Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

Розробка системи мікросервісів з використанням Apache Kafka

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення» 121

Керівник курсової роботи

старший викладач

Борозенний С.О.

_____

“____” _____ 2022 р.

Виконав студент БП ІПЗ-4

Круковська Я.В.

“____” _____ 2022 р

Київ 2022

ЗМІСТ

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри мультимед. систем,

Канд. ф.-м.н.

_____О. П. Жежерун

(підпис)

„____”_____2022 р

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Круковській Яні факультету інформатики 4 курсу
ТЕМА: Розробка системи мікросервісів з використанням Apache Kafka

Вихідні дані:

- Система сервісів, розроблена з використанням фреймворку Spring Boot та брокера повідомлень Apache Kafka

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
1 Огляд предметної області
2 Проектування та розробка
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2022 р. Керівник _____
(підпис)

Завдання отримав _____
(підпис)

Календарний план виконання курсової роботи

Тема: Розробка системи мікросервісів з використанням Apache Kafka

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	жовтень 2021 р.	
2.	Огляд літератури за темою роботи	листопад – грудень 2021 р.	
3.	Огляд технологій потрібних для реалізації практичної частини	січень 2022 р.	
4.	Написання теоретичної частини роботи	Січень, березень 2022 р.	
5.	Створення практичної частини роботи	березень – квітень 2022 р.	
6.	Коригування виконаної роботи	травень 2022 р.	
7.	Захист курсової роботи	травень 2022	

Студент Круковська Я.В.

Керівник Борозенний С.О.

“_____” _____

Вступ

У сучасному світі активно відбувається автоматизація багатьох сфер життєдіяльності: медицини, освіти, управління державою, тощо. Програмні рішення пройшли шлях від застосування у великих корпораціях та органах влади до малого бізнесу. Кількість користувачів онлайн рішеннями щороку зростає, тож стає важливою можливість створення надійних легко масштабованих розподілених систем. Для їх побудови існує чимало варіантів архітектури застосунків.

Застосунки пройшли шлях від монолітів, які важко масштабуються, не мають засобів для забезпечення відмово стійкості та дуже складні для додавання нового функціоналу або зміни існуючого, до мікросервісної архітектури, яка позбавлена цих недоліків. Проте це не означає, що остання є оптимальним рішенням для кожного нового застосунку. Розуміння переваг та недоліків кожної архітектури є важливим для побудови системи, яка відповідає вищезгаданим вимогам.

Метою роботи є виокремлення одних з найбільш поширених видів архітектури застосунків, їх опис, визначення переваг та недоліків кожної. На основі мікросервісної архітектури розглянути типи комунікацій між розподіленими сервісами і практично перевірити їх на прикладі власної реалізації системи автоматизації піцерії.

Текстова частина роботи складається з двох розділів. Перший з них логічно поділений на три частини. Перший підрозділ описує існуючі види архітектури. Другий підрозділ містить теоретичні відомості про типи взаємодій у розподілених системах, а третій розглядає їх на конкретних прикладах брокерів повідомлень, а саме Apache Kafka та RabbitMQ. Для дослідження цих тем використовувалися роботи відомих розробників таких як Кріс Річардсон (Chris Richardson), Алістер Кокберн (Alistair Cockburn), Мартін Фаулер (Martin Fowler).

Практична цінність цієї частини полягає у тому, що вона може бути використана як інструкція при виборі архітектури та типу обміну даними під час програмних рішень.

Другий розділ присвячений створенню власної розподіленої системи. Описано структуру кожного сервісу, модель бази даних та інтеграцію з Apache Kafka, а також продемонстровано роботу отриманого рішення.

Практична цінність другого розділу виявляється у тому, що його можна використати як основу для побудови власного рішення.

1. Огляд існуючих рішень

1.1 Огляд типів архітектури сучасних застосунків

Для початку огляду різних типів архітектури потрібно розуміти різницю між шаром та рівнем у контексті застосунків.

Шар (layer) - означає логічно відокремлену одиницю застосунку. Наприклад: користувацький інтерфейс та база даних є окремими шарами.

Рівень (tier) - частина застосунку, яка запускається на окремій від інших інфраструктурі. Користувацький інтерфейс та база даних можуть розгортатися разом - тоді це один рівень, а можуть бути і окремо - тоді це будуть два різних рівні [1].

Архітектура застосунку описує шаблони програмування та дизайн, які необхідні для побудови застосунку. Серед найвідоміших підходів:

- N-рівнева: зазвичай використовується трирівнева архітектура – рівень даних, бізнес логіка та користувацький інтерфейс.
- Монолітна: вся функціональність застосунку знаходиться на одному рівні.
- Мікросервісна: застосунок розбитий на незалежні сервіси.
- Подійно-орієнтована: в основі застосунку лежить спілкування через події, а не традиційні запити.
- Сервісно-орієнтована: застосунок поділений на сервісі, що спілкуються між собою за допомогою сервісної шини підприємства.

У багаторівневій та монолітній усі частини застосунку тісно зв’язані, у подійно-орієнтованій та сервісно-орієнтованій слабо зв’язані, поки у мікросервісній зв'язування взагала відсутнє. Деякі з цих архітектур будуть детальніше розглянуті далі у цій роботі [2].

1.1.2. Монолітна архітектура

Монолітна архітектура - найпростіша для розуміння та реалізації архітектура. Інколи її називають традиційною. Водночас вона може бути найскладнішою, якщо необхідно розгорнути складний застосунок, оскільки всі можливі рівні - база даних, бізнес логіка, користувацький інтерфейс, тощо - розміщуються на одному рівні [3].

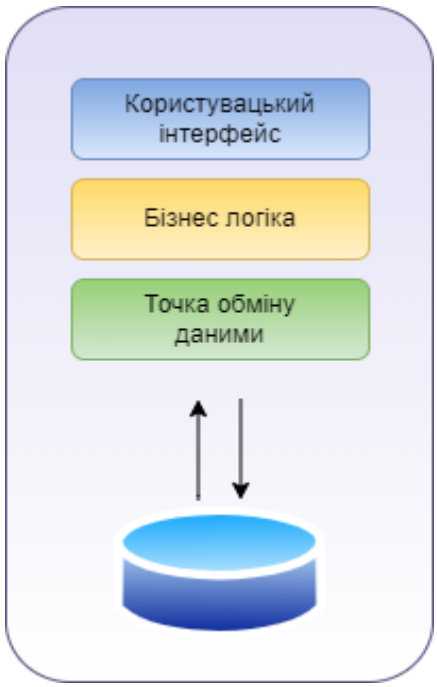

Рисунок 1.1 Приклад монолітної архітектури

Переваги моноліту:

- Проста розробка - весь код знаходиться в одному місці. Крім того значно легше забезпечити логування, кешування, перевірку швидкості роботи.
- Просте розгортання - дію потрібно повторити лише з одним артефактом: папкою або файлом.
- Просте масштабування - потрібно лише запустити додаткові копії застосунку

- Прості дебаг та тестування – оскільки застосунок є одним цілим значно легше провести end-to-end тестування [4].

Проте коли застосунок росте у розмірі, а розмір команди збільшується, з’являється значно більше проблем, ніж переваг, а вище описані переваги перетворюються у недоліки:

- Оскільки такі проекти не поділені на модулі, у коді стає важко орієнтуватися, особливо новим розробникам. Це в свою чергу уповільнює процес розробки.
- Велика кількість коду уповільнює роботу середовищ розробки, що в результаті уповільнює і саму розробку.
- Чим більше застосунок, тим повільніше він запускається, тим важче організувати безперервне розгортання.
- Якщо застосунок потребує велику кількість даних, то при масштабуванні кожна копія потребуватиме всю цю кількість, що сильно знижує ефективність [3].

1.1.3. Мікросервісна архітектура

Розподілена система - це середовище розробки, де різні компоненти розподілені в мережі між різними комп'ютерами або іншими засобами обчислень. Вони розподіляють між собою роботу, щоб виконати роботу ефективніше, ніж якби це робив один девайс [5].

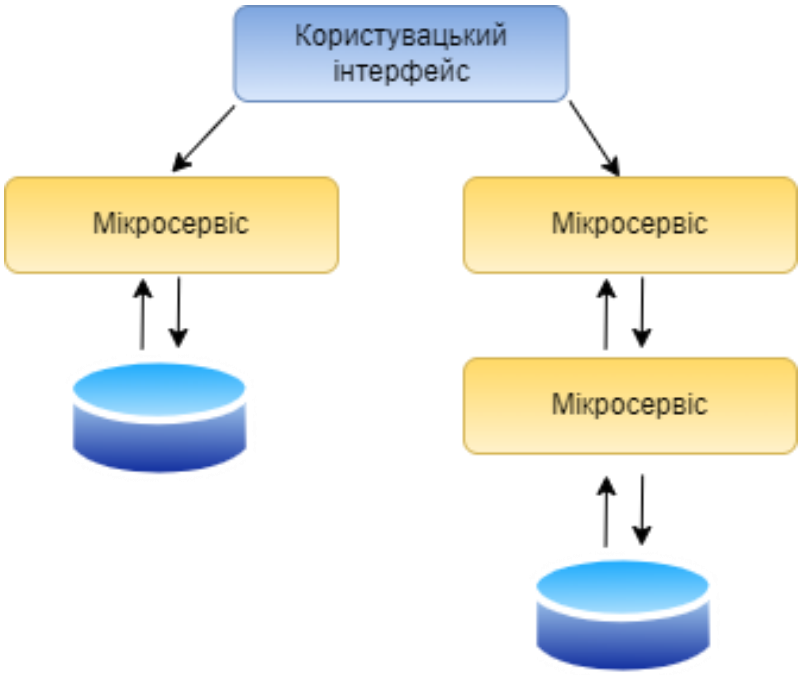

Рисунок 1.2 Приклад мікросервісної архітектури

Першу проблему, яку вирішує мікросервісна архітектура, це складність. Вона розподіляє великий монолітний застосунок на набір сервісів. У кожного з сервісів є чітко визначені межі в формі спілкування через виклик віддалених процедур чи через API на основі пересилання повідомлень. Такий сервіс значно швидше написати та простіше зрозуміти і підтримувати.

По-друге, завдяки цій архітектурі кожен сервіс може розробляти окрема команда. Розробники можуть використовувати будь-які технології, поки того дозволяють вимоги API. Це дозволяє підтримувати код актуальним та легко

заміняти технології, якщо вони стають застарими. На противагу монолітний застосунок значно важче переписати із-за його розмірів.

Третьою перевагою є можливість незалежно розгортати кожен мікросервіс. Відпадає необхідність розробникам координувати впровадження змін, що є локальними для їхнього сервісу. Розгортання змін можливе одразу після закінчення їх тестування. Тобто мікросервісна архітектура робить можливим безперервне розгортання.

Крім того, цей підхід дає можливість окремо масштабувати кожен сервіс. Кількість екземплярів сервісу залежить від того, які у нього обмеження пропускної здатності та доступності. До того ж можна використовувати обладнання, яке найкраще відповідає вимогам ресурсів [6].

Водночас у мікросервісів є багато власних недоліків.

З одного боку, мікросервісний усуває проблему складності моноліту, а з іншого, так само ускладнює роботу:

- Комунікації між сервісами можуть бути складними, особливо якщо сервісів сотні, а спілкування має бути безпечним.
- Пошук джерела якоїсь проблеми ускладнюється, коли у кожного сервісу власний набір логів.
- Поки юніт тестування стає простішим, інтеграційне тестування навпаки ускладнюється, бо розробники не можуть протестувати усю систему на одній своїй машині.

По-друге, поки у сам код можна вносити зміни, при цьому ніяк не впливаючи на зовнішні системи, ви не можете змінити API, не зачепивши усі сервіси, що використовують цей мікросервіс.

Крім того, збільшується кількість інфраструктури, яка необхідна для хостингу інфраструктури, а також кваліфікованих людей, для підтримки та обслуговування [7].

1.1.4. Подійно-орієнтована архітектура

Подійно-орієнтована архітектура – це приклад асинхронної архітектури, яка використовується для застосунків, що легко масштабуються. Його легко пристосувати і для невеликих застосунків, і для великих складних. Архітектура складається з максимально незв'язаних компонентів, які асинхронного надсилають та отримують події [8].

При такому шаблоні програмування мікросервіси реагують на зміни стану, які називаються подіями. Події можуть мати стан (щось, що описує їх, наприклад, ціна товару чи адреса доставки), або можуть бути ідентифікаторами (наприклад, сповіщення отримання чи відправлення замовлення). Події тригерять мікросервіси, які працюють разом над спільною метою, але не повинні знати нічого один про одного, окрім формату події. Хоча вони працюють разом, вони можуть виконувати різну бізнес-логіку та мати різні власні вихідні події.

Подія має наступні характеристики:

- Це запис інформації про те, що сталося.
- Вона фіксує незмінний факт, який не підлягає зміні чи видаленню.
- Вона відбувається незалежно від того, чи застосовує сервіс будь-яку логіку при його споживанні.
- Її можна зберігати нескінченно довго, у великих масштабах і використовувати стільки разів, скільки необхідно [9].

В основі архітектури орієнтованої на події стоять поняття медіатора та брокера:

- **Медіатор** використовується для оркестрування багатьох дій відносно однієї події через посередника.
- **Брокер** використовується, коли є необхідність об'єднати кілька подій без участі центрального посередника.

Медіатор є кращим для варіантів для складніших ситуацій, коли для обробки події необхідні кілька кроків, бо це в свою чергу вимагає координації або оркестрування обробки подій. Топологія медіатора складається з 4 основних компонентів: черги, посередника (медіатора), каналів та процесорів.

Подія надсилає повідомлення через канал попередньої обробки подій до черги з однієї події, яка транспортує її до посередника, тим самим ініціюючи потік подій, який направляє події до відповідних процесорів. Оркестрування виконує медіатор шляхом надсилання асинхронних подій в різні канали, які будуть виконувати кожен окремий крок процесу. Процесори подій, тим часом, прослуховують канали та отримують подію від посередника. Тоді вони виконують необхідну для обробки бізнес-логіку.

Медіатор створює окрему подію обробки події, яку можуть отримати процесори при перетворенні отриманої події. Це не означає, що він виконує якусь бізнес-логіку – це лише надання інструкції з обробки для процесора. Вся логічна обробка виконується певним процесором подій, який має всю необхідну для конкретного завдання логіку програми. Інші процесори не використовуються.

Брокер найкраще підходить, якщо є простий потік подій, який не вимагає оркестрування подій.

Топологія брокера може використовувати або модель черги, або легкі теми повідомлень, або обидва варіанти, як і потій подій у медіатора. Єдиними компонентами є брокер і процесори. Брокер містить канали подій, що використовуються в потоці подій. Така модель є набагато простішою, але вона підтримує багатоканальний зв’язок. При цьому у кожного каналу є власний ідентифікатор.

У топології брокера немає черги подій або медіатора. Натомість задача отримати всі події, опублікувати та обробити події, що сигналізують про закінчення обробки, лежить на процесорах. Це зберігає слабку звязаність та

забезпечує асинхронність завдань. Коли завдання завершається, запускається зворотний виклик. Останнє повідомлення жодна інша подія не споживає, тож програма залишається відкритою для майбутніх оновлень [10].

1.1.5. Гексагональна архітектура

Гексагональна архітектура, також відома як архітектура портів та адаптерів, була запропонована у 2005 Алістером Кокберном. У своїй першій статті про неї, «Hexagonal architecture» причиною її появи він називає три проблеми, які є наслідками змішуванням коду бізнес логіки та користувацького інтерфейсу:

1) Систему не можна точно протестувати автоматизованими тестами, бо частина логіки залежить від візуальних деталей, як-от розмір поля чи розташування кнопки.
2) Та сама причина унеможливлює перехід з керованого людиною використання до керованої програмно системи.
3) Тому ж стає важко або неможливо дозволити виконання програми іншою програмою [11].

Тож першою метою гексагональної архітектури стало надання застосунку можливості бути однаково керованою користувачами, програмами, автоматизованими тестами або програмними пакетам, та ізольованої від девайсів та баз даних розробки й тестування.

Для цього було виділено три принципи архітектури:

1) Явне розділення на користувацьку сторону, бізнес логіку та серверну частину: класи з користувацької та серверної частин

- Сторона користувача: сторона, через яку відбувається взаємодія користувача з застосунком. Сюди входить код, який дозволяє цим взаємодіям відбуватися, HTTP шляхи для API, серіалізація JSON. Алістер Кокберн називає цю сторону «лівою стороною».
- Бізнес логіка: весь код, що забезпечує виконання бізнес логіки. В ідеалі експерт з домену міг би прочитати шматок коду та вказати на неточності. Центральна сторона.

- Серверна сторона: інфраструктура та код, який дозволяє застосунку працювати. Наприклад, код, що спілкується з базою даних, чи код, що оброблює HTTP запити. Права сторона.

Такий розподіл дозволяє існувати кожній стороні незалежно від інших. Це полегшує тестування та модифікацію кожної з них,

2) Залежності з користувацької сторони та серверної йдуть до бізнес логіки: бізнес логіка має бути позбавлена залежностей від інших частин. Натомість може бути клас на серверній частині, який імплементує інтерфейс з бізнес логіки.
3) Ізоляція портів та адаптерів через інтерфейси: інтерфейси у бізнес частині відіграють роль портів, а класи, що їх імплементують у серверній та користувацькій частинах є адаптерами [12].

1.2 Огляд типів комунікацій між розподіленими сервісами

У монолітній архітектурі компоненти системи взаємодіють один з одним за допомогою викликів методів або функцій на рівні мови, а мікросервісна архітектура - це якраз розподілена система, що працює на декількох машинах. Зазвичай, кожен екземпляр сервісу є процесом. Як наслідок процеси мають спілкуватися, використовуючи один з механізмів взаємодії між процесами (inter-process communication, IPC).

IPC взаємодії можна розділити за двома напрямками: кількістю процесів, що спілкуються, та синхронністю.

	Один-до-одного	**Один-до-багатьох**
Синхронний	Запит / відповідь	-
Асинхронний	Повідомлення	Публікація / підписка
	Запит / асинхронна-відповідь	Публікація /асинхронна відповідь

Кожен сервіс зазвичай використовує комбінацію цих видів взаємодії. Деяким сервісам достатньо одного механізму IPC. Для інших може знадобитися використовувати комбінацію різних видів [13].

1.1.1. Синхронний та асинхронний

- Синхронний вид зв'язку – клієнт очікує своєчасної відповіді від сервісу і може навіть бути заблокованим, поки очікує на відповідь.
- Асинхронний – клієнт не блокується під час очікування відповіді, і відповідь, якщо є, не обов'язково надсилається сервісом негайно [13].

Для синхронного спілкування використовуються протоколи HTTP та HTTPS.

Найпопулярнішим протоколом для асинхронного спілкування є AMQP – Advanced Message Queuing Protocol. В рамках нього клієнт надсилає повідомлення за допомогою системи з брокером повідомлень, як-от Apache Kafka та RabbitMQ. Тоді той, хто створює повідомлення, не очікує на відповідь.

Асинхронне спілкування може бути реалізоване у двох варіантах: один до одного та один до багатьох [14].

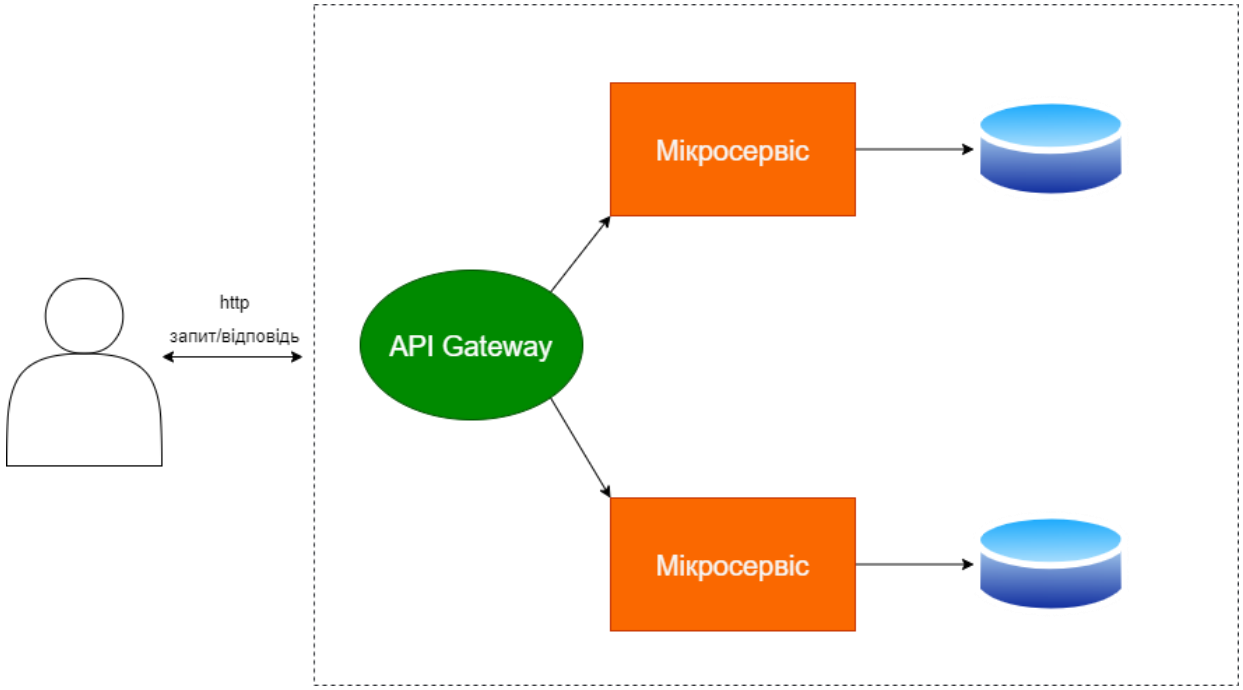

Рисунок 1.3 Приклад синхронної комунікації у мікросервісах

Один до одного (one-to-one) - за такого типу комунікації кожен запит клієнта обробляється або одним екземпляром сервісу. Є кілька прикладів такого спілкування:

- Запит / відповідь - клієнт робить запит до сервісу та чекає на відповідь. Клієнт очікує, що відповідь надійде вчасно. У застосунку на основі потоків потік, який робить запит, може навіть блокуватися під час очікування.
- Повідомлення (він же односторонній запит) – клієнт надсилає запит до сервісу, але відповіді не очікується чи надсилається.
- Запит / асинхронна відповідь – клієнт надсилає запит до сервісу, який відповідає асинхронно. Клієнт не блокується під час очікування і розроблений з припущенням, що відповідь може не надходити деякий час [13].

В основі комунікації лежить черга («queue»). Вона надає легкий буфер, що тимчасово зберігає повідомлення, та кінцеві точки зв’язку, які дозволяють програмним компонентам підключатися до черги для надсилання та отримання повідомлень. Повідомлення зазвичай невеликі і можуть бути представлені у формі запитів, відповідей, повідомлень про помилки або просто інформації. Щоб надіслати повідомлення, компонент, виробник («producer»), додає повідомлення до черги. Воно зберігається там, поки інший компонент, споживач («consumer»), не отримає повідомлення і не обробить його [15].

1.1.2. Один до багатьох

Один до багатьох (one-to-many) – кожен запит обробляється декількома екземплярами сервісу.

Існують наступні види взаємодій один до багатьох:

- Публікація / підписка – клієнт публікує сповіщення, яке обробляється зацікавленими (підписаними) сервісами, яких може і не бути.
- Публікація / публікація-асинхронна відповідь – клієнт публікує повідомлення із запитом, а потім чекає певну кількість часу на відповідь від зацікавлених сервісів [13].

Інша назва підходу - pub/sub messaging. В його основі лежить тема («topic»). Тема повідомлень, як і черга, надає легкий механізм для широкомовної передачі асинхронних сповіщень про події та кінцеві точки зв'язку, за допомогою яких програмні компоненти можуть підключатися до теми для надсилання та отримування цих повідомлень. Компонент, що створює повідомлення, видавець («publisher») просто надсилає повідомлення до теми. На відміну від черги, яка тримає пакетні повідомлення, поки їх не отримає споживач, тема повідомлень передає повідомлення або без черги, або з невеликою затримкою, та негайно надсилає їх усім підписникам («subscriber»). За умови відсутності встановлених фільтрів на повідомлення, усі компоненти, що підписалися на тему, отримають кожне повідомлення.

Підписники можуть виконувати з повідомленнями різні дії та робити це паралельно. Видавець не мусить знати, хто використовує отриману інформацію, а підписник не має знати, хто її надіслав. Це відрізняється від черг, де виробник часто знає, куди він надсилає повідомлення [16].

1.3 Огляд існуючих брокерів повідомлень

Одними з найвідоміших та найпопулярніших брокерів повідомлень є Apache Kafka та RabbitMQ. Хоч вони однаково виконують роль посередника для повідомлень між рідними застосунками, реалізовані вона двома різними підходами. Apache Kafka реалізовує одразу потокову передачу подій і традиційну обробку повідомлень через черги повідомлень, поки RabbitMQ має лише друге. Черги вже були розглянуті в розділі про різні типи комунікації між розподіленими сервісами.

Потокова передача подій (event streaming) – це практика збору даних у режимі реального часу. Джерелами подій можуть бути бази даних, датчики, мобільні пристрої, хмарні сервіси та застосунки у вигляді потоків подій. Також це тривале збереження цих потоків подій для подальшого використання та їх обробка у реальному часі, направлення потоків подій до різних технологій за необхідності. Таким чином, потокова передача подій забезпечує безперервний потік та обробку даних, щоб потрібна інформація була в потрібному місці і в потрібний час.

Практичні приклади використання потокової передачі подій:

- Обробка платежів і фінансових операцій у режимі реального часу на фондових біржах, у банках і в страхових компаніях.
- Відстеження та моніторинг автомобілів, вантажівок, автопарків та вантажів у режимі реального часу у логістиці та автомобільній промисловості.
- Безперервний збір та аналіз даних з пристроїв Інтернету речей.
- Збір та миттєве реагування на дії та замовлення клієнтів у роздрібній торгівлі, готельній і туристичній індустрії та мобільних додатках, тощо.
- Стеження за пацієнтами, прогнозування змін стану для забезпечення своєчасного лікування у невідкладних випадках [17].

1.3.1 Apache Kafka

Kafka - це розподілена система та платформа для обробки потоків. Її можна розгорнути на «голому» залізі, віртуальних машинах і контейнерах у локальних та хмарних середовищах.

Одні з основних компонентів системи – це сервери та клієнти, що спілкуються через мережевий протокол TCP.

Сервери: Kafka запускається як кластер з одного або кількох серверів, які можуть охоплювати кілька центрів обробки даних або хмарних регіонів. Брокерами називаються деякі сервери, що утворюють шар зберігання даних. Для постійного експорту та імпорту даних, а також інтеграції з іншими існуючими системами та кластерами користувача, інші сервери запускають Kafka Connect. Кластери Kafka мають високу масштабованість і стійкість до відмов. За умови виходу будь-якого з серверів з ладу, його роботу на себе візьмуть інші сервери. Це в свою чергу забезпечує безперервну роботу без втрати даних.

Клієнти дозволяють писати розподілені програми та мікросервіси, які здатні паралельно зчитувати, записувати та обробляти потоки подій навіть у разі проблем мережі або перебоїв у роботі машини.

Також тут присутні відомі нам зі зв’язку «один до одного» поняття споживачів та виробників. У Kafka вони повністю не зв’язані одне з одним, що дозволяє легко забезпечити легку масштабованість. Наприклад, система пропонує різні гарантії щодо обробки подій:

- Максимум один раз – повідомлення можуть втрачені, але ніколи не доставлені повторно.
- Хоча б один раз – повідомлення ніколи не втрачаються і можуть бути доставлені повторно.
- Рівно раз – повідомлення надсилається лише один раз.

Події організовано зберігаються у темах, про які згадувалися у типі зв'язку «один до багатьох». Тема нагадує папку у файловій системі, де події є файлами в цій папці. Тема може або не мати, або мати один чи багато виробників, які записують до неї події, а також нуль, один чи багато споживачів, які підписуються на ці події. Кількість прочитань необмежена, бо на відміну від традиційних систем обміну повідомленнями, події не видаляються після використання. Замість цього розробник сам визначає, як довго Kafka має зберігати події через налаштування конфігурації для кожної теми. Старі події будуть усунуті лише після вказаного проміжку часу. Продуктивність Kafka не залежить від розміру даних, тому зберігання даних протягом тривалого часу цілком нормальна і ні на що не впливає.

Рисунок 1.4 Архітектура брокера

В свою чергу теми також розділені, тобто їхні дані розподілені між кількома відрами («bucket») у різних брокерів системи. Таке розміщення даних необхідне для забезпечення масштабованості, бо воно дозволяє клієнтським програмам і читати, і записувати дані від/до багатьох брокерів одночасно. При публікації нової події до теми, вона насправді додається до одного з розділів теми. Події з одним і тим же ключем події (ідентифікатором) записуються в той самий розділ. Одна з гарантій Kafka полягає у тому, що будь-який споживач певного розділу теми

завжди читатиме події цього розділу в тому самому порядку, в якому вони були записані.

Задля забезпечення відмово стійкості та високої доступності, можлива реплікація кожної теми, навіть якщо це означає розподіл між кількома географічними регіонами чи центрами обробки даних. Це надає кілька брокерів, що мають копію даних на випадок, коли щось піде не так або проводиться запланованe технічне обслуговування брокерів. Зазвичай обирається коефіцієнт реплікації 3, тобто завжди буде 3 копії ваших даних. Процес реплікації виконується на рівні розділів теми [17].

Для підтримки даних про назви і конфігурації в Kafka та забезпечення гнучкої та надійної синхронізації в розподілених системах використовується централізований сервіс Zookeeper. Він перевіряє статус вузлів кластера системи, відстежує теми, розділи, тощо.

Крім того Zookeeper надає можливість кільком клієнтом одночасно виконувати дії читання та запису, бо він виконує роль сервісу зі спільною в системі конфігурацією. Атомарне мовлення та видання впорядкованих оновлень забезпечує протокол атомарного мовлення (Zookeeper Atomic Broadcast) [18].

Протокол ZAB гарантує, що реплікація Zookeeper виконується в правильному порядку, а також відповідає за вибір вузлів-лідерів і відновлення будь-яких вузлів, які вийшли з ладу. В кожному кластері екосистеми Zookeeper є вузол-лідер, решта є його послідовниками. Він має обовязок відтворити всі клієнтські запити на зміни стану, що находять, собі та всім своїм послідовникам. Вхідні запити на читання балансуються між лідером та його послідовниками [19].

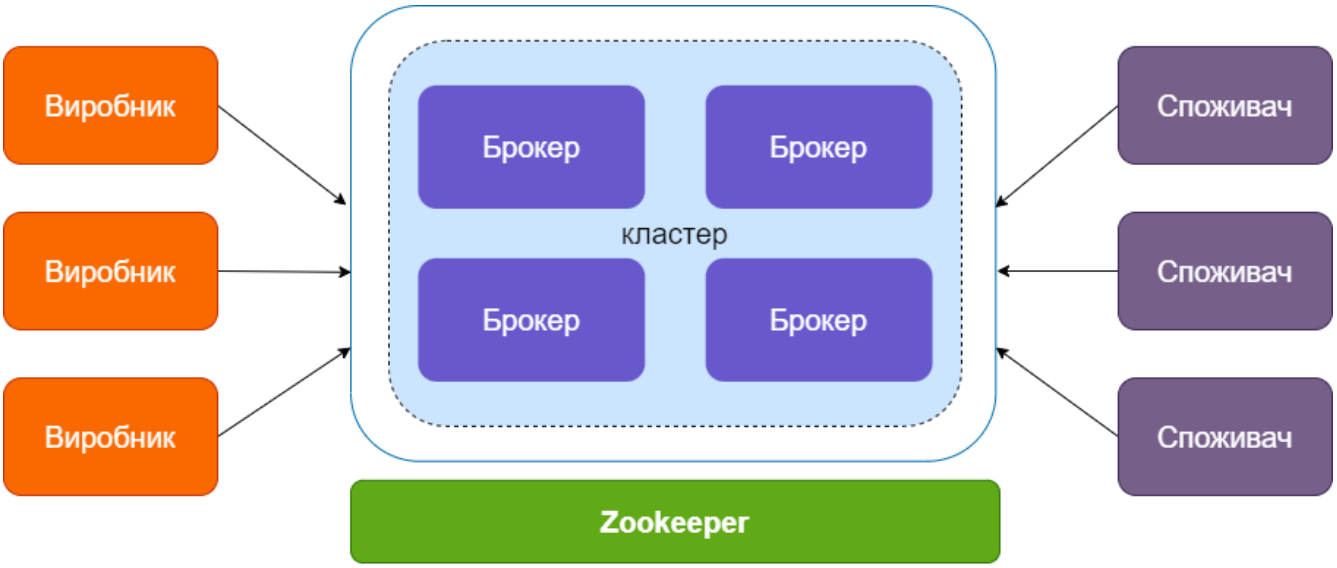

Рисунок 1.5 Архітектура однокластерного багатоброкерного Zookeeper

Надійність даних Kafka забезпечує через підтвердження («acknowledgements», «acks»). Виробник вважатиме запит завершеним за умови отримання виробником певної кількості підтверджень про отримання повідомлення брокерами:

- 0 підтверджень – в цьому випадку виробник не чекатиме ні на які підтвердження. Повідомлення вважатиметься відправленим, але немає гарантії, що воно було доставлене.
- 1 підтвердження – очікується підтвердження лише від брокера-лідера. Якщо у брокера щось пішло не так, а інші брокери не встигли відтворити запис у себе, він буде втраченим.
- Всі підтвердження – лідер буде чекати, поки усі інші брокери не скопіюють запис до себе. Це гарантує те, що запис житиме стільки часу, поки існує хоча б одна синхронізована репліка [17].

1.3.2 RabbitMQ

RabbitMQ – це традиційний брокер повідомлень. В основі його роботи лежить вже згаданий протокол Advanced Message Queing Protocol (AMQP), який дозволяє клієнтським застосункам спілкуватися з брокерами.

Як і Apache Kafka у RabbitMQ є брокери повідомлень, що отримують повідомлення від виробників та направляють їх до споживачів.

В рамках моделі повідомлення надсилаються до точок обміну («exchange»). Вони в свою чергу поширюють копії повідомлень у черги за допомогою правил, які називаються прив'язками («bindings»). Після цього брокер або доставляє повідомлення споживачам, які підписалися на черги, або споживачі витягують повідомлення з черг за необхідністю.

У повідомлень є метадані, які виробники можуть вказати під час публікації. Брокер може використати деякі з цих даних, проте зазвичай він не знає цю інформацію і нею користуються лише застосунки, що отримують повідомлення.

Для коректної роботи за ненадійної мережі протокол AMQP вводить поняття підтвердження повідомлення («acknowledge»). Після отримання повідомлення споживач повідомляє про це брокера: або автоматично, або у момент, який визначить розробник. Брокер повністю видалить повідомлення з черги після після отримання про отримання цього повідомлення або групи повідомлень.

Бувають випадки, коли повідомлення неможливо перенаправити. Тоді його можуть повернути виробникам, відкинути, або помістити у «чергу мертвих листів» («dead letter queue»), за умови, що брокер реалізує це розширення. Виробники самі обирають, як діяти в таких ситуаціях, за допомогою налаштування певних параметрів повідомлення.

У RabbitMQ є 5 видів точок обміну:

- Прямий обмін – в цьому випадку повідомлення надсилається у чергу визначену ключем маршрутизації. Це найкращий варіант для одноадресної маршрутизації повідомлень, але може використовуватися і для багатоадресної. Коли точка обміну отримує повідомлення, вона перевіряє його ключ і надсилає у чергу з таким самим ключем.

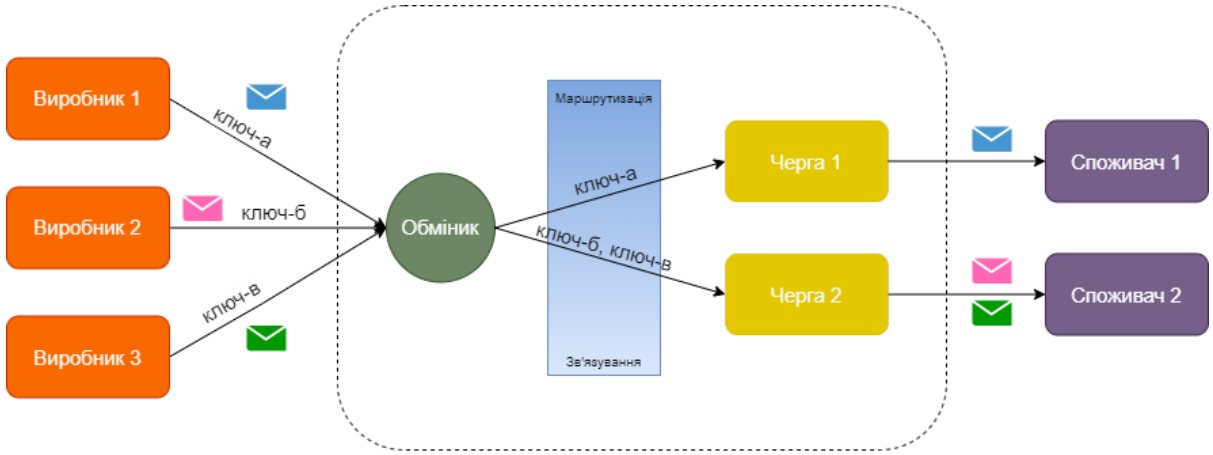

Рисунок 1.6 Прямий обмін в RabbitMQ

- Розширений обмін – повідомлення надсилаються у всі черги, які прив’язані до точки, а значення ключа маршрутизації ігнорується. Це ідеальний варіант для багатоадресної маршрутизації.

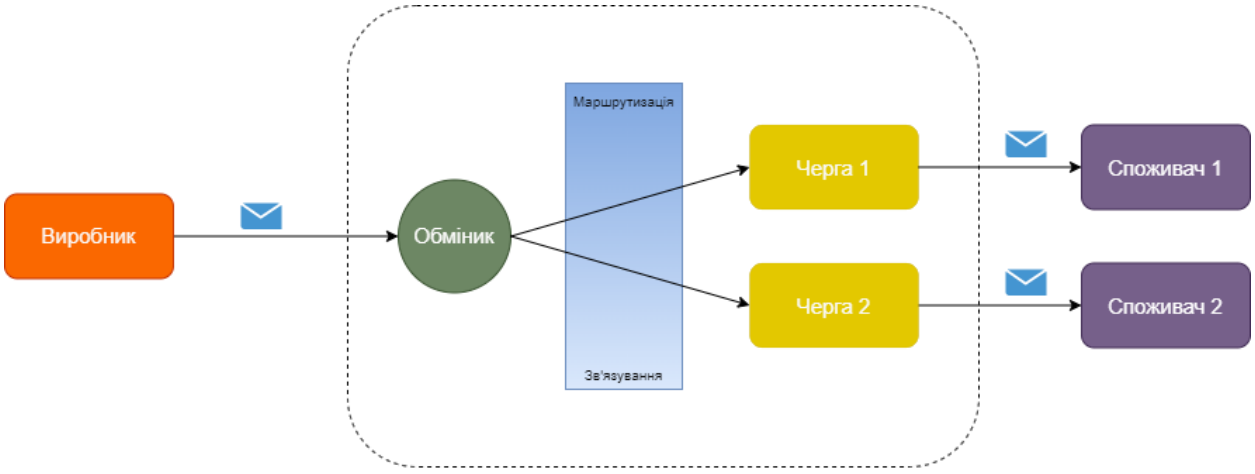

Рисунок 1.7 Розширений обмін в RabbitMQ

- Тематичний обмін – ідея схожа на прямий обмін, але тут ключ порівнюється не з конкретним значенням ключа черги, а з шаблоном. Цей підхід використовується для реалізації патерну «публікація / підписка». Використовується коли споживач вибірково обирає повідомлення, які він хоче отримувати.

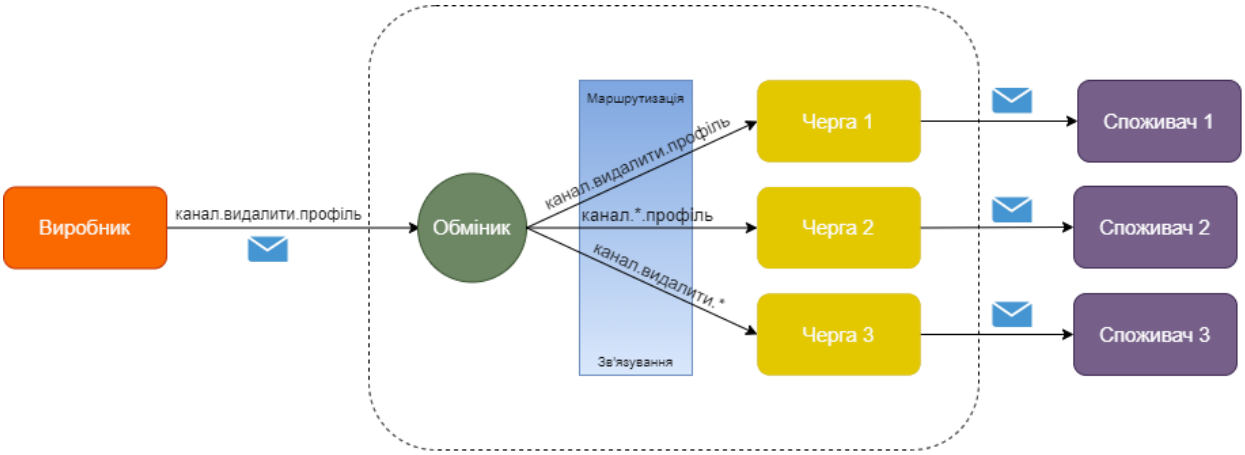

Рисунок 1.8 Шаблонний обмін в RabbitMQ

- Обмін заголовками – такий обмін призначений для маршрутизації за кількома атрибутами, які легше виразити через заголовки («headers»), ніж ключі маршрутизації. Оскільки привязати можна за кількома заголовками, розробник має вказати брокеру, чи збіг має бути за усіма заголовками, чи достатньо одного.

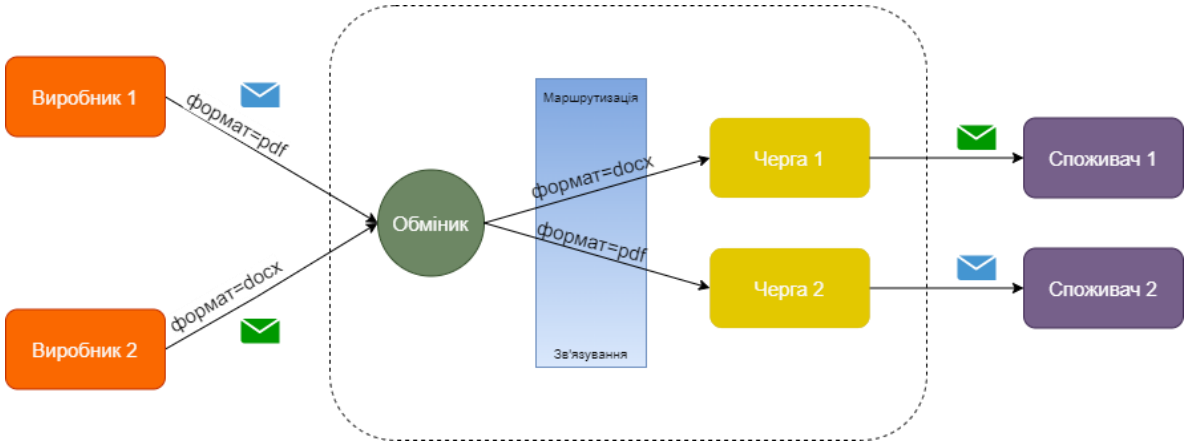

Рисунок 1.9 Обмін заголовками в RabbitMQ

- Точка обміну за замовчуванням – це прямий обмін повідомленнями без імені, який попередньо визначає брокер. Замість цього використовується пустий рядок. Кожна створена черга прив’язується до самої себе через ключ маршрутизації, що збігається з назвою черги. Завдяки цьому це найзручніший варіант для простих застосунків [20].

1.3.3 Порівняльний аналіз

Однією з головних відмінностей між Apache Kafka та RabbitMQ – це модель отримання повідомлень з черги, тобто тип взаємодії між брокером та споживачем.

Push модель – повідомлення штовхаються від брокера до споживача, тож другий отримує їх пасивно. За такого підходу повідомлення може бути відправлене споживачу одразу після надходження. Але може статися так, що брокери пересилають значно більше повідомлень, ніж можуть чи мають опрацювати споживачі.

Pull модель – споживачі мають надсилати запити на отримання повідомлення та самостійно витягати їх. Такий варіант більше підходить застосункам, які працюють з величезною кількістю повідомлень: брокери зберігають повідомлення і споживачі витягають в міру своїх можливістю. Недоліком цієї моделі є те, що споживачі не знають, коли у брокера з'явилось нове повідомлення, а занадто часто робити запити не можна [21].

Kafka використовує pull модель, а RabbitMQ – push. Тобто перша система більше підходить для застосунків, які оброблюють великі об'єми даних. Оскільки споживач має самостійно витягати повідомлення, це дає свободу реалізації цього процесу.

Перевагою RabbitMQ проти Kafka є складніша логіка маршрутизації повідомлень. Завдяки різним видам точок обміну користувач може легко отримати лише ту частину повідомлень, яка його цікавить, а не всі. Такий підхід може бути легше для інтегрування у застосунки.

2. Розробка власного рішення

Метою практичної частини роботи була визначена розробка власної системи мікросервісів з використанням сервісу Apache Kafka. Тема системи – піцерія «Kafka & Garfunkel». Необхідні компоненти:

- База даних: для зберігання інформації про замовлення.
- Клієнтська сторона: через цей застосунок відбувається створення користувачем замовлення та перегляду його статусу.
- Кур’єрська сторона: застосунок для відслідковування кур’єрами замовлень для їх подальшого виконання
- Сторона піцерії: застосунок для підготовлення замовлень користувачів.
- Дві теми у Kafka: для замовлень та нотифікацій щодо змінення статусу замовлень.

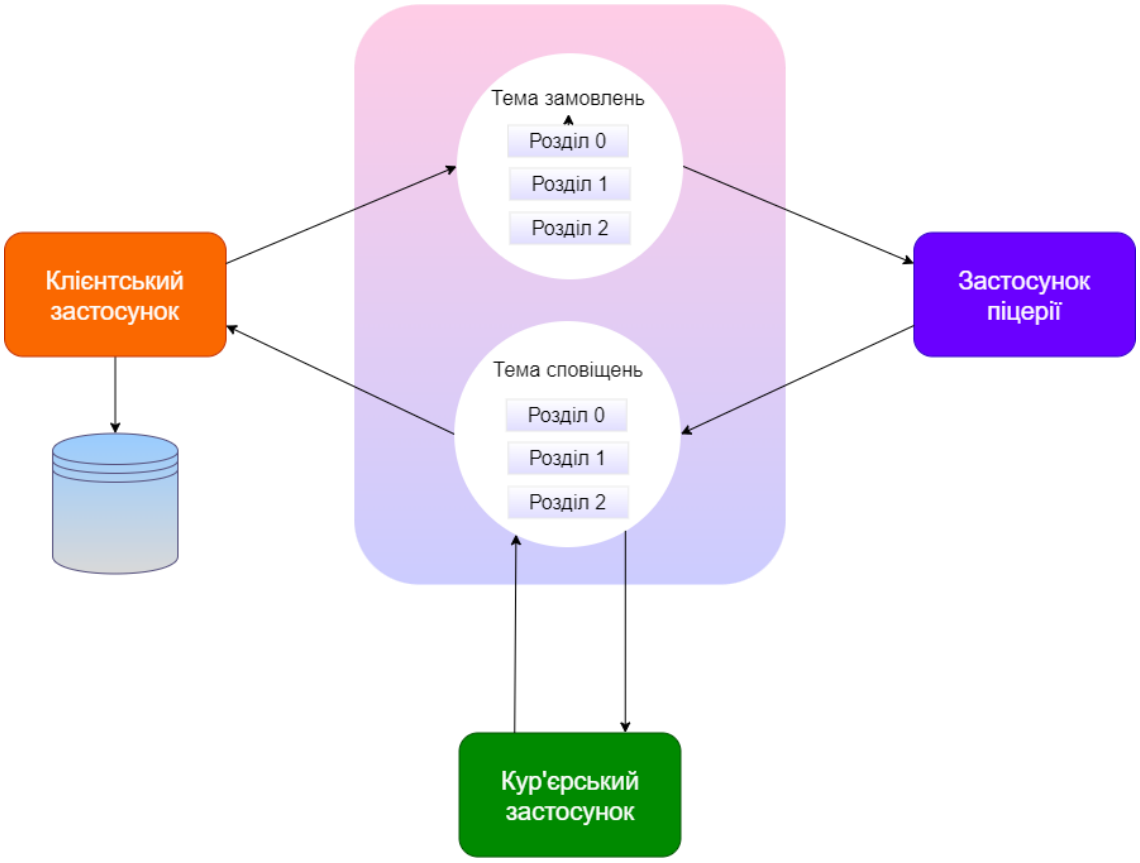

Рисунок 2.1 Структура системи

Вимоги до застосування Kafka:

- У кожного замовлення має бути кореляційний ідентифікатор.
- У кожного сервісу має бути 3 споживачі – по одному на кожен сервіс.
- Брокер має повідомляти про отримання запису «хоча б раз» - acks=1

Вимоги до застосунків клієнта, кур’єра та піцерії:

- REST API для взаємодії з користувачем

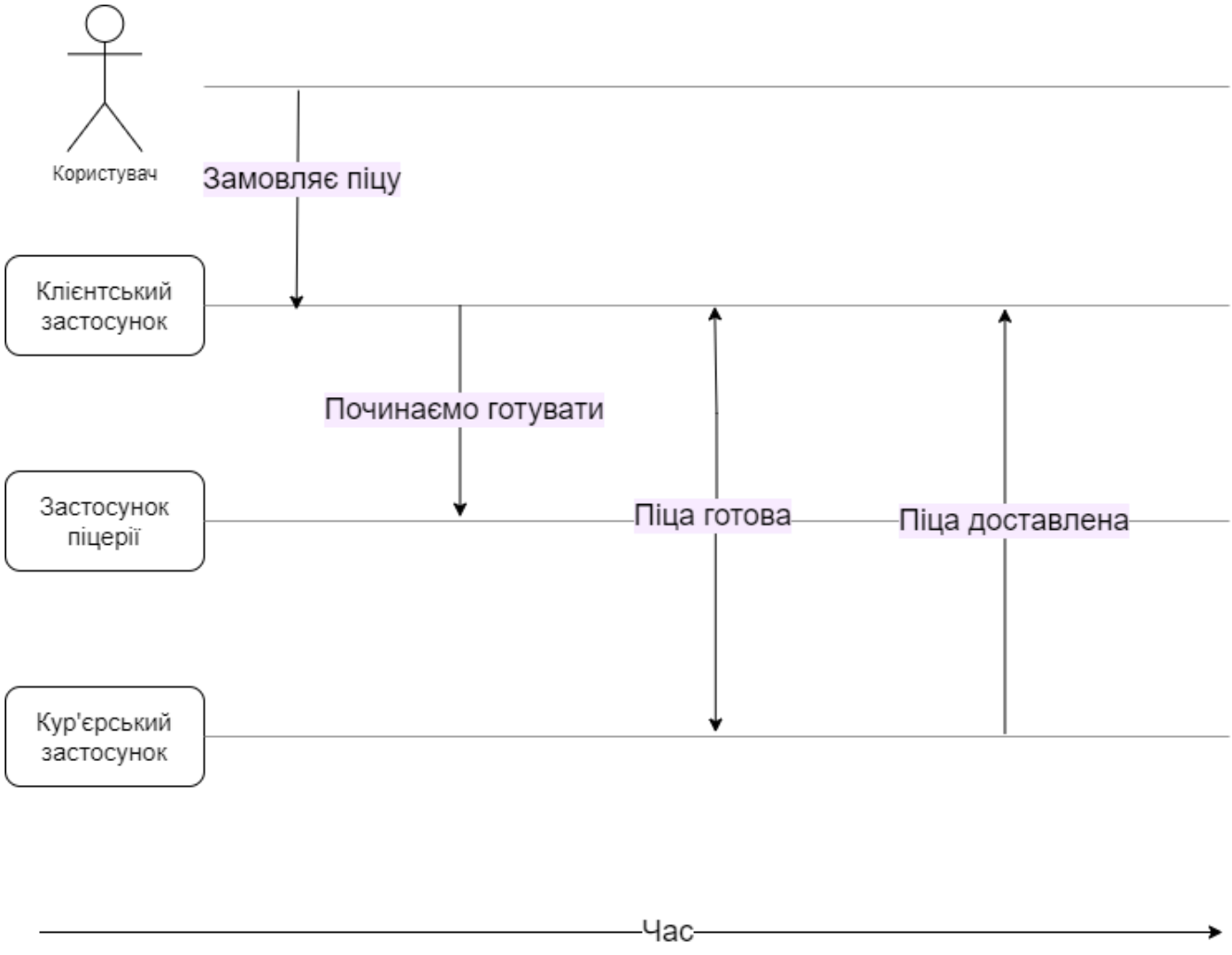

Рисунок 2.2 Діаграма процесів системи

2.1 База даних

Для реалізації шару даних практичної частини була обрана реляційна базу даних H2, оскільки завдання містить більше ознайомчий характер. У випадку створення реального продукту використовувалася б, наприклад, PostgreSQL.

Оскільки завдання передбачає збереження лише інформації про замовлення, була створена тільки одна таблиця, orders, яка відповідає сутності Order у коді застосунку.

Поле id – єдине унікальне. За ним і буде відбуватися ідентифікація замовлення.

Поля «ім’я клієнта» та «опис» є необов’язковими.

Поле «статус» може мати один з 5 статусів: нове (new), в процесі (in progress), готове (ready), в дорозі (on the way), доставлене (delivered).

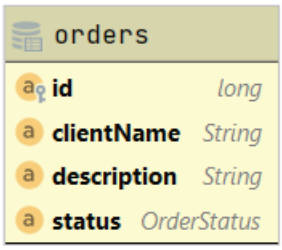

Рисунок 2.3 Діаграма бази даних застосунку

2.2Сервіси для користувачів

Оскільки є три види користувачів: клієнт, кур’єр та виробничий офіс, було створено три сервіси. Кожен з них є Spring Boot застосунком, написаним мовою програмування Java.

Клієнтський сервіс:

Для взаємодії з користувачем створено RestController з необхідними Get та Post методами для створення замовлення та отримання його статусу.

Запити до бази даних формуються через JpaRepository і транслюються у SQL запити за допомогою Hibernate.

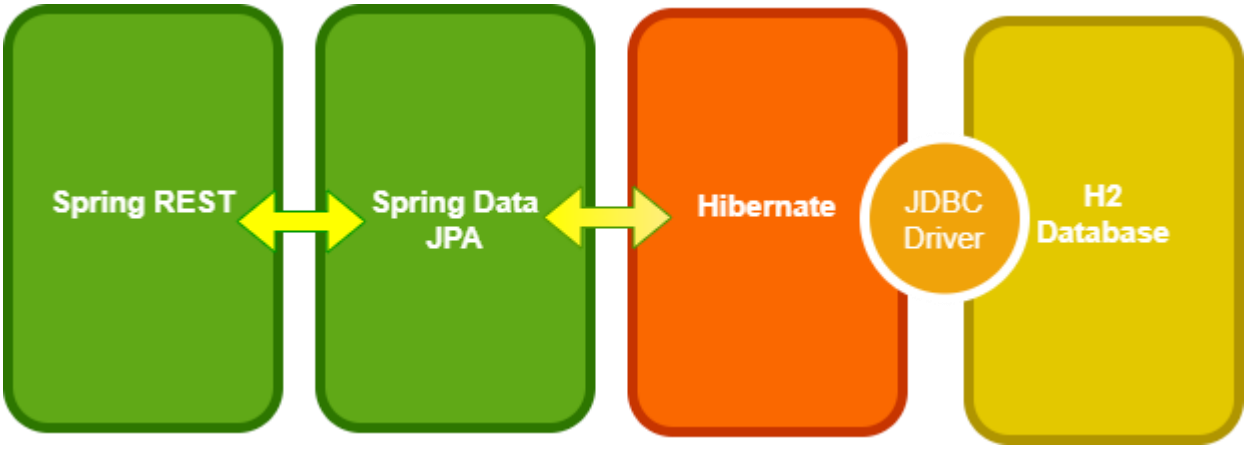

Рисунок 2.4 Структура роботи клієнтського сервісу

Кур’єрський та офісний сервіси мають схожі сервіси для оновлення інформації про замовлення.

Кожен сервіс передбачає можливість створення до нього користувацького інтерфейсу, але в межах даної роботи ця функціональність не була додана. За потреби кожен сервіс можна розгорнути окремо від інших.

2.3Інтеграція з Apache Kafka

Інтеграція усіх сервісів з Kafka відбулась за схожою послідовністю дій:

1) Додавання анотації EnableKafka до конфігурації споживачів чи виробників. Вона активує прослуховування кінцевих точок зв'язку Kafka.
2) Виставлення основних налаштувань фабрики : адреси Kafka сервера, ідентифікатор групи споживачів та клас для перетворення json об'єкта у необхідний нам клас, якщо це налаштування споживача.

```
@EnableKafka
@Configuration
public class KafkaConsumerConfig {

    @Value("${spring.kafka.boostrap-servers}")
    private String bootstrapAddress;

    @Value("${kafka.groupId}")
    private String groupId;

    @Bean
    public ConsumerFactory<Long, Order> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
        props.put(GROUP_ID_CONFIG, groupId);
        props.put(KEY_DESERIALIZER_CLASS_CONFIG, LongDeserializer.class);
        JsonDeserializer<Order> jsonDeserializer = new JsonDeserializer<>(Order.class);
        jsonDeserializer.ignoreTypeHeaders();
        return new DefaultKafkaConsumerFactory<>(props, new LongDeserializer(), jsonDeserializer);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<Long, Order> kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<Long, Order> factory = new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }
}
```

Рисунок 2.5 Налаштування параметрів спроживача

Після цього у кожному сервісі були створені слухачі для кожної необхідної для нього теми. Наприклад, кур'єр слухає тему «сповіщення». У випадку надходження нового запису, він перевіряє його статус. Якщо він дорівнює «готове», сервіс змінює його на «в дорозі», а після симуляції доставки ставить

значення «доставлено». Після кожного оновлення статусу він надсилає нову подію до Kafka.

```
@KafkaListener(topics = "notifications_topic", concurrency = "3")
public void notificationListener(ConsumerRecord<Long, Order> record) {
    log.info("Received order={}", record.value());

    Order order = record.value();

    if (order.getStatus() == READY) {
        log.info("Order id={} is ready", order.getId());
        updateStatus(order, ON_THE_WAY);
        simulateWaiting();
        updateStatus(order, DELIVERED);
    }
}


private void updateStatus(Order order, OrderStatus orderStatus) {
    Order updatedOrder = orderService.update(order, orderStatus);
    kafkaService.sendOrder( topic: "notifications_topic", updatedOrder);
}
```

Рисунок 2.6 Створення слухача для теми Kafka

Для роботи з Kafka у фреймворку Spring передбачений модуль spring-kafka, який надає клас KafkaTemplate, через який надсилаються записи до Kafka сервера та отримується інформацію про те, чи була успішною ця операція.

```
private final KafkaTemplate<Long, Order> kafkaTemplate;

@Autowired
public KafkaServiceImpl(KafkaTemplate<Long, Order> kafkaTemplate) { this.kafkaTemplate = kafkaTemplate; }

@Override
public void sendOrder(String topic, Order order) {
    ListenableFuture<SendResult<Long, Order>> future = kafkaTemplate.send(topic, order);

    future.addCallback(new ListenableFutureCallback<>() {

        @Override
        public void onSuccess(SendResult<Long, Order> result) {
            RecordMetadata recordMetadata = result.getRecordMetadata();
            log.info("Sent order {} with offset {} to topic {} partition {}", order, recordMetadata.offset(),
                    recordMetadata.topic(), recordMetadata.partition());
        }

        @Override
        public void onFailure(Throwable e) { log.info("Unable to send order {} due to {}", order, e.getMessage()); }
    });
}
```

Рисунок 2.7 Відправлення запису за допомогою KafkaTemplate

2.4 Запуск системи

Для запуску Zookeeper та брокера був використаний Docker та наступний docker-compose файл:

```yaml
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.0.1
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-kafka:7.0.1
    container_name: broker
    ports:
      - "9092:9092"
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_INTERNAL:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092,PLAINTEXT_INTERNAL://broker:29092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

Рисунок 2.8 Docker-compose файл для запуску в Docker брокера та Zookeeper

Для створення тем була запущена наступна команда, де замість topic_name були вказані назви notifications_topic та order_topic:
docker exec broker kafka-topics --bootstrap-server broker:9092 --create --topic <topic_name>

Після цього за допомогою HTTP POST запит було створено нове замовлення. Через GET запит відбувалася перевірка статусу замовлення. Для

детального огляду процесу обробки замовлення сервісами дивіться Додаток А, Додаток Б та Додаток В.

Рисунок 2.9 Результати запиту на отримання статусу замовлення під час роботи сервісів на різних етапах

Висновки

Під час виконання теоретичної частини роботи було розглянуто різні види архітектури застосунків, їхні переваги та недоліки. На основі мікросервісної архітектури як розподіленої системи було розглянуто різні типи комунікацій між сервісами. Крім того було детально досліджено Apache Kafka та RabbitMQ і проведено порівняльний аналіз цих двох брокерів повідомлень.

Отримані знання переконали у важливості та доцільності розробки мікросервісних застосунків з використанням Apache Kafka. Саме така система була реалізована в рамках практичної частини роботи. Результатом виконання стали три застосунки для уявної піцерії. Було продемонстровано роботу Kafka на прикладі пересилання між сервісами даних про замовлення.

З результатів виконання роботи можна зробити висновок, що Apache Kafka – це потужний інструмент для роботи з великою кількістю даних, який добре підходить для слабозв’язаних мікросервісних застосунків.

Список літератури

1. What is Three-Tier Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud/learn/three-tier-architecture
2. What is an application architecture? [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture
3. Richardson C. Pattern: Monolithic Architecture [Електронний ресурс] / Chris Richardson – Режим доступу до ресурсу: https://microservices.io/patterns/monolithic.html.
4. Gnatyk R. Microservices vs Monolith: which architecture is the best choice for your business? [Електронний ресурс] / Romana Gnatyk. – 2018. – Режим доступу до ресурсу: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/.
5. What Are Distributed Systems? [Електронний ресурс] – Режим доступу до ресурсу: https://www.splunk.com/en_us/data-insider/what-are-distributed-systems.html.
6. Richardson C. Introduction to Microservices [Електронний ресурс] / Chris Richardson. – 2015. – Режим доступу до ресурсу: https://www.nginx.com/blog/introduction-to-microservices.
7. Diguer S. Microservices Advantages and Disadvantages: Everything You Need to Know [Електронний ресурс] / Sarah Diguer. – 2020. – Режим доступу до ресурсу: https://solace.com/blog/microservices-advantages-and-disadvantages.
8. Richards M. Software Architecture Patterns [Електронний ресурс] / Mark Richards // O'Reilly Media, Inc.. – 2015. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html.

9. Event-driven architectures [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/eventarc/docs/event-driven-architectures.
10. EVENT-DRIVEN ARCHITECTURE TOPOLOGIES – BROKER AND MEDIATOR [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.3pillarglobal.com/insights/event-driven-architecture-topologies-broker-and-mediator/.
11. Cockburn A. Hexagonal architecture [Електронний ресурс] / Alistair Cockburn. – 2005. – Режим доступу до ресурсу: https://alistair.cockburn.us/hexagonal-architecture/.
12. Alliaume E. Hexagonal Architecture: three principles and an implementation example [Електронний ресурс] / E. Alliaume, S. Roccaserra. – 2018. – Режим доступу до ресурсу: https://blog.octo.com/hexagonal-architecture-three-principles-and-an-implementation-example.
13. Richardson C. Building Microservices: Inter-Process Communication in a Microservices Architecture [Електронний ресурс] / Chris Richardson. – 2015. – Режим доступу до ресурсу: https://www.nginx.com/blog/building-microservices-inter-process-communication/.
14. Ozkaya M. Microservices Communications [Електронний ресурс] / Mehmet Ozkaya. – 2021. – Режим доступу до ресурсу: https://medium.com/design-microservices-architecture-with-patterns/microservices-communications-f319f8d76b71.
15. What is a Message Queue? [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/message-queue.
16. What is Pub/Sub Messaging? [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/pub-sub-messaging.
17. DOCUMENTATION [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://kafka.apache.org/documentation.

18. Vinka E. What is Zookeeper and why is it needed for Apache Kafka? [Електронний ресурс] / Elin Vinka. – 2018. – Режим доступу до ресурсу: https://www.cloudkarafka.com/blog/cloudkarafka-what-is-zookeeper.html.
19. Vinka E. Zookeeper Atomic Broadcast Protocol (ZAB) and implementation of Zookeeper. [Електронний ресурс] / Elin Vinka. – 2018. – Режим доступу до ресурсу: https://www.cloudkarafka.com/blog/cloudkarafka-zab.html.
20. AMQP 0-9-1 Model Explained [Електронний ресурс] – Режим доступу до ресурсу: https://www.rabbitmq.com/tutorials/amqp-concepts.html.
21. Push or pull message queue, how do rocketmq and Kafka do? [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://developpaper.com/push-or-pull-message-queue-how-do-rocketmq-and-kafka-do.

Додатки

Додаток А. Інформація про обробку замовлення клієнтським сервісом у лог-файлах

```
2022-05-20 00:42:56.043  INFO 21336 --- [nio-8092-exec-3] c.k.client.controller.OrderController    : Created new order Order(id=5, clientName=Jane Doe, description=Cheese
 border, status=NEW)
2022-05-20 00:42:56.053  INFO 21336 --- [ad | producer-1] c.k.c.k.service.impl.KafkaServiceImpl    : Sent order Order(id=5, clientName=Jane Doe, description=Cheese border,
 status=NEW) with offset 4 to topic order_topic partition 0
2022-05-20 00:42:56.062  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.ClientListeners     : Client listener received order Order(id=5, clientName=Jane Doe,
 description=Cheese border, status=IN_PROGRESS)
2022-05-20 00:42:56.063  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.service.impl.OrderServiceImpl      : Update order id=5
2022-05-20 00:42:58.270  INFO 21336 --- [nio-8092-exec-4] c.k.client.controller.OrderController    : Calling get order endpoint, id=5
2022-05-20 00:43:00.586  INFO 21336 --- [nio-8092-exec-7] c.k.client.controller.OrderController    : Calling get order endpoint, id=5
2022-05-20 00:43:01.083  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.ClientListeners     : Client listener received order Order(id=5, clientName=Jane Doe,
 description=Cheese border, status=READY)
2022-05-20 00:43:01.083  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.service.impl.OrderServiceImpl      : Update order id=5
2022-05-20 00:43:01.095  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.ClientListeners     : Client listener received order Order(id=5, clientName=Jane Doe,
 description=Cheese border, status=ON_THE_WAY)
2022-05-20 00:43:01.095  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.service.impl.OrderServiceImpl      : Update order id=5
2022-05-20 00:43:03.046  INFO 21336 --- [nio-8092-exec-8] c.k.client.controller.OrderController    : Calling get order endpoint, id=5
2022-05-20 00:43:05.974  INFO 21336 --- [nio-8092-exec-1] c.k.client.controller.OrderController    : Calling get order endpoint, id=5
2022-05-20 00:43:06.097  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.ClientListeners     : Client listener received order Order(id=5, clientName=Jane Doe,
 description=Cheese border, status=DELIVERED)
2022-05-20 00:43:06.097  INFO 21336 --- [ntainer#0-0-C-1] c.k.c.service.impl.OrderServiceImpl      : Update order id=5
2022-05-20 00:43:07.982  INFO 21336 --- [nio-8092-exec-2] c.k.client.controller.OrderController    : Calling get order endpoint, id=5
```

Додаток Б. Інформація про обробку замовлення сервісом виробничого офісу у лог-файлах

```
2022-05-20 00:42:56.055  INFO 8668 --- [ntainer#0-0-C-1] c.k.k.k.l.KafkaAndGarfunkelListeners     : Received order=Order(id=5, clientName=Jane Doe, description=Cheese border,
 status=NEW)
2022-05-20 00:42:56.055  INFO 8668 --- [ntainer#0-0-C-1] c.k.k.service.impl.OrderServiceImpl      : Update order id=5, set newStatus=IN_PROGRESS
2022-05-20 00:42:56.061  INFO 8668 --- [ad | producer-1] c.k.k.k.service.impl.KafkaServiceImpl    : Sent order Order(id=5, clientName=Jane Doe, description=Cheese border,
 status=IN_PROGRESS) with offset 16 to topic notifications_topic partition 0
2022-05-20 00:43:01.066  INFO 8668 --- [ntainer#0-0-C-1] c.k.k.service.impl.OrderServiceImpl      : Update order id=5, set newStatus=READY
2022-05-20 00:43:01.077  INFO 8668 --- [ad | producer-1] c.k.k.k.service.impl.KafkaServiceImpl    : Sent order Order(id=5, clientName=Jane Doe, description=Cheese border,
 status=READY) with offset 17 to topic notifications_topic partition 0
```

Додаток В. Інформація про обробку замовлення кур’єрським сервісом у лог-файлах

```
2022-05-20 00:42:56.062  INFO 23836 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.CourierListeners    : Received order=Order(id=5, clientName=Jane Doe, description=Cheese border,
  status=IN_PROGRESS)
2022-05-20 00:43:01.080  INFO 23836 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.CourierListeners    : Received order=Order(id=5, clientName=Jane Doe, description=Cheese border,
  status=READY)
2022-05-20 00:43:01.082  INFO 23836 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.CourierListeners    : Order id=5 is ready
2022-05-20 00:43:01.082  INFO 23836 --- [ntainer#0-0-C-1] c.k.c.service.impl.OrderServiceImpl      : Update order id=5, set newStatus=ON_THE_WAY
2022-05-20 00:43:01.092  INFO 23836 --- [ad | producer-1] c.k.c.k.service.impl.KafkaServiceImpl    : Sent order Order(id=5, clientName=Jane Doe, description=Cheese border,
 status=ON_THE_WAY) with offset 18 to topic notifications_topic partition 0
2022-05-20 00:43:06.091  INFO 23836 --- [ntainer#0-0-C-1] c.k.c.service.impl.OrderServiceImpl      : Update order id=5, set newStatus=DELIVERED
2022-05-20 00:43:06.096  INFO 23836 --- [ad | producer-1] c.k.c.k.service.impl.KafkaServiceImpl    : Sent order Order(id=5, clientName=Jane Doe, description=Cheese border,
 status=DELIVERED) with offset 19 to topic notifications_topic partition 0
2022-05-20 00:43:06.099  INFO 23836 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.CourierListeners    : Received order=Order(id=5, clientName=Jane Doe, description=Cheese border,
  status=ON_THE_WAY)
2022-05-20 00:43:06.105  INFO 23836 --- [ntainer#0-0-C-1] c.k.c.kafka.listener.CourierListeners    : Received order=Order(id=5, clientName=Jane Doe, description=Cheese border,
  status=DELIVERED)
```