ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота

на тему: «МІКРОСЕРВІСНА АРХІТЕКТУРА НА БАЗІ JAVA ТА SPRING BOOT»

на здобуття освітнього ступеня магістра
зі спеціальності __122 Комп'ютерні науки_____
(код, найменування спеціальності)
освітньо-професійної програми ___Комп'ютерні науки _____
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ ___Андрій БОНДАРЕНКО_____
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. _____

_____Андрій БОНДАРЕНКО_____
(Ім'я, ПРІЗВИЩЕ)

Керівник: __д.ф, доцент Петро КРАВЧУК_____
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Київ – 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально–науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук
Ступінь вищої освіти − «Магістр»
Спеціальність підготовки − «122 Комп'ютерні науки»
Освітньо-професійна програма − «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____Віктор Вишнівський
"____" _____2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

_____Бондаренко Андрій Олександович_____
(прізвище, ім'я, по батькові)

1.Тема кваліфікаційної роботи: «Мікросервісна архітектура на базі Java і Spring Boot»
Керівник кваліфікаційної роботи __Петро Кравчук, д.ф., доцент._____,
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу № 320 від 15.10.2024р.
2. Строк подання студентом роботи 15.12.2024 р.
3. Вихідні дані до роботи: Мікросервіси, Монолітна Архітектура, Spring Boot, Spring Framework, Java, Postgresql, Масштабованість, Гнучкість, Стійкість, Rest Api, Багатопоточність, Тестування, Взаємодія Сервісів, Api Gateway,Мікросервісний Дизайн, Jdk, Intellij Idea, Програмування, Розподілені Системи, Багатоплатформність, Інтеграція.

---

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Сучасні рішення щодо застосування мікросервісної архітектури
2. Монолітна і мікросервісна модель на приклади розробленого додатку
3. Дослідження варіантів розгортання мікросервісної архітектури
4. Дослідження варіантів вибору патернів мікросервісної архітектури

5. Перелік ілюстративного матеріалу: *презентація*
1. Мета роботи, об`єкт та предмет дослідження
2. Порівняння монолітної архітектури і мікросервісної
3. Головні відмінності мікросервісів і моноліту
4. Приклади використання моноліту
5. Приклади використання мікросервісів
6. Дослідження переваг і недоліків моноліту
7. Дослідження переваг і недоліків мікросервісів
8. Розробка додатку Quizzer
9. Версія монолітної архітектури Quizzer
10. Версія мікросервісної архітектури Quizzer
11. Тестування моноліту
12. Тестування мікросервісів
13. Рекомендації вибору
14. Висновки

6. Дата видачі завдання 05.09.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури щодо мікросервісної архітектури та Spring Boot	17–24 жовтня	вик.
2	Аналіз існуючих досліджень та методик реалізації мікросервісів	25 жовтня – 1 листопада	вик.
3	Розробка концепції додатку на Java та Spring Boot	2–10 листопада	вик.
4	Реалізація базової архітектури мікросервісного додатку	11–20 листопада	вик.
5	Тестування додатку та збір результатів	21–30 листопада	вик.
6	Вступ, висновки, реферат	1–10 грудня	вик.
7	Розробка демонстраційних креслень	11–15 грудня	вик.

Здобувач вищої освіти _____ __Андрій БОНДАРЕНКО__
(підпис) (Ім'я, ПРІЗВИЩЕ)

Керівник кваліфікаційної роботи _____ __ Петро КРАВЧУК_____
(підпис) (Ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 92 сторінки, 14 таблиць, 92 рисунка, 15 джерел.

Об'єкт дослідження – дві версії програми Quizzer (опитування), які представлені у монолітному і мікросервісному виконанні.

Предмет дослідження – мікросервісна архітектура, переваги і недоліки мікросервісної архітектури у порівнянні з монолітною.

Мета роботи – дослідження мікросервісних архітектурних характеристик у контексті ефективності, масштабованості, гнучкості та стійкості до помилок.

Короткий зміст роботи:

1. Аналіз монолітної та мікросервісної архітектури, визначення ключових аспектів обох підходів;
2. Аналіз фреймворку Spring Boot у мові Java та інших технологій, які дозволяють програмно реалізувати монолітну та мікросервісну архітектури, вибір оптимального поєднання програмних конструкцій та алгоритмів;
3. Розробка програмного додатку Quizzer на мові Java з використанням фреймворку Spring Boot, який дозволяє створювати тести для перевірки теоретичних знань програмування. Репрезентація Quizzer в монолітному і мікросервісному виглядах, тестування програми та аналіз отриманих результатів.

Метод дослідження – аналітичний, з використанням комп'ютерних технологій.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСИ, МОНОЛІТНА АРХІТЕКТУРА, SPRING BOOT, SPRING FRAMEWORK, JAVA, POSTGRESQL, REST API, ВЗАЄМОДІЯ СЕРВІСІВ, API GATEWAY, JDK, INTELLIJ IDEA.

ABSTRACT

Text part of a bachelor's thesis: 92 pages, 14 tables, 92 figures, 15 sources.

The object of research: two versions of the Quizzer software, which are presented in a monolithic and microservice version.

The subject of the study: microservice architecture, advantages and disadvantages of microservice architecture in comparison with monolithic architecture.

The purpose of the study: investigate microservice architectural characteristics in the context of efficiency, scalability, flexibility and error tolerance.

Summary of the work:

1) Analyse monolithic and microservice architectures, identify key aspects of both approaches;

2) Analysing the Spring Boot framework in Java and other technologies that allow for the software implementation of monolithic and microservice architectures, selecting the optimal combination of software structures and algorithms;

3) Development of the Quizzer software application in Java using the Spring Boot framework, which allows you to create tests to test theoretical programming knowledge. Representation of Quizzer in monolithic and microservice forms, testing of the application and analysis of the results.

The research method is analytical, using computer technologies.

KEYWORDS: MICROSERVICES, MONOLITHIC ARCHITECTURE, SPRING BOOT, SPRING FRAMEWORK, JAVA, POSTGRESQL, REST API, SERVICE INTERACTION, API GATEWAY, JDK, INTELLIJ IDEA.

ЗМІСТ

ВСТУП

Перед початком будь-якого проєкту у спеціалістів завжди виникає питання: чи буде проєкт успішним, гнучким і довговічним? Наприклад, у будівництві успіх визначається функціональністю, комфортом і довговічністю будівлі. Але що забезпечує ці ключові характеристики? Відповідь — архітектура.

Архітектура — це не лише мистецтво, а й наука, що визначає основні принципи побудови. Правильно обрана архітектура здатна закласти фундамент для успішного завершення проєкту. У наші дні цей термін поширюється далеко за межі будівництва, зокрема, у сферу інформаційних технологій. Тут архітектура відіграє ключову роль, забезпечуючи ефективну взаємодію між компонентами та сервісами програмного забезпечення.

Одними з нопулярніших підходів у проєктуванні є монолітна та мікросервісна архітектури. Монолітна архітектура нагадує традиційний будинок, де всі частини (стіни, дах, комунікації) тісно інтегровані одна з одною. Зміна одного елемента може викликати непередбачувані наслідки для всієї конструкції, але така архітектура легша у плануванні й контролі, оскільки вся система є єдиним цілим.

Натомість мікросервісна архітектура подібна до модульного будинку: кожен модуль (наприклад, кухня чи спальня) функціонує незалежно. Це дозволяє легко додавати нові модулі чи оновлювати існуючі, без ризику вплинути на інші частини системи. Така гнучкість робить мікросервісну архітектуру ідеальною для великих проєктів із частими змінами. Однак вона вимагає більше зусиль на етапі планування, адже необхідно чітко продумати взаємодію між модулями.

У підсумку, монолітна архітектура підходить для невеликих проєктів із мінімальною потребою в масштабуванні, тоді як мікросервісна — для складних систем, що потребують гнучкості й частих оновлень. Успіх проєкту багато в чому залежить від правильного вибору архітектури відповідно до масштабу та потреб системи.

Дана магістерська робота спрямована на дослідження переваг і недоліків мікросервісної архітектури та її порівняння з монолітною як альтернативою.

1 ПІДХОДИ ДО АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: МОНОЛІТ ТА МІКРОСЕРВІСИ

1.1 Монолітна архітектура: структура, переваги та недоліки

У монолітній архітектурі всі сервіси (окремі функціональні частини програми) пишуться в одній кодовій базі і працюють як єдина система. Це означає, що всі компоненти програми (від користувацького інтерфейсу до обробки даних) тісно взаємодіють і знаходяться в одному місці, утворюючи суцільну систему, за що і була надана назва монолітна архітектура — асоціація з геологічним утворенням, що являє собою цілісну кам'яну брилу. Згідно з Кембриджським словником, прикметник «монолітний» також означає «занадто великий і не піддається зміні» [1]. Зв'язок між користувачем і монолітом здійснюється через балансувальник навантаження — спеціальний компонент, який розподіляє вхідний трафік (запити від користувачів) між різними інстанціями програми. Даний підхід допомагає уникнути перевантаження серверів, оптимізує використання ресурсів і підтримує стабільну продуктивність системи. Як заведено, монолітна програма оперує тільки однією базою даних, через яку обмінюються інформацією всі тісно інтегровані компоненти програми.

Приклад класичної монолітной архітектури зображено на рис. 1.1.

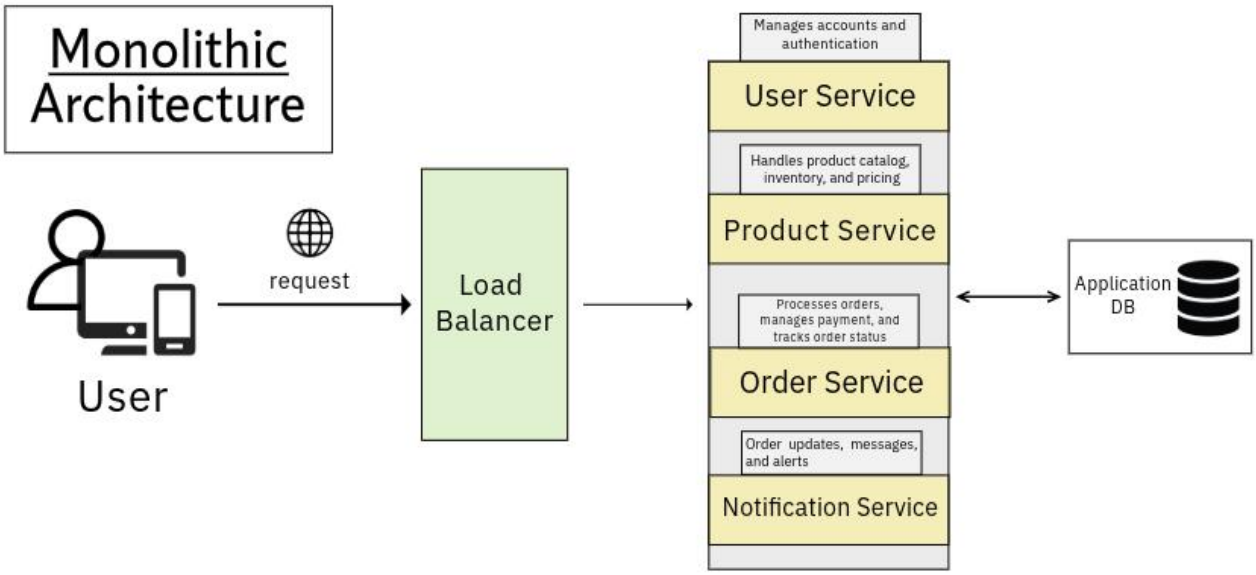


Рисунок 1.1 - Монолітна архітектура

Монолітна архітектура зазвичай вибирається для малих додатків, оскільки вона має низьку вартість реалізації і є простішою для розробки. Всі компоненти працюють в одній кодовій базі, тому немає необхідності в складній взаємодії між модулями (окремими частинами програми). Це позитивно впливає на контроль версій (управління змінами в коді), вибір ресурсів під час розробки і тестування програми. Тому монолітна архітектура є оптимальним варіантом для стартапів або малих команд, яким необхідна швидка і недорога розробка програмного забезпечення.

Попри переваги, монолітна архітектура має свої обмеження. Вона обмежує гнучкість у виборі технологічного стеку (суми інструментів і технологій для розробки програми), стійкість (можливість витримувати високі навантаження і помилки) і масштабованість (можливість розширення програми). Оскільки всі компоненти пов'язані між собою, стає важко масштабувати окремі частини системи без впливу на всю систему. Якщо монолітна система була написана на Java, то ви не зможите використовувати Python для масштабування її. Крім того, з ростом складності програми та збільшенням розміру команди, моноліт втрачає свою ефективність, і підримка його стає важчою. Розробка монолітного додатку потребує більшої уваги до чіткого розподілу завдань і обов’язків у команді, так як розподілення обов’язків всередені команди, наприклад команда А працює над сервісом авторизації, а команда В відповідальна за сервіс каталогу продуктів, створює ймовірність до похибок через саму сутність уніфікації в монолітній системі. Навіть незважлива помилка розробника в одному з рядків коду в сервісі призведе до критичної відмови або неправильної роботи інших сервісів, що призводить до краху всієї системи. Крім того, прикладом компанії, що впровадила зі своїм розвитком і зростанням є eBay, яка напочатку використовувала монолітну архітектуру, але зі збільшенням комплексності, компанія дійшла висновку, що їхня система маркетплейсу потребує модернізації, а з монолітом, у гігантській мережі залежностей і тісних відносин між компонентами, це реалізувати дуже складно і невигідно. Невагаючись eBay почала перехід з моноліту на мікросервіси

у 2006 році, розділивший свої монолітні сервіси на окремі незалежні. Таким чином, за даними сайту AltIndex, eBay отримує 69 мільйонів відвідувань кожен місяць, що є незапечним фактом популярності маркетплейсу серед онлайн-споживачив і показником ефективності мікросервісів.

1.2 Мікросервісна архітектура: структура, переваги та недоліки

В сучасному світі інформаційних технологій, де системи постійно розвиваються та ускладнюються, питання ефективної архітектури та управління даними стає все більш актуальним[2]. На сьогоднішній день, відомі великі компанії перейшли на використання мікросервісної архітектури замість монолітної. Дане архітектурне рішення пов’язане із швидким технологічним прогресом в сфері інформаційних технологій. Тож щоб залишатися конкурентоспроможними на ринку, нові послуги повинні швидко та ефективно впроваджуватися задля збереження попиту серед користувачів. Зі слів видатного інженера та автора книг Мартіна Фаулера: “Мікросервісний архітектурний стиль — це підхід до розробки єдиного додатка як набору сервісів, кожен із яких працює у власному процесі й взаємодіє з іншими сервісами за допомогою легких механізмів, часто через HTTP/REST API”[3]. Таким чином, наявність окремих сервісів — основна перевага мікросервісної архітектури над монолітною архітектурою, де всі компоненти тісно пов’язані й залежні один від одного, що ускладнює додавання нових функцій у великих програмах з комплексною логікою.

Сервіси формуються на основі функціональних можливостей бізнесу, а не для окремої технічної функції. Узявши за приклад онлайн магазин, тоді сервіси такі як: продукти, кошик, особистий кабінет, розрахунок, тощо, будуть реалізовувати функції бізнесу з продаж в інтернеті. Неприпустимо, щоб незалежні сервіси було об’єднано в єдиний сервіс, приміром, об’єднання сервісів “оплата” і “кошик” знижуватиме ефективність системи через створення залежності між сервісами. Обслуговування відбувається окремо для кожного з сервісів. Так як сервіси незалежні, вони мають власні актуальні версії. Конфігурація версій значно зпрощує орієнтування в оновлені застарілих сервісів і проєктуванні майбутніх

версій.

Мікросервісна архітектура була б марною, якщо сервіси не мали б змоги спілкуватися між собою, але на щастя мікросервісна архітектура використовує достатній вибір способів взаємодії між сервісами. Є три сучасних типів взаємодії: синхронна, асинхронна і Service Mesh. Синхронна взаємодія (рис 1.2) — сервіс-одержувач звертається до сервісу-відправника, очікуючи на відповідь. Дана взаємодія є найпоширенішою, сервіси використовують HTTP/REST API запити (GET, POST, PUT, DELETE). Але існує більш швидкий протокол у порівнянні з REST — це gRPC, заснований на протоколі HTTP/2. Синхронна взаємодія має перевагу в простоті реалізації та забезпечує негайну відповідь вхідним запитам, але сплачує, як недолік, збільшенням затримок у системі.

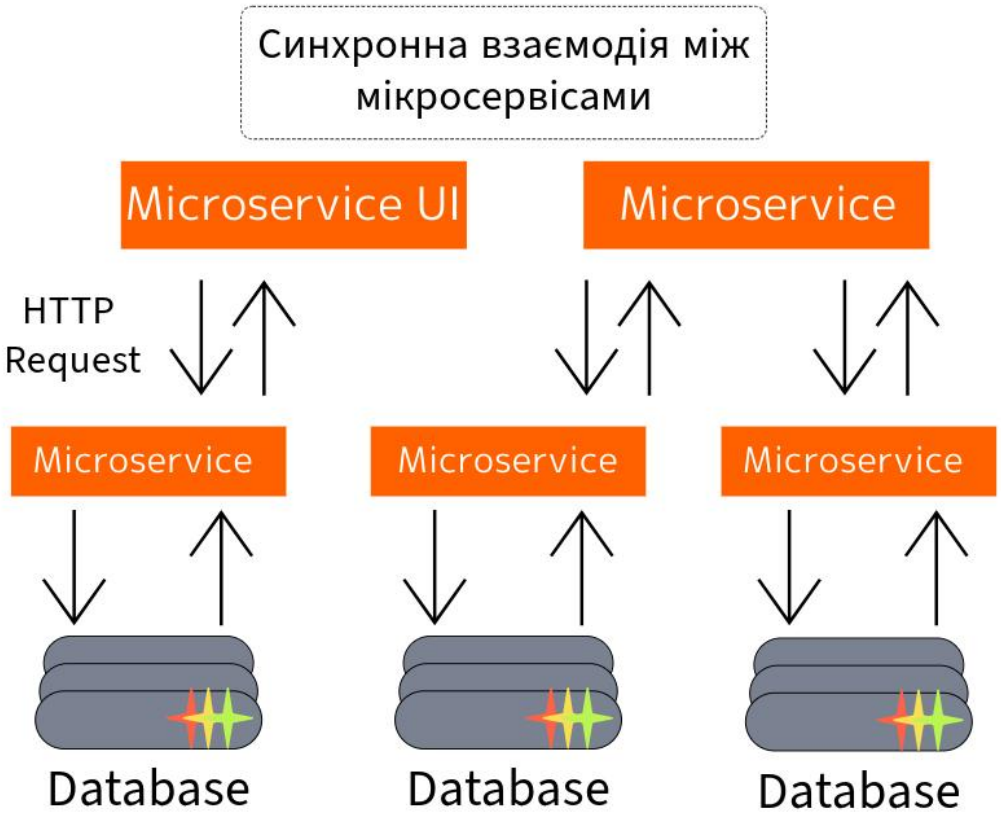


Рисунок 1.2 - Синхронна взаємодія між сервісами через HTTP запити

З іншого боку, асинхронна взаємодія виправляє недоліки синхроної взаємодії щодо можливості обробки великих обсягів даних із мінімальною затримкою, але асинхронна взаємодія є більш складною у реалізації та

моніторингу, до того ж потрібне додаткове налагодження управління брокерами подій. Робота асинхронної взаємодії виглядає наступним чином (рис 1.3) — сервіс-одержувач відправляє повідомлення (message) у чергу, не чекаючи на негайну відповідь, інші сервіси-відправники отримують повідомлення сервіса-одержувача і обробляють його.

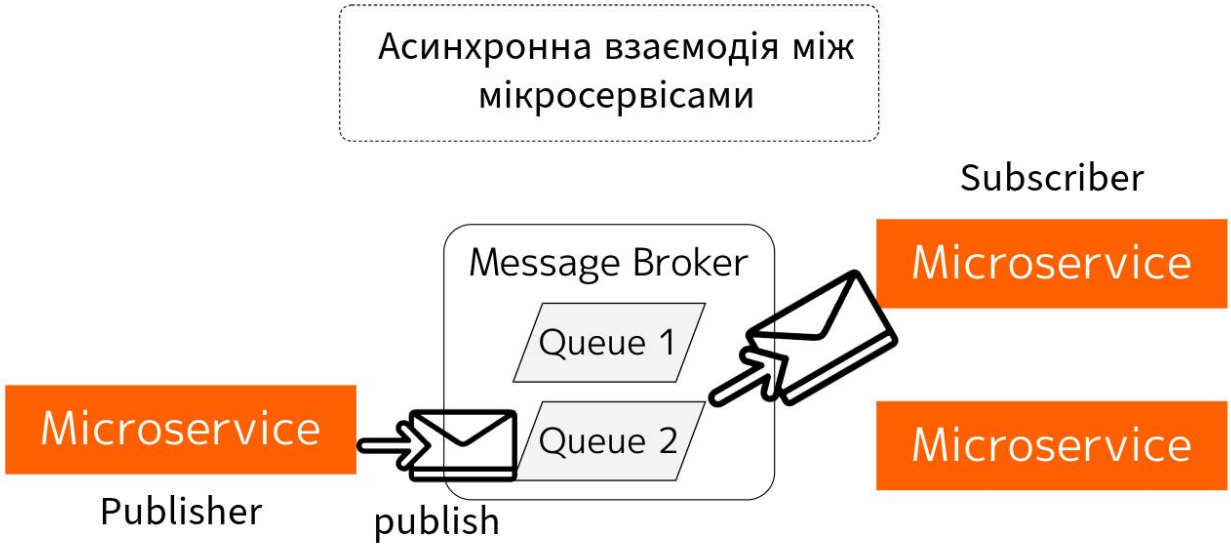


Рисунок 1.3 - Асинхронна взаємодія між сервісами через повідомлення і черги

На додаток, Service Mesh (рис 1.4) — це нова, стрімко набираюча популярність технологія управління комунікацією між мікросервісами. У Service Mesh взаємодією між сервісами займається один допоміжний мікросервіс, який уособлює в собі панель управління комунікацією, маршрутизацією запитів, балансуванням навантаження, безпекою, моніторингом та іншими аспектами взаємодії між сервісами. Реалізація Service Mesh забезпечується за допомогою проксі-серверів. Проксі-сервери розгортаються для всіх сервісів і розміщуються у капсулі. Перевагами Service Mesh є: зменшення складності, єдиний центр управління і масштабованість. Щодо недоліків Service Mesh можна відмітити: складність розгортання, перевантаження ресурсів через обробку трафіку через проксі-сервери і новизна підходу, що є викликом для багатьох розробників. Service Mesh комунікацію активно впроваджують, де є велика кількість мікросервісів, щоб забезпечити високий рівень безпеки, моніторингу та

масштабованості.

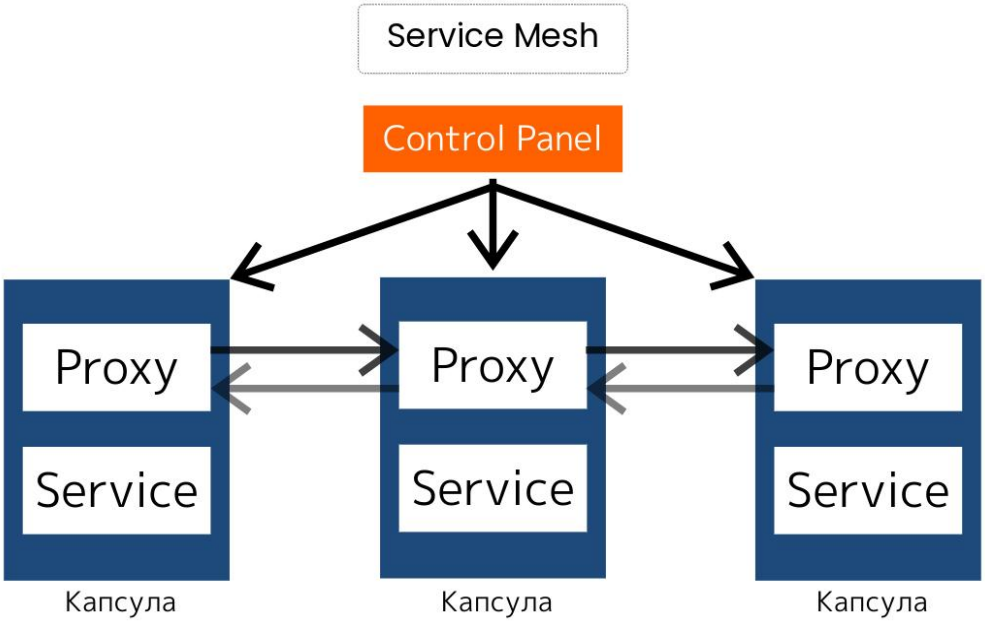


Рисунок 1.4 - Service Mesh взаємодія через Control Panel і проксі-сервери

Термін «мікросервіси» почав активно використовуватися та формалізуватися лише приблизно з 2011 року в межах спільноти розробників програмного забезпечення. Мікросервіс має низку значних переваг. По-перше, він забезпечує гнучкість у розробці та підтримці додатків, даючи змогу швидко вносити зміни та додавати нову функціональність. Крім того, завдяки декомпозиції додатка на незалежні сервіси, мікросервіси забезпечують поліпшену масштабованість і відмовостійкість[4]. Концепція мікросервісів не має єдиного винахідника, вона еволюціонувала з попередніх практик розробки, спрямованих на підвищення модульності, масштабованості та зручності обслуговування у великих системах. Наприклад, компанія Netflix була однією з перших, хто ще у 2009 році перейшла на архітектуру, схожу на мікросервісну. Термін “мікросервіси” було окреслено у 2014, і перехід на цю архітектуру став свідомим вибором сталих компаній. Схема мікросервісної архітектури показана на рис. 1.5.

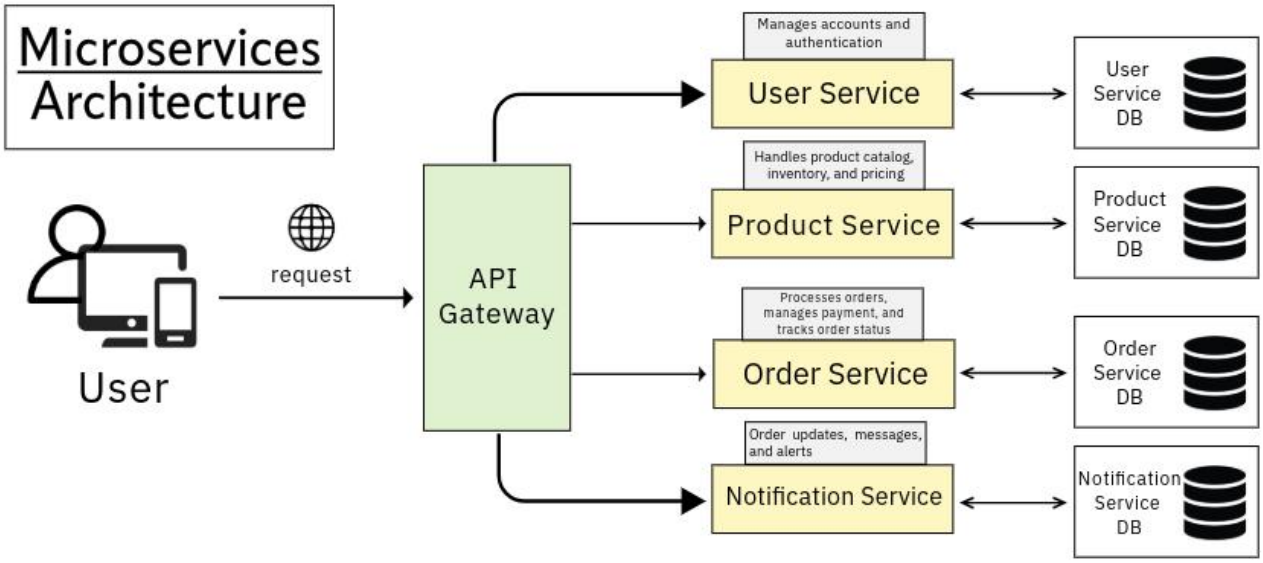


Рисунок 1.5 - Мікросервісна архітектура

Мікросервісна архітектура використовує незалежні одиниці, які обробляють дані у вигляді окремих сервісів. Кожен сервіс досягає цієї мети завдяки наявності власної логіки, бази даних і виконанню конкретних завдань. Таким чином, мікросервіси розділені на окремі модулі, а вся взаємодія між ними здійснюється через API-запити або системи обміну повідомленнями.

Перехід з моноліту на мікросервіси забезпечує:

- Гнучкість у виборі технологій для кожного сервісу. Наприклад, в онлайн-магазині більшість сервісів написані на Java, але сервіс користувачів реалізований на Python.
- Масштабованість, що довозвляє розширювати сервіс, який знаходиться у високому попиті, не призупиняючи інші сервіси, щоб розвантажити трафік задля його розширення.
- Мікросервіси стійкі до помилок, що запобігає згортанню всієї системи у разі виникнення проблем у одного із сервісів. Якщо один із сервісів раптово припиняє роботу, інші сервіси продовжують функціонувати. Це дозволяє швидко виявити та усунути проблему з непрацюючим сервісом.

Попри численні переваги, мікросервісна архітектура має певні недоліки:

1) Довший процес планування на етапі проєктування, оскільки необхідно чітко

визначити, як окремі сервіси будуть взаємодіяти один з одним.

2) Мікросервіси вимагають налаштування механізмів комунікації, моніторингу та забезпечення безпеки між модулями.
3) Необхідність у спеціалізованих інструментах і досвідченій команді для забезпечення стабільної роботи та обслуговування всіх сервісів.

1.3 Порівняльний аналіз монолітної та мікросервісної архітектур

У порівняльному аналізі визначено ключові критерії вибору між монолітною і мікросервісною архітектурами (таблиця 1.1):

Таблиця 1.1- Порівняння монолітної і мікросервісної архітектур

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
структура	Єдиний цілісний додаток	Незалежні сервіси
масштабованість	Вся система як єдине ціле	Масштабування окремих сервісів
гнучкість технологій	Обмежена	Висока
стійкість до збоїв	Вразлива до збоїв	Стійка
продуктивність	Висока для простих додатків	Оптимальна для великих систем
розробка	Швидка на початкових етапах	Більш комплексна
вартість	Низька	Висока

1. **Структура системи**

- Монолітна архітектура:

  Всі компоненти зосереджено в межах одного додатку, сервіси працюють разом і залежать один від одного, а передача даних відбувається через спільну базу даних.

- Мікросервісна архітектура:

  Система розділена на незалежні сервіси, кожен з яких має свою базу даних,

логіку та виконує специфічні завдання. Взаємодія між сервісами здійснюється через API запити, Message Broker або Service Mesh.

2. **Масштабованість**

- Монолітна архітектура:
  Масштабування потребує резервних копій усього додатка. Масштабування має вплив на весь код сервісів, недбалі і недоречні зміни у коді можуть негативно вплинути на роботу додатка або використання ресурсів у додатку.
- Мікросервісна архітектура:
  Масштабування відбувається окремо для кожного сервіса. Можна робити резервні копії сервісу і масштабувати його без впливу на інші сервіси.

3. **Гнучкість у виборі технологій**

- Монолітна архітектура:
  Має один технічний стек для усіх компонентів. Це забезпечує кращий контроль розробки, але за умови, що у команді розробників всі будуть знати однакову мову та базу даних, адже використання різних технологій може погіршити роботу у команді і призвести до проблем взаємодії у коді.
- Мікросервісна архітектура:
  Гнучкий стек гарно працює у великих командах. Через незалежність компонентів, сервіси можуть обслуговуватися командами з різними технічними стеками.

4. **Стійкість до збоїв**

- Монолітна архітектура:
  Збій в одному компоненті може зупинити роботу всієї системи, оскільки всі сервіси мють тісну взаємодію між собою.
- Мікросервісна архітектура:
  Збій в одиному сервісі не вплине на роботу інших сервісів, оскільки всі сервіси працюють окремо один від одного.

5. **Продуктивність**

- Монолітна архітектура:
  Компоненти належать до єдиного виконуваного процесу, мають одну базу даних. Все це забезпечує високу продуктивність у невеликих системах.
- Мікросервісна архітектура:
  Комунікація між сервісами відбувається через API-запити, що у свою чергу може створювати затримки через високу кількість запитів, але цей недолік компенсується масштабуванням і розподілом навантаження.

6. **Розробка та обслуговування**

- Монолітна архітектура:
  Простіша в розробці, але зі зростанням системи підтримка стає складнішою через взаємозалежність компонентів.
- Мікросервісна архітектура:
  Складна в розробці, вимагає значних зусиль в налаштуванні інфраструктури, моніторингу та тестування. Однак всі ці недоліки нівелюються спрощенням мастшабування та роботи у великих командах.

7. **Вартість**

- Монолітна архітектура:
  Низькі початкові витрати через просту інфраструктуру.
- Мікросервісна архітектура:
  Більші початкові витрати через складність інфраструктури.

1.4 Аналіз патернів проєктування мікросервісної архітектури

Мікросервісна архітектура має значний список патернів проєктування, які використовуються у специфічних ситуаціях для забезпечення ефективності масштабування, відмовостійкості та комунікації в системі. Патерни містять

перевірені і надійні рішення до розповсюджених проблем, що значно підвищує ефективність системи у цілому. Разом із тим, покращується орієнтування в проєкті. Коли проєкт реалізовано за правилами одного з патернів, обізнані розробники простіше розумітимуть роботу процесів у системі, і це позитивно впливатиме на подальшу розробку проєкта. Мікросервісна архітектура має десять часто вживаних патернів проєктування, такі як: API Gateway Pattern, Database per Service, Service Registry and Discovery, Circuit Breaker Pattern, Event Sourcing, Saga Pattern, CQRS (Command Query Responsibility Segregation), Bulkhead Pattern, Sidecar Pattern, Strangler Fig Pattern. Перелік патернів:

1. API Gateway (рис 1.6) — це підхід до розробки в мікросервісній архітектурі, де є єдина централізовна точка входу. API Gateway діє як посередник між користувачем і сервісом. Він обробляє вхідні запити користувачів та маршрутизує їх до відповідних мікросервісів. Даний патерн підійде для систем, де треба знизити навантаження на сервіси, оскільки API Gateway бере на себе обробку запитів, таких як автентифікація, кешування, обмеження швидкості, та обробку помилок. Імплементація API Gateway значно знижує навантаження на мікросервіси, що дозваляє їм фокусуватися виключно на виконанні бізнес-логіки. Покращує безпеку системи: API Gateway застосовує контроль доступу, захищаючи основні мікросервіси. До того ж, централізація API Gateway робить легким моніторинг та збір аналітики, що дійсно важливо у великих корпоративних системах. Порівняння API Gateway зображено на таблиці 1.2.

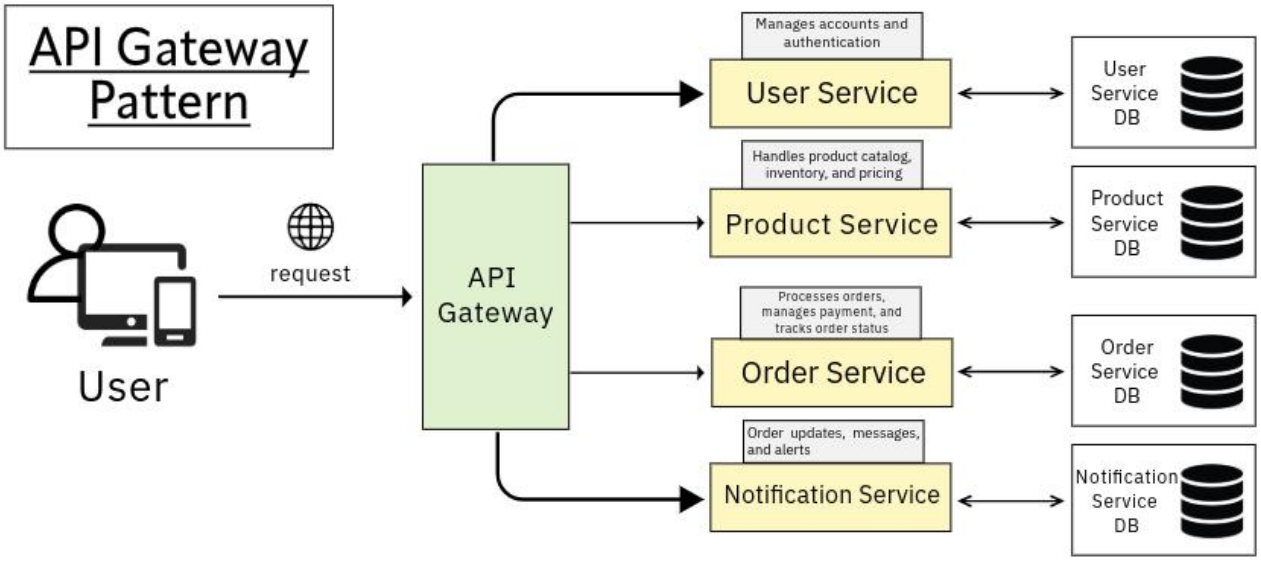


Рисунок 1.6 - API Gateway Pattern

Таблиця 1.2 - Порівняння API Gateway

Переваги	**Недоліки**
Спрощує взаємодію з клієнтами	Єдина точка відмови у разі несправності
Централізоване управління	Може збільшувати час обробки запитів через затримки без належної оптимізації
Підвищує безпеку	Накладає додаткові витрати на продуктивність

2. Database per Service (рис 1.7) — патерн, за яким кожен мікросервіс має свою особисту базу даних, де доступ встановлюється за допомогою власних API. Мікросервіси, що мають власну базу даних, обмежені в доступі до сусідскіх баз даних інших мікросервісів.

Порівняння Database per Service зображено на таблиці 1.3.

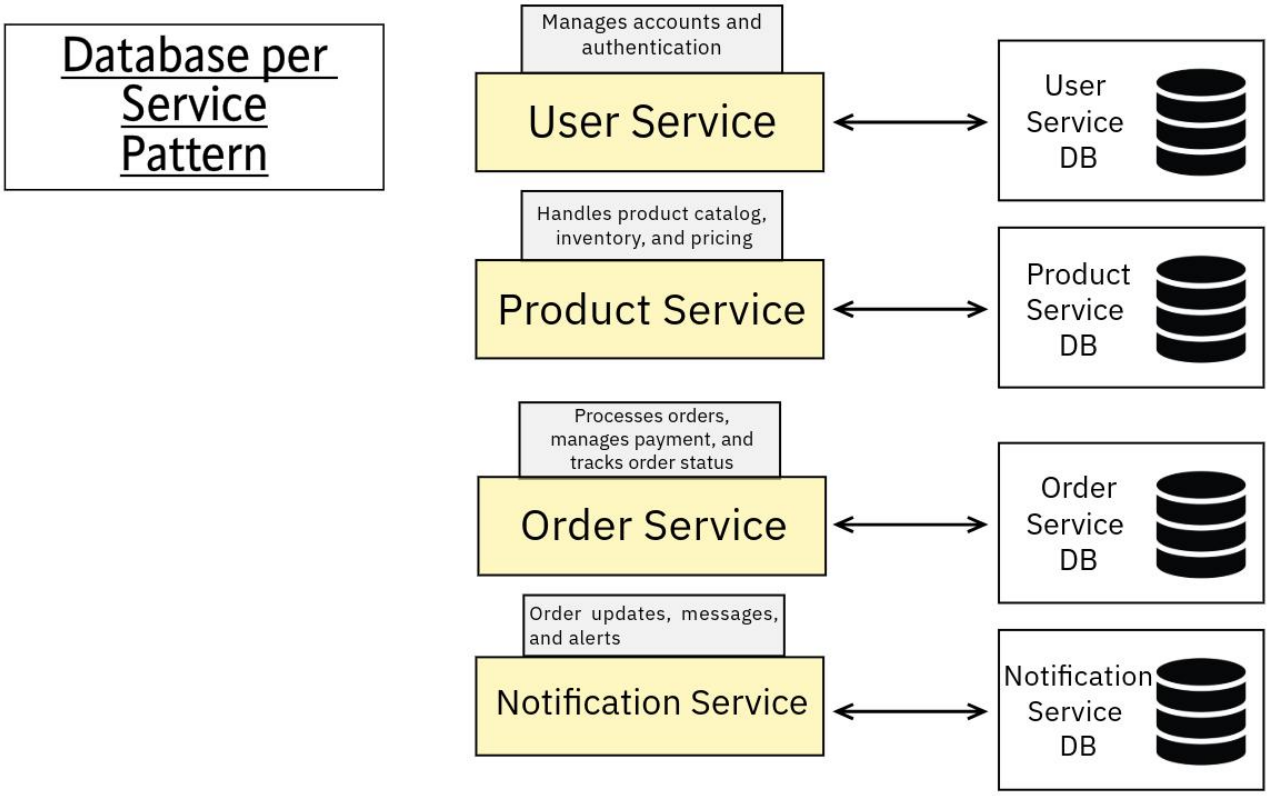


Рисунок 1.7 - Database per Service Pattern

Таблиця 1.3 - Порівняння Database per Service

Переваги	**Недоліки**
Знижує зв’язаність між сервісами	Складність підтримання консистентності даних
Гнучкість у виборі технологій	Складність управління кількома базами даних
Незалежність у масштабуванні	Ускладнені міжсервісні запити

3. Backend for Frontend (BFF) (рис 1.8) — це підхід до розробки в мікросервісній архітектурі, де кожен користувацький інтерфейс має свою власну backend частину. Наприклад, даний патерн проєктування дозволяє розгортати застосунок у формі: веб додатку, мобільного додатку або прикладної програми із забезпеченням власної backend частини для кожного компонента. Кожен BFF проєктується, коли

є спеціальні потреби у впровадженні додаткових backend функцій у frontend: обробка збору даних, масштабування та комунікація з основними мікросервісами або API. Патерн BFF має застосування у системах, де присутня багатоплатформність, що має декілька frontend застосунків, які потребують backend рішень до їх функціональних вимог. Порівняння Backend for Frontend зображено на таблиці 1.4.

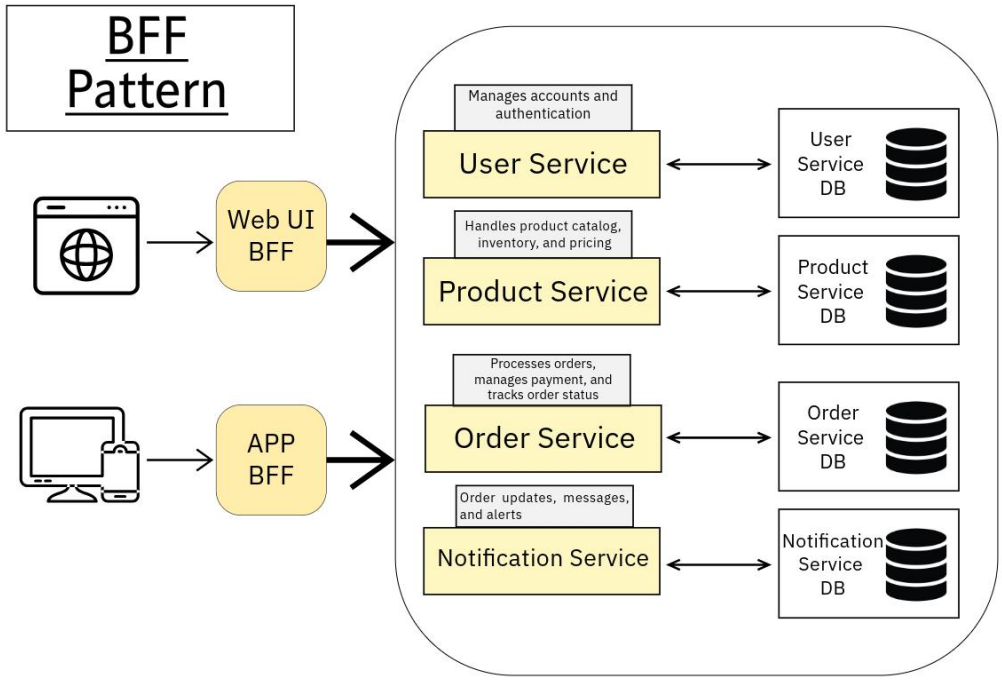


Рисунок 1.8 - Backend for Frontend (BFF) Pattern

Таблиця 1.4 - Порівняння Backend for Frontend

Переваги	**Недоліки**
Оптимізовано для конкретного інтерфейсу	Складніше підтримувати та розробляти
Спрощує роботу frontend	Можливе дублювання коду
Дозволяє незалежний розвиток	Підтримка консистентності може бути складною

4. Command Query Responsibility Segregation (CQRS) (рис 1.9) — це підхід до розробки в мікросервісній архітектурі, де читання (queries) і запис (commands) даних розділяється на окремі моделі — Command Model і Query Model. У звичайних системах єдиний компонент обробляє одночасно і читання, і записи. Це ускладнює код і знижує продуктивність у великих масштабах. Отже, використовуючи CQRS, розділення обов'язків читання та запису усуває недоліки уніфікованої системи. Імплементація CQRS покращує продуктивність, масштабованість і читання коду. Цей патерн надає можливість адаптації кожної моделі до її конкретної функції. Query Model оптимізує отримання данних, використовучи метод денормалізації бази даних або кешування. Command Model оптимізується для опанування складної бізнес логіки і змінів стану. Порівняння Command Query Responsibility Segregation зображено на таблиці 1.5.

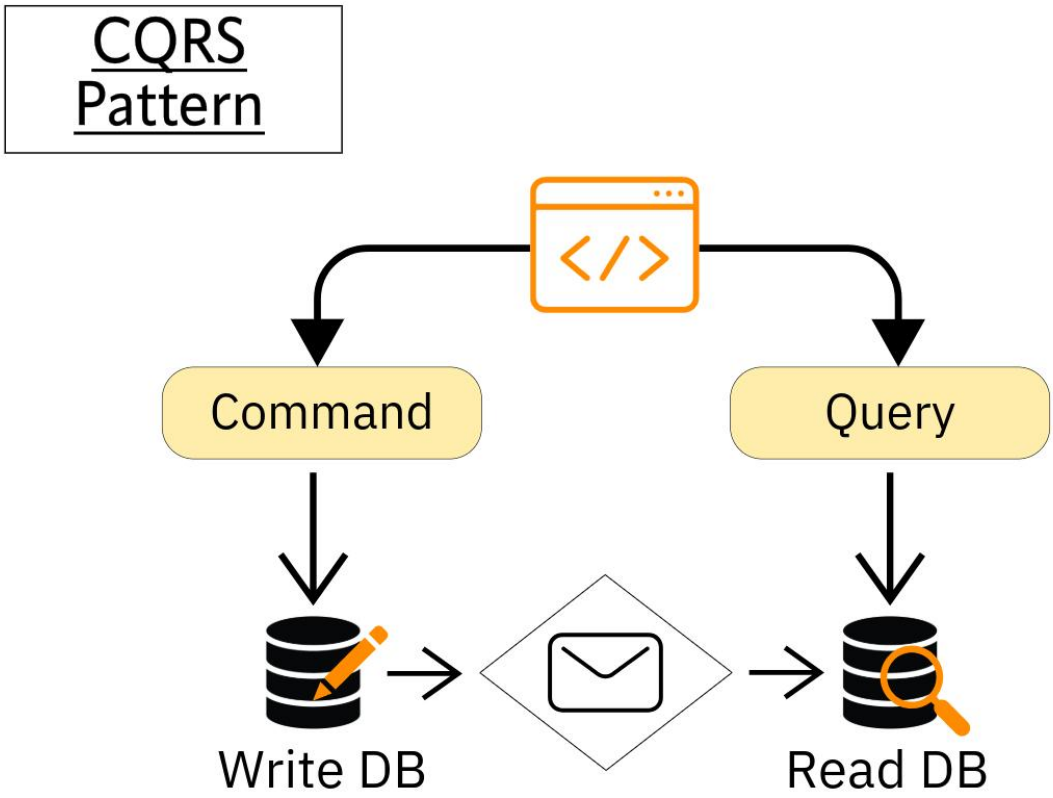


Рисунок 1.9 - Command Query Responsibility Segregation (CQRS) Pattern

Таблиця 1.5 - Порівняння Command Query Responsibility Segregation

Переваги	**Недоліки**	**Переваги**
Підвищує продуктивність запитів	Потребує окремого управління моделями	Підвищує продуктивність запитів
Незалежне масштабування операцій	Важко підтримувати синхронізацію даних	Незалежне масштабування операцій

5. Event Sourcing (рис 1.10) — це підхід до розробки в мікросервісній архітектурі, де зміни станів у додатку зберігаються у вигляді послідновності подій, а не шляхом прямого оновлення бази даних. Події зберігаються в сховищі подій. Сховище подій містить всю історію подій, що відбулися в системі. Даний підхід допомагає уникати повторення подій, які вже відбулися. Кожна подія може містити унікальний ідентифікатор, що дозволяє уникати дублювання, таким чином події обробляються лише один раз. Поточний стан об’єкта створюється шляхом відтворення журналу подій. Для оптимізації процесу використовуються знімки стану (snapshots), які тимчасово зберігають стан, зменшуючи кількість подій, які потрібно відтворювати. Event Sourcing підтримує незмінність: події залишаються незмінними, таким чином їх неможливо змінити або видалити, що забезпечує точний журнал змін. Порівняння Event Sourcing зображено на таблиці 1.6.

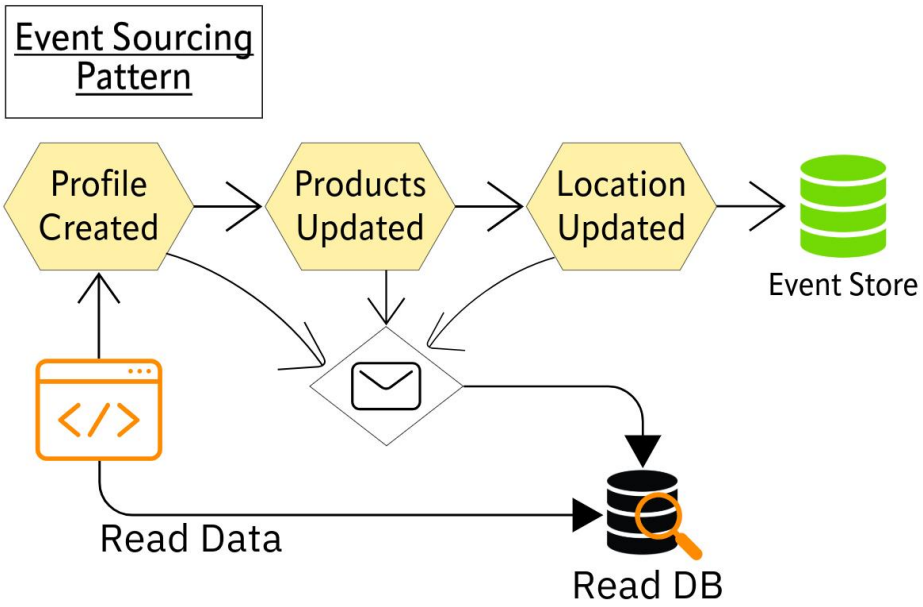


Рисунок 1.10 - Event Sourcing Pattern

Таблиця 1.6 - Порівняння Command Query Responsibility Segregation

Переваги	**Недоліки**	**Переваги**
Дає повну історію змін даних	Потребує великого обсягу пам’яті	Дає повну історію змін даних
Легко масштабується для записів	Ускладнює запити до даних	Легко масштабується для записів

5. Saga Pattern (рис 1.11) — це підхід до розробки в мікросервісній архітектурі, де існує розподілені системи. Saga використовується в розподілених системах для керування бізнес-транзакціями в мікросервісах і базах даних, розділяючи довготривалу транзакцію на послідовність локальних операцій. Якщо вона не порушує бізнес-правила, тоді Saga виконує серію компенсуючих транзакцій, які скасовують зміни, внесені попередніми локальними транзакціями [5].Взаємодія з транзакціями оновлює стан системи, а у випадку помилки, спрацьовують компенсуючі транзакції для скасування змін (рис 1.12.) Saga не потребує блокування ресурсів для забезпечення цілісності даних і запобігання конфліктам під час транзакцій, так як використовує остаточну узгодженість

(eventual cosistency). Остаточна узгодженість означає, що після завершення оновлення транзакції, усі копії даних у системі стануть однаковими через певний час, зміни в даних поступово синхронізуються між послідовними локальними операціями і на це треба час. Порівняння Saga Pattern зображено на таблиці 1.7.

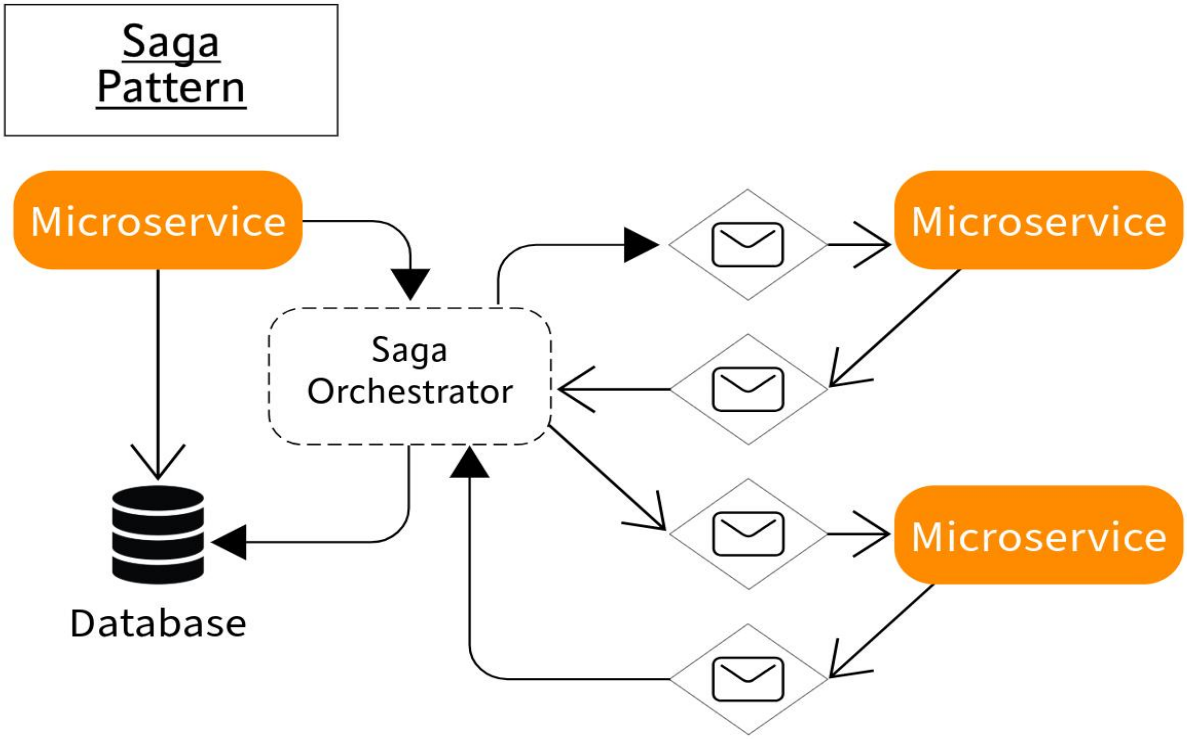


Рисунок 1.11 - Saga Pattern

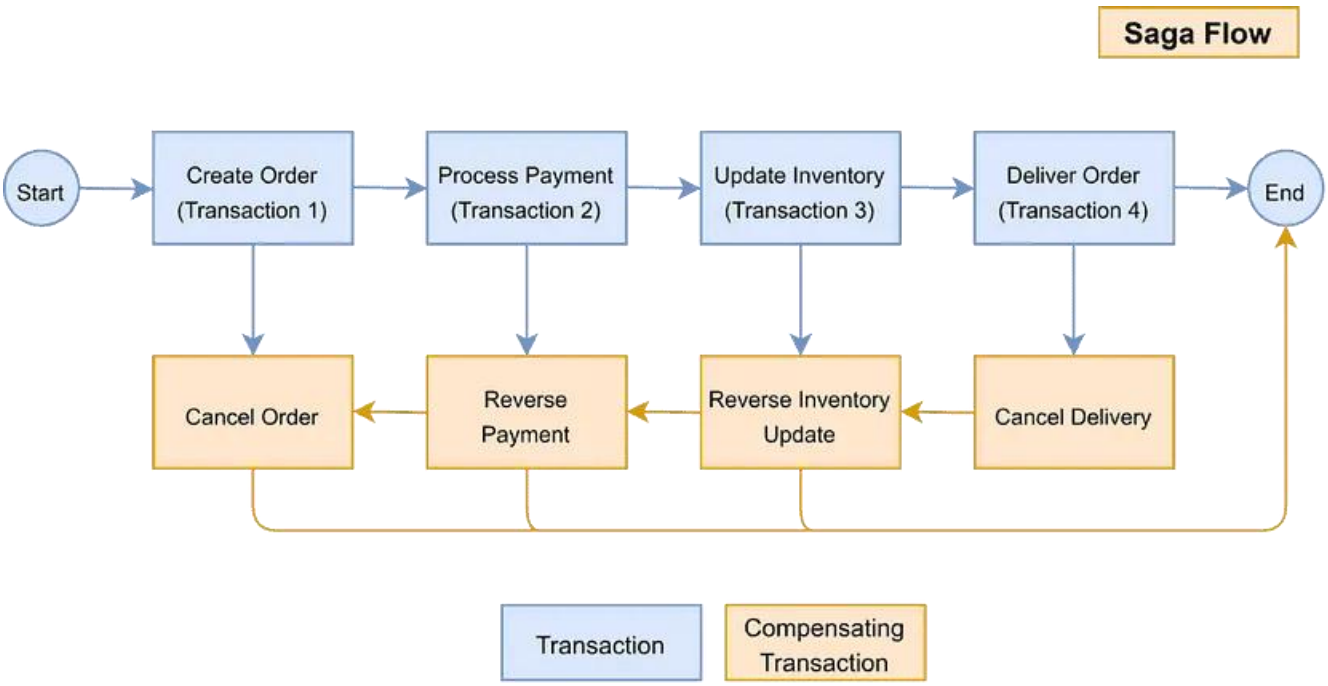


Рисунок 1.12 - Схема алгоритму компенсуючих транзакцій Saga Pattern[6]

Таблиця 1.7 - Порівняння Saga Pattern

Переваги	**Недоліки**
Забезпечує остаточну узгодженість	Складно реалізувати та відлагодити
Добре працює з розподіленими транзакціями	Відсутній автоматичний відкат транзакцій

Управління у Saga Pattern досягається за допомогою двох способів: оркестрації і хореографії.

Використання способу оркестрації (рис 1.13) означає, що в системі управління розподіленими транзакціями усі процеси виконання і послідовності дій контролюються за допомогою центрального координатора — оркестратора. Використання оркестатора вводить централізоване управління, де керування транзакціями зосереджені в одному місці, що полегшує взаємодію із системою. Центральний оркерстратор контролює процеси в Saga, повідомляючі кожному мікросервісу, коли треба виконувати транзакцію, супроводжуючі дані дії протягом усього потокового процесу. Порівняння способу оркестрації зображено на таблиці 1.8.

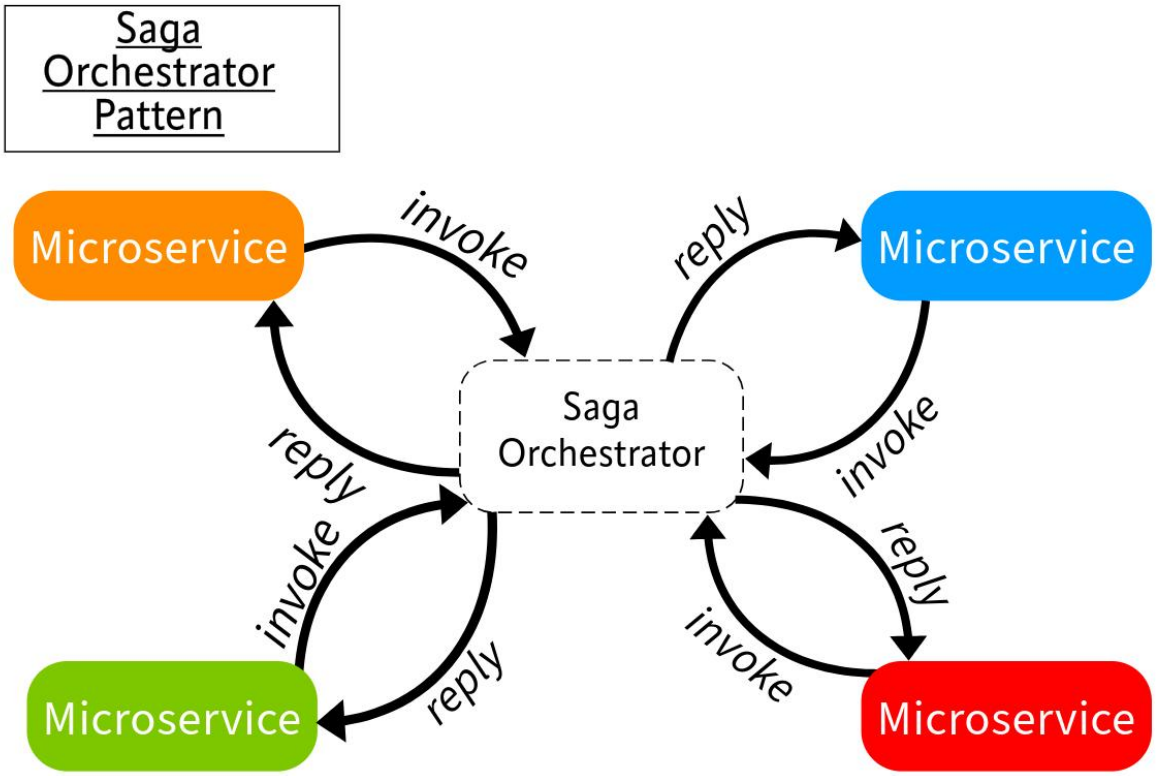


Рисунок 1.13. - Saga Orchestrator Pattern

Таблиця 1.8 - Порівняння Orchestration Pattern

Переваги	**Недоліки**
Централізований контроль	Єдина точка відмови
Просте управління послідовністю дій	Збільшення складності
Прозорість	Затримка виконання
Зручний моніторинг	Менша гнучкість
Просте відновлення	Навантаження на координатор

Використання способу хореографії (рис 1.14) означає, що в системі управління розподіленими транзакціями усі процеси виконання і послідовності дій виконуються автономно, реагуючи на події без централізованого контролю. У хореографії немає єдиного координатора, який б визначав порядок дій, тому сервіси повинні реагувати на події орієнтуючись на генерацію подій іншими

сервісами. Порівняння способу хореографії зображено на таблиці 1.9.

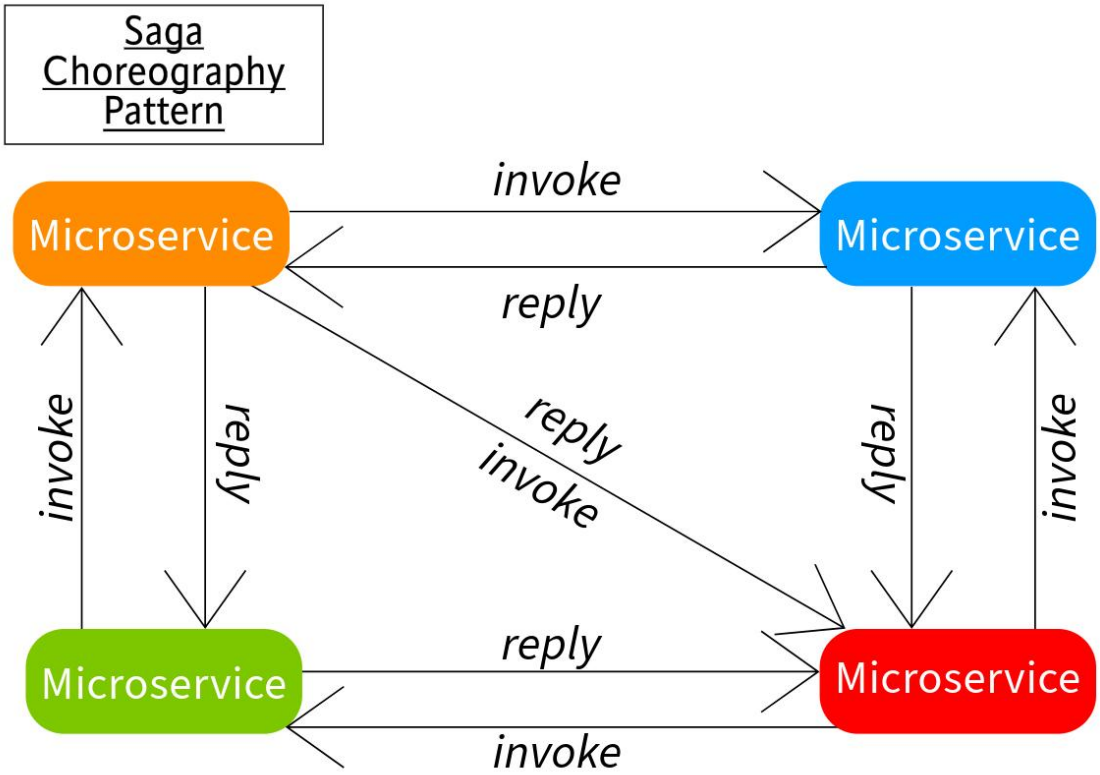


Рисунок 1.14 - Saga Choreography Pattern

Таблиця 1.9 - Порівняння Choreography Pattern

Переваги	**Недоліки**
Відсутність єдиної точки відмови	Складність управління
Краща масштабованість	Складна відлагодження
Гнучкість і автономність	Менше прозорості
Децентралізація	Складність координації
Швидше виконання	Проблеми узгодженості

5. Sidecar Pattern (рис 1.15) — це підхід до розробки в мікросервісній архітектурі, де розгортається допоміжний сервіс (sidecar) для основного сервісу в одному середовищі. Допоміжний сервіс спрощує та покращує функціональність основного сервіса, беручи на себе відповідальність над процесами: авторизації, моніторингу і безпеки. Sidecar розвиває гілки модульності та масштабованості,

бере на себе другорядну роботу, дозволяючи основному сервісу виконувати профільну роботу без відволікань. Порівняння Sidecar Pattern зображено на таблиці 1.10.

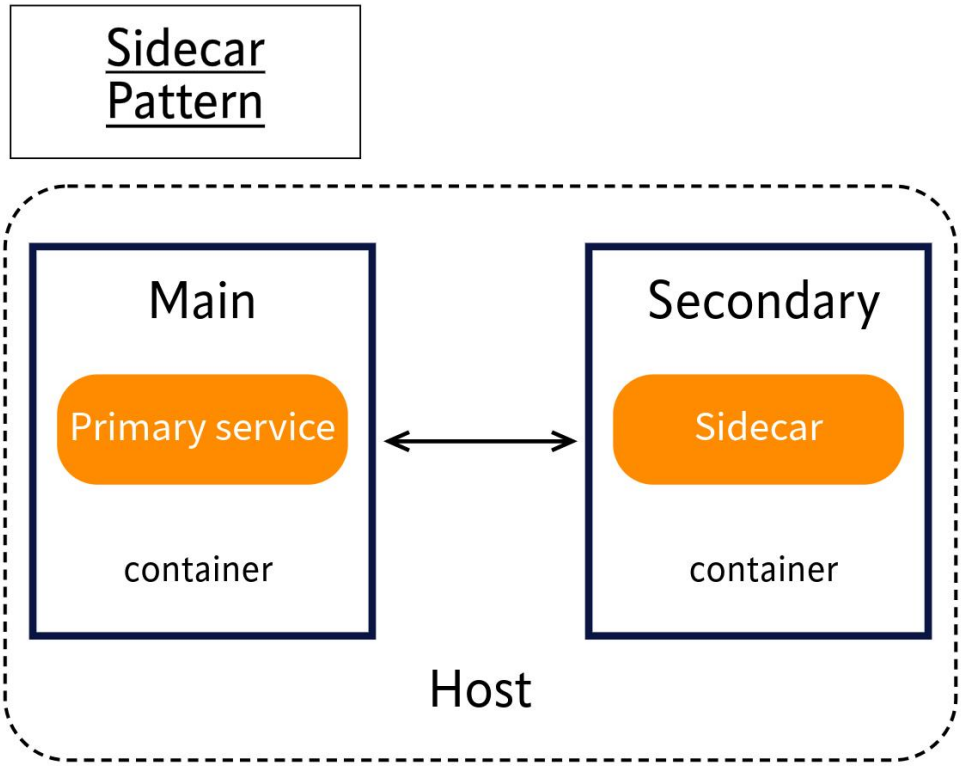


Рисунок 1.15 - Sidecar Pattern

Таблиця 1.10 - Порівняння Sidecar Pattern

Переваги	**Недоліки**
Забезпечує остаточну узгодженість	Складно реалізувати та відлагодити
Добре працює з розподіленими транзакціями	Відсутній автоматичний відкат транзакцій
Підвищує стійкість системи	Відсутня ізоляція між паралельними транзакціями

5. Circuit Breaker Pattern (рис 1.16) — це підхід до розробки в мікросервісній архітектурі, де стійкість і надійність в розподілених системах забезпечується автоматичним вимикачем, що запобігає каскадним збоям, зупиняючи запити

до несправного сервісу, і надає час для його відновлення. Circuit Breaker працює у трьох станах: замкнутий, відкритий і напіввідкритий. У замкнутому стані запити потрапляють до сервісу, що дозволяє відстежити відповіді або збої. Якщо збої перевищують встановлений поріг, то автоматичний вимикач переходить у відкритий стан. У відкритому стані відубаваються превентивні дії щодо переривання запитів у відказавшому сервісі, під час перерви сервіс має визначиний час на відновлення своєї роботи. Коли настає етап відновлення, свою роботу починає напіввідкритий стан, який пропускає обмежену кількість запитів для перевірки, щоб переконатися у справності відновленого сервісу. Якщо запит був успішними, то автоматичий вимикач переходить у відкритий стан, інакше, якщо запит був невдалим, то перемикач переходить до замкнутого стану. Порівняння Sidecar Pattern зображено на таблиці 1.11. Іллюстрація роботи станів автоматичного вимикача зображено на рисунку 1.17.

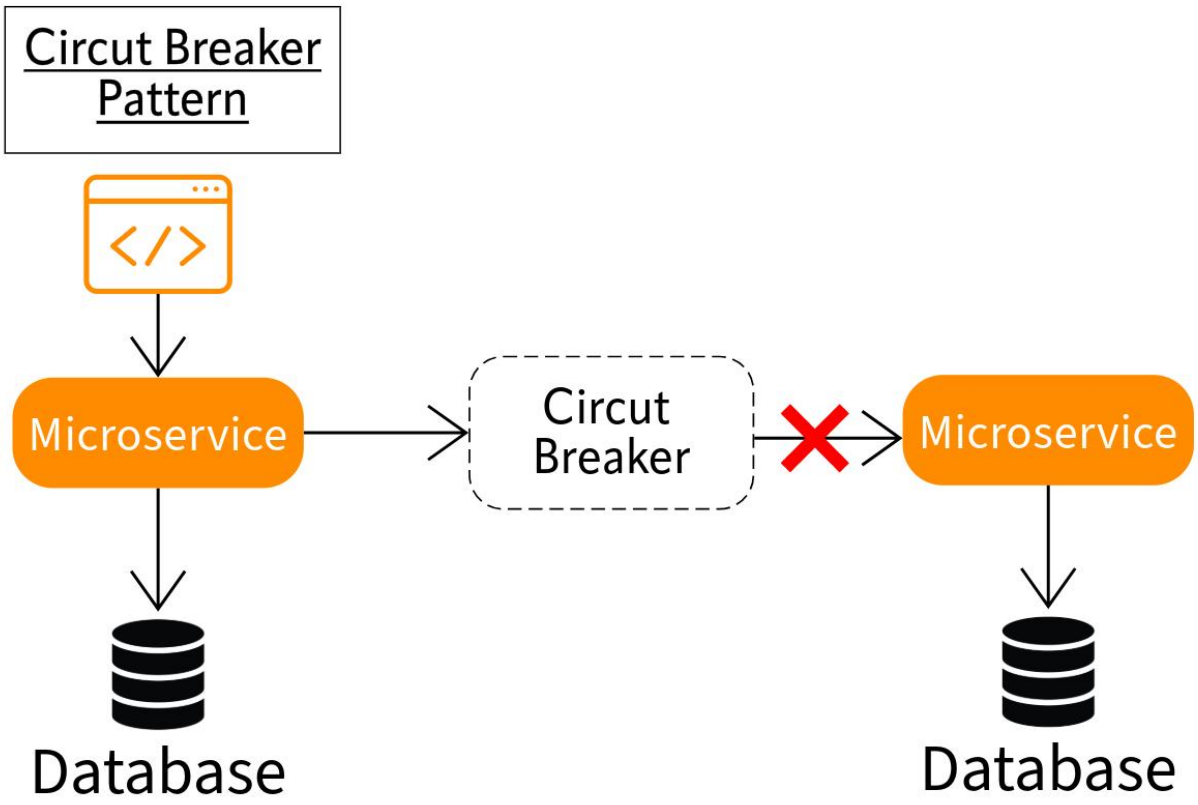


Рисунок 1.16 - Sidecar Pattern

Таблиця 1.11 - Порівняння Circuit Breaker Pattern

Переваги	Недоліки
Захист від каскадних збоїв	Складність конфігурації
Покращена стійкість	Можливість помилкового спрацьовування
Ефективне відновлення	Накладає додаткові витрати на продуктивність

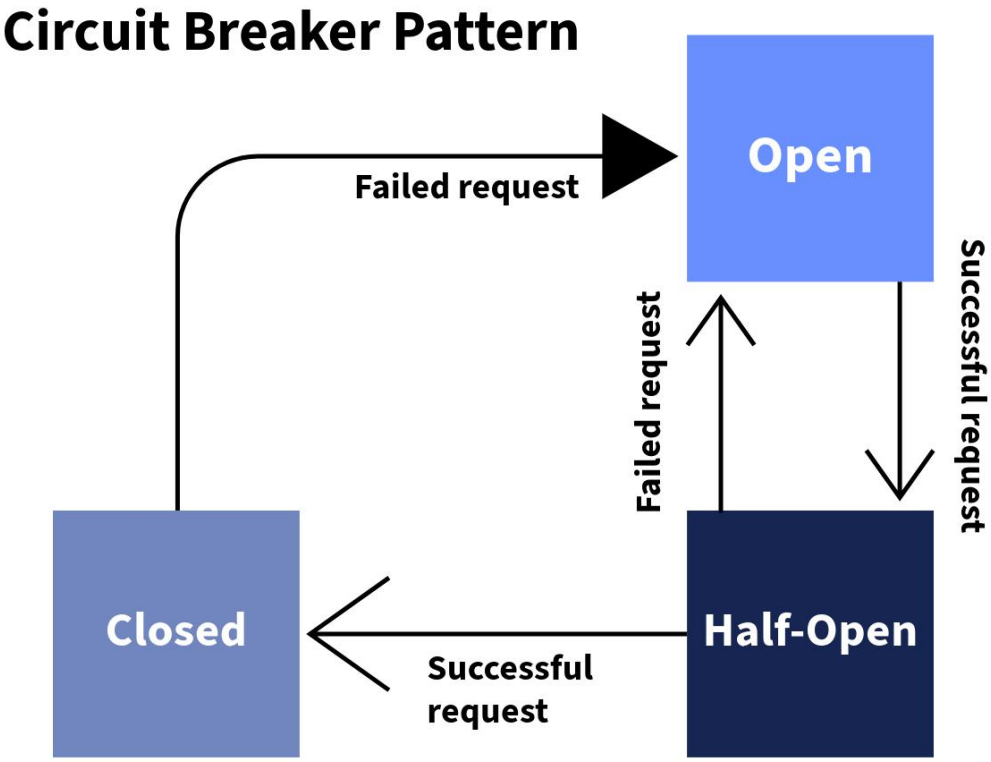


Рисунок 1.17 - Стани автоматичного вимикача

5. Anti-Corruption Layer (ACL) (рис 1.18) — це підхід до розробки в мікросервісній архітектурі, де ізолюються різні частини системи з метою запобігання пошкодження даних або бізнес-логіки через корупцію (перенесення непотрібної чи помилкової логіки, яка може вплинути на правильність роботи системи). Даний патерн діє як бар'єр або посередник між

двома системами, перешкоджає прямому доступу до зовнішніх систем або застарілих інетерфейсів, які можуть передавати некоректні або непотрібні дані. Anti-Corruption Layer допомагає при інтеграції нових систем до доменої моделі без ризиків несподіваних помилок або неправильної логіки. Порівняння Anti-Corruption Layer Pattern зображено на таблиці 1.12.

Таблиця 1.12 Порівняння Anti-Corruption Layer Pattern

Переваги	**Недоліки**
Ізоляція систем	Додаткова складність
Чистота логіки	Витрати на розробку

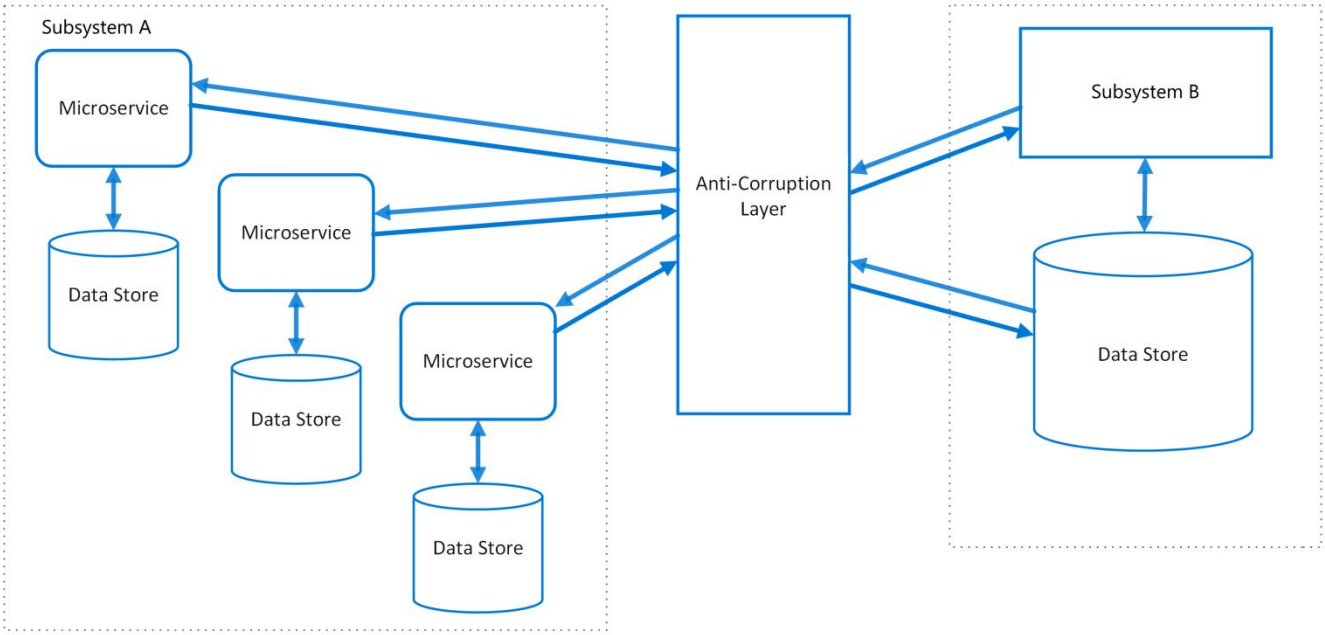


Рисунок 1.18 - Anti-Corruption Layer Pattern[7]

5. Aggregator Pattern (рис 1.19) — це підхід до розробки в мікросервісній архітектурі, де консолідація даних або відповіді із кількох джерел зібрано у єдиний уніфікований результат для повернення його користовачу. Агрегатор відповідає за збір даних в джерелах сервісів, отримує, об’єднує та обробляє. Порівняння Aggregator Pattern зображено на таблиці 1.13.

Таблиця 1.13 - Порівняння Anti-Corruption Layer Pattern

Переваги	Недоліки
Зменшення кількості запитів	Точка відмови
Спрощення клієнтської логіки	Затримки у відповіді
Інкапсуляція складності	Складність управління

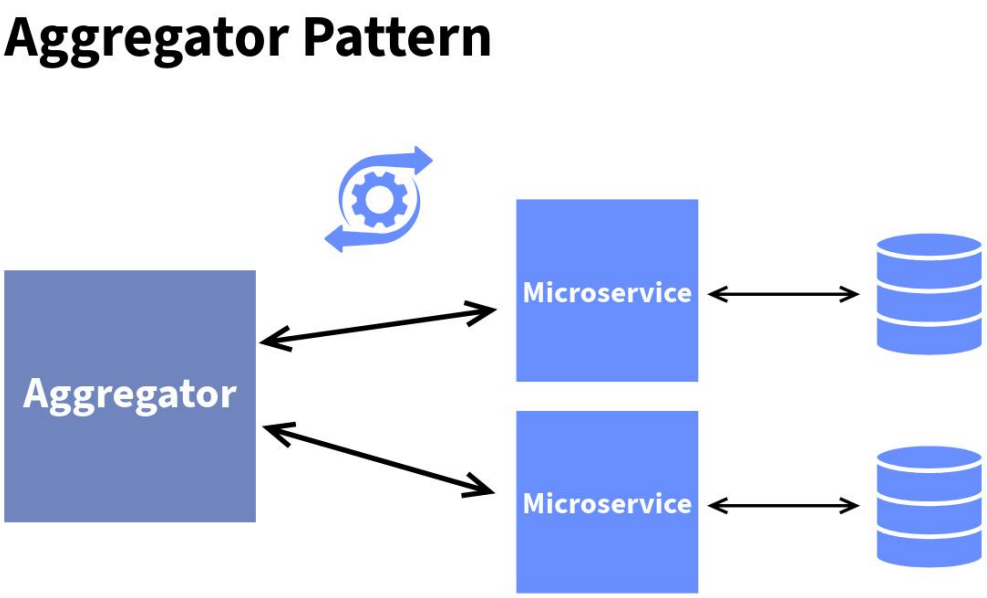


Рисунок 1.19 - Aggregator Pattern

5. Strangler Fig Pattern (рис 1.20) — це підхід до розробки в мікросервісній архітектурі, де реалізується стратегія поступової міграції з монолітної архітектури до мікросервісної архітектури. У перекладі з англійської на українську, назва "strangler fig" означає "фікус-душитель". Назва натхненна поведінкою дерева фікуса-душителя (рис. 1.21), яке росте навколо дерева-господаря і поступово його замінює (рис 1.22), і це є прямою аналогією заміни мікросервісами монолітної основи, розбиттям в набір дрібніших, слабко зв’язаних сервісів, які у повній реалізації всіх функцій моноліта повністю заміняють його.

Рисунок 1.21 - Фікус-душитель на стовбурі сизигіума, Новий Південний Уельс, Австралія[8]

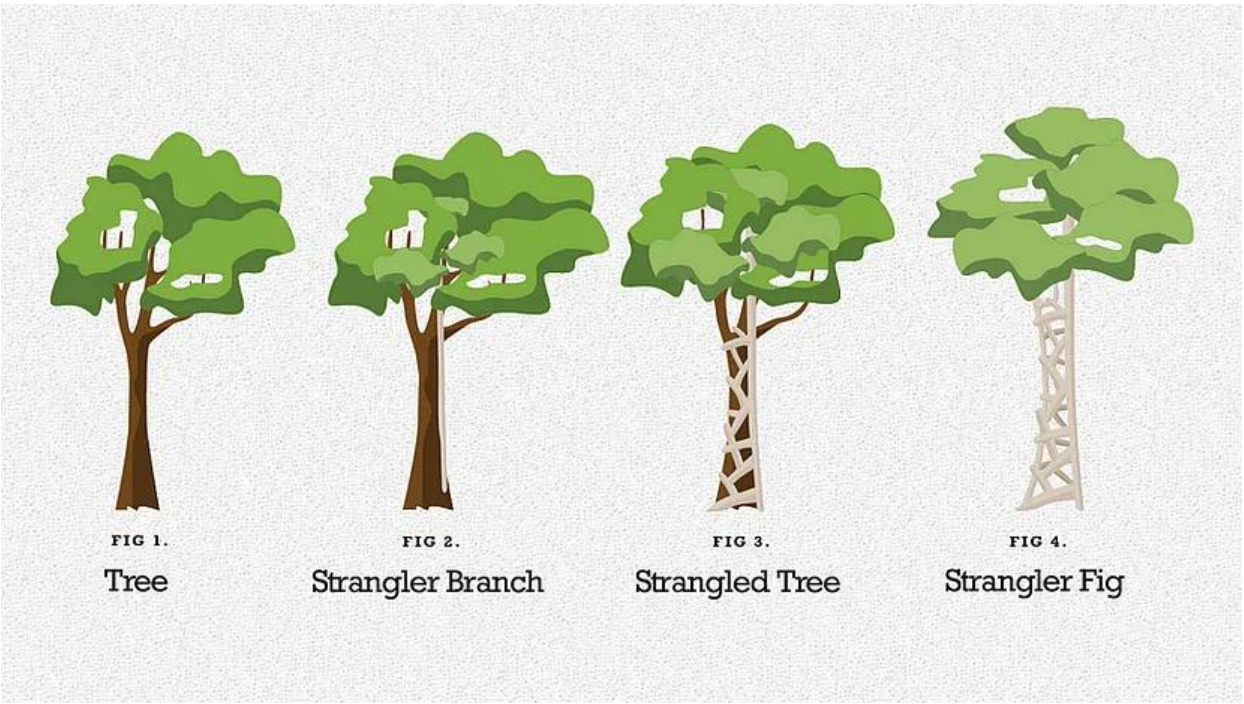


Рисунок 1.22 - Етапи росту Фікуса-душителя

Отже, досить старі кодові бази під впливом часу починають містити застарілий код, який не адаптований до сучасних вимог. Код вважається застарілим, коли його функціонал реалізовано на більш ефективному, простому і

лаконічному новому стандарті кодування. Таким чином, забезпечення гнучкості, масштабованості та підтримуваності системи зрештою завершується повним переписанням програми. Переписання програми — це безумовно стресовий процес, під час якого помилковий підхід до міграції систем може спричинити порушення логіки взаємодії між компонентами, що призведе до несподіваних помилок та ускладнень. На щастя, використовуючи патерн Strangler Fig, розробники розбивають програму на окремі частини, з'ясовують, які компоненти краще об'єднати в один, а які розділити. Цей процес схожий на побудову будинку, коли поступово кладеться цеглина за цеглиною, а в контексті переведення програмних архітектур — поступово сервіс за сервісом, мігруючи їх до нової архітектури. Замисел патерну Strangler Fig полягає в тому, щоб розробники не поспішали з повним переписом монолітної архітектури на мікросервісну. Перехід має бути поступовим, і кожен сервіс повинен ретельно аналізуватися, щоб підвищити ефективність та оптимізацію в новій архітектурі. Іллюстрований процес переходу зі старої кодової бази на нову зображено на рисунку 1.23. Порівняння Strangler Fig Pattern зображено на таблиці 1.14.

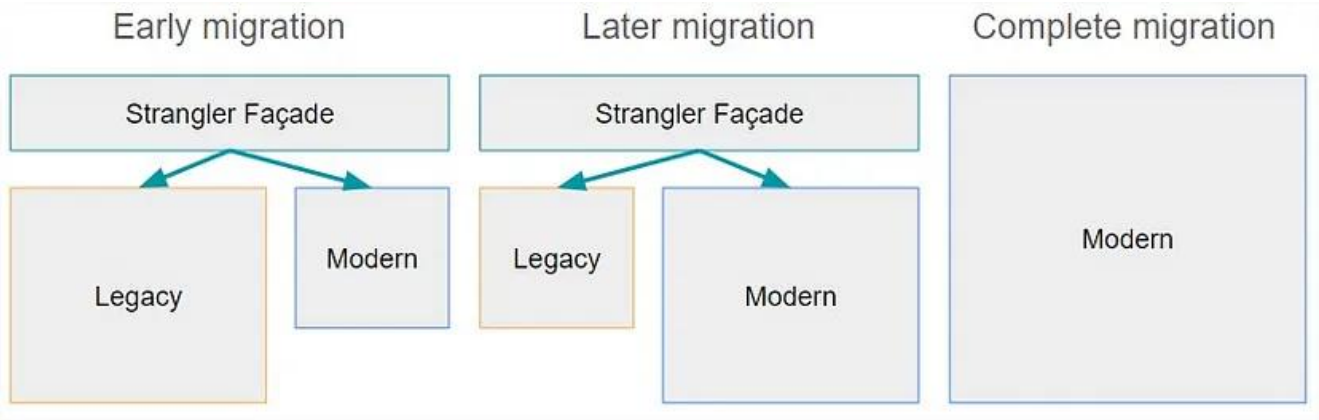


Рисунок 1.23 - Етапи мігрування старої кодової бази у нову[9]

Таблиця 1.14 - Порівняння Strangler Fig Pattern

Переваги	**Недоліки**
Зменшення кількості запитів	Точка відмови
Спрощення клієнтської логіки	Затримки у відповіді

Міграцію можна розділити на 4 технічні етапи:

1) Визначення функціональності або блоку коду, який потребує оновлення, та розгортання його як єдиного окремого елемента.
2) Забезпечення фасаду (проксі-шару) між створеним блоком і основною кодовою базою. Фасад (facade) — це проміжний шар (проксі-шар), який виступає як посередник між клієнтами (користувачами або іншими компонентами системи) і основною кодовою базою. Його завдання полягає у спрямуванні запитів або на монолітний застосунок, або на нові мікросервіси.
3) Реалізація нової функціональності в новій системі шляхом зміни посилання між фасадом (проксі-шаром) і старим кодом на нове посилання, яке об’єднує фасад із новоствореним мікросервісом.
4) Інкрементування процесу доти, доки блок коду у фасаді (проксі-шарі) не буде повністю інтегровано до мікросервісу. Після успішної міграції код із моноліту видаляється.

Покрокові етапи перетворення моноліту на мікросервіс зображено на рисунках 1.24 - 1.30.

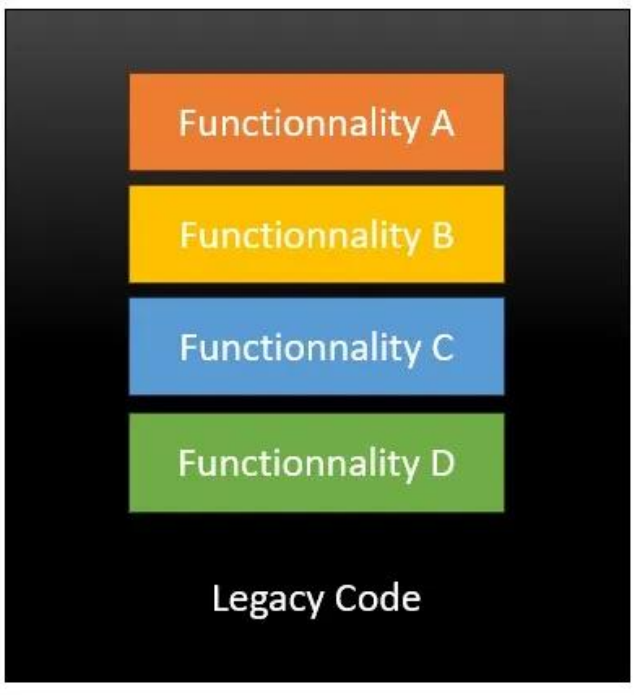


Рисунок 1.24 - Початкова кодова база моноліту[9]

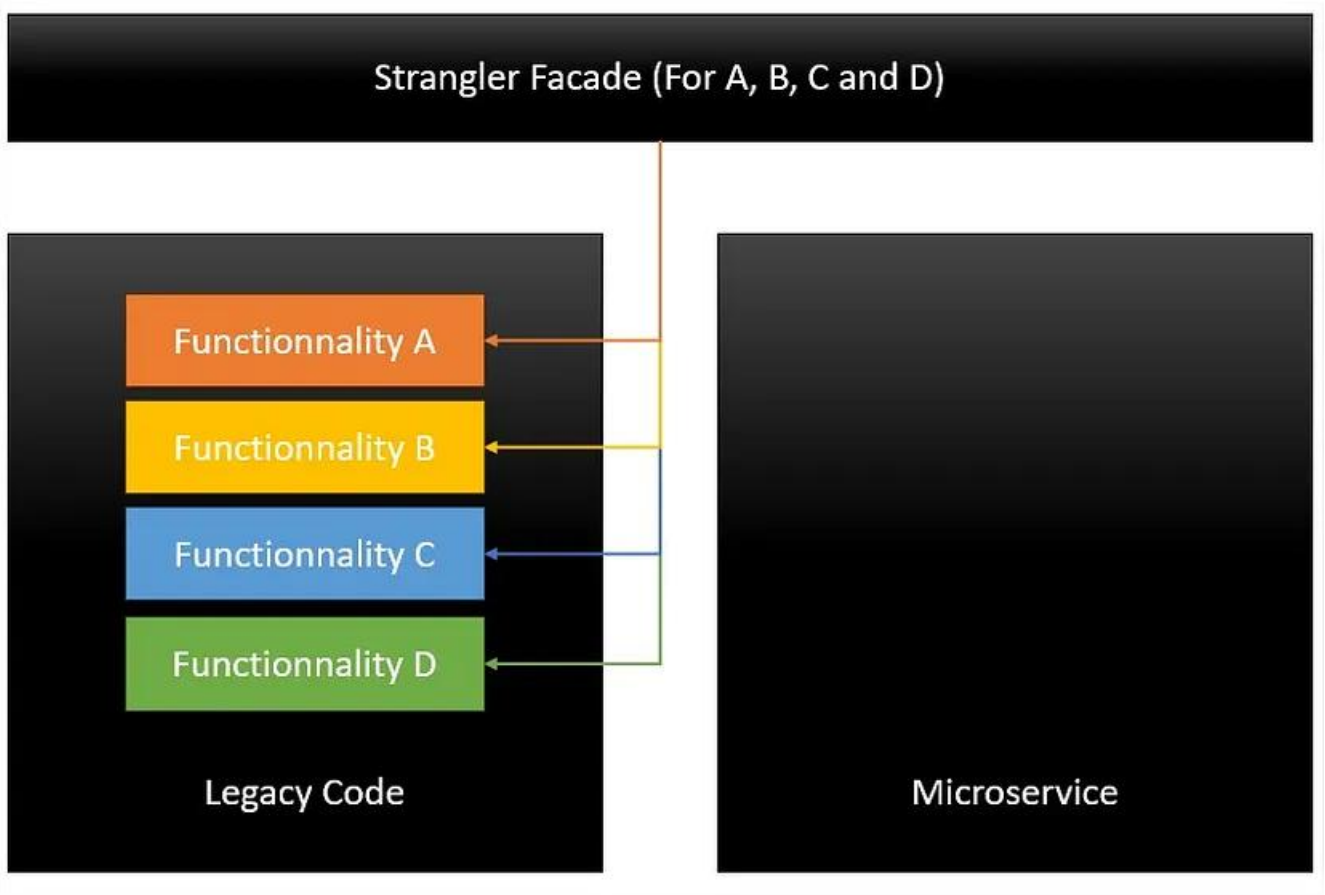


Рисунок 1.25 - Створення фасаду для мігрування[9]

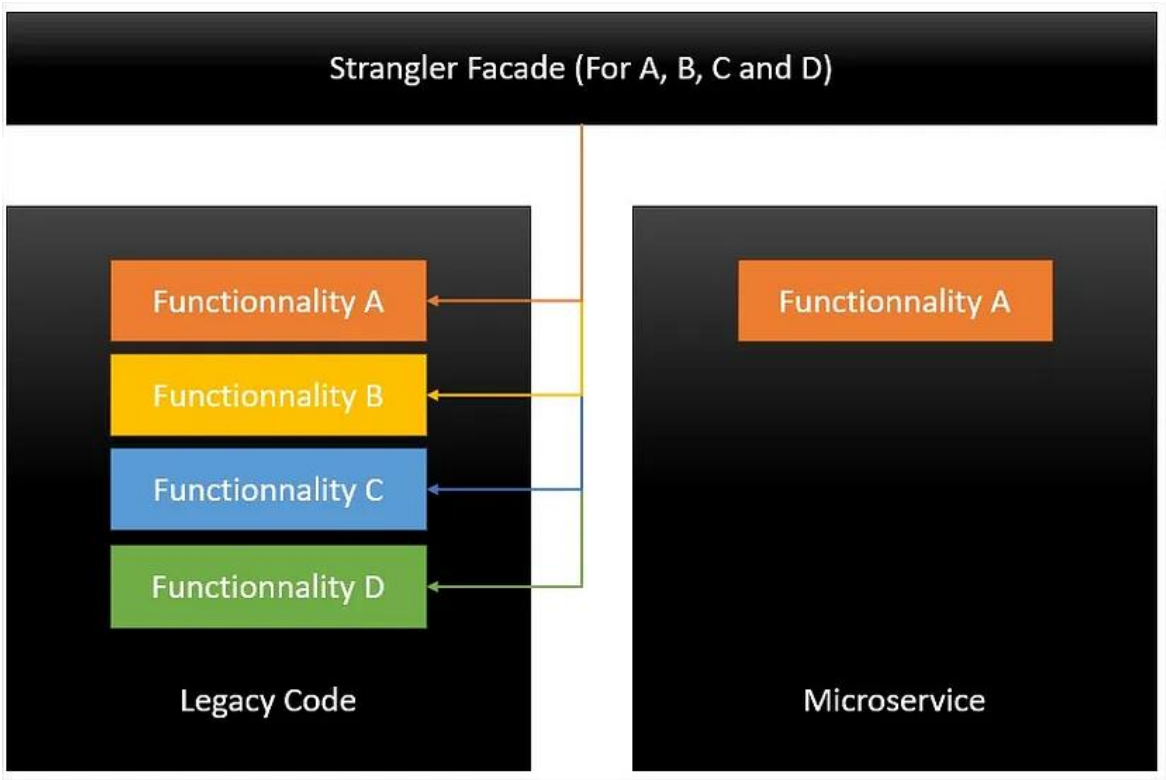


Рисунок 1.26 - Початкова кодова база моноліту[9]

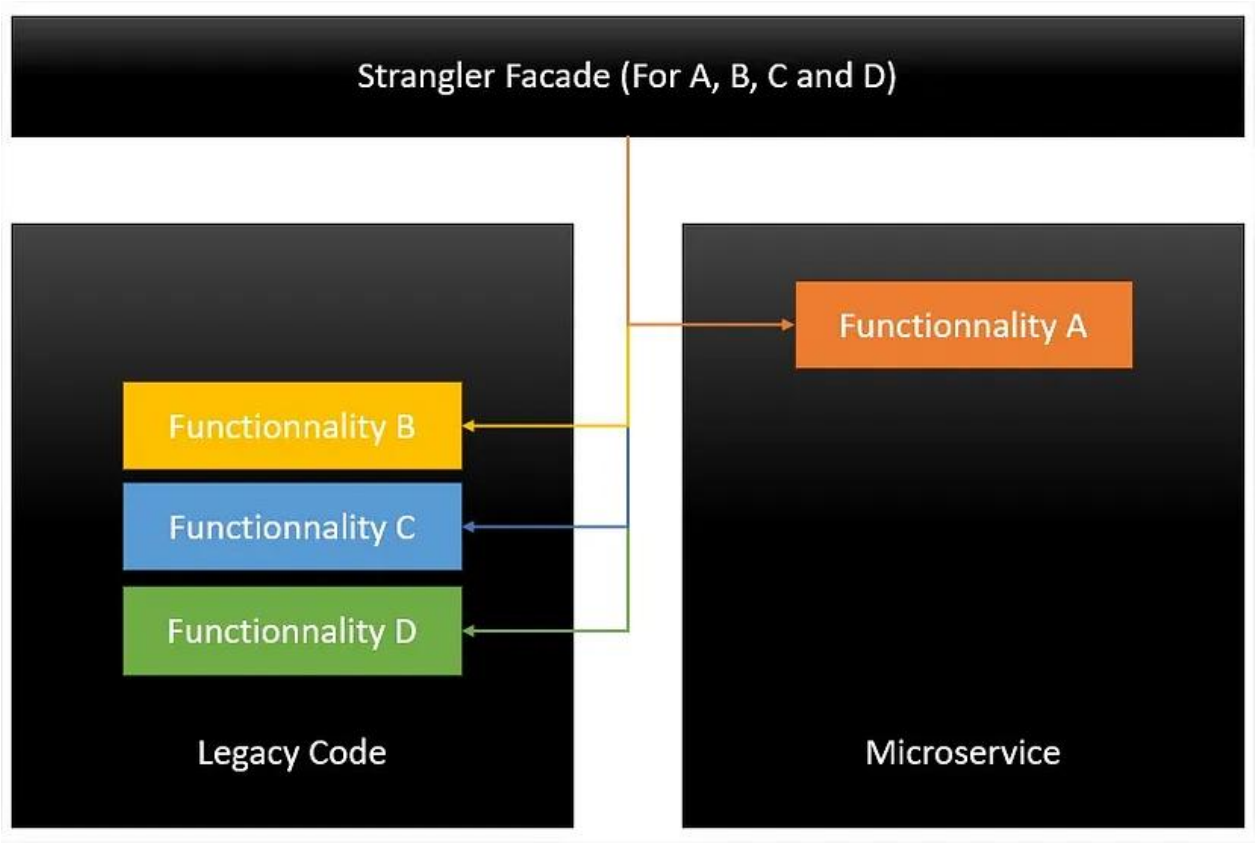


Рисунок 1.27 - Зміна посилання між фасадом і старим кодом на нове посилання, яке об’єднує фасад із новостворених мікросервісом, видалення функціональності A з моноліту.[9]

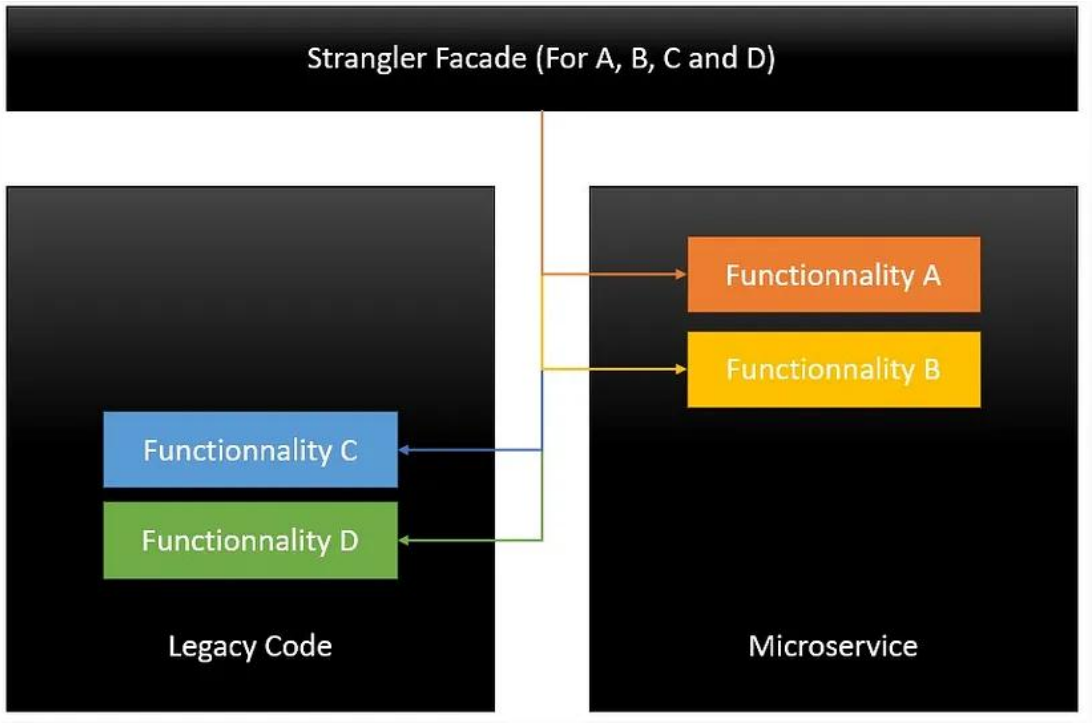


Рисунок 1.28 - Інкрементування процесу[9]

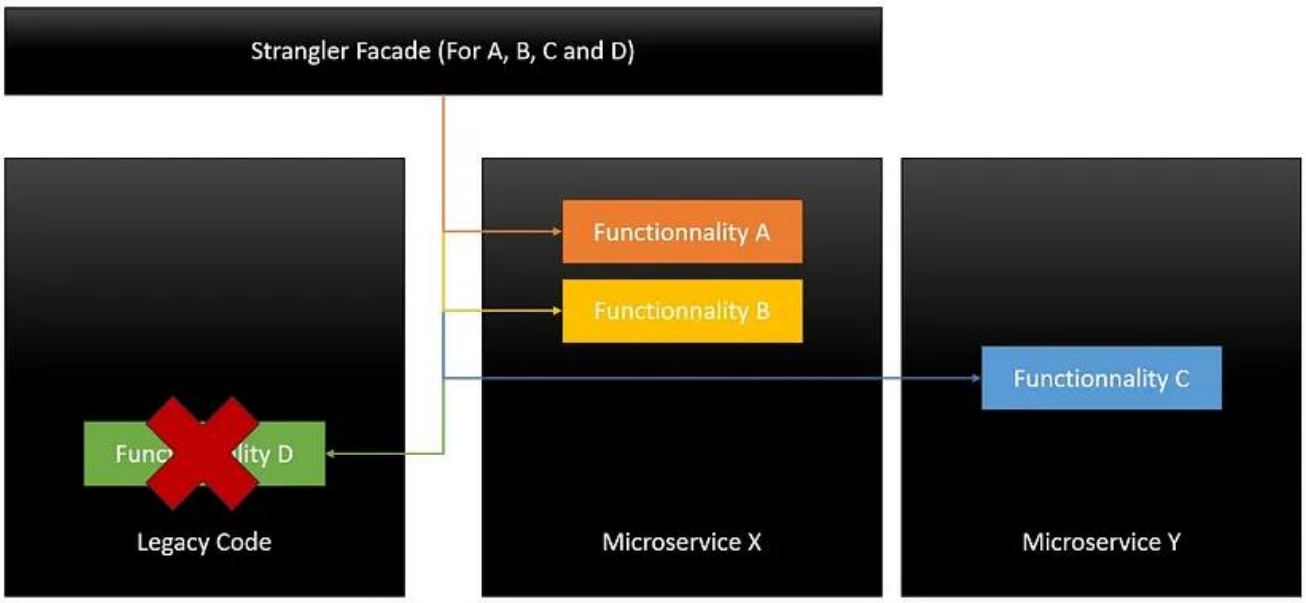


Рисунок 1.29 - Міграція функціоналу C до іншого мікросервісу (Y) із збереженням того ж процесу, видалення зайвого функціоналу D[9]

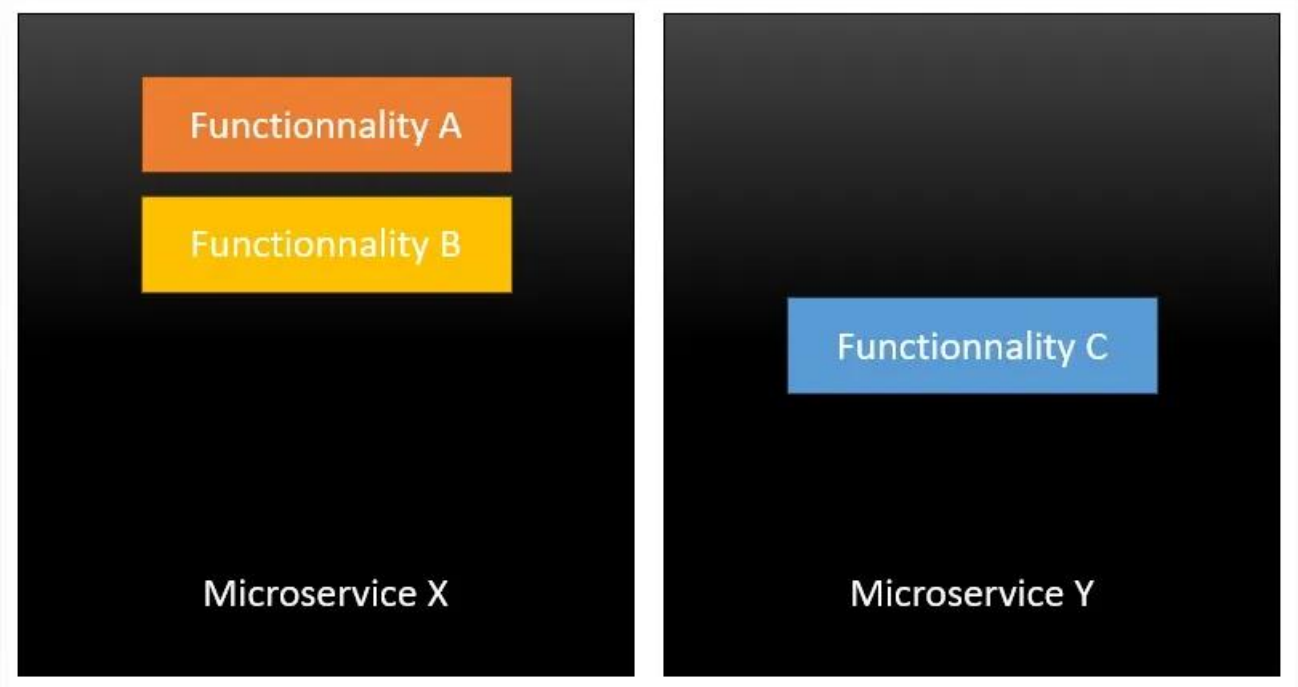


Рисунок 1.30 - Після успішної міграції код із моноліту видаляється[9]

На додаток, патерни можна об’єднувати разом для вирішення спільних проблем, у програмуванні – це поширена і часто необхідна практика в складних системах. Спільна робота патернів може допомогти досягти оптимальної архітектури, адже кожен патерн має свої сильні і слабкі сторони. Наприклад,

поєднання патернів API Gateway і Strangler Fig. Strangler Fig допоможе з міграцією монолітної архітектури до мікросервісної, а API Gateway обробляти вхідні запити користувачів та маршрутизувати їх до відповідних мікросервісів.

1.5 Висновок до 1 розділу

У першому розділі було детально розглянуто монолітну і мікросервісну архітектури, проведено їх порівняльний аналіз, а також дослідження ключових патернів проєктування мікросервісної архітектури. Висновок першого розділу можна зробити на основі чотирьох підрозділів, які описують перший розділ.

1. **Монолітна архітектура:**

- Монолітна архітектура — це цілісна структура, яка має просту реалізацію та повне централізоване управління над компонентами програми.
- Монолітна архітектура має низькі витрати на початковому етапі розробки, зручність у невеликих проєктах та високу продуктивність у системах з мінімальною потребою у масштабуванні.
- Із ростом системи, коли впроваджується значне масштабування, у монолітної архітектури виникають проблеми і обмеження пов’язані зі складністю масштабування, стійкістю до збоїв та впровадженням оновлень.

2. **Мікросервісна архітектура:**

- Мікросервісна архітектура основана на приципі незалежних сервісах, які можуть бути автономно розроблятися, масштабуватися і оновлюватися.
- Мікросервісна архітектура має низку переваг: гнучкість, висока масштабованість, стійкість до збоїв та можливість використання різних технологій для кожного сервісу.
- Мікросервісна архітектура для реалізації потребує більшої відповідальності та досвіду, так як мікросервіси можуть розроблятися різними стеками технологій, тому треба більше знань і навичок оперування даними технологіями у команді, як наслідок, зі збільшенням інфраструктури, існує потреба в досвідченій

команді для впровадження і підтримки мікросервісної архітектури. На додаток, мікросервісна архітектура складніше розгортається, потребує складнішого планування та будування складної інфраструктури.

3. **Порівняльний аналіз двох архітектур:**

- Представлено переваги і недоліки монолітної і мікросервісної архітектур та їхне табличне порівняння. У таблиці порівняння були представлені ключові аспекти вибору між монолітною та мікросервісною архітектурами за такими критеріями: масштабованість, продуктивність, гнучкість у виборі технологій, стійкість до збоїв та вартість реалізації.
- Зроблено висновок, що вибір архітектури залежить від масштабу проєкту, його складності та довгострокових цілей.

4. **Патерни проєктування мікросервісів**:

- Розглянуто патерни проєктуванння мікросервісної архітектури: API Gateway, Database per Service, Backend for Frontend, Event Sourcing, Saga Pattern та інші. Патерни проєктування відіграють важливу роль у розбудові, управлінні та масштабуванні розподільчих систем.
- Кожен із патернів має свої переваги та недоліки, які необхідно враховувати залежно від конкретних вимог проєкту.
- Патерни можна об’єднувати для підвищення продуктивності системи, але важливо враховувати сумісність патернів.

Висновок, вибір монолітної архітектури зазвичай обґрунтований легкістю реалізації, простотою взаємодії, високою продуктивністю та низькими витратами на розробку, але водночас із подальшим розвитком системи монолітна архітектура починає стикатися з проблемами масштабування, помилки в коді складніше прослідковуються, і для того, щоб розширити один із сервісів, треба переписувати код усього додатку через тісну залежність між компонентами. Проте, мікросервісна архітектура використовує відокремлені між собою сервіси, що дозволяє кожному сервісу розвиватися незалежно. Це значно спрощує

масштабування системи та локалізацію помилок, бо межі оперування скорочуються до одного сервіса. Мікросервісна архітектура набагато гнучкіша за монолітну. Гнучкість визначається застосуванням різних технологій, мов програмування та баз даних до кожного окремого сервіса, що робить систему більш адаптивною до змін. Мікросервісна архітектура відзначається стійкістю до збоїв: вихід з ладу одного сервісу не вплине на роботу інших незалежних сервісів. Однак така архітектура містить і свої недоліки, мікросервісна архітектура має складність управління розподіленими даними, потребує додаткових зусиль для забезпечення комунікації між сервісами та вищі витрати на інфраструктуру. Для вирішення дилеми вибору архітектури потрібно враховувати вимоги проєкту, масштаб системи та майбутні інтеграції й оновлення. Невеликі проєкти з чітко визначеним функціоналом зазвичай орієнтуються на монолітний архітектурний підхід, тоді як великі або динамічні проєкти з потребою частого оновлення і масштабування надають перевагу мікросервісній архітектурі.

2 ПОБУДОВА МОНОЛІТНОГО ДОДАТКУ НА JAVA ТА SPRING BOOT

2.1 Обґрунтування вибору технологій

Монолітний підхід був обраний, щоб продемонструвати структуру монолітної архітектури на реальному прикладі додатку Quizzer, який допомагає з організацією і керуванням тестувань. Усі компоненти додатку розміщені у єдиній кодовій базі, що притаманно монолітній архітектурі. Додаток простий в розгортанні, адже він невиликого розміру і споживає меньше системних ресурсів порівняно з мікросервісною архітектурою, оскільки не потребує підтримки окремих сервісів. Таким чином, при виборі монолітної архітектури враховується її простота та легкість реалізації при малих розмірах додатку, що дозволяє швидко отримати робочій продукт, який відповідає поставленим функціональним вимогам.

Технологічна основа додатку базується на таких технологіях: Java, Spring Boot 3 і PostgreSQL. Ці технології було обрано через їхню актуальність, надійність і сумісність один з одним:

1. Java (рис 2.1) — це об’єктно-орієнтована мова програмування високого рівня, яка використовується в різних галузях розробки, зокрема для створення веб-додатків, мобільних додатків, серверних рішень, корпоративних систем та навіть вбудованих пристроїв. Java має високу продуктивність, підходить для невиликих проєктів, так і для великих корпоративних, де підтверджує свою надійність і безпеку. Java має багатий набір бібліотек і фреймворків, які полегшують розробку додатків, наприклад фреймворк Spring, що використовується у веб-розробці.

Рисунок 2.1 - Логотип мови програмування Java

2. Spring Boot 3 (рис 2.2) — це фреймворк Java, який полегшує створення та запуск Java-додатків. Він спрощує процес конфігурації та налаштування, дозволяючи розробникам більше зосередитися на написанні коду для своїх додатків [10].

Рисунок 2.2. - Логотип фреймворку Spring Boot 3

3. PostgreSQL (рис 2.3) — це об'єктно-реляційна система управління базами даних (СКБД) на основі POSTGRES із потужними можливостями для обробки даних, підтримкою складних запитів і високою надійністю[12].

Рисунок 2.3. - Логотип бази даних PostgreSQL

Quizzer — веб-орієнтований додаток, тому для його реалізації було обрано мову програмування Java, щоб скористатися фреймворком Spring Boot, який використовується для розробки веб-додатків. Spring boot надає стартові інструменти для початку розробки додатків, зокрема:

- Автоматична конфігурація: на відміну з фреймворком Spring, який потребує ручної конфігурації для старту роботи, Spring Boot має автоматичну конфігурацію, що зменшує обсяг налаштування додатку.
- Вбудовані веб-сервери: Spring Boot підтримує інтеграцію із серверами, такими як Tomcat і Jetty, що дозволяють створювати локальні сесії для тестування додатку.
- Підтримка REST API: Spring Boot реалізує RESTful веб-сервіси.
- Гнучка інтеграція з базами даних: Spring boot оперує базою даних PostgreSQL через Spring Data JPA, що надає швидкий доступ до даних.

У Spring Boot використовуються інтерфейси замість конкретних класів для розділення рівнів додатків у поєднанні з ін’єкцією залежностей. Це зменшує зв’язаність між компонентами програми, підвищує гнучкість коду та спрощує його тестування. Spring Boot має в наявності готові конфігурації та стартові

пакети, які економлять ресурси і час, наприклад, модуль “spring-boot-starter-web” для швидкого створення веб-додатків або "spring-boot-starter-data-jpa" для роботи з базами даних. Завдяки цьому розробники можуть зосередитися на реалізації бізнес-логіки, а не витрачати час на налаштування інфраструктури. Хоча Spring Boot часто використовують для мікросервісної архітектури, не зважаючи на це, Sprting Boot чудово підоходить для монолітної архітектури завдяки високій продуктивності та зручності інтеграції. Завдяки великій кількості вбудованих бібліотек у Spring Boot, спрощується вирішення технічних проблем. Наприклад, біблотеки: Spring Data JPA, Spring Security, Validation API, тощо. Використання Java і Spring Boot забезпечує довгострокові перспективи, легке масштабування, модернізацію і інтеграції з іншими сервісами.

PostgrerSQL — це реляційна база даних, яка застосовується при роботі зі складними типами даних та операціями над ними. PostgreSQL дозволяє створювати власні функції, оператори, типи даних і навіть власні індекси, чого немає в багатьох інших базах, таких як MySQL. PostgreSQL має більший спектр функцій для проєктів. PostgreSQL забезпечує строгі транзакції, які виконуються за принципом ACID (Atomicity, Consistency, Isolation, Durability), що гарантує цілісність даних при помилках або збоях.

Отже, використовуючи даний стек технологій, було розроблено монолітну версію додатка Quizzer. Завдяки Java і Spring Boot 3 вдалося реалізувати швидку розробку, чітку архітектуру та легку інтеграцію різних модулів, таких як: "spring-boot-starter-data-jpa", "spring-boot-starter-web", "postgresql", "lombok", "spring-boot-starter-test" і "spring-boot-maven-plugin". База даних PostgreSQL, у свою чергу, забезпечує збереження та обробку даних із дотриманням строгих транзакцій.

2.2 Огляд функціональних вимог до системи

Додаток з назвою Quizzer призначений для створення, адміністрування та проходження тестування. Основні функції додатку:

- Створення опитування: можливість створювати тестування, визначаючи категорію, складність, кількість питань і назву.
- Перегляд питань опитування: отримання списку питань, що входять до тесту.
- Видалення опитування: видалення тестів за їх ідентифікатором.
- Оновлення питань в опитуванні: перевибір питань для існуючого тесту на основі категорії та рівня складності.
- Список тестів: отримання переліку всіх тестів із можливістю фільтрування за категорією та сортування за назвою, датою створення або складністю.
- Відповіді і результати: надання користувачами відповідей на питання тесту та отримання результатів.
- Доступ до дати створення опитування: перегляд дати створення тесту.

Додаток працює з питаннями, організовує їх в опитування, яке має певну тематику і категорію. Робота з питаннями включає такі дії: додавання нових питань із зазначенням тексту, видалення існуючих питань та перегляд усіх наявних питань. Кожне питання має декілька варіантів відповідей і одну або більше правильну відповіть. Всі відповіді зберігаються і перевіряються для підрахунку результату тестування. Для кожного опитування є можливість автоматичної генерації питань у заданій кількості, а для збереження актуальності, виконується динамічне оновлення питань в опитуваннях.

2.3 Проєктування додатку

Додаток побудований на основі кількох ключових моделей, сервісів та контролерів, що забезпечують його функціональність. Модель Quiz представляє опитування, містить основні атрибути, такі як: назва, категорія, складність, список питань та дата створення. Водночас модель Question описує окреме питання, яке має: текст, варіанти відповідей, правильну відповідь, категорію та складність. Для передачі питань у зручному для користувача форматі використовується клас QuestionWrapper, який обмежує доступ до правильних відповідей. Модель

Response служить для збереження відповідей користувачів. Дані про тести та питання зберігаються та обробляються за допомогою DAO-шару: QuizDao та QuestionDao, які надають доступ до бази даних. Логіка роботи з тестами зосереджена в сервісі QuizzerService, який забезпечує створення тестів, отримання питань та їх видалення. Сервіс QuestionService відповідає за роботу з питаннями, включаючи їх додавання та пошук за категоріями або складністю. Контролери QuizzerController та QuestionController забезпечують взаємодію з користувачами через HTTP-запити, викликаючи відповідні методи сервісів. Між моделями Quiz та Question існує зв'язок "один-до-багатьох", оскільки кожен тест може містити багато питань.

UML-діаграма представляє структуру програми. На ній зображено основні класи, які взаємодіять між собою (рис. 2.4).

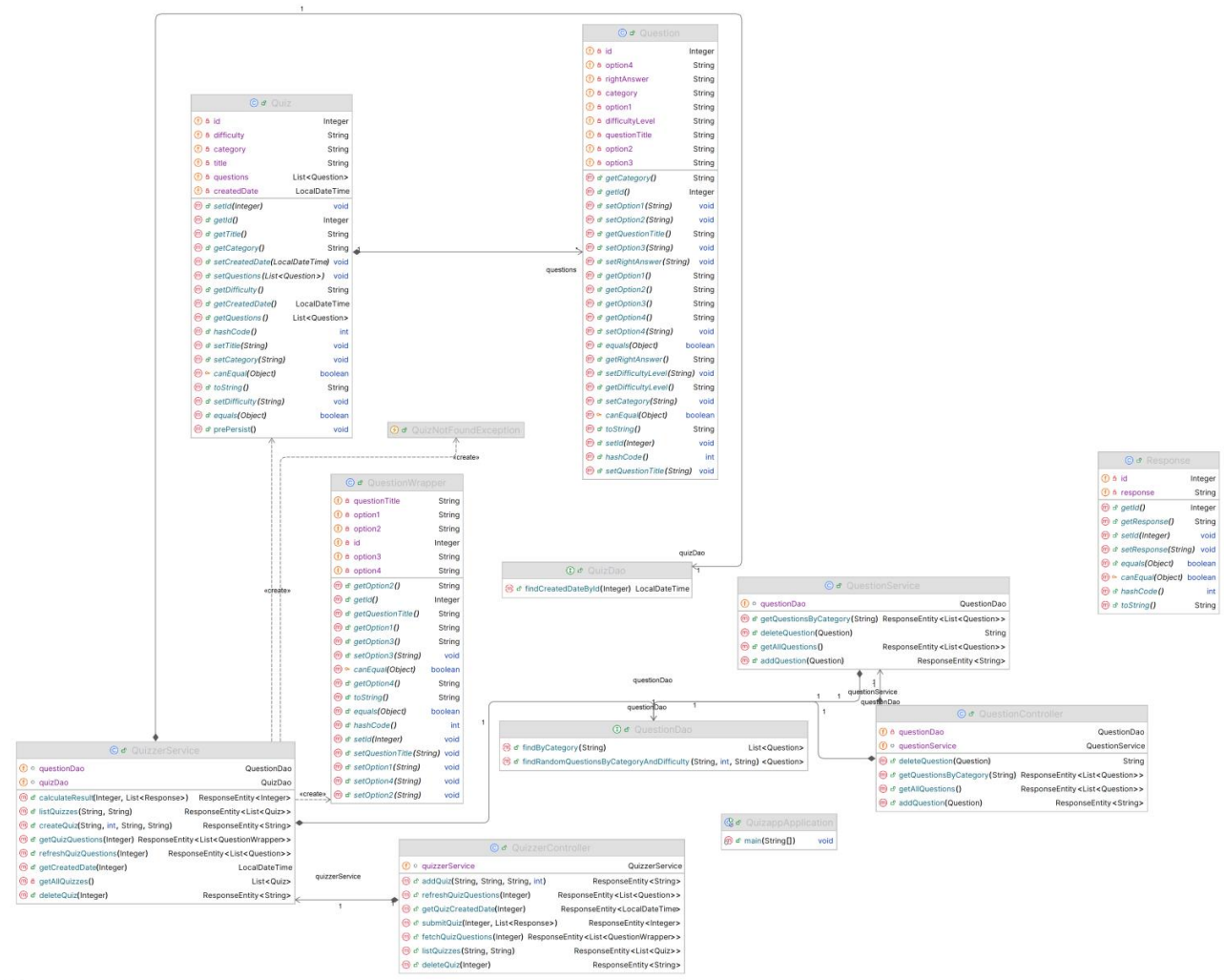


Рисунок 2.4 - UML-діаграма Quizzer

а) Класи-моделі і допоміжні сервіси:

1. Quiz:

Клас Quiz є моделлю опитування та відіграє роль контейнера для питань.

До атрибутів і методів Quiz належать:

- title, category, difficulty — загальна інформація про тест
- questions — список питань, пов'язаних із тестом
- createdDate — дата створення тесту
- prePersist() — автоматично встановлює дату створення (createdDate) перед збереженням у базу

2. Question:

Клас Question є моделлю питання. Даний клас відповідає за фомування питання.

До атрибутів Question належать:

- id — унікальний питання
- questionTitle, options — текст питання та варіанти відповідей
- rightAnswer — правильна відповідь
- category, difficulty — категорія та складність питання

3. QuestionWrapper:

Ключовий клас, який повторює клас Question, але без атрибуту rightAnswer, щоб надавати питання користувачу

Характеристики QuestionWrapper:

- DTO (Data Transfer Object) для передачі питань з опитування, обмежує доступ до полів, таких як rightAnswer (правильна відповідь)
- Містить атрибути, що відповідають атрибутам питання, але у форматі, зручному для користувача

4. Response:

Клас для збереження відповіді користувача.

До атрибутів Question належать:

- id, response — ідентифікатор відповіді та сам текст відповіді

5. QuizDao і QuestionDao:

Допоміжні сервіси, які використовуються для роботи з базою даних. QuizDao і QuestionDao виконують свої індивідуальні функції:

- QuizDao реалізує методи для пошуку опитувань за різними критеріями
- QuestionDao надає методи для отримання питань на основі категорій, складності тощо

b) Основні сервіси, які відповідають за логіку програми:

1. QuizService:

Відповідає за логіку роботи з тестами.

Основні методи:

- createQuiz(String category, int numQ, String title, String difficulty) — створює новий тест із заданими параметрами
- getQuizQuestions(Integer id) — повертає питання конкретного тесту у вигляді QuestionWrapper
- calculateResult(Integer id, List<Response> responses) — обчислює результат тесту на основі наданих відповідей
- deleteQuiz(Integer id) — видаляє тест за ідентифікатором
- refreshQuizQuestions(Integer quizId) — оновлює список питань у тесті
- getCreatedDate(Integer id) — повертає дату створення тесту
- listQuizzes(String category, String sortBy) — повертає список тестів з можливістю сортування та фільтрації

2. QuestionService

Відповідає за управління питаннями у програмі

Методи:

- getAllQuestions() — повертає список усіх питань
- getQuestionsByCategory(String category) — повертає питання, відфільтровані за категорією
- addQuestion(Question question) — додає нове питання
- deleteQuestion(Question question) — видаляє вказане питання

c) Контролери:

1. QuizzerController

Відповідає за управління опитуванями.

Методи:

- addQuiz() — додає новий тест
- deleteQuiz() — видаляє тест за ID
- fetchQuizQuestions() — повертає питання тесту
- getQuizCreatedDate() — отримує дату створення тесту
- refreshQuizQuestions() — оновлює питання тесту
- listQuizzes() — повертає список тестів з фільтрацією
- submitQuiz() — обчислює результат тесту

2. QuestionController

Відповідає за управління питаннями.

Методи:

- getAllQuestions() — повертає список усіх питань
- getQuestionsByCategory() — отримує питання за категорією
- addQuestion() — додає нове питання
- deleteQuestion() — видаляє питання

2.4 Реалізація монолітного додатку

Монолітний проєкт Quizzer було реалізовано використовуючи: Java 21, Spring Boot 3 і PostgreSQL. Для розробки було обрано середовище озробки IntelliJ IDEA Ultimate через його популярність, простоту, розумний редактор коду і наявність великого спектру плагінів та технологій.

1. Створення проєкту

При створені проєкту, IntelliJ IDEA має список генераторів, які визначають, якою мовою чи фреймворком буде реалізовано проєкт. Так як проєкт реалізовано на фреймворку Spring Boot 3, його було обрано в списку генераторів (рис 2.5). IntelliJ IDEA безпосердньо, для генерації проєкту на Spring Boot, використовує посилання на офіційний сервіс Spring Initializr, який автоматично генерує готовий для роботи проєкт на фрейморку Spring. Доступ до Spring Initializr не залежить від середовища розробки, так як це безкоштовний сервіс, звернутися до нього можна через браузер за посиланням https://start.spring.io[13].

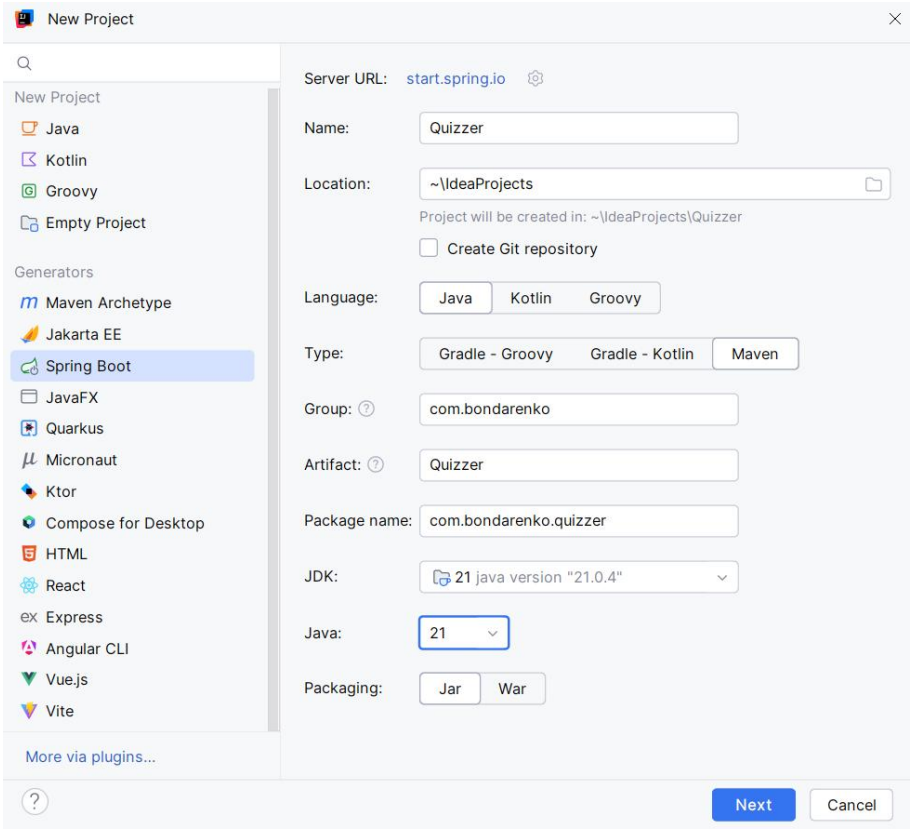


Рисунок 2.5 - Вікно налаштування проєкту

У вікні створення проєкта пропонується надати: ім’я, місце знаходження в системі, мову програмування, тип управління проєктом і т.д. Для типу управління

проєктом було обрано інструмент управління Maven. Maven — це інструмент для управління проєктами та системою складання (build tool) у Java, який використовується для автоматизації процесів створення, компіляції, тестування та розгортання програмного забезпечення[13]. Maven має основний конфігураційний файл pom.xml (Project Object Model), в котрому визначаються усі налаштування проєкту: залежності, плагіни та інші метадані. В pom-файлі проєкту містяться залежності такі як: spring-boot-starter-data-jpa, spring-boot-starter-web, postgresql, lombok і spring-boot-starter-test. У вікні створення проєкту можна відразу обрати збірку проєкту під конкретну задачу (рис. 2.6).

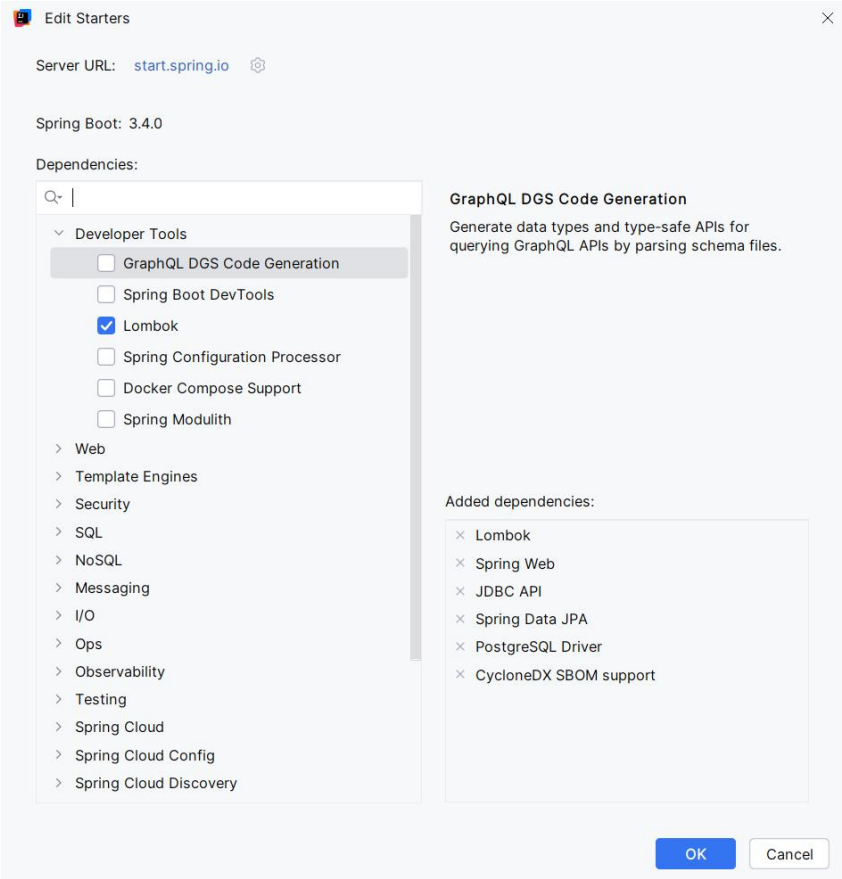


Рисунок 2.6 - Вікно вибору залежностей

Після вибору залежностей генерується структура проєтку з файлом pom.xml, де відбувається конфігурація проєкту. Фрагмент pom.xml із залежностями знаходиться на рисунках 2.7 - 2.8.

```
<dependencies> Edit Starters...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

Рисунок 2.7 - Фрагмент pom.xml із залежностями Data JPA і Starter Web

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

Рисунок 2.8 - Фрагмент pom.xml із залежностями PostgreSQL і Lombok

Після завершення конфігурації pom.xml, далі слідує конфігурація файлу Spring application properties (рис. 2.9). У цьому файлі відбуваються налаштування для підключення до бази даних PostgreSQL.

```
1  spring.application.name=quizzer
2  spring.datasource.url=jdbc:postgresql://localhost:5432/questiondb
3  spring.datasource.username=postgres
4  spring.datasource.password=1111
5  spring.jpa.hibernate.ddl-auto=update
6  spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Рисунок 2.9 - Фрагмент pom.xml

2. Реалізація REST-контролерів

Контролери відповідають за обробку HTTP-запитів та взаємодіють із сервісами. Налаштування контролерів в pom.xml не потребується завдяки

анотаціям, які мають механізм автоконфігурації. У додатку реалізовано 2 контролери: QuizzerContoller і QuestionController.

QuizzerContoller відповідає за інтеракцію з тестами. У QuizzerContoller є анотація @RestController, яка вказує, що клас є REST-контролером та виконує обробку HTTP-запитів і повертає дані в форматі JSON або XML, а анотація @RequestMapping("quizzer") задає базовий шлях для всіх методів у класі. Всі запити до цього контролера починатимуться з "quizzer". (рис. 2.10).

```
15    @RestController
16    @RequestMapping("quizzer")
17    public class QuizzerController {
18        @Autowired
19        QuizzerService quizzerService;
```

Рисунок 2.10 - Фрагмент контролера QuizzerControler з анотаціями

QuizzerControler має один атрибут quizzerService під типом даних QuizzerService, який має анотацію @Autowired. Дана анотація позбавляє потреби створення екземплярів для змінних, залежність передається безпосередньо в поле класу, цей процес називається автоматичною ін'єкцією залежностей. Приклад методів QuizzerControler зображено на рисунку 2.11.

```
27    @PostMapping("addQuiz")
28    public ResponseEntity<String> addQuiz(@RequestParam String title, @RequestParam String category,
29        return quizzerService.createQuiz(category, numQuestions, title, difficulty);
30    }
31
32    @DeleteMapping("delete/{id}")
33    public ResponseEntity<String> deleteQuiz(@PathVariable Integer id) { return quizzerService.delete
36    @GetMapping("get/{id}")
37    public ResponseEntity<List<QuestionWrapper>> fetchQuizQuestions(@PathVariable Integer id) {
38        return quizzerService.getQuizQuestions(id);
39    }
40
41    @GetMapping("{id}/createdDate")
42    public ResponseEntity<LocalDateTime> getQuizCreatedDate(@PathVariable Integer id) {
43        LocalDateTime createdDate = quizzerService.getCreatedDate(id);
44        return ResponseEntity.ok(createdDate);
```

Рисунок 2.11 - Фрагмент методів QuizzerControler

QuestionController відповідає за обробку запитів, які пов’язані з питаннями. Він містить індентичні анотації, що має QuizzerController, але в @RequestMapping використовує посилання "question" (рис. 2.12).

```
12  @RestController
13  @RequestMapping("question")
14  public class QuestionController {
15      @Autowired
16      QuestionService questionService;
17
```

Рисунок 2.12 - Фрагмент контролера QuestionController з анотаціями

QuestionController має один атрибут під назвою questionService та методи роботи з питаннями (рис 2.13).

```
17
18      @GetMapping("allQuestions")
19      public ResponseEntity<List<Question>> getAllQuestions() {
20          return questionService.getAllQuestions();
21      }
22
23      @GetMapping("category/{category}")
24      public ResponseEntity<List<Question>> getQuestionsByCategory(@PathVariable String category) {
25          return questionService.getQuestionsByCategory(category);
26      }
27
28      @PostMapping("add")
29      public ResponseEntity<String> addQuestion(@RequestBody Question question) {
30          return questionService.addQuestion(question);
31
32      }
```

Рисунок 2.13 - Фрагмент методів QuestionController

3. Сервіси для обробки логіки

Сервіси в архітектурі Spring застосовуються для реалізації логіки, яка відповідає за виконання основних завдань додатка. Вони є проміжним шаром між контролерами (які обробляють запити користувачів) і DAO (які безпосередньо працюють із базою даних). Кожен сервіс виконує чітко визначені завдання та співпрацює з контролерами, отрумає дані від контролера та обробляє їх в

потрібний формат DAO. Клас-сервіс позначається анотацією @Service, яка вказує Spring, що цей клас є компонентом логіки і повинен бути доданий до контексту додатка. Сервіс може отримувати доступ до DAO чи інших сервісів через ін'єкцію залежностей (наприклад, @Autowired).

QuizzerService відповідає за логіку взаємодії з тестами. QuizerService містить два атрибути: quizDao і questionDao (рис. 2.14). Ці атрибути допомагають при взаємодії з базою даних. Поле quizDao відповідає за доступ до таблиці тестів у базі даних та CRUD-операцій із тестами, наприклад: створення, отримання, оновлення та видалення тестів. Інше поле questionDao відповідає за доступ до таблиці питань та дозволяє отримувати, додавати або видаляти питання, забезпечує пошук питань за категоріями або складністю.

```
18
19    @Service  1 usage
20    public class QuizzerService {
21        @Autowired
22        QuizDao quizDao;
23        @Autowired
24        QuestionDao questionDao;
25
```

Рисунок 2.14 - Фрагмент атрибутів сервісу QuizzerService

QuizzerService не був би сервісом, якщоб не вмів нічого робити з даними у базі даних. Методи описують логіку сервісу і забезпечують: створення, видалення, отримання питань і оновлення тестів. Основний метод для створення тесту називається createQuiz (рис. 2.15).

```
public ResponseEntity<String> createQuiz(String category, int numQ, String title, String difficulty) {  1 usage
    Quiz quiz = new Quiz();
    List<Question> questions = questionDao.findRandomQuestionsByCategoryAndDifficulty(category, numQ, difficulty);
    quiz.setTitle(title);
    quiz.setQuestions(questions);
    quiz.setCategory(category);
    quiz.setDifficulty(difficulty);
    quizDao.save(quiz);
    return new ResponseEntity<>( body: "Success", HttpStatus.CREATED);
}
```

Рисунок 2.15 - Метод createQuiz() класу QuizzerService

Метод createQuiz() приймає 4 параметри, які потрібні для опису тесту. В тілі метода створюється екземпляр класу Quiz для виклику методів-сеттерів: setTitle(), setQuestions(), setCategory() і setDifficulty(). Варто зазначити, що після екземпляру сторюється список questions, який ініціалізується викликом методу findRandomQuestionsByCategoryAndDifficulty(), який належить класу questionDao, де виконується операція з базою даних для отримання бажаного результату (рис 2.16). Якщо робота методу createQuiz() виявилася успішною, метод повертає об'єкт ResponseEntity<>("Success", HttpStatus.CREATED), де "Success" означає, що робота була успішна, а HttpStatus.CREATED (код 201) показує, що опитування було успішно створено.

```
10
11  @Repository  5 usages
12  public interface QuestionDao extends JpaRepository<Question, Integer> {
13      List<Question> findByCategory(String category);  1 usage
14
15      @Query(value = "SELECT * FROM question q WHERE q.category = :category AND q.difficulty_level = " +  2 usages
16              ":difficulty ORDER BY RANDOM() LIMIT :numQ", nativeQuery = true)
17      List<Question> findRandomQuestionsByCategoryAndDifficulty(
18              @Param("category") String category,
19              @Param("numQ") int numQ,
20              @Param("difficulty") String difficulty
21      );
22
23  }
```

Рисунок 2.16 - Метод findRandomQuestionsByCategoryAndDifficulty()

Фрагменти інших методів QuizzerService знаходяться на рисунках 2.17 - 2.19.

```
38      public ResponseEntity<List<QuestionWrapper>> getQuizQuestions(Integer id) {  1 usage
39          Optional<Quiz> quiz = quizDao.findById(id);
40          List<Question> questionsFromDB = quiz.get().getQuestions();
41          List<QuestionWrapper> questionsForUser = new ArrayList<>();
42          for (Question question : questionsFromDB) {
43              QuestionWrapper qw = new QuestionWrapper(question.getId(), question.getQuestionTitle(), question.getOption1(), question.getOption2(),
44                      question.getOption3(), question.getOption4());
45              questionsForUser.add(qw);
46          }
47          return new ResponseEntity<>(questionsForUser, HttpStatus.OK);
48      }
49
50      public ResponseEntity<Integer> calculateResult(Integer id, List<Response> responses) {  1 usage
51          Quiz quiz = quizDao.findById(id).get();
52          List<Question> questions = quiz.getQuestions();
53          int rightAnswers = 0;
54          int counter = 0;
55          for (Response response : responses) {
56              if (response.getResponse().equals(questions.get(counter).getRightAnswer()))
57                  rightAnswers++;
58              counter++;
59          }
60          return new ResponseEntity<>(rightAnswers, HttpStatus.OK);
61      }
```

Рисунок 2.17 - Методи getQuizQuestions() і calculateResult() класу QuestionService

```java
    public ResponseEntity<List<Quiz>> listQuizzes(String category, String sortBy) {  1 usage
        List<Quiz> quizzes = getAllQuizzes();
        if (category != null && !category.isEmpty()) {
            quizzes = quizzes.stream()
                    .filter(quiz -> quiz.getCategory().equalsIgnoreCase(category))
                    .collect(Collectors.toList());
        }
        if (sortBy != null) {
            switch (sortBy.toLowerCase()) {
                case "title":
                    quizzes.sort(Comparator.comparing(Quiz::getTitle));
                    break;
                case "created":
                    quizzes.sort(Comparator.comparing(Quiz::getCreatedDate));
                    break;
                case "difficulty":
                    quizzes.sort(Comparator.comparing(Quiz::getDifficulty));
                    break;
                default:
                    break;
            }
        }
        if (quizzes.isEmpty()) {
            return ResponseEntity.noContent().build();
        }

        return ResponseEntity.ok(quizzes);
    }
```

Рисунок 2.18 - Метод listQuizzes() класу QuestionService

```java
public ResponseEntity<List<Question>> refreshQuizQuestions(Integer quizId) {  1 usage
    // Handle Optional correctly
    Quiz quiz = quizDao.findById(quizId).orElseThrow(() -> new QuizNotFoundException(quizId));

    // Fetch new random questions based on the quiz's category and difficulty
    List<Question> newQuestions = questionDao.findRandomQuestionsByCategoryAndDifficulty(quiz.getCategory(), quiz.getQuestions().size(), quiz.getDifficulty());
    Collections.shuffle(newQuestions); // Shuffle to ensure randomness

    // Clear the existing questions and set new ones
    quiz.getQuestions().clear();
    quiz.setQuestions(newQuestions);

    // Save the updated quiz to the database
    quizDao.save(quiz);

    // Return the new list of questions as the response
    return ResponseEntity.ok(newQuestions);
}
```

Рисунок 2.19 - Метод refreshQuizQuestions() класу QuestionService

4. DAO (Data Access Object) для роботи з базою даних

DAO-інтерфейси забезпечують взаємодію з базою даних через Spring Data JPA.

QuizDao працює з таблицею опитувань у базі даних (рис. 2.20).

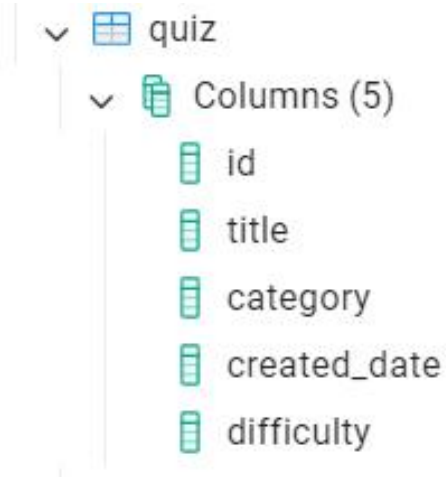


Рисунок 2.20 - Таблиця quiz в базі даних PostgreSQL

QuizDao (рис. 2.21) — це інтерфейс, який розширює JpaRepository у Spring Data JPA. Методи, які має цей інтерфейс, або автоматично надаються фреймворком, або оголошуються користувачем (рис. 2.22 - 2.23). QuizDao має анотацію @Repository для позначення, що цей інтерфейс працює з базою даних.

```
10
11  @Repository  2 usages
12  public interface QuizDao extends JpaRepository<Quiz,Integer> {
13      @Query("SELECT q.createdDate FROM Quiz q WHERE q.id = :id")  1 usage
14      LocalDateTime findCreatedDateById(@Param("id") Integer id);
15  }
16
17
```

Рисунок 2.21 - Інтерфейс QuizDao

```
68      public ResponseEntity<List<Question>> refreshQuizQuestions(Integer quizId) {  1 usage
69          // Handle Optional correctly
70          Quiz quiz = quizDao.findById(quizId).orElseThrow(() -> new QuizNotFoundException(quizId));
71
72          // Fetch new random questions based on the quiz's category and difficulty
73          List<Question> newQuestions = questionDao.findRandomQuestionsByCategoryAndDifficulty(quiz.getCategory(), quiz.getQuestion
74          Collections.shuffle(newQuestions); // Shuffle to ensure randomness
75
76          // Clear the existing questions and set new ones
77          quiz.getQuestions().clear();
78          quiz.setQuestions(newQuestions);
79
80          // Save the updated quiz to the database
81          quizDao.save(quiz);
82
83          // Return the new list of questions as the response
84          return ResponseEntity.ok(newQuestions);
85      }
```

Рисунок 2.22 - Використання методу save() для зберігання оновленого тесту

```
119
120 @       private List<Quiz> getAllQuizzes() {  1 usage
121             return quizDao.findAll();
122         }
123     }
```

Рисунок 2.23 - Використання методу findAll() для пошуку усіх тестів в базі даних

QuestionDao працює з таблицею питань у базі даних (рис. 2.24).

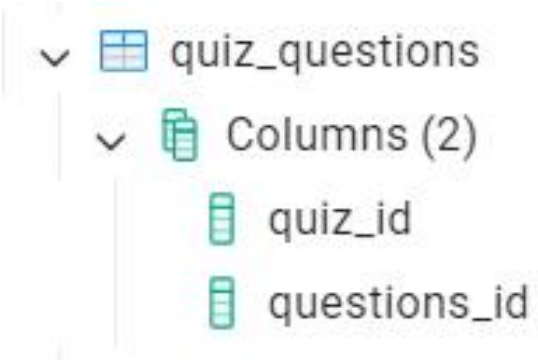


Рисунок 2.24 - Таблиця quiz_questions в базі даних PostgreSQL

QuestionDao (рис 2.25) — це інтерфейс для роботи з таблицею питань у базі даних, що реалізує патерн DAO (Data Access Object). Він дозволяє виконувати CRUD-операції (створення, читання, оновлення, видалення) та забезпечує доступ

до даних через Spring Data JPA. QuestionDao має анотацію @Repository для позначення, що цей інтерфейс працює з базою даних. QustionDao має спеціальні методи для роботи з питаннями:

- findByCategory() — повертає список питань, що належать до заданої категорії, використовуючи автоматично згенерований Spring Data JPA запит.
- findRandomQuestionsByCategoryAndDifficulty() — виконує нативний SQL-запит для вибору випадкових питань з бази даних на основі заданої категорії, рівня складності та кількості питань (numQ). Запит сортує записи випадковим чином за допомогою ORDER BY RANDOM() і обмежує кількість повернених результатів через LIMIT.

```
10
11      @Repository  5 usages
12      public interface QuestionDao extends JpaRepository<Question, Integer> {
13          List<Question> findByCategory(String category);  1 usage
14
15          @Query(value = "SELECT * FROM question q WHERE q.category = :category AND q.difficulty_level " +  2 usages
16                  "= :difficulty ORDER BY RANDOM() LIMIT :numQ", nativeQuery = true)
17          List<Question> findRandomQuestionsByCategoryAndDifficulty(
18                  @Param("category") String category,
19                  @Param("numQ") int numQ,
20                  @Param("difficulty") String difficulty
21          );
22
23      }
```

Рисунок 2.25 - Інтерфейс QuestionDao і методи

5. Моделі даних

Моделі даних (рис. 2.26) — це класи, які визначають структуру даних, з якими буде працювати база даних. Quizzer використовує 4 моделі даних: Question, Quiz і т.д. Дані класи моделюють опитування, питання і відповіді.

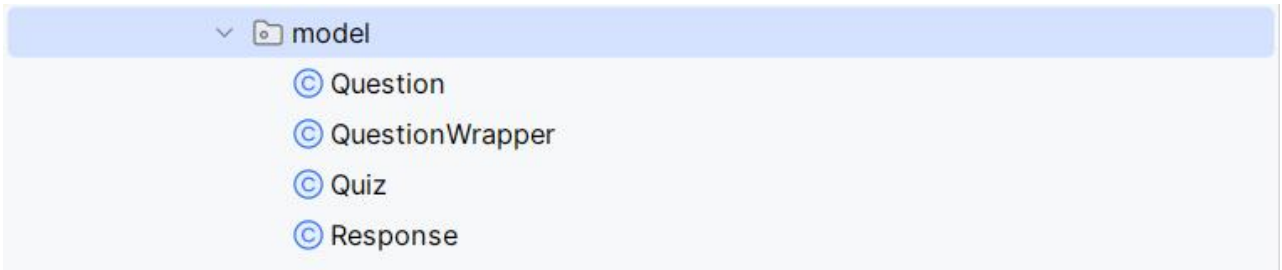


Рисунок 2.26 - Пакет model

Клас Quiz (рис. 2.27) є моделлю опитування. Він містить такі дані як: унікальний ідентифікатор, інформація про опитування, список питань і дату створення. В класі Quiz використовуються анотації: @Entity (сутність JPA), @Data (анотація lombok, автоматичне додавання методів), @Id (первинний ключ таблиці), @GeneratedValue(strategy = GenerationType.IDENTITY) (автоматична генерація первинного ключа зі стратегією автоінкремент IDENTITY), @Column(nullable = false, updatable = false) (налаштування колонки в таблиці), @ManyToMany (вказує зв'язок "багато-до-багатьох" між Quiz і Question) і @PrePersist (автоматичного встановлення значень).

```
16    @Entity  17 usages
17    @Data
18    public class Quiz {
19
20        @Id
21        @GeneratedValue(strategy = GenerationType.IDENTITY)
22        private Integer id;
23        private String title;
24        private String category;
25        private String difficulty;
26
27        @Column(nullable = false, updatable = false)
28        private LocalDateTime createdDate;
29
30        @ManyToMany
31        private List<Question> questions;
32
33
34        @PrePersist
35        public void prePersist() { this.createdDate = LocalDateTime.now(); }
38
39    }
```

Рисунок 2.27. - Клас-модель Quiz

Клас Question (рис. 2.28) — модель питання. Фактично описує питання загальновживаними атрибутами притаманними питанню, такими як: унікальний ідентифікатор, текст питання, варіанти відповідей, правильна відповідь, категорія та складність. В класі Question використовуються анотації: @Data (сутність JPA), @Id (первинний ключ таблиці), @GeneratedValue(strategy =

GenerationType.IDENTITY) (автоматична генерація первинного ключа зі стратегією автоінкремент IDENTITY).

```java
@Data  24 usages
@Entity
public class Question {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String questionTitle;
    private String option1;
    private String option2;
    private String option3;
    private String option4;
    private String rightAnswer;
    private String difficultyLevel;
    private String category;
}
```

Рисунок 2.28 - Клас-модель Question

QuestionWrapper (рис. 2.29) — це DTO (Data Transfer Object), який використовується для передачі питань від сервера до клієнта. Він створений для того, щоб приховати правильну відповідь (rightAnswer) від користувача, зберігаючи лише необхідні дані для відображення питання та варіантів відповідей.

```java
@Data 7 usages
public class QuestionWrapper {
    private Integer id;
    private String questionTitle;
    private String option1;
    private String option2;
    private String option3;
    private String option4;

    public QuestionWrapper(Integer id, String questionTitle, String option1, String option2, String option3, String option4) {
        this.id = id;
        this.questionTitle = questionTitle;
        this.option1 = option1;
        this.option2 = option2;
        this.option3 = option3;
        this.option4 = option4;
    }
}
```

Рисунок 2.29 - Клас-модель QuestionWrapper

Response (рис. 2.30) — це модель, яка використовується для збереження відповідей, наданих користувачем під час проходження опитування. Вона може бути передана від клієнта до сервера, щоб перевірити правильність відповідей.

```
6    @Data  5 usages
7    @RequiredArgsConstructor
8    public class Response {
9        private Integer id;
10       private String response;
11   }
12
```

Рисунок 2.30 - Клас-модель Response

2.5 Тестування та перевірка роботи монолітного додатку

Тестування додатку було проведено в програмі Postman. Postman (рис. 2.31) — це програма, яка дозволяє розробникам тестувати, документувати та ділитися API (інтерфейси прикладного програмування). Розробники широко використовують його для спрощення процесу тестування API, надаючи зручний інтерфейс для створення запитів, перегляду відповідей та налагодження проблем[13].

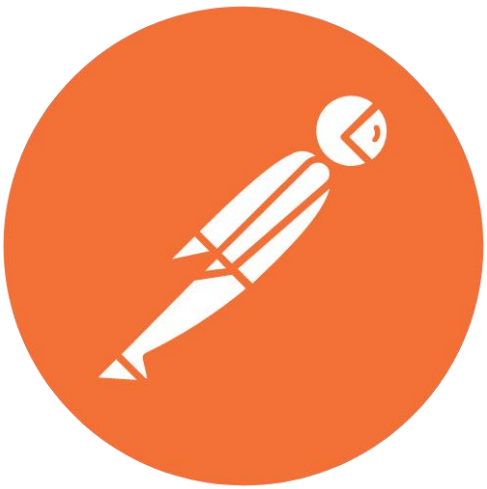


Рисунок 2.31 - Postman

Для того, щоб здійснити тестування треба звернутися до контролерів через HTTP-запити. Першим контролером було протестовано QuestionController. Даний контролер відповідає за модерацію дій над питаннями. QuestionController є ключовим елементом, без якого створення тестів не може бути можливим.

Тестування QuestionController:

1) Тестування функції створення питання (рис. 2.32).

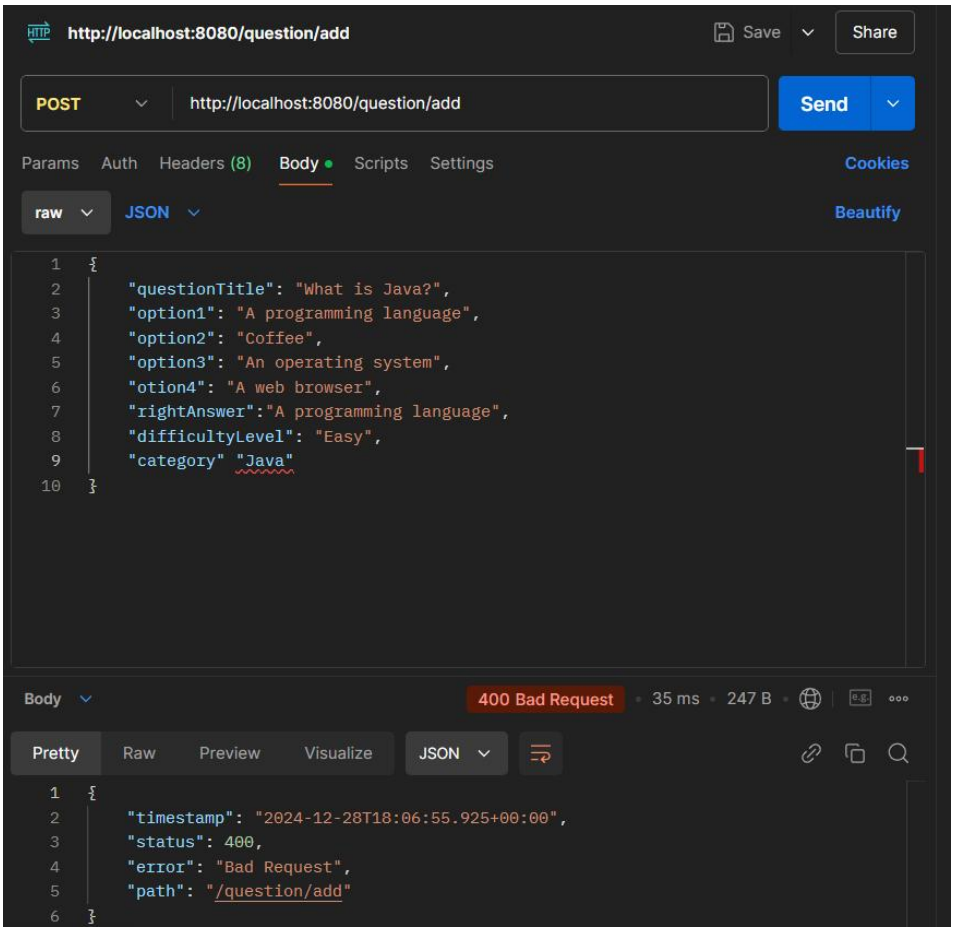


Рисунок 2.32 - Невдала спроба додавання питання через помилку у 9 рядку

Додавання питання відбувається через HTTP-запити, наприклад запит POST: "http://localhost:8080/question/add". Додавання питання вимагає правильного синтаксису, неправильний запис може викликати помилку, яка має зарезервований HTTP статус "400", що означає "Bad Request". Після аналізу характеру похибки і виправлення маємо успішний результат, який підтверджує повідомлення "Question added" і HTTP статус "201". (рис. 2.33).

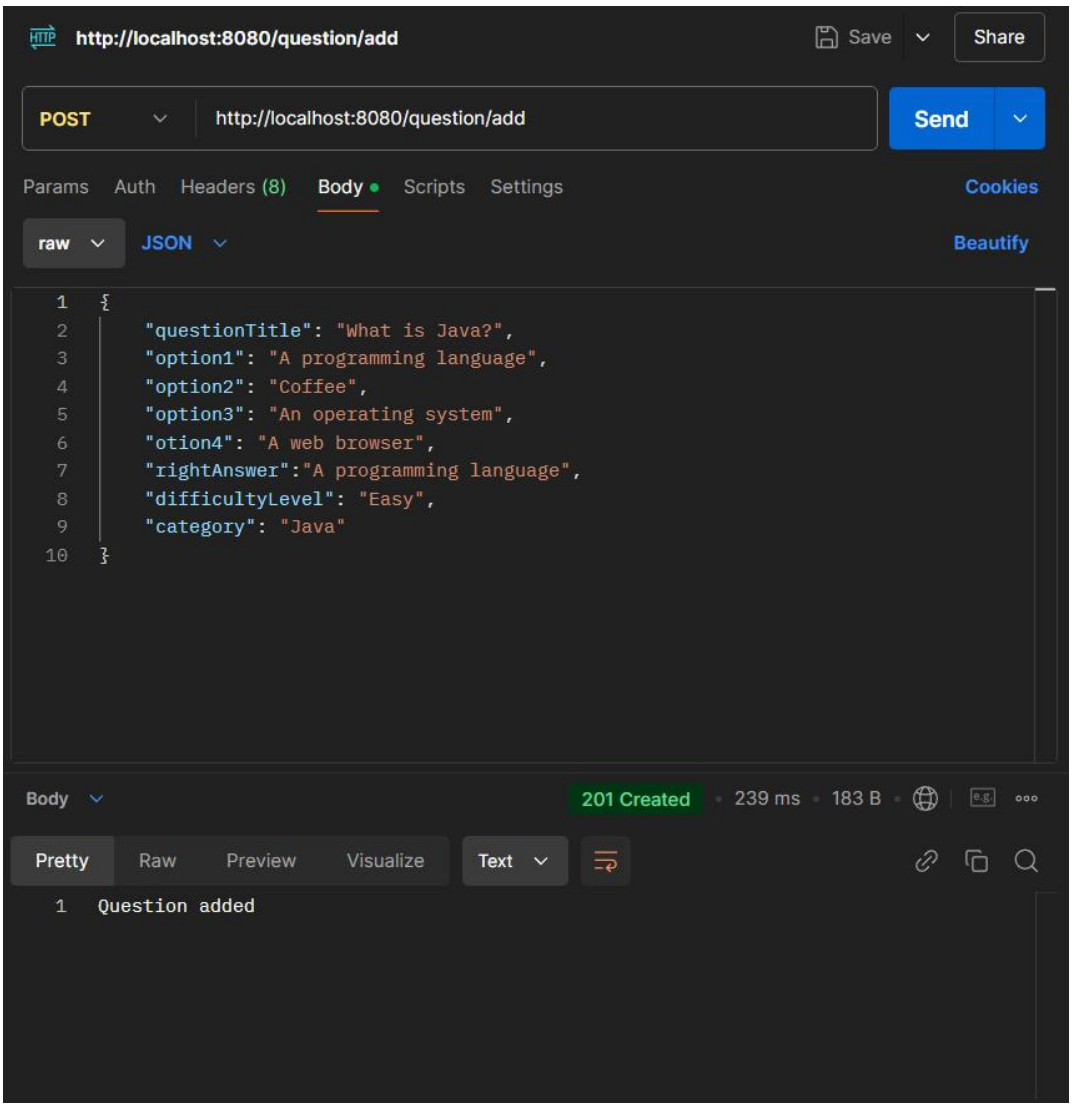


Рисунок 2.33 - Успішний результат POST

Для перевірки, чи питання було додано до списку питань у базі даних, перевіряємо безпосередньо в PostgreSQL (рис. 2.34).

Рисунок 2.34 - Додане питання у базі даних

2) Тестування отримання усіх питань (рис. 2.35).

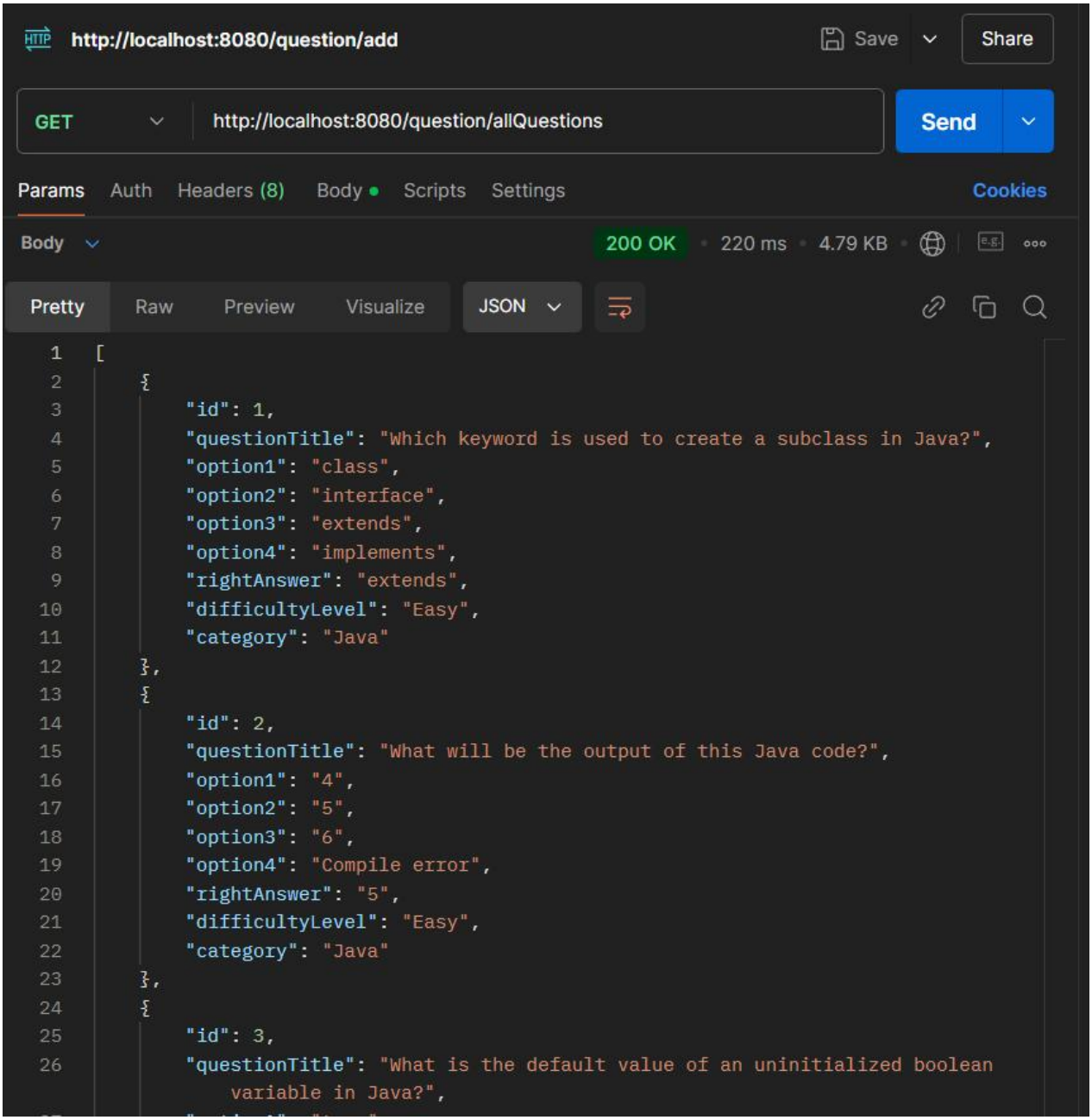


Рисунок 2.35 - Отримання усіх питань

3) Тестування отримання питань за категорією (рис. 2.36).

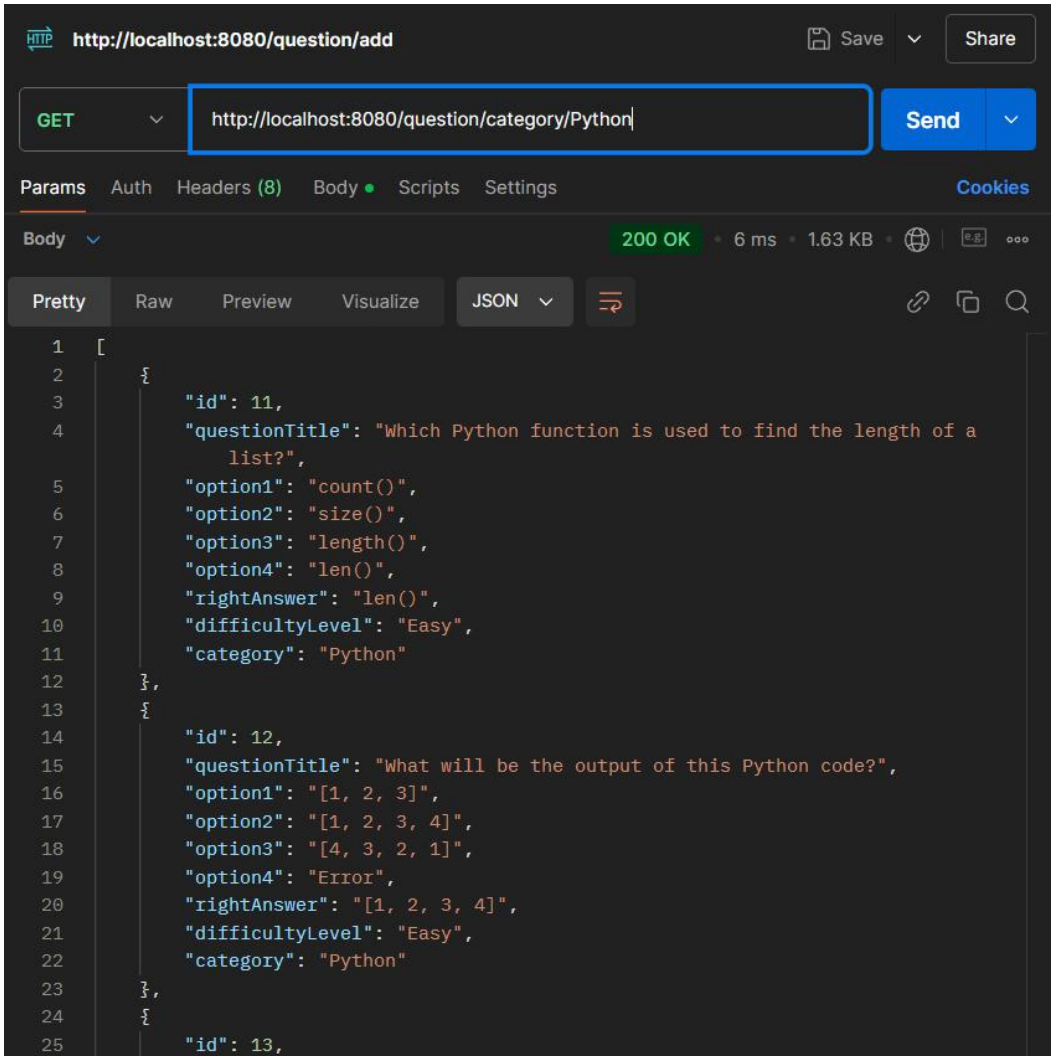


Рисунок 2.36 - Отримання питань за категорією

4) Тестування видалення питання (рис. 2.37).

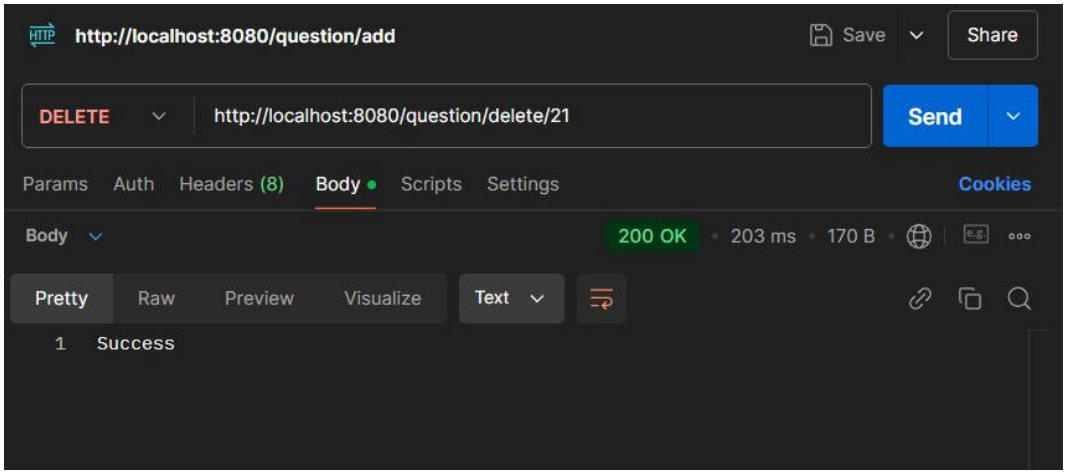


Рисунок 2.37 - Видалення питання

Тестування QuizzerController:

1) Тестування функції створення тесту (2.38).

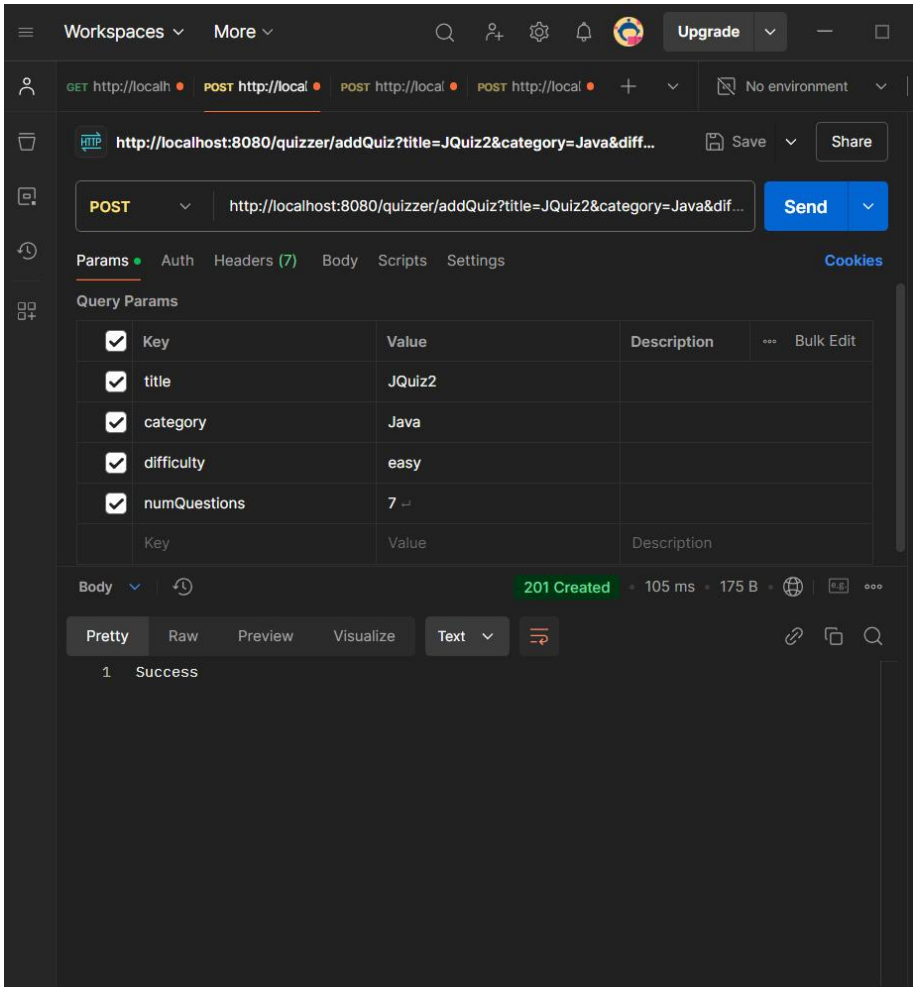
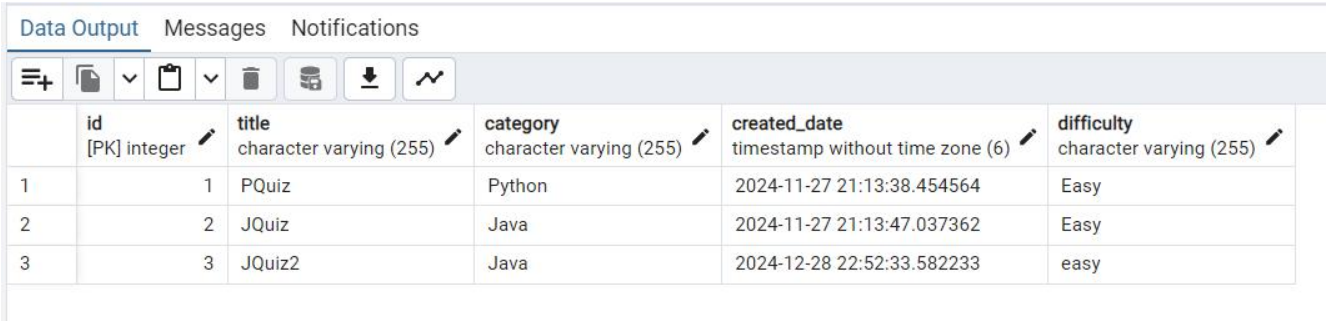


Рисунок 2.38 - Додавання опитування

Перевірка наявності опитування у базі даних (рис. 2.39.).



	id [PK] integer	title character varying (255)	category character varying (255)	created_date timestamp without time zone (6)	difficulty character varying (255)
1	1	PQuiz	Python	2024-11-27 21:13:38.454564	Easy
2	2	JQuiz	Java	2024-11-27 21:13:47.037362	Easy
3	3	JQuiz2	Java	2024-12-28 22:52:33.582233	easy

Рисунок 2.39 - Наявність створеного опитування у базі даних

2) Тестування отримання тесту (2.40).

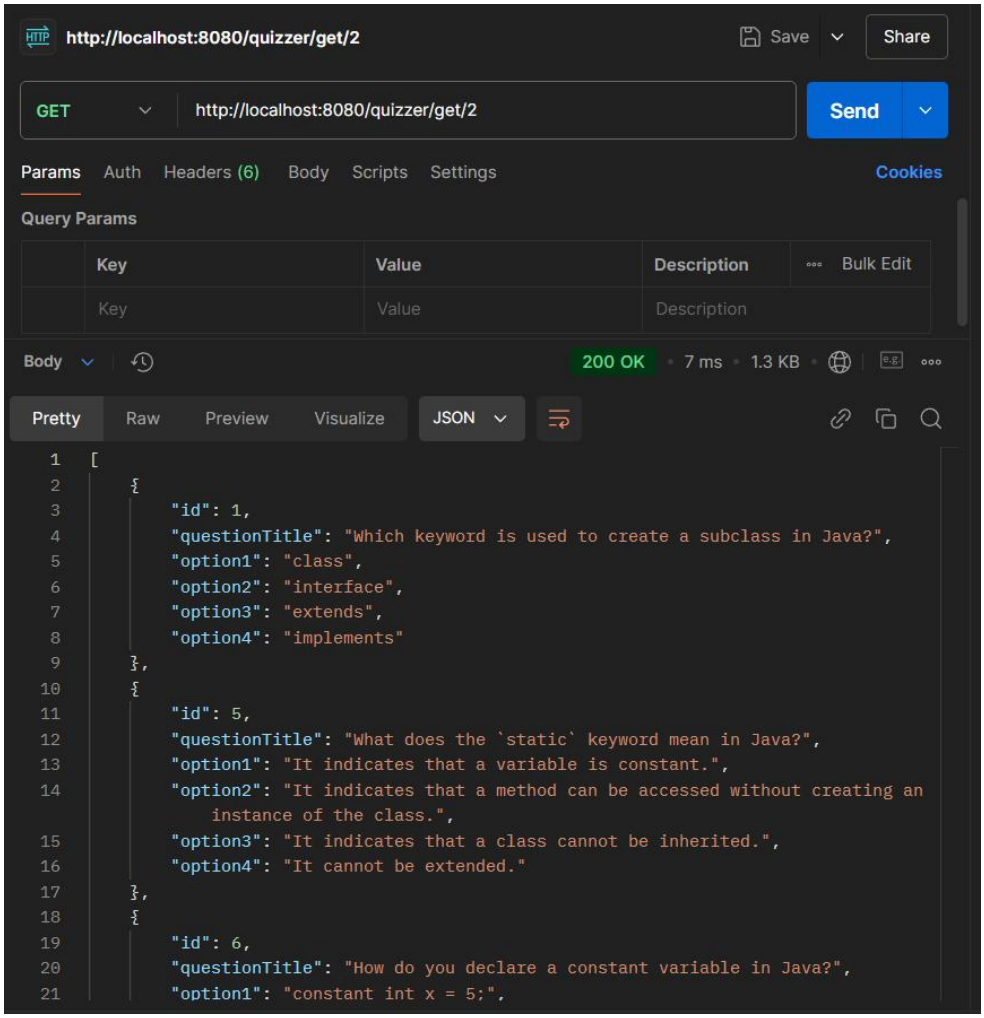


Рисунок 2.40 - Отримання тесту

3) Тестування видалення тесту (2.41).

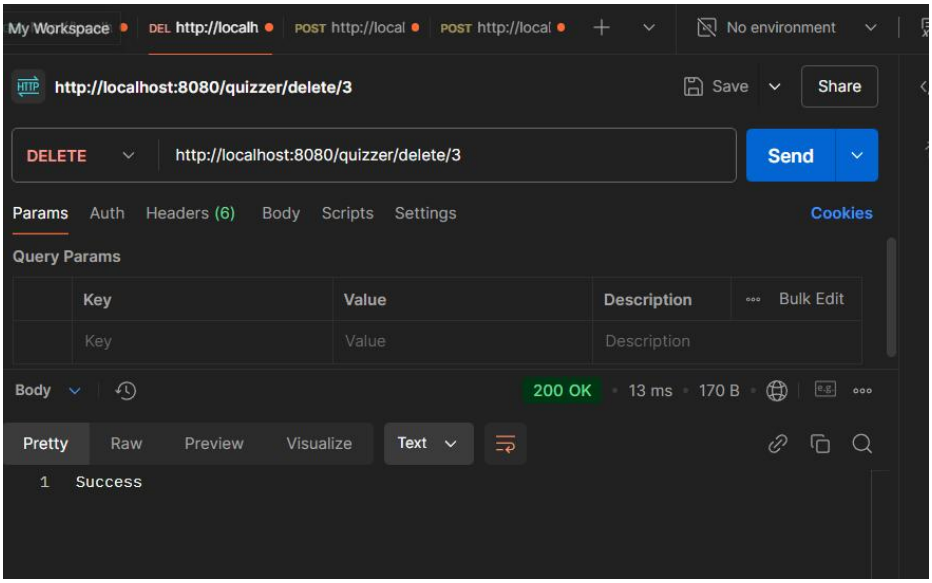


Рисунок 2.41 - Видалення тесту

4) Тестування отримання дати тесту (рис. 2.42).

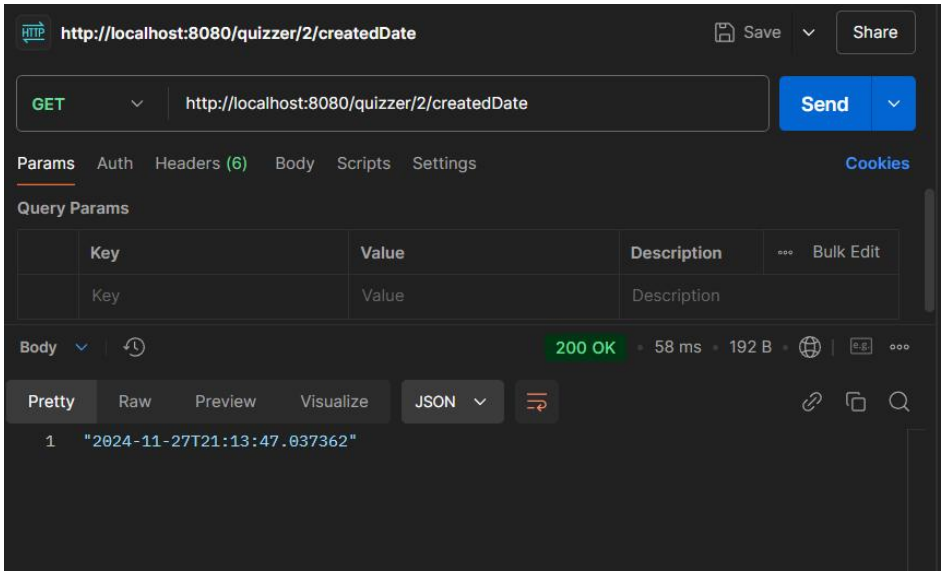


Рисунок 2.42 - Отримання дати тесту

5) Тестування оновлення питань в існуючому опитуванні (2.43).

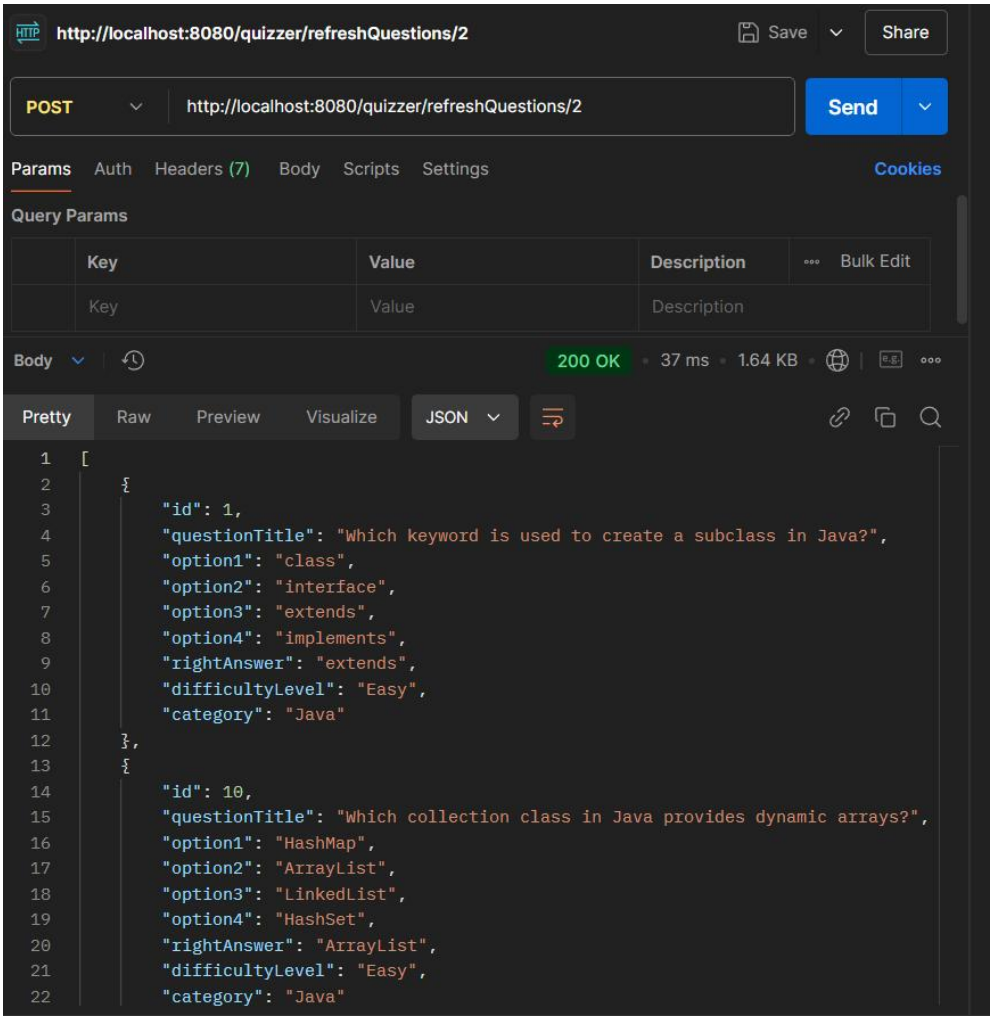


Рисунок 2.43 - Оновлення питань тесту

6) Тестування виклику списку усіх опитувань (2.44).

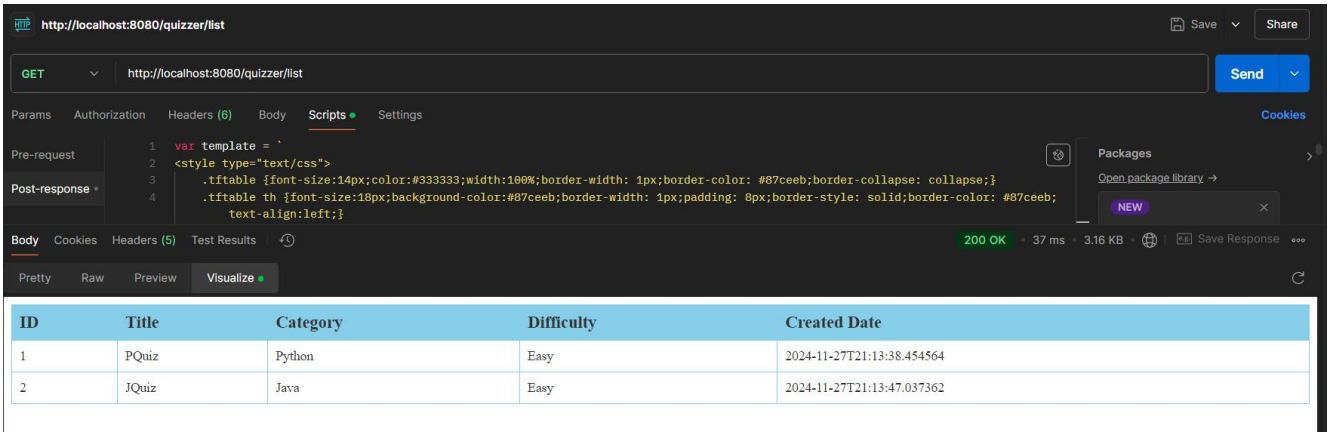


Рисунок 2.44 - Список опитувань тесту

7) Тестування проходження тесту (рис. 2.45).

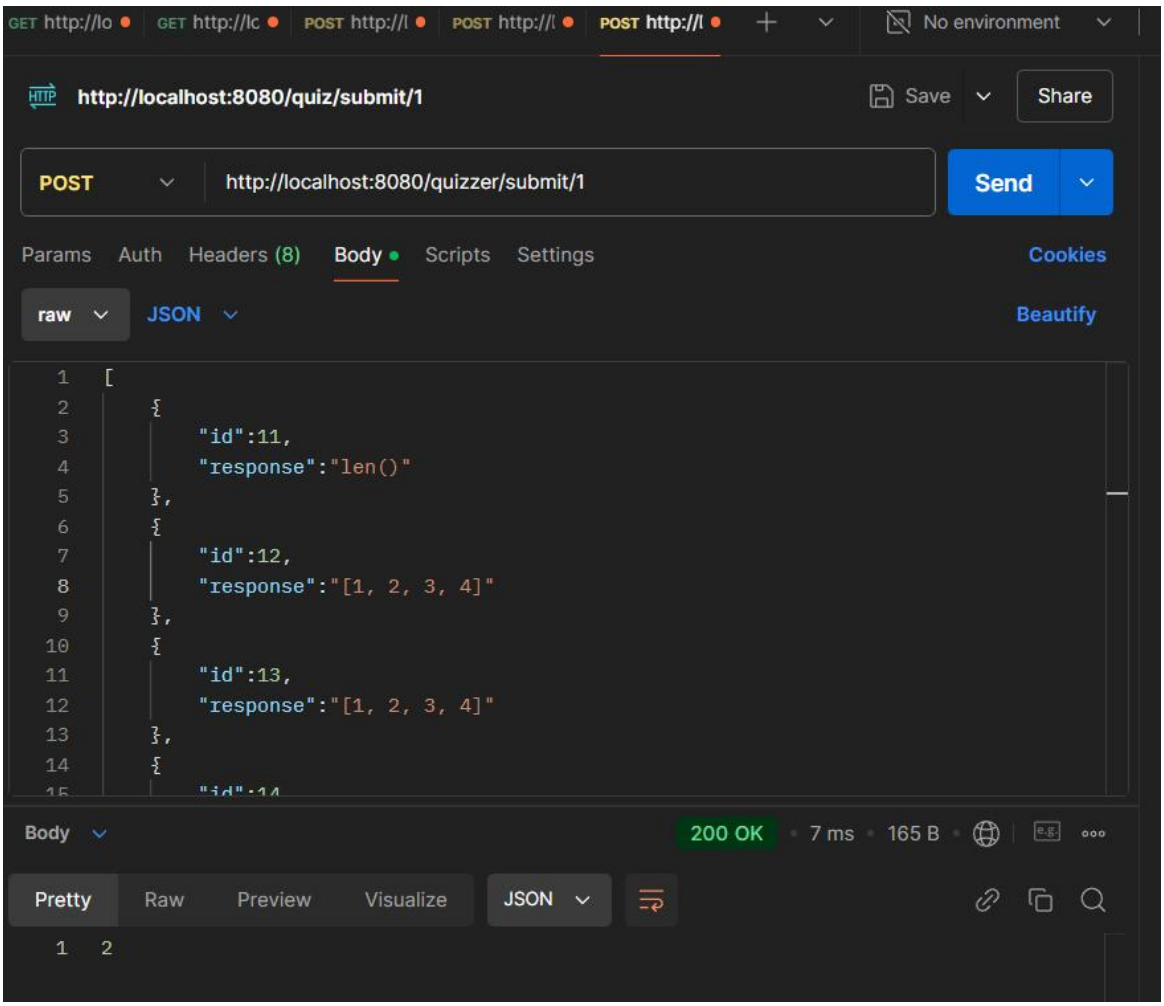


Рисунок 2.45 - Проходження тесту

2.6 Висновок до 2 розділу

У другому розділі проведено розробку монолітного додатку Quizzer на основі технологій Java, Spring Boot, і PostgreSQL. Ця частина роботи висвітлює всі аспекти створення монолітного програмного забезпечення, починаючи від обґрунтування вибору архітектури, технологій, і закінчуючи реалізацією функціональних модулів.

Монолітна архітектура була обрана через її простоту, високу продуктивність на початкових етапах розробки та легкість у впровадженні, що дозволило швидко створити робочий продукт для тестування концепцій. Додаток Quizzer є веб-орієнтованим, що дозволяє користувачам створювати та проходити тести, а також оцінювати результати.

При реалізації додатку:

- Використано Spring Boot 3 для автоматичної конфігурації, інтеграції з базою даних, створення RESTful API, що значно скоротило час розробки;
- PostgreSQL забезпечила зберігання і обробку даних, а також стабільну роботу з великими обсягами інформації;
- Додаток структурований з урахуванням принципів SOLID, що гарантує легкість у підтримці та подальшій модернізації.

На практиці було розроблено REST-контролери, сервіси для обробки бізнес-логіки, а також DAO-рівень для взаємодії з базою даних. Усі компоненти реалізовані в єдиній кодовій базі, що є характерною ознакою монолітної архітектури.

У підсумку, розробка монолітного додатку дала можливість глибше зрозуміти переваги та недоліки цього архітектурного підходу, а також забезпечила базу для подальшого переходу до мікросервісної архітектури, що буде розглянуто у наступному розділі.

3 ПЕРЕХІД ДО МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1 Підготовка до розбиття додатку на мікросервіси

Розбиття монолітного додатку на окремі сервіси потребує підготовчих дій. Підготовчі дії спрямовані на аналіз існуючої системи, визначення основних компонентів, які можуть бути розгорнуті в окремі сервіси зі створенням інфраструктури для їхньої взаємодії (рис. 3.1).

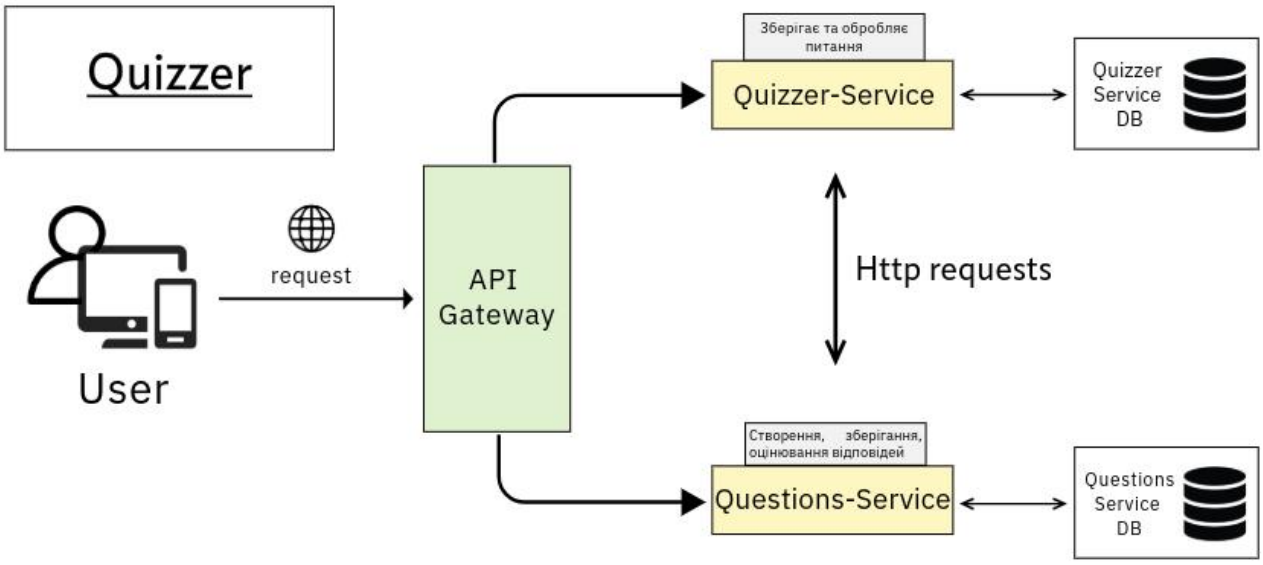


Рисунок 3.1 - Схема мікросервісного додатку Quizzer

Першим кроком є детальний аналіз існуючого монолітного додатку, щоб зрозуміти його структуру та взаємозв’язки між модулями:

а) Аналіз монолітної архітектури:

1) Визначення логіки:

- Аналізуються основні функціональні блоки додатку.

2) Визначення залежностей:

- Аналізуються місця тісного зв’язку між різними частинами додатку, які потрібно мінізувати під час розбиття.

3) Визначення зон відповідальності:

- Аналіз функцій, які можуть бути реалізовані окремо.

b) Виділення потенційних мікросервісів:

1) Quizzer Service:

- Управління опитуваннями (створення, отримання, оцінка результатів).

2) Question Service:

- Робота з питаннями (зберігання, пошук, редагування).

c) Декомпозиція бази даних:

1) Ідентифікація таблиць:

- Визначення таблиць, які належать до кожного потенційного сервісу.

2) Зменшення зв’язків:

- Видалення прямих зв’язків між таблицями різних сервісів. Замість цього використовується обмін даними через API.

3) Дублювання даних.

- У разі потреби деякі дані можуть дублюватися між сервісами для зменшення залежностей.

d) Підготовка інфраструктури:

1) Вибір інструментів:

- Spring Boot: Для створення окремих сервісів.
- Spring Cloud: Для управління мікросервісами, включаючи реєстрацію, маршрутизацію та балансування.
- Eureka: Як сервіс реєстрації.
- Spring Cloud Gateway: Як API Gateway для маршрутизації запитів.

2) Налаштування середовища:

- Визначення портів для кожного сервісу.

e) Вибір стратегії міграції:

- Паралельний запуск: Моноліт залишається в роботі, доки всі частини не будуть перенесені у мікросервіси.
- Ітеративне розбиття: Виділення частин додатку поетапно, починаючи з найменш залежних компонентів.
- Перенаправлення запитів через Gateway: Клієнтські запити поступово перенаправляються з моноліту до нових мікросервісів через API Gateway.

f) Результати підготовки:

- Проведено аналіз монолітного додатку та визначено основні модулі для виділення у мікросервіси.
- Розроблено інфраструктуру для підтримки мікросервісної архітектури.
- Налаштовано інструменти для реєстрації, маршрутизації та тестування сервісів.
- Визначено стратегію поступового переходу від моноліту до мікросервісів.

3.2 Розбиття додатку на мікросервіси

Перехід до мікросервісної архітектури був реалізований через виділення ключових функціональних модулів системи в окремі сервіси: registry-service, api-

gateway, questions-service та quizzer-service. Кожен із цих сервісів виконує свою чітко визначену роль.

Було вирішено виділити наступні сервіси:

1) Registry Service (Eureka) — централізована служба реєстрації для динамічного виявлення мікросервісів.
2) API Gateway — єдина точка входу для клієнтських запитів, яка відповідає за маршрутизацію та авторизацію.
3) Questions Service — сервіс для роботи з питаннями, включаючи зберігання, пошук і управління.
4) Quizzer Service — сервіс для управління опитуваннями, включаючи створення, оцінювання результатів та зберігання інформації про опитування.

1. Налаштування Registry Service

Eureka Server був реалізований як центральний реєстратор сервісів. Кожен сервіс, включаючи API Gateway, Questions Service та Quizzer Service, реєструється в Eureka. Це дозволяє сервісам динамічно знаходити один одного без необхідності вручну прописувати їхні адреси.

Налаштування Eureka для Registry Service зображенно на рисунках 3.2 - 3.4.

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

Рисунок 3.2 - Додавання залежності Eureka у файлі pom.xml класу Registry Service

```
spring.application.name=registry-service
server.port=8761
eureka.instance.hostname=localhost
eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
```

Рисунок 3.3 - Реєстрування клієнту Eureka у application.properties для Registry Service

```
@EnableEurekaServer
@SpringBootApplication
public class RegistryServiceApplication {

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

}
```

Рисунок 3.4 - Анотація @EnableEurekaServer для активації сервера Eureka

2. Реалізація API Gateway

API Gateway працює як єдина точка входу для всіх клієнтських запитів. Його основна роль полягає у перенаправленні запитів до відповідних мікросервісів, зареєстрованих у Eureka.

Реалізація API Gateway зображенно на рисунках 3.5.-3.6.

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

Рисунок 3.5 - Додавання залежності Eureka у файлі pom.xml класу Registry Service

```
spring.application.name=api-gateway
server.port=8765

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lower-case-service-id=true
```

Рисунок 3.6 - Конфігурація для Spring Cloud Gateway

Завдяки цьому Gateway автоматично перенаправляє запити, наприклад:

- http://localhost:8765/questions-service/questions звертається до Questions Service.
- http://localhost:8765/quizzer-service/quizzes звертається до Quizzer Service.

3. Questions Service

Questions Service відповідає за роботу з питаннями. Сервіс має власну базу даних, яка містить таблицю questions. Основна функція сервісу — надавати список питань за певними критеріями, такими як категорія чи рівень складності.

Для передення Questions Service на модель мікросервісної архітектури було застосавано наступні дії:

1) Переміщення сервісу до окремого проєкту (рис 3.7 - 3.8).
2) Налаштування Eureka (рис 3.9).
3) Звертання до об’єктів питань через id (рис 3.10).

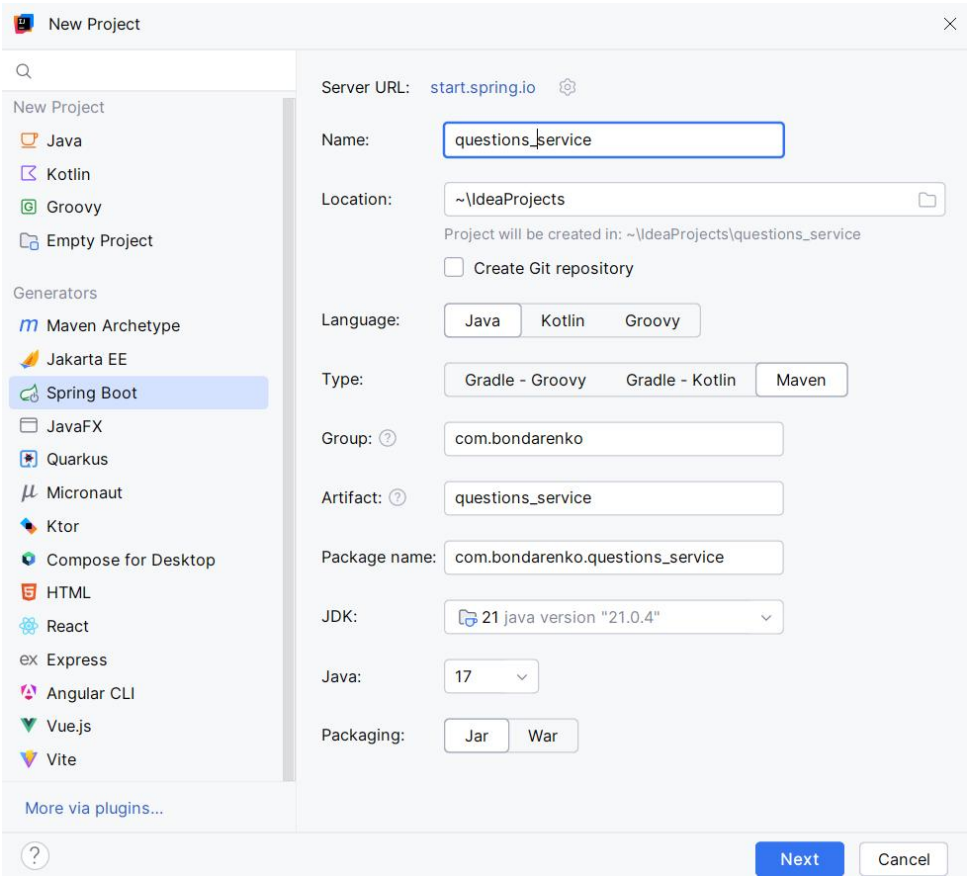


Рисунок 3.7 - Налаштування проєкту Questions Service

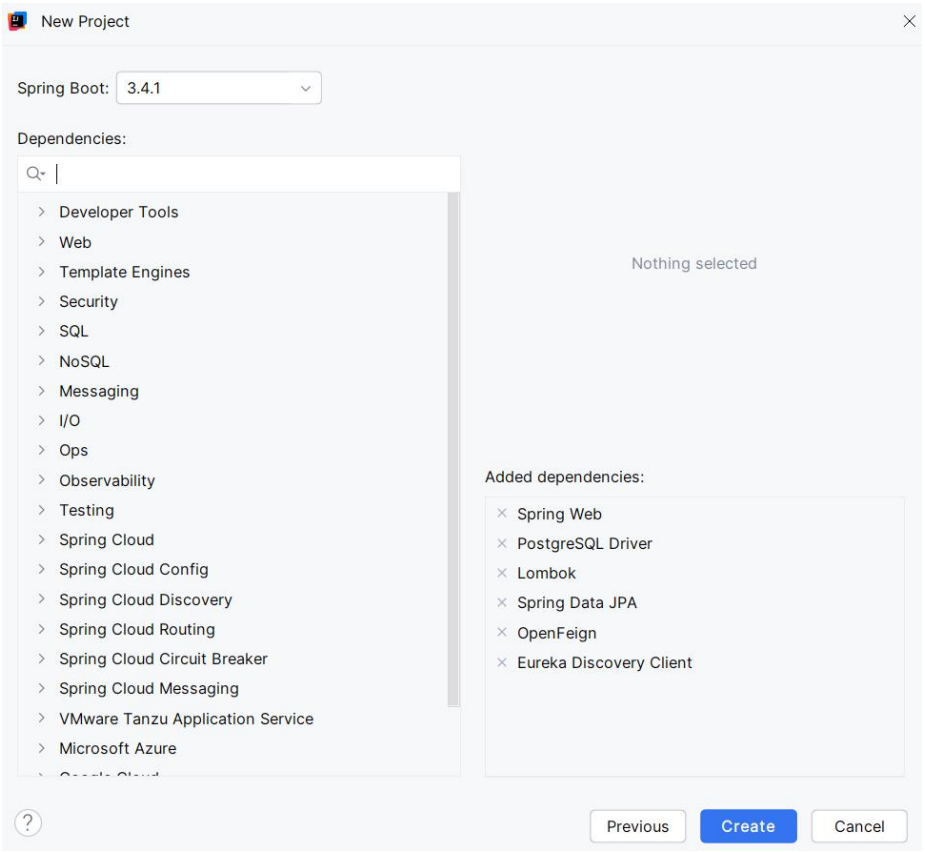


Рисунок 3.8 - Налаштування залежностей Questions Service

```
spring.application.name=questions-service
server.port=8090
spring.datasource.url=jdbc:postgresql://localhost:5432/questiondb
spring.datasource.username=postgres
spring.datasource.password=1111
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Рисунок 3.9 - Налаштування application.properties для Questions Service

```
public ResponseEntity<List<QuestionWrapper>> getQuestionsID(List<Integer> questionsID) {
    List<QuestionWrapper> wrappers = new ArrayList<>();
    List<Question> questions = new ArrayList<>();
    for (Integer id : questionsID) {
        questions.add(questionDao.findById(id).get());
    }

    for (Question question : questions) {
        QuestionWrapper wrapper = new QuestionWrapper();
        wrapper.setId(question.getId());
        wrapper.setQuestionTitle(question.getQuestionTitle());
        wrapper.setOption1(question.getOption1());
        wrapper.setOption2(question.getOption2());
        wrapper.setOption3(question.getOption3());
        wrapper.setOption4(question.getOption4());
        wrappers.add(wrapper);
    }

    return new ResponseEntity<>(wrappers, HttpStatus.OK);
}
```

Рисунок 3.10 - Метод для отримання питання за ідентифікаційним номером

4) Quizzer Service

Quizzer Service керує опитуваннями, зберігає інформацію про їх створення, обробляє відповіді та обчислює результати. Сервіс використовує Feign для взаємодії з Questions Service для отримання списку питань (рис 3.11).

```
14
15  @FeignClient("QUESTIONS-SERVICE")  2 usages
16  public interface QuizzerInterface {
17      @GetMapping("question/display")
18      public ResponseEntity<List<Integer>> getQuestions(@RequestParam String categoryName, @RequestParam String title,
19
20      @PostMapping("question/getQuestions")
21      public ResponseEntity<List<QuestionWrapper>> getQuestionsFromId(@RequestBody List<Integer> questionsID);
22
23      @PostMapping("question/getScore")
24      public ResponseEntity<Integer> getQuestionScore(@RequestBody List<Response> responses);
25  }
```

Рисунок 3.11 - Використання Feign для взаємодії з Questions Service

Контролер у Quizzer Service реалізує REST API для управління квізами (рис. 3.12).

```
@RequestMapping("quizzer")
public class QuizzerController {
    @Autowired
    QuizzerService quizzerService;


    @PostMapping("addQuiz")
    public ResponseEntity<String> addQuiz(@RequestBody QuizDTO quizDTO) {
//        QuizDTO quizDTO,String category, int numQ, String title, String difficulty
        return quizzerService.createQuiz(quizDTO, quizDTO.getCategoryName(),quizDTO.getNumQuestions(),quizDTO.getTitle()
    }

    @DeleteMapping("delete/{id}")
    public ResponseEntity<String> deleteQuiz(@PathVariable Integer id) { return quizzerService.deleteQuiz(id); }

    @GetMapping("get/{id}")
    public ResponseEntity<List<QuestionWrapper>> fetchQuizQuestions(@PathVariable Integer id) {
        return quizzerService.getQuizQuestions(id);
    }

    @GetMapping("{id}/createdDate")
    public ResponseEntity<LocalDateTime> getQuizCreatedDate(@PathVariable Integer id) {
        LocalDateTime createdDate = quizzerService.getCreatedDate(id);
        return ResponseEntity.ok(createdDate);
```

Рисунок 3.12 - REST API для управління квізами

Клас QuizDTO (Data Transfer Object) використовується для передачі даних при створенні або обробці квізів. Він містить основну інформацію про квіз, яка потрібна для його створення чи взаємодії між сервісами (рис 3.13).

```
@Data 4 usages
public class QuizDTO {
    String categoryName;
    Integer numQuestions;
    String title;
    String difficulty;
```

Рисунок 3.13. - Клас QuizDTO

Таким чином, QuizDTO виступає як контейнер для даних, що мінімізує залежність між компонентами системи.

3.3 Тестування та перевірка роботи мікросервісного додатку

Тестування та перевірка роботи мікросервісного додатку із використанням API Gateway дозволяє переконатися, що всі сервіси взаємодіють коректно, а запити клієнтів маршрутизуються до відповідних мікросервісів. API Gateway є єдиною точкою входу, через яку проходять усі клієнтські запити, що забезпечує централізований контроль над доступом і маршрутизацією.

Процес тестування:

1) Додавання опитування (рис 3.14).

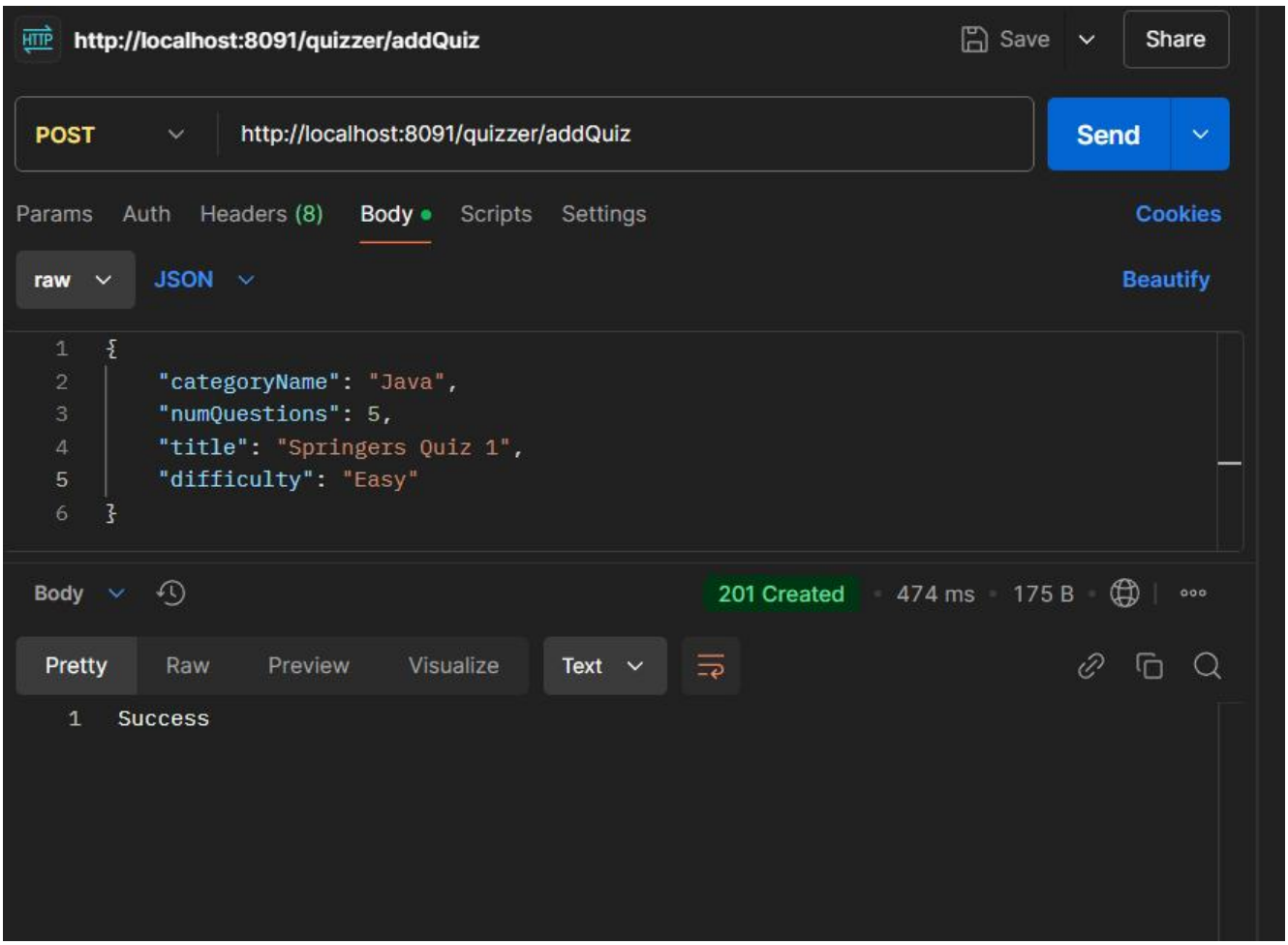


Рисунок 3.14 - Запит на додавання опитування

2) Перевірка чи опитування було додано (рис. 3.15).

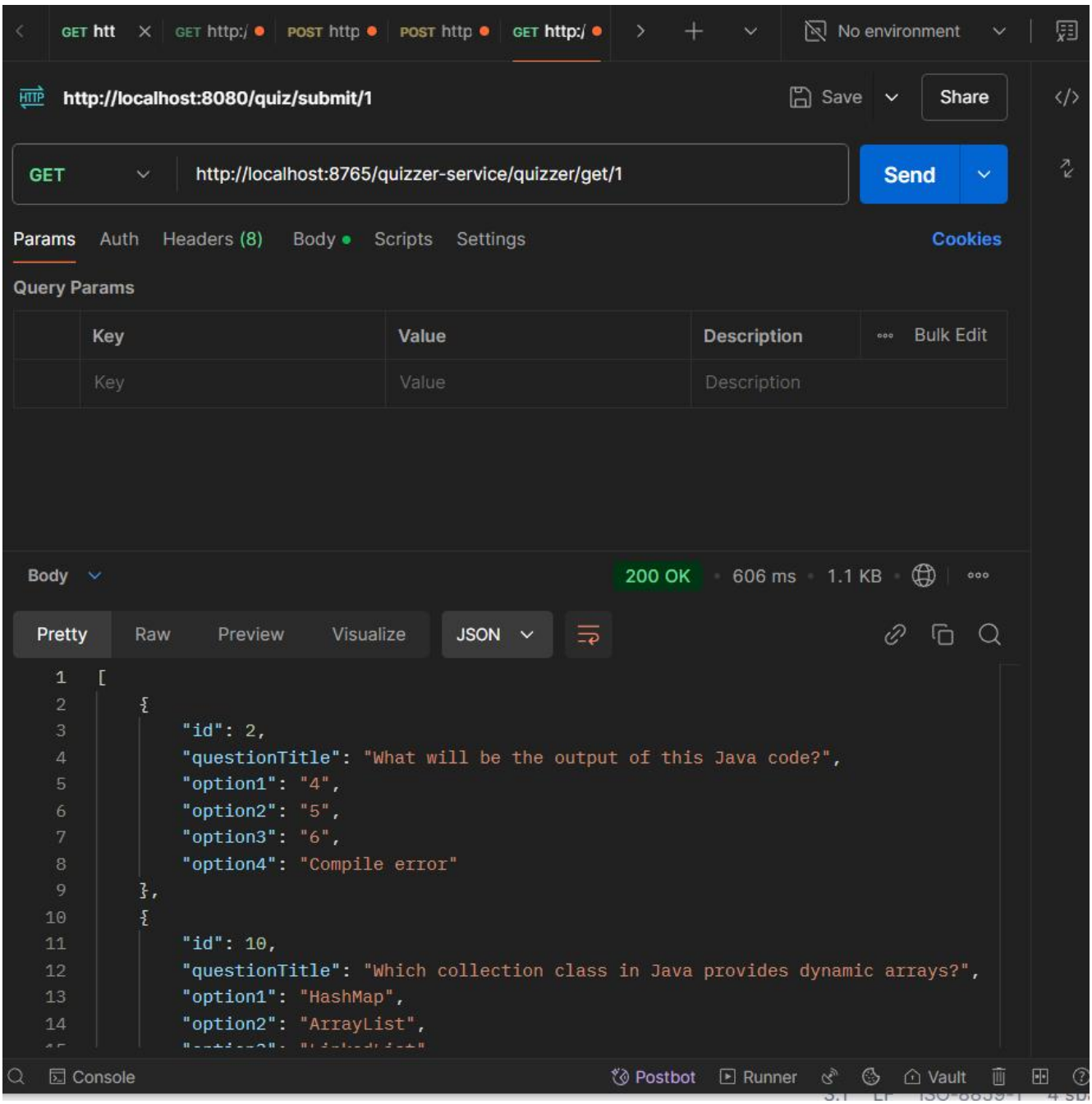


Рисунок 3.15 - Запит на отримання опитування

3) Перевірка чи усі питання було передано (рис. 3.16).

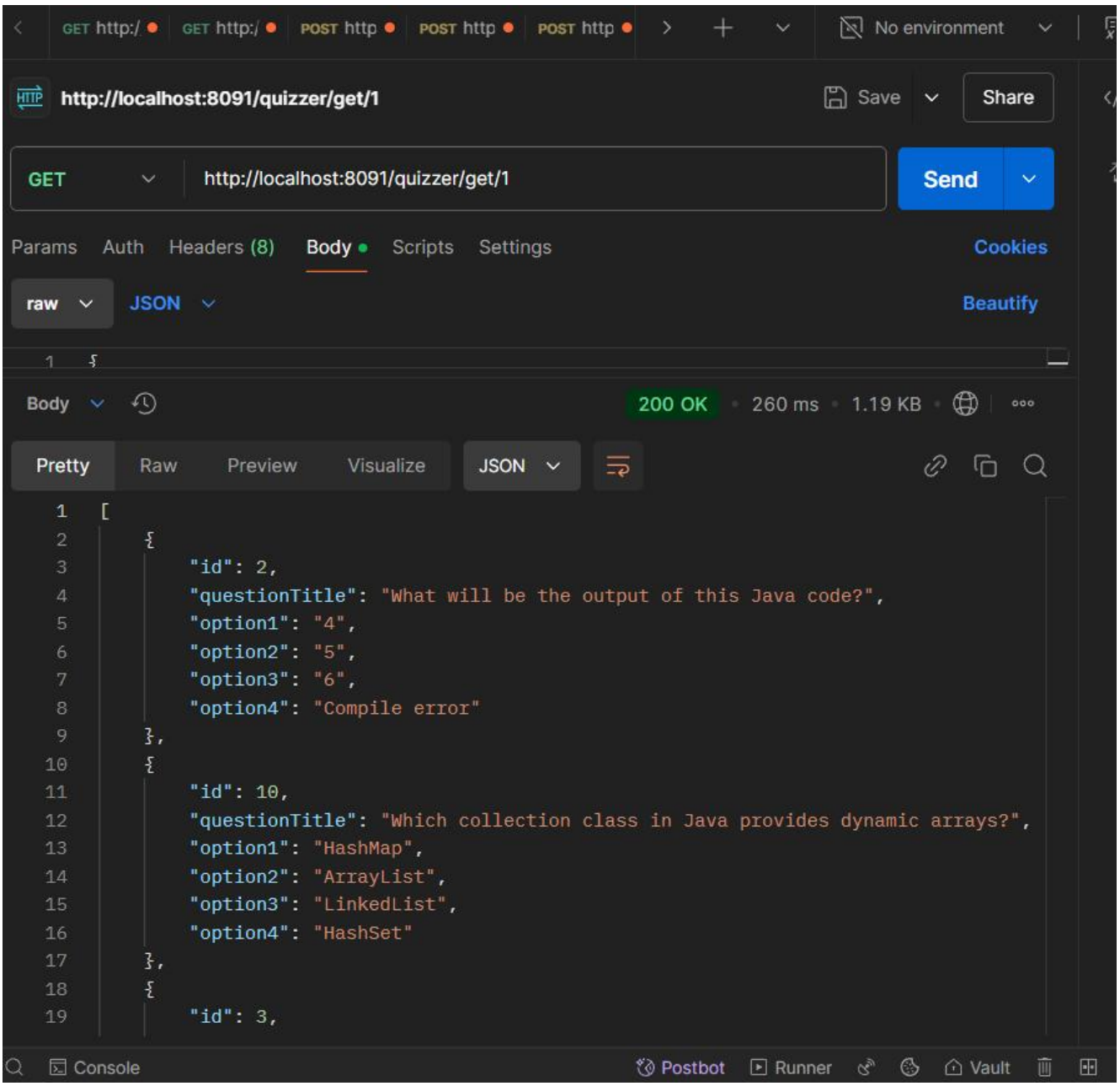


Рисунок 3.16 - Запит на отримання опитування

4) Перевірка процесу тестування (надання відповідей на питання) (рис.3.17).

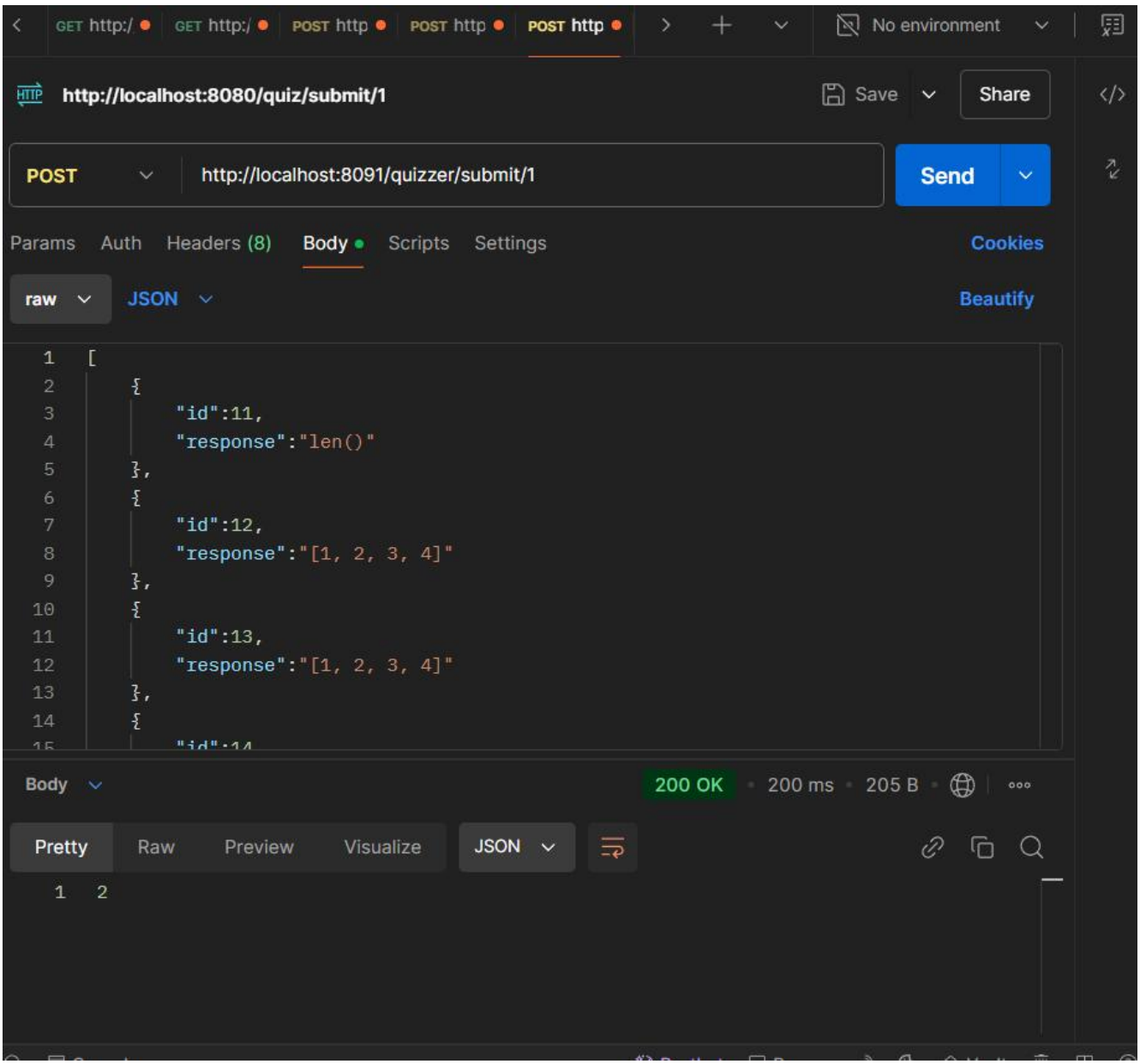


Рисунок 3.17 - Надання відповідей на питання

3.4 Висновок до розділу 3

У третьому розділі було проведено дослідження мікросервісної архітектури на базі Java та Spring Boot, що дозволило розробити та протестувати програмний додаток Quizzer у двох версіях — монолітній та мікросервісній. На основі виконаних досліджень можна зробити наступні висновки:

1. Розробка додатку:

- Було створено дві версії додатку Quizzer: монолітну та мікросервісну.
- Для реалізації використано стек технологій Java, Spring Boot та PostgreSQL.

2. Порівняння архітектур:

- Монолітна версія додатку показала простоту розробки та високу продуктивність у невеликих системах. Однак вона виявила обмеження у масштабованості та стійкості до збоїв.
- Мікросервісна версія додатку дозволила продемонструвати незалежність сервісів, що сприяє підвищенню масштабованості, стійкості до помилок та гнучкості в оновленнях.

3. Рекомендації щодо вибору архітектури:

- Для невеликих проєктів з чітко визначеними функціями та обмеженими ресурсами доцільно використовувати монолітну архітектуру.
- Для великих, динамічних систем із потребою у частих оновленнях та високій масштабованості перевагу слід надавати мікросервісній архітектурі.

Проведене дослідження підтвердило, що вибір архітектури значною мірою залежить від потреб проєкту, його складності та довгострокових цілей. Мікросервісна архітектура має значні переваги для складних і масштабованих систем, тоді як монолітна архітектура лишається ефективним вибором для простих і малих проєктів.

ВИСНОВКИ

У висновках до магістерської роботи підведено підсумки проведених досліджень, розробки та тестування програмного додатку Quizzer, що демонструє переваги і недоліки монолітної та мікросервісної архітектур.

1. Дослідження архітектур:

- Було проведено глибокий аналіз монолітної та мікросервісної архітектур, включаючи їх переваги, недоліки та сфери застосування.
- Здійснено розробку двох версій додатку Quizzer — монолітної та мікросервісної, що дало змогу порівняти їх ефективність у реальних умовах.

2. Результати тестування:

- Тестування додатків показало, що монолітна архітектура є більш простою у реалізації та ефективною для невеликих систем із мінімальними вимогами до масштабування.
- Мікросервісна архітектура виявила свою перевагу у масштабованості, стійкості до помилок та адаптивності до змін, що робить її оптимальним вибором для великих та динамічних проєктів.

3. Рекомендації щодо вибору архітектури:

- Монолітна архітектура доцільна для невеликих проєктів із обмеженим бюджетом та стабільними вимогами.
- Мікросервісна архітектура рекомендується для складних систем із високими вимогами до масштабованості, стійкості та частих оновлень.

4. Вплив використаних технологій:

- Використання Java, Spring Boot та PostgreSQL забезпечило надійну основу для реалізації як монолітної, так і мікросервісної архітектур.

- Технологічний стек дозволив створити ефективний додаток із високими показниками продуктивності та надійності.

Загалом, результати магістерської роботи підтвердили, що вибір архітектурного підходу повинен базуватися на специфічних вимогах проєкту, його масштабі та довгострокових цілях. Мікросервісна архітектура має значний потенціал для забезпечення гнучкості та стійкості сучасних систем, тоді як монолітна архітектура лишається ефективним рішенням для простих і малих проєктів.

Список використаної літератури

1. Монолітна архітектура. [Електронний ресурс] URL: https://www.techtarget.com/whatis/definition/monolithic-architecture
2. Анісімов В. Г., Кунанець Н.Е.Перехід від монолітної до мікросервісної архітектури: методологія та досвід впровадженн. Науковий журнал "Комп'ютерно-інтегровані технології: освіта, наука, виробництво". Луцьк,2024. Випуск №55.
3. MartinFowler, Microservices. [Електронний ресурс] URL: https://martinfowler.com/articles/microservices.html .
4. Про мікросервісну архітектуру. [Електронний ресурс] URL: https://foxminded.ua/mikroservisna-arkhitektura/
5. Pattern: Microservice Architecture. [Електронний ресурс] URL: https://microservices.io/patterns/microservices.html
6. Baeldung. Saga Pattern in Microservices. [Електронний ресурс] URL: https://www.baeldung.com/cs/saga-pattern-microservices
7. Microsoft. Anti-Corruption Layer Pattern [Електронний ресурс] URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer
8. Wikipedia. Strangler fig. [Електронний ресурс]. URL: https://en.wikipedia.org/wiki/Strangler_fig
9. Ticet, C. (2020). The Strangler Fig Pattern is what you need to migrate monolithic application with legacy code to microservices [Електронний ресурс]. Medium. URL: https://medium.com/@sylvain.tiset/the-strangler-fig-pattern-is-what-you-need-to-migrate-monolithic-application-with-legacy-code-to-ec24cf7168eb
10. Geeksforgeeks. Spring Boot Tutorial. [Електронний ресурс]. URL: https://www.geeksforgeeks.org/spring-boot/
11. Oracle. Java. [Електронний ресурс]. URL: https://www.oracle.com/java/
12. PostgreSQL [Електронний ресурс]. URL: https://www.postgresql.org
13. Spring Initializr [Електронний ресурс]. URL: https://start.spring.io
14. Apache Maven [Електронний ресурс]. URL: https://maven.apache.org

15. David Demir. What is Postman (A Tutorial for Beginners). Apidog. URL: https://apidog.com/blog/what-is-postman/

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

ДИПЛОМНА РОБОТА
на ступінь вищої освіти магістр
із спеціальності 122 Комп'ютерні науки

МІКРОСЕРВІСНА АРХІТЕКТУРА НА БАЗІ JAVA ТА SPRING BOOT

Виконав: студент 6 курсу, групи КНДМ-61
Бондаренко Андрій Олександрович
Керівник: Доцент кафедри, доктор філософії
Кравчук П.О.

Київ - 2024

Слайд 1

1

ЗАГАЛЬНА ХАРАКТЕРИСТИКА ДИПЛОМНОЇ РОБОТИ

Тема	ДОСЛІДЖЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ НА БАЗІ JAVA ТА SPRING BOOT
Мета дослідження	Дослідження мікросервісних архітектурних характеристик у контексті ефективності, масштабованості, гнучкості та стійкості до помилок
Об'єкт дослідження	Дві версії програми Quizzer (опитування), які представлені у монолітному і мікросервісному виконанні.
Предмет дослідження	Мікросервісна архітектура, переваги і недоліки мікросервісної архітектури у порівнянні з монолітною.

Слайд 2

2

Архітектури: моноліт і мікросервіси

Монолітна архітектура

Всі компоненти програми зібрані в один єдиний блок. Всі функції додатку (від обробки запитів до бази даних) взаємодіють в межах одного коду та часто знаходяться на одному сервері.

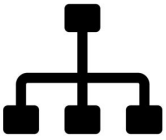
Мікросервісна архітектура

Додаток складається з кількох незалежних мікросервісів, кожен з яких виконує одну функцію і може працювати автономно. Мікросервіси взаємодіють між собою через API.

Слайд 3

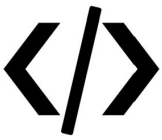
3

Головні відмінності: моноліт і мікросервіси



- **Структура:**
- Моноліт — все в одному блоці.
- Мікросервіси — окремі компоненти з автономними базами даних.

- **Масштабованість:**
- Моноліт — масштабування цілої системи.
- Мікросервіси — кожен мікросервіс масштабується окремо.



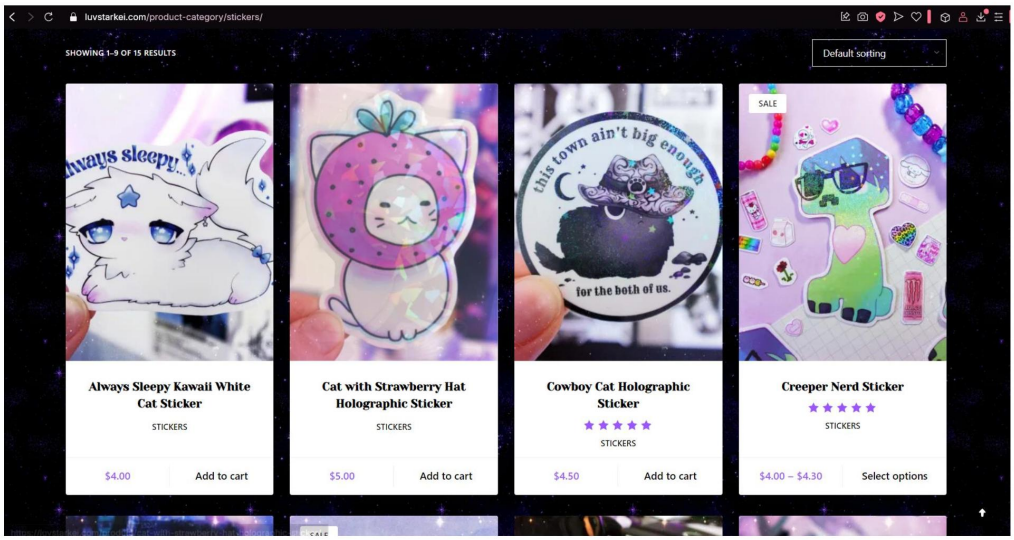
- **Розробка та розгортання:**
- Моноліт — одне розгортання для всього додатку.
- Мікросервіси — кожен сервіс розгортається та оновлюється окремо.

- **Надійність:**
- Моноліт — збій в одній частині може вплинути на всю систему.
- Мікросервіси — збій одного мікросервісу не впливає на інші

Слайд 4

4

Приклади використання: моноліт і мікросервіси

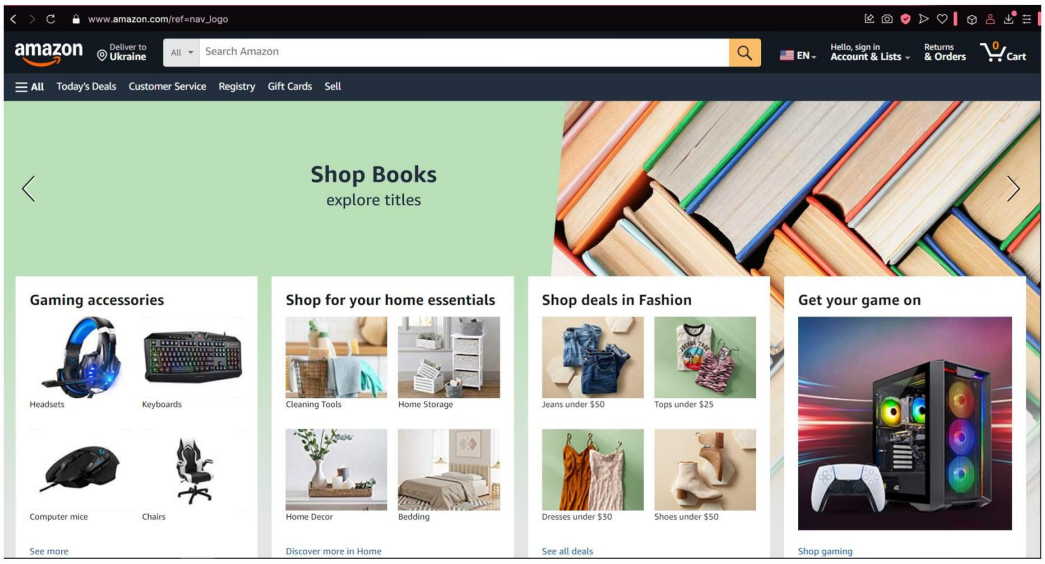


Простий веб-додаток з продажу стікерів для маленького бізнесу.

Слайд 5

5

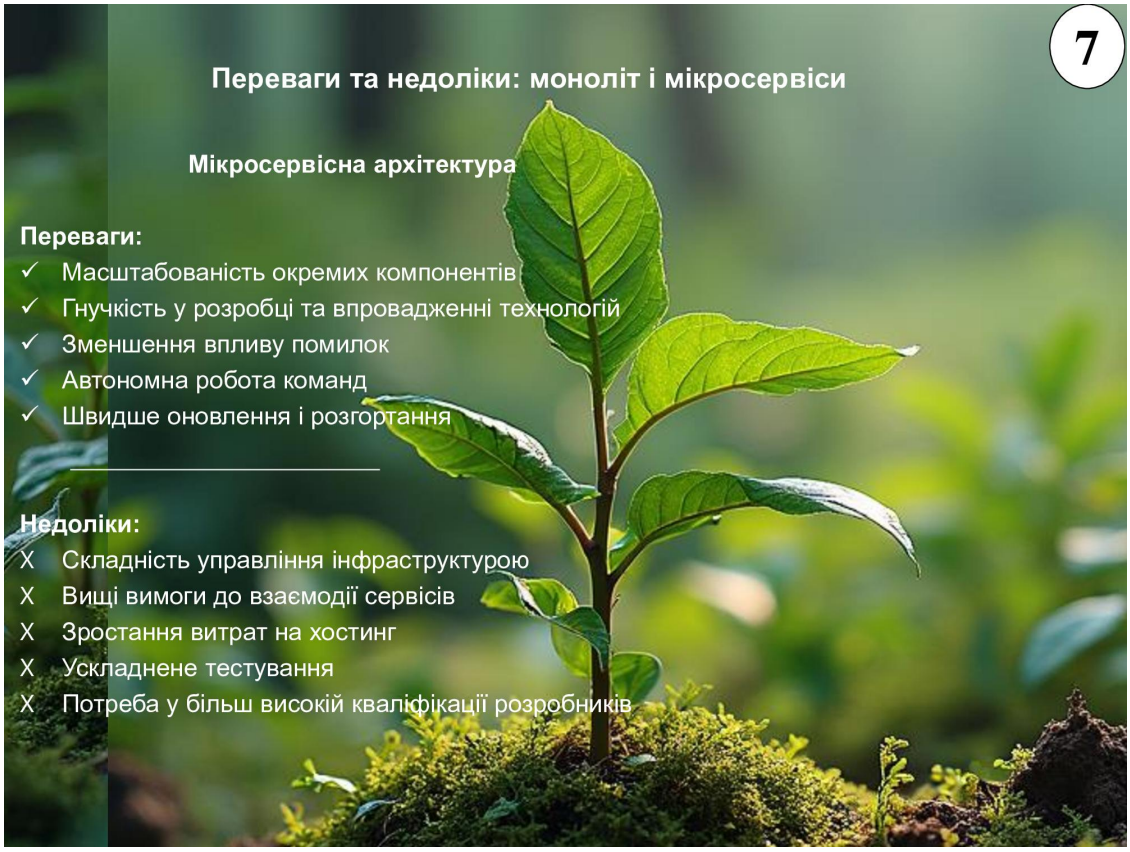
Приклади використання: моноліт і мікросервіси



Великі платформи з високим навантаженням, наприклад, Netflix, Amazon, Uber.

Слайд 6

Слайд 7



Слайд 8

Розробка додатку Quizzer

Використані технології:

- **Мова програмування: Java.**
- **Фреймворк: Spring Boot.**
- **База даних: PostgreSQL.**

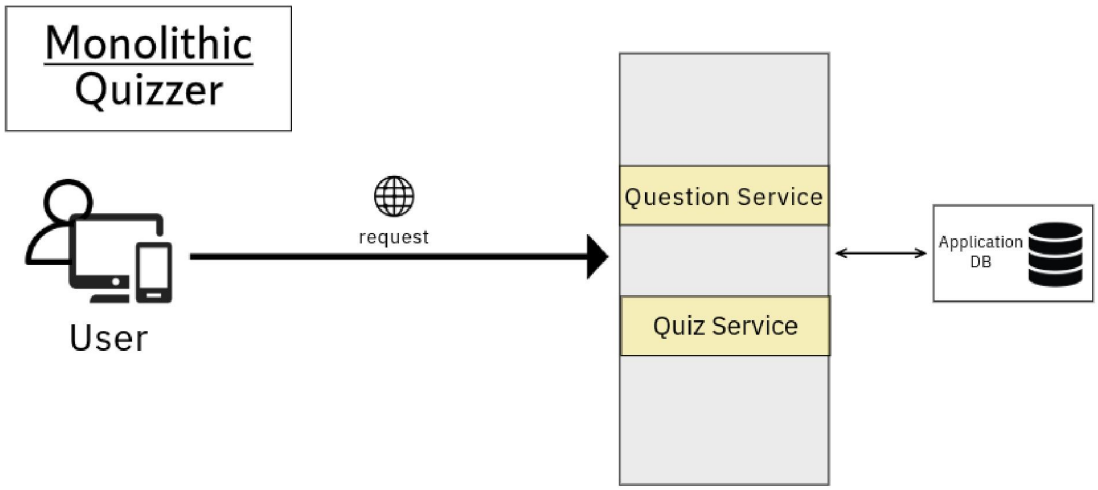
Мета Quizzer:

- **Створення тестів для перевірки теоретичних знань з програмування.**
- **Реалізація як у монолітній, так і в мікросервісній архітектурі.**
- **Тестування програми та аналіз результатів для порівняння архітектур**

Слайд 9

Розробка додатку Quizzer

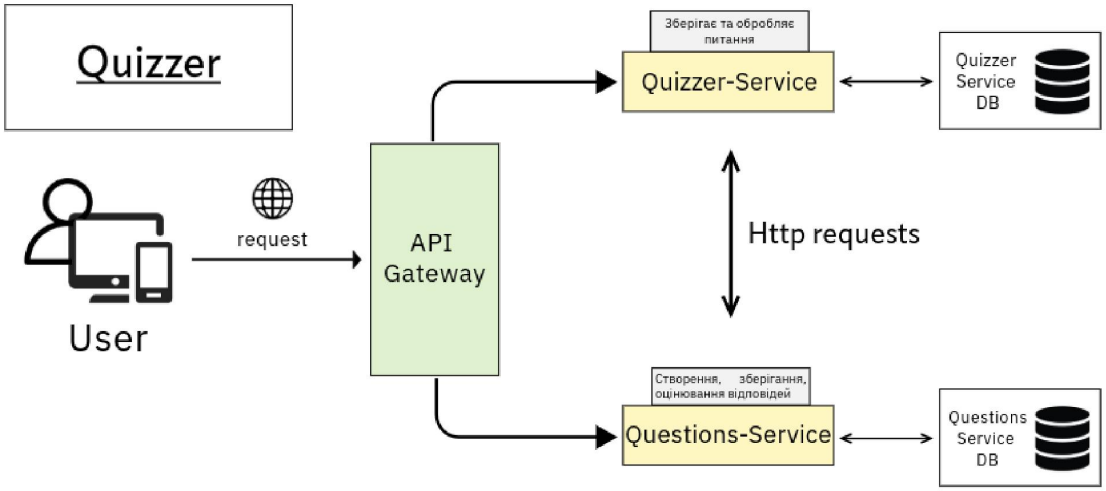
Версія для монолітної архітектури



Слайд 10

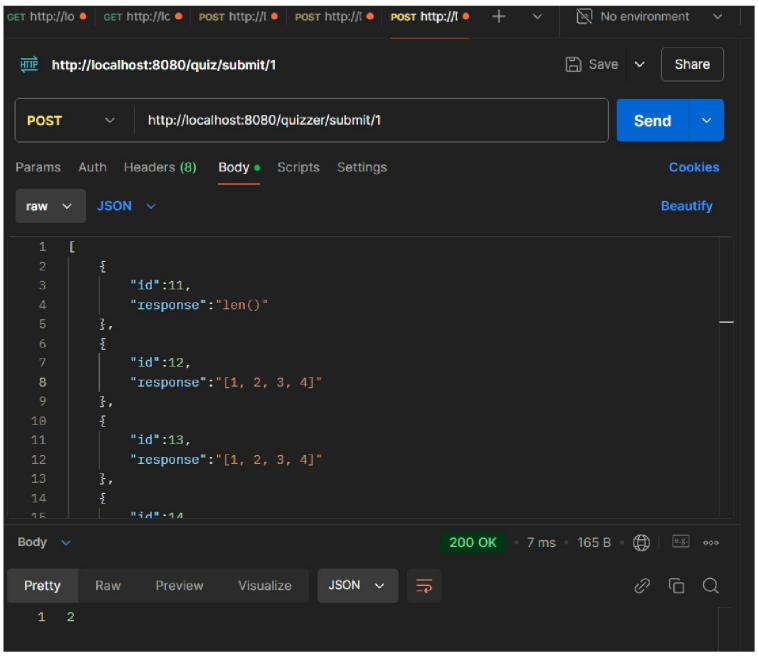
Розробка додатку Quizzer

Версія для мікросервісної архітектури



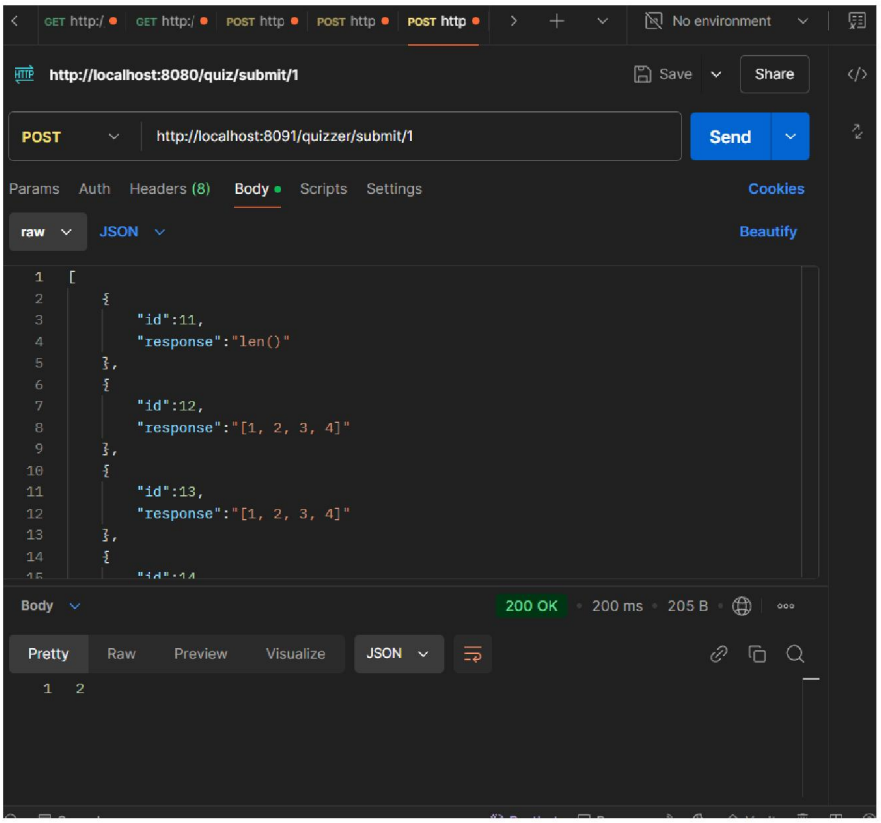
Слайд 11

Тестування Моноліт



Слайд 12

Тестування
Мікросервіси



Слайд 13

Рекомендації

- **Для малих проєктів -> монолітна архітектура.**
- **Для великих і динамічних систем -> мікросервісна архітектура.**

Слайд 14

Висновки

Мікросервісна архітектура є перспективним підходом для створення складних і масштабованих систем, але її застосування доцільне лише за умови готовності до інвестицій у складну інфраструктуру та спеціалізовані команди. Монолітна архітектура залишається актуальною для невеликих або менш динамічних проектів. Вибір архітектури повинен базуватися на специфіці проекту, його масштабі та довгострокових цілях.

Слайд 15