МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Інститут прикладного системного аналізу .

(назва факультету, інституту)

Кафедра системного проектування .

(назва кафедри)

До захисту допущено

Завідувач кафедри

_____ А.І. Петренко .

(підпис) (ініціали, прізвище)

“___”_____2017 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломного проекту (роботи) освітньо-кваліфікаційного рівня “**спеціаліст** ”

(назва ОКР)

з напряму підготовки (спеціальності) _7.05010102, «Інформаційні технології проектування»

на тему: Розробка лабораторного практикуму з дисципліни «Основи сервіс-орієнтованих обчислень і архітектур»

Студент групи ДА-51с Акінфієва Анастасія _____

(шифр групи) (прізвище, ім’я, по батькові) (підпис)

Керівник проекту к.т.н., доц. Харченко К.В. _____

(вчені ступінь та звання, прізвище, ініціали) (підпис)

Консультанти:

нормоконтроль к.т.н., доц. Стіканов В.Ю. _____

(назва розділу ДП (ДР)) (вчені ступінь та звання, прізвище, ініціали) (підпис)

рецензент д.т.н., проф. кафедри Обчислювальної Техніки НТУУ КПІ ім. І.Сікорського Стіренко С.Г.. _____

(назва розділу ДП (ДР)) (вчені ступінь та звання, прізвище, ініціали) (підпис)

Київ – 2017

Національний технічний університет України
“Київський політехнічний інститут”

Факультет (інститут) **Інститут прикладного системного аналізу** .
(повна назва)

Кафедра **Системного проектування** .
(повна назва)

Напрям підготовки **050101, Комп’ютерні науки** .
(код, назва)

Спеціальність **7.05010102 «Інформаційні технології проектування».**
(код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ А.І. Петренко .
(підпис) (ініціали, прізвище)

“___”_____ 2016 р.

ЗАВДАННЯ
на дипломний проект (роботу) освітньо-кваліфікаційного рівня
“спеціаліст” .
(назва рівня)

студенту Акінфієва Анастасія Олегівна
(прізвище, ім’я, по батькові)

1. **Тема проекту** (роботи) Розробка лабораторного практикуму з дисципліни «Основи сервіс-орієнтованих обчислень і архітектур»
затверджена наказом по університету від «__»_____ 2016р. №_____

2. **Термін здачі** студентом закінченого проекту (роботи) “15” 01 2017р.

3. **Вихідні дані до проекту** (роботи)

Нефункціональні вимоги: сервіс-орієнтована архітектура, мова програмування Java, мікросервісний фреймворк Spring Cloud, API Gateway, Service Discovery, Inter-Process Communication, EDD, завдання та методичні вказівки для лабораторної роботи по практичному налаштуванню Spring Cloud на прикладі Java додатків.
Функціональні вимоги: розробка мікросервісу на базі Spring Cloud платформи. Лабораторний практикум (Додаток 1)

4. **Перелік питань, які мають бути розроблені**

- Провести аналіз мікросервісної архітектури та порівняти з монолітною архітектурою;
- Розглянути шаблони проектування мікросервісів;
- Розглянути особливості використання мікросервісної архітектури;
- Порівняти Java-фреймворки, які використовуються при створенні мікросервісів;
- Встановити та налаштувати базовий проект на Docker з використанням фреймворку Spring Cloud;
- Створити документ з методичними вказівками до лабораторних робіт.

5. **Перелік графічного (ілюстративного) матеріалу**

Структура монолітного додатку (1-й плакат)
Структура мікросервісного додатку(2-й плакат)
Структура API Gateway (3-й плакат)
Алгоритм виявлення сервісів (креслення 1)
Схема розподілених БД для мікросервісів (креслення 2)
Схема міжпроцесних комунікацій (4-й плакат)

6. **Консультанти**

Нормоконтроль к.т.н., доц. Стіканов В.Ю.

7. **Дата видачі завдання** “ 07 ” 09 2016 р.

Керівник дипломного проекту (роботи) _____ Харченко К.В.
(підпис) (ініціали, прізвище)

Завдання прийняв до виконання _____ Акінфієва А.О.
(підпис) (ініціали, прізвище)

ЗАТВЕРДЖУЮ

Керівник
дипломного проекту (роботи)
_____ __Харченко К.В._
(підпис) (ініціали, прізвище)
“___”_____2016 р.

КАЛЕНДАРНИЙ ПЛАН-ГРАФІК

виконання дипломного проекту (роботи)

студентом _____Акінфієвої А.О._____
(прізвище, ініціали)

№ з/п	Назва етапів роботи та питань, які повинні бути розроблені відповідно до завдання	Термін виконання	Позначки керівника про виконання завдань
1	Ознайомлення з технічною літературою і підготовка теоретичної частини роботи	10.09.2016 – 30.09.2016	
2	Аналіз вимог завдання, вибір методів і засобів розв’язання поставленої задачі	15.10.2016	
3	Проектування модулів пристрою, розробка моделей.	15.11.2016	
4	Тестування розроблених моделей модулів. Перевірка відповідності завданню.	30.12.2016	
5	Підготовка графічного матеріалу, оформлення пояснювальної записки, підготовка до захисту	05.01.2017	
6	Проходження нормоконтролю, отримання відгуку, рецензії, передача роботи в ДЕК	10.01.2017	
7	Захист дипломної роботи	15.01.2017	

Студент _____
(підпис)

АНОТАЦІЯ

до роботи спеціаліста Акінфієвої Анастасії Олегівни

на тему: «Розробка лабораторного практикуму з дисципліни «Основи сервіс-орієнтованих обчислень і архітектур»»

Дипломна робота присвячена дослідженню застосування мікросервісної архітектури для розробки складних систем. Виконано порівняння даної архітектури з підходом побудови монолітної системи, визначено основні переваги та недоліки обох. Зроблено детальний опис компонентів необхідних для функціонування ПЗ, побудованого за таким принципом Наведений огляд основних шаблонів проектування мікросервісних додатків. Для того, щоб створити власний мікросервісну програму, було детально проаналізовано та порівняно найбільш популярні фреймворки і вибрано найоптимальніші з них для поставлених задач. Була розроблена примітивна система для ознайомлення із мікросервісною архітектурою. Дана система була розгорнута та протестована на локальному комп'ютері. На основі цього було розроблено короткі методичні відомості та лабораторний практикум з курсу «Основи сервіс-орієнтованих обчислень і архітектур» з акцентом на мікросервісні технології. Дану роботу рекомендовано використовувати в якості прототипу для створення об'ємних та змістовних методичних матеріалів, які в подальшому можуть бути застосовані для здобуття студентами практичних навичок в проектуванні систем такого роду.

Робота складається з 117 сторінок, з них обсяг основної частини –92 сторінок, 27 рисунків, 3 таблиці, 12 посилань та 1 додаток.

Перелік ключових слів: Мікросервісна архітектура, SOA, REST, Docker.

АННОТАЦИЯ

к работе специалиста Акинфиевой Анастасии Олеговны

на тему: «Разработка лабораторного практикума по дисциплине «Основы сервис-ориентированных вычислений и архитектур»

Дипломная работа посвящена исследованию применения микросервисной архитектуры для разработки сложных систем. Выполнено сравнение данной архитектуры с подходом построения монолитной системы, определены основные преимущества и недостатки обоих. Сделано детальное описание компонентов, необходимых для функционирования ПО, построенного по такому принципу, приведён обзор основных шаблонов проектирования микросервисних приложений. Для того, чтобы создать собственную микросервисную программу, детально проанализированы и сравнены наиболее популярные фреймворки и выбраны самые оптимальные из них для поставленных задач. Была разработана примитивная система для ознакомления с микросервисною архитектурой. Данная система была развернута и протестирована на локальном компьютере. На основе этого были разработаны краткие методические сведения и лабораторный практикум по курсу «Основы сервис-ориентированных вычислений и архитектур» с акцентом на микросервисные технологии. Данную работу рекомендовано использовать в качестве прототипа для создания объемных и содержательных методических материалов, которые в дальнейшем могут быть использованы для получения студентами практических навыков в проектировании систем такого рода.

Работа состоит из 117 страниц, из них объем основной части -92 страниц, 27 рисунков, 3 таблицы, 12 ссылок и 1 приложение.

Перечень ключевых слов: Микросервисная архитектура, SOA, REST, Docker.

ANNOTATION

to work of specialist Akinfieva Anastasia Olegovna

on the topic of: «Development of a laboratory practical work on discipline "Fundamentals of service-oriented computing and architectures»

Graduate work is devoted to research microservice application architecture for the development of complex systems. A comparison of the architecture with the approach of constructing a monolithic system done, the basic advantages and disadvantages of both are identified. Made a detailed description of the components required for the functioning of the software, which was built on this principle, gives an overview of basic design patterns microservice applications. To create your own microservice program analyzed in detail and compared to the most popular frameworks and choose the most optimal one for the task. Primitive system was developed to study the microservice architecture. This system has been developed and tested on the local computer. On this basis, developed a brief procedural information and laboratory practical work on the course "Fundamentals service-oriented computing architectures and 'with a focus on technology microservice. This work is recommended to be used as a prototype for creating volume and content of teaching materials, which can later be used to obtain students practical skills in the design of such systems.

The work consists of 117 pages, of which the main part of the volume of -92 pages, 27 figures, 3 tables, 12 references and 1 app.

List of key words: Microservice architecture, SOA, REST, Docker.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ 13
ВВЕДЕННЯ 14
1. АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ. ПОРІВНЯННЯ З МОНОЛІТНОЮ АРХІТЕКТУРОЮ. 17
1.1. ПОБУДОВА МОНОЛІТНИХ ДОДАТКІВ 17
1.2. НЕДОЛІКИ МОНОЛІТУ 18
1.3. ГОЛОВНІ ІДЕЇ МІКРОСЕРВІСІВ 22
1.4. ПЕРЕВАГИ МІКРОСЕРВІСІВ 27
1.5. НЕДОЛІКИ МІКРОСЕРВІСІВ 29
1.6. ВИСНОВКИ 32
2. ШАБЛОНИ ПРОЕКТУВАННЯ МІКРОСЕРВІСІВ 33
2.1. АГРЕГАТОР (AGGREGATOR) 33
2.2. ПРОКСІ (PROXY) 35
2.3. ЛАНЦЮГОВИЙ (CHAINED) 36
2.4. ГІЛКА (BRANCH) 37
2.5. ДАНІ СПІЛЬНОГО ВИКОРИСТАННЯ (SHARED DATA) 38
2.6. АСИНХРОННИЙ ОБМІН ПОВІДОМЛЕННЯМИ (ASYNCHRONOUS MESSAGING) 39
2.7. ВИСНОВКИ 40
3. КЕРУВАННЯ ТЕРМІНАМИ ВИКОНАННЯ ДИПЛОМНОЇ РОБОТИ. КЕРУВАННЯ РИЗИКАМИ 41
3.1 КЕРУВАННЯ ТЕРМІНАМИ ВИКОНАННЯ ДИПЛОМНОЇ РОБОТИ 41
3.2 КЕРУВАННЯ РИЗИКАМИ 45
3.3 ВИСНОВКИ 46
4. КЛЮЧОВІ ОСОБЛИВОСТІ ВИКОРИСТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ 47
4.1. ВИКОРИСТАННЯ API GATEWAY 47

4.2. МІЖПРОЦЕСНІ КОМУНІКАЦІЇ 49
4.3. SERVICE DISCOVERY 53
4.4. УПРАВЛІННЯ ДАНИМИ 58
4.5. ВИСНОВКИ 62
5. ПОРІВНЯННЯ ВІДОМИХ JAVA-ФРЕЙМВОРКІВ ДЛЯ СТВОРЕННЯ МІКРОСЕРВІСІВ 63
5.1. ВИМОГИ 63
5.2. КАНДИДАТИ 64
5.3. DROPWIZARD 65
5.3.1 ГОЛОВНЕ 65
5.3.2 РЕСУРСИ 66
5.3.3. HTTPS 67
5.3.4.REST-КЛІЄНТ 67
5.3.5. БЕЗПЕКА 68
5.3.6. МОНІТОРИНГ 68
5.3.7. ВИСНОВКИ 68
5.4. VERTX 68
5.4.1 ГОЛОВНЕ 69
5.4.2 РЕСУРСИ 69
5.4.3. HTTPS 70
5.4.4.REST-КЛІЄНТ 70
5.4.5. БЕЗПЕКА 72
5.4.6. МОНІТОРИНГ 72
5.4.7. ВИСНОВКИ 72
5.5. SPRING BOOT 73
5.5.1 ГОЛОВНЕ 73
5.5.2 РЕСУРСИ 73
5.5.3. HTTPS 74
5.5.4.REST-КЛІЄНТ 74

5.5.5. БЕЗПЕКА....75
5.5.6. МОНІТОРИНГ....76
5.5.7. ВИСНОВКИ....76
5.6. RESTLET....76
5.6.1 ГОЛОВНЕ....76
5.6.2 РЕСУРСИ....77
5.6.3. HTTPS....78
5.6.4.REST-КЛІЄНТ....78
5.6.5. БЕЗПЕКА....79
5.6.6. МОНІТОРИНГ....79
5.6.7. ВИСНОВКИ....79
5.7. SPARK JAVA....79
5.7.1 ГОЛОВНЕ....79
5.7.2 РЕСУРСИ....80
5.7.3. HTTPS....80
5.7.4.REST-КЛІЄНТ....81
5.7.5. БЕЗПЕКА....81
5.7.6. МОНІТОРИНГ....81
5.7.7. ВИСНОВКИ....81
5.8. ВИСНОВКИ....82
6. ПРИКЛАД БАЗОВОГО ПРОЕКТУ НА ФРЕЙМФОРКУ SPRING CLOUD З ПОДАЛЬШИМ РОЗГОРТАННЯМ НА DOCKER....83
6.1.SPRING CLOUD....83
6.2. SPRING BOOT....83
6.3. SERVICE DISCOVERY ТА ІНТЕЛЕКТУАЛЬНА МАРШРУТИЗАЦІЯ....84
6.3.1. СЕРВІС КОНФІГУРАЦІЇ (CONFIGURATION SERVICE)....85
6.3.2. СЕРВІС ВИЯВЛЕННЯ(DISCOVERY SERVICE)....86
6.3.3. GATEWAY API....87
6.4. ПРИКЛАД ПРОЕКТУ....89

6.4.1. DOCKER 89
6.4.2. БАГАТОМОВНІСТЬ 89
6.4.3. МІКРОСЕРВІСНА АРХІТЕКТУРА 89
6.4.4. СЕРВІС ВІДКРИТТЯ 90
6.4.5. GATEWAY API 90
6.5. РОЗГОРТАННЯ ПРОЕКТУ НА DOCKER 90
6.5.1. ЗАВАНТАЖЕННЯ DOCKER 90
6.5.2. ВИМОГИ 91
6.5.3. ВСТАНОВЛЕННЯ ПРОЕКТУ 91
6.5.4. ЗАПУСК КЛАСТЕРА З DOCKER COMPOSE 92
6.6. ВИСНОВКИ 96
ВИСНОВКИ 97
ПЕРЕЛІК ПОСИЛАНЬ 99
ДОДАТОК А. ЛАБОРАТОРНИЙ ПРАКТИКУМ 101

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

MSA – Micro Service Architecture (мікросервісна архітектура)

SOA – Service-oriented Architecture (сервіс-орієнтована архітектура)

БД – база даних

ПЗ – програмне забезпечення

IDE – Integrated Development Environment (інтегроване середовище розробки)

VM – Virtual Machine (віртуальна машина)

IPC – Inter-Process Communication (міжпроцесні зв’язки)

РСУБД – Реляційна система управління базами даних

YAML (YAML Ain't Markup Language) – зручний для читання людиною формат серіалізації даних, концептуально близький до мов розмітки, але орієнтований на зручність введення-виведення типових структур даних багатьох мов програмування.

REST (Representational State Transfer) – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів.

API (Application Programming Interface) – набір визначень взаємодії різнотипного програмного забезпечення.

HTTP (Hyper Text Transfer Protocol) – протокол передачі даних, що використовується в комп'ютерних мережах

JSON (JavaScript Object Notation) – це текстовий формат обміну даними міжкомп’ютерами. Базується на тексті і може бути з легкістю прочитаний людиною.

ВВЕДЕННЯ

Пошуки найкращих способів побудови систем велися багато років. З ростом обсягу коду і функціональності програмного продукту виникає необхідність в управлінні складністю програми. Добре продумана архітектура і правильне розбиття програми на модулі допомагають справлятися з поставленим завданням. Варіантом реалізації архітектури може бути монолітна система, коли вся або більша частина бізнес-завдань має одну кодову базу. Альтернативою є система, яка побудована на так званих мікросервісах. В такій системі загальне бізнес-завдання розбите на окремі частини, кожна з яких має окремий додаток (мікросервіс) зі своєю кодовою базою. Сьогодні до такого принципу побудови своїх систем вдаються такі всесвітньовідомі компанії, як Amazon, Netflix, The Guardian.

Мікросервіси – це невеликі модулі, розділені за принципом виконання одного бізнес-завдання або одного класу задач. Основна мета такого поділу – можливість редагувати окремо взятий мікросервіс, не змінюючи при цьому пов’язаних з ним компонентів. Бізнес-логіка програми розбивається на окремі частини, кожна з яких представляє собою невеликий додаток, що виконує одне бізнес-завдання (single responsibility). Число таких додатків нічим не обмежено і між собою вони спілкуються, використовуючи API, побудований, наприклад, на основі HTTP (REST API). Якщо коротко, то мікросервісна архітектура (Micro Service Architecture, MSA) – це коли додаток являє собою велику кількість невеликих (буквально кілька сотень рядків коду) сервісів, які взаємодіють між собою обмінюючись повідомленнями.

Об’єктивно, ідея такої архітектури не особливо нова. Доволі довго існує такий підхід розробки програмного забезпечення, як SOA (Service-oriented architecture). Очевидно, що відмінність між SOA та MSA полягає в розмірі сервісів. У випадку з MSA вихідний код кожного окремого сервісу не повинен перевищувати декількох сотень рядків коду. SOA ж такого обмеження не

накладає, тому в кожному сервісі можуть бути десятки або сотні тисяч рядків коду. З цього випливає, що MSA – це окремий випадок SOA.

Основні переваги використання MSA:

1. Позбавляємося від високого зачеплення коду, розділяючи додаток на мікросервіси, – тобто у разі відмови одного з мікросервісів цілком можливо, що велика частина програми все одно залишиться працездатною.
2. Проста масштабованість. Забезпечується за рахунок розміщення мікросервісів на різних серверах.
3. Різні сервіси можуть розроблятися різними групами розробників.
4. Кожен мікросервіс може використовувати будь-яку технологію, застосування якої доцільно в рамках поставленого завдання

Основні складності при використанні MSA:

1. Складніше реалізувати загальну поведінку для всіх мікросервісів (авторизація запитів, агрегація даних з різних мікросервісів).
2. Мікросервіси можуть створювати додаткові затрати при обробці запитів користувачів.
3. Постійне врахування того, що будь-який з мікросервісів може бути недоступним у будь-який момент.

В даний час існує два підходи для побудови системи з використанням мікросервісної архітектури, не допустивши в ній грубих помилок. Один підхід пропонує спочатку побудувати монолітну систему, а потім, розібравшись в предметній області, розбивати його на мікросервіси. Другий же пропонує приділити більше уваги проектуванню і відразу починати реалізацію мікросервісів.

В цілому ж, вибір тієї чи іншої архітектури залежить від багатьох конкретних обставин. В цьому полягає актуальність даної роботи. Тобто, якомога детальніше розглянути мікросервісну архітектуру та підтвердити потрібність застосування її при тих чи інших обставинах. Також актуальною проблемою є те, що існує мало концентрованої інформації по темі

проектування мікросервісів, яку можна надавати студентам вузів для вивчення даної теми.

Метою даної роботи є проведення дослідження можливих рішень побудови додатків з використанням мікросервісів, їх порівняння, порівняння даної архітектури з іншими можливими підходами побудови схожих систем. Також метою є створення примітивного прототипу такого додатку для більш детального ознайомлення. У ході цього було проведено збір інформації та її систематизація для створення лабораторного практикуму та коротких методичних вказівок до нього для подальшого надання його студентам для вивчення та практичного ознайомлення із мікросервісною архітектурою побудови додатків. Даний лабораторний практикум можна буде застосовувати в подальшому на кафедрі у рамках навчальної програми.

1. ГОЛОВНІ ІДЕЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ. ПОРІВНЯННЯ З МОНОЛІТНОЮ АРХІТЕКТУРОЮ.

1.1. ПОБУДОВА МОНОЛІТНИХ ДОДАТКІВ

Для прикладу візьмемо реалізацію абсолютно нового додатку для виклику таксі, призначений для конкуренції з Uber. Після збору вимог, буде створено новий проект або повністю вручну, або з використання генератору шаблонів, який поставляється з такою платформою, як Rails, Spring Boot, Play або Maven. Цей додаток матиме наступну архітектуру (рис.1.1):

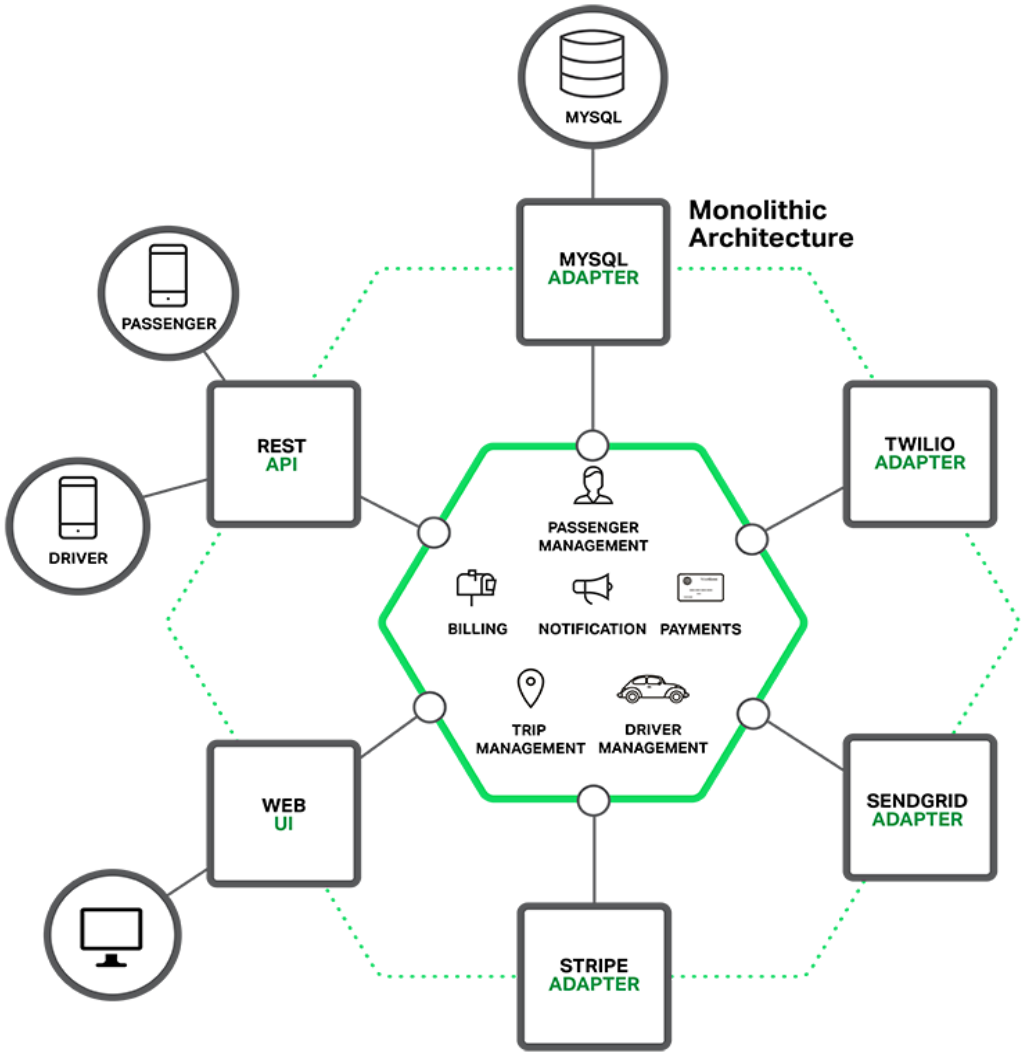

Рисунок 1.1 – Приклад монолітної архітектури додатку для виклику таксі

Ядром програми буде бізнес-логіка, яка реалізується через модулі визначення послуг, об’єктів домену та подій. Навколо ядра будуть адаптери для взаємодії з навколишнім світом. Наприклад, такі адаптери включають в себе компоненти доступу до БД, компоненти обміну повідомленнями, які створюють та зчитують повідомлення, а також веб-компоненти, які можуть надавати власний API чи реалізовувати користувацький інтерфейс[1].

Не беручи до уваги наявність логічно змодульованої архітектури, додаток в такому випадку скомпонований та розгорнутий як одна цілісна монолітна структура. Фактично, формат залежить від мови програмування, на якій написано додаток та фреймворку. Наприклад, багато Java-додатків скомпоновано як WAR-файли та розгорнуто на таких серверах, як, наприклад, Tomcat чи Jetty. Інші Java-додатки скомпоновано схожим чином з файлами JAR формату, які виконуються автоматично на стороні сервера. Аналогічно, Rails та Node.js додатки скомпоновано у вигляді ієрархії директорій. Додатки, написані в такому стилі, є надзвичайно поширеними. Вони прості для розробки, оскільки IDE та інші інструменти націлено на побудову єдиного додатку. До того ж, тестувати такий вид додатків легко. Можна реалізувати повне тестування лише запустивши додаток і протестувавши інтерфейс користувача через Selenium чи інший пакет для тестування. Монолітні додатки прості в розгортанні. Необхідно лише скопіювати скомпонований додаток на сервер, або ж змасштабувати додаток, запустивши декілька копій після запуску балансувальника навантаженя. На початковому етапі він дуже добре працює[1,2].

1.2. НЕДОЛІКИ МОНОЛІТУ

На жаль, цей простий підхід має величезне обмеження. Успішні додатки мають тенденцію до зростання з плином часу до величезних розмірів. Під час кожного спринту команда розробників реалізує все новий і новий функціонал,

що означає додавання великої кількості рядків коду. Через кілька років малий та простий додаток перетворюється в жахливий та величезний моноліт. Як тільки додаток стає великим, складним монолітом, організація процесу розробки, ймовірніше всього, почне страждати від цього. Будь-які спроби швидкої розробки і постійного постачання продукту будуть збиватися. Однією з основних проблем є те, що додаток занадто великий для будь-якого окремо взятого розробника, щоб повністю зрозуміти його. В результаті, виправлення помилок і реалізація нових можливостей правильним чином стає важкою і нетривіальною задачею, яка забирає багато часу. Більш того, це має тенденцію іти вниз по спіралі. Тобто, якщо код важко зрозуміти, то зміни не будуть виконані правильно. В кінцевому підсумку такий додаток перетвориться в жахливу, незрозумілу велику “кулю бруду”.

Великий розмір програми буде також сповільнювати її розвиток. Чим більше додатків, тим довший час його розгортання та запуску. Зазвичай, запуск таких додатків займає від 12 до 40 хвилин. Якщо розробники регулярно повинні перезапускати сервер, то значну частину дня буде витрачено на очікування, і їх продуктивність від цього буде страждати. Ще одна проблема, пов’язана з великим і складним монолітним додатком, полягає в тому, що така архітектура є значною перешкодою для безперервного розгортання. Це дуже важко зробити зі складним монолітом, так як потрібно перерозгорнути весь додаток, щоб оновити будь-яку одну з його частин. Тривалий час запуску, про що здавалось раніше, не дозволяє робити це часто. Крім того, оскільки вплив зміни, як правило, не відразу зрозумілий, цілком ймовірно, потрібно провести велике ручне тестування такої системи. Отже, безперервне розгортання майже неможливо зробити[2,3].

Монолітні додатки також важко масштабувати, коли різні модулі мають суперечливі потреби в ресурсах. Наприклад, один модуль може реалізовувати CPU-інтенсивну логіку обробки зображень і, в ідеалі, розгорнутий у вигляді екземпляру Amazon EC2 Compute Optimized. Інший модуль може бути в

оперативній пам'яті БД і найкраще підходить для екземплярів EC2 Memory-optimized. Однак, оскільки ці модулі будуть розгорнуті разом, потрібно йти на компроміс при виборі апаратного забезпечення.

Ще однією проблемою монолітних додатків є їх надійність. Оскільки всі модулі працюють в межах одного ж процесу, помилка в будь-якому модулі, така як втрата пам'яті, потенційно може звести на нівець весь процес. Крім того, оскільки всі екземпляри додатку ідентичні, то помилка буде впливати на доступність всієї програми.

І останнє, що також досить важливе, монолітні додатки роблять процес впровадження нових фреймворків та мов програмування надзвичайно важким, майже неможливим. Наприклад, існує 2 мільйони рядків коду, написаного з використанням рамок XYZ. Було б надзвичайно дорого (оцінюючи час і вартість) переписати весь додаток, щоб використовувати нову структуру, навіть якщо ця структура буде значно кращою. В результаті, існує величезний бар'єр для впровадження нових технологій. Тобто існує сталість та незмінність будь-якої технології, вибір якої було зроблено на початку проекту.

Виділимо найбільш суттєві недоліки монолітного додатку:

- Обмежена гнучкість. Будь-яка найменша зміна в додатку приводить до його перерозгортання на сервері. Це означає, що коли потрібно змінити тільки малу частину функціоналу додатку, прийдеться повністю перезбирати проект, генерувати веб-архів та розгортати, хоча інші частини і функціонал програми в цілому не зміняться. Тобто розробникам потрібно буде чекати, поки це відбудеться, щоб побачити вплив цих дрібних змін та кінцевий результат функціонування додатку. В більшості випадків це непотрібно, але в моноліті виникає ситуація, що всі функції додатку тісно зв’язані між собою. Через це виникають простої у роботі команди, що зменшує її продуктивність та частоту додавання нових функцій.

- Неможливість неперервного впровадження на стороні клієнта. Знову ж таки випливає із повільного розгортання архіву на сервері та постійного перезбирання при додаванні нового функціоналу. Особливо це актуально у мобільних додатках, де користувачі очікують швидкого впровадження нових функцій, без втрати можливості користування додатком.
- Сталість стеку технологій. Технології, які будуть застосовуватись у такому додатку оцінюються та вибираються ще до початку розробки. У всій команді повинні використовуватись однакові технології, сталі сховища даних, однакові інструменти для розробки. Але не завжди раціонально застосовувати один і той же стек технологій на всьому проекті. Часто виникає необхідність змінити певні з них посеред процесу розробки.
- Технічні погрішності. “Працює - не чіпай!” - найбільш поширена методологія при розробці програмного забезпечення, яка є дуже актуальною при написанні монолітного додатку, але не зовсім правильна. Так, це зручно і дозволяє підтримувати додаток у робочому стані. Але це призводить до поганої архітектури або погано написаного програмного коду, який може використовуватись у найбільш несподіваних місцях та дуже “цікавими” способами. Ентропія такої системи з часом зростає, якщо вчасно не переробити її. Зазвичай, погрішності в таких програмах накопичуються, команда змінюється і систему стає майже неможливо підтримувати, а також змінювати.

Підводячи підсумок: існує успішний бізнес-додаток, який виріс в жахливий моноліт, який далеко не всі, якщо такі є, розробники розуміють. Він написаний з використанням застарілої, непродуктивної технології, що робить найм талановитих розробників важким. Додаток важко масштабувати і він є ненадійним. В результаті, гнучка розробка і впровадження додатків неможлива[2,3].

1.3. ГОЛОВНІ ІДЕЇ МІКРОСЕРВІСІВ

Багато компаній, такі як Amazon, eBay і Netflix, вирішили проблеми пов’язані із монолітною архітектурою, прийнявши рішення, яке відоме зараз під назвою MSA. Замість того щоб будувати один цілісний, великий, монолітний додаток, проводиться декомпозиція завдань, та система розбивається на безліч дрібніших, пов'язаних між собою сервісів.

Сервіс, як правило, реалізує безліч різних функцій або функціональних можливостей, таких як управління замовленнями, управління клієнтами і т.д. Кожен мікросервіс є міні-додатком, який має свою власну архітектуру, що, в свою чергу, реалізує бізнес-логіку, а також надає інтерфейс для взаємодії з ним. Тобто мікросервіси надають свій API, який використовується іншими мікросервісами або клієнтами програми. Також мікросервіси можуть реалізовувати веб-інтерфейс. Під час запуску, кожен мікросервіс розгортається в хмарі на VM або контейнері Docker[2,3].

Наприклад, декомпозиція системи, яка згадувалась раніше показана на наступній схемі (рис.1.2):

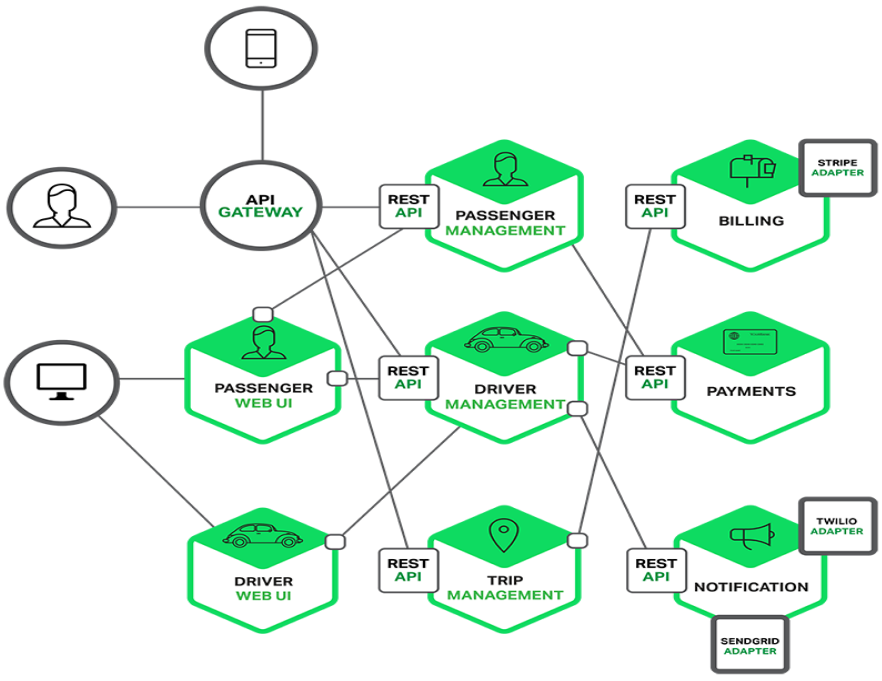

Рисунок 1.2 – Мікросервісна архітектура додатку для виклику таксі

Кожна функціональна сфера додатку в даному випадку реалізується своїм власним мікросервісом. Крім того, веб-додаток розбивається на низку більш простих веб-додатків. Це полегшує розгортання враховуючи потреби конкретних користувачів, пристроїв або спеціалізованих випадків використання.

Кожна сервіс надає REST API який використовують інші сервіси для взаємодії з ним. Наприклад, Driver Management використовує сервер сповіщень, щоб повідомити доступного водія про потенційну поїздку. Сервіси UI посилаються на інші сервіси, для того, щоб надавати веб-сторінки для користувача. Сервіси можуть також використовувати асинхронний, обмін повідомленнями для міжпроцесної комунікації.

Деякі REST API піддаються впливу мобільних додатків, які використовуються водіями і пасажирами (як у прикладі, який розглядається). Такі додатки не мають прямого доступу до внутрішніх сервісів. Замість цього вони використовують посередника, який відомий під назвою API Gateway. Він відповідає за такі завдання, як балансування навантаження, кешування, контроль доступу, облік API і моніторинг (рис.1.3).

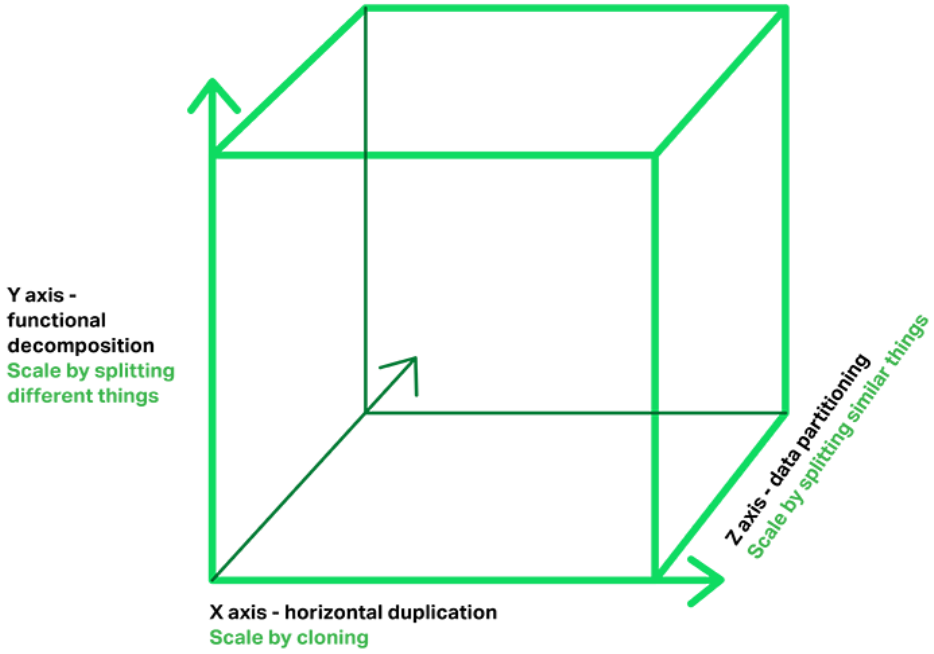

Рисунок 1.3 – Куб масштабування додатку

MSA дозволяє масштабування по осі Y. На рисунку зображено модель куба, який являє собою 3D-модель масштабованості. Інші дві осі - це вісь X, яка показує можливість запуску кількох ідентичних копій програми, що дозволяє балансувальник навантаження і вісь Z яка показує поділ даних між мікросервісами[2,3].

Всі додатки зазвичай використовують три типи масштабування одночасно. Масштабування по осі Y ділить додаток на мікросервіси, як було показано вище в цьому розділі. При масштабуванні по осі X відбувається запуск кількох екземплярів кожного сервіса використовуючи балансувальник навантаження для потрібної пропускної здатності та доступності. Деякі додатки можуть також використовувати масштабування по осі Z для поділу послуг. На наступному рисунку (рис.1.4) показано, як сервіс управління поїздкою може бути розгорнутий з використанням Docker на Amazon EC2 (приклад, який розглядається у даному розділі):

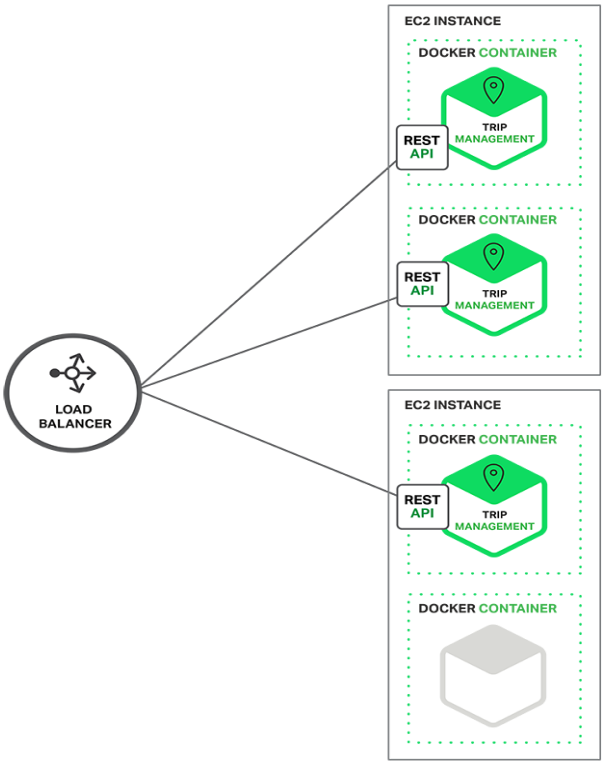

Рисунок 1.4 – Розгортання сервісу управління поїздкою з використанням Docker

Під час виконання, сервіс Trip Management має декілька екземплярів. Кожен екземпляр сервісу є контейнером Docker. Для того, щоб постійно бути доступними, контейнери працюють на декількох хмарних VM одночасно. Всі сервіси обов'язково зв'язані із балансувальником навантаження, який розподіляє запити між екземплярами. Балансувальник навантаження також може обробляти і інші проблеми, такі як кешування, контроль доступу, облік API і моніторинг.

MSA істотно впливає на взаємозв'язок між додатком і БД. Замість того, щоб ділити одну БД між сервісами, кожна з них має свою власну БД. З одного боку, такий підхід суперечить ідеї моделі даних в масштабах підприємства. Крім того, це часто призводить до дублювання деяких даних. Однак, мати БД для кожної послуги є важливим, якщо потрібно отримати вигоду від використання мікросервісів, оскільки такий підхід забезпечує слабку зв'язність. На наступному рисунку показана архітектура бази даних для прикладу, який розглядається (рис.1.5):

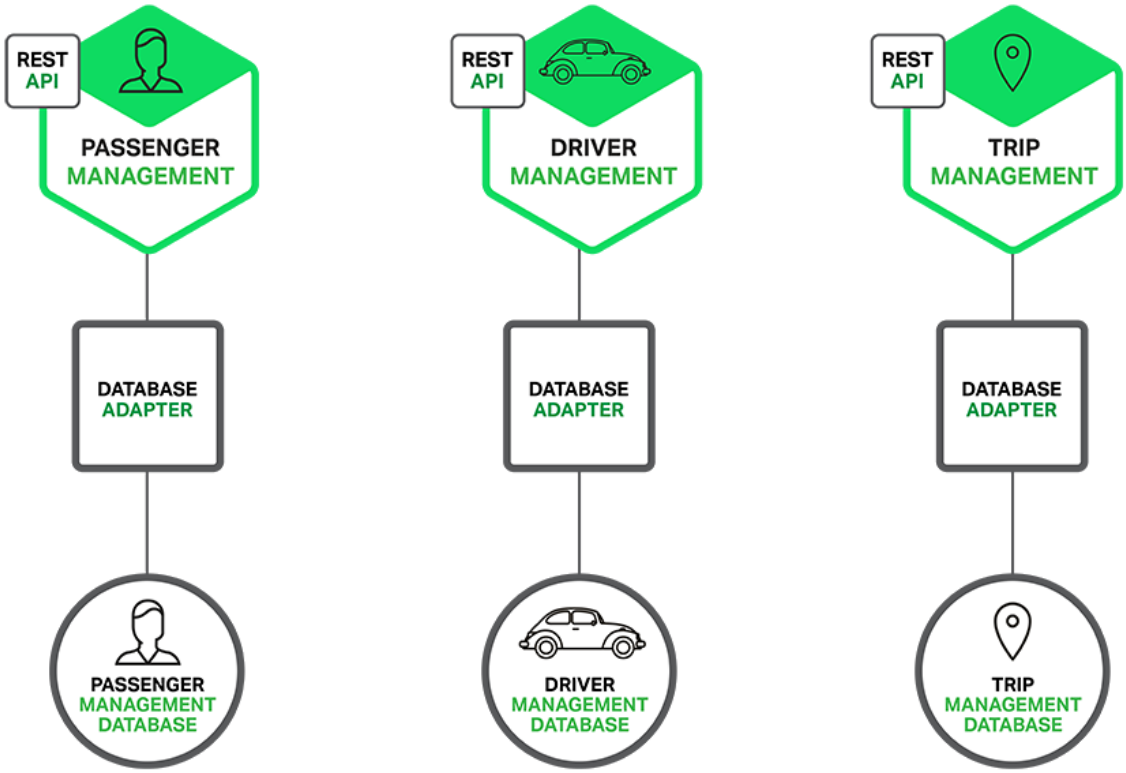

Рисунок 1.5 – Окрема база даних для кожного мікросервіса

Кожен сервіс має свою власну базу даних. Крім того, сервіс може використовувати тип бази даних, яка найкраще підходить для його потреб, тобто з’являється можливість багатомовності. Наприклад, сервісу Driver Management, який знаходить водіїв, які близькі до потенційного пасажира, необхідно використовувати базу даних, яка підтримує ефективні гео-запити[2,3].

На перший погляд MSA дуже схожа на SOA. В обох підходах, архітектура складається з набору сервісів. Мікросервісні додатки віддають перевагу більш простим та легким протоколам, таким як REST, а не WS-*. Вони також уникають потреби використання ESB, замість цього реалізовують ESB-подібну функціональність в самих мікросервісах. MSA також відкидає і інші канонічні поняття SOA. Саме тому не можна говорити, що це одне і те ж саме. Швидше MSA є окремим випадком SOA, як зображено на наступному рисунку (рис.1.6):

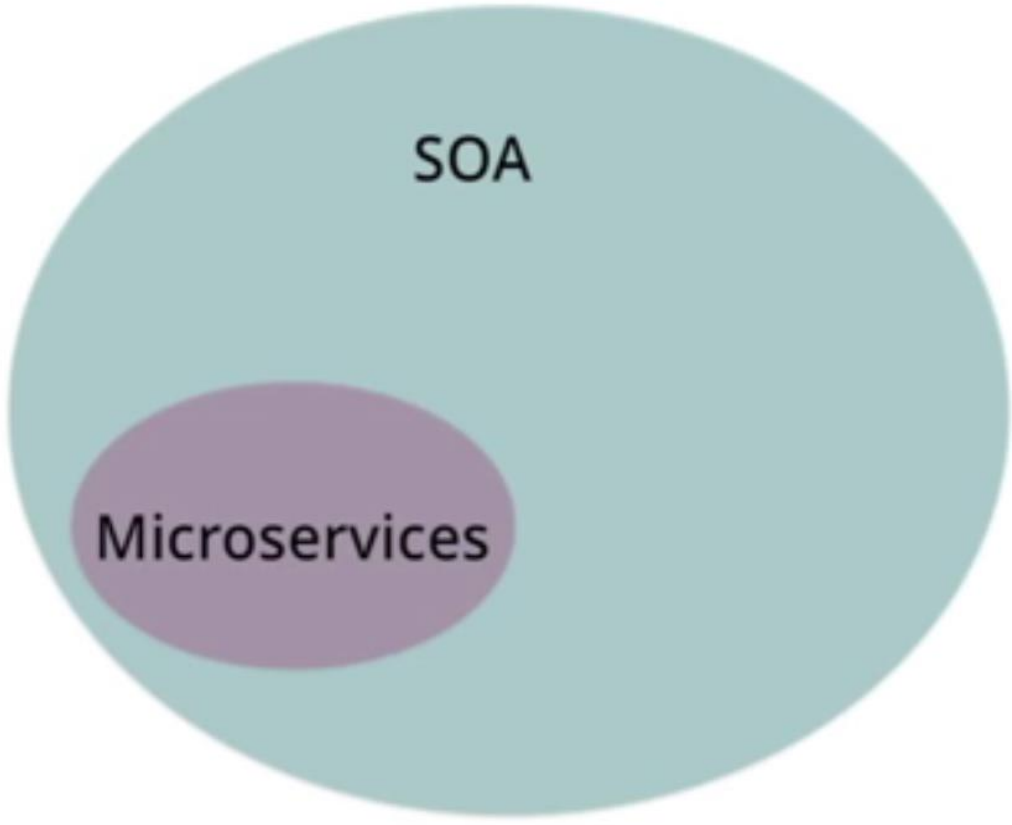

Рисунок 1.6 – Мікросервісна архітектура в контексті SOA

1.4. ПЕРЕВАГИ МІКРОСЕРВІСІВ

Підхід з використанням MSA має ряд важливих переваг. По-перше, він вирішує проблему складності. За допомогою нього громіздкий монолітний додаток може бути розбитий на набір незалежних сервісів. У той час як загальна функціональність системи не змінюється, вона буде розбита на частини, якими дуже легко керувати та підтримувати. Кожна служба має має чіткий функціонал, який керується за допомогою віддаленого виклику процедур (RPC) або повідомленнями API.

MSA забезпечує дотримання модульності, чого на практиці дуже важко досягти в монолітній архітектурі. Отже, окремі сервіси можуть набагато швидше розвиватися, і вони набагато легші для розуміння та підтримки.

По-друге, ця архітектура дозволяє розробляти кожний сервіс незалежними командами, які працюють над ними. Розробники можуть обирати будь-які технології, які є найбільш доречними, але при умові, що сервіс надає коректний API, для взаємодії з ним. Звісно, більшість організацій захоче уникнути повного безладу ввівши обмеження стеку технології. Тим не менше, така своєрідна свобода для розробників робить непотрібним використання технології, що, можливо, вже застарілі, адже вони існували із самого початку проекту. Під час написання нового сервісу, вони мають можливість використовувати актуальні зараз, сучасні технології. Більше того, оскільки сервіси відносно малі,редагування їх чи внесення нового функціоналу стає значно продуктивнішим та швидшим[3,4].

По-третє, MSA дозволяє розробляти кожен мікросервсіс окремо та незалежно. В розробників ніколи не постане необхідність глобальної координації розробки змін на їх локальному сервісі. Такі зміни можна впроваджувати одразу після тестування.

MSA надає можливість безперервного розгортання. Також цей підхід надає можливість кожному сервісу масштабуватися незалежним чином. Можна

розгортати ту кількість екземлярів сервісу, яка задовольнить потребу бізнес-завдання. Крім того, можна використовувати те обладнання, яке відповідає кращим потребам у ресурсах для певного сервісу.

Тобто можна виділити наступні основні переваги MSA:

- Легші для розвитку, розуміння та підтримки: код в мікросервісі обмежується однією функцією бізнесу і, таким чином, легкий для розуміння. В IDE може бути завантажено невелику частину коду, що є дуже просто та дозволяє підвищити продуктивність розробників.
- Швидший запуск, в порівняннні з монолітом. Об’єм кожного мікросервісу набагато менший, ніж моноліт, і це призводить до меншого вихідного архіву. В результаті, розгортання і запуск відбуваються набагато швидше, що знову ж таки підвищує продуктивність розробників.
- Легке розгортання локальних змін. Кожен сервіс може бути розгорнутий незалежно від інших сервісів. Будь-яка локальна зміна в сервісі може бути легко зроблена розробником і не потребувати комунікацій з командами, які розробляють інші сервіси. В результаті, це надає гнучкості мікросервісам, в порівнянні з монолітом, а також дозволяє легко впровадити CI/CD.
- Незалежне масштабування. Кожен сервіс може незалежно масштабуватися по осі X та бути поділеним по осі Z, якщо це потрібно. Це дуже відрізняється від монолітних додатків, які мають дуже багато різних вимог, але все рівно повинні бути розгорнуті одночасно.
- Краща ізоляція збоїв. Збій сервісу, наприклад втрата пам’яті чи незакриті з’єднання із БД, буде впливати тільки на даний сервіс, на відміну від цілісного монолітного додатку. Це покращує ізоляцію збою і дозволяє не завершувати роботу всього додатку, а лише перезапустити сервіс, де виник цей збій.

- Немає прив’язки до однієї технології. Розробники можуть вільно вирішувати та обирати технології, які краще підходять для реалізації функціоналу того чи іншого сервісу. Також в будь-який момент можна безболісно перейти на ту чи іншу технологію. Це дає свободу та доцільність вибору стеку технологій та можливість іти в ногу з трендами[3,4].

1.5. НЕДОЛІКИ МІКРОСЕРВІСІВ

Здається, що мікросервіси - це вирішення усіх проблем зв’язаних із монолітом. Але, насправді, їх використання створює низку нових труднощів, які, в першу чергу, зв’язані із тим, що MSA по своїй суті є розподіленою системою.

Це дуже важливо, оскільки тепер один монолітний додаток розділений між кількома мікросервісами і вони повинні комунікувати один з одним. Кожен мікросервіс може використовувати іншу платформу, стек технологій, сховище даних і, таким чином, буде мати різні вимоги до моніторингу та управління. Кожен сервіс може з часом незалежно бути масштабованим по осі X і Z. Кожен сервіс може перерозгортатись кілька разів протягом дня.

Одним з недоліків є сама назва. Термін “мікросервіс” ставить надмірний акцент на розмірі послуг. Насправді, існують деякі розробники, які виступають за розробку тільки дуже дрібних (10-100 рядків коду) сервісів. У той час, як невеликі сервіси є кращими, важливо пам'ятати, що вони - це лише засіб для досягнення мети, а не основна мета. Метою мікросервісів є те, щоб провести декомпозицію додатку, щоб зробити процес розробки і розгортання додатків більш гнучким.

Іншим суттєвим недоліком мікросервісів, як було сказано вище, є складність, яка випливає з того факту, що MSA є розподіленою системою. Розробники повинні вибрати і впровадити механізм взаємодії між процесами на

основі обміну повідомленнями або RPC. Крім того, потрібно реалізувати засоби обробки часткових збоїв, тому що адресат запиту може бути досить повільним або взагалі недоступним. Тому в цьому плані це набагато складніше, в порівнянні з монолітним додатком, де модулі посилаються один на одного за допомогою мови програмування на рівні методів/викликів процедур.

Ще одна проблема, зв’язана з мікросервісами, - це розподілена архітектура бази даних. Бізнес-операції, які оновлюють кілька суб'єктів, є досить поширеним явищем. Такі транзакції тривіальні для реалізації в монолітному додатку, оскільки існує єдина база даних. При виникненні таких випадків в MSA, необхідно оновлювати кілька баз даних, що належать різним сервісам. Використання розподілених транзакцій, як правило, не підходить. Вони просто не підтримуються багатьма з сучасних високомасштабованих NoSQL баз даних та брокерів обміну повідомленнями. В кінцевому підсумку доводиться використовувати логічно-базований підхід, який є більш складним завданням для розробників.

Тестування мікросервісного додатку також набагато складніше завдання. Наприклад, за допомогою сучасних фреймворків, наприклад Spring Boot, реалізувати тестовий клас, який застосовується для монолітного веб-додатку і тестує його REST API, є тривіальною задачею. На противагу цьому, подібний тестовий клас для сервісу повинен буде запустити цей сервісі, будь-які сервіси, від яких він залежить, чи принаймні налаштувати заглушки для цих сервісів. Знову ж таки, не потрібно недооцінювати складність цього.

Ще одна серйозна проблема зв’язана з MSA - це реалізація зміни, яка охоплює декілька сервісів. Для прикладу візьмемо реалізацію архітектури, яка вимагає внесення змін в сервіси A, B, і C, де A залежить від B і B залежить від C. У монолітному додатку можна просто змінити відповідні модулі, інтегрувати зміни, і відразу розгорнути їх. На противагу цьому, при використанні MSA, необхідно ретельно планувати і координувати розгортання змін для кожного сервісу окремо. Тобто потрібно буде оновити сервіс C, а потім сервіс B, а потім,

нарешті, сервіс А. Але у більшості типових випадків зміни відносяться тільки до одного сервісу: мультисервісні зміни, які вимагають ретельної координації дій - це рідкість.

Розгортати додатки на основі мікросервісів також набагато складніше. Монолітний додаток просто розгортається на безлічі ідентичних серверів. Інфраструктура (тобто бази даних і брокер повідомлень) кожного екземпляру такого додатку налаштовується відразу і це не викликає додаткових труднощів. На відміну від цього, мікросервісний додаток, як правило, складається з великої кількості послуг.

Кожен сервіс матиме кілька екземплярів під час виконання. Це додаткова кількість елементів, які повинні бути налаштовані, розгорнуті, масштабовані і контрольовані. Крім того, також необхідно впровадити механізм пошуку серівісів, який дозволяє їм виявляти місце розташування (хости і порти) будь-яких інших сервісів, з якими вони повинні комунікувати. Традиційний ручний підхід не підходить для системи такого рівня складності. Отже, успішне розгортання мікросервісної програми вимагає більшого контролю розробниками і високого рівня автоматизації.

Один з підходів автоматизації є використання PaaS, наприклад Cloud Foundry. PaaS надає розробникам простий спосіб розгортання і управління їх мікросервісами. Він дозволяє уникнути проблем, таких як добування і налаштування IT-ресурсів.

Інший спосіб автоматизувати розгортання мікросервісів полягає в розробці своєї власної PaaS. Типовою стартовою точкою для цього є використання кластерного рішення, наприклад, Kubernetes, в поєднанні з технологією контейнерів Docker[3,4].

1.6. ВИСНОВКИ

Побудова складних комплексних додатків по своїй суті є задачею непростою. Застосування монолітної архітектури має сенс тільки при створенні не дуже масштабних та легковісних рішень. Якщо все ж застосовувати цей підхід для створення великих систем, в кінцевому підсумку з’явиться багато проблем, які буде неможливо вирішити.

Незважаючи на низку недоліків, MSA є найкращим вибором для реалізації складних додатків, які в подальшому будуть активно та швидко розвиватись та розширюватись. У наступних розділах буде більш детально розглянуто різних аспекти мікросервісного підходу створення складних систем.

2. ШАБЛОНИ ПРОЕКТУВАННЯ МІКРОСЕРВІСІВ

Функціональна декомпозиція додатка та команди, яка розробляє його, є ключем до створення успішної MSA. Це дозволяє досягти слабкої зв'язності (REST інтерфейсів) та високої згуртованості (декілька менших сервісів можуть визначати один більший композитний вищий за рівнем сервіс або додаток).

Функціональна декомпозиція дає швидкість, гнучкість, масштабованість і інші переваги, але бізнес метою, як і раніше, залишається створення ефективної програми. Тому постійно потрібно вирішувати, як же скомпонувати створені мікросервіси для забезпечення цієї ефективності та потрібного функціоналу.

В цьому розділі буде описано рекомендовані шаблони компонування мікросервісів для створення ефективної MSA.

2.1. АГРЕГАТОР (AGGREGATOR)

Найпершим і, напевно, найбільш поширеним є такий шаблон проектування мікросервісів, як агрегатор.

У найпростішій формі, агрегатор буде звичайною веб-сторінкою, яка викликає декілька служб для досягнення функціональності, яка вимагається додатком. Кожен із сервісів (Service A, Service B та Service C) надає легковісний REST, за допомогою якого веб-сторінка може отримати дані та обробити/відобразити їх відповідним чином. Якщо якийсь із видів обробки вимагає, скажімо, застосування бізнес-логіки для даних, отриманих від індивідуальних сервісів, то, ймовірно, це буде CDI компонент, який буде перетворювати дані таким чином, щоб їх можна було коректно відобразити на веб-сторінці (рис.2.1).

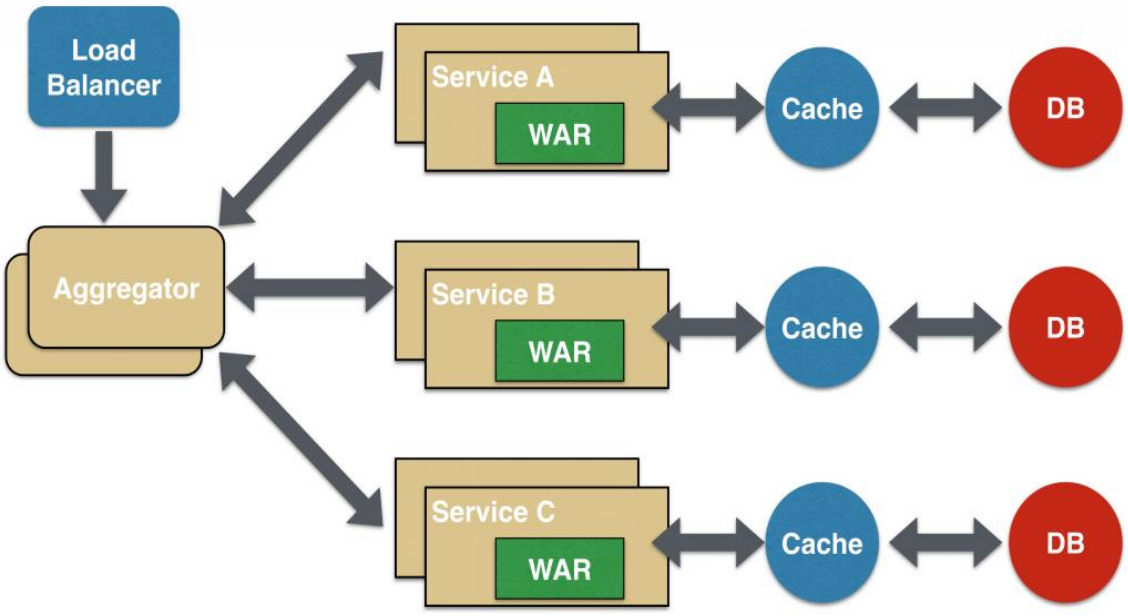

Рисунок 2.1 – Шаблон «Агрегатор»

Інший варіант шаблону Агрегатор, коли не потрібно відображати дані, а замість цього потрібно просто створити композитний мікросервіс, який буде використовуватись іншими сервісами. В цьому випадку агрегатор повинен зібрати дані від інших мікросервісів, застосувати бізнес-логіку до них та, в подальшому, опублікувати їх в якості кінцевої точки REST. Ці дані потім можуть бути використані іншими сервісами, які їх потребують.

Цей шаблон проектування слідує принцип DRY. Якщо існує кілька сервісів, яким потрібно отримати доступ до сервісів A, B і C, то рекомендується абстрагувати цю логіку в композитному мікросервісі та агрегувати її в одну службу. Перевагою абстрагування на цьому рівні є те, що створюються окремі сервіси, тобто сервіси A, B, і C, і вони можуть розвиватися незалежно один від одного, та бізнес-потреби, як і раніше, забезпечуються композитним мікросервісом[5].

Потрібно звернути увагу, що кожен окремий мікросервіс має свої власні (як опція) кешування і базу даних. Якщо Агрегатор є композитним мікросервісом, то він також може мати свої власні кешування і базу даних.

Також Агрегатор дуже добре масштабується по X і Z-осі. Тобто, якщо це веб-сторінка, то можна розгорнути додаткові веб-сервери або, якщо це композитний мікросервіс, можна розгорнути додаткові WildFly екземпляри для можливості подальшого росту при необхідності.

2.2. ПРОКСІ (PROXY)

Шаблон проектування мікросервісів Проксі є різновидом шаблону Агрегатор. У цьому випадку немає агрегації, яка вимагається на клієнті, але різні мікросервіси можуть викликатися за потребою бізнес-логіки (рис.2.2).

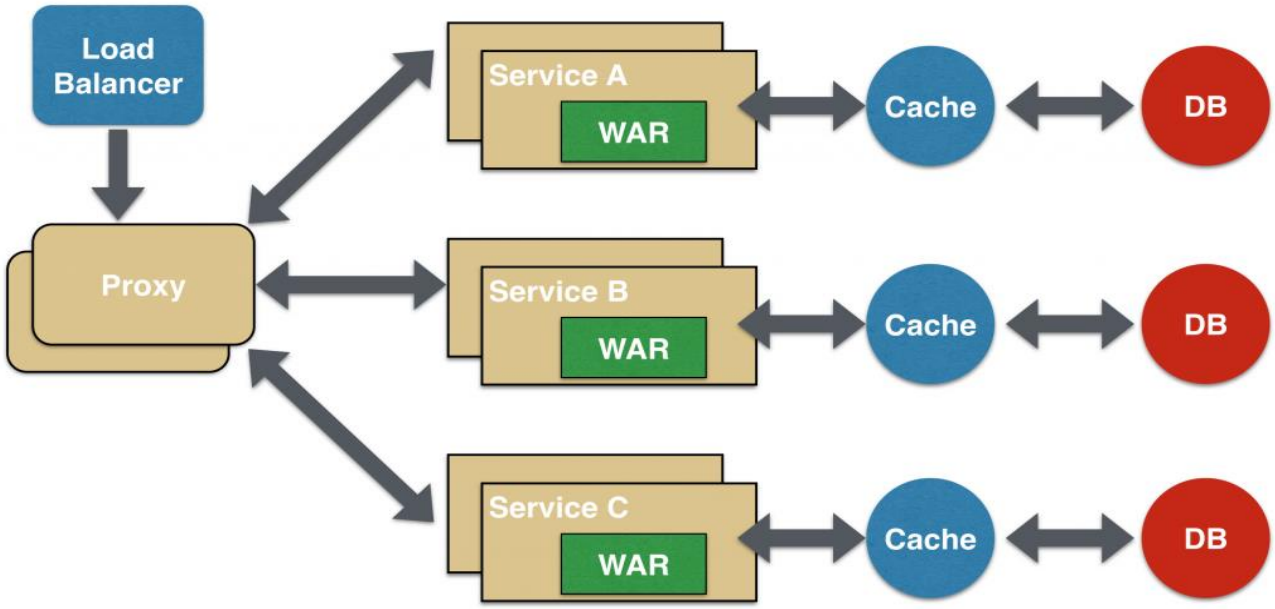

Рисунок 2.2 – Шаблон «Проксі»

Так само, як Агрегатор, Проксі можна масштабувати незалежно один від одного по X і Z-осі. Це можна зробити непомітно для користувача, оскільки кожен окремий сервіс повинен проходити через спеціальний інтерфейс.

Проксі-сервер може бути “простим посередником”, який в цьому випадку делегує запити на один із сервісів. В якості альтернативи він може бути “розумним посередником”, який робить певні перетворення даних, перед подачею їх клієнту. Хорошим прикладом цього може буди рівень представлення даних для різних пристроїв який реалізований, як смарт-проксі.

2.3. ЛАНЦЮГОВИЙ (CHAINED)

Ланцюговий шаблон проектування мікросервісів надає єдину зведену відповідь на певний запит. У цьому випадку запит від клієнта отримано сервісом А, який в свою чергу з'єднаний із сервісом В, який може мати зв'язок з сервісом С. Всі сервіси, ймовірніше всього, використовують синхронний обмін повідомленнями по HTTP (рис.2.3).

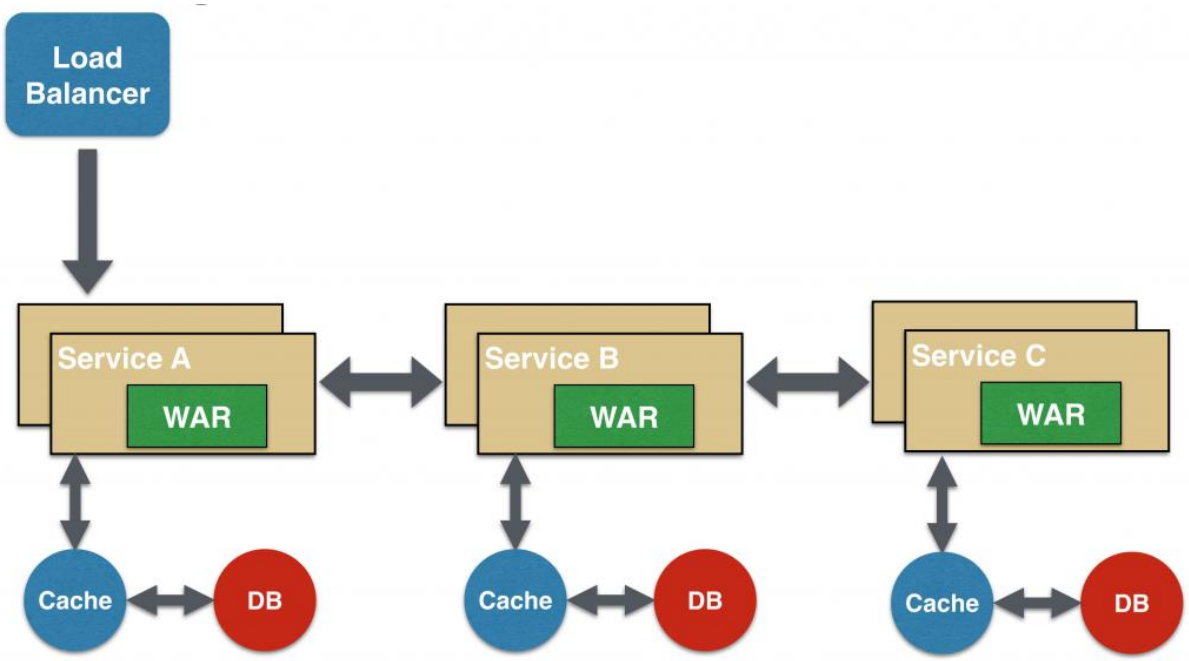

Рисунок 2.3 – Шаблон «Ланцюговий»

Головне, про що потрібно пам'ятати, - це те, що клієнт буде заблокований, поки не закінчиться повний ланцюжок запитів/відповідей між сервісами, тобто Service A<->Service B і Service B<->Service C, буде виконано. Запит від Service B до Service C може дуже відрізнятися від запиту Service A до Service B. Аналогічна ситуація і з відповідями[5].

Ще один важливий аспект полягає у тому, щоб не зробити ланцюг сервісів надто довгим. Це важливо тому, що його синхронна природа змушуватиме довго чекати відповідь на стороні клієнта. Цю проблему дозволяє вирішити наступний шаблон проектування.

Ланцюг з одним мікросервісом називається Singleton ланцюгом. Це дозволяє розширити його на більш пізньому етапі розробки.

2.4. ГІЛКА (BRANCH)

Шаблон проектування мікросервісів Гілка розширює шаблон Агрегатор та надає можливість одночасної обробки відповіді від двох, які не виключають один одного, ланцюгів мікросервісів. Цей шаблон також може бути використаний для виклику різних гілок мікросервісів, грунтуючись на потребах бізнес-логіки (рис.2.4).

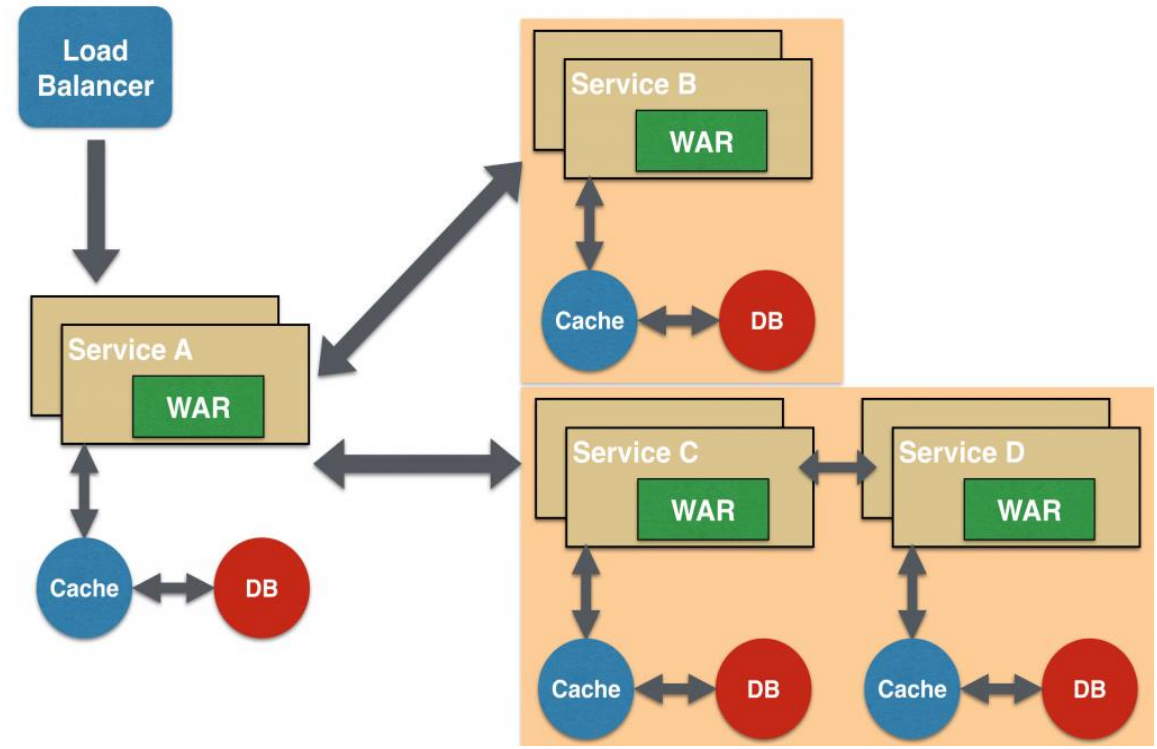

Рисунок 2.4 – Шаблон «Гілка»

Сервіс А, веб-сторінка чи композитний мікросервіс може викликати дві різні гілки одночасно, тоді це буде нагадувати шаблон проектування Агрегатор. В якості альтернативи, Сервіс А може викликати тільки одну гілку мікросервісів, яка вказана у запиті, отриманому від клієнта[5].

Це може бути налаштовано за допомогою маршрутизації JAX-RS і потребувати динамічного налаштування.

2.5. ДАНІ СПІЛЬНОГО ВИКОРИСТАННЯ (SHARED DATA)

Однією з конструктивних особливостей мікросервісів є їх автономність. Це означає, що кожен сервіс є повноцінним і має контроль над усіма компонентами - UI, проміжне програмне забезпечення, транзакції. Це дозволяє сервісу бути багатомовним і використовувати потрібні інструменти для коректної роботи. Наприклад використання NoSQL бази даних замість звичайної SQL.

Однак, типова проблема в такій ситуації, особливо при рефакторингу монолітної архітектури в мікросервісну, - це нормалізація бази даних. Кожен сервіс повинен отримати дані, які йому потрібні - не більше і не менше. Навіть, якщо в додатку використовується тільки база даних SQL - перехід мо мікросервісів призведе до її денормалізації, тобто виникне дублювання даних, можливо навіть конфлікти. В таких випадках потрібно використовувати шаблон даних спільного використання.

У цьому шаблоні деякі мікросервіси, подібно ланцюговому шаблону, можуть спільно використовувати кешування та сховища даних. Це матиме сенс тільки при наявності сильного зв’язку між двома сервісами. В багатьох випадках, це можна назвати антишаблоном, але потреби бізнес-логіки інколи вимагають його використання (рис.2.5).

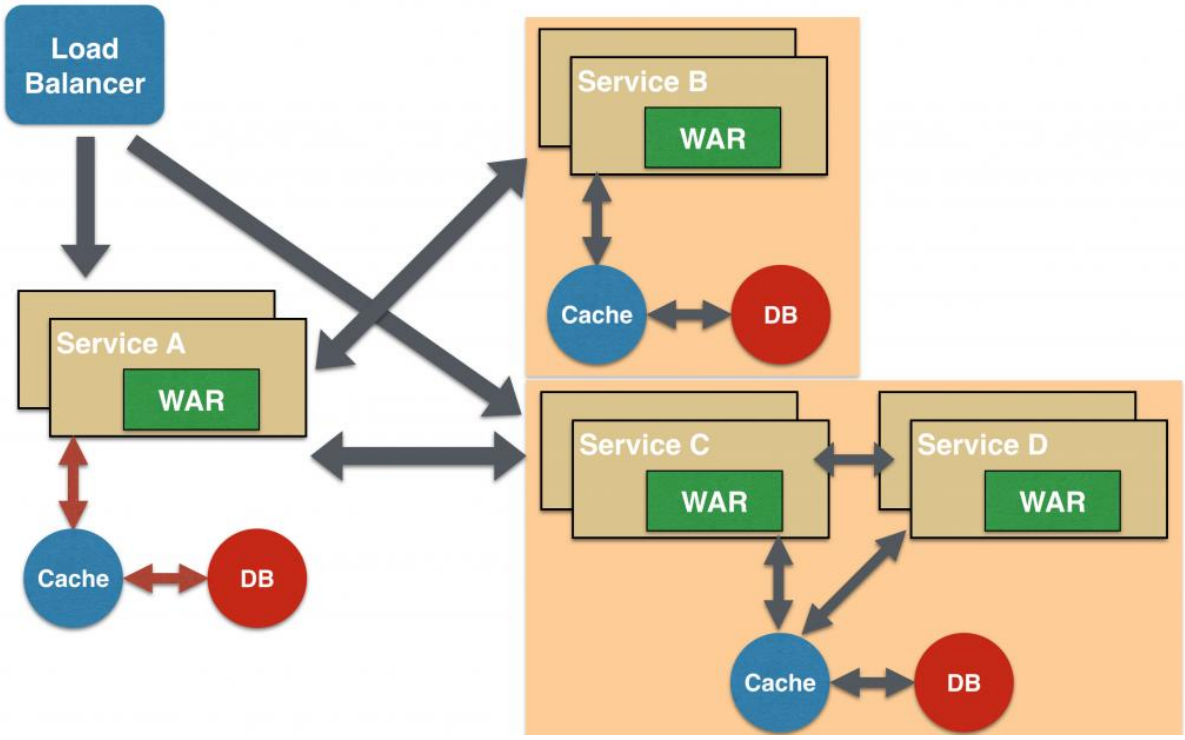

Рисунок 2.5 – Шаблон «Дані спільного використання»

Такий шаблон можна також використовувати, як перехідний варіант від монолітної архітектури до мікросервісної, в якій всі сервіси є повністю незалежними[5].

2.6. АСИНХРОННИЙ ОБМІН ПОВІДОМЛЕННЯМИ (ASYNCHRONOUS MESSAGING)

У той час як використання REST дуже поширене та добре зрозуміле, він накладає обмеження на те щоб бути синхронним та неблокуючим. Асинхронність може бути досягнена, але тільки специфічним прикладним шляхом. У деяких випадках мікросервісні архітектури можуть використовувати черги повідомленнь замість запитів/відповідей у REST (рис.2.6).

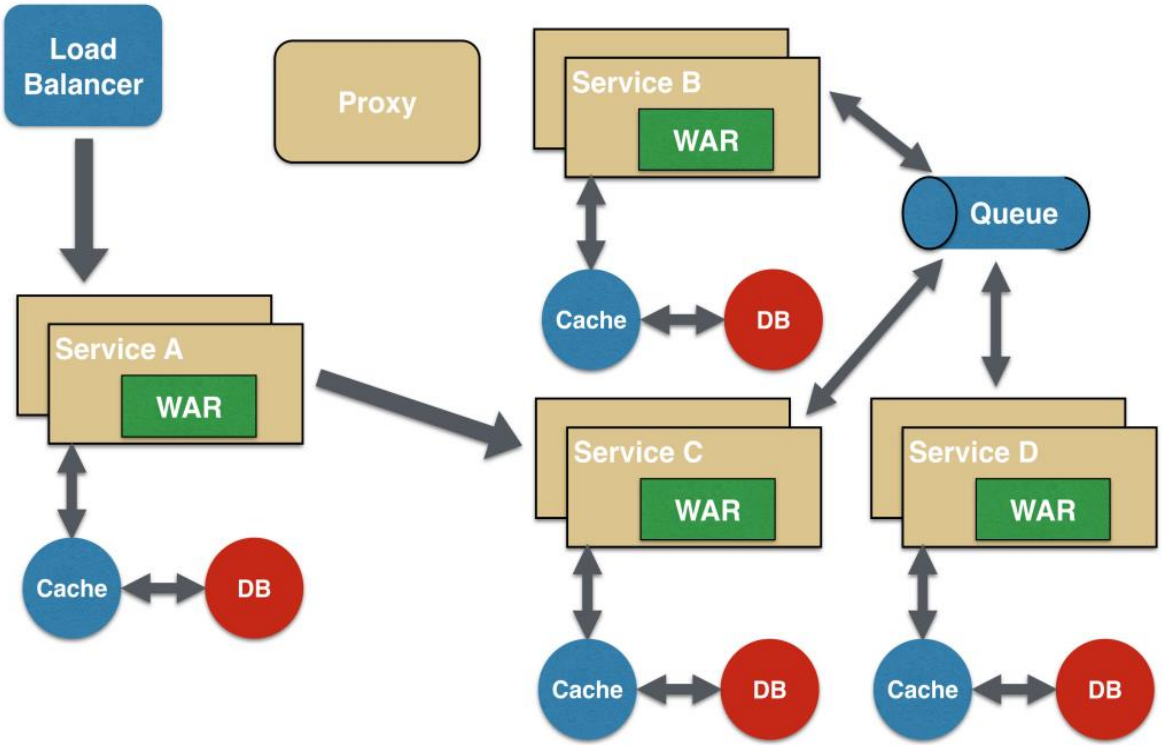

Рисунок 2.6 – Шаблон «Асинхронний обмін повідомленнями»

У цьому шаблоні проектування, Service A може викликати Service C синхронно, який в свою чергу зв'язаний з Service B і D в асинхронному режимі з використанням загальної черги повідомлень.

Зв’язок Service A -> Service C може бути асинхронним, можливо, з використанням WebSockets, для досягнення бажаної масштабованості.

Поєднання запитів/відповідей у REST із чергами повідомлень може бути використано для реалізації завдань бізнес-логіки.

2.7. ВИСНОВКИ

У даному розділі було розглянуто основні шаблони проектування, які використовуються при створенні додатків з мікросервісною архітектурою. Кожен з шаблонів було детально описано проілюстровано їх роботу та надано принцип реалізації.

3. КЕРУВАННЯ ТЕРМІНАМИ ВИКОНАННЯ ДИПЛОМНОЇ РОБОТИ. КЕРУВАННЯ РИЗИКАМИ

Тема: «Розробка лабораторного практикуму з дисципліни «Основи сервіс-орієнтованих обчислень і архітектур»»

Об’єкт дослідження – засоби створення мікросервісної архітектури.

Предмет дослідження – використання засобів створення мікросервісної архітектури для створення прикладу додатку для лабораторного практикуму.

Задачі:

1. Дослідження засобів для створення мікросервісної архітектури
2. Створення та запуск тестового прикладу додатку
3. Створення методичних вказівок для лабораторного практикуму

3.1 КЕРУВАННЯ ТЕРМІНАМИ ВИКОНАННЯ ДИПЛОМНОЇ РОБОТИ

Діаграма Ганта (рис.3.1):

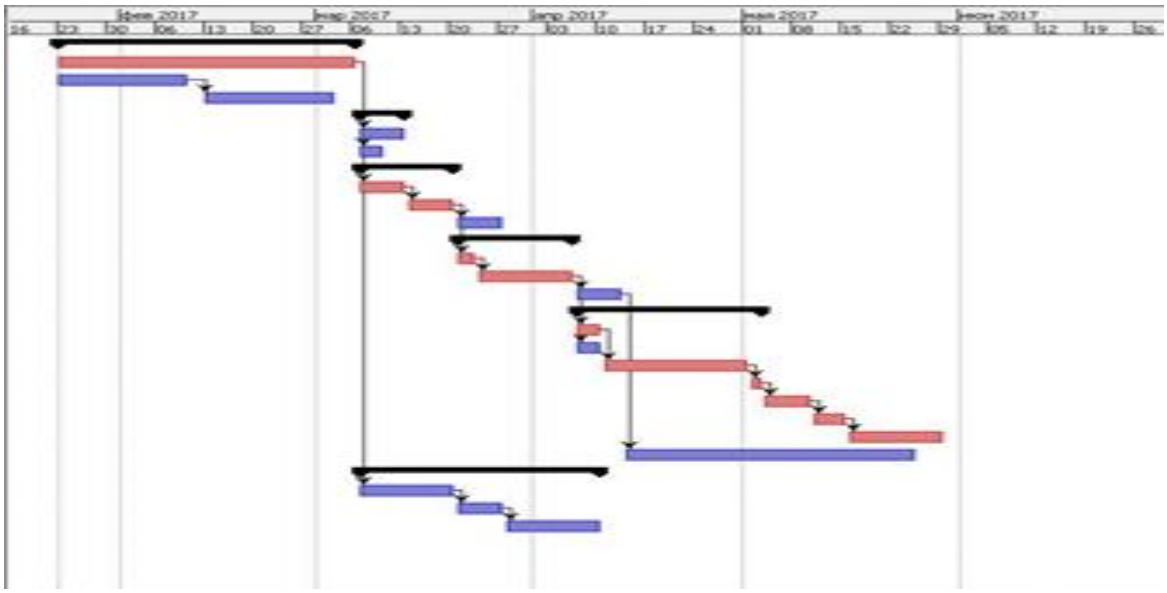

Рисунок 3.1 – Діаграма Ганта по процесу роботи над дипломним проектом

План роботи:

1. Аналіз літератури
   1.1.Читання основних джерел
   1.2.Пошук додаткових джерел
   1.3.Опрацювання знайдених джерел
2. Робота над першим розділом
   2.1.Розбір та аналіз мікросервісної архітектури
   2.2.Оформлення протоколу першого розділу
3. Робота над практичною частиною
   3.1.Вибір базового фреймворку
   3.2.Створення простого мікросервісного додатку
   3.3.Розгортання додатку на локальному комп’ютері
4. Порівняння відомих фреймворків
   4.1. Пошук рішень для порівняння
   4.2. Вибір критеріїв
   4.3.Порівняння за обраними критеріями
5. Написання методичних вказівок
   5.1. Пошук доступної інформації
   5.2. Вибір можливих рішень
   5.3. Написання ЛР1
   5.4. Написання ЛР2
   5.5. Написання ЛР3
   5.6. Написання ЛР4
   5.7. Повторне проходження пунктів лабораторних робіт
   5.8. Написання висновків
6. Підготовка до захисту
   6.1.Оформлення висновків
   6.2.Презентація
   6.3.Підготовка до захисту

Оцінка тривалості робіт (оптимістична (**О)**– песимістична (**Р)** – реалістична (**М)**).

1) Читання основних джерел літератури – 40 – 31 – 26 днів
2) Аналіз методів побудови мікросервісних систем – 8– 5 – 3 днів
3) Дослідження даних методів – 8 – 5 – 3 днів
4) Визначення завдань, до яких будуть застосовані методи – 8 – 5 – 2 днів
5) Побудова алгоритмів – 15 – 10 – 5 днів
6) Проектування ПЗ – 3 – 2 – 1 днів
7) Розробка – 20 – 15 – 10 днів
8) Тестування – 4 – 2 – 1 днів
9) Оформлення лабораторного практикуму – 7 – 5 – 3 днів
10) Написання висновків – 4 – 3 – 2 днів
11) Створення плакатів та презентації – 12 – 10 – 5 днів

Таблиця 3.1 – Оцінка параметрів дипломної роботи

№	Пес.	Реал.	Оптим.	Дисп.	Сер.ст.от.	Те
1	41	31	26	37,5	6,123724	31,83333
2	8	5	3	4,166667	2,041241	5,166667
3	8	5	3	4,166667	2,041241	5,166667
4	8	5	2	6	2,44949	5
5	15	10	5	16,66667	4,082483	10
6	3	2	1	0,666667	0,816497	2
7	20	15	10	16,66667	4,082483	15
8	4	2	1	1,5	1,224745	2,166667
9	7	5	3	2,666667	1,632993	5
10	4	3	2	0,666667	0,816497	3
11	12	10	5	8,166667	2,857738	9,5
∑	**130**	**93**	**61**	**98,83333**	**28,16913**	**93,83333**

Z = 0,02958321; F(z) = **0.51**

Відповідно, із вірогідністю 51% проект буде завершений за 93 дні

3.2 КЕРУВАННЯ РИЗИКАМИ

Таблиця 3.2 – Таблиця ризиків проекту

Ризик	Подія	Вірогідність	Вплив
Поломка персонального комп'ютера	Збереження всіх даних в хмарі	Низька	Сильний
Зміна правил оформлення дипломної роботи, додавання нових вимог до розділів (самих розділів)	Бути готовим морально. Використання засобів редактора, щоб при внесенні змін не виникало проблем	Середня	Сильний
Зміна ліцензій фреймворків, які використовуються при розробці	Створення ПЗ спочатку, використовуючи фреймворки з відкритими ліцензіями	Низька	Сильний
Зайнятість керівника науковою роботою, яка не зв'язана з моїми дослідженнями	Конкретизація питань до дипломного керівника	Середня	Сильний
Відсутність в інтернеті достатньої кількості літератури по темі.	Заздалегідь дізнатися назви наукових статей з даної теми. Знайти необхідну літературу.	Середня	Середній
Незнання архітектури	Виділення більшої кількості часу	Висока	Слабкий

3.3 ВИСНОВКИ

У даному розділі було сплановано процес підготовки дипломного проекту. Також була розрахована вірогідність закінчення роботи в запланований термін дорівнює 51%, що не є дуже хорошим результатом, але таким, який може застосовуватись. Але, якщо внести не дуже великі зміни в план, вірогідність збільшиться суттєво. Також було описано таблицю ризиків та засоби попередження цих ризиків, відповідно вони і були застосовані.

4. КЛЮЧОВІ ОСОБЛИВОСТІ ВИКОРИСТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

4.1. ВИКОРИСТАННЯ API GATEWAY

При застосуванні MSA для побудови додатку, необхідно вирішити, як його користувачі будуть взаємодіяти з мікросервісами. В моноліті є тільки один набір (зазвичай тиражований, з використанням балансувальника навантаження) кінцевих точок. В MSA кожен з мікросервіс надає набір своїх кінцевих точок. Тому потрібно розглянути, як це впливає на взаємодію користувач-додаток і що пропонує підхід, який використовує API Gateway.

API Gateway є сервером, який є єдиною точкою входу в систему. Він подібний зразку фасаду із об'єктно-орієнтованого підходу. API Gateway інкапсулює внутрішню архітектуру системи і надає API, який адаптований для кожного клієнта. Також він може виконувати такі функції, як аутентифікація, моніторинг, вирівнювання навантаження, кешування запитів, їх формування і управління ними, а також статична обробка відповіді.

На наступному рисунку (4.1) показано API Gateway в архітектурі додатку:

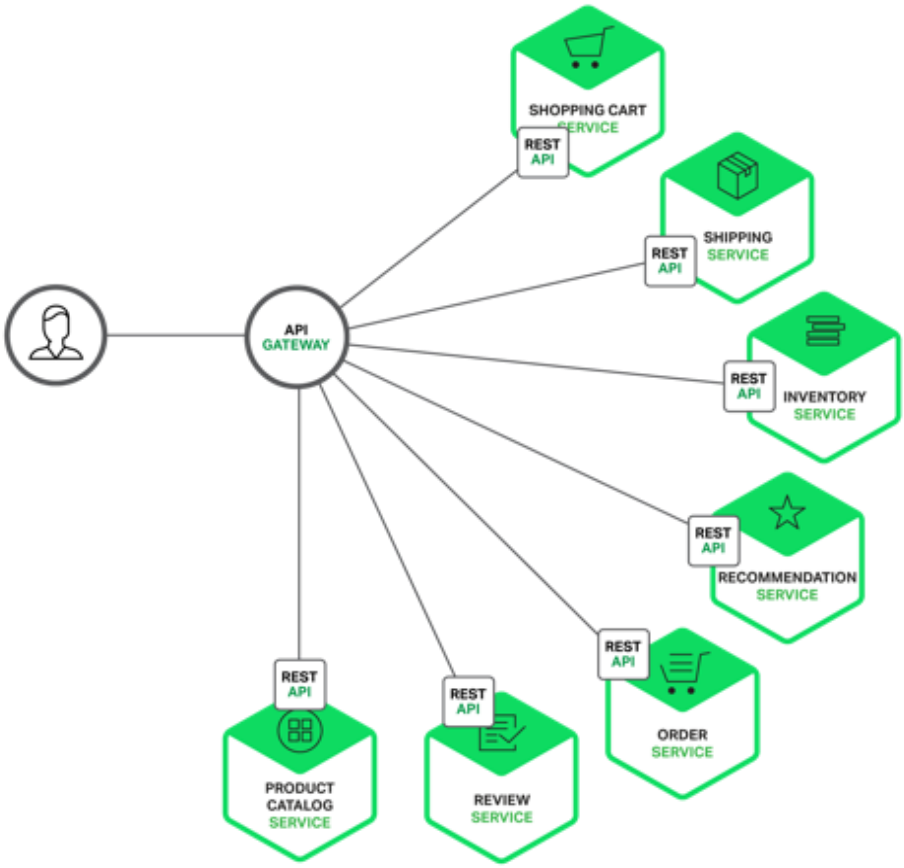

Рисунок 4.1 – API Gateway в архітектурі мікросервісного додатку

API Gateway відповідає за маршрутизацію запиту, його компонування і перетворення протоколів. Всі запити від користувачів спочатку проходять через API Gateway. Потім він направляє запити до відповідного мікросервісу. API Gateway частіше всього обробляє запит шляхом виклику декількох мікросервісів і агрегуванням результатів. Він може працювати між такими веб-протоколами, як HTTP і WebSocket, і протоколами, які не відносяться до веб, і які використовуються всередині кластеру мікросервісів. Gateway API може також надаватися кожному клієнту за допомогою API користувача. Це, як правило, надає один цілісний API для мобільних клієнтів. Розглянемо, наприклад, сценарій деталей певного продукту. API Gateway може надати кінцеву точку (/productdetails?productid=*xxx*), яку використовує мобільний клієнт, щоб отримати всі відомості про продукт за допомогою одного запиту.

Gateway API обробляє запит, викликаючи різні сервіси - інформація про продукт, рекомендації, відгуки і т.д. - і об'єднує результати. Відмінним прикладом API Gateway є Netflix API Gateway. Він обробляє мільярди запитів в день[6].

Як і слід було очікувати, використання API Gateway має як переваги, так і недоліки. Основна перевага використання його в тому, що він інкапсулює внутрішню структуру програми. Замість того, щоб посилатися на конкретні послуги, клієнти просто звертаються до даного інтерфейсу. Gateway API підтримує клієнта кожного виду певним потрібним API. Це зменшує кількість повторних повідомлень між клієнтом і додатком. Це також спрощує код клієнта.

Gateway API також має деякі недоліки. З’являється ще одна велика і доступна частина, яка має бути розроблена, розгорнута та якою потрібно керувати. Існує також ризик того, що API Gateway стає вузьким місцем для подальшого розвитку. Розробники повинні постійно оновлювати Gateway API, щоб додавати кінцеві точки кожного мікросервісу. Важливо, щоб процес оновлення API Gateway був якомога більш поверхневим. В іншому випадку, розробники будуть змушені чекати, щоб оновити його. Незважаючи на ці недоліки, для більшості реальних додатків використання API Gateway має досить великий сенс.

4.2. МІЖПРОЦЕСНІ КОМУНІКАЦІЇ

У монолітному додатку, його компоненти посилаються один на одного за допомогою методів на рівні мови програмування або викликів функцій. На противагу цьому, додаток з використанням мікросервісів є розподіленою системою, що працює на декількох машинах. Кожен екземпляр сервісу, як правило, окремий процес.

Отже, як показує наступний рисунок, сервіси повинні взаємодіяти з використанням механізму взаємодії між процесами (IPC) (рис.4.2).

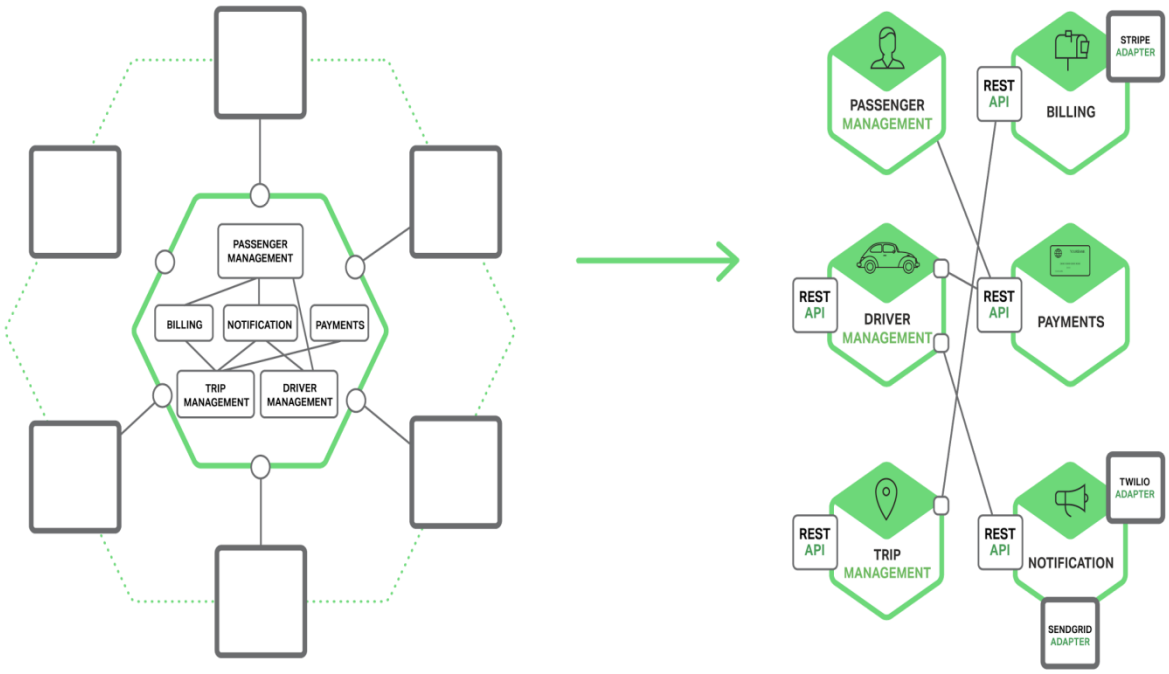

Рисунок 4.2 – Механізм взаємодії мікросервісів

При виборі механізму IPC для сервісу, потрібно спочатку подумати про те, як взаємодіють сервіси. Є багато стилів клієнт-сервісної взаємодії.

Вони можуть бути класифіковані за двома характеристиками. Перша - чи це взаємодія один-до-одного чи один-до-багатьох:

- Один-до-одного - кожен запит клієнта обробляється рівно одним екземпляром сервісу.
- Один-ко-багатьох - кожен запит обробляється декількома екземплярами сервісу.

Друга характеристика - це чи взаємодія є синхронною чи асинхронною:

Синхронна - клієнт очікує, своєчасної відповіді від сервісу і навіть може бути заблокованим, поки він чекає.

Асинхронна - клієнт не блокується під час очікування відповіді, і відповіді, якщо такі є, не обов'язково відправляються негайно.

У наступній таблиці наведені різні типи взаємодії (табл. 4.1):

Таблиця 4.1 – Типи взаємодії мікросервісів

	Один-до-одного	Один-до-багатьох
Синхронна	Запит/відповідь	—
Асинхронна	Сповіщення	Публікація/підписка
	Запит/асинхронна відповідь	Публікація/асинхронні відповіді

Існують наступні види взаємодій типу “один-до-одного”:

- Запит / відповідь - клієнт робить запит до сервісу і чекає відповіді. Клієнт очікує прибуття відповіді своєчасно. Потік, який робить запит може бути заблокованим під час очікування.
- Повідомлення (запит в одну сторону) - клієнт посилає запит до сервісу, але жодної відповіді не очікується.
- Запит/асинхронна відповідь - клієнт надсилає запит до сервісу, який відповідає асинхронно. Клієнт не блокується під час очікування і розроблений з врахуванням того, що відповідь не може прибути певний час.

Існують наступні види взаємодій типу “один-до-багатьох”:

- Публікація/підписка - клієнт публікує повідомлення, яке використовується зацікавленими сервісами або не використовується взагалі.
- Публікація/асинхронні відповіді - клієнт публікує повідомлення про запит, а потім очікує певний час відповідей від зацікавлених сервісів[7].

Кожен сервіс, як правило, комбінує ці стилі взаємодії. Для деяких, достатньо одного механізм IPC. Інші ж, використовують поєднання механізмів IPC. На

наступній діаграмі показано, як сервіси в додатку виклику таксі (який був взятий за приклад раніше) можуть взаємодіяти, коли користувач запитує поїздку (рис.4.3).

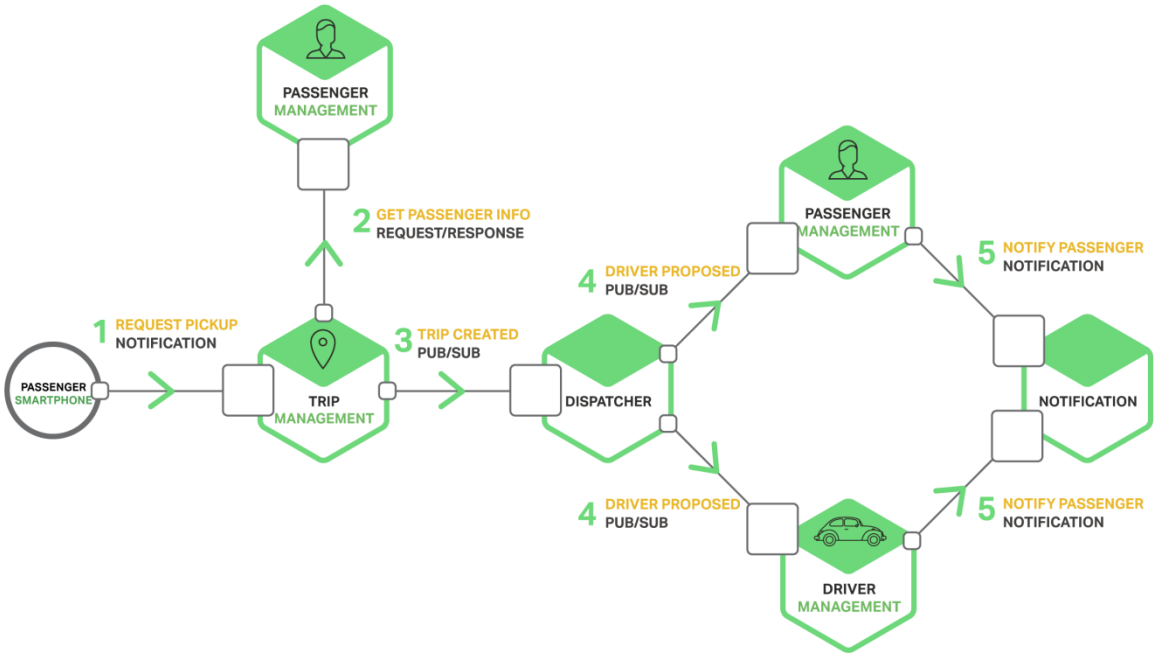

Рисунок 4.3 – Приклад взаємодії сервісів при виклику таксі

Сервіси використовують комбінацію сповіщень, запитів/відповідей і публікацій/підписок. Наприклад, смартфон пасажира відправляє повідомлення в сервіс Trip Managment. Той в свою чергу перевіряє, що обліковий запис пасажира активний, використовуючи запит/відповідь для виклику Passenger Service. Сервіс Trip Managment потім створює поїздку і використовує публікації/підписки для повідомлення інших сервісів, включаючи Dispatcher, який знаходить доступного водія.

Існує багато різних технологій IPC. Сервіси можуть використовувати синхронні запит/відповідь на основі механізмів зв'язку, таких як HTTP (REST). В якості альтернативи, вони можуть використовувати асинхронний підхід, на основі повідомлень, такі як AMQP або STOMP. Є також безліч різних форматів повідомлень. Сервіси можуть використовувати навть читабельні для людини,

текстові формати, такі як JSON або XML. В якості альтернативи, вони можуть використовувати бінарний формат (який більш ефективний), такі як Avro або Protocol Buffers[7].

4.3. SERVICE DISCOVERY

Для прикладу візьмемо написання коду, який викликає сервіс, що має REST API. Для того, щоб зробити запит, код повинен знати місце розташування екземпляру сервісу в мережі (IP-адреса і порт). Зазвичай, додаток працює на фізичному обладнанні, мережеве розташування екземпляру сервісу відносно статичне. Наприклад, код може читати мережеві адреси з файлу конфігурації, який періодично оновлюється. В сучасних, хмарно-орієнтованих мікросервісних додатках це, напевно, найбільш складна проблема, яка потребує вирішення, показана на рисунку 4.4.

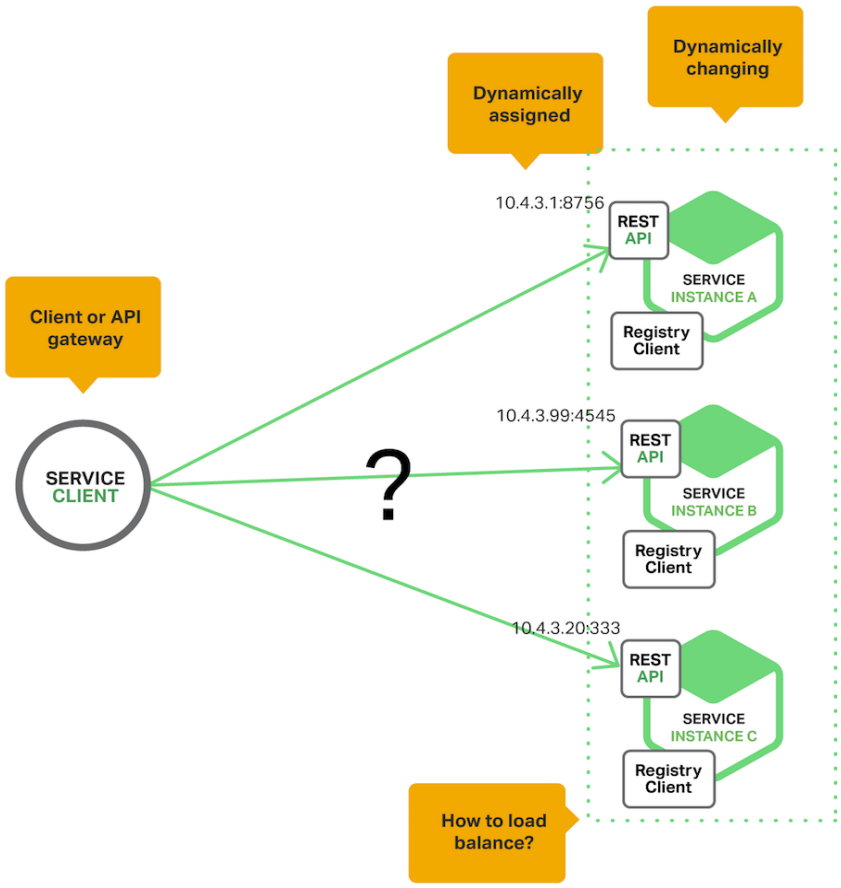

Рисунок 4.4 – Проблема виявлення сервісів

Екземплярам сервісів динамічно призначили місця розташування в мережі. Крім того, безліч екземплярів сервісів динамічно змінюються через автомасштабування, невдачі та модернізацію. Отже, клієнтський код потребує більш складного механізму виявлення сервісів.

Є дві основні моделі виявлення служби: на стороні клієнта і на стороні сервера. Спочатку розглянемо виявлення на стороні клієнта.

При використанні такого підходу, клієнт несе відповідальність за визначення знаходження в мережі, доступних екземплярів сервісів та запитів балансування навантаження через них. Клієнт запитує сервіс реєстру, який представляє собою базу даних, що містить всі доступні екземпляри сервісів. Потім клієнт використовує алгоритм вирівнювання навантаження, щоб вибрати один з доступних екземплярів сервісів, і робить запит. На наступному рисунку (рис.4.5) зображено даний підхід.

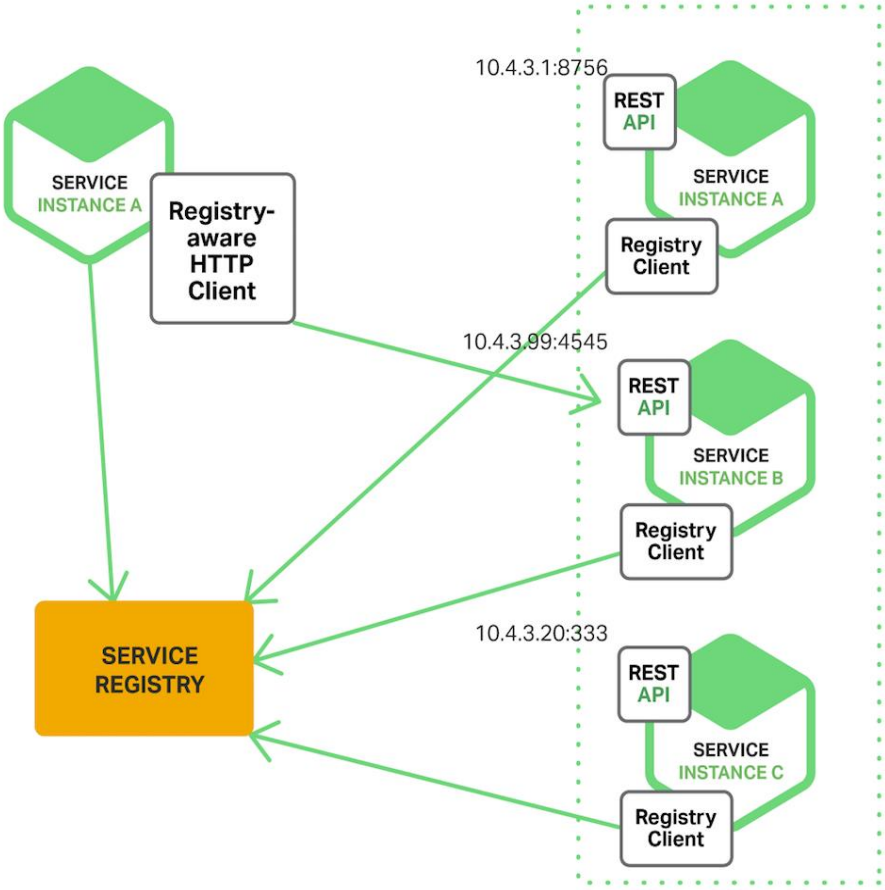

Рисунок 4.5 – Виявлення на стороні клієнта

Мережеве розташування екземпляру сервісу реєструється при його запуску. Воно видаляється з реєстру сервісів, коли екземпляр завершує свою роботу. Реєстр екземплярів сервісів, як правило, періодично оновлюється з використанням механізму моніторингу[8].

Даний підхід має ряд своїх переваг і недоліків. Ця модель відносно проста і, за винятком реєстру сервісів, немає ніяких інших рухомих частин. Крім того, оскільки клієнт знає про доступні екземпляри сервісів, він може зробити інтелектуальні, специфічні для даного додатка, рішення балансування навантаження, такі як використання послідовного хешування. Одним із суттєвих недоліків цієї моделі є те, що клієнт пов'язаний з реєстром сервісів. Необхідно реалізувати логіку виявлення сервісів на стороні клієнта для кожної мови програмування і фреймворків, які використовуються в сервісах.

Далі розглянемо виявлення на стороні сервера. На наступному рисунку (рис.4.6) зображено структуру даного підходу.

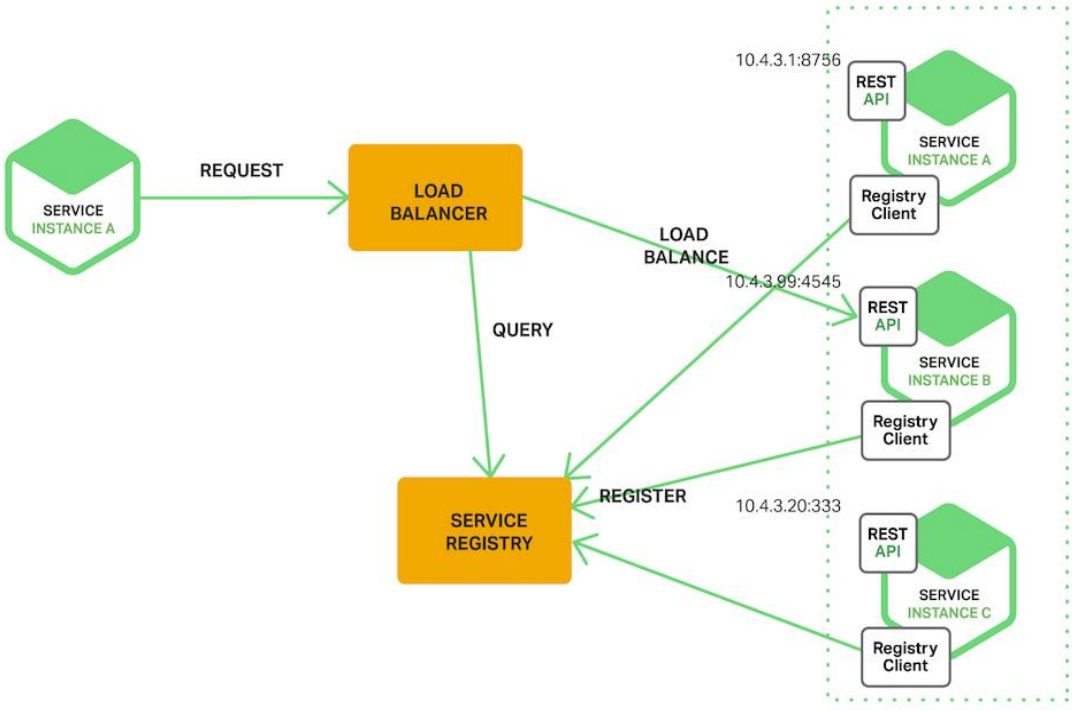

Рисунок 4.6 – Виявлення на стороні сервера

Клієнт робить запит до сервісу через вирівнювач навантаження. Вирівнювач навантаження запитує реєстр сервісів і спрямовує кожен запит на доступний екземпляр сервісу. Як і в разі виявлення на клієнтській стороні, екземпляри служб реєструються і видаляються у сервісі реєстру сервісів.

Такий підхід також має ряд своїх переваг і недоліків. Однією великою перевагою цієї моделі є те, що частини виявлення абстрагуються від клієнта. Клієнти просто виконують запити до балансувальника навантаження. Це усуває необхідність відпрацювання логіки виявлення для кожної мови програмування і фреймворка, які використовуються у сервісах. На жаль, ця модель також має деякі недоліки. Якщо вирівнювач навантаження не передбачує середовища розгортання, з'являється ще один компонент,який потрібно налаштувати і ним керувати.

Як бачимо, реєстр сервісів є ключовою частиною виявлення сервісу. Він являє собою БД, що містить мережеві розташування екземплярів сервісу. Реєстр сервісів повинен бути постійно доступним і актуальним. Клієнти можуть кешувати мережеві розташування, отримані з реєстру сервісів. Проте, ця інформація в результаті стає неактуальною, і клієнти стають не в змозі виявити екземпляри сервісів. Отже, реєстр сервісів складається з кластера серверів, які використовують протокол реплікації для забезпечення узгодженості. Як уже згадувалося раніше, Netflix Eureka - хороший приклад реєстру сервісів. Він забезпечує REST API для реєстрації та виконання запитів до екземплярів сервісів. Екземпляр сервісу реєструє своє розташування в мережі за допомогою запиту POST. Кожні 30 секунд він повинен поновлювати свою реєстрацію за допомогою запиту PUT. Реєстрація видаляється або через запит DELETE або через вихід екземпляру за рамкиі часу. Як і слід було очікувати, клієнт може отримати зареєстрований екземпляр служб за допомогою запиту GET.

У мікросервісному додатку, набір запущених екземплярів сервісів динамічно змінюється. Вони мають динамічно призначені мережеві розташування. Отже, для того, щоб клієнт міг зробити запит до сервісу, він

повинен використовувати механізм виявлення сервісів. Ключовою частиною виявлення сервісу є реєстр сервісів. Реєстр сервісів являє собою базу даних доступних екземплярів сервісів. Реєстр сервісів надає API керування і API запитів. Примірники служб реєструються і видаляються з реєстру сервісів за допомогою API керування. API запитів використовується системними компонентами, щоб виявити доступні екземпляри сервісів. Існують дві основні моделі обслуговування виявлення: виявлення на стороні клієнта і виявлення на стороні сервера. У системах, що використовують виявлення сервісів на стороні клієнта, клієнтські запити реєстру сервісу обирають доступний екземпляр і роблять запит. У системах, які використовують виявлення на стороні сарвера, клієнти роблять запити через маршрутизатор, який запитує реєстр сервісів і направляє запит до доступного екземпляру[8]. Є два основних способи, за допомогою яких екземпляри сервісів реєструються і видаляються з реєстру сервісів. Одним з варіантів, що є для екземплярів сервісів, полягає в реєстрації самого себе в реєстрі сервісу - самостійної реєстрації шаблону. Інший варіант підходить для будь-якої іншої компоненти системи для обробки реєстрації та скасування реєстрації від імені служби, тобто, реєстрації шаблону третьою стороною. У деяких середовищах розгортання необхідно створити свою власну інфраструктуру обслуговування процедури виявлення з використанням реєстру послуг, таких як Netflix Eureka. В інших середовищах розгортання, виявлення сервісів вбудовано. Наприклад, Kubernetes і Marathon з реєстрацією екземпляру сервісу та видалення його з реєстру. Вони також запускають проксі-сервер на кожному вузлі кластера, який грає роль виявлення маршрутизатора на стороні сервера.

4.4. УПРАВЛІННЯ ДАНИМИ

Монолітний додаток зазвичай має одну реляційну базу даних. Основною перевагою використання реляційних баз даних є те, що додаток може використовувати ACID транзакції, які забезпечують деякі важливі принципи:

- Atomicity - Атомарність гарантує, що жодна транзакція не буде виконана частково. Будуть або виконані всі операції, що беруть участь у транзакції, або не виконано жодної. Якщо протягом роботи однієї з операцій виникне помилка і операцію буде відхилено, то будуть відхилені також усі інші зміни, здійснені в межах транзакції.
- Consistency - Узгодженість. Відповідно до вимоги узгодженості, система повинна перебувати в узгодженому, несуперечливому стані до початку дії транзакції і по її завершенню. При цьому вона може перебувати в неузгодженому стані протягом виконання транзакції, проте ця неузгодженість завдяки іншим властивостям — атомарності та ізольованості — не буде видимою за межами транзакції.
- Isolation - Ізольованість означає, що жодні проміжні зміни не будуть видимі за межами транзакції аж до її завершення. Питання ізоляції стає актуальним при одночасній роботі багатьох транзакцій з одними й тими самими даними. Згідно з цією вимогою, якщо дві транзакції намагатимуться змінити одні й ті самі дані, то одну з них буде відхилено або призупинено до завершення другої.
- Durability - Довговічність гарантує, що незалежно від інших проблем після відновлення працездатності системи результати завершених транзакцій будуть збережені. Іншими словами, якщо користувач отримав повідомлення про успішне завершення транзакції, то він може бути впевнений, що дані будуть збережені та відновлені у випадку збоїв.

Ще однією великою перевагою використання реляційних баз даних є те, що вона надає SQL, який є повноцінною, декларативною, і стандартизований

мовою запитів. Можна легко написати запит, який поєднує в собі дані з декількох таблиць. Планувальник запитів РСУБД потім визначає найбільш оптимальний спосіб для його виконання. Так як всі дані додатку в одній БД, то їх легко отримати[9].

На жаль, доступ до даних стає набагато складнішою задачею, коли ми переходимо до MSA. Це відбувається тому, що дані, які належать кожному мікросервісу є приватним до нього і можуть бути доступні тільки через його API. Інкапсуляція даних гарантує, що мікросервіси слабо пов'язані і можуть розвиватися незалежно один від одного.

Для багатьох додатків, найкращим рішенням є використовувати архітектуру, керовану подіями. У цій архітектурі мікросервіс публікує подію, коли відбувається щось помітне, наприклад, коли він оновлює бізнес-об'єкт. Інші мікросервіси підписані на ці події. Коли мікросервіс отримує подію він може оновити свої власні бізнес-структури, які могли б привести до публікації більшої кількості подій.

Можна використовувати події для виконання бізнес-операцій, які охоплюють кілька сервісів. Транзакція складається з ряду етапів. Кожен крок складається з оновлення мікросервісом бізнес-об'єкта та публікації події, яка запускає наступний крок. На рисунках в кожному кроці показано, як можна використовувати даний підхід для перевірки доступного кредиту при створенні замовлення. У мікросервісів обмін подіями відбувається через Message Broker.

1. Сервіс замовлення створює замовлення зі статусом Новий і видає наказ створити подію (рис.4.7).

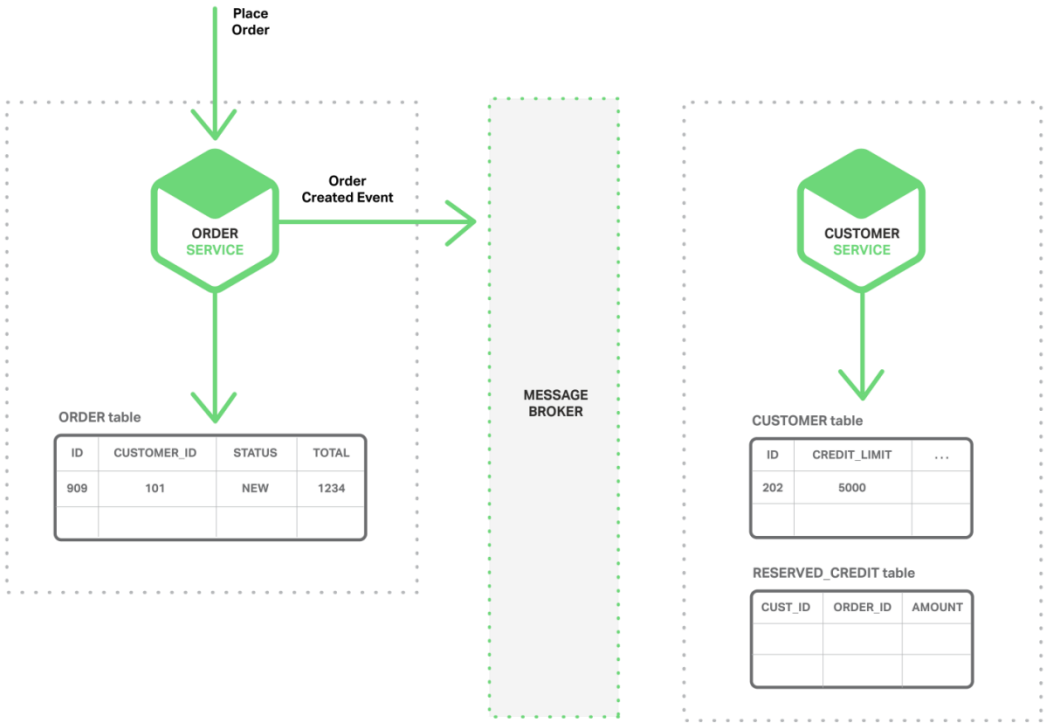

Рисунок 4.7 – Створення події мікросервісом

2. Сервіс клієнтів приймає подію створення замовлення, резервує кредит на замовлення, і видає Credit Reserved подія (рис.4.8).

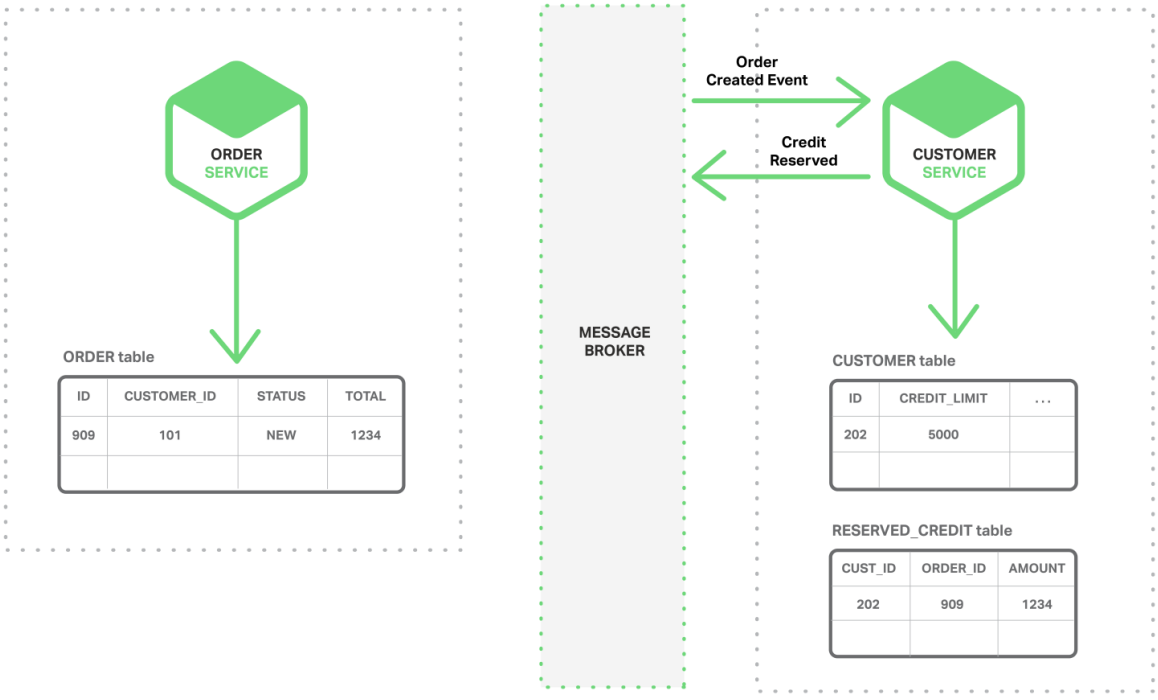

Рисунок 4.8 – Прийом події та створення нової

3. Служба замовлення приймає подію Credit Reserved, і змінює статус замовлення на OPEN (рис.4.9).

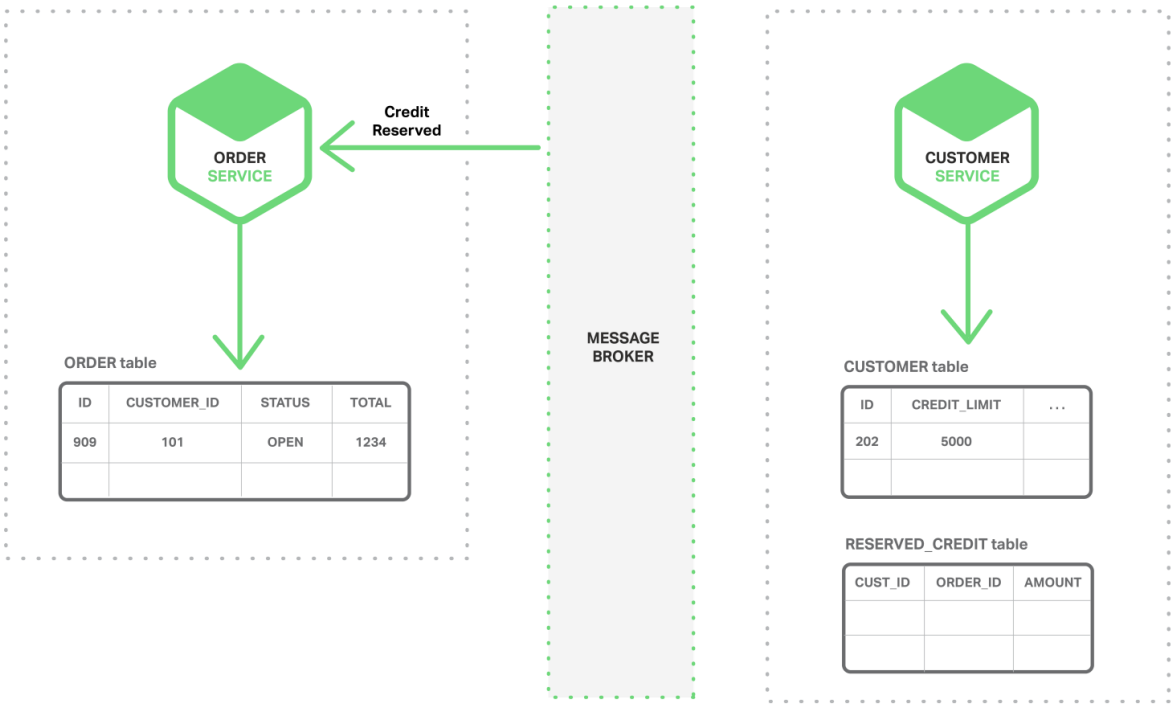

Рисунок 4.9 – Прийом події та зміна статусу замовлення

Більш складний сценарій може включати в себе додаткові етапи.

Архітектура подій має ряд своїх переваг і недоліків. Вона дозволяє реалізувати транзакції, які охоплюють кілька сервісів і в кінцевому підсумку є узгодженими. Один із недоліків полягає в тому, що дана модель є більш складною, щоб її запрограмувати ніж при використанні ACID транзакцій. Часто потрібно здійснити компенсуючі транзакції для відновлення після збоїв на рівні додатку. Крім того, додатки будуть мати справу з суперечливими даними. Інший недолік полягає в тому, що сервіси повинні виявляти і ігнорувати події, які аовторюються[9].

В MSA, кожен мікросервіс має свою власну БД. Різні мікросервіси можуть використовувати різні SQL або NoSQL БД. У той час як така архітектура має значні переваги, вона створює деякі проблеми в галузі управління розподіленими даними. Перше завдання полягає в тому, як

реалізувати бізнес-операції, які підтримують узгодженість між декількома сервісами. Друга проблема полягає в тому, як реалізувати запити, які витягають дані з декількох сервісів.

Для багатьох додатків підходить рішення використовувати архітектуру, керовану подіями. Основна проблема її впровадження - це як атомарно оновлювати стан і як публікувати події.

4.5. ВИСНОВКИ

У розділі було описано основні структурні елементи та підходи до реалізації мікросервісної архітектури, без яких неможливо її побудувати правильно. Дуже важливо дотримуватись описаних принципів, щоб додаток видав найкращу продуктивність, а також працював повноцінно і правильно

5. ПОРІВНЯННЯ ВІДОМИХ JAVA-ФРЕЙМВОРКІВ ДЛЯ СТВОРЕННЯ МІКРОСЕРВІСІВ

Мікросервіси, ймовірно, один з найбільш часто використовуваних термінів сьогодні, коли мова йде про архітектуру ПЗ. Не зважаючи на те, що основні поняття є не зовсім новими, це напевно найбільш відомий і вживаний принцип побудови ПЗ, який дуже популярний протягом останніх двох років.

Підводячи підсумок по попередніх розділах, можна зробити висновки, що велика монолітна архітектура має тенденцію ставати не дуже підтримуваною і розширюваною, коли її функціонал з часом розростається та нагромаджується. Крім того, вона майже не масштабується (масштабування шляхом множення великих додатків), і, по-суті, неможливо замінити старі частини.

Однією з найбільших переваг MSA є вирішення цих проблем: замість створення цілого додатку як одного блоку, можна побудувати його як набір послуг, які будуть здійснювати зв'язок через певну систему обміну повідомленнями (частіше всього REST через HTTP-протокол). Завдяки цьому, з’являється можливість замінити тільки одну потрібну частину; можна масштабувати тільки необхідний елемент, і так далі.

Але з іншої сторони: якщо будувати розподілену систему, то із цього випливають недоліки розподіленої системи. Побудова розподіленої системи в якості MSA не є тривіальною задачею, але існує багато рішень, які можуть спростити цю задачу. У даному розділі будуть представлені деякі із таких рішень.

5.1. ВИМОГИ

Це не буде порівняння фреймворків, яке засноване на створенні додатків типу “Hello World”. Більшість з них дозволяють створити та розгорнути на

сервері такі додатки, приклавши найменші зусилля. Це не є відображенням реального життя та вимог, які ставляться перед такими рішеннями.

При створенні реальних додатків не використовується звичайний HTTP. Обов'язковою вимогою є використання REST/HTTPS, а також повинен існувати HTTPS-клієнт. Після успішного створення сервісу повинен бути створений заголовочний файл з його місцерозташуванням та додані посилання на JSON або XML представлення. Також повинні бути включені метрики для контролю додатку. Важлива також присутність якогось виду протоколу безпеки для захисту API.

Отже, список того, що буде оцінюватись:

- Буде побудовано примітивний RESTful API. UI не розглядається.
- Використання тільки HTTPS для доступу до API.
- API матиме два набори ресурсів (тобто два мікросервіси). Для простоти, вони будуть розміщені на одному сервері, але будуть обмінюватися даними по протоколу HTTPS.
- Ресурс матиме JSON представлення.
- Кожне представлення матиме, як мінімум, власне посилання на ресурс.
- Після створення ресурсу, клієнт отримує Заголовок розташування(Location header, посилання ресурсу).
- Кожен API повинен мати об'єкт моніторингу/метрик
- Кожен сервіс може бути упакований в вигляді автономного jar пакету, який можна запустити з використанням Java -jar.

5.2. КАНДИДАТИ

Список фреймворків, які будуть порівнюватись:

- Dropwizard
- Vertx

- Spring Boot
- Restlet
- Spark + Unirest(REST-клієнт, сам Spark не надає REST-клієнт)

Потрібно звернути увагу, що Restlet і Spark не претендують на роль мікросервісів, але Restlet, як відомо, надає дуже хорошу основу для REST і Spark дуже легковісний. Саме тому не потрібно обходити дані рішення стороною, надаючи перевагу таким потужним фреймворкам, як, наприклад, Spring.

5.3. DROPWIZARD

Як заявляє веб-сайт: ”Dropwizard - це Java-фреймворк для дружніх до DevOps, високопродуктивних RESTful веб-сервісів”. Він включає в себе такі успішні рішення, як Jetty, Jersey та Jackson. Також присутня дуже хороша документація, що не менш важливо - не потрібно довго шукати речі, що здаються складними, адже вони, насправді, доволі прості.

5.3.1 ГОЛОВНЕ

Основний метод досить простий, створюється екземпляр додатку після чого він запускається:

```
public static void main(String[] args) throws Exception {
new DropwizardApplication().run(args);
}
```

Клас DropwizardApplication містить все необхідне: перевірку ресурсів та реєстрації, Guice bootstrapping і конфігурацію Jackson’s ObjectMapper. Він надає екземпляр класу конфігурації (DropwizardServerConfiguration), який є POJO, отримує конфігурації з файлу YAML, переданого в якості параметра для нашого додатку.

Додаток запускається з параметрами: Java -jar app.jar server config.yml. Параметр server повинен бути вказаний Dropwizard, щоб запустити сервер. Також можна працювати з базою даних.

5.3.2 РЕСУРСИ

Всередині, Dropwizard використовує Jersey, так що ресурси є просто POJO анотовані (більшість) JAX-RS анотаціями:

```
@Path("/cars/{id}")
@Produces(MediaType.APPLICATION_JSON)
public class CarResource {
   @Context
   UriInfo uriInfo;
   @Inject
   private CarRepository carRepository;
   @GET
   public Response byId(@Auth User user, @PathParam("id") int carId) {
      Optional<Car> car = carRepository.byId(carId);
      return car.map(c -> {
         CarRepresentation carRepresentation = new CarRepresentation(c);

carRepresentation.addLink(Link.self(uriInfo.getAbsolutePathBuilder().build(c.getId()
).toString()));
         return Response.ok(carRepresentation).build();
      }).orElse(Response.status(Response.Status.NOT_FOUND).build());
   }
}
```

Вся основна обробка HTTP виконується фреймворком, непотрібен обов'язковий доступ до об'єктів запитів і відповідей. У цьому випадку

повертається відповідь, хоча міг бути повернений об’єкт; Однак, код відповіді не буде правильним (201), таким чином, щоб мати повний контроль над ним, такий варіант є найкращим рішенням. Крім того на відповідь встановлюється 404 (Status.NOT_FOUND).

Для серіалізації/десеріалізації Dropwizard використовує Jackson, об'єкт повертається в форматі JSON, тому не потрібно робити нічого зайвого. Обов’язково потрібно налаштувати ObjectMapper - відключити в ньому помилки на невідомі властивості.

5.3.3. HTTPS

HTTPS налаштовується у файлі конфігурації YAML; фреймворк ігнорує стандартні властивості Java. Весь процес налаштування описаний в документації і не являє собою нічого складного.

5.3.4.REST-КЛІЄНТ

Клієнт REST побудований у вигляді Singleton за допомогою Guice. Це не вказано у документації та досить важко заставити його працювати. З іншої сторони більше нічого особливого в клієнті, API є легким для сприйняттяі простим:

```
@Override
public List<Car> getAllCars(String auth) {
   WebTarget target = client.target("https://localhost:8443/app/cars");
   Invocation invocation = target.request(MediaType.APPLICATION_JSON)
         .header("Authorization", "Bearer " + auth)
         .build(HttpMethod.GET);
   Car[] cars = invocation.invoke(Car[].class);
   return asList(cars);
}
```

Client, на цей раз, використовує стандартні властивості.

5.3.5. БЕЗПЕКА

Аутентифікація вимагає двох речей: по-перше, реалізувати інтерфейс Authenticator. По-друге, потрібно зареєструвати аутентифікатор в Jersey:

```
environment.jersey().register(AuthFactory.binder(
        new
OAuthFactory<>(guiceBundle.getInjector().getInstance(FacebookTokenAuthenticato
r.class),
                getName() + "-Realm",
                User.class)));
```

5.3.6. МОНІТОРИНГ

Dropwizard має вбудовану систему моніторингу. Можна зареєструвати healthchecks, щоб переконатися, що стан додатку, і кожного ресурсу можна виміряти за допомогою анотацій. Крім того, можна додавати власні метрики, за допомогою реєстру метрик, отриманого з Environment.

5.3.7. ВИСНОВКИ

Не зважаючи на труднощі встановлення, Dropwizard є доволі хорошим фреймворком. Він забезпечує всі функції, необхідні для побудови програми на MSA. Проте, для створення невеликих сервісів він є занадто громіздким. Тому в такому випадку його краще не використовувати, оскільки можна легко отримати не те, що очікувалось.

5.4. VERTX

"Vertx це набір інструментів для створення реактивних додатків на JVM". Звичайно, за допомогою нього можна розробляти на Java але також підримується багато мов, які працюють на JVM (JavaScript, Scala, Ruby, Python, Clojure).

Він також забезпечує актороподібну систему, яка дозволяє розгортання незалежних, паралельних, і, можливо, написаних на різних мовах, сервісів, які обмінюються даними через шину подій.

5.4.1 ГОЛОВНЕ

Фреймворк абстрагує низькорівневу обробку HTTP, але потрібно створювати сервер вручну:

```
Vertx vertx = Vertx.create();
HttpServer server = vertx.createHttpServer(serverOptions);
```

Основний метод створює HTTP-клієнт, встановлює систему аутентифікації і пов'язує «ресурси» шляхами.

5.4.2 РЕСУРСИ

В VertX немає класу для обробки ресурсів. Порібно просто надати обробники шляхів:

```
CarResource carResource = new CarResource(carRepository);
router.get("/cars/:id").produces("application/json").handler(carResource::byId);
```

CarResource є простим POJO, має метод byId з параметром RoutingContext:

```
public void byId(RoutingContext routingContext) {
   HttpServerResponse response = routingContext.response();
   String idParam = routingContext.request().getParam("id");
   if (idParam == null) {
      response.setStatusCode(400).end();
   } else {
      Optional<Car> car = carRepository.byId(Integer.parseInt(idParam));
      if (car.isPresent()) {
         CarRepresentation carRepresentation = new CarRepresentation(car.get());
         carRepresentation.addLink(self(routingContext.request().absoluteURI()));
```

```
        response.putHeader("content-type", "application/json")
                .end(Json.encode(carRepresentation));
    } else {
        response.setStatusCode(404).end();
    }   }}
```

У нас є повний контроль над відповіддю. Ми знаємо точно, що ми робимо і що отимаємо. JSON кодування виконується за допомогою Jackson знову, тому доведеться відключити функцію "fail on unknown property". Також це не працюватиме без точного налаштування шляхів

```
router.route("/cars*").handler(BodyHandler.create());
```

За замовчуванням, VertX ігнорує тіло, тому потрібно явно вказати те, що потрібно його прочитати. В іншому випадку, не буте отримано зміст тіла. Також потрібно звернути увагу, що не існує залежності ін'єкції, все робиться вручну.

5.4.3. HTTPS

Визначення serverOptions в секції Main:

```
HttpServerOptions serverOptions = new HttpServerOptions()
        .setSsl(true)
        .setKeyStoreOptions(new JksOptions()
                .setPath(System.getProperty("javax.net.ssl.keyStorePath"))
                .setPassword(System.getProperty("javax.net.ssl.keyStorePassword")))
        .setPort(8090);
```

Цей об'єкт дозволяє встановити властивості порту і SSL. VertX автоматично не отримує стандартні властивості, тому це потрібно зробити вручну.

5.4.4.REST-КЛІЄНТ

Побудова клієнта така ж, як створення сервера:

```
HttpClientOptions clientOptions = new HttpClientOptions()
        .setSsl(true)
        .setTrustStoreOptions(new JksOptions()
        .setPath(System.getProperty("javax.net.ssl.trustStore"))
        .setPassword(System.getProperty("javax.net.ssl.trustStorePassword")));
HttpClient httpClient = vertx.createHttpClient(clientOptions);
```

Дуже легко у використанні:

```
httpClient.getAbs("https://localhost:8090/cars")
        .putHeader("Accept", "application/json")
        .putHeader("Authorization", "Bearer " +
routingContext.user().principal().getString("token"))
        .handler(response ->
            response.bodyHandler(buffer -> {
              if (response.statusCode() == 200) {
                List<Car> cars = new
ArrayList<>(asList(Json.decodeValue(buffer.toString(), Car[].class)));
                routingContext.response()
                    .putHeader("content-type", "test/plain")
                    .setChunked(true)

.write(cars.stream().map(Car::getName).collect(toList()).toString())
                    .write(", and then Hello World from Vert.x-Web!")
                    .end();
              } else {
                routingContext.response()
                    .setStatusCode(response.statusCode())
                    .putHeader("content-type", "test/plain")
                    .setChunked(true)
                    .write("Oops, something went wrong: " +
```

```
response.statusMessage())
                        .end();
                }
            }))
        .end();
```

Як бачимо, ми маємо повний контроль (подібно із сирою відповіддю сервера).

5.4.5. БЕЗПЕКА

Існує модуль OAuth, але він не відповідає вимогам. Тому це було зроблено вручну. Він складається з двох класів, AuthHandler і AuthProvider. Після цього порібно встановити перший як обробник для всіх шляхів:

```
AuthHandler auth = new BearerAuthHandler(new
FacebookOauthTokenVerifier(httpClient));
router.route("/*").handler(auth);
```

5.4.6. МОНІТОРИНГ

VertX пропонує систему метрик, з безліччю різних метрик. Звичайно, присутня можливість зареєструвати свої власні метрики за допомогою MetricsRegistry.

5.4.7. ВИСНОВКИ

VertX є дійсно завершеною системою. Він побудований на вершині Netty, і пропонує дуже продуктивну систему. Фреймворк написано на Java 8, тому можна написати дуже елегантний код. Також VertX є асинхронним, з чого також випливає велика кількість переваг.

5.5. SPRING BOOT

Напевно всі, хто використовував Java, чули про такий фреймворк, як Spring. Він також надає можливості для реалізації MSA. Тому його потрібно обов’язково розглянути.

5.5.1 ГОЛОВНЕ

Дуже важко ще більш спростити головний клас:

```
@SpringBootApplication
public class Main {
   public static void main(String[] args) {
      SpringApplication.run(Main.class);
   }
}
```

@SpringBootApplication Запускає компонент сканування і налаштування програми Spring.

Потрібно звернути увагу, що можна перемкнути основний веб-сервер, а також вибрати між Tomcat, Jetty і Undertow. У прикладі використано Jetty. Це робиться шляхом вибору модуля, який потрібен в системі управління залежностями (класу, в даному випадку).

5.5.2 РЕСУРСИ

Класи ресурсів анотуються як @RestController. В іншому випадку, так само, як JAX-RS: @RequestMapping для @Path, ResponseEntity для Response і так далі.

```
@RestController
@RequestMapping(value = "/cars", produces = "application/json")
public class CarsController {
```

```
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public ResponseEntity<CarRepresentation> byId(@PathVariable("id") String
carId) {
        Optional<Car> car = carRepository.byId(Integer.parseInt(carId));
        return car.map(c -> {
            CarRepresentation rep = toCarRepresentation(c);
            return new ResponseEntity<>(rep, HttpStatus.OK);
        })
                .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    private CarRepresentation toCarRepresentation(Car c) {
        String carId = String.valueOf(c.getId());
        CarRepresentation rep = new CarRepresentation(c);
        rep.add(linkTo(methodOn(CarsController.class).byId(carId)).withSelfRel());
        return rep;
    }
```

Зв'язування здійснюється за допомогою модуля spring-hateoas замість ручного рішення. Spring використовує Jackson для серіалізації JSON, але не потрібно налаштовувати параметр "ignore unknown property": він відключений за замовчуванням.

5.5.3. HTTPS

HTTPS налаштовується в application.yml (або configuration.properties) файлі. Це більше, ніж в Dropwizard, наприклад. Просто потрібно слідувати документації.

5.5.4.REST-КЛІЄНТ

Клієнт RestTemplate. Він дуже простий у використанні:

```
MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
headers.add("Accept", "application/json");
headers.add("Authorization", authToken);
HttpEntity<MultiValueMap<String, String>> requestEntity = new HttpEntity<>(null,
headers);
RestTemplate rest = new RestTemplate();
ResponseEntity<Car[]> responseEntity = rest.exchange("https://localhost:8443/cars",
HttpMethod.GET, requestEntity, Car[].class);
List<Car> cars = asList(responseEntity.getBody());
```

Spring Boot використовує стандартні властивості ssl.

5.5.5. БЕЗПЕКА

Досить просто налаштувати: просто потрібно анотувати клас ресурсів і налаштувати кінцеву точки, де можна перевірити маркер:

```
@EnableOAuth2Resource
@RestController
public class SampleController {
        (...)
}
spring:
  oauth2:
    resource:
      userInfoUri: https://graph.facebook.com/v2.4/me
      preferTokenInfo: false
```

@EnableOAuth2Resource не поставляється в Spring Boot, але включений в Spring Cloud Security.

5.5.6. МОНІТОРИНГ

Просто потрібно перенести модуль для активації метрики: Spring Boot Actuator. За замовчуванням, існують метрики, зареєстровані у фреймворку, але можна легко додати свої власні. Всередині Spring Boot використовує метрики Dropwizard.

5.5.7. ВИСНОВКИ

Фреймворк дійсно пришвидшує роботу. Головне просто слідувати документації. За допомогою Spring Boot дуже легко побудувати додаток на MSA.

5.6. RESTLET

“Restlet Framework є найбільш широко використовуваним рішенням з відкритим вихідним кодом для Java-розробників, які хочуть створювати і використовувати API”. Насправді ж Restlet - фреймворк, який надає широкі можливості використання REST. Про це рішення багато хто чув, але мало хто його використовував.

5.6.1 ГОЛОВНЕ

Головний клас досить простий:

```
public static void main(String[] args) throws Exception {
Injector injector = Guice.createInjector(new SelfInjectingServerResourceModule(),
        new RestletInfraModule(),
        new CarModule(),
        new HelloModule());
   RestComponent component = injector.getInstance(RestComponent.class);
   component.start();
}
```

RestComponent містить все необхідне, наприклад, конфігурації шляхів, налаштування HTTPS, і так далі. У Restlet, немає файлу конфігурації, все робиться за допомогою коду. Це хороший підхід, якому надають перевагу більшість розробників.

5.6.2 РЕСУРСИ

В Restlet використовується менше анотацій, ніж в інших фреймворках:

```
public class CarsResource extends ServerResource {

   @Inject
   private CarRepository carRepository;

   @Get("json")
   public List<CarRepresentation> all() {
      List<io.github.cdelmas.spike.common.domain.Car> cars = carRepository.all();
      getResponse().getHeaders().add("total-count", String.valueOf(cars.size()));
      return cars.stream().map(c -> {
         CarRepresentation carRepresentation = new CarRepresentation(c);
         carRepresentation.addLink(Link.self(new
Reference(getReference()).addSegment(String.valueOf(c.getId())).toString()));
         return carRepresentation;
      }).collect(toList());
   }

   @Post("json")
   public void createCar(io.github.cdelmas.spike.common.domain.Car car) {
      carRepository.save(car);
      setLocationRef(getReference().addSegment(String.valueOf(car.getId())));
      setStatus(Status.SUCCESS_CREATED);
```

```
    }
}
```

Використовуючи дане рішення потрібно розширити клас середовища. Це не дуже важко. Анотації прості у використанні. Потрібно звернути увагу, що треба встановити статус, локацію класу, а не об'єкту відповіді. Те ж саме для заголовків (GetResponse(). GetHeaders() і т. д.). Атрибути і параметри витягуються з об'єкта ресурсу теж. Це не дуже хороший підхід.

Найбільш неочевидною (і, як виявилось, не документованою), є конфігурація ObjectMapper. По-перше, потрібно написати спеціальний конвертер (повний клас), а по-друге потрібно замінити існуючий конвертер після повної ітерації над перетворювачами.

5.6.3. HTTPS

Знову ж таки, дуже легко встановити:

```
getClients().add(Protocol.HTTPS);
Server secureServer = getServers().add(Protocol.HTTPS, 8043);
Series<Parameter> parameters = secureServer.getContext().getParameters();
parameters.add("sslContextFactory",
"org.restlet.engine.ssl.DefaultSslContextFactory");
parameters.add("keyStorePath", System.getProperty("javax.net.ssl.keyStorePath"));
```

Restlet використовує майже стандартні властивості. Але keystorePath, на подив, не використовує стандартні властивості. З іншої сторони, це досить добре задокументовано і легко зробити.

5.6.4.REST-КЛІЄНТ

Існує клієнт, який надається Restlet. Насправді, він не простий у використанні. Після установки властивостей HTTPS, потрібно використовувати інтерфейс з анотованим методом (ресурсо-подібний інтерфейс) оберненим

клієнтом. Все таки це клієнтський ресурс, а не простий HTTP-клієнт. Це доволі складний підхід для необізнаного користувача.

5.6.5. БЕЗПЕКА

Для захисту API, потрібно додати захист на шлях. Захист запускає верифікатор, який є зазвичай один, і, потрібно сказати, що його не дуже просто реалізувати. Після цього все працює добре. В документації це не вказано.

5.6.6. МОНІТОРИНГ

Restlet не надає засіб моніторингу.Це лише основа для REST, і не конкурує із Spring Boot або Dropwizard.

Звичайно, можна додати метрики Dropwizard, але це дуже багато додаткової роботи.

5.6.7. ВИСНОВКИ

Restlet не є основою для MSA. Проте, він дуже підходить в якості основи REST, до тих пір, поки не прийдеться налаштувати його. Тоді виникає багато неочевидних неприємностей.

5.7. SPARK JAVA

Не потрібно плутати Spark із Apache Spark. Вони не мають нічого спільного. “Spark - це маленький фреймворк для створення веб-додатків з мінімальними затратами”. Насправді він маленький, але доволі-таки функціональний.

5.7.1 ГОЛОВНЕ

Головний метод в значній мірі нічим не відрізняється від інших фреймворків. Потрібно звернути увагу, що Spark не використовує Jackson в якості маппера. Але, в якості опції, використовує маппер, який передається в

якості параметра деяких викликів конфігурації шляху. Маппер - це просто метод, який приймає об’єкт в якості параметра і повертає рядок. Всі налаштування виконуються викликаючи статичні методи стандартних класів Spark.

5.7.2 РЕСУРСИ

Spark не має класу ресурсів, існують тільки шляхи. Шлях - це просто метод, який приймає запит і відповідь в якості параметрів, і може повертати нічого в якості результату.

```
public String createCar(Request request, Response response) {
   Car car;
   try {
      car = objectMapper.readValue(request.body(), Car.class);
      carRepository.save(car);
      response.header("Location", request.url() + "/" + car.getId());
      response.status(201);
   } catch (IOException e) {
      response.status(400);
   }
   return "";
}
```

5.7.3. HTTPS

В Spark, є статичний метод виклику установки HTTPS:

```
String keystoreFile = System.getProperty("javax.net.ssl.keyStorePath");
String keystorePassword = System.getProperty("javax.net.ssl.keyStorePassword");
String truststoreFile = System.getProperty("javax.net.ssl.trustStore");
String truststorePassword = System.getProperty("javax.net.ssl.trustStorePassword");
secure(keystoreFile, keystorePassword, truststoreFile, truststorePassword);
```

5.7.4.REST-КЛІЄНТ

Spark є серверним фреймворком і не надає клієнтську частину. Щоб вирішити це, необхідно використовувати сторонні рішення, наприклад, Unirest. Він дуже схожий із Spark, та все конфігурується викликами статичних методів класу Unirest.

Unirest не використовує Jackson в якості маппера, але надає інтерфейс (спрощену версію ObjectMapper Jackson, яка називається ObjectMapper), який можна реалізувати за допомогою Jackson. Все легко реалізується, слідуючи документації.

5.7.5. БЕЗПЕКА

Спарк дозволяє визначити фільтри, використовуючи before і after. У даному випадку, використовується before фільтр:

```
AccessTokenVerificationCommandFactory
accessTokenVerificationCommandFactory = new
AccessTokenVerificationCommandFactory();
AuthenticationFilter authenticationFilter = new
AuthenticationFilter(accessTokenVerificationCommandFactory);
before(authenticationFilter::filter);
```

Filter, як завжди в Spark, метод, який приймає запит і відповідь як параметри.

5.7.6. МОНІТОРИНГ

Як простий фреймворк, Spark не надає можливостей моніторингу. Проте, є проста бібліотека: spark-metrics. Це набір декораторів шляхів.

5.7.7. ВИСНОВКИ

За допомогою Spark можна дуже швидко реалізувати повноцінний сервер. Він легковісний, працює швидко і зручний у використанні. Але у ньому немає

всіх тих функцій, які присутні у більш важких фреймворках. Хоча, це легко виправляється сторонніми бібліотеками і рішеннями. Тому Spark можна рекомендувати для створення проектів будь-яких розмірів[10].

5.8. ВИСНОВКИ

В ході порівняння даних фреймворків для реалізації MSA було виявлено, що кожен з них має свої плюси і свої мінуси. Всі фреймворки показали себе з найкращої сторони при запуску (який тривав буквально кілька мілісекунд), але невідомо, як вони будуть вести при великих навантаженнях у громіздких корпоративних додатках. Найбільш рекомендованими для ознайомлення, вивчення та подальшого застосування виявились Spring Boot та Spark. Саме тому вони будуть застосовані у наступних розділах.

6. ПРИКЛАД ПРОСТОГО ДОДАТКУ НА ФРЕЙМФОРКУ SPRING CLOUD З ПОДАЛЬШИМ РОЗГОРТАННЯМ НА DOCKER

В цьому розділі будуть описані деякі з основних концепцій побудови MSA з використанням Spring Cloud і Docker.

6.1.SPRING CLOUD

Spring Cloud представляє собою набір інструментів від команії Pivotal, які надають необхідні засоби при побудові розподілених систем, що найбільш часто зустрічаються. Spring Cloud використовує основні блоки та принципи відомого фреймворку Spring Framework.

Серед рішень, що надаються Spring Cloud, є інструменти для виконання таких завдань:

- Configuration management
- Service discovery
- Circuit breakers
- Distributed sessions

6.2. SPRING BOOT

Spring Boot дозволяє лекго створювати повноцінні, виробничого класу Spring-додатки, про які можна сказати - "просто запусти". В Spring-платформу включено і сторонні бібліотеки, щоб була можливість запустити додаток з мінімальними зусиллями. Більшості Spring Boot додатків потрібна лише примітивна Spring-конфігурація[11].

Spring Boot надає наступні можливості:

- Створення повноцінних Spring додатків

- Вбудований Tomcat або Jetty (не потребує встановлення WAR файлів)
- Забезпечує 'початкові' POMs для спрощення Maven конфігурації
- Автоматична конфігурірація Spring коли це є можливим
- По замовчуванню надаються метрики, моніторинг станами і розширена конфігурація

6.3. SERVICE DISCOVERY ТА ІНТЕЛЕКТУАЛЬНА МАРШРУТИЗАЦІЯ

Кожен сервіс має свою мету і завдання в MSA. При реалізації даного підходу на Spring Cloud перші два мікросервіси, які потрібно створити це сервіс конфігурації і сервіс виявлення.

На рисунку 6.1 наведено приклад налаштування чотирьох мікросервісів, із зв’язками між ними, що вказують на залежність.

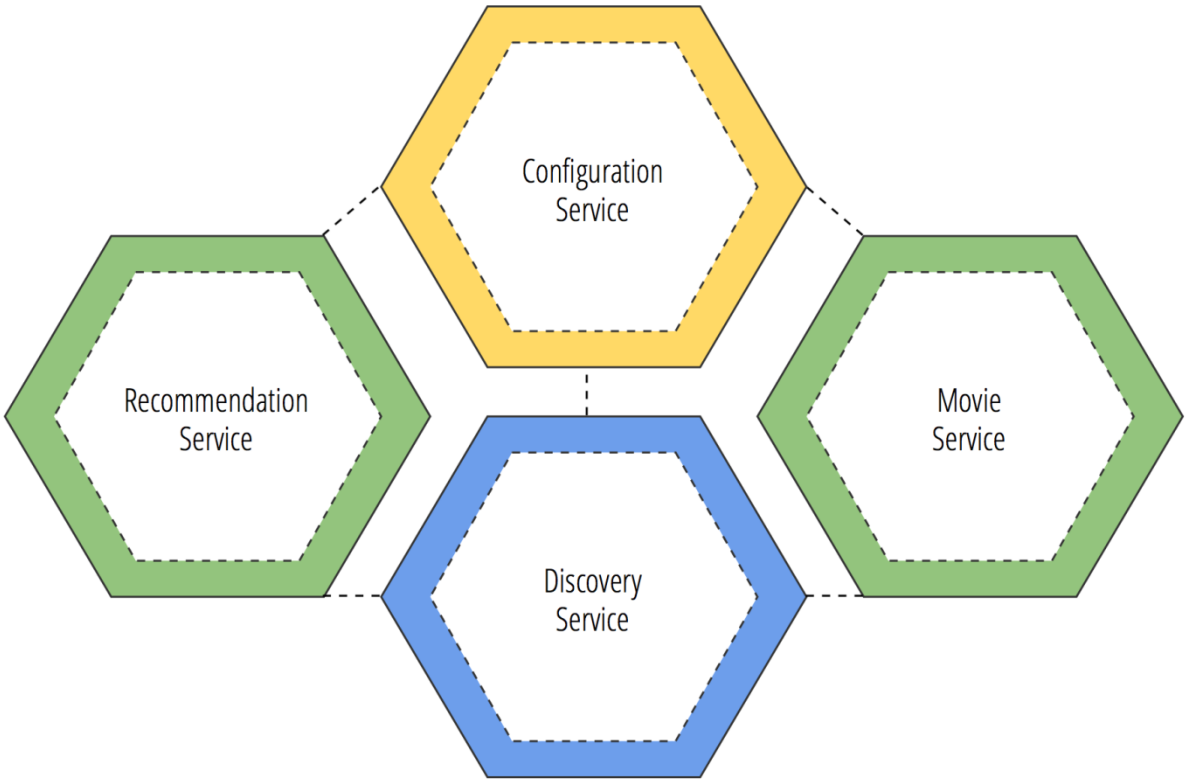

Рисунок 6.1 – Приклад налаштування мікросервісів зі зв’язками

Сервіс конфігурації знаходиться зверху, позначений жовтим кольором, і залежить від інших мікросервісів. Сервіс виявлення розташований знизу, виділений синім кольором, і також залежить від від інших мікросервісів.

Зеленим кольором позначено два мікросервіси, які реалізують основні функції прикладу: сервіс фільмів і рекомендацій.

6.3.1. СЕРВІС КОНФІГУРАЦІЇ (CONFIGURATION SERVICE)

Сервіс конфігурації є життєво важливим компонентом будь-якої MSA. На основі методології створення таких додатків, конфігурації для мікросервісів повинні зберігатися в навколишньому середовищі, а не в проекті.

Сервіс конфігурації має важливе значення, оскільки він обробляє конфігурації для всіх сервісів за допомогою простого виклику сервісу точка-точка (P2P) для отримання цих конфігурацій.

Припустимо, що у нас є декілька середовищ для розгортання. Якщо існують проміжне середовище та продакшн середовище, то конфігурації для них будуть відрізнятися. Сервіс конфігурації може мати власний репозиторій Git для конфігурацій цього середовища. Жодне з інших середовищ не матиме доступ до цієї конфігурації, вона доступна тільки для сервісу конфігурації, що працює в цьому середовищі (рис.6.2).

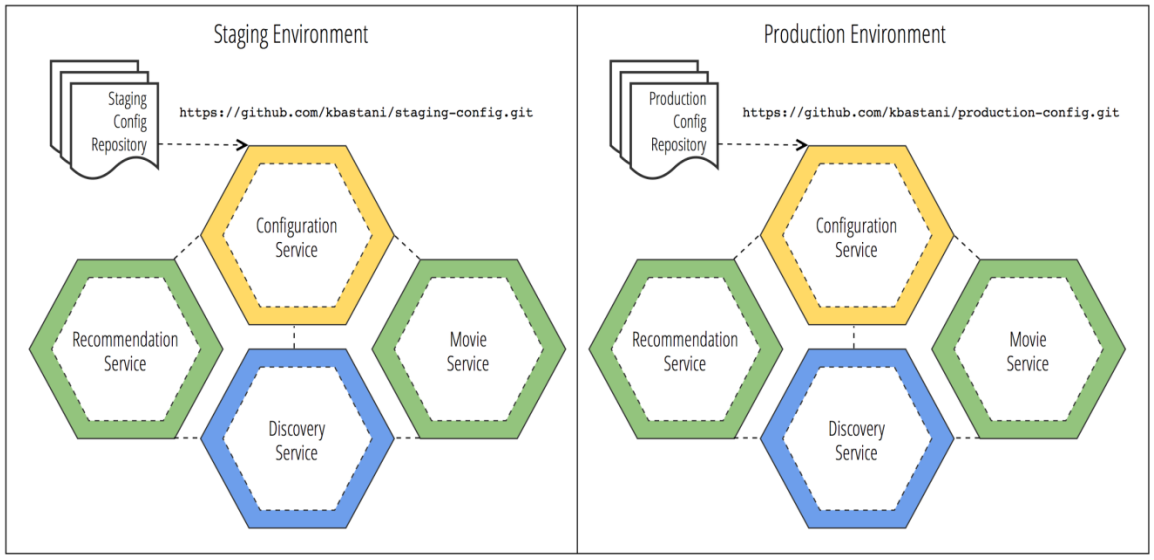

Рисунок 6.2 – Власний репозиторій сервісу конфігурації

Коли сервіс конфігурації запускається, він буде посилатися на шлях файлів конфігурації і надає їх мікросервісам, яким вони потрібні. Кожен мікросервіс може мати свій конфігураційний файл налаштований виходячи зі специфіки навколишнього середовища, в якому він працює. При цьому конфігурація знаходиться в одному місці (Git-репозиторій) і може бути переглянута без необхідності перезапускати сервіс, щоб змінити конфігурацію.

З кінцевими точками управління, доступних в Spring Cloud, можна внести зміни конфігурації із навколишнього середовища, після чого сигнал поступить до сервісу виявлення, що змусить всі сервіси надати нові, актуальні конфігурації.

6.3.2. СЕРВІС ВИЯВЛЕННЯ(DISCOVERY SERVICE)

Сервіс виявлення є ще одним важливим компонентом MSA. Він здійснює ведення списку екземплярів сервісів, які доступні для роботи в кластері. У додатках, виклики сервіс-сервіс здійснюється за допомогою клієнтів. Для цього прикладу проекту було використано Spring Cloud Feign, клієнт-орієнтований API для RESTful мікросервісів.

```
@FeignClient("movie")
public interface MovieClient {
   @RequestMapping(method = RequestMethod.GET, value = "/movies")
   PagedResources findAll();


   @RequestMapping(method = RequestMethod.GET, value = "/movies/{id}")
   Movie findById(@RequestParam("id") String id);
   @RequestMapping(method = RequestMethod.POST, value = "/movies",
     produces = MediaType.APPLICATION_JSON_VALUE)
   void createMovie(@RequestBody Movie movie);
}
```

У прикладі коду, який приведений вище, створюється клієнт Feign, який мапиться до методів REST API, які надаються сервісом кіно. Використовуючи анотацію @FeignClient, спочатку вказуємо, що потрібно створити клієнтський API для мікросервісу фільмів. Далі вказується мапінг сервісу, який буде використовуватись. Це здійснюється, оголосивши шаблони URL над методами для маршрутів для REST API.

Найлегша частина - створення Feign клієнтів, оскільки лише потрібно знати ідентифікатор сервісу, клієнт якого потрібно створити. URL сервісів автоматично налаштовується під час виконання, так як кожен мікросервіс в кластері буде реєструватися в сервісі виявлення з його ServiceID при запуску.

Те ж саме вірно і для всіх інших служб. Все, що потрібно знати - це ServiceID сервісу, з яким потрібен зв’язок, а все інше буде автоматично налаштовано Spring.

6.3.3. GATEWAY API

Сервіс Gateway API є також дуже важливим компонентом, якщо потрібно створити кластер сервісів. Зелені шестикутники, які приведені на рисунку нижче - це сервіси управління даними, які керують їх власними об’єктами домену і навіть мають свої власні БД. При додаванні сервісу Gateway API, можна створити проксі-сервер кожного маршруту API, які надаються зеленими сервісами (рис.6.3).

Рисунок 6.3 – Gateway API

Припустимо, що і сервіс рекомендації, і сервіс фільму надають свої власні REST API над об'єктами домену, якими вони керують. Gateway API виявить ці сервіси через сервіс виявлення і введе проксі на основі маршрутів методів API від інших сервісів. Таким чином, як сервіс рекомендації і сервіс фільмів матиме повне визначення маршрутів, доступних на місцевому рівні з усіх мікросервісів, які надають REST API. Gateway API буде повторно направляти запит на екземпляри сервісів, які мають маршрут, що запитується через HTTP.

6.4. ПРИКЛАД ПРОЕКТУ

В даній роботі реалізовано приклад проекту, який демонструє приклад практичного застосування MSA в хмарно-орієнтованому додатку на Spring Cloud.

Продемонстровано наступні поняття:

- Тестування інтеграції з використанням Docker
- Багатомовність
- MSA
- Виявлення сервісів
- Gateway API

6.4.1. DOCKER

Кожен сервіс побудовано і розгорнуто з використанням Docker. Тестування інтеграції може бути зроблено на комп’ютері розробника, використовуючи Docker compose[12].

6.4.2. БАГАТОМОВНІСТЬ

Однією з основних концепцій цього проекту є те, як багатомовність може бути застосована на практиці. Мікросервіси в проекті використовують свою власну базу даних при інтеграції з даними з інших сервісів через REST або шину повідомлень. Наприклад, можна мати окрему БД для кожного сервісу (Neo4j(графічна), MongoDB(документна), MySQL (реляційна)).

6.4.3. МІКРОСЕРВІСНА АРХІТЕКТУРА

Цей приклад демонструє, як з нуля створити проект на MSA, на відміну від монолітного додатку. Оскільки кожен мікросервіс в проекті є окремим модулем, розробники мають перевагу в тому, щоб працювати з кожним мікросервісом на своєму локальному комп'ютері. Додавання нового

мікросервісу також є легким, так як сервіс виявлення буде автоматично знаходити нові сервіси, запущені на кластері.

6.4.4. СЕРВІС ВІДКРИТТЯ

Цей проект містить два сервіса виявлення, один на Netflix Eureka, а інший використовує Consul від Hashicorp. Наявність декількох сервісів відкриття дає можливість використовувати один (Consul) в якості DNS постачальника для кластера, а інший (Eureka) в якості проксі для Gateway API.

6.4.5. GATEWAY API

Кожен мікросервіс координуватиме свої дії з Eureka для отримання маршрутів API всередині всього кластера. Використовуючи цю стратегію кожен мікросервіс в кластері може бути балансувальником навантаження і працювати через один Gateway API. Кожен сервіс буде автоматично виявляти і маршрутизувати запити API до сервісу, якому належить шлях. Цей метод однаково корисний при розробці користувацьких інтерфейсів, так як повний API платформи доступний через свій власний хост в якості проксі-сервера.

6.5. РОЗГОРТАННЯ ПРОЕКТУ НА DOCKER

Щоб приступити до роботи, потібно відвідати GitHub де знаходиться приклад проекту.

Далі потрібно склонувати репозиторій проекту на локальний комп’ютер. Після завантаження, потрібно буде використовувати Maven і Docker для компіляції і побудови образу локально.

6.5.1. ЗАВАНТАЖЕННЯ DOCKER

Потрібно завантажити Docker, якщо це ще не зроблено. Також необхідно встановити Docker Compose[12].

6.5.2. ВИМОГИ

Вимоги для запуску цього прикладу на локальній машині:

- Maven 3
- Java 8
- Docker
- Docker Compose

6.5.3. ВСТАНОВЛЕННЯ ПРОЕКТУ

Щоб скомпонувати проект з терміналу, потрібно виконати наступну команду в кореневому каталозі проекту

```
$ mvn clean install
```

Проект потім завантажить всі необхідні залежності і відбудеться компіляція кожного з артефактів проекту. Кожен сервіс буде побудований, а потім плагін Maven Docker автоматично додасть образ кожного сервісу локально в Docker. Він повинен бути запущений і доступний з командного рядка, в якому виконується команда mvn clean install.

Після того, як проект успішно скомпонується, отримаємо наступний вивід:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] spring-cloud-microservice-example-parent ..... SUCCESS [  0.268 s]
[INFO] users-microservice ..... SUCCESS [ 11.929 s]
[INFO] discovery-microservice ..... SUCCESS [  5.640 s]
[INFO] api-gateway-microservice ..... SUCCESS [  5.156 s]
[INFO] recommendation-microservice ..... SUCCESS [  7.732 s]
[INFO] config-microservice ..... SUCCESS [  4.711 s]
[INFO] hystrix-dashboard ..... SUCCESS [  4.251 s]
```

```
[INFO] consul-microservice ..... SUCCESS [ 6.763 s]
[INFO] movie-microservice ..... SUCCESS [ 8.359 s]
[INFO] movies-ui ..... SUCCESS [ 15.833 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

6.5.4. ЗАПУСК КЛАСТЕРА З DOCKER COMPOSE

Тепер, коли кожен з образів був успішно побудований, ми можемо за допомогою Docker Compose розгорнути наш кластер. Для цього в прикладі проекту міститься налаштований Docker Compose YAML файл.

З кореневого каталогу проекту, потрібно перейти до директорії spring-cloud-microservice-example/docker.

Тепер для запуску кластера мікросервісів, потрібно виконати наступну команду:

```
$ docker-compose up
```

Якщо все налаштовано правильно, кожен з контейнерів, які ми раніше побудували буде запущений в межах їх власної VM контейнера на Docker і об'єднані в мережу для автоматичного виявлення сервісу. В консолі з’явиться активний вивід повідомлень з кожного сервісу, оскільки вони послідовно запускаються.Це може тривати декілька хвилин, в залежності від машини, на якій запущено програму.

Після того як запуск буде завершено, можна перейти на хост Eureka і подивитися, які послуги зареєструвалися в сервісі виявлення.

Потрібно скопіювати та вставити наступну команду в термінал, в якому Docker може отримати доступ за допомогою змінної оточення $DOCKER_HOST

```
$ open $(echo \"$(echo $DOCKER_HOST)\"|
  \sed 's/tcp:\/\//http:\/\//g'|
  \sed 's/[0-9]\{4,\}/8761/g'|
  \sed 's/\"//g')
```

Якщо Eureka правильно запущена, відкриється вікно браузера з панеллю сервісу Eureka, як показано нижче на рисунку.

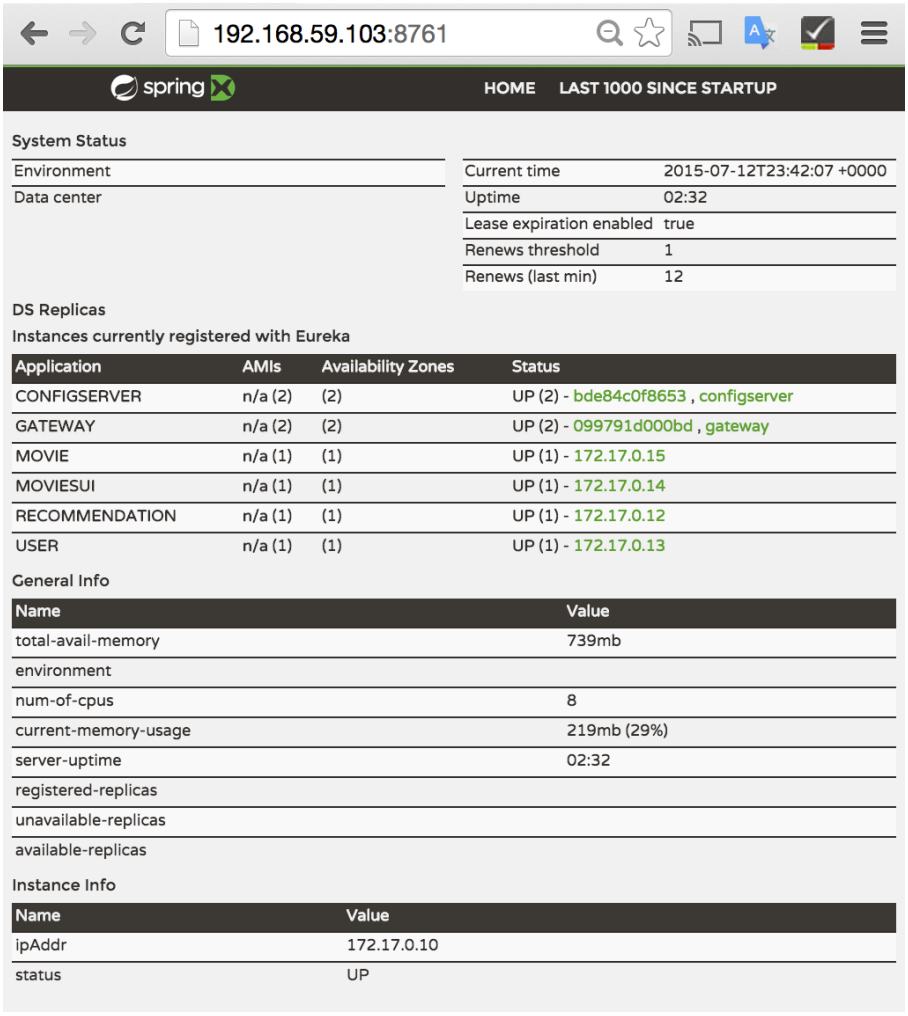

Рисунок 6.4 – Панель управління Eureka

Можна бачити кожен з екземплярів та статус сервісів, які працюють. Також можна отримати доступ до одного із сервісів, наприклад сервіс фільмів, виконавши команду

```
$ open $(echo \"$(echo $DOCKER_HOST)/movie\"|
 \sed 's/tcp:\/\//http:\/\//g'|
 \sed 's/[0-9]\{4,\}/10000/g'|
 \sed 's/\"//g')
```

Ця команда перейде до кінцевої точки Gateway API і проксі REST API, який надається сервісом фільмів. Ці REST API були налаштовані на використання HATEOAS, який підтримує автоматичне виявлення всіх функціональних можливостей сервісу, як вбудовані в посилання. Отримаємо наступний результат (JSON опис):


```
{
 "_links" : {
  "self" : {
   "href" : "http://192.168.59.103:10000/movie"
  },
  "resume" : {
   "href" : "http://192.168.59.103:10000/movie/resume"
  },
  "pause" : {
   "href" : "http://192.168.59.103:10000/movie/pause"
  },
  "restart" : {
   "href" : "http://192.168.59.103:10000/movie/restart"
  },
  "metrics" : {
   "href" : "http://192.168.59.103:10000/movie/metrics"
  },
  "env" : [ {
   "href" : "http://192.168.59.103:10000/movie/env"
```

```
}, {
  "href" : "http://192.168.59.103:10000/movie/env"
} ],
"archaius" : {
  "href" : "http://192.168.59.103:10000/movie/archaius"
},
"beans" : {
  "href" : "http://192.168.59.103:10000/movie/beans"
},
"configprops" : {
  "href" : "http://192.168.59.103:10000/movie/configprops"
},
"trace" : {
  "href" : "http://192.168.59.103:10000/movie/trace"
},
"info" : {
  "href" : "http://192.168.59.103:10000/movie/info"
},
"health" : {
  "href" : "http://192.168.59.103:10000/movie/health"
},
"hystrix.stream" : {
  "href" : "http://192.168.59.103:10000/movie/hystrix.stream"
},
"routes" : {
  "href" : "http://192.168.59.103:10000/movie/routes"
},
"dump" : {
  "href" : "http://192.168.59.103:10000/movie/dump"
```

```
    },
    "refresh" : {
      "href" : "http://192.168.59.103:10000/movie/refresh"
    },
    "mappings" : {
      "href" : "http://192.168.59.103:10000/movie/mappings"
    },
    "autoconfig" : {
      "href" : "http://192.168.59.103:10000/movie/autoconfig"
    }
  }
}
```


6.6. ВИСНОВКИ

У цьому розділі було створено приклад додатку, який розкриває наступні поняття мікросервісної архітектури:

- Service Discovery
- Externalized Configuration
- API Gateway
- Service Orchestration з Docker Compose

ВИСНОВКИ

В даній роботі було детально розглянуто сучасний підхід до побудови додатків, який базується на мікросервісній архітектурі. Його було також порівняно із добре відомою, та яка використовується майже скрізь, монолітною архітектурою.

Монолітна архітектура дуже хороший підхід при розробці ПЗ. Але з ростом і розвитком світу технологій, ростуть і потреби, які покладаються на дані додатки. Через це моноліт перестає бути актуальним, оскільки він є не дуже гнучким при розробці та підтримці. Також великою проблемою є те, що команди, які працюють за сучасними гнучкими методологіями розробки, не можуть ефективно розробляти програми із використанням даної архітектури. Саме тому з'явились і активно розвиваються мікросервіси, які вирішують проблеми монолітних програм.

В роботі було детально описано всі принципи створення мікросервісної системи. Було наведено та розібрано декілька прикладів таких систем. Описано основні підходи та шаблони проектування при створенні таких додатку.

Було ретельно проаналізовано доступні рішення (фреймворки) для створення мікросервісної архітектури, підібрано найкращі з них та порівняно за основними параметрами, які можуть вплинути на розробку. При аналізі даних рішень було розгорнуто примітивні додатки, оцінено складність розгортання та реалізації і враховано якомога більше факторів які в подальшому можуть вплинути не тільки на розробку, а й на безпеку, стабільність тощо.

Базуючись на цьому було реалізовано власну примітивну систему, яку можна далі легко розвивати та застосовувати в навчальних цілях. При створенні системи було використано дуже зручний та актуальний на даний час інструмен для віртуалізації, як Docker.

На основі цієї системи було створено лабораторний практикум по курсу «Основи сервіс-орієнтованих архітектур», саме з акцентом на мікросервісну

архітектуру. В майбутньому, при введені даного курсу можливим є на основі цієї роботи створити повноцінні розгорнуті методичні матеріали, які будуть використовуватись для розвитку практичних навичок при роботі з даною архітектурою. Оскільки на даний момент не існує україномовних джерел інформації по цій темі – робота є актуальною в цьому плані. В подальшому може бути застосована, як короткі методичні вказівки для проведення лабораторного практикуму на кафедрі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ньюмен С. Создание микросервисов / Ньюмен С. — СПб.: Питер, 2016. — 304 с.
2. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74p.
3. Офіційний сайт Microservices.io. – Режим доступу: http://www.microservices.io.microservices.io – Дата доступу : 10.11.2016.
4. Introduction to microservices. – Режим доступу: https://www.nginx.com/blog/introduction-to-microservices/ – Дата доступу : 15.11.2016.
5. Microservices design patterns – Режим доступу: https://www.javacodegeeks.com/2015/04/microservice-design-patterns.html – Дата доступу : 12.11.2016.
6. Using an API Gateway – Режим доступу: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/?utm_source=introduction-to-microservices&utm_medium=blog&utm_campaign=Microservices – Дата доступу : 23.11.2016.
7. Inter-Process Communication in a Microservices Architecture – Режим доступу: https://www.nginx.com/blog/building-microservices-inter-process-communication/?utm_source=building-microservices-using-an-api-gateway&utm_medium=blog&utm_campaign=Microservices – Дата доступу : 5.12.2016.
8. Service Discovery – Режим доступу: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/?utm_source=building-microservices-inter-process-communication&utm_medium=blog&utm_campaign=Microservices – Дата доступу : 8.12.2016.

9. Event-driven Data Managment – Режим доступу: https://www.nginx.com/blog/event-driven-data-management-microservices/?utm_source=service-discovery-in-a-microservices-architecture&utm_medium=blog&utm_campaign=Microservices – Дата доступу : 10.12.2016.
10. Офіційний сайт Spark Java. – Режим доступу: http://sparkjava.com/ – Дата доступу : 20.12.2016.
11. Офіційний сайт Spring Framework. – Режим доступу: https://spring.io/ – Дата доступу : 25.12.2016.
12. Офіційний сайт Docker. – Режим доступу: https://www.docker.com/ – Дата доступу : 5.01.2017.

ДОДАТОК А. ЛАБОРАТОРНИЙ ПРАКТИКУМ

Лабораторна робота № 1. Налаштування середовища для роботи з мікросервісами

Підготовка середовища для розробки додатків з використанням мікросервісів

Мета роботи: Підготувати середовище розробки для виконання наступних лабораторних робіт.

Задача: провести необхідні налаштування середовища для роботи з мікросервісами.

Короткі теоретичні відомості

Необхідні компонети:

Linux (Ubuntu)

Maven 3

Java 8

Docker

Docker Compose

1. Встановлення JDK

Для встановлення потрібної версії JDK, в терміналі Ubuntu потрібно виконати наступну команду:

sudo apt-get install oracle-java8-installer

2. Встановлення Maven

Для встановлення Maven 3 потрібно, щоб на комп’ютері було встановлено JDK8, щ обуло зроблено у попередньому пункті.

Далі потрібно виконати наступну команду :

sudo apt-get install maven

І для перевірки того, що встановлення пройшло успішно виконати:

mvn -version

3. Встановлення Docker.

Докер має дві важливих вимоги до встановлення:

Докер працює тільки на 64-розрядній системі Linux.

Докер вимагає версію ядра Linux 3.10 або вище.

Щоб перевірити поточну версію ядра, відкрийте термінал і виконайте наступну команду, яка виведе:

$ uname -r

3.11.0-15-generic

Перед встановленням обов’язково відвідати сторінку https://docs.docker.com/engine/installation/linux/ubuntulinux/#/install-the-latest-version та уважно підготувати середовище для встановлення.

Зайти в термінал з правами Sudo

Оновити індекси APT пакетів, виконавши наступну команду:

sudo apt-get update

Встановити Docker, виконавши наступну команду:

sudo apt-get install docker-engine

Запустити демон Docker, виконавши наступну команду:

sudo service docker start

Переконатися в тому, що Docker встановлений правильно, запустивши образ hello-world, виконавши наступну команду:

sudo docker run hello-world

Ця команда завантажує тестовий образ і запускає його в контейнері. При запуску контейнера, він виводить інформаційне повідомлення і завершує роботу.

4. Встановлення Docker Compose.

Для встановлення Docker Compose потрібно, щоб на комп’ютері було спочатку встановлено Docker, що було описано в попередньому пункті.

Перейти на сторінку Compose https://github.com/docker/compose/releases

Слідуючи інструкціям на даній сторінці запустити команду curl в терміналі (Якщо ви отримуєте помилку "Permission denied", директорія /usr/local/bin, ймовірно, недоступна для запису, і необхідно встановити Compose від суперюзера. Наступні команди запускати за допомогою sudo -i). Приклад команди:

curl -L "https://github.com/docker/compose/releases/download/1.9.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Змінити права, виконавши команду:

chmod +x /usr/local/bin/docker-compose

Протестувати правильність встановлення, виконавши команду:

docker-compose --version

docker-compose version: 1.9.0

Альтернативні варіанти встановлення Compose можна знайти, перейшовши за посиланням https://docs.docker.com/compose/install/

Завдання

1. Встановити всі необхідні компоненти середовища.

2. Описати процес встановлення Docker.

3. Продемонструвати функціональність, запустивши тестовий образ.

4. Описати процес встановлення Docker Compose.

Зміст звіту

1. Мета роботи.

2. Завдання роботи.

3. Оформлення результатів роботи.

4. Опис процесів налаштування середовища.

5. Висновки.

Контрольні питання

1. Що таке Docker та для чого він потрібен?

2. Що таке Docker Compose та для чого він потрібен?

3. Пояснити необхідність використання даного інструментарію в контексті мікросервісної архітектури?

Лабораторна робота № 2. Запуск тестового мікросервісного додатку

Мета роботи: набути практичних навичок при розгортанні мікросервісного додатку на локальному комп’ютері.

Задача: розгорнути базований на Spring фреймворку мікросервісний додаток на локальному комп’ютері із встановленим та налаштованим у Лабораторній роботі №1 середовищем.

Короткі теоретичні відомості

Приклад додатку на Java

Приклад надано для мови програмування Java, наявність будь-якого середовища розробки не обов’язкова. Команди та шляхи до файлів вказані для Linux-систем.

1. Загальна структура файлів деякого проекту Project:

Приклад структури ресурсів демо-програми spring-cloud-microservice-example:

https://github.com/kbastani/spring-cloud-microservice-example

Склонуємо або скачаємо проект наданий за посиланням та ознайомимось з його структурою. Потрібно звернути увагу на основні елементи мікросервісної архітектури, та знайти їх розміщення у директорії проекту.

2. Вихідні файли – код простого сервісу Application.java (spring-cloud-microservice-example-master/movie-microservice/src/main/java/data)

```
package data;


import org.neo4j.graphdb.GraphDatabaseService;

import org.neo4j.graphdb.factory.GraphDatabaseFactory;

import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.context.annotation.Bean;
import org.springframework.data.neo4j.config.EnableNeo4jRepositories;
import org.springframework.data.neo4j.config.Neo4jConfiguration;

@SpringBootApplication
@EnableNeo4jRepositories
@EnableDiscoveryClient
@EnableZuulProxy
@EnableHystrix
public class Application extends Neo4jConfiguration {

    public Application() {
        setBasePackage("data");
    }

    @Bean(destroyMethod = "shutdown")
    public GraphDatabaseService graphDatabaseService() {
```

```
        return new
GraphDatabaseFactory().newEmbeddedDatabase("target/movie.db");
    }

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

}
```

3. Файл конфігурації сервісу – application.yml (spring-cloud-microservice-example-master/movie-microservice/src/main/resources/web.xml)

```
server:
  port: 9005

eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    preferIpAddress: true

ribbon:
```

```
  eureka:
    enabled: true
```

4. Додатковий файл Docker конфігурації для сервісу – Dockerfile (spring-cloud-microservice-example-master/movie-microservice/src/main/docker/)

```
FROM java:8
VOLUME /tmp
ADD movie-microservice-0.1.0.jar app.jar
RUN bash -c 'touch /app.jar'
EXPOSE 9000
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

Обов’язково ознайомитись з вихідними файлами та файлами конфігурації для кожного сервісу.

4.Файл Docker compose конфігурації для всього проекту (spring-cloud-microservice-example-master/docker)

```
hystrix:
  image: kbastani/hystrix-dashboard
  ports:
   - "7979:7979"
  links:
   - gateway
   - discovery
discovery:
  image: kbastani/discovery-microservice
```

```
  ports:
   - "8761:8761"
configserver:
  image: kbastani/config-microservice
  ports:
   - "8888:8888"
  links:
   - discovery
gateway:
  image: kbastani/api-gateway-microservice
  ports:
   - "10000:10000"
  links:
   - discovery
   - configserver
   - user
   - movie
   - recommendation
user:
  image: kbastani/users-microservice
  links:
   - discovery
```

```
    - configserver
movie:
  image: kbastani/movie-microservice
  links:
    - discovery
    - configserver
recommendation:
  image: kbastani/recommendation-microservice
  links:
    - discovery
    - configserver
moviesui:
  image: kbastani/movies-ui
  ports:
     - "9006:9006"
  links:
    - discovery
    - configserver
```

5.Компіляція

Щоб скомпонувати проект з терміналу, потрібно виконати наступну команду в кореневому каталозі проекту

$ mvn clean install

Проект потім завантажить всі необхідні залежності і відбудеться компіляція кожного з артефактів проекту. Кожен сервіс буде побудований, а потім плагін Maven Docker автоматично додасть образ кожного сервісу локально в Docker. Він повинен бути запущений і доступний з командного рядка, в якому виконується команда mvn clean install.

Після того, як проект успішно скомпонується, отримаємо наступний вивід:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] spring-cloud-microservice-example-parent ..... SUCCESS [  0.268 s]
[INFO] users-microservice ..... SUCCESS [ 11.929 s]
[INFO] discovery-microservice ..... SUCCESS [  5.640 s]
[INFO] api-gateway-microservice ..... SUCCESS [  5.156 s]
[INFO] recommendation-microservice ..... SUCCESS [  7.732 s]
[INFO] config-microservice ..... SUCCESS [  4.711 s]
[INFO] hystrix-dashboard ..... SUCCESS [  4.251 s]
[INFO] consul-microservice ..... SUCCESS [  6.763 s]
[INFO] movie-microservice ..... SUCCESS [  8.359 s]
[INFO] movies-ui ..... SUCCESS [ 15.833 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

6. Запуск кластера мікросервісів та його перевірка

Тепер, коли кожен з образів був успішно побудований, ми можемо за допомогою Docker Compose розгорнути наш кластер. Для цього в прикладі проекту міститься налаштований Docker Compose YAML файл.

З кореневого каталогу проекту, потрібно перейти до директорії spring-cloud-microservice-example/docker.

Тепер для запуску кластера мікросервісів, потрібно виконати наступну команду:

$ docker-compose up

Після того як запуск буде завершено, можна перейти на хост Eureka і подивитися, які послуги зареєструвалися в сервісі виявлення.

Потрібно скопіювати та вставити наступну команду в термінал, в якому Docker може отримати доступ за допомогою змінної оточення $DOCKER_HOST

```
$ open $(echo \"$(echo $DOCKER_HOST)\"|
  \sed 's/tcp:\/\//http:\/\//g'|
  \sed 's/[0-9]\{4,\}/8761/g'|
  \sed 's/\"//g')
```

Якщо Eureka правильно запущена, відкриється вікно браузера з панеллю сервісу Eureka, як показано нижче на рисунку.

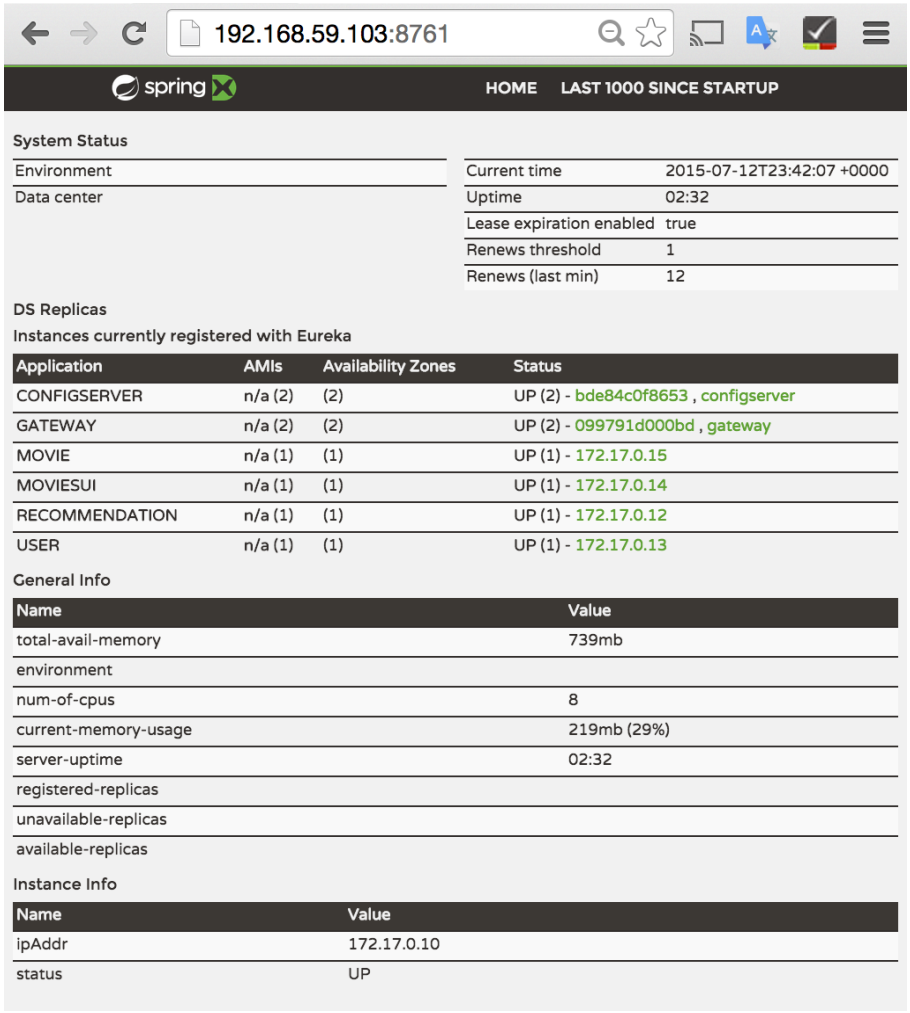

Рисунок 1 – Панель управління

Можна бачити кожен з екземплярів та статус сервісів, які працюють. Також можна отримати доступ до одного із сервісів, наприклад сервіс фільмів, виконавши команду

```
$ open $(echo \"$(echo $DOCKER_HOST)/movie\"|
        \sed 's/tcp:\/\//http:\/\//g'|
        \sed 's/[0-9]\{4,\}/10000/g'|
        \sed 's/\"//g')
```

Завдання

1. Описати структуру мікросервісного додатку.

2. Описати процес розгортання прикладу. Навести копії екрану.

4. Протестувати додаток. Отримати JSON-опис, звернувшись до будь-якого з мікросервісів.

Зміст звіту

1. Мета роботи.

2. Завдання роботи.

3. Оформлення результатів роботи.

4. Опис коду програм мікросервісів.

5. Опис процедури розгортання додатку на локальному комп’ютері.

6. Опис процесу тестування.

7. Висновки.

Контрольні питання

1. Яка структура мікросервісного додатку?

2. Для чого застосовується Spring фреймворк при побудові мікросервісів?

3. Для чого і як застосовується Docker compose при розгортанні мікросервісного додатку?

Лабораторна робота № 3. Створення власного мікросервісу.

Мета роботи: вивчити структуру мікросервісного додатку та принцип комунікації сервісів у ньому. Створити власний примітивний мікросервіс.

Задача: на основі прикладу, описаного у Лабораторній роботі №2, створити власний мікросервіс.

Завдання

1. Додати до прикладу, описаного у попередній роботі, власний мікросервіс.

2. Підготувати опис кроків, які було зроблено при підключенні власного сервісу.

3. Протестувати роботу власного сервісу та проілюструвати це.

Зміст звіту

1. Мета роботи.

2. Завдання роботи.

3. Оформлення результатів роботи.

4. Опис розробки власного мікросервісу та його тестування.

5. Висновки.

Контрольні питання

1. Для чого потрібен API Gateway та принцип його роботи?

2. Для чого потрібен Service Discovery та принцип його роботи?

3. На що необхідно звернути увагу підключення нового мікросервісу?

Лабораторна робота № 4. Розгортання додатку, в мікросервісах якого застосовуються різні технології

Мета роботи: ознайомитись з можливістю мікросервісної архітектури, яка дозволяє застосовувати свою технологію кожному сервісу.

Задача: на основі попередніх робіт розгорнути «багатомовний» додаток на локальному комп'ютері, протестувати його та дослідити принцип роботи з різними технологіями на рівні мікросервісів.

Короткі теоретичні відомості

Найпростіший варіант ознайомлення з процесом розробки та розгортання мікросервісного додатку без прив'язки до стеку технологій – це вивчити простий приклад, який доступний на git-репозиторії:

git clone

https://github.com/GoogleCloudPlatform/appengine-endpoints-helloworld-java-maven

Його розгортання відбувається за кроками, описаними у Лабораторній роботі №2. А також для успішного виконання завдання потрібне середовище налаштоване згідно із завданням Лабораторної роботи №1. Додаткову інформацію можна знайти за посиланням http://www.kennybastani.com/2015/08/polyglot-persistence-spring-cloud-docker.html .

Завдання

1. Заватажити та детально ознайомитись зі структурою наданого тестового прикладу.

2. Підготувати опис структури та компонентів.

3. Підготувати взаємодії сервісів у системі даного типу.

4. Розгорнути даний додаток на локальному комп’ютері.

5. Виконати тестування роботи кожного мікросервісу.

Зміст звіту

1. Мета роботи.

2. Завдання роботи.

3. Оформлення результатів роботи.

4. Опис взаємодії сервісів та компонентів системи.

5. Результат тестування кожного мікросервісу (JSON-опис).

6. Висновки.

Контрольні питання

1. Які переваги надає використання своїх технологій у кожному мікросервісі?

2. Які переваги має формат JSON? Які є недоліки?

3. Як відбувається взаємодія між мікросервісами такого типу?