МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ
ТА БІОТЕХНОЛОГІЙ ІМ. С.З. ГЖИЦЬКОГО

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

Другого (магістерського) рівня вищої освіти

на тему: “ Розробка мікросервісної архітектури для веб-орієнтованої інформаційної системи ”

Виконав: студент 6 курсу групи ІТ-61
Спеціальності 126 – „Інформаційні системи та технології”
(шифр і назва)

Шевців Михайло Степанович
(Прізвище та ініціали)

Керівник: к.т.н., доц. Падюка Р.І.
(Прізвище та ініціали)

Рецензенти: к.т.н., доц. Кригуль Р.Є.
(Прізвище та ініціали)

(Прізвище та ініціали)

ДУБЛЯНИ-2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМ. С.З. ГЖИЦЬКОГО

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Другий (магістерський) рівень вищої освіти
Спеціальність 126 "Інформаційні системи та технології"

“ЗАТВЕРДЖУЮ”
Завідувач кафедри_____
д.т.н., проф. А.М. Тригуба
“____”_____2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу студенту
Шевціву Михайлу Степановичу

1. Тема роботи: «Розробка мікросервісної архітектури для веб-орієнтованої інформаційної системи»

Керівник роботи Падюка Роман Іванович, к.т.н., доцент.

Затверджені наказом по університету від 28 лютого 2025 року № 140/к-с.

2. Строк подання студентом роботи 05.12.2025 р.

3. Початкові дані до роботи: 1. Опис предметної області веб-орієнтованої інформаційної системи; 2) Аналітичні матеріали щодо недоліків монолітних архітектур; 3) Методологічна база для проєктування мікросервісів; 4) Методи дослідження ефективності архітектурних підходів.

4. Зміст розрахунково-пояснювальної записки:

1. Аналіз стану питання в практиці та теорії
2. Обґрунтування, вибір та реалізація інструментарію вирішення задачі
3. Результати вирішення задачі
4. Охорона праці та безпека в надзвичайних ситуаціях
5. Визначення ефективності розроблення та впровадження інформаційної системи

Висновки та пропозиції.
Бібліографічний список.
Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень): Тема, автор, керівник магістерської роботи; Мета,завдання, об’єкт, предмет дослідження; Передумови та актуальність; Основні передумови переходу до мікросервісів; Технологічний стек розробки; Реєстрація та балансування сервісів; Забезпечення відмовостійкості; Загальна концепція дослідження; Загальна архітектура системи; Моделі даних та організація сховища; Контейнеризація та розгортання; Результати тестування та порівняння архітектур.

6. Консультанти з розділів:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1, 2, 3, 5	*Падюка Р.І., доцент кафедри інформаційних технологій*		
4	*Городецький І.М., доцент кафедри інженерної механіки*		

7. Дата видачі завдання 03 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Приміт-ка
1	*Написання першого розділу та означення головних завдань роботи*	03.03.-01.04.25	
2	*Виконання другого розділу та формування головних показників для розрахунків*	02.04.-01.05.25	
3.	*Виконання третього розділу та формування початкових даних*	02.05.-01.06.25	
4.	*Виконання четвертого розділу та узагальнення отриманих результатів магістерської роботи*	02.06.-01.09.25	
5.	*Вартісне оцінення ефективності пропозицій роботи*	02.09.-01.10.25	
6.	*Завершення оформлення розрахунково-пояснювальної записки та аркушів графічної частини*	02.10.-01.11.25	
7.	*Завершення роботи в цілому*	02.11.-05.12.25	

Студент _____ Шевців М.С.
(підпис)

Керівник роботи _____ Падюка Р.І.
(підпис)

УДК 004.75:004.42

Розробка мікросервісної архітектури для веб-орієнтованої інформаційної системи – Шевців М.С. Магістерська робота. Кафедра ІТ. – Дубляни, Львівський НУВМБ ім. С.З. Гжицького, 2025.

89 с. текст. част., 50 рис., 4 табл., 14 арк. графічної частини, 26 літ. джерел

У кваліфікаційній роботі проведено дослідження особливостей проєктування та реалізації веб-орієнтованої інформаційної системи на основі мікросервісної архітектури. Розглянуто теоретичні засади побудови розподілених систем, виконано порівняння мікросервісного та монолітного підходів, обґрунтовано вибір технологічного стеку та інструментів. Реалізовано два серверні варіанти системи, розроблено клієнтську частину та інфраструктуру розгортання в контейнерному середовищі. Проведено експериментальне тестування продуктивності, що дозволило визначити переваги й обмеження обох архітектур. Отримані результати можуть бути використані в проєктуванні масштабованих корпоративних застосунків.

Ключові слова: мікросервісна архітектура; моноліт; веб-орієнтована система; Spring Framework; контейнеризація; Kubernetes; тестування продуктивності.

Keywords: microservice architecture, monolithic architecture, web-oriented information systems, distributed systems, scalability, performance testing, Spring Framework.

ЗМІСТ

ВСТУП

Стрімкий розвиток цифрових технологій та зростання складності сучасних інформаційних систем зумовлюють необхідність використання нових архітектурних підходів до їх проєктування та побудови. Традиційні монолітні моделі дедалі рідше відповідають вимогам щодо масштабованості, гнучкості, відмовостійкості та можливості швидкого розширення функціональності. У цьому контексті мікросервісна архітектура виступає одним із ключових напрямів розвитку програмної інженерії, адже забезпечує незалежність компонентів, автономність їхнього розгортання, підтримку різних технологічних стеків та підвищену стійкість до збоїв у системі.

Актуальність обраної теми визначається потребою побудови високопродуктивних, масштабованих та безпечних веб-орієнтованих інформаційних систем, здатних ефективно функціонувати у розподіленому середовищі та обробляти значну кількість паралельних запитів. Подібні системи широко використовуються у сферах електронної комерції, телекомунікації, охорони здоров’я, державного управління, промислових цифрових сервісах та корпоративних інформаційних середовищах. Зростання вимог до надійності, безпеки та швидкої модифікації функціональних можливостей підтверджує необхідність глибокого дослідження можливостей мікросервісного підходу.

Об’єктом дослідження у роботі є процес розроблення та функціонування веборієнтованих інформаційних систем у розподіленому програмному середовищі.

Предметом дослідження є методи, алгоритми, архітектурні рішення та інструменти побудови мікросервісної архітектури, що забезпечують її масштабованість, безпеку, надійність та узгодженість роботи сервісів.

Метою кваліфікаційної роботи є проєктування та реалізація веб-орієнтованої інформаційної системи на основі мікросервісної архітектури, що забезпечує підвищену масштабованість, відмовостійкість, безпечність обробки даних та можливість швидкого розширення функціональних можливостей.

Результати роботи можуть бути застосовані в галузях, що потребують розподілених високонавантажених застосунків: у сфері електронного бізнесу, корпоративних інформаційних системах, хмарних сервісах, цифрових сервісах органів влади та аналітичних платформах.

Для досягнення поставленої мети необхідно вирішити такі **завдання:**

- проаналізувати наявні архітектурні стилі та технологічні підходи до побудови веб-орієнтованих систем та визначити їхні переваги й недоліки;
- обґрунтувати вибір мікросервісної архітектури як основи для розроблюваної системи;
- визначити вимоги до функціональної та нефункціональної частин системи;
- розробити структурну схему мікросервісної архітектури та описати взаємодію її компонентів;
- обрати технологічний стек та засоби розгортання сервісів;
- реалізувати окремі сервіси та інтегрувати їх у єдине інформаційне середовище;
- провести тестування та оцінити ефективність запропонованої архітектури порівняно з традиційними підходами;
- сформувати рекомендації щодо подальшого розвитку та масштабування системи.

РОЗДІЛ 1

АНАЛІЗ СТАСТАНУ ПИТАННЯ В ТЕОРІЇ ТА ПРАКТИЦІ

1.1 Основні концепції мікросервісної архітектури

Під час розробки будь-якого мікросервісного застосунку важливо враховувати дві фундаментальні концепції: зв'язуваність (coupling) та згуртованість (cohesion). Хоча ці концепції можуть бути не найважливішими елементами при побудові мікросервісів, а також не є виключними для цієї області, будь-яка методологія, що використовується при створенні мікросервісного застосунку, фундаментально базується на простому принципі: «Слабке зв'язування і висока згуртованість».

У контексті архітектури мікросервісів, зв'язок стосується значення або метрики, яка вказує на ступінь зв'язку або залежності між двома програмними модулями. Більший ступінь зв'язку між цими модулями призводить до збільшення складності їх незалежного використання.

Зрозуміло, що в нашому прикладі одиничні модулі є мікросервісами. Чим менший зв'язок між мікросервісами, тим краще, оскільки ключовими характеристиками мікросервісів є:

- Незалежне розгортання та взаємозамінність з будь-яким іншим мікросервісом, що виконує аналогічну функціональність.
- Відсутність необхідності модифікувати всю систему при заміні компонента заданого типу компонентом іншого типу.

Основною помилкою, що призводить до високого зв'язку, є невідповідний вибір методів інтеграції сервісів (наприклад, вибір неправильного протоколу або стандарту обміну даними, відсутність/відмова від використання спільних типів даних).

Це призводить до необхідності відповідних змін в інших сервісах, коли змінюється поведінка деяких сервісів.

Слабко зв'язана система знає лише мінімальні дані про сервіси навколо неї (в першу чергу інформацію про реєстрацію, балансування навантаження або сервіси баз даних), а також обмежений набір запитів та набір одержувачів цих запитів. Очевидно, що зі зменшенням кількості запитів до одного модуля/сервісу в системі зменшується також і зв'язок. У програмуванні було визначено кілька типів зв'язку на основі їх причин та ступеня:

- Зв'язування контенту
- Загальне зв'язування
- Зовнішнє зв'язування
- Зв'язування керування
- Зв'язування даних
- Зв'язування повідомлень
- Тимчасове зв'язування
- Зв'язування підкласів
- Динамічне зв'язування
- Семантичне зв'язування
- Логічне зв'язування

Однак, під час створення мікросервісних застосунків ми в першу чергу зосереджуємося на таких типах зв'язку: зв'язок реалізації (як описано вище), часовий зв'язок (однаково важливий), зв'язок розгортання та зв'язок домену.

Зв'язок реалізацій часто описується як залежність двох або більше мікросервісів один від одного настільки, що коли мікросервіс А змінюється, мікросервіс В, який надмірно залежить від А, також повинен відповідно змінитися, навіть якщо сам В не повинен базуватися на А. В архітектурах мікросервісів ця залежність найчастіше виникає через неправильне спільне використання бази даних. Як правило, рішенням є створення високоякісних, надійних проміжних інтерфейсів для мікросервісів, щоб вони могли взаємодіяти один з одним та з базою даних. [2, 3]

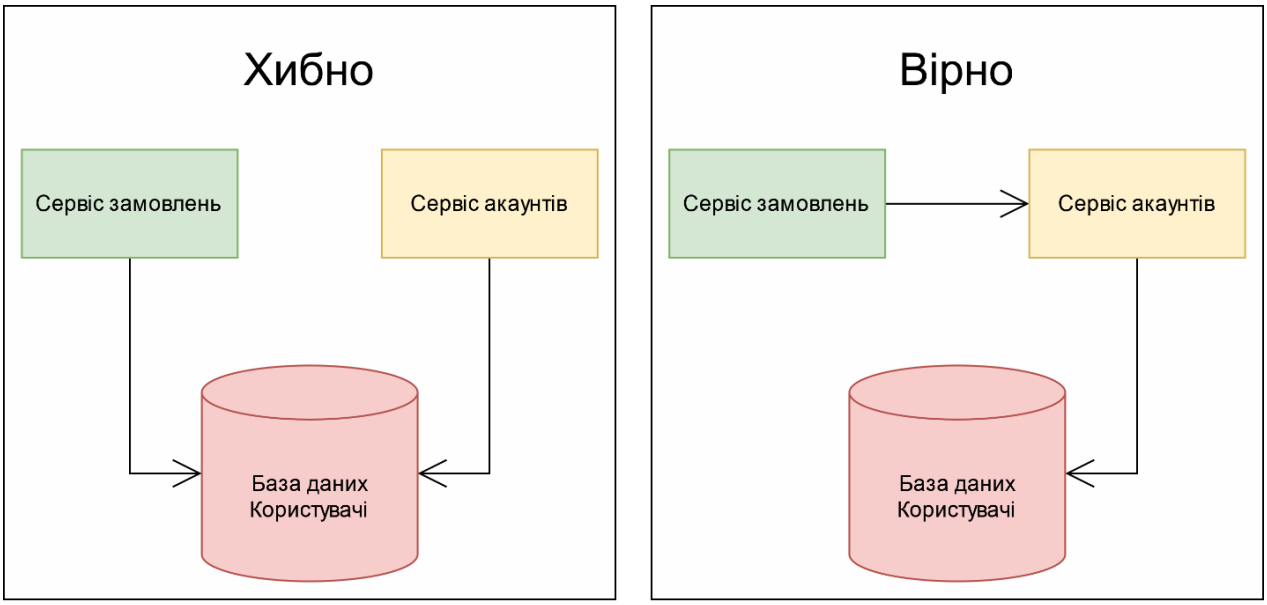


Рисунок 1.1. a (ліворуч) – ілюструє випадок неправильного спільного використання бази даних. b (праворуч) – демонструє відповідний метод взаємодії в цьому сценарії.

Як показано на рисунку 1.1, служба облікових записів має повний доступ до бази даних і служить надійним інтерфейсом до служби замовлень. Служба облікових записів може самостійно обробляти всю необхідну інформацію та надає службі замовлень лише ті дані, які необхідні для її роботи, що дозволяє як зменшувати навантаження на канал передачі даних, так і підвищити безпеку даних користувачів.

Найголовніше, це зменшує зв'язок між базою даних та службою замовлень. Служба замовлень більше не залежить від конкретної реалізації бази даних, а може надсилати загальні запити через чітко визначений та контрольований інтерфейс у службі облікових записів, яка вже знає, як маніпулювати базою даних. Таким чином, ми зменшуємо кількість компонентів, що залежать від конкретної реалізації, з двох до одного, оскільки всі залежності тепер знаходяться в службі облікових записів і можуть бути легко переписані, замінені або доповнені до нових типів баз даних.

Ідея про те, що проблеми з підключенням – це лише архітектурна проблема, яку можна вирішити на папері та в діаграмах, і що проблем з підключенням можна уникнути на етапі «композиції» програми, ймовірно, є

досить хибною. На жаль, тимчасове підключення є найбільшим ворогом будь-якої мікросервісної програми, оскільки воно виникає під час так званого «виконання», тобто під час роботи програми.

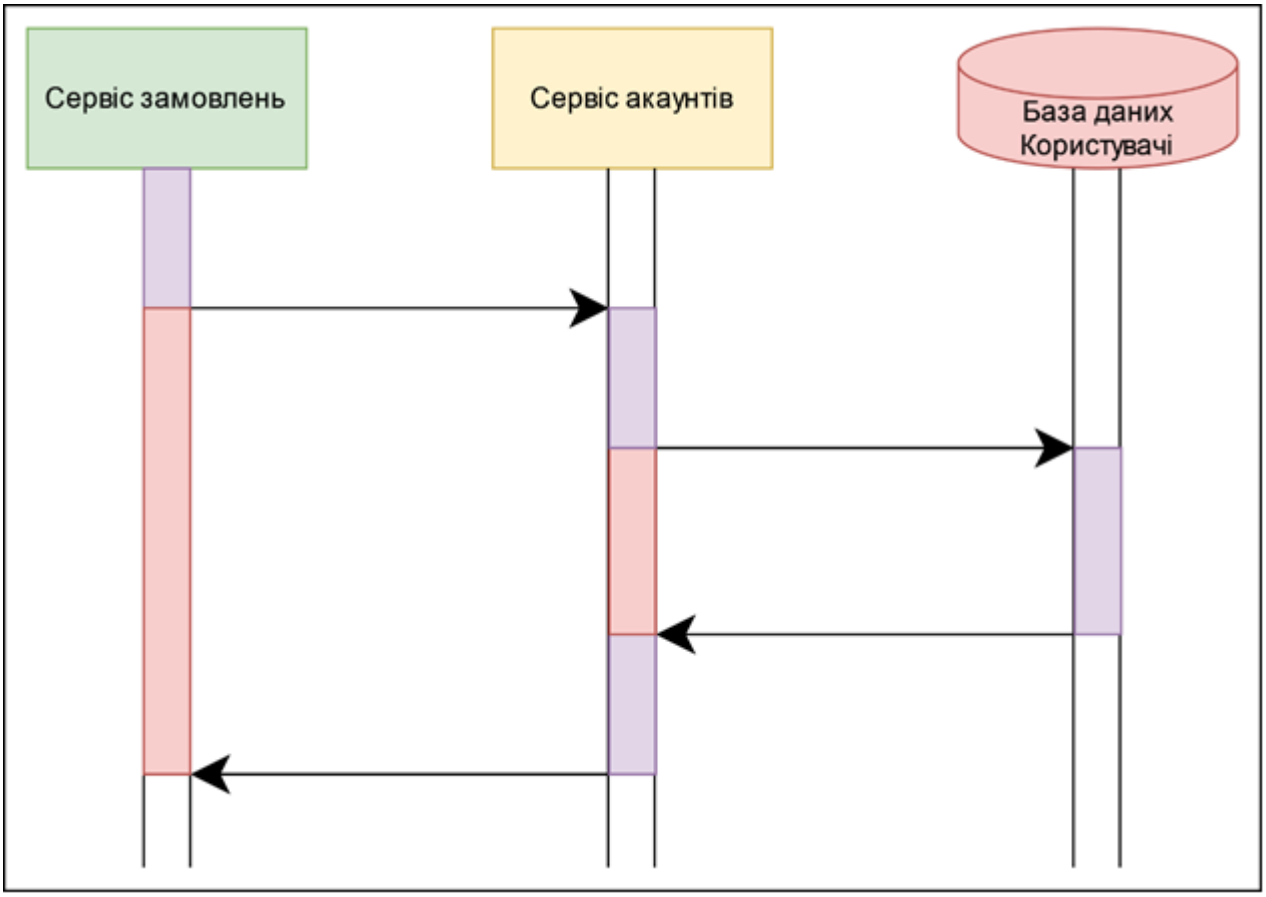


Рисунок 1.2. Ілюстрація темпоральної залежності в мікросервісній програмі.

У мікросервісних програмах це підключення проявляється як залежності, які вимагають синхронних викликів до різних служб. Найпоширеніший сценарій – це коли є мало реплік, але велика кількість викликів, або коли служби утворюють складний ланцюжок послідовних викликів, що вимагає очікування результату однієї служби, перш ніж інша зможе продовжити роботу. Це підключення значно знижує масштабованість та можливості паралелізації архітектури мікросервісу. Для вирішення цієї проблеми підключення зазвичай використовуються два компоненти: кешування даних та асинхронні мікросервіси. [3] Як показано на рисунку 1.2, темпоральні залежності неминучі, оскільки інтерфейс використовується як частина служби облікових записів.

Крім того, на рисунку 1.2 показано, що мікросервіси, відповідальні за фактичну обробку даних, найбільше зазнають впливу цього типу залежності, і вплив цього підключення поступово зменшується, коли обробка запитів наближається до завершення. (Час простою позначено червоним кольором на рисунку. Зверніть увагу, що затримка передачі даних опущена.)

Обидва спостереження підтверджують аргумент, що ця проблема підключення є лише тимчасовою проблемою під час роботи системи та повинна вирішуватися на технічному рівні. Як згадувалося раніше, існує два підходи до цієї проблеми:

Кешування – цей підхід зменшує підключення, використовуючи різні рівні кешування (подібно до кешів L1, L2 та L3 у процесорі), тим самим уникаючи запитів до інших служб. Подібно до ЦП, чим ближче кеш до служби, тим швидша швидкість доступу, але тим швидше кеш закінчується. Аналогічно ЦП, найшвидший кеш для служби замовлень розташований у самій службі замовлень, повільніший кеш знаходиться в службі облікових записів, а найповільніший кеш – це прямий доступ до бази даних. Важливо зазначити, що використання кешування за своєю суттю є ймовірнісним (дані можуть бути відсутні в кеші або термін їх дії закінчився, що може тимчасово збільшити час виконання запиту) та є менш надійним, коли дані бази даних часто оновлюються. Однак цей метод дуже простий у використанні та реалізації, і досить універсальний для завдань на стороні сервера, що запитують дані. Він особливо популярний у системах CDN.

Асинхронна обробка – це другий підхід до обробки затримки. На відміну від кешування, асинхронна обробка безпосередньо не бореться з тривалістю запиту, натомість вона намагається дозволити сервісу виконувати інші завдання під час обробки запиту. Іншими словами, сервіс не простоює та не витрачає час на очікування даних. Цей підхід до обробки узгодженості часу складніший у використанні та ставить вищі вимоги до завдань на стороні сервера, оскільки деякі сервіси можуть сильно залежати від даних і не мати змоги виконувати інші завдання без них. Однак цей підхід досить стабільний у виконанні; час обробки

запитів не зазнає неочікуваних стрибків і не залежить від частоти оновлень бази даних.

Узгодженість розгортання. Це найменш очевидний і найбільш технічно залежний тип узгодженості. По суті, ця узгодженість описує ситуації, коли програмні реалізації мікросервісів залежать одна від одної; коли випускається бінарний пакет для мікросервісу А, бінарні пакети для всіх інших залежних мікросервісів повинні бути надані одночасно.

Ця узгодженість може здаватися такою, що перетинається з узгодженістю реалізації. Однак насправді це не так, оскільки контекст розгортання підкреслює, що програмна реалізація залежить не лише від ступеня залежності від зовнішніх інтерфейсів, але й від рівня бінарних пакетів або процесів розгортання, що призведе до створення залежностей. Однак, на рівні впровадження цього бракує.

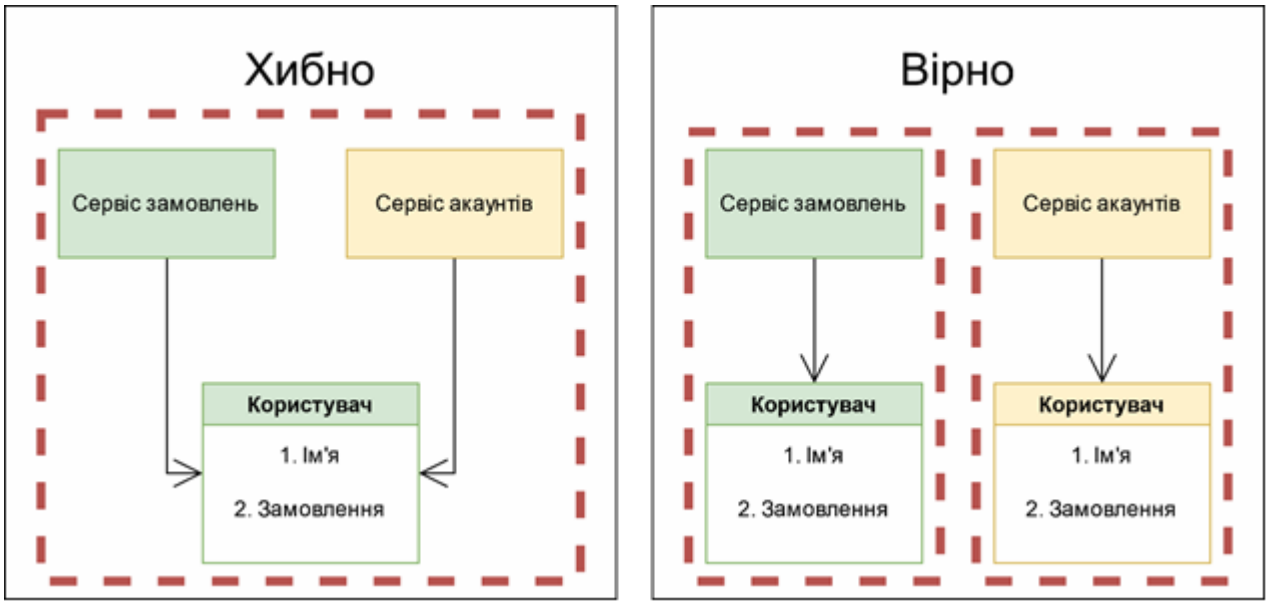


Рисунок 1.3. a (ліворуч) – ілюстрація структурної залежності. b (праворуч) – відповідний метод деконструкції залежності.

Помилки в процесі можуть створювати залежності між сервісами, які насправді не є взаємозалежними. Крім того, високий рівень зв'язку розгортання вказує на те, що мікросервіси охоплюють занадто багато обов'язків, що призводить до численних залежностей, яких можна уникнути, розбивши модулі на менші компоненти.

На рисунку 1.3 наведено приклад зв'язку розгортання. Як показано на рисунку 1.3a, два сервіси працюють за сценарієм на рисунку 1.1: сервіс замовлень очікує структуру користувача на рисунку 1.3 на рівні реалізації, тоді як сервіс облікових записів повертає цю структуру як відповідь на запит.

Ідея створення пов'язаних регіонів у вигляді структури "користувач" сама по собі не є неправильною і є необхідною умовою для формування інтерфейсів (обговорюється в наступному абзаці). Однак це не означає, що мікросервіси повинні бути пов'язані на рівні бінарного пакета, оскільки в прикладі на рисунку 3a будь-яка зміна сервісу замовлень або структури "користувач" призведе до зміни сервісу облікових записів. Отже, на рисунку 1.3b показано простий спосіб подолати цю залежність — створення структур користувачів окремо в реалізаціях служб замовлень та облікових записів, щоб зменшити кількість спільних елементів. Червоні пунктирні лінії на рисунку 1.3b представляють межі бінарних пакетів.

Тепер ці два сервіси можна змінювати незалежно один від одного. Єдине, що потрібно забезпечити, це те, щоб структура "користувача" одного сервісу була зіставлена зі структурою "користувача" іншого сервісу. Це не складно і не спричинить таких проблем, як збільшення часу доставки оновлень або помилки під час непотрібного розгортання.

Ще один метод підключення розгортання це спільне використання тестових даних або середовища. Це схоже на проблему використання бінарних пакетів, але рішення простіше (копіювання тестових даних для кожного сервісу), хоча воно також несе більші ризики, оскільки ця проблема не тільки збільшує час розгортання, але й може вплинути на точність результатів тестування для кожного сервісу.

Узгодженість тем є ключовим питанням у всіх методологіях створення додатків на основі мікросервісів. Чітко визначений та детальний домен теми забезпечує низьку загальну узгодженість системи та чітко розрізняє межі кожного мікросервісу та теми (об'єкти) – типи та структури даних, з якими

працюють мікросервіси та які доступні в додатку, а також області узгодженості – точки взаємодії між мікросервісами.

Ці три пункти допомагають нам окреслити та чітко визначити обов'язки кожного мікросервісу, тим самим уникаючи проблем узгодженості розгортання; водночас вони також визначають типи даних, з якими працюють сервіси, долаючи проблеми узгодженості реалізації та темпоральні узгодженості. У поєднанні з чітко визначеними областями узгодженості ми можемо визначити суворо надійні інтерфейси, тим самим забезпечуючи стабільність системи та відмовостійкість, а також уникаючи зв'язку реалізації та розгортання.

Зв'язок в архітектурі мікросервісів. Цей термін (або метрика) описує ступінь організації всього залежного коду. Високий зв'язок означає, що весь залежний код знаходиться в одному модулі, тоді як низький зв'язок означає, що цей код розподілений по всьому проекту. Строго кажучи, дана метрика є протилежністю низькому зв'язку, оскільки високий зв'язок коду природно призводить до низького зв'язку мікросервісів.

У сфері мікросервісів низький зв'язок описує, скільки сервісів або модулів розробки нам потрібно оновити перед розгортанням оновленої версії системи або програми.

1.2 Аналіз монолітної та мікросервісної архітектур

Загалом, у даній кваліфікаційній роботі ми розглянемо два основні способи класифікації архітектури системи:

- Монолітна архітектура – ця архітектура будує систему як єдине ціле.
- Це класична та досить поширена архітектура в галузі програмного забезпечення.
- Мікросервісна архітектура, яка є основою дослідження в цій кваліфікаційній роботі. Вона, на відміну від монолітної, є дуже гнучкою та швидко розвивається.

Перш ніж далі описувати та аналізувати ці архітектури, слід зазначити, що хоча мікросервісна архітектура загалом вважається кращою, все ж є деякі моменти, які слід враховувати, а саме слід врахувати, що кожна архітектура має свої переваги та недоліки, які будуть обговорені у відповідних розділах даної роботи.

Крім того, слід зазначити, що певні області розробки ПЗ абсолютно не підтримують використання певної архітектури або те що певна архітектура має очевидні недоліки при розробці в даній області.

Нарешті, слід додати, що хоча мікросервісна архітектура є дуже гнучкою, але, якщо її неправильно реалізувати, її результати можуть бути значно гіршими за результати монолітної архітектури.

1.2.1 Аналіз особливостей монолітної архітектури.

Як згадувалося вище, концепція монолітної архітектури є вирішальною при розробці мікросервісних додатків. Не вдаючись у її визначення та опис, можна сказати, що з точки зору архітектора мікросервісних додатків, монолітна архітектура має такі характеристики:

- Можливий небажаний результат, оскільки будь-яка мікросервісна архітектура може неявно виродитися в монолітну архітектуру. Це особливо важливо, коли система проектується з самого початку.
- Є відправною точкою для побудови мікросервісних додатків.

Точніше, ця ситуація проявляється двома способами: один – коли архітектура змінюється з монолітної на мікросервісну архітектуру, а інший – коли початковий дизайн архітектури є монолітною архітектурою. Останнє особливо цікаво, оскільки в цьому випадку з технічної точки зору іноді набагато легше визначити критичні точки з'єднання. [1]

Монолітна архітектура вже давно визнана стандартом для розробки додатків, особливо розробки desktop додатків. Монолітний додаток проектується та реалізується як єдина та неподільна одиниця виконання.

Як правило, структуру монолітного рішення можна проілюструвати так, як на рисунку 1.4.

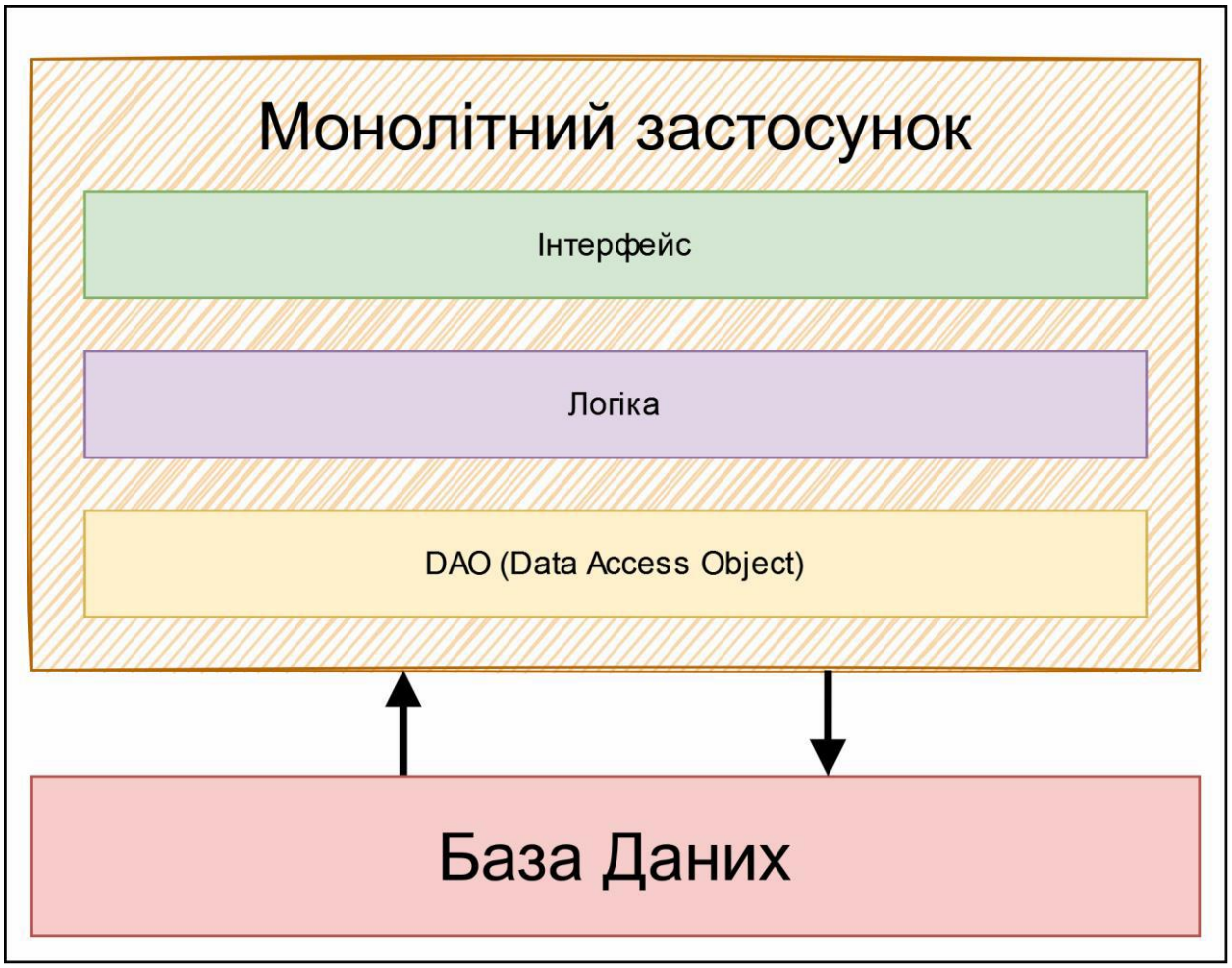


Рисунок 1.4. Схематична ілюстрація будови монолітного застосунку.

На рисунку 1.4 показано такі важливі елементи використання монолітної архітектури, як:

- Інтерфейс користувача на стороні клієнта.
- Програма на цільовій платформі (якою може бути ПК користувача або віддалений сервер).
- База даних.

Зазвичай монолітна архітектура складається з одного або кількох великих проектів, які розділені (але не обов'язково) на кілька модулів. Фактично, розмір кодової бази або наявність модулів не є основними характеристиками монолітної архітектури. Її головна характеристика полягає в тому, що для надання нових функцій користувачам зміна будь-якої частини кодової бази вимагає повної заміни бінарного пакета (тобто повного розгортання всієї програми). Це обмеження часто можна обійти за допомогою різних технічних хитрощів (таких як спільні об'єкти в Linux або DLL у Windows), але це не змінює фундаментальної природи монолітної архітектури.

Переваги монолітної архітектури головним чином відображаються в деяких функціях, які загалом не є очевидними та легко пропускаються. Молоді техніки, які не мають досвіду налагодження, часто пропускають ці функції, серед яких варто виділити наступні:

Загальна простота. Це не означає, що архітектура монолітного додатку проста, а також не означає, що будь-яке програмне забезпечення в монолітній архітектурі може працювати безпосередньо. Однак, планування та початкові кроки розробки монолітного додатку набагато простіші.

Легше налагоджувати. Як правило, монолітні додатки мають менше технологій, що полегшує їх налагодження. Крім того, зв'язок між модулями в монолітному додатку є простим і не вимагає спеціальних навичок.

(Однак важливо пам'ятати, що налагодження будь-якого додатку ніколи не буває легким.)

Продуктивність та швидкість. Монолітні додатки в ідеалі зберігають свої модулі в одному адресному просторі пам'яті (або, в гіршому випадку, в інших процесах на комп'ютері), що покращує продуктивність таких додатків.

Легше розгортати. Хоча кожна зміна в монолітному додатку може вимагати повного розгортання, яке може зайняти багато часу, це все одно набагато простіше, ніж розгортання мікросервісного додатку.

Легше тестувати. Монолітні додатки та системи легше тестувати, ніж мікросервісні додатки.

Їхні слабкі сторони часто включають архітектурні проблеми, які особливо очевидні під час розробки певних типів програм і систем, серед них відзначимо:

Низьку відмовостійкість за замовчуванням. Монолітні програми повинні обробляти помилки; інакше необроблені або фізично необроблені помилки можуть призвести до збою програми. Ситуація посилюється, якщо програма має тривалий час запуску.

Масштабованість. Монолітні програми, як правило, важко масштабувати. Навіть якщо потрібна лише невелика частина функціональності, можна запустити лише нову копію всієї програми.

Легко зрозуміти. Монолітні програми мають дуже великі та складні для розуміння кодові бази. Хоча монолітна архітектура не обов'язково призводить до високої чи низької узгодженості, навіть за цих хороших показників обсяг коду, який потрібно зрозуміти, все одно може швидко зростати.

Повільна швидкість оновлення. Час розгортання та збірки монолітних програм все залишається досить довгим.

1.2.2 Аналіз особливостей мікросервісної архітектури.

Монолітний додаток – це окремий блок або кодова база, тоді як мікросервісна архітектура розбиває його на серію менших, більш незалежних та більш автономних виконавчих блоків.

В ідеалі ці блоки (мікросервіси) виконують чітко визначене завдання, ізолюючи процес виконання від інших модулів додатку. Іноді елементами мікросервісної архітектури є сервіси (особливо важливі в концепції операційної системи) або модулі.

Як згадувалося вище, всі сервіси інкапсулюють власну логіку, сервіси зберігання даних, а також мають чітко визначені інтерфейси та відповідні чітко визначені набори функціональності. В мікросервісній архітектурі додаток працює через взаємодію між цими незалежними модулями, які взаємодіють за допомогою протоколів обміну даними.

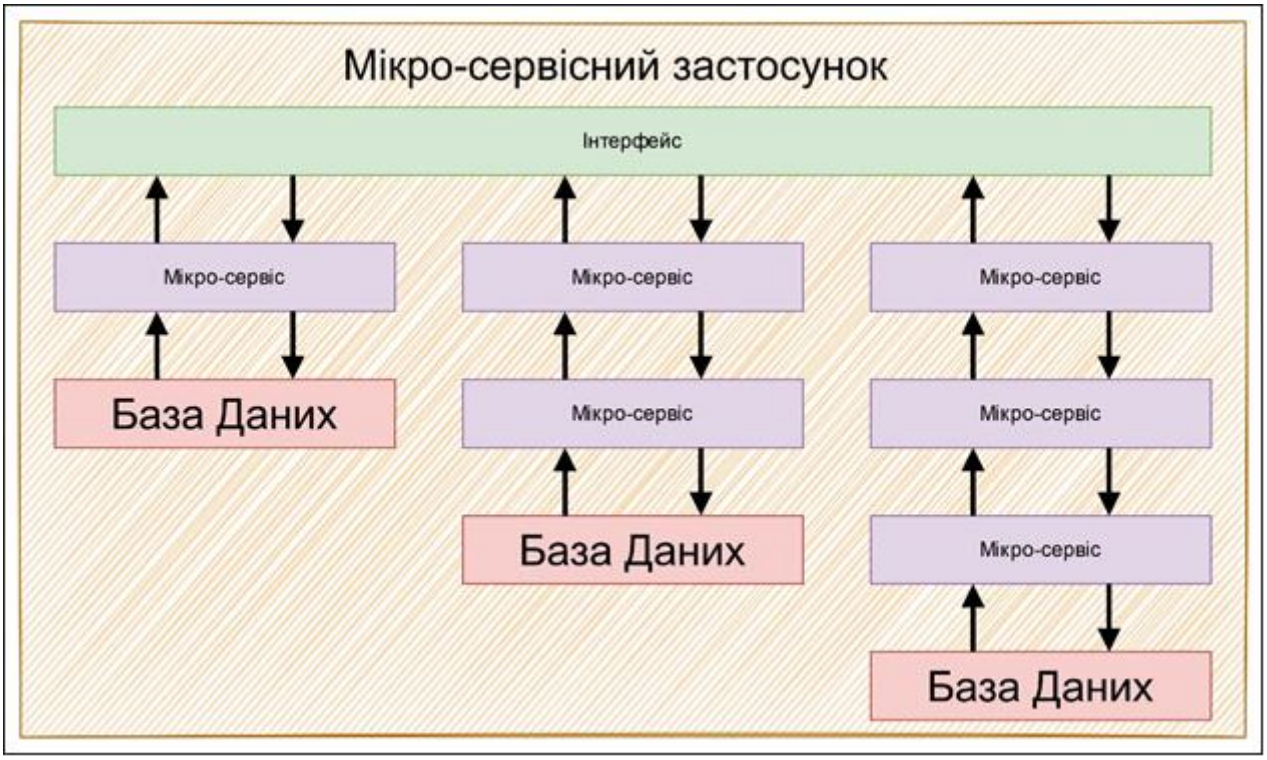


Рисунок 1.5. Схематична ілюстрація будови мікро-сервісного застосунку.

На рис. 1.5 показано форму зв'язків між елементами в архітектурі мікросервісів. Загалом, наступні принципи допомагають дотримуватися цієї форми в архітектурі (Томас Ерл перераховує 8 основних принципів у своїй книзі)[4]:

1. Стандартизація – усі мікросервіси в системі повинні відповідати певним стандартам. Наприклад, використовувати єдиний протокол зв'язку.

2. Низький зв'язок – сервіси повинні бути максимально незалежними.

3. Абстракція – сервіси повинні приховувати деталі своєї реалізації. Тобто, сервіси повинні ізолювати свою внутрішню поведінку від інших сервісів.

4. Можливість повторного використання – сервіси повинні бути розроблені таким чином, щоб їх було легко використовувати в інших проектах або інших елементах системи.

5. Автономія (незалежність) – сервіси повинні мати можливість виконувати свої функції без впливу стану середовища. Іншими словами, на

роботу сервісу впливає лише його внутрішній стан (поняття стану є досить абстрактним і стосується даних, що зберігаються у власній пам'яті) та вхідні дані (які також можна вважати внутрішнім станом).

6. Відсутність внутрішнього стану – це означає, що поведінка сервісу не залежить від результату попередніх запитів і є детермінованою, залишаючись незмінною між різними запитами.

7. Відкритість – сервіс повинен мати можливість надавати метадані про себе, щоб вся система могла знайти необхідну інформацію про цей сервіс.

8. Взаємодія – мікросервіси повинні бути побудовані таким чином, щоб вони могли успішно працювати з іншими мікросервісами, не змінюючи їхньої внутрішньої поведінки [4].

Дотримання всіх вищезазначених принципів дозволяє побудувати досить гнучку архітектуру. Ці принципи досить інтуїтивно зрозумілі та в основному базуються на практичному досвіді розробки програмного забезпечення.

Переваги мікросервісної архітектури зазвичай включають усі функції, що сприяють горизонтальному та вертикальному масштабуванню програм, а також функції, що допомагають створювати відмовостійкі програми.

Основною характеристикою мікросервісів є їхня масштабованість. Тому весь процес розробки та розгортання є більш економічно ефективним та менш трудомістким. Це особливо важливо порівняно з монолітними архітектурами, де навіть якщо нам потрібно лише додати кількість певних модулів, єдиний спосіб масштабувати програму – це запустити ще одну копію. Очевидно, що чим більше користувачів, тим більше проблем із масштабованістю зіткнеться з програмою. [5, 6]

Завдяки своєму слабкому зв'язку, мікросервісні програми природно мають здатність ізолювати помилки до конкретних екземплярів мікросервісів. Якщо модуль системи перестає працювати, його можна легко вимкнути, замінити або налагодити, не зупиняючи всю систему.

Кожен мікросервіс у системі може мати власний технологічний стек. Крім того, цей технологічний стек можна дуже легко оновлювати. Звичайно,

неможливо вибрати всі технології виключно на основі мікросервісів, але можливості все ж таки більші, ніж у монолітних додатків.

Кожен мікросервіс менший за повний монолітний додаток. Водночас, зв'язок коду в межах мікросервісу високий, тоді як кодова база набагато простіша через низький зв'язок усієї системи.

Розгортання застосовується лише до окремих елементів системи (мікросервісів), а не до всього додатку (системи).

Хоча мікросервісні додатки мають багато переваг, вони також мають свої недоліки. Деякі недоліки цієї архітектури відсутні в монолітних додатках, що знову ж таки підкреслює важливість правильного використання та впровадження для кожної з архітектур. [1, 4, 5]

До недоліків цієї архітектури слід віднести загальну складність системи, де кожен окремий мікросервіс може бути легким для розуміння, але якщо розглядати всю систему як єдине ціле, вся архітектура мікросервісу стає досить складною та важчою для розуміння, ніж монолітна архітектура.

Мікросервісна архітектура базується на ідеї незалежності мікросервісів. Основна складність полягає саме у взаємодії між цими мікросервісами та досить складних протоколах зв'язку.

Виходячи з вищезазначених проблем, можна зробити висновок, що налагодження всієї системи проходить набагато складніше, ніж налагодження монолітного застосунку.

1.3. Аналіз особливостей сервіс-орієнтованої архітектури (SOA)

Наступним рішенням архітектури програмного забезпечення, яке ми розглянемо в цьому дослідженні, є сервісно-орієнтована архітектура (SOA). SOA – це стиль архітектури програмного забезпечення, який розглядає систему як сукупність слабо пов'язаних компонентів, кожен з яких виконує певну функцію. Цей підхід поділяє компоненти системи на дві основні ролі: постачальники послуг та споживачі послуг. Обидві ролі можуть виконувати програмні

компоненти. Концепція SOA передбачає, що всі модулі системи легко інтегрувати та повторно використовувати. У свою чергу, компоненти повинні відповідати наступним вимогам [3]:

1. Єдиний стандарт для інтерфейсів компонентів.
2. Абстракція.
3. Слабкий зв'язок між компонентами.
4. Автономність кожного компонента.
5. Можливість використання компонентів в інших системах.

Кожен компонент (сервіс) – це незалежний функціональний модуль, який може фізично знаходитися на іншому комп'ютері, а взаємодія між цими модулями здійснюється через комп'ютерну мережу. У базовому представленні така система містить три основні елементи: постачальника послуг, реєстр послуг та споживача послуг. Алгоритм взаємодії між елементами такий: постачальник послуг вносить дані свого сервера до реєстру, а споживач надсилає запит на зареєстровану послугу до реєстру. Зв'язок здійснюється за допомогою одного з уніфікованих протоколів передачі даних (SOAP, XML).

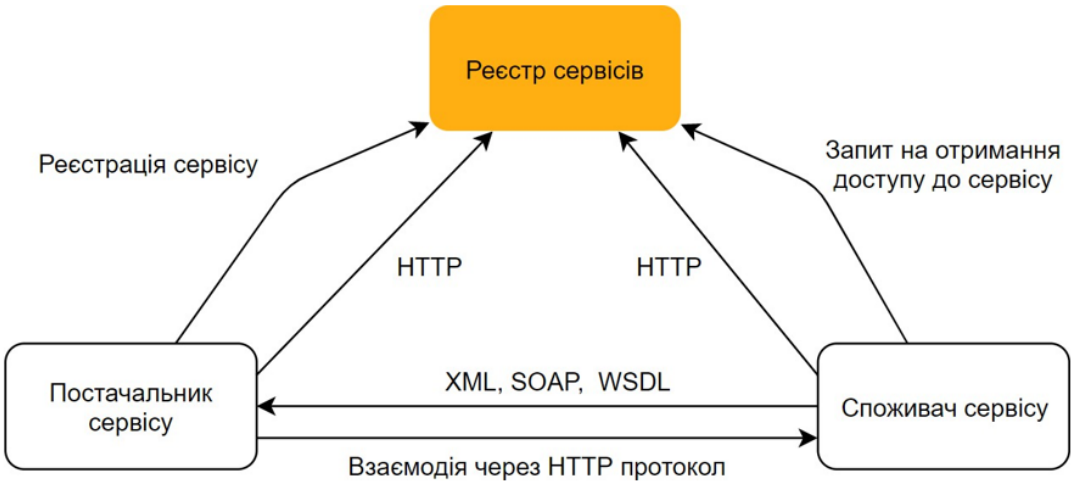


Рисунок 1.6. Ілюстрація схеми взаємодії архітектурних елементів в сервісно-орієнтованій архітектурі (SOA).

До переваг архітектури SOA можна віднести:

1. Можливість повторного використання сервісів: Завдяки автономності та слабкому зв'язку компонентів у сервісно-орієнтованих додатках, ці компоненти можна повторно використовувати в інших програмних додатках, не втручаючись у роботу інших компонентів системи.

2. Простота обслуговування: Оскільки кожен сервіс у програмному додатку є незалежним блоком, його можна легко оновлювати та підтримувати, не змінюючи інші сервіси. Тому керувати великими корпоративними додатками шляхом їх розбиття на кілька сервісів набагато простіше.

3. Висока надійність: Порівняно з великими, пов'язаними модулями в монолітній архітектурі, окремі сервіси легше налагоджувати та тестувати. Тому системи на основі SOA є надійнішими.

4. Паралельна розробка: Сервісно-орієнтована архітектура підтримує паралелізм у процесі розробки, оскільки вона розбивається на незалежні компоненти. Усі сервіси можна розробляти паралельно та завершувати одночасно.

Водночас, до недоліків архітектури SOA відносяться:

1. Складність управління: Основним недоліком сервісно-орієнтованої архітектури є її складність. Кожен сервіс повинен забезпечувати своєчасну доставку повідомлень. Велика кількість таких повідомлень значно збільшує складність управління всіма сервісами.

2. Високі інвестиційні витрати: Розробка систем з використанням сервісно-орієнтованого підходу вимагає значних початкових інвестицій у людські та технічні ресурси.

3. Додаткове навантаження. Оскільки взаємодія між сервісами досягається через обмін повідомленнями, використання кількох сервісів значно збільшує час відгуку та знижує загальну продуктивність системи.

Отже, підсумовуючи, архітектура SOA найкраще підходить для складних корпоративних систем. Наприклад, розбити банківську систему на мікросервіси надзвичайно складно, а монолітні архітектури не підходять, оскільки відмова одного модуля може призвести до збою всієї програми. Тому для таких систем

оптимальним рішенням є застосування SOA-підходу та організація складних системних модулів у незалежні, самодостатні сервіси.

1.4. Постановка задачі.

Сучасні веб-орієнтовані інформаційні системи характеризуються складністю бізнес-логіки, високими вимогами до масштабованості, надійності та швидкості обробки даних. Традиційні монолітні архітектури, хоча й залишаються поширеними, дедалі частіше виявляються недостатньо гнучкими для динамічних умов експлуатації, оскільки ускладнюють оновлення функціональних модулів, горизонтальне масштабування та впровадження нових сервісів. У таких умовах актуальним є впровадження мікросервісної архітектури як підходу, що забезпечує структурну декомпозицію системи, незалежний розвиток компонентів, можливість їх автономного розгортання та ефективний розподіл обчислювальних ресурсів.

Об’єктом розробки в цій роботі є веб-орієнтована інформаційна система, що потребує удосконалення архітектури для забезпечення стабільності, масштабованості та адаптивності до змін середовища. Предметом розробки є методи та засоби побудови мікросервісної архітектури, механізми взаємодії сервісів, а також інструменти забезпечення їхньої цілісності, доступності та узгодженості даних.

Метою розробки є проєктування та реалізація мікросервісної архітектури для веб-орієнтованої інформаційної системи, що дозволить підвищити ефективність її функціонування, забезпечити гнучкість масштабування та полегшити подальший розвиток функціональних модулів.

Для досягнення поставленої мети необхідно вирішити такі задачі:

- проаналізувати наявні архітектурні стилі та технологічні підходи до побудови веб-орієнтованих систем та визначити їхні переваги й недоліки;
- обґрунтувати вибір мікросервісної архітектури як основи для розроблюваної системи;

- визначити вимоги до функціональної та нефункціональної частин системи;
- розробити структурну схему мікросервісної архітектури та описати взаємодію її компонентів;
- обрати технологічний стек та засоби розгортання сервісів;
- реалізувати окремі сервіси та інтегрувати їх у єдине інформаційне середовище;
- провести тестування та оцінити ефективність запропонованої архітектури порівняно з традиційними підходами;
- сформувати рекомендації щодо подальшого розвитку та масштабування системи.

РОЗДІЛ 2.
ОБҐРУНТУВАННЯ, ВИБІР ТА РЕАЛІЗАЦІЯ ІНСТРУМЕНТАРІЮ ВИРІШЕННЯ ЗАДАЧІ.

2.1 Вибір інструментів для побудови мікросервісної архітектури

Одним із ключових інструментів, що широко застосовується для створення мікросервісних систем на платформі Java, є Spring Framework. Це комплексна екосистема бібліотек та модулів, покликана спростити процес розробки програмного забезпечення шляхом застосування механізму ін’єкції залежностей. Використання Spring забезпечує зручний доступ до даних, підтримку валідації, засоби інтернаціоналізації, а також можливість застосування аспектно-орієнтованого програмування. Завдяки сумісності з іншими JVM-мовами, зокрема Kotlin та Groovy, Spring став універсальним рішенням для побудови корпоративних застосунків різної складності [13].

Особливе значення в контексті мікросервісної архітектури має Spring Boot – надбудова над базовим Spring Framework, яка дозволяє створювати самодостатні застосунки зі вбудованим веб-сервером (наприклад, Tomcat) та автоматичним налаштуванням основних компонентів системи. Головною перевагою Spring Boot є мінімізація ручної конфігурації та наявність готових рішень для типових задач. Це дає змогу розробникам зосередитися на бізнес-логіці, а не на низькорівневих технічних налаштуваннях.

Широка популярність екосистеми Spring зумовлена також наявністю спеціалізованих модулів, які значно прискорюють реалізацію різних аспектів роботи застосунку. Серед них варто виділити:

Spring Data – забезпечує уніфікований доступ до реляційних та NoSQL-джерел даних і зменшує необхідність написання рутинного SQL-коду;

Spring Batch – пропонує інструменти для обробки великих обсягів даних та виконання пакетних операцій;

Spring Security – призначений для інтеграції механізмів автентифікації та авторизації, ізолюючи розробника від складних аспектів реалізації безпеки;

Spring Cloud – надає інструменти для створення та підтримки поширених шаблонів у розподілених системах;

Spring Integration – забезпечує реалізацію корпоративних інтеграційних рішень на основі обміну повідомленнями та адаптерів взаємодії.

Вибір Spring як основного програмного фреймворку пояснюється тим, що він значно спрощує побудову структурованої та розподіленої системи, надаючи великий набір готових рішень для ін’єкції залежностей, відокремлення рівнів застосунку та швидкої інтеграції сторонніх модулів. Завдяки вбудованим можливостям автоматичної конфігурації Spring мінімізує потребу в ручному налаштуванні та прискорює процес розгортання.

Spring Boot підтримує використання готових стартових наборів залежностей (starter packages), що включають необхідні бібліотеки для виконання певних задач. Наприклад, пакет spring-boot-starter-web полегшує створення веб-застосунків, а бібліотека Jackson використовується для обробки JSON-структур.

У контексті мікросервісної архітектури важливе місце займає Spring Cloud – набір інструментів, орієнтований на реалізацію типових компонентів розподілених систем, таких як сервісна конфігурація, виявлення сервісів у мережі, інтелектуальна маршрутизація, балансування навантаження, керування відмовами та інші механізми забезпечення стійкості. Використання Spring Cloud дозволяє розробникам швидко будувати повноцінні мікросервісні рішення та ефективно управляти їх взаємодією.

Серед можливостей Spring Cloud можна виділити підтримку централізованого керування конфігураціями, динамічної маршрутизації запитів, сервіс-реєстрації, комунікацій між сервісами, автоматичних перемикачів (circuit breakers), безпечної передачі даних і механізмів узгодження станів у кластері. Завдяки цим інструментам архітектор і розробник можуть швидко створити гнучку, масштабовану та відмовостійку мікросервісну систему.

2.2. Проєктування архітектури мікросервісів на основі сучасного стеку технологій на Java

Проєктування мікросервісної архітектури передбачає застосування комплексу технологій, що забезпечують надійну взаємодію компонентів, їхню масштабованість, відмовостійкість та керованість. Сучасний Java-орієнтований стек пропонує широкий набір інструментів, які дають змогу побудувати повноцінну інфраструктуру для веборієнтованих систем. У межах цієї роботи використано Spring Cloud Gateway, Netflix Eureka, Spring Cloud Config, Netflix Ribbon, механізми автоматичного вимикача, Zipkin, Spring Cloud Sleuth та RabbitMQ, що у поєднанні формують архітектуру, здатну підтримувати високу продуктивність і стійкість.

Шлюз API виконує роль єдиної точки входу в систему та працює як посередник між клієнтом і внутрішніми мікросервісами. Він приймає всі запити, порівнює їх із конфігурацією маршрутів та перенаправляє до потрібних сервісів, що забезпечує приховування внутрішньої структури системи, централізоване управління трафіком та реалізацію нефункціональних вимог. Застосування Spring Cloud Gateway дозволяє використовувати неблокуючий підхід до обробки запитів, що сприяє підвищенню пропускної здатності та зниженню затримок. Окрім маршрутизації, шлюз забезпечує перевірку автентифікації клієнтів, моніторинг проходження запитів, обмеження частоти запитів, аналіз продуктивності, фіксацію подій у журналах і забезпечення безпечної комунікації. На рисунку 2.1 наведено узагальнену схему шаблону API Gateway, побудованого на основі Spring Cloud Gateway [15]. Механізм перенаправлення реалізовано через Java-конфігурацію: коли користувач надсилає запит, наприклад `GET http://localhost:9191/api/payment`, шлюз аналізує відповідність маршруту, пропускає запит через фільтри та переспрямовує його до відповідного мікросервісу, який повертає інформацію про оплату.

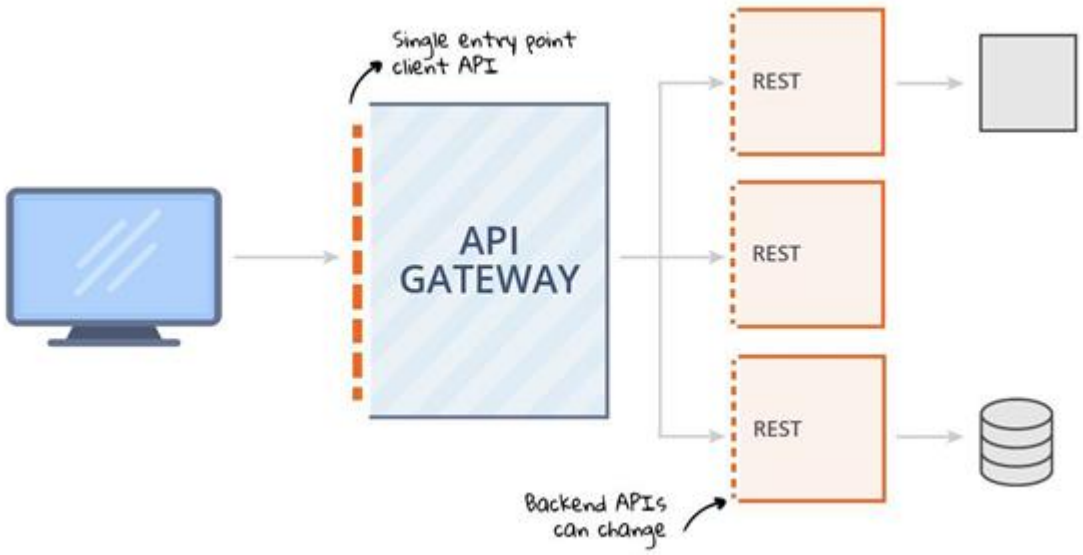


Рисунок 2.1. Шаблон шлюзу API

Функціонування мікросервісної системи неможливе без динамічного визначення розташування сервісів, оскільки їхні екземпляри можуть змінюватися залежно від навантаження або оновлень. Механізм виявлення сервісів (Service Discovery) забезпечує автоматичну реєстрацію всіх сервісів у мережі та їх подальший пошук. Застосування Netflix Eureka дає можливість створити централізований реєстр мікросервісів, куди кожна програма передає відомості про свій стан, IP-адресу, порт та іншу інформацію [16].

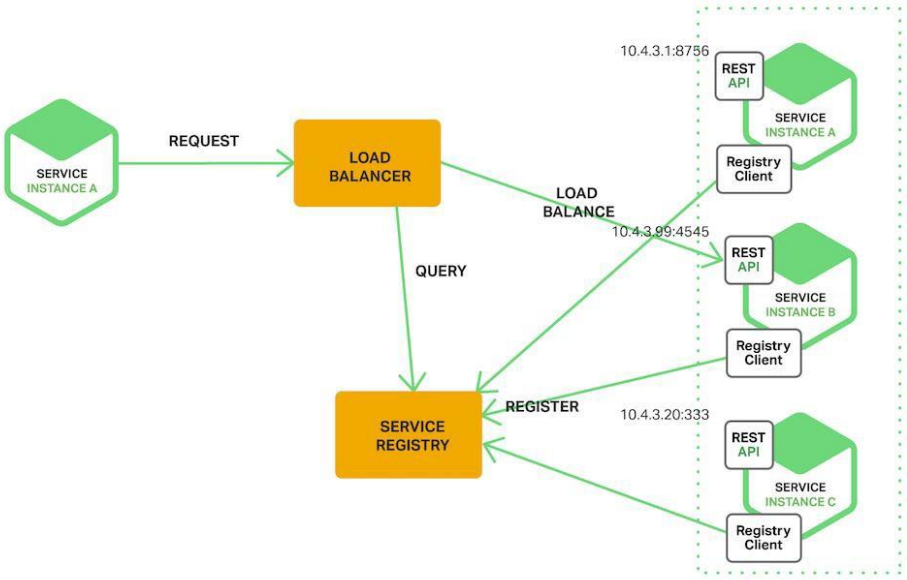


Рисунок 2.2 Service Discovery - інструмент пошуку мікросервісів у мережі

Сервіси спілкуються з Eureka Server і періодично надсилають сигнали активності, що дозволяє своєчасно виявляти недоступні екземпляри. Клієнти використовують кешовану інформацію про сервіси, що зменшує затримки при встановленні з'єднання. На рисунку 2.2 зображено принцип роботи механізму Service Discovery і процес реєстрації екземплярів мікросервісів у реєстрі.

Централізоване керування конфігурацією відіграє важливу роль у зменшенні дублювання параметрів у мікросервісній архітектурі. Використання Spring Cloud Config забезпечує розміщення та обслуговування зовнішніх конфігурацій у єдиному віддаленому репозиторії. Завдяки цьому кожен сервіс отримує актуальні параметри, а зміни вносяться централізовано без необхідності редагувати локальні файли. Spring Cloud Config підтримує інтеграцію з різними сховищами, такими як Git, Consul або Eureka, хоча Git є найбільш зручним завдяки контролю версій та широким можливостям керування історією змін [17]. На рисунку 2.3 зображено структурну схему Spring Cloud Config Server, який виконує роль централізованого сховища конфігурацій.

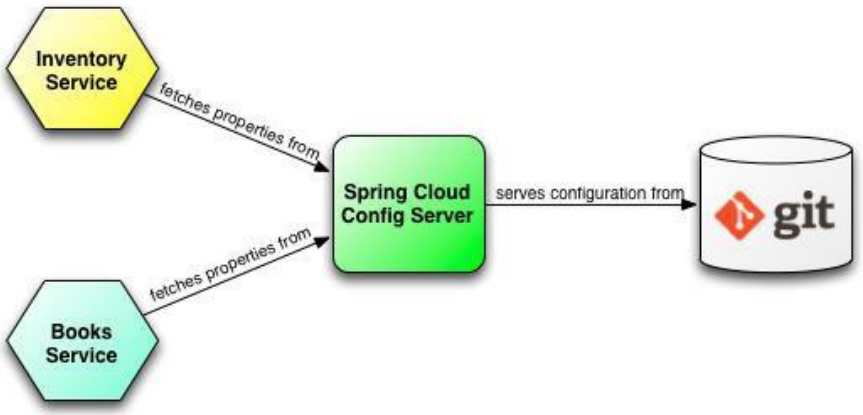


Рисунок 2.3. Spring Cloud Config Server — централізоване сховище конфігурації

Забезпечення рівномірного розподілу навантаження між екземплярами мікросервісів є ключовим завданням для підтримання стабільної роботи системи. Балансування навантаження сприяє оптимізації використання ресурсів, запобігає перевантаженню окремих вузлів та зменшує час відгуку сервісів. Ribbon, який

застосовується в цій роботі, реалізує балансування навантаження на стороні клієнта та дозволяє застосункам самостійно визначати, до якого екземпляра сервісу звертатися [18]. Внутрішні алгоритми Ribbon дозволяють обирати екземпляр за найменшим часом очікування, найменшою кількістю підключень або іншими критеріями. На рисунку 2.4 подано принципову схему клієнтського балансування навантаження.

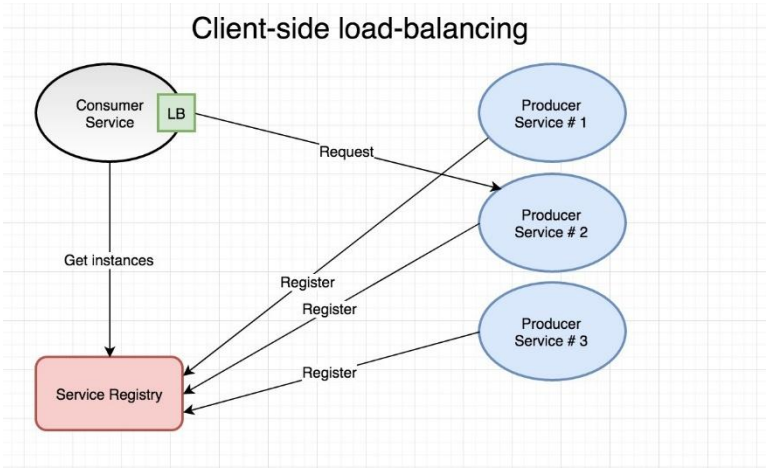


Рисунок 2.4. Балансування навантаження на стороні клієнта

З огляду на можливість часткових збоїв у розподілених системах важливо застосовувати механізми відмовостійкості. Одним із таких рішень є шаблон автоматичного вимикача (circuit breaker), який спрацьовує у разі надмірної кількості помилок або надто великих затримок під час виконання запиту. Автоматичний вимикач запобігає каскадним збоям, блокуючи подальші запити до несправного мікросервісу та повертаючи клієнту контрольоване повідомлення про помилку [18]. Після певного часу система перевіряє доступність сервісу, здійснюючи перехід із «розімкнутого» стану в «напіврозімкнутий», а у випадку успішного відновлення – повертається до нормального режиму. На рисунку 2.5 наведено модель роботи автоматичного вимикача та можливі стани елемента.

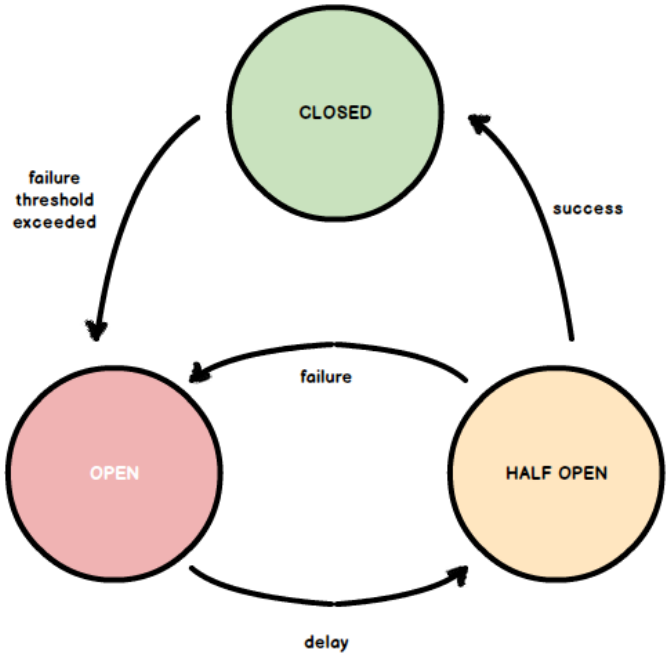


Рисунок 2.5. Модель роботи автоматичного вимикача та можливі стани елемента

Розподілене трасування є необхідним інструментом для відстеження шляху запитів у складній системі, де окремий запит може проходити через кілька мікросервісів і баз даних. За допомогою Zipkin і Spring Cloud Sleuth стає можливим визначення затримок, діагностування помилок та побудова деталізованого графу викликів, що дає змогу виявляти проблемні ділянки архітектури [19]. Кожен запит отримує унікальний trace id, а його часткові операції – span id, що дозволяє реконструювати повну картину виконання.

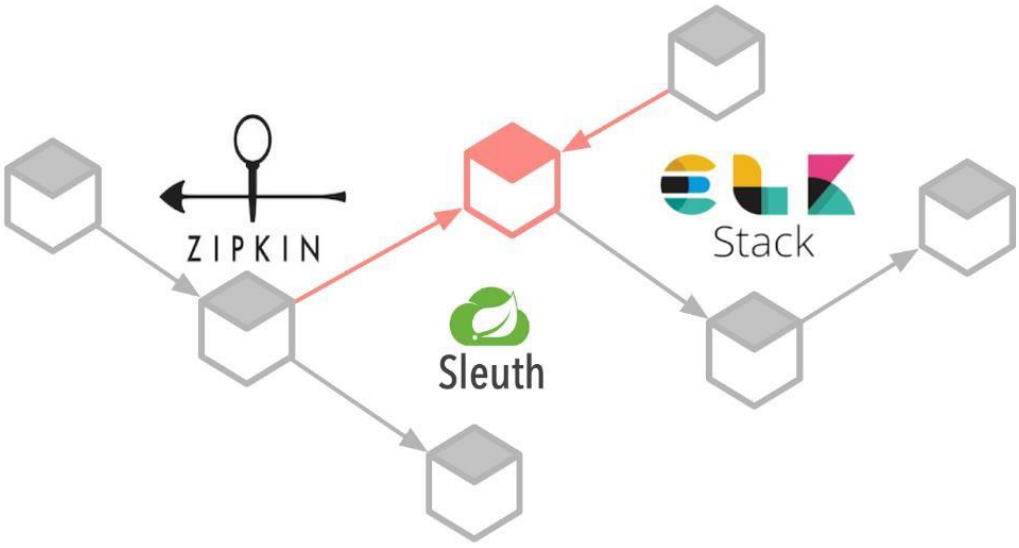


Рисунок 2.6. Бібліотека розподіленого трасування

На рисунку 2.6 зображено загальний принцип функціонування системи розподіленого трасування.

Для асинхронної передачі повідомлень і побудови подієво-орієнтованої взаємодії між сервісами у роботі використано RabbitMQ, який реалізує протокол AMQP. Брокер повідомлень забезпечує приймання, зберігання та доставку повідомлень між сервісами, завдяки чому можна зменшити затримки у відповідях, оптимізувати використання ресурсів і забезпечити стійкість системи до пікових навантажень. Повідомлення передаються між виробником і споживачем через канали, потоки обміну та черги, що дозволяє досягти гнучкої та надійної комунікації [21; 22].

У комплексі застосовані інструменти формують гнучку та стійку мікросервісну архітектуру, у якій забезпечено централізоване керування конфігураціями, динамічне виявлення сервісів, рівномірний розподіл навантаження, механізми захисту від збоїв, прозорість взаємодії компонентів та асинхронний обмін повідомленнями. Такий підхід дає змогу гарантувати високу ефективність, масштабованість і доступність веборієнтованої інформаційної системи.

2.3. Технологічні компоненти та програмні засоби підтримки мікросервісної архітектури.

Побудова мікросервісного застосунку потребує комплексного використання спеціалізованих бібліотек та інструментів, які забезпечують безпеку, оптимізують взаємодію з базою даних, підтримують валідацію, логування, а також коректну роботу веб-рівня застосунку. Основу безпекової інфраструктури формує Spring Security, який забезпечує механізми автентифікації та авторизації користувачів, а також захист від мережевих атак. Він дозволяє створювати гнучкі політики доступу, застосовувати шифрування даних та впроваджувати перевірку прав доступу на основі URL-адрес і регулярних виразів. Архітектура обробки запитів Spring Security наведена на рис.

2.7., де продемонстровано процес проходження автентифікації та авторизації в застосунку.

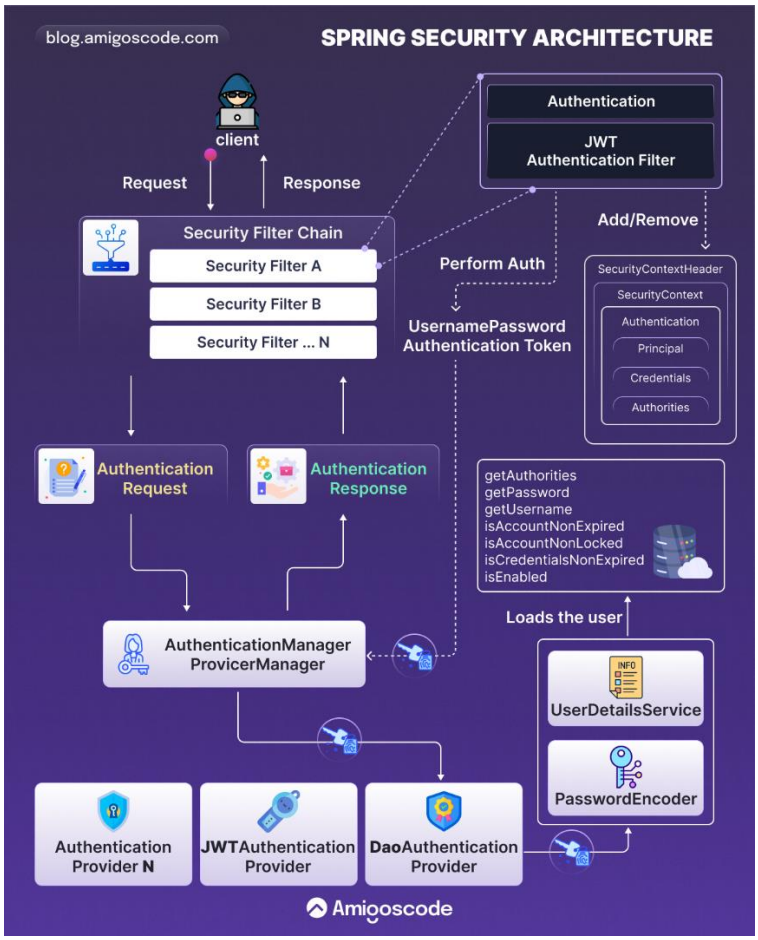


Рис. 2.7. Архітектура механізмів автентифікації та авторизації Spring Security.

Важливим компонентом, що забезпечує ефективну роботу з базою даних, є HikariCP – високопродуктивний пул з’єднань, який оптимізує створення, повторне використання та закриття з’єднань із СУБД. Його архітектура, представлена на рисунку 2.8., демонструє механізм обробки запитів, використання проксі-об’єктів та роботу з попередньо підготовленими з’єднаннями. Завдяки цьому HikariCP забезпечує стабільність та високу швидкість доступу до даних, що особливо важливо у розподілених системах із великою кількістю паралельних операцій.

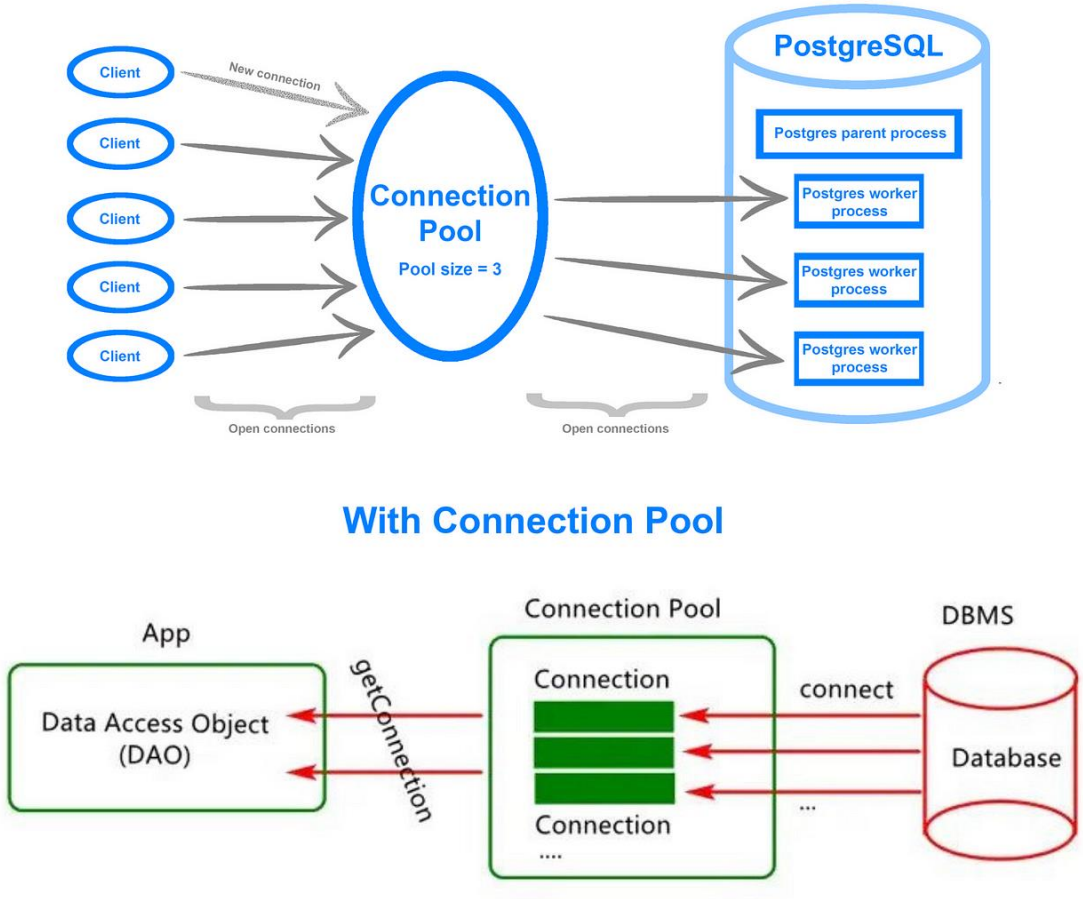


Рис. 2.8. Загальна схема роботи пулу з’єднань HikariCP

Для зменшення ризику передавання некоректних даних у мікросервісах використовується бібліотека spring-boot-starter-validation, яка дозволяє реалізувати валідацію на основі анотацій. Це дає змогу виконувати автоматичну перевірку параметрів HTTP-запитів, тіл об’єктів та змінних шляху без надмірного дублювання коду, що сприяє чистоті та структурованості бізнес-логіки.

Роботу веб-рівня застосунку забезпечує spring-boot-starter-web, що надає механізми створення REST-контролерів, маршрутизації запитів та обробки HTTP-повідомлень. Використання цього інструменту дозволяє формувати повноцінний веб-застосунок зі вбудованим сервером та стандартизованою структурою API.

Суттєвим елементом безпеки у мікросервісній системі є використання JWT-токенів. Їх реалізацію забезпечує бібліотека java-jwt, яка використовується

для генерації, підпису та верифікації токенів доступу. Процес їх формування та використання подано на рис. 2.9., де схематично зображено обмін інформацією між клієнтом та сервером під час авторизованих запитів (рис. 3). Наявність токена у заголовку запиту забезпечує можливість доступу до захищених ресурсів без потреби виконання повторної автентифікації.

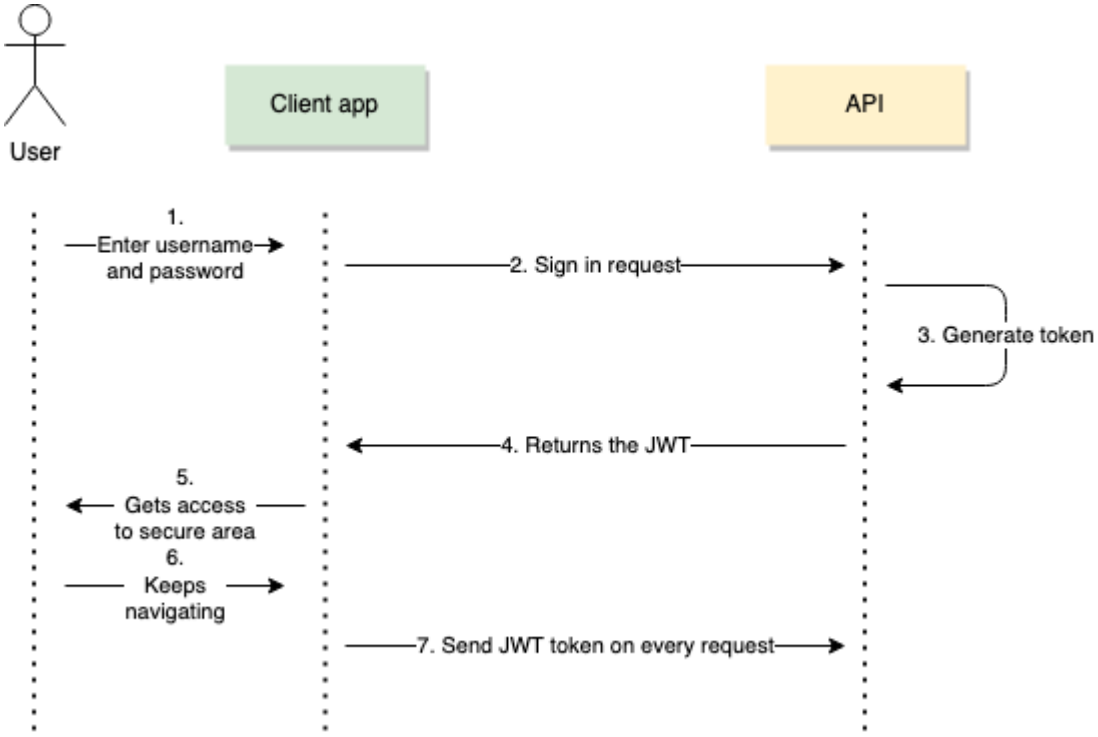


Рис. 2.9. Загальна схема авторизації користувача за допомогою JWT

Для забезпечення централізованого логування в мікросервісному середовищі застосовується Log4j, що дозволяє фіксувати події, помилки та діагностичну інформацію в різних режимах. Така система логування є критично необхідною в умовах масштабованої архітектури, де відстеження потоків запитів допомагає виявляти нестандартну поведінку та своєчасно реагувати на збої.

Доступ до бази даних реалізовано з використанням Spring Data JPA, який значно спрощує взаємодію з ORM і дозволяє автоматизувати створення репозиторіїв, виконання запитів та мапінг об’єктів. Це дає змогу уникнути значної частини шаблонного коду та зосередитися на бізнес-логіці застосунку.

Забезпечення узгодженості структури бази даних здійснюється за допомогою Flyway, що надає гнучкі інструменти для виконання міграцій. На рис. 2.10 наведено принцип роботи Flyway, де показано процес послідовного

застосування SQL-файлів для оновлення схеми бази даних. Flyway забезпечує автоматизацію змін структури БД та виключає ризики розсинхронізації між різними середовищами розгортання.

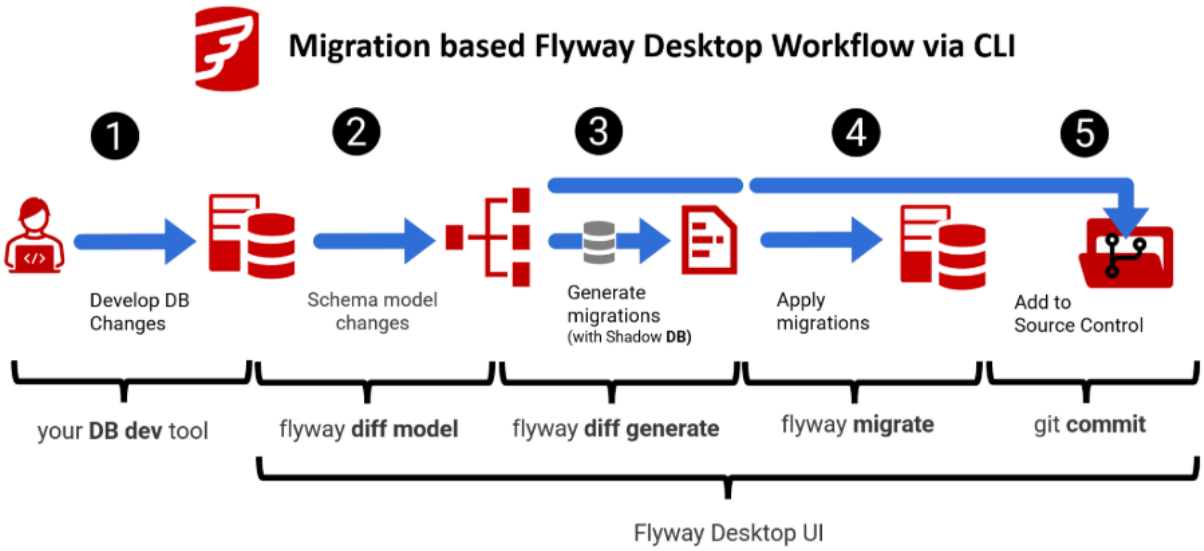


Рис. 2.10 . Принцип роботи механізму міграцій бази даних у Flyway

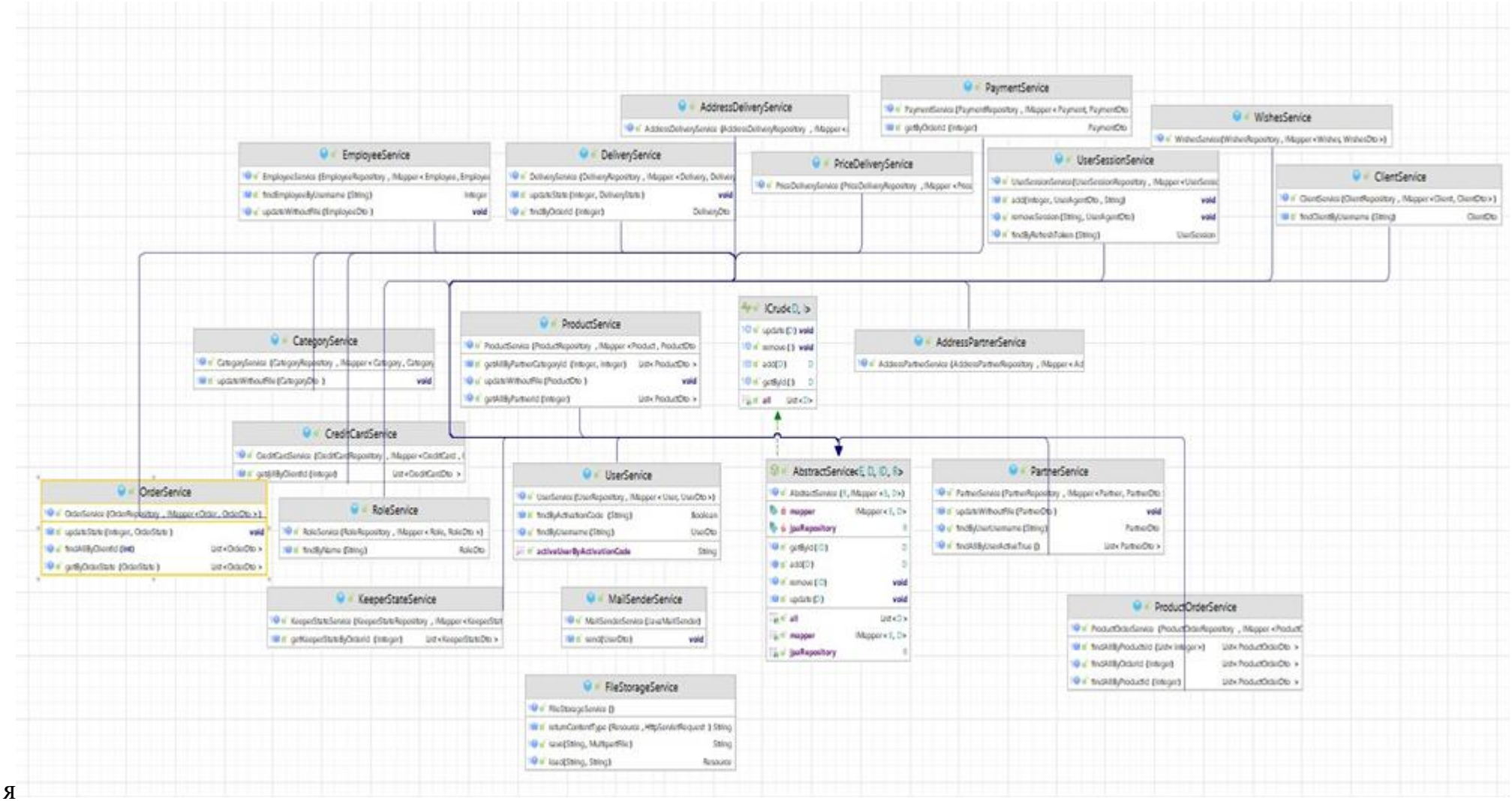
Узгоджена взаємодія перелічених інструментів формує технологічний фундамент мікросервісного застосунку, забезпечуючи його безпеку, надійність, масштабованість та продуктивність. Завдяки поєднанню компонентів безпеки, інструментів доступу до даних, систем логування та механізмів управління схемою бази даних, створюється стабільна архітектура, здатна ефективно функціонувати у високонавантажених розподілених середовищах.

РОЗДІЛ 3.
РУЗУЛЬТАТИ ВИРІШЕННЯ ЗАДАЧІ

3.1 Концепція та вимоги до розробки веб-орієнтованої інформаційної системи

У межах даної кваліфікаційної роботи було поставлено завдання створити комплексну експериментальну інфраструктуру, що складається з двох веб-орієнтованих серверних застосунків, які реалізують однакову предметну область, але побудовані на принципово різних архітектурних концепціях – монолітній та мікросервісній. Такий підхід дає можливість не лише теоретично порівняти обидві архітектури, але й на практиці продемонструвати їх вплив на роботу реального застосунку, поведінку системи під навантаженням, гнучкість у розвитку та здатність до масштабування. Вибір цих архітектур є цілком закономірним у контексті сучасних тенденцій проєктування корпоративних систем, де мікросервіси дедалі частіше замінюють традиційні моноліти, але при цьому обидва підходи все ще активно використовуються у промисловій розробці. Саме тому й було важливо створити два еквівалентні за функціональністю рішення, що дозволяють без зміни бізнес-логіки оцінити відмінності саме на рівні архітектурного дизайну.

Монолітний застосунок у дослідженні реалізовано як цілісну структуру, у якій усі програмні компоненти – сервіси, контролери, шари бізнес-логіки, механізми взаємодії з базою даних – об’єднані в один програмний продукт. Подібний підхід історично є класичним для корпоративних систем і характеризується високою щільністю зв’язків між компонентами та наявністю єдиного загального сховища даних. У межах розробленої системи відповідну структуру продемонстровано на рисунку 3.1, де наочно показано, як інтерфейс ICrud, абстрактний сервісний компонент і конкретні сервіси формують ієрархію доступу до даних і визначають спосіб реалізації CRUD-операцій у межах однієї центральної архітектури.



я

Рис. 3.1 Діаграма класів монолітної архітектури

На противагу моноліту, мікросервісна архітектура, створена у межах цього дослідження, побудована як сукупність незалежних сервісів, кожен із яких реалізує окрему функціональну підсистему, взаємодіє через мережеві протоколи, має власний життєвий цикл, власний механізм розгортання та ізольоване сховище даних. Децентралізація логіки дозволяє проєктувати систему таким чином, щоб кожен сервіс був мінімально залежний від інших, що забезпечує автономність розвитку та оновлення. Детальну структуру сервісів подано на рисунках 3.2 – 3.4., які демонструють, як саме побудовані сервіси постачання (“Delivery”), платежів (“Payment”), замовлень (“Order”), продуктів (“Products”), користувачів (“Users”) і безпеки (“Security”), та яким чином вони організовують взаємодію між собою у розподіленому середовищі.

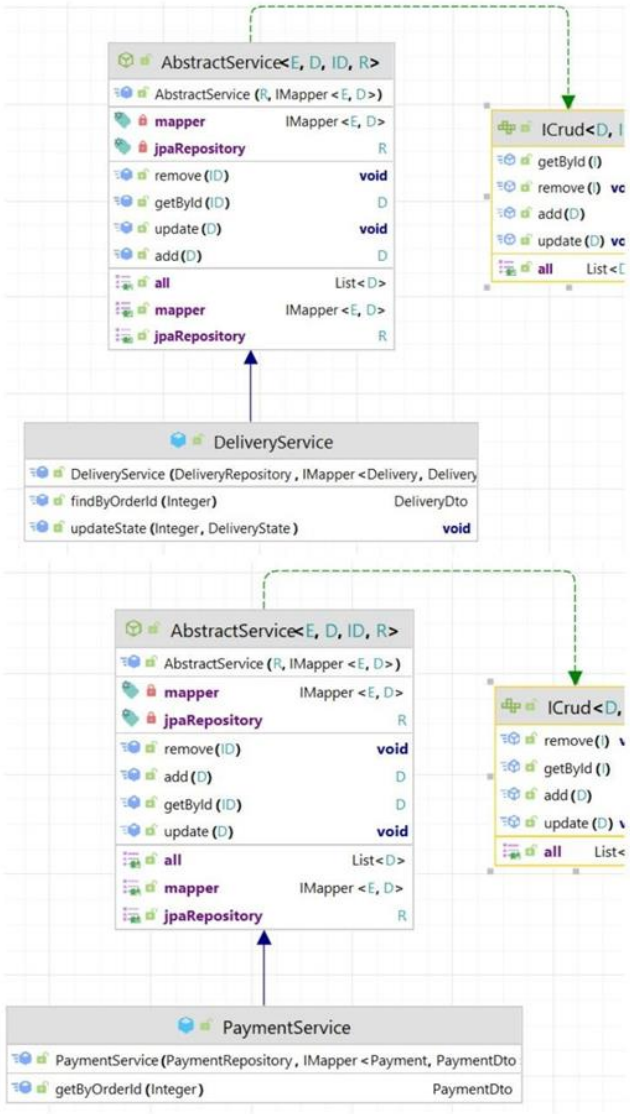


Рис. 3.2. Діаграма класів мікросервісів постачання та платежів.

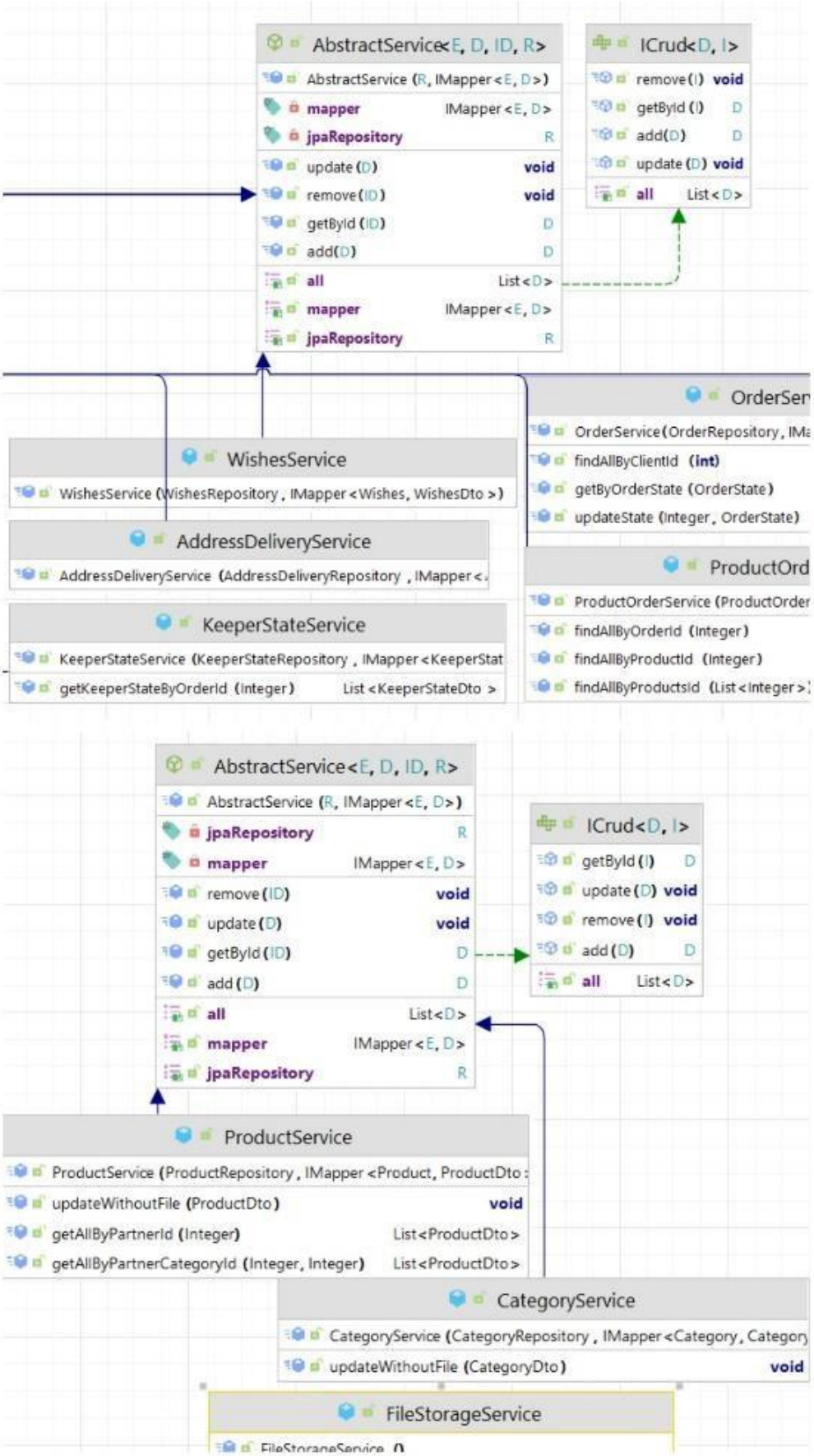


Рис. 3.3. Діаграма класів мікросервісів замовлення та продуктів.

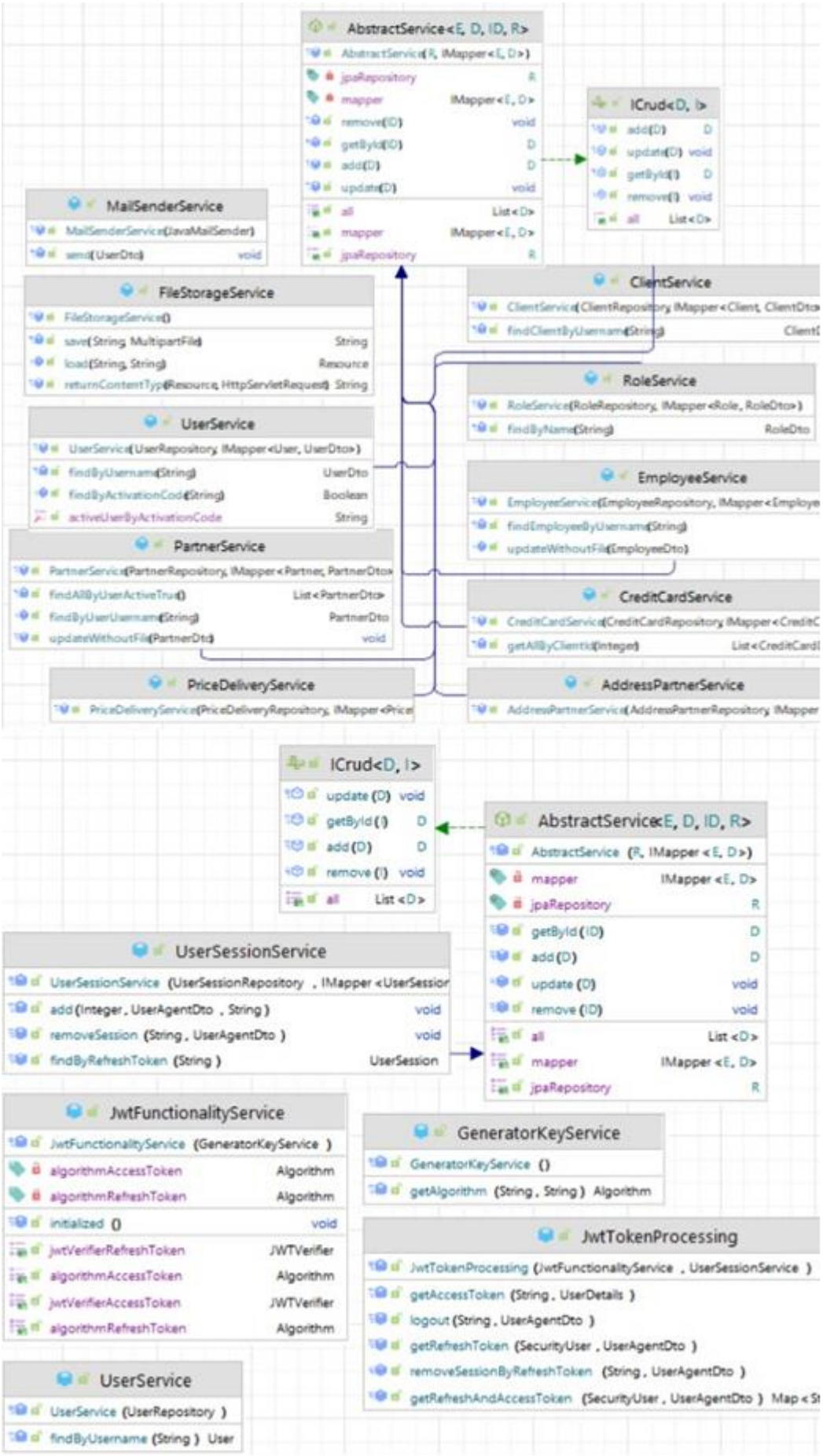


Рис. 3.4. Діаграма класів мікросервісів користувачів та безпеки.

Побудова обох варіантів застосунку вимагала дотримання принципу функціональної еквівалентності, що забезпечувало можливість подальшого коректного порівняння отриманих архітектурних рішень. Для цього у монолітному та мікросервісному варіантах використовувався однаковий набір сутностей предметної області, які відтворювали основні бізнес-процеси: створення та обробка замовлень, взаємодія з користувачами, керування інформацією про продукти, обробка платежів і відстеження станів доставки. Модель однієї з таких сутностей – класу користувача, що відповідає таблиці “Users” у базі даних, – подано на рисунку 3.5. Цей приклад демонструє, як одна й та сама логічна одиниця реалізується у двох архітектурних підходах, але з різними принципами роботи з даними.

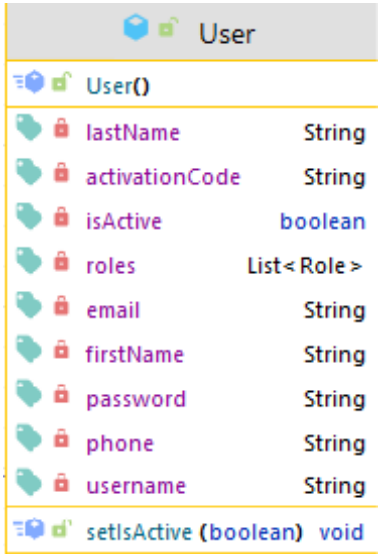


Рис. 3.5. Модель класу користувача.

Додатково у системі враховувалися особливості зміни станів даних. Наприклад, механізм переходів між станами замовлення реалізовано у вигляді перелічувального типу OrderState, що також застосовується в обох архітектурах, але в моноліті обробляється централізовано, тоді як у мікросервісній архітектурі – незалежним сервісом замовлень. Відповідну реалізацію показано на рис. 3.6.

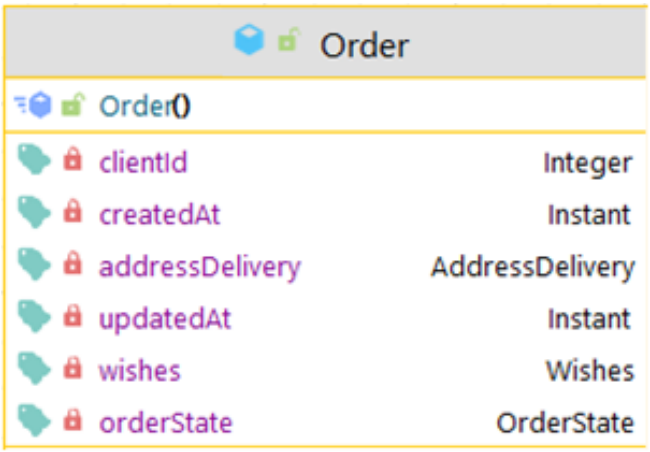


Рис. 3.6. Реалізація перелічуваного типу даних OrderState

Застосування двох архітектурних стилів дозволяє всебічно дослідити ключові аспекти роботи сучасних інформаційних систем. Монолітна архітектура надає можливість продемонструвати простоту початкового розгортання, мінімальні витрати на комунікацію між компонентами та швидку інтеграцію змін на ранніх етапах розробки. Натомість мікросервісна архітектура відкриває ширші можливості для масштабування, дозволяючи збільшувати ресурсну базу окремих сервісів незалежно від інших, а також забезпечує стійкість до відмов шляхом дублювання окремих компонентів або перенесення їх між серверами. Така структура є особливо важливою для високонавантажених систем, де кількість одночасних запитів постійно зростає.

Створені застосунки використовуються як експериментальна платформа для формування комплексного порівняльного аналізу, що охоплює продуктивність, затримки при обробці запитів, час реакції на пікові навантаження, поведінку під час збоїв, відновлення після аварійних ситуацій, а також загальну зручність у супроводі та розширенні системи. Важливо, що в рамках дослідження оцінюється і вплив архітектури на розробницькі процеси, оскільки моноліт дозволяє швидко інтегрувати зміни, а мікросервіси – прискорюють командну роботу, розподіляючи відповідальність між окремими сервісами.

Тому розробка обох архітектурних моделей в рамках цієї кваліфікаційної роботи має на меті забезпечити глибоке практичне розуміння принципів

проєктування сучасних веб-орієнтованих інформаційних систем. Створення тестових застосунків дало змогу не лише визначити ключові переваги та недоліки монолітного і мікросервісного підходів, але й сформувати практичні рекомендації щодо вибору оптимальної архітектури залежно від масштабу, навантаження, бізнес-вимог і подальших планів розвитку системи. Застосунки, описані у цьому розділі, стали основою для ґрунтовного аналізу, результати якого наведено у подальших частинах роботи.

Проєктування веб-орієнтованої інформаційної системи з двома архітектурними варіантами – монолітним і мікросервісним – потребувало формування чітких функціональних та нефункціональних вимог, які визначають, яким саме чином система повинна взаємодіяти з користувачами та якою має бути її якість під час реальної експлуатації. Вимоги були уніфіковані для обох архітектурних підходів, що дозволило створити коректну основу для подальшого порівняння. У рамках розробки було визначено декілька ключових груп користувачів: клієнти системи, кур’єри, партнери та адміністратори. Кожна з цих ролей виконує взаємодії, які формують загальну поведінкову модель застосунку.

Функціональні вимоги охоплювали повний цикл роботи із системою – від перегляду інформації до обробки замовлень та виконання сервісних операцій. Оскільки предметна область системи моделювала структуру корпоративної платформи з обліковими записами, товарами, замовленнями та інфраструктурою обслуговування, для кожної ролі були визначені окремі функції, які повинні забезпечувати як монолітна, так і мікросервісна реалізації. Усі дії користувачів було відображено у вигляді діаграм варіантів використання, що демонструють загальну логіку роботи системи у різних сценаріях.

Поведінка клієнта системи відображена на рисунку 3.7., де представлено типовий набір взаємодій: перегляд інформації, робота з особистим кабінетом, формування замовлень, здійснення оплати та відстеження статусу доставки. Ці сценарії становлять центральну частину бізнес-логіки і мають бути однаково відтворені в обох архітектурах, незалежно від того, чи обробляється запит у монолітному застосунку, чи передається між мікросервісами через API-шлюз.

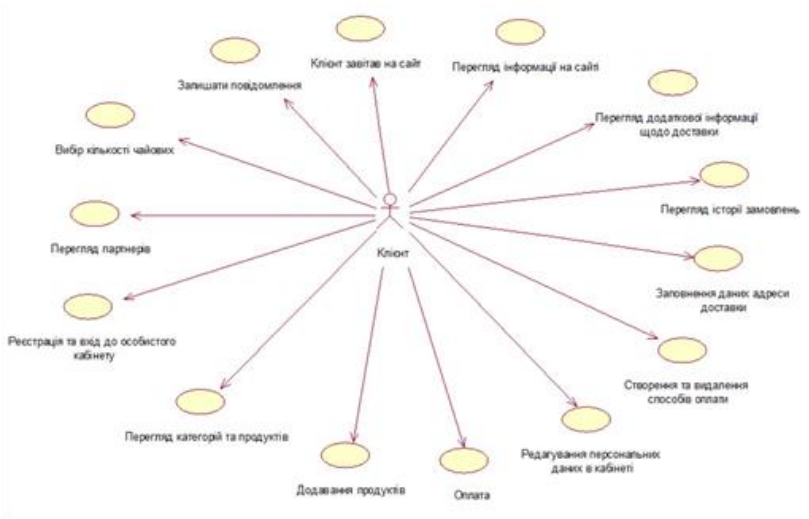


Рис. 3.7. Сценарії використання для клієнта.

Аналогічним чином структуру взаємодії кур’єра подано на рис. 3.8. У межах системи кур’єр отримує доступ до замовлень, змінює статуси виконання та переглядає деталі доставки. У мікросервісній архітектурі такі дії відбуваються через окремий сервіс доставки, тоді як монолітна система обробляє їх у межах одного комплексу.

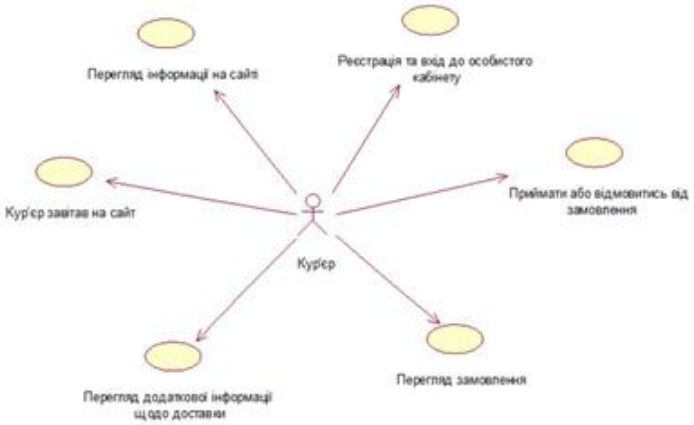


Рис.3.8. Сценарії використання для кур’єра

Для партнерів системи, які відповідають за управління продуктами та взаємодію із замовленнями, на рис. 3.9. показано перелік їхніх можливих сценаріїв використання. Вони включають керування асортиментом, перегляд замовлень та зміну інформації про товари. У мікросервісній архітектурі ці

функції реалізовані окремим сервісом продуктів, що дозволяє масштабувати його незалежно від інших елементів системи.

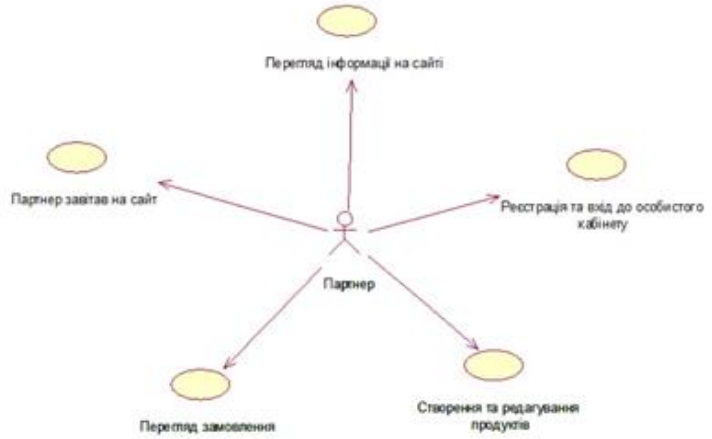


Рис. 3.9. Сценарії використання для партнера

Адміністратор системи має найширші можливості, оскільки забезпечує управління даними всіх інших категорій користувачів. На рис. 3.10. представлено адміністративні сценарії, що передбачають перегляд, редагування та контроль над усіма сутностями системи. У монолітній архітектурі адміністраторська панель функціонує як частина одного процесу, тоді як у мікросервісній вона координує запити до різних сервісів.

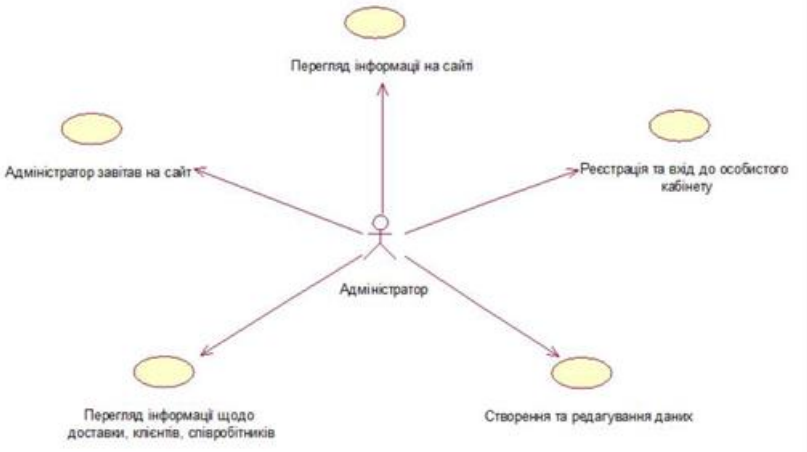


Рис. 3.10. Сценарії використання для адміністратора

Окрім визначення варіантів поведінки, важливу роль відіграло моделювання послідовності дій користувачів у вигляді діаграм послідовності. Такі діаграми дозволяють описати динаміку взаємодії між клієнтом та системою

та показати, як саме передаються дані між різними компонентами. Наприклад, процес реєстрації клієнта наведено на рис. 3.11. У цьому випадку заявка проходить перевірку, формується лист підтвердження, а після активації облікового запису клієнт отримує доступ до внутрішньої частини системи.

Рис. 3.11. Послідовність дій під час реєстрації клієнта

Другий ключовий сценарій – це оформлення замовлення, який продемонстровано на рис. 3.12. Цей процес є найбільш комплексним, включаючи вибір ресторану або партнера, вибір продуктів, формування кошика, здійснення оплати та формування детальної інформації про виконання. У монолітній архітектурі всі ці етапи виконуються послідовно в межах одного застосунку, тоді як у мікросервісній частини логіки обробляються окремими сервісами, що взаємодіють через REST-запити.

Рис. 3.12. Послідовність дій під час оформлення замовлення

На рис. 3.13. подано послідовність дій під час створення та видалення кредитної картки в системі. Це важливий сценарій, оскільки робота з конфіденційними даними вимагає реалізації додаткових механізмів безпеки, зокрема токенізації, шифрування та валідації інформації. Для обох архітектур збережено однакову модель захисту, але у мікросервісному варіанті її виконання є частиною сервісу автентифікації та сервісу платежів.

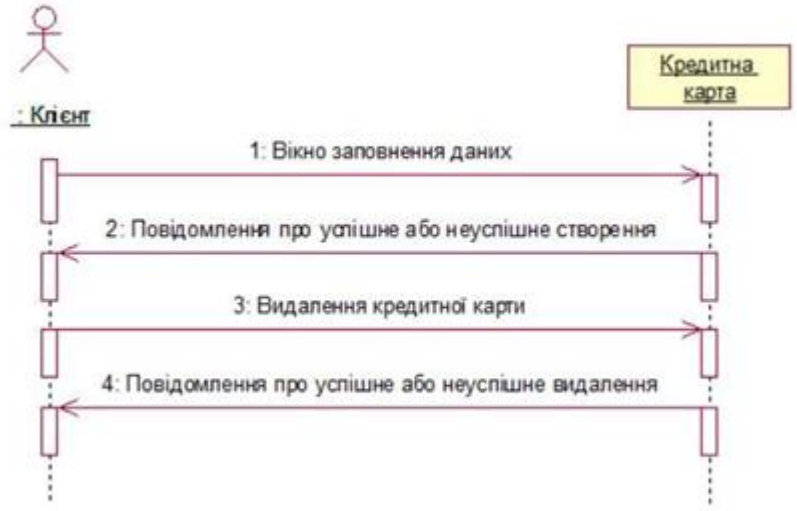


Рис. 3.13. Послідовність дій під час створення та видалення картки

Окрему увагу було приділено моделюванню входу співробітників у систему адміністрування, що продемонстровано на рис.3.14. Даний сценарій охоплює як процес автентифікації, так і авторизацію з використанням токенів, що забезпечує коректний доступ до ресурсів відповідно до ролі користувача.

Рис. 3.14. Послідовність дій співробітників під час входу в систему

Нефункціональні вимоги, що були сформовані під час проєктування, визначають якість функціонування системи. Оскільки застосунок повинен витримувати навантаження та забезпечувати стабільність роботи в умовах великої кількості одночасних запитів, особливу увагу було приділено швидкодії, відмовостійкості, захисту даних і простоті користувацької взаємодії. Мікросервісна архітектура дозволяє досягти високого рівня масштабованості та забезпечити стійкість до відмов окремих компонентів, тоді як моноліт надає стабільність внутрішніх комунікацій та передбачувану поведінку в межах єдиного серверного середовища. Зрештою, обидві архітектури повинні гарантувати якісний користувацький досвід, швидку обробку запитів та безпечне виконання операцій з даними.

Проєктування інтерфейсу веб-орієнтованої інформаційної системи у межах даної кваліфікаційної роботи базувалося на принципах простоти, інтуїтивності та відповідності потребам кожної категорії користувачів. Оскільки система реалізована у двох архітектурних варіантах, важливо було забезпечити повну ідентичність зовнішнього вигляду та логіки роботи інтерфейсу для монолітної та мікросервісної версій застосунку. Це дозволяє гарантувати, що різниця у поведінці продуктів зумовлена саме архітектурними особливостями, а не відмінностями у користувацькому середовищі.

Інтерфейс було розроблено відповідно до сучасних вимог UI/UX-дизайну, що передбачає зручність взаємодії навіть для користувачів, які раніше не працювали з подібними платформами. Головна сторінка застосунку слугує точкою доступу до ключових функцій системи та містить інформацію про партнерів платформи, можливості сервісу та базові інструкції для нових користувачів. На рис. 3.14. подано зовнішній вигляд головної сторінки, що адаптована для швидкого ознайомлення зі структурою застосунку.

Рис. 3.14. Головна сторінка веб-застосунку.

Подальша навігація забезпечує користувачеві доступ до переліку партнерів, де можна переглядати доступні категорії товарів та здійснювати замовлення. На рис. 3.15. представлено інтерфейс перегляду партнерів, що демонструє структуроване розташування інформації, адаптоване під потреби клієнтів, які шукають товари або послуги за певними критеріями.

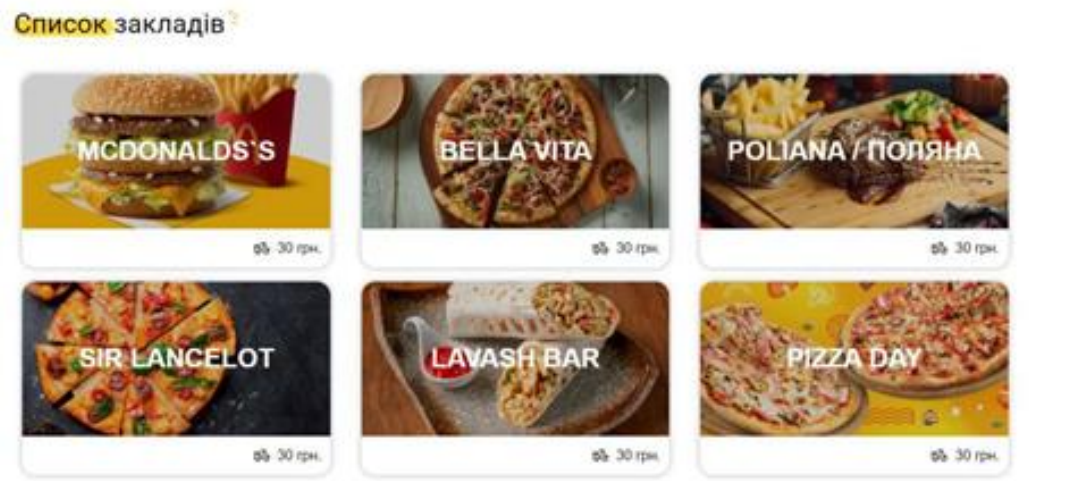


Рис. 3.15. Інтерфейс перегляду партнерів

Для партнерів платформи було створено окреме інтерфейсне середовище, призначене для керування асортиментом, перегляду замовлень та взаємодії зі співробітниками сервісу. Розробка цієї частини інтерфейсу вимагала врахування значно ширшого набору функцій, оскільки партнер має здійснювати повний контроль над товарами та інформацією, що надходить від клієнтів. На рис. 3.16. зображено сторінку керування категоріями та продуктами партнера, яка відображає всі елементи контенту в зручному і логічно впорядкованому вигляді.

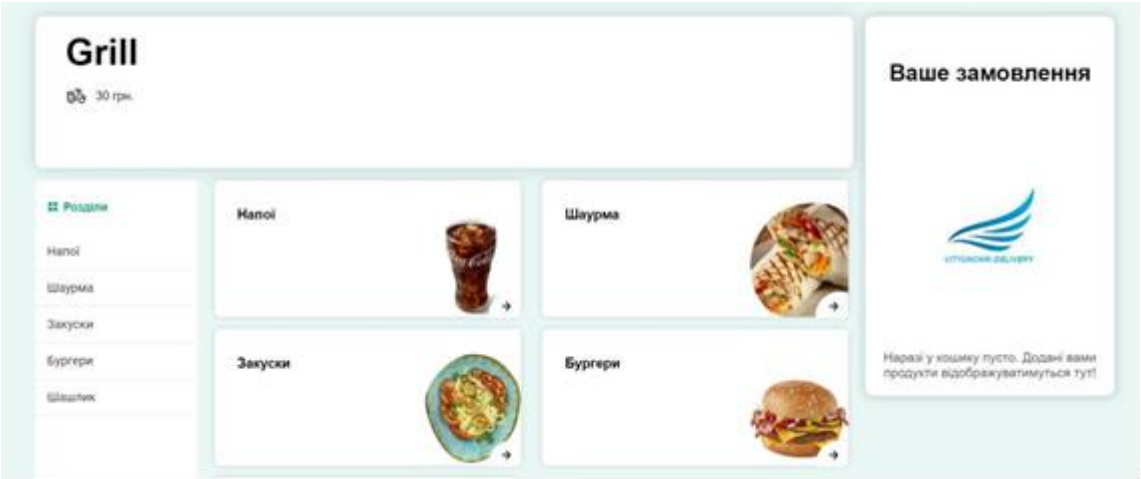


Рис. 3.16. Сторінка категорій партнера

Інтерфейс для кур’єрів зосереджений на оперативній роботі із замовленнями. Його структура передбачає доступ до списку активних замовлень, можливість запуску доставки, перегляду деталей та завершення виконання.

Інтерфейс подано на рис. 3.17., що демонструє простоту використання та швидкий доступ до ключових функцій, необхідних у процесі виконання доставки.

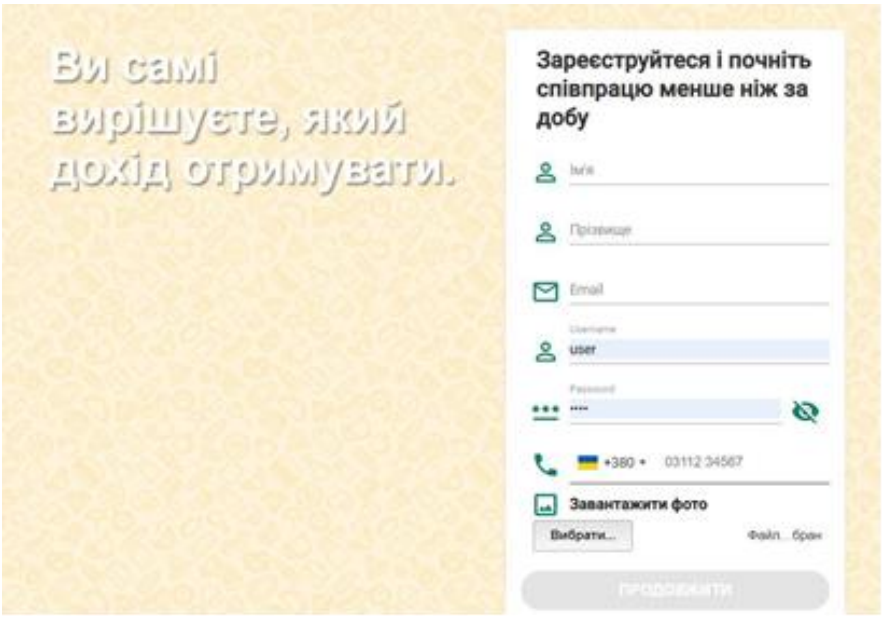


Рис. 3.17. Сторінка реєстрації нового співробітника (кур’єра)

На рис. 3.18. продемонстровано сторінку управління продуктами, що містить перелік товарів партнера в межах відповідної категорії. Цей елемент інтерфейсу забезпечує можливість взаємодії з товарними позиціями, включно зі зміною опису, додаванням нових продуктів або видаленням непотрібних.

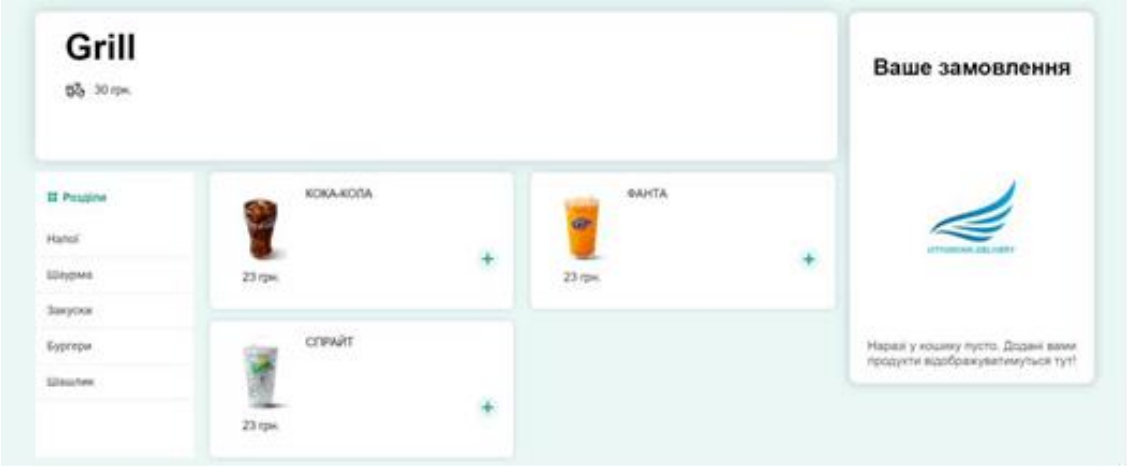


Рис. 3.18. Сторінка для перегляду продуктів партнера

Загалом, вимоги до інтерфейсу були сфокусовані на побудові універсального та зручного користувацького середовища, здатного забезпечити швидке та комфортне виконання бізнес-операцій. Незалежно від обраної архітектури, (монолітної чи мікросервісної) інтерфейс залишається незмінним, адже саме єдність користувацького досвіду дозволяє об’єктивно оцінити, як саме архітектура впливає на продуктивність і стабільність роботи системи. Усі інтерфейсні модулі було реалізовано на основі Angular Framework, що забезпечило високу адаптивність, інтерактивність та можливість подальшого масштабування фронтенда без прив’язки до конкретної серверної архітектури.

3.2. Засоби реалізації та технологічне середовище розробки

Розроблення корпоративних застосунків у рамках даної роботи здійснювалося із використанням сучасного технологічного стеку, орієнтованого на забезпечення ефективної роботи як монолітної, так і мікросервісної архітектур. Вибір засобів реалізації був зумовлений необхідністю підтримання масштабованості, надійності, високої продуктивності серверної частини та зручності створення клієнтського інтерфейсу. Технологічне середовище формувалося таким чином, щоб забезпечити модульність, повторне використання коду та можливість подальшого розширення функціональності системи.

Середовище розробки IntelliJ IDEA виступало основним інструментом для створення серверної частини, оскільки воно забезпечує широкі можливості для роботи з мовою Java, інтегрованими модулями проєктування та налагодження, а також містить інструменти для автоматизації збірки та аналізу коду. Вибір мови Java був цілком логічним, адже вона є однією з найбільш поширених мов для створення корпоративних застосунків, надає розвинуту екосистему бібліотек і фреймворків та підтримує потужні механізми багатопоточної обробки, що є критично важливим як у монолітних, так і в розподілених системах.

Java Spring Framework став ключовим інструментом для реалізації серверної логіки. Архітектура монолітного застосунку використовувала базові можливості Spring для створення контролерів, сервісів, репозиторіїв та конфігураційних компонентів. Використання механізму ін’єкції залежностей забезпечило слабке зв’язування між компонентами, що значно спрощувало супровід та розвиток системи. Spring Boot, як невід’ємна частина екосистеми, дозволив спростити процес запуску й конфігурації застосунків, надаючи автоматизоване налаштування типових компонентів, власний вбудований сервер та шаблони конфігурацій, що особливо важливо при розробці мікросервісів.

Для реалізації мікросервісної архітектури використовувалися додаткові компоненти Spring Cloud, які забезпечують роботу механізмів сервіс-дискавері, конфігураційного сервера, маршрутизації запитів та fault-tolerance. Завдяки цим інструментам кожен сервіс функціонує автономно, має можливість динамічної реєстрації в загальному реєстрі, отримує централізовані конфігурації та взаємодіє з іншими сервісами незалежно від їхнього фізичного розташування. Такий підхід робить систему адаптивною до зміни навантаження, а також дозволяє безболісно додавати нові модулі або модифікувати існуючі.

У ролі системи керування базами даних обрано PostgreSQL, що забезпечує надійність, високу продуктивність та відповідність вимогам ACID. У монолітній системі вона виступає єдиним сховищем, тоді як у мікросервісній архітектурі кожен сервіс має окрему базу даних, що повністю відповідає патерну «Одна база даних на один сервіс». Такий підхід забезпечує незалежність між сервісами, можливість різного масштабування та мінімізує ризики перехресного впливу на дані.

Для керування міграціями використовувалася система Flyway, яка дозволила централізовано контролювати еволюцію структури бази даних у різних середовищах. Під час розроблення мікросервісів Flyway застосовувався окремо для кожного сервісу, що гарантувало синхронність структур даних у межах локальних баз. Механізм міграцій особливо важливий для великих

систем, де будь-яка зміна структури БД повинна бути детермінованою та відтворюваною.

Захист серверної частини реалізовувався через Spring Security, який забезпечував автентифікацію, авторизацію й контроль доступу до ресурсів. Додатково використовувалася бібліотека JWT для генерації токенів доступу, що дозволяло реалізувати безстанну модель авторизації, зручну для мікросервісної архітектури. Токени містили ідентифікаційні дані користувача та підписувалися секретним ключем, що гарантувало їхню цілісність і захищеність від підроблення.

Клієнтську частину було реалізовано засобами Angular Framework, який забезпечує модульність інтерфейсу, підтримку двостороннього зв’язування, високий рівень інтерактивності та можливість масштабування. Angular дозволив організувати інтерфейс у вигляді набору компонентів, що полегшило повторне використання UI-елементів, структурування коду та оптимізацію логіки відображення. Для клієнтської частини особливо важливою була можливість швидкого реагування на користувацькі дії, що досягається завдяки механізмам реактивності фреймворку.

Сукупність використаних інструментів забезпечила створення повноцінної інфраструктури для порівняння монолітної та мікросервісної моделей. Обидва застосунки реалізовано в єдиному технологічному середовищі, що гарантує коректність порівняння за продуктивністю, швидкістю обробки запитів, масштабованістю й рівнем адаптивності до зміни навантаження. Завдяки застосуванню зрілих фреймворків і перевірених технологій отримано стабільні й легко підтримувані системи, здатні відтворювати реальні умови функціонування корпоративних веб-орієнтованих платформ.

3.3. Модулі та алгоритми функціонування серверної частини

Архітектура корпоративної системи, побудованої у межах даної роботи, включає набір окремих модулів, що реалізують ключові алгоритми обробки

даних, забезпечення безпеки, взаємодії з файловою системою та виконання бізнес-логіки. Незалежно від типу архітектури, кожен із цих модулів виконує детерміновану частину операцій, забезпечуючи узгодженість даних, надійність функціонування та відповідність вимогам до систем корпоративного рівня. Концептуально модулі системи можна поділити на групи, що відповідають за конфігурацію безпеки, управління даними користувачів, роботу з файлами та забезпечення коректних переходів статусів у процесах замовлень і платежів.

Одним із центральних елементів системи є модуль конфігурації безпеки, який регламентує правила обробки HTTP-запитів та визначає, за яких умов користувач може отримати доступ до захищених ресурсів. У межах цього модуля реалізовано алгоритм, що описує поведінку сервера при взаємодії з автентифікованими та неавтентифікованими клієнтами. Конфігурація включає вимкнення механізму CSRF-захисту в тих середовищах, де система працює у відкритому або тестовому режимі, після чого визначаються сегменти запитів, доступні без автентифікації, а також ті, що потребують перевірки доступу на основі JWT-токена. За допомогою ланцюжка викликів методів встановлюються політики створення сесій, точки входу для обробки помилок автентифікації, правила доступу до ресурсів для різних ролей користувачів, а також додається фільтр авторизації запитів. Усе це реалізується в методі configure(HttpSecurity http), фрагмент якого наведено на рис. .3.19.

Цей фрагмент демонструє, як на рівні конфігурації формується політика безпеки: з одного боку, відкриваються загальнодоступні ресурси (наприклад, статичні файли), з іншого — захищаються критичні API-методи, доступ до яких дозволений лише користувачам з певними ролями. Саме через такі конфігурації реалізуються принципи розмежування доступу, що є важливими як для монолітної, так і для мікросервісної реалізації.

```
@Override
public void configure(HttpSecurity http) throws Exception {

    http
        .csrf().disable()
        .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .exceptionHandling()
            .authenticationEntryPoint(jwtAuthenticationEntryPoint)
        .and()
        .authorizeRequests()
            .mvcMatchers("/assets/**").permitAll()
            .mvcMatchers(this.api + "/partner")
                .hasAnyAuthority(Roles.ADMIN.getType(), Roles.PARTNER.getType())
            .mvcMatchers(this.api + "/role")
                .hasAnyAuthority(Roles.ADMIN.getType(), Roles.PARTNER.getType())
            .mvcMatchers(this.api + "/address-partner")
                .hasAnyAuthority(Roles.ADMIN.getType(), Roles.PARTNER.getType())
            .mvcMatchers(this.api + "/category")
                .hasAnyAuthority(Roles.ADMIN.getType())
            .mvcMatchers(this.api + "/product")
                .hasAnyAuthority(Roles.ADMIN.getType(), Roles.PARTNER.getType())
            .mvcMatchers(this.api + "/credit-card")
                .hasAnyAuthority(Roles.CLIENT.getType())
            .mvcMatchers(this.api + "/product/**")
                .hasAnyAuthority(Roles.ADMIN.getType(), Roles.PARTNER.getType())
            .anyRequest().authenticated()
        .and()
        .logout()
            .logoutUrl(this.api + "/auth/logout")
        .and()
        .addFilterBefore(
            new RequestAuthorizationFilter(jwtFunctionalityService, tokenHeader),
            UsernamePasswordAuthenticationFilter.class
        );
}
```

Рис. 3.19. Фрагмент конфігурації безпеки застосунку

Іншим важливим компонентом є модуль управління даними клієнтів, який відповідає за отримання, обробку та передачу інформації між сервісом та користувачем. Алгоритм роботи цього модуля полягає в тому, що запит надходить до контролера, який делегує його обробку сервісному шару. Сервіс взаємодіє з репозиторієм, отримує список клієнтів у вигляді DTO-структур, а результат повертається клієнту разом із відповідним HTTP-кодом. У випадку успішної обробки сервіс повертає відповідь з даними, у разі помилки — її перехоплює глобальний обробник винятків. Приклад такого контролера наведено на рис. 3.20.

```
@GetMapping
public ResponseEntity<List<ClientDto>> getAll() {
    LOGGER.debug("Received clients!");
    return new ResponseEntity<>(clientService.getAll(), HttpStatus.OK);
}
```

Рис. 3.20. Метод отримання списку клієнтів

Цей метод ілюструє типовий патерн роботи контролера: виклик сервісу, обробка результату та формування HTTP-відповіді. Логування дозволяє відстежувати виконання запиту, а використання ResponseEntity забезпечує гнучкість щодо встановлення як тіла відповіді, так і статусу.

У системі також присутній модуль роботи з файлами, який оперує механізмами збереження та завантаження об’єктів файлової системи. Алгоритм збереження передбачає визначення базового каталогу для запису файлів, формування унікального імені файлу з використанням генератора UUID, перевірку наявності потенційних конфліктів і фізичне копіювання вмісту в цільовий каталог. У разі виникнення помилки генерується виняток, а відповідна інформація фіксується у журналі. Приклад реалізації такого підходу наведено в коді, який зображено на рис. 3.21.

```
public String save(String path, MultipartFile file) throws IOException {
    try {
        Path root = Paths.get(path);
        String fileName = UUID.randomUUID().toString() + "-" + file.getOriginalFilename();
        Path filePath = root.resolve(fileName);

        if (Files.exists(filePath)) {
            fileName = UUID.randomUUID().toString() + "-" + file.getOriginalFilename();
            filePath = root.resolve(fileName);
        }

        Files.copy(file.getInputStream(), filePath);
        return fileName;
    } catch (IOException e) {
        logger.error(e.getMessage(), e);
        throw new IOException(e.getMessage());
    }
}
```

Рис. 3.21.Фрагмент коду реалізації процесу збереження файлу на сервері

Алгоритм завантаження файлу з сервера, навпаки, перевіряє наявність ресурсу за заданим шляхом, читає його в пам’ять і повертає користувачу у вигляді об’єкта Resource. Якщо файл відсутній або недоступний, генерується виняток, який фіксується в журналі, а клієнтові може бути повернута відповідь із повідомленням про помилку. Фрагмент цього механізму наведено на рис. 3.21.

```
public Resource loadFileWithResources(String path, String filename)
        throws IOException, URISyntaxException {

    String root = path + "/" + filename;

    try (InputStream url = this.getClass()
                               .getClassLoader()
                               .getResourceAsStream(root)) {

        if (url != null) {
            Resource resource = new ByteArrayResource(url.readAllBytes());
            if (resource.exists() || resource.isReadable()) {
                return resource;
            } else {
                throw new FileNotFoundException("Could not read the file!");
            }
        } else {
            throw new FileNotFoundException("File does not exist!");
        }
    } catch (FileNotFoundException e) {
        logger.error(e.getMessage(), e);
        throw new FileNotFoundException("Error: " + e.getMessage());
    }
}
```

Рис. 3.21.Фрагмент коду реалізації процесу завантаження файлу з сервера

Ще одним ключовим компонентом є модуль, що відповідає за зміну статусів у процесах замовлень і платежів. Алгоритм його роботи полягає у визначенні поточного стану оплати та встановленні нового значення статусу відповідно до бізнес-правил. Якщо клієнт здійснив оплату безготівково, статус замовлення встановлюється як «сплачено», а якщо тип оплати передбачає готівковий розрахунок, статус може залишатися «не сплачено» до моменту фактичного розрахунку. Після обчислення нового стану сервіс оновлює запис у базі даних і повертає відповідну відповідь клієнту. Приклад такого методу наведено на рис. 3.22.

```
@GetMapping("order-payment/{order-id}")
public ResponseEntity<String> changeOrderPaymentState(
        @PathVariable("order-id") Integer orderId) {

    PaymentDto payment = this.paymentService.getByOrderId(orderId);

    OrderState state = payment.getPaymentType()
                              .equals(PaymentType.MONEY.getType())
                       ? OrderState.NOT_PAID
                       : OrderState.PAID;

    this.orderService.updateState(orderId, state);
    LOGGER.debug("Received order-state {}!", state);

    return new ResponseEntity<>(state.getType(), HttpStatus.OK);
}
```

Рис. 3.22. Зміна статусу оплати замовлення

Цей фрагмент демонструє, як бізнес-логіка оплати інтегрується з механізмом зміни стану замовлення. Всі переходи між станами фіксуються, що дозволяє відстежувати історію змін та забезпечувати узгодженість даних між модулем платежів та модулем замовлень.

Усі перелічені модулі працюють у взаємозв’язку, формуючи єдине серверне середовище, здатне забезпечити стабільне функціонування системи

незалежно від обраної архітектурної моделі. Їхня взаємодія, вибудувана на принципах слабкого зв’язування, чітких контрактів та обмеженої зони відповідальності, створює основу для масштабованості, ефективності та стійкості системи. Саме ці модулі й алгоритми визначають внутрішню логіку корпоративного застосунку та забезпечують можливість його подальшої еволюції, інтеграції нових компонентів та адаптації до змін бізнес-вимог.

3.4. Технологічні засади та процес розгортання серверних застосунків з монолітною і мікросервісною архітектурами.

Процес розгортання серверної частини інформаційної системи є критично важливим етапом життєвого циклу програмного забезпечення. Незалежно від того, чи використовується монолітна модель, чи мікросервісна архітектура, підготовка застосунку до експлуатації передбачає низку технічних операцій, що забезпечують коректну компіляцію, збирання, пакування та подальше розміщення програмних компонентів у відповідному середовищі. Хоча підходи до розгортання двох архітектурних варіантів суттєво різняться за складністю та рівнем автоматизації, їх можна розглядати в єдиному контексті технологічного забезпечення серверної частини системи.

У випадку монолітної архітектури основою процесу розгортання є система керування проєктами Maven, що забезпечує формування структурованого POM-файлу, компіляцію модулів і створення виконуваного JAR-файлу. Файл pom.xml, який містить опис монолітного застосунку, наведено у лістингу на рис. 3.23. Він визначає метадані проєкту, залежності, посилання на плагіни та параметри збирання. На основі цієї конфігурації Maven послідовно виконує очищення структури проєкту, валідацію, компіляцію, тестування та формування кінцевого пакета, що готовий до розміщення на сервері або у внутрішньому сховищі артефактів.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.6</version>
        <relativePath/>
    </parent>

    <groupId>com.server</groupId>
    <artifactId>itdelivery</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <java.version>14</java.version>
    </properties>

    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>com.zaxxer</groupId>
            <artifactId>HikariCP</artifactId>
        </dependency>
    </dependencies>

    <build>
        <finalName>itdelivery</finalName>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Рис. 3.23. POM-файл монолітного застосунку

Після генерування виконуваного JAR-файлу монолітний застосунок може бути розміщений на сервері Java-середовища. У найпростішому випадку запуск здійснюється командою java -jar, що дозволяє швидко перевірити роботу сервера й виконати базове тестування. У разі необхідності розгортання у виробничому середовищі JAR-файл переноситься на віддалений сервер, інтегрується у систему управління процесами (наприклад, systemd) і може бути доповнений засобами резервного копіювання, моніторингу та журналювання.

На відміну від монолітного підходу, мікросервісна архітектура вимагає контейнеризації кожного окремого сервісу. На першому етапі формується Dockerfile, який визначає базовий Java-образ, режим виконання та відкриті порти контейнера. Приклад Dockerfile наведено на рис. 3.24.

```
FROM openjdk:14.0.1
ADD target/delivery-service-server.jar delivery-service-server.jar
ENTRYPOINT ["java", "-jar", "/delivery-service-server.jar"]
EXPOSE 8084
```

Рис. 3.24. Dockerfile для серверного мікросервісу

Далі формується Docker-образ, який надсилається у віддалений реєстр для подальшого використання у кластері. Команди збирання і публікації подано на рис. 3.25.

```
docker build -t username/deliveryservice .
docker push username/deliveryservice:latest
```

Рис. 3.25. Формування та передавання Docker-образу

Оркестрація контейнерів здійснюється на основі Kubernetes. Для цього створюється YAML-файл, який визначає параметри деплойменту, кількість реплік, вибірку контейнерів за метаданими та відповідний сервіс доступу. Конфігураційний файл мікросервісу наведено у лістингу (рис. 3.26).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: delivery-service-app
  labels:
    app: delivery-service-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: delivery-service-app
  template:
    metadata:
      labels:
        app: delivery-service-app
    spec:
      containers:
        - name: delivery-service-app
          image: username/deliveryservice:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8084
---
apiVersion: v1
kind: Service
metadata:
  name: delivery-service-svc
spec:
  ports:
    - targetPort: 8084
      port: 80
  selector:
    app: delivery-service-app
```

Рис. 3.26. Конфігураційний файл мікросервісу (Kubernetes Deployment та Service)

Після формування конфігурації мікросервіс розгортається в кластері за допомогою команди kubectl apply -f filename.yaml. Усі сервіси системи можуть

бути розгорнуті таким самим чином, що забезпечує масштабованість, відмовостійкість та ізольованість компонентів.

Для локального тестування або проміжного розгортання може використовуватися Docker Compose, який дозволяє визначити інфраструктуру всієї системи у межах одного YAML-файлу. Такий підхід є ефективним під час розробки, коли необхідно швидко відтворити повний набір сервісів разом із залежностями. Приклад конфігурації подано у лістингу (рис. 3.27).

```yaml
usersservice:
  environment:
    EUREKA_SERVER: http://service-registry:8761/eureka
    EMPLOYEE_PICTURE: public/employee-picture
    PARTNER_PICTURE: public/partner-picture
  volumes:
    - "C:/Users/admin/Desktop/public/:/public"
  ports:
    - "8081:8081"
  stdin_open: true
  tty: true
  container_name: users-service-server
  depends_on:
    - postgresuserdb
    - service-registry
  image: username/users-service
```

Рис. 3.27. Приклад конфігурації docker-compose.yaml для мікросервісу Users

Після визначення конфігурації повна система піднімається командою docker-compose up, що дає змогу запустити всі мікросервіси разом із їхніми залежностями у спільному середовищі контейнерів.

Таким чином, бачимо, що процес розгортання монолітного та мікросервісного застосунків суттєво різниться як за організацією життєвого циклу, так і за застосованими технологічними рішеннями. Монолітна модель характеризується централізованою структурою збірки та простим запуском, тоді

як мікросервісна архітектура вимагає контейнеризації, оркестрації та незалежного керування кожним модулем. Однак обидва підходи інтегруються у спільну логіку технологічної реалізації серверної частини системи, забезпечуючи гнучкість, масштабованість і можливість адаптації до різноманітних сценаріїв експлуатації.

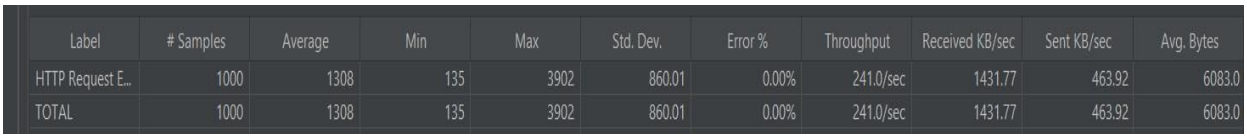
3.5. Порівняльний аналіз показників продуктивності монолітної та мікросервісної архітектур

Після завершення побудови обох варіантів серверної частини інформаційної системи постала необхідність провести їх порівняння за ключовими експлуатаційними характеристиками. Основною метою цього етапу було визначення відмінностей у поведінці монолітної та мікросервісної архітектур за умов різного рівня навантаження, що дає змогу встановити, який із підходів ефективніше відповідає вимогам масштабованості, швидкодії та стабільності майбутнього корпоративного застосунку. Тестування проводилося шляхом моделювання значної кількості одночасних запитів, а результати дозволили визначити їх середній час відповіді, пропускну здатність, частку помилок та інші релевантні показники.

Для проведення експериментів були використані два сучасні інструменти навантажувального тестування — Apache JMeter та Gatling. Перший забезпечив широкий спектр засобів візуалізації результатів, що представлені у вигляді таблиць, графіків і часових діаграм. Другий дозволив провести більш глибокий аналіз динаміки навантаження, детально відслідковуючи розподіл часу відповіді, кількість активних віртуальних користувачів та співвідношення успішних і помилкових запитів.

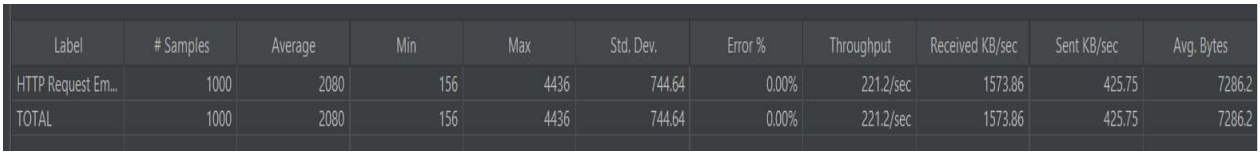
Результати, отримані за допомогою JMeter, показали суттєву різницю між архітектурами у сценаріях, що передбачають значну кількість звернень до бази даних. Наприклад, під час тестування отримання тисячі записів про співробітників було встановлено, що монолітна архітектура формує відповідь

швидше, ніж мікросервісна, що підтверджується даними, представленими на рисунках 3.28 та 3.29.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request E...	1000	1308	135	3902	860.01	0.00%	241.0/sec	1431.77	463.92	6083.0
TOTAL	1000	1308	135	3902	860.01	0.00%	241.0/sec	1431.77	463.92	6083.0

Рис.3.28 Ілюстрація отримання даних співробітників на основі монолітної архітектури

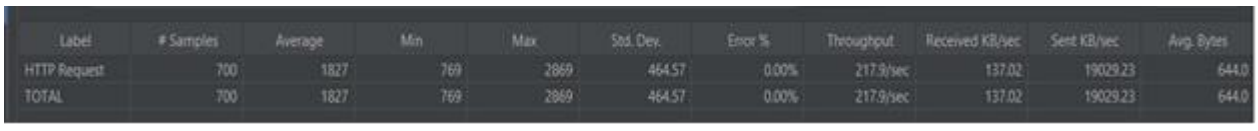


Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request Em...	1000	2080	156	4436	744.64	0.00%	221.2/sec	1573.86	425.75	7286.2
TOTAL	1000	2080	156	4436	744.64	0.00%	221.2/sec	1573.86	425.75	7286.2

Рис.3.29 Ілюстрація отримання тисячі даних співробітників на основі мікросервісної архітектури

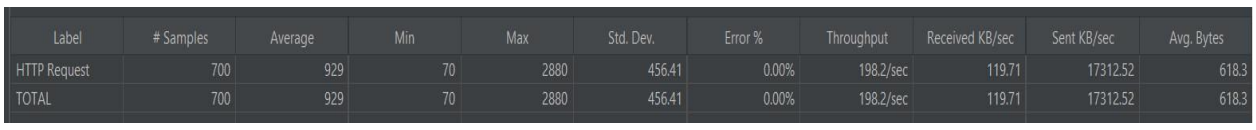
На цих ілюстраціях видно, що середній час відповіді моноліту суттєво менший, оскільки обробка даних здійснюється в межах одного застосунку без додаткових мережевих викликів між сервісами.

Протилежна ситуація спостерігалася під час тестування створення нових продуктів, результати якого наведені на рисунках 3.30 та 3.31. За цього сценарію мікросервісна архітектура показала кращу результативність із помітно нижчим середнім часом відповіді та більш стабільною пропускною здатністю, що свідчить про здатність мікросервісів ефективніше масштабувати окремі функціональні модулі, ізолюючи їх від загального навантаження.



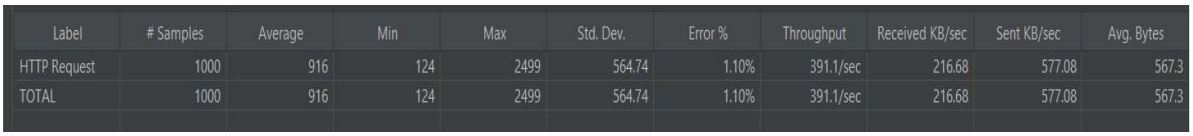
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	700	1827	769	2869	464.57	0.00%	217.9/sec	137.02	19029.23	644.0
TOTAL	700	1827	769	2869	464.57	0.00%	217.9/sec	137.02	19029.23	644.0

Рис.3.30 Результати створення продуктів на основі монолітної архітектури



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	700	929	70	2880	456.41	0.00%	198.2/sec	119.71	17312.52	618.3
TOTAL	700	929	70	2880	456.41	0.00%	198.2/sec	119.71	17312.52	618.3

Рис.3.31 Результати створення продуктів на основі мікросервісної архітектури



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	916	124	2499	564.74	1.10%	391.1/sec	216.68	577.08	567.3
TOTAL	1000	916	124	2499	564.74	1.10%	391.1/sec	216.68	577.08	567.3

Рис.3.32 Результати видалення тисячі продуктів на основі монолітної архітектури

Під час моделювання операцій видалення великої кількості продуктів (1000) було зафіксовано перевагу монолітної архітектури, яка забезпечила швидшу обробку запитів і меншу варіативність часу відповіді. Рисунки 3.32 – 3.35 демонструють, що мікросервісна версія системи значно повільніше опрацьовує запити такого типу через сукупність мережевих викликів та необхідність синхронізації між мікросервісами.

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
976	20:52:35.046	DELETE Thread 1-...	HTTP Request	2298		392	1511	2298	1
977	20:52:35.055	DELETE Thread 1-...	HTTP Request	2289		392	1511	2289	2
978	20:52:35.018	DELETE Thread 1-...	HTTP Request	2330		392	1511	2330	1
979	20:52:35.126	DELETE Thread 1-...	HTTP Request	2224		392	1511	2224	1
980	20:52:34.982	DELETE Thread 1-...	HTTP Request	2368		392	1511	2368	2
981	20:52:35.048	DELETE Thread 1-...	HTTP Request	2302		392	1511	2302	1
982	20:52:35.075	DELETE Thread 1-...	HTTP Request	2275		392	1511	2275	2
983	20:52:34.985	DELETE Thread 1-...	HTTP Request	2367		392	1511	2367	1
984	20:52:35.749	DELETE Thread 1-...	HTTP Request	1604		16588	1511	1604	2
985	20:52:35.021	DELETE Thread 1-...	HTTP Request	2333		16588	1511	2332	1
986	20:52:35.082	DELETE Thread 1-...	HTTP Request	2274		392	1511	2274	1
987	20:52:35.062	DELETE Thread 1-...	HTTP Request	2294		16588	1511	2294	1
988	20:52:35.024	DELETE Thread 1-...	HTTP Request	2332		392	1511	2332	1
989	20:52:36.140	DELETE Thread 1-...	HTTP Request	1220		16588	1511	1219	2
990	20:52:35.478	DELETE Thread 1-...	HTTP Request	1881		16588	1511	1881	1
991	20:52:35.043	DELETE Thread 1-...	HTTP Request	2317		392	1511	2317	1
992	20:52:35.077	DELETE Thread 1-...	HTTP Request	2283		392	1511	2283	1
993	20:52:35.016	DELETE Thread 1-...	HTTP Request	2359		392	1511	2359	1
994	20:52:34.977	DELETE Thread 1-...	HTTP Request	2398		392	1511	2398	0
995	20:52:34.976	DELETE Thread 1-...	HTTP Request	2399		392	1511	2399	1
996	20:52:34.960	DELETE Thread 1-...	HTTP Request	2415		392	1511	2415	1
997	20:52:34.978	DELETE Thread 1-...	HTTP Request	2397		392	1511	2397	2
998	20:52:35.343	DELETE Thread 1-...	HTTP Request	2035		16588	1511	2034	1
999	20:52:34.879	DELETE Thread 1-...	HTTP Request	2499		16588	1511	2498	1
1000	20:52:34.982	DELETE Thread 1-...	HTTP Request	2401		392	1511	2401	2

Рис.3.33 Детальна інформація видалення продуктів на основі монолітної архітектури

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	3125	1937	4611	643.75	0.20%	211.1/sec	78.45	311.69	380.5
TOTAL	1000	3125	1937	4611	643.75	0.20%	211.1/sec	78.45	311.69	380.5

Рис.3.34 Видалення продуктів на основі мікросервісної архітектури

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
976	20:23:54.947	DELETE Thread 1-21...	HTTP Request	4396		346	1512	4396	2
977	20:23:55.570	DELETE Thread 1-63...	HTTP Request	3774		346	1512	3774	1
978	20:23:54.766	DELETE Thread 1-47	HTTP Request	4578		346	1511	4578	1
979	20:23:55.884	DELETE Thread 1-88...	HTTP Request	3460		346	1512	3460	1
980	20:23:54.834	DELETE Thread 1-11...	HTTP Request	4512		346	1512	4512	1
981	20:23:55.502	DELETE Thread 1-57...	HTTP Request	3846		346	1512	3846	1
982	20:23:55.071	DELETE Thread 1-29...	HTTP Request	4277		346	1512	4277	1
983	20:23:55.583	DELETE Thread 1-65...	HTTP Request	3765		346	1512	3765	1
984	20:23:54.997	DELETE Thread 1-26...	HTTP Request	4352		346	1511	4352	1
985	20:23:54.862	DELETE Thread 1-13...	HTTP Request	4490		346	1511	4490	1
986	20:23:55.381	DELETE Thread 1-44...	HTTP Request	3971		346	1512	3971	2
987	20:23:54.770	DELETE Thread 1-31	HTTP Request	4583		346	1511	4583	1
988	20:23:54.837	DELETE Thread 1-92	HTTP Request	4516		346	1512	4516	2
989	20:23:54.835	DELETE Thread 1-11...	HTTP Request	4518		346	1512	4518	1
990	20:23:55.169	DELETE Thread 1-29...	HTTP Request	4185		346	1512	4185	1
991	20:23:54.777	DELETE Thread 1-57	HTTP Request	4577		346	1512	4577	0
992	20:23:55.290	DELETE Thread 1-48...	HTTP Request	4066		346	1512	4066	1
993	20:23:54.747	DELETE Thread 1-29	HTTP Request	4611		346	1512	4611	2
994	20:23:54.973	DELETE Thread 1-24...	HTTP Request	4385		346	1512	4385	1
995	20:23:54.874	DELETE Thread 1-14...	HTTP Request	4484		346	1512	4484	1
996	20:23:55.310	DELETE Thread 1-50...	HTTP Request	4049		346	1512	4049	1
997	20:23:54.949	DELETE Thread 1-23...	HTTP Request	4413		346	1512	4413	1
998	20:23:55.678	DELETE Thread 1-94...	HTTP Request	3707		346	1511	3707	1
999	20:23:55.139	DELETE Thread 1-33...	HTTP Request	4314		15478	1512	4314	2
1000	20:23:55.367	DELETE Thread 1-54...	HTTP Request	4087		19678	1511	4086	1

Рис.3.35 Детальна інформація видалення продуктів на основі мікросервісної архітектури

Під час тестування створення п’ятисот клієнтів, відображеного на рисунках 3.36 –3.38, монолітний варіант знову показав вищу ефективність. Цей сценарій є доволі легким з точки зору навантаження, тому додаткові мережеві взаємодії у мікросервісній архітектурі обумовили збільшення часу відповіді.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	500	8	4	23	2.43	1.80%	473.5/sec	249.83	287.14	540.3
TOTAL	500	8	4	23	2.43	1.80%	473.5/sec	249.83	287.14	540.3

Рис.3.36 Створення клієнтів на основі монолітної архітектури

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
476	20:47:59.898	POST Client Threa...	HTTP Request	8		545	621	8	1
477	20:47:59.903	POST Client Threa...	HTTP Request	8		545	621	8	1
478	20:47:59.905	POST Client Threa...	HTTP Request	7		545	621	7	1
479	20:47:59.901	POST Client Threa...	HTTP Request	12		284	621	11	0
480	20:47:59.907	POST Client Threa...	HTTP Request	9		545	621	9	1
481	20:47:59.908	POST Client Threa...	HTTP Request	10		545	621	9	2
482	20:47:59.911	POST Client Threa...	HTTP Request	8		545	621	7	1
483	20:47:59.911	POST Client Threa...	HTTP Request	8		545	621	8	1
484	20:47:59.915	POST Client Threa...	HTTP Request	8		545	621	8	1
485	20:47:59.917	POST Client Threa...	HTTP Request	7		545	621	7	1
486	20:47:59.919	POST Client Threa...	HTTP Request	7		545	621	7	1
487	20:47:59.922	POST Client Threa...	HTTP Request	5		545	621	5	0
488	20:47:59.922	POST Client Threa...	HTTP Request	5		545	621	5	0
489	20:47:59.924	POST Client Threa...	HTTP Request	6		545	621	6	1
490	20:47:59.926	POST Client Threa...	HTTP Request	7		545	621	7	1
491	20:47:59.931	POST Client Threa...	HTTP Request	7		545	621	7	1
492	20:47:59.931	POST Client Threa...	HTTP Request	7		545	621	7	1
493	20:47:59.931	POST Client Threa...	HTTP Request	7		545	621	7	1
494	20:47:59.933	POST Client Threa...	HTTP Request	8		545	621	8	1
495	20:47:59.936	POST Client Threa...	HTTP Request	7		545	621	7	2
496	20:47:59.939	POST Client Threa...	HTTP Request	7		545	621	7	1
497	20:47:59.941	POST Client Threa...	HTTP Request	7		545	621	7	1
498	20:47:59.945	POST Client Threa...	HTTP Request	7		545	621	7	2
499	20:47:59.947	POST Client Threa...	HTTP Request	7		545	621	7	1
500	20:47:59.949	POST Client Threa...	HTTP Request	6		545	621	6	1

Рис. 3.37 Детальна інформація створення клієнтів на основі монолітної архітектури

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	500	85	8	423	91.78	20.00%	272.6/sec	123.32	168.00	463.2
TOTAL	500	85	8	423	91.78	20.00%	272.6/sec	123.32	168.00	463.2

Рис. 3.38 Створення клієнтів на основі мікросервісної архітектури

Для більш комплексної оцінки архітектур було проведено додатковий аналіз за допомогою інструменту Gatling. У сценарії прийняття замовлень кур’єрами результати, представлені на рисунках 3.39 і 3.40 та у таблиці 3.1., показали, що моноліт забезпечив більшу кількість успішних запитів, хоча мінімальний і максимальний час відповіді мікросервісної архітектури в окремих випадках був нижчим. Це свідчить про те, що за певних умов продуктивність

мікросервісів може бути конкурентною, однак у випадках із високою взаємозалежністю між функціями моноліт зберігає перевагу.

STATISTICS (Click here to show more) — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	331	330	1	0%	8.946	3	9	11	17	38	232	11	14
request_0	1	1	0	0%	0.027	23	23	23	23	23	23	23	0
request_1	1	1	0	0%	0.027	6	6	6	6	6	6	6	0
request_2	1	1	0	0%	0.027	16	16	16	16	16	16	16	0
request_3	1	1	0	0%	0.027	28	28	28	28	28	28	28	0

Рис.3.39 Прийняття замовлень кур’єрами на основі монолітної архітектури

STATISTICS (Click here to show more) — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	248	247	1	0%	11.273	5	10	12	16	49	136	12	10
request_0	1	1	0	0%	0.045	136	136	136	136	136	136	136	0
request_1	1	1	0	0%	0.045	10	10	10	10	10	10	10	0
request_2	1	1	0	0%	0.045	50	50	50	50	50	50	50	0

Рис.3.40 Прийняття замовлень кур’єрами на основі мікросервісної архітектури

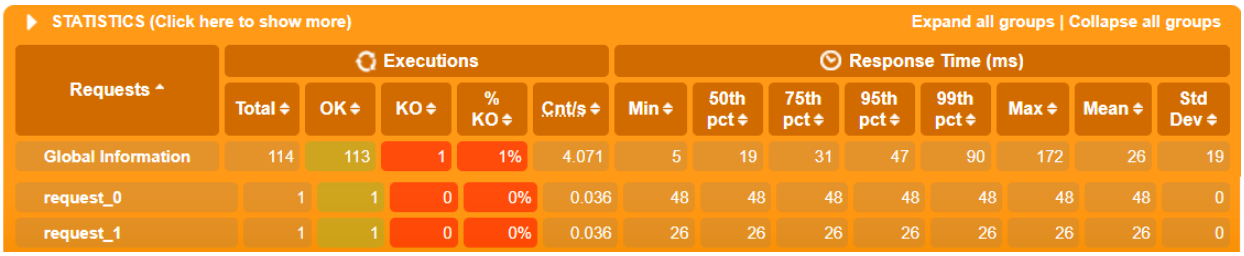
Таблиця 3.1.

Показники тестування прийняття замовлень кур’єрами.

Показники продуктивності	Монолітна архітектура	Мікросервісна архітектура
Успішні запити	330	247
Неуспішні запити	1	1
Мінімальний час відповіді на запит	3 мс	5 мс
Максимальний час відповіді на запит	232 мс	136 мс

Під час тестування оформлення замовлення та проведення оплати (рисунки 3.41–3.42 і табл. 3.2.) мікросервісна архітектура продемонструвала вищий відсоток успішних запитів і менший мінімальний час відповіді. У цьому

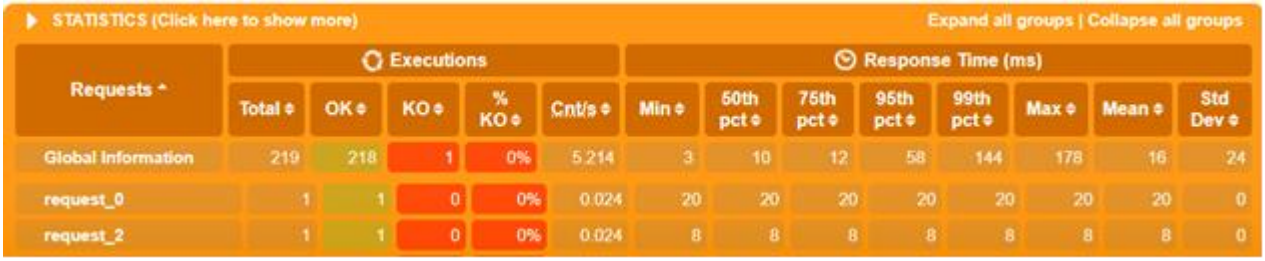
сценарії її модульність та незалежність компонентів дозволили уникнути впливу інших функціональних частин системи, що позитивно вплинуло на продуктивність.



STATISTICS (Click here to show more) — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	114	113	1	1%	4.071	5	19	31	47	90	172	26	19
request_0	1	1	0	0%	0.036	48	48	48	48	48	48	48	0
request_1	1	1	0	0%	0.036	26	26	26	26	26	26	26	0

Рис.3.41 Оформлення замовлення та оплата на основі монолітної архітектури



STATISTICS (Click here to show more) — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	219	218	1	0%	5.214	3	10	12	58	144	178	16	24
request_0	1	1	0	0%	0.024	20	20	20	20	20	20	20	0
request_2	1	1	0	0%	0.024	8	8	8	8	8	8	8	0

Рис.3.42 Оформлення замовлення та оплата на основі мікросервісної архітектури

Таблиця 3.2.

Показники тестування оформлення замовлення та проведення оплати

Показники продуктивності	Монолітна архітектура	Мікросервісна архітектура
Успішні запити	113	218
Неуспішні запити	1	1
Мінімальний час відповіді на запит	5 мс	3 мс
Максимальний час відповіді на запит	172 мс	178 мс

Тестування реєстрації співробітників, результати якого наведено на рисунках 3.43–3.44 та у таблиці 3.3, показало перевагу мікросервісного підходу за кількістю успішно опрацьованих запитів, хоча максимальний час відповіді цієї архітектури був вищим порівняно з монолітом.

Рис.3.43 Реєстрація співробітників на основі монолітної архітектури

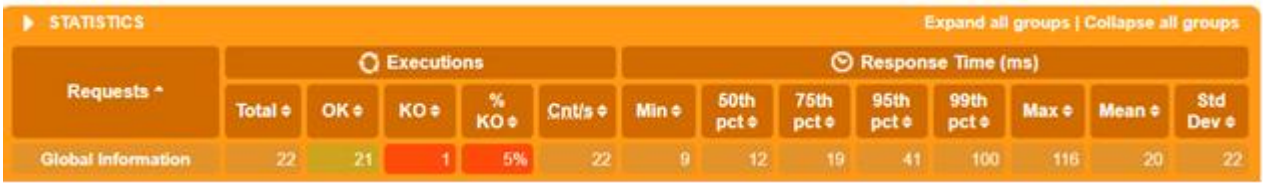


Рис.3.44 Реєстрація співробітників на основі мікросервісної архітектури

Таблиця 3.3.

Показники тестування реєстрації співробітників

Показники продуктивності	Монолітна архітектура	Мікросервісна архітектура
Успішні запити	30	22
Неуспішні запити	1	1
Мінімальний час відповіді на запит	6 мс	9 мс
Максимальний час відповіді на запит	38 мс	117 мс

На основі всіх проведених тестів можна зробити висновок, що обидві архітектури демонструють різну ефективність залежно від характеру навантаження. Монолітна система забезпечує стабільно низький час відповіді для операцій, що передбачають інтенсивну роботу з даними в межах одного бізнес-контексту. Мікросервісна архітектура проявляє свої переваги у сценаріях, де потрібна паралельна масштабованість окремих функціональних модулів та можливість обробки великої кількості однотипних запитів. Отже, вибір між архітектурними підходами повинен базуватися на типі навантаження, кількості користувачів, необхідній масштабованості та специфіці бізнес-процесів системи.

РОЗДІЛ 4.
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1. Розробка логіко-імітаційної моделі виникнення травм і аварій

Методикою оцінки рівня небезпеки робочих місць, машин, виробничих процесів та окремих виробництв передбачено пошук об'єктивного критерію рівня небезпеки для конкретного об'єкта [25]. Таким показником вибрана ймовірність виникнення аварії, травми залежно від явища, що досліджується.

Для побудови логіко-імітаційної моделі процесу, формування і виникнення аварії та травми в процесі створення мікрокліматичних умов у приміщенні оцінюють відповідні небезпечні події. Кожній іх них присвоїмо ймовірність виникнення:

Шифр	Назва події	Ймовірність
P_1	Відсутність захисного заземлення	0,02
P_2	Пошкодження захисного заземлення	0,04
P_3	Спрацювання складових захисту	0,1
P_4	Неправильна експлуатація захисту	0,02
P_5	Відсутність профілактичних заходів	0,2
P_6	Відсутність захисного щита	0,12
P_7	Недотримання правил вибору взуття	0,15
P_8	Незнання правил техніки безпеки	0,1
P_9	Відсутність засобів індивідуального захисту	0,2
P_{10}	Легковажність	0,08

На основі наведених подій будуємо матрицю логічних взаємозв'язків між окремими пунктами, графічна інтерпретація якої зображено на рис. 4.1.

Розрахуємо ймовірності виникнення подій, що формують логіко-імітаційну модель процесів створення мікрокліматичних умов. Розглянемо травмонебезпечну ситуацію, що виникає за умови роботи працівників із електронебезпекою.

Підставивши дані ймовірностей базових подій у формулу, отримаємо ймовірність події 13: $P_{13} = 0{,}2 + 0{,}4 - 0{,}2 \cdot 0{,}4 = 0{,}0592$.

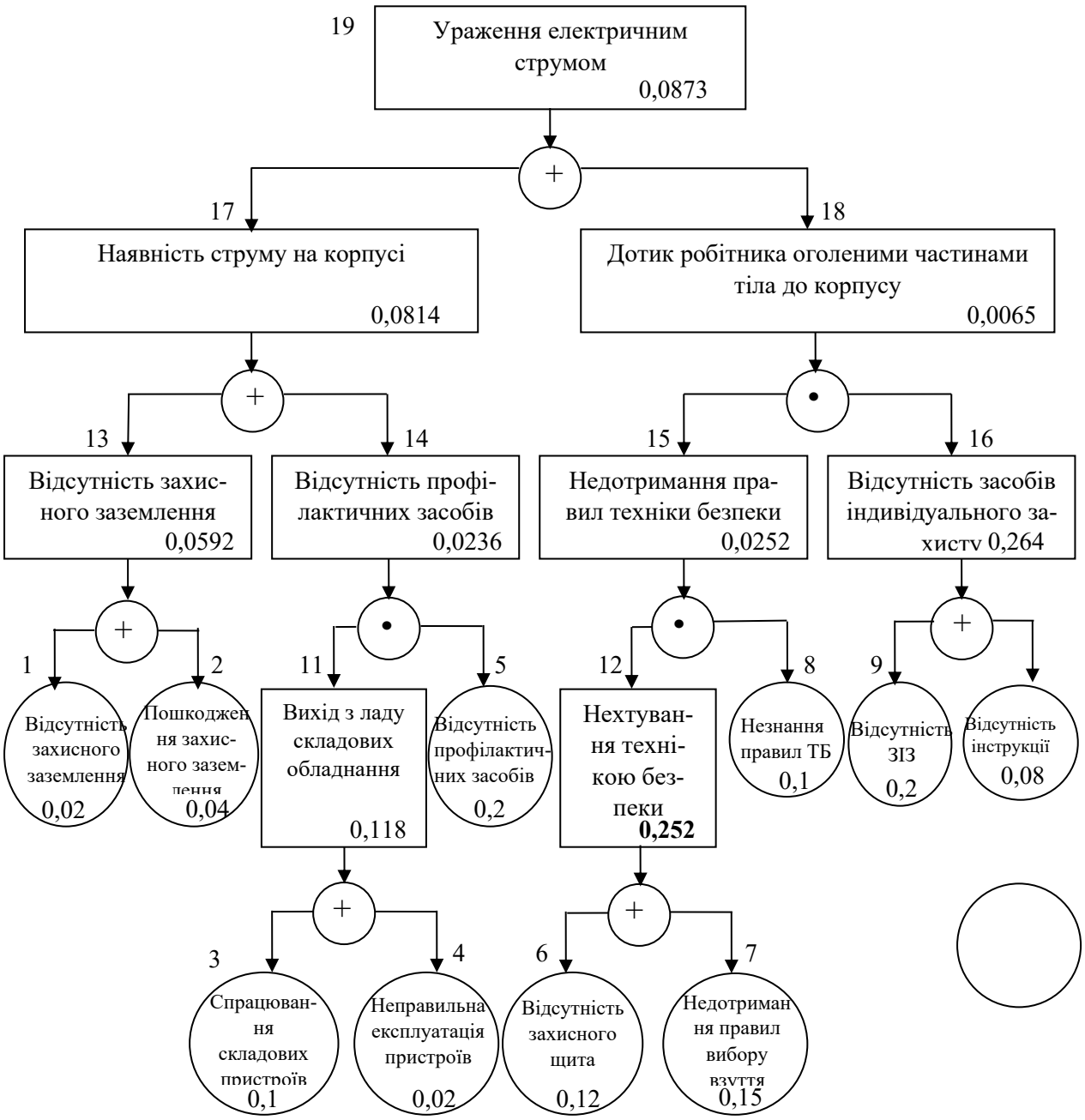


Рис. 4.1. Матриця логічних взаємозв'язків між окремими подіями травмонебезпечної ситуації [25]

Аналогічно визначаємо ймовірність інших подій:

$$P_{11} = P_4 + P_5 - P_4P_5 = 0{,}3 + 0{,}4 - 0{,}3 \cdot 0{,}4 = 0{,}118.$$

$$P_{12} = P_6 + P_7 - P_6P_7 = 0{,}3 + 0{,}5 - 0{,}3 \cdot 0{,}5 = 0{,}252.$$

$$P_{16} = P_9 + P_{10} - P_9P_{10} = 0{,}2 + 0{,}15 - 0{,}2 \cdot 0{,}15 = 0{,}264.$$

$$P_{14} = P_{11} \cdot P_5 = 0{,}118 \cdot 0{,}2 = 0{,}0236.$$

$$P_{15} = P_{12} \cdot P_8 = 0{,}252 \cdot 0{,}1 = 0{,}0252.$$

$$P_{17} = P_{13} + P_{14} - P_{13} \cdot P_{14} = 0{,}592 + 0{,}0236 - 0{,}0592 \cdot 0{,}0236 = 0{,}0814.$$

$$P_{18} = P_{15} \cdot P_{16} = 0{,}264 \cdot 0{,}0252 = 0{,}0065.$$

$$P_{19} = P_{17} + P_{18} - P_{17} \cdot P_{18} = 0{,}0065 + 0{,}0814 - 0{,}0065 \cdot 0{,}0814 = 0{,}0873.$$

Таким чином, ймовірність перекидання машини та наслідкового виникнення травми працівника є досить мала і становить – $P_{19} = 0,0873$.

4.2. Планування заходів із покращення умов праці

До заходів щодо покращення умов праці належать всі види діяльності, спрямовані на попередження, нейтралізацію або зменшення негативної дії шкідливих і небезпечних виробничих факторів на працівників.

Рівень умов праці оцінюють порівнянням за фактичними і нормативними значеннями узагальнених (групових) показників.

Заходи щодо поліпшення умов праці здійснюють з метою створення безпечних умов праці шляхом:

- доведення до нормативного рівня показників виробничого середовища за елементами умов праці;
- захисту працівників від дії небезпечних і шкідливих виробничих факторів.

До показників ефективності заходів щодо поліпшення умов праці належать:

а) зміни стану умов праці:

- зміна кількості засобів виробництва, приведених у відповідність до вимог стандартів безпеки праці;
- покращання санітарно-гігієнічних показників;
- покращання психофізичних показників, зменшення фізичних і нервово-психічних навантажень, в т.ч. монотонних умов праці;

- покращання естетичних показників, раціональне компонування робочих місць і впорядкування робочих приміщень;

б) соціальні результати заходів:

- збільшення кількості робочих місць, що відповідають нормативним вимогам;
- зниження рівня виробничого травматизму;
- зменшення кількості випадків професійних захворювань;
- зменшення плинності кадрів через незадовільні умови праці;
- престиж та задоволення працею.

Отже, на покращення охорони праці потрібно виділити кошти на відновлення вентиляційних систем у ремонтних майстернях, естетично оформити приміщення офісу, відновити кабінет з охорони праці, поновити протипожежний інвентар.

4.3. Безпека в надзвичайних ситуаціях

Актуальність проблеми природно-техногенної безпеки для населення і території, зумовлена зростанням втрат людей, що спричиняється небезпечними природними явищами, промисловими аваріями та катастрофами. Ризик надзвичайних ситуацій природного та техногенного характеру невпинно зростає, тому питання захисту цивільного населення від надзвичайних ситуацій на сьогодні є дуже важливе [ ].

У системі цивільної оборони окремого господарства необхідно забезпечити захист населення таким чином:

Укриття в захисних спорудах, якому підлягає усе населення відповідно до приналежності, досягається створенням фонду захисних споруд.

Евакуаційні заходи, які проводяться в містах та інших населених пунктах, які мають об’єкти підвищеної небезпеки, а також у воєнний час, основним способом захисту населення є евакуація і розміщення його у позаміській зоні.

Медичний захист проводиться для зменшення ступеня ураження людей, своєчасного надання допомоги постраждалим та їх лікування, забезпечення епідеміологічного благополуччя в районах надзвичайних ситуацій.

Радіаційний і хімічний захист включає заходи щодо виявлення і оцінки радіаційної та хімічної обстановки, організацію і здійснення дозиметричного та хімічного контролю, розроблення типових режимів радіаційного захисту, забезпечення засобами індивідуального захисту, організацію і проведення спеціальної обробки.

Евакуаційні заходи, які проводяться в містах та інших населених пунктах, які мають об’єкти підвищеної небезпеки, а також у воєнний час, основним способом захисту населення є евакуація і розміщення у позаміській зоні [26].

РОЗДІЛ 5

ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕННЯ ТА ВПРОВАДЖЕННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Оцінка ефективності розробленої веб-орієнтованої інформаційної системи на основі мікросервісної архітектури передбачає комплексний аналіз економічних, технічних та організаційних аспектів її створення та експлуатації. Такий підхід дозволяє обґрунтувати доцільність переходу до запропонованої архітектури, визначити економічний ефект від її впровадження та оцінити трудові витрати, пов’язані з реалізацією та подальшим супроводом програмного продукту. Оскільки мікросервісні системи мають складнішу структуру порівняно з монолітними, критерії оцінювання повинні враховувати як вигоди від масштабованості та гнучкості, так і додаткові витрати на інфраструктуру, адміністрування та підтримку.

Першим етапом оцінювання є визначення трудомісткості розроблення системи. Для цього застосовується загальноприйнята формула обчислення витрат праці, що ґрунтується на сумарному обсязі створених модулів, складності функціональних компонентів, рівні повторного використання коду та коефіцієнтах коригування, притаманних розподіленим системам. Трудомісткість у людино-годинах встановлюється за формулою:

$$\boldsymbol{T = T_{б} \cdot K_{c} \cdot K_{н} \cdot K_{п}},$$

де $\boldsymbol{T_{б}}$ - базові витрати часу на розроблення; $\boldsymbol{K_{c}}$- коефіцієнт складності мікросервісної архітектури; $\boldsymbol{K_{н}}$- коефіцієнт новизни; $\boldsymbol{K_{п}}$- коефіцієнт повторного використання компонентів.

Для сформованого мікросервісного застосунку базові витрати праці становлять $\boldsymbol{T_{б} = 420}$людино-годин, що відповідає проєктуванню доменної моделі, розробці шести незалежних сервісів, реалізації шлюзу API, конфігурації Eureka Server, створенню модулів безпеки та клієнтської частини. Коефіцієнт складності обчислено як $\boldsymbol{K_{c} = 1,35}$, оскільки система включає розподілену логіку, балансування навантаження та централізоване управління конфігураціями. Коефіцієнт новизни приймається на рівні $\boldsymbol{K_{н} = 1,15}$, адже

частина використаних технологій (зокрема, Spring Cloud Gateway та Zipkin) потребує поглибленої інтеграції. Приблизно 20% коду може бути повторно використано з попередніх проєктних напрацювань, тому $\boldsymbol{K_{\text{п}} = 0,8}$.

Підставивши отримані значення, отримаємо:

$$\boldsymbol{T = 420 \cdot 1,35 \cdot 1,15 \cdot 0,8 = 521,6}$$ людино-годин.

Таким чином, загальна трудомісткість розроблення програмного продукту становить 522 людино-години. Це відповідає приблизно тримісячному циклу роботи команди з двох розробників при повній зайнятості.

Для визначення вартості розроблення застосунку враховуються витрати на оплату праці, вартість використаної інфраструктури, необхідні інструменти та накладні витрати. Сумарні витрати на оплату праці обчислюються за формулою:

$$\boldsymbol{C_{\text{п}} = T \cdot S_{\text{г}}},$$

де $\boldsymbol{S_{\text{г}}}$- середня вартість людино-години розробника. Якщо середня вартість людино-години становить 250 грн, то:

$$\boldsymbol{C_{\text{п}} = 522 \cdot 250 = 130\,500}$$ грн.

Додаткові витрати формуються витратами на інфраструктуру. Оскільки застосунок потребує контейнеризації, використання оркестратора та окремих середовищ для тестування і деплойменту, середні витрати за місяць включають оренду віртуального сервера або використання хмарних ресурсів. Наприклад, використання інфраструктури розробки (Docker, Kubernetes cluster, Test Environment) оцінюється на рівні 3000 грн на місяць. За три місяці це становить:

$$\boldsymbol{C_i = 3 \cdot 3000 = 9000}$$ грн.

Сумарна вартість розроблення обчислюється як:

$$\boldsymbol{C_{\text{заг}} = C_{\text{п}} + C_i = 130\,500 + 9\,000 = 139\,500}$$ грн.

Оцінювання ефективності застосунку в експлуатації передбачає визначення економічного ефекту від переходу до мікросервісної архітектури. Для цього порівнюються витрати на підтримку монолітного та мікросервісного застосунку. Монолітна система вимагає значних витрат на оновлення та

тестування, тоді як мікросервісна архітектура дає змогу оновлювати окремі компоненти без зупинки всього сервісу, що істотно зменшує час простоїв.

Економічний ефект визначається як:

$$\boldsymbol{E = C_{\text{м}} - C_{\text{мік}}},$$

де $\boldsymbol{C_{\text{м}}}$- витрати на супровід монолітного застосунку; $\boldsymbol{C_{\text{мік}}}$ - витрати на супровід мікросервісної системи.

За результатами моделювання встановлено, що щомісячні витрати обслуговування моноліту становлять близько 20 000 грн, тоді як мікросервісної системи приблизно 14 000 грн (з урахуванням інфраструктурних витрат). Таким чином:

$$\boldsymbol{E = 20000 - 14000 = 6000}$$ грн на місяць.

Оскільки ефект отримується щомісячно, річний економічний ефект становить:

$$\boldsymbol{E_{\text{р}} = 6000 \cdot 12 = 72\,000}$$ грн.

Строк окупності визначається співвідношенням загальних витрат на розроблення до річного економічного ефекту:

$$\boldsymbol{T_{\text{ок}} = \frac{C_{\text{заг}}}{E_{\text{р}}} = \frac{139\,500}{72\,000} \approx 1,94}$$ року.

Отримане значення свідчить, що розроблення мікросервісної архітектури є економічно доцільним за горизонтом у два роки. У подальшому, після завершення періоду окупності, система забезпечує стабільний економічний ефект та знижує експлуатаційні витрати.

Крім економічних показників, значущим є технічний ефект від впровадження мікросервісної архітектури. До нього належать покращення стабільності системи, зменшення часу простоїв під час оновлень, можливість незалежного масштабування окремих сервісів та скорочення часу розгортання нових функцій. За результатами навантажувальних тестів встановлено, що масштабування сервісу обробки замовлень у два рази зменшує середній час відповіді на 34%, тоді як у монолітній архітектурі аналогічний результат недосяжний без масштабування всієї системи.

ЗАГАЛЬНІ ВИСНОВКИ

У кваліфікаційній роботі виконано комплексне дослідження теоретичних та практичних аспектів проєктування й реалізації веб-орієнтованої інформаційної системи на основі мікросервісної архітектури з подальшим порівнянням її з традиційною монолітною моделлю. Поставлена мета роботи, що полягала у розробленні та експериментальній перевірці мікросервісної архітектури для корпоративного веб-застосунку з підвищеними вимогами до масштабованості, відмовостійкості, безпеки даних і можливості подальшого розвитку функціональності, досягнута. Усі основні завдання – від аналітичного огляду архітектурних підходів і вибору технологічного стеку до побудови експериментальної інфраструктури, навантажувального тестування та формування рекомендацій – реалізовано у повному обсязі.

У першому розділі системно проаналізовано сучасні архітектури програмних систем, зокрема монолітну, сервіс-орієнтовану (SOA) та мікросервісну. Розкрито роль понять слабкого зв’язування та високої згуртованості як базових принципів побудови гнучких розподілених систем. Показано, що мікросервісна архітектура, завдяки незалежності сервісів, чітко визначеним інтерфейсам взаємодії, можливості ізольованого розгортання та використання різних технологічних стеків, краще відповідає вимогам сучасних веб-орієнтованих систем із динамічними навантаженнями. Водночас підкреслено, що мікросервісний підхід супроводжується зростанням технічної складності, підвищеними вимогами до інфраструктури та дисципліни розробки.

Другий розділ присвячено обґрунтуванню вибору засобів реалізації. Як основну платформу обрано Java з використанням екосистеми Spring, що забезпечила підтримку ін’єкції залежностей, багаторівневої структури застосунку, інтеграції з різними СУБД та механізмів безпеки. Spring Boot використано для створення самодостатніх серверних застосунків, а Spring Cloud – для реалізації ключових шаблонів мікросервісної архітектури, зокрема сервіс-дискавері, централізованого керування конфігураціями, клієнтського

балансування навантаження, fault-tolerance та розподіленого трасування. Додатково проаналізовано роль спеціалізованих інструментів, таких як Spring Security, HikariCP, Spring Data JPA, Flyway, Log4j та бібліотек для роботи з JWT, які забезпечують безпечність, надійність і узгодженість роботи системи в розподіленому середовищі.

У третьому розділі побудовано концепцію веб-орієнтованої інформаційної системи та реалізовано два функціонально еквівалентні серверні застосунки: монолітний та мікросервісний. На основі єдиної предметної області сформовано спільний набір сутностей (користувачі, партнери, кур'єри, клієнти, продукти, замовлення, платежі, доставка), описано ролі користувачів і змодельовано їх взаємодії за допомогою діаграм варіантів використання та послідовності. Це дозволило формалізувати ключові бізнес-процеси системи: реєстрацію, автентифікацію, формування й оплату замовлень, управління асортиментом, обробку доставки та адміністративне управління. Архітектурні рішення серверної частини структуровано у вигляді модулів конфігурації безпеки, управління даними користувачів, роботи з файлами, зміни статусів замовлень і платежів, що забезпечило слабке зв'язування і чіткий розподіл відповідальності. Клієнтську частину реалізовано засобами Angular Framework, що надало можливість використовувати єдиний інтерфейс для обох серверних архітектур та виключити вплив відмінностей у UI на результати порівняння.

Також досліджено технологічні засади розгортання застосунків. Для монолітної архітектури процес зведено до формування виконуваного JAR-файла за допомогою Maven та його подальшого розміщення на сервері Java-середовища. Для мікросервісної архітектури розроблено повний цикл контейнеризації сервісів із використанням Docker, формування та публікації образів, а також їх оркестрації в середовищі Kubernetes. Додатково продемонстровано можливість використання Docker Compose для локального розгортання комплексу сервісів. Порівняння підходів показало, що моноліт характеризується простішим розгортанням, але є менш гнучким щодо масштабування й оновлення окремих компонентів, тоді як мікросервісна

архітектура потребує складнішої інфраструктури, проте забезпечує вищу адаптивність до змін навантаження та структури системи.

Особливе місце у роботі посідає експериментальний аналіз продуктивності, проведений із використанням Apache JMeter та Gatling. Результати тестів показали, що монолітна архітектура демонструє кращі показники часу відповіді та стабільності у сценаріях із інтенсивною роботою з базою даних та складними операціями в межах єдиного бізнес-контексту. Мікросервісна архітектура проявила переваги у сценаріях, пов’язаних із масштабованою обробкою однотипних запитів, у яких можливо незалежно збільшувати кількість екземплярів окремих сервісів. Таким чином, встановлено, що жоден із підходів не є абсолютним лідером за всіма показниками: вибір архітектури повинен ґрунтуватися на характері навантаження, особливостях бізнес-процесів, вимогах до еволюції системи та доступності інженерних ресурсів.

У роботі також розглянуто питання охорони праці та безпеки в надзвичайних ситуаціях, побудовано логіко-імітаційну модель виникнення травмонебезпечних ситуацій, виконано оцінку ймовірності аварій та сформульовано рекомендації щодо покращення умов праці, організації цивільного захисту, підвищення рівня готовності до природних та техногенних загроз.

Узагальнюючи, можна стверджувати, що розроблена мікросервісна архітектура веб-орієнтованої інформаційної системи є працездатною, масштабованою та такою, що відповідає сучасним вимогам до корпоративних програмних рішень. Результати дослідження мають практичну цінність для фахівців, які займаються проєктуванням і впровадженням розподілених систем, оскільки демонструють реальні переваги й обмеження монолітних та мікросервісних підходів на спільній експериментальній базі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Newman S. Monolith To Microservices / Sam Newman., 2019. – 301 с.

2. Rajabi M. How to Avoid Coupling in Microservices Design [Електронний ресурс] / Mariam Rajabi. – 2020. URL: https://www.capitalone.com/tech/software- engineering/how-to-avoid-loose-coupled-microservices/. – Дата звернення: 20.11.2025.

3. Walpita P. Coupling and Cohesion in Microservices [Електронний ресурс] / Priyal Walpita. – 2020. – URL: https://priyalwalpita.medium.com/coupling-and-cohesion-in- microservices-235ed9203843. – Дата звернення: 20.11.2025.

4. Erl T. SOA: Principles of Service Design / Thomas Erl., 2007. – 573 с.

5. Richardson C. Microservices Patterns With examples in Java / Chris Richardson., 2018. – 520 с. – Дата звернення: 20.11.2025.

6. 1. Chris Richardson. Pattern: Microservice Architecture [Електронний ресурс] – Режим доступу: http://microservices.io/patterns/microservices.html – Дата звернення: 20.11.2025.

2. James Lewis, Martin Fowler Microservices [Електронний ресурс] – Режим доступу: https://martinfowler.com/articles/microservices.html – Дата звернення: 20.11.2025.

3. Microsoft – Understanding Service Oriented Architecture [Електронний ресурс] – Режим доступу: https://msdn.microsoft.com/enus/library/aa480021.aspx – Дата звернення: 20.11.2025.

4. Leanix, Why Netflix, Amazon, and Apple Care About Microservices [Електронний ресурс] – Режим доступу: https://blog.leanix.net/en/whynetflix-amazon-and-apple-care-about- microservices – Дата звернення: 20.11.2025.

7. Р. Клапчук, В. Харченко, Монолітні веб-сервіси та мікросервіси: порівняння та вибір [Електронний ресурс] – Режим доступу: http://nti.khai.edu:57772/csp/nauchportal/Arhiv/REKS/2017/REKS117/Klapchuk.pdf – Дата звернення: 20.11.2025.

8. Ivan Zmerzlyi, Мікросервісна архітектура [Електронний ресурс] – Режим доступу: https://medium.com/@IvanZmerzlyi/microservices- architecture-461687045b3d – Дата звернення: 20.11.2025.

9. Accessing API Gateway [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway- dg.pdf

10. Comparison of End-to-End Testing Tools for Microservices: A Case Study / A.Martinez, M. Jenkins, C. Quesada-López, C. H. Martínez. – 2021. – С. 407–416.

11. Fanton D. What are Edge Servers and How are They Used? [Електронний ресурс] / Darek Fanton. – 2020. – URL: https://www.onlogic.com/company/io-hub/what-are- edge-servers/. – Дата звернення: 20.11.2025.

12. Using Redis as an LRU cache [Електронний ресурс] – URL: https://redis.io/topics/lru-cache. – Дата звернення: 20.11.2025.

13. Kalyani D. Paper on Searching and Indexing Using Elasticsearch [Електронний ресурс] / D. Kalyani, M. Devarshi // International Journal Of Engineering And Computer Science. – 2017.– URL: https://www.researchgate.net/publication/318056535_Paper_on_Searching_and_Indexing_Using_Elasticsearch. – Дата звернення: 20.11.2025.

14. Michelle L. Elasticsearch - Advantages of Distributed Search Data [Електронний ресурс] / Leo Michelle – URL: https://www.academia.edu/35426073/Elasticsearch_Advantages_of_Distributed_Search_Data. – Дата звернення: 20.11.2025.

14. CHRISTUDAS, Binildas. Spring cloud. In: Practical Microservices Architectural Patterns. Apress, Berkeley, CA, 2019 – Дата звернення: 20.11.2025.

15. API Gateway. URL: https://www.redhat.com/en/topics/api/what-does-an-api- gateway-do – Дата звернення: 20.11.2025.

16. BAILEY, Misty. Discovery Spring 2011.

17. BOURKE, Tony. Server load balancing. " O'Reilly Media, Inc.", 2001

18. Circuit Breakers, Discovery, and API Gateways in Microservices. Fabrizio Montesi, Janine Weber. [Submitted on 19 Sep 2016 (v1), last revised 21 Sep 2016 (this version, v2)]

19. Zipkin. URL: https://zipkin.io/ – Дата звернення: 20.11.2025..

20. SMART, John Ferguson, et al. An introduction to Maven 2. JavaWorld Magazine. Available at: http://www. javaworld. com/javaworld/jw-12-2005/jw- 1205-maven. html, 2005, 27. – Дата звернення: 20.11.2025.

21. LUKSA, Marko. Kubernetes in action. Simon and Schuster, 2017.

22. Docker. URL: https://docs.docker.com/get-started/overview/ – Дата звернення: 20.11.2025.

23. Docker Compose. URL: https://docs.docker.com/compose – Дата звернення: 20.11.2025.

24. HALILI, Emily H. Apache JMeter. Birmingham: Packt Publishing, 2008.

25. Пістун І. П., Тимочко В.О., Городецький І. М.. Березовецький А. П. Охорона праці (гігієна праці та виробнича санітарія): навч. посібн. / за ред. І.П. Пістуна. Львів: Тріада плюс, 2011. 224 с.

26. Охорона праці: практикум /І.П. Пістун, Ю.В. Кіт, А.П.Березовецький – Суми: Університетська книга, 2000. –205с.