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

РОЗРОБКА МЕТОДОЛОГІЇ РЕАЛІЗАЦІЇ РОЗПОДІЛЕНИХ ТРАНЗАКЦІЙ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

Текстова частина
магістерської роботи
за спеціальністю „Комп'ютерні науки" 122

Керівник магістерської роботи
д.т.н., доц. Глибовець А.М.

____
(підпис)
"____" _________ 2021 р.

Виконав студент
Кладько. Я.Т.
"____" _________ 2021 р.

Київ 2021

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

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики
к.ф-м.н., доц. Гороховський С.С
____
(підпис)
"_____" ___________2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Комп'ютерні Науки Кладьку Ярославу Тарасовичу
Розробити Методологію реалізації розподілених транзакцій у мікросервісній архітектурі

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Проблематика транзакцій у системах із мікросервісною архітектурою
2 Методологія реалізації транзакцій за допомогою саг
3 Розробка методології для організації системи на основі мікросервісної архітектури із використанням оркестраційних саг
Висновки по роботі та рекомендації для подальших досліджень
Список літератури
Додатки

Дата видачі "_____" __________ 2020 р.

Керівник
А.М. Глибовець, доктор технічних наук, доцент ___________________
(підпис)
Завдання отримав
Я. Т. Кладько ___________________
(підпис)

Тема: Розробка методології реалізації розподілених транзакцій у мікросервісній архітектурі

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	02.11.2020	
2.	Огляд технічної літератури за темою роботи	16.11.2020	
3.	Виконання аналізу сучасних рішень	30.11.2020	
4.	Порівняння інструментів для реалізації бекенд-сервісів	18.12.2020	
5.	Порівняння підходів реалізації оркестраційних та хореографічних саг	18.01.2021	
6.	Вибір інструментів для реалізації оркестраційної саги	15.02.2021	
7.	Вибір інструментів для реалізації CQRS	15.03.2021	
8.	Огляд можливостей Axon Framework для побудови розподілених сервісів та event store	05.04.2021	
9.	Побудова архітектури розподіленого застосунку із використанням оркестраційної саги та event sourcing	12.04.2021	
10.	Створення слайдів для доповіді та написання доповіді	27.04.2021	
11.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	05.05.2021	
12.	Коригування роботи за результатами попереднього захисту	21.05.2021	
13.	Остаточне оформлення пояснювальної записки та слайдів	31.05.2021	
14.	Захист магістерської роботи (проекту)	14.06.2021	

Студент Кладько Я.Т.

Керівник Глибовець А.М. “____”_________________ 2021 р.

Зміст

РОЗДІЛ 1. ПРОБЛЕМАТИКА ТРАНЗАКЦІЙ У СИСТЕМАХ ІЗ МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ 5

1.1 Проблеми паралельного доступу з використанням транзакцій 6

1.2 Проблематика підходів збереження даних.. 9

1.3 Патерни розподілених транзакцій 9

РОЗДІЛ 2. МЕТОДОЛОГІЯ РЕАЛІЗАЦІЇ ТРАНЗАКЦІЙ ЗА ДОПОМОГОЮ САГ........................... 16

2.1 Поняття саги 16

2.2 Організація хореографічних саг 17

2.3 Організація оркестраційних саг.. 19

2.3.1 Моделювання оркестраційних саг на основі скінченних автоматів 19

2.3.2 Механізми забезпечення ізоляції у окрестраційних сагах.. 21

2.4 Domain driven design 22

2.5 Організація CQRS 23

2.6 Event sourcing 25

2.7 Висновок 30

РОЗДІЛ 3. ОРГАНІЗАЦІЯ СИСТЕМИ ІЗ ВИКОРИСТАННЯМ ОРКЕСТРАЦІЙНИХ САГ.................. 31

3.1 Архітектура побудованої системи 31

3.2 Реалізація CQRS-патерну 35

3.3 Обробка помилок та відкат транзакцій 37

Висновок 41

Посилання.. 45

Анотація

У рамках цієї дипломної роботи був проведений аналіз різних методів організації розподілених транзакцій у системах із мікросервісною архітектурою, розглянуті методи зберігання даних у реляційних та NoSQL базах даних для збереження історичності модифікації даних та роботи із окремими моделями для читання та запису, описані механізми семантичного блокування та підходи до моделювання оркестраційних об’єктів керування транзакцією. В результаті були запропоновані підходи та принципи до побудови архітектури.

Ключові слова: системи із мікросервісною архітектурою, розподілені транзакції, CQRS, event sourcing, Axon Framework, read-only database, write-only database, orchestration Saga, choreography Saga, Event Bus, Command Gateway, Query Gateway.

РОЗДІЛ 1. ПРОБЛЕМАТИКА ТРАНЗАКЦІЙ У СИСТЕМАХ ІЗ МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

1.1 Проблеми паралельного доступу з використанням транзакцій

Транзакції у базі даних – робоча операційна одиниця [1] для роботи із базою даних, певна послідовність змін, що виконуються в логічному порядку користувачем або програмою, яка працює із базою даних. Якщо відбувається будь-яка операція із CRUD-інтерфейсу, у базі даних виконується транзакція. Основні концепції транзакцій описуються абревіатурою ACID - Atomicity, Consistency, Isolation, Durability (Атомарність, Узгодженість, Ізольованість, Довговічність), що гарантується системою баз даних.

Атомарність гарантує, що будь-яка транзакція буде зафіксована тільки цілком, кожен крок із транзакції виконається повністю. Якщо одна з операцій в послідовності завершиться із помилкою, то вся транзакція буде скасована. Існує поняття «відкату змін» (rollback), при якому всі зміни у зворотньому порядку будуть скасовуватися. У результаті, кінцевий користувач, у разі виникнення помилки під час проходження транзакції, не побачить ніяких змін.

Під узгодженістю мається на увазі, що будь-яка завершена транзакція або ж транзакція, яка досягла свого завершення, фіксує тільки допустимі результати. При виконанні принципу узгодженості, база даних повинна завжди переходити із одного несуперечливого стану в інший несуперечливий стан. Іншими словами, кожна успішна транзакція за визначенням фіксує тільки допустимі результати. Узгодженість повинна гарантувати інваріативність бази даних – всі дані, запис яких проводиться, повинні відповідати визначеним правилам, обмеженням, зберігати каскадність та відповідати іншим правилами запису даних у базу. Також, варто додати, що умова узгодженості є необхідною для підтримки четвертого властивості транзакції - довговічності.

Під ізольованістю розуміється, що результат кожної транзакції не повинен залежати від виконання інших паралельних транзакцій. На практиці, повна ізольованість важкодосяжна, тому що це – дорога операція. Тому

вводиться поняття «рівні ізольованості», що, по-факту означає, що транзакція в більшості випадків ізольована не повністю.
До чого може привести послаблення ізольованості транзакцій? Це досить відомі проблеми – «брудне читання» (dirty read), «втрачене оновлення» (lost update), «брудний запис» (dirty write), «невідтворюване читання» (non-repeatable read), «фантомне читання» (phantom read), «перекошене читання або запис» (read and write skew)[2]. У системах із мікросервісною архітектурою, де застосовуються транзакції, розподілені між кількома сервісами, проблема ізольованості постає особливо гостро.

Серед рівнів ізольованості визначають такі: read uncommitted, read commited, repeatable read, serializable [3]. Ці чотири рівні ізоляції, визначені стандартом SQL [4], містять конкретні правила, які встановлюють, які саме зміни видно іншим, паралельним транзакціям всередині однієї, а які ні. Перший із рівнів ізоляції – read uncommitted. Він використовується рідко, оскільки його продуктивність та швидкість виконання транзакції не суттєво вища, ніж у інших рівнів. На цьому рівні паралельні транзакції можуть бачити проміжні результати інших транзакцій, тобто відбувається процес «брудного читання». Також на цьому рівні ізоляції можуть з'явитися і інші проблеми, наприклад, невідтворюване і фантомне читання, суть яких буде відкрита у наступному абзаці.

Другий по порядку тип ізольованості це read committed – тут мається на увазі, що нова транзакція побачить тільки ті зміни, які були вже зафіксовані іншими транзакціями до моменту її початку. Здійснені нею зміни будуть невидимими для інших транзакцій, поки нова транзакція не буде виконана повністю. На цьому рівні, як і на попередньому, можлива ситуація невідтворюваного читання (non-repeatable read), тобто при прочитанні одного і того ж рядка в одній транзакції у різний час, повернуться різні результати. Це означає, що під час виконання цієї транзакції, рядок був поміняним або видаленим іншою транзакцією, тому і отримується зовсім інший результат.

Repeatable read - цей рівень ізоляції для у більшості баз даних, наприклад PostgreSQL, MySQL та MariaDB, встановлений за замовчуванням. Він гарантує,

що будь-які рядки, які зчитуються в контексті транзакції, будуть виглядати такими ж при послідовних операціях читання в межах однієї і тієї ж транзакції, однак теоретично на цьому рівні можливий ситуація фантомного читання (phantom read). Вона виникає у разі, якщо транзакція вибираєте певний діапазон рядків, потім інша транзакція вставляє новий рядок в цей діапазон, після чого перша транзакція знову виконує вибірку по цьому діапазону. В результаті утворюється новий, фантомний рядок, який не був присутній під час першої вибірки.

Serializable - найвищий рівень ізоляції, вирішує проблеми фантомного читання, змушуючи транзакції виконуватися в такому порядку, щоб виключити можливість конфлікту, можливість брудного та невідтворюваного читання. Цей рівень блокує кожен рядок, яку транзакція читає. На цьому рівні може виникати безліч затримок і конфліктів при блокування. На практиці цей рівень ізоляції використовується нечасто, оскільки база даних у такому випадку стає «вузьким горлечком» або найбільшим місцем затримок у системі.

Концепція довговічності гарантує збереження даних у випадку, якщо було отримано підтвердження виконання транзакції, а потім відбувся збій системи. При такому варіанті розвитку подій, система забезпечує збереження даних, які були внесені чи змінені транзакцією.

Для роботи із транзакціями у системах баз даних зазвичай існують команди, які дозволяють контролювати стан транзакції – команди, які зберігають зміни, скасовують зміни, створюють безпечні точки, до яких транзакція може відкотитися та дають певній транзакції конкретне ім’я. Також існує журнал транзакцій, який зберігає зміни виконані ними та забезпечує атомарність і стійкість даних у разі збою системи. Журнал містить значення, які дані мали до виконання транзакції, і після її виконання. Це реалізується за допомогою стратегії «записування наперед» (write-ahead log strategy), яка зобов’язує журнал вносити зміни у відповідному порядку. У разі раптової зупинки система бази даних читає список виконаних операцій у зворотному порядку і скасовує зміни, зроблені транзакціями. Таким чином зберігається стійкість транзакцій, які вже були зафіксовані і атомарність перерваної транзакції.

1.2 Проблематика підходів збереження даних

На сьогодні найвідомішою моделлю даних є, мабуть, SQL, заснована на реляційній моделі, запропонованій Едгаром Коддом у 1970 р.: дані впорядковані у сутності або таблиці, де кожна сутність містить невпорядковану за замовчуванням сукупність рядків [5].
Реляційна модель була теоретичною пропозицією, і багато людей у той час сумнівалися, чи можна її ефективно впровадити. Однак до середини 80-х років реляційні системи управління базами даних (СУБД) та SQL стали інструментами вибору для більшості людей, яким потрібно було зберігати та запитувати дані із регулярною структурою. Основною перевагою реляційних баз даних є властивості ACID, про які йшла мова у першому розділі. За допомогою гарантування системою баз даних цих властивостей, стає можливою обробка життєво важливих транзакцій.
Інші бази даних змушують розробників додатків думати про внутрішнє представлення даних у базі даних. Метою реляційної моделі є представлення даних у вигляді зрозумілої структури із доступним інтерфейсом.

Існують моделі, для яких використання NoSQL (not only SQL)-баз даних застосовується краще. Для структури даних, яка описує сама себе, наприклад резюме, представлення її у вигляді JSON-документу є більш відповідним[5].
Також такі моделі, які містять багато інформації про різноманітні деталі певного механізму, наприклад, літака, де кожна сутність не так часто повторюється, краще зберігати у документо-орієнтовану базу даних. Таке «горизонтальне розміщення» даних більш оптимальне для неструктурованих баз даних, на відміну від «вертикального розміщення» даних у реляційних базах даних, де однакових сутностей багато, а, відповідно, і їх екземплярів, які зберігаються у одній таблиці [6].

1.3 Патерни розподілених транзакцій

Розподілені транзакції – на відміну від звичайний сервісів із однією базою даних здійснюються між кількома сервісами. Сервісу часто потрібно

публікувати повідомлення як частину транзакції, яка оновлює базу даних. Якщо сервіс не виконує операції атомарно, збій може призвести до невідповідності у даних. У разі помилки на будь-якому із кроків, транзакція може повернути стан бази даних у попередній.
Однак у випадку мікросервісної архітектури, транзакція може бути розподіленою між кількома базами даних, тобто бути розподіленою транзакцією. Постає логічне запитання - як зберегти транзакцію атомарною? У системі баз даних атомарність означає, що в транзакції виконано або всі кроки або жоден крок не буде виконаний. Система, що базується на мікросервісній архітектурі, за замовчуванням не має глобального координатора транзакцій. Для цього використовують менеджери транзакцій або ж шини подій.

Мікросервісна архітектуру потребує підтримки розподілений транзакцій в силу розподілення бізнес-логіки між окремими сервісами, що відповідає принципу проектування «single responsibility». Традиційним підходом до підтримки консистентності даних у кількох сервісах, базах даних або брокерах повідомлень є використання розподілених операцій. Існує де-факто-стандарт, відповідно до [7], X/Open XA (“eXtended Architecture”) для реалізації розподілених транзакцій методом двофазного коміту. Проблематика цього методу описана нижче. Також до списку проблем розподілених транзакцій можна віднести те, що багато сучасних технологій - бази даних Nosql, наприклад, Mongodb та Cassandra, не підтримують транзакційності. Розподілені транзакції не підтримуються сучасними брокерами повідомлень - RabbitMQ та Apache Kafka. В результаті, при потребі збереження консистентності даних між сервісами, доводиться шукати інші рішення.

Ще одна проблема з розподіленими операціями полягає в тому, що вони є формою синхронної взаємодії між процесами (IPC), що зменшує відсоток доступності. Для того, щоб розподілена транзакція була здійснена, всі сервіси, які беруть участь у ній, повинні бути доступними. Як стверджує Кріс Річардсон у [7], доступність системи є сумою доступності всіх її складових. Якщо розподілена операція включає в себе два сервіси, які на 99,5% доступні, то загальна доступність становить 99%, що суттєво менше. Існує навіть Ерік

Брювача Cap Theo- REM, в якому зазначається, що система може мати лише два з наступних трьох властивостей:

З появою мікросервісної архітектури ми втрачаємо основну властивість реляційних баз даних – ACID [8]. З використанням підходу «database per service» транзакції тепер можуть охоплювати декілька мікросервісів і, як висновок, баз даних. Обробка паралельних запитів теж непроста - об'єкт з будь-якої з мікросервісу зберігається до бази даних, і в той же час інший запит читає той самий об'єкт. Потрібно врегулювати паралельну зміну і читання одного і того ж об'єкту. Які саме дані повинен повернути сервіс - старі чи нові?

Одним зі способів реалізувати транзакції у мікросервісній архітектурі є патерн Transaction Outbox. Уявімо, що у нашій програмі використовується реляційна база даних. Простий спосіб надійно публікувати повідомлення - застосовувати шаблон «вихідних повідомлень» транзакцій. Цей шаблон використовує таблицю бази даних як тимчасову чергу повідомлень. Кожна сутність зберігається як запис у базі даних, і має обов'язковий атрибут - список повідомлень, які потрібно опублікувати. Коли сервіс оновлює сутність у базі даних, він додає повідомлення до цього списку. Атомарність гарантована, оскільки це робиться за допомогою однієї локальної операції з базою даних. Проблема, однак, полягає у ефективному пошуку тих сутностей, які мають події, та їх публікації. Потрібно мати спосіб публікувати події або повідомлення із бази даних до посередника повідомлень. Існує декілька різних способів [7]. Перше – це «публікація голосування» (polling publisher). Цей простий підхід працює досить добре при невеликому масштабі застосунку з використання реляційних баз даних. Використання підходу із NoSQL-рішеннями, залежить від конкретних запитів застосунку до бази даних. Недоліком «публікації голосування» є те, що часто це – дорога операція. Також повинен існувати компонент, який зчитує повідомлення із OUTBOX-таблиці, та публікує їх у брокер повідомлень. Він періодично зчитує повідомлення із таблиці для публікації повідомлень, надсилає їх до брокера повідомлень, який, зі свого боку, надсилає їх до підписаного на повідомлення цієї теми, сервісу. Після цього компонент запускає транзакцію видалення повідомлень із таблиці

OUTBOX. Постійні запити створюють додаткове навантаження на базу даних, тому часто це неефективно, особливо при застосунках чималого розміру. Розглядається ще один спосіб «хвостового логування транзакцій». Суть цього рішення полягає в тому, щоб компонент опрацьовував журнал транзакцій бази даних. Кожна зміна стану об’єкту бази даних представляється як запис у журналі транзакцій бази даних. Компонент журналу транзакцій читає журнал транзакцій та публікує кожну зміну у вигляді повідомлення брокеру повідомлень. Принцип цього підходу описано на рисунку 3 [7].

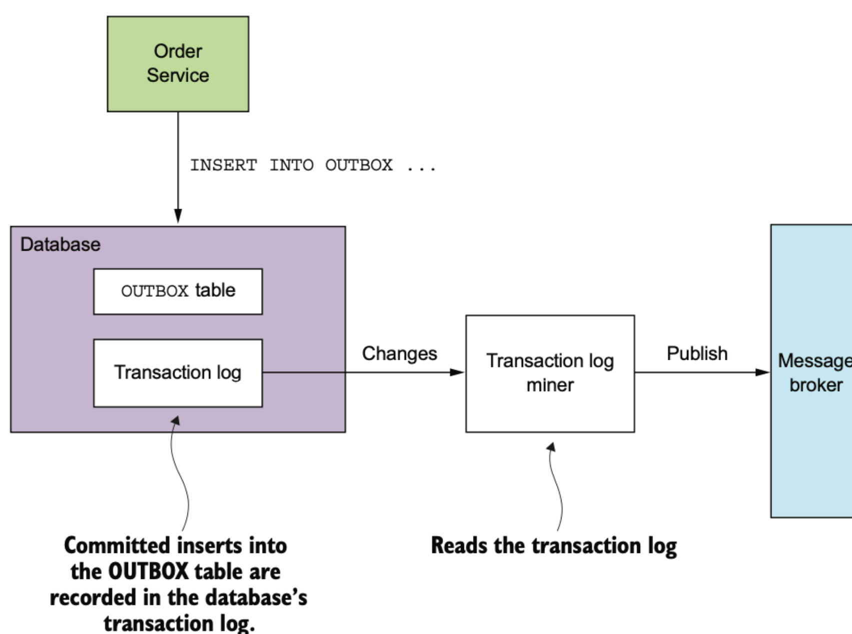

Рис. 3

Кожне повідомлення перетворюється із рядку з бази даних у відповідний об’єкт повідомлення. Існує декілька готових рішень за допомогою цього підходу, наприклад Debezium - проект з відкритим кодом, який публікує зміни в базі даних для брокера повідомлень Apache Kafka. По-суті, це набір різних конекторів для систем баз даних та Apache Kafka. Він базується на принципі CDC (change data capture) та реалізує стрімінг зміни даних на рівні зміни рядків із бази даних із низькою затримкою, високою надійність та доступністю. Останні два пункти досягаються за допомогою використання кластеру Apache Kafka для збереження CDC-подій [9]. Недоліком цього рішення є те, що за допомогою нього фіксацію змін на рівні бази здійснити неможливо. Для деяких систем баз даних функціонал Debezium не відслідковує запити на саму зміну структури бази даних, тобто інформації для запитів типу DDL у брокер

повідомлень не буде надано. Також, у разі використання реплікацій, конектор можна під'єднати тільки до лідерської бази даних, що обмежує функціонал комунікації із не-лідерськими базами даних. Іншим інструментом може бути, наприклад, DynamoDB streams - потоки DynamoDB містять впорядковану за часом послідовність змін - створення, оновлення та видалення, внесених до елементів у таблицю DynamoDB за останні 24 години. Додаток може читати ці зміни з потоку і публікувати їх як події. Однак для вирішення таких проблем існують вже готові рішення.
В іншому випадку неможливо визначити, чи транзакція успішно завершена. Наступні дві моделі можуть вирішити проблему: «двофазовий коміт» (2PC) та паттерн «Сага».

Шаблон «двофазовий коміт» [10] (2PC) широко використовується в системах баз даних. Для деяких ситуацій його можна використати і у мікросервісах. Однак його необхідно використовувати обережно, оскільки не у всіх ситуаціях він підходить. Часом може бути просто непрактичним.
Отже, що таке двофазне фіксація? Як його назва підказки, 2PC має дві фази: підготувати зміни та здійснити їх. У фазі підготовки змін всі мікросервісам пропонується послідовно внести їх до своїх баз даних, причому зробити це атомарно. Після цієї фази мікросервіси вносять фактичні зміни.

Серед переваг можна виокремити підтримку протоколу узгодженості – атомарність та цілісність – консистентність. Також транзакція буде ізольованою, що вирішує вищезгадані проблеми. Зміни не будуть зчитані паралельними транзакціями до того часу, що не внесуться у систему баз даних, оскільки тут працює підхід «read-write isolation». Цей процес є повністю синхронним, тому клієнт буде повідомлений про результат операції – успішний або ні. Однак існує цілий ряд недоліків, найголовніший із яких – блокування. Для синхронного внесення змін, протокол повинен буде блокувати об'єкт, який буде змінений до завершення транзакції. У системі бази даних транзакції, як правило, швидко, як правило, протягом 20 - 100 мс, в залежності від розміру самої транзакції. У мікросервісному середовищі існують також затримки у мережі, тому до загального часу виконання транзакції додається також час,

необхідний для передачі даних між його початковим джерелом та пунктом призначення. Також якщо у сервісі існує інтеграція із платіжними системами, час затримки ще збільшується, враховують повільність та складність роботи платіжних систем[10]. Також, блокування рядків у базі даних створює можливість взаємного блокування, де дві транзакції блокують роботу одна одної. Повне блокування рядків у базах даних для проходження однієї транзакції може стати «горлечком пляшки» або просто вузьким місцем, що відобразиться на загальній продуктивності такої системи. Роботу «двофазового коміту» описано на рисунку 1 [8].

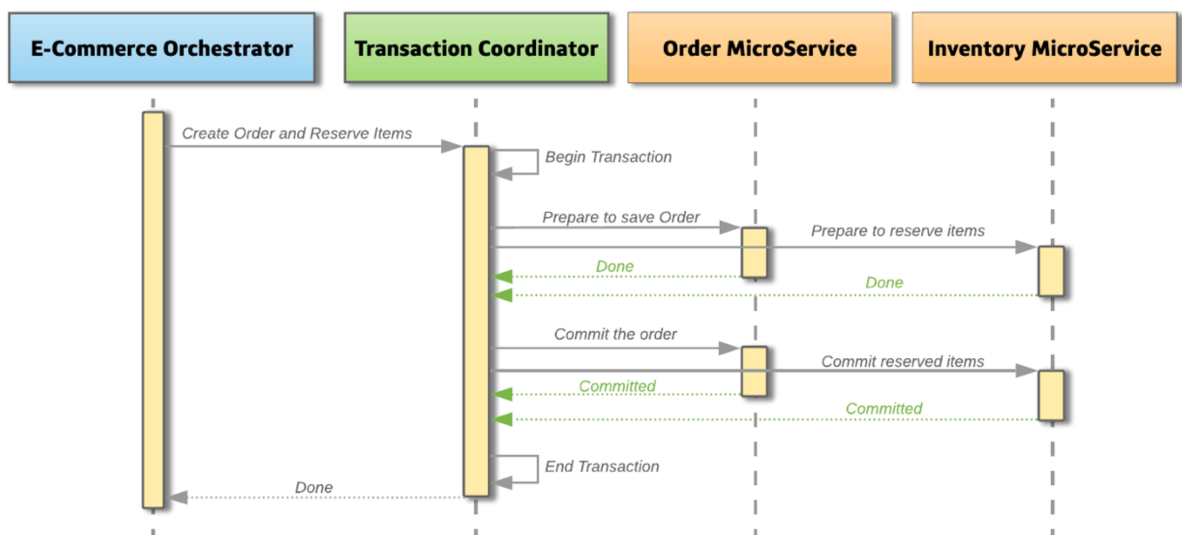

Рис. 1

Шаблон SAGA - це ще один широко використовується шаблон для розподілених транзакцій. Він відрізняється від двофазового коміту, який синхронний. Шаблон SAGA є асинхронним, реактивним та, взагалі кажучи, узгодженим. У шаблоні SAGA розподілена транзакція виконується асинхронними локальними операціями із кожним, задіяним у транзакції, сервісом. Саме спілкування між сервісами здійснюється через шину подій. Кожен сервіс публікує подію, під час оновлення своїх даних. Інший сервіс підписується на цю подію, вона ж, зі свого боку, надсилається до менеджера або шину подій першим сервісом. Коли подія буде отримана сервісом-підписником, він оновлює свої дані. Якщо у будь-якому сервісі виникне помилка, під час виконання транзакції, інші сервіси, у яких частина цієї

транзакції вже відбулася, будуть запускати компенсаційні операції із відкатом змін.

Мікросервісна архітектура з використанням шаблону SAGA передбачає «узгодженість в кінцевому результаті» через свою децентралізованість у збереженні та управління даними, вона позбавлена узгодженості в звичному розумінні цього слова. При традиційній монолітній архітектурі, зрозуміло, що можна оновити дані разом у єдиній транзакції.

Переваги SAGA полягають у практично безпомилковому та швидкому проходженню великих транзакцій, оскільки кожен із сервісів виконує локально лише частину всієї транзакції, та не блокуються під час того, коли інші сервіси виконують інші частини цієї ж транзакції. SAGA є досить складним паттерном для імлементації, особливо, коли кількість мікросервісів, що задіяні у системі, значна. Часто, при застосуванні цього шаблону, використовують менеджер подій, який відповідає за збереження самих подій та розсилання трігерів для сервісів, які прослуховують певний блок подій. Цей патерн має чимало переваг над «двофазовим комітом», але і викликів, через які доведеться пройти команді розробників та тестувальників більше, враховуючи складність цього шаблону. Роботу цього підходу описано на рисунку 2 [8].

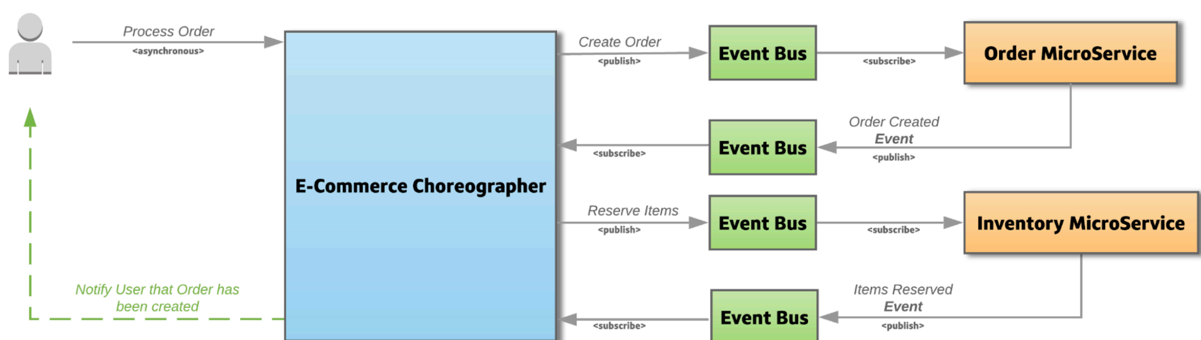

Рис. 2

РОЗДІЛ 2. МЕТОДОЛОГІЯ РЕАЛІЗАЦІЇ ТРАНЗАКЦІЙ ЗА ДОПОМОГОЮ САГ

2.1 Поняття саги

Саги в мікросервісній архітектурі - це механізм для забезпечення узгодженості даних без використання розподілених транзакцій. Він полягає у визначенні команд для оброки даних та виконання операцій із об’єктами, які оновлюються у кількох сервісах. Саги відрізняються від звичайних ACID-транзакцій кількома важливими елементами. Взагалі кажучи, саги не повністю ізольовані, однак існують механізми для забезпечення суттєвого рівня ізоляції. Сага розділяється на три складові – компенсаційні транзакції, опорні транзакції та транзакції, які в загальному випадку не можуть завершитися із помилкою – пошук та вибірка інформації або незворотні операції, логіка яких не може бути відмінена.

Виконання саги передбачає організацію послідовності локальних транзакцій. Кожна локальна транзакція оновлює дані в межах однієї сервісу за допомогою роботи із своєю базою даних та шини подій. Об’єкт, який відповідає за визначення саги також ініціює її перший крок. Після завершення локальної транзакції створюється подія, яка надсилається до шини подій, яку контролює об’єкт саги. Далі сага віддає команду виконання наступної локальної транзакції. Зазвичай така координація відбувається за допомогою асинхронних повідомлень [7]. Важливою перевагою асинхронних повідомлень є те, що вони забезпечують виконання всіх етапів саги, навіть якщо один або кілька учасників саги тимчасово недоступні. У разі виникнення помилки у одному із сервісів, де локальна транзакція не може бути виконана або завершена, сага повинна відновити стан системи до попереднього, який був перед виконанням локальних транзакцій. Цей механізм забезпечують компенсаційні транзакції.

Через забезпечення принципу ізольованості традиційними системами баз даних, будь-яка із транзакцій, що проводиться у ізольованій базі даних, може бути легко відкочена до попереднього стану. Зазвичай використовується оператор «rollback», і система керуванням бази даних скасовує всі внесені транзацією зміни. У випадку використання саг не все так просто – їх не можна

автоматично відкотити назад, оскільки кожен крок фіксує свої зміни в локальній базі даних певного сервісу. Це означає, наприклад, що якщо авторизацію кредитної картки не вдається виконати на четвертому етапі створення замовлення, яке контролюється за допомогою Saga, то механізму необхідно явно скасувати зміни, внесені першими трьома кроками.

Припустимо, що (n + 1) транзакція саги завершилася із помилкою. Отже, наслідки попередніх n транзакцій повинні бути скасовані. Концептуально, кожен з етапів проходження локальних транзакцій, Ti, має відповідну компенсаційну транзакцію Ci, яка скасовує ефекти Ti. Щоб скасувати зміни, внесені першими n-кроками, сага повинна виконати кожен Ci у зворотному порядку. Тому якщо транзакція Tn + 1 не виконалася успішно, це вимагає скасування кроків T1 ... Tn. Сага виконує компенсаційні операції у зворотному порядку до форвардних операцій: Cn ... C1. Завершення Ci має спричинити виконання Ci-1. Важливо зазначити, що не всі етапи потребують компенсаційних транзакцій. Кроки, де здійснюється лише для читання, не потребують компенсаційних транзакцій. Існують опорні транзакції саги (pivot transaction), тому що після них відбуваються кроки, які не можуть не відбутися. Реалізація саги складається з логіки, яка координує кроки саги. Коли сага розпочинає транзакцію, логіка координації повинна обрати першого учасника транзакції та віддати команду на виконання локальної транзакції. Існує кілька різних способів побудови логіки координації саги – хореографічний та оркестраційний.

2.2 Організація хореографічних саг

Хореографічний підхід полягає у розподілі прийняття рішень та, відповідно, послідовності подій серед учасників саги. Комунікація між сервісами відбувається шляхом обміну об'єктами подій. Це реалізація децентралізованого способу координування виконання саги, при якому немає об'єкта, відповідального за прийняття рішень. Учасники саги підписуються на події один одного і відповідають кожен на свою.

Є декілька питань щодо дизайну системи, які ви повинні вирішити.

Перша проблема зв’язана із міжсервісною комунікацією хореографічної саги.

Перше питання полягає в тому, щоб учасник саги оновлював свою базу даних та публікував подію як частину транзакції бази даних. Кожен сервіс повинен оновити свою базу даних та опублікувати відповідну подію. Важливо, щоб оновлення бази даних та публікація події відбувались в межах локальної транзакції. Тобто, транзакція певного сервісу повністю інкапсулює логіку, яку виконує учасник транзакції, і здійснює це атомарно. В іншому випадку сервіс може оновити свою базу даних, а потім, при виконанні іншої частини логіки, завершити операцію із помилкою, перед відправкою повідомлення. Якщо сервіс не виконає ці дві операції атомарно, це приведе до неузгодженості даних. Тут може використовуватися замість розподілених транзакцій, Transaction Outbox або ж Polling Publisher, описані вище.

Друге питання, питання маппінгу, стосується того, що кожна подія повинна містити ідентифікатор структури даних або об’єкту, яким оперує сервіс. Цей ідентифікатор часто називають кореляційним ідентифікатором. Він використовується для відображення подій та об’єктів, що логічно зв’язані транзакцією, на дані, які зберігає сервіс. Для прикладу, якщо клієнт створює замовлення товарів, сервіс, який відповідає за їх видачу, повинен знати, якого саме замовлення стосується запит про видачу товарів, тому у цьому випадку ідентифікатором кореляції буде ідентифікатор замовлення.

Серед переваг імплементації розподілених транзакцій на основі хореографічних саг можна виділити низьку зв’язність. Це підпадає під один із принципів проектування GRASP – loose coupling and high cohesion. Низька зв’язність проявляється у тому, що кожен із учасників транзакції не знає нічого, не містить інформації про інші сервіси, а лише підписується на подій від них. Іншою перевагою є простота – сервіс оновив дані та створив подію, жодної складної логіки, зв’язаної із загальним проходженням саги, тут не передбачено. Але звідси й випливає недолік – через відсутність централізованого місце конфігурації саги, розуміння її стає суттєво складнішим. Логіка проходження саги є зоною відповідальності всіх учасників транзакції, тому розробникам, які

працюють із таким підходом, потрібно спершу зрозуміти, як взагалі визначається сама сага.
Також існує проблема циклічних залежностей сервісів. Досить часто це не критично, але для систем із таким дизайном рекомендується провести повторну декомпозицію [8].

Іншим недоліком є те, що при складній логіці проходження транзакції може втратитися її основна перевага – низька зв'язність. Якщо у транзакції беруть участь багато сервісів, кожному учаснику саги потрібно підписатись на всі події, які їх стосуються. Відповідно, існує ризик того, що при оновленні одного сервісу, потрібно буде оновлювати всі події або об'єкти інших сервісів, які взаємодіють із оновленим сервісом. Як висновок, хореографічні саги краще підходять для реалізації простих транзакцій у системах із мікросервісною архітектурою.

2.3 Організація оркестраційних саг

2.3.1 Моделювання оркестраційних саг на основі скінченних автоматів

Іншим способом організації саги є оркестраційний метод. Оркестрація полягає у централізації логіки координації саги у одному об'єкті. Координатор саг надсилає повідомлення-команди учасникам саги, в залежності від операцій, які необхідно виконати. Комунікація між сервісами відбувається за допомогою відправки асинхронних повідомлень.
Причому, як наголошує Кріс Річардсон у [10], це не реалізація «pub-sub»-паттерну. Після виконання своєї частини логіки транзакції, сервіс надсилає подію з результатом її виконання та необхідними даними. Оркестратор на основі цієї інформації, вирішує, що відбудеться далі. Кожен крок саги, що базується на оркестраційному підході, складається із сервісу, який бере участь у проходженні транзакції, оновлення бази даних та публікації повідомлення у шину подій, за якою стежить об'єкт оркестратора.

Моделювання оркестраційних саг може проводитися як і моделювання скінченних автоматів, принцип роботи – такий ж.
Як відомо, скінченний автомат – це п'ятірка M = <Q, E, сигма, q0, F>, де

Q = {q0, q1, …, q n-1} - скінченна множина станів автомата;
E = {a1, a2, …, am} – скінченна множина вхідних символів або ж вхідний алфавіт;
Сигма – відображення множини Q*E у множину P(Q), або ж функція переходів;
q0 є (із множини) Q – початковий стан автомата;
F c Q (q підмножина ф) – множина заключних станів, де елементи з F – заключні або фінальні стани [11].

У нашому випадку множиною Q станів саги є множина всіх сервісів, які беруть участь у виконанні транзакції. Початковим станом саги є метод, який створює об’єкт саги. Вхідним алфавітом є список команд, які віддає координатор транзакції. Переходи між станами викликаються подіями, які містять інформацію про завершення локальної транзакції певного сервісу. Кінцевими станами є метод, що завершує сагу із успішним виконанням проведених операцій, або метод, який сигналізує про виникнення помилки в ході проведення транзакції та проведену компенсацію змін. Як висновок, моделювання оркестраційних саг на основі детермінованих скінченних автоматів полегшує процес впровадження, проектування та тестування саг.

Серед переваг саг, спроектованих на основі оркестраційного підходу, можна виділити зрозумілу логіку виконання транзакції. Централізованість логіки у одному об’єкті є перевагою через реалізацію концепції «простоти коду», на якій наголошує «дядько Боб», Роберт Мартінс у [12]. Координатор транзакції орієнтований на читача-програміста, і саме завдяки цьому його легко зрозуміти та змінити. Також хочеться зауважити, що існує низка проблем розподілених транзакцій, які полягають у тому, що відкотити зміни під час розподіленої транзакції буває складно, дані можуть бути каскадно видалені, одноразові тригери у базі даних чи додатку можуть спрацювати. У оркестраційному підході такий ряд проблем вирішується збереженням стану проходження транзакції у сховище.

Іншою перевагою є те, що при складній логіці проходження транзакції у системі із мікросервісною архітектурою залишається низький ступінь зв’язності, оскільки одним сервісам не потрібно знати інформації про інші, ця

відповідальність покладена на об’єкт, який координує сагу. У разі змін у одному із сервісів, достатньо поміняти тільки його та координатора. Також, серед переваг окрестраційної саги є відсутність утворення циклічних залежностей. Через централізованість керування транзакцією, сервіси не підписуються на події один одного, а роблять це через шину подій, внаслідок цього, вони не залежать один від одного. Як рекомендує Кріс Річардсон у [8], класи саги повинні бути простими та зрозумілими. Через нагромадження великої кількості логіки, клас саги може перетворитися на складний об’єкт, і щоб цього не допустити, краще провести декомпозицію системи із розбиттям складної логіки, на блоки простішої, зокрема, делегувати виконання методів та користуватися абстрактними класами.

2.3.2 Механізми забезпечення ізоляції у окрестраційних сагах

Взагалі кажучи, механізм саг гарантує ACD і неповну ізоляцію. Через проблеми із ізоляцією, можуть виникати аномалії даних, суть яких описувалася вище. Це означає, що результат паралельного виконання саг, може не дорівнювати послідовному виконанню транзакцій. Для того, щоб запобігти неузгодженості даних використовується декілька механізмів, які важливо виокремити.

Перший із них – використання семантичних блокувань. Він полягає у логічній зміні поля у сутності, яке буде свідчити про те, що із об’єкт на поточний момент бере участь у транзакції. Наприклад, якщо існує чек із товарами, і цей об’єкт чекає, поки прийде підтвердження із сервісу оплати. У чеку може бути поле статусу, яке буде містити значення «PENDING». У випадку, якщо метою іншої транзакції буде зміна списку товарів у чеку, логіка семантичного блокування спрацює і список товарів у чеку залишиться незмінним. Після проходження першої транзакції, значення поля поміняється, і об’єкт буде доступним для інший транзакцій.

Тому, якщо в транзакції використовуються операції, для яких неможливо здійснити відкат або створити компенсаційну транзакцію, то їх краще

помістити у кінець транзакції. Це може бути спрацювання тригерів у базі даних, надсилання е-мейлу або інші незворотні операції.

Для компенсаційних транзакцій рекомендується використовувати комутативні оновлення. Це допомагає реалізувати роботу механізми повернення системи до попереднього стану. Якщо мова іде про банківський рахунок, комутативність можна забезпечити парою операцій – дебет та кредит. При помилковому завершенню однієї, координатор транзакції віддає команду розпочати іншу.

2.4 Domain driven design

Як і у випадку мікросервісної архітектури, у event sourcing одним із важливих елементів, які необхідно роглянути, є доменно-орієнтований дизайн (domain-driven design). Це підхід, який націлений на вивчення предметної області конкретних бізнес-задач та бізнес-процесів. Однією із характерних його особливостей та вимог для дотримання є простота. Застосування доменно-орієнтованого дизайну повинне знижувати складність на нагромадження бізнес-логіки у одному місці настільки, наскільки це можливо. Команди розробки, пишучи код, більше уваги приділяють технологіям та інфраструктурі. Підхід DDD, навідміну, говорить про те, що бізнес – важливіший, і його вимоги повинні бути у пріоритеті [13]. Тому, виділяють два важливих елементи доменно-орієнтованого дизайну – єдину мову (ubiquitous language) та контекст прредметної області (bounded context).

Часом, у професії програміста, сформулювати вимоги до продукту буває складніше, ніж реалізувати його. Різні предметні області складні по-своєму, тому вводиться поняття єдиної мови, як механізм боротьби зі складністю. Її суть полягає у тому, щоб називати речі однаково як в специфікації, так і в коді. Вона може виглядати як псевдокод, зате із легкістю розуміється навіть нетехнічним спеціалістом. Однак слід врахувати, що розробка єдиної мови для конкретної предметної області – не безкоштовна. Її процес створення вимагає клопіткого та ретельного аналізу вимог і консультацій з експертами предметної області. Розробляючи програми в рамках єдиної мови багато часу інвестується

у створення внутрішньої бази знань клієнта. Проблема в тому, що майже вся ця робота не закладена в бюджет на розробку застосунку. Але часто така інвестиція себе виправдовує [14].

Обмежений контекст - ключовий інструмент доменно-орієнтованого дизайну, це явна межа, всередині якої міститься модель предметної області конкретного продукту. Вона створює відображення єдиної мови у ієрархію модулів програмного забезпечення. Контексти реалізовують принцип «розділяй і пануй», це досягається методом «розрізу» предметної області. Зрозуміло, що єдина мова не може описувати всі поняття і терміни у лише одному контексті. Для повного розуміння, необхідно висвітлити поняття у кількох контекстах, а це, зі свого боку, означає, що один і той ж термін у різних контекстах може означати різні речі. Однак, через реалізацію принципу «розділяй та пануй» за контекстами, програму значно легше ділити на пакети та компоненти. Такий розподіл реалізовує принцип проектування, при якому зберігається низька зв’язність модулів.

2.5 Організація CQRS

CQRS - ідея концепції досить стара, вона ще виникла на основі того, як створювалося проектування методів. Є хороше правило і практика, яке полягає у тому, що метод повинен виконувати або команду або запит. Якщо він виконує і це і інше, відповідно, цей код буде важко підтримувати і читати. У такому випадку є сенс переглянути, дизайн методу. Зараз ця сама концепція використовується на більш високому і абстрактному рівні.

Припустимо, що існує API, і всі операції змін відбуваються через спеціально відведену для цього модель команд. Створення, зміна чи видалення об’єктів відбувається через командну модель. Під час побудови командної моделі, одним із основних об’єктів є агрегат. В ролі агрегату використовується модель, яка містить у собі набір полів та характеристик кількох сутностей. Ця концепція бере свій початок від DDD, описуючи себе як "групу асоційованих сутностей, які використовуються для змін даних і діють як одна одиниця". Крім того, зовнішні посилання на агрегат обмежуються одним членом, позначеним

як корінь агрегату. Набір правил узгодженості даних застосовується в межах одного агрегату. Один агрегат не може охоплювати всю систему, використовуються менші керовані групи сутностей. Якби він охоплював великий шматок всієї системи, не було б жодного способу гарантувати високий ступінь узгодженості та одночасно працювати, оскільки весь агрегат повинен бути заблокований для кожної операції [15]. Вводиться, також, концепція моделі для читання, завдання якої полягає у відповіді на запити. Читання, пошук та отримання вибірки за певними критеріями – це зона відповідальності другої моделі.

Ці дві моделі можуть міститися у різних репозиторіях, тому команди розробки можуть працювати розподілено і оптимізовувати роботу своїх моделей. Тим не менш, важливим кроком є синхронізація моделей. Якщо вони не синхронізовані, дані будуть неактуальні, що неприпустимо. Конкретно ідеться про те, щоб дані із бази для збереження історії об’єктів, із якою працює командна модель були синхронізовані із базою даних для читання. Цей процес може нагадувати матеріалізовані представлення у базі даних. Принцип роботи дуже схожий.

Матеріалізоване представлення зберігає дані, повернуті запитом про визначення представлення, і автоматично оновлюється після змін даних таблицях бази. Це підвищує продуктивність та швидкість виконання складних запитів, наприклад, із багатьма об'єднаннями та агрегатними функціями. Завдяки можливості автоматичного зіставлення SQL-запиту, виконання матеріалізованого представлення не потрібно вказувати в самому тілі запиту, оптимізатор виконання запитів автоматично врахує його, при поверненні результату [1]. Ця можливість дозволяє реалізувати матеріалізовані представлення як механізм підвищення швидкості запиту.
Це також працює як наслідок того, що, наприклад, у Oracle вже існує вбудований механізм для створення матеріалізованих розрізів на рівні мови PL/SQL командою «CREATE MATERIALIZED VIEW». У Transact-SQL, який використовується в SQL Server, такого механізму немає, але це не означає, що створити матеріалізоване представлення в цій системі керуванням баз даних

неможливо [16]. Як випливає з визначення, щоб представлення працювало як матеріалізоване, воно повинно зберігати результати виконання запиту на фізичному рівні. У SQL Server це забезпечується шляхом створення кластерного індексу для кожного представлення. Враховуючи, що результати зберігається фізично в індексі, час звернення до них зменшиться.

Серед мінусів використання матеріалізованих представлень у CQRS є те, що логіка синхронізації зашивається саме у базу даних та контролюється її розробниками. Це важко тестувати і це впливає на загальну продуктивність та швидкодію системи. У CQRS логіка синхронізації лежить саме в у додатку, який розроблюється. Переваги CQRS у проектуванні систем за цих паттерном у випадках, коли тільки читання або тільки запис здійснюється під великим навантаженням. Розділення системи на читання і запис дозволяє масштабувати тільки те, що дійсно вимагає масштабування. Також, серед переваг можна виділити те, що якщо буде існувати два окремих сервіси і один з них стане недоступним, то шансів що другий буде доступний - набагато більше, і також навпаки. У моделі для читання можливо представити наші агрегати як цього буде вимагати бізнес-логіка - зробити окрему структуру для списків, звітів та, наприклад, графіків і, як результат, можливо підготувати агреговані об’єкти, які найкраще будуть описувану нашу предметну область.

2.6 Event sourcing

Традиційні додатки працюють з фінальним станом об’єкту. Після отримання об'єкту, він зберігається у базу. Після зміни об’єкту, ми не володіємо інформацією про попередній стан об’єкту. Ідея event sourcing – проста - ми не зберігаємо фінальний стан, ми зберігаємо низку подій. Якщо прийшов запит на створення, ми додали першу подію у event store. Після проходження певного часу, додається наступна подія для об’єкту, але вона містить тільки різницю того, що змінилося. Такий процес відбується абсолютно на кожну дію, яка відбувається із агрегатом до того моменту, поки не наступить остання подія. По-суті, event sourcing - це про збереження конкретної послідовності подій. Підхід полягає у тому, що всі дані про зміни зберігаються

у одному сховищі – у event store. Система запам'ятовує всі дії користувачів саме в тій послідовності, в якій вони прийшли. Кожна подія це маленька дельта змін, частина історії об’єкту. Це може сильно нагадувати систему контролю версій, наприклад, гіт – є комміти та їх історія. У кожному із них, можна подивитися, що ж змінилося. Якщо потрібно відновити стан коду на якийсь момент, це теж можна зробити із легкістю. Дельти змін у event sourcing це саме, те що змінилося в рамках певної події. Визначають також ряд правил в event sourcing – не можна видаляти та змінювати події. У іншому випадку можна втратити наріжний камінь event sourcing – історичність. Відновити стан після зміни або видалення попередніх подій було б неможливо або дуже важко.

Прикладом event sourcing із життя слугуватиме влаштування на роботу. Коли людина влаштовується у компанію із нею укладають контракт в ньому прописана її зарплата, посада, зона відповідальності та інші дані. Після певного часу, зарплата зростає і компанія дає доповнення до контракту, часто старий договір не переписується [17]. Потім міняється посада, і компанія дає ще одне доповнення до контракту. Для того щоб зрозуміти яка ж ситуація на поточний момент, який стан об’єкту «контракт», можна взяти його із всіма доповненнями і отримати всю історію інформації. Можна так само дізнатися що було рік чи два роки тому із самого моменту створення.

Поняття із event sourcing, яке теж потребує розгляду, називається «знімок» (snapshot). Він необхідний для того, щоб не перераховувати всі ті події які відбувалися із об’єктом протягом його історії. Він об'єднує всю інформацію, яка здійснювалася із об’єктом протягом певного часу. Іноді не потрібно зберігати всю історичність даних, достатньо лише її частину. Також, серед причин його виникнення чітко окреслюється проблема продуктивності застосунку. Щоб виконати команду із об’єктом, необхідно повністю відтворити його поточний стан, а отже, пройтися по списку всіх подій агрегату та послідовно їх відтворити. Для того, що б не відновлювати всі події, які відбулися раніше для певного об’єкту, на визначеному етапі створюється знімок. Часто такий етап – це певна кількість подій, після яких зберігається поточний стан об’єкту, а попередні події можна видалити.

По-друге це зміна стану. Якщо розглядати приклад із конференцією, то існує декілька кілька варіантів, коли можна зберігати фінальний стан події. У класичному підході, для збереження кінцевого стану, інформація записується у базу даних. Відбувається зміна характеристики об’єкту, наприклад зміна місця проведення конференції. Старий запис дістаємо і перезаписуємо, якщо змінилося хоча б одне поле. При кожній новій зміні, процес знову повторюється. При будь-якій дії у випадку event sourcing,
ми б робили все приблизно так: для початку ми додаємо першу подію,
яке несе базову інформацію про об’єкт, після цього додаємо другу подію
і важливий моментом буде звернути увагу на назву цієї події – вона описує зміну, характерну для цього об’єкту. Не менш важливою складовою події є сама дельта, наприклад, нове місце проведення конференції. Іноді бувають ситуації, коли не потрібно зберігати навіть саму дельту - ім'я події вже пояснює що ж помінялося. Наприклад, якщо подія називається «ConferenceCancelled» за її іменем зрозуміло, що статус події повинен помінятися на «CANCELLED» і у такому випадку дельта не потрібно.

Event sourcing і CQRS добре поєднуються. У такому разі визначається модель для команд і всі події будуть зберігатися у event store. Читання буде відбуватися через модель для читання, запис – через модель для команд, при цьому потрібно не забувати синхронізувати дві моделі.
Можна використовувати під зберігання подій SQL або NoSQL бази даних - тут вибір не обмежується і є незалежним. Серед мінусів це постійна склейка подій, відтворення стану об’єкта, але для вирішення цієї проблеми існують снепшоти та CQRS.

У рішеннях, де важливо зберігати історію об’єктів, можна застосувати Hibernate Envers. Він генерує додаткові записи до основних таблиць, де описуються зміни, проведені у них. Також існує Audit4J. Він не часто оновлюється, тому не підійде для бізнес-рішень. Події в основному зберігаються у форматі JSON, хоча теж можна використати як і XML. Якщо використовується event sourcing – то історичність, аудит та логування зразу наявні у додатку – все в одному, все «із коробки»[7].

Важливо розуміти основну ідею domain-driven design - код і додаток повинні максимально розповідати про те, яку предметну область ми вирішуємо. Один із основних термінів з «domain-driven design» це сутність – елемент, основний критерій якого це ідентифікатор за допомогою якого ми можемо відрізнити її, порівняти з іншими сутностями. Є, також, value-об'єкт він схожий на сутність, але сам по собі він не представляє ніякої значущості. Тому він або належить сутності або належить якомусь агрегату. Що ж таке агрегат? Агрегат - ця група сутностей, пов'язаних одним корененевим елементом. Агрегат може належати сутності, а може належати кореневому елементу. Останній термін, який потрібно розглянути у event sourcing – домен, він складається із декількох агрегатів. Якщо взяти, для прикладу, конференцію - вона складається з доповідей, спонсорів та документів. Якщо ми захочемо в якийсь момент порахувати скільки коштує ця конференція, скільки грошей було витрачено, ми додаємо ще один агрегат який називається «накладна» і, відповідно, щоб зв'язати ці два агрегати, створюється посилання між накладною та конференцією. Одним із правил у domain-driven design є те, що якщо існує сторонній кореневий елемент, то він може посилатися тільки на корені інших агрегатів, але не може створювати чи тримати посилання на конкретні елементи, які лежать під коренем.

Побудова стану агрегату – отримується список станів об'єкту, подій, які відбулися із цим об'єктом. В залежності від списку подій, ми отримуємо агрегат у певному стані. Кожна подія зберігається у JSON. Важливим етапом створення події є можливість створювати обмеження даних. У AXON сервері або у PostgreSQL доступна можливість створювати складені ключі для подій. Зазвичай у ролі ключа виступає ідентифікатор агрегату та порядковий номер події. У event sourcing місцем збереження подій можуть застосовуватися Redis, MongoDB, Elasticsearch та інші інструменти. Якщо важливий швидкий запис – зазвичай дані не зберігаються у одній таблиці, відповідно до нормалізації та, як мінімум, першої нормальної форми. Дані пишуться у різні таблиці, але потрібно пам'ятати, що запис – це далеко не завжди створення нових рядків,

але і їх оновлення. Операція оновлення займає більше ресурсів, ніж операція створення [17].

Переваги event sourcing – немає оновлень, тільки операції вставки, одна таблиця для збереження подій, немає обмежень даних, оскільки вони зберігаються в JSON, також немає індексів, оскільки в них немає потреби. Одним із підводних каменів є паралельна зміна стану агрегату. У традиційному підході проводиться просте блокування фінального стану об’єкту і почергово зносяться зміни. Але у випадку event sourcing фінального стану немає, тому необхідно придумати якийсь механізм вирішення конфліктів. Припустимо, що одночасно приходять два запити на зміну стану агрегату – складеним ключем кожного із них є ідентифікатор агрегату та номер події, де у випадку двох одночасних подій їх номери будуть однакові. Тут на допомогу приходить порівняння типів подій. Якщо типи подій різні, тобто кожна із них несе зміни, які не несе інша, тоді для будь-якої із подій оновлюється її номер – збільшується на один, і дві події зберігаються у базу подій. У протилежному випадку, якщо типи подій однакові, потрібно їх обробити відповідним чином. Якщо зміни даних у двох подіях пересікаються, не завжди можна визначити, якій події потрібно віддати перевагу. У такому випадку, можна на одну із них повернути помилку. Якщо все ж конфлікт можна визначити на семантичному рівні, наприклад, якщо два пацієнти записалися на однаковий час на прийом до лікаря, одного із них можна посунути на наступний вільний слот, зразу після закінчення прийому першого пацієнта, знову ж таки, способом збільшення номеру події – ця логіка вже залежить від конкретних бізнес-вимог. Для реалізації такої логіки вже існують інструменти, наприклад, Spring Retry.

Іншим підводним каменем є зернистість подій. Якщо дві події крупнозернисті, наприклад, оновлення даних і їх видалення. Тут приблизно зрозуміло, що робити – якщо видалення прийшло після оновлення даних, то операції здійснюються у логічному порядку. Якщо ж навпаки – це погано, треба розбиратися і вникати у бізнес-логіку, щоб зрозуміти, як система повинна реагувати у такому випадку. Зрозуміло, що не можна оновити сутність, видалення якої відбулося у попередній події. Але є інший варіант розвитку

подій – якщо прийшли дві операції оновлення. Тут тип конфлікту неочевидний, і треба дивитися на дельту змін. Цей тип подій називають дрібнозернистим [17]. Якщо для певної резервації прийому до лікаря прийшла подія зміни імені пацієнта, і також одночасно у іншій події відбулася зміна заголовку резервації слоту, наприклад, те, що у пацієнта не лише висока температура, але й болить горло – ці дві події несуть менше інформації, а, отже, вирішити такий конфлікт буде легше. Чим менша дельта змін у події, тим швидше можна вибудувати збереження стану об’єкту у певній послідовності.

2.7 Висновок

Event sourcing – підходить для подійних моделей, коли важливо зберігати кожен стан, в якому перебували об’єкти. Наприклад, організація черг або запис до лікаря – подія створення графіку прийому, резервування пацієнтом, зміна часу резервації, її відміна. Для таких моделей event sourcing підійде краще, однак існують моделі, для яких його застосування буде нераціональним. Наприклад, коли існують форми із збереженням великої кількості даних, які рідко міняються – тут використання event sourcing не буде оптимальним. Із життя це може бути приклад із особистою інформацією у Facebook – більшість полів заповнюються один раз і рідко міняються.

Цих пунктів достатньо для виокремлення найважливіших деталей реалізації саг. Завдяки підходу із використанням доменно-орієнтованого дизайну, легко описати процес проходження транзакції на зрозумілій для людей мові. За наявності кількох сервісів, виокремлюється один, як найбільш логічна сутність для організації класу-менеджера, що керуватиме процесом проходження транзакції. Зрозумілість кожного кроку транзакції, спроектованого на моделі скінченного автомату, дає можливість швидко реалізувати бізнес-логіку та надати замовнику результати.

РОЗДІЛ 3. ОРГАНІЗАЦІЯ СИСТЕМИ ІЗ ВИКОРИСТАННЯМ ОРКЕСТРАЦІЙНИХ САГ

3.1 Архітектура побудованої системи

Реалізація системи із використанням оркестраційної саги. Система побудована на основі традиційної мікросервісної архітектури, у якій існує шість мікросервісів та один незалежний модуль із спільною для різних сервісів логікою. Архітектура виконана із використанням підходу, у якому використовується одна база даних для кожного із сервісів («database per service»).

За потреби мікросервісу можна масштабувати. Наприклад, якщо навантаження на один сервіс збільшується, можна запустити його екземплярів одному сервері. Такий паттерн називається «multiple instances per host». Але якщо фізично на одному сервері запускається декілька сервісів, потрібно, щоб порти для з’єднання були різними. До реалізації цього ми повернемося пізніше.

Eureka Server - сервер імен або реєстр сервісів, який зберігає інформацію про всі сервіси, які використовуються у системі. Кожен сервіс реєструється на сервері Eureka – відправляє повідомлення про те, що він активний. У реєстрі міститься інформація про всі клієнтські застосунки, наприклад, інформація про транспортний шар кожного із сервісів - на якому порті працює застосунок, яка у нього IP-адреса та назва. Eureka Server відомий як Discovery Server. На цей момент також існують його аналоги, наприклад, Zookeeper, Consul та Cloud Foundry [18].

Мікросервісна архітектура складається із різних патернів проектування, деякі із них використані і детально розглядаються у цій роботі. Реалізація сервісів виконана на мові Java із застосуванням Spring Boot. Для реєстрації сервіс повинен бути позначений як @EnableEurekaClient, а сервер - @EnableEurekaServer.

Мінімальні налаштування, які дозволяють скористатися сервісом, представлені далі. Для кожного екземпляру сервісу генерується випадковий порт, вказуються параметри доступу до бази даних та використовується процесор, який підписується на події, які генеруються потоком, що працює із

агрегатами та обробниками подій без використання інших потоків. У файлі application.properties:

```
eureka.client.service-url.defaultZone = http://localhost:8761/eureka
spring.application.name = products-service
server.port = 0
eureka.instance.instance-id = ${spring.application.name}:$
{instanceId:${random.value}}
spring.datasource.url = jdbc: mysql://localhost:3306/products
spring.datasource.username = root
spring.datasource.password =
spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto = update
spring.error.include-messages = always
spring.error.include-binding-errors = always
axon.eventhandling.processors.product-group.mode = subscribing
logging.level.org.axonframework.axonserver.connector.event.axon.AxonServerEvent
Store = DEBUG
```

Управління різними налаштуваннями конфігурації для кожного із сервісів стає складнішим пропорційно до росту кількості запущених сервісів. Рішенням є використання централізованої конфігурації у окремому сервісі конфігурацій. Це мінімальні необхідні налаштування для запуску Eureka Server. Він надає дружній користувацький веб-інтерфейс, де адміністратор системи може слідкувати за станом сервісів. У проекті «Eureka» виділяються такі поняття: Eureka Server, Eureka Service, Eureka Instance та Eureka Client [19]. Суть сервера зрозуміла, тому потрібно розуміти чим відрізняється, наприклад, сервіс від клієнта. Клієнт – будь-який додаток, який створює запит на сервера, із метою отримати інформацію про всі доступні сервіси. Сервіс зі свого боку, це група екземплярів одного застосунку, які містяться у реєстрі Eureka Server, він має свій ідентифікатор та може тримати посилання на один або декілька екземплярів сервісу. Eureka Instance – екземпляр сервісу. Якщо не існує

реплікації сервісів, один сервіс це Eureka Instance, Eureka Client та Eureka Service одночасно.

Якщо реплікація існує, то потрібно, щоб кожен екзепляр запускався на своєму окремому порті, це налаштування створюється за допомогою Eureka API, його можна визначити у файлі application.properties для кожного сервісу. Генерація порту для запущених екземплярів відбувається автоматично. Однак тоді постає запитання як клієнтській програмі, якою користується кінцевий користувач знати, до якого саме сервісу звернутися, якщо інформація про його розташування визначається у runtime? Рішення полягає у використанні API-шлюзу. Це реалізація інтеграційного паттерну мікросервісної архітектури, відомого як «API Gateway». Шлюз API діє як центральний пункт входу для всіх запитів які надсилаються до системи. Потрібно лише знати адресу самого шлюзу, а його зоною відповідальності буде зрозуміти, куди далі маршрутизувати запит.

Для цього API Gateway дізнається місцезнаходження кожного сервісу через комунікацію із сервісом виявлення. Якщо існує декілька екземплярів певного сервісу, то API Gateway повинен балансувати запити між ними для рівномірного завантаження кожного із екземлярів. Було використано окремий модуль «sping-cloud-starter-gateway» для імплементації центрального шлюзу. API цього модулю має вбудований балансувальник навантаження, який розподіляє запити, що надходять від клієнтських програм, однаково між екземплярами кожного сервісу. Це робить наш доступність системи більш надійною, коли навантаження зростає. У шлюзі є налаштування фільтрів, які, наприклад, можуть використовуватися для перевірки прав доступ користувача до певних сервісів. Є вбудовані фільтри, але також існує можливість створювати власні і виконувати у них будь-яку логіку. Наприклад, після перевірки публічного токену JWT, сервіс може згенерувати новий внутрішній маркер JWT та для всіх запитів із таким маркером відправляти їх по спеціально визначеному окремому маршруту.

API Gateway відсилає запит до сервісу замовлень. Він містить клас, який контролює проходження саги. Цей клас виділяється, як логічна одиниця, яка є

центральним об’єктом у транзакції. Інші сервіси – залежні, і вони тільки виконують команди. Спершу об’єкт замовлення створюється всередині сервісу замовлень, на основі інформації від користувача. Контролер, який приймає такий тип запитів надсилає інформацію до сервіса, який, зі свого боку, ініціалізує об’єкт замовлення. Після його ініціалізації, сервіс надсилає подію про створення замовлення у шину подій. Клас, який координує сагу, реагує на цей тип події і розпочинає транзакцію. Він дістає список продуктів конкретного замовлення та надсилає його сервісу продуктів для резервації через шину команд. Сервіс продуктів, який прослуховує шину команд, отримує команду і, після перевірки достатньої кількості продуктів на складі, міняє інформацію про них, зменшуючи кількість доступних продуктів. Після цього він надсилає подію у шину подій про те, що продукти для конкретного замовлення зарезервовані. Клас координатора, після отримання події про успішну резервацію, надсилає запит у користувацький сервіс, для отримання даних про користувача через шину запитів. Сервіс, який відповідає за обробку інформації про користувачів системи, дістає із сховища платіжні дані певного користувача, та повертає їх за допомогою об’єкту CompletableFuture<UserDetails>. Комунікація між цими сервісами здійснюється у асинхронний спосіб. Центральний клас проходження саги після отримання даних користувача, надсилає запит у платіжний сервіс, де моделюється оплата товарів за, наприклад, технологією LiqPay, та виставляє дедлайн оплати. Об’єкт дедлайну контролює валідність оплати протягом трьох хвилин, і якщо за цей час клас саги не зміг отримати відповідь від сервісу оплати, транзакція завершується із помилкою та починається процес відкату. Комунікація між сервісом замовлень та сервісом оплати здійснюється через шину команд у синхронний спосіб, де сервіс оплати блокує потік виконання до момент отримання результату. Протягом всього цього часу об’єкт SubscriptionQueryResult<I, U>, де I – це повідомлення, на яке очікує об’єкт від класу координатора, а U – значення інкременту для агрегованого об’єкту, тримає контроль над запитом про створення об’єкту замовлення у базі для читання. Його поява у моделі для читання свідчить про успішне виконання

транзакції. Якщо перед його появою, виникає помилка виконання транзакції, це повідомлення одразу ж відправляється користувачу. Якщо модель для читання оновилася, дані із неї відправляються у контролер моделі для запитів через шину запитів, який повертає результат виконання транзакції.

3.2 Реалізація CQRS-патерну

Побудована система реалізує мікросервісний паттерн архітектури для баз даних – CQRS. Ми можемо кожен запит, який надсилається до нашої системи класифікувати як операцію створення, читання, оновлення або видалення. Отже, можливо розподілити ці запити до двох моделей – одна для пошуку та читання інформації, інша для запису. У термінах CQRS розрізняють три типи повідомлень, кожне із яких представлене у коді відповідними класами – команда (command), запит (query) та подія (event). Подія створюється як реакція на зміну стану системи. Якщо небхідно модифікувати об’єкт, створюється команда, яка відсилається у шину команд через інтерфейс CommandGateway із модуля «org.axonframework». Команда означає намір викликати дію. Запит – об’єкт, який контролюється класом саги, він містить всю необхідну інформацію для формування, наприклад, SQL-запиту для того, щоб отримати результат вибірки із бази даних. Агрегування команд та запитів здійснюється в межах одного сервісу. Архітектура такого сервісу буде складатися із модуля для читання і модуля для змін, і, за потреби, може бути розділена на два окремих ще менших сервіси. Це може бути використано при масштабуванні, в залежності від більшого навантаження на читання або запис. Кожна із цих складових може бути розгорнута незалежно одна від одної і запущена у різній кількості екземплярів. Організація високої швидкості читання також можна досягти способом використання бази даних, яка налаштована під швидке зчитування.

Кожен контролер має компонент обробника команд або запитів, який через відповідний інтерфейс надсилає повідомлення у шину подій. Шина подій прослуховується компонентами, які реагують на появу нової подій із командою або запитом. Після обробки події, інформація вноситься або зчитується із бази

даних. Очевидне питання, яке спадає на думку, полягає у синхронізації двох баз даних – як записані дані потрапляють у базу даних для зчитування? Компоненти обробки команд за допомогою обміну повідомленнями синхронізують стан двох баз даних. Після збереження даних у одну базу, сервіс опублікує подію про створення об’єкту. У моделі для читання компонент, який відповідає за обробку подій, за допомогою анотації @EventHandler, побачить, що подія створення відбулася, і викличе логіку збереження даних із події у модель для читання. Це правило стосується всіх сервісів, які спостерігають за певною подією. Так реалізується принцип прозорості розташування («location transparency»), суть якого полягає у використанні ресурсів за назвою, без прив’язки до розташування об’єкту виклику. Сервіси не знають про місцезнаходження один одного і видають подію, яка не містить інформації, скільки сервісів повинно відреагувати на це повідомлення або подію, і яка логіка повинна викликатися далі.

Архітектура частини сервісу, призначеної для читання, виглядає так на рисунку 3:

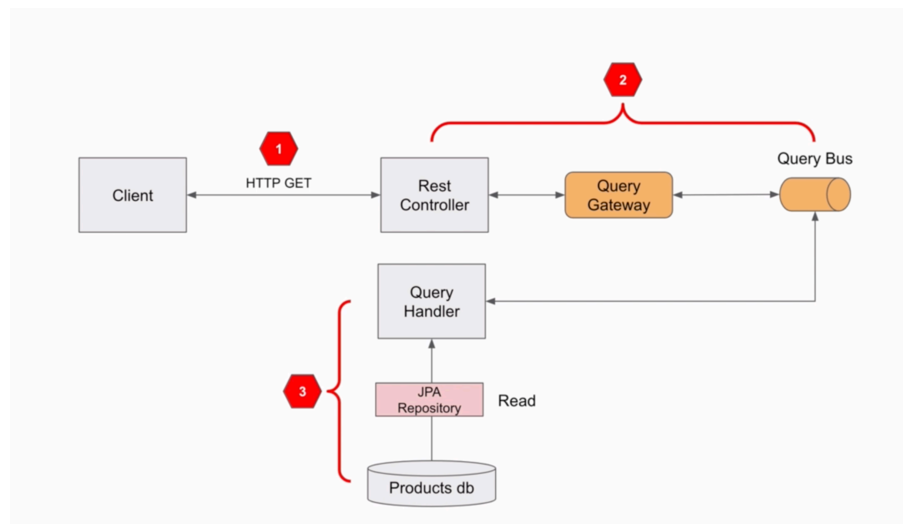

Рис.3

Рест-контролер для обробки запитів містить у собі об’єкт, через інтерфейс якого він спілкується із шиною подій. Є також компонент-обробник, який реагує на події. Він зв’язується із об’єктом репозиторію, а цей, зі свого боку, взаємодіє із базою даних. Також репозиторій відповідає за надання доступу до агрегації іншим сервісам. Зміна стану агрегації призводить до

генерації подій домену. Події предметної області і агрегування утворюють модель предметної області, яку ми більш детально у першому розділі.

3.3 Обробка помилок та відкат транзакцій

Із боку моделі команд, необхідно провести валідацію. У моделі для читання вона, зрозуміло, не потрібна. У побудованій моделі використано валідацію у трьох місцях –у Java Bean, MessageDispatchInterceptor та у EventsHandler. Кожен із цих об'єкті містить логіку перевірки інформації на коректність. Спершу Java Bean Validation перевіряє поля на допустимі значення, обмежуючи їх встановленою бізнес-логікою. Перед надсиланням команди у шину подій, валідацію необхідно провести для самої команди, переконавшись, що вона побудована правильно і необхідні поля заповнені. Опісля, валідація проводиться на боці обробника подій, оскільки для цього процесу потрібно перевірити достатню наявність інформації у події. Такий приклад – це реалізація принципу програмування «WET» (write everything twice), дані перевіряються декілька раз, для того, щоб переконатися кожен раз у тому, що вони складають собою правильну структуру, враховуючи те, що не завжди інформація приходить із контролера – інші сервіси теж беруть участь у цьому процесі рисунок 4.

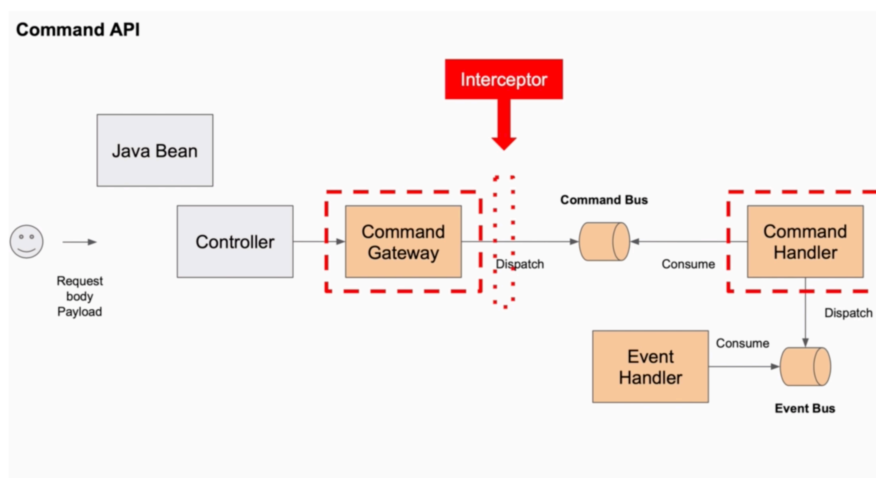

Рис. 4

При використанні шаблону CQRS, під час написання системи із мікросервісною архітектурою, постає логічне запитання - як перевірити, чи вже існують об'єкти у таблиці бази даних, перш ніж створювати нові?

Наприклад, якщо створюється новий продукт у системі, як перевірити наявність продукту із таким самим ідентифікатором і переконатися, що такий продукт ще не існує? Моделі для запитів і команд – розділені і саме на командній моделі лежить відповідальність за підтримку узгодженості даних. Розроблений API спочатку збереже подію у сховищі подій, і тільки після цього подія буде опублікована для синхронізації даних між сховищем подій та базою даних запитів. Зв'язок між моделлю команд та API запитів здійснюється за допомогою обміну повідомленнями про події, це не миттєвий процес. Синхронізація між двома моделями може зайняти деякий час, що у нашому випадку створило б суттєві затримки при зверненні до командної моделі. Детально ця проблема розглядається у [20]. Рішенням цієї проблеми, буде створення додаткової таблиці, яка буде містити ідентифікатори продуктів та назви. Враховуючи, що командна модель може працювати із даними різних баз, то нова таблиця ідентифікаторів, буде частиною командної моделі, і цей функціонал не буде доступний для API запитів. Він буде використовуватися командною моделлю тільки тоді, коли API команди має на меті створити продукт.

І спочатку він звернеться до таблиці пошуку, щоб дізнатись, чи вже є продукт із певним ідентифікатором та назвою, і якщо так, то функціонал сервісу створення завершиться із помилкою, де буде вказано, що такий продукт вже існує.

Таблиця пошуку не повинна містити точно таку саму інформацію про продукт, як і база даних. Ця таблиця має зберігати лише ті поля продукту, які потрібні для пошуку та перевірки, чи такий продукт існує.

Для того, щоб зробити запит у таблицю пошуку, перш ніж команда створення продукту дійде до шини подій, її необхідно перехопити і пересвідчитися, чи створення об'єкту безпечне. Додатковий обробник події створення, буде взаємодіяти із репозиторієм сутності пошуку та перевіряти її наявність у базі даних. На Рисунку 5 зображена модель продуктового сервісу, де таблиця перевірки наявності продуктів виділена червоним кольором.

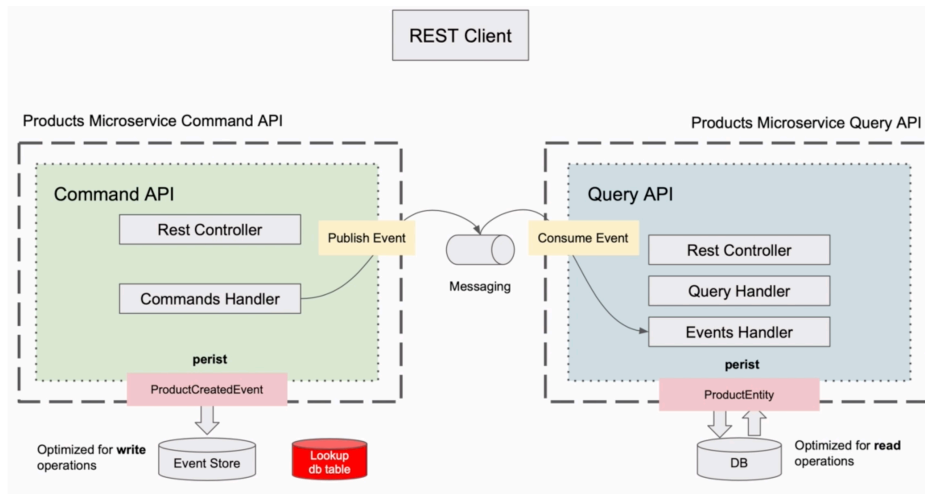

Рис.5

Обробка помилок і відкат транзакції є важливою складовою дизайну застосунку. Повідомлення про виняткові ситуації будуть оброблятися в одному центральному класі ServiceExceptionHandler. Цей клас допоможе обробити помилки, які можуть виникнути в ході виконання програми і за їх типом сформувати повідомлення від сервісу про те, яка саме помилка виникла. Це допоможе позбутися шаблонного коду для кожного із методів у класах із логікою обробки подій, за допомогою такого підходу не потрібно щоразу оточувати виклик методу блоками «try-catch». Також така організація необхідна для того, щоб не читати цілий список викликів методів всередині сервера, як це відбувається при відсутності належної обробки помилок, а зразу побачити повідомлення помилки у дружній для користувача формі. Такі класи-обробники зазвичай позначають анотацією @ControllerAdvice.

Однак тут існує підводний камінь. Клас, який відповідає за обробку помилок повинен мати перевизначений метод onError() інтерфейсу ListenerInvocationErrorHandler, бути зареєстрованим конфігураційним процесором для певної групи та, відповідно, передавати помилку у контролер. Якщо цей клас не зареєструвати конфігураційним процесором, він не зможе поширити помилку так, щоб транзакція розпочала відкат. Для того, щоб передати повідомлення помилки у контролер, потрібно, щоб процесор відстеження розповсюдив помилку, транзакція перейшла у стан відкату, а саму помилку можна було зловити у іншому місці. Після виявлення помилки

процесор відстеження передає повідомлення у клас контролера, який повертає його для клієнтської програми. Після поширення помилки процесором відстеження, клас, який контролює сагу, теж одразу про це дізнається, і одразу віддає команду про запущення компенсаційної транзакції. Кожен клас, анотований за допомогою @Saga,
проінформує Axon про те, що це клас, який керує транзакцією, він має початковий і кінцеві методи, де виконання останніх інформує про результат виконання саги. Анотація @StartSaga вказує на початок життєвого циклу саги та запускає перший метод. Він створює команду та надсилає її у шину подій. Організація класу саги здійснюється у такий спосіб, що обробка подій методами повинна здійснюватися через @SagaEventHandler, де необхідно вказати ідентифікатор асоціації – поле, яке буде ідентифікатором для події, на яку повинен відреагувати цей метод. Подія обробляється, це спричинить створення нової команди, яка відправиться у шину подій. Декілька методів класу саги можуть мати анотацію @EndSaga, яка сигналізуватиме про те, що сага закінчується і більше не може бути використана. Кожен анотований метод обробляє подію за її типом. Для кожної події може бути викликано лише один анотований метод у класі саги. Клас може обробляти одночасно декілька подій, якщо контролює зразу дві або більше транзакції. Axon Framework буде визначати який саме екземпляр використати для обробки різних подій. У разі виникнення помилки у одній із подій, транзакція буде автоматично переведена у стан відкату і компенсаційні команди будуть надіслані до шини подій.

Висновок

У рамках цієї роботи був проведений аналіз різних методів організації розподілених транзакцій у системах із мікросервісною архітектурою, розглянуті методи зберігання даних у реляційних та NoSQL базах даних для збереження історичності модифікації даних та роботи із окремими моделями для читання та запису, описані механізми семантичного блокування та підходи до моделювання оркестраційних об'єктів керування транзакцією. В результаті були запропоновані підходи та принципи до побудови архітектури та способи вирішення проблеми паралельного доступу з використанням транзакцій. За допомогою паттернів розподілених транзакцій та паттернів мікросервісної архітектури було реалізовано систему із організацією оркестраційних саг для проведення транзакції між кількома сервісами. За допомогою функціоналу Axon Framework було виокремлено модель для читання та запису. Вибір необхідного підходу саги – оркестраційний чи хореографічний це проблема, яку розробникам застосунків необхідно розглядати насамперед. Якщо застосунок невеликий, тоді є сенс скористуватися хореографічним методом. Він краще підходить для нескладної логіки та його доставка у бізнес-середовище буде швидшою, ніж у окрестраційного, за відсутності експертизи у команді розробки [22]. Ще більше деталей можна знайти в основоположника реалізації саг у мові Java, автора фреймворку «Eventuate», Кріса Річардсона у [7] та [23].

Визначити перелік незворотних операцій саги для переміщення їх у кінцеву фазу. Користуватися правилами єдиної мови із доменно-орієнтованого дизайну для виокремлення таких операцій для системи, що розроблюється.

Якщо було обрано оркестраційний підхід, то важливо звернути увагу на такі моменти:

- Спосіб комунікації між сервісами – REST-клієнти чи шини повідомлень. Якщо рест-клієнти, то такі у мові Java можна скористатися FeignClient або WebClient, як альтернатива старому RestTemplate. Якщо шини повідомлень, то у фреймворку Axon є можливість організувати комунікацію через вбудовані командну, подійну та шину для запитів. Також доступні анотації, після завантаження залежності «axon-

framework», наприклад, @Saga – дозволяє визначити клас, як оркестраційний об’єкт для проведення транзакції. @Aggregate – спосіб зробити складену сутність із кількох.

- Змоделювати оркестраційну сагу як скінченний автомат. Множину станів саги Q визначити як перелік всіх сервісів, які беруть участь у транзакції. Початковий стан q0 визначити як оркестраційний об’єкт, який контролює проходження саги. Вхідний алфавіт E визначити як скінченний набір команд. Функцію переходу між станами (сигма) визначити як об’єкт події, який містить всю необхідну інформацію про те, що відбулося у системі. Множину кінцевих станів F визначити як набір методів оркестраційного класу, які відповідають за результат проходження саги.
- Під час розподілу відповідальності за виконання логіки у рамках певного сервісу користуватися принципом «розділяй і пануй». Проектувати декілька викликів у один сервіс, якщо це необхідно, це зменшить зв’язність коду. Оркестраційний підхід часто передбачає більшу зв’язність коду, ніж хореографічний, тому розподіл логіки і менші за об’ємом виконаних операцій команди кращі, ніж великі [25].
- Якщо використовуються семантичні блокування, визначити поля та статуси, за допомогою яких буде реалізовуватися блокування для досягнення ізоляції. Визначити послідовність переходів між статусами за допомогою списку подій, які можуть відбутися під час проходження транзакції. На рисунку [] зображено приклад використання семантичних блокувань для статусу замовлення. Кріс Річардсон у [4] рекомендує моделювати семантичні блокування як скінченні автомати.

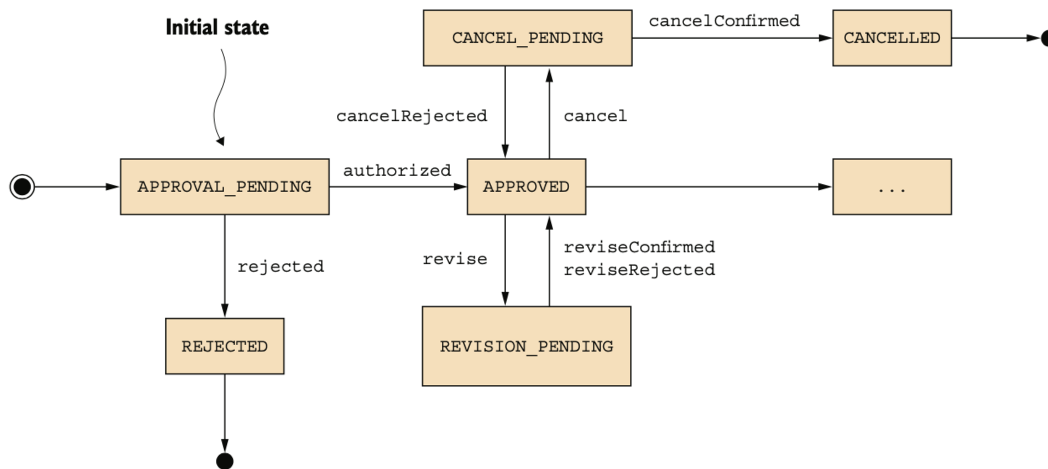

Рисунок 6

- Для реалізації комутативних оновлень, сформувати список компенсаційних транзакцій. Визначити відповідність між подіями на компенсаційними транзакціями. Логіку реалізувати у класі-оркестраторі.
- Користуючись єдиною мовою із доменно-орієнтованого дизайну, створити дві моделі – для читання та запису. Розробити архітектуру кожної із моделей, підібрати read-only та write-only бази-даних. Реалізувати механізми обрання лідера та шардінгу, якщо необхідно.

Цих пунктів достатньо для виокремлення найважливіших деталей реалізації саг. Завдяки підходу із використанням доменно-орієнтованого дизайну, легко описати процес проходження транзакції на зрозумілій для людей мові. За наявності кількох сервісів, виокремлюється один, як найбільш логічна сутність для організації класу-менеджера, що керуватиме процесом проходження транзакції. Зрозумілість кожного кроку транзакції, спроектованого на моделі скінченного автомату, дає можливість швидко реалізувати бізнес-логіку та надати замовнику результати. Axon – легковаговий інструмент для реалізації транзакцій у системах із мікросервісною архітектурою. Він допомагає створювати добре масштабовані додатки та реалізувати архітектурний шаблон розподілу системи для читання та запису CQRS). Фрейморк дозволяє створювати складні сутності, наприклад, агрегацію, та надає можливість керувати ними за допомогою шини подій. Легке конфігурування фреймворку досягається через підтримку анотацій, які дозволяють створювати агрегати, прослуховувачі та обробники подій без

прив'язки коду до логіки, специфічної для Axon. Призначення цих об’єктів визначається одним рядком коду, а відповідальність за реалізацію логіки проксі-класів лягає вже на Spring Framework за допомогою CJLIB та Dynamic Proxy. Це допомагає зосередитися на бізнес-логіці та спрощує unit-тестування коду. Варто зауважити, що не кожен додаток отримає користь від використання Axon. Прості додатки без складної бізнес-логіки, які не плануються для розширення, не отримають вигоди від CQRS та Axon. Їх важливо використовувати, коли застосунки будуть розширюватися за рахунок додачі нового функціоналу протягом тривалого часу. Наприклад, інтернет-магазин може почати з системи виконання модуля замовлення. На більш пізньому етапі це можна доповнити інформацією про запаси, щоб гарантувати, що інвентар оновлюється при продажу. Потім клієнт може попросити реалізувати реєстрацію фінансової статистики продажів, графіки та діаграми, тому застосунок повинен мати високу масштабованість. Якщо це інтернет-магазин, то ймовірно, що коефіцієнт читання буде вищим, ніж коефіцієнт запису. У такому випадку модель, буде зручно налаштувати під читання і оптимізувати для швидких запитів.

Історичність створення даних надає переваги, наприклад, деякі програми щомісяця відправляють електронні листи, щоб повідомити користувачів про зміни, зв’язані з ними. Це позбавляє необхідності створювати складні запити до бази даних, щоб побачити різницю станів. Інший приклад – статистика та інструменти звітності. Вони збирають інформацію у звіти і показують зміну даних у часі. Axon можна сконфігурувати для оновлення даних в режимі реального часу або за розкладом [26]. Axon має хорошу інтеграцію з іншими додатками, наприклад, Spring Boot, Node.js, Django та із іншими. Використання команд, подій і запитів може полегшити інтеграцію із зовнішніми додатками, зокрема через можливість прослуховувати події, які генеруються застосунком.

Посилання:

1. https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/
2. https://medium.com/swlh/handling-transactions-in-the-microservice-world-c77b275813e0
3. https://martinfowler.com/articles/microservice-trade-offs.html#consistency
4. Book – Chris Richardson, Microservices
5. https://habr.com/ru/company/flant/blog/523510/
6. https://habr.com/ru/company/otus/blog/501294/
7. https://proselyte.net/tutorials/sql/sql-transactions/
8. https://www.youtube.com/watch?v=cpdL73GsM5c&t=7s Chris Richardson - Managing data consistency in a microservice architecture using Sagas
9. Лекції Миколи Миколайовича Глибовця із АСД
10. https://docs.microsoft.com/ru-ru/sql/t-sql/statements/create-materialized-view-as-select-transact-sql?view=azure-sqldw-latest
11. https://streletzcoder.ru/materializovannyie-predstavleniya-v-sql-server/
12. https://www.youtube.com/watch?v=AKGT7wkVd34 Ануар Нурмаканов - Event Sourcing і CQRS на конкретному прикладі
13. https://www.garb.ru/blog/transaction.html
14. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt
15. Book – Мартін Клепман, Desingning Data-Intensive Applications
16. Лекції Гулаєвої Наталії Михайлівни із «Нереляційних баз даних», приклад про літак
17. Michael Pogrebinsky - Distributed Systems & Cloud Computing with Java
18. https://habr.com/ru/company/dododev/blog/489352/
19. https://habr.com/ru/post/232881/
20. Book – Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software
21. Book – Robert Martins, Clean Code

22. https://medium.com/@kirill.sereda/spring-cloud-netflix-eureka-по-русски-5b7829481717
23. https://axoniq.io/blog-overview/set-based-validation
24. https://russianblogs.com/article/1970800907/
25. https://robertleggett.blog/2019/03/17/when-should-you-choose-choreography-vs-orchestration-for-your-microservices-architecture/
26. https://www.youtube.com/watch?v=YPbGW3Fnmbc