КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

Кваліфікаційна робота

на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки

на тему:

РОЗРОБКА ВИСОКО МАСШТАБОВАНОГО ДОДАТКА З ВИКОРИСТАННЯМ АРХІТЕКТУРИ МІКРОСЕРВІСІВ

Виконав студент 4-го курсу
Піонтковський Ігор Ігорович

_____
(підпис)

Науковий керівник:
доцент, кандидат фіз-мат наук
Кузенко Володимир Федорович

_____
(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 теорії та технології програмування
«____» _____ 202_ р.,
протокол № ___
Завідувач кафедри
М. С. Нікітченко _____
(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 46 сторінок 18 ілюстрацій 1 таблиця 14 джерел.

МІКРОСЕРВІСИ, МАСШТАБУВАННЯ СЕРВІСІВ, JAVA, SPRING

Об’єктом даної роботи є розробка програмного засобу, який являє собою базову систему, розроблнну з використанням мікросервісної архітектури та з можливим безперешкодним розширенням та масштабуванням.

Метою даної роботи є створення скелету масштабованого додатку з використанням мікросервісної архітектури.

Методи розроблення: аналіз існуючих проблем, проектування майбутньої архітектури програми, покрокова розробка з розбиттям на під задачі.

Інструменти розроблення: Для розробки даної програмної реалізації було обрано мову Java. Під час розробки у якості інтегрованого середовища було обрано Intelij IDEA. Для збірки та компіляції програми було використано інструменти maven та docker.

Результат роботи: проведено аналіз проблем з якими стикаються під час розробки систем з використанням мікросервісної архітектури. Розроблено програмний засіб використовуючи мікросервісну архітектуру який легко розширювати, легко масштабується як горизонтально так і вертикально та який має інтеграції з різними допоміжними системи.

Програмний засіб можна застосувати як основу для розробки більш специфічних проектних рішень на основі високомасштабованої мікросервісної архітектури.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

CPU - central processing unit

RAM - Random-access memory

IOPS - Input/Output Operations Per Second

SaaS -Software as a service

API - application programming interface

IDC - Interntional Data Corporation

SOA - Service-oriented architecture

ESB - enterprise service bus

ADC - analog-to-digital converter

ACID - (atomicity, consistency, isolation, durability) is a set of properties of database transactions

CQRS - Command Query Responsibility Segregation

2PC - 2 phase commit protocol

XA - eXtended Architecture

JWT - JSON Web Token

IP - Internet Protocol

SOA - service-oriented architecture

URL - Uniform Resource Locator,

HTTP - HyperText Transfer Protocol

REST - Representational State Transfer

IaC - Infrastructure-as-Code

CRUD - create, read, update, delete

JVM - Java Virtual Machine

GC - Garbage Collection

ВСТУП

Оцінка сучасного стану об'єкта розробки. Успішний веб-додаток повинен плавно та ефективно забезпечувати розширенням та бути розробленим з урахуванням масштабованості. Масштабована веб-програма зможе з легкістю пристосуватись до збільшення кількості користувачів та навантаження. Критерій масштабованості - має найбільше значення для управління розширенням додатку.

Мікросервіси та контейнери за своєю суттю є більш масштабованими, ніж застаріла IT-інфраструктура та моделі розробки додатків, але деякі проблеми все ще існують.

Актуальність роботи та підстави для її виконання. У сучасному світі люди хочуть отримати дані відразу, миттєво. Не існує концепції очікування завантаження веб-сторінки, завантаження зображення чи обробки форми. Якщо ваш додаток не розроблений належним чином і не може впоратися зі збільшенням кількості користувачів та робочого навантаження, то він неминуче залишиться в пилу. Тому, на мою думку, є актуальною тема розробки скелету масштабованого додатку з використанням мікросервісної архітектури.

Мета й завдання роботи. Метою даної роботи є створення скелету масштабованого додатку з використанням мікросервісної архітектури. Для досягнення цієї мети поставлено такі завдання:

- Дослідити проблеми які виникають під час розробки застосунків з мікросервісної архітектури;
- Дослідити шляхи досягнення високомасштабованого додатку;
- Дослідити особливості високомасштабованої архітектури мікросервісів;
- Розробити скелет високомасштабованого застосунку який буде мати наступний функціонал:

- Повинен мати API для аутентифікації та авторизації;
- Повинен мати API для взаємодії з даними для користувачів;

Об'єкт, методи й засоби розроблення. Отже, об’єктом є розробка програмного засобу, який являє собою базову систему з використанням мікросервісної архітектури та з можливим безперешкодним розширенням та масштабуванням. Основними технологіями які були використані під час розробки стали: Spring Cloud, Spring Boot та Docker.

Для розробки програмного засобу був проведений огляд основних проблеми з якими можуть стикатися розробники під час процесу розробки застосунків використовуючи мікросервісну архітектуру, було наведено шляхи вирішення цих проблем. Під час розробки було використано як представлені напрацювання так і повністю нові ідеї. Проект було розроблено на ОС Ubuntu 20.04. Для створення самого проекту IntelliJ IDEA та Java 8 .

Можливі сфери застосування. Програмний засіб можна застосувати як основу для розробки більш специфічних проектних рішень на основі високомасштабованої мікросервісної архітектури.

РОЗДІЛ 1. ОСОБЛИВОСТІ МАСШТАБУВАННЯ ЗАСТОСУНКІВ

Успішний веб-додаток повинен плавно та ефективно забезпечувати розширення та бути розробленим з урахуванням масштабованості. Масштабована веб-програма зможе з легкістю пристосуватись до збільшення кількості користувачів та навантаження. Критерій масштабованості - має найбільше значення для управління розширення додатку.

1.1 Чому масштабованість важлива

У сучасному світі люди хочуть отримати дані відразу, миттєво. Не існує концепції очікування завантаження веб-сторінки, завантаження зображення чи обробки форми. Якщо додаток не розроблений належним чином і не може впоратися зі збільшенням кількості користувачів та робочого навантаження, то він неминуче залишиться в пилу.

Масштабованість має вирішальне значення для життя будь-якого веб-додатку, інакше його просто не вдасться застосовувати. Розуміння того, що таке масштабованість, і як використовувати принципи та стандарти від дизайну до впровадження - це те, що в наш час не вдається більшості компаній, що розробляють веб-сайти та програми.

Масштабованість допоможе вирішити кілька проблем, пов’язаних з продуктивністю та обслуговуванням коду. Коли впроваджується масштабована розробка, проблеми, пов'язані з продуктивністю, можна вирішити безпосередньо, а код можна організувати набагато ефективніше. Створення масштабованого веб-додатку - це не нова мова чи структура, це методологія та набір принципів, які приймає бізнес.

Створюючи масштабований веб-додаток, слід враховувати загальний досвід користувачів, час завантаження сторінки, час, необхідний для внесення змін у код, і навіть вартість оновлення всієї програми.

Щоб веб-додаток вважався масштабованим, необхідно враховувати кілька важливих факторів. Тут торкаємось декількох:

- Продуктивність: У якій точці навантаження веб-програма почне відчувати проблеми? Все це зводиться до балансування. Наприклад, як швидко веб-програма може впоратися зі збільшенням кількості користувачів?
- Відновлюваність: Як швидко система може відновити початковий робочий стан? Після виходу з ладу необхідна можливість швидкого відновлення роботи. Коли веб-сайт не працює навіть протягом короткого періоду часу, це може коштувати компаніям мільйонів доходу. Тривалість роботи є критичною.
- Простота управління: Наскільки легко підтримувати та оновлювати веб-програми та процеси? Масштабований розвиток повинен це враховувати.

1.2 Інструменти та процеси

Інструменти та процеси, які допомагають компаніям створювати масштабовані програми, включають хмарне сховище, програмне забезпечення для балансування навантаження, реалізацію мікросервісів та кешування.

- Хмарне сховище: зберігання даних «у хмарі» - це просто дані вашого додатку, які зберігаються на віддалених серверах, доступні з будь-якого місця та мають можливість розширення за потреби. Постачальники хмарних сховищ беруть на себе відповідальність зробити ваші дані доступними та безпечними.
- Програмне забезпечення для балансування навантаження: це програмне забезпечення відстежує використання трафіку веб-додатком і через набір серверів розподіляє користувачів для розподілу навантаження на

декілька серверів. Це запобігає проблемі, коли одному серверу доведеться обробляти приплив користувачів.

- Мікросервіси: метод розробки децентралізованих систем та полегшених протоколів, які мають модульну конструкцію та дозволяють модернізувати та обслуговувати без необхідності впливати на всю програму.
- Кешування: Ефективне використання кешування у веб-програмі заощадить витрати на завантаження даних та збільшить можливість повторного використання. Це дозволяє швидше отримувати дані, а не завжди повертатися до вихідного джерела (тобто бази даних).

1.3 Масштабування по горизонталі

Масштабування по горизонталі - вид масштабування, який в основному базується на додаванні більшої кількості машин або налаштування кластера або розподіленого середовища для вашої програмної системи. Зазвичай для цього потрібна програма балансування навантаження, яка є середнім компонентом у стандартній 3-рівневій архітектурній моделі клієнт-сервер.

Розподілювач навантаження відповідає за розподіл запитів користувачів (навантаження) між різними фоновими системами / машинами / вузлами в кластері. Кожна з цих внутрішніх машин запускає копію програмного забезпечення і, отже, здатна обслуговувати запити. Це лише одна з різних функцій, які може виконувати розподілювач навантаження. Ще однією дуже поширеною відповідальністю є "перевірка працездатності", коли розподілювач навантаження використовує протокол "ping-echo" або обмінюється повідомленнями про стан з усіма серверами, щоб переконатися, що вони працюють і працюють нормально.

Розподілювач навантаження розподіляє навантаження, підтримуючи стан кожної машини - скільки запитів обслуговує кожна машина, яка

машина перебуває в режимі очікування, яка машина перевантажена запитами в черзі тощо. Отже, алгоритм балансування навантаження розглядає такі речі перед перенаправленням запит на відповідну машину сервера. Він також враховує накладні витрати на мережу і може обрати сервер у найближчому центрі обробки даних за умови, що він доступний для обслуговування запитів.

Запит-відповідь також можна зробити двома різними способами:

- Розподілювач навантаження завжди виступає посередницькою програмою для кожної відповіді. У цьому випадку, після того, як розподілювач навантаження передає запит серверу, будь-яка відповідь від сервера користувачеві пройде через розподілювач навантаження. Отже, серверні машини, які насправді обслуговують запит, ніколи не будуть безпосередньо взаємодіяти з користувацькою машиною, на якій запущено клієнтську програму. Машина, на якій розміщена програма балансування навантаження, буде обробляти всі запити / відповіді до користувача та від нього.
- Розподілювач навантаження не виступає посередником для відповідей, що надходять із серверної машини - в цьому випадку, коли сервер отримає запит від розподілювача навантаження, він обходить розподілювач навантаження і передає свої відповіді безпосередньо клієнту.

Налаштування кластера та балансу навантаження як інтерфейсу для клієнтської програми насправді не завершує масштабовану архітектуру та дизайн [1]. Є ще багато критичних питань, на які потрібно відповісти, і прийняти ряд ключових проектних рішень, які вплинуть на загальні властивості нашої системи.

1.4 Вертикальне масштабування

Вертикальне масштабування стосується додавання ресурсів (CPU / RAM / DISK) на сервер (сервер бази даних або додатків все ще залишається одним) за запитом.

Вертикальне масштабування найчастіше використовується в додатках та продуктах середнього, а також малого бізнесу. Одним з найпоширеніших прикладів віртуального вертикального масштабування є придбання дорогого обладнання та використання його як гіпервізора віртуальної машини (VMWare ESX).

Вертикальне масштабування зазвичай означає оновлення серверного обладнання. Деякі з причин вертикального масштабування включають збільшення IOPS, збільшення ємності процесора / оперативної пам'яті, а також ємності диска.

Однак навіть після використання віртуалізації, яка націлена на покращення продуктивності, ризик простоїв з нею набагато вищий, ніж використання горизонтального масштабування [2].

1.5 Відмінності Вертикального та Горизонтального масштабування

Таблиця 1.1 Відмінності масштабування

	Масштабування по горизонталі	Вертикальне масштабування
Бази даних	У світі баз даних горизонтальне масштабування, як правило, базується на розділенні даних (кожен вузол містить лише частину даних).	При вертикальному масштабуванні дані живуть на одному вузлу, і масштабування здійснюється за допомогою розподілу навантаження між ресурсами центрального процесора та оперативної пам'яті машини.
Час простою	Теоретично, додавання більшої кількості машин до існуючого пулу означає, що не існує обмеження ємності одного блоку, що робить можливим масштабування із меншим простоєм.	Вертикальне масштабування обмежується ємністю однієї машини, масштабування, що перевищує цю потужність, може спричиняти простої та має верхню жорстку межу, тобто масштаб обладнання, на якому ви працюєте.

Продовження таблиці 1.1

	Масштабування по горизонталі	Вертикальне масштабування
Паралельність	Також це описується як розподілене програмування, оскільки воно передбачає розподіл завдань між машинами по мережі. Кілька моделей, пов'язаних з цією моделлю: Master/Worker*, Tuple Spaces, Blackboard, MapReduce.	Акторна модель: одночасне програмування на багатоядерних машинах часто виконується за допомогою багатопоточності та в процесі передачі повідомлень.
Обмін даними	У розподілених обчисленнях відсутність спільного адресного простору ускладнює обмін даними. Це також робить процес обміну, передачі або оновлення даних дорожчим, оскільки вам потрібно передавати копії даних.	У багатопоточному сценарії ви можете припустити існування спільного адресного простору, тому обмін даними та передача повідомлень можна здійснити, передавши посилання.

Отже: плавний перехід між двома моделями?

Не завжди має сенс вибирати між горизонтальним та вертикальним масштабуванням. Переміщення між двома моделями часто є кращим вибором[3]. Наприклад, у сховищі ми часто хочемо перемикатися між одним локальним диском та розподіленою системою зберігання.

Вбудовування гнучкості в систему, де деякі рівні програми працюють на машинах з вертикальним масштабом, а інші шари на горизонтально масштабованій інфраструктурі залишається предметом проектування для розпаралелювання. Щоб досягти цього, спроектуємо його з самого початку як відокремлений набір послуг, що полегшує переміщення коду, тобто можливо додавати більше ресурсів за потреби, не порушуючи зв'язків між наборами кодів; і розділимо вашу програму та модель даних, щоб паралельні блоки нічого не поділяли [10].

Цілком ймовірно, що галузь все частіше буде переходити до горизонтально розподіленого підходу до масштабування архітектури. Ця тенденція обумовлена попитом на більшу надійність через стратегію надмірності та потребою в покращеному використанні за рахунок спільного використання ресурсів в результаті переходу до середовищ хмар / SaaS. Однак поєднання цього з підходом вертикального масштабування може дозволити нам скористатися обома парадигмами.

1.6 Мікросервіси

Wolff Eberhard визначає мікросервіси як [7]. "... стиль архітектури програмного забезпечення, в якому складні додатки складаються з невеликих незалежних процесів, що взаємодіють між собою за допомогою мовних агностичних API. Ці сервіси невеликі, сильно роз'єднані та зосереджені на виконанні невеликого завдання, полегшуючи модульний підхід до побудови системи ".

Al Hilwa, керівник програми з розробки програмного забезпечення в IDC, описує це так [6]: "Мікросервіси - це архітектурний підхід, який спирається на довгий досвід, що розвивається у розробці програмного забезпечення та проектуванні систем, включаючи зусилля SOA за останні два десятиліття. Архітектура мікровсервісів є уможливленням різноманітних категорій інструментів, але в першу чергу це архітектурний

підхід до проектування системи, який також вимагає значних організаційних та культурних налаштувань для успішного виконання ".

Лаконічніше, мікросервіси - це загальний термін, який застосовується до розбиття IT-систем та додатків до менших, більш детальних елементів. Контейнери переносять програми та послуги на самостійний, компонентний рівень, а DevOps забезпечує основу для IT-інфраструктури та автоматизації для розробки, розгортання та управління навколишнім середовищем.

Jason Bloomberg, президент Intellyx, розповідає про різницю між типовим веб-сервісом та мікросервісом[5], аргументуючи проти тенденції намагатися просто ребрендингувати веб-сервіси як мікросервіси. "Мікросервіс, на відміну від цього, є дбайливим, цілісним блоком виконання. Це, безумовно, не сам програмний інтерфейс, хоча він, очевидно, такий має. Натомість, в основі мікросервісу лежить сам запущений код. Мікросервіси також містять власні середовища виконання , тому їм не потрібно працювати на ESB ".

Мікросервіси приносять макровигоди. Al Hilwa зазначає: "Мікросервіси, як правило, розробляються на основі сучасних еластичних та часто бездержавних архітектурах, але це не означає, що вони автоматично масштабуються. Архітектори повинні бути особливо обережними, щоб забезпечити масштабованість централізованих служб або баз даних. Мікросервіси також чинять великий тиск на API, підкреслюючи важливість сильної технології управління API у застосованому стеку програмного забезпечення "[6].

Сама природа контейнерів робить їх за своєю суттю більш масштабованими, ніж традиційна IT-інфраструктура - особливо інфраструктура фізичних серверів у локальному центрі обробки даних. Основа мікросервісів побудована на хмарі та віртуалізації, що включає масштабованість як основну функцію. Віртуалізація дозволяє створювати

та надавати сервери одним натисканням кнопки у міру стрибків попиту, а також просто вимикати та видаляти, коли потреба стихає.

Те саме стосується контейнерів. Контейнери - це самостійні компоненти, які можна легко дублювати та розширювати у міру збільшення попиту. Нові екземпляри контейнерів можна створювати автоматично або програмно для масштабування відповідно до попиту.

Jason Bloomberg застерігає: "Мікросервіси не потребують контейнерів (або навпаки), але їх легко закріпити за дизайном. Крім того, якщо ви реалізуєте контейнери, важко і, як правило, нерозумно розміщувати в них будь-який новий виконуваний код, крім мікросервісів " [5].

Мікросервіси, контейнери за своєю суттю є більш масштабованими, ніж застаріла IT-інфраструктура та моделі розробки додатків, але вирішення проблем все ще існує. Ми все ще маємо подібні занепокоєння щодо масштабованості, але підхід до масштабування відрізняється, коли ми маємо справу з мікросервісами.

1.7 Кубічна модель масштабованості

Martin Abbot та Fisher Michael описують масштабованість за допомогою кубічної моделі [4], "Куб масштабу" складається з осі X, осі Y та осі Z. Традиційним методом масштабування шляхом запуску кількох копій програми, збалансованої навантаженням на серверах, є вісь X.

Загальний підхід мікросервісів знаходиться вздовж осі Y. Масштабування за віссю Y розбиває додаток на його компоненти та служби. Архітектор програмного забезпечення Chris Richardson пояснив цей метод у дописі в блозі: "Кожний сервіс відповідає за одну або декілька тісно пов'язаних між собою функцій. Існує кілька різних способів розкладання програми на сервіси. Один із підходів полягає у використанні декомпозиції на основі дієслів та визначенні сервісів, що реалізують

одноразовий випадок використання, наприклад, оплата. Інший варіант - розкласти додаток за допомогою іменника та створити сервіси, відповідальні за всі операції, пов’язані з певною сутністю, наприклад, управління клієнтами. Додаток може використовувати комбінацію дієслівних та іменникових на основі розкладання "[4].

Це залишає вісь Z. Вісь X - це традиційне масштабування балансу навантаження, а вісь Y охоплює мікросервіси. Вісь Z застосовує подібний підхід до осі X - запускає однакові копії коду на декількох серверах. Що робить масштабування осі Z унікальним, так це те, що він також запозичує від осі Y, тому кожен сервер відповідає лише за підмножину програми, а не за програму в цілому[4].

1.8 Масштабування мікросервісів

Тепер давайте складемо це все разом. Попит і потреба у масштабованості з мікросервісами все одно будуть, як і у традиційної IT-інфраструктури. Оскільки сама програма розбита на більш дрібні компоненти, які можуть розповсюджуватися на окремих серверах, нам слід підійти до масштабованості дещо інакше.

Незважаючи на переваги мікросервісів, це стосується додаткової складності, коли мова заходить про масштабованість. Замість того, щоб мати справу з одним додатком, що працює на одному сервері - або з збалансованою навантаженням системою на кількох серверах - можуть бути елементи програми, написані різними мовами програмування, завантажені на різному обладнанні, що працюють на різних гіпервізорах віртуалізації та розгорнуті через різні хмарні та локальні локації. Коли попит на програму зростає, усі основні компоненти повинні бути скоординовані за масштабом, або повинні мати можливість визначити, які окремі елементи потрібно масштабувати, щоб задовольнити сплеск попиту.

Використання підходу масштабування по осі Z від кубічної моделі дозволяє розділити дані між різними серверами на основі критеріїв маршрутизації. Ви можете направляти запити на основі первинного ключа даних, до яких ви отримуєте доступ, або на основі типу клієнта - надсилання платних або преміум-клієнтів на сервери з більшою пропускною здатністю та пропускною здатністю для забезпечення кращої продуктивності.

Завдяки застарілому підходу до IT-інфраструктури та розгортанню додатків, весь додаток мав розглядатися як монолітна сутність. Якщо попит зростав, потрібно було помножити весь додаток, щоб прийняти навантаження, що означало примноження серверів або екземплярів віртуального сервера, на яких працювала програма.

З мікросервісами все є більш детальним, включаючи масштабованість та управління стрибками попиту. Попит може збільшитися на один компонент програми або певну підмножину даних, а архітектура мікросервісів дозволяє масштабувати лише ті компоненти програми, на які це впливає, а не всю програму та базову інфраструктуру.

Незалежно від того, як розглядаються проблеми масштабованості мікросервісів, з точки зору замовника чи кінцевого користувача, важливою є продуктивність самого додатка. З цієї точки зору має сенс використовувати якийсь проксі-сервер програми або контролер доставки додатків (ADC) як посередника для виявлення проблем із продуктивністю та полегшення автоматизації масштабування, коли це необхідно.

Традиційні ADC підходять до проблеми з точки зору "один на один". Вони оптимізовані для управління програмами, які існують на одній апаратній платформі, що працює в одному місці. Мікросервіси змінюють це співвідношення так, що для цього потрібен ADC, який обізнаний про мікросервіси.

1.9 Проблеми з базами даних мікросервісів

Монолітний додаток взаємодіє з єдиною базою даних. Дані обмінюються між усіма компонентами програми. На відміну від цього, у програмі з мікросервісною архітектурою право власності на дані децентралізоване. Кожний сервіс є автономним і має власне сховище даних, що відповідає його функціональності. Це означає, що один сервіс не може змінювати будь-які дані, що зберігаються в базі даних іншого сервісу. І тут починаються проблеми.

Незалежно від того, який додаток створюється, його мікросервіси повинні взаємодіяти та обмінюватися даними. Оскільки, якщо цього не сталося, існує ризик мати проблеми узгодженості, такі як дублювання даних. Проблема полягає в тому, що в мікросервісах не можна використовувати підхід ACID для транзакцій за межами однієї служби. Наявність приватних баз даних замість однієї спільної бази ускладнює реалізацію запитів та транзакцій, що охоплюють кілька сервісів.

1.10 Загальні моделі даних

Розглянемо колекцію шаблонів, пов’язаних з управлінням даними мікросервісів.

1.10.1 Шаблон бази даних на сервіс

Основною характеристикою архітектури мікросервісів є вільне поєднання функціоналу. Для цього кожен сервіс повинен мати власне сховище даних. Отже, побудова архітектури бази даних для мікросервісів майже завжди вимагає дотримання схеми бази даних на сервіс. Принаймні до того моменту, поки додаток ще не стане надто складним, із залученням численних сервісів.

Розглянемо додаток Інтернет-магазину (рисунок 1.1). Служба замовлення та обслуговування клієнтів зберігають дані у власних базах даних. Зміни в одній базі даних не впливають на інші мікросервіси[8].

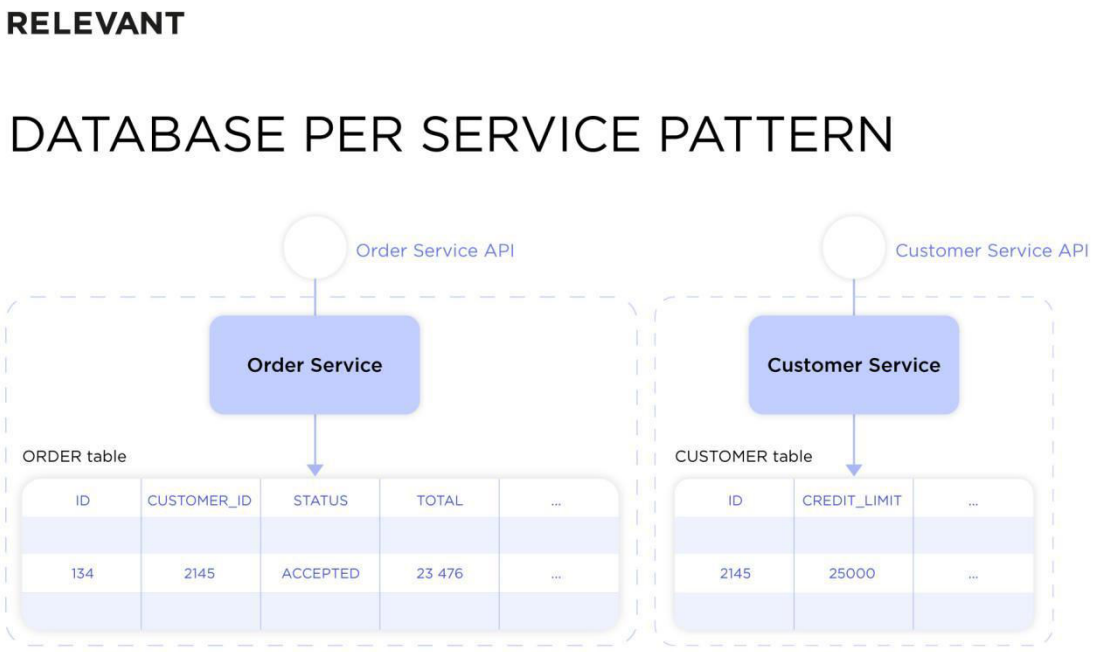

Рисунок 1.1 - Схема шаблону Database-per-Service [8]

Інші мікросервіси не можуть отримати безпосередній доступ до бази даних служби. Доступ до постійних даних кожної служби можна отримати лише через API.

1.10.2 Шаблон Saga

Використання шаблону бази даних на сервіс створює необхідність у прийнятті шаблону Saga, оскільки саги - це спосіб здійснення транзакцій, які охоплюють сервіси та підтримують узгодженість даних. У мікросервісах замість використання традиційних розподілених транзакцій (на основі XA/2PC) доведеться використовувати послідовність локальних транзакцій (вони ж Saga)[8].

Ось як це працює: одна локальна транзакція оновлює базу даних, а потім запускає наступну транзакцію за допомогою обміну повідомленнями (рисунок 1.2).

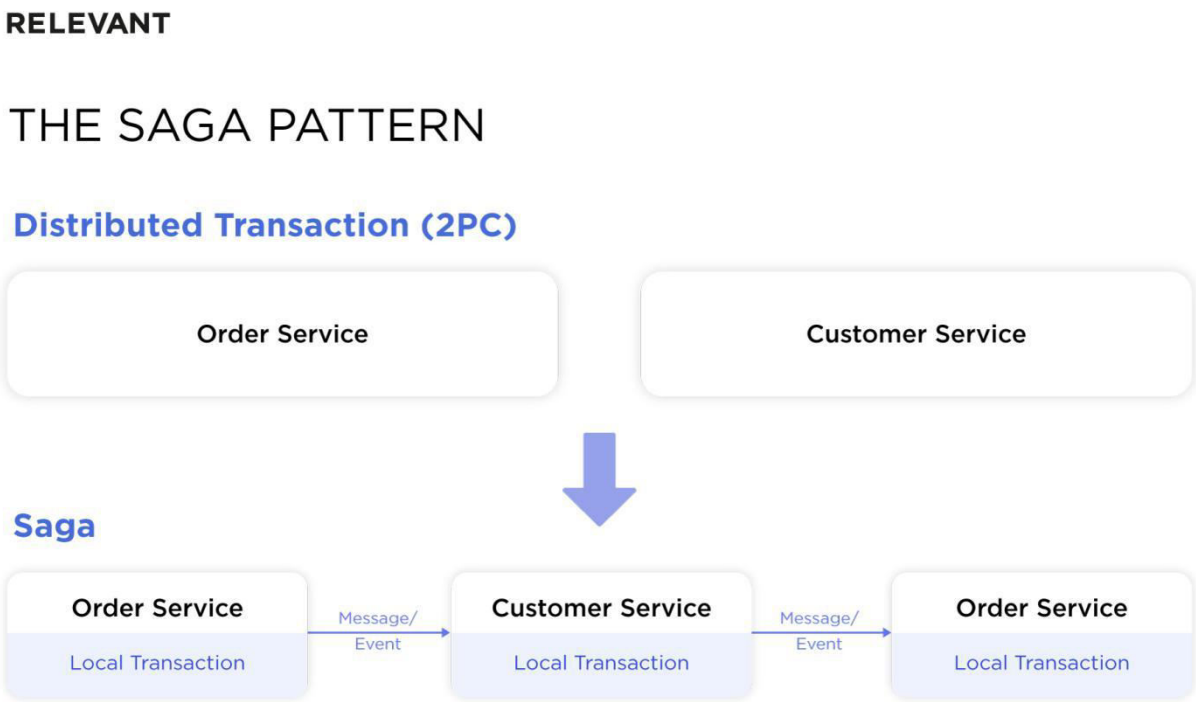

Рисунок - 1.2 Схема шаблону Saga [8]

Існує два різних підходи до координації саг:

- choreography - коли обмін подіями відбувається без контрольних точок
- orchestration - коли у вас є централізовані контролери.

1.10.3 Шаблон API-композиції

Тепер давайте заглибимося в те, як можемо отримувати дані, які розкидані між різними сервісами. Для цього, очевидно, не можна використовувати традиційний механізм розподілених запитів. Але замість цього можна використовувати шаблон API композиції. Він реалізує запит,

викликаючи певні служби, що володіють даними, та комбінує результати. Нижче можете побачити, як реалізована операція запиту[8]. API Composer отримує дані з трьох служб провайдера(рисунок 1.3).

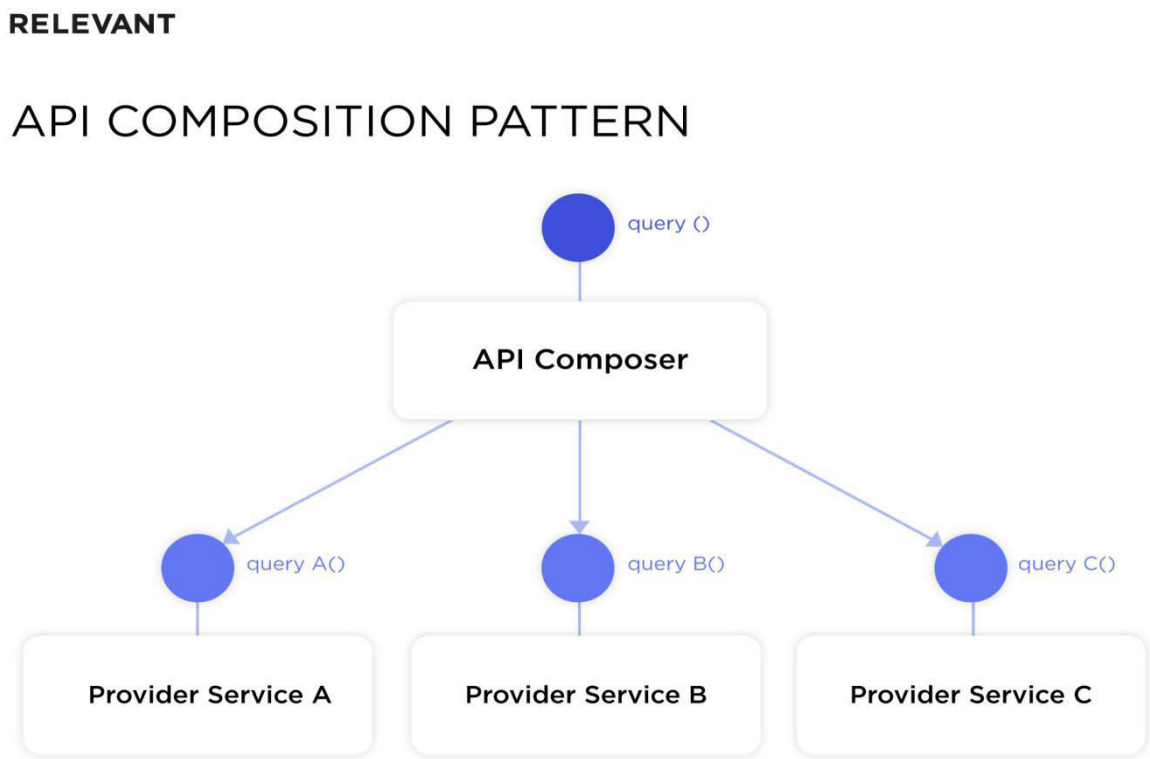

Рисунок 1.3 - Схема шаблону API Composition [8]

1.10.4 Шаблон CQRS

The command query responsibility segregation (CQRS) - це ще одна модель запитів, яку можна використовувати. Справа в тому, що API Composition має деякі обмеження: його не можна використовувати для складних запитів, оскільки можливо отримати неефективні об’єднання в пам’яті. Крім того, потрібно уникнути перевантаження сервісів.

Саме в такому випадку потрібен CQRS. Він реалізує запити з використанням view databases. CQRS відокремлює команди від запитів. Модулі на стороні запитів реалізують запити та підтримують

синхронізацію моделі даних із моделлю даних на стороні команди. Що стосується модулів командної сторони, вони керують операціями[8].

Можна застосувати CQRS у межах сервісу та використовувати її для визначення сервісних запитів. API служби запитів складається лише з операцій запиту. Він запитує базу даних та постійно її оновлює, підписуючись на події, опубліковані сервісами, що володіють даними (рисунок 1.4).

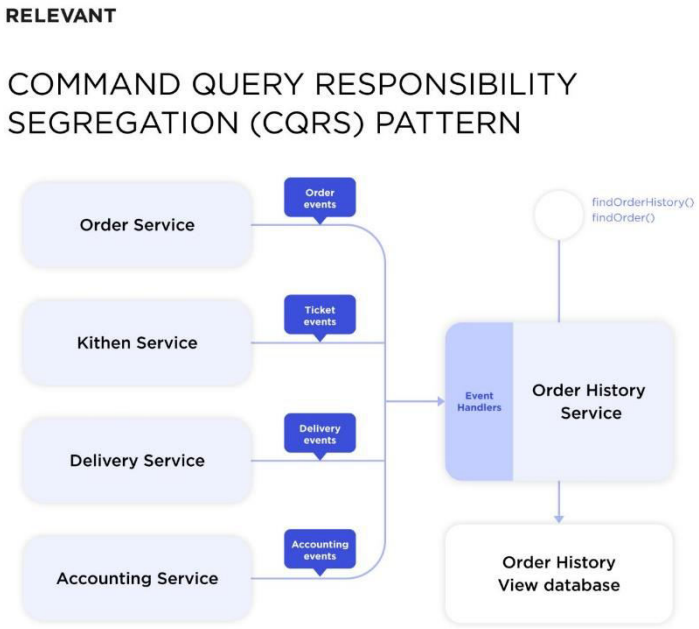

Рисунок 1.4 - Схема шаблону CQRS [8]

1.11 Результати

Отже, можна зробити висновок що не завжди має сенс вибирати між горизонтальним та вертикальним масштабуванням. Переміщення між двома моделями часто є кращим вибором. Незважаючи на переваги мікросервісів, починаються проблеми коли мова заходить про масштабованість. Існує декілька прийнятих шаблонів для вирішення проблем з управління даними в застосунках на мікросервісній архітектурі,

на мою думку, сьогодні, шаблон CQRS - є найбільш корисним, незважаючи на його порівняну складність.

РОЗДІЛ 2.
ПРОЕКТУВАННЯ ТА РОЗРОБКА БАЗОВОГО ЗАСТОСУНКУ

Додатки що базуються на мікросервісній архітектурі містять у собі чималу кількість малих, незалежних сервісів, які інтегровані між собою щоб забезпечувати бажаний функціонал. Хоч і сервіси за складністю та об'ємом порівняно з класичними додатками набагато простіші, існує значна складність яка виникає внаслідок їх внутрішньої взаємодії для організації бажаних операцій. На практиці мікросервісна архітектура працює чудово, але з масштабуванням зростає і складність.

2.1 Проектування

У нашій системі кожен невеликий сервіс - це додаток Spring Boot. Кожний сервіс буде упакований в вигляді файлу jar і буде використовувати вбудований Tomcat в якості цільового середовища виконання для обслуговування.

Ми будемо розробляти додаток, який буде складатися з наступних компонентів:

- OAuth Server
- User service
- API Gateway
- Eureka server
- Software Circuit Breaker with Hystrix
- Monitoring service
- Ribbon Load Balance server
- Configuration server

- Log Management
- Application Management
- Infrastructure Management

Структура застосунку та взаємодія між компонентами зображена на рисунку 2.1.

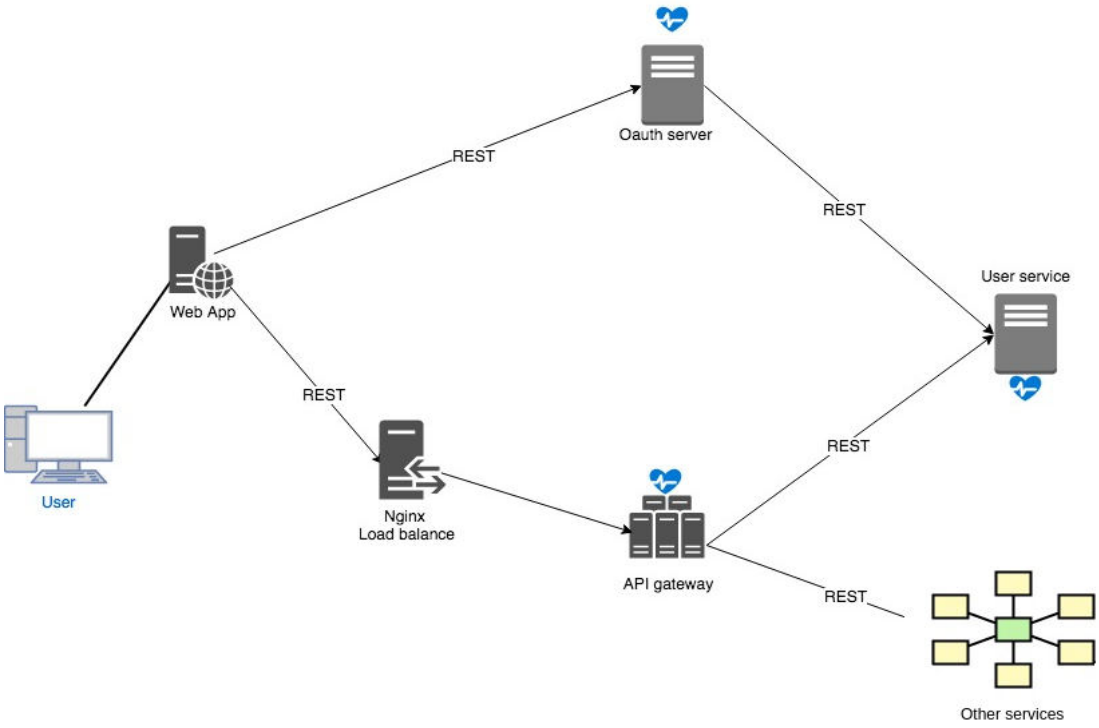

Рисунок 2.1 - Структура мікросервісного застосунку

На рис 2.1 під компонентом Web App розуміється портал, який буде використовувати потенційні мікросервіси які можуть бути розроблені використовуючи даний застосунок як основу.

2.2 Сервіс обслуговування користувачів

В базовому рішенні сервіс обслуговування користувачів має виконувати функцію сервісу, який працює з даними користувача. Для цього було створену сутність акаунту(рисунок 2.2).

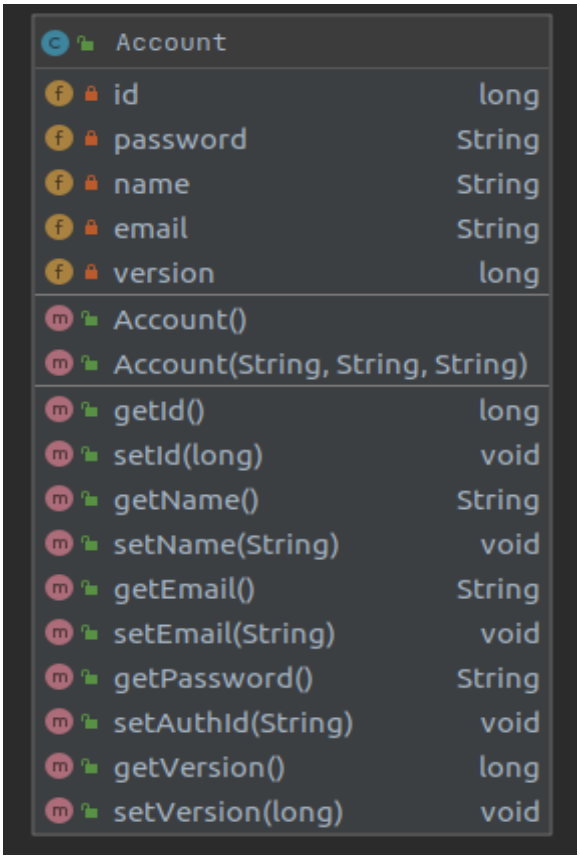

Рисунок 2.2 - Сутність account

На цьому етапі сутність має тільки базову інформацію про користувача але вона може бути легко розширена, наприклад для зберігання вподобань акаунту та інших акаунт-специфічних даних.

Для спрощення розробленого додатку була використана база даних H2 оскільки в демо версії застосунку не має потреби зберігати велику кількість даних. База даних може бути з легкістю замінена на будь яку іншу, її можна розгорнути всередині контейнера або використовувати

віддалену базу, для цього достатньо змінити конфігурацію а мікросервіс сам створить схему та завантажить необхідні початкові дані.

Сервіс обслуговування користувачів буде використовуватись сервісом OAuth для отримання деталей аутентифікації користувача, а також використовуватись сервісом API Gateway для отримання інформації про користувача.

Для цього було створено rest controller (рисунок 2.3).

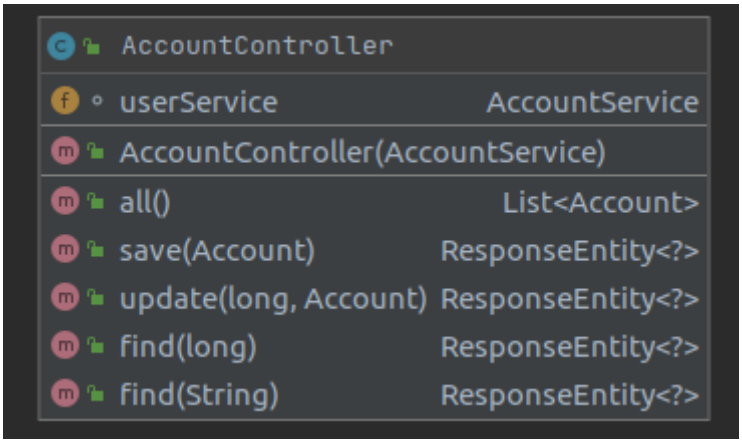

Рисунок 2.3 - Account controller

Оскільки сервіс обслуговування користувачів потенційно може використовуватись на порталі,то було передбачено доступ до сервісу ззовні, але тільки з використанням JWT токену.

2.3 Сервер Eureka

Для реалізації менеджменту застосунку будемо використовувати Eureka Server - це програма, яка містить інформацію про всі клієнтські сервіси (рисунок 2.4). Кожен мікросервіс реєструється на сервері Eureka, і Eureka знає всі клієнтські додатки, що працюють на кожному порту і їх IP-адресу. Eureka Server також відомий як Discovery Server[11].

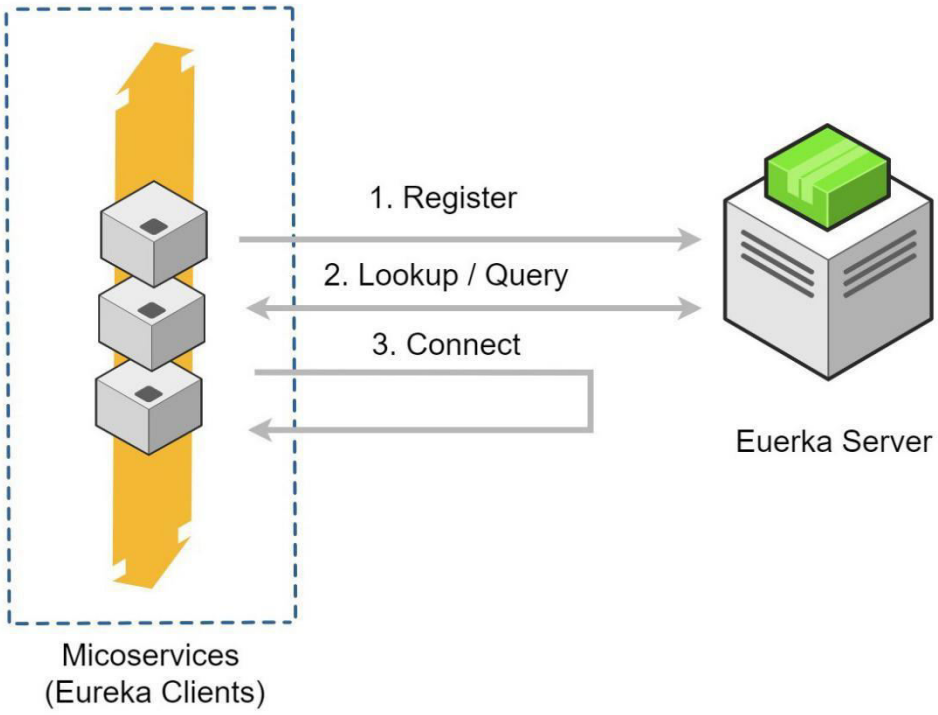

Рисунок 2.4 - Взаємодія з Eureka сервером [17]

Eureka чимось нагадує платформу SOA, тому що існує безліч сервісів. Коли якийсь інший сервіс хоче спілкуватися з уже зареєстрованою службою, вони запитують у сервера Eureka про базову URL-адресу цієї послуги. Кілька екземплярів одного і того ж сервісу можуть бути зареєстровані в Eureka, в цьому випадку Eureka може допомогти у виконанні балансування навантаження.

2.4 Автоматичний переривач

Мікросервісний застосунок повинен будуватись таким чином, щоб він був спроможний витримувати відмову деяких частин, тобто деяких мікросервісів. Щоб система продовжувала роботу при відмові якогось з мікросервісів Netflix представила концепцію автоматичного переривача (circuit breaker) (рисунок 2.5).

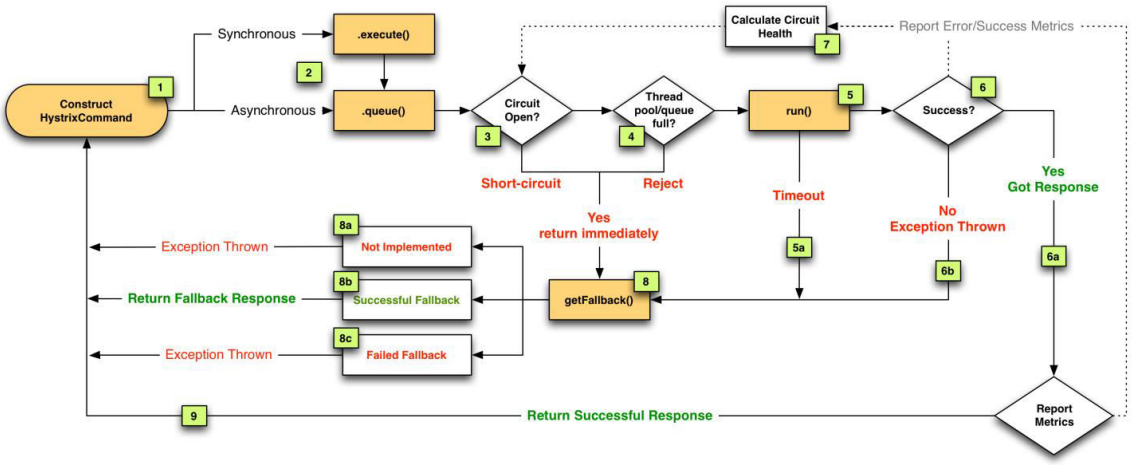

Рисунок 2.5 - Обробка командою hystrix [16]

Circuit Breaker - забезпечує альтернативну поведінку, якщо певний мікросервіс не працює. Таким чином система буде працювати в альтернативному режимі доки сервіс не відновиться, тобто система в цілому буде працювати, що дає можливість запобігти хвильового ефекту відмови в обробці. В даному застосунку буде використано Hystrix.

2.5 Клієнт для балансування навантаження

Ribbon є клієнтом для балансування навантаження, який призначений для роботи з Eureka сервером. Ribbon обмінюючись повідомленням з сервером Eureka використовує його для отримання базової URL-адреси одного з розглянутих примірників мікросервісів.

Ми використовуємо Netflix Foreign для декларативного REST клієнта та інтеграції Ribbon з Eureka для надання HTTP-клієнта з балансировкою навантажень[14].

2.6 Сервіс конфігурації

У традиційному додатку конфігураційні файли прив'язані до коду або статично упаковані, тому будь-які зміни конфігурації означають перебудову і повторне розгортання додатка, що є порушенням принципу мікросервісів.

За допомогою мікросервісів ми створюємо центральний сервер для конфігурації, де всі параметри мікросервісів записуються, контролюються версіями і управляються централізовано. Перевага центрального сервера конфігурацій полягає в тому, що якщо ми змінимо властивості для мікросервіса, він може відобразити це на льоту без повторного розгортання мікросервіса(рисунок 2.6).

Spring Cloud Config - це клієнт-серверний підхід Spring для зберігання та обслуговування розподілених конфігурацій у кількох додатках та середовищах. Хоча він дуже добре підходить для програм Spring, використовуючи всі підтримувані формати файлів конфігурації разом із такими конструкціями, як Environment, PropertySource або @Value, його можна використовувати в будь-якому середовищі, на якому працює будь-яка мова програмування[9].

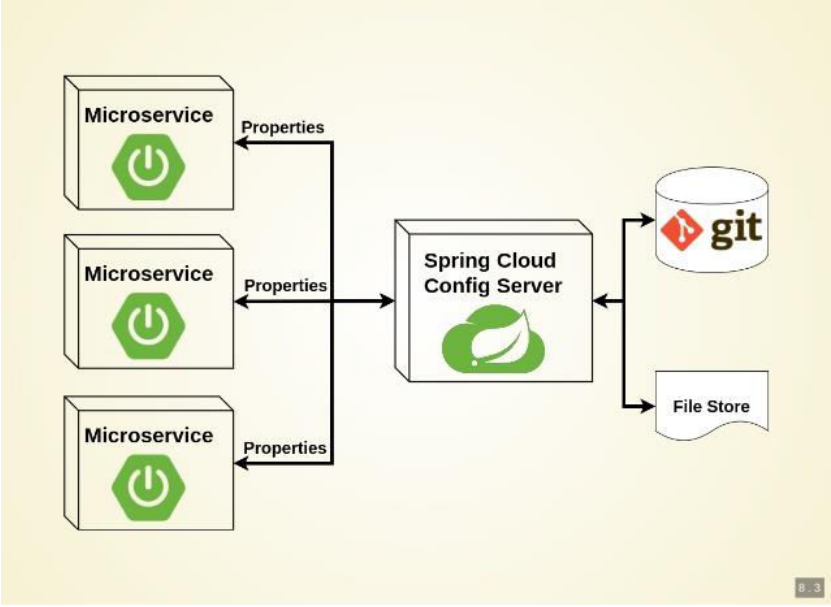

Рисунок 2.6 - Структура конфігураційного серверу [15]

За замовчуванням наш конфігураційний сервер не використовує Git, ми поміщаємо наші властивості в локальну файлову структуру

2.7 OAuth сервіс

Коли ми розробляємо будь-яку систему, безпека дуже важлива, і в ситуації з мікросервісами архітектура не відрізняється. OAuth2 - дуже гарне рішення, це добре відома технологія авторизації, вона широко використовується в Google, Facebook та GitHub для їх API [12]. Неможливо говорити про безпеку і не згадувати Spring Security. В нашому проекті ми використовуємо її разом з OAuth2.

Spring Security та OAuth2 є очевидним вибором, коли мова йде про захищені розподілені системи. Ми додаємо ще один елемент до нашої системи безпеки: JSON Web Token. Використовуючи тільки OAuth нам буде потрібен сервер для аутентифікації користувача та генерації токену, а також ендпоінт для клієнтів, щоб перевіряти чи дійсний токен і які дозволи він має, потребуючи тим самим вдвічі більше запитів всередині системи. JWT надає простий спосіб передачі дозволів і даних користувачів, і коли всі дані містяться в токені сервісам не потрібно запитувати у сервера чи валідний токен та які користувацькі дані він представляє. Вся інформація спочатку серіалізуются в JSON, кодується за допомогою base64 і, нарешті, підписується закритим ключем RSA. Передбачається, що всі сервіси матимуть відкритий ключ, щоб перевірити, чи був токен підписаний для правильного закритого ключа, і десеріалізувати токен для отримання інформації.

2.8 Управління аудитом

Наявність добре спланованих стратегій ведення журналів, моніторингу та аналітики - ключ до цього типу проектів. Їх слід впроваджувати з самого початку проекту, щоб збільшити швидкість розробки і тестування, а також забезпечити швидке усунення неполадок. Таку важливу частину світу мікросервісів не можна було виключити з нашого проекту, і тут ми вирішили її за допомогою рішення ELK, де ми налаштували агент в кожному контейнері, відповідальний за відправку журналів на сервер управління журналами, де він буде зберігати всі журнали з наших компонентів мікросервісів і забезпечувати відмінний користувальницький інтерфейс для пошуку і створення звітів на основі відправлених даних.

2.9 Менеджмент застосунку

Для управління системою, що складається з декількох мікросервісів, необхідно зібрати всю необхідну інформацію в одному централізованому місці. Це відноситься до журналів і відомостей про стан всіх примірників додатків, які зараз працюють в нашому кластері мікросервісів. Ми використовуємо Spring Boot Admin, щоб допомогти нам керувати нашими додатками Java. Це просте рішення, створене для управління і моніторингу додатків Spring Boot, яке отримує дані з кінцевих точок за допомогою Actuator-а і надає інформацію про всі зареєстровані додатки на єдиній панелі інструментів.

2.10 Менеджмент інфраструктури

Мікросервіси добре поєднуються з контейнерними технологіями і часто працюють разом. Контейнери часто є кращим вибором, оскільки

вони автономні і швидко ініціалізуються або клонуються [13]. У нашому проекті всі компоненти динамічно управляються за допомогою Docker.

Маючи окремі служби, ми керуватимемо інфраструктурою для кожної служби. Інфраструктура як код (IaC) народилася як рішення цієї проблеми. Все, що потрібно нашому додатку, буде описано в файлі (Dockerfile). Поряд з файлом docker-compose ми організуємо додаток з декількома контейнерами, і вся конфігурація сервісу буде версіонуватися, що спростить процес створення та розгортання всього проекту.

2.11 Результати

Отже, можна зробити висновок, що сьогодні існує велика кількість готових рішень для певного функціоналу на мікросервісній архітектурі, а також що самі сервіси по своїй суті не складні та не об’ємні, що допомагає легше розуміти процеси, що відбуваються всередині них, але коли мова заходить про взаємодію всередині застосунку між сервісами, складність виростає в рази, і на сьогоднішній день не існує очевидного рішення цієї проблеми.

РОЗДІЛ 3. ОСОБЛИВОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ

Наш застосунок заснований на концепціях розподілених систем і намагається запропонувати рішення для загальних проблем.

3.1 Конфігурація мікросервісів

В нашому додатку ми використовуємо Spring Cloud Config Server, який є окремим застосунком, для якого застосовано анотацію @EnableConfigServer, який маркує застосунок. Він має набір ендпоінтів з якими можна взаємодіяти, наприклад отримати поточну конфігурацію яка задана для кластера. Наприклад, конфігурація для сервісу адміністрування (рисунок 3.1) :

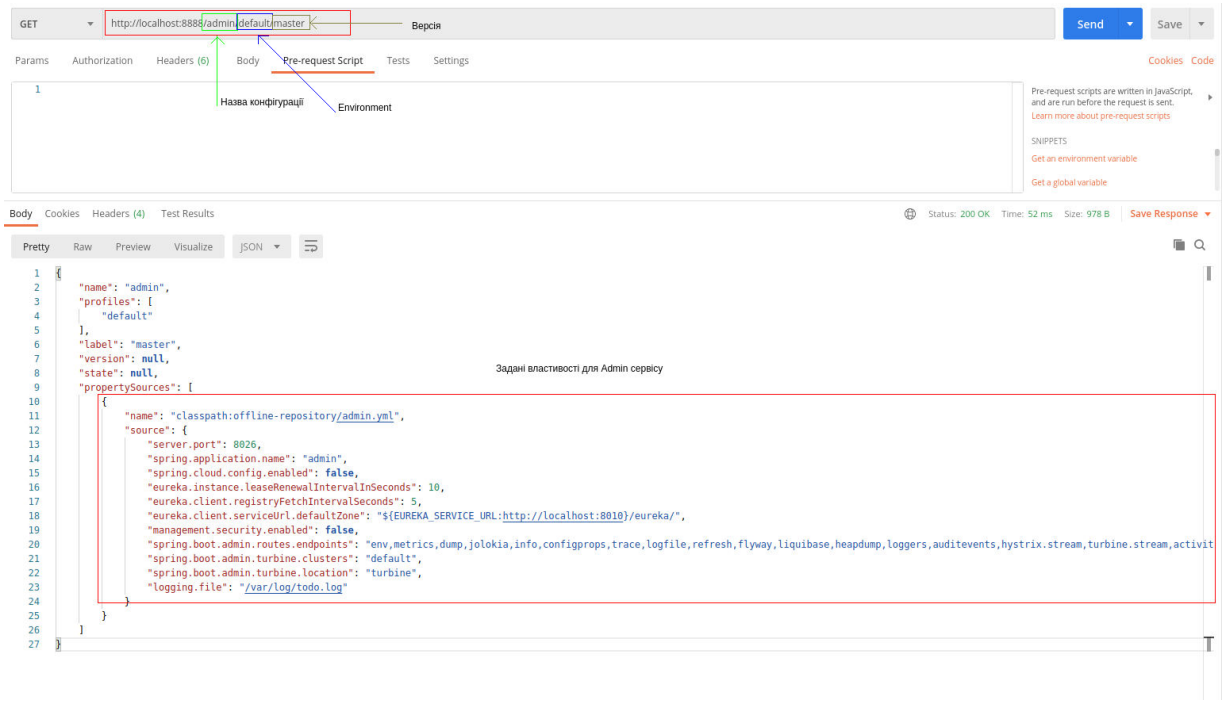

Рисунок 3.1 - Приклад роботи конфігураційного сервісу

Для конфігурацій існує паттерн для відображення конфігурацій, URL його виглядає наступним чином: http://xxx.xx.xx:8888/{appName}/{environemnt} - де appName власне назва сервісу, для якого запитуємо властивості, environment - це середовище, для якого йде запит. Використання environment в даному випадку не є раціональним,оскільки на даному етапі розробки не планується використовувати віддалені або різні середовища для розгортання застосунку. Але потенційно це дуже корисна властивість нашої системи. Усі властивості зберігаються локально в файлах формату *.yml або *.properties. Структура сервісу зображена на рисунку 3.2.

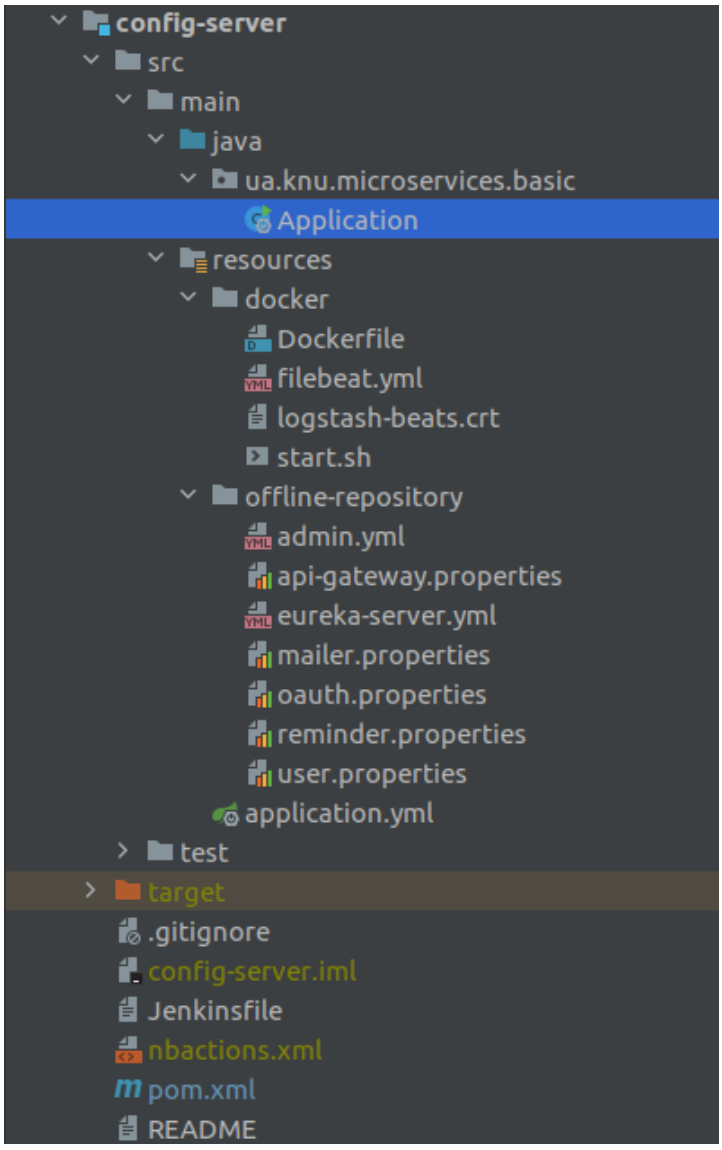

Рисунок 3.2 - Структура config серверу

3.2 Авторизація та аутентифікація

Механізм авторизації та аутентифікації використовує OAuth сервіс, який в свою чергу пов'язаний з сервісом обслуговування користувачів. Оскільки ми одночасно використовуємо Spring Security та Oauth2, то в нашій системі можливі дві каскадні авторизації. Перша за допомогою Spring Security - використовується для авторизації юзера. Для цього використовуємо базову аутентифікацію(рисунок 3.3).

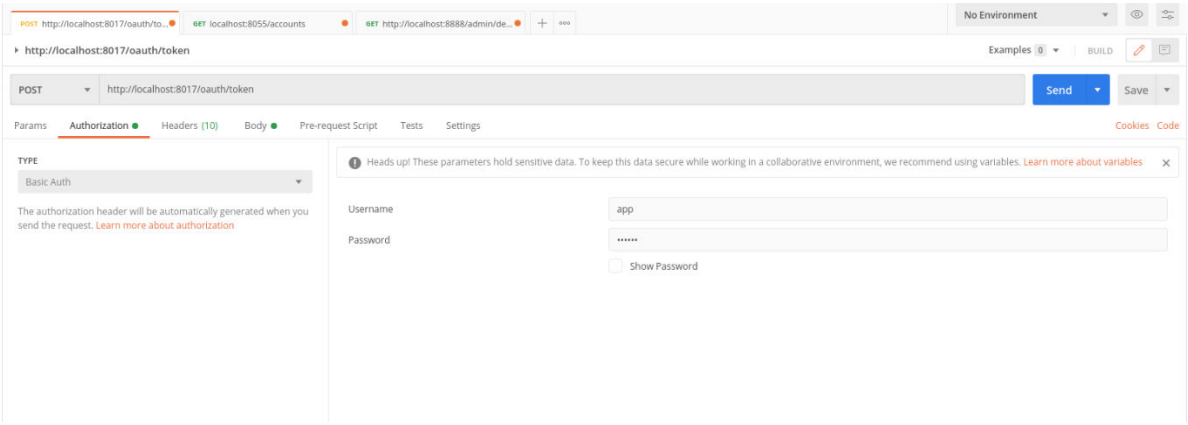

Рисунок 3.3 - Аутентифікація за допомогою Spring Security.

У нашій системі користувач потенційно може мати декілька акаунтів з різними дозволами на використання тих чи інших сервісів. Для аутентифікації всередині використовується Oauth2(рисунок 3.4). Він має стандартний функціонал, та на запит http:/xxx.xx.xx:8017/oauth/token - повертає JWT токен підписаний закритим ключем. Саме цей доступ використовується для взаємодії всередині між сервісами, але водночас цей токен не доступний ззовні, оскільки для доступу ззовні використовується Spring Security.

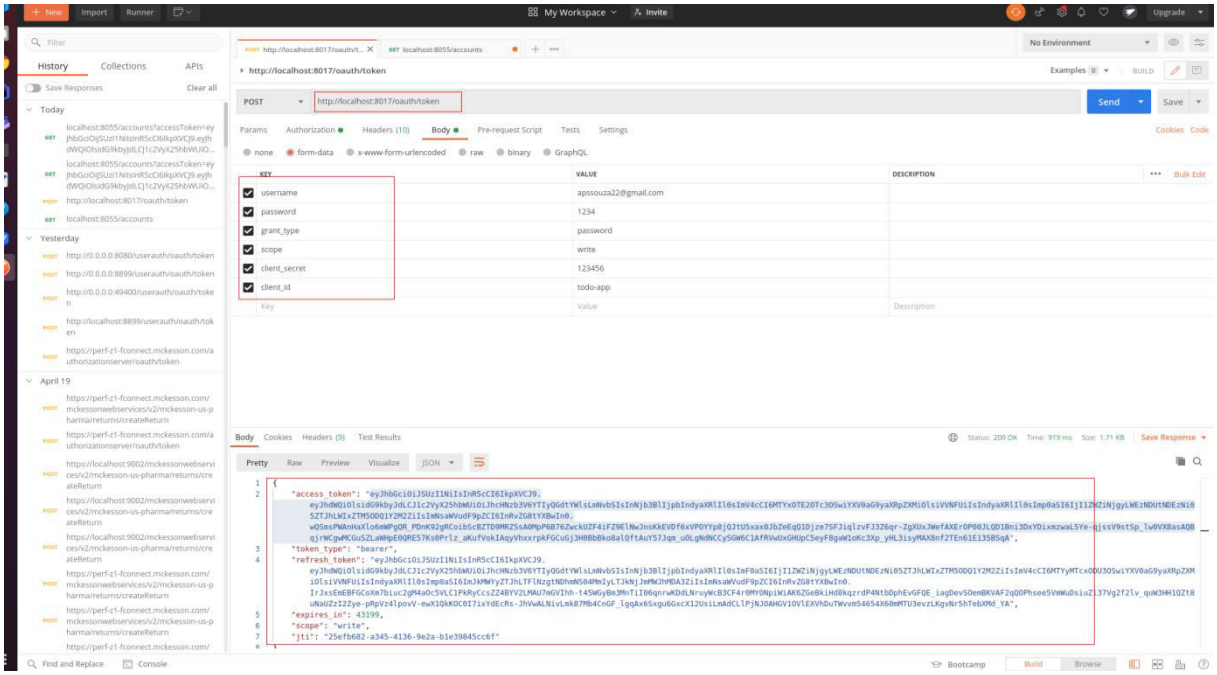

Рисунок 3.4 - Аутентифікація за допомогою OAuth2.

За замовчуванням токен доступу, наданий сервісом авторизації, недовговічний і закінчується в залежності від наданого значення "expires_in".

Якщо звертатися до захищеного ресурсу з вичерпаним токеном, він відповість, що термін дії токена закінчився. У цьому сценарії додаток може запитати інший токен доступу з сервера авторизації за допомогою токена поновлення. Припустимо, що ми вже отримали токен доступу для надання пароля, тоді ми можемо виконати такий запит POST що має grant_type = refresh_token і необхідно вказати значення refresh_token, яке було отримано у відповіді на надання пароля(рисунок 3.5).

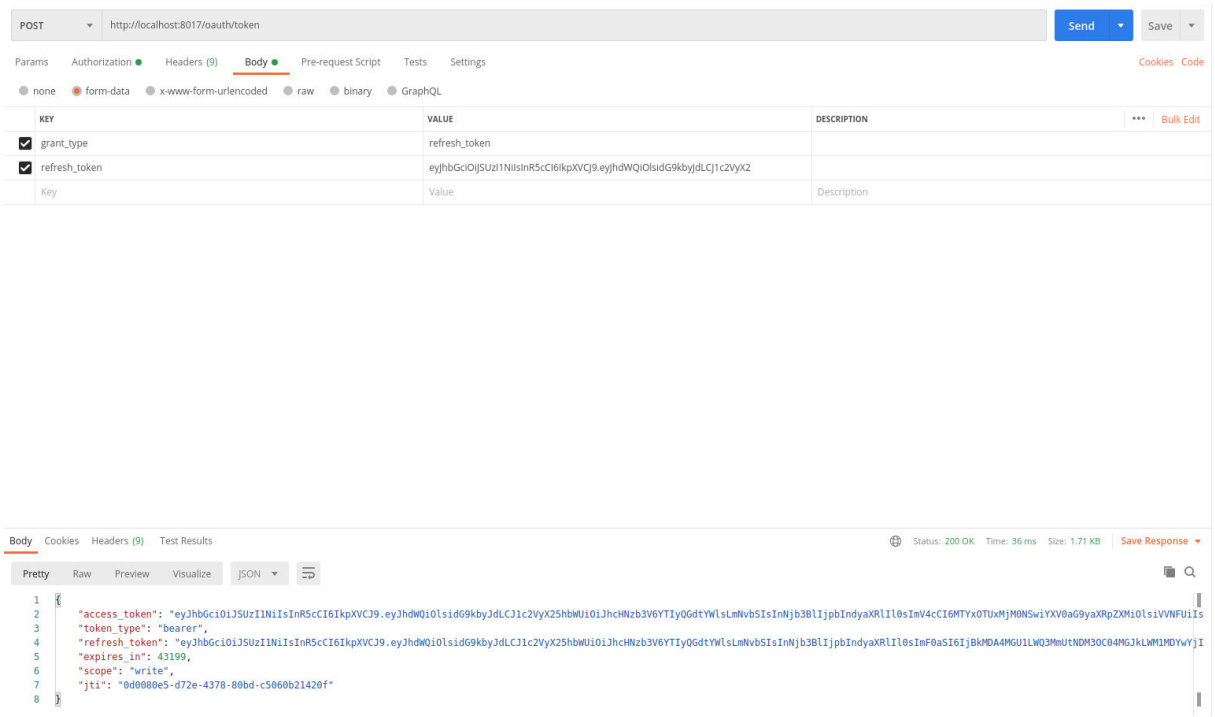

Рисунок 3.5 - Оновлення токену з допомогою OAuth

Сервіс обслуговування користувачів є RESTful сервісом, весь функціонал якого полягає в базових CRUD операціях. Доступ до сервісу обслуговування користувачів з OAuth2 сервісу безпосередній, з інших сервісів тільки за допомогою JWT токену. Має наступні rest ендпоінти:

- GET /accounts - для отримання всіх доступних акаунтів
- POST /accounts - для створення аккаунту
- PUT /accounts/{id} - для зміни аккаунту
- GET /accounts/{id} - для отримання аккаунту за допомогою id.
- GET /accounts?email= - для отримання аккаунту за допомогою електронної адреси.

Усі команди між OAuth сервісом та сервісом обслуговування користувачів виконуються з використанням hystrix команд, саме тому збій в роботі будь якого з них не є критичним для системи, в таких випадках вона буде переходити на альтернативний режим.

3.3 Eureka сервер та Spring admin

Eureka server та Spring admin - є основними сервісами які відображають стан системи, саме за їх допомогою можна слідкувати за станом системи, перевіряти журнали, моніторити використання ресурсів та ін.

Spring admin дозволяє розглянути стан кожного із сервісів більш детально, а саме він може відобразити всю інформацію щодо статусу кожного із сервісів(рисунок 3.6), а також має функціонал журналу статусів сервісів, який відображає статус сервісу та його переходи(рисунок 3.7);

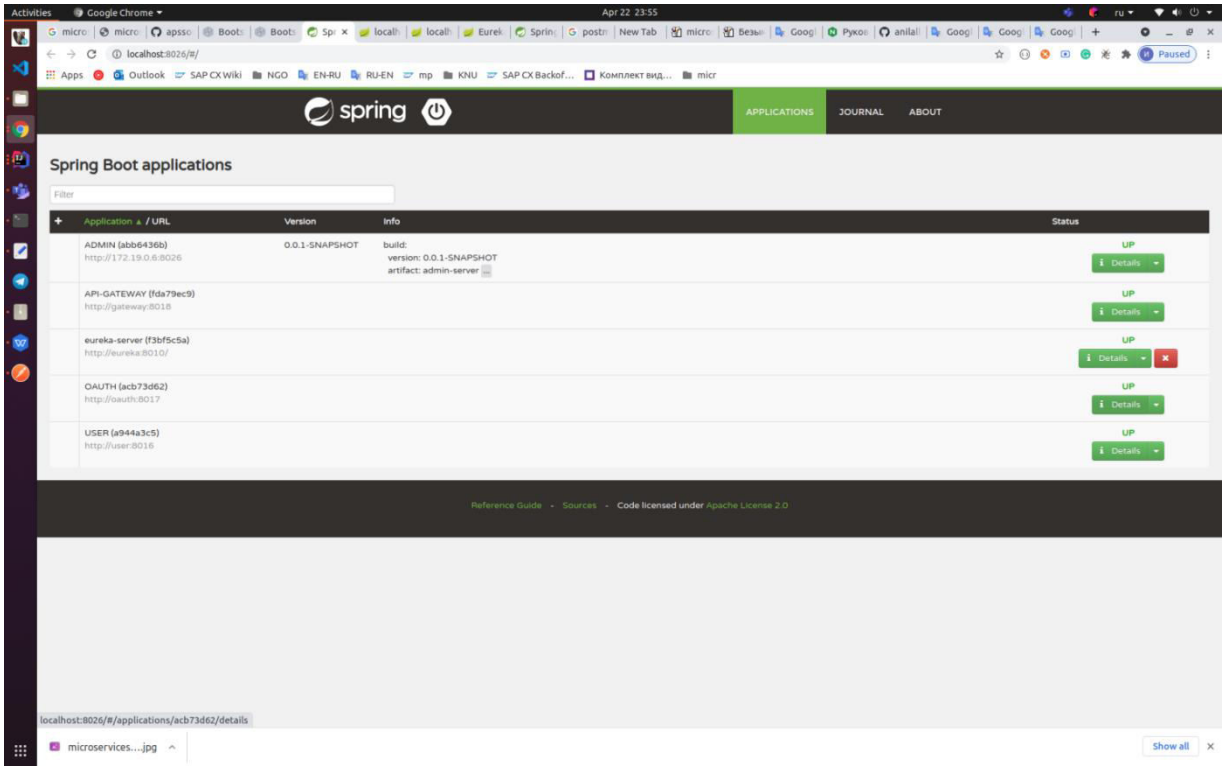

Рисунок 3.6 - Spring Admin main tab

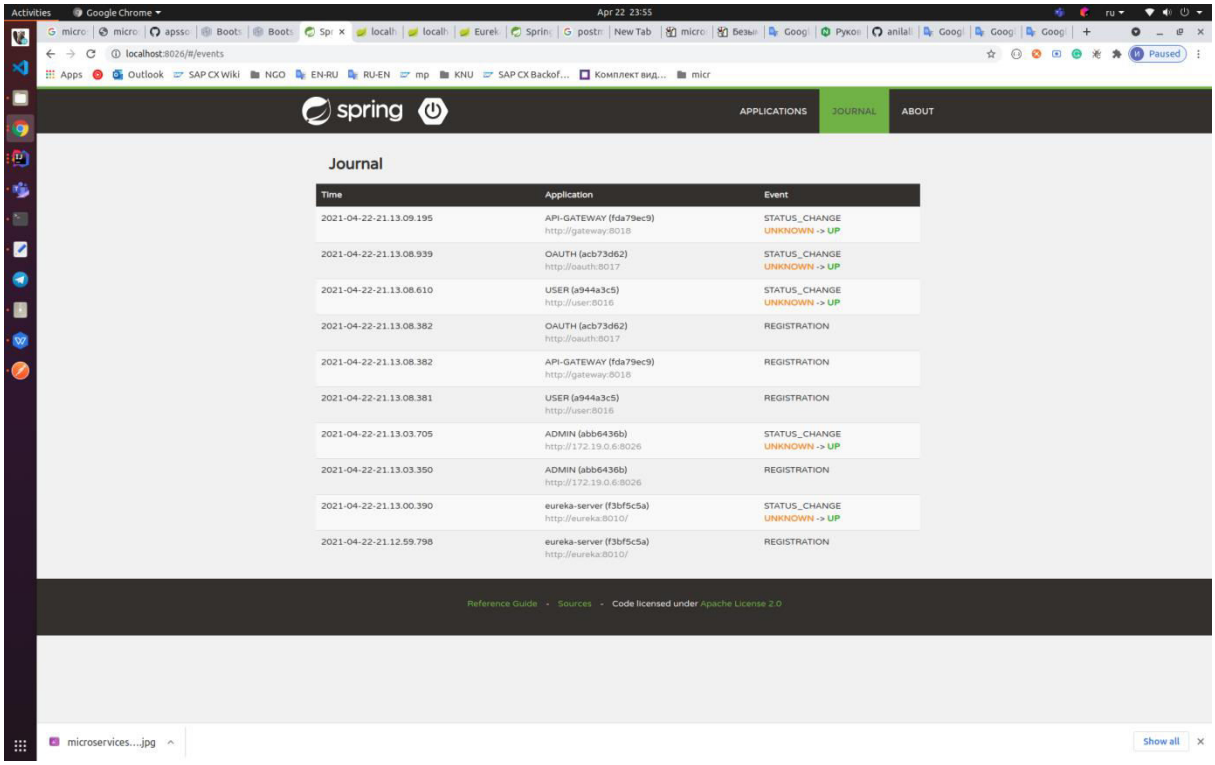

Рисунок 3.7 - Spring Admin Journal Tab

Що до деталей сервісу то в Spring Admin-і можна знайти інформацію про використання пам’яті, інформацію що до стану JVM, GC та ін. (рисунок 3.8).

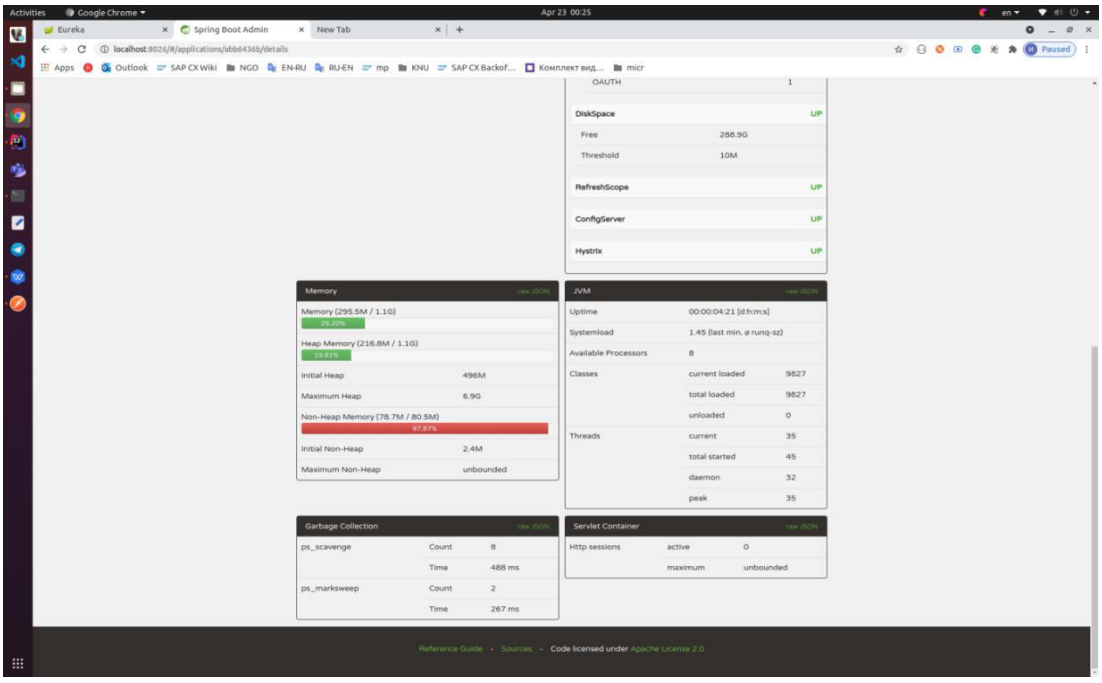

Рисунок 3.8 - Admin service info tab

Також в ньому можливо змінювати змінні середовища на льоту, змінювати рівень логування на льоту, отримати інформацію що до тредів та їх статусів отримати heap dump сервісу в реальному часі та ін.

За допомогою Eureka сервісу можна переглянути усі зареєстровані сервіси а також їх реальні адреси в мережі кластера, статус, кількість розгорнутих сервісів, основну інформацію та ін.

На мою думку впровадження таких сервісів на ранньому етапі розробки полегшує подальшу розробку та верифікацію.

3.4 Масштабування розробленої системи

Масштабування розробленої системи відбувається досить легко. Її можна масштабувати як горизонтально так і вертикально. Вважаючи що контейнерні системи та клауд сервери досить гарна зв'язка, так і за потреби масштабувати вертикально збільшуючи обсяг оперативної пам’яті досить легко. Для класичних серверів вертикальне масштабування провести дещо складніше, адже це потребує зупинки сервера. Для горизонтального масштабування достатньо розгорнути в кластері два або більше примірники потрібного сервісу та вказати для розподілювача навантаження на яких портах знаходяться сервіси, наприклад:

```
upstream backend {
   server gateway:8018;
   server gateway:DYNAMICPORT;
   server gateway:DYNAMICPORT;
}
```

Після чого розгорнути з вказаною кількістю сервісів які використовуються, або додати ці конфігурації динамічно.

Бази даних сервісів будуть використовувати CQRS патерн для обробки запитів до бази даних.

3.5 Результати

Отже, можна зробити наступні висновки: розробка з використанням мікросервісної архітектури досить складна на етапі взаємодії між сервісами, але переваги такої архітектури переважають її недоліки. Система досить легко масштабується, має вбудований функціонал для моніторингу, логування, та в цілому має інструменти для полегшення подальшої розробки.

ВИСНОВКИ

У даній роботі було проведено детальний аналіз проблем, які виникають під час розробки масштабованих мікросервісів та можливі шляхи їх рішення. Було розроблено програмний засіб, який є легко масштабованим та розширюваним а також містить в собі велику кількість вбудованого функціоналу.

1. Проведено порівняльний аналіз видів масштабування мікросервісів, виділено переваги та недоліки кожного з них, розглянуто архітектуру мікросервісів, наведено огляд основних компонентів цієї архітектури на яких вона базується. Проведено аналіз проблем, які виникають під час розробки, розглянуто шляхи подолання проблем під час масштабування бази даних.

2. Запропоновано підхід до масштабування проектів що використовують мікросервіси. Надано архітектурне рішення до такого підходу.

3. Реалізовано проект на платформі Java Spring, який можна вважати шаблонним(скелетним) для забезпечення масштабованості при орієнтації на мікросервісну архітектуру.

Розробка з використанням мікросервісної архітектури є досить актуальною темою, адже кількість систем що в наш час мігрують з монолітної архітектури на мікросервісну досить велика. Також залишається актуальною проблема пошуку рішень для завдань з якими стикаються під час розробки з використанням цієї архітектури, оскільки не існує ідеального рішення, кожне з них - це деякий компроміс , який підходить для тої чи іншої системи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Difference between scaling horizontally and vertically [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://github.com/vaquarkhan/vaquarkhan/wiki/Difference-between-scaling-horizontally-and-vertically.
2. What is Vertical Scaling & Horizontal Scaling? [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.esds.co.in/blog/vertical-scaling-horizontal-scaling/#sthash.DWNJOy61.dpbs.
3. Scaling Horizontally vs. Scaling Vertically [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.section.io/blog/scaling-horizontally-vs-vertically/.
4. ABBOT M. L. THE ART OF SCALABILITY / M. L. ABBOT, M. T. FISHER., 2015. – 624 с. – (Second Edition).
5. Bloomberg J. The dangers of "microservices-washing": Get to the value, strip away the hype [Електронний ресурс] / Jason Bloomberg – Режим доступу до ресурсу: https://techbeacon.com/enterprise-it/dangers-microservices-washing-get-value-strip-away-hype.
6. Hilwa A. Analyst View: A microservices checklist [Електронний ресурс] / A. Hilwa – Режим доступу до ресурсу: https://sdtimes.com/agile/analyst-watch-a-microservices-checklist/.
7. Eberhard W. Microservices: Flexible Software Architectures. / Wolff Eberhard., 2016. – 432 с. – (1st edition).
8. Feoktistov I. All You Need to Know About Microservices Database Management [Електронний ресурс] / Ihor Feoktistov – Режим доступу до ресурсу: https://relevant.software/blog/microservices-database-management/.
9. Quick Intro to Spring Cloud [Електронний ресурс] // baeldung – Режим доступу до ресурсу: https://www.baeldung.com/spring-cloud-configuration#:~:text=Spring%20Cloud%20Config%20is%20Spring's,be%20modified%20at%20application%20runtime..
10. Wahlstrom S. Comparing scaling benefits of monolithic and microservices architectures implemented in Java and Go / Simon Wahlstrom. – Swedem: Faculty of Computing Blekinge Institute of Technology, 2019.
11. Minkowski P. Mastering Spring Cloud / Piotr Minkowski., 2018.

12. Richards M. Software Architecture Pattern / Richards., 2017.
13. Merkel D. Docker: lightweight linux containers for consistent development and deployment / Merkel., 2014. – 239 с. – (2).
14. Carnell J. Spring Microservices in Action / John Carnell., 2017. – 384 с.
15. Zheng Y. Build a Eureka Server [Електронний ресурс] / Yang Zheng. – 2018. – Режим доступу до ресурсу: https://medium.com/fullstackwebdevelopers/simplest-spring-cloud-tutorial-in-2018-1-build-a-eureka-server-e61eeaed3b4b.
16. Защитите свое приложение с помощью Hystrix [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://coderlessons.com/articles/java/zashchitite-svoe-prilozhenie-s-pomoshchiu-hystrix.
17. Microservices — Service Registration and Discovery with Netflix Eureka [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://medium.com/@ijayakantha/microservices-service-registration-and-discovery-with-netflix-eureka-9a2aa729da96.