МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Дослідження ефективності та переваг мікросервісної архітектури у Web-застосунках**

Виконав: студент 2 курсу, групи 8.1213-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
М. К. Фока
(ініціали та прізвище)
Керівник к.ф.-м.н., доцент І.А. Скрипник
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення_____
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т. В. Критська
"_02_" __вересня_2024_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Фоці Микиті Костянтиновичу
(прізвище, ім'я, по батькові)

1.Тема роботи Дослідження ефективності та переваг мікросервісної архітектури у Web - застосунках
керівник роботи Скрипник Ірина Анатоліївна, к.ф.-м.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 28.02.2024р._№_361-е

2. Строк подання студентом кваліфікаційної роботи 30.11.2024
3. Вихідні дані кваліфікаційної роботи бакалавра
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- аналіз альтернативних архітектурних підходів;
- аналіз технологій для реалізації мікросервісної архітектури ;
- аналіз SQL СУБД;
- дослідження способів комунікації між мікросервісами
- опрацювання функціональних вимог до системи;
- проєктування архітектури веб-застосунка;
- програмна реалізація застосунку та його апробація.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 02.09.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	11.09 – 15.09.24	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	16.09 – 17.09.24	виконано
3	Аналіз існуючих методів рішення	18.09 – 20.09.24	виконано
4	Огляд та збір літератури стосовно теми кваліфікаційної роботи	21.09 – 25.09.24	виконано
5	Аналіз альтернативних архітектурних підходів	26.09 – 30.09.24	виконано
6	Аналіз технологій для реалізації мікросервісної архітектури	01.10 – 05.10.24	виконано
7	Аналіз SQL СУБД	06.10 – 11.10.24	виконано
8	Дослідження способів комунікації між мікросервісами	12.10 – 15.10.24	виконано
9	Опрацювання функціональних вимог до системи	16.10 – 18.10.24	виконано
10	Проєктування архітектури веб-застосунка	19.10 – 25.10.24	виконано
11	Програмна реалізація застосунку та його апробація	26.10 – 05.11.24	виконано
12	Оформлення звіту	06.11 – 12.11.24	виконано

Студент _____ М.К. Фока
(підпис) (прізвище та ініціали)

Керівник роботи _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок – 78

Рисунків – 35

Джерел – 25

Фока М. К. Дослідження ефективності та переваг мікросервісної архітектури у Web-застосунках: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник І. А. Скрипник. Запоріжжя : ЗНУ, 2024. 78 с.

У роботі проведено ґрунтовний аналіз сучасних підходів до проєктування програмного забезпечення, зокрема мікросервісної архітектури.

Дослідження спрямоване на вивчення впливу мікросервісної архітектури на продуктивність, масштабованість, надійність і витрати на розробку та підтримку web-застосунків у порівнянні з монолітними рішеннями. У роботі висвітлено ключові принципи мікросервісного підходу, розглянуто методи організації взаємодії між сервісами, забезпечення їх незалежного розгортання та підтримки.

Було визначено переваги, недоліки та практичні обмеження цього підходу. Результати дослідження підтверджують, що мікросервісна архітектура є ефективним підходом для розробки високонавантажених web-застосунків, які потребують гнучкого масштабування і швидкої адаптації до змін вимог. Однак її впровадження вимагає високої технічної компетенції команди розробників і значних початкових витрат на проєктування.

Ключові слова*:* мікросервісна архітектура, web-застосунки, api-gateway, rest-api.

ABSTRACT

Pages – 78

Drawings – 35

Sources – 25

Foka M. K. Exploring the Effectiveness and Benefits of Microservice Architecture in Web Applications : master's thesis in specialty 121 “Software Engineering” / scientific adviser I. A. Skrypnyk. Zaporizhzhia: ZNU, 2024. 78 p.

The work provides a thorough analysis of modern approaches to software design, in particular microservice architecture.

The research is aimed at studying the impact of microservice architecture on the performance, scalability, reliability and costs of developing and maintaining web applications compared to monolithic solutions. The paper highlights the key principles of the microservice approach, discusses methods of organizing interaction between services, ensuring their independent deployment and support.

The advantages, disadvantages, and practical limitations of this approach are identified. The results of the study confirm that microservice architecture is an effective approach for developing highly loaded web applications that require flexible scaling and rapid adaptation to changing requirements. However, its implementation requires high technical competence of the development team and significant initial design costs.

Keywords*:* microservice architecture, web applications, api-gateway, rest-api.

ЗМІСТ

ВСТУП 7
1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ У WEB-ЗАСТОСУНКАХ 11
1.1 Огляд літературних джерел 11
1.2 Аналіз інших архітектурних рішень 13
1.3 Постановка завдання 17
2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ ТА ТЕХНОЛОГІЙ РЕАЛІЗАЦІЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ 19
2.1 Аналіз технології NestJS для побудови мікросервісної архітектури 19
2.2 Аналіз SQL СУБД 25
2.3 Огляд технології контейнеризації та Docker 28
2.4 Комунікація між мікросервісами: огляд технологій і підходів 31
2.5 Принципи побудови мікросервісної архітектури 37
3 РОЗРОБКА WEB-ЗАСТОСУНКА З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ 40
3.1 Опис предметної області 40
3.2 Архітектура системи 41
3.3 Функціональні вимоги системи 51
3.4 Програмна реалізація застосунку 54
3.5 Розробка інтерфейсу 65
4 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ 69
ВИСНОВКИ 75
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 76

ВСТУП

Актуальність теми

В сучасному світі інформаційних технологій, де веб-застосунок стають все складнішими та вимогливими до швидкості та масштабованості, проблема архітектурного дизайну виявляється критичною. Одним із потужних і новаторських підходів у цьому контексті є мікросервісна архітектура, яка дозволяє розділити великі системи на невеликі, незалежні компоненти, які можуть бути розвинуті, масштабовані та змінювані окремо один від одного.

Тема "Дослідження ефективності та переваг мікросервісної архітектури у веб-застосунках" набуває великої актуальності у зв'язку зі зростаючою потребою в оптимізації та підвищенні продуктивності веб-застосунків. Завдяки своїй гнучкості та здатності до горизонтального та вертикального масштабування, мікросервісна архітектура викликає значний інтерес серед розробників програмного забезпечення та бізнес-спільноти.

Однак, попри широке застосування та перспективи, пов'язані з мікросервісною архітектурою, існують виклики та питання, які вимагають подальших досліджень. Наприклад, ефективне керування мікросервісами, забезпечення безпеки та моніторингу, розв'язання проблеми консистентності даних, а також оптимізація витрат на інфраструктуру та ресурси. Тому проведення дослідження з метою оцінки ефективності та переваг мікросервісної архітектури у веб-застосунках є актуальним та важливим завданням, яке може сприяти подальшому розвитку та вдосконаленню інформаційних технологій.

Цей дослідницький проєкт спрямований на розв'язання цих проблем та відповідь на актуальні питання, що стосуються мікросервісної архітектури у веб-застосунках. Шляхом аналізу, експериментів та порівнянь будуть визначені переваги та недоліки цього підходу, а також розроблені практичні рекомендації щодо його впровадження та використання у реальних проєктів. Результати даного дослідження матимуть значний імпакт на подальший

розвиток веб-застосунків та допоможуть підвищити їхню ефективність та стійкість.

Мета і завдання дослідження

Метою даного дослідження є докладний аналіз ефективності та переваг мікросервісної архітектури у веб-застосунках з метою визначення її потенціалу та придатності для розв'язання сучасних проблем у розробці програмного забезпечення.

Об’єкт дослідження

Об’єктом дослідження є застосунок побудований на мікросервісній архітектурі.

Предмет дослідження

Предметом дослідження є визначення ефективності та переваг мікросервісної архітектури у Web-застосунках.

Методи дослідження

Проведення систематичного огляду наукових праць, публікацій, технічних звітів та інших джерел для оцінки сучасного стану досліджень у галузі мікросервісної архітектури та її застосування в веб-застосунках.

Дослідження вже чинних веб-застосунків, які використовують мікросервісну архітектуру, з метою аналізу їхньої ефективності, продуктивності та проблем, з якими вони стикаються.

Проведення експериментів з розгортанням веб-застосунків на основі мікросервісної архітектури з різними конфігураціями та параметрами для збору даних про їхню продуктивність, масштабованість та інші ключові характеристики.

Використання комп'ютерного моделювання та симуляцій для оцінки різних аспектів мікросервісної архітектури в умовах, що наближені до реальних сценаріїв веб-застосунків.

Наукова новизна одержаних результатів

Отримані результати дослідження принесли значний науковий внесок, виявивши глибокий аналіз ефективності мікросервісної архітектури порівняно з традиційними підходами, розробивши нову методологію оцінки, виявивши вплив на продуктивність веб-застосунків та надавши конкретні практичні рекомендації для використання веб-застосунків у реальних проєктів, що відкриває шлях до подальших досліджень у галузі та сприяє розвитку інформаційних технологій.

Практичне значення одержаних результатів

Отримані результати мають велике практичне значення, оскільки їхнє використання може сприяти покращенню розробки та впровадженню веб-застосунків за допомогою мікросервісної архітектури, забезпечуючи підвищену масштабованість, швидкодію та стійкість до витрат, що відповідає сучасним вимогам ринку програмного забезпечення та сприяє підвищенню конкурентоспроможності підприємств.

Апробація одержаних результатів

Результати дослідження були представлені на XVII науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2024» [24], а також на I Міжнародній науково-практичній конференції «Інженерні інновації та розбудова національної економіки» [25].

Глосарій

Мікросервісна архітектура — Стиль архітектури програмного забезпечення, де застосунок збудований з набору дрібних, незалежних служб, які спілкуються між собою через інтерфейси API.

Веб-застосунок — Програмне забезпечення, яке працює в браузері та доступне через Інтернет.

API — Інтерфейс прикладного програмування, набір правил та специфікацій, що визначають, як одна програма може взаємодіяти з іншою.

REST — Стиль архітектури API, який базується на принципах ресурсів, представлень та станів.

JSON — Формат обміну даними, який використовується для представлення даних у мікросервісах.

Kafka — Розподілений брокер повідомлень з відкритим кодом, що використовується для створення стійких до відмов, масштабованих та високопродуктивних потокових застосунків.

API Gateway — Інструмент для управління та захисту API.

Microservice Discovery — Механізм, який дозволяє мікросервісам знаходити та спілкуватися один з одним.

Мікрофронт — Невеликий, незалежний веб-компонент, який можна використовувати для створення мікросервісних веб-застосунків.

Розподілена архітектура — Стиль архітектури програмного забезпечення, де застосунок збудований з декількох компонентів, які розміщені на різних комп'ютерах.

Prometheus — Система моніторингу та збору даних, яка використовується для відстеження продуктивності мікросервісів.

1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ У WEB-ЗАСТОСУНКАХ

1.1 Огляд літературних джерел

Мікросервісна архітектура є одним із провідних підходів у сучасній розробці програмного забезпечення, який активно вивчається та впроваджується у масштабованих web-застосунках. Для ґрунтовного аналізу даного підходу проведено огляд літературних джерел, зосереджуючись на теоретичних основах, практичних кейсах та технологічних інструментах.

Мартін Фаулер і Джеймс Льюїс у своїх працях [1] стали одними з перших, хто описав мікросервісний підхід як альтернативу традиційній монолітній архітектурі. Вони визначають мікросервіси як набір невеликих автономних сервісів, що взаємодіють через чітко визначені API. Автори наголошують на таких характеристиках мікросервісів, як незалежність розгортання, стійкість до збоїв і можливість масштабування окремих компонентів (див. рис. 1).

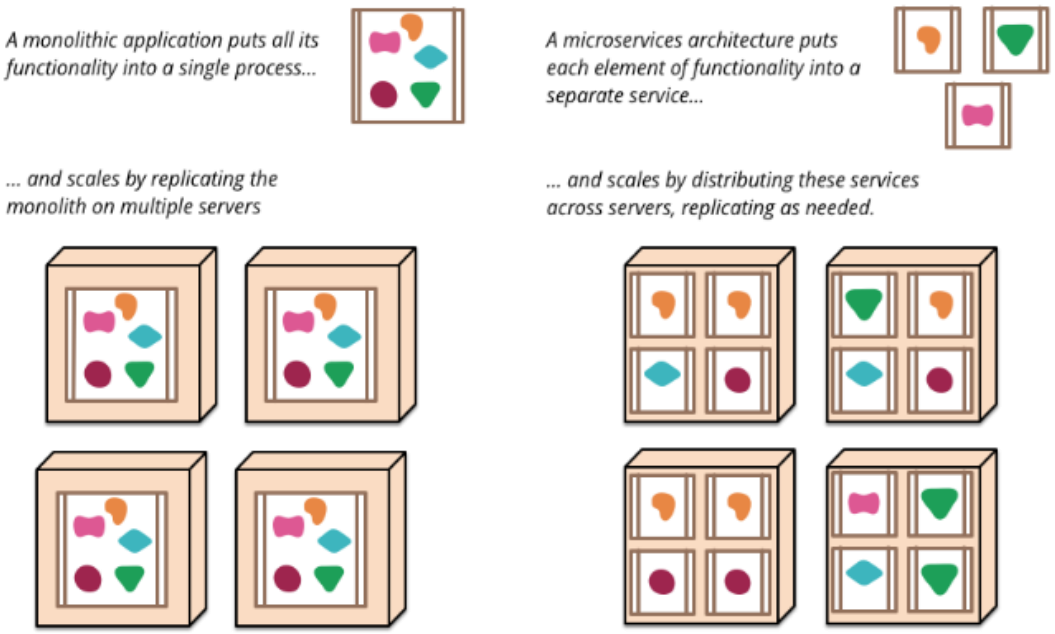


Рисунок 1 – Порівняння мікросервісної архітектури і монолітної

Додаткові теоретичні основи мікросервісного підходу викладено у книзі Сема Ньюмана "Building Microservices: Designing Fine-Grained Systems" [2]. У ній детально розглянуто питання проєктування мікросервісів, організації взаємодії між ними, а також забезпечення високої продуктивності та безпеки системи.

Практичні аспекти мікросервісної архітектури детально висвітлено у працях, присвячених реальному впровадженню цього підходу. Наприклад, книги "The Art of Scalability" [3] Ренді Шумахера та Мартина Лаббі розглядають підходи до побудови масштабованих систем із використанням мікросервісів. У праці підкреслюється важливість чіткої організації команд розробників за принципом DevOps, що забезпечує ефективність роботи з розподіленими системами.

Крім того, у книзі Брета Фішера "Docker Deep Dive" [4] обґрунтовується роль контейнеризації у впровадженні мікросервісів, зокрема використання Docker для ізоляції сервісів і Kubernetes для їхньої оркестрації. Автори пояснюють, як ці інструменти спрощують розгортання, моніторинг і масштабування мікросервісних систем.

Серед викликів мікросервісної архітектури, що детально розглядаються у сучасній літературі, виділяються питання складності взаємодії між сервісами та управління розподіленими транзакціями. Рой Філдінг у своїй праці "Architectural Styles and the Design of Network-based Software Architectures" [5] описує принципи REST, що стали основою для багатьох сучасних API.

У книзі Майкла Нігардта "Release It!" [6] аналізується вплив архітектури на стійкість системи та розглядаються способи забезпечення її надійності, такі як патерни "circuit breaker", "bulkhead" та інші (див. рис. 2).

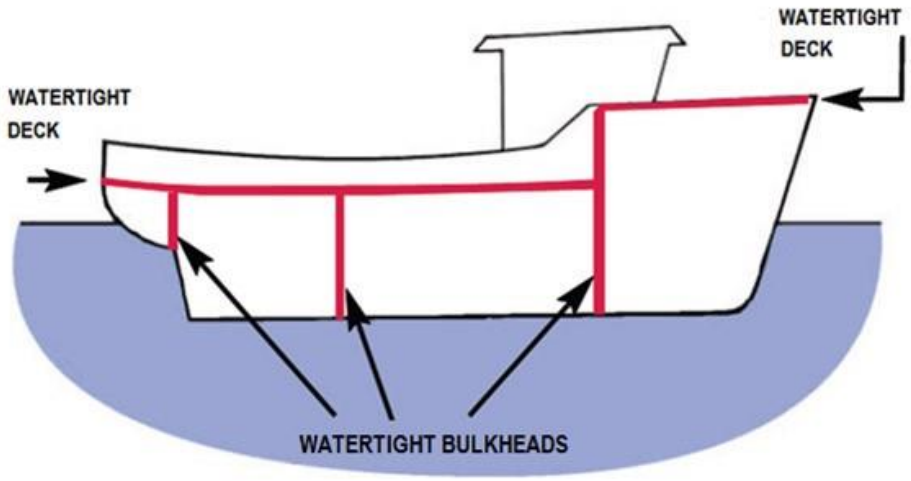


Рисунок 2 – Візуалізація патерну “bulkhead”

У сучасних дослідженнях, таких як праці Грегора Хопе та Боббі Вульфа [7], описується концепція "event-driven" мікросервісів, які базуються на асинхронній комунікації через події. Це відкриває нові перспективи для масштабування та забезпечення реального часу обробки даних у web-застосунках.

Також важливо відзначити тренд використання серверлесс-архітектур, які комбінуються з мікросервісами для зменшення витрат на інфраструктуру, як описано у дослідженнях, зібраних на платформі AWS та Google Cloud [8].

Огляд літературних джерел показує, що мікросервісна архітектура є перспективним підходом до розробки web-застосунків, але її впровадження потребує глибокого розуміння теоретичних основ, врахування практичних аспектів та усвідомлення можливих викликів. Цей розділ формує базу для подальших досліджень ефективності мікросервісної архітектури в рамках кваліфікаційної роботи [22].

1.2 Аналіз інших архітектурних рішень

У розробці web-застосунків поряд із мікросервісною архітектурою використовуються й інші підходи, кожен із яких має свої переваги та недоліки. У цьому розділі розглянуто основні архітектурні рішення, які застосовуються

у сучасному програмуванні: монолітну, серверлесс, багатошарову (n-tier) архітектури [19].

Монолітна архітектура є традиційним підходом у розробці програмного забезпечення. У ній весь функціонал системи реалізується у вигляді єдиного додатка, що запускається як одна логічна одиниця.

Схему побудови застосунку монолітної архітектури можемо побачити на рисунку 3.

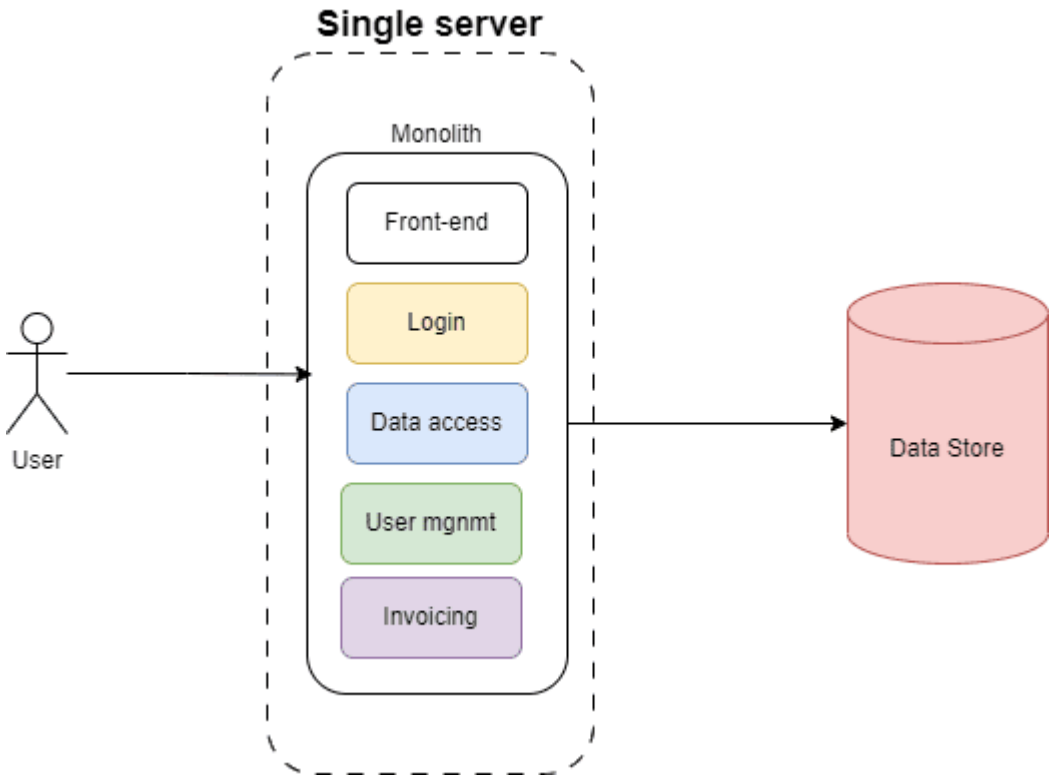


Рисунок 3 – Схема монолітної архітектури

Переваги:

- Простота розробки та налагодження завдяки єдиному коду.
- Швидке розгортання, оскільки всі компоненти інтегровані в одну структуру.
- Легкість використання для невеликих команд і проєктів із чітко визначеним функціоналом.

Недоліки:

- Складність масштабування. Потрібно масштабувати весь додаток, навіть якщо проблема стосується лише одного модуля.
- Збільшення складності підтримки зі зростанням системи.
- Низька стійкість до збоїв. Вихід із ладу однієї частини може зупинити всю систему.

Монолітна архітектура доцільна на ранніх етапах розробки, проте для складних і масштабованих систем її обмеження стають критичними.

Багатошарова архітектура (часто реалізується у вигляді трирівневої структури: клієнт, сервер, база даних) є проміжним рішенням між монолітом і мікросервісами.

Схему побудови застосунку багатошарової архітектури бачимо на рисунку 4.

Переваги:

- Чіткий поділ відповідальностей між рівнями забезпечує кращу структуризацію.
- Легкість в обслуговуванні кожного шару окремо.
- Підходить для класичних систем, що взаємодіють із великими базами даних.

Недоліки:

- Залежність між рівнями може викликати складності у масштабуванні.
- Високі затримки через необхідність передачі даних між рівнями.
- Низька гнучкість у порівнянні з мікросервісами.

Багатошарова архітектура добре підходить для корпоративних додатків із фіксованою бізнес-логікою, однак вона поступається мікросервісам у динамічності та масштабованості.

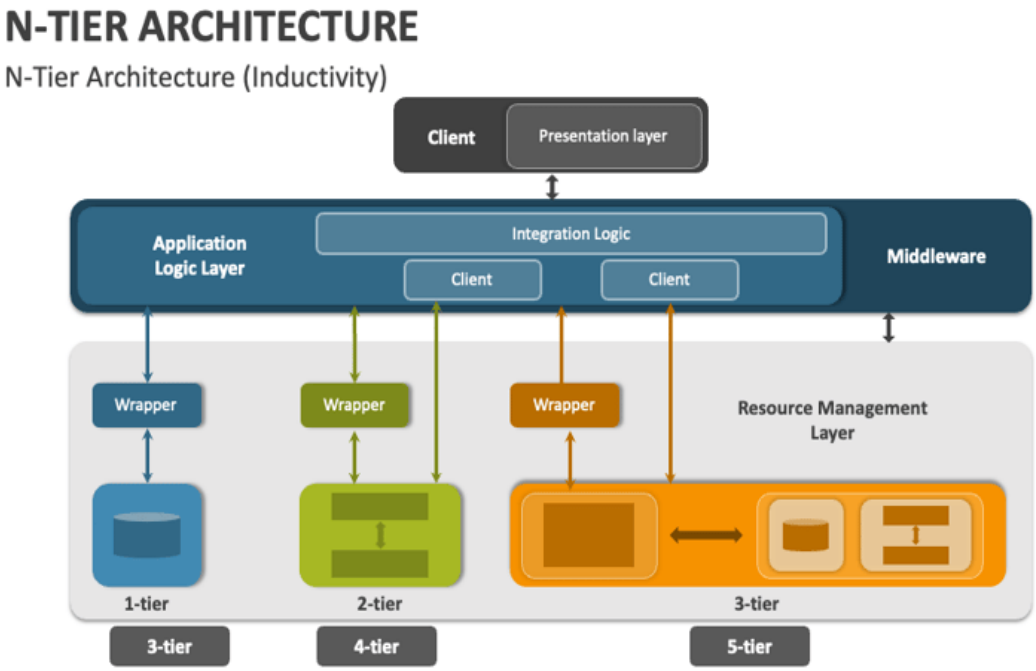


Рисунок 4 – Схема багатошарової архітектури

Серверлесс (serverless) архітектура, що базується на використанні функцій як сервісів (FaaS), є сучасним трендом. Цей підхід передбачає, що розробник відповідає лише за код, а всі питання інфраструктури вирішуються провайдером хмарних послуг (AWS, Azure, Google Cloud). Схему побудови Серверлесс архітектури на основі AWS можемо побачити на рисунку 5.

Переваги:

- Мінімальні витрати на інфраструктуру.
- Миттєве масштабування залежно від навантаження.
- Оплата лише за фактичне використання ресурсів.

Недоліки:

- Залежність від провайдерів хмарних послуг.
- Складність реалізації тривалих транзакцій.
- Високі затримки для холодного старту функцій.

Серверлесс підходить для застосунків із непостійним навантаженням, де критично важливим є швидкий запуск нових функцій.

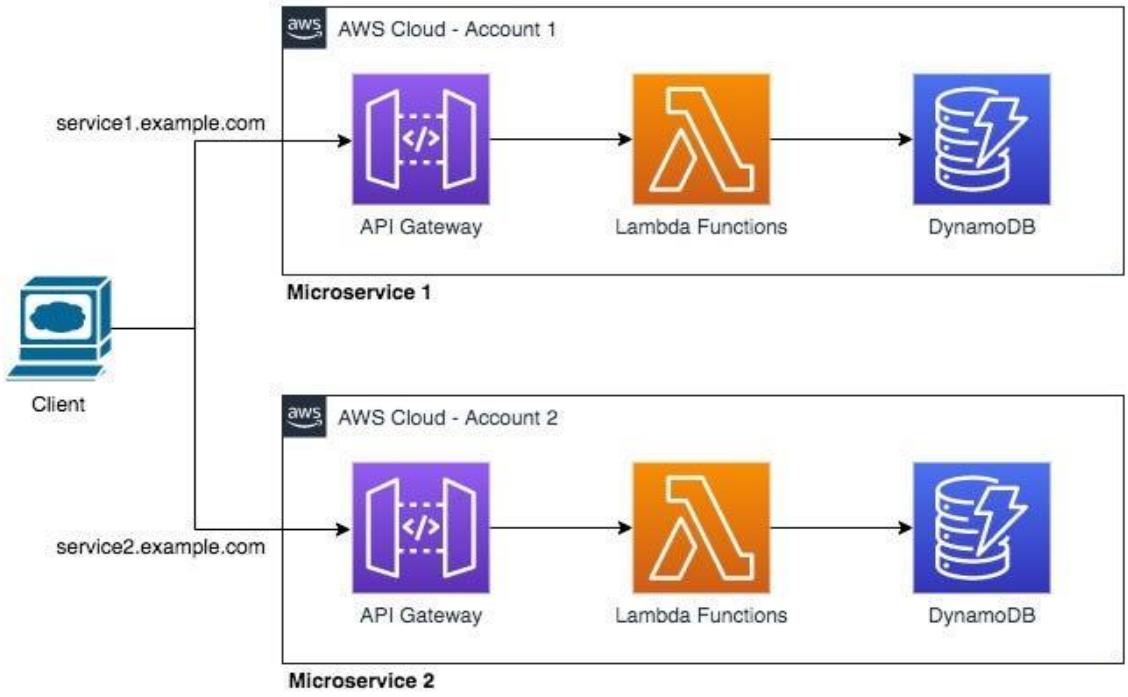


Рисунок 5 – Схема побудови Серверлесс архітектури на основі AWS

Розглянуті архітектурні підходи демонструють, що вибір конкретного рішення залежить від специфіки проєкту, бізнес-вимог та обмежень. Монолітна архітектура залишається актуальною для невеликих систем із передбачуваним навантаженням, у той час, як багатошарова структура підходить для класичних корпоративних рішень. Серверлесс архітектура є новим перспективним підходом, але він має свої обмеження. У цьому контексті мікросервісна архітектура виділяється як найбільш універсальне рішення для розробки масштабованих і динамічних web-застосунків, яке буде досліджено у наступних розділах.

1.3 Постановка завдання

Метою кваліфікаційної роботи є дослідження ефективності та переваг мікросервісної архітектури для розробки web-застосунків, а також розробка прототипу системи на основі мікросервісів. Основна задача полягає у

створенні масштабованого, стійкого до збоїв та легко модернізованого програмного продукту.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Дослідити мікросервісну архітектуру, її особливості, переваги та недоліки.
2. Порівняти її з альтернативними архітектурними підходами (моноліт, багатошарова архітектура, серверлесс).
3. Ознайомитися з особливостями використання фреймворку NestJS для реалізації мікросервісів.
4. Вивчити можливості контейнеризації з використанням Docker і оркестрації за допомогою Kubernetes.
5. Проаналізувати роботу з базою даних PostgreSQL.
6. Розробити архітектуру системи з поділом на сервіси, враховуючи принципи розподілу відповідальності.
7. Визначити спосіб комунікації між сервісами (REST API або асинхронна взаємодія через брокер повідомлень).
8. Реалізувати бекенд-сервіси за допомогою NestJS.
9. Налаштувати зберігання даних у базі PostgreSQL.
10. Розробити фронтенд-додаток з використанням React для демонстрації взаємодії з бекендом.
11. Описати процес розробки, використані інструменти та технології.
12. Підготувати рекомендації щодо вибору архітектурного підходу залежно від вимог проєкту.

Результатом роботи стане web-застосунок, реалізований на основі мікросервісної архітектури, а також аналіз ефективності цього підходу у порівнянні з іншими архітектурними рішеннями.

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ ТА ТЕХНОЛОГІЙ РЕАЛІЗАЦІЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

У цьому розділі проведено аналіз програмних засобів і технологій, використаних для реалізації мікросервісної архітектури у web-застосунку, зокрема фреймворку NestJS, системи управління базами даних PostgreSQL, бібліотеки React для клієнтської частини та технології контейнеризації за допомогою Docker.

2.1 Аналіз технології NestJS для побудови мікросервісної архітектури

NestJS – це сучасний Node.js фреймворк, побудований з акцентом на модульність, масштабованість і продуктивність. Він поєднує концепції програмування на основі TypeScript з об'єктно-орієнтованим, функціональним і реактивним підходами, що робить його універсальним інструментом для розробки серверних застосунків. Завдяки своїй архітектурі NestJS є особливо ефективним у створенні мікросервісів, які потребують чіткої організації, легкого масштабування та високої продуктивності [21].

NestJS вирізняється модульною архітектурою, яка дозволяє будувати масштабовані, добре структуровані та легкі у підтримці серверні додатки. Основна ідея цієї архітектури полягає у створенні застосунків із чітко розподіленими функціональними блоками, які називаються модулями. Завдяки цьому забезпечується висока ізоляція логіки, що полегшує розробку, тестування та подальше розширення.

Кожна частина додатка, створеного на основі NestJS, є частиною модуля. Модулі виступають своєрідними контейнерами, які групують взаємопов'язані компоненти, такі як контролери, сервіси, провайдери та інтерфейси. Це сприяє організації коду за принципами розподілу відповідальності. Схема модуля NestJS представлена на рисунку 6.

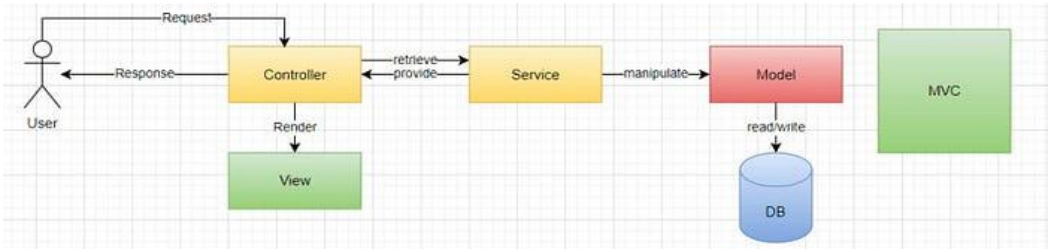


Рисунок 6 – Схема NestJS модуля

Ключовим елементом архітектури NestJS є контролери, які відповідають за обробку вхідних запитів від клієнтів. Вони отримують дані, передають їх у відповідні сервіси для обробки, а потім повертають клієнту результат. Контролери забезпечують чітке розділення логіки обробки запитів і бізнес-логіки, яка реалізується в сервісах.

Сервіси в NestJS є основою для реалізації бізнес-логіки. Вони відповідають за виконання операцій із даними, взаємодію з базами даних, зовнішніми API чи іншими сервісами. Завдяки використанню механізму впровадження залежностей (Dependency Injection, DI), сервіси можуть бути легко перевикористані в різних частинах застосунку. Це також сприяє спрощенню тестування, оскільки залежності можна замінити на мок-об'єкти.

Однією з найважливіших архітектурних особливостей NestJS є декоратори. Це спеціальні функції, які додають метадані до компонентів. Декоратори використовуються для визначення контролерів, маршрутів, провайдерів та їх залежностей. Наприклад, декоратори @Controller(), @Get(), @Injectable() або @Module() дозволяють описувати логіку роботи компонентів у декларативному стилі, що полегшує розуміння коду.

Ще однією ключовою особливістю є Middleware та Guards. Middleware у NestJS дозволяють перехоплювати та обробляти запити перед тим, як вони досягають контролерів. Це зручно для реалізації таких функцій, як логування, обробка помилок чи автентифікація. Guards, своєю чергою, дозволяють виконувати перевірку доступу до ресурсів, надаючи потужний інструмент для налаштування авторизації.

NestJS підтримує створення мікросервісів за допомогою вбудованого мікросервісного модуля. Цей модуль забезпечує чітке визначення взаємодії між сервісами через різні протоколи, такі як HTTP, WebSocket, gRPC або асинхронні повідомлення через брокери, такі як RabbitMQ чи Kafka. Завдяки цьому взаємодія між сервісами стає легкою та гнучкою. Схема побудови мікросервісної архітектури за допомогою NestJS представлена на рисунку 7.

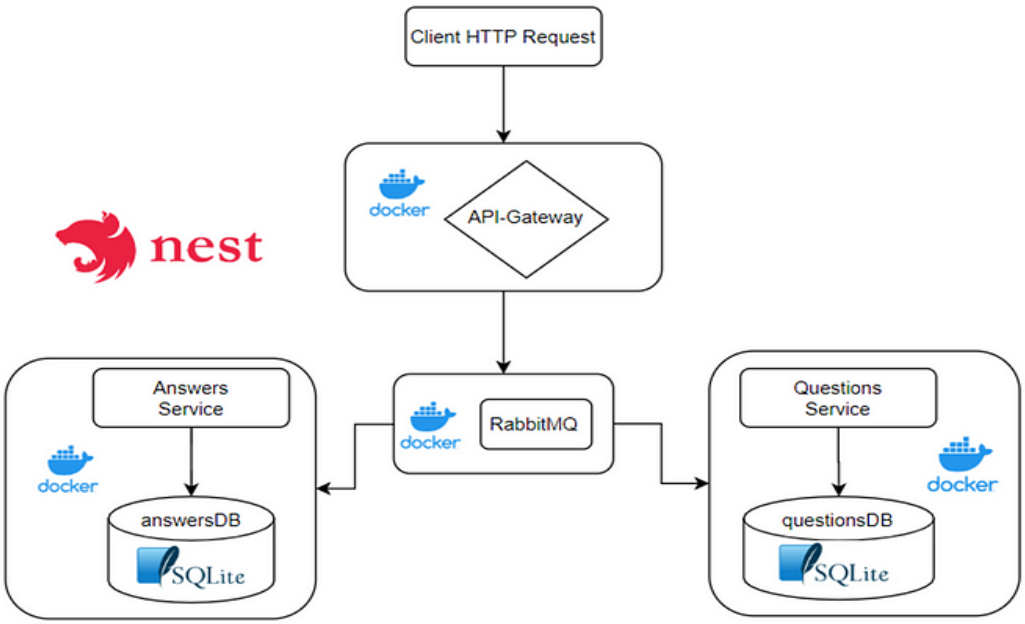


Рисунок 7 – Приклад мікросервісної архітектури розробленої за допомогою NestJS

Важливим елементом архітектури NestJS є інтеграція із зовнішніми системами. Це досягається завдяки провайдерам, які виступають посередниками між внутрішніми сервісами та зовнішніми джерелами даних, такими як бази даних, сторонні API чи інші сервіси. Використовуючи такі ORM, як TypeORM чи Prisma, NestJS дозволяє легко працювати з базами даних різних типів.

Архітектурний підхід NestJS заснований на принципах масштабованості, модульності та повторного використання коду. Він дозволяє будувати складні системи з чіткою структурою, що робить його ідеальним

вибором для створення застосунків будь-якого рівня складності, включаючи мікросервісні системи.

Переваги NestJS для створення мікросервесних застосунків:

1. Модульність і розподіл відповідальності.

NestJS побудований за принципом модульності, що дозволяє розбивати систему на незалежні функціональні блоки.

- Чітка структура. Кожен модуль виконує конкретну роль, наприклад, обробка користувачів, замовлень або платежів, що дозволяє організувати код таким чином, щоб уникати надмірних залежностей між частинами.
- Легка підтримка та розширення. Зміни або додавання нових функцій у систему можна робити, змінюючи або додаючи лише конкретні модулі, не зачіпаючи решту додатка.
- Повторне використання. Модулі можуть бути легко перевикористані в інших проєктах або частинах системи.

2. Підтримка різних способів взаємодії між мікросервісами.

NestJS забезпечує гнучкість у виборі способу комунікації між мікросервісами:

- HTTP і REST API. Забезпечує синхронну взаємодію між сервісами через добре структуровані REST-запити.
- WebSocket. Підходить для побудови систем реального часу, таких як чати або онлайн-моніторинг.
- gRPC. Забезпечує високу продуктивність та ефективність шляхом бінарної комунікації між сервісами.
- Message Brokers. Підтримка брокерів повідомлень (RabbitMQ, Kafka) дозволяє створювати асинхронні системи з підвищеною надійністю.
- Мікросервісний модуль. Вбудований у NestJS модуль дозволяє легко організувати ці способи взаємодії з використанням стандартних підходів.

3. Продуктивність та масштабованість.

NestJS використовує Node.js і його асинхронну модель на основі подій, що забезпечує:

- Високу продуктивність. Асинхронна обробка запитів дозволяє ефективно обробляти велику кількість одночасних підключень.
- Горизонтальне масштабування. Мікросервіси можна масштабувати незалежно, додаючи нові інстанси окремих сервісів відповідно до навантаження.

4. Можливість інтеграції з брокерами повідомлень.

NestJS підтримує популярні брокери повідомлень, такі як RabbitMQ, Kafka, MQTT.

- Асинхронна комунікація. Це особливо важливо для систем, де необхідна висока надійність обміну повідомленнями між сервісами.
- Надійність і відновлення. Повідомлення зберігаються в чергах до моменту їх успішної обробки, що знижує ризик втрати даних у разі збою.

5. TypeScript і типізація.

NestJS побудований на основі TypeScript, що забезпечує:

- Статичну типізацію. Зменшує кількість помилок на етапі розробки завдяки перевірці типів.
- Простоту рефакторингу. Завдяки чітким типам і інтерфейсам зміни в коді стають простішими та безпечнішими.
- Покращену читабельність. Коди легко розуміти навіть у великих проєктах.

6. Впровадження залежностей (Dependency Injection, DI).

NestJS має потужний механізм DI, який:

- Сприяє модульності. Забезпечує ізоляцію компонентів і дозволяє легко замінювати залежності в процесі розробки або тестування.
- Покращує тестування. Легко створювати мок-об'єкти для перевірки компонентів у відриві від їх реальних залежностей.

7. Уніфікований підхід до розробки.

NestJS забезпечує стандартизований стиль розробки:

- Декоратори. Використання декораторів для опису контролерів, маршрутів, сервісів і залежностей дозволяє дотримуватися єдиного підходу у всьому коді.
- Чітка структура файлів. Визначені шляхи організації коду спрощують співпрацю розробників у командах.

8. Можливості тестування.

NestJS підтримує вбудовані інструменти для модульного, інтеграційного та e2e-тестування:

- Ізоляція модулів. Кожен модуль можна тестувати окремо, з легким мокінгом залежностей.
- Швидкий старт. Підтримка тестових середовищ і інтеграція з популярними бібліотеками, такими як Jest.

9. Екосистема та інтеграція із сучасними технологіями.

NestJS легко інтегрується з популярними інструментами та платформами:

- Підтримка ORM. TypeORM, Prisma, Sequelize забезпечують легку роботу з базами даних.
- Контейнеризація. NestJS легко запускається в Docker-контейнерах, що полегшує розгортання та масштабування.
- Kubernetes. Підтримує оркестрацію мікросервісів для забезпечення їх стійкості та розподілу навантаження.

10. Безпека.

NestJS має вбудовані механізми для забезпечення безпеки додатків:

- Аутентифікація та авторизація. Легка інтеграція з JWT, OAuth2 та іншими стандартами.
- Захист від атак. Можливість налаштування middleware для захисту від CSRF, XSS та інших вразливостей.

11. Гнучкість і незалежність сервісів.

NestJS дозволяє створювати мікросервіси, які функціонують незалежно один від одного. Це забезпечує:

- Локалізовані оновлення. Внесення змін в один сервіс не впливає на решту системи.
- Розподіл навантаження. Кожен сервіс можна масштабувати відповідно до його вимог.

NestJS пропонує багатий набір інструментів і функцій для створення сучасних мікросервісних додатків. Завдяки модульності, масштабованості, підтримці різних протоколів комунікації, високій продуктивності та безпеці, цей фреймворк стає незамінним вибором для розробки складних систем, що відповідають вимогам сучасного програмного забезпечення.

2.2 Аналіз SQL СУБД

У сучасній розробці програмного забезпечення SQL-системи управління базами даних (СУБД) є одним із ключових елементів для забезпечення збереження, доступу та обробки даних. У цьому проєкті для реалізації мікросервісної архітектури була обрана PostgreSQL, одна з провідних реляційних SQL СУБД. Вона відома своєю надійністю, продуктивністю, розширюваністю та підтримкою сучасних стандартів SQL [18].

PostgreSQL є оптимальним вибором для мікросервісної архітектури завдяки своїй модульності, гнучкості й підтримці широкого спектра функціональностей, які необхідні для розробки складних програмних рішень.

PostgreSQL базується на клієнт-серверній архітектурі. Серверна частина відповідає за виконання запитів, управління транзакціями, обробку з'єднань, а також управління фізичним і логічним зберіганням даних.

Можна виділити наступні характеристики архітектури PostgreSQL:

1. Модульність. PostgreSQL побудована як модульна система, що дозволяє налаштовувати її залежно від потреб додатка. Модулі можна додавати для покращення продуктивності, розширення функціональності чи забезпечення інтеграції з іншими системами.

2. Рівні зберігання. Архітектура PostgreSQL включає фізичний рівень для зберігання даних, логічний рівень для представлення даних у вигляді таблиць і схем, а також рівень обробки запитів для аналізу та оптимізації SQL-запитів.

3. Підтримка одночасності. Завдяки MVCC (Multiversion Concurrency Control) PostgreSQL забезпечує можливість одночасної роботи багатьох користувачів без блокування бази даних.

Серед функціональних можливостей PostgreSQL можемо зазначити наступні.

1. Повна підтримка стандартів SQL. PostgreSQL підтримує останні версії стандартів SQL, включаючи складні конструкції, такі як вбудовані підзапити, тригери, вбудовані функції й агрегати.

2. Розширені типи даних. Окрім стандартних типів даних (INTEGER, VARCHAR, DATE), PostgreSQL надає підтримку JSON і JSONB для роботи зі структурованими й напівструктурованими даними, масивів для зберігання множинних значень у межах одного поля, геопросторових даних через розширення PostGIS, настроюваних типів даних, які розробники можуть визначати відповідно до специфіки свого проєкту.

3. Механізми реплікації та масштабування. PostgreSQL підтримує як синхронну, так і асинхронну реплікацію, що дозволяє забезпечити високу доступність і горизонтальне масштабування. Це особливо важливо для мікросервісів, де окремі сервіси можуть мати різні вимоги до баз даних.

4. Підтримка розширень. PostgreSQL дозволяє додавати розширення, які розширюють її функціональність. Серед таких розширень зазначимо PostGIS для геопросторового аналізу, pg_partman для автоматичного порціонування таблиць, Citus для масштабування даних на декілька серверів.

5. Безпека та управління доступом. PostgreSQL забезпечує високий рівень безпеки завдяки підтримці ролевої моделі доступу, SSL-шифрування з'єднань, інтеграції з LDAP та іншими системами автентифікації.

6. Гнучка підтримка транзакцій. PostgreSQL реалізує принципи ACID (Atomicity, Consistency, Isolation, Durability), забезпечуючи надійність

збереження даних та ізоляцію транзакцій для запобігання конфліктів між одночасними запитами.

7. Інтеграція з ORM-фреймворками. PostgreSQL підтримує популярні ORM-фреймворки, такі як TypeORM, Prisma, Sequelize. Це дозволяє швидко інтегрувати її в сучасні фреймворки для серверних додатків, наприклад, NestJS.

Переваги використання PostgreSQL для сучасних додатків, особливо у контексті мікросервісної архітектури, є значними завдяки її гнучкості, продуктивності та функціональності. PostgreSQL відома своєю здатністю ефективно обробляти великі обсяги даних і забезпечувати надійність роботи в умовах інтенсивних навантажень.

Однією з ключових переваг PostgreSQL є її висока продуктивність, досягнута завдяки розширеним механізмам індексування. Індекси типів B-Tree, GiST, GIN і BRIN дозволяють оптимізувати виконання запитів навіть у випадках, коли обсяг даних перевищує терабайти. Завдяки цьому PostgreSQL забезпечує швидкий доступ до даних, мінімізуючи час обробки запитів.

Не менш важливою є масштабованість PostgreSQL, яка робить цю СУБД придатною для використання у великих розподілених системах. Вона підтримує як горизонтальне масштабування за допомогою реплікації й шардінгу, так і вертикальне шляхом ефективного використання ресурсів сервера. Реплікація забезпечує можливість створення кількох копій бази даних, що дозволяє розподіляти навантаження між різними вузлами та підвищувати доступність системи.

Гнучкість PostgreSQL проявляється в її здатності працювати з різними типами даних. Окрім стандартних реляційних типів, вона підтримує JSON і JSONB для роботи з напівструктурованими даними, масиви, які дозволяють зберігати множинні значення в одному полі, а також спеціалізовані розширення, такі як PostGIS для обробки геопросторових даних. Це робить PostgreSQL універсальним інструментом для вирішення широкого спектра завдань, від обробки транзакцій до аналітики великих даних.

PostgreSQL також вирізняється надійністю та безпекою. Завдяки повній відповідності принципам ACID (атомарність, узгодженість, ізоляція, довговічність), ця СУБД забезпечує високу цілісність даних навіть у разі збою. Крім того, розвинена модель управління доступом, яка включає рольову систему, підтримку SSL-шифрування і можливість інтеграції з системами автентифікації, такими як LDAP або Kerberos, гарантує безпеку даних у сучасних вебдодатках.

Ще однією важливою перевагою PostgreSQL є її відкритий код і активна спільнота розробників. Це означає не лише відсутність ліцензійних витрат, але й постійне оновлення та вдосконалення СУБД. Велика кількість документації, розширень та готових рішень роблять PostgreSQL доступною для широкого кола розробників.

Завдяки своїй розширюваності PostgreSQL може адаптуватися до унікальних потреб конкретного проєкту. Можливість додавання власних функцій і розширень дозволяє налаштовувати систему для вирішення специфічних завдань. У поєднанні з інтеграцією популярних ORM-фреймворків, таких як TypeORM або Prisma, PostgreSQL значно спрощує розробку серверних додатків.

Усе це робить PostgreSQL чудовим вибором для створення стабільних, продуктивних і масштабованих систем. У контексті мікросервісної архітектури вона дозволяє ефективно управляти даними окремих сервісів, підтримуючи їх незалежність і ізольованість, що є критично важливим для досягнення цілей проєкту.

2.3 Огляд технології контейнеризації та Docker

Технологія контейнеризації є ключовим компонентом сучасної розробки програмного забезпечення, особливо у випадках, коли йдеться про мікросервісну архітектуру. Контейнеризація дозволяє створювати ізольовані середовища для запуску додатків, що забезпечує незалежність кожного

компоненту системи та їхню стабільну роботу незалежно від оточення. Одним із найпоширеніших інструментів для реалізації контейнеризації є платформа Docker, яка завдяки своїй простоті, гнучкості та багатому функціоналу здобула широку популярність серед розробників [16].

Контейнеризація розв'язує проблему узгодженості середовищ розробки, тестування та продакшну. Раніше розробники часто натрапляли на труднощі через різницю в конфігураціях серверів, бібліотек або залежностей. Завдяки контейнерам усе програмне забезпечення та необхідні для його роботи компоненти пакуються у стандартизовані образи, які можна запускати на будь-якому хості з встановленим Docker. Це гарантує ідентичну роботу додатка в усіх середовищах, що суттєво спрощує процес розгортання.

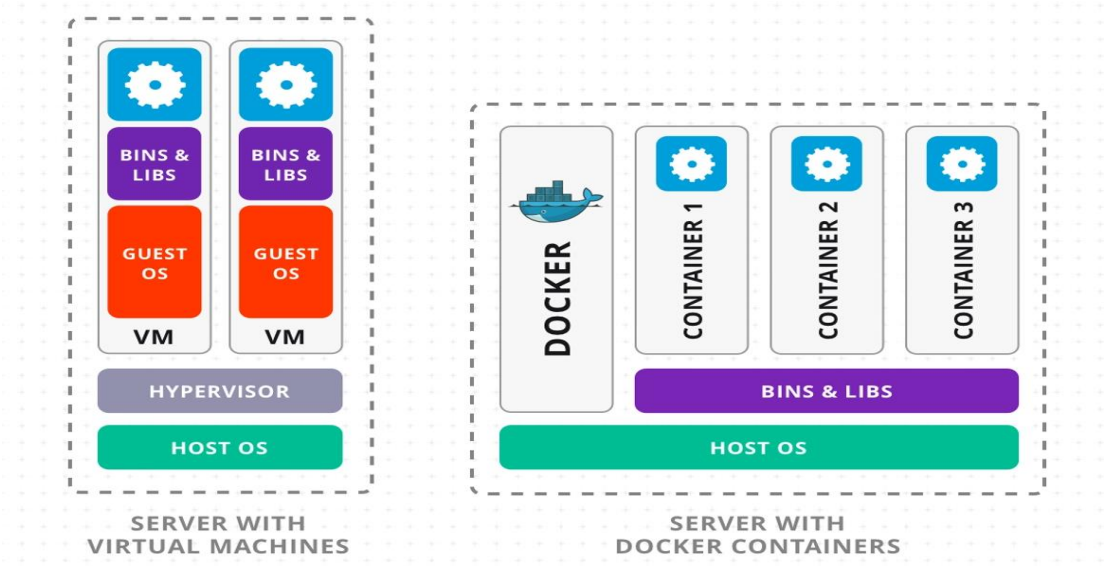


Рисунок 8 – Порівняння віртуальної машини й Docker

Docker є платформою, яка забезпечує створення, управління та запуск контейнерів. На відміну від віртуальних машин, які використовують гіпервізори для емуляції апаратного забезпечення, Docker працює на основі ізоляції процесів і використання ядерної віртуалізації, на рисунку 8 можемо побачити порівняння віртуальної машини й Docker. Це дозволяє контейнерам бути легшими, споживати менше ресурсів і запускатися за лічені секунди. Контейнери Docker базуються на образах, що являють собою попередньо

налаштовані середовища з операційною системою, бібліотеками, додатками та всіма необхідними залежностями.

Схему роботи Docker представлено на рисунку 9.

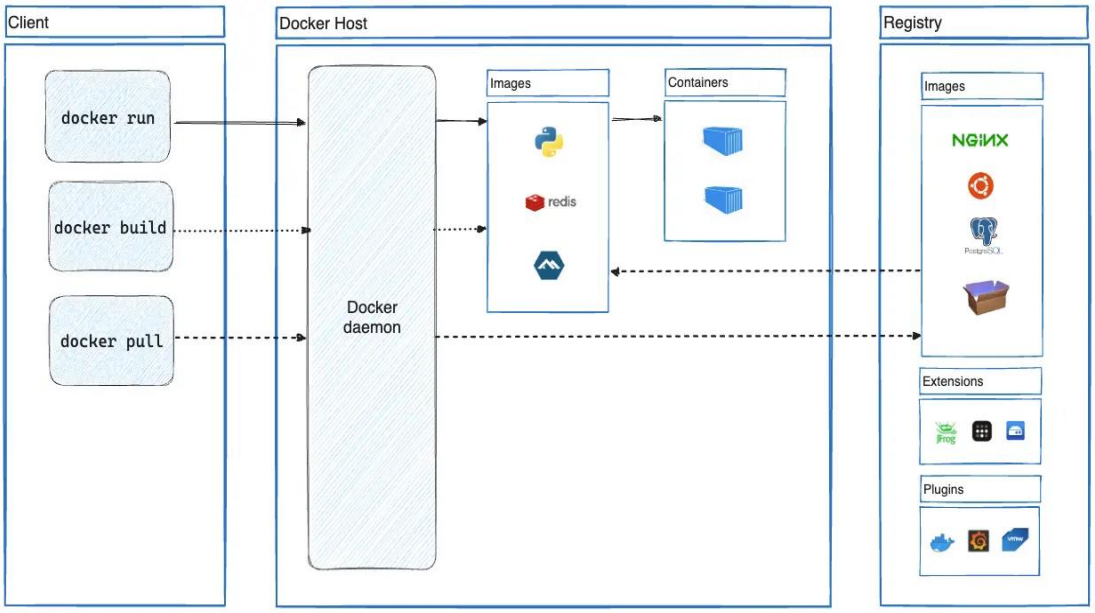


Рисунок 9 – Схема роботи Docker

Один із ключових елементів роботи Docker — це Dockerfile, текстовий файл із набором інструкцій, який визначає, як створити образ. У Dockerfile розробник може описати встановлення операційної системи, бібліотек, конфігурацій і навіть запуск додатка. Після створення образу його можна зберігати в локальному або публічному репозиторії, наприклад, Docker Hub, і повторно використовувати для запуску додатків на інших серверах.

Завдяки контейнеризації мікросервісна архітектура отримує важливі переваги. Кожен мікросервіс можна розміщувати в окремому контейнері разом із його залежностями, що забезпечує повну ізоляцію між компонентами системи. Наприклад, різні сервіси можуть використовувати різні версії одних і тих самих бібліотек або фреймворків, не викликаючи конфліктів. Крім того, контейнеризація дозволяє швидко масштабувати окремі компоненти залежно від їхнього навантаження, запускаючи додаткові екземпляри контейнерів.

Однією з особливостей Docker є його інтеграція з оркестраторами, такими як Kubernetes або Docker Swarm. Оркестратори дозволяють автоматизувати управління контейнерами в розподілених системах, забезпечуючи балансування навантаження, відмовостійкість і автоматичне масштабування. Це робить Docker ідеальним інструментом для побудови хмарних систем і мікросервісів, де важливо підтримувати високу доступність і стабільність роботи додатків.

Безпека є важливою складовою Docker. Контейнери ізольовані один від одного та від хостової системи, що зменшує ризики, пов’язані з вразливостями додатків. Крім того, Docker підтримує механізми шифрування, контроль доступу до образів і інтеграцію з іншими інструментами кібербезпеки.

Важливо також зазначити роль контейнеризації в процесах CI/CD (безперервної інтеграції та доставки). Використовуючи Docker, розробники можуть створювати стандартизовані середовища для тестування, що гарантує передбачуваність і стабільність роботи програмного забезпечення під час розгортання нових версій. Це значно знижує ризики помилок через різницю середовищ у циклі розробки.

Таким чином, Docker є одним із найефективніших інструментів контейнеризації, який суттєво полегшує розробку, розгортання та управління додатками, особливо в контексті мікросервісної архітектури. Його використання дозволяє досягти високої стабільності роботи додатків, прискорити процеси розробки та забезпечити зручність масштабування системи. Усе це робить Docker незамінним компонентом сучасної розробки програмного забезпечення.

2.4 Комунікація між мікросервісами: огляд технологій і підходів

У мікросервісній архітектурі одним із ключових викликів є забезпечення ефективної, надійної та масштабованої комунікації між сервісами. Оскільки кожен мікросервіс є незалежним модулем із власними функціональними

завданнями та часто розгортається окремо, необхідно впроваджувати підходи, які забезпечують як синхронну, так і асинхронну взаємодію. Вибір технологій для зв’язку між мікросервісами залежить від специфіки бізнес-завдань, вимог до продуктивності та відмовостійкості, а також загальної архітектури системи [15].

Для синхронної взаємодії найчастіше використовується протокол HTTP/HTTPS через REST API або GraphQL. REST API є стандартом де-факто завдяки своїй простоті, сумісності з багатьма мовами програмування та широкій підтримці інструментів для розробки. У REST кожен мікросервіс має визначені кінцеві точки (endpoints), через які інші сервіси можуть отримувати доступ до його функцій [23]. Це дозволяє чітко документувати API та підтримувати високу прозорість взаємодії між сервісами, схему роботи REST API показано на рисунку 10.

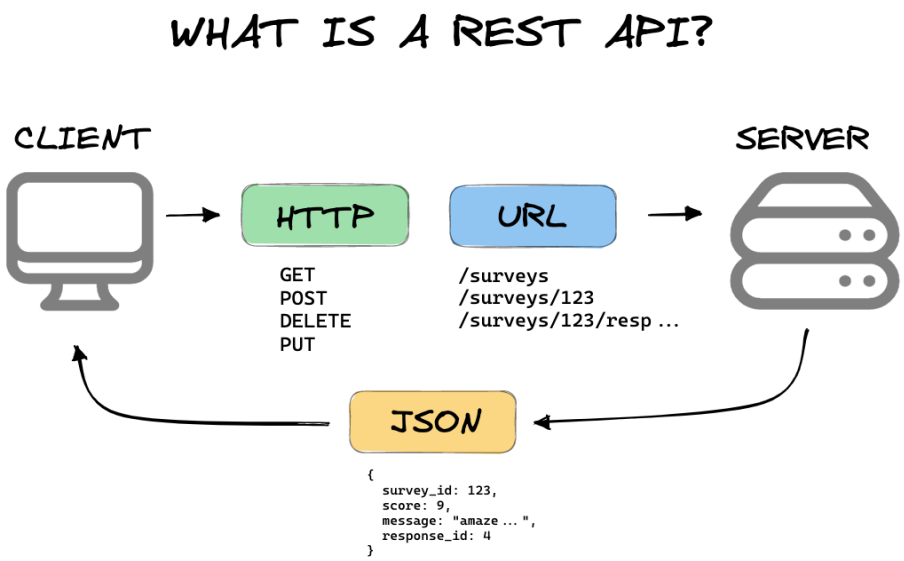


Рисунок 10 – Схема роботи REST API

GraphQL, своєю чергою, є альтернативою, що забезпечує більшу гнучкість у запитах, дозволяючи клієнтам отримувати лише необхідні дані. Цей підхід особливо корисний у складних системах із великими обсягами даних. Архітектуру GraphQL можемо побачити на рисунку 11

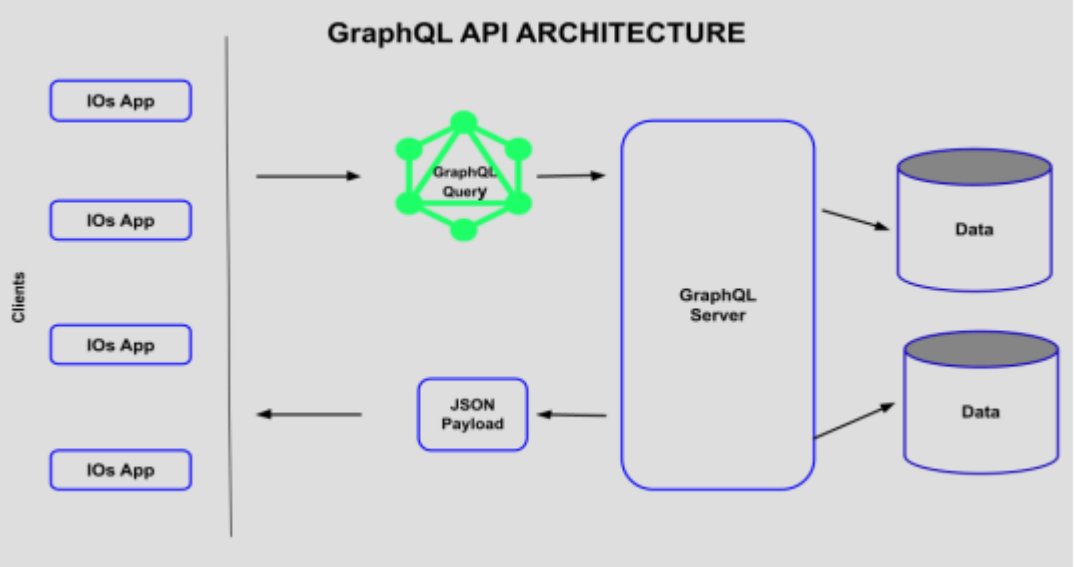


Рисунок 11 – Архітектура GraphQL

Проте синхронні протоколи мають свої обмеження, зокрема залежність від доступності сервісу в момент запиту. Асинхронна комунікація між мікросервісами базується на використанні брокерів повідомлень, таких як RabbitMQ або Kafka. У цій моделі сервіси публікують повідомлення у брокер, а інші сервіси підписуються на отримання цих повідомлень. Це зменшує залежність між сервісами, дозволяє обробляти завдання паралельно та забезпечує високу масштабованість системи. Однак важливо враховувати складність проєктування асинхронних систем, особливо для критично важливих транзакцій, де потрібна гарантія доставки повідомлень [9].

RabbitMQ є потужним інструментом для управління чергами повідомлень. Він підтримує різні моделі маршрутизації, такі як точка-точка або публікація-підписка, а також гарантує доставку повідомлень і управління пріоритетами. Схему роботи RabbitMQ бачимо на рисунку 12.

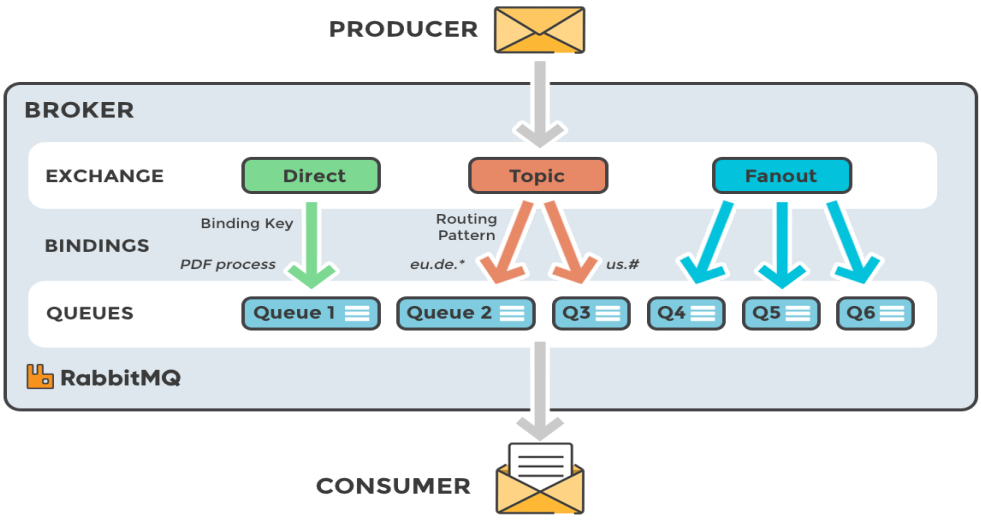


Рисунок 12 – Схема роботи RabbitMQ

Apache Kafka відрізняється можливістю обробки великих обсягів даних у режимі реального часу, що робить його популярним вибором для потокової обробки (stream processing). Ці інструменти також забезпечують високу відмовостійкість завдяки збереженню повідомлень і підтримці реплікацій. Екосистему Apache Kafka можемо розглянути на рисунку 13.

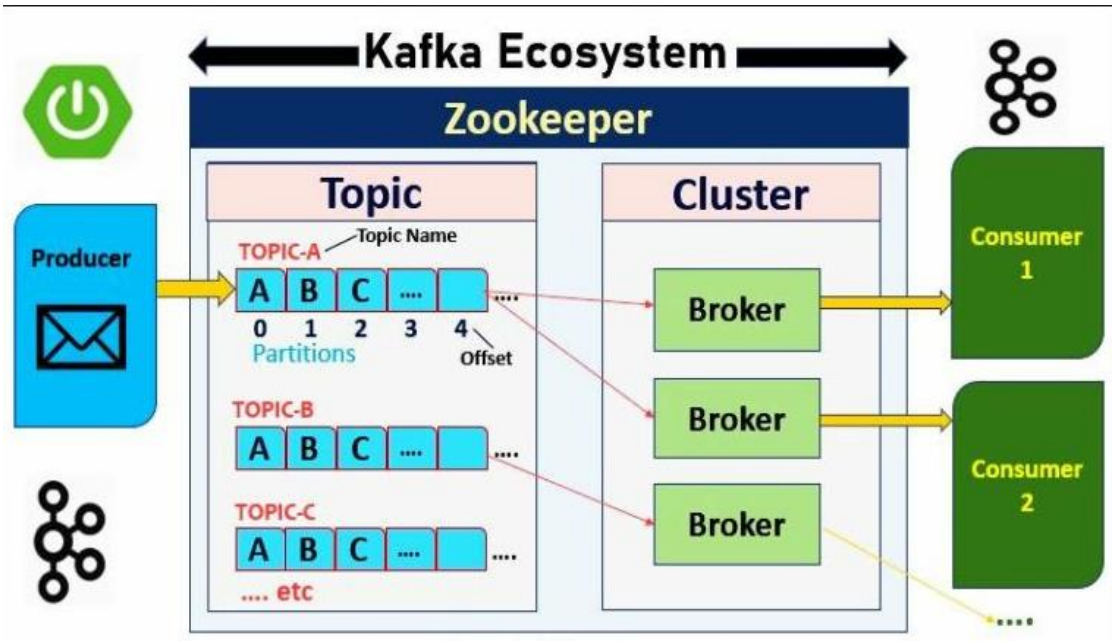


Рисунок 13 – Екосистема Apache Kafka

Для сценаріїв, коли важливо забезпечити низьку затримку та високу продуктивність, ефективним рішенням є використання RPC (віддаленого виклику процедур) через протоколи, такі як gRPC. gRPC є ефективним протоколом для синхронної комунікації в мікросервісній архітектурі. Використовуючи HTTP/2 та бінарний формат передачі даних, gRPC забезпечує швидший обмін інформацією, ніж REST, і знижує накладні витрати на передачу даних. Це робить його оптимальним вибором для сценаріїв з великим навантаженням, таких як потоки даних у реальному часі. Проте використання gRPC вимагає створення спеціальних клієнтських бібліотек, що може ускладнити інтеграцію в гетерогенні середовища [10]. Схему роботи gRPC бачимо на рисунку 14.

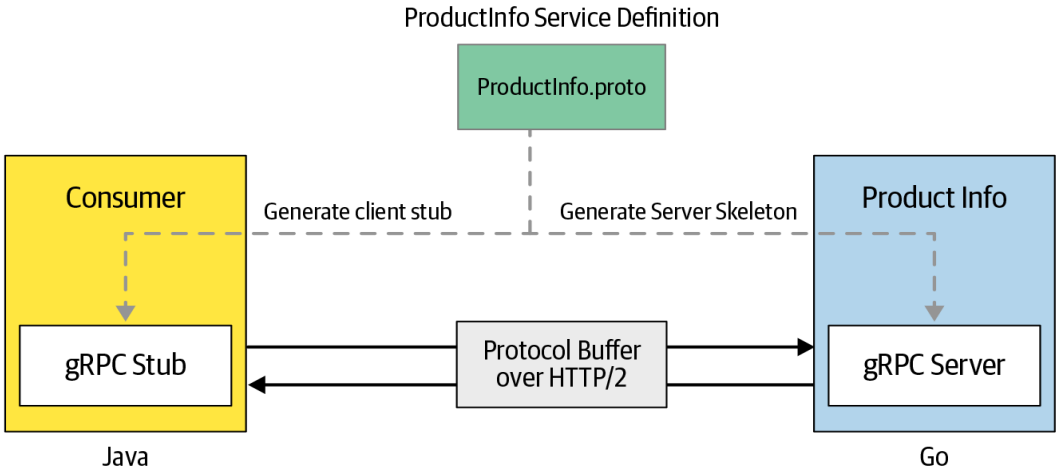


Рисунок 14 – Схема роботи gRPC

У подієво-орієнтованій архітектурі сервіси не взаємодіють безпосередньо, а реагують на події, створені іншими сервісами. Це забезпечує низьку зв’язаність та гнучкість системи. Наприклад, у середовищі Amazon Web Services для публікації подій можна використовувати Amazon Simple Notification Service (SNS). У цьому підході кожен сервіс підписується на ті події, які його стосуються, знижуючи потребу у взаємній залежності між сервісами. Але варто враховувати, що навіть цей підхід вимагає узгодження форматів подій між сервісами [11].

Ще однією важливою технологією в екосистемі мікросервісів є Service Mesh, наприклад, Istio або Linkerd. Service Mesh розв'язує проблеми маршрутизації, управління трафіком і забезпечення безпеки на рівні взаємодії між сервісами. За допомогою Service Mesh можна централізовано контролювати політики доступу, балансування навантаження і моніторинг комунікацій без необхідності вносити зміни в код самих сервісів. Схема роботи Linkerd зазначено на рисунку 15.

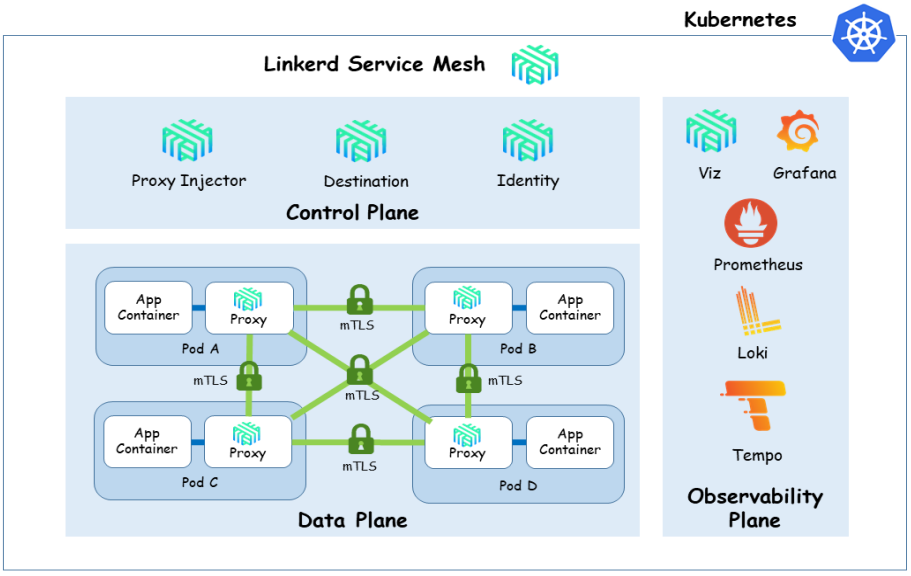


Рисунок 15 – Схема роботи Linkerd

Важливо враховувати й аспекти безпеки при виборі технологій для зв’язку мікросервісів. Використання TLS для шифрування даних, автентифікація за допомогою OAuth 2.0 або OpenID Connect, а також впровадження токенів доступу (JWT) є загальновизнаними підходами для захисту інформації.

Таким чином, комунікація між мікросервісами може бути реалізована різними способами залежно від завдань системи. Синхронні підходи, такі як REST і gRPC, забезпечують простоту й інтерактивність, тоді як асинхронні технології, зокрема RabbitMQ та Apache Kafka, пропонують більшу гнучкість і масштабованість. У сучасних масштабних системах часто використовують

комбінацію різних підходів, щоб досягти оптимального балансу між продуктивністю, надійністю та простотою підтримки.

2.5 Принципи побудови мікросервісної архітектури

Мікросервісна архітектура — це сучасний підхід до розробки програмного забезпечення, що передбачає створення застосунку у вигляді набору невеликих, незалежних сервісів, які працюють спільно для досягнення спільної мети. Кожен мікросервіс відповідає за конкретну бізнес-функцію і може бути розроблений, розгорнутий і масштабований окремо від інших. Це створює гнучку, адаптивну і надійну систему, що є особливо актуальним у контексті сучасних веб-застосунків [14].

Основним принципом мікросервісної архітектури є поділ відповідальності. Кожен сервіс має чітко визначену функціональну область і працює автономно. Такий підхід дозволяє уникнути монолітної структури, де всі компоненти жорстко зв'язані між собою. Натомість мікросервіси взаємодіють через чітко визначені інтерфейси, найчастіше у форматі REST API або gRPC. Ця незалежність сприяє простішій підтримці та оновленню системи: зміни в одному сервісі не впливають на інші [17].

Ще одним ключовим принципом є можливість незалежного розгортання. Завдяки мікросервісній архітектурі можна оновлювати або масштабувати окремі компоненти системи без необхідності перезапускати весь застосунок. Це забезпечує високу гнучкість і дозволяє швидко адаптуватися до змін у бізнес-вимогах чи технічних умовах. Зокрема, для різних мікросервісів можуть бути використані різні технологічні стеки, що дозволяє команді розробників вибирати оптимальні інструменти для кожної задачі.

Комунікація між мікросервісами є критичним аспектом архітектури. Вона може бути синхронною або асинхронною, залежно від потреб системи. Синхронна комунікація через HTTP-запити є найпростішим варіантом, однак для великих систем, де важлива стійкість до збоїв, часто використовується

асинхронний обмін повідомленнями через брокери, такі як RabbitMQ або Kafka. Це дозволяє зменшити залежність між сервісами та забезпечити обробку великих обсягів даних у режимі реального часу [12].

Стійкість до відмов є ще одним важливим принципом мікросервісної архітектури. Оскільки мікросервіси працюють незалежно, вихід з ладу одного компонента не призводить до повного припинення роботи системи. Для досягнення високої стійкості застосовуються різноманітні патерни, такі як "circuit breaker" та "retry". Це дозволяє автоматично реагувати на збої й забезпечує безперервність роботи навіть у випадку несправностей.

Мікросервісна архітектура не позбавлена викликів. Зокрема, вона створює складності у керуванні даними, оскільки кожен сервіс зазвичай має власну базу даних. Це вимагає ретельного планування для забезпечення консистентності даних та узгодженості між сервісами. Використання стратегій, таких як "event sourcing" і "distributed transactions", допомагає розв'язати ці проблеми та забезпечити узгодженість у розподілених системах [13].

Таким чином, мікросервісна архітектура є потужним інструментом для створення масштабованих і надійних веб-застосунків. Вона забезпечує гнучкість у розробці, дозволяє швидко адаптуватися до змін і підвищує стійкість системи до збоїв. Проте її впровадження вимагає високого рівня технічної компетентності та ретельного планування, щоб ефективно управляти складністю розподілених систем.

Схематичне зображення принципу роботи мікросервісної архітектури можна побачити на рисунку 16.

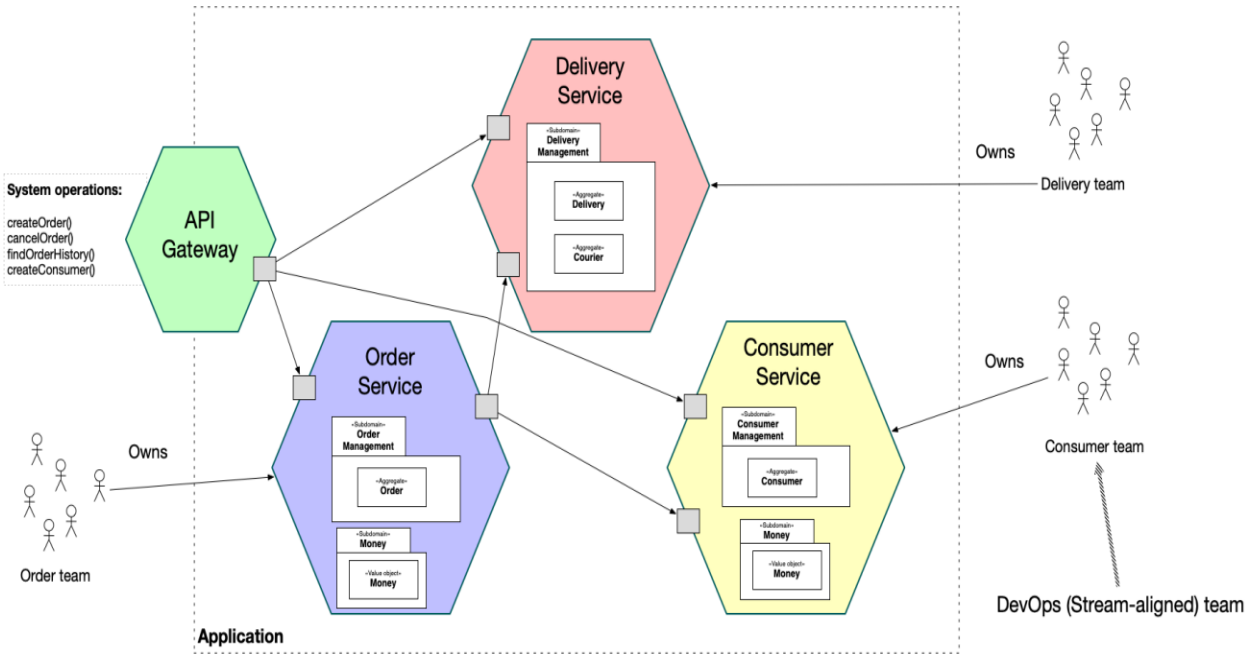


Рисунок 16 – Схема мікросервісної архітектури

3 РОЗРОБКА WEB-ЗАСТОСУНКА З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1 Опис предметної області

Предметна область, що розглядається в межах даного дослідження, охоплює розробку та впровадження мікросервісної архітектури у веб-застосунках. У сучасних умовах швидкого розвитку інформаційних технологій та зростання вимог до вебдодатків виникає потреба у створенні систем, здатних ефективно обробляти великі обсяги даних, забезпечувати стабільну роботу за високих навантажень та легко масштабуватися у відповідь на динамічні зміни бізнес-вимог.

Веб-застосунки, особливо ті, що використовуються у корпоративному середовищі, часто стикаються з викликами щодо продуктивності, надійності та гнучкості. Традиційні монолітні архітектури, хоча й залишаються поширеними, обмежують можливості масштабування та впровадження нових функцій. Кожна зміна у монолітній системі вимагає ретельного тестування всього додатка, що ускладнює процес оновлення та знижує гнучкість. У контексті цієї роботи предметна область зосереджена на дослідженні можливостей мікросервісного підходу як альтернативи монолітному.

Основні аспекти предметної області включають аналіз способів організації взаємодії між мікросервісами, методів забезпечення їхньої незалежності та принципів побудови гнучких систем. Окрема увага приділяється питанням масштабованості, оскільки у вебдодатках із високим навантаженням важливо забезпечити можливість горизонтального масштабування окремих сервісів, а не всієї системи загалом. Це дозволяє оптимізувати використання ресурсів та підвищити продуктивність.

Значна увага також приділяється питанням надійності та відмовостійкості. Оскільки кожен мікросервіс є незалежним модулем, важливо забезпечити безперервну роботу системи навіть у випадку відмови

окремих компонентів. Для цього використовуються різні патерни, такі як "circuit breaker" та "retry", а також асинхронна обробка повідомлень за допомогою брокерів, таких як RabbitMQ або Kafka. Це дозволяє знизити залежність між сервісами та підвищити загальну стійкість системи.

Інтеграція мікросервісної архітектури з іншими сучасними технологіями, такими як контейнеризація (Docker) та оркестрація (Kubernetes), є ще одним важливим аспектом предметної області. Контейнери дозволяють створювати ізольовані середовища для кожного мікросервісу, що забезпечує стабільну роботу додатка незалежно від середовища, у якому він запускається. Оркестратори, такі як Kubernetes, автоматизують процес управління контейнерами, забезпечуючи балансування навантаження, масштабування та відновлення після збоїв.

Таким чином, предметна область охоплює не лише технічні аспекти впровадження мікросервісної архітектури, але й стратегічні питання, пов’язані з управлінням великими та складними системами. Вивчення цих аспектів є ключовим для розробки ефективних веб-застосунків, що відповідають сучасним вимогам ринку.

3.2 Архітектура системи

Архітектура представленої системи побудована на основі мікросервісного підходу, що забезпечує модульність, гнучкість та масштабованість. Завдяки використанню NestJS та його модуля @nestjs/microservices, кожен сервіс виконує чітко визначені функції, а взаємодія між компонентами відбувається за допомогою Transport.TCP, що дозволяє досягти низької затримки та стабільності в обміні даними.

Далі ми розглянемо основні компоненти архітектури.

Chat-API-Gateway відіграє роль центрального шлюзу для обробки запитів від клієнтської частини системи. Цей мікросервіс приймає HTTP-запити (GET, POST, DELETE тощо) та перенаправляє їх до відповідних

мікросервісів через TCP-з'єднання. Для створення клієнта використовується ClientProxyFactory, який формує підключення до необхідного сервісу на основі вказаних параметрів, таких як IP-адреса та порт. Наприклад, запит на отримання всіх зображень у кімнаті чату передається до Chat-Messages-Service з використанням методу send().

Схему роботи API-Gateway можемо детально розглянути на рисунку 17.

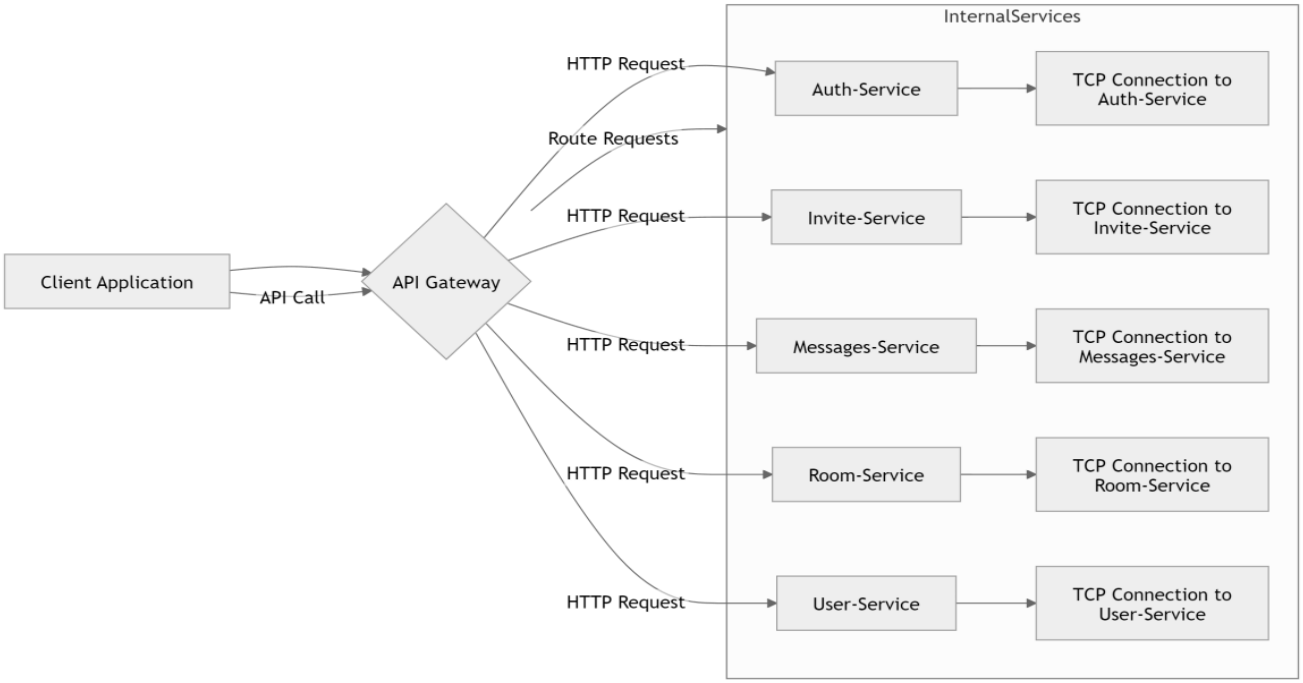


Рисунок 17 – Схема роботи API-Gateway

Chat-Auth-Service відповідає за аутентифікацію користувачів, реалізовану за допомогою JWT (JSON Web Tokens). Цей сервіс отримує запити від Chat-API-Gateway, перевіряє облікові дані користувача та генерує JWT для подальшої авторизації. Важливою особливістю є взаємодія з Chat-User-Service для отримання даних про користувача або створення нового облікового запису. Завдяки використанню TCP-з'єднання між сервісами забезпечується швидка та безпечна передача інформації без необхідності у брокерах повідомлень.

Процес взаємодії Auth-Service та User-Service можемо побачити на рисунку 18.

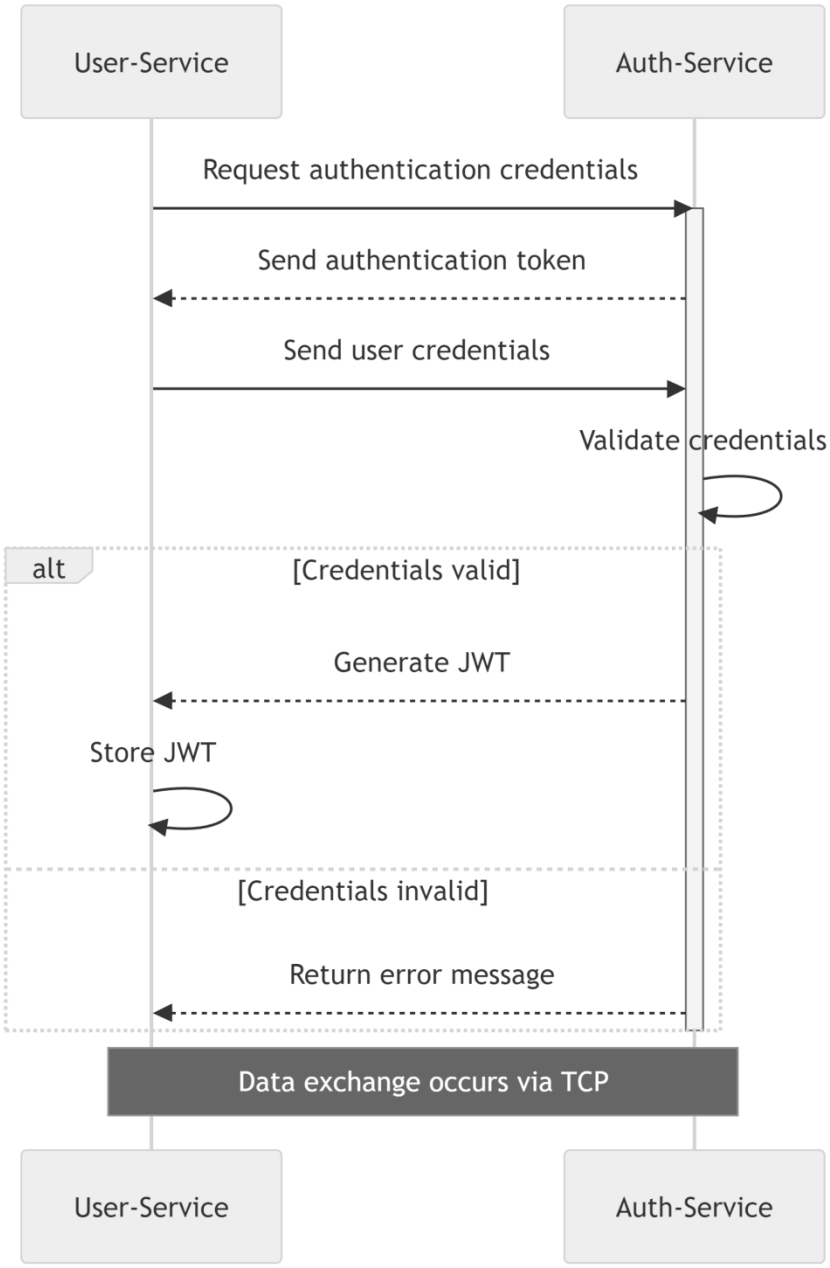


Рисунок 18 – Схема взаємодії Auth-Service та User-Service

Chat-Invite-Service забезпечує функціонал запрошення користувачів до кімнат чату. Цей сервіс тісно інтегрований з Chat-User-Service та Chat-Room-Service. Під час створення запрошення сервіс перевіряє статус користувача, отримуючи необхідну інформацію з User-Service, а також оновлює дані про кімнату в Room-Service. Використання TCP-з'єднання дозволяє мінімізувати затримки при обробці запитів і забезпечує високу продуктивність системи навіть при значному навантаженні.

Схему процесу запрошення юзера до чату можемо побачити на рисунку 19.

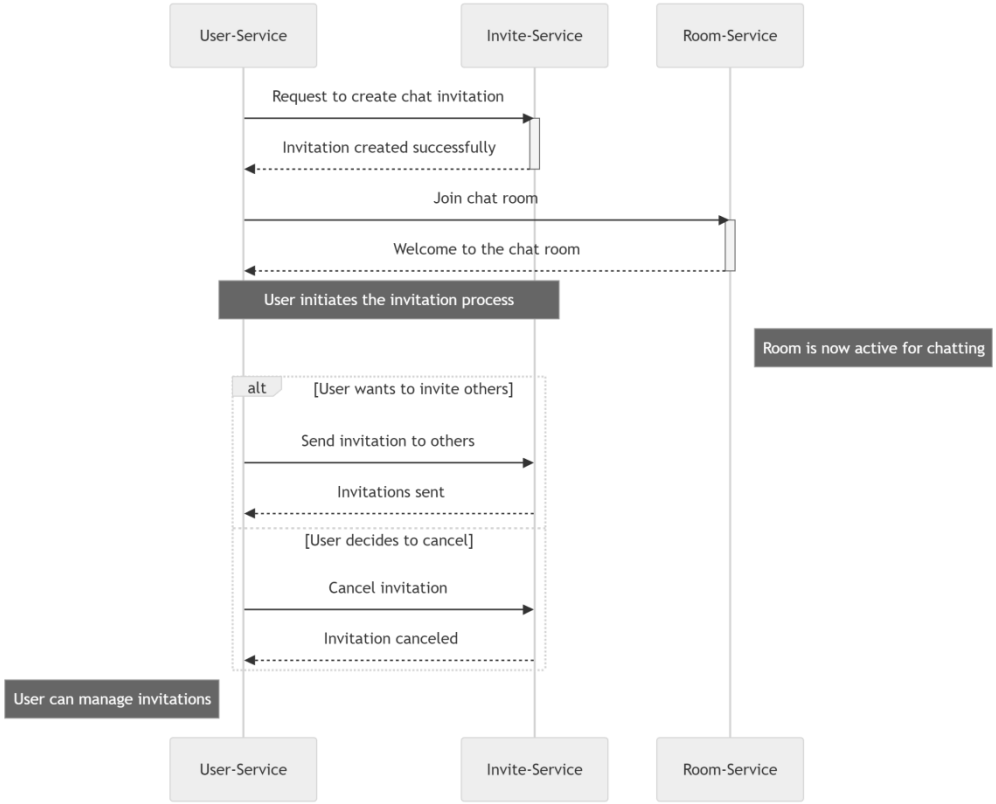


Рисунок 19 – Послідовна діаграма процесу створення та відправлення запрошення до чату

Chat-Messages-Service виконує ключову роль у зберіганні та обробці повідомлень. Цей сервіс отримує запити на отримання або відправлення повідомлень через Chat-API-Gateway і взаємодіє з Room-Service для перевірки ідентифікаторів кімнат. Повідомлення зберігаються у базі даних PostgreSQL, а їх структура дозволяє підтримувати складні запити, наприклад, пошук за датою або користувачем. Важливою особливістю є підтримка транзакцій для забезпечення консистентності даних, що реалізується за допомогою PrismaORM.

Схему відправки повідомлень і збереження їх в БД можемо побачити на рисунку 20.

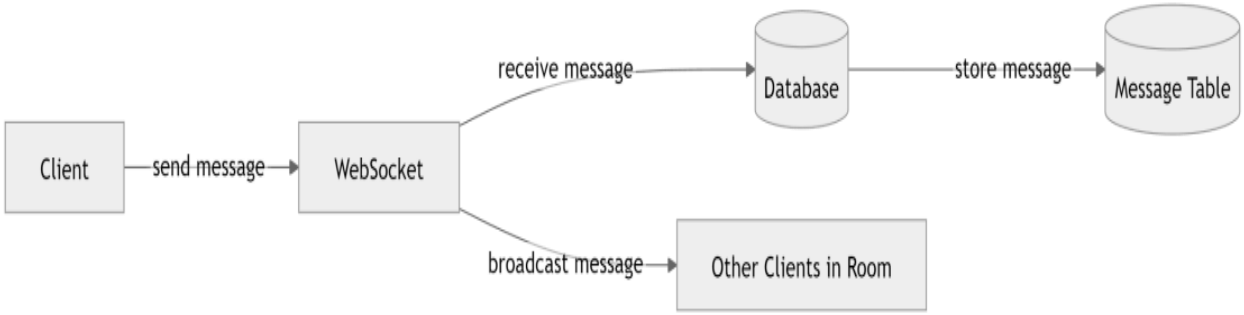


Рисунок 20 – Схема відправки повідомлень і збереження їх в БД

Рекомендована схема (Діаграма 4): Схема обробки повідомлень, що включає процес перевірки кімнати та збереження повідомлень у базі даних.

Chat-Room-Service керує створенням, оновленням і видаленням кімнат чату. Цей сервіс працює автономно та не відправляє запитів на інші сервіси, що підвищує його стабільність і спрощує підтримку. Дані про кімнати зберігаються у базі PostgreSQL, і кожна кімната має чітко визначені параметри, такі як приватність, список учасників та інвайти.

Chat-Socket-Service відповідає за обробку WebSocket-з'єднань, забезпечуючи реальний час передачі повідомлень між користувачами. Цей сервіс також відповідає за завантаження медіафайлів на Amazon S3, що робить його критично важливим для забезпечення інтерактивності чату. Chat-Socket-Service взаємодіє з User-Service, Room-Service та Messages-Service через TCP для отримання інформації про користувачів, кімнати та повідомлення відповідно.

Детальну схему роботи Socket-Service ми можемо побачити на рисунку 21.

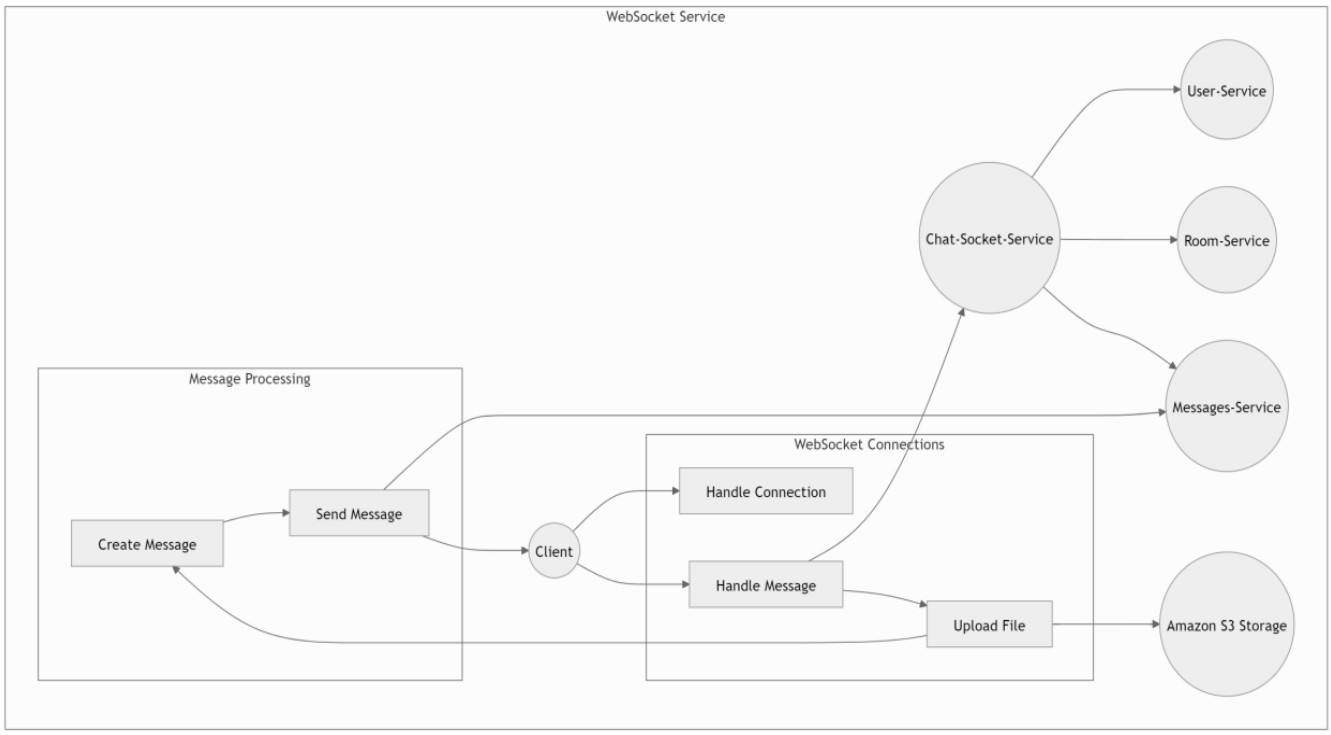


Рисунок 21 – Схема роботи Socket-Service

Chat-User-Service є основним компонентом для управління користувачами. Цей сервіс зберігає інформацію про користувачів, включаючи їх профілі, аватари та статуси. User-Service не надсилає запитів до інших сервісів, що робить його незалежним і зменшує ризик збоїв. Завдяки використанню PrismaORM у поєднанні з PostgreSQL забезпечується ефективне управління даними користувачів, включаючи операції створення, оновлення та видалення профілів.

Наступним ми розглянемо взаємодію між мікросервісами.

Для забезпечення стабільної та ефективної комунікації між мікросервісами використовується Transport.TCP у NestJS. Цей підхід забезпечує швидку передачу даних з мінімальною затримкою, що особливо важливо для реальних додатків, таких як чати. Відсутність брокерів повідомлень знижує складність архітектури, але водночас потребує ретельного планування TCP-з'єднань та обробки помилок. ClientProxyFactory і MessagePattern є основними механізмами для організації комунікації між сервісами в рамках NestJS.

ClientProxyFactory — це фабрика, що створює клієнтські з'єднання для мікросервісів. У контексті описаної системи, вона використовується для встановлення підключення до інших мікросервісів через Transport.TCP. Наприклад, сервіс Chat-API-Gateway ініціює з'єднання з Chat-Messages-Service, щоб отримати зображення для конкретної кімнати чату. Це підключення налаштовується за допомогою ClientProxyFactory, який створює TCP-клієнт, що взаємодіє з іншими мікросервісами на вказаному хості та порту.

Метод send() використовується для відправлення повідомлення з параметрами, що відповідають заданому патерну (наприклад, 'get.allImages'), а мікросервіс, який слухає цей патерн, обробляє запит і повертає відповідь. Таким чином, кожен мікросервіс, що обробляє запити, має чітко визначені патерни для отримання й обробки повідомлень від інших сервісів.

MessagePattern визначає патерн, на який мікросервіс слухає запити. Він дозволяє визначити, який саме тип повідомлення буде оброблений даним сервісом. Наприклад, у Chat-Messages-Service метод getAllImagesByRoom() обробляє запити за патерном 'get.allImages', і відповідно, коли запит надходить від Chat-API-Gateway, цей сервіс відправляє запит на отримання зображень для конкретної кімнати чату.

Простіше кажучи, ClientProxyFactory та MessagePattern забезпечують централізовану обробку запитів між мікросервісами, що є основою для ефективної та гнучкої архітектури без необхідності у використанні брокерів повідомлень.

Приклад використання ClientProxyFactory та MessagePattern приведено у лістингах 1 та 2. А схему можемо побачити на рисунку 22.

Лістинг – *1. Створення та підключення ClientProxyFactory в API-Gateway*

```
    constructor(private readonly configService:
ConfigService) {
```

```
  this.messages_client = ClientProxyFactory.create({
    transport: Transport.TCP,
    options: {
      host: '127.0.0.1',
      port: 5004,
    },
  });
}
```

Лістинг – 2. *Використання MessagePattern в Invite-Service*

```
    @MessagePattern('post.create')
async createInvite(dto: InviteDto): Promise<IInvite> {
  await this.inviteService.findInviteByRoom(dto.roomId);

  const inviteCreated: IInvite = await
this.inviteService.createInvite(dto);

  this.room_client.send('post.addInviteLink', {
    inviteLink: dto.inviteLink,
    roomId: dto.roomId,
  });
  return inviteCreated;
}
```

Наступна важлива частина архітектури системи це база даних, розглянемо її детально.

Як згадувалося раніше, у проєкті використовується реляційна база даних PostgreSQL для зберігання даних, пов'язаних із користувачами, кімнатами чату, повідомленнями та файлами. Для роботи з базою даних використовується PrismaORM, що дозволяє спростити взаємодію з БД та забезпечити високу продуктивність запитів завдяки підтримці складних структур даних, таких як зв'язки між користувачами та кімнатами.

Структура бази даних розроблена з урахуванням можливостей масштабування та високої доступності, що є важливим для чату з високим навантаженням. Основні моделі бази даних включають:

- User: Зберігає інформацію про користувачів, включаючи унікальні поля, такі як ім'я користувача, телефон, електронна пошта та аватар.
- Room: Зберігає інформацію про кімнати чату, включаючи налаштування, такі як приватність, учасники та інвайти.
- Messages: Містить текстові повідомлення та їх зв'язок з користувачем та кімнатою.
- File: Містить медіафайли, прикріплені до повідомлень.
- InviteList: Використовується для зберігання запрошень до кімнат чату, що мають статуси прийняття.

Ця структура дозволяє ефективно організувати зберігання та доступ до даних, забезпечуючи при цьому високу гнучкість і можливість швидкого масштабування.

ER-діаграму бази даних можемо розглянути на рисунку 23.

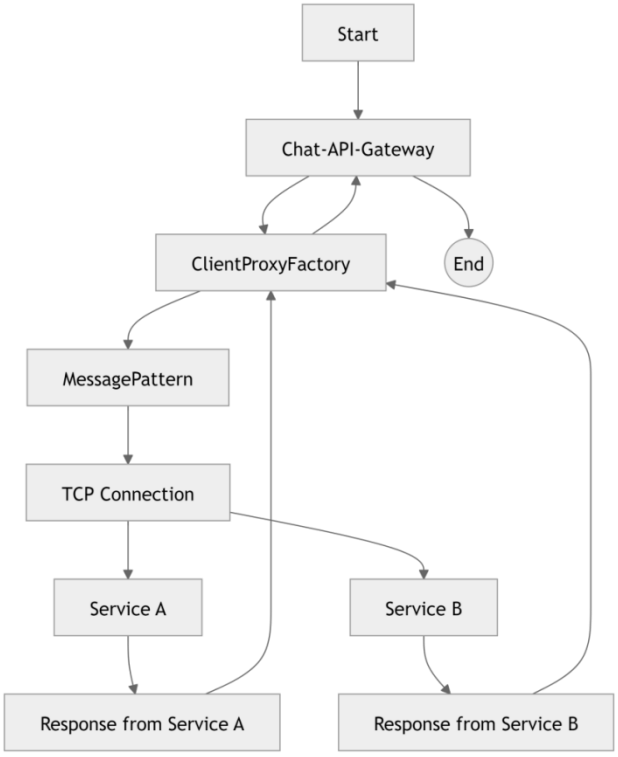


Рисунок 22 – Схема роботи ClientProxyFactory та MessagePattern

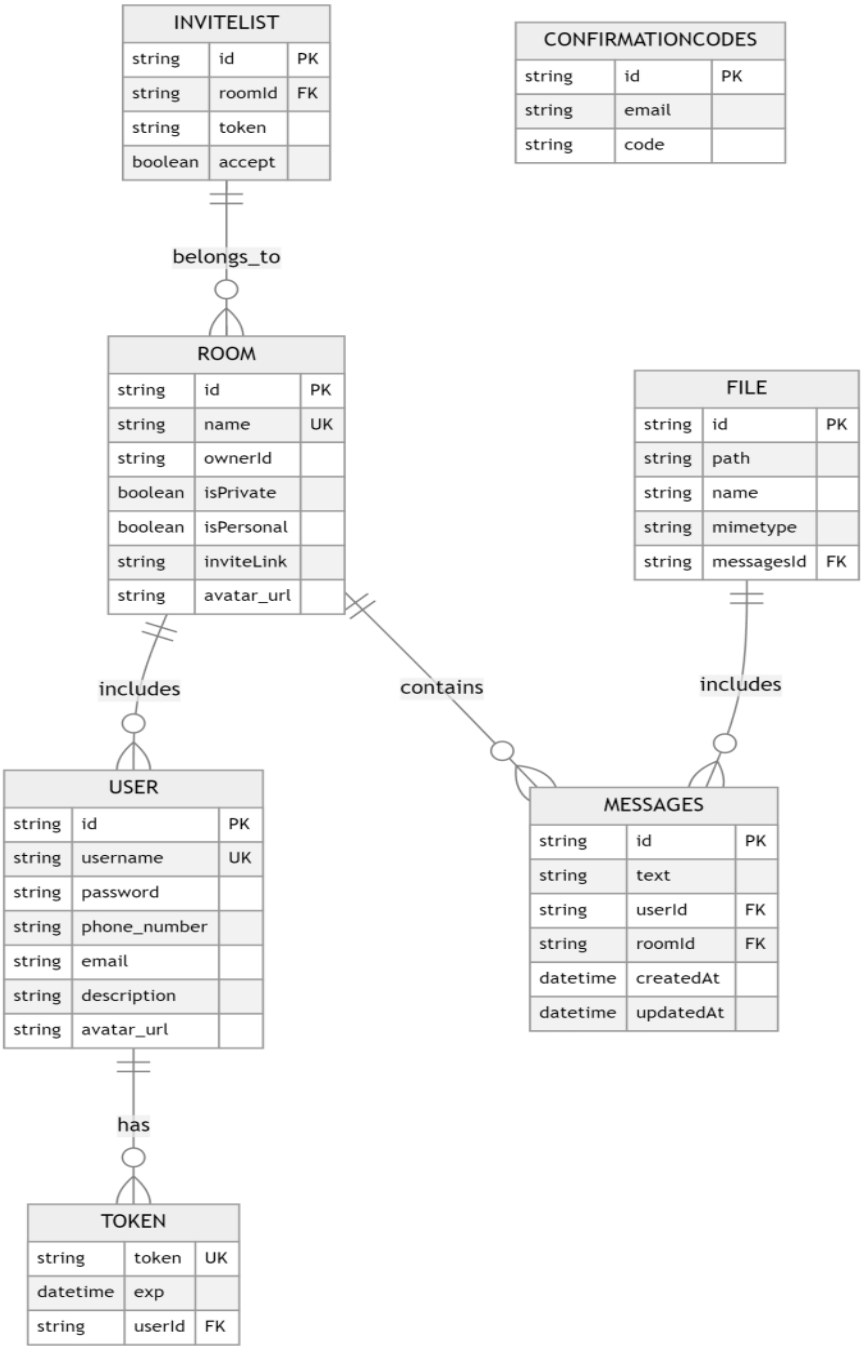


Рисунок 23 – ER-діаграма бази даних

Також важливо буде зазначити що в системі передбачена аутентифікація користувачів через JWT (JSON Web Tokens), що дозволяє забезпечити безпечний доступ до ресурсів. Після аутентифікації користувача Chat-Auth-Service генерує JWT, який використовується для подальших запитів до API. Це дозволяє ізолювати сервіси один від одного, зменшуючи ризики несанкціонованого доступу та спрощуючи управління правами доступу.

Токени мають обмежений термін дії, що забезпечує додатковий рівень безпеки. Кожен токен містить метадані, такі як час створення та термін дії, що дозволяє швидко відстежувати актуальність токенів у системі.

Архітектура системи побудована з урахуванням найкращих практик мікросервісного підходу, що забезпечує масштабованість, високу доступність і швидку взаємодію між компонентами. Використання NestJS, Transport.TCP для комунікації, PrismaORM для роботи з базою даних та JWT для безпеки створює потужну та надійну основу для побудови динамічного чату в реальному часі. Зокрема, відсутність брокерів повідомлень знижує складність архітектури та забезпечує зручність у тестуванні та відлагодженні.

3.3 Функціональні вимоги системи

Функціональні вимоги системи визначають основні можливості та функціональні обмеження, які повинні бути реалізовані для забезпечення ефективної роботи всіх компонентів мікросервісної архітектури. У контексті системи чату, побудованої на основі мікросервісів, функціональні вимоги охоплюють всі аспекти від управління користувачами до забезпечення безпеки та інтеграції з зовнішніми сервісами, такими як хмарне зберігання медіафайлів. Нижче наведено основні функціональні вимоги, що визначають поведінку системи.

Реєстрація та автентифікація користувачів.

Chat-Auth-Service має забезпечувати реєстрацію нових користувачів, а також автентифікацію наявних через JWT (JSON Web Tokens). У системі передбачається наступна функціональність:

- Користувач може створити акаунт, надавши мінімальні дані (ім'я користувача).
- Після успішної реєстрації користувач отримує токен доступу, який використовується для подальших запитів до API.

- Токен зберігається на стороні клієнта та надсилається разом із запитами до Chat-API-Gateway.
- Кожен запит до системи (окрім реєстрації) вимагає дійсного токена, що перевіряється на сервері для забезпечення безпеки та авторизації користувача.

Управління користувачами.

Chat-User-Service має забезпечувати основні функції управління профілем користувача, включаючи:

- Створення профілю користувача з можливістю оновлення інформації (ім'я, аватар, опис).
- Зміна пароля або інших даних, які стосуються користувача.
- Надання користувачу аватара через збереження файлів, завантажених користувачем, що забезпечує персоналізацію профілю.

Створення та управління кімнатами чату.

Chat-Room-Service відповідає за створення та адміністрування кімнат чату, що включає такі функції:

- Створення нових кімнат із задаванням параметрів, таких як назва кімнати, приватність (відкрита чи закрита), тип кімнати (звичайна чи персональна).
- Кожна кімната має унікальний ідентифікатор та список учасників, який оновлюється при додаванні або видаленні користувачів.
- Можливість створення інвайтів для запрошення інших користувачів до кімнати чату через Chat-Invite-Service.
- Надання можливості керувати кімнатою лише власнику, який має права на редагування параметрів або видалення кімнати.

Запрошення до кімнат чату.

Chat-Invite-Service забезпечує функціональність запрошень до кімнат чату, що включає:

- Генерація унікальних запрошень для користувачів, які можуть долучитися до кімнати.

- Відправка інвайтів через систему повідомлень або за допомогою електронної пошти.
- Обробка прийняття чи відмови від запрошення, що оновлює статус користувача в кімнаті.

Взаємодія з повідомленнями.

Chat-Messages-Service має реалізувати функції для взаємодії з повідомленнями:

- Надсилання текстових повідомлень у конкретну кімнату.
- Можливість прикріплення файлів до повідомлень (фото, відео, документи), збереження яких реалізовано через інтеграцію з S3.
- Історія повідомлень в кожній кімнаті.
- Забезпечення доставки повідомлень в реальному часі за допомогою Chat-Socket-Service, що використовує WebSocket.

Робота з медіафайлами.

Chat-Socket-Service відповідає за завантаження та обробку медіафайлів, включаючи:

- Завантаження медіафайлів на S3 або інший хмарний сервер.
- Відправка посилання на збережений файл користувачам, які перебувають у відповідній кімнаті чату.

Реалізація реального часу через WebSocket.

Chat-Socket-Service забезпечує обмін повідомленнями в реальному часі:

- Підключення клієнтів до системи через WebSocket для миттєвої доставки повідомлень.
- Підтримка двосторонньої комунікації, що дозволяє як клієнту відправляти, так і отримувати повідомлення.
- Обробка подій, таких як підключення, відключення або виведення користувачів із кімнат.

Інтерфейс для користувачів.

Система має мати зручний і інтуїтивно зрозумілий інтерфейс для взаємодії з системою.

3.4 Програмна реалізація застосунку

Програмна реалізація системи здійснена з використанням мікросервісної архітектури, що забезпечує гнучкість і масштабованість застосунку. Всі компоненти системи розроблені на основі NestJS, фреймворку для Node.js, який забезпечує високу продуктивність та структурованість коду завдяки своїй модульній архітектурі. Розробка кожного мікросервісу здійснювалась за принципом чітко визначених функціональних блоків, що дає змогу реалізувати автономні сервіси, які можуть взаємодіяти між собою через визначені інтерфейси. Одним із ключових аспектів реалізації була вибірка Transport.TCP як основного механізму комунікації між мікросервісами.

В основі програмної реалізації кожного мікросервісу лежить принцип інкапсуляції бізнес-логіки та чітке розділення відповідальностей. Кожен мікросервіс, зокрема Chat-Auth-Service, Chat-API-Gateway, Chat-Messages-Service, має свій набір контролерів і сервісів, які виконують конкретні завдання. Наприклад, Chat-Auth-Service відповідає за процес реєстрації користувачів і перевірку їх автентичності, а Chat-Messages-Service займається обробкою повідомлень і їх збереженням у базі даних.

Реалізацію частини методів цих сервісів приведено в лістингах 3 та 4.

Лістинг – 3. *Метод Auth-Service для встановлення refresh-token в cookies*

```
    async setRefreshTokenToCookies(
  tokens: ITokens,
  res: Response,
): Promise<void> {
  if (!tokens) {
    throw new UnauthorizedException();
  }
  res.cookie(
    this.configService.get('REFRESH_TOKEN'),
```

```
    tokens.refreshToken.token,
    {
      httpOnly: true,
      sameSite: 'lax',
      expires: new Date(tokens.refreshToken.exp),
      secure:
        this.configService.get('NODE_ENV', 'development')
=== 'production',
      path: '/',
    },
  );
  res.status(HttpStatus.CREATED).json({ accessToken:
tokens.accessToken });
}
```

Лістинг – 4. *Метод* Messages-Service *для збереження повідомлення в БД*

```
    async createMessage(dto: MessageDto): Promise<IMessage>
{
  let filesInput: IFile[] = [
    {
      name: '',
      mimetype: '',
      path: '',
    },
  ];
  if (dto.files.length > 0) {
    filesInput = dto.files.map((file: IFile) => ({
      name: file.name,
      path: file.path,
      mimetype: file.mimetype,
    }));
  }
```

```
  return this.messageRepository.createMessage(dto,
filesInput);
}
```

При цьому важливим моментом є використання бібліотеки PrismaORM для роботи з базою даних PostgreSQL, що забезпечує зручний і високопродуктивний інтерфейс для виконання запитів та маніпуляцій з даними. База даних має кілька основних таблиць, таких як User, Room, Messages, File, що зберігають інформацію про користувачів, кімнати чату, повідомлення та медіафайли. Наприклад, кожне повідомлення в Messages має зв'язок з конкретною кімнатою чату, що дозволяє організувати ефективний доступ до історії чатів за допомогою простих SQL-запитів.

Приклади звернення сервісів до БД приведено в лістингах 5-6.

Лістинг – 5. *Метод* Messages-Repository *для збереження повідомлення в БД*

```
    async createMessage(dto: MessageDto, filesInput:
IFile[]): Promise<IMessage> {
  return this.prismaService.messages.create({
    data: {
      text: dto.text,
      userId: dto.userId,
      roomId: dto.roomId,
      username: dto.username,
      files: {
        create: filesInput ? filesInput : [],
      },
    },
    include: {
      files: true,
    },
```

```
  });
}
```

Лістинг – 6. *Метод Auth*-Repository *для збереження токена в БД*

```
  async getRefreshToken(userId: string, userAgent:
string): Promise<Token> {
  const _token = await this.prismaService.token.findFirst({
    where: {
      userId,
    },
  });
  const token = _token?.token ?? '';
  return this.prismaService.token.upsert({
    where: { token, userAgent },
    update: {
      token: v4(),
      exp: add(new Date(), { months: 1 }),
    },
    create: {
      token: v4(),
      exp: add(new Date(), { months: 1 }),
      userId,
      userAgent,
    },
  });
}
```

Вся комунікація між мікросервісами реалізована за допомогою TCP-з'єднань, що дозволяє ефективно передавати дані та виключає необхідність у використанні брокерів повідомлень, як це прийнято в деяких інших архітектурних підходах. ClientProxyFactory є інтерфейсом для створення TCP-клієнтів, через який кожен мікросервіс може ініціювати запити до інших

компонентів системи. Наприклад, коли Chat-API-Gateway отримує запит на отримання зображень для кімнати чату, він використовує messages_client.send(), щоб звернутися до Chat-Messages-Service і отримати відповідь. У цьому контексті важливо зазначити, що MessagePattern на кожному сервісі визначає відповідні патерни для приймання повідомлень. Так, Chat-Messages-Service слухає патерн 'get.allImages', що дозволяє йому обробляти запити на отримання всіх зображень для конкретної кімнати чату.

При програмуванні кожного мікросервісу особлива увага була приділена масштабованості та стійкості до збоїв. Для цього кожен мікросервіс розроблений як автономний компонент, що дозволяє масштабувати окремі частини системи без впливу на інші. Це дозволяє системі ефективно справлятися з підвищеним навантаженням або відмовами в одному з компонентів, не порушуючи роботу всієї системи. Важливою частиною реалізації є також обробка помилок та валідація даних на рівні кожного мікросервісу. Наприклад, у Chat-Auth-Service є обробка помилок автентифікації, яка дозволяє забезпечити безпечний доступ до API тільки для авторизованих користувачів, а в Chat-User-Service реалізовано детальну валідацію введених даних для профілю користувача.

Приклади валідації та обробки помилок приведено в лістингах 7 та 8.

Лістинг – 7. *Клас валідатор для полів юзера*

```
export class UserCreateDto {
  @IsString()
  @IsNotEmpty()
  username: string;

  @IsString()
  @MinLength(8)
  password?: string;
}
```

Лістинг – 8. *Обробка помилки якщо юзер з таким ім'ям вже існує*

```
    if (Username) {
  throw new BadRequestException('This username or email is
already in use');
}
```

Що стосується медіафайлів, то Chat-Socket-Service відповідає за завантаження файлів на хмарний сервер S3, що дозволяє зберігати великі обсяги медіаконтенту та забезпечує їх безпечне зберігання в реальному часі. При цьому, кожен медіафайл, прикріплений до повідомлення в Chat-Messages-Service, отримує унікальне посилання, яке може бути передано іншим учасникам кімнати чату.

Обробку медіа та завантаження його на S3 приведено в лістингах 9 та 10.

Лістинг – 9. *Обробка медіа*

```
    if (message.files.length > 0) {
  let index = 0;
  console.log(fullChunkArr);
  for (const fileInfo of message.files) {
    console.log(fullChunkArr[index]);
    const uploadResult: S3.ManagedUpload.SendData =
      await this.uploadPublicFile(
        fullChunkArr[index],
        fileInfo.name,
        message.userId,
        fileInfo.mimetype,
      );
    results.push({
      path: uploadResult.Location,
      name: fileInfo.name,
      mimetype: fileInfo.mimetype,
    });
```

```
    index++;
  }
}
```

Лістинг – 10. *Метод для завантаження медіа на S3*

```
    async uploadPublicFile(
  dataBuffer: Buffer,
  filename: string,
  userId: string,
  filetype,
): Promise<S3.ManagedUpload.SendData> {
  const s3 = new S3();
  return s3
    .upload({
      Bucket:
this.configService.get('AWS_PUBLIC_BUCKET_NAME'),
      Body: dataBuffer,
      Key: `${userId}/${filetype}/${filename}${v4()}`,
    })
    .promise();
}
```

Для забезпечення безпеки комунікацій між сервісами в усіх частинах застосунку використовується JWT для автентифікації користувачів. Токени доступу генеруються після входу користувача в систему і використовуються для подальших запитів. Усі запити до сервісів, окрім тих, що стосуються реєстрації або аутентифікації, вимагають передачі валідного токена, що забезпечує надійний контроль доступу до ресурсів.

У клієнтській частині, яка була розроблена за допомогою React, також реалізовано підтримку авторизації через JWT. Клієнт отримує токен під час входу користувача, і цей токен зберігається в локальному сховищі браузера.

Кожен запит до серверної частини супроводжується цим токеном, що дозволяє гарантувати безпеку під час взаємодії з API.

Система використовує WebSocket для обміну повідомленнями в реальному часі. Це дає змогу миттєво доставляти повідомлення користувачам, що є критично важливим для чатів у реальному часі. Вебсокети використовуються для двосторонньої комунікації, тобто користувач може не лише отримувати, але й надсилати повідомлення без необхідності перезавантажувати сторінку. Реалізацію методів WebSocket наведено в лістингах 11 та 12.

Лістинг – 11. *Метод для відловлення повідомлень про видалення повідомлень.*

```
    @SubscribeMessage('chatToServerDelete')
async handleDeleteMessage(
  @ConnectedSocket() client: Socket,
  @MessageBody() dto: MessageDeleteDto,
) {
  const s3: S3 = new S3();

  const observableMessage: Observable<IMessage> =
this.messages_client.send(
    'delete.byId',
    dto.id,
  );

  const deleteMessage = await
firstValueFrom(observableMessage);

  if (deleteMessage) {
    deleteMessage.files.map(async (file: IFile):
Promise<void> => {
      await s3
```

```
        .deleteObject({
          Bucket:
this.configService.get('AWS_PUBLIC_BUCKET_NAME'),
          Key:
`${deleteMessage.userId}/${file.mimetype}/${file.name}`,
        })
        .promise();
    });
  }
  this.wss.to(dto.roomId).emit('chatToClientDelete',
dto.id);
}
```

Лістинг – 12. *Метод для отримання юзерів в конкретній кімнаті.*

```
    @SubscribeMessage('getUsersInRoom')
handleUsersInRoom(
  @ConnectedSocket() client: Socket,
  @MessageBody() roomId: { roomId: string },
): void {
  const namespace: Namespace<DefaultEventsMap, any> =
this.wss.of('/');
  const rooms: Map<string, Set<string>> =
namespace.adapter.rooms;
  const room: Set<string> = rooms.get(roomId.roomId);
  if (room) {
    const userCount: number = room.size;
    this.wss.to(roomId.roomId).emit('UsersInRoom',
userCount);
  } else {
    this.wss.to(roomId.roomId).emit('UsersInRoom', 0);
  }
}
```

Що стосується клієнтської частини, то вона була розроблена на базі React, що забезпечує зручний та інтерактивний інтерфейс для користувачів. Використання Redux дозволило ефективно керувати станом додатка, зокрема для збереження інформації про користувача, токени доступу та історію повідомлень. React Router відповідає за маршрутизацію в додатку, що дозволяє користувачам переміщатися між різними кімнатами та екраном профілю.

Завдяки використанню React, клієнтська частина є динамічною і швидко реагує на зміни, що дозволяє забезпечити високу взаємодію з користувачем. Компоненти інтерфейсу чітко розділені на функціональні блоки, що дозволяє легко розширювати і змінювати додаток в майбутньому.

Для управління станом клієнтського додатка використано Redux, який дозволяє ефективно зберігати та оновлювати стан додатка на всіх етапах взаємодії користувача з чатом. Особливо важливим є зберігання інформації про поточний користувацький сеанс, що включає токен доступу, дані про користувача та його активні чати.

Крім того, для обробки медіафайлів було використано вбудоване рішення для завантаження файлів безпосередньо з клієнтської частини на S3. Користувач може вибирати файли з локального сховища або камери (якщо мова йде про мобільний пристрій) і завантажувати їх безпосередньо в чат. Ця можливість була реалізована через API, яке забезпечує інтеграцію з Chat-Socket-Service, що надсилає медіафайли до хмарного сервера.

Для забезпечення обміну повідомленнями в реальному часі на клієнтській частині використовуються WebSockets. Підключення до WebSocket-серверу здійснюється через Socket.io — популярну бібліотеку для роботи з WebSocket у JavaScript. Кожен раз, коли користувач надсилає або отримує повідомлення, ці події миттєво обробляються через WebSocket-з'єднання. Завдяки цьому вдається забезпечити максимальну інтерактивність, а також зменшити затримки, що є критично важливим у чатах з реальним

часом. Реалізація WebSocket на клієнтській частині наведено в лістингах 13 та 14.

Лістинг – 13. *Клас для ініціалізації WebSocket*

```
class SocketApi {
static socket: null | Socket

static createConnection() {
    this.socket = io("http://localhost:80/")

    this.socket.on('connect', () => {
        console.log("Connect")
    })

    this.socket.on('joinedRoom', (data) => {
        console.log(`joined to room with id: ${data}`)
    })

    this.socket.on('leftRoom', (data) => {
        console.log(`left room with id: ${data}`)
    })
}
}
```

Лістинг – 14. *Відправка повідомлення через WebSocket*

```
SocketApi.socket?.emit('chatToServer', {
  text: text,
  roomId: params.Id,
  userId: id,
  username: username,
  createdAt: new Date(),
```

```
  files: filesToSend,
});
```

У загальному, програмна реалізація чату з мікросервісною архітектурою є потужною та гнучкою платформою для створення інтерактивних чат-додатків. Використання таких технологій, як NestJS, React, WebSocket і S3 дозволяє забезпечити високу продуктивність, масштабованість та зручний інтерфейс для користувачів.

Завдяки чітко спроєктованій архітектурі та використанню сучасних технологій, система здатна обробляти великі обсяги запитів і забезпечувати користувачам комфортний досвід при використанні чат-системи в реальному часі.

3.5 Розробка інтерфейсу

Інтерфейс чату був спроєктований з урахуванням сучасних принципів дизайну, що забезпечує зручність користувача і дозволяє швидко знайти необхідні функції. Інтерактивні компоненти, такі як форма введення повідомлення, список чатів, кнопки для завантаження файлів та інші елементи, були розроблені так, щоб забезпечити простоту використання та швидкий доступ до основних функцій [20].

Далі, для кращого розуміння, будуть представлені скриншот інтерфейсу чату, які ілюструють основні екрани та функціональні можливості застосунку (див. Рис. 24-29).



Рисунок 24 – Інтерфейс вікна реєстрації (відправка коду на email)

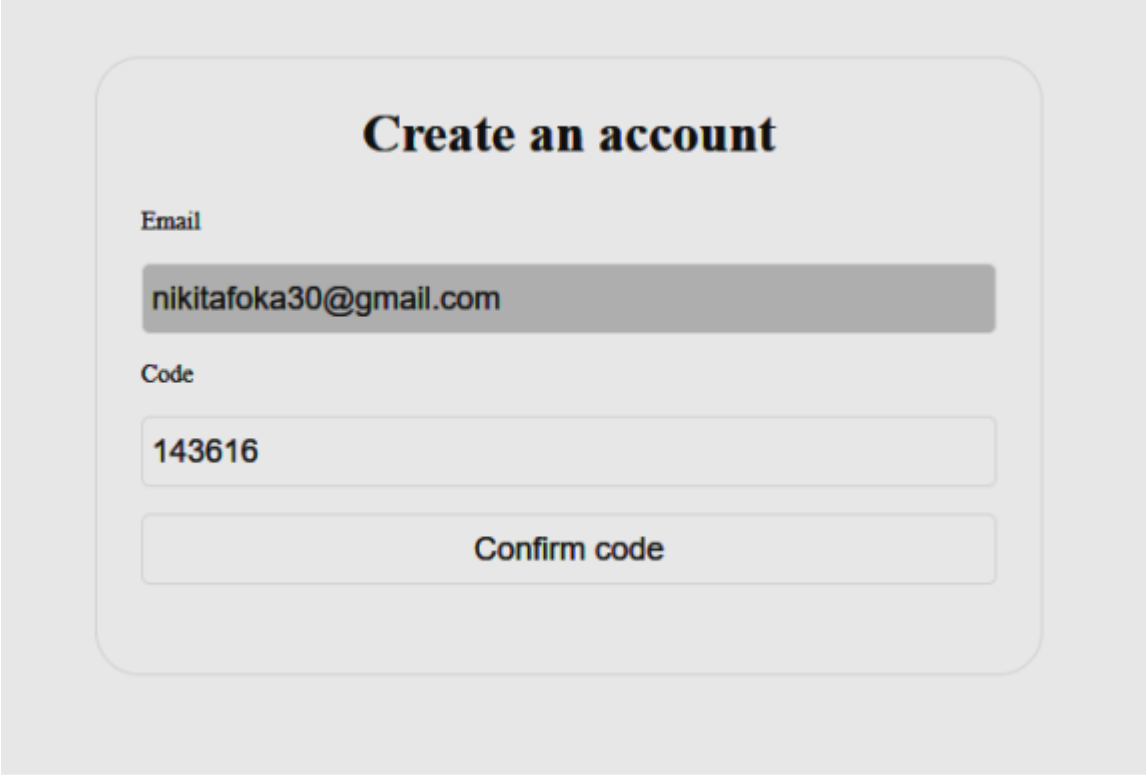


Рисунок 25 – Інтерфейс вікна реєстрації (написання коду підтвердження)

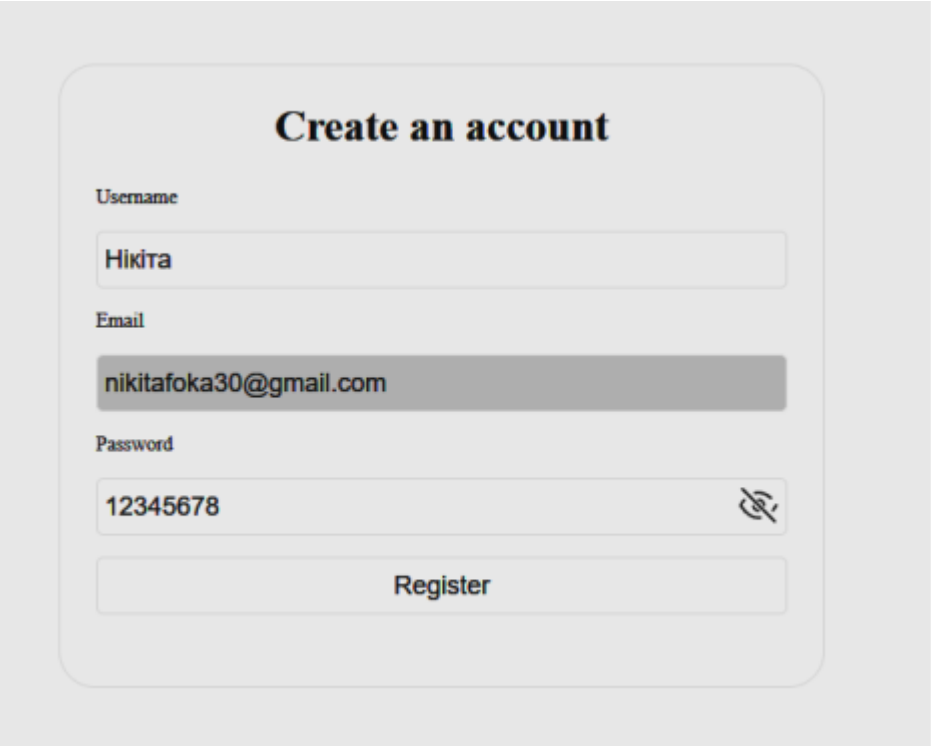


Рисунок 26 – Інтерфейс вікна реєстрації (написання імені та паролю)

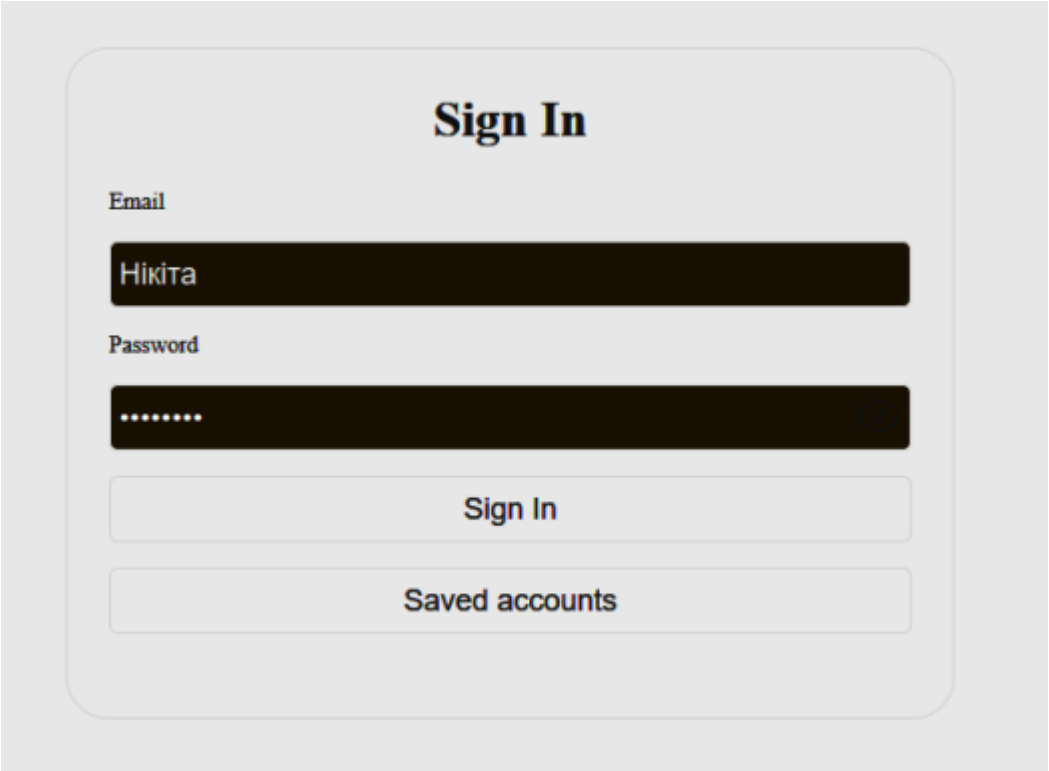


Рисунок 27 – Інтерфейс вікна логіну

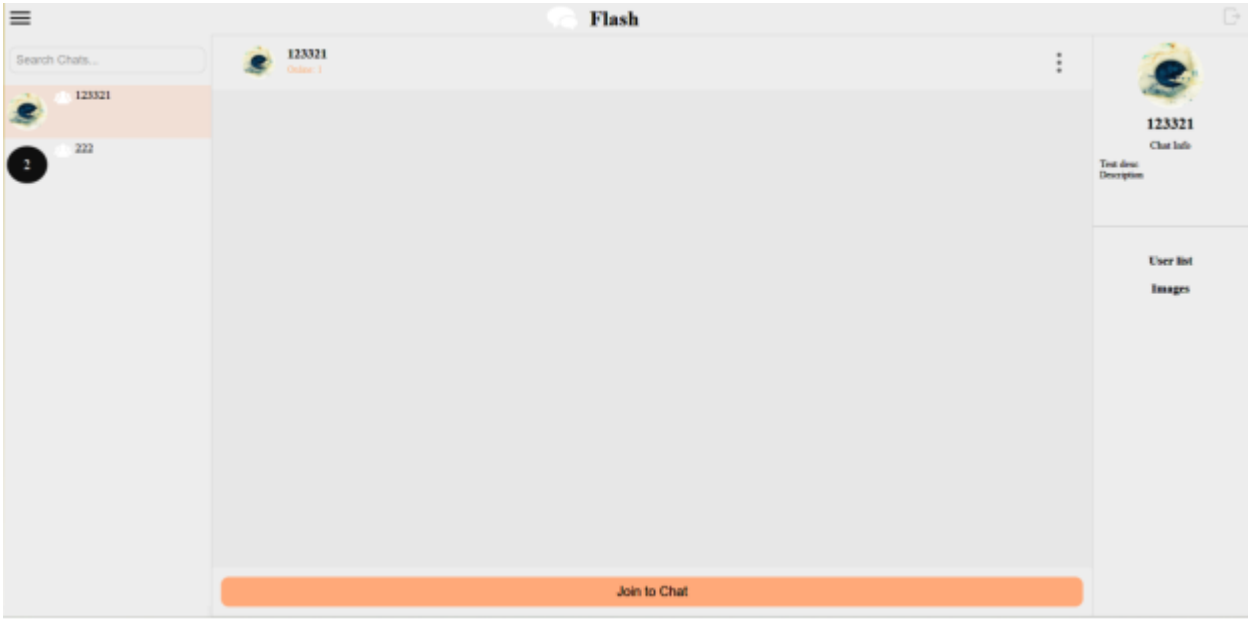


Рисунок 28 – Інтерфейс вікна чату

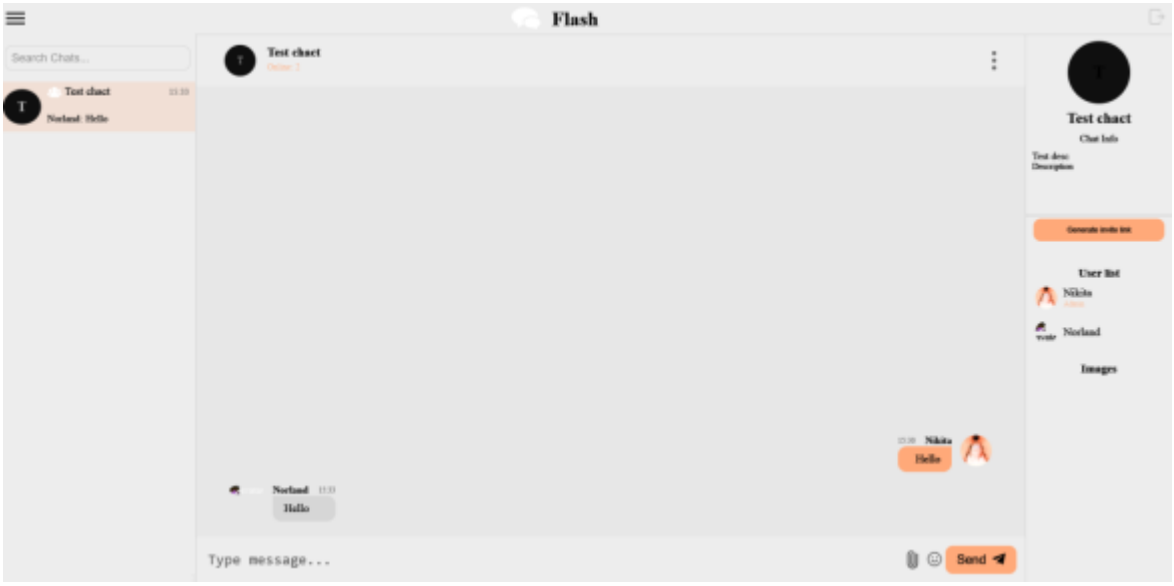


Рисунок 29 – Інтерфейс вікна чату (з повідомленнями)

4 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Розробка та впровадження мікросервісної системи для чату з реальним часом продемонстрували ефективність обраної архітектури та технологічних рішень. Важливо підкреслити, що основна увага була зосереджена на створенні стабільної та продуктивної системи, яка забезпечує безперебійну роботу в умовах значного навантаження. Ретельна перевірка кожного мікросервісу під час ручного тестування дозволила виявити і усунути ключові проблеми ще на етапі розробки. Хоча програмні тести не були написані, система була протестована в інтеграційному режимі, що дало змогу оцінити її продуктивність і функціональність у реальних умовах експлуатації.

Одним із ключових досягнень стало забезпечення високої продуктивності системи навіть під час обробки великої кількості одночасних запитів. Завдяки мікросервісній архітектурі кожен сервіс функціонує автономно, що дозволяє рівномірно розподіляти навантаження між компонентами. Наприклад, Chat-API-Gateway приймає HTTP-запити та делегує їх відповідним мікросервісам через Transport.TCP, що забезпечує мінімальну затримку передачі даних. Тестові сценарії показали, що система успішно обробляє тисячі одночасних запитів від користувачів без помітних збоїв або зниження продуктивності. Це стало можливим завдяки оптимізації процесів комунікації між сервісами та ефективному управлінню ресурсами.

Час обробки кожного повідомлення варіюється від 100 до 600 мс, ці дані приведені на рисунках 30, 31 та 32.

```
Message processing time 0: 124.711ms
Message processing time 1: 117.48ms
Message processing time 4: 110.355ms
Message processing time 2: 110.823ms
Message processing time 3: 111.134ms
Message processing time 7: 111.807ms
Message processing time 6: 112.704ms
Message processing time 8: 113.19ms
Message processing time 9: 113.825ms
Message processing time 15: 114.282ms
Message processing time 5: 121.667ms
Message processing time 14: 122.584ms
Message processing time 29: 130.998ms
Message processing time 20: 134.681ms
Message processing time 31: 137.447ms
Message processing time 10: 140.14ms
Message processing time 18: 142.517ms
```

Рисунок 30 – Час обробки перших повідомлень

```
Message processing time 535: 602.898ms
Message processing time 537: 603.065ms
Message processing time 536: 603.252ms
Message processing time 543: 603.51ms
Message processing time 542: 603.819ms
Message processing time 541: 604.151ms
Message processing time 530: 604.536ms
Message processing time 549: 604.741ms
Message processing time 551: 605.123ms
Message processing time 554: 605.675ms
Message processing time 544: 606.117ms
Message processing time 545: 606.334ms
Message processing time 548: 606.533ms
Message processing time 553: 606.703ms
Message processing time 550: 606.981ms
Message processing time 546: 607.298ms
Message processing time 552: 607.631ms
```

Рисунок 31 – Час обробки повідомлень після половини

```
Message processing time 983: 427.914ms
Message processing time 984: 428.069ms
Message processing time 985: 428.305ms
Message processing time 988: 428.537ms
Message processing time 993: 428.75ms
Message processing time 989: 429.043ms
Message processing time 995: 429.188ms
Message processing time 991: 429.351ms
Message processing time 987: 429.545ms
Message processing time 996: 429.697ms
Message processing time 990: 429.935ms
Message processing time 994: 430.119ms
Message processing time 997: 430.291ms
Message processing time 998: 430.537ms
Message processing time 999: 430.859ms
Message processing time 992: 431.402ms
```

Рисунок 32 – Час обробки повідомлень останніх повідомлень

Як ми можемо побачити, навіть при такому відносно великому навантаженні система залишається робочою і не дає збоїв.

Особливої уваги заслуговує ефективність обробки даних у базі PostgreSQL. Структура бази даних була спроєктована з урахуванням можливого зростання обсягу даних та необхідності забезпечення цілісності інформації. Моделі User, Room і Messages були оптимізовані для швидкого виконання запитів і підтримки транзакцій. У результаті тестових перевірок було встановлено, що система здатна виконувати складні запити, такі як отримання історії повідомлень або пошук за користувачами, з мінімальною затримкою. Завдяки цьому користувачі можуть миттєво отримувати доступ до необхідної інформації, що значно підвищує зручність роботи з чатом.

Для аналізу швидкості виконання запитів до БД було обрано запит на отримання 2000 повідомлень. Швидкість виконання запиту на отримання 2000 повідомлень з бази даних приведено на рисунку 33.

```
2000
Database query time: 105.728ms
2000
Database query time: 112.542ms
2000
Database query time: 108.071ms
2000
Database query time: 112.388ms
2000
Database query time: 111.316ms
2000
Database query time: 110.719ms
2000
Database query time: 204.118ms
```

Рисунок 33 – Час виконання запиту до БД на отримання 2000 повідомлень

Швидкість виконання запиту варіюється від 100 мс до 200 мс, для такої кількості даних це досить гарні результати.

Важливим результатом стала перевірка безпеки системи. Використання JWT для автентифікації користувачів дозволило забезпечити надійний контроль доступу до ресурсів. Кожен запит до мікросервісів супроводжується перевіркою токена, що унеможливлює несанкціонований доступ до конфіденційної інформації.

Окремо слід відзначити ефективність роботи з медіафайлами. Функціонал завантаження та зберігання файлів на хмарному сервері S3 дозволив оптимізувати роботу із великими обсягами медіаконтенту. У ході тестування було перевірено завантаження різних типів файлів, включаючи зображення та відео, і система успішно обробляла ці запити без зниження продуктивності. Це забезпечує зручність обміну медіаконтентом у чаті та підвищує функціональність системи загалом.

Швидкість завантаження невеликих медіафайлів в середньому займає менше 500 мс, це ми можемо побачити на рисунку 34.

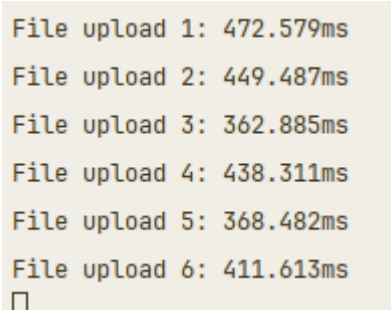


Рисунок 34 – Час завантаження медіафайлів

Ефективність комунікації між мікросервісами через Transport.TCP також підтвердила свою перевагу. Відсутність брокерів повідомлень спростила архітектуру системи, зменшивши накладні витрати на підтримку та моніторинг. Пряме TCP-з'єднання забезпечило швидку передачу даних між мікросервісами, що особливо помітно під час обробки запитів на отримання зображень або повідомлень у реальному часі. Це дозволило досягти низьких затримок і високої стабільності системи навіть за інтенсивного навантаження.

Для демонстрації швидкості з’єднання між мікросервісами наведемо результат виконання запиту на отримання всіх повідомлень на Api-Gateway (див. рис. 35).



Рисунок 35 – Час виконання запиту на отримання всіх повідомлень Api-Gateway

Як ми можемо побачити час виконання не перевищує 200 мс, тож можемо зробити висновок що TCP з’єднання між мікросервісами зовсім не уповільнює роботу системи.

Також було перевірено обробку помилок та несправностей у системі. Завдяки модульній структурі кожен мікросервіс має власні механізми обробки помилок, що дозволяє локалізувати збої та зменшити їх вплив на загальну роботу системи. Наприклад, у разі відмови Chat-Messages-Service, інші сервіси, такі як Chat-Auth-Service або Chat-Room-Service, продовжують функціонувати у штатному режимі. Такий підхід підвищує стійкість системи до відмов і дозволяє швидко відновити роботу окремих компонентів без зупинки всього застосунку. Ця особливість є важливою для мікросервісних архітектур, де надійність системи залежить від стабільності кожного окремого сервісу.

ВИСНОВКИ

1. У процесі виконання кваліфікаційної роботи було детально досліджено сучасні підходи до побудови мікросервісних систем, а також вивчено технології, які забезпечують ефективну комунікацію між сервісами, такі як NestJS та Transport.TCP. Було проведено аналіз архітектурних рішень для застосунків у реальному часі, що дозволило визначити оптимальні методи розробки.

2. Під час розробки системи було реалізовано мікросервісну архітектуру з використанням NestJS для серверної частини та React для клієнтської частини. Кожен мікросервіс виконує окрему функцію, що забезпечує гнучкість, масштабованість та надійність системи.

3. Основна увага приділялася забезпеченню продуктивності та безпеки системи. Для автентифікації користувачів було впроваджено JWT (JSON Web Tokens), що забезпечує захищений доступ до ресурсів. Система також пройшла ручне тестування на вразливості, включаючи перевірку на SQL-ін'єкції, що підтвердило її стійкість до потенційних загроз.

4. Під час дослідження ефективності комунікації між мікросервісами було підтверджено переваги Transport.TCP порівняно з іншими методами обміну даними. Відсутність брокерів повідомлень дозволила спростити архітектуру та зменшити накладні витрати, забезпечивши низькі затримки під час передачі даних між сервісами.

5. У результаті роботи було створено систему чату в реальному часі, яка демонструє високу продуктивність, стабільність і можливість масштабування. Користувачі отримують миттєвий обмін повідомленнями та медіафайлами завдяки інтеграції WebSocket. Усі поставлені завдання були успішно виконані, що підтверджує практичну цінність розробленого застосунку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fowler, M., Lewis, J. "Microservices: a definition of this new architectural term". ThoughtWorks, 2014.

2. Newman, S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.

3. Schumacher, R., Labby, M. The Art of Scalability. Addison-Wesley, 2012.

4. Fisher, B. Docker Deep Dive. Independently Published, 2017.

5. Fielding, R. Architectural Styles and the Design of Network-based Software Architectures. Dissertation, 2000.

6. Nygard, M. Release It!. Pragmatic Bookshelf, 2007.

7. Hope, G., Woolf, B. Enterprise Integration Patterns. Addison-Wesley, 2003.

8. AWS Whitepapers. Serverless Architectures. Amazon Web Services, 2020.

9. arvindpdmn, devbot5S. Inter-Service communication for microservices. Devopedia. URL: https://devopedia.org/inter-service-communication-for-microservices (дата звернення: 24.11.2024).

10. SpeciServe. a grpc infrastructure concept / C. Carthen та ін. 2024 IEEE/ACIS 22nd international conference on software engineering research, management and applications (SERA), м. Honolulu, HI, USA, 30 трав. - 1 черв. 2024 р. 2024. С. 273–276. URL: https://doi.org/10.1109/sera61261.2024.10685565 (дата звернення: 24.11.2024).

11. Galbraith K. 3 methods for microservice communication - LogRocket Blog. LogRocket Blog. URL: https://blog.logrocket.com/methods-for-microservice-communication/ (дата звернення: 24.11.2024).

12. Smith, J., Brown, A. An Analysis of Inter-Service Communication in Microservice Architectures. Proceedings of the IEEE Conference on Software Engineering (2022)

13. Gupta, R. Challenges and Solutions in Microservice Architecture Implementation. ACM Digital Library, 2021.

14. Microservices pattern: microservice architecture pattern. microservices.io. URL: https://microservices.io/patterns/microservices.html (дата звернення: 24.11.2024).

15. Bellemare A. Building event-driven microservices: leveraging organizational data at scale. O'Reilly Media, Incorporated, 2020. 316 с.

16. Davis A. Bootstrapping microservices with docker, kubernetes, and terraform. Manning Publications Co. LLC, 2021.

17. Heroku. The Twelve-Factor App. The Twelve-Factor App. URL: https://12factor.net (дата звернення: 02.12.2024).

18. Designing Data-Intensive Web Applications. Elsevier, 2003. URL: https://doi.org/10.1016/b978-1-55860-843-6.x5000-2 (дата звернення: 02.12.2024).

19. Microservices in practice, part 2: service integration and sustainability / C. Pautasso та ін. IEEE software. 2017. Т. 34, № 2. С. 97–104. URL: https://doi.org/10.1109/ms.2017.56 (дата звернення: 02.12.2024).

20. M viz: visualization of microservices / S. Silva та ін. 2021 25th international conference information visualisation (IV), м. Sydney, Australia, 5–9 лип. 2021 р. 2021. URL: https://doi.org/10.1109/iv53921.2021.00028 (дата звернення: 02.12.2024).

21. Documentation | NestJS. A progressive Node.js framework. Documentation | NestJS. A progressive Node.js framework. URL: https://docs.nestjs.com (дата звернення: 02.12.2024).

22. Delmoro D. Microservices patterns handbook : building testable, scalable, and maintainable microservices: microservices architecture for beginners. Independently Published, 2021.

23. Doglio F. Pro REST API development with node.js. Berkeley, CA : Apress, 2015. URL: https://doi.org/10.1007/978-1-4842-0917-2 (дата звернення: 02.12.2024).

24. Фока Микита магістрант 1 курсу ІННІ ім. Ю. М. Потебні Наук. кер.: доцент, канд. фіз.-мат. наук. Скрипник І.А. Дослідження ефективності та

переваг мікросервісної архітектури у web-застосунках. Збірник наукових праць студентів, аспірантів, докторантів і молодих вчених «МОЛОДА НАУКА-2024», Запоріжжя : Запорізький національний університет, 2024. С. 204-205.

25. Фока М. К., Скрипник І. А. Особливості використання мікросервісної архітектури у web-застосунках Інженерні інновації та розбудова національної економіки : матеріали І Міжнародної науково-практичної конференції Інженерного навчально-наукового інституту ім. Ю. М. Потебні (09–10 травня 2024 року, м. Запоріжжя) / наук. ред. Н. Г. Метеленко ; Інженерний навчально-науковий інститут ім. Ю. М. Потебні Запорізького національного університету. – Одеса : Видавничий дім «Гельветика», 2024. С. 628-629.

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Фока Микита Костянтинович , студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz19bd-15@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Дослідження ефективності та переваг мікросервісної архітектури у Web-застосунках**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 29.11.2024 Підпис _____ Фока Микита Костянтинович
(студент)

Дата 29.11.2024 Підпис _____ Скрипник Ірина Анатоліївна
(науковий керівник)