МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ “КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ”
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП’ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____Юрій ІСКРЕНКО
“_____” _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ “МАГІСТР”
ЗА СПЕЦІАЛЬНІСТЮ 123 “КОМП'ЮТЕРНА ІНЖЕНЕРІЯ”

Тема: Система управління високонавантаженими розподіленими сервісами на основі подієво-орієнтованої архітектури

Виконавець: Даніал РЕХМАН

Керівник: Ігор ТЕЛЕШКО

Нормоконтролер: Наталія ФОМІНА

Київ 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ “КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ”
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП’ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

ЗАТВЕРДЖУЮ
Завідувач кафедри КСМ
_____ Юрій ІСКРЕНКО
“_____” _____ 2025 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Рехмана Даніала Хабібовича

(прізвище, ім'я, по батькові здобувача вищої освіти в родовому відмінку)

1. Тема кваліфікаційної роботи: Система управління високонавантаженими розподіленими сервісами на основі подієво-орієнтованої архітектури
Затверджена наказом ректора від “ 24 ” жовтня 2025 р. № 2344/ст
2. Термін виконання роботи (проєкту): з 29.09.2025 по 31.12.2025
3. Вихідні дані до роботи (проєкту): *Java*, *Spring Boot*, *Spring Cloud*, *Spring Security*, *PostgreSQL*, *Apache Kafka*, *Docker*, *Prometheus*, *Grafana*.
4. Зміст пояснювальної записки: вступ, теоретичні основи побудови розподілених високонавантажених систем, аналіз предметної області та постановка задачі, проектування системи, реалізація системи, висновки, список використаних джерел.
5. Перелік обов'язкового графічного (ілюстративного) матеріалу: результати роботи представлені у вигляді презентації.

6. Календарний план

№ п/п	Завдання	Термін виконання етапів	Відмітка про виконання
1.	Затвердження теми роботи	29.09.2025	
2.	Вивчення та аналіз принципів роботи високонавантажених та розподілених систем	01.10.2025	
3.	Опрацювання теоретичних матеріалів щодо мікросервісної та подієво-орієнтованої архітектур	07.10.2025	
4.	Постановка задачі та формування вимог до системи	21.10.2025	
5.	Проєктування архітектури системи та взаємодії сервісів	28.10.2025	
6.	Реалізація мікросервісів (*User*, *Product*, *Order*, *Payment*, *Notification*)	04.11.2025	
7.	Проведення навантажувального тестування (*Apache JMeter*)	10.12.2025	
8.	Оформлення пояснювальної записки	14.12.2025	
9.	Підготовка та оформлення презентації	19.12.2025	
10.	Захист кваліфікаційної роботи	23.12.2025-31.12.2025	

7. Консультація з окремого(мих) розділу(ів):

Назва розділу	Консультант (посада, П.І.Б.)	Дата, підпис	
		Завдання видав	Завдання прийняв

8. Дата видачі завдання: «29» вересня 2025 р.

Керівник кваліфікаційної роботи (проєкту): Ігор ТЕЛЕШКО _____

(підпис виконавця)

Завдання прийняв до виконання: Даніал РЕХМАН _____

(підпис виконавця)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи “Система управління високонавантаженими розподіленими сервісами на основі подієво-орієнтованої архітектури”**:** 95 сторінок, 15 рисунків, 3 таблиці, 34 літературних джерела.

Об’єкт дослідження – високонавантажена розподілена програмна система, що функціонує у середовищі мікросервісної та подієво-орієнтованої архітектури.

Предмет дослідження – принципи побудови подієво-орієнтованих мікросервісних систем, методи забезпечення їх масштабованості, надійності, узгодженості даних, а також засоби моніторингу, логування та автоматизації розгортання в контейнеризованих середовищах.

Мета роботи – розробка системи управління високонавантаженими розподіленими сервісами на основі подієво-орієнтованої архітектури, яка забезпечує масштабованість, відмовостійкість та ефективну взаємодію між компонентами.

Технічні та програмні засоби – мова програмування *Java*, фреймворк *Spring Boot, Spring Security, JWT, Spring Data JPA*, брокер повідомлень *Apache Kafka*, система керування базами даних *PostgreSQL*, система контейнеризації *Docker*, система управління проєктами *Maven*, інструменти моніторингу *Prometheus* і *Grafana, Apache JMeter* для навантажувального тестування.

Основні характеристики та показники – система складається з п’яти незалежних мікросервісів (*User, Product, Order, Payment, Notification*), що взаємодіють через подієво-орієнтований механізм на основі *Kafka*; кожен сервіс працює з окремою базою даних; забезпечується горизонтальне масштабування; реалізовано автентифікацію та авторизацію на основі *JWT*; налаштовано агрегований моніторинг метрик, логування сервісів та збір статистики навантаження. Система витримує пікове навантаження більше 10 000 запитів відповідно до сценаріїв *JMeter*.

Отримані результати та їх новизна – реалізовано повноцінну мікросервісну екосистему з подієвою взаємодією, яка демонструє високу продуктивність завдяки використанню неблокуючих асинхронних механізмів обміну подіями. Створено масштабовану архітектуру з розділенням бізнес-логіки, що дозволяє незалежно оновлювати та розгортати сервіси. Налаштовано централізований моніторинг, що забезпечує оперативний аналіз продуктивності та стабільності системи. Новизна полягає в інтеграції сучасних практик розробки з практичним застосуванням під час побудови високонавантаженої екосистеми з урахуванням вимог до надійності та відмовостійкості.

Рекомендації щодо використання результатів – отримані результати можуть застосовуватися при побудові комерційних систем електронної комерції, аналітичних сервісів, платформ з високими вимогами до продуктивності та масштабованості. Архітектурні підходи та розроблені механізми можуть бути використані для створення гнучких мікросервісних інфраструктур у корпоративних рішеннях, а також як навчальний матеріал для підготовки фахівців з розподілених систем та *DevOps*-практик.

РОЗПОДІЛЕНІ СИСТЕМИ, МІКРОСЕРВІСНА АРХІТЕКТУРА, *APACHE KAFKA*, ВИСОКОНАВАНТАЖЕНІ СИСТЕМИ, *DOCKER, PROMETHEUS, GRAFANA, JWT*, ПОДІЄВО-ОРІЄНТОВАНА ВЗАЄМОДІЯ.

ABSTRACT

Explanatory note to the qualification work “Management System for High-Load Distributed Services Based on Event-Driven Architecture”: 95 pages, 15 figures, 3 tables, 34 bibliographic sources.

Object of research – a high-load distributed software system operating within a microservice and event-driven architecture environment.

Subject of research – principles of designing event-driven microservice systems, methods of ensuring their scalability, reliability, data consistency, and tools for monitoring, logging, and automated deployment in containerized environments.

Purpose of the work – development of a management system for high-load distributed services based on event-driven architecture that ensures scalability, fault tolerance, and efficient interaction between components.

Technical and software tools – Java programming language, Spring Boot framework, Spring Security, JWT authentication, Spring Data JPA, *Apache Kafka* message broker, PostgreSQL relational database system, Docker containerization platform, Maven build system, Prometheus and Grafana monitoring tools, and Apache JMeter for load testing.

Key characteristics and indicators – the system consists of five independent microservices (User, Product, Order, Payment, Notification), communicating via an event-driven mechanism built on Kafka; each service uses an isolated database; horizontal scaling is supported; JWT-based authentication and authorization are implemented; centralized monitoring and logging are configured to collect metrics and performance data. The system can withstand peak loads of more than 10,000 requests according to JMeter scripts.

Results obtained and their novelty – a fully functional microservice ecosystem with event-driven communication has been developed, demonstrating high performance through asynchronous, non-blocking message processing. A scalable architecture with clearly separated business logic allows independent deployment and

updates of services. A centralized monitoring system has been configured, enabling real-time performance tracking and system stability assessment. The novelty of the work lies in the integration of modern architectural and DevOps practices into the development of a high-load distributed system with strict requirements for reliability, resilience, and efficiency.

Recommendations for the practical use of results – the developed approaches can be applied in commercial e-commerce systems, data analytics platforms, and other solutions requiring scalability and high availability. The architectural principles and implemented mechanisms can be used to build flexible microservice infrastructures in enterprise environments as well as serve as educational material for training specialists in distributed systems, microservices, and DevOps practices.

DISTRIBUTED SYSTEMS, MICROSERVICE ARCHITECTURE, *APACHE KAFKA*, HIGH-LOAD SYSTEMS, DOCKER, *PROMETHEUS*, *GRAFANA*, JWT, EVENT-DRIVEN ARCHITECTURE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ. ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface (інтерфейс прикладного програмування)

AWS – Amazon Web Services (хмарна платформа від *Amazon*)

CAP – Consistency, Availability, Partition Tolerance (теорема узгодженості, доступності та стійкості до поділу)

CLI – Command Line Interface (інтерфейс командного рядка)

CPU – Central Processing Unit (центральний процесор)

CRUD – Create, Read, Update, Delete (створення, читання, оновлення, видалення)

CSRF – Cross-Site Request Forgery (міжсайтове підроблення запитів)

CI/CD – Continuous Integration / Continuous Deployment (безперервна інтеграція та безперервне розгортання)

DevOps – Development & Operations (підхід до розробки та експлуатації ПЗ)

DTO – Data Transfer Object (об’єкт передачі даних)

EDA – Event-Driven Architecture (подієво-орієнтована архітектура)

ELK Stack – Elasticsearch, Logstash, Kibana (стек для логування та аналізу даних)

gRPC – Google Remote Procedure Call (високопродуктивний протокол віддаленого виклику процедур)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

ID – Identifier (унікальний ідентифікатор)

JSON – JavaScript Object Notation (формат обміну даними)

JPA – Java Persistence API (інтерфейс роботи з базами даних у *Java*)

JWT – JSON Web Token (токен для автентифікації та авторизації)

JVM – Java Virtual Machine (віртуальна машина *Java*)

PACELC – Partition, Availability, Consistency, Else, Latency, Consistency (розширена теорема компромісів розподілених систем)

RAM – Random Access Memory (оперативна пам’ять)

RBAC – Role-Based Access Control (керування доступом на основі ролей)

REST – Representational State Transfer (архітектурний стиль взаємодії)

SAGA – Saga Pattern (патерн управління розподіленими транзакціями)

SMTP – Simple Mail Transfer Protocol (протокол передавання електронної пошти)

SOLID – Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion (принципи об’єктно-орієнтованого програмування)

SQL – Structured Query Language (мова структурованих запитів)

TLS – Transport Layer Security (захист транспортного рівня)

URL – Uniform Resource Locator (уніфікований покажчик ресурсу)

XML – eXtensible Markup Language (розширювана мова розмітки)

XSS – Cross-Site Scripting (міжсайтове виконання шкідливого коду)

YAML – Yet Another Markup Language (зручна мова розмітки конфігураційних файлів)

БД – база даних

IT – інформаційні технології

ПЗ – програмне забезпечення

ВСТУП

Сучасні інформаційні технології стрімко розвиваються, що зумовлює безперервне збільшення обсягів даних та кількості користувачів, які одночасно взаємодіють із системами. Традиційні монолітні програмні архітектури стають малоефективними в умовах високих навантажень, оскільки їх масштабування та підтримка потребують значних ресурсів і можуть призводити до зниження продуктивності.

Для забезпечення стабільної роботи розподілених систем у реальному часі необхідні нові підходи до організації обчислень та комунікації між компонентами. Мікросервісна архітектура дозволяє посилити масштабованість, гнучкість і надійність систем шляхом розподілу функціональності на незалежні сервіси. Використання подієво-орієнтованої архітектури забезпечує асинхронну взаємодію між сервісами, що дає можливість ефективно обробляти великі обсяги подій та зменшувати затримки в роботі системи.

Особливого значення набуває питання управління такими сервісами: моніторинг, оркестрація, балансування навантаження, забезпечення відмовостійкості та інформаційної безпеки. Саме ці аспекти визначають здатність системи відповідати вимогам до високої доступності, продуктивності та надійності.

Таким чином, дослідження та розробка системи управління високонавантаженими розподіленими сервісами на основі подієво-орієнтованої архітектури є актуальним завданням, що має як наукову, так і прикладну цінність. Результати можуть бути використані у сферах електронної комерції, фінансових технологій, інтернет-сервісів та інших галузях, де критично важливою є швидка та надійна обробка даних.

Мета роботи — розробка системи управління високонавантаженими розподіленими сервісами на основі подієво-орієнтованої архітектури, яка

забезпечує масштабованість, відмовостійкість та ефективну взаємодію між компонентами.

Об’єкт розробки — високонавантажена розподілена програмна система, що функціонує у середовищі мікросервісної та подієво-орієнтованої архітектури.

Для досягнення визначеної мети у роботі визначено такі завдання:

1) Проаналізувати існуючі підходи до побудови розподілених високонавантажених систем та визначити їх переваги й обмеження;

2) Дослідити принципи мікросервісної та подієво-орієнтованої архітектур як основи для створення масштабованої та надійної системи;

3) Вивчити сучасні протоколи комунікації та інструменти управління сервісами (*Docker*, *Kafka*, системи моніторингу);

4) Розробити архітектуру системи управління сервісами з урахуванням вимог до продуктивності, відмовостійкості та безпеки;

5) Реалізувати прототип системи ґрунтуючись на обраному стеку технологій;

6) Провести навантажувальне тестування та дослідження для оцінки масштабованості й відмовостійкості.

Практичне значення отриманих результатів полягає у можливості їх використання під час проєктування та впровадження реальних розподілених систем у сферах електронної комерції, фінансових технологій, телекомунікацій та хмарних сервісів. Запропоновані рішення забезпечують підвищену стійкість до збоїв і масштабованість, що підтверджує їхню придатність до застосування в умовах реальних високих навантажень.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ РОЗПОДІЛЕНИХ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ

Розподілені системи є провідною концепцією сучасної інформатики, вони дозволяють розв'язувати складні завдання шляхом розподілу обчислень між кількома вузлами. Вони особливо актуальні в контексті великих даних, *IoT* та хмарних обчислень, де централізовані підходи недостатньо ефективні.

1.1 Розподілені системи: основні характеристики та класифікація

У сучасних умовах інтенсивного розвитку інформаційних технологій та глобальних мереж зростає потреба у таких обчислювальних системах, які спроможні опрацьовувати значні обсяги інформації, забезпечувати високу доступність сервісів і підтримувати роботу значної кількості користувачів у реальному часі. Звичайні централізовані рішення виявляються недостатньо гнучкими й надійними, що зумовило широке поширення розподілених систем.

Розподілена система — це сукупність комп'ютерних програм, які використовують обчислювальні ресурси на кількох окремих обчислювальних вузлах для досягнення спільної мети [1][2].

Розподілені системи мають наступні характерні риси:

1) Першим аспектом є просторова розподіленість компонент розподіленої системи. Вони вступають у взаємодію або локально або віддалено.

2) Компоненти розподіленої системи можуть працювати паралельно, через що швидкість роботи зростає в порівнянні з послідовною роботою.

4) Кожний стан компоненти розглядається локально, тобто з погляду певного обчислювального процесу, запущеного з локального робочого місця.

5) Компоненти працюють незалежно й можуть «випадати», не руйнуючи систему в цілому, також незалежно одна від одної. Розподілені системи підлягають, таким чином, частковому системному «випаданню».

6) Система працює асинхронно. Процеси комунікації й обробки не управляються глобальним системним часом. Зміни й процеси синхронізуються.

7) У розподіленій системі функції управління розподіляються між різними автономними компонентами. При цьому ніякий окремий компонент не може здійснювати весь контроль. Це гарантує певну міру автономії.

8) Розподілена система може утворюватися як об'єднання вже існуючих систем. Отже, потрібне контекстно-повне управління іменами, що дає можливість однозначно інтерпретувати найменування (імена) в рамках адміністративної або технологічної області. В такому випадку говорять про федеративне управління іменами.

9) Щоб підвищити потужність розподіленої системи, програми й дані можуть переміщатися між різними вузлами, ця концепція називається міграцією. При цьому потрібно використовувати додаткові механізми, які протоколюють положення програм і даних.

10) Розподілена система повинна бути в змозі використовувати динамічні зміни структури. Ця динамічна реконфігурація потрібна, наприклад, тоді, коли протягом певного часу повинні з'являтися нові з'єднання.

11) Архітектура комп'ютерів може використовувати різні топології й механізми, зокрема, якщо апаратура надходить від різних виробників. Ця характеристика називається гетерогенністю.

12) Розподілена система підлягає еволюції, тобто за час її життя відбуваються різні зміни.

13) Джерела відомостей, одиниці обробки й користувачі можуть бути фізично мобільні. Програми й дані можуть переміщатися між вузлами, для одержання мобільності системи або збільшення потужності [3].

1.2 Вимоги до високонавантажених систем

Високонавантажені системи — це розподілені системи, призначені для обробки значних обсягів запитів, даних та користувачів без суттєвого зниження продуктивності. Такі системи, як правило, застосовуються в сферах електронної комерції, соціальних мереж, хмарних сервісів та великих даних, де навантаження може досягати мільйонів транзакцій на секунду. Основні вимоги до них включають масштабованість, відмовостійкість, доступність, узгодженість, надійність та латентність. Кожна вимога пов'язана з іншими, і їх оптимізація часто вимагає компромісів.

Нижче розглянуто ключові вимоги з детальним описом, прикладами та методами досягнення.

Масштабованість — це здатність системи обробляти збільшене навантаження шляхом додавання ресурсів. У розподілених системах масштабованість є важливоюдля підтримки продуктивності та надійності в міру зростання попиту. Якщо система не може масштабуватися, вона ризикує стати млявою або не реагувати під інтенсивним трафіком [4].

Відмовостійкість — це здатність системи підтримувати належну роботу, незважаючи на збої або несправності в одному або кількох її компонентах. Ця здатність є важливою для систем високої доступності, критично важливих або навіть життєво важливих систем [5].

Доступність — властивість системи працювати на високому рівні, безперервно, без втручання, протягом заданого періоду часу. Інфраструктура високої доступності налаштована для забезпечення якісної продуктивності та обробки різних навантажень і збоїв з мінімальним або нульовим часом простою [6].

Узгодженість у розподілених системах означає, що кожен вузол/репліка має однаковий вигляд даних у певний момент часу, незалежно від того, який клієнт оновив дані. Коли ви робите запит до будь-якого вузла, ви отримуєте

абсолютно однакову відповідь (навіть якщо це помилка), тому ззовні виглядає так, ніби всі операції виконує один вузол [7].

Надійність — запит повинен завжди повертати користувачам одні й ті ж дані, щоб користувачі були впевнені, що якщо якісь дані будуть записані/введені в систему, то під час подальшого вилучення можна буде розраховувати на їх незмінність та безпеку [8].

Латентність в розподіленій системі стосується часу, необхідного для передачі запиту від джерела до пункту призначення та для повернення відповідної відповіді до джерела. Ця затримка є критичним фактором продуктивності системи та може залежати від різних елементів. У розподіленій системі, де кілька комп'ютерів або вузлів працюють разом через мережу, на затримку можуть впливати час передачі даних через мережу, процеси серіалізації та десеріалізації даних, а також час, необхідний різним компонентам для обробки даних [9].

Ці вимоги часто конфліктують між собою, як описано в теоремі *CAP* (*Consistency, Availability, Partition Tolerance*) та її розширенні *PACELC*, які стверджують, що в розподілених системах неможливо одночасно забезпечити всі три властивості під час мережевих розділень, а також балансувати консистентність з латентністю в нормальних умовах.

1.3 Архітектурні підходи: моноліт, мікросервіси, подієво-орієнтована архітектура

Розвиток розподілених систем нерозривно пов’язаний з еволюцією архітектурних підходів. Залежно від вимог до продуктивності, масштабованості, гнучкості та відмовостійкості, програмні системи можуть будуватися за різними моделями.

Монолітна архітектура (*Monolithic Architecture*) — це традиційний підхід до розробки програмного забезпечення, при якому весь додаток розробляється

як одна єдина технологічна система. Всі компоненти додатку взаємодіють один з одним і розгортання відбувається на одному сервері або групі серверів.

Монолітна архітектура має декілька переваг. Вона зазвичай простіша в розробці та тестуванні, оскільки всі компоненти додатку взаємодіють безпосередньо один з одним, що дозволяє швидко та легко вирішувати проблеми. Крім того, монолітні додатки можуть бути менш складними в управлінні, оскільки все програмне забезпечення працює на одній технологічній платформі.

Однак, монолітні додатки можуть стати проблемою при масштабуванні, оскільки розширення системи може бути обмеженим через те, що всі компоненти додатку залежать один від одного. Крім того, відносно складна структура монолітних додатків може ускладнити розробку та підтримку.

Отже, монолітна архітектура підходить для невеликих та середніх проектів зі стабільним обсягом функціоналу [10].

Мікросервісна архітектура (*Micro Service Architecture*) – це підхід до створення програмних продуктів, що будується на реалізації незалежних одне від одного модулів (мікросервісів).

Кожен такий модуль невеликий і націлений на вирішення лише одного бізнес-завдання. Мікросервіси тісно взаємодіють одне з одним через спеціалізовані інтерфейси, зберігаючи при цьому незалежність та власну монолітність. Кожен мікросервіс можна оновлювати, змінювати, розгортати та масштабувати без ризиків для цілісності та працездатності системи.

У мікросервісної архітектури немає винахідника або родоначального продукту. Цей підхід поступово та природно набував популярності у розробці ПЗ протягом минулих 20 років. З ускладненням софту та масовим поширенням мережі бізнес почав все частіше покладатися на розподілені системи для підтримки своїх веб-сервісів. Як приклад: будь-яка сучасна стрімінгова платформа складається з мікросервісів: один відповідає за відтворення відео через мережу, другий – за механізми рекомендації контенту, третій – за аналітику, четвертий – за обробку платежів тощо. Реалізувати такий складний продукт в монолітній архітектурі було б неможливо [11].

Коли програмне забезпечення будується за принципом мікросервісів, це дає розробникам багато корисних можливостей і полегшує їхню роботу.

Не слід применшувати важливість мікросервісного підходу для комерційного успіху компаній. В наш час це стало одним із ключових елементів, що визначає прибутковість та продуктивність програмних рішень.

На рисунку 1.1 можемо побачити переваги мікросервісної архітектури.

Рисунок 1.1 — Переваги мікросервісної архітектури

Не слід применшувати важливість мікросервісного підходу для комерційного успіху компаній. В наш час це стало одним із ключових елементів, що визначає прибутковість та продуктивність програмних рішень.

Коли бізнес використовує мікросервіси, він отримує можливість швидше реагувати на зміни ринку, легше адаптувати продукт під потреби клієнтів та економити ресурси на розробці. Це напряму впливає на конкурентоспроможність — компанія може швидше випускати оновлення, краще керувати витратами на інфраструктуру та зменшувати ризики збоїв системи. Фактично, мікросервіси стали важливим інструментом для досягнення бізнес-цілей у сфері інформаційних технологій. Більше того, завдяки

незалежності сервісів, бізнес може масштабувати лише ті компоненти, які справді потребують додаткових ресурсів, що підвищує ефективність роботи системи.

Нижче в таблиці 1.1 наведена порівняльна характеристика моноліту та мікросервісів.

Таблиця 1.1 — Порівняльна характеристика моноліту та мікросервісів

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
Гнучкість та масштабованість	Проста в розробці та легка в запуску, проте втрачає гнучкість при розростанні функцій	Висока масштабованість і гнучкість, але труднощі в розгортанні та керуванні мережею
Управління даними	Зручно працювати з даними в одній базі, однак модифікація структури викликає труднощі	Можливість обирати оптимальні сховища для кожного модуля та уникати залежності від однієї бази, потребує ретельного проектування
Складність розгортання та оновлення	Швидке розгортання завдяки цільній структурі, але оновлення ускладнюється через вплив на весь код	Незалежне оновлення окремих модулів і стабільність при збоях одного компонента, але вимагає складнішого контролю версій

Подійно-орієнтована архітектура *(англ. Event-Driven Architecture, EDA)* — це шаблон проєктування програмного забезпечення, який зосереджується на створенні, виявленні, споживанні та реагуванні на події. Вона дозволяє системам працювати незалежно та реагувати в режимі реального часу на значні зміни або події всередині системи. У *EDA* події виходять на перший план і можуть

походити з різних джерел, тоді як інші компоненти реагують відповідним чином на ці події [12].

У подійно-орієнтованій архітектурі виділяють кілька ключових компонентів:

1) Події — повідомлення про значущі зміни в системі. Вони можуть відображати різноманітні дії, зокрема створення нового запису, взаємодію користувача з інтерфейсом або дані, отримані від сенсора. Події слугують сигналами, які ініціюють реакцію інших компонентів.

2) Виробники подій — це джерела, що генерують події. Ними можуть виступати користувачі, програмні додатки або апаратні пристрої, які створюють повідомлення про зміни стану системи.

3) Споживачі подій реагують на отримані повідомлення та виконують відповідні дії, наприклад оновлення бази даних, надсилання сповіщень чи запуск додаткових процесів.

4) Посередник подій виступає центральним елементом, який приймає події від виробників і забезпечує їх доставку споживачам. Завдяки цьому підтримується надійний і ефективний обмін даними між різними частинами системи.

Подійно-орієнтована архітектура має низку переваг, які роблять її ефективним підходом до проєктування сучасних розподілених систем.

По-перше, роз’єднання компонентів забезпечує незалежність окремих частин системи. Взаємодія здійснюється через події, що знижує рівень залежностей між сервісами та сприяє підвищенню гнучкості й масштабованості.

По-друге, *EDA* підтримує реакцію в режимі реального часу, дозволяючи додаткам оперативно реагувати на зміни. Це особливо важливо для потокових сценаріїв та систем, чутливих до часу, де швидкість обробки подій має ключове значення.

Ще однією перевагою є масштабованість: архітектура дає змогу безболісно додавати або видаляти виробників і споживачів подій, не впливаючи на роботу

всієї системи. Завдяки цьому забезпечується здатність системи адаптуватися до зростаючих обсягів даних.

Нарешті, *EDA* сприяє модульності та повторному використанню компонентів. Події виконують роль стандартизованих інтерфейсів для комунікації, що полегшує оновлення, заміну або інтеграцію нових модулів без ризику порушення функціонування всієї архітектури.

Отже, подійно-орієнтована архітектура формує підхід, у якому саме події стають головним чинником функціонування системи. Такий спосіб організації забезпечує природний механізм взаємодії між компонентами, зменшує їхню залежність і спрощує масштабування. Через здатність обробляти дані у реальному часі *EDA* створює основу для побудови стійких, продуктивних і гнучких рішень, що відповідають вимогам сучасних високонавантажених додатків.

1.4 Методи управління мікросервісними системами

Зі зростанням кількості мікросервісів у розподілених системах виникає потреба у спеціальних методах та інструментах управління. Традиційні підходи адміністрування та розгортання програмного забезпечення виявляються неефективними, оскільки мікросервісна архітектура передбачає десятки або сотні незалежних компонентів, що постійно змінюються. Тому розроблені спеціалізовані підходи до управління: оркестрація контейнерів, сервіс-меш, системи конфігурації та виявлення сервісів.

Оркестрація контейнерів

Контейнер — це уніфікована програмна оболонка, що містить у собі програмний код разом з усіма необхідними компонентами, завдяки чому застосунок може стабільно та ефективно функціонувати незалежно від того, в якому технічному середовищі він запускається.

Оркестрація контейнерів — це процес, який автоматизує управління, масштабування та розгортання великої кількості контейнерів, що працюють у

системах та інших середовищах. На відміну від традиційних практик, контейнери легкі та портативні, що робить їх ідеальними для архітектур мікросервісів та хмарних додатків. Однак, зі збільшенням кількості контейнерів, ручне керування ними стає складнішим. Саме тут і стають у пригоді інструменти оркестрації контейнерів [13].

Інструменти управління контейнерами беруть на себе рутинні операції - від початку роботи та зупинки до зміни масштабу та налаштування мережевих з'єднань між контейнерами. Такі платформи постійно моніторять роботу додатків і самостійно реагують на проблеми, щоб підтримувати стабільне функціонування системи.

Завдяки розумному розподілу навантаження та ефективному використанню серверних потужностей, ці рішення допомагають компаніям економити на інфраструктурі. У сучасних підходах до створення та запуску програмного забезпечення координація контейнерів стала незамінним елементом робочого процесу.

Головне призначення оркестрації контейнерів полягає в тому, щоб зробити процес створення та запуску програм швидшим, стабільнішим та економічно вигіднішим. Завдяки таким рішенням розробники можуть концентруватися на написанні коду та реалізації бізнес-логіки, не відволікаючись на технічні аспекти інфраструктури.

Управління контейнерами активно підтримує методології *DevOps* і забезпечує плавну роботу процесів автоматизованої збірки та доставки коду (*CI/CD*). Платформи на кшталт Kubernetes значно спрощують адміністрування контейнеризованих застосунків, завдяки чому команди стають більш адаптивними та продуктивними в своїй роботі.

Сервіс-меш (*Service Mesh*)

З розвитком мікросервісних архітектур постало питання ефективного управління величезною кількістю взаємодій між окремими сервісами. Якщо на ранніх етапах масштабування достатньо використовувати прості механізми взаємодії (*REST*-запити, брокери повідомлень, балансувальники навантаження),

то в умовах десятків і сотень мікросервісів цього вже не вистачає. Для забезпечення надійності, безпеки та контрольованості з'явився спеціалізований підхід — *Service Mesh***.**

Сервіс-меш — це інфраструктурний рівень, який автоматизує управління мережею сервісів у мікросервісній архітектурі. Він бере на себе завдання комунікації між сервісами, надаючи стандартизований спосіб організації з'єднань, балансування навантаження, маршрутизації запитів, забезпечення безпеки та збору телеметрії.

Сервісна сітка складається з мережевих проксі, що підключаються до кожного сервісу в додатку, і набору процесів управління завданнями. Проксі називають робочою площиною (data plane), а процеси управління — керуючою площиною (control plane). Робоча площина перехоплює виклики між різними сервісами та обробляє їх; керуюча площина є «мозком» сітки, яка координує роботу проксі і надає *API* для операційного та технічного персоналу для управління і спостереження за всією мережею [14].

Управління конфігураціями

У традиційних монолітних застосунках конфігурація (налаштування доступу до БД, зовнішніх *API*, ключі безпеки, параметри середовища) часто зберігається у вигляді файлів (*application.properties*, *config.yaml* тощо). У мікросервісних архітектурах ця проблема ускладнюється, оскільки:

- кількість сервісів може сягати десятків і сотень;
- сервіси розгортаються у різних середовищах (*development*, *staging*, *production*);
- конфігурація постійно змінюється та має бути узгодженою;
- деякі параметри є чутливими (паролі, токени, ключі шифрування).

З вище перерахованого переліку проблем виникає необхідність у безпечному та централізованому керуванні конфігураціями.

Управління конфігураціями — це процес організації, відстеження та контролю змін, внесених до програмного забезпечення, апаратного забезпечення та документації організації. Це забезпечує, що конфігурації систем організації

залишаються послідовними, безпечними та ефективно керованими. Завдяки впровадженню управління конфігураціями, організації можуть ефективно справлятися зі складностями IT-інфраструктури та забезпечувати, що зміни вносяться контрольованим і авторизованим чином [15].

***Service Discovery* (виявлення сервісів)**

У розподілених системах сервіси є динамічними — вони можуть масштабуватися (створювати нові інстанси), переміщуватися між вузлами кластера або перезапускатися у разі збоїв. Тому статичні адреси та порти для доступу до сервісів стають непридатними: будь-яка зміна конфігурації призводить до відмов у взаємодії між компонентами.

Щоб забезпечити безперервну роботу системи, необхідний механізм, який дозволяє автоматично виявляти доступні сервіси та їхні адреси. Цю задачу вирішує *Service Discovery*.

Виявлення сервісів — це процес автоматичного виявлення пристроїв і служб у комп'ютерній мережі. Його метою є зменшення зусиль, пов'язаних з ручним налаштуванням, які вимагаються від користувачів та адміністраторів. Протокол виявлення служб – це мережевий протокол, який допомагає здійснювати виявлення служб [16].

Отже, разом ці методи створюють середовище, у якому сервіси можуть функціонувати незалежно, але водночас координовано, а адміністрування та моніторинг системи стають керованими та передбачуваними. Це дозволяє вирішувати поточні завдання високонавантажених розподілених систем, а також закладати фундамент для їхнього еволюційного розвитку, де масштабованість, відмовостійкість і безпека стають природною властивістю архітектури, а не додатковим зусиллям розробників.

1.5 Протоколи комунікації та інтеграційні патерни

Обмін даними між сервісами відбувається через чітко визначені протоколи комунікації та інтеграційні патерни. Вибір правильного підходу визначає всю

подальшу взаємодію та роботу у системі. Протоколи можна умовно поділити на синхронні та асинхронні, а патерни інтеграції — на прямі виклики, брокерські системи та подієво-орієнтовані підходи.

Синхронна комунікація — це спосіб взаємодії між компонентами системи, при якому відправник запиту чекає на отримання відповіді перед тим, як продовжити виконання наступних операцій.

Переваги такої форми комунікації в простоті реалізації та передбачуваності. Недоліки — затримка в часі на очікування результату, а також блокування ресурсів.

Приклади синхронної комунікації:

1) *HTTP*-запити — коли браузер відправляє запит на веб-сервер (наприклад, для завантаження сторінки), він чекає на відповідь перед тим, як відобразити контент користувачу. Під час цього очікування браузер може показувати індикатор завантаження.

2) *REST API* виклики - мобільний додаток, що запитує дані про погоду з сервера, зупиняє виконання цієї частини коду доки не отримає JSON-відповідь з температурою та прогнозом.

Синхронний зв'язок корисний, якщо він здійснюється лише між кількома мікросервісами. Однак якщо система передбачає ланцюгові виклики між багатьма сервісами з очікуванням на завершення складних або тривалих процесів, краще застосовувати асинхронні підходи. Інакше тісна взаємозалежність між компонентами призведе до появи критичних точок відмови та архітектурних проблем у системі.

Коли в системі присутні багато мікросервісів, що потребують взаємодії один з одним, і ви прагнете мінімізувати їхню залежність або досягти слабкого зв'язування, доцільно застосовувати асинхронну комунікацію через обмін повідомленнями. Завдяки тому, що асинхронна комунікація через повідомлення оперує подіями, саме ці події стають основою для взаємодії між мікросервісами. Такий тип зв'язку називається подієво-орієнтованою архітектурою або *event-driven* комунікацією.

На рисунку 1.2 наведено приклад роботи синхронної комунікації.

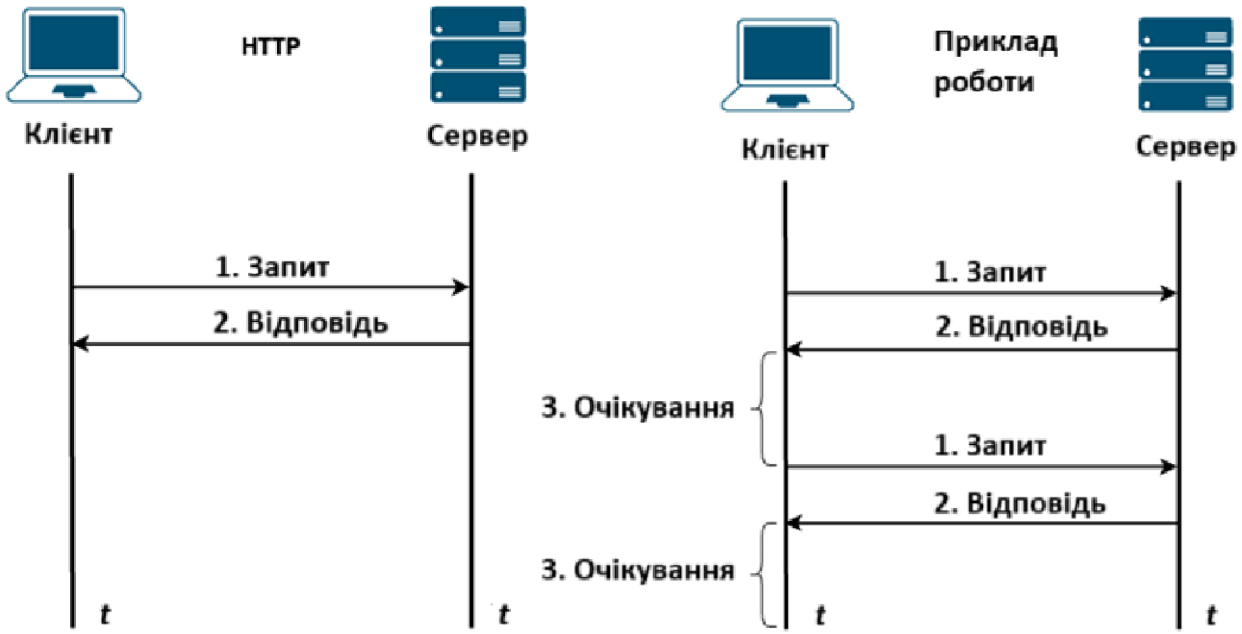

Рисунок 1.2 — Синхронна комунікація

Існує два види асинхронної комунікації:

1) зв'язок на основі повідомлень з одним одержувачем (один до одного);

2) зв'язок на основі повідомлень з кількома одержувачами (публікація/підписка).

Зв'язок на основі повідомлень з одним одержувачем в основному призначений для здійснення зв'язку один-до-одного або точка-точка. Якщо ми надсилаємо один запит конкретному споживачеві, і ця операція триватиме довго, тоді доцільно використовувати цей асинхронний зв'язок один-до-одного з одним одержувачем.

Якщо розглянути діаграму, можна побачити приклад використання такої комунікації - запит СтворитиЗамовлення надходить від мікросервісу Кошик, і оскільки процес створення замовлення може тривати досить довго, система не надсилає миттєвої відповіді назад до мікросервісу Кошик.

На рисунку 1.3 зображено комунікацію на основі повідомлень з одним отримувачем.

Рисунок 1.3 — Комунікація на основі повідомлень з одним отримувачем

Зв'язок на основі повідомлень з кількома одержувачами призначений для реалізації механізму публікація/підписка з кількома отримувачами. У такій архітектурі сервіс-постачальник розміщує повідомлення, яке споживається декількома мікросервісами, що підписані на ці повідомлення через систему брокера повідомлень. Для здійснення операцій публікація/підписка необхідна шина подій, що дозволяє публікувати події для всіх підписників.

Завдяки такому підходу до комунікації видавець не потребує інформації про підписників, що означає відсутність будь-яких залежностей між учасниками взаємодії.

Переважно такий асинхронний підхід застосовується в системах з подієво-орієнтованою архітектурою. При асинхронній подієвій взаємодії мікросервіс генерує подію у момент настання певної зміни. Наприклад, коли відбувається корегування вартості товару в сервісі управління продуктами. На подію "Оновлення ціни" може підписатися сервіс "Торговельний кошик", щоб автоматично актуалізувати вартість товарів у кошику покупця без синхронного очікування.

Нижче на рисунку 1.4 можна побачити комунікацію з множинними отримувачами.

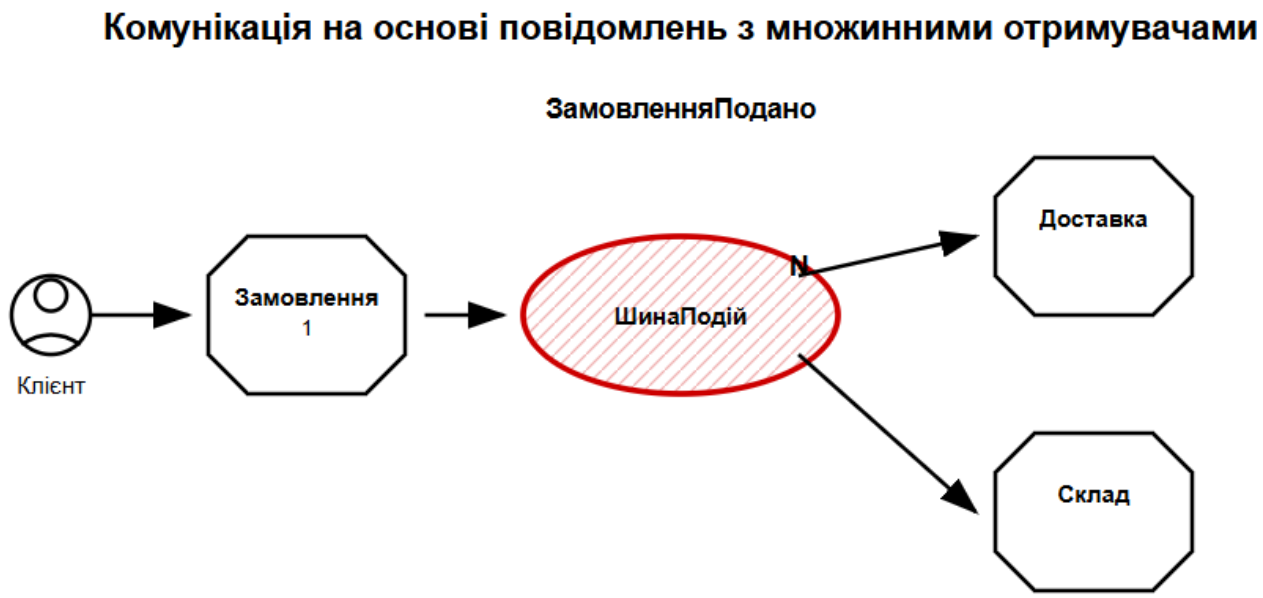

Рисунок 1.4 — Комунікація на основі повідомлень з кількома одержувачами

Ці паттерни публікація/підписка реалізуються за допомогою шини подій. Шина подій також може мати реалізації через системи брокерів повідомлень, які підтримують асинхронну комунікацію та модель публікація/підписка, такі як *Kafka* та *RabbitMQ*.

Розглянемо наведену вище діаграму - це приклад обміну повідомленнями за принципом публікація/підписка на основі подієво-керованої комунікації. Мікросервіс Замовлення публікує події ЗамовленняПодано в шину подій, а мікросервіси Доставка та Склад можуть підписуватись на цю подію та виконувати свої внутрішні дії відповідно до деталей події.

Протоколи комунікації та інтеграційні патерни формують «мозок» взаємодії мікросервісів, визначаючи, як окремі компоненти системи обмінюються інформацією та реагують на зміни у навантаженні. Синхронні протоколи, такі як *REST* та *gRPC*, забезпечують простоту та передбачуваність запитів у сценаріях із негайною відповіддю, тоді як асинхронні механізми, зокрема брокери повідомлень та архітектура *event-driven*, дозволяють створювати масштабовані й відмовостійкі системи, здатні обробляти величезні потоки даних без блокування критичних ресурсів.

Правильне поєднання протоколів і патернів дає змогу не лише забезпечити безперервність роботи та ефективність взаємодії сервісів, а й закласти основу для еволюції системи: нові сервіси можна підключати швидко, зміни в архітектурі не порушують існуючу функціональність, а масштабування відбувається прозоро для користувачів.

Тобто, протоколи комунікації та інтеграційні патерни стають не просто технічним інструментом, а стратегічним компонентом, який визначає, наскільки розподілена система здатна відповідати вимогам сучасних високонавантажених середовищ.

1.6 Сучасні технології для управління сервісами

Сучасні розподілені високонавантажені системи вимагають не лише продуманої архітектури, а й надійних технологічних інструментів для управління сервісами на всіх рівнях — від контейнеризації та оркестрації до моніторингу та спостережливості. Еволюція таких технологій відбулася від простих скриптів розгортання до повноцінних оркестраційних і спостережливих платформ, що автоматизують управління життєвим циклом додатків [17].

Контейнеризація

Контейнери стали стандартом де-факто для упакування та розгортання мікросервісів. *Docker*, як провідна технологія контейнеризації, дозволяє створювати легкі, портативні та ізольовані середовища для виконання додатків. *Docker* спрощує процес *CI/CD*, забезпечуючи однаковість середовища на етапах розробки, тестування й продакшну [18].

Основні переваги контейнеризації включають:

- ізоляцію процесів (кожен сервіс працює у власному середовищі);
- портативність (контейнери можуть бути запущені на будь-якому сервері з *Docker Engine)*;
- ефективність використання ресурсів (контейнери споживають менше ресурсів, ніж віртуальні машини);

- швидкість розгортання (контейнер запускається за секунди).

Docker також підтримує власну систему оркестрації — *Docker Swarm*, яка забезпечує просте масштабування та балансування навантаження.

Платформи оркестрації контейнерів

Kubernetes — це портативна, розширювана платформа з відкритим кодом для керування контейнеризованими робочими навантаженнями та сервісами, яка спрощує як декларативне налаштування, так і автоматизацію [19].

Основні принципи проєктування *Kubernetes*:

- безпечність (ізоляція *namespaces*, контроль доступу *RBAC*, *secrets management)*;
- зручність використання (керування через *CLI* (*kubectl*) або *YAML*-конфігурації);
- розширюваність (підтримка *CRD* (*Custom Resource Definitions*), що дозволяє додавати власні логічні ресурси).

Kubernetes підтримує інтеграцію з *Service Mesh*-рішеннями, *API Gateway* і системами моніторингу.

Docker Swarm представляє собою вбудовану в *Docker* систему оркестрації, яка надає простіший підхід до управління кластерами контейнерів. Хоча вона менш функціональна за *Kubernetes*, *Docker Swarm* ідеально підходить для невеликих та середніх проєктів завдяки своїй простоті налаштування та використання.

Apache Mesos пропонує абстракцію ресурсів датацентру, дозволяючи запускати різні типи робочих навантажень на одному кластері. Платформа особливо корисна в гетерогенних середовищах, де потрібно управляти не тільки контейнерами, а й іншими типами додатків.

***Service Mesh* технології**

Istio є одним із найпопулярніших рішень *Service Mesh*, що інтегрується з *Kubernetes* та надає широкий спектр можливостей:

- керування трафіком (інтелектуальне управління трафіком, включаючи *load balancing*, *circuit breaking*, та *retry* логіку);
- безпека (автоматичне шифрування трафіку між сервісами, аутентифікація та авторизація);
- спостережуваність (детальні метрики, логи та трейси для всіх викликів між сервісами).

Linkerd позиціонується як легший та простіший у налаштуванні *Service Mesh*. Він забезпечує основні функції управління трафіком та безпеки з мінімальним впливом на продуктивність системи.

***API Gateway* рішення**

API Gateway виступає як єдина точка входу для клієнтів, що взаємодіють з мікросервісною архітектурою. Ці рішення забезпечують централізоване управління *API*, безпеку та моніторинг.

Kong є високопродуктивним *API Gateway* з відкритим вихідним кодом, що надає функції аутентифікації, *rate limiting*, логування та аналітики. Платформа легко масштабується та інтегрується з різними системами аутентифікації.

Ambassador, побудований на базі *Envoy Proxy*, пропонує *Kubernetes-native* підхід до управління *API Gateway* функціональністю з акцентом на простоту конфігурації через *Kubernetes* ресурси.

Amazon API Gateway надає повністю керований сервіс для створення, публікації та управління *API* в хмарному середовищі *AWS* з інтеграцією до інших *AWS* сервісів.

Моніторинг та спостережуваність

Ефективне управління мікросервісами неможливе без комплексного моніторингу та систем спостережливості, тому давайте розглянемо існуючі для цього інструменти.

Prometheus служить системою збору метрик з тайм-серіями, оптимізованою для моніторингу динамічних середовищ. Grafana забезпечує потужні можливості візуалізації цих метрик через гнучкі дашборди.

ELK Stack надає комплексне рішення для централізованого збору, обробки та аналізу логів від усіх сервісів в системі.

Сучасні технології для управління сервісами надають розробникам та операційним командам потужні інструменти для побудови надійних, масштабованих та ефективних мікросервісних архітектур. Вибір конкретних технологій залежить від специфічних вимог проєкту, розміру команди та існуючої інфраструктури організації.

1.7 Технології для побудови сервісів

Побудова сервісів вимагає ретельного вибору технологій, які забезпечать всі ті якісні характеристики які ми розкривали вище. Сучасні розробники мають доступ до широкого спектру мов програмування, фреймворків та інструментів, кожен з яких має свої особливості та сфери оптимального застосування.

Екосистема *Java*

Java вже понад два десятиліття залишається однією з найпопулярніших мов програмування для корпоративної розробки, і це не випадковість. Мова була створена з філософією "*write once, run anywhere*" - написати один раз, запускати скрізь. Ця концепція виявилася надзвичайно цінною в світі мікросервісів, де додатки можуть працювати в різних середовищах — від локальної машини розробника до продакшн кластерів в хмарі.

Зрілість екосистеми *Java* неможливо переоцінити. Протягом десятиліть навколо Java сформувалася величезна кількість бібліотек, інструментів та фреймворків, які вирішують практично будь-які завдання — від роботи з базами даних до інтеграції з хмарними сервісами. Це означає, що розробникам рідко доводиться "винаходити велосипед" майже для кожної потреби вже існує перевірена, добре документована бібліотека.

Важливою перевагою є статична типізація, яка забезпечує високу надійність коду. Багато помилок виявляються ще на етапі компіляції, що значно

знижує ймовірність збоїв у продакшн-середовищі. Це особливо критично в мікросервісах, де збій одного сервісу може вплинути на всю систему.

Корпоративна екосистема *Java* включає широкий набір інструментів, які стали стандартом у професійній розробці:

- *IntelliJ IDEA ma Eclipse* — інтегровані середовища розробки;
- *Maven* та *Gradle* — системи збірки та управління залежностями;
- інструменти для тестування, профілювання та моніторингу, що полегшують підтримку складних систем.

Spring Framework революціонізував *Java* розробку, коли з'явився на початку 2000-х, і з тих пір став стандартом для корпоративної розробки. Основна ідея *Spring* - це інверсія управління (*Inversion of Control*) та впровадження залежностей (*Dependency Injection*), які дозволяють створювати слабо зв'язані, легко тестовані додатки.

Spring Boot, який з'явився в 2014 році, вивів зручність *Java* розробки на новий рівень. Головна перевага *Spring Boot* - це концепція "*convention over configuration*", коли фреймворк сам приймає розумні рішення за розробника. Замість написання сотень рядків *XML* конфігурації (як було в оригінальному Spring), розробник просто додає відповідні залежності, і *Spring Boot* автоматично налаштовує все необхідне.

Для мікросервісів особливо важливим є *Spring Boot Actuator*, який надає готові *endpoint'*и для моніторингу здоров'я додатка, збору метрик і логів. Він легко інтегрується з *Prometheus*, *Micrometer* та іншими системами моніторингу. Додатково, екосистема *Spring Cloud* пропонує готові рішення для *service discovery (Eureka), circuit breakers (Hystrix), API gateway (Zuul)* та централізованої конфігурації *(Config Server)*, що дозволяє зосередитися на бізнес-логіці.

Окремої уваги заслуговує *Spring Security*, який забезпечує комплексну безпеку з мінімальними налаштуваннями. У мікросервісних системах він легко інтегрується з *JWT, OAuth2* чи корпоративними системами автентифікації, а декларативний підхід із використанням анотацій робить конфігурацію простою та прозорою.

Не менш важливою частиною є підтримка тестування. У *Spring Boot* передбачені зручні механізми для перевірки різних шарів додатку:

- *@SpringBootTest* для інтеграційних тестів;
- *@WebMvcTest* для тестування *web layer*;
- *@DataJpaTest* для перевірки *JPA repositories*.

Завдяки цьому можна швидко писати надійні тести, що охоплюють різні рівні системи.

Величезна спільнота та документація *Spring* роблять цю технологію особливо привабливою для корпоративних команд. Будь-яка проблема, з якою ви можете зіткнутися, скоріш за все, вже була вирішена кимось іншим, а рішення детально описане в документації або на *Stack Overflow*. Це значно знижує ризики проекту та прискорює розробку.

Node.js та JavaScript

Node.js — це середовище виконання *JavaScript* поза браузером, побудоване на движку *Google*. Воно реалізує подієво-орієнтовану, неблокуючу модель програмування, що робить його ідеальним для побудови систем, які повинні обробляти тисячі одночасних запитів із мінімальною затримкою. Саме ця особливість дозволяє *Node.js* широко застосовувати у високонавантажених веб-додатках, мікросервісах та сервісах реального часу (чатах, стрімінгах, фінансових платформах).

Переваги *Node.js*:

- асинхронність і масштабованість;
- єдина мова (*JavaScript*) на клієнті та сервері;
- велика екосистема пакетів (*npm*).

Серед популярних фреймворків на базі *Node.js* особливо вирізняється *Express.js*, який завдяки своїй мінімалістичній структурі та простій системі маршрутизації став стандартом для створення веб-додатків і невеликих мікросервісів. Для проєктів, де критичною є продуктивність і швидкість обробки запитів, дедалі частіше обирають *Fastify*, який завдяки асинхронній архітектурі

за замовчуванням та вбудованій підтримці *JSON Schema* для валідації забезпечує кращі показники ефективності. А для великих корпоративних рішень використовується *NestJS*, побудований поверх *Express* або *Fastify*, який завдяки підтримці *TypeScript*, системі декораторів і розвинутому dependency injection пропонує масштабовану архітектуру рівня enterprise.

Python

Python залишається однією з найпопулярніших мов програмування для побудови мікросервісів завдяки простому синтаксису, високій швидкості розробки та широкій екосистемі бібліотек. Цю мову часто застосовують у сервісах, орієнтованих на обробку даних, інтеграцію зі штучним інтелектом, машинне навчання та аналітику, однак вона не менш ефективна й для класичних бізнес-додатків. Основними перевагами *Python* є простота засвоєння, велика кількість інструментів для різних завдань і потужна спільнота.

Серед найбільш поширених фреймворків у контексті мікросервісної архітектури можна відзначити *FastAPI*, який завдяки нативній підтримці *async/await*, автоматичній генерації *OpenAPI/Swagger*-документації та інтеграції з бібліотекою Pydantic став одним із найсучасніших інструментів для створення API. У випадках, коли потрібне легке та мінімалістичне рішення, популярним вибором є *Flask*, який надає лише базові інструменти для створення веб-додатків та *API*, залишаючи розробнику повну свободу у виборі додаткових бібліотек і архітектурних підходів.

Висновки до розділу

У першому розділі розглянуто теоретичні основи побудови розподілених високонавантажених систем, що дозволило сформувати цілісне уявлення про принципи їх функціонування та сучасні підходи до проєктування. Проаналізовано основні характеристики розподілених систем, їх класифікацію та ключові вимоги, зокрема масштабованість, відмовостійкість, продуктивність

і безпеку, які є критичними для систем, що працюють в умовах високого навантаження.

Окрему увагу приділено порівнянню архітектурних підходів, таких як монолітна, мікросервісна та подійно-орієнтована архітектури, що дало змогу обґрунтувати доцільність використання мікросервісної та подійно-орієнтованої моделей для побудови гнучких і масштабованих систем. Розглянуто методи управління мікросервісними системами, сучасні протоколи комунікації та інтеграційні патерни, які забезпечують ефективну взаємодію між компонентами системи.

Також проаналізовано сучасні технології та інструменти для управління сервісами і побудови програмних компонентів, що широко застосовуються у промислових рішеннях. Отримані теоретичні положення стали основою для подальшого аналізу предметної області, формування вимог до системи та вибору архітектурних і технологічних рішень у наступних розділах роботи.

РОЗДІЛ 2
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

2.1 Вибір предметної області

Для практичної реалізації системи управління високонавантаженими розподіленими сервісами обрано предметну область електронної комерції, оскільки вона є однією з найбільш показових у контексті побудови масштабованих і подієво-орієнтованих систем. Інтернет-магазин об’єднує широкий спектр бізнес-процесів — від управління користувачами та товарами до оформлення замовлень, платежів, обробки сповіщень і аналітики. Така система природно розділяється на окремі сервіси, що робить її зручною для реалізації мікросервісної архітектури.

У межах цієї предметної області передбачено створення розподіленої системи, де кожен мікросервіс відповідає за конкретну бізнес-функцію: автентифікацію користувачів, управління товарами, обробку замовлень, оплату, а також взаємодію між ними за допомогою подієво-орієнтованого підходу. Така модель забезпечує незалежний розвиток окремих компонентів, спрощує масштабування та підвищує відмовостійкість системи.

Вибір інтернет-магазину також зумовлений наявністю реальних сценаріїв високого навантаження, коли одночасно тисячі користувачів можуть виконувати запити на перегляд товарів, додавання їх до кошика або оформлення замовлень. Це створює умови для дослідження продуктивності, оптимізації комунікації між сервісами та застосування механізмів асинхронної обробки подій.

Таким чином, предметна область інтернет-магазину є оптимальною для реалізації та тестування принципів подієво-орієнтованої мікросервісної архітектури, що дозволяє на практиці продемонструвати її переваги у

масштабованості, стійкості до збоїв та швидкості обробки даних у реальному середовищі.

2.2 Вимоги до системи (функціональні та нефункціональні)

Після визначення предметної області інтернет-магазину необхідно сформулювати вимоги до майбутньої системи. Вони визначають очікувану поведінку, можливості та якісні характеристики програмного продукту. Усі вимоги поділяються на функціональні, що описують конкретні функції системи, та нефункціональні, які визначають її технічні, експлуатаційні та якісні показники.

Функціональні вимоги

Система повинна забезпечувати повний цикл взаємодії користувача з інтернет-магазином, включаючи реєстрацію, перегляд товарів, оформлення замовлень та оплату. Основні функціональні вимоги можна сформулювати таким чином.

Управління користувачами:

- реєстрація користувача з верифікацією електронної пошти через email-посилання;
- авторизація користувача з отриманням *JWT* токена;
- вихід із системи із внесенням токена до “чорного списку”;
- підтвердження облікового запису через токен;
- перегляд списку всіх користувачів (для адміністратора) та власного профілю;
- отримання, оновлення або видалення користувача за email.

Управління категоріями товарів:

- перегляд усіх категорій, батьківських і підкатегорій;
- перегляд конкретної категорії за *ID*;

- створення, редагування та видалення категорій (доступно адміністраторам);
- підтримка ієрархічної структури категорій (*parent–child*).

Управління товарами:

- перегляд усіх товарів або товарів за категорією;
- пошук товарів за ключовим словом;
- отримання детальної інформації про товар за його ID;
- створення, оновлення та видалення товарів адміністраторами;
- можливість прив’язки товару до категорії.

Управління зображеннями товарів:

- додавання нового зображення до товару;
- перегляд усіх зображень певного товару;
- видалення зображень за *ID*.

Управління замовленнями:

- перегляд усіх замовлень (для адміністратора) або лише власних замовлень;
- перегляд замовлення за *ID*;
- створення нового замовлення;
- зміна статусу замовлення (наприклад, “оплачено”, “доставлено”);
- скасування замовлення користувачем;
- видалення замовлення (для адміністратора).

Управління товарами в замовленні:

- створення елемента замовлення;
- отримання елементів замовлення за *ID* або за замовленням;
- оновлення кількості товару в замовленні;
- видалення окремого елемента або всіх елементів певного замовлення.

Управління оплатами:

- отримання інформації про оплату за *ID* або за номером замовлення;

- перегляд усіх оплат (для адміністратора);
- оновлення статусу оплати *(/{paymentId}/status)*, наприклад, “успішно”, “неуспішно”.

Система сповіщень:

- автоматичне надсилання email-повідомлень при: реєстрації користувача (вітальне повідомлення); створенні замовлення; зміні статусу замовлення; успішній або невдалій оплаті; запиті на верифікацію електронної пошти.
- використання *Kafka* для асинхронної обробки подій: *user-registered, order-created, payment-processed, payment-failed, email-verification.*

Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи, її надійність, масштабованість, безпеку та зручність у підтримці. Вони забезпечують стабільну роботу системи при зростанні навантаження та гарантують зручність експлуатації для користувачів і розробників.

Система має базуватись на мікросервісній архітектурі з чітким розподілом відповідальностей між сервісами. Кожен сервіс повинен бути автономним, мати власну базу даних і спілкуватися з іншими через *REST API* або повідомлення *Kafka*.

Система повинна забезпечувати обробку великої кількості одночасних запитів без суттєвого зниження продуктивності. Підтримується горизонтальне масштабування окремих сервісів у контейнерах, що дозволяє ефективно реагувати на збільшення навантаження.

У разі збою одного з компонентів система має продовжувати роботу завдяки реплікації, механізмам повторної відправки подій (*retry*) та використанню ідемпотентних операцій.

Реалізована через аутентифікацію за допомогою *JWT*, централізовану перевірку токенів і систему ролей (користувач, адміністратор). Передбачено захист від найпоширеніших атак — *SQL Injection, XSS, CSRF, brute-force.* Конфіденційні дані зберігаються у зашифрованому вигляді.

Кожен сервіс розгортається в *Docker*-контейнері, що спрощує розгортання та підвищує портативність системи. Для оркестрації контейнерів застосовується Kubernetes, який забезпечує автоматичне масштабування, балансування навантаження та оновлення без простоїв.

Код має відповідати принципам *Clean Architecture* та *SOLID*, бути модульним, тестованим і легко розширюваним.

Передбачено юніт-, інтеграційні та контрактні тести (*JUnit, Spring Boot Test*) для перевірки стабільності компонентів.

Система повинна мати зрозумілу документацію *API* (через *OpenAPI/Swagger*), що забезпечує просту інтеграцію з іншими сервісами та мінімізує витрати часу на підтримку.

2.3 Оцінка навантаження і масштабованості

При проєктуванні системи важливо не лише визначити її функціональну структуру, а й оцінити очікуване навантаження та можливості масштабування. Це дозволяє передбачити потенційні вузькі місця, оптимізувати архітектуру та забезпечити стабільну роботу системи в умовах зростання кількості користувачів і транзакцій.

Оцінка навантаження проводиться з урахуванням кількості активних користувачів, частоти звернень до системи та обсягу операцій, що виконуються щодня.

Для базового сценарію передбачається, що інтернет-магазин обслуговує:

- близько 10 000 зареєстрованих користувачів, з яких одночасно активними є близько 500–1000;
- орієнтовно 200–300 замовлень на день у звичайному режимі та до 2000–3000 у періоди пікових навантажень (наприклад, під час акцій або святкових розпродажів);
- до 50 000 товарів у каталозі з великою кількістю запитів на пошук, фільтрацію та перегляд.

Зважаючи на ці показники, середній рівень навантаження на систему можна класифікувати як середній із потенційним зростанням до високого. Це вимагає використання архітектурних рішень, здатних ефективно масштабуватися горизонтально.

Масштабованість є одним із ключових нефункціональних критеріїв якості високонавантажених систем. Запропонована архітектура базується на мікросервісному підході, який природно підтримує масштабування за рахунок незалежного розгортання та масштабування окремих компонентів.

Нам потрібне масштабування для побудови стійкої системи та покращення взаємодії з користувачем. Справлятися зі зростаючим навантаженням користувачів та трафіком. Забезпечувати високу доступність та надійність. Підтримувати продуктивність та час відгуку. Підтримувати зростаючі потреби в даних та сховищі.

Горизонтальне масштабування, також відоме як масштабування назовні, стосується процесу збільшення потужності або продуктивності системи шляхом додавання більшої кількості машин або серверів для розподілу робочого навантаження між більшою кількістю окремих пристроїв.

Розподілена архітектура дозволяє суттєво збільшити ємність системи, оскільки додавання нових вузлів або екземплярів сервісів дає змогу обробляти значно більшу кількість вхідних запитів. Завдяки розподілу навантаження між кількома серверами підвищується загальна продуктивність системи та зменшується ризик перевантаження окремих компонентів. Крім того, така архітектура забезпечує вищу відмовостійкість: у разі виходу з ладу одного вузла запити можуть бути перенаправлені на інші, що мінімізує простої та підвищує надійність роботи системи в цілому.

Водночас розподілені системи потребують більш складної архітектури, яка включає балансувальники навантаження, механізми реплікації та розподілені бази даних, що ускладнює проєктування та впровадження. Підтримка суворої узгодженості між вузлами є складним завданням і вимагає використання синхронізації, обміну повідомленнями або спеціальних протоколів реплікації.

Збільшення кількості серверів призводить до зростання витрат на мережеві ресурси, обчислювальні потужності та обслуговування, а також потребує застосування інструментів оркестрації, таких як Kubernetes, для централізованого керування інфраструктурою. Крім того, у розподіленому середовищі помилки можуть поширюватися між вузлами, що ускладнює пошук першопричин проблем, а взаємодія між сервісами через мережу додає додаткові затримки та підвищує загальну складність системи.

Вертикальне масштабування, також відоме як масштабування вгору, полягає у збільшенні обчислювальної потужності або можливостей окремого апаратного чи програмного компонента системи. Такий підхід передбачає модернізацію одного сервера шляхом додавання процесорних ресурсів, оперативної пам’яті або інших апаратних компонентів, що безпосередньо підвищує його продуктивність і здатність обробляти більшу кількість вхідних запитів. Однією з основних переваг вертикального масштабування є відносна простота керування, оскільки адміністрування одного вузла, як правило, є менш складним у порівнянні з підтримкою та синхронізацією кількох серверів.

Водночас вертикальне масштабування має низку суттєвих обмежень. Масштабованість у цьому випадку визначається фізичними можливостями обладнання, що не дозволяє необмежено нарощувати ресурси, на відміну від горизонтального масштабування. Крім того, збереження всієї обробки запитів на одному сервері підвищує ризик повного простою системи у разі його відмови. Процес масштабування часто потребує перезапуску або заміни машини, що може призводити до тимчасової недоступності сервісу та негативно впливати на безперервність його роботи [20]. Наприкінці доцільно відзначити, що на відповідному рисунку наочно продемонстровано відмінності між вертикальним та горизонтальним масштабуванням.

На рисунку 2.1 зображена різниця між вертикальним та горизонтальним маштабуванням.

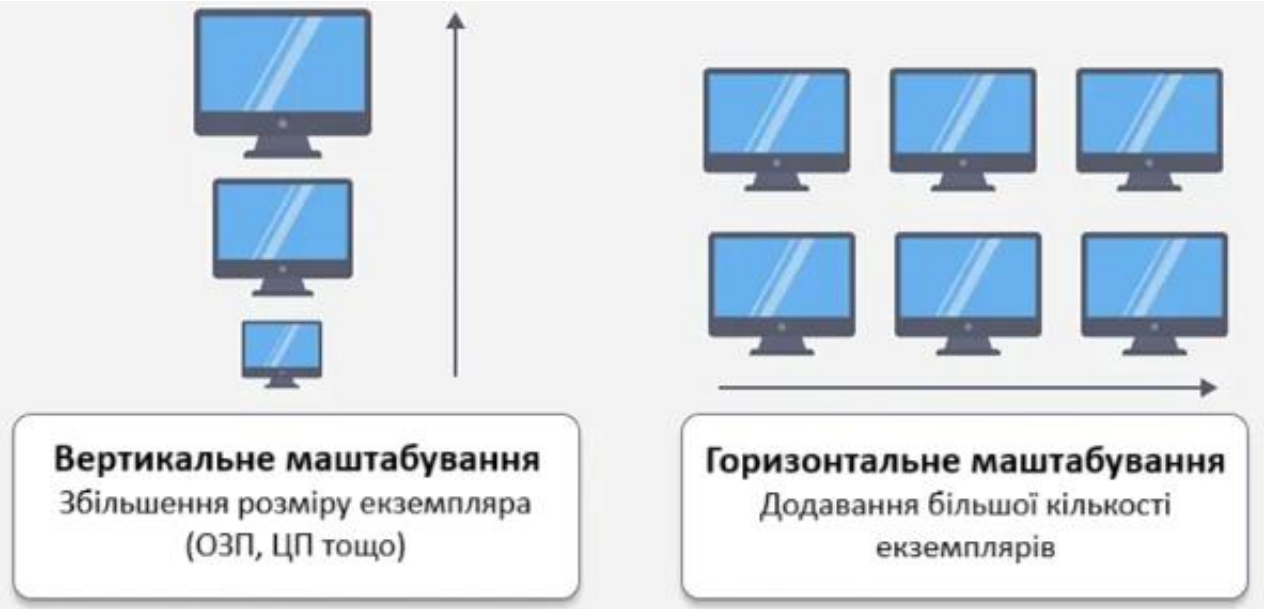

Рисунок 2.1 — Вертикальне та горизонтальне маштабування

Давайте коротко порівняємо ці два підходи, враховуючи їх переваги та недоліки. Обидва типи масштабування мають різну сферу застосування та є доцільними залежно від вимог до системи, характеру навантаження та обмежень інфраструктури. Вертикальне масштабування є простішим у реалізації та не потребує складної координації між компонентами, проте має фізичні межі й нижчу відмовостійкість. Натомість горизонтальне масштабування забезпечує кращу гнучкість, масштабованість і стійкість до збоїв за рахунок використання кількох вузлів, але потребує більш складної архітектури та додаткових механізмів керування. Вибір підходу масштабування безпосередньо впливає на подальшу архітектуру системи, витрати на її підтримку та можливості розвитку, а також визначає здатність системи ефективно реагувати на зростання навантаження у довгостроковій перспективі. Крім того, на практиці часто застосовується комбінований підхід, який поєднує вертикальне та горизонтальне масштабування, що дозволяє досягти оптимального балансу між простотою реалізації, продуктивністю та надійністю системи.

Далі наведена порівняльна таблиця 2.1 горизонтального та вертикального масштабування.

Таблиця 2.1 — Порівняльна таблиця горизонтального та вертикального маштабування

Аспект	Горизонтальне масштабування	Вертикальне масштабування
Додавання ресурсів	Додає нові машини або сервери для розподілу навантаження між ними	Підвищує ресурси (*CPU*, *RAM*, диск) окремого сервера або компонента
Гнучкість	Забезпечує високу гнучкість — можна легко додавати або вилучати вузли	Обмежена гнучкість, оскільки апгрейд одного сервера має фізичні межі
Відмовостійкість	Підвищує відмовостійкість завдяки розподілу навантаження між кількома вузлами	Менш відмовостійке, оскільки вся система залежить від одного сервера
Продуктивність	Продуктивність зростає зі збільшенням кількості вузлів, що розподіляють навантаження	Продуктивність покращується до певної межі, після чого обмежується можливостями апаратури
Сфера застосування	Оптимальне рішення для систем, що потребують високої масштабованості та розподілених обчислень	Доцільне для помірних навантажень або на ранніх етапах розвитку системи

Горизонтальне масштабування є більш придатним для сучасних розподілених систем, оскільки забезпечує гнучкість, високу відмовостійкість і можливість обробляти зростаючі навантаження без суттєвих змін архітектури. Натомість вертикальне масштабування може бути ефективним на початкових етапах розвитку проєкту або для систем із помірними вимогами до продуктивності, але має обмеження, пов’язані з апаратними ресурсами та ризиком єдиної точки відмови.

2.4 Архітектурні виклики та обмеження

Під час проєктування високонавантажених розподілених систем одним із ключових аспектів є вибір архітектурного підходу, який забезпечить баланс між продуктивністю, масштабованістю та надійністю. Незважаючи на переваги мікросервісної архітектури, її реалізація пов’язана з низкою технічних і організаційних викликів.

Керування складністю системи

Мікросервіси створюють велику кількість незалежних компонентів, що потребують координації, централізованого логування, моніторингу та автоматизованого розгортання. Для підтримки стабільної роботи всієї екосистеми застосовуються такі інструменти, як *Docker*, *Kubernetes*, *Prometheus*, *Grafana* та *CI/CD* конвеєри.

Важливо також визначати чіткі контракти між сервісами та стандартизовані формати обміну даними, що дозволяє зменшити ризик несумісності та спростити масштабування системи.

Міжсервісна комунікація

Через використання протоколів *HTTP*, *gRPC* або систем повідомлень на кшталт *Kafka* виникає потреба в ретельному проєктуванні *API*, управлінні помилками, повторних запитах (*retry*) та забезпеченні ідемпотентності операцій [21][22].

Будь-яка затримка або збій одного сервісу може вплинути на загальну роботу системи, тому застосовуються механізми *fault tolerance*, такі як:

- *circuit breaker* (ізоляція несправних компонентів);
- *retry* (повторні спроби виконання запиту);
- *timeout* (обмеження часу очікування);
- *rate limiting* (контроль кількості запитів);
- *load balancing* (рівномірний розподіл навантаження).

Узгодженість даних і безпека

У розподіленому середовищі транзакції можуть охоплювати кілька сервісів, що ускладнює забезпечення атомарності операцій. Для вирішення цих проблем застосовуються патерни *SAGA* та *Eventual Consistency* [23][24].

Щодо безпеки, необхідно впровадити централізовану аутентифікацію та авторизацію, шифрування трафіку, безпечне зберігання токенів та керування правами доступу [25][26].

Крім технічних обмежень, існують і організаційні виклики, пов’язані з масштабуванням команд розробки, необхідністю узгоджених стандартів кодування та підтриманням сумісності між версіями сервісів.

Узагальнюючи, архітектурні виклики мікросервісних і розподілених систем полягають у пошуку оптимального співвідношення між гнучкістю, продуктивністю та складністю підтримки. Їх подолання потребує не лише технічних рішень, але й чіткої організації процесів розробки та експлуатації.

Висновки до розділу

У другому розділі було проаналізовано предметну область та сформульовано основні вимоги до розроблюваної системи. Обґрунтовано вибір предметної області з урахуванням актуальності задачі побудови високонавантаженої розподіленої системи та практичної доцільності її

реалізації. На основі аналізу було визначено ключові бізнес-процеси та сценарії використання, які система повинна підтримувати.

Сформульовано функціональні та нефункціональні вимоги, зокрема вимоги до продуктивності, масштабованості, надійності, безпеки та доступності системи. Проведено оцінку очікуваного навантаження, що дозволило визначити необхідність використання масштабованої архітектури та механізмів обробки великої кількості запитів у реальному часі.

Також у розділі було розглянуто основні архітектурні виклики та обмеження, пов’язані з розподіленістю системи, узгодженістю даних, затримками мережевої взаємодії та забезпеченням відмовостійкості. Результати даного аналізу стали підґрунтям для вибору мікросервісної та подійно-орієнтованої архітектури, а також визначили ключові рішення, які були реалізовані на етапах проєктування та розробки системи.

РОЗДІЛ 3
ПРОЄКТУВАННЯ СИСТЕМИ

Етап проєктування є ключовим у створенні будь-якої розподіленої системи, оскільки саме тут визначається структура компонентів, взаємодія між ними та технологічні засади реалізації. На цьому етапі формуються логічна структура сервісів, способи комунікації між ними, моделі даних, а також підходи до інтеграції та управління навантаженням.

3.1 Архітектурна діаграма системи

Архітектура розробленої системи побудована на принципах мікросервісної архітектури, де кожен сервіс відповідає за окрему бізнес-функцію, є незалежним у розгортанні та має власну базу даних. Такий підхід забезпечує можливість незалежного масштабування окремих компонентів, спрощує підтримку та підвищує стійкість системи до відмов.

Для подійно-орієнтованої взаємодії використовується брокер повідомлень, який забезпечує слабке зв’язування сервісів, зменшує залежності між ними та підвищує загальну відмовостійкість системи. Архітектурна діаграма наочно відображає структуру системи, основні компоненти та канали їх взаємодії, що спрощує розуміння логіки роботи системи, її масштабування та подальший розвиток.

На рисунку 3.1 представлено архітектурну діаграму мікросервісної системи інтернет-магазину.

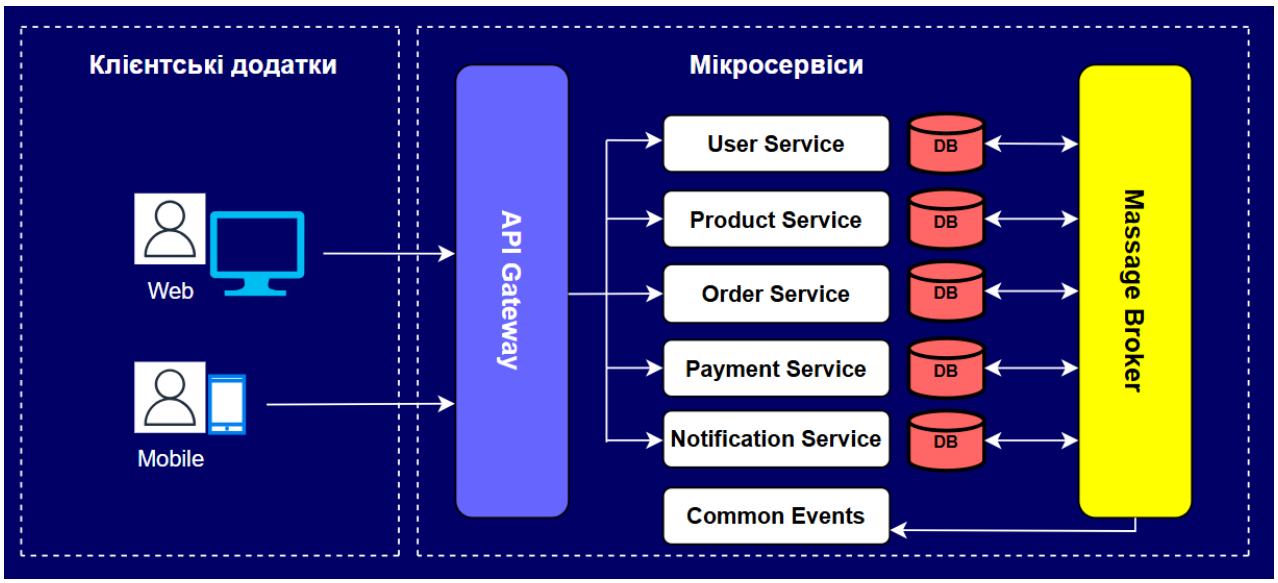

Рисунок 3.1 — Архітектурна діаграма системи

Архітектура системи складається з трьох основних рівнів:

- клієнтські додатки;
- *API Gateway;*
- мікросервісна екосистема, що взаємодіє через *Message Broker*.

Інтеракція користувачів із системою здійснюється через веб- та мобільні інтерфейси. Всі запити спрямовуються до єдиного входу — *API Gateway*, що відповідає за маршрутизацію, авторизацію, балансування навантаження та централізовану безпеку.

Такий підхід дозволяє відокремити клієнтську логіку від серверної, зменшуючи навантаження на бекенд і підвищуючи масштабованість. Для цього можуть використовуватись технології *React*, *Angular* або *Flutter* як фронтенд-рішення, які взаємодіють з бекендом через *REST* або *GraphQL API* [27].

API Gateway виступає центральним фасадом системи, через який клієнти взаємодіють із сервісами. Він приховує внутрішню структуру мікросервісної екосистеми, забезпечує перевірку *JWT*-токенів, логування запитів і контроль доступу до сервісів [28].

Центральне місце у проєктуванні займає мікросервісна екосистема, де кожен сервіс відповідає за конкретну бізнес-функцію та має власну базу даних.

Така ізоляція дозволяє сервісам бути незалежними у розгортанні та оновленні, що відповідає принципам доменного проєктування [29].

Основні компоненти екосистеми:

- *User Service* — обробка реєстрації, авторизації, управління обліковими записами та ролями користувачів;
- *Product Service* — управління товарами, категоріями, пошуком і фільтрацією;
- *Order Service* — створення, перегляд, зміна статусів і скасування замовлень;
- *Payment Service* — обробка оплат, підтвердження транзакцій і інтеграція з платіжними шлюзами;
- *Notification Service* — відправлення email- та push-сповіщень користувачам на основі подій;
- *Common Events* — обробка загальних системних подій, які впливають на кілька сервісів одночасно.

Ключовим компонентом системи є *Message Broker (Kafka)*, який реалізує асинхронну комунікацію між сервісами. Це дозволяє сервісам обмінюватися подіями у форматі *publisher–subscriber*, не створюючи жорстких залежностей між ними [30].

Наприклад, після створення замовлення *Order Service* надсилає подію *order-created*. *Payment Service* отримує цю подію та ініціює процес оплати. *Notification Service* реагує на події *order-created*, *payment-successful* або *payment-failed*, надсилаючи відповідні повідомлення користувачам.

Масштабованість такої архітектури досягається за рахунок можливості незалежного горизонтального розширення кожного сервісу залежно від навантаження.

Крім того, використання брокера повідомлень підвищує відмовостійкість системи, оскільки тимчасова недоступність окремих сервісів не призводить до втрати подій.

3.2 Сервіси та їх подієво-орієнтована взаємодія

Система реалізована за принципами, коли кожен сервіс відповідає за власну бізнес-доменну область та взаємодіє з іншими через події, що передаються за допомогою *Message Broker — Apache Kafka*. Такий підхід дозволяє забезпечити слабке зв'язування, масштабованість і стійкість до відмов, що є ключовими характеристиками розподілених систем.

Основна ідея подієво-орієнтованої взаємодії

Замість безпосереднього виклику методів між сервісами (через *REST* чи *gRPC*), сервіси обмінюються івентами, які описують певні зміни стану системи.

Наприклад, коли користувач реєструється — публікується подія *UserRegistered*; коли замовлення створене — публікується *OrderCreated*; коли платіж успішно проведено — *PaymentProcessed*.

Інші сервіси підписуються на ці події та реагують на них відповідними діями.

Основні компоненти системи

Серед основних компонентів також є додаткові, наведені нижче:

- *common-events* — спільний модуль, що містить модель подій (DTO), конфігурації Kafka Topics та серіалізатори/десеріалізатори;
- *docker-compose.yml* — використовується для підняття всієї інфраструктури, включно з *Kafka, Zookeeper* і сервісами.

User Service у процесі своєї роботи сервіс публікує події, які інформують інші компоненти системи про зміну стану користувача. Отримувачами цих подій є сервіс сповіщень, який використовується для надсилання інформаційних повідомлень користувачам, а також сервіс замовлень, що може використовувати ці дані для подальшої персоналізації або формування історії взаємодії з користувачем.

Product Service використовуються сервісом замовлень для актуалізації даних про товари у процесі оформлення замовлень, а також сервісом сповіщень для інформування користувачів про зміни або появу нових товарів.

Order Service ініціює події, пов’язані зі зміною статусу замовлення, які передаються до сервісу оплати для запуску платіжного процесу, до сервісу сповіщень для інформування користувача та до сервісу товарів для резервування або оновлення кількості продукції на складі.

Payment Service реагує на події, пов’язані зі створенням замовлення, після чого виконує оплату та публікує результат виконання операції. Інформація про успішну або неуспішну оплату передається до сервісу замовлень для оновлення його статусу, а також до сервісу сповіщень для повідомлення користувача про результат платіжної операції.

Notification Service призначений для централізованого надсилання *email* та *push*-повідомлень користувачам. Він підписується на події, що генеруються іншими сервісами, та на їх основі формує відповідні повідомлення. Такий підхід дозволяє винести логіку інформування користувачів в окремий компонент і забезпечити масштабованість та гнучкість системи загалом.

На рисунку 3.2 зображено структуру проєкту.

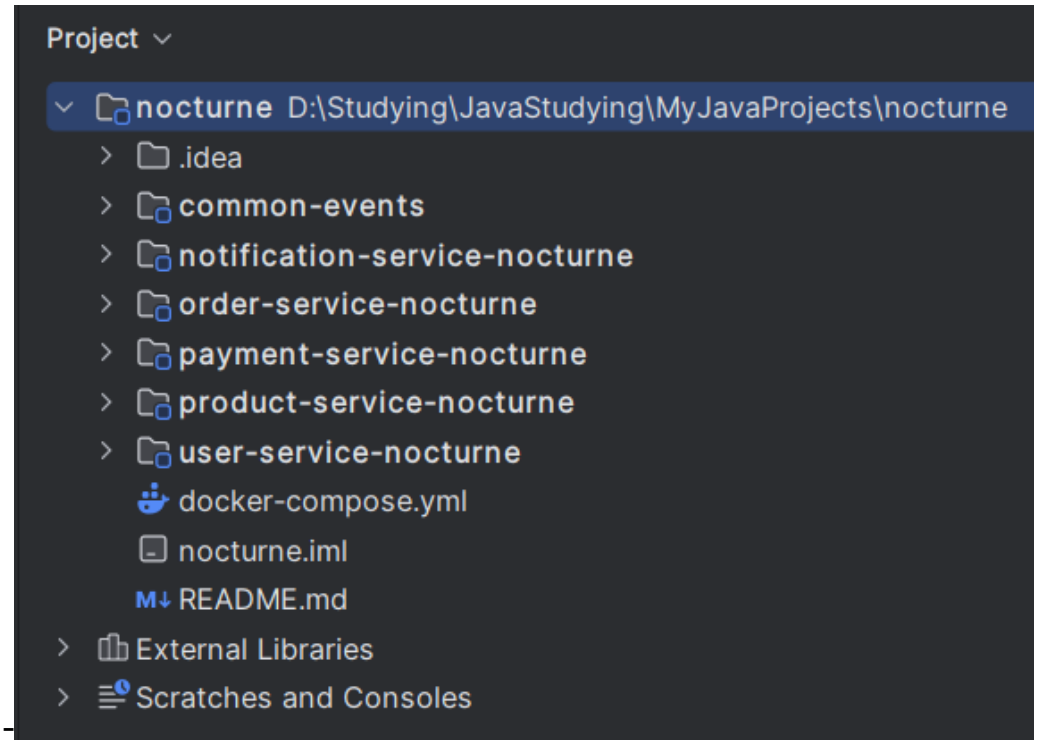

Рисунок 3.2 — Мікросервіси системи

Нижче наведена таблиця 3.1 з основними принципами взаємодії сервісів.

Таблиця 3.1 — Основні принципи взаємодії сервісів

Принцип	Опис
Слабке зв’язування	Сервіси не знають внутрішню реалізацію один одного, лише обмінюються подіями.
Незалежне масштабування	Кожен сервіс можна масштабувати окремо без впливу на інші.
Надійність	Якщо один сервіс недоступний, події залишаються в Kafka і будуть оброблені пізніше.
Ідемпотентність	Повторна обробка події не змінює результат (запобігання дублюванню операцій).
Спостережуваність	Використання traceId і correlationId для відстеження повного бізнес-потоку.

Подієво-орієнтована взаємодія між мікросервісами *User Service, Product Service, Order Service, Payment Service і Notification Service* за допомогою *Kafka* дозволяє побудувати асинхронну, стійку та масштабовану екосистему. Такий підхід зменшує жорстку зв’язаність між компонентами, підвищує ізоляцію даних і спрощує масштабування та еволюцію кожного мікросервісу незалежно від інших.

3.3 Проєктування баз даних

Кожен мікросервіс у системі розробляється з принципом *Database per Service*. Це означає, що кожен мікросервіс володіє власною базою даних, яка повністю відповідає за зберігання лише свого доменного контексту. Усі взаємодії між сервісами здійснюються асинхронно через брокер повідомлень, що усуває необхідність прямих *SQL*-запитів до чужих баз.

User Service

Основні таблиці:

- *app_user* — містить дані користувачів: електронну пошту (як первинний ключ), ім’я користувача, пароль, роль, статус верифікації та зображення профілю;
- *verification_token* — зберігає токени підтвердження електронної пошти, пов’язані з користувачем через зовнішній ключ *user_email*.

На рисунку 3.3 зображено скріншот діаграми бази даних користувачів.

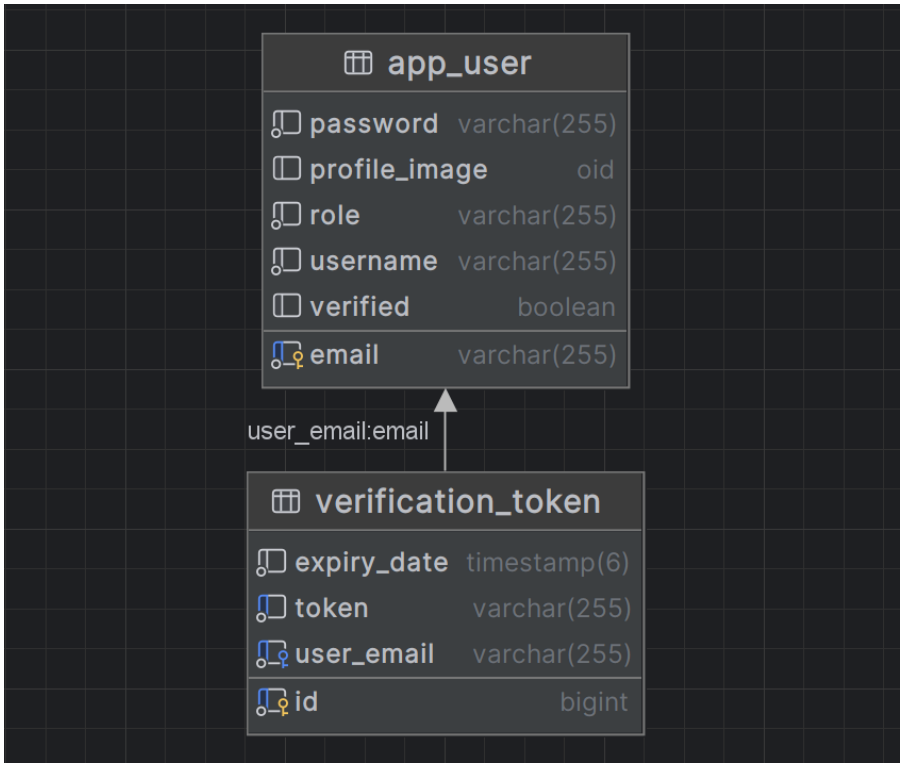

Рисунок 3.3 — Скріншот діаграми бази даних користувачів

Product Service

Основні таблиці:

- *category* — ієрархічна структура категорій із рекурсивним зв’язком parent_id → category.id;
- *product* — описує основні атрибути товару (назва, опис, ціна, кількість, статус доступності, посилання на категорію);

- *product_image* — містить URL-посилання на зображення, зв’язані з продуктом.

На рисунку 3.4 зображено скріншот бази даних категорій та товарів.

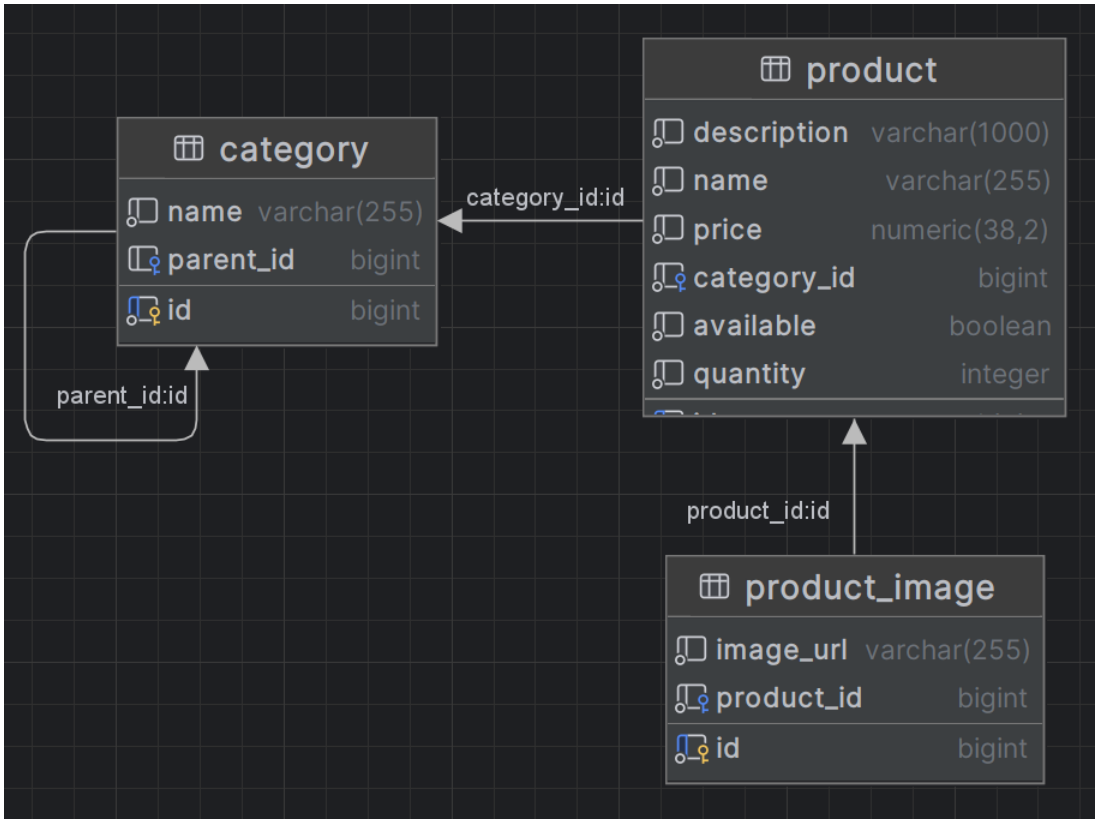

Рисунок 3.4 — Скріншот діаграми бази даних категорій та товарів

Ключові особливості: при зміні кількості товару або його статусу сервіс генерує події *ProductUpdatedEvent*, які можуть споживатися *Order Service*: забезпечується консистентність інвентарю через подійно-орієнтовану модель.

Order Service

Основні таблиці:

- *order_table* — зберігає загальну інформацію про замовлення: дату, статус, електронну пошту користувача;
- *order_item* — описує товари, що входять до складу конкретного замовлення, із фіксацією ціни на момент покупки.

На рисунку 3.5 зображено скріншот діаграми бази даних замовлень.

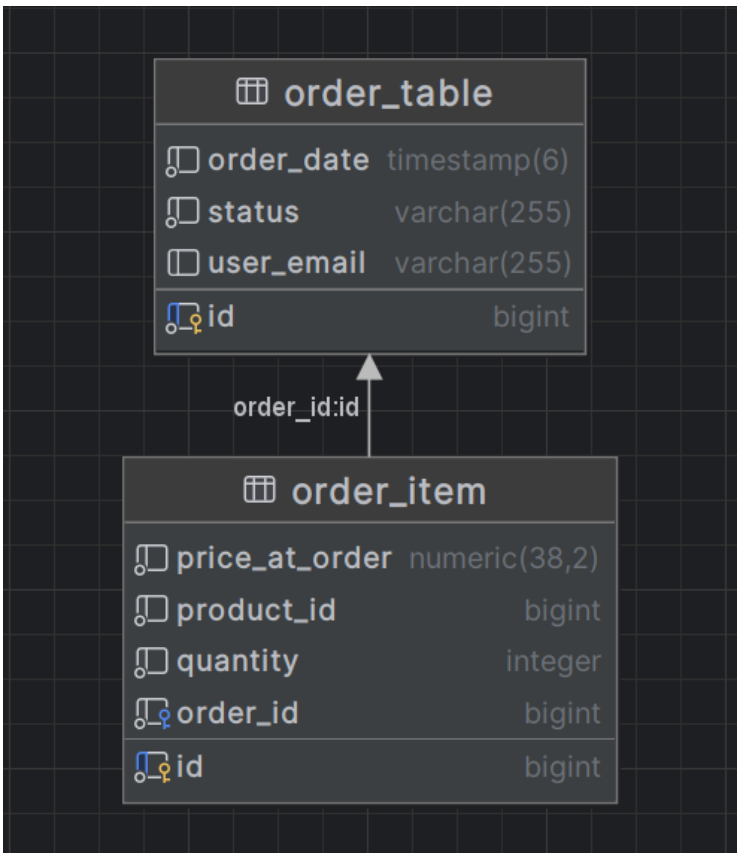

Рисунок 3.5 — Скріншот діаграми бази даних замовлень

Ключові особливості: таблиця *order_table* має обмеження *CHECK*, що визначає допустимі статуси (від *NEW* до *COMPLETED*); при створенні нового замовлення генерується подія *OrderCreatedEvent*, яку споживає *Payment Service* для ініціації платежу.

Payment Service

Основна таблиця: *payment* — містить суму платежу, метод оплати (*CREDIT_CARD*, *PAYPAL*, *BANK_TRANSFER*, *MOCK*), час створення, статус (*PENDING*, *SUCCESS*, *FAILED*) і посилання на відповідне замовлення.

Ключові особливості: зміна статусу платежу (наприклад, *SUCCESS*) породжує подію *PaymentCompletedEvent*, яку споживає Order Service для оновлення стану замовлення; підтримується історія оновлень через поле *updated_at*.

На рисунку 3.6 зображено скріншот діаграми бази даних сплати.

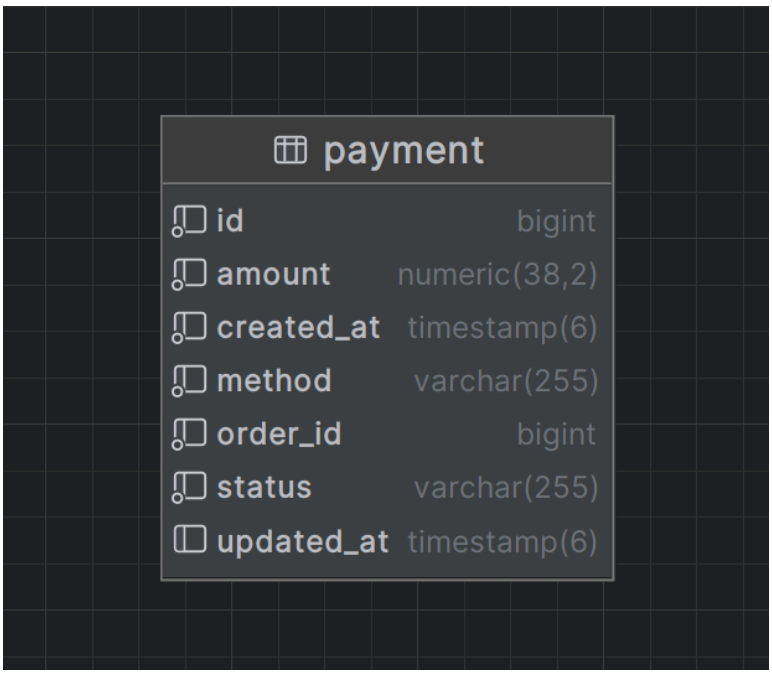

Рисунок 3.6 — Скріншот діаграми бази даних сплати

NotificationService

Notification Service не має власної бази даних, оскільки його логіка зосереджена на споживанні повідомлень з *Kafka* та зовнішній інтеграції з поштовими сервісами.

Підхід до проєктування баз даних із принципом *Database per Service* забезпечує високу ізоляцію даних і незалежність кожного мікросервісу. Завдяки цьому архітектурному рішенню усуваються прямі зв’язки між сервісами на рівні баз даних, що знижує ризики міжсервісних конфліктів і спрощує модифікацію структури даних у межах одного домену. Кожен сервіс володіє повною автономією у виборі типу сховища, оптимального для його задач — реляційного або документно-орієнтованого, що підвищує гнучкість системи в цілому.

Завдяки подійно-орієнтованій взаємодії через брокер повідомлень підтримується узгодженість бізнес-процесів без необхідності централізованих транзакцій. Замість складних синхронних викликів між сервісами, система використовує асинхронні події, які зберігають послідовність дій та дозволяють сервісам реагувати лише на ті події, що стосуються їхнього контексту.

3.4 Безпека та авторизація

У системі безпека відіграє ключову роль, оскільки вона забезпечує захист даних користувачів і контроль доступу до різних частин функціоналу. У межах мікросервісної архітектури безпека реалізується на рівні кожного сервісу, що дозволяє уникнути централізованих точок вразливості та забезпечити незалежність у перевірці доступу.

Основним механізмом аутентифікації та авторизації обрано *JWT* (*JSON Web Token*). Цей підхід забезпечує зручну і масштабовану модель управління доступом, у якій після успішної аутентифікації користувач отримує токен із зашифрованою інформацією про свою особу та роль у системі. *JWT*-токен додається до кожного запиту до мікросервісів через *HTTP*-заголовок *Authorization*, що дозволяє сервісам перевіряти дійсність токена без необхідності постійного звернення до централізованої бази користувачів [31].

На рисунку 3.7 зображена структура *JWT*-токена.

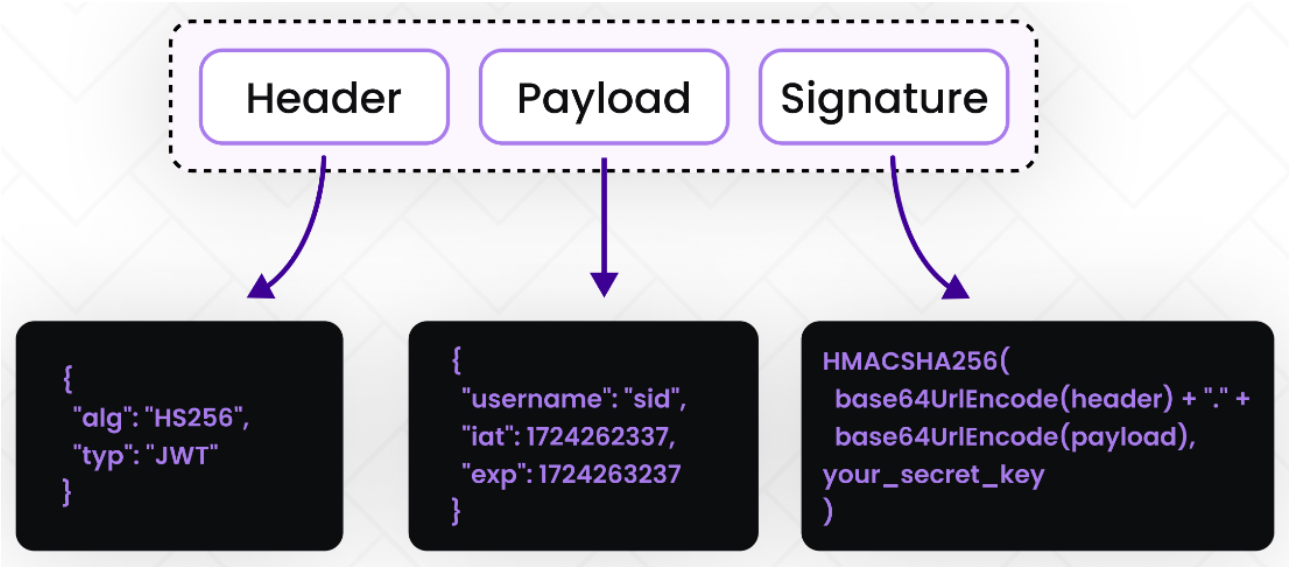

Рисунок 3.7 — Структура *JWT*-токена

Для забезпечення балансу між безпекою та зручністю користування система використовує два типи *JWT*-токенів — *access token* і *refresh token*.

Access token має короткий термін дії (наприклад, 15–30 хвилин) і використовується для доступу до більшості захищених ресурсів системи.

Його короткий життєвий цикл знижує ризик компрометації, адже навіть у разі викрадення токен швидко стає недійсним.

Refresh token має довший термін дії (зазвичай від кількох годин до кількох днів) і зберігається на клієнті або в безпечному. Після закінчення дії *access token* користувач може отримати новий, надіславши *refresh token* на спеціальний *endpoint*. Це дозволяє підтримувати безперервну автентифікацію без повторного входу в систему.

Для підвищення безпеки *refresh token* може бути відкликано вручну або автоматично, наприклад, у разі виходу користувача із системи чи виявлення підозрілої активності. Такий підхід запобігає несанкціонованому доступу до системи та забезпечує контрольований життєвий цикл сесій.

Не менш важливою частиною є ролі користувачів (наприклад, *ROLE_USER, ROLE_ADMIN, ROLE_SELLER*), вони визначають рівень доступу до ресурсів і функцій. Наприклад, звичайний користувач може переглядати товари та створювати замовлення, продавець — додавати чи оновлювати товари, а адміністратор має повний контроль над усіма аспектами системи. Такий підхід базується на принципі *Role-Based Access Control (RBAC)*, що спрощує управління дозволами та підвищує безпеку системи [32].

У контексті мікросервісної архітектури аутентифікація виконується централізовано в *User Service*, який відповідає за створення та перевірку токенів. Інші сервіси, отримуючи запити з валідними *JWT*-токенами, самостійно перевіряють їхню підпис і вбудовану роль користувача, не залежачи від інших компонентів.

Таким чином, використання *JWT* забезпечує масштабовану, безпечну та розподілену модель аутентифікації, яка гармонійно поєднується з мікросервісною архітектурою та сприяє побудові стійкої до збоїв і безпечної екосистеми.

3.5 Моніторинг та логування

Ефективний моніторинг і логування є невід'ємною частиною будь-якої розподіленої системи, особливо у мікросервісній архітектурі, де кожен сервіс функціонує автономно, але взаємодіє з іншими компонентами через мережу. Їх головна мета — забезпечити спостережуваність системи: можливість розуміти, що відбувається у кожному сервісі, виявляти збої, аналізувати продуктивність та вчасно реагувати на проблеми.

Моніторинг у мікросервісній архітектурі зазвичай реалізується через збір метрик і логів. Найкращою практикою є впровадження централізованої системи моніторингу, яка агрегує дані від усіх сервісів у єдиному інтерфейсі.

Prometheus — система збору та зберігання метрик з мікросервісів. Кожен сервіс експортує свої показники (*CPU*, пам'ять, кількість запитів, затримка відповіді тощо) через /actuator/prometheus.

Grafana — візуалізаційна платформа, що підключається до Prometheus і дозволяє створювати дашборди для реального часу, які показують стан кожного мікросервісу [33].

Alertmanager — використовується для надсилання сповіщень про критичні події (наприклад, падіння сервісу або перевищення часу відповіді) до *Slack,* електронної пошти або *Telegram*.

Важливо, щоб усі метрики мали єдині стандарти іменування та мітки — це спрощує аналіз і фільтрацію даних.

Логування має бути структурованим**,** централізованим і контекстним. Кожен мікросервіс повинен формувати логи у форматі *JSON*, що дозволяє легко їх обробляти та шукати потрібні події.

Для централізованого збору логів зазвичай використовують *ELK*-стек (*Elasticsearch, Logstash, Kibana*) або його сучаснішу версію *EFK* (*Elasticsearch, Fluentd, Kibana*):

- *Elasticsearch* зберігає та індексує логи;

- *Logstash* або *Fluentd* збирають їх із контейнерів (наприклад, *Docker*) і передають далі;
- *Kibana* забезпечує пошук, фільтрацію та візуалізацію логів у зручному веб-інтерфейсі.

Для самої системи логування в *Spring Boot* зазвичай використовується *SLF4J* із реалізацією *Logback*, який забезпечує продуктивне та структуроване логування. Усі повідомлення мають бути стандартизованими (наприклад, у форматі *JSON*) та містити *traceId* — унікальний ідентифікатор запиту, що передається між мікросервісами через *Kafka* або *REST*. Це дозволяє відстежити шлях запиту по всій системі, що є важливою частиною *distributed tracing*.

Завдяки впровадженню систем моніторингу та централізованого логування мікросервісна архітектура набуває високої спостережуваності та керованості. Це дозволяє оперативно реагувати на збої, аналізувати роботу окремих компонентів, підвищувати продуктивність системи та забезпечувати стабільність її функціонування навіть під високим навантаженням.

Висновки до розділу

У третьому розділі було виконано проєктування архітектури розподіленої високонавантаженої системи з урахуванням визначених на попередніх етапах вимог та обмежень. Розроблена архітектурна модель базується на принципах мікросервісного підходу та подійно-орієнтованої взаємодії, що забезпечує слабку зв’язаність компонентів, гнучкість системи та можливість її незалежного масштабування.

У межах розділу визначено склад ключових сервісів і механізми їх взаємодії через події, що дозволяє ефективно координувати бізнес-процеси без жорстких синхронних залежностей. Окрему увагу приділено проєктуванню баз даних для кожного сервісу, що відповідає принципу ізоляції даних та підвищує надійність і стійкість системи до збоїв. Також розглянуто підходи до

забезпечення безпеки та авторизації користувачів, які гарантують захист даних і контроль доступу на рівні сервісів.

Крім того, у розділі визначено підходи до моніторингу та логування, які є критично важливими для експлуатації розподілених систем, забезпечуючи спостережуваність, своєчасне виявлення проблем та аналіз роботи сервісів. У результаті проведеного проєктування сформовано цілісну архітектурну основу системи, яка створює передумови для її ефективної реалізації, тестування та подальшого розвитку.

РОЗДІЛ 4
РЕАЛІЗАЦІЯ СИСТЕМИ

У цьому розділі описано практичні аспекти розробки системи, побудованої на основі мікросервісної архітектури. Основна увага приділяється вибору технологічного стеку, реалізації ключових сервісів, забезпеченню взаємодії між ними, а також впровадженню засобів безпеки, моніторингу та логування.

4.1 Вибір стеку технологій

Для реалізації системи було обрано сучасний та перевірений стек технологій, який забезпечує надійність, гнучкість, безпеку та можливість масштабування. Основним середовищем розробки є *Java*, що є однією з найпопулярніших мов програмування для корпоративних і розподілених систем.

Фреймворк *Spring Boot* використовується для швидкої розробки мікросервісів, надаючи зручні інструменти для конфігурації, запуску та тестування. *Spring Security* відповідає за реалізацію механізмів безпеки, включаючи аутентифікацію та авторизацію користувачів, а *JWT (JSON Web Token)* використовується для передачі даних про користувача між сервісами у захищеному вигляді.

Для взаємодії з базами даних використовується *Spring Data JPA*, що забезпечує абстракцію над *SQL*-запитами та спрощує роботу з персистентними даними. Як система управління базами даних обрано *PostgreSQL*, яка є потужною, відкритою та оптимізованою для складних транзакційних операцій.

Обмін подіями між мікросервісами здійснюється за допомогою *Apache Kafka* — високопродуктивного брокера повідомлень, який забезпечує надійність та асинхронну комунікацію між сервісами. Це дозволяє будувати подієво-орієнтовану архітектуру, де сервіси реагують на події, не потребуючи прямих синхронних викликів.

Для контейнеризації застосунків використовується *Docker*, що дозволяє ізолювати кожен мікросервіс у власному середовищі та забезпечує зручне розгортання системи в будь-якому середовищі — локальному чи хмарному. Керування залежностями та збіркою проєкту виконується за допомогою *Maven*, а контроль версій — через *Git*, що забезпечує ефективну командну роботу та відстеження змін у коді.

Для реалізації моніторингу та логування застосовуються *Spring Boot Actuator, Prometheus* і *Grafana* для збору та візуалізації метрик, а також *ELK Stack (Elasticsearch, Logstash, Kibana)* для централізованого збору логів. Це дозволяє розробникам і адміністраторам виявляти проблеми в реальному часі, аналізувати продуктивність системи та забезпечувати стабільну роботу мікросервісної екосистеми.

Таким чином, обраний стек технологій забезпечує повний цикл розробки — від створення і тестування окремих сервісів до їх розгортання, моніторингу та масштабування в середовищі реального використання.

4.2 Реалізація ключових мікросервісів

У цьому підрозділі розглядається практична реалізація ключових мікросервісів розробленої системи, кожен з яких відповідає за окрему бізнес-функцію та функціонує як незалежний компонент.

4.2.1 Реалізація *User Service*

Нижче наведено опис реалізації *User Service* — одного з ключових мікросервісів системи. У матеріалі поєднано архітектурні рішення, пояснення реалізованих механізмів (аутентифікація/авторизація, верифікація email,

публікація подій у *Kafka*, робота з БД) та витягнуті з проєкту фрагменти коду з поясненнями їх призначення.

User Service реалізовано з урахуванням принципів *Single Responsibility* та *Database per Service*: сервіс володіє власною базою даних (таблиці *app_user*, *verification_token*), надає *REST API* для роботи з користувачами і публікує події в *Kafka* (топіки *email-verification*, *user-registered*). Безпека базується на *JWT* і *Spring Security*; фільтрація запитів виконується *JwtFilter*.

Модель предметної області — сутності (*Entity*), *User* — основна сутність користувача, наведена нижче.

```
@Id
@Column(nullable = false, unique = true)
String email;
@Column(nullable = false)
String username;
@Column(nullable = false)
String password;
@Column(nullable = false)
@Enumerated(EnumType.STRING)
Role role;
```

Сутність *User* використовує email як первинний ключ (*PK*). Поле role зберігається як *Enum* (*Role*), що полегшує реалізацію *RBAC*. Прапорець *verified* визначає, чи підтверджена електронна пошта — це важливий атрибут, що впливає на доступ.

Role — перелік ролей, код наведений нижче.

```
ROLE_USER, ROLE_ADMIN, ROLE_SELLER
```

Ролі використовуються при формуванні *GrantedAuthority* для *Spring Security*.

Клас *UserServiceImpl* містить основну бізнес-логіку: реєстрація користувача, створення верифікаційного токену, підтвердження акаунта, аутентифікація (через *AuthenticationManager*) і адміністративні дії (отримання списку, оновлення, видалення).

Реєстрація (*signUpUser*): перевірка унікальності *email*; хешування пароля через *PasswordEncoder* (*BCrypt*); збереження *User* у БД; створення *VerificationToken* і публікація події *VerificationEmailEvent* у *Kafka* (*Notification Service* відправить лист).

Фрагмент (логіка викликів описана в коді) наведено нижче.

```
VerificationToken vt =
verificationTokenService.createTokenForUser(savedUser);
// Відправляємо верифікаційний лист
VerificationEmailEvent verificationEvent =
VerificationEmailEvent.builder().email(savedUser.getEmail()).u
sername(savedUser.getUsername()).token(vt.getToken())
.verificationUrl("http://localhost:3000/verify?token=" +
vt.getToken()).build();
userEventProducer.publishVerificationEmail(verificationEvent);
```

Аутентифікація / авторизація

JwtService — генерація і валідація токенів, код наведений нижче.

```
public String generateToken(String email, String role) {
Map<String, Object> claims = new HashMap<>();
claims.put("role", role); long expirationTimeMillis = 1000 *
60 * 60 * 24; // 24 hours
```

JwtService формує токен із роллю як додатковою претензією (*claim*). Термін життя токена — 24 години (за поточною конфігурацією). Секретний ключ читається з конфігурації (*jwt.secret*) і використовується для підпису *HMAC-SHA*.

SecurityConfig — конфігурація безпеки, фрагмент коду наведений нижче.

```
.requestMatchers("/api/users/verify/**").permitAll().anyReques
t().authenticated().httpBasic(Customizer.withDefaults()).sessi
onManagement(session -> session.sessionCreationPolicy
(SessionCreationPolicy.STATELESS).addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class).authenticationProv
ider(authenticationProvider()).build();
```

Конфігурація дозволяє публічні ендпойнти для реєстрації, логіну, верифікації та підключає *JwtFilter* до фільтрації запитів.

Інтеграція з *Kafka* — публікація подій

KafkaConfig — producer factory і *KafkaTemplate*, код наведений нижче.

```
public ProducerFactory<String, Object> producerFactory(){
Map<String, Object> configProps = new HashMap<>();
configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");configProps.put(ProducerConfig.KEY_SERIALIZ
ER_CLASS_CONFIG
```

Конфігурація забезпечує серіалізацію повідомлень у форматі *JSON* (*JsonSerializer*).

UserEventProducer — продюсер подій, код наведений нижче.

```
log.info("Publishing UserRegisteredEvent for email={}",
event.getEmail());
kafkaTemplate.send("user-registered", event);}
public void publishVerificationEmail(VerificationEmailEvent
event) {log.info("Publishing VerificationEmailEvent for
email={}", event.getEmail()); kafkaTemplate.send("email-
verification", event.getEmail(), event);
```

Producer відправляє події у відповідні топіки. Виклик *publishVerificationEmail* використовується відразу після реєстрації, а *publishUserRegistered* — після успішного підтвердження email.

***REST API* — контролер користувачів**

Контролер *UserController* реалізує ендпойнти для реєстрації, логіну, логауту, підтвердження токена, читання/оновлення профілю (код нижче).

```
@PostMapping("/signup")
public ResponseEntity<UserResponseDTO> signup(@Valid
@RequestBody UserSignUpRequestDTO requestDTO) { … }
@PostMapping("/login")
public ResponseEntity<AuthResponseDTO> login(@RequestBody
AuthRequestDTO authRequestDTO) { … }
@PostMapping("/logout")
public ResponseEntity<Void> logout(HttpServletRequest request)
{ … }
```

Ендпойнт *logout* реалізовано через механізм чорного списку токенів (*TokenBlacklistService*), що дозволяє інвалідувати JWT до закінчення його терміну дії.

4.2.2 Реалізація *Product Service*

Нижче подано опис реалізації *Product Service* — сервісу, що відповідає за каталог товарів, категорії, зображення, управління запасами та інтеграцію з іншими сервісами через *Kafka*.

Архітектурне призначення

Product Service реалізує домен «товари» і відповідає за:

- *CRUD* для товарів та категорій;
- зберігання метаданих зображень товарів;
- управління кількістю на складі (*reserve / release*);
- перевірку наявності товару у відповідь на зовнішні запити (через *Kafka*) і резервування (*atomic stock change*);

- публікацію/реакцію на події товарного домену.

Сервіс ізольовано зберігає свої дані у власній БД (таблиці *product, category, product_image*) і обмінюється інформацією з рештою системи асинхронно через *Kafka* (теми *product-check, product-check-response, stock-release*).

Сервісний шар — бізнес-правила та валідація

Клас *CategoryServiceImpl* забезпечує створення, оновлення та видалення категорій.

У реалізації присутні:

- контроль прав доступу *(isAdmin(authentication));*
- перевірка дублювання (*existsByName*);
- коректне встановлення батьків для підкатегорій;
- валідація (не дозволяється зробити категорію батьком самої себе).

Такий підхід гарантує коректність структури категорій на рівні бізнес-логіки до звернення до БД.

ProductServiceImpl — логіка роботи з товарами.

Основні моменти реалізації:

- валідація вхідних даних (ім'я не пусте, опис, позитивна ціна);
- перевірка прав (тільки адміністратори створюють, оновлюють, видаляють товар);
- при наявності *category.id* виконується пошук категорії в БД і встановлення посилання;
- методи *reserveStock* і *releaseStock* працюють через репозиторій (нижній рівень, який реалізує атомарні *SQL*-операції);
- пошук через *searchProducts*, *getProductsByCategory*, *getAllProducts*.

Фрагмент коду ключових перевірок у *createProduct*/*updateProduct* нижче.

```
if (!isAdmin(authentication))
    throw new CustomAccessDeniedException("Only admin can
create product.");
if (productRepository.existsByName(product.getName())) {
```

```
    throw new DuplicateResourceException("Product with name '"
+ product.getName() + "' already exists.");}
```

***REST API* — контролери**

Контролери *CategoryController*, *ProductController*, ProductImageController надають стандартні CRUD-ендпойнти. Фрагмент коду наведений нижче.

```
public ResponseEntity<ProductResponseDTO>
createProduct(@RequestBody ProductRequestDTO dto,
Authentication authentication) {Product product =
productService .createProduct(productMapper.toEntity(dto),
authentication);return
ResponseEntity.status(HttpStatus.CREATED).body(productMapper.t
oDto(product));}
```

Інтеграція через *Kafka*

Фрагмент коду конфігурації *Kafka* нижче.

```
//ProducerFactory і KafkaTemplate<ProductCheckMessage>
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);// ConsumerFactory для
ProductCheckMessage
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "product-service-
group");
```

StockReleaseListener — звільнення запасів (фрагмент коду).

```
@KafkaListener(topics = "stock-release", groupId = "product-
service-group")
```

```
public void onRelease(ProductCheckMessage request) {
    productService.releaseStock(request.getProductId(),
request.getQuantity());
}
```

Слухач отримує повідомлення про необхідність повернути резерв (наприклад, коли замовлення скасовано або платіж не пройшов) і викликає *releaseStock*.

4.2.3 Реалізація *Order Service*

Order Service відповідає за управління життєвим циклом замовлення: прийом і валідація запитів на створення, координацію резервації товарів на складі, оновлення статусів замовлень у відповідь на результати платежів, а також обробку операцій скасування і видалення. У реалізації поєднано синхронну *REST*-логіку (контролери → сервіси → репозиторії) з асинхронною координацією через *Kafka* (*request–reply* для перевірки товарів, події *order-created*, прослуховування результатів платежів). Далі наведено докладний опис ключових компонентів, сценаріїв та важливих проектних рішень, а також рекомендації щодо підвищення надійності та масштабованості.

Загальний сценарій створення замовлення (*createOrder*)

Реалізація *createOrder* демонструє використання *Kafka* для перевірки наявності товарів у *Product Service*.

Основні кроки процедури:

1) Клієнт робить *POST /api/orders* з *OrderRequestDTO* (список позицій);

2) *Order Service* для кожної позиції відправляє асинхронний запит на резервування товару через *ProductCheckProducer.check(productId, quantity)*. Метод повертає *CompletableFuture<ProductCheckMessage>;*

3) *Order Service* збирає всі *CompletableFuture* і чекає відповіді з таймаутом (*3s*). Якщо будь-яка відповідь показує *available = false* або майбутнє завершується з помилкою/таймаутом — вже зарезервовані позиції звільняються (*releaseReservedStock*) та операція відкатується;

4) Якщо всі товари успішно зарезервовано, створюється сутність Order з елементами *OrderItem* (ціна береться з відповіді *product-check*), зберігається в БД і публікується подія *OrderCreatedEvent* у топік *order-created*.

Компоненти асинхронної взаємодії

ProductCheckProducer — відправник запитів на перевірку. *Producer* створює *ProductCheckMessage* з унікальним *correlationId*, реєструє *future* у *ResponseStorage*, відправляє повідомлення і повертає *CompletableFuture*.

Фрагмент коду *ProductCheckProducer* нижче.

```
public CompletableFuture<ProductCheckMessage> check(Long id,
int qty) String corr = UUID.randomUUID().toString();
ProductCheckMessage msg =
ProductCheckMessage.builder().correlationId(corr).productId(id
).quantity(qty).available(false).build();
return fut.orTimeout(3, TimeUnit.SECONDS);
```

Таймаут тут критичний: якщо відповідь не надійде вчасно, *future* завершується з *TimeoutException* — сервіс виконає відкат.

ResponseStorage — тимчасове очікування відповідей (код нижче).

```
public void register(String id,
CompletableFuture<ProductCheckMessage> future)
{waiting.put(id, future);}
    public void complete(String id, ProductCheckMessage resp){
        Optional.ofNullable(waiting.remove(id)).ifPresent(f ->
f.complete(resp));}}
```

ResponseStorage пов'язує correlationId із клієнтським *future*. *Listener*, який приймає responses, використовує *storage.complete(corr, resp)*.

Обробка результатів платежів (*payment events*)

Order Service слухає дві теми: *payment-processed* і *payment-failed*. Лістенери делегують логіку до *OrderServiceImpl*, як у коді нижче.

```
    public void onPaymentProcessed(PaymentProcessedEvent
event) {orderService.handlePaymentResult(event);}
    public void onPaymentFailed(PaymentFailedEvent event) {
        orderService.handlePaymentFailed(event);}
```

Отже, *handlePaymentResult* оновлює статус замовлення до *PAID*; *handlePaymentFailed* переводить у *WAITING_FOR_PAYMENT*. У реальних системах також варто публікувати події (audit trail) та ініціювати компенсаційні дії у разі невдач.

Скасування замовлення і звільнення резервів

Метод *cancelOrder* оновлює статус на *CANCELLED* і відправляє повідомлення *stock-release* для кожного елементу замовлення, приклад нижче.

```
for (OrderItem item : order.getItems()) {
    productCheckProducer.release(item.getProductId(),
item.getQuantity());} order.setStatus(OrderStatus.CANCELLED);
return orderRepository.save(order);
```

Product Service слухає *stock-release* і викликає *releaseStock* — таким чином резерв повертається.

Контролери та інтерфейс *API*

Контролер *OrderController* надає *REST*-ендпойнти для створення, перегляду, скасування і зміни статусу замовлення. Контроль доступу здійснюється через *Authentication* і ролі (адміністратор може переглядати всі замовлення).

4.2.4 Реалізація *Payment Service*

Payment Service відповідає за прийом подій про створення замовлення, ініціалізацію та обробку платежів, збереження історії транзакцій та інформування інших сервісів про результат оплати (успіх або невдача). У реалізації використано підхід подієвої взаємодії: *Order Service* публікує

OrderCreatedEvent, *Payment Service* його споживає, обробляє платіж (у поточному прототипі — емітація) і публікує *PaymentProcessedEvent* або *PaymentFailedEvent*.

Обробка події створення замовлення — *createAndProcessPayment*

Payment Service підписаний на топік *order-created*. Ключовий сервісний метод створює запис про платіж, емітує його обробку, зберігає результат і публікує відповідну подію.

Фрагмент коду реалізації *Payment Service.*

```
payment = paymentRepository.save(payment);
PaymentStatus result = processPaymentMock(payment);
payment.setStatus(result);
payment.setUpdatedAt(LocalDateTime.now());
paymentRepository.save(payment);
if (result == PaymentStatus.SUCCESS) {
eventProducer.publishPaymentProcessed(response);} else {
eventProducer.publishPaymentFailed(failedEvent);
```

Імітація платіжного шлюзу

У прототипі обробка платіжної транзакції реалізована методом *processPaymentMock*, код наведено нижче.

```
if (payment.getMethod() == PaymentMethod.MOCK)
return PaymentStatus.SUCCESS;
return new Random().nextInt(10) < 8 ? PaymentStatus.SUCCESS :
PaymentStatus.FAILED;
```

Для демонстрації та тестування цього достатньо, але для продакшену потрібно:

- інтегрувати реальний платіжний шлюз (*Stripe*, *Adyen*, *PayPal* тощо) через *SDK/webhook*;

- підтримати асинхронні *webhook*-и (платіж може проходити поза часовим вікном процесингу);
- зберігати *transactionId*, відповідь шлюзу, коди помилок і деталі для аудиту.

Публікація результатів — *PaymentEventProducer*

Після обробки платіжна служба інформує інші сервіси, код нижче.

```
public void publishPaymentProcessed(PaymentProcessedEvent
event) { kafkaTemplate.send("payment-processed",
String.valueOf(event.getOrderId()), event); public void
publishPaymentFailed(PaymentFailedEvent event) {
kafkaTemplate.send("payment-failed",
String.valueOf(event.getOrderId()), event);
```

Публікація з ключем *orderId* забезпечує кореляцію і рівномірне розподілення повідомлень по partition. Order Service підписаний на ці топіки і відповідно оновлює статус замовлення (*PAID* або *WAITING_FOR_PAYMENT*) та ініціює подальші дії (обробка доставки, повідомлення користувача).

***API* для адміністрації та перегляду платежів**

REST-контролер дозволяє:

- отримати платіж за *id* (тільки адміністратор або власник);
- отримати всі платежі по *orderId* (перевіряючи, чи має запитувач права);
- адміністратор може змінювати статус платежу вручну (корисно для адмін-інтервенції).

Це реалізовано в зручному вигляді через *PaymentController*, який використовує *PaymentMapper* для перетворення сутностей у *DTO*.

***Kafka*-конфігурація і лістенер**

Payment Service підключається до *Kafka* наступними компонентами:

- *ConsumerFactory* + *ConcurrentKafkaListenerContainerFactory* для *OrderCreatedEvent*;
- *KafkaTemplate* для публікації *payment-processed* / *payment-failed*.

Лістенер обробляє *OrderCreatedEvent* і викликає *createAndProcessPayment*, як показано на фрагменті коду.

```
public void onOrderCreated(OrderCreatedEvent event) {
paymentService.createAndProcessPayment(event);
```

Реалізований *Payment Service* забезпечує повний життєвий цикл платіжної операції в рамках подієво-орієнтованої мікросервісної екосистеми: прийняття *OrderCreatedEvent*, створення запису про платіж, імітація/обробка транзакції, збереження результату і повідомлення інших сервісів про результат. У поточному прототипі приділено увагу ідемпотентності (перевірка існуючих платежів) та коректній публікації подій.

4.2.5 Реалізація *Notification Service*

Notification Service виконує роль "інформаційного шлюзу" — приймає події з *Kafka* (реєстрація користувача, створення замовлення, результати платежів, верифікація *e-mail*) і відправляє відповідні повідомлення користувачам на електронну пошту.

***EmailService* — відправка листів**

Нижче наведено фрагмент коду реалізації відравки листів користувачам.

```
MimeMessage message = mailSender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message,
true, "UTF-8");
helper.setFrom("theviperbeast03@gmail.com", "Nocturne Music
Store");helper.setTo(to);helper.setSubject(subject);helper.set
Text(body, false); // false = обычный текст, true = html
mailSender.send(message);} catch (Exception e) {
throw new RuntimeException("Failed to send email", e);
```

JavaMailSender використовується як адаптер для *SMTP*; конфігурація (*host, port, username, password, TLS*) повинна міститися у *application.properties* або краще — у секретах.

MimeMessageHelper дозволяє відправляти *plain/text* або *HTML*, вкладення, *inline*-зображення.

Помилки піднімаються далі, це дозволяє лістенеру вирішувати, що робити з невдалою обробкою (*retry*, відправка в *DLQ*).

***NotificationListener — Kafka-listeners* для подій**

Фрагмент коду реалізації.

```
public void onUserRegistered(UserRegisteredEvent event) {}
public void onOrderCreated(OrderCreatedEvent event) {}
public void onPaymentProcessed(PaymentProcessedEvent event) {}
public void onPaymentFailed(PaymentFailedEvent event) {}
public void onVerificationEmail(VerificationEmailEvent event)
{}
```

Для кожного типу події є окремий *listener* із відповідним *containerFactory*, що дозволяє налаштовувати десеріалізацію під конкретний клас повідомлення.

Listener не кидає винятків назовні без обробки — помилки логуються; допустимо реалізувати retry-політику або відправку повідомлення в *DLQ*.

4.3 Контейнеризація та оркестрація

Для забезпечення стабільності, гнучкості та легкості розгортання система *Nocturne Music Store* була повністю контейнеризована за допомогою *Docker*. Кожен мікросервіс, база даних та допоміжний компонент розгортаються у власному контейнері, що забезпечує ізоляцію середовищ, відтворюваність конфігурації та незалежність оновлень.

Організацію взаємодії між контейнерами реалізовано за допомогою *Docker Compose*, який визначає повну екосистему системи — від брокера повідомлень до окремих бізнес-сервісів.

Архітектура контейнерного середовища

Першим кроком є розгортання подієво-орієнтованої інфраструктури на основі *Apache Kafka*, яка відповідає за асинхронну комунікацію між сервісами.

У файлі *docker-compose.yml* визначено основні компоненти, нижче наведено фрагмент коду.

```
kafka:
  image: confluentinc/cp-kafka:7.6.0
  container_name: kafka
  depends_on:    - zookeeper
  ports:    - "9092:9092"
```

Zookeeper виконує роль координатора для брокера. *Kafka* виступає центральним вузлом подієвої комунікації між мікросервісами.

Завдяки налаштуванню *KAFKA_ADVERTISED_LISTENERS* сервіси в контейнерах підключаються до брокера через *kafka:9092*, а розробник може підключитися з хост-машини через *localhost:9092*.

База даних для кожного мікросервісу

Кожен мікросервіс має власну базу даних *PostgreSQL*, що відповідає принципу *Database per Service Pattern*. Це забезпечує незалежність даних та зменшує зв’язність між сервісами.

Нижче наведено фрагмент коду з налаштуванням.

```
postgres-product:
  image: postgres:latest
  container_name: postgres-product
  environment:
    POSTGRES_DB: product-service-nocturne
```

```
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: root
  ports:
    - "5434:5432"
  networks:
    - default
```

Кожна база працює в окремому контейнері з унікальним портом, що дозволяє розробнику одночасно підключатися до всіх баз для тестування або міграцій. Така архітектура також дозволяє масштабувати сервіси незалежно один від одного без ризику конфліктів даних.

Мікросервіси системи

Основні бізнес-компоненти (*User, Product, Order, Payment, Notification*) реалізовані як окремі *Docker*-сервіси. Кожен контейнер будується з власного *Dockerfile* і має ізольовані параметри середовища.

Нижче наведено фрагмент коду.

```
user-service:  build: ./user-service-nocturne  container_name:
user-service  ports: - "8080:8080"
  depends_on: - kafka   - postgres-user  environment:
    SPRING_DATASOURCE_URL: jdbc:postgresql://postgres-
user:5432/user-service-nocturne
    SPRING_DATASOURCE_USERNAME: postgres
SPRING_DATASOURCE_PASSWORD: root
    SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092  networks:    -
default
```

Аналогічні конфігурації застосовані для інших мікросервісів (*product-service, order-service, payment-service*), кожен з яких має власну базу та незалежно підключається до *Kafka*.

Notification Service відповідає за надсилання повідомлень електронною поштою на основі подій, отриманих із *Kafka*.

Він слухає такі топіки, як *user-registered, order-created, payment-processed, payment-failed, email-verification*.

Нижче наведено фрагмент коду.

```
notification-service:
  build: ./notification-service-nocturne
  container_name: notification-service
  ports:    - "8084:8084"
  depends_on:  - kafka  environment:
    SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:9092
    MAIL_HOST: smtp.gmail.com    MAIL_PORT: 587
MAIL_USERNAME: theviperbeast03@gmail.com
    MAIL_PASSWORD: hzqwnpeezmvlrxbi  networks:    - default
```

Використання середовищних змінних для параметрів пошти (*SMTP*, логін, пароль) дозволяє легко адаптувати систему для різних поштових провайдерів або середовищ розгортання.

Завдяки *Docker Compose* забезпечено простоту запуску всієї системи однією командою, незалежність компонентів і можливість швидкого переходу до *Kubernetes*-оркестрації. Такий підхід забезпечує стабільність, гнучкість і масштабованість у середовищі реального навантаження.

4.4 Механізми управління (*Prometheus* та *Grafana*)

Для реалізації моніторингу в розробленій системі було обрано *Prometheus* як систему збору та зберігання метрик і *Grafana* як інструмент для їх візуалізації.

Prometheus збирає дані про роботу кожного мікросервісу через спеціальні ендпоїнти */actuator/prometheus*, які генерує *Spring Boot Actuator*. *Grafana* підключається до *Prometheus* як до джерела даних і будує динамічні графіки для наочного відображення стану системи.

Налаштування *Prometheus*

Prometheus було розгорнуто за допомогою *Docker Compose*, що забезпечує легку інтеграцію з іншими мікросервісами. У конфігураційному файлі *prometheus.yml* задано список сервісів, з яких *Prometheus* збирає метрики.

Фрагмент коду конфігурації *prometheus.yml*.

```
global:  scrape_interval: 10s
scrape_configs:  - job_name: 'spring-boot-services'
metrics_path: '/actuator/prometheus'    static_configs:      -
targets:  - 'host.docker.internal:8080' -
'host.docker.internal:8081'- 'host.docker.internal:8082' -
'host.docker.internal:8083'
```

Така конфігурація визначає, що *Prometheus* буде опитувати основні сервіси кожні 10 секунд.

Дані, отримані з ендпоїнтів */actuator/prometheus*, містять інформацію про:

1) *HTTP* запити: кількість запитів за статусами *HTTP* (*2xx, 3xx, 4xx, 5xx*); час обробки запитів (період часу); час обробки запитів за статусами *HTTP*; час обробки запитів за *HTTP* методами; час обробки запитів за *HTTP* шляхами.

2) *JVM* метрики: використання пам'яті (*heap ma non-heap*); час збору сміття (*garbage collection*); кількість потоків; кількість класів; час виконання методів.

3) Бази даних: кількість запитів до бази даних; час виконання запитів до бази даних; кількість активних з'єднань.

4) *HikariCP* (пул з'єднань): кількість активних з'єднань; кількість очікуючих з'єднань; час очікування з'єднання; час обробки запитів.

5) Метрики додатка: кількість запитів до конкретних методів; час виконання конкретних методів; кількість викликів конкретних бінів.

6) Метрики кешування: кількість операцій додавання до кешу; кількість операцій видалення з кешу; кількість операцій отримання з кешу; час виконання операцій з кешем.

Інтеграція *Prometheus* з мікросервісами

У кожному з мікросервісів було підключено *Spring Boot Actuator* та *Micrometer*, які забезпечують автоматичну генерацію метрик.

Фрагмент коду конфігурації для одного з мікросервісів (*User Service application.yml*) виглядає наступним чином.

```
management.endpoints.web.exposure.include=health,info,prometheus,metrics management.endpoint.prometheus.enabled=true
management.metrics.enable.all=true
```

Даний фрагмент відкриває ендпоїнти */actuator/health, /actuator/info* та */actuator/prometheus*, які використовуються *Prometheus* для збору інформації про стан сервісу.

Налаштування *Grafana*

Grafana підключається до *Prometheus* як джерела даних. Після успішного з'єднання було створено дашборд для моніторингу стану системи, який включає ключових графіків описаних вище.

Grafana автоматично підключається до *Prometheus*, який працює на порту 9090, і дозволяє створювати динамічні панелі моніторингу.

На графіку можна побачити, як у реальному часі відстежується час роботи програми та використання ресурсів процесору.

Інтеграція *Prometheus* та *Grafana* в систему забезпечила створення ефективного механізму моніторингу, який гарантує високу прозорість функціонування мікросервісів. Цей метод сприяє підвищенню надійності системи, полегшує процес налагодження та дає змогу оперативно реагувати на виникнення проблем у реальному середовищі експлуатації.

На рисунку 4.1 показано приклад дашборду, який відображає метрики роботи мікросервісів у реальному часі.

Рисунок 4.1 — Скріншот дашборду *Grafana* з показниками мікросервісів

Моніторингова інфраструктура відіграє ключову роль у підтримці та розвитку системи, особливо під час її масштабування, оскільки дозволяє вчасно виявляти потенційні слабкі місця та оптимізувати використання ресурсів.

4.5 Навантажувальне тестування

У процесі розроблення розподіленої мікросервісної системи важливим етапом є її тестування, яке забезпечує перевірку коректності роботи функціональних модулів, їхньої взаємодії, надійності, продуктивності та безпеки. Тестування дозволяє виявити можливі помилки ще до розгортання системи у реальному середовищі, зменшуючи ризики збоїв і втрати даних.

Термін «навантажувальне тестування» або «стрес-тестування» використовується по-різному у професійній спільноті тестувальників програмного забезпечення. Навантажувальне тестування зазвичай стосується практики моделювання очікуваного використання програмного забезпечення шляхом імітації одночасного доступу кількох користувачів до програми. Таким чином, це тестування найбільш актуальне для багатокористувацьких систем; часто таких, що побудовані за моделлю клієнт/сервер, таких як веб-сервери [34].

Для проведення навантажувального тестування було використано *Apache JMeter* — потужний інструмент з відкритим вихідним кодом, який дозволяє моделювати різні сценарії поведінки користувачів та вимірювати показники продуктивності веб-додатків, *REST API* та мікросервісів.

JMeter підтримує створення тестових планів із багатьма потоками (threads), які імітують одночасних користувачів, що виконують запити до системи. Основними компонентами тесту є:

- *Thread Group* — визначає кількість користувачів, тривалість тесту та кількість повторень;
- *HTTP Request* — описує конкретний запит до *API* або веб-сервісу;
- *Listeners* — збирають і візуалізують результати тестування (наприклад, час відгуку, кількість успішних/помилкових запитів, середній трафік тощо).

Для оцінки продуктивності мікросервісної архітектури *Nocturne Music Store* було розроблено тестовий план, який охоплював роботу основних *REST API* методів *User Service, Product Service, Order Service та Payment Service*.

Мета тесту — перевірити, як система реагує на велику кількість одночасних користувачів, які виконують повторювані запити до сервісів.

У конфігурації тесту були задані такі параметри:

- *Number of Threads (users)* — моделює одночасне підключення 3000 користувачів;
- *Ramp-up Period (seconds)* — усі користувачі підключаються поступово протягом 1 секунди;
- Loop Count — кожен користувач повторює запит 10 разів.

Це дозволяє оцінити поведінку системи як під короткочасним навантаженням, так і в умовах тривалого інтенсивного використання.

На рисунку 4.2 зображена конфігурація *Thread Group у Apache Jmeter*:

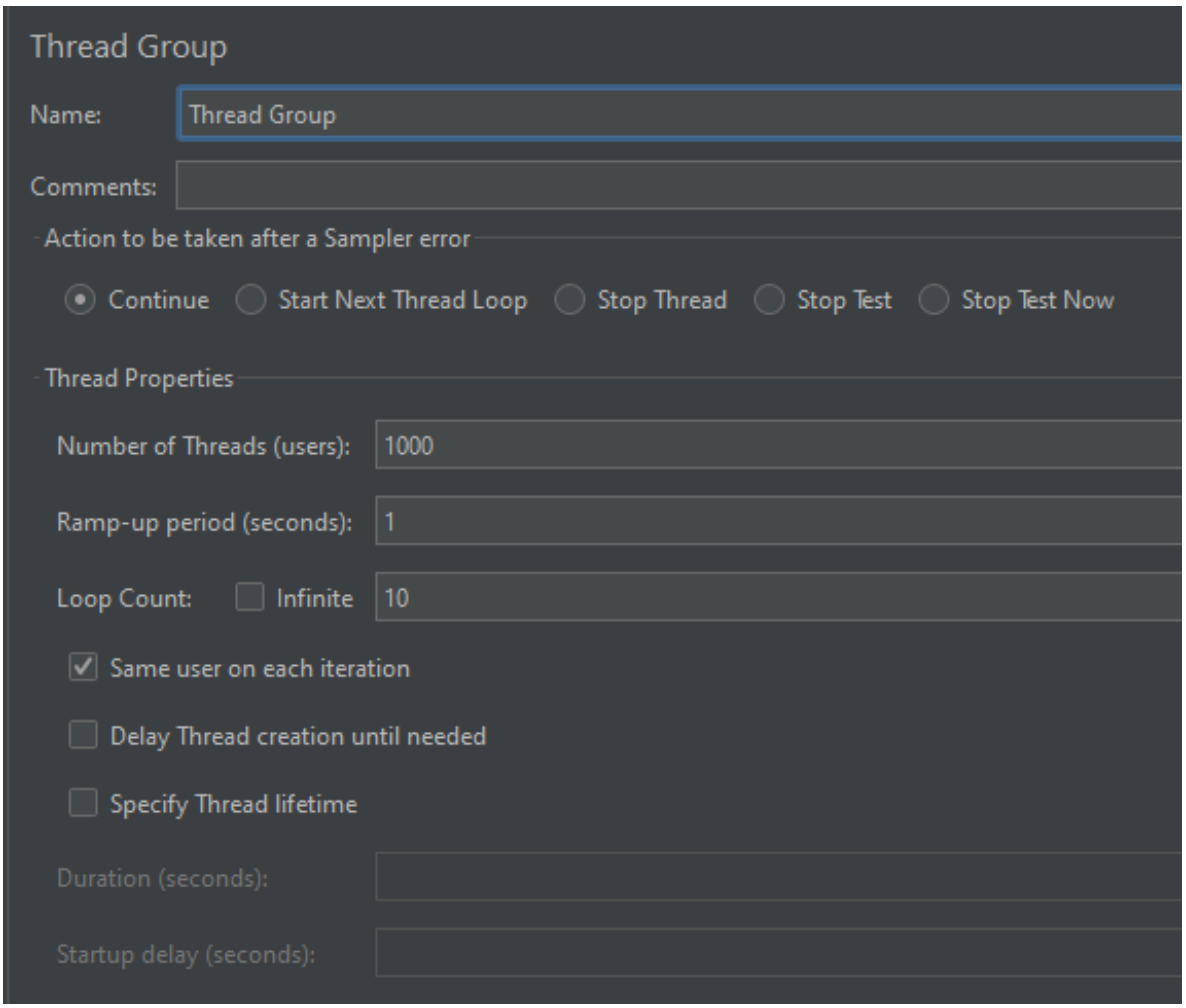

Рисунок 4.2 — Скріншот конфігурації Thread Group у *Apache Jmeter*

Для підтвердження результатів тестування наведено відповідний скріншот з середовища *Apache JMeter*, який демонструє отримані метрики.

Крім того, отримані показники дають змогу виявити потенційні вузькі місця та оцінити стабільність роботи системи в різних режимах. Аналіз зібраних метрик дозволяє зробити висновки щодо продуктивності, часу відгуку та рівня помилок під час виконання запитів.

Отримані результати показали стабільну роботу мікросервісів при високому рівні навантаження. Система витримала понад 10 000 *HTTP*-запитів (1000 користувачів × 10 ітерацій) без жодних помимилок та збоїв.

На рисунку 4.3 зображено вікно *Summary Report* із загальною статистикою тесту:

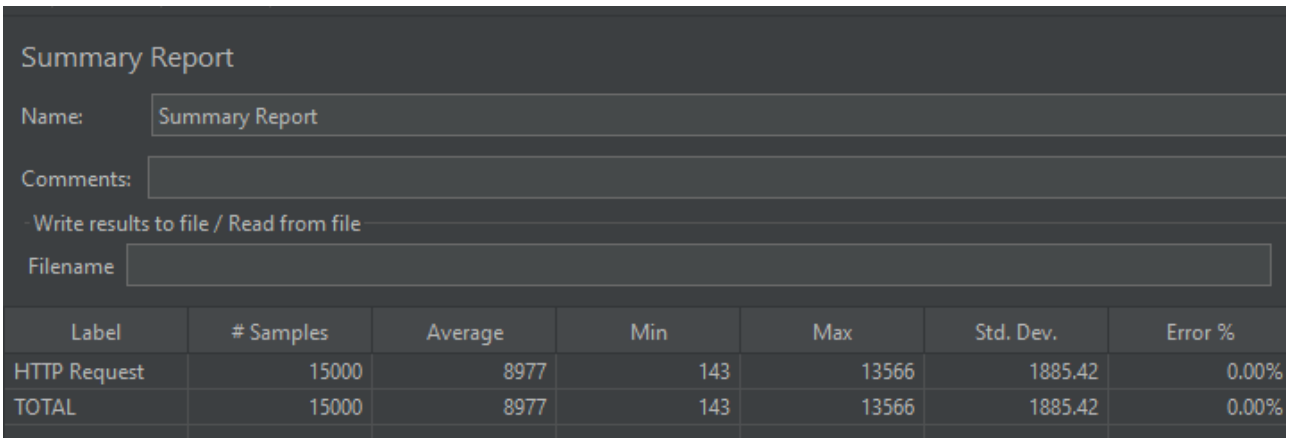

Рисунок 4.3 — Вікно *Summary Report* із загальною статистикою тесту

Основні результати навантажувального тестування:

- найшвидша відповідь — 0.14 с, найповільніша — 13.5 с;
- середній час відповіді ≈ 8 секунд;
- відсоток помилок — 0%;
- система обробляє ~ 79 запитів за секунду.

Отже, можна зробити висновок, що система успішно витримує навантаження від 1000 одночасних користувачів, демонструючи стабільну роботу та відсутність збоїв. Подальша оптимізація може бути спрямована на зменшення часу відповіді за рахунок покращення продуктивності бази даних, кешування результатів запитів або застосування асинхронної обробки.

Висновки до розділу

У четвертому розділі було детально розглянуто процес реалізації розробленої системи, починаючи з вибору стеку технологій і закінчуючи перевіркою продуктивності за допомогою навантажувального тестування.

Було реалізовано ключові мікросервіси: *User Service*, *Product Service*, *Order Service*, *Payment Service* та *Notification Service*, кожен з яких виконує свою спеціалізовану функцію та взаємодіє з іншими компонентами системи через

стандартизовані *API*. Така модульність забезпечує легкість у розвитку, тестуванні та підтримці системи.

Контейнеризація мікросервісів та їх оркестрація дозволили автоматизувати розгортання, масштабування та управління сервісами, що підвищує стабільність та ефективність роботи системи. Використання Prometheus та Grafana забезпечило моніторинг і візуалізацію ключових метрик, що дозволяє швидко виявляти і усувати проблеми в роботі сервісів.

Навантажувальне тестування підтвердило здатність системи ефективно обробляти високі обсяги запитів, що свідчить про її готовність до використання в реальному середовищі. Таким чином, реалізація системи відповідає поставленим вимогам і забезпечує надійну основу для подальшого розвитку та масштабування проекту.

ВИСНОВКИ

У даній магістерській роботі було виконано повний цикл проєктування, реалізації та дослідження розподіленої високонавантаженої інформаційної системи на основі мікросервісної та подійно-орієнтованої архітектури. У межах роботи послідовно розглянуто теоретичні основи побудови розподілених систем, проаналізовано сучасні архітектурні підходи, методи інтеграції сервісів, принципи масштабування, відмовостійкості, безпеки, а також інструменти управління та моніторингу мікросервісних систем.

У першому розділі систематизовано ключові поняття розподілених і високонавантажених систем, визначено їх основні характеристики та вимоги. Проведено порівняльний аналіз монолітної, мікросервісної та подійно-орієнтованої архітектур, що дозволило обґрунтовано обрати мікросервісну подійно-орієнтовану архітектуру як найбільш доцільну для побудови масштабованих і гнучких систем. Також розглянуто сучасні протоколи комунікації, інтеграційні патерни та технології управління сервісами.

У другому розділі здійснено аналіз предметної області, сформульовано функціональні та нефункціональні вимоги до системи, виконано оцінку очікуваного навантаження та потреб у масштабованості. Окрему увагу приділено архітектурним викликам та обмеженням, пов’язаним із високою кількістю запитів, асинхронною взаємодією сервісів, забезпеченням надійності, безпеки та узгодженості даних у розподіленому середовищі.

У третьому розділі розроблено архітектуру системи, визначено склад ключових мікросервісів та принципи їх подійно-орієнтованої взаємодії за допомогою брокера повідомлень *Apache Kafka*. Спроєктовано структури баз даних кожного сервісу, описано механізми безпеки й авторизації на основі *JWT* та ролей користувачів, а також запропоновано підхід до моніторингу та централізованого логування.

У четвертому розділі реалізовано програмну частину системи із використанням сучасного стеку технологій, зокрема *Java, Spring Boot, Spring Security, Apache Kafka, PostgreSQL, Docker, Prometheus та Grafana*. Було розроблено та налаштовано основні мікросервіси: *User Service, Product Service, Order Service, Payment Service та Notification Service*. Виконано контейнеризацію системи за допомогою *Docker* та організовано її роботу в єдиному середовищі за допомогою *docker-compose*. Реалізовано систему моніторингу та візуалізації показників роботи сервісів із використанням *Prometheus* і *Grafana*. Також у межах цього розділу проведено навантажувальне тестування розробленої системи із використанням *Apache JMeter*. Отримані результати підтвердили здатність системи стабільно обробляти значні обсяги запитів, коректно функціонувати під навантаженням та демонструвати належні показники продуктивності, що свідчить про правильність обраних архітектурних рішень.

У результаті виконання магістерської роботи було досягнуто поставленої мети — спроєктовано та реалізовано сучасну розподілену високонавантажену систему з мікросервісною подійно-орієнтованою архітектурою, яка є масштабованою, відмовостійкою, безпечною та придатною для подальшого розвитку. Розроблене програмне рішення може бути використане як основа для реальних комерційних систем або як навчальний приклад побудови сучасних мікросервісних платформ.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Distributed architecture* [Електронний ресурс]. – Режим доступу: [https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture] (дата звернення: 01.10.2025).

2. *Distributed Computing Concepts and Models* // *IBM Documentation* [Електронний ресурс] – Режим доступу: [*https://www.ibm.com/docs/en/distributed-computing*] (дата звернення: 01.10.2025).

3. Розподілені обчислювальні системи [Електронний ресурс] // *Studfile*. – Режим доступу: [*https://studfile.net/preview/3271234/page:6/*] (дата звернення: 02.10.2025).

4. *Scalability patterns for distributed systems guide* [Електронний ресурс] // *Imaginary Cloud*. – Режим доступу: [*https://www.imaginarycloud.com/blog/scalability-patterns-for-distributed-systems-guide*] (дата звернення: 03.10.2025).

5. *Fault Tolerance Concepts and Patterns* // *Microsoft Azure Architecture Center* [Електронний ресурс]. – Режим доступу: [*https://learn.microsoft.com/azure/architecture/framework/resiliency/fault-tolerance*] (дата звернення: 04.10.2025).

6. *What is high availability?* [Електронний ресурс] // *Cisco*. – Режим доступу: [*https://www.cisco.com/site/us/en/learn/topics/networking/what-is-high-availability.html*] (дата звернення: 05.10.2025).

7. *Consistency guarantees in distributed systems explained simply* [Електронний ресурс] // *Medium*. – Режим доступу: [*https://kousiknath.medium.com/consistency-guarantees-in-distributed-systems-explained-simply-720caa034116*] (дата звернення: 07.10.2025).

8. *Introduction to high load: what is it?* [Електронний ресурс] // *Geniusee*. – Режим доступу: [*https://geniusee.com/single-blog/introduction-to-high-load-what-is-it*] (дата звернення: 07.10.2025).

9. *Latency in distributed system* [Електронний ресурс] // *GeeksforGeeks*. – Режим доступу: [*https://www.geeksforgeeks.org/system-design/latency-in-distributed-system/*] (дата звернення: 08.10.2025).

10. Що таке монолітна архітектура [Електронний ресурс] // *QALight*. – Режим доступу: [*https://qalight.ua/baza-znaniy/shho-take-monolitna-arhitektura/*] (дата звернення: 10.10.2025).

11. Що таке мікросервісна архітектура: значення, складові, переваги [Електронний ресурс] // *Wezom*. – Режим доступу: [*https://wezom.com.ua/ua/blog/scho-take-mikroservisna-arhitektura-znachennya-skladovi-perevagi*] (дата звернення: 12.10.2025).

12. *Event-driven architecture* [Електронний ресурс] // *VPN Unlimited*. – Режим доступу: [*https://www.vpnunlimited.com/ua/help/cybersecurity/event-driven-architecture*] (дата звернення: 14.10.2025).

13. Оркестрація *Docker* та контейнерів [Електронний ресурс] // *Hostragons*. Режим доступу: [*https://www.hostragons.com/uk/%d0%b1%d0%bb%d0%be%d0%b3/%d0%be%d1%80%d0%ba%d0%b5%d1%81%d1%82%d1%80%d0%b0%d1%86%d1%96%d1%8f-docker-%d1%82%d0%b0-%d0%ba%d0%be%d0%bd%d1%82%d0%b5%d0%b9%d0%bd%d0%b5%d1%80%d1%96%d0%b2-%d0%be%d0%bf%d0%b5%d1%80%d0%b0%d1%86%d1%96/#Konteyner_Orkestrasyonu_Nedir*] (дата звернення: 21.10.2025).

14. *Service Mesh Architecture Explained* // [Електронний ресурс]. – Режим доступу: [*https://istio.io/latest/docs/concepts/what-is-istio/*] (дата звернення: 21.10.2025).

15. *Configuration management* [Електронний ресурс] // *VPN Unlimited*. – Режим доступу: [*https://www.vpnunlimited.com/ua/help/cybersecurity/configuration-management?srsltid=AfmBOoonCTiv9TDgbv87RdrfCNynJ5Bfw2kgxv4uAFz0OVQKUlO-VFE-*] (дата звернення: 22.10.2025).

16. *Service discovery* // [Електронний ресурс] // *AWS*. – Режим доступу: [*https://aws.amazon.com/microservices/service-discovery/*] (дата звернення: 28.10.2025).

17. *Burns, B., Beda, J., Hightower, K. Kubernetes: Up and Running: Dive into the Future of Infrastructure. — 3rd ed. — O'Reilly Media*, 2023. — 350 с. (дата звернення: 28.10.2025)

18. *Merkel, D. Docker: Up & Running. — 3rd ed. — O'Reilly Media*, 2023. — 420 с. (дата звернення: 28.10.2025)

19. *Kubernetes // Kubernetes* [Електронний ресурс]. – Режим доступу: [*https://kubernetes.io/docs/concepts/overview/*] (дата звернення: 28.10.2025).

20. *Horizontal and Vertical Scaling | System Design* [Електронний ресурс] // *Geeks for Geeks*. Режим доступу: [*https://www.geeksforgeeks.org/system-design/system-design-horizontal-and-vertical-scaling/*] (дата звернення: 28.10.2025).

21. *Richards M., Ford N. Fundamentals of Software Architecture. — O'Reilly Media*, 2021. — 448 с. (дата звернення: 28.10.2025).

22. *Richardson, C., Smith, F. Microservices Security in Action. — Manning Publications*, 2021. — 400 с. (дата звернення: 01.11.2025)

23. *Richardson C. Microservices Patterns, 2nd Edition. — Manning Publications*, 2023. — 600 с. (дата звернення: 02.11.2025)

24. *Kleppmann M. Designing Data-Intensive Applications. — 2nd ed. — O'Reilly Media,* 2024. — 650 с. (дата звернення: 03.11.2025)

25. *Hoffmann A. Web Application Security: Modern Attacks and Defenses. — Packt Publishing*, 2022. — 380 с. (дата звернення: 04.11.2025)

26. *OWASP Top Ten* [Електронний ресурс] // *OWASP*. Режим доступу: [https://owasp.org/www-project-top-ten/] (дата звернення: 04.11.2025).

27. *Flanagan D. JavaScript: The Definitive Guide. — 8th ed. — O'Reilly Media,* 2023. — 720 с. (дата звернення: 05.11.2025)

28. *AWS Documentation* [Електронний ресурс] // *Amazon API Gateway Developer Guide*. Режим доступу: [https://docs.aws.amazon.com/apigateway/] (дата звернення: 05.11.2025).

29. *Vernon V. Domain-Driven Design Distilled. — 2nd ed. — Addison-Wesley*, 2021. — 240 с. (дата звернення: 05.11.2025)

30. *Kafka Documentation* [Електронний ресурс] // *Apache Kafka Architecture*. Режим доступу: [*https://kafka.apache.org/documentation/#architecture*] (дата звернення: 05.11.2025).

31. *Jones M., Bradley J., Sakimura N. JSON Web Token* [Електронний ресурс] // *JWT*. Режим доступу: [*https://datatracker.ietf.org/doc/html/rfc7519*] (дата звернення: 07.11.2025).

32. *Ferraiolo D., Kuhn R., Chandramouli R. Role-Based Access Control. — Artech House*, 2022. — 320 с. (дата звернення: 09.11.2025).

33. *Grafana Labs* [Електронний ресурс] // *Grafana and Prometheus Documentation*. Режим доступу: [*https://grafana.com/docs/*] (дата звернення: 09.11.2025).

34. *Load Testing Concepts and Best Practices* [Електронний ресурс] // *Apache JMeter Documentation.* . Режим доступу: [https://jmeter.apache.org/usermanual/index.html] (дата звернення: 10.11.2025).