Міністерство освіти і науки України
Національний університет «Києво-Могилянська Академія»
Кафедра мережних технологій

КУРСОВА РОБОТА

за спеціальністю «Інженерія програмного забезпечення» 121

на тему:

Огляд та дослідження засобів побудови застосунку з мікросервісною архітектурою на основі можливостей Spring Boot та Spring Cloud

Науковий керівник:
доцент Франчук О. В.

Виконала:
студентка 1-го курсу Чернова Т. А.

Київ – 2022
Міністерство освіти і науки України

Національний університет «Києво-Могилянська Академія»

Кафедра мережних технологій

ЗАТВЕРДЖУЮ

Зав. кафедри мережних технологій,

професор, д.ф.-м.н.,

_____Г. І. Малошонок

“____”_____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенці Черновій Тетяні 1-го курсу магістратури факультету інформатики

ТЕМА: Огляд та дослідження засобів побудови застосунку з мікросервісною архітектурою на основі можливостей Spring Boot та Spring Cloud

Зміст ТЧ до курсової роботи:

Анотація

Вступ

1. Огляд мікросервісного підходу побудови RESTful веб-сервісів
2. Проблематика впровадження мікросервісного підходу для побудови застосунків
3. Відомі патерни розв’язання проблем мікросервісної архітектури та їх реалізації в Spring Cloud
4. Реалізація практичної частини роботи

Висновки

Список джерел

Додатки

Дата видачі “___” _____ 2022 р.

Керівник _____ Завдання отримано _____

Календарний план виконання курсової роботи

Тема: Огляд та дослідження засобів побудови застосунку з мікросервісною архітектурою на основі можливостей Spring Boot та Spring Cloud

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	жовтень 2021 р.	
2.	Огляд документації та літератури за темою роботи	жовтень - листопад 2021 р.	
3.	Оволодіння навичками розробки мікросервісної архітектури на основі Spring Cloud	листопад - січень 2022 р.	
4.	Реалізація практичної частини роботи	січень - березень 2022 р.	
6.	Написання теоретичної частини курсової роботи	лютий - березень 2022 р.	
7.	Надання роботи керівнику для перевірки, демонстрація практики	березень 2022 р.	
8.	Корегування роботи за результатами перевірки керівником	березень 2022 р.	
9.	Остаточне оформлення теоретичної частини та слайдів доповіді	квітень 2022 р.	
10.	Захист курсової роботи	травень 2022 р.	

Студент Чернова Т. А.

Керівник Франчук О. В.

“_____” _____ _____ р.

Зміст

Анотація 5

Вступ 6

1 Огляд мікросервісного підходу побудови RESTful веб-сервісів 7

1.1 Популярність мікросервісного підходу 7

1.2 Переваги використання мікросервісного підходу 8

2. Проблематика впровадження мікросервісного підходу для побудови застосунків 11

3. Відомі патерни розв'язання проблем мікросервісної архітектури та їх реалізації в Spring Cloud 14

3.1 Service Discovery 16

3.2 API Gateway 19

3.3 Circuit Breaker 21

3.4 Database per service 24

3.5 Saga 28

3.5.1 Choreography-based saga 30

3.5.2 Orchestration-based saga 31

4. Реалізація практичної частини роботи 33

4.1 Реалізація Discovery service 33

4.2 Реалізація API Gateway 37

Рисунок 4.2.1 - Spring Cloud Gateway залежності 37

Рисунок 4.2.2 - Spring Cloud Gateway налаштування 37

4.3 Реалізація Circuit Breaker 38

Висновки 40

Список використаних джерел 41

Анотація

Дана курсова робота присвячена огляду та дослідженню мікросервісного підходу при побудові веб-застосунків на базі технологій Spring Boot та Spring Cloud.

В теоретичній частині роботи розглянено основні переваги та проблематику мікросервісного підходу, найпоширеніші проблеми, що зустрічаються при побудові застосунків з мікросервісною архітектурою та відомі шляхи їх вирішення, реалізація цих рішень в Spring Cloud.

В практичній частині роботи було використано найбільш відомі мікросервісні патерни та наведені інструкції щодо їх впровадження до застосунку на основі фреймворку Spring.

Ключові слова: мікросервіси, мікросервісна архітектура, Spring Cloud, Service Discovery, Eureka, API Gateway, Circuit Breaker, API Composer, Database per Service, Saga, Choreography-based Saga, Orchestration-based Saga.

Вступ

На сьогоднішній день безліч компаній використовують, або хоча б раз задумувались над ідеєю використання мікросервісного підходу для побудови продукту. Чому ж даний підхід так вабить багатьох розробників та користується таким попитом? Серед основних ідей є легкомасштабованість застосунку, незв'язність компонентів продукту, підтримка різноманітних технологій. Ці та безліч інших переваг спонукають та заохочують девелоперів до переходу до мікросервісного підходу. Звісно ж, не обійтись і без проблем, які виникають на шляху впровадження таких мікросервісних рішень. Історично розробники при написанні мікросервісів виділяли та описували основні шляхи подолання проблем, що зустрічають при написанні додатків, що базуються на використанні мікросервісного підходу. Цими проблемами є і проблема комунікації різних сервісів, і збереження та консистентність даних, проблеми проведення транзакцій та відмовостійкість системи.

Обрана мною тематика дослідження мікросервісної архітектури для побудови веб-застосунків на основі Spring Boot та Spring Cloud не припиняє бути популярною, оскільки мікросервісний підхід продовжує набирати обертів на сьогоднішній день. Чому підхід популярний, які переваги він надає розробникам, що будуть веб-застосунки, які найуживаніші мікросервісні патерни, як долати проблеми пов'язані з мікросервісною архітектурою – усе це описано в даній роботі. За допомогою практичної складової роботи описано впровадження в застосунок Spring Cloud реалізації мікросервісних патернів, та приклади використання найпопулярніших мікросервісних рішень.

1 Огляд мікросервісного підходу побудови RESTful веб-сервісів

1.1 Популярність мікросервісного підходу

Термін “Мікросервісний підхід” або ж “мікросервісна архітектура” виник ще в середині 2010-х років та трактувався як один з особливих та сучасних стилів розробки застосунків. Його також скорочено називають “мікросервіси”. За ці десять років існування, підхід не припиняє здобувати увагу різноманітних форумів, курсів, презентацій та дискусій. На даний момент гіганти IT індустрії такі як Uber, Netflix, Amazon, SoundCloud, eBay, Walmart, Spotify, Twitter, PayPal, та Medium давно впровадили даний підхід до розробки всесвітньо відомих застосунків. І, мабуть, не залишилось ще жодної маленької компанії, що не намагалась розглянути релевантність впровадження даного підходу в своїй системі.

Мікросервіси [1] – сучасний підхід побудови єдиного застосунку, що формується як сукупність невеликих, самодостатніх, незалежних та не тісно зв’язаних між собою сервісів. Комунікація даних сервісів може відбуватись за допомогою відкритих REST API, тобто використовуючи механізм спілкування HTTP, або з використанням інших засобів, як то AMQP, RabbitMQ. При даному підході сервіси побудовані навколо бізнес потреб застосунку, що зумовлює використання єдиної відповідальності кожного окремого мікросервісу.

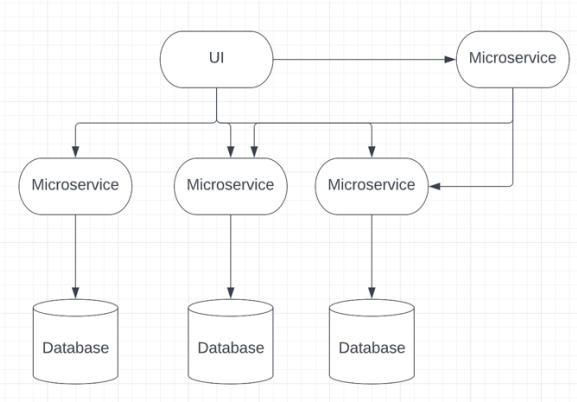

Рисунок 1.1.1 – мікросервісна архітектура

Популярність даного архітектурного рішення супроводжується рядом значних переваг. Серед них: можливість побудови складно масштабованих застосунків та корпоративних додатків, забезпечення гнучкості розробки та неприв'язка для конкретного технологічного стеку розробки [1]. Розглянемо деякі з переваг мікросервісного підходу детальніше.

1.2 Переваги використання мікросервісного підходу

- Кожен мікросервіс є невеликим та легко керованим сервісом

Мікросервіси є відносно малими, що дозволяє розробникам швидше розбиратись в структурі та цілях кожного окремого мікросервісу при першій зустрічі з ним. Малий кодбейз пришвидшує роботу та ефективність розробників та пришвидшує запуск та тестування мікросервісів порівняно з іншими архітектурними рішеннями.

- Автономність

Оскільки кожен мікросервіс існує незалежно від інших мікросервісів, це дозволяє відповідати йому за окрему ділянку усієї бізнес логіки. За рахунок автономності компоненти системи є слабозв'язними.

- Надійність

Мікросервіси надійні в плані відмовостійкості системи та інших ряд проблем, що можуть виникати під час роботи усього застосунку. У випадку поламки або ж просто неспроможності мікросервісу обробляти запити в конкретний момент часу, система дозволяє динамічно скеровувати запит до інших сервісів не завдаючи шкоди всій системі. Таким чином ми не маємо однієї точки відмови та можемо швидко знайти проблему та відновити роботу усієї системи.

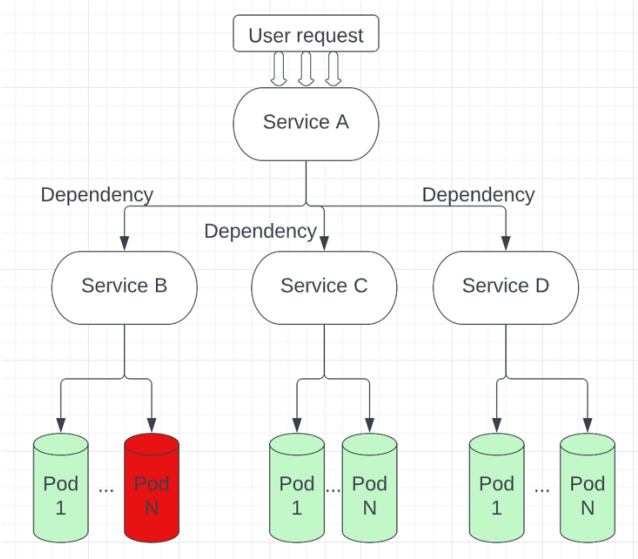

Рисунок 1.2.1 – надійність мікросервісів

Також гарним даної прикладом переваги є memory leak. Якщо дана проблема з пам'яттю виникає на одному з мікросервісів, це ніяк не зумовить та не повпливає на ефективність інших автономних одиниць.

- Незалежне масштабування

Кожен мікросервіс усієї архітектури може бути масштабованим окремо та ізольовано від інших мікросервісів з використанням різноманітних підходів масштабування, наприклад горизонтального або вертикального для збільшення конкретних показників потужності та якості мікросервісу.

- Можливість впровадження нових технологій

Дана перевага вважається чи не однією з найпопулярніших при обранні конкретної архітектури для усього застосунку. За рахунок того, що всі компоненти системи (мікросервіси) автономні та існують окремо один від одного, для розробника постає можливість створювати окремий технологічний стек для кожного мікросервісу переслідуючи свої технологічні цілі. Іншими словами, коли створюється новий мікросервіс, розробники вільні у виборі мови програмування, фреймворку та усіх технологій для конкретного нового мікросервісу.

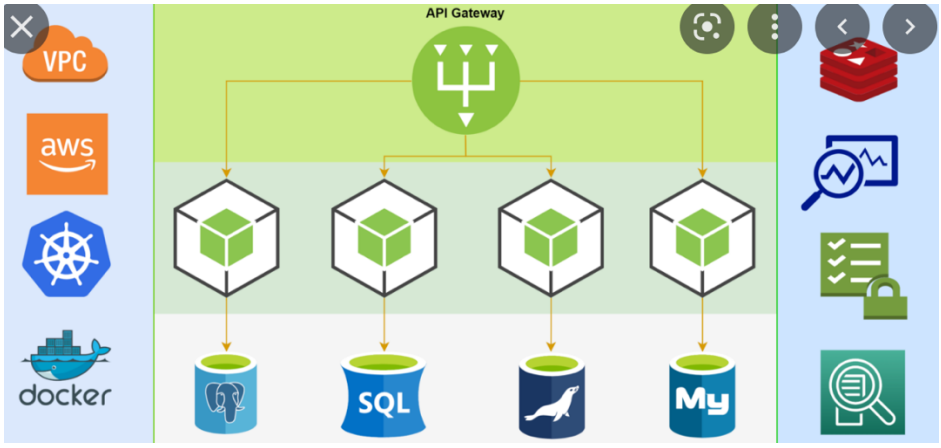

Рисунок 1.2.2 – впровадження нових технологій у мікросервісній архітектурі

- Легкість розгортання великих застосунків

При роботі з мікросервісною архітектурою, розгортання застосунку стає не важким завданням. Розгортання цього застосунку відбувається за рахунок розгортання окремих мікросервісів, відповідно до чого, усі процеси та підетапи розгортання такі як компіляція валідація, тестування та усі етапи CI/CD проходять значно швидше.

2. Проблематика впровадження мікросервісного підходу для побудови застосунків

Розглянувши попередньо велику кількість переваг щодо використання мікросервісної архітектури, постає питання, чому ж досі існують на ринку продукти, що надають перевагу іншим архітектурним рішенням? Для відповіді на це питання варто розглянути основні недоліки та проблеми, які постають перед розробником системи, що обрав мікросервіси як архітектурне рішення.

Серед основних проблем використання мікросервісної архітектури можна вважати ряд наступних:

- Визначення правильного набору мікросервісів для системи

Зважений і добре продуманий набір мікросервісів – це чи не найважливіше та найважче завдання при розробці системи. Деякі розробники вважають цей крок в побудові застосунком справжнім мистецтвом та приділяють йому багато уваги та часу. Наприклад при прийняття рішення переходу з моноліту до мікросервісної архітектури окреслити набір мікросервісів так, щоб правильно створити декомпозицію системи завдання не з найпростіших. При неправильному визначенні набору мікросервісів в результаті можна отримати розподілений моноліт замість мікросервісного застосунку [2]. Розподілений моноліт в обох підходах (мікросервісному та моноліті) несе за собою безліч недопрацювань та недоліків.

- Складність розподілених систем

Іншою не менш важливою проблемою, з якою зустрічається розробник на своєму шляху є додаткова складність при створення розподіленої системи. Чим більше компонентів поміщає в собі система, тим важче дається процес комунікації контролю та трейсінгу [2]. Перш за все, сервіси мусять мати налагоджений процес комунікації один між одним та правильний

підхід впровадження даного спілкування. Це в кільканадцять разів важчий процес, аніж виклик методу з іншого модуля (якщо брати до порівняння моноліт). При дизайнингу спілкування між мікросервісами, потрібно враховувати можливі затримки в мережі та загалом зупинки з відповіддю, можливі проблеми з недоступністю віддаленого серверу, затримки між запитами. Відмовостійкість – ще один з важливих етапів, на які варто звернути увагу при налагодженості комунікації між мікросервісами. Що робити у випадку помилкової відповіді, що робити у випадку недоступності мікросервісу у конкретний момент часу при створенні запиту до нього? Інколи дана задача стає ще складнішою, коли один запит може охоплювати різні мікросервіси та реагувати різними способами відповідно до відповіді від конкретного сервера. Такий варіант створює зв’язність між сервісами та ускладнює комунікацію між ними. Варто також брати до уваги можливість імплементації механізму повторних запитів. Більш детально про варіанти вирішення проблем комунікацій можна буде дізнатись у наступному розділі.

Мікросервісна архітектура також представляє собою значну операційну складність. Чим більше компонентів, додаткових інстансів сервісів, тим більше потреби в запровадженні контролю ними в різних середовищах.

- Координація мікросервісів при створення нового функціоналу

Іншим важким завданням при роботі з мікросервісною архітектурою є створення та публікація нового функціоналу системи. Інколи новий функціонал задіює та несе з собою зміни в багатьох мікросервісах та частинках однієї системи, що значно ускладнює процес публікації новинки. Потрібно створити детальний план публікації фічі базуючись на залежностях від різних мікросервісів.

- Адаптація до мікросервісного рішення

При першому запуску проекту з мікросервісною архітектурою фактично не виникає проблем, або ж проблеми є, але вони незначні. Проте якщо

компанія або команда розробників схиляється до рішення перейти до нової мікросервісної архітектури, рефакторинг коду та декомпозиція проекту може зайняти велику частку часу. Тому також потрібно враховувати найліпший момент для зміни архітектури системи. З іншої сторони, якщо ви учасник стартапу, обрання мікросервісної архітектури може цілком сповільнити процес запуску продукту на ринок, оскільки вимагає багато зусиль в плані дизайну та розробки системи. Тому часто маленькі стартапи відмовляються на початкових етапах від мікросервісної архітектури та переходять до неї з часом, якщо потреба в цьому не зникає.

Як можна побачити, мікросервісна архітектура разом з безліччю переваг, тягне за собою важкі рішення та можливі проблеми при розробці систему. Тому до розробки не можна відноситись легковажно. Детальні проблеми та рішення боротьби з ними буде розглянуто в інших розділах цієї роботи.

3. Відомі патерни розв’язання проблем мікросервісної архітектури та їх реалізації в Spring Cloud

Мікросервісна архітектура, як і інші архітектурні рішення, несе за собою безліч проблем, з якими розробники мусять стикатись. Це і проблема комунікації між сервісами, проблема спільного логування, спільних конфігураційних файлів, реєстрація сервісів, механізми повторних запитів у випадку невдач. Один з найкращих шляхів опису проблем – мова патернів. За всі роки свого існування, було розглянено, запропоновано та затверджено набір патернів, що вирішують проблеми мікросервісного підходу. Патерн – рішення проблеми, що може бути багаторазово перевикористане в конкретному контексті задачі [2]. За рахунок того, що патерн окреслює конкретний контекст та дане запропоноване рішення може не працювати в рамках інших умов, постійне покращення патерна має місце на існування. Також виділяють таке поняття як pattern language, що означає набір патернів що відносяться до конкретного домену. Саме набір патернів на високому архітектурному рівню дозволяє зі старту розробникам приймати рішення щодо вибору та впровадження конкретного архітектурного рішення. Говорячи про мікросервіси, даних підхід також має свою pattern language. Відповідно, якщо мікросервісна архітерктура є хорошим вибором для системи, pattern language дозволить ефективно вирішити ряд непростих архітектурних та дизайн-проблем.

Кожен патерн має безліч робочих імплементацій, в залежності від того, яку технологію ви виберете за основу для впровадження мікросервісного підходу. В даній роботі буде розглянено імплементації популярних мікросервісних патернів для рішення проблем з використанням технології Spring Cloud. Spring Cloud – окремий модуль Spring, більшість функціоналу якого було написано відомою компанією Netflix. Spring Cloud модуль надає

користувачам реалізацію відомих патернів мікросервісної архітектури за допомогою невеликої кількості анотацій та конфігурацій застосунку. Нижче наведена таблиця популярних проблем, мікросервісні патерни та їх імплементації у Spring Cloud.

	Проблема	Мікросервісний патерн	Spring Cloud імплементація патерну
1	Інформація про локацію сервісу для створення запиту до нього	Service Discovery	Eureka Service registry
2	Доступ до індивідуальних сервісів	API composition	API Gateway
3	Помилка одного сервісу може каскадно спричиняти помилку інших сервісів задіяних в запиті	Circuit breaker	Resilience4J
4	Проблеми використання спільної бази для всіх мікросервісів	Database per service	Database per service
5	Консистентність даних між мікросервісами	Saga: Choreography / Orchestration	Saga

Тепер розглянемо кожну проблему, шляхи її рішення та конкретну імплементацію більше детально.

3.1 Service Discovery

Розглянемо наступну проблему: припустимо, що у нашій системі є сервіс, що робить запити до REST API іншого сервісу. Звісно, якщо два сервіси в мають спільний codebase, як до прикладу при монолітному підході, то у нас не виникне проблем зробити запит від сервісу до сервісу. Проте, у випадку мікросервісного підходу, запит буде здійснено до іншої мережі: першому сервісу необхідно знати адресу (IP address та порт) другого сервісу, або ж його конкретного інстанса. Для рішення цієї проблеми ми можемо зберігати захардкоджені значення у конфігураційних файлах та при створенні запиту до іншого сервісу читати усю необхідну інформацію про іншу мережу. Проте в сучасних мікросервісних рішеннях з використанням хмарних технологій, порт та айпі адреса можуть змінюватись динамічно при старті сервісу.

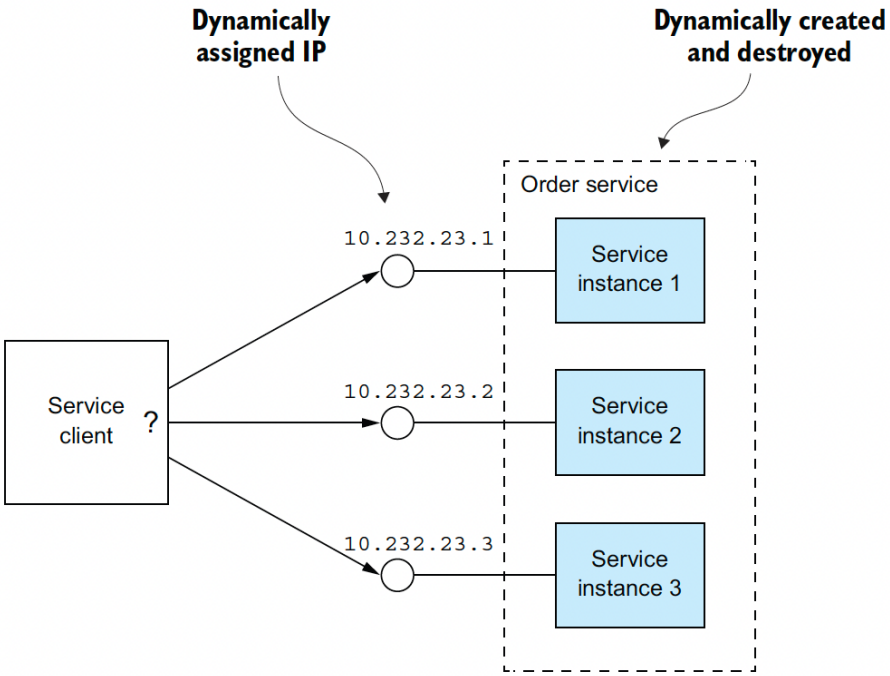

Рисунок 3.1.1 – Service Discovery

Джерело – https://microservices.io/patterns/

Кожному окремому інстансу сервісу буде присвоєна динамічна адреса в мережі. Більше того, набір інстансів конкретного сервісу можуть вирішувати змінювати динамічно свої адреси у випадку оновлень, помилок та автоскейлінгу.

Для рішення описаних проблем в дію вступає Discovery service. Сервіси взаємодіють з так званим сервісом реєстрації вказуючи інформацію про свою адресу та дозволяючи іншим сервісам отримувати по запиту цю інформацію. Окрім того, даний механізм дозволяє перевіряти стан системи за допомогою ряду операцій типу health check.

Discovery Service є поєднанням двох патернів: Self registration pattern та Client-side discovery pattern [2]. У випадку Self registration патерну, сервіси самостійно викликають service registry для того, щоб зареєструвати свою нетворк локацію. Даний патерн також надає API для перевірки стану конкретного зареєстрованого сервісу. Цей API викликає періодично сам service registry, окрім цього API доступний іншим користувачам при налаштуванні конфігурацій системи. Другий патерн - Client-side discovery дозволяє іншим сервісам створювати запит до service registry для отримання детальної інформації про розташування інших сервісів та їх інстансів у мережі.

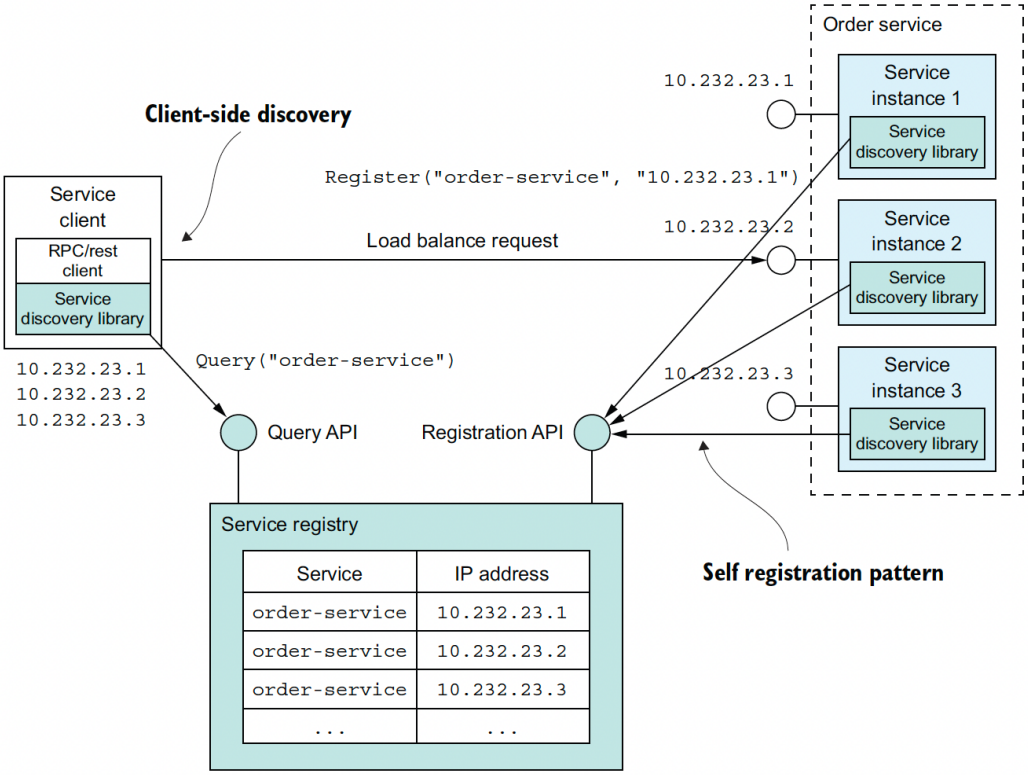

Рисунок 3.1.2 – Self registration та Client-side discovery patterns

Джерело – https://microservices.io/patterns/

Застосувавши даний патерн до згаданої вище проблеми, при запуску сервісів сервіси самостійно зареєструються в service registry. Коли перший сервіс захоче створити запит до другого сервісу, першим його кроком до цілі буде звернення до service registry з запитом на отримання інформації про мережеві дані другого сервісу. Таким чином сервіси дізнаються про розташування інших сервісів у мережі за допомогою посередника та самореєструються у нього.

В Spring Cloud даний патерн реалізується за допомогою Eureka Service Registry.

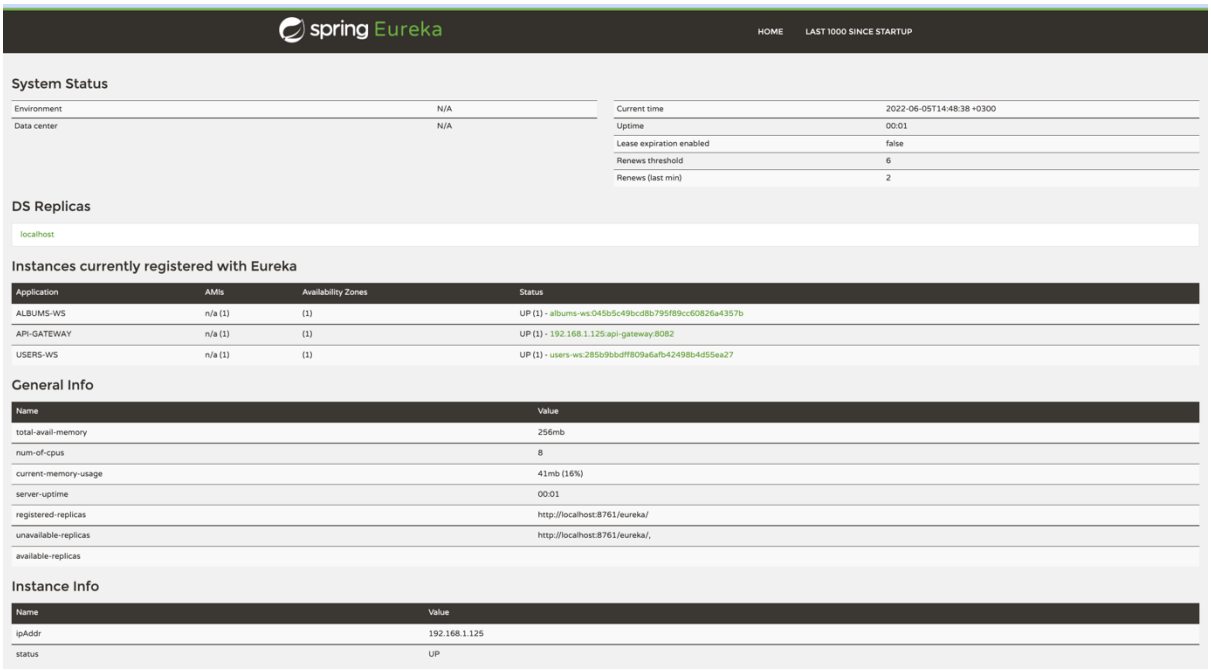

Рисунок 3.1.3 - Eureka

Завдяки Eureka сервіси самореєструються та дізнаються про знаходження інших сервісів, щоб створити запити до них, звернення до інших сервісів відбувається по назві сервісу, який кожен окремий сервіс вкаже під час самореєстрації. Eureka надає зручну користувацьку панель, що відображає поточний стан системи включаючи список зареєстрованих сервісів, кількість їх інстансів та стан, мережеву інформацію сервісу.

3.2 API Gateway

Уявімо, що ми розробляємо платформу для пошуку та продажу книг на базі мікросервісної архітектури. Кожна окрема сторінка міститиме інформацію про книгу, її ціну, кількість доступних книг, кількість замовлень. В результаті чого, для відображення такої інформації потрібно зробити кілька запитів до різних сервісів. Цими сервісами можуть бути: BookService, PricingService, InventoryService, BookOrderService. При використанні монолітного рішення, дана дія виконується легко: ми створюємо один SELECT запит до бази, де об’єднуємо таблиці та повертаємо усю необхідну інформацію в результати. Говорячи про

використання мікросервісного підходу, навіть якщо з технічної точки зору таку дію можна реалізувати, тим не менш це порушуватиме один з принципів об’єктно-орієнтованого програмування – encapsulation.

На допомогу приходить мікросервісний патерн - API composition pattern. Даний патерн є single entry point для сервісів, що намагаються створити складні запити. Іншими словами - API composition є єдиною точною доступу, що в свою сергу викликає усі необхідні сервіси що володіють даними та комбінує їх у результат. Структура даного патерну складається з двох частин: API Composer, що імплементує операції запиту викликаючи різні необхідні сервіси, та Provider service, що містить якісь дані що потрібні для результату [2].

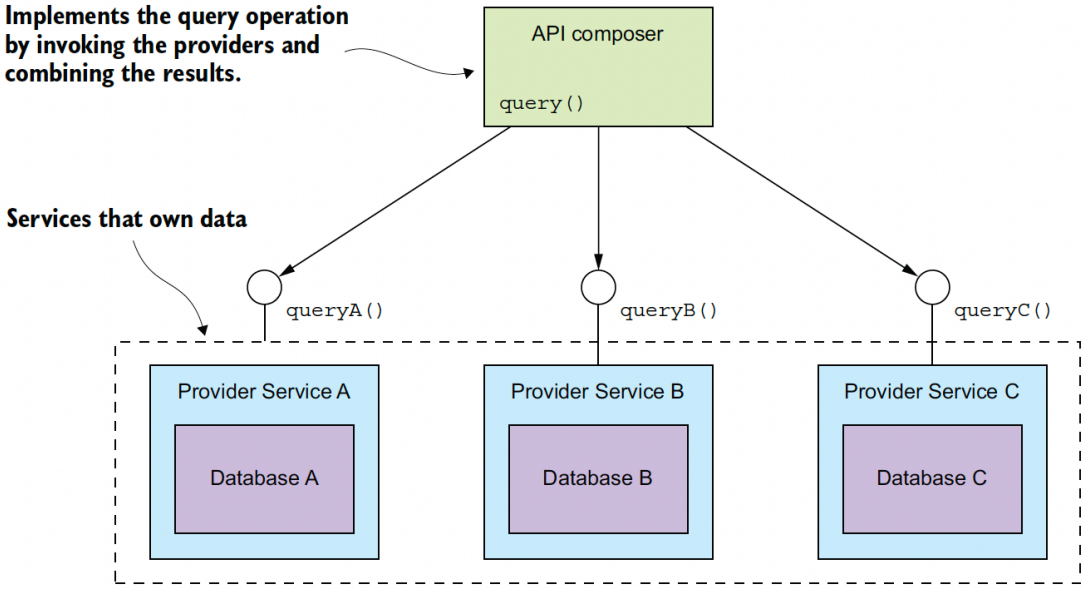

Рисунок 3.2.1 – API Composer

Джерело – https://microservices.io/patterns/

API Composer збирає дані з різних Provider Services та агрегує дані в результат. На місці API Composer може бути клієнт, до прикладу web application, що потребує дані для відображення сторінки, також це може бути окремий сервіс, як API Gateway.

Spring Cloud реалізує даний патерн у вигляді окремого сервісу, що має назву API Gateway. Даний патерн має ряд переваг:

- Ізолює клієнтів від застосунку що має мікросервісну архітектуру
- Ізолює клієнтів від проблеми з визначенням місця розташування інстансів окремих сервісів
- Надає оптимальний API для кожного клієнта
- Зменшує кількість запитів (roundtrips)
- У випадку з мобільними клієнтами є шлюзом
- Перекладає запити зі стандартного web-friendly API protocol до будь-якого іншого необхідного для спілкування з окремим провайдером даних

Також виділяють рад недоліків:

- Ще один сервіс, який потрібно менеджити та деплоїти
- Можливе збільшення часу відповіді через ще один проміжний шлюз

3.3 Circuit Breaker

В розподілених системах, коли сервіси роблять ряд синхронних запитів до інших сервісів, завжди є ризик виникнення помилок. Якщо у сервісі виникає помилка, до прикладу мережева помилка, є ризик утворення каскаду помилок від усіх інших сервісів, що беруть участь в запиті. Не менш ризикованою є ситуація, коли конкретний сервіс перенавантажений, що спричиняє затримку у відповідях інших сервісів. Якщо клієнт заблокований очікуванням на відповіді, виникає каскадна затримка у відповіді клієнтів клієнта і тд.

Рішенням цієї проблеми є патерн Circuit Breaker. Концепція Circuit Breaker полягає у тому, щоб уникнути викликів мікросервісів до інших сервісів, що потенційно можуть створювати помилки викликів, затримки або ряд інших проблем. Ціль патерна – уникнути ситуацій коли клієнт

витрачає час та ресурси на виклики до сервісів з потенційним ризиком виникнення помилок.

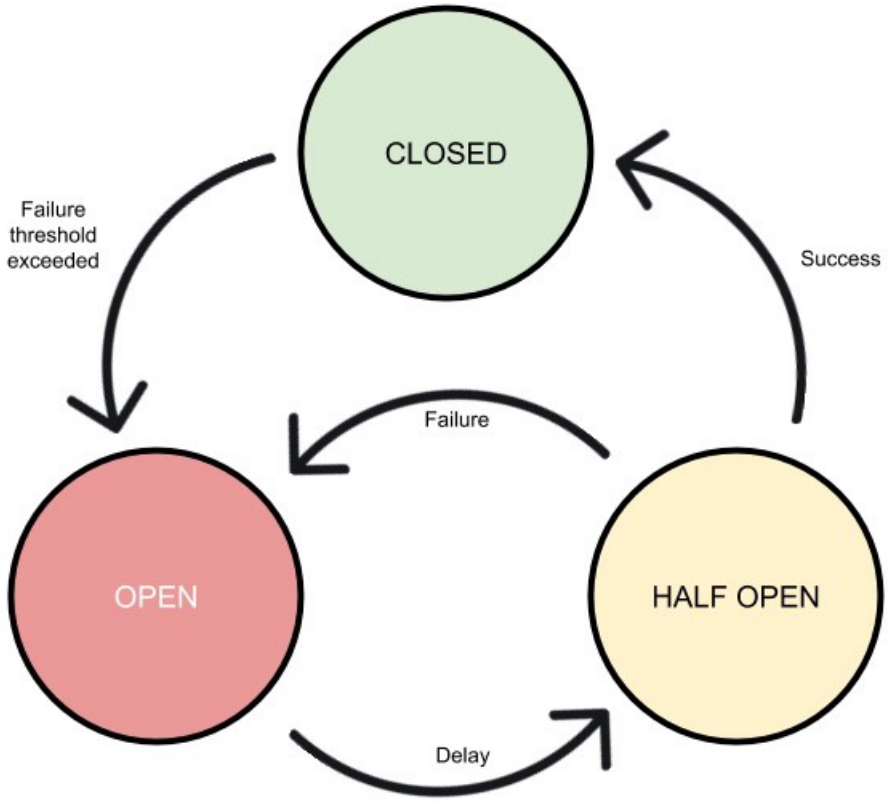

Рисунок 3.3.1 – Circuit Breaker states

Джерело – https://medium.com/bonniernewstech/circuit-breaker-pattern-what-and-why-a17f8babbec0

Патерн Circuit Breaker описує кілька станів:

- Closed: запит проходить успішно
- Open: коли з’являється помилка за певним попередньо визначеним критерієм. В цьому випадку запити до мікросервісу не будуть виконані, натомість буде розглянено можливість виклику fail-fast або ж fallback сценаріїв. Якщо запит пройде певний часовий ліміт, то відбудеться перехід до Half open стану

- Half Open: декілька запитів буде здійснено для розуміння того, чи мікросервіси до яких ми звертаємось перебувають у нормальному стані та готові до обробки запитів. Якщо успішно – перехід до Closed стану, якщо ні - переходимо знову в Open стан.

Реалізацією даного патерна в Spring є популярний Resilience4J.

Resilience4J надає для використання елегантний спосіб обробки помилок та можливість створення запасного варіанту у випадку помилки іншого сервісу.

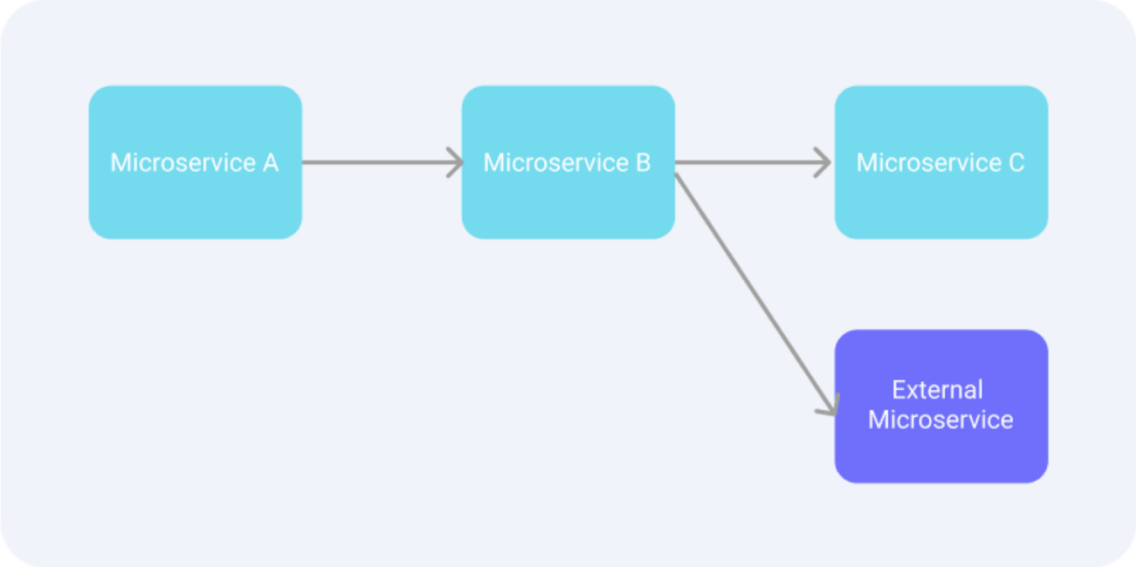

Рисунок 3.3.2 – Resilience4J fallback mechanism

До прикладу у випадку помилки запиту від Microservice B до Microservice C відбудеться інший запит до External Microservice.

```
# Resiliece4j Configuration
resilience4j.circuitbreaker.configs.shared.register-health-indicator=true
resilience4j.circuitbreaker.configs.shared.sliding-window-type=count_based
resilience4j.circuitbreaker.configs.shared.sliding-window-size=5
resilience4j.circuitbreaker.configs.shared.failure-rate-threshold=40
resilience4j.circuitbreaker.configs.shared.slow-call-rate-threshold=40
resilience4j.circuitbreaker.configs.shared.permitted-number-of-calls-in-half-open-state=1
resilience4j.circuitbreaker.configs.shared.max-wait-duration-in-half-open-state=10s
resilience4j.circuitbreaker.configs.shared.wait-duration-in-open-state=10s
resilience4j.circuitbreaker.configs.shared.slow-call-duration-threshold=2s
resilience4j.circuitbreaker.configs.shared.writable-stack-trace-enabled=true
resilience4j.circuitbreaker.configs.shared.automatic-transition-from-open-to-half-open-ena

resilience4j.circuitbreaker.instances.example.base-config=shared
```

application.properties hosted with ❤️ by GitHub view raw

Рисунок 3.3.3 – Resilience4J settings

Resilience4J надає безліч можливостей для конфігурації даного підходу для досконалої та детальної обробки таких помилкових ситуацій.

3.4 Database per service

Одним з найважливіших та найбільш спірним питанням при розробці застосунку з використанням мікросервісного підходу є архітектурне питання баз даних. Використання однієї бази даних на усі мікросервіси, чи інші рішення такі як: Private-tables-per-service, Schema-per-service, Database-server-per-service. Є безліч за та проти різних підходів, проте shared database підхід (підхід, при якому використовується одна база даних для усіх мікросервісів) вважається не типовим для мікросервісного підходу побудови застосунків та навіть його прозвали антипатерном з ряду доведених на теорії та практиці причин.

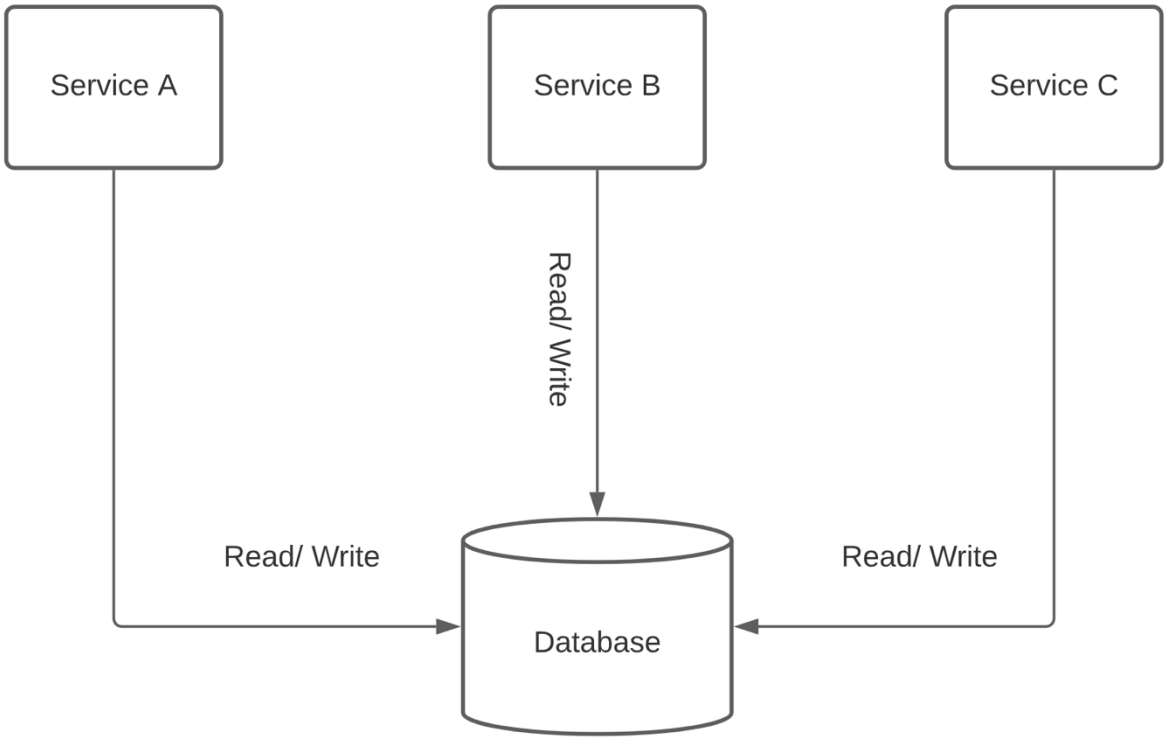

Рисунок 3.4.1 – shared database

Перш за все давайте розглянемо переваги даного підходу та чому ж ще й досі хтось сперечається щодо потреби в його використанні. Безперечно, даний підхід дозволяє без додаткових зусиль робити запити та акумулювати результат з різних таблиць без додаткових зусиль [3]. Ще одна важлива річ – відсутність необхідності обміну збереженими даними між мікросервісами. Таким чином не виникає проблем із узгодженістю даних і стану бази даних у зв’язку з різноманітними невдачами. Також shared database підхід не створює проблем з керуванням транзакціями, які в свою чергу є однією з найболючіших проблем інших підходів.

Проте чому ж цей підхід вважається не притаманним для мікросервісів і може використовуватись разом з ними лише на міграційному етапі з моноліту? А також чому даний підхід вважається чи не найвідомішим мікросервісним антипатерном? Розглянемо основні умови, що призводять до відмови від використання однієї бази даних для різних мікросервісів [3]:

- різні сервіси можуть мати різні вимоги щодо зберігання даних, тому об’єктивним є рішення для різних сервісів використовувати бази даних за їх внутрішньою потребою (один мікросервіс може використовувати MongoDB, інший – PostgreSQL і тд)

- мікросервіси повинні наслідувати принцип lose coupling

- development time coupling: сповільнюється час розробки, оскільки виникає додаткова зв’язність та потреба в координації впровадження змін до баз даних та загалом щодо роботи з базою даних

- runtime coupling: оскільки всі мікросервіси працюють та роблять запити до спільної бази даних, інколи в конкретну одиницю часу звернення до бази даних можуть блокувати операції інших сервісів та сповільнювати загальний час відповіді сервера

Ці, та ряд інших причин приводить нас до іншого архітектурного рішення щодо баз даних для мікросервісів – Database per Service.

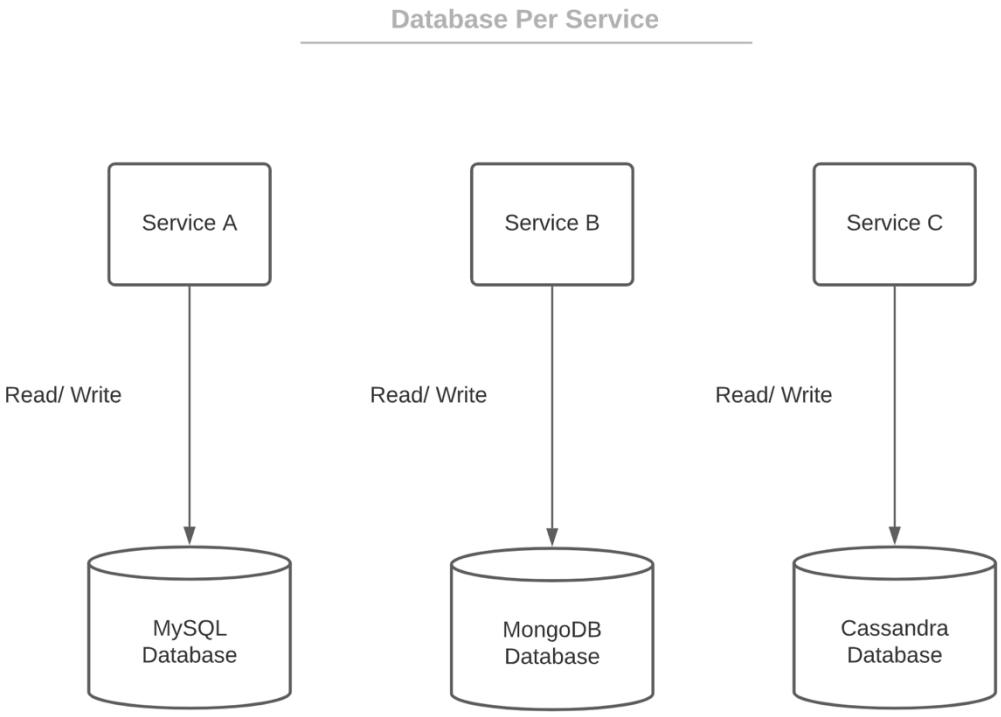

Рисунок 3.4.2 – database per service

Database per Service акумулює в собі просту ідею: кожен мікросервіс має свій власний datastore (це може бути як і ціла схема, так і окрема таблиця) і при цьому інші мікросервіси не мають доступу до нього напряму. Даний підхід робить архітектуру баз даних легко масштабовану, сервіси не зв'язними, дозволяє розробникам приймати рішення щодо створення та використання для різних мікросервісів різних технологій баз даних (це ще називають polyglot persistence) [4]. Тож один мікросервіс може використовувати реляційну БД, а інший ні. Також не меншим плюсом щодо впровадження даного підходу є відсутність однієї точки відмови. І ще одним вважливим моментом є те, що дані є енкапсульованими в межах конкретного мікросервісу, що робить його сприйняття разом з базою даних як щось єдине; розробникам більш зрозумілі зв'язки цілі та запити до БД.

Spring надає можливість підтримки та роботи одного застосунку з різними ресурсами баз даних за допомогою своїх влаштованих конфігурацій.

```
#Appointments Datasource
spring.appointments.datasource.jdbcUrl=jdbc:postgresql://localhost:5432/appointmentsdb
spring.appointments.datasource.username=postgres
spring.appointments.datasource.password=root

#Admissions Datasource
spring.admissions.datasource.jdbcUrl=jdbc:mysql://localhost:3301/admissionsdb?serverTimezone=UTC
spring.admissions.datasource.username=root
spring.admissions.datasource.password=root
spring.admissions.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
```

Рисунок 3.4.2 –database per service Spring support

Не дивлячись на усі вищезгадані переваги, використання однієї бази даних для одного сервісу несе за собою ряд недоліків таких як: контроль виконання транзакцій, проблема обміну даними між мікросервісами, налагодження комунікації між мікросервісами та організація об'єднаних запитів [4]. Деякі з цих проблем будуть більш детально прокоментовані разом з їх готовими рішеннями в ході дослідження наступного патерну – SAGA.

3.5 Saga

Налаштування комунікації між мікросервісами є дуже важливим компонентом в злагодженості роботи усієї системи. Для консистентності даних запити виконують в рамках транзакцій та в монолітній архітектурі слідують принципу ACID (atomicity, consistency, isolation, and durability).

ACID транзакції призводять розробників до ілюзії, що кожна транзакція має унікальний доступ до даних, що гарантує консистентність даних, ізольованість та атомарність операцій до баз даних. Проте в мікросервісній архітектурі ми не можемо використовувати даний принцип тому головна проблема полягатиме в тому, як виконати транзакцію, що є розподіленою між різних мікросервісів [5]. До прикладу розглянемо операцію створення замовлення товару. Дана операція включає в себе роботу кількох сервісів: Inventory service для отримання даних про доступність товару на складі, Accounting service для перевірки залишку коштів на балансі користувача та його спроможність оплатити товар, Order service для створення замовлення. Такого плану операції повинні включати в себе добре продуманий механізм управління транзакціями, що працює між мікросервісами.

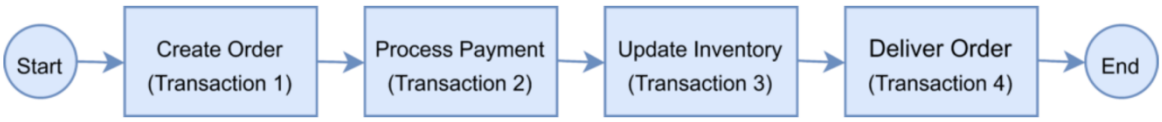

Рисунок 3.5.2 – розподілена транзакція

Джерело – https://www.baeldung.com/cs/saga-pattern-microservices

Відомим рішенням цієї проблеми є патерн розподілених транзакцій Saga – послідовність локальних транзакцій. Найбільшим випробуванням saga є те, що порівняно з ACID, saga є ACD, тобто не містять в собі Isolation. Іншими словами це означає, що коли один мікросервіс зберігає дані до бази а інший читає ці ж дані, то які дані мусять з’явитись в результаті читання:

старі чи нові? Рівень ізоляції в ACID визначає кількість даних, які відображаються в транзакції, коли інші мікросервіси одночасно отримують доступ до тих самих даних. Відсутність рівня ізоляції призводить до потреби у використанні додаткових принципів/підходів для його забезпечення, оскільки основним принципом в розподілених транзакціях є збереження ACID.

Saga працює за наступним принципом: після виконання локальної транзакції оновлюється база даних та публікується повідомлення про подію для ініціації наступної локальної транзакції до баз даних. У saga виділяють ряд наступних термінів [2]:

- Compensating transaction – транзакція, що скасовує зміни виконані попередньою транзакцією
- Countermeasure – дизайн техніка, що використовується для компенсації нестачі рівня ізоляції
- Compensatable transaction – транзакція, що потребує compensating transaction тому що наступна транзакція, що слідує даній може виконатись з помилкою
- Pivot transaction – “A transaction that is the saga’s go/no-go point” [2]. Має роль маркера для саги: якщо транзакція успішна то сага продовжуватиме ряд дій для завершення послідовності транзакцій
- Retriable transaction – транзакція, що слідує pivot transaction і мусить гарантовано виконатись

Як можна побачити, Saga включає в себе добре продуману логіку для забезпечення правильності виконання розподіленої між мікросервісами транзакції. Кожну локальну транзакцію saga можна компенсувати за допомогою її Compensating transaction. Крім того, Saga гарантує, що або всі операції завершені успішно, або відповідні компенсаційні транзакції будуть запущені для скасування раніше виконаної роботи [6].

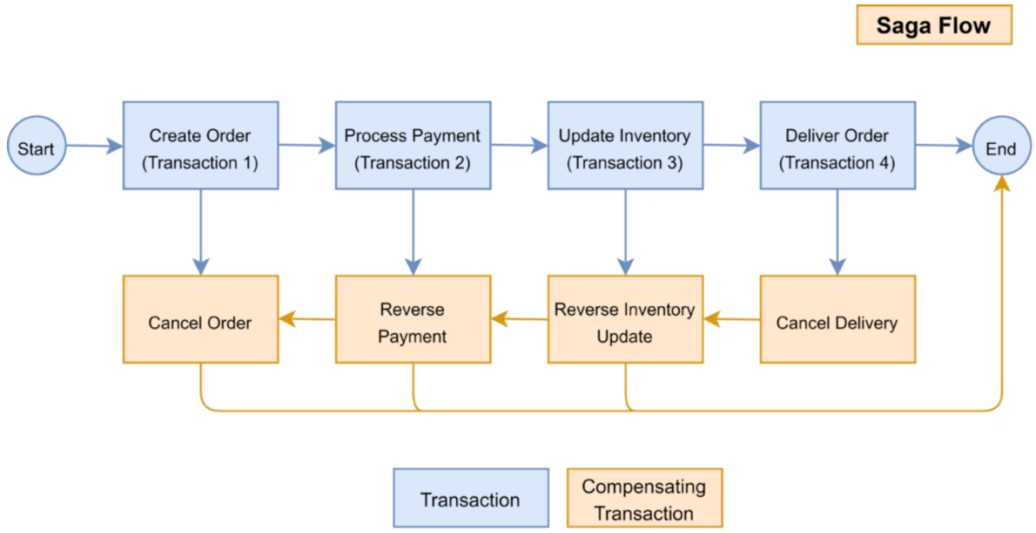

Рисунок 3.5.1 – Saga

Джерело – https://www.baeldung.com/cs/saga-pattern-microservices

Є два види координації транзакцій в Saga [5]:

- Choreography
- Orchestration

Розглянемо детальніше кожен з цих видів координації на прикладі створення замовлень.

3.5.1 Choreography-based saga

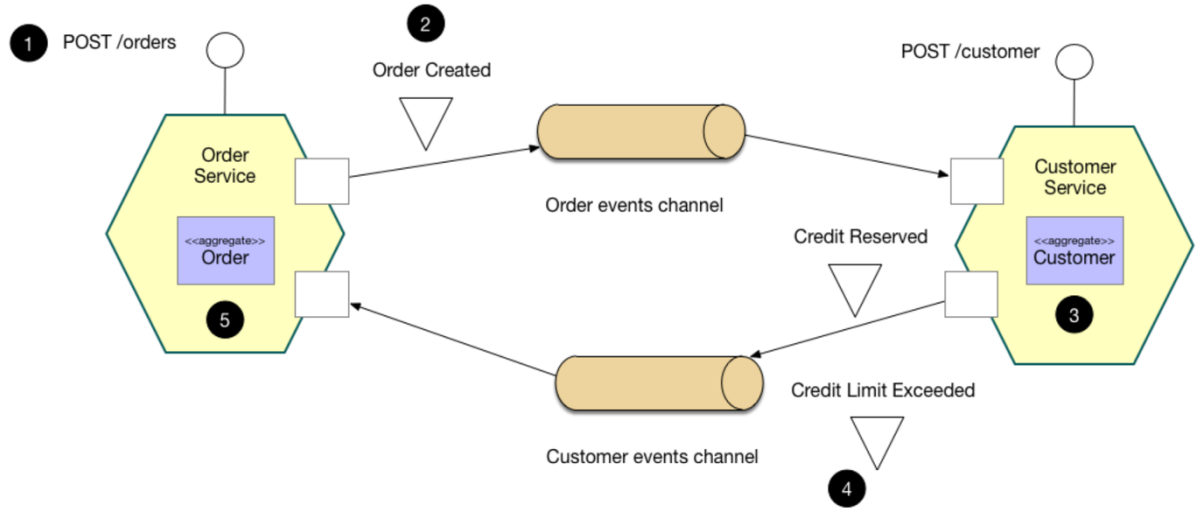

Рисунок 3.5.1.1 – choreography-based saga

Джерело – https://microservices.io/patterns/data/saga.html

Даний підхід визначає незалежне спілкування між сервісами через брокери повідомлень шляхом публікації подій. При даному виді координації транзакцій не потрібна наявність централізованого керування, оскільки мікросервіси в праві самостійно відправляти та обробляти дані від інших мікросервісів. На малюнку вище можна розглянути процес створення замовлення: користувач відправляє запит на сервіс замовлень, зберігає замовлення зі статусом “в обробці” та публікує подію про створення замовлення. Інші сервіси беруть в обробку подію та дають результат обробки, від чого залежить подальший статус замовлення. Важливо ідентифікувати замовлення певним статусом, щоб у випадку не успішної обробки події іншими сервісами можна було повернути процес замовлення до свого початкового етапу шляхом застосування компенсаційних транзакцій.

3.5.2 Orchestration-based saga

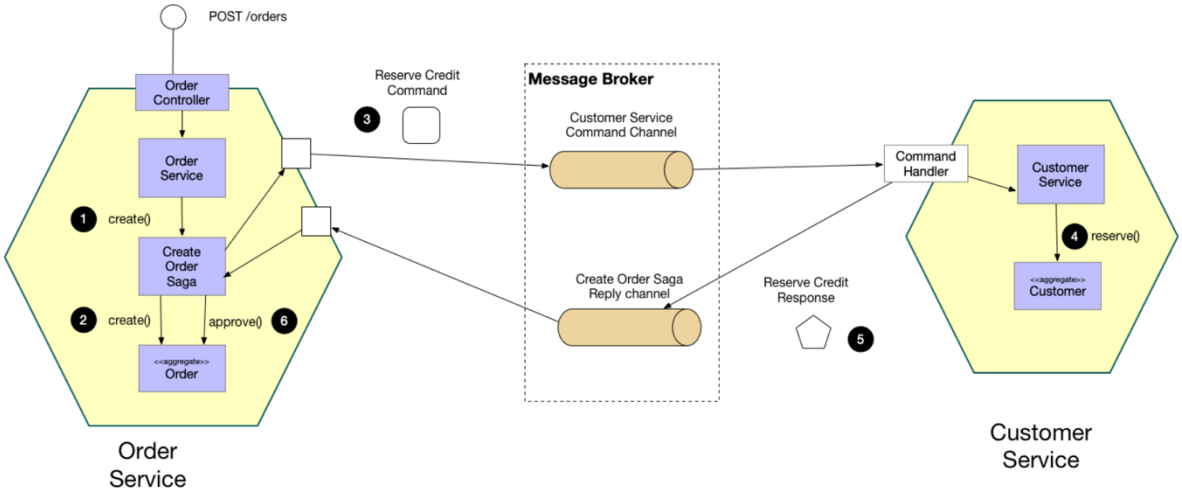

Рисунок 3.5.2.1 – orchestration-based saga

Джерело – https://microservices.io/patterns/data/saga.html

Даний підхід визначає централізовану систему керування транзакціями – Orchestrator. Orchestrator має контроль над результатами та

процесом обробки запису усіма мікросервісами. Якщо хоч один сервіс повертає помилку, то Orchestrator відправляє всім іншим сервісам команду на відміну змін, які були зроблені в ході транзакції. Відміна змін відбувається за допомогою рядку компенсаційних транзакцій.

4. Реалізація практичної частини роботи

Застосунок є практичною реалізацією мікросервісної архітектури з використанням базових, найбільш відомих мікросервісних патернів.
Ціллю розробки застосунку було набуття навичків практичної імплементації мікросервісної архітектури та перевірка швидкості та масштабованості такої архітектури на базі фреймворку Spring та Spring Cloud.

В ході роботи було використано ряд мікросервісних патернів та їх Spring Cloud реалізації. Отже, пропоную детальніше розглянути способи впровадження мікросервісних патернів у застосунок та розглянути способи використання Spring Cloud інструментів для досягнення цілей.

4.1 Реалізація Discovery service

Одним з найперших архітектурних рішень для побудови мікросервісного застосунку було впровадження патерну Discovery Service. Оскільки даний патерн вирішує критичну проблему інформування мікросервісів щодо місцезнаходження в мережі інших мікросервісів для їх подальшого спілкування, його наявність в застосунку є надважливим рішенням.

Для того щоб надати підтримку даного патерну в застосунку, достатньо додати залежність в pom.xml файл та анотацію EnableEurekaServer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Рисунок 4.1.1 – Eukera server залежність

```java
@SpringBootApplication
@EnableEurekaServer
public class PhotoAppDiscoveryServiceApplication {

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

}
```

Рисунок 4.1.2 – Eukera server налаштування

Таким чином ми створюємо власний Spring Cloud Netflix Eureka Server для подальшої реєстрації в ньому інших мікросервісів. Для того, щоб різноманітні мікросервіси могли реєструватись у вищезгаданому Eureka server, вони мусять виступати в ролі Spring Cloud Netflix Eureka Client. Якщо добавити наступні залежності та конфігурації до потенційних клієнтів, при запуску нових інстансів, вони зможуть реєструватись у service registry свою нетворк локацію.

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Рисунок 4.1.3 – Eukera client залежність

```java
@SpringBootApplication
@EnableDiscoveryClient
public class PhotoAppApiUsersApplication {
```

Рисунок 4.1.4 – Eukera client налаштування

Додавши наступні налаштування клієнта до application.properties файлу ми можемо кастомізувати налаштування за замовчуванням, які Eureka пропонує кожному клієнту для відображення в service registry.

```
# we will see this name in eureka discovery dashboard
spring.application.name=users-ws
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
spring.devtools.restart.enabled=true
eureka.instance.instance-id=${spring.application.name}:${spring.application.instance-id:${random.value}}
```

Рисунок 4.1.5 – Eukera client додаткові налаштування

Сам service registry виглядає наступним чином:

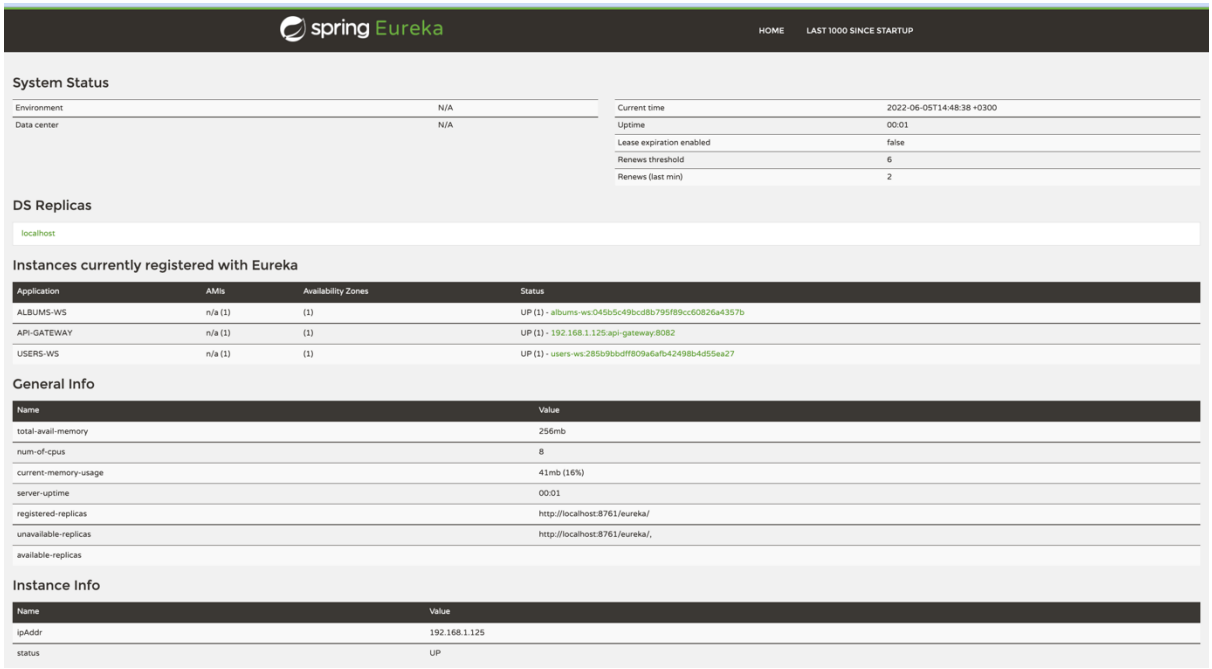

Рисунок 4.1.6 – Eukera дашборд

Ми бачимо, що в користувацькій панелі доступна інформація про динамічні IP адреси мікросервісів, їх статуси та кількість інстансів. Окрім цього, статуси окремих мікросервісів ми можемо отримувати за допомогою health check API та технології Spring boot actuator скопіювавши адресу мікросервіса з Eureka dashboard:

```
},
"db":{
   "status":"UP",
   "details":{
      "database":"MySQL",
      "validationQuery":"isValid()"
   }
},
"discoveryComposite":{
   "status":"UP",
   "components":{
      "discoveryClient":{
         "status":"UP",
         "details":{
            "services":[
               "albums-ws",
               "api-gateway",
               "users-ws"
            ]
         }
      },
      "eureka":{
         "description":"Remote status from Eureka server",
         "status":"UP",
         "details":{
            "applications":{
               "USERS-WS":1,
               "ALBUMS-WS":1,
               "API-GATEWAY":1
            }
         }
      }
   }
},
"diskSpace":{
   "status":"UP",
   "details":{
      "total":499963174912,
      "free":131764912128,
      "threshold":10485760,
      "exists":true
   }
},
```


Рисунок 4.1.7 – health check

4.2 Реалізація API Gateway

Для реалізації API Gateway патерна, використаємо зручну технологію спрінга – Spring Cloud Gateway. Для його використання нам достатньо лише добавити такі залежності:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Рисунок 4.2.1 - Spring Cloud Gateway залежності

Також для налаштування routes (Spring Cloud використовує routs для делегації запитів клієнта до внутрішніх сервісів) необхідно описати конфігурації в application.properties файлі таким чином:

```
spring.cloud.gateway.routes[0].id=users-status-check
#lb stands here for load balancer,
# this will make spring cloud api gateway to use build-in load balancer client and route http request to one of the running instances
spring.cloud.gateway.routes[0].uri=lb://users-ws
#it is used to match the http request using address or http method,
# if the request matches, than it will be routed to the destination uri
# both of the conditions from predicates should be true
spring.cloud.gateway.routes[0].predicates[0]=Path=/users-ws/users/status/check
spring.cloud.gateway.routes[0].predicates[1]=Method=GET
#Header is a build-in predicate, .* - regular expression for any characters any numbers
spring.cloud.gateway.routes[0].predicates[2]=Header=Authorization, Bearer (.*)
#apply filter that removes request header cookies. For example RemoveRequestHeader - is a build-in filter
spring.cloud.gateway.routes[0].filters[0]=RemoveRequestHeader=Cookie
#filter will expect external path and route the request to the internal path
spring.cloud.gateway.routes[0].filters[1]=RewritePath=/users-ws/users/status/check, /users/status/check
spring.cloud.gateway.routes[0].filters[2]=AuthorizationHeaderFilter
```

Рисунок 4.2.2 - Spring Cloud Gateway налаштування

Отже, для кожного нового route необхідно описувати певні конфігурації у вищезгаданому файлі. Також варто звернути увагу на те, що Spring Cloud Gateway має вбудованого балансувальника навантаження. На рисунку можна побачити, що в налаштуванні route в URI destination вказано на початку префікс “lb”. Це означає, що запити будуть збалансовані та

розподілені до різних інстансів мікросервісу users-ws з використанням Spring Cloud Netflix Ribbon load-balancing. Балансувальник навантаження розподіляє запити між декількома інстансами мікросервіса для рівномірного навантаження. Тим самим ми зменшуємо навантаження на цілу систему.

4.3 Реалізація Circuit Breaker

Для вирішення проблеми з можливістю утворення каскадних помилок при порушенні роботи одного з мікросервісів, о бере участь в ланцюгу викликів, в застосунку було використано патерн Circuit Breaker, а зокрема його конкретну реалізацію в Spring Cloud - Resilience4J. Resilience4J дозволяє в дуже зручний спосіб уникати ситуацій, коли клієнт витрачає час та ресурси на виклики до сервісів з потенційним ризиком виникнення помилок.

Для того, щоб додати Resilience4J до застосунку, необхідно підключити наступну залежність в pom.xml файлі:

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>1.7.0</version>
</dependency>
```

Рисунок 4.3.1 - Resilience4J залежності

Resilience4J надає багато гнучких налаштувань для обробки помилок та можливість створення запасного варіанту у випадку помилки іншого сервісу [7].

```
#value in percentage
resilience4j.circuitbreaker.instances.albums-ws.failure-rate-threshold=50
#no call is needed to trigger this transition
resilience4j.circuitbreaker.instances.albums-ws.automatic-transition-from-open-to-half-open-enabled=true
#time for waiting from open to half-open state
resilience4j.circuitbreaker.instances.albums-ws.wait-duration-in-open-state=10000ms
#can be COUNT_BASED, TIME_BASED. COUNT_BASED aggregates the outcome of the last number of calls
resilience4j.circuitbreaker.instances.albums-ws.sliding-window-type=COUNT_BASED
#or demonstration (number of calls when CR should be closed).
# Than means if 50 percent of 2 calls fail, CR will move to state close.
resilience4j.circuitbreaker.instances.albums-ws.sliding-window-size=2
#minimum number of calls which are required per sliding window period before CR can calculate an error rate
resilience4j.circuitbreaker.instances.albums-ws.minimum-number-of-calls=1
#monitoring events using actuator - list of last 10 events
resilience4j.circuitbreaker.instances.albums-ws.event-consumer-buffer-size=10
```

Рисунок 4.3.2 - Resilience4J налаштування

Висновки

Мікросервісний підхід до написання веб застосунків ще не втратив свою популярність, та більше того продовжуватиме користуватись нею наступні кілька років. Він зацікавлює розробників та цілі команди своєю свободою в прийнятті технологічних рішень, можливістю будувати великі застосунки з легким процесом масштабування та деплою, різноманітними готовими рішеннями для розв’язку складних процесів, що виникають на етапі розробки.

В ході даної роботи було досліджено перспективи впровадження мікросервісної архітектури для веб-застосунків, що створені на основі Spring Boot та Spring Cloud. Зокрема було проаналізовано основні переваги даного підходу, а також найпопулярніші проблеми, з якими зустрічаються розробники в ході написання мікросервісів та їх популярні патерни для рішення.

В ході виконання практичної частини роботи, було розглянено конкретні реалізація мікросервісних патернів за допомогою Spring Cloud. Серед проаналізованих патернів наступні: Service Discovery, API Gateway, Self registration, Client-side discovery, Circuit Breaker, Shared database, Database per Service, Choreography based Saga, Orchestration based Saga.

Список використаних джерел

[1] - https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d

[2] - https://microservices.io/patterns/

[3] - https://microservices.io/patterns/data/shared-database.html

[4] - https://www.baeldung.com/cs/microservices-db-design

[5] - https://microservices.io/patterns/data/saga.html

[6] - https://www.baeldung.com/cs/saga-pattern-microservices

[7] - https://resilience4j.readme.io/docs/getting-started-3