НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»
ім. Ігоря Сікорського

Навчально-науковий комплекс «Інститут прикладного системного аналізу»
(повна назва інституту/факультету)

Кафедра Системного проектування
(повна назва кафедри)

«До захисту допущено»
Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)
"___"_____20__р.

Дипломна робота

на здобуття ступеня бакалавра

з напряму підготовки 6.050101 Комп'ютерні науки
(код і назва)

на тему: Побудова мікросервісів за допомогою різних програмних платформ

_____

Виконав (-ла): студент (-ка) _4_ курсу, групи _ДА-32_
(шифр групи)

Зеленін Віктор Анатолійович _____ _____
(прізвище, ім'я, по батькові) (підпис)

Керівник к.т.н., доц. Безносик О.Ю. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант економічний _____ _____
_____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Нормоконтроль старший викладач Британов О.А. _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

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

Київ – 2017 року

Національний технічний університет України
«Київський політехнічний інститут»
ім. Ігоря Сікорського

Інститут (факультет) ННК «Інститут прикладного системного аналізу
(повна назва)

Кафедра Системного проектування
(повна назва)

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки 6.050101 Комп'ютерні науки
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)

«___»_____20__р.

ЗАВДАННЯ
на дипломну роботу студенту
Зеленіну Віктору Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Побудова мікросервісів за допомогою різних програмних платформ _____
_____
_____,
керівник роботи Безносик О. Ю., к.т.н., доц._____,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом роботи _____10.06.2017_____

3. Вихідні дані до роботи

1.Мікросервісна архітектура як приклад сервіс-орієнтованого проектування

2. Платформи для побудови мікросервісів для різних мов програмування

3. Розробка мікросервісів на мовах Java та Python.

4. Зміст роботи

1. Дослідити основні принципи побудови мікросервісних додатків в порівнянні з сервіс-орієнтованим підходом.
2. Порівняти мікросервісну архітектуру з монолітною архітектурою.
3. Проаналізувати та порівняти існуючі програмні платформи для побудови мікросервісних додатків.
4. Спроектувати структуру тестового додатку на базі мікросервісної архітектури
5. Розробити тестову програмну систему.
6. Здійснити функціонально-вартісний аналіз програмного продукту.
7. Проаналізувати можливість існування даних концепцій на прикладі розробленої системи.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

1. Типова архітектура для монолітних додатків – плакат.
2. Типова архітектура для мікросервісних додатків – плакат.
3. Порівняльна характеристика різних програмних платформ – плакат.
4. Структура тестової програмної системи – плакат.

6. Консультанти розділів роботи*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н.В., к.е.н.		

7. Дата видачі завдання ___01.02.2017 _____

* Консультантом не може бути зазначено керівника дипломної роботи.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	01.02.2017	
2	Збір інформації	15.02.2017	
3	Вивчення принципів розподілених систем	01.03.2017	
4	Вивчення концепцій мікросервісної архітектури	30.03.2017	
5	Дослідження різних програмних платформ для побудови мікросервісів	29.04.2017	
6	Створення архітектури тестового додатку	10.05.2017	
7	Розробка тестової програми на базі мікросервісів	20.05.2017	
8	Оформлення дипломної роботи	31.05.2017	
9	Отримання допуску до захисту та подача роботи в ДЕК	08.06.2017	

Студент _____ ____В.А.Зеленін_____
(підпис) (ініціали, прізвище)

Керівник роботи _____ ____О.Ю.Безносик____
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

бакалаврської дипломної роботи Зеленіна Віктора Анатолійовича на тему:
“Побудова мікросервісів за допомоги різних програмних платформ”

Дана дипломна робота присвячена дослідженню основних підходів до побудови мікросервісної архітектури. Метою роботи є дослідження існуючих концепцій для створення мікросервісних систем, ознайомлення з наявними програмними платформами та інструментами, які дозволяють швидко та ефективно розгортати дані системи. А також реалізація тестового додатку для підтвердження працездатності даного архітектурного стилю.

У роботі було розглянуто відомі програмні платформи для деяких мов програмування, що використовуються для розробки мікросервісних систем. Проаналізовано інструменти, які допомагають налагодити взаємодію між сервісами. Розроблено додаток з використанням мов Java та Python. Також було детально перевірено результати роботи створеної системи. Дану роботу пропонується використовувати в якості методичного матеріалу розробниками ПЗ під час проектування та розробки мікросервісних систем.

Загальний обсяг роботи: 96 сторінок, 27 рисунків, 7 таблиць, 19 посилань.

Ключові слова: мікросервісна архітектура, сервіс-орієнтоване проектування, фреймворк, горизонтальне масштабування, Spring framework, Python Flask, безперервна інтеграція, контейнеризація.

АННОТАЦИЯ

бакалаврской дипломной работы Зеленина Виктора Анатольевича на тему:
“Построение микросервисов при помощи различных программных платформ”

Данная дипломная работа посвящена исследованию основных подходов к построению микросервисной архитектуры. Целью работы является исследование существующих концепций для создания микросервисных систем, ознакомление с имеющимися программными платформами и инструментами, которые позволяют быстро и эффективно разворачивать данные системы. А также реализация тестового приложения для подтверждения работоспособности данного архитектурного стиля.

В работе были рассмотрены известные программные платформы для некоторых языков программирования, используемых для разработки микросервисных систем. Проанализированы инструменты, которые помогают наладить взаимодействие между сервисами. Было разработано приложение с использованием языков Java и Python. Также детально проверено результаты работы созданной системы. Данную работу предлагается использовать в качестве методического материала разработчиками ПО при проектировании и разработке микросервисных систем.

Общий объем работы: 96 страниц, 27 рисунков, 7 таблиц, 19 ссылок.

Ключевые слова: микросервисная архитектура, сервис-ориентированное проектирование, фреймворк, горизонтальное масштабирование, Spring framework, Python Flask, непрерывная интеграция, контейнеризация.

ABSTRACT

for the bachelor thesis of Zelenin Victor Anatolievich on “Building microservices using different programming platforms”

Thesis is devoted to the researching main approaches of building microservice architecture. The aim is to deeply analyze existing principles for creating microservice systems and acquaint with the available programming platforms and tools which allow deploying these systems quickly and efficiently. Also test application was implemented to proof the concept of this architecture style.

In the course of the thesis popular programming platforms used for creating microservice systems by several programming languages were investigated. Also tools which are used for establish communication between services were analyzed. An application was developed using Java and Python. Also the results of the created systems were verified in details.

The results of research are encouraged to use as teaching materials for software developers during designing and development of microservice systems.

Total volume of work: 96 pages, 27 figures, 7 tables, 19 links.

Keywords: Microservice architecture, service-oriented design, framework, horizontal scaling, Spring framework, Python Flask, continuous integration, containerization.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

MSA	MicroService Architecture (мікросервісна архітектура)
SOA	Service-Oriented Architecture (сервіс-орієнтована архітектура)
REST	Representational State Transfer, архітектурний стиль
ПЗ	Програмне забезпечення
RPC	Remote Procedure Call, протокол для взаємодії між комп'ютерами
БД	База даних
API	Application Programming Interface, набір інтерфейсів для взаємодії різнотипного програмного забезпечення
HTTP	Hyper Text Transfer Protocol, протокол передачі даних, на якому побудована мережа Інтернет
HTTPS	Hyper Text Transfer Protocol S ecure, захищений HTTP
JSON	JavaScript Object Notation, текстовий формат обміну даними
SOAP	Simple Object Access Protocol, протокол обміну структурованими повідомленнями в розподілених системах
CI	Continuous Integration
CD	Continuous Delivery
ООП	Об'єктно-орієнтоване програмування
SAML	Security Assertation Markup Language, мова розмітки, відкритийстандарт обміну даними аутентифікації та авторизації
AWS	Amazon Web Services, інфраструктура хмарних платформ
CGI	Common Gateway Interface, стандарт інтерфейсу, який використовується для зв'язку зовнішньої програми та серверу.
J2EE	Java Enterprise Edition, java платформа для корпоративних додатків
OSGi	Open Service Gateway Initative, специфікація модульної системи
GWT	Google Web Toolkit, Java-фреймворк
XML	eXtensible Markup Language, розширювана мова розмітки

ВСТУП

Проблема проектування та створення якісного програмного забезпечення є надзвичайно важливою у сучасному інформаційному світі. З розвитком IT-індустрії було знайдено багато різних підходів та концепцій до побудови складних програмних систем. Показником гарно побудованої програми є, звісно, її архітектура, яка правильно описує предметну область та є формальною моделлю системи. Архітектурою можна вважати набір певних структурних компонентів зв'язаних між собою, які задають поведінку всієї системи. Основною задачею архітектури є управління складністю, елегантне та доцільне відображення предметної області. Довгий час провідне місце займала так звана "монолітна архітектура". При даному підході вся система являє собою моноліт, який фізично розташовується на єдиній машині, запускається в одному процесі та виконує всі бізнес-операції системи.

Монолітний додаток піддається лише горизонтальному масштабуванню шляхом запуску декількох окремих серверів із кожним окремим монолітом. Але з плином часу знаходилися інші ідеї та підходи, саме таким стала сервіс-орієнтована архітектура (SOA), на відміну від монолітної системи, при SOA вся програма являє собою розподілену систему, яка обмінюється повідомленнями за певним протоколом. Вся система складається з набору незалежних сервісів, які фокусуються на власній задачі. SOA націлена на боротьбу з великими монолітними системами. Сама по собі ідея SOA чудова, але питання як правильно та якісно організовувати сервіс-орієнтовану архітектуру залишається відкритим. Основні вузькі місця відносяться до протоколів обміну даними, таких як SOAP, а також неправильні місця розділу системи.

Пізніше було запропоновано новий підхід до організації SOA, так звана мікросервісна архітектура (MSA). Мікросервісну архітектуру можна вважати підмножиною SOA, але все ж таки MSA відрізняється від класичного SOA. Основна відмінність — це невелика кількість кодової бази на кожен сервіс, в той час як в SOA не важливий об'єм кодової бази. Також важливим місцем для MSA є

те, що кожен сервіс має мати власний обмежений контекст для цієї предметної області.

Обмежень на кількість існуючих сервісів немає, але кожен сервіс має працювати лише над одною бізнес-задачею. Для обміну інформацією мікросервіси використовують стандартизовані протоколи передачі даних (наприклад, HTTP), як правило кожен сервіс має своє API для спілкування з іншими мікросервісами. Всі сервіси можуть бути написані на абсолютно різних мовах програмування та використовуючи будь-які бібліотеки, також має місце децентралізоване збереження даних, тобто кожен сервіс має свою власну базу даних.

Можна виділити основні переваги використання мікросервісної архітектури:

1. Низька зв’язність між основними компонентами системи та висока зчепленність коду в окремо взятому сервісі
2. Відносно просте розгортання, кожен сервіс розгортається незалежно від інших, на відміну від монолітного додатку, де вся система запускаться в єдиному процесі та динамічно вносити зміни не є можливим.
3. Просте масштабування системи. Ми можемо запускати будь-яку кількість сервісів на окремих серверах
4. Відмовостійкість. При виведенні з ладу одного сервісу вся програма ще може вірно працювати
5. Застосування різних мов програмування та технологій
6. Для створення мікросервісів можна використовувати компактні групи розробників, які є відповідальними за власний сервіс.

До мінусів мікросервісної архітектури можна віднести:

1. Відносна складність розробки, прямо пропорційно залежить від кількості обраних мов програмування та фреймворків.
2. Витрачаються додаткові ресурси на пересилання повідомлень між сервісами та на їх серіалізацію та десеріалізацію
3. Проблеми з версіонуванням

4. Відносно складне інтеграційне тестування

Мікросервісну архітектуру доцільно використовувати, якщо всю систему планується розготати в хмарі, оскільки монолітний додаток не можна зручно масшабувати горизонтально.

В цілому, мікросервісна архітектура не є “срібною кулею”, а вирішує лише певний набір задач і залежить від різних обставин. Тому в даній роботі буде розглянуто, які обмеження висуваються до MSA та доцільність використання цієї архітектури.

Метою даної роботи є дослідження побудови додатків на базі мікросервісів, огляд існуючих платформ для створення мікросервісів та їх порівняння. Для більш детального ознайомлення слід також розробити тестову програму для демонстрації роботи мікросервісної системи.

1 ПРОБЛЕМИ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Основні складнощі проектування

Проектування архітектури інформаційної системи є однією із найважливіших етапів створення проекту. Перш за все, встановимо, що мається на увазі під поняттям “архітектура інформаційної системи”. Існує безліч трактувань даного терміну, можемо описати архітектуру ІС як організацію системи через набір компонент, їх взаємовідносини та зв’язок зі зовнішнім середовищем [1]. Під час проектування системи як правило приймаються відповідні проектні рішення, після прийняття яких зміна поведінки ІС стає надзвичайно складною.

Основною ціллю програмної архітектури є боротьба з потенційною складністю, яка властива складним програмним системам, корпоративним додаткам. Як правило, складність зростає значно швидше ніж кількість програмного коду, тому якщо заздалегідь не передбачити якісну організацію майбутнього коду, то цілком ймовірно настане такий момент, коли підтримувати дану складність не буде можливим. Вдало розроблена архітектура заощадить велику кількість часу та зусиль. Гарно спроектована архітектурна модель має бути гнучкою в місцях, які потенційно мають найчастіше змінюватися чи розширюватись, але жорсткою в інших. Також правильно побудовану програму легко супроводжувати, тестувати та підтримувати. Можна виділити ключові моменти, які описують правильно побудовану архітектуру [2]:

- Ефективність та працездатність системи. Перш за все, ІС має вирішувати поставлені перед нею функціональні вимоги, при будь-яких обставинах.
- Гнучкість системи. З розвитком системи завжди будуть з’являтися та змінюватися вимоги. Показник правильно створеної архітектури — здатність швидко та зручно змінювати систему, тому при аналізі

предметної області та проектуванні моделі завжди необхідно оцінювати та знаходити місця, які потенційно будуть змінюватися, щоб в майбутньому не витрачати додатковий час на зміну підсистем, якщо це буде, взагалі, можливим.

- Розширюваність системи. Іншим важливим показником є здатність додавати нові сутності та функції, не змінюючи та не порушуючи її загальної структури та поведінки, при чому щоб на додавання нових функцій витрачалось найменш можлива кількість часу.
- Продуктивність. Швидкодія та високонавантаженність є однією з ключових вимог для деяких ІС. Програма повинна витримувати належне навантаження зі сторони користувачів та відповідати за допустимий проміжок часу. Одним із варіантів рішення є здатність до масштабування системи.
- Здатність до тестування. Добре протестований код не тільки буде мінімізувати кількість помилок, а буде показником добре сформованої системи, оскільки це означитиме, що наявна низька зв'язність. Існують навіть окремі методології по створенню ПЗ на основі тестів — розробка через тестування.
- Повторне використання. Систему бажано проектувати так, щоб її деякі складові можна було використовувати в інших ІС.

Основними показниками неправильно сформованої архітектури є:

- Жорсткість. Дану програму тяжко модифікувати, при зміні одного компоненту, змінюються інші частини системи.
- Крихкість. При внесенні чи зміні нових елементів, інші частини системи виходять з ладу.
- Висока зв'язність. Створену програму важко протестувати, оскільки всі компоненти сильно зв'язані між собою.

Для створення програмної архітектури слід дотримуватися базових концепцій декомпозиції та програмного проектування.

1.2 Багатошарова архітектура та SOA

Існують два основних підходи до створення складних інформаційних систем: монолітна багатошарова архітектура (як правило, тришарова) та розподілена сервіс-орієнтована архітектура. Кожен з наведених підходів має свої як переваги так і недоліки.

Концепція багатошарової моделі є давно відомою та найпопулярнішою на даний час, вона базується на розподілі всієї системи на окремі ключові функціональні частини. Розглянемо класичну тришарову архітектуру, як найпопулярніший приклад багатошарової архітектури, вона включає шар доступу до даних, шар бізнес правил та шар представлення. На рисунку 1.1 зображено основні частини тришарової архітектури.

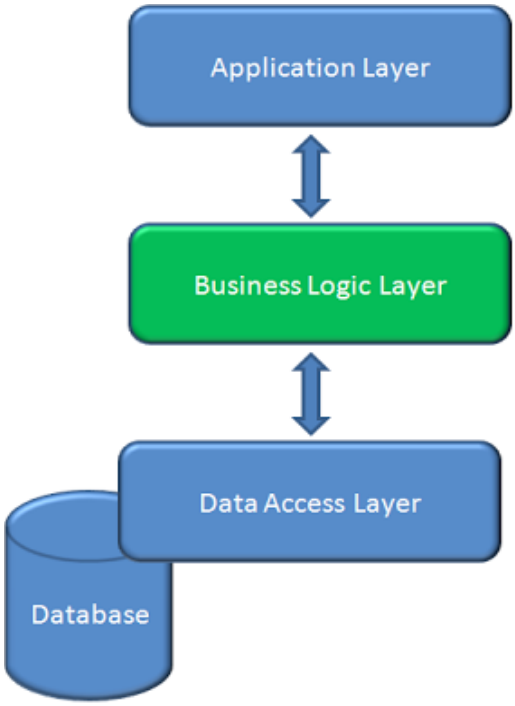

Рисунок 1.1 — Приклад класичної тришарової архітектури

Опишемо основні функції кожного зі трьох модулів. В основі всієї системи лежить шар доступу до даних, який надає зручний інтерфейс для доступу до джерел даних для вищих прикладних рівнів програми.

Наступним модулем є шар бізнес-логіки. Даний рівень містить набір алгоритмів та бізнес-функцій, які задають та описують предметну область, даний шар має бути найбільш гнучким, тому що саме до нього в майбутньому буде застосовуватися найбільше змін. Також використовує API рівня доступу до даних, щоб реалізовувати прикладні методи.

Найвище місце займає рівень представлення, який взаємодіє безпосередньо зі користувачем та надає вхідні дані для всієї системи.

Іншим підходом по створення високонавантажених систем є сервіс-орієнтована архітектура (Service Oriented Architecture, SOA). Вона представляє ідею, що програма має складатися з набору сервісів, які взаємодіють один з одним через стандартизовані протоколи та інтерфейси. В свою чергу до сервісів висуваються наступні вимоги [3]:

1. Стандартизовані інтерфейси.
2. Слабка зв’язність.
3. Абстракція
4. Перевикористання
5. Автономність
6. Відсутність стану

Сервіс можна уявляти як окремий функціональний модуль, який може знаходитися на іншому віддаленому комп’ютері, взаємодія між даними модулями відбувається через мережу. В найпростішому випадку існує три основних елементи: постачальник сервісу, споживач сервісу та реєстр сервісів. Схема взаємодії наведена на рисунку 1.2

Взаємодія між даними елементами виглядає наступним чином: постачальник сервісу реєструє свої сервіси, а споживач звертається до реєстру із запитом. Спілкування відбувається за певним уніфікованим протоколом передачі даних(SOAP, XML).

Рисунок 1.2 – Базова схема SOA

Інтерфейси – ключові компоненти SOA. Вони мають бути нейтральними до специфіки реалізації сервісу, які визначаються апаратною платформою, операційною системою чи мовою програмування [4].

Таблиця 1.1 – Основні переваги трирівневої архітектури та SOA

Трирівнева архітектура	**SOA**
Простота	Часткове розгортання
Узгодженість	Відмовостійкість
Міжмодульний рефакторинг	Відсутність стану
	Гетерогенність

Розглянемо переваги кожної з вище описаних архітектурних стилів, які представлені в таблиці 1.1

Для монолітної трирівневої архітектури:

1. Простота. Даний архітектурний підхід є простішим в реалізації, управлінні та розгортанні.
2. Узгодженність. Для монолітного додатку простіше слідкувати за

узгодженністю коду, опрацьовувати програмні помилки.

3. Міжмодульний рефакторинг. Єдиний кодовий репозиторій та цілісність структури полегшує роботу в ситуаціях, коли декілька модулів повинні взаємодіяти між собою або у випадку, коли необхідно перемістити деяку програмну логіку з одного модуля в інший. У сервіс-орієнтованому випадку, ми чітко обмежені границями модулів.

Таким чином, сервіс-орієнтована архітектура володіє необхідною гнучкістю, яку вимагають бізнес-процеси.

Перейдемо до основних переваг SOA:

1. Часткове розгортання. Розгортання всієї системи складається з розгортання окремих сервісів-модулів, які є незалежними. У випадку монолітного додатку, при деякій зміні – нам необхідно перезбирати та знову розгортати всю систему.
2. Відмовостійкість. При виходу зі строю окремих сервісів, вся система може залишатися дієздатною за рахунок резервних модулів.
3. Відсутність стану. SOA гарантує відсутність спільних станів між модулями.
4. Гетерогенність. Незалежність та стандартизованість інтерфейсів сервісів та протоколів передачі даних дозволяють створювати сервіси використовуючи різні мови програмування та різні фреймворки.

1.3 Мікросервісний підхід

Ідеї SOA розвинув, так званий, мікросервісний підхід. В цілому, SOA представляє гарні рішення, але не існує чітких правил та настанов як саме правильно досягти успіху в сервіс-орієнтованій архітектурі.

Найбільш вузьке місце в SOA – це складнощі пов’язані з протоколами обміну даних(таких як SOAP), відсутність методик, які дозволяють оцінити та встановити ступінь деталізації сервісів, а також місця розділу системи на сервіси.

Архітектурний стиль мікросервісів (MicroServices Architecture, MSA) – це підхід, при якому вся система являє собою набір невеликих сервісів, кожен з яких

розгортається та працює в окремому процесі та взаємодіє з іншими використовуючи легкі механізми, як правило HTTP. Описані сервіси побудовані навколо бізнес потреб та реалізують свій окремий обмежений предметний контекст з визначеними границями. Всі сервіси можуть бути розроблені на різних мовах програмуванням, використовуючи будь-які фреймворки та засоби зберігання даних.

Мікросервісну архітектуру вважають підмножиною SOA, як зображено на рисунку 1.3, тому що вона, аналогічно SOA, орієнтована на сервіси та розподілена, використовує уніфіковані протоколи обміну даними. MSA слід розглядати як підхід до реалізації SOA.

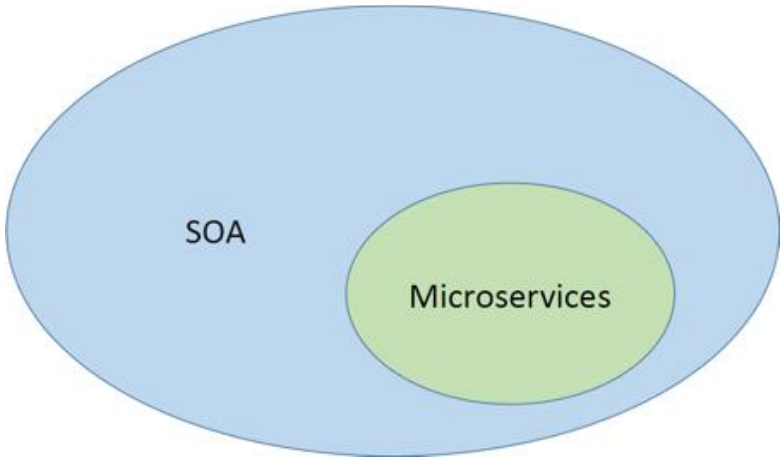

Рисунок 1.3 – Взаємовідношення MSA та SOA

Сам стиль не є технічною інновацією, він перейняв основні принципи проектування, які були використані в Unix.

Розглянемо ключові положення мікросервісної архітектури. Головним моментом є те, що кожен сервіс має бути невеликим та чітко сфокусованим на своїй задачі в межах певного контексту. Трактувати поняття «невеликий» можна різним чином.

Найпростішим та першим, що спадає на думку, — оцінювати складність кількість рядків коду, але даний підхід є неуніверсальним, оскільки різні мови програмування мають різну виразність. Також деякі сервіси можуть мати велику

кодову базу, оскільки вони описують явища, які є складними за визначенням, тому і вимагають велику кількість рядків коду. Іншою оцінкою складності є те, що над одним сервісом працює одна команда розробників із 5-6 чоловік та кожен із них може зрозуміти весь сервіс, в іншому випадку сервіс слід розділити на менші складові.

Іншим поняттям, яке характеризує мікросервіси є «сфокусований», тобто сервіс вирішує одну обмежену бізнес-задачу і не більше того. Даний принцип є іншим формулюванням базового принципу єдиної відповідальності (Single Responsibility Principle), якого слід дотримуватися при створенні будь-якого програмного забезпечення [5].

Також кожен мікросервіс має бути слабо зв'язаним та сильно зчепленим. Слабке зв'язування передбачає використання інтерфейсів та інструментів впровадження залежностей (Dependency Injection, DI), які дозволяють вносити зміни в один модуль не змінюючи інший. «Сильно зчепленний» означає, що компонент має всі необхідні методи рішення поставленої задачі.

Ідеологія мікросервісів закликає використовувати розумні приймачі та прості канали передачі, замість складних протоколів, типу WS-* або BPEL, слід використовувати неперевантажені протоколи.

Важливим аспектом будь-якої інформаційної системи є організація та управління даними. При мікросервісному підході практикується децентралізоване управління даними, приклад якого представлено на рисунку 1.4. Для стандартного монолітного додатку існує лише одна база даних, яка обслуговує безліч різних компонентів бізнес-логіки системи.

Для мікросервісної архітектури, коли кожен компонент бізнес-логіки являє собою мікросервіс, всі компоненти мають свої власні бази даних, які недоступні іншим мікросервісам. Дані елемента доступні для читання чи запису тільки через відповідний прикладний інтерфейс [6]. Також такий підхід дозволяє використовувати будь-яку кількість різних технологій збереження даних, тобто можливо в одному проекті використовувати реляційні бази даних, графові бази даних, нереляційні бази даних.

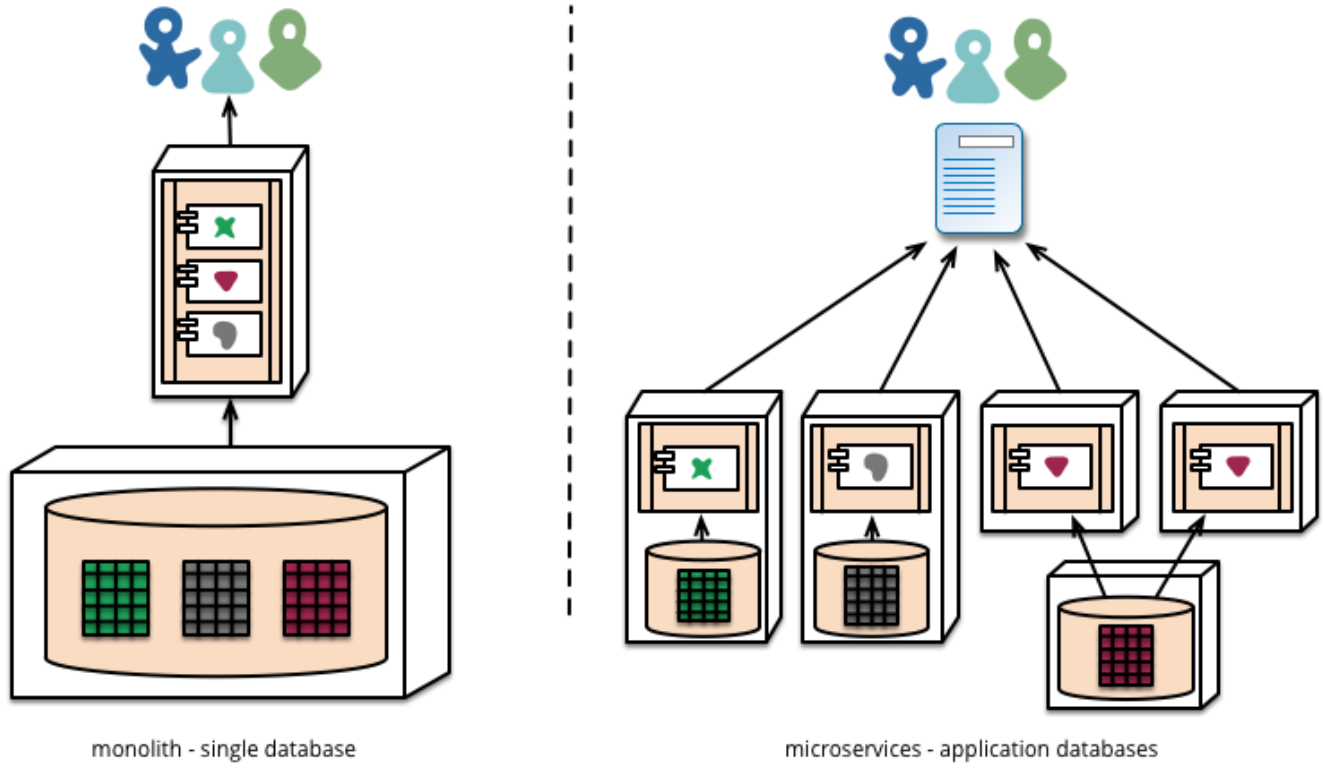

Рисунок 1.4 – Порівняння підходів до управління даними [6]

Даний підхід до управління даними має назву Polyglot Persistance. Децентралізація відповідальності за дані серед мікросервісів впливає на те, як ці дані змінюються. Загальний підхід до зміни даних полягає у використанні транзакцій для забезпечення консистентності при зміні даних. Даний підхід характерний для монолітних систем, оскільки забезпечує наступні важливі фактори: узгодженність даних, атомарність та ізольованість, тривалість.

Для мікросервісної архітектури реалізація транзакцій є нетривіальним завданням і може вирішуватися дизайном основаним на подіях. В цьому випадку мікросервіс генерує подію на певну зміну стану, а інші мікросервіси, які підписані на цю подію, певним чином оновлюють власні дані, які можуть спричинити створення нової події. Для подієвого управління даними характерна eventual consistency – узгодженність в кінцевому рахунку, тобто ми знаємо, що протягом деякого короткого періоду часу дані можуть бути не узгодженні. Також доцільно використовувати події для реалізації бізнес-задач, які покривають декілька мікросервісів. Дизайн оснований на подіях має свої переваги та недоліки, він

дозволяє реалізувати розподілені транзакції, які все ж таки є в кінцевому результаті узгодженими, але дана модель має вищу складність та система може мати справу з деякими суперечливими даними, окрім цього сервіси повинні виявляти та ігнорувати повторювані події. Іншим важливим питанням при організації управлінням даними за вище описаним дизайном є створення запитів, які витягують дані з декількох мікросервісів.

1.3.1 Переваги мікросервісів

Мікросервісна архітектура має багато різноманітних переваг. Деякі з них властиві будь-яким розподіленим системам. Головною ціллю даного архітектурного стилю, як і будь-якої архітектури, це управління складністю.

Мікросервісна архітектура вирішує проблему складності для моноліту. Переваги мікросервісів можна розглядати з двох різних аспектів: платформного та програмного. До платформних переваг можна віднести:

- Масштабованість. Кожен сервіс можна незалежно масштабувати на стільки на скільки потрібно. На противагу, в громіздких монолітних системах розширювати потрібно все одночасно, якщо навіть одна невелика частина системи може мати обмежену продуктивність.
- Незалежне розгортання. При використанні мікросервісів можна вносити зміни в окремий модуль та розгортати його незалежно від всіх інших компонентів. Це значно пришвидшує процес розгортання. Якщо певна проблема була зафіксована, то вона відноситься до конкретного сервісу, який легко можна ізолювати та перезібрати або відкинути нові зміни.
- Гетерогенність технологій. Для реалізації мікросервісів можна обирати будь-яку технологію, яка пасує для поставленої задачі. Якщо для деякої частини критично важлива продуктивність, слід обирати відповідний інструментарій. Це саме стосується і сховищ для даних: в одній системі, наприклад, можна використовувати одночасно реляційну базу даних і NoSQL, чи будь-яку довільну базу даних.

- Стійкість системи. При виходженні з ладу системи, ми можемо локалізувати причину в рамках конкретного мікросервісу. Тому ми можемо не перезавантажувати всю систему, а лише відновити роботу зламаного компоненту.

Зі сторони платформних переваг можна виділити наступні [2]:

- Модульність. Мікросервісна архітектура дозволяє зберігати модульність та інкапсуляцію, вони забезпечують логічний розподіл системи на модулі за рахунок явного фізичного розділу по серверам. Фізична ізольованість захищає від порушення границь обмежених контекстів.

Також зауважимо, що мікросервіси є легші для розуміння та підтримки, не потрібно вникати одразу в усі подробиці загальної системи. Крім того, можна обирати для кожного сервісу необхідне апаратне забезпечення.

Час запуску та впровадження є значно швидшим, ніж для стандартного тришарового додатку.

MSA надає можливість безперервного розгортання. Також цей підхід надає можливість кожному сервісу масштабуватися незалежним чином. Можливо розгортати таку кількість екземплярів сервісу, яка задовольнить потребу бізнес-потреб. 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.

1.3.2 Недоліки мікросервісів

Мікросервісна архітектура не призначена для розв’язку всіх можливих задач та має властиві розподіленим системам недоліки. Найбільше труднощів виникає у питаннях взаємодії мікросервісів, їх інтеграції. Також даний термін з’явився відносно недавно, тому не існує загальноприйнятих специфікацій та рекомендацій для створення якісних додатків.

Побудова мікросервісів є не тривіальною задачею, як і побудова будь-якої ефективної розподіленої системи, з цього випливають проблеми пов’язані з

впровадженням взаємодії між процесами на основі обміну повідомлення або викликом певного методу через RPC, при цьому треба розглянути безліч питань, які стосуються обробки мережевих збоїв та клієнтських помилок.

Наведемо перелік основних складнощів з якими можна зіткнутися при створенні мікросервісного додатку:

- Складність розробки. Оскільки ми маємо справу з розподіленою системою, то слід багато уваги виділяти опрацюванню запитів та їх маршрутизації між сервісами. Ситуація може також погіршуватися, коли віддалені виклики відпрацьовують із певними затримками.
- Управління даними. Організація транзакцій є досить складною задачею для розподілених систем, в той час коли для моноліту створення транзакцій є тривіальною задачею, оскільки існує лише єдина база даних. Частим є випадок, коли бізнес-операції оновлюють кілька суб'єктів, у випадку мікросервісної архітектури необхідно оновлювати кілька баз даних, що належать різним мікросервісам. Використання розподілених транзакцій, як правило, не підходить. Вони просто не підтримуються багатьма з сучасних високомасштабованих NoSQL баз даних та брокерів обміну повідомленнями. В решті решт, доводиться використовувати логічно-базований підхід, який є більш складним для розуміння та реалізації.
- Збільшення використання ресурсів. Мікросервісна архітектура вимагає більше ресурсів ніж монолітна, оскільки кожен мікросервіс необхідно забезпечити власним контейнером з розгорнутим програмним середовищем.
- Збільшення навантаження на мережу. Для взаємодії мікросервіси використовують стандартні протоколи обміну мережею, коли компоненти моноліту спілкуються в рамках єдиного процесу і не вимагають додаткових мережевих викликів [4].
- Тестування системи. Інтеграційне та модульне тестування значно складніше, ніж у випадку монолітного додатку, але з появою нових програмних інструментів це питання може бути вирішене.

- Моніторинг системи. Вартість моніторингу значно вища, але, як і у випадку із тестуванням, ці проблеми можуть бути вирішені новими ефективними програмними інструментами.

Також до недоліків можна віднести складність рефакторингу, особливо якщо рефакторинг вимагає перенесення деякої логіки між сервісами.

Вибір мікросервісної архітектури без попереднього аналізу може привести до безладного проектування та невдачі всього проекту.

1.4 Висновки

Для вирішення задач проектування існує безліч ефективних підходів і дуже важливо максимально уважно та ретельно підійти до вибору архітектури системи, оскільки це визначить подальші перспективи всього проекту. Побудова складних нетривіальних програмних системи є задачею не з простих. Для її вирішення можна застосовувати як монолітний підхід, так і розподілений дивлячись із контексту та потреб бізнесу.

Застосування монолітної архітектури має сенс тільки при створенні відносно невеликих та легких систем, якщо ж все таки 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 ПОБУДОВА МІКРОСЕРВІСНИХ СИСТЕМ

2.1 Інтеграція мікросервісів

Правильна інтеграція є найважливішим технічним аспектом під час проектування та реалізації мікросервісів. При належному виконанні мікросервіси збережуть свою автономію, і в той час можна буде вносити в них зміни і випускати їх нові версії незалежно від решти системи. При неправильному виконанні вас чекають серйозні неприємності. Нижче будуть наведені основні проблеми та підходи до їх вирішення.

Перш за все розглянемо основні шаблони для створення MSA.

2.1.1 Шаблони проектування мікросервісів

1. Шаблон «Агрегатор»

Перший та, мабуть, найбільш поширений шаблон проектування при створенні мікросервісів – «агрегатор».

У найпростішому випадку, агрегатор є звичайною веб-сторінкою, яка викликає безліч сервісів для реалізації функціональності, необхідної додатком. Схема патерну наведена на рисунку 2.1

Оскільки всі сервіси (Service A, Service B і Service C) надаються за допомогою легкого REST-механізму, веб-сторінка може отримати дані й опрацювати їх як потрібно. Якщо необхідне будь-яке додаткове опрацювання, наприклад, застосувати бізнес-логіки до даних, отриманих від окремих сервісів, то для цього у вас може бути CDI-компонент, що перетворює дані таким чином, щоб їх можна було показати на веб-сторінці. Агрегатор може використовуватися і в тих випадках, коли не потрібно нічого відображати, а потрібен лише більш високорівневий мікросервіс, який може використовувати інші сервіси.

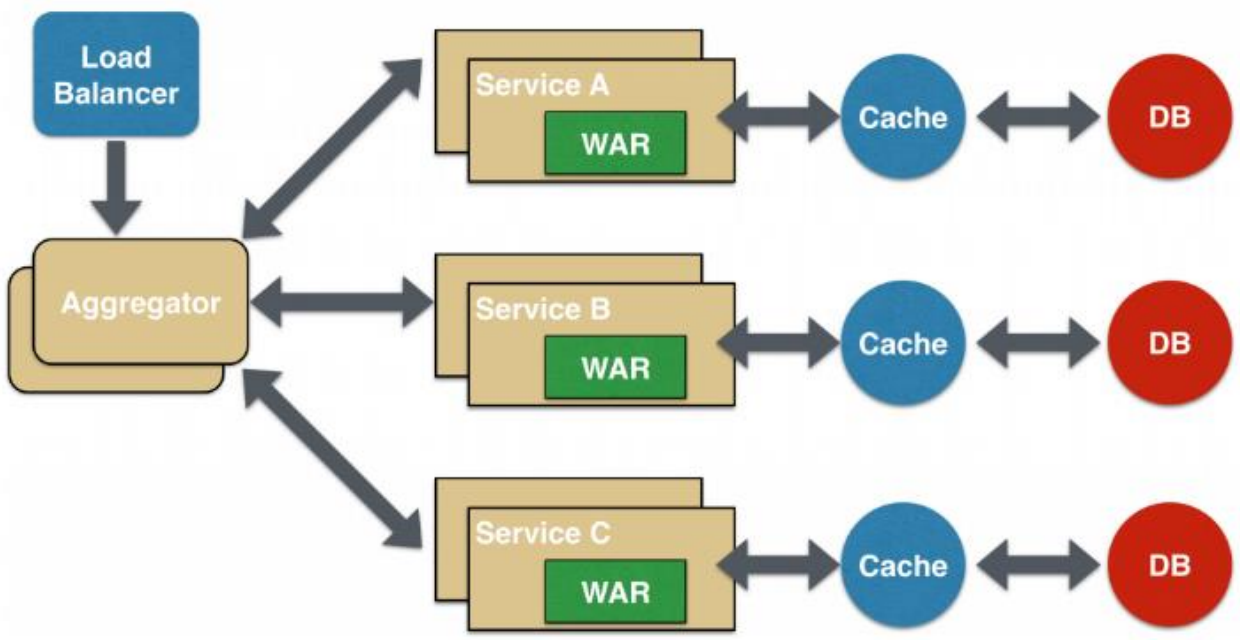

Рисунок 2.1 – Схема шаблону «Агрегатор»

Цей патерн дотримується принципу DRY. Якщо існує безліч сервісів, які повинні звертатися до сервісів A, B і C, то рекомендується абстрагувати цю логіку в мікросервіс і агрегувати її у вигляді окремого сервісу. Перевага абстрагування на цьому рівні полягає в тому, що окремі сервіси, скажімо, A, B і C, можуть розвиватися незалежно, а бізнес-логіку буде як і раніше виконувати композитний мікросервіс.

2. Шаблон «Посередник»

Патерн «посередник» при роботі з мікросервісами – це окремий варіант агрегатора. В такому випадку агрегація повинна відбуватися на клієнті, але в залежності від бізнес-вимог при цьому може викликатися додатковий мікросервіс.

На рисунку 2.2 наведена схема даного шаблону, як і агрегатор, посередник може незалежно масштабуватися по горизонталі і по вертикалі. Це може знадобитися в ситуації, коли кожен окремий сервіс потрібно не надавати споживачеві, а запускати через інтерфейс.

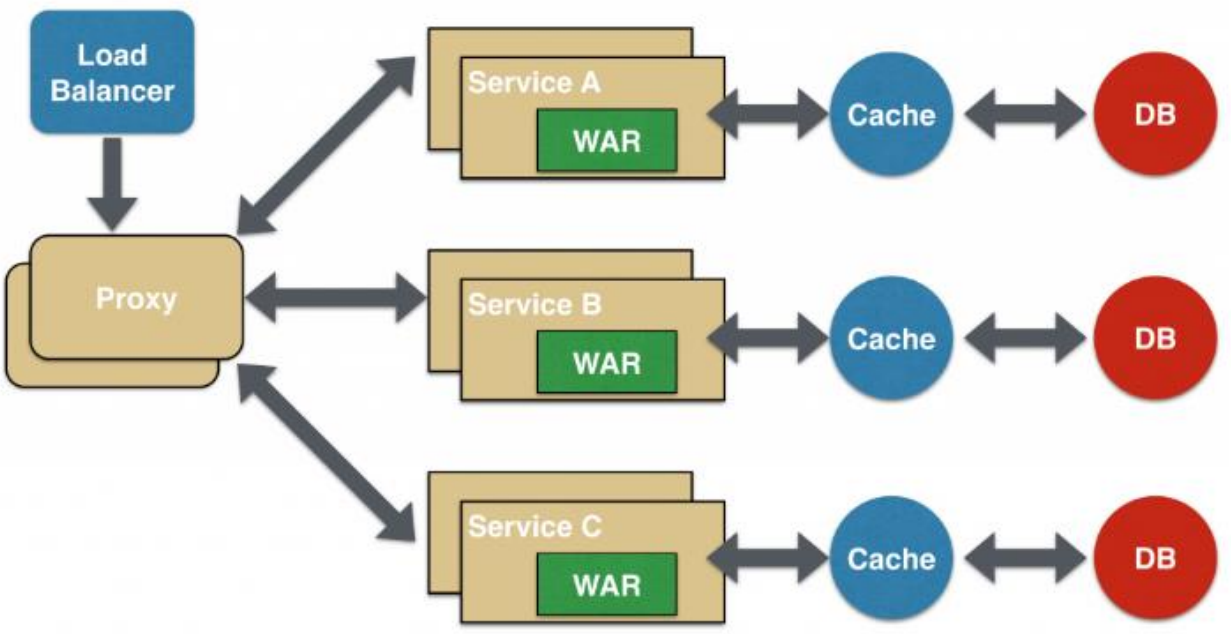

Рисунок 2.2 – Схема шаблону «Посередник»

Посередник може бути формальним, в такому випадку він просто делегує запит одному з сервісів. Він може бути і розумним, в такому випадку дані перед відправкою клієнту піддаються тим чи іншим перетворенням. Наприклад, рівень представлення для різних пристроїв може бути інкапсульований в розумного посередника.

3. Шаблон «Ланцюг»

Мікросервісний патерн проектування «Ланцюг» видає єдину консолідовану відповідь на запит. В даному випадку сервіс А отримує запит від клієнта, зв'язується з сервісом В, який, в свою чергу, може зв'язатися з сервісом С.

Архітектура побудови мікросервісів за моделлю «Ланцюг» наведена на рисунку 2.3. Всі ці сервіси, як правило, обмінюються синхронними повідомленнями «запит / відповідь» по протоколу HTTP. Найважливішим моментом є те, що клієнт блокується до тих пір, поки не виконається вся комунікаційна послідовність запитів і відповідей, тобто Service A - Service B і Service B - Service C. Запит від Service B до Service C може виглядати зовсім інакше, ніж від Service A до Service B. Це найбільш важливо у всіх випадках, коли бізнес-цінність декількох сервісів додається.

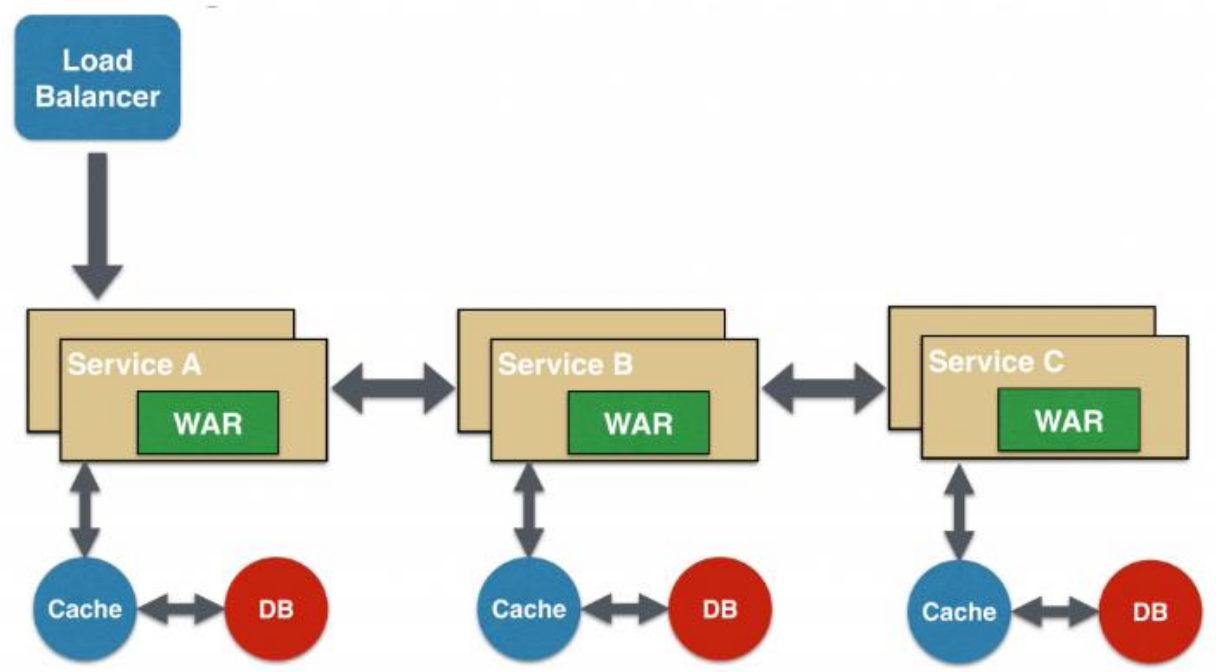

Рисунок 2.3 – Схема шаблону «Ланцюг»

Також важливо зрозуміти, що не можна робити ланцюг занадто довгим. Це критично, оскільки ланцюг синхронний за своєю природою, і чим він довший, тим довше доведеться чекати клієнтові, особливо якщо відгук полягає у виведенні веб-сторінки на екран. Існують способи обійти такий блокуючий механізм запитів і відгуків, і вони розглядаються в наступному шаблоні.

4. Шаблон «Гілка»

Мікросервісний шаблон проектування «Гілка» розширює шаблон «Агрегатор» і забезпечує одночасну обробку відповідей від двох ланцюгів мікросервісів, які можуть бути взаємовиключними. Цей патерн також може застосовуватися для виклику різних ланцюгів, або одного і того ж ланцюга - в залежності від потреб. Приклад взаємодії сервісів наведено на рисунку 2.4.

В іншому випадку сервіс А може викликати лише один ланцюг в залежності від того, який запит отримає від клієнта. Такий механізм можна конфігурувати, реалізувавши маршрутизацію кінцевих точок JAX-RS, в такому випадку конфігурація повинна бути динамічною.

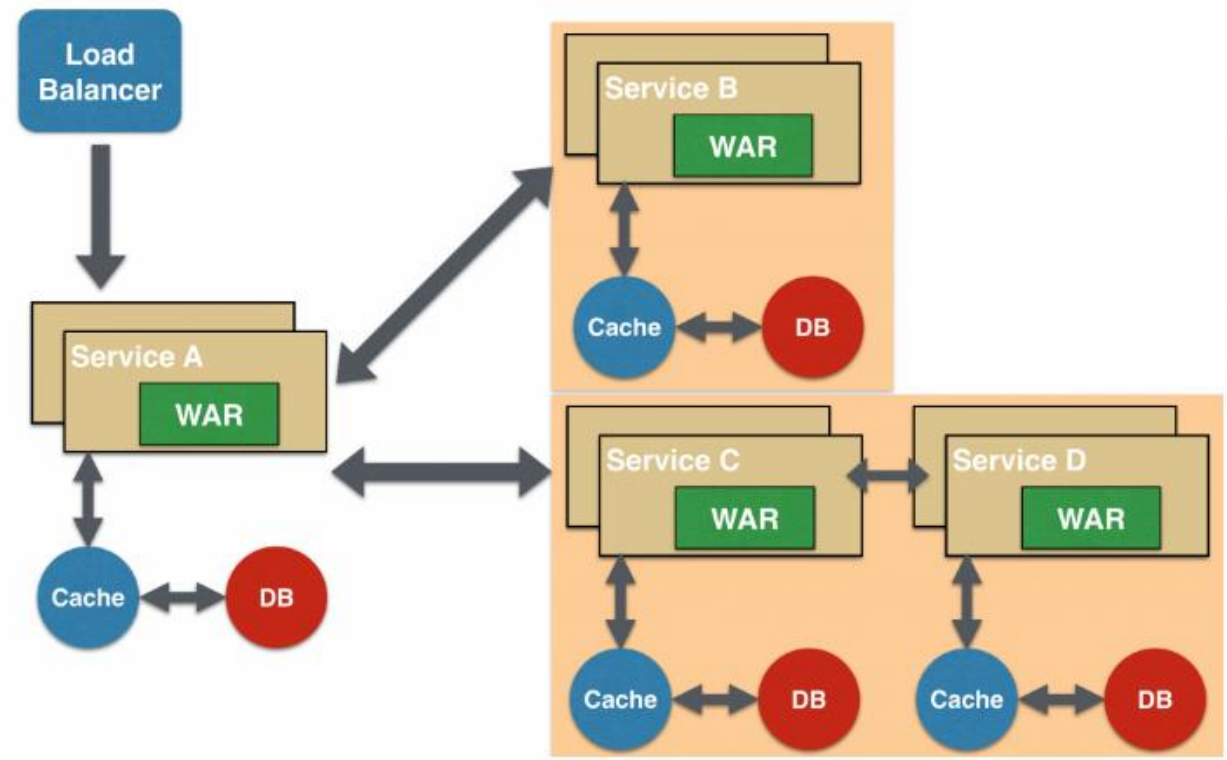

Рисунок 2.4 – Схема шаблону «Гілка»

5. Шаблон «Дані спільного використання»

Один з принципів проектування мікросервісів — автономність. Це означає, що сервіс повностековий і контролює всі компоненти: інтерфейс призначений для користувача, проміжне ПЗ, транзакції. В такому випадку сервіс може бути багатомовним і вирішувати кожну задачу за допомогою найбільш відповідних інструментів. Наприклад, якщо при необхідності можна застосувати сховище даних NoSQL, то краще зробити саме так, а не додавати всю цю інформацію в базу даних SQL. Однак, типова проблема, особливо при рефакторингу наявного монолітного додатку, пов'язана з нормалізацією бази даних — так, щоб у кожного мікросервісу був строго визначений обсяг інформації. На рисунку 2.5 продемонстровано базову схему даного патерну. За цим паттерном кілька мікросервісів можуть працювати по ланцюгу і спільно використовувати сховища кеша і бази даних. Це доцільно лише в разі, якщо між двома сервісами існує сильний зв'язок. Деякі можуть вбачати в цьому антипаттерн, але в деяких бізнес-ситуаціях такий шаблон дійсно доречний.

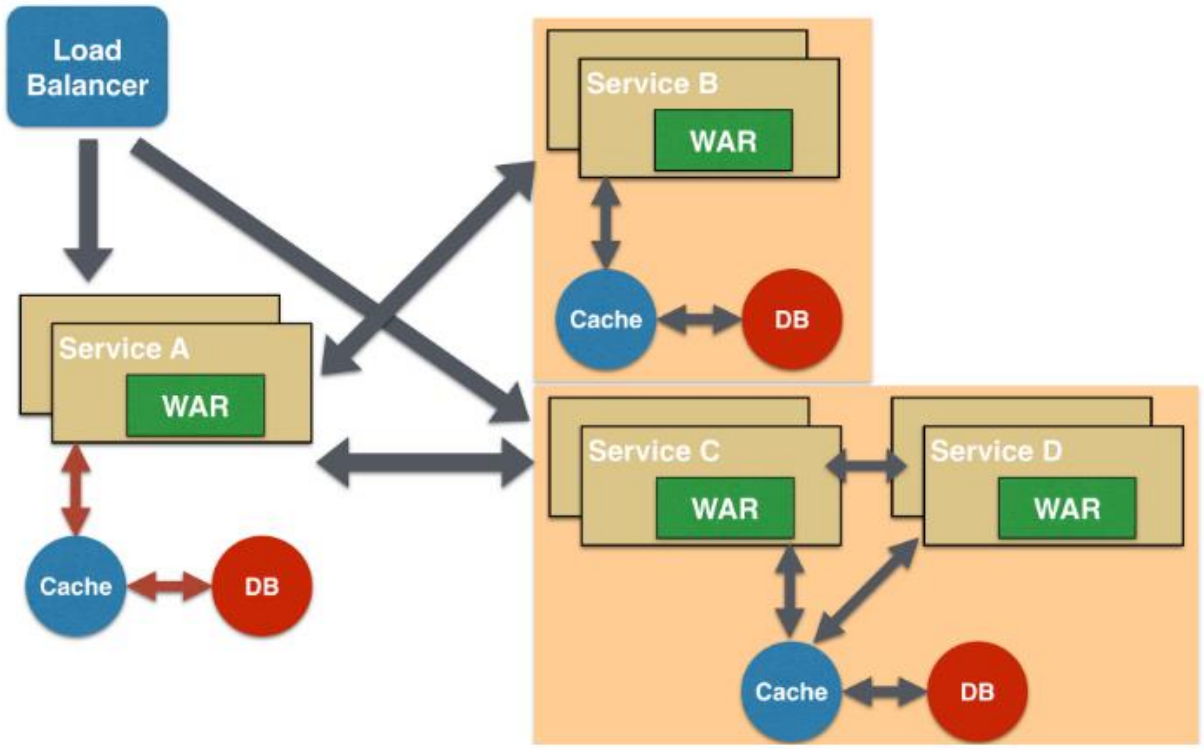

Рисунок 2.5 – Схема шаблону «Дані спільного використання»

Крім того, його можна розглядати як проміжний етап, який потрібно подолати, поки мікросервіси не стануть повністю автономними.

6. Шаблон «Асинхронні повідомлення»

При всій поширеності і зрозумілості підходу REST, у нього є важливе обмеження, а саме: він синхронний і, отже, блокуючий. Забезпечити асинхронність можна, але це робиться по-своєму в кожному додатку. Тому в деяких мікросервісних архітектурах можуть використовуватися черги повідомлень, а не модель REST - запит / відповідь.

На рисунку 2.6 описано як сервіс A може синхронно викликати сервіс C, який потім буде асинхронно зв'язуватися з сервісами B і D за допомогою черги повідомлень. Комунікація між сервісами A та C може бути асинхронною, скажімо, з використанням веб-сокетів; так досягається бажана масштабованість.

Комбінація моделі REST(запит/відповідь) та обміну повідомленнями (видавник / підписник) також можуть використовуватися для досягнення поставлених цілей.

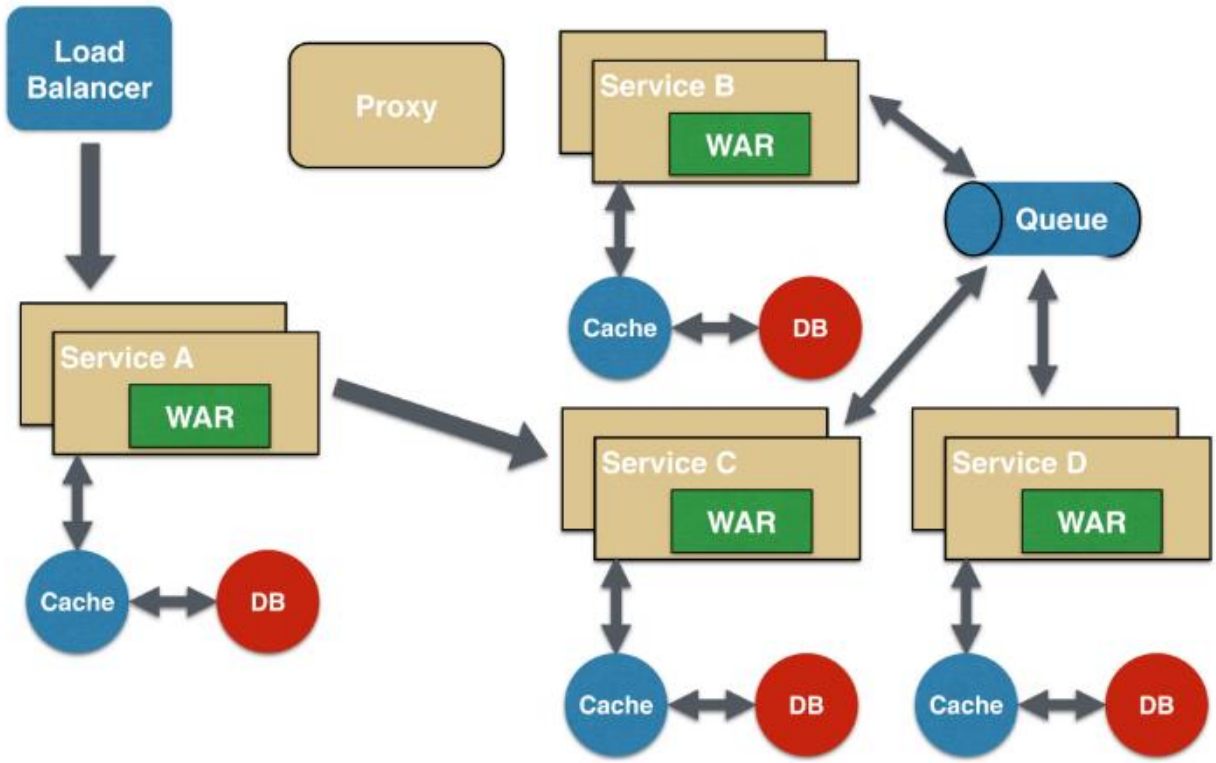

Рисунок 2.6 – Схема шаблону «Асинхронні повідомлення»

2.1.2 Типи комунікації мікросервісів

Існує два основних підходи до організації комунікації мікросервісів, які лежать в основі всіх шаблонів наведених вище: синхронний (REST, RPC) та асинхронний обмін повідомленнями.

Розглянемо синхронний підхід на прикладі REST. Передача репрезентативного стану являє собою архітектурний стиль.

У REST-стилю є безліч принципів і обмежень, але ми сконцентруємо увагу на тих з них, які дійсно допоможуть нам при зустрічі з інтеграційними складнощами при побудові мікросервісів та пошуку стилів інтерфейсів для наших сервісів, які виступають в якості альтернативи RPC. Найбільш важливим є поняття ресурсів. Під ресурсами можна розуміти те, про що знає сам сервіс, наприклад сутність «Користувач». У запиті сервер створює різні образи (або представлення) цього об'єкта. Клієнт, наприклад, може запросити JSON-репрезентацію об'єкта «Користувач», навіть якщо він збережений абсолютно в іншому форматі.

Отримавши представлення цього об'єкта, він може робити запити на його зміну і сервер може їх виконувати або не виконувати.

Деякі з властивостей, що надаються протоколом HTTP в якості частини своєї специфікації, спрощують реалізацію REST по HTTP, тоді як при використанні інших протоколів доводиться справлятися з реалізацією подібних властивостей самостійно. У самому протоколі HTTP визначається ряд дуже корисних можливостей, які дуже добре працюють на реалізацію REST-стилю. Наприклад, в HTTP-специфікації методи, такі як GET, POST і PUT, мають цілком зрозумілий сенс, який визначає характер їх роботи з ресурсами. Фактично архітектурний стиль REST підказує нам, що ці методи будуть вести себе так само і по відношенню до всіх ресурсів, і виходить, що HTTP-специфікація вже визначила той набір методів, якими ми можемо скористатися. GET витягує ресурс ідемпотентним способом, а POST створює новий ресурс.

Перейдемо до асинхронного обміну даними на основі подій. Розглянемо два основні моменти: спосіб надання мікросервісами подій і спосіб визначення споживачами моменту настання тієї або іншої події. Традиційно такі брокери повідомлень, як RabbitMQ, намагаються охопити відразу обидві задачі.

Постачальники використовують API для публікації події брокеру. Брокер опрацьовує підписки, дозволяючи споживачам отримати інформацію при настанні тієї чи іншої події. Такі брокери можуть навіть обробляти стан споживачів, наприклад сприяючи відстеженню того, які повідомлення вони бачили раніше.

Будь-яка кількість постачальників може відправляти повідомлення в одну чергу, також будь-яку кількість підписників може отримувати повідомлення з однієї черги. Підписник - програма, яка приймає повідомлення. Зазвичай, підписник знаходиться в стані очікування повідомлень.

Ці системи, як правило, розробляються з можливостями масштабування. Можливо, доведеться поплатитися ускладненням процесу розгортання, оскільки для розробки і тестування сервісів може знадобитися запуск ще однієї системи. Для збереження працездатності цієї інфраструктури можуть також знадобитися додаткові машини і наявність певного досвіду. Але якщо вийде справитися з усіма

труднощами, це може стати дуже ефективним способом реалізації слабо пов'язаних архітектур, керованих подіями.

2.2 Розбиття моноліту на частини

Майже всі успішні приклади використання мікросервісної архітектури, розпочиналися з моноліту, який з плином часу розростався. І доволі частими є випадки, коли проект розпочинався з мікросервісів і не досягав своєї мети.

Тому можна зробити висновок, що краще розпочинати новий проект з моноліту, навіть якщо ви знаєте, що ваш проект буде достатньо великим для використання мікросервісів. MSA є корисною та ефективною архітектурою, але всі її переваги доцільні лише для великих та складних систем. Для простих систем набагато краще підходить суцільна монолітна архітектура.

Перша причина дотримуватися принципу «спочатку - моноліт» — класичний принцип «Вам це не знадобиться». Коли ви починаєте розробляти новий додаток, необхідно впевнитись, що він буде корисним для користувачів. Кращий спосіб перевірити, чи буде додаток користуватися попитом — це створення його спрощеної версії. Спочатку на першому місці стоїть швидкість розробки (а значить, і швидкість отримання зворотного зв'язку від потенційних користувачів), а розробка мікросервісів займає набагато більше часу.

Наступна проблема полягає в тому, що мікросервіси працюють добре, якщо ви досягли чітких, стабільних меж між окремими сервісами — для цього потрібно отримати правильний набір обмежених контекстів. Будь-який рефакторинг функціональності між сервісами складніший аналогічного в моноліті.

Побудувавши монолітний додаток, ви зможете визначити вірні межі, перш ніж використовувати мікросервіси. Це також дасть вам час підготувати все необхідне для створення сервісів з більш чіткими обмеженими контекстами.
Існують різні шляхи реалізації стратегії «спочатку - моноліт».

- Логічний шлях – це проектувати моноліт з усією обережністю, звертаючи увагу на модульність в програмному забезпеченні, межі API та спосіб

зберігання даних. Якщо зробити це добре, то перехід до мікросервісів буде відносно простим.

- Більш загальний підхід – почати з моноліту і поступово відокремлювати від нього мікросервіси. У цьому випадку значна частина початкового моноліту може залишитися в якості центральної в мікросервісній архітектурі, але основна частина нової розробки буде відбуватися в сервісах, залишаючи моноліт без великих змін.
- Також поширений підхід з повною заміною моноліту. Даний підхід не є дуже ефективним, але він має право на існування.
- Ще один варіант розбиття моноліту – почати з декількох сервісів, які більші за тих, що очікуються в кінці. Використовуйте ці великі сервіси, щоб навчитися працювати в мультисервісному середовищі [7].

Хоча багато переваг надає підхід «спочатку - моноліт», далеко не всі розділяють такий метод. Контраргумент полягає в тому, що починаючи з мікросервісів, ви звикаєте до ритму розробки в такому оточенні.

Потрібно дуже багато зусиль, щоб побудувати монолітний додаток в модульному вигляді, достатньому для простого розбиття на мікросервіси. Починаючи з мікросервісів, ви працюєте в невеликих командах, розділених межами сервісів, це дозволить вам прискорити розробку за рахунок додаткових кадрових ресурсів, як тільки це буде потрібно. Такий підхід особливо добре працює в разі заміни системи, коли у вас є більше шансів отримати досить стабільні кордони на ранньому етапі.

2.3 Розгортання та масштабування мікросервісів

Монолітні системи розгортаються досить просто, у мікросервісному випадку все не так тривіально. Перш за все розглянемо питання безперервної інтеграції і безперервної доставки.

2.3.1 Основні підходи до розгортання мікросервісів

При використанні CI основною метою є підтримка загального синхронізованого стану, яка досягається шляхом перевірки того, що щойно введений в експлуатацію код інтегрується з існуючим кодом належним чином.

Для досягнення цієї мети CI-сервер визначає переданий код і здійснює ряд перевірок, переконуючись в тому, що код скомпільований та тести з ним проходять без збоїв. В якості частини даного процесу часто створюється артефакт (або артефакти), які використовуються для подальшої перевірки, наприклад розгортання працюючого сервісу з метою запуску для нього ряду тестів. Бажано створювати ці артефакти тільки один раз і використовувати їх для всіх розгортань тієї чи іншої версії коду.

Щоб допустити повторне використання цих артефактів, ми розміщуємо їх в особливе сховище, яке надається CI-інструментарієм, або знаходиться в окремій системі. Розмірковуючи про мікросервіси та безперервну інтеграцію, треба думати про те як засіб CI створює відображення на окремо взяті мікросервіси.
В найпростішому випадку, все розміщується в одному великому сховищі, де знаходиться весь код(рис. 2.7). Будь-яка перевірка цього вихідного коду запустить збірку, де ми запустимо всі етапи верифікації, пов'язані з мікросервісами, і створимо кілька артефактів, які мають зворотний зв'язок з тією ж самою збіркою.

Така модель може працювати досить успішно за умови, що ви дотримуєтесь задуму жорстко регламентованих випусків, де вас не турбує одночасне розгортання відразу декількох сервісів.

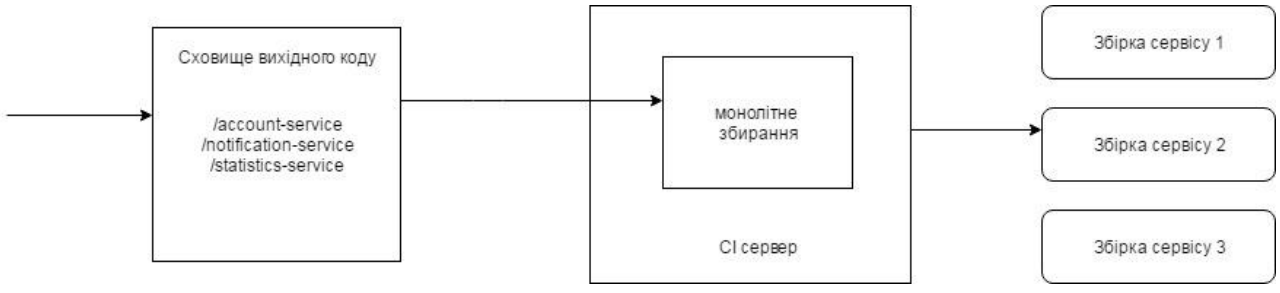

Рисунок 2.7 – Збирання за спільним сховищем коду

Взагалі, це шаблон, якого слід уникати, але на ранній стадії проекту, особливо якщо над проектом працює одна команда, його короткочасне використання має певний сенс.

У даної моделі є безліч недоліків. При внесенні навіть невеликих змін в будь-який сервіс, будуть зібрані та перевірені всі інші сервіси. На це може піти багато часу, оскільки необхідно буде чекати проходження тестів там, де це не є потрібним. Різновидом вище наведеного підходу є наявність єдиного дерева для всього вихідного коду і декількох CI-збірок, які відображаються на частини цього дерева. Приклад даного збирання проекту наведено на рисунку 2.8

Даний підхід є кращим ніж попередній, але все однак він передбачає перевірку всього вихідного коду, хоча збирання буде здійснюватися для кожного сервісу окремо.

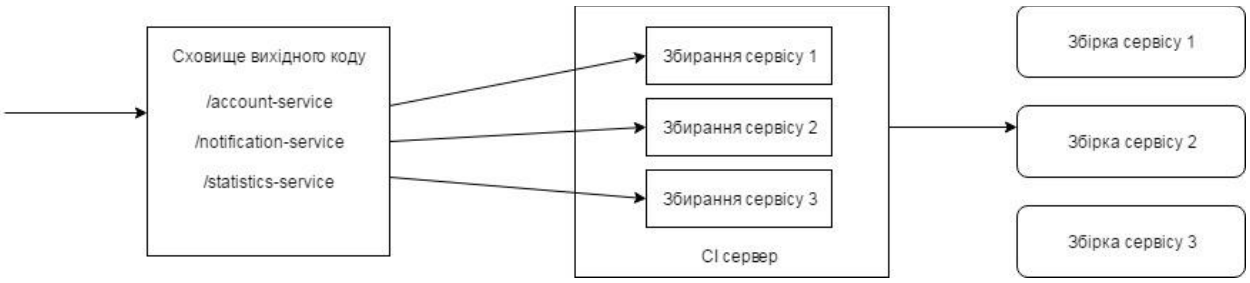

Рисунок 2.8 – Збирання на базі деревоподібної структури

Найбільш практичним є підхід зображений на рисунку 2.9, коли у кожного мікросервісу існує своє окреме сховище для вихідного коду, яке відображається на його власну CI-збірку. При внесенні змін запускається та тестується тільки необхідна збірка.

Але при цьому виникає додаткова складність при внесенні змін, які зачіпають відразу декілька сховищ.

Концепція конвеєрного збирання надає відмінний спосіб відстеження ходу обробки програми у міру прояснення ситуації на кожній стадії, допомагаючи з'ясувати якість програми. Ми здійснюємо збірку нашого артефакту, і цей артефакт використовується на всьому протязі конвеєра.

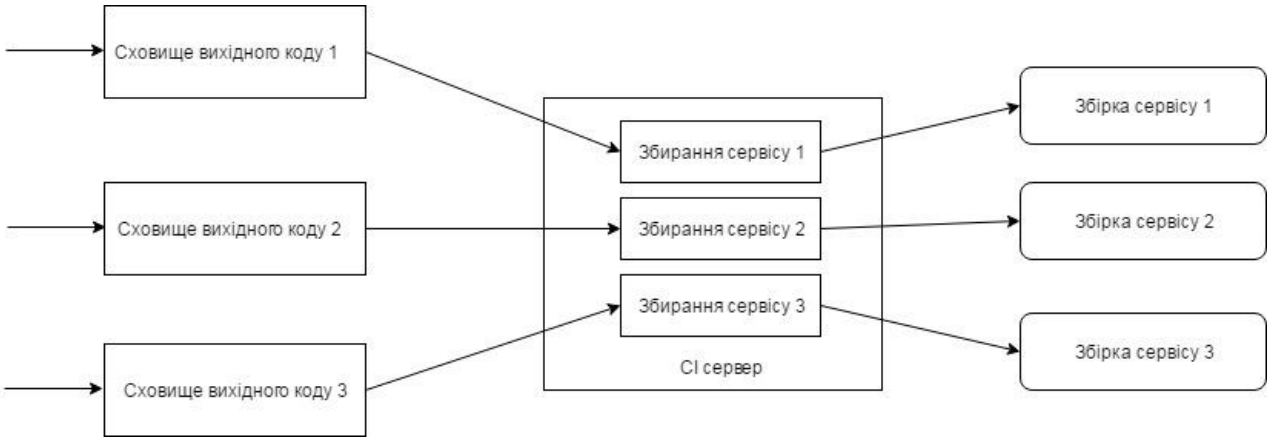

Рисунок 2.9 – Збирання при наявності репозиторію для кожного сервісу

2.3.2 Масштабування мікросервісів

Володіти системою здатною автоматично масштабуватись та відповідно реагувати на збільшення навантаження або відмову деяких вузлів є пріоритетною вимогою, але для деяких випадків це виявиться не актуальним і ресурсозатратним.

Коли розглядаються питання про необхідність і способи масштабування системи, що дозволяє краще впоратися з навантаженням або збоями, висуваються наступні вимоги:

- Час відгуку / затримки.
- Доступність.
- Збереження даних.

Важливою частиною створення відмовостійкої системи, особливо коли функціональні можливості розподіляються серед декількох мікросервісів, які можуть перебувати як в робочому, так і в неробочому стані, є забезпечення її спроможності безпечно знижувати рівень функціональності. При роботі з єдиним монолітним додатком нам не доводиться приймати безліч рішень. Працездатність системи залежить від роботи двійкового коду. Але при використанні архітектури мікросервісів потрібно розглядати набагато складніші ситуації. Чим більше один сервіс залежить від залучення інших сервісів, тим більше успішна робота одного сервісу впливає на виконання завдань іншими сервісами. Використання

технологій інтеграції, що дозволяють переводити нижчий сервер в режим автономної роботи, може знизити ймовірність впливу простоїв, як планових, так і позапланових збоїв на вищі сервіси.

При проведенні ідемпотентних операцій результат після першого застосування не змінюється, навіть якщо операція послідовно виконується ще кілька разів. Якщо операції є ідемпотентними, ми можемо повторювати виклик кілька разів без негативного впливу. Це нам дуже знадобиться, якщо необхідно повторно відтворити повідомлення, коли немає впевненості, що вони оброблені. Це є досить поширеним способом відновлення після помилок. В основному масштабування систем виконується з двох причин.

По-перше, для того, щоб легше було впоратися зі збоями: якщо ми переживаємо за відмову будь-якого компонента, то допомогти зможе наявність такого ж додаткового компонента.

По-друге, для підвищення продуктивності, що дозволяє або впоратися з більш високим навантаженням, або знизити час відгуку, або досягти обох результатів.

Розглянемо ряд найбільш поширених технологій масштабування, якими можна буде скористатися, і подумаємо про їх застосування до архітектури мікросервісів.

1. Нарощування потужностей

Від нарощування потужностей деякі операції можуть тільки виграти. Більш об'ємний корпус з більш швидким центральним процесором і більш ефективної підсистемою введення-виведення часто здатні зменшити затримки і підвищити пропускну здатність, дозволяючи виконувати більший обсяг робіт за менший час.

Але такий різновид масштабування, яку часто називають вертикальним масштабуванням, може бути занадто витратним: іноді один великий сервер може коштувати набагато більше, ніж два невеликих сервера нижчої потужності.

2. Розподіл робочих навантажень

Наявність єдиного мікросервіса на кожному хості, безумовно, краще моделі, яка передбачає наявність на хості відразу декількох мікросервісів. Але спочатку з

метою зниження вартості обладнання або спрощення управління хостом багато хто приймає рішення про співіснування кількох мікросервісів на одній фізичній машині. Оскільки мікросервіси запускаються в незалежних процесах, які обмінюються даними по мережі, завдання подальшого їх переміщення на власні хости з метою підвищення пропускної спроможності і масштабування не представляє особливої складності.

3. Балансування навантаження

Коли сервісу потрібна відмовостійкість, вам знадобляться способи обходу критичних місць збоїв. Для мікросервісу, який надає синхронну кінцеву точку по HTTP, найбільш простим способом вирішення цього завдання (рис. 2.10) буде використання декількох хостів із запущеними на них екземплярами мікросервісу, що знаходяться за балансувальником навантаження. Споживачі мікросервісу не знають, чи пов'язані вони з одним його екземпляром або з сотнею таких екземплярів.

Рисунок 2.10 – Схема розподілу навантаження

4. Системи на основі виконавців

Застосування балансувальника не є єдиним способом поділу навантаження серед кількох екземплярів сервісу та зменшення їх крихкості. Залежно від характеру операцій настільки ж ефективною може бути і система на основі виконавців. Дана модель також добре працює при пікових навантаженнях, де в міру зростання потреб можуть запускатися додаткові екземпляри для відповідності вхідного навантаження. Поки сама черга робіт буде зберігати стійкість, ця модель може використовувати масштабування для підвищення як пропускної здатності робіт, так і відмовостійкості, оскільки стає простіше впоратися з впливом відмовив (або відсутнього) виконавця. Робота займе більше часу, але нічого при цьому не втратиться [8].

Також важливим питанням є масштабування баз даних. Масштабування мікросервісів без збереження стану проводиться досить просто. А що робити, якщо ми зберігаємо дані в базі даних? Різні типи баз даних вимагають різних форм масштабування, і розуміння того, яка з цих форм пасуватиме найкращим чином саме для вашого випадку.

Багато сервісів в основному займаються зчитуванням даних. Масштабування для читання дається значно легше масштабування для запису. Тут велику роль може зіграти кешування даних.

В системі керування базами даних (RDBMS), подібної MySQL або Postgres, дані можна буде скопіювати з основного вузла в одну або кілька реплік. Сервіс може направляти всі запити на запис до єдиного основного вузла, але при цьому розподіляти запити на читання між декількома репліками, призначеними для зчитування даних (рис. 2.11).

Створення резервних копій з основної бази даних до реплік відбувається через деякий час після запису. Це означає, що при такій технології зчитування до завершення реплікації дані можуть бути застарілими. Через деякий час для операцій читання стануть доступні вже актуальні дані.

Рисунок 2.11 – Приклад організації взаємодії з БД

2.4 Інструменти для побудови MSA

Для успішної реалізації взаємодії мікросервісів та всієї внутрішньої роботи необхідно використовувати додаткові інструменти, які будуть описані нижче.

2.4.1 Єдина точка входу (API Gateway)

Мікросервісна система може складатися з десятків або, можливо, навіть сотень сервісів, тому клієнту досить тяжко взаємодіяти з всіма мікросервісами індивідуально. Більше того, навіть неможливо знайти всі мікросервіси без додаткових інструментів. У такій ситуації було прийнято використовувати шаблон API Gateway. Даний патерн виступає в якості єдиної точки входу для набору пов’язаних мікросервісів або навіть всієї системи.

Приклад єдиного шлюзу для входу наведено на рисунку 2.12. Цей шлюз визначає та перенаправляє кожен клієнтський запит на відповідний сервіс. Як вхідна точка до системи, API Gateway також відповідає за підтримку безпеки, перевірки OAuth-токенів та перевірки прав доступу клієнтів до конкретних сервісів.

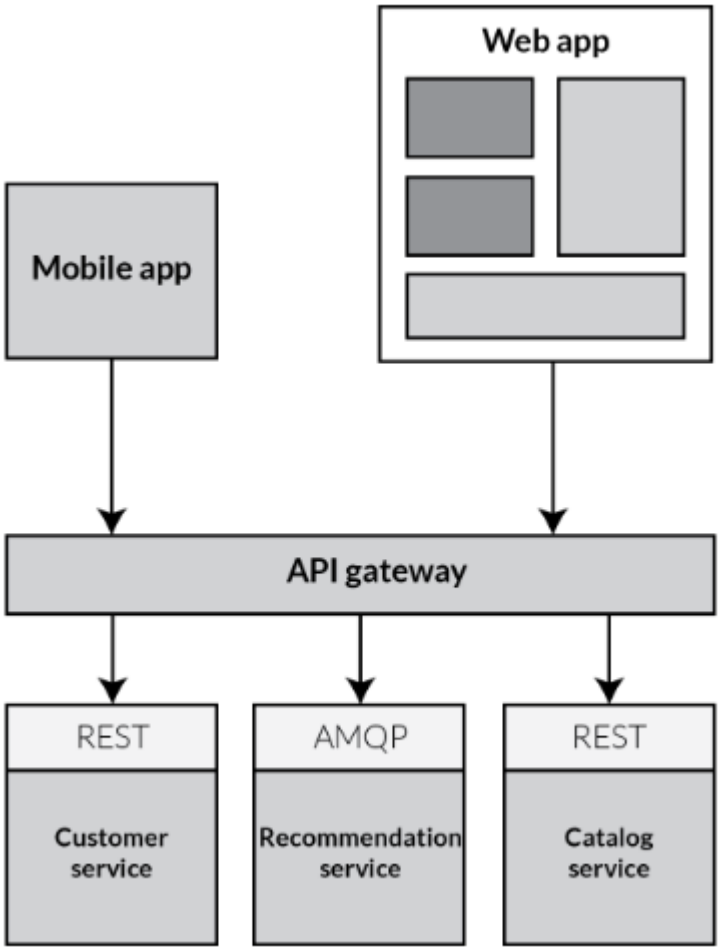

Рисунок 2.12 – Взаємодія зовнішніх клієнтів з мікросервісами [8]

Переваги використання спільної точки входу [9]:

- Єдиний адрес набагато зручніший сотні індивідуальних адресів API
- Зручно реалізовувати обмеження на кількість запитів в єдиному місці
- Вся система стає гнучкішою – можна змінювати внутрішню структуру
- Можна кешувати відповіді
- Можна поєднувати відповіді від різних сервісів.

Єдиний шлюз є деяким аналогом стандартного ООП-шаблону фасад, який надає єдиний зручний інтерфейс для роботи зі складною системою.

Даний шаблон має деякі недоліки. З'являється ще одна додатковий сервіс, який має бути створений та розгорнутий. Існує також ризик того, що єдина точка входу стане вузьким місцем для подальшого розвитку. Розробники повинні постійно оновлювати шлюз, щоб додавати кінцеві точки кожного мікросервісу.

Важливо щоб процес оновлення був якомога більш поверхневим. В іншому випадку, розробники будуть змушені чекати, щоб оновити його. Незважаючи на ці недоліки, для більшості реальних додатків використання спільного входу має досить великий сенс.

2.4.2 Виявлення сервісів (Service Discovery)

У мікросервісному середовищі, екземпляри регулярно додаються або видаляються при масштабуванні. Оскільки сервери та порти часто призначаються автоматично, клієнти, а також інші мікросервіси повинні бути в змозі знайти та ідентифікувати їх, щоб взаємодіяти. Це завдання вирішується за допомоги концепції виявлення сервісів, в якому ключовим є компонент, який називають реєстром сервісів, що відстежує всі доступні мікросервіси в системі. Кожен сервіс реєструється при запуску, використовуючи клієнт на самому сервісі, який здійснює зв'язок з реєстром, або через сторонній додаток, який контролює екземпляри сервісів в середовищі та зберігає свій статус в реєстрі. Розглянемо детальніше кожен з двох варіантів відкриття сервісів.

1. Виявлення на стороні клієнта.

При використанні даного підходу, клієнт відповідає за визначення місць екземплярів сервісів в мережі та балансування запитів через них. Клієнт запитує реєстр, який є базою даних доступних екземплярів сервісів. Потім клієнт використовує певний алгоритм балансування, щоб обрати один з наявних сервісів та зробити запит до нього. Даний підхід зображено на рисунку 2.13

Розташування в мережі екземпляру сервісу реєструється при його запуску в реєстрі сервісів, а також видаляється з цього реєстру при завершенні роботи відповідного сервісу. Також відбувається періодичне оновлення реєстру сервісів. Патерн виявлення на стороні клієнта має свої переваги та недоліки. Ця модель відносно проста і, за винятком реєстру сервісів, не вимагає інших додаткових 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.

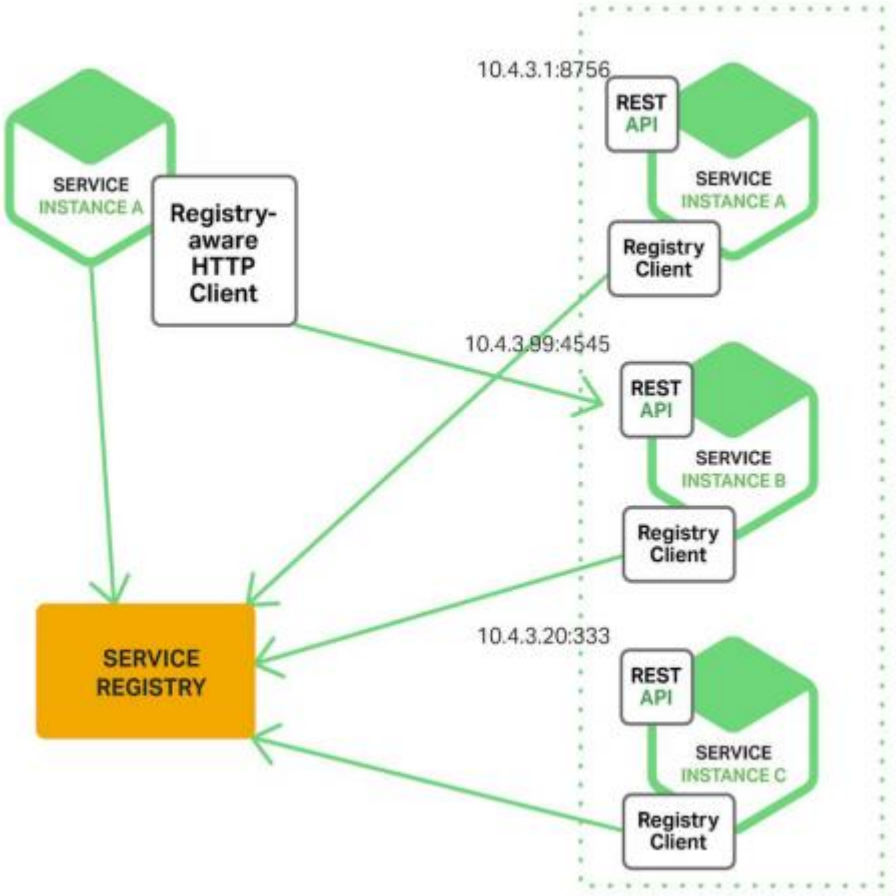

Рисунок 2.13 – Виявлення на стороні клієнта [9]

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.

2. Виявлення на стороні сервера

Клієнт робить запити до сервісу через балансувальник навантаження. Балансувальник навантаження посилає запит до реєстру сервісів та перенаправляє кожен запит до доступного сервісу. Як і в разі виявлення на клієнтській стороні, екземпляри служб реєструються і видаляються у сервісі реєстру сервісів (рис 2.14). Патерн виявлення на стороні сервера має власні переваги та недоліки. Важливим плюсом є те, що виявлення абстрагується від клієнта. Клієнти нічого не знають додатково, а просто звертаються до балансувальника навантаження.

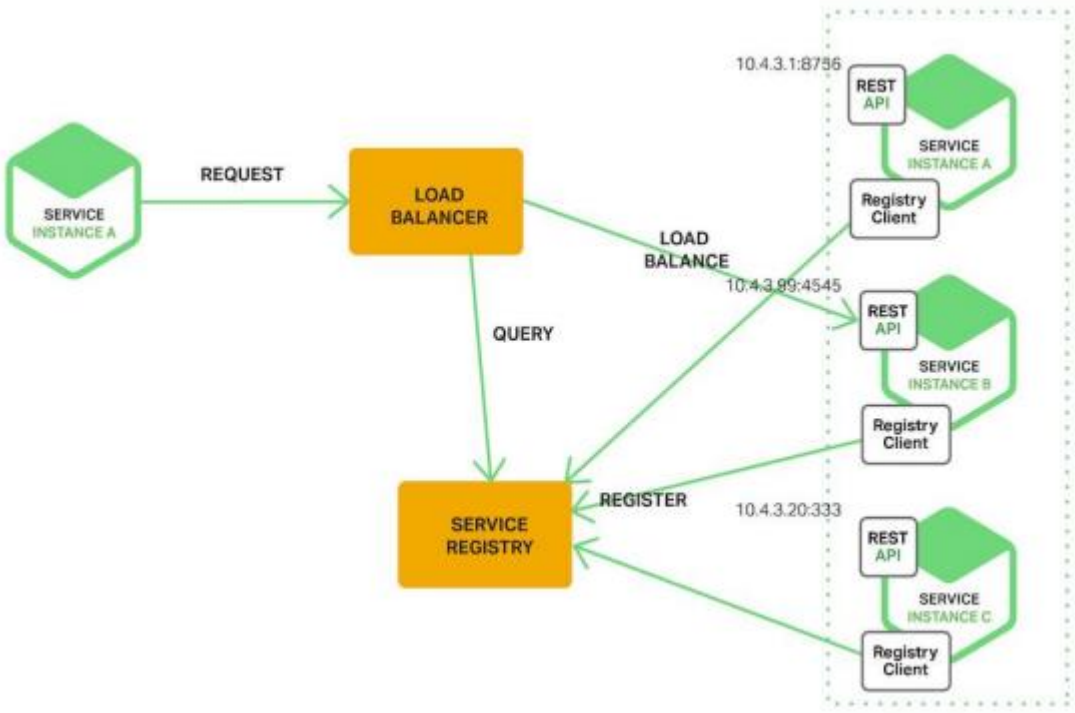

Рисунок 2.14 – Виявлення на стороні сервера [9]

Це усуває необхідність виконання логіки виявлення для кожної мови програмування та використовуваних фреймворків, які використовуються сервісними клієнтами. Крім того, деякі середовища розгортання пропонують дану функціональність безкоштовно [10].

До недоліків даного шаблону можна віднести те, що необхідно мати та підтримувати що один додатковий компонент системи – балансувальник навантаження.

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. Отже, реєстр сервісів складається з кластера серверів, які використовують протокол реплікації для забезпечення узгодженості.

2.4.3 Автоматичний вимикач (Circuit Breaker)

Мікросервіси часто співпрацюють при опрацюванні запитів. Коли один сервіс викликає синхронно інший, завжди є можливість того, що сервіс буде

недоступним або матиме великий час відгуку. Це може привести до виснаження ресурсів, які можуть зробити викликаючий сервіс не здатним опрацьовувати інші запити. Відмова одного сервісу може каскадним чином зупинити роботу всієї системи.

Для забезпечення надійності роботи системи використовують шаблон автоматичний вимикач, основна ідея якого відображена на рисунку 2.14 Необхідно обернути виклик об'єктом автоматичного вимикача, який буде стежити за виключними ситуаціями в системі. Як тільки несправність досягає деякого порогу, то вимикач спрацьовує та всі подальші виклики до вимикача повертаються з помилкою, а перший виклик не виконується взагалі.

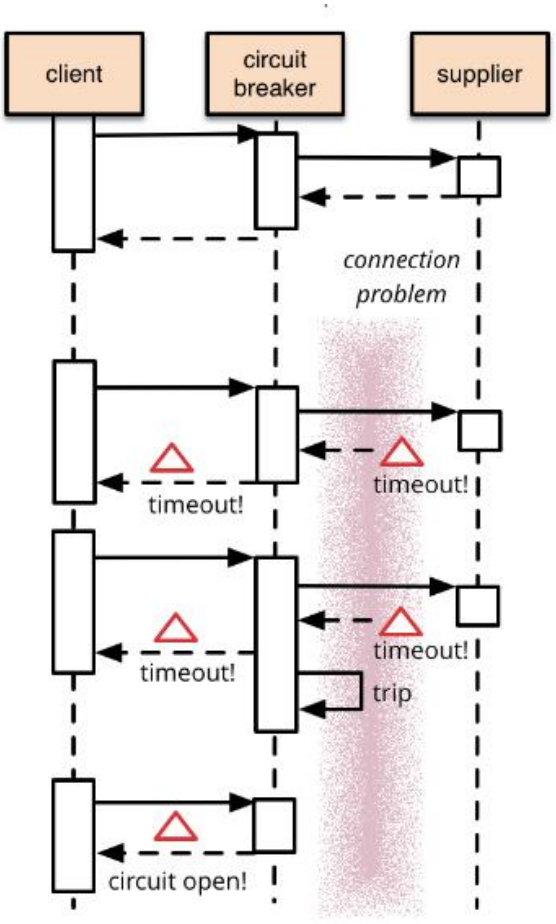

Рисунок 2.15 – Приклад роботи автоматичного вимикача

Зазвичай, необхідно мати додатковий інструмент, який сповіщає якщо вимикач спрацьовує.

Автоматичні вимикачі допомагають зменшити споживання ресурсів, які пов’язані з нестабільними операціями. Ви уникаєте очікування таймаутів, а також вимикач зменшує навантаження на сервер. Даний шаблон можна використовувати не тільки при віддалених викликах, а також в будь-якій програмній системі, щоб забезпечити надійність програми від збоїв в інших частинах. Для мікросервісів даний шаблон особливо цінний, оскільки він надає необхідну стійкість. Якщо він реалізований правильно, то виникнення каскадних збоїв не є можливим.

Реалізаціями даного патерну для мікросервісів є:

1. Netflix Hystrix
2. Nginx Circuit Breaker

2.4.4 Docker

Оскільки кількість мікросервісів може бути доволі великою та їхня технологічна природа може дуже відрізнятися, то команда розробників стикається з проблемою управління різноманітними середовищами для запуску сервісів. Для простоти створення різних необхідних програмних середовищ використовуються контейнери для інкапсуляції мікросервісів.

Найпопулярнішим інструментом для контейнеризації є Docker. Контейнеризація як альтернатива віртуалізації, завжди мала потенціал змінити шлях розгортання додатків. Docker як реалізація інструменту контейнеризації, часто порівнюється з віртуальними машинами. Віртуальні машини були створені з метою оптимізації використання обчислювальних ресурсів. На одному сервері можливо запустити декілька віртуальних машин та розгорнути окремі додатки на кожній з віртуальних машин. За цією моделлю, кожна з віртуальних машин надає стабільне програмне середовище для кожного додатку. Але при масштабуванні додатку ми отримуємо значні проблеми з продуктивністю, оскільки віртуальна машина споживає багато системних ресурсів.

Оскільки мікросервіси є відносно маленькими програмами, які необхідно розміщати в окремому програмному середовищі, то створення цілої віртуальної машини не є ефективним підходом. З Docker можливо зменшити витрати на

продуктивність та розгортати велику кількість сервісів на єдиному сервері, тому що Docker-контейнер вимагає набагато менше ресурсів.

Основні переваги використання Docker:

- Швидкий час запуску. Контейнер запускається в межах декількох секунд, оскільки він є процесом операційної системи, в той час запуск окремої віртуальної машини займе декілька хвилин.
- Швидше розгортання. Для контейнеру немає потреби кожен раз налаштовувати середовище, а лише необхідно завантажити відповідний образ.
- Просте управління та масштабування контейнерів.
- Краще використання обчислювальних ресурсів.
- Підтримка великої кількості різних операційних систем [11].

2.5 Висновки

У даному розділі було розглянуто найважливіші аспекти та головні шаблони реалізації мікросервісних додатків, яких рекомендовано дотримуватися для побудови ефективних програм. Також було здійснено огляд основних інструментів для забезпечення правильного функціонування всієї інфраструктури мікросервісів.

Використавши всі вище описані концепції, розроблений додаток повинен правильно та ефективно працювати, а також володіти необхідною відмовостійкістю.

3 ОГЛЯД РІЗНИХ ПРОГРАМНИХ ПЛАТФОРМ ДЛЯ ПОБУДОВИ MSA

Для створення мікросервісної архітектури існує багато інструментів для різних мов програмування. Відомо, що мікросервісний підхід дозволяє використання будь-яких технологій, які підтримують певні протоколи та інтерфейси. Для багатьох сучасних мов програмування існують вже готові рішення для створення та розгортання мікросервісів.

В даному розділі наведено перелік програмних платформ для найпопулярніших мов програмування, їх особливості при використанні.

3.1 Інструменти для C++

C++ широко використовується для створення продуктивного програмного забезпечення, мова має багату стандартну бібліотеку, яка включає в себе безліч корисних інструментів, які полегшують створення програм. Також C++ поєднує якості високорівневих та низькорівневих мов програмування.

Основні переваги, які надає мова для побудови мікросервісів:

- Висока обчислювальна продуктивність. За рахунок того, що C++ має доступ до всіх апаратних ресурсів та компілюється в машинні коди, то він дає перевагу в швидкодії над іншими мовами.
- Підтримка різних методологій програмування, таких як структурне, об’єктно-орієнтоване, узагальнене, функціональне, породжуюче програмування та інші.
- Наявність великої кількості навчальної літератури
- Широкі можливості даної мови програмування

Розглянемо фреймворки для створення мікросервісів на цій мові.

3.1.1 C++ MicroServices

CppMicroServices – це відкрита бібліотека для створення модульних програмних систем, які базуються на ідеях OSGi, які описують модель ля побудови додатку із компонентів, які пов'язані друг з другом через сервіси. Дана бібліотека має набір компонентів для створення модульних, динамічних сервіс-орієнтованих додатків [12].

Даний фреймворк надає наступні можливості:

- Повторне використання компонентів
- Низька зв'язність між сервісами
- Розподіл обов'язків відповідно до SOA
- Чисте API для сервісних інтерфейсів
- Створення розширюваних та гнучких систем

По суті, CppMicroServices надає потужний динамічний реєстр сервісів поверх потужного фреймворку, також керує, крім того, логічними модульними одиницями, які називаються пакетами, які розташовуються в загальних або статичних бібліотеках. Кожен пакет в бібліотеці поєднаний з певним контекстом через який доступний реєстр сервісів.

Для створення мікросервісу необхідно реалізувати один чи більше відповідних інтерфейсів, у випадку C++ інтерфейс – це абстрактний клас з одними віртуальними методами, без будь-яких членів класу.

Пакет – це набір специфічного ініціалізаційного коду, метаданих, які зберігаються в маніфест файлі та інших ресурсних файлах. Декілька пакетів можуть бути частиною тієї ж самої або різних бібліотек чи виконуваних файлів.

Для створення та використання сервісу, необхідно отримати екземпляр *BundleContext,* через який кожен пакет має доступ до C++ MicroServices API. Кожен пакет пов'язаний з унікальним контекстом, який доступний з будь-якого місця пакета через *GetBundleContext()* метод.

Даний фреймворк надає елегантне та чисте API та Discovery Service але багато функцій ще знаходяться в розробці.

3.1.2 Pistache framework

Pistache – це REST фреймворк, який надає можливість створення високопродуктивного програмного забезпечення для HTTP-серверу, а також для HTTP-клієнта. Дана бібліотека є відкрита та безкоштовна, вона написана на чистій C++11 версії без зовнішніх залежностей та забезпечує низькорівневу HTTP абстракцію.

Pistache забезпечує наступними компонентами «з коробки»:

- Багатопоточний HTTP сервер для створення власних API
- Асинхронний HTTP клієнт
- HTTP маршрутизатор, який перенаправляє запити до C++ функцій

Запити, які отримані Pistache опрацьовуються спеціальним класом – Http::Handler. Цей клас оголошує набір абстрактних методів, які можуть бути перевизначені для опрацювання спеціальних подій, які можуть виникнути при налаштуванні зв'язку між сокетами. Метод *onRequest()* повинен бути перевизначений, оскільки дана функція викликається завжди коли Pistache отримує дані та правильно розпізнає їх як http запит [13].

HTTP маршрутизація в даній бібліотеці складається зі зв'язування HTTP шляху з функцією зворотного виклику C++. Спеціальний компонент називається HTTP маршрутизатор, який відповідає за перенаправлення HTTP викликів на відповідні C++ функції. Маршрут складається із HTTP методу пов'язаного з певним ресурсом.

Даний фреймворк підтримує також асинхронне неблокуюче програмування. Прикладом є метод *send()* інтерфейсу *ResponseWriter*. Даний метод повертає кількість байтів записаних до файлового дескриптору сокета. Однак, замість того щоб повернути значення виклику з відповідним блокуванням, він обгортає значення в компонент, який називають *Promise*.

Promise – це реалізація Promises/A+ стандарту, який доступний багатьма JavaScript реалізаціями. Під час асинхронного виклику, *Promise* розділяє запуск асинхронної операції від отримання відповідних результатів.

Отже, Pistache є сучасною бібліотекою для створення якісних та

високопродуктивних додатків на C++. Вона є гарним вибором якщо вам необхідно створити асинхронну програму за REST архітектурою.

3.1.3 UServer / ULib

ULib – це високооптимізована бібліотека для написання C++ додатків. Вона була створена для розробки різноманітних програмних систем. ULib – дуже легка C++ бібліотека, яка полегшує використання шаблонів проектування для систем, які використовують uclibs, а також підтримує потоки через POSIX. Цей фреймворк відключає функції мови, які споживають додаткову пам'ять чи вводять накладні витрати під час виконання, наприклад RTTI та опрацювання виняткових ситуацій. Також передбачає, що будуть використані бібліотеки, які базуються на чистому C, а не перевантажені C++ бібліотеки, тому вона має переваги у швидкодії в порівнянні з іншими відомими C++ бібліотеками.

Дана бібліотека підтримує наступні елементи:

- Підтримка протоколів HTTP/1.0 та HTTP/2
- Підтримка CGI
- Використання динамічних USP сторінок

Дана бібліотека надає можливість створення динамічних веб-сторінок подібних до JSP(Java Server Pages) чи ASP(Active Server Pages).

ULib рекомендовано використовувати, якщо необхідно мати високу швидкодію та зручний у використані C-сервер.

3.2 Інструменти для Java

Java – одна з найпопулярніших та найпотужніших мов програмування сучасності. Програми на Java транслюються в байт-код, який виконується віртуальною машиною Java(JVM) – програмою, яка опрацьовує байтовий код та передає інструкції обладнанню як інтерпретатор. Перевага подібного способу виконання програм є повна незалежність байт-коду від операційної системи та апаратури, що дозволяє виконувати Java-додатки на будь-якому пристрої, для якого існує відповідна віртуальна машина.

Java, без пребільшення, є найпопулярнішою мовою для побудови мікросервісів. Багато відомих компаній створюють свої системи саме на цій мові, прикладом є Netflix, Amazon.

3.2.1 Spring framework

Spring framework – найпотужніша Java-бібліотека, яка дозволяє створювати програмні системи будь-якої складності. Виник як альтернатива J2EE платформи. Для побудови мікросервісів найважливіші модулі Spring це Spring Boot та Spring Cloud.

Spring Boot – складова екосистеми бібліотеки Spring. Spring Boot дозволяє легко створювати повноцінні, ефективні Spring-додатки. Для налаштування програмної системи необхідно відносно мало конфігурацій, в порівнянні з Spring MVC.

Основні особливості Spring Boot:

- Створення повноцінних Spring додатків;
- Вбудований сервер Tomcat або Jetty;
- Автоматична конфігурація Spring framework;
- Забезпечує можливостями моніторингу стану системи;
- Не вимагає написання конфігураційних файлів на XML [14].

Spring Boot є основою для більш складного фреймворку Spring Cloud, який призначений для створення розподілених систем та має широкий інструментарій для розробки.

З основних переваг використання Spring Cloud виділимо наступні:

- наявність реєстру сервісів та системи виявлення сервісів, маршрутизації, міжсервісних викликів;
- балансування навантаження;
- наявність автоматичного вимикача;
- розподілений обмін повідомленнями.

Spring Cloud надає декларативний підхід до створення програмного забезпечення.

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

В даному прикладі використовується лише єдина анотація *@EnableDiscoveryClient* для введення в систему механізму виявлення сервісів. Також Spring Cloud має чудову інтеграцію з Netflix OSS, та надає дані інструменти «з коробки»:

```
@SpringBootApplication
@EnableEurikaServer
public class ServiceRegistrationServer {
    public static void main(String[] args) {
        System.setProperty(“spring-config.yml”,
“registration-server”);

SpringApplication.run(ServiceRegistrationServer.class,
args);
    }
}
```

Вище наведено приклад використання компоненту Netflix OSS через Spring Cloud. Eurika – це сервер, який виконує функцію реєстрації сервісів.

Для включення балансувальника навантаження також необхідно докласти мінімально зусиль, а саме включити анотацію *@LoadBalanced* над відповідним Spring-компонентом.

```
@Autowired
@LoadBalanced
protected RestTemplate restTemplate;
```

Також Spring Cloud надає горизонтально масштабоване сховище конфігурацій для розподілених систем. Як джерело даних на даний момент підтримується Git, Subversion та прості файли, що зберігаються локально. За замовчуванням Spring Cloud Config віддає файли, які відповідають імені викликаючого Spring додатку.

Spring Cloud надає зручні анотації та автоконфігурації для забезпечення аутентифікації, створення токенів OAuth2 для доступу до ресурсів бекенду. Spring

Cloud спрощує підключення до сервісів та отримання можливостей середовища в хмарних платформах, таких як Cloud Foundry і Heroku. Особлива підтримка Spring-додатків через Java і XML-конфігурації робить підключення до хмарних сервісів тривіальним завданням. Ви можете використовувати існуючі хмарні коннектори або написати власний для вашої хмарної платформи. "З коробки" підтримуються найбільш популярні сервіси (реляційні СУБД, MongoDB, Redis, Rabbit), але також можливе розширення для ваших сервісів. Жоден з сервісів не вимагає зміни самого Spring Cloud, досить просто додати необхідну вам jar-бібліотеку в область видимості classpath [14].

3.2.2 Spark framework

Spark framework – це проста та легка Java веб-бібліотека для швидкої розробки програмних систем [15]. Ціль даного фреймворку це забезпечити інструменти для побудови ефективних та елегантних веб-додатків на Java. Основною особливістю є те, що Spark побудований за філософією лямбда-виразів, які були введені в Java 8.

Даний фреймворк є гарним вибором для побудови мікросервісів, оскільки він вимагає написання невеликих об’ємів коду. Spark загалом використовується для створення REST API, але одночасно підтримує велику кількість різноманітних шаблонів. Наведемо приклад програми написаної на Spark framework.

```
import static spark.Spark

public class Main {
    public static void main(String[] args) {
        get(“/hello”, (req, res) -> “Hello world”);
    }
}
```

Даний приклад демонструє створення простого серверного додатку, яке повертає сторінку з текстовим рядоком на GET виклик за адресою http://localhost:8080/hello.

Spark є ідеальним рішенням для невеликих та ефективних веб-додатків.

3.2.3 Restlet

Restlet – це бібліотека для створення потужних RESTful веб-додатків, яка має великий набір інструментів та можливостей. Вона доступна для більшості платформ основаних на JVM (Java SE/EE, Google App Engine, GWT, Android).

Restlet пропонує платформу для побудови REST API за концепцією «Перш за все API» («API First»). Дана концепція передбачає те, що API має найвищий пріоритет та для якої користувач API є головним користувачем системи.

RestComponent володіє всім необхідним для конфігурації серверу та налаштування додаткових функцій, таких як безпека та інше. В даному фреймворку надається високе значення поняттю ресурса, як і для REST стилю в цілому [16].

Приклад простого Restlet-клієнта під платформу Android:

```
    ClientResource resource = new
ClientResource("http://localhost/test")
    resource.setRequestEntityBuffering(true);
    UserResource testResource =
resource.wrap(UserResource.class);
    User user = testResource.retrieve();
```

Даний приклад демонструє отримання ресурсу від віддаленого сервера, який в даному прикладі виконує localhost.

Restlet на даний момент не є настільки популярним як Spring framework чи Spark, але пропонує елегантний підхід до створення REST API та підтримує велику кількість існуючих платформ.

3.3 Інструменти для Python

Python – це мова програмування, яка орієнтована на збільшення продуктивності розробника та читабельність коду. Синтаксис ядра Python мінімалістичний, а стандартна бібліотека включає великий об'єм корисних функцій.

Основні напрямки використання Python:

- Розробка веб-додатків(фреймворк Django)
- Аналіз даних та машинне навчання
- Швидке прототипування ідей в бізнесі за рахунок існування готових бібліотек, низького порогу входження та високої продуктивності Python-програмістів
- Написання скриптів для автоматизації

Python значно поступається в швидкодії мовам зі статичною типізацією, таким як C/C++ та Java, але його перевага полягає у ефективності та швидкості написання коду.

Розглянемо головні фреймворки для створення мікросервісів.

3.3.1 Flask

Flask – це фреймворк для побудови веб-додатків на мові Python, який є значно легший від більш популярного Django. Також існує додаткова бібліотека Flask-Injector, яка надає можливості для впровадження залежностей.

Приклад простого веб-додатку на Flask:

```
from flask import Flask

app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello, World!'
```

В даному коді ми імпортуємо клас Flask. Екземпляр даного класу буде нашим WSGI додатком. Далі ми створюємо об'єкт даного класу з аргументом, який є ім'ям модуля чи пакету програми. Потім ми використовуємо декоратор route(), щоб задати відображення URL до певного методу [17]. В кінці, функція повертає повідомлення, яке ми хочемо відобразити на браузері користувача.

Щоб запустити додаток необхідно зробити наступне:

```
$ export FLASK_APP = app.py
$ flask run
* Running on http://127.0.0.1:5000/
```

В даному прикладі, запускається простий сервер, який може бути використаний у цілях тестування. Flask залежить від двох важливих зовнішніх бібліотек: Jinja2 – шаблонізатор та Wekzeug – набір інструментів для WSGI.

Даний фреймворк є гарним рішенням для реалізації мікросервісів на мові Python.

3.3.2 Tornado

Tornado – веб-фреймворк та асинхронна мережева бібліотека для створення продуктивних програмних систем. Використовуючи неблокуючі мережеві операції введення-виведення, Tornado дає можливість підтримувати десятки тисяч відкритих з'єднань, що робить даний фреймворк ідеальним для довгого опитування(long polling) та технології веб-сокетів та додатків, які вимагають довготривалих активних з'єднань на кожного користувача [18].

Проста програма має вигляд:

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, World!")
```

```
        def make_app():
            return tornado.web.Application([“/”,
MainHandler])
    if __name == “__main__”
        app = make_app()
        app.listen(8888)
        tornado.ioloop.IOLoop.current().start()
```

Дана програма демонструє просте створення сервера, без використання всіх асинхронних можливостей Tornado.

Наведену бібліотеку рекомендовано використовувати, якщо перед вами поставлена задача створення сервісів, які працюють в режимі реального часу та вимагають великої кількості активних з’єднань. В традиційних синхронних серверах використовується підхід, коли на кожного користувача виділяється окремий потік, створення якого може бути дуже ресурсоємним.

3.3.3 Nameko

Nameko – це бібліотека для створення мікросервісів для Python, яка дозволяє розробникам сервісу сконцентруватися на логіці додатку та закликає до спроможності до тестування.

Основні можливості, які підтримує даний фреймворк:

- Підтримка AMQP, RPC та подій
- Підтримка веб-сокетів
- Підтримка командного рядку для простої та швидкої розробки
- Утиліти для інтеграційних та юніт тестів

Сервіс написаний на Nameko виглядає як:

```
from nameko.rpc import rpc

class DummyService:
    name = “dummy_service”
```

```
@rpc
def hello(self, name):
    return “Hello, {}!”.format(name)
```

Щоб запустити сервіс необхідно викликати наступну команду:

```
$ nameko run app
* starting services: dummy_service
```

Nameko здатний до масштабування від одного сервісу до кластеру, який складається з багатьох екземплярів різних сервісів. Бібліотека також надає інструменти для клієнтів, які ви можете використовувати для написання Python програм для взаємодії з існуючим кластером Nameko [19].

3.4 Висновки

В даному розділі було розглянуто найвідоміші програмні платформи для деяких мов прорамування. З переглянутих мов найбільше виділяється Java та її фреймворк Spring Cloud, який надає багатий інструментарій у вигляді інфраструктурних сервісів створений компанією Netflix.

Python є гарним рішенням для швидкої розробки та підтвердження концепції майбутньої системи. Найкращим вибором фреймворку під дану мову є бібліотека Flask.

Вибір C++ у якості інструменту для створення мікросервісів має сенс, якщо швидкодія системи має вирішальне значення, в іншому разі краще використати мову зі зручнішими бібліотеками та швидкістю розробки.

4 ПРИКЛАД СТВОРЕННЯ ТЕСТОВОЇ СИСТЕМИ НА БАЗІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

В даному розділі буде розглянуто приклад побудови програмної системи за мікросервісною архітектурою на мовах Java та Python. Для інтеграції сервісів використаємо інструменти компанії Netflix, які знаходяться у вільному доступі, а саме: Zuul, Hystrix, Eureka.

Тестова система надає можливість здійснити реєстрацію користувачу, а також отримати перелік всіх фільмів, здійснити пошук по жанрах, а також замовити деякий фільм.

Всю систему можна умовно розподілити на 2 основні частини: набір допоміжних сервісів, таких як точка для єдиного входу, автоматичний вимикач, а також незалежні самостійні мікросервіси. Всього в системі три мікросервіси: Movie Service, User Service, Booking Service. Система побудована за шаблоном агрегатор.

Movie Service надає системі можливість отримати список фільмів, які представлені у окремій базі даних, а також здійснити фільтрування по певному жанру фільму.

User Service оперує даними про користувачів та тісно співпрацює із іншими сервісами в системі. Наприклад, при відображенні списку обраних фільмів для користувача ми звертаємось до Movie Service, а для перегляду всіх замовлень користувача, ми йдемо до Booking Service.

Booking Service забезпечує систему функціоналом для замовлення та обирання фільмів користувачами.

Вся взаємодія між сервісами є синхронною та відбувається шляхом звертання до певної точки REST API кожного сервісу. Для розгортання кожного сервісу використовується окремий Docker-контейнер із налаштованим середовищем.

Основні компоненти системи наведені на рис 4.1

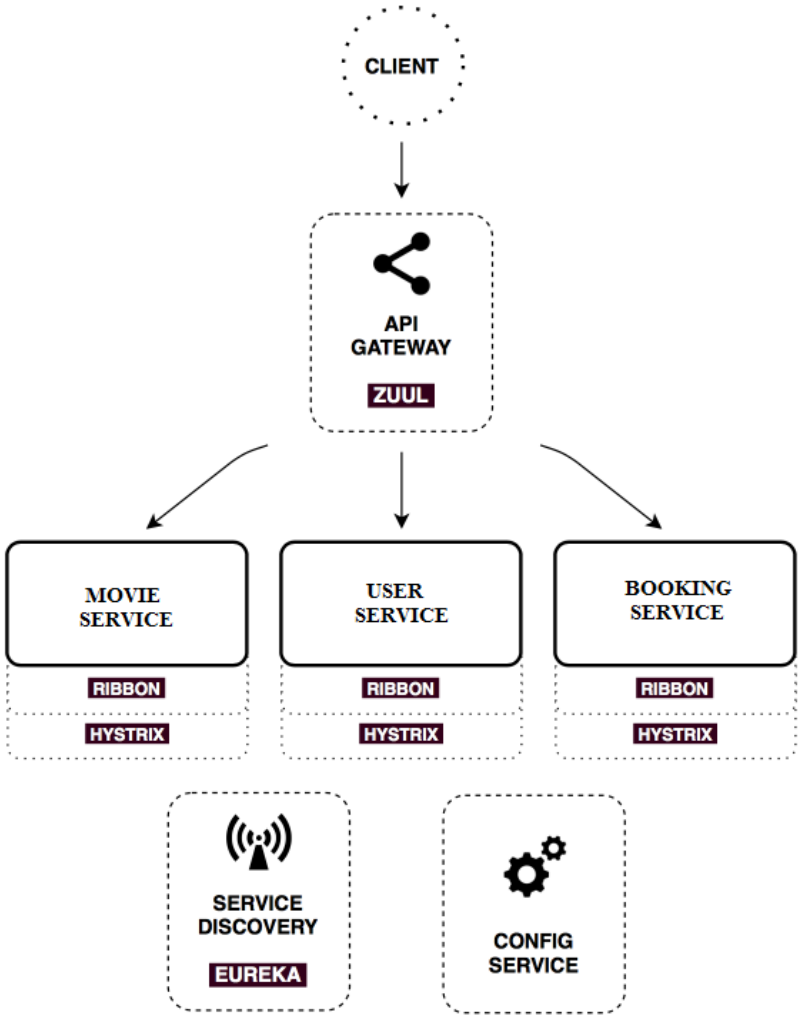

Рисунок 4.1 – Загальна архітектура додатку

Далі буде детально розглянуто кожен із складових компонентів системи.

4.1 Інфраструктурні сервіси

Для забезпечення ефективної роботи системи необхідно вводити додаткові мікросервіси, які реалізують шаблони мікросервісної архітектури, такі як API Gateway, Service Discovery, Circuit Breaker.

Розглянемо детальніше інструменти, які були використані для реалізації вищезазначених патернів.

4.1.1 Netflix Zuul

Netflix Zuul – це реалізація API Gateway від Netflix. Для налаштування Zuul

серверу було використано можливості фреймворку Spring Cloud, який дозволяє створити даний сервер використавши лише одну анотацію над стандартним java-класом.

```
@SpringBootApplication
@Controller
@EnableZuulProxy
public class ZuulServerApplication {
    public static void main(String[] args) {
        new
SpringApplicationBuilder(ZuulServerApplication.class).web(t
rue).run(args);
    }
}
```

Zuul працює як інтерфейс, який надає єдиний вхід до всієї системи та делегує виклики до підсистем. Кожен з мікросервісів мають власні REST APIs, які використовуються для маршрутизації даним сервером.

Файл конфігурації серверу application.yml має наступний вигляд:

```
zuul:
    ignoredServices: “*”
    routes:
       movies: /movies/**
       users: /users/**
```

Даний елемент конфігураційного файлу описує як саме необхідно перенаправляти запити, які надходять на Zuul.

4.1.2 Netflix Eureka

Service Discovery є одним з ключових принципів мікросервісної архітектури. Задача виявлення сервісів була вирішена використанням Netflix Eureka. Eureka – це сервер для реєстрації всіх сервісів, які знаходяться в системі. Для запуску серверу треба створити простий Spring Boot додаток з анотацією @EnableEurekaServer:

```
@SpringBootApplication
@EnableEurikaServer
public class EurekaServer {
    public static void main(String[] args) {
```

```
        SpringApplication.run(EurekaServer.class, args);
    }
}
```

При правильному налаштуванні серверу, відкриється вікно браузера з панеллю управління, приклад якої наведено на рисунку 4.2

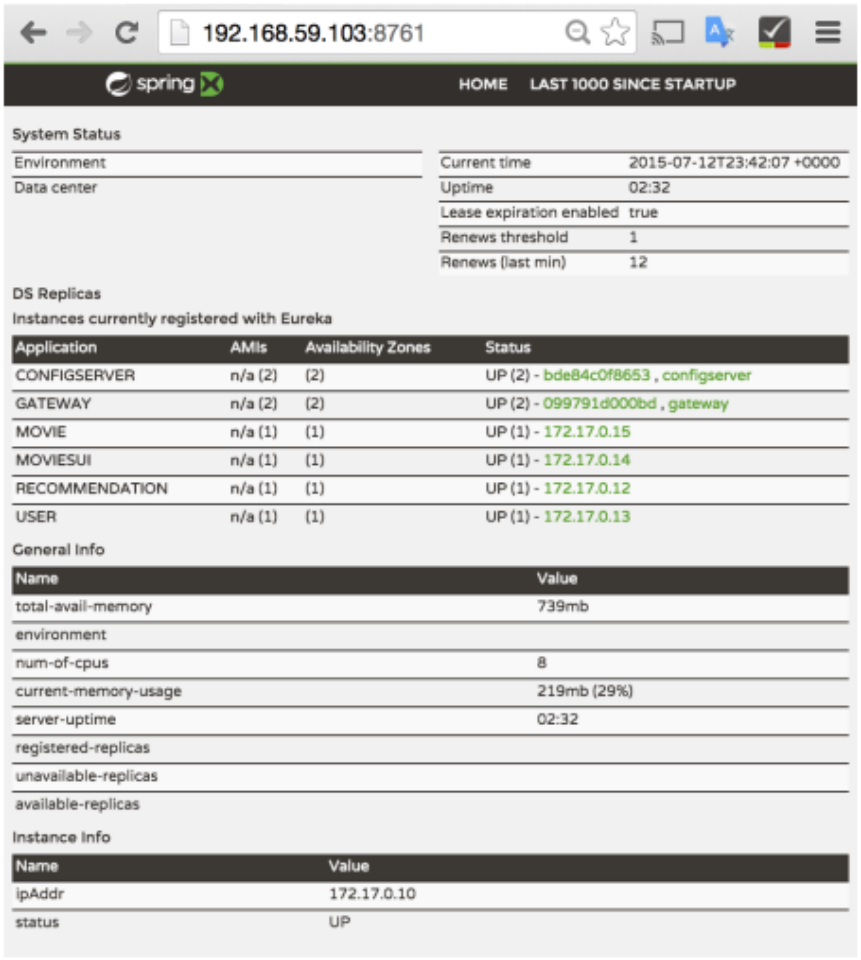

Рисунок 4.2 – Панель сервісу Netflix Eureka

Дана панель демонструє всі сервіси, які зареєстровані в системі. Окрім того, можна отримати доступ до одного із сервісів, наприклад, до сервісу користувачів, виконавши команду:

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

Вище зазначена команда перейде до кінцевої точки Gateway API і проксі REST API, який надається сервісом користувачів. Ці REST API були налаштовані

на використання HATEOAS, який підтримує автоматичне виявлення всіх функціональних можливостей сервісу. Отримаємо наступний результат в JSON-нотації:

```
{
    "_link": {
        "_self": {
            "href": "http://127.0.0.1:1000/user"
        },
        "resume":{
         "href": "http://127.0.0.1:1000/user/resume"
     },
     "pause": {
         "href": "http://127.0.0.1:1000/user/pause"
     },
     "restart": {
         "href": "http://127.0.0.1:1000/user/restart"
     },
     "metrics": {
         "href": "http://127.0.0.1:1000/user/metrics"
     },
     "env": [{
         "href": "http://127.0.0.1:1000/user/env"
    }],
     "beans": {
         "href": "http://127.0.0.1:1000/user/beans"
     },
     "info": {
         "href": "http://127.0.0.1:1000/user/info"
     },
     "routes": {
         "href": "http://127.0.0.1:1000/user/routes"
```

```
        }
      }
}
```

4.1.3 Netflix Hystrix

Також в цьому додатку було використано автоматичний вимикач для кожного сервісу, який реалізовано в Netflix Hystrix Dashboard. Якщо мікросервіс не відповідає через деякі внутрішні помилки, Hystrix може перенаправити запит на спеціальний внутрішній метод. У випадку, якщо сервіс працює з постійними збоями, Hystrix буде вимикати сервіс та викликати спеціальний метод, який опрацьовує дану ситуацію та не буде викликати даний сервіс поки він не стане знову працездатним. Для того щоб визначити чи доступний сервіс знову, Hystrix дозволяє надсилати декілька запитів, щоб дізнатися про стан мікросервісу.

Для створення спеціального методу, який буде опрацьовувати виключні ситуації просто додаємо відповідну анотацію:

```
@HystrixCommand(fallbackMethod = "defaultUsers")
public Object getMovies(final Map<String, Object>
parameters) {
    // do stuff that might fail
}
```

Метод *getMovies()* буде викликано лише при виникненні виключних ситуацій, прикладом яких можуть бути збої в мережі або самого серверу.

Після правильного запуску Hystrix переходимо на відповідний сервер та можемо спостерігати наступну панель. Дана панель надає опис станів системи та кожного автоматичного вимикача, приклад якої наведено на рисунку 4.3

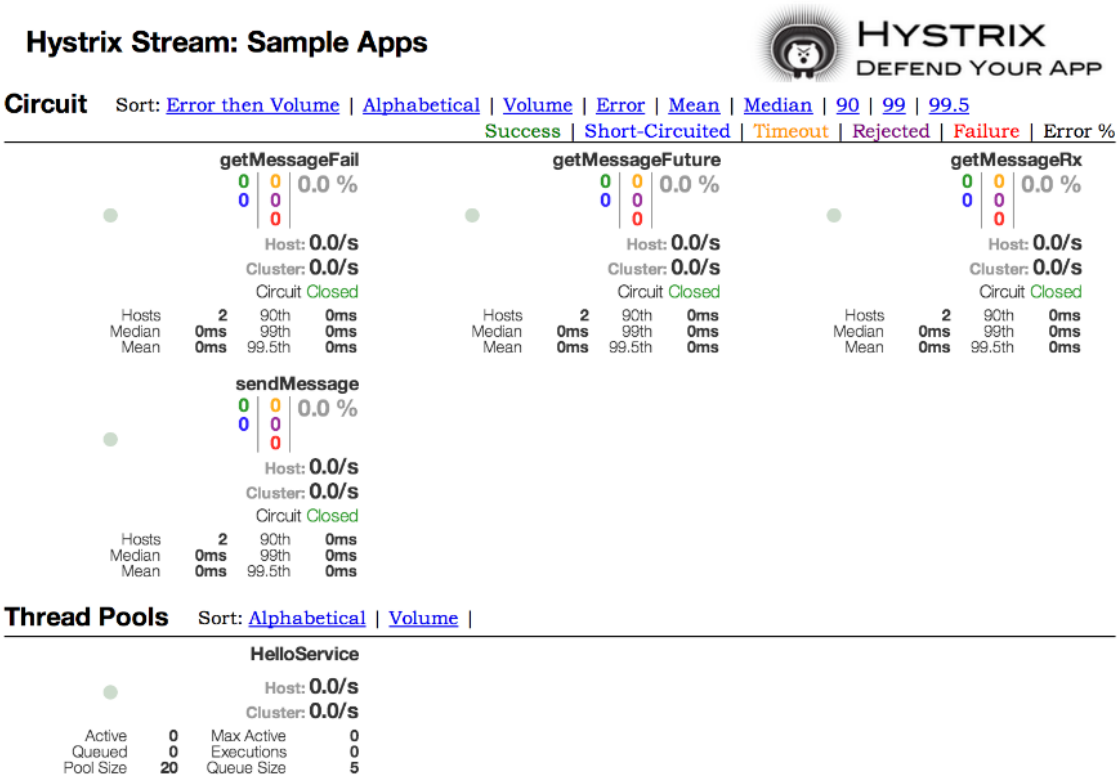

Рисунок 4.3 – Приклад Hystrix dashboard

4.2 Movie Service

Даний мікросервіс є важливою частиною всієї системи та зберігає інформацію про фільми та здійснює пошук по жанрах. Він був створений на платформі Spring framework та розгортається в окремому процесі, використовуючи Docker. Для зберігання даних використана графова база даних neo4j.

Даний сервіс співпрацює із іншими сервісами для надання представлення фільмів для користувачів. Він являє собою простий CRUD-додаток, для створення якого було використано інструменти Spring Data, а саме шаблон репозиторій з поєднанням анотації *@RepositoryRestResource*, яка створює відповідний контролер для взаємодії з певним репозиторієм:

```
@RepositoryRestResource(collectionResourceRel = “genres”,
path = “genres”)
public interface GenreRepository extends
```

```
PagingAndSortingRepository<Genre, Long> {
List<Genre> findByName(@Param(“0”) String name);
}
```

Даний клас дозволяє робити виклики до даного репозиторію не створюючи явно необхідні контролери, приклади HTTP-викликів та їх результати:

```
$ curl –i –X POST –H “Content-Type:application/json” –d
“{\”name\”:\”thriller\”}” http://localhost/genres
```

HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
Location: http://localhost/genres/1
Content-Length: 0
Date: Sun, 21 May 2017 14:03:46 GMT

Параметри команди *curl* :

- -i – гарантує, що ви побачите відповідь із заголовками
- -X POST – виклик HTTP-POST методу, який забезпечує створення нової сутності
- -d – дані, які будуть відіслані серверу

Для перерахунку всіх жанрів, необхідно зробити наступний виклик:

```
$ curl http://localhost/genres
```

4.3 User Service

Даний сервіс було створено за допомоги мови програмування Python та фреймворку Flask. Вказаний мікросервіс оперує даними про користувачів та взаємодіє із іншими сервісами в системі через REST по протоколу HTTP. Спілкуючись з Movie сервісом, система надає користувачеві список фільмів, по певних жанрах.

Також цей сервіс повертає інформацію про користувачів існуючих в системі.

В якості сховища даних мікросервіс використовує стандартну реляційну базу даних.

Для реалізації контролерів використаємо можливості фреймворку Flask, приклад реалізації контролера:

```
@app.route("/users/<username>", methods=['GET'])
def user_record(username):
    if username not in users:
        raise NotFound

    return get_user(username)
```

Анотацією *@app.route* ми вказуємо шлях та HTTP-метод, що викликає вказаний метод.

Викликавши *curl* отримаємо наступний результат:

```
$ curl http://localhost:5000/users/john_smith
{
    "id": "john_smith",
    "last_active": "1360031625",
    "name": "John Smith"
}
```

Слід зауважити, що для python мікросервісу небхідно використовувати додаткові інструменти для інтеграції з Hystrix та Eureka.

Приклад створення автоматичного вимикача для python додатку.

```
from hystrix import Command
class FallbackCommand(Command):
    def run(self):
        return default()
```

Також необхідно використовувати додатковий клієнт для Netflix Eureka:

```
from eureka.client import EurekaClient
ec = EurekaClient("UserService",
```

```
                    eureka_domain_name="test.domain.net",
                    port=80, secure_port=443)
ec.register()
ec.update_status("UP")
```

4.4 Booking Service

Даний мікросервіс відповідає за збереження та пошук інформації про замовлення користувачів та співпрацює із користувацьким мікросервісом. Реалізований на мові програмування Python та фреймворку Flask аналогічно попередньому мікросервісу. Щоб отримати всі замовлення в системі, треба звернутися до екземпляру сервісу, який знаходиться, наприклад, за адресою http://127.0.0.1:5003, на /bookings, і сервіс поверне список замовлень.

```
GET /bookings
{
    "john_smith": {
        "20170521": [
            "267eedb8-0f5d-42d5-8f43-72426b9fb3e6"
        ]
    }
}
```

Також можна отримати інформацію про замовлення конкретного користувача:

```
GET /bookings/john_smith
{
    "20170521": [
        "7daf7208-be4d-4944-a3ae-c1c2f516f3e6",
        "267eedb8-0f5d-42d5-8f43-72426b9fb3e6"
    ],
    "20170522": [
        "a8034f44-aee4-44cf-b32c-74cf452aaaae",
```

```
        "276c79ec-a26a-40a6-b3d3-fb242a5947b6"
    ]
}
```

Аналогічно до User Service, необхідно забезпечити інтеграцію з загальною системою, використавши додаткові налаштування для серверу Eureka та Hystrix.

4.5 Висновки

У даному розділі було розглянуто створення тестового додатку, який складається з декількох сервісів, які написані на різних мовах програмування та на різних фреймворках. Також було використано такі шаблони мікросервісної архітектури як Gateway API, Service Discovery та Circuit Breaker за допомоги відповідних інструментів. Дана система піддається горизонтальному масштабуванню шляхом збільшення вузлів в системі та володіє необхідною відмовостійкістю.

5 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

5.1 Вступ

У даному розділі проводиться аналіз варіантів реалізації програмного продукту з метою вибору оптимальної, з економічної точки зору. Буде здійснено функціонально-вартісний аналіз (ФВА). Програмний продукт є крос-платформенним та рекомендується для використання на персональних комп'ютерах під управлінням операційних систем Windows, Linux чи Mac.

Нижче наведено аналіз різних варіантів реалізації модулю з метою вибору оптимальної, з огляду при цьому як на економічні фактори, так і на характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. Як прямі, так і побічні витрати розподіляються по продуктам та послугам у залежності від потрібних на кожному етапі виробництва обсягів ресурсів. Виконані на цих етапах дії у контексті метода ФВА називаються функціями.

Мета ФВА полягає у забезпеченні правильного розподілу ресурсів, виділених на виробництво продукції або надання послуг, на прямі та непрямі витрати. У даному випадку – аналізу функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Фактично цей метод працює за таким алгоритмом:

– визначається послідовність функцій, необхідних для виробництва продукту. Спочатку – всі можливі, потім вони розподіляються по двом групам: ті, що впливають на вартість продукту і ті, що не впливають. На цьому ж етапі

оптимізується сама послідовність скороченням кроків, що не впливають на цінність і відповідно витрат.

– для кожної функції визначаються повні річні витрати й кількість робочих часів.

– для кожної функції наоснові оцінок попереднього пункту визначається кількісна характеристика джерел витрат.

– після того, як для кожної функції будуть визначені їх джерела витрат, проводиться кінцевий розрахунок витрат на виробництво продукту.

5.2 Постановка задачі техніко-економічного аналізу

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки. Оскільки основні проектні рішення стосуються всієї системи, кожна окрема підсистема має їм задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для організації системи пропонування фільмів за мікросервісною архітектурою. Відповідно цьому варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

– програмний продукт повинен функціонувати на будь-яких системах з певним набором компонент;

– забезпечувати високу швидкодію для великої кількості запитів від користувачів до всієї системи та до кожного мікросервіса окремо;

– забезпечити простий та зручний користувацький інтерфейс;

– забезпечити завадостійкість системи у критичних ситуаціях;

– передбачати мінімальні витрати на впровадження програмного продукту.

5.2.1 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукта, який підтверджує концепцію мікросервісної архітектури на практичному рівні.

Виходячи з конкретної мети, можна виділити наступні основні функції ПП:

F_1 – використання інструменту для розгортання;

F_2 – вибір інструменту для реалізації сервісу відкриття;

F_3 – вибір підходу до інтеграції мікросервісів.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

а) Oracle VM;

б) Docker;

Функція F_2:

а) Netflix Eureka;

б) HashiCorp Consul;

Функція F_3:

а) черга повідомлень;

б) взаємодія через REST API

5.2.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 5.1). На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 5.1). На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам.

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, вони не відповідають поставленим перед програмним продуктом задачам.

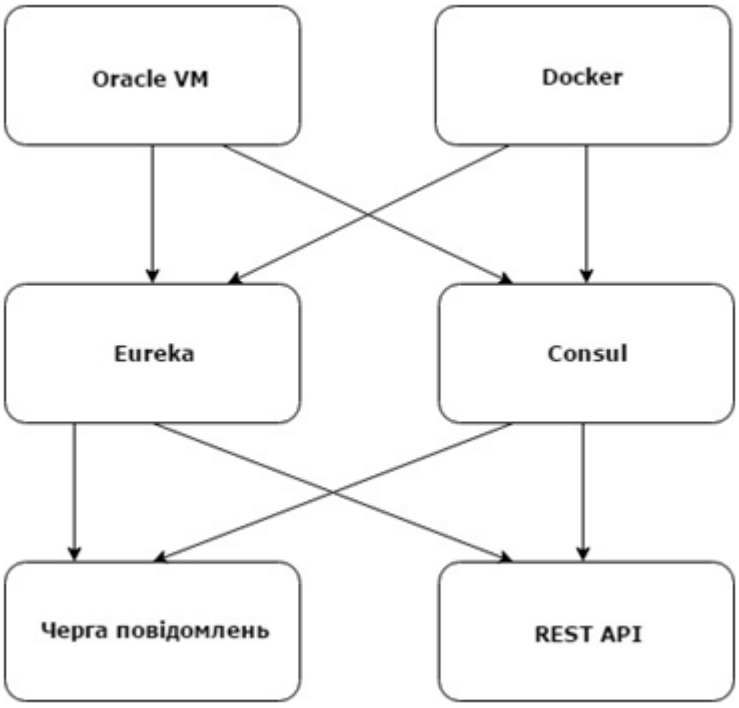

Рисунок 5.1 – Морфологічна карта

Морфологічна карта відображає всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам.

Функція *F1:*

Оскільки нам необхідно швидко розгортати та впроваджувати наш продукт, тому варіант а) має бути відкинутим.

Функція *F2:*

Оскільки нам необхідно підтримувати будь-які мови програмування одночасно, то варіант б) має бути відкинутим.

Функція *F3:*

Оскільки тип інтеграції не впливає на ключові параметри системи, то вважаємо варіанти а) та б) рівноможливими.

Таблиця 5.1 – Позитивно-негативна матриця

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F1	*А*	Повнофункціональний інструмент, гнучке налаштування	Вимагає багато ресурсів
	Б	Легкий та швидкий для розгортання середовищ	Менша гнучкість у використанні
F2	А	Інтеграція з іншими інструментами для підтримки системи	Складність налаштування
	Б	Висока продуктивність	Прив’язаний до однієї програмної платформи
F3	*А*	Завадостійкість, можливість виконувати задачі паралельно	Вищі витрати часу на написання коду
	Б	Простіший код, який легше писати та розуміти	Обмеженність системи у гнучкості

Таким чином, будемо розглядати такі варіанти реалізації ПП:

1. F1б – F2а – F3а
2. F1б – F2а – F3б

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

5.3 Обґрунтування системи параметрів ПП

5.3.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо

використовувати наступні параметри:

– *X1* – швидкодія інформаційної системи;

– *X2* – простота використання програмного забезпечення;

– *X3* – кількість програмного коду;

– *X4* – надійність системи.

X1: Відображає швидкодію операцій залежно від завантаженності системи.

X2: Відображає ступінь складності взаємодії користувача зі системою.

X3: Показує розмір програмного коду який необхідно створити безпосередньо розробнику.

X4: Характеризує здатність системи підтримувати працездатність у критичних ситуаціях.

5.3.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у табл. 5.2.

Таблиця 5.2 – Основні параметри ПП

Назва Параметра	**Умовні позначення**	**Одиниці виміру**	**Значення параметра**		
			гірші	**середні**	**кращі**
Швидкодія інформаційної системи	X1	Оп/мс	2000	11000	19000
Простота використання програмного забезпечення	X2	Секунди на розгортання систем	600	300	60
Кількість програмного коду	X3	Кількість строк коду	2000	1500	1000
Надійність системи	X4	Сер. час до відмови	10	35	70

За даними таблиці 5.2 будуються графічні характеристики параметрів – рис. 5.2 – рис. 5.5.

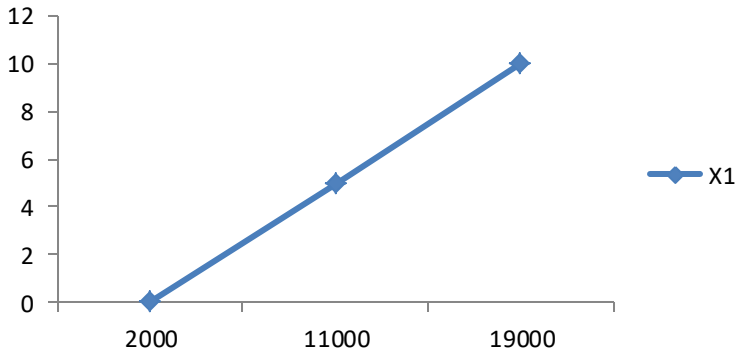

Рисунок 5.2 – X1, швидкодія системи

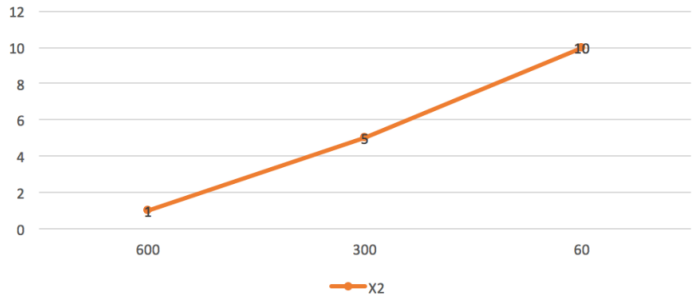

Рисунок 5.3 – X2, простота використання програмного забезпечення

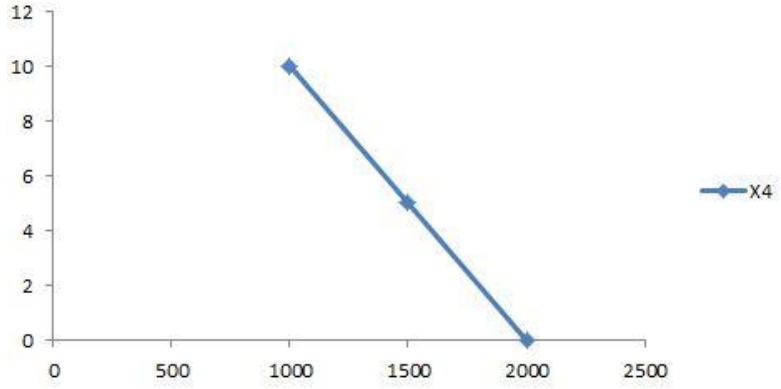

Рисунок 5.4 – X3, кількість програмного коду

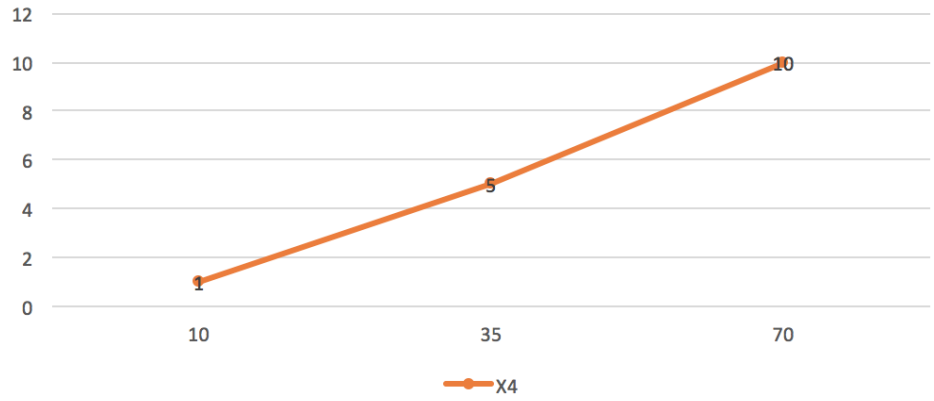

Рисунок 5.5 – X4, надійність системи

5.3.3 Кількісна оцінка параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

– визначення рівня значимості параметра шляхом присвоєння різних рангів;

– перевірку придатності експертних оцінок для подальшого використання;

– визначення оцінки попарного пріоритету параметрів;

– обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 5.3.

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 105 \qquad (5.1)$$

де N – число експертів, n – кількість параметрів;

Таблиця 5.3 – Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія системи	Оп/мс	4	3	4	4	4	4	4	27	0.75	0.56
X2	Простота використання програмного забезпечення	Секунда	4	4	4	3	4	3	3	25	-1.25	1.56
X3	Кількість програмного коду	кількість строк коду	2	2	1	2	1	2	2	12	-14.25	203.06
X4	Надійність системи	Сер. час до відмови	5	6	6	6	6	6	6	41	14.75	217.56
	Разом		15	15	15	15	15	15	15	105	0	420.75

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 26.25 \tag{5.2}$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T \tag{5.3}$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 420.75 \tag{5.4}$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 420.75}{7^2(5^3 - 5)} = 1{,}03 > W_k = 0.67 \tag{5.5}$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0.67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 5.4.

Таблиця 5.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	=	>	=	<	=	<	<	<	0.5
X1 і X3	<	<	<	<	<	<	<	<	0.5
X1 і X4	>	>	>	>	>	>	>	>	1.5
X2 і X3	<	<	<	<	<	<	<	<	0.5
X2 і X4	>	>	>	>	>	>	>	>	1.5
X3 і X4	>	>	>	>	>	>	>	>	1.5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases} \quad (5.6)$$

З отриманих числових оцінок переваги складемо матрицю A=$\| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами:

$$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i}, \quad (5.7)$$

де b_i розраховується за настурною формулою:

$$b_i = \sum_{i=1}^{N} a_{ij}. \quad (5.8)$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%).На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{ві}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \quad (5.9)$$

де b_i' розраховується за наступною формулою

$$b_i' = \sum_{i=1}^{N} a_{ij} b_j. \quad (5.10)$$

Як видно з таблиці 5.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 5.5 – Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1.0	0.5	0.5	1.5	3.5	0.219	22.25	0.216	100	0.215
X2	1.5	1.0	0.5	1.5	4.5	0.281	27.25	0.282	124.25	0.283
X3	1.5	1.5	1.0	1.5	5.5	0.344	34.25	0.347	156	0.348
X4	0.5	0.5	0.5	1.0	2.5	0.156	14.25	0.155	64.75	0.154
Всього:					16	1	98	1	445	1

5.4 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X1* (швидкодія системи) та *X2* (простота використання програмного забезпечення) відповідають технічним вимогам умов функціонування даного ПП.1

Варіант б) більш простий для реалізації: при його використанні потрібно 1150 рядків коду (X3), у той час як варіант а) вимагає 1700 рядків. За умови вибору варіанту а) реалізація прогнозуючої моделі займе більший час: швидкість написання коду (X4) становитиме 30 строк/година, тоді як варіант б) забезпечує швидкість на рівні 60 строк/година.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 5.6):

$$K_K(j) = \sum_{i=1}^{n} K_{\text{в}i,j} B_{i,j}, \tag{5.11}$$

де n – кількість параметрів; $K_{вi}$– коефіцієнт вагомості i–го параметра;B_i – оцінка i–го параметра в балах.

Таблиця 5.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Абсолютне значення параметра	Бальна Оцінка Параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1(X1)	Б	11000	5	0,215	1,075
F3(X4)	А	30	4	0,348	1,392
	Б	60	8		2,784
F3(X3)	Б	1150	8,5	0,154	1,309
	А	1700	3		0,462
F2(X2)	А	160	7,9	0,283	2,2357

За даними з таблиці 5.6 за формулою

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}],$$

визначаємо рівень якості кожного з варіантів:

$K_{K1} = 1{,}075 + 1{,}392 + 0{,}462 + 2{,}2357 = 5{,}1647$

$K_{K2} = 1{,}075 + 2{,}784 + 1{,}309 + 2{,}2357 = 7{,}4037$

Як видно з розрахунків, кращим є другий варіант, для якого коефіцієнт технічного рівня має найбільше значення.

визначаємо рівень якості кожного з варіантів:

$K_{K1} = 1{,}075 + 1{,}392 + 0{,}462 + 2{,}2357 = 5{,}1647$

$K_{K2} = 1{,}075 + 2{,}784 + 1{,}309 + 2{,}2357 = 7{,}4037$

Як видно з розрахунків, кращим є другий варіант, для якого коефіцієнт технічного рівня має найбільше значення.

5.5 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної реалізації системи;

При реалізації варіанту а) з'являється додаткова задача до завдання 2:

3. Реалізація алгоритмів синхронізації взаємодії мікросервісів.

Варіант б) вимагає виконання такої підзадачі у контексті виконання задачі 2:

4. Дослідження програмних бібліотек.

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 2.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється як

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}, \quad (5.12)$$

де T_P – трудомісткість розробки ПП;$K_П$ – поправочний коефіцієнт;$K_{СК}$ – коефіцієнт на складність вхідної інформації; K_M – коефіцієнт рівня мови програмування;$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: T_P =90 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для

всіх чотирьох завдань рівний 1: $К_{СК} = 1$. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $К_{СТ} = 0.8$. Тоді, за формулою 5.1, загальна трудомісткість програмування першого завдання дорівнює:

$Т_1 = 90*1.7*0.8 = 122.4$ людино-днів.

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм 2-ї групи складності, степінь новизни Б), тобто $Т_Р = 27$ людино-днів, $К_П = 1.08$, $К_{СК} = 1$,$К_{СТ} = 0.8$:

$Т_2 = 27 * 1.08 * 0.8 = 23.328$ людино-днів.

Для третього завдання (використовується алгоритм 1-ї групи складності, степінь новизни В), тобто $Т_Р = 43$ людино-дні, $К_П = 0,81$, $К_{СК} = 1$,$К_{СТ} = 0.8$:

$Т_2 = 43 * 0.81 * 0.8 = 27.864$ людино-днів.

Для четвертого завдання (використовується алгоритм 3-ї групи складності, степінь новизни В), тобто $Т_Р = 12$ людино-днів, $К_П = 0,60$, $К_{СК} = 1$,$К_{СТ} = 0.8$:

$Т_2 = 12 * 0.60 * 0.8 = 5.76$ людино-днів.

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$Т_I = (122.4 + 23.328 + 27.864) \cdot 8 = 1388.74$ людино-годин;

$Т_{II} = (122.4 + 23.328 + 5.76) \cdot 8 = 1211.904$ людино-годин;

Найвищу трудомісткість має варіант I.

В розробці беруть участь два програмісти з окладом 8000 грн., один архітектор програмного забезпечення з окладом 10000грн. Визначимо зарплату за годину за формулою:

$$СЧ = \frac{М}{T_m \cdot t} \text{ грн.}, \quad (5.12)$$

де М – місячний оклад працівників;T_m – кількість робочих днів у місяць; t – кількість робочих годин в день.

$$С_ч = \frac{8000 + 8000 + 10000}{3 \cdot 21 \cdot 8} = 51,59 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою

$$СЗП = С_ч \cdot Т_i \cdot КД, \quad (5.13)$$

де $С_ч$– величина погодинної оплати праці програміста;$Т_i$ – трудомісткість відповідного завдання; $К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 51,59 \cdot 1388.74 \cdot 1.2 = 85974,12$ грн.

II. $С_{ЗП} = 51,59 \cdot 1211.904 \cdot 1.2 = 75026,55$ грн.

Відрахування на єдиний соціальний внесок в залежності від групи професійного ризику (II клас) становить 36,77%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.3677 = 85974,12 \cdot 0.3677 = 31612$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.3677 = 75026,55 \cdot 0.3677 = 27587$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 8000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 8000 \cdot 0,2 = 19200$ грн.

З урахуванням додаткової заробітної плати:

$С_{ЗП} = С_Г \cdot (1 + К_З) = 19200 \cdot (1 + 0.2) = 23040$ грн.

Відрахування на єдиний соціальний внесок:

$С_{ВІД} = С_{ЗП} \cdot 0.22 = 23040 \cdot 0.22 = 5068.8$ грн.

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 8000 грн.

$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 8000 = 2300$ грн.,

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача; $К_А$– річна норма амортизації; $Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 8000 \cdot 0.05 = 460$ грн.,

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 8 – 16) \cdot 8 \cdot 0.9 = 1706.4$ годин,

де $Д_К$ – календарна кількість днів у році; $Д_В$, $Д_С$ – відповідно кількість

вихідних та святкових днів; $Д_Р$ – кількість днів планових ремонтів устаткування; t –кількість робочих годин в день; $К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1706,4 \cdot 0,156 \cdot 0,2 \cdot 1,94177 = 103.379$грн.,

де $N_С$ – середньо-споживча потужність приладу; $К_З$– коефіцієнтом зайнятості приладу; $Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$С_Н = Ц_{ПР} \cdot 0.67 = 8000 \cdot 0,67 = 5360$ грн.

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н$$

$С_{ЕКС} = 23040 + 5068.8 + 2300 + 460 + 103.379 + 5360 = 36332.179$ грн.

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 36332.179 / 1706,4 = 21.29$ грн/час.

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$С_М = С_{М\text{-}Г} \cdot Т$$

I. $С_М = 21.29 \cdot 1388.74 = 29566.28$ грн.;

II. $С_М = 21.29 \cdot 1211.904 = 25801.44$ грн.;

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} \cdot 0,67$$

I. $С_Н = 85974,12 \cdot 0,67 = 57602.66$ грн.;

II. $С_Н = 75026,55 \cdot 0,67 = 50267.79$ грн.;

Отже, вартість розробки ПП за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{ВІД} + С_М + С_Н$$

I. $С_{ПП} = 85974.12 + 18914.31 + 29566.28 + 57602.66 = 192057.37$грн.;

II. $С_{ПП} = 75026.55 + 16505.84 + 25801.44 + 50267.79 = 167601.62$грн.;

5.6 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕРj} = К_{Кj} / С_{Фj}, \quad (5.14)$$

$К_{ТЕР1} = 5.1647 / 215811.09 = 2.39 \cdot 10^{-5}$;

$К_{ТЕР2} = 7.4037 / 188365.89 = 3.93 \cdot 10^{-5}$;

Як бачимо, найбільш ефективним є другий варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 3.93 \cdot 10^{-5}$.

5.7 Висновки

В даному розділі проведено повний функціонально-вартісний аналіз ПП, який було розроблено в рамках дипломного проекту. Процес аналізу можна умовно розділити на дві частини.

В першій з них проведено дослідження ПП з технічної точки зору: було визначено основні функції ПП та сформовано множину варіантів їх реалізації; на основі обчислених значень параметрів, а також експертних оцінок їх важливості було обчислено коефіцієнт технічного рівня, який і дав змогу визначити оптимальну з технічної точки зору альтернативу реалізації функцій ПП.

Другу частину ФВА присвячено вибору із альтернативних варіантів реалізації найбільш економічно обґрунтованого. Порівняння запропонованих варіантів реалізації в рамках даної частини виконувалось за коефіцієнтом ефективності, для обчислення якого були обчислені такі допоміжні параметри, як трудомісткість, витрати на заробітну плату, накладні витрати.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є другий варіант реалізації програмного продукту.

У нього виявився найкращий показник техніко-економічного рівня якості. $К_{ТЕР} = 3.93 \cdot 10^{-5}$.

Цей варіант реалізації програмного продукту має такі параметри:

– інструмент для розгортання – Docker.

– сервіс відкриття – Netflix Eureka;

– взаємодія через REST API

ВИСНОВКИ

В даній дипломній роботі було досліджено основні концепції побудови додатків на базі мікросервісної архітектури. Розглянуто основні особливості, переваги та недоліки даного підходу до побудови програмних систем, порівняно з класичним монолітним рішенням. Монолітна архітектура дуже добре розв'язує свої задачі, але із зростанням складності вона вже не може якісно вирішувати свої функції, тому розвинулися сервіс-орієнтовані архітектури, прикладом якої є мікросервісна архітектура.

В даному проекті було детально розглянуто основні принципи проектування та розгортання даних систем, основні шаблони інтеграції мікросервісів, технології комунікації. В тестовому прикладі було продемонстровано працездатність вищенаведених концепцій.

Також було розглянуто ключові компоненти для побудови мікросервісних систем, які утворюють інфраструктурний рівень та надають необхідну гнучкість всій системі. До даних компонентів відносяться: сервіс єдиного входу, сервіс відкриття, балансувальних навантаження, автоматичний вимикач. Дані шаблони було перевірено на існуючих відкритих реалізаціях.

Однією з головних переваг мікросервісної архітектури є її гетерогенність, тому було досліджено базові бібліотеки для створення незалежних сервісних додатків на різних мовах програмування та програмних платформах. Значну роль у розвитку та створенні програми є підтримка необхідного середовища, що забезпечується системами віртуалізації та постійної інтеграції. Було розглянуто систему Docker для забезпечення відповідного програмного середовища, а також проаналізовано основні підходи до налаштування процесів постійної інтеграції. Також було наведено основні питання які стосуються проблем безпеки міжсервісного спілкування в системі, а також підходів до масштабування додатку.

Як результат всіх проведених досліджень, було розроблено тестову програму, яка підтверджує концепцію даної архітектури та демонструє її працездатність. Для реалізації даної системи було використано дві різні мови

програмування – Java та Python, а також інструменти для забезпечення інтеграції та внутрішні бібліотеки, які надають змогу розгортати мережеві розподілені додатки.

Окрім цього, було проведено повний функціонально-вартісний аналіз ПП, який було розроблено в рамках дипломної роботи.

Підсумовуючи, можна стверджувати, що мікросервісна архітектура має право на існування, але не претендує на звання «срібної кулі» для побудови інформаційних систем. Більше того, мікросервіси не рекомендовано створювати без глибокого попереднього аналізу предметної області та чіткого виділення обмежених контекстів, а також пропонується створювати мікросервіси на базі існуючого моноліту.

Дана архітектура є молодою та перспективною у сучасному проектуванні інформаційних систем та, цілком можливо, її використання набуде масовий характер.

ПЕРЕЛІК ПОСИЛАНЬ

1. М. Фаулер. Архитектура корпоративных программных приложений / М. Фаулер. – Издательский дом Вильямс, 2006 – 544 с.
2. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.: Питер, 2016 – 304 с.
3. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.
4. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74 p.
5. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans – Addison-Wesley, 2003 – 560 p.
6. Martin Fowler – Microservices – Режим доступу: http://martinfowler.com/articles/microservices.html – Дата доступу: 28.05.2017
7. I. Nadareishvili. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O'Reilly Media, 2016 – 146 p.
8. Introduction to microservices. – Режим доступу: https://nginx.com/blog/introduction-to-microservices/ – Дата доступу: 29.05.2017
9. Using an API Gateway. – Режим доступу: https://nginx.com/blog/building-microservices-using-an-api-gateway/ – Дата доступу: 27.05.2017
10. Service Discovery. – Режим доступу: https://nginx.com/blog/service-discovery-in-a-microservices-architecture/ – Дата доступу: 27.05.2017
11. Офіційний сайт Docker. – Режим доступу: https://docker.com/ – Дата доступу: 27.05.2017
12. Офіційний сайт C++ Micro Services. – Режим доступу: http://cppmicroservices.org/ – Дата доступу: 29.05.2017
13. Офіційний сайт Pistache framework. – Режим доступу: http://pistache.io/ – Дата доступу: 29.05.2017
14. Офіційний сайт Spring framework. – Режим доступу: https://spring.io – Дата доступу: 29.05.2017

15. Офіційний сайт Spark framework. – Режим доступу: https://sparkjava.com/ – Дата доступу: 29.05.2017
16. Офіційний сайт Restlet. – Режим доступу: https://restlet.com/ – Дата доступу: 29.05.2017
17. Офіційний сайт Flask. – Режим доступу: http://flask.pocoo.org/ – Дата доступу: 29.05.2017
18. Офіційний сайт Tornado. – Режим доступу: http://tornadoweb.org/ – Дата доступу: 29.05.2017
19. Офіційний сайт Nameko. – Режим доступу: https://nameko.readthedocs.io – Дата доступу: 29.05.2017