МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ СОЦІАЛЬНОЇ МЕРЕЖІ З ВИКОРИСТАННЯМ NODE.JS ТА RABBITMQ»**

Виконав: студент 2 курсу, групи 8.1213-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.С. Іванов
(ініціали та прізвище)

Керівник завідувач кафедри програмної інженерії, доцент, к.ф.-м.н. Лісняк А.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Іванову Артему Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка мікросервісної архітектури соціальної мережі з використанням Node.js та RabbitMQ

керівник роботи Лісняк Андрій Олександрович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 27 » вересня 2024 року № 1469-с

2. Строк подання студентом роботи 25.11.2024 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд предметної області.
2. Проєктування застосунку.
3. Реалізація мікросервісної архітектури для соціальної мережі.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

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

7. Дата видачі завдання 01.10.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	04.10.2024	
2.	Збір вихідних даних.	10.10.2024	
3.	Обробка методичних та теоретичних	16.10.2024	
	джерел.		
4.	Розробка першого та другого розділу.	01.11.2024	
5.	Розробка третього розділу.	18.11.2024	
6.	Оформлення та нормоконтроль	20.11.2024	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	11.12.2024	

Студент _____ А.С. Іванов
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.О. Лісняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка мікросервісної архітектури соціальної мережі з використанням Node.js та RabbitMQ»: 52 с., 21 рис., 1 табл.,14 джерел, 2 додатки.

АСИНХРОННА ОБРОБКА, ВІДМОВОСТІЙКІСТЬ, ГОРИЗОНТАЛЬНЕ МАСШТАБУВАННЯ, МІКРОСЕРВІСНА АРХІТЕКТУРА, ОБМІН ПОВІДОМЛЕННЯМИ, ПРОДУКТИВНІСТЬ, СОЦІАЛЬНА МЕРЕЖА, NODE.JS, RABBITMQ.

Об’єкт дослідження – процес розробки соціальної мережі використовуючи мікросервісну архітектуру.

Мета роботи: розробка мікросервісної архітектури соціальної мережі, використовуючи Node.js та RabbitMQ, яка буде відповідати вимогам масштабованості, відмовостійкості та гнучкості.

Метод дослідження – структурний аналіз для розробки модулів системи, моделювання архітектури для відображення взаємодії сервісів, тестування продуктивності для оцінки масштабованості та відмовостійкості.

Основний зміст роботи: У роботі описані сучасні підходи до розробки соціальних мереж та особливості побудови мікросервісних архітектур. Проведено огляд технічних рішень для створення мікросервісів на базі Node.js та застосування системи RabbitMQ для асинхронного обміну повідомленнями між сервісами. Обґрунтовано вибір даних технологій з точки зору їх відповідності вимогам продуктивності та надійності в умовах динамічного навантаження, притаманного соціальним мережам. Розроблено та протестовано ключові компоненти системи, включно з модулями авторизації, управління профілями користувачів. Проведено модульне та інтеграційне тестування для оцінки надійності та продуктивності розробленої системи.

SUMMARY

Master's qualifying paper "Development of a Social Network Microservice Architecture Using Node.js and RabbitMQ": 52 pages, 21 figures, 1 table, 14 references, 2 supplements.

ASYNCHRONOUS PROCESSING, FAULT TOLERANCE, HORIZONTAL SCALING, MICROSERVICES ARCHITECTURE, MESSAGE EXCHANGE, PERFORMANCE, SOCIAL NETWORK, NODE.JS, RABBITMQ.

Object of research: the process of developing a social network using microservices-architecture.

The aim of the work: to develop a microservices architecture for a social network using Node.js and RabbitMQ that meets the requirements for scalability.

Research method: structural analysis for system module development, architecture modeling to reflect service interaction, performance testing to evaluate scalability and fault tolerance.

Main content: this work describes modern approaches to the development of social networks and the peculiarities of building microservices architectures. An overview of technical solutions for creating microservices based on Node.js and using RabbitMQ for asynchronous message exchange between services is provided. The choice of these technologies is justified in terms of their compliance with performance and reliability requirements under the dynamic load typical of social networks. Key system components were developed and tested, including modules for authentication and user profile management. Modular and integration testing was conducted to evaluate the reliability and performance of the developed system.

ЗМІСТ

Завдання на кваліфікаційну роботу..…… 2
Реферат 4
Summary 5
Вступ..... 8
1 Огляд предметної області для соціальної мережі 10
1.1 Опис предметної області 10
1.2 Опис системи..... 10
1.3 Огляд мікросервісної архітектури..... 11
1.4 Технології для розробки мікросервісів..... 13
1.5 Переваги та недоліки мікросервісної архітектури..... 15
1.6 Висновки до розділу 1 17
2 Проєктування застосунку 18
2.1 Використання UML під час розробки системи..... 18
2.2 Діаграма варіантів використання 19
2.2.1 Опис варіантів використання..... 21
2.3 Діаграма діяльності..... 24
2.4 Діаграма послідовності 26
2.5 Діаграма розгортання..... 28
2.6 Діаграма сутностей і зв’язків..... 30
2.7 Висновки до розділу 2 31
3 Реалізація мікросервісної архітектури для соціальної мережі..... 32
3.1 Опис інстурментів розробки для соціальної мережі 32
3.2 Основні сервіси соціальной мережі 33
3.3 Тестування за допомогою Swagger..... 35
3.4 Реалізація бази даних..... 37
3.5 Огляд застосунку..... 39
3.5.1 Реєстрація в системі..... 39

3.5.2 Відправка коментарів 42
3.5.3 Вигляд чату 41
3.6 Висновки до розділу 3 43
Висновки 44
Перелік посилань 45
Додаток А Код сервісу RabbitMQ 46
Додаток Б Код прослуховування черг 51

ВСТУП

Соціальні мережі стали невід’ємною частиною сучасного життя та комунікації. Щоденно мільйони людей у всьому світі спілкуються, діляться думками, новинами та досягненнями через різноманітні платформи. Проте часто користувачі стикаються з обмеженнями існуючих сервісів, які не завжди відповідають їхнім потребам, особливо в контексті масштабованості, швидкості обміну даними та персоналізації.

Сучасні користувачі соціальних мереж прагнуть більш зручних, безпечних та функціональних платформ, які враховують їхні індивідуальні потреби. Запровадження мікросервісної архітектури може забезпечити гнучкість і ефективність у розробці та роботі соціальної мережі. Такий підхід дозволить створити динамічну екосистему, яка сприятиме комфортній взаємодії між користувачами та забезпечить високий рівень надійності.

Соціальна мережа на основі мікросервісної архітектури дозволить розробникам легко масштабувати систему, інтегрувати нові функції та адаптувати платформу до сучасних вимог. Використання RabbitMQ забезпечить надійний обмін повідомленнями між мікросервісами, що значно покращить продуктивність і стабільність системи.

Виходячи з цього, було вирішено розробити соціальну мережу, яка базуватиметься на мікросервісній архітектурі та використовуватиме RabbitMQ для обміну повідомленнями. Це дозволить забезпечити гнучкість, масштабованість і високу продуктивність платформи.

Актуальність дослідження: актуальність теми зумовлена необхідністю створення ефективних і масштабованих соціальних мереж із використанням сучасних підходів до архітектури систем.

Мета: розробити соціальну мережу на основі мікросервісної архітектури з використанням RabbitMQ.

Задачі:

– сформувати вимоги до соціальної мережі;
– спроєктувати архітектуру мікросервісів;
– реалізувати систему обміну повідомленнями між мікросервісами за допомогою RabbitMQ;
– протестувати роботу соціальної мережі та її компонентів.

Об’єкт дослідження: процес розробки соціальної мережі на основі мікросервісної архітектури.

Предмет дослідження: використання RabbitMQ для забезпечення взаємодії між мікросервісами.

Методи дослідження: моделювання, проєктування, програмний, аналітичний.

У першому розділі проведено аналіз вимог до соціальної мережі, здійснено огляд подібних систем та визначено функціональність, яку буде реалізовано.

Другий розділ присвячено проєктуванню мікросервісної архітектури, детальному опису прецедентів використання та побудові відповідних діаграм.

Третій розділ охоплює процес реалізації системи, її тестування та включає опис функціоналу для користувачів, який демонструє роботу соціальної мережі з використанням RabbitMQ.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ДЛЯ СОЦІАЛЬНОЇ МЕРЕЖІ

1.1 Опис предметної області

Предметною областю є розробка мікросервісної архітектури для соціальної мережі. Ця система повинна надавати користувачам можливість спілкування, створення та обміну контентом, а також взаємодії з іншими користувачами.

Процес взаємодії з системою залежить від типу користувача. Гості можуть лише переглядати загальнодоступний контент та реєструвати профіль. Зареєстровані користувачі отримують розширений доступ: створення постів, коментарів, обмін повідомленнями, лайки та підписки.

Процес створення посту відбувається наступним чином. Користувач реєструється або авторизується в системі, потім натискає на кнопку «Створити пост». Система відкриває інтерфейс для створення посту.

Користувач вводить текст або додає мультимедійний контент (зображення, відео) та натискає кнопку «Опублікувати». Система перевіряє вміст на відповідність політикам платформи, зберігає пост в базі даних і відображає його в стрічці новин.

Після публікації пост стає доступним іншим користувачам відповідно до налаштувань конфіденційності. Користувачі можуть взаємодіяти з постом: лайкати, коментувати чи ділитися ним.

1.2 Опис системи

На сьогодні соціальні мережі залишаються потужним інструментом комунікації та взаємодії. Щодня мільйони користувачів обмінюються інформацією, контентом і думками. Зростаюча потреба в спеціалізованих

платформах для обміну різноплановим контентом вимагає створення інноваційних рішень.

Наша система покликана задовольнити потреби користувачів у зручному середовищі для створення, обміну та взаємодії з контентом. Вона надає можливість переглядати стрічку новин, спілкуватися через повідомлення, підписуватися на профілі інших користувачів, створювати пости з текстом або мультимедіа, редагувати чи видаляти їх, а також ділитися контентом.

У перспективі планується впровадження монетизації контенту, організації тематичних спільнот та покращених інструментів для персоналізації. Розширені можливості допоможуть платформі задовольнити різноманітні потреби аудиторії, забезпечуючи зручний і безпечний обмін контентом в межах спеціалізованої соціальної мережі.

1.3 Огляд мікросервісної архітектури

Мікросервісна архітектура – це підхід, що став особливо популярним у сучасній розробці програмного забезпечення, завдяки його здатності вирішувати обмеження, властиві традиційним монолітним системам [9]. У монолітній архітектурі вся логіка програми реалізується в одному великому додатку, що робить систему менш гнучкою, менш масштабованою та складною у підтримці. Натомість мікросервіси розбивають додаток на набір окремих, незалежних сервісів, кожен з яких виконує конкретну задачу. Це дозволяє досягти високої модульності, яка має критичне значення для сучасних великих систем, таких як соціальні мережі, маркетплейси та фінансові платформи.

Кожен мікросервіс в такій архітектурі має власний незалежний цикл розробки, може бути розроблений окремою командою розробників та розгорнутий у продакшн, не впливаючи на роботу інших частин системи. Завдяки цьому мікросервісний підхід дає змогу масштабувати команду розробки та підвищити ефективність роботи, адже кожна команда може

працювати незалежно над своїм компонентом, використовуючи найбільш відповідну для нього технологію, мовну платформу, інструменти або базу даних.

Щоб забезпечити злагоджену роботу всіх мікросервісів, важливо налаштувати коректну комунікацію між ними [13]. У більшості випадків для цього використовують протоколи HTTP/REST або, у випадках, де потрібна висока надійність і стійкість до збоїв, застосовують черги повідомлень, такі як RabbitMQ або Apache Kafka. Наприклад, коли один мікросервіс, відповідальний за аутентифікацію, потребує передати дані до сервісу управління профілями, він відправляє запит через REST API або повідомлення в чергу. RabbitMQ в цьому випадку служить для обробки та розподілу повідомлень, що дозволяє зменшити навантаження на основні сервіси і забезпечити їх безперебійну роботу навіть при великих обсягах запитів.

Однією з важливих переваг мікросервісної архітектури є можливість безперервного розгортання (CI/CD), що значно пришвидшує впровадження нових функцій. Це означає, що новий функціонал або оновлення можна додати до конкретного мікросервісу, не зупиняючи всю систему. Таким чином, ризик помилки мінімізується, і навіть при збої в роботі одного мікросервісу інші залишаються активними, що забезпечує відмовостійкість.

До недоліків мікросервісної архітектури можна віднести складність налаштування та управління всією системою. Велика кількість мікросервісів потребує надійної системи моніторингу для забезпечення працездатності кожного компонента. Відповідно, потрібні ефективні інструменти для автоматизації розгортання, контролю продуктивності, а також забезпечення високого рівня безпеки при передачі даних між сервісами. З часом обсяг взаємодій між мікросервісами може збільшуватись, що потребує ретельного підходу до організації взаємозв’язків і управління інфраструктурою.

Окрім основних переваг мікросервісної архітектури, таких як масштабованість, гнучкість і відмовостійкість, слід зазначити, що вона також сприяє кращій адаптивності системи до потреб бізнесу. Наприклад, якщо

вимоги до певної функції змінюються, відповідний мікросервіс можна швидко оновити або навіть переписати з використанням іншої мови програмування. Це дозволяє легко впроваджувати нові бізнес-моделі, додавати нові функції чи змінювати логіку роботи без необхідності глобальних змін у всій системі.

Таким чином, мікросервісна архітектура є сучасним, потужним інструментом для створення гнучких, масштабованих і відмовостійких систем, здатних витримувати навантаження сучасних динамічних ринків і великих користувацьких баз. Її реалізація вимагає ретельного планування та підбору відповідних інструментів для забезпечення коректної роботи всіх компонентів, однак надає значні переваги для розвитку великомасштабних систем, таких як соціальні мережі.

1.4 Технології для розробки мікросервісів

Node.js та RabbitMQ – це два ключові інструменти, які забезпечують надійність, продуктивність і відмовостійкість мікросервісної архітектури соціальної мережі. Використання Node.js як основної платформи для серверної частини обумовлено його здатністю ефективно обробляти одночасні запити в асинхронному режимі, що є критичним для систем з великою кількістю активних користувачів. Node.js працює на основі подійно-орієнтованої моделі, що дозволяє обробляти запити без блокування процесу, завдяки чому сервер залишається доступним для нових запитів навіть при великих навантаженнях [7]. Це робить Node.js ідеальним для систем, де потрібне швидке реагування і мінімальні затримки, наприклад, для миттєвого відображення оновлень у стрічці новин чи обміну повідомленнями між користувачами.

Node.js також має великий екосистемний ресурс у вигляді бібліотек і фреймворків, таких як Express.js, що значно полегшують розробку REST API, необхідних для взаємодії між мікросервісами. Інші бібліотеки, наприклад, для роботи з базами даних чи кешування даних, інтегруються з Node.js без зайвих складнощів, дозволяючи прискорити процес розробки. Додатково,

інструменти для тестування і розгортання, такі як Mocha для модульного тестування чи PM2 для управління процесами, також оптимізовані для роботи з Node.js, що сприяє створенню надійних і швидкодіючих мікросервісів.

RabbitMQ, в свою чергу, забезпечує ефективний асинхронний обмін даними між мікросервісами [1]. Використання черг RabbitMQ дозволяє налаштовувати маршрутизацію повідомлень між сервісами таким чином, що навіть при збоях чи пікових навантаженнях повідомлення не губляться. Це особливо важливо в соціальних мережах, де користувачі можуть надсилати повідомлення, оновлювати статуси та інші дані, які потребують швидкої та безпечної обробки. RabbitMQ не тільки підтримує черги повідомлень, але й дозволяє налаштовувати підтвердження доставки, що гарантує, що повідомлення успішно надійшли і були оброблені.

Крім основної функції передачі даних, RabbitMQ також дозволяє інтегрувати різні шаблони обміну повідомленнями, такі як «публікація/підписка» або «робочі черги», що є надзвичайно корисним для соціальних мереж. Наприклад, якщо мікросервіс обробки новин оновлює дані про нові дописи, інші мікросервіси, такі як сервіс сповіщень або аналітики, можуть підписатися на оновлення і автоматично отримувати актуальну інформацію для подальшої обробки. Це дозволяє створити складні сценарії взаємодії між сервісами, які працюють без зайвих затримок.

RabbitMQ також має розвинені інструменти моніторингу, які дозволяють відстежувати стан черг, кількість повідомлень, що очікують обробки, та інші ключові метрики, що є важливими для відстеження продуктивності системи. Це сприяє забезпеченню надійної роботи соціальної мережі в реальному часі і дозволяє миттєво реагувати на можливі збої або навантаження.

Загалом, поєднання Node.js і RabbitMQ дає змогу створити сучасну, високо масштабовану систему, здатну витримувати великі навантаження, забезпечуючи швидкий і надійний обмін даними між компонентами [10]. Це рішення сприяє побудові систем, які можуть рости та розширюватись залежно від потреб бізнесу та кількості користувачів, залишаючись стійкими до збоїв і забезпечуючи високу якість сервісу.

1.5 Переваги та недоліки мікросервісної архітектури

Мікросервісна архітектура пропонує великий набір переваг, особливо для розробки соціальних мереж та інших складних, масштабованих систем. Основною перевагою є можливість масштабування, адже кожен мікросервіс може бути масштабований окремо, що дозволяє оптимально використовувати ресурси. Це означає, що сервіси, які піддаються більшому навантаженню, наприклад, сервіс аутентифікації або обробки медіафайлів, можуть мати більше ресурсів без необхідності масштабування всієї системи. Такий підхід дозволяє значно підвищити ефективність та уникати ситуацій, коли система надмірно ускладнюється через непотрібні оновлення в усіх сервісах одночасно.

Гнучкість мікросервісної архітектури також дозволяє командам розробників вибирати найбільш підходящі технології та інструменти для кожного сервісу окремо. Наприклад, для сервісу обробки даних можна використовувати Python, завдяки його широким можливостям в аналізі даних, а для сервісу обробки запитів від користувачів – Node.js, завдяки його асинхронній обробці. Це дозволяє створювати більш продуктивну, швидку та економну систему, що відповідає вимогам сучасних користувачів.

Асинхронний обмін повідомленнями між мікросервісами забезпечує високу продуктивність, адже сервіси можуть працювати незалежно і не чекати завершення роботи інших компонентів. RabbitMQ, наприклад, дозволяє організувати обробку запитів, коли один сервіс може надіслати повідомлення іншому сервісу для обробки, що дає змогу ефективніше розподіляти робоче навантаження. Завдяки асинхронній обробці даних система може швидше реагувати на запити користувачів, що є критично важливим для соціальних мереж, де швидкість реакції напряму впливає на задоволення користувачів.

Однак, незважаючи на всі переваги, мікросервісна архітектура має і свої недоліки. Одним з основних є підвищена складність управління великою кількістю сервісів, що вимагає детального моніторингу та аудиту. У мікросервісній архітектурі кожен сервіс має власну базу даних, логіку і

залежності, що вимагає складної системи управління і взаємодії. Щоб ефективно управляти такою системою, потрібні спеціалізовані інструменти для моніторингу, такі як Prometheus чи Grafana, а також системи оркестрації, як Kubernetes, для контейнеризації та управління ресурсами. Це може створювати додаткове навантаження на команду розробників і адміністраторів, що може вплинути на вартість і час реалізації проєкту.

Необхідність наявності висококваліфікованих розробників є ще одним недоліком. Взаємодія між мікросервісами має бути чітко організованою і структурованою, щоб уникнути конфліктів даних і забезпечити належний обмін інформацією між компонентами системи. Наприклад, для реалізації безпечного обміну даними між сервісами можуть використовуватися JWT-токени або інші засоби аутентифікації, що потребує знань в області безпеки [4].

Додатково, мікросервісна архітектура вимагає більшої уваги до тестування та процесу деплою, оскільки кожен мікросервіс розгортається окремо, що створює потенційну складність у підтримці цілісності системи [6]. Тестування мікросервісів потребує ретельної перевірки кожного окремого компонента, а також їхньої взаємодії між собою. Для цього використовують спеціальні підходи до тестування, такі як інтеграційне і контрактне тестування, що дозволяє переконатися у правильній роботі взаємозалежних сервісів.

У таблиці 1.1 наведено порівняння брокерів повідомлень.

Таблиця 1.1 – Порівняння брокерів повідомлень

Характеристика	**Amazon SQS**	**Apache Kafka**	**RabbitMQ**
Тип архітектури	Хмарний сервіс	Розподілена платформа потокових даних	Розподілений брокер повідомлень
Простота використання	Висока	Середня, потребує додаткових налаштувань	Середня, з необхідністю налаштувань
Масштабованість	Автоматичне масштабування	Висока, але потребує додаткових налаштувань	Висока при належній конфігурації

Продовження таблиці 1.1

Характеристика	Amazon SQS	Apache Kafka	RabbitMQ
Продуктивність	Середня	Висока, оптимальна для потокових даних	Середня, з гнучкими механізмами обробки
Гарантії доставки	Висока (з підтвердженням отримання)	Висока, з можливістю відновлення	Висока, з підтвердженням отримання

Незважаючи на ці складнощі, мікросервісна архітектура є оптимальним вибором для масштабованих систем, як-от соціальні мережі, оскільки вона забезпечує баланс між продуктивністю, стійкістю до збоїв та гнучкістю розвитку проєкту.

1.6 Висновки до розділу 1

У результаті розробки мікросервісної архітектури для соціальної мережі можна зробити кілька ключових висновків щодо її ефективності та потенціалу. Мікросервісний підхід дозволяє досягти високої масштабованості, гнучкості та відмовостійкості, що є критичними вимогами для соціальних мереж з великою кількістю користувачів і високим навантаженням.

Інтеграція з RabbitMQ для обміну повідомленнями між мікросервісами дозволяє забезпечити асинхронний процес обробки запитів, що позитивно впливає на ефективність взаємодії між сервісами і знижує ймовірність виникнення збоїв у системі.

Таким чином, мікросервісна архітектура є перспективним рішенням для розробки соціальних мереж, але вимагає ретельного планування та впровадження з урахуванням технічних та організаційних складнощів. У разі правильного підходу вона здатна забезпечити високу ефективність, масштабованість та стійкість системи.

2 ПРОЄКТУВАННЯ ЗАСТОСУНКУ

2.1 Використання UML під час розробки системи

UML (Unified Modeling Language) став одним із ключових інструментів на етапі проєктування соціальної мережі з мікросервісною архітектурою [14]. Його використання дозволило створити зрозумілу та чітку модель системи, що значно полегшило процес розробки, забезпечило узгодженість між командою розробників і зменшило ризики помилок.

Крім того, UML був корисним для проєктування внутрішньої структури сервісів. Діаграми класів описували структури даних, такі як користувачі, пости, коментарі чи повідомлення, а також відображали відносини між об’єктами, включаючи зв’язки типу «один до одного», «один до багатьох» та «багато до багатьох». Це забезпечило чітке розуміння моделей, які необхідно було реалізувати у базі даних MySQL з використанням ORM-бібліотеки Sequelize.

Ще однією важливою частиною було створення діаграм розгортання, які показали фізичну архітектуру системи. Це включало відображення серверів, на яких розміщені окремі сервіси, базу даних MySQL, брокер повідомлень RabbitMQ та клієнтські додатки. Такі діаграми допомогли визначити, як компоненти взаємодіють у реальному середовищі, що було особливо важливо для забезпечення масштабованості та надійності системи.

UML також відіграв важливу роль у комунікації між учасниками команди. Завдяки створеним діаграмам розробники, тестувальники, бізнес-аналітики та інші члени команди могли швидко зрозуміти, як працює система. Це сприяло злагодженій роботі команди та допомогло уникнути непорозумінь під час реалізації проєкту.

Таким чином, UML забезпечив структурований та інтегрований підхід до розробки системи, дозволив спростити складні концепції і створити якісну

документацію. Його використання було ключовим фактором, що сприяв успішній реалізації соціальної мережі на основі мікросервісної архітектури.

2.2 Діаграма варіантів використання

Діаграма варіантів використання (use case diagram) є одним з ключових елементів UML, що служить для візуалізації та документування функціональних вимог системи з точки зору користувачів і акторів, які з нею взаємодіють. Ця діаграма дозволяє зрозуміти основні функції системи та взаємозв’язки між ними.

На діаграмі варіантів використання зображуються актори (користувачі, інші системи або зовнішні сутності, які взаємодіють із системою) та варіанти використання (дії чи послідовності дій, які система виконує для надання певної функціональності). Варіанти використання представлені овальними фігурами, а актори – фігурками людини або іншими графічними елементами. Діаграма варіантів використання є корисним інструментом на ранніх етапах розробки програмного забезпечення, оскільки вона допомагає зрозуміти та узгодити вимоги між усіма зацікавленими сторонами. Однак вона не замінює детальнішу документацію вимог і технічних специфікацій, а є візуальним доповненням для кращого розуміння загальної картини системи.

На рисунку 2.1 представлена загальна діаграма варіантів використання додатку для соціальної мережі.

На цій діаграмі показані актори «Гість» та «Користувач», кожен з яких може взаємодіяти з системою на різних рівнях. Актор «Користувач» включає в себе варіанти використання актора «Гість».

Виділено основний варіант використання – «Створення посту». Після того як користувач переходить до розділу «Створити пост», система відображає інтерфейс для завантаження контенту. Користувач може додати текстовий або мультимедійний контент (фото, відео) та натискати на кнопку

«Опублікувати». Після цього система зберігає дані в базі даних і відображає пост у стрічці новин.

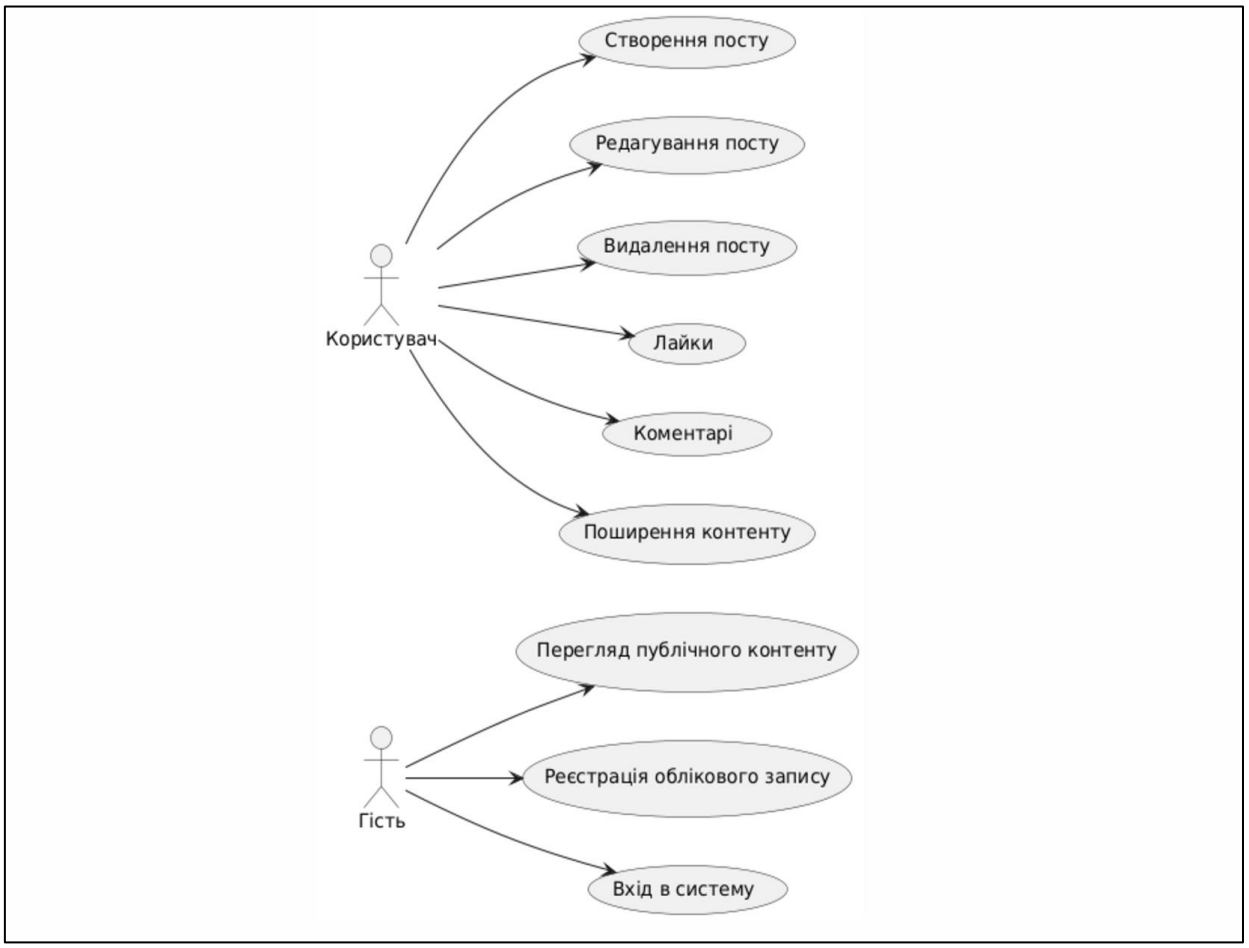


Рисунок 2.1 – Діаграма варіантів використання

На рисунку 2.2 показано, які функції доступні для актора «Гість», зокрема перегляд публічного контенту, реєстрація облікового запису та вхід в систему.

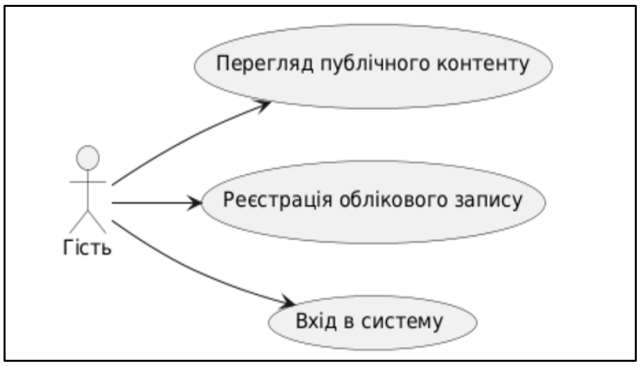


Рисунок 2.2 – Діаграма варіантів використання «Гість»

Діаграма на рисунку 2.3 відображає варіанти використання для актора «Користувач», включаючи створення постів, редагування, видалення, а також взаємодію з контентом через лайки, коментарі та поширення.

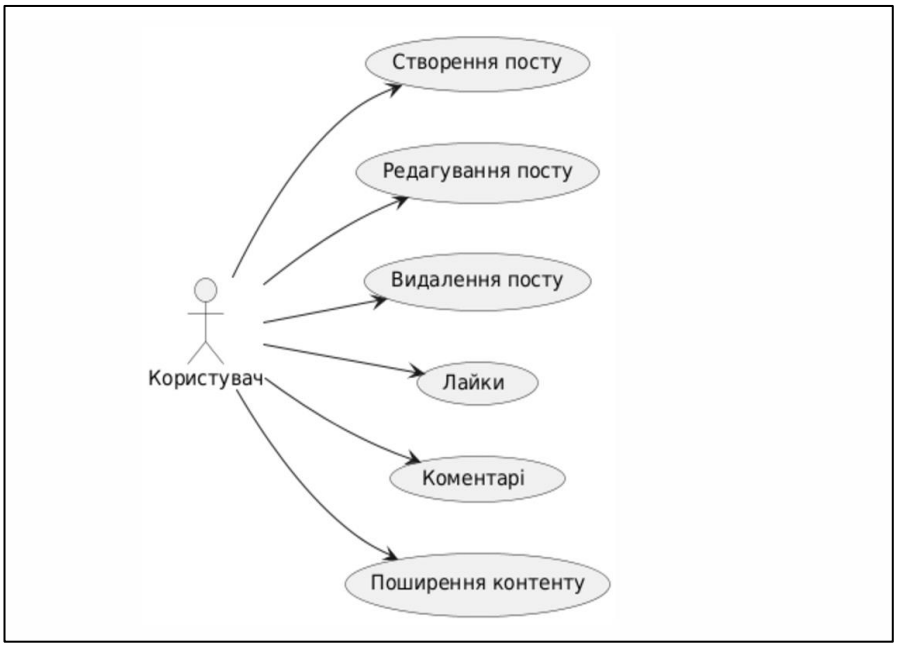


Рисунок 2.3 – Діаграма варіантів використання «Користувач»

Кожен варіант використання відображає певну послідовність кроків або дій, які актор може виконати для досягнення поставленої мети. Ці сценарії відображають способи застосування функціоналу системи, що забезпечують моделювання та документування вимог до поведінки системи. Варіанти використання допомагають візуалізувати різні сценарії взаємодії користувачів та інших акторів з системою, що забезпечує зрозуміле і точне визначення функціональних вимог.

2.2.1 Опис варіантів використання

Прецедент «Реєстрація в системі».

Призначення: даний варіант використання надає можливість гостю зареєструватися в системі.

Основний потік подій: даний варіант використання починає виконуватися, коли гість натискає на пункт меню «Login/Register» та обирає пункт «Register». Система відображає форму реєстрації. Після введення необхідних даних, гість натискає на кнопку «Submit», система створює нового користувач та зберігає його параметри.

Виняткова ситуація 1: невалідні дані – система відображає повідомлення про помилку, гість може повторно ввести дані.

Виняткова ситуація 2: такий користувач вже зареєстрований – система відображає повідомлення про помилку, гість може змінити дані.

Виняткова ситуація 3: помилка сервера – система відображає відповідне повідомлення.

Прецедент «Вхід до системи».

Призначення: даний варіант використання надає можливість гостю входити до системи.

Основний потік подій: даний варіант використання починає виконуватися, коли гість натискає на пункт меню «Login/Register» та обирає пункт «Login». Система відображає форму авторизації. Після введення пошти та паролю, гість натискає на кнопку «Submit», система відображає додаткові функції та інструментарій.

Передумова: перед початком виконання даного варіанта використання гість повинен зареєструватися в системі.

Виняткова ситуація 1: невалідні дані – система відображає повідомлення про помилку, гість може повторно ввести дані.

Виняткова ситуація 2: такого користувача не існує – система відображає повідомлення про помилку, гість може змінити дані.

Виняткова ситуація 3: помилка сервера – система відображає відповідне повідомлення.

Прецедент «Вихід з системи».

Призначення: даний варіант використання надає можливість користувачу виходити з системи.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає на пункт меню «Logout». Система видаляє дані про сесію користувача та відображає головну сторінку.

Прецедент «Створення поста».

Призначення: даний варіант використання надає можливість користувачу створювати новий пост.

Основний потік подій: процес починається, коли користувач натискає кнопку «Create Post». Система відкриває форму для створення поста, де користувач вводить необхідні дані, такі як заголовок, текст або додає медіафайли. Після заповнення форми користувач натискає кнопку «Submit», і система перевіряє коректність введених даних. Успішно створений пост додається до стрічки новин. У разі помилок у даних або технічного збою система відображає відповідне повідомлення, дозволяючи повторити дію.

Виняткові ситуація 1: невалідні дані – система відображає повідомлення про помилку, користувач може виправити дані та повторити спробу.

Виняткові ситуація 2: помилка сервера – система відображає відповідне повідомлення.

Прецедент «Написання коментаря».

Призначення: даний варіант використання надає можливість користувачу залишати коментарі до постів.

Основний потік подій: коли користувач відкриває пост і вводить текст коментаря, система відображає поле для підтвердження дії. Після натискання кнопки «Submit» система перевіряє валідність тексту, додає коментар до списку та робить його видимим для інших. Якщо виникає проблема, пов’язана з помилками у введенні тексту чи збоями сервера, система інформує користувача про необхідність повторної спроби.

Виняткові ситуація 1: невалідні дані – система відображає повідомлення про помилку, користувач може виправити текст коментаря та повторити спробу.

Виняткові ситуація 2: помилка сервера – система відображає відповідне повідомлення.

Прецедент «Відправка повідомлення».

Призначення: користувач може надсилати приватні повідомлення іншим користувачам.

Основний потік подій: відправка повідомлення починається з відкриття чату зі співрозмовником. Після введення тексту в полі вводу користувач натискає кнопку «Send», що запускає перевірку даних. Система додає повідомлення до історії чату та передає його співрозмовнику. Якщо користувач намагається відправити порожнє повідомлення, звертається до невідомого співрозмовника чи виникає технічна помилка, система інформує про це, надаючи можливість виправити ситуацію.

Виняткові ситуація 1: невалідні дані – система відображає повідомлення про помилку, користувач може виправити текст повідомлення та повторити спробу.

Виняткові ситуація 2: помилка сервера – система відображає відповідне повідомлення.

2.3 Діаграма діяльності

Діаграма діяльності (Activity Diagram) є одним із основних інструментів UML, який використовується для моделювання динамічних аспектів системи. Вона дозволяє описати послідовність виконання дій у процесах, показуючи, як користувачі або компоненти системи взаємодіють із нею для досягнення певних цілей. У розробці соціальної мережі діаграма діяльності відіграла важливу роль у візуалізації ключових бізнес-процесів і сценаріїв взаємодії користувачів із системою (див. рис. 2.4).

Одним із прикладів застосування діаграми діяльності є опис процесу реєстрації користувача. Діаграма відображає, як користувач заповнює форму

реєстрації, система перевіряє правильність введених даних, зберігає їх у базу даних і повертає підтвердження успішної реєстрації. Це допомагає зрозуміти всі етапи процесу та виявити можливі точки відмови, наприклад, неправильне заповнення форми або технічні збої.

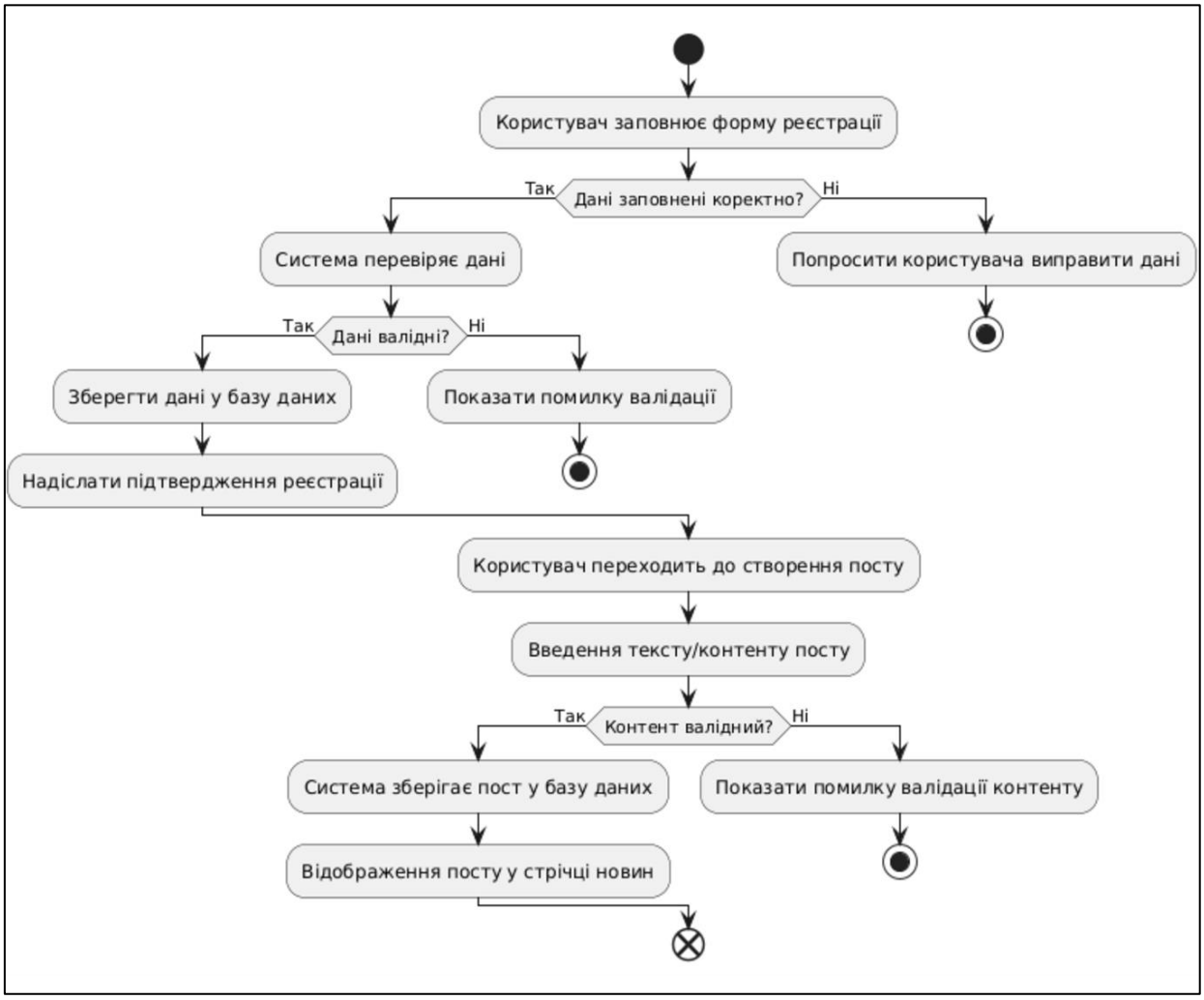


Рисунок 2.4 – Діаграма діяльності

Ще одним важливим сценарієм, змодельованим за допомогою діаграми діяльності, є процес відправки повідомлення в чаті. Цей процес включає кілька послідовних етапів: користувач надсилає текст повідомлення, система передає його через WebSocket до сервера, сервер перевіряє отримані дані, зберігає повідомлення в базу даних і надсилає його одержувачу. Діаграма діяльності дозволила детально описати всі ці кроки, забезпечуючи їх коректну реалізацію.

Окрім моделювання взаємодії користувачів із системою, діаграма діяльності виявилася корисною для опису внутрішніх процесів у мікросервісній архітектурі. Наприклад, під час обробки повідомлень у RabbitMQ, діаграма діяльності показувала, як повідомлення потрапляє до черги, обробляється споживачем і передається далі до відповідного мікросервісу.

Таким чином, використання діаграми діяльності дозволило отримати візуальне уявлення про основні процеси в системі, що значно полегшило їх проєктування, реалізацію та тестування. Це сприяло підвищенню ефективності розробки та забезпеченню високої якості функціональності соціальної мережі.

2.4 Діаграма послідовності

Діаграма послідовності (Sequence Diagram) в UML є важливим інструментом для візуалізації динамічних аспектів системи, зокрема, для зображення порядку взаємодії об’єктів чи компонентів у часі. Вона дозволяє проаналізувати, як різні елементи системи обмінюються повідомленнями, виконуючи певні функціональні операції або сценарії використання. Ці діаграми дають змогу детально уявити хронологічний процес, в якому об’єкти або компоненти взаємодіють для досягнення загальної мети.

На діаграмі послідовності об’єкти, що беруть участь у взаємодії, зображуються прямокутниками, що розташовуються горизонтально в верхній частині діаграми. Кожен об’єкт має свою лінію життя – вертикальну лінію, що символізує період існування об’єкта під час взаємодії. Стрілки між лініями життя представляють повідомлення, які передаються від одного об’єкта до іншого, і вказують на напрямок та тип взаємодії (див. рис. 2.5).

Порядок обміну повідомленнями – чітко показується, як повідомлення передаються між об’єктами в хронологічному порядку, що дозволяє зрозуміти логіку взаємодії.

Ролі об’єктів – діаграма надає змогу виявити, які об’єкти чи компоненти беруть участь у взаємодії, а також які їхні функції та обов’язки в процесі.

Створення та знищення об’єктів – на діаграмі можна побачити, коли саме об’єкти створюються та знищуються в ході виконання взаємодії, що є важливою частиною для коректної моделі системи.

Часові відношення – завдяки розташуванню елементів діаграми можна легко простежити відношення між різними повідомленнями по часу.

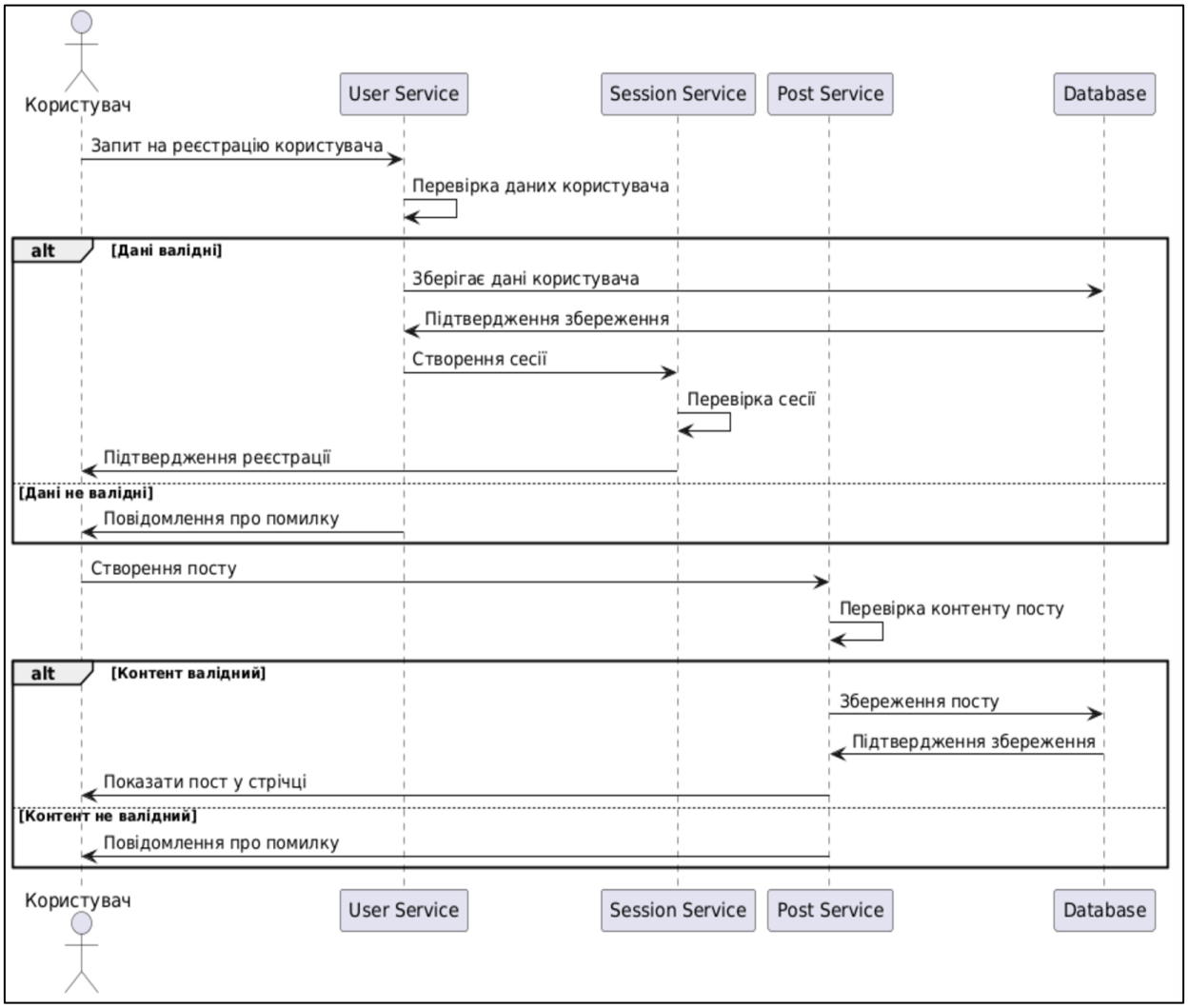


Рисунок 2.5 – Діаграма послідовності

Діаграми послідовності є потужним інструментом для опису складних взаємодій між об’єктами системи, особливо в об’єктно-орієнтованому програмуванні, де взаємодії між об’єктами займають важливе місце. Вони дають змогу чітко описати порядок виконання операцій, а також полегшують

розуміння того, як об’єкти та компоненти взаємодіють для реалізації конкретного функціоналу.

Ці діаграми особливо корисні на етапах проєктування програмного забезпечення, коли необхідно чітко задокументувати взаємодію між різними частинами системи. Вони допомагають розробникам зрозуміти, як реалізується логіка процесів у часі, а також виявити можливі помилки або неточності в проєкті на ранніх етапах розробки.

2.5 Діаграма розгортання

Діаграма розгортання (Deployment Diagram) в UML є важливим інструментом для візуалізації фізичної архітектури програмних систем. Вона дає змогу відобразити, як програмні компоненти розміщуються на різних фізичних вузлах або апаратних пристроях, а також показує взаємодії між ними. Основною метою діаграми є створення карти того, як різні частини системи будуть працювати разом в реальному світі, на конкретних фізичних пристроях, таких як сервери, робочі станції чи мобільні пристрої.

Ця діаграма дає змогу зрозуміти, як система буде розгорнута в реальній інфраструктурі, а також які ресурси будуть необхідні для її функціонування. Вона вказує на зв’язки між програмними компонентами (наприклад, сервісами, базами даних або вебдодатками) та фізичними вузлами, на яких ці компоненти будуть працювати. За допомогою діаграми розгортання можна не лише спланувати ефективне використання ресурсів, а й виявити потенційні проблеми з масштабуванням та ефективністю системи.

Розглянемо основні елементи діаграми розгортання (рис. 2.6).

Вузол (Node) – фізичний ресурс, на якому розміщуються програмні компоненти. Це може бути сервер, робоча станція, мобільний пристрій або навіть хмарний сервіс. Вузли зображуються у вигляді 3D-кубиків або паралелепіпедів, що представляють фізичні машини чи віртуальні машини.

Компонент (Component) – це програмний артефакт, який може бути виконуваним файлом, бібліотекою, вебсервісом чи іншим програмним елементом. Компоненти зображуються прямокутниками з вкладеними символами, що вказують на різні типи програмного забезпечення або обробки.

З’єднання (Connector) – це лінії або стрілки, що з’єднують компоненти або вузли і показують шляхи комунікації між ними. Це може бути фізичне з’єднання (кабель) або логічне з’єднання (мережеві з’єднання або API). З’єднання може відображати різні типи протоколів або канали зв’язку між компонентами.

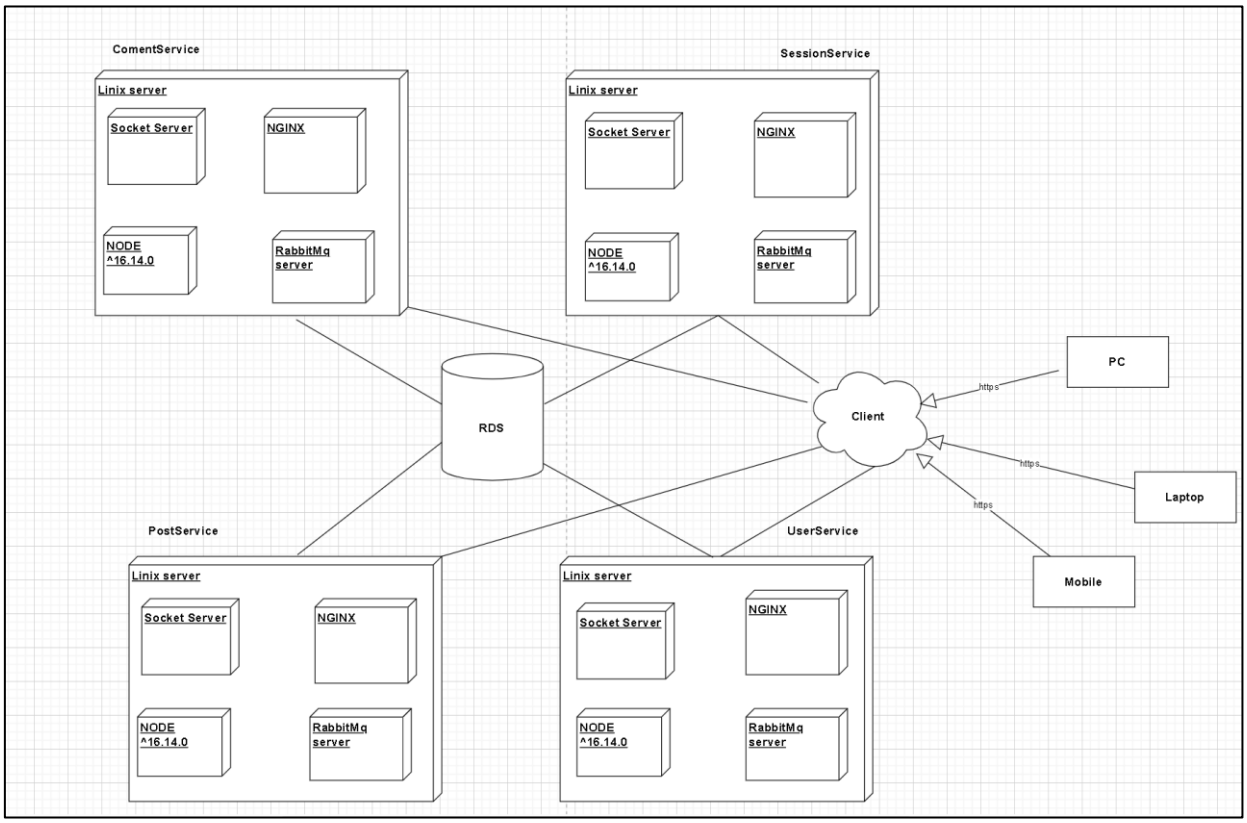


Рисунок 2.6 – Діаграма розгортання

Діаграма розгортання зазвичай використовується разом з іншими діаграмами UML, такими як діаграми компонентів, діаграми пакетів і діаграми класів, для створення повної карти архітектури системи. Вона допомагає отримати загальне уявлення про те, як програмні компоненти будуть працювати разом на фізичних пристроях, що важливо для тестування,

планування масштабування та підтримки на всіх етапах життєвого циклу системи.

Це особливо корисно для складних розподілених систем, де важливо не лише правильно спроєктувати програмну частину, а й гарантувати, що компоненти будуть ефективно працювати на фізичних вузлах, а також можуть бути легко масштабовані для задоволення зростаючих вимог.

2.6 Діаграма сутностей і зв’язків

Діаграма сутностей і зв’язків (ER-діаграма) використовується для моделювання структури бази даних (див. рис. 2.7). Вона дозволяє наочно представити сутності системи, їхні атрибути та зв’язки між ними. Даний підхід допомагає зрозуміти логіку взаємодії між даними, визначити первинні та зовнішні ключі, а також спроєктувати ефективну структуру для зберігання інформації.

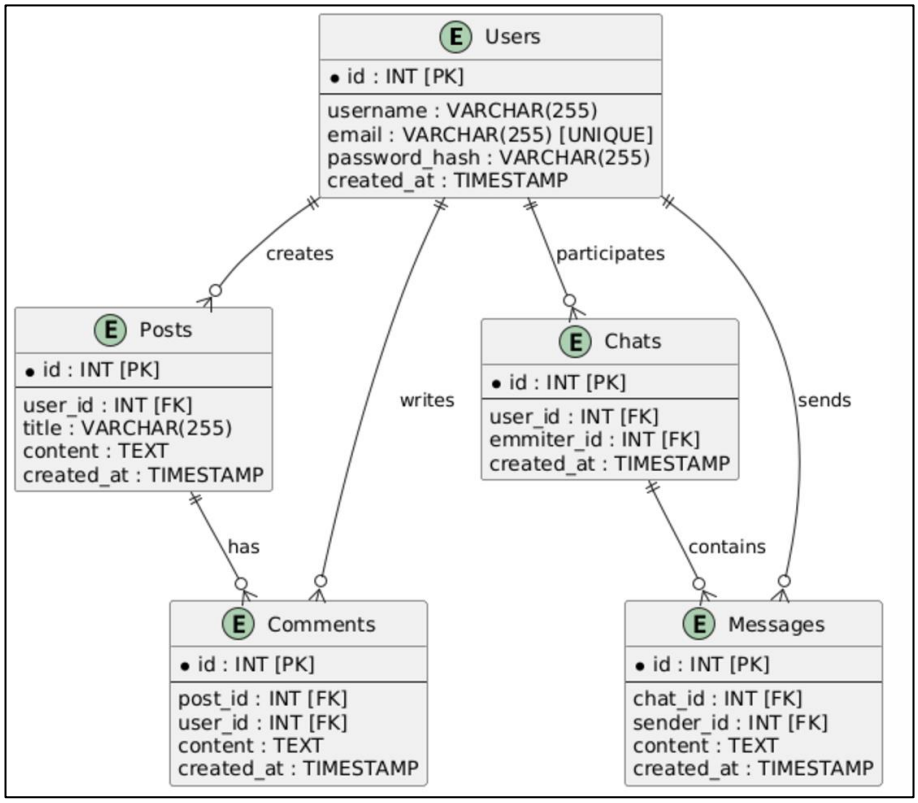


Рисунок 2.7 – Діаграма сутностей і зв’язків

На діаграмі відображено основні таблиці бази даних розробленої соціальної мережі, включаючи їхні атрибути та типи зв’язків. Сутності (таблиці) представляють ключові елементи системи, такі як користувачі, пости, коментарі, чати тощо. Зв’язки між таблицями визначають взаємодію між цими елементами, наприклад, зв’язок між користувачами та постами реалізується через зовнішній ключ, що вказує, який користувач створив конкретний пост.

ER-діаграма також допомагає ідентифікувати можливі помилки на етапі проєктування бази даних, такі як некоректні зв’язки або відсутність необхідних атрибутів. Це забезпечує створення структури, що відповідає вимогам системи і підтримує її масштабованість та надійність.

2.7 Висновки до розділу 2

У цьому розділі було детально розглянуто застосування UML для проєктування соціальної мережі на основі мікросервісної архітектури. Використання UML-діаграм дозволило чітко візуалізувати складні аспекти системи, що значно покращило розуміння її структури та функціональності, як для розробників, так і для інших зацікавлених осіб.

Завдяки діаграмам класів, послідовності, діяльності та компонентів вдалося створити об’єктивне уявлення про структуру системи та її компоненти, їх взаємодії та процеси. Використання діаграм класів допомогло чітко визначити структуру даних, взаємозв’язки між різними сервісами та забезпечити коректне функціонування кожного окремого елемента.

Діаграми послідовності стали корисними для детального опису обміну повідомленнями між компонентами в рамках кожного конкретного сценарію, що дозволило мінімізувати потенційні помилки при реалізації та тестуванні мікросервісів.

3 РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ДЛЯ СОЦІАЛЬНОЇ МЕРЕЖІ

3.1 Опис інстурментів розробки для соціальної мережі

Для реалізації соціальної мережі на основі мікросервісної архітектури було обрано інструменти, які забезпечують ефективність розробки, продуктивність системи та можливість масштабування. Основна увага приділяється використанню Node.js та RabbitMQ для побудови взаємодії між мікросервісами.

Node.js – це платформа для серверної розробки, що базується на JavaScript. Її переваги включають асинхронну модель роботи, яка дозволяє обробляти велику кількість одночасних запитів. Це ідеальний вибір для розробки мікросервісів, які повинні бути швидкими та масштабованими.

Для розробки REST API використовується Express.js – фреймворк, що забезпечує легку маршрутизацію та підтримку middleware [8]. Його гнучкість дозволяє швидко налаштовувати мікросервіси для виконання специфічних задач, таких як реєстрація користувачів, обробка постів, коментарів або обмін повідомленнями.

RabbitMQ використовується як система черг повідомлень для забезпечення асинхронної комунікації між мікросервісами. Наприклад, сервіс авторизації може передавати дані до сервісу сповіщень або аналітики, використовуючи черги RabbitMQ. Це дозволяє забезпечити стійкість системи та ізоляцію компонентів.

Для взаємодії з базою даних використовується ORM-бібліотека Sequelize [2]. Вона дозволяє уникнути написання сирого SQL-коду, забезпечуючи зручне управління моделями, міграціями та відносинами між таблицями.

MySQL обрано як основну базу даних для зберігання структурованої інформації, такої як профілі користувачів, пости, коментарі та повідомлення [5]. Завдяки своїй популярності, легкості у використанні та

продуктивності, MySQL забезпечує стабільну роботу системи. Її підтримка складних запитів і транзакцій робить її надійним вибором для зберігання даних соціальної мережі.

3.2 Основні сервіси соціальной мережі

Для забезпечення функціональності соціальної мережі на основі мікросервісної архітектури впроваджено декілька ключових сервісів, які взаємодіють між собою через черги повідомлень і вебсокети. Одним із основних компонентів є сервіс обміну повідомленнями, який реалізує миттєву комунікацію між користувачами (рис. 3.1, 3.2).

```
async registerHandler(queue, handler) {
    await this.channel.assertQueue(queue, { durable: false });

    this.channel.consume(queue, async (msg) => {
        if (msg !== null) {
            try {
                const messageContent = JSON.parse(msg.content.toString());
                await handler(messageContent, () => {
                    this.channel.ack(msg);
                }, () => {
                    this.channel.nack(msg);
                });
            } catch (error) {
                console.error(`Error processing message from queue ${queue}:`, error);
                this.channel.nack(msg);
            }
        }
    }, { noAck: false });
}

async sendToQueue(queue, message, options = {}) {
    await this.channel.assertQueue(queue, { durable: false });

    this.channel.sendToQueue(queue, Buffer.from(JSON.stringify(message)), {
        persistent: true,
        options,
    });
}

async consume(queue) {
    await this.channel.consume(queue, (msg) => {
        console.log(`Message received [${queue}]: ${msg.content.toString()}`);
        this.channel.ack(msg)
    });
}
```

Рисунок 3.1 – Приклад прослуховування черг та відправку повідомлення в exchange

```
async waitForReply(correlationId) {
    return new Promise((resolve, reject) => {
        this.replyHandlers[correlationId] = (messageContent) => {
            resolve(messageContent);
        };
    });
}

async consumeReplies() {
    console.log('Starting to consume replies...');
    const replyQueue = this.config.rabbitMq.queues.replyQueueToCommentsService;

    await this.channel.assertQueue(replyQueue, { durable: false });
    this.channel.consume(replyQueue, (msg) => {
        if (msg) {
            const messageContent = JSON.parse(msg.content.toString());
            const correlationId = messageContent.correlationId;
            try {
                if (this.replyHandlers[correlationId]) {
                    this.replyHandlers[correlationId](messageContent);
                    delete this.replyHandlers[correlationId];
                }
            } catch (error) {
                console.error('Error parsing message content:', error);
            }
        } else {
            console.log('No message received');
        }
    }, { noAck: true });
}
```

Рисунок 3.2 – Приклад прослуховування та отримання відповіді черги від consumer

Розглянемо головні сервіси.

SocketSercices – забезпечує двосторонню комунікацію між клієнтом і сервером, дозволяючи передавати дані без затримок. Коли один користувач надсилає повідомлення, воно через відкритий сокет передається серверу та доставляється адресату [3]. Це дозволяє реалізувати такі функції, як миттєва доставка текстових повідомлень, статус «онлайн/офлайн» користувача та індикація набору тексту.

RabbitMqService – після обробки повідомлення через вебсокети дані також передаються в чергу RabbitMQ. Це потрібно для забезпечення асинхронної взаємодії між сервісами. Наприклад, сервіс зберігання повідомлень отримує дані про нове повідомлення з черги та записує його в базу даних для подальшого доступу користувачами.

BaseSercvice – сервіс для роботи с базою даних, де реалізовано основні методи для роботи с ними (див. рис. 3.3).



```
export class BaseService {
    /**
     * @param {sequelize.Model} model
     */
    constructor(model) {
        this.model = model;
    }

    /**
     * Get count
     * @param scopes
     * @param countOptions
     * @returns {Promise.<void>}
     */
    getCount(scopes = [], countOptions) {
        return this.model
            .scope(scopes)
            .count(countOptions);
    }

    /**
     * Get list
     * @param query
     * @param scopes
     * @returns {Promise<Array<Model>>}
     */
    getList(query, scopes = []) {
        return this.model
            .scope(scopes)
            .findAll({ limit: query.limit, offset: query.offset });
    }

    /**
     * Get list without pagination
     * @param scopes
     * @returns {Promise<Array<Model>>}
     */
    getListWithoutPagination(scopes = []) {
        return this.model
            .scope(scopes)
            .findAll();
    }
```

Рисунок 3.3 – Приклад BaseService для роботи с базою даних

3.3 Тестування за допомогою Swagger

Swagger – це потужний інструмент для документування та тестування REST API. Завдяки своєму інтуїтивно зрозумілому інтерфейсу, він дозволяє розробникам і тестувальникам швидко взаємодіяти з API, перевіряючи його функціональність без необхідності використання зовнішніх інструментів (див. рис. 3.4 та 3.5).

Розглянемо основні переваги використання Swagger для тестування API.

Інтерактивна документація: Swagger автоматично генерує документацію на основі опису API (наприклад, OpenAPI Specification). Це дозволяє швидко ознайомитися з доступними ендпоінтами, методами запитів, параметрами, відповідями та кодами статусів.

Тестування в реальному часі: користувач може безпосередньо відправляти запити до API через вебінтерфейс Swagger і отримувати відповіді без необхідності писати додатковий код або використовувати інші інструменти, як-от Postman.

Підтримка різних типів запитів: Swagger дозволяє тестувати GET, POST, PUT, DELETE та інші типи HTTP-запитів, що охоплює більшість сценаріїв тестування API.

Валідація введених даних: інструмент перевіряє правильність переданих параметрів (тіло запиту, заголовки, query-параметри) на основі схем, описаних у документації.

```
    buildSwagger(application, appBuilder, version) {
        const url = version ? `/swagger/v${version}`: '/swagger';
        application.use(url, swagger.serveFiles(appBuilder.getDocs(version || INITIAL_VERSION)));
        const apiHtml = swagger.generateHTML(appBuilder.getDocs(version || INITIAL_VERSION));
        application.get(url, (req, res) => {
            res.send(apiHtml);
        });
    }
}
```

Рисунок 3.4 – Приклад інтеграції Swagger

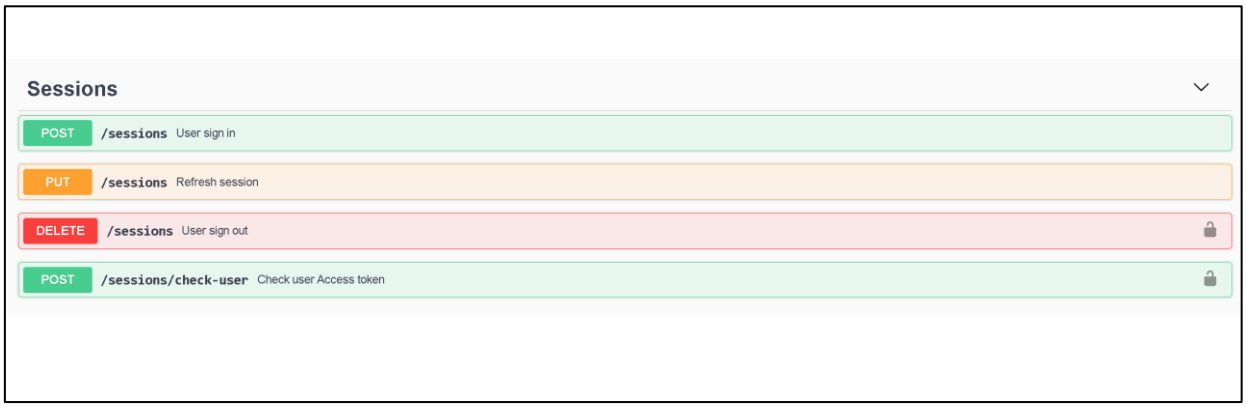


Рисунок 3.5 – Приклад API в Swagger

3.4 Реалізація бази даних

Для реалізації бази даних соціальної мережі було обрано реляційну систему управління базами даних MySQL, яка є стабільною та перевіреною технологією для зберігання структурованої інформації. MySQL забезпечує високу продуктивність, підтримку складних запитів та транзакцій, що є важливими для соціальної мережі з великою кількістю користувачів і даних.

У рамках мікросервісної архітектури кожен сервіс може взаємодіяти з базою даних через ORM-бібліотеку Sequelize. Це дозволяє уникнути написання сирого SQL-коду та працювати з базою даних через об’єкти JavaScript, що значно спрощує розробку та підтримку коду (рис. 3.6 – 3.8).

Розглянемо основні компоненти бази даних.

Моделі даних: Sequelize використовується для визначення моделей, які відображають таблиці в базі даних. Наприклад, модель користувача може включати такі поля, як ім’я, електронна пошта, пароль, а також зв’язки з іншими моделями (наприклад, пости, коментарі). Моделі забезпечують абстракцію над SQL-запитами та дозволяють працювати з даними через зручні методи.

Міграції: для забезпечення гнучкості та можливості змінювати структуру бази даних без втрати даних використовуються міграції. Вони дозволяють створювати, змінювати та видаляти таблиці і стовпці, зберігаючи при цьому збереження інформації між оновленнями.

Зв’язки між таблицями: у соціальній мережі часто необхідно встановлювати зв’язки між різними сутностями. Наприклад, користувачі можуть мати пости, коментарі можуть належати до постів, а користувачі можуть мати друзів. Для цього використовуються зв’язки типу hasMany, belongsTo, manyToMany, які дозволяють ефективно управляти відносинами між даними.

Індекси та оптимізація запитів: оскільки в соціальній мережі обробляється великий обсяг даних, важливо забезпечити ефективність

виконання запитів. Для цього використовуються індекси на часто запитувані поля (наприклад, на поле електронної пошти користувача), що дозволяє значно знизити час виконання запитів.

```
export class UserModelFactory {
    /**
     * @param {sequelize.Sequelize} sequelize
     * @return {UserModel}
     */
    static define(sequelize) {
        const User = sequelize.define(
            'User',
            ModelDefinitionBuilder.properties({
                name: {
                    type: sequelize.Sequelize.STRING(50),
                    allowNull: true,
                },

                email: {
                    type: sequelize.Sequelize.STRING(190),
                    allowNull: false,
                },

                password: {
                    type: sequelize.Sequelize.STRING(255),
                    allowNull: false,
                },

                salt: {
                    type: sequelize.Sequelize.STRING(255),
                    allowNull: true,
                }

            }),
            Object.assign(ModelDefinitionBuilder.options('users'), {
                scopes: Object.assign(ModelDefinitionBuilder.scopes(), {
                    byEmail: (email) => ({ where: { email } })
                })
            })
        );

        return User;
    }
}
```

Рисунок 3.6 – Приклад структури моделі користувача

```
'use strict';

module.exports = {
  up(queryInterface) {
    const createPostTableSql = `
          CREATE TABLE IF NOT EXISTS posts (
            id INTEGER PRIMARY KEY AUTO_INCREMENT,

            title VARCHAR(50) NOT NULL,
            text VARCHAR(255) NOT NULL,
            userId INTEGER NOT NULL,

            created_at DATETIME NULL DEFAULT CURRENT_TIMESTAMP,
            updated_at DATETIME NULL DEFAULT CURRENT_TIMESTAMP,
            CONSTRAINT postsUserIdFk FOREIGN KEY (userId) REFERENCES users (id)
          ) ENGINE=INNODB CHARACTER SET=UTF8MB4 COLLATE UTF8MB4_UNICODE_CI;`;

    return queryInterface.sequelize.query(createPostTableSql);
  },

  down(queryInterface) {
    return queryInterface.sequelize.query('DROP TABLE IF EXISTS posts;');
  }
};
```

Рисунок 3.7 – Приклад міграції для таблиці постів

```
export class PostModelFactory {
    /**
     * @param {sequelize.Sequelize} sequelize
     * @return {PostModel}
     */
    static define(sequelize) {
        /**
         * @type {PostModel}
         */
        const Post = sequelize.define(
            'Post',
            ModelDefinitionBuilder.properties({
                title: {
                    type: sequelize.Sequelize.STRING(50),
                    allowNull: false,
                },

                text: {
                    type: sequelize.Sequelize.STRING(2550),
                    allowNull: false,
                },

                userId: {
                    type: sequelize.Sequelize.INTEGER,
                    allowNull: false,
                },

            }),
            Object.assign(ModelDefinitionBuilder.options('posts'), {
                scopes: Object.assign(ModelDefinitionBuilder.scopes(), {
                    byUser: (emmiterIds) => ({ where: { userId: emmiterIds } })
                })
            })
        );

        Post.belongsTo(sequelize.models.User, { foreignKey: 'userId', as: 'user' });

        return Post;
    }
}
```

Рисунок 3.8 – Приклад структури моделі постів

Таким чином, реалізація бази даних з використанням MySQL та Sequelize забезпечує ефективне зберігання і обробку даних у соціальній мережі, дозволяючи підтримувати високу продуктивність та надійність системи при великих обсягах інформації.

3.5 Огляд застосунку

3.5.1 Реєстрація в системі

Перед початком використання додатку, користувачу необхідно зареєструватися в системі. Для цього потрібно натиснути на пункт меню «Login/Register», що знаходиться на панелі меню в шапці додатку (рис. 3.9).



Рисунок 3.9 – Пунк меню Реєстрації/Авторизації

Система відображає форму з можливістю вибрати один з двох режимів: вхід або реєстрація. Для створення нового облікового запису необхідно обрати пункт «Register». Система відображає форму реєстрації (рис. 3.10).

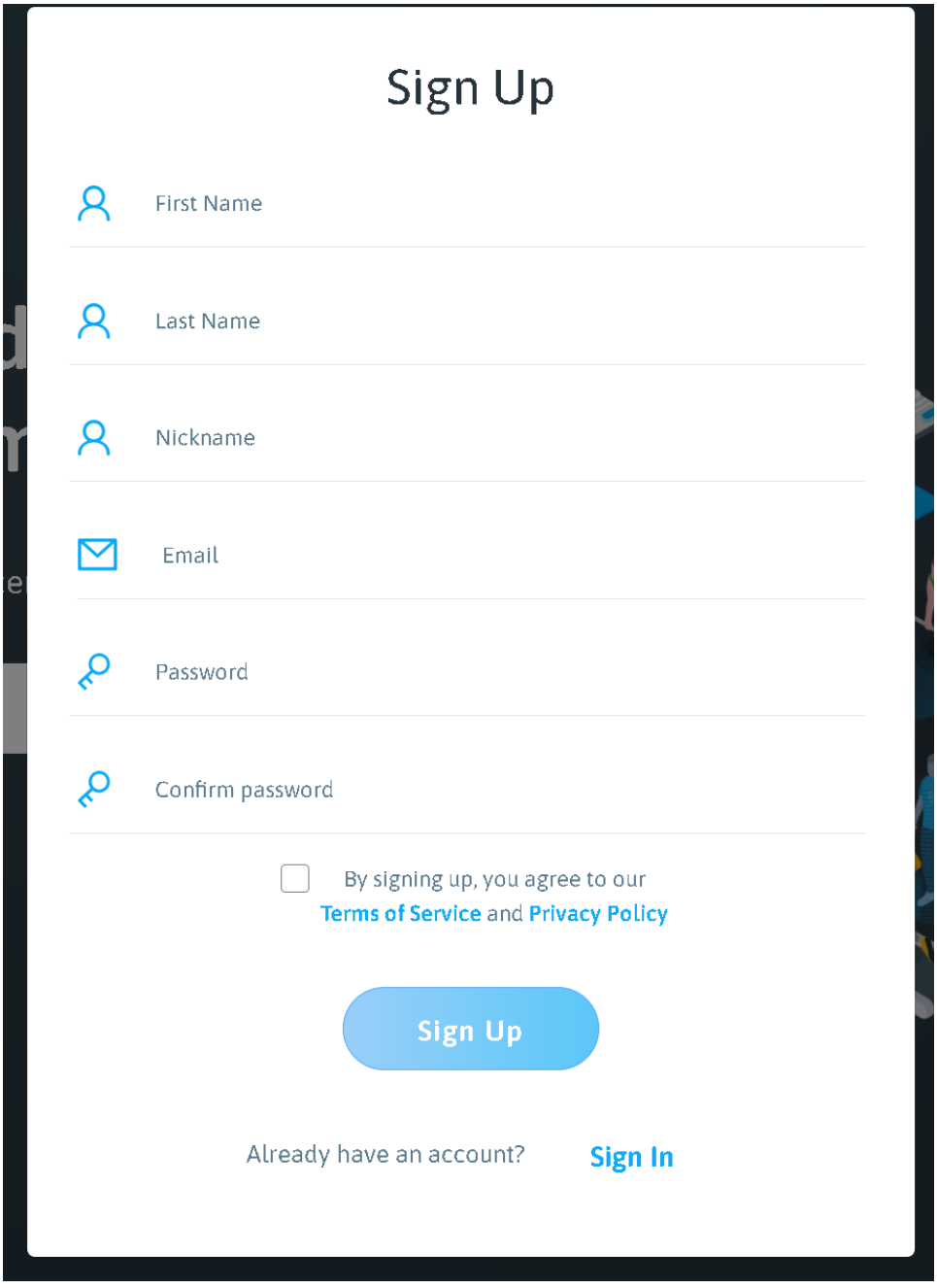


Рисунок 3.10 – Форма для реєстрації

Користувачу необхідно заповнити всі поля відповідними даними, якщо, при заповненні, користувач допускає помилку, тобто вводить невалідні дані, то форма про це сповіщає у вигляді відповідного повідомлення під полем, де було введено невалідні дані (див. рис. 3.11).

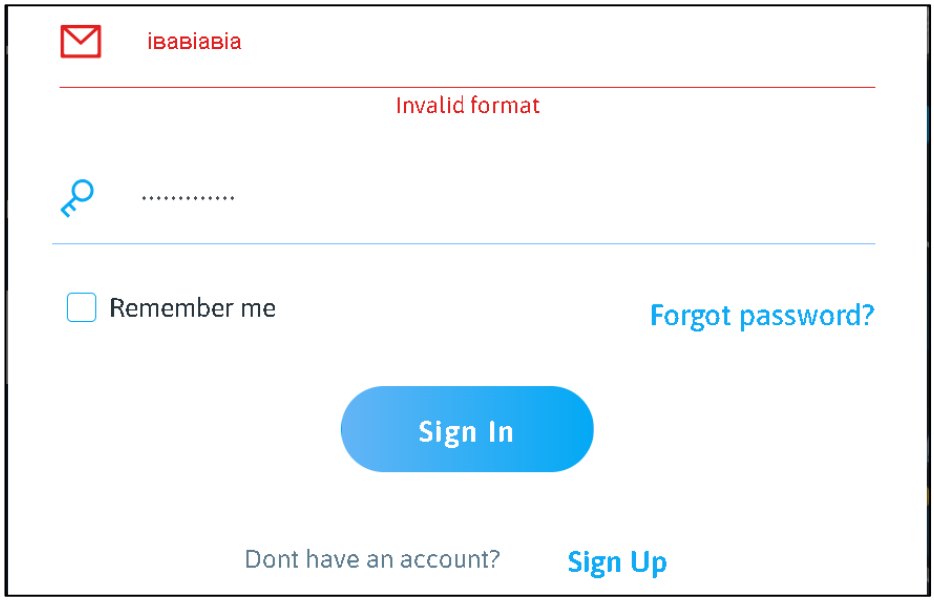


Рисунок 3.11 – Приклад невалідного email

3.5.2 Відправка коментарів

Користувачі можуть залишати свої коментарі до постів інших користувачів (див. рис. 3.12). Присутня валідаці для тексту коментаря (див. рис. 3.13).



Рисунок 3.12 – Приклад можливості залишити коментар під постом

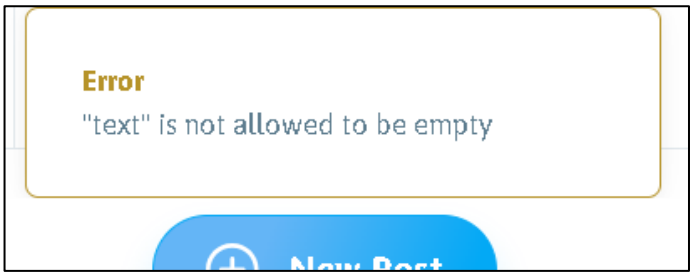


Рисунок 3.13 – Помилка при відправці пустого коментаря

3.5.3 Вигляд чату

Система відображає лайв чат між юзерами, а також відображае дату, коли повідомлення було відправлене (рис. 3.14).

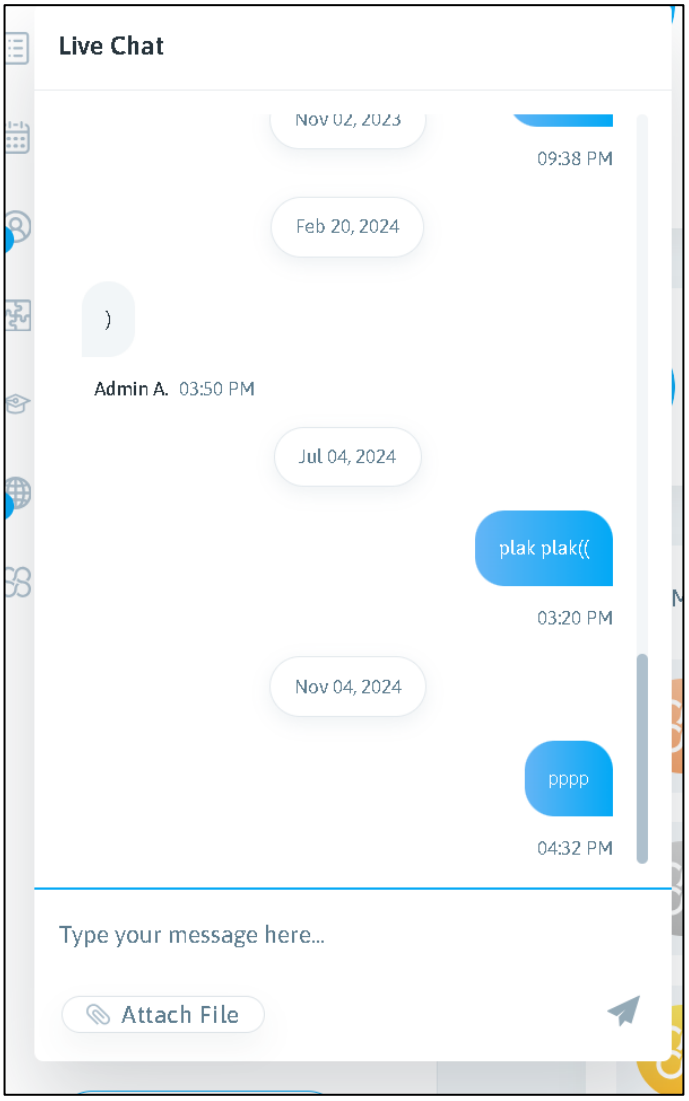


Рисунок 3.14 – Приклад чату

3.6 Висновки до розділу 3

Розробка соціальної мережі з використанням мікросервісної архітектури дозволила створити модульну та масштабовану систему, яка забезпечує високий рівень продуктивності та гнучкості. Впровадження ключових сервісів, таких як система обміну повідомленнями, реєстрація користувачів та коментування постів, забезпечило необхідний функціонал для інтерактивної взаємодії між користувачами.

Кожен із компонентів додатку був реалізований із застосуванням сучасних технологій, таких як Node.js, RabbitMQ, MySQL та Sequelize, що сприяє стійкості системи, швидкій обробці даних та ефективній взаємодії між сервісами. Використання Swagger значно спростило процес тестування та документування REST API, а впровадження валідації введених даних забезпечило високий рівень надійності та зручності використання.

Таким чином, розроблена соціальна мережа відповідає сучасним вимогам до програмних систем, забезпечуючи користувачів інтуїтивно зрозумілим інтерфейсом, функціональністю реального часу та стабільною роботою.

ВИСНОВКИ

Мікросервісна архітектура обрана завдяки її актуальності у сучасному середовищі розробки, особливо для складних та динамічних систем, таких як соціальні мережі. Така архітектура надає можливість створювати модульні, незалежні компоненти, кожен з яких відповідає за конкретну функцію та має власну базу даних. Це дозволяє гнучко підходити до розробки та підтримки системи, масштабувати компоненти відповідно до навантаження, спрощувати оновлення та впровадження нових функцій. Зокрема, під час реалізації проєкту було створено окремі мікросервіси для аутентифікації користувачів, обміну повідомленнями, управління профілями тощо, що забезпечило можливість гнучкого управління їхньою роботою.

Node.js був обраний як основна платформа для серверної частини завдяки своїй продуктивності, асинхронності та здатності працювати з великим потоком одночасних запитів [12]. Ці можливості виявилися особливо корисними у контексті соціальної мережі, де важливо забезпечити швидкий відгук системи та стабільну обробку великої кількості запитів. RabbitMQ, як система управління чергами повідомлень, дозволив ефективно організувати комунікацію між мікросервісами. Він забезпечив асинхронний, надійний обмін повідомленнями, що сприяє зменшенню затримок у роботі системи та підвищенню її стійкості до збоїв.

До головних переваг обраної архітектури належать її масштабованість та можливість оновлювати кожен мікросервіс незалежно від інших [11]. Ця властивість має особливе значення для соціальної мережі, де різні модулі можуть зазнавати різного навантаження залежно від активності користувачів. Завдяки цьому можна ефективно розподіляти ресурси і швидко адаптувати систему до зростання користувацької бази. Проте реалізація мікросервісної архітектури також вимагає подолання певних труднощів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Документація RabbitMQ. *RabbitMQ$_{TM}$*. URL: https://www.rabbitmq.com/documentation.html (дата звернення: 12.10.2024).

2. Документація Sequelize. URL: https://sequelize.org/ (дата звернення: 12.10.2024).

3. Документація про Socket.IO. URL: https://socket.io/docs/ (дата звернення: 13.10.2024).

4. Документація JWT (JSON Web Tokens). URL: https://jwt.io/introduction/ (дата звернення: 13.10.2024).

5. Документація MySQL. URL: https://dev.mysql.com/doc/ (дата звернення: 13.10.2024).

6. Додаткове посилання – Документація про Docker. URL: https://docs.docker.com/ (дата звернення: 14.10.2024).

7. Офіційна документація Node.js. URL: https://nodejs.org/en/docs (дата звернення: 14.10.2024).

8. Офіційна документація Express.js. URL: https://expressjs.com/ (дата звернення: 15.10.2024).

9. Newman S. Building Microservices. O'Reilly Media, Sebastopol, 2015. 280 p.

10. Resende D. Hands-On Microservices with Node.js. Packt Publishing, Birmingham, 2019. 356 p.

11. Hussain F., Hussain K. Mastering Firebase: The Complete Guide to Building and Scaling Apps. Independently published, Michigan, 2024. 228 p.

12. Casciaro M., Mammino L. Node.js Design Patterns. Packt Publishing, Birmingham, 2020. 646 p.

13. Roy G. M. RabbitMQ in Depth. Manning Publications,Shelter Island, 2017. 375 p.

14. Unhelkar B. Software Engineering with UML. Auerbach Publications, Boca Raton, 2005. 504 p.

ДОДАТОК А

Код сервісу RabbitMQ

```
import amqp from 'amqplib'

export default class RabbitMqService {

constructor(container) {

/*

@type {object}

*/

this.config = container.config

/*

@type {object}

*/

this.options = {

credentials: amqp.credentials.plain(this.config.rabbitMq.username,
this.config.rabbitMq.password)

}

/*

@type {object}

*/

this.replyHandlers = {}

/*

@type {object}
```

```
*/
this.channel;
/*
@type {object}
*/
this.connection;
}
async init() {
this.connection = await amqp.connect(this.config.get('rabbitMq').url);

this.channel = await this.connection.createChannel();
}
async registerHandler(queue, handler) {
await this.channel.assertQueue(queue, { durable: false });

this.channel.consume(queue, async (msg) => {
if (msg !== null) {
try {
const messageContent = JSON.parse(msg.content.toString());
await handler(messageContent, () => {
this.channel.ack(msg);
}, () => {
this.channel.nack(msg);
```

```
});
} catch (error) {
console.error(`Error processing message from queue ${queue}:`, error);
this.channel.nack(msg);
}
}
}, { noAck: false });
}
async sendToQueue(queue, message, options = {}) {

await this.channel.assertQueue(queue, { durable: false });

this.channel.sendToQueue(queue, Buffer.from(JSON.stringify(message)), {
persistent: true,
options,
});
}
async consume(queue) {
await this.channel.consume(queue, (msg) => {
this.channel.ack(msg)
});
}
async waitForReply(correlationId) {
```

```
return new Promise((resolve, reject) => {
this.replyHandlers[correlationId] = (messageContent) => {
resolve(messageContent);
};
});
}
async consumeReplies() {
console.log('Starting to consume replies...');
const replyQueue = this.config.rabbitMq.queues.replyQueueToUsersService;

await this.channel.assertQueue(replyQueue, { durable: false });

this.channel.consume(replyQueue, (msg) => {
if (msg) {
const messageContent = JSON.parse(msg.content.toString());

const correlationId = messageContent.correlationId;

try {
if (this.replyHandlers[correlationId]) {
this.replyHandlers[correlationId](messageContent);

delete this.replyHandlers[correlationId];
```

```
}
} catch (error) {
console.error('Error parsing message content:', error);
}
} else {
console.log('No message received');
}
}, { noAck: true });
}
}
```

ДОДАТОК Б

Код прослуховування черг

```
async setupRabbitMqListeners() {
try {
await this.rabbitMqService.init();
console.log('RabbitMQ service initialized.');
await this.rabbitMqService.consumeReplies();
const queues = config.get('rabbitMq').queues;
for (let nameQueue in queues) {
const queueName = queues[nameQueue];
if (this.commentsServices.hasHandlerForQueue(queueName)) {
console.log(`Setting up handler for queue: ${queueName}`);
this.rabbitMqService.registerHandler(queueName, async (message, ack, nack) =>
{
try {
console.log(`Message received in queue ${queueName}:`, message);
await this.postsService.commentsServices[queueName](message);
ack();
} catch (error) {
console.error(`Error handling message in queue ${queueName}:`, error);
}
});
```

```
console.log(`Subscribed to queue: ${queueName}`);
} else {
console.log(`No handler found for queue: ${queueName}, skipping registration.`);
}
}
} catch (error) {
console.error('Error initializing RabbitMQ service:', error);
}
}
```