ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка моделі обробки запитів до серверу з мікросервісною архітектурою на основі технології RabbitMQ»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
(код, найменування спеціальності)
освітньо-професійної програми «Інженерія програмного забезпечення»
(назва)

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

_____ Дмитрій ПРИХОДЬКО
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-64

Дмитрій ПРИХОДЬКО

Керівник: Вікторія КОРЕЦЬКА
к.пед н., доцент

Рецензент: _____
науковий ступінь, вчене звання Ім’я, ПРІЗВИЩЕ

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Прихольку Дмитрію Анатолійовичу

1. Тема кваліфікаційної роботи: Розробка моделі обробки запитів до серверу з мікросервісною архітектурою на основі технології RabbitMQ

керівник кваліфікаційної роботи Вікторія КОРЕЦЬКА к.пед н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023р. №145

2. Строк подання кваліфікаційної роботи «29» грудня 2023р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, документація RabbitMQ, вимоги до розробки мікросервісної архітектури за використання програмних брокерів сповіщень.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1.Дослідження вимог до організації коду в мікросервісній архітектурі
2.Аналіз функцій та можливостей програмного брокеру повіщень RabbitMQ

3. Розробка вимог до розміщення компонентів системи, моделювання програмної моделі обробки запитів

5. Перелік графічного матеріалу: *презентація*
   1. Порівняльна таблиця технологій та протоколів комунікації між мікросервісами
   2. Діагарами переходу та взаємодії системних компонентів
   3. Програмна можель визначення черг запитів
   4. Формальна модель обробки запитів
   5. Оцінка ефективності обробки запитів
   6. Програмна модель алгоритму обробки запитів
   7. Схема розміщення системних компонентів
   8. Результати моделювання обробки запитів з використанням технології RabbitMQ

6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
2	Вивчення матеріалів для аналізу реалізації мікросервісної архітектури	06.11-12.11.23	
3	Дослідження програмних брокерів сповіщень	13.11-19.11.23	
4	Аналіз особливостей використання програмних брокерів сповіщень як способу комунікації між сервісами	20.11-26.11.23	
5	Розробка програмної та функціональної моделі обробки запитів	27.11-03.12.23	
6	Проектування схеми розміщення компонентів	04.12-10.12.23	
7	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
8	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти _____ Дмитрій ПРИХОДЬКО
(підпис)

Керівник
кваліфікаційної роботи _____ Вікторія КОРЕЦЬКА
(підпис)

РЕФЕРАТ

Текстова частина магістерської роботи: 70с., 34 рис., 5 табл., 25 джерел.

Об`єкт дослідження – масштабування та розподілення навантаження на серверну частину додатку.

Предмет дослідження – методи та підходи до реалізації мікросервісної архітектури.

Мета роботи – підвищення стабільності та загального перформансу роботи додатку за рахунок впровадження мікросервісної архітектури та використання програмних брокерів сповіщень .

Методи дослідження – методи проектування мікросервісної архітектури, методи проектування та розробки програмного забезпечення, технології мультипарадигменного та об’єктно-орієнтованого програмування.

У роботі проведено аналіз існуючих підходів до реалізації мікросервісної архітектури та використання поштових брокерів сповіщень як способу комунікації між сервісами, виконано огляд існуючих методик та підходів до організації коду в мікросервісній архітектурі, розглянуто наявні технології реалізації програмних брокерів сповіщень. Здійснено розробку програмної та функціональної моделей обробки запитів та схему розміщення компонентів. Розроблено програмне забезпечення для реалізації моделі обробки запитів за допомогою програмного брокера сповіщень RabbitMQ.

Проведено аналіз отриманих результатів моделювання та визначено переваги використання мікросервіної архітектури за використання програмного брокеру сповіщень RabbitMQ.

КЛЮЧОВІ СЛОВА: DOCKER, МІКРОСЕРВІСИ, ПРОГРАМНІ БРОКЕРИ СПОВІЩЕНЬ ,МІКРОСЕРВІСНА АРХІТЕКТУРА, RABBITMQ, AMQP, REST

ABSTRACT

Text part of the master's qualification work:

70 pages, 34 pictures, 5 tables, 25 sources.

Research object – scaling and load distribution to the server part of the application.

Research subject – methods and approaches to implementing microservices architecture.

Research aim – improving the stability and overall performance of the application through the implementation of microservices architecture and the use of message brokers.

Research methods – methods of designing microservices architecture, methods of designing and developing software, technologies of multi paradigm and object-oriented programming.

The paper analyzes existing approaches to implementing microservices architecture and the use of message brokers as a means of communication between services. An overview of existing methods and approaches to organizing code in microservices architecture is provided. The available technologies for implementing message brokers are considered. The development of software and functional models for processing requests and a diagram of component placement are carried out. Software is developed to implement the request processing model using the RabbitMQ message broker.

An analysis of the obtained simulation results is conducted, and the advantages of using microservices architecture over using the RabbitMQ message broker are identified.

KEYWORDS: DOCKER, MICROSERVICES, MESSAGE BROKERS, MICROSERVICES ARCHITECTURE, RABBITMQ, AMQP, RE

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД - База Даних

AMQP - Advanced Message Queuing Protocol

REST - Representational State Transfer

ПБС - Програмні брокери сповіщень

ВСТУП

В сучасному світі швидкого технологічного розвитку та зростаючих вимог до програмного забезпечення використання мікросервісної архітектури для масштабування та розподілення навантаження на серверну частину додатку стає надзвичайно актуальним. Ця архітектурна парадигма надає ефективний інструмент для розвитку та управління складними програмними системами. Забезпечуючи гнучкість у розробці, ефективне масштабування, швидку поставку продукту, оптимальне використання ресурсів та високу доступність, мікросервісна архітектура допомагає вирішувати виклики, пов'язані з вимогами до сучасних програмних рішень. Зокрема, це важливо в умовах постійних змін в бізнес-вимогах та технологічних тенденціях, дозволяючи системам бути більш адаптованими, надійними та легко розвиватися з плином часу, поштові брокери сповіщень в свою чергу є гнучким та надійним способом комунікації між сервісами, що надає додаткових можливостей для швидкого внесення змін.

Таким чином, розробка програмної моделі обробки запитів за використання поштових брокерів сповіщень є актуальним і важливим завданням.

Об`єкт дослідження – масштабування та розподілення навантаження на серверну частину додатку.

Предмет дослідження – методи та підходи до реалізації мікросервісної архітектури.

Мета роботи – підвищення загального перформансу роботи серверу, підвищення надійності та стабільності роботи серверу, шляхом розробки програмної моделі обробки запитів та схеми розміщення компонентів.

Методи дослідження – методи визначення черг запитів, методи реалізації мікросервіної архітектури, методи оптимізації розподілення навантаження між сервісами.

Практична значущість результатів полягає в використанні розробленої моделі для оптимізації розподілення навантаження на серверну частину додатку та підвищенню загального рівня стабільності серверу.

Для досягнення мети вирішувалися наступні завдання.

1. Аналіз існуючих підходів до побудови мікросервіснох архітектури.
2. Дослідження методів визначення черг запитів.
3. Розробка програмної та функціональної моделі обробки запитів при використанні поштових брокерів сповіщень.
4. Розробка програмного забезпечення для реалізації моделі.
5. Проведення моделювання схеми розміщення компонентів системи.

1 АНАЛІЗ ПІДХОДІВ ВИКОРИСТАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА ПРОГРАМНИХ БРОКЕРІВ СПОВІЩЕНЬ

1.1 Мікросервісна архітектура

Мікросервісна архітектура - це підхід до розробки програмного забезпечення, при якому програма розбивається на невеликі, незалежні сервіси, які взаємодіють між собою через визначений API (інтерфейс програмування застосунків).Кожен мікросервіс відповідає за конкретну бізнес-функціональність та може бути розгорнутий та масштабований незалежно від інших сервісів. Такий підхід розглядає програмне забезпечення як набір взаємодіючих сервісів, кожен з яких може бути розвинений, вдосконалений та масштабований незалежно.

Основні риси мікросервісної архітектури включають:

- Розбиття на сервіси: Програмне забезпечення розбивається на невеликі, автономні сервіси, які можуть бути розвинуті, використовуючи різні технології та мови програмування.
- Незалежність: Кожен мікросервіс може бути розгляданий як окремий модуль, і він може бути розроблений, оновлений, та масштабований незалежно від інших сервісів.
- Комунікація через API: Сервіси взаємодіють між собою через чітко визначені API, які можуть бути HTTP/REST або іншими механізмами комунікації.
- Автоматизація деплоїв: Мікросервіси можуть бути незалежно деплоєні та масштабовані, що сприяє швидкому впровадженню нових функцій та змін.
- Ізоляція відказів: Відмова одного сервісу не повинна впливати на роботу інших сервісів. Система повинна бути резилієнтною до відмов.

- Самостійність команд: Кожна команда розробників може бути відповідальною за свій власний мікросервіс, сприяючи швидкому розвитку та інноваціям.
- Гнучкість та масштабованість: Мікросервісна архітектура надає гнучкість в розробці та можливість масштабування окремих компонентів системи.

Концепція мікросервісної архітектури була вперше представлена у 2011 році. Мартін Фаулер та Джеймс Льюїс у своєму статті "Microservices" визначили основні принципи цього підходу. З того часу ідеї мікросервісів поширилися та стали базою для багатьох технологічних інновацій, виникла мікросервісна архітектура як реакція на необхідність підвищеної гнучкості та швидкості в розробці, а також відповідь на високу конкуренцію та швидкі зміни у вимогах ринку. Цей підхід дозволяє компаніям бути більш гнучкими та ефективними у своїй розробці та обслуговуванні програмного забезпечення. На рис. 1.1. представлена різниця взаємодії системних компонентів між монолітною та мікросервісними архітектурами.

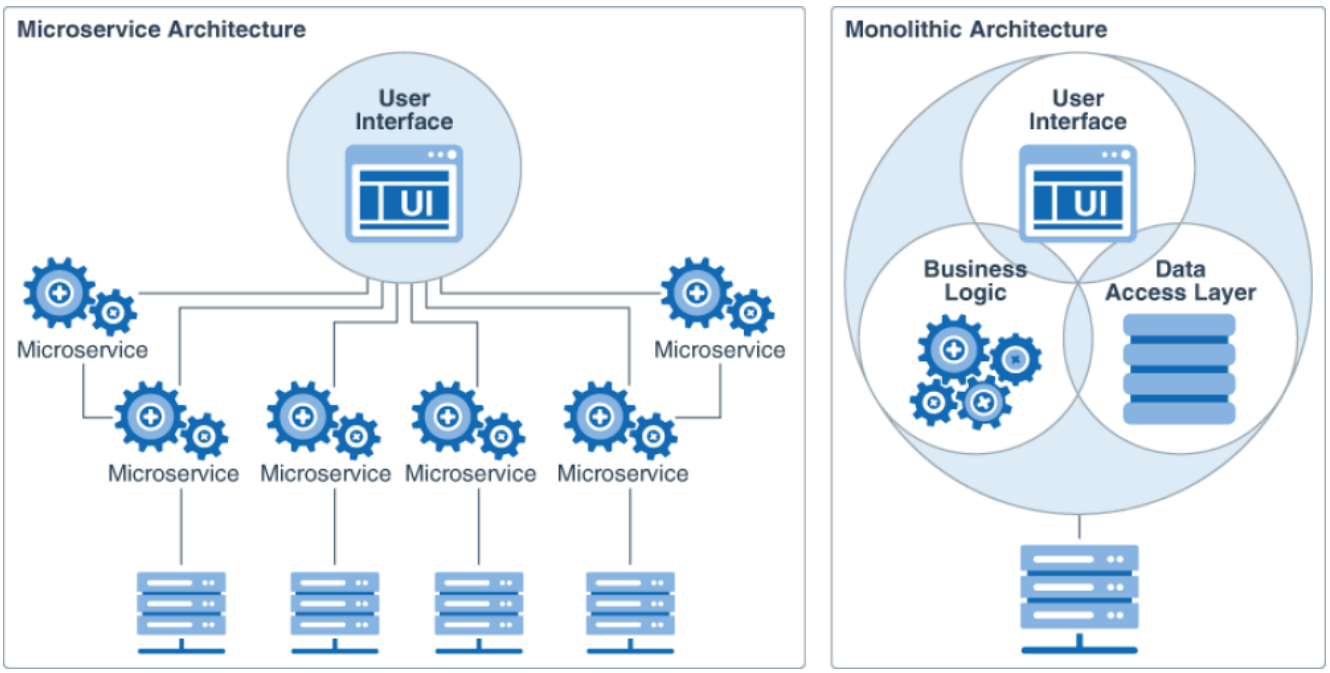


Рис. 1.1 Взаємодії системних компонентів

1.2 Підходи організації коду в мікросервісної архітектури

Мікросервісна архітектура як концепція розглядає серверверну частину як набір атомарних незалежних сервісів розробка та підтримка яких може бути не легким завданням, кожна команда розробників обирає для себе підходи до організації коду виходячи з методів розгортання, тестування та підтримки кодовох бази, але все ж до найбільш використовуваних можна віднести такі методи як використання монолітного репозиторію, та винесення кодової бази кожного сервісу в окремий репозиторій.

Mono-repo (монолітний репозиторій) - це підхід за якого кодова база всіх мікросервісів знаходиться в одному репозиторії схематичне представлення див. рис. 1.2, такий підхід може бути зручним для невеликих команд розробників або в випадках коли команда розробників часто змінюється, оскільки розробники мають доступ до усіх наявних сервісів та загальну документацію, що значно зменшує час для того щоб розібратись з кодовю базою, також такий підхід дозволяє створювати власні бібліотеки з перевикористовуваним кодом прямо в середині інфраструктури проекту, оскільки більшість мов програмування та фреймворків наразі підтримують функціонал для розміщення бібліотек між сервісами. Звісно в такого підходу є і свої мінуси такі як постійна необхідність в оновленні кодової бази оскільки репозиторій є спільним, внесення змін в окремий сервіс вимагає оновлення всієї кодової бази через можливе виникнення не сумісності версій або можливість виникнення конфліктів у системі контролю версій, також такий підхід з часом ускладнює підтримку кодової бази оскільки всі сервіси знаходяться в одному репозиторії збільшення їх кількості, розширення функціоналу, процес розгортання та тестування стає доволі важкою задачею.

Multi-repo - підхід за якого кожен окремий срвіс має власний репозиторій схематичне представлення див. рис. 1.2, є гарним рішенням коли команди розробників закріплюються за окремими сервісами або надалі сервіс може розглядатись як окремий додаток для інтеграції з сторонніми API. За такого

підходу, підтримка, розгортання та тестування кодової бази стає більш гнучким та легшим у реалізації, також такий підхід не вимагає від розробників постійної орієнтації в змінах інших сервісів, що значно збільшує продуктивність та зменшує час розробки MVP версій сервісів, при такому підході бібліотеки, компілюються та вивантажуються на репозиторії і потім встановлюється як окремі залежності для сервісів. Основними мінусами цього підходу є необхідність великої команди розробників, та більш важкий та тривалий процес початкового процесу розгортання та інтеграції між сервісами.

Розробка та підтримка безпосередньо самих сервіси можу визначитись за такими критеріями як розміри навантажень та організації коду навколо бізнес-процесів, перший критерій визначає чи може один сервіс мати більше одного або декількох сценаріїв реалізації, якщо наприклад сервіс займається складними обрахунками або обробкою великих файлів, такий сервіс доцільніше буде залишити з одним сценарієм роботи так як важкі операції можуть блокувати виконання інших сценаріїв або призвести до перевантаження та виходу з ладу системи в такому разі інші операції не будуть опрацьовуватись до завершення попереднього процесу або відновлення дієздатності сервісу, другий критерій спрямований на більше чітке визначення функціональності сервісу та його можливостей.

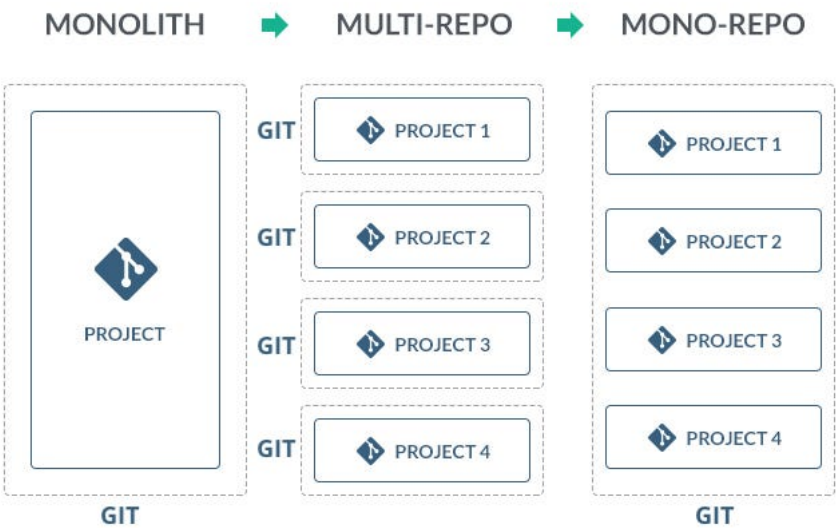


Рис. 1.2 Схематичне представлення розміщення репозиторїв та сервісів

1.3 Способи комунікації між мікросервісами

В мікросервісній архітектурі, ефективна комунікація між різними мікросервісами є ключовою для забезпечення гнучкості та масштабованості системи. Розглядання різних методів комунікації допомагає вибрати оптимальний підхід, що враховує специфіку проекту та його вимог, способи комунікації між мікросервісами представлені в Таб. 1.1.

Таблиця 1.1

Порівняння протоколів обміну даних

Назва технології	Короткий опис	Переваги	Недоліки
HTTP / Rest API (протокол)	Опис: REST API є стандартом архітектури для створення взаємодій між компонентами системи через мережу. Використовує HTTP-протокол та визначені ресурси з унікальними URI для обміну даними. Основні юзкейси: Клієнт-серверна комунікація для доступу до ресурсів (GET, POST, PUT, DELETE). Розробка веб-застосунків та мобільних додатків. Інтеграція з різними технологіями та платформами.	● Стандартний та широко використовуваний протокол. ● Легко взаємодіяти з іншими технологіями та інструментами. ● Простота реалізації та розуміння.	● Може виникнути накладні витрати через надмірні запити. ● Велика залежність від мережі.

Продовження таблиці 1.1

Порівняння протоколів обміну даних

Назва технології	Короткий опис	Переваги	Недоліки
Message Brokers / AMQP (протокол)	Опис: AMQP - протокол для передачі повідомлень у розподіленому середовищі з підтримкою повідомлення через брокера. Основні юзкейси: Системи обробки повідомлень та черг. Бізнес-застосунки, які потребують гарантовану доставку повідомлень. Реалізація різноманітних шаблонів обміну повідомленнями.	● Асинхронна комунікація ● Здатність до обробки великої кількості повідомлень ● Можливість реалізації патернів публікації-підписки.	● Збільшення складності системи через введення посередника.
Message Brokers / MQTT (протокол)	Опис: MQTT - протокол для обміну повідомленнями, розроблений для ефективної передачі даних в умовах обмеженої доступності та надійного з'єднання Основні юзкейси: Інтернет речі (IoT). Системи реального часу, де низька затримка є важливою.	● Легкий та ефективний протокол для обміну повідомленнями. ● Низькі накладні витрати та швидка доставка повідомлень. ● Підтримка як синхронної, так і асинхронної комунікації.	● Не так широко використовується, як HTTP/REST API. ● Потребує підтримки та реалізації брокера повідомлень. ● Має набагато вужчий функціонал ніж AMQP протокол.

Продовження таблиці 1.1

Порівняння протоколів обміну даних

Назва технології	Короткий опис	Переваги	Недоліки
GraphQL (технологія написана за використання HTTP протоколу)	Опис: GraphQL - мова запитів та сервер для взаємодії з даними. Дозволяє клієнтам отримувати тільки ті дані, які їм потрібні. Основні юзкейси: Розробка веб-застосунків, де потрібно гнучке отримання даних. Зменшення кількості запитів на сервер через вибіркове отримання даних. Реалізація API, яке підтримує різноманітні запити та вимоги клієнтів.	● Гнучка модель запиту-відповіді. ● Зменшення кількості запитів через вибіркове отримання даних. ● Легше виявлення і усунення зайвих даних.	● Вивчення та впровадження може вимагати додаткових зусиль. ● Підвищена складність валідації запитів та дозвіллю доступу.

При порівнянні способів комунікації, HTTP/REST API вирізняється своєю простотою та загальною доступністю, гарно підходить для реалізації MVP систем які працюють з малими об’ємами даних та не зав’язані на складних операціях обробки даних, тоді як Message Brokers надають можливість асинхронної комунікації між сервісами, що надає змогу оптимізувати процес обробки даних за допомогою створення різних черг та налаштуванням пріоритетів їх обробки, також надає зберігання повідомлень, що зменшує кількість відказів системи та втрати користувацьких запитів, ще однією перевагою цього методу є MQTT протокол який є одним з найкращих рішень для мобільних застосунків через свою

легкість та швидкість, GraphQL забезпечує гнучкий підхід до обміну даними та може бути дуже зручним для реалізації мікросервісів, оскільки кількість кінцевих точок для отримання даних є значно меншою ніж у інших систем, що значно полегшує процес розробки оскільки при зміні вимог до отримання даних одного сервісу, код іншого може залишатись незмінним, та має зменшення обсягу трафіку за допомогою визначення схем даних які необхідно отримати.

При розробці та впровадженні мікросервісної архітектури одними з найважливіших аспектів є зменшення кількості відмов, розподілення навантаження на серверну частину та покращення рівня надійності системи (в даному випадку під надійністю мається на увазі, збереження та забезпечення стовідсоткової обробки кожного запиту користувача) тому Message Brokers (AMQP / MQTT протоколи) є найкращим рішенням для реалізації обміну даними між сервісами, оскільки за його використання та введення в загальну інфраструктуру він бере на себе роль посередника в обміні даних, займаючись збереженням запитів, що в свою чергу забезпечує надійність обробки запитів оскільки при виходу з ладу одного з сервісів під час обробки ПБС буде знати що здача не була виконана і буде зберігати його до того часу доки один з інших сервісів його не обробить або поточний червіс не відновить свою роботу для завершення задачі. Також за допомогою ПБС можна швидко та легко налаштувати кількість запитів які одночасно може обробляти один сервіс, таким чином можна розподілити навантаження між сервісами уникаючи додаткових інфраструктурних рішень, як load-balancers(задачею яких є аналізувати поточний рівень навантаження та делегувати запити між сервісами). Визначившись з спобом обміну важливо також обрати технологію яка буде реалізовувати даний функціонал, най простішим рішенням є використання MQTT протоколу оскільки він є найлегшим у реалізації та швидкості роботи проте втрачає у кількості функціоналу та технологій реалізації порівняно з AMQP протоколом, який в свою чергу надає додаткові можливості обробки черги, створюючи різні типи черг та дозволяє реалізовувати синтетичні події, які будуть викликатись за певного сценарію. Спираючись на поточні тенденції розробки в яких найважливішими

пунктами є гнучкість та масштабованість системи, AMQP протокол є більш релевантним рішенням, оскільки при швидкій зміні бізнес правил, його легше адаптувати або розширити до необхідного функціоналу.

1.4 Розгортання та підтримка мікросервісної архітектури

Мікросервісна архітектура, яка передбачає розгортання програмного забезпечення як набору невеликих та автономних сервісів, вимагає специфічних підходів до деплою та підтримки. У цьому контексті, ефективне впровадження, масштабування та підтримка є ключовими елементами для успішного функціонування системи. В «Таб. 1.2» проведено аналіз та порівняння існуючих та найпопулярніших технологій та методологій для розгортання та підтримки кодової бази при використанні мікросервісної архітектури.

Таблиця 1.2

Порівняння технологій та методологій розгортання мікросервісної архітектури

Назва технології / методології	Короткий опис	Переваги	Недоліки
Контейнеризация (Docker, Podman)	Упаковка програмного забезпечення та його залежностей разом у контейнер, який може бути виконаний на будь-якому середовищі, що підтримує контейнери.	● Стандартизація середовищ виконання. ● Легке розгортання та реплікація. ● икористання оркестраторів (Kubernetes) для автоматизованого управління.	● Збільшення складності конфігурації та підтримки контейнерів.

Продовження таблиці 1.2

Порівняння технологій та методологій розгортання мікросервісної архітектури

Назва технології / методології	Короткий опис	Переваги	Недоліки
Оркестрація (Kubernetes, Docker Swarm)	Автоматизація управління та масштабуванням контейнерів у розподіленому середовищі.	● Автоматизоване розгортання та масштабування. ● Самозагоювання та відновлення системи після збоїв.	● Висока порогова вартість вивчення для новачків. ● Потребує додаткових ресурсів для підтримки інфраструктури.
Serverless	Розробка і виконання коду без необхідності управління серверами; обчислення автоматично масштабуються залежно від навантаження.	● Автоматизована масштабованість. ● Оплата лише за використання ресурсів.	● Знижена контроль над ресурсами та середовищем виконання. ● Специфічність для певних платформ та обмеження у виборі технологій
Традиційний Деплой на Віртуальні Машини	Використання віртуальних машин для розгортання та виконання програмного забезпечення.	● Велика контроль над конфігурацією та ресурсами. ● Забезпечення ізольованого середовища для кожного сервісу.	● Витрати на управління віртуальними машинами. ● Сповільнення швидкості розгортання та масштабування.

Продовження таблиці 1.2

Порівняння технологій та методологій розгортання мікросервісної архітектури

Гібридні Підходи	Використання комбінації різних підходів залежно від потреб проекту.	● Забезпечення гнучкості та варіативності вибору інструментів для конкретних завдань. ● Зменшення негативних аспектів кожного методу завдяки поєднанню їхніх переваг.	● Потребує уважного управління та конфігурації для уникнення конфліктів та збереження стабільності системи.

Загалом вибір конкретного варіанту залежить від вимог проекту, внутрішньої експертизи команди, а також від наявності ресурсів для обслуговування інфраструктури та інші фактори. В данній роботі розгортання та підтримка коду реалізовується через контейнеризацію оскільки цей підхід є одним з най гнучкіших та легко розгортуваних основними перевагами у порівнянні з іншими є:

1. Простота та швидкість реалізації контейнеризації, наразі в відкритому доступі є купа образів для запуску майже любої технології та її версії;
2. Легкість в розгортанні досягається тим що контейнери можна легко розмістити як на своїй локальній машині так і на любому відкритому хмарному сервісі який підтримує контейнеризацію;
3. За допомогою контейнерів можна виділяти кількість ресурсу(пам’яті) необхідної для функціонування конкретного сервісу, за необхідності ці параметри можна змінювати;

4. Також контейнеризація легко може бути розширена до оркестрації за необхідності оскільки вже готові контейнери можна розмістити в Kubernetes.

1.5 Оптимізації та зменшення навантаження на серверну частину

Оптимізація та зменшення навантаження на серверну частину є критично важливими аспектами при використанні мікросервісної архітектури. Забезпечення ефективної роботи та високої продуктивності мікросервісів є однією з основних задач при їх впровадженні в «Таб. 1.3» розглянуто загальні підходи та техніки для зменшення навантаження на серверну частину.

Таблиця 1.3

Порівняння технологій та методологій зменшення навантаження на серверну частину

Назва технології / методології	Короткий опис	Переваги	Недоліки
Кешування Даних	Реалізація може бути завдяки впровадженню баз даних які зберігають інформацію в оперативній пам'яті або збережених в пам’ять на пряму	● Зменшення часу відповіді на запити. ● Відсутність необхідності повторного виконання однотипних запитів.	● Можливість застаріння кешованих даних.

Продовження таблиці 1.3

Порівняння технологій та методологій зменшення навантаження на серверну частину

Назва технології / методології	Короткий опис	Переваги	Недоліки
Масштабованість	Розгортання додаткових сутностей мікросервісу, та розподілення навантаження між ними	● Збільшення кількості екземплярів мікросервісів для розділення навантаження.	● Потребує додаткових ресурсів для управління та підтримки.
Оптимізація Бази Даних	Розширення кодової бази SQL	● Індексація та оптимізація запитів для швидшого доступу до даних. ● Використання кешування для зменшення навантаження на базу даних.	● Потребує досвіду у тонко налаштуванні та моніторингу бази даних.
Використання CDN	Впровадження додаткової маршрутизації	● Зменшення часу завантаження ресурсів для клієнтів. ● Розподілення навантаження за допомогою розподілених серверів.	● Додаткові витрати на обслуговування CDN.

Продовження таблиці 1.3

Порівняння технологій та методологій зменшення навантаження на серверну частину

Асинхронні Запити та Обробка	Впровадження ПБС в систему	● Покращення паралельності та швидкості обробки. ● Зменшення блокуючих операцій.	● Специфічність роботи з асинхронним кодом.
Розвиток Завантажених Запитів	Збереження отриманих результатів в оперативну пам’ять на клієнтській частині	● Відправлення узгоджених та обмежених запитів до серверів. ● Зменшення кількості непотрібних або зайвих запитів.	● Вимагає обробки на стороні клієнта.
Моніторинг та Аналіз Продуктивності	Логування значень навантаження на систему за певних подій	● Ідентифікація слабких місць та можливостей для оптимізації. ● Покращення ефективності та стабільності системи.	● Потребує постійного моніторингу та управління.

Більшість цих технологій та методологій є загальнимим та можуть бути використані як при монолітній архітектурі так і при за провадженні мікросервісної, проте серед наведених методологій можна виділити такі як моніторинг та аналіз продуктивності, асинхронні запити та обробка і масштабованість оскільки вони краще себе проявляють або можуть бути реалізовані саме при запровадженні мікросервісної архітектури.

Так наприклад асинхронні запити та обробка і масштабованість можуть бути реалізовані виключно при роботі з мікросервісною архітектурою оскільки передбачають обмін даними між декількома сервісами, асинхронність досягається за впровадження ПБС, оскільки запити опрацьовуються в певній черзі та з пріоритетами, та можуть бути переадресовані на ті сервіси які наразі менше навантажені, масштабованість же це один з найголовніших аспектів мікросервісної архітектури і від його реалізації напряму залежить рівень перформансу системи. Логування може бути корисним як при постійному моніторингу навантаження на серверну частину для своєчасного введення додаткового інстансу мікросервісу, так і для аналізування та подальшої зміни розподілення ресурсів в системі для покращення рівня загального перформансу.

1.6 Інтеграція та взаємодія модулів з іншими системами

Інтеграція та взаємодія модулів з іншими системами є невід'ємною частиною мікросервісної архітектури, де різні сервіси співпрацюють для досягнення спільних цілей. Забезпечення ефективної інтеграції - це ключовий елемент успішного функціонування системи та надання послуг клієнтам. У цьому контексті, важливо розглянути різні аспекти і стратегії інтеграції для оптимальної взаємодії мікросервісів з іншими системами.

Одним з найважливіших кроків при інтеграції одних незалежних систем до інших є своєчасно та структуровано розроблена API документація, яка надасть іншим розробникам можливість детально ознайомитися з можливим функціоналом сервісу, та отримати інформацію про необхідні формати даних та протоколи взаємодії для успішної інтеграції. Сама по собі API документація також буде корисною для нових розробників або команди підтримки існуючих сервісів, оскільки це значно зменшить час на їх адаптацію приклад документації API див. Рис 1.3.

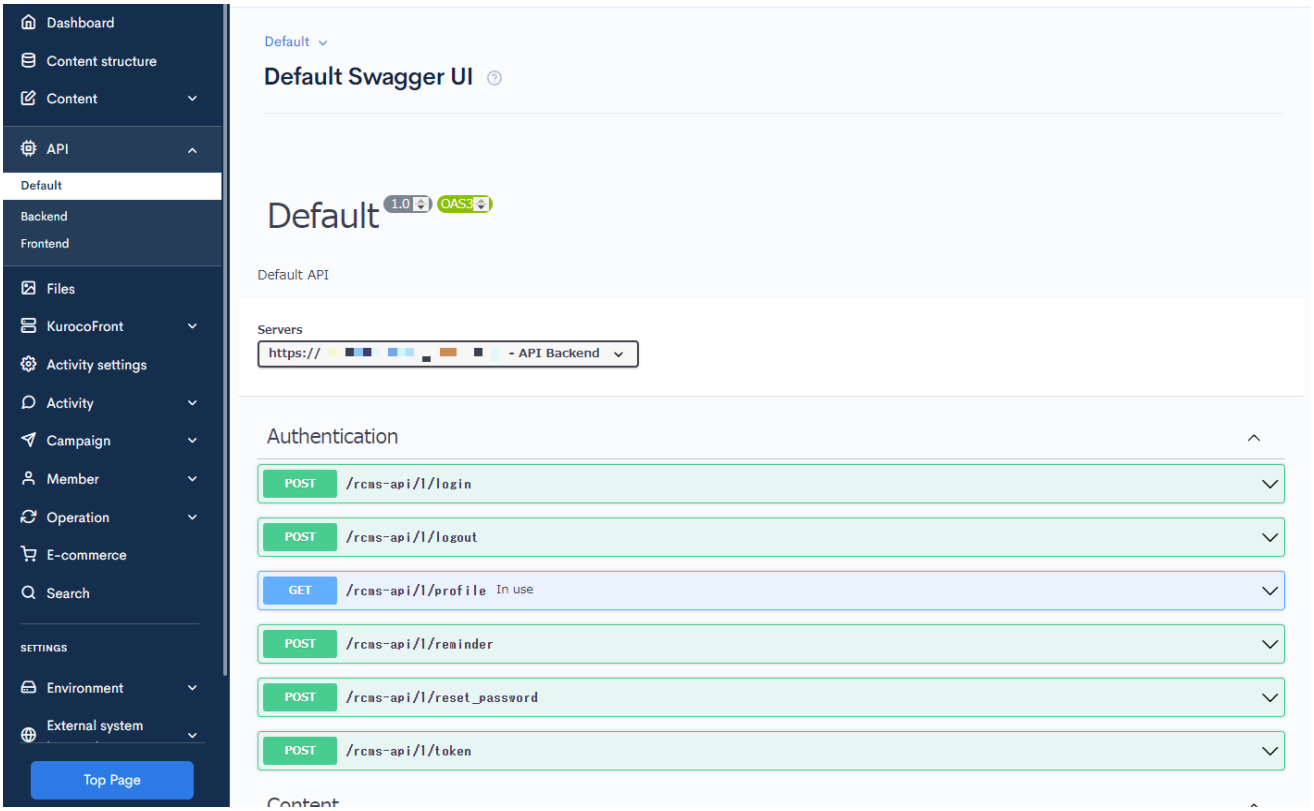


Рис. 1.3 Приклад документації коду за допомогою Swagger

З урахуванням того, що сторонні сервіси можуть мати власні вимоги до безпеки, важливо враховувати механізми шифрування та авторизації в обміні даними, оскільки API є відкритим до нього може звернутись абсолютно любий користувач маючи назву шляху та параметри для запиту. Забезпечення безпеки взаємодії з екосистемою сторонніх сервісів має високий пріоритет, для любих систем особливо для систем які працюють як з особистими даними користувачів або з їх фінансами.

Для ефективного контролю за взаємодією із сторонніми сервісами, важливо використовувати інструменти моніторингу та логування, так як проаналізувавши ці дані можна швидше та легше усувати виявлені недоліки в системі, також на основі логів можна організувати антиспам систему, або заблокувати IP адреси користувачів, які відправляли або намагались відправити SQL ін'єкції або інші скрипти для взламування серверної частини.

Тестування та валідація відіграють також значну роль при запровадженні незалежних сервісів, оскільки з API працюють не тільки команда розробників,

часто сервер може отримати не валідні дані або дані з шахрайським вмістом, ці дані можуть потрапити до серверу навіть з клієнтської частини якщо система працює з обробкою строкових даних які вводять користувачі, тож своєчасне тестування та валідація даних є необхідними для збереження доступності та надійності системи.

Розробка механізмів резервного копіювання та відновлення також є невід'ємною частиною незалежних сервісів, оскільки вони є в відкритому доступі і можливість взламати їх або відправити невалідний запит в рази зростає, що робить сервіс більш вразливим до хакерських атак, або відказу системи.

1.7 Роль програмних брокерів сповіщень у розробці програмного забезпечення

Програмні брокери сповіщень — це спеціалізовані компоненти, які відповідають за передачу повідомлень між різними частинами програмної системи. Основною їх роллю є забезпечення асинхронного взаємодії між модулями програми, а також розподіл подій та сповіщень у системі.

Однією з ключових переваг програмних брокерів сповіщень є зменшення зв'язку між компонентами системи. Замість того, щоб модулі взаємодіяли напряму один з одним, вони можуть використовувати брокери сповіщень для обміну інформацією. Це дозволяє забезпечити більшу гнучкість у внесенні змін у програмний код, оскільки зміни в одному модулі не обов'язково впливають на інші.

Крім того, програмні брокери сповіщень використовують подійно-орієнтовану архітектуру, що полегшує розробку подій та реакцію на них. Модулі можуть підписуватися на конкретні події і реагувати на них відповідно до своєї функціональності, що робить систему більш гнучкою та легко розширюваною, також це приносить асинхронність до обміну даними що робить більш гнучким процес їх обробки та налаштування черг.

Іншим аспектом важливості програмних брокерів сповіщень є підтримка роботи в розподіленому середовищі де забезпечення надійності та відновлення

помилок є одним з найбільших викликів. У сучасних системах, де компоненти можуть знаходитися на різних серверах або працювати в хмарному середовищі, важливо мати ефективний механізм для обміну даними. Програмні брокери сповіщень дозволяють забезпечити цю можливість, знижуючи навантаження на мережу та забезпечуючи швидку і надійну передачу повідомлень з подальшим їх збереженням що значно підвищує рівень надійності оскільки, кожен запит стовідсотково буде оброблено навіть при виході з ладу одного з основних модулів системи.

1.7 Аналіз існуючих технологій для реалізації поштових брокерів повіщень

Таблиця 1.4

Порівняння існуючих технологій для реалізації поштових брокерів повіщень

Брокер сповіщень	Мови програмування	Протоколи	Доступність	Основний функціонал та переваги
RabbitMQ	Erlang	AMQP	Висока	● Гарантує доставку повідомлень ● Гнучка конфігурація та маркування повідомлень ● Кластеризація для високої доступності та масштабованості ● Підтримка великої кількості мов програмування

Продовження таблиці 1.4

Порівняння існуючих технологій для реалізації поштових брокерів повіщень

Брокер сповіщень	Мови програмування	Протоколи	Доступність	Основний функціонал та переваги
Apache Kafka	Java	Kafka	Висока	● Висока пропускна здатність та швидкодія ● Утримання великої кількості повідомлень та логів ● Горизонтально масштабований з реплікацією даних
ActiveMQ	Java	OpenWire,MQTT	Висока	● Різноманітність протоколів сповіщень ● Висока доступність та масштабованість ● Підтримка JMS (Java Message Service) ●
MQTT (Mosquitto)	C	MQTT	Висока	● Легкість та ефективність для IoT застосувань ● Простий протокол з низьким рівнем обміну повідомленнями

Продовження таблиці 1.4

Порівняння існуючих технологій для реалізації поштових брокерів повіщень

Брокер сповіщень	Мови програмування	Протоколи	Доступність	Основний функціонал та переваги
Apache Pulsar	Java	Pulsar	Висока	● Гарантія принаймні одного разу вручення (at-least-once delivery) ● Географічна реплікація для високої доступності ● Архітектура "побудована з коробки" з підтримкою множини мов програмування
ZeroMQ	C++, Python	Custom	Висока	● Низька латентність та мінімальна накладна ● Різноманітність шаблонів обміну повідомленнями ● Не використовує централізовані сервери, але вимагає програмної логіки

Проаналізувавши існуючі технології для реалізації поштових брокерів сповіщень, було прийнято рішення зупинитись на RabbitMQ спираючись на такі фактори:

- Гарантована Доставка - RabbitMQ забезпечує надійну доставку повідомлень, що робить його ідеальним для сценаріїв, де важлива точність та надійність передачі даних.
- Гнучка Конфігурація - Його гнучка конфігурація та можливості маркування повідомлень роблять RabbitMQ пристосованим до різних потреб проектів, дозволяючи розробникам вибирати оптимальні налаштування.
- Кластеризація для Високої Доступності - RabbitMQ підтримує кластеризацію, що дозволяє створювати розподілені системи з високою доступністю. Це особливо важливо для уникнення вузьких місць та забезпечення безперервної роботи.
- Підтримка Різних Мов Програмування - Здатність використовувати RabbitMQ з різними мовами програмування (включаючи Erlang, Java, Python, та інші) робить його універсальним і легким у використанні для команд з різних технічних стеків.
- Відкритий Код - RabbitMQ має відкритий код, що надає можливість перевірити, адаптувати або виправляти код відповідно до конкретних потреб проекту.
- Розширені Можливості Маршрутизації - Багатофункціональна система маршрутизації дозволяє ефективно направляти повідомлення від виробників до споживачів відповідно до заданих правил.
- Широкий Спектр Розширень - Доступність різних плагінів та розширень робить RabbitMQ гнучким для різних сценаріїв використання, включаючи реалізацію додаткових протоколів та забезпечення додаткових можливостей.

2 РОЗРОБКА МОДЕЛІ ЗАПИТІВ ТА ЇХ ОБРОБКИ

2.1 Постановка задачі

У сучасних умовах розвитку програмних продуктів, особливо в контексті мікросервісної архітектури, виникає необхідність вдосконалення систем обробки запитів до серверу. Одним із ефективних та передових рішень для реалізації цього є використання технології RabbitMQ.

У зв'язку зі зростанням складності та обсягу функціональності сучасних програмних систем, виникають виклики в ефективності обробки запитів. Традиційні монолітні архітектури не завжди можуть забезпечити надійність роботи системи та необхідний рівень масштабованості та гнучкості,

Метою даного розділу є розробка моделі обробки запитів до серверу на основі мікросервісної архітектури з використанням технології RabbitMQ. Основні завдання включають:

Проектування архітектури системи: Розробка структури мікросервісів, їх взаємодії та інтерфейсів для забезпечення ефективної обробки запитів.

Вибір технологій: Аналіз та вибір технологій, що найкраще підходять для реалізації мікросервісів та взаємодії між ними, з особливим акцентом на RabbitMQ.

Розробка моделі обміну повідомленнями: Реалізація механізму обміну повідомленнями між мікросервісами, використовуючи RabbitMQ як надійний та масштабований посередник.

Інтеграція з сервером: Побудова зв'язку між розробленою моделлю та основним сервером для забезпечення взаємодії з клієнтами та обробки їх запитів.

Забезпечення безпеки та надійності: Впровадження механізмів забезпечення безпеки та високої доступності для забезпечення стійкої роботи системи.

В результаті виконання поставлених завдань передбачається отримання моделі обробки запитів, що базується на мікросервісній архітектурі та використовує технологію RabbitMQ для ефективного обміну повідомленнями. Система повинна бути гнучкою, масштабованою та готовою до високих навантажень.

Цей проект направлений на поліпшення продуктивності та гнучкості обробки запитів у сучасних програмних системах, що реалізовані в умовах мікросервісної архітектури за допомогою передових технологій, таких як RabbitMQ.

2.2 Функціональна модель обробки запитів

В контексті обміну даними між сервісами математичні моделі виступають як інструменти для формалізації та аналізу процесів обробки запитів, що дозволяє розробникам та інженерам систем точно визначати оптимальні стратегії роботи серверів.

При використанні таких моделей відбувається абстракція складних взаємодій компонентів системи, дозволяючи отримувати кількісні та якісні характеристики функціонування, розглядатимуться не лише теоретичні аспекти моделювання, а й практичні використання отриманих результатів для покращення функціональності програмних систем.

Насамперед важливо скласти математичні моделі REST та ПБС для подальшого аналізу та визначення структурних компонентів їх взаємодії та важливості.

REST (Representational State Transfer):

У REST-архітектурі, клієнт взаємодіє з сервером, використовуючи стандартні HTTP-методи (GET, POST, PUT, DELETE) для взаємодії з ресурсами.

Функціонально це можна виразити через формалізований опис структури запитів та відповідей, де використовуються HTTP-методи, URI (Uniform Resource Identifier) та формат обміну даними (наприклад, JSON або XML).

Message Brokers:

Message Brokers використовуються для асинхронної комунікації між компонентами системи.

Функціонально модель може включати опис структури повідомлень, обмін повідомленнями між виробниками (senders) та споживачами (consumers), а також механізми обробки помилок та гарантії доставки.

Ключові елементи включають у себе топіки (topics) або черги, де повідомлення публікуються та підписуються.

Приклад функціональної моделі:

Модель для REST:

Множини:

- *Req* - множина можливих HTTP-запитів (GET, POST, PUT, DELETE);
- *Res* - множина можливих HTTP-відповідей;

Функції:

- *Endpoint* : $Req \rightarrow Res$ - функція, яка визначає, як REST-сервіс обробляє HTTP-запити та генерує HTTP-відповіді;
- *Validation* : $Req \rightarrow \{true, false\}$ - функція перевірки валідності;

Параметри:

- *URI* - адреса ресурсу;
- *Headers* - заголовки HTTP запиту;
- *Body* - тіло HTTP запиту;
- *Method* - метод HTTP запиту;

Функціональна модель :

- Припустимо, $req \in Req$ - конкретний HTTP запит;
- Математична модель може бути визначена :

  $req = \{URI, Method, Headers, Body\}$

$Validation(req) \rightarrow \{true, false\}$

$Endpoint(req) \rightarrow res \in Res$

Модель для Message Brokers:

Множини:

- *Msg* - множина повідомлень, що можуть бути розміщені в черзі.
- *Components* - множина компонентів системи, які можуть підписуватися на черги.

Функції:

- *Publish*: *Message* → *Queue* - функція для розміщення повідомлень в черзі.;
- *Subscribe*: *Queue* → *Components* - функція для підписки компонентів на черги;
- *Process*: *Message* → *Result* - функція обробки повідомлень;
- *SaveToStorage*: *Queue* → *Queue* - функція збереження повідомлення;

Параметри:

- *Message* - адреса ресурсу;
- *Queue* - заголовки HTTP запиту;

Функціональна модель :

- Припустимо, $msg \in Msg$ - конкретне повідомлення;
- Функціональна модель може бути визначена :

  $msg = \{Body, Headers, Queue\}$

  $Publish(msg) \rightarrow Queue$

  $SaveToStorage(Queue) \rightarrow Queue$

  $Subscribe(Queue) \rightarrow Components$

Process(msg) → Result

Щоб провести оцінку ефективності впровадження ПБС до мікросервісної архітектури насамперед варто визначати основні критерії ефективності, до них можна віднести, час відгуку на запит, масштабованість, навантаження, ймовірність виходу з ладу сервісу, якість обробки помилок.

Розрахунок ефективності впровадження ПБС в порівнянні з REST :

Введемо змінні для визначення ефективності обробки запитів де *R* - REST, а *M* - Message Brokers :

- $T_{R,M}$ - час відгуку (в секундах).
- $S_{R,M}$ - масштабованість (відносний показник).
- L - навантаження (в запитах на секунду).
- $F_{R,M}$ - ймовірність виходу з ладу (від 0 до 1)
- $Q_{R,M}$ - якість обробки помилок (від 0 до 1)

Розрахунок ефективність розподілення навантаження E:

$$E_M = \frac{L}{S_M} * T_M \quad (2.1)$$

$$E_R = L * T_R \quad (2.2)$$

Розрахунок доступність сервісів A:

$$A_M = 1 - F_M * S_M \quad (2.3)$$

$$A_R = 1 - F_R. \quad (2.4)$$

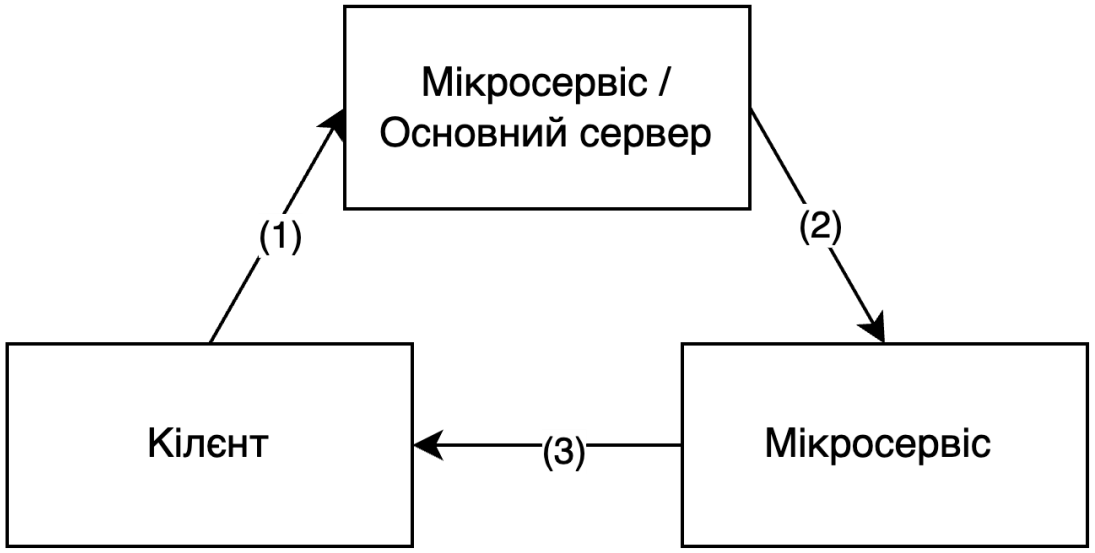
2.3 Аналіз існуючих програмних моделей обміну даними

Програмну модель обробки запитів можна визначити як набір окремих процесів, обмін даними між сервісами та іншими програмними модулями системи

такими як клієнтська частина та основний сервер, також до цих процесів можна віднести збереження та обробку даних.

Першочергово варто виділити та провести аналіз основних компонентів системи при запровадженні REST та ПБС.

На Рис 2.1 представлені основні компоненти системи та їх взаємодія, за впровадження REST, основними компонентами тут є мікросервіси та клієнт, кількість мікросервісів може варіюватись в залежності від бізнес вимог, перевагами такого підходу є простота в реалізації, відносно невелика кількість ресурсів необхідних для підтримки всієї системи, мінусами є повна залежність від інтернет трафіку, відсутність розподілення навантаження між сервісами, не можливість обробки відказів системи, та відсутність резервного копіювання даних. Такий підхід може бути виправданий при розробці MVP версії продукту.



1. Відправка клієнтських данних до мікросервісу
2. Обмін даннимим між мікросервісами
3. Відправка оброблених данних на клієнт

Рис. 2.1 Діаграма взаємодії компонентів за впровадження REST

Алгоритм роботи обробки запитів представлений на Рис. 2.2.

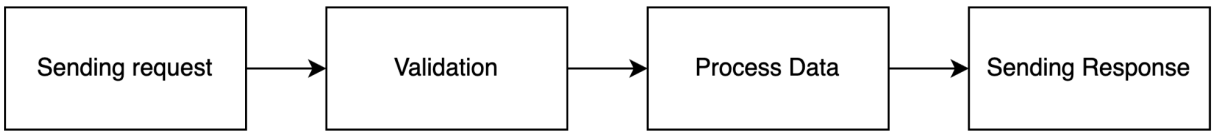


Рис. 2.2 Діаграма послідовностей за впровадження REST

На Рис 2.3 представлена діаграма компонентів для REST з введенням нових компонентів для вирішення проблем з розподілом навантаження (load balancer component) та підтримкою системи відказів (log script), для підтримки системи відказів також можна використати інший підхід з ввдененням додаткового сервісу, або окремо розподіленої бази даних по типу Redis (зберігання даних в таких базах відбувається одразу в оперативну пам'ять), такий підхід є значно кращим за попередній, проте вимагає обробки більш складних сценаріїв, таких як постійний моніторинг лог файлів, на наявність не оброблених запитів та оновлення конфігурації load balancer при введенні нових сутностей мікросервісу.

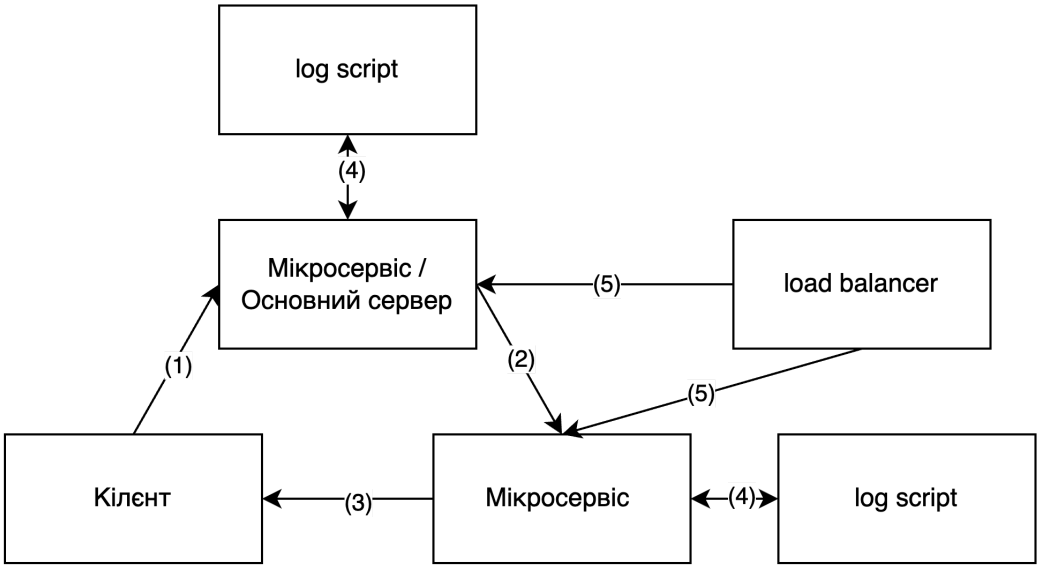


Рис. 2.3 Діаграма взаємодії компонентів за впровадження REST з додатковими компонентами оптимізацій

Діаграма станів для REST з урахуванням оптимізацій та введенням системи відказів є дещо складнішою ніж у звичайного підходу, оскільки сюди додаються такі процеси як запис логів та їх видалення основними виклаиками за такого підходу є зміна кодової бази, оскільки необхідно постійно проводити моніторинг лог файлів на перевірку наявності не опрацьованих запитів див Рис. 2.4

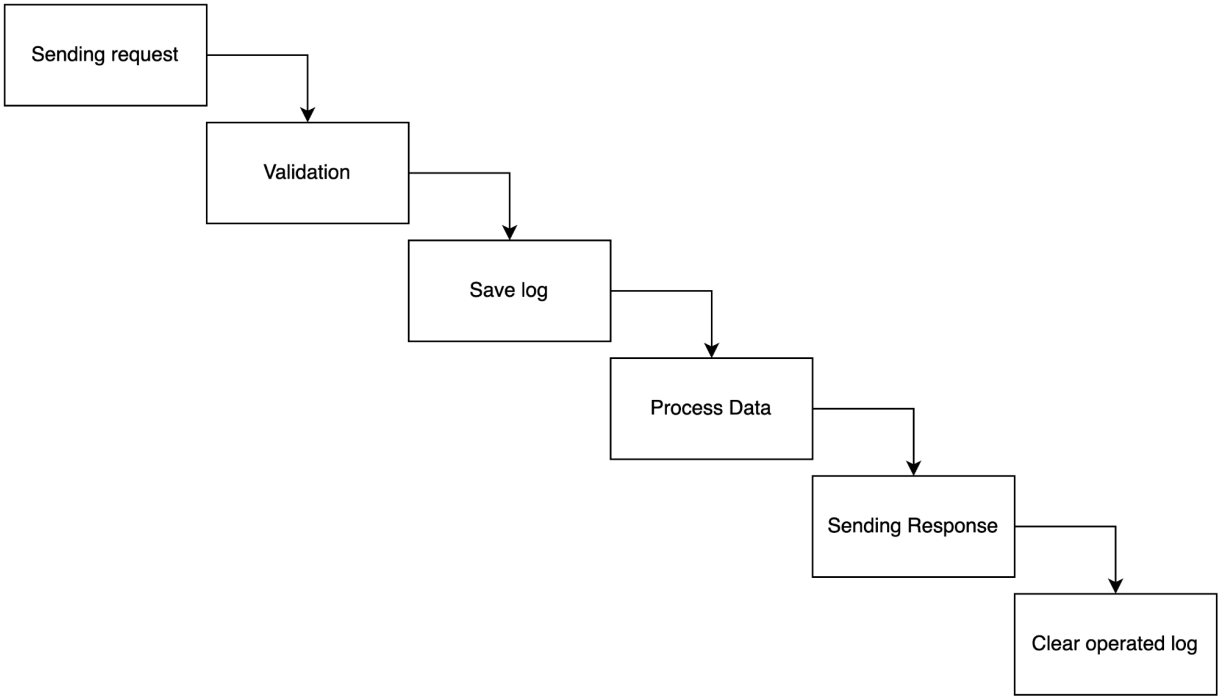
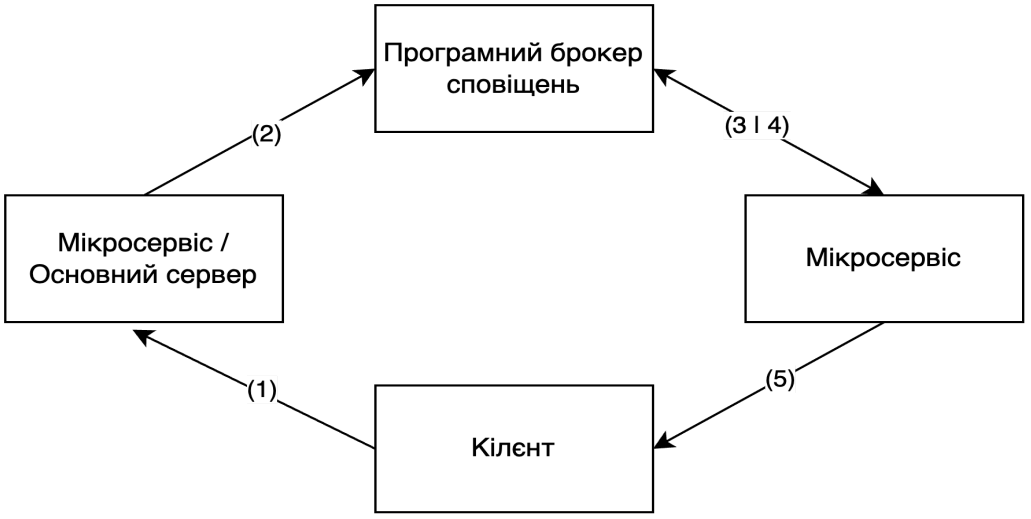


Рис. 2.4 Діаграма послідовностей за впровадження REST з додатковими компонентами оптимізацій

Визначивши недоліки REST як способу комунікації між сервісами, можна перейти до огляду Поштових брокерів сповіщень див Рис. 2.5..

Основними перевагами використання ПБС є асинхронний спосіб обміну даними між сервісами, більша швидкість в обміні даними та менша затримка, іншим важливим аспектом ПБС є збереження запитів до локального сховища, що значно спрощує процеси обробки відмов, також наявність роботи з чергами, пріоритизація та можливість обмежувати кількість запитів які може обробляти

сервіс одночасно робить процес оптимізації значно гнучкішим та легшим в реалізації, на Рис 2.6. відображений алгоритм роботи обробки запитів в ПБС.



1. Відправка клієнтських данних до мікросервісу
2. Відправка данних до поштового брокера
3. Відправка даних на обробку до мікросервіса
4. Відправка данних до брокеру про закінчення процесу обробки
5. Відправка оброблених данних на клієнт

Рис. 2.5 Діаграма взаємодії компонентів за впровадження ПБС

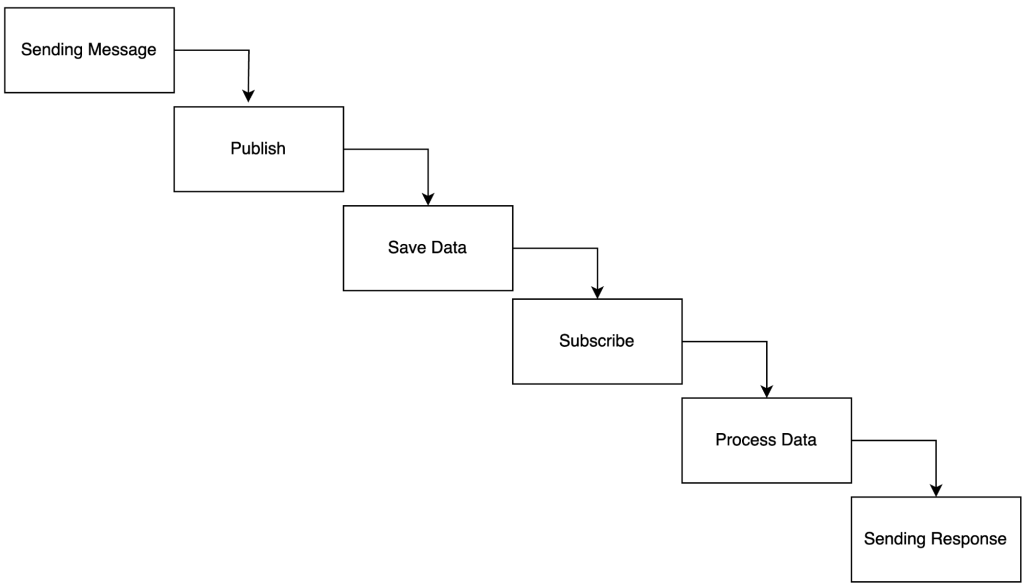


Рис. 2.6 Діаграма послідовностей за впровадження ПБС

2.4 Розробка програмної моделі запитів

Загалом визначення черг в поштових брокерах сповіщень відбувається в залежності від операцій які виконує певний сервіс, таким чином можна визначити стандартну модель визначення черги запитів запитів.

Стандартна модель визначення черги запиту

Типи операцій: operationTypes = [Type1, Type 2 …]
Кількість черг: operationTypes.length
Визначення назви черги: operationType[n]
Визначення кількості запитів для одночасної
обробки залежить від кількості ресурсів сервісу.

На Рис.2.7. представлена схема розподілення повідомлень сервісами з черги за звичайного визначення черги, за такого визначення черг ми має кількість сервісів яка дорівнює кількості типів обробки даних, також на схемі представлені додаткові сутності сервісів вони вводяться для додаткового розподілення навантаження на серверну частину, (N) при розподіленні це максимальна кількість запитів на одночасну обробку сервісом, визначається кількістю ресурсів якими володіє сервіс.

Мінусами такого підходу є те що великі операції які потрапляють в чергу раніше за інші, можуть блокувати потоки обробки або перенавантажувати оперативну пам’ять, що збільшить затримку отримання результів всім наступним користувачам.

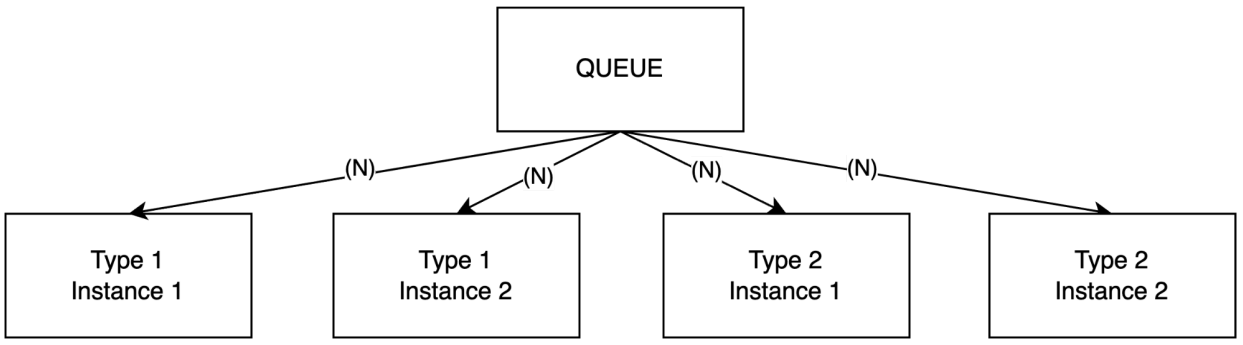


Рис. 2.7 Схема розподілення повідомлень сервісами з черги за звичайного визначення черги

Розроблена модель визначення черги запиту

Тип операції: operationTypes = [Type1, Type 2 …]

Складність операції: operationComplexity = [opComp1, opComp2 …]

Розмір вхідних даних: dataWeight = [dataWeight1, dataWeight2 …]

`Визначення градацій: gradeTypes = getGrade(dataWeight, operationComplexity) → [gradeType1, gradeType2 ⋯]`

Кількість черг: operationTypes.lenght * gradeTypes.lenght

Визначення назви черги: operationType[n] + gradeTypes[n]

Визначення кількості запитів для одночасної обробки залежить від кількості ресурсів сервісу та градації.

На Рис.2.8. представлена схема розподілення повідомлень сервісами з черги за градованого визначення черги, за такого визначення черг ми має кількість сервісів яка дорівнює кількості типів обробки даних помножених на кількість градацій, градації в свою чергу визначаються важкістю операції і об’ємом даних які визначені на таку операцію, також на схемі представлені додаткові сутності сервісів вони вводяться для додаткового розподілення навантаження на серверну частину, (N) при розподіленні це максимальна кількість запитів на одночасну обробку сервісом, визначається кількістю ресурсів якими володіє сервіс та градацією операції.

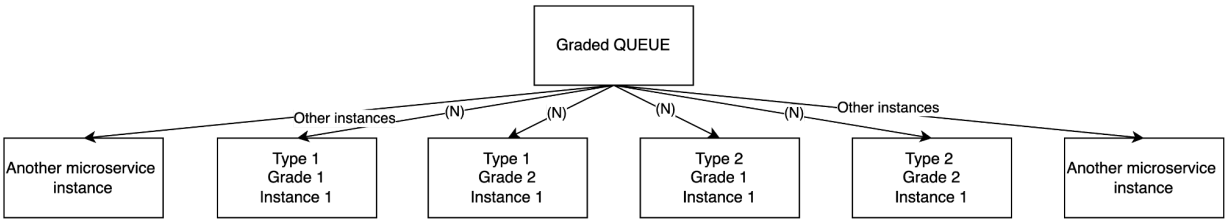


Рис. 2.8 Схема обробки даних сервісами з черги за градованого визначення черги

Звісно визначений спосіб розподілення черг має певні обмеження, вони регулюються кількістю загального ресурсу який можна виділити накількість необхідних мікросервісів, і загальним часом опрацювання задач, але взамін надає набагато гнучкіший сценарій обробки данних при якому ймовірність блокування потоку зводиться до мінімума, навантаження рівно розподіляється між усіма сервісами.

3 РЕАЛІЗАЦІЯ МАТЕМАТИЧНОЇ МОДЕЛІ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Інструменти реалізації

RabbitMQ та Nest.js ефективно поєднуються для реалізації мікросервісної архітектури через їхню взаємодію в асинхронному середовищі. RabbitMQ, як повідомлення-орієнтований брокер, сприяє надсиланню та отриманню повідомлень між різними компонентами додатку. Використання асинхронності в обох технологіях сприяє зменшенню затримок та покращенню масштабованості.

Nest.js, фреймворк для створення серверних застосунків на Node.js, підтримує асинхронний підхід та вбудовує принципи модульності та мікросервісної структури. Його можливості легкої інтеграції допомагають спрощувати роботу з різними частинами системи та підтримують принципи масштабованості.

Разом з тим RabbitMQ надає можливість організації мікросервісної архітектури через обмін повідомленнями, а Nest.js сприяє кращому керуванню кодом та розширюваності. Обидві технології допомагають легко інтегрувати різні частини системи та забезпечують ефективне управління повідомленнями та асинхронною обробкою.

3.1.1 Nest.Js

Основними програмними засобами під час розробки моделі обробки запитів за використання поштових брокерів повщень є Nest.Js та RabbitMQ.

Nest.js - це фреймворк для розробки серверних додатків на мові програмування TypeScript. Він базується на елементах Angular, що робить його досить зручним для розробки великих, масштабованих та легко тестуємих додатків. Основні особливості та можливості Nest.js включають:

- Модульність: Nest.js використовує концепцію модульності, що дозволяє

легко організовувати код у вигляді невеликих, відокремлених модулів.

- Dependency Injection: Використання ін'єкції залежностей допомагає полегшити тестування та розширення функціоналу додатку.
- Express та Fastify підтримка: Nest.js може працювати як повноцінний сервер на базі Express або Fastify, що відкриває доступ до багатьох плагінів та middleware цих фреймворків.
- WebSocket підтримка: Забезпечує підтримку WebSocket, що робить його ідеальним вибором для додатків, які потребують реального часу.
- Маршрутизація: Використовуючи декоратори, можна легко визначати маршрути та обробники запитів, що робить код більш читабельним.
- Мідлвари: Nest.js надає можливість використання мідлвар для обробки запитів на різних етапах їх обробки.

Щодо мікросервісної архітектури, Nest.js добре підходить через такі фактори:

- Модульність та Організація Коду: Завдяки концепції модулів, Nest.js сприяє організації коду у вигляді невеликих, відокремлених частин. Це особливо важливо в мікросервісній архітектурі, де кожен мікросервіс може бути розглянутий як окремий модуль.
- Масштабованість: Nest.js дозволяє легко масштабувати додатки, що робить його ідеальним для систем з багатьма мікросервісами.
- Декларативна Конфігурація: Nest.js використовує декларативний підхід до конфігурації, що полегшує розгортання та управління різними мікросервісами.
- Легка Інтеграція з Іншими Системами: Беручи підтримку від Express та Fastify, Nest.js може легко інтегруватися з іншими мікросервісами, які використовують ці фреймворки.

Основними компонентами в Nest.js є Controllers, Services та Modules також з додаткових компонентів можна виділити Pipes, Entities та Middlewares.

Controller - це компонент який відповідає за доступ до кінцевих точок отримання і відправки даних у ньому конфігурується шляхи до цих точок, та

реалізується логіка сервісів, приклад реалізації сервісу див. Рис. 3.1.

cats.controller.ts JS

```
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
```

Рис. 3.1 Реалізація компонента Controller

Наступним важливим компонентом є Service цей компонент відповідає за реалізацію бізнес логіки, в ньому проводиться робота з базою даних, складні обчислення та обробка даних, одним з найважливіших аспектів є Injectable підхід який дозволяє використовувати абсолютно різні сутності інших сервісів, таким чином якщо ми маємо 2 сервіси з однаковими методами але різною реалізацією, ми можемо швидко та зручно їх змінними не переписуючи при цьому лоігку інтеграції сервісу, це значно спрощує та пришвидшує впровадження змін, приклад реалізації сервісу див. Рис. 3.2.

cats.service.ts JS

```
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  create(cat: Cat) {
    this.cats.push(cat);
  }

  findAll(): Cat[] {
    return this.cats;
  }
}
```

Рис. 3.2 Реалізація компонента Service

Наступним і найважливішим компонентом є Module, модулі це компоненти які розбиваються за певними бізнес процесами, включають в себе сервіси та контролери, таким чином зберігається абстракція та композиція, основна задача визначити головні модулі системи і їх взаємодію між собою. Різні модулі модулі можуть включати в себе однакові сервіси таким чином можна перед використовувати код, що значно спрощує роботу, також робить цілісною роботу бізнес логіки, приклад реалізації модулю див. Рис. 3.3.

cats/cats.module.ts JS

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

Рис. 3.3 Реалізація компонента Module

Наступним компонентом є допоміжний компонент Pipes, самі Pipes можуть бути реалізовані як на рівні всього додатку так і на окремих методах контролерів, їх основною задачею є валідація даних, наприклад якщо ми хоче валідувати абсолютно усі запити то більш доречним буде включити цей пайп на основний рівень, якщо саме деякі то на рівні контроллера, також різні контролери можуть використовувати різні пайпи, тому на рівні всього додатку такі пайпи визначити неможливо, приклад реалізація пайпу див. Рис. 3.4.

cats.controller.ts JS

```
@Post()
@UsePipes(new ZodValidationPipe(createCatSchema))
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Рис. 3.4 Реалізація компонента Pipe

Останнім з найважливіших компонентів є Middlewares ці компоненти можуть бути включені абсолютно до любого рівня компонентів системи, їх задача зробити сайд-ефекти, такі як логування, додаткове збереження даних в окремо розподілені бази, нормалізація та серіалізація даних то що, реалізація Middlewares див. Рис. 3.5., діаграма переходу див. Рис. 3.6.

app.module.ts JS

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('cats');
  }
}
```

Рис. 3.5 Реалізація компонента Middleware

Рис. 3.6 Діаграма переходу станів при обробці Middleware

Взаємодія компонентів та структура проекту представлені на Рис. 3.7 - 3.8.

Так наприклад на на Рис. 3.7 можна побачити що деякі модулі можуть бути самостійними, а деякі можуть бути використані як допоміжні, також модулі можуть бути циклічно залежними, але в такому випадку необхідно перевірити конфігурацію їх підключення, та додати додатковий параметр який вкаже, на взаємозалежність цих модулів.

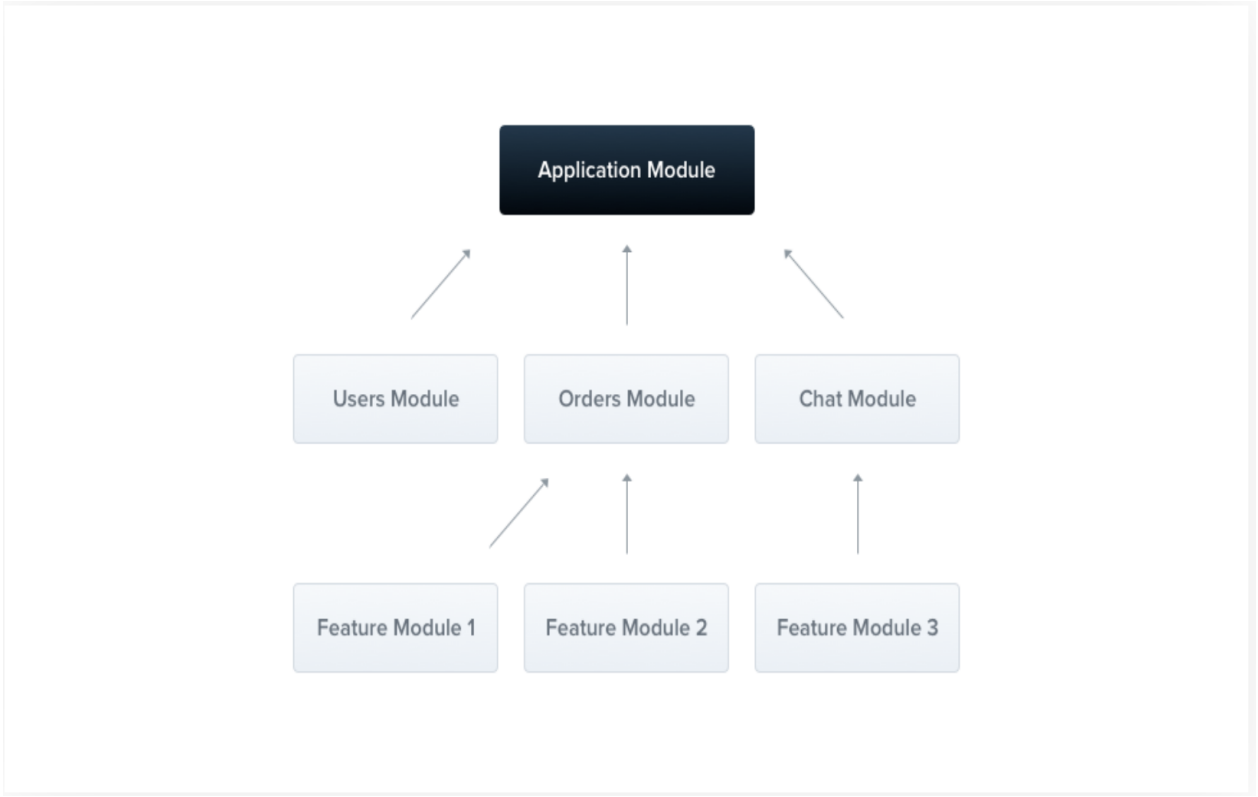


Рис. 3.7 Діаграма взаємодії компонентів

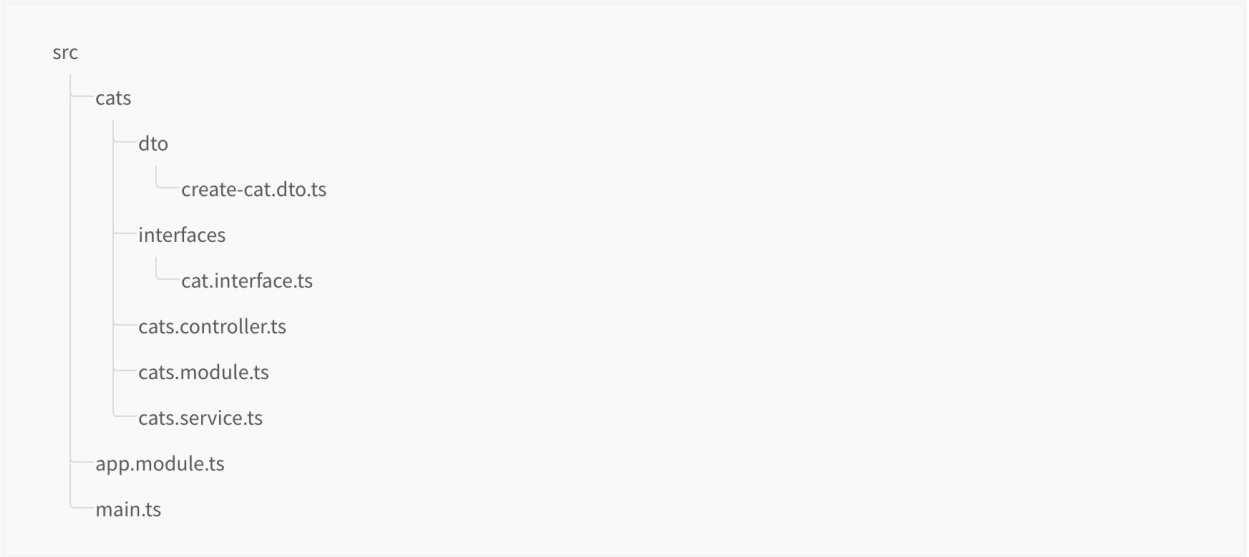


Рис. 3.8 Структура проекту

3.1.2 RabbitMQ

RabbitMQ - це популярний брокер повідомлень (message broker), який використовує протокол AMQP (Advanced Message Queuing Protocol). Основна його задача полягає в організації ефективного обміну повідомленнями між різними компонентами системи. В основі лежить концепція черги (queue), яка дозволяє одній частині системи відправити повідомлення, а іншій – його прийняти та обробити. Основний функціонал та можливості RabbitMQ включають:

- Черги (Queues): RabbitMQ дозволяє створювати різні черги для зберігання повідомлень. Це дозволяє вирішувати проблеми з різницею в швидкості обробки даних між виробниками та споживачами.
- Обмін повідомленнями (Message Exchange): Різні компоненти можуть взаємодіяти через обмін повідомленнями, що дозволяє гнучко налаштовувати логіку обробки повідомлень.
- Маршрутизація повідомлень: RabbitMQ підтримує різні стратегії маршрутизації повідомлень, що дозволяє гнучко керувати тим, як повідомлення доходять до черг та обмінів.
- Підтримка різних протоколів: Окрім AMQP, RabbitMQ підтримує інші протоколи комунікації, такі як MQTT, STOMP та інші.
- Доставка повідомлень з різними гарантіями: RabbitMQ підтримує різні рівні

надійності, такі як "at most once", "at least once", "exactly once", що дозволяє вибрати необхідний рівень гарантії доставки відповідно до вимог додатка.

- Гнучкість та Розширюваність: RabbitMQ може бути легко налаштований та розширений за допомогою плагінів. Це дозволяє використовувати його в різних випадках використання та інтегрувати з іншими технологіями.

Зважаючи на ці можливості, RabbitMQ ідеально підходить для розробки мікросервісних архітектур з кількома причинами:

- Локалізація помилок: Через використання черг і брокера повідомлень, помилки в одному мікросервісі не призведуть до негайної недоступності або втрати повідомлень. Вони можуть бути збережені у черзі та оброблені в подальшому.
- Масштабованість: RabbitMQ дозволяє гнучко масштабувати систему, додаючи нові мікросервіси або ефективно розподіляючи навантаження між ними.
- Гнучка маршрутизація: Можливість гнучкої маршрутизації повідомлень дозволяє ефективно взаємодіяти між різними компонентами системи, навіть якщо вони розгорнуті в різних місцях.
- Забезпечення надійності: RabbitMQ дозволяє налаштовувати рівні надійності, щоб відповідати вимогам конкретного застосунку.
- Інтеграція з іншими технологіями: RabbitMQ може легко інтегруватися з різними технологіями, що дозволяє використовувати його як частину більших архітектурних рішень.

Всі ці фактори роблять RabbitMQ потужним інструментом для створення розподілених систем і особливо підходять для мікросервісних архітектур.

На Рис. 3.9 представлена схема обробки повідомлень за впровадження поштових брокерів сповіщень.

Першочергово отримані дані з клієнтської частини потрапляють на основний сервер на якому визначається тип черги, далі повідомлення відправляється до поштового брокеру сповіщень, там визначається порядок опрацювання запиту згідно черги, наступний крок це опрацювання запиту

сервісом на який його передав поштовий брокер, і збереження або відправка даних до нової черги.

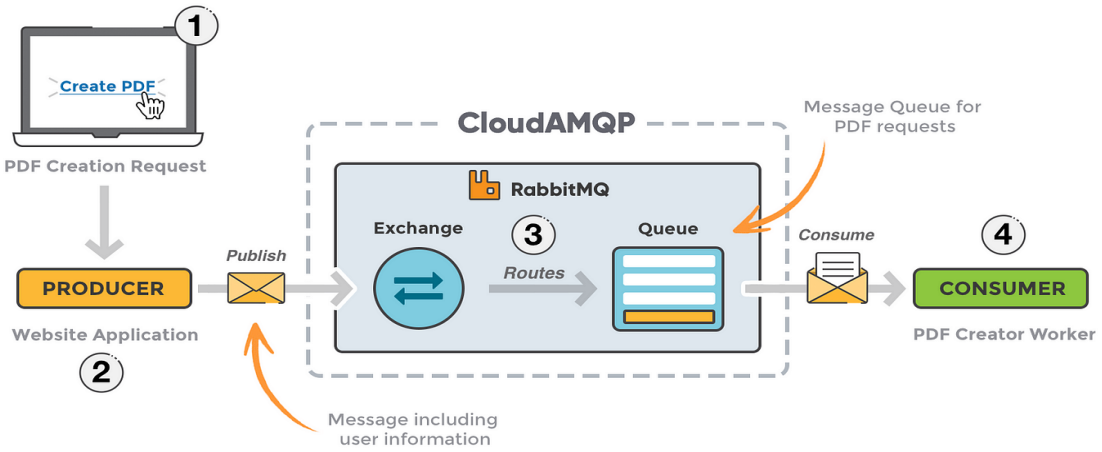


Рис. 3.9 Схема обробки повідомлень за допомогою поштових брокерів сповіщень

На Рис 3.10 представлена схема розподілення повідомлень по чергам, ця схема може змінюватись, в залежності від налаштувань сервісів, інколи черги можуть віддавати запити на один сервіс, якщо він підтримує кілька видів черг.

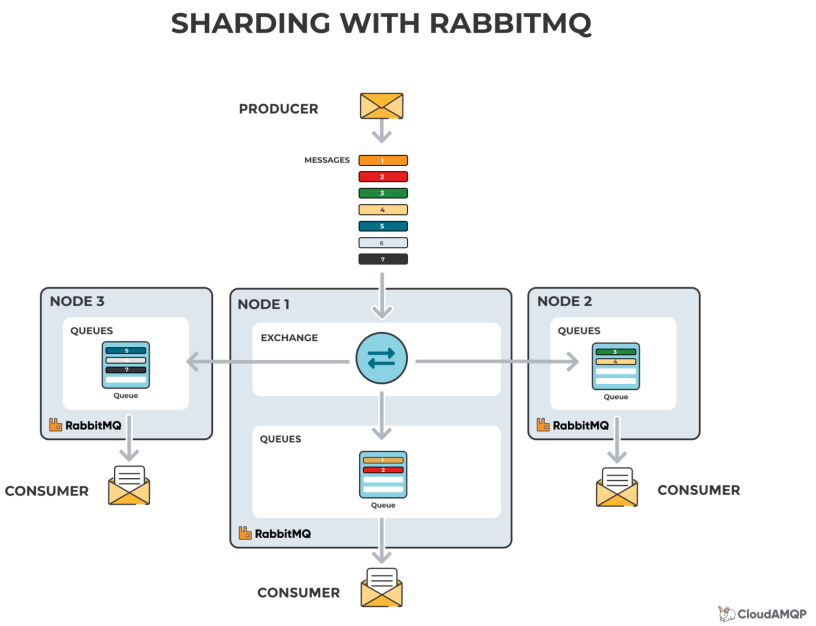


Рис. 3.10 Схема розподілення повідомлень у черзі

Також важливо зауважити, що поштові брокери сповіщень, гарно працюють з ізоляцією відказів див Рис. 3.11 та 3.12, на схемах відображено, поведінку сервісів під час обробки даних, брокер завжди чекає від сервісу повідомлення про успішне опрацювання даних, для видалення їх з черги та пам’яті, якщо запит був не опрацьований або опрацьований з помилкою, сервіс наділшде повідомлення до черги помилок з запитом, в майбутньому ці запити можна знову відправити на обробку або провести аналіз помилки яка сталась під час обробки.

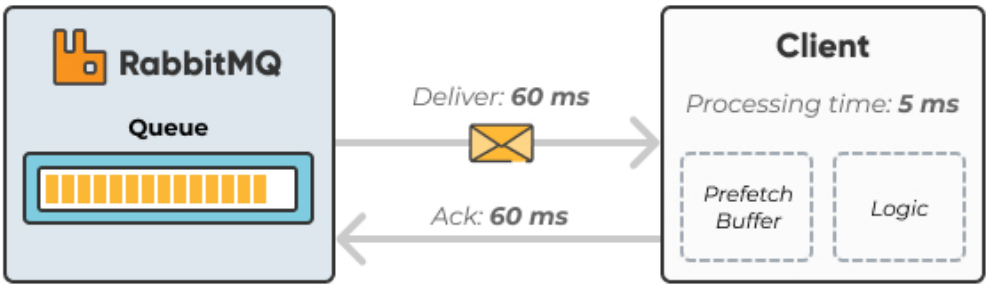


Рис. 3.11 Схема обміну повідомленнями між сервісом і поштовим брокером сповіщень

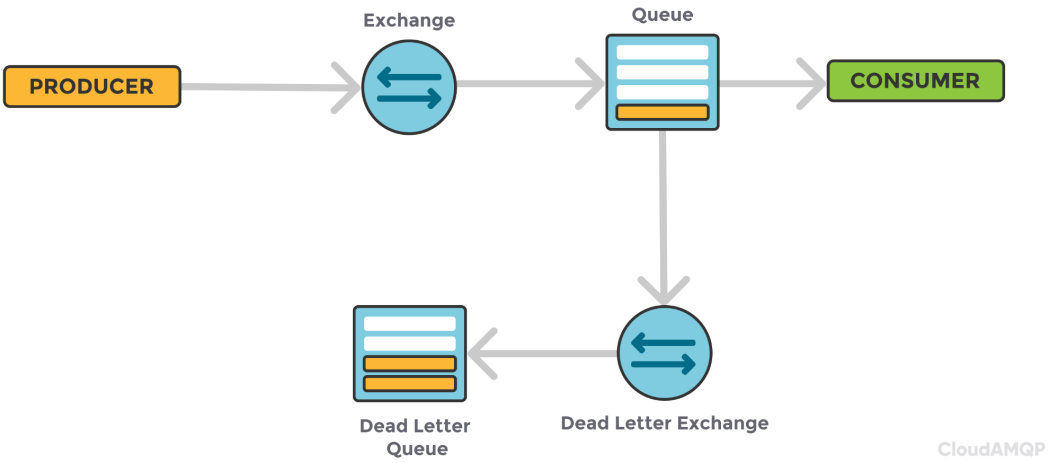


Рис. 3.12 Схема валідації запитів та відправки помилок обробки

На Рис 3.13 представлена діаграма класів RabbitMQ, з детальним описом всіх компонентів з яких він складається та їх методів.

Основними класами з якими відбувається взаємодія під час розробки є:

- IMessageService - використовується метод GetStatus для отримання статусу обробки запиту, або кількості запитів які наразі знаходяться в обробці;
- IMessage - є об’ктом який передається з сервісу до поштового брокеру сповіщень;
- IMessageQueueClient - класс за допомогою якого відкривається з’єднання між сервісом та брокером, також через цей клас відбувається передача статусу обробки запиту, та викликаються події отримання даних, їх розміщення та видалення;

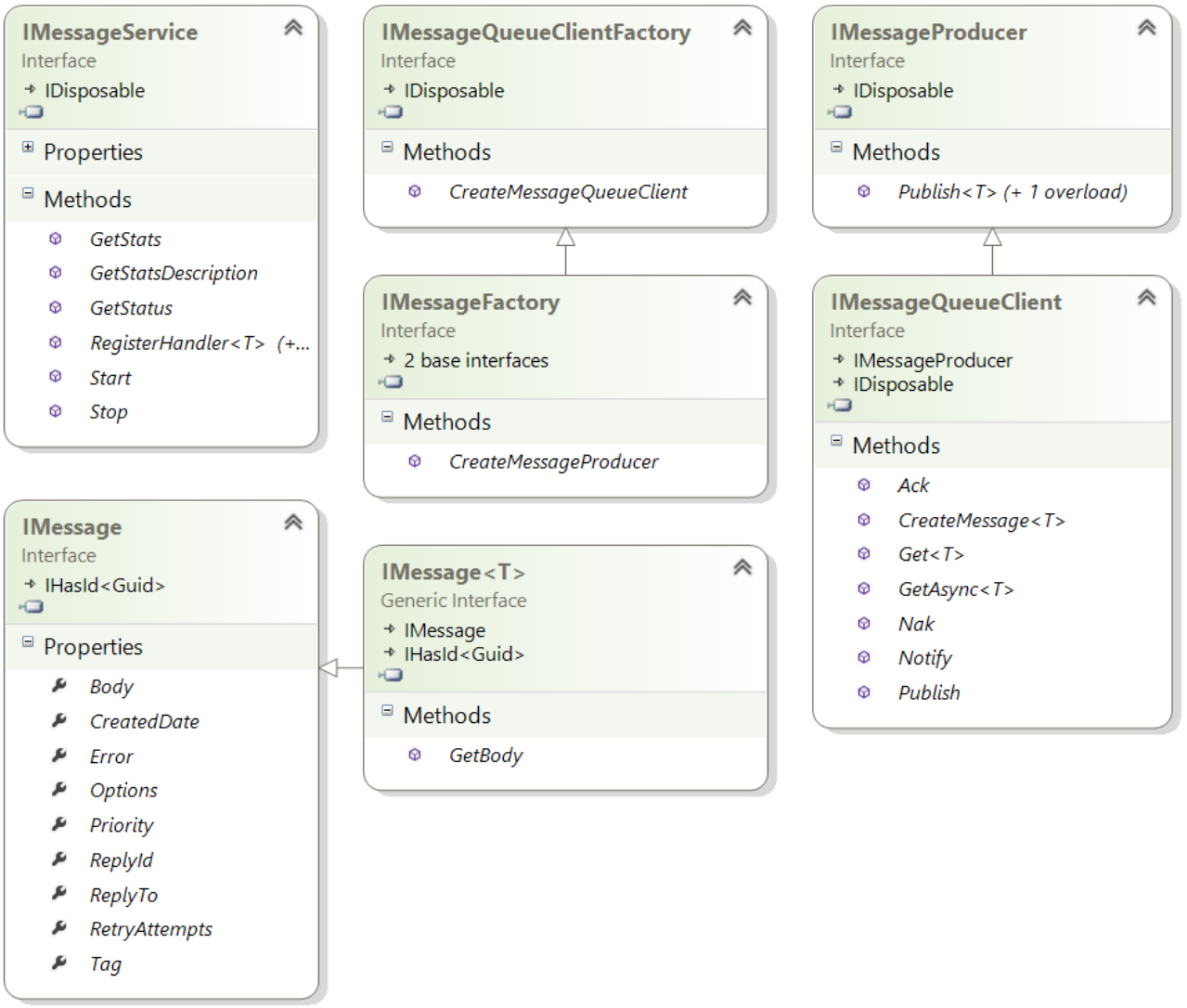


Рис. 3.13 Діаграма класів RabbitMQ

3.2 Опис структури проекту та ключових файлів

Оскільки архітектурою додатку було обрано мікросервісну, було розроблено декілька сервісів та основний сервер який отримує клієнтські запити віддає статичні файли та дані за запитами, сервіси виконують обробку певних запитів.

Підходом до організації коду обрано mono-repo, оскільки в розробці проекту приймала участь маленька команда, і всі сервіси мають перевикористовувані сервіси, такий підхід до організації є повністю виправданим, він скорочує час розробки через зменшення необхідності виносити кожну пере використовувати функцію в окремий пакет, та має зручну конфігурацію для розгортання усього проекту одразу, структура проекту показана на Рис. 3.14.

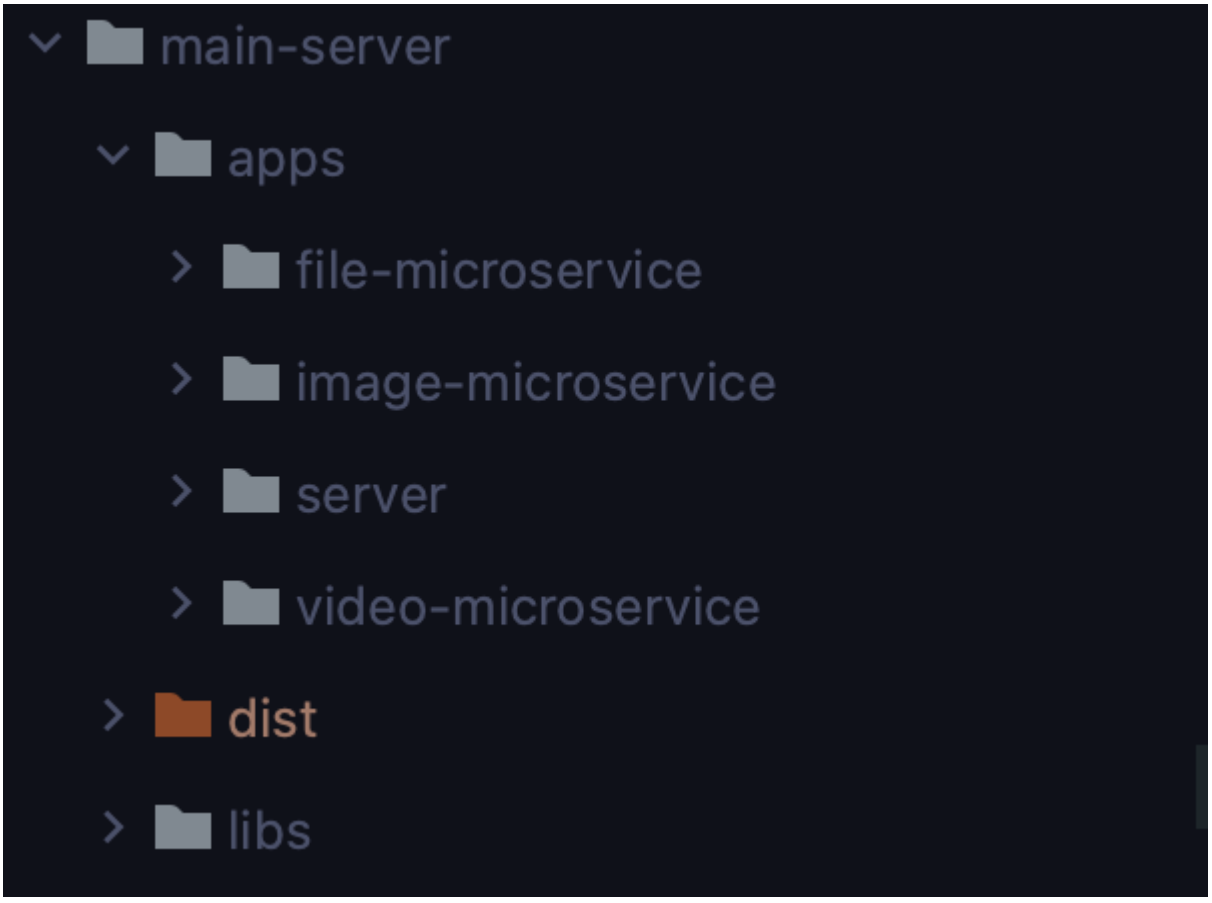


Рис. 3.14 Структура проекту

На Рис. 3.15. представлений main.ts файл сервісу, він відповідає за запуск

сервісу, та приймає налаштування поштового брокеру сповіщень, у вигляді домену розміщення брокера, назви черги яку він опрацьовує, кількості запитів, та можливості повторної обробки не опрацьованих вбо відхилених запитів всі ці параметри передаються через строку запуску сервера для більшої легкості масштабування та гнучкості розподілення навантаження.

```
import {Transport} from "@nestjs/microservices";
import {FileMicroserviceModule} from "./file-microservice.module";
import * as process from "process";

no usages  Dmytrii Prykhodko *
async function bootstrap() {
  const port = process.env.PORT || 4001
  const app = await NestFactory.createMicroservice(FileMicroserviceModule, options: {
    transport: Transport.RMQ,
    options: {
      urls: ['amqps://whnfsqgc:leXnBeQYDzH8xX0trx9cnqMbFRchwJ63@sparrow.rmq.cloudamqp.com/whnfsqgc'],
      queue: `link_queue_${process.env.GRADE || 1}`,
      port,
      queueOptions: {
        durable: false
      },
      consumerOptions: {
        prefetchCount:process.env.PREFETCH_COUNT || 5 ,
      },
    },
  });

  await app.listen().then(()=>{
    console.log("Microservice listening")
  }).catch(()=>{
    console.log("Something went wrong with microservice")
  })
}
```

Рис. 3.15 Файл main.ts для запуску сервісу

На Рис. 3.16 представна конфігурація основного моду, головного сервісу. Конфігурація включає типи черг на які основний сервер буде віддавати повідомлення до поштового брокера сповіщень, також цей сервіс включає основні сутності для роботи з базою даних, та визначені для нього контроллери для отримання запитів з кілєнтськох частини, і сервіси які роблять попередню обробку даних для визначення черг.

```
4 usages   Dmytrii Prykhodko *
@Module( metadata: {
  imports: [
    ClientsModule.register( options: [
      {
        name:"LINK_QUEUE_1",
        transport:Transport.RMQ,
        options:{
          urls: ['amqps://whnfsqgc:leXnBeQYDzH8xX0trx9cnqMbFRchwJ63@sparrow.rmq.cloudamqp.com/whnfsqgc'],
          queue: `link_queue_1`,
        }
      },
      {
        name:"LINK_QUEUE_2",
        transport:Transport.RMQ,
        options:{
          urls: ['amqps://whnfsqgc:leXnBeQYDzH8xX0trx9cnqMbFRchwJ63@sparrow.rmq.cloudamqp.com/whnfsqgc'],
          queue: `link_queue_2`,
        }
      },
      {
        name:"LINK_QUEUE_3",
        transport:Transport.RMQ,
        options:{
          urls: ['amqps://whnfsqgc:leXnBeQYDzH8xX0trx9cnqMbFRchwJ63@sparrow.rmq.cloudamqp.com/whnfsqgc'],
          queue: `link_queue_3`,
        }
      }
    ]),
```

Рис. 3.16 Опис конфігурації модуля головного серверу

Рис 3.17 представляє опис контролера для отримання даних з клієнтської частини, має метод uploadFile який спрацьовує коли користувач завантажує файл, далі реалізує метод для визначення черги, і в залежності від визначених черги відправляє повідомлення до поштового брокера, на даному скріншоті також продемонстровано реалізацію Injectable паттерну, таким чином наприклад clientOne, clientTwo та clientThree можуть бути замінені абсолютно на любі інші сервіси які мають такі самі методи, Injectable паттерн є неймовірно зручним рішенням, для реалізації класів оскільки дозволяє швидко змінювати необхідні классы на інші при цьому не змінюючи основну кодову базу класу в який вони були запроваджені.

```
@Controller( prefix: 'link-handler')
export class LinkHandlerController {
    no usages
    constructor(private assetService:AssetService, @Inject( token: 'LINK_QUEUE_1') private readonly clientOne:ClientProxy,
                no usages
                @Inject( token: 'LINK_QUEUE_2') private readonly clientTwo:ClientProxy,
                no usages
                @Inject( token: 'LINK_QUEUE_3') private readonly clientThree:ClientProxy) {
    }

    no usages
    @Post( path: "download")
    uploadLink(@Body() linkDto:LinkDto){
        const queue : string = getQueue()
        switch (queue) {
            case "LINK_QUEUE_1":
                this.clientOne.send(queue,linkDto)
                break;
            case "LINK_QUEUE_2":
                this.clientTwo.send(queue,linkDto)
                break;
            case "LINK_QUEUE_3":
                this.clientThree.send(queue,linkDto)
                break;
        }

    }
}
```

Рис. 3.17 Опис контролера для отримання даних з клієнтської частини

Як вже було зазначено в розділі 2.4 визначення черги відбувається за врахуванням типу черги тобто операції яку треба провести на данимим та від об’єму даних які потрібно обробити, тому можна ввести граничні значення велчини данних, вони можуть варіюватись від бізнес правил та від кількості наявного ресурсу для роботи сервісів.

Визначивши об’єм даних можна розрахувати градацію, та сформувати тип черги як суму рядків назва черги та її градація, роботу методу визначення черги представлено на Рис. 3.18.

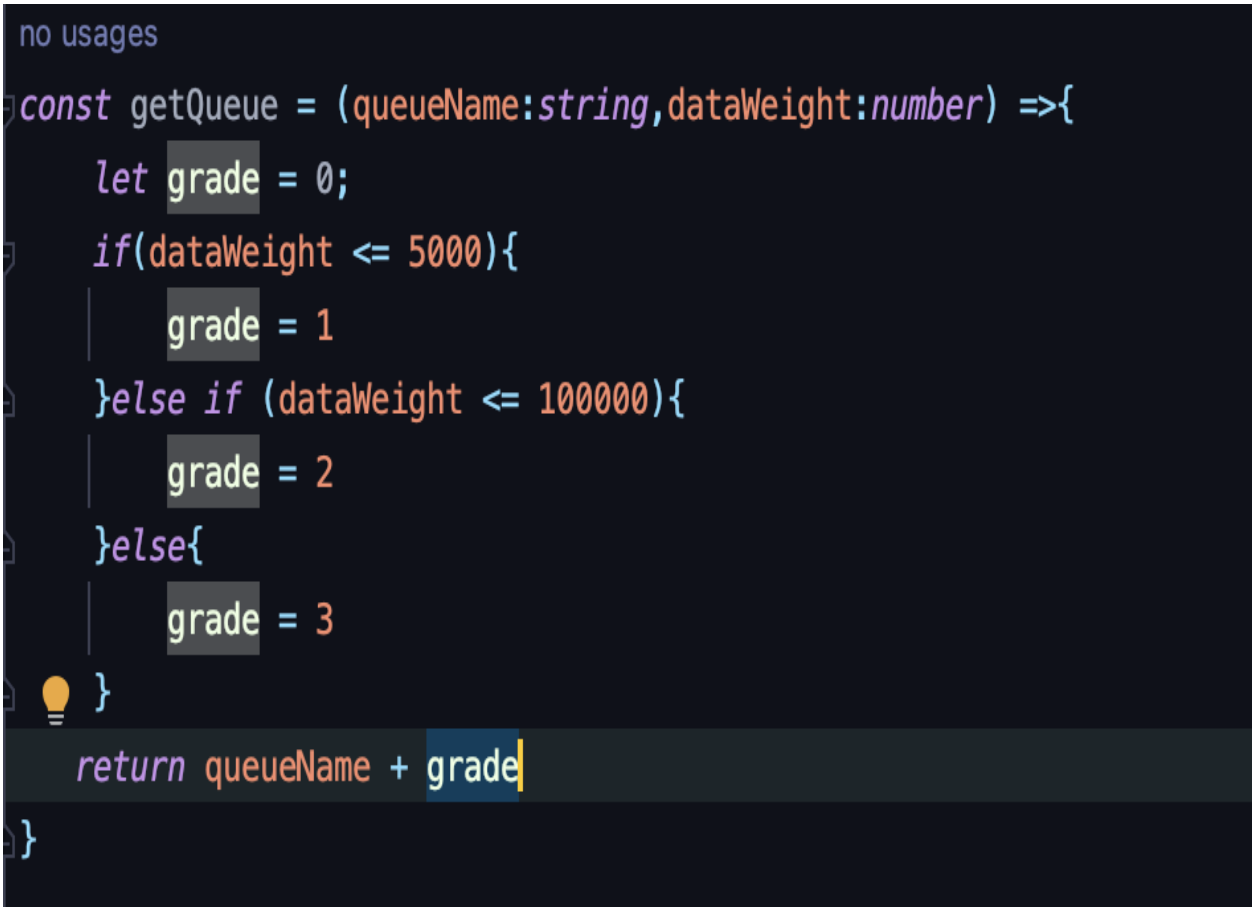


Рис. 3.18 Опис методу визначення черги

Після відправки повідомлень до поштового брокера сповіщень, до сервісів які підтримують відповідні типи черг, та виконує тип операції зазначений для даного типу черги, так на Рис. 3.19 представлений файл контролера сервіса який приймає відправлені файли, та обробляє їх, також цей сервіс зберігає сам файл у статичну папку основного сервер для подальшої роботи з ним, крім самого файлу сервіс може приймати посилання на скачування, за такого сценарію запровадження мікросервісна архітектури є дуже важливим, оскільки посилання можуть містити шкідливі файли, або скрипти ін’єкції для отримання або видалення даних з бази, оскільки сервіс сам на пряму не займається збереженням даних і є ізольованим від інших сервісів, такі сценарії ніяким чином не вивезуть систему з ладу.

```
1 usage
@MessagePattern("download_asset")
async downloadAsset (fileUrl:string):Promise<string>{
    const response = await axios.get(fileUrl,  config: { responseType: 'stream' });
    const contentDisposition = response.headers['content-disposition'];
    const fileSize = parseInt(response.headers['content-length'],  radix: 10);
    const fileType = response.headers['content-type'];
    let fileName1 = 'downloaded-photo'; // Default filename
    if (contentDisposition) {
        const matches = contentDisposition.match(/filename="(.*?)"/);
        if (matches && matches[1]) {
            fileName1 = matches[1];
        }
    }
    console.log(fileName1,fileType,fileSize)
    const fileName = uuidVersionFour() + '.jpg';
    const filePath = path.resolve(__dirname, '..', 'static')
    if (!fs.existsSync(filePath)) {
        fs.mkdirSync(filePath,  options: {recursive: true})
    }
    response.data.pipe(fs.createWriteStream(path.join(filePath, fileName)));

    return ""
}
```

Рис. 3.19 Файл контроллера сервісу для обробки файлів

3.3 Опис розробленої схеми розміщення та взаємодії компонентів

Оскільки способом розгортання та підтримки компонентів було обрано контейнериацію, то однією з найважливіших задач є визначення розміщення компонентів у контейнерах, для більш зручної взаємодії та розгортання компонентів, було розроблено схему розміщення компонентів, яка включає в себе такі компоненти як основний сервер на яку розміщуються, сервіси, база даних, поштовий брокер сповіщень і головний сервіс.

Всі сутності розбиваються по власним Docker контейнерам, ці контейнери дозволяють досягти ізольованості кожного модуля, тобто не залежності від інших,

таким чином при виході з ладу одного модуля чи контейнера, всі інші частини будуть надалі функціонувати, також такий підхід дозволяє, більш зручно розгортати ці сервіси та виділяти на них ресурси.

Під час вибору способу збереження даних (використовуючи одну базу для всіх сервісів, або окрему базу для кожного) було розглянуто обидва підходи та визначення їх переваги.

Одна загальна база для всіх мікросервісів:

- Зручність управління:

Загальна база даних спрощує управління та підтримку, оскільки необхідно слідкувати тільки за однією системою.

- Легкість узгодження:

Для деяких сценаріїв це може полегшити узгодження даних між різними мікросервісами.

Окрема база для кожного мікросервісу:

- Незалежність:

Кожен мікросервіс має власну базу, що забезпечує йому повну незалежність в управлінні своїми даними.

- Гнучкість та масштабованість:

Дозволяє гнучко змінювати та масштабувати окремі частини системи незалежно одна від одної.

- Схильність до помилок:

Окремі бази даних можуть зменшити вплив помилок в одному сервісі на інші.

- Забезпечення консистентності:

Вибір повинен враховувати необхідність забезпечення консистентності даних між мікросервісами.

Проаналізувавши дані підходи було обрано використовувати одну базу даних, та розмістити її в одному контейнері з основним сервером для зменшення часу відгуку на обробку запиту також таким чином легко досягти узгодженості даних так як всі сервіси та основний сервер мають лише одне джерело правди.

Всі інші сутності було винесено до окремих контейнерів, для більш зручного розгортання та обмеженню залежності роботи одного сервісу від іншого,

схема розміщення компонентів представлена на Рис. 3.22, взаємодія компонентів див. Рис. 3.23.

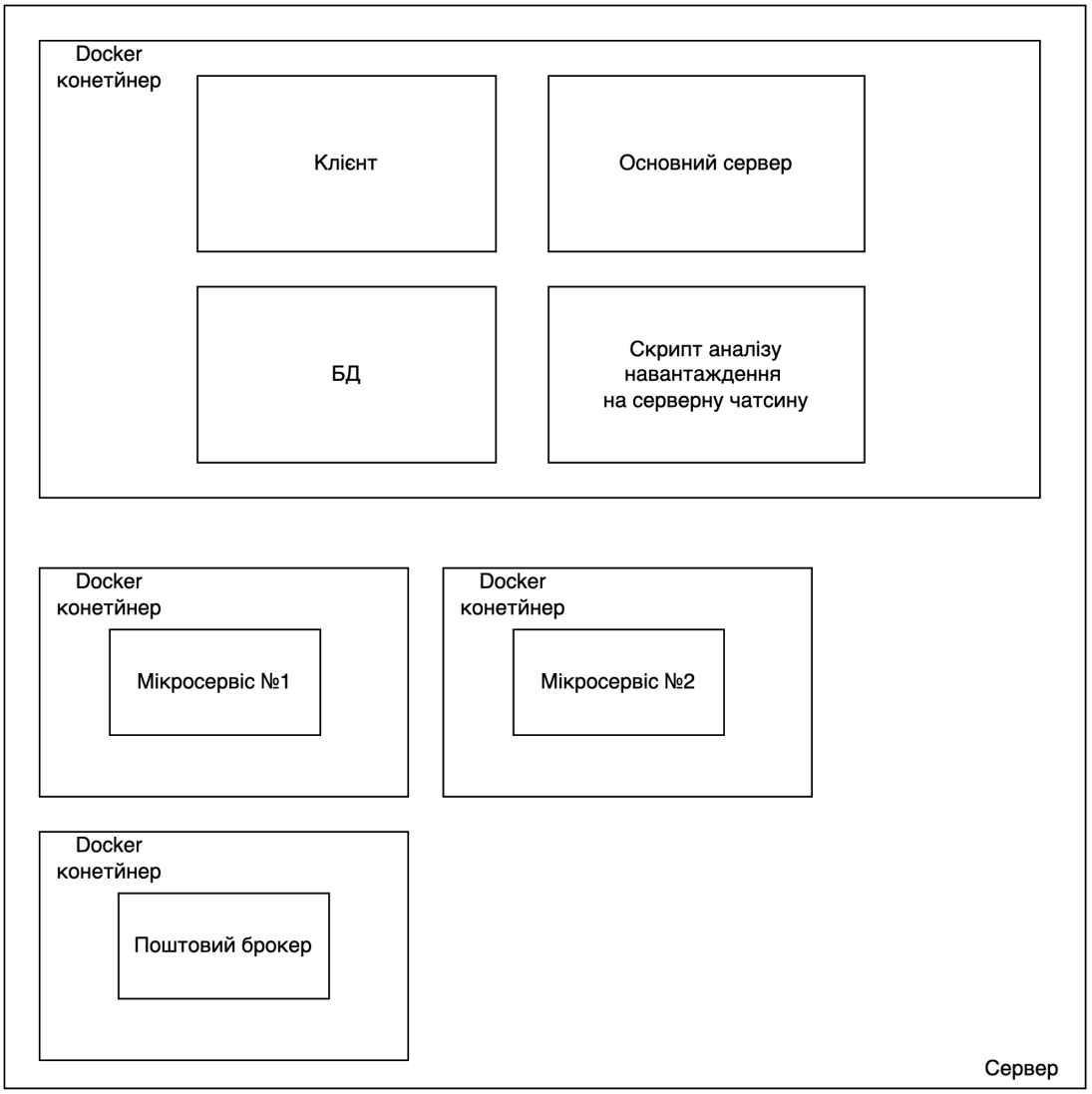


Рис. 3.20 Схема розміщення компонентів системи

Контейнеризацію було обрано як спосіб розгортання компонентів оскільки вона надає можливість швидко розгортати сервіси як на власному сервері так і на віддаленому, оскільки при розміщенні, під кожен контейнер буде запущено віртуальну машину та встановлено, саме ті залежності які необхідні для функціонування даного сервісу, також в майбутньому таку архітектуру можна розміщувати на хмерних сервісах які підтримують контейнеризацію, це може зменшити кількість витрат на обслуговування даних сервісів та їх підтримку.

Контейнеризація легко може мігрувати до оркестраційного рівня, при введенні Kubernetes, основною задачею якого є оркестурвати надані контейнери тобто, виділяти необхідну па’мять під функціонування сервісів, займатись процесами відновлення стабільної роботи сервісу, вмикати та вимикати додаткові сутності сервісів, проте оркестрація вимагає значної експертизи в даному напрямку, і може бути не доречною на стадіях розробки або при розгортанні MVP версії системи.

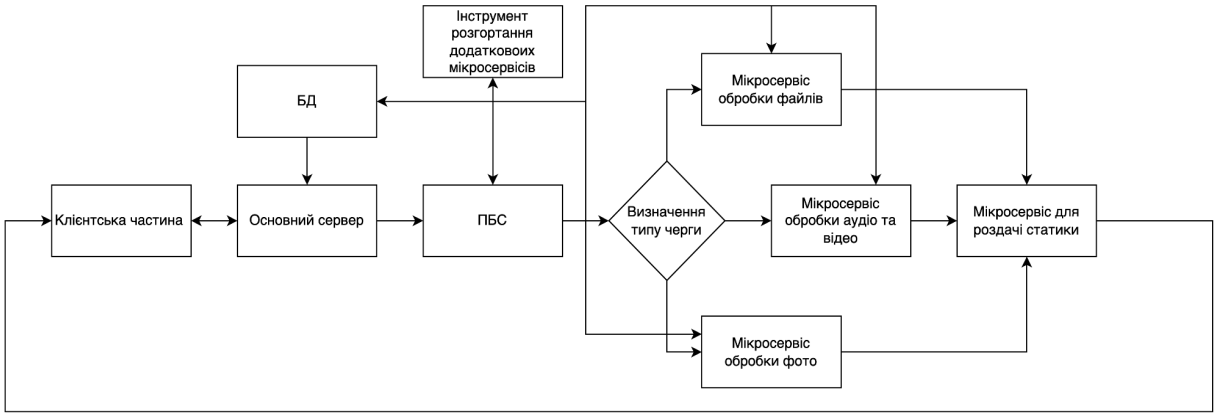


Рис. 3.21 Блок-схема взаємодії компонентів

3.4 Аналіз результатів

Для отримання даних для аналізу було виконано 50 тестових запитів на сервер з протоколом обміну даних REST та однією сутністю кожного сервісу, та 50 запитів на сервер з протоколом обміну даних AMQP та за використання поштового брокера сповіщень RabbitMQ, програмну модель для обробки запитів описано в розділі 2.6, також на Рис. 3.24 наведено блок схему алгоритму обробки запитів, яка детально описує логіку обробки запитів мікросервісами та розподілення запитів у чергах.

Основні показники швидкості обробки запитів було підвищено завдяки таким аспектам як, зміна протоколу обміну даними, AMQP протокол показав себе значно краще, оскільки він зменшує час відгуку сервера завдяки одноразовому відкриттю TCP з’єднання між сервісами та поштовим брокером, в той час як REST при кожному новому запиті відкриває нове з’єднання між сервісами.

Масштабованість було збільшено за допомогою ввдення додаткової сутності мікросервісів, оскільки з’єднання RabbitMQ з сервісами не вимагає від останнього знати про кількість і розташування сервісів, процес налаштування та підключення додаткових сутностей не піднімає нових викликів, та є доволі гнучким та прозорим, в той час як для REST протоколу було б необхідно задіяти додатковий компонент в вигляді Load Balancer, який вимагає налаштування конкретної кількості сутностей сервісів та має знати точне розташування кожного з них.

Ізоляція відмов також була покращена введенням RabbitMQ оскільки він нативно підтримує збереження отриманих запитів , також всі помилки під час обробки можна покласти в додаткову чергу помилок і за можливості знову їх обробити.

Доступність було збільшено за допомогою розробки програмної моделі обробки запитів, яка збільшує ефективність обробки за допомогою, винесення складних операцій на окремі сервіси які не блокують поток обробки інших більш легких запитів, таким чином всі користувачі отримують результат рівномірно та з мінімальною затримкою.

Для відправки запитів використовувалась програма Postman оскільки клієнтської частини не було в розробці, приклад відправки запиту див Рис. 3.25.

Детальні результати наведено в Таблиці 3.1, всі розрахунки проводились згідно формул визначених в розділі 2.2.

Рис. 3.22 Блок-схема алгоритму обробки запитів

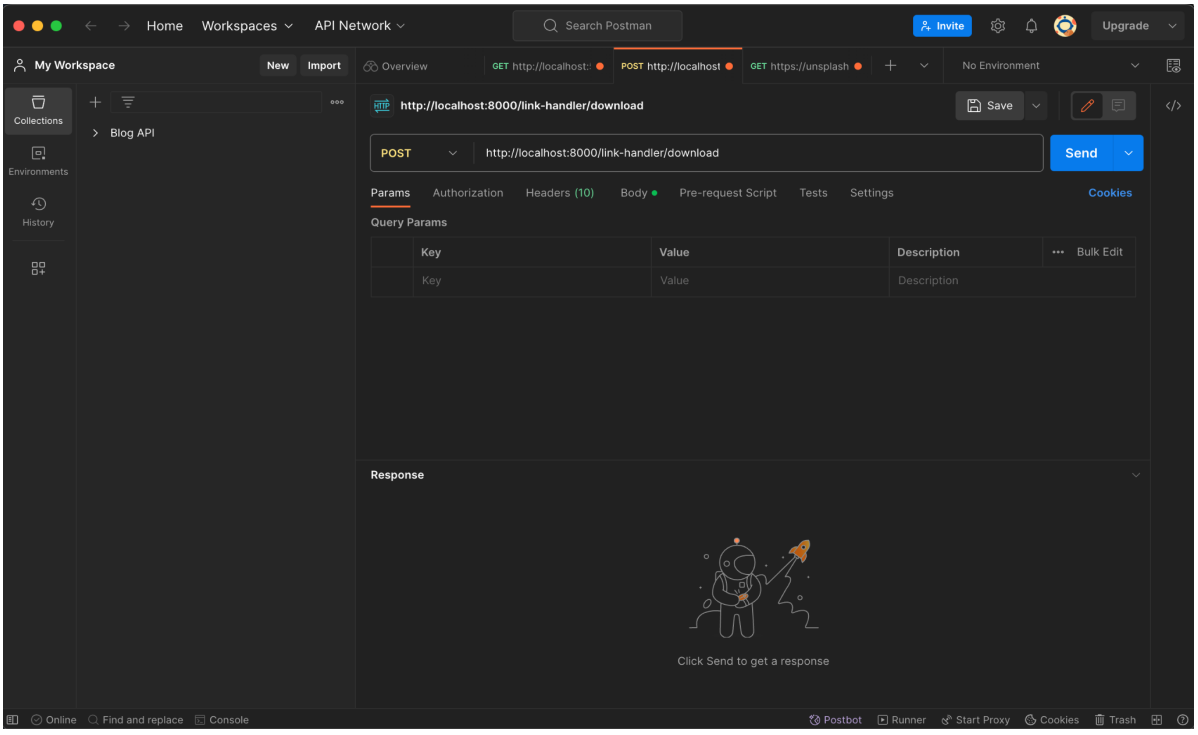


Рис. 3.23 Скріншот відправки запитів з Postman

Таблиця 3.1

Результати моделювання обробки запитів з використанням технології RabbitMQ

Показники	REST	RabbitMQ
Час відгуку (сек.)	0.6	0.5
Масштабованість (кількість сервісів)	1	2
Навантаження (кількість запитів)	20	20
Ймовірність виходу з ладу сервісу (від 0 до 1)	0.5	0.5
Ізоляція відмов (від 0 до 1)	0	1
Ефективність розподілення навантаження (час на обробку запитів в сек.)	12	5
Доступність сервісів	0.5	1

ВИСНОВКИ

В процесі виконання магістерської роботи, що включала в себе дослідження пов’язані з темою роботи, а саме використання мікро сервісної архітектури та поштових брокерів сповіщень як спосіб комунікації між сервісами для оптимізації обробки запитів, було досягнуто наступних результатів:

1. Проаналізовано існуючі методи комунікації між мікро сервісами з подальшим визначенням їх основних недоліків та переваг;
2. Розроблено програмну модель для визначення черг запитів та максимальної кількості запитів для розподілення навантаження;
3. Розробено функціональну модель обробки запитів між сервісами.
4. Розроблено схему розміщення та взаємодії системних компонентів;
5. Покращено час відгуку між сервісами та швидкість обробки запитів;
6. Підвищено рівень доступності та продуктивності сервісів за допомогою введення додаткових сутностей сервісів;

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація «AMQP (Advanced Message Queuing Protocol)» [Електронний ресурс] - Режим доступу: https://www.amqp.org/ 18.10.2023
2. Документація «RabbitMQ»:[Електронний ресурс] - Режим доступу: https://www.rabbitmq.com/ 18.10.2023
3. Сатаття «Why Google Stores Billions of Lines of Code in a Single Repository»: авторство - Rachel Potvin and Josh Levenberg [Електронний ресурс] - Режим доступу: https://dl.acm.org/doi/pdf/10.1145/2854146 18.10.2023
4. Книга "Мікросервіси. Патерни розробки та рефакторинг" (Microservices: Design Patterns and Refactoring): авторство - G. Макклелланд, М. Джонсон, С. Хуттон.
5. Книга "Шаблони Мікросервісів" за авторства Кріс Річардсон
6. Книга "Мікросервіси З Першого Дня" за авторства Кловес Карнейро та Тім Шмельмер
7. Книга "Від Моноліту до Мікро Сервісів: Еволюційні Патерни для Трансформації Моноліту" за авторства Сем Ньюмен
8. Книга "Архітектура Мікросервісів: Зробіть Крок до Мікросервісів" за авторства Кріс Річардсон
9. Книга "RabbitMQ У Глибину" за авторства Ґевін М. Рой
10. Книга "Вивчення RabbitMQ" за авторства Мартін Тошев
11. "Впровадження архітектур мікросервісів" Боріс Шоль, Лео Штудер, Майк Амундсен
12. "Контейнеризація мікросервісів: Навчіться розгортати, управляти та масштабувати додатки мікросервісів за допомогою Docker" Алок Шріваства
13. "Мікросервіси: Запуск та робота" Ронні Мітра, Іраклі Надарейшвілі, Метт Макларті, Майк Амундсен
14. "Шаблони проектування сервісів: Фундаментальні рішення для SOAP/WSDL та RESTful веб-сервісів" Роберт Деньно
15. RabbitMQ: Шаблони для застосунків" Дерік Бейлі

16."RabbitMQ: Розуміння поштових брокерів" Джейсон Дж.В. Вільямс
17."Володіння RabbitMQ" Емрах Аяноглу
18."Мікросервіси: Запуск та Розвиток" Ронні Мітра, Іраклі Надарейшвілі, Метт МакЛарти
19."Проектування Розподілених Систем" Брендан Бернс
20."Готові до Виробництва Мікросервіси" Сюзан Фаулер
21."Впровадження Мікросервісів" Боріс Шолл, Філіп Ґадір
22."Сервісно-Орієнтована Архітектура: Концепції, Технології та Дизайн" Томас Ерл
23."RabbitMQ в Дії" Альваро Відела, Джейсон Дж.В. Вільямс
24."Основи RabbitMQ" Девід Доссо

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

(Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Розробка моделі обробки запитів до серверу з мікросервісною архітектурою на основі технології RabbitMQ»

Виконав: студент групи ПДМ-64 Приходько Дмитрій Анатолійович

Керівник: : к.п.н., доцент, зав. кафедри ІТ Корецька Вікторія Олександрівна

Київ - 2024

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація процесів обробки запитів за впровадження мікросервісної архітектури.

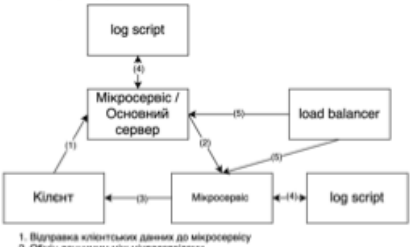
Об'єкт дослідження: масштабування та розподілення навантаження на серверну частину додатку.

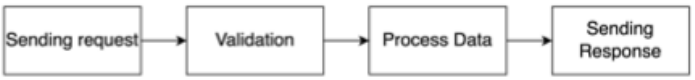
Предмет дослідження: методи та підходи до реалізації мікросервісної архітектури.

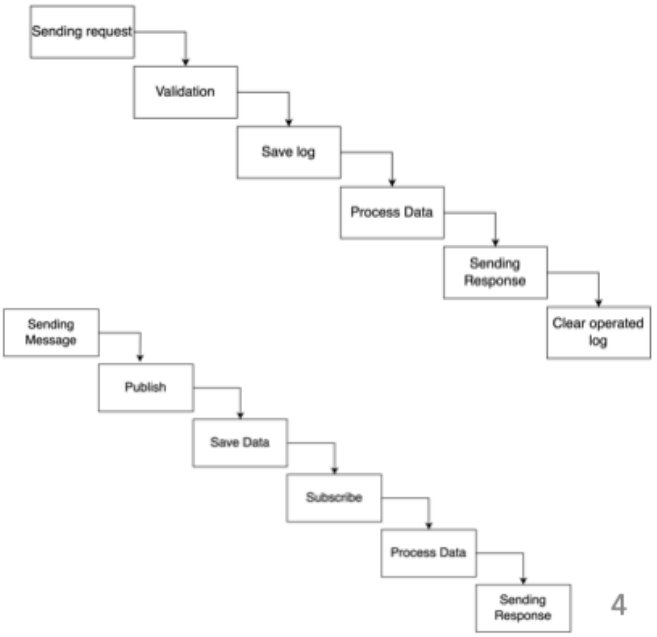
ПОРІВНЯЛЬНА ТАБЛИЦЯ ТЕХНОЛОГІЙ ТА ПРОТОКОЛІВ КОМУНІКАЦІЇ МІЖ МІКРОСЕРВІСАМИ

Назва технології / протоколу	Переваги	Недоліки
HTTP / Rest API (протокол)	• Стандартний та широко використовуваний протокол. • Легко взаємодіяти з іншими технологіями та інструментами. • Простота реалізації та розуміння.	• Може виникнути накладні витрати через надмірні запити. • Велика залежність від мережі.
GraphQL (техологія)	• Гнучка модель запиту-відповіді. • Зменшення кількості запитів через вибіркове отримання даних. • Легше виявлення і усунення зайвих даних.	• Вивчення та впровадження може вимагати додаткових зусиль. • Підвищена складність валідації запитів та доступу.
Message Brokers / MQTT (протокол)	• Легкий та ефективний протокол для обміну повідомленнями. • Низькі накладні витрати та швидка доставка повідомлень. • Підтримка як синхронної, так і асинхронної комунікації.	• Має набагато вужчий функціонал ніж AMQP протокол
Message Brokers / AMQP (протокол)	• Асинхронна комунікація • Здатність до обробки великої кількості повідомлень • Робота з чергами • Ізоляція відмов	• Збільшення складності системи через введення посередника.

ДІАГРАМИ ПЕРЕХОДУ ТА ВЗАЄМОДІЇ СИСТЕМНИХ КОМПОНЕНТІВ



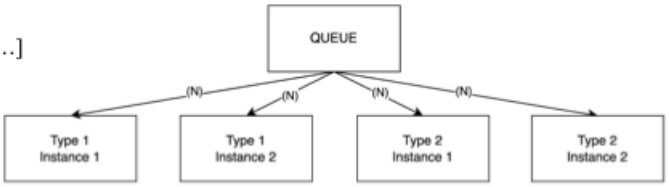




ПРОГРАМНА МОДЕЛЬ ВИЗНАЧЕННЯ ЧЕРГ ЗАПИТІВ

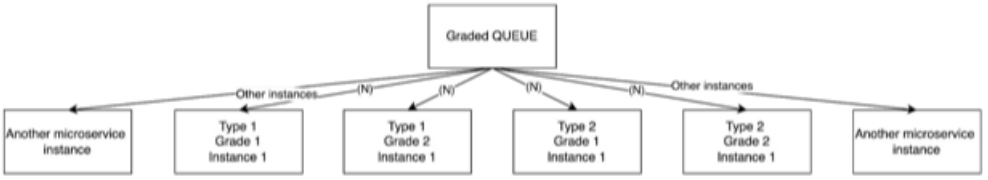
Стандартна модель визначення черги запиту

Типи операцій : operationTypes = [Type1, Type 2 …]
Кількість черг : operationTypes.length
Визначення назви черги: operationType[n]
Визначення кількості запитів для одночасної обробки залежить від кількості ресурсів сервісу.



Розроблена модель визначення черги запиту

Тип операції : operationTypes = [Type1, Type 2 …]
Складність операції: operationComplexity = [opComp1, opComp2 …]
Розмір вхідних даних: dataWeight = [dataWeight1, dataWeight2 …]
Визначення градацій: gradeTypes = getGrade(dataWeight, operationComplexity) → [gradeType1, gradeType2 …]
Кількість черг: operationTypes.lenght * gradeTypes.lenght
Визначення назви черги: operationType[n] + gradeTypes[n]
Визначення кількості запитів для одночасної обробки залежить від кількості ресурсів сервісу та градації.



5

ФОРМАЛЬНА МОДЕЛЬ ОБРОБКИ ЗАПИТІВ

Модель для REST:

$req = \{URI, Method, Headers, Body\}$

$Validation(req) \rightarrow \{true, false\}$

$Endpoint(req) \rightarrow res \in Res$

Модель для Message Brokers:

$QueueAnalize(Data) \rightarrow msg$

$msg = \{Body, Headers, Queue\}$

$Publish(msg) \rightarrow Queue$

$SaveToStorage(Queue) \rightarrow Queue$

$Subscribe(Queue) \rightarrow Components$

$Process(msg) \rightarrow Result$

6

ОЦІНКА ЕФЕКТИВНОСТІ ОБРОБКИ ЗАПИТІВ

Критерії ефективності:

$T_{R,M}$ - час відгуку (в секундах);

$S_{R,M}$ - масштабованість (кількість серверів);

L - навантаження (кількість запитів);

$F_{R,M}$ - ймовірність виходу сервісу з ладу (від 0 до 1);

$Q_{R,M}$ - ізоляція відмов(від 0 до 1);

де R - REST, а M - Message Brokers

Розрахунок ефективності розподілення навантаження:

$$E_M = \frac{L}{S_M} * T_M \quad 1)$$

$$E_R = L * T_R \quad 2)$$

Розрахунок доступність сервісів:

$$A_M = 1 - F_M * S_M \quad 3)$$

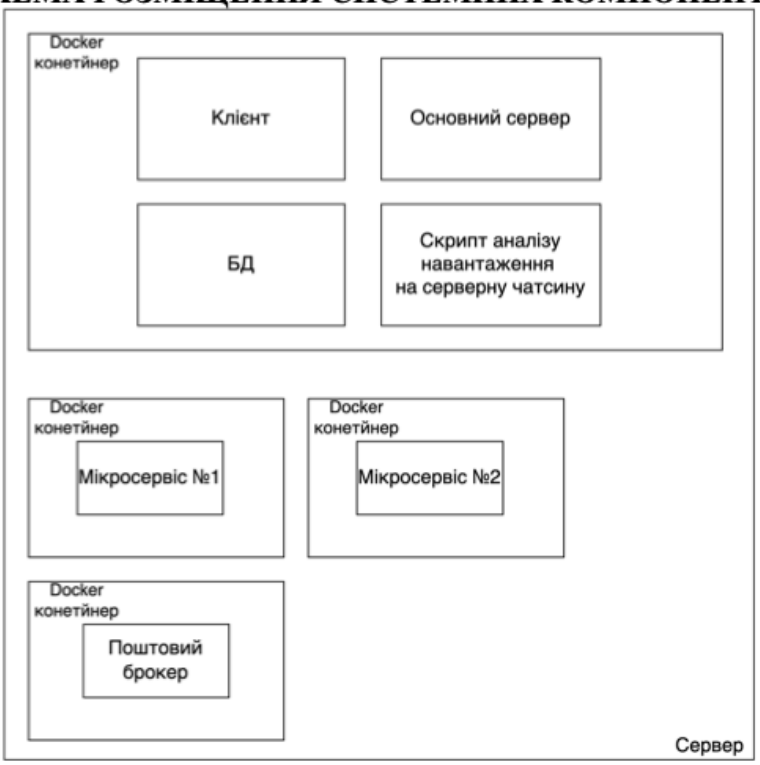
$$A_R = 1 - F_R \quad 4)$$

7

ПРОГРАМНА МОДЕЛЬ АЛГОРИТМУ ОБРОБКИ ЗАПИТІВ

8

СХЕМА РОЗМІЩЕННЯ СИСТЕМНИХ КОМПОНЕНТІВ



9

РЕЗУЛЬТАТИ МОДЕЛЮВАННЯ ОБРОБКИ ЗАПИТІВ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ RABBITMQ

Критерії ефективності	REST	RabbitMQ
Час відгуку (сек.) $T_{R,M}$	0.6	0.5
Масштабованість (кількість сервісів) $S_{R,M}$	1	2
Навантаження (кількість запитів) L	20	20
Ймовірність виходу з ладу сервісу (від 0 до 1) $F_{R,M}$	0.5	0.5
Ізоляція відмов (від 0 до 1) $Q_{R,M}$	0	1
Ефективність розподілення навантаження (час на обробку запитів в сек.) $E_{R,M}$	12	5
Доступність сервісів $A_{R,M}$	0.5	1

10

ВИСНОВКИ

1.Проаналізовано існуючі методи комунікації між мікросервісами з подальшим визначенням їх основних недоліків та переваг;

2.Розроблено програмну модель для визначення черг запитів та максимальної кількості запитів для розподілення навантаження;

3.Розроблено функціональну модель обробки запитів між сервісами.

4.Вдосконалено процес розгортання додаткових сутностей сервісів за допомогою введення сторонніх API для аналізу навантаження та використанням поштових брокерів сповіщень;

5.Розроблено схему розміщення системних компонентів;

6.Покращено час відгуку між сервісами та швидкість обробки запитів.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Приходько Д.А., Корецька В.О. Використання мікросервісної архітектури для масштабування та розподілення навантаження на серверну частину додатку. // Телекомунікаційні та інформаційні технології. Номер 1 2024.

Тези доповідей:

Тези доповідей:

1. Приходько Д.А., Корецька В.О. Використання мікросервісної архітектури для масштабування та розподілення навантаження на серверну частину додатку.// Науково-практична конференція «Проблеми комп'ютерної інженерії». – Київ: ДУТ, 2023.
2. Приходько Д.А., Корецька В.О. Використання програмних брокерів сповіщень для реалізації мікросервісної архітектури. // V Всеукраїнська науково-практична конференція «Telecommunication: problems and innovation» – Київ: ДУТ, 2023.

ДЯКУЮ ЗА УВАГУ!