Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження засобів і технологій міжсервісного зв’язку
в мікросервісній архітектурі програмних систем
(тема)

Виконав: студент 2 курсу, групи ІПЗм-17-2
спеціальності 121-Інженерія програмного забезпечення
(код і повна назва спеціальності)
Освітньо-наукової програми Інженерія програмного забезпечення

Синкевич М.Е.
(прізвище, ініціали)
Керівник проф. Лєсна Н.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________________ З.В.Дудар

2019 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121–Інженерія програмного забезпечення
(код і повна назва)

Освітньо-наукова програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«_____»________________ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові Синкевич Максиму Едуардовичу
(прізвище, ім'я, по батькові)

1.Тема роботи Дослідження засобів і технологій міжсервісного зв'язку в мікросервісній архітектурі програмних систем затверджена наказом по університету від «18» квітня 2019р №546Ст

2. Термін подання студентом роботи до екзаменаційної комісії «18» 06 2019 р.

3. Вихідні дані до роботи *засоби і технології міжсервісної комунікації, OS Windows, об'єктно-орієнтована мова програмування Kotlin, середовище розробки IntelliJ IDEA.*

4. Перелік питань, що потрібно опрацювати в роботі *аналіз предметної галузі і постановка задачі, засоби і технології міжсервісного зв'язку, опис програмної системи дослідження продуктивності технологій міжсервісного зв'язку в мікросервісній архітектурі, оцінювання ефективності міжсервісного зв'язку з використанням обраних технологій.*

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій *причини виникнення проблем комунікації в мікросервісній архітектурі програмних систем, формат сериалізації даних, критерії оцінювання ефективності, засоби комунікації, структура програмної системи дослідження продуктивності, обрані засоби та технології, структура таблиць, які містять тестові дані, методика оцінювання ефективності, задовільнення технологій обраним критеріям, результати моделювання, рекомендації по використанню.*

6 Консультанти розділів роботи

Найменування Розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Снецчастина	проф. Лєсна Н.С.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	Аналіз предметної галузі	22 квітня 2019р.	
2.	Опрацювання першоджерел та постановка задачі	30 квірня2019р.	
3.	Розробка методики та моделі	6 травня 2019р.	
4.	Оформлення результатів дослідження	20 травня2019р.	
5.	Підготовка пояснювальної записки	27 травня 2019р.	
6.	Підготовка презентації та доповіді	1 червня 2019р.	
7.	Перевірка роботи на антиплагіат	05 червня 2019р.	
8.	Нормоконтроль	11 червня 2019р.	
9.	Рецензування	12 червня 2019р.	
10.	Занесення атестаційної роботи в електронний архів	13 червня 2019р.	
11.	Попередній захист	14 червня 2019р.	
12.	Допуск до захисту у зав. кафедри	14 червня 2019р.	

Дата видачі завдання 22 квітня 2019 р.

Студент ____________________________________
(підпис)

Керівник роботи ________________________ __проф. Лєсна Н.С.__________
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 97 с., 36 рис., 3 табл., 4 додатки, 22 джерел.

АРХІТЕКТУРА, МІКРОСЕРВІСИ, КОМУНІКАЦІЯ, ПРОДУКТИВНІСТЬ, REST, GRAPHQL, KOTLIN

Об’єктом дослідження є процес міжсервісного зв’язку в мікросервісній архітектурі програмних систем.

Метою роботи є виявлення найбільш ефективних засобів і технологій міжсервісного зв’язку в мікросервісній архітектурі програмних систем.

У результаті роботи розроблена методика оцінювання ефективності засобів і технологій міжсервісного зв’язку. Розроблена програмна система моделювання продуктивності обраних технологій міжсервісної комунікації.

ARCHITECTURE, MICROSERVICES, COMMUNICATION, PRODUCTIVITY, REST, GRAPHQL, KOTLIN

The object of the research is the process of inter-service communication in the micro-service architecture of software systems.

The purpose of the work is to identify the most effective means and technologies of inter-service communication in the micro-service architecture of software systems.

As a result of work the method of estimation of efficiency of means and technologies of inter-service communication is developed. The software system for modeling the productivity of selected inter-service communication technologies has been developed.

ЗМІСТ

Вступ 5
1 Аналіз предметної галузі і постановка задачі 8
1.1 Архітектури програмного забезпечення. Переваги і недоліки 8
1.2 Мікросервісна архітектура програмних систем 15
1.3 Проблеми процесу міжсервісної комунікації 20
1.4 Постановка задачі 24
2 Засоби і технології міжсервісного зв'язку 26
2.1 Аналіз критеріїв вибору технології реалізації міжсервісного зв’язку 26
2.2 Формат серіалізації даних 30
2.3 Дослідження засобів і технологій міжсервісного зв’язку 32
3 Опис програмної системи дослідження продуктивності технологій міжсервісного зв'язку в мікросервісній архітектурі 37
3.1 Дизайн програмної системи 37
3.2 Опис обраних технологій 40
3.3 Програмна реалізація 47
3.4 Тестування програмної системи 53
4 Оцінювання ефективності міжсервісного зв'язку з використанням обраних технологій 62
4.1 Методика оцінювання ефективності 62
4.2 Моделювання продуктивності технологій міжсервісного зв’язку 64
4.3 Рекомендації по використанню технологій міжсервісного зв’язку 69
Висновки 71
Перелік посилань 72
Додаток А Слайди презентації 75
Додаток Б Лістинг коду програми 86

Додаток В Наукові публікації 88
Додаток Г Електронні матеріали 96

ВСТУП

У сучасному світі надзвичайно важливою є проблема створення та проектування якісного програмного забезпечення. З розвитком індустрії інформаційних технологій визначено багато різноманітних концепцій та підходів до побудови масштабних програмних систем. Показником коректно побудованої програмної системи є її архітектура, яка правильно описує предметну галузь 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 та доцільне відображення предметної галузі. Провідне місце довгий час посідала так звана «монолітна архітектура». При цьому підході система цілком являє собою моноліт, що фізично розташовується на одному сервері, запускається в єдиному процесі та виконує всі операції системи.

Монолітний додаток завдяки архітектурі піддається тільки горизонтальному масштабуванню, тобто запуску декількох окремих серверів з монолітом на них. З плином часу з’являлися нові ідеї та підходи і саме таким стала сервісно-орієнтована архітектура (Service-oriented architecture — SOA). У противагу монолітній системі, при SOA вся програма являє собою розподілену систему, що обмінюється повідомленнями за певним визначеним протоколом. Уся система складається з набору незалежних сервісів, які фокусуються на конкретній власній задачі. SOA має на меті боротьбу з великими монолітними системами. Основні слабкі місця відносяться до протоколів обміну даними, наприклад, таких як SOAP, а також до нераціонального розподілу системи на сервіси. Пізніше було запропоновано іще один підхід до організації SOA — мікросервісна архітектура (Microservices architecture — MSA). MSA можна вважати підмножиною SOA, але все ж таки мікросервісна архітектура відрізняється від класичної сервісно-орієнтованої. Основна відмінність — це

невеликий об'єм кодової бази для кожного сервісу, в той час як в SOA об'єм кодової бази не є важливим. Також вагомим фактором для MSA є наявність власного обмеженого контексту предметної галузі у кожного з сервісів. Обмеження на кількість сервісів відсутні, але кожен з них має працювати лише над однією задачею предметної галузі. Для обміну інформацією у мікросервісному підході використовуються стандартизовані протоколи передачі даних (такі як HTTP). Кожен мікросервіс має власний API для спілкування з іншими мікросервісами. Кожен з мікросервісів може бути написаний на будь-якій мові програмування незалежно від інших сервісів, використовуючи будь-які бібліотеки. Такий підхід має значні переваги у порівнянні з монолітними застосунками, такі, наприклад, як краща масштабованість, менша зв'язаність між модулями, кращий контроль на етапах розробки, тестування та розгортування. Незважаючи на переваги архітектурного стилю, він не позбавлений недоліків. Серед основних проблем при впровадженні можна відмітити наступні: якщо у модулях, що виконують декілька функцій, взаємодія відбувається локально, то мікросервісна архітектура накладає вимогу атомізації модулів та вимогу взаємодії за допомогою мережі.

Існує два основних шаблони обміну повідомленнями, які мікросервіси можуть використовувати для зв'язку з іншими мікросервісами. При синхронному спілкуванні сервіс викликає API, який надається іншим сервісом, використовуючи протокол, такий як HTTP або gRPC. Цей варіант є синхронною схемою обміну повідомленнями, так як сторона, що викликає, очікує відповіді від отримувача. При асинхронному спілкуванні сервіс відправляє повідомлення без очікування відповіді, та один або декілька різних сервісів обробляють повідомлення асинхронно.

Мікросервісна архітектура передбачає, що, швидше за все, сервіси будуть доступні для декількох різних клієнтів, що працюють на різних платформах. Тому необхідно використовувати протоколи зв'язку або технології, які можуть використовувати як масштабні десктопні програми, так і мобільні додатки. В зв'язку з цим, важливо виявити, які засоби і технології підходять якнайкраще для

використання відповідно до їх продуктивності та кросплатформової функціональності.

Актуальність теми атестаційної роботи обумовлена наявністю активної комунікації між сервісами у програмних системах з мікросервісною архітектурою, яка накладає часові затримки при транспортуванні та обробці повідомлень, що спричиняє пряму залежність між продуктивністю системи та ефективністю технології міжсервісного зв’язку.

Метою роботи є виявлення найбільш ефективних засобів і технологій міжсервісного зв’язку в мікросервісній архітектурі програмних систем.

Об’єктом дослідження є процес міжсервісного зв’язку в мікросервісній архітектурі програмних систем.

Пердметом дослідження є засоби і технології міжсервісного зв’язку в мікросервісній архітектурі програмних систем.

В ході даної роботи задля дослідження ефективності використаних програмних засобів і технологій застосовувались емпіричні методи програмної інженерії, загальнолоігчні методи наукового пізнання, а також порівняльний метод.

Наукова новизна. Запропонована методика оцінювання ефективності технологій міжсервісного зв’язку в мікросервісній архітектурі програмних систем.

Практичне значення. Результати дослідження продуктивності та вироблені рекомендації щодо вибору засобу міжсервісної комунікацій дозволять найбільш ефективно організувати міжсервісний зв’язок, що сприятиме підвищенню продуктивності системи в цілому.

Публікації. Результати дослідження, проведеного в атестаційній роботі, пройшли апробацію в рамках двадцять третього міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті», опубліковано тези доповіді. Також результати досліджень опубліковані в статті «Дослідження засобів і технологій міжсервісного зв’язку в мікросервісній архітектурі» міжнародного наукового електроного журналу «Наука онлайн».

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ І ПОСТАНОВКА ЗАДАЧІ

1.1 Архітектури програмного забезпечення. Переваги і недоліки

Проектування архітектури інформаційної системи (ІС) є одним із найважливіших етапів створення проекту. Перш за все встановимо, що мається на увазі під поняттям «архітектура інформаційної системи». Існує безліч трактувань даного терміну, можна описати архітектуру ІС як організацію системи через набір компонент, їх взаємовідносини та зв'язок зі зовнішнім середовищем. Під час проектування системи, як правило, розглядаються архітектурні рішення, після прийняття яких зміна поведінки ІС ускладнюється.

Основною ціллю програмної архітектури є боротьба з потенційною складністю, яка властива масштабним програмним системам, корпоративним додаткам. Як правило, складність зростає значно швидше ніж обсяг програмного коду, тому якщо заздалегідь не передбачити якісну організацію майбутнього вихідного коду, то цілком ймовірно настане момент, коли підтримувати дану складність буде неможливо. Вдало розроблена архітектура заощадить багато часу та зусиль. Гарно спроектована архітектурна модель має бути гнучкою в місцях, які потенційно мають найчастіше змінюватися чи розширюватись, та жорсткою в інших. Також правильно побудовану програму легко супроводжувати, тестувати та підтримувати. Можна виділити ключові моменти, які описують правильно побудовану архітектуру [1]:

– ефективність та працездатність системи: перш за все, інформаційна система 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 галузі та проектуванні моделі завжди необхідно оцінювати та знаходити місця, які потенційно будуть

змінюватися, для того, 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 загальної структури та поведінки системи, при тому щоб на додавання нових функцій витрачалась найменш можлива кількість часу;

– продуктивність: швидкодія та високонавантаженість є однією з ключових вимог для більшості інформаційних систем. Програма повинна витримувати належне навантаження зі сторони користувачів та відповідати за допустимий проміжок часу. Одним із варіантів рішення є здатність до швидкого і ефективного масштабування системи;

– здатність до тестування: добре протестований вихідний 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 програмного забезпечення 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: програмну систему 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 програмну систему важко протестувати, оскільки всі компоненти сильно зв’язані між собою.

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

З розвитком інформаційних технологій вимоги щодо доступності та потужності до інформаційних систем, як до існуючих, так і для розроблюваних, неперервно зростали. Для того щоб система відповідала таким неперервно зростаючим вимогам відбувалося підвищення обчислювальних потужностей, застосовувалось так зване «вертикальне масштабування» — збільшення продуктивності компонентів системи з метою підвищення загальної продуктивності. Масштабованість за своїм визначенням є властивістю системи обробляти зростаючий об'єм роботи шляхом додавання ресурсів до системи.

Масштабованість у контексті збільшення продуктивності означає можливість автоматично замінювати в існуючій системі компоненти більш потужними і швидкими протягом зростання вимог і розвитку технологій. Це найпростіший спосіб масштабування, так як він не вимагає ніяких змін в прикладних програмах, що працюють на таких системах, адже необхідно замінювати лише середовище розгортання.

Однак ефективність вертикального масштабування виявилась достатньо обмеженою, так як приріст обчислювальних потужностей конкретного серверу не давав необхідного приросту потужності інформаційної системи, що призвело до виникнення горизонтального масштабування — розбиття системи на більш дрібні структурні компоненти та рознесення їх по окремим фізичним машинам (або їх групам), і збільшення кількості серверів, паралельно виконуючих одну і ту ж функцію. Масштабованість в цьому контексті означає можливість додавати до системи нові вузли, сервери, процесори для збільшення загальної продуктивності програмної системи. Цей спосіб масштабування може вимагати внесення змін до програм, щоб програми могли повною мірою використовувати дедалі більшу кількість доступних ресурсів.

Загалом існує три стратегії масштабування: по осі X — горизонтальна дублікація або масштабування клонуванням, по осі Y — функціональна декомпозиція та по осі Z — розділення даних. Ці стратегії утворюють між собою так званий «куб

масштабування», що являє собою модель нескінченного масштабування шляхом впровадження кожної з перелічених осей (див. рис. 1.1).

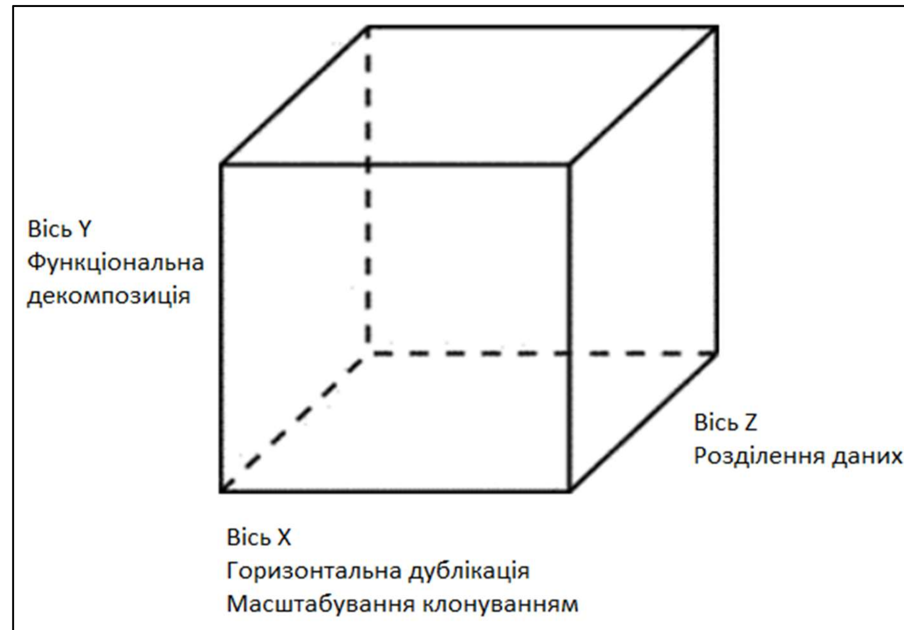

Рисунок 1.1 — «Куб масштабування»

Масштабування по осі X — це класична стратегія горизонтального масштабування, яка використовує балансування навантаження. Це простий і ефективний спосіб масштабування для багатьох типів додатків. Даний підхід перевірений часом і простий в реалізації, він працює навіть в тих випадках, коли при розробці програми спочатку не малося на увазі його масштабувати. Монолітні додатки в основному масштабуються за допомогою цього підходу, оскільки під час їх розробки не враховувалися інші стратегії, зо призвело до сильної зв’язності компонентів. Також залежність від стану сесії користувачів унеможливлює використання інших моделей.

Модель масштабування по осі X може бути просто реалізована за допомогою рівномірного балансування навантаження. Таке балансування може бути виконане на 4-му рівні моделі OSI (Open Systems Interconnection — абстрактна мережева модель для комунікацій і розробки мережевих протоколів, яка представляє рівневий підхід до

мережі, де кожен рівень обслуговує свою частину процесу взаємодії) — TCP, якщо стан не важливий. Але найчастіше потрібне балансування, яке працює на 7 рівні OSI — HTTP, для перевірки http-заголовків або інших складових запиту, для того щоб перенаправити його в потрібний додаток (див. рис. 1.2).

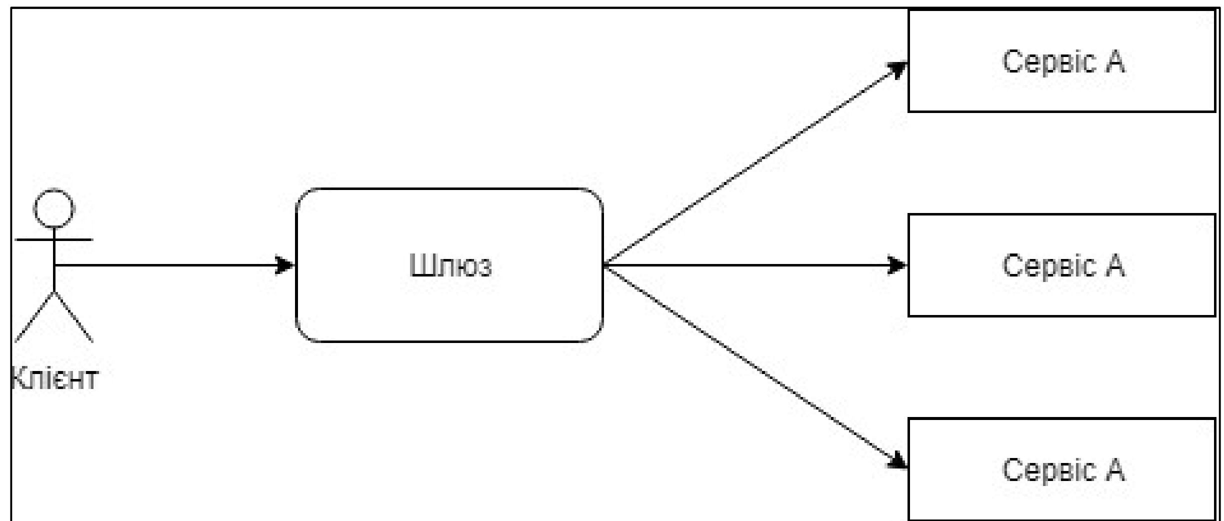

Рисунок 1.2 — Модель масштабування по осі X

Одним з недоліків цього підходу є те, що оскільки кожна копія потенційно отримує доступ до всіх даних, кеш вимагає більшого обсягу пам'яті для ефективності. Іншою проблемою цього підходу є те, що він не вирішує проблеми підвищення складності та розвитку.

Масштабування по осі Y являє собою декомпозицію додатку на різні сервіси. Це найбільш підходящий спосіб масштабування для сервісів, які групують свою функціональність. Дана модель реалізується з використанням механізму (зазвичай із застосуванням балансування навантаження на 7 рівні моделі OSI), який переглядає URL або заголовки запиту і перенаправляє запит в потрібний сервіс (див. рис. 1.3). Кожний сервіс відповідає за одну або декілька близьких функцій. Є кілька різних способів декомпозиції застосунку на сервіси. Один з підходів полягає у використанні

декомпозиції на основі дієслова та визначення сервісів, які реалізують один випадок використання, наприклад, виконання замовлення.

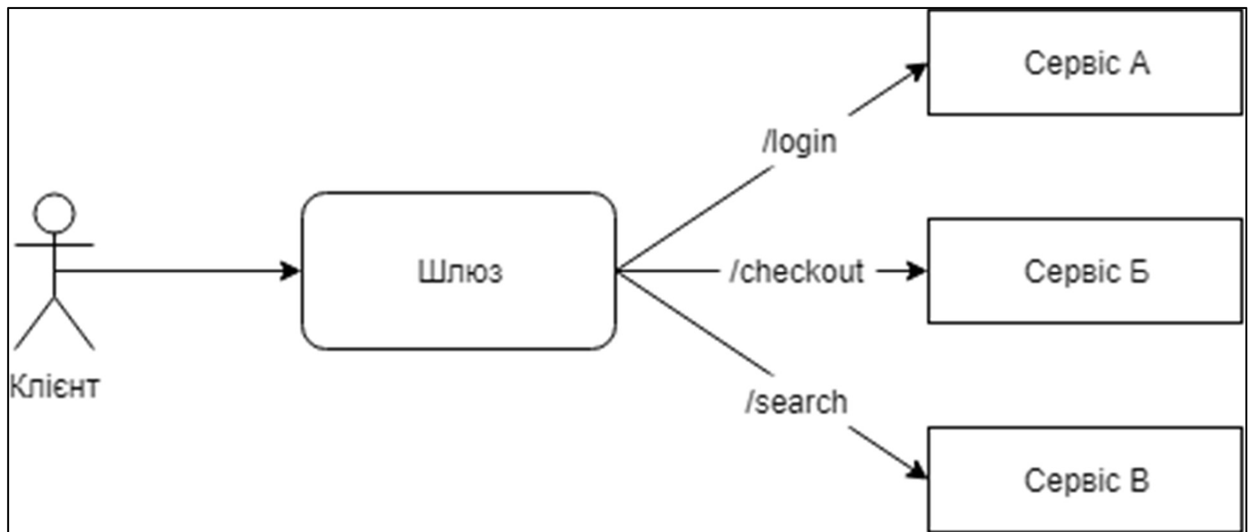

Рисунок 1.3 — Модель масштабування по осі Y

Інший варіант полягає в тому, щоб розкласти додаток на іменник і створити служби, що відповідають за всі операції, пов'язані з конкретним об'єктом, наприклад, управління клієнтами. Програма може використовувати комбінацію розбиття на основі дієслова та на основі іменника. Хоча даний шаблон можна використовувати в монолітних додатках, в яких є можливість розділити внутрішню функціональність на різні сервіси за допомогою розбору і аналізу даних в URL або в заголовках, робити це потрібно обережно і стежити за станом (дані сесії в пам'яті). Наприклад, не можна перенаправити запит до сервісу B для оплати якщо він ссилається на дані сесії, які зберігаються в сервісі A або C.

При виконанні масштабування по осі Z кожен сервер виконує ідентичну копію коду. У цьому відношенні воно подібне до масштабування по осі X. Велика різниця полягає в тому, що кожен сервер відповідає лише за підмножину даних. Деякий компонент системи відповідає за маршрутизацію кожного запиту на відповідний сервер. Одним з часто використовуваних критеріїв маршрутизації є атрибут запиту,

такий як первинний ключ об'єкта, до якого здійснюється доступ. Іншим поширеним критерієм маршрутизації є тип клієнта. Наприклад, програмна система може надавати одним клієнтам більш високий рівень обслуговування, ніж іншим, шляхом перенаправлення їх запитів до іншого набору серверів з більшою ємністю. Розбиття осі Z зазвичай використовується для масштабування баз даних. Дані розділяються між нобором серверів на основі атрибутів кожного запису (див. рис. 1.4).

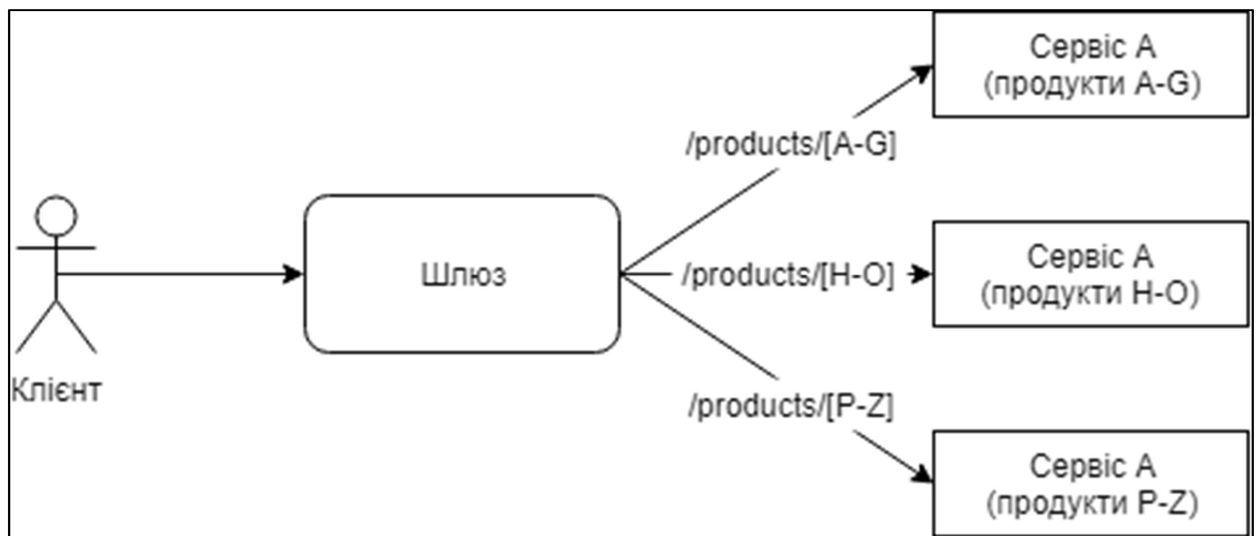

Рисунок 1.4 — Модель масштабування по осі Z

Масштабування по осі Z може також застосовуватися і до додатків. Таке масштабування має ряд переваг: кожен сервер має справу тільки з підмножиною даних, це покращує використання кешу і зменшує використання пам'яті та трафік вводу-виводу, це також покращує масштабованість транзакцій, оскільки запити зазвичай розподіляються на декількох серверах, крім того, масштабування по осі Z покращує ізоляцію несправності, оскільки збій тільки робить частину даних недоступною. Також таке масштабування не позбавлене недоліків: збільшення складності програми, адже потрібно реалізувати схему розділення, яка може бути нетривіальною, особливо якщо коли-небудь буде потрібно перерозподілити дані. Іншим недоліком масштабування по осі Z є те, що воно не вирішує проблеми

збільшення об’єму і складності розробки застосування.Тож наступним кроком у розвитку монолітної архітектури стала сервісно-орієнтована архітектура (SOA).

1.2 Мікросервісна архітектура програмних систем

Сервісно-орієнтована архітектура представляє собою ідею, що програма має складатися з набору сервісів, які взаємодіють один з іншим через стандартизовані протоколи та інтерфейси. В свою чергу до сервісів висуваються наступні вимоги:

– стандартизовані інтерфейси;

– слабка зв’язність;

– абстракція;

– перевикористання;

– автономність;

– 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 існують три основних елементи: постачальник сервісу, споживач сервісу та реєстр сервісів. Взаємодія між даними елементами виглядає наступним чином: постачальник сервісу реєструє свої сервіси, а споживач звертається до реєстру з запитом. Спілкування відбувається за певним уніфікованим протоколом передачі даних.

Ідеї сервісно-орієнтованої архітектури розвинув, так званий, мікросервісний підхід. В цілому, SOA представляє гарні рішення, але не існує чітких правил та настанов як саме досягти найбільшої ефективності при використанні сервіс-орієнтованій архітектурі. Найбільш вузьке місце в SOA — це складнощі, які пов’язані з протоколами обміну даних (такими як SOAP), відсутність методик, які дозволяють

оцінити та встановити ступінь деталізації сервісів, а також місця розподілу системи на сервіси. Архітектурний стиль мікросервісів (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 розповсюджені механізми, як правило HTTP. Описані сервіси будуються навколо бізнес потреб та реалізують свій окремий обмежений предметний контекст з визначеними границями. Всі сервіси можуть бути розроблені на різних мовах програмування, використовуючи будь-які фреймворки та засоби зберігання даних. Мікросервісну архітектуру вважають підмножиною SOA, тому що вона, аналогічно SOA, орієнтована на сервіси та розподілена, використовує унифіковані протоколи обміну даними. MSA слід розглядати як підхід до реалізації SOA. Сам стиль реалізації не є технічною інновацією, він перейняв основні принципи проектування, які були використані в Unix. 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, є оцінювати складність кількістю рядків коду, але даний підхід не є універсальним, 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. 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 дотримуватись при створенні будь-якого програмного забезпечення [2]. Також кожен мікросервіс має бути слабо зв’язаним та сильно зчепленим. Слабке зв’язування передбачає використання інтерфейсів та інструментів впровадження залежностей (Dependency Injection), 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. Для стандартного монолітного додатку зазвичай 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 компонента доступні для читання чи запису тільки через відповідний прикладний інтерфейс. 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. 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, що базується 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, що базується 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 використовують дані з декількох 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, є 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, можна локалізувати причину в рамках конкретного мікросервісу, не перезавантажуючи усю систему, а лише відновити роботу зламаного компоненту.

Стосовно програмних переваг можна виділити наступні: модульність — мікросервісна архітектура дозволяє зберігати модульність та інкапсуляцію, вона забезпечує логічний розподіл системи на модулі за рахунок явного фізичного розділу по серверам; фізична ізольованість захищає від порушення границь обмежених контекстів; також мікросервіси є легші для розуміння та підтримки, не потрібно вникати одразу в усі подробиці загальної системи [3]. Крім того, можна обирати для кожного сервісу необхідне апаратне забезпечення. Термін часу для запуску та впровадження є значно меншим, ніж для стандартного тришарового додатку. MSA дозволяє реалізувати процес безперервного розгортання. Також такий підхід надає можливість сервісам масштабуватися незалежно від системи в цілому. Можливо розгортати таку кількість екземплярів сервісу, яка задовольнить бізнеспотребу. Будь-яка локальна зміна в сервісі може бути легко виконана розробником і не потребує додаткового узгодження з командами, що розробляють інші сервіси. Як результат, це надає мікросервісам гнучкості в порівнянні з монолітними системами і дозволяє без перешкод впровадити процеси неперервної інтеграції та неперервної доставки.

З іншого боку мікросервісна архітектура не призначена для розв’язування всіх можливих задач та має властиві розподіленим системам недоліки. Найбільше труднощів виникає у питаннях взаємодії мікросервісів, їх інтеграції. Побудова мікросервісів є не тривіальною задачею, як і побудова будь-якої ефективної розподіленої системи, з цього випливають проблеми, які пов’язані з впровадженням взаємодії між процесами на основі обміну повідомленнями або викликами певних методів через RPC, при цьому треба розглядати безліч питань, які стосуються обробки мережевих збоїв та клієнтських помилок. До основних складнощів з якими можна зіштовхнутися при створенні мікросервісного додатку можна віднести: складність розробки — оскільки це розподілена система, то слід багато уваги приділяти 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. 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, в той час коли компоненти моноліту спілкуються в рамках єдиного процесу і не вимагають додаткових мережевих викликів. Тестування системи — інтеграційне та модульне тестування значно складніше, ніж у випадку монолітного додатку, але з появою нових програмних інструментів це питання може бути вирішене. Вартість моніторингу системи 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 він вимагає перенесення деякої логіки між сервісами. Вибір мікросервісної архітектури без попереднього аналізу може привести до безладного проектування та невдачі всього проекту.

1.3 Проблеми процесу міжсервісної комунікації

Одним з найважливіших аспектів в розробці з використанням мікросервісної, а не монолітної архітектури, є міжсервісна комунікація. У монолітному застосунку, що запускається як єдиний процес, виклики між компонентами реалізовані викликами

методів на рівні мови програмування: якщо під час розробки дотримуватися шаблону дизайну MVC, то зазвичай є моделі класів, які відображають реляційні бази даних на об'єктну модель. Також є компоненти, що надають методи, які допомагають виконувати стандартні операції з таблицями бази даних, такі як створення, читання, оновлення та видалення. Компоненти, які найчастіше називаються об'єктами DAO або сховищами, не повинні безпосередньо викликатися з контролерів, тому існує додатковий шар компонентів, який також може додавати деяку частину бізнес-логіки, якщо це необхідно. Для мікросервісної ж архітектури цей процес дещо ускладнюється, адже система стає розподіленою і виконання операцій у більшості випадків означає взаємодію з іншими сервісами, а це означає мережевий виклик. Такий виклик може бути з використанням синхронного або асинхронного протоколу [4].

При синхронному зв'язку з веб-додатками протокол HTTP є стандартом протягом багатьох років, і це не відрізняється для мікросервісів. HTTP — це синхронний протокол без стану, який має свої недоліки. Однак вони не мають негативного впливу на його популярність. При такому підході до комунікації клієнт надсилає запит і чекає відповіді від сервісу. Використовуючи цей протокол, клієнт може спілкуватися асинхронно з сервером, що означає, що потік не заблокований, і відповідь досягне зворотного виклику. Популярним прикладом такої бібліотеки, яка забезпечує найпоширенішу схему синхронної REST-комунікації є Spring Cloud Netflix. Для асинхронного зворотного виклику існують фреймворки, такі як платформа Vert.x або Node.js.

При асинхронному зв’язку ключовим моментом є те, що клієнт не повинен заблокувати потік, очікуючи відповіді. У більшості випадків таке спілкування здійснюється з брокерами обміну повідомленнями. Видавник повідомлення зазвичай не чекає відповіді. Він лише чекає підтвердження, що повідомлення було отримано брокером. Найбільш популярним протоколом для цього типу зв'язку є AMQP (Advanced Message QueuingProtocol), який підтримується багатьма операційними

системами та постачальниками хмарних систем. Асинхронна система обміну повідомленнями може бути реалізована в режимі «один-до-одного» (черга) або «один-до-багатьох» (топік). У зв'язку «один-до-одного» кожен клієнтський запит обробляється рівно одним екземпляром конкретного сервісу. Варто зазначити, що одне повідомлення отримується різними службами, але зазвичай воно не повинно отримуватися різними екземплярами однієї служби. Мікросервісні фраемворки зазвичай реалізують механізми групування споживачів, за допомогою якого різні екземпляри однієї програми розміщуються в конкуруючих споживчих відносинах, в яких тільки один екземпляр повинен обробляти вхідне повідомлення. Найпопулярнішими брокерами повідомлень є RabbitMQ (програмне забезпечення з відкритим вихідним кодом, яке спочатку реалізовувало протокол AMQP і було розширене за допомогою архітектури плагінів для підтримки STOMP, MQTT та інших протоколів) і Apache Kafka (програмна платформа для обробки потоків даних з відкритим вихідним кодом, розроблена компанією LinkedIn і передана Apache Software Foundation). Популярним фраемвором, який передбачає механізми створення мікросервісів, орієнтованих на повідомлення, на основі цих брокерів є Spring Cloud Stream.

Вважається, що побудова мікросервісів базується на тому ж принципі, що й REST з використанням JSON формату. Звичайно, це найпоширеніший метод, але, як можна помітити, не єдиний. Інше часте порівняння — мікросервісної архітектури з архітектурою SOA. У SOA найпоширенішим протоколом є SOAP. Існує багато обговорень щодо того, чи SOAP краще, ніж REST, або навпаки. Кожен з них має свої переваги і недоліки, але REST легкий і незалежний від мови, тому він має вагому перевагу для сучасних додатків. Зв'язок між сервісами має бути ефективним та надійним. Завдяки великій кількості дрібних сервісів, які взаємодіють для завершення однієї транзакції, це може бути непростою задачею.

Існує ряд проблем, що виникають у зв'язку з комунікацією між сервісами. Наприклад: еластичність — можуть існувати десятки або навіть сотні екземплярів

будь-якого заданого мікросервісу. Кожний такий єкземпляр може вийти з ладу з будь-якої причини. Може виникнути помилка на рівні вузла, наприклад, апаратний збій або перезавантаження віртуальної машини. Екземпляр може зламатися або бути перевантажений запитами і не в змозі обробити нові запити. Будь-яка з цих подій може призвести до помилки мережевого виклику. Існують два популярні шаблони проектування, які створені для того щоб зробити обслуговування мережевого виклику більш надійним: повторні спроби (Retry) та вимикач (Circuit Breaker). Виклик у мережі може завершитися невдало через короткочасний збій, який полагодиться сам по собі з часом. Замість того, щоб призвести до відмови, викликач, як правило, повинен повторити операцію певну кількість разів, або поки не закінчиться налаштований період тайм-ауту. Однак, якщо операція не є ідемпотентною, спроби можуть викликати небажані побічні ефекти. Вихідний виклик може бути успішним, але абонент ніколи не отримає відповіді. Якщо виклик виникає повторно, операція може бути виконана двічі. Як правило, небезпечно повторювати методи POST або PATCH, оскільки вони гарантовано не є ідемпотентними. Що стосується Circuit Breaker, то велика кількість невдалих запитів може спричинити вузьке місце, оскільки відкладені запити накопичуються в черзі. Ці заблоковані запити можуть захоплювати критичні системні ресурси, такі як пам'ять, потоки, підключення до бази даних тощо, що може викликати каскадні збої. Шаблон Circuit Breaker може запобігти повторному виклику операції сервісу, яка може призвести до помилки. Ще одна проблема — це балансування навантаження: коли сервіс «А» викликає сервіс «В», запит повинен досягти запущеного екземпляру служби «В». За замовчуванням буде обраний випадковий єкземпляр сервісу, хоча варто застосовувати більш інтелектуальні алгоритми вирівнювання навантаження, засновані на спостереженні затримок, навантаженні, розподіленості за геолокацією або інших показниках. Не менш важливим є розподілене трасування. Одна транзакція може охоплювати кілька сервісів. Це може ускладнити контроль за загальною продуктивністю та станом системи.

Також нетривіальною задачею є версіонування сервісів. Коли команда розгортає нову версію сервісу, потрібно уникати будь-яких впливів на роботу інших сервісів або зовнішніх клієнтів, які залежать від них. Крім того, досить розповсюджна вимога мати можливість запустити кілька версій сервісу поруч та маршрутизувати запити до певної версії. Варто уваги шифрування TLS і взаємна автентифікація TLS. З міркувань безпеки, можливо, потрібно шифрувати трафік між сервісами за допомогою TLS і використовувати взаємну автентифікацію TLS для аутентифікації абонентів. Додатки, що базуються на мікросервісній архітектурі є за природою розподіленими системами, що працюються у декількох процесах або сервісах, як правило, навіть на декількох серверах або вузлах. Кожен екземпляр сервісу зазвичай є процесом. Таким чином, сервіси повинні взаємодіяти за допомогою протоколу міжсервісної комунікаії. У мікросервісній архітектурі сервіси взаємодіють між собою за допомогою передачі повідомлень мережею. Загальноприйнятими методами реалізації такої взаємодії є передача повідомлень за допомогою викликів REST API або моделі підписник-видавець, де сервіси можуть виконувати підписку на канал та отримувати нотифікацію про те, що нове повідомлення було опубліковано до каналу. Тож протоколом може бути HTTP, SOAP, AMQP, RPC, gRPC та багато інших. Кожен з протоколів має як свої переваги, так і недоліки, які необхідно враховувати при проектуванні та виборі певного з них, виходячи з вимог, які висуваються до програмної системи в цілому та до конкретних операцій, які вона реалізує.

1.4 Постановка задачі

На основі виконаного аналізу предметної галузі метою роботи стає встановлення найбільш ефективних засобів і технологій міжсервісного зв’язку у мікросервісній архітектурі програмних систем, виявлення критеріїв їх ефективності,

недоліків та переваг. Також важливим є питання формату обміну повідомленнями у досліджуваних засобах, розробка та тестування програмної системи для дослідження ефективності обраних засобів комунікації.

Однією з основних задач дослідження є виявлення технологій, що дозволяють встановлювати процес міжсервісної взаємодії найбільш ефективно, виходячи з ступеню задовільнення обраним критеріям.

Поставлена задача може бути розділена на такі підзадачі:

– провести аналіз проблем, що існують при міжсервісній комунікації та визначити причини їх виникнення;

– дослідити існуючі технології та протоколи передачі інформації;

– дослідити існуючі засоби серіалізації та десеріалізації даних;

– розробити методику оцінювання ефективності;

– виявити критерії оцінювання ефективності;

– розробити програмну систему для дослідження продуктивності обраних технологій;

– виявити найбільш ефективні технології міжсервісного зв’язку та засоби серіалізації даних у відповідності до обраних критеріїв;

– сформулювати рекомендації щодо впровадження REST та GraphQL з JSON у якості формату серіалізації та десеріалізації.

Це дозволить організовувати мікросервісну взаємодію найбільш ефективним чином, що підвищить значною мірою продукутивність системи, так як передача повідомлень відіграє значну роль у розподілених системах. Також це покращить процес користування системою.

2 ЗАСОБИ І ТЕХНОЛОГІЇ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ

2.1 Аналіз критеріїв вибору технології реалізації міжсервісного зв'язку

Мікросервісна архітектура робить додаток більш масштабованим і легшим для розгортання в порівнянні з монолітним підходом, але наявність декількох сервісів замість монолітного програмного застосунку призводить до збільшення складності системи. Тому важливо, щоб сервіси проектувалися у відповідності до вимог системи. Мікросервіси призначені для багаторазового використання в різних системах і передбачають використяння клієнтами, побудованими за допомогою різних мов та інструментів. Тому важливо використовувати комунікаційні засоби, які можуть бути реалізовані і використані на усіх основних мовах програмування.

При проектуванні архітектури та виборі конкретних засобів комунікації необхідно зважити усі переваги та розуміти компроміси. Серед критеріїв, за якими варто оцінити засоби, можна виділити наступні:

– зчепленість;

– частота комунікацій;

– складність клієнтів;

– когнітивна складність;

– кешування;

– зрозумілість;

– версіонування.

Слабо зчеплена система є такою, в якій кожен з її компонентів має обмаль знань про інші компоненти або зовсім не має, а також використовує обмежену функціональність інших компонентів або зовсім не використовує. Вільне зчеплення інтерфейсів може бути досягнуте шляхом публікації даних у стандартному форматі (такому як JSON або XML). Прикладом може бути REST, у якому клієнт і сервер слабо зчеплені між собою, що надає клієнтам і серверам екстремальні обсяги свободи в

маніпуляції ресурсами. Завдяки цьому швидка інтеграція, еволюція сервера, еластичність надання ресурсів та інші подібні властивості доступні і підтримуються.

В залежності від повноти інформації, що надається інтерфейсом на запит, може варіюватися частота комунікацій між сервісами. У той же час детальна відповідь має більший обсяг і, відповідно, вимагає більше мережевих ресурсів, що призводить до можливості ситуації, коли задля необхідної відповіді у декілька рядків необхідно отримати об’ємний документ.

Складність клієнтів (бібліотек або засобів, що дозволяють взаємодіяти з сервісом) є вартим уваги моментом, адже час, який розробники будуть використовувати на розробку, тестування та підтримку клієнтів, може відчутно відрізнятися в залежності від обраної технології взаємодії.

Когнітивна складність — це міра того, наскільки важко інтуїтивно зрозуміти одиницю коду. На відміну від цикломатичної складності, яка визначає, наскільки важко буде перевірити код, когнітивна складність говорить, як важко буде читати і розуміти ваш код, а зокрема і технологію взаємодії.

Кешування є тимчасовим зберіганням даних задля зменшення серверних затримок. Можливі декілька шарів кешування в залежності від реального випадку використання і вузьких місць. Крім того, кешування корисно для зменшення навантаження з шару збереження даних. Завдання для всіх механізмів кешування полягає в тому, щоб знайти правильний баланс між хорошим показником влучань запитів у кеш та актаульністю і консистентністю даних. В залежності від технології міжсервісної взаємодії змінюється складність реалізації механізму кешвання.

Під критерієм зрозумілості мається на увазі інтуітивна зрузумілість правил користування сервером для клієнтів. Наприклад, у стилі REST є певна домовленість вперше сформульована Леонаром Річардсоном як «Richardson Maturity Model» [5]. Модель розбиває основні елементи підходу REST на три рівні: ресурси, http дієслова і гіпермедіа. Завдяки цьому, вперше зіштовхуючись з новим REST API, розробник може швидше виконати інтеграцію, адже такий API підпорядковується певним

правилам. Виконуючи GET запит до ендпоінту /books, можна очікувати у відповідь усі доступні книги, а виконуючи GET запит до ендпоінту /books/1, можна очікувати у відповідь інформацію про книгу з ідентифікатором 1. У той же час, використовуючи gRPC, важко передбачити, як буде називатися метод отримання усіх доступних книг: books(), getAll(), getAllBook(), listBooks() тощо.

Коли API досягає точки розширення за межами його початкової мети та можливостей, настає час розглянути наступну версію. Незважаючи на те, що наступна ітерація може бути повноцінною версією або просто розширенням функціоналу, важливо враховувати плюси та мінуси того, як можна сповістити про цю зміну користувачів. На відміну від традиційних версій програмного забезпечення версії API можуть мати складні наслідки для продуктів, що використовують його. Широкі випуски версій зазвичай вказують на значну віху в коді API. У ньому зазначається суттєва зміна вимог до споживання та реалізації API. Додаткові функції, які не обов'язково змінюють існуючі виклики, є частиною органічного зростання продукту та не підпадають під ті самі міркування. Стратегії версіонування та складність їх реалізації також значною мірою залежать від обраної технології взаємодії.

Програми, призначені для спілкування з мікросервісами, потребують один або декілька API, які чітко визначають доступні дані та способи доступу до них. Кожен мікросервіс може виставляти назовні публічний інтерфейс для взаємодії або трафік може проходити через шлюз, що має публічний інтерфейс, а з мікросервісами взаємодіє у внутрішній мережі.

Найбільш прямолінійний підхід до проектування публічно доступного інтерфейсу для мікросервісів — це реалізація відкритого API в кожному сервісі, який може безпосередньо споживатися клієнтами. Наявність окремого API у кожному клієнті гарантує, що система залишається децентралізованою і надійною в тому сенсі, що спілкування з клієнтами не спирається на єдиний канал зв’язку. Однак безпосереднє спілкування має свої недоліки. Якщо клієнти хочуть отримувати дані з мікросервісів, які перетинаються і, спираються на дані з інших служб, це призведе до

кількох зайвих викликів між клієнтом і сервісом. Це призведе до великої затримки, що буде результатом неприємного досвіду користування.

У другому підході зі шлюзом API-gateway є окремим сервісом, який об'єднує ресурси, що надаються мікросервісами програмної системи. Шлюз відкриває публічний комунікаційний інтерфейс, який клієнт може споживати і агрегує необхідні ресурси сервісу. Додавання API-шлюзів до мікросервісної архітектури вирішує проблему зайвих викликів, яку породжує альтернатива прямої комунікації. Однак додавання шлюзу до мікросервісних систем означає деградацію часу відгуку при комунікації, оскільки повідомлення потрібно передати два рази замість одного: клієнт викликає сервіс А — одна передача повідомлення, клієнт викликає API-шлюз, API-шлюз викликає сервіс А — дві передачі повідомлення. Загальна практика використання API-шлюзів полягає у розробці одного шлюзу для кожного кінцевого для користувача додатку, який розроблений для зв'язку з мікросервісами. Наприклад, один API-шлюз для frontend додатку та один для мобільного додатку. Цей стиль називається «backend for frontend» і це чудовий спосіб розділити відповідальність зовнішньої комунікації у декількох службах та усунути єдину точку відмови. Недоліком такого патерну є значна дублікація коду, а також збільшення кількості сервісів, які мають обслуговуватись та моніторитись.

У мікросервісній архітектурі кожені з сервісів виконується незалежно і у власному потоці. Тож мікросервіси не повинні мати ніяких залежностей від платформ, інструментів або мов програмування і повинні бути розроблені за допомогою інструментів, які найкраще підходять для конкретної цілі. Найбільш імовірно, що кожен з сервісів використовується декількома різними клієнтами, серед яких інші мікросервіси системи, що розроблені і працюють на різних платформах. Тому важливо використовувати комунікаційні протоколи та засоби, що можуть ефективно використовуватися як мобільним додатком чи web застосунком, так і компонентами системи. Для того, щоб комунікація між веб-службами була ефективною, повідомлення, надіслані по мережі, повинні бути невеликими і мати низький час

серіалізації та десеріалізації. Необхідно виконати моделювання мікросервісної системи, порівнянти та обрати найбільш ефективний засіб міжсервісної комунікації.

2.2 Формат серіалізації даних

Серіалізація — це процес перестворення даних в попередньо узгоджений формат, який можна деконструювати і реконструювати на іншій машині. Серед розповсюджених форматів серіалізації є JSON, XML та двійковий формат. JavaScript Object Notation (JSON) — це текстовий формат, який використовується для серіалізації даних. JSON представляє дані зі структурованими та примітивними типами даних. Підтримувані примітивні типи: string, number, boolean та null. Структуровані типи можуть бути об'єктом або масивом. Об'єкт у форматі JSON містить пари імен і значень, де ім'я має бути строкою, а значення може бути будь-якого підтримуваного примітивного типу або масиву. Об'єкти представлені у вигляді фігурних дужок, які містять довільне число пар імен та значень, розділених комами. Ім'я, яке відповідає значенню, має бути унікальним, щоб зробити розбір об'єкта JSON більш керованим, але це не обов'язково. Масиви представлені квадратними дужками, як у більшості мов програмування. Масив може містити нуль або більше значень будь-якого типу, значення в масиві не повинні бути одного типу [6]. Extensible Markup Language (XML) — це мова розмітки, яка використовується для представлення структурованих даних у веб-службах. XML використовує структуру дерева для опису залежностей даних та ідентифікації ресурсів у якій назва елемента обертається навколо значення ресурсу [7]. Дані, серіалізовані у двійковий формат, мають у результаті менший розмір порівняно з XML або JSON. Однак двійковий формат не є текстовою формою, яку можно було б прочитати та сприйняти. Також двійкова серіалізація вимагає спеціальних фреймворків та інструментів для серіалізації та

десеріалізації об’єктів. Двома найбільш розповсюдженими форматами двійкової серіалізації є Apache Thrift [8] та Google Protocol buffers [9].

Розмір серіалізованого об'єкту значною мірою відрізняється в залежності від обраного формату. Продуктивність серіалізації форматів описана і розглянута у роботах Маеда [10], Поп [11] та Сумарай з Камі Маккі [12]. Результати відносно розміру серіалізованого об’єкту наведені у таблиці 2.1.

Таблиця 2.1 — Розмір об’єктів після серіалізації

Розмір	Найбільший	Проміжний	Проміжний	Найменший
Формат	XML	JSON	Thrift Binary	Protocol Buffers

XML породжує найбільші об'єкти, у той час як Protocol Buffers — найменші. Об'єкти, серіалізовані до бінарного вигляду Thrift були значно більшими ніж об’єкти Protocol Buffers, але усе ще малими у порівнянні з JSON або XML об’єктами. Таблиця 2.2 показує порівняння часу серіалізації для обраних форматів.

Таблиця 2.2 — Час на серіалізацію об’єктів

Час	Найдовший	Проміжний	Проміжний	Найшвидший
Формат	XML	JSON	Protocol Buffers Thrift Binary	Protocol Buffers Thrift Binary

І Маеда і Сумарай з Камі Маккі зробили висновок, що XML займає найдовший час для серіалізації і відтісняє JSON на чітку другу позицію. Проте вони не згодні у питанні часу серіалізації для Thrift і Protocol Buffers. Не зважаючи на це, у обох статтях чітко видно, що обидва двійкові формати мають менший час серіалізації порівняно з JSON і XML.

Час десеріалізації для кожного формату був представлений з різними результатами. Існує згода щодо того, що XML має найдовший час десеріалізації, але

результати для Protocol Buffers, Thrift і JSON змінюються. Маеда доводить, що використання фреймворку Thrift для десеріалізації JSON показує найкращі результати. Однак він також доводить, що використання JSON в цілому повільніше, ніж десеріалізація двійкового формату Thrift або Protocol Buffers. Оскільки JSON показує задовільну продуктивність, хоча і меншу, ніж двійкові формати, та має текстовий формат, який легко сприймати, саме його обрано у якості формату серіалізації для виконання дослідження.

2.3 Дослідження засобів і технологій міжсервісного зв’язку

Розглянемо комунікаційні технології організації міжсервісного зв’язку, які широко використовуються і мають певний ступінь підтримки багатоплатформовості.

Simple Object Access Protocol (SOAP) — це багатоплатформовий протокол для зв'язку через Інтернет. SOAP використовує XML для інкапсуляції повідомлень і для визначення, як їх передавати та отримувати. Спочатку SOAP розроблювався для реалізації віддаленого виклику процедур (RPC). На даний час протокол використовується для обміну повідомленнями в XML форматі, а не тільки для виклику процедур. SOAP є розширенням мови XML-RPC. SOAP можна використовувати незалежно від протоколу прикладного рівня: HTTP, FTP, SMTP та інші. Проте варто мати на увазі, що його взаємодія з кожним із цих протоколів має свої особливості. Найчастіше SOAP використовується разом з HTTP. Архітектуру SOAP можна розділити на три частини: постачальники сервісу, реєстр сервісів і споживач сервісу. Постачальник послуг створює SOAP сервіс, який відображає дані зі сховища в повідомлення SOAP і надає опис доступного сервісу до реєстру сервісів. Реєстр сервісів надає описи сервісів і засоби спілкування з ними. Споживач сервісу

може отримати доступ до реєстру, щоб дізнатися, які сервіси доступні, і потім спілкуватися з таким сервісом за допомогою SOAP-повідомлень [13].

Remote procedure call (RPC) — протокол, що дозволяє програмі, запущеній на одному комп'ютері, звертатись до функцій (процедур) програми, що виконується на іншому комп'ютері, подібно до того, як програма звертається до власних локальних функцій, тобто це виклик процедури одного процесу у іншому через мережу. Існує кілька способів реалізації RPC, більшість сучасних мов програмування має підтримку вбудованого RPC, але існують також фреймворки, які побудовані спеціально для забезпечення міжплатформних RPC. Такі фреймворки роблять виклики міжплатформовими, описуючи дані ресурсів, процедур, їх вхідних параметрів і очікуваних вихідних даних, використовуючи незалежну мову визначення інтерфейсу (IDL — Interface Definition Language). Потім процедури, описані за допомогою IDL, можуть використовуватися для генерації коду сервера і клієнта, які реалізують процедури на різних мовах програмування, на різних машинах. Віддалені виклики здійснюються клієнтом, який викликає локально реалізовану процедуру, повідомлення потім упаковується разом з параметрами і надсилається на сервер, який реалізував таку ж процедуру. Сервер виконує процедуру і передає відповідь разом з результатом [14].

Apache Thrift — це мова опису інтерфейсів, що використовується для визначення і створення сервісів під різні мови програмування. Являє собою фреймворк до RPC. Використовується компанією Facebook у якості масштабованого кросмовного сервісу з розробки. Інакше кажучи це фреймворк для міжмовної комунікації додатків. Thrift також має функціональність як протокол зв'язку віддаленого виклику процедур між веб-службами. Оскільки Thrift — це фреймворк, він не є повністю незалежним від платформи. Він має підтримку 14 різних мов, включаючи Java, C#, C++, JavaScript, Python і PHP. Альтернативами серіалізації, які представлені у всіх мовах в рамках Thrift є JSON і двійковий формат. Thrift має

підтримку як для протоколу HTTP, так і для протоколу TCP у якості транспортного шару.

Фреймворк gRPC також є інструментом для крос-платформних віддалених викликів процедур, також як і Thrift. У gRPC Protocol Buffers використовується як формат серіалізації для повідомлення, а також як мова визначення інтерфейсу, що робить міжплатформні виклики можливими. Фреймворк gRPC доступний на 10 різних мовах, включаючи Java, C #, C++, JavaScript і Python. Також gRPC має підтримку для нового протоколу HTTP 2, але також може використовувати TCP у якості транспортного шару. HTTP 2 — це друга версія HTTP протоколу, яка стандартизована у 2016 році і базується на попередній версії HTTP 1.1. HTTP 1.1 була стандартизована RFC 2616 в 1999 році. Основною відмінністю між версіями є те, що HTTP 2 є бінарним. Зазнали зміни і способи розбиття даних на фрагменти, їх передача між сервером і клієнтом. У другій версії сервер може відіслати дані, які ще не були запитані клієнтом, але потенційно знадобляться клієнту для побудови сторінок [15].

Representational state transfer (REST) — це архітектурний стиль для зв'язку між службами. REST не прив'язаний до жодного конкретного транспортного протоколу або методу серіалізації, але транспортування даних через HTTP і серіалізація у JSON або XML стали найбільш широко використовуваними альтернативами. Доступ до ресурсів даних здійснюється через індивідуальний уніфікований ідентифікатор ресурсу (URI). Коли REST реалізується з HTTP в якості засобу для транспортування, ресурси ідентифікуються унікальною URL-адресою. Ресурси керуються за допомогою використання HTTP методів-дієслів GET, POST, PUT і DELETE. Дані запитуються в архітектурному стилі REST за допомогою запитів GET до URL-адреси, що ідентифікує певний ресурс. Якщо необхідно додати або змінити дані, POST запит відправляється до застосунку з повідомленням, що містить інформацію про цей ресурс. REST став однією з найбільш популярних альтернатив комунікації між сервісами завдяки тому, що він легкий і може бути прийнятий будь-якою мовою.

GraphQL є мовою запитів для інтерфейсів прикладного програмування, що моделює дані у вигляді графів JSON-об'єктів. GraphQL призначена для реалізації як прикладний рівень поверх базової системи зберігання даних. Ця мова може бути реалізована в будь-якій системі незалежно від базової моделі зберігання даних. У GraphQL запит моделюється як JSON-об'єкт, де імена полів даних у запиті вказують, яка інформація очікується у відповідь. GraphQL розбирає запит і додає відсутні значення в полях запиту [16]. Клієнт отримає необхідні дані зі значеннями доданими до вказаних полів у запиті. GraphQL не прив'язана ні до якого транспортного протоколу, але використання HTTP ендпоінтів для передачі та прийому даних у вигляді GET і POST запитів стали стандартним підходом. Приклад запиту GraphQL показаний на рисунку 2.1.

```
query {
      person {
            name
            ID
      }
      friends {
            name
      }
}
```

Рисунок 2.1 — Приклад запиту GraphQL

Запит структурований як об'єкт JSON, але без значень для кожної пари імен та значення. Фреймворк GraphQL розбирає запит і додає значення до бажаних полів, вказаних у запиті. Поле friend у цьому випадку визначається на сервері як тип person. GraphQL сервер визначає, які запити є прийнятними і поля, які вони можуть містити виходячи з створеної попередньо схеми. Клієнтський додаток має можливість відправити запит з переліком лише тих полів, які йому необхідні, завдяки цьому ніколи не буде викликане зайве навантаження мережі через надлишкові дані. Приклад

відповіді GraphQL на друзів запит показано на рисунку 2.2, з якого видно, що у відповіді присутня лише запитувана інформація.


```
{
      "data": {
            "person": {
                  "name" : " John Smith ",
                  "ID" : "12345",
                  "friends": [
                        {
                              "name" : "Bill Smith"
                        },
                        {
                              "name" : "Bob Smith"
                        }
                  ]
            }
      }
}
```


Рисунок 2.2 — Приклад відповіді GraphQL

Немає чіткої відповіді на питання, який метод комунікації найкращий для мікросервісів. Хоча фреймворки gRPC і Thrift потенційно пропонують найкращу продуктивність, вони додають залежності від архітектури мікросервісів, змушуючи її розвиватися на певних мовах. REST, який є повністю незалежним від платформи, оскільки він є лише архітектурним стилем, може бути реалізований на будь-якій платформі, з будь-якою мовою, використовуючи будь-який транспорт і будь-яку серіалізацію. Проте REST не має структури, яку пропонують фреймворки і протоколи, і часто слід використовувати інструменти третіх сторін для відстеження його інтерфейсів. SOAP пропонує структурований спосіб відправки повідомлень між службами зі строгими правилами, що визначають структуру повідомлень і способи їх обробки. Основний недолік SOAP полягає в тому, що він прив'язаний до використання XML, який, безумовно, має найгіршу продуктивність із всіх перелічених форматів серіалізації.

3 ОПИС ПРОГРАМНОЇ СИСТЕМИ ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ТЕХНОЛОГІЙ МІЖСЕРВІСНОГО ЗВ’ЯЗКУ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

3.1 Дизайн програмної системи

Для дослідження продуктивності обраних технологій розроблено мікросервісну систему, що допоможе відповісти питання: чи може зв'язок між сервісами у мікросервісній архітектурі стати більш ефективними з використанням мови запитів GraphQL, порівняно до HTTP-запитів до REST API? Тобто система дозволить виконати тестування на різних обсягах даних з використнням обраних засобів комунікації.

Дизайн архітектури програмної системи — це перше завдання, до якого слід підходити, створюючи систему мікросервісів. Для кожного проекту, що використовує мікросервісний підхід, обов’язковими є два загальних компоненти, що полегшують побудову системи, а саме:

– Service discovery: сервіс, що визначає перелік діючих компонентів у системі. Він зберігає реєстр діючих сервісів, їх актуальни адреси у мережі та дозволяє виконувати реєстрацію у переліку;

– API Gateway: сервіс, що є єдиною точкою входу до системи, роутер який направляє запити до необхідних сервісів та може виконувати загальну логіку з обробки запитів.

Також до тестового середовища входять чотири різних мікросервіси і тестовий клієнт, створений як веб застосунок. Доступ до мікросервісів здійснюється через балансувальник навантаження, який перенаправляє трафік з відкритих точок доступу до REST або GraphQL шлюзу (api gateway). Один з шлюзів надає REST API, другий GraphQL API. Кожен з шлюзів спілкується з кожним з мікросервісів, використовуючи відповідний протокол. Кожен з чотирьох мікросервісів має реалізацію та підтримку

як REST API, так і GraphQL API. Також кожен з сервісів має доступ до ізольованої бази даних у хмарному RDS сховищі. На рисунку 3.1 зображено графічний опис структури системи.

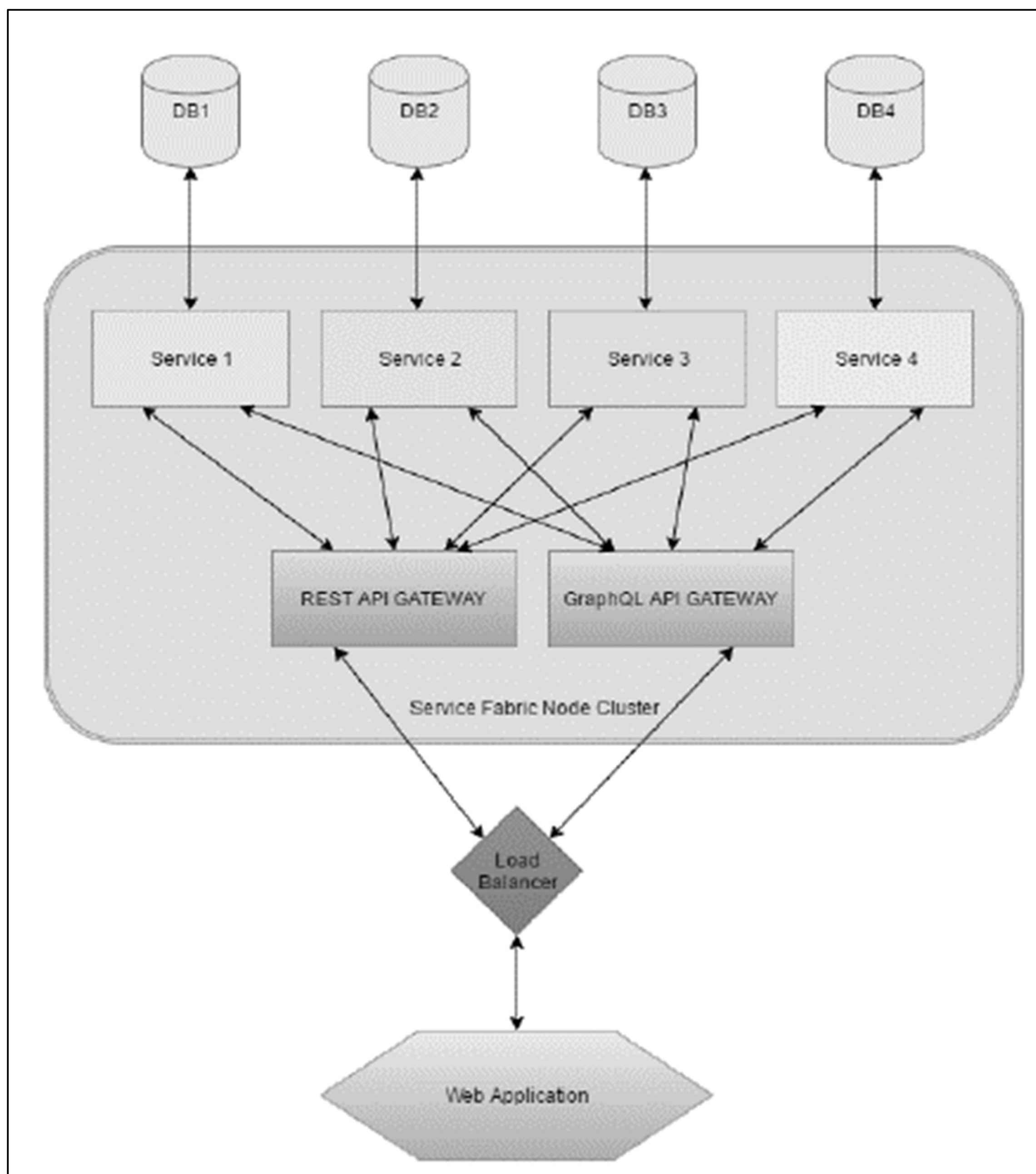

Рисунок 3.1 — Графічний опис структури системи

Мікросервіси в цій системі мають виконувати тільки доступ до бази даних і не виконувати жодних фактичних обчислень. Кожний сервіс має доступ до даних, що

доступні тільки для цього конкретного сервісу. Шлюзи взаємодіють з сервісами відповідно до обраного протоколу. У якості бази даних використовується MariaDB, яка обслуговується окремо від кластеру мікросервісів і надається як сервіс. MariaDB обрано через високу продуктивність та простоту інтеграції. Кожен сервіс представляє собою кубернетис под утворений, з машини типу «n1-standard-1», що має характеристики, які наведені у таблиці 3.1.

Таблиця 3.1 — Специфікація серверу для мікросервісу

Характеристика	Значення
Місце положення	Західна Європа
CPU	Intel Xenon E5-2660 @2.2 GHz, одне ядро
RAM	3.75 GB
Операційна система	Centos 7

Структура даних, які доступні системі, зображена на рисунку 3.2

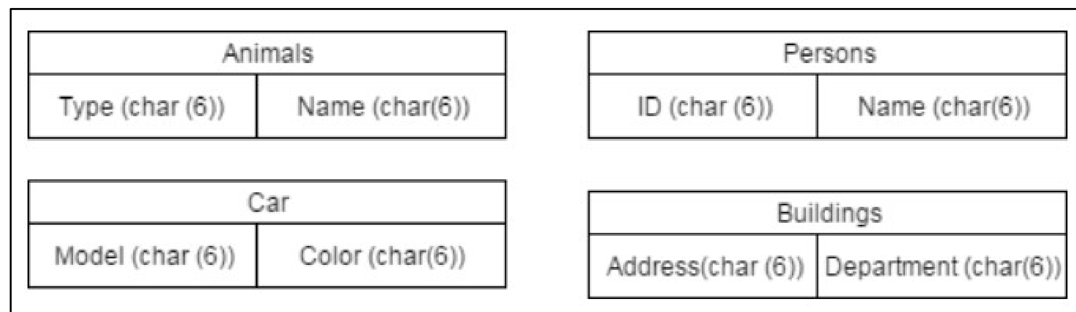

Рисунок 3.2 — Структура таблиць які містять тестові дані

Кожен Сервіс має доступ до однієї з таблиць: animals, cars, persons або buildings. Кожне поле таблиці має обмеження на довжину у 6 символів і містить точно 6 символів, для того щоб отримувати передбачувані та рівнозначні об'єкти даних.

REST API реалізується на мові програмування Kotlin з використанням фраемворку Spring MVC. API надає ендпоінти для отримання усіх даних, які може надати кожен з сервісів. Для серіалізації API приймає лише повідомлення у JSON форматі. HTTP едпоінти мають наступний вигляд:

– /api/person;

– /api/animal;

– /api/car;

– /api/building.

GraphQL API ендпоінт має вигляд /api/graphql. HTTP ендпоінт реалізований також з використанням фраемворку Spring MVC. У якості GraphQL двигуна, який оброблює запити та будує відповіді обрано graphql-java. GraphQL сервіс реалізовано з використанням схеми, яка підтримує один тип запитів. Запит має чотири поля даних, де кожне з полів — лист даних одного з сервісів.

Веб-застосунок побудований для отримання даних від мікросервісів таким чином, щоб можливо було заміряти продуктивність GraphQL та REST. Клієнт розроблений з використанням HTML та JavaScript з фраемворком React. Доступ до REST API виконано з використанням HTTP клієнту Axios [17]. Для форматування та відправки GraphQL запитів використано GraphQL клієнт для React — Apollo. Apollo побудований для облегшення комунікації з використанням GraphQL через HTTP. Клієнт вимірює загальний час, необхідний для отримання відповіді на запит від GraphQL і REST мікросервісів.

3.2 Опис обраних технологій

У якості основної мови розробки обрано Kotlin — статично типізована мова програмування, що працює на JVM і розробляється компанією JetBrains. Також має

можливість компіляції у JavaScript. Мову названо на честь острова Котлін у Фінській затоці. Автори ставили перед собою за мету створення лаконічнішої та типобезпечнішої мови, ніж Java, і простішої мови, ніж Scala. Наслідком спрощення у порівнянні зі Scala стали також швидша компіляція та краща підтримка IDE. 17 травня 2017 року Kotlin увійшла у список офіційно мов, що офіційно підтримуються для розробки додатків на платформі Android. Синтаксис Kotlin дуже схожий на Java і завдяки тому, що обидві мови працюють під керуванням JVM, є сумісними, наявна можливість використання бібліотек і фреймворків написаних на Java (об'єктно-орієнтована мова програмування, розроблена компанією Sun та придбана корпорацією Oracle). Java застосування зазвичай компілюються в спеціальний байт-код, завдяки чому вони можуть працювати на будь-якій JVM незалежно від архітектури комп'ютера. Перевагою даного способу виконання програм є повна незалежність байт-коду від операційної системи та апаратного забезпечення, що дозволяє виконувати Java застосування на будь-якому пристрої, для якого розроблено реалізацію JVM. Іншою не менш важливою особливістю Java є гнучка система безпеки, завдяки тому, що виконання програми повністю керується віртуальною машиною. Операції, що намагаються підвищити встановлені повноваження програми, викликають негайне переривання. Не рідко до недоліків концепції віртуальної машини відносять той факт, що виконання віртуальною машиною байт-коду може зменшувати продуктивність програм та алгоритмів, реалізованих на Java. В останній час було внесено ряд вдосконалень, котрі збільшили швидкість виконання програм на Java. Безсумнівно, Java зараз є однією з найпопулярніших мов, для неї існує безліч інструментів.

Середовищем розробки обрано IntelliJ IDEA. Вона являє собою інтегроване середовище розробки від компанії JetBrains для різноманітних мов програмування (Java, Scala, PHP, Python та ін.). Система розповсюджується у двох версіях: у вигляді урізаної за функціональністю безкоштовної версії «Community» і комерційної повнофункціональної версії «Ultimate». Для останньої мають можливість отримати

безкоштовну ліцензію активні розробники відкритих проектів. Вихідний код Community версії поширюються під ліцензією Apache 2.0. Двійкові збірки розповсюджуються для Linux, Mac OS X і Windows. Community версія середовища має підтримку інструментів (у вигляді плагінів) для проведення тестування JUnit та TestNG, систем контролю версій Git, Subversion, CVS, Mercurial, засоби виконання збірок Gradle, Maven, Ant, мови програмування Java, Scala, Groovy, Clojure та багато іншиї. Підтримується розробка мобільних застосунків для платформи Android. До складу такоє входить XML-редактор, Swing UI Designer (модуль візуального проектування GUI-інтерфейсу), редактор RegExp виразів, система перевірки коду на коректність, система контролю виконання завдань і плагін для виконання імпорту та експорту проектів розроблених у середовищі Eclipse. Доступними є засоби інтеграції з системами трекінгу помилок: JIRA, YouTrack, Trac, Pivotal Tracker, Redmine, GitHub, Lighthouse. Платна версія «Ultimate Edition» у свою чергу додатково має більшої кількості мов програмування (наприклад: Ruby, JavaScript, CoffeeScript та інші), відрізняється підтримкою технологій Java Enterprise Edition, UML моделювання, можливістю оцінити покриття коду тестами, можливістю працювати з різноманітними фреймворками, засобами інтеграції з багатьма системами.

У якості фреймворку, що реалізує патерни Dependency Injection та Inversion of Control обрано Spring — це програмний каркас (фреймворк) з відкритим сирцевим кодом та контейнер з підтримкою інверсії управління для платформи Java [18]. Основні властивості Spring Framework можуть використовуватися будь-яким застосунком Java. Також існує розширення для розробки веб-застосунків з використанням платформи Java EE. Не дивлячись на це, Spring не нав'язує жодної конкретної моделі програмування. Spring Framework здобув популярності в співтоваристві Java в якості альтернативи (заміна Enterprise JavaBean (EJB) моделі). Spring може бути розглянутий як колекція менших фреймворків або фреймворків у фреймворку, більшість з яких може працювати незалежно один від одного, проте вони забезпечують більшу функціональність при спільному їх використанні і поділяються

на наступні структурні елементи типових комплексних програм:

– контейнер інверсії керування: управління життєвим циклом Java об'єктів і конфігурація компонентів застосунків, здійснюється в основному за допомогою інверсії керування;

– доступ до даних: робота з реляційною СУБД на Java з застосуванням об'єктно-реляційного відображення;

– управління транзакціями;

– аспекто-орієнтоване програмування: допомога у реалізації наскрізних процедур;

– авторизація і аутентифікація: налаштування процесів безпеки, які підтримують ряд стандартів, інструментів і практик за допомогою Spring Security (у минулому система безпеки Acer Spring);

– модель-вигляд-управління (MVC): програмний фреймворк на основі HTTP сервлету, який забезпечує створення веб-додатків і веб-служб;

– тестування: підтримка інструментів, що забезпечують написання інтеграційних та юніт тестів;

– віддалене керування: конфігураційний вплив і керування Java-об'єктами для локальної або віддаленої конфігурації через JMX.

Для того щоб мікросервіси якнайшвидше і якнайпростіше можна було зробити виконуваними і готовими до запуску використано Spring Boot —рішення Spring для створення автономних застосунків, які можна «просто запустити». Підключаючи Spring і сторонні бібліотеки, можна запустити додаток не докладаючи великих зусиль. Переважній більшості Spring Boot застосунків потрібно зовсім маленька Spring-конфігурація. Розгянемо можливості, які Spring Boot надає:

– створення повноцінних Spring додатків;

– вбудований контейнер сервлетів Tomcat або Jetty (не потрібна установка WAR файлів);

– забезпечує початкові POM файли для спрощення Maven (або Gradle) конфігурації;

– надає механізм автоматичної конфігурація Spring, коли це можливо.

У якості системи керування базами даних обрано MariaDB — реляційна СУБД, створена на початку 2009 як відгалуження (форк) MySQL. MariaDB поширюється під вільною та відкритою ліцензією GNU GPL.

Для моніторингу системи та зручності виконання замірів використано Zipkin та Hystrix Dashboard, що працює разом з Turbine і мають нативну інтеграцію зі Spring Framework. Zipkin являє собою розподілену систему відстеження, яка допомагає збирати часові характеристики, необхідні для усунення проблем затримки в мікросервісних архітектурах. Zipkin керує як збиранням, так і пошуком цих даних. Hystrix — це бібліотека, призначена для ізоляції точок доступу до віддалених систем, сервісів і бібліотек третьої сторони, припинення каскадних збоїв і забезпечення стійкості в складних розподілених системах.

Для контролю версій використовувалась система Git. Git являє собою розподілену систему керування версіями файлів. Вона є однією з найефективніших, високопродуктивних і надійних систем контролю версій, яка надає засоби нелінійної розробки, що базуються на відгалуженні (checkout) і злитті (merge) гілок (branch). Для гарантування цілосності історії та стійкості до змін дати оновлення використовуються криптографічні методи, також наявна можливість прив’язки цифрових підписів розробників до комітів і тегів. Віддалений доступ до git репозиторіїв забезпечується так званим git-демоном та SSH або HTTP сервером. TCP-сервіс git-daemon постачається з дистрибутивом Git і на ряду з SSH надійним і найпоширенішим методом доступу. Доступ за HTTP, не дивлячись на деякі обмеження, достатньо популярний в контрольованих мережах, через те що дозволяє використання готових конфігурацій мережевих фільтрів.

Для розгортання системи у мережі використана хмарна платформа Google Cloud Platform, яка є запропонованим компанією Google набором хмарних служб, що

виконуються з використанням тієї ж інфраструктури, яку Google застосовує для власних продуктів, призначених для кінцевих користувачів, таких як Google Search та YouTube. Крім інструментів для керування, надається ряд хмарних модульних служб, таких як зберігання даних, аналіз даних, обчислення та машинне навчання. GCP надає такі можливості як: інфраструктура як послуга (IAAS), платформа як послуга (PAAS) та безсерверні обчислення (serverless computing). GCP є частиною Google Cloud, який включає G Suite, корпоративні версії Chrome OS та Android, а також надає API для Google Maps та машинного навчання. Ця платформа обрана завдяки тому, що вона надає нативну підтримку Kubernetes, а саме ним і оркеструються розроблені мікросервіси.

Kubernetes є відкритою системою автоматичного розгортання, масштабування та управління застосунками у контейнерах. Система розроблена компанією «Google» та підтримує набір інструментаріїв з управління контейнерами, до яких входить Docker. Kubernetes (k8s) визначає набір «будівельних» блоків (так званих «примітивів»), які забезпечують механізми для розгортання, масштабування та підтримки застосунків. K8s являється слабко зв'язним та розширюваним, завдяки чому може налаштовуватися під різноманітні робочі навантаження. Розширюваність забезпечується API k8s, який використовується його внутрішніми модулями, крім того розширеннями і контейнерами, що працюють під керуванням k8s.

Артефактами оркестрації в розробленій системі є docker-контейнери. Docker — програмне забезпечення для автоматизації розгортання і управління додатками в середовищі віртуалізації на рівні операційної системи. Дозволяє «спакувати» додаток з усім його оточенням і залежностями в контейнер, який може бути перенесений на будь-яку Linux-систему з підтримкою cgroups в ядрі, а також надає середовище з управління контейнерами. Docker спочатку використовував можливості LXC, з 2015 року застосовував власну бібліотеку, що абстрагує віртуалізаційні можливості ядра Linux — libcontainer. З появою Open Container Initiative почався перехід від монолітної до модульній архітектурі. До складу програмних засобів входить демон — сервер

контейнерів (запускається командою docker -d), клієнтські засоби, що дозволяють з інтерфейсу командного рядка управляти образами і контейнерами, а також API, що дозволяє в стилі REST управляти контейнерами програмно. Демон забезпечує повну ізоляцію та запускається на вузлі контейнерів на рівні файлової системи (у кожного контейнера власна коренева файлова система), на рівні процесів (процеси мають доступ тільки до власної файлової системи контейнера, а ресурси розділені засобами libcontainer), на рівні мережі (кожен контейнер має доступ тільки до прив'язаного до нього мережевого простору імен і відповідним віртуальним мережевим інтерфейсам).

Набір клієнтських засобів дозволяє запускати процеси в нових контейнерах (docker run), зупиняти і запускати контейнери (docker stop і docker start), припиняти і відновлювати процеси в контейнерах (docker pause і docker unpause). Серія команд дозволяє здійснювати моніторинг запущених процесів (docker ps за аналогією з ps в Unix-системах, docker top по аналогії з top і інші). Нові образи можливо створювати зі спеціального сценарного файлу (docker build, файл сценарію носить назву dockerfile), можливо записати всі зміни, зроблені в контейнері в новий образ (docker commit). Всі команди можуть працювати як з docker-демоном локальної системи, так і з будь-яким сервером Docker, доступним через мережу.

Збірка артефактів системи організована з використанням системи складання Gradle. Gradle — система автоматичного складання, побудована на принципах Apache Ant і Apache Maven, але надає DSL(domain specific language — мова програмування, спеціалізована для конкретної області застосування) на мові Groovy замість традиційної XML-подібної форми подання конфігурації проекту. На відміну від Apache Maven, заснованого на концепції життєвого циклу проекту, і Apache Ant, в якому порядок виконання завдань визначається відносинами залежностей, Gradle використовує спрямований ациклічний граф для визначення порядку виконання завдань. Gradle був розроблений для розширюваних багато-проектних збірок, і підтримує інкрементальні збірки, визначаючи, які компоненти дерева збірки не змінилися і які завдання, залежні від цих частин, не вимагають перезапуску. Основні

плагіни призначені для розробки і розгортання Java, Groovy і Scala додатків. Також розроблюються плагіни для інших мов програмування.

Тестування виконано за допомогою утиліти Apache JMeter — це програмне забезпечення з відкритим вихідним кодом, що призначене для тестування навантаження, функціональної поведінки та вимірювання продуктивності веб-сайтів.

Apache JMeter може використовуватися для перевірки функціональності та продуктивності як на статичних, так і на динамічних ресурсах. Він може бути використаний для симуляції важкого навантаження на сервер, мережу або об'єкт для перевірки його сили або аналізу загальної продуктивності при різних типах навантаження.

3.3 Програмна реалізація

Система являє собою мультимодульний Gradle проект. До підмодулів входять 4 мікросервіси, які моделюють сервіси, що виконують бізнес логіку: animal-service, building-service, car-service, discovery-service, person-service. Для забезпечення роботи створених сервісів необхідна також інфраструктура, до якої входять додаткові 3 мікросервіси: api-gateway, config-service, discovery-service. Структура проекту зображена на рисунку 3.3.

Api-gateway-service являє собою точку входу до системи, яка вміє взаємодіяти із іншими мікросервісами використовуючи як REST, так і GraphQL API. У якості конкретної імплементації обрано Netflix Zuul [19].

Під час роботи з мікросервісами важливо памятати про можливість конфігурації змінних так, щоб не доводилось змінювати код сервісу. Саме для таких цілей було розроблено мікросервісний патерн «Configuration server». Цей сервер є сховищем конфігураційних змінних. Кожен сервіс залежить залежати від нього та скачує свої

конфігурації. Окрім конфігураційний сервервер надає можливість уникнути дублікації конфігураційних змінних, якщо вони використаються у декількох сервісах. У якості конкретної реалізації патерну обрано Cloud Configuration Server [20]. Імплементацією є розроблений config-service, що відповідає за керування конфігурацією та розповсюджує відповідні налаштування для кожного з компонентів системи.

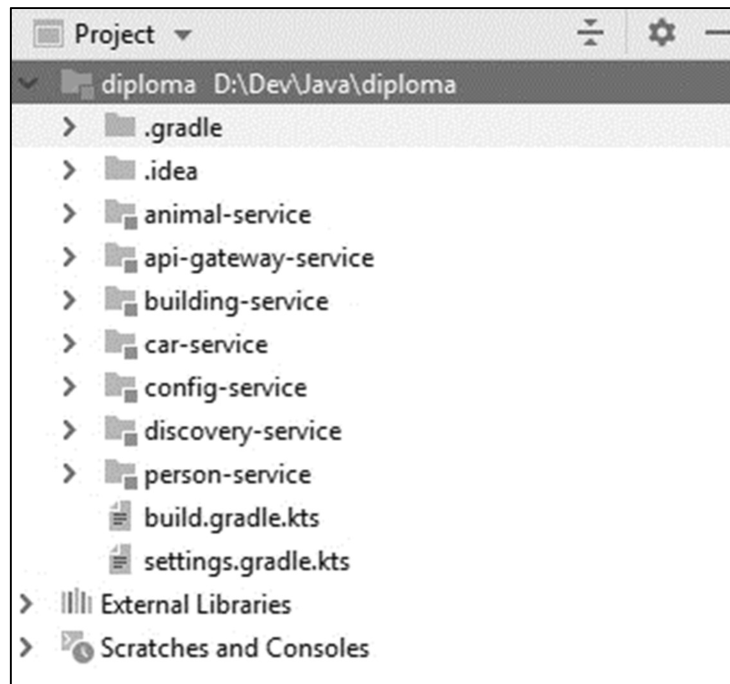

Рисунок 3.3 — структура створеного проекту у середовищі розробки IntelliJ IDEA

Discovery-service є реалізацією патерну мікросервісної архітектури «Service Discovery», що дозволяє мікросервісам реєструватися у ньому, та отримувати перелік активних сервісів з їх актуальними адресами у мережі для облегшення процесу керування сервісами та їх взаємодії. У якості конкретної реалізації обрано Netflix Eureka [21].

Кожен з сервісів, що відповідають за сутності не виконує ніяких обчислень, а лише надає API, за допомогою якого можна отримати інформацію, що міститься у базі

даних. Необхідний функціонал для реалізації REST API надає залежність «org.springframework.boot:spring-boot-starter-data-rest». Розроблені ендпоінти мають вигляд подібний до зображеного на рисунку 3.4.

```
@RestController
class PersonController(
    private val personService: PersonService,
    private val personConverter: PersonToDtoConverter
) {

    @GetMapping( value: "/person")
    fun list() = personService.all()
        .map(personConverter)
}
```

Рисунок 3.4 — REST ендпоінт отримання усіх об’єктів person

Реалізація GraphQL API дещо складніша. Перш за все необхідно описати схему даних. У розроблюваному випадку вона досить тривіальна. Для типу «Person», який виглядає як «data class Person(val id: String, val name: String)» схема, що описує тип та запит для отримання усіх сутностей зображена на рисунку 3.5.

```
type Person {
  id: ID!
  name: String!
}

extend type Query {
  list: [Person]
}
```

Рисунок 3.5 — Схема даних GraphQL для типу Person

Після декларації схеми необхідно створити так звані резолвери — компоненти, які вміють оброблювати запити описані GraphQL схемою. Коли резолвери

імплементовані, система готова приймати та оброблювати GraphQL запит, наведений на рисунку 3.6.

```
{
  query {
    list {
      name
      id
    }
  }
}
```

Рисунок 3.6 — Запит GraphQL на отримання усіх імен та ідентифікаторів для персон

Оскільки використано Spring Boot фраемворк, результатом компіляції та збірки проекту є артефакт з вмонтованим веб-сервером та усіма залежностями, які потрібні для незалежного запуску мікросервісу. Залишається написати спеціальний файл для декларації середовища запуску контейнеру з сервісом (див. рис. 3.7). Він залишається актуальним для всіх сервісів системи без винятку.

```
FROM java:8-jre-alpine
VOLUME /tmp
ADD build/libs/*.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Рисунок 3.7 — Докерфайл дл середовища запуску сервісу

Сервісом, що контролює процес розгортання у хмарі (Google Kubernetes Engine) є Helm. Для його використання потрібно задекларувати стан відповідних абстракцій у файлах типу Yaml. Це обов‘язкових крок для всіх сервісів. Наведемо файл розгортання:

```
apiVersion: apps/v1
kind: Deployment
```

```
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    metadata:
      labels:
        app: {{ template "appname" . }}
        release: {{ .Release.Name }}
    spec:
      containers:
      - name: server
        image: "{{ .Values.img.repo }}:{{ .Values.img.tag }}"
        imagePullPolicy: {{ .Values.img.policy }}
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: {{ .Values.application.profiles.active }}
        ports:
        - name: server
          containerPort: {{ .Values.application.port }}
        livenessProbe:
          httpGet:
            path: {{ .Values.application.healthPath }}
            port: server
          periodSeconds: 30
          initialDelaySeconds: 60
          timeoutSeconds: 5
        readinessProbe:
          httpGet:
            path: {{ .Values.application.healthPath }}
            port: server
          periodSeconds: 30
          initialDelaySeconds: 60
          timeoutSeconds: 5
        resources:
          requests:
            cpu: 100m
            memory: 300M
```

Серед важливих параметрів: replicas — кількість екземплярів сервісу, що повинно буде створитися; image — назва образу, що створюється завдяки Dockerfile; env — значення для системних змінних; livenessProbe та readinessProbe — конфігурація, для отримання інформації про стан сервісу, його життєздатність та готовність до роботи.

CI/CD процес для сервісів впроваджено засобами Gitlab. Для кожного з мікросервісів створено окремий репозиторій. Кожен з репозиторіїв містить файл .gitlab-ci.yml. Цей файл відповідає за декларацію поведінки системи безперервної інтеграції та розбиття цієї поведінки на певні логічні етапи:

```
image: alpine:latest
```

```yaml
variables:
  KUBERNETES_VERSION: 1.10.9
  HELM_VERSION: 2.11.0
  DOCKER_DRIVER: overlay2
stages:
  - build
  - package
  - production
build_jar:
  stage: build
  image: openjdk:8-alpine
  script: "chmod a+x gradlew && ./gradlew build"
  artifacts:
    paths:
      - build/libs/*.jar
  only:
    - branches
build_image:
  stage: package
  image: docker:stable
  services:
    - docker:stable-dind
  script:
    - setup_docker
    - build_image
  only:
    - branches
production:
  stage: production
  script:
    - check_kube_domain
    - install_dependencies
    - download_chart
    - ensure_namespace
    - initialize_tiller
    - create_secret
    - deploy
  environment:
    name: production
    url: http://$CI_PROJ_PATH.$AUTO_DEVOPS_DOMAIN
  only:
    refs:
      - master
  except:
    variables:
      - $KUBECONFIG == null
```

Пайплайн складається з трьох етапів. Першою є збірка Gradle проекту, потім виконується створення Docker образу і завершується процес розгортанням щойно створеного образу у Kubernetes кластері за допомогою сервісу Helm. Після виконання

Gitlab CI пайплайну наявна можливість побачити результат запуску у веб інтерфейсі Gitlab (див. рис. 3.8).

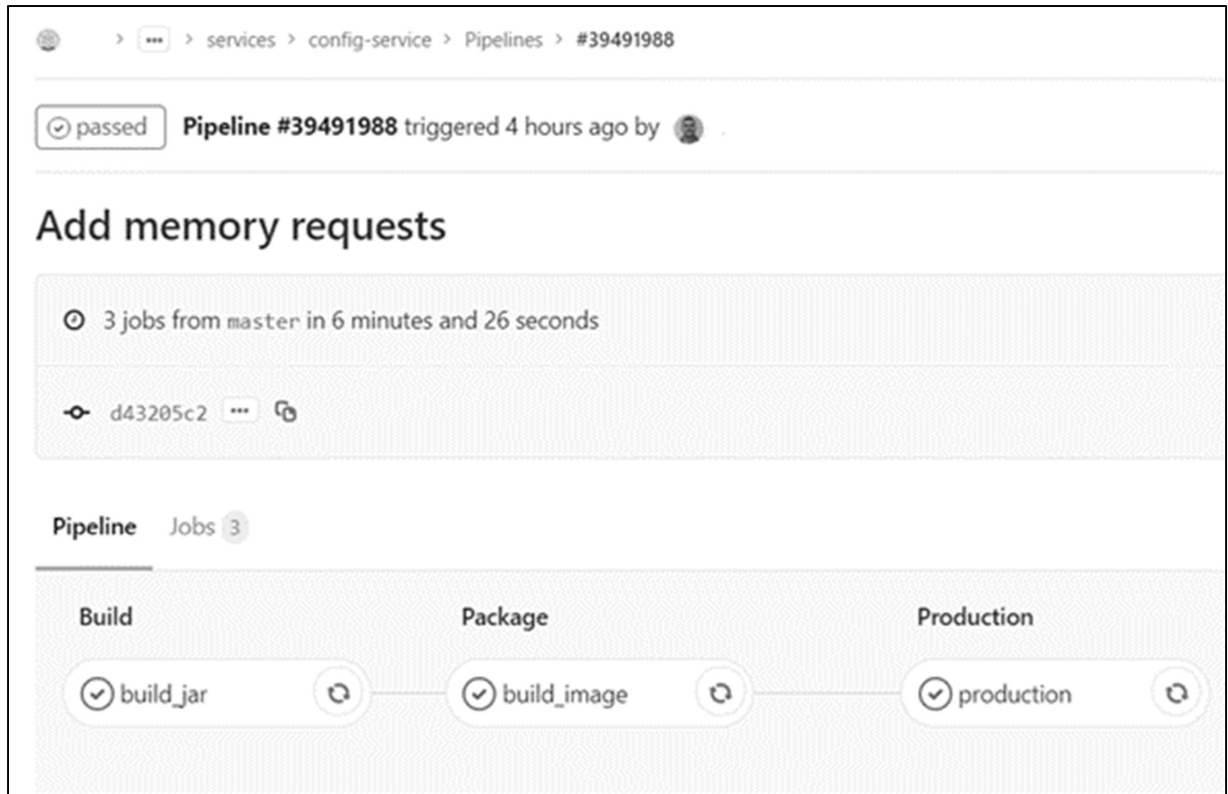

Рисунок 3.8 — Успішно пройдений Gitlab CI

Таким чином маємо готову систему з семи сервісів. Артефактами системи є докер образи. Процес CI/CD вставновлений засобами Gitlab. Розгортання відбувається у Google Cloud Platform використовуючи Google Kubernetes Engine.

3.4 Тестування програмної системи

Тестування програмного забезпечення являє собою процес технічного дослідження, що має на меті виявлення інформації щодо якості продукту відносно

оточення, в якому він має бути використаний. Техніка тестування включає в себе як процес пошуку невідповідностей, помилок, так і випробування програмних компонентів з метою їх оцінювання. При виконанні тестування програмна система може оцінюватися за наступними критеріями:

– коректна відповідь для усіх варіацій вхідних даних;

– дотримання вимог, якими керувалися розробники;

– практичність;

– виконання функцій за прийнятний час;

– сумісність з існуючим програмним забезпеченням та існуючими операційними системами.

Оскільки число теоретично можливих тестових випадків навіть для нескладних програмних компонент наближається до нескінченності, то тратегія тестування полягає проведенні усіх можливих тестів враховуючи наявні ресурси та час. У результаті ПЗ тестується стандартним виконанням програми з метою виявлення помилок або інших дефектів. Тестування програмного забезпечення може надавати об'єктивну, незалежну інформацію про ризики відмови та якість, як для кінцевих користувачів, так і для замовників. Тестування може проводитись одразу після створення виконуваного коду. Процес розробки повинен передбачати коли і яким чином буде проходити тестування. Наприклад, поетапний процес передбачає виконання більшості тестів після визначення системних вимог, далі вони реалізуються у тестових програмах. На противагу, відповідно до вимог гнучкої розробки програмного забезпечення, і програмування, і тестування не рідко виконується одночасно.

Для розробленої системи у якості методу тестування обрано функціональне тестування — це тестування програмного забезпечення з метою перевірки можливості функціональних вимог бути реалізованими, тобто здатності програмного забезпечення в певних умовах вирішувати завдяння необхідні кінцевим користувачам.

Функціональні вимоги визначаються, що саме робить програмне забезпечення, які завдання воно вирішує.

Для проведення функціонального тестування системи виділено на перевірено наступні вимоги:

– за наявності даних у таблиці persons GET запит до ендпоінту /api/person повинен отримати у відповідь масив об’єктів Person у форматі JSON, що відповідають даним у таблиці;

– за наявності даних у таблиці animals GET запит до ендпоінту /api/animal повинен отримати у відповідь масив об’єктів Animal у форматі JSON, що відповідають даним у таблиці;

– за наявності даних у таблиці cars GET запит до ендпоінту /api/car повинен отримати у відповідь масив об’єктів Car у форматі JSON, що відповідають даним у таблиці;

– за наявності даних у таблиці buildings GET запит до ендпоінту /api/building повинен отримати у відповідь масив об’єктів Building у форматі JSON, що відповідають даним у таблиці;

– POST запит до ендпоінту /api/graphql з валідним GraphQL запитом повинен отримати у відповідь результат виконання запиту;

– будь-який інший запит до ендпоінту не переліченого вище повинен отримати у відповідь 404 помилку з пустим тілом відповіді;

– якщо тип запиту до перелічених ендпоінтів не співпадає з зазначеним, то у відповідь повинна бути надіслана 405 помилка з пустим тілом відповіді.

Кожна з висунутих умов може бути розглянута у якості тестового випадку. Розглянемо дані тестові випадки.

Передумова: таблиця persons містить два записи. Запис 1 — id = 1, name = John. Запис 2 — id = 2, name = Bob. Очікуваний результат: GET запит до ендпоінту /api/person у відповідь отримає масив із двох елементів типу Person у форматі JSON. Значення полів кожного з елементів співпадають з значеннями у базі даних. Результат

тесту: GET запит до ендпоінту /api/person отримав у відповідь JSON масив із двох елементів типу Person у форматі JSON. Значення полів кожного з елементів співпали з значеннями у базі даних (див. рис. 3.9).

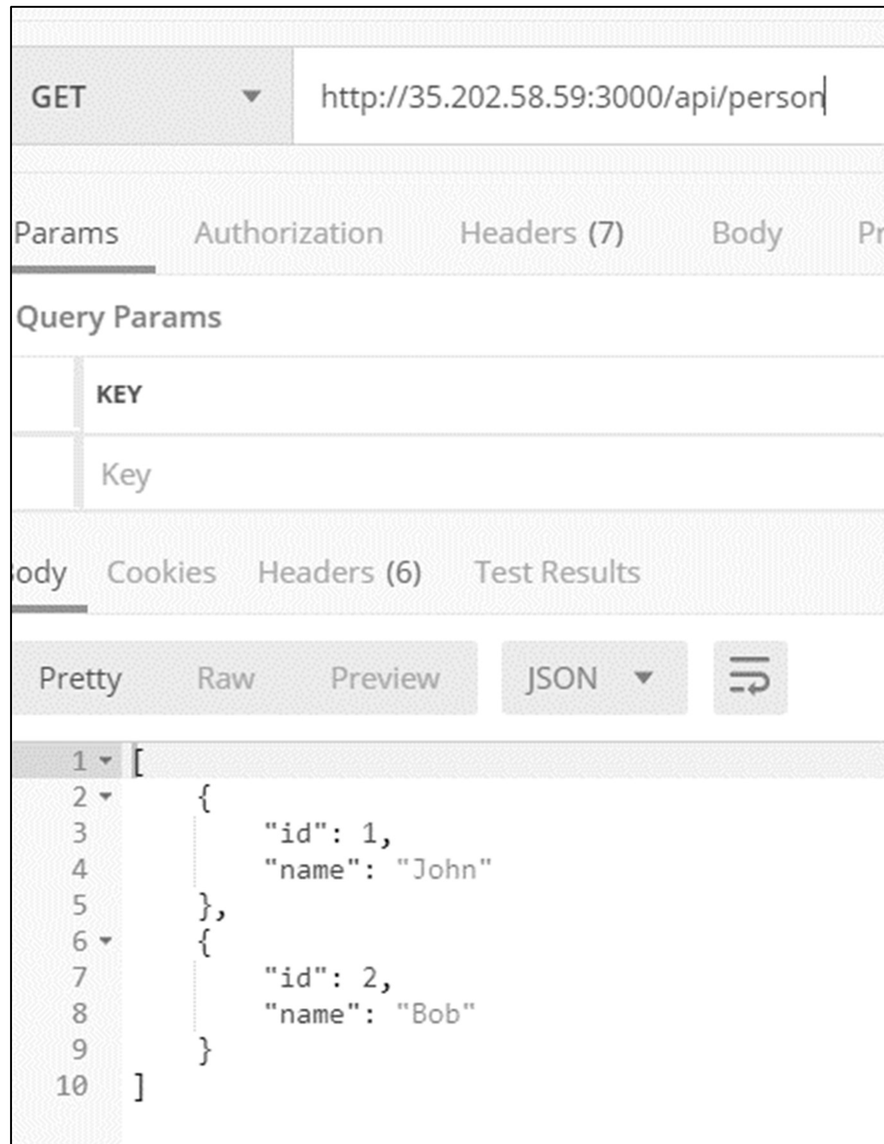

Рисунок 3.9 — Відповідь на GET запит до ендпоінту /api/person

Передумова: таблиця animals містить два записи. Запис 1 — type = mammal, name = cat. Запис 2 — type = insect, name = ant. Очікуваний результат: GET запит до ендпоінту /api/animals у відповідь отримає масив із двох елементів типу Animal у форматі JSON. Значення полів кожного з елементів співпадають з значеннями у базі

даних. Результат тесту: GET запит до ендпоінту /api/animals отримав у відповідь JSON масив із двох елементів типу Animal у форматі JSON. Значення полів кожного з елементів співпали з значеннями у базі даних (див. рис. 3.10).

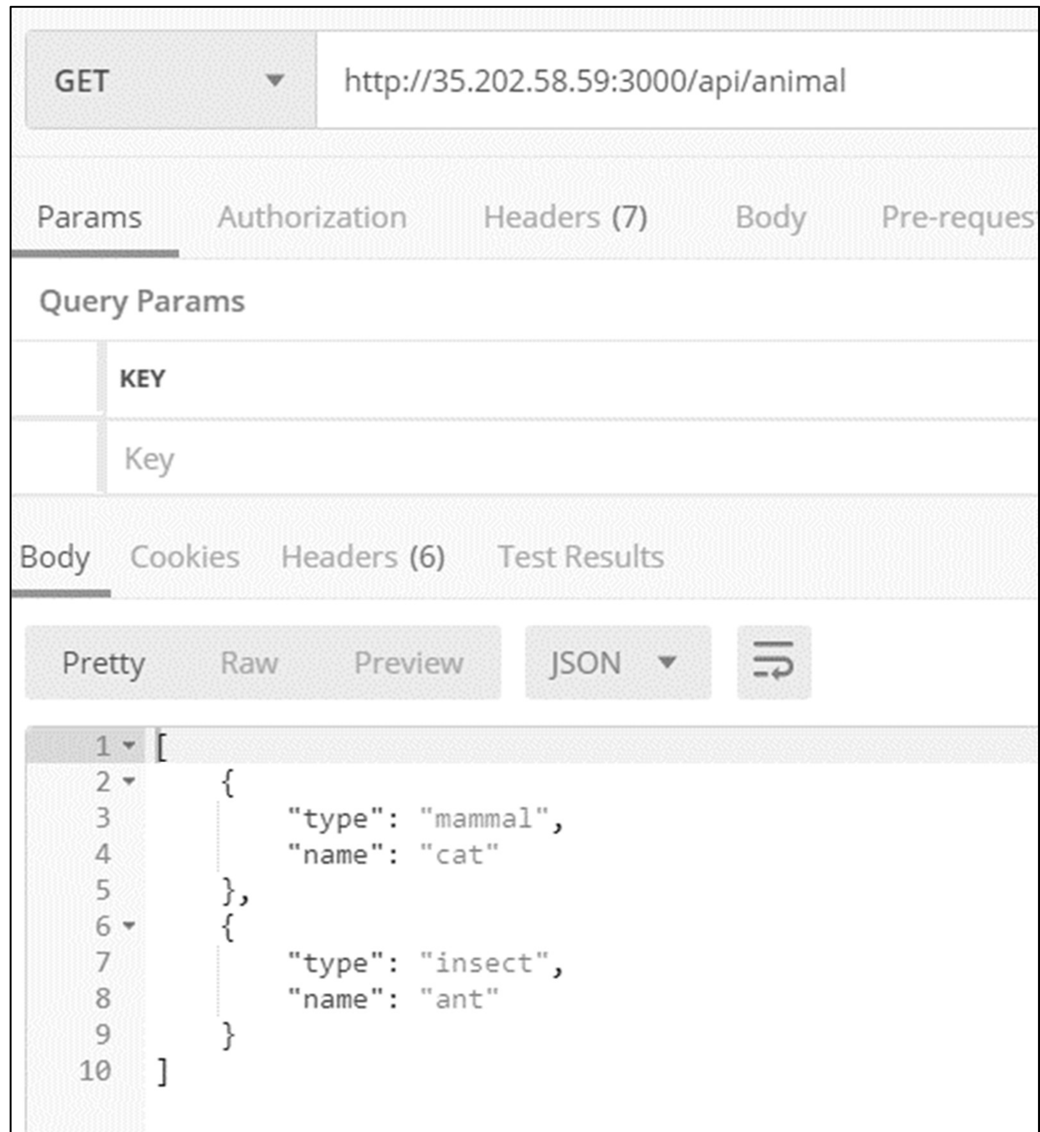

Рисунок 3.10 — Відповідь на GET запит до ендпоінту /api/animal

Передумова: таблиця cars містить два записи. Запис 1 — model = sedan, color = red. Запис 2 — model = coupe, color =white. Очікуваний результат: GET запит до ендпоінту /api/car у відповідь отримає масив із двох елементів типу Car у форматі JSON. Значення полів кожного з елементів співпадають з значеннями у базі даних.

Результат тесту: GET запит до ендпоінту /api/car отримав у відповідь JSON масив із двох елементів типу Car у форматі JSON. Значення полів кожного з елементів співпали з значеннями у базі даних (див. рис. 3.11).

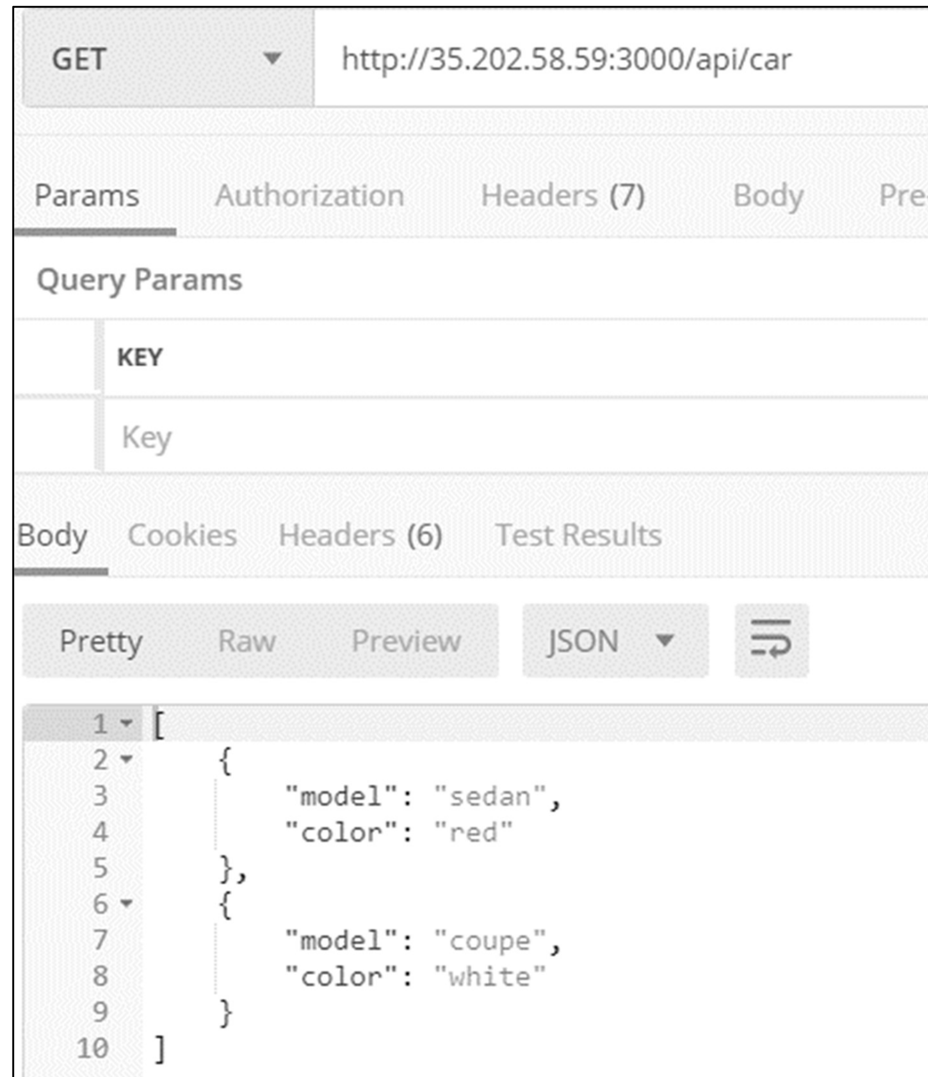

Рисунок 3.11 — Відповідь на GET запит до ендпоінту /api/car

Передумова: таблиця buildings містить два записи. Запис 1 — address = addr1, department = dep. Запис 2 — address = addr2, department =subdep. Очікуваний результат: GET запит до ендпоінту /api/building у відповідь отримає масив із двох елементів типу Building у форматі JSON. Значення полів кожного з елементів співпадають з значеннями у базі даних. Результат тесту: GET запит до ендпоінту /api/building

отримав у відповідь JSON масив із двох елементів типу Building у форматі JSON. Значення полів кожного з елементів співпали з значеннями у базі даних (див. рис. 3.12).

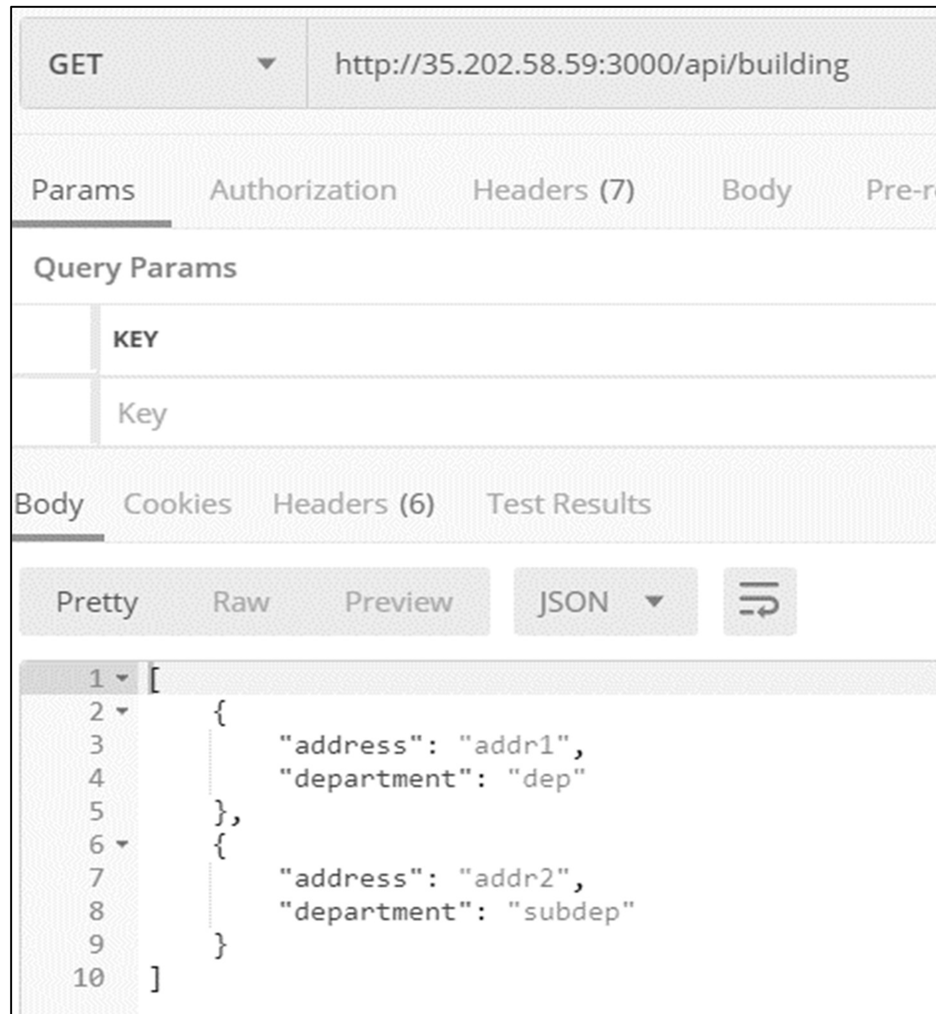

Рисунок 3.12 — Відповідь на GET запит до ендпоінту /api/building

Передумова: persons містить два записи. Запис 1 — id = 1, name = John. Запис 2 — id = 2, name = Bob. Очікуваний результат: POST запит до ендпоінту /api/graphql з тілом запиту «{ persons { id name } }» у відповідь отримає GraphQL відповідь у форматі JSON, яка буде містити два елементи типу Person. Значення полів кожного з елементів співпадають з значеннями у базі даних. Результат тесту: POST запит до ендпоінту /api/graphql отримав у відповідь GraphQL відповідь у форматі JSON з двома

елементами типу Person. Значення полів кожного з елементів співпали з значеннями у базі даних (див. рис. 3.13).

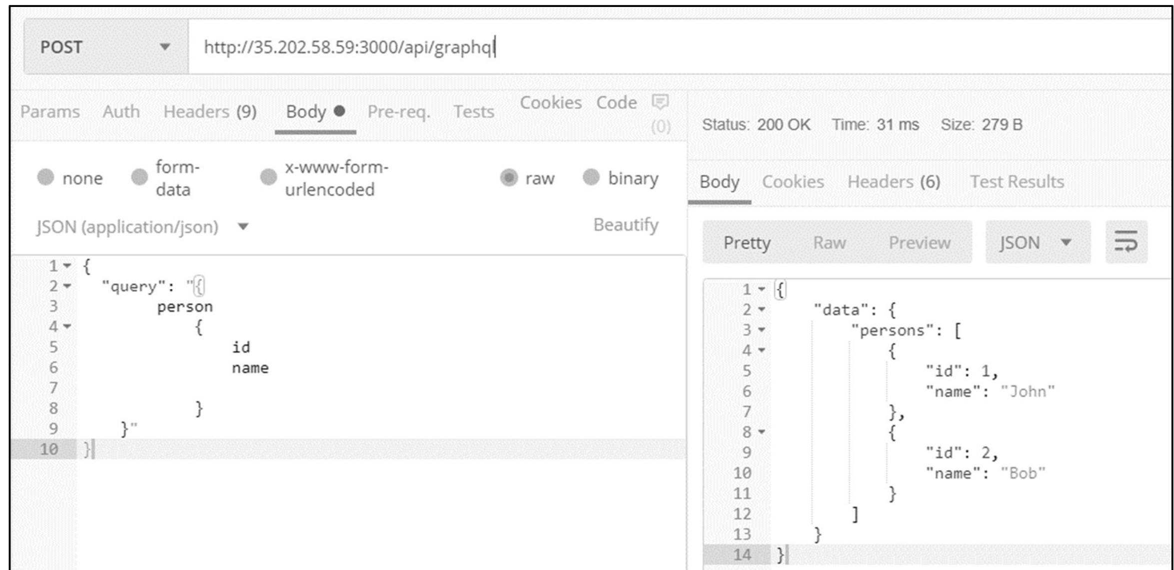

Рисунок 3.13 — Відповідь на POST запит до ендпоінту /api/graphql

Передумова: відсутня. Очікуваний результат: GET запит до ендпоінту /api/test у відповідь отримає HTTP код 404 та пусте тіло відповіді. Результат тесту: GET запит до ендпоінту /api/test отримав у відповідь HTTP код 404 та пусте тіло відповіді (див. рис. 3.14).

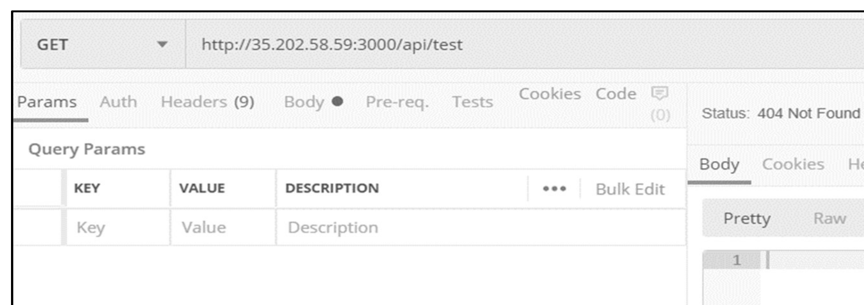

Рисунок 3.14 — Відповідь на GET запит до ендпоінту /api/test

Передумова: відсутня. Очікуваний результат: POST запит до ендпоінту /api/person у відповідь отримає HTTP код 405 та пусте тіло відповіді. Результат тесту:

POST запит до ендпоінту /api/person отримав у відповідь HTTP код 405 та пусте тіло відповіді (див. рис. 3.15).

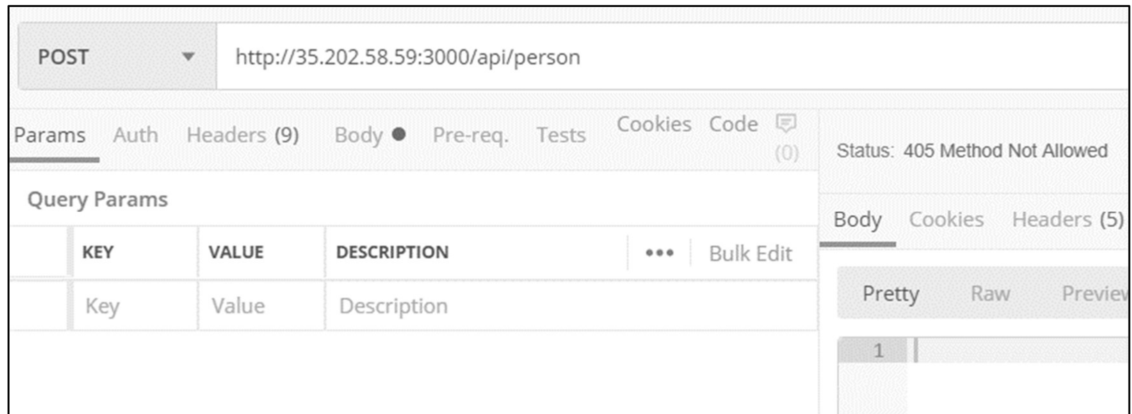

Рисунок 3.15 — Відповідь на POST запит до ендпоінту /api/person

Для розробленої програмної системи визначено перелік вимог для перевірки, створені тестові сценарії та проведено функціональне тестування. З результатів виконання тестування програмної системи можна зробити висновок, що система працює коректно та відповідає поставленим до неї вимогам.

4 ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ З ВИКОРИСТАННЯМ ОБРАНИХ ТЕХНОЛОГІЙ

4.1 Методика оцінювання ефективності

Методика оцінювання ефективності засобів і технологій міжсервісного зв'язку полягає у наступній послідовності дій.

1. Здійснити вибір критеріїв оцінювання ефективності засобів і технологій міжсервісного зв'язку.

2. Виконати моделювання продуктивності для обраних за критеріями технологій.

3. Розробити рекомендації щодо використання розглянутих технологій міжсервісного зв'язку.

4. Здійснити вибір технології, спираючись на значення найбільш вагомих для системи критеріїв.

В результаті дослідження засобів і технологій міжсервісної комунікації визначимо перелік основних критеріїв, за якими їх можливо оцінювати. На рисунку 4.1 наочно показано відповідність кожної з технологій обраним критеріям враховуючи відсутність додаткових втручань у реалізацію, що дозволяє виділити найкращі технології, спираючись на задовільнення обраним критеріям.

Відповідність критеріям встановлена в залежності від того, чи підтримуються вони технологією без прикладання додаткових зусиль з точки зору розробки та проектування. Кожен з критеріїв може мати різну вагу при виборі технології в залежності від вимог, що висуваються до системи. Таким чином, якщо однією з вимог до системи є жорстка стандартизація, використання XML прийнятне, а продуктивність не є першочерговою вимогою до системи, то SOAP буде найліпшим кандидатом, так як SOAP пропонує структурований спосіб відправки повідомлень між

службами зі строгими правилами, що визначають структуру повідомлень і способи їх обробки.

	REST	GraphQL	SOAP	Apache Thrift	gRPC
Мультиплатформність	+	+	+	—	—
Незалежність від формату даних	+	+	—	+	—
Стандартизація	—	—	+	—	—
Когнітивна зрозумілість	+	+	—	+	+
Простота використання	+	—	—	—	+
Розповсюдженість	+	+	—	—	+
Версіонування	+	—	+	+	—
Кешування	+	—	+	—	—

Рисунок 4.1 — Задовільнення технологій обраним критеріям

Якщо продуктивність все ж є важливою вимогою, то SOAP не повинен бути обраним, так як він прив'язаний до використання XML, який, безумовно, має найгіршу продуктивність із усіх розглянутих форматів серіалізації. Високу продуктивність пропонують RPC технології, до яких відносяться Apache Thrift та gRPC. Обидві технології зрозумілі при використанні, за тією лише відмінністю, що gRPC дозволяє використовувати Protocol Buffer не тільки у якості формату серіалізації та десеріалізації, а і у якості мови визначення інтерфейсу. Проте Apache Thrift позбавлений підтримки стрімінгу великих обсягів даних. Тож обираючи між gRPC та Apache Thrift варто опиратися на необхідність використання Protocol Buffer та необхідність великих обсягів даних. Незважаючи на високу продуктивність, RPC протоколи мають ряд недоліків, серед яких: тісну зв’язаність, слабку когнітивну зрозумілість.

Якщо важлива зрозумілість використання та мультиплатформовість технології то REST може бути використаний для встановлення міжсервісного зв'язку. REST став однією з найбільш популярних альтернатив комунікації між сервісами завдяки тому, що він легкий і може бути прийнятий будь-якою мовою. GraphQL пропонує новий спосіб запиту даних у форматі, який легко зрозуміти. Також технологія дозволяє передавати лише необхідні дані, що позитивно впливає на обсяг трафіку, що проходить мережею.

GraphQL пропонує абстрактне уявлення про ресурси даних як вузли в графі, де будь-які вузли можуть бути об'єднані для формування запиту. GraphQL був побудований для об'єднання даних в єдиний сервіс і дуже добре відповідає шаблону API шлюзу (api gateway), але додає небажаних зайвих залежностей до системи, оскільки для цього потрібна структура для розбору запитів і створення відповідей на них. Якщо фреймворк GraphQL може проаналізувати і серіалізувати дані без великих втрат продуктивності в порівнянні з архітектурою REST з використянням JSON у якості формату серіалізації та десеріалізації, це може бути життєздатною альтернативою.

4.2 Моделювання продуктивності технологій міжсервісного зв'язку

На основі аналізу відповідності технологій критеріям, що наведені на рисунку 4.1, можна зробити висновок, що найбільш підходящими для використання є GraphQL та REST.

Моделювання продуктивності для обраних технологій здійснюється за допомогою розробленої програмної системи. Для виконання порівняння продуктивності обраних засобів комунікації необхідно виконати моделювання продуктивності, тобто набір тестів з різним обсягом даних [22].

Перший тест складається з отримання досить невеликої кількості даних від кожного сервісу з використання інтерфейсів REST і GraphQL. Таблиці містять по 5 записів. Затримка під час отримання даних від системи, розміщеної у хмарному середовищі, може сильно відрізнятися від запиту до запиту. Для отримання достовірних результатів надіслано 100 запитів один за одним з секундною затримкою між ними. Затримка впроваджується для того щоб бути впевненим, що навантаження на систему має мінімальний вплив на результат. 1 секунда обрана як порогове значення для часу виконання запиту. За час відповіді взято повний час знаходження запиту у системі, включаючи час, який займає виконання обробки запиту мікросервісом та включаючи час на витяг та обробку даних з бази даних. Час відповіді для кожного запиту до кожного із інтерфейсів зображено на рисунку 4.2.

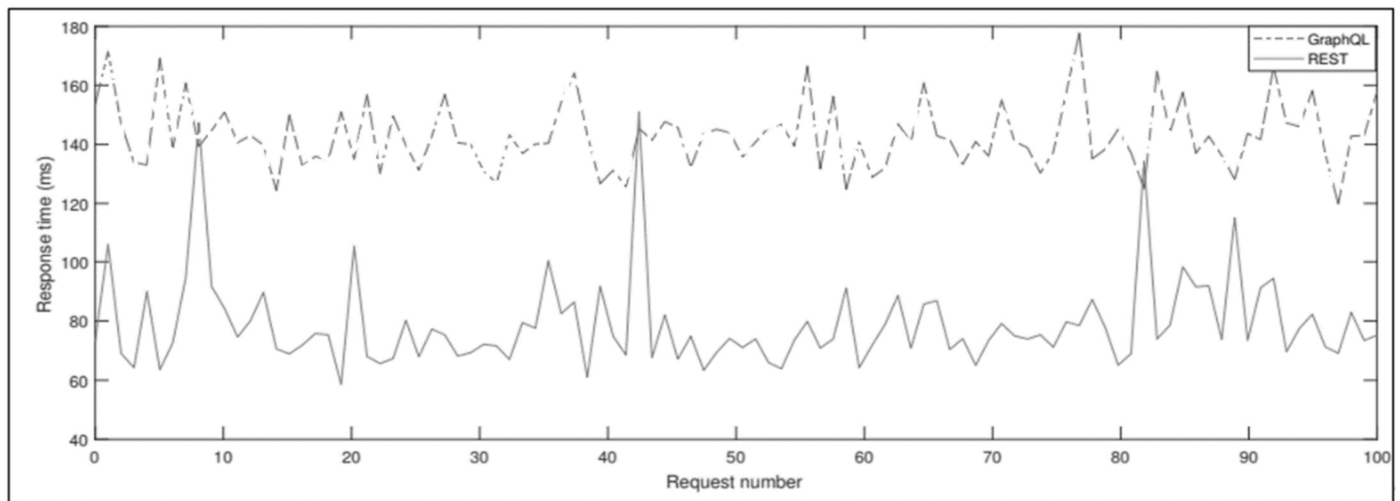

Рисунок 4.2 — Час відповіді при запиті даних за допомого GraphQL та REST інтерфейсів

По горизонтальній осі розміщені запити у порядку зростання їх порядкового номеру. По вертикальній осі відкладено час виконання кожного з запитів у мілісекундах. Цільну лінію утворюють REST запити, а пунктирною лінією позначено GraphQL запити. Комунікація з системою засобами REST має нижчий час відповіді у порівнянні з GraphQL. Найшвидший час відповіді для REST становить 58,53 мс, а

найповільніший — 151,08 мс. Взаємодія з використанням GraphQL має значно гірший результат з найшвидшим часом відгуку у 119,86 мс і найповільнішим часом відгуку у 177,72 мс.

Порівняння середнього часу відгуку для REST і GraphQL показано на рисунку 4.3.

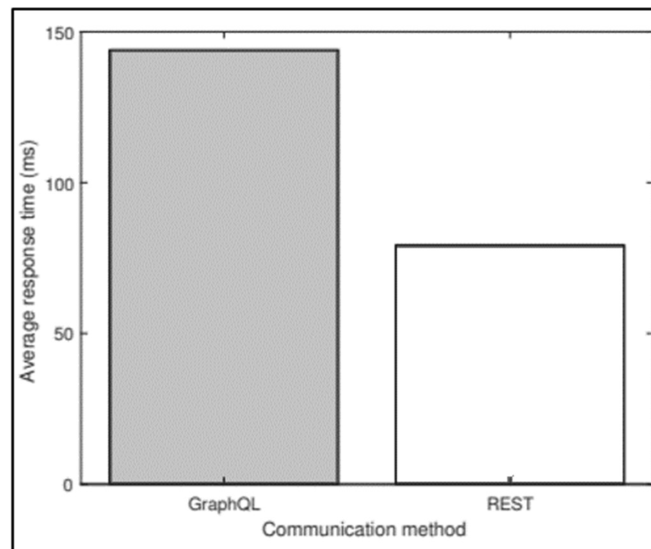

Рисунок 4.3 — Порівняння середнього часу відгуку для GraphQL та REST

По горизонтальній осі розміщені методи взаємодії, а по вертикальній осі відповідний середній час виконання запиту. Маємо, що REST отримав середній час відгуку рівний 78,94 мс, а GraphQL має середнє значення у 142,92 мс. Це робить GraphQL на 81,05% повільніше в середньому.

Другий тестовий випадок використовує теж саме тестове оточення що і попередній, але у цьому випадку виміряна продуктивність при обробці запитів, що містять більший об’єм даних. Кожна з чотирьох таблиць, що запитуються мікросервісами, розширена до 25 записів.

Заміряний час відгуку у цьому тесті показав, що для GraphQL затримка помітно зростає зі збільшенням обсягу даних. Заміри часу для 100 виконанних запитів з

використанням кожного с інтерфейсів, що приймають участь при моделюванні зображені на рисунку 4.4.

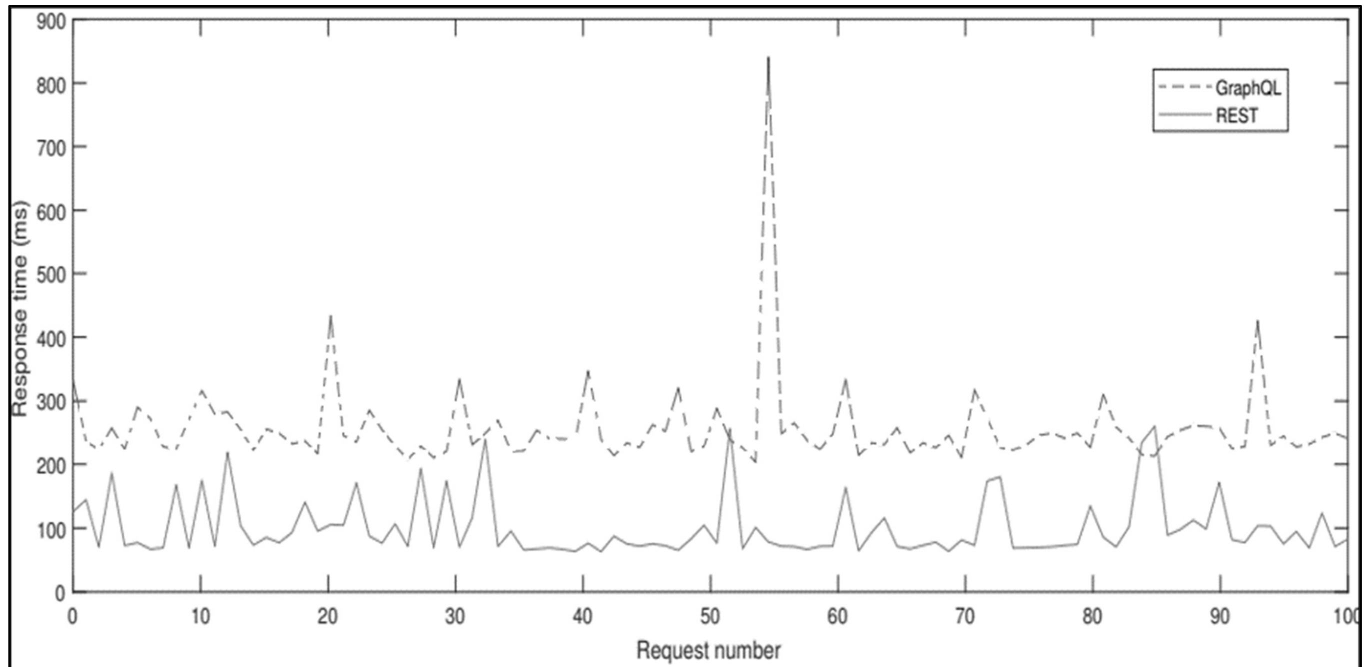

Рисунок 4.4 — Час відповіді при запиті даних для другого тестового випадку

Найшвидший час відповіді с використанням REST становить 63,04 мс, а найповільніший становить 260,39 мс. GraphQL не співпадає у цифрах з REST і має найшвидший час відгуку у 203,00 мс і найповільніший часом відгуку у 840,56 мс. Порівняння середнього часу відгуку при виконанні запитів для REST і GraphQL показано на рисунку 4.5.

Отримання даних через інтерфейс REST зайняло у середньому 100,35 мс, а вибірка даних за допомогою GraphQL в середньому становила 256,80 мс. Тож маємо, що отримання даних за допомогою GraphQL у другому тестовому випадку було в середньому на 256% повільніше, ніж вибірка даних з використанням REST.

Для третього тестового випадку збільшено кількість тестових даних до 50 записів у кожній з таблиць до яких звертаються мікросервіси. Процедура виконання

викликів та оточення залишається таким же, як і для попередніх сценаріїв моделювання.

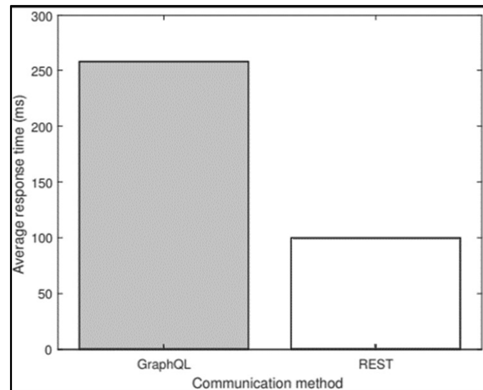

Рисунок 4.5 — Порівняння середнього часу відгуку GraphQL та REST для другого тестового випадку

Результати запитів для третього тестового випадку доводять, що фраемворк GraphQL не справляється у конкуренції за продуктивність з архітектурою REST, як і при виконанні попередніх сценаріїв. Час відповіді для GraphQL та REST відображений на рисунку 4.6.

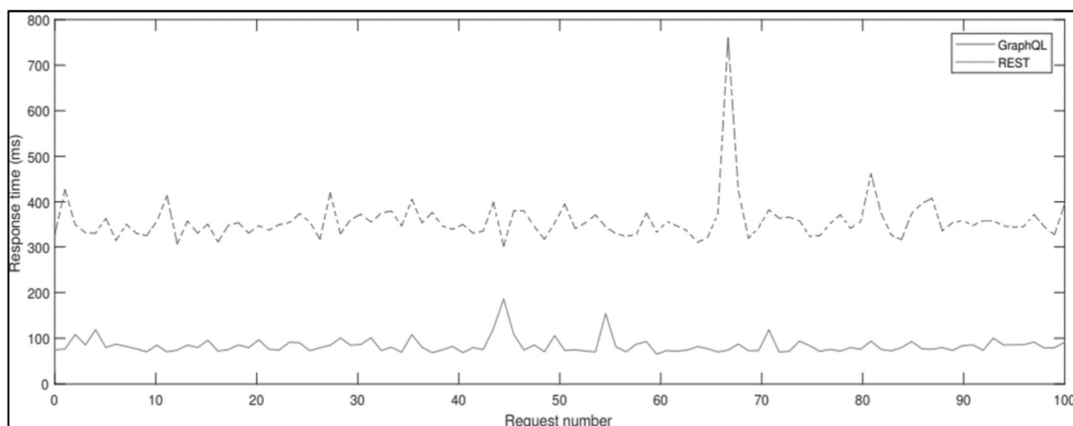

Рисунок 4.6 — Час відповіді при запиті даних для третього тестового випадку

Для REST мінімальний час відповіді становить 65,25 мс, максимальний — 185,94 мс. Час для GraphQL значно гірший: мінімальна затримка — 301,14 мс, максимальна

затримка — 759,84 мс. Порівняння середнього часу відгуку для REST і GraphQL у третьому тестовому випадку показано на рисунку 4.7.

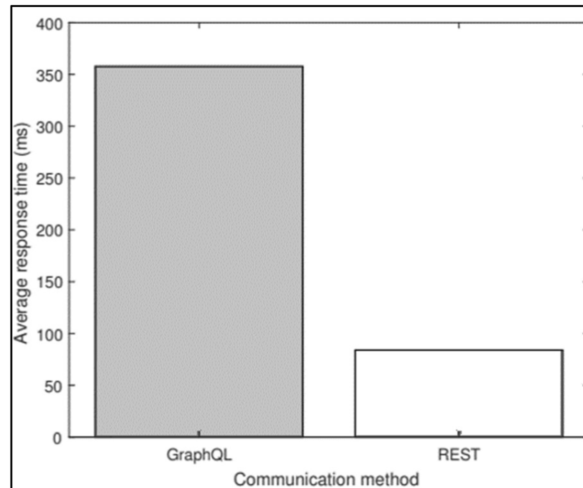

Рисунок 4.7 — Порівняння середнього часу відгуку для третього тестового випадку

Помітне зниження продуктивності при використанні REST у цьому тестовому випадку у порівнянні з другим тестовим випадком, проте середній час відповіді REST фактично швидше, ніж у другому тестовому випадку — 84 мс. Середній час відповіді для GraphQL в цьому тесті склав 357,89 мс. Тобто середній час для GraphQL на 426% більший, ніж відповідь REST та на 179 відсотків більший у порівнянні з тестом 2.

4.3 Рекомендації по використанню технологій міжсервісного зв’язку

Результати моделювання продуктивності показують, що GraphQL не може конкурувати з архітектурою REST з використанням JSON формату з точки зору часу відгуку. Навіть при отриманні незначного обсягу даних обробка GraphQL викликає просідання часу відповіді. Так як GraphQL прив'язаний до використання JSON у якості формату повідомлень, то він обмежений у техніках серіалізації.

Що пропонує GraphQL замість високої продуктивності, так це набагато простіший інтерфейс вибору необхідних даних, який включає у відповідь лише поля, які запитуються. Проблема незручності REST інтерфейсу полягає у тому, що кожен ресурс відображається на URI і якщо різні програми-споживачі мають потребу у специфічній структурі даних, то для кожного такого випадку необхідно створювати новий специфічний REST ендпоінт.

У противагу, при використанні GraphQL потрібно створити тільки докладну структуру, що описує наявні типи і поля. Використовуючи такий ресурс кожен клієнт-споживач може точно вказати, які поля він бажає отримати. Це виключає великі інтерфейси, які важко підтримувати і які не мають чітких правил щодо того, як обробляти декілька версій одного API. У сценарії, коли запитується невеликий обсяг даних, GraphQL є гарною ідеєю для використання. Але якщо продуктивність є важливим критерієм при виборі технології, то GraphQL не є сильним претендентом.

Архітектура REST має можливість навіть кращої продуктивності, з точки зору меншого часу відгуку і меншого розміру повідомлення, якщо замість формату JSON використовувати двійковий протокол серіалізації. Проте неможливо сказати, що використання фреймворку RPC, такого як gRPC або Thrift, і бінарного протоколу серіалізації краще в архітектурі мікросервісів. Використання таких фреймворків і бінарної серіалізації змушує відмовитися від такої свободи мікросервісів, як справжня толерантність до кросплатформовості. В кінцевому підсумку вибір технології взаємодії в мікросервісній архітектурі є компромісом: обмеження використання системою конкретної мови та інструменту, більш ефективна комунікація, складніша реалізація.

ВИСНОВКИ

Атестаційна робота присвячена актуальній темі — дослідженню засобів і технологій міжсервісного зв'язку в мікросервісній архітектурі програмних систем. В процесі виконання атестаційної роботи проведено алаліз предметної галузі та виявлено існуючі проблеми міжсервісної комунікації.

Розглянуті існуючі технології міжсервісної комунікації та засоби серіалізації і десеріалізації даних, виявлені їх недоліки та переваги.

Розроблена методика оцінювання ефективності засобів і технологій міжсервісного зв'язку, сформульовані критерії оцінювання ефективності засобів комунікації та їх вплив на вибір засобу.

Розроблена програмна система, що дозволяє провести моделювання продуктивності обраних технологій міжсервісної комунікації.

За допомогою розробленої програмної системи проведено дослідження продуктивності наступних засобів міжсервісної комунікації: REST та GraphQL з використанням JSON в якості формату серіалізації та десеріалізації даних.

Базуючись на результатах проведеного дослідження, виявлено найбільш ефективні технології міжсервісного зв'язку.

Розроблено рекомендації щодо впровадження технологій міжсервісного зв'язку з урахуванням ваги кожного з обраних критеріїв.

Результати роботи дозволять організовувати мікросервісну взаємодію найбільш ефективним чином, що значною мірою підвищить продукутивність розроблюваних програмних систем та покращить процес користування системами.

Отримані результати пройшли апробацію в рамках двадцять третього міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті». Також за результатами досліджень опублікована стаття в міжнародному науковому електроному журналу «Наука онлайн»

ПЕРЕЛІК ПОСИЛАНЬ

1. М. Фаулер. Архитектура корпоративных программных приложений / М. Фаулер. – Издательский дом Вильямс, 2006 – 544 с.

2. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans – Addison-Wesley, 2003 – p. 560

3. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб: Питер, 2016 – 304 с.

4. М. Синкевич. Технологии межсервисной связи в микросервисной архитектуре программных систем // Матеріали XXIII міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті» 16-18 квітня 2019р. Том 6 — 151 с.

5. Richardson Maturity Model [Електронний ресурс]. — Режим доступу: https://martinfowler.com/articles/richardsonMaturityModel.html (дата звернення 20.04.2019).

6. The javascript object notation(json) data interchange format rfc 7159 (ietf) [Електронний ресурс]. — Режим доступу: https://tools.ietf.org/html/rfc7159 (дата звернення: 20.04.2019).

7. Extensible markup language (xml) 1.0 (fifth edition) [Електронний ресурс]. — Режим доступу: https://www.w3.org/TR/REC-xml (дата звернення: 14.05.2019)

8. Apache thrift documentation [Електронний ресурс]. — Режим доступу: https://thrift.apache.org (дата звернення: 14.05.2019).

9. Protocol buffers documentation [Електронний ресурс]. — Режим доступу: https://developers.google.com/ protocol-buffers (дата звернення: 14.05.2019).

10. Popic et al. Performance evaluation of using protocol buffers in the internet of things communication. In 2016 International Conference on Smart Systems and Technologies (SST), 2016.

11. K. Maeda. Performance evaluation of object serialization libraries in xml, json and binary formats. In Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on, 2012.

12. S. Kami Makki A. Sumaray. A comparison of data serialization formats for optimal efficiency on a mobile platform. In ICUIMC '12 Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, 2012.

13. Hypertext Transfer Protocol Bis (httpbis) [Електронний ресурс]. — Режим доступу: https://datatracker.ietf.org/wg/httpbis/charter (дата звернення: 29.04.2019).

14. Remote procedure call [Електронний ресурс]. — Режим доступу https://en.wikipedia.org/wiki/ Remote_procedure_call (дата звернення: 01.05.2019)

15. J. Grabis J. Tihomirovs. Comparison of soap and rest based web services using software evaluation metrics. Information Technology and Manage- ment Science 19 — p. 92

16. A data query language [Електронний ресурс]. — Режим доступу http://graphql.org/blog/ graphql-a-query-language (дата звернення: 01.05.2019)

17. Axios on github [Електронний ресурс]. — Режим доступу https://github.com/mzabriskie/axios (дата звернення: 01.05.2019)

18. Spring Framework [Електронний ресурс]. — Режим доступу: http://spring-projects.ru/projects/spring-framework (дата звернення: 05.05.2019).

19. GitHub - Netflix/zuul: Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more. [Електронний ресурс]. — Режим доступу: https://github.com/Netflix/zuul (дата звернення 10.05.2019).

20. GitHub - spring-cloud/spring-cloud-config: External configuration (server and client) for Spring Cloud [Електронний ресурс]. — Режим доступу: https://github.com/spring-cloud/spring-cloud-config (дата звернення 10.05.2019).

21. GitHub - Netflix/eureka: AWS Service registry for resilient mid-tier load balancing and failover. [Електронний ресурс]. — Режим доступу: https://github.com/Netflix/eureka (дата звернення 10.05.2019).

22. М. Синкевич, Н. Лєсна. Дослідження засобів і технологій міжсервісного зв’язку в мікросервісній архітектурі. International Electronic Scientific Journal «Science Online», 2019