МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Оксана ШОВКОПЛЯС
____
(підпис)

________ грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»
освітньо-професійної програми «Інформатика»
на тему: Інформаційна технологія розробки розподілених систем на основі Java
здобувача групи ІН.м-32 Руденка Єгора Андрійовича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

________ Єгор РУДЕНКО
(підпис)

Керівник
доцент кафедри комп'ютерних наук, к.т.н.
Наталія БАРЧЕНКО

____
(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_________ Оксана ШОВКОПЛЯС
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 «Комп'ютерні науки», освітньо-професійної програми «Інформатика»
здобувача групи ІН.м-32 Руденка Єгора Андрійовича

1. Тема роботи: Інформаційна технологія розробки розподілених систем на основі Java
затверджую наказом по СумДУ від *«03» грудня 2024 року № 1257-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 06 грудня 2024 року*
3. Вхідні дані до кваліфікаційної роботи ________
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження.
2) Моделювання та проєктування розподіленої системи
3) Програмна реалізація розподіленої системи
4) Аналіз результатів.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)________
6. Консультанти до проекту (роботи), із зазначенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» ________________ 20 ___ р.

Завдання прийняв до виконання _______________ Керівник _______________
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
	Аналіз проблеми та аналітичний огляд існуючих рішень	*01.09 – 12.09*	
	Постановка завдання та вибір методів реалізації	*15.09 – 26.09*	
	Моделювання та проєктування розподіленої системи	*29.09 – 30.10*	
	Програмна реалізація розподіленої системи	*03.11 – 28.11*	
	Оформлення пояснювальної записки до кваліфікаційної роботи	*01.12 – 10.12*	

Здобувач вищої освіти _______________ Керівник _______________
(підпис) (підпис)

АНОТАЦІЯ

Записка: 96 стр., 12 рис., 45 додаток, 25 використаних джерел.

Обґрунтування актуальності теми роботи - Робота присвячена розробці розподілених систем на Java, що є актуальним у зв'язку з потребою в масштабованих, надійних і продуктивних рішеннях для сучасних бізнесів, хмарних технологій та IoT.

Об'єкт дослідження - Розподілені системи на Java, їх архітектура та взаємодія компонентів.

Предмет дослідження - Методи розробки розподілених систем із використанням фреймворку Spring.

Мета - Створення інформаційної технології з використанням мікросервісної архітектури та Spring для забезпечення ефективності та надійності.

Результати - Реалізовано інформаційну технологію розробки розподілених систем з мікросервісною архітектурою, протестовану на продуктивність, масштабованість і безпеку. Надано рекомендації щодо її використання.

РОЗПОДІЛЕНІ СИСТЕМИ, JAVA, SPRING, МІКРОСЕРВІСИ, АРХІТЕКТУРНІ ШАБЛОНИ, ІНТЕГРАЦІЯ, ТЕСТУВАННЯ.

ЗМІСТ

ВСТУП.. 5

1. АНАЛІТИЧНИЙ ОГЛЯД ТА ПОСТАНОВКА ЗАВДАННЯ ДОСЛІДЖЕННЯ.. 7

1.1 Огляд існуючих підходів до розробки розподілених систем........... 7

1.2 Аналіз можливостей Java для будівництва розподілених систем .. 13

1.3 Постановка завдання та вибір інструментів розробки.................... 17

2. ВИБІР ІНСТРУМЕНТІВ ДЛЯ РОЗВ’ЯЗАННЯ ЗАДАЧІ 21

2.1 Вибір архітектури для реалізації розподіленої системи................. 21

2.2 Вибір методології та інструментів для реалізації розподіленої системи 24

3. ПРОГРАМНА РЕАЛІЗАЦІЯ РОЗПОДІЛЕНОЇ СИСТЕМИ 30

3.1 Розробка моделі інформаційної системи 30

3.2 Програмна реалізація розподіленої системи 34

3.3 Тестування та верифікація системи.. 52

ВИСНОВКИ.. 63

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 65

ДОДАТКИ.. 68

ВСТУП

Обґрунтування вибору теми роботи.

Тема «Інформаційна технологія розробки розподілених систем на основі Java» є важливою через зростаючу потребу в сучасних рішеннях для обробки великих обсягів даних, забезпечення безперервної роботи сервісів і оптимізації інфраструктури. Розподілені системи є фундаментом хмарних обчислень, дата-центрів та IoT-рішень.

Мова програмування Java широко використовується для розробки таких систем завдяки своїй портативності, масштабованості, багатопоточності та розвиненій екосистемі бібліотек і фреймворків. Її можливості забезпечують інтеграцію компонентів, ефективну обробку даних у реальному часі та створення надійних і продуктивних систем, що відповідають сучасним вимогам.

Актуальність. Тема «Інформаційна технологія розробки розподілених систем на основі Java» є актуальною через зростання попиту на розподілені обчислювальні системи, які забезпечують масштабованість, ефективність та надійність для сучасних бізнесів, наукових установ і державних організацій. Розвиток таких технологій, як мікросервіси, хмарні обчислення, Docker, Kubernetes, і мобільних застосунків підсилює потребу в оптимальних рішеннях. Використання Java у розробці розподілених систем є обґрунтованим завдяки її портативності, підтримці багатопоточності, багатій екосистемі фреймворків та бібліотек, а також потужним засобам для створення надійних та продуктивних систем.

Об'єкт дослідження. Розподілені інформаційні системи, розроблені на основі Java, їх архітектура, компоненти, механізми взаємодії та середовища виконання.

Предмет дослідження. Процеси розробки, реалізації та оптимізації розподілених систем із використанням Java, а також фреймворків та бібліотек, які сприяють їх створенню.

Новизна. Робота пропонує комплексний підхід до створення розподілених систем на Java з акцентом на використання фреймворка Spring. У дослідженні розроблено нові моделі взаємодії між компонентами системи для підвищення їх продуктивності, масштабованості та безпеки. Також запропоновано підходи до тестування та верифікації систем для забезпечення надійності і відповідності сучасним вимогам.

Структура. Дана робота складається зі вступу, аналітичного огляду, постановки задачі, вибір методу розв’язання поставленої задачі, опису програмного забезпечення інформаційної системи, висновків, списку використаних джерел та додатків.

1. АНАЛІТИЧНИЙ ОГЛЯД ТА ПОСТАНОВКА ЗАВДАННЯ ДОСЛІДЖЕННЯ

1.1 Огляд існуючих підходів до розробки розподілених систем

Огляд існуючих підходів до розробки розподілених систем охоплює різноманітні архітектурні моделі, технології та методології, які забезпечують ефективну взаємодію між компонентами, що функціонують на різних машинах або в різних мережах. Одним із найпоширеніших підходів є клієнт-серверна архітектура, в якій клієнти запитують ресурси або сервіси у центрального сервера, який обробляє ці запити та надає відповідну інформацію. Цей підхід широко використовується у веб-додатках, де браузер виступає клієнтом, а сервер обробляє HTTP-запити. Наприклад, у системах електронної комерції клієнт може надсилати запит на отримання товарів, а сервер відповідає даними про доступні продукти [1].

Іншим важливим підходом є архітектура мікросервісів, що передбачає розподіл системи на невеликі, незалежні сервіси, які можуть взаємодіяти через легкі протоколи, такі як HTTP або AMQP. Кожен мікросервіс виконує конкретну функцію і може бути розроблений, протестований і розгорнутий незалежно від інших, що дозволяє досягти більшої гнучкості і масштабованості. Наприклад, в електронній комерції можна реалізувати окремі сервіси для обробки платежів, управління запасами, користувацьких даних тощо. Це дозволяє швидше впроваджувати нові функції, оскільки зміни в одному сервісі не впливають на інші.

Об’єктно-орієнтовані принципи також відіграють важливу роль у розробці розподілених систем. Вони дозволяють структурувати код у вигляді об’єктів, що полегшує його повторне використання та тестування. Підходи, такі як інверсія управління (IoC) і аспектно-орієнтоване програмування (AOP), допомагають розв'язати проблеми, пов'язані з керуванням

залежностями і крос-кутовою логікою, що є критичними для великих розподілених систем.

Протоколи комунікації, такі як REST (Representational State Transfer) та SOAP (Simple Object Access Protocol), забезпечують стандартизований спосіб обміну даними між компонентами системи. REST, який заснований на HTTP, є більш легким і простим у використанні, тоді як SOAP надає більш складні механізми безпеки та транзакцій. Використання REST API стало стандартом для більшості веб-додатків, оскільки воно спрощує інтеграцію між різними сервісами [2].

Додатково, технології, такі як Java RMI (Remote Method Invocation) та CORBA (Common Object Request Broker Architecture), надають можливості для віддаленого доступу до об’єктів і реалізації складніших механізмів взаємодії. Java RMI дозволяє об’єктам на різних JVM (Java Virtual Machines) взаємодіяти, використовуючи сервіси віддаленого виклику методів [3]. CORBA, у свою чергу, забезпечує можливість міжплатформеного взаємодії, що є важливим для інтеграції систем, які функціонують на різних технологічних платформах.

У останні роки спостерігається зростання популярності платформ для розробки розподілених систем, зокрема Spring, яка пропонує потужні інструменти для створення, налаштування та управління компонентами. Spring Framework забезпечує інверсію управління, що дозволяє знижувати з’єднання між компонентами, а також пропонує модулі для роботи з мікросервісами, безпекою, управлінням транзакціями та іншими аспектами. Завдяки Spring Boot, розробники можуть швидко налаштувати нові проекти з мінімальними зусиллями, що значно підвищує продуктивність.

Також важливим є використання контейнеризації, зокрема Docker, для полегшення розгортання та масштабування розподілених систем. Контейнери дозволяють упаковувати додатки разом з усіма їх залежностями, що спрощує процес впровадження і забезпечує однакову поведінку у різних середовищах. Хмарні технології, такі як Kubernetes, автоматизують управління

контейнерами, забезпечуючи високу доступність та масштабованість систем. Використання цих технологій дозволяє організаціям швидко реагувати на зміни в навантаженнях і вимогах, що є критично важливим у сучасних умовах бізнесу [4,5].

Таким чином, існуючі підходи до розробки розподілених систем демонструють постійний розвиток і адаптацію до нових технологічних викликів. Вибір архітектурних патернів і технологій залежить від конкретних вимог проекту, що є критично важливим для успішної реалізації розподілених систем. Системи, побудовані за допомогою цих підходів, забезпечують ефективність, надійність та масштабованість, що робить їх придатними для широкого спектру застосувань у різних галузях.

У контексті розробки розподілених систем також важливо враховувати питання безпеки. З огляду на те, що компоненти розподілених систем взаємодіють через мережі, забезпечення конфіденційності та цілісності даних стає критично важливим. Застосування протоколів шифрування, таких як SSL/TLS для захисту даних під час передачі, а також механізмів аутентифікації і авторизації, таких як OAuth 2.0, є обов’язковими для запобігання несанкціонованому доступу та зловживанням. Додатково, важливо впроваджувати стратегії управління ризиками, які враховують потенційні загрози і вразливості розподілених систем [6].

Одним із сучасних підходів до розробки розподілених систем є використання архітектури на основі подій (event-driven architecture). У такій архітектурі компоненти системи взаємодіють через події, що знижує з'єднання між ними і підвищує гнучкість. Події можуть генеруватися різними джерелами, такими як користувацькі дії, зміни стану системи або зовнішні сигнали. Компоненти, які підписані на ці події, можуть реагувати на них асинхронно, що дозволяє досягти високої продуктивності та зменшує затримки у взаємодії. Застосування таких технологій, як Apache Kafka або RabbitMQ, дозволяє реалізувати механізми обробки подій у реальному часі,

що є особливо важливим для систем, які потребують швидкого реагування на зміни [7].

Також варто зазначити, що методології Agile та DevOps стають все більш популярними у розробці розподілених систем. Agile дозволяє адаптуватися до змін у вимогах, що є критично важливим в умовах динамічного ринку. DevOps, у свою чергу, об'єднує команди розробників і операційних фахівців, що забезпечує безперервну інтеграцію та доставку (CI/CD) програмного забезпечення. Завдяки цим методологіям компанії можуть швидше впроваджувати нові функції, знижувати ризики і покращувати якість продуктів.

Слід також розглянути питання масштабування розподілених систем. З часом навантаження на системи може значно зростати, що вимагає відповідного масштабування. Масштабування може бути горизонтальним (додавання нових вузлів) або вертикальним (збільшення ресурсів існуючих вузлів) [8]. Вибір підходу залежить від архітектури системи та специфічних вимог до продуктивності. Системи, побудовані на основі мікросервісів, часто підлягають горизонтальному масштабуванню, що дозволяє розподілити навантаження між кількома серверами та забезпечити високу доступність.

Важливою складовою успішної розробки розподілених систем є моніторинг і управління їхньою продуктивністю. Використання інструментів моніторингу, таких як Prometheus або Grafana, дозволяє фахівцям відстежувати стан системи в реальному часі, виявляти проблеми та оперативно на них реагувати. Аналітика даних, зібраних під час роботи системи, може бути використана для оптимізації її продуктивності та виявлення тенденцій у навантаженнях, що допомагає в плануванні ресурсів.

У цілому, існуючі підходи до розробки розподілених систем демонструють постійний розвиток і вдосконалення. Вибір архітектурних моделей, технологій та методологій є критично важливим для успішної реалізації проектів. Зосередження на безпеці, масштабованості, моніторингу та управлінні продуктивністю дозволяє забезпечити надійність і ефективність

розподілених систем у різних сферах застосування, від бізнес-додатків до наукових досліджень та соціальних платформ. Цей огляд показує, що успішна розробка розподілених систем є комплексним завданням, яке вимагає глибокого розуміння не лише технологій, а й методів управління проектами, що підкреслює важливість міждисциплінарного підходу в цій галузі.

Подальший розвиток розподілених систем спостерігається завдяки впровадженню нових технологій, таких як штучний інтелект (ШІ) та машинне навчання. Ці технології дозволяють системам адаптуватися до змінних умов, прогнозувати навантаження та оптимізувати роботу компонентів у реальному часі. Наприклад, у системах моніторингу можуть використовуватися алгоритми машинного навчання для виявлення аномалій у даних, що свідчить про потенційні проблеми у роботі системи. Це дозволяє своєчасно реагувати на загрози і забезпечувати безперервну роботу.

Також важливим аспектом є розробка та впровадження нових стандартів для забезпечення сумісності між різними компонентами розподілених систем. Розробка відкритих стандартів дозволяє знизити ризики, пов’язані з інтеграцією різних технологій, і забезпечує більшу гнучкість у виборі постачальників і технологій. Це стає критично важливим у контексті глобалізації та розширення ринків, де компанії повинні мати можливість швидко адаптувати свої системи до нових умов.

З метою підвищення продуктивності розподілених систем також активно впроваджуються технології оркестрації, такі як Kubernetes, які автоматизують управління контейнерами. Ці технології забезпечують автоматичне масштабування, управління навантаженням і відновлення систем у випадку збоїв, що робить системи більш надійними і зручними в експлуатації. Завдяки оркестрації компанії можуть ефективніше використовувати ресурси, знижуючи витрати на обслуговування та управління.

Ще одним перспективним напрямком є використання блокчейн-технологій у розподілених системах. Блокчейн надає механізми для забезпечення безпеки і прозорості транзакцій, що є особливо важливим у сферах, де конфіденційність і цілісність даних є критичними. Ця технологія дозволяє створювати дистрибуційні реєстри, які забезпечують верифікацію даних без необхідності залучення центрального управління, що підвищує довіру між учасниками системи [9].

Крім того, сучасні системи все більше використовують підходи до автоматизації тестування та CI/CD, що дозволяє зменшити час виходу на ринок нових продуктів. Інструменти автоматизації, такі як Jenkins, CircleCI та GitLab CI, інтегруються у процес розробки, що дозволяє забезпечити безперервний контроль якості програмного забезпечення. Завдяки автоматизації тестування можна швидше виявляти помилки і недоліки в коді, що сприяє підвищенню загальної якості продуктів.

Важливо зазначити, що розробка розподілених систем потребує врахування аспектів управління даними. В умовах зростаючої кількості даних і складності їх обробки, з’являється потреба у використанні нових рішень для зберігання та аналізу даних. Системи, такі як Apache Hadoop і NoSQL бази даних (наприклад, MongoDB, Cassandra), дозволяють ефективно управляти великими обсягами даних, що необхідно для сучасних розподілених систем. Вони забезпечують гнучкість у структурі даних і можливість масштабування, що є критично важливим для динамічно змінюваних бізнес-середовищ.

Розподілені системи також стикаються з викликами, пов'язаними з управлінням конфігурацією та версіями. Зі збільшенням кількості мікросервісів і компонентів управління версіями стає все більш складним. Використання контейнерів і систем оркестрації дозволяє вирішити ці проблеми, але все ж потребує впровадження чітких практик і інструментів для управління конфігурацією, таких як Ansible або Terraform. Ці інструменти забезпечують автоматизацію процесів налаштування і

управління середовищами, що дозволяє знизити ризики помилок і покращити ефективність.

На завершення, сучасні підходи до розробки розподілених систем демонструють інтеграцію численних технологій і методологій, які сприяють підвищенню продуктивності, надійності та гнучкості. Завдяки використанню новітніх технологій, таких як ШІ, блокчейн, оркестрація і автоматизація, компанії можуть створювати складні та ефективні системи, що відповідають вимогам сучасного бізнес-середовища. Переосмислення традиційних підходів у поєднанні з новими технологічними досягненнями відкриває нові можливості для розробки та вдосконалення розподілених систем у найближчому майбутньому.

1.2 Аналіз можливостей Java для будівництва розподілених систем

Java, як об'єктно-орієнтована мова програмування, відіграє значну роль у будівництві розподілених систем завдяки своїй простоті, універсальності та потужному набору інструментів і бібліотек. Однією з основних переваг Java є її незалежність від платформи, що забезпечується механізмом Java Virtual Machine (JVM). Ця характеристика дозволяє розробникам створювати програми, які можуть виконуватися на різних операційних системах без необхідності модифікації коду. Завдяки цьому Java є ідеальним вибором для розподілених систем, які можуть мати різноманітні апаратні платформи та програмні середовища.

Однією з ключових можливостей Java є потужна система роботи з мережами, яка забезпечує простоту створення клієнт-серверних додатків. Java надає широкий набір API для роботи з мережевими протоколами, такими як TCP/IP, UDP, HTTP та інші, що дозволяє розробникам реалізувати різноманітні сценарії взаємодії між компонентами системи. Java RMI (Remote Method Invocation) дозволяє викликати методи об'єктів, які знаходяться на віддалених машинах, що спрощує реалізацію розподілених обчислень. RMI

автоматично обробляє серіалізацію об'єктів, що забезпечує безшовну передачу даних між різними вузлами.

Крім того, Java підтримує різні парадигми програмування, включаючи об'єктно-орієнтоване, функціональне та імперативне програмування. Це дозволяє розробникам вибирати підходящий стиль програмування в залежності від специфіки проекту. Об'єктно-орієнтоване програмування особливо корисне в контексті розподілених систем, оскільки дозволяє структурувати код у вигляді об'єктів, що моделюють реальні сутності, і полегшує підтримку та розширення системи.

Java також має потужні інструменти для реалізації багатопотоковості, що є критично важливим для розподілених систем, де паралельна обробка запитів може значно підвищити продуктивність. Завдяки вбудованим механізмам для створення та управління потоками, розробники можуть легко реалізовувати багатопотокові програми, які ефективно використовують ресурси системи. Java Concurrency API, що включає в себе класи для роботи з потоками, синхронізації та управління ресурсами, дозволяє створювати складні паралельні алгоритми з меншими зусиллями.

Java також підтримує важливі компоненти для будівництва розподілених систем, такі як сервлети та JavaServer Pages (JSP). Вони дозволяють створювати динамічні веб-додатки, які можуть обслуговувати численні клієнтські запити одночасно. Використання фреймворків, таких як Spring, ще більше спрощує розробку розподілених систем, надаючи інструменти для побудови RESTful API, що дозволяє реалізувати мікросервісну архітектуру. Spring Boot, зокрема, полегшує процес налаштування та розгортання мікросервісів, забезпечуючи автоматичну конфігурацію та інтеграцію з різними технологіями.

У Java також реалізовані інструменти для роботи з базами даних, такі як Java Database Connectivity (JDBC) та ORM-фреймворки, наприклад, Hibernate. Це дозволяє легко інтегрувати розподілену систему з реляційними та нереляційними базами даних, що забезпечує зберігання та доступ до

даних, необхідних для роботи системи. Використання JPA (Java Persistence API) з Hibernate дозволяє зручно працювати з даними в об'єктно-орієнтованому стилі, що спрощує обробку даних і взаємодію з базами.

Java також надає різноманітні бібліотеки для реалізації обробки подій та асинхронної взаємодії, що є важливими аспектами розподілених систем. Бібліотеки, такі як CompletableFuture і Reactor, дозволяють реалізовувати асинхронні обчислення, які сприяють більш ефективному використанню ресурсів і покращують загальну продуктивність системи [10].

Слід також зазначити, що Java має велику спільноту розробників, що забезпечує доступ до численних ресурсів, документації та бібліотек, які можуть бути використані при розробці розподілених систем. Спільнота активно підтримує розвиток мови, пропонуючи нові фреймворки та рішення, що сприяє еволюції Java як платформи для розробки.

Завдяки своїм можливостям, Java продовжує залишатися однією з найбільш популярних мов програмування для розробки розподілених систем. Її багатий набір інструментів, бібліотек і фреймворків, а також здатність інтегрувати новітні технології роблять Java ідеальним вибором для створення складних, масштабованих і надійних розподілених рішень у різних сферах, від фінансів до медицини та електронної комерції. Це робить Java не лише потужним інструментом, а й універсальним рішенням для сучасних бізнес-завдань, що постійно змінюються.

Переходячи до конкретних інструментів та фреймворків, слід зазначити, що Java надає широкий спектр рішень для різних аспектів розробки розподілених систем. Наприклад, Spring Framework став основою для багатьох сучасних Java-додатків, забезпечуючи потужну інфраструктуру для управління залежностями, обробки запитів та інтеграції з іншими системами. Spring дозволяє розробникам швидко створювати масштабовані мікросервіси, використовуючи такі компоненти, як Spring Boot для автоматизації налаштувань, Spring Data для спрощення доступу до бази

даних і Spring Cloud для роботи з мікросервісними архітектурами в розподілених середовищах.

Крім того, Spring Cloud надає різні інструменти для реалізації важливих аспектів мікросервісної архітектури, таких як управління конфігурацією, маршрутизація запитів, балансування навантаження, а також обробка помилок і збоїв у системі. Зокрема, Eureka забезпечує можливості служби реєстрації, що дозволяє мікросервісам знаходити один одного, а Zuul відповідає за маршрутизацію запитів до відповідних сервісів. Це дозволяє створювати динамічні й адаптивні архітектури, які можуть легко масштабуватися та змінюватися в залежності від потреб бізнесу.

Java також підтримує реалізацію систем на основі контейнеризації, що надає додаткові можливості для розгортання та управління розподіленими системами. Інтеграція з Docker та Kubernetes дозволяє розробникам створювати контейнеризовані мікросервіси, які можна легко переносити між різними середовищами, автоматизувати процеси CI/CD (безперервна інтеграція/безперервне постачання) та забезпечувати високу доступність й відмовостійкість системи.

Ще однією важливою характеристикою Java є підтримка технологій, які забезпечують безпеку розподілених систем. Бібліотеки, такі як Spring Security, дозволяють реалізовувати різноманітні механізми аутентифікації та авторизації, забезпечуючи захист даних і ресурсів системи. Також Java має вбудовані механізми для роботи з шифруванням, що дозволяє захистити чутливу інформацію під час її передачі через мережу.

Важливим аспектом при створенні розподілених систем є також моніторинг і управління. Java надає можливості для інтеграції з рішеннями для моніторингу, такими як Prometheus, Grafana і ELK Stack (Elasticsearch, Logstash, Kibana). Це дозволяє розробникам відслідковувати продуктивність системи, аналізувати журнали та виявляти потенційні проблеми на ранніх стадіях, що є критично важливим для підтримки надійності та стабільності розподілених додатків.

Щодо тестування, Java має розвинену екосистему для автоматизації тестування, включаючи фреймворки, такі як JUnit і Mockito. Це дозволяє реалізувати юніт-тестування, інтеграційне тестування та тестування на рівні системи, що є необхідним для забезпечення якості та стабільності розподілених рішень. Застосування автоматизованого тестування підвищує впевненість у тому, що система відповідає вимогам і працює належним чином в умовах реальної експлуатації.

У підсумку, аналіз можливостей Java для будівництва розподілених систем демонструє, що ця мова програмування не лише має великий набір інструментів та бібліотек, але й активно підтримує новітні технології та підходи в розробці. Завдяки своїй простоті, гнучкості та потужним механізмам для роботи з мережею, Java залишається одним із найкращих виборів для створення надійних, масштабованих і безпечних розподілених систем у різних галузях. Від фінансових установ до медичних закладів та електронної комерції, Java продовжує задовольняти вимоги сучасного бізнесу, адаптуючись до змінюваних умов та технологічних викликів.

1.3 Постановка завдання та вибір інструментів розробки

Постановка завдання для розробки розподіленої системи на основі Java є критично важливим етапом, який визначає успіх усього проекту. Основним завданням є створення ефективної, масштабованої та надійної системи, яка здатна обслуговувати велику кількість користувачів, підтримувати асинхронні запити та забезпечувати високу продуктивність. У рамках цього завдання необхідно чітко визначити вимоги до системи, зокрема, які функції вона повинна виконувати, які бізнес-процеси автоматизувати і які дані обробляти. Залежно від специфіки проекту, це можуть бути системи електронної комерції, управління даними, фінансові платформи, або системи управління вмістом.

На початковому етапі важливо зібрати інформацію про цільову аудиторію та її потреби. Визначення основних функціональних вимог, таких як можливість реєстрації та аутентифікації користувачів, управління профілями, обробка платежів, інтеграція з зовнішніми сервісами та забезпечення високого рівня безпеки, дозволяє створити детальну специфікацію системи. Важливо також врахувати не функціональні вимоги, які стосуються продуктивності, надійності, безпеки, масштабованості та зручності користування. Наприклад, система повинна бути спроектована так, щоб витримувати великі навантаження та швидко реагувати на запити користувачів, навіть у пікові години.

Вибір інструментів для розробки системи є наступним важливим кроком. Java, як мова програмування, надає потужний набір бібліотек та фреймворків, які можуть суттєво спростити процес розробки. Одним із найбільш популярних фреймворків для створення розподілених систем є Spring, який пропонує широкий спектр функцій для управління додатками, а також інтеграції з різними технологіями. Spring Boot, зокрема, є чудовим вибором для швидкого налаштування мікросервісів завдяки своїй здатності автоматично конфігурувати проекти на основі залежностей, що включаються.

Основною метою вибору Spring є забезпечення високого рівня абстракції та спрощення налаштування. Spring пропонує такі компоненти, як Spring MVC для створення веб-додатків, Spring Data для роботи з базами даних та Spring Security для забезпечення безпеки. Завдяки використанню Dependency Injection (DI) та Aspect-Oriented Programming (AOP), Spring дозволяє зменшити зв'язність між компонентами, що покращує тестування та обслуговування коду.

Для реалізації розподіленої архітектури, Spring Cloud надає необхідні інструменти для управління мікросервісами, включаючи механізми реєстрації, маршрутизації, управління конфігурацією та обробки відмов. Інтеграція з такими сервісами, як Netflix Eureka, Ribbon та Zuul, дозволяє

реалізувати високоавтоматизовані рішення для балансування навантаження та резервування, що є критично важливим для забезпечення високої доступності системи [11].

Додатково, важливо також врахувати інтеграцію з контейнеризацією, такою як Docker, що дозволяє легко розгортати мікросервіси в різних середовищах. Це, у свою чергу, підвищує портативність додатків та дозволяє реалізувати ефективні CI/CD процеси для автоматизації розгортання. Використання Kubernetes у поєднанні зі Spring також забезпечує можливості для управління кластеризованими сервісами, що значно спрощує обробку великої кількості запитів у розподілених системах.

Окрім того, вибір Spring для розробки розподіленої системи забезпечує легкість у реалізації тестування. Завдяки вбудованим механізмам для юніт-тестування, таких як Spring Test, розробники можуть легко створювати автоматизовані тести, що гарантують правильність функціонування системи на кожному етапі розробки. Це особливо важливо для розподілених систем, де кількість компонентів та їх взаємодій може призводити до складнощів у підтримці якості коду.

Отже, метою даної роботи є розробка розподіленої системи з використанням мови програмування Java та фреймворку Spring, яка відповідатиме сучасним вимогам до масштабованості, надійності та продуктивності. Для досягнення цієї мети необхідно виконати такі завдання:

1. Аналіз предметної області та вибір інструментів:
   1. Дослідити існуючі підходи до побудови розподілених систем.
   2. Оцінити доступні технології та інструменти (бібліотеки, протоколи комунікації, системи черг повідомлень) з точки зору їхньої відповідності вимогам до системи.
   3. Обрати оптимальний стек технологій для реалізації ефективної та масштабованої системи.
2. Проєктування архітектури розподіленої системи:

   1. Розробити архітектуру, яка відповідатиме визначеним функціональним і нефункціональним вимогам (висока доступність, масштабованість, стійкість до відмов).
   2. Обрати підхід до організації взаємодії між компонентами (наприклад, мікросервіси, клієнт-серверна модель, подієво-орієнтована архітектура).
   3. Визначити ключові компоненти системи та їхні функції.
3. Програмна реалізація:
   1. Розробити основні компоненти системи, використовуючи Java та Spring (Spring Boot, Spring Cloud, Spring Jdbc тощо).
   2. Реалізувати механізми комунікації між компонентами, наприклад, через REST API, WebSocket, чи брокери повідомлень (Kafka, RabbitMQ).
   3. Забезпечити роботу зі сховищем даних для зберігання інформації.
4. Тестування та оптимізація:
   1. Провести функціональне тестування системи для виявлення і усунення помилок.
   2. Оптимізувати код та налаштування системи для покращення її характеристик.

2. ВИБІР ІНСТРУМЕНТІВ ДЛЯ РОЗВ’ЯЗАННЯ ЗАДАЧІ

2.1 Вибір архітектури для реалізації розподіленої системи

Архітектура розподілених систем є складним, але ключовим елементом при розробці програмних рішень, що функціонують у середовищі, де компоненти системи можуть бути розташовані на різних фізичних або віртуальних машинах. У таких системах важливо, щоб усі частини працювали разом для досягнення загальної мети, забезпечуючи надійність, продуктивність та масштабованість. Основною метою розподіленої архітектури є створення ефективного середовища, в якому окремі компоненти можуть взаємодіяти між собою, незалежно від їх фізичного розташування.

Однією з основних характеристик розподілених систем є їхня модульність, що дозволяє розробникам створювати окремі компоненти, які можуть бути замінені або оновлені без значних змін у загальному функціонуванні системи. Це досягається через використання мікросервісної архітектури, в якій кожен мікросервіс відповідає за конкретну функціональність, наприклад, управління користувачами, обробку замовлень або управління продуктами. Такий підхід дозволяє легко масштабувати систему, адже кожен мікросервіс може бути розгорнутий незалежно в залежності від навантаження.

У розподілених системах важливо також забезпечити надійний зв'язок між компонентами. Для цього використовуються різні протоколи та технології, такі як HTTP, REST, WebSocket, gRPC та інші, які дозволяють передавати дані між сервісами. Вибір протоколу залежить від специфіки системи та вимог до продуктивності. Наприклад, для високошвидкісної передачі даних може використовуватися gRPC, який оптимізований для роботи з потоками даних, тоді як REST-інтерфейси є популярними завдяки простоті використання та широкій підтримці.

Однією з основних проблем, з якими стикаються розподілені системи, є затримка в передачі даних і мережеві збої. У зв'язку з цим архітектура повинна включати механізми для обробки помилок, що дозволяють системі продовжувати функціонувати навіть за несприятливих умов. Це може включати повторні спроби виконання запитів, кешування даних для зменшення навантаження на мережу та реалізацію асинхронних комунікацій для поліпшення швидкості обробки запитів.

Отже, архітектура розподілених систем на основі Java повинна бути гнучкою, масштабованою, надійною та безпечною. Використання сучасних технологій, таких як Spring Framework, контейнеризація, мікросервіси та інструменти моніторингу, дозволяє створювати потужні рішення, які відповідають вимогам бізнесу та забезпечують високу продуктивність. Завдяки модульному підходу та можливості легкого масштабування, розподілені системи можуть адаптуватися до змінюваних умов та підтримувати розвиток бізнесу в умовах швидко змінюваного середовища.

Розробка архітектури розподілених систем також передбачає врахування принципів проектування, таких як розподіл відповідальності, обмеження побічних ефектів та принципи SOLID. Ці принципи допомагають створити систему, яка є зрозумілою, легкою в супроводі та надійною. Наприклад, принцип єдиного відповідальності (Single Responsibility Principle) вказує на те, що кожен мікросервіс повинен відповідати лише за одну функціональність, що спрощує його тестування та обслуговування. Таким чином, при внесенні змін до одного мікросервісу не виникає ризиків для роботи інших частин системи.

Отже, проаналізувавши всі етапи розробки розподіленої системи, була обрана **мікросервісна архітектура**, яка дозволяє розділити функціональність системи на незалежні компоненти, кожен з яких виконує окрему задачу.
Основні характеристики обраної архітектури:

1. Розподіл на мікросервіси:

Кожен компонент системи реалізує конкретну бізнес-логіку, працює автономно і може розвиватися незалежно від інших. Це забезпечує модульність та полегшує підтримку.

2. Комунікація через синхронні та асинхронні протоколи:
   - Для синхронної взаємодії використовується протокол HTTP/HTTPS, що дозволяє мікросервісам безпосередньо викликати один одного через API.
   - Для асинхронної передачі даних використовується брокер повідомлень, який дає змогу передавати події між сервісами без необхідності прямого зв'язку.
3. Подієво-орієнтований підхід:

   Частина комунікації між компонентами реалізована через генерацію та обробку подій. Це дозволяє сервісам реагувати на зміну стану системи без жорсткої залежності один від одного.
4. Сервісна ізоляція:

   Кожен сервіс розгортається окремо, має власні ресурси та дані, що дозволяє уникати конфліктів і підвищує надійність системи в цілому.

Переваги обраної архітектури:

1. Масштабованість:

   Мікросервіси можуть масштабуватися незалежно один від одного залежно від навантаження на кожен з них.
2. Гнучкість:

   Завдяки незалежності сервісів, можливо швидко додавати нові функції або змінювати існуючі без впливу на інші компоненти системи.
3. Стійкість до відмов:

   Збої в одному сервісі не призводять до повного виходу системи з ладу. Асинхронна комунікація через брокер повідомлень допомагає компенсувати тимчасову недоступність окремих сервісів.
4. Простота розробки та підтримки:

Кожен сервіс є відносно простим у реалізації та тестуванні завдяки чіткому визначенню його функціональності.

5. Можливість технологічного різноманіття:

   Для кожного мікросервісу можна використовувати окремі технології або фреймворки, найбільш відповідні для вирішення конкретного завдання.

6. Підтримка асинхронних процесів:

   Використання подій дозволяє мінімізувати затримки у виконанні завдань, забезпечуючи плавність і швидкість роботи системи.

2.2 Вибір методології та інструментів для реалізації розподіленої системи

Вибір методології та інструментів для реалізації розподілених систем є ключовим етапом у розробці, оскільки він впливає на загальну архітектуру, продуктивність, масштабованість і безпеку системи. Основним елементом цього вибору є впровадження Agile-методологій, які забезпечують гнучкість у процесі розробки, дозволяючи командам швидко реагувати на зміни вимог і зменшувати час виходу на ринок. Методології Scrum та Kanban є найбільш популярними у світі програмної розробки, оскільки вони фокусуються на постійному вдосконаленні, співпраці та управлінні проектами за допомогою невеликих ітерацій [5].

З точки зору інструментів, Spring Framework пропонує широкий спектр можливостей для реалізації розподілених систем. Важливою частиною є Spring Boot, який спрощує налаштування та запуск нових проектів, дозволяючи розробникам швидко створювати автономні додатки з вбудованими веб-серверами, такими як Tomcat або Jetty. Spring Boot також має потужну екосистему стартерів, які забезпечують готові рішення для

інтеграції з різними технологіями, такими як бази даних, системи обміну повідомленнями та RESTful веб-сервіси.

Ще одним важливим інструментом є Spring Cloud, який надає серію проектів для створення мікросервісних архітектур. Він включає в себе інструменти для управління конфігурацією, маршрутизації запитів, балансування навантаження, а також для забезпечення спостережуваності через сервіси моніторингу, такі як Spring Cloud Sleuth та Zipkin. Використання Spring Cloud дозволяє створювати гнучкі системи, які можуть адаптуватися до зміни навантаження та вимог бізнесу.

У контексті взаємодії між мікросервісами, важливо розглянути використання API Gateway, який забезпечує єдину точку входу для всіх клієнтських запитів [12] . Spring Cloud Gateway є оптимальним рішенням для реалізації API Gateway, забезпечуючи простий спосіб маршрутизації запитів до відповідних сервісів, а також можливості для аутентифікації, обробки помилок і контролю навантаження [13].

Вибір інструментів також повинен враховувати безпеку. Spring Security надає потужні засоби для захисту мікросервісів, забезпечуючи аутентифікацію та авторизацію через OAuth 2.0 та OpenID Connect. Це дозволяє реалізувати централізовану модель безпеки, яка спрощує управління доступом до ресурсів у розподіленій системі [14].

Крім того, для забезпечення високої продуктивності та масштабованості важливо використовувати механізми кешування, які можуть бути реалізовані за допомогою Spring Cache [15]. Це дозволяє зменшити навантаження на базу даних і покращити швидкість відповіді системи на запити користувачів.

Важливим аспектом є також інтеграція з контейнеризованими середовищами, такими як Docker та Kubernetes, що дозволяє автоматизувати процеси розгортання та масштабування мікросервісів. Spring Boot і Spring Cloud добре інтегруються з цими технологіями, забезпечуючи зручний спосіб управління контейнерами та автоматизації DevOps-процесів.

Загалом, вибір методології та інструментів для реалізації розподілених систем на базі Spring потребує комплексного підходу, що охоплює всі етапи розробки – від проектування до впровадження. Гнучкість Agile-методологій у поєднанні з потужними можливостями Spring Framework дозволяє створювати високоякісні, масштабовані та безпечні розподілені системи, що відповідають сучасним вимогам бізнесу та технологічного середовища.

Вибір правильних інструментів також включає в себе використання системи управління версіями, такої як Git, для контролю за змінами в коді. Це дозволяє команді координувати роботу над проектом, відстежувати внесені зміни та повертатися до попередніх версій при необхідності. Використання Git у поєднанні з платформами для спільної роботи, такими як GitHub або GitLab, забезпечує можливість проведення код-рев'ю, автоматизації тестування та інтеграції, що є важливими аспектами сучасної розробки.

Не менш важливою є робота з даними в розподілених системах. Використання реляційних баз даних (наприклад, PostgreSQL) та NoSQL-рішень (таких як MongoDB) дозволяє реалізувати гнучкі стратегії зберігання даних, відповідно до вимог про швидкість і типи запитів. Spring Data надає потужні абстракції для роботи з різними типами баз даних, що дозволяє зменшити обсяг рутинного коду, пов'язаного з доступом до даних [16].

Крім того, важливим є використання API-дизайну, заснованого на REST або GraphQL, що дозволяє створювати чіткі, документовані та легко інтегруємі API для взаємодії між мікросервісами та з зовнішніми системами. Spring MVC і Spring WebFlux надають потужні можливості для реалізації RESTful API, включаючи обробку запитів, валідацію даних, управління помилками та реалізацію безпеки.

Загалом, реалізація розподілених систем на базі Spring вимагає інтегрованого підходу до вибору методології та інструментів, який враховує не лише технологічні аспекти, а й потреби команди, бізнес-цілі та специфіку проекту . Система повинна бути гнучкою, масштабованою, безпечною та

готовою до змін, щоб успішно виконувати поставлені задачі в умовах швидко змінюваного технологічного середовища. Цей підхід забезпечить максимальну продуктивність та ефективність в реалізації розподілених систем, дозволяючи підприємствам адаптуватися до нових викликів і зберігати конкурентоспроможність на ринку.

Проаналізувавши існуючі інструменти розробки розподілених систем на базі Spring Framework, для і реалізації серверної частини розподіленої системи та її архітектури, було розглянуто і обґрунтовано вибір наступних компонентів:

1. Spring Framework та його модулі

   Spring є одним із найпотужніших і найпоширеніших фреймворків для розробки серверної частини додатків.

   - Spring Boot

     Забезпечує автоматичну конфігурацію та швидкий запуск додатків, знижуючи кількість рутинного коду. Це дозволяє зосередитися на бізнес-логіці замість налаштування інфраструктури. Spring Boot також надає вбудований вебсервер (Tomcat або Jetty), що спрощує розгортання.

     Переваги:

     - Швидкий старт проєкту
     - Автоматизація конфігурації компонентів
     - Висока інтеграція з іншими модулями Spring

   - Spring JDBC

     Використовується для взаємодії з базою даних. Цей модуль забезпечує спрощений підхід до виконання SQL-запитів і роботи з транзакціями [17].

     Переваги:

     - Зменшення кількості коду завдяки зручному API
     - Підтримка інтеграції з іншими модулями Spring, такими як Spring Data

2. Resiliency4j

Resiliency4j — бібліотека, що забезпечує відмовостійкість додатків шляхом реалізації шаблонів стійкості (Circuit Breaker, Retry, Rate Limiter тощо) [21].

Роль у розподіленій системі:

- Запобігає каскадним збоям при відмові одного з сервісів.
- Автоматично виконує повторні запити в разі тимчасових помилок.
- Забезпечує контроль навантаження на сервіси.

3. Кешування (Cache)

Використання кешу дозволяє знизити навантаження на базу даних і прискорити доступ до часто використовуваних даних. У контексті Spring можна інтегрувати такі інструменти, як Redis або Ehcache.

Переваги кешування:

- Зменшення затримок при запитах
- Підвищення масштабованості додатку
- Зменшення операцій на стороні бази даних

4. Kafka (брокер повідомлень

Apache Kafka є ключовим компонентом для обміну подіями в розподіленій системі. Вона дозволяє сервісам комунікувати через асинхронний обмін повідомленнями, що підвищує гнучкість і відмовостійкість системи.

Переваги Kafka:

- Висока продуктивність і здатність обробляти великий обсяг подій
- Гарантія доставки повідомлень
- Масштабованість і підтримка стійкості до відмов

Використання Kafka допомагає організувати подієво-орієнтовану архітектуру, де сервіси реагують на зміну стану системи в режимі реального часу [18].

5. PostgreSQL (реляційна база даних)

PostgreSQL є потужною реляційною базою даних з відкритим вихідним кодом, яка забезпечує високий рівень надійності, продуктивності та безпеки.

Причини вибору:

- Підтримка ACID-транзакцій для забезпечення консистентності даних
- Потужна система роботи зі складними запитами
- Можливість масштабування та підтримка розширень

3. ПРОГРАМНА РЕАЛІЗАЦІЯ РОЗПОДІЛЕНОЇ СИСТЕМИ

3.1 Розробка моделі інформаційної системи

На етапі розробки моделі інформаційної системи здійснюється формалізація основних компонентів, їхніх взаємозв'язків і процесів, які реалізовуються в межах системи. Створення моделі є важливим етапом, який дозволяє зрозуміти структуру системи, визначити основні потоки даних, а також забезпечити базу для реалізації програмних компонентів [19].

Моделювання включає розробку таких ключових артефактів:

- Діаграма прецедентів (Use Case Diagram):
  Відображає основні сценарії взаємодії користувачів і зовнішніх систем із компонентами розподіленої системи.
- Діаграма компонентів (Component Diagram)
  Визначає основні програмні модулі системи та їхні взаємозв'язки, включаючи клієнтські та серверні компоненти, брокери повідомлень, базу даних тощо.
- Діаграма послідовності (Sequence Diagram)
  Показує послідовність взаємодії між компонентами для виконання ключових бізнес-процесів.
- Діаграма потоків даних (Data Flow Diagram)
  Візуалізує передачу даних між різними модулями системи, включаючи входи, виходи та проміжні обробки [20].

У цьому розділі буде представлено наведені вище діаграми з описом їхньої ролі в системі. Вони слугують не лише засобом проєктування, а й інструментом для комунікації між розробниками, зацікавленими сторонами та іншими учасниками процесу. Моделі допоможуть забезпечити узгодженість між вимогами, архітектурою та реалізацією, а також створити основу для подальшого розвитку системи.

Варто почати з діаграми прецедентів (Use Case Diagram), вона відображає основні сценарії використання системи, в яких беруть участь як кінцеві користувачі, так і зовнішні системи. Діаграма (рис. 3.1.1) демонструє, які функціональні можливості забезпечує система та як користувачі взаємодіють із нею.

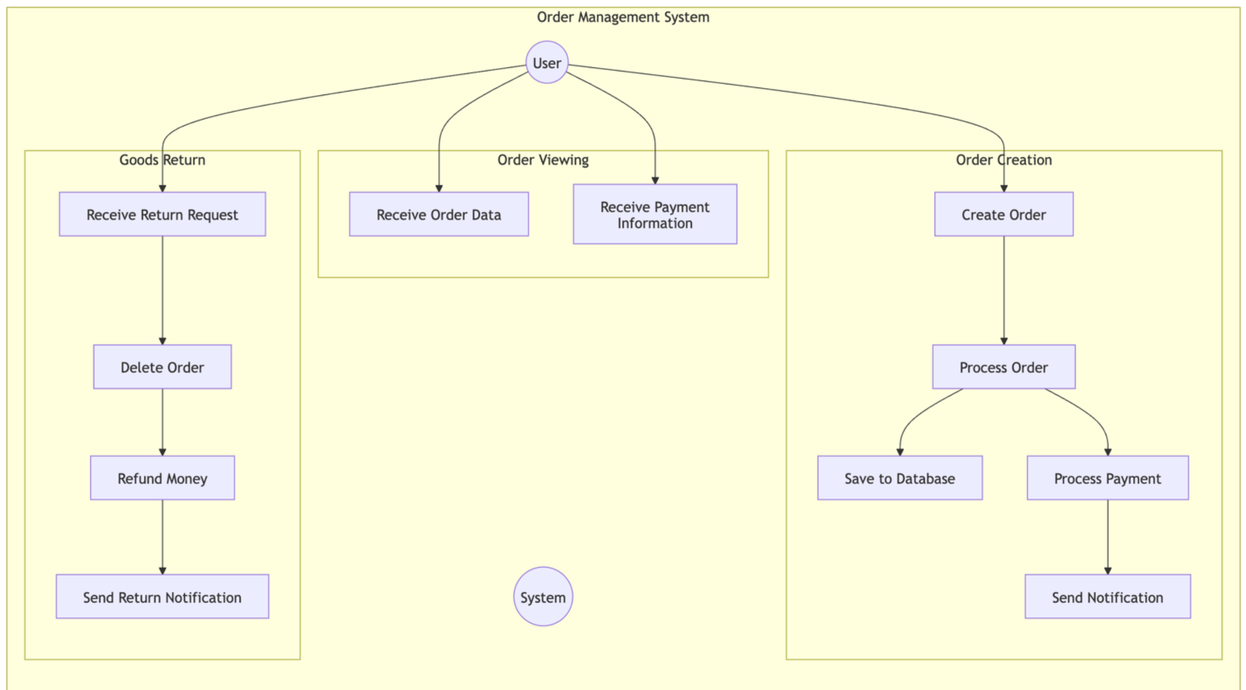


Рисунок 3.1.1 – Діаграма прецедентів (Use Case Diagram)

Наступним важливим етапом є створення діаграми компонентів (рис. 3.1.2), вона представляє ключові програмні модулі системи, їхні ролі та зв’язки. У цій діаграмі видно, як компоненти взаємодіють між собою, наприклад, через HTTP-запити або брокер повідомлень, а також їхній зв’язок із базою даних.

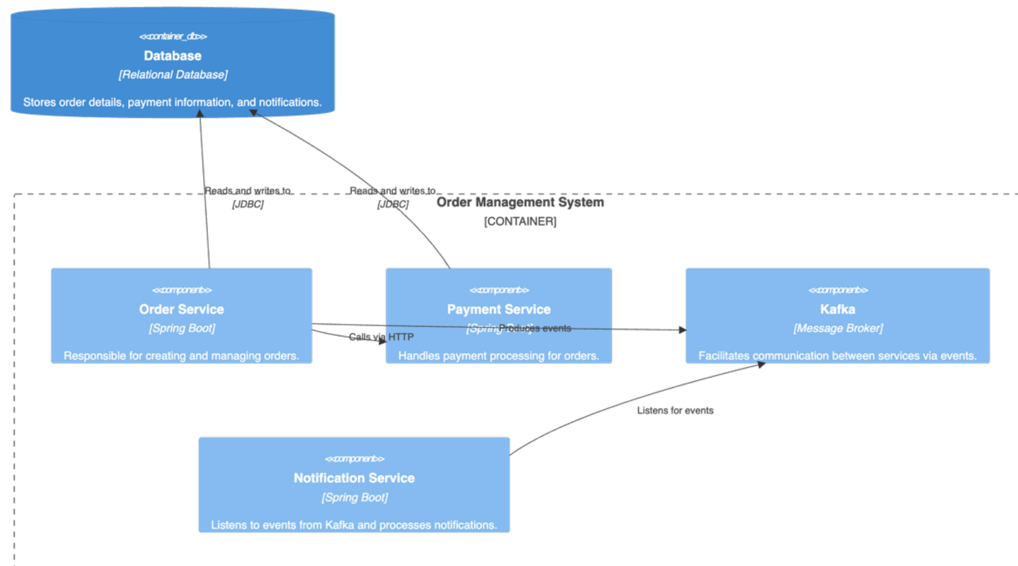


Рисунок 3.1.2 – Діаграма компонентів (Component Diagram)

Наступна по важливості діаграма - це діаграма послідовності (Sequence Diagram). Ця діаграма (рис. 3.1.3) демонструє послідовність взаємодії між компонентами для виконання конкретних бізнес-процесів, таких як створення замовлення, проведення оплати або надсилання сповіщення користувачу.

І остання діаграма, яка візуалізує, як дані переміщуються між компонентами, зокрема, між серверною частиною, базою даних і брокером повідомлень. Ця діаграма (рис. 3.1.4) ілюструє потоки інформації у системі, забезпечуючи розуміння того, як обробляються та зберігаються дані. Вона має назву діаграма потоків даних (Data Flow Diagram)

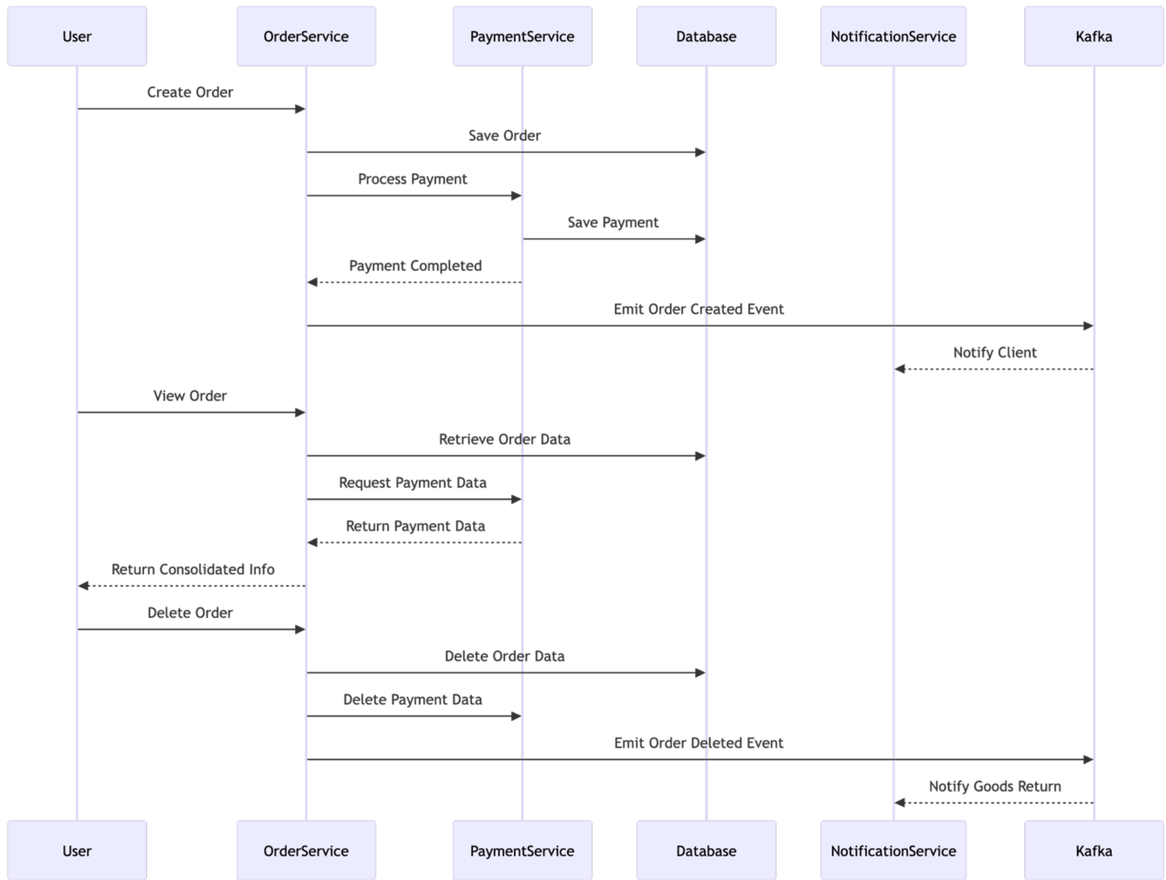


Рисунок 3.1.3 – Діаграма послідовності (Sequence Diagram)

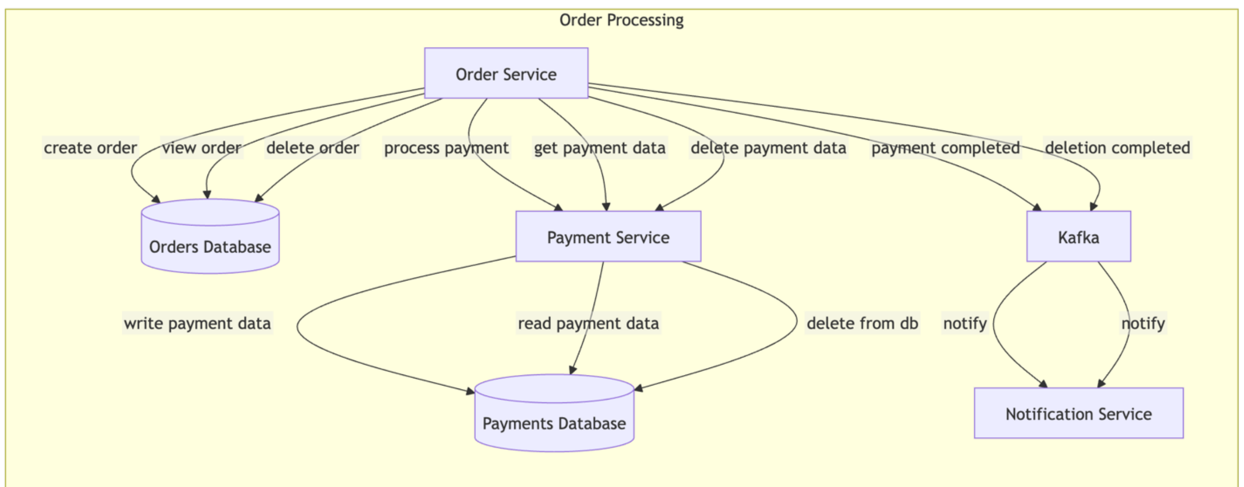


Рисунок 3.1.4 – Діаграма потоків даних (Data Flow Diagram)

Представлені діаграми забезпечують комплексне уявлення про архітектуру розподіленої системи, полегшують її аналіз і сприяють чіткішому розумінню ключових процесів. Вони є важливим інструментом для узгодження між проєктувальниками, розробниками та іншими зацікавленими сторонами.

3.2 Програмна реалізація розподіленої системи

У процесі розробки розподілених систем особливу увагу необхідно приділяти взаємодії між основними компонентами, адже ефективна комунікація та чітке розмежування відповідальностей між сервісами є запорукою стабільної роботи всієї системи. Побудова такої архітектури дозволяє забезпечити масштабованість, гнучкість та простоту підтримки додатків. Одним із ключових етапів розробки є детальне планування дизайну та архітектури системи, адже саме це визначає успішність реалізації. Для досягнення цієї мети було створено низку діаграм, які відображають взаємодію компонентів, основні бізнес-процеси та технічні аспекти роботи системи. Таке попереднє моделювання дозволило уникнути потенційних помилок та забезпечити гармонійну інтеграцію сервісів.

У цій роботі було реалізовано три основні мікросервіси, кожен з яких виконує свої унікальні завдання та відповідає за певний аспект бізнес-логіки. Кожен сервіс побудований з урахуванням принципів мікросервісної архітектури, що включає модульність, автономність компонентів і використання стандартизованих протоколів для обміну даними.

Payment-Service

Сервіс, відповідальний за виконання платіжних операцій. Його основне завдання — опрацювання запитів на оплату товарів, повернення коштів та отримання інформації про платежі. Для реалізації цієї функціональності він інтегрується зі стороннім API для виконання фінансових операцій. Після завершення транзакції інформація про платіж зберігається у базі даних. Цей

сервіс надає три основні ендпоінти: createPayment, refundPayment та getPayment.

Notification-Service

Сервіс, призначений для обробки подій із системи обміну повідомленнями Kafka. Залежно від типу події, сервіс генерує та надсилає електронні листи з інформацією про замовлення. Наприклад, клієнт отримує сповіщення про успішну оплату товару, невдалу спробу оплати, успішне повернення коштів або невдалий процес повернення. Цей компонент забезпечує зворотний асинхронний зв'язок із користувачем та покращує взаємодію з системою.

Order-Service

Основний сервіс, який забезпечує управління замовленнями. Він обробляє запити на створення замовлень, повернення замовлень та отримання детальної інформації про замовлення. Для здійснення платіжних операцій або отримання інформації про платежі Order-Service взаємодіє з Payment-Service через HTTP. Успішно оброблені замовлення зберігаються у базі даних. Окрім цього, Order-Service генерує події про стан замовлення та відправляє їх у Kafka, щоб Notification-Service міг забезпечити відповідне інформування користувача.

Таким чином, усі компоненти системи взаємодіють, утворюючи єдину екосистему, яка забезпечує виконання ключових бізнес-процесів. У цьому розділі буде детально описано технічну реалізацію кожного з цих компонентів, основні нюанси розробки та ключові моменти їхньої інтеграції.

Під час процесу створення розподіленої системи одним із перших і головних етапів є налаштування інфраструктури, яка забезпечить коректну роботу всіх компонентів. Важливо, щоб кожен сервіс мав доступ до необхідних ресурсів, таких як база даних, система обміну повідомленнями та інші залежності. Для реалізації цього етапу у даній роботі було обрано технологію контейнеризації Docker, яка дозволяє ізолювати додатки та їх залежності в контейнерах, забезпечуючи їхню портативність і

передбачуваність роботи. В рамках реалізації даної розподіленої системи, немає необхідності запускати ресурси на окремих, віддалених серверах, тому контейнеризація необхідних компонентів і запуск на локальній машині – це одна з ключових задач сервісу Docker. Ця технологія дозволяє створити стабільне середовище для всіх мікросервісів системи. Завдяки Docker Compose вдалося описати конфігурацію усієї інфраструктури в одному файлі. Це включає базу даних Postgres, брокер повідомлень Kafka, а також інші компоненти, необхідні для роботи сервісів. Кожен контейнер працює незалежно, що дозволяє легко масштабувати окремі сервіси та підтримувати систему в цілому.

Для налаштування бази даних було створено скрипт ініціалізації, який автоматично виконується при запуску контейнера Postgres. Цей скрипт створює необхідні таблиці для зберігання даних замовлень і платежів, а також встановлює початкові структури для коректної роботи мікросервісів. Нижче наведено приклади конфігураційного файлу docker-compose.yml для запуску всіх необхідних компнентів:

```yaml
version: '3.8'

services:
  postgres:
    image: postgres:latest
    container_name: postgres_db
    environment:
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
      POSTGRES_DB: mydatabase
    ports:
      - "5432:5432"
    volumes:
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql

  kafka:
    image: confluentinc/cp-kafka:latest
    container_name: kafka_broker
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

```
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181

volumes:
  postgres_data:
    driver: local
```

Після налаштування інфраструктури наступним кроком стала розробка основних компонентів системи. Першим компонентом, реалізація якого була розпочата, є *payment-service*. Вибір цього модуля для старту розробки обґрунтований його критичною роллю в загальній архітектурі системи. Саме payment-service відповідає за здійснення платежів, обробку повернень та зберігання інформації про транзакції, що є основою для взаємодії між іншими сервісами. Усі наступні компоненти залежать від цього сервісу, тому його реалізація є ключовим етапом.

Контроллер Payment-Service

Важливо, щоб дизайн API був інтуїтивно зрозумілим і відповідав принципам REST. Правильно обрані маршрути для ендпоінтів полегшують інтеграцію з іншими системами та забезпечують стандартизовану взаємодію між компонентами. Для payment-service було створено контроллер з трьома основними ендпоінтами:

1. Створення платежу - забезпечує виконання нових транзакцій.
2. Отримання інформації про платіж - дозволяє дізнатися деталі про вже виконаний платіж.
3. Видалення платежу - забезпечує обробку повернень коштів.

Нижче наведено реалізацію контроллера:

```
@RestController
@RequestMapping("/payments")
@RequiredArgsConstructor
public class PaymentController {

    private final PaymentServiceImpl paymentService;
```

```
    @PostMapping
    public ResponseEntity<Payment> createPayment(
            @RequestHeader("UserId") UUID userId,
            @RequestBody PaymentRequest paymentRequest) {
        return ResponseEntity.ok(paymentService.createPayment(userId,
paymentRequest));
    }

    @GetMapping("/{id}")
    public ResponseEntity<Payment> getPayment(
            @RequestHeader("UserId") UUID userId,
            @PathVariable String id) {
        return ResponseEntity.ok(paymentService.getPaymentById(userId, id));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Payment> deletePayment(
            @RequestHeader("UserId") UUID userId,
            @PathVariable String id) {
        return ResponseEntity.ok(paymentService.deletePayment(userId, id));
    }
}
```

Цей контроллер демонструє базову структуру REST API.

- Маршрути вибрані відповідно до принципу *Resource-Based URLs*: кожна операція працює з ресурсом payments.
- HTTP-методи відповідають CRUD операціям: POST для створення, GET для отримання інформації, DELETE для видалення.
- RequestHeader з UserId забезпечує передачу ідентифікатора користувача, що дозволяє сервісу обробляти запити індивідуально для кожного клієнта.

Цей контроллер є основою для взаємодії між клієнтом і payment-service, забезпечуючи логіку виконання транзакцій та обробки даних. Наступним кроком є розробка бізнес-логіки сервісу.

Для реалізації основної бізнес-логіки компоненту було створено сервісний клас PaymentServiceImpl, який імплементує інтерфейс PaymentService, де створений контракт методів які мають бути реалізовані. Він відповідає за обробку запитів на створення платежів, повернення коштів та отримання інформації про платежі. Кожна операція має чітко визначений алгоритм виконання, що забезпечує коректну роботу сервісу.

Створення платежу

Метод createPayment виконує необхідні кроки для обробки нового платежу:

```
@Override
public Payment createPayment(UUID userId, PaymentRequest request) {
        var processorResponse =
paymentProcessor.processPayment(request.getFirstName(),
request.getLastName(), request.getCardNumber(), request.getAmount());

        var payment = Payment.builder()
                .id(UUID.randomUUID().toString())
                .cardNumber(request.getCardNumber())
                .firstName(request.getFirstName())
                .lastName(request.getLastName())
                .amount(request.getAmount())
                .status(processorResponse.isSuccess() ? SUCCESS.name() :
FAILED.name())
                .transactionId(processorResponse.getTransactionId())
                .userId(userId.toString())
                .build();

        paymentRepository.save(payment);
        return payment;
}
```

Цей метод починає роботу з виклику процесора платежів PaymentProcessor, який інтегрується зі сторонньою платіжною системою. На основі результатів процесора створюється об’єкт Payment, який включає:

- Дані користувача (ім’я, прізвище, номер картки).
- Унікальний ідентифікатор транзакції.
- Суму платежу.
- Статус (SUCCESS або FAILED).

Після створення, дані зберігаються у базі за допомогою репозиторію PaymentRepository.

Отримання інформації про платіж

Метод getPaymentById надає можливість отримати деталі про конкретний платіж за ідентифікатором та користувачем:

```
@Override
public Payment getPaymentById(UUID userId, String id) {
    return paymentRepository.findPayment(id, userId.toString());
}
```

Реалізація є досить простою, оскільки основну роботу виконує репозиторій. Метод забезпечує доступ до необхідних даних без додаткової обробки.

Видалення платежу та повернення коштів

Метод deletePayment виконує складнішу логіку, яка включає взаємодію зі стороннім API для обробки повернення коштів:

```
@Override
public Payment deletePayment(UUID userId, String id) {
    Payment payment = getPaymentById(userId, id);
    if (payment == null) {
        throw new PaymentNotFoundException("Payment not found.");
    }

    PaymentResponse refundResponse =
paymentProcessor.refundPayment(payment.getTransactionId());
    payment.setStatus(refundResponse.isSuccess() ? REFUNDED.name() :
FAILED.name());

    paymentRepository.deletePayment(id, userId.toString());
    return payment;
}
```

Логіка методу складається з таких етапів:

1. Отримання даних про платіж із бази через метод getPaymentById.
2. Перевірка наявності платежу. Якщо дані не знайдено, виконується генерація виключення PaymentNotFoundException, який буде оброблений в загальному обробнику помилок.
3. Виклик процесора платежів для виконання повернення коштів (refundPayment).
4. Оновлення статусу платежу (REFUNDED або FAILED) залежно від результату.
5. Видалення платежу з бази даних через репозиторій.

Ця логіка дозволяє ефективно працювати з операціями повернення коштів, інтегруючись із платіжною системою та забезпечуючи актуальність даних у базі. Такий підхід до реалізації бізнес-логіки дозволяє чітко структурувати процеси, розділити обов’язки між компонентами та забезпечити стабільність і масштабованість системи.

Наступним етапом реалізації є створення компонента для взаємодії з базою даних. PaymentRepository виконує ці задачі, надаючи методи для збереження, пошуку та оновлення записів про платежі. Цей компонент

побудований на основі NamedParameterJdbcTemplate, що дозволяє гнучко працювати з SQL-запитами, передаючи параметри у вигляді іменованих змінних. Нижче наведено пояснення кожного методу:

Метод deletePayment

Метод оновлює статус платежу, позначаючи його як "DELETED".

```
@CircuitBreaker(name = "database")
public void deletePayment(String id, String userId) {
    final String sql = "UPDATE Payments SET status = 'DELETED' WHERE
paymentId = :paymentId AND userId = :userId";
    var params = new MapSqlParameterSource()
            .addValue("paymentId", id)
            .addValue("userId", userId);
    jdbcTemplate.update(sql, params);
}
```

У даному методі використовується SQL-запит для оновлення статусу платежу. Завдяки параметрам :paymentId та :userId забезпечується прив’язка операції до конкретного користувача, що підвищує точність і безпеку роботи з базою даних.

Метод findPayment

Метод findPayment використовується для пошуку конкретного платежу за його ідентифікатором та ідентифікатором користувача.

```
@CircuitBreaker(name = "database")
public Payment findPayment(String id, String userId) {
        final String sql = "SELECT * FROM Payments WHERE paymentId =
:paymentId AND userId = :userId";
        var params = new MapSqlParameterSource()
                .addValue("paymentId", id)
                .addValue("userId", userId);
        return jdbcTemplate.queryForObject(sql, params, new
BeanPropertyRowMapper<>(Payment.class));
}
```

Цей метод повертає об’єкт Payment, використовуючи SQL-запит для отримання даних із таблиці Payments.

Метод save

Метод save відповідає за збереження нового платежу в базі даних.

```
@CircuitBreaker(name = "database")
public void save(Payment payment) {
    final String sql = """
                INSERT INTO Payments (paymentId, cardNumber, firstName,
lastName, amount, status, transactionId, userId)
                VALUES (:paymentId, :cardNumber, :firstName, :lastName,
:amount, :status, :transactionId, :userId)
```

```
            """;
    SqlParameterSource params = new BeanPropertySqlParameterSource(payment);
    jdbcTemplate.update(sql, params);
}
```

Цей метод використовує SQL-запит для вставки нового запису в таблицю Payments. Завдяки використанню BeanPropertySqlParameterSource значення параметрів автоматично прив’язуються до полів об’єкта Payment.

Як можна було побачити, у всіх методах репозиторію застосовується анотація @CircuitBreaker. Вона належить до бібліотеки Resilience4j і забезпечує контроль стабільності з'єднання з базою даних.

Анотація @CircuitBreaker виконує такі функції:

- Захист від помилок: якщо під час виконання запиту до бази даних виникають часті помилки (наприклад, через перевантаження системи), механізм Circuit Breaker тимчасово припиняє спроби з'єднання.
- Моніторинг стану з'єднання: після закінчення певного часу або кількості успішних спроб з'єднання стан повертається до "нормального", і запити до бази даних відновлюються.
- Покращення стабільності системи: у разі проблем із базою даних інші сервіси продовжують працювати без перебоїв [21].

У випадку цього компонента конфігурація для Circuit Breaker була задана у файлі application.properties

```
# Resilience4j Circuit Breaker Configuration for payment-api
resilience4j.circuitbreaker.instances.payment-api.slidingWindowSize=10
resilience4j.circuitbreaker.instances.payment-api.failureRateThreshold=50
resilience4j.circuitbreaker.instances.payment-api.waitDurationInOpenState=10s

# Resilience4j Circuit Breaker Configuration for database
resilience4j.circuitbreaker.instances.database.slidingWindowSize=10
resilience4j.circuitbreaker.instances.database.failureRateThreshold=40
resilience4j.circuitbreaker.instances.database.waitDurationInOpenState=15s
```

Вона визначає основну поведінку під час виникнення помилок при роботі з ресурсами.

Payment-Service, як було вже зазначено, займається створенням, оновленням та отриманням інформації про платежі. Проте, процес обробки

замовлення часто пов'язаний із необхідністю миттєвого інформування користувачів про його результати. Наприклад, успішна оплата замовлення та створення замовлення має супроводжуватися підтвердженням на електронну пошту, а у разі помилки – детальним описом проблеми, щоб користувач міг оперативно зреагувати. Саме для таких задач розроблений компонент – Notification-Service.

Notification-Service є важливою ланкою системи, адже він дозволяє забезпечити зворотний зв'язок із користувачами в автоматизованому режимі. Цей сервіс побудований як Kafka Consumer, що обробляє події, які генеруються іншими сервісами, зокрема Order-Service. Наприклад, після успішного чи невдалого замовлення Order-Service відправляє відповідну подію у Kafka, а Notification-Service приймає її, інтерпретує і надсилає електронний лист користувачеві.

Ця архітектура забезпечує низку важливих переваг:

- Масштабованість. Завдяки Kafka Notification-Service легко масштабувати для обробки великої кількості подій.
- Незалежність. Компонент працює окремо від інших сервісів, що зменшує взаємозалежність і підвищує стабільність системи.
- Гнучкість. Можливість налаштовувати логіку обробки повідомлень для різних типів подій.

Конфігурація Kafka

Перший крок у створенні Notification-Service – це налаштування Kafka. Для цього необхідно створити конфігураційний клас, який визначатиме параметри підключення до брокера, групи конс'юмерів і механізми десеріалізації даних.

```
@EnableKafka
@Configuration
public class KafkaConfig {

    @Bean
    public ConsumerFactory<String, Object> consumerFactory() {
```

```
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "email-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(props);
    }

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

Для передачі даних між сервісами через Kafka необхідно визначити формат подій, які оброблятиме Notification-Service. Модель події має наступний вигляд:

```
@Data
public class Event {
    private String name;
    private String surname;
    private String email;
    private EventType eventType;

    public enum EventType {
        ORDER_PAYMENT_SUCCESS,
        ORDER_PAYMENT_FAILED,
        RETURN_SUCCESS,
        RETURN_FAILED
    }
}
```

Для структуризації даних, у моделі присутні поля:

- Ім'я та прізвище: використовуються для персоналізації повідомлень.
- Email: адреса, на яку буде відправлено лист.
- Тип події (EventType): визначає контекст повідомлення (успішна оплата, помилка тощо).

Наступним кроком є створення обробника подій, який прийматиме ці події через Kafka і виконуватиме надсилання електронних листів, адже обробка подій є центральною частиною Notification-Service. Цей компонент

приймає повідомлення з топіку Kafka, перевіряє їхній тип і ініціює відправку відповідних електронних листів.

```
@Component
@RequiredArgsConstructor
public class EventConsumer {

    private final EmailService emailService;

    @KafkaListener(topics = "event-topic", groupId = "email-group")
    public void consumeEvent(Event event) {
        emailService.processEvent(event);
    }
}
```

Для надсилання електронних листів у сервісі використовується компонент EmailService. Він відповідає за обробку подій і відправку відповідних повідомлень користувачам. В реалізації цього класу є метод processEvent, який отримує об'єкт події Event і на основі типу цієї події формує відповідний лист для користувача. Залежно від значення eventType, яке може бути одним із наступних:

- ORDER_PAYMENT_SUCCESS — замовлення успішно опрацьовано.
- ORDER_PAYMENT_FAILED — не вдалося обробити платіж.
- RETURN_SUCCESS — повернення товару успішне.
- RETURN_FAILED — повернення товару не вдалося.

Цей метод формує тему та текст листа, використовуючи ім'я та прізвище користувача для відправки персоналізованого повідомлення.

```
@Service
@RequiredArgsConstructor
public class EmailService {

    private final JavaMailSender mailSender;

    public void processEvent(Event event) {
        String subject;
        String message = switch (event.getEventType()) {
            case ORDER_PAYMENT_SUCCESS -> {
                subject = "Order Successful!";
                yield String.format("Dear %s %s, your order has been
successfully processed.", event.getName(), event.getSurname());
            }
            case ORDER_PAYMENT_FAILED -> {
                subject = "Payment Failed!";
```

```
                yield String.format("Dear %s %s, unfortunately, your payment
failed. Please try again.", event.getName(), event.getSurname());
            }
            case RETURN_SUCCESS -> {
                subject = "Return Successful!";
                yield String.format("Dear %s %s, your return has been
successfully processed.", event.getName(), event.getSurname());
            }
            case RETURN_FAILED -> {
                subject = "Return Failed!";
                yield String.format("Dear %s %s, unfortunately, your return
could not be processed. Please contact support.", event.getName(),
event.getSurname());
            }
            default -> throw new IllegalArgumentException("Unsupported event
type");
        };

        sendEmail(event.getEmail(), subject, message);
    }

    private void sendEmail(String to, String subject, String text) {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setTo(to);
        message.setSubject(subject);
        message.setText(text);
        mailSender.send(message);
    }
}
```

Цей сервіс інтегрується з JavaMailSender, що є частиною бібліотеки spring-boot-starter-mail. Цей компонент спрощує роботу з електронною поштою, автоматизуючи налаштування поштового клієнта та процес відправки листів [22]. Для конфігурації цього компоненту були використані наступні налаштування:

```
spring.mail.host=smtp.gmail.com
spring.mail.username=*******
spring.mail.password=*******
spring.mail.properties.mail.transport.protocol=smtp
spring.mail.properties.mail.smtp.port=25
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
```

Підсумовуючи, компонент notification-service відіграє важливу роль у розподіленій системі, адже забезпечуючи автоматичне сповіщення користувачів через електронну пошту на основі подій, що відбуваються в системі. З використанням Kafka та Spring Mail цей компонент має

можливість ефективно обробляти різноманітні типи подій і направляти повідомлення про важливі зміни. Цей сервіс не тільки підвищує зручність для користувачів, а й дозволяє бізнесу підтримувати прозорість взаємодії з клієнтами.

Тепер, коли було розглянуто, механізми сповіщень та роботи з платіжками, важливо зосередитися на основному компоненті системи — order-service. Цей сервіс є серцем платформи, оскільки він відповідає за обробку замовлень користувачів, включаючи їх створення, оплату та відміну. Саме в order-service зосереджена основна логіка бізнес-процесів, і він взаємодіє з іншими мікросервісами. Цей компонент не тільки керує життєвим циклом замовлень, але й відповідає за ініціацію подій для інших сервісів. Таким чином, це дуже важливий компонент, який пов’язує різні частини системи в єдину логіку.

Компонент OrderController є точкою входу для клієнтських запитів, які стосуються обробки замовлень. Цей контролер реалізує набір REST-ендпоінтів, що дозволяють користувачам створювати, переглядати та видаляти замовлення. Він є зручним інтерфейсом для взаємодії з бізнес-логікою OrderService.

Для для створення нового замовлення, необхідно використати createOrder ендпоінт, він приймає інформацію про замовлення та платіж. Після успішного створення повертає відповідь із кодом 201 (Created) та деталями створеного замовлення.

```
@PostMapping
public ResponseEntity<OrderDetails> createOrder(@RequestBody
OrderDetailsRequest orderDetails,
                                                @RequestHeader("UserId") UUID
userId,
                                                UriComponentsBuilder uriBuilder) {
    final var createdOrder = orderService.createOrder(orderDetails.order(),
orderDetails.payment(), userId);
    return
ResponseEntity.created(uriBuilder.path("/orders/{id}").build(createdOrder.get
Order().getId())).body(createdOrder);
}
```

Цей ендпоінт дозволяє отримати деталі конкретного замовлення за його ідентифікатором. Повертає замовлення, якщо воно належить зазначеному користувачу.

```
@GetMapping("/{id}")
public ResponseEntity<OrderDetails> getOrder(
        @PathVariable long id,
        @RequestHeader("UserId") UUID userId
) {
    return ResponseEntity.ok(orderService.getOrder(id, userId));
}
```

Для видалення замовлень і подальшого повернення коштів, існує ендпоінт deleteOrder, він приймає ідентифікатор замовлення і після успішного видалення повертає видалене замовлення і 200 (Ok) статус код.

Ключовим сервісним компонентом є OrderServiceImpl, він реалізує основну бізнес-логіку для управління замовленнями. Його методи відповідають за створення, отримання та видалення замовлень. У коді реалізовано взаємодію з іншими компонентами, такими як репозиторії, сервіси платежів, а також конвертація даних і публікація подій у Kafka. Оскільки структура коду схожа на інші компоненти, розглянемо лише ключові аспекти, специфічні для цього сервісу.

Бізнес логіка створення замовлення включає в себе наступні кроки:

- Перевірка на наявність продукту в каталозі за допомогою ProductRepository.
- Збереження замовлення у статусі PENDING у базу даних через OrderRepository.
- Виконання платежу за допомогою PaymentService.
- Оновлення замовлення відповідно до статусу платежу PAYMENT_SUCCESS або PAYMENT_FAILED.
- Публікація події в меседж брокер Kafka для подальшої обробки в notification-service

Під час виконання операції отримання замовлення, виконується наступні дії:

- Завантаження замовлення з бази даних через OrderRepository.
- Завантаження інформації про платіж через PaymentService.

- Повернення об'єкту OrderDetails, який включає всю інформацію про замовлення.

І також для видалення замовлення:

- Виконується запит до бази даних для отримання замовлення.
- Ініціюється повернення коштів через метод refundPayment у PaymentService.
- Якщо повернення успішне, статус замовлення змінюється на DELETED.
- У Kafka публікується подія про повернення коштів.

Особливу увагу в цьому сервісі приділено інтеграції з Kafka через OrderEventPublisher. Він є спеціалізованим компонентом для публікації подій, пов’язаних із замовленнями, у Kafka. Основна мета цього компонента — відправляти повідомлення про ключові події, такі як виконання платежу або повернення коштів, до топіка Kafka, де ці події будуть оброблятися іншим сервісом.

```
@Component
@RequiredArgsConstructor
public class OrderEventPublisher {
    private final KafkaTemplate<String, OrderEvent> kafkaTemplate;

    public void publishPaymentEvent(Order order, PaymentResponse
paymentResponse, UUID userId) {
        OrderEvent orderEvent = createPaymentEvent(order, paymentResponse,
EventType.PAYMENT_EVENT, userId);
        kafkaTemplate.send("payment-events", orderEvent);
    }

    public void publishRefundEvent(Order order, PaymentResponse
refundResponse, UUID userId) {
        OrderEvent refundEvent = createPaymentEvent(order, refundResponse,
EventType.REFUND_EVENT, userId);
        kafkaTemplate.send("payment-events", refundEvent);
    }

    private OrderEvent createPaymentEvent(Order order, PaymentResponse
paymentResponse, EventType eventType, UUID userId) {
        return OrderEvent.builder()
                .orderId(order.getId())
                .userId(userId)
                .cardNumber(paymentResponse.getCardNumber())
```

```
                .transactionId(paymentResponse.getTransactionId())
                .amount(paymentResponse.getAmount())
                .status(paymentResponse.getStatus())
                .eventType(eventType.toString())
                .build();
    }

    private enum EventType {
        PAYMENT_EVENT,
        REFUND_EVENT
    }
}
```

Не менш важливим компонентом у order-service є репозиторій, що відповідає за роботу з базою даних для зберігання та обробки замовлень. Цей компонент реалізує інтерфейс OrderRepository і використовує NamedParameterJdbcTemplate для виконання SQL-запитів. Репозиторій дозволяє створювати нові замовлення, отримувати інформацію про них, позначати замовлення як видалені, а також оновлювати їхній статус або пов'язану платіжну інформацію. Як і у випадку payment-service, тут також використовується @CircuitBreaker анотація, що забезпечує стабільність сервісу навіть у разі тимчасових збоїв.

```
@Repository
@RequiredArgsConstructor
public class OrderRepositoryImpl implements OrderRepository {
    private final NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    @CircuitBreaker(name = "database")
    public long createOrder(OrderDb order, UUID userId) {
        return new SimpleJdbcInsert(jdbcTemplate.getJdbcTemplate())
                .withTableName("Orders")
                .usingGeneratedKeyColumns("orderid")
                .executeAndReturnKey(new
BeanPropertySqlParameterSource(order))
                .longValue();
    }

    @Override
    @CircuitBreaker(name = "database")
    public OrderDb getOrder(long id, UUID userId) {
        final var params = Map.of("orderId", id, "userId",
userId.toString());
        return jdbcTemplate.queryForObject(
                "SELECT * FROM Orders WHERE orderId = :orderId AND userId =
:userId",
                params,
                new BeanPropertyRowMapper<>(OrderDb.class)
        );
```

```
    }

    @Override
    @CircuitBreaker(name = "database")
    public void markOrderAsDeleted(long id, UUID userId) {
        final var params = Map.of("orderId", id, "userId",
userId.toString());
        jdbcTemplate.update(
                "UPDATE Orders SET status = 'DELETED' WHERE orderId =
:orderId AND userId = :userId",
                params
        );
    }

    @Override
    @CircuitBreaker(name = "database")
    public void updateOrderPayment(OrderDb orderDb, String userId) {
        String query = """
                UPDATE Orders
                SET
                    paymentId = :paymentId,
                    status = :status
                WHERE orderId = :orderId AND userId = :userId;
                """;
        Map<String, Object> params = Map.of(
                "paymentId", orderDb.getPaymentId(),
                "status", orderDb.getStatus(),
                "orderId", orderDb.getOrderId(),
                "userId", userId
        );
        jdbcTemplate.update(query, params);
    }
}
```

На завершення розгляду order-service можна сказати, що цей компонент є одним із ключових у загальній архітектурі системи. Його логіка роботи охоплює повний цикл обробки замовлення: від створення запису в базі даних, перевірки доступності товару, обробки платежів, до генерації подій, які передаються іншим компонентам. Завдяки чіткій структурі та інтеграції з іншими сервісами, такими як payment-service та notification-service, order-service виконує роль центру управління процесами в системі.

Завершивши демонстрацію ключових елементів реалізації, логічно перейти до наступного важливого етапу — тестування. Це дозволить переконатися, що всі компоненти працюють коректно, а взаємодія між ними відповідає очікуванням. Тестування також дає змогу виявити та виправити можливі

помилки на ранніх етапах, забезпечуючи стабільність та надійність системи в цілому.

3.3 Тестування та верифікація системи

У розподілених системах, де кожен компонент має свою чітко визначену функціональність, тестування є одним із найважливіших етапів розробки. Воно забезпечує впевненість у тому, що система відповідає очікуванням з точки зору функціональності, продуктивності та стабільності. Без якісного тестування існує ризик, що навіть дрібні недоліки окремих компонентів можуть призвести до критичних збоїв у роботі системи загалом [22].

Тестування дає змогу виявити помилки на ранніх етапах, запобігти їх накопиченню, а також забезпечити високу якість взаємодії між компонентами. Особливо це важливо в сервіс-орієнтованій архітектурі, де сервіси взаємодіють через чітко визначені контракти, такі як API або події. Крім функціональності, перевірка охоплює такі аспекти, як відповідність бізнес-вимогам, стійкість до помилок, масштабованість і відповідь на нестандартні сценарії використання.

Процес тестування можна умовно розділити на два ключові етапи: тестування окремих компонентів і end-to-end тестування системи в цілому [22]. Перший етап зосереджений на перевірці роботи кожного сервісу окремо, включаючи коректність його логіки та відповідність контрактам. Другий етап дає змогу оцінити, як усі компоненти взаємодіють між собою, і забезпечити правильність виконання бізнес-процесів, що охоплюють одразу кілька сервісів.

Процес розпочинається із компонентного тестування. Спершу буде перевірено notification-service, оскільки його робота критично залежить від отриманих подій і правильної обробки повідомлень. Наступним кроком стане тестування payment-service, яке забезпечує обробку платіжних транзакцій.

Завершальним етапом компонентного тестування стане перевірка order-service, який координує роботу всієї системи на рівні замовлень.

Після перевірки кожного окремого сервісу необхідно виконати end-to-end тестування, яке дозволяє оцінити функціонування всієї системи в реальних умовах. Цей етап забезпечує впевненість у тому, що сервіси коректно взаємодіють між собою та відповідають очікуванням кінцевих користувачів.

Перед початком тестування, важливо забезпечити ізольоване і стабільне середовище для коректної роботи системи. Для цього в першу чергу необхідно виконати команду у терміналі для запуску попередньо створених контейнерів, для цього потрібно виконати команду docker-compose up. Після того як усі контейнери запущені, варто подбати про ізольованість між компонентами, для цього необхідно запустити їх на різних портах, так наприклад notification-service запускається на порту 8081, payment-service на 8082, а order-service — на 8083, що забезпечує чітку маршрутизацію запитів під час тестування та роботи всієї системи.

Для перевірки функціональності сервісів зручно використовувати Postman — потужний інструмент для надсилання HTTP-запитів. Завдяки Postman можна легко створювати запити до API, перевіряти відповіді, а також автоматизувати деякі аспекти тестування.

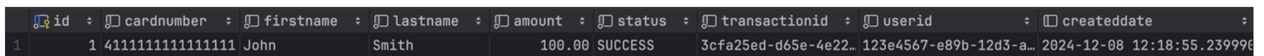
Почнемо з сервісу payment-service, а саме з ендпоінта для створення платежу, виконаємо запит за допомогою Postman і проаналізуємо респонс, а також стан бази даних після виконання цієї операції. Запит виглядає наступним чином:

```
curl -X POST http://localhost:8082/payments \
-H "Content-Type: application/json" \
-H "UserId: 123e4567-e89b-12d3-a456-426614174000" \
-d '{
  "cardNumber": "4111111111111111",
  "firstName": "John",
  "lastName": "Smith",
  "amount": 100.0
}'
```

Після виконання цього запиту, сервіс повертає наступну відповідь:

```json
{
  "id": "1",
  "cardNumber": "4111111111111111",
  "firstName": "John",
  "lastName": "Smith",
  "amount": 100.0,
  "status": "SUCCESS",
  "transactionId": "3cfa25ed-d65e-4e22-85a9-ad181b68a868",
  "userId": "123e4567-e89b-12d3-a456-426614174000"
}
```

Проаналізувавши цей json, можна побачити що запит на створення платежу виконався успішно, для перевірки коректності роботи бізнес логіки, перевіримо стан бази даних, щоб упевнитися, що платіж збережений. Виконаємо SQL-запит безпосередньо до бази (рис. 3.1.1).



	id	cardnumber	firstname	lastname	amount	status	transactionid	userid	createddate
1	1	4111111111111111	John	Smith	100.00	SUCCESS	3cfa25ed-d65e-4e22…	123e4567-e89b-12d3-a…	2024-12-08 12:18:55.23999

Рисунок 3.3.1 «Доданий запис після операції створення платежу»

Таким чином, запит працює коректно: створений платіж успішно записаний до бази даних із відповідним статусом і деталями.

Наступним етапом необхідно перевірити, чи повертає сервіс коректну інформацію про існуючий платіж, для цього виконаємо запит з методом GET для отримання інформації щодо замовлення по його ідентифікатору:

```
curl -X GET http://localhost:8082/payments/1 \
-H "Content-Type: application/json" \
-H "UserId: 123e4567-e89b-12d3-a456-426614174000"
```

```json
{
  "id": "1",
  "cardNumber": "4111111111111111",
  "firstName": "John",
  "lastName": "Smith",
  "amount": 100.0,
  "status": "SUCCESS",
  "transactionId": "3cfa25ed-d65e-4e22-85a9-ad181b68a868",
```

```
  "userId": "123e4567-e89b-12d3-a456-426614174000"
}
```

Інформація щодо замовлення не змінилася, тому можна зробити висновок що логіка цього ендпоінту коректна.

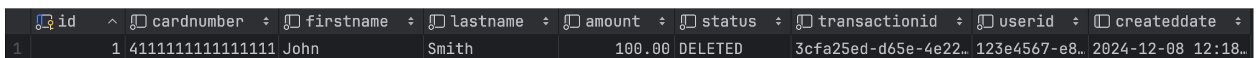
Останній ендпоінт який необхідно протестувати, це видалення замовлення. Бізнес логіка цього ендпоінту насправді не видаляє з бази даних цей запис, а просто маркує його як видалений, а також оновлює статус що кошти повернуті, тому виконаємо запит до цього ендпоінту вказавши метод запиту як DELETE.

```
curl -X DELETE http://localhost:8082/payments/1 \
-H "Content-Type: application/json" \
-H "UserId: 123e4567-e89b-12d3-a456-426614174000"
```

І в якості відповіді сервіс надіслав наступний json:

```
{
  "id": "1",
  "cardNumber": "4111111111111111",
  "firstName": "John",
  "lastName": "Smith",
  "amount": 100.0,
  "status": "DELETED",
  "transactionId": "3cfa25ed-d65e-4e22-85a9-ad181b68a868",
  "userId": "123e4567-e89b-12d3-a456-426614174000"
}
```

Після аналізу цієї відповіді, можна побачити що статус замовлення змінився на DELETED. Щоб упевнитись в коректності роботи цієї кінцевої точки, проведемо аналіз бази даних (рис.3.3.2):



	id	cardnumber	firstname	lastname	amount	status	transactionid	userid	createddate
1	1	4111111111111111	John	Smith	100.00	DELETED	3cfa25ed-d65e-4e22…	123e4567-e8…	2024-12-08 12:18…

Рисунок 3.3.2 – «Видалений запис після операції видалення»

Підсумовуючи тестування сервісу payment-service за допомогою Postman, можна зробити висновок що цей сервіс працює коректно. Створення, отримання та видалення платежів працюють відповідно до очікувань, і результати чітко відображаються в базі даних.

Наступним етапом буде тестування notification-service. Цей компонент є ключовим компонентом системи, що відповідає за відправку електронних листів на основі подій, отриманих з Kafka. Для перевірки його роботи необхідно не лише створити тестові події, але й налаштувати реальну інтеграцію з поштовим сервісом.

Один із найзручніших способів тестування — використання існуючого облікового запису Gmail. Однак важливо пам'ятати, що Gmail потребує налаштування спеціального "App Password", оскільки прямий доступ до пароля для застосунків за замовчуванням заборонений.

Після налаштування такого акаунту, можна перейти до тестування, першим кроком необхідно за допомогою терміналу виконати команду для створення події і надсилання її в Kafka. Насправді інструментарій роботи з цим меседж брокером доволі різноманітний і якщо ваша система потребує надсилання великої кількості різних подій, варто розглянути desktop клієнти для взаємодії з Kafka. Це дає можливість створювати події у більш зручному форматі, через UI. Але для випадку тестування невеликої кількості подій в нашому випадку буде достатньо функціоналу доступного в терміналі. Для створення тестової події використаємо команду kafka-console-producer для надсилання в потрібну тему.

```
kafka-console-producer --broker-list localhost:9092 --topic notification-
events {"name": "John", "surname": "Smith", "email": "john.smith@gmail.com",
"eventType": "ORDER_PAYMENT_SUCCESS"}
```

Очікуваним результатом після обробки цієї події буде відправка листа на електронну адресу, яка була вказана з тілі події, а саме john.smith@gmail.com. Тому перевіримо електронну скриньку (рис.3.3.3)

Order Successful! External Inbox ×

Dear John Smith, your order has been successfully processed.

Рисунок 3.3.3 – «Надісланий лист після обробки події»

Після перевірки можна побачити, що лист був відправлений з необхідної темою і вмістом, з правильно підставленими ім’ям та прізвищем.

Тепер по аналогії, створимо одразу три події, з наступними статусами: ORDER_PAYMENT_FAILED, RETURN_SUCCESS, RETURN_FAILED і перевіримо листи які були відправлені на передану електронну пошту.

```
kafka-console-producer --broker-list localhost:9092 --topic notification-events

{"name": "John", "surname": "Smith", "email": "john.smith@gmail.com", "eventType": "ORDER_PAYMENT_FAILED"}

{"name": "John", "surname": "Smith", "email": "john.smith@gmail.com", "eventType": "RETURN_SUCCESS"}

{"name": "John", "surname": "Smith", "email": "john.smith@gmail.com", "eventType": "RETURN_FAILED"}
```

Після надсилання цих подій Notification Service має обробити кожну з них та надіслати відповідні електронні листи (рис.3.3.4 - 3.3.6):

Payment Failed! External Inbox ×

Dear John Smith, unfortunately, your payment failed. Please try again.

Рисунок 3.3.4 – «Лист про невдалу спробу платежу»

Return Successful! External Inbox ×

Dear John Smith, your return has been successfully processed.

Рисунок 3.3.5 – «Лист про вдалу спробу повернення товару»

Return Failed! External Inbox ×

Dear John Smith, unfortunately, your return could not be processed. Please contact support.

Рисунок 3.3.6 – «Лист про невдалу спробу повернення товару»

Таким чином, Notification Service успішно продемонстрував свою здатність обробляти події, що надходять з Kafka, та генерувати відповідні електронні листи для користувачів. Тому можна підвести підсумок, що цей компонент відповідає вимогам до функціональності та готовий до інтеграції з іншими компонентами системи.

Коли всі залежні компоненти успішно протестовані (Notification Service, Payment Service), можна перейти до тестування всієї системи в цілому. End-to-end тестування дозволяє перевірити повний процес роботи Order Service, включаючи створення замовлення, отримання інформації про нього та видалення. Це тестування імітує реальні сценарії взаємодії користувача з системою, забезпечуючи перевірку інтеграції всіх компонентів (Order Service, Notification Service, Payment Service) та правильності роботи бази даних.

Почати варто зі створення нового замовлення. Для тестування цього необхідно використати HTTP POST-запит до /orders. У запиті передати дані про замовлення та платіжну інформацію. Запит буде мати наступний вигляд:

```
curl -X POST http://localhost:8080/orders \
-H "Content-Type: application/json" \
-H "UserId: 123e4567-e89b-12d3-a456-426614174000" \
-d '{
  "order": {
    "productId": 1,
    "amount": 2
  },
  "payment": {
    "name": "John",
    "surname": "Smith",
    "cardNumber": "1234567812345678"
  }
}'
```

В якості відповіді серверу отримано наступний json:


```json
{
  "order": {
    "id": 1,
    "productId": 1,
    "totalAmount": 2,
    "createdDate": "2024-12-07T10:00:00",
    "orderStatus": "PAYMENT_SUCCESS"
  },
  "paymentInfo": {
    "name": "John",
    "surname": "Smith",
    "cardNumber": "1234567812345678",
    "transactionId": "5b35469d-4a5d-403f-a5f3-a9d00736c807",
    "amount": 200.0
  }
}
```

Тепер необхідно виконати SQL запит для верифікації правильності створеного замовлення і інформації про нього (рис. 3.3.7)

	orderid	productid	userid	totalamount	status	createddate
1	1	1.00	123e4567-e89b-12d3-a456-426614174000	2.00	SUCCESS	2024-12-08 13:49:33.099164

Рисунок 3.3.7 – «Створений запис замовлення в базі даних»

Наступною кінцевою точкою для тестування, буде отримання замовлення. Для цього використовується HTTP GET-запит до /orders/{id}.

```
curl -X GET http://localhost:8080/orders/1 \
-H "UserId: 123e4567-e89b-12d3-a456-426614174000"
```

Сервер відсилає наступний json об’єкт:

```
{
  "order": {
    "id": 1,
    "productId": 1,
    "totalAmount": 2,
    "createdDate": "2024-12-07T10:00:00",
    "orderStatus": "PAYMENT_SUCCESS"
  },
  "paymentInfo": {
    "name": "John",
    "surname": "Smith",
    "cardNumber": "1234567812345678",
    "transactionId": "5b35469d-4a5d-403f-a5f3-a9d00736c807",
    "amount": 200.0
  }
}
```

І останнім кроком є тестування операції видалення замовлення, для перевірки видалення надсилається HTTP DELETE-запит до /orders/{id}.

```
curl -X DELETE http://localhost:8080/orders/1 \
-H "UserId: 123e4567-e89b-12d3-a456-426614174000"
```

Сервер надсилає наступну відповідь у форматі json:

```
{
  "order": {
    "id": 1,
    "productId": 1,
    "totalAmount": 2,
    "createdDate": "2024-12-07T10:00:00",
    "orderStatus": "DELETED"
  },
  "paymentInfo": {
    "name": "John",
    "surname": "Smith",
    "cardNumber": "1234567812345678",
    "transactionId": "5b35469d-4a5d-403f-a5f3-a9d00736c807",
    "amount": 200.0
  }
}
```


Таким чином, Order Service успішно пройшов тестування. Підсумовуючи, можна зробити наступний висновок - тестування є невід'ємною складовою процесу розробки сучасних програмних систем. Завдяки комплексному підходу, який охоплює як тестування окремих компонентів, так і end-to-end перевірку всієї системи, вдалося переконатися у її коректній роботі, надійності та готовності до використання у реальних умовах.

Ретельне тестування Notification Service підтвердило здатність системи обробляти події та надсилати коректні сповіщення на електронну пошту, що є ключовим аспектом для взаємодії з користувачем. Payment Service продемонстрував стійкість і правильність обробки платіжних транзакцій, забезпечуючи прозорість і точність роботи платіжної системи. Усі можливі сценарії — успішні платежі, невдалі транзакції та повернення коштів — були перевірені, що знижує ризики виникнення критичних помилок у реальному середовищі.

End-to-end тестування всієї системи стало кульмінацією цього процесу, перевіривши злагоджену роботу компонентів у реальних сценаріях. Процеси створення, отримання інформації та видалення замовлення, а також їх взаємодія з платіжним і нотифікаційним сервісами підтвердили цілісність

архітектури. Крім того, були перевірені всі зміни у базі даних, що забезпечило впевненість у збереженні цілісності даних.

ВИСНОВКИ

У даній роботі було проаналізовано ключові аспекти розробки розподілених систем, зосереджено увагу на створенні сервісно-орієнтованої архітектури, виборі інструментів та технологій для ефективної реалізації, а також забезпеченні високої якості системи шляхом тестування. Особливий акцент було зроблено на мікросервісній архітектурі, яка дозволяє забезпечити масштабованість, модульність і зручність підтримки програмних рішень.

Основні етапи, виконані в межах роботи**:**

1. Аналіз і вибір технологій.

   Було обґрунтовано вибір Java як основної мови програмування та Spring як фреймворка, який забезпечує зручну роботу з мікросервісами, інтеграцію з популярними інструментами (Kafka, PostgreSQL тощо) та автоматизацію конфігурацій. Крім того, було досліджено інструменти для забезпечення стійкості системи, такі як Resilience4j, а також брокер повідомлень Kafka для організації подійно-орієнтованої взаємодії між сервісами.

2. Проєктування архітектури системи.

   Сформовано архітектурну модель, яка відображає взаємодію компонентів системи, зокрема сервісів для обробки замовлень, платежів та сповіщень. Запропонована архітектура враховує принципи розподілених систем, включаючи горизонтальне масштабування, поділ відповідальності між сервісами та організацію асинхронної комунікації.

3. Програмна реалізація основних компонентів.

   У межах роботи було реалізовано три ключові компоненти: *Order Service*, *Payment Service* і *Notification Service*. Їх розробка включала створення бізнес-логіки, налаштування комунікацій між сервісами через Kafka, обробку HTTP-запитів і забезпечення взаємодії з базами даних. Важливим

аспектом стала розробка моделей даних і API, які гарантують узгодженість і зручність взаємодії між компонентами.

4. Тестування та верифікація системи.

   Було проведено повне тестування компонентів та всієї системи. Компонентне тестування дозволило перевірити роботу кожного сервісу окремо: коректність обробки подій Notification Service, виконання транзакцій Payment Service та операцій з замовленнями у Order Service. Завдяки end-to-end тестуванню вдалося підтвердити злагоджену роботу компонентів у межах повного життєвого циклу замовлення, включаючи створення, оновлення та видалення даних.

5. Розробка середовища розгортання.

   Було описано процес розгортання сервісів у Docker-контейнерах із використанням налаштування окремих портів для кожного компонента. Це забезпечило ізольованість середовищ, зручність тестування та підготовку до масштабованого розгортання в реальному середовищі.

Висновки та результати:

У результаті виконаної роботи було створено прототип розподіленої системи, який відповідає сучасним вимогам до продуктивності, масштабованості та стійкості. Вибір технологій і підходів дозволив забезпечити модульність системи, її гнучкість і здатність до подальшого розвитку. Тестування підтвердило надійність рішень, а запропоновані архітектурні рішення створили міцну основу для впровадження системи в реальному середовищі.

Дослідження підкреслило важливість комплексного підходу до розробки, що включає аналіз технологій, розробку архітектури, програмну реалізацію та тестування. Результати можуть бути використані як для практичного впровадження розробленої системи, так і як керівництво для розробників, які працюють над створенням розподілених систем із подібною архітектурою.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ali S., Alauldeen R., Ruaa A. What is Client-Server System: Architecture, Issues and Challenge of Client-Server System // HBRP Publication. 2020. № February.
2. Tihomirovs J., Grabis J. Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics // Information Technology and Management Science. 2017. Vol. 19, № 1.
3. Grosso W., Reilly P.O. Java RMI // October. 2001. № October.
4. Reis D. et al. Developing Docker and Docker-Compose Specifications: A Developers' Survey // IEEE Access. 2022. Vol. 10.
5. Al-Saqqa S., Sawalha S., Abdelnabi H. Agile software development: Methodologies and trends // International Journal of Interactive Mobile Technologies. 2020. Vol. 14, № 11.
6. Ottinger J.B., Lombardi A. Spring Security // Beginning Spring 6. 2024.
7. Lazzari L., Farias K. Event-driven architecture and REST architectural style: An exploratory study on modularity // Journal of Applied Research and Technology. 2023. Vol. 21, № 3.
8. Ali A.H., Abdullah M.Z. A survey on vertical and horizontal scaling platforms for big data analytics // International Journal of Integrated Engineering. 2019. Vol. 11, № 6.
9. Cunha P.R. da, Soja P., Themistocleous M. Blockchain for development: a guiding framework // Information Technology for Development. 2021. Vol. 27, № 3.
10. Setlow R., Black J., Blyth A. Effective Java // Addison-Wesley. 2006.
11. Gutierrez F. Spring Framework 5 // Pro Spring Boot 2. 2019.
12. Zuo X. et al. An API gateway design strategy optimized for persistence and coupling // Advances in Engineering Software. 2020. Vol. 148.

13. Deinum M., Cosmina I. Pro spring MVC with WebFlux: Web development in spring framework 5 and spring boot 2 // Pro Spring MVC with WebFlux: Web Development in Spring Framework 5 and Spring Boot 2. 2021.
14. Zhang F. et al. Design and implementation of energy management system based on spring boot framework // Information (Switzerland). 2021. Vol. 12, № 11.
15. Paschos G.S. et al. The role of caching in future communication systems and networks // IEEE Journal on Selected Areas in Communications. 2018. Vol. 36, № 6.
16. Xplenty. The SQL vs NoSQL Difference: MySQL vs MongoDB // Xplent Blog. 2017.
17. Bales D. Java Programming With Oracle JDBC // Java Programming With Oracle JDBC. 2002.
18. Raptis T.P. et al. Engineering Resource-Efficient Data Management for Smart Cities with Apache Kafka † // Future Internet. 2023. Vol. 15, № 2.
19. Berardi D., Calvanese D., De Giacomo G. Reasoning on UML class diagrams // Artif Intell. 2005. Vol. 168, № 1–2.
20. Bhatt B., Nandu M. An Overview of Structural UML Diagrams // International Research Journal of Engineering and Technology. 2021. Vol. 8, № 8.
21. Punithavathy E., Priya N. Auto retry circuit breaker for enhanced performance in microservice applications // International Journal of Electrical and Computer Engineering. 2024. Vol. 14, № 2.
22. Kirinuki H., Tanno H. Automating End-to-End Web Testing via Manual Testing // Journal of Information Processing. 2022. Vol. 30.
23. Xplenty. The SQL vs NoSQL Difference: MySQL vs MongoDB // Xplent Blog. 2017.
24. Загальна інформація про модуль Spring Mail // Електронна версія на сайті https://docs.spring.io/spring-boot/reference/io/email.html

25. Загальна інформація про бібліотеку Resiliency4j // Електронна версія на сайті https://resilience4j.readme.io/

ДОДАТКИ

Payment-service

Додаток А (PaymentController.java)

```
package org.example.paymentservice.api;

import lombok.RequiredArgsConstructor;
import org.example.paymentservice.model.Payment;
import org.example.paymentservice.model.PaymentRequest;
import org.example.paymentservice.service.PaymentServiceImpl;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.UUID;

@RestController
@RequestMapping("/payments")
@RequiredArgsConstructor
public class PaymentController {

    private final PaymentServiceImpl paymentService;

    @PostMapping
    public ResponseEntity<Payment> createPayment(
            @RequestHeader("UserId") UUID userId,
            @RequestBody PaymentRequest paymentRequest) {
        return          ResponseEntity.ok(paymentService.createPayment(userId,
paymentRequest));
    }

    @GetMapping("/{id}")
    public ResponseEntity<Payment> getPayment(
            @RequestHeader("UserId") UUID userId,
            @PathVariable String id) {
```

```
        return ResponseEntity.ok(paymentService.getPaymentById(userId, id));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Payment> deletePayment(
            @RequestHeader("UserId") UUID userId,
            @PathVariable String id) {
        return ResponseEntity.ok(paymentService.deletePayment(userId, id));
    }
}
```

Додаток B (WebConfig.java)

```
package org.example.paymentservice.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class WebConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Додаток C (PaymentProcessor.java)

```
package org.example.paymentservice.data;

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.example.paymentservice.model.PaymentResponse;
import org.springframework.stereotype.Component;

import java.util.UUID;

@Component
public class PaymentProcessor {

    @CircuitBreaker(name = "payment-api")
    public PaymentResponse processPayment(String firstName, String lastName,
String cardNumber, double amount) {
        String transactionId = UUID.randomUUID().toString();
```

```
        String status = "SUCCESS";

        return PaymentResponse.builder()
                .id(UUID.randomUUID().toString())
                .cardNumber(cardNumber)
                .firstName(firstName)
                .lastName(lastName)
                .amount(amount)
                .status(status)
                .transactionId(transactionId)
                .build();
    }

    @CircuitBreaker(name = "payment-api")
    public PaymentResponse refundPayment(String transactionId) {
        return PaymentResponse.builder()
                .status("SUCCESS")
                .transactionId(transactionId)
                .build();
    }
}
```

Додаток D (PaymentRepository.java)

```
package org.example.paymentservice.data;

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import lombok.RequiredArgsConstructor;
import org.example.paymentservice.model.Payment;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import
org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.stereotype.Repository;

@Repository
@RequiredArgsConstructor
public class PaymentRepository {

    private final NamedParameterJdbcTemplate jdbcTemplate;
```

```
    @CircuitBreaker(name = "database")
    public Payment findPayment(String id, String userId) {
        final String sql = "SELECT * FROM Payments WHERE paymentId = :paymentId AND userId = :userId";
        var params = new MapSqlParameterSource()
                .addValue("paymentId", id)
                .addValue("userId", userId);
        return jdbcTemplate.queryForObject(sql, params, new BeanPropertyRowMapper<>(Payment.class));
    }

    @CircuitBreaker(name = "database")
    public void deletePayment(String id, String userId) {
        final String sql = "UPDATE Payments SET status = 'DELETED' WHERE paymentId = :paymentId AND userId = :userId";
        var params = new MapSqlParameterSource()
                .addValue("paymentId", id)
                .addValue("userId", userId);
        jdbcTemplate.update(sql, params);
    }

    @CircuitBreaker(name = "database")
    public void save(Payment payment) {
        final String sql = """
                    INSERT INTO Payments (paymentId, cardNumber, firstName, lastName, amount, status, transactionId, userId)
                    VALUES (:paymentId, :cardNumber, :firstName, :lastName, :amount, :status, :transactionId, :userId)
                """;
        SqlParameterSource params = new BeanPropertySqlParameterSource(payment);
        jdbcTemplate.update(sql, params);
    }
}
```

Додаток Е (Payment.java)

```
package org.example.paymentservice.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
```

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Payment {
    private String id;
    private String cardNumber;
    private String firstName;
    private String lastName;
    private double amount;
    private String status;
    private String transactionId;
    private String userId;
}
```

Додаток F (PaymentRequest.java)

```
package org.example.paymentservice.model;

import lombok.Data;

@Data
public class PaymentRequest {
    private String cardNumber;
    private String firstName;
    private String lastName;
    private double amount;
}
```

Додаток G (PaymentResponse.java)

```
package org.example.paymentservice.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;

@Data
@Builder
@AllArgsConstructor
public class PaymentResponse {

    private String id;
    private String cardNumber;
    private String firstName;
```

```
    private String lastName;
    private double amount;
    private String status;
    private String transactionId;

    public boolean isSuccess() {
        return "SUCCESS" .equals(status);
    }
}
```

Додаток H (PaymentStatus.java)

```
package org.example.paymentservice.model;

public enum PaymentStatus {
    REFUNDED,
    SUCCESS,
    FAILED
}
```

Додаток I (PaymentService.java)

```
package org.example.paymentservice.service;

import org.example.paymentservice.model.Payment;
import org.example.paymentservice.model.PaymentRequest;

import java.util.UUID;

public interface PaymentService {
    Payment createPayment(UUID userId, PaymentRequest request);

    Payment getPaymentById(UUID userId, String id);

    Payment deletePayment(UUID userId, String id);
}
```

Додаток J (PaymentServiceImpl.java)

```
package org.example.paymentservice.service;


import lombok.RequiredArgsConstructor;
import org.example.paymentservice.PaymentNotFoundException;
import org.example.paymentservice.data.PaymentProcessor;
import org.example.paymentservice.data.PaymentRepository;
import org.example.paymentservice.model.Payment;
```

```
import org.example.paymentservice.model.PaymentRequest;
import org.example.paymentservice.model.PaymentResponse;
import org.springframework.stereotype.Service;

import java.util.UUID;

import static org.example.paymentservice.model.PaymentStatus.FAILED;
import static org.example.paymentservice.model.PaymentStatus.REFUNDED;
import static org.example.paymentservice.model.PaymentStatus.SUCCESS;

@Service
@RequiredArgsConstructor
public class PaymentServiceImpl implements PaymentService{
    private final PaymentRepository paymentRepository;
    private final PaymentProcessor paymentProcessor;

    @Override
    public Payment createPayment(UUID userId, PaymentRequest request) {
        var                         processorResponse                        =
paymentProcessor.processPayment(request.getFirstName(),
request.getLastName(), request.getCardNumber(), request.getAmount());

        var payment = Payment.builder()
                .id(UUID.randomUUID().toString())
                .cardNumber(request.getCardNumber())
                .firstName(request.getFirstName())
                .lastName(request.getLastName())
                .amount(request.getAmount())
                .status(processorResponse.isSuccess()   ?   SUCCESS.name()   :
FAILED.name())
                .transactionId(processorResponse.getTransactionId())
                .userId(userId.toString())
                .build();

        paymentRepository.save(payment);
        return payment;
    }

    @Override
    public Payment getPaymentById(UUID userId, String id) {
        return paymentRepository.findPayment(id, userId.toString());
    }
```

```
    @Override
    public Payment deletePayment(UUID userId, String id) {
        Payment payment = getPaymentById(userId, id);
        if (payment == null) {
            throw new PaymentNotFoundException("Payment not found.");
        }

        PaymentResponse refundResponse =
paymentProcessor.refundPayment(payment.getTransactionId());
        payment.setStatus(refundResponse.isSuccess() ? REFUNDED.name() :
FAILED.name());

        paymentRepository.deletePayment(id, userId.toString());
        return payment;
    }
}
```

Додаток K (PaymentNotFoundException.java)

```
package org.example.paymentservice;

public class PaymentNotFoundException extends RuntimeException {
    public PaymentNotFoundException(String message) {
        super(message);
    }
}
```

Додаток L (PaymentServiceApplication.java)

```
package org.example.paymentservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class PaymentServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(PaymentServiceApplication.class, args);
    }

}
```

Додаток M (application.properties)

```
spring.application.name=payment-service
spring.datasource.url=jdbc:postgresql://localhost:5432/payments
```

```
spring.datasource.username=user
spring.datasource.password=password

# Resilience4j Circuit Breaker Configuration for payment-api
resilience4j.circuitbreaker.instances.payment-api.slidingWindowSize=10
resilience4j.circuitbreaker.instances.payment-api.failureRateThreshold=50
resilience4j.circuitbreaker.instances.payment-api.waitDurationInOpenState=10s

# Resilience4j Circuit Breaker Configuration for database
resilience4j.circuitbreaker.instances.database.slidingWindowSize=10
resilience4j.circuitbreaker.instances.database.failureRateThreshold=40
resilience4j.circuitbreaker.instances.database.waitDurationInOpenState=15s
```

Notification-service

Додаток А (KafkaConfig.java)

```
package org.example.notificationservice.config;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import
org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;

import java.util.HashMap;
import java.util.Map;

@EnableKafka
@Configuration
public class KafkaConfig {

    @Bean
    public ConsumerFactory<String, Object> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "email-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
```

```
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(props);
    }

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

Додаток B (EventConsumer.java)

```
package org.example.notificationservice.consumer;

import lombok.RequiredArgsConstructor;
import org.example.notificationservice.model.Event;
import org.example.notificationservice.service.EmailService;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
@RequiredArgsConstructor
public class EventConsumer {

    private final EmailService emailService;

    @KafkaListener(topics = "event-topic", groupId = "email-group")
    public void consumeEvent(Event event) {
        emailService.processEvent(event);
    }
}
```

Додаток C (Event.java)

```
package org.example.notificationservice.model;

import lombok.Data;

@Data
public class Event {
    private String name;
```

```
    private String surname;
    private String email;
    private EventType eventType;

    public enum EventType {
        ORDER_PAYMENT_SUCCESS,
        ORDER_PAYMENT_FAILED,
        RETURN_SUCCESS,
        RETURN_FAILED
    }
}
```

Додаток D (EmailService.java)

```
package org.example.notificationservice.service;

import lombok.RequiredArgsConstructor;
import org.example.notificationservice.model.Event;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class EmailService {

    private final JavaMailSender mailSender;

    public void processEvent(Event event) {
        String subject;
        String message = switch (event.getEventType()) {
            case ORDER_PAYMENT_SUCCESS -> {
                subject = "Order Successful!";
                yield String.format("Dear %s %s, your order has been successfully processed.", event.getName(), event.getSurname());
            }
            case ORDER_PAYMENT_FAILED -> {
                subject = "Payment Failed!";
                yield String.format("Dear %s %s, unfortunately, your payment failed. Please try again.", event.getName(), event.getSurname());
            }
            case RETURN_SUCCESS -> {
                subject = "Return Successful!";
```

```
                yield String.format("Dear %s %s, your return has been successfully processed.", event.getName(), event.getSurname());
            }
            case RETURN_FAILED -> {
                subject = "Return Failed!";
                yield String.format("Dear %s %s, unfortunately, your return could not be processed. Please contact support.", event.getName(), event.getSurname());
            }
            default -> throw new IllegalArgumentException("Unsupported event type");
        };

        sendEmail(event.getEmail(), subject, message);
    }

    private void sendEmail(String to, String subject, String text) {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setTo(to);
        message.setSubject(subject);
        message.setText(text);
        mailSender.send(message);
    }
}
```

Додаток E (application.properties)

```
spring.application.name=notification-service
spring.mail.host=smtp.gmail.com
spring.mail.username=*******
spring.mail.password=*******
spring.mail.properties.mail.transport.protocol=smtp
spring.mail.properties.mail.smtp.port=25
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
```

Order-service

Додаток A (OrderController.java)

```
package org.example.orderservice.api;

import lombok.RequiredArgsConstructor;
import org.example.orderservice.model.OrderDetailsRequest;
```

```
import org.example.orderservice.model.OrderDetails;
import org.example.orderservice.service.OrderService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.util.UriComponentsBuilder;

import java.util.UUID;

@RestController
@RequiredArgsConstructor
@RequestMapping("/orders")
public class OrderController {
    private final OrderService orderService;

    @PostMapping
    public        ResponseEntity<OrderDetails>        createOrder(@RequestBody
OrderDetailsRequest orderDetails,
                                             @RequestHeader("UserId")     UUID
userId,
                                             UriComponentsBuilder  uriBuilder)
{
        final                var                createdOrder                =
orderService.createOrder(orderDetails.order(),        orderDetails.payment(),
userId);
        return
ResponseEntity.created(uriBuilder.path("/orders/{id}").build(createdOrder.get
Order().getId())).body(createdOrder);
    }

    @GetMapping("/{id}")
    public ResponseEntity<OrderDetails> getOrder(
            @PathVariable long id,
            @RequestHeader("UserId") UUID userId
    ) {
        return ResponseEntity.ok(orderService.getOrder(id, userId));
    }
```

```
    @DeleteMapping("/{id}")
    public ResponseEntity<OrderDetails> deleteOrder(
            @PathVariable long id,
            @RequestHeader("UserId") UUID userId
    ) {
        return ResponseEntity.ok(orderService.deleteOrder(id, userId));
    }
}
```

Додаток B (CustomConversionService.java)

```
package org.example.orderservice.config;

import org.example.orderservice.service.converter.OrderDbToOrderConverter;
import org.example.orderservice.service.converter.OrderToOrderDbConverter;
import org.springframework.core.convert.support.GenericConversionService;
import org.springframework.stereotype.Component;

@Component
public class CustomConversionService extends GenericConversionService {

    public CustomConversionService(
            OrderToOrderDbConverter orderToOrderDbConverter,
            OrderDbToOrderConverter orderDbToOrderConverter
    ) {
        addConverter(orderToOrderDbConverter);
        addConverter(orderDbToOrderConverter);
    }
}
```

Додаток C (KafkaConfig.java)

```
package org.example.orderservice.config;

import com.fasterxml.jackson.databind.JsonSerializer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.example.orderservice.events.model.OrderEvent;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;

import java.util.HashMap;
```

```
import java.util.Map;

@Configuration
public class KafkaConfig {
    @Bean
    public ProducerFactory<String, OrderEvent> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    @Bean
    public KafkaTemplate<String, OrderEvent> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

Додаток D (WebConfig.java)

```
package org.example.orderservice.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class WebConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Додаток E (OrderDb.java)

```
package org.example.orderservice.data.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
```

```
import lombok.NoArgsConstructor;

import java.util.Date;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class OrderDb {
    private long orderId;
    private int productId;
    private String paymentId;
    private String userId;
    private int totalAmount;
    private String status;
    private Date createdDate;
}
```

Додаток F (OrderRepository.java)

```
package org.example.orderservice.data;

import org.example.orderservice.data.model.OrderDb;

import java.util.UUID;

public interface OrderRepository {
    long createOrder(OrderDb order, UUID userId);

    OrderDb getOrder(long id, UUID userId);

    void markOrderAsDeleted(long id, UUID userId);

    void updateOrderPayment(OrderDb orderDb, String userId);
}
```

Додаток G (OrderRepositoryImpl.java)

```
package org.example.orderservice.data;

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import lombok.RequiredArgsConstructor;
import org.example.orderservice.data.model.OrderDb;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import
org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
```

```
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;

import java.util.Map;
import java.util.UUID;

@Repository
@RequiredArgsConstructor
public class OrderRepositoryImpl implements OrderRepository {
    private final NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    @CircuitBreaker(name = "database")
    public long createOrder(OrderDb order, UUID userId) {
        return new SimpleJdbcInsert(jdbcTemplate.getJdbcTemplate())
                .withTableName("Orders")
                .usingGeneratedKeyColumns("orderid")
                .executeAndReturnKey(new
BeanPropertySqlParameterSource(order))
                .longValue();
    }

    @Override
    @CircuitBreaker(name = "database")
    public OrderDb getOrder(long id, UUID userId) {
        final var params = Map.of("orderId", id, "userId",
userId.toString());
        return jdbcTemplate.queryForObject("SELECT * FROM Orders where
orderId = :orderId and userId = :userId", params, new
BeanPropertyRowMapper<>(OrderDb.class));
    }

    @Override
    @CircuitBreaker(name = "database")
    public void markOrderAsDeleted(long id, UUID userId) {
        final var params = Map.of("orderId", id, "userId",
userId.toString());
        jdbcTemplate.update("UPDATE Orders SET status = 'DELETED' where
orderId = :orderId and userId = :userId", params);
    }

    @Override
```

```
    @CircuitBreaker(name = "database")
    public void updateOrderPayment(OrderDb orderDb, String userId) {
        String query = """
                UPDATE Orders
                SET
                    paymentId   = :paymentId,
                    status      = :status
                WHERE orderId = :orderId AND userId = :userId;
                """;
        Map<String, Object> params = Map.of("status", orderDb.getStatus(),
                "id", orderDb.getProductId(),
                "paymentId", orderDb.getPaymentId(),
                "userId", userId);
        jdbcTemplate.update(query, params);
    }
}
```

Додаток H (ProductRepository.java)

```
package org.example.orderservice.data;

import org.example.orderservice.model.Product;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class ProductRepository {

    private final List<Product> availableProducts = List.of(
            new Product(1, 20, 2),
            new Product(2, 55, 10),
            new Product(3, 220, 5)
    );

    public Product getProductInfo(int id) {
        return availableProducts.stream()
                .filter(product -> product.getId() == id)
                .findFirst().orElseThrow(() -> new RuntimeException("Product
not found"));
    }
}
```

Додаток I (OrderEvent.java)

```
package org.example.orderservice.events.model;
```

```
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.UUID;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class OrderEvent {
    private long orderId;
    private UUID userId;
    private String cardNumber;
    private String transactionId;
    private double amount;
    private String status;
    private String eventType;
}
```

Додаток J (OrderEventPublisher.java)

```
package org.example.orderservice.events.producer;

import lombok.RequiredArgsConstructor;
import org.example.orderservice.events.model.OrderEvent;
import org.example.orderservice.model.Order;
import org.example.orderservice.model.PaymentResponse;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

import java.util.UUID;

@Component
@RequiredArgsConstructor
public class OrderEventPublisher {
    private final KafkaTemplate<String, OrderEvent> kafkaTemplate;

    public void publishPaymentEvent(Order order, PaymentResponse
paymentResponse, UUID userId) {
        OrderEvent orderEvent = createPaymentEvent(order, paymentResponse,
EventType.PAYMENT_EVENT, userId);
```

```
        kafkaTemplate.send("payment-events", orderEvent);
    }

    public void publishRefundEvent(Order order, PaymentResponse refundResponse, UUID userId) {
        OrderEvent refundEvent = createPaymentEvent(order, refundResponse, EventType.REFUND_EVENT, userId);
        kafkaTemplate.send("payment-events", refundEvent);
    }

    private OrderEvent createPaymentEvent(Order order, PaymentResponse paymentResponse, EventType eventType, UUID userId) {
        return OrderEvent.builder()
                .orderId(order.getId())
                .userId(userId)
                .cardNumber(paymentResponse.getCardNumber())
                .transactionId(paymentResponse.getTransactionId())
                .amount(paymentResponse.getAmount())
                .status(paymentResponse.getStatus())
                .eventType(eventType.toString())
                .build();
    }

    private enum EventType {
        PAYMENT_EVENT,
        REFUND_EVENT
    }
}
```

Додаток K (ProductNotFoundException.java)

```
package org.example.orderservice.exceptions;

public class ProductNotFoundException extends RuntimeException {
    public ProductNotFoundException(String message) {
        super(message);
    }
}
```

Додаток L (ProductOutOfStockException.java)

```
package org.example.orderservice.exceptions;

public class ProductOutOfStockException extends RuntimeException {
    public ProductOutOfStockException(String message) {
        super(message);
```

```
    }
}
```

Додаток M (Order.java)

```
package org.example.orderservice.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Order {
    private long id;
    private long productId;
    private int totalAmount;
    private Date createdDate;
    private OrderStatus orderStatus;
}
```

Додаток N (OrderDetails.java)

```
package org.example.orderservice.model;

import lombok.Data;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
public class OrderDetails {
    private final Order order;
    private final PaymentInfo paymentInfo;
}
```

Додаток O (OrderDetailsRequest.java)

```
package org.example.orderservice.model;

public record OrderDetailsRequest(OrderRequest order, PaymentInfo payment) {
}
```

Додаток P (OrderRequest.java)

```java
package org.example.orderservice.model;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class OrderRequest {
    private int productId;
    private int amount;
    private double price;
}
```

Додаток Q (OrderStatus.java)

```java
package org.example.orderservice.model;

public enum OrderStatus {
    PENDING,
    PAYMENT_FAILED,
    PAYMENT_SUCCESS,
    REFUND_FAILED,
    REFUND_SUCCESS
}
```

Додаток R (PaymentInfo.java)

```java
package org.example.orderservice.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class PaymentInfo {
    private String name;
    private String surname;
    private String cardNumber;
    private String transactionId;
    private double amount;
}
```

Додаток S (PaymentRequest.java)

```java
package org.example.orderservice.model;
```

```
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PaymentRequest {
    private String cardNumber;
    private String firstName;
    private String lastName;
    private double amount;
}
```

Додаток T (PaymentResponse.java)

```
package org.example.orderservice.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;

@Data
@Builder
@AllArgsConstructor
public class PaymentResponse {
    private String id;
    private String cardNumber;
    private String firstName;
    private String lastName;
    private double amount;
    private String status;
    private String transactionId;

    public boolean isSuccess() {
        return "SUCCESS" .equals(status);
    }
}
```

Додаток U (Product.java)

```
package org.example.orderservice.model;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class Product {
    private int id;
    private double price;
    private int count;
}
```

Додаток V (OrderDbToOrderConverter.java)

```
package org.example.orderservice.service.converter;

import org.example.orderservice.data.model.OrderDb;
import org.example.orderservice.model.Order;
import org.example.orderservice.model.OrderStatus;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

@Component
public class OrderDbToOrderConverter implements Converter<OrderDb, Order> {

    @Override
    public Order convert(OrderDb source) {
        return Order.builder()
                .orderStatus(OrderStatus.valueOf(source.getStatus()))
                .totalAmount(source.getTotalAmount())
                .createdDate(source.getCreatedDate())
                .id(source.getOrderId())
                .productId(source.getProductId())
                .build();
    }
}
```

Додаток W (OrderToOrderDbConverter.java)

```
package org.example.orderservice.service.converter;

import org.example.orderservice.data.model.OrderDb;
import org.example.orderservice.model.Order;
import org.example.orderservice.model.OrderStatus;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;
```

```
import static java.util.Objects.nonNull;

@Component
public class OrderToOrderDbConverter implements Converter<Order, OrderDb> {
    @Override
    public OrderDb convert(Order source) {
        return OrderDb.builder()
                .productId(source.getProductId())
                .totalAmount(source.getTotalAmount())
                .status(nonNull(source.getOrderStatus())                    ?
source.getOrderStatus().toString() : OrderStatus.CREATED.toString())
                .createdDate(source.getCreatedDate())
                .build();
    }
}
```

Додаток X (OrderService.java)

```
package org.example.orderservice.service;

import org.example.orderservice.model.OrderRequest;
import org.example.orderservice.model.OrderDetails;
import org.example.orderservice.model.PaymentInfo;

import java.util.UUID;

public interface OrderService {
    OrderDetails  createOrder(OrderRequest  order,  PaymentInfo  payment,  UUID
userId);

    OrderDetails getOrder(long id, UUID userId);

    OrderDetails deleteOrder(long id, UUID userId);
}
```

Додаток Y (OrderServiceImpl.java)

```
package org.example.orderservice.service;

import lombok.RequiredArgsConstructor;
import org.example.orderservice.exceptions.ProductNotFoundException;
import org.example.orderservice.exceptions.ProductOutOfStockException;
import org.example.orderservice.model.OrderRequest;
import org.example.orderservice.config.CustomConversionService;
import org.example.orderservice.data.OrderRepository;
import org.example.orderservice.data.ProductRepository;
```

```
import org.example.orderservice.data.model.OrderDb;
import org.example.orderservice.events.producer.OrderEventPublisher;
import org.example.orderservice.model.Order;
import org.example.orderservice.model.OrderDetails;
import org.example.orderservice.model.OrderStatus;
import org.example.orderservice.model.PaymentInfo;
import org.example.orderservice.model.PaymentRequest;
import org.example.orderservice.model.Product;
import org.springframework.stereotype.Service;

import java.util.Date;
import java.util.Objects;
import java.util.UUID;

import static java.lang.String.valueOf;
import static java.util.Objects.isNull;
import static org.example.orderservice.model.OrderStatus.PAYMENT_FAILED;
import static org.example.orderservice.model.OrderStatus.PAYMENT_SUCCESS;

@Service
@RequiredArgsConstructor
public class OrderServiceImpl implements OrderService {
    private final OrderRepository orderRepository;
    private final ProductRepository productRepository;
    private final CustomConversionService conversionService;
    private final PaymentService paymentService;
    private final OrderEventPublisher kafkaEventPublisher;

    @Override
    public OrderDetails createOrder(OrderRequest order, PaymentInfo payment,
UUID userId) {
        final var product = getProduct(order);

        final var orderDbBuilder = OrderDb.builder()
                .productId(order.getProductId())
                .totalAmount(order.getAmount())
                .userId(userId.toString())
                .createdDate(new Date())
                .status(OrderStatus.PENDING.name());
        final var orderId =
orderRepository.createOrder(orderDbBuilder.build(), userId);
        orderDbBuilder.orderId(orderId);
        final var paymentRequest = PaymentRequest.builder()
```

```
                .cardNumber(payment.getCardNumber())
                .firstName(payment.getName())
                .lastName(payment.getSurname())
                .amount(product.getPrice())
                .build();
        final var paymentResponse =
paymentService.processPayment(paymentRequest);
        final var orderDb = orderDbBuilder
                .paymentId(paymentResponse.getTransactionId())
                .status("FAILED".equals(paymentResponse.getStatus()) ?
PAYMENT_FAILED.name() : PAYMENT_SUCCESS.name())
                .build();
        orderRepository.updateOrderPayment(orderDb, userId.toString());

        final var savedOrder = conversionService.convert(orderDb,
Order.class);
        final var paymentInfo = conversionService.convert(paymentResponse,
PaymentInfo.class);
        kafkaEventPublisher.publishPaymentEvent(savedOrder, paymentResponse,
userId);
        return new OrderDetails(savedOrder, paymentInfo);
    }

    @Override
    public OrderDetails getOrder(long id, UUID userId) {
        final var orderDb = orderRepository.getOrder(id, userId);
        final var payment =
paymentService.getPayment(orderDb.getPaymentId());
        return new OrderDetails(conversionService.convert(orderDb,
Order.class), conversionService.convert(payment, PaymentInfo.class));
    }

    @Override
    public OrderDetails deleteOrder(long id, UUID userId) {
        final var orderDb = orderRepository.getOrder(id, userId);
        final var paymentResponse =
paymentService.refundPayment(orderDb.getPaymentId());
        if (paymentResponse.isSuccess())
            orderRepository.markOrderAsDeleted(id, userId);

kafkaEventPublisher.publishRefundEvent(conversionService.convert(orderDb,
Order.class), paymentResponse, userId);
        return getOrder(id, userId);
```

```
    }

    private Product getProduct(OrderRequest order) {
        final var productId = order.getProductId();
        final var product = productRepository.getProductInfo(productId);
        if (isNull(product))
            throw new ProductNotFoundException("Product with id %s does not exist".formatted(productId));
        else if (order.getAmount() > product.getCount())
            throw new ProductOutOfStockException("Product with id %s is out of stock.".formatted(productId));
        return product;
    }
}
```

Додаток Z (PaymentService.java)

```
package org.example.orderservice.service;

import lombok.RequiredArgsConstructor;
import org.example.orderservice.model.PaymentRequest;
import org.example.orderservice.model.PaymentResponse;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
@RequiredArgsConstructor
public class PaymentService {

    private final RestTemplate restTemplate;

    @Value("${payment.service.base-url}")
    private String paymentServiceBaseUrl;

    public PaymentResponse processPayment(PaymentRequest request) {
        String url = paymentServiceBaseUrl + "/payments";
        return restTemplate.postForObject(url, request, PaymentResponse.class);
    }

    public PaymentResponse refundPayment(String paymentId) {
        String url = paymentServiceBaseUrl + "/payments/" + paymentId + "/refund";
```

```
        return restTemplate.getForObject(url, PaymentResponse.class);
    }

    public PaymentResponse getPayment(String paymentId) {
        String url = paymentServiceBaseUrl + "/payments/" + paymentId;
        return restTemplate.getForObject(url, PaymentResponse.class);
    }
}
```