

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

ННК «інститут прикладного системного аналізу»

Кафедра Системного проектування

«До захисту допущено»

Завідувач кафедри

_____ В. Є. Мухін

«___» _____ 2021 р.

Дипломна робота

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Інтелектуальні сервіс-
орієнтовані розподілені обчислювання»**

спеціальності 122 «Комп'ютерні науки»

**на тему: «Аналіз та практичне застосування фреймворків Java при
розробці програм на основі мікросервісів»**

Виконав:

студент IV курсу, групи IT-ZP91

Назаренко Максим Анатолійович

Керівник:

доцент кафедри СП, к.т.н.

Булах Б. В.

Консультант з економіки:

доцент, к.е.н.

Рощина Н. В.

Рецензент:

в.о. зав. кафедри ММСА, к.т.н.

Тимощук О. Л.

“Simplicity is prerequisite for reliability.”

Edsger W. Dijkstra

Аналіз та практичне застосування фреймворків Java при розробці програм на основі мікросервісів

Назаренка Максима Анатолійовича

Анотація

У дипломній роботі розглянуто мікросервісну архітектуру, мікросервісні фреймворки мови програмування Java та розроблено два мікросервіси, що реалізують функціонал обміну повідомленнями між собою для типових задач. Досліджено історичні передумови, що призвели до появи мікросервісної архітектури, проведено огляд ринку мікросервісних фреймворків для мови програмування Java.

Розглянуто переваги та недоліки популярних мікросервісних фреймворків мови програмування Java.

Було розроблено два мікросервіси, що надають користувачу можливість обміну інформацією про складські приміщення та товари. Було протестовано REST API мікросервісів.

Загальний об'єм роботи: 93 сторінки, 48 рисунків, 8 таблиць, 20 посилань.

Ключові слова: Java, мікросервісні фреймворки, Spring Boot, веб-додатки, архітектура програмного забезпечення, мікросервіси, REST API.

Analysis and practical application of Java frameworks in development of microservice based applications

by Maksym Nazarenko

Abstract

In the thesis was considered the microservice architecture, microservice frameworks of the Java programming language and developed two microservices that implement the functionality of exchanging messages with each other for typical tasks. Has been studied the historical preconditions that led to the emergence of microservice architecture and has been reviewed the market of microservice frameworks for Java programming language.

Also, has been considered advantages and disadvantages of popular microservice frameworks of the Java programming language.

At the end, have been developed two microservices that allow the user to exchange information about warehouses and goods. Has been tested the REST API of microservices.

Total volume of work: 93 pages, 48 figures, 8 tables, 20 links.

Keywords: Java, microservice frameworks, Spring Boot, web applications, software architecture, microservices, REST API.

Подяка

Я хотів би подякувати моїй родині, друзям та колегам, які підтримували мене все життя через усі труднощі. Також висловлюю подяку Інституту прикладного системного аналізу за отримані знання та досвід, а також особисто Булаху Богдану Вікторовичу за його час, ідеї та допомогу.

Acknowledgements

I would like to thank family, friends and colleagues which have been supporting me all my life through all the difficulties. I also wish to thank the Institute for Applied System Analysis for knowledge and experience and personally to Bogdan Bulakh for his time, ideas and help.

ЗМІСТ

Завдання	i
Анотація	iii
Подяка	v
Перелік рисунків	ix
Перелік таблиць	xi
Список скорочень та визначень	xii
1 Вступ	1
2 Еволюція розробки програмного забезпечення	5
2.1 Монолітна архітектура	5
2.2 Об'єктно орієнтоване програмування	8
2.3 Сервіс-орієнтована архітектура	8
2.4 Сучасні потреби ринку до розробки	9
3 Мікросервіси	11
3.1 Характеристика мікросервісної архітектури	12
3.2 Порівняння мікросервісів та монолітів	19
3.3 Порівняння мікросервісів та SOA	20
3.3.1 Систематика сервісів	21
3.3.2 Розмір сервісів	23
3.3.3 Взаємодія команд розробки сервісів	23
3.4 Переваги мікросервісної архітектури	25

4	Мікросервісні Java фреймворки	29
4.1	Spring Boot	30
4.2	Dropwizard	34
4.3	WildFly Swarm	37
4.4	Eclipse Vert.X	39
4.5	Quarkus	42
4.6	Micronaut	44
5	Застосування мікросервісних фреймворків	46
5.1	Spring Boot	46
5.2	Quarkus	51
5.3	Micronaut	56
5.4	Вибір мікросервісного фреймворку для розробки програми	60
5.5	Розробка мікросервісів за допомогою Spring Boot	62
6	Функціонально-вартісний аналіз програмного продукту	71
6.1	Постановка задачі техніко-економічного аналізу	72
6.1.1	Обґрунтування функцій програмного продукту	73
6.1.2	Варіанти реалізації основних функцій	74
6.2	Обґрунтування системи параметрів програмного продукту	77
6.2.1	Опис параметрів	77
6.2.2	Кількісна оцінка параметрів	77
6.2.3	Аналіз експертного оцінювання параметрів	80
6.3	Аналіз рівня якості варіантів реалізації функцій	84
6.4	Економічний аналіз варіантів розробки програмного продукту	85
6.5	Висновки	90
7	Висновки та пропозиції	92
	Список використаних джерел	94

Перелік рисунків

2.1	Схема монолітної архітектури	6
3.1	Куб масштабованості ПЗ	16
3.2	X-масштабування	16
3.3	Z-масштабування	17
3.4	Y-масштабування	18
3.5	Таксономія сервісу в мікросервісній архітектурі	21
3.6	Таксономія сервісу в сервіс орієнтованій архітектурі	22
3.7	Взаємодія команд розробки мікросервісів	24
3.8	Взаємодія команд розробки сервіс-орієнтованої архітектури	25
3.9	Організація мікросервісної архітектури	27
4.1	Популярність пошукових запитів Google на тему Spring та Spring Boot .	33
4.2	Популярність пошукових запитів Google на тему Spring Boot та Dropwizard	34
4.3	Популярність пошукових запитів Google на тему Spring Boot та WildFly Swarm	37
4.4	Популярність пошукових запитів Google на тему Spring Boot та Vert.x .	40
4.5	Популярність пошукових запитів Google на тему Spring Boot та Quarkus	43
4.6	Популярність пошукових запитів Google на тему Spring Boot та Micronaut	44
5.1	Стартова сторінка spring initializr	47
5.2	Початкова структура проекту Spring Boot	48
5.3	Код контролера Spring Boot	48
5.4	Кількість класів програми на Spring Boot	49
5.5	Розмір heap пам'яті програми на Spring Boot	49
5.6	GET запит до програми на Spring Boot	50

5.7	Стартова сторінка конфігуратора Quarkus	52
5.8	Початкова структура проекту Quarkus	52
5.9	Код контролера Quarkus	53
5.10	Кількість класів програми на Quarkus	53
5.11	Розмір heap пам'яті програми на Quarkus	54
5.12	GET запит до програми на Quarkus	54
5.13	Стартова сторінка Micronaut launch	56
5.14	Початкова структура проекту Micronaut	57
5.15	Код контролера Micronaut	57
5.16	Кількість класів програми на Micronaut	58
5.17	Розмір heap пам'яті програми на Micronaut	58
5.18	GET запит до програми на Micronaut	59
5.19	Ініціалізація проекту Spring Boot	63
5.20	Кількість класів мікросервісу складів	65
5.21	Кількість класів мікросервісу товарів	66
5.22	Розмір heap пам'яті мікросервісу складів	66
5.23	Розмір heap пам'яті мікросервісу товарів	67
5.24	Створення складу через POST запит до мікросервісу складів	68
5.25	Стягнення даних про склад через GET запит до мікросервісу складів . .	68
5.26	Створення товару через POST запит до мікросервісу товарів	69
5.27	Стягнення даних про товар через GET запит до мікросервісу товарів . .	70
6.1	Морфологічна карта варіантів реалізації функцій	74
6.2	Кількість класів	78
6.3	Об'єм пам'яті	79
6.4	Час запуску	79
6.5	Потенційний об'єм програмного коду	80

Перелік таблиць

3.1	Ключові відмінності між мікросервісною та монолітною архітектурою .	20
5.1	Порівняння мікросервісних фреймворків за метриками	60
6.1	Позитивно-негативна матриця варіантів основних функцій	75
6.2	Основні параметри програмного продукту	78
6.3	Результати ранжування показників	81
6.4	Результати ранжування параметрів	82
6.5	Розрахунок вагомості параметрів	83
6.6	Розрахунок показників якості	85

Список скорочень та визначень

БД	–	База Даних
ІС	–	Інформаційна Система
ПЗ	–	Програмне Забезпечення
API	–	Application Programming Interface прикладний програмний інтерфейс
AWS	–	Amazon Web Services веб-сервіси Amazon
Bean	–	класи написані мовою Java, що відповідають певному набору правил використовуються для об'єднання кількох об'єктів в один для зручної передачі даних
BPEL	–	Business Process Execution Language мова виконання бізнес процесів
CPU	–	Central Processing Unit центральний процесор
DevOps	–	Development & Operations взаємодія практик розробки і обслуговування
DI	–	Dependency Injection впровадження залежності
EAR	–	Enterprise Application aRchive Архів застосунку Java EE
EAS	–	Enterprise Application Software корпоративне програмне забезпечення
HTTP	–	Hypertext Transfer Protocol протокол передачі гіпертекстових документів
IDE	–	Integrated Development Environment інтегроване середовище розробки

IoC	–	Inversion of Control інверсія керування
JAR	–	Java Archive Java архів
Java	–	Мова програмування Java
JDK		Java Development Kit набір розробки для Java
JSP	–	JavaServer Pages технологія, що дозволяє динамічно генерувати HTML, XML та інші веб-сторінки
JVM	–	Java Virtual Machine віртуальна машина Java
MSA	–	MicroService Architecture мікросервісна архітектура
REST	–	Representational State Transfer підхід до архітектури мережеских протоколів, які надають доступ до інформаційних ресурсів
SOA	–	Service Oriented Architecture сервіс-орієнтована архітектура
SOAP	–	Simple Object Access Protocol простий протокол доступу до об'єктів
SSH	–	Secure Shell мережеский протокол рівня застосунків, що дозволяє проводити віддалене управління комп'ютером і тунелювання TCP-з'єднань
URL	–	Uniform Resource Locator єдиний вказівник на ресурс
WAR	–	Web Application Archive Архів веб застосунку
WSDL	–	Web Services Description Language мова опису інтерфейсів вебсервісу
XML	–	Extensible Markup Language розширювана мова розмітки

Dedicated to my beloved family and best friends.

Розділ 1

Вступ

Сучасні умови ведення бізнесу, проведення операцій та вирішення процесів постійно змінюються та адаптуються до потреб. Інтернет став рушієм та домінантною силою для компаній та організацій. Для підтримки змін цих процесів потрібним стала зміна підходів до дизайну та розробки систем. При проектуванні програмних продуктів інженери мають бажання побудувати програму, що буде релевантною для поточних та прогнозованих майбутніх потреб бізнесу. Однак, програми мають тенденцію з часом ставати складнішими та більшими. Впровадження незначних змін до програми може призвести до неочікуваного та небажаного ефекту на цілу програму. Складність і потреба в адаптації до світу, що постійно змінюється, відобразилась в потребі до інноваційних підходів та передбачливих рішень протягом стадії дизайну архітектури програми.

Раніше, проектуючи ІС архітектори програмного забезпечення використовували монолітну архітектуру, яка була розділена на 3 великі компоненти (клієнт, сервер, БД). За кожен з цих компонентів відповідала окрема команда розробників. На сьогодні створено багато інструментів та додаткового ПЗ, метою яких є допомогти розробникам, зокрема пришвидшити та зробити процес розробки комфортнішим, а програми – ефективнішими.

Запуск ІС корпоративного рівня (EAS) потребує команди великого розміру, що нерідко складається з десятків, або навіть сотень розробників. Добре поставлений процес взаємодії розробників як в команді, так і між командами є необхідною мірою при розробці EAS. Більше того, умови ведення бізнесу диктують потребу не тільки підтримувати існуючу систему, а й постійно розширювати її, добавляти новий функціонал. Дослідження існуючих технічних рішень дозволяє швидше додавати новий функціонал, чи покращувати існуючий. При такому підході розробникам потрібно зрозуміти існуючі рішення, за необхідності розширити їх та впровадити в свою ІС.

Системою корпоративного рівня можуть користуватися тисячі, а то й мільйони людей, тому вона повинна бути добре масштабованою. Сучасні рішення проблеми масштабування пропонують концепцію горизонтального масштабування, при якій ПЗ працює на декількох фізичних машинах (серверах). Вона надає можливість підвищити ефективність програми додавши додаткові сервери, не змінюючи продуктивність окремої машини [23].

Монолітні EAS також мають недолік розширення. У монолітній архітектурі всі процеси тісно зчеплені (strong coupling) та працюють як один сервіс. Якщо навантаження на одну частину ІС збільшується, то необхідно розширити архітектуру всієї програми. Тісно пов'язані процеси моноліту також означають низький рівень модульності системи, що збільшує ризик помилок окремих процесів та появу проблем з розширюваністю системи через обмежену можливість перевикористання існуючих компонентів. А також це накладає на розробників додаткові обмеження при розробці багатопотокових модулів програми, при яких блокуючий код одного модулю може вповільнити роботу всієї програми [8].

Розширення та вдосконалення монолітної системи ускладнюються разом із зростанням бази коду, що перешкоджає експериментувати та реалізовувати нові ідеї, натомість розробникам доводиться щоразу тестувати та редеплоїти (передислоковувати, розгортати) всю систему вносячи навіть не великі зміни. Велика база коду ускладнює введення нових членів команди до розробки, збільшує потребу в координації команди та зменшує можливість використовувати нові технології (напр. фреймворки), що в цілому негативно впливає на рівень продуктивності.

Поява та популяризація мікросервісної архітектури стала закономірністю при вирішенні проблем монолітних систем. Мікросервіси призначені полегшити масштабування програм та пришвидшити розробку, надавши можливість впровадження інновацій та зменшивши час виходу на ринок нового функціоналу.

Завдяки мікросервісам система будується з незалежних компонентів, які можна запускати як окремий процес (сервіс) програми. Ці сервіси розробляються враховуючи потреби бізнесу, і кожен сервіс має виконувати одну функцію. Оскільки вони працюють незалежно, то кожен сервіс можна незалежно підтримувати, оновляти, розгортати та масштабувати, задовольняючи попит на конкретний функціонал програми необхідний бізнесу [23].

Однак мікросервіси мають свої недоліки. Розподіл великої системи на малі частини може призвести до ризиків безпеки, зокрема тому, що мікросервіси спілкуються між собою за допомогою обміну даними по мережі. Ситуація, коли кожен сервіс має один або декілька серверів, ускладнює обробку помилок (errors, exceptions). Масштабованість мікросервісів викликає потребу ефективного розміщення серверів, а так як система може мати сотні, або й тисячі мікросервісів, то задачу ефективного

розміщення серверів доводиться вирішувати для кожного мікросервіса, що займає велику кількість часу та зусиль.

Мікросервісні фреймворки покликані допомогти розробникам та спростити завдання вирішення проблем мікросервісної архітектури, надаючи можливість впровадження необхідного функціоналу в систему.

В даній роботі буде розглянуто теоретичний аспект мікросервісної архітектури, знайдено та проаналізовано існуючі мікросервісні фреймворки для мови Java, обрано популярний фреймворк та реалізовано за його допомогою програму, проаналізовано процес розробки та результат, зроблено висновки та пропозиції.

Метою роботи є вивчення та порівняння різних мікросервісних фреймворків, для виділення найбільш перспективних та використання їх при розробці програми на Java. Після реалізації програми – аналіз процесу розробки та результату, для з'ясування, наскільки зручно та ефективно використовувати мікросервісні фреймворки при розробці ПЗ.

Розділ 2

Еволюція розробки програмного забезпечення

Архітектура системи це те, що дозволяє їй еволюціонувати, адаптуватись та бути гнучкою до змін з часом для задоволення потреб. Якісно розроблена архітектура надає змогу часто впроваджувати зміни протягом процесів розробки та підтримки системи. Декілька десятиліть інженери програмного забезпечення працювали над розробкою архітектури, яка змогла б надати системам можливість задовольнити функціональні вимоги та відповідати необхідним якісним показникам. З плином багатьох років інженери змогли створити архітектуру, методи та шаблони, які підтримують розвиток гнучких систем.

2.1 Монолітна архітектура

Традиційним підходом до розробки ІС стала монолітна архітектура. При такому підході всі модулі та сервіси системи містяться в одному застосунку, який розгортають (деплойть) єдиним цілим артефактом. На рис. 2.1 наведено приклад монолітного застосунку, що містить декілька сервісів бізнес логіки в одному застосунку, який запаковано та розгорнуто єдиним war файлом.

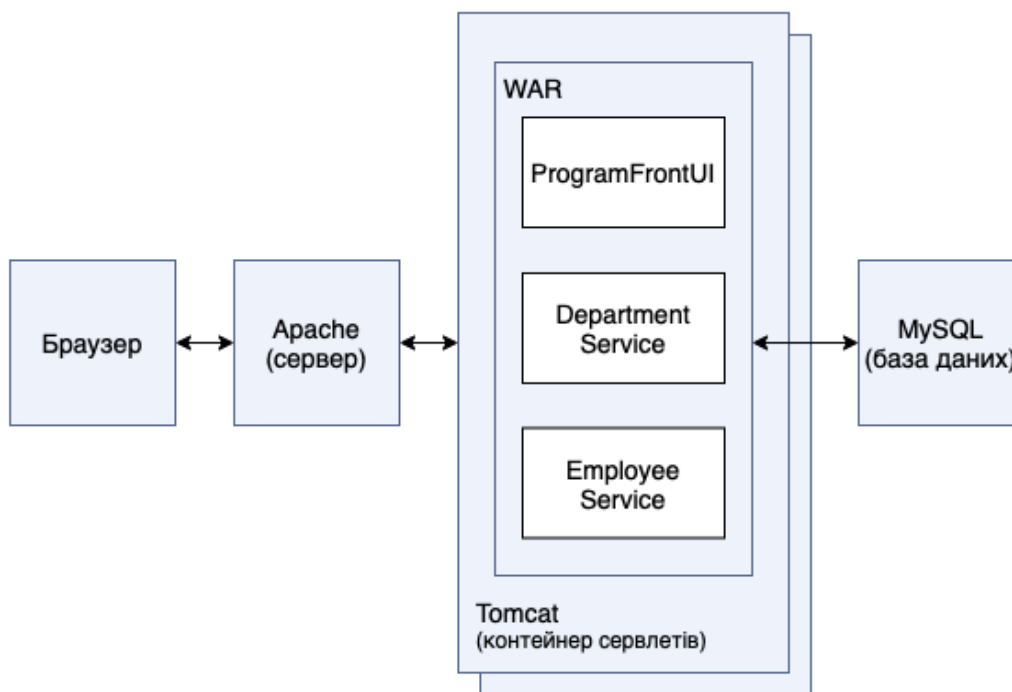


Рис. 2.1 – Схема монолітної архітектури

Підхід монолітної архітектури найкраще підходить для певних потреб і зручний для не великих застосунків [23]. Його відносно легко розробляти та тестувати. Для розгортання моноліту потрібно створити єдиний артефакт та перемістити його на сервер. Розширення досягається горизонтально – за допомогою повторення застосунку на декількох серверах та використання балансувальника навантаження для розподілу трафіку.

З часом застосунків збільшується, база коду стає більшою і процес розробки ускладнюється, зокрема по таким причинам:

- 1) **Складність програми:** через зміни потреб та вимог бізнесу, програмне забезпечення постійно змінюється, а функціональність розширюється. З ростом бази коду моноліту застосунок стає складніше розвивати та підтримувати через його складність.
- 2) **Мінімальна гнучкість:** розробка моноліту передбачає залучення декількох команд розробки для роботи над єдиною базою коду. Впровадження нового функціоналу потребує, щоб завдання було

розподілено між різними командами, а розроблений функціонал – інтегровано в єдину систему. Кожна, навіть невелика зміна, потребує нового розгортання цілого застосунку. Такий процес займає велику кількість часу та робочого часу інженерів. Повільний процес доставки рішень може призвести до ситуації втрачених можливостей компанією.

- 3) **Крихкість:** в монолітному застосунку модулі мають тенденцію до тісної зв'язаності (tight coupling) і з часом стають побудованими між різними залежностями. Внесення простих змін в один модуль програми може визвати помилки в роботі залежних модулів.
- 4) **Обмежена розширюваність:** моноліти зазвичай мають багато сервісів в єдиній програмі. Проблеми виникають коли певні сервіси мають вищий попит ніж інші сервіси, і їх потрібно розширити. Загальноприйнята стратегія менеджменту росту трафіку в монолітах передбачає копіювання програми між різними серверами і використання балансувальника навантаження. Так як всі сервіси містяться в єдиній програмі, то розширювати доводиться цілу програму з усіма її сервісами, а не тільки сервіси попит на яких зріс. Тож необхідним стає велика кількість серверів, ресурси яких не будуть повністю завантаженими.
- 5) **Обмеження технологій:** рішення щодо технологій, які будуть використовуватись при розробці монолітного програмного продукту приймаються протягом початкових стадій процесу аналізу вимог. Команди розробки опиняються в стані технологічного замку, де вони змушені використовувати ті ж технології (напр. мова програмування, фреймворки) протягом всього процесу життя програми. При цьому вимоги до програми

постійно змінюються, а розвиток технологій може запропонувати кращі рішення, ніж ті, що використовує моноліт.

2.2 Об'єктно орієнтоване програмування

Проблеми, що виникають при розробці великих інформаційних систем не є новиною. Сфера проектування програмного забезпечення пройшла інтенсивні дослідження, щоб знайти методи управління масштабною розробкою програмного забезпечення. Велика кількість досліджень щодо практичного впровадження архітектури програмного забезпечення та його концепцій у розробці програмного забезпечення призвела до появи об'єктно-орієнтованого програмування (ООП) у 1980-х роках. Парадигма проектування ООП передбачає кілька підходів до перетворення дизайну програмного забезпечення в код за допомогою групи стійких рішень, відомих як шаблони. Популярний прикладом шаблону архітектурного проектування ООП, який часто використовується розробниками для управління великомасштабним програмним забезпеченням, є шаблон Model-View-Controller (MVC) [23].

2.3 Сервіс-орієнтована архітектура

Концепція відокремлення проблем – це принцип проектування, який наголошується в ООП і який набув популярності як широко рекомендована концепція проектування великомасштабних інформаційних систем. Розробка на основі компонентів забезпечує більші можливості управління

складною системою та її просте обслуговування. Сервіс-орієнтована архітектура (SOA), виникла завдяки поєднанню концепцій ООП та розподілу програми на компоненти. При такій архітектурі програма розділена на кілька частин – сервісів. Сервіс надає функціональні можливості, доступні іншим службам за допомогою різних протоколів, таких як простий протокол доступу до об'єктів (SOAP, Simple Object Access Protocol).

2.4 Сучасні потреби ринку до розробки

Сучасні потреби та очікування споживачів змусили компанії переорієнтуватися, щоб розвиватись та процвітати на сучасному споживчому ринку.

Хмарні технології широко використовуються у всіх галузях промисловості у всьому світі. В індустрії інформаційних технологій хмарні технології за визначенням означають величезну мережу серверів, розповсюджених по всьому світу, але з'єднаних між собою через Інтернет. В результаті формується одна єдина гармонійна екосистема. Вся мережа може бути використана для зберігання, управління та доставки контенту до своїх користувачів. За допомогою хмарних технологій дані є високодоступними і до них можна легко отримати доступ у будь-якому бажаному місці та в будь-який час за допомогою будь-якого пристрою, підключеного до Інтернету, усуваючи обмеження для локального зберігання та вилучення з фізичних систем зберігання даних [9].

Впровадження хмарних технологій компаніями стало неминучим процесом. Хмарні технології полегшують розгортання програм таким

чином, щоб масштабування обчислювальних ресурсів відбувалось в швидкому режимі та забезпечувало безперервну доставку (continuous delivery). Потреба компаній у запуску своїх додатків у хмарах була прискорена зростанням попиту на ефективність операцій та високу доступність. З тієї ж причини компаніям необхідно впроваджувати інновації якомога швидше, а отже, зростає потреба у безперервному впровадженні (continuous deployment).

Сьогодні, в час хмарних технологій, спроби бізнесу розгортати монолітні додатки в хмарах призводить до серйозних проблем, неможливості в повній мірі використати переваги хмарних технологій та, як наслідок, втрата конкурентних переваг.

Монолітний підхід до розробки програмного забезпечення був сприятливим в епоху фізичних комп'ютерів та веб-сайтів. У наш час системи стали надзвичайно великими, тому розгортати їх єдиним артефактом стало недоцільно. Для того, щоб подолати труднощі, пов'язані з розгортанням монолітів у хмарі, і в повній мірі скористатися можливостями хмарних технологій – з'явилась мікросервісна архітектура, яка є наступником SOA зі зменшеним розміром бази коду програми та її складності.

Перша ітерація мікросервісної архітектури була зосереджена на ідеї інкапсуляції та компонування, тоді як остання ітерація зосереджена на незалежній розробці та розгортанні. Архітектура мікросервісів дає орієнтир для проектування та реалізації програми як набору невеликих розподілених сервісів.

Розділ 3

Мікросервіси

Сучасний мікросервісний підхід походить від добре відомої сервіс-орієнтованої архітектури (SOA). Ідея SOA полягає в поділі різних систем на окремі, менші сервіси. Однак існувала проблема відсутності стандартів щодо створення сервіс-орієнтованої системи. Кожен розробник сам інтерпретував архітектуру, наприклад, з чого складатиметься певний сервіс, та визначав наскільки великим він буде. Мікросервісна архітектура з'явилася для вирішення проблем, пов'язаних із SOA надавши розробникам більше стандартів щодо створення мікросервісів. Можна стверджувати, що мікросервісна архітектура є спеціалізацією сервіс-орієнтованої архітектури з чітко визначеними стандартами.

Існують наступні визначення мікросервісної архітектури:

- 1) Архітектурний стиль мікросервісів - це підхід до розробки єдиної програми як набору невеликих служб, кожна з яких працює у своєму власному процесі та спілкується з легкими механізмами, часто через інтерфейс HTTP ресурсів. Ці сервіси побудовані на основі потреб бізнесу та незалежно розгортаються за допомогою повністю автоматизованої техніки розгортання. Існує мінімум централізованого управління цими службами, яке може бути

написане різними мовами програмування та використовувати різні технології зберігання даних [21].

- 2) Мікросервіси це малі, автономні сервіси, що працюють разом [2].
- 3) Мікросервіси (архітектура мікросервісів) – це хмарний архітектурний підхід, в якому одна програма складається з багатьох вільно пов'язаних і незалежно розгортуваних менших компонентів або служб [32].

Тож, можна сказати, що мікросервіси – це архітектурний та організаційний підхід до розробки ПЗ, яке складається з невеликих незалежних сервісів, що взаємодіють за допомогою чітко визначеного API.

3.1 Характеристика мікросервісної архітектури

Мікросервіси виступають архітектурним стилем, який структурує ПЗ як сукупність сервісів, яким характерні:

- 1) зручність підтримання та тестування;
- 2) слабка зчепленість (low coupling);
- 3) незалежне розгортання;
- 4) відповідність потребам бізнесу;
- 5) розробка малими командами.

Мікросервісна архітектура дозволяє бізнесу швидко, часто і надійно доставляти споживачам великі, складні програми з необхідним функціоналом. А також мікросервіси роблять можливим впровадження нових технологій в технічний стек команд розробки.

Додатковими особливостями, що характеризують мікросервіси є:

- 1) **Єдина мета:** базується на принципі, згідно з яким програма повинна робити одну річ, і вона повинна робити це добре. Монолітні програми надають кілька сервісів у межах однієї кодової бази, яка може містити мільйони рядків коду. Наприклад, комерційна програма оброблятиме департаменти, працівників, замовлення, інвентар, продукти та ціни в одній програмі. Завдяки архітектурі мікросервісів кожна служба несе одну відповідальність. Зазвичай очікується, що кодова база сервісу з часом збільшуватиметься, щоб забезпечити більше функціональних можливостей у міру розвитку бізнесу. Однак роздуття та підвищену складність можна уникнути, делегуючи відповідальність за розробку кожної послуги невеликим командам із 2-15 розробників. Якщо це число потрібно перевищити, це може бути ознакою того, що мікросервіс робить занадто багато справ і, ймовірно, його слід розділити на дві частини. Менша команда розробників також, швидше за все, збереже фокус та гарантує, що мікросервіс не виходить за межі вирішення однієї відповідальності.
- 2) **Інкапсуляція:** мікросервіси повинні володіти своїми даними та приховувати їх реалізацію. Кожен мікросервіс повинен мати власний обсяг зберігання даних і зберігати його приватним. До постійних даних мікросервісу можна отримати доступ лише через чітко визначений API. Недоліки спільного використання даних та доступу до них – це тісне зчеплення (high coupling), оскільки доступність мікросервісу стає безпосередньо залежною від бази даних, якою керує інший мікросервіс. Інкапсуляція забезпечує вільне зв'язування (low coupling) між службами та зберігає явні

залежності, тому уникає необґрунтованих складнощів. Ще однією важливою перевагою інкапсуляції є той факт, що будь-які зміни, внесені до бази даних певної служби, не заважають будь-якій іншій службі.

- 3) **Власність:** у великому монолітному додатку часто працює велика команда розробників. Наприклад, додаток, що складається з десятків мільйонів рядків коду, може мати команду з близько 100 розробників. Індивідуальний розробник може привнести лише незначний відсоток користі. Таким чином, програма, як правило, розглядається як чорна скринька, яку ніхто повністю не розуміє, і в розробників рідко існує стимул взяти на себе відповідальність за неї. Іншими словами, серед членів команди в основному немає почуття власності. Додаток ризикує ускладнитися через те, що розробники діють у своїх інтересах, а не для покращення програми. Мікросервісна архітектура навпаки розвивається в основному завдяки власності. Організація розробників навколо мікросервісу призводить до створення невеликих автономних команд, які відповідають за мікросервіс від його етапу розробки до розгортання. Команда є незалежною та приймає власні рішення, тому підвищується мотивація та ефективність.
- 4) **Автономність:** команда, що відповідає за мікросервіс, є незалежною і повинна мати можливість проектувати, розробляти, розгортати та підтримувати мікросервіс у повній ізоляції без необхідності координації з іншою командою. Команда несе повну відповідальність за результати роботи мікросервісу і тому має право приймати рішення з усіх питань, інструментів, технологій та методологій. Різні мікросервіси вирішують різні бізнес-проблеми. Кожна команда повинна мати можливість вибрати стек технологій,

який найкраще відповідає їхнім потребам. Зрозуміло, що організаціям потрібно стандартизувати деякі зовнішні деталі, такі як протоколи API, оповіщення, надсилання повідомлень та реєстрація. Кожен мікросервіс повинен відкривати лише свій API, отже, технічні особливості внутрішнього впровадження не мають істотних наслідків.

- 5) **Багатоверсійність:** можливість одночасного розгортання декількох версій мікросервісу в одному середовищі. Наприклад, можна мати різні версії мікросервісу, які працюють у виробничому середовищі одночасно. Це стало можливим завдяки використанню URL адреси, наприклад (*/products/v1.1*). Клієнти можуть надсилати http запити до певної версії мікросервісу. Це особливо вигідно для компаній, які прагнуть дуже швидко випускати своїм клієнтам мінімально життєздатні версії свого продукту (MVP). Можливість архітектури мікросервісів підтримувати декілька версій є винятковою характеристикою, не притаманною, зокрема монолітній архітектурі.

Масштабованість мікросервісної архітектури наглядно зображено на кубі масштабованості (scalability) ПЗ, рис. 3.1.

Куб масштабованості зображує три способи масштабування програми [26]:

- 1) Х-масштабування (клонування), яке розподіляє запити між декількома ідентичними інстансами (серверами);
- 2) Z-масштабування (поділ подібних даних), яке спрямовує запити в залежності від атрибуту запиту;
- 3) Y-масштабування (поділ функціоналу, що відрізняється), яке розподіє систему на окремі сервіси.

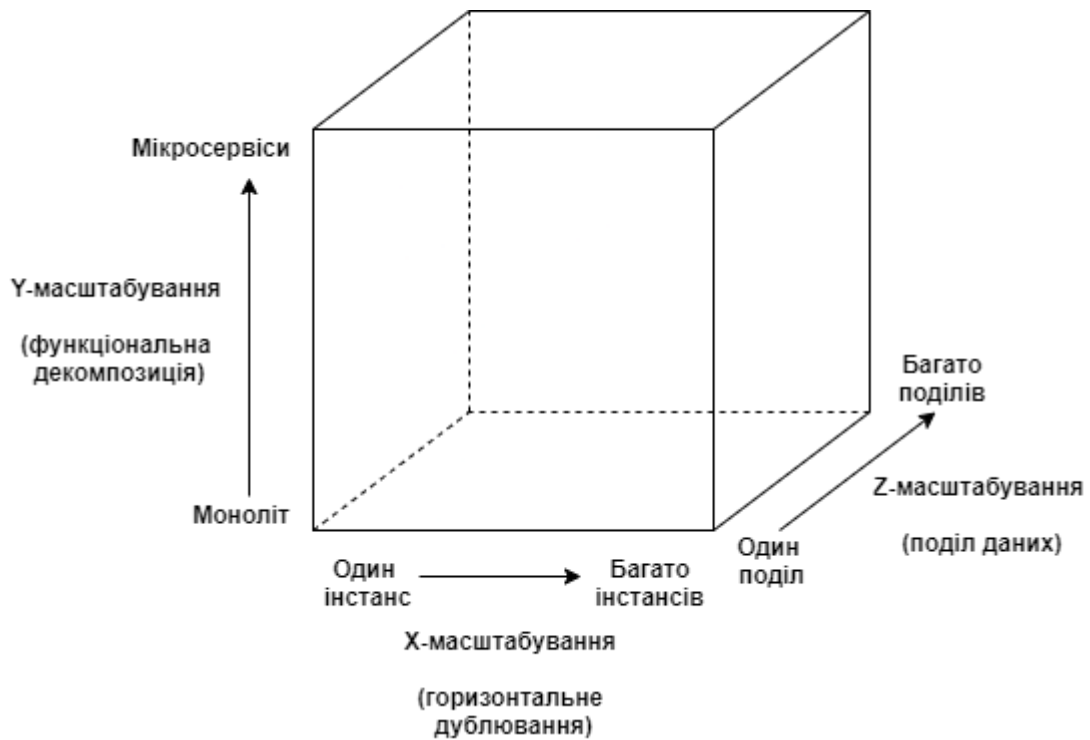


Рис. 3.1 – Куб масштабованості ПЗ

X-масштабування є загальноприйнятим методом масштабування монолітної системи.

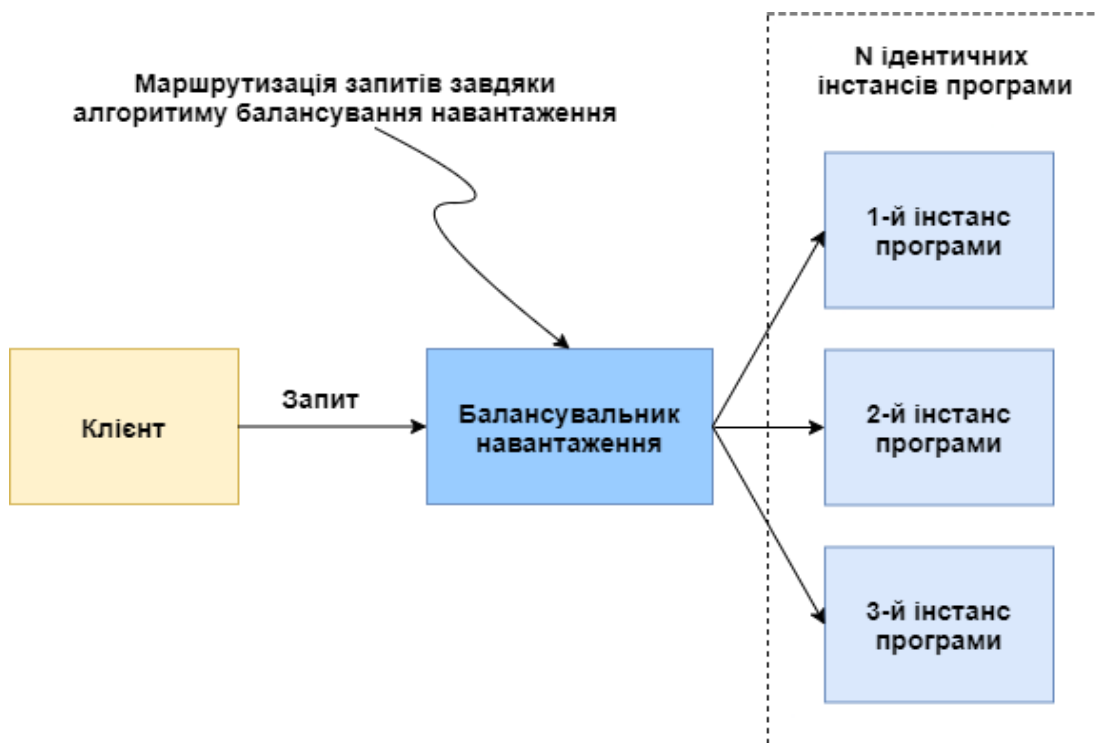


Рис. 3.2 – X-масштабування

На рисунку 3.2 показано, як працює X-масштабування. Такий спосіб масштабування досягається через запуск ідентичних екземплярів (інстансів) програми за допомогою балансувальника навантаження (load balancer). Балансувальник розподіляє запити між N ідентичними інстансами програми. Таким чином покращується потужність та доступність програми.

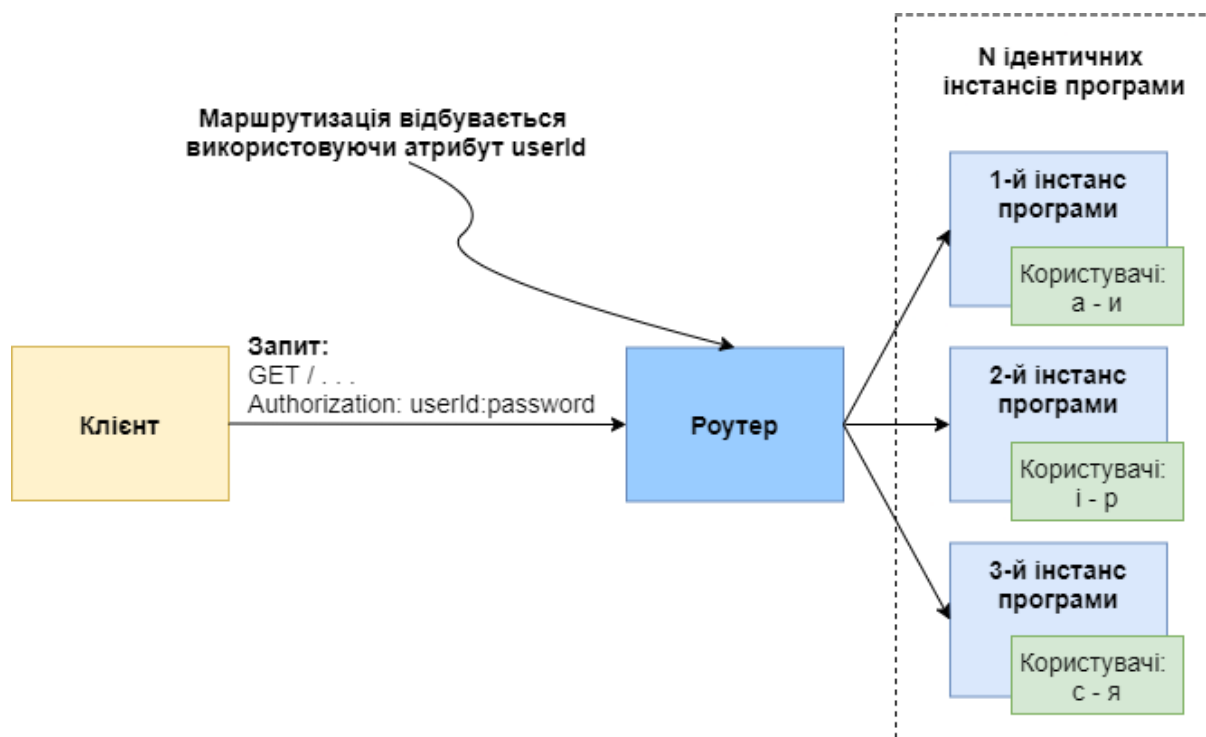


Рис. 3.3 – Z-масштабування

Z-масштабування також полягає в запуску кількох інстансів монолітної програми, але на відміну від X-масштабування, кожен інстанс відповідає лише за підмножину даних. На рисунку 3.3 показано, як це працює. Маршрутизатор перед інстансами використовує атрибут запиту для направлення цього запиту до відповідного інстансу. Наприклад по атрибуту `userId` роутер буде направляти до відповідних інстансів розподілених між користувачами.

На прикладі Z-масштабування кожен інстанс програми відповідає за підмножину користувачів. Маршрутизатор використовує `userId`, зазначений

у заголовку запиту, для вибору одного з N ідентичних інстансів програми. Таке масштабування доцільно використовувати при обробці зростаючих обсягів транзакцій та даних.

X- та Z-масштабування покращують потужність та доступність системи. Але вони не вирішують проблему збільшення складності розробки та накопиченої бази коду. Для вирішення цих проблем потрібно застосувати Y-масштабування (функціональну декомпозицію). На рис. 3.4 показано, як завдяки Y-масштабуванню відбувається розподіл монолітної системи на мікросервіси.

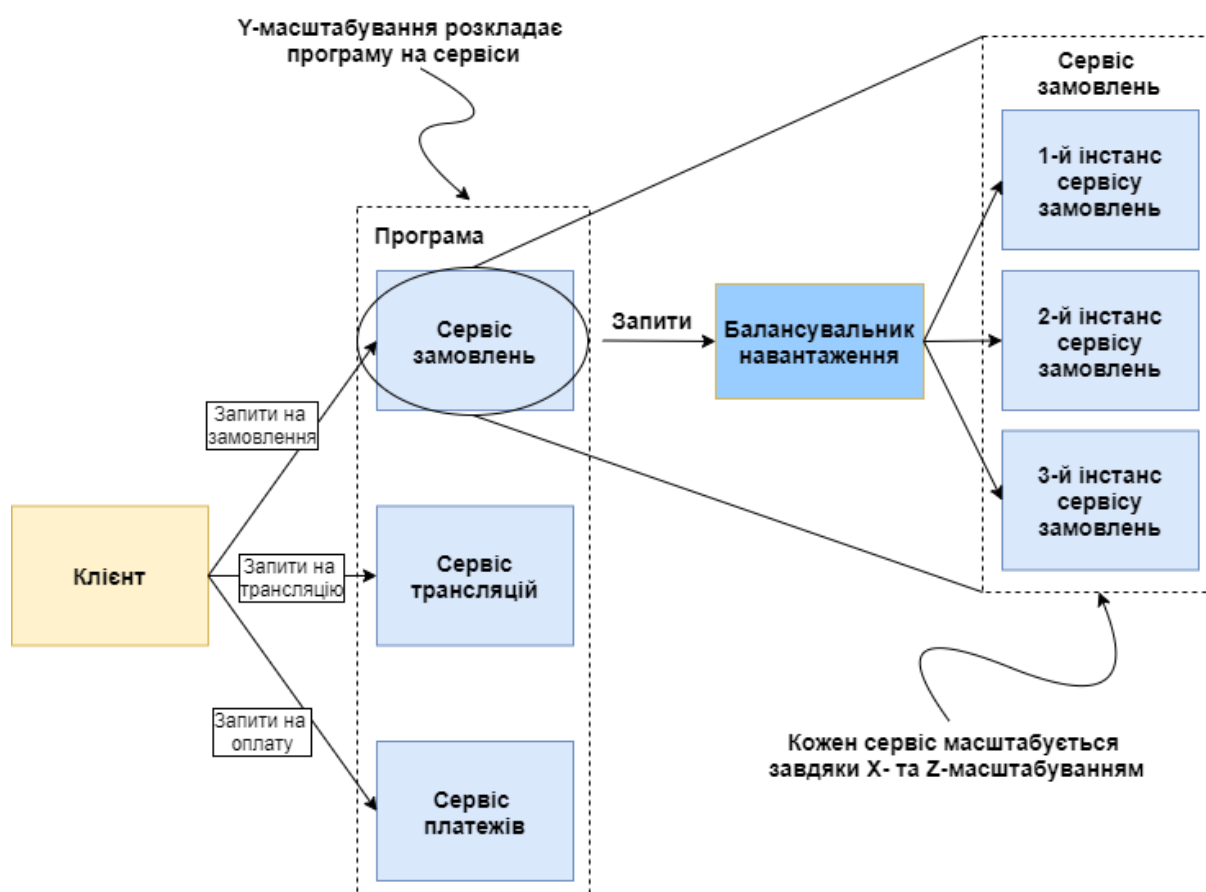


Рис. 3.4 – Y-масштабування

Як бачимо з рис. 3.4 сервіс виглядає як міні-програма, що реалізує певний функціонал, наприклад замовлення, трансляції, платежі. Самі сервіси можна масштабувати X- та Z-масштабуванням. Кожен мікросервіс

відповідає лише за один, власний функціонал. У великій системі багато таких сервісів можуть також взаємодіяти між собою. Такий підхід надає сильну згуртованість (high cohesion) та слабку зчепленість (low coupling).

Модульність має важливе значення при розробці великих, складних систем, зокрема корпоративного рівня. Зазвичай сучасні програми занадто великі, щоб їх було можливо розробляти одному розробнику. Велику базу коду може бути дуже складно зрозуміти. Велику програму потрібно розкласти на менші модулі, які буде можливо розробляти та розуміти розробникам. У монолітній програмі модульність досягається за допомогою комбінації конструкцій мови програмування (напр. пакети (packages) Java) та артефактів (напр. JAR, WAR). Однак, на практиці такий підхід не завжди працює належним чином. Довговічні монолітні програми зазвичай накопичують базу коду до міри, яка не зацікавить жодного розробника.

Натомість в мікросервісній архітектурі кожен сервіс є окремим модулем. Сервіси обмінюються даними за допомогою конкретного API, який неможливо обійти. Не використавши API розробники не зможуть отримати доступ до внутрішнього класу, як це можливо за допомогою конструкції пакетів Java. Результатом чого є краще збереження модульності програми з часом. Можливість незалежного розгортання та розширення також є незаперечними перевагами модульності.

3.2 Порівняння мікросервісів та монолітів

Монолітна архітектура структурована абсолютно не так, як архітектура мікросервісів, як з технологічних, так і з організаційних

аспектів. У наведеній нижче таблиці узагальнено найбільш суттєві відмінності між двома архітектурами.

Таблиця 3.1 – Ключові відмінності між мікросервісною та монолітною архітектурою

Мікросервісна архітектура	Монолітна архітектура
Множина багатьох мікросервісів з обмеженим функціоналом	Єдина програма
Мікросервіси можуть бути розгорнутими незалежно	Вся програма повинна бути розгорнута
Кожен мікросервіс має власну базу даних	Єдина база даних для всієї програми
Обмін повідомленнями відбувається віддалено, зокрема використовуючи REST запити через HTTP протокол	Обмін повідомленнями всередині програми
Співробітництво між розробниками та спеціалістами з операцій для забезпечення стабільності операцій	Відокремлення команди розробників від спеціалістів із операцій
Стан зберігається централізовано, окремі інстанси є без стану (stateless)	Стан залежить від зовнішнього застосування в режимі роботи

3.3 Порівняння мікросервісів та SOA

З точки зору загального визначення, мікросервіси та SOA схожі. Обидва архітектурні підходи роблять акцент на поділі великих систем на розподілені сервіси. Сервіси стають основними компонентами архітектури та для реалізації системних функцій та бізнес функцій.

Відмінності між мікросервісами та SOA можна побачити порівнявши характеристики їх сервісів та їх можливостей.

3.3.1 Систематика сервісів

Під систематикою сервісів мається на увазі класифікація сервісів за архітектурним шаблоном. Існує кілька способів класифікації сервісів. Сервіси можна класифікувати відповідно до завдань, які вони виконують в архітектурі, наприклад, деякі сервіси реалізують функціональні можливості бізнесу, а інші реалізують небізнесові функції, такі як безпека або реєстрація.

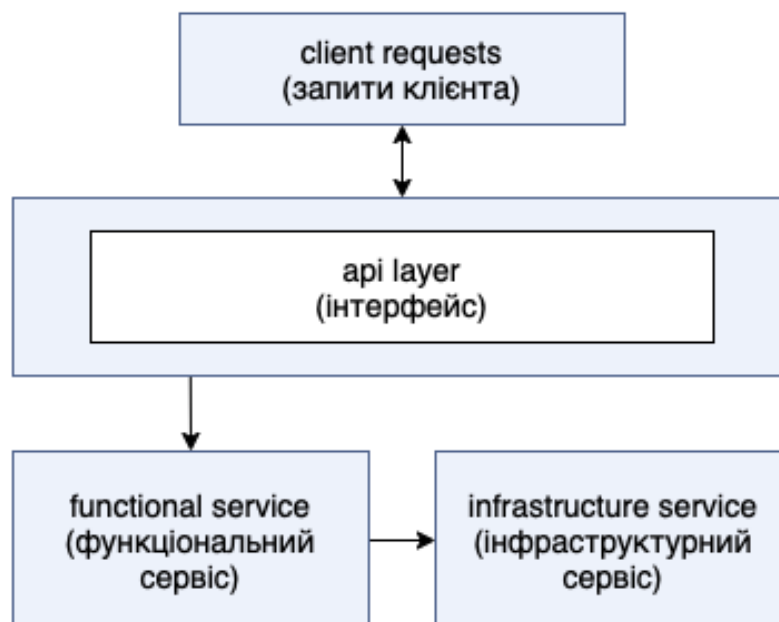


Рис. 3.5 – Таксономія сервісу в мікросервісній архітектурі

Таксономія сервісів мікросервісної архітектури обмежена лише двома класифікаціями сервісів, а саме функціональними сервісами та інфраструктурними сервісами.

Функціональні сервіси складаються з сервісів, які реалізують та підтримують певні бізнес-функції чи операції. З іншого боку, сервіси інфраструктури виконують такі допоміжні завдання, як авторизація, автентифікація, моніторинг та реєстрація. Важливим для цього порівняння є той факт, що інфраструктурні сервіси в мікросервісах недоступні поза архітектурою, і вони розподіляються лише внутрішньо між сервісами. Функціональні сервіси доступні зовнішньому світу за допомогою реалізації API.

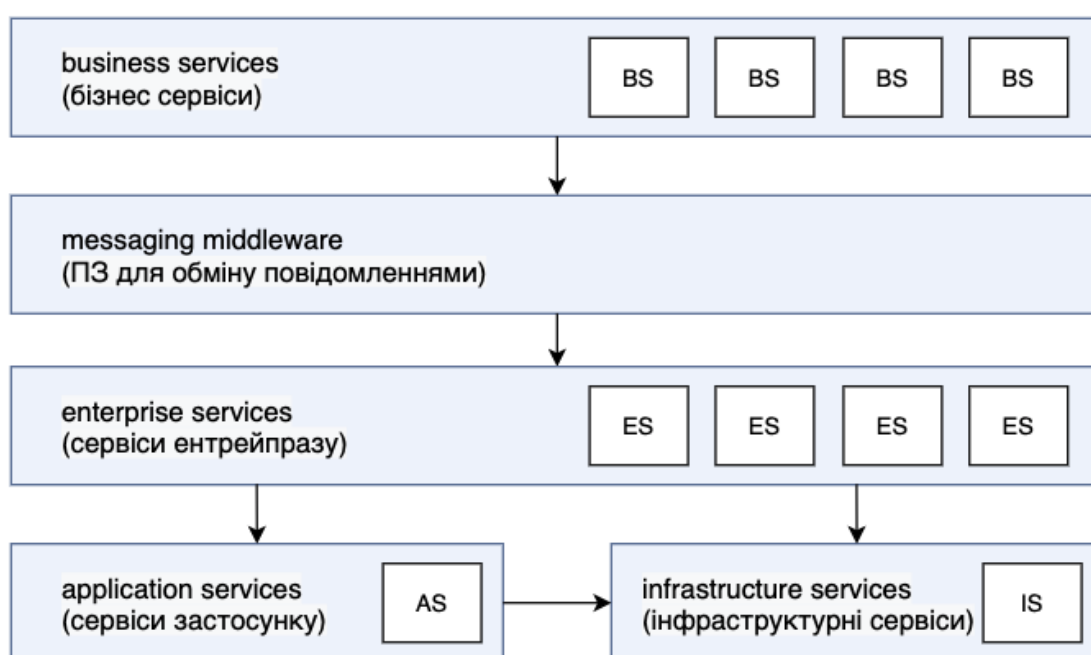


Рис. 3.6 – Таксономія сервісу в сервіс-орієнтованій архітектурі

Таксономія сервісу в архітектурі SOA формалізована та чітко класифікує сервіси за чотирма типами, як показано на рис. 3.6. Бізнес-сервіси це високорівневі сервіси, які визначаються основними бізнес операціями. Їх представлення зазвичай здійснюється за допомогою XML, мови виконання бізнес-процесів (BPEL) або мови визначення веб-служб (WSDL).

Ентерпрайз сервіси – це конкретні сервіси, які реалізують бізнес функціональність, визначену бізнес сервісами, і зазвичай є спільними між

іншими сервісами в архітектурі. Ентерпрайз сервіси є залежними від інфраструктурних сервісів та сервісів застосунку в ході виконання запитів. Сервіси застосунку це сервіси, які реалізують бізнес функції, що є дуже специфічними в контексті програми. Інфраструктурні сервіси, як і в мікросервісній архітектурі, використовуються для реалізації нефункціональних допоміжних завдань, напр. аудит та моніторинг [15].

3.3.2 Розмір сервісів

Розмір сервісів між двома архітектурами істотно відрізняється. Як передбачає префікс «мікро» в мікросервісах, сервіси мають бути добре деталізованими і невеликими. Вони мають мати єдину відповідальність, робити лише одне і робити це якісно. Їх сфера застосування обмежена реалізацією єдиної бізнес-функції.

З іншого боку, розмір сервісів SOA варіюється від добре деталізованих прикладних сервісів до великих корпоративних сервісів з чітким напрямом. Не рідкість зустріти SOA сервіс, яка представляє великий бізнес-продукт або навіть цілу підсистему. SOA залежить від кількох сервісів для виконання окремого бізнес-запиту.

3.3.3 Взаємодія команд розробки сервісів

Право власності на сервіс відноситься до команди в організації, яка відповідає за розробку, розвиток та підтримку сервісу. Через те, що мікросервіси мають меншу таксономію (інфраструктурні послуги та функціональні послуги), команди розробників цієї архітектури можуть мати право власності як на інфраструктуру, так і на функціональні сервіси.

Команди не спеціалізуються на розробці одного виду сервісів, а складаються з експертів з різних галузей і тому є самодостатніми [14].

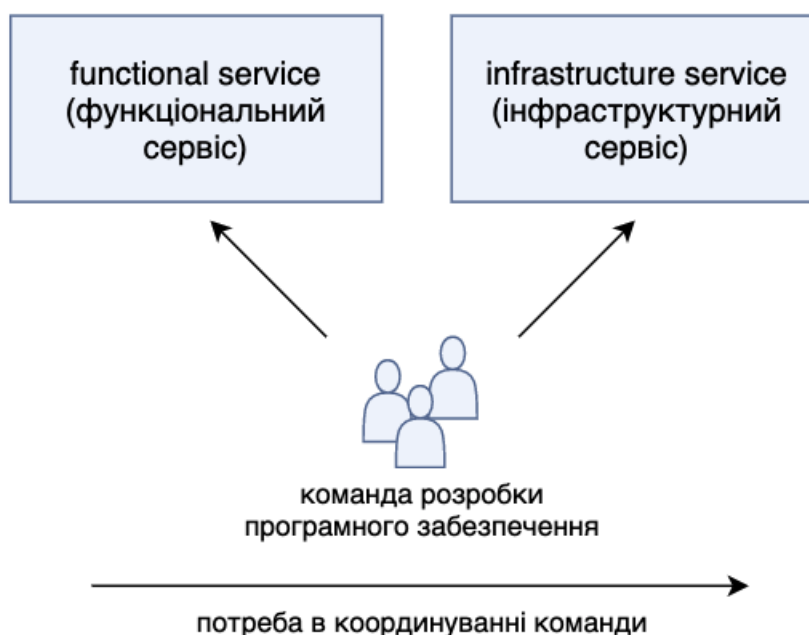


Рис. 3.7 – Взаємодія команд розробки мікросервісів

З іншого боку, сервіси в архітектурі SOA мають різних власників. Бізнес-послуги, як правило, належать бізнес-користувачам в організації. Архітектори та команди спільних сервісів володіють корпоративними сервісами. За сервіси прикладних програм несуть відповідальність команди розробників додатків, тоді як інфраструктурні сервіси можуть належати або команді з розробки додатків, або команді з розробки інфраструктури. На рис. 3.8 показано взаємодію команд розробки сервісів в архітектурі SOA.

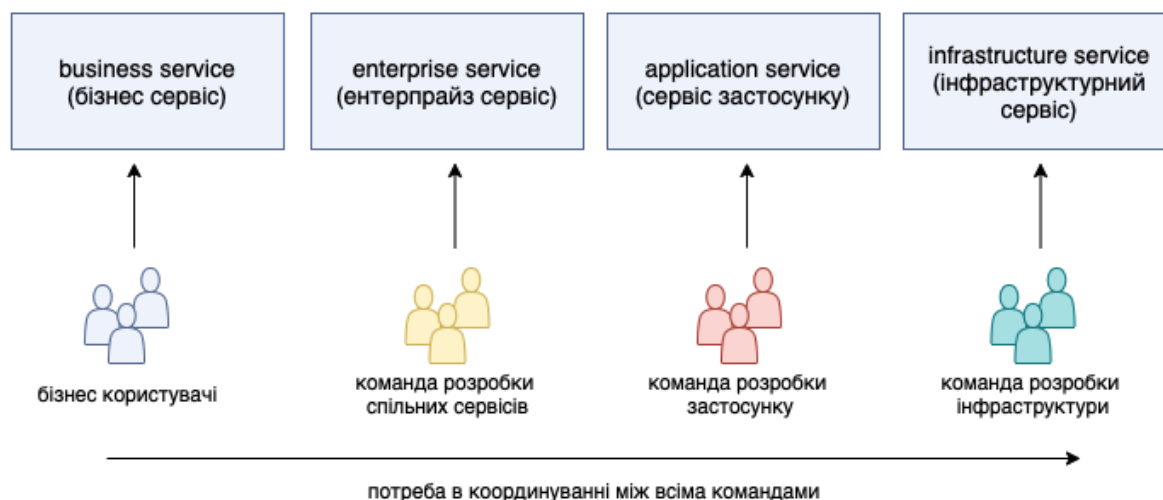


Рис. 3.8 – Взаємодія команд розробки сервіс-орієнтованої архітектури

Актуальність приналежності сервісів командам стає очевидною в загальній координації сервісів. При SOA розробка повного бізнес-запиту вимагає координації між кількома командами. Постійна потреба в консультаціях між власниками рівнів сервісів перешкоджає швидкому розвитку. При мікросервісному підході потреба в координації команд для виконання єдиного бізнес-запиту є мінімальна. Координація між командами, якщо це необхідно, досягається створенням невеликих команд, які можуть швидко розробляти, тестувати та розгортати сервіси.

3.4 Переваги мікросервісної архітектури

Перевагами мікросервісної архітектури є:

- 1) постійна доставка (continuous delivery) та розгортання великих, складних програм;
- 2) сервіси невеликі та легко обслуговуються;

- 3) незалежне розгортання;
- 4) незалежне масштабування;
- 5) дозволяє командам бути автономними;
- 6) можливість легко експериментувати та впроваджувати нові технології;
- 7) краща ізоляція помилок.

Найважливішою перевагою MSA є те, що вона забезпечує постійну доставку та розгортання великих, складних програм. Безперервна доставка та розгортання є частиною DevOps, набору практик для швидкої, частої та надійної доставки ПЗ.

Мікросервісна архітектура має такі переваги для надання безперервної доставки та розгортання ПЗ:

- 1) **наявність необхідних умов для тестування:** автоматизоване тестування є ключовою практикою безперервної доставки та розгортання. Кожен сервіс порівняно невеликий, а автоматизовані тести набагато простіше писати та швидше виконувати. Як результат, програма має менше помилок;
- 2) **наявність необхідних умов для розгортання:** кожен сервіс можна розгорнути незалежно від інших сервісів. Якщо розробникам, відповідальним за певний сервіс, потрібно щось локально змінити, то вони зможуть зробити це без узгодження з іншими розробниками. Як результат, можна легко та часто впроваджувати необхідні зміни;
- 3) **мікросервіси надають можливість командам розробників бути автономними та незалежними:** стає можливим структурування

організації на маленькі команди розробки. Кожна команда несе відповідальність за розробку та впровадження одного або декількох мікросервісів. Як показано на рис. 3.9, кожна команда може розробляти, розгортати та масштабувати свій сервіс незалежно від усіх інших команд. Як результат, швидкість розробки значно вища.

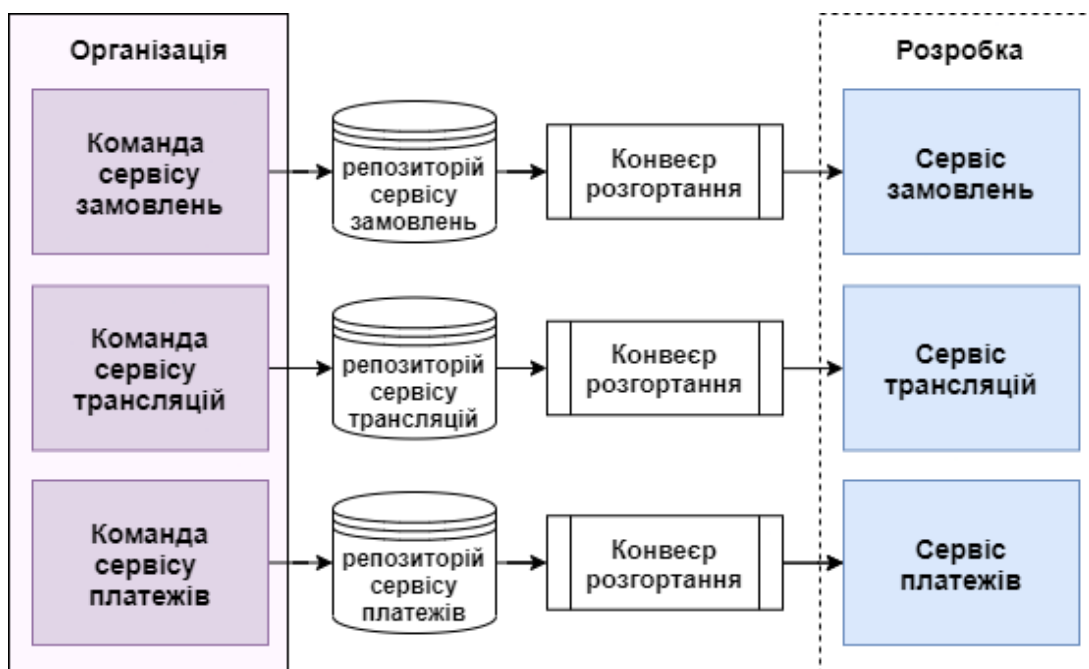


Рис. 3.9 – Організація мікросервісної архітектури

Технічні переваги DevOps мають наслідком такі бізнесові переваги:

- 1) низький час виходу оновлень на ринок, що надає можливість швидко реагувати на відгуки споживачів;
- 2) з'являється можливість надавати необхідний сервіс, який очікують споживачі;
- 3) підвищується задоволеність працівників, адже вони можуть витратити більше часу на корисні розробки, ніж на виправлення помилок.

Друга перевага MSA в тому, що сервіси не великі та легко обслуговуються. База коду таких сервісів відносно мала, тому її легше зрозуміти. DE не перегружені таким розміром коду, що дозволяє розробникам бути більш ефективними. А також кожен сервіс зазвичай можна запустити значно швидше ніж монолітну програму, що пришвидшує розгортання.

Наступна перевага незалежного розширення призводить до того, що кожен сервіс можна розширювати незалежно від інших сервісів використовуючи X- та Z- масштабування. Більш того, кожен сервіс можна розгорнути на апаратному забезпечення, що найкращим чином підходить під вимоги цього сервісу (напр. CPU-залежні, залежні по пам'яті).

Краща ізоляція помилок означає, що за умови, якщо з одним сервісом відбувається помилка, то це не вплине на інші сервіси. Інші сервіси продовжать свою роботу в звичайному режимі.

Перевага мікросервісів у легкій зміні технологічного стеку надає розробникам можливість легко експериментувати та впроваджувати нові технології. Розпочинаючи розробку нового сервісу розробники можуть обрати будь яку мову програмування та фреймворк, які найкращим чином підходять для цього сервісу. А також команди розробки не обмежені минулими технологічними рішеннями.

Більше того, малий розмір сервісів надає можливість переробити існуючий сервіс з мінімальними витратами. Якщо спроба з певними технологіями не вдалась, то команда може змінити технологію та швидко переписати мікросервіс не впливаючи на всю програму.

Розділ 4

Мікросервісні Java фреймворки

В даний час на ринку існує безліч фреймворків на різних етапах розробки, кожен фреймворк задовольняє певні потреби на ринку. Залежно від того, який тип програми розробляється, можуть існувати різні типи функціональних можливостей, які можна класифікувати як корисні або, навіть, необхідні.

Також, перш ніж приступити до розробки системи потрібно знати вимоги цієї системи, а потім розподіляти її по мікросервісах. Процес впровадження MSA можна розділити на два етапи:

- 1) першим етапом потрібно забезпечити, щоб система була сумісна з мікросервісним підходом;
- 2) другим етапом потрібно впровадити різні частини мікросервісів разом з необхідним функціоналом.

Провівши пошук популярних мікросервісних фреймворків для Java можна виділити наступні:

- 1) Spring Boot;
- 2) Dropwizard;
- 3) WildFly Swarm;

- 4) Eclipse Vert.X;
- 5) Quarkus;
- 6) Micronaut.

4.1 Spring Boot

Spring Boot – це популярна платформа Java для написання мікросервісів. Вона надає велику кількість додаткових розширень під Spring Cloud для створення повноцінних мікросервісів. Spring Boot дозволяє створювати масштабні системи, запускаючи просту архітектуру з набору пов'язаних компонентів, які можуть надати розробникам можливість створювати мікросервіси із готовими шаблонами та конфігураціями. Його можна використовувати для побудови невеликих, а також масштабних систем. Spring Boot дуже легко інтегрувати з іншими популярними фреймворками завдяки інверсії управління (IoC) [27].

Код Spring Boot є відкритим, а функціонал широким, фреймворк має велику спільноту, до якої можна звернутися за підтримкою. Мікросервіси Spring Boot можна легко розгорнути на різних платформах, будь то Docker або фізичні сервери. Він також має вбудовані функціональні можливості, такі як автоконфігурація безпеки, яка разом із стартовими залежностями дозволяють швидко розробляти програми на Java.

Успіх Spring Boot досягає цього за допомогою:

1) автоматичної конфігурації:

Spring історично був важким для налаштування. Однак, фреймворк вдосконалювався на основі інших моделей компонентів (EJB 1.x, 2.x тощо),

він продовжував йти разом з власним набором надважких патернів використання. Spring потребував багато XML конфігурації та глибокого розуміння окремих бінів (bean) необхідних для конструювання JdbcTemplates, JmsTemplates, BeanFactory, хуків життєвого циклу, слухачів сервлетів та багатьох інших компонентів. Написання простого «Hello, World» з Spring MVC вимагало розуміння DispatcherServlet та цілого ряду класів Model-View-Controller.

Spring Boot має на меті усунути складну та типову конфігурацію шаблону використовуючи певні непрямі умови та спрощені примітки. Хоча можливість тонко налаштувати базові компоненти залишається присутня.

2) набір популярних стартових залежностей (dependencies):

Spring застосовують у великих корпоративних програмах, які зазвичай використовували багато різних технологій для виконання складних завдань: бази даних JDBC, черги повідомлень, файлові системи, кешування на рівні програм тощо. Розробнику доводилось переключати контекст над тим, що потрібно програмі втрачаючи багато часу на вирішення невідповідностей версій чи інших проблем. Spring Boot надає велику колекцію стартових залежностей для додавання необхідного функціоналу, зокрема:

- a) JPA persistence;
- b) бази даних NoSQL, такі як MongoDB, Cassandra та Couchbase;
- c) кешування за допомогою Redis;
- d) контейнери сервлетів Tomcat, Jetty;
- e) JTA транзакції.

Додавання підмодулю до програми приносить контрольований набір перехідних залежностей та версій, які можуть працювати разом. Це спрощує

роботу розробникам забираючи необхідність самостійно вирішувати проблеми залежностей.

3) спрощене упакування (packaging) програм:

Spring Boot являє собою набір завантажувальних бібліотек з певними умовами для конфігурацій, але можливість запустити додаток Spring Boot на наявних серверах додатків (як WAR) збережена. Spring Boot – це автономна упаковка JAR для їх застосування. Це означає, що Spring Boot пакує всі залежності та код програми в автономний JAR із завантажувачем класів. Це полегшує розуміння запуску програми, впорядкування залежностей та операторів логуювання, але що ще важливіше, це допомагає зменшити кількість рухомих частин, необхідних для безпечного запуску програми у виробничий стан. Це означає, що не потрібно брати додаток і завантажувати його на сервер додатків; після створення, програма вже готова до роботи, так як є автономною, тобто має в собі вбудованого контейнера сервлетів, при необхідності роботи з сервлетами.

4) середовище розробки з програмними метриками:

Spring Boot поставляється з модулем під назвою actuator, який включає такі параметри, як показники та статистичні дані про додаток. Наприклад, за його допомогою можна збирати логи, переглядати показники, виконувати дампи потоків, показувати змінні середовища, розуміти процес збирання сміття та показувати, які компоненти налаштовані у BeanFactory. Ця інформація може бути відкрита через HTTP, JMX запити або навіть увійшовши безпосередньо до процесу через SSH.

За допомогою Spring Boot стає можливим використання потужності роботи Spring Framework, при цьому зі зменшеною конфігурацією та шаблонним кодом для більш швидкого створення потужних, готових до застосування мікросервісів [28].

Почати роботу з Spring Boot можна відвідавши сторінку start.spring.io, яка допоможе автоматично запустити шаблон програми. Також доступні різні тренінги та сертифікації по Spring Boot. Фреймворк можуть використовувати навіть менш досвідчені команди для вирішення складних бізнес-проблем. На практиці Spring Spring зайняв місце найефективнішого стандарту для розвитку мікросервісів.

Порівняння популярності пошукових запитів Google та зацікавленості розробників на тему Spring та Spring Boot за останні 10 років відображено на рис. 4.1.

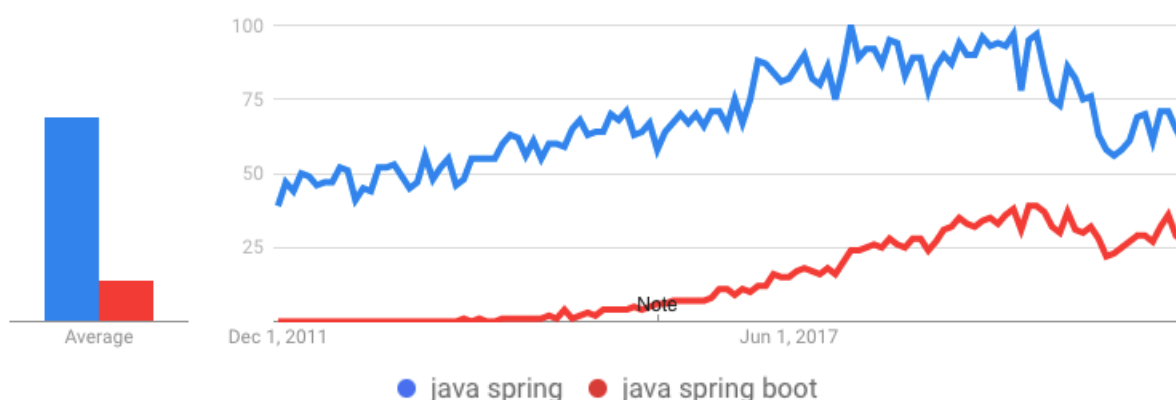


Рис. 4.1 – Популярність пошукових запитів Google на тему Spring та Spring Boot

Перша робоча публічна версія Spring Framework була представлена в березні 2004 року, а Spring Boot – в квітні 2014 року. Як бачимо зацікавленість розробників в Spring Framework останні 10 років значно зросла та продовжує утримувати свої позиції. Популярність Spring Boot почала рости з моменту публічного релізу та щороку набирає популярність. Тож, можна стверджувати, що Spring Framework та Spring Boot завоювали своє місце на ринку та продовжують нарощувати популярність.

4.2 Dropwizard

Dropwizard було створено задовго до Spring Boot. Його перший реліз, v0.1.0, вийшов у грудні 2011 року. Розробником Dropwizard є компанія Coda Hale в Yammer. Метою створення фреймворку було управління архітектурами розподілених систем компанії (мікросервісів), які використовували JVM. Фреймворк розпочався як набір коду, який об'єднував деякі потужні бібліотеки для написання веб-сервісів REST, і з тих пір розвивався, хоча й досі зберігає свою ідентичність як мінімалістичний, готовий до застосування, простий у використанні веб-фреймворк [10].

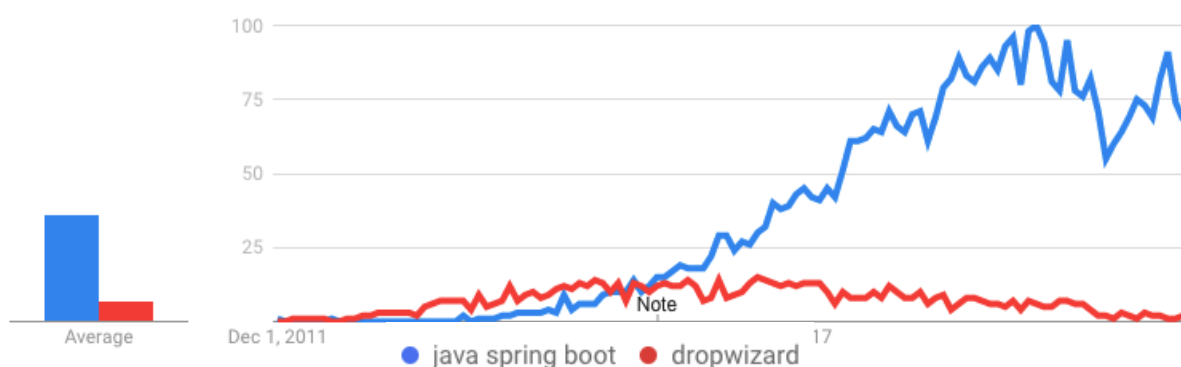


Рис. 4.2 – Популярність пошукових запитів Google на тему Spring Boot та Dropwizard

Як бачимо з рис. 4.2 зацікавленість розробників в Dropwizard до 2015 року була вищою, ніж Spring Boot, що можна пояснити тим, що Dropwizard встиг завоювати певну частину ринку, поки Spring Boot ще не став публічно доступним. Але з 2015 року бачимо поступову втрату зацікавленості в Dropwizard, та значний підйом популярності Spring Boot.

Dropwizard подібний на Spring Boot. Однак в ньому є деякі компоненти, які є лише частиною фреймворку і їх неможливо легко змінити.

Фреймворк добре підходить для написання веб-додатків та мікросервісів на основі REST без надто вигадливих надмірностей. Dropwizard обрав формати контейнера Servlet (Jetty), бібліотеки REST (Jersey), а також серіалізації та десеріалізації (Jackson). Змінити їх на якісь інші не дуже просто.

Dropwizard також не поставляється з конфігурацією введення залежностей (DI). Фреймворк виступає за те, щоб розробка мікросервісів була простою та прямолінійною, без магії залежностей. Spring Boot приховує багато проблем, оскільки сам Spring є досить складним (напр. задача налаштування всіх бінів, необхідних для запуску Spring, не є тривіальною) і приховує велику кількість взаємопов'язань бінів з анотаціями Java. Незважаючи на те, що анотації можуть бути корисними прибираючи багато коду, що повторюється, у деяких областях, під час налагодження виробничих програм, чим більше магії анотацій, тим складніше. Dropwizard вважає за краще тримати все відкритим і очевидним. Якщо потрібно протестувати код, то номери рядків і трасування стека повинні дуже добре збігатися з вихідним кодом.

Як і Spring Boot, Dropwizard вважає за краще зібрати весь проект в один, виконуваний JAR. Таким чином, розробники не турбуються про те, на якому сервері він повинен працювати, і як розгорнути та налаштувати сервер. Завантажувач класів у програмі Dropwizard є плоским (flat), що є суттєвою відмінністю від запуску програми на сервері, де може бути багато ієрархій. Визначення впорядкування завантаження класів, яке може відрізнятися між серверами, часто призводить до складного розгортання із зіткненнями залежностей та проблемами під час виконання (наприклад, `NoSuchMethodError`). Запуск мікросервісів у їх власному процесі забезпечує ізоляцію між програмами, тому можна налаштовувати кожну JVM окремо за необхідності та контролювати їх за допомогою інструментів операційної системи. Зникли виняткові ситуації збірника сміття та

OutOfMemoryExceptions, які дозволяють одній програмі видаляти цілий набір програм лише тому, що вони мають спільний простір пам'яті процесу.

Dropwizard надає деякі інтуїтивно зрозумілі абстракції поверх потужних бібліотек, з метою спрощення написання мікросервісів:

- 1) контейнер сервлетів – Jetty;
- 2) імплементація REST/JAX-RS – Jersey;
- 3) серіалізація та десеріалізація JSON файлів – Jackson;
- 4) Hibernate валідатор;
- 5) бібліотека Guava;
- 6) бібліотека метрик – Dropwizard Metrics;
- 7) логування – Logback та SLF4J;
- 8) робота з базами даних за допомогою бібліотеки JDBI.

Dropwizard маж на меті надати розробникам можливість легко перейти до написання коду. Компроміс фреймворку полягає в тому, що якщо ви хочете змінити стек технологій за замовчуванням – це не дуже просто. З іншого боку, можна швидко почати працювати доставляючи цінність бізнесу, адже Jetty, Jersey та Jackson це відомі бібліотеки виробничого рівня для написання послуг на основі REST, бібліотека Google Guava надає багато утиліт, Dropwizard Metrics це дуже потужна бібліотека показників, яка надає достатньо інформації для управління мікросервісами у виробництві.

4.3 WildFly Swarm

WildFly Swarm використовує перевірену функціональність Java EE на сервері додатків JBoss WildFly. WildFly Swarm повністю розбиває сервер WildFly на роздрібнені багаторазові компоненти, що називаються фракціями (fractions), які можна зібрати та сформувати у додаток для мікросервісів, який використовує Java EE API. Для збірки цих фракцій потрібно включити залежності у файл збірки Java Maven (або Gradle), а про все інше подбає WildFly Swarm [33].



Рис. 4.3 – Популярність пошукових запитів Google на тему Spring Boot та WildFly Swarm

Як бачимо з рис. 4.3 зацікавленість розробників в WildFly Swarm значно нижча ніж в Spring Boot. WildFly Swarm не зміг завоювати значну частину ринку, але він існує і підходить для вирішення певних завдань.

Сервери додатків та Java EE давно відомі серед корпоративних Java додатків. WildFly (раніше сервер додатків JBoss) – став сервером додатків з відкритим кодом, сумісним із ентерпрайз розробкою. Багато підприємств вкладають значні кошти в технологію Java EE, наймають талантів, а також вкладають в навчання та інструментарій та управління. Java EE завжди була

здатною допомогти розробникам створювати багаторівневі програми, пропонуючи такі функції, як сервлети (JSP), транзакції, моделі компонентів, обмін повідомленнями та цілісність. Розгортання програм Java EE упакованих як EAR, зазвичай містили багато WAR, JAR та відповідну конфігурацію. Після того, як є архівний файл Java, потрібним стає знайти сервер, переконатися, що він коректно налаштований та встановити на ньому архів. Це означало, що архіви можуть бути досить оццадливими і містити лише потрібний бізнес код. На жаль, це призвело до завищених потреб в реалізації серверів Java EE, які повинні були враховувати будь які функції, які можуть знадобитися додатку. Це також призвело до надмірної оптимізації залежностей якими потрібно ділитись та тими, які потребують ізоляції, оскільки вони змінюватимуться з різною швидкістю.

Сервер додатків надає єдину точку для управління, розгортання та налаштування кількох програм у межах одного екземпляра сервера. Зазвичай їх кластеризують для високої доступності, створюючи точні екземпляри сервера на різних вузлах. Проблеми починають виникати, коли занадто багато програм спільно використовують одну модель розгортання, єдиний процес та єдину JVM. Спротив з'являється, коли кілька команд, які розробляють програми, що працюють на сервері додатків, мають різні типи додатків, швидкість змін, продуктивність або потреби тощо. Архітектура мікросервісів передбачає швидкі зміни, інновації та автономію, а керування набором програм як єдиним, універсальним сервером на Java EE не дозволяє швидких змін. Крім того, з боку операцій стає дуже складним точне управління та моніторинг служб і програм, що працюють на одному сервері. Теоретично однією JVM легше керувати, але всі програми в JVM є незалежно розгорнутими і їх доводиться окремо керувати. Вирішення деяких із цих проблем може бути досягнуто розгортаючи лише одну програму на сервері [33].

Навіть незважаючи на те, що розгортання та управління програмами в середовищі Java EE може не відповідати середовищу мікросервісів, моделі компонентів, API та бібліотеки, які Java EE надає розробникам все ще велику цінність. Присутня можливість використовувати постійність (persistence), транзакції, безпеку, введення залежностей тощо, але потреба їх використовувати має бути передбачена по мірі необхідності.

WildFly Swarm оцінює pom.xml (або файл Gradle) і визначає, які залежності Java EE використовує мікросервіс (наприклад, CDI, обмін повідомленнями та сервлет), а потім створює JAR (так само, як Spring Boot та Dropwizard), який включає мінімальний API та реалізації Java EE, необхідні для запуску сервісу. Це дозволяє продовжувати використовувати відомі та потрібні API Java EE, а також розгортати їх як мікросервіси, так і у традиційному стилі додатків. Також можна використовувати існуючі проекти WAR, а WildFly Swarm може автоматично їх аналізувати та належним чином включати необхідні API/фракції Java EE без необхідності чітко їх вказувати. Це дуже потужний спосіб переміщення наявних програм до розгортання у стилі мікросервісів.

4.4 Eclipse Vert.X

Eclipse Vert.X це багатофункціональна платформа програм, керована подіями, яка працює на віртуальній машині Java. Її було розроблено Eclipse Foundation, реліз відбувся в грудні 2020 року, а початок роботи над Vert.x був ще у 2011 році. Цей фреймворк підтримує декілька мов. Підходить для команд що працюють з Java та Kotlin, а також JavaScript. Vert.x має набір інструментів для створення реактивних мікросервісів, що працюють на JVM [11].

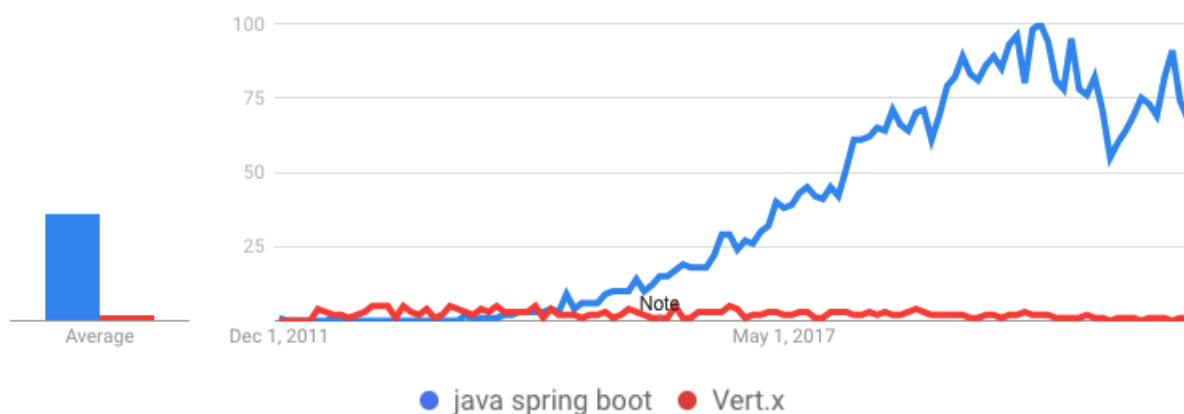


Рис. 4.4 – Популярність пошукових запитів Google на тему Spring Boot та Vert.x

Як бачимо з рис. 4.4 зацікавленість розробників в Vert.x значно нижча ніж в Spring Boot. Але публічний реліз Vert.x менше року назад, тож фреймворк може ще набрати популярності.

Eclipse Vert.x керується подіями та не блокується. Це означає, що додаток може обробляти велику кількість завдань одночасно використовуючи невелику кількість потоків ядра. Vert.x дозволяє додатку масштабуватися з мінімальним обладнанням.

Vert.x використовує бібліотеку вводу-виводу низького рівня Netty, а також включає такі функції:

- 1) компоненти програми можна писати на Java, JavaScript, Groovy, Ruby, Scala, Kotlin та Ceylon;
- 2) весь виконуваний код однопоточковий, що звільняє розробників від проблем з багатопотоковим програмування;
- 3) проста асинхронна модель програмування для написання масштабованих неблокуючих програм;

- 4) розподілена шина подій, що охоплює сторону клієнта та сервера. Шина подій також проникає в браузер JavaScript, що дозволяє створювати так звані веб-додатки в режимі реального часу;
- 5) модель актора та загальнодоступне сховище для повторного використання та спільного використання компонентів;
- 6) легковаговість – ядро Vert.x займає приблизно 650kB;
- 7) Vert.x не є сервером додатків, тобто це не монолітний інстанс на якому розгортається додаток. З його допомогою можна запускати додаток де завгодно;
- 8) підтримка модульності.

Vert.x пропонує різні компоненти та бібліотеки для створення програм мікросервісу, зокрема:

- 1) Vert.x Service Discovery: дозволяє публікувати, шукати та прив'язувати до будь якого типу сервісів;
- 2) Vert.x Circuit Breaker: забезпечує реалізацію патерну вимикача схеми;
- 3) Vert.x Config; забезпечує розширений спосіб налаштування програм;
- 4) Кластеризація та масштабованість: управління групою кластерів реалізовано в менеджерах кластерів, які можна підключити. Менеджер кластерів за замовчуванням використовує Hazelcast. Але також можна підключити Apache Zookeeper, Ignite;
- 5) Observability: компонент Vert.x Health Checks забезпечує простий спосіб перевірки стану справ;

- 6) Тестування: Vert.x Unit призначений для написання асинхронних юніт тестів з API поліглота та запуску цих тестів у JVM. Vertx Unit API запозичено з JUnit або QUnit;
- 7) Підтримка gRPC: Vert.x gRPC – це модуль, який вирівнює стиль програмування Google gRPC зі стилем Vert.x;
- 8) Служба проксі: можливість ізоляції функціоналу від одних додатків та залишення доступності для решти додатків;
- 9) Devops: підтримка додатка під час роботи у виробництві за допомогою, зокрема Micrometer, Dropwizard.

4.5 Quarkus

Цей фреймворк відносно новий і добре підходить для Kubernetes. Quarkus – це хмарний фреймворк, створений Red Hat для написання Java-додатків. Quarkus є внутрішньою платформою Java Kubernetes, розробленою для GraalVM та HotSpot, створений з найкращих бібліотек та стандартів Java. Мета Quarkus – зробити Java провідною платформою в Kubernetes та безсерверному середовищі, одночасно пропонуючи розробникам уніфіковану реактивну та імперативну модель програмування для оптимального вирішення більш широкого кола архітектур розподілених додатків [25].

У березні 2019 року Quarkus був представлений спільноті з відкритим кодом. Фреймворк може надзвичайно швидко завантажуватись та використовувати низьким обсяг пам'яті. Quarkus також пропонує майже миттєве розширення та використання великої щільності на платформах для

оркестровки контейнерів, таких як Kubernetes. За допомогою тих самих апаратних ресурсів можна запускати ще багато екземплярів програми.

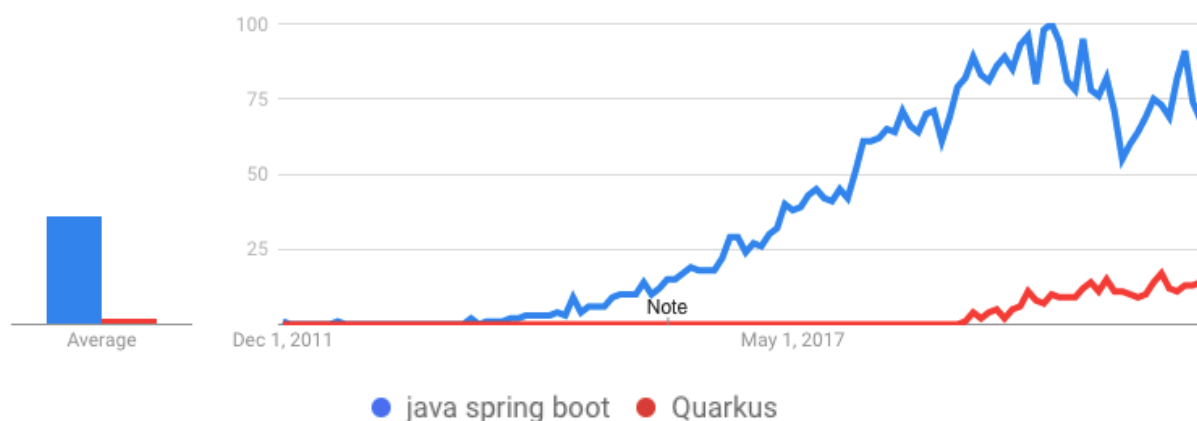


Рис. 4.5 – Популярність пошукових запитів Google на тему Spring Boot та Quarkus

Як бачимо з рис. 4.5 зацікавленість розробників в Quarkus почала з'являтися з моменту публічного релізу і продовжує зростати надалі.

З самого початку Quarkus був розроблений на основі філософії Container First та Kubernetes, що передбачає оптимізацію низького використання пам'яті та швидкого часу запуску. Під час збірки виконується якомога більше обробок. В більшості випадків увесь код, який не має шляху виконання під час роботи програми, не завантажується в JVM.

У Quarkus класи, які потрібні лише під час запуску програми, викликаються під час збірки та не завантажуються у JVM під час виконання. Quarkus також максимально уникає рефлексії, натомість надає перевагу статичному прив'язанню класу. Ці принципи проектування зменшують розмір і, в кінцевому рахунку, обсяг пам'яті програми, що працює на JVM.

Має підтримку імперативного та реактивного коду передбачену для того, щоб безперешкодно поєднувати звичний імперативний код стилю та неблокуючий реактивний стиль під час розробки програм.

4.6 Micronaut

Micronaut – це сучасна система з повним стеком мікросервісів на базі JVM, розроблена для створення модульних додатків мікросервісів, що легко тестуються [19].

Micronaut розроблений творцями фреймворка Grails і черпає натхнення з уроків, отриманих роками, створюючи реальні програми від монолітів до мікросервісів за допомогою Spring, Spring Boot та Grails.

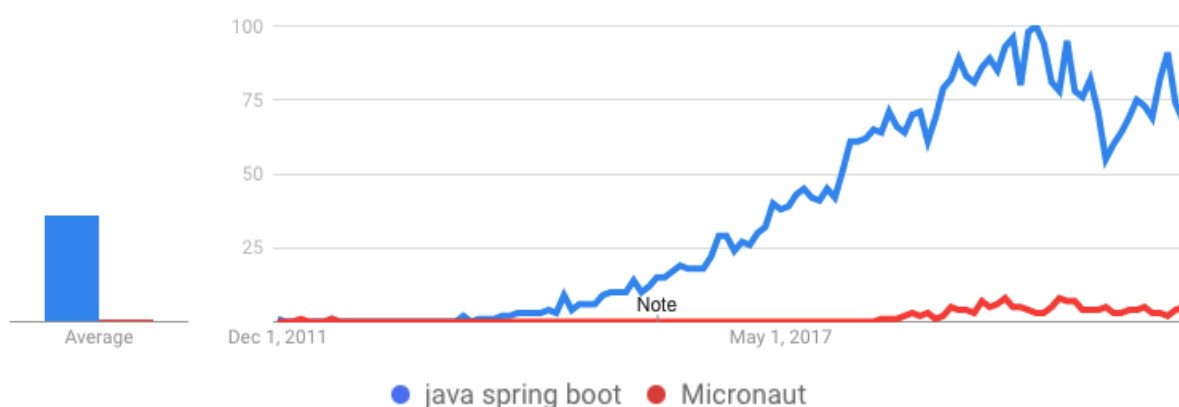


Рис. 4.6 – Популярність пошукових запитів Google на тему Spring Boot та Micronaut

Як бачимо з рис. 4.6 зацікавленість розробників в Micronaut присутня з 2019 року і тримається надалі, хоча Spring Boot значно популярніший.

Micronaut прагне надати всі інструменти, необхідні для створення повнофункціональних додатків мікросервісу, включаючи:

- 1) Ін'єкція залежностей та інверсія управління (IoC);
- 2) Чутливі значення за замовчуванням та автоматична конфігурація;
- 3) Конфігурація та спільний доступ;
- 4) Доступність сервісів;

- 5) Маршрутизація HTTP;
- 6) HTTP-клієнт з балансуванням навантаження на стороні клієнта.

Фреймворк призначений для створення мікросервісів з реалізацією безсерверного (serverless) функціоналу та з можливістю легкого тестування. Micronaut дозволяє швидко запускати програми. Ще однією особливістю Micronaut є вбудована підтримка хмарного функціоналу, з яким легко працювати на AWS.

Micronaut був розроблений на основі багаторічного досвіду від застосування Spring та Spring Boot. Як результат, багато API в рамках фреймворку натхнені Spring, що полегшує новим розробникам розібратись в Micronaut. Структура цього фреймворка позбавлена від деяких недоліків, властивих Spring та Spring Boot, та має зокрема:

- 1) знижені вимоги до обсягу пам'яті;
- 2) швидший час запуску;
- 3) просте модульне (unit) тестування;
- 4) менше використання проксі (proxies) та рефлексії (reflections).

Розділ 5

Застосування мікросервісних фреймворків

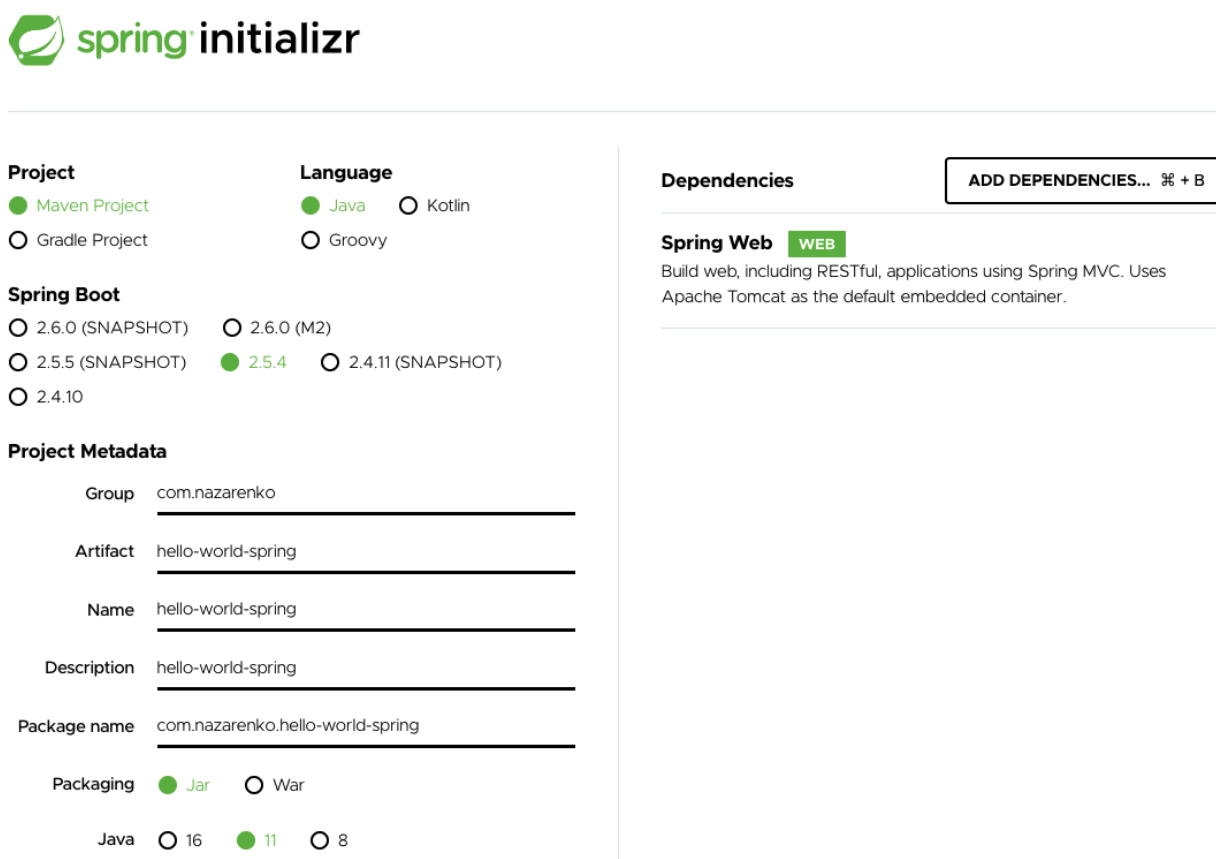
Для досягнення мети дипломної роботи розробимо базові програми на таких фреймворках, як Spring Boot, Quarkus та Micronaut, адже вони демонструють свої технологічні переваги, що підходять для досягнення мети дипломної роботи, а також завоювали популярність серед розробників. Проведемо аналіз по метрикам, які надає програма jconsole (поставляється з JDK) та часу запуску програми, а також розглянемо процес розробки. Потім оберемо найзручніший фреймворк для розробки мікросервісів та реалізуємо два незалежно працюючих мікросервіси, які матимуть функціонал спілкуватись між собою.

5.1 Spring Boot

Для початку роботи зі Spring Boot достатньо перейти за посиланням <https://start.spring.io/> на якому буде доступний ініціалізатор (spring initializr). Ініціалізатор дозволяє розробникам зручно та швидко обрати програму збірки проекту (Maven або Gradle), мову JVM (Java, Kotlin, Groovy), версію фреймворку Spring Boot, вказати метадані проекту, обрати тип архіву для запакування (Jar або War) та версію Java. А також доступно вибір великого

переліку популярних залежностей, зокрема Spring Web, Lombok, Spring Security, JDBC API, Spring Data JPA, драйвери SQL та NoSQL баз даних тощо. Додаткові залежності також можна буде додати в процесі розробки програми вказавши їх назви в файлі програми збірки проекту.

Для розробки програми оберемо Maven, Spring Boot 2.5.4, Java 11. Запакування передбачимо через Jar. Із доступних залежностей оберемо Spring Web. Початкове налаштування зображено на рис. 5.1.



The screenshot shows the Spring Initializr web application interface. It is divided into several sections for configuring a new project:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions 2.6.0 (SNAPSHOT), 2.6.0 (M2), 2.5.5 (SNAPSHOT), **2.5.4** (selected), 2.4.11 (SNAPSHOT), and 2.4.10.
- Project Metadata:** A form with input fields for:
 - Group:** com.nazarenko
 - Artifact:** hello-world-spring
 - Name:** hello-world-spring
 - Description:** hello-world-spring
 - Package name:** com.nazarenko.hello-world-spring
- Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Java:** Includes radio buttons for versions 16, **11** (selected), and 8.
- Dependencies:** A section with a button "ADD DEPENDENCIES... % + B" and a list of dependencies. The first dependency is **Spring Web** (selected), with a description: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container."

Рис. 5.1 – Стартова сторінка spring initializr

Згенерувавши проект отримуємо zip архів. Після розархівації проект потрібно запустити через інтегровану середу розробки (IDE), використаємо IntelliJ IDEA. Початкову структуру проекту зображено на рис. 5.2.

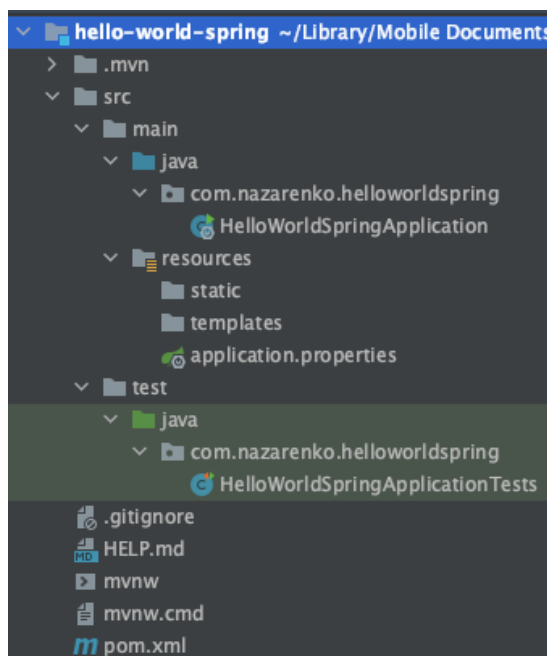


Рис. 5.2 – Початкова структура проекту Spring Boot

Початковий проект займає 122 KB пам'яті.

Добавимо контролер який по HTTP запиту GET буде повертати строку «Hello, World!». Код контролера зображено на рис. 5.3.

```
@RestController
public class HelloWorld {

    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

Рис. 5.3 – Код контролера Spring Boot

Цього коду достатньо для реалізації програми HelloWorld використовуючи Spring Boot. Наступним кроком скомпілюємо проект та запустимо сервер. Запуск проекту тривав 1,261 секунду. Проект займає 126,9 KB пам'яті.

Проаналізуємо запущений проект за допомогою jconsole. На рис. 5.4 зображено кількість класів, які програма загрузила для своєї роботи, як бачимо було загрузжено 6326 класів. На рис. 5.5 зображено розмір heap пам'яті програми, як бачимо він становить 57,1 МВ.

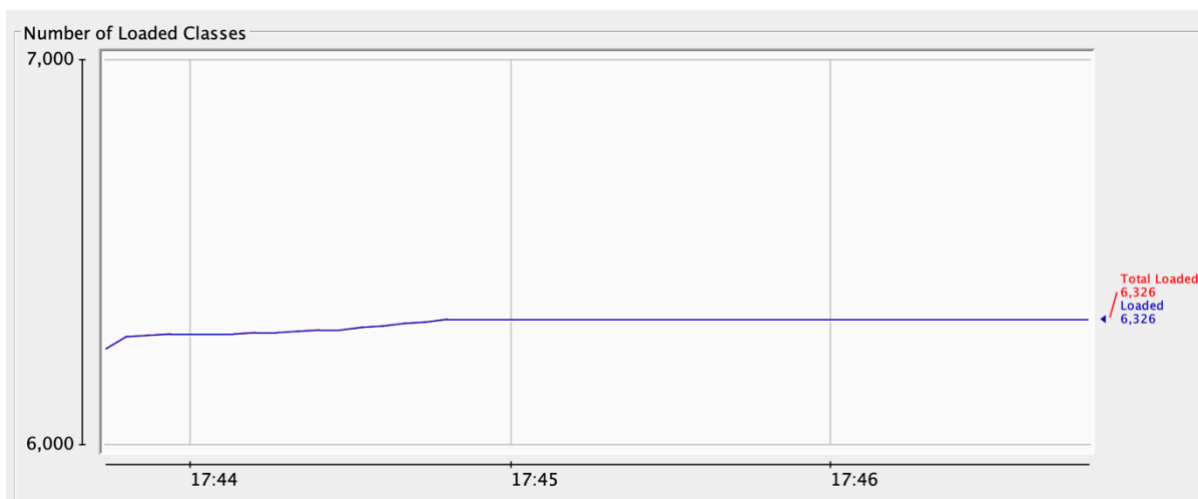


Рис. 5.4 – Кількість класів програми на Spring Boot

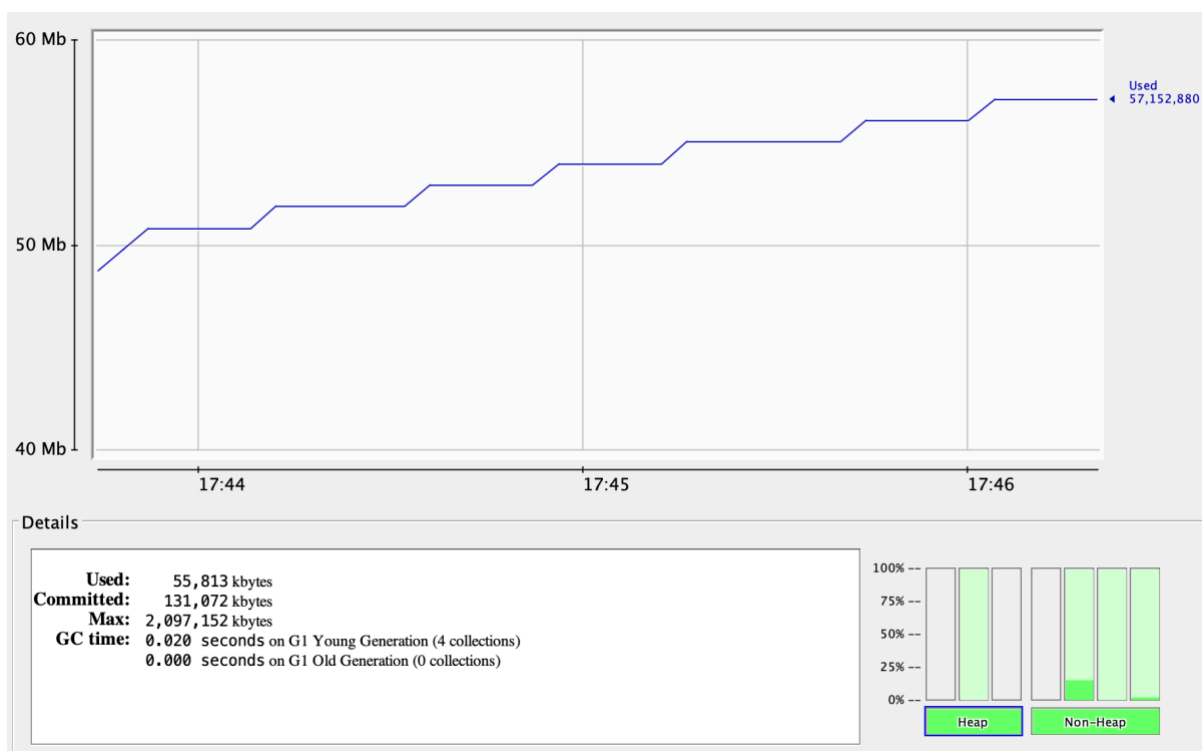


Рис. 5.5 – Розмір heap пам'яті програми на Spring Boot

За допомогою програми Postman перевіримо коректність роботи застосунку відправивши GET запит `http://localhost:8080/hello` на сервер застосунку. Результат перевірки зображено на рис. 5.6.

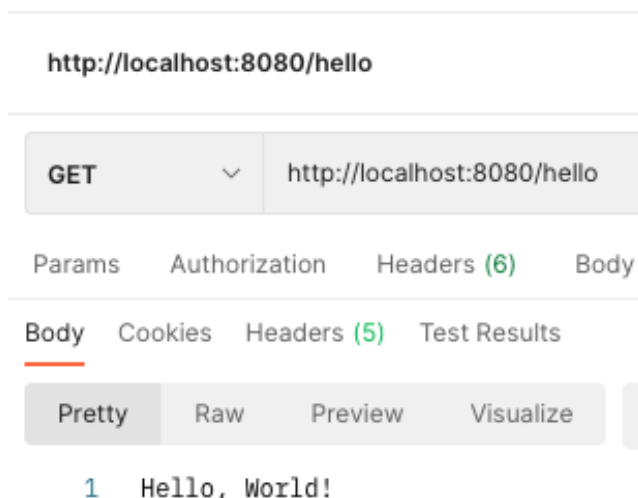


Рис. 5.6 – GET запит до програми на Spring Boot

Як бачимо у відповідь сервер повернув «Hello, World!», що свідчить про коректність роботи програми.

В процесі розробки базової програми за допомогою Spring Boot було прочитано необхідну документацію, що надається розробниками, та додаткову інформацію від спільноти фреймворку. Можна відзначити, що документація є чіткою та зрозумілою, а інформація від спільноти присутня щодо великої кількості різнобічних питань та проблем із можливими прикладами їх вирішення. Також, на сайті Spring Boot наявний навчальний розділ з описом процесу розробки та застосування певних можливостей фреймворку. База готових проектів, що надана розробниками в якості прикладів, дозволяє поглибити розуміння принципів та можливостей із баченням їх реалізації самими розробниками Spring.

Визначимо суб'єктивні показники для оцінки фреймворку:

- 1) час на опанування фреймворку: відзначу, що доступна база для навчання, наявність великої кількості демонстраційних проектів, значна кількість інформації від найбільшої спільноти розробників Spring дозволяє опанувати базові поняття фреймворку за прийнятний час та в зручному режимі;
- 2) складність вивчення: зважаючи на вищесказане, інформації для вивчення фреймворку достатньо, тому складність обмежується бажанням до вивчення. А простота запуску, велика кількість готового набору залежностей, підтримка всіх актуальних технологій тільки допомагає розпочати вивчення та роботу з фреймворком;
- 3) підтримка: Spring присутній на ринку більше 16 років, а Spring Boot – більше 6 і за цей час вони зайняли перше місце серед мікросервісних фреймворків для Java, що свідчить не лише про їх технологічні переваги, а й про вчасну підтримку та оновлення;
- 4) проекти: Spring має найбільшу кількість справжніх проектів рівня корпорацій, що працюють на JVM, а якість їх розробки забезпечена досвідом багатьох років успішної роботи.

5.2 Quarkus

Для початку роботи із Quarkus, подібно до Spring Boot, потрібно перейти за посиланням <https://code.quarkus.io/> по якому буде доступно конфігурація застосунку. В конфігураторі можна вказати назву пакетів, назву артефакту та програму збірки проекту. Також доступно вибір популярних залежностей в таких напрямках як робота з Web, Data, Messaging,

Core, Reactive, Cloud, Observability, Security, Serialization тощо. Додаткові залежності також можна буде додати в процесі розробки програми вказавши їх назви в файлі програми збірки проекту.

Для розробки програми оберемо Maven. Початкове налаштування зображено на рис. 5.7.

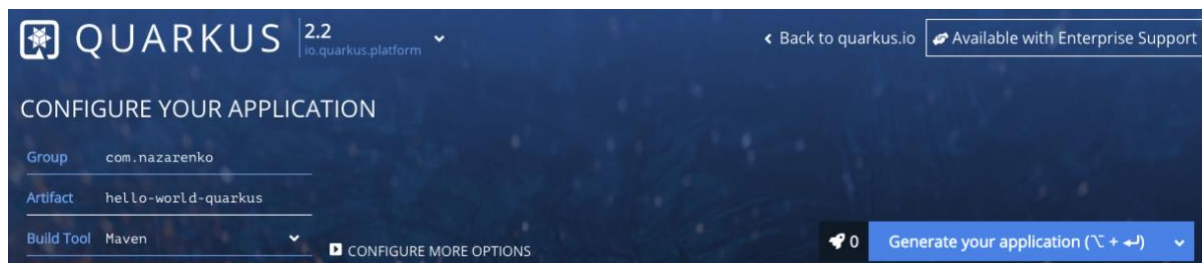


Рис. 5.7 – Стартова сторінка конфігуратора Quarkus

Згенерувавши проект отримуємо zip архів. Після розархівації проект потрібно запустити через інтегровану середу розробки (IDE), використаємо IntelliJ IDEA. Базовий проект займає 157 KB пам'яті. Початкову структуру проекту зображено на рис. 5.8.

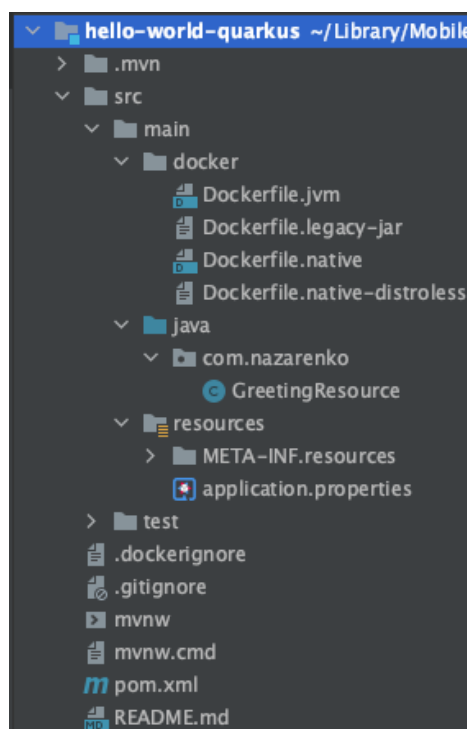


Рис. 5.8 – Початкова структура проекту Quarkus

Добавимо контролер який по HTTP запиту GET буде повертати строку «Hello, World!». Код контролера зображено на рис. 5.9.

```
@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "Hello, World from Quarkus!";
    }
}
```

Рис. 5.9 – Код контролера Quarkus

Цього коду достатньо для реалізації програми HelloWorld використовуючи Quarkus. Фреймворк дозволяє запустити сервер за допомогою технології ahead of time реалізуючи можливість змінювати код програми не перекомпілюючи весь проект. Для такого запуску потрібно з корневої папки виконати команду: `./mvnw compile quarkus:dev`. Запустимо сервер. Запуск проекту тривав 0,709 секунди. Проект займає 265,2 KB пам'яті.

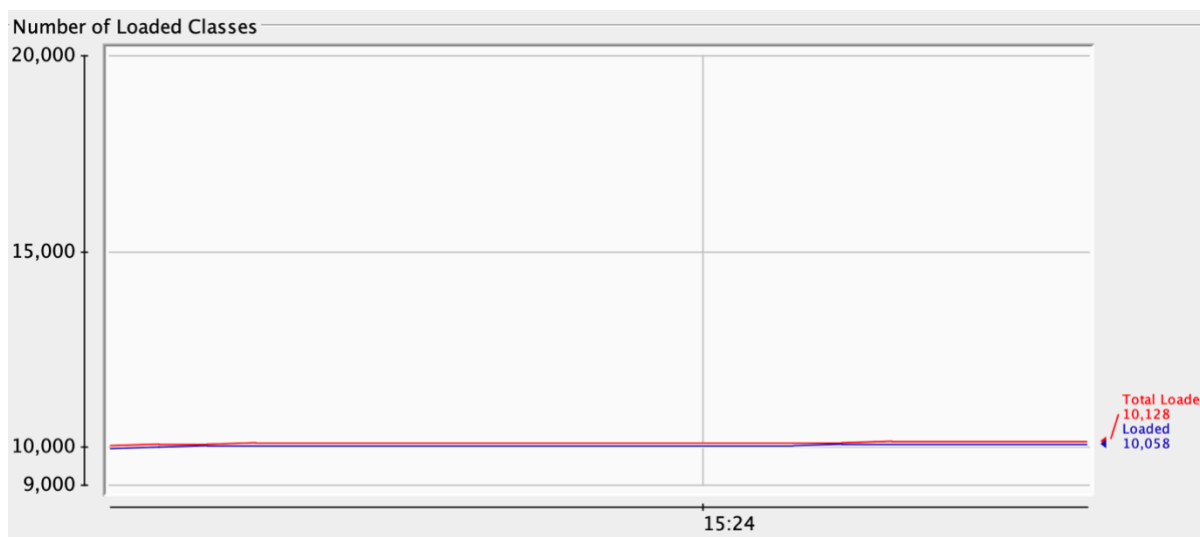


Рис. 5.10 – Кількість класів програми на Quarkus

Проаналізуємо запущений проект за допомогою jconsole. На рис. 5.10 зображено кількість класів, які програма загрузила для своєї роботи, як

бачимо було загружено 10128 класів. На рис. 5.11 зображено розмір heap пам'яті програми, як бачимо він становить 121,8 МВ.

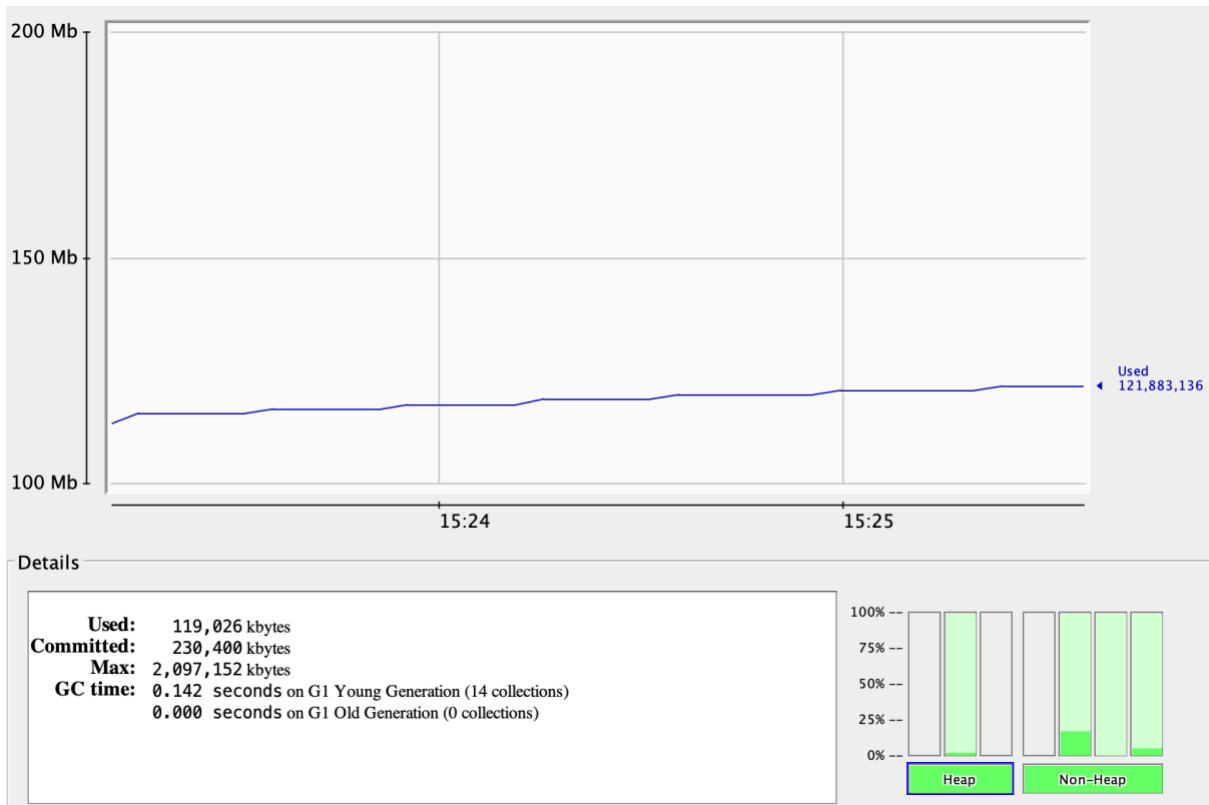


Рис. 5.11 – Розмір heap пам'яті програми на Quarkus

За допомогою програми Postman перевіримо коректність роботи застосунку відправивши HTTP GET запит *http://localhost:8080/hello* на сервер застосунку.

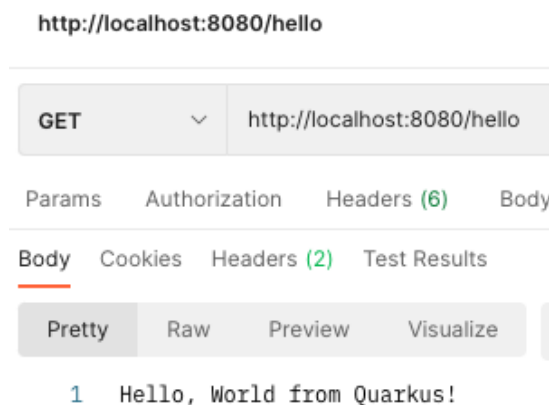


Рис. 5.12 – GET запит до програми на Quarkus

Як бачимо з рис 5.12 у відповідь сервер повернув «Hello, World from Quarkus!» що свідчить про коректність роботи програми.

В процесі розробки базової програми за допомогою Quarkus було прочитано необхідну документацію, що надається розробниками. Можна відзначити, що документація є великою та відображає всі потрібні аспекти. Інформація від спільноти користувачів Quarkus присутня в обмеженій кількості. Також порівняно з Spring Boot база готових проектів на Quarkus незначна.

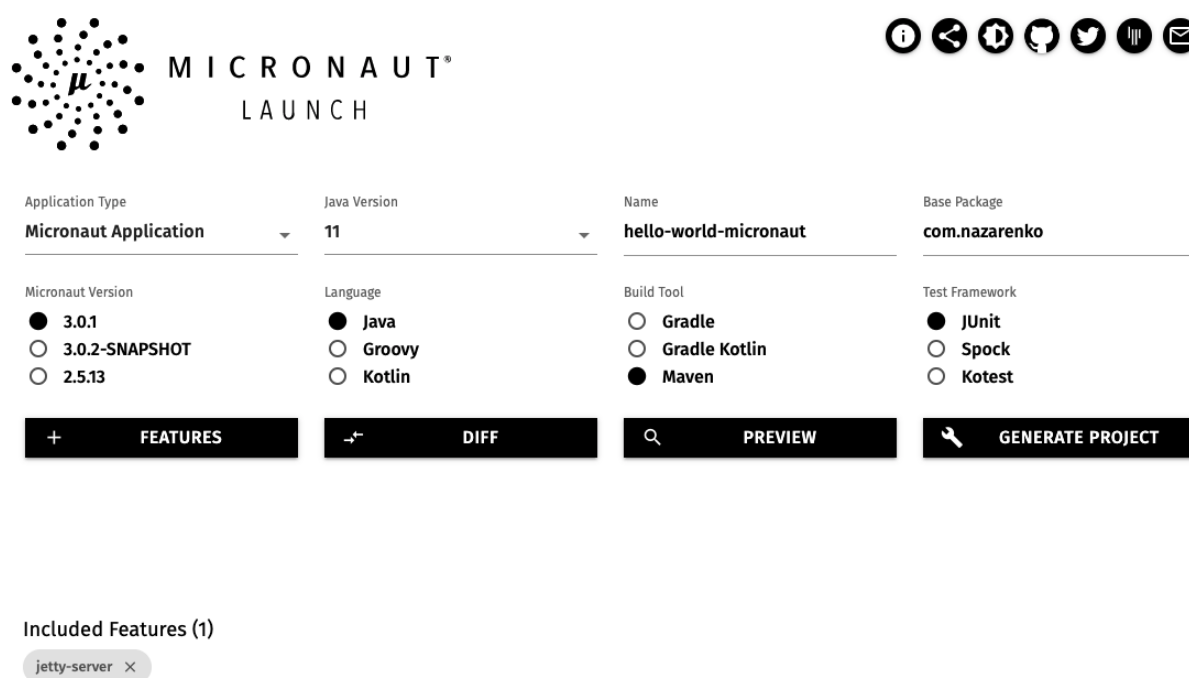
Визначимо суб'єктивні показники для оцінки фреймворку:

- 1) час на опанування фреймворку: хоча інформації в документації від розробників фреймворку достатньо, але знайти реальні приклади та вирішення проблем порівняно важче, тож необхідно більше часу на опанування фреймворку;
- 2) складність вивчення: механізми, що використовує Quarkus добре описані і зрозумілі, але процес виправлення помилок ускладнюється обмеженою базою знань, адже фреймворк відносно новий та не встиг набрати велику кількість інформації щодо нюансів реалізації того чи іншого функціоналу;
- 3) підтримка: протягом трьох років розробки фреймворк отримав 8 версій з впровадженням нового функціоналу та виправленнями наявних помилок, що може свідчити про бажання та готовність розробників задовольняти потреби ринку та завойовувати популярність;
- 4) проекти: на даний момент в мережі Інтернет присутня незначна кількість відкритих проектів.

5.3 Micronaut

Для початку роботи із Micronaut доступний сайт за посиланням <https://micronaut.io/launch/>. Лаунчер фреймворка дозволяє зручно та швидко обрати тип застосунка (Micronaut Application, Command Line Application, Function Application for Serverless, gRPC Application, Messaging-Driven Application), мову розробки (Java, Groovy, Kotlin), версію Java, версію Micronaut, програму збірки проекту (Maven, Gradle, Gradle Kotlin), фреймворк для тестів (JUnit, Spock, Kotest), а також вказати метадані проекту. Також доступно вибір великого переліку популярних залежностей. Додаткові залежності також можна буде додати в процесі розробки програми вказавши їх назви в файлі програми збірки проекту.

Для розробки програми оберемо Maven, Micronaut 3.0.1, Java 11. Додавимо залежність jetty-server.



The screenshot shows the Micronaut Launch website interface. At the top, there is a logo and navigation links. The main configuration area is divided into four columns:

- Application Type:** Micronaut Application (selected)
- Java Version:** 11 (selected)
- Name:** hello-world-micronaut
- Base Package:** com.nazarenko

Below these, there are four rows of radio button options:

- Micronaut Version:** 3.0.1 (selected), 3.0.2-SNAPSHOT, 2.5.13
- Language:** Java (selected), Groovy, Kotlin
- Build Tool:** Gradle, Gradle Kotlin, Maven (selected)
- Test Framework:** JUnit (selected), Spock, Kotest

At the bottom, there are four buttons: FEATURES, DIFF, PREVIEW, and GENERATE PROJECT. Below the buttons, there is a section titled "Included Features (1)" with a single feature listed: jetty-server.

Рис. 5.13 – Стартова сторінка Micronaut launch

Згенерувавши проект отримуємо zip архів. Після розархівації проект потрібно запустити через інтегровану середу розробки (IDE), використаємо IntelliJ IDEA. Початковий проект займає 122 KB пам'яті. Початкову структуру проекту зображено на рис. 5.14.

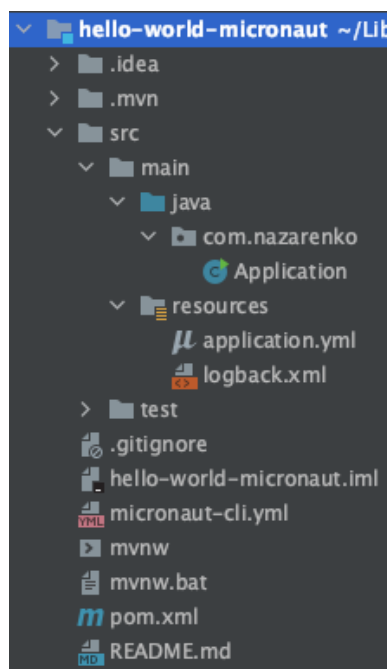


Рис. 5.14 – Початкова структура проекту Micronaut

Добавимо контролер який по HTTP запиту GET буде повертати строку «Hello, World from Micronaut!».

```
@Controller("/hello")
public class HelloWorldController {

    @Get
    public String hello() {
        return "Hello, World from Micronaut!";
    }
}
```

Рис. 5.15 – Код контролера Micronaut

Цього коду достатньо для реалізації програми HelloWorld використовуючи Micronaut. Наступним кроком скомпілюємо проект та запустимо сервер. Запуск проекту тривав 1,564 секунду. Проект займає 160,17 KB пам'яті.

Проаналізуємо запущений проект за допомогою jconsole. На рис. 5.16 зображено кількість класів, які програма загрузила для своєї роботи, як бачимо було загрузжено 6408 класів. На рис. 5.17 зображено розмір heap пам'яті програми, як бачимо він становить 31,28 МВ.

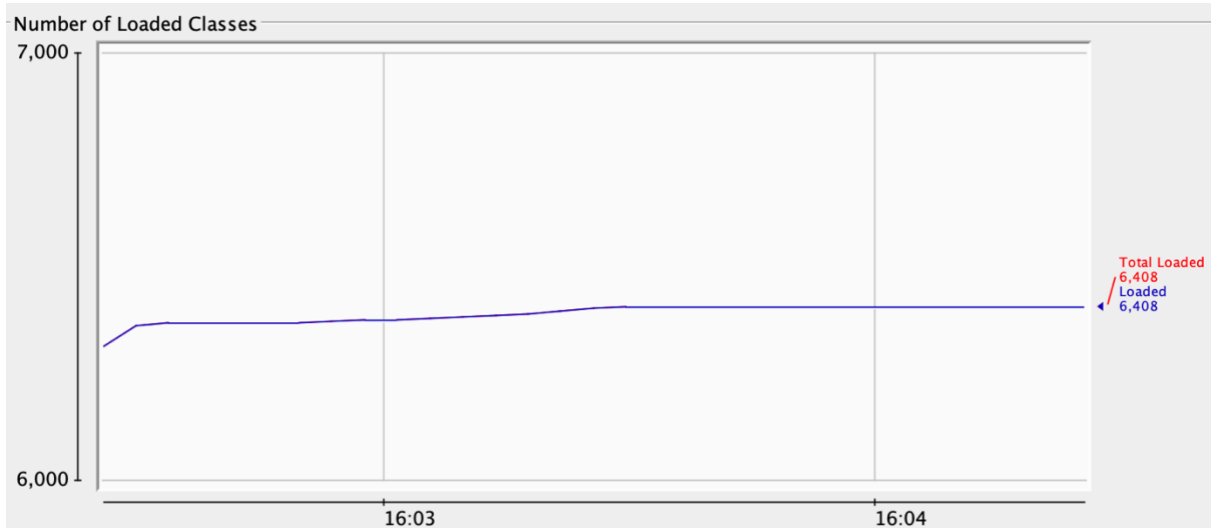


Рис. 5.16 – Кількість класів програми на Micronaut

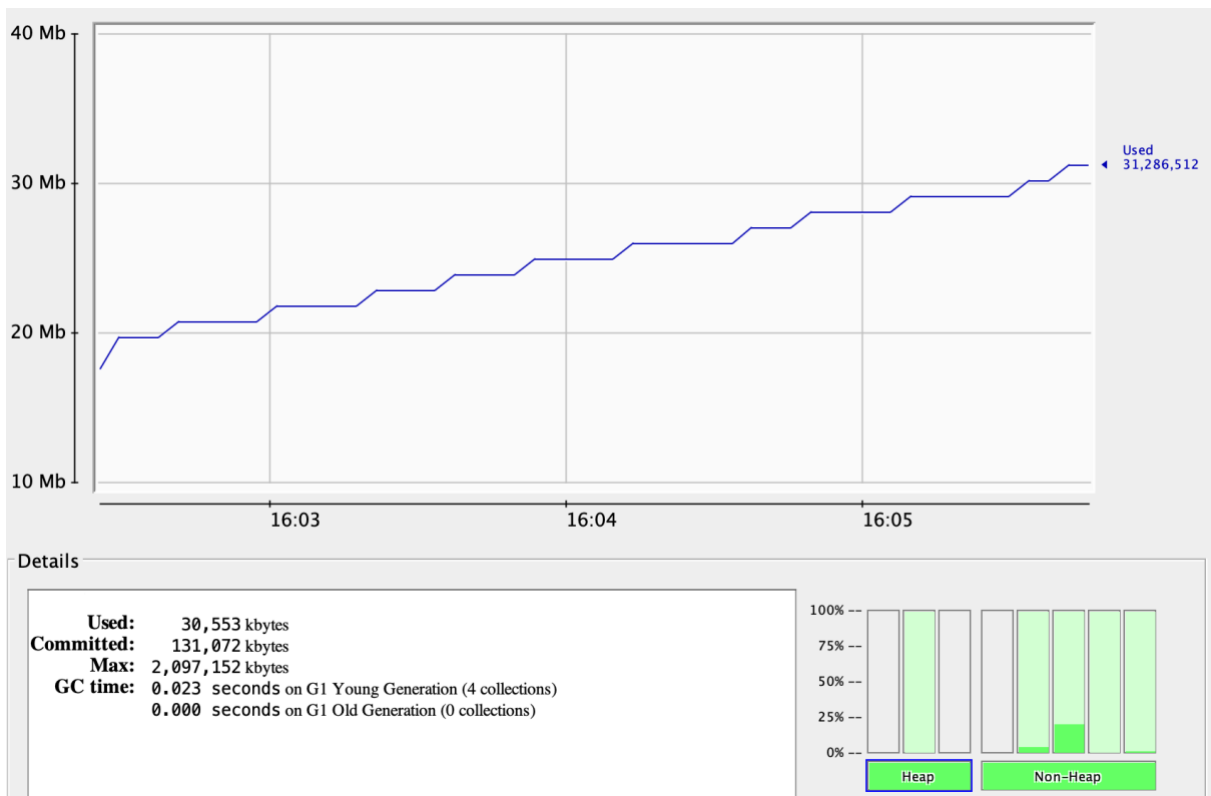


Рис. 5.17 – Розмір heap пам'яті програми на Micronaut

За допомогою програми Postman перевіримо коректність роботи застосунку відправивши HTTP GET запит `http://localhost:8080/hello` на сервер застосунку.

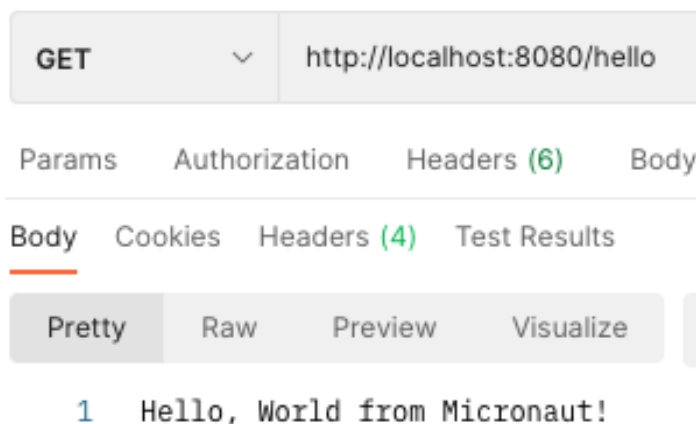


Рис. 5.18 – GET запит до програми на Micronaut

Як бачимо у відповідь сервер повернув «Hello, World from Micronaut!» що свідчить про коректність роботи програми.

В процесі розробки базової програми за допомогою Micronaut було прочитано необхідну документацію, що надається розробниками. Можна відзначити, що документація є достатньою для розуміння.

Визначимо суб'єктивні показники для оцінки фреймворку:

- 1) час на опанування фреймворку: можна відзначити, що процес опанування Micronaut подібний до процесу опанування Quarkus;
- 2) складність вивчення: принципи та механізми фреймворку хоча й відрізняються від таких у попередніх фреймворків, але не значно, що дозволяє розібратись в Micronaut відносно швидко;
- 3) підтримка: фреймворк активно підтримується розробниками виправляючи наявні помилки та додаючи новий функціонал;

4) проекти: зважаючи на меншу популярність фреймворку, кількість відкритих проектів відповідно є меншою.

5.4 Вибір мікросервісного фреймворку для розробки програми

Розробивши базові програми на таких фреймворках, як Spring Boot, Quarkus та Micronaut можна відзначити наступне. Кожен з фреймворків продемонстрував свої технологічні переваги та тонкощі в реалізації. Як бачимо підхід та логіка в створеннях програм за допомогою цих фреймворків є подібною, але водночас відрізняється ключовими словами та деталями закладеними в фреймворки їх розробниками.

Таблиця 5.1 – Порівняння мікросервісних фреймворків за метриками

	Spring Boot	Quarkus	Micronaut
Розмір вихідного коду після генерації, KB	122	157	122
Розмір вихідного коду після збірки, KB	126,9	265,2	160,17
Час запуску проекту, с.	1,261	0,709	1,564
Кількість класів запущеного проекту, од.	6326	10128	6408
Розмір heap пам'яті проекту, MB	57,1	121,8	31,3

Як бачимо з таблиці, проект на Spring Boot має найменший розмір вихідного коду після генерації та збірки та розміру heap множини пам'яті

віртуальної машини Java, а також потребує найменшу кількість класів для роботи програми. Час запуску проекту займає 1,261 секунду, що становить друге місце по швидкозапуску серед досліджуваних фреймворків.

Код Quarkus, в свою чергу, займає найбільше пам'яті, потребує 10128 класів для роботи програми та займає 121,8 MB heap простору пам'яті JVM, при цьому програма запускається найшвидше, всього лише 0,709 секунди.

Micronaut показав подібні найменші вимоги до пам'яті порівняно з іншими фреймворками, хоча кількість класів на 82 одиниці більша ніж в Spring Boot, а запуск проекту тривав найдовший час порівняно з програмами на інших фреймворках.

Також, в ході розробки базових програм було відмічено, що спільнота розробників на фреймворку Spring Boot (Spring) значно більша, ніж на інших фреймворках, а це дозволяє значно легше знайти відповіді на питання щодо нюансів реалізації функціоналу фреймворків та вирішення помилок.

Час на опанування та складність вивчення є найприйнятнішими стосовно Spring Boot. Підтримка всіх фреймворків відповідає сучасним потребам ринку розробки. Кількість доступних проектів є найбільшою в Spring, що забезпечує найширший розбір можливих варіантів реалізації програм, зважаючи на приклади як від розробників фреймворку, так і від спільноти, що використовує Spring.

Зважаючи на результат аналізу фреймворків та вимоги поставлені до реалізації програми, що забезпечить досягнення мети дипломної роботи, вибір фреймворку Spring Boot буде найоптимальнішим технологічним рішенням.

5.5 Розробка мікросервісів за допомогою Spring Boot

Програма реалізовуватиме два мікросервіси призначені для обліку складів та товарів. Перший мікросервіс матиме контекст складу, та міститиме в собі номер складу, назву складу, адресу складу, код складу. Контекст другого мікросервісу полягатиме в товарах та міститиме в собі номер товару, назву товару, номер складу на якому знаходиться товар.

Вимоги до реалізації наступні: потрібно реалізувати створення складу та товару, створюючи товар потрібно передбачити можливість вказати номер складу, на якому цей товар буде знаходитись. Також потрібно реалізувати інтерфейс для отримання інформації про склади та про товари.

Для розробки проекту буде використано мову Java версії 11, фреймворк Spring Boot версії 2.5.4 та такі залежності, як Spring Web, Spring Data JPA, H2 Database, Lombok. Програмою для збирання проекту обрано Maven. Кожен мікросервіс буде запаковано в Jar.

Розпочнемо реалізацію вимог з розробки мікросервісу складів. Даний мікросервіс має містити клас Складу, інтерфейс Репозиторію складу, сервіс та контролер складу. Також, передбачимо щоб при запуску мікросервісу на локальному комп'ютеру він мав порт 8081.

Створимо проект Spring Boot за допомогою сайту <https://start.spring.io/>. Вкажемо обрані технології та згенеруємо проект, результат зображено на рис. 5.19.



Project
☒ Maven Project
☐ Gradle Project

Language
☒ Java ☐ Kotlin
☐ Groovy

Spring Boot
☐ 2.6.0 (SNAPSHOT) ☐ 2.6.0 (M2)
☐ 2.5.5 (SNAPSHOT) ☒ 2.5.4 ☐ 2.4.11 (SNAPSHOT)
☐ 2.4.10

Project Metadata

Group	com.nazarenko
Artifact	storage-microservice
Name	storage-microservice
Description	Storage microservice.
Package name	com.nazarenko.storage

Packaging ☒ Jar ☐ War

Java ☐ 16 ☒ 11 ☐ 8

Dependencies ADD DEPENDENCIES... ⌘ + B

Spring Web WEB
 Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL
 Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL
 Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Lombok DEVELOPER TOOLS
 Java annotation library which helps to reduce boilerplate code.

Рис. 5.19 – Ініціалізація проекту Spring Boot

В пакеті `entity` створимо клас Складу, який являтиме собою сутність складів. Кожен склад матиме власний ідентифікаційний номер, генерація якого відбуватиметься автоматично при створенні складу. Конструктори буде реалізовано за допомогою анотацій Lombok.

В пакеті `repository` створимо інтерфейс який міститиме функцію знаходження складу за його ідентифікаційним номером. Для забезпечення цілісності даних інтерфейс розширимо `JpaRepository` з бібліотеки Spring Framework.

Пакет `service` міститиме клас сервісу складів, який в свою чергу міститиме репозиторій складів та функції збереження складу та пошуку складу за ідентифікатором. Також, буде реалізовано логування за допомогою анотації `Slf4j`. Логування відбуватиметься при кожному виклику функції сервісу складів.

Останнім при розробці мікросервісу складу буде реалізація контролеру складу. Клас буде анотовано як `RestController`, який кінцева точка якого, буде досяжна за посиланням `«/storages»`. В решті клас контролер повторюватиме функціонал класу сервісу складів.

Номер порту 8081 вкажемо в файлі `application.yml`, що знаходитиметься в пакеті `resources`.

Код мікросервісу складів доступний в Додатку 1.

Наступним кроком розробимо мікросервіс товарів. Даний проект міститиме такі ж залежності, як і попередній мікросервіс складів. Сутність товарів, відповідно до визначених вимог, міститиме ідентифікаційний номер, назву та код товару. За допомогою інтерфейсу репозиторію товару буде передбачено функцію знаходження товару за ідентифікаційним номером.

Для реалізації пов'язування товарів зі складами необхідним є реалізація класу товарів аналогічному, що є в мікросервісі товарів, та класу який агрегуватиме в собі товари та склади. Ці два класи міститимуться в окремому пакеті `valueObject`. Також потрібним є реалізація `RestTemplate` зі `Spring Framework` з анотацією `Bean` в головному класі мікросервісу.

Приступаючи до розробки сервісу товарів окрім збереження товару потрібно реалізувати знаходження товару зі складу по ідентифікатору товару. Для цього буде застосовано агрегуючий клас, в який за допомогою сетерів буде добавлено товар за його ідентифікатором та склад за допомогою функціоналу `RestTemplate`, що повертає об'єкт складу за параметрами URL-адресу з ідентифікатором товару та самого класу складу.

Подібно до мікросервісу складів, контролер мікросервісу товарів буде повторювати функціонал класу сервісу складів. За виключенням того, що

цей клас буде анотовано як `RestController`, який кінцева точка якого, буде досяжна за посиланням `«/commodities»`. В решті клас контролер повторюватиме функціонал класу сервісу складів.

Номер порту 8082 вкажемо в файлі `application.yml`, що знаходитиметься в пакеті `resources`.

Код мікросервісу товарів доступний в Додатку 2.

Запустимо обидва мікросервіси та протестуємо їх функціонал і коректність реалізації вимог. Мікросервіс складів було запущено за 2,474 секунди, мікросервіс товарів за 2,889 секунди. Розмір бази коду мікросервісу складів становить 155,26 KB, бази коду мікросервісу товарів 164,23 KB. Проаналізуємо кількість завантажених класів та обсяг heap простору пам'яті мікросервісів за допомогою програми `jconsole`.

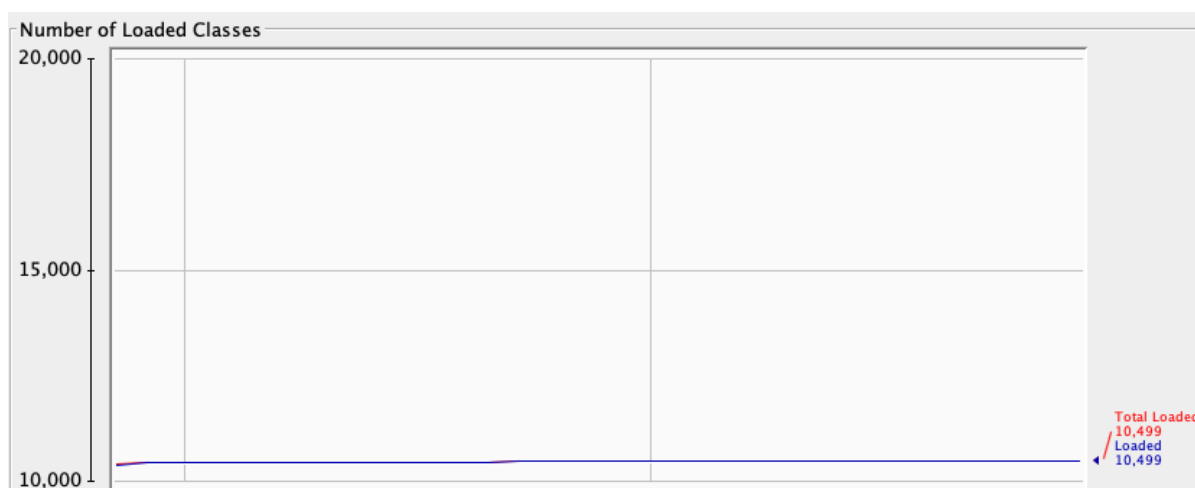


Рис. 5.20 – Кількість класів мікросервісу складів



Рис. 5.21 – Кількість класів мікросервісу товарів

Як бачимо з рис. 5.20 та 5.21 кількість класів мікросервісу складів становить 10499, а кількість класів мікросервісу товарів становить 10536. Можемо зробити висновок, що завдяки тому, що в мікросервісі товарів було передбачено додаткові класи, для пов'язування складів та товарів, зокрема RestTemplate з Spring Framework, цей мікросервіс потребує більшу кількість класів для своєї роботи.

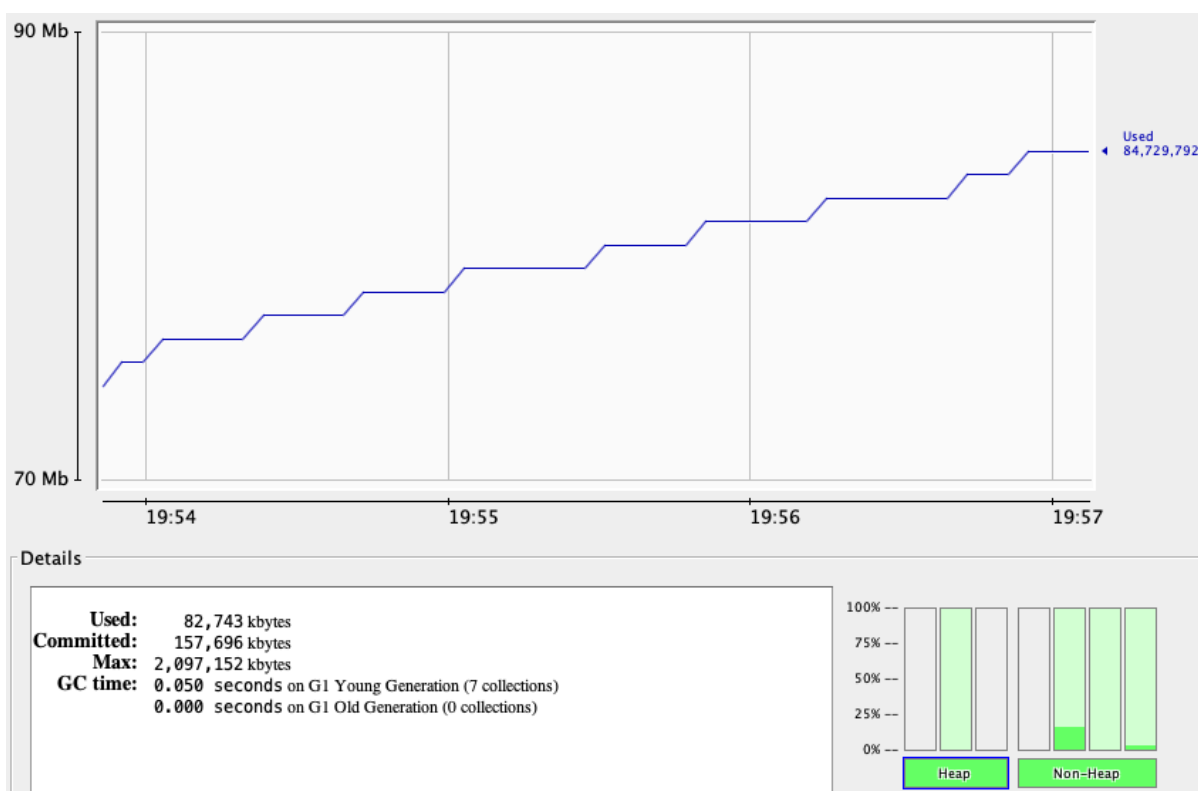


Рис. 5.22 – Розмір heap пам'яті мікросервісу складів

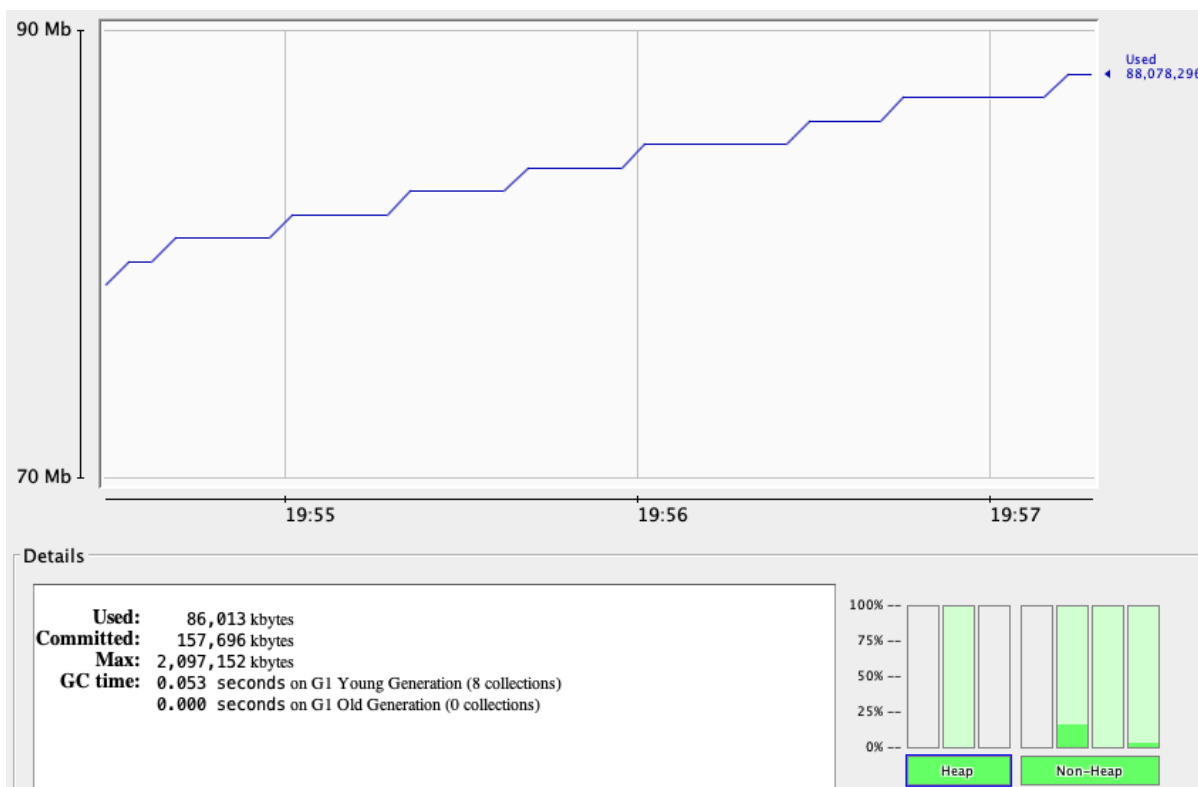


Рис. 5.23 – Розмір heap пам'яті мікросервісу товарів

Подібно до кількості класів, як бачимо з рис. 5.22 та 5.23, розмір heap простору пам'яті віртуальної машини Java в мікросервісу товарів становить 88,078 MB, що на 3,349 MB більше, ніж в мікросервісу складів.

Наступним кроком протестуємо мікросервіси за допомогою програми Postman відправляючи HTTP POST та GET запити по відповідним адресам портів мікросервісів.

Створимо новий склад відправивши POST запит за адресою *http://localhost:8081/storages/* вказавши дані про склад в JSON форматі:

```
{
  "storageName": "Computers",
  "storageAddress": "37, Peremohy Ave, Kyiv",
  "storageCode": "IT-01"
}
```

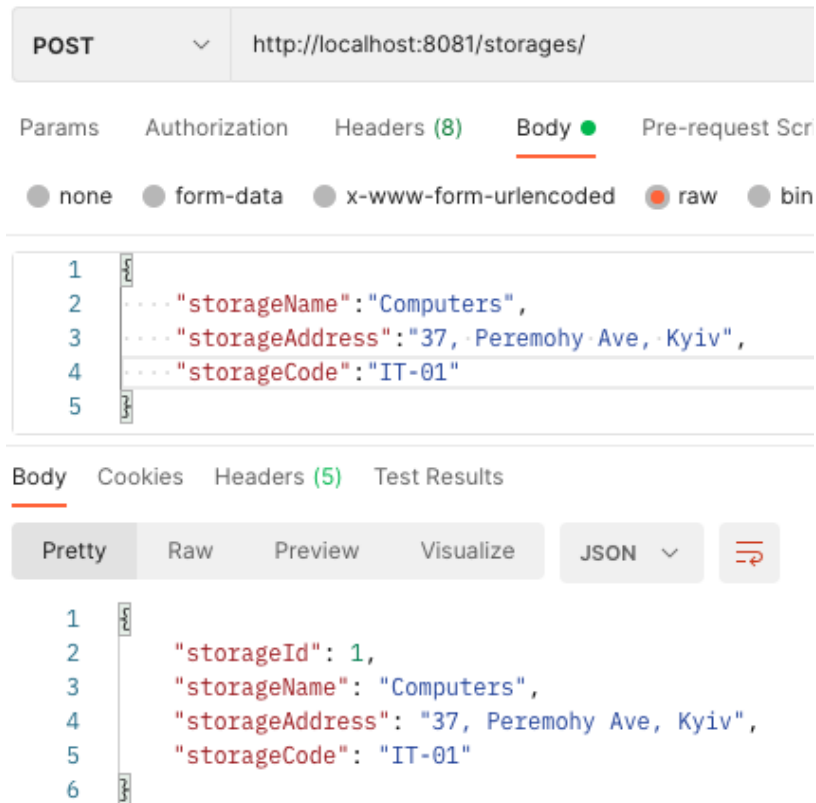


Рис. 5.24 – Створення складу через POST запит до мікросервісу складів

Перевіримо створення складу за допомогою GET запиту по адресу `http://localhost:8081/storages/1`, що зображено на рис 5.25.

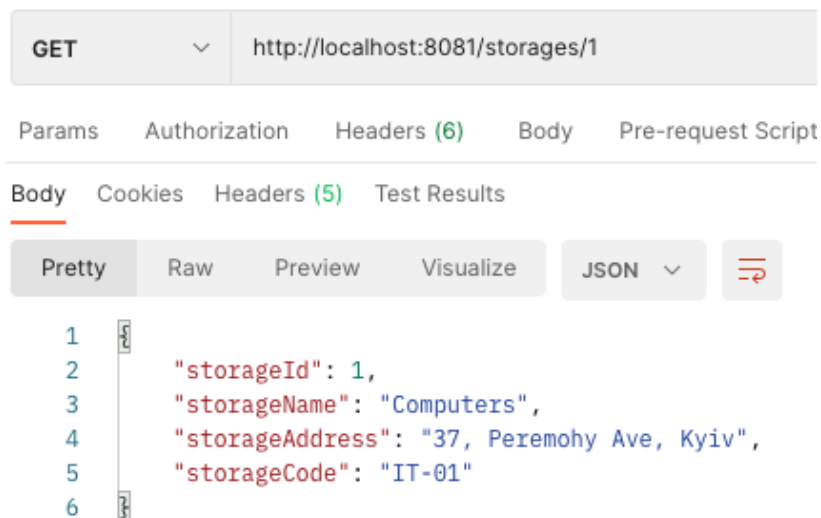


Рис. 5.25 – Стягнення даних про склад через GET запит до мікросервісу складів

Наступним кроком створимо товар, який додамо в склад з ідентифікатором № 1 за допомогою POST запиту за адресою *http://localhost:8082/commodities/* вказавши дані про товар в JSON форматі:

```
{
  "commodityName": "Apple iMac pro",
  "commodityCode": "amp1",
  "storageId": "1"
}
```

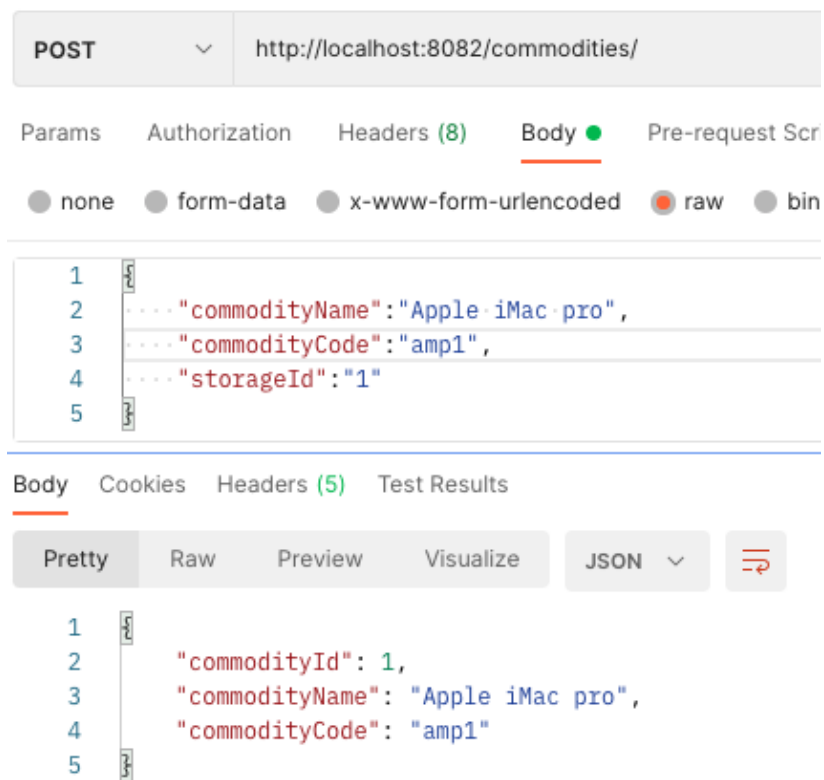


Рис. 5.26 – Створення товару через POST запит до мікросервісу товарів

Перевіримо створення складу за допомогою GET запиту по адресу *http://localhost:8082/commodities/1*, що зображено на рис 5.27.

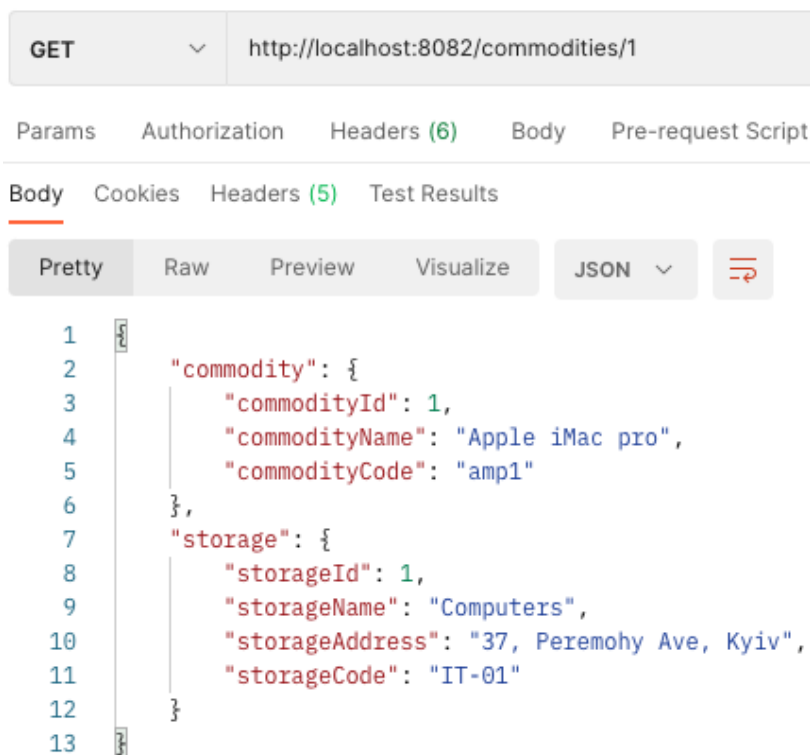


Рис. 5.27 – Стягнення даних про товар через GET запит до мікросервісу товарів

На рис 5.27 зображено відповідь в якій містяться дані про товар та склад, в якому цей товар знаходиться.

Як бачимо з результатів тестування розроблені мікросервіси працюють коректно та відповідають поставленим вимогам до розробки.

В процесі розробки мікросервісів за допомогою Spring Boot було застосовано можливості мікросервісного фреймворку, фреймворку Spring, залежностей Spring Web, Spring Data JPA, Spring H2 та Lombok. Детальна документація вказаних технологій є беззаперечною перевагою порівняно з конкурентами, а велика спільнота, що продукує корисну інформацію з відповідями на часті проблеми та помилки допомагає розробити програму відповідно до вимог.

Розділ 6

Функціонально-вартісний аналіз програмного продукту

У даному розділі проведена оцінка основних функціональних та вартісних характеристик програмного продукту, розроблено відповідно до мікросервісної архітектури.

Програмний продукт призначено для використання як незалежні мікросервіси на персональних комп'ютерах під управлінням будь-якої операційної системи.

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

Функціонально-вартісний аналіз – це технологія оцінки реальної вартості продукту або послуги незалежно від організаційної структури компанії. Прямі та побічні витрати розподіляються по продуктам та послугам у залежності від потрібних обсягів ресурсів на кожному етапі

виробництва. Виконані на цих етапах дії у контексті метода ФВА називаються функціями.

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

Метод ФВА починається з визначення послідовності функцій, необхідних для виробництва продукту. Спочатку йдуть всі можливі функції, які розподіляються по двом групам: ті, що впливають на вартість продукту і ті, що не впливають. Також на цьому етапі оптимізується сама послідовність скороченням кроків, що не впливають на витрати.

Для кожної функції визначається повний обсяг річних витрат та кількість робочих часів. На основі даних оцінок визначається кількісна характеристика джерел витрат. Після визначення джерел витрат проводиться кінцевий розрахунок витрат на виробництво продукту.

6.1 Постановка задачі техніко-економічного аналізу

Метод ФВА був застосований для проведення техніко-економічний аналізу розробки мікросервісів. Оскільки основні проектні рішення стосуються кожного окремого мікросервісу, кожна окрема підсистема має їх задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для обмін повідомленнями між мікросервісами з певною метою.

До продукту були визначені наступні технічні вимоги:

- 1) можливість виконання на персональних комп'ютерах із стандартною конфігурацією;
- 2) висока швидкість обробки запитів та надання відповіді користувачам у реальному часі;
- 3) зручність та простота взаємодії;
- 4) можливість зручного налаштування, масштабування та обслуговування;
- 5) мінімальні витрати на впровадження програмного продукту.

6.1.1 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукту, який вирішує задачу обміну повідомленнями.

Виходячи з конкретної мети, можна виділити наступні основні функції програмного продукту:

F_1 – вибір версії мови програмування Java;

F_2 – вибір середовища розробки;

F_3 – вибір мікросервісних фреймворків.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1 :

- 1) Java 8;
- 2) Java 11.

Функція F_2 :

- 1) IntelliJ IDEA;
- 2) Eclipse.

Функція F_3 :

- 1) Spring Boot;
- 2) Quarkus;
- 3) Micronaut.

6.1.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи на рисунку 6.1. Морфологічна карта відображує всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

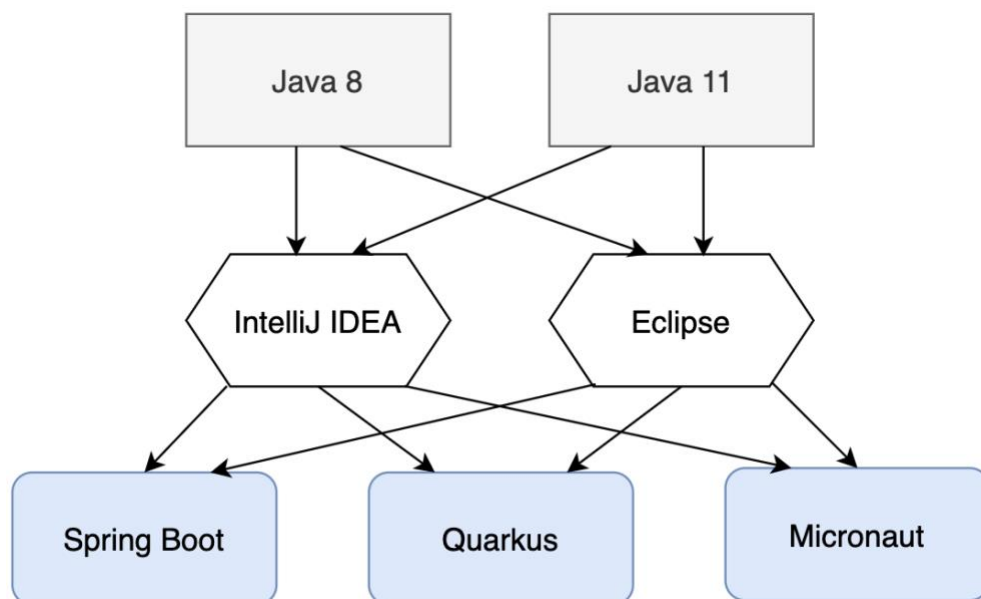


Рис. 6.1 – Морфологічна карта варіантів реалізації функцій

На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (табл. 6.1).

Таблиця 6.1 – Позитивно-негативна матриця варіантів основних функцій

Функція	Варіант реалізації	Переваги	Недоліки
F ₁	А	Має більшу базу знань, присутній довше на ринку	Офіційна підтримка закінчиться раніше
	Б	Має більший набір функціоналу, офіційна підтримка триватиме довше	Присутній на ринку меншу кількість часу
F ₂	А	Безкоштовна версія, індексація коду, наявність додаткових функцій що спрощують роботу, зручна автогенерація коду, зручніший процес тестування коду	Платна розширена версія
	Б	Безкоштовний, безкоштовна інтеграція з системами контролю версій, більша кількість плагінів	Менш зручний графічний інтерфейс користувача та загальний користувацький досвід
F ₃	А	Дуже обширна документація, постійно оновлюється, надійність, для різних задач	Довший час запуску програми
	Б	Швидший час запуску програми, АОТ компіляція	Нижчий рівень документації та підтримки
	В	Швидший час запуску програми	Нижчий рівень документації та підтримки

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

програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція F1:

Для довшої підтримки мікросервісів та ширшого функціоналу написання коду потрібно обрати варіант Б.

Функція F2:

Вибір інтегрованої середи розробки не відіграє велику роль у даному програмному продукту, тому вважаємо варіанти А та Б гідними розгляду.

Функція F3:

Вибір мікросервісного фреймворку відіграє важливу роль в написанні мікросервісів. Оскільки варіант А найкраще зарекомендував себе на ринку, існує великий ринок розробників на даному фреймворку та він відповідає всіх вимогам до програмного продукту варто обрати його.

Таким чином, будемо розглядати такі варіанти реалізації програмного продукту:

1) $F_1(B) - F_2(A) - F_3(A)$

2) $F_1(B) - F_2(B) - F_3(A)$

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

6.2 Обґрунтування системи параметрів програмного продукту

6.2.1 Опис параметрів

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

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- 1) X1 – кількість класів, що використовує фреймворк для роботи програми;
- 2) X2 – об'єм пам'яті необхідний для виконання програми;
- 3) X3 – час, який витрачається на запуск сервісу;
- 4) X4 – потенційний об'єм програмного коду, який необхідно створити безпосередньо розробнику.

6.2.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту як показано у таблиці 6.2.

Таблиця 6.2 – Основні параметри програмного продукту

Опис параметру	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Кількість класів для функціонування програми	X1	Од	15000	12000	10000
Об'єм пам'яті для функціонування програми	X2	МВ	200	120	80
Час запуску сервісу	X3	с	10	4	2
Потенційний об'єм програмного коду	X4	строк коду	1000	400	300

За даними таблиці 6.2 будуються графічні характеристики параметрів (див. рис. 6.2 - 6.5).

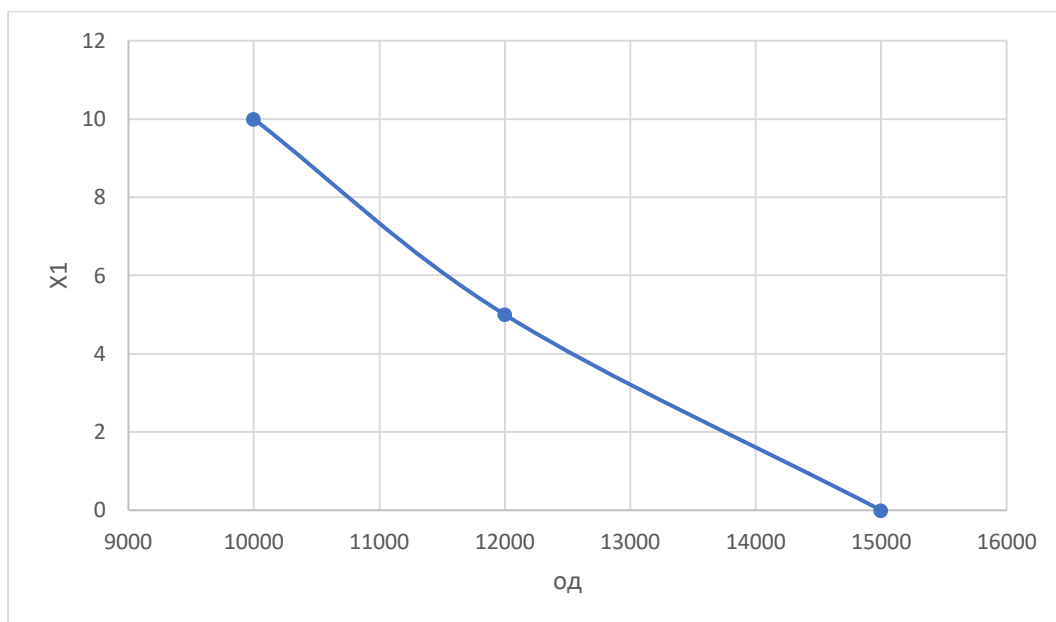


Рис. 6.2 – Кількість класів

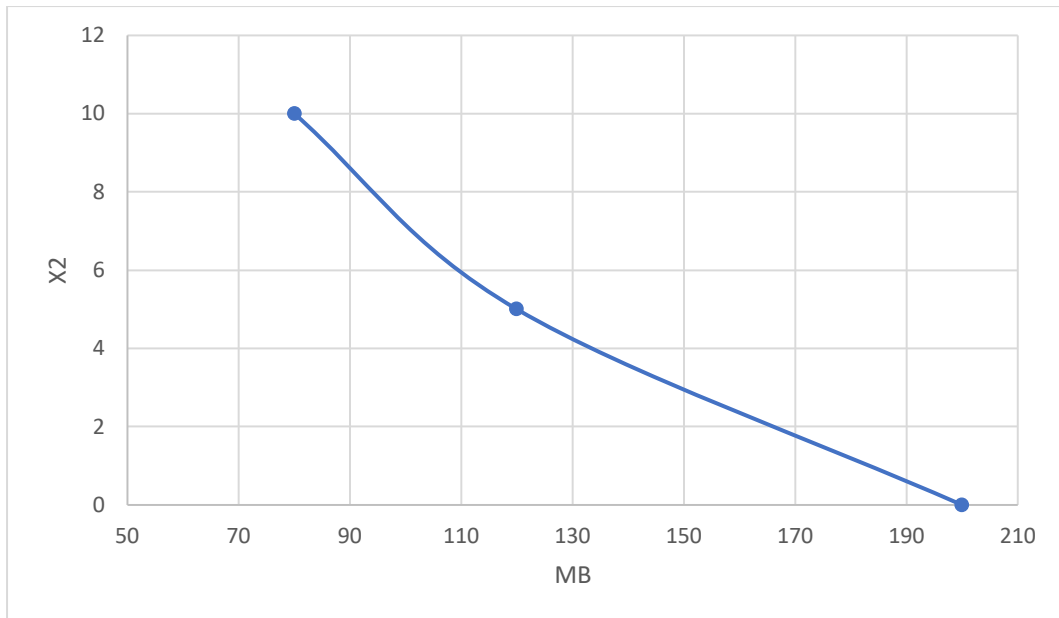


Рис. 6.3 – Об'єм пам'яті

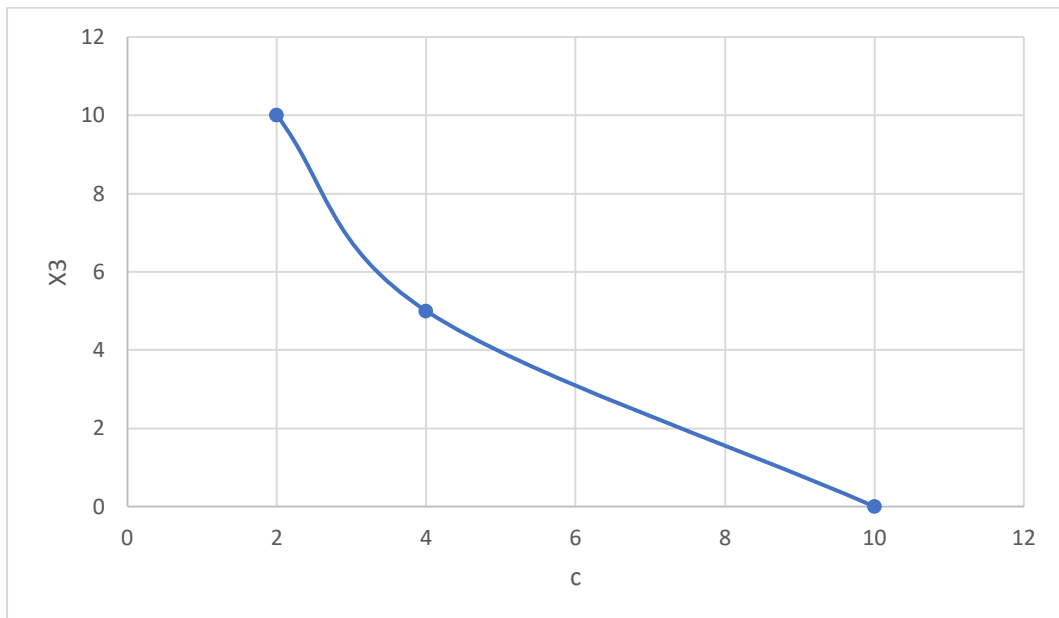


Рис. 6.4 – Час запуску

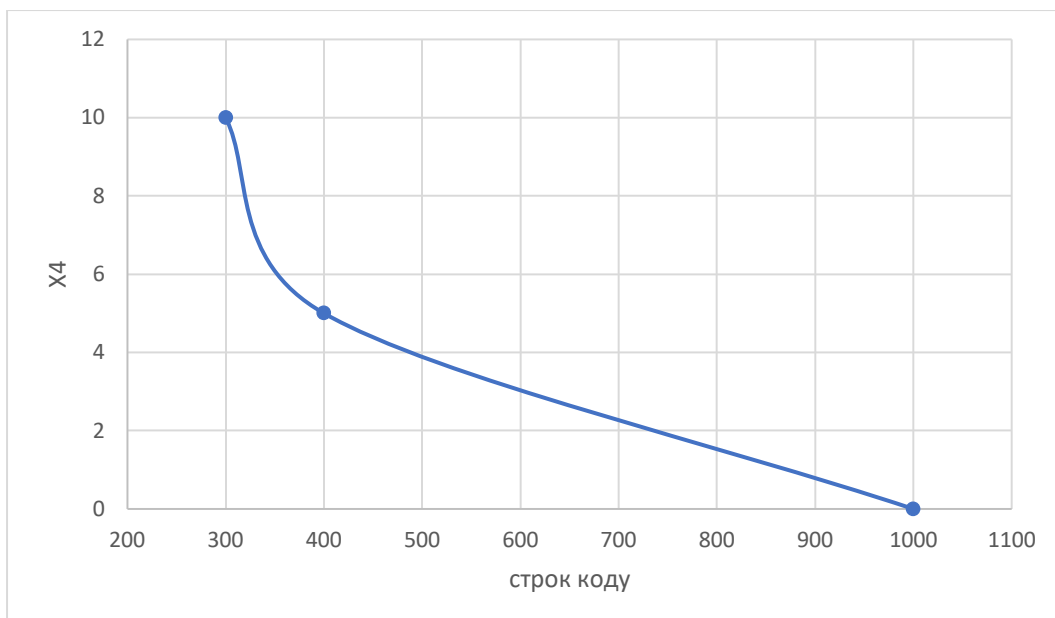


Рис. 6.5 – Потенційний об'єм програмного коду

6.2.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який має найбільш зручний інтерфейс та зрозумілу взаємодію з користувачем.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 5 людей. Визначення коефіцієнтів значимості передбачає:

- 1) визначення рівня значимості параметра шляхом присвоєння різних рангів;
- 2) перевірку придатності експертних оцінок для подальшого використання;
- 3) визначення оцінки попарного пріоритету параметрів;
- 4) обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 6.3.

Таблиця 6.3 – Результати ранжування показників

Параметр	Ранг параметра за оцінкою експерта					Сума рангів	Відхилення, Δ_i	Δ_i^2
	1	2	3	4	5			
X1	1	2	1	2	2	7	-5,5	30,25
X2	4	4	4	3	5	20	+7,5	56,25
X3	3	2	3	3	2	13	+0,5	0,25
X4	2	2	2	2	1	10	-2,5	6,25
Разом	10	10	10	10	10	50	0	93

Для перевірки ступеню достовірності експертних оцінок, визначимо наступні параметри:

- 1) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^N r_{ij} = 50,$$

де r_{ij} – ранг i -го параметра, визначений j -м експертом;

N – число експертів.

- 2) середня сума рангів T :

$$T = \frac{1}{n} R_i = 12,5.$$

- 3) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T.$$

4) загальна сума квадратів відхилення:

$$S = \sum_{i=1}^n \Delta_i^2 = 93.$$

5) коефіцієнт узгодженості (конкордації):

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 93}{5^2(4^3 - 4)} = 0,744 > W_k = 0,67.$$

Ранжирування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67. Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 6.4. Числове значення, що визначає ступінь переваги i -го параметра над j -тим, a_{ij} визначається за формулою:

$$a_{ij} = \{1,5x_i > x_j; 1,0x_i = x_j; 0,5x_i < x_j.$$

Таблиця 6.4 – Результати ранжування параметрів

Параметри	Експерти					Підсумкова оцінка	Числове значення коефіцієнтів переваги
	1	2	3	4	5		
X1,X2	<	<	<	<	<	<	0,5
X1,X3	<	<	<	<	=	<	0,5
X1,X4	<	=	<	<	>	<	0,5
X2,X3	>	>	>	=	>	>	1,5
X2,X4	>	>	>	=	>	>	1,5
X3,X4	>	=	>	=	>	>	1,5

З отриманих числових оцінок переваги складемо матрицю $A = \|a_{ij}\|$. Для кожного параметра розрахунок вагомості K_{B_i} проводиться за наступною формулою:

$$K_{B_i} = \frac{b_i}{\sum_{i=1}^n b_i},$$

де $b_i = \sum_{j=1}^N a_{ij}$ – вагомість i -го параметра за результатами оцінок всіх експертів;

a_{ij} – коефіцієнт переваги i -го на j -тим параметром.

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

$$K_{B_i} = \frac{b'_i}{\sum_{i=1}^n b'_i},$$

де $b'_i = \sum_{j=1}^N a_{ij} b_j$.

Як видно з таблиці 6.5, різниця значень коефіцієнтів вагомості після другої ітерації не перевищує 2%, тому додаткові ітерації не потрібні.

Таблиця 6.5 – Розрахунок вагомості параметрів

i	j				Перша ітерація		Друга ітерація	
	X1	X2	X3	X4	B_i	K_{B_i}	B_i^1	$K_{B_i}^1$
X1	1,0	1,5	1,5	1,5	5,5	0,344	21,25	0,36
X2	0,5	1,0	0,5	0,5	2,5	0,156	9,25	0,157
X3	0,5	1,5	1,0	0,5	3,5	0,219	12,25	0,208
X4	0,5	1,5	1,5	1,0	4,5	0,281	16,25	0,275
Разом					16,0	1,0	59,0	1,0

6.3 Аналіз рівня якості варіантів реалізації функцій

Рівень якості кожного варіанту виконання основних функцій визначається окремо.

Абсолютні значення параметрів X_1 (кількість класів), X_3 (час запуску) та X_4 (кількість строк коду) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра X_2 (об'єм пам'яті) обрано не найгіршим (не максимальним), тобто це значення відповідає варіанту б) 120 або в) 80 с.

Коефіцієнт технічного рівня якості для кожного варіанта реалізації ПП розраховується за формулою:

$$K_{TP} = \sum_{i=1}^n K_{B_i} B_i,$$

де n – кількість параметрів;

K_{B_i} – коефіцієнт вагомості i -го параметра;

B_i – оцінка i -го параметра в балах.

Розрахунок показників рівня якості представлено, відповідно, в таблиці 6.6.

Таблиця 6.6 – Розрахунок показників якості

Основні функції	Варіант реалізації	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F ₁	A	12000	5	0,36	1,8
F ₂	A	120	5	0,157	0,78
	Б	80	10	0,157	1,57
F ₃	A	4	5	0,208	1,04
F ₄	A	400	5	0,275	1,37

За цими даними визначаємо рівень якості кожного з варіантів:

$$1) F_1(B) - F_2(A) - F_3(A) = 1,8 + 0,78 + 1,04 + 1,37 = 4,99$$

$$2) F_1(B) - F_2(B) - F_3(A) = 1,8 + 1,47 + 1,04 + 1,37 = 5,68$$

Отже, найкращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

6.4 Економічний аналіз варіантів розробки програмного продукту

Для визначення вартості розробки програмного продукту спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе завдання розробки проекту програмного продукту.

Завдання за ступенем новизни відноситься до групи А. За складністю алгоритми, які використовуються в завданні належать до групи 1.

Для реалізації завдання використовує інформацію у вигляді даних. Проведемо розрахунок норм часу на розробку та програмування для завдання. Загальна трудомісткість обчислюється за формулою.

$$T_0 = T_p \cdot K_{\Pi} \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}$$

де T_p – трудомісткість розробки програмного продукту;

K_{Π} – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

K_M – коефіцієнт рівня мови програмування;

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для завдання, виходячи із норм часу для завдань розрахункового характеру ступеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $T_p = 90$ людино-днів. Поправочний коефіцієнт, який враховує вид вхідної інформації для першого завдання: $K_{\Pi} = 1,7$. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації рівний $K_{СК} = 1$. Оскільки при розробці завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $K_{СТ} = 0,8$. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 90 \cdot 1,7 \cdot 0,8 = 122,4 \text{ людино-днів.}$$

Загальна трудомісткість складає:

$$T_0 = 122,4 \cdot 8 = 979,2 \text{ людино-годин};$$

В розробці беруть участь програміст з окладом 25000 грн., один аналітик в області даних з окладом 20000 грн, та один інженер даних з окладом 15000 грн. Визначимо середню зарплату за годину за формулою:

$$C_q = \frac{M}{T_m \cdot t} \text{ грн.},$$

де M – місячний оклад працівників;

T_m – кількість робочих днів на місяць;

t – кількість робочих годин в день.

$$C_q = \frac{25000 + 20000 + 15000}{3 \cdot 20 \cdot 8} = 125 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою:

$$C_{зп} = C_q \cdot T_i \cdot K_d,$$

де C_q – величина погодинної оплати праці програміста;

T_i – трудомісткість відповідного завдання;

K_d – норматив, який враховує додаткову заробітну плату.

Зарплата розробників становить:

$$C_{зп} = 125 \cdot 979,2 \cdot 1,2 = 146880 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$C_{св} = C_{зп} \cdot 0,22 = 146880 \cdot 0,22 = 32313,6 \text{ грн.}$$

Тепер визначимо витрати на оплату однієї машино-години. Оскільки одна ЕОМ обслуговується одним інженером апаратного забезпечення з

окладом 13000 грн. та коефіцієнтом зайнятості $K_3 = 0,2$ то для однієї машини отримаємо:

$$C_{\Gamma} = 12 \cdot M \cdot K_3 = 12 \cdot 13000 \cdot 0,2 = 31200 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$C_{3П} = C_{\Gamma} \cdot (1 + K_3) = 31200 \cdot (1 + 0,2) = 37440 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$C_{CB} = C_{3П} \cdot 0,22 = 37440 \cdot 0,22 = 8236,8 \text{ грн.}$$

Амортизаційні відрахування розраховуємо за формулою при амортизації 25% та вартості ЕОМ – 40 000 грн.:

$$C_A = K_{TM} \cdot K_A \cdot C_{ПР} = 1,15 \cdot 0,25 \cdot 40000 = 11500 \text{ грн.}$$

де K_{TM} – коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

K_A – річна норма амортизації;

$C_{ПР}$ – договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо за формулою:

$$C_P = K_{TM} \cdot K_P \cdot C_{ПР} = 1,15 \cdot 0,05 \cdot 40000 = 2300 \text{ грн.}$$

де K_P – відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$T_{\text{ЕФ}} = (D_K - D_B - D_C - D_P) \cdot t \cdot K_B, T_{\text{ЕФ}} = (365 - 104 - 12 - 16) \cdot 8 \cdot 0,9 \\ = 1667,6 \text{ год.}$$

де D_K – календарна кількість днів у році;

D_B, D_C – відповідно кількість вихідних та святкових днів;

D_P – кількість днів планових ремонтів устаткування;

t – кількість робочих годин в день;

K_B – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$C_{\text{ЕЛ}} = T_{\text{ЕФ}} \cdot N_C \cdot K_3 \cdot C_{\text{ЕЛ}} = 1677,6 \cdot 0,68 \cdot 0,2 \cdot 2,92 = 666,21 \text{ грн.}$$

де N_C – середньо-споживча потужність приладу;

K_3 – коефіцієнтом зайнятості приладу;

$C_{\text{ЕЛ}}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$C_H = C_{\text{ПР}} \cdot 0,67 = 40000 \cdot 0,67 = 26800 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть складати:

$$C_{\text{ЕК}} = C_{\text{ЗП}} + C_{\text{СВ}} + C_A + C_P + C_{\text{ЕЛ}} + C_H, C_{\text{ЕК}} \\ = 37440 + 8236,8 + 11500 + 2300 + 666,21 + 26800 \\ = 86943,01 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{\text{МГ}} = \frac{C_{\text{ЕК}}}{T_{\text{ЕФ}}} = \frac{86943,01}{1667,6} = 52,14 \frac{\text{грн}}{\text{год.}}$$

Оскільки в даному випадку всі роботи, які пов'язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу складають:

$$C_M = C_{MG} \cdot T = 52,14 \cdot 979,2 = 51055,49 \text{ грн.}$$

Накладні витрати складають 67% від заробітної плати:

$$C_H = C_{ЗП} \cdot 0,67 = 146880 \cdot 0,67 = 98409,6 \text{ грн.}$$

Отже, вартість розробки програмного продукту становить:

$$\begin{aligned} C_{ПП} &= C_{ЗП} + C_{СВ} + C_M + C_H, C_{ПП} \\ &= 146880 + 32313,6 + 51055,49 + 98409,6 = 328658,69 \text{ грн.} \end{aligned}$$

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$K_{\text{ТЕР}} = \frac{K_K}{C_{ПП}} = \frac{7,16}{328658,69} = 2,178 \cdot 10^{-5}$$

6.5 Висновки

В результаті виконання економічного розділу були систематизовані і закріплені теоретичні знання в галузі економіки та організації виробництва використанням їх для техніко-економічного обґрунтування розробки методом функціонально-вартісного аналізу.

На основі даних про зміст основних функцій, які повинен реалізувати програмний продукт, були визначені два найбільш перспективні варіанти реалізації продукту. Найбільш ефективним виявився другий варіант реалізації функцій ПП, який дає максимальну величину коефіцієнта техніко-

економічного рівня, вартість витрат для нього становить $C_{\text{пп}} = 328658,69$ грн.

Цей варіант передбачає:

- 1) версію мови програмування Java 11;
- 2) інтегровану середу розробки IntelliJ IDEA;
- 3) мікросервісний фреймворк Spring Boot.

Розділ 7

Висновки та пропозиції

В ході виконання дипломної роботи першим етапом, якому посвячений перший розділ, було виявлено стан та проблематику тематики дипломної роботи. В другому розділі було розглянуто еволюцію розробки програмного забезпечення, поняття, переваги, недоліки та причини появи монолітної архітектури, сервіс орієнтованої архітектури, можливості об'єктно орієнтованого програмування та сучасні потреби ринку до розробки.

Третій розділ дипломної роботи присвячений мікросервісам. При його написанні було досліджено мікросервісну архітектуру, її характеристику, порівняно мікросервісна застосунку з монолітними та сервіс орієнтованими застосунками, проведено систематику сервісів, визначено підходи до оптимальних розмірів сервісів, методи взаємодії команд розробників при розробці сервісів, а також узагальнено переваги мікросервісної архітектури.

Четвертий розділ посвячений дослідженню стани ринку мікросервісних фреймворків та їх аналізу, зокрема таких фреймворків, як Spring Boot, Dropwizard, WildFly Swarm, Eclipse Vert.X, Quarkus, Micronaut.

В п'ятому розділі було застосовано три мікросервісних фреймворки, що були визначені як найоптимальніші для задоволення потреб до розробки

мікросервісів дипломної роботи. При застосуванні трьох фреймворків було розроблено, проаналізовано по метрикам пам'яті бази коду, швидкості запуску, кількості класів та пам'яті heap простору програм та протестовано REST API за допомогою програми Postman три базові програми з метою поглибленого практичного порівняння процесу розробки програм за їх допомогою. Наступним кроком було обрано фреймворк для розробки програми та розроблено два мікросервіси, відповідно до поставлених вимог до їх розробки. В процесі написання розділу було описано процес розробки мікросервісів за допомогою Spring Boot, визначено метрики кожного мікросервісу та протестовано їх роботу та взаємодію між собою.

В шостому розділі було проведено функціонально-вартісний аналіз програмного продукту.

Підсумовуючи результати написання дипломної роботи та виконання завдань передбачених тематикою роботи бачимо, що теми розкриті в дипломній роботі показують актуальність та значущість повноцінного розкриття тематики сучасних фреймворків для розробки мікросервісних застосунків на Java, а також незаперечні переваги та потребу в їх застосуванні у процесі бізнес розробки інформаційних систем.

Також, враховуючи процес дослідження, вивчення, аналізу та впровадження мікросервісних фреймворків в розробку мікросервісних програм можна надати пропозиції, щодо переваги фреймворку Spring Boot при виборі технологічного стеку для розробки програм, особливо корпоративного рівня.

Список використаних джерел

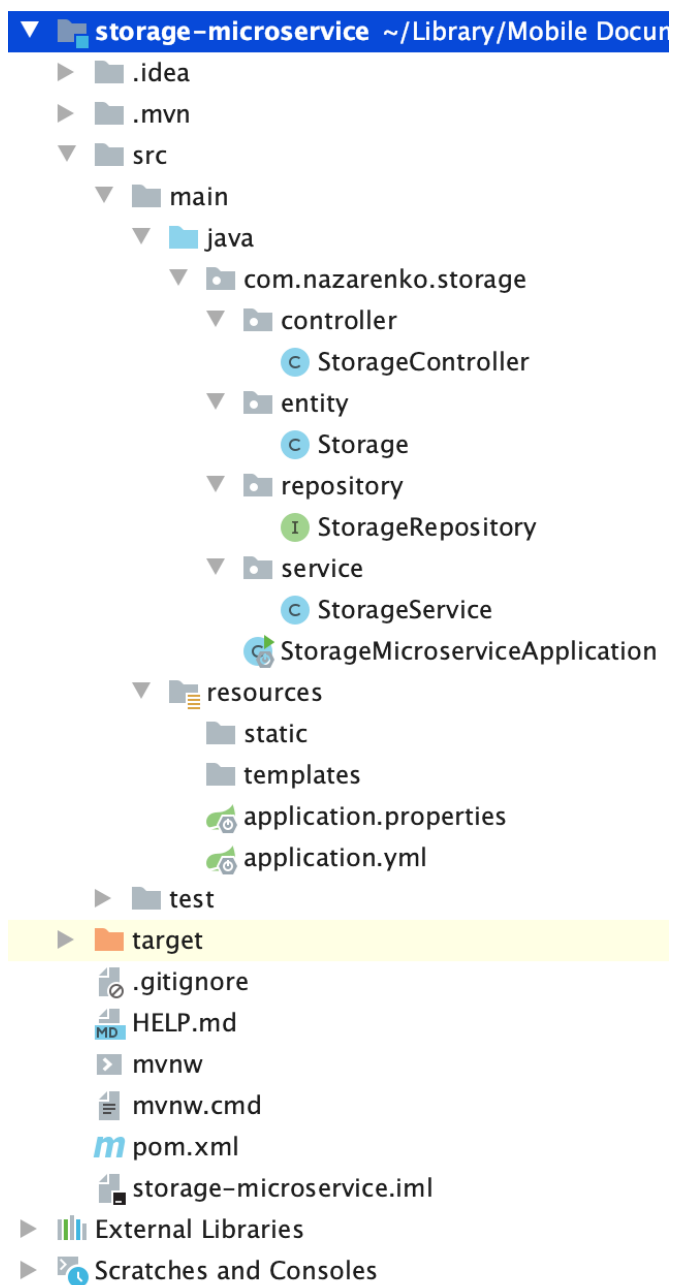
1. API Architecture: The Big Picture for Building APIs. /Matthias Biehl/ – CreateSpace Independent Publishing Platform; 1st edition, May 28, 2015. – 190 p.
2. Building Microservices: Designing Fine-Grained Systems. /Sam Newman/ – O'Reilly Media; 1st edition, March 3, 2015. – 280 p.
3. Clean Architecture: A Craftsman's Guide to Software Structure and Design. /Robert Martin/ – Pearson; 1st edition, September 10, 2017. – 432 p.
4. Clean Code: A Handbook of Agile Software Craftsmanship. /Robert Martin/ – Pearson; 1st edition, August 1, 2008. – 464 p.
5. Core Java Volume I–Fundamentals. /Cay S. Horstmann/ – Pearson; 11th edition, September 28, 2020. – 916 p.
6. Стандарти ISO. Електронний ресурс – <https://www.iso.org/standards.html>
7. Design Patterns: Elements of Reusable Object-Oriented Software. /Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, Grady Booch/ – Addison-Wesley Professional; 1st edition, October 31, 1994. – 540 p.
8. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. /Brendan Burns/ – O'Reilly Media; 1st edition, February 20, 2018. – 163 p.
9. Designing Web APIs: Building APIs That Developers Love. /Brenda Jin, Saurabh Sahni, Amir Shevat/ – O'Reilly Media; 1st edition, October 2, 2018. – 232 p.
10. Dropwizard Framework. Електронний ресурс – <https://www.dropwizard.io/en/latest/>
11. Eclipse Vert.x Reactive applications on the JVM. Електронний ресурс – <https://vertx.io/>
12. Effective Java. /Bloch Joshua/ – Addison-Wesley Professional; 3rd edition, December 18, 2017. – 414 p.

13. Engineering Economic Analysis. /Donald G. Newnan, Ted G. Eschenbach, Jerome P. Lavelle/ – Oxford University Press; 13th edition, January 20, 2017. – 740 p.
14. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. /Hohpe Gregor, Woolf Bobby/ – Addison-Wesley Professional; 1st edition, March 9, 2012. – 741 p.
15. Fundamentals of Software Architecture: An Engineering Approach. /Mark Richards, Neal Ford/ – O'Reilly Media; 1st edition, January 28, 2020. – 478 p.
16. Guide to the Software Engineering Body of Knowledge. /IEEE Computer Society, Pierre Bourque, Richard E. Fairley/ – IEEE Computer Society Press; 3rd edition, January 17, 2014. – 346 p.
17. Head First Java: A Brain-Friendly Guide. /Kathy Sierra, Bert Bates/ – O'Reilly Media; 2nd edition, February 9, 2005. – 1232 p.
18. Managing Information Systems: Managing the Digital Firm. /Laudon Kenneth C., Laudon Jane P./ – Pearson; 16th edition, January 1, 2019. – 656 p.
19. Micronaut Framework. Электронный ресурс – <https://micronaut.io/>
20. Microservice TradeOff. M. Fowler, 2015. Электронный ресурс – <https://martinfowler.com/articles/microservicetrade-offs.html>
21. Microservices a definition of this new architectural term. J. Lewis & M. Fowler, 2014. Электронный ресурс – <https://martinfowler.com/articles/microservices.html>
22. Microservices Patterns: With examples in Java. /Chris Richardson/ – Manning; 1st edition, November 19, 2018. – 520 p.
23. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. /Sam Newman/ – O'Reilly Media; 1st edition, November 14, 2019. – 327 p.
24. Object-Oriented Thought Process. /Matt Weisfeld/ – Addison-Wesley Professional; 5th edition, April 4, 2019. – 240 p.
25. Quarkus. Supersonic, subatomic Java. Электронный ресурс – <https://quarkus.io/>
26. Refactoring: Improving the Design of Existing Code. /Fowler Martin/ – Addison-Wesley Professional; 2nd edition, November 20, 2018. – 432 p.
27. Spring Boot. Электронный ресурс – <https://spring.io/projects/spring-boot>

28. Spring Framework. Электронный ресурс – https://en.wikipedia.org/wiki/Spring_Framework
29. The Pragmatic Programmer: Your Journey To Mastery, 20th Anniversary Edition. /David Thomas, Andrew Hunt/ – Addison-Wesley Professional; 2nd edition, September 13, 2019. – 352 p.
30. Unit Testing. Software Testing Fundamentals. Электронный ресурс – <http://softwaretestingfundamentals.com/unit-testing/>
31. What are Microservices? Amazon. Электронный ресурс – <https://aws.amazon.com/ru/microservices/>
32. What are microservices? IBM. Электронный ресурс – <https://www.ibm.com/cloud/learn/microservices>
33. WildFly Swarm Framework. Электронный ресурс – <https://docs.thorntail.io/2018.3.3/>

Додаток 1

Структура проекту мікросервісу складів:



Лістинг *StorageMicroserviceApplication*:

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

@SpringBootApplication
public class StorageMicroserviceApplication {

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

}
```

Лістинг *Storage*:

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Storage {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long storageId;
    private String storageName;
    private String storageAddress;
    private String storageCode;

}
```


Лістинг *StorageRepository*:

```
import com.nazarenko.storage.entity.Storage;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StorageRepository extends JpaRepository<Storage,
Long> {
    Storage findById(Long storageId);
}
```

Лістинг *StorageService*:

```
import com.nazarenko.storage.entity.Storage;
import com.nazarenko.storage.repository.StorageRepository;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
@Slf4j
public class StorageService {

    @Autowired
    private StorageRepository storageRepository;

    public Storage saveStorage(Storage storage) {
        log.info("Inside saveStorage of StorageService");
        return storageRepository.save(storage);
    }

    public Storage findStorageById(Long storageId) {
        log.info("Inside findStorageById of StorageService");
        return storageRepository.findById(storageId);
    }
}
```

Лістинг *StorageController*:

```

import com.nazarenko.storage.entity.Storage;
import com.nazarenko.storage.service.StorageService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/storages")
@Slf4j
public class StorageController {

    @Autowired
    private StorageService storageService;

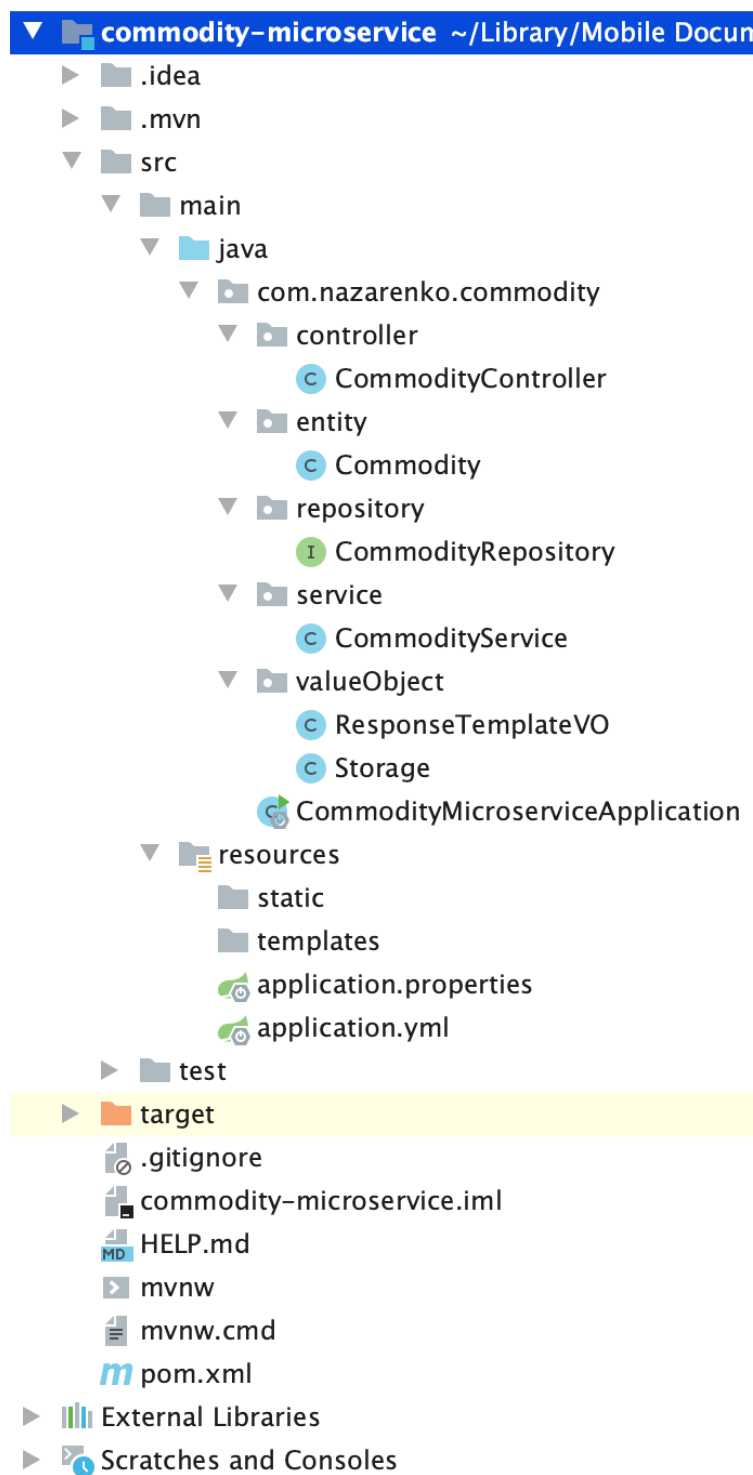
    @PostMapping("/")
    public Storage saveStorage(@RequestBody Storage storage) {
        log.info("Inside saveStorage method of StorageController");
        return storageService.saveStorage(storage);
    }

    @GetMapping("/{id}")
    public Storage findStorageById(@PathVariable("id") Long storageId)
    {
        log.info("Inside findStorageById method of StorageController");
        return storageService.findStorageById(storageId);
    }
}

```

Додаток 2

Структура проекту мікросервісу товарів:



Лістинг *CommodityMicroserviceApplication*:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

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

    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

Лістинг *Commodity*:

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Commodity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long commodityId;
    private String commodityName;
    private String commodityCode;
}
```

Лістинг *CommodityRepository*:

```
import com.nazarenko.commodity.entity.Commodity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CommodityRepository extends
JpaRepository<Commodity, Long> {
    Commodity findById(Long commodityId);
}
```

Лістинг *Storage*:

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Storage {
    private Long storageId;
    private String storageName;
    private String storageAddress;
    private String storageCode;
}
```

Лістинг *ResponseTemplateVO*:

```
import com.nazarenko.commodity.entity.Commodity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class ResponseTemplateVO {
    private Commodity commodity;
    private Storage storage;
}
```

Лістинг *CommodityService*:

```
import com.nazarenko.commodity.entity.Commodity;
import com.nazarenko.commodity.repository.CommodityRepository;
import com.nazarenko.commodity.valueObject.ResponseTemplateVO;
import com.nazarenko.commodity.valueObject.Storage;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
@Slf4j
public class CommodityService {

    @Autowired
    private CommodityRepository commodityRepository;

    @Autowired
    private RestTemplate restTemplate;

    public Commodity saveCommodity(Commodity commodity) {
        log.info("Inside saveCommodity of CommodityService");
        return commodityRepository.save(commodity);
    }

    public ResponseTemplateVO getCommodityWithStorage(Long
commodityId) {
        log.info("Inside getCommodityWithStorage of
CommodityService");
        ResponseTemplateVO vo = new ResponseTemplateVO();
        Commodity commodity =
commodityRepository.findById(commodityId);
        Storage storage =
restTemplate.getForObject("http://localhost:8081/storages/" +
commodity.getCommodityId(), Storage.class);
        vo.setCommodity(commodity);
        vo.setStorage(storage);
        return vo;
    }
}
```

Лістинг *CommodityController*:

```

import com.nazarenko.commodity.entity.Commodity;
import com.nazarenko.commodity.service.CommodityService;
import com.nazarenko.commodity.valueObject.ResponseTemplateVO;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/commodities")
@Slf4j
public class CommodityController {

    @Autowired
    private CommodityService commodityService;

    @PostMapping("/")
    public Commodity saveCommodity(@RequestBody Commodity
commodity) {
        log.info("Inside saveCommodity of CommodityController");
        return commodityService.saveCommodity(commodity);
    }

    @GetMapping("/{id}")
    public ResponseTemplateVO
getCommodityWithStorage(@PathVariable("id") Long commodityId) {
        log.info("Inside getCommodityWithStorage of
CommodityController");
        return
commodityService.getCommodityWithStorage(commodityId);
    }
}

```