КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота
на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

АНАЛІЗ ТА РЕАЛІЗАЦІЯ МЕТОДІВ ОПТИМІЗАЦІЇ РОБОТИ МІКРОСЕРВІСІВ

Виконала студентка 4 курсу
Софія ЯРМОЛЕНКО _____
(підпис)

Науковий керівник:
асистент, кандидат технічних наук
Олексій ФЕДОРУС _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
«____» _____
2023 р., протокол № ___
Завідувач кафедри
Василь ТЕРЕЩЕНКО _____
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 46 сторінок, 34 ілюстрації, 13 джерел посилань.

Об’єктом роботи є процес полегшення розробки, рефакторингу та планування мікросервісів. Предметом роботи є програмний засіб для полегшення інтеграції архітектурного рішення CQRS та результати аналізу методів оптимізації роботи мікросервісів, що допомагає при плануванні архітектури.

Метою роботи є створення фреймворку для веб сервісів, що полегшить розробку та інтеграцію архітектури CQRS. Додаткова мета - кількісний аналіз методів покращення часу виконання різноманітних запитів у існуючих сервісах зі своєю архітектурою.

Інструменти розробки: мова програмування Java, фреймворк Spring, фреймворк Hibernate, хмарне середовище Confluent Cloud, кластер Kafka, бібліотека JavaPoet, кластер ksqlDb, інструмент Jmeter, оркестратор контейнерів Kubernetes, платформа для контейнеризації Docker та кластер Minikube.

Результати роботи: виконано огляд основних архітектурних підходів типу CRUD, які використовуються для створення веб сервісів, проаналізовано недоліки кожного з них та альтернативний принцип CQRS. Запропоновано власне рішення для полегшення інтеграції цього архітектурного патерну. Також проаналізовано час виконання запитів та недоліки при використанні різних методів оптимізації.

Програмний продукт може бути корисним у всіх сферах, де використовуються веб сервіси, написані на java. Особливо це актуально в програмах, де потрібно робити багато аналітики даних.

ЗМІСТ

ВСТУП

Актуальність роботи та підстави для її виконання. На сьогоднішній день існує велика кількість веб-сервісів, якими користуються щоденно мільйони людей. Тому дуже важливо зробити досвід використання продукту приємним, адже конкуренція висока. Одне з найчастіших очікувань користувача – це те, що все буде працювати максимально швидко та якісно. Для цього потрібно завчасно проектувати систему, задумуючись про її ефективність, або ж правильно рефакторити існуючу. В роботі досліджено декілька варіантів оптимізації, включаючи архітектурний підхід з самого початку та можливості рефакторити існуючі рішення.

Оцінка сучасного стану об’єкта розробки. В роботі проводиться аналіз плюсів та мінусів сучасних методів оптимізації мікросервісів, які можна використовувати при розробці високонавантажених програм та систем. Також перший розділ приурочений створенню власного рішення для полегшення імплементації сервісів за принципом сучасного підходу CQRS. При цьому використано найновіші технології для збереження даних та роботи з ними. Тому об’єкт розробки можна вважати сучасним та легко інтегрованим в існуючі методи імплементації веб сервісів.

Мета й завдання роботи. Метою роботи є створення фреймворку для сервісів, написаних на java та відповідному сучасному стеку технологій, що полегшить розробку та інтеграцію архітектури CQRS. Так як це потребує масштабних змін, то додаткова мета - проведення аналізу методів покращення часу виконання різноманітних запитів у існуючих сервісах зі своєю архітектурою. Для досягнення цього поставлені наступні завдання:

- Знайти оптимальну хмарну інфраструктуру на основі якої буде інтегруватися CQRS

- Створити узагальнену архітектуру реалізації на основі обраної інфраструктури
- Розробити фреймворк для реалізації принципу CQRS, що можна легко інтегрувати в клієнтські сервіси
- Дослідити існуючі рішення для реалізації CQRS
- Протестувати рішення на прикладі домену онлайн магазину
- Проаналізувати ефективність різних методів оптимізації мікросервісів та сфери їх використання

Об'єкт, методи й засоби розробки. Об’єктом розробки є власний фреймворк, який можна легко інтегрувати в різні домени, що збереже час на впровадження архітектури за принципом CQRS. Для цього використано сучасний стек технологій. А саме: JavaPoet для автоматичної генерації коду, Spring Framework для швидкої інтеграції, клієнти для Kafka та ksqlDb щоб оптимально зберігати події, пов’язані з цим принципом.

Щоб зробити аналіз різноманітних методів рефакторингу, проводиться тестування з навантаженням. Для імітації цього обрано Jmeter. Сервіси для проведення аналізу написані на мові java, з використанням таких фреймворків як Spring та Hibernate. Також використано Docker як найпопулярнішу платформу для контейнеризації. Для оркестрації контейнерів створено локальний кластер Minikube (спрощена версія кластера для роботи з Kubernetes), але аналогічні методи можна використовувати і в хмарному середовищі.

Можливі сфери застосування. Розробка рішення для інтеграції CQRS може бути використана у багатьох сферах, яким потрібні веб сервіси. Особливо це буде корисно, якщо сутності домену не живуть довго, але над ними потрібно проводити часто аналіз. Яскраві приклади - це онлайн банкінг, магазин товарів, системи для керування персоналом, тощо.

Аналіз інших методів корисний для рефакторингу та планування архітектури веб сервісів у майбутньому.

РОЗДІЛ 1. АНАЛІЗ ТА СТВОРЕННЯ ОПТИМАЛЬНОЇ АРХІТЕКТУРИ ДЛЯ ЗБЕРЕЖЕННЯ ДАНИХ

1.1 Огляд архітектур, які працюють за принципом збереження поточного стану

Мікросервіси самі по собі вже вважаються гнучким рішенням, якщо порівнювати з монолітними система. Але це дуже абстрактне поняття, яке можна реалізувати абсолютно різними методами. Більшість компаній використовують патерни CRUD, де варіюється кількість реляційних баз даних та можливість доступу до них. Спільна риса для цього - це зберігання даних у форматі поточного стану та строге дотримання відокремлення сутностей в таблицях (нормалізовані бази даних).

Недолік такого підходу очевидний - велике навантаження на загальну базу даних, коли сервісів стає дуже багато. Адже всі можуть одночасно робити багато запитів на читання та оновлення, що створюватиме колосальне навантаження. Особливо це помітно коли виконуються sql query, що потребують операцій join.

Як альтернативне покращення - це розділення даних на декілька CRUD баз, до яких мають доступ не всі сервіси. Таке розділення робиться основуючись на домені. Але воно також не ідеальне, адже вимагає багато ресурсів та все ще блокує на певний час базу, коли виконуються запити на читання, що потребують багато операцій join. Особливо це стає помітно, коли систему використовують дуже багато користувачів по всьому світу. Також як недолік створюються додаткові запити між сервісами щоб дістати дані. [3] Схематично такий підхід можна зобразити як на рисунку 1:

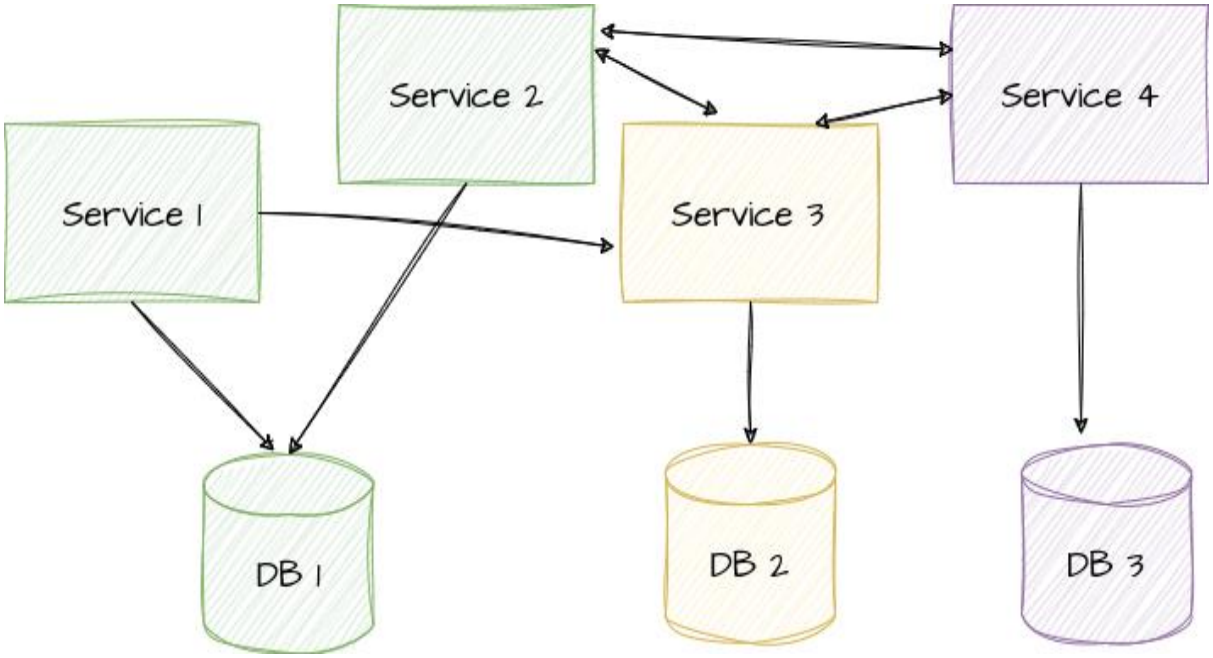

Рисунок 1: Архітектура з розділенням даних по різних базах

Наступний варіант оптимізації - Multitenancy. Це спосіб колективної оренди, коли клієнти (наприклад з різних компаній чи регіонів) використовують сервіс на ізольованих від інших окремих базах даних. З одного боку це набагато безпечніше, а з іншого також покращує швидкість роботи з базою, адже даних набагато менше. [7] Але це все одно не рятує нас від того, що ми читаємо та записуємо дані в спільну базу. Операцій на запис набагато менше відносно читання, що блокує запис на більший час ніж він міг бути. Також однаковий формат даних, але не оптимальний для читання, може сповільнювати його. Таку архітектуру схематично можна зобразити як на рисунку 2:

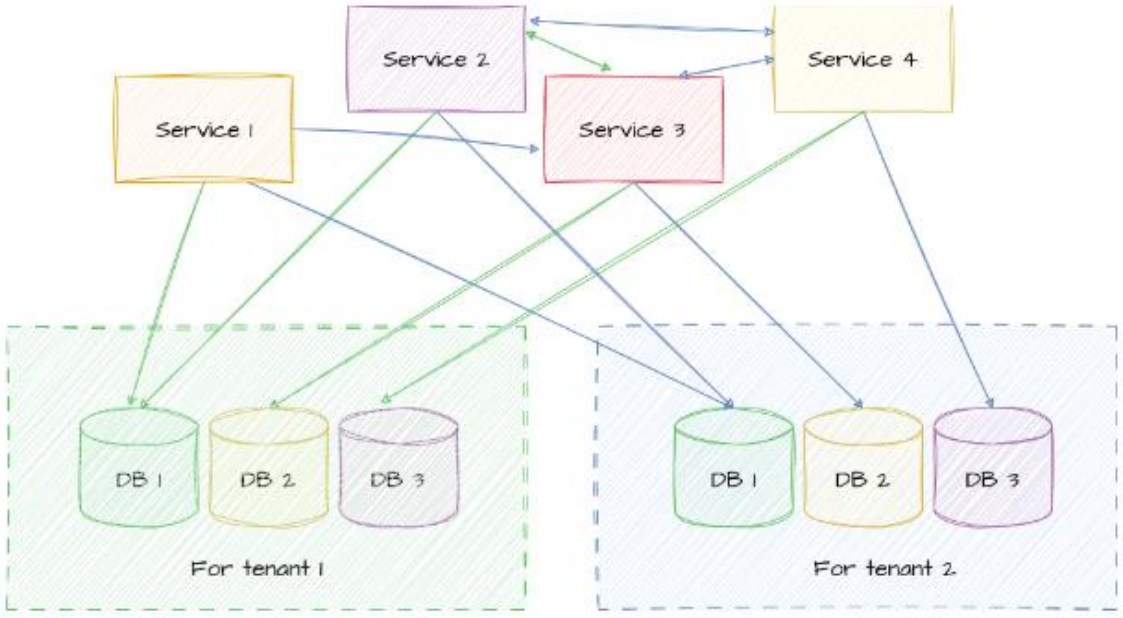

Рисунок 2: Архітектура з використанням Multitenancy

Отже, можемо бачити, що такі підходи мають свої недоліки. Вони недостатньо гнучкі. Особливо це помітно в програмах, в яких не вимагається довге життя сутностей з великою кількістю подій, які змінюють їх стан, але при цьому потрібно робити багато аналізу та часто створювати різноманітні запити на отримання даних в різних формах. В наступному розділі розглядається гнучкий альтернативний метод побудови мікросервісів. Також описуються найяскравіші плюси та мінуси такого рішення.

1.2 CQRS в поєднанні з Event Sourcing як альтернативний архітектурний підхід

1.2.1 Теоретичні відомості про CQRS

“Розподіл відповідальності за команди та запити” (Command and Query Responsibility Segregation, далі в роботі позначається як CQRS), перш за все походить від принципу Бертрана Мейера “Розділення команд та запитів” (Command and Query Separation, надалі як CQS), який він увів в

1985, працюючи над мовою програмування Eiffel. Головна ідея CQS - це те, що кожен метод має бути або командою, яка виконує дію, або запитом, який повертає дані, але не обома одночасно. Перефразовуючи, постановка питання не повинна змінювати відповідь.

На початку, CQRS вважався лише розширенням цієї концепції, тобто як CQS на більш високому архітектурному рівні, але зрештою ці поняття розділили.

В CQRS використовується те саме визначення команд і запитів, які сформулював Мейєр, і дотримується точка зору, що вони повинні бути строго визначені та окремі. Принципова відмінність полягає в тому, що в CQRS системи розділені на дві повністю незалежні моделі, які існують в різних контекстах. Одна з них відповідає за виконання команд для зміни стану усієї системи, а інша обробляє запити на отримання даних про цю систему.[11] Абстрагуючись від реалізації це виглядає приблизно як на рисунку 3:

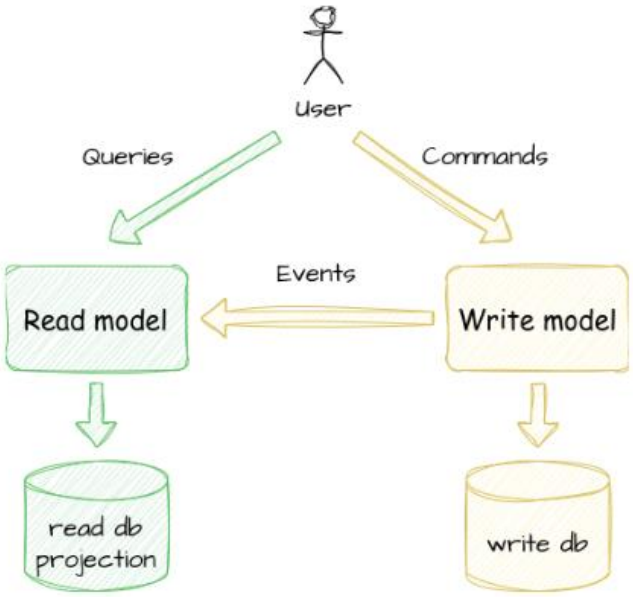

Рисунок 3: Схематичне зображення CQRS

Розділення обов’язків дозволяє використовувати для реалізації CQRS окремі бази такі як “read db projection” та “write db”, які повністю не залежать одна від одної. Це великий плюс, адже дані в них можуть бути представлені в будь-якому форматі, при чому різному. Наприклад, “write db” може бути якоюсь не нормалізованою NoSQL базою, а “read db projection” навпаки реляційною SQL. Це вже залежатиме від постановки задачі, яку потрібно вирішувати. Для прикладу, якщо нам потрібно часто виконувати запити на читання, які б у реляційному представленні даних потребували б багато операцій join, то ми можемо використати NoSQL базу для створення проекції, в якій дані будуть денормалізовано зберігатись у форматі документів. Це в рази пришвидшить виконання запитів на читання.

Інший значний плюс такого архітектурного підходу - це можливість гнучкого масштабування. За рахунок того, що всі команди у нас постійно зберігатимуться в “write db”, то ми можемо на основі цього створювати скільки завгодно проекцій, які вирішуватимуть різні проблеми читання та додатково розподілятимуть query моделі, що розділить навантаження між усіма. Недолік - додаткове використання ресурсів, що не безкоштовно. Як приклад без деталей реалізації можна розглянути рисунок 4:

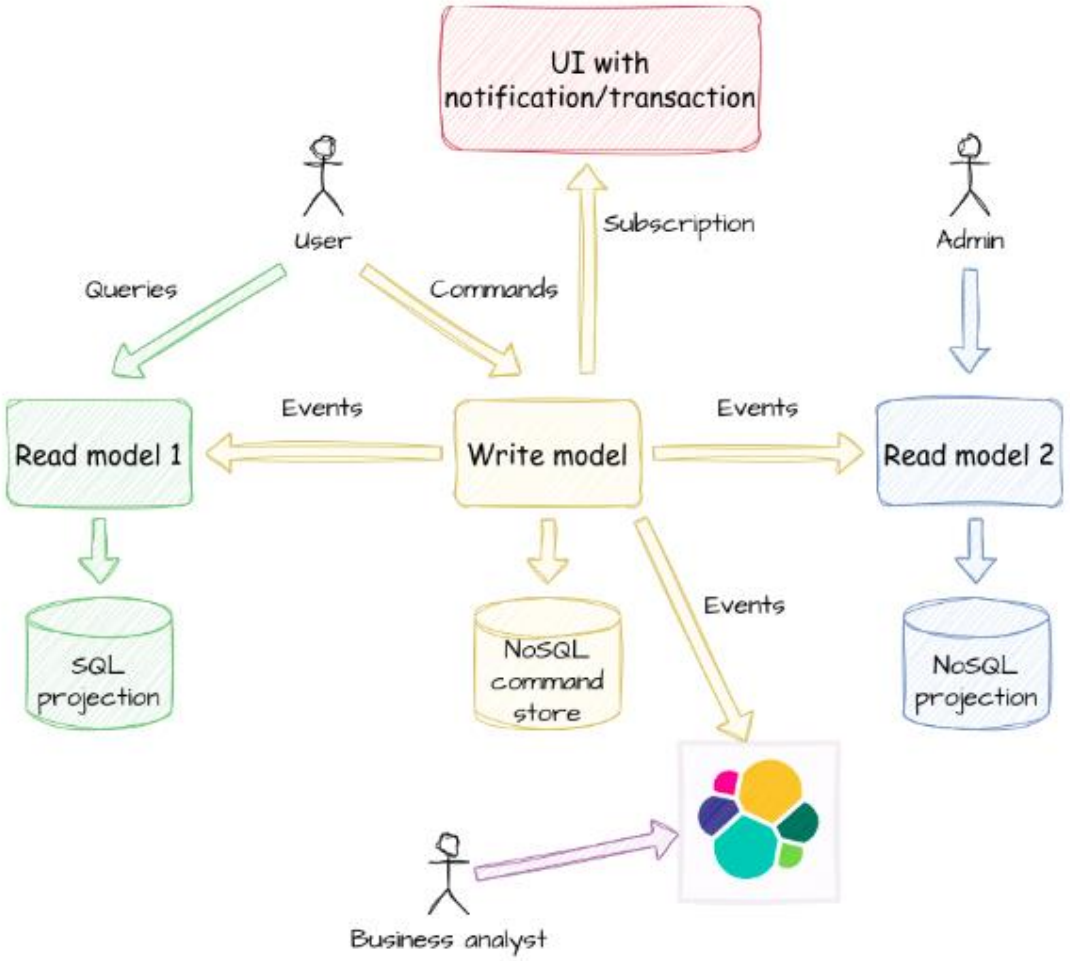

Рисунок 4: Збереження даних в проекціях різного типу

Як можна бачити, на діаграмі присутній користувач User, який надсилає команди, наприклад це може бути “Взяти в оренду велосипед” та “Повернути велосипед”. Потім “Write model” обробляє ці команди, зберігає їх в своєму NoSQL сховищі та відправляє повідомлення про те, що “велосипед в оренді” чи “велосипед вільний”. Далі маємо чотири зовсім різноманітні “Read model” та їх незалежні проекції, в яких дані зберігаються у різних форматах в залежності від найбільш оптимального вигляду для виконання запитів на читання різними користувачами для різних потреб. Також як плюс можна визначити те, що в будь-який момент легко додається нова “Read model”, для якої можливо по-своєму створити проекцію, синхронізуючись з “command store”.

Як одним з плюсів також можна вважати те, що саме існування понять command та query більш підходить для патерну проектування систем на основі предметної області (domain driven design, надалі як DDD), адже вони розширюють список формулювання запитів, не обмежуючись операціями CRUD (create, read, update, delete) над сутністю. Якщо ж розглядати швидкість виконання, то за рахунок різних моделей для читання та запису, буде менше операцій над базою та більш важкі по часу запити не блокуватимуть запис.

1.2.2 Головний недолік CQRS та теорема CAP (теорема Брюера)

Системи, створені за принципом CQRS вважаються розподіленими, адже у нас існують паралельно дві моделі для запису та читання, які потрібно синхронізувати між собою, що займає час. З теперішніми технологіями це триває секунди, але в production середовищі навіть в цей період може виникнути невідповідність між query та write model. Теорема CAP (Consistency, Availability and Partition Tolerance) яскраво пояснює, що це задовільний сценарій, адже в розподілених системах нам потрібно чимось жертвувати.

Формулювання самої теореми:

Теорема CAP (відома також як теорема Брюера) — евристичне твердження про те, що в будь-якій реалізації розподілених систем можливо забезпечити не більше двох із трьох таких властивостей:

- **узгодженість даних (Consistency)** - у всіх обчислювальних вузлах в один момент часу дані не суперечать одна одній
- **доступність (Availability)** - будь-який запит до розподіленої системи завершується коректним відгуком, проте без гарантії, що відповіді всіх вузлів системи збігаються

- **стійкість до поділу (Partition tolerance)** — розділення розподіленої системи на кілька ізольованих секцій не призводить до некоректності роботи кожної з секцій.

Ілюстрацію цієї теореми можна зобразити таким чином, як на рисунку 5:

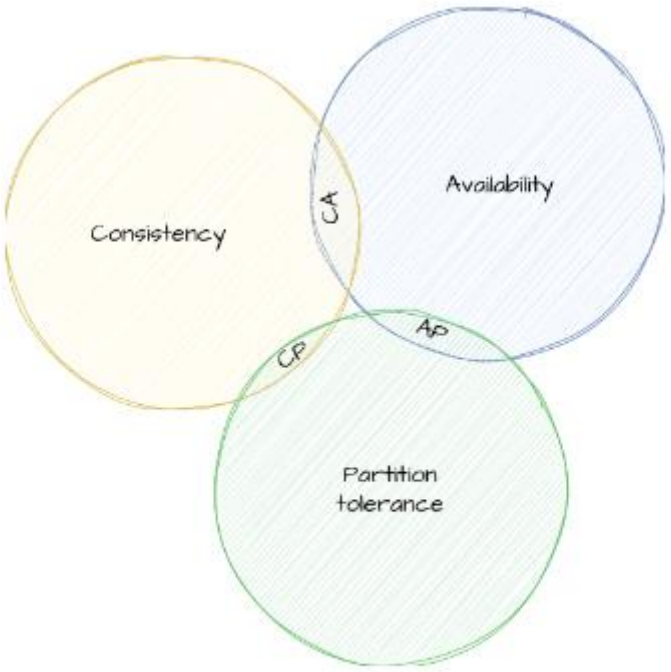

Рисунок 5: графічне зображення теореми CAP

Теорема PACELC — розширення теореми CAP, яке свідчить, що у разі поділу мережі (P від partition) у розподіленій комп'ютерній системі необхідно вибирати між доступністю (A від availability) та узгодженістю (C від consistency) (відповідно до теореми CAP), але в будь-якому випадку, навіть якщо система працює нормально без поділу (E), потрібно вибирати між затримками (L від latency) і узгодженістю (C від consistency).

Отже, якщо в CQRS ми маємо стійкість до поділу та доступність, то ми не можемо мати завжди узгодженість даних[1].

1.2.3 Теоретичні відомості про Event Sourcing

Головна концепція Event Sourcing - це використання баз даних для збереження повної серії подій, які відбулись з даними домена, замість того,

щоб зберігати лише поточний стан у домені. Ідея полягає в тому, що з повної інформації про події над даними можна завжди відтворити їх поточний стан, але не навпаки. Це може спростити завдання в складних доменах, уникаючи необхідності синхронізувати модель даних і бізнес-домен, одночасно покращуючи продуктивність, масштабованість і швидкість реагування.

Завдяки такому історичному підходу підтримуються повні журнали аудиту, за якими завжди можна як завгодно і коли завгодно відтворити стан системи. Також це дозволяє в будь-який момент створити нову логіку для аналізу поведінки користувача, адже фактично зберігаються всі його дії, а не тільки їх результат. [11]

Головний недолік - це тривалий час для відтворення поточного стану, якщо такий потрібен, адже він не зберігається.

1.2.4 Теоретичні відомості про поєднання CQRS та Event Sourcing

Поєднанням CQRS і Event Sourcing компенсуються їх недоліки. Команди з CQRS можна зберігати у форматі івентів та потім синхронізовувати всі проекції за допомогою Event Sourcing, який виступає механізмом зберігання даних для домену. Проблема знаходження поточного стану в свою чергу вирішується за допомогою моделі для читання.

Найскладніша частина використання CQRS + Event Sourcing підходу - це реалізація синхронізації даних між моделями та підтримки оптимального сховища моделі для письма. Потрібне сховище, яке гарантуватиме послідовність збереження подій, гарантію їх доставки до всіх проекцій та довготривале збереження, яке дозволятиме ефективно підключати нові проекції та відтворювати за допомогою Event Sourcing стан домену від самого початку. [10]

Ще один недолік - це те, що потрібно витрачати час на одні і ті ж схожі дії для реалізації та підтримки інфраструктури. Однією з найголовніших цілей роботи та декількох наступних розділів є саме реалізація загального рішення, яке можна використовувати для побудови різноманітних систем із своїм власним доменом за принципом CQRS.

1.2.5 Створення власного фреймворку та обґрунтування вибору інструментів для цього

Перша мета реалізації - це serverless архітектура. Тобто потрібно було реалізувати фреймворк, який допомагатиме з найменшою кількістю зусиль на підтримку архітектури для Event Sourcing використовувати провайдер хмарних рішень.

На сьогоднішній день проблема обміну даних в мікро-сервісах не нова. Якщо брати за основу тільки види синхронізації у вигляді запит-відповідь, то узгодженість даних сильно б порушувалась на більший термін. Тому популярне загальноприйняте рішення для цього - це Event Streaming, або обмін повідомленнями. Головна ідея - це те, що мікросервіс може слати повідомлення, якщо в його домені щось змінилося і цим самим синхронізуватися з іншими. З усіх івентів і формується безперервний потік (Event Stream). На даний момент найпопулярніші брокери повідомлень або платформи для Event Streaming - це RabbitMQ та Kafka. Розглядаються саме найпопулярніше за рахунок того, що багато сервісів вже їх використовують, тому інтеграція буде легшою. Обрано саме Kafka через наступні причини. Інструмент гарантує доставку повідомлень та їх порядок. Вона активно підтримується Confluent, а отже постійно розвивається та на основі стрімінгу івентів створюються додаткові сервіси, що дає багато плюсів, адже це нові можливості для автоматизації. Плюсом Kafka також можна вважати її гнучку масштабованість при великому навантаженні. Також для

створення тем, конс'юмерів та продюсерів з івентами в Kafka є зручний java client. [13]

Для Event Streaming Kafka підходить, але вона не задовольняє критерії для ефективного збереження подій та Event Sourcing. Реалізація Event Sourcing вимагає оптимального дизайну тем, які служать сховищем для додавання та читання подій, об'єднаних спільним агрегатом. Можна почати з базового дизайну, маючи одну тему, яка буде зберігати події, що відносяться до однієї сутності в домені (агрегатом всіх подій буде ідентифікатор цієї сутності). Це гарантуватиме строгий порядок додавання подій, а читання стану може бути таким же простим, як читання теми зі зсувом один починаючи з самого початку. Але стає цілком очевидним, що це може призвести до проблеми, коли ми матимемо величезну кількість агрегатів, що призведе до суттєвої кількості тем. Це може спричинити погіршення перфомансу сервісу. Іншим рішенням може бути спільна розділена тема для всіх типів агрегатів, але через це зчитування подій відбувається надзвичайно повільно, оскільки потрібно сканувати весь набір даних і фільтрувати потрібний агрегат.

Але якщо об'єднати ці рішення і додати базу даних, щоб забезпечити більш ефективне зберігання для івентів та команд, можна зіткнутися з проблемою розподілених транзакцій, а нам потрібно гарантувати відсутність стану гонки. Вирішити цю проблему можна використовуючи інший потужний інструмент платформи Confluent Cloud – ksqlDb. Вона забезпечує концепцію сховищ стану, яка може зберігати потік подій, що відносяться до певного агрегату, як знімки(snapshots). Це вирішує проблему розподілених сховищ, а також операцій читання для побудови агрегатного стану, оскільки зберігається лише поточний знімок стану агрегату. Інструмент зручний у використанні, адже для нього також є java клієнт. ksqlDb Table - це розподілена колекція, моделі якої змінюються з часом і відображають теперішній стан агрегату. Поточний стан таблиці

зберігається локально та тимчасово на певному сервері за допомогою RocksDB. Послідовність змін, застосованих до таблиці, може безкінечно зберігатися в темі.[12] Для оптимального безкінечного зберігання величезних обсягів даних у Kafka можна використати інший інструмент - Tired Storage.[9]

Ще один з плюсів ksqlDb - це можливість легкого створення конекторів для синхронізації даних про події з іншими базами даних та сервісами. В контексті CQRS - це можливість автоматичного створення проекцій напряму з івентів чи навпаки поступово мігрувати CRUD систему в CQRS, синхронізуючі події в існуючій базі даних та Event Storage . Наприклад, ми можемо інтегрувати конектор для MySql бази даних та зробити її джерелом правди.[12] Запит для створення такого конектора можна побачити на рисунку 6:

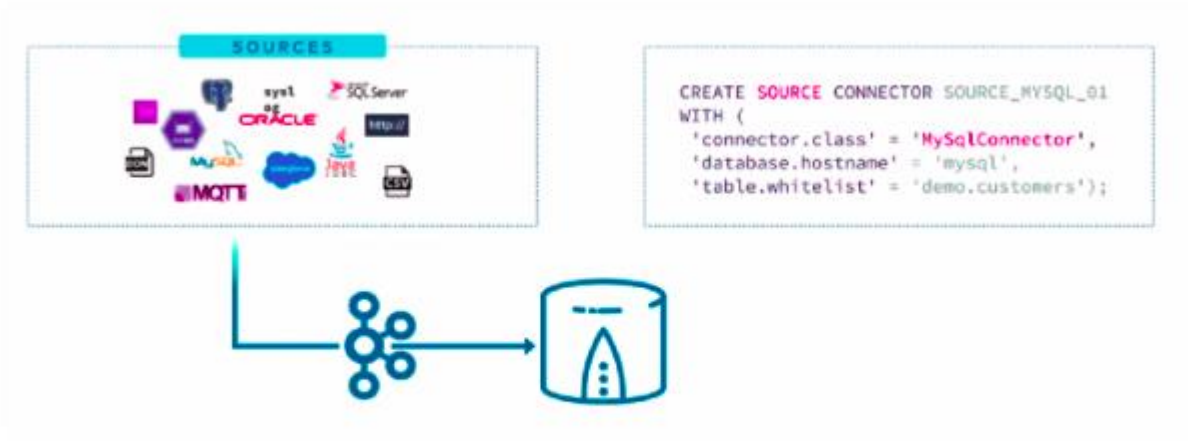

Рисунок 6: Приклад запиту для створення конектору з джерелом

Також можемо навпаки з ksqlDb стрімити дані, наприклад, в elasticSearch щоб в майбутньому робити аналіз цих даних чи певні звіти.

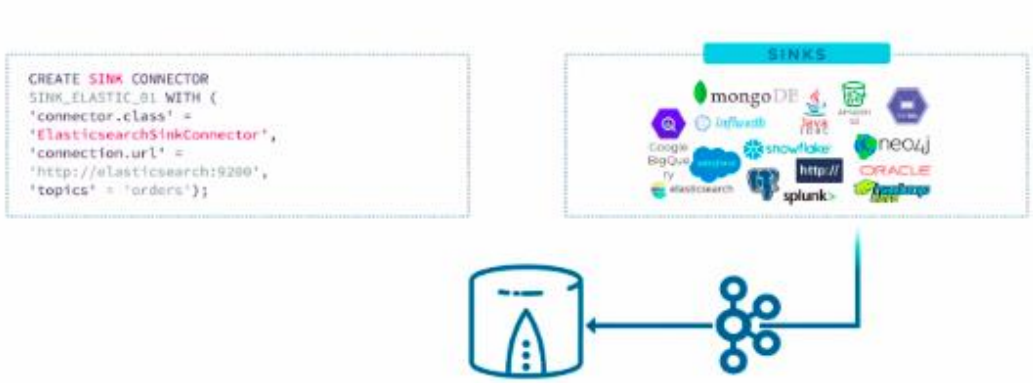

Рисунок 7: Приклад для запиту для створення вихідного конектора

Так як перша версія фреймворку зроблена тільки для сервісів, написаних на java, то розробка інтегрована з Spring Framework, адже це найпопулярніший фреймворк для реалізації веб програм на даний момент через його зручність у використанні та великий набір функцій та адаптерів.

Щоб зробити максимально клієнто-орієнтоване використання, де не потрібно буде писати кожен раз багато нового коду, вирішено генерувати автоматично реалізацію підключення до Kafka та ksqlDb в Confluent Cloud, а також реалізацію сервісів, які будуть “під капотом” обробляти команди, події та реалізувати Event Sourcing. Для генерації коду обрано бібліотеку JavaPoet. Вона чудово підходить щоб автоматично створювати код під час обробки анотацій. Тобто ідея в тому, що якщо, наприклад, поставити над методом анотацію @CommandHandler, то “під капотом” створюється шматок коду з реалізацією імплементації роботи з інфраструктурою. Це суттєво зекономить час на розробку.

Також для зручності використання реалізовано можливість підписки до моделі запису за допомогою GraphQL. Це надає можливість отримувати сконфігуровані в підписці поля, а також дає повний і зрозумілий опис даних у API, а в нашому варіанті - це в схемі моделі даних в kafka. Це дає клієнтам можливість вимагати саме те, що їм потрібно, напряму з kafka, не використовуючи додаткову обробку подій. Це полегшує розвиток API та UI

з часом. В залежності від конфігурації підписки, UI, наприклад, безперервним потоком отримує саме те, що йому потрібно, не більше і не менше. Програми, які використовують GraphQL, є швидкими та стабільними, оскільки вони контролюють дані, які вони отримують, а не сервер.[8]

1.2.6 Архітектура рішення для фреймворку

Розглянемо архітектурне рішення, яке було узагальнено реалізоване. На рисунку 8 можемо бачити область “Service that use CQRS with own domain”. Це саме клієнтський сервіс, який і буде користуватися фреймворком. Все, що поза цією областю генерується автоматично та працює на основі ksqlDb та Kafka. Кластер потрібно створити самостійно, наприклад, в Confluent Cloud, та прописати параметри для доступу до нього.

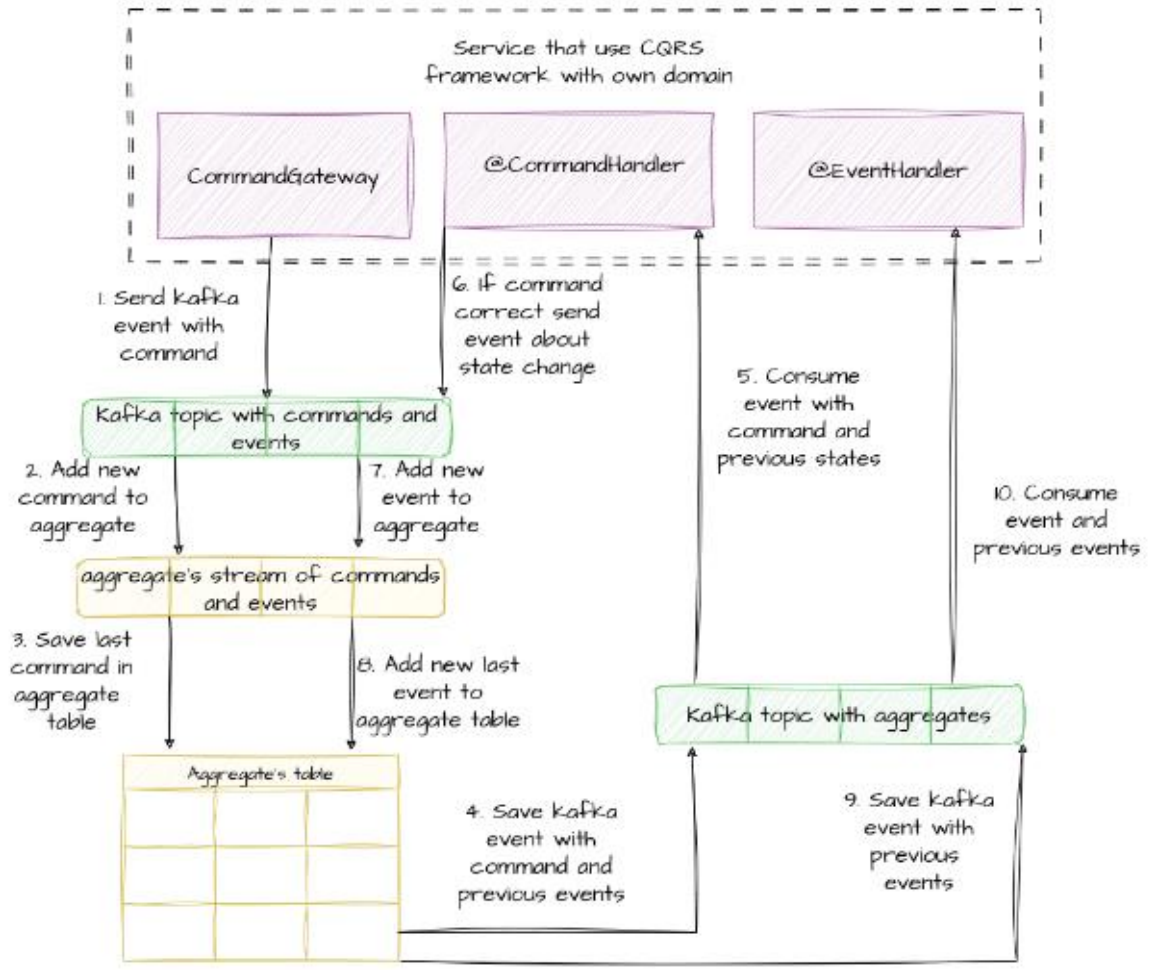

Рисунок 8: Діаграма реалізації CQRS

Код для CommandGateway прописаний також у реалізованій бібліотеці. Головна його функція - це відправляти повідомлення з командою в тему “commands-and-events” та пов’язувати їх з агрегатом. Агрегатом можу бути, наприклад, сутність з домену. Тоді всі події, які пов’язані суто з нею будуть надсилатися в тему і будуть помічатися, що вони належать певному агрегату за допомогою його ідентифікатора. На даний момент створюється окрема тема для подій про створення команд та івентів, але це можна легко змінити.

Після того, як повідомлення про створення команди потрапляє в тему “commands-and-events” стан системи обробляється потоком з ksqlDb та

поточний стан агрегату зберігається в “Aggregate’s table”. Команди також зберігаються в стані агрегату для того, щоб не було ситуації гонки і ми точно не могли відправити дві команди підряд, які суперечать одна одній.

Після того, як команда разом з станом агрегату (попередніми івентами) відправляється в тему “Kafka topic with aggregates”, яку слухає CommandHandler. Його реалізація також створена в реалізованій бібліотеці. Єдине, що повинен зробити користувач - це поставити анотацію @CommandHandler над методом, де потрібно реалізувати обробку команди. Це може бути перевірка на валідність команди, враховуючи попередній стан системи. Якщо команда задовольняє правилам домену, то після цього CommandHandler викликає метод з EventGateway, який надсилає повідомлення знову в тему “commands-and-events”. Це повідомлення знову проходить шлях через поток ksqlDb та потрапляє в таблицю станів. Таким чином стан агрегату знову оновлюється і до нього додається нове повідомлення. Це повідомлення потім надсилається в тему “Kafka topic with aggregates”, яку слухає ще EventHandler. Принцип роботи його схожий на CommandHandler, але вони слухають різні події, хоч і з однієї теми. Користувач також повинен реалізувати логіку для обробки повідомлення. Наприклад, це буде збереження нових даних в базу NoSQL. Таким чином реалізується повністю модель написання та її синхронізація з моделлю читання.

1.2.7 Реалізація фреймворку

Головна ціль написання бібліотеки - це зменшення кількості роботи для підключення CQRS. Для цього створено анотації @Command, @Event, @Aggregate, @AggregateId, @BaseAggregateId, @CommandHandler та @EventHandler. На основі них генерується додаткова логіка за допомогою JavaPoet під час обробки всіх анотацій, які існують в проекті. @Command

означає, що клас, який помічений нею, реалізує абстрактний клас Command, який CommandGateway приймає в параметрах, та який потім надсилається в тему “commands-and-events”. Аналогічно для @Event, тільки він говорить про наслідування класу Event та обробляється EventGateway. У кожного класу, поміченого як @Event чи @Command повинне бути поле, помічене як @AggregateId для того, щоб було зрозуміло до якого агрегату треба прив’язувати всі команди та івенти.

Для прикладу розглянемо використання оголошення команд та івентів, які відповідають домену магазину товарів, зображене на рисунку 9:

```
data class CreateOrderCommand(@AggregateId val orderId: String, val productId: String)
data class ConfirmOrderCommand(@AggregateId val orderId: String)
data class ShipOrderCommand(@AggregateId val orderId: String)

data class OrderCreatedEvent(@AggregateId val orderId: String, val productId: String)
data class OrderShippedEvent(@AggregateId val orderId: String)
data class OrderConfirmedEvent(@AggregateId val orderId: String)
```

Рисунок 9: Приклад створення команд та подій

Можемо бачити, що orderId виступає саме ідентифікатором агрегату. Тобто ці команди та івенти потім будуть в темі, прив’язані до певного товару.

Над класом, який узагальнює агрегат повинна стояти анотація @Aggregate, яка говорить про те, що поточний стан агрегату повинен відображатися в цьому класі. Також клас агрегату має містити поле з @BaseAggregateId для того, щоб створити ідентифікатор агрегату та прив’язувати події та команди до нього. Для прикладу можна розглянути рисунок 10, на якому можна побачити як створюється агрегат товару.

```java
@Aggregate
public class Order {

    3 usages
    @BaseAggregateId
    private String orderId;
    4 usages
    private boolean orderConfirmed;
```

Рисунок 10: Створення агрегату

@CommandHandler та @EventHandler анотації повинні стояти над методами, які реалізують обробку команд та івентів разом з поточним станом агрегату.

Для реалізації підключення до кластера використано джава клієнти для kafka та ksqlDb. Всі конфігурації заносяться в application.properties. Приклад конфігурацій для використання Confluent Cloud наведено на наступному рисунку 11.

```
# Required connection configs for Kafka producer, consumer, and admin
spring.kafka.sasl.mechanism=PLAIN
spring.kafka.bootstrap.servers=pkc-xmzwx.europe-central2.gcp.confluent.cloud:9092
spring.kafka.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule requ
spring.kafka.security.protocol=SASL_SSL

spring.ksqlDb.server.host=https://pksqlc-1oz6j.europe-central2.gcp.confluent.cloud
spring.ksqlDb.server.port=443
```

Рисунок 11: Приклад конфігурації для Confluent Cloud

Для цього було створено тестові кластери та використано інформацію для підключення до них. За допомогою цього фреймворк потім створює біни з конфігурацією для підключення до хмарного середовища та автоматичного створення тем, таблиці та потоку.

Щоб підключитися до кластера і створити теми додається бін KafkaAdmin, в якому використані всі параметри, що користувач прописує у файлі application.properties.

```
@Bean
public KafkaAdmin kafkaAdmin() {
    Map<String, Object> configs = new HashMap<>();
    configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
    configs.put(AdminClientConfig.SECURITY_PROTOCOL_CONFIG, securityProtocol);
    configs.put(SaslConfigs.SASL_MECHANISM, saslMechanism);
    configs.put(SaslConfigs.SASL_JAAS_CONFIG, saslJaas);
    return new KafkaAdmin(configs);
}
```

Рисунок 12: Створення Kafka Admin

Самі ж параметри автоматично створюються з файлу за допомогою Spring анотації @Value. Це має вигляд як на рисунку 13:

```
@Value(value = "${spring.kafka.bootstrap.servers}")
private String bootstrapAddress;
@Value(value = "${spring.kafka.security.protocol}")
private String securityProtocol;
@Value(value = "${spring.kafka.sasl.jaas.config}")
private String saslJaas;
@Value(value = "${spring.kafka.sasl.mechanism}")
private String saslMechanism;
```

Рисунок 13: Автоматичне вбудовування конфігурації для кластера

Як було сказано раніше, потрібні нам теми також створюються автоматично за допомогою біна kafkaAdmin. Разом з створенням біна commandsAndEventsTopic та aggragatesTopic робиться запит на кластер для створення відповідних тем там. Реалізація створення тем наведена в наступному скріншоті:

```
@Bean
public NewTopic commandsAndEventsTopic() {
    return TopicBuilder.name(COMMANDS_AND_EVENTS_TOPIC)
            .partitions( partitionCount: 20)
            .replicas( replicaCount: 1)
            .compact()
            .build();
}

@Bean
public NewTopic aggregatesTopic() {
    return TopicBuilder.name(AGGREGATES_TOPIC)
            .partitions( partitionCount: 20)
            .replicas( replicaCount: 1)
            .compact()
            .build();
}
```

Рисунок 14: Створення нових тем в Kafka

Щоб відправляти повідомлення з нашого сервісу автоматично конфігурується також ProducerFactory, якому потрібно передати всю інформацію для доступу в кластер. Надсилати івенти ми будемо за допомогою KafkaTemplate, який створений на основі ProducerFactory. Детальну реалізацію можна побачити на рисунку 15:

```java
@Bean
public ProducerFactory<String, String> producerFactory() {
    Map<String, Object> configProps = new HashMap<>();
    configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
    configProps.put(AdminClientConfig.SECURITY_PROTOCOL_CONFIG, securityProtocol);
    configProps.put(SaslConfigs.SASL_MECHANISM, saslMechanism);
    configProps.put(SaslConfigs.SASL_JAAS_CONFIG, saslJaas);
    configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

    return new DefaultKafkaProducerFactory<>(configProps);
}

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

Рисунок 15: Створення Producer Factory та Kafka Template

Для підключення до ksqlDb та створення потрібних таблиці та потоку нам також необхідні налаштування для цього. А саме host, port, username і password. При бажанні можна уникнути запис конфіденційної інформації в файл та записати його в зашифрованому вигляді. Дані для створення клієнту ksqlDb дістаються з файлів так само за допомогою анотації @Value від Spring як показано на рисунку 16.

```java
@Value(value = "${spring.ksqlDb.server.host}")
private String serverHost;
@Value(value = "${spring.ksqlDb.server.port}")
private int serverPort;
@Value(value = "${spring.ksqlDb.auth.username}")
private String basicAuthUsername;
@Value(value = "${spring.ksqlDb.auth.password}")
private String basicAuthPassword;
```

Рисунок 16: Автоматичне створення конфігурації для ksqlDb

Клієнт створюється на основі цих даних та надсилає автоматично запити до нещодавно доданих адрес кластера за допомогою HTTP2: запити

Pull і Push обслуговуються кінцевою точкою (endpoint-ом) “/query-stream”, а запис - точкою “/inserts-stream”. Усі інші запити обробляються за посиланням “/ksql”. Клієнт сумісний лише з розгортаннями ksqlDB версії 0.10.0 або новішої. В коді це виглядає набагато простіше, а саме як на рисунку 17:

```
@Bean
public Client createClient() {
    ClientOptions options = ClientOptions.create()
            .setHost(serverHost)
            .setPort(serverPort)
            .setUseTls(true)
            .setBasicAuthCredentials(basicAuthUsername, basicAuthPassword);

    return Client.create(options);
}

@Bean
public Map<String, String> executeProperties() {
    return Map.of(
            k1: "auto.offset.reset",  v1: "earliest",
            k2: "log.retention.ms",  v2: "-1",
            k3: "log.retention.bytes",  v3: "-1"
    );
}
```

Рисунок 17: Створення клієнта для роботи з ksqlDb

Бін “executeProperties” використовуються для виконання запитів над ksqlDb прямо з клієнтського сервісу. Для того, щоб була можливість читати історичну інформацію з самого початку властивості “auto.offset.rest” присвоєно значення “earliest”.

Операції з ksqlDb виглядають так само, як звичайні SQL запити. Щоб потоки та таблиці створювалась автоматично, додано запити в метод, який виконується відразу після старту клієнтського сервісу. Це можливо завдяки слухачам подій про Application Context (місце, де зберігаються всі біни в програмах, написаних з використанням Spring). Для конкретного варіанту виконання логіки відразу після створення контексту, обрано

ContextStartedEvent, а метод, який на нього реагує та робить запити в ksqlDb помічено анотацією @EventListener.

Для створення потоку нам потрібно виконати запит, наведений на рисунку 18:

```
CREATE STREAM IF NOT EXISTS COMMANDS_AND_EVENTS (
  AGGREGATE_ID STRING KEY,
  IS_COMMAND BOOLEAN,
  TYPE STRING,
  BODY STRING
) WITH (
  KAFKA_TOPIC='commands-and-events',
  PARTITIONS=20,
  REPLICAS=1,
  VALUE_FORMAT='JSON'
);
```

Рисунок 18: Запит для створення потоку в ksqlDb

Як можемо бачити, цей потік створюється на основі теми “commands-and-events” та збирає інформацію про ідентифікатор агрегату, його тип, тіло та інформацію чи це команда.

Аналогічно до потоку виконується також запит на побудову ksql таблиці. Він має вигляд як на рисунку 19. Можемо бачити, що ця таблиця створюється на основі потоку commands_and_events і збирає інформацію про всі стани агрегату, включаючи тип події та тіло. Також вона поєднана з темою “aggregates”, з якою потім читають повідомлення про подію EventHandler та CommandHandler, адже ця тема тепер безперервним потоком буде посилати події про зміну стану агрегата.

```
CREATE TABLE IF NOT EXISTS AGGREGATES WITH (
                                                KAFKA_TOPIC='aggregates',
                                                PARTITIONS=20,
                                                REPLICAS=1,
                                                VALUE_FORMAT='JSON'
                                                ) AS SELECT
    AGGREGATE_ID,
    COLLECT_LIST(IS_COMMAND) AS IS_COMMAND_LIST,
    COLLECT_LIST(TYPE) AS TYPE_LIST,
    COLLECT_LIST(JSON_DATA) AS JSON_DATA_LIST
    FROM COMMANDS_AND_EVENTS
    GROUP BY AGGREGATE_ID
    EMIT CHANGES;
```

Рисунок 19: Запит для створення таблиці в ksqlDb

1.2.8 Огляд існуючих фреймворків для використання CQRS та Event Sourcing. AxonIQ

Через відносно недавню появу та поширення концепції CQRS, на даний момент фреймворків та сервісів, які б допомагали з реалізацією, небагато. Найпопулярніше рішення розробляється AxonIQ.

AXON - це фреймворк з відкритим вихідним кодом для керованих подіями мікросервісів і проектування на основі домену. Це платформа Java, яка надає розробникам доступ до коду для реалізації важливих компонентів, включаючи агрегати, команди та шини подій, а також легку інтеграцію архітектурного шаблону CQRS. Принципи архітектури, такі як DDD і CQRS, є основою фреймворку. Крім того, сервіс надає можливість інтегрувати бізнес-логіку в численні розподілені мікросервіси за допомогою інфраструктури AXON, а також створювати неймовірно масштабовані та гнучкі програми. [10]

Якщо нам потрібне хмарне рішення без сервера для цього є Axon Server, який виступає сховищем подій та їх координацією. Але головний недолік - це висока вартість, яка рахується за кількість надісланих подій. Тому системам з великим навантаженням це обійдеться дорого, особливо якщо вони додатково все одно ще використовують Confluent Cloud для інших цілей.

РОЗДІЛ 2: АНАЛІЗ ОПТИМІЗАЦІЇ БЕЗ ЗМІНИ АРХІТЕКТУРНОГО ПІДХОДУ

2.1 Огляд системи для проведення тестувань

Для прикладу, в роботі методи перевіряються на основі псевдо реальної системи для друку банківських документів-звітів. Це може бути розглянутим як приклад маленької частини логіки реального проекту, що показує всі проблеми з продуктивністю у найменшому масштабі, адже зазвичай веб-системи мають великий обсяг обов'язків. Система складається з двох мікросервісів: printing-service та report-generation service. report-generation відповідальний за пошук персональних даних з бази за іменем користувача, підрахунок середньої суми банківських транзакцій за іменем клієнта та за його країною. Потім генерується звіт у форматі документу і відправляється назад до printing-service, який повинен викликати фізичний принтер для друку. Проектування системи не найбільш вдале, особливо враховуючи, що один цикл логіки займає 8 секунд. Також є багато вразливих місць, які можна проаналізувати.

Для імітації навантаження на систему обрано Jmeter. Сервіси для проведення аналізу написані на мові Java, з використанням таких фреймворків як Spring та Hibernate. Також обрано Docker як найпопулярнішу платформу для контейнеризації. Для оркестрації контейнерів створено локальний кластер Minikube (спрощена версія кластера для роботи з Kubernetes), але аналогічні методи можна використовувати і в хмарному середовищі. На даний момент це один з найпопулярніших стеків технологій для написання веб-сервісів. Також багато великих систем вже написано на Java, тому приведені методи актуальні і для рефакторингу та оптимізації роботи існуючих сервісів.

Як середовище для виконання було обрано Kubernetes, тому далі будемо використовувати таке поняття як:

pod – це група з одного або декількох контейнерів (таких як Docker або rkt), що може мати спільне сховище даних (volumes), унікальну IP-адресу і містить інформацію як їх запустити.

На рисунку 20 бачимо, що у нас у статусі “Running” усі потрібні нам pods: екземпляри printing, report-generation та бази даних PostgreSQL.

```
syarmolenko@syarmolenko-2nout:~$ kubectl get pods -n=default
NAME                                READY   STATUS    RESTARTS   AGE
hello-node-67949d9db-96tmb          1/1     Running   0          20m
printing-57cf68d46-rzhtk            1/1     Running   0          6m31s
psql-test-postgresql-0              1/1     Running   0          3h51m
report-generation-85fb657f4c-4nvbz  1/1     Running   0          24m
```

Рисунок 20: Інформація про систему, яка тестується в Kubernetes

При першому ж тестуванні з навантаженням на рисунку 21 видно, що при 100 користувачах найбільший час очікування близько 10 секунд.

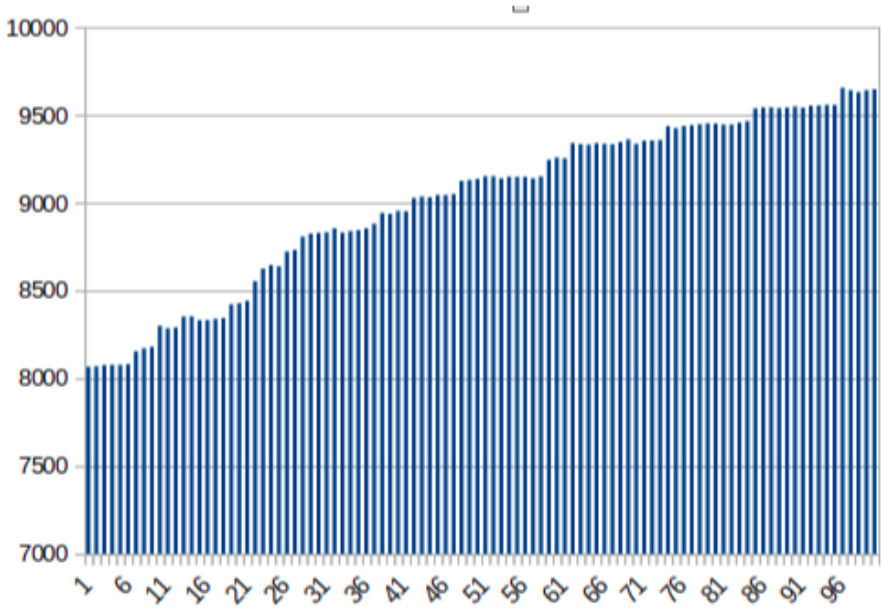

Рисунок 21: Навантаження системи при 100 користувачах

На перший погляд 10 секунд це небагато, але так здається через те, що кількість користувачів далека від реальності. Якщо змінити це і зробити запити від імені 1000 користувачів, то час виконання підходить майже до 45 секунд, як показано на рисунку 21. Звичайно, це не в межах норми.

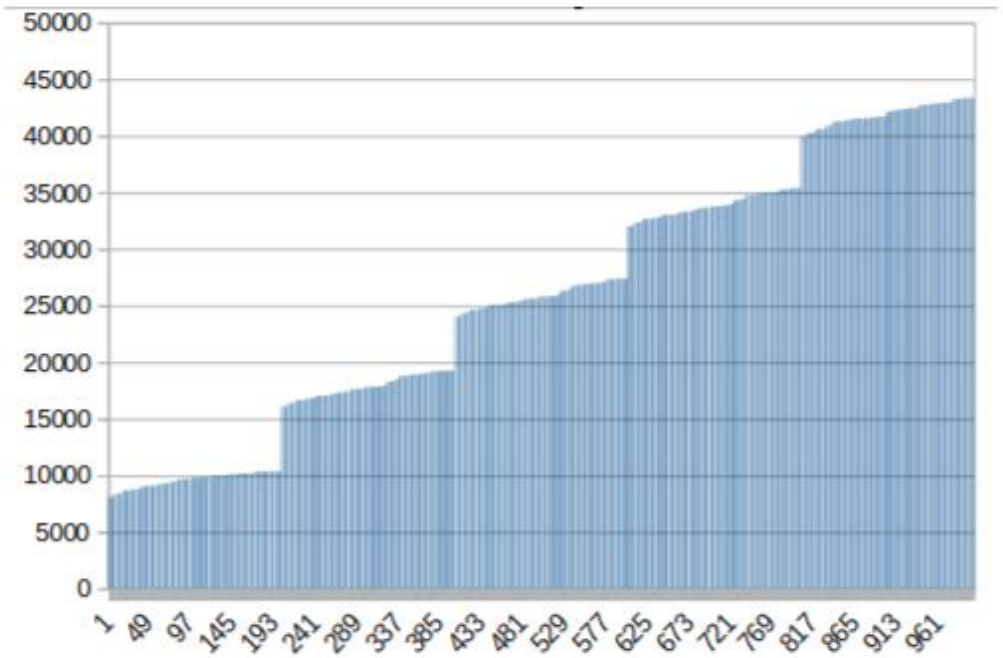

Рисунок 22: Навантаження системи при 1000 користувачах

2.2 Масштабування мікросервісів

В системах з великою кількістю користувачів рано чи пізно настає момент, коли їх логіка вже максимально оптимізована, але навантаження настільки велике, що просто не вистачає ресурсів для оптимальної обробки запитів від користувачів. В такий момент потрібно задуматись про масштабування сервісів.

Вертикальне масштабування означає збільшення hardware ресурсів для досягнення оптимізації роботи системи (або умовних ресурсів таких як CPU та пам’ять, які надаються середовищем). З недоліків – це доволі дороге рішення.

В нашому випадку розглянемо горизонтальне масштабування, яке реагує на збільшення навантаження розгортанням більшої кількості екземплярів сервісу на наявних ресурсах. В таких середовищах як PaaS (платформа як сервіс) - Cloud Foundry, Amazon, Heroku, або PaaS – CaaS(контейнер як сервіс) – Kubernetes, не потрібно вручну створювати нові екземпляри сервісу та слідкувати за балансуванням навантаження, адже можна використовувати готовий функціонал, що налаштовується досить швидко.

За приклад обрано Horizontal Pod Autoscaler, який пропонує Kubernetes. Принцип його роботи схематично зображений на рисунку 23:

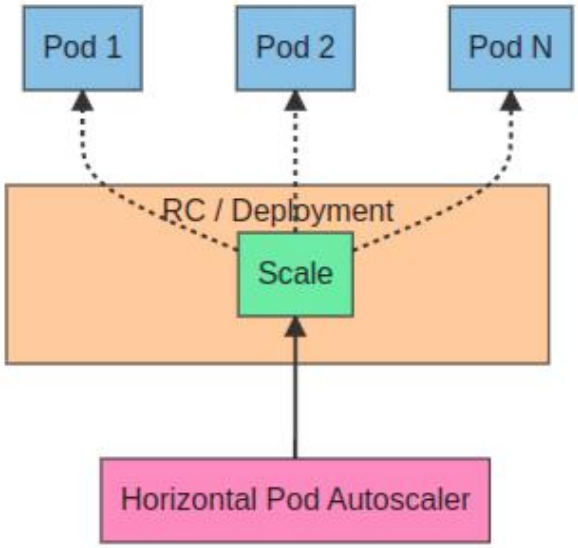

Рисунок 23: Схематичне зображення роботи Horizontal Pod Autoscaler

Kubernetes контролює кількість екземплярів сервісу з певною періодичністю, тобто це не безперервний процес. Інтервал встановлюється параметром --horizontal-pod-autoscaler-sync-period (за замовчуванням це 15 секунд).[2]

Один раз протягом кожного періоду диспетчер контролера порівнює використання ресурсів з максимальними значеннями, указаними в кожному визначенні Horizontal Pod Autoscaler (наприклад, CPU=80%). Менеджер

контролера знаходить цільовий ресурс, визначений scaleTargetRef, потім вибирає модулі на основі міток .spec.selector цільового ресурсу та отримує показники з API метрик ресурсів (це може бути використання CPU). Для моніторингу метрик можна налаштувати існуючі адони Kubernetes. Приклад налаштування metrics-server показаний на рисунку 24:

Рисунок 24: Налаштування адону metrics-server

Якщо максимальне значення досягається, контролер обчислює використання як відсоток від еквівалентного запиту ресурсу для контейнерів у кожному існуючому екземплярі. Потім бере середнє значення використання для всіх екземплярів і створює співвідношення, яке використовується для масштабування кількості бажаних реплік. Якщо для деяких pods неможливо знайти статистику по запиту ресурсу, автоматичне масштабування не виконуватиме жодних дій щодо цього показника. Контролер Horizontal Pod Autoscaler отримує доступ до відповідних ресурсів робочого навантаження, які підтримують масштабування (наприклад, Deployments і StatefulSet). Кожен із цих ресурсів має підресурс під назвою scale, інтерфейс, який дозволяє динамічно встановлювати кількість реплік і перевіряти кожен із їхніх поточних станів. Створюємо автоматичне масштабування для проекту як показано на рисунку 25:

Рисунок 25: Створення автоматичного масштабування

Після успішного виконання цієї команди можемо бачити на рисунку 26, що інструменти для автоматичного масштабування створились та готові до роботи.

```
syarmolenko@syarmolenko-2nout:~$ kubectl get hpa
NAME                REFERENCE                      TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
printing            Deployment/printing            <unknown>/50%   1         6         1          3m28s
report-generation   Deployment/report-generation   <unknown>/50%   1         6         1          4m44s
```

Рисунок 26: Результат створення автоматичного масштабування

Починаємо робити навантаження на систему за допомогою Jmeter, імітуючи запити від 1000 користувачів. Виконавши команду, зображену на рисунку 27, бачимо, що утворилися нові екземпляри сервісів. Тепер навантаження буде розподілятися між ними і контролюватиметься за допомогою іншого інструменту - LoadBalancer. В Kubernetes за замовчуванням використовується принцип Round-Robin. Він розподіляє запити по колу. Але можна створити і свою реалізацію LoadBalancer, якщо потрібна додаткова логіка :

```
syarmolenko@syarmolenko-2nout:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS        AGE
hello-node-67949d9db-96tmb          1/1     Running   1 (3h10m ago)   14h
printing-57cf68d46-k5nlk            1/1     Running   0               64s
printing-57cf68d46-m9vb7            1/1     Running   0               84s
printing-57cf68d46-qscw6            1/1     Running   0               43s
psql-test-postgresql-0              1/1     Running   0               3h7m
report-generation-85fb657f4c-f5vmr  1/1     Running   0               3s
report-generation-85fb657f4c-f9fj2  1/1     Running   0               22s
report-generation-85fb657f4c-ksx7v  1/1     Running   0               107s
```

Рисунок 27: Результат виконання роботи автоматичного масштабування

Проаналізувавши час виконання, маємо діаграму, зображену на рисунку 28:

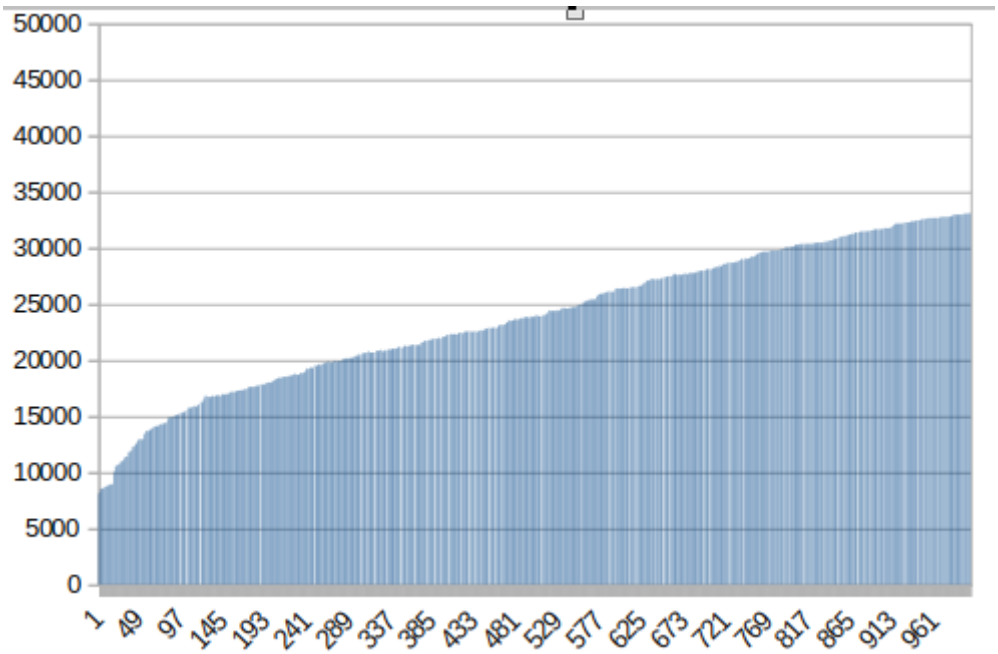

Рисунок 28: Навантаження системи при 1000 користувачах та роботі автоматичного масштабування

Можна бачити, що в порівнянні з одним екземпляром три працюють набагато краще та стабільніше. А останній запит виконується на 10-14 секунд швидше.

Проаналізувавши такий підхід на практиці можна виділити кілька недоліків такого масштабування. Головний – це те, що потрібно більш уважно ставитись до того, які сервіси дозволяти дублювати, а які ні. Яскраві приклади, для яких це не підходить:

- сервіси, які слухають повідомлення про події та логічно їх обробляють (приклад – повідомлення про подію через Kafka або Rabbit MQ)
- сервіси, які містять логіку виконання періодичних обробок даних з бази (наприклад, починають обробку збережених іншим сервісом даних, якщо вони відповідають певним умовам).

Щоб уникнути потенційних проблем з багаторазовим виконанням якоїсь логіки рекомендовано виносити її окремо в інші сервіси, які не будуть масштабуватись. Наприклад, у випадку event можна винести його consumer в окремий сервіс-демон, який буде слухати повідомлення та викликати інший (можливо масштабований) сервіс через його API для обробки.

2.3 Використання gRPC замість REST

На даний момент найбільш розповсюджений та популярний API для зовнішньої та внутрішньої комунікації мікросервісів – це REST API. Головним молодим його конкурентом вважається саме gRPC – метод віддаленого виклику процедур, розроблений Google на основі протоколу HTTP2. Одна з переваг gRPC заключається в тому, що в порівнянні з REST, який передає дані в форматі JSON/XML, він використовую Protobuf. Цей “буфер протоколу” серіалізує і десеріалізує структуровані дані для передачі в двоїчному форматі, тобто доволі зжато, що вирішує проблему зі швидкістю доставки та ваги повідомлень.

Друга перевага gRPC – це протокол HTTP2, який дає можливість обслуговувати одночасно більше одного клієнта, адже для кожного може відкритись новий потоковий зв’язок. В REST всі запити ж стають у чергу і чекають на свою відповідь, що може сильно сповільнювати роботу, якщо для одного запиту потрібно багато часу на обробку.[4]

Зсилаючись на результату проведеного аналізу, gRPC приблизно в 8 раз швидше REST при отриманні даних та в 10 раз швидше при їх відправці.

Головною проблемою gRPC, як парадоксально б це не звучало, є його новизна та невелика розповсюдженість. Для використання REST вже адаптовані фреймворки та різні зовнішні сервіси. Також більшість компаній вже звикли до такого підходу і очікують публічні API саме з REST. Тому на даний момент краще розглядати gRPC як метод комунікації між

мікросервісами з внутрішньою логікою, які не будуть перевикористані іншими клієнтами.

Також імплементація сервісів на основі REST займає набагато менше часу ніж на основі gRPC. Особливо це впливає на бажання компанії рефакторити існуючі сервіси. В додаток, незважаючи на всі переваги, Protobuf, на відміну від JSON/XML, незручний для читання і ним важко оперувати людям.

2.4 Кешування даних з бази за допомогою ORM фреймворків

Неправильна робота з базою даних може дуже сильно уповільнювати всю систему. Особливо це актуально, коли ми часто викликаємо запит в базу на отримання даних, що дуже рідко змінюються (результат однаковий, але часу можна втратити багато). Для оптимізації таких випадків ідеально відходить кеш в ORM фреймворках.[6] Розглянемо його на прикладі Hibernate.

Кеш 2-го рівня — це кеш із областю процесу, пов’язаний з SessionFactory. Він переживе Session (одиниця роботи з базою, або один сеанс запитів) та може повторно використовуватися в новому сеансі. SessionFactory, як правило, одна на програму. За замовчуванням кеш 2-го рівня не ввімкнено, адже не для всіх сутностей він підходить, а саме для тих, що часто оновлюються. Тому нам потрібно вирішувати для яких увімкнути кеш, а для яких ні.

Кеш не зберігає екземпляри сутності — натомість Hibernate використовує так званий “dehydrated” стан. Його можна розглядати як десеріалізовану сутність, схожу на масив рядків, цілих чисел тощо, а ідентифікатор сутності є вказівником на зневоднену сутність. Концептуально можна розглядати це як Map, яка містить ідентифікатор як ключ і масив як значення.

Також кеш контролює стан сутностей і оновлює кешоване значення при його зміні автоматично. Це дуже зручно, адже не потрібно вручну контролювати інвалідацію даних.

Фактична реалізація кешу 2-го рівня не виконується Hibernate. Натомість Hibernate має концепцію плагіна для кешування, яка використовується, наприклад, EHCache.

В нашому випадку існує сутність ClientInformation(firstName, lastName, email, company,), яка змінюється дуже рідко, але ми дістаємо її при кожному запиті на генерування звіту. Тому вона ідеально підходить для кешування. Середній час запиту займає 333 мілісекунди (в базі 2000 клієнтів) як показано на рисунку 29:

Рисунок 29: Результат виконання запиту на отримання інформації про клієнта

Підключивши кеш зменшили це значення до 8, як показано на рисунку 30, при тому, що наша база даних не містить навіть частини реальної інформації з існуючих систем.

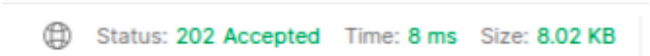

Рисунок 30: Результат виконання запиту на отримання інформації про клієнта, використовуючи ORM кеш

Якщо робити 1000 запитів на всю логіку, то час виконання приблизно виглядає як на рисунку 31. В порівнянні з початковим варіантом без кешування це економить 0,5 секунд:

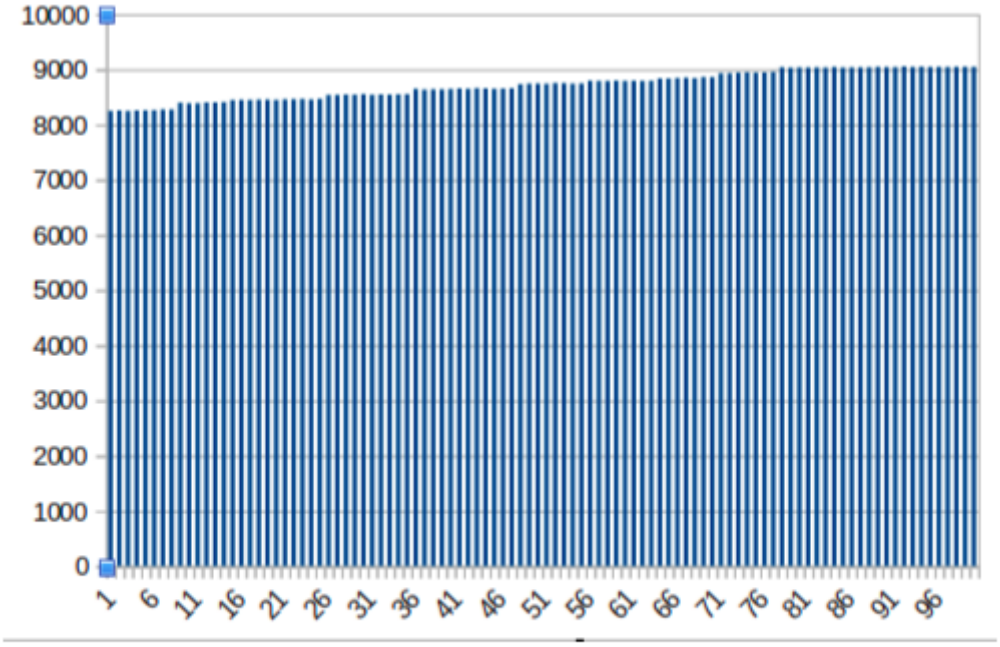

Рисунок 31: Час виконання запиту, який використовує ORM кеш

Кеш другого рівня можна увімкнути також для session Query. Виконання ж нативних update запитів призведе до інвалідації всього кеша, адже Hibernate не може зрозуміти що саме оновлюється. Але це вирішується додаванням інформації про те, для якого регіону чи сутності виконується оновлення. Приклад конфігурації показаний на рисунку 32

```
Query q = em.createNativeQuery("");
q.unwrap(NativeQuery.class).addSynchronizedEntityClass(Entity.class);
q.executeUpdate();
```

Рисунок 32: Конфігурація кеша для нативних запитів

2.5 Кешування результатів обробки за допомогою NoSQL

Існують випадки коли потрібно кешувати не тільки дані з бази, а й результати їх обробки. В нашому сервісі ми генеруємо середнє значення транзакції для всіх людей з певної країни.

По-перше, генерувати кожен раз значення для людей з однієї країни невигідно, адже воно не буде сильно відрізнятись. По-друге, це середнє значення, тому можна не дивитись на похибку.

Для цього ідеально підходить Redis, адже можна зберігати середнє значення транзакції, наприклад, за назвою країни і діставати дані швидше з Redis за ключем, ніж знову обробляти дані з бази, яка може бути дуже великою.[3]

Реалізація дала нам хоч і небагато виграшу, але його можна побачити в різниці часу виконання запиту на рисунку 33:

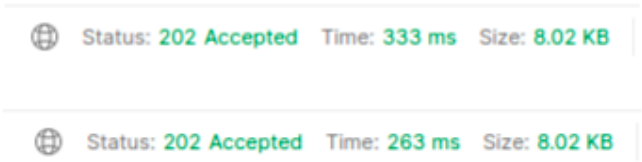

Рисунок 33: Порівняння часу виконання запиту використовуючи Redis та не використовуючи

2.6 Асинхронна обробка запитів Rest API

Один з способів не блокувати користувачів очікуванням на результат – це використання тредів для асинхронної обробки запиту.[5] У випадку тестованої системи додано @Async для друку документів, адже нам важливо запустити процес і успіхом буде фізичний роздрукований документ.

Отримали виконання запиту за 3 мілісекунди замість 8 секунд (звичайно, треди будуть його виконувати 8 секунд, але для користувачів це вже не важливо).

Зручність @Async в тому, що нам досить поставити анотацію і вся логіка з пулом тредів буде автоматичною. Але якщо потрібно, ми можемо написати свою власну чергу та executor. Це може знадобитись, якщо ми хочемо, наприклад, забезпечити користувачів з різних тенантів однаковими правами і брати запит на виконання по-черзі з кожного теннанту.

Отримані результати можна побачити на рисунку 34:

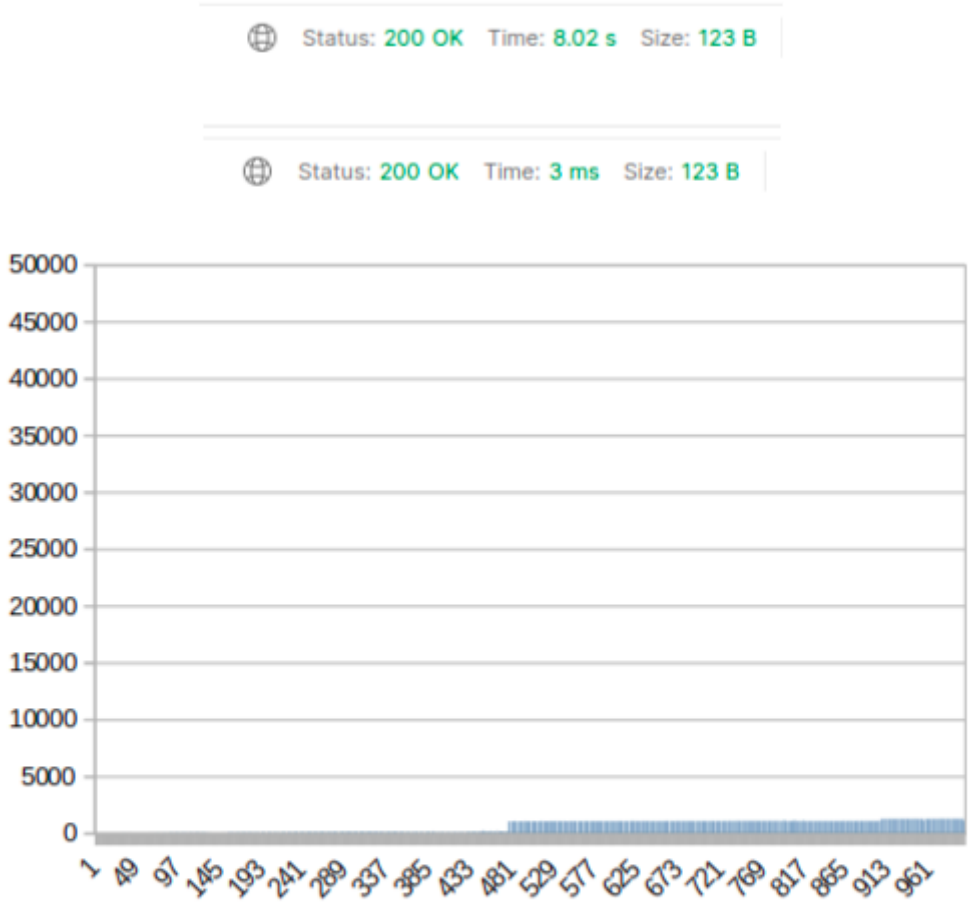

Рисунок 34: Результати використання @Async

ВИСНОВКИ

Питання оптимізації роботи мікросервісів завжди дуже нагальне. Адже це напряму впливає на враження клієнтів від продукту.

В ході виконання проекту було розглянуто питання пов’язані з розробкою фреймворка для полегшення інтеграції принципу CQRS, що дозволить ефективно маніпулювати даними в веб сервісах.

Після постановки задачі, було обрано інструменти та інфраструктуру для вирішення даної проблеми.

Протягом виконання даної роботи було покращено навички роботи з мовою програмування Java та пов’язаним стеком технологій, а саме та meter фреймворків Spring та Hibernate, також йкраще опановано навички, пов’язані з Docker, Minikube, Kubernetes, ksqlDb, Kafka, JavaPoet. Було поглиблено знання про проектування архітектури веб сервісів.

Було розроблено програмний застосунок, що спрощує інтеграцію CQRS.

Після виконання даної роботи була досягнута фінальна мета – було розроблено фреймворк та проаналізовано найпопулярніші методи оптимізації, які дали позитивні результати. У роботі розглянуто базові методи для покращення роботи мікросервісів, а саме масштабування, кеш, асинхронний виклик сервісу, gRPC. Також наведені мінуси та випадки, коли метод використовувати не варто. В нашому конкретному прикладі всі способи показали дуже продуктивну роботу. Але кожен проект потрібно аналізувати окремо, знаходячи вразливі місця та обираючи найкраще рішення.

Головний висновок, який можна винести з роботи - це те, що потрібно вкладати ресурси в коректне планування систем з самого початку, адже це вкінці виявиться набагато дешевшим ніж рефакторинг.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A. Tanenbaum, Distributed Systems, 3rd ed. / A. Tanenbaum, M. Steen., 2017
2. Horizontal Pod Autoscaling [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io
3. Bruce M. Microservices in Action / M. Bruce, P. Pereira.
4. gRPC documentation [Електронний ресурс] – Режим доступу до ресурсу: https://grpc.io/docs
5. Wall C. Spring in Action, 5th Edition / Craig Wall
6. Hibernate ORM documentation [Електронний ресурс] – Режим доступу до ресурсу: https://hibernate.org/orm/documentation/6.1/
7. Zaidman A. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare / Zaidman Andy
8. GraphQl documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.confluent.io/blog/intro-to-graphql-an-api-for-kafka-data/
9. Kafka documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.confluent.io/blog/infinite-kafka-storage-in-confluent-platform/
10. Naireen V. Axon and Kafka - How does Axon compare to Apache Kafka? [Електронний ресурс] / Vijay Naireen – Режим доступу до ресурсу: https://www.axoniq.io/blog/axon-and-kafka.
11. Young G. CQRS Documents / Greg Young
12. ksqlDb documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.ksqldb.io/en/latest/
13. Confluent Cloud documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.confluent.io/confluent-cloud/