається з набору мікросервісів: Auth Service, User Service, Catalog Service, Order Service, Inventory Service, Delivery Service, реалізованих за допомогою Spring Boot. Виконано аналіз бізнес-процесів електронної комерції, сформовано функціональні та нефункціональні вимоги, обґрунтовано вибір технологічного стеку (Spring Boot, PostgreSQL, RabbitMQ, Keycloak) для забезпечення безпеки, масштабованості, синхронної та асинхронної взаємодії. Проведено тестування прототипу на відповідність вимогам продуктивності під час пікових навантажень.

Отримані результати можуть бути застосовані для автоматизації бізнес-процесів у малих і середніх платформах електронної комерції, сприяючи підвищенню ефективності управління замовленнями та зниженню операційних витрат.

SUMMARY

Order Management System in Electronic Commerce Based on Microservice Architecture.

The project contains 73 pages of text, 30 figures, 11 tables, references to 16 literary sources, 2 annexes and 4 design documents.

Keywords: E-COMMERCE, KEYCLOAK, MICROSERVICE ARCHITECTURE, ORDER MANAGEMENT SYSTEM, POSTGRESQL, RABBITMQ, SPRING BOOT.

The object of development is an e-commerce order management system for the B2C model.

The goal of the development is to create a scalable and flexible software solution for automating order processing, catalog management, warehouse inventory, delivery, and customer support.

The thesis project developed a prototype system consisting of a set of microservices: Auth Service, User Service, Catalog Service, Order Service, Inventory Service, Delivery Service, implemented using Spring Boot. The analysis of e-commerce business processes is carried out, functional and non-functional requirements are formed, the choice of technology stack (Spring Boot, PostgreSQL, RabbitMQ, Keycloak) is substantiated to ensure security, scalability, synchronous and asynchronous interaction. The prototype was tested for compliance with performance requirements during peak loads.

The obtained results can be used to automate business processes in small and medium-sized e-commerce platforms, contributing to the efficiency of order management and reducing operating costs.

Номер рядка	Формат	Позначення	Найменування	Кільк. аркушів	Номер екзем.	Примітка
1			<u>Документація загальна</u>			
2						
3			Знову розроблена			
4						
5	А4	ІС13.290БАК.005 ПЗ	Пояснювальна записка	73		
6						
7	А3	ІС13.290БАК.005 Д1	Система управління замовлен-	1		
8			нями в електронній комерції на			
9			базі мікросервісної архітектури.			
10			Схема архітектури системи			
11						
12	А3	ІС13.290БАК.005 Д2	Система управління замовлен-	1		
13			нями в електронній комерції на			
14			базі мікросервісної архітек-			
15			тури. Схема моделювання про-			
16			цесу створення замовлення			
17						
18	А3	ІС13.290БАК.005 Д3	Система управління замовлен-	1		
19			нями в електронній комерції на			
20			базі мікросервісної архітектури.			
21			Діаграма прецедентів системи			
22						
23	А3	ІС13.290БАК.005 Д4	Система управління замовлен-	1		
24			нями в електронній комерції на			
25			базі мікросервісної архітектури.			
26			Схема взаємодії баз даних та			
27			DTO			
28						

Зм.	Лист	№ докум.	Підпис	Дата	ІС13.290БАК.005 ТП		
Розробив		Хрущ			Система управління замовленнями в електронній комерції на базі мікросервісної архітектури. Відомість дипломного проєкту	Літ.	Аркуш	Аркушів
Перевірив		Рибачук				Т	1	1
						КПІ ім. Ігоря Сікорського Група ІС-13		
Затв.								

Пояснювальна записка
до дипломного проєкту
на тему: «Система управління
замовленнями в електронній
комерції на базі мікросервісної
архітектури»

Київ - 2025 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 3
ВСТУП 5
1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ 7
1.1 Сутність електронної комерції та її ключові процеси 7
1.2 Бізнес-процеси управління замовленнями в e-commerce 8
1.3 Роль інформаційних систем у підтримці управлінських рішень 10
Висновки до розділу 1 12
2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ.. 13
2.1 Порівняння монолітних і мікросервісних архітектур 13
2.2 Огляд технологій для реалізації мікросервісів 14
2.3 Приклади мікросервісних систем у провідних e-commerce 16
Висновки до розділу 2 19
3 ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ 21
3.1 Функціональні вимоги до системи управління замовленнями 21
3.2 Нефункціональні вимоги до системи управління замовленнями 22
3.3 Критерії оцінки мікросервісної архітектури системи 24
Висновки до розділу 3 25
4 ВИБІР ТЕХНОЛОГІЙ РОЗРОБКИ 26
4.1 Аналіз мов програмування та фреймворків 26
4.2 Огляд інструментів для синхронної та асинхронної взаємодії 28
4.3 Вибір інструментів контейнеризації, бази даних та безпеки 31
Висновки до розділу 4 33
5 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ.. 34
5.1 Проєктування архітектури системи 34
5.1.1 Розподіл системи на мікросервіси... 34
5.1.2 Організація взаємодії між мікросервісами 36

					ІС13.290БАК.005 ПЗ			
Зм.	Лист	№ докум.	Підпис					
Розробив		Хрущ А.В.			Система управління замовленнями в електронній комерції на базі мікросервісної архітектури. Пояснювальна записка	Літ.	Арк.	Аркушів
Перевірив		Рибачук Л.В.				Т	2	74
						КПІ ім. Ігоря Сікорського Група ІС-13		
Затв.								

5.1.3 Архітектура баз даних 41
5.2 Реалізація прототипу системи 42
5.2.1 Використання Java Spring і Docker для контейнеризації 42
5.2.2 Розробка внутрішньої структури мікросервісу 45
Висновки до розділу 5 48
6 МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 50
6.1 Постановка задачі 50
6.1.1 Змістовна постановка задачі 50
6.1.2 Математична постановка задачі 51
6.2 Обґрунтування методу розв'язання 52
6.3 Опис методу розв'язання 53
Висновки до розділу 6 62
7 ТЕСТУВАННЯ СИСТЕМИ 63
7.1 Мета та планування тестування 63
7.2 Тестування навантаженням 64
7.3 Тестування обробки некоректних запитів 66
Висновки до розділу 7 69
ВИСНОВКИ 70
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 72

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (інтерфейс програмування додатків, набір правил і засобів для взаємодії між програмними компонентами).

DB – Database (база даних, організоване сховище даних для зберігання та обробки інформації в системі).

Docker – платформа для контейнеризації, яка дозволяє упаковувати додатки разом із їхніми залежностями в ізольовані контейнери.

DTO – Data Transfer Object (об'єкт, який передає інформацію між шарами програми, полегшуючи комунікацію з-поміж них).

E-commerce – Electronic Commerce (електронна комерція, діяльність із купівлі-продажу товарів і послуг через Інтернет).

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту, основа для обміну даними в Інтернеті між клієнтом і сервером).

JPA – Java Persistence API (інтерфейс для роботи з базами даних у Java, який спрощує збереження об’єктів у реляційних базах).

JSON – JavaScript Object Notation (нотація об’єктів JavaScript, легкий формат обміну даними, який використовується для передачі структурованої інформації).

RabbitMQ – брокер повідомлень із відкритим кодом, який забезпечує асинхронну взаємодію між компонентами системи через черги.

REST – Representational State Transfer (передача стану представлення, архітектурний стиль для створення веб-сервісів із використанням HTTP-запитів).

Spring Boot – фреймворк для створення додатків на мові Java, який спрощує конфігурацію та розгортання програм.

Spring Cloud – набір інструментів для побудови розподілених систем на базі Spring, включає підтримку мікросервісів і їхньої координації.

ІС – інформаційні системи (комплекси програмного забезпечення, які забезпечують збір, обробку, зберігання та аналіз даних для підтримки бізнес-процесів і прийняття рішень).

ВСТУП

Сучасний світ електронної комерції переживає стрімкий розвиток обумовлений глобалізацією ринків, зростанням кількості онлайн користувачів та підвищенням очікувань щодо швидкості й якості обслуговування. Обсяг світового ринку e-commerce у 2024 році перевищив 6 трильйонів доларів США, і прогнозується його подальше зростання на 9–15% щорічно, за даними аналітичних звітів у роботі [1]. Це змушує компанії адаптувати свої інформаційні системи до високих навантажень, частих змін у бізнес-процесах і необхідності швидкого впровадження нових функцій. Монолітні архітектури тривалий час мали великий попит в розробці програмного забезпечення, але наразі вони частіше виявляються недостатньо гнучкими для задоволення цих потреб. Натомість мікросервісна архітектура пропонує модульний підхід, який дозволяє розподіляти функціональність системи на окремі незалежні компоненти, кожен із яких відповідає за конкретну задачу. Це забезпечує не лише високу масштабованість, а й можливість паралельної розробки та швидкого розгортання оновлень.

Особливо актуальною мікросервісна архітектура є для систем управління замовленнями в електронній комерції. Такі системи є центральним елементом будь-якої e-commerce платформи, адже від їхньої ефективності залежить швидкість обробки замовлень, точність управління складськими запасами та взаємодія з користувачами. У високонавантажених системах, таких як Amazon, Alibaba чи eBay, затримка в обробці замовлення навіть на кілька секунд може призвести до втрати клієнтів і значних фінансових збитків. Мікросервіси дозволяють оптимізувати ці процеси, розподіляючи навантаження між різними сервісами, такими як сервіс каталогів, користувачів чи платежів, забезпечуючи їхню автономну роботу. Більш того, використання сучасних технологій спрощує реалізацію мікросервісів, надаючи готові інструменти для управління конфігурацією, обміну повідомленнями та моніторингу. Розробка систем управління замовленнями на базі мікросервісної архітектури є актуальною задачею, яка відповідає сучасним викликам електронної комерції та сприяє підвищенню конкурентоспроможності бізнесу.

Метою дипломного проєкту є розробка системи управління замовленнями в електронній комерції на базі мікросервісної архітектури, яка забезпечить підтримку управлінських рішень шляхом підвищення ефективності обробки замовлень, масштабованості та адаптивності до змін бізнес вимог. Запропонована система має стати інструментом, який дозволить менеджерам оперативно реагувати на ринкові зміни, оптимізувати розподіл ресурсів і підвищувати якість обслуговування клієнтів, як згадується у статті [2]. У процесі досягнення цієї мети передбачається вирішення низки завдань, які охоплюють як теоретичні, так і практичні аспекти розробки. Основні завдання дослідження включають:

– провести аналіз вимог до системи управління замовленнями в умовах високого навантаження, враховуючи специфіку бізнес-процесів електронної комерції;

– розробити архітектуру системи на основі мікросервісів, визначивши ключові компоненти та принципи їхньої взаємодії;

– реалізувати прототип системи з використанням сучасних інструментів розробки, обраних на основі аналізу вимог;

– забезпечити продуктивність системи шляхом реалізації механізмів розподілу навантаження та оцінки її роботи під різними сценаріями;

– оцінити ефективність розробленої системи за визначеними критеріями, проаналізувавши її вплив на управлінські процеси та визначивши перспективи подальшого розвитку.

У результаті виконання проєкту очікується створення прототипу системи, який продемонструє переваги мікросервісної архітектури в контексті управління замовленнями та пройде оцінку на відповідність функціональним і нефункціональним вимогам. Такий підхід не лише сприятиме вирішенню практичних задач e-commerce, а й стане основою для подальших досліджень у сфері оптимізації високонавантажених систем.

1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Сутність електронної комерції та її ключові процеси

Електронна торгівля є однією з найдинамічніших галузей сучасної економіки, яка охоплює купівлю, продаж, обмін товарами, послугами та інформацією через Інтернет. За даними аналітичної компанії Statista, очікуване зростання світового ринку e-commerce на 2027 рік становить 8 трильйонів доларів США зі середньорічним темпом зростання (CAGR) 9,5%, що ґрунтується на звіті роботи [3]. Такий швидкий розвиток обумовлений декількома чинниками: зростанням кількості користувачів Інтернету (за даними Statista, у 2024 році їх число перевищило 5 мільярд осіб), доступністю смартфонів та зміною споживацької звички, коли споживачі надають перевагу онлайн шопінгу через його зручність, швидкість та можливість для порівняння цін.

Електронна комерція поділяється на кілька основних моделей залежно від учасників транзакцій:

– бізнес продає товари чи послуги кінцевим споживачам (b2c, наприклад, amazon, rozetka);

– бізнес продає товари чи послуги іншому бізнесу (b2b, наприклад, alibaba);

– споживачі продають товари один одному через платформи (c2c, наприклад, ebay, olx);

– споживачі пропонують продукти чи послуги бізнесу (c2b, наприклад, фріланс-платформи, такі як upwork).

У контексті теми дипломного проєкту основна увага приділяється моделі B2C, оскільки система управління замовленнями орієнтована на кінцевих споживачів, які купують товари через онлайн-магазин.

Процеси електронної комерції в моделі B2C представлені в таблиці 1.1. Дані процеси взаємопов’язані та потребують автоматизації для забезпечення швидкості, точності та конкурентоспроможності. Наприклад, 65% покупців відмовляються від повторних покупок, якщо час доставки перевищує 3 дні, згідно зі статтею у роботі [4]. Це твердження підкреслює важливість ефективного управління логістикою.

Таблиця 1.1 – Ключові процеси електронної комерції в моделі B2C

Процес	Опис
Аутентифікація та авторизація	Перевірка облікових даних клієнтів, видача токенів доступу та забезпечення безпеки доступу
Управління користувачами	Реєстрація, управління профілями та персональними даними клієнтів
Управління каталогом товарів	Формування, оновлення, категоризація, фільтрація та відображення асортименту товарів для клієнтів
Обробка замовлень	Створення замовлення, вибір товарів, підтвердження та відстеження статусу замовлення
Управління складом	Контроль наявності товарів, резервування та оновлення запасів
Координація доставки	Організація та відстеження статусу доставки

1.2 Бізнес-процеси управління замовленнями в e-commerce

Управління замовленнями в електронній комерції є центральним бізнес-процесом, що відповідає за узгодження кожного етапу – від моменту вибору товару до фінальної підготовки, включаючи взаємодію з клієнтами та ресурсами компанії. У моделі B2C цей процес відіграє ключову роль, визначаючи рівень задоволення кінцевих споживачів та конкурентоспроможність онлайн-платформи. Головна задача управління замовленнями полягає в оптимізації часу обробки, точності виконання та інтеграції з іншими бізнес-процесами, такими як управління каталогом та складом.

Бізнес-процес управління замовленнями в e-commerce можна розбити на кілька етапів, кожен із яких має свої особливості та взаємозв’язки. Ці етапи представлені в таблиці 1.2.

Таблиця 1.2 – Основні етапи бізнес-процесу управління замовленнями

Етап	Опис
Ініціація замовлення	Клієнт обирає товари з каталогу, додає їх до кошика та підтверджує намір покупки
Перевірка доступності	Система перевіряє наявність обраних товарів на складі та їхній статус (доступні чи під замовлення)
Оформлення замовлення	Клієнт вводить дані для доставки, а система генерує унікальний ідентифікатор замовлення
Виконання замовлення	Резервування товарів на складі та оновлення статусу замовлення для підготовки до подальших дій
Повернення та підтримка	Обробка скарг, повернень товарів та надання консультацій клієнтам після завершення замовлення

Етапи бізнес-процесу тісно пов'язані між собою, формуючи безперервний цикл. Наприклад, успішна ініціація замовлення залежить від актуальності даних у каталозі товарів (згаданого в підрозділі 1.1), а перевірка доступності напряму впливає на процес оформлення. Помилка на етапі виконання замовлення, наприклад, через відсутність товарів на складі, може призвести до необхідності обробки повернення. За даними аналітичних звітів, компанії, які оптимізують ці етапи, можуть зменшити час обробки замовлень на 20–30%, що позитивно впливає на повторні покупки.

У високонавантажених системах, таких як великі e-commerce платформи, бізнес-процеси управління стикаються з додатковими викликами. Під час пікових періодів, наприклад сезонних розпродажів, обсяг замовлень може зростати у 5–10 разів, що вимагає ефективного розподілу навантаження між етапами. Автоматизація процесів, таких як перевірка доступності та резервування товарів стає критичною для уникнення помилок і забезпечення безперервності обслуговування. До того ж, інтеграція з внутрішніми системами (управлінням складом) і зовнішніми каналами (підтримкою клієнтів) потребує високого рівня синхронізації.

Ефективне управління замовленнями сприяє підвищенню рівня задоволеності клієнтів, зменшенню операційних витрат і створенню конкурентної переваги. Скорочення часу доставки на 1 день може збільшити відсоток повторних покупок на 5–7%, що підтверджується дослідженнями у сфері e-commerce. Водночас збої на будь-якому етапі можуть призвести до втрати репутації та фінансових збитків, що підкреслює необхідність розробки гнучких і масштабованих рішень.

1.3 Роль інформаційних систем у підтримці управлінських рішень

Інформаційні системи відіграють важливу роль у забезпеченні ефективного управління бізнес-процесами в електронній комерції, в тому числі у досліджуваній моделі B2C. Вони виступають інструментом для збору, обробки та аналізу даних, що дозволяє менеджерам приймати обґрунтовані управлінські рішення в умовах високого навантаження та динамічних ринкових змін. У контексті описаних у підрозділах 1.1 і 1.2 ключових процесів (управління каталогом, складом, замовленнями) та етапів створення замовлень (вибір товарів, перевірка наявності, резервування) ІС забезпечують автоматизацію, інтеграцію та доступ до реального часу, що є важливим для конкурентоспроможності.

Інформаційні системи виконують кілька функцій, які підтримують управлінські рішення:

– завдяки збору і зберіганню даних ІС акумулюють інформацію про клієнтів, товари, запаси та статуси замовлень, створюючи єдину базу для аналізу;

– через автоматизацію процесів, зокрема перевірки наявності чи резервування товарів, ІС зменшують час обробки та помилки, що дозволяє менеджерам зосередитися на стратегічних задачах;

– за допомогою аналізу і прогнозування ІС надають аналітичні інструменти для оцінки попиту, виявлення тенденцій і прогнозування пікових навантажень, що допомагає оптимізувати розподіл ресурсів;

– завдяки моніторингу і звітності системи забезпечують оперативний доступ до даних про виконання замовлень, що дозволяє менеджерам реагувати на збої чи зміни в реальному часі.

Одним із сучасних підходів до реалізації інформаційних систем є використання мікросервісної архітектури, яка передбачає розподіл функціональності на незалежні модулі. Такий підхід дозволяє кожному модулю працювати автономно, що забезпечує гнучкість і масштабованість системи. У контексті підтримки управлінських рішень, мікросервіси сприяють швидкому доступу до даних, оскільки кожен модуль може обробляти запити паралельно, зменшуючи затримки під час пікових навантажень.

Інформаційні системи інтегрують бізнес-процеси забезпечуючи їхню узгодженість. Наприклад, дані про наявність товарів, зібрані в ІС, дозволяють автоматизувати перевірку на етапі створення замовлення, а аналітика попиту допомагає менеджерам планувати оновлення каталогу чи поповнення складів. За даними досліджень, компанії, які використовують ІС для автоматизації та аналізу, можуть підвищити операційну ефективність на 15–25% і зменшити витрати на управління запасами на 10–15%, що згадується у роботі [5]. Взаємодію ІС з бізнес-процесами можна переглянути на рисунку 1.1.

Управлінські рішення в електронній комерції враховують такі дані, як розподіл ресурсів або коригування асортименту, які надходять з ІС. Наприклад, історичні дані про замовлення можна використовувати для прогнозування попиту на певні продукти, а моніторинг статусів замовлення можна використовувати для виявлення вузьких місць у процесі. ІС балансує обсяг замовлень із доступними ресурсами у високонавантажених системах, щоб уникнути перевантажень і підвищити задоволеність клієнтів. Таку здатність додатково підсилює архітектура мікросервісів, яка представляє агреговані звіти з різних модулів системи полегшуючи стратегічне планування.

Рисунок 1.1 – Схема взаємодії інформаційних систем із бізнес-процесами та управлінськими рішеннями

Висновки до розділу 1

Отже, аналіз показав, що модель B2C повинна бути обрана як основа для створення системи управління замовленнями в рамках цього дослідження, оскільки вона забезпечує швидкість, зручність та доступність онлайн-покупок для кінцевих споживачів. З'ясувалося, що ефективність бізнес-процесів цієї моделі безпосередньо залежить від того, наскільки добре автоматизовані такі етапи, як автентифікація, обробка замовлення, а також узгодження та доставка товарів, що допомагає знизити транзакційні ризики та дати конкурентні переваги платформі на тлі зростаючого ринку e-commerce.

Важливим висновком дослідження вважається необхідність створення інформаційної системи на мікросервісній архітектурі, здатної підтримувати управлінські рішення, що забезпечують гнучкість та масштабованість в умовах високого навантаження. Це дає ґрунт для подальшої розробки системи, яка б допомогла ефективно управляти замовленнями, враховуючи особливості бізнес-процесів та змінні вимоги ринку на прикладі e-commerce.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Розробка систем управління замовленнями в електронній комерції потребує вибору відповідної архітектури, яка забезпечить високу продуктивність, масштабованість і адаптивність до змін бізнес-вимог. На сьогодні основними підходами до побудови таких систем є монолітна та мікросервісна архітектури. Аналіз їхніх особливостей дозволяє визначити оптимальний підхід для реалізації системи, орієнтованої на обробку замовлень у високонавантажених умовах e-commerce.

2.1 Порівняння монолітних і мікросервісних архітектур

Інформаційні системи для електронної комерції традиційно розроблялися на базі монолітних архітектур, де всі компоненти системи інтегруються в єдине ціле. Проте з ростом складності бізнес-процесів та необхідністю підтримки управлінських рішень, такі архітектури стикаються з обмеженнями. Альтернативою є мікросервісна архітектура, яка набирає популярності завдяки своїй модульності та адаптивності. Порівняння цих підходів дозволяє оцінити їхні переваги та недоліки в контексті e-commerce.

Монолітна архітектура передбачає єдину програму, де всі функціональні модулі (наприклад, обробка замовлень, управління запасами) об'єднані в одному кодовому базисі. Це спрощує початкову розробку та розгортання, але ускладнює масштабування та оновлення. Мікросервісна архітектура, навпаки, розподіляє систему на незалежні сервіси, кожен із яких відповідає за конкретну задачу. Такі сервіси можуть працювати автономно, що забезпечує гнучкість і паралельну розробку. Порівняння монолітної та мікросервісної архітектур за ключовими критеріями представлено в таблиці 2.1.

У контексті електронної комерції монолітні архітектури підходять для невеликих платформ зі стабільним робочим навантаженням, де швидке впровадження є пріоритетом. Однак із зростанням обсягів замовлень і необхідністю швидко реагу-

вати на зміни ринку, монолітні системи стають неефективними. Архітектури мікросервісів більше підходять для великих платформ, яким необхідно обробляти високі навантаження (наприклад, у періоди сезонних розпродажів).

Таблиця 2.1 – Порівняння монолітних і мікросервісних архітектур

Критерій	Монолітна архітектура	Мікросервісна архітектура
Масштабованість	Обмежена. Масштабування потребує розгортання всієї системи, що збільшує витрати ресурсів	Висока. Кожен мікросервіс масштабується незалежно (наприклад, лише Order Service під час пікових навантажень)
Гнучкість	Низька. Зміни в одному модулі потребують перекомпіляції всієї системи	Висока. Мікросервіси оновлюються незалежно, що спрощує додавання нових функцій
Складність розробки	Простіша на початковому етапі. Єдиний код і база даних знижують складність інтеграції	Висока. Потребує координації між сервісами, налаштування API, брокерів повідомлень
Продуктивність	Висока в малих системах. Відсутність мережевих затримок між модулями	Може бути нижчою через мережеві виклики між сервісами, але оптимізується кешуванням і асинхронністю

2.2 Огляд технологій для реалізації мікросервісів

Мікросервісна архітектура, як перспективний підхід до побудови інформаційних систем, вимагає використання певних методів і принципів реалізації, які забезпечують її ефективність. Реалізація такої архітектури базується на кількох основних принципах, які забезпечують її функціональність:

– завдяки розподілу на незалежні сервіси кожен компонент виконує специфічну задачу, наприклад, сервіс обробки замовлень чи сервіс перевірки наявності товарів;

– через ізоляцію даних кожен сервіс має власну базу, наприклад, PostgreSQL для сервісу каталогів і MongoDB для сервісу замовлень, що підвищує безпеку та автономність;

– за допомогою асинхронної взаємодії між сервісами, наприклад, через брокери повідомлень типу RabbitMQ, зменшуються залежності і підвищується продуктивність під час пікових навантажень;

– завдяки централізованому управлінню конфігураціями, наприклад, за допомогою Spring Cloud Config, система швидко адаптується до змін без перебудови всіх компонентів.

Для розгортання мікросервісів застосовуються різні технології, які адаптують систему до специфічних потреб. Розподілене розгортання передбачає розміщення сервісів на окремих обчислювальних вузлах – наприклад, сервіс управління складом може працювати на одному сервері, а сервіс обробки замовлень – на іншому, що оптимізує використання ресурсів. Контейнеризація за допомогою Docker ізолює середовище кожного сервісу, створюючи окремі контейнери для сервісів каталогів і замовлень, що спрощує їхнє розгортання та оновлення. Оркестрація через Kubernetes забезпечує координацію між сервісами, автоматично масштабуючи їх під час високих навантажень, таких як сезонні розпродажі. Ці методи дозволяють платформам e-commerce ефективно реагувати на зміни в обсягах замовлень.

Використання конкретних технологій для мікросервісів супроводжується певними труднощами. Складність координації між сервісами, наприклад, узгодження баз даних PostgreSQL і MongoDB, може призводити до помилок під час оновлень. Моніторинг продуктивності вимагає додаткових інструментів, таких як Prometheus для збору метрик і ELK Stack для аналізу логів, щоб оперативно виявляти збої в кожному сервісі. Незважаючи на ці виклики, мікросервіси забезпечують швидке відновлення після помилок і адаптацію до змін.

Розвиток технологій відкриває нові можливості для мікросервісів у електронній комерції. Оркестрація через Kubernetes дозволяє автоматизувати масштабування, додаючи нові екземпляри сервісу обробки замовлень під час пікових навантажень. Безсерверні архітектури, такі як AWS Lambda, зменшують витрати на інфраструктуру, запускаючи сервіси лише за потреби. Ці інновації особливо актуальні для платформ, що стикаються з високими навантаженнями, як описано в підрозділі 1.2, і потребують гнучкості для підтримки стратегічних рішень.

2.3 Приклади мікросервісних систем у провідних e-commerce

У цьому підрозділі розглядаються приклади впровадження мікросервісної архітектури в провідних компаніях: Amazon, Netflix, eBay та Uber. Ці компанії демонструють інноваційні підходи до трансформації цифрових екосистем, підкреслюючи ключову роль мікросервісів у створенні адаптивних і високопродуктивних платформ для сучасної комерції та технологічних рішень.

Amazon, глобальний лідер у сфері електронної комерції, успішно впровадив мікросервісну архітектуру для обробки величезних обсягів замовлень, особливо під час подій, таких як Prime Day. Кожен сервіс функціонує незалежно, що дозволяє масштабувати його під час підвищення попиту. Компанія використовує контейнеризацію через Docker для ізоляції сервісів і оркестрацію за допомогою Kubernetes, це забезпечує автоматичне розподілення навантаження. Такий підхід гарантує безперервну доступність платформи навіть за мільйонів одночасних користувачів, підтримуючи швидку обробку замовлень і дозволяє менеджерам оперативно адаптувати ресурси.

Netflix, відомий переважно потоковим контентом, адаптував мікросервіси для забезпечення стійкості та персоналізації, що можна застосувати в e-commerce. Система розподілена на сотні мікросервісів, кожен із яких обробляє специфічні задачі, наприклад, рекомендації чи управління каталогом контенту. Компанія використовує Eureka для сервісного виявлення, що дозволяє уникати збоїв – якщо один сервіс виходить із ладу, інші продовжують працювати. Цей підхід демонструє, як

мікросервіси підтримують безперервність бізнес-процесів, на рисунку 2.1 представлений фрагмент мікросервісів Netflix з роботи [6]. Схема демонструє складну мережу, де Apache Kafka забезпечує обмін подіями, а Cassandra і MySQL підтримують дані, що гарантує стійкість навіть при відмовах окремих компонентів.

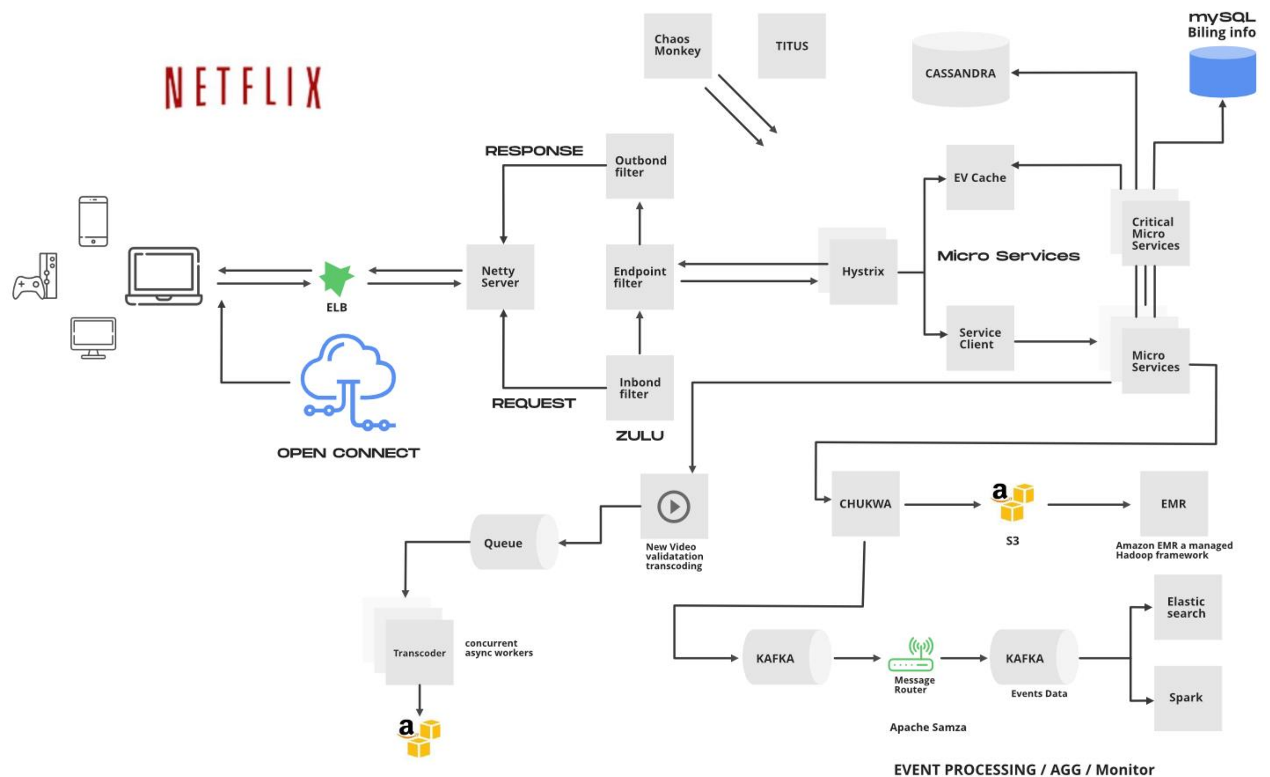

Рисунок 2.1 – Схема мікросервісної архітектури Netflix

eBay впровадив мікросервісну архітектуру, щоб забезпечити гнучкість у оновленнях своїх платформ. Наприклад, сервіс управління замовленнями може бути модернізований без впливу на сервіс каталогів, що дозволяє швидко додавати функції, такі як покращені фільтри пошуку. Компанія використовує Docker для контейнеризації сервісів і Spring Boot для їхньої розробки. Така гнучкість полегшує адаптацію до ринкових змін, що є важливим для прийняття управлінських рішень, як зазначено в підрозділі 1.3, і допомагає уникати тривалих простоїв. Рисунок 2.2 відображає модульну структуру з окремими доменами (Homepage, Search, Product), де кожен модуль-будівельник (Module Builder) інтегрується через власні сервіси, забезпечуючи незалежне оновлення без порушення роботи всієї системи.

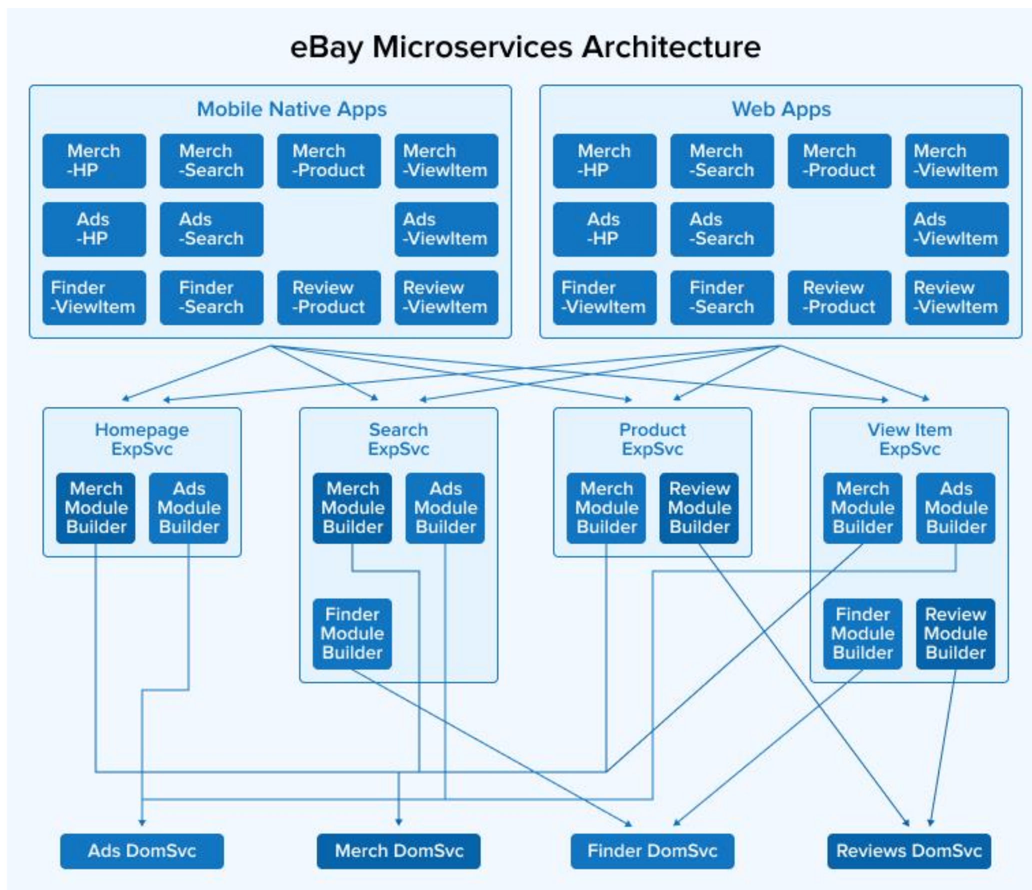

Рисунок 2.2 – Схема мікросервісної архітектури eBay

Uber, хоча відомий як сервіс таксі, адаптував мікросервіси для оптимізації операцій, що може бути прикладом для платформ подібних до розроблюваної. Система розподілена на мікросервіси, які обробляють різні аспекти, наприклад, управління доступністю водіїв чи маршрутами, що аналогічно до управління запасами в e-commerce. Компанія використовує gRPC для швидкої взаємодії, що забезпечує високу продуктивність. Цей підхід дозволяє швидко реагувати на зміни в попиті. На рисунку 2.3 наведена схема архітектури мікросервісів Uber взята з роботи [7] та проаналізована у статті [8]. Схема Uber демонструє централізований API-шлюз, який координує сервіси, такі як Passenger Management і Billing, через REST API, забезпечуючи ефективну обробку запитів від веб- і мобільних інтерфейсів. Дана схема найбільше підходить до реалізації системи дипломного проєкту, оскільки вона забезпечує чітку координацію між сервісами.

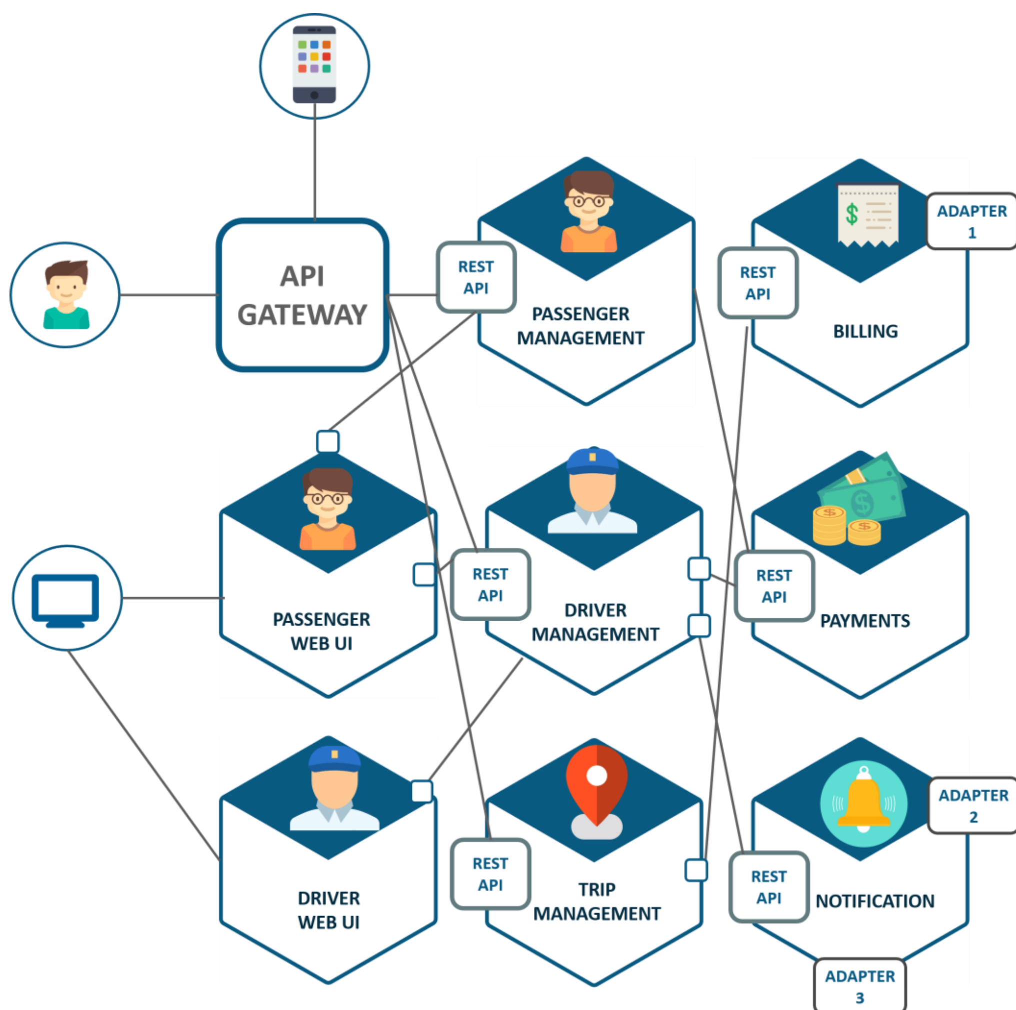

Рисунок 2.3 – Схема мікросервісної архітектури Uber

Висновки до розділу 2

Розглянуті вище компанії покладаються на технології, подібні до тих, що згадані в підрозділі 2.2, такі як Docker для контейнеризації, Kubernetes для оркестрації та Apache Kafka для асинхронної взаємодії. Amazon і Uber активно використовують гнучкі API на основі REST, тоді як Netflix і eBay інтегрують gRPC для підвищення продуктивності. Аналіз даних кейсів підкреслює, що успіх мікросервісів залежить від чіткого розподілу сервісів і надійного моніторингу, наприклад, через Prometheus, що може бути адаптовано для менших платформ e-commerce. Усе це демонструє важливість планування архітектури з урахуванням специфіки бізнес-процесів.

У рамках аналізу існуючих рішень можна зробити висновок, що мікросервісна архітектура має значні переваги над монолітною для систем e-commerce, зокрема у забезпеченні гнучкості, масштабованості та швидкості обробки запитів у високонавантажених умовах, хоча й потребує складнішого управління та координації між сервісами, як показано в порівнянні монолітних і мікросервісних архітектур у підрозділі 2.1. До того ж, мікросервісна модель дозволяє легко інтегрувати нові функції, але вимагає ретельного тестування для уникнення неочікуваних збоїв.

Огляд технологій для реалізації мікросервісів показав необхідність вибору стеку технологій, зокрема інструментів для контейнеризації, які забезпечують ізоляцію сервісів, засобів для оркестрації, що дозволяють простіше управляти розподіленими модулями, а також фреймворку з мовою програмування для розробки незалежних сервісів, які сприяють швидкій інтеграції та обробці даних у реальному часі. Приклади мікросервісних систем у провідних e-commerce платформах вказують їхню здатність забезпечувати швидке масштабування та адаптацію до пікових навантажень, що є перевагою для конкурентоспроможності, водночас підкреслюючи важливість ретельного планування для уникнення надмірної складності.

3 ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ

3.1 Функціональні вимоги до системи управління замовленнями

Система керування замовленнями, розроблена для моделі B2C, повинна гарантувати ефективне виконання бізнес-процесів, що включають в себе створення, контроль та резервацію замовлень, описаних в підрозділі 1.2, а також забезпечувати підтримку управлінських рішень в режимі реального часу, відповідно до підрозділу 1.3. Функціональні потреби сформульовані з урахуванням мікросервісної архітектури, обраної після аналізу досвіду компанії Uber, схема якої найбільш придатна для втілення цієї системи. Діаграма прецедентів наведена на кресленику ІС13.290БАК.005 Д3, вона показує заплановану функціональність системи. Далі подано багатоступеневий перелік вимог, що визначають ключові функціональні можливості системи:

а) управління каталогом товарів:
   1) перегляд списку товарів користувачем із базовою інформацією;
   2) перегляд детальної інформації про товар користувачем;
   3) додавання нового товару до каталогу менеджером;
   4) редагування даних товару менеджером;
   5) видалення товару з каталогу менеджером із оновленням залишків;
   6) перегляд повного списку товарів із додатковими даними адміном;

б) створення та обробка замовлень:
   1) формування замовлення користувачем шляхом вибору товарів;
   2) перегляд деталей створеного замовлення перед його збереженням;
   3) збереження замовлення з присвоєнням унікального ідентифікатора;
   4) скасування замовлення користувачем до завершення обробки;
   5) перегляд менеджером усіх створених замовлень користувача;
   6) перегляд історії всіх замовлень адміністратором;

в) облік користувачів:
   1) реєстрація нового клієнта з вказанням електронної пошти та пароля;
   2) перегляд профілю клієнта з інформацією про його замовлення;

3) перегляд списку всіх користувачів адміністратором;

4) деактивація облікового запису адміністратором;

5) призначення ролі менеджера новому користувачу адміністратором;

г) автентифікація та безпека доступу:

1) вхід у систему за допомогою електронної пошти та пароля;

2) оновлення контактних даних у профілі авторизованим клієнтом;

3) зміна пароля авторизованим користувачем;

4) вимкнення профілю клієнта за його запитом зі збереженням даних;

5) автоматичне завершення сесії після певного часу неактивності;

6) обмеження доступу до функцій залежно від ролі користувача;

д) управління доставкою:

1) створення запису про доставку під час формування замовлення;

2) відображення деталей доставки для клієнта;

3) перегляд усіх доставок, пов’язаних із замовленнями, менеджером;

4) оновлення статусу доставки (наприклад, "в обробці") менеджером;

5) перегляд історії змін статусів доставки адміністратором;

6) інформування клієнта про статус доставки через профіль;

е) контроль запасів:

1) перевірка наявності товарів на складі перед створенням замовлення;

2) резервування товарів на складі після створення замовлення;

3) оновлення даних про запаси після обробки замовлення чи скасування;

4) додавання нових запасів менеджером із зазначенням кількості;

5) редагування кількості запасів для конкретного товару менеджером;

6) перегляд повної історії змін запасів.

3.2 Нефункціональні вимоги до системи управління замовленнями

Визначення нефункціональних вимог необхідно для забезпечення оптимальної роботи системи, її надійності та безпеки, що є основою для ефективної обробки

запитів і підтримки бізнес-процесів. Дані вимоги гарантують стабільність серверної частини під час зростання навантаження та взаємодії з різними користувачами.

Система має забезпечувати високу швидкість обробки запитів, зокрема час обробки запиту на перегляд списку товарів сервером не повинен перевищувати 2 секунди за умов нормального навантаження. Створення замовлення на сервері має завершуватися протягом не більше ніж 5 секунд. Обробка запиту адміністратора на перегляд усіх замовлень має виконуватися за 5 секунд для бази даних, що містить до 10 000 записів.

Серверна частина системи повинна підтримувати збільшення навантаження, витримувати одночасну обробку запитів від 60 активних користувачів без зниження продуктивності. Кожен мікросервіс, зокрема для управління каталогом чи запасами, має бути здатним до горизонтального масштабування при зростанні навантаження, а додавання нового мікросервісу не повинно порушувати роботу існуючих серверних компонентів.

Система має гарантувати високу доступність та стійкість до збоїв, забезпечуючи доступність бекенду на рівні не менше 99 відсотків часу протягом місяця, у разі відмови одного мікросервісу, інші сервіси, такі як каталог чи замовлення, повинні продовжувати обробку запитів. Відновлення бекенду після збою одного сервісу має відбуватися не довше ніж за 5 хвилин із мінімальною втратою даних.

Забезпечення безпеки даних є пріоритетом системи, усі дані, що зберігаються на сервері, зокрема паролі, повинні бути зашифровані. Доступ до серверних функцій має бути обмежений ролями, такими як адміністратор, менеджер або користувач.

Інтерфейс бекенду повинен бути зручним і надійним, усі запити до бекенду мають проходити валідацію на відповідність формату перед обробкою. Взаємодія між бекендом і фронтендом має здійснюватися виключно через формат JSON. Помилки, що повертаються сервером, наприклад, у разі відсутності товару, повинні супроводжуватися зрозумілими повідомленнями для користувача.

Система має бути сумісною з різними платформами та стандартами, бекенд повинен підтримувати REST API для взаємодії з фронтендом. Серверні компоненти

мають коректно функціонувати на операційних системах, таких як Linux та Windows Server, а мікросервіси бекенду повинні відповідати стандартизованим протоколам для майбутньої інтеграції з іншими системами.

Для зручності модифікації та аналізу системи код бекенду має супроводжуватися коментарями та документацією. Бекенд повинен вести деталізоване логування подій для відстеження та аналізу помилок, а оновлення одного мікросервісу не повинно призводити до зупинки роботи інших серверних компонентів.

3.3 Критерії оцінки мікросервісної архітектури системи

Критерії оцінки мікросервісної архітектури дозволяють визначити, наскільки успішно реалізована розподілена структура. Оцінка проводиться з урахуванням потреб різних ролей користувачів (адміністраторів, менеджерів, клієнтів), але з фокусом на технічні характеристики архітектури.

Автономність мікросервісів є основним критерієм їхньої ефективності. Кожен мікросервіс має працювати незалежно, дозволяючи виконувати операції, такі як перегляд товарів, навіть якщо сервіс доставки тимчасово недоступний. Час, необхідний для перезапуску одного сервісу після внесення змін, має бути меншим за 2 хвилини. Кількість залежностей між мікросервісами має бути мінімізованою, щоб уникнути каскадних збоїв у разі відмови одного з них.

Ефективність взаємодії між мікросервісами є важливим показником якості архітектури. За прогнозами, час обміну повідомленнями між сервісами має бути меншим за 200 мілісекунд. Кількість помилок під час взаємодії між сервісами (наприклад, через некоректні API-виклики) має бути меншою за 1% від загальної кількості транзакцій за місяць. Обсяг даних, переданих між сервісами за один виклик API, має бути оптимізованим і не перевищувати 10 КБ, щоб зменшити навантаження на мережу.

Швидкість розгортання та оновлення мікросервісів визначає гнучкість системи. Час розгортання нового мікросервісу (наприклад, сервісу для управління про-

філями користувачів) на сервері має бути меншим за 10 хвилин, що дозволяє швидко додавати нові функції. Оновлення одного сервісу має відбуватися без зупинки функціонування інших сервісів, забезпечуючи безперервність роботи системи. Кількість успішних розгортань без збоїв має становити не менше 95% від усіх спроб за місяць, що свідчить про стабільність процесу.

Висновки до розділу 3

Формування функціональних і нефункціональних вимог до системи управління замовленнями завершило підготовчий етап для створення ефективного рішення для моделі B2C, що відповідає сучасним викликам електронної комерції. Систематизація вимог створює основу для впровадження достатньо гнучкої та ефективної платформи. Запропоновані сучасніші критерії оцінки архітектури мікросервісу підкреслюють її потенціал для досягнення високої автономності сервісів і швидкості їх взаємодії, що відкриває перспективи масштабування системи та адаптації до мінливих умов ринку. Таким чином, сформовані вимоги розкривають можливості системи щодо забезпечення конкурентоспроможності платформ електронної комерції, які будуть детально розглянуті на етапах її впровадження та тестування в наступних розділах.

4 ВИБІР ТЕХНОЛОГІЙ РОЗРОБКИ

4.1 Аналіз мов програмування та фреймворків

Однією з провідних мов програмування для мікросервісів є Java, яка відома своєю високою продуктивністю та стабільністю завдяки сильним типам і розвиненій екосистемі. Фреймворк Spring Boot, побудований на Java, дозволяє швидко створювати мікросервіси з вбудованими функціями, такими як автоматичне налаштування API та управління конфігураціями, що робить його популярним серед компаній, подібних до Uber, де Java використовується для складних інтеграцій. Однак Java може мати складність у налаштуваннях і більший розмір бінарних файлів, що впливає на час розгортання.

Альтернативою є мова Go (Golang), яка набирає популярності завдяки легкості, швидкості виконання та вбудованій підтримці конкурентності, також застосовується в Uber для легковагових сервісів. Фреймворк Gin, написаний для Go, забезпечує просту розробку API з мінімальним споживанням ресурсів, але має обмежену екосистему бібліотек.

Іншою опцією є Python, що вирізняється простотою синтаксису та широким набором бібліотек для розробки. Фреймворк Flask, побудований на Python, підходить для створення невеликих мікросервісів завдяки своїй легкості та гнучкості, але поступається у продуктивності при високих навантаженнях.

Node.js, базований на JavaScript, пропонує асинхронну обробку запитів і високу швидкість завдяки моделі подій, що робить його популярним для реального часу, однак потребує ретельного управління пам’яттю для уникнення витоків. Фреймворк Express у поєднанні з Node.js дозволяє легко створювати API, але не завжди відповідає вимогам до складних систем.

Окрім того, .NET є потужною платформою від Microsoft, яка підтримує кросплатформність і продуктивність завдяки фреймворку ASP.NET. Використання .NET популярне в компаніях із великими корпоративними системами, але може бути надмірним для легких мікросервісів через складність інфраструктури.

Порівняння технологій наведено в таблиці 4.1, воно базується на таких характеристиках, як продуктивність, легкість розробки, підтримка мікросервісів та ресурсоємність. Аналіз показує, що Java з Spring Boot забезпечує оптимальну комбінацію стабільності, розвиненої екосистеми та інтеграційних можливостей, що ідеально підходить для реалізації складної мікросервісної архітектури. Go з Gin є конкурентним через швидкість і легкість, але поступається в інтеграційних можливостях. Python з Flask та Node.js з Express підходять для швидкої розробки, але менш ефективні при великих навантаженнях, а .NET Core з ASP.NET є сильним рішенням для корпоративних систем, але надмірним для поточних потреб. Таким чином, остаточний вибір припадає на Java з фреймворком Spring Boot, що буде детально обґрунтовано в наступних підрозділах.

Таблиця 4.1 – Порівняння мов програмування та фреймворків

Мова/Фреймворк	Переваги	Недоліки	Підходить для систем
Java / Spring Boot	Висока продуктивність, інтеграції, стабільність	Складність налаштувань, великий розмір	Складні мікросервісні системи
Go / Gin	Швидкість, легкість, низьке споживання	Обмежена екосистема бібліотек	Легковагові та масштабовані сервіси
Python / Flask	Простота, гнучкість	Низька продуктивність при навантаженні	Прототипи та невеликі сервіси
Node.js / Express	Асинхронність, швидка розробка	Ризик витоків пам'яті	Системи реального часу
.NET / ASP.NET Core	Кросплатформність, продуктивність	Складність інфраструктури	Корпоративні та кросплатформні системи

4.2 Огляд інструментів для синхронної та асинхронної взаємодії

Використання синхронних та асинхронних інструментів є необхідним для забезпечення ефективної взаємодії між окремими сервісами у системі. Синхронні інструменти дозволяють отримувати негайні відповіді на запити, що важливо для операцій, де користувач чекає миттєвого результату. Асинхронні інструменти, навпаки, забезпечують обробку завдань у фоновому режимі, наприклад, оновлення статусу доставки чи резервування товарів, що зменшує навантаження на систему та покращує її масштабованість. Такий підхід дозволяє адаптувати архітектуру до різних сценаріїв використання, оптимізуючи продуктивність і стабільність, що є ключовим для розподілених систем, де комбінація цих механізмів забезпечує ефективну взаємодію між сервісами. З огляду на вибір Java з фреймворком Spring Boot, інструменти мають бути сумісними з цією технологією, а також відповідати критеріям оцінки архітектури, визначеним у підрозділі 3.3.

Синхронна взаємодія між мікросервісами зазвичай реалізується через REST API, який забезпечує простий і стандартизований спосіб обміну даними. У Spring Boot REST API підтримується через вбудовані контролери, що дозволяють створювати HTTP-запити для таких операцій, як перегляд списку товарів чи створення замовлення. REST є ефективним для сценаріїв, де потрібна негайна відповідь, хоча може створювати затримки при високих навантаженнях через блокуючий характер запитів. Робота REST взаємодії проілюстрована на рисунку 4.1. Крім того, для підвищення гнучкості системи REST API може бути інтегровано з проміжним програмним забезпеченням, таким як API Gateway, яке оптимізує маршрутизацію запитів і забезпечує додатковий рівень безпеки.

Альтернативою REST є gRPC, який використовує протокол HTTP/2 і забезпечує високу продуктивність завдяки бінарному формату даних (Protobuf). gRPC підтримується в Spring Boot через бібліотеки, такі як grpc-spring-boot-starter, і є кращим для сценаріїв із великою кількістю викликів, наприклад, між сервісом замовлень і запасів, але вимагає додаткових зусиль для налаштування. Схема технології наведена на рисунку 4.2. Для поточної системи перевага віддається REST API через

його простоту, широку підтримку в Spring Boot і достатню продуктивність для базових сценаріїв синхронної взаємодії, таких як перегляд даних.

Рисунок 4.1 – Взаємодія між клієнтом та сервером з використанням REST API

Рисунок 4.2 – Схема взаємодії між клієнтом та сервером з використанням gRPC

Асинхронна взаємодія необхідна для сценаріїв, де затримки є неприпустимими, наприклад, для оновлення статусу доставки чи резервування товарів після створення замовлення. Apache Kafka є одним із найпопулярніших інструментів для асинхронного обміну повідомленнями, що підтримує модель "видавець-підписник"

і забезпечує високу масштабованість. У Spring Boot Kafka інтегрується через бібліотеку Spring Kafka, дозволяючи сервісам обмінюватися повідомленнями через топіки, наприклад, для сповіщення сервісу запасів про нове замовлення, але потребує додаткової інфраструктури для управління брокерами.

Альтернативою є RabbitMQ, який використовує модель черг повідомлень і підтримує протоколи, такі як AMQP. RabbitMQ легко інтегрується зі Spring Boot через Spring AMQP і підходить для сценаріїв із менш інтенсивним потоком даних, наприклад, для сповіщень про оновлення статусу, хоча поступається Kafka у масштабованість. З урахуванням потреб системи, де обсяг асинхронних операцій є помірним, а простота інтеграції є пріоритетом, перевагу віддано RabbitMQ як основному інструменту для асинхронної взаємодії. Нижче на рисунку 4.3 наведена схема брокеру повідомлень від RabbitMQ. Більше інформації про роботу RabbitMQ можна отримати з офіційної документації [9].

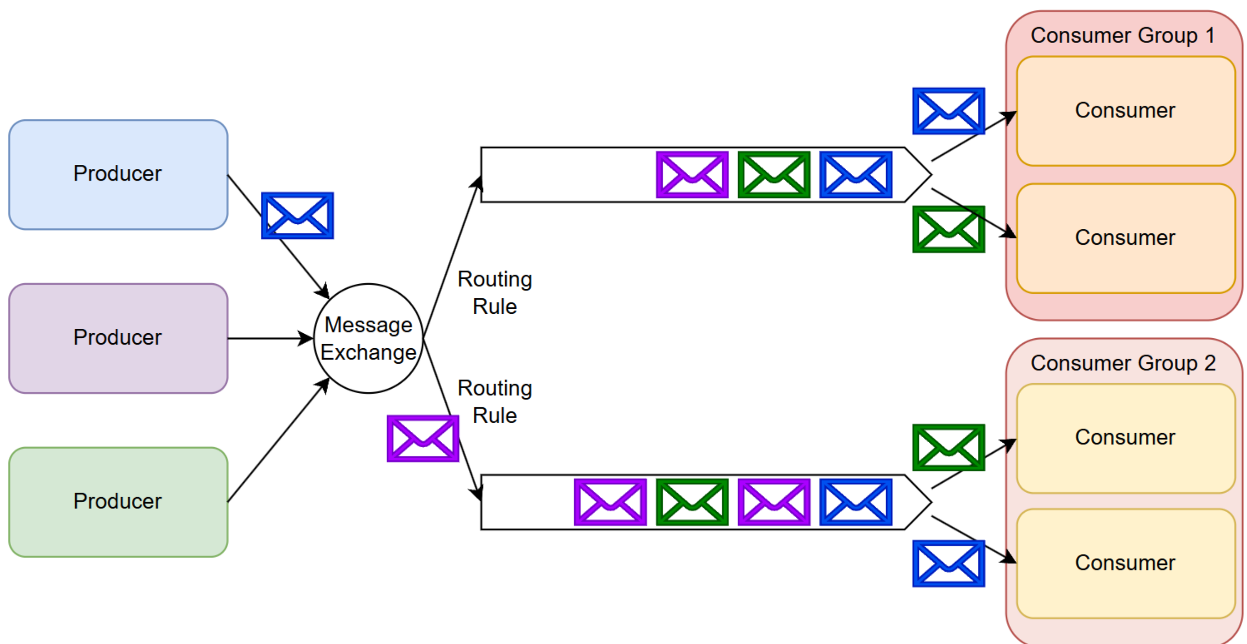

Рисунок 4.3 – Схема брокеру повідомлень для взаємодії між сервісами

Для визначення якості інструментів було оцінено ряд показників, серед яких швидкість роботи, здатність до масштабування, простота впровадження та взаємодія з екосистемою Spring Boot. REST API обрано для синхронної взаємодії через

його простоту, універсальну підтримку та достатню продуктивність для більшості операцій в межах системи. Щодо асинхронної взаємодії, RabbitMQ обрано завдяки простоті інтеграції зі Spring Boot та відповідності сценаріям з помірним об'ємом даних, наприклад, для сповіщень про статуси. Kafka, у свою чергу, була б більш оптимальною для систем, де обсяг транзакцій значно вищий.

4.3 Вибір інструментів контейнеризації, бази даних та безпеки

Для контейнеризації чудово підходить Docker – він дає змогу зібрати мікросервіси з їхніми залежностями у легкі контейнери. Docker відмінно працює із Spring Boot, що дозволяє створювати образи для сервісів, які містять REST API бібліотеки та RabbitMQ. Завдяки цьому, забезпечується стабільна робота в різних середовищах, що спрощує розгортання та полегшує процес тестування. Як альтернативу можна розглянути Podman, що пропонує контейнеризацію без використання центрального демона, це покращує безпеку, однак дана технологія має меншу екосистему та складніше інтегрується із Java-додатками, поступаючись Docker у зручності та підтримці.

Для бази даних було обрано PostgreSQL – рішення з відкритим кодом, яке відрізняється потужністю та функціональністю реляційних систем, включаючи підтримку комплексних запитів та транзакцій. PostgreSQL є оптимальним варіантом для структурованих даних, таких як відомості про замовлення, товари та їх наявність на складі. Інтеграція з Spring Boot здійснюється через JDBC або Hibernate. Це гарантує високу стабільність, що критично важливо з огляду на потенційне збільшення обсягів інформації. Альтернативою є MySQL, теж реляційна база даних, але вона програє PostgreSQL в контексті підтримки розширених типів даних та швидкості роботи при складних операціях. Це робить її менш відповідною для поточних вимог.

Для гарантування безпеки було обрано Keycloak, платформу з відкритим кодом для управління ідентифікацією та доступом (IAM). Keycloak взаємодіє з Spring Boot за допомогою адаптерів, що дає змогу реалізувати процеси аутентифікації та

авторизації для всіх користувачів з різними ролями (адміністраторів, менеджерів, клієнтів), підтримуючи протоколи OAuth2 та OpenID Connect. Завдяки цьому система може легко налаштовувати політики доступу для кожної ролі, що значно спрощує адміністрування прав користувачів. Такий підхід забезпечує захист REST API та RabbitMQ від несанкціонованого доступу, що є вкрай важливим для систем, які оперують конфіденційними даними. Як альтернатива розглядалась бібліотека Spring Security, інтегрована в Spring Boot, але її використання потребувало б значно більших зусиль для конфігурування складних сценаріїв аутентифікації, що робить Keycloak більш практичним і гнучким рішенням. Потенційно корисні інструменти наведено у таблиці 4.2. Ще Keycloak підтримує інтеграцію з LDAP і Active Directory, що дозволяє використовувати корпоративні системи автентифікації для великих організацій.

Таблиця 4.2 – Аналіз інструментів контейнеризації, бази даних та безпеки

Інструмент	Тип	Переваги	Недоліки
Docker	Контейнеризація	Широка підтримка, легкість створення образів	Ручне масштабування
Podman	Контейнеризація	Безпека, відсутність демона	Менша екосистема
PostgreSQL	Реляційна база даних	Надійність, підтримка складних запитів	Вимоги до ресурсів
MySQL	Реляційна база даних	Простота, широка підтримка	Обмежена підтримка розширень
MongoDB	NoSQL база даних	Гнучкість, швидкість при масштабуванні	Менша консистентність
Keycloak	Безпека	Гнучкість IAM, інтеграція з Spring	Складність початкового налаштування

Spring Security	Безпека	Вбудований у Spring Boot	Складність для різних сценаріїв

Висновки до розділу 4

Вибрані технології складають єдиний комплекс інструментів, який гарантує стійкість та адаптивність системи на всіх етапах її створення та функціонування. Docker дає змогу результативно керувати ізольованими оточеннями для кожного мікросервісу, тим самим спрощуючи подальше масштабування системи. PostgreSQL виступає фундаментом для надійного зберігання та обробки інформації, що є основою для реалізації бізнес-логіки. Keycloak додає рівень захисту, забезпечуючи безпеку взаємодії між користувачами та сервісами, створюючи передумови для стабільної роботи системи в реальних умовах.

5 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

5.1 Проєктування архітектури системи

5.1.1 Розподіл системи на мікросервіси

Система спроєктована у вигляді сукупності семи самостійних мікросервісів, кожен із яких відповідає за конкретний функціональний напрямок. У формуванні архітектури значну роль відіграла робота, зазначена в джерелі [10], а схема архітектури представлена на кресленику ІС13.290БАК.005 Д1. Визначені модулі взаємодіють із зовнішніми контрагентами, реалізуючи основні бізнес-операції, описані в підрозділі 3.1.

Перший мікросервіс, Auth Service, забезпечує аутентифікацію та авторизацію користувачів, відповідаючи за видачу токенів доступу та перевірку прав доступу. Ключовими точками доступу є ендпоінти для реєстрації та логіну, а саме POST /auth/register і POST /auth/login. Цей сервіс слугує основою для авторизації, до якої звертаються клієнти, менеджери та адміністратори перед доступом до інших сервісів.

Другий мікросервіс, User Service, управляє профілями користувачів, включаючи створення, оновлення та деактивацію акаунтів. Основними ендпоінтами є GET /user/me для отримання даних авторизованого користувача, GET /user/all для перегляду всіх користувачів адміністратором та DELETE /user/me для деактивації акаунта. Адміністратори використовують цей сервіс для адміністрування, тоді як клієнти та менеджери оновлюють власні профілі.

Третій мікросервіс, Catalog Service, відповідає за управління каталогом товарів, включаючи додавання, оновлення та деактивацію продуктів. Ключовими ендпоінтами слугують GET /product для перегляду каталогу, POST /product/add для додавання товару та POST /product/{ID}/deactivate для деактивації товару. Клієнти переглядають каталог, а менеджери здійснюють управління товарами.

Четвертий мікросервіс, Order Service, обробляє створення, оновлення та відстеження замовлень. Основними ендпоінтами є POST /order/create для створення

замовлення, GET /order/me для перегляду замовлень користувача та GET /order/{ID}/status для перевірки статусу. Клієнти створюють замовлення та відстежують їх, а менеджери переглядають для подальшої обробки.

П'ятий мікросервіс, Inventory Service, управляє складськими запасами, включаючи резервування та оновлення кількості товарів. Ключовими ендпоінтами виступають GET /inventory/all для перегляду запасів та POST /inventory/add для додавання товарів на склад. Цей сервіс використовується менеджерами для управління запасами.

Шостий мікросервіс, Delivery Service, координує процес доставки, включаючи відстеження та оновлення статусу. Основними ендпоінтами є GET /delivery/me для перегляду доставок користувача та GET /delivery/{ID}/status для перевірки статусу доставки. Клієнти відстежують доставку, а менеджери керують процесом.

Сьомий мікросервіс, API Gateway, служить єдиною точкою входу для зовнішніх запитів, маршрутизуючи їх до відповідних мікросервісів та забезпечуючи базову авторизацію. Усі актори, такі як клієнти, менеджери та адміністратори, звертаються до системи через цей шлюз.

Структура програмного застосунку вказана на рисунку 5.1.

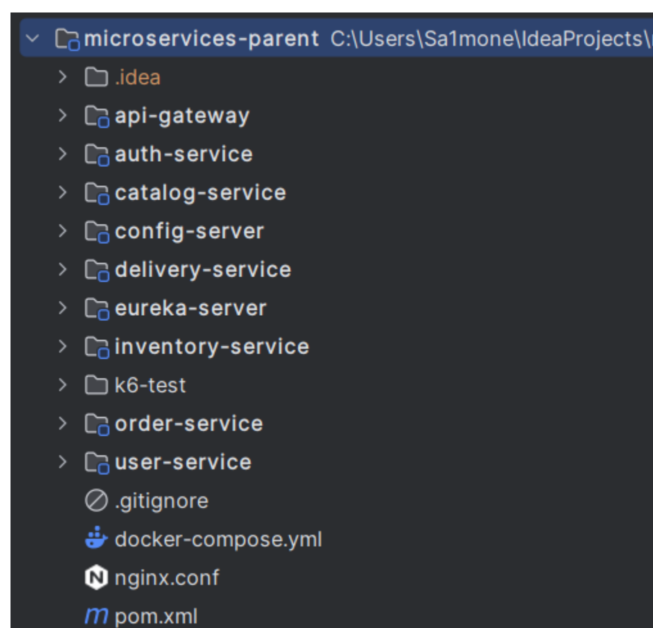

Рисунок 5.1 – Вигляд структури проекту в IDE

Кожен мікросервіс розроблено з урахуванням принципу єдиної відповідальності, що відкриває можливості для незалежного оновлення та масштабування окремих складових. Це дає змогу оптимізувати використання системних ресурсів, пристосовуючи кожен сервіс до конкретних пікових навантажень, а також спрощує підтримку та розширення функціональності в майбутньому. Такий підхід також забезпечує ізоляцію помилок, адже збій в одному сервісі не впливає на роботу інших, що підвищує загальну надійність системи. Ще завдяки такій структурі з'являється можливість інтеграції зі сторонніми сервісами, наприклад, кур'єрською службою, для автоматизації доставки. Мікросервіси та їхні ендпоінти детально проілюстровано на рисунку 5.1, який відображає сценарії використання системи.

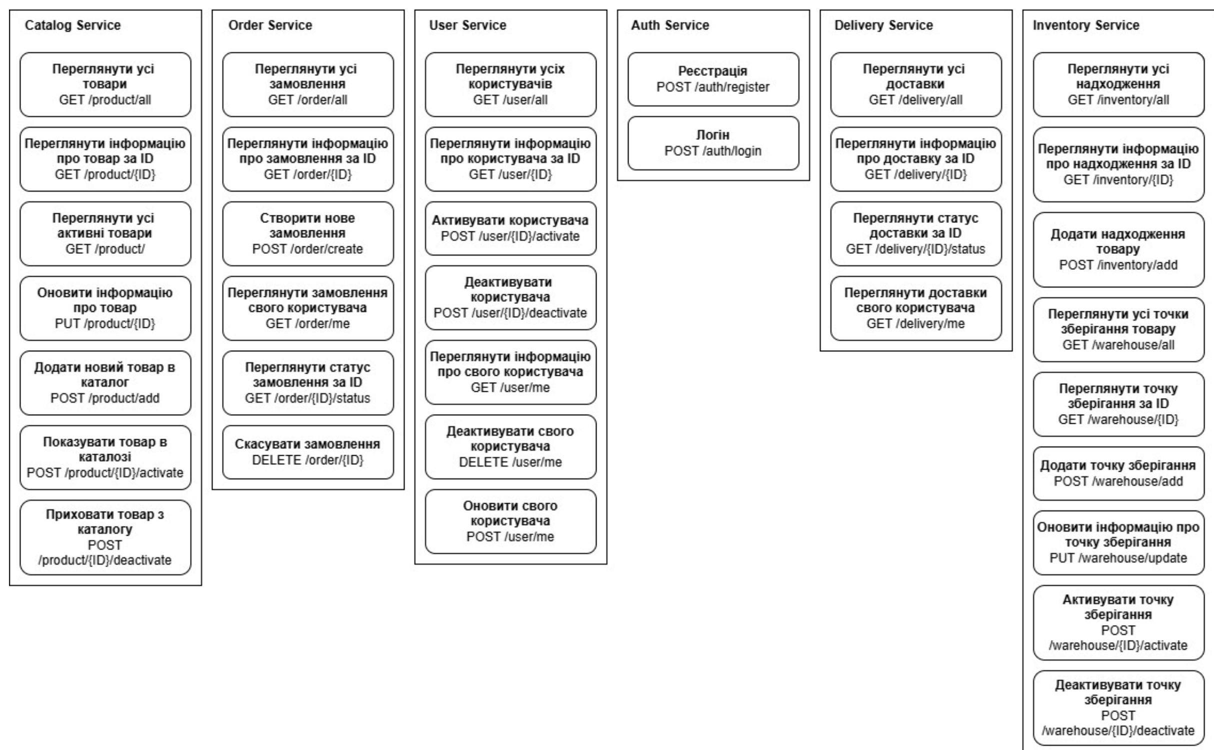

Рисунок 5.1 – Розподіл мікросервісів із зазначенням ендпоінтів

5.1.2 Організація взаємодії між мікросервісами

З технічної точки зору REST API в системі базується на Spring Boot із контролерами, що обробляють HTTP-запити. Кожен мікросервіс має набір кінцевих точок (ендпоінтів), які сприяють синхронному обміну інформацією. Для прикладу,

Catalog Service реагує на запит від API Gateway до ендпоінту для відображення товарів, передаючи інформацію про доступність до Order Service шляхом окремого виклику. Взаємодія здійснюється через HTTP/1.1 з застосуванням стандартних методів (GET, POST), а авторизація реалізована за допомогою JWT токенів, що видаються Auth Service і верифікуються Keycloak до початку маршрутизації. На рисунку 5.3 представлена відповідь системи на такий запит.

Рисунок 5.3 – Відповідь системи на запит перегляду каталогу товарів

У випадку помилок, таких як відсутність токена, система відповідає відповідним HTTP-статусом 401 Unauthorized. Для обробки помилок у кожному мікросервісі налаштовано глобальний обробник винятків через клас GlobalExceptionHandler проілюстрований на рисунку 5.4. Цей клас перехоплює різні винятки та формує стандартизовані відповіді. Наприклад, при виникненні винятку IllegalArgument

Exception (якщо запит на створення вже існуючої сутності) обробник повертає HTTP-статус 409 Conflict із повідомленням про помилку. У разі EntityNot FoundException (наприклад, якщо запитаний товар не існує) повертається HTTP-статус 404 Not Found. Для всіх інших непередбачених помилок обробник повертає HTTP-статус 500 Internal Server Error із загальним повідомленням.

```java
11  @RestControllerAdvice  Saimone2
12  public class GlobalExceptionHandler {
13
14      @ExceptionHandler(IllegalArgumentException.class)  Saimone2
15      public ResponseEntity<Map<String, Object>> handleIllegalArgumentException(IllegalArgumentException ex) {
16          return ResponseEntity.status(HttpStatus.CONFLICT).body(Map.of(
17                  k1: "success",  v1: false,
18                  k2: "message", ex.getMessage()
19          ));
20      }
21
22      @ExceptionHandler(EntityNotFoundException.class)  Saimone2
23      public ResponseEntity<Map<String, Object>> handleEntityNotFoundException(EntityNotFoundException ex) {
24          return ResponseEntity.status(HttpStatus.NOT_FOUND).body(Map.of(
25                  k1: "success",  v1: false,
26                  k2: "message", ex.getMessage()
27          ));
28      }
29
30      @ExceptionHandler(Exception.class)  Saimone2
31      public ResponseEntity<Map<String, Object>> handleGenericException() {
32          return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(Map.of(
33                  k1: "success",  v1: false,
34                  k2: "message",  v2: "An unexpected error occurred"
35          ));
36      }
37  }
```

Рисунок 5.4 – Фрагмент класу GlobalExceptionHandler в сервісу каталогу

Коли кошик користувача непустий, він відправляє запит на створення замовлення. Фронтенд надсилає запит до бекенду у формі JSON із кількістю товару та ідентифікатором товару, через POST /order/create до API Gateway, який перенаправляє його до Order Service. Order Service обробляє запит, перевіряє доступність товарів через запит до Inventory Service і, у разі успіху, створює замовлення, після чого ініціює асинхронний процес для оновлення запасів і статусу доставки. Рисунок 5.5 вказує послідовність із запитів, які виконуються для створення замовлення. На рисунку 5.6 наведений приклад запиту сформованого фронтендом.

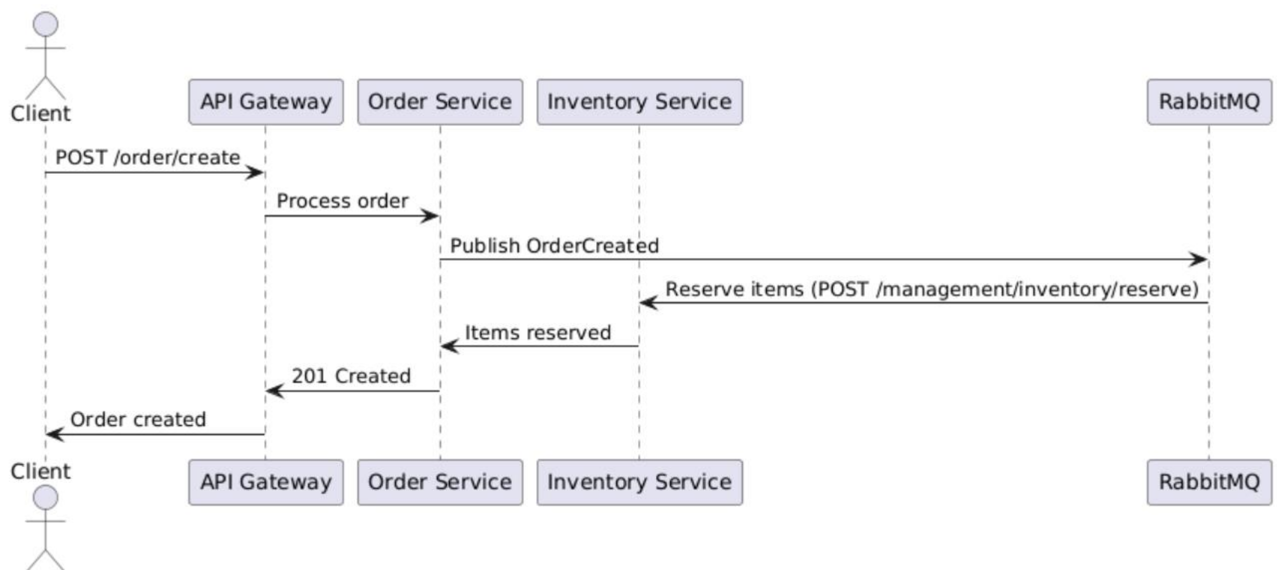

Рисунок 5.5 – Схема послідовності запитів для створення замовлення в системі

POST http://localhost:8080/order/create

Params Authorization Headers (9) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1  {
2      "deliveryAddress": "123 Main St",
3      "items": [
4          {
5              "productId": "e5a28697-d526-49cd-a459-87f451d1e016",
6              "quantity": 2
7          },
8          {
9              "productId": "3b26bcba-37a4-4975-a202-d3a6557597bd",
10             "quantity": 3
11         },
12         {
13             "productId": "331c53c2-f645-4a3e-ae70-ed6dedc76e5e",
14             "quantity": 1
15         }
16     ]
17 }
```

Рисунок 5.6 – Тіло запиту для створення замовлення

RabbitMQ у системі допомагає обробляти завдання, які не потрібно виконувати одразу, наприклад, оновлення кількості товарів на складі. Для цього використовується спеціальний "розподільник" під назвою inventory.exchange, який направ-

ляє повідомлення в потрібні черги. Є три типи повідомлень: для надходження нових товарів (new_stock), для резервування товару під замовлення (reserved_stock) і для повернення зарезервованого товару, якщо замовлення скасували (restored_stock). Наприклад, коли на склад надійшла нова партія товару, система через метод sendStockUpdate надсилає повідомлення в inventory.exchange із позначкою new_stock, вказавши ідентифікатор товару і кількість. Це повідомлення потрапляє в чергу inventory.queue.new_stock, де його "ловить" слухач у класі InventoryConsumer. Слухач витягує з повідомлення ідентифікатор товару і кількість, а потім викликає команду, щоб додати ці товари на склад. Так само працюють резервування і скасування резервування: слухачі в чергах inventory.queue.reserved_stock і inventory.queue.restored_stock отримують повідомлення і оновлюють кількість товарів на складі. На рисунку 5.7 показано, як виглядає інтерфейс RabbitMQ із чергами оновлення залишків. Налаштовування проводилося на основі роботи [11].

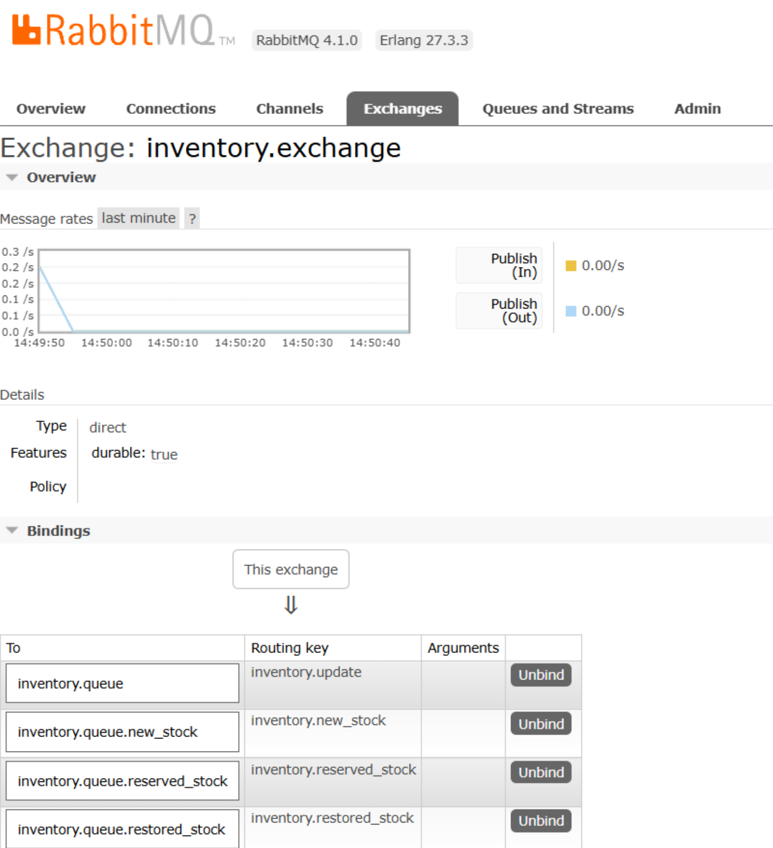

Рисунок 5.7 – Інтерфейс RabbitMQ із чергами Inventory Service

5.1.3 Архітектура баз даних

Оскільки кожен мікросервіс має власну базу даних, таблиці різних сервісів не мають прямих зв'язків, а обмін інформацією між ними здійснюється через REST API і RabbitMQ. А от у межах одного мікросервісу таблиці можуть бути пов'язані між собою. Наприклад, у базі даних Order Service таблиця Orders пов'язана з таблицею OrderItems через зв'язок "один-до-багатьох": одне замовлення може містити кілька позицій, де OrderItems посилається на Orders через зовнішній ключ order_id. Цей зв'язок дозволяє зберігати деталі замовлення (наприклад, product_id, quantity) у межах одного сервісу, а інформація про продукти (з Catalog Service) отримується через REST API (GET /management/product/find-product-by-id). Поля created_at і updated_at у цих таблицях відображають час створення замовлення та оновлення його статусу, що корисно для моніторингу обробки замовлень. Приклад наповнення users наведений на рисунку 5.8.

	id	first_name	last_name	email	address	phone_number	is_active	created_at	last_login	role
1	b3d6b94a-7562-4c15-94bb-aa8623386bcb	Andrew	Khrushch	admin@gmail.com	10 Kyivska st.	+380967933134	true	2025-05-04 06:25:02.996627	2025-05-12 11:49:34.229970	ADMIN
2	c6b9b1ad-300b-471d-8898-b79efd6daba8	Maria	Petrova	product_manager1@gmail.com	11 Kyivska st.	+380967933135	true	2025-05-04 06:25:03.113432	2025-05-04 06:25:03.113432	PRODUCT_MANAGER
3	6a40bb79-5f22-4900-906a-86d2e840eb42	John	Doe	product_manager2@gmail.com	12 Kyivska st.	+380967933136	true	2025-05-04 06:25:03.172667	2025-05-04 06:25:03.172667	PRODUCT_MANAGER
4	bb90285e-5df3-4d86-be43-8ee52354f94d	User_456	Test_31	user_980494826@gmail.com	70 Kyivska st.	+380961949753	true	2025-05-04 06:46:14.962120	2025-05-04 06:46:27.152039	USER
5	c5f084c2-74f8-4415-8cff-9e8ab1ffb8ab	User_247	Test_995	user_379296623@gmail.com	62 Kyivska st.	+380963206493	true	2025-05-04 06:45:12.230838	2025-05-04 06:45:13.279992	USER
6	ff9fa432-30e6-49bf-891d-e9cfe24b6eb5	User_24	Test_414	user_962591068@gmail.com	68 Kyivska st.	+380965601527	true	2025-05-04 06:45:12.460445	2025-05-04 06:45:13.509030	USER
7	efb1edc7-1afa-48be-864b-b895ca454228	User_293	Test_575	user_735591988@gmail.com	80 Kyivska st.	+380963669359	true	2025-05-04 06:45:12.108131	2025-05-04 06:45:13.161330	USER
8	aaa2711f-51f2-42de-b546-c02f2ab70665	User_13	Test_688	user_957037319@gmail.com	55 Kyivska st.	+380965405702	true	2025-05-04 06:45:12.048759	2025-05-04 06:45:13.102328	USER
9	639f51ca-a0f0-438c-be96-3b893cca37fe	User_825	Test_791	user_791871542@gmail.com	83 Kyivska st.	+380964704799	true	2025-05-04 06:45:12.747765	2025-05-04 06:45:13.795850	USER
10	ed222b80-b9a2-45ea-909a-c494c35a7436	User_100	Test_743	user_218505139@gmail.com	73 Kyivska st.	+380966012127	true	2025-05-04 06:45:12.165645	2025-05-04 06:45:13.218185	USER
11	60ee1f24-10c1-4571-8256-a0bea09c313c	User_511	Test_172	user_827912406@gmail.com	4 Kyivska st.	+380962501781	true	2025-05-04 06:45:12.990948	2025-05-04 06:45:14.062881	USER
12	5cd063f2-abcb-4a0c-bea0-943edc96fb9f	User_240	Test_780	user_632449391@gmail.com	39 Kyivska st.	+380964675674	true	2025-05-04 06:45:12.279325	2025-05-04 06:45:13.334016	USER
13	755238cd-af8a-4487-82ef-a1e560fa2228	User_943	Test_483	user_475867972@gmail.com	19 Kyivska st.	+380962784957	true	2025-05-04 06:45:12.331291	2025-05-04 06:45:13.386072	USER
14	62357091-5109-4ec3-814a-7a984581385d	User_231	Test_826	user_386118796@gmail.com	87 Kyivska st.	+380967716587	true	2025-05-04 06:45:12.388199	2025-05-04 06:45:13.444598	USER
15	983db09c-3680-4206-b047-57e9c83571a7	User_52	Test_178	user_449379241@gmail.com	85 Kyivska st.	+380963976171	true	2025-05-04 06:45:12.569528	2025-05-04 06:45:13.615480	USER
16	8a96a3d6-3659-4760-8975-fe199d6ead79	User_62	Test_826	user_238514111@gmail.com	94 Kyivska st.	+380964527097	true	2025-05-04 06:45:13.303757	2025-05-04 06:45:14.397834	USER
17	7ef7aaf3-9101-4d07-986c-4471f28677ca	User_482	Test_603	user_645220825@gmail.com	79 Kyivska st.	+380963053120	true	2025-05-04 06:45:12.630542	2025-05-04 06:45:13.682835	USER
18	f013493e-4462-44df-961e-ba5df28c17c2	User_526	Test_768	user_644401380@gmail.com	20 Kyivska st.	+380967534831	true	2025-05-04 06:45:12.689178	2025-05-04 06:45:13.737970	USER
19	dab6d0a9-046f-460a-abd9-01199473bfba	User_11	Test_586	user_802993015@gmail.com	1 Kyivska st.	+380966478016	true	2025-05-04 06:45:13.509891	2025-05-04 06:45:14.580157	USER
20	69268805-bcb3-4c3e-84c9-eee73f3d3294	User_203	Test_864	user_181202801@gmail.com	90 Kyivska st.	+380962836059	true	2025-05-04 06:45:12.809498	2025-05-04 06:45:13.857078	USER

Рисунок 5.8 – Наповнення таблиці users тестовими користувачами

User Service має таблицю Users, яка частково дублює дані з внутрішньої бази Keycloak, але з різним призначенням: Keycloak зберігає шифровані паролі та ідентифікатори для автентифікації, тоді як Users у User Service містить контактну інформацію, таку як електронна пошта, номер телефону та адресу доставки. Синхронізація між цими таблицями (наприклад, user_id) відбувається через REST API, що викликається після автентифікації в Keycloak, забезпечуючи ізоляцію чутливих даних, таких як паролі.

База Catalog Service має таблицю Products, яка не пов'язана напряму з таблицею Inventory в Inventory Service. Натомість зв'язок між продуктами та запасами реалізується через обмін даними. Асинхронне оновлення запасів (наприклад, резервування товару) відбувається через RabbitMQ, де Order Service надсилає повідомлення до черги "inventory-reserve". Поля created_at і updated_at у таблиці Products дозволяють відстежувати час додавання товару, а в Inventory — час оновлення запасів.

База Delivery Service містить таблицю Deliveries, яка не має прямого зв'язку з таблицею Orders у Order Service. Замість цього інформація про замовлення передається через REST API, а оновлення статусу доставки (наприклад, PENDING, DELIVERED) надсилається асинхронно через RabbitMQ до черги "order-updates", що дозволяє Order Service оновити статус замовлення.

Креслення ІС13.290БАК.005 Д4 ілюструє схему, яка відображає взаємозв'язки між таблицями в окремих базах даних PostgreSQL для різних мікросервісів з DTO. У подальшій розробці таблиці будуть тісно інтегровані з Spring JPA, що забезпечить автоматичне відображення структури бази даних на основі коду сутностей. Кожна сутність, визначена в Java-класах (наприклад, @Entity для Users, Products тощо), відповідатиме таблиці в PostgreSQL, а анотації, такі як @Id, @ManyToOne і @OneToMany, будуть управляти зв'язками та оновленнями. Це дозволить динамічно адаптувати схему бази даних до змін у коді, наприклад, при додаванні нових полів чи зв'язків, що спрощує розробку та підтримку системи.

5.2 Реалізація прототипу системи

5.2.1 Використання Java Spring і Docker для контейнеризації

Розробка системи базується на фреймворку Java Spring, який надає зручні інструменти для створення мікросервісів та розгорнуту документацію наведену за посиланнями [12] та [13]. У проєкті використано низку бібліотек, перелічених у файлі pom.xml і керованих через Maven, що централізує управління залежностями та автоматизує збірку JAR-файлів для всіх сервісів. Spring Boot Starter Web став

основою для створення REST API, дозволяючи швидко налаштувати контролери та обробники запитів. Spring Cloud Starter Netflix Eureka Client інтегрував сервіси з Eureka Server, забезпечуючи їхню реєстрацію та виявлення, що значно спрощує координацію між Auth Service, Order Service та іншими компонентами.

Jakarta Validation API допомагає перевіряти вхідні дані. Spring Boot Starter Actuator додає інструменти моніторингу, які використовуються для перевірки стану контейнерів. Spring Boot Starter Data JPA спростив роботу з базами даних PostgreSQL, дозволяючи створювати репозиторії з анотаціями, такими як @Entity, для усіх таблиць системи. Spring Boot Starter AMQP полегшив інтеграцію з RabbitMQ, дозволяючи налаштовувати черги та слухачів, як описано в підрозділі 5.1.2. Бібліотека PostgreSQL забезпечувала з'єднання з базою даних для зберігання даних, таких як запаси чи замовлення.

Springdoc OpenAPI Starter Webmvc UI автоматично генерувала документацію API, що прискорило тестування та інтеграцію. Lombok, позначений як provided, скоротив код, генеруючи геттери, сеттери та конструктори, що зробило розробку чистішою та менш трудомісткою. Завдяки Maven усі ці залежності інтегрувалися централізовано, що гарантувало однакову конфігурацію для всіх мікросервісів і спростило процес розробки, фрагмент конфігурації представлено на рисунку 5.9.

```
12      <dependencies> Add Starters...
13          <dependency>
14              <groupId>org.springframework.boot</groupId>
15              <artifactId>spring-boot-starter-web</artifactId>
16          </dependency>
17          <dependency>
18              <groupId>org.springdoc</groupId>
19              <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
20          </dependency>
21          <dependency>
22              <groupId>org.springframework.boot</groupId>
23              <artifactId>spring-boot-starter-actuator</artifactId>
24          </dependency>
25          <dependency>
26              <groupId>org.springframework.boot</groupId>
27              <artifactId>spring-boot-starter-validation</artifactId>
28          </dependency>
```

Рисунок 5.9 – Підключення бібліотек до мікросервісу через pom.xml

Для контейнеризації мікросервісів використано Docker, що дозволяє ізолювати кожен сервіс у власному контейнері та спрощує розгортання. Розробка ґрунтувалася на офіційній документації [14]. Кожен модуль упаковано в Docker-образ, створений на основі Dockerfile, який визначає залежності та конфігурацію. На рисунку 5.10 представлено даний файл для збирання кожної частини системи.

Рисунок 5.10 – Вміст Dockerfile для Order Service

Цей Dockerfile створює образ із Java 21, копіює скомпільований JAR-файл і відкриває порт 8083 для взаємодії. Аналогічні образи створено для інших мікросервісів, а також для допоміжних компонентів, таких як PostgreSQL, RabbitMQ і Keycloak.

Для координації контейнерів використано Docker Compose, який визначає залежності між сервісами в єдиному файлі docker-compose.yml. Приклад файлу наведений на рисунку 5.11. Змінна середовища EUREKA_CLIENT_SERVICEURL_DEFAULTZONE налаштовує клієнта для реєстрації сервісу в Eureka Server за адресою http://eureka-server:8761/eureka/. Залежності від Eureka Server, Keycloak і User Service забезпечують коректний порядок запуску: Auth Service стартує лише після того, як ці сервіси стануть "здоровими". Для перевірки стану сервісу налаштовано healthcheck, який періодично звертається до ендпоінту Spring Actuator /actuator/health, що дозволяє Docker контролювати стабільність роботи.

```yaml
services:
  # Authentication Service
  auth-service:
    build:
      context: ./auth-service
      dockerfile: Dockerfile
    ports:
      - "8087:8087"
    container_name: auth-service
    environment:
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://eureka-server:8761/eureka/
    depends_on:
      eureka-server:
        condition: service_healthy
      keycloak:
        condition: service_healthy
      user-service:
        condition: service_healthy
    networks:
      - microservices-net
    healthcheck:
      test: [ "CMD", "wget", "--spider", "-q", "http://auth-service:8087/actuator/health" ]
      interval: 10s
      timeout: 5s
      retries: 3
```

Рисунок 5.11 – Фрагмент docker-compose.yml файлу для Auth Service

У майбутньому планується використання Kubernetes для оркестрації контейнерів у продакшені, що забезпечить автоматичне масштабування та відновлення. Завдяки Kubernetes система зможе автоматично додавати нові контейнери під час зростання навантаження та відновлювати їх у разі збоїв, підвищуючи надійність і доступність.

5.2.2 Розробка внутрішньої структури мікросервісу

Система побудована за багатошаровою архітектурою, що включає шари контролера, сервісу та репозиторію, кожен із яких виконує свою роль. На рисунку 5.12 представлено архітектуру REST API, яка відображає багатошарову структуру сервісу та взаємодію між внутрішніми компонентами. Ця структура прийнята для REST API, вона гарантує чіткий розподіл відповідальності кожного відокремленого шару.

Рисунок 5.12 – Схема архітектури REST API сервісу

Контролер, відомий як Presentation Layer, є першим шаром, який взаємодіє з зовнішнім світом, обробляючи HTTP-запити від фронтенду чи API Gateway. Він приймає вхідні дані, наприклад, JSON із інформацією про замовлення, і передає їх до сервісного шару для подальшої обробки. У Order Service контролер реалізовано через клас OrderController, де метод для створення замовлення (POST /order/create) викликає відповідний сервіс, формуючи HTTP-відповідь із результатом, наприклад статусом 201 Created для коректної відповіді. Цей шар також уміє обробляти помилки, передаючи їх через глобальний обробник, що підвищує надійність взаємодії з користувачами та іншими сервісами. Фрагмент контролеру показано на рисунку 5.13, тут з заголовків отримується email користувача, таким чином замовлення створюється конкретно від нього.

```
@PostMapping("/create")  Saimone2 *
public ResponseEntity<Order> createOrder(
        @RequestHeader("X-User-Email") String email, @Valid @RequestBody OrderRequest request) {
    Order order = orderService.createOrder(email, request);
    return ResponseEntity.status(HttpStatus.CREATED).body(order);
}
```

Рисунок 5.13 – Фрагмент коду класу OrderController

Сервісний шар, або Service Layer, є серцем мікросервісу, де реалізується бізнес-логіка та координація між компонентами. Він приймає дані від контролера, виконує необхідні обчислення та взаємодіє з іншими шарами чи зовнішніми сервісами. Наприклад, у OrderServiceImpl метод для отримання замовлень користувача звертається до User Service через RestTemplate, використовуючи електронну пошту для отримання ідентифікатора користувача, а потім шукає відповідні замовлення в локальній базі PostgreSQL. Цей шар також надсилає асинхронні повідомлення через RabbitMQ, наприклад, для оновлення запасів у Inventory Service, що забезпечує гнучкість і розподілену обробку. Метод сервісного шару проілюстровано на рисунку 5.14.

```
@Override  1 usage  Saimone2
public List<OrderResponse> getAuthenticatedUserOrders(String email) {
    UserResponse userResponse = fetchUserInfoByEmail(email);

    List<Order> orders = getOrdersByUserId(userResponse.getId());
    return orders.stream() Stream<Order>
            .map(this::mapOrderToResponse) Stream<OrderResponse>
            .collect(Collectors.toList());
}
```

Рисунок 5.14 – Фрагмент коду класу OrderServiceImpl

Репозиторій, або Repository Layer, відповідає за доступ до бази даних і виконує операції читання та запису даних, використовуючи Spring Data JPA. Цей шар абстрагує складність SQL-запитів, дозволяючи розробникам працювати з об’єктами Java, такими як сутності Order чи Product, через анотації, наприклад, @Repository. У OrderRepository метод пошуку замовлень за ідентифікатором користувача спрощує доступ до даних, оптимізуючи продуктивність. Для передачі даних між шарами та сервісами застосовуються об’єкти DTO, які містять лише необхідні поля, наприклад, OrderRequest для створення замовлення (з товарами та адресою) чи Order для повернення результатів, що зменшує навантаження і покращує безпеку. Клас репозиторію проілюстровано на рисунку 5.15.

```
@Repository
public interface OrderRepository extends JpaRepository<Order, UUID> {
    Optional<List<Order>> findByUserId(UUID customerId);
}
```

Рисунок 5.15 – Фрагмент коду класу OrderRepository

Використання DTO дозволяє приховувати поля, які не потрібні користувачу, але важливі для внутрішнього функціонування системи. Найкращі практики використання DTO були взяті зі статті [15]. На рисунку 5.16 порівнюється відповідь системи, сформована для фронтенду, із сутністю для бази даних: у відповіді передаються лише ключові дані, такі як назва товару, його характеристики, ціна, доступна кількість, тоді як поля для адміністрування, created_at, updated_at та is_active залишаються прихованими. Це підвищує безпеку, зменшує обсяг переданих даних і спрощує взаємодію з користувачем.

```
@Entity
@Table(name = "products")
@Data
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;
    @NotBlank
    private String name;

    @Column(length = 1000)
    private String description;

    @Min(0)
    private double price;

    @Min(0)
    private int stockQuantity;

    private Boolean isActive = true;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
}
```

```
import lombok.Data;

@Data
public class ProductResponse {
    private String name;
    private String description;
    private double price;
    private int quantity;
}
```

Рисунок 5.16 – Порівняння сутності, що зберігається в БД, та відповіді системи

Висновки до розділу 5

Проєктування архітектури системи на основі 7 мікросервісів завершилось створенням розподіленої структури, яка відповідає функціональним вимогам обраної моделі. Використання принципів єдиної відповідальності та ізоляції помилок,

підкріплене інтеграцією з RabbitMQ для асинхронної взаємодії та REST API для синхронного обміну.

Реалізація прототипу з використанням Java Spring, Docker і Spring Data JPA продемонструвала успішне впровадження багатошарової архітектури, що включає контролери, сервісний шар і репозиторії, а також оптимізацію обміну даними через DTO. Автоматизація контейнеризації та координація через Docker Compose забезпечують високу доступність і готовність системи до майбутнього масштабування. Це створює передумови для подальшого тестування та вдосконалення, що буде детально розглянуто в наступних розділах роботи.

Готовий проєкт розміщено на акаунті GitHub у публічному доступі. До репозиторію створено QR-код, який можна відсканувати у додатку А. Документація проєкту оформлена відповідно до стандартів, з чіткою структурою та описом функціональності. Код структуровано, прокоментовано та відповідає принципам чистого коду для зручності подальшого використання.

6 МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

6.1 Постановка задачі

6.1.1 Змістовна постановка задачі

Інформаційна система управління замовленнями в електронній комерції уже реалізована, і на поточному етапі її необхідно адаптувати для забезпечення комфортної взаємодії з користувачами. Хоча система загалом характеризується високою швидкістю обробки запитів, проведений аналіз виявив основне вузьке місце, пов'язане з операцією створення замовлення через ендпоінт "/order/create". Ця операція ініціює послідовність внутрішніх запитів до інших сервісів, зокрема для перевірки даних користувача, отримання інформації про товари, оновлення даних про запаси на складі та організації доставки на завершальному етапі. Через таку каскадну структуру обробки час виконання запиту на створення замовлення є найбільшим серед усіх операцій у системі, що обґрунтовує необхідність спрямування зусиль на його оптимізацію.

Аналіз функціонування системи виявив, що ключовим вузьким місцем є Catalog Service, де час обробки залежить від кількості унікальних товарів k у замовленні через алгоритмічну складність порядку $O(n)$, де $n = k$. Ця залежність зумовлена циклічним виконанням k підзапитів "/management/product/find-product-by-id" для кожного товару, що призводить до значного накопичення затримок, особливо при великих значеннях k, коли послідовна обробка стає критичною.

Система характеризується такими технічними параметрами, основний екземпляр Catalog Service обладнано буфером об'ємом 800 запитів, і резервний екземпляр має аналогічний буфер на 800 запитів, який слугує додатковим резервом у разі перевантаження. Резервний екземпляр за замовчуванням перебуває в неактивному стані для зменшення експлуатаційних витрат і активується лише за умови перевантаження основного екземпляра, коли буфер заповнений або час очікування перевищує встановлені межі.

Метою дослідження є мінімізація середнього часу очікування W до цільового значення 5000 мс (5 секунд) у пікових сценаріях роботи системи, де інтенсивність

вхідного потоку λ досягає 50 запитів за секунду, а кількість унікальних товарів k — 8. Для досягнення цієї мети необхідно провести аналіз впливу ключових характеристик системи (λ, k, α, m) на час очікування W, а також дослідити взаємозалежності між цими параметрами. Результатом має стати ідентифікація основної проблеми, що найбільше впливає на продуктивність, та розробка рекомендацій щодо оптимального налаштування параметрів системи для подальшої оптимізації.

6.1.2 Математична постановка задачі

Нижче наведено змінні математичної постановки задачі:

– λ, інтенсивність вхідного потоку запитів, $0 \le \lambda \le 50$ (запитів/с);

– k, кількість унікальних товарів у замовленні, $1 \le k \le 8$ (шт);

– m, кількість екземплярів Catalog Service, m = 1 (лише основний) або m = 2 (основний працює паралельно з резервним);

– α, час обробки одного запиту в Catalog Service, α = 0,015 (с);

– B, місткість буфера основного та резервного екземпляра Catalog Service, B = 800 (од. запитів в черзі);

– W, середній час очікування обробки запиту, цільове значення $W \le 5000$ (мс).

Вхідний потік запитів із інтенсивністю λ, який визначається через інтервали між прибуттям запитів, що мають експоненціальний розподіл. Середній інтервал між запитами становить 0.033 секунди, що дає інтенсивність, яка обраховується за формулою (6.1)

$$\lambda = \frac{1}{0.033} \approx 30 \frac{\text{запитів}}{\text{с}}. \quad (6.1)$$

Час обробки запиту в Catalog Service представлений формулою (6.2), він залежить від k і розподіляється між m екземплярами

$$T_{\text{обр}} = k * \frac{\alpha}{m}. \quad (6.2)$$

Загальний час очікування W визначається як сума часу обробки всіх сервісів із урахуванням черг і поданий формулою (6.3)

$$W = W_{order} + W_{user} + W_{cat} + W_{del}, \quad (6.3)$$

де W_{order} – час очікування на етапі Order Service, W_{user} – час очікування на етапі User Service, W_{cat} – час очікування на етапі Catalog Service, W_{del} – час очікування на етапі Delivery Service.

Отже, цільова функція представлена формулою (6.4)

$$W \rightarrow min. \quad (6.4)$$

6.2 Обґрунтування методу розв’язання

Для вирішення задачі мінімізації часу очікування W у системі можна застосувати кілька методів:

– аналітичне моделювання за допомогою систем масового обслуговування (СМО);

– імітаційне моделювання в середовищі Simulink;

– методи машинного навчання (наприклад, регресія, нейронні мережі);

– евристичні методи (наприклад, генетичні алгоритми).

Аналітичне моделювання за допомогою СМО базується на теорії масового обслуговування, де система моделюється як черга M / M / m. Перевага даного методу — можливість отримати точні формули для W, але метод обмежений припущеннями, що не завжди відповідає реальним системам із залежністю O(k).

Імітаційне моделювання дозволяє моделювати систему з урахуванням реальних характеристик (послідовність сервісів, буфери, залежність від k) шляхом створення імітаційної моделі. Перевагами методу є гнучкість, можливість експериментального аналізу залежностей між λ, k, m, α, але недоліком можна врахувати потребу значних обчислювальних ресурсів для точних результатів.

Методи машинного навчання використовуються для прогнозування часу очікування на основі історичних даних і оптимізації параметрів. Перевага полягає у високій точності прогнозів, якщо доступна достатня кількість даних. Недолік же полягає в тому, що необхідна велика вибірка даних, яка не завжди доступна на етапі дослідження.

Евристичні методи застосовуються для пошуку оптимальних значень кількості екземплярів і часу обробки, однак вони потребують значного часу на обчислення і не гарантують досягнення глобального оптимуму.

Методи машинного навчання відхилені через відсутність достатньої кількості історичних даних, тоді як евристичні методи відкинуто, оскільки їхня реалізація є складною та характеризується високою обчислювальною вартістю для даної задачі. Для вирішення поставленої задачі обрано метод імітаційного моделювання в Simulink. Вибір обґрунтовується такими причинами:

– система має складну структуру (послідовна обробка запитів п’ятьма сервісами, залежність від k, буфери, активація резервного екземпляра), що ускладнює аналітичне моделювання систем масового обслуговування (СМО) через нелінійні залежності та нестаціонарність у пікових сценаріях;

– імітаційне моделювання дає змогу експериментально дослідити вплив ключових параметрів (λ, k, m, α) та їхньої взаємодії на W, що відповідає меті задачі — виявити основну проблему для оптимізації;

– середовище Simulink забезпечує зручний інструментарій для моделювання вхідних потоків (λ = 30,3 запитів/с), реалізації буферів (місткість 800 запитів) і налаштування умов активації резервного екземпляра;

– дозволяє легко інтегрувати реальні дані та проводити повторні експерименти з різними конфігураціями, що значно прискорює процес оптимізації системи.

6.3 Опис методу розв’язання

Час очікування W у системі залежить від кількох ключових параметрів, які потрібно дослідити:

– інтенсивність вхідного потоку λ, що визначається експоненціальним розподілом зі значенням λ = 30 запитів/с (з можливим зростанням до пікових λ = 50 запитів/с), збільшення якої підвищує навантаження на сервіси та може спричинити переповнення буферів, що впливає на W;

– кількість унікальних продуктів k, яка визначає кількість підзапитів у Catalog Service зі складністю O(k), де час обробки T пропорційно збільшує час очікування W_{cat} і, відповідно, загальний час W;

– кількість екземплярів Catalog Service m, яка впливає на розподіл навантаження, де m = 1 відповідає лише основному екземпляру, а m = 2 включає резервний екземпляр, що активується при перевантаженні основного, зменшуючи час очікування W шляхом паралельної обробки запитів;

– час обробки одного запиту в Catalog Service α, який становить 0.015 с і визначає швидкість обробки запитів у цьому сервісі, причому збільшення α пропорційно підвищує час очікування W_{cat}, впливаючи на загальний час W.

Усі подальші експерименти проводяться в Simulink протягом 60 од. часу, що еквівалентно 1 хвилині у реальній системі. Повна схема моделювання процесу створення замовлення наведена на кресленику ІС13.290БАК.005 Д2. У першому експерименті досліджується вплив інтенсивності вхідного потоку λ та кількості унікальних продуктів k на час очікування W у системі. Мета цього аналізу — оцінити, як одночасне зростання навантаження та складності обробки замовлень впливає на продуктивність. Аналіз цих даних дозволить запропонувати рекомендації щодо оптимізації конфігурації системи. А результати допоможуть зрозуміти критичні комбінації λ та k, які можуть призводити до перевантаження системи, та визначити межі її стабільної роботи. Усі 4 прогони системи з різними параметрами для експерименту 1 наведено в таблиці 6.1.

Таблиця 6.1 – Результати вимірювання W залежно від λ та k для експерименту 1

№	Інтенсивність вхідного потоку λ, (запитів/с)	Кількість унікальних продуктів k, (шт)	Час очікування W, (мс)

1	30	2	388.1
2	50	2	5383
3	30	8	22070
4	50	8	26050

Для проведення регресійного аналізу перетворимо вхідні змінні X_i у нормовані змінні x_i за формулою (6.5)

$$x_i = \frac{X_i - X_{i0}}{\Delta_i}, \tag{6.5}$$

де $X_{i0} = \frac{X_{i\,max} + X_{i\,min}}{2}$; $\Delta_i = \frac{X_{i\,max} - X_{i\,min}}{2}$.

Результати експерименту відповідають повному факторному плану для двох факторів і представлені формулами (6.6) – (6.7)

$$X_1 = \frac{50+30}{2} = 40; \quad \Delta_1 = \frac{50-30}{2} = 10; \tag{6.6}$$

$$X_2 = \frac{8+2}{2} = 5; \quad \Delta_2 = \frac{8-2}{2} = 3. \tag{6.7}$$

Отже, два фактори продані формулами (6.8)

$$x_1 = \frac{X_1 - 40}{10}, \qquad x_2 = \frac{X_2 - 5}{3}. \tag{6.8}$$

Нижче наведена таблиця 6.2 з матрицею планування повного факторного експерименту.

Таблиця 6.2 – Матриця планування ПФЕ для експерименту 1

2^2	x_0	x_1	x_2	x_1x_2	y

+	+	+	+	26050
+	-	+	-	22070
+	+	-	-	5383
+	-	-	+	388.1

Коефіцієнт рівняння регресії дорівнює добутку стовпця результатів на відповідний стовпець матриці планування, поділеному на кількість експериментів. Коефіцієнт рівняння поданий формулою (6.9)

$$b_k = \frac{\sum_{i=1}^{N} y_i x_{in}}{N}, \qquad n = 0,1,2,3 \tag{6.9}$$

де b_k - коефіцієнт регресії для n змінної, y_i - значення залежної змінної (час очікування в і-ому експерименті), x_{in} — значення n незалежної змінної (наприклад, інтенсивність вхідного потоку або кількість унікальних продуктів) в і-ому експерименті, N — загальна кількість експериментів, n — індекс змінної (0, 1, 2, 3), що відповідає вільному члену та факторам.

За результатами експерименту обраховуємо коефіцієнти використовуючи формулу (6.9) і отримуємо формули (6.10) – (6.13)

$$b_0 = \frac{26050 + 22070 + 5383 + 388.1}{4} = 13472.78 \tag{6.10}$$

$$b_1 = \frac{26050 + (-22070) + 5383 + (-388.1)}{4} = 2243.73 \tag{6.11}$$

$$b_2 = \frac{26050 + 22070 + (-5383) + (-388.1)}{4} = 10587.23 \tag{6.12}$$

$$b_3 = \frac{26050 + (-22070) + (-5383) + 388.1}{4} = -253.73 \tag{6.13}$$

Таким чином, рівняння регресії представлена формулою (6.14)

$$y = 13472.78 + 2243.73x_1 + 10587.23x_2 - 253.73x_1x_2, \tag{6.14}$$

де y - час очікування W у мілісекундах, x_1 - нормалізована змінна для λ, x_2 - нормалізована змінна для k, а x_1x_2 - їхня взаємодія.

Коефіцієнт при факторі x_2 становить 10587.23, і є найбільшим, вказуючи на те, що зі збільшенням кількості унікальних продуктів час очікування зростає найбільш суттєво. Дані показують, що при високих значеннях k (наприклад 8) час очікування різко зростає (до 22070 – 26050 мс), це дуже негативно впливає на систему і не може вважатися оптимальним, оскільки в рази перевищує цільові 5 секунд.

Інтенсивність вхідного потоку λ, представлена коефіцієнтом b_1, має помірний вплив на результат. Це свідчить про те, що зростання навантаження також збільшує час очікування, але його вплив менший порівняно з k, і він є менш визначальним фактором для ефективності системи в даній зв’язці. Проте, для мінімізації перевантаження та оптимізації роботи системи варто продовжувати аналіз цього параметра в комбінації з іншими факторами.

Коефіцієнт взаємодії факторів b_3 при x_1x_2 вказує на слабкий синергічний ефект між λ і k. Вплив є другорядним і не суттєво відбивається на загальний результат порівняно з окремими впливами λ і k.

Другий експеримент спрямований на вивчення впливу інтенсивності вхідного потоку λ та кількості екземплярів Catalog Service m на час очікування W. Основна мета — визначити, як збільшення m (від одного до двох екземплярів) може компенсувати зростання навантаження, викликане підвищенням λ, та уникнути активації резервного екземпляра в критичних ситуаціях. Це дозволить оцінити ефективність горизонтального масштабування для підтримки цільового часу очікування. Результати другого експерименту представлені у таблиці 6.3. Графіки заповнення буферу кожного екземпляру наведено у додатку Б.

Таблиця 6.3 – Результати вимірювання W залежно від λ та m для експерименту 2

№	Інтенсивність вхідного потоку λ, (запитів/с)	Кількість екземплярів Catalog Service *m*, (шт)	Час очікування W, (мс)

1	30	1	8546
2	50	1	16140
3	30	2	3843
4	50	2	7376

Результати експерименту відповідають повному факторному плану для двох факторів та представлені формулами (6.15) – (6.16)

$$X_1 = \frac{50+30}{2} = 40; \quad \Delta_1 = \frac{50-30}{2} = 10; \qquad (6.15)$$

$$X_2 = \frac{2+1}{2} = 1.5; \quad \Delta_2 = \frac{2-1}{2} = 0.5. \qquad (6.16)$$

Для даних характеристик два фактори подані, як формули (6.17)

$$x_1 = \frac{X_1 - 40}{10}, \qquad x_2 = \frac{X_2 - 1.5}{0.5}. \qquad (6.17)$$

Нижче наведена матриця планування ПФЕ для другого експерименту.

Таблиця 6.4 – Матриця планування ПФЕ для експерименту 2

2^2	x_0	x_1	x_2	x_1x_2	y
	+	+	+	+	7376
	+	-	+	-	3843
	+	+	-	-	16140
	+	-	-	+	8546

На основі матриці обчислюються коефієнти рівняння за формулою (6.9) та отримуємо формули (6.18) – (6.21)

$$b_0 = \frac{7376 + 3843 + 16140 + 8546}{4} = 8976.25; \qquad (6.18)$$

$$b_1 = \frac{7376 + (-3843) + 16140 + (-8546)}{4} = 11127; \quad (6.19)$$

$$b_2 = \frac{7376 + 3843 + (-16140) + (-8546)}{4} = -3366.75; \quad (6.20)$$

$$b_3 = \frac{7376 + (-3843) + (-16140) + 8546}{4} = -1015.25. \quad (6.21)$$

Результуюче рівняння регресії для другого експерименту подане формулою (6.22)

$$y = 8976.25 + 11127x_1 - 3366.75x_2 - 1015.25x_1x_2. \quad (6.22)$$

Отже, коефіцієнт при факторі x_1 дорівнює 11127, що є найбільшим за модулем і вказує на значний вплив зростання інтенсивності вхідного потоку на збільшення часу очікування. Високі значення часу свідчать про перевантаження системи, що негативно позначається на її продуктивності та вимагає додаткових заходів для стабілізації.

Кількість екземплярів (фактор x_2) представлена коефіцієнтом b_2 = -3366.75. Збільшення кількості екземплярів, наприклад через активацію резервного, сприяє ефективному зменшенню часу очікування, частково компенсуючи зростання навантаження.

Коефіцієнт взаємодії між факторами дорівнює -1015.25. Одночасне підвищення навантаження та кількості екземплярів забезпечує компенсуючий ефект, дозволяючи системі краще витримувати пікові навантаження, але цього все одно не достатньо, щоб витримувати пікові навантаження на систему.

У третьому експерименті досліджується залежність часу очікування W від часу обробки одного запиту в Catalog Service α та кількості унікальних продуктів k. Мета цього аналізу — оцінити, як зміна часу обробки α у поєднанні зі зростанням k впливає на продуктивність системи. Результати дозволять зрозуміти, наскільки оптимізація часу обробки може зменшити вплив складних замовлень і наблизити W до цільового значення 5 секунд.

Таблиця 6.5 – Результати вимірювання W залежно від α та k для експерименту 3

№	Кількість унікальних продуктів k, (шт)	Час обробки Catalog Service α, (мс)	Час очікування W, (мс)
1	2	0.008	193.4
2	8	0.008	15790
3	2	0.015	388.1
4	8	0.015	25930

Як і для попередніх експериментів необхідно знайти нормовані факторні змінні, а також мартицю ПФЕ, яка представлена в таблиці 6.6, ось так будуть виглядати їх формули (6.23) – (6.25)

$$X_1 = \frac{8+2}{2} = 5; \quad \Delta_1 = \frac{8-2}{2} = 3; \tag{6.23}$$

$$X_2 = \frac{0.015+0.008}{2} = 0.0115; \quad \Delta_2 = \frac{0.015-0.008}{2} = 0.0035; \tag{6.24}$$

$$x_1 = \frac{X_1 - 40}{10}, \qquad x_2 = \frac{X_2 - 0.0115}{0.0035}; \tag{6.25}$$

Таблиця 6.6 – Матриця планування для експерименту 3

2^2	x_0	x_1	x_2	x_1x_2	y
	+	+	+	+	25930
	+	-	+	-	388.1
	+	+	-	-	15790
	+	-	-	+	193.4

Далі знаходяться коефіцієнти і формується рівняння для 3-го експерименту формулами (6.26) – (6.30)

$$b_0 = \frac{25930 + 388.1 + 15790 + 193.4}{4} = 10575.38; \quad (6.26)$$

$$b_1 = \frac{25930 + (-388.1) + 15790 + (-193.4)}{4} = 10284.63; \quad (6.27)$$

$$b_2 = \frac{25930 + 388.1 + (-15790) + (-193.4)}{4} = 2583.68; \quad (6.28)$$

$$b_3 = \frac{25930 + (-388.1) + (-15790) + 193.4}{4} = 2486.33; \quad (6.29)$$

$$y = 10575.38 + 10284.63x_1 + 2583.68x_2 + 2486.33x_1x_2. \quad (6.30)$$

Проведені розрахунки та побудоване рівняння регресії виявили фактори, які впливають на час очікування. Коефіцієнт при факторі x_1 дорівнює 10284.63 і є найбільшим. Таке суттєве підвищення часу очікування свідчить про критичну залежність системи від складності замовлень, приблизно те ж саме можна спостерігати у першому експерименті.

А час обробки одного запиту, представлений коефіцієнтом $b_2 = 2583.68$, чинить помірний вплив на результат. Зростання часу обробки сприяє збільшенню часу очікування, однак його ефект менш виражений порівняно з впливом кількості продуктів і відіграє другорядну роль у визначенні ефективності системи.

Отже, результатами аналізу було встановлено, що основною причиною затримок у системі є Catalog Service, де час очікування найбільше залежить від кількості унікальних товарів у замовленні. Алгоритмічна складність, викликана виконанням підзапитів для кожного товару, значно збільшує час обробки, особливо при великих значеннях, що призводить до перевищення цільового часу очікування в 5 секунд у кілька разів. Інтенсивність вхідного потоку та час обробки також впливають на час очікування, але їхній ефект менший порівняно з кількістю товарів і залишається в межах допустимого. Збільшення кількості екземплярів до 2 частково компенсує навантаження, однак не усуває проблему, пов’язану з алгоритмічною залежністю. Аналіз залежностей між параметрами допоміг ідентифікувати вузьке місце та підтвердив, що оптимізація часу обробки

недостатня для досягнення цільового часу очікування без усунення основної причини.

Висновки до розділу 6

На основі проведеного аналізу було зроблено висновок, що метод імітаційного моделювання за допомогою Simulink ідеально підійшов для знаходження проблеми та рішення для оптимізації роботи системи. В результаті було досліджено декілька змінних характеристик системи і виявлено слабке місце, яке полягає у послідовному виконанні підзапитів у Catalog Service, що можна вдосконалити завдяки впровадженню батчевої передачі ID унікальних товарів одним запитом. Такий підхід усуває залежність від кількості товарів, стабілізуючи час обробки, забезпечує цільовий час очікування до 5 секунд навіть при пікових значеннях інтенсивності вхідного потоку, а також зменшує потребу в другому екземплярі Catalog Service, що оптимізує ресурси та спрощує архітектуру.

7 ТЕСТУВАННЯ СИСТЕМИ

7.1 Мета та планування тестування

Мета тестування полягає у перевірці стійкості мікросервісної програми до навантаження та її здатності коректно обробляти нестандартні запити для забезпечення надійності й стабільності роботи.

Для реалізації поставленої мети планується провести такі типи тестування:

– тестування навантаженням передбачає оцінку поведінки системи під час високого обсягу запитів, імітуючи реальні пікові навантаження за допомогою інструменту k6 із фокусом на аналіз часу відповіді та стабільності роботи;

– тестування коректності запитів спрямоване на перевірку реакції системи на некоректні запити з неправильними даними, неаутентифіковані запити без авторизації та запити з помилками у форматі JSON.

Реалізація тестування передбачає налаштування інструменту k6 для моделювання різних сценаріїв навантаження та типів запитів із визначенням базових і пікових умов роботи. Для відправки запитів, зокрема некоректних, неаутентифікованих і з помилками, використовуватиметься Postman, який дозволяє створювати й надсилати запити з різними параметрами, авторизаційними заголовками та форматами даних, а також аналізувати відповіді сервера для оцінки коректності роботи системи. У системі реалізовано Swagger, що забезпечує автоматично згенеровану документацію API, дозволяючи переглядати доступні ендпоінти, їх параметри та формати відповідей, а також тестувати запити безпосередньо через вебінтерфейс. Swagger також дозволяє моделювати сценарії з граничними значеннями даних, що допомагає виявити потенційні слабкі місця в обробці нестандартних запитів. Тестування проводитимуть у кілька етапів із поступовим нарощуванням інтенсивності, фіксуючи основні показники, такі як час обробки, кількість оброблених запитів і частота помилок. Отримані результати використають для аналізу відповідності системи вимогам і виявлення можливих вразливостей чи напрямів для вдосконалення , а також для оцінки впливу затримок на користувацький досвід, що дозволить підвищити зручність взаємодії з платформою.

7.2 Тестування навантаженням

Інструмент Grafana k6 дозволяє гнучко налаштовувати кількість користувачів, тривалість тесту та інтенсивність запитів, що дало змогу моделювати реальні сценарії навантаження з поступовим нарощуванням інтенсивності. Результати фіксувалися з урахуванням середнього, мінімального, максимального часу відповіді та перцентильних показників, що відображають поведінку системи під навантаженням.

Першим етапом тестування стало моделювання процесів реєстрації та логіну користувача з конфігурацією, що включає три етапи навантаження: початковий із невеликою кількістю користувачів, основний із піковим навантаженням та завершальний із поверненням до спокійного стану. Результати представлено на рисунку 7.1, де видно, що система демонструє стабільну роботу, хоча час відповіді для реєстрації виявляється вищим, ніж для логіну, що може бути пов’язано з більшим обсягом обробки даних. Нюансом є відносно високі максимальні значення, які вказують на можливі пікові затримки (приблизно 1.5 секунди на реєстрацію), але основна маса запитів у межах прийнятного діапазону.

```
█ TOTAL RESULTS

  checks_total.......................: 2832    46.550614/s
  checks_succeeded...................: 100.00% 2832 out of 2832
  checks_failed......................: 0.00%   0 out of 2832

  ✓ registration status is 201
  ✓ login success

  CUSTOM
  login_response_time...............................................: avg=174.613586 min=25.423179 med=114.663186 max=687.919813  p(90)=420.207628  p(95)=497.915199
  registration_response_time........................................: avg=630.12673  min=32.46827  med=531.209904 max=1772.694091 p(90)=1412.926467 p(95)=1527.438908

  HTTP
  http_req_duration.................................................: avg=402.37ms   min=25.42ms   med=187.37ms   max=1.77s       p(90)=1.22s       p(95)=1.41s
    { expected_response:true }......................................: avg=402.37ms   min=25.42ms   med=187.37ms   max=1.77s       p(90)=1.22s       p(95)=1.41s
  http_req_failed...................................................: 0.00%  0 out of 2832
  http_reqs.........................................................: 2832   46.550614/s

  EXECUTION
  iteration_duration................................................: avg=1.8s       min=1.05s     med=1.73s      max=3.02s       p(90)=2.66s       p(95)=2.78s
  iterations........................................................: 1416   23.275307/s
  vus...............................................................: 2      min=1         max=100
  vus_max...........................................................: 100    min=100       max=100

  NETWORK
  data_received.....................................................: 4.3 MB 72 kB/s
  data_sent.........................................................: 744 kB 12 kB/s
```

Рисунок 7.1 – Результати тестування навантаженням реєстрації та логіну

Другим етапом тестування стало моделювання створення замовлення, яке попередньо оптимізовано завдяки батчевій відправці групових запитів до Catalog Service, що скоротило кількість окремих викликів і підвищило ефективність. Аналіз даної проблеми детально описано у розділі 6. Конфігурація тестування передбачала постійне навантаження протягом хвилини з фіксованою кількістю запитів за секунду. Результати відображено на рисунку 7.2, де система показує високу продуктивність із середнім часом відповіді, який відповідає стандартам для бекенду. Максимальні значення як і в попередньому тесті дещо високі, що може свідчити про локальні перевантаження, але p95 залишається в межах оптимального діапазону, це підтверджує стабільність системи.

```
█ TOTAL RESULTS

  checks_total.......................: 3005    50.003187/s
  checks_succeeded...................: 100.00% 3005 out of 3005
  checks_failed......................: 0.00%   0 out of 3005

  ✓ order creation status is 201

  CUSTOM
  order_creation_response_time............................................: avg=16.806154 min=8.074674 med=15.055767 max=115.479098 p(90)=24.851034 p(95)=30.302474

  HTTP
  http_req_duration.......................................................: avg=16.8ms    min=8.07ms   med=15.05ms   max=115.47ms   p(90)=24.85ms   p(95)=30.3ms
    { expected_response:true }............................................: avg=16.8ms    min=8.07ms   med=15.05ms   max=115.47ms   p(90)=24.85ms   p(95)=30.3ms
  http_req_failed.........................................................: 0.00%  0 out of 3005
  http_reqs...............................................................: 3005   50.003187/s

  EXECUTION
  iteration_duration......................................................: avg=99.91ms   min=41.13ms  med=99.86ms   max=185.82ms   p(90)=106.76ms  p(95)=109.23ms
  iterations..............................................................: 3005   50.003187/s
  vus.....................................................................: 5      min=5         max=5
  vus_max.................................................................: 5      min=5         max=5

  NETWORK
  data_received...........................................................: 2.2 MB 36 kB/s
  data_sent...............................................................: 1.2 MB 20 kB/s
```

Рисунок 7.2 – Результати тестування навантаженням створення замовлення

Система впоралася з обома сценаріями навантаження на високому рівні. Тестування реєстрації та логіну засвідчило прийнятну продуктивність навіть при піковому навантаженні, а оптимізоване створення замовлення продемонструвало вражаючу швидкість відповідно сучасним вимогам до вебзастосунків Ці результати підтверджують, що система готова до використання в реальних умовах, забезпечуючи стабільність і швидкодію для користувачів.

7.3 Тестування обробки некоректних запитів

Тестування обробки некоректних запитів виконано для оцінки стійкості системи до помилкових або невідповідних вхідних даних, що є важливим аспектом забезпечення її надійності в реальних умовах експлуатації. Документація API системи представлена у Swagger, де детально описано усі доступні ендпоінти, включаючи їх параметри та можливі відповіді, вигляд представлений на рисунку 7.3. POST та PUT запити потребують передачі тіла запиту, яке має відповідати визначеній логіці, що передбачає обов'язкову валідацію вхідних даних на сервері. Також Swagger дуже зручний своєю подачею відповідей для запитів. Наприклад, запит GET /product/{ID}, який повертає інформацію про продукт за ідентифікатором, у Swagger описує різні сценарії відповідей, від успішного повернення даних до помилок. Коректну відповідь проілюстровано на рисунку 7.4.

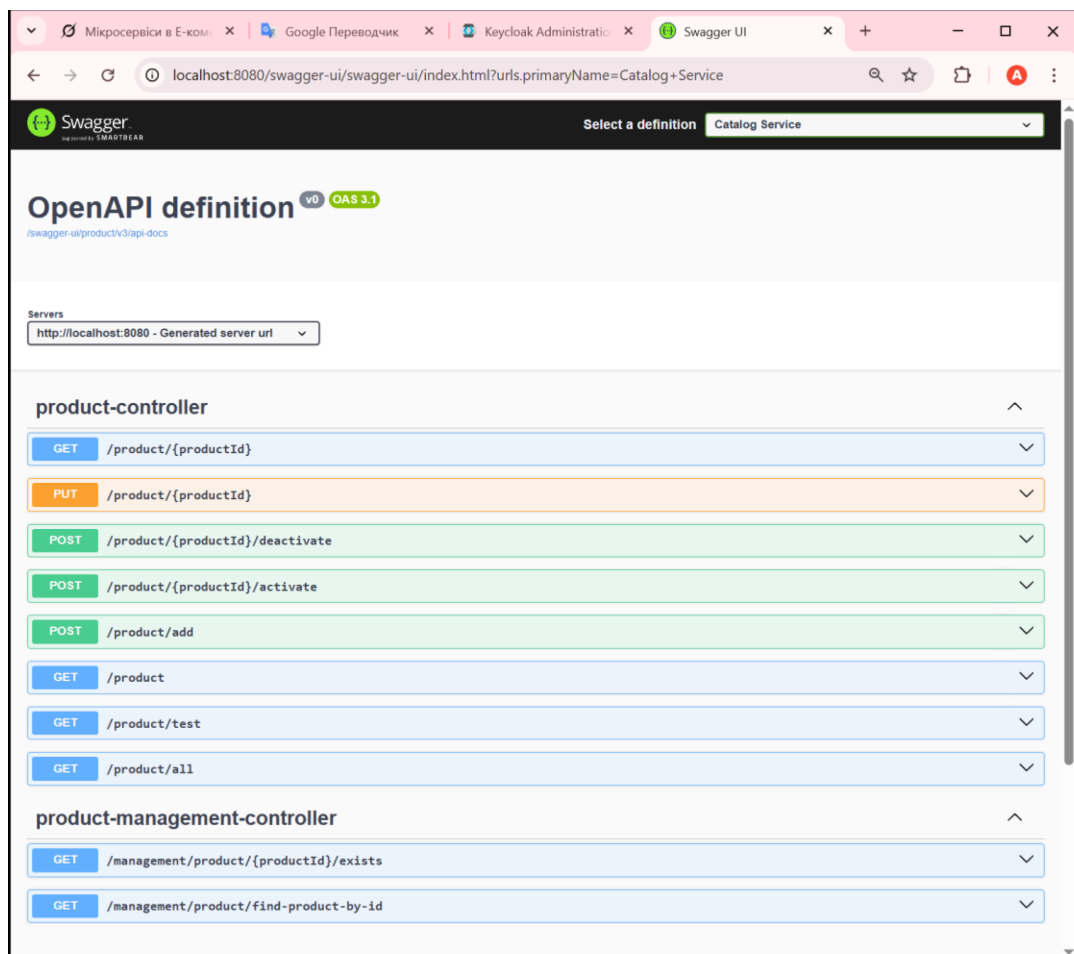

Рисунок 7.3 – Вигляд документації API у Swagger

Рисунок 7.4 – Коректна відповідь отримання товару за ID

На першому етапі тестування розглянуто запит на створення замовлення, де коректно сформований запит із валідними даними був успішно оброблений системою, що підтверджує правильність реалізації ендпоінта, як на рисунку 7.5. Це демонструє, що система здатна коректно обробляти стандартні сценарії використання, коли усі параметри запиту відповідають вимогам API.

Рисунок 7.5 – Коректна відповідь запиту створення замовлення

Далі протестовано сценарій неавторизованого доступу до цього ж ендпоінта, коли запит виконувався без передачі токена авторизації, що призвело до повернення помилки 401, що проілюстровано на рисунку 7.6. Така поведінка є очікуваною, оскільки API захищено від несанкціонованого доступу на етапі проходження через API Gateway, а механізм автентифікації працює коректно за допомогою Keycloak, який видає токени доступу.

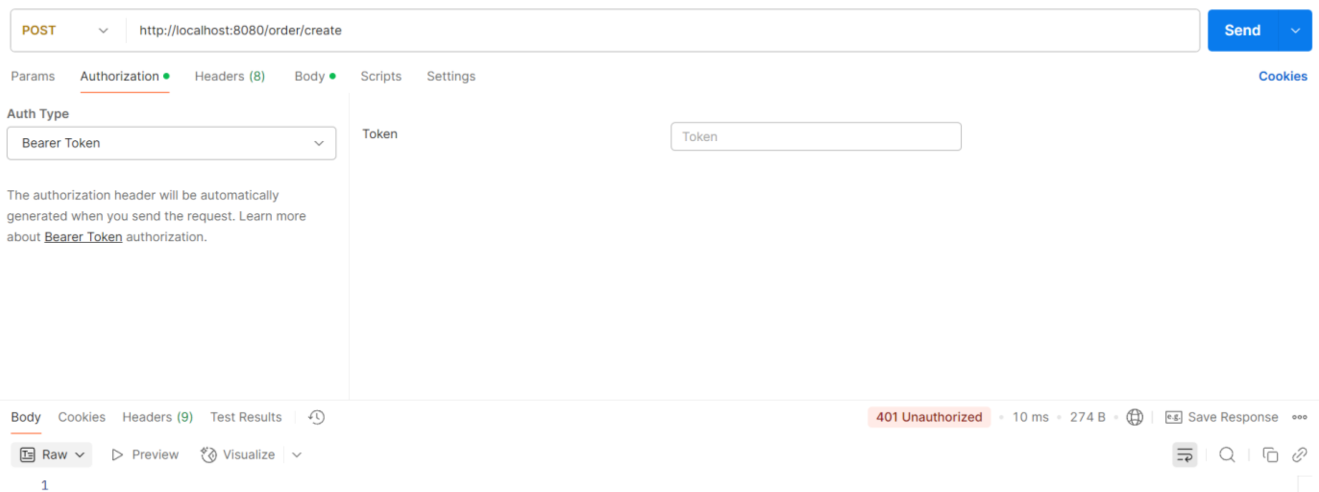

Рисунок 7.6 – Неавторизований доступ до створення замовлення

Наступним етапом стала перевірка сценарію, коли запит на створення замовлення містив кількість товарів, яка перевищує доступний запас, що спричинило помилку 409 Conflict, як на рисунку 7.7. Це підтверджує правильну роботу валідації запасів на стороні сервера, яка запобігає некоректним операціям із товарами, що відсутні в достатній кількості. Система продемонструвала надійність у обробці некоректного запиту, повернувши відповідь з кодом помилки та поясненням "Not enough stock for product", що відповідає специфікації API та забезпечує зручність для клієнтів у розумінні причин відмови. Такий підхід дозволяє уникнути непорозумінь між системою та користувачем. Чітка обробка помилок сприяє підвищенню довіри до системи, адже клієнти можуть швидко виправити запит, спираючись на зрозуміле повідомлення про помилку.

POST http://localhost:8080/order/create

Params Authorization Headers (9) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON


```
{
    "deliveryAddress": "123 Main St",
    "items": [
        {
            "productId": "ad8af56e-f70b-46f1-91a6-a205c10e1369",
            "quantity": 9
        },
        {
            "productId": "54c812f7-7a40-44c7-8099-8340f528c9a4",
            "quantity": 13
        }
    ]
}
```


Body Cookies Headers (10) Test Results 409 Conflict

{} JSON Preview Visualize


```
{
    "success": false,
    "message": "Not enough stock for product: ad8af56e-f70b-46f1-91a6-a205c10e1369"
}
```


Рисунок 7.7 – Відповідь про недостатню кількістю залишків товару для створення замовлення

Висновки до розділу 7

У рамках тестування описано не всі можливі варіанти некоректних запитів, наприклад, від’ємна кількість товарів чи інші нестандартні сценарії, однак на кінцевому етапі розробки усі ендпоінти було протестовано для підготовки системи до розгортання в продакшн, що забезпечило її готовність до реальної експлуатації.

Розділ тестування охопив ключові аспекти перевірки функціональності, продуктивності та стійкості. Тестування навантаженням засвідчило високу швидкість обробки запитів і стабільність навіть при пікових навантаженнях, особливо після оптимізації створення замовлення. Перевірка обробки некоректних запитів підтвердила надійність валідації даних і захисту від несанкціонованого доступу, що дозволяє системі коректно реагувати на помилкові сценарії. Загалом система має готовність до використання в реальних умовах.

У висновку, проведене тестування продемонструвало, що система відповідає встановленим нефункційним вимогам щодо продуктивності та безпеки, забезпечуючи надійну роботу в різних сценаріях. Це дозволяє впевнено рекомендувати її для впровадження в реальних умовах електронної комерції.

ВИСНОВКИ

Створення системи управління замовленнями на базі мікросервісів стало важливим кроком, дозволивши реалізувати нове, сучасне та гнучке рішення для сфери електронної комерції. У даному випадку система складається з окремих частин, що забезпечують контроль доступу, реалізацію каталогу товарів, обробку замовлень, облік товарів та управління доставкою, і все складено у рамках загальної архітектури. Кожен мікросервіс має окрему програму, але всі разом вони працюють гармонійно, що дозволяє обробляти велику кількість замовлень з мінімальною затримкою. За підсумками тестів система показала гарну продуктивність, витримуючи пікове навантаження, дозволяючи швидко і легко розміщувати замовлення і оновлювати дані в реальному часі залишаючись стабільною при збоях. Багатошарова структура оптимізує внутрішню роботу, а завдяки використанню DTO приховуються непотрібні для користувача дані.

Система повністю відповідає вимогам як функціональним, так і нефункціональним. Користувачі легко підбирають потрібні товари, оформляють замовлення та стежать за їх виконанням, а адміністратори можуть ефективно керувати асортиментом та логістикою. Крім того, система швидка та безпомилкова на всіх етапах, завдяки чому бізнес може підвищити лояльність клієнтів та оптимізувати витрати. Модульність та здатність до запуску в контейнерах через Docker дозволяють масштабувати систему. Створювати послуги та додавати функції можна навіть без значних змін у структурі. Плани на використання Kubernetes надалі лише наголошують, що система готова до роботи з наростаючими навантаженнями. Вона ідеально підійде великим платформам e-commerce.

Крім електронної комерції, система може знайти застосування і в суміжних галузях. У логістиці — покращувати маршрути доставки та керування складськими запасами, а у виробництві — контролювати постачання матеріалів та відстежувати, чи виконуються замовлення. Гнучкість системи дозволяє адаптувати її як для невеликого бізнесу, який тільки-но почав свою діяльність в інтернеті, так і для великих корпорацій, на кожного з яких в день припадає тисячі замовлень.

Подальший розвиток системи може значно розширити його функціональність та цінність для бізнесу. Інтеграція з платіжними системами, впровадження аналітики на основі штучного інтелекту для прогнозування попиту чи персоналізації пропозицій можуть зробити платформу більш конкурентоспроможною. Система має потенціал стати ядром цифрових екосистем, що поєднують постачальників, клієнтів та логістичні компанії в єдиному інформаційному просторі. Впровадження модулів для автоматизації маркетингових кампаній та аналізу поведінки клієнтів може додатково підвищити її привабливість для бізнесу. Це рішення не тільки оптимізує поточні процеси, а й створює передумови для трансформації бізнес-моделей у напрямку більшої цифровізації та клієнтоорієнтованості. Проект демонструє, як сучасні технології здатні вирішувати реальні бізнес-завдання, сприяючи підвищенню ефективності та створенню інноваційних моделей у цифровій економіці.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Statista. Global retail e-commerce sales forecast 2014–2027. URL: https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales.

2. McKinsey & Company. Think fast: How to accelerate e-commerce growth. URL: https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/think-fast-how-to-accelerate-e-commerce-growth.

3. Ebiesuwa S., Gegeleso B., Falana T., Adegbenjo A., Bamisile O. Impact of Information Systems on Operational Efficiency: A Comprehensive Analysis // Research Gate, 2023. URL: https://www.researchgate.net/publication/373267444_Impact_of_Information_Systems_on_Operational_Efficiency_A_Comprehensive_Analysis.

4. Statista. Global: e-commerce retail sales CAGR 2024–2029. URL: https://www.statista.com/statistics/1252481/social-buyers-worldwide-countries.

5. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 615 p.

6. Netflix Architecture / Bool.dev. URL: https://bool.dev/blog/detail/netflix-architecture.

7. Internet Banking with Microservices Architecture / ITSOL. URL: https://itsol.vn/portfolio/internet-banking-with-microservices-architecture.

8. Edureka. Microservice Architecture. Explore UBER's Microservice Architecture. URL: https://medium.com/edureka/microservice-architecture-5e7f056b90f1.

9. RabbitMQ Team. RabbitMQ Documentation. URL: https://www.rabbitmq.com/documentation.html.

10. Videla A., Williams J. J. W. RabbitMQ in Action: Distributed Messaging for Everyone. Shelter Island: Manning Publications, 2012. 312 p.

11. Richardson C. Microservices Patterns: With examples in Java. Shelter Island : Manning Publications, 2018. 520 p.

12. Spring Team. Spring Boot Documentation. URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle.

13. Spring Team. Spring Data JPA Documentation. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html.

14. Docker Team. Docker Documentation. URL: https://docs.docker.com.

15. Microsoft Azure. RESTful API Design: Best Practices. URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design.

16. Fowler M., Lewis J. Microservices: a definition of this new architectural term. URL: https://martinfowler.com/articles/microservices.html.