Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мережних технологій

Магістерська робота

Освітній ступінь: магістр

на тему**: «РОЗРОБКА МЕТОДОЛОГІЇ ІМПЛЕМЕНТАЦІЇ ТРАНЗАКЦІЙ В РОЗПОДІЛЕНИХ СИСТЕМАХ»**

Виконала:
студентка 2-го року навчання,
Спеціальності
121 Інженерія програмного забезпечення
Чернова Тетяна Андріївна

Керівник Глибовець А.М.
доктор технічних наук, доцент

Рецензент: Гороховський С. С.

Магістерська робота захищена
з оцінкою _________________
Секретар ЕК _______________
«___» ______________2023

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра мережних технологій

ЗАТВЕРДЖУЮ
Доктор технічних наук, доцент
____________А. М. Глибовець
(підпис)
„____”________________2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студентці Черновій Тетяні 2-го курсу магістерської програми факультету інформатики

ТЕМА Розробка методології імплементації транзакцій в розподілених системах

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Огляд предметної області
2 Проблематика та запропоноване рішення
3 Практична реалізація
Висновки
Список використаної літератури

Дата видачі „___” 2022 р.

Керівник (підпис)

Завдання отримано (підпис)

Календарний план виконання роботи:

Тема: “Розробка методології імплементації транзакцій в розподілених системах”

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	жовтень 2022 р.	
2.	Огляд документації та літератури за темою роботи	жовтень – листопад 2022 р	
3.	Аналіз існуючих рішень	грудень 2022 р.	
4.	Оволодіння навичками роботи з Debezium, Apache Kafka, Spring Boot	січень 2023 р.	
5.	Реалізація теоретичної частини роботи	січень - квітень 2023 р.	
6.	Реалізація практичної частини роботи	лютий - квітень 2023 р.	
7.	Надання роботи керівнику для перевірки, демонстрація практики	квітень 2023 р.	
8.	Корегування роботи за результатами перевірки керівником та результатами попереднього захисту	травень 2023 р.	
9.	Остаточне оформлення теоретичної частини та слайдів доповіді	травень 2023 р.	
10.	Захист магістерської роботи	червень 2023 р.	

Студентка: Чернова Тетяна Андріївна

Керівник: Глибовець Андрій Миколайович

“_______”________________

Зміст

Анотація 4

Вступ 5

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ 6

1.1 Поняття розподілених систем, мікросервісної архітектури 6

1.2 Проблеми в мікросервісній архітектурі 7

1.3 Консистентність даних в розподілених системах 8

1.4 Розподілені транзакції 9

1.5 Патерни, що полегшують роботу з розподіленими транзакціями 10

РОЗДІЛ 2. ПРОБЛЕМАТИКА ТА ЗАПРОПОНОВАНЕ РІШЕННЯ 15

2.1 Проблема обміну даними між мікросервісами 15

2.2 Асинхронних підхід обміну даними між мікросервісами 16

2.3 Transactional Outbox як асинхронний спосіб вирішення проблеми обміну даними в розподілених системах 18

2.4 Базовий підхід імплементації Transactional Outbox 20

2.5 Поняття WAL та CDC 22

2.5.1 WAL 22

2.5.2 CDC 23

2.6 Debezium як спосіб імплементації патерну Transactional Outbox 24

2.7 Spring Boot Starter та його роль в реалізації Transactional Outbox 27

2.8 Методологія розробки 29

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ 33

3.1 Опис стартеру 33

3.2 Підключення, вимоги та налаштування стартеру 34

3.4 Рекомендації щодо налаштування системи на базі розробленої методології 39

3.5 Оцінювання 41

Висновки 44

Список використаної літератури 45

Анотація

Дана робота присвячена аналізу проблематики використання транзакцій в розподілених системах, реалізації відомого мікросервісного патерну – Transactional Outbox у вигляді Spring starter, що додається до системи, конфігурується та полегшує роботу використання транзакцій і публікацію подій, що є частинами транзакції в мікросервісній архітектурі. Також вагому частину цієї роботи було присвячено створенню загальної методології роботи розподілених транзакцій на базі черг повідомлень, з використанням вищезазначеного стартеру, опису конфігурацій та налаштування черг повідомлень для коректної роботи транзакцій в розподілених системах. Результатом роботи є стартер, опис його конфігурації та загальна методологія роботи транзакцій в розподілених системах на базі черг повідомлень.

Ключові слова: розподілена система, розподілені транзакції, мікросервісна архітектура, Transactional Outbox патерн, асинхронне спілкування, Kafka, Debezium, Retryable Topic, Dead Letter Topic, неблокуюче читання.

Вступ

У сучасному світі мікросервісна архітектура все більше популяризується серед підходів до розробки додатків. Це рішення дозволяє розбивати систему на невеликі, незалежні компоненти, та отримувати швидку масштабованість та гнучкість. Однак, в таких розподілених середовищах виникають певні труднощі керування транзакціями, оскільки вони вимагають забезпечення атомарності, консистентності, ізольованості та стійкості над декількома сервісами.

У даному контексті виникає необхідність у знаходженні ефективних рішень для роботи з транзакціями в розподілених системах. Оскільки середовище має розподілений характер, традиційні підходи до роботи з транзакціями не можна застосувати. По цій причині виникли патерни та методології, які пропонують використання підходів, таких як "Transactional Outbox", для забезпечення атомарності операцій та публікації подій як частини транзакційного процесу.

Метою даної роботи є створення загального рішення для імплементації транзакцій в розподілених системах, використовуючи паттерн "Transactional Outbox" та черги повідомлень. Для досягнення цієї мети було розроблено стартер для полегшення роботи з розподіленими транзакціями, а також розроблено методологію, яка описує процес роботи з розподіленими транзакціями та публікацією подій у мікросервісній архітектурі. Далі в роботі буде надано огляд цієї методології, огляд стартеру та результати їх використання для розробки прототипу розподіленої системи.

Загальний висновок після огляду цієї методології та оцінювання прототипу дозволить зробити підсумки щодо ефективності та придатності даного підходу для роботи з розподіленими транзакціями в мікросервісній архітектурі.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Поняття розподілених систем, мікросервісної архітектури

Бурхливий розвиток засобів автоматизації, комп'ютеризації та електроніки привів до появи такого поняття як розподілена система. Розподілена система - це мережа незалежних компонентів, які взаємодіють між собою та координують свої дії для досягнення спільної мети [1]. Ці компоненти можуть бути апаратними або програмними, можуть знаходитись на різних комп'ютерах та можуть бути пов'язані спільною мережею. Серед прикладів розподілених систем можна вирізнити веб-додатки, платформи хмарного обчислення та мережі типу peer-to-peer.

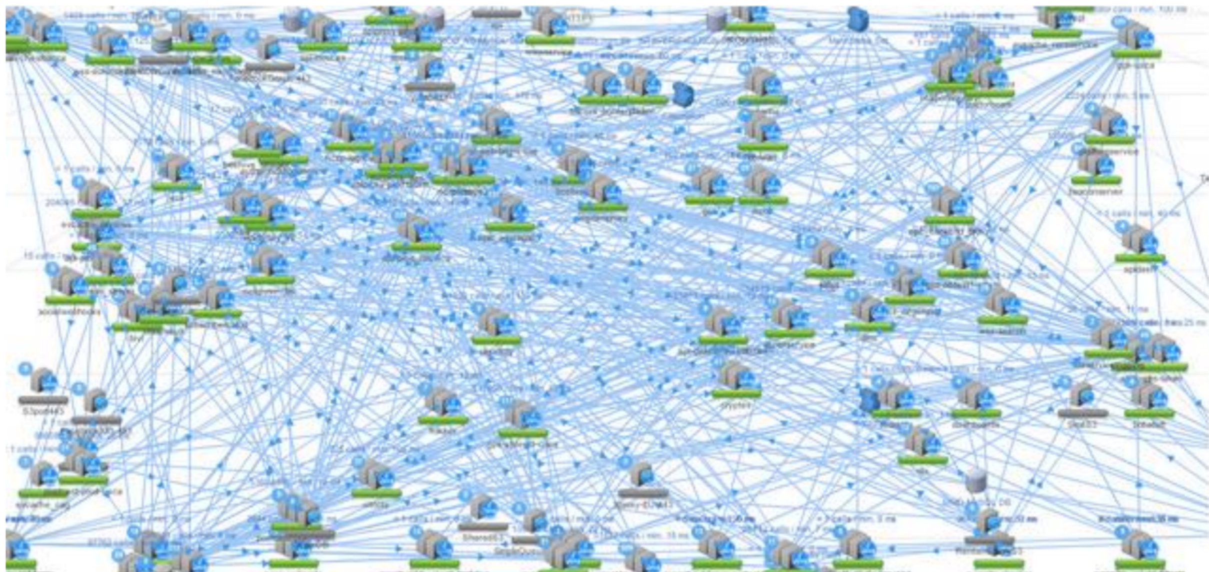

Рисунок 1 - Складна розподілена система [2]

У розподілених системах компоненти взаємодіють між собою передаючи повідомлення, віддалені виклики процедур або інші форми комунікації. Даний принцип взаємодії дозволяє кожному компоненту працювати незалежно, при цьому вони всі працюють спільно заради однієї мети та вносять свій вклад у функціональність загальної системи.

За рахунок “розподіленості”, управління такою системою може бути складним процесом, оскільки вимагає вирішення ряд проблем, таких як багатопоточність, консистентність даних, стійкість до помилок, збереження цілісності даних та масштабованості системи.

Мікросервісна архітектура - це підхід до побудови програмних систем, який використовує принципи розподілених систем [3]. У мікросервісній архітектурі застосунок – це набір невеликих, незалежних між собою сервісів, кожен з яких має певну бізнес-функціональність. Ці сервіси можуть бути розроблені, розгорнуті та масштабовані незалежно один від одного, що дозволяє досягти більшої гнучкості та спритності порівняно з традиційними монолітними архітектурними рішеннями.

Спілкування між сервісами в мікросервісній архітектурі відбувається за використанням чітко визначених API, використовуючи протоколи комунікації, такі як REST, черги повідомлень. Це дозволяє системі легше розвиватися та масштабуватися, оскільки зміни в одному сервісі не обов'язково потребують змін у всій системі. Однак, архітектура мікросервісів також вносить свій власний набір викликів, включаючи управління консистентністю даних, виявлення інших мікросервісів, балансування навантаження та моніторинг, трейсинг та логування [4]. Ці виклики пов'язані з тими ж проблемами, що і розподілені системи, тому їх потрібно ретельно вивчити та розробляти, щоб забезпечити стійкість та масштабованість системи.

Отже, розподілені системи та архітектура мікросервісів тісно пов'язані, оскільки мікросервіси використовують принципи розподілених систем, щоб забезпечити більшу гнучкість та масштабованість.

1.2 Проблеми в мікросервісній архітектурі

Управління розподіленими системами та мікросервісами потребує ретельного вивчення проблематики, що супроводжує дане рішення,

включаючи комунікацію між вузлами системи, збереження консистентності даних між різними мікросервісами, стійкість до помилок, масштабованість та багато інших.

Чим більше компонентів включає система, тим складніше стає контролювати та відстежувати процес комунікації між ними. Однією з найпоширеніших та найбільш базових проблем з якими можна зустрітись - налаштування правильної комунікації між сервісами. Це набагато складніше, ніж виклик методу з іншого модуля (як у монолітній архітектурі). Під час проектування комунікації між мікросервісами потрібно враховувати можливі затримки в мережі, можливі проблеми з доступністю серверів та затримки між запитами.

Відмовостійкість є ще одним важливим етапом при налаштуванні комунікації між мікросервісами. Якщо відбувається помилкова відповідь або мікросервіс недоступний у конкретний момент часу при створенні запиту до нього, необхідно визначити, як обробляти такі випадки: чи потрібно по-іншому обробляти запит, чи потрібно організувати процес повторнох запитів до недоступних мікросервісів [5]. У деяких випадках запит може охоплювати різні мікросервіси та реагувати на них по-різному. Це створює залежність між сервісами та ускладнює комунікацію між ними.

Також не менш важливим викликом для розподілених систем є консистентність даних в усіх його вузлах.

1.3 Консистентність даних в розподілених системах

Одним з викликів архітектури мікросервісів є забезпечення консистентності даних в розподіленій системі. Під консистентністю даних мається на увазі забезпечити точність та актуальність даних на всіх вузлах в розподіленій системі. У розподіленій системі дані можуть бути розподілені по кількох серверах та місцях, і кілька процесів або потоків можуть одночасно отримувати доступ до тих самих даних [6]. Забезпечення

консистентності в такому сценарії є складним завданням. До прикладу атомарне оновлення даних є не простою задачею при побудові мікросервісного рішення. Тобто якщо хоч одне оновлення не змогло відбутися, то й усі інші оновлення, що входили до набору, також мусять скасуватись та залишити систему в попередньому несуперечливому стані.

Управління консистентністю даних у розподіленій системі включає розв'язання двох основних проблем: роботу з багатопоточністю та відмовостійкістю. В багатопоточному середовищі, коли кілька процесів або потоків намагаються одночасно отримувати доступ та змінювати ті самі дані може виникати неконсистентність даних, оскільки різні вузли можуть мати різні варіації тих самих даних. Відмова ж може відбуватися, коли вузол або мережеве з'єднання зазнають проблем, наприклад сервіс в поточний момент часу не відповідає, або повертає помилку, що призводить до часткових або неконсистентних оновлень даних.

Існує кілька підходів до управління консистентністю даних у розподілених системах. Один з підходів - використання розподілених транзакцій, які забезпечують виконання групи операцій на кількох вузлах атомарно, тобто або всі операції успішні, або жодна не успішна [7]. Однак варто також враховувати що розподілені транзакції можуть бути складні для реалізації. У наступних розділах будуть наведені основні проблеми, які можуть виникати при імплементації розподілених транзакцій в мікросервісній архітектурі.

1.4 Розподілені транзакції

Розподілені транзакції - це транзакції, що охоплюють кілька баз даних або систем, які можуть знаходитися на різних серверах або навіть в різних географічних місцях [8]. Вони використовуються для забезпечення консистентності даних в кількох системах, гарантуючи, що або всі операції в транзакції завершаться, або жодна з них не буде виконана.

У розподіленій транзакції координатор транзакції відповідає за координацію дій усіх вузлів системи. Координатор спочатку запускає транзакцію, а потім надсилає запити до учасників систем для виконання потрібних операцій. Якщо всі вузли успішно виконують свої операції, координатор підтверджує транзакцію. Однак, якщо будь-яка з систем не виконує свою операцію, то координатор скасовує всю транзакцію, при цьому скасовуючи всі зміни, зроблені учасниками систем.

Розподілені транзакції можуть використовуватися для вирішення проблем консистентності даних в архітектурі мікросервісів, забезпечуючи, що всі сервіси, задіяні у транзакції, або підтверджують свої зміни, або відкатують їх координовано. Наприклад, якщо взяти до уваги сценарій, де користувач робить замовлення, і дані замовлення потрібно оновити у кількох сервісах, таких як сервіс управління запасами, сервіс обробки платежів та сервіс доставки. Без розподілених транзакцій складно забезпечити правильне оновлення даних у всіх сервісах і уникнути виникнення неузгодженості даних. Однак з розподіленими транзакціями, координатор може забезпечити, що всі сервіси здійснюють зміну або скасовують зміну в координований спосіб, забезпечуючи, що дані залишаються узгодженими у всіх сервісах [9].

Хоча розподілені транзакції можуть бути корисні для керування узгодженістю даних у мікросервісній архітектурі, вони також можуть створювати свій власний набір проблем та труднощів, таких як додаткове навантаження на продуктивність та операційну складність.

1.5 Патерни, що полегшують роботу з розподіленими транзакціями

Розглянемо деякі відомі мікросервісні патерни, що вирішують проблеми пов’язані з розподіленими транзакціями та їх популярні сучасні реалізації. Ці патерни можуть стати корисними для управління

транзакціями в мікросервісній архітектурі, оскільки вони надають альтернативні підходи до забезпечення консистентності даних між різними сервісами.

1. Двофазний коміт (2PC): Шаблон двофазного коміту є класичним розподіленим транзакційним шаблоном, який використовується для координації транзакцій між декількома системами [10]. У цьому патерні існує координатор, що ініціює транзакцію, і інші учасники (мікросервіси, вузли системи), що отримують повідомлення від координатора та зберігають транзакцію. Координатор визначає готовність кожного вузла зберегти транзакцію і, у випадку готовності усіх сервісів, координатор відправляє друге повідомлення, щоб вказати всім сервісам зберегти транзакцію. Якщо якийсь сервіс не готовий зберегти транзакцію, координатор відправляє повідомлення про відмову від транзакції. Даний патерн вирішує проблему консистентності даних в розподілених системах забезпечуючи те, що всі сервіси здійснюють або скасовують транзакцію у координований спосіб (за допомогою сервісу координатора). Він надає можливість забезпечити атомарність та консистентність між кількома системами, але може спричиняти проблеми, пов'язані з масштабованістю, доступністю та продуктивністю.

   Серед відомих реалізацій даного патерну можна виділити наступні:

   Atomikos: система керування транзакціями, заснована на Java, яка підтримує розподілені транзакції за допомогою патерну 2PC. Bitronix: менеджер транзакцій, заснований на Java, який підтримує розподілені транзакції за допомогою 2PC та інших патернів.

   NServiceBus: платформа для меседжингу та інтеграції на основі .NET, яка містить підтримку транзакцій 2PC.

2. Шаблон Saga, що є альтернативою першого шаблону 2PC.

   Принцип роботи шаблону Saga полягає у тому, що після виконання локальної транзакції база даних оновлюється, а повідомлення про подію публікується для ініціювання наступної локальної транзакції до баз даних [11]. У Saga використовуються такі терміни, як: Compensating transaction (транзакція, що скасовує зміни виконані попередньою транзакцією), Countermeasure (дизайн техніка, що використовується для компенсації нестачі рівня ізоляції), Compensatable transaction (транзакція, яку можна компенсувати), Pivot transaction (транзакція, що має роль маркера для саги: якщо транзакція успішна, то сага продовжуватиме ряд дій для завершення послідовності транзакцій) та retriable transaction (повторна транзакція). Цей шаблон дозволяє забезпечити правильність виконання розподіленої між мікросервісами транзакції, а кожну локальну транзакцію можна компенсувати за допомогою її компенсуючої транзакції. Saga також гарантує, що всі операції будуть завершені успішно або відповідні компенсаційні транзакції будуть запущені для скасування раніше виконаної роботи. Є два види координації транзакцій у Saga: хореографія та оркестрація.

   Серед відомих реалізацій даного патерну можна виділити наступні:

   Uber Cadence: Розподілена система оркестрування робочих процесів, яка містить підтримку як саг на основі хореографії, так і саг на основі оркестрації.

   AxonIQ: Відкрита платформа для створення мікросервісів, заснованих на подіях, яка включає підтримку саг.

   Lightbend Akka: Набір інструментів та середовище виконання для побудови високопродуктивних, розподілених та відмовостійких систем, яка містить підтримку саг.

3. Eventual consistency – це спосіб управління консистентністю даних в розподілених системах без використання традиційних транзакцій. У цій моделі дозволяється сервісам незалежно оновлювати свої локальні сховища даних, а послідовна консистентність досягається за допомогою асинхронних оновлень та вирішення конфліктів [12]. Хоча цей підхід може бути більш масштабованим та гнучким, ніж традиційні транзакції, він потребує обережного керування конфліктами даних та може призводити до тимчасових неузгодженостей.
4. Компенсаційний патерн - це спосіб скасувати наслідки невдалої транзакції без потреби повного відкату системи. У цьому патерні виконується компенсуюча транзакція, щоб скасувати наслідки початкової транзакції [13]. Цей патерн часто використовується в поєднанні зі патерном саги для керування локальними транзакціями в межах сервісу.
5. Transactional Outbox: даний патерн надає спосіб забезпечення консистентності даних у розподіленій системі, коли сервіс повинен публікувати події як частину транзакції. Це легкий та ефективний спосіб керування координацією подій між декількома системами, не уникаючи утруднощів традиційних механізмів розподіленої транзакції [14]. Він дозволяє сервісу публікувати події у спосіб, який консистентний з локальною базою даних, не потребуючи механізмів 2PC або інших складних механізмів, як от saga, чи компенсаційний патерн. Це легкий і ефективний спосіб керування координацією подій між декількома системами.

Серед відомих реалізацій даного патерну можна виділити наступні : Spring Cloud Stream: це фреймворк для створення мікросервісних додатків, що працюють на основі подій. Він дозволяє публікувати та

отримувати події за допомогою різних систем повідомлень, таких як Apache Kafka, RabbitMQ та Google Pub/Sub. Його модель програмування для реалізації патерну Transactional Outbox базується на API Spring Transaction.

Eventuate.io є платформою для створення та управління мікросервісами, що працюють за принципом event-driven applications. Вона включає підтримку для шаблону Transactional Outbox, надає набір клієнтських бібліотек для Java та Spring, які спрощують інтеграцію шаблону "Transactional Outbox" в додатки Spring Boot.

Говорячи про відомі мікросервісні патерни, варто зазначити, що їх можна імплементувати різноманітними способами, підлаштовуючись під клієнтські запити та технічну базу застосунку.

РОЗДІЛ 2. ПРОБЛЕМАТИКА ТА ЗАПРОПОНОВАНЕ РІШЕННЯ

У сучасних розподілених системах забезпечення консистентності даних, обміну даних та надійної комунікації між сервісами є важливою задачею. Також розподілені системи часто стикаються з проблемами підтримки цілісності даних між кількома мікросервісами, гарантією доставки важливих подій, забезпечення масштабованості та продуктивності при великому навантаженні, а також з проблемами коректної обробки відмов та збоїв мікросервісів та системи в цілому. Варто зазначити, що мікросервіси відповідають не лише за керування своїми локальними даними, а й за обмін оновленнями з іншими сервісами. Це, в свою чергу, також ставить задачу перед забезпеченням консистентності даних та надійним обміном інформацією між сервісами.

2.1 Проблема обміну даними між мікросервісами

При роботі з мікросервісами одним з найскладніших аспектів є керування обміном даних між ними. Мікросервіси рідко працюють відокремлено один від одного і часто потребують передачі даних та змін між собою.

Яскравим прикладом вищезазначеної проблеми була б наступна ситуація: уявімо систему, що керує замовленнями товарів. При створенні нового замовлення виникає потреба повідомити відповідні сервіси, такі як сервіс доставки і сервіс клієнта, про створення нового замовлення. Сервіс доставки використовує цю інформацію для організації доставки, а сервіс клієнта, до прикладу, може потребувати цю інформацію для оновлення балансу кредиту клієнта. Для повідомленнях сервісів про створення замовлення існують різні підходи. Одним з варіантів є синхронні виклики API, такі як REST або gRPC, до інших сервісів. Однак, такий підхід може привести до небажаного зв'язування та залежностей між сервісами, оскільки сервіси повинні знати, які сервіси викликати та де вони

знаходяться. Крім того потрібно брати до уваги можливі ситуації коли сервіси не доступні та коректно обробляти такого роду помилки.

Головною проблемою такого синхронного підходу є той факт, що один сервіс не може функціонувати без іншого сервісу, який він викликає та від результату виклику якого він є залежним. Не менш важливою проблемою є унеможливлення нових користувачів "підписатись" на минулі події, оскільки сервіси вже відправили запити один між одним та не можна отримати історію потоку подій з самого початку.

На противагу синхронному підходу для ефективного рішення двох вищезгаданих проблем можна використати асинхронний підхід обміну даними. Замість залежності від синхронної комунікації, сервіси, такі як сервіс замовлень, сервіс інвентаризації, сервіс клієнта та інші, можуть обмінюватись подіями через черги повідомлень, наприклад, Apache Kafka [15]. Шляхом підписки на ці події, кожен сервіс отримуватиме сповіщення про зміни даних від інших сервісів.

2.2 Асинхронних підхід обміну даними між мікросервісами

Використання асинхронного підходу, такого як повідомлення через брокер, для обміну даними між мікросервісами має кілька переваг порівняно з синхронним підходом. Ось основні переваги:

1. Loose Coupling: Асинхронна комунікація робить зв'язність мікросервісів слабшою, дозволяючи їм самостійно розвиватися. Сервіси можуть публікувати повідомлення в брокер повідомлень, не знаючи конкретних споживачів повідомлень, сприяючи меншій зв'язності компонентів та зменшенню залежностей між сервісами (так само споживачі можуть зачитувати повідомлення не знаючи хто їх публікує) [16].

2. Масштабованість: Асинхронна комунікація дозволяє досягти кращої масштабованості в мікросервісній архітектурі, розділяючи відправника і отримувача. Сервіси можуть публікувати повідомлення в брокер повідомлень у своєму власному темпі, а споживаючі сервіси можуть асинхронно обробляти повідомлення залежно від своєї доступності та потужності обробки. Це дозволяє горизонтальне масштабування сервісів та ефективне використання ресурсів.
3. Fault Tolerance: Асинхронна комунікація підвищує стійкість до відмов. Якщо сервіс тимчасово стає недоступним або зазнає перебоїв, повідомлення можуть бути буферизовані в брокері повідомлень, поки сервіс знову не стане доступним. Це запобігає втраті повідомлень і забезпечує надійну доставку повідомлень навіть при виникненні відмов [16].
4. Асинхронна обробка: За допомогою асинхронної комунікації мікросервіси можуть виконувати завдання паралельно та асинхронно, що зумовлює покращення загальної продуктивності продукту.
5. Event-Driven Architecture: Асинхронна комунікація добре поєднується з архітектурою, заснованою на подіях, де сервіси реагують на події та поширюють зміни. Події можна публікувати в брокері повідомлень, а зацікавлені сервіси можуть підписуватися на відповідні події, що дозволяє отримувати оновлення в реальному часі та ефективно поширювати події [17].
6. Flexibility and Extensibility Асинхронна комунікація надає гнучкість для додавання або вилучення сервісів без порушення роботи існуючої системи. Нові сервіси можуть легко підписатися на відповідні події або повідомлення від брокера повідомлень, що дозволяє розширювати систему та адаптуватися до змін у її архітектурі з плином часу.

За допомогою асинхронної комунікації з використанням брокера повідомлень, мікросервіси можуть досягти меншої зв’язності, стати менш

залежними від інших компонентів, досягти масштабованості, стійкості до відмов, асинхронної обробки, архітектури, заснованої на подіях, а також гнучкості, що призводить до більш надійних, масштабованих та розрізнених розподілених систем.

2.3 Transactional Outbox як асинхронний спосіб вирішення проблеми обміну даними в розподілених системах

При використання мікросервісної архітектури поширеним є явище використання своїх локальних сховищ даних - database per service [18]. До прикладу сервіс, що відповідає за створення замовлень зберігає замовлення в реляційну базу даних. Водночас сервіс може бажати надіслати подію про нове замовлення до Apache Kafka, щоб поширити цю інформацію на інші зацікавлені сервіси. Виконання цих двох дій може призвести до неузгодженості даних наступним чином: нове замовлення буде збережене в локальній базі даних, але відповідного повідомлення до Kafka не буде надіслано (наприклад, через проблему з мережею). Або, навпаки, ми можемо надіслати повідомлення до Kafka, але не зможемо зберегти замовлення в локальній базі даних. Обидві ситуації є небажаними та можуть призвести до того, що нібито успішно створене замовлення не буде відправлено. Або буде створений запит на відправку, але в самому сервісі замовлень не буде відомостей про відповідне замовлення.

Оптимальним рішенням цієї проблеми була б зміна лише одного ресурсу: або ж збереження даних на сервері або ж відправка повідомлення і в подальшому оновлювати інший ресурс на основі успішності першого. Якщо розглянути спершу зміну ресурсу Apache Kafka, то послідовність подій виглядатиме наступним чином: сервіс що відповідає за збереження замовлень не буде зберігати замовлення в базу, натомість відправить подію в брокер повідомлень та підпишеться на цей же топік, щоб отримати результат про успішну або неуспішну відправку. Таким чином, змінюється

лише один ресурс за раз, і якщо щось піде не так, ми одразу дізнаємося про це та повідомимо замовнику, що запит не вдалося виконати. Оскільки сервіс підписується на топік в Kafka, він буде отримувати сповіщення про доставку нового повідомлення в цей же топік і зможе зберегти нове замовлення на закупівлю у своїй базі даних.

Чим погане дане рішення? Якщо в системі виникатиме потреба добавити функціонал на зчитування усіх замовлень в базі, то наш сервіс не зможе читати свої власні записи (read your own write semantic). Припустимо, що сервіс замовлень також має API для пошуку всіх замовлень певного клієнта. При виклику цього API безпосередньо після розміщення нового замовлення через асинхронну обробку повідомлень з топіка Kafka може статися так, що замовлення на закупівлю ще не було збережено в базі даних сервісу і, отже, його не буде повернуто цим топіком. Це може призвести до дуже заплутаного користувацького досвіду, оскільки користувачі, наприклад, можуть пропустити новостворені замовлення в своїй історії покупок та загалом втратити вкрай важливу інформацію про замовлення.

Існують способи впоратися з цією ситуацією, наприклад, сервіс може зберігати нові замовлення на закупівлю в оперативній пам'яті та відповідати на наступні запити на основі цих даних. Однак, це швидко стає непростою задачею при реалізації складніших запитів або врахуванні того, що сервіс замовлень може складатися з кількох вузлів у кластерному середовищі, що потребуватиме передачі цих даних в межах кластера. Більш правильним рішенням буде зміна спершу стану бази, а потім відправка повідомлення базуючись на зміні даного ресурсу. Дане рішення якраз таки можна імплементувати використовуючи Transactional Outbox pattern.

Transactional Outbox pattern вирішує завдання збереження консистентності та надійності даних у розподілених транзакційних середовищах. У контексті розподілених транзакцій, шаблон "Transactional Outbox" зберігає дані та події пов’язані з ними в межах однієї бази даних,

та гарантує публікацію збережених подій. Публіковані події відображають зміни стану системи, які потрібно повідомити іншим вузлам розподіленої системи. Додаючи події в таблицю "outbox", даний патерн забезпечує їх надійне зберігання в межах тієї самої атомарної транзакції разом із основною операцією бізнес логіки. Цей підхід надає гарантії консистентності даних, оскільки або всі зміни комітяться разом, або жодна з них не виконується (при скасуванні однієї скасовуються усі). Також існує окремий фоновий процес, відомий як паблішер подій або диспетчер подій, що потім читає події з таблиці "outbox" та публікує їх у відповідний посередник повідомлень або систему обміну повідомленнями – це до прикладу може бути Apache Kafka, RabbitMQ та інші. Шляхом розподілення процесу публікації подій від основної транзакції, шаблон "Transactional Outbox" допомагає покращити продуктивність, масштабованість та стійкість системи.

Загалом, шаблон "Transactional Outbox" вирішує проблему досягнення надійної та консистентної комунікації, заснованої на подіях, у розподілених системах, “огортаючи” події в межі тієї самої транзакції та асинхронно відправляючи їх для забезпечення консистентності даних та покращення масштабованості та стійкості системи.

2.4 Базовий підхід імплементації Transactional Outbox

Один з базових підходів реалізації патерну Transactional Outbox є розробка та використання власних сервісів, що будуть зчитувати події з таблички Outbox, менеджити їх подальший життєвий цикл та керувати обробкою різноманітних помилкових сценаріїв. Основна перевага такого підходу полягає в його гнучкості, оскільки він дозволяє налаштувати логіку обробки повідомлень залежно від конкретних вимог.

У патерні "Transactional outbox" кожний мікросервіс підтримує таблицю "outbox" у своїй локальній базі даних. Ця таблиця виступає в якості

буфера, де зберігаються події та повідомлення до їх поширення у зовнішні системи (наприклад до публікування у меседж брокер). Структура таблиці outbox table зазвичай залежить від конкретної реалізації та вимог архітектури мікросервісів. Однак, є кілька загальних атрибутів, які часто присутні в таблиці вихідної скриньки, такі як message_id, message_payload, status, timestamp та інші. Конкретна структура та додаткові поля можуть варіюватися залежно від деталей реалізації та вимог архітектури мікросервісів [19].

В даній реалізації патерну існують спеціальні споживачі, які відповідають за періодичні запити до таблиці "outbox" для отримання непереданих та незчитаних раніше повідомлень. Такими споживачами можуть бути окремі мікросервіси або окремі модулі власного застосунку. Після отримання повідомлення з таблиці "outbox" спеціальний споживач обробляє його відповідно до визначеної бізнес-логіки. Цей процес може включати форматування повідомлення, виклик API або сервісів та обробку можливих помилок чи повторних спроб. При розробці такого споживача варто зважати на те, що необхідно надати гарантоване читання, відправку та підтвердження оброблених повідомлень. Для підтвердження читання до прикладу можна оновлювати статус у таблиці "outbox" для позначення успішної обробки.

Підхід з використанням спеціальних споживачів надає деталізований контроль і налаштування споживання та обробки повідомлень. Він дозволяє адаптувати логіку обробки, щоб задовольнити конкретні вимоги, такі як перетворення даних, перевірка безпеки або правил валідації. Крім того, він пропонує гнучкість у поєднанні з різними зовнішніми системами та адаптацію до їх конкретних API або протоколів.

Хоча даний варіант реалізації патерну надає гнучкість і можливість налаштування, він також має кілька потенційних недоліків, які варто врахувати. Серед вагомих недоліків можна вважати збільшений обсяг

розробки та необхідність в експлуатації. Необхідно проектувати, реалізовувати, тестувати та підтримувати компоненти власних споживачів, що збільшує загальний обсяг роботи з розробки та більше зусиль для їх підтримки. Масштабування власних споживачів може також приносити певні складнощі, особливо при роботі з великою кількістю повідомлень та подій або одночасним доступом до таблиці outbox. Потрібно використовувати відповідні механізми балансування навантаження та дбати про ефективну та масштабовану обробку повідомлень.

2.5 Поняття WAL та CDC

Change Data Capture (CDC) та Write-Ahead Log (WAL) є пов'язаними концепціями в контексті систем баз даних, але вони служать різним цілям.

2.5.1 WAL

"Write-Ahead Log" (WAL) є технікою, яка часто використовується в системах баз даних для забезпечення стійкості даних та підтримки консистентності транзакцій [20]. Вона передбачає запис змін, пов'язаних з транзакціями, у журнальний файл перед їх застосуванням до фактичних файлів даних бази даних. WAL виступає як послідовний журнал всіх змін, зроблених в базі даних, включаючи як успішно здійснені, так і незавершені транзакції.

Коли транзакція змінює дані в базі даних, відповідні зміни спочатку записуються в WAL. Це гарантує безпечне зберігання журналу на надійному носії, перш ніж модифікації застосовуються до самої бази даних. Завдяки цьому підходу система бази даних гарантує, що в разі збою або аварії системи зміни, записані в WAL, можуть бути відтворені та застосовані для відновлення бази даних в консистентний стан.

WAL надає кілька переваг. Вона покращує продуктивність бази даних, дозволяючи буферизувати модифікації в пам'яті та записувати їх на

диск у більш ефективному послідовному порядку. Вона також забезпечує цілісність даних, надаючи надійний запис транзакцій, що дозволяє відновлення після збоїв та скасування операцій. Крім того, WAL дозволяє реплікацію бази даних та реалізацію резервних серверів, реплікуючи журнал на інші системи для потреб синхронізації.

Загалом, використання WAL підвищує стійкість, надійність та можливість відновлення систем баз даних, роблячи його фундаментальною технікою для забезпечення консистентності та доступності даних.

2.5.2 CDC

Change Data Capture (CDC) - це техніка, що використовується у галузі інтеграції та синхронізації даних для захоплення та передачі змін даних у реальному часі з джерел баз даних до цільових систем [21]. CDC дозволяє постійно відстежувати та витягувати зміни даних, такі як вставки, оновлення та видалення, по мірі їх виникнення в джерелі бази даних. Ці зловлені зміни даних потім перетворюються у формат, який може бути використаний іншими додатками або реплікований до інших баз даних. CDC дозволяє ефективну та майже в реальному часі реплікацію даних, синхронізацію та інтеграцію між різноманітними системами, забезпечуючи, що дані залишаються консистентними та актуальними у всьому екосистемі. Це широко використовується в сценаріях, де своєчасна та точна передача даних є критичною, таких як побудова архітектури розподілених систем.

Зв'язок між CDC та WAL полягає в тому, що CDC часто використовує інформацію, записану в WAL, для захоплення та відстеження змін. Оскільки WAL містить послідовний журнал всіх модифікацій, зроблених в базі даних, його можна використовувати як надійне джерело для видобутку окремих змін та їх передачі іншим системам або споживачам.

Аналізуючи WAL, механізми CDC можуть ідентифікувати конкретні зміни, що сталися, включаючи редаговані рядки, стовпці та значення. Потім

вони можуть зафіксувати ці зміни та перетворити їх у формат, придатний для передачі, як наприклад, створення подій зміни або оновлення реплік бази даних.

Підсумовуючи, в той час як WAL забезпечує стійкість даних та консистентність в межах бази даних, CDC використовує інформацію, збережену в WAL, для захоплення та передачі окремих змін, зроблених в базі даних, іншим системам або споживачам. WAL виступає як базовий компонент, що дозволяє надійне відстеження та видобуток змін для потреб CDC.

2.6 Debezium як спосіб імплементації патерну Transactional Outbox

Debezium - це розподілена платформа з відкритим кодом, яка використовує техніку Capture Data Capture (CDC) для отримання та передачі змін даних в реальному часі з журналів транзакцій бази даних (використовуючи WAL) [22]. Вона інтегрується з різноманітними популярними базами даних, такими як MySQL, PostgreSQL, MongoDB та інші за рахунок конекторів до баз даних, а також надає незамінні функції, такі як здатність захоплювати зміни в усіх аспектах обробки даних, включаючи вставки, оновлення та видалення. При налаштуванні для роботи з певною базою даних Debezium підключається до журналу транзакцій бази даних, який, як правило, реалізується за допомогою WAL. WAL реєструє всі модифікації, зроблені в базі даних, включаючи вставки, оновлення та видалення, в послідовному та стійкому журнальному файлі. Debezium зчитує зміни, записані в WAL, та перетворює їх в потік подій змін. Ці події змін потім перетворюються в стандартизований формат, такий як JSON або Avro, і стають доступними для споживання додатками або системами вниз по ланцюжку.

Завдяки використанню підходу CDC на основі WAL, Debezium забезпечує надійний та мінімальний вплив захоплення даних, не навантажуючи додатково джерело бази даних [23]. Така ефективність доповнюється легко розширюваною архітектурою, яка може пристосовуватися до ситуацій стійкості до відмов, сприяючи простій синхронізації та інтеграції з численними платформами. Тому внесок Debezium у створення процесів стрімінгу в реальному часі, разом з наданням масштабованих реактивних потоків для сучасних розподілених фреймворків, неможливо недооцінити. Цей підхід мінімізує вплив на продуктивність джерела бази даних та забезпечує майже реальний час та точну реплікацію та інтеграцію даних між різними системами.

Говорячи про імплементацію Transactional Outbox патерна базуючись на Debezium, варто зазначити, що Change Data Capture (CDC) ідеально підходить для зчитування нових записів у таблиці outbox та передачі їх до Apache Kafka, оскільки сам Debezium побудований на основі Apache Kafka та використовує технологію Kafka Connect. На відміну від підходу заснованого на пулінгу даних, Debazium отримує події з дуже низьким навантаженням на систему та практично в режимі реального часу. Debezium поставляється з CDC-конекторами для кількох баз даних, таких як MySQL, Postgres та SQL Server і після отримання даних може передавати їх інших сервісам, до прикладу може публікувати подію в меседж брокер [19]. Надійність та запобігання втрати даних є важливими аспектами при використанні шаблону "Transactional Outbox" з Debezium та Apache Kafka.

Debezium надає пріоритет надійності, використовуючи журнали транзакцій, такі як Write-Ahead Log, які надаються системами баз даних, з якими він інтегрується. Ці журнали служать надійним та міцним джерелом для отримання інформації про зміни даних. Debezium підключається до журналу транзакцій і зчитує записані в ньому зміни, гарантуючи, що жодні зміни не пропускаються або не втрачаються. Під час захоплення змін з

журналу транзакцій Debezium запам’ятовує свою останню оброблену подію, щоб у разі перезапуску або відмови системи Debezium міг продовжити читати події з того місця, де він зупинився, і продовжити читання змін без дублювання тобто не читати повторно одні і ті ж дані. Цей механізм забезпечує цілісність даних та запобігає втраті даних у разі будь-яких збоїв або відмов.

Apache Kafka, брокер повідомлень, який використовується разом з Debezium, також відіграє важливу роль у забезпеченні надійності та міцності даних. Kafka - це меседж брокер, що розроблений як високодоступна та надійна розподілена система [24]. Він реплікує дані на кількох брокерах у кластері, забезпечуючи зайвість та міцність в разі відмов. Кожне повідомлення або подія, яке публікується в Kafka, зберігається у сховищі, яке називається "топіками". Топіки розбиваються на частини і кожна частина реплікується на кількох брокерах. Цей механізм реплікації забезпечує, що навіть якщо деякі брокери або вузли відмовлять, дані все ще доступні і можуть бути спожиті споживачами.

Debezium використовує ці та інші можливості надійності Kafka, публікуючи отримані з таблиці outbox зміни подій в топіки Kafka. Після публікації Kafka відповідає за забезпечення їх міцності та доступності. Події можуть бути оброблені споживачами або іншими додатками.

У підсумку, Debezium забезпечує надійність та запобігає втраті даних. Debezium надійно читає зміни з журналів транзакцій, запам’ятовує останню прочитану в межах журналу та відновлює роботу з останньої позиції читання в разі збоїв. Kafka, на основі якої розроблений Debezium надає міцне сховище та механізми реплікації, що гарантують доступність подій та їх споживання споживачами навіть у разі відмов. Ця комбінація надає надійне та ефективне рішення для реалізації шаблону "Transactional Outbox", знижуючи ризик втрати даних.

2.7 Spring Boot Starter та його роль в реалізації Transactional Outbox

Spring Boot Starter - є ключовою складовою частиною фреймворку Spring Boot та зручною і потужною функцією даного фреймворку [25]. Вона надає спрощений спосіб налаштування та запуску додатків на основі Spring шляхом об'єднання залежностей та налаштувань для конкретних функціональностей або компонентів. Зазвичай Spring Boot Starter включає підібраний набір бібліотек, класів автоконфігурації та налаштувань за замовчуванням, необхідних для включення певної функції або інтеграції з певною технологією.

За допомогою Spring Boot Starter розробники можуть швидко додавати необхідні можливості та функціонал до своїх додатків, не потребуючи ручного налаштування залежностей або написання стандартного коду. Ці стартери упаковують необхідні залежності та конфігурацію, що дозволяє розробникам зосередитися на написанні бізнес-логіки, а не витрачати час на складнощі налаштування та інтеграції різних компонентів.

Spring Boot Starter надає широкий спектр функціоналу, включаючи підключення до баз даних, веб-розробку, безпеку, обмін повідомленнями, кешування та багато іншого. Кожен стартер слідує певному підходу і надає послідовний спосіб інтеграції пов'язаних технологій в додатки Spring Boot. На прикладі стартеру "spring-boot-starter-web", можна побачити залежності та конфігурації, необхідні для розробки веб-додатків з використанням Spring MVC, включає в себе даний стартер. При доданні залежность spring-boot-starter-web у проект, вона автоматично імпортує кілька інших залежностей [26], таких як:

- Spring MVC: основний компонент фреймворку Spring для створення веб-додатків, що надає архітектуру MVC та обробляє відображення маршрутів запитів, обробку запитів та генерацію відповідей.

- Embedded Servlet Container: Spring Boot включає в себе вбудований контейнер сервлетів (наприклад, Tomcat, Jetty або Undertow), що дозволяє запускати веб-додатки без потреби в зовнішній конфігурації сервера.
- Spring Web: модуль, що надає додаткові функції та утиліти для роботи з вебом, включаючи обробку HTTP-запитів та відповідей, обробку форм, зв'язування даних, валідацію та обробку помилок.
- Jackson JSON: дана бібліотека включена для підтримки серіалізації та десеріалізації даних у форматі JSON (JavaScript Object Notation). Вона дозволяє легко конвертувати об'єкти Java в JSON та навпаки.

Крім наданих стартерів, розробники також можуть створювати власні стартери для упакування компонентів або бібліотек, специфічних для їх додатків чи домену. Це сприяє модульності коду, повторному використовувані та стандартизації в проектах організації.

Для імплементації Transactional Outbox було прийнято рішення використовувати можливості Spring Boot starter. В логіку стартера було закладено ідею створення таблички outbox в клієнтській базі, заповнення та читання даних до цієї таблички, інтергацію з технологією Debezium, публікування даних до меседж брокера. Саме Debezium з його конекторами до сховищ даних та Apache Kafka є одним з ключових компонентів, інтегрованих з цим стартером. Він відповідає за читання подій з таблиці Outbox та пересилання їх до брокера повідомлень, такого як Kafka. Debezium спрощує процес захоплення та потокової передачі подій, забезпечуючи надійну та миттєву реплікацію даних.

Стартер бере на себе роботу з необхідними залежностями, конфігураціями та кодом інфраструктури, що дозволяє розробнику сконцентруватися на основній логіці застосунку. Використовуючи стартовий пакет Spring Boot для транзакційного шаблону Outbox, можна досягти надійної та масштабованої архітектури для обробки комунікації на

основі подій. Стартер абстрагує від складнощів налаштування таблиці Outbox, читання даних з цієї таблиці фактично в реальному часі, керування транзакціями та інтеграції з Debezium, що дозволяє безпроблемно та ефективно реалізувати шаблон.

Загалом, цей стартер надає цілісне рішення для реалізації патерну Transactional Outbox у додатках Spring Boot. Він спрощує процес розробки, сприяє модульності та повторному використанню коду і допомагає інтегрувати комунікацію на основі подій за допомогою Debezium та брокеру повідомлень, такого як Apache Kafka.

2.8 Методологія розробки

Реалізація транзакцій в розподілених системах вимагає ретельного врахування можливих викликів та проблемних ситуацій, які ставлять перед нами природа розподіленої системи. Тому, проаналізувавши та врахувавши усі можливі проблеми транзакцій в розподілених системах, була розроблена методологія імплементації транзакцій в розподілених системах на базі патерну Transactional Outbox та черг повідомлень. Transactional Outbox патерн може бути використаним для вирішення цих проблем і забезпечення послідовності, консистентності та надійності даних у множинних сервісах.

Transactional Outbox патерн передбачає використання таблиці outbox table, яка виступає в ролі буфера для запису подій або повідомлень, які потрібно надіслати іншим сервісам в рамках транзакції [14]. Основні кроки реалізації цього патерну такі:

- Початок транзакції: розпочати транзакцію в базі даних для забезпечення атомарності та послідовності операцій.
- Виконання операцій з базою даних: виконати необхідні операції в межах транзакції, такі як вставка, оновлення або видалення даних.
- Запис подій у таблицю outbox: як частину транзакції, записати відповідні події або повідомлення в таблицю outbox. Ці події

представляють дії, які потрібно передати чи виконати в інших сервісах. Вони можуть бути опубліковані до певного топіку в меседж брокері, як то Apache Kafka, та пізніше оброблені різноманітними споживачами за задоволення власних цілей

- Асинхронна обробка подій: необхідно використовувати окремий процес або фонове завдання для читання подій з таблиці outbox та надсилання їх відповідним сервісам або брокерам повідомлень. Цей процес можна реалізувати за допомогою інструментів, таких як Debezium, який може захоплювати події з таблиці outbox та пересилати їх до черг повідомлень. Також необхідно подбати про уникнення повторного читання даних, уникнення втрати даних після читання, відновлення читання з того ж місця, де було останнє читання, читання даних практично в реальному стані
- Завершення або скасування транзакції: якщо всі операції в межах транзакції пройшли успішно, необхідно зафіксувати транзакцію в базі даних. Це забезпечує атомарність змін до даних та записані події у табличці outbox. Якщо відбувалась помилка під час записування даних в клієнтську базу, чи в таблицю outbox, або ж помилка під час публікації та подальшої обробки подій то необхідно зафіксувати проблему на стороні клієнта (наприклад змінити статус замовлення на скасований)

Нижче наведена запропонована методологія імплементації транзакцій в розподілених системах на базі предметної області – системи замовлень товарів

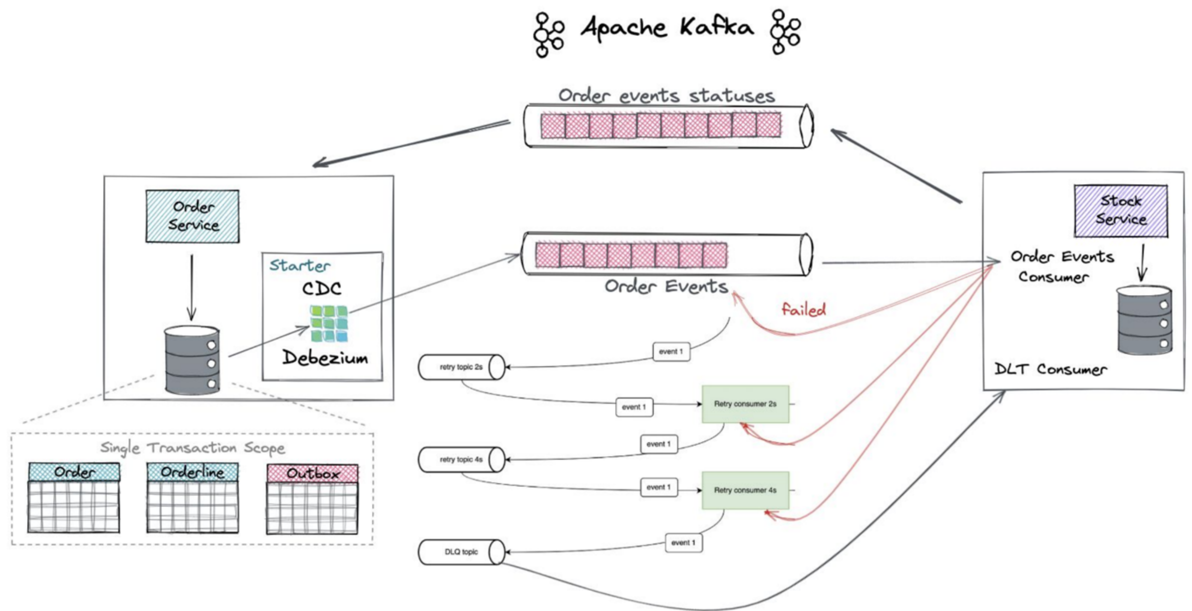

Рисунок 2 – Розроблена методологія імплементації транзакцій в розподілених системах

Дана система реалізована у вигляді двох мікросервісів: Order Service, що отримує здійснювати замовлення товарів, та Stock Service, що на базі отриманих замовлень проводить певні перевірки товару та інші маніпуляції для успішного завершення життєвого циклу замовлення. Обидва мікросервіси реалізовані на Java використовуючи Spring Boot Framework, JPA/Hibernate для доступу до їхніх відповідних баз даних, та в якості баз даних в обох сервісах була обрана MySQL. Проте для даної методології немає чіткої прив'язки до бази даних чи то фреймфорку, оскільки дана методологія дозволяє реалізацію з використанням різноманітних технологій, до прикладу таких як: CDI, PostgreSQL, WildFly та інших. Сервіс замовлень надає простий REST API для створення замовлень. Сервіс складу за допомогою Apache Kafka отримує події, експортовані сервісом замовлень, та створює відповідні записи у своїй власній базі даних MySQL.

Для експортування подій у Apache Kafka, сервіс замовлень використовує розроблений Spring Boot Starter, що дбає про налаштування таблички outbox, запис подій в дану табличку в межах тієї ж транзакції, що і збереження замовлення в базу даних. Стартер самостійно дбає про

моніторинг та зчитування даних в табличці outbox використовуючи WAL, CDC, Debezium, та публікацію зчитаних даних до Apache Kafka.

Після зчитування подій з Apache Kafka, Stock Service сповіщає про вдале або провальне зчитування та обробку даних використовуючи Apache Kafka. Ці події пізніше сервіс замовлень читає та змінює статус замовлення з pending до closed / canceled. Також даний сервіс дбає про відмовостійкість за рахунок повторного читання даних за необхідності, гарантує обробку даних та відповідь у вигляді публікації статусу обробки у відповідний топік в Apache Kafka.

Дотримуючись цієї методології, можна забезпечити надійний обмін даними між мікросервісами, обробку операцій в межах транзакції та відповідних подій, відправлених іншим сервісам, атомарно та послідовно. Навіть якщо виникають проблеми під час надсилання подій, їх можна спробувати повторно відправити або відтворити з таблиці outbox, забезпечуючи кінцеву послідовність в розподіленій системі. Також пріоритет стоїть за збереженням цілісності та послідовність даних, одночасно дозволяючи асинхронну комунікацію між сервісами, що робить його ефективною методологією для реалізації транзакцій в розподілених системах.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ

У рамках даної роботи був розроблений Spring boot Starter, який полегшує конфігурацію роботи з транзакціями в розподілених системах, що публікують події як частину транзакції, розроблено загальну методологію роботи з транзакціями в розподілених системах на базі черг повідомлень яка використовує стартер. Також проведено тестування прототипу та оцінено його роботу.

Нижче в даному розділі будуть наведені деталі роботи стартеру, покроково описано процес інтеграції клієнтських сервісів з даним стартером, описано процес підключення, задокументовано базові налаштування стартеру, а також описано доступний користувачеві функціонал наданий стартером.

Також даний розділ включає в себе деталі розробленої методології імплементації транзакцій в розподілених системах, опис загальних конфігурацій необхідних для коректної та надійної роботи розподіленої системи, поради щодо налаштувань мікросервісів, інтеграцією їх з Apache Kafka.

Не менш важливою частиною практичної реалізації є оцінювання розробленого прототипу та перевірка системи на коректну роботу з помилковими сценаріями, відпрацювання системи в залежності від різних сценаріїв та перевірка системи на відмовостійкість, консистентність даних.

3.1 Опис стартеру

Розроблений Spring Boot Starter має призначення спеціальної бібліотеки для полегшення впровадження патерну Transactional Outbox в застосунок, побудований на основі Java. Його основною метою є спрощення процесу збереження та публікації подій у розподілених системах.

Основною складовою цього стартера є інтеграція з Debezium - потужним інструментом для зчитування та моніторингу змін у базах даних

у реальному часі. Оскільки Debezium дозволяє отримувати зміни даних, які відбуваються у клієнтській базі даних, ці зчитані події можна пізніше перетворювати їх у потокові події та використовувати для розповсюдження даних до інших систем, наприклад, до Apache Kafka. Застосування даного стартеру у своєму проекті дозволяє розробникам швидко та ефективно впроваджувати Transactional outbox патерн. Стартер включає необхідні залежності та конфігурації, а також надає зручні методи для збереження подій у відведеній таблиці (так званому outbox) і передачі їх у Debezium для подальшого розповсюдження.

Використовуючи цей стартер, розробники можуть ефективно впроваджувати Transactional Outbox патерн, покладаючись на потужність Debezium для зчитування та розповсюдження подій у розподілених системах, а також на гарантовану доставку та цілісність даних за рахунок використання Apache Kafka як базової технологій для Debezium.

3.2 Підключення, вимоги та налаштування стартеру

Для того, щоб інтегрувати Java застосунок з розробленим стартером необхідно перевірити застосунок на наявність усіх необхідних вимог для роботи з стартером:

- Підтримка бази даних: Debezium потребує підтримки певної бази даних, яка може створювати журнал транзакцій або події змін даних (CDC). На даний момент Debezium підтримує бази даних, такі як MySQL, PostgreSQL, SQL Server, Oracle, MongoDB, Cassandra, DB2 і SQLite.
- CDC: База даних, яку використовується застосунок, що інтегрується зі стартером, повинна мати увімкнений журнал транзакцій або механізм CDC. Це дозволяє Debezium в реальному часі отримувати та відстежувати зміни, що відбуваються в базі даних.

- Права користувача бази даних: Користувач бази даних, який налаштований в конфігурації Debezium, повинен мати відповідні дозволи для доступу до журналу транзакцій або подій CDC. Ці дозволи необхідні для того, щоб Debezium міг читати зміни бази даних.
- Kafka: Debezium ґрунтується на Apache Kafka для публікації отриманих змін бази даних як подій. Тому потрібно налаштувати та мати доступний працюючий кластер Kafka або сумісну систему повідомлень.
- Налаштування: Debezium потребує певних конфігурації для підключення до бази даних, визначення тем, вказання формату серіалізації та налаштування інших параметрів. Ці властивості повинні бути належним чином встановлені в файлі конфігурації Debezium або передані як параметри при запуску Debezium. (нижче в даному розділі будуть описані необхідні конфігурації)
- Належна мережева доступність: Система, на якій розташований Debezium, і сервер бази даних повинні мати належну мережеву доступність. Переконайтеся, що необхідні порти відкриті і доступні між екземпляром Debezium, базою даних та системою повідомлень.

Інтеграція застосунку зі стартером розпочинається з його підключення у вигляді залежності:

```xml
<dependency>
    <groupId>com.chernova</groupId>
    <artifactId>transactional-outbox-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

Рисунок 3 – Підключення стартеру до клієнтського застосунку

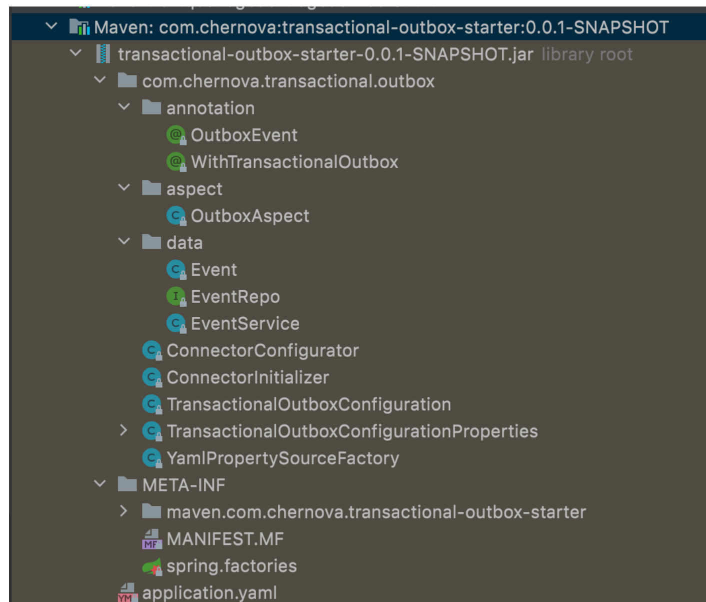

Рисунок 4 – Підключений стартер

Так виглядатиме доданий стартер до застосунку. Для коректної роботи стартеру та підтримки роботи з outbox table та Apache Kafka, необхідно передати наступні конфігурації, що будуть зчитані стартером при ініціалізації проекту:

- connector.class: вказує клас конектора, який використовується для підключення до джерела даних. До прикладу значення конектор класу io.debezium.connector.mysql.MySqlConnector, означає підключення до MySQL бази даних
- database.hostname: вказує ім'я хоста, на якому розташована база даних клієнта
- database.port: вказує порт, на якому доступна база даних клієнта
- database.user: вказує ім'я користувача для підключення до бази даних клієнта
- database.password: вказує пароль для підключення до бази даних клієнта
- database.server.id: унікальний ідентифікатор сервера, який використовується для розпізнавання транзакцій

- database.server.name: унікальне ім'я сервера, яке використовується для ідентифікації каналу змін даних у Debezium
- database.history.kafka.bootstrap.servers: вказує адресу та порт сервера Apache Kafka, який використовується для зберігання історії змін бази даних (до прикладу “broker:9092”)
- database.history.kafka.topic: вказує тему Kafka, в яку зберігається історія змін бази даних
- key.converter.schema.registry.url: вказує адресу та порт сервера реєстрації схем Avro.
- value.converter.schema.registry.url: вказує адресу та порт сервера реєстрації схем Avro.

Дані конфігурації повинні бути надані у yaml файлі. Ці конфігурації використовуються для налаштування Debezium з метою підключення до бази даних MySQL, зчитування змін у реальному часі та перетворення їх на події, які можна розповсюджувати через Apache Kafka. Для того щоб керувати підключенням чи відключенням роботи стартеру при запуску застосунку, необхідно також задати значення true/false для конфігураці transactionaloutbox.enabled.

```yaml
transactionaloutbox:
  enabled: true
  connector:
    class: io.debezium.connector.mysql.MySqlConnector
  database:
    hostname: mysql
    port: 3306
    user: debezium
    password: dbz
    server:
      id: 42
      name: asgard
    history:
      kafka:
        topic: dbhistory.demo
        bootstrap:
          servers: broker:9092
    key:
      converter:
        schema:
          registry:
            url: http://schema-registry:8081
    value:
      converter:
        schema:
          registry:
            url: http://schema-registry:8081
```

Рисунок 4 – Приклад конфігурації стартеру в yaml файлі

Для того, щоб стартер розпочав процес збереження подій до outbox table, зчитування подій з outbox table та публікацію їх до Apache Kafka, необхідно додати до методу сервісу анотацію @WithTransactionalOutbox:

```java
@Transactional
@WithTransactionalOutbox
public PurchaseOrder addOrder(@OutboxEvent PurchaseOrder order) {
    return purchaseOrderRepo.save(order);
}
```

Рисунок 5 – Використання анотації WithTransactionalOutbox доданого стартера

В даному прикладі стартер в межах однієї транзакції зі збереженням замовлення до бази збереже подію до таблиці outbox. Для реалізації такого

функціоналу стартер використовує можливості Spring, а саме Spring AOP:

```
@Around(value = "@annotation(withTransactionalOutboxAnnotation)")
public Object withTransactionalOutbox(ProceedingJoinPoint joinPoint, WithTransactionalOutbox withTransactionalOutboxAnnotation)
        throws Throwable {
    Object result = joinPoint.proceed();
    String jsonData = "{}";

    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    Annotation[][] annotations = signature.getMethod().getParameterAnnotations();
    Object[] args = joinPoint.getArgs();

    Object annotatedArgument;
    for (int i = 0; i < args.length; i++) {
        for (Annotation annotation : annotations[i]) {
            if (annotation instanceof OutboxEvent) {
                annotatedArgument = args[i];
                jsonData = objectMapper.writeValueAsString(annotatedArgument);
                break;
            }
        }
    }
    ...

    return result;
}
```

Рисунок 6 – Використання можливостей Spring AOP для реалізації логіки стартера

Отже, застосувавши усі необхідні налаштування до стартеру, можна інтегрувати застосунок з Transactional Outbox патерном та таким чином забезпечити атомарність операцій та консистентність даних між різними сервісами. Більше того, додаючи стартер до свого застосунку можна отримати ряд можливостей які надаються технологіями Debezium та Apache Kafka.

3.4 Рекомендації щодо налаштування системи на базі розробленої методології

При проектуванні та реалізації розподілених систем, які використовують методологію імплементації транзакцій в розподілених системах, варто врахувати кілька рекомендацій.

Перш за все, важливо переконатися, що компоненти системи мають слабку зв'язність та спілкуються за допомогою асинхронного обміну повідомленнями для розділення транзакційної обробки від зовнішніх сервісів [27]. Даний підхід спілкування сприяє стійкості та

масштабованості. Не менш важливим є використання надійного брокеру повідомлень для комунікації між мікросервісами та гарантованої надійної доставки повідомлень.

Однією з важливих компонентів при розробці дистрибутивної системи є впровадження механізмів обробки помилок та повторного читання даних в разі помилок, управління відмовами та забезпечення консистентності даних. Для досягнення даних цілей можна використовувати механізми Retryable Topic та Dead Letter Topic. Механізм "Retryable Topic" - це функціонал, що надається деякими фреймворками або бібліотеками обробки повідомлень, що дозволяє визначати конкретні топіки повідомлень, які піддаються повторній спробі читання даних споживачами у разі невдалих попередніх спроб [28]. Він працює наступним чином: у разі виникнення помилки під час обробки повідомлення фреймворк автоматично повторює операцію на основі налаштованих конфігурацій. Використання даного механізм усуває необхідність у самостійних реалізаціях сценаріїв повторного читання та, таким чином, значно полегшує роботу розробника не нехтуючи при цьому консистентності даних та знижуючи ризик втрати або некоректності даних. Dead Letter Topic (або Dead Letter Queue) виступає в якості топіку для повідомлень, які не вдалось успішно зчитати та обробити ретрай механізмами [29]. Невдалі повідомлення перенаправляються до DLT топіку, що не ніяк не впливає на роботу системи, та дозволяє в подальшому паралельно проводити читання повідомлень з черг, оброблювати дані і не зупиняти роботу системи. Моніторинг DLT дозволяє отримувати інформацію про помилки, які найчастіше виникають у системі, виявляти патерни та знаходити рішення проблем. Також варто зазначити, що Retryable Topic та Dead Letter Topic механізми забезпечують систему неблокуючим читанням даних з брокера.

При розробці розподілених систем на базі вищезгаданої методології важливу увагу потрібно надати ідемподентності споживачів. Оскільки споживачі виконують обробку повідомлень з черги та виконують певні операції або змінюють стан системи на основі цих повідомлень, важливо забезпечити, щоб ці операції гарантували безпеку та надійність системи, призводили обробку даних до однакового результату незалежно від кількості повторних повідомлень. Це важливо для забезпечення консистентності даних та уникнення небажаних побічних ефектів. Наприклад, якщо споживач створює запис у базі даних на основі отриманого повідомлення, ідемподентність забезпечує, що у базі даних буде створено тільки один унікальний запис навіть якщо повідомлення буде оброблено кілька разів. Також важливим при проектуванні споживачів є використання унікальних ідентифікаторів для оброблених повідомлень або збереження стану процесу обробки, що забезпечить ідемподентну обробку.

3.5 Оцінювання

Оцінка розробленого прототипу – невід'ємна частина розробки. Це важливий крок у визначенні його ефективності та придатності для використання в розподілених системах. Тому для оцінювання даного прототипу включено аналіз різних аспектів реалізації та їх впливу надійність та консистентність даних системи, здатність системи коректно обробляти помилкові сценарії, реагувати на можливі збої системи, відсутність доступу до окремих компонентів розподіленої системи.

Один із аспектів, який було оцінено, це інтеграція Debezium з існуючими компонентами та технологіями в системі. Це включило перевірку забезпечення правильної конфігурації та налаштування стартеру який базується на імплементації Transactional outbox патерну. Також було оцінено інтеграцію з брокером повідомлень Kafka, та перевірено публікацію повідомлень до брокеру, наявність повторного читання даних

при необхідності, відсутність блокування читання у випадку повторних обробок повідомлень. Першим кроком було перевірено успішний сценарій збереження замовлення товару до клієнтського сховища даних, збереження відповідної події з корисною інформацією замовлення, публікацію даної події та зміну статусу замовлення на closed. Пізніше було проведення оцінювання реагування системи на різноманітного роду помилкових сценаріїв.

Оскільки помилки можуть виникати на різних етапах транзакційного процесу, таких як публікація подій в аутбоксі, обробка подій споживачами або взаємодія з зовнішніми сервісами, вкрай важливо мати надійні механізми обробки помилок. Помилки повинні бути фіксовані, реєструвані в журналі і вживатися відповідні заходи для їх обробки. Це може включати повторну спробу операції читання даних, застосування компенсаційних заходів для збереження цілісності даних. На даному прототипі були проведені перевірки поведінки системи при наступних сценаріях:

- виникнення помилки при спробі збереження замовлення до клієнтської бази та перевірка системи на скасування публікації повідомлення
- виникнення помилки при спробі публікації повідомлення до брокера повідомлень та перевірка системи на скасування збереження даних про замовлення до клієнтської бази
- помилка обробки повідомлення споживачем в Stock Service та перевірка системи на повторне читання з використанням retryable механізму
- після кількох невданих спроб обробки повідомлень, споживач в Stock Service не може обробити повідомлення та перевірка системи на відправку повідомлення до DLT топіку
- після успішної обробки повідомлення Stock Service відправив статус успішної обробки до відповідного топіку і сервіс замовлень

базуючись на даній відповіді змінив статус замовлення на завершений

- після неуспішної обробки повідомлення Stock Service відправив статус проблеми обробки до відповідного топіку і сервіс замовлень, базуючись на даній відповіді, змінив статус замовлення на скасований

Order Service	OutboxStarter	Stock Service	Order & Stock Services
order saved in DB event saved in outbox table	notification was sent	notification was consumed	success event was published -> order status changed from created to closed
order saved in DB event saved in outbox table	notification was sent	notification was eventually consumed by retry mechanism	success event was published -> order status changed from created to closed
order saved in DB event saved in outbox table	notification was sent	notification was eventually consumed by DLQ mechanism	success event was published -> order status changed from created to closed
order saved in DB event saved in outbox table	notification was sent	notification was consumed with error	error event was published -> order status changed from created to cancelled
order saved in DB event saved in outbox table	notification was sent	notification was not eventually consumed by retry and DLQ mechanisms	error event was published -> order status changed from created to cancelled
order saved in DB event saved in outbox table	notification was not sent because the problem occurred	——	transaction was declined
order was not saved DB / event was not saved in outbox table because the problem occurred	——	——	transaction was declined

Рисунок 7 – Результати оцінювання прототипу

Як результат оцінювання система коректно відпрацювала в обох успішних та помилкових сценаріях. Варто зазначити, що даний прототип має правильно налаштовані механізми повторної спроби читання та обробки даних, оскільки у випадку виникнення помилки, спроектований механізм повторної спроби автоматично повторював операцію, щоб подолати тимчасові проблеми, такі як проблеми з мережею або тимчасова недоступність ресурсів. Повторні спроби обробки виконувались з використанням стратегій публікацій подій до DLT топіку, щоб уникнути перевантаження системи та забезпечити затримку між послідовними спробами при цьому не блокуючи читання інших даних.

Висновки

Проведений аналіз проблематики роботи з транзакціями в розподілених системах підтвердив складність даного аспекту розробки. Багатофазові операції, зміни в стані бази даних та ненадійна комунікація між сервісами можуть призводити до непередбачуваних проблем, таких як втрата даних або відсутність консистентності даних.

В процесі огляду сучасних підходів та рішень для роботи з транзакціями в розподілених системах, було виявлено, що одним з ефективних рішень є використання патерну Transactional outbox. Даний патерн дозволяє зберігати події, пов'язані з транзакціями, в окремій таблиці, що уможливлює їх подальшу інтеграцію з іншими сервісами через додатки або механізми, наприклад, через брокер повідомлень Apache Kafka.

Здобуті знання дозволили розробити методологію імплементації транзакцій в розподілених системах, а також створити стартер для полегшення такої роботи з розподіленими транзакціями та публікацією подій, що є частинами транзакції в чергу повідомлень. Цей стартер надає зручний і стандартизований спосіб інтеграції патерну Transactional Outbox в Spring Boot додатки, забезпечуючи автоматичне керування транзакціями та публікацію подій.

Оцінювання прототипу показало ефективність впровадження даного підходу в розподілені системи. Розроблена методологія та стартер можуть допомогти забезпечити надійну обробку розподілених транзакцій та публікацію подій.

В результаті, дана робота підтвердила значну користь і потенціал використання розробленого стартеру та методології для роботи з розподіленими транзакціями. Дані рішення можуть допомогти забезпечити консистентність та надійність в розподілених системах, спрощуючи розробку та підтримку таких систем.

Список використаної літератури

1. Maarten van Steen, Andrew S. Tanenbaum. "A brief introduction to distributed systems." Web. 2016. https://www.distributed-systems.net/my-data/papers/2016.computing.pdf
2. NTT, "Distributed System," NTT, Stuttgart, 2017.
3. Fowler, Martin. "Microservices: A Definition of This New Architectural Term." Web. 2014 https://martinfowler.com/articles/microservices.html.
4. Trzaska, Mariusz. “Technical challenges of creating an IT system in microservices architectural style using cloud services” Web. 2022 https://users.pja.edu.pl/~mtrzaska/Files/PraceMagisterskie/220217-Grabowski.pdf
5. Yadav, Arpit. “Reaching High Availability in Connected Car Backend Applications” Web. 2017 https://monarch.qucosa.de/api/qucosa%3A20780/attachment/ATT-0/
6. Kizilpinar, Dilfuruz. "Data Consistency in Microservices Architecture" Web. 2021 https://medium.com/p/5c67e0f65256
7. S. Newman, Building Microservices, O’Reilly Media, 2015.
8. Ozkaya, Mehmet. “Microservices Distributed Transactions” Web. https://medium.com/design-microservices-architecture-with-patterns/microservices-distributed-transactions-a71a996e5db8
9. Srivastana, Jolly. “Distributed Transaction” Web. https://medium.com/system-design-concepts/distributed-transaction-ea2eb25d7808
10. Fowler, Martin “Two Phase Commit” Web. https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html
11. Richardson, Chris. “Pattern: Saga” Web. https://microservices.io/patterns/data/saga.html

12. A. Homer, J. Sharp, L. Brader, T. Swanson and M. Narumoto, "Cloud Design Patters," 2014.
13. Microsoft, "Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications [Online]. Available: https://msdn.microsoft.com/enus/library/dn568099.aspx
14. Richardson, Chris. "Pattern: Transactional Outbox" Web. https://microservices.io/patterns/data/transactional-outbox.html
15. Mielle. "Microservices patterns: synchronous vs asynchronous communication" Web. https://greeeg.com/en/issues/microservices-patterns-synchronous-vs-asynchronous
16. Medium. "Difference between Synchronous vs Asynchronous Communication in Microservices" Web. https://medium.com/javarevisited/how-microservices-communicates-with-each-other-synchronous-vs-asynchronous-communication-pattern-31ca01027c53
17. Microsoft. "Asynchronous message-based communication" Web. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication
18. Richardson, Chris. "Pattern: Database per service" Web. https://microservices.io/patterns/data/database-per-service.html
19. Morling, Gunnar. "Reliable Microservices Data Exchange With the Outbox Pattern" Web. https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/
20. Chola, Abhinal "Understanding Write Ahead Logging: 4 Comprehensive Aspects" Web. https://hevodata.com/learn/write-ahead-logging/
21. Spring Blog. "Case Study: Change Data Capture (CDC) Analysis with CDC Debezium source and Analytics sink in Real-Time" Web.

https://spring.io/blog/2020/12/14/case-study-change-data-capture-cdc-analysis-with-cdc-debezium-source-and-analytics-sink-in-real-time

22. Debezium Documentation. Web. https://debezium.io/documentation/reference/stable/index.html
23. Hevodata. "3 Easy Steps to Decode CDC Using Debezium Spring Boot" Web. https://hevodata.com/learn/debezium-spring-boot/
24. Apache Kafka Documentation. Web. https://kafka.apache.org/
25. Baeldung. Web. https://www.baeldung.com/spring-boot-starters
26. Educba Blog. "Spring Boot Starter Web" Web. https://www.educba.com/spring-boot-starter-web/
27. Microsoft, "Designing Web applications" Web. https://msdn.microsoft.com/enus/library/ee658087.aspx
28. Baeldung Blog "Implementing Retry in Kafka Consumer" Web. https://www.baeldung.com/spring-retry-kafka-consumer
29. Medium. "Dead Letter Queue (DLQ) in Kafka" Web. https://towardsdatascience.com/dead-letter-queue-dlq-in-kafka-29418e0ec6cf
30. Richardson, Chris. "Pattern: Idempotent Consumer" Web. https://microservices.io/patterns/communication-style/idempotent-consumer.html