Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Магістр
(назва освітнього ступеня)

на тему: Організація взаємодії мікросервісів у задачах розробки веб-застосунків

Виконав: студент 6 курсу, групи СНм-61
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

(підпис) Семчишин П.М. (прізвище та ініціали)

Керівник (підпис) доц. Фриз М.Є. (прізвище та ініціали)

Нормоконтроль (підпис) доц. Дуда О.М. (прізвище та ініціали)

Завідувач кафедри (підпис) доц. Боднарчук І.О. (прізвище та ініціали)

Рецензент (підпис) проф. Ясній О.П. (прізвище та ініціали)

м. Тернопіль – 2025
Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп’ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ доц. Боднарчук І.О.
(підпис) (прізвище та ініціали)
« 17 » 11 20_25_ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр
(назва освітнього ступеня)

за спеціальністю 122 Комп’ютерні науки
(шифр і назва спеціальності)

студенту Семчишину Павлу Миколайловичу

1. Тема роботи Організація взаємодії мікросервісів у задачах розробки веб-застосунків

Керівник роботи Фриз Михайло Євгенович, к.т.н, доц. каф. КН
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом по університету від «__27__» _листопада_ 2025 року №__4/7-1042_

2. Термін подання студентом роботи 15.12.2025

3. Вихідні дані до роботи наукові літературні джерела

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1 Аналіз предметної області.
2. Проектування архітектури, основних технічних рішень. розробка веб-застосунку.
3. Порівняння способів взаємодії мікросервісів між собою
4. Охорона праці та безпека в надзвичайних ситуаціях

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень, слайдів)
1. Тема роботи. 2. Актуальність. 3. Мета, задачі дослідження. 4. Об’єкт, предмет дослідження
наукова новизна, практичне значення роботи. 5. Схема мікросервісної архітектури
6. Способи взаємодії мікросервісів. 7. Архітектурна схема застосунку
8. Програмні засоби і способи розробки. 9. Структура бази даних. Сервіс авторизації
10. Сервіс замовлень. 11. Сервіси товарів, повідомлень, відгуків
12. Розгортання системи. 13. Вхідні дані для порівняння методів взаємодії між собою
14. Час відгуку. 15. Пропускна спроможність. 16. Навантаження на процесор.
17. Споживання пам’яті. 18. Основні результати дослідження.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Сенчишин В.С., доцент каф. МТ		
Безпека в надзвичайних			
ситуаціях	Теслюк В.М., проректор з АГРБ		

7. Дата видачі завдання 17.11.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Затвердження теми кваліфікаційної роботи	27.11.25	Виконано
2	Аналіз літературних джерел	28.11.25–30.11.25	Виконано
3	Обґрунтування актуальності дослідження	01.12.25–03.12.25	Виконано
4	Аналіз предмету дослідження та предметної області	04.12.25–06.12.25	Виконано
5	Проведення дослідження методів та засобів аналітичного опрацювання даних	07.12.25 –08.12.25	Виконано
6	Оформлення розділу «Аналіз предметної області»	09.12.25–10.12.25	Виконано
7	Оформлення розділу «Проектування архітектури, основних технічних рішень. розробка веб-застосунку»	12.12.25–13.12.25	Виконано
8	Оформлення розділу «Порівняння способів взаємодії мікросервісів між собою»	13.12.25–14.12.25	Виконано
9	Оформлення розділу «Охорона праці та безпека в надзвичайних ситуаціях»	15.12.25–16.12.25	Виконано
10	Нормоконтроль	15.12.25–17.12.25	Виконано
11	Перевірка на плагіат	15.12.25–17.12.25	Виконано
12	Попередній захист роботи	15.12.25	Виконано
13	Захист кваліфікаційної роботи	22.12.25	

Студент _____ Семчишин П.М.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Фриз М.Є.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Організація взаємодії мікросервісів у задачах розробки веб-застосунків // Кваліфікаційна робота освітнього рівня «Магістр» // Семчишин Павло Миколайлович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем та програмної інженерії, кафедра комп'ютерних наук, група СНм–61 // Тернопіль, 2025 // С. – 72, рис. – 19, табл.– 17, слайдів – 18, додат. – 5, бібліогр. – 35.

Ключові слова: kafka, rabbitmq, rest api, rpc, взаємодія мікросервісів, веб-застосунок, передача даних

Робота спрямована на вивчення та практичне застосування сучасних підходів до розробки веб-застосунків на основі мікросервісної архітектури, а також на виявлення їх переваг та недоліків у порівнянні з традиційним монолітним підходом.

У процесі виконання роботи було здійснено дослідження популярних у наш час архітектур для розробки, а також методів обміну даними між системами/сервісами. Кожен із цих підходів володіє певними особливостями і може бути застосованим залежно від визначенх вимог проекту та бізнес-цілей. Було спроектовано та розроблено веб-застосунок на базі мікросервісної архітектури. На основі створеної програми було проведено порівняння способів взаємодії мікросервісів (REST API, gRPC, RabbitMq, Kafka) за такими параметрами: час відгуку, пропускна здатність, навантаження на процесор, споживання пам'яті.

Результати проведених експериментів свідчать, що Kafka у всіх випадках продемонструвала середні результати, таким чином Kafka є найбільш універсальним програмним продуктом для втілення такого роду задач.

ANNOTATION

Organizing the interaction of microservices in web application development tasks // Semchychyn Mykola // Ternopil Ivan Pul'uj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, SNm-61 group // Ternopil, 2025 // P. - 72, Fig. - 19, Table – 17, Slide - 18, References - 35.

Keywords: web application, kafka, rabbitmq, rest api, rpc, microservices interaction, data transfer

Thesis is aimed at studying and practical application of modern approaches to developing web applications based on microservice architecture, as well as identifying their advantages and disadvantages compared to the traditional monolithic approach.

In the process of carrying out the work, a study was carried out of currently popular architectures for development, as well as methods of data exchange between systems/services. Each of these approaches has certain features and can be applied depending on the specific requirements of the project and business goals. A web application based on microservice architecture was designed and developed. Based on the created program, a comparison of microservice interaction methods (REST API, gRPC, RabbitMq, Kafka) was carried out according to the following parameters: response time, bandwidth, processor load, memory consumption.

The results of the experiments show that Kafka in all cases demonstrated average results, thus Kafka is the most versatile software product for implementing this type of tasks.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ СКОРОЧЕНЬ І ТЕРМІНІВ

API (Application Programming Interface) – програмний інтерфейс.

REST API (Representational State Transfer API) — архітектурний стиль, що використовується для побудови вебсервісів, де клієнт та сервер взаємодіють через HTTP.

RPC (Remote Procedure Call) – архітектурний стиль, призначений для побудови розподілених систем, заснованих на веб-сервісах.

SOAP (Simple Object Access Protocol) – протокол обміну повідомленнями, використовується створення розподілених систем, заснованих на веб – сервісах.

БД – база даних.

ЗМІСТ

ВСТУП

Актуальність теми. У розробці веб-застосунків сьогодні можна виділити два основні підходи. Перший — це монолітна архітектура, де весь функціонал програми зосереджений в одному великому застосунку. Другий ключовий підхід полягає у використанні мікросервісної архітектури. Цей підхід є декомпозицією програми на невеликі, незалежні компоненти, звані мікросервісами, кожен з яких займається вирішенням певної задачі або функціоналу. Такий підхід забезпечує гнучкість, прекрасну масштабованість та легкість підтримання системи.

Мікросервіси стали невід'ємною частиною інструментарію сучасних розробників веб-застосунків [1]. Вони дозволяють створювати складні та високонавантажені системи, враховуючи сучасні вимоги до продуктивності, надійності та масштабованості. Однак за всіма перевагами мікросервісів стоять певні виклики.

Тема роботи є актуальною, оскільки в наш час веб- застосунки є надзвичайно популярні, і все частіше для їх розробки стала використовуватися саме мікросервісна архітектура.

Мета дослідження: проектування та розробка веб-застосунку, заснованого на принципах мікросервісної архітектури, а також порівняння різних методів взаємодії між сервісами.

Будуть розглянуті різні аспекти цієї проблеми, включаючи переваги та недоліки використання мікросервісів, основні методи та технології взаємодії між ними, а також буде проаналізовано різні підходи до організації взаємодії та їх застосування в конкретних сценаріях розробки.

Для досягнення мети, в роботі поставлено та розв’язано **такі задачі:**

– вивчено основні принципи мікросервісної архітектури з метою розуміння її переваг та особливостей;

– проведено аналіз та виявлено переваги мікросервісної архітектури порівняно з монолітною архітектурою в контексті розробки веб-застосунків;

– спроектовано структуру та функціонал веб-застосунку на базі мікросервісної архітектури, включаючи визначення компонентів та їх взаємозв'язків;

– розроблено окремі мікросервіси, включаючи їх функціонал, інтерфейси та способи взаємодії;

– створено серверну (бекенд) частину веб-застосунку, включаючи реалізацію серверної логіки та бази даних для мікросервісів;

– проведено аналіз та порівняння різних методів та технологій взаємодії між мікросервісами в різних сценаріях використання.

Об’єкт дослідження: мікросервісна архітектура.

Предмет дослідження: взаємодія мікросервісів у веб-застосунку.

Наукова новизна роботи: для втілення порівняння способів взаємодії мікросервісів запропоновано використати такі параметри, як час відгуку, пропускна здатність, навантаження на процесор, споживання пам'яті.

Практичне значення одержаних результатів. Результати роботи допоможуть розробникам краще розуміти проблеми та можливості, пов'язані з організацією взаємодії мікросервісів у веб-розробці, та приймати обґрунтовані рішення при проектуванні та розгортанні сучасних веб-застосунків.

Апробація. Окремі результати роботи були представлені на наукових конференціях як опубліковані тези:

1. Семчишин П.М. Архітектурні рішення для розробки веб-застосунків // XIV Міжнародна науково-практична конференція молодих учених та студентів «Актуальні задачі сучасних технологій», Тернопіль, 11-12. Грудня 2025 р. с. 340-341.

2. Семчишин П.М. Проектування мікросервісної архітектури веб-застосунку // Інформаційні моделі, системи та технології: Праці XIII наук.-техн. конф. Тернопіль, 17 – 18 грудня 2025 р. с. 84.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Архітектурні рішення

Коли йдеться про веб-застосунки, при виборі архітектурного рішення є два основні варіанти:

– монолітна архітектура;

– мікросервісна архітектура.

1.1.1 Монолітна архітектура

Монолітна архітектура в веб-застосунках є підходом, при якому весь функціонал програми виконується в одному цілому, зазвичай у вигляді єдиного виконуваного файлу або програми [2]. Основні компоненти програми, такі як інтерфейс користувача, бізнес-логіка та доступ до даних, знаходяться всередині однієї програми та взаємодіють безпосередньо (рис. 1.1).

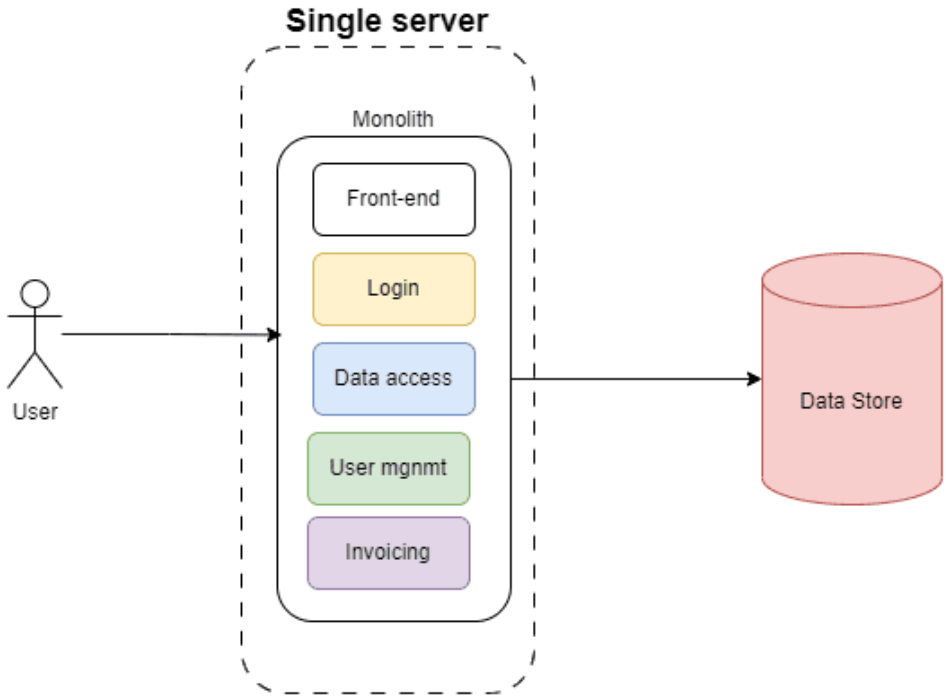

Рисунок 1.1 – Схема монолітної архітектури

Перевагами такого виду архітектури є [3]:

– простота розгортання - оскільки всі компоненти знаходяться в одному місці, розгортання програми зазвичай відбувається швидше та простіше;

– простота розробки - відсутність складної інфраструктури полегшує процес розробки та налагодження програми;

– зручність масштабування на початку - при невеликому обсязі трафіку користувача монолітний застосунок може забезпечити достатню продуктивність без необхідності поділу на окремі сервіси.

Недоліки монолітної архітектури [3]:

– складність підтримки - при збільшенні розміру програми та додаванні нових функцій може виникнути складність у підтримці та зміні коду через його єдиний монолітний характер;

– обмежена масштабованість - при досягненні межі продуктивності або необхідності масштабування окремих компонентів програми можуть виникнути труднощі через їх взаємозв'язок усередині моноліту.

– обмежена гнучкість - через єдиний характер застосування зміни в одній його частині можуть торкнутися інші компоненти, що ускладнює зміни та впровадження нових технологій.

У такий спосіб, монолітна архітектура може бути хорошим вибором для простих застосунків з невисоким навантаженням та невеликими вимогами до масштабованості та гнучкості [2].

1.1.2 Мікросервісна архітектура

Мікросервісна архітектура у веб-застосунках є підходом, у якому застосунок розбивається на невеликі автономні сервіси, кожен із яких відповідає окремому функціоналу (рис. 1.2). Кожен сервіс, зазвичай, розгортається та масштабується незалежно [3].

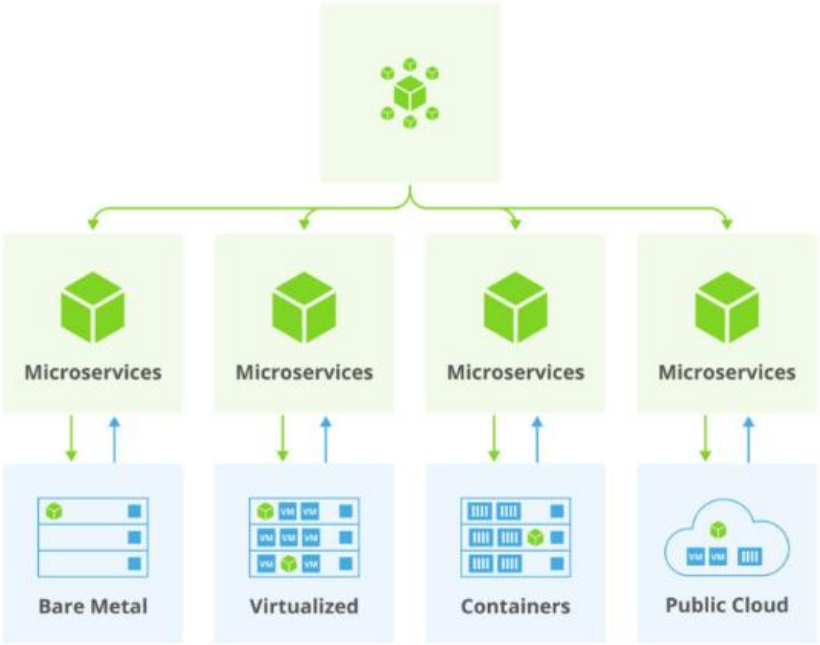

Рисунок 1.2 – Схема мікросервісної архітектури

Переваги такої архітектури [3]:

– гнучкість та масштабованість - завдяки поділу програми на незалежні сервіси, кожен з них можна масштабувати та оновлювати незалежно від інших компонентів, що покращує гнучкість та масштабованість всієї програми;

– легкість підтримки та розробки - кожен сервіс зазвичай меншого розміру та має чітко визначені межі відповідальності, що полегшує підтримку та розробку програми;

– технологічна різноманітність - різні сервіси можуть бути написані різними мовами програмування або використовувати різні технології, що дозволяє вибирати найбільш потрібні інструменти для кожного конкретного завдання.

Проте є і недоліки [3]:

– складність конфігурації та управління - управління великою кількістю незалежних сервісів вимагає наявності складної інфраструктури та механізмів управління, що може спричинити додаткові складності та витрати;

– мережева затримка - оскільки кожен сервіс взаємодіє з іншими через мережу, виникає затримка, яка може вплинути на загальну продуктивність програми;

– складність тестування - тестування мікросервісів може бути складним через необхідність керування їх взаємодією та залежностями [4].

Таким чином, мікросервісна архітектура часто застосовується у великих та складних застосунках, де потрібна висока гнучкість, масштабованість та можливість використання різних технологій.

Якщо підбити підсумок щодо архітектурних рішень, то можна виділити те, що головний плюс мікросервісної архітектури - це велика гнучкість і масштабованість. Розбиття програми на окремі сервіси дозволяє розробляти і впроваджувати новий функціонал швидше і ефективніше, оскільки зміни в одному сервісі не торкаються інших. Це особливо важливо для сучасних веб-застосунків, які передбачають високу динамічність та швидке впровадження нових можливостей [3]. Використання мікросервісної архітектури також може допомогти прискорити розробку, оскільки робота над кожним сервісом ведеться незалежно і може виконуватися різними людьми/командами в різні терміни. Кожен мікросервіс може бути масштабований незалежно від його навантаження, що забезпечує значно ефективніше застосування обчислювальних ресурсів і збільшує надійність усієї системи. Але також дана архітектура має багато нюансів, які можуть викликати проблеми. Наприклад, складність побудови інфраструктури та складність управління всім, можуть (але не обов'язково) виникнути проблеми з безпекою, так як доводиться часто передавати дані між сервісами, відповідно зростає потенційний ризик витоків інформації, ну і той факт, що дані розділені за різними процесами всієї інформації складніше [5].

Що ж до монолітної архітектури, то всіх цих мінусів можна уникнути, але монолітна архітектура зручна рівно до певних розмірів застосунка, далі його стає досить проблематично підтримувати, є навіть такий термін “Big Ball of Mud” (термін, що використовується в контексті монолітної архітектури, що позначає ситуацію, коли застосунко стає надто складним і важким для підтримки та розвитку їх складно розділити на більш дрібні та незалежні частини). Що стосується масштабованості, то там і зовсім є межа.

1.2 Способи взаємодії мікросервісів

Є ключовим аспектом при розробці веб-застосунків на основі мікросервісної архітектури. Якщо за монолітної архітектурі інформація може легко передаватися між різними класами всередині застосунку, то у мікросервісах її треба відправляти до інших систем. Для забезпечення ефективної роботи системи необхідно вибрати відповідні методи взаємодії, які відповідатимуть вимогам проекту та забезпечуватимуть високу продуктивність, надійність та масштабованість [6]. У цьому підрозділі буде розглянуто основні способи взаємодії мікросервісів.

Усього можна виділити 2 основні напрямки взаємодії мікросервісів: синхронний та асинхронний.

Синхронний - це процес обміну інформацією, при якому відправник очікує відповіді від одержувача перед тим, як продовжити виконання своєї роботи. У цьому випадку, коли мікросервіс **A** надсилає запит мікросервісу **B** , він блокує свою роботу і чекає, поки мікросервіс B обробить запит і поверне відповідь. Таким чином, синхронна взаємодія має на меті пряму залежність між відправником та одержувачем, де відправник блокується до отримання відповіді від одержувача. Цей тип взаємодії часто використовується у випадках, коли відповідь виходить негайно та без значних затримок.

Асинхронний - це спосіб обміну інформацією, при якому відправник не блокує свою роботу в очікуванні відповіді від одержувача. Натомість відправник відправляє запит одержувачу і продовжує свою роботу без очікування відповіді. Отримувач обробляє запит асинхронно та відправляє відповідь, коли вона буде готовою. У разі асинхронної взаємодії відправник може продовжувати виконувати інші завдання, не зупиняючись на очікуванні відповіді від одержувача. Асинхронна взаємодія широко використовується в розподілених системах, де відповідь може зайняти значний час, наприклад, коли потрібна обробка великих обсягів даних або коли одержувач не доступний в даний момент. Цей підхід дозволяє збільшити пропускну спроможність системи

і підвищити стійкість до відмови, оскільки відправник не блокується в очікуванні відповіді і може продовжувати свою роботу незалежно від стану одержувача [7].

Розглянемо кілька варіантів синхронної взаємодії мікросервісів [8].

1.2.1 API

Є описом способів взаємодії однієї комп'ютерної програми з іншими за допомогою різних протоколів. API є представником синхронного спрямування.

Синхронна взаємодія мікросервісів з використанням API відбувається шляхом надсилання HTTP -запитів (або будь-яких інших) від одного сервісу до іншого. Наприклад, мікросервіс **A** може надіслати HTTP- запит на певну URL - адресу мікросервісу **B** , передаючи при цьому необхідні дані. Мікросервіс **B** обробляє запит, виконує необхідні операції та повертає HTTP -відповідь з результатом обробки назад мікросервісу **A** . При цьому мікросервіс **A** блокується і чекає на відповідь від мікросервісу **B** .

API є одним із найбільш поширених підходів до взаємодії між мікросервісами в сучасних веб-застосунках завдяки своїй простоті, гнучкості та підтримці стандартних протоколів HTTP.

1.2.2 RPC

В рамках RPC –архітектури [9], кожен мікросервіс є набором віддалених процедур, до яких можна звертатися для виконання певних операцій (рис. 1.3). RPC як і REST є синхронним способом.

Синхронна взаємодія мікросервісів із використанням RPC відбувається шляхом виклику віддалених процедур. Наприклад, мікросервіс **A** може викликати віддалену процедуру на мікросервісі **B** , передаючи необхідні параметри. Мікросервіс **B** обробляє запит і повертає результат обробки назад

мікросервісу **A.** При цьому мікросервіс **A** також блокується і очікує на завершення виклику віддаленої процедури.

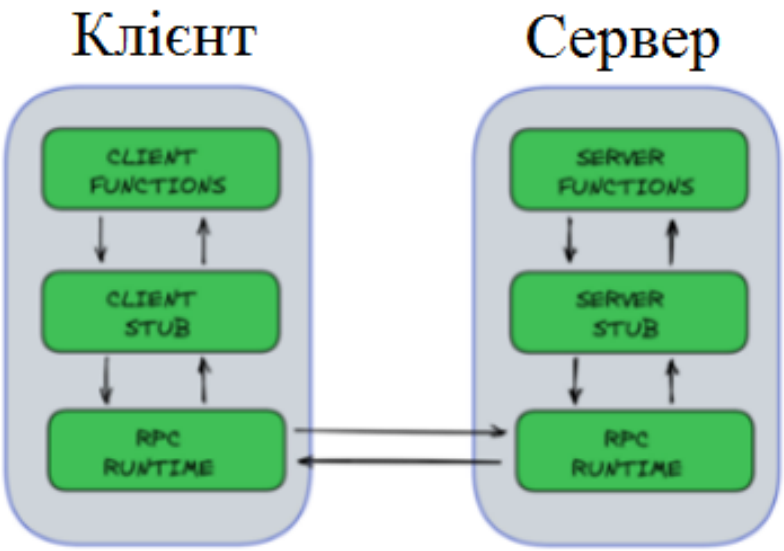

Рисунок 1.3 – Взаємодія за допомогою RPC

Основні компоненти RPC включають клієнтський інтерфейс, який забезпечує виклик віддалених процедур, і серверний інтерфейс, який обробляє ці виклики і виконує відповідні операції. Популярними протоколами для реалізації RPC є gRPC, Apache Thrift та JSON-RPC.

RPC також є найпоширенішим підходом до взаємодії між мікросервісами у веб-застосунках. Він забезпечує високу продуктивність та ефективне використання мережевих ресурсів завдяки компактним форматам даних та оптимізованим протоколам передачі даних.

1.2.3 SOAP

В рамках SOAP –архітектури [10], кожен мікросервіс є веб-сервісом, до якого можна звертатися для виконання певних операцій.

Синхронна взаємодія мікросервісів з використанням SOAP (рис. 1.4) відбувається шляхом надсилання SOAP -повідомлень від одного сервісу до іншого. Наприклад, мікросервіс **A** може відправити SOAP -запит на певний

ендпоінт мікросервісу **B** , що містить інформацію про метод, що викликається і параметри, що передаються. Мікросервіс **B** обробляє запит, виконує необхідні операції і повертає SOAP -відповідь з результатом обробки мікросервісу **A** .

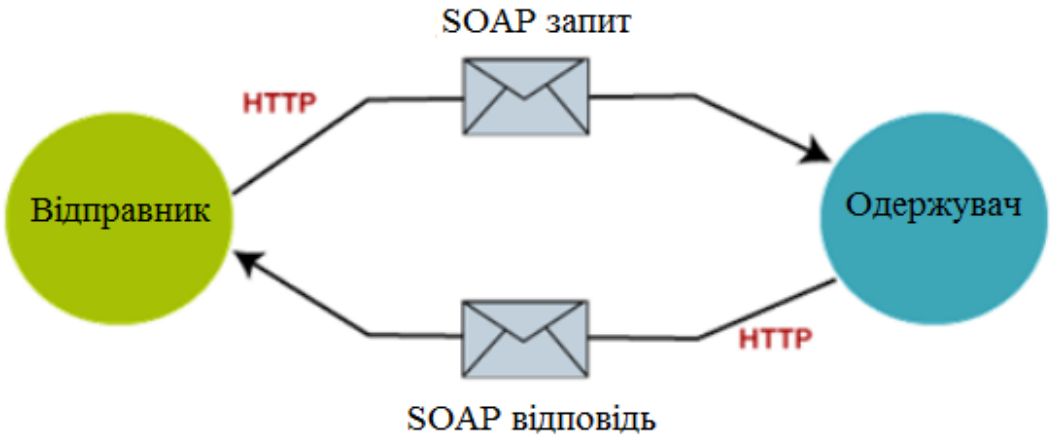

Рисунок 1.4 – Взаємодія за допомогою SOAP

Основні компоненти SOAP включають XML- схеми для визначення структури повідомлень, WSDL для опису доступних операцій і портів, а також SOAP -заголовки для передачі метаданих і додаткової інформації.

SOAP забезпечує високу надійність та безпеку взаємодії між мікросервісами завдяки використанню промислових стандартів та протоколів, таких як WS-Security для підтримки конфіденційності і цілісності інформації.

Хоча SOAP є менш поширеним у порівнянні з REST та RPC, він, як і раніше, використовується в різних сценаріях, де потрібна висока надійність та безпека при обміні даними між мікросервісами.

1.2.4 Обмін повідомленнями

Обмін повідомленнями або меседжинг - це архітектурний підхід [8], що використовується для формування розподілених систем, котрий базується на передачі повідомлень між мікросервісами. У межах меседжинга, кожен мікросервіс може публікувати повідомлення певні теми (у такому випадку сервіс називається Producer) чи черги, інші мікросервіси можуть підписуватися

ці теми чи черги щоб одержати повідомлень (у цьому випадку його називають Consumer) (рис. 1.5). Таким чином меседжинг – це основний представник асинхронного підходу до спілкування між мікросервісами (і не тільки).

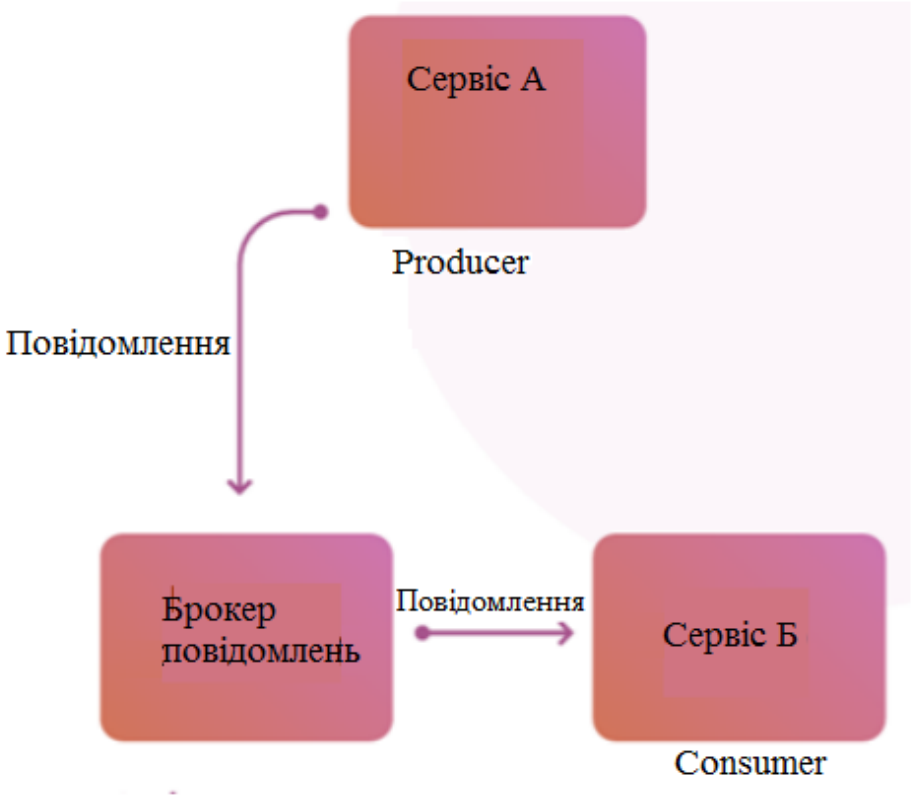

Рисунок 1.5 – Спілкування за допомогою обміну повідомленнями

Взаємодія мікросервісів з використанням меседжингу відбувається асинхронно. Наприклад, мікросервіс А може опублікувати повідомлення в чергу, а мікросервіс, який підписаний на цю чергу, отримає повідомлення і обробить його відповідно до логіки свого застосунку.

Основні компоненти месседжинга включають брокер повідомлень (наприклад, ZeroMq, ActiveMq, RabbitMQ), який відповідає за зберігання і маршрутизацію повідомлень, а також клієнтські бібліотеки або адаптери, які використовуються мікросервісами для відправлення та отримання повідомлень.

Меседжинг забезпечує гнучкість та масштабованість взаємодії між мікросервісами, дозволяючи їм обмінюватися даними незалежно від свого поточного стану чи доступності інших сервісів. Крім того, меседжинг дозволяє

реалізувати асинхронні процеси та забезпечує відмовостійкість шляхом збереження повідомлень навіть у разі тимчасової недоступності одержувача.

Хоча меседжинг є більш складним підходом до взаємодії між мікросервісами, він широко використовується у великих та розподілених системах, де потрібна значна гнучкість, висока масштабованість та надійність обмінювання даними.

1.2.5 Стрімінг

Це архітектурний підхід, який використовується для передачі даних у реальному часі між мікросервісами в розподілених системах. В рамках стрімінгу дані передаються як безперервний потік подій або повідомлень від одного мікросервісу до іншого.

Взаємодія мікросервісів з використанням стрімінгу відбувається асинхронно та безперервно. Наприклад, мікросервіс **A** може створити стриму даних і почати передавати його мікросервіс **B** . Мікросервіс **B** може передплатити цей стрим і отримувати дані в режимі прямої трансляції, опрацьовуючи їх у міру їх надходження.

Основні компоненти стрімінгу включають стрімінгові платформи (наприклад, Apache Kafka, Apache Flink, Apache Pulsar), які відповідають за керування потоками даних, а також клієнтські бібліотеки або адаптери, які використовуються мікросервісами для створення та підписки на стрими даних.

Стрімінг забезпечує можливість обробки даних у реальному часі, що є особливо важливим для застосунків, що вимагають миттєвого реагування на події або операції в реальному часі. На додачу, стрімінг дозволяє обробляти значні обсяги даних при мінімальній затримці та забезпечує відмовостійкість шляхом реплікації даних та обробки помилок у реалі.

Хоча стрімінг є більш складним підходом до взаємодії між мікросервісами, він широко використовується в застосунках, що потребують обробки даних в даний момент, таких як системи моніторингу, аналітики,

обробки потоків подій тощо.

Стрімінг за своєю суттю дуже нагадує меседжинг, але їх основна відмінність – перший орієнтований на обробку даних у режимі стрімінгу, надаючи можливість безперервної передачі та обробки даних у реальному часі. На противагу месенджингу, який частіше використовується для передачі повідомлень між компонентами системи, стрімінг дозволяє опрацьовувати значні за обсягом потоки даних, ефективно працюючи з високими навантаженнями та забезпечуючи надійну та масштабовану архітектуру для обробки потоків даних.

1.3 Висновки до першого розділу

Варто наголосити, що мікросервісна архітектура стала популярним варіантом для створення сучасних застосунків завдяки низці переваг і великій варіативності, завдяки чому фахівці все частіше вибирають її при розробці своїх застосунків.

У першому розділі був проведений огляд деяких підходів взаємодії, але окрім проаналізованих, є ще багато інших, ми розглянули тільки основні і найпопулярніші. Кожен із цих підходів має свої особливості та може бути обраний залежно від конкретних вимог проекту та бізнес-цілей. Важливо вибирати підхід, який найкраще відповідає цілям проекту та забезпечує ефективну взаємодію між компонентами системи.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ, ОСНОВНИХ ТЕХНІЧНИХ РІШЕНЬ. РОЗРОБКА ВЕБ-ЗАСТОСУНКУ

2.1 Архітектура системи, що розробляється

Як розроблювана система було обрано веб-застосунок для інтернет-магазину одягу. Для його створення використовуватиметься клієнт - серверна модель, де основна частина функціоналу знаходиться на сервері. Це забезпечує високий рівень безпеки разом із захистом інформації, оскільки на сервері відбувається обробка основних операцій із даними [11]. Також такий підхід сприяє підвищенню сумісності різних програмних продуктів та платформ, що спрощує їх інтеграцію та знижує вимоги до них.

Основні функціональні можливості програми включають авторизацію користувачів, можливість оформлення замовлень, отримання повідомлень і залишення відгуків. Ці функції забезпечують зручність та функціональність для користувачів, роблячи процес покупок та взаємодії з магазином більш ефективним та приємним.

Архітектура сервера буде мікросервісною, а також потрібна буде БД. Було вирішено використовувати патерн “Database per service” [12], тобто коли в кожного сервісу своя окрема БД. Цей спосіб зручний тим, що застосунок буде мати слабку пов'язаність сервісів, що полегшує роботу з їх модифікацією, а також при потребі, кожен сервіс може використовувати різні БД (наприклад, для деяких варіантів можливо NoSql бази буде зручніше, ніж звичайний Sql), проте є і недолік даного способу - складність управління даними. Дуже складно добути інформацію, шматки якої розкидані по різних сервісах, також складніше організовувати будь-які бізнес-транзакції, які охоплюють кілька сервісів.

Для доступу користувача до різних послуг буде використовуватися API Gateway. Це серверний проксі, який надає інтерфейс для клієнтів (застосунків, пристроїв, користувачів) для доступу до веб-сервісів або до мікросервісів. Він виконує функцію шлюзу, контролюючи та маршрутизуючи запити від клієнтів

до відповідних служб та забезпечує централізоване управління та безпеку для всіх запитів, що проходять через веб-застосунок [11].

Схема архітектури застосунку представлена на рис. 2.1.

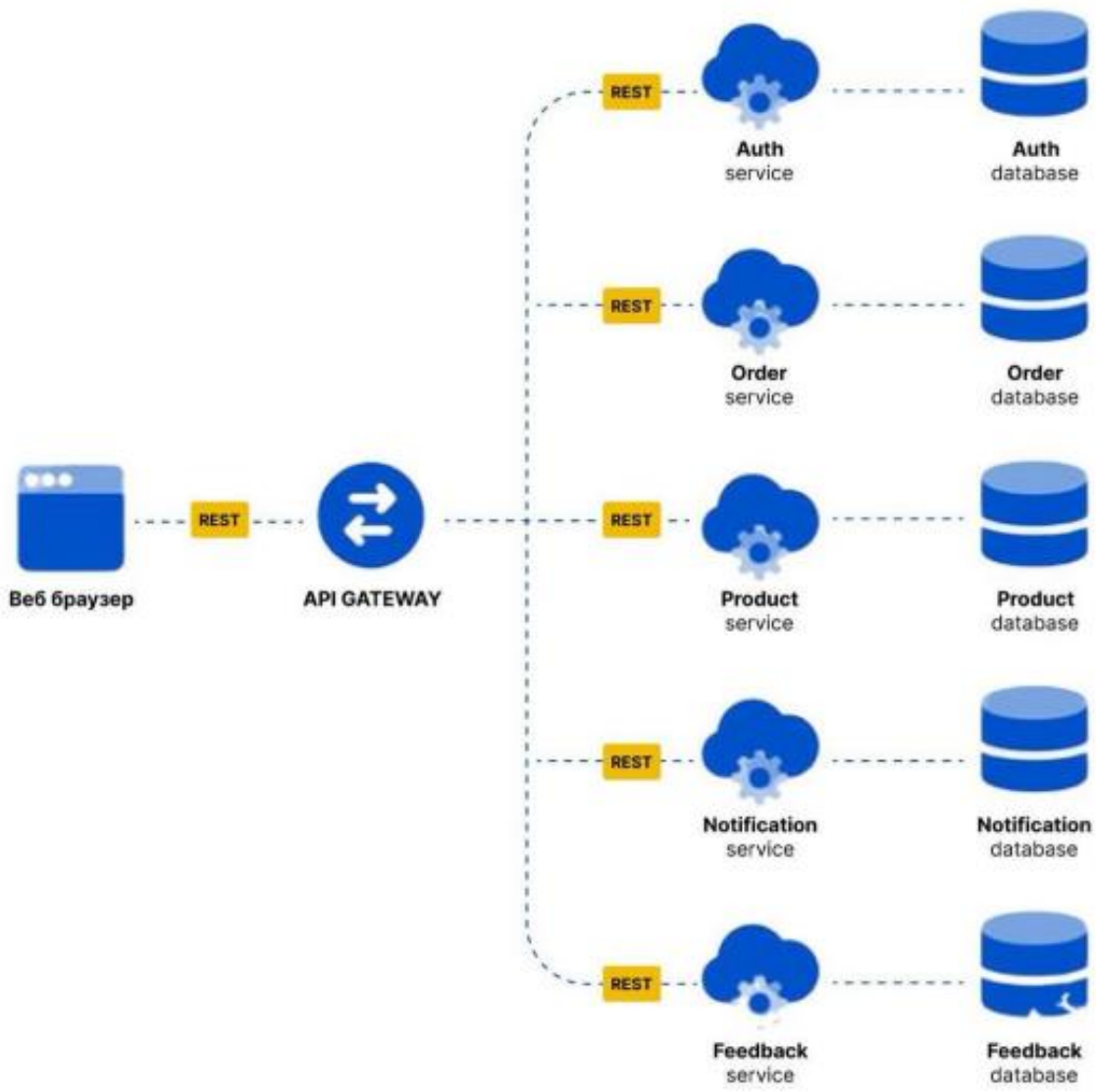

Рисунок 2.1 – Архітектурна схема програми

З рис. 2.1 видно, що у застосунка буде п'ять основних сервісів:

– авторизації;

– продуктів;

– замовлень;

– повідомлень;

– відгуків.

2.2 Вибір засобів розробки

Для розробки веб-програми інтернет-магазину одягу було прийнято рішення використовувати мову програмування Java [13] з фреймворком Spring [14]. Цей вибір був обґрунтований низкою переваг, які перераховані нижче.

1. Широка популярність та екосистема: Java є однією з найпоширеніших мов програмування, що забезпечує велику кількість інструментів, бібліотек та ресурсів для розробників. Фреймворк Spring, зі свого брку, надає потужні інструменти для побудови масштабованих та надійних веб-застосунків.

2. Простота розробки: Java та Spring надають високий рівень абстракції та інструменти для вирішення типових завдань розробки, таких як управління залежностями, обробка HTTP -запитів та взаємодія з БД, що значно спрощує розроблення та підтримання програми.

3. Надійність та продуктивність: Java має високий ступінь стабільності та продуктивності, що робить її привабливим варіантом для створення критично важливих застосунків, таких як інтернет-магазини. Spring, так само, надає безліч інструментів для оптимізації продуктивності та забезпечення надійності програми.

4. Широка підтримка спільноти: Java та Spring активно підтримуються потужною спільнотою фахівців і мають велику документацію, туторіали та форуми підтримки, що забезпечує доступ до великих ресурсів для вирішення проблем і питань, що виникають.

Як система управління БД була обрана PostgreSQL [15]. Цей вибір був зумовлений наступними причинами.

1. Надійність та стабільність: PostgreSQL є однією з найнадійніших та стабільних систем управління БД, забезпечуючи високий рівень цілісності даних та відмовостійкості.

Потужний функціонал: PostgreSQL пропонує багатий вибір функцій і можливостей, як то підтримку складних запитів, транзакцій, індексацію та розширюваність, що робить його привабливим варіантом для створення

інтернет-магазину з різноманітними вимогами до даних.

2. Відкритий вихідний код: PostgreSQL поширюється під ліцензією відкритого вихідного коду, що надає вільний доступ до вихідного коду, активну спільноту розробників та можливість участі у розвитку та покращенні продукту.

Таким чином, вибір Java з використанням Spring для розробки веб-застосунку інтернет-магазину одягу, а також PostgreSQL як система управління БД був зроблений з урахуванням вимог до надійності, продуктивності, а також зручності розробки.

Для взаємодія мікросервісів було обрано такі способи:

– REST API та gPRC як представники синхронної взаємодії [16];

– RabbitMQ для обміну повідомленнями [17];

– Kafka для стримінгу [18].

2.2.1 REST API

Є архітектурним підходом, котрий встановлює обмеження для API: як вони мають бути влаштовані та які функції підтримувати. Це дозволяє стандартизувати роботу програмних інтерфейсів, зробити їх зручнішими та продуктивнішими. За фактом REST це не окремий протокол взаємодії, а просто звід рекомендацій, які зі свого боку застосовуються для розробки API. Перший спосіб спілкування в даному випадку - це HTTP протокол, до якого застосовується низка загальних правил для стандартизації, зручності та лаконічності коду (рис. 2.2).

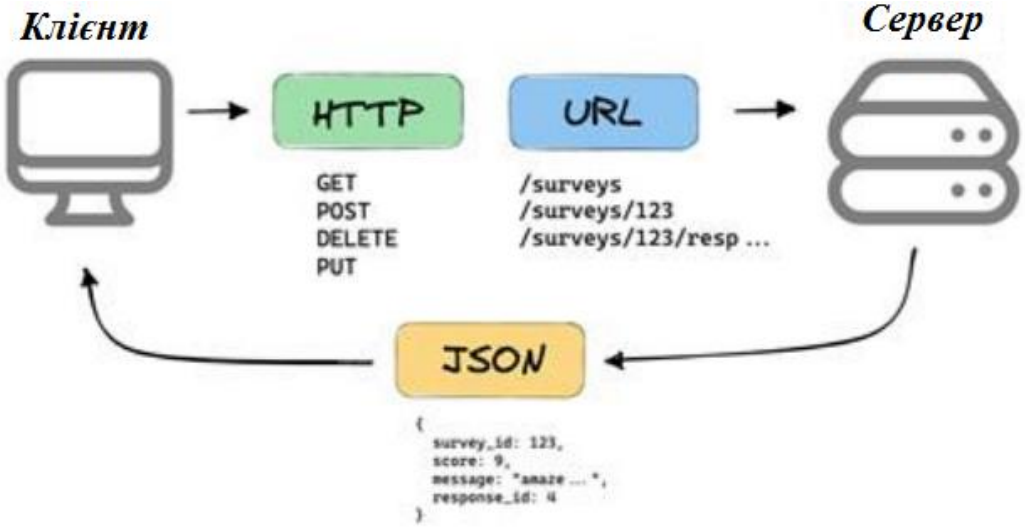

Рисунок 2.2 – REST API взаємодія

Сам обмін відбуватиметься в такий спосіб. Кожен мікросервіс є ресурсом, до якого можна звертатися за певною URL- адресою (endpoint) і виконувати операції CRUD (Create, Read, Update, Delete) з використанням стандартних HTTP- методів (GET, POST, PUT, DELETE). Тобто з клієнта надсилатиметься запит на будь-який сервіс, за допомогою HTTP запиту, який буде містити в собі якісь дані, далі ці дані обробляються якимось чином і формується відповідь, котра і відправляється у вигляді JSON назад клієнту.

2.2.2 gRPC

Цей фреймворк створений компанією Google для RPC, функціонує над протоколом HTTP/2. gRPC є простим у застосуванні, прекрасно підходить для розробки розподілених систем (мікросервісів) та API. Володіє вбудованою підтримкою для балансування навантаження, а також трасування, аутентифікації і перевіряння життєздатності сервісів. Хорошої продуктивність вдається досягнути завдяки застосування HTTP/2 та Protocol Buffers [19].

Схема роботи gPRC наведена на рис. 2.3.

Спеціальний формат серіалізації (Protobuf) застосовується за замовчуванням для передавання даних від клієнта до сервера. Застосовуючи строгу типізацію полів та бінарне представлення передачі структурованих

даних, фреймворк споживає значно менше ресурсів. Тривалість реалізаціх процесу серіалізації/десеріалізації є суттєво меншою, це ж мона сказати і за розмір повідомлень на противагу JSON/XML.

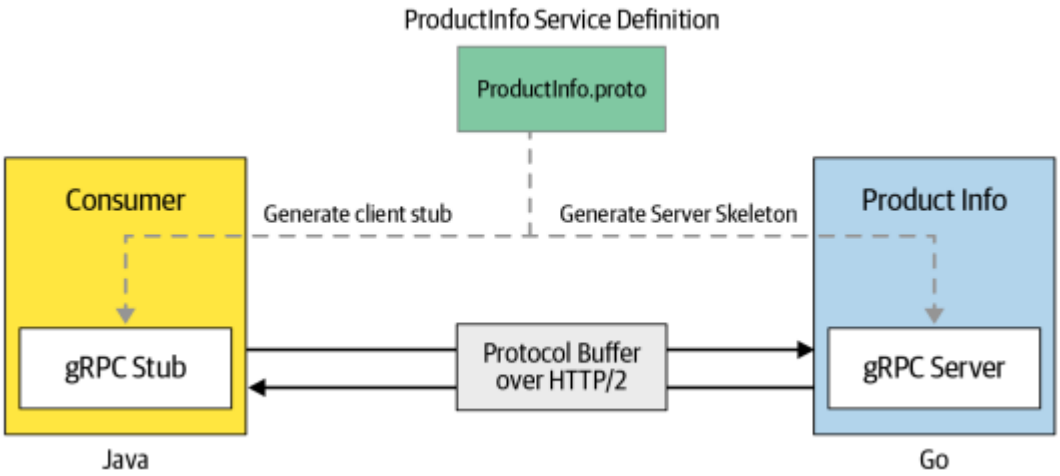

Рисунок 2.3 – Схема роботи gRPC

Якщо підсумувати, то gRPC також є способом розробки API, але є відмінності від REST. У gRPC один компонент (клієнт) викликає певні функції в іншому програмному компоненті (сервері), інакше це називається RPC. У REST замість запиту функцій клієнт запитує або оновлює дані на сервері. gRPC також використовує протокол Protobuf над даними, що дозволяє скоротити використовувані ресурси для їх передачі.

2.2.3 RabbitMq

Це програмний продукт для реалізації брокера повідомлень на основі протоколу AMQP. Простими словами, програма, яка приймає завдання від іншої програми чи іншої частини тієї ж програми. Використовується для виконання асинхронних запитів, не змушуючи користувача чекати, доки програма обробить запит, який можна обробити у фоні.

Принцип роботи: сервісу необхідно передати повідомлення (дані, запит) іншому сервісу. Але потрібно зробити це асинхронно (не чекати поки інший сервіс відповість, а продовжувати роботу), у такому разі використовується черга

для повідомлень і RabbitMq поміщає це повідомлення в цю чергу. Далі, коли другий сервіс звільниться і зможе прийняти це повідомлення, він звертається в цю чергу і бере перше повідомлення за принципом FIFO (first in, first out), тобто перше відправлене повідомлення буде прийматися також першим (рис. 2.4).

Також існує так звана DLQ (Dead-letter queue) або черга недоставлених листів, це окрема черга, призначена для повідомлень, які з якихось причин не можуть бути оброблені або доставлені (тобто коли сервіс, який має прийняти повідомлення, відмовляється від нього з якихось причин).

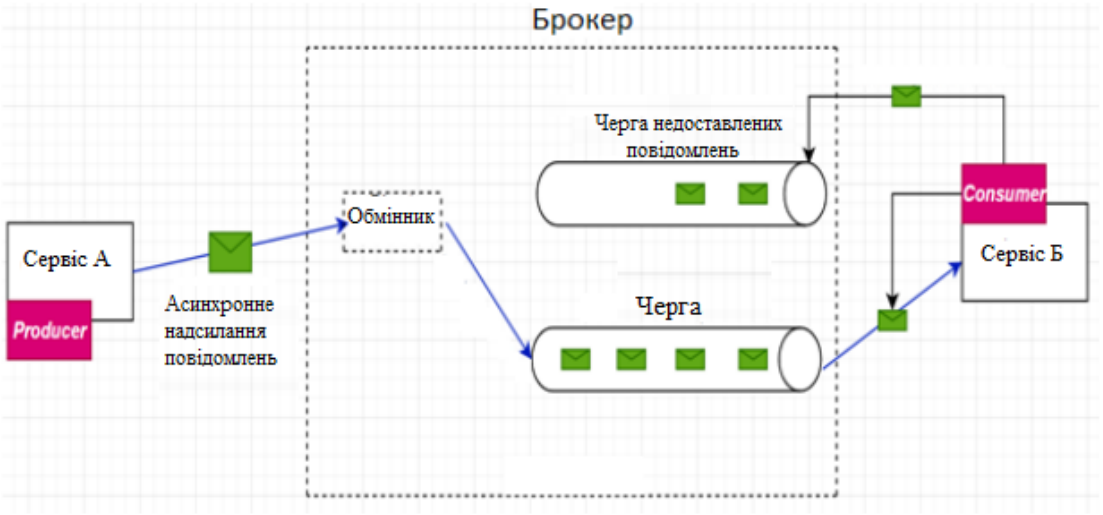

Рисунок 2.4 – Взаємодія за допомогою RabbitMq

2.2.4 Kafka

Apache Kafka - це програмний продукт, призначений для стримінгу даних у реальному часі на основі публікації-підписки. Простими словами, це система, яка дозволяє передавати дані від одного компонента до іншого, забезпечуючи надійну та ефективну передачу потоків даних.

У контексті стрімінгу даних, Kafka є посередником між виробниками даних (наприклад, застосунками, пристроями, давачами) і споживачами даних (наприклад, аналітичними застосунками, сховищами даних, системами сповіщень). Виробники поміщають дані в теми (topics) Kafka, а споживачі читають ці дані з тем для їх подальшої обробки (рис. 2.5).

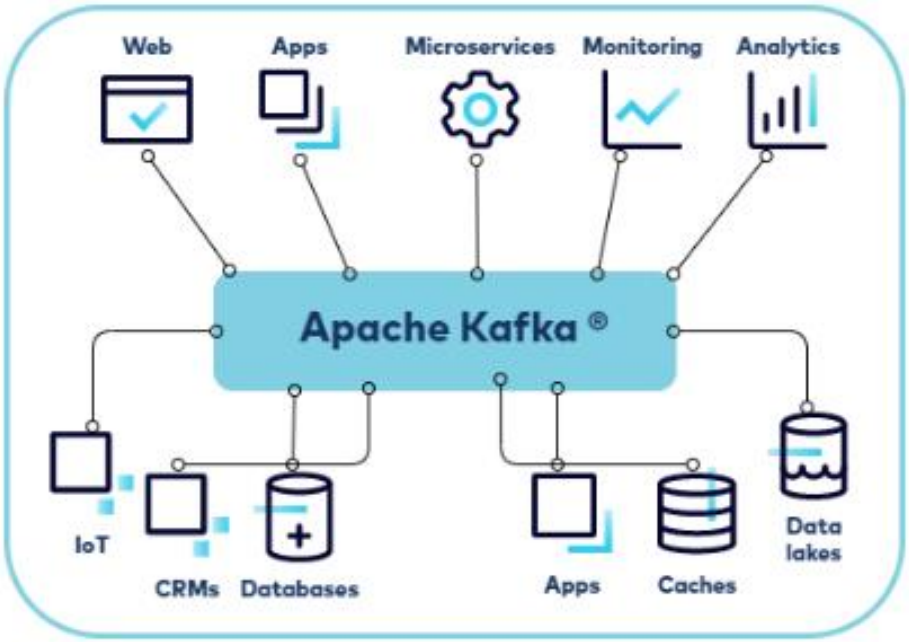

Рисунок 2.5 – Apache Kafka

Принцип роботи Kafka такий: дані публікуються у теми Kafka і зберігаються у вигляді безперервного потоку. Споживачі можуть читати дані з теми в реальному часі, а Kafka гарантує доставку даних у тому порядку, в якому вони були опубліковані (принцип FIFO). Це забезпечує надійну та послідовну обробку даних.

Крім того, у Kafka також існує поняття Dead-letter queue (DLQ) - це механізм обробки повідомлень, які не можуть бути успішно опрацьовані або доставлені. Такі повідомлення можуть бути надіслані в окрему чергу для подальшого аналізу або обробки.

Таким чином, ми бачимо подібність Kafka та RabbitMq. Їхня відмінність полягає в тому, що перший безперервно відправляє дані і другий сервіс може їх читати в реальному часі, а другий відправляє окремі повідомлення.

2.3 Розробка структури бази даних

Як вже було згадано у п.2.1, буде п'ять різних сервісів і для кожного нам потрібна буде БД. Розглянемо схеми БД для кожному за них.

2.3.1 Сервіс авторизації

Сервіс авторизації відповідальний за роботу з користувачами, відповідно ми маємо таблицю користувачів (рис. 2.6), з усіма потрібними даними, також є таблиця ролей, яка допоможе відрізнити працівників від користувачів.

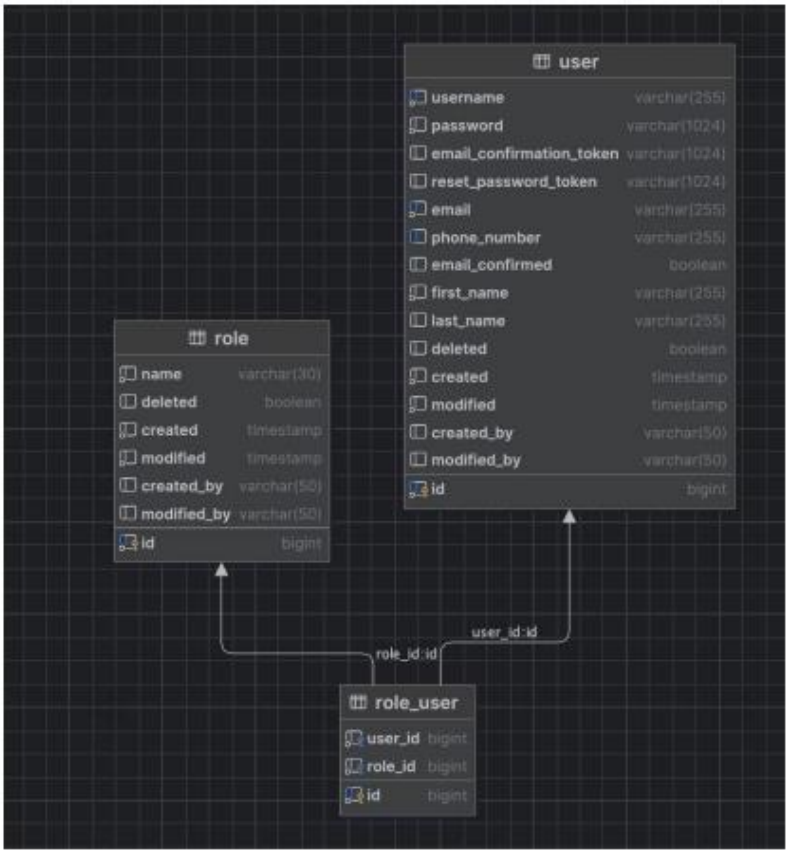

Рисунок 2.6 – Діаграма БД сервісу авторизації

2.3.2 Сервіс товарів

Сервіс продуктів відповідатиме за зберігання інформації про товар (рис. 2.7), наприклад, його тип, ціна, опис та назву.

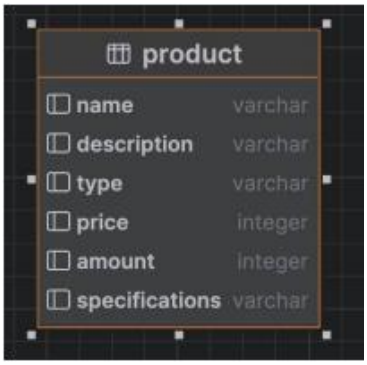

Рисунок 2.7 – Діаграма БД сервісу продуктів

2.3.3 Сервіс замовлень

Сервіс замовлень є відповідальним за створення та відстеження замовлень, тому нам потрібні таблиці (рис. 2.8) для всього замовлення, для одного предмета із замовлення, а також таблиці для відстеження статусів.

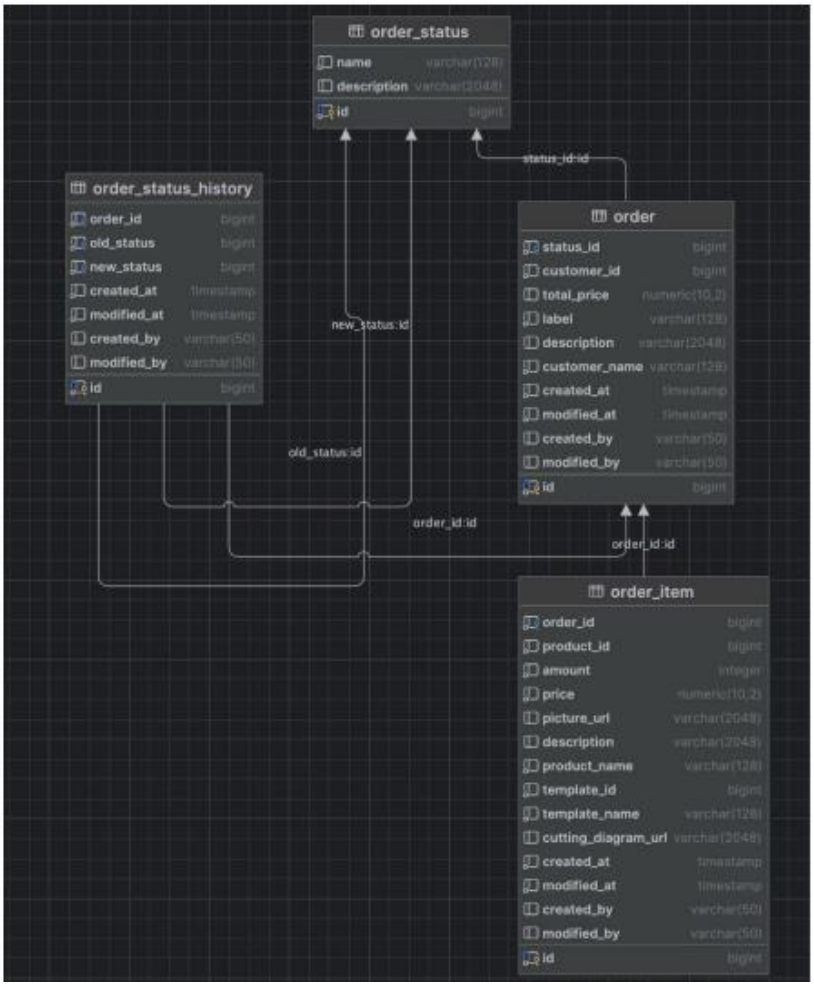

Рисунок 2.8 – Діаграма БД сервісу замовлень

2.3.4 Сервіс повідомлень

Цей сервіс відповідає за складання та відправлення повідомлень, тому все, що йому потрібно зберігати - це шаблони повідомлень (рис. 2.9).

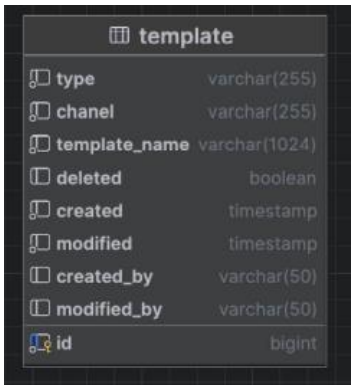

Рисунок 2.9 – Діаграма БД сервісу повідомлень

2.3.5 Сервіс відгуків

Сервіс відповідатиме за створення та публікацію відгуків, тому для зберігання йому потрібен шаблон самого відгуку (рис. 2.10).

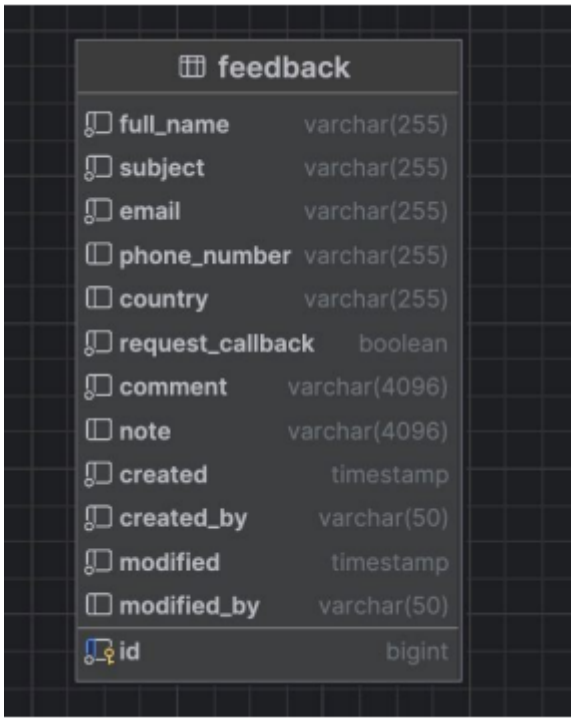

Рисунок 2.10 – Діаграма БД сервісу відгуків

2.4 Розробка веб-застосунку

У цьому підрозділі буде описано процес розробки кожного сервісу веб-застосунку.

2.4.1 API -шлюз

API -шлюз або API-Gateway – це базовий патерн програмування, навколо якого будується мікросервісна архітектура [20]. За змістом це маршрутизатор API -запитів, який отримує на вхід усі запити клієнта, обробляє та розсилає різним сервісам, а потім відсилає відповідь назад клієнту.

Шлюз виконує такі функції:

– маршрутизація: перенаправляє запити до відповідних мікросервісів;

– безпека: може забезпечувати аутентифікацію та авторизацію запитів;

– обробка: може виконувати додаткові операції над запитами та відповідями, такі як логування, моніторинг, кешування та інші.

Основне завдання api-gateway — спростити взаємодію клієнтів із мікросервісами, приховуючи складність внутрішньої архітектури та надаючи єдиний API для доступу до різних сервісів.

Основою цього сервісу є клас RoutesConfig (Додаток Б), який відповідає за конфігурацію маршрутів для перенаправлення запитів до відповідних мікросервісів. Цей клас анотований як @Configuration, що робить його класом конфігураційним Spring.

Основні функції класу RoutesConfig:

– читання конфігураційних параметрів: клас використовує анотацію @Value для читання URL- адрес мікросервісів з конфігураційних файлів;

– визначення маршрутів: метод customRouteLocator визначає маршрути для різних мікросервісів за допомогою об'єкта RouteLocatorBuilder. Усередині методу використовуються методи route для вказівки шляху запиту та відповідної URL -адреси мікросервісу;

– фільтри та переписування шляхів: для кожного маршруту використовується фільтр rewritePath для переписування шляху запиту. Також додається фільтр PathFilter для додаткової обробки запитів.

Як фільтр користувача використовується клас PathFilter, який реалізує інтерфейс «GatewayFilter», надаючи метод filter, який використовується для обробки кожного вхідного запиту перед його передачею на цільовий мікросервіс. Завдання фільтра полягає в тому, щоб отримати оригінальний шлях запиту, модифікувати його, додаючи туди свій заголовок і передати цей запит далі по ланцюжку.

Приклади запитів:

– auth_route: перенаправляє запити з /api/auth/** до сервісу аутентифікації;

– order_route: перенаправляє запити з /api/order/** до сервісу замовлень;

– notification_route: перенаправляє запити з шляхами /api/notification/** до сервісу сповіщень;

– product_route: перенаправляє запити з /api/product/** до сервісу продуктів;

– feedback_route: перенаправляє запити з /api/feedback/** до сервісу зворотного зв'язку.

У результаті даний сервіс дозволяє централізовано керувати маршрутами та конфігураціями запитів, забезпечуючи гнучкість і зручність зміну різних маршрутів, шляхом відкидання необхідності змінювати код кожного окремого сервісу, і налаштування всього, що стосується маршрутизації в одному місці.

2.4.2. Сервіс авторизації

Цей сервіс потрібен для того, щоб реалізувати аутентифікацію та авторизацію користувача до програми. Це потрібно для того, щоб визначати та зберігати інформацію про конкретного користувача, його права та ролі на сайті.

Даний сервіс містить 2 основні сутності: Роль та Користувач. Ці сутності

прив'язані до відповідної таблиці в БД.

Сутність "Роль" відповідає за зберігання ролей користувача. Роль має лише два поля, представлені у табл. 2.1.

Таблиця 2.1 – Основні поля сутності «Роль»

Тип даних	Назва	Опис
long	id	Ідентифікатор ролі
string	name	Назва ролі

Сутність «Користувач» відповідає за зберігання інформації про людину, її поля представлені у табл. 2.2.

Таблиця 2.2 – Основні поля сутності «Користувач»

Тип даних	Назва	Опис
long	id	Ідентифікатор користувача
string	username	Логін користувача у додаток
string	password	Пароль користувача у зашифрованому вигляді
string	firstName	Ім'я користувача
string	lastName	Прізвище користувача
string	email	Пошта
string	phoneNumber	Номер телефону
role	roles	Список ролей користувача
boolean	isEmailConfirmed	Прапори для підтвердження пошти
string	resetPasswordToken	Токен для скидання пароля

Обидві ці сутності мають відповідні таблиці у БД. Ще є третя таблиця – role_user. Вона зберігає всі зв'язки між користувачами та ролями.

Аутентифікація користувача працює на базі фреймворку Spring Security

для Java, який має різні варіації автентифікації та авторизації користувача. У моєму випадку процес працює за таким принципом:

1. Користувач вводить логін та пароль.
2. Створюється токен з ім'ям користувача та його паролем.
3. Spring завантажує дані користувача, декодує пароль та перевіряє його.
4. Якщо всі перевірки пройшли успішно, то повертає автентифікований об'єкт.
5. Якщо автентифікація не вдалася, викидає відповідний виняток.

Після того, як користувач автентифікувався, він додається до SecurityContext, який надає глобальний доступ до контексту безпеки. Це потрібно для наступних дзвінків у межах поточного потоку, оскільки вони можуть використовувати дані про поточного автентифікованого користувача.

Також варто відзначити той факт, що SecurityContext зберігає інформацію про аутентифікацію користувача тільки в рамках поточного запиту і для того, щоб при кожному наступному запиті нам не доводилося проводити аутентифікацію заново, ми генеруємо accessToken, який зберігається на клієнтській частині і кожного запиту передається на сервер. Наявність цього токена вказує на те, що власник токена має право доступу до всіх захищених ресурсів.

Сервіс авторизації має своє API, яке реалізовано у класі AuthControllerjava (Додаток В) та містить п’ять методів:

– InviteUser – метод для запрошення користувача. На даний момент працює як реєстрація;
– SignIn – метод для аутентифікації користувача;
– RefreshAccessToken – метод для оновлення accessToken;
– RefreshPassword – метод для оновлення поточного пароля;
– ConfirmEmail – метод для підтвердження пошти.

Сервіс авторизації спілкується з сервісом повідомлень та надсилає повідомлення користувачу на пошту. Це відбувається, коли користувачеві потрібно підтвердити email або, наприклад, оновити пароль. Сервіси

спілкуються за допомогою RabbitMq і в сервісі авторизації за відправку повідомлень відповідає клас RabbitMqMessageProducer. Клас має всього два методи, перший готує об'єкт передачі, другий конвертує об'єкт в json і відправляє в обмінник, який у свою чергу розподіляє повідомлення в потрібні черги.

У результаті даний сервіс відіграє дуже важливу роль, він аутентифікує та авторизує користувача, і генерує для нього його особистий токен доступу (accessToken), який потрібен користувачеві, щоб мати можливість надсилати запити до інших послуг, не проходячи при цьому повторну авторизацію для кожного нового запиту. Всі інші послуги мають свій власний функціонал, який отримує всю інформацію про користувача з токена.

2.4.3 Сервіс повідомлень

Даний сервіс відповідає за надсилання повідомлень користувачам. Поки що реалізовано надсилання повідомлень тільки на пошту користувача.

Сервіс повідомлень отримує повідомлень від інших сервісів та надсилає їх. Отримання повідомлень реалізовано за допомогою RabbitMq та працює асинхронно. Тобто, коли інший сервіс надсилає нам повідомлення, яке ми повинні надіслати користувачеві, цей процес виконується фоново і аж ніяк не впливає на поточну роботу програми.

Конфігурація RabbitMq задається у класі RabbitMQConfig (Додаток Г), який позначений як клас конфігурації Spring. У цьому класі відбувається таке:

– створюються всі основні компоненти обміну повідомленнями, якщо конкретніше, це: обмінник (повідомлення спочатку надходять у обмінник, а звідси він їх маршрутизує у необхідні черги) і черги (як основна, і dead-letter);

– біндінг (черги прикріплюються до обмінника, використовуючи спеціальні ключі);

– конфігурація connectionFactory (це об'єкт підключення сервісу до RabbitMq);

– налаштування шаблонів для надсилання повідомлень;

– ініціалізація конвертора, який перетворює повідомлення на JSON і назад (як конвертор використовується бібліотека Jackson).

"Точкою" прийому повідомлень є клас NotificationConsumer.java (Consumer - споживач), який за розкладом отримує повідомлення з черги. Класу відправки повідомлень (Producer) немає, оскільки сервіс повідомлень сам не надсилає повідомлення іншим сервісам.

Для надсилання повідомлень на пошту використовується клас EmailService.java, який має лише 2 методи:

– prepareMessage - готує повідомлення, проставляючи тему та відправника;

– sendMessage - надсилає повідомлення за допомогою JavaMailSender із фреймворку Spring.

Підбиваючи підсумки сервісу повідомлень, зараз він вміє надсилати повідомлень на пошту користувачам, у майбутньому будуть й інші види повідомлень. Сервіс використовує механізми роботи з поштою Spring, а з іншими сервісами спілкується за допомогою RabbitMq. На даний момент сервіс повідомлень отримує повідомлення лише від сервісу авторизації.

2.4.4 Сервіс замовлень

Цей сервіс відповідає за створення, отримання та зберігання інформації про замовлення. Він є звичайним API, що приймає запити на створення, отримання, видалення та зміни замовлень.

Сервіс має сутність замовлення та позиції замовлення, які зберігаються в БД.

Сутність замовлення - "Order" зберігається в таблиці order, її поля відображені в табл. 2.3.

Таблиця 2.3 – Основні поля сутності «Замовлення»

Тип даних	Назва	Опис
long	id	Ідентифікатор замовлення
string	status	Статус замовлення
long	customerId	Ідентифікатор покупця
bigDecimal	totalPrice	Підсумкова ціна
string	label	Назва замовлення
string	description	Опис замовлення
string	customerName	Ім'я покупця

Сутність позиції замовлення - OrderItem зберігається в таблиці order_item, його поля відображені в табл. 2.4.

Таблиця 2.4 – Основні поля сутності «Позиція»

Тип даних	Назва	Опис
long	orderId	Ідентифікатор замовлення
long	productId	Ідентифікатор продукту
integer	amount	Кількість
bigDecimal	price	Ціна
string	pictureUrl	Зображення позиції
string	description	Опис позиції
string	productName	Назва продукти

Замовлення безпосередньо не зберігає в собі інформацію про свої позиції. У формах, де потрібно відобразити всі позиції замовлення, вони будуть підтягуватися з таблиці order_item за ідентифікатором замовлення.

Також є додаткова таблиця, яка зберігає інформацію про статуси, вона називається order_status та зберігає його ім'я та опис. Вона теж безпосередньо не пов'язана із замовленням, але отримати статус замовлення можна з цієї таблиці

за його ідентифікатором.

Остання таблиця у цьому сервісі – order_status_history, яка зберігає історію статусів певного замовлення.

Основою даного сервісу є клас OrderController.java, який містить усі методи, які доступні користувачеві за запитом API . Він містить такі методи:

– getOrderById - метод отримання статусу за його ідентифікатором;
– createOrder - спосіб створення замовлення;
– updateOrder – змінити інформацію в існуючому замовленні;
– deleteOrderById - видалення замовлення за його ідентифікатором;
– FindAllOrders - спосіб отримання всіх замовлень;
– updateOrderStatus – метод оновлення статусу замовлення.

Сервіс замовлень є API і дозволяє проводити базові операції над замовленням: створення, видалення, отримання, оновлення. Також зберігає ці замовлення у БД. На даний момент сервіс майже не має іншої логіки, тільки автоматичне формування деяких полів (наприклад, label), але в майбутньому можливо з'являться ще якісь вимоги до обробки замовлення.

2.4.5 Сервіс товарів

Він товарів має схожу будову із сервісом замовленням. Сервіс товарів виконує роль сховища та API для товару.

Клас має одну сутність «Товар» та відповідну таблицю у БД - product, її поля відображені у табл. 2.5.

За API відповідає ProductController, який має базові CRUD методи - створення, видалення, оновлення, отримання.

В результаті сервіс має дуже невелику функціональність, але був виділений окремо для зручності і для того, щоб зберігати велику кількість продуктів в окремому місці і не навантажувати цим інші послуги.

Таблиця 2.5 – Основні поля сутності «Товар»

Тип даних	Назва	Опис
long	id	Ідентифікатор товару
String	name	Назва товару
string	type	Категорія товару
bigDecimal	price	Ціна товару
string	description	Опис товару
string	specifications	Характеристики
integer	amount	Кількість

2.4.6 Сервіс відгуків

Останній сервіс – це сервіс відгуків. Сервіс відповідає за зберігання та роботу з відгуками.

Клас має одну сутність «Відгук» і таблицю feedback, у якій і зберігаються відгуки. Основні поля сутності відображені у табл. 2.6.

Таблиця 2.6 – Основні поля сутності «Відгук»

Тип даних	Назва	Опис
long	id	Ідентифікатор товару
string	fullName	Ім'я та прізвище користувача
string	subject	Тема
string	email	Пошта
string	phoneNumber	Номер телефону
string	country	Країна
string	comment	Коментар
string	note	Нотатка
boolean	requestCallback	Прапори, що показують, чи потрібно передзвонювати клієнту

Відгуки можна залишати під якимось конкретним товаром або просто у спеціальній формі. Також можна буде поставити галочку та попросити, щоб компанія передзвонила клієнту та відповіла на його запитання. Відгук у такому разі передбачає не обговорення будь-якого товару, а швидше залишення заявки з його придбання.

Сам сервіс також має структуру звичайного API і має такі методи у контролері (FeedbackController):

– createFeedback - створення відгуку;
– getFeedbackById - отримання відгуку щодо ідентифікатора;
– getFeedbackResponsePage - отримує усі відгуки;
– addNoteToFeedBack - додасть нотатку до відгуку.

В даний час сервіс має невеликий функціонал, але в майбутньому планується більше нових функцій. Наприклад, автоматична модерація відгуків, автоматизація відповідей для відгуків, які вимагають втручання співробітників [21]. Як було сказано, зараз відгуки мають на увазі швидше заявку на те, щоб з вами зв'язалися, але в майбутньому також буде реалізована можливість обговорювати товари та ставити свої оцінки.

2.4.7 Розгортання системи

Для розгортання використовується Docker. Це система для контейнеризації додатків, вона допомагає запакувати всі додатки (сервіси) в один контейнер і розташувати їх в одному середовищі [22]. Кожен сервіс має спеціальний файл Dockerfile, який відповідає за складання свого сервісу. Як приклад можна навести файл сервісу замовлень (лістинг 2.1).

Лістинг 2.1 – Dockerfile сервісу замовлень

```
FROM maven:3.9.6-eclipse-temurin-21-alpine AS build
WORKDIR /app
COPY pom.xml ./
RUN mvn dependency:go-offline
COPY src ./src
RUN mvn package

FROM openjdk:21-slim
WORKDIR /app
COPY --from=build /app/target/order-1-exec.jar ./app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Цей файл містить усі основні етапи складання для сервісу. На першому етапі ми завантажуємо всі потрібні образи, копіюємо конфігураційний файл і всі вихідні коди, а потім упаковуємо всю програму в jar файл. На другому етапі ми копіюємо готовий jar файл у потрібну директорію та запускаємо його. У такий спосіб запускається сервіс.

Для того щоб можна було розгортати всі сервіси за раз, а не по одному, існує файл docker-compose.yml (Додаток Д). Docker compose – це інструмент для визначення та керування кількома контейнерами за один раз [23]. Він містить опис кожного сервісу і може розгорнути всі послуги за раз. В описі кожного контейнера є параметр «context», який містить шлях до Dockerfile потрібного сервісу, цим шляхом docker compose запускає потрібний контейнер і таким чином розгортає кожен сервіс. Також в опис контейнерів є інші параметри, які, наприклад, визначають змінні оточення для кожного сервісу, відповідність портів та залежності сервісів один від одного (наприклад, коли один сервіс використовує інший і повинен запуститися пізніше за нього).

2.5 Висновки до другого розділу

У цьому розділі було розглянуто основні засоби розробки для програми, а також спроектовано схему самого застосунку та спроектовано схеми БД для кожного сервісу.

Також було описано процес розробки веб-застосунку на мікросервісній архітектурі. Описано призначення кожного сервісу, його функціонал та основні точки доступу (для API). Представлено атрибутивний склад основних сутностей застосунку, і описано їхню взаємодію. Поки що спілкування між сервісами працює тільки для двох з них, але в майбутньому планується додати ще більше взаємодій. Також було розібрано спосіб розгортання мікросервісної програми за допомогою системи контейнеризації Docker.

3 ПОРІВНЯННЯ СПОСОБІВ ВЗАЄМОДІЇ МІКРОСЕРВІСІВ МІЖ СОБОЮ

3.1 Загальна інформація для процесу порівняння

Як дослідження було ухвалено рішення порівняти методи взаємодії між собою. Загалом якщо заглибиться в документацію, можна зрозуміти який спосіб краще у яких ситуаціях. Очевидно наступне:

– Kafka краще використовувати, коли потрібно безперервно відправляти потік якихось даних;

– Rest API зручний для маленьких додатків, тому що дуже простий у розгортці і зрозумілий у використанні;

– gRPC найчастіше використовують при великих обсягах завдяки його вбудованій особливості стиснення даних;

– RabbitMq зручний у випадках, коли потрібно просто надіслати повідомлення та забути про нього.

Здається, все цілком зрозуміло, але як поведуть себе ці технології не в «своїй» ситуації, чи можливо, що при роботі з невеликою кількістю даних Kafka буде працювати теж цілком впевнено, або що в деяких, здавалося б, не відповідних сценаріях асинхронність RabbitMq буде цілком доречною. Щоб з'ясувати це, я провів невелике дослідження. Суть дослідження полягає в тому, що є два сервіси, між якими відправляються різні дані та заміряються показники.

Для порівняння було обрано такі метрики:

1. Час відгуку - час, необхідний для надсилання повідомлення від одного сервісу до іншого.

2. Пропускна спроможність – кількість повідомлень, які можуть бути оброблені за певну кількість часу.

3. Навантаження на процесор - кількість ресурсів процесора при передачі даних.

4. Використання пам'яті - кількість пам'яті, що використовується під час надсилання повідомлень.

Для тестування використовуватимуться сценарії з різним розміром даних (1 Мб, 100 Мб, 500 Мб). Дані будуть передаватися в json форматі, так як формат не надто впливає швидкість передачі, він, швидше, впливає на час серіалізації/дессеріалізації даних.

Тестування буде проводитись на готовому застосунку. Дані будуть надсилатися між сервісом авторизації та сервісом повідомлень. Як дані використовуватимуться повідомлення різної довжини, які потрібно відправляти користувачеві.

Тестування проводилося на Macbook Air M1, 16 Гб оперативної пам'яті, 8 ядер.

3.2 Час відгуку

Час відгуку - це час, необхідний для відправлення повідомлення від одного сервісу до іншого. Для підготовки до цих тестів потрібно, щоб обидва сервіси були розгорнуті, перший (сервіс авторизації) надсилатиме повідомлення, другий (сервіс повідомлень) отримуватиме їх і повертатиме відповідь, і вже після отримання відповіді записуватиметься час. У другому сервісі не буде проводитися жодних маніпуляцій над даними, відразу надсилатиметься відповідь.

Візьмемо як приклад Rest API і подивимося, як це виглядає.

Сервіс авторизації відповідатиме за надсилання документа. Він містить один клас RestSenderjava (лістинг 3.1). Нам потрібна лише одна залежність для цього класу – RestTemplate. Це службовий клас у Spring Framework, який спрощує відправку HTTP -повідомлень та обробку відповіді. І також нам потрібна константа, яка визначатиме URL адресу, на яку ми надсилатимемо запит.

Лістинг 3.1 – Клас RestSender.java для надсилання повідомлень за допомогою Rest API

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class RestSender {
    private static final String SERVER_URL =
"http://localhost:8080/api/data";
    private RestTemplate restTemplate = new RestTemplate();

    public long sendRequest(byte[] data) {
        long startTime = System.currentTimeMillis();
        ResponseEntity<String> response =
restTemplate.postForEntity(SERVER_URL, data, String.class);
        long endTime = System.currentTimeMillis();
        return endTime - startTime;
    }

    public void prepareMessage() {
        byte[] data = new byte[1024 * 1024]; // 1 MB
        for (int i = 0; i < 10; i++) {
            long latency = sendRequest(data);
            System.out.println("Latency: " + latency + " ms");
        }
    }
}
```

У цьому класі є метод sendRequest, на вхід якого приходить масив байт, який ми готуємо в методі prepareMessage. Для зручності тестування передається масив байт потрібного розміру, а чи не рядок. Спочатку потрібно заміряти початковий час у змінній startTime, в яку записується поточний час, а далі просто надсилаємо запит за допомогою методу postForEntity із бібліотеки RestTemplate. Конвертувати масив байт в json не потрібно, тому що метод postForEntity робить це автоматично. Після того як я отримав відповідь залишається тільки заміряти час закінчення роботи endTime і як результат беремо різницю між кінцевим і початковим часом, це і буде час, за який відбулася відправка даних та отримання відповіді.

Сервіс повідомлень прийматиме дані та надсилатиме відповідь. Він також має новий клас, DataController.java (лістинг 3.2), який містить один метод - recieveData. Цей метод не робить жодної обробки отриманих даних, а просто

відразу надсилає нам відповідь.

Лістинг 3.2 – Клас DataController.java для отримання надісланого повідомлення

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DataController {
    @PostMapping("/api/data")
    public String receiveData(@RequestBody byte[] data) {
        // Обработка данных
        return "Received";
    }
}
```

Результати для набору даних вагою 1 Мб, 100 Мб та 500 Мб представлені на табл. 3.1, 3.2 та 3.3 відповідно.

Таблиця 3.1 - Час відгуку для даних об'ємом 1 МБ

№ з/п	Rest API (мс)	gRPC (мс)	RabbitMQ (мс)	Kafka (мс)
1.	150	50	120	80
2.	148	52	115	79
3.	151	49	122	81
4.	149	51	118	82
5.	150	50	121	80
6.	147	53	117	83
7.	152	48	119	78
8.	148	50	120	81
9.	150	52	118	80
10.	149	51	121	79
Середнє	**149.4**	**50.6**	**119.1**	**80.3**

Таблиця 3.2 - Час відгуку для даних об'ємом 100 МБ

№ з/п	Rest API (мс)	gRPC (мс)	RabbitMQ (мс)	Kafka (мс)
1.	1700	500	1200	800
2.	1680	520	1150	790
3.	1710	490	1220	810
4.	1690	510	1180	820
5.	1700	500	1210	800
6.	1670	530	1170	830
7.	1720	480	1190	780
8.	1680	500	1200	810
9.	1700	520	1180	800
10.	1690	510	1210	790
Середнє	**1694.0**	**506.0**	**1191.0**	**802.0**

Таблиця 3.3 - Час відгуку для даних об'ємом 500 МБ

№ з/п	Rest API (мс)	gRPC (мс)	RabbitMQ (мс)	Kafka (мс)
1.	8500	2500	6000	4000
2.	8400	2600	5750	3950
3.	8550	2450	6100	4050
4.	8450	2550	5900	4100
5.	8500	2500	6050	4000
6.	8350	2650	5850	4150
7.	8600	2400	5950	3900
8.	8400	2500	6000	4050
9.	8500	2600	5900	4000
10.	8450	2550	6050	3950
Середнє	**8475.0**	**2530.0**	**5955.0**	**4015.0**

Для кожного набору даних і кожної технології було проведено по 10 тестових запусків з даних і можна стверджувати наступне:

– gRPC показує найменший час відгуку для обох форматів даних завдяки бінарній серіалізації та використанню HTTP/2;

– Kafka також демонструє хорошу продуктивність з помірним часом відгуку;

– RabbitMQ показує середні результати, які можуть бути пов'язані з асинхронною природою системи;

– Rest API має найвищий час відгуку, завдяки overhead. Over head - це різні накладні витрати, які відбуваються під час передачі даних, наприклад, якщо мова про HTTP, він включає додаткові заголовки в запит і у відповідь, що трохи обтяжує їх, а також витрачається час на їх додавання.

Спираючись на результатии, наведені на рис. 3.1, можна зробити такий висновок, якщо ви збираєтеся передавати важкі дані, то найкращим способом буде є gRPC, так як у нього є вбудована бінарна серіалізація даних, що сильно полегшує ваш запит, також Kafka показує себе досить непогано, гірше, ніж gRPC, але набагато краще, ніж решта способів.

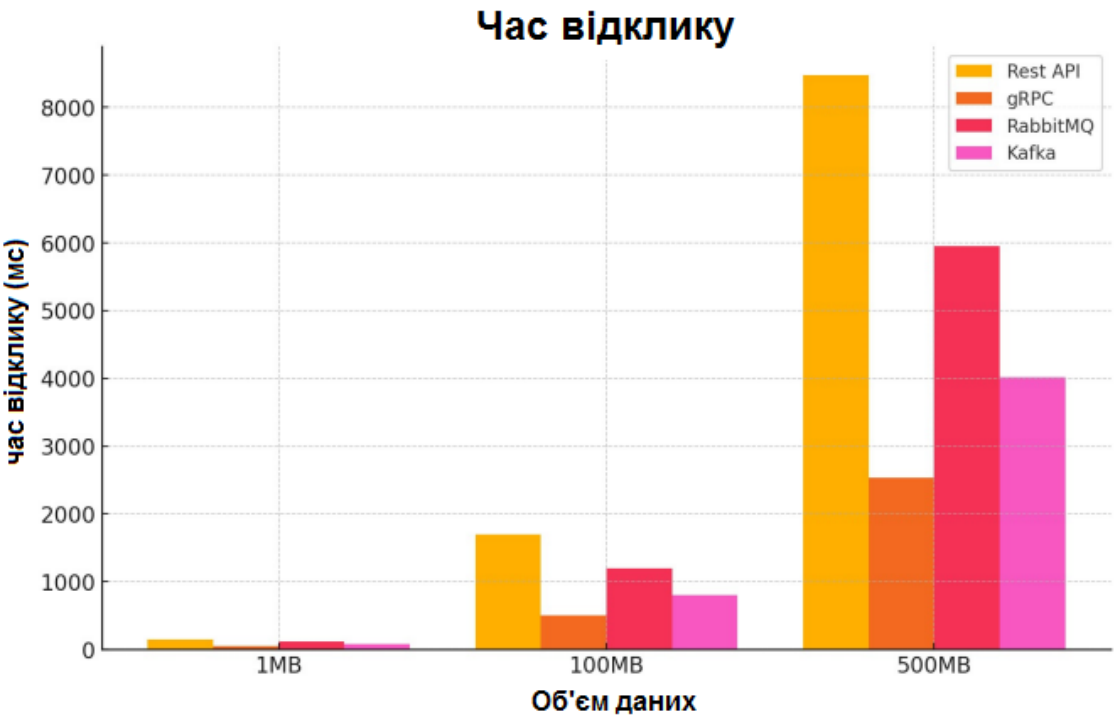

Рисунок 3.1 – Діаграма порівняння способів взаємодії за часом відгуку

Таким чином можна відзначити, що, незважаючи на те, що основною перевагою Kafka є безперервна передача даних, але і при простій відправці він показує досить хороші результати і його можна використовувати в ситуаціях, коли вам не потрібна безперервна передача. Що стосується інших способів, то Rest API в цілому погано підходить для передачі великих даних, якщо вам

важлива швидкість, але його дуже рідко використовують для спілкування між сервісами, найчастіше використовують для виконання запитів з клієнтської частини. Щодо RabbitMQ, тут теж все досить очікувано, тому що через асинхронність він не розрахований на швидкість передачі, як правило при використанні RabbitMq ви відправляєте повідомлення в чергу і вам не дуже важливо, як довго воно буде оброблятися (в розумних межах звичайно).

3.3 Пропускна спроможність

Пропускна спроможність - це кількість повідомлень, які можуть бути оброблені за певний період. Зміст достатньо простий, щоб виміряти пропускну спроможність, нам потрібні середні значення часу, за який сервіс виконуватиме запит, тобто це середні значення часу відгуку, які ми вимірювали в попередньому підрозділі. Далі для визначення пропускної спроможності нам потрібно скористатися формулою (3.1):

$$\text{Пропускна спроможність } (MB/s) = \frac{\text{Середня затримка } (s)}{\text{Розмір даних } (MB)} \tag{3.1}$$

Результати середньої затримки, виражені в секундах, можна побачити на табл. 3.4.

Таблиця 3.4 - Середня затримка в секундах

Обсяг даних	Rest API (c)	gRPC (c)	RabbitMQ (c)	Kafka (c)
1 Мб	0,1494	0,0506	0,1191	0,0803
100 Мб	1,694	0,506	1,191	0,802
500 Мб	8,475	2,53	5,955	4,015

Тепер, якщо порахувати пропускну здатність за формулою (3.1), то отримаємо результати, наведені в табл. 3.5.

Таблиця 3.5 - Середні значення пропускної спроможності

Обсяг даних	Rest API (Мб/с)	gRPC (Мб/с)	RabbitMQ (Мб/с)	Kafka (Мб/с)
1 Мб	6,69	19,76	8,39	12,46
100 Мб	59,02	197,63	84	124,69
500 Мб	58,98	197,63	84,01	124,56

З результатів, представлених на рис. 3.2, можна зробити висновки, що пропускна здатність для великих обсягів даних (100 і 500 МБ) практично постійна, тому що середня затримка (вимірюється в секундах) залишається практично постійною для цих обсягів даних. Це відбувається тому, що час передачі даних щодо обсягу даних зберігається на відносно постійному рівні завдяки ефективній обробці та передачі даних за обраними методами зв'язку (Rest API, gRPC, RabbitMQ, Kafka).

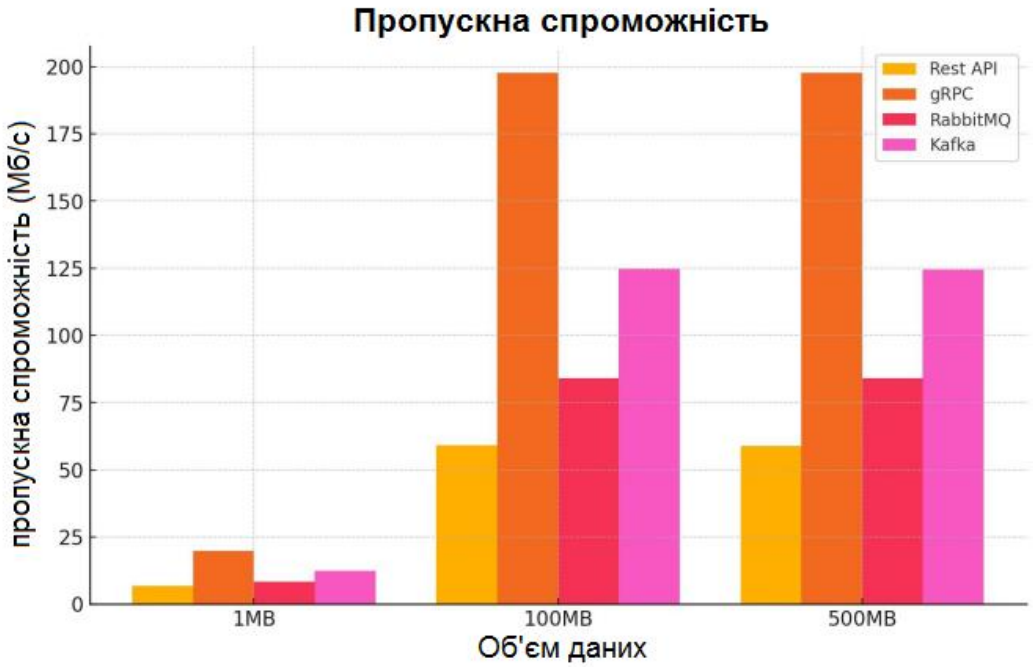

Рисунок 3.2 – Діаграма порівняння способів взаємодії щодо пропускної спроможності

Таким чином, при збільшенні обсягу даних середня затримка залишається стабільною, що призводить до майже постійної пропускної здатності для різних методів взаємодії навіть за різних обсягів даних. Ще можна відзначити, що

gRPC є найкращим вибором для високопродуктивних систем, Kafka також виглядає досить середньо, чого не скажеш про Rest Api і RabbitMq, їх результати виглядають не дуже вражаючими для ситуацій, коли нам потрібно надсилати багато важких повідомлень за обмежений період часу.

3.4 Навантаження на процесор

У цьому тесті необхідно перевірити, яке навантаження кожен спосіб відправки створює на процесор комп'ютера.

Для виконання тестування нам потрібно налаштувати всі послуги. Далі при надсиланні повідомлення нам потрібно просто дивитися моніторинг нашого процесу, записуючи показники навантаження на CPU.

Для перегляду процесів я використовую вбудовану утиліту "ps", також був підготовлений bash скрипт, який записуватиме значення навантаження на процесор кожну секунду. Для того, щоб відокремити кожну відправку повідомлень одну від одної, після кожного запиту виставили паузу в 1 секунду, за допомогою стандартного функціоналу java. Підсумкові результати даних розміром 1 Мб, 100 Мб і 500 Мб можна побачити на табл. 3.6, 3.7 і 3.8 відповідно.

Таблиця 3.6 - Результати тестування навантаження на процесор для даних об'ємом 1 Мб

№ з/п	Rest API (%)	gRPC (%)	RabbitMQ	Kafka (%)
1.	20	15	10	12
2.	18	14	11	11
3.	21	16	10	13
4.	19	15	9	12
5.	20	14	10	11
6.	22	15	11	12
7.	19	14	10	12
8.	18	16	9	11
9.	21	15	10	13
10.	20	15	11	12
Середнє	**19,8**	**14,9**	**10,1**	**11,9**

Таблиця 3.7 - Результати тестування навантаження на процесор для даних об'ємом 100 Мб

№ з/п	Rest API (%)	gRPC (%)	RabbitMQ (%)	Kafka (%)
1.	30	25	22	20
2.	29	24	21	19
3.	31	26	23	21
4.	30	25	22	20
5.	29	24	21	19
6.	31	25	23	20
7.	30	24	22	19
8.	29	26	21	21
9.	31	25	23	20
10.	30	25	22	20
Середнє	**30.0**	**24,9**	**21,9**	**19,9**

Таблиця 3.8 - Результати тестування навантаження на процесор для даних об'ємом 500 Мб

№ з/п	Rest API (%)	gRPC (%)	RabbitMQ	Kafka (%)
1.	35	28	25	22
2.	34	27	24	21
3.	36	29	26	23
4.	35	28	25	22
5.	34	27	24	21
6.	36	28	26	22
7.	35	27	25	21
8.	34	29	24	23
9.	36	28	26	22
10.	35	28	25	22
Середнє	**34.9**	**27,9**	**24,9**	**21,9**

За результатами на рис. 3.3 видно, що RabbitMq і Kafka навантажують процесор набагато менше, пов'язано це з їх асинхронною природою, так як застосунок за фактом не чекає відповіді відразу (не зависає, поки не прийде відповідь), і тому навантаження краще розподіляється за часом, також це досить сучасні технології і вони мають ряд переваг з ними знижує рівень

навантаження, та й головне, це те, що вони не мають накладних витрат пов'язаних з протоколом HTTP.

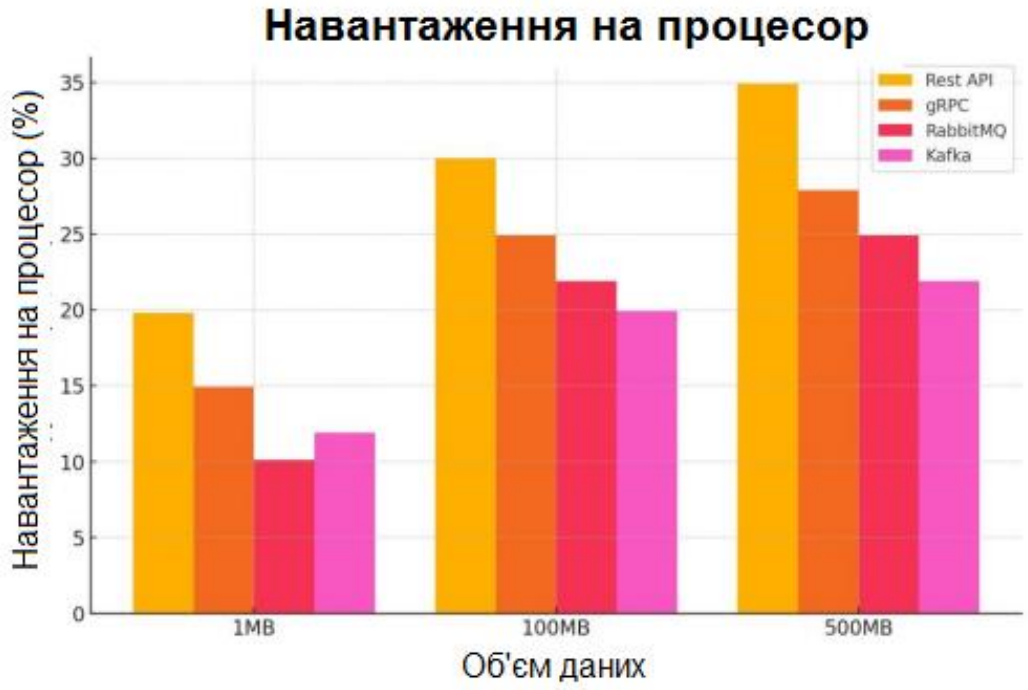

Рисунок 3.3 – Діаграма порівняння способів взаємодії по навантаженню процесора

Звідси можна дійти висновку, що RabbitMq і Kafka є більш вдалими варіантами, якщо йдеться про завдання, котрі потребують високої продуктивності та ефективного використання ресурсів. Це може бути корисним у ситуаціях, коли у вас дуже великі обсяги даних або, наприклад, дуже обмежені ресурси.

3.5 Споживання пам'яті

В останньому тесті потрібно вимірюватиму обсяг пам'яті, що використовується при передачі повідомлень.

Для того, щоб виміряти кількість пам'яті, що споживається, нам потрібно зробити все те саме, що і з навантаженням на процесор, тільки тепер дивимося на споживання пам'яті.

Для вимірювань буде також використовувати утиліту «ps» і той же скрипт, в якому був змінений параметр, що записується.

Результати для 1 Мб, 100 Мб, 500 Мб можна побачити на табл. 3.9, 3.10 та 3.11 відповідно.

Таблиця 3.9 - Результати тестування споживання пам'яті даних об'ємом 1 Мб

№ з/п	Rest API (%)	gRPC (%)	RabbitMQ (%)	Kafka (%)
1.	1	0,8	0,5	0,6
2.	1,1	0,9	0,6	0,7
3.	1	0,8	0,5	0,6
4.	1,1	0,9	0,6	0,7
5.	1	0,8	0,5	0,6
6.	1,1	0,9	0,6	0,7
7.	1	0,8	0,5	0,6
8.	1,1	0,9	0,6	0,7
9.	1	0,8	0,5	0,6
10.	1,1	0,9	0,6	0,7
Середнє	**1,05**	**0,85**	**0,55**	**0,65**

Таблиця 3.10 - Результати тестування споживання пам'яті даних об'ємом 100 Мб

№ з/п	Rest API	gRPC (%)	RabbitMQ	Kafka (%)
1.	2	1,6	1,2	1,3
2.	2,1	1,7	1,3	1,4
3.	2	1,6	1,2	1,3
4.	2,1	1,7	1,3	1,4
5.	2	1,6	1,2	1,3
6.	2,1	1,7	1,3	1,4
7.	2	1,6	1,2	1,3
8.	2,1	1,7	1,3	1,4
9.	2	1,6	1,2	1,3
10.	2,1	1,7	1,3	1,4
Середнє	**2,05**	**1,65**	**1,25**	**1,35**

Таблиця 3.11 - Результати тестування споживання пам'яті даних об'ємом 500 Мб

№ з/п	Rest API (%)	gRPC (%)	RabbitMQ (%)	Kafka (%)
1.	5	4	3,2	3,5
2.	5,1	4,1	3,3	3,6
3.	5	4	3,2	3,5
4.	5,1	4,1	3,3	3,6
5.	5	4	3,2	3,5
6.	5,1	4,1	3,3	3,6
7.	5	4	3,2	3,5
8.	5,1	4,1	3,3	3,6
9.	5	4	3,2	3,5
10.	5,1	4,1	3,3	3,6
Середнє	**5,05**	**4,05**	**3,25**	**3,55**

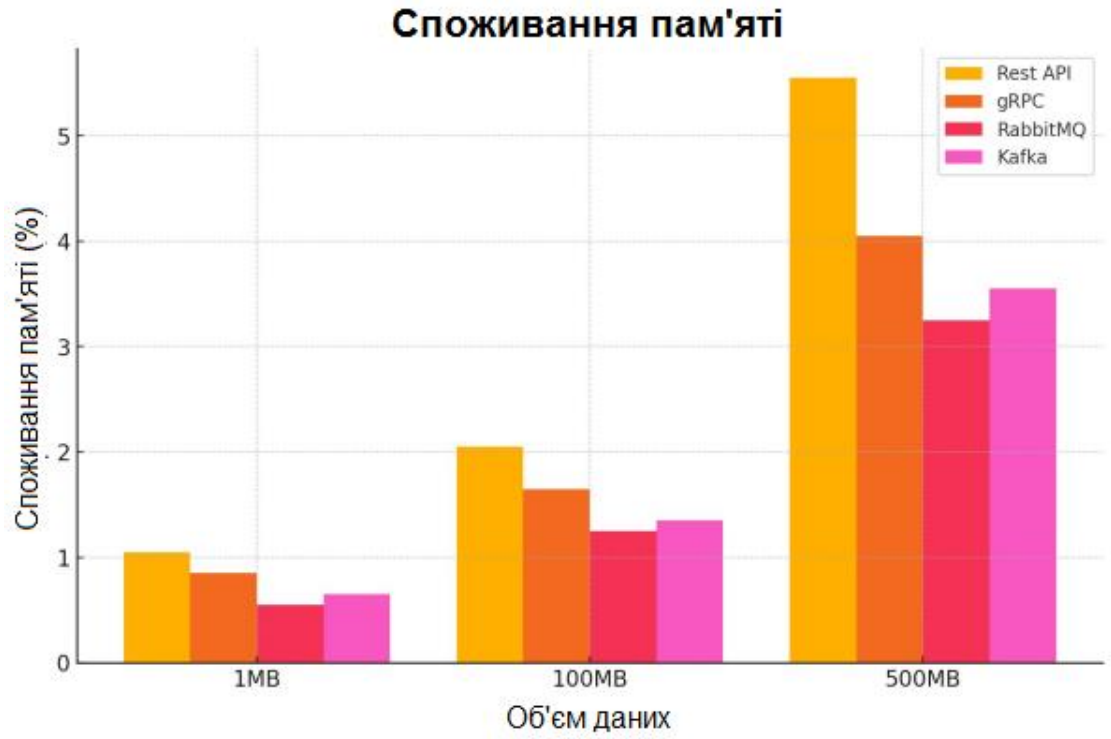

Рисунок 3.4 – Діаграма порівняння способів взаємодії споживання пам'яті

За результатами, наведеними на рис. 3.4, можна сказати наступне:

– Rest API поводиться досить погано, пов'язано це з накладними витратами HTTP;

– gRPC показує результати краще, багато завдяки використанню двійкового формату Protobuf;

– RabbitMq та Kafka мають досить хороші результати, знову ж таки через те, що передають двійкові дані, а також завдяки своїм алгоритмам.

3.4 Висновки до третього розділу

У третьому розділі роботи проведено дослідження, порівнюючи 4 найпопулярніші способи передачі даних між сервісами. Способи досить сильно відрізняються один від одного та використовують різні технології. Це було зроблено в першу чергу з метою - визначити в яких «нестандартних» для того чи іншого способу ситуаціях можна використовувати. RestApi часто використовують для маленьких додатків, тому що вважають, що немає необхідності використовувати щось більш складне (з точки зору конфігурації - Rest API найпростіший спосіб), але з результатів видно, що навіть коли у вас невеликий обсяг даних або невелика кількість запитів, то Rest API виглядає досить слабким у порівнянні наприклад з gRPC, який завдяки протоколу Protobuf і бінарному формату показує для всіх форматів Protobuf і бінарному форматі. Також можна зрозуміти, що Kafka виглядає дуже оптимістично з точки зору споживання ресурсів і має досить середній час відгуку, що в цілому дає йому можливість змагатися з gRPC з точки зору швидкості і з RabbitMQ з точки зору споживання ресурсів. Якщо говорити про RabbitMQ, то він у всіх тестах знаходиться в середині, він дуже хороший для своїх звичних завдань (наприклад, відправлення повідомлень), але в цілому його можна використовувати і в інших ситуаціях, коли не потрібна асинхронність.

Якщо виникає питання з вибором способу взаємодії, то документація відповість на ваші питання, але в певних ситуаціях можна використовувати і нестандартні підходи, які можуть вирішити вашу проблему. Наприклад, якщо вам просто легше працювати з якоюсь технологією або це можна використовувати як заділ на майбутнє. Наприклад, зараз вам не потрібні

асинхронні запити під час розробки, але надалі будуть потрібні. Ви можете застосувати їх зараз, щоб потім не довелося переписувати весь сервіс, і при цьому ви не втратите занадто багато з точки зору продуктивності.

Що стосується конкретного прикладу на розробленому застосунку, то так як відправлення повідомлень не вимагає негайної реакції, то швидкість відгуку не так важлива, а ось показники споживання ресурсів досить важливі, оскільки може бути ситуація, коли доведеться відправляти досить об'ємне повідомлення (наприклад, файл з результатами чогось) і велике споживання ресурсів у такому випадку робить великі витрати на збільшення цих самих ресурсів, саме з такої причини для відправлення повідомлень використовуються асинхронні способи передачі, які програють у швидкості, але при цьому не зупиняють роботу всього застосунку і більш економні з точки зору ресурсів.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Закордонний досвід організації охорони праці в ІТ-компаніях

Стан справ з охороною праці у світі стає все більш актуальною проблемою як для профспілок, так і для міждержавних структур, насамперед Міжнародної організації праці (МОП). МОП розглядає цю тему як частину своєї Програми гідної праці. Підвищена увага до проблем безпеки праці пояснюється в першу чергу тим, що з кожним роком, незважаючи на заходи, що вживаються, у різних країнах зростає рівень виробничого травматизму, у тому числі зі смертельними наслідками, і кількість профзахворювань.

На перший погляд, робота за комп'ютером при дослідженні моделей та побудові системи ПФПК здається безпечною, але саме легковажність до неї може призвести до певних проблем у здоров'ї людини. Професія програміста та інших фахівців ІТ-технологій пов'язана з колосальним розумовим напруженням. Розробники – настільки захоплені люди, що навіть відволікаючись від роботи над проектом, продовжують думати про роботу. Нерідко відпочинком вони вважають паралельну заміну основної діяльності, наприклад, читання профільної літератури, верстку сайтів, вивчення нових мов програмування. Однак мозок не може до безкінечності приймати виключно корисну інформацію, яку розробник прагне направляти в русло особистісного та професійного зростання [28].

Зарубіжний досвід охорони праці при використанні новітніх інформаційних технологій та сучасного комп’ютерного обладнання передбачає з метою попередження наслідків монотонної праці, підвищення рівня рухової активності і покращення розумової працездатності фахівців ІТ-індустрії під час технологічних перерв участь у спеціальних облаштованих приміщеннях необхідним спортивним інвентарем та різними тренажерами відповідних фізичних вправ, індивідуальних тренінгових завдань відповідно до віку, статі та категорії зорової роботи. Такий підхід дозволяє зняти надлишкове

психофізіологічне перевантаження, підвищити працездатність центральної нервової системи, попередити перевтому зорового аналізатора. Показана ефективність проведення різноманітних за своєю спрямованістю вправ робітників цієї галузі (приблизно на 5-30%) [28].

Зараз багато ІТ-компаній обладнують свої офіси кімнатами відпочинку та лаунж-зонами, які забезпечують психофізіологічне розвантаження працівників. Адже окремим робочим столом з ноутбуком вже давно нікого не здивуєш. Тому, бажаючи підвищити продуктивність працівників, міжнародні компанії змагаються, перетворюючи нудні одноманітні офіси в креативні простори, де нові ідеї народжуються без титанічних зусиль. Наприклад, винахідники компанії Inventionland трудяться в казкових декораціях. Тут і гігантський гоночний трек, і пряниковий будиночок, і дуже реалістичний піратський корабель, на палубі якого розташувалися комп'ютерні столи, ніжками яких служать винні бочки. Робочі місця співробітників компанії Google в Цюріху нагадують гігантські вулики, а офіс шведського інтернет-провайдера Bahnhof розташувався в бомбосховищі часів холодної війни і походить на підземний притулок землян після глобальної катастрофи. А щоб співробітників не тягнуло додому, роботодавці створюють і можливість релаксувати, не відходячи від робочого місця, обладнавши басейни, ігрові кімнати та спортзали [28].

Корпорація Google піклується і про санітарно-гігієнічні умови праці та використовує систему очищення повітря, яка видаляє з атмосфери всі токсини та тяжкі домішки [29]. Також незвичайними на території офісу є спеціальні ізолюючі світло й звук капсули, де втомлений співробітник може відпочити, так само як і в спеціалізованих кімнатах у зоні відпочинку з приглушеним світлом та заспокійливою музикою, масажних кабінетах, ігрових кімнатах. Культивації активного відпочинку та спорту в Google відводиться особлива увага, оскільки одні з найпоширеніших хвороб програмістів пов’язані зі спиною. Саме тому в головному офісі міститься спортзал, в якому можуть одночасно займатися велика частка співробітників, до того ж на території є плавальний басейн з черговим рятувальником, тому, фактично, людина може працювати прямо

звідти (пам’ятаючи про дотримання правил техніки безпеки) [30]. Для переміщення по великій території Гуглмістечка «гуглівці» (так себе називають співробітники цієї компанії) користуються самокатами, які мають моторчики. Звичайно, усі ці зручності безкоштовні.

Намагається не відставати від Google і корпорація Facebook. Так, дана фірма пропонує своїм співробітникам чудові умови праці: якісне різноманітне харчування, зручне апаратне забезпечення, важливою складовою якого є широкі монітори ПКз високою роздільною здатністю [31]. Особливістю Facebook є те, що співробітники не мають ані своїх кабінетів, ані офісних комірок – усі сидять разом, оскільки це стимулює комунікацію та обмін ідеями, підвищує продуктивність роботи. Загалом у компанії царює атмосфера позитиву. Усі співробітники привітні та посміхаються. На стінах можна побачити карикатури на топменеджерів, а робітники одягають футболки зі смішними написами [31]. Співробітники тут, на відміну від Google, пересуваються по офісу не на самокатах, а на скейтах. У Facebook також наявна велика кількість тематичних та ігрових зон, де кожен може розслабитися та відпочити, і це все також абсолютно безкоштовно.

Поява та впровадження нових інформаційно-комунікаційних технологій зумовлює необхідність подальшого вдосконалення охорони праці фахівців ІТ-індустрії. З метою належного правового забезпечення необхідно розширити ти доповнити перелік основних професій комп’ютерної галузі у національному класифікаторі [32], а також підготувати відповідний випуск у кваліфікаційному довіднику посад фахівців ІТ-індустрії, що сприятиме вирішенню питань їх соціального захисту, пенсійного забезпечення, атестації робочих місць основних професій за умовами праці на предмет подальших певних видів пільг та компенсацій за важкі шкідливі і небезпечні умови праці. Важливим напрямом стосовно визначення професійної придатності фахівців з ІТ є проведення психофізіологічної експертизи відповідно до 5 статті [33].

ІТ-фахівці, як і будь-які інші працівники, повинні проходити навчання і перевірку знань з охорони праці або в навчальному центрі, або в самій

організації. Якщо в ній є комісія з перевірки знань з охорони праці, атестованих в спеціалізованому навчальному центрі. Навчання охорони праці в організації проводять по самостійно розробленими програмами. Їх складають, спираючись на типові програми, а також з огляду на особливості галузі, в якій працює організація.

Робота з комп'ютерами нового покоління характеризується певним психофізіологічними перенавантаженнями, втомою зорового аналізатора, гіпокінезією, відсутність диференційованих норм праці при роботі з новою комп'ютерною технікою в залежності від віку, статі, категорії зорової роботи, режимів праці і відпочинку (протягом робочого дня, тижня, щорічного режиму відпусток). Все це потребує розробки нових нормативно-правових актів з регламентації праці та відпочинку фахівців ІТ-індустрії і стандартів підприємств, центрів комп'ютерної техніки, центрів інформаційних технологій, сучасних комп'ютерних класів. Особлива роль з точки зору збереження та відновлення здоров'я працюючих в комп'ютерні галузі належить попереднім та періодичним наглядам з подальшої психофізіологічної експертизи і встановленням професійної придатності при роботі з комп'ютерами нового покоління, який супроводжується виникненням певних факторів професійного ризику електротравматизму при їх ремонті та обслуговуванні. В цьому зв'язку необхідне запровадження експертизи на предмет безпечної експлуатації ПЕОМ, тобто офіційне підтвердження фактичних параметрів електробезпеки, їх відповідності вимогам нормативної документації фахівців, які проводять таку експертизу повинні пройти навчання і перевірку знань відповідно до вимог [34]. За результатами експертизи повинні прийматися рішення про відповідність ПЕОМ нормам безпеки, терміни чергової експертизи, оформлюються протоколи вимірювань і випробувань, проведені у разі потреби розрахунки та експертний висновок. Для підвищення розумової працездатності то зорової роботи повинна здійснюватися ергономічна оптимізація в рамках системи «оператор-термінал», яка сприятиме результативній фізичній та інтелектуальний працездатності і відновленню психосоматичного здоров'я фахівців ІТ-індустрії.

Заслуговує на увагу зарубіжний досвід створення у приміщеннях та в зоні їх розміщення на територіях підприємств спеціальних візуальних комфортних умов та забезпечення вимог виробничої естетики, дотримання норм рівнів виробничого шуму та акустичної тиші за межами офісу. Також дуже важливим є використання в офісних приміщеннях та кабінетах психофізіологічного розвантаження функціональної музики, яка сприяє попередженню перевтоми і підтриманню необхідного рівня розумової працездатності фахівців комп'ютерної галузі. В цьому напрямі заслуговує на увагу створення при великих центрах інформаційних технологій кімнат (кабінетів) психофізіологічного розвантаження працівників галузі (на 5 місць) [29].

Всі наведені заходи щодо вдосконалення охорони праці фахівців ІТ-індустрії повинні контролюватися службою охорони праці та комісією з охорони праці підприємства. Особливе значення у соціальному захисті цієї категорії працівників належить прийняття комплексного договору, який може забезпечити фахівців додатковими пільгами та компенсаціями.

Таким чином, використання новітніх технологій вимагає від фахівців ІТ-індустрії додержання певних правил та вимог з точки зору безпеки праці, її нормування з урахуванням віку працюючих та загального інформаційного навантаження, розробки та впровадження індивідуальних, щотижневих та щорічних режимів праці та відпочинку, які сприятимуть профілактиці перевтомлення і підвищенню розумової працездатності працюючих. Особливу роль в цьому напрямі повинні відігравати ергономічні заходи стосовно створення робочих місць, оптимізації взаємодії людини в рамках системи «оператор-термінал».

4.2 Оцінка дії електромагнітного імпульсу на елементи комп'ютерної системи

У воєнний час при застосуванні ядерної зброї проти України на електронно-обчислювальне обладнання в першу чергу буде впливати електромагнітний

імпульс (EMI) ядерного вибуху у вигляді короткого імпульсу, який вражає головним чином електричну та електронну апаратуру [35].

EMI виникають в основному в результаті взаємодії гамма-випромінювання з атомами навколишнього середовища. На утворення EMI йде невелика кількість ядерної енергії, але він здатен викликати високі імпульси струмів та напруг в кабелях повітряних і підземних ліній зв'язку, сигналізації, управління, електропередачі, в антенах радіостанцій. Вплив EMI може привести до згортання чутливих електронних та електричних елементів, зв'язаних з великими антенами чи відкритими дротами, а також до порушень в обчислювальних пристроях.

Особливістю EMI, як вражаючого фактору є його здатність розповсюджуватись на десятки і сотні кілометрів в оточуючому середовищі. Тому EMI може вплинути своєю дією на об'єкти, там де вибухова хвиля, світлове випромінювання, проникаюча радіація втрачають своє значення, як вражаючі фактори. При наземних та низьких повітряних вибухах в лініях зв'язку та електрозабезпечення виникають напруги, які можуть викликати пробій ізоляції провідників та кабелів відносно землі, пробій ізоляції елементів приладів підключених до повітряних і підземних ліній. Степінь враження залежить від наведеного імпульсу напруги чи струму і також електричної міцності обладнання.

Найбільш піддані впливу EMI системи зв'язку, сигналізації, управління. Використані в цих системах кабелі та апаратура мають обмежену електричну міцність не більше 10кВ імпульсної напруги, тоді як наведені імпульси напруги від EMI можуть перевищувати ці значення. Найбільш піддана впливу EMI апаратура виконана на напівпровідниках та інтегральних схемах, працюючих на малих струмах і напругах, і значить відчутних до впливу зовнішніх електричних і магнітних кіл, в тому числі і елементи програмного засобу для управління процесом міграції віртуальних машин в обчислювальній хмарі. EMI пробиває ізоляцію, спалює елементи електричних схем радіоапаратури, викликає коротке замикання в радіопристроях, іонізацію діелектриків, змінює або повністю стирає магнітний запис. Встановлено, що при дії EMI на апаратуру найбільша напруга

наводиться на вході. В транзисторах відбувається така залежність: чим більший коефіцієнт підсилення транзистора, тим менша його електрична міцність.

ЕМІ пошкоджує також резистори, викликає іскріння в їх міжконтактних з'єднаннях і деяких областях провідної поверхні. Найбільшу небезпеку ЕМІ представляє для апаратури, яка встановлена в особливо міцних спорудах, які витримують великі тиски ударної хвилі. В цих спорудах апаратура не виходить з ладу від механічних пошкоджень, але ЕМІ може вивести з ладу всю незахищену апаратуру системи зв'язку, сигналізації і керування. Найбільших значень досягають напруги, які наводяться між кабелем і землею.

Розглянемо можливі шляхи рішення задачі захисту від ЕМІ сервісу для адміністрування і обліку роботи автомобільної пар. Ідеальним захистом від ЕМІ виявилося б повне укриття приміщення, в якому розміщена радіоелектронна апаратура, металевим екраном. Водночас зрозуміло, що практично забезпечити такий захист у ряді випадків неможливо, тому що для роботи апаратури часто потрібно забезпечити її електричний зв'язок із зовнішніми пристроями. Тому використовуються менш надійні засоби захисту, такі, як струмопровідні сітки, або плівкові покриття для вікон, щільникові металеві конструкції для повітрезабірників і вентиляційних отворів і контактні пружинні прокладки, розміщувані по периметру дверей і люків.

Більш складною технічною проблемою рахується захист від проникнення ЕМІ в апаратуру через різноманітні кабельні входи. Радикальним рішенням даної проблеми міг би стати перехід від електричних мереж зв'язку до практично не схильних до впливу ЕМІ волоконно-оптичних. Проте заміна напівпровідникових приладів у всьому спектрі виконуваних ними функцій електронно-оптичними пристроями можлива тільки у віддаленому майбутньому. Тому в даний час в якості засобів захисту кабельних входів найбільші широко використовуються фільтри, у тому числі волоконні, а також іскрові розрядники, металлоокисні варистори та ін. [35].

Металоокисні варистори є напівпровідниковими приладами, що різко підвищують свою провідність при високій напрузі. Проте, при застосуванні цих

приладів у якості засобів захисту від EMI варто враховувати їх недостатньо високу швидкодію і погіршення характеристик при кількаразовому впливі навантажень. Ці недоліки відсутні у високошвидкісних зенеровських діодах, дія яких заснована на різкій лавиноподібній зміні опору від високого значення практично до нуля, при перевищенні прикладеної до них напруги граничного розміру. Крім того на відміну від варисторів характеристики зенеровських діодів після багатократних впливів високих напруг і переключень режимів не погіршуються.

Найбільш раціональним підходом до проектування засобів захисту від EMI кабельних входів є створення таких роз'ємів у конструкції яких передбачені спеціальні заходи, що забезпечують формування елементів фільтрів і установку вмонтованих зенеровських діодів. Подібне рішення сприяє одержанню дуже малих значень ємності й індуктивності, що необхідно для забезпечення захисту від імпульсів, що мають незначну тривалість і, отже, потужну високочастотну складову.

4.3 Висновки до четвертого розділу

В цьому розділі розглянуто важливі питання охорони праці та безпеки в надзвичайних ситуаціях, зокрема описано закордонний досвід організації охорони праці в IT-компаніях та виконано оцінку дії на елементи комп’ютерної системи.

ВИСНОВКИ

При виконанні дослідження було зроблено огляд того, які існують архітектурні підходи для реалізації веб-застосунків, після якого було обрано варіант мікросервісної архітектури. Також було проведено огляд існуючих варіантів взаємодії сервісів між собою і було розібрано принцип роботи найпопулярніших з них, які надалі використовувалися для реалізації серверної частини веб-додатку для інтернет магазину одягу. Застосунок було спроектовано та описано використовувані технології. У результаті вдалося реалізувати веб-застосунок на основі запропонованих технологій, і представити опис роботи кожного сервісу окремо, а також спосіб розгортання цілої програми. Ще було проведено невелике дослідження щодо порівняння різних способів взаємодій між сервісами. Мета дослідження полягала в тому, щоб зрозуміти, які результати будуть показувати способи взаємодії в нестандартних умовах застосування, що вдалося зробити та оформити деякі висновки з цього приводу.

Щодо висновків, який власне спосіб краще вибрати, то все залежить від конкретної ситуації та цілей, але якщо дивитися на «цифри», то явно видно, що Kafka у всіх протестованих сценаріях показує середні результати, що показує її оптимальною за швидкістю та споживанням ресурсів, тому можна зробити висновок, що з усіх наведених способів – Kafka найбільш універсальна.

У майбутньому планується доопрацювати програму, додавши більше функціоналу для кожного сервісу. Як доопрацювання також планується додати більше взаємодій для сервісів (наприклад, відправлення повідомлень користувачам, замовлення яких змінює статус).

ПЕРЕЛІК ДЖЕРЕЛ

1. Що таке мікросервісна архітектура: значення, складові, переваги. [Електронний ресурс] – Режим доступу: https://wezom.com.ua/ua/blog/scho-take-mikroservisna-arhitektura-znachennya-skladovi-perevagi (дата звертання: 02.11.2025).

2. Що краще моноліт чи мікросервіси? Як обрати архітектуру проєкту? [Електронний ресурс] – Режим доступу: https://iampm.club/ua/blog/shho-krashhe-monolit-chi-mikroservisi-yak-obrati-arhitekturu-projektu/ (дата звертання: 03.11.2025).

3. Семчишин П.М. Архітектурні рішення для розробки веб-застосунків // XIV Міжнародна науково-практична конференція молодих учених та студентів «Актуальні задачі сучасних технологій», Тернопіль, 11-12. Грудня 2025 р. с. 340-341.

4. Що таке мікросервісна архітектура: шлях до гнучкого та масштабованого середовища розробки. [Електронний ресурс] – Режим доступу: https://blog.colobridge.net/uk/2024/01/what-is-microservices-architecture-ua/ (дата звертання: 04.11.2025).

5. From Monolithic Architecture to Microservices Architecture / Lorenzo De Lauretis. – 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)

6. Архітектура мікросервісів: особливості, переваги, реальні приклади: [Електронний ресурс] – Режим доступу: https://www.hostzealot.com.ua/blog/about-solutions/arxitektura-mikroservisiv-osoblivosti-perevagi-realni-prikladi (дата звертання: 10.11.2025).

7. Bellemare A. Building Event-Driven Microservices: Leveraging Organizational Data at Scale 1st Edition. Себастопол: O'Reilly Media, Inc., 2020. 322 с.

8. Тарновецька О.Ю., Осадчук Р.Р. Дослідження методів взаємодії між сервісами при мікросервісній архітектурі програмного забезпечення. Вчені

записки ТНУ імені В.І. Вернадського. Серія: Технічні науки. Том 35 (74). № 4, 2024. С. 208 – 217. DOI https://doi.org/10.32782/2663-5941/2024.4/31

9. Що таке RPC? [Електронний ресурс] – Режим доступу: https://artjoker.ua/tech/e-commerce/rpc/https://artjoker.ua/tech/e-commerce/rpc/ (дата звертання: 14.11.2025).

10. Протокол простого доступу до об'єктів (SOAP): протокол обміну повідомленнями. [Електронний ресурс] – Режим доступу: https://cqr.company/ua/wiki/protocols/simple-object-access-protocol-soap-a-messaging-protocol/ (дата звертання: 15.11.2025).

11. Семчишин П.М. Проектування мікросервісної архітектури веб-застосунку // Інформаційні моделі, системи та технології: Праці XIII наук.-техн. конф. Тернопіль, 2025. с. 84.

12. Database Per Service Pattern for Microservices. [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/system-design/database-per-service-pattern-for-microservices/ (дата звертання: 18.11.2025).

13. Олексій Васильєв. Програмування мовою Java. Навчальна книга «Богдан», 2020. 626 с.

14. Spring. [Електронний ресурс] – Режим доступу: https://spring.io// (дата звертання: 20.11.2025).

15. PostgreSQL: The World's Most Advanced Open Source Relational Database. [Електронний ресурс] – Режим доступу: https://www.postgresql.org/ (дата звертання: 20.11.2025)

16. Фреймворк gRPC: як із ним працювати та чим він кращий за REST API. [Електронний ресурс] – Режим доступу: https://proit.ua/frieimvork-grpc-iak-iz-nim-pratsiuvati-ta-chim-vin-krashchii-za-rest-api/ (дата звертання: 20.11.2025).

17. RabbitMQ. One broker to queue them all. [Електронний ресурс] – Режим доступу: https://www.rabbitmq.com/ (дата звертання: 20.11.2025).

18. APACHE KAFKA. [Електронний ресурс] – Режим доступу: https://kafka.apache.org/ (дата звертання: 20.11.2025).

19. Cinque M., Della Corte R., Pecchia A., “Advancing Monitoring in Microservices Systems,” 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany, 2019, P. 122-123, DOI: https://doi.org/10.1109/ISSREW.2019.00060.

20. Архітектура шлюзу API та зв’язок між мікросервісами. [Електронний ресурс] – Режим доступу: https://www.hostragons.com/uk/блог/звязок-архітектури-шлюзу-api-між-мікрос/ (дата звертання: 05.12.2025).

21. Ivan Stefanyshyn, Oleh Pastukh, Volodymyr Stefanyshyn1, Ihor Baran1, and Igor Boyko. Robustness of AI algorithms for neurocomputer interfaces based on software and hardware technologies. CEUR Workshop Proceedings., 2024, 3742, pp. 137–149. https://ceur-ws.org/Vol-3742/paper10.pdf

22. Що таке Docker: [Електронний ресурс] – Режим доступу: https://www.oracle.com/cis/cloud/cloudnative/container-registry/what-is-docker/ (дата звертання: 05.12.2025).

23. Docker compose overview: [Електронний ресурс] – Режим доступу: https://docs.docker.com/compose/ (дата звертання: 05.12.2025).

24. Skorenkyy, Yu., Kozak, R., Zagorodna, N., Kramar, O., Baran, I. Use of augmented reality-enabled prototyping of cyber-physical systems for improving cyber-security education. Journal of Physics: Conference Series., 2021, 1840(1), 012026. https://iopscience.iop.org/article/10.1088/1742-6596/1840/1/012026/pdf.

25. Vyacheslav Nykytyuk, Vasyl Dozorskyy, Nataliia Kunanets, Volodymyr Pasichnyk, Oleksandr Matsiuk, Ihor Bodnarchuk: Electrical Probe-Signal Processing and Criterion for the Determination of Time Parameters of the Teeth Filling Material Polymerization Process in Dentistry. 4th IDDM 2021: Valencia, Spain. P. 54-63

26. Zagorodna, N., Skorenkyy, Y., Kunanets, N., Baran, I., Stadnyk, M. Augmented Reality Enhanced Learning Tools Development for Cybersecurity Major. CEUR Workshop Proceedings., 2022, 3309, pp. 25–32. https://ceur-ws.org/Vol-3309/short1.pdf.

27. Koroliuk, R., Nykytyuk, V., Tymoshchuk, V., Soyka, V., & Tymoshchuk, D. (2025). Automated monitoring of bee colony movement in the hive during winter season.

28. Сьогодні UA [Електронний ресурс] – Режим доступу: https://www.segodnya.ua/lifestyle/fun/pochti-kak-u-googlechemudivlyayut-ofisy-ukrainskih-it-kompaniy--764025.html (дата звертання 10.12.2025).

29. Як працюють в Google? Умови, в яких хочеться трудитися. [Електронний ресурс] – Режим доступу: http://www.clevers.com.ua/articles-cleveradvertising-agency/success-stories/245-google2 (дата звертання 10.12.2025).

30. Офіс мрії: Робота в компанії Google. [Електронний ресурс] – Режим доступу:http://bigpicture.ua/?p=187580 (дата звертання 11.12.2025).

31. Офіс Facebook: Репортаж із RMA SiliconTrip. [Електронний ресурс] – Режим доступу:https://habrahabr.ru/company/rma/blog/103800/ (дата звертання 11.12.2025).

32. Класифікатор професій ДК 003:2010. [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/rada/show/va327609-10 (дата звертання 11.12.2025).

33. Закон України «Про охорону праці». [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/laws/show/2694-12 (дата звертання 12.12.2025).

34. ДНАОП 0.00-8.20-99. Порядок проведення експертизи електроустановок споживачів/ [Електронний ресурс] – Режим доступу: https://dnaop.com/html/43255/doc-%D0%94%D0%9D%D0%90%D0%9E%D0 %9F_ 0.00-8.20-99 - (дата звертання 12.12.2025).

35. Безпека в надзвичайних ситуаціях. Методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання / укл.: Стручок В. С. Тернопіль: ФОП Паляниця В. А., 2022. 156 с.

ДОДАТКИ

ДОДАТОК А
Тези конференції

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя
Маріборский університет (Словенія)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедиміната (Литва)
Краківський економічний університет (Польща)
Вроцлавський економічний університет (Польща)
Університет «Опольська Політехніка» (Польща)
Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Вінницький національний аграрний університет
Львівський національний університет ім. І. Франка
Головне управління Пенсійного фонду в Тернопільській області
Наукове товариство ім. Шевченка
Тернопільський обласний комунальний інститут післядипломної педагогічної освіти
Сумський державний педагогічний університет
Запорізький національний університет

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

Збірник

тез доповідей

XIV Міжнародної науково-технічної конференції молодих учених та студентів

11-12 грудня 2025 року

УКРАЇНА
ТЕРНОПІЛЬ – 2025

УДК 004.415
П.М. Семчишин, студент гр. СНм-61
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

АРХІТЕКТУРНІ РІШЕННЯ ДЛЯ РОЗРОБКИ ВЕБ-ЗАСТОСУНКІВ

P.M. Semchyshyn

ARCHITECTURAL SOLUTIONS FOR WEB APPLICATION DEVELOPMENT

Коли йдеться про проектування та розробку веб-застосунків, при виборі архітектурного рішення є два основні варіанти – монолітна або мікросервісна архітектури.

Монолітна архітектура в веб-застосунках є підходом, при якому весь функціонал програми виконується в одному цілому, зазвичай у вигляді єдиного виконуваного файлу або програми [1]. Основні компоненти програми, такі як інтерфейс користувача, бізнес-логіка та доступ до даних, містяться всередині однієї програми.

Перевагами такого виду архітектури є:

– простота розгортання - оскільки всі компоненти знаходяться в одному місці, розгортання програми, зазвичай, відбувається швидше та простіше;

– простота розробки - відсутність складної інфраструктури полегшує процес розробки та налагодження програми;

– зручність масштабування на початку - при невеликому обсязі трафіку користувача монолітний застосунок може забезпечити достатню продуктивність без необхідності поділу на окремі сервіси.

Недоліки монолітної архітектури:

– складність підтримки - при збільшенні розміру програми та додаванні нових функцій може виникнути складність у підтримці та зміні коду через його єдиний монолітний характер;

– обмежена масштабованість - при досягненні межі продуктивності або необхідності масштабування окремих компонентів програми можуть виникнути труднощі через їх взаємозв'язок усередині моноліту.

– обмежена гнучкість - через єдиний характер застосування зміни в одній його частині можуть торкнутися інші компоненти, що ускладнює зміни та впровадження нових технологій.

Монолітна архітектура може бути хорошим вибором для простих застосунків з невисоким навантаженням та невеликими вимогами до масштабованості та гнучкості [1].

Мікросервісна архітектура у веб-застосунках є підходом, у якому застосунок розбивається на невеликі автономні сервіси, кожен із яких відповідає окремому функціоналу. Кожен сервіс, зазвичай, розгортається та масштабується незалежно [2].

Переваги такої архітектури:

– гнучкість та масштабованість - завдяки поділу програми на незалежні сервіси, кожен з них можна масштабувати та оновлювати незалежно від інших компонентів, що покращує гнучкість та масштабованість всієї програми;

– легкість підтримки та розробки - кожен сервіс меншого розміру та має чітко визначені межі відповідальності, що полегшує підтримку та розробку програми;

– технологічна різноманітність - різні сервіси можуть бути написані різними мовами програмування або використовувати різні технології, що дозволяє вибирати найбільш потрібні інструменти для кожного конкретного завдання.

Проте є і недоліки:

– складність конфігурації та управління - управління великою кількістю незалежних сервісів вимагає наявності складної інфраструктури та механізмів управління, що може спричинити додаткові складності та витрати;

– мережева затримка - оскільки кожен сервіс взаємодіє з іншими через мережу, виникає затримка, яка може вплинути на загальну продуктивність програми;

– складність тестування - тестування мікросервісів може бути складним через необхідність керування їх взаємодією та залежностями [2].

Таким чином, мікросервісна архітектура часто застосовується у великих та складних застосунках, де потрібна висока гнучкість, масштабованість та можливість використання різних технологій.

Якщо підбити підсумок щодо архітектурних рішень, то можна виділити те, що головний плюс мікросервісної архітектури - це велика гнучкість і масштабованість. Розбиття програми на окремі сервіси дозволяє розробляти і впроваджувати новий функціонал швидше і ефективніше, оскільки зміни в одному сервісі не торкаються інших. Це особливо важливо для сучасних веб-застосунків, які передбачають високу динамічність та швидке впровадження нових можливостей.

Використання мікросервісної архітектури також може допомогти прискорити розробку, оскільки робота над кожним сервісом ведеться незалежно і може виконуватися різними людьми/командами в різні терміни.

Кожен мікросервіс може бути масштабований незалежно від його навантаження, що забезпечує значно ефективніше застосування обчислювальних ресурсів і збільшує надійність усієї системи. Але також дана архітектура має багато нюансів, які можуть викликати проблеми. Наприклад, складність побудови інфраструктури та складність управління всім, можуть (але не обов'язково) виникнути проблеми з безпекою, так як доводиться часто передавати дані між сервісами, відповідно зростає потенційний ризик витоків інформації, ну і той факт, що дані розділені за різними процесами всієї інформації складніше.

Що ж до монолітної архітектура, то всіх цих мінусів можна уникнути, але така архітектура зручна рівно до певних розмірів застосунка, далі його стає досить проблематично підтримувати, є такий термін "Big Ball of Mud" (позначає ситуацію, коли застосунок стає надто складним і важким для підтримки та розвитку, їх складно розділити на більш дрібні та незалежні частини). Що стосується масштабованості, то там і зовсім є межа.

Способи взаємодії мікросервісів є ключовим аспектом при розробці веб-застосунків на основі мікросервісної архітектура. Якщо за монолітної архітектурі інформація може легко передаватися між різними класами всередині застосунку, то у мікросервісах її треба відправляти до інших систем. Для забезпечення ефективної роботи системи необхідно вибрати відповідні методи взаємодії, які відповідатимуть вимогам проекту та забезпечуватимуть високу продуктивність, надійність та масштабованість.

Література

1. Що краще моноліт чи мікросервіси? Як обрати архітектуру проекту? URL: https://iampm.club/ua/blog/shho-krashhe-monolit-chi-mikroservisi-yak-obrati-arhitekturu-proektu/ (дата звертання: 03.11.2025).

2. Що таке мікросервісна архітектура: шлях до гнучкого та масштабованого середовища розробки. URL: https://blog.colobridge.net/uk/2024/01/what-is-microservices-architecture-ua/ (дата звертання: 04.11.2025).

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

XIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

17-18 грудня 2025 року

ТЕРНОПІЛЬ
2025

УДК 004.415
П. Семчишин
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ПРОЕКТУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ВЕБ-ЗАСТОСУНКУ

UDC 004.415
P. Semchyshyn

DESIGNING A MICROSERVICE ARCHITECTURE FOR A WEB APPLICATION

Як розроблювана система було обрано веб-застосунок для інтернет-магазину одягу. Для його створення використовуватиметься клієнт – серверна модель, де основна частина функціоналу знаходиться на сервері. Це забезпечує високий рівень безпеки разом із захистом інформації, оскільки на сервері відбувається обробка основних операцій із даними. Також такий підхід сприяє підвищенню сумісності різних програмних продуктів та платформ, що спрощує їх інтеграцію та знижує вимоги до них.

Основні функціональні можливості програми включають авторизацію користувачів, можливість оформлення замовлень, отримання повідомлень і залишення відгуків.

Архітектура сервера буде мікросервісною, а також потрібна буде БД. Було вирішено використовувати патерн «Database per service», тобто коли в кожного сервісу своя окрема БД. Цей спосіб зручний тим, що застосунок буде мати слабку пов'язаність сервісів, що полегшує роботу з їх модифікацією, а також при потребі, кожен сервіс може використовувати різні БД (наприклад, для деяких варіантів можливо NoSql бази буде зручніше, ніж звичайний Sql), проте є і недолік даного способу – складність управління даними. Дуже складно добути інформацію, шматки якої розкидані по різних сервісах, складніше організовувати будь-які бізнес-транзакції, які охоплюють кілька сервісів.

Для доступу користувача до різних послуг буде використовуватися API Gateway. Це серверний проксі, який надає інтерфейс для клієнтів (застосунків, пристроїв, користувачів) для доступу до веб-сервісів або до мікросервісів. Він виконує функцію шлюзу, контролюючи та маршрутизуючи запити від клієнтів до відповідних служб та забезпечує централізоване управління та безпеку для всіх запитів, що проходять через веб-застосунок.

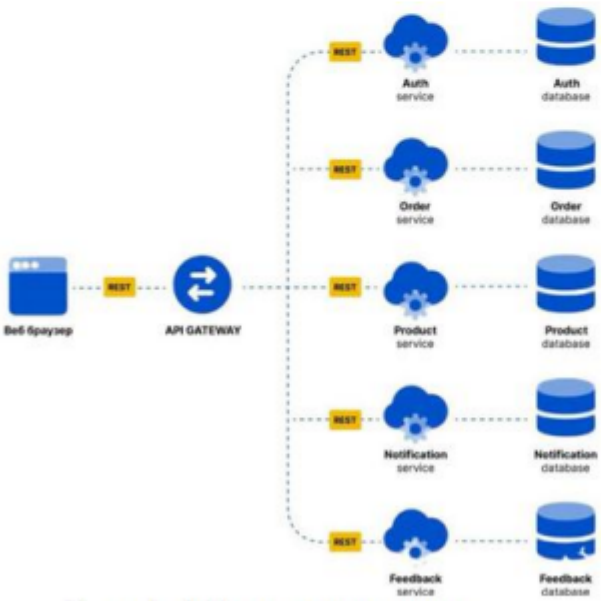

Рисунок 1 - Архітектурна схема застосунку

Схема архітектури застосунку представлена на рис. 1. У застосунка буде п'ять основних сервісів: автори-
зації; продуктів; замовлень; повідомлень; відгуків.

ДОДАТОК Б

Клас RoutesConfig.java, який відповідає за маршрутизацію запитів

```
package com.vixr.apigateway.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.gateway.route.RouteLocator;
import
org.springframework.cloud.gateway.route.builder.RouteLocatorBuilde
r;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RoutesConfig {

  @Value("${route.url.order}")
  private String orderUrl;

  @Value("${route.url.auth}")
  private String authUrl;

  @Value("${route.url.notification}")
  private String notificationUrl;

  @Value("${route.url.product}")
  private String productUrl;

  @Value("${route.url.feedback}")
  private String feedbackUrl;

  @Bean
  public RouteLocator customRouteLocator(RouteLocatorBuilder
builder) {
    return builder.routes()
        .route("auth_route", r -> r.path("/api/auth/**")
            .filters(f -> f
                .rewritePath("/api/auth/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(authUrl))

        .route("order_route", r -> r.path("/api/order/**")
            .filters(f -> f
                .rewritePath("/api/order/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(orderUrl))
```

```
        .route("notification_route", r ->
r.path("/api/notification/**")
            .filters(f -> f
                .rewritePath("/api/notification/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(notificationUrl))

        .route("product_route", r -> r.path("/api/product/**")
            .filters(f -> f
                .rewritePath("/api/product/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(productUrl))

        .route("feedback_route", r -> r.path("/api/feedback/**")
            .filters(f -> f
                .rewritePath("/api/feedback/(?<remaining>.*)",
"/${remaining}")
                .filter(new PathFilter())
            )
            .uri(feedbackUrl))
        .build();
  }

}
```

ДОДАТОК В

Класс AuthController.java

```
package com.vixr.auth.api.controller;

import com.vixr.auth.api.dto.request.InviteUserRequest;
import com.vixr.auth.api.dto.request.RefreshPasswordRequest;
import com.vixr.auth.api.dto.request.ResetPasswordRequest;
import com.vixr.auth.api.dto.request.SignInUserRequest;
import com.vixr.auth.api.dto.request.TokenRefreshRequest;
import com.vixr.auth.api.dto.response.SignedInUserResponse;
import com.vixr.auth.api.facade.AuthFacade;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import static org.springframework.http.MediaType.TEXT_HTML;

@Slf4j
@CrossOrigin
@RestController
@RequiredArgsConstructor
@RequestMapping("/v1/auth")
public class AuthController {

  private final AuthFacade facade;

  @PostMapping("/invite")
  public ResponseEntity<Void> inviteUser(@Valid @RequestBody
InviteUserRequest request) {
    log.info("Invite user: {} request", request.getEmail());
    facade.inviteUser(request);
    return ResponseEntity.noContent().build();
  }

  @PostMapping("/signin")
  public ResponseEntity<SignedInUserResponse> signInUser(@Valid
@RequestBody SignInUserRequest request) {
    SignedInUserResponse response = facade.signInUser(request);
    log.info("User [{}] signed in.", response.getUsername());
    return ResponseEntity.ok(response);
  }
```

```
  @PostMapping("/refresh-token")
  public ResponseEntity<SignedInUserResponse>
refreshAccessToken(@Valid @RequestBody TokenRefreshRequest
request) {
    log.info("Refresh token request: {}",
request.getRefreshToken());
    SignedInUserResponse response = facade.refreshTokens(request);
    return ResponseEntity.ok(response);
  }

  @PostMapping("/reset-password")
  public ResponseEntity<Void> resetPassword(@Valid @RequestBody
ResetPasswordRequest r) {
    facade.resetPassword(r);
    log.info("Password with email: [{}] requested for reset.",
r.getEmail());
    return ResponseEntity.noContent().build();
  }

  @PostMapping("/refresh-password")
  public ResponseEntity<Void> refreshPassword(@Valid @RequestBody
RefreshPasswordRequest r) {
    facade.refreshPassword(r);
    log.info("New password for user with email: [{}] was set",
r.getEmail());
    return ResponseEntity.noContent().build();
  }

  @GetMapping("/confirmation")
  public ResponseEntity<String>
confirmEmail(@RequestParam("token") String token) {
    facade.confirmEmail(token);
    log.info("Email was confirmed. token: [{}]", token);
    return
ResponseEntity.ok().contentType(TEXT_HTML).body("<h1>Now you can
close window</h1>");
  }
}
```

ДОДАТОК Г

Клас конфігурації RabbitMq

```
package com.vixr.notification.config;

import lombok.RequiredArgsConstructor;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.QueueBuilder;
import
org.springframework.amqp.rabbit.config.SimpleRabbitListenerContain
erFactory;
import
org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
org.springframework.amqp.support.converter.Jackson2JsonMessageConv
erter;
import
org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@RequiredArgsConstructor
public class RabbitMQConfig {

    @Value("${rabbitmq.notification.exchange}")
    private String exchange;

    @Value("${rabbitmq.notification.queue}")
    private String queue;

    @Value("${rabbitmq.notification.queue-dlq}")
    private String queueDlq;

    @Value("${rabbitmq.notification.routing-key}")
    private String routingKey;

    @Value("${rabbitmq.notification.routing-key-dlq}")
    private String routingKeyDlq;

    private final ConnectionFactory connectionFactory;

    private static final String X_DEAD_LETTER_EXCHANGE = "x-dead-
letter-exchange";
```

```
    private static final String X_DEAD_LETTER_ROUTING_KEY = "x-
dead-letter-routing-key";

    @Bean
    DirectExchange exchange() {
        return new DirectExchange(exchange);
    }

    @Bean
    Queue dlq() {
        return QueueBuilder.durable(queueDlq).build();
    }

    @Bean
    Queue queue() {
        return
QueueBuilder.durable(queue).withArgument(X_DEAD_LETTER_EXCHANGE,
exchange)
                .withArgument(X_DEAD_LETTER_ROUTING_KEY,
routingKeyDlq).build();
    }

    @Bean
    Binding binding() {
        return
BindingBuilder.bind(queue()).to(exchange()).with(routingKey);
    }

    @Bean
    public SimpleRabbitListenerContainerFactory
simpleRabbitListenerContainerFactory() {
        SimpleRabbitListenerContainerFactory factory = new
SimpleRabbitListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setMessageConverter(jacksonConverter());
        return factory;
    }

    @Bean
    public AmqpTemplate amqpTemplate() {
        RabbitTemplate rabbitTemplate = new
RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(jacksonConverter());
        return rabbitTemplate;
    }

    @Bean
    public MessageConverter jacksonConverter() {
        return new Jackson2JsonMessageConverter();
    }
}
```

ДОДАТОК Д

Файл docker-compose.yml для розгортання всієї програми

```
version: '3'

services:
  postgre:
    image: postgres:13-alpine
    restart: on-failure
    container_name: postgre
    volumes:
      - ./local-initdb.sql:/docker-entrypoint-initdb.d/local-
initdb.sql
    ports:
      - '5432:5432'
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: root

  rabbit:
    image: rabbitmq:3.8-management
    hostname: rabbitmq
    container_name: rabbit
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      RABBITMQ_DEFAULT_USER: admin
      RABBITMQ_DEFAULT_PASS: admin
    volumes:
      - ./rabbitmq:/var/lib/rabbitmq


  api-gateway:
    container_name: api-gateway
    depends_on:
      auth:
        condition: service_started
    build:
      context: ./api-gateway
    ports:
      - "8080:8080"
    environment:
      AUTH_HOST: auth:8080
      ORDER_HOST: order:8080
      CUSTOMER_HOST: customer:8080
      NOTIFICATION_HOST: notification:8080
      INTEGRATION_HOST: integration:8080
      PRODUCT_HOST: product:8080
```

```
auth:
  container_name: auth
  depends_on:
    postgre:
      condition: service_started
  build:
    context: ./auth
  environment:
    USER_DB_HOST: postgre
    NOTIFICATION_URL: notification

order:
  container_name: order
  depends_on:
    api-gateway:
      condition: service_started
  build:
    context: ./order
  environment:
    ORDER_DB_HOST: postgre
    NOTIFICATION_URL: notification


notification:
  container_name: notification
  depends_on:
    api-gateway:
      condition: service_started
  build:
    context: ./notification
  environment:
    NOTIFICATION_DB_HOST: postgre

product:
  container_name: product
  depends_on:
    api-gateway:
      condition: service_started
  build:
    context: ./product
  environment:
    PRODUCT_DB_HOST: postgre
    NOTIFICATION_URL: notification
```