Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Факультет комп'ютерних наук
Кафедра теоретичної та прикладної системотехніки

«Затверджую»
Зав. кафедри теоретичної та
прикладної системотехніки
_____ д.т.н., проф. С. І. Шматков
«___» _____ 2024 р.

Пояснювальна записка

до кваліфікаційної роботи
бакалавра

на тему: «**Модель комерційного програмного додатку на основі мікросервісної архітектури**»

Захищено на засіданні
Атестаційної комісії № 42
протокол № __ від __.06.2024 р.
Оцінка _____ / _____
Голова Атестаційної комісії
_____ **СКОБ Ю. О.**

Виконав:
студент 4 курсу, групи КІ–41
Галузь знань: 12 – Інформаційні технології
Спеціальність: 123 – Комп'ютерна інженерія.
ДЕЙКУН Дмитро Юрійович_____

Керівник: к.т.н., доцент ЗВО кафедри теоретичної та прикладної системотехніки
СТРІЛЕЦЬ Вікторія Євгенівна

Рецензент: к.т.н., доцент ЗВО, в.о. зав. кафедри теоретичної та прикладної інформатики
МЕНЯЙЛОВ Євген Сергійович____

Харків – 2024

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел і чотирьох додатків. Загальний обсяг роботи складає 73 сторінки, із яких 50 сторінок основної частини з 25 рисунками, 29 найменуваннями списку використаних джерел та чотири додатки.

Метою кваліфікаційної роботи є забезпечення масштабованості та підвищення надійності комерційного програмного додатку через застосування мікросервісної архітектури.

Предмет дослідження – моделі архітектур програмних додатків та методи їх інтеграції для створення ефективного та масштабованого програмного забезпечення.

Об'єкт дослідження – процес розробки комерційних програмних додатків.

Проблема, яка вирішується в кваліфікаційній роботі полягає в тому, щоб застосувати сучасні підходи та інструменти для розробки програмного забезпечення на базі мікросервісів з метою удосконалення процесу розробки та забезпечення масштабованості й надійності роботи комерційних додатків.

Ключові слова: мікросервісна архітектура, монолітна архітектура, Spring Boot, Spring Cloud, Discovery Server, REST API, масштабування, балансування навантаження.

ABSTRACT

The explanatory note for the bachelor's qualification work consists of an introduction, three chapters, conclusions, a list of references, and four appendices. The total length of the work is 73 pages, of which 50 pages are the main part with 25 images, 29 items in the list of references, and four appendices.

The purpose of the qualification work is to develop a model of a commercial software application based on a microservice architecture.

The subject of research is microservices models and methods for their integration to create efficient and scalable software.

The object of research is the process of developing commercial software applications.

The problem that is solved in the qualification work is to apply modern approaches and tools for software development based on microservices to improve development processes and ensure scalability, as well as application reliability.

Keywords: microservice architecture, monolithic architecture, Spring Boot, Spring Cloud, Discovery Server, REST API, scaling, load balancing.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

БД	–	база даних;
URL	–	Uniform Resource Locator (англ.) – уніфікована адреса певного ресурсу в інтернеті
REST API	–	(Representational State Transfer англ.) - це стиль архітектури програмного забезпечення, що використовується для створення веб-служб. Основною ідеєю REST є використання стандартних протоколів HTTP для взаємодії між клієнтами і серверами.
JSON	–	(JavaScript Object Notation англ.) - це легкий формат обміну даними, який використовується для передачі структурованих даних між системами
JWT	–	це стандарт токена доступу на основі JSON, стандартизованого в RFC 7519. Як правило, використовується для передачі даних для аутентифікації в клієнт-серверних програмах.
XML	–	(Extensible Markup Language) - це мова розмітки, призначена для представлення структурованих даних у вигляді тексту.
CQRS	–	це архітектурний патерн, що розділяє операції читання і запису в окремі моделі. Основна ідея полягає в тому, що різні моделі повинні використовуватись для операцій читання (запитів) та запису (команд), що дозволяє оптимізувати обидві операції і підвищити продуктивність та масштабованість системи.
HTTP	–	протокол передачі даних, що використовується в комп'ютерних мережах. Назва скорочена від HyperText Transfer Protocol, протокол передачі гіпертекстових документів
API	–	набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

ВСТУП

Актуальність роботи. В сучасному світі, де технології швидко розвиваються, бізнес-середовище стає все більш конкурентним. Компанії шукають способи покращити ефективність своєї роботи, залучити клієнтів та забезпечити якісний продукт або послугу. Одним із ключових факторів успіху є використання сучасних технологій та архітектурних підходів, що дозволять побудувати масштабовані, надійні, відмовостійкі та зручні для користувача системи.

Мікросервісна архітектура є однією з таких сучасних парадигм. Вона дозволяє розбити великі програмні системи на менші, незалежні компоненти (мікросервіси), які можуть розвиватися та масштабуватися окремо, що в свою чергу дозволяє розширяти програмну функціональність, не впливаючи на інші її частини. Це робить процес розробки, тестування та супроводження програмного забезпечення більш простим та ефективним.

Метою дослідження є забезпечення масштабованості та підвищення надійності комерційного програмного додатку через застосування мікросервісної архітектури.

Об’єкт дослідження – програмний продукт, розроблений за допомогою мікросервісної архітектури, включаючи його архітектурну структуру, технологічний стек, методи розробки та інструменти підтримки.

Методи дослідження: аналіз літературних джерел щодо мікросервісної архітектури та сучасних підходів до розробки програмного забезпечення; експериментальна реалізація моделі програмного додатку на основі мікросервісної архітектури для перевірки ефективності пропонованих підходів; зіставлення результатів експерименту з відомими практиками розробки програмного забезпечення для визначення переваг та недоліків запропонованого підходу.

Предмет дослідження – моделі мікросервісів та методи їх інтеграції для створення ефективного та масштабованого програмного забезпечення.

Завдання дослідження:

1. Аналіз існуючих архітектур програмного забезпечення, їх особливостей і обмежень.
2. Розробка моделі комерційного програмного додатку на основі мікросервісної архітектури.
3. Програмна реалізація моделі комерційного програмного додатку.
4. Тестування створеного комерційного програмного додатку та надання рекомендацій щодо його використання.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Загальне поняття архітектури програмного забезпечення

Архітектура програмного забезпечення — це структурна основа системи програмного забезпечення, що складається з її компонентів, зв'язків між ними, а також принципів та настанов для проєктування та розвитку системи з часом. Архітектура визначає високорівневу структуру системи та способи взаємодії її компонентів, що забезпечує виконання функціональних та нефункціональних вимог.

Основні цілі архітектури програмного забезпечення:

1. **Модульність** – розбиття системи на незалежні модулі або компоненти, що дозволяє спрощувати розробку, тестування, обслуговування та масштабування системи.

2. **Масштабованість** – забезпечення здатності системи ефективно функціонувати при зростанні обсягу даних, користувачів або транзакцій.

3. **Продуктивність** – оптимізація часу відгуку та пропускної здатності системи для задоволення вимог користувачів.

4. **Надійність** – гарантія безперебійної роботи системи, мінімізація відмов та забезпечення відновлення після збоїв.

5. **Захищеність** – захист системи та її даних від несанкціонованого доступу та атак.

Архітектурні підходи можуть варіюватися в залежності від типу системи та вимог до неї. До поширених архітектурних підходів відносять монолітну архітектуру, мікросервісну архітектуру, клієнт-серверну архітектуру, сервіс-орієнтовану архітектуру тощо.

Одним з ключових аспектів архітектури програмного забезпечення є її гнучкість та здатність адаптуватися до змінних вимог і технологій. Це досягається через використання шаблонів архітектури, дотримання принципів

SOLID (єдина відповідальність, відкритість/закритість, підстановку Лісков, поділ інтерфейсів, інверсію залежностей) та кращих практик розробки програмного забезпечення [1].

1.2 Традиційні підходи до побудови програмного забезпечення

У світі розробки програмного забезпечення існує чимало підходів до побудови архітектури системи, кожен з яких має свої переваги та недоліки. Два найбільш розповсюджених підходи – це монолітна архітектура та мікросервісна архітектура.

1.2.1 Монолітна архітектура

Монолітна архітектура – один з архітектурних підходів до проектування програмного забезпечення, коли всі компоненти системи інкапсульовані в один великий блок або додаток (рис. 1.1), який розгортається та масштабується як єдине ціле. При розробці додатку з монолітною архітектурою вся кодова база знаходиться в одному місці, програмне забезпечення розгортається на єдиному сервері та взаємодіє з єдиною базою даних. Такий підхід є оптимальним для невеликих проєктів, в яких не передбачається додавання великої кількості нового функціоналу.

Монолітна архітектура має чітку структуру та централізоване управління, що дозволяє уніфікувати підхід до розробки та підтримки програмного забезпечення. Завдяки цьому, уніфіковані процеси налагодження та відлагодження програмного коду, а також забезпечення якості, сприяють стабільності та надійності системи. Єдина кодова база та єдина платформа для виконання програмного забезпечення дозволяють розробникам зосередитися на бізнес-логіці та функціональності додатку, не витрачаючи час на вирішення проблем сумісності з окремими модулями [1].

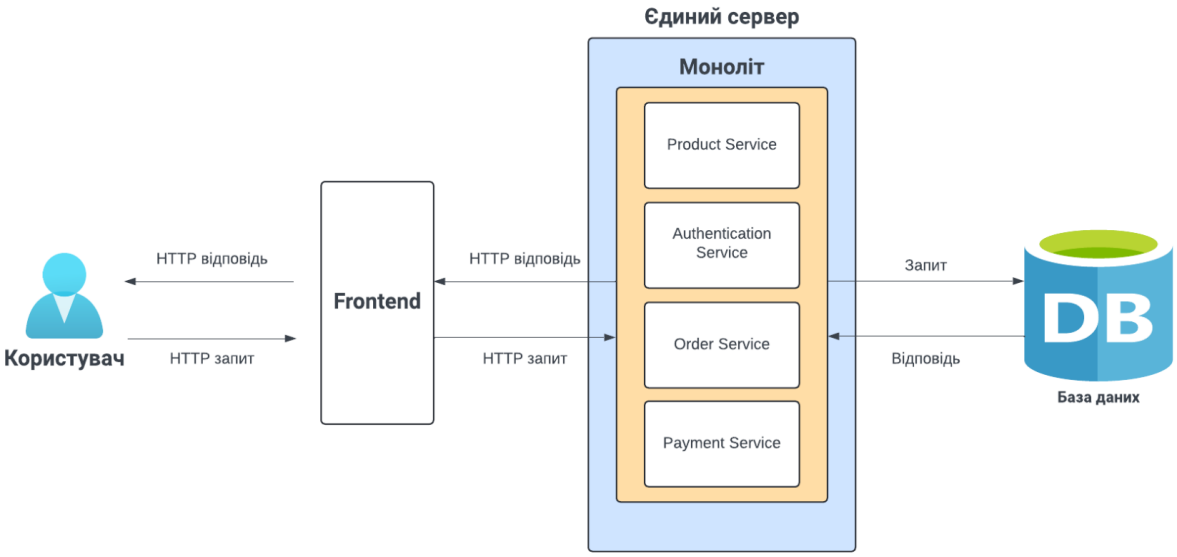

Рисунок 1.1 – Представлення монолітної архітектури

Переваги монолітної архітектури:

– **Простота.** Всі компоненти системи зосереджені в одному місці, що спрощує розробку, тестування та розгортання програмного забезпечення.

– **Ефективність.** Компоненти в моноліті зазвичай спільно використовують оперативну пам’ять, тому взаємодія між ними відбувається швидше, на відміну від сервісів, що обмінюються даними за допомогою HTTP тощо.

– **Зменшення кількості помилок.** Всі частини системи взаємодіють одна з одною безпосередньо, що зменшує кількість помилок, що пов'язана з міжкомпонентною комунікацією.

– **Єдиний технологічний стек.** Програмне забезпечення, яке побудовано на основі монолітної архітектури, використовує єдиний технологічний стек, що спрощує адаптацію нових членів команди та не потребує великої кількості різнопланових знань.

Недоліки монолітної архітектури:

– **Повнозв’язність.** Помилка в будь-якому компоненті системи може призвести до зупинки роботи одразу всього додатку.

– **Труднощі з масштабуванням.** Неможливість масштабування окремих компонентів може призводити до зниження продуктивності та неефективного використання ресурсів.

– **Ускладнення підтримки.** Чим більшою стає кодова база - тим складніше підтримувати додаток, оскільки будь-які зміни зміни в одному компоненті можуть вплинути на інші компоненти системи.

– **Оновлення.** Через єдину велику кодову базу та тісне зв’язування компонентів один з одним, додаток повинен розгортатись заново після кожног оновлення.

– **Єдиний технологічний стек.** Єдиний технологічний стек може завадити використовувати різні технології для різних частин проєкту. Зміни в технологічному стеці коштують дорого, як з точки зору грошових витрат, так і з точки зору часу.

1.2.2 Мікросервісна архітектура

Мікросервісна архітектура – це сучасний підхід до розробки програмного забезпечення, що ґрунтується на декомпозиції системи на сукупність автономних служб або сервісів. Кожен мікросервіс є самодостатнім компонентом, який виконує чітко визначену функцію і взаємодіє з іншими мікросервісами через стандартизовані інтерфейси, зазвичай протоколи HTTP або системи обміну повідомленнями (наприклад, RabbitMQ або Apache Kafka). Базове представлення мікросервісної архітектури зображено на рис. 1.2 [2].

Основною характеристикою мікросервісної архітектури є незалежність сервісів, що забезпечує можливість їх розробки, тестування, розгортання та масштабування окремо один від одного. Це досягається завдяки тому, що кожен мікросервіс має власну базу даних та не покладається на спільні бази даних, що дозволяє уникнути проблем, пов'язаних з тісною взаємозалежністю компонентів. Взаємодія між мікросервісами відбувається через чітко визначені API, що забезпечує ізольованість і гнучкість системи.

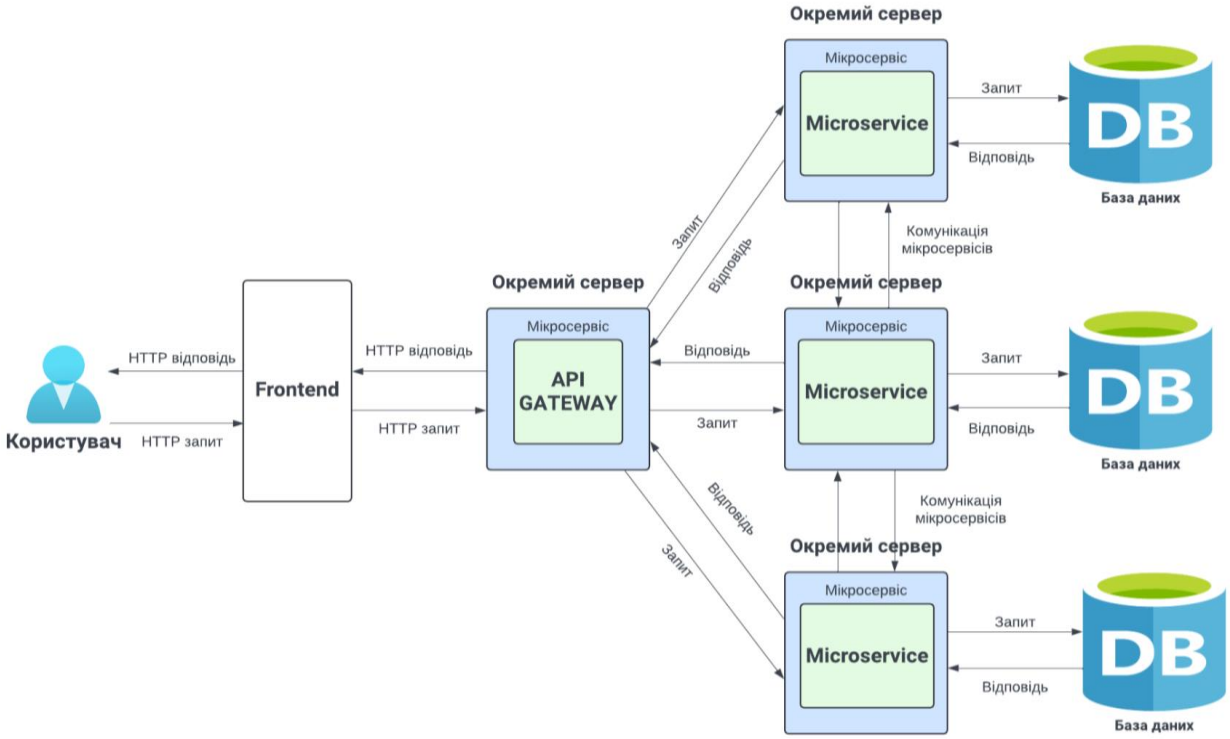

Рисунок 1.2 – Представлення мікросервісної архітектури

Важливим аспектом мікросервісної архітектури є використання різноманітних технологій і мов програмування для реалізації окремих мікросервісів. Це дозволяє вибирати оптимальні інструменти для вирішення конкретних завдань, зокрема, один мікросервіс може бути розроблений на мові програмування Java, інший – на Python, а ще інший – на Go. Головне, щоб мікросервіси взаємодіяли через узгоджені інтерфейси.

Клієнтська частина додатку (фронтенд) взаємодіє з системою мікросервісів через шлюз API (API Gateway), який виступає посередником між клієнтом та мікросервісами. Шлюз API отримує запити від фронтенду та відповідно до своєї логіки розподіляє їх між відповідними мікросервісами. Це забезпечує централізовану точку доступу для клієнтських запитів, спрощує маршрутизацію, балансування навантаження, а також сприяє підвищенню безпеки та контролю доступу. Шлюз API також може здійснювати додаткові функції, такі як об'єднання відповідей від кількох мікросервісів, кешування та управління сесіями.

Управління даними є критичним аспектом мікросервісної архітектури. Оскільки кожен мікросервіс має власну базу даних, виникає проблема узгодженості даних між різними сервісами. Для вирішення цієї проблеми використовуються патерни узгодження, такі як Event Sourcing та CQRS (Command Query Responsibility Segregation). Event Sourcing передбачає збереження усіх змін стану як подій, що дозволяє відновити будь-який попередній стан системи. CQRS, у свою чергу, розділяє операції на командні та запитні, що дозволяє оптимізувати продуктивність і масштабованість системи.

1.3 Переваги і недоліки мікросервісної архітектури

Підхід до побудови програмного забезпечення на основі мікросервісної архітектури має ряд *переваг*, серед яких:

– **Масштабованість.** Мікросервіси дозволяють масштабувати окремі компоненти системи незалежно один від одного, що в свою чергу дозволяє ефективніше використовувати ресурси та підвищувати продуктивність.

– **Надійність.** Відмова або збій в одному з мікросервісів не впливає на роботу інших частин системи, а лише порушує роботу тих функцій, за які він відповідає. Це означає, що система продовжить функціонувати далі, навіть, при відмові якоїсь з частин цієї системи.

– **Гнучкість у розробці.** Кожен окремий мікросервіс може використовувати свій технологічний стек, що дозволяє оптимально обрати інструменти для кожного окремого сервісу.

– **Незалежність розгортання.** Кожен окремий мікросервіс може бути розгорнутий та оновлений, незалежно від інших сервісів, що у свою чергу дозволяє швидше впроваджувати нову функціональність та виправляти існуючі помилки.

– **Змінюваність.** Мікросервіс може бути замінений на інший незалежно від інших сервісів [3].

Крім переваг також існують і *недоліки*, серед яких:

– **Складність управління.** Збільшення кількості мікросервісів потребує більшої кількості додаткових інструментів та підходів, а також ускладнює управління системою, відстежування взаємодії сервісів, моніторингу та логування.

– **Затримка в комунікації.** Виклики між мікросервісами здійснюються не напряму, як у монолітній архітектурі, а через мережу, що може призводити до додаткових затримок і впливати не продуктивність системи.

– **Розподілена природа транзакцій.** Забезпечення консистентності даних у розподілених системах є складнішим і вимагає спеціальних підходів до керування транзакціями, таких як патерн Сага, або паттерн двофазного коміту.

– **Високі вимоги до інфраструктури.** Для ефективного функціонування мікросервісної архітектури потрібні більш складні інфраструктурні рішення, такі як оркестрація контейнерів, управління конфігураціями тощо.

– **Збільшення накладних витрат.** Кожен мікросервіс потребує окремого середовища виконання, що може призвести до зростання накладних витрат на обчислювальні ресурси та їх управління [3].

1.4 Типи комунікації між мікросервісами

У мікросервісній архітектурі обмін даними між мікросервісами є критично важливим аспектом, який забезпечує координацію та співпрацю між окремими компонентами системи. Вибір методів комунікації між мікросервісами безпосередньо впливає на продуктивність, надійність та масштабованість системи, що робить цей аспект одним з ключових при проєктуванні та впровадженні мікросервісних рішень.

При виборі методу комунікації слід також враховувати питання безпеки, управління транзакціями та відстеження виконання запитів. Важливо забезпечити належний рівень захисту даних під час їх передачі, реалізувати механізми транзакційного контролю для збереження цілісності даних, а також впровадити інструменти для моніторингу та логування взаємодій між мікросервісами.

Комунікацію між мікросервісами можна класифікувати відповідно до типу взаємодії. Розрізняють синхронну, асинхронну комунікацію та гібридну (поєднання синхронної та асинхронної).

1.4.1 Синхронна комунікація між мікросервісами

Синхронна комунікація передбачає обмін повідомленнями між мікросервісами в реальному часі, коли відправник очікує відповіді від одержувача перш ніж продовжити виконання.

Характеристики синхронної комунікації [4]:

– **Блокування процесів.** У синхронній комунікації процес відправника залишається заблокованим до отримання відповіді від сервісу-одержувача (рис. 1.3). Це забезпечує послідовність виконання операцій, але може призвести до збільшення часу очікування і потенційних затримок у випадку високої навантаженості або несправностей.

– **Взаємозалежність сервісів.** Високий рівень взаємозалежності між сервісами може призвести до проблем з надійністю системи. Якщо один з мікросервісів стає недоступним, це може вплинути на загальну працездатність системи, оскільки інші сервіси, що залежать від нього, не зможуть продовжити роботу без отримання відповіді.

– **Транзакційна цілісність.** Синхронна комунікація може бути корисною для забезпечення транзакційної цілісності в межах одного бізнес-процесу. Завдяки послідовному виконанню запитів і відповідей, можна легко відстежувати стан транзакції та забезпечувати необхідні перевірки на кожному етапі.

Рисунок 1.3 – Приклад взаємодії мікросервісів з використанням синхронної комунікації

Види синхронної комунікації між мікросервісами:

– **REST API –** це архітектурний стиль, який використовує HTTP протокол для доступу до ресурсів за допомогою методів (GET, POST, PUT, DELETE) і повертає відповідь у форматі JSON або XML.

– **gRPC –** це високопродуктивний інструмент для взаємодії між різними сервісами. Він використовує HTTP/2 - більш нову та сучасну версію протоколу HTTP, для взаємодії між клієнтом та сервером.

– **SOAP –** це протокол обміну повідомленнями, який використовує XML для форматування повідомлень. SOAP містить визначення повідомлень, типів даних і правил обробки помилок, що робить його більш складним у використанні, ніж інші протоколи.

Переваги синхронної комунікації між мікросервісами [4]:

– **Прямолінійність моделі.** Синхронна модель є легкою для розуміння та відлагодження, оскільки кожен запит чітко відповідає певній відповіді.

– **Негайний зворотний зв'язок.** Можливість отримати миттєву відповідь дозволяє швидко реагувати на результати запитів та приймати подальші рішення.

– **Простота виявлення та обробки помилок.** Оскільки кожен запит супроводжується негайною відповіддю, помилки можуть бути виявлені та оброблені в реальному часі.

Недоліки синхронної комунікації між мікросервісами [4]:

– **Затримки.** Синхронні запити можуть спричиняти затримки, особливо у випадку проблем з мережею або високої завантаженості сервісів.

– **Надійність.** Взаємозалежність між сервісами приводить до збільшення ризику виникнення проблем у разі відмови одного з компонентів.

– **Масштабованість.** Підтримка великої кількості синхронних запитів може потребувати великої кількості ресурсів та складної інфраструктури, що може ускладнити масштабування системи.

1.4.2 Асинхронна комунікація між мікросервісами

Асинхронна комунікація передбачає обмін повідомленнями без очікування відповіді в реальному часі. Відправник і одержувач можуть функціонувати, незалежно один від одного [4].

Характеристики асинхронної комунікації:

– **Розділення компонентів.** Асинхронна комунікація дозволяє відправникам та одержувачам працювати незалежно один від одного, що зменшує взаємозалежність між мікросервісами. Це робить систему більш гнучкою та стійкою до збоїв окремих компонентів.

– **Відсутність блокування.** Відправникам не потребно чекати відповіді, що дозволяє їм продовжувати виконання інших завдань одразу після відправлення повідомлення (рис. 1.4). Це значно покращує продуктивність та дозволяє більш ефективно використовувати ресурси.

Рисунок 1.4 – Приклад взаємодії мікросервісів з використанням асинхронної комунікації

Переваги асинхронної комунікації між мікросервісами [4]:

– **Масштабованість**. Завдяки можливості обробляти повідомлення асинхронно, системи можуть легше масштабуватися для обробки великих обсягів даних та пікових навантажень.

– **Стійкість до збоїв.** Асинхронні системи менш чутливі до збоїв окремих компонентів, оскільки відправник не покладається на негайну доступність одержувача. Це покращує загальну надійність системи.

– **Гнучкість.** Асинхронна комунікація дозволяє легше впроваджувати нові мікросервіси або змінювати існуючі без значного впливу на інші компоненти системи.

Недоліки асинхронної комунікації [4]:

– **Складність**. Реалізація асинхронної комунікації може бути більш складною, оскільки потребує додаткових механізмів для обробки повідомлень, гарантування їх доставки та відстеження стану системи.

– **Відкладена обробка.** Через те, що повідомлення не обробляються негайно, може виникнути затримка в отриманні та обробці інформації, що не завжди прийнятно для критичних операцій.

– **Управління станом.** Відсутність прямого зворотного зв’язку ускладнює управління транзакціями та відстеження виконання запитів, що може вимагати впровадження додаткових механізмів для забезпечення цілісності даних.

Види асинхронної комунікації між мікросервісами:

1. **Черга повідомлень**.
    a. **Архітектура та Модель Використання [5]**:
        i. Точка-точка (Point-to-Point): Відправник надсилає повідомлення в чергу, з якої його може забрати один споживач. Це підходить для задач, де кожне повідомлення повинно бути оброблене лише один раз.
        ii. Споживачі: Повідомлення доставляється одному споживачу. Якщо споживач обробив повідомлення, воно видаляється з черги.
    b. Механізм Доставки [5]:
        i. Повідомлення зберігається в черзі до тих пір, поки його не забере один зі споживачів.
        ii. Забезпечує надійну доставку повідомлень навіть у випадку збою одного зі споживачів.
2. **Потік подій**.
    a. **Архітектура та Модель Використання [6]**:
        i. Публікація/Підписка (Publish/Subscribe): Відправник (публікатор) надсилає події в потік, на який можуть

підписуватися кілька споживачів. Кожен споживач отримає копію події.

ii. Споживачі: Кожен споживач, підписаний на потік, отримує всі події з потоку. Події не видаляються після їх отримання, а зберігаються певний час.

b. Механізм Доставки [6]:

i. Події зберігаються у потоці для подальшого споживання всіма підписаними споживачами.

ii. Дозволяє обробляти події в режимі реального часу та ретроспективно переглядати події з потоку.

1.5. Інструменти та платформи для реалізації мікросервісної архітектури

Розробка та підтримка мікросервісної архітектури вимагає використання різноманітних інструментів і платформ, які забезпечують ефективне управління сервісами, їхню оркестрацію, моніторинг, а також безпеку і конфігурацію. Розглянемо основні інструменти та платформи, що використовуються для реалізації мікросервісної архітектури, а також їхні переваги та недоліки.

Серед найпопулярніх інструментів та платформ для розробки та підтримки мікросервісної архітектури виділяють:

1. **Docker** – це платформа для автоматизації розгортання додатків у контейнерах, що дозволяє ізолювати додатки та їхні залежності від операційної системи [7].

Переваги:

– **Ізоляція.** Кожен мікросервіс працює в своєму контейнері, що забезпечує ізоляцію і незалежність від інших сервісів.

– **Портативність.** Контейнери можна запускати на будь-якому сервері або хмарній платформі, що підтримує Docker.

– **Швидке розгортання.** Створення та запуск контейнерів відбувається значно швидше порівняно з віртуальними машинами.

Недоліки:

– **Безпека.** Незважаючи на ізоляцію, контейнери використовують загальне ядро операційної системи, що може створювати ризики безпеки.

– **Комплексність.** Управління великою кількістю контейнерів може бути складним завданням без додаткових інструментів для оркестрації.

2. **Kubernetes** – це платформа для оркестрації контейнеризованих додатків, яка автоматизує розгортання, управління та масштабування контейнерів [8].

Переваги:

– **Автоматизація.** Kubernetes забезпечує автоматичне розгортання, масштабування та відновлення контейнерів.

– **Масштабованість.** Підтримує управління великими кластерами контейнерів з можливістю горизонтального масштабування.

– **Спільнота та підтримка.** Широка підтримка спільноти та активний розвиток платформи.

Недоліки:

– **Складність.** Висока кривина навчання для новачків, складність налаштування та управління.

– **Ресурси.** Вимогливість до ресурсів, що може бути проблемою для невеликих проектів або обмежених середовищ.

3. **Istio** – це платформа для управління мережею мікросервісів, яка надає функціональність для балансування навантаження, аутентифікації, авторизації та моніторингу [9].

Переваги:

– **Безпека.** Підтримка TLS-шифрування, аутентифікація та авторизація між сервісами.

– **Моніторинг та трасування.** Інтеграція з інструментами моніторингу та трасування, такими як Prometheus і Jaeger.

– **Політики трафіку.** Гнучке управління трафіком, включаючи маршрутизацію, балансування навантаження та обмеження швидкості.

Недоліки:

– **Складність впровадження.** Висока складність налаштування і інтеграції з існуючими системами.

– **Продуктивність.** Додаткові накладні витрати на управління мережею можуть вплинути на продуктивність додатків.

4. **Prometheus** – це система моніторингу та оповіщення, спеціально розроблена для роботи з контейнеризованими додатками та мікросервісами [10].

Переваги:

– **Інтеграція.** Проста інтеграція з Kubernetes та іншими платформами оркестрації.

– **Ефективність.** Висока ефективність збору та зберігання метрик.

– **Спільнота.** Активна спільнота та велика кількість інтеграцій з іншими інструментами.

Недоліки:

– **Обмеження на довгострокове зберігання.** Нативне зберігання даних може бути обмеженим для довгострокових метрик, вимагає зовнішніх рішень.

– **Складність конфігурації.** Налаштування складних правил оповіщення та агрегування може бути складним для нових користувачів.

5. **ELK Stack (Elasticsearch, Logstash, Kibana)** — це набір інструментів для збору, обробки та візуалізації логів [11].

Переваги [11]:

– **Масштабованість**. Підтримка масштабування для великих обсягів даних.

– **Пошук та аналіз**. Потужні можливості пошуку та аналізу логів за допомогою Elasticsearch.

– **Візуалізація**. Інтуїтивно зрозумілі інструменти для візуалізації даних за допомогою Kibana.

Недоліки [11]:

– **Ресурси.** Вимогливість до ресурсів, особливо при обробці великих обсягів даних.

– **Складність налаштування.** Налаштування та підтримка всіх компонентів може вимагати значних зусиль.

Наведені інструменти та платформи забезпечують потужні можливості для розробки, розгортання та підтримки мікросервісної архітектури, але кожен з них має свої особливості, які слід враховувати при виборі відповідного рішення для конкретного проекту.

Висновки за розділом 1

Мікросервісна архітектура є надзвичайно ефективною для розробки та підтримки складних програмних додатків у порівнянні з традиційним монолітним підходом. У відміну від монолітних додатків, де весь функціонал розміщується в одному додатку, мікросервіси дозволяють розбити програму на невеликі, незалежні сервіси, які можуть бути розгорнуті та масштабовані окремо.

Однією з ключових переваг мікросервісної архітектури є гнучкість. Кожен мікросервіс може бути розроблений, тестований, вдосконалений та відділений від інших, що дозволяє зменшити ризики та підвищити швидкість розробки нових функцій. Крім того, мікросервіси дозволяють використовувати різні технології та мови програмування для кожного сервісу, що підвищує гнучкість розробки.

Однак, мікросервісна архітектура має свої виклики порівняно з монолітним підходом. Управління багатьма невеликими сервісами може бути складним завданням, а синхронізація даних та забезпечення консистентності

може стати проблемою при великому обсязі трафіку. Вартість підтримки та розгортання такої архітектури може бути вищою, ніж у традиційних монолітних додатків.

У порівнянні з монолітною архітектурою, де весь функціонал зосереджений в одному монолітному додатку, мікросервісна архітектура виявляється більш гнучкою, масштабованою та здатною ефективно відповідати на зміни вимог до програмного забезпечення.

РОЗДІЛ 2
РОЗРОБКА МОДЕЛІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ПРОГРАМНОГО ДОДАТКУ

2.1 Архітектурний дизайн та компоненти системи

Система побудована на основі мікросервісної архітектури та складається з семи ключових мікросервісів: Discovery Server, API Gateway, Order Service, Payment Service, Notification Service, Product Service та Authentication Service. Кожен з цих мікросервісів виконує свою унікальну роль та взаємодіє з іншими для забезпечення загальної функціональності системи. Нижче наведено перелік компонентів системи та їх обов'язки в рамках цієї системи.

1. Реєстр сервісів (Discovery Server) відповідає за реєстрацію та виявлення мікросервісів у системі. Для реалізації Discovery Server використовується Eureka Server від Spring Cloud. Цей сервер виконує наступні функції:

1. **Реєстрація мікросервісів**: кожен мікросервіс при запуску реєструється в Eureka Server, надаючи інформацію про свою адресу, порт та інші метадані.

2. **Виявлення мікросервісів:** мікросервіси можуть динамічно знаходити інші мікросервіси через Eureka Server. Це дозволяє забезпечити автоматичне масштабування та оновлення без потреби в ручному налаштуванні конфігурацій.

3. **Моніторинг стану мікросервісів**: Eureka Server періодично отримує від мікросервісів сигнали про їхній стан, що дозволяє виявляти та обробляти збої.

Кожен мікросервіс в системі є клієнтом (Eureka Client) реєстру серверів (Eureka Server). На рисунку 2.1 зображена архітектура роботи Discovery Server, що ілюструє процеси реєстрації, виявлення та моніторингу мікросервісів. Етапи цих процесів позначені числами: реєстрація мікросервісів (1), виявлення мікросервісів (2), моніторинг стану мікросервісів (3).

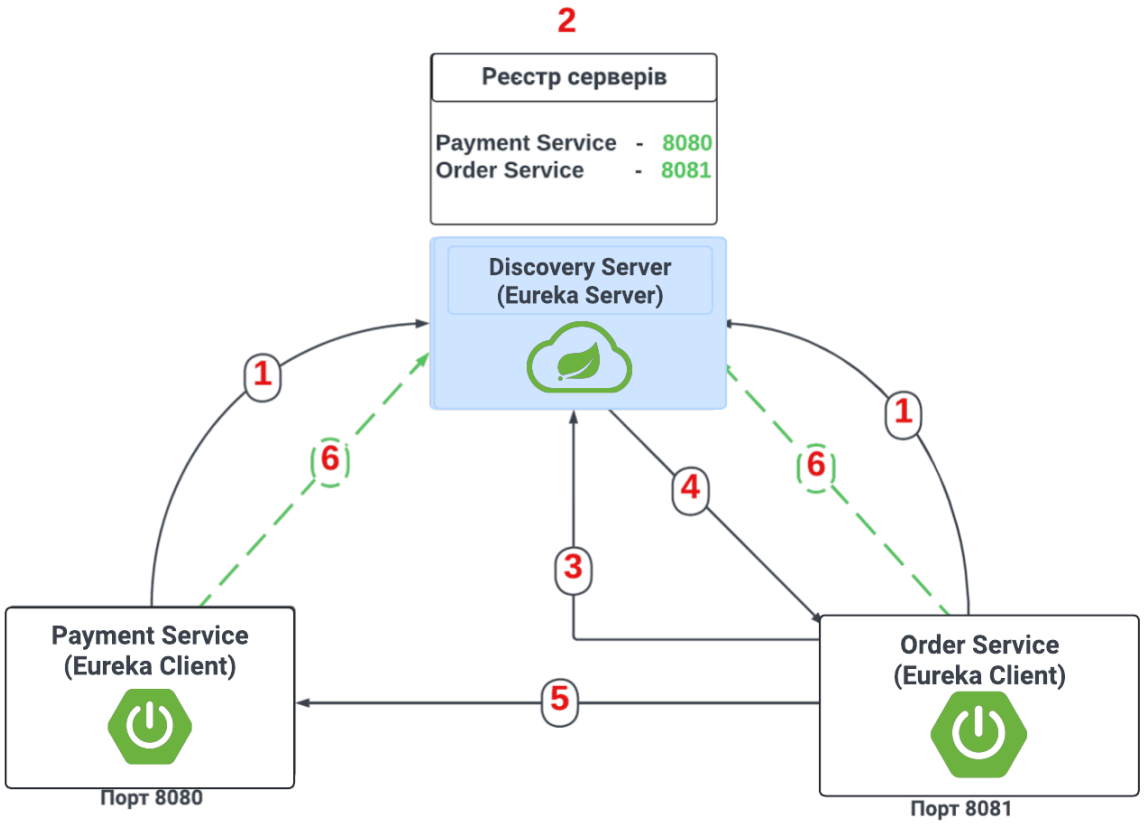

Рисунок 2.1 – Архітектура роботи Discovery Server

На рис. 2.1 позначено кожен з етапів роботи з Discovery Server:

1 – відправка з мікросервісів до Discovery Server запитів на реєстрацію

2 – реєстрація мікросервісів в Discovery Server

3 – Order Service хоче звернутись до Payment Service, тому відправляє запит до Discovery Server для отримання URL, по якому можна звернутись

4 – Discovery Server повертає URL Payment Service до Order Service

5 – Order Service звертається за URL до Payment Service

6 – кожен мікросервіс з певною періодичністю відправляє, так звані, heartbeat (серцебиття) запити, для того, щоб Discovery Server знав, що мікросервіс все ще працює. Якщо Discovery Server не отримує такий запит довгий час, то він помічає мікросервіс, як недоступний.

Використання Eureka Server з Spring Cloud дозволяє створити надійну і гнучку систему управління мікросервісами, що значно полегшує їх інтеграцію та підтримку.

2. API Gateway виконує роль єдиного входу для клієнтських запитів до системи. Він забезпечує маршрутизацію запитів до відповідних мікросервісів, керує аутентифікацією та авторизацією. Використання API Gateway дозволяє знизити навантаження на окремі мікросервіси та підвищити безпеку системи.

3. Сервіс замовлень (Order Service) забезпечує функціональність управління замовленнями. Він приймає запити на створення, оновлення та видалення замовлень, взаємодіє з іншими мікросервісами для перевірки наявності товарів, обробки платежів та надсилання сповіщень про статус замовлень. Цей мікросервіс є одним з ключових компонентів системи, оскільки безпосередньо пов'язаний з основною бізнес-логікою додатку.

4. Платіжний сервіс (Payment Service) займається обробкою платежів. Він інтегрується з платіжною системою Stripe. Цей мікросервіс забезпечує безпечну та надійну обробку платежів, а також взаємодіє з Order Service для підтвердження успішних транзакцій.

5. Сервіс сповіщень (Notification Service) відповідає за надсилання сповіщень користувачам системи через email. Notification Service тісно інтегрується з Order Service для надсилання сповіщень про успішне оформлене замовлення за допомогою асинхронного спілкування через брокер повідомлень Apache Kafka.

6. Сервіс продуктів (Product Service) управляє інформацією про продукти. Він забезпечує CRUD-операції для товарів, включаючи створення, читання, оновлення та видалення даних про продукти.

7. Сервіс аутентифікації (Authentication Service) відповідає за перевірку ідентичності користувачів і надання їм доступу до системи. Він використовується для аутентифікації користувачів, які намагаються отримати доступ до ресурсів системи, і забезпечує безпеку та конфіденційність даних. Сервіс аутентифікації використовується для ідентифікації користувачів і

перевірки їхньої автентичності перед наданням доступу до захищених ресурсів системи. Основна функція сервісу аутентифікації полягає в перевірці введених користувачем даних для входу в систему та наданні або відмові в доступі до ресурсів системи на основі цих даних. У випадку успішної аутентифікації сервіс генерує токен, який користувач може використовувати для отримання доступу до інших частин системи.

Зазначені вище мікросервіси ілюструють ключові компоненти системи. Детальнішу архітектуру та взаємодію між цими компонентами можна побачити на рисунку 2.2. Графічне зображення архітектури допомагає краще зрозуміти розподілену природу системи та роль кожного мікросервісу у цілому функціонуванні додатку.

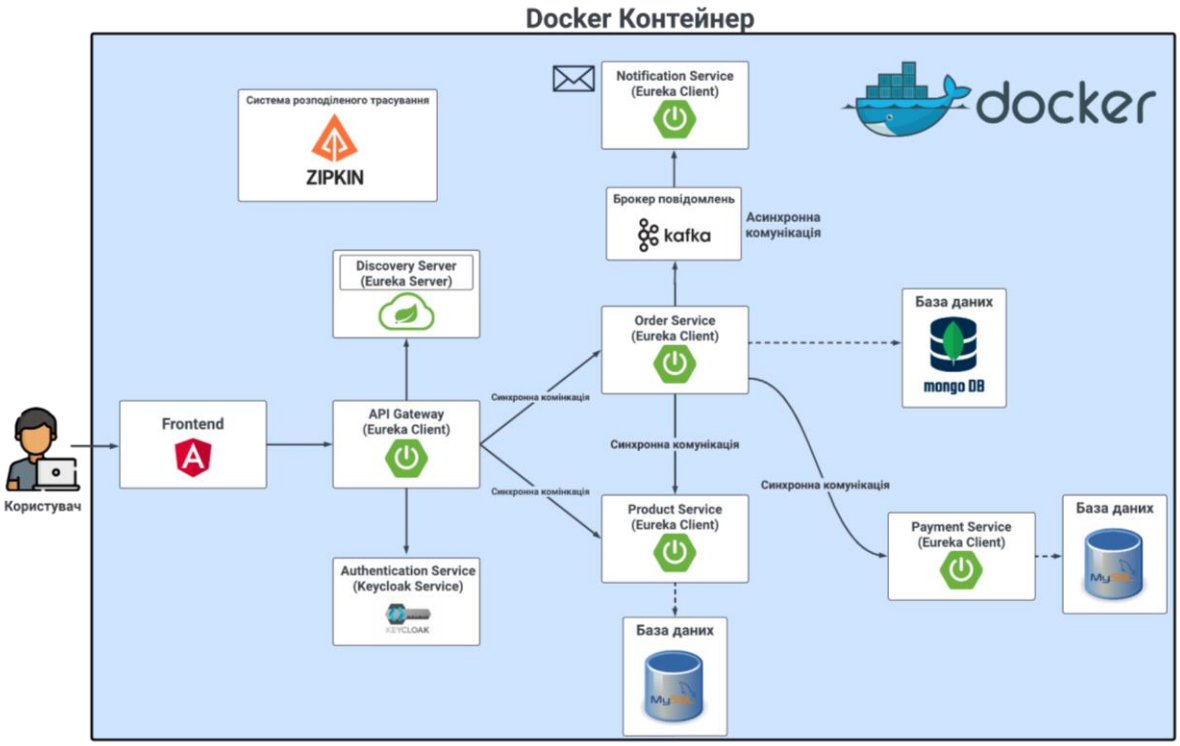

Рисунок 2.2 – Архітектура системи на основі мікросервісної архітектури

2.2 Взаємодія між мікросервісами

Взаємодія між мікросервісами у системі відбувається за допомогою різних механізмів комунікації, що дозволяє забезпечити гнучкість та

надійність архітектури. Кожен механізм має свої переваги та використовується для конкретних потреб системи.

Оскільки для системи не критично наскільки швидко відправиться нотифікація до користувача, для спілкування між Order Service та Notification Service використовується асинхронна комунікація за допомогою Apache Kafka. Order Service може публікувати повідомлення про нові замовлення у Kafka, які потім будуть споживатися Notification Service для відправлення сповіщень покупцям. Використання Apache Kafka дозволяє розділити функціональність та забезпечити надійність взаємодії між сервісами. Крім того, асинхронна комунікація дозволяє зменшити залежність між сервісами та підвищити швидкодію системи в цілому, оскільки сервіси можуть продовжувати свою роботу незалежно одне від одного після відправки повідомлення у Kafka.

Для обміну даними між Order Service та Payment Service, а також між Product Service та Payment Service використовується протокол HTTP. Цей протокол забезпечує синхронну комунікацію між сервісами, де запит та відповідь обробляються негайно.

Використання HTTP для цієї взаємодії дозволяє забезпечити надійність та швидкість обміну даними між сервісами. HTTP є стандартним протоколом для взаємодії в мережі Інтернет, і використання його в мікросервісній архітектурі дозволяє забезпечити сумісність та простоту інтеграції між сервісами.

Синхронна комунікація між Order Service та Payment Service дозволяє отримувати підтвердження від Payment Service про успішну обробку платежу без затримок, що важливо для правильної обробки замовлень та уникнення помилок.

2.3 Забезпечення безпеки та надійності

Для забезпечення безпеки системи використовується сервіс аутентифікації (Authentication Service). Цей сервіс дозволяє аутентифікувати користувача перед доступом до системи. Після успішної аутентифікації

генерується JWT токен, який містить інформацію про користувача та дозволяє ідентифікувати його при подальших запитах до системи.

Аутентифікация системи виконується за наступним сценарієм:

1. **Користувач надсилає запит**: Користувач надсилає запит на доступ до ресурсу через API Gateway.
2. **Перевірка білого списку:** API Gateway перевіряє, чи знаходиться URL запиту в білому списку. Якщо URL адреса знаходиться в whitelist, запит може продовжити обробку без додаткової аутентифікації.
3. **Перенаправлення на сервіс аутентифікації**: Якщо URL ресурсу не знаходиться в білому списку або потрібна додаткова аутентифікація, API Gateway перенаправляє користувача на сервіс аутентифікації.
4. **Аутентифікація**: Сервіс аутентифікації перевіряє дані користувача для підтвердження його ідентичності. Наприклад, використовується JWT токен для аутентифікації.
5. **Отримання токену**: Якщо аутентифікація пройшла успішно, сервіс аутентифікації генерує та повертає JWT токен. Якщо аутентифікація пройшла неуспішно, то користувачу повертається HTTP-статус 401 (Неавторизований)
6. **Повернення на API Gateway**: Користувач повертається на API Gateway з отриманим JWT токеном.
7. **Перевірка токену**: API Gateway перевіряє JWT токен. Якщо токен валідний, запит може продовжити обробку і дозволити доступ до ресурсу. Якщо токен недійсний, доступ може бути відхилений або перенаправлений на сторінку помилки.
8. **Доступ до ресурсу:** Якщо всі перевірки успішно пройдені, API Gateway передає запит до відповідного мікросервісу для обробки запиту.
9. **Відповідь**: Мікросервіс обробляє запит і повертає відповідь, яку API Gateway передає користувачеві.

Під час розробки архітектури даної системи були враховані всі можливі випадки несанкціонованого доступу до мікросервісів. До білогосписку увійшли всі URL сервісу замовлень (URL для розміщення замовлення), а

також URL сервісу продуктів для отримання списку продуктів та категорій, оскільки користувач повинен мати можливість отримати цей список. Лише URL для створення продукту або категорії потребує аутентифікації. Також, до білого списку не потрапили сервіс оплати та сервіс нотифікацій, оскільки користувач не повинен мати доступ до них.

Для забезпечення надійності системи реалізовано можливість запуску декількох екземплярів сервісів на різних портах. Це дозволяє системі працювати стабільно навіть у разі відмови одного з екземплярів. При виявленні відмови сервісу, Discovery Server автоматично перенаправляє запити на інший екземпляр, що забезпечує безперебійну роботу системи.

Для ілюстрації ефективності запуску декількох екземплярів сервісів на різних портах у разі відмови одного з них, розглянемо два випадки.

Випадок 1: відсутність додаткових екземплярів (рис. 2.3). У даному випадку мікросервіс має лише один екземпляр. Коли цей екземпляр перестає працювати, система не може забезпечити безперебійну роботу, оскільки відсутні інші екземпляри для обробки запитів. Це може призвести до зупинки частини функціоналу системи та негативно вплинути на її надійність.

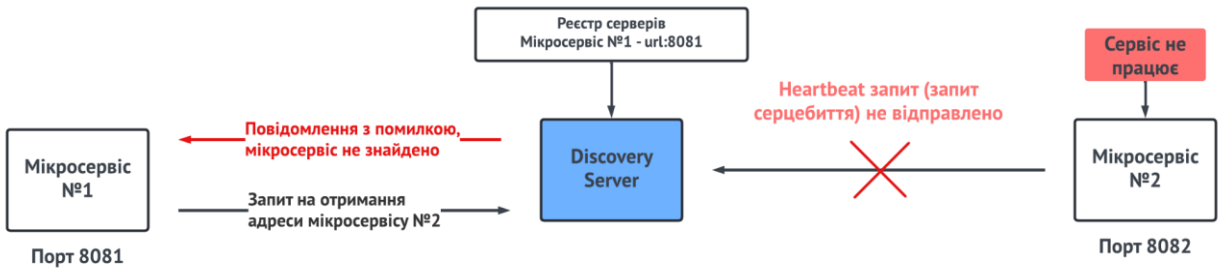

Рисунок 2.3 – Сценарій відсутності додаткових екземплярів

Випадок 2: наявність декількох екземплярів (рис. 2.4). У цьому випадку мікросервіс має кілька екземплярів, які працюють на різних портах. Коли один з екземплярів перестає працювати, Discovery Server автоматично перенаправляє запити на інший працюючий екземпляр. Це дозволяє системі продовжувати працювати стабільно навіть у разі відмови одного з

екземплярів, що підвищує надійність та забезпечує безперебійну роботу системи.

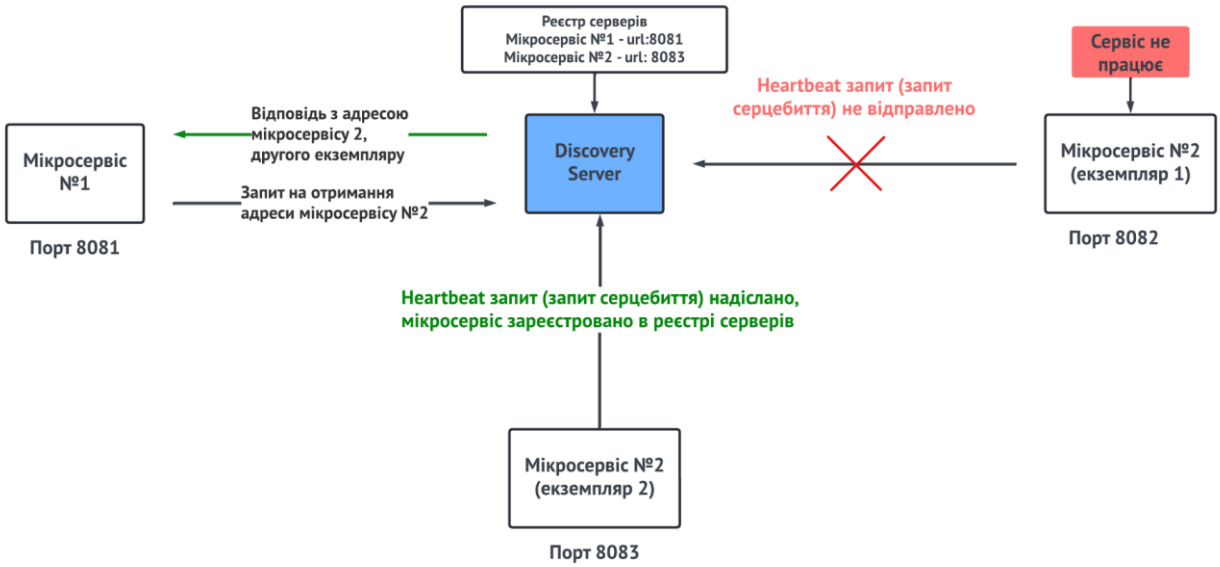

Рисунок 2.3 – Сценарій наявності декількох екземплярів сервісу

Паралельно з цим, варто врахувати, що використання декількох екземплярів сервісів також може створювати додаткові вимоги до механізмів синхронізації та управління ресурсами, що потребує уважного планування та реалізації. Однак, з правильним підходом до архітектури та конфігурації, система може бути надійною і стійкою до можливих відмов, що є ключовим аспектом у забезпеченні безперебійного функціонування великих дистриб'ютивних систем.

Також, для покращення надійності та безпеки, шляхом моніторингу та аналізу взаємодії між мікросервісами, можуть бути використані такі системи відстеження, як Zipkin. Відстеження запитів між сервісами дозволяє виявляти потенційні проблеми безпеки, такі як несанкціонований доступ або недостатній контроль доступу. Крім того, система відстеження допомагає виявляти та усувати неполадки, що забезпечує більшу надійність системи.

2.4 Модель комерційного програмного додатку

Модель комерційного програмного додатку, яка побудована на основі мікросервісної архітектури, дозволяє створювати складні, але гнучкі системи, які легко підтримувати та масштабувати. У такій моделі додаток розбивається на невеликі, незалежні сервіси, кожен з яких відповідає за певну бізнес-логіку. Цей підхід дозволяє підвищити продуктивність та забезпечити відмовостійкість системи.

На рис. 2.4 представлено use case діаграму – модель додатку, яка демонструє основні сценарії взаємодії користувачів з комерційним програмним додатком на основі мікросервісної архітектури.

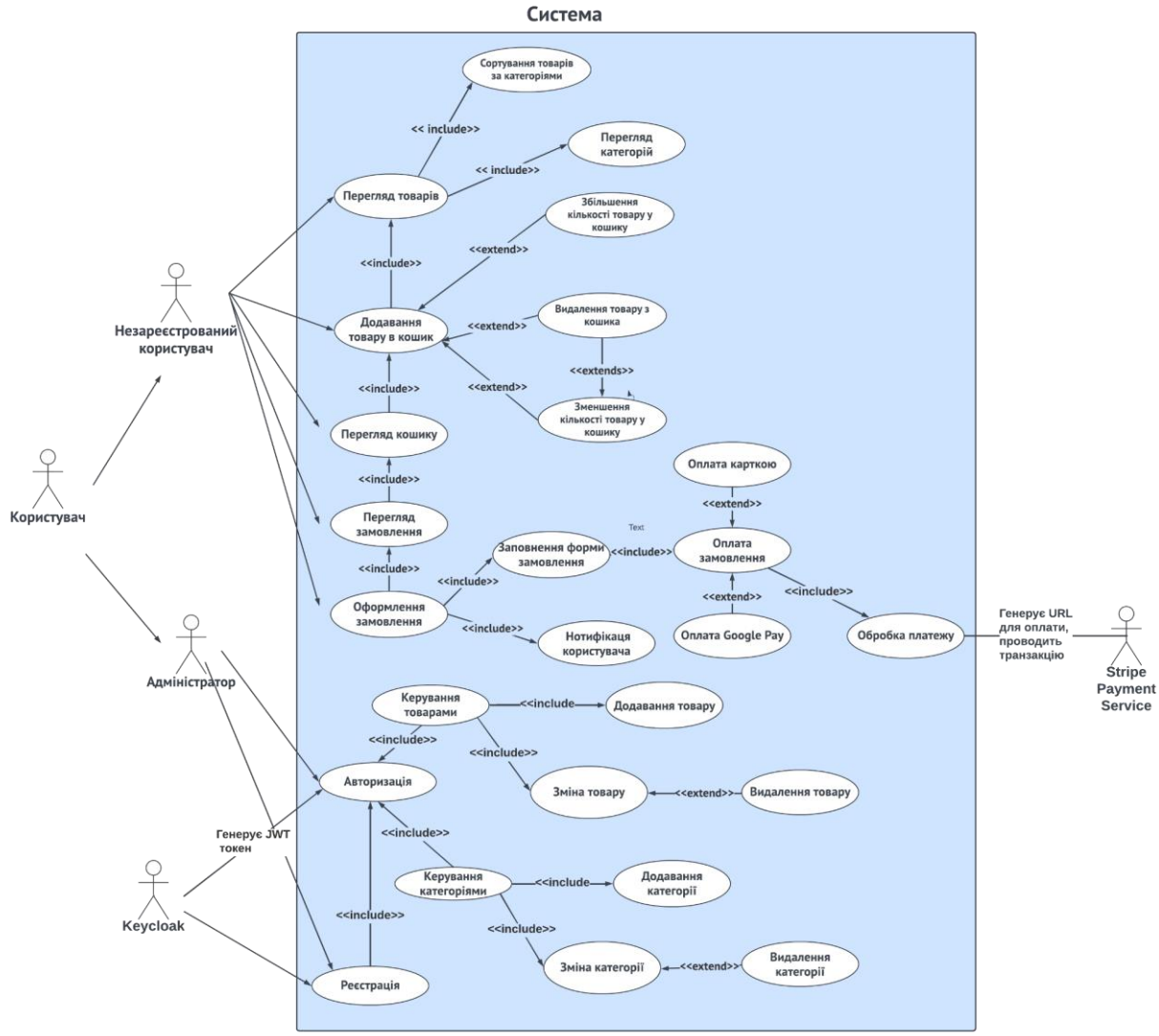

Рисунок 2.5 – Use Case діаграма програмного додатку

Use case діаграма містить такі елементи:

1. Актори – це зовнішні об’єкти, які взаємодіють із системою. У моделі введені такі актори:
    - Незареєстрований користувач – кінцевий користувач системи, який взаємодіє з додатком через клієнтську частину.
    - Адміністратор – користувач з розширеними правами, який виконує адміністрування системи.
    - Зовнішня система – інші системи або сервіси, з якими взаємодіє наш додаток, зокрема:
        i. Stripe Payment Service – платіжний сервіс, що забезпечує обробку платежів та інтеграцію з фінансовими системами.
        ii. Keycloak – сервіс, що відповідає за авторизацію та автентифікацію користувачів.
2. Випадки використання (Use Cases) – це функціональні можливості, які надає система своїм користувачам. Основні випадки використання, представлені на діаграмі, містять:
    - Реєстрація – процес створення нового облікового запису користувача.
    - Авторизація – процес входу користувача в систему.
    - Перегляд товарів – можливість перегляду наявних товаріву системі.
    - Додавання товару до кошика – процес вибору продукту для придбання.
    - Перегляд кошику – можливість перегляду товарів, доданих до кошику.
    - Перегляд замовлення – можливість перегляду замовлення перед оформленням замовлення.

- Оформлення замовлення – завершення процесу покупки та підтвердження замовлення.
- Керування товарами – адміністраторські дії з управління інформацією про товари.
- Керування категоріями – адміністраторські дії з управління інформацією про категорії.
- Оплата замовлення – взаємодія з платіжним сервісом для проведення фінансових транзакцій.

3. Зв’язки між акторами та випадками використання показують, яким чином актори взаємодіють з системою через конкретні випадки використання. Наприклад:

- Користувач взаємодіє з системою через випадки використання "Перегляд товарів", "Додавання товару до кошика", "Перегляд кошику", "Перегляд замовлення", "Оформлення замовлення".
- Адміністратор має доступ до випадків використання "Реєстрація", "Авторизація", "Керування товарами", "Керування категоріями"
- Зовнішня система, зокрема Stripe Payment Service, взаємодіє з додатком через випадок використання "Обробка платежу".
- Зовнішня система, зокрема Keycloak, взаємодії з користувачем через "Авторизація" та "Реєстрація".

Висновки за розділом 2

У даному розділі розглянуто архітектурний дизайн та компоненти системи, взаємодію між мікросервісами, забезпечення безпеки та надійності, а також наведено модель комерційного програмного додатку. Виокремлено ключові аспекти, які впливають на розробку та функціонування мікросервісної архітектури, такі як розділення функціональності та масштабованість. Досліджено принципи відмовостійкості та надійності у контексті розподілених систем.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ МОДЕЛІ КОМЕРЦІЙНОГО ДОДАТКА

3.1 Інструменти і технології

У даному розділі розглядаються інструменти та технології, використані для розробки моделі комерційного програмного додатку на основі мікросервісної архітектури. Вибір відповідних технологій є критичним для забезпечення надійності, масштабованості та ефективності розробленого додатку. Ретельний аналіз та обґрунтований підхід до вибору технологій дозволяє створити стабільну та продуктивну систему, здатну ефективно реагувати на зміни бізнес-вимог та зростання навантаження.

3.1.1 Інструменти та технології для серверної частини

Для побудови серверної частини додатку були обрані наступні інструменти та технології.

Мова програмування: Java – це високорівнева, класоорієнтована, об'єктно-орієнтована мова програмування, яка розроблена з мінімальною кількістю залежностей від реалізації. Це мова програмування загального призначення, яка дозволяє програмістам писати код один раз і запускати його будь-де [12], тобто скомпільований код Java може працювати на всіх платформах, які підтримують Java, без необхідності перекомпіляції [13]. Java-додатки зазвичай компілюються в байт-код, який може виконуватися на будь-якій віртуальній машині Java, незалежно від архітектури комп'ютера [14].

Фреймворки:

1. **Spring Framework** - це фреймворк для розробки додатків та контейнер інверсії управління для платформи Java [15].
2. **Spring Boot** – це відкритий Java-фреймворк, який використовується для розробки автономних додатків виробничого рівня на основі Spring з мінімальними зусиллями [16]. Spring Boot є розширенням платформи Spring, що базується на принципі "конвенція важливіша за конфігурацію" і

призначене для мінімізації налаштувань під час створення додатків на основі Spring [17, 18].

3. **Spring Cloud** - це набір інструментів для розробників, що дозволяють швидко реалізувати деякі загальні шаблони у розподілених системах (наприклад, управління конфігурацією, виявлення сервісів, circuit breakers, інтелектуальна маршрутизація, мікропроксі, контрольна шина, короткоживучі мікросервіси та контрактне тестування). Координація розподілених систем призводить до повторюваних шаблонів, і за допомогою Spring Cloud розробники можуть швидко створювати сервіси та додатки, що реалізують ці шаблони [19].

4. **Hibernate** – це інструмент об'єктно-реляційного відображення для мови програмування Java [20]. Він надає фреймворк для відображення об'єктно-орієнтованої доменної моделі на реляційну базу даних. Hibernate вирішує проблеми невідповідності об'єктно-реляційного відображення, замінюючи прямий доступ до бази даних високорівневими функціями обробки об'єктів [21].

Сторонні сервіси:

Keycloak – продукт з відкритим кодом для реалізації single sign-on з можливістю менеджменту ідентифікації та керування доступом націлений на сучасні застосування і сервіси [22].

3.1.2 Інструменти та технології для клієнтської частини

Для побудови клієнтської частини додатку були обрані наступні інструменти і технології:

Мова програмування: TypeScript – це надбудова JavaScript, яка додає необов'язкове статичне типізування та розширені можливості до мови програмування JavaScript [23].

Платформа: Node.js – платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript [24].

Фреймворк: Angular – це безкоштовний відкритий фреймворк для створення односторінкових веб-додатків на основі TypeScript, що працює на платформі Node.js [25].

3.1.3 Системи управління базами даних

Вибір баз даних для мікросервісів потребує ретельного розгляду з урахуванням вимог до системи, її складності та потреб користувачів. У цьому випадку, обрано використовувати MySQL для всіх мікросервісів, за винятком Order Service, для якого вибрано MongoDB. Такий підхід може бути обґрунтований наступними перевагами:

1. **MySQL для інших мікросервісів**: MySQL є реляційною базою даних з довголітнім і стабільним репутацією. Вона підходить для використання в мікросервісах, які вимагають складних взаємозв'язків між даними, наприклад, для управління користувачами, продуктами та іншою структурованою інформацією [26].

2. **MongoDB для Order Service**: MongoDB відома своєю гнучкістю та здатністю до зберігання неструктурованих даних. Оскільки Order Service може мати складну структуру замовлень з великою кількістю додаткових даних (наприклад, деталі замовлення, історія замовлень тощо), використання MongoDB дозволяє ефективно зберігати та обробляти ці дані без необхідності додаткової обробки.

Такий підхід до вибору баз даних для мікросервісної архітектури може допомогти забезпечити оптимальну продуктивність та ефективність системи, враховуючи специфіку кожного мікросервісу та його потреби в обробці даних.

3.1.4 Інструменти та технології для інфраструктури

Для побудови інфраструктури системи було обрано наступні інструменти:

Docker – це платформа для контейнеризації додатків, що дозволяє створювати, розгортати та управляти контейнерами. Контейнери забезпечують ізоляцію додатків та їх залежностей, сприяючи портативності та

передбачуваності середовища виконання. Використання Docker дозволяє легко масштабувати додаток та автоматизувати процеси розгортання [27].

Apache Kafka – це розподілена платформа для обробки потоків даних у реальному часі. Kafka забезпечує високу продуктивність, стійкість до відмов та можливість горизонтального масштабування. Вона використовується для організації асинхронної комунікації між мікросервісами, що дозволяє ефективно обробляти великі обсяги даних [6].

Zipkin – це інструмент для відстеження розподілених систем, який дозволяє аналізувати та відстежувати шляхи виконання запитів у розподіленому середовищі мікросервісів. За допомогою Zipkin можна визначити, як запити пройшли через різні мікросервіси, як довго вони виконувались в кожному сервісі та які запити викликали проблеми або затримки у виконанні. Zipkin збирає дані про кожен запит, включаючи інформацію про час початку та завершення запиту в кожному сервісі, а також інформацію про залежності між сервісами. Ці дані візуалізуються у вигляді графіка, який дозволяє аналізувати шляхи виконання запитів та виявляти можливі проблеми у роботі системи [28].

3.2 Ключові функціональні можливості програмного додатку

Програмний додаток включає в себе ряд ключових функціональних можливостей, які забезпечують його ефективне використання в комерційних цілях. Серед них:

- **Перегляд списку товарів:** можливість користувачів переглядати повний список доступних товарів з коротким описом кожного з них (рис.3.1).

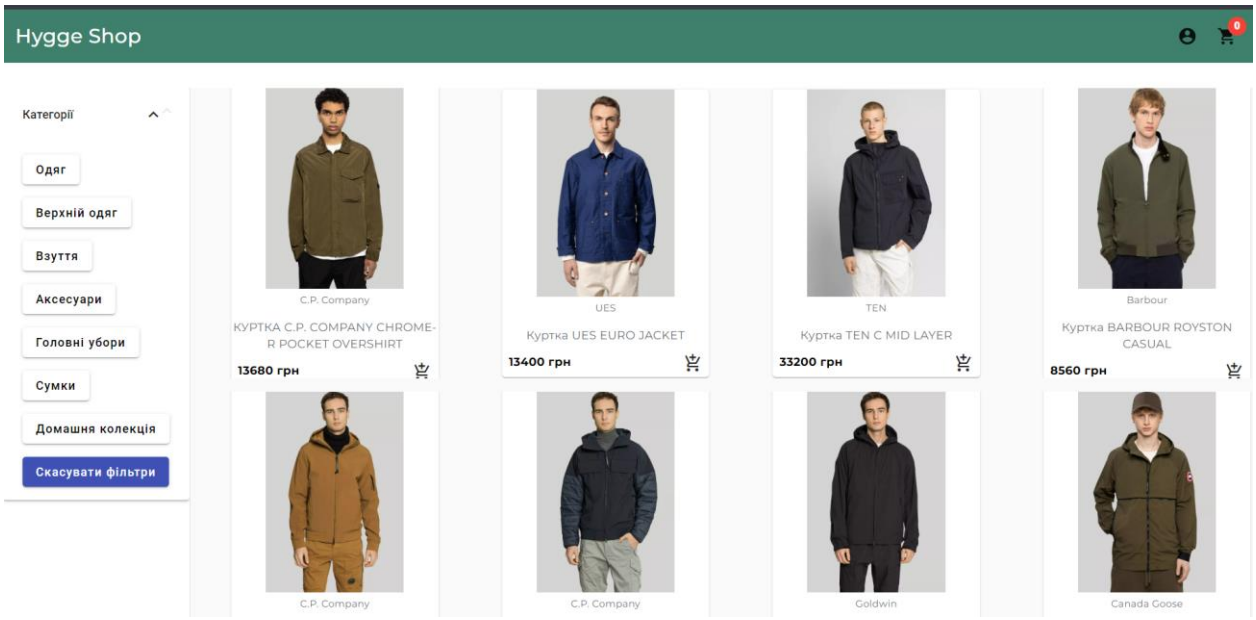

Рисунок 3.1 – Перегляд списку товарів

- **Сортування за категоріями:** користувачі можуть сортувати товари за різними категоріями, що полегшує пошук потрібного товару та покращує зручність використання додатку(рис. 3.2).

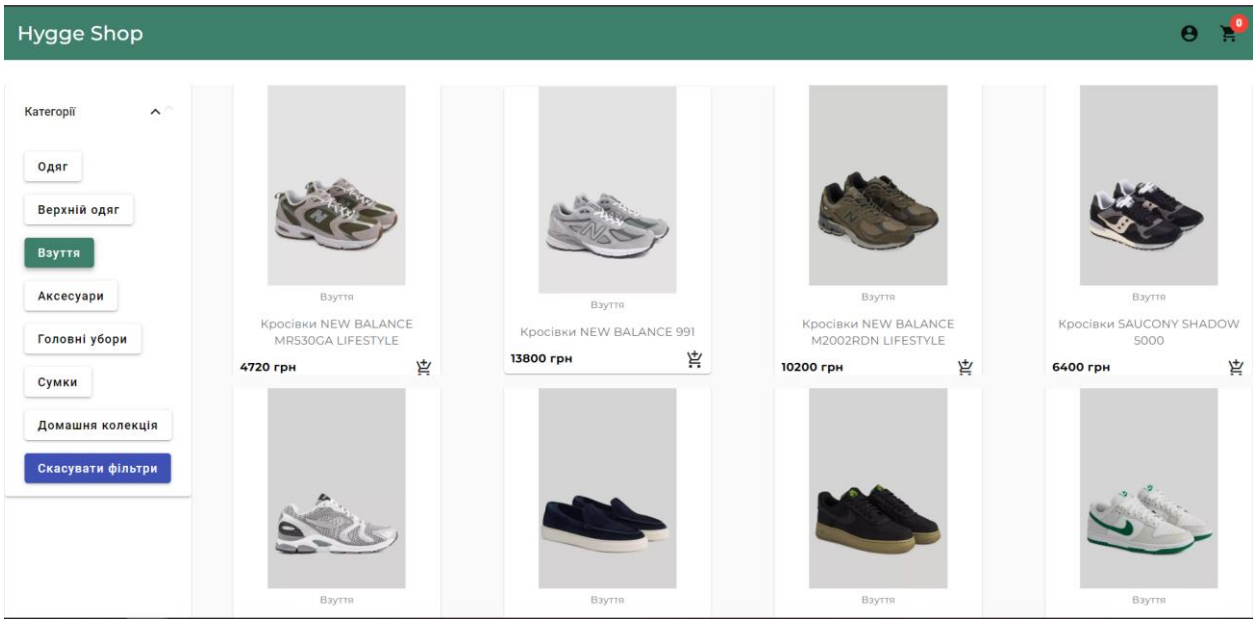

Рисунок 3.2 – Перегляд списку товар за категорією “Взуття”

- **Кошик покупок:** можливість користувачам додавати товари до кошика, змінювати їх кількість та здійснювати замовлення (рис. 3.3).

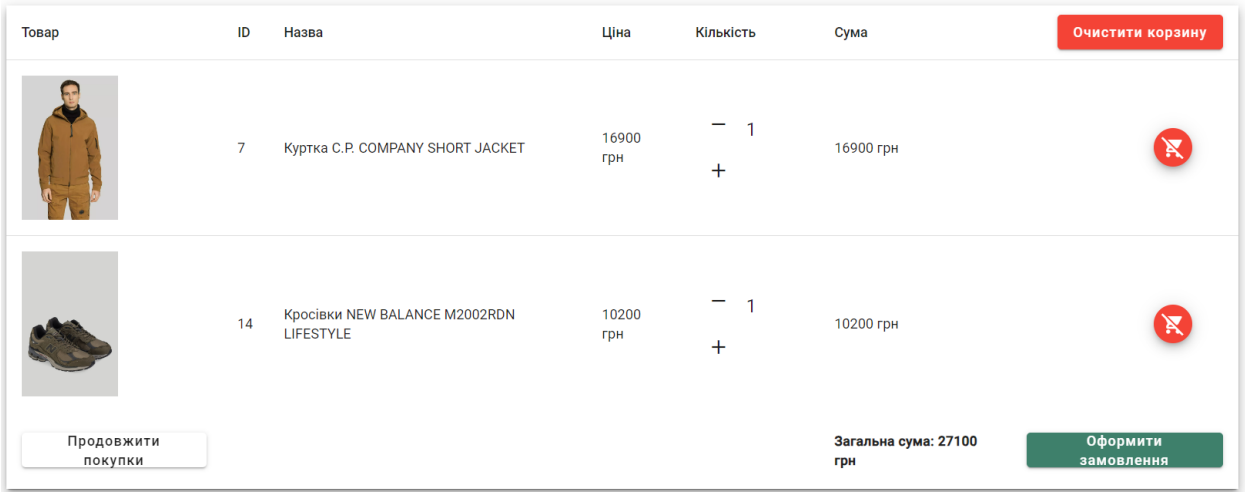

Рисунок 3.3 – Перегляд кошику

- **Обробка замовлень:** автоматизація процесу прийому та обробки замовлень, яка містить у собі сповіщення клієнтів про статус їх замовлень (рис. 3.4).

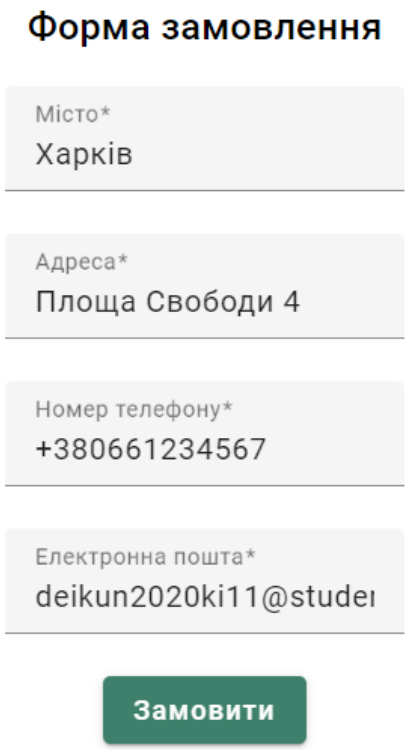

Рисунок 3.4 – Заповнення форми до замовлення

- **Онлайн оплата:** можливість користувачам здійснювати оплату замовлень онлайн, використовуючи різні платіжні системи. Це забезпечує зручність та швидкість обробки платежів, що значно покращує користувацький досвід (рис. 3.5).

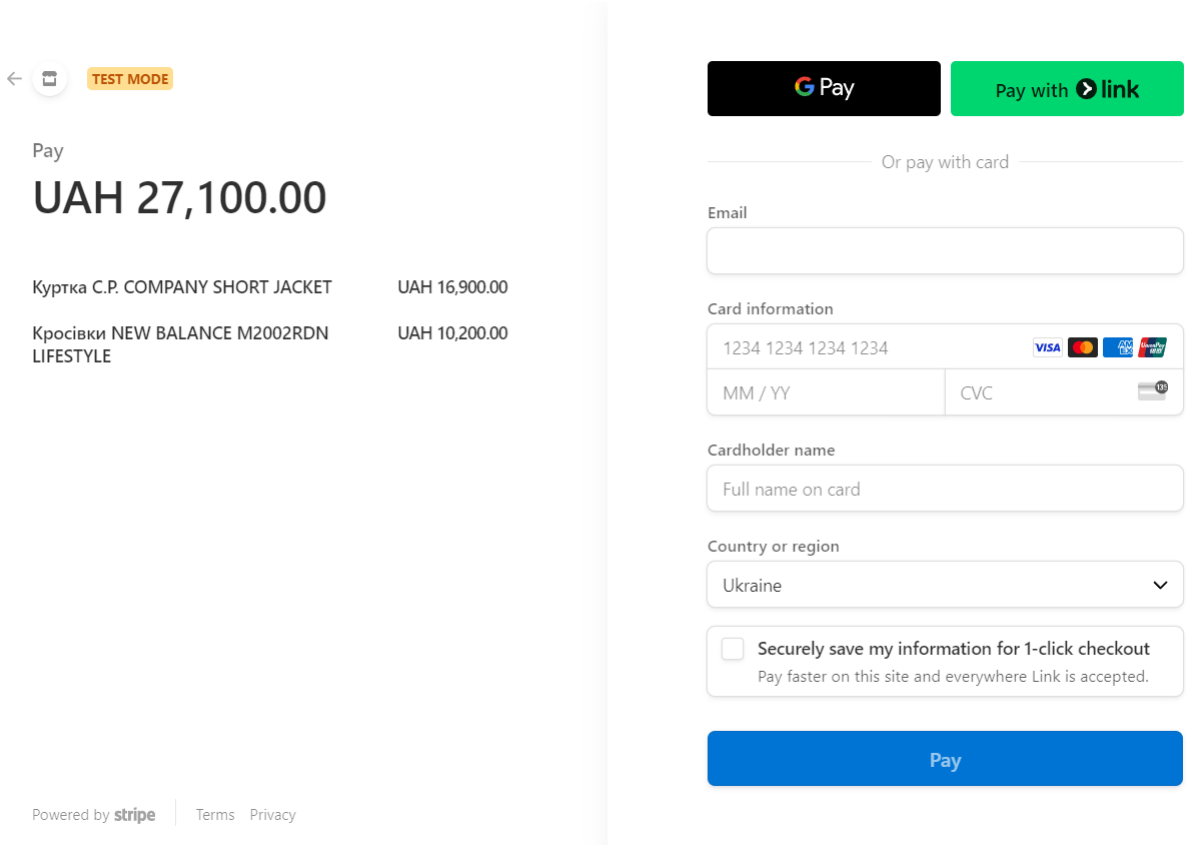

Рисунок 3.5 – Можливість здійснити онлайн оплату замовлення

3.3 Приклад взаємодії користувача з додатком

У цьому підрозділі наведено приклад взаємодії користувача з онлайн-магазином одягу, який ілюструє послідовність дій користувача при взаємодії з сайтом, починаючи з перегляду асортименту товарів і закінчуючи оплатою замовлення. Даний приклад допоможе краще зрозуміти, як працює магазин та які можливості надаються клієнтам для онлайн-покупок у магазині на сайті.

На рисунку 3.6 наведена діаграма дій користувача (User Flow) під час користування додатком.

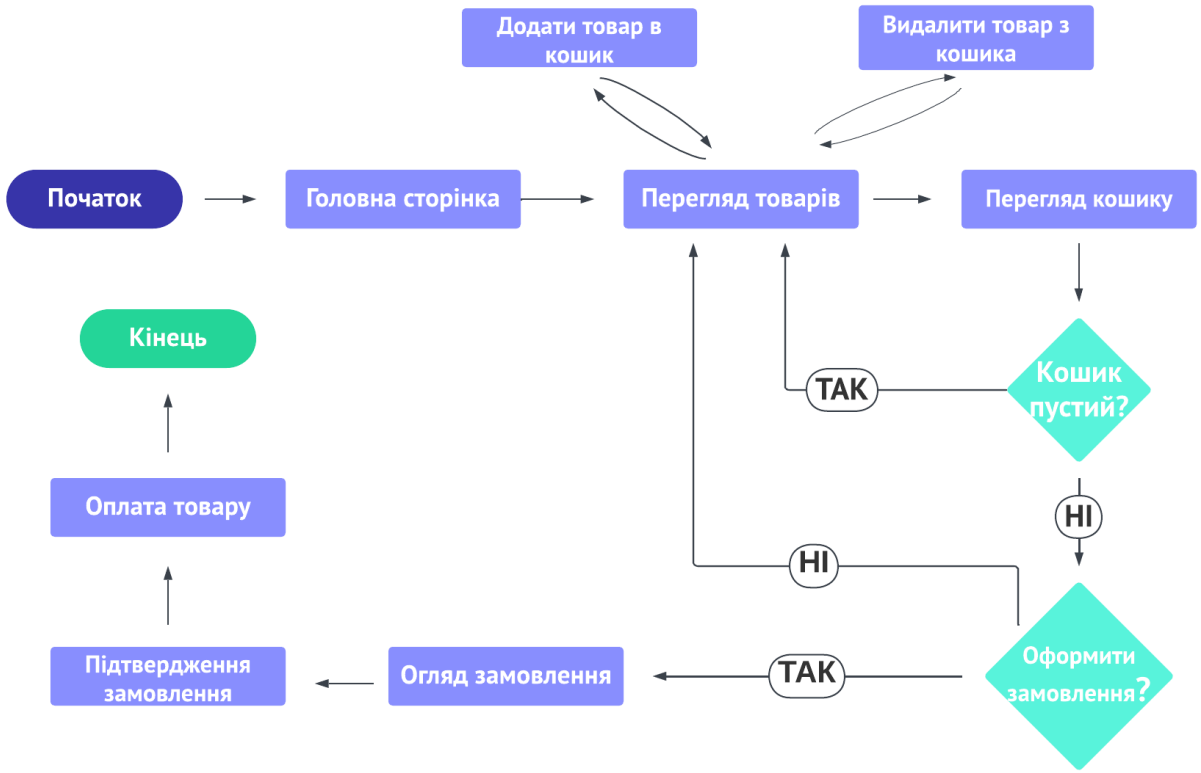

Рисунок 3.6 – Діаграма дій користувача під час користування сайтом

Порядок дій користувача під час користуванням сайту наступний:

1. Користувач заходить на головну сторінку сайту та бачить перед собою список товарів та список категорій, за якими може відфільтрувати товари

2. Користувач переглядає товари, він має можливість додати товари до кошика, або видалити товари з кошика

3. Якщо кошик пустий, то користувачу пропонується далі переглядати товари та обрати, як мінімум один товар для оформлення замовлення

4. Якщо в кошику користувача є хоча б один товар, то він має можливість переглянути своє замовлення та перейти до оформлення замовлення

5. Користувач повторно переглядає товари, які він хоче придбати, на цьому етапі він все ще має можливість змінювати кількість кожного товару в кошику

6. Користувач підтверджує замовлення, заповнює форму, де йому потрібно вказати дані доставки та контактні дані для подальшого уточнення деталей.

7. Користувач переходить до сторінки оплати замовлення, де може ввести дані для оплати

8. У випадку успішної оплати користувач переходить на сторінку, яка інформує його про успішну оплату. У випадку неуспішної оплати, користувач переходить на сторінку, яка інформує його про неуспішну оплату.

3.4 Тестування та відлагодження

У цьому підрозділі проводиться тестування та відлагодження розробленої системи з мікросервісною архітектурою.

Для проведення тестування був обраний інструмент Postman. Postman – це HTTP-клієнт для тестування API. Користуючись Postman можна:

– Складати і відправляти HTTP-запити;

– Створювати колекції та папки запитів для скорочення часу на тестування;

– Змінювати параметри запитів;

– Додавати до виклику API контрольні точки;

– Змінювати оточення для одних і тих же запитів;

– Проводити автоматизоване тестування API з колекції запитів за допомогою Collection Runner.

Для роботи із серверами програма Postman використовує протокол HTTP. Тестувальник відправляє тестові запити від клієнта на сервер та отримує відповідь, чи є помилка в роботі API [29].

Проведення тестування безпеки

Тест 1

Ситуація: неавторизований користувач намагається отримати доступ до публічного ресурсу http://localhost:8080/api/products/categories для отримання списку категорій (рис. 3.7).

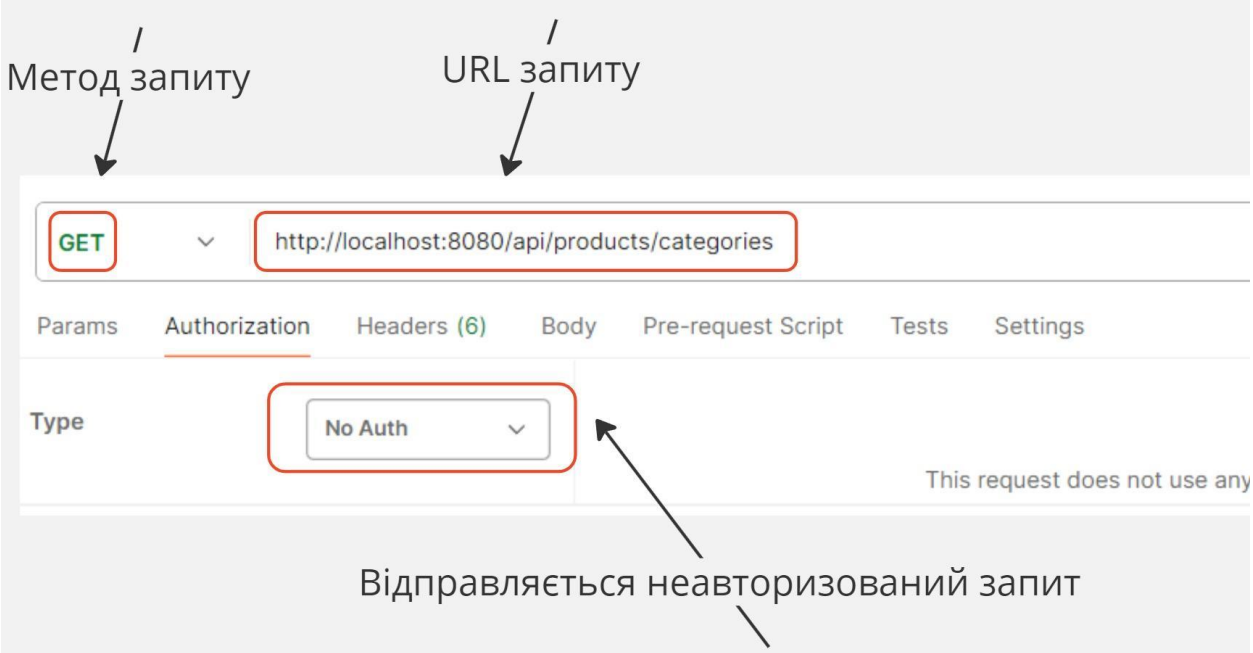

Рисунок 3.7 – Користувач відправляє запит до публічного ресурсу без авторизації

Очікуваний результат: користувач успішно відправляє запит, сервер повертає список категорій та HTTP статус 200 (OK).

Фактичний результат: запит виконався успішно, отримано список категорій і HTTP статус 200 (OK) (рис. 3.8).

Результат виконання тесту 1: тест пройдено успішно.

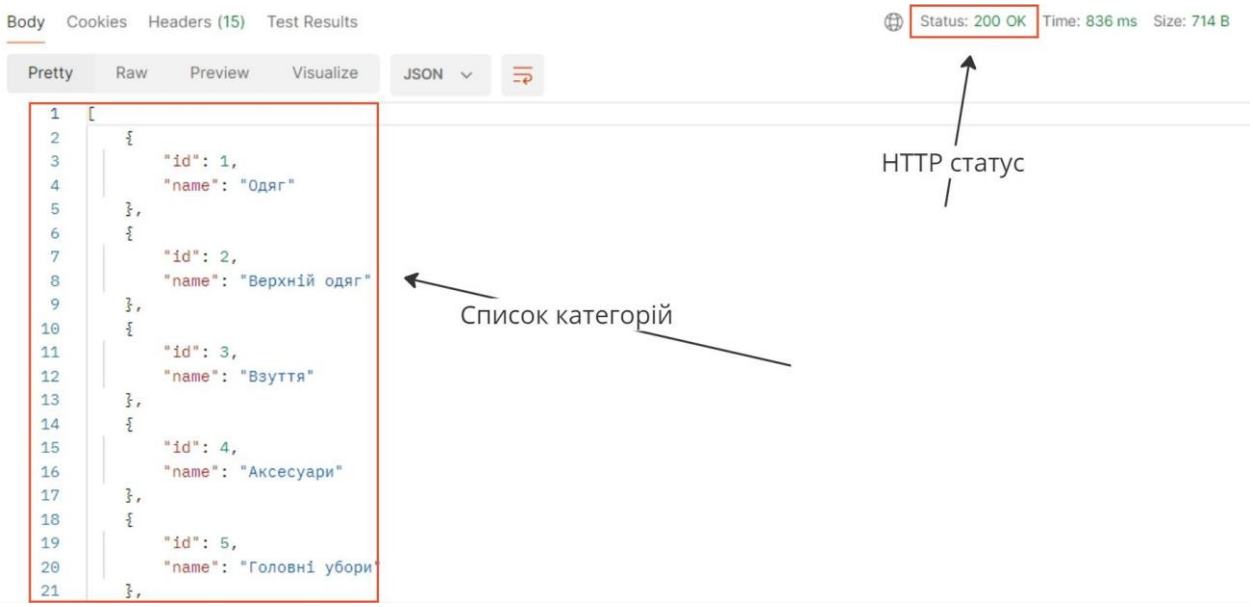

Рисунок 3.8 – Результат виконання запиту на отримання списку категорій без авторизації

Тест 2

Ситуація: неавторизований користувач намагається отримати доступ до закритого ресурсу http://localhost:8080/api/products для створення продукту (рис. 3.9).

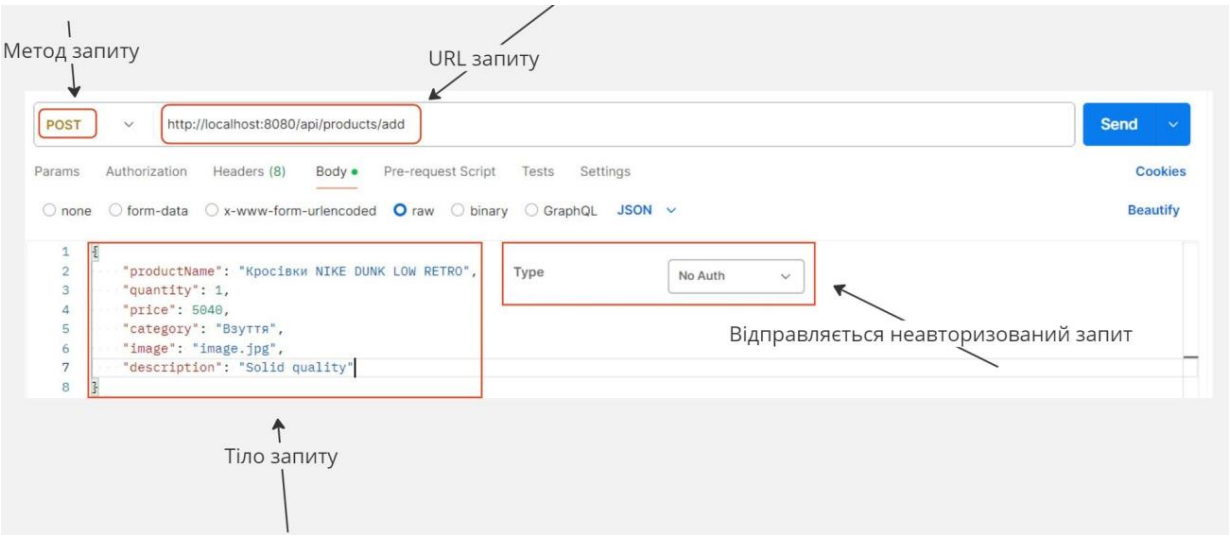

Рисунок 3.9 – Користувач відправляє запит до закритого ресурсу без авторизації

Очікуваний результат: запит повертає у відповідь HTTP статус 401 (Неавторизований), продукт не створюється.

Фактичний результат: запит повернув HTTP статус 401 (Неавторизований), продукт не був створений (рис. 3.10).

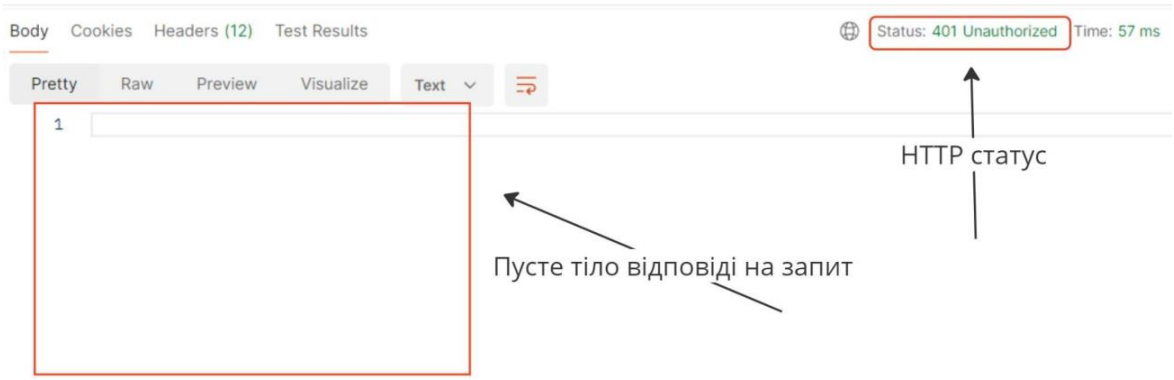

Рисунок 3.10 – Результат виконання запиту на створення продукту без авторизації

Результат виконання тесту 2: тест пройдено успішно.

Тест 3

Ситуація: авторизований користувач намагається отримати доступ до закритого ресурсу http://localhost:8080/api/products для створення продукту (рис. 3.11).

Очікуваний результат: запит повертає у відповідь HTTP статус 201 (Створено), продукт створюється.

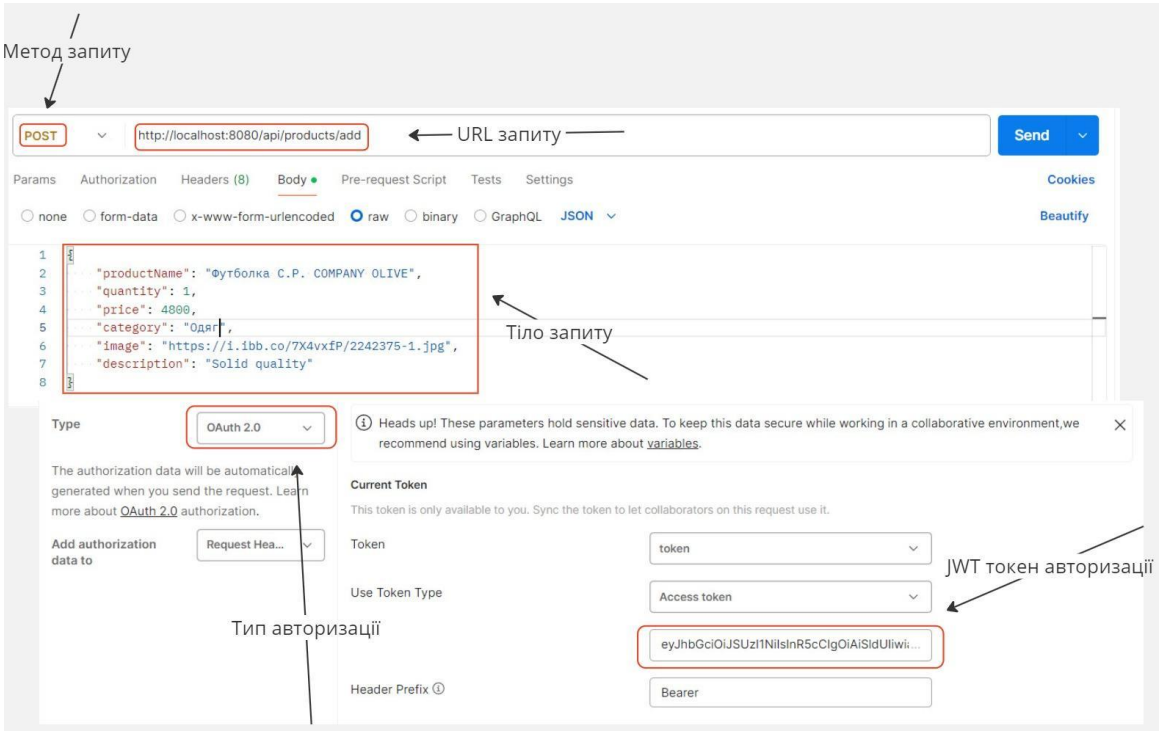

Рисунок 3.11 – Авторизований користувач відправляє запит до закритого ресурсу

Фактичний результат: запит повернув HTTP статус 201 (Створено) (рис. 3.12), продукт створився (рис. 3.13).

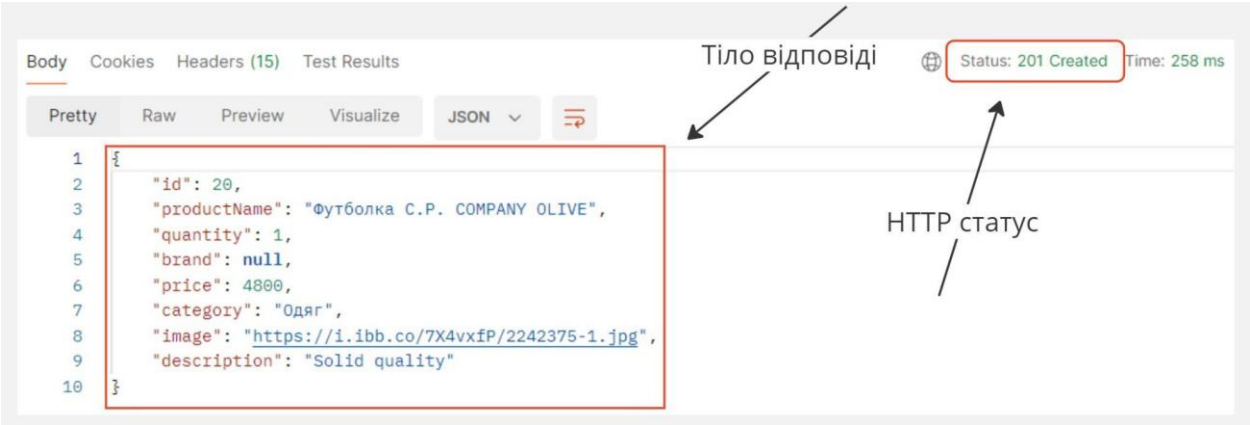

Рисунок 3.12 – Результат виконання авторизованого запиту на створення продукту

Рисунок 3.13 – Результат виконання запиту – створений продукт в магазині

Результат виконання тесту 3: тест пройдено успішно.

Проведення тестування відмовостійкості системи

Тест 4

Ситуація: в запущеному додатку працюють всі сервіси(рис. 3.14), кожен по одному екземпляру. Сервіс для оплати має два працюючих екземпляри в системі - екземпляр №1 та екземпляр №2. Генерується ситуація, при якій один з екземплярів виходить з ладу та перестає працювати.

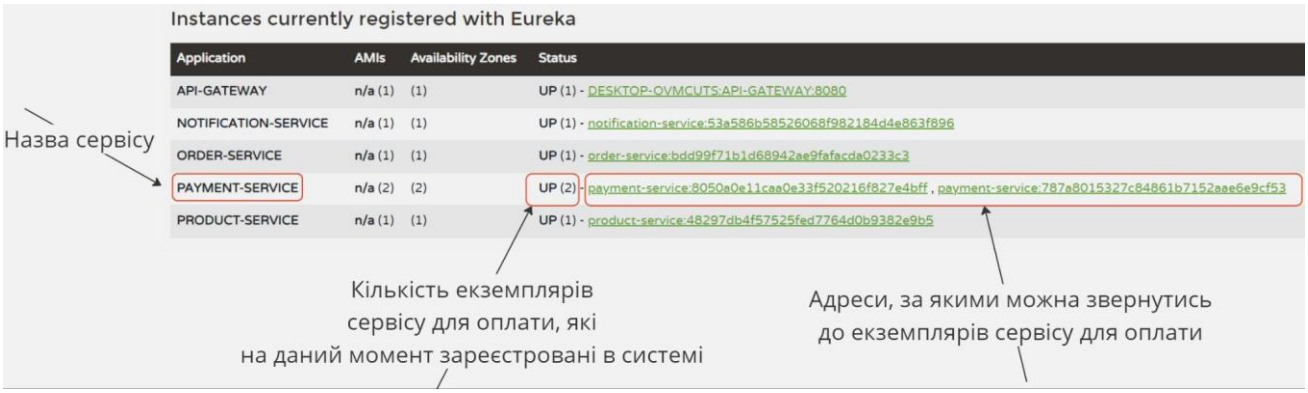

Рисунок 3.14 – Реєстр сервісів в Discovery Server

Очікуваний результат: Після виходу з ладу одного екземпляру сервісу для оплати, система має справно працювати, оформлювати замовлення та приймати оплату надалі. Discovery Server, на запит для отримання адреси сервісу оплати, має повертати адресу другого екземпляру.

Фактичний результат: При коректній роботі двух екземплярів сервісів для оплати маємо успішний результат (рис. 3.16).

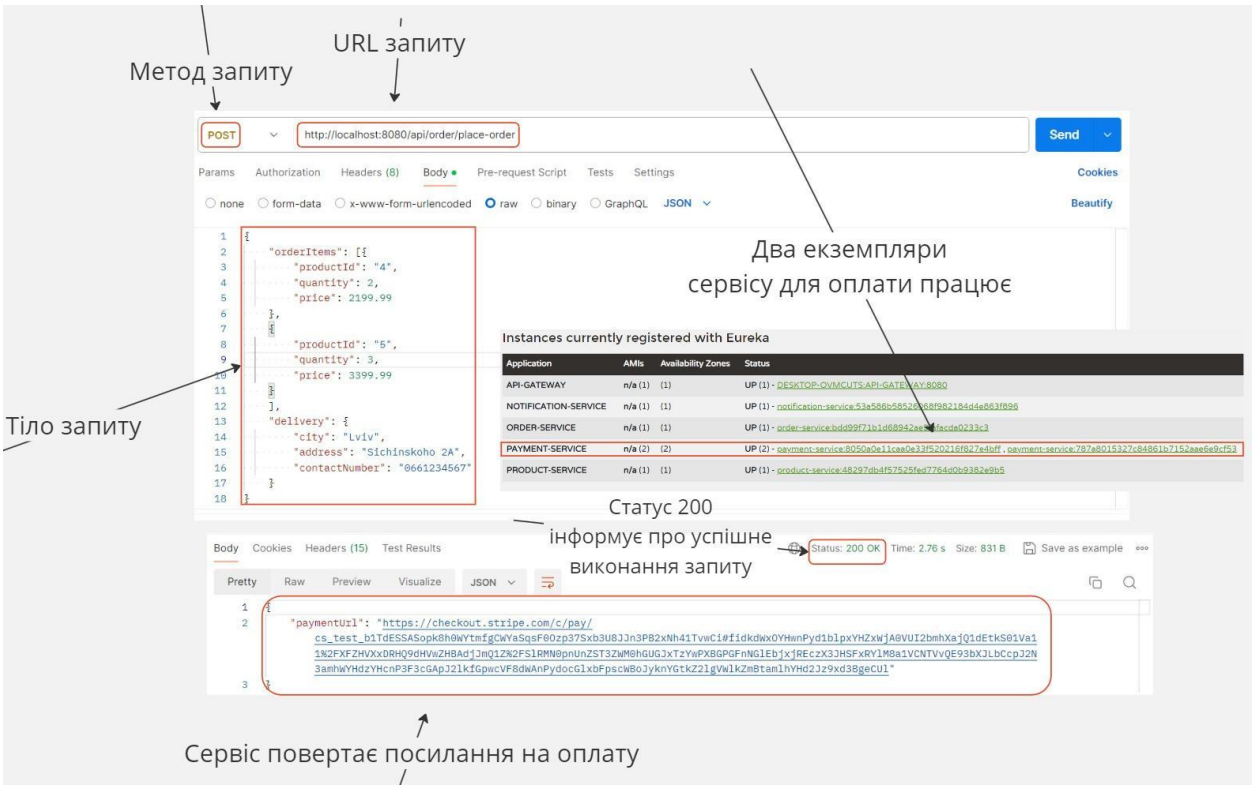

Рисунок 3.16 – Система успішно відправляє запити на оформлення замовлення при коректній роботі двох сервісів оплати

Згенерувавши ситуацію виходу з ладу одного з екземплярів сервісу для оплати (вимикаємо сервіс), маємо успішний результат, система продовжує коректно обробляти запити на оплату (рис. 3.17).

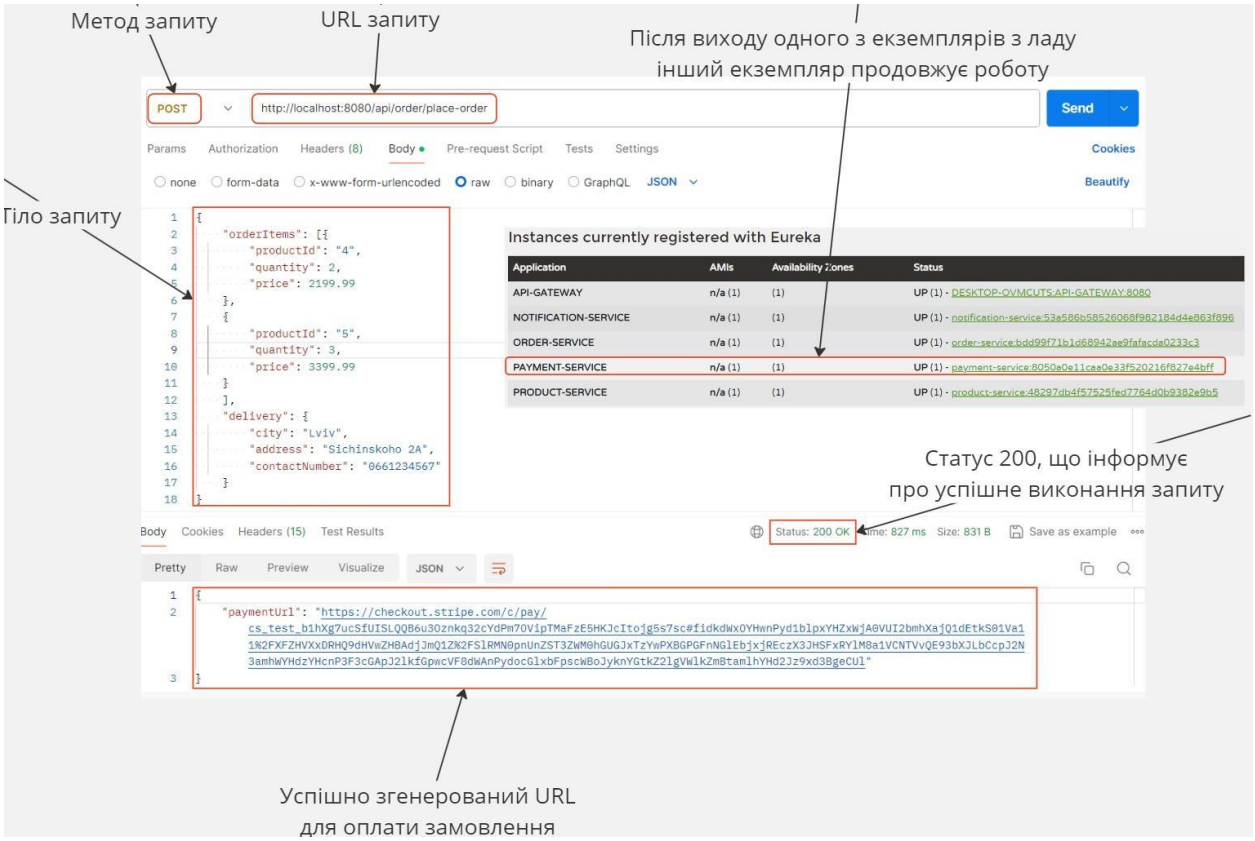

Рисунок 3.17 – Система продовжує коректно працювати при виході з ладу одного з екземплярів

Результат виконання тесту 4: тест пройдено успішно.

Висновки за розділом 3

У розділі 3 було розглянуто інструменти та технології, використані для розробки моделі комерційного програмного додатку на основі мікросервісної архітектури. Вибір відповідних інструментів та технологій є критичним для забезпечення надійності, масштабованості та ефективності розробленого додатку.

Також було розглянуто ключові функціональні можливості програмного додатку, що забезпечують ефективне його використання в комерційних цілях. Представлено можливості перегляду списку товарів, сортування за категоріями, роботи з кошиком покупок, обробки замовлень та онлайн оплати. Описано приклад взаємодії користувача з додатком, що відображає послідовність дій при відвідуванні сайту та здійсненні покупок. Дана

інформація важлива для розуміння функціональності та коректності роботи програмного забезпечення, спрямованого на покращення користувацького досвіду та підвищення ефективності онлайн-торгівлі.

Обрані інструменти та технології дозволили побудувати надійний, масштабований та захищений додаток на основі мікросервісної архітектури, що підтверджується проведеним тестуванням та відлагодженням.

ВИСНОВКИ

У кваліфікаційній роботі виконано детальний аналіз існуючих архітектур програмних додатків, зокрема мікросервісної та монолітної архітектур. Проведено порівняння цих архітектур з точки зору їх переваг та недоліків, способів комунікації між компонентами, а також інструментів та технологій, які використовуються для їх реалізації. На основі аналізу було розроблено та програмно реалізовано модель комерційного програмного додатку, яка базується на мікросервісній архітектурі. В процесі розробки використано сучасні інструменти та технології, що дозволяють забезпечити високу продуктивність, гнучкість та масштабованість додатку.

Особлива увага була приділена моделі відмовостійкості. Було розроблено і впроваджено механізми, які забезпечують надійну роботу додатку навіть у випадку збоїв окремих компонентів. Для цього використано технології реплікації та балансування навантаження, а також методи автоматичного відновлення роботи після відмов. Виконане тестування та аналіз показали, що розроблена модель дозволяє досягти високого рівня надійності та відмовостійкості додатку. Завдяки використанню мікросервісної архітектури, у майбутньому стане можливим швидко адаптувати програмний продукт до змін у вимогах та легко інтегрувати нові функції без суттєвого впливу на існуючі компоненти.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bass, L., Clements, P., & Kazman, R. Software Architecture in Practice (3rd ed.). Addison-Wesley Professional.

2. Мікросервісна архітектура ПЗ [Електронний ресурс]. – режим доступу: URL: https://qalight.ua/baza-znaniy/shho-take-mikroservisna-arhitektura-pz/ (дата звернення – 29.05.2024).

3. Palahno, A. Мікро, міні та макросервіси. Що за чим стоїть та що обрати для проєкту [Електронний ресурс]. – режим доступу: URL: https://dou.ua/forums/topic/38217/ (дата звернення – 24.05.2024).

4. Suleymani, T. Microservice architecture: Synchronous and Asynchronous communication [Електронний ресурс]. – режим доступу: URL: https://www.c-sharpcorner.com/article/microservice-architecture-synchronous-and-asynchronous-communication/ (дата звернення – 28.05.2024).

5. Pivotal Software, Inc. RabbitMQ Documentation [Електронний ресурс]. – режим доступу: URL: https://www.rabbitmq.com/documentation.html (дата звернення – 24.05.2024).

6. Apache Software Foundation. Apache Kafka Documentation [Електронний ресурс]. – режим доступу: URL: https://kafka.apache.org/documentation/ (дата звернення – 24.05.2024).

7. Docker, Inc. Docker Documentation [Електронний ресурс]. – режим доступу: URL: https://docs.docker.com/ (дата звернення – 24.05.2024).

8. Kubernetes Documentation [Електронний ресурс]. – режим доступу: URL: https://kubernetes.io/uk/docs/home/ (дата звернення – 24.05.2024).

9. Istio Documentation [Електронний ресурс]. – режим доступу: URL: https://istio.io/latest/docs/ (дата звернення – 24.05.2024).

10. Prometheus Documentation [Електронний ресурс]. – режим доступу: URL: https://prometheus.io/docs/introduction/overview/ (дата звернення – 24.05.2024).

11. Hazel, T. 5 Elk Stack Pros and Cons [Електронний ресурс]. – режим доступу: URL: https://www.chaossearch.io/blog/elk-stack-pros-and-cons (дата звернення – 24.05.2024).

12. Write once, run anywhere? Computer Weekly [Електронний ресурс]. – режим доступу: URL: https://www.computerweekly.com/feature/Write-once-run-anywhere (дата звернення – 27.07.2009).

13. Oracle. Design Goals of the Java Programming Language [Електронний ресурс]. – режим доступу: URL: https://www.oracle.com/technetwork/java/architecture-142923.html (дата звернення – 14.01.2013).

14. Wikipedia contributors. Java (programming language) [Електронний ресурс]. – режим доступу: URL: https://en.wikipedia.org/wiki/Java_(programming_language) (дата звернення – 24.05.2024).

15. Deinum, M., Long, J., Mak, G., & Rubio, D. Spring Recipes. Berkeley, CA: Apress. doi:10.1007/978-1-4302-5909-1. ISBN 978-1-4302-5908-4.

16. GeeksforGeeks. Spring Boot Tutorial - Learn Spring Boot [Електронний ресурс]. – режим доступу: URL: https://www.geeksforgeeks.org/spring-boot/ (дата звернення – 24.05.2024).

17. Walls, C. Spring Boot in Action. Manning. ISBN 978-1-61729-254-5.

18. Walls, Craig. Spring Boot in Action. Jan 3, 2016. Manning. ISBN 978-1-61729-254-5, p. vii, §foreword.

19. Spring Team. Spring Cloud Documentation [Електронний ресурс]. – режим доступу: URL: https://spring.io/projects/spring-cloud (дата звернення – 24.05.2024).

20. Bauer, C., King, G., & Gregory, G. Java Persistence with Hibernate. Manning Publications. ISBN 1-61729-045-9.

21. Wikipedia contributors. Hibernate (framework) [Електронний ресурс]. – режим доступу: URL:

https://en.wikipedia.org/wiki/Hibernate_(framework) (дата звернення – 24.05.2024).

22. Wikipedia contributors. Keycloak [Електронний ресурс]. – режим доступу: URL: https://uk.wikipedia.org/wiki/Keycloak (дата звернення – 26.05.2024).

23. Kinsta. What is TypeScript? [Електронний ресурс]. – режим доступу: URL: https://kinsta.com/knowledgebase/what-is-typescript/ (дата звернення – 24.05.2024).

24. Wikipedia contributors. Node.js [Електронний ресурс]. – режим доступу: URL: https://uk.wikipedia.org/wiki/Node.js (дата звернення – 24.05.2024).

25. Wikipedia contributors. Angular (web framework) [Електронний ресурс]. – режим доступу: URL: https://en.wikipedia.org/wiki/Angular_(web_framework) (дата звернення – 24.05.2024).

26. Wikipedia contributors. MySQL [Електронний ресурс]. – режим доступу: URL: https://uk.wikipedia.org/wiki/MySQL (дата звернення – 24.05.2024).

27. Kinsta. What is Docker? [Електронний ресурс]. – режим доступу: URL: https://kinsta.com/knowledgebase/what-is-docker/ (дата звернення – 24.05.2024).

28. Zipkin - Distributed Tracing System [Електронний ресурс]. – режим доступу: URL: https://zipkin.io/ (дата звернення – 25.05.2024).

29. Використання Postman у тестуванні. QATestLab training center [Електронний ресурс]. – режим доступу: URL: https://training.qatestlab.com/blog/technical-articles/use-postman-in-testing/ (дата звернення – 26.05.2024).

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет комп'ютерних наук
Кафедра теоретичної та прикладної системотехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Галузь знань: 12 – Інформаційні технології
Спеціальність: 123 – Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри теоретичної
та прикладної системотехніки
_____ д.т.н., проф. Шматков С. І.
«21» грудня 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дейкуна Дмитра Юрійовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи «**Модель комерційного програмного додатку на основі мікросервісної архітектури**»

керівник роботи Стрілець Вікторія Євгенівна, канд. техн. наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «03» травня 2024року № 4101-5/909

2. Строк подання студентом роботи 31 травня 2024року

3. Перелік питань, які потрібно розробити
1) Аналіз існуючих архітектур програмного забезпечення, їх особливостей і обмежень.
2) Розробка моделі комерційного програмного додатку на основі мікросервісної архітектури.
3) Програмна реалізація моделі комерційного програмного додатку.
4) Тестування створеного комерційного програмного додатку та надання рекомендацій щодо його використання.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Аналіз існуючих архітектур програмного забезпечення, їх особливостей і обмежень	16.02.2024 - 25.02.2024
2	Порівняння мікросервісної та монолотіної архітектури комерційних програмних додатків	26.02.2024 - 05.03.2024
3	Вивчення мікросервісної архітектури	06.03.2024 - 25.03.2024
4	Аналіз відомих комерційних програмних додатків, визначення вимог до їх створення	26.03.2024 - 01.04.2024
5	Вибір технічного стеку для розробки комерційного додатку	02.04.2024 - 06.04.2024
6	Розробка архітектури комерційного програмного додатку	02.04.2024 - 14.04.2024
7	Створення моделі комерційного програмного додатку на основі мікросервісної архітектури та її програмна реалізація	15.04.2024 - 10.05.2024
8	Оформлення пояснювальної записки	10.05.2024 - 13.05.2024
9	Передзахист кваліфікаційної роботи	05.2024
10	Представлення кваліфікаційної роботи керівнику та рецензенту	05.2024

5. Дата видачі завдання 21.12.2023 р.

Студент Д.Ю. Дейкун
ініціали, прізвище

підпис

Керівник роботи В.Є. Стрілець
ініціали, прізвище

підпис

Додаток Б

Затверджую
_____
«_____»_____2024 р.

Технічне завдання
на розробку програмного виробу «Модель комерційного програмного додатку на основі мікросервісної архітектури**»**

1.	Вступ	1.1. Назва програмного вироби – Модель комерційного програмного додатку на основі мікросервісної архітектури 1.2. Галузь застосування – розробка програмного забезпечення
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 123 – Комп'ютерна інженерія 2.2 Завдання на кваліфікаційну роботу бакалавра № 4101-5/909 від «03» травня 2024 року (представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3.	Призначення розробки	3.1. Мета розробки програмного виробу – створення ефективної та масштабованої моделі комерційного програмного додатку, що базується на мікросервісній архітектурі для вирішення конкретних бізнес-задач. 3.2. Призначення програмного виробу – забезпечення високої доступності, масштабованості та стабільності застосунку шляхом розподіленої архітектури мікросервісів. 3.3. Початкові дані для розробки: вимоги до функціональності та характеристики сервісів, які будуть включені до архітектури мікросервісів.
4.	Технічні вимоги до програмного виробу	4.1. Вимоги до функціональних характеристик: 1) Розробити програмну реалізацію, що відповідає вимогам бізнесу та специфікаціям мікросервісної архітектури. 2) Додати вірогідну оцінку тривалості проекту з урахуванням розробки, тестування та впровадження системи. 4.2. Вимоги до надійності: 1) Система повинна бути відмовостійкою та оброблювати можливі відмови чи помилки без перерви у роботі

<table>
<tr><td></td><td></td><td colspan="2">4.3. Вимоги до умов експлуатації: Доступ до глобальної мережі інтернет.
4.4. Вимоги до складу і параметрів технічних засобів: Додаток повинен працювати на будь-якій ОС.
4.5. Вимоги до інформаційної та програмної сумісності: Підтримка OS Linux, MacOS, Windows.
4.6. Вимоги до маркування та упаковки:Відсутні.
4.7. Вимоги до транспортування і зберігання: Відсутні.
4.8. Спеціальні вимоги: Не пред'являються.</td></tr>
<tr><td>5.</td><td>Вимоги до програмної документації</td><td colspan="2">Програмною документацією до виробу «Модель комерційного програмного додатку на основі мікросервісної архітектури» вважати:
1) Справжнє Технічне завдання на розробку програмного вироби (представити у вигляді Додатку А до пояснювальної записки до дипломної роботи).
2) Програму і методику випробувань розробленого програмного вироби (представити у вигляді Додатку В до пояснювальної записки до дипломної роботи).
3) Опис програмного виробу (представити в розділі 3 пояснювальної записки до дипломної роботи).
4) Текст програми (представити в Додатку Г до пояснювальної записки до дипломної роботи).</td></tr>
<tr><td>6.</td><td>Вимоги до техніко-економічних показників</td><td colspan="2">Програмною документацією до виробу «Модель комерційного програмного додатку на основі мікросервісної архітектури» вважати:
1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи).
2) Методику розрахунку інформативності змінних стану (у вигляді глав 3.3 та 3.4 пояснювальної записки до кваліфікаційної роботи).
3) Опис виробу (представити в розділі 3 пояснювальної записки до кваліфікаційної роботи)</td></tr>
<tr><td rowspan="2">7.</td><td rowspan="2">Стадії і етапи розробки</td><td>Дата</td><td>Назва етапу</td></tr>
<tr><td>від 16 лютого 2024 до 25 лютого 2024

від 26 лютого 2024 до 5 березня 2024</td><td>Аналіз існуючих архітектур програмного забезпечення, їх особливостей і обмежень

Порівняння мікросервісної архітектури та монолітної архітектури комерційних програмних додатків</td></tr>
</table>

		від 6 березня 2024 до 25 березня 2024	Вивчення мікросервісної архітектури
		від 26 березня 2024 до 1 квітня	Аналіз відомих комерційних програмних додатків, визначення вимог до їх створення
		від 2 квітня 2024 до 6 квітня 2024	Вибір технічного стеку для розробки комерційного додатку
		від 2 квітня до 14 квітня	Розробка архітектури комерційного програмного додатку
		від 15 квітня 2024 до 10 травня 2024	Створення моделі комерційного програмного додатку на основі мікросервісної архітектури та її програмна реалізація
		від 10 травня 2024 до 13 травня 2024	Оформлення пояснювальної записки
		від 13 травня 2024	Передзахист кваліфікаційної роботи
		від 13 травня 2024	Представлення кваліфікаційної роботи керівнику та рецензенту
8.	Порядок контролю і приймання програмного продукту (моделі)	1) Перевірку ходу розробки програмного виробу керівнику робіт виконувати раз в 3 тижні. 2) Захист розробленої моделі провести на засіданні Атестаційної комісії. 3) Пояснювальну записку подати на паперових носіях в 1 примірнику і в електронному вигляді в 1 примірнику на CD R компакт-диску.	

Виконавець
студент групи КІ- 41
ДЕЙКУН Д.Ю.
_____

Замовник
к. т. н..
СТРІЛЕЦЬ В.Є.
_____

Додаток В

Програма і методика випробувань програмного виробу

«Модель комерційного програмного додатку на основі мікросервісної архітектури»

1. Об'єкт випробувань

1. Назва програмного виробу : «Модель комерційного програмного додатку на основі мікросервісної архітектури»

2. Галузь застосування: Інформаційні технології

3. Перераховані відомості запозичуються з відповідних розділів Технічного завдання.

2. Мета випробувань

Перевірка відповідності функціональності програмної реалізації системи заявленим функціональним можливостям в технічному завданні (Додаток Б до пояснювальної записки до кваліфікаційної роботи).

3. Загальні положення

1. Підстави для проведення випробувань

Підставою для проведення випробувань є наказ про призначення атестаційної комісії.

2. Місце і тривалість випробувань

Приймальні (приймально-здавальні) випробування проводяться на базі комп'ютерного класу кафедри в період роботи атестаційної комісії.

3. Обсяг випробувань

Приймальні випробування програмного вироби проводяться в обсязі відповідному цієї програми і методики випробувань.

4. Організації, які беруть участь у випробуваннях

Приймальні випробування проводяться атестаційною комісією напередодні засідання (або в процесі засідання) за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

Модель повинна задовольняти наступним вимогам:

1. працювати на основних операційних системах: Windows, Linux, MacOS;
2. вимоги до надійності;
3. передбачити захист від некоректних дій користувача;
4. сумісність з іншими програмними продуктами;
5. зменшити об’єм програмного коду необхідного для створення веб-додатків;
6. бути легко розширюваною;
7. елементи програми повинні бути ізольовані одне від одного для зменшення їх впливу на роботи програми під час редагування програмного коду;

8. вимоги до складу і параметрів технічних засобів (не висуваються);
9. вимоги до маркування та упаковки (не висуваються);
10. вимоги до транспортування і зберігання (не висуваються).

Спеціальні вимоги (не пред'являються).

5. Вимоги до програмної документації

Програмною документацією до виробу «Модель комерційного програмного додатку на основі мікросервісної архітектури» вважати:

1. Програмною документацією щодо розроблюваного програмного продукту вважати:
2. справжнє технічне завдання на розробку програми (представити як Додаток Б до пояснювальної записки до кваліфікаційної роботи);
3. Програму і методику випробувань розробленої програми (представити як Додаток В до пояснювальної записки до кваліфікаційної роботи);
4. рекомендацій щодо застосування створеної програмної стандартизації у проєктах (представити в Розділі 3 пояснювальної записки до кваліфікаційної роботи).
5. Текст програми(представити як Додаток Г до пояснювальної записки до кваліфікаційної роботи).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Для проведення випробувань використовується платформа Postman.

6.2 Порядок проведення випробувань

Як правило, випробування проводяться в два етапи:

- ознайомчий (1-й етап);
- випробування програмного виробу (2-й етап).

Перелік перевірок, що проводяться на 1 етапі випробувань, включає в себе:

1. Перевірку комплектності програмної документації.
2. Перевірка комплектності складу програмної документації здійснюється за критерієм наявності зазначеної в ТЗ документації.
3. Перевірку комплектності складу технічних і програмних засобів.
4. Методику проведення перевірок на 1 етапі випробувань.
5. Якість програмної документації перевіряється на відповідність вимогам стандартів ЕСПД.

Перелік перевірок, що проводяться на 2 етапі випробувань, включає в себе:

1. Перевірку відповідності технічних характеристик програми вимогам технічного завдання;
2. перевірку ступеня виконання функціональних вимог до програми;
3. методику проведення перевірок, що входять до переліку по 2 етапу випробувань.

1. Програма працює відповідно до умов експлуатації операційних систем MS Windows, Linux та MacOS.
2. Для роботи необхідний Docker.
3. Порядок проведення випробувань:

3.1. Запуск програми здійснюється за домопогою запуску docker-контейнера, в якому зберігаються всі сервіси.

3.2. Після запуску програми необхідно у браузері комп'ютеру перейти за посиланням: http://localhost:4200;

3.3. У вкладці браузеру з'явиться сторінка онлайн-магазин, де можна взаємодіяти з програмним додатком.

Для проведення випробувань пропонується тест 1, тест 2, тест 3 та тест 4.

Тест 1

Ситуація: Неавторизований користувач намагається отримати доступ до публічного ресурсу http://localhost:8080/api/products/categories для отримання списку категорій (рис. В.1).

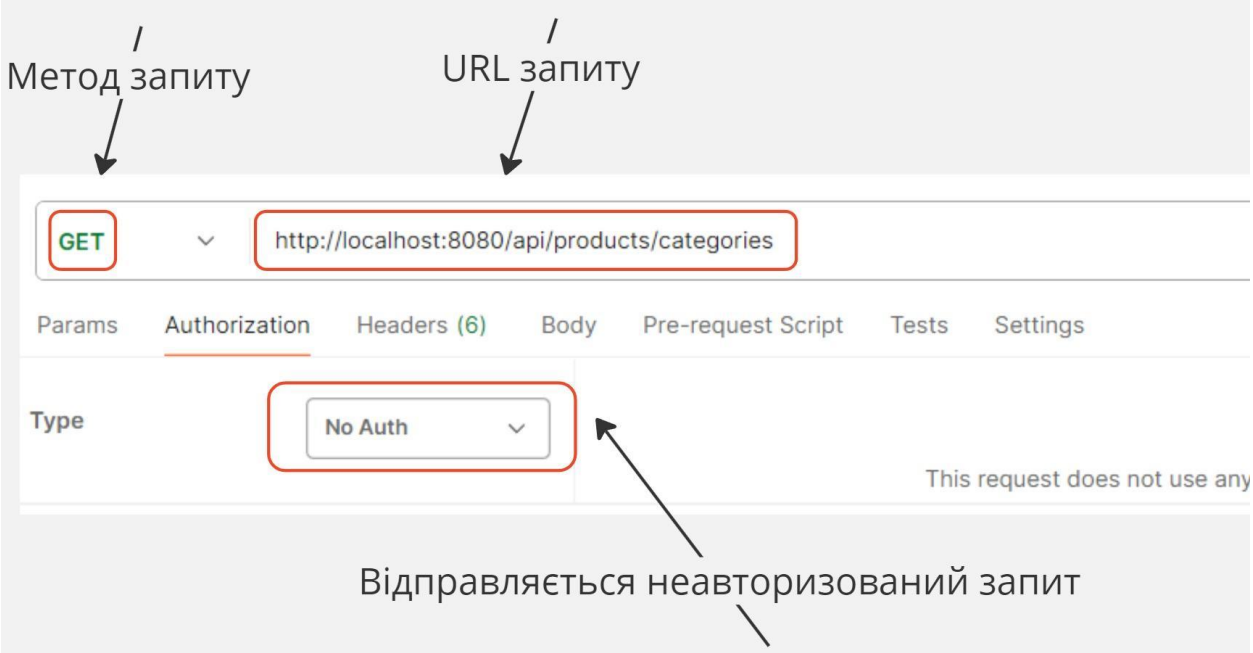

Рисунок В.1 – Користувач відправляє запит до публічного ресурсу без авторизації

Очікуваний результат: Користувач успішно відправляє запит, сервер повертає список категорій та HTTP статус 200 (OK).

Фактичний результат:

Запит виконався успішно, отримано список категорій і HTTP статус 200 (OK) (рис. В.2)

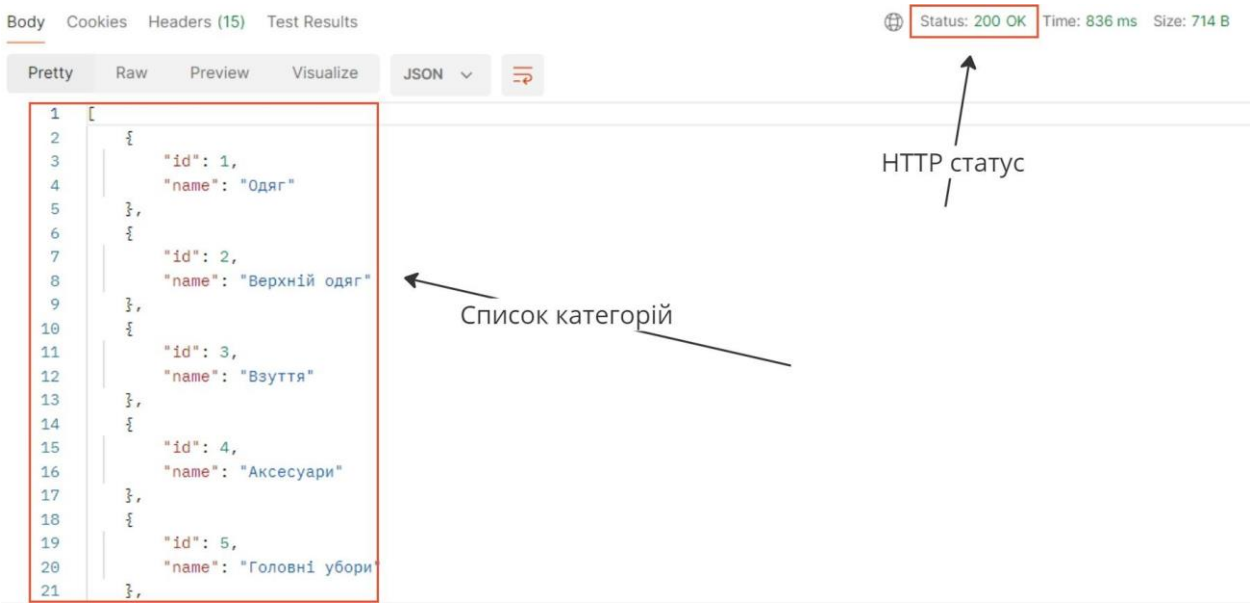

Рисунок В.2 – Результат виконання запиту на отримання списку категорій без авторизації

Результат виконання тесту 1: Тест пройдено успішно

Тест 2

Ситуація: Неавторизований користувач намагається отримати доступ до закритого ресурсу http://localhost:8080/api/products для створення продукту (рис. В.3).

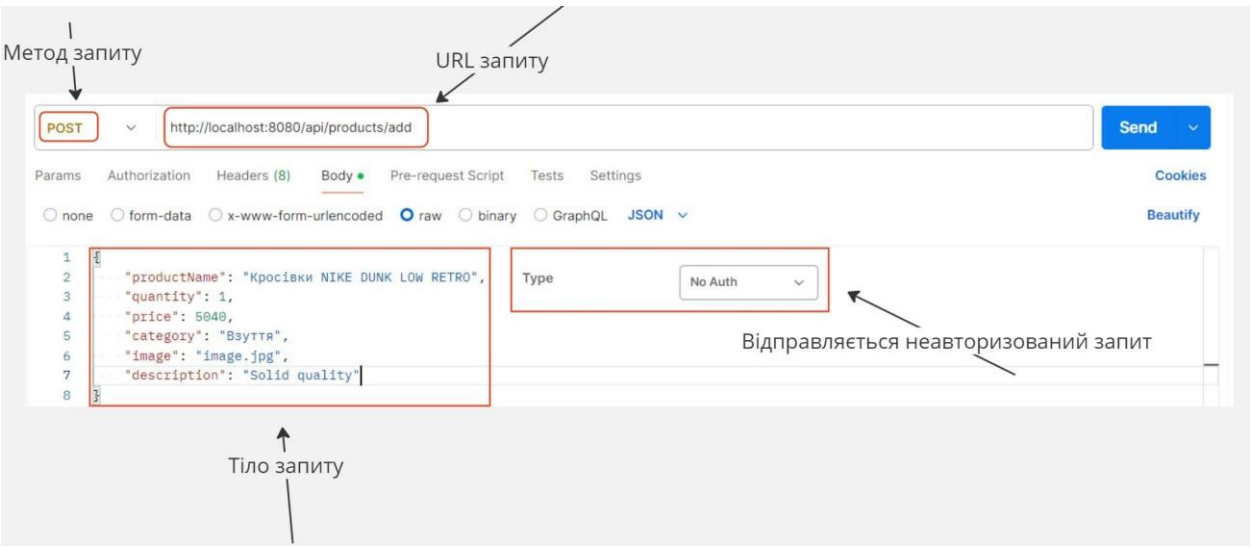

Рисунок В.3 – Користувач відправляє запит до закритого ресурсу без авторизації

Очікуваний результат: Запит повертає у відповідь HTTP статус 401 (Неавторизований), продукт не створюється.

Фактичний результат: Запит повернув HTTP статус 401 (Неавторизований), продукт не створився (рис. В.4).

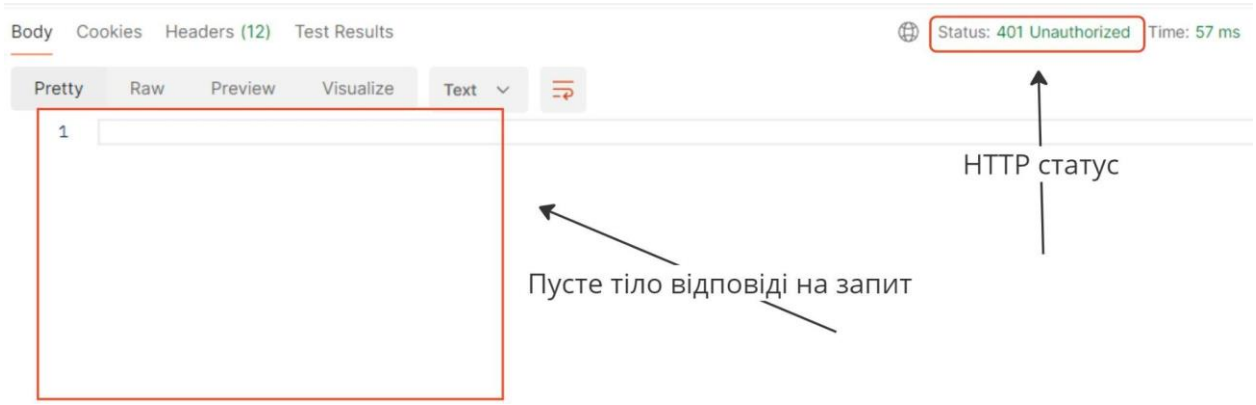

Рисунок В.4 – Результат виконання запиту на створення продукту без авторизації

Результат виконання тесту 2: Тест пройдено успішно

Тест 3

Ситуація: Авторизований користувач намагається отримати доступ до закритого ресурсу http://localhost:8080/api/products для створення продукту (рис. В.5).

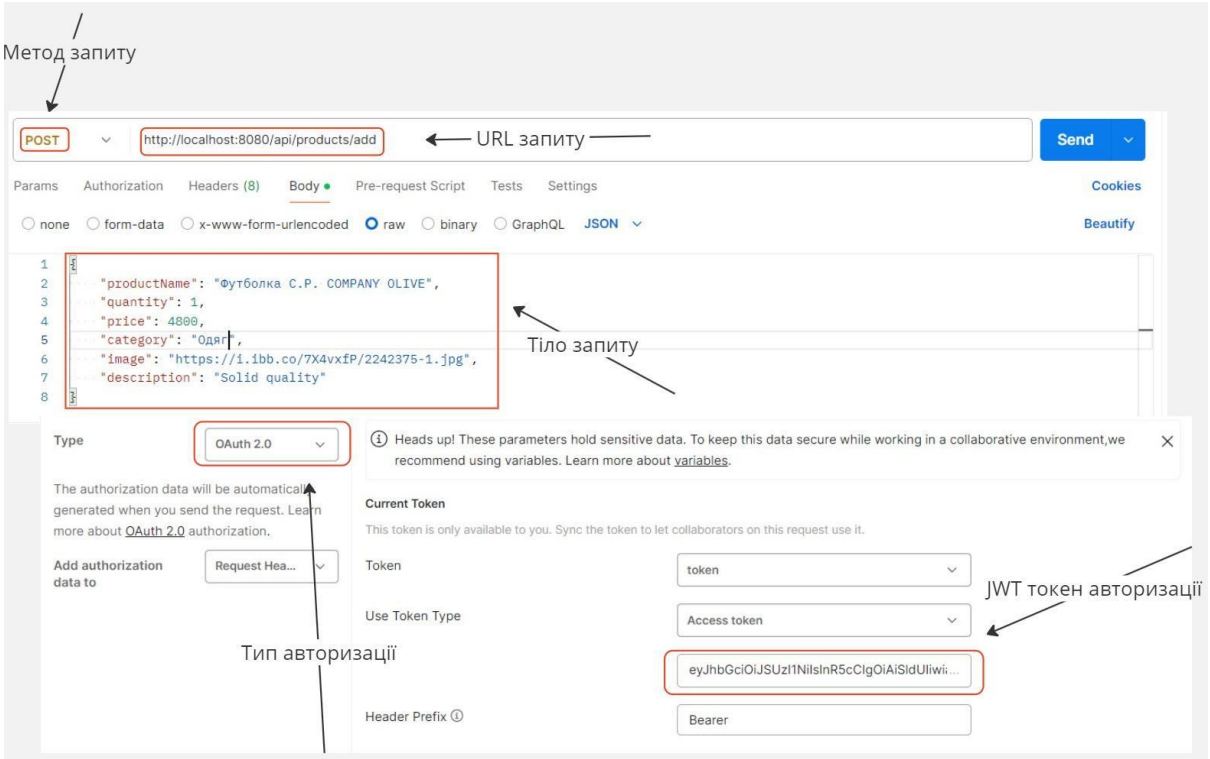

Рисунок В.5 – Авторизований користувач відправляє запит до закритого ресурсу

Очікуваний результат: Запит повертає у відповідь HTTP статус 201 (Створено), продукт створюється.

Фактичний результат: Запит повернув HTTP статус 201 (Створено) (рис. В.6), продукт створився (рис. В.7).

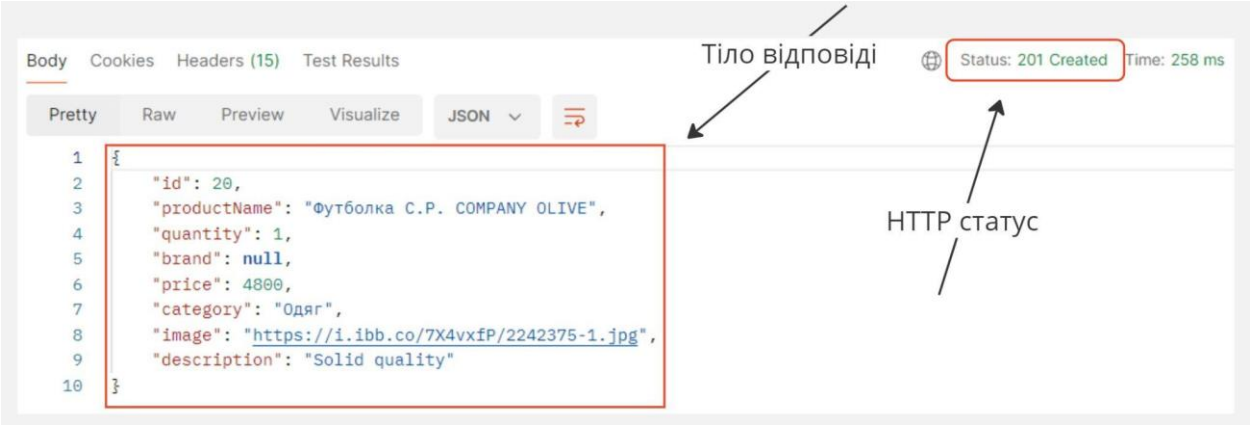

Рисунок В.6 – Результат виконання авторизованого запиту на створення продукту

Рисунок В.7 – Результат виконання запиту – створений продукт в магазині

Результат виконання тесту 3: Тест пройдено успішно

Проведення тестування відмовостійкості системи:

Тест 4

Ситуація: В запущеному додатку працюють всі сервіси(рис. В.8), кожен по одному екземпляру. Сервіс для оплати має два працюючих екземпляри в системі - екземпляр №1 та екземпляр №2. Генерується ситуація, при якій один з екземплярів виходить з ладу та перестає працювати.

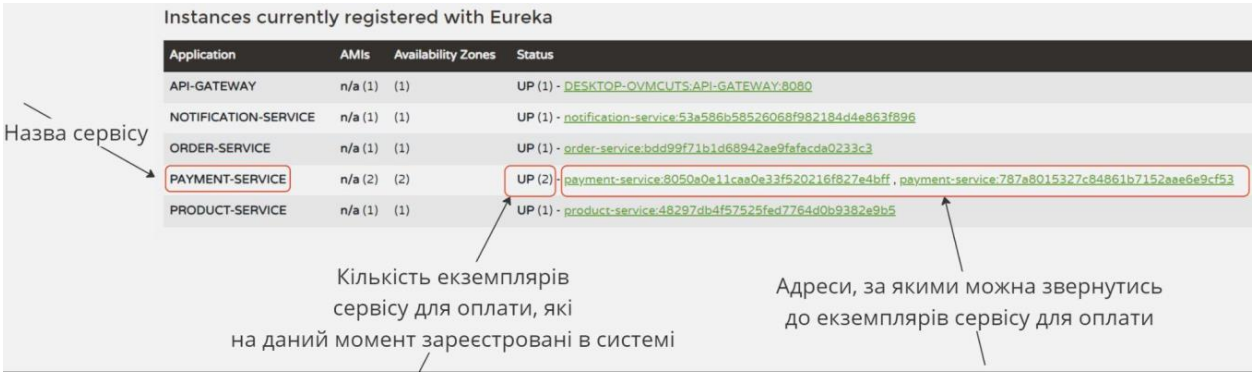

Рисунок В.8 – Реєстр сервісів в Discovery Server

Очікуваний результат: Після виходу з ладу одного екземпляру сервісу для оплати, система має справно працювати, оформлювати замовлення та приймати оплату надалі. Discovery Server, на запит для отримання адреси сервісу оплати, має повертати адресу другого екземпляру.

Фактичний результат: При коректній роботі двох екземплярів сервісів для оплати маємо успішний результат (рис. В.9)

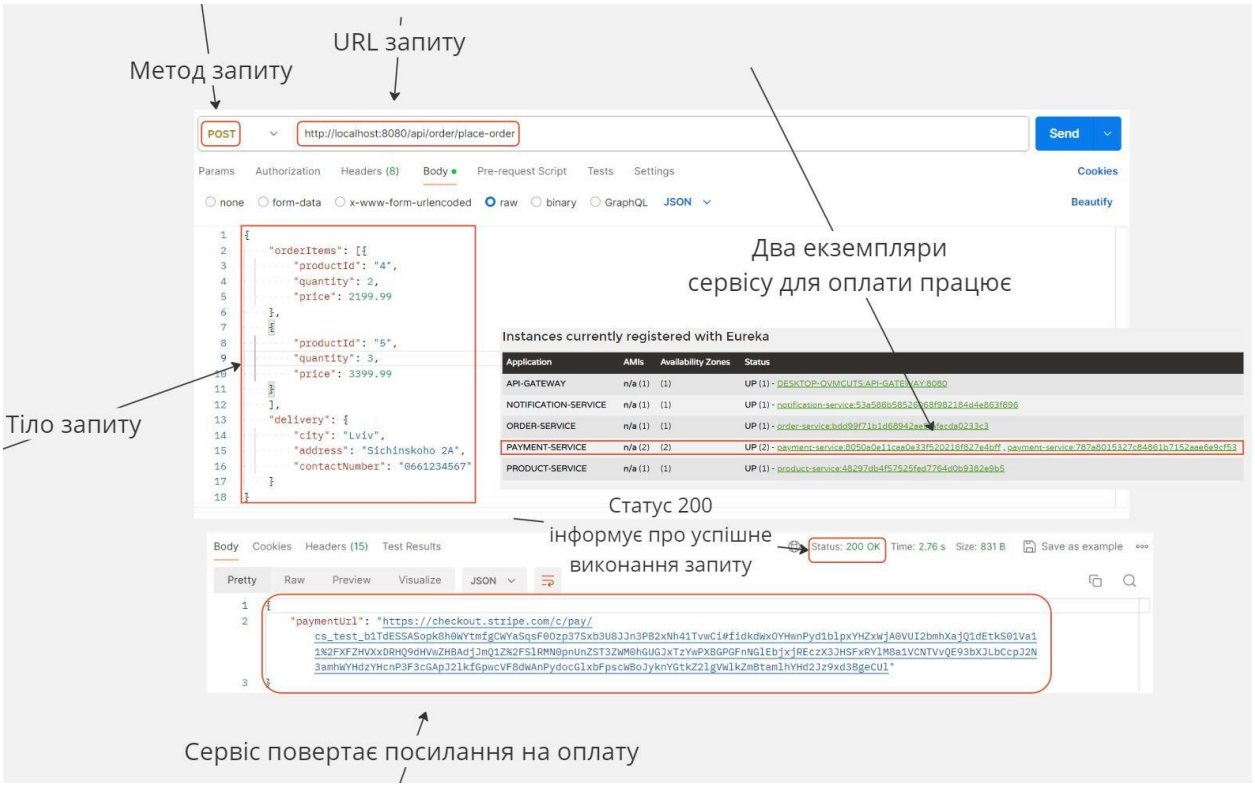

Рисунок В.9 – Система успішно відправляє запити на оформлення замовлення при коректній роботі двох сервісів оплати

Згенерувавши ситуацію виходу з ладу одного з екземплярів сервісу для оплати (вимикаємо сервіс), маємо успішний результат, система продовжує коректно обробляти запити на оплату (рис. В.10).

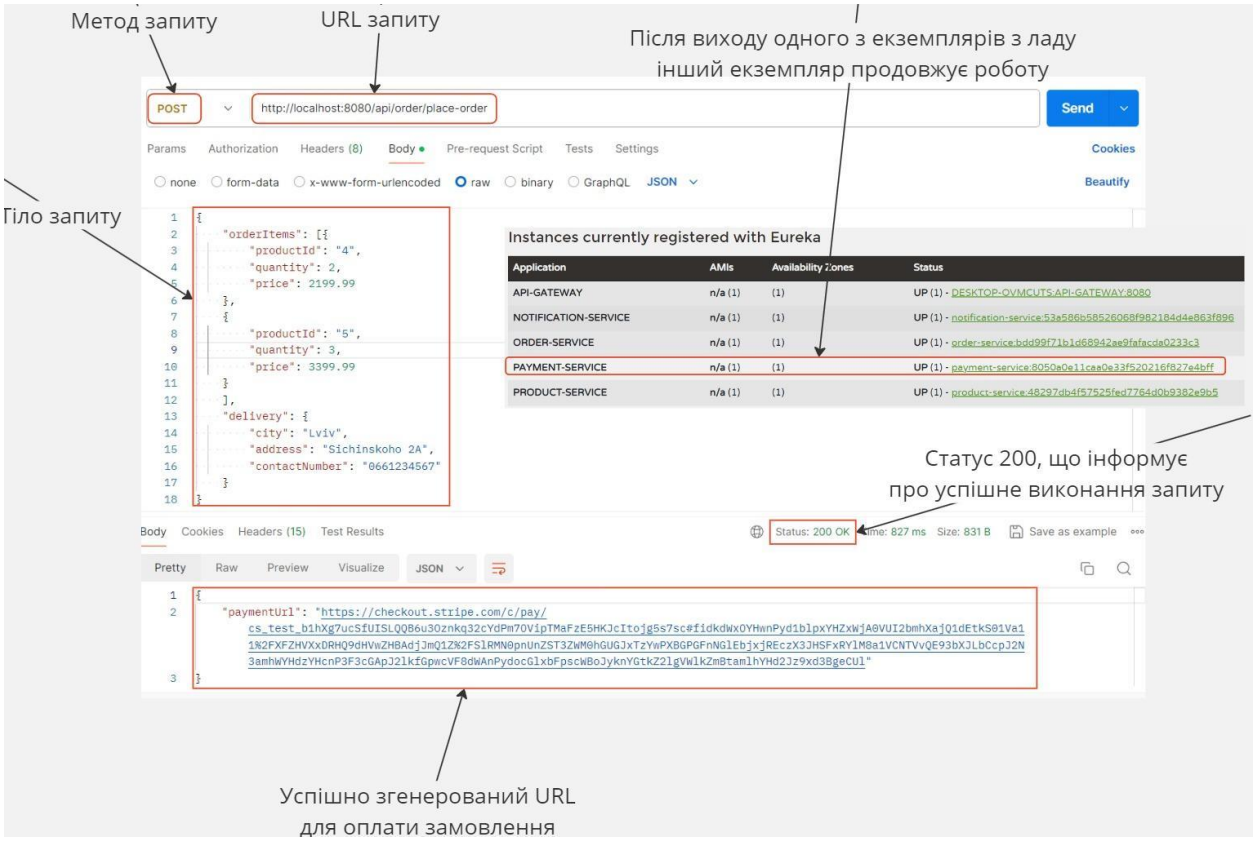

Рисунок В.10 – Система продовжує коректно працювати при виході з ладу одного з екземплярів

Результат виконання тесту 4: Тест пройдено успішно.

Висновки: Тест 1, тест 2, тест 3 та тест 4 пройдено успішно. Випробування пройдено успішно.

Виконавець: студент групи КІ-41, Дейкун Д.Ю.

Додаток Г

ФРАГМЕНТ ЛІСТИНГУ
програмного виробу «Модель комерційного програмного додатку на основі мікросервісної архітектури**»**

Лістинг конфігурації Discovery Server:

```
spring.main.web-application-type=reactive
spring.application.name=API-GATEWAY
spring.security.oauth2.resourceserver.jwt.issuer-
uri=http://localhost:8181/realms/hygge-realm
spring.cloud.gateway.globalcors.corsConfigurations.[/**].allowedOrigins=http://lo
calhost:4200
spring.cloud.gateway.globalcors.corsConfigurations.[/**].allowedMethods=*
spring.cloud.gateway.globalcors.corsConfigurations.[/**].allowedHeaders=*
spring.cloud.gateway.routes[0].id=product-service
spring.cloud.gateway.routes[0].uri=lb://product-service
spring.cloud.gateway.routes[0].predicates[0]=Path=/api/products/**
spring.cloud.gateway.routes[1].id=payment-service
spring.cloud.gateway.routes[1].uri=lb://payment-service
spring.cloud.gateway.routes[1].predicates[0]=Path=/api/payment/**
spring.cloud.gateway.routes[2].id=order-service
spring.cloud.gateway.routes[2].uri=lb://order-service
spring.cloud.gateway.routes[2].predicates[0]=Path=/api/order/**
server.port=8080
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
eureka.client.service-
url.defaultZone=${EUREKA_SERVER_ADDRESS:http://localhost:8761/eureka}
```

Лістинг Order Controller компоненту:

```
@RestController
@RequestMapping("/api/order")
@RequiredArgsConstructor
@Slf4j
public class OrderController {
  private final OrderService orderService;

  @PostMapping("/place-order")
  public PaymentResponse placeOrder(@RequestBody OrderRequest
orderRequest) {
     return orderService.placeOrder(orderRequest);
  }
```

```
}
```

Лістинг Product Service компоненту:

```
@Service
@Slf4j
@RequiredArgsConstructor
public class ProductServiceImpl implements ProductService {

  private final ProductRepository productRepository;
  private final CategoryService categoryService;
  private final ProductMapper productMapper;

  @Override
  public ProductResponseDto create(ProductRequestDto productRequest) {
    Product product = Product.builder()
        .productName(productRequest.getProductName())
        .price(productRequest.getPrice())
        .quantity(productRequest.getQuantity())
        .brand(productRequest.getBrand())
        .imageLink(productRequest.getImage())
        .category(categoryService.getCategoryByName(productRequest.ge
tCategory()))
        .description(productRequest.getDescription())
        .build();
    return
productMapper.toResponseDto(productRepository.save(product));
  }

  @Override
  public List<ProductResponseDto> getAllProducts() {
    return productRepository.findAll().stream()
        .map(productMapper::toResponseDto)
        .toList();
  }

  @Override
  public ProductNameResponseDto getProductNameById(Long id) {
    return new
ProductNameResponseDto(productRepository.findById(id).get().getProductName(
));
  }

  @Override
  public Product getProductById(Long productId) {
    return productRepository
```

```
                .findById(productId)
                .orElseThrow(() -> new EntityNotFoundException("Can't find
product with id " + productId));
        }

        @Override
        public List<ProductResponseDto> getProductsByCategory(String
categoryName) {
            return productRepository
                .getAllByCategory(categoryService.getCategoryByName(category
Name))
                .stream()
                .map(productMapper::toResponseDto)
                .toList();
        }

        @Override
        public void update(Long productId, ProductRequestDto productRequest) {
            Product product = productRepository
                .findById(productId)
                .orElseThrow(() -> new EntityNotFoundException("Can't find
product with id " + productId));
            product.setPrice(productRequest.getPrice());
            product.setProductName(productRequest.getProductName());
            product.setDescription(productRequest.getDescription());
            product.setBrand(productRequest.getBrand());
            productRepository.save(product);
        }

        @Override
        public void deleteById(Long productId) {
            productRepository.deleteById(productId);
        }
    }
```