НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп'ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД

Виконав : студент __4__ курсу, групи _ІО-15_
(шифр групи)

_____Жадько Назар Юрійович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____ас. Русінов Володимир Володимирович_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль)_ ас. Пономаренко Артем Миколайович___
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп'ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Жадька Назара Юрійовича

1. Тема проєкту Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД
керівник проєкту Русінов Володимир Володимирович, асистент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23 квітня 2025 року №1705-с

2. Термін здачі студентом закінченого проєкту *11.06.2025*.

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Огляд існуючих підходів до побудови мікросервісної архітектури з розподіленою СУБД.

Розділ 2. Огляд технологій для розробки системи.

Розділ 3. Проектування архітектури та розробка системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) принципова схема, функціональна схема, структурна схема системи.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	ас. Пономаренко А.М.		

7. Дата видачі завдання _____.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	20.11.2024	
2.	*Вивчення та аналіз завдання*	10.03.2025	
3.	*Розробка архітектури та загальної структури системи*	24.03.2025	
4.	*Розробка структур окремих підсистем*	07.04.2025	
5.	*Програмна реалізація системи*	05.05.2025	
6.	*Оформлення пояснювальної записки*	02.05.2025	
7.	*Захист програмного продукту*	09.06.2025	
8.	*Передзахист*	12.06.2025	
9.	*Захист*	16.06.2025	

Студент-дипломник _____ Назар ЖАДЬКО
(підпис)

Керівник проєкту _____ Володимир РУСІНОВ
(підпис)

АНОТАЦІЯ

У даній роботі розглянуто побудову сучасних веб-сервісів з використанням мікросервісної архітектури та розподіленої СУБД. В якості прикладу розроблено веб-сервіс для взаємодії з особистими проєктами, який забезпечує ефективну комунікацію між інвесторами, менторами, засновниками та іншими учасниками стартап-екосистеми. У роботі детально проаналізовано вимоги до системи, спроєктовано архітектуру та реалізовано серверну логіку з використанням мови програмування Python. Особливу увагу приділено організації обміну даними між мікросервісами, підтримці масштабованості та забезпеченню високої доступності завдяки використанню розподіленої СУБД. Також обговорюються проблеми, пов'язані з координацією роботи сервісів та узгодженістю даних, і шляхи їх подолання.

Ключові слова: веб-сервіс, стартап, мікросервісна архітектура, розподілена база даних, Python, взаємодія з користувачем.

ANNOTATION

This qualification work considers approaches to building modern web services using microservice architecture and distributed DBMS. As an example, we developed a web service for interaction with personal startup projects, which ensures effective communication between investors, mentors, founders and other participants in the startup ecosystem. The paper analyzes the system requirements in detail, designs the architecture, and implements the server logic using the Python programming language. Particular attention is paid to organizing data exchange between microservices, maintaining scalability, and ensuring high availability through the use of a distributed DBMS. The challenges associated with service coordination and data consistency and ways to overcome them are also discussed.

Keywords: web service, microservice architecture, distributed database, Python, user interaction.

Справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземплярів	*Додаток*
			Документація загальна			
	А4	***ІАЛЦ.467100.002 ТЗ***	Сервіс для взаємодії з проєктами	3		
			з використанням мікросервісної			
			архітектури та РСУБД			
			Технічне завдання			
	А4	***ІАЛЦ.467100.003 ПЗ***	Сервіс для взаємодії з проєктами	61		
			з використанням мікросервісної			
			архітектури та РСУБД			
			Пояснювальна записка			
	А4	***ІАЛЦ.467100.004 Д1***	Сервіс для взаємодії з проєктами	1		
			з використанням мікросервісної			
			архітектури та РСУБД			
			Принципова схема			
	А4	***ІАЛЦ.467100.005 Д2***	Сервіс для взаємодії з проєктами	1		
			з використанням мікросервісної			
			архітектури та РСУБД			
			Функціональна схема			
	А4	***ІАЛЦ.467100.006 Д3***	Сервіс для взаємодії з проєктами	1		
			з використанням мікросервісної			
			архітектури та РСУБД			
			Структурна схема			
	А4	***ІАЛЦ.467100.007 Д4***	Сервіс для взаємодії з проєктами	106		
			з використанням мікросервісної			
			архітектури та РСУБД			
			Текст програмного коду			

					ІАЛЦ.467100.001 ОА
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	
Розроб		Жадько Н.Ю.			*Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД* Опис альбому
Перев		Русінов В. В.			

Літ.	Аркуш	Аркушів
	1	1

НТУУ "КПІ" ФІОТ
ІО-15

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД*»

Київ – 2025

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку сервісу для взаємодії зі стартапами з використанням мікросервісної архітектури та розподіленої системи управління базами даних (СУБД).

Область застосування веб-сервісу охоплює процеси ініціації, розвитку та підтримки стартапів, а також взаємодію між інвесторами, менторами, засновниками проєктів та іншими зацікавленими сторонами. Основна мета сервісу — забезпечити централізовану платформу для ефективної взаємодії користувачів з стартапами, що сприяє першому знайомству з проєктом, прозорій комунікації, обміну інформацією, рішеннями та ідеями.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання на виконання кваліфікаційної роботи бакалавра в рамках освітньо-професійної програми за напрямом «Інженерія програмного забезпечення», затверджене факультетом інформатики та обчислювальної техніки кафедри комп'ютерної інженерії Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка веб-сервісу для взаємодії зі стартапами з використанням мікросервісної архітектури, що забезпечить ефективну комунікацію між учасниками стартап-екосистеми. Сервіс покликаний спростити процес пошуку інвесторів, менторів та партнерів, покращити видимість стартапів, а також підтримувати їхній розвиток завдяки залученню учасників стартап-екосистеми.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації для виконання цієї роботи є технічна література, офіційна документація та інтернет-статті.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

- Користувачі повинні мати змогу зареєструвати обліковий запис та проходити автентифікацію з використанням електронної пошти.
- Користувачі повинні мати можливість створювати проєкти з описом ідеї, прикріпленням фото, формуванням команди.
- Користувачі повинні мати можливість писати коментарі до проєкту, залишати заявки для участі в розробці, інвестуванні чи менторстві.
- Сервіси повинні мати зрозумілу документацію API.
- Система повинна бути побудована за мікросервісною архітектурою з використанням розподіленої СУБД

5.2. Вимоги до програмного забезпечення

- Операційна система: Windows, macOS або Linux.
- Сумісність із сучасними версіями веб-браузерів.

5.3. Вимоги до апаратної частини

- Процесор: не нижче Intel® Core™ i3-2100T або еквівалентний.
- Пам’ять: щонайменше 4 ГБ оперативної пам’яті (RAM), 10 ГБ вільного простору на накопичувальному пристрої (ROM).
- Постійне та стабільне підключення до мережі Інтернет.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	20.11.2024
Вивчення та аналіз завдання	10.03.2025
Розробка архітектури та загальної структури системи	24.03.2025
Розробка структур окремих частин системи	07.04.2025
Програмна реалізація системи	05.05.2025
Виправлення помилок	09.05.2025
Оформлення пояснювальної записки	02.05.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

SSR	(Server-Side Rendering) Рендеринг на стороні серверу
СУБД	Система управління базами даних
РСУБД	Розподілена система управління базами даних
JSON	(JavaScript Object Notation) Текстовий формат обміну даними
API	Application Programming Interface (інтерфейс прикладного програмування)
CRUD	Create, Read, Update, Delete (основні операції над даними)
JWT	JSON Web Token (токен автентифікації)
HTTP	Hypertext Transfer Protocol (протокол передачі гіпертексту)
ORM	Object-Relational Mapping (об'єктно-реляційне відображення)
AMQP	Advanced Message Queuing Protocol (протокол обміну повідомленнями, RabbitMQ)
CQRS	(Command Query Responsibility Segregation) Розділення команд та запитів
SOA	(Service-Oriented Architecture) Орієнтована на сервіси архітектура

ВСТУП

У сучасному цифровому середовищі дедалі більше людей реалізують власні ініціативи — від творчих і навчальних проєктів до стартапів та власних ІТ-рішень. Для успішної реалізації і розвитку особистих проєктів необхідно знаходитись в стартап-екосистемі, де ви можете протестувати ідею, знайти партнерів, залучити інвесторів, отримати перших користувачів і цінний зворотній зв’язок. Відсутність такої екосистеми ускладнює реалізацію власних проєктів особливо для індивідуальних розробників і малих команд.

У цій бакалаврській роботі зосереджено проєктування та розробка веб-сервісу для взаємодії з проєктами з використанням мікросервісної архітектури та систем управління базами даних. Метою є створення масштабованого, зручного та функціонального веб-сервісу для платформи, яка дозволить користувачам презентувати власні проєкти, взаємодіяти з потенційними партнерами, менторами та інвесторами, а також отримувати зворотний зв’язок і залучати перших користувачів.

Розробка сервісу передбачає використання сучасних технологій бекенд розробки, реалізації оркестрації, а також впровадження мікросервісної архітектури, яка забезпечує гнучкість, надійність, можливість незалежного масштабування та зручної підтримки модулів. У рамках роботи буде виконано огляд існуючих підходів, вибір технологій, проєктування архітектури, реалізацію основного функціоналу, тестування та розгортання.

Очікується, що результатом стане сервіс, який сприятиме ефективній реалізації та розвитку особистих проєктів, надаючи користувачам інструменти для комунікації, перевірки ідей та пошуку підтримки. Робота також має на меті поглибити практичні знання у сфері веб-розробки, мікросервісного проєктування та організації баз даних у сучасних інформаційних системах.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ ПІДХОДІВ ДО ПОБУДОВИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ З РОЗПОДІЛЕНОЮ СУБД

1.1 Архітектурні підходи: монолітна та мікросервісна архітектури.

Одним із ключових етапів розробки сучасного програмного забезпечення є вибір архітектурного підходу, який визначає гнучкість, масштабованість, продуктивність і швидкість розгортання системи. Найпоширенішими підходами на практиці залишаються монолітні та мікросервісні архітектури. Вибір між ними залежить від багатьох факторів: масштабу проекту, кількості членів команди, рівня складності бізнес-логіки та вимог до інфраструктури. Обидва підходи мають свої переваги та недоліки, і кожен краще підходить для різних типів систем.

Монолітна архітектура реалізує всю систему як єдиний програмний блок, що поєднує в собі всі компоненти: інтерфейс користувача, бізнес-логіку та доступ до даних. Усі частини працюють у спільному середовищі, що значно спрощує розробку, тестування та початкове розгортання. Оскільки вся система інтегрована, немає необхідності створювати окрему інфраструктуру для послуг, балансування навантаження або контейнеризації. Тому моноліт підходить для невеликих або пілотних проектів, де важлива швидкість реалізації. Схематичне зображення монолітної архітектури показано на рисунку 1.1.

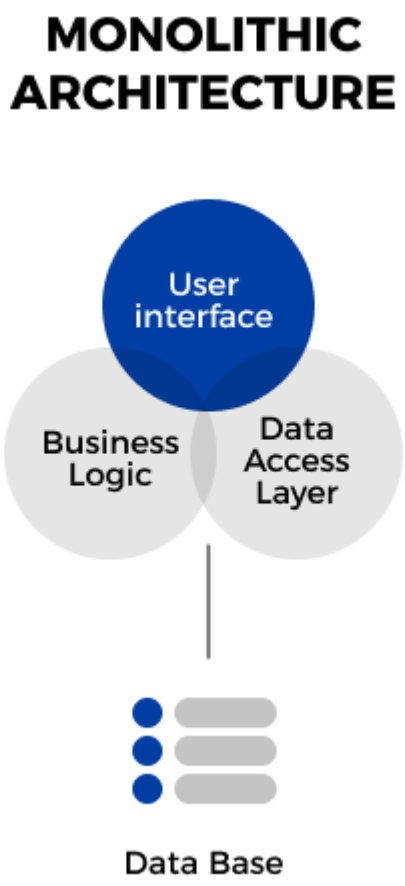


Рисунок 1.1 – Зображення структури монолітної архітектури

Архітектура мікросервісів є альтернативою моноліту і передбачає побудову системи як набору незалежних сервісів, кожен з яких реалізує окрему бізнес-функцію. Кожен мікросервіс має власну базу даних, логіку, інтерфейс доступу та життєвий цикл. Завдяки цьому кожна команда може працювати автономно, застосовуючи різні технології та підходи для реалізації конкретної послуги. Цей підхід забезпечує велику гнучкість і масштабованість: сервіси можна оновлювати, масштабувати та розгортати окремо. У разі несправності мікросервіса решта системи залишиться в робочому стані, підвищуючи надійність і покращуючи роботу користувача. Схематичне зображення базової структури архітектури мікросервісів показано на рисунку 1.2.

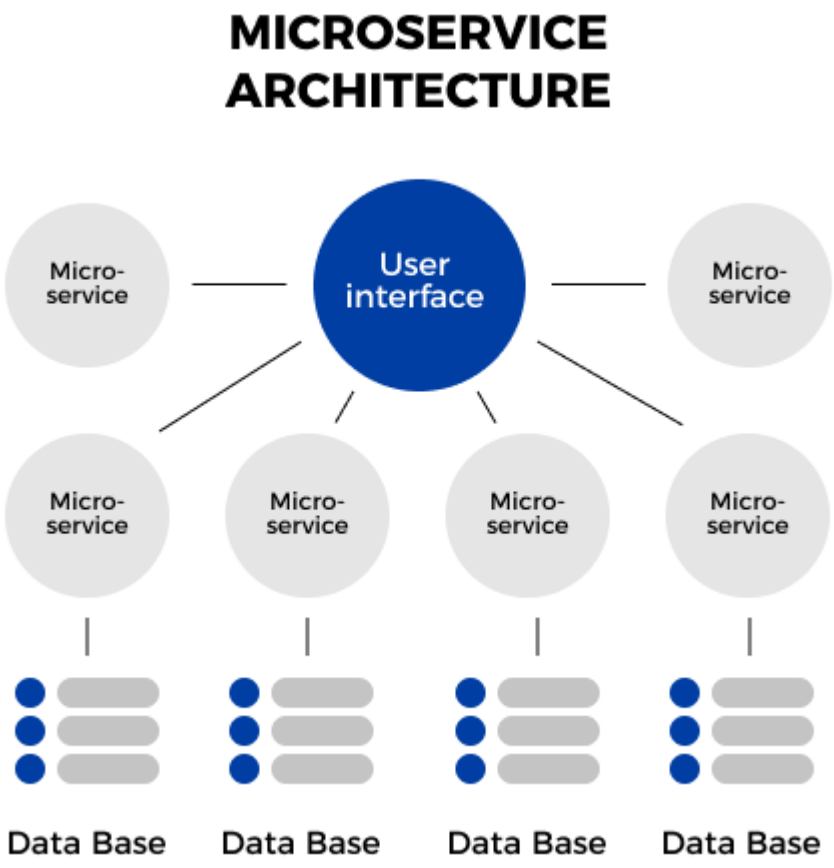


Рисунок 1.2 – Зображення структури мікросервісної архітектури

Однак із зростанням складності проекту монолітна архітектура починає створювати обмеження. Велика кодова база ускладнює командну розробку: внесення змін вимагає координації між різними частинами системи, що сповільнює впровадження нових функцій. Відсутність ізоляції між модулями означає, що збій або помилка в одному компоненті може вплинути на всю систему. Крім того, у разі високого навантаження необхідно масштабувати весь додаток, навіть якщо навантаження спостерігається лише в одному функціональному блоці. Також важко змінити частину технологічного стеку, не ризикуючи вплинути на інші компоненти, оскільки все реалізовано в одному середовищі.

Крім того, мікросервісна архітектура вимагає значно вищого рівня технічної зрілості. Для ефективного впровадження потрібна розвинена

інфраструктура: інструменти CI/CD, системи реєстрації та моніторингу, шлюзи API та механізми відстеження запитів між сервісами. Також важливо забезпечити механізми для уніфікації транзакцій, оскільки використання окремих баз даних у різних сервісах іноді робить неможливим використання регулярних транзакцій ACID. Іншою проблемою є звітування та аналіз: дані розподіляються між сервісами, що ускладнює агрегування інформації. Через це вартість впровадження та підтримки архітектури мікросервісів, особливо на ранніх етапах, може бути досить високою. Іноді під час повернення мікросервісу до попередньої версії може виникнути проблема сумісності мікросервісів.

В рамках даної дипломної роботи, враховуючи потенціал зростання сервісу, який би підтримував взаємодію з численними персональними проєктами, а також необхідність самостійної розробки функціональних модулів, була обрана архітектура мікросервісів. Такий підхід дозволить нам гнучко масштабувати систему в майбутньому, інтегрувати нові функції, не порушуючи роботу платформи, і ефективно розподіляти відповідальність між сервісами.

1.2 Принципи мікросервісної архітектури

Побудова мікросервісної архітектури ґрунтується на низці ключових принципів, дотримання яких є критично важливим для досягнення стабільності, масштабованості та ефективного розвитку системи. Нижче розглянуто основні з них.

1. Принцип єдиної відповідальності (Single Responsibility Principle)

Кожен мікросервіс повинен мати чітко визначене функціональне призначення та вирішувати лише одну бізнес-задачу. Це узгоджується з принципами SOLID та дозволяє уникнути надмірної складності, сприяє повторному використанню коду та спрощує супровід. Мікросервіси мають бути максимально ізольованими — як логічно, так і на рівні даних. Тому кожен

сервіс повинен мати власну базу даних або сховище, не розділяючи його з іншими компонентами системи.

2. Орієнтація на бізнес-функції

Мікросервіси формуються не навколо технічних задач, а навколо бізнес-можливостей (business capabilities). Такий підхід дозволяє створювати автономні модулі, кожен з яких повністю реалізує певну бізнес-ділянку, наприклад, управління користувачами, обробка платежів або логістика. Однією з головних переваг є можливість використання різних технологічних стеків у межах однієї системи. Наприклад, один сервіс може бути написаний на Python, інший — на Node.js або Go, залежно від потреб конкретного завдання.

3. Незалежність і відмовостійкість

Система повинна залишатися функціональною навіть у разі виходу з ладу одного або кількох мікросервісів. Архітектура має бути побудована за принципом розробки з урахуванням збоїв (designed for failure). Наприклад, при відмові сервісу логування або аналітики — основна бізнес-логіка повинна продовжити працювати. Це досягається за допомогою розподілу відповідальності, резервного кешування, тайм-аутів, механізмів автоматичного відновлення або ізоляції (circuit breakers, retries, fallback-стратегії).

4. Автономність розробки та розгортання

Кожен мікросервіс розробляється, тестується, масштабується та розгортається незалежно від інших. Це дозволяє командам працювати паралельно, впроваджувати нові функції без впливу на інші частини системи та здійснювати оновлення з мінімальним простоєм. Такий підхід добре поєднується з DevOps-культурою та CI/CD процесами.

5. Взаємодія через стандартизовані інтерфейси

Комунікація між мікросервісами відбувається виключно через чітко визначені контракти — зазвичай через REST API, gRPC або подієву шину. Це дозволяє зменшити зв’язаність сервісів та спростити обслуговування. Деякі сервіси можуть надавати веб-інтерфейси, інші — лише машинне API. Усі вони

повинні дотримуватися стабільного інтерфейсу для взаємодії з іншими модулями.

6. Контейнеризація і гнучке розгортання

У мікросервісній архітектурі зазвичай використовується контейнеризація (наприклад, Docker), що дозволяє запускати сервіси у відокремленому середовищі. Це забезпечує повторюваність середовища, масштабованість та зручність розгортання у хмарній інфраструктурі. На практиці кожен мікросервіс є окремим процесом або контейнером, який можна запускати на різних хостах, у хмарах або кластеризованих середовищах на зразок Kubernetes.

1.3 Патерни побудови мікросервісної архітектури

З моменту створення перших програмних систем пройшло чимало часу, і з розвитком IT-галузі зросла складність проєктів, а разом із нею — і вимоги до їх гнучкості, масштабованості та здатності до змін. Монолітні підходи виявилися обмеженими в контексті швидкого розвитку функціональності, тестування та масштабування окремих компонентів. У відповідь на ці виклики були сформульовані принципи побудови мікросервісної архітектури, а також розроблено набір архітектурних шаблонів (патернів), які допомагають ефективно реалізовувати і підтримувати розподілені системи.

Одними з найпоширеніших шаблонів, які використовуються при побудові мікросервісної архітектури, є наступні:

Decompose by Subdomain

Цей патерн передбачає поділ великого домену (особливо в існуючих монолітних системах) на менші піддоменні частини, кожна з яких реалізується у вигляді окремого мікросервісу. Такий підхід ґрунтується на концепціях доменно-орієнтованого проєктування (DDD) і дозволяє поступово перетворювати моноліт на систему з чітко окресленими межами відповідальності між модулями. [15]

Піддомени можна класифікувати як: основні, допоміжні, загальні.

Переваги такого підходу полягають у гнучкості, масштабованості та простоті повторного використання компонентів. Водночас, недоліком може стати велика кількість мікросервісів, що ускладнює їхню інтеграцію та обслуговування.

Decompose by Business Capability

Ще один важливий патерн — розподіл системи за бізнес-можливостями. Відповідно до цього шаблону, кожен мікросервіс реалізує конкретну функціональність, яка відповідає певній бізнес-ролі, наприклад, керування категоріями, користувачами, товарами тощо. Перевагою є слабкий рівень зв’язаності між сервісами, чітке розмежування відповідальностей і зручність у масштабуванні команди: розробник легко розуміє структуру сервісу та його призначення.

Одним із недоліків є те, що побудова сервісу тісно пов’язана з бізнес-моделлю, отже, архітектор повинен мати глибоке розуміння предметної області, інакше виникають ризики надмірної фрагментації або дублювання логіки.

Database per Service

Один із фундаментальних принципів мікросервісної архітектури — це ізоляція даних. Кожен мікросервіс повинен мати власну базу даних або сховище, до яких не мають прямого доступу інші сервіси. Це забезпечує автономність і дозволяє використовувати ті бази даних, які найкраще відповідають технічним або бізнес-вимогам конкретного сервісу (наприклад, PostgreSQL, MongoDB, Redis). Існує кілька способів реалізації ізоляції: приватні таблиці в загальній БД, окрема схема для кожного сервісу, окремий сервер бази даних на сервіс.

Основною перевагою цього підходу є незалежність — зміни в одному сервісі не впливають на інші. Проте виникають труднощі з реалізацією транзакцій, які охоплюють кілька сервісів (наприклад, підтвердження

замовлення та списання коштів), а також з виконанням звітних запитів, що потребують об’єднання даних із різних баз.

Saga

Патерн Saga дозволяє реалізувати розподілені транзакції між кількома мікросервісами, які не мають спільної бази даних. Замість однієї глобальної транзакції виконується послідовність локальних транзакцій, кожна з яких публікує подію для ініціації наступної. У разі виникнення помилки, попередні транзакції компенсуються за допомогою спеціальних компенсуючих операцій.

Існує два способи реалізації Saga: хореографія, де кожен сервіс самостійно реагує на події без централізованого контролю, та оркестрація, коли спеціальний сервіс управляє послідовністю транзакцій і координує взаємодію між мікросервісами.

Композиція API та CQRS

У випадках, коли потрібно отримати об'єднані дані з кількох сервісів, використовуються патерни API-композиції або CQRS. Наприклад, щоб сформувати інтерфейс «Мій профіль», система спочатку звертається до сервісу користувача, а потім — до сервісу замовлень, об’єднуючи відповіді на рівні API-шлюзу або окремого агрегуючого компонента. CQRS дозволяє окремо обробляти запити на зміну даних (команди) та на отримання (запити), що дозволяє краще оптимізувати архітектуру під кожен з цих процесів.

Усі згадані патерни спрямовані на вирішення конкретних задач, з якими стикається команда розробки при переході до мікросервісної архітектури. Їх правильне застосування дозволяє досягнути гнучкої, масштабованої та стійкої до збоїв системи, здатної адаптуватися до зміни бізнес-вимог і технічних умов. У дипломній роботі деякі з цих патернів будуть реалізовані практично — зокрема, буде застосовано принцип поділу за бізнес-можливостями, ізоляцію даних на рівні сервісів, а також орієнтацію на слабку зв’язаність та незалежне розгортання компонентів.

1.4 Основні поняття розподіленої системи управління базами даних

Технологія розподілених баз даних, яка зараз широко використовується, сприяє переходу від централізованої до децентралізованої обробки даних. Розробка розподілених систем керування базами даних є одним із найбільших досягнень у галузі баз даних. Основною причиною розвитку інформаційних систем на основі баз даних є бажання об'єднати всі дані, що обробляються в компанії, в єдине ціле та забезпечити контрольований доступ до них. [17]

База даних — це впорядкована сукупність пов'язаних даних, створена з певною метою. Базу даних можна організувати як сукупність кількох таблиць, де кожна таблиця представляє елемент або сутність реального світу. Кожна таблиця містить різні поля, які відображають характерні ознаки сутності. Наприклад, корпоративна база даних може містити таблиці для проектів, співробітників, відділів, продуктів і фінансових даних. Поля в таблиці співробітників можуть бути ім'ям, ідентифікатором компанії, датою початку тощо.

Система керування базами даних — це сукупність програм, які дозволяють створювати та підтримувати базу даних. СУБД — це програмний пакет, який полегшує визначення, створення, редагування та обмін даними в базі даних. Побудова бази даних передбачає зберігання даних на будь-якому носії інформації. Редагування передбачає отримання інформації з бази даних, її оновлення та формування звітів. Спільний доступ до даних дає змогу отримувати доступ до даних різним користувачам або програмам.

Розподілена база даних містить в собі кілька взаємопов'язаних баз даних, які розподілені через комп'ютерну мережу або Інтернет. Розподілена система управління базою даних (РСУБД) керує розподіленою базою даних і забезпечує механізми, щоб зробити бази даних прозорими для користувачів. У цих системах дані систематично розподіляються між кількома вузлами, щоб оптимально використовувати всі обчислювальні ресурси компанії.

Розподілена база даних — це сукупність взаємозв’язаних баз даних, які фізично розподілені в різних місцях і взаємодіють через Інтернет або комп’ютерну мережу. Приклад розподіленої бази даних наведено на рисунку 1.3. Також, можна виділити такі особливості розподіленої бази даних:

- Бази даних у системі логічно пов’язані між собою та часто розглядаються як єдина логічна структура.
- Дані зберігаються фізично на різних вузлах, причому кожен вузол може обслуговуватись окремою СУБД, незалежно від інших.
- Обчислювальні ресурси (процесори) на кожній локації з'єднані мережею, але не утворюють єдиної багатопроцесорної системи.
- Розподілена база даних не є простою слабо інтегрованою файловою системою.
- Хоча розподілена база даних підтримує обробку транзакцій, вона не є повною заміною традиційної системи транзакційної обробки.

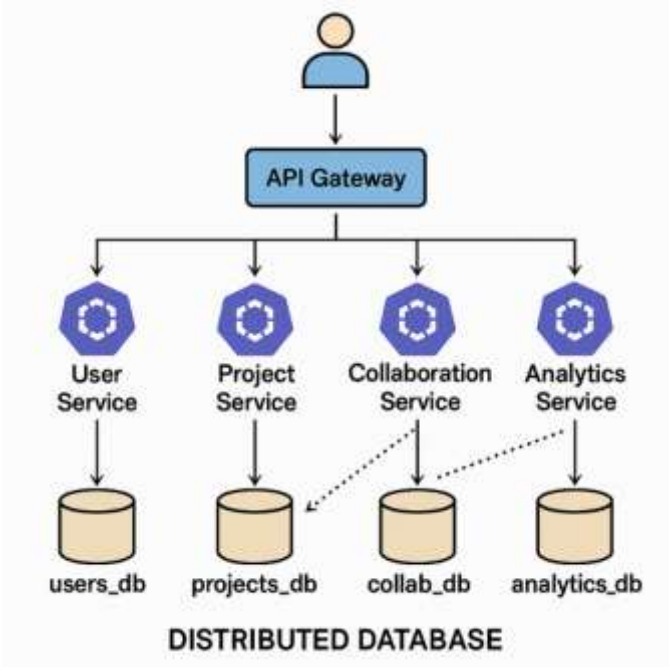


Рис.1.3 – Приклад мікросервісної архітектури з розподіленою базою даних

Розподілена система управління базами даних (РСУБД) — це централізований рівень програмного забезпечення, який керує кількома базами даних у різних місцях, роблячи розповсюдження невидимим для користувачів. Це дозволяє створювати, отримувати, оновлювати та видаляти розподілені дані, одночасно забезпечуючи синхронізацію та узгодженість у всіх місцях.

Однією з ключових переваг РСУБД є її модульна масштабованість. Розширення розподіленої системи на нові місця або організаційні підрозділи відбувається з мінімальними збоями — зазвичай нові вузли просто додаються та підключаються до системи. Розподілені бази даних також забезпечують більшу надійність, оскільки компоненти системи можуть продовжувати працювати, навіть якщо одне місце виходить з ладу. Розміщуючи дані, до яких часто звертаються, ближче до потрібного місця, розподілені системи також можуть зменшити витрати на зв’язок і покращити час відповіді на запити порівняно з централізованими базами даних.

Однак РСУБД також викликають проблеми. Їм потрібне складне та часто дороге програмне забезпечення для забезпечення прозорості, координації та узгодженості в багатьох місцях. Підтримка цілісності реплікованих або спільних даних спричиняє значні витрати на обробку, особливо під час оновлень або синхронізації. Крім того, невідповідний розподіл даних може вплинути на продуктивність системи через низький час відгуку та неефективні операції. Незважаючи на ці недоліки, переваги гнучкості, відмовостійкості та продуктивності роблять РСУБД важливим вибором для великих організацій, що керуються даними.

При проєктуванні розподіленої бази даних дуже важливо розуміти наслідки теореми CAP, сформульованої Еріком Брюером. Теорема стверджує, що розподілена система не може одночасно гарантувати узгодженість, доступність і толерантність до розділів. У випадку розділу мережі, система повинна вибирати між узгодженістю та доступністю.

- Узгодженість гарантує, що всі вузли бачать одні й ті ж дані в один і той же час. Це критично важливо для систем, де цілісність даних є важливою, наприклад, для фінансових послуг.
- Доступність гарантує, що кожен запит отримає відповідь, навіть якщо деякі вузли недоступні. Це важливо для систем, орієнтованих на користувача, де час безвідмовної роботи є пріоритетом.
- Стійкість до розділення дозволяє системі продовжувати функціонувати, навіть якщо стався збій мережі, що розділяє частини системи.

У реальних умовах розподілені системи часто жертвують суворою узгодженістю на користь доступності та стійкості до розділення, застосовуючи такі методи, як реплікація даних, запис на основі кворуму або евентуальна узгодженість. У випадку нашого сервісу для взаємодії з проєктами варто надавати перевагу доступності та стійкості до розділення, щоб забезпечити швидкість реагування та відмовостійкість розподілених компонентів.

1.5 Аналіз існуючих рішень, які побудовані на мікросервісній архітектурі

Netflix є одним із найвідоміших прикладів компанії, яка здійснила перехід від монолітної архітектури до сервісно-орієнтованої архітектури (SOA). Щодня Netflix обробляє понад мільярд запитів до своїх сервісів, причому ці запити надходять з більш ніж 800 різних типів пристроїв для забезпечення потокової трансляції відео. Один запит до API Netflix у фоновій системі може породжувати до п’яти додаткових викликів до інших сервісів.

Ще одним вагомим прикладом компанії, яка перейшла до мікросервісної архітектури, є Amazon. Враховуючи масштаб і різноманітність сервісів Amazon, компанія обробляє величезну кількість запитів у своїх системах щодня. Архітектура Amazon включає не лише інтерфейси прикладного програмування (API), а й сам вебсайт, який використовує ці API. З огляду на масштаби

компанії, її стара двоярусна архітектура виявилася непридатною для обробки такої кількості запитів, що і стало причиною переходу до мікросервісів.

Ще одним прикладом є eBay — відомий аукціонний сервіс. У процесі масштабування своєї платформи компанія змінила підхід до архітектури та почала використовувати набір окремих автономних застосунків. Кожен з них реалізує бізнес-логіку окремої функціональної області, що дозволяє системі залишатися гнучкою та стійкою до змін.

ВИСНОВОК ДО РОЗДІЛУ 1

Перший розділ містить огляд сучасних підходів до побудови архітектури програмних систем, включаючи порівняння особливостей монолітної та мікросервісної архітектур. Обговорюються ключові принципи мікросервісного підходу, зокрема автономність сервісів, орієнтація на бізнес-можливості, незалежне розгортання, слабкий зв'язок та ізоляція даних. Також детально проаналізовано загальні архітектурні патерни мікросервісів: поділ за субдоменами та бізнес-функціями, використання окремої бази даних для кожного сервісу, патерни Saga, композиції API та CQRS, які забезпечують гнучкість, масштабованість та відмовостійкість системи.

Особлива увага приділяється розподіленим системам управління базами даних (СУБД) як технічній основі мікросервісної архітектури. Розкрито переваги розподілених СУБД в контексті зберігання даних, реплікації, прозорості доступу та подолання проблеми «інформаційних островів», коли дані в різних підсистемах є ізольованими та недоступними для інтеграції. Також були описані принципи побудови СУБД, їх архітектура та роль у побудові сучасних веб-сервісів.

Огляд існуючих рішень підтверджує ефективність переходу на мікросервісну архітектуру в умовах високих навантажень, великої кількості функціональних модулів та динамічного розвитку платформи.

На основі виконаного аналізу обґрунтовано вибір мікросервісного підходу для реалізації сервісу взаємодії з персональними проєктами. Така архітектура забезпечує гнучкість, масштабованість окремих модулів, розподіл обов'язків та легкість інтеграції нових функцій у майбутньому. У наступних розділах буде розглянуто дизайн системи, модулі, що входять до її складу, а також процес впровадження та тестування розробленого рішення.

РОЗДІЛ 2. ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1. Мова програмування Python

Для реалізації серверної частини системи було обрано мову Python, яка є однією з найпопулярніших мов програмування і широко використовується у веб-розробці, розробці мікросервісів, обробці даних та штучному інтелекті. У проєкті Python використовується для написання бізнес-логіки мікросервісів, взаємодії з базами даних, реалізації API, обробки подій через брокер RabbitMQ, а також для побудови системи аутентифікації та авторизації. [1, 11]

Переваги:

- Простота та читабельність коду: Python має зрозумілий синтаксис, що пришвидшує розробку, знижує поріг входження для нових членів команди та полегшує підтримку проєкту.
- Широка екосистема бібліотек: Python має великий набір бібліотек та фреймворків, які спрощують інтеграцію з базами даних (наприклад, SQLAlchemy, SQLModel), API (FastAPI), асинхронне програмування (asyncio, aio-pika), управління міграцією (Alembic) тощо.
- Асинхронна модель виконання: Python, починаючи з версії 3.5 і вище, підтримує ключові конструкції для асинхронного програмування (async/await), що особливо корисно в мікросервісних архітектурах з високою конкуренцією запитів. [11]
- Спільнота та підтримка: Python має одну з найбільших спільнот розробників у світі, що гарантує велику кількість готових рішень, документації та навчальних посібників.

Недоліки:

- Нижча продуктивність порівняно зі компільованими мовами: через інтерпретований характер Python програми можуть працювати

повільніше, ніж, наприклад, Go або C++. Однак у більшості випадків це компенсується простотою розробки та можливістю горизонтального масштабування.

- Один потік виконання: Python має глобальний інтерпретатор (GIL), що ускладнює використання багатопоточності в деяких сценаріях. Однак для операцій вводу-виводу цей недолік майже непомітний.

Таким чином, Python є оптимальним вибором для побудови серверної логіки веб-сервісу через його простоту, гнучкість і підтримку великої кількості інструментів, необхідних для розробки архітектури мікросервісу. Крім того, мій практичний комерційний досвід розробки програмного забезпечення на Python дозволив мені ефективно застосовувати перевірені підходи, шаблони та інструменти, що позитивно вплинуло на якість та масштабованість реалізованого рішення.

2.2. Фреймворк FastAPI для реалізації REST API

В рамках дипломного проєкту для побудови REST API було обрано фреймворк FastAPI, який є сучасним асинхронним веб-фреймворком для Python. FastAPI спеціально розроблений для створення високопродуктивних API та сервісів і виділяється серед своїх аналогів (таких як Flask або Django REST Framework) швидкістю роботи, підтримкою асинхронного програмування та глибокою інтеграцією з бібліотеками Pydantic та OpenAPI. [2]

FastAPI використовується в проєкті на рівні кожного мікросервісу для створення ізольованих REST-інтерфейсів, через які зовнішні або внутрішні сервіси можуть взаємодіяти з логікою модуля. Це стосується, зокрема, сервісу аутентифікації, сервісу проєкту, сервісу коментарів тощо. Завдяки простій декларативній структурі FastAPI дозволяє швидко створювати добре документовані API, автоматично генеруючи специфікацію OpenAPI, що значно полегшує тестування та інтеграцію. [2]

Ключові переваги FastAPI включають автоматичну перевірку вхідних даних, зручне декларування схеми через Pydantic і повну підтримку асинхронних обробників на основі async def, що дозволяє масштабувати навантаження і ефективно обробляти паралельні запити. FastAPI також підтримує ін'єкцію залежностей на рівні фреймворку, що підвищує модульність і тестуємість коду.

До недоліків можна віднести відносно невелику кількість навчальних посібників та рішень у порівнянні зі іншими фреймворками, а також складність налагодження для початківців через асинхронну природу запитів. Однак ці обмеження не є критичними для проєкту, особливо з огляду на наявний досвід розробки FastAPI.

FastAPI поширюється під ліцензією MIT Free License, яка дозволяє його вільне використання як у відкритих, так і в комерційних проєктах без обмежень.[2]

2.3 Docker Swarm для оркестрації

Для організації ефективної роботи архітектури мікросервісів у розподіленому середовищі в дипломному проєкті було обрано Docker Swarm як основний інструмент оркестрування. Docker Swarm - це інтегрований в Docker механізм управління кластерами, який дозволяє розгортати, масштабувати та підтримувати стабільну роботу контейнерних сервісів у виробничому середовищі. [13]

Docker Swarm забезпечує автоматичне балансування навантаження, сервіси обходу відмов та оновлення. Ці функції особливо корисні в розподіленій архітектурі мікросервісів, де кожен сервіс повинен бути автономним, але в той же час тісно взаємодіяти з іншими компонентами.

Незважаючи на свої переваги, Docker Swarm має певні обмеження. Зокрема, порівняно з Kubernetes, він менш гнучкий у налаштуванні, має обмежену екосистему плагінів і не завжди є першим вибором для великих

промислових рішень. Однак для малих і середніх проєктів, таких як цей сервіс для взаємодії з проєктами, Swarm надає все необхідне для стабільної оркестрації і контролю. [4]

Docker Swarm - це програмне забезпечення з відкритим вихідним кодом під ліцензією Apache 2.0, що дозволяє вільно використовувати його як в освітніх, так і в комерційних цілях без обмежень. [13]

2.4 RabbitMQ як брокер повідомлень для асинхронної взаємодії

Для забезпечення асинхронної взаємодії між мікросервісами був обраний RabbitMQ, надійний брокер повідомлень, який реалізує шаблон «черги повідомлень» і підтримує протокол AMQP. RabbitMQ дозволяє сервісам обмінюватися повідомленнями без необхідності прямих синхронних залежностей, роблячи архітектуру більш гнучкою, масштабованою та відмовостійкою.

RabbitMQ використовується для публікації подій, пов’язаних із діяльністю користувачів або змінами стану даних, наприклад створення або видалення коментарів. Інші мікросервіси можуть підписатися на ці події, обробляти їх у фоновому режимі та реагувати відповідним чином - наприклад, оновлення лічильників, створення історії подій або активація сповіщень. Такий підхід дозволяє уникнути жорсткого зв’язку між мікросервісами та ефективно розподілити навантаження. [6]

RabbitMQ поширюється на умовах публічної ліцензії Mozilla 2.0 (MPL 2.0), яка дозволяє безкоштовне використання в освітніх і комерційних проєктах.[6]

2.5 Розподілена система управління базами даних PostgreSQL

У проєкті використовується PostgreSQL як базову СУБД у кожному мікросервісі відповідно до шаблону Database per Service. Це означає, що кожен мікросервіс має власну ізольовану базу даних, яка зберігає лише ті сутності, які належать до відповідної бізнес-сфери. [3,8]

Розподілений характер системи забезпечується фізичним або логічним розділенням баз даних між мікросервісами, які можуть бути розташовані на різних вузлах або контейнерах. PostgreSQL має можливості реплікації, що дозволяє як горизонтальне масштабування, так і резервування даних у майбутньому. [7]

Серед недоліків PostgreSQL варто відзначити складність горизонтального масштабування з самого початку порівняно з деякими рішеннями NoSQL. Крім того, побудова дійсно розподіленої логіки транзакцій вимагає додаткових механізмів, таких як шаблон Saga або використання зовнішніх координаторів транзакцій.

PostgreSQL чудово підходить для інтеграції з програмами Python через бібліотеки ORM, такі як SQLAlchemy або SQLModel, які використовуються в цьому проєкті. [7]

2.6 SQLModel для роботи з БД. Alembic для керування міграціями

Для взаємодії з базою даних у сервісах дипломних проєктів була обрана бібліотека SQLModel, яка поєднує в собі можливості популярного фреймворку ORM SQLAlchemy з можливостями перевірки та серіалізації, властивими Pydantic. SQLModel дозволяє одночасно описувати схеми баз даних і моделі для REST API, значно спрощуючи структуру коду, зменшуючи дублювання та підвищуючи читабельність. [10, 16]

Недоліком SQLModel є відносна «молодість» проєкту: SQLModel все ще знаходиться в активній стадії розробки, має обмежену документацію та спільноту порівняно з повноцінним SQLAlchemy.

Щоб керувати схемою бази даних, а також створювати та застосовувати міграції, проєкт використовує Alembic, офіційний інструмент міграції в екосистемі SQLAlchemy. Він дозволяє відстежувати зміни в структурі моделі, автоматично генерувати сценарії міграції та застосовувати їх до різних середовищ (локального, тестового, робочого). Alembic інтегрується з SQLModel

безпосередньо через SQLAlchemy, що дозволяє зберігати повну гнучкість при зміні схеми бази даних. [9]

Alembic, у свою чергу, є стабільним і перевіреним інструментом, але має певну складність у налаштуванні, особливо в поєднанні з асинхронними моделями або нестандартними конфігураціями ORM.

SQLModel ліцензується згідно з ліцензією MIT, яка дозволяє безкоштовне використання, модифікацію та розповсюдження навіть у комерційних проєктах без необхідності розкривати вихідний код. Alembic поширюється за ліцензією BSD, яка також є відкритою та сумісною з ліцензією PostgreSQL. [9]

2.7. Poetry як система управління залежностями та пакетування

У дипломному проєкті використовується інструмент Poetry для керування залежностями, створення віртуального середовища та генерування конфігурацій.

Поезія була обрана через потребу в зручному та надійному інструменті, що дозволяє централізовано контролювати залежності всіх мікросервісів проєкту. На відміну від класичного підходу з requirements.txt, Poetry зберігає декларативний файл pyproject.toml і автоматично згенерований poetry.lock, який забезпечує блокування версій усіх пакетів і відтворюваність середовища. Це особливо важливо для підтримки стабільності проєкту в архітектурі мікросервісів, де кожен сервіс має власні залежності.

Poetry також інтегрується з системами CI/CD і дозволяє легко створювати пакети, публікувати та керувати версіями, що буде корисно в майбутньому розвитку системи як платформи. [12]

До недоліків Poetry можна віднести відносно великий початковий розмір інсталяції та певні обмеження при роботі з нестандартними пакетами Python або спеціальними структурами проєкту. Крім того, іноді виникають труднощі під час інтеграції з Docker у багатоетапних збірках.

Poetry поширюється за відкритою ліцензією MIT, яка дозволяє вільне використання у відкритих і комерційних проєктах без будь-яких обмежень на модифікацію чи розповсюдження. [12]

ВИСНОВОК ДО РОЗДІЛУ 2

У цьому розділі детально розглянуто основні технології, які використовуються для реалізації веб-сервісу на основі архітектури мікросервісів. Кожен з них був обраний з урахуванням вимог масштабованості, надійності, простоти розробки та підтримки.

Мова Python стала основою для логіки на стороні сервера завдяки своїй гнучкості та великій екосистемі бібліотек. FastAPI дозволяє створювати швидкі, асинхронні API, що особливо важливо в контексті взаємодії мікросервісів.

Docker і Docker Swarm забезпечують легку в налаштуванні оркестрацію контейнерів, спрощуючи розгортання та масштабування компонентів системи. RabbitMQ використовується для асинхронного зв’язку між мікросервісами, забезпечуючи надійну доставку повідомлень.

Зберігання та обробка даних реалізовано через PostgreSQL, розподілену СУБД, яка забезпечує високу цілісність і масштабованість. SQLModel і Alembic спрощують роботу з базами даних і керування міграцією. Poetry використовується для контролю залежностей і структурування проєкту.

Загалом обрані технології добре взаємодіють між собою та утворюють надійну інфраструктуру, що відповідає сучасним вимогам розробки розподілених програмних систем.

РОЗДІЛ 3. ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА РОЗРОБКА СИСТЕМИ

3.1 Загальна архітектура системи

Розроблена система використовує мікросервісну архітектуру, де кожна основна функціональна область реалізована як незалежний сервіс, що розгортається незалежно. Такий архітектурний вибір забезпечує модульну розробку, спрощує масштабування і полегшує довгострокове обслуговування. Кожен мікросервіс інкапсулює окремий набір бізнес-логіки та взаємодіє з іншими сервісами через RESTful API або асинхронний обмін повідомленнями.[3, 5]

Обґрунтування використання мікросервісів:

- Шлюз API (api-gateway) - слугує єдиною точкою входу для всіх зовнішніх клієнтських запитів. Займається маршрутизацією, генерацією та декодуванням JWT токенів, спільним використанням ресурсів різних джерел (CORS) і тротлінгом вхідних запитів. Він перенаправляє вхідні запити до відповідного бекенд-сервісу на основі шаблонів URL-адрес і методів HTTP. Такий дизайн архітектури спрощує взаємодію з клієнтом і централізує наскрізні проблеми.
- Сервіс користувачів (user-service) - відповідає за реєстрацію, автентифікацію, керуванням профілю та верифікацію користувачів за поштою. Він надійно зберігає облікові дані користувача, а також дозволяє створювати команди та додавати інших користувачів, як учасників команди.
- Сервіс проєктів (project-service) - керує операціями, пов'язаними з проектами, включаючи створення, модифікацію та пошук. Він також підтримує право власності на проект, метадані та пов'язані з ними бізнес-правила.

- Сервіс коментарів (comment-service) - керує операціями, пов'язаними з коментарями до проєкту. Дозволяє користувачам створювати, читати і змінювати коментарі, забезпечуючи при цьому належний зв'язок між користувачами і проектами.
- Сервіс заявок (application-service) - керує заявками користувачів або запитами на участь в проєкті. Дозволяє створювати заявку, відкликати, підтверджувати і відхиляти з прикріпленням коментаря.

Архітектурні рішення:

- Ізоляція сервісів: кожен мікросервіс зберігає ексклюзивний контроль над своєю базою даних, підвищуючи цілісність даних та підтримуючи незалежну розробку та розгортання.
- Модель шлюзу API: централізація функцій маршрутизації та безпеки в шлюзі API спрощує зовнішній доступ до API і стандартизує автентифікацію та застосування політик.
- Асинхронний обмін повідомленнями: використання RabbitMQ дозволяє сервісам взаємодіяти без прямих залежностей. Це підтримує кінцеву узгодженість і покращує масштабованість системи.[6]
- Кешування та обмеження швидкості: Redis використовується для мінімізації затримки відповіді та захисту сервісів від надмірного трафіку. Він відіграє ключову роль у забезпеченні швидкої та стабільної роботи користувачів.[8]
- База даних для кожного сервісу: принцип «база даних на сервіс» забезпечує суворе розмежування даних. Кожен мікросервіс має власну схему і логіку збереження даних, що дозволяє уникнути проблем, пов'язаних із загальною базою даних.

3.2 Взаємодія мережі та сервісів

У цьому проекті всі взаємодії з клієнтами централізовані через шлюз API, який служить центральною точкою доступу до платформи. Для вхідних запитів шлюз визначає відповідний мікросервіс на основі URL-адреси та перенаправляє їх всередину, гарантуючи, що лише дійсні та авторизовані запити досягають внутрішніх сервісів. Це архітектурне рішення спрощує логіку на стороні клієнта та забезпечує узгоджений контроль доступу для всіх сервісів. [2]

Щоб забезпечити безпечну та ефективну взаємодію між сервісами, усі компоненти, включаючи мікросервіси, бази даних і системи обміну повідомленнями, розгортаються в накладній мережі, якою керує Docker Swarm. Ця мережа, яку зазвичай називають внутрішньою мережею, забезпечує зв’язок між мікросервісами за допомогою передбачуваних імен на основі DNS (наприклад, user-service, project-service) без необхідності вручну налаштовувати IP-адресу. Лише шлюз API доступний зовнішньому світу, а всі інші мікросервіси залишаються ізольованими у внутрішній мережі. Це зменшує потенційну поверхню атаки та підвищує загальну безпеку. [4, 13]

Між сервісами використовуються дві моделі зв’язку: синхронна та асинхронна. Для критично важливих операцій, таких як операції CRUD, синхронний зв’язок HTTP реалізується через RESTful API. Однак для фонових завдань, сповіщень і слабозв’язаних процесів асинхронний зв’язок реалізується за допомогою RabbitMQ. Цей брокер повідомлень дозволяє сервісам публікувати події та підписуватися на них (наприклад, application.withdrawn, comment.created). Це дає змогу масштабувати керовані подіями робочі процеси та покращує відмовостійкість.

Кожен мікросервіс також керує власним екземпляром бази даних PostgreSQL відповідно до схеми «база даних на сервіс». Цей підхід сприяє ізоляції даних і автономності сервісу, дозволяючи кожній команді або сервісу розробляти свою схему незалежно без ризику конфліктів даних між сервісами. Загалом така архітектура забезпечує безпечне, придатне для обслуговування та

масштабоване середовище для надійної роботи платформи під різними навантаженнями та з функціональними розширеннями.

3.3 Інфраструктура оркестрації

Система використовує Docker Swarm як рівень оркестрації для керування та розгортання контейнерів у кластері хостів Docker. Swarm дозволяє обробляти кілька вузлів Docker як єдину логічну одиницю, підтримуючи високу доступність, виявлення сервісів, балансування навантаження та автоматизацію життєвого циклу контейнерів. [4, 13]

Кожен мікросервіс та компонент інфраструктури, як наприклад API gateway, user-service, project-service, RabbitMQ, Redis та окремі бази даних PostgreSQL, розгортаються як незалежний стек, визначений у власному файлі «docker-compose.yml». Стеки — це автономні групи сервісів, мереж і томів, які можна незалежно запускати та керувати ними за допомогою команди розгортання стека докерів. Цей модульний підхід покращує ізоляцію помилок і дозволяє окремим командам розробляти, оновлювати та масштабувати свої сервіси, не впливаючи на інших. Ізоляція між стеками зберігається, але зв’язок між ними продовжується через спільні накладені мережі.

Docker Swarm також надає потужні інтегровані інструменти для масштабування, моніторингу та керування. Сервіси в стеку можна збільшити або зменшити за допомогою docker service scale, а стан системи можна відстежувати за допомогою таких команд, як docker stack ls, docker service ps і docker node ls. Docker Swarm також підтримує постійні оновлення та відкат, що забезпечує безпечне розгортання з мінімальним простоєм. [13]

Варто зазначити, що вся система об’єднана через загальну накладену мережу під назвою «backend». Також, розроблено скрипт розгортання deploy_all.sh для автоматизації послідовного запуску, вміст якого зображено на рисунку 3.1.

deploy_all.sh

```
#!/bin/bash

set -e

docker info | grep -q "Swarm: active" || docker swarm init

docker network ls | grep -q backend || docker network create --driver overlay backend

echo "🔨 Building service images..."
docker build -t user-service ./user-service
docker build -t project-service ./project-service
docker build -t comment-service ./comment-service
docker build -t application-service ./application-service
docker build -t api-gateway-service ./api-gateway-service

echo "🐇 Deploying RabbitMQ..."
docker stack deploy -c rabbitmq/docker-compose.yml messaging

echo "👤 Deploying user-service..."
docker stack deploy -c user-service/docker-compose.yml user

echo "📁 Deploying project-service..."
docker stack deploy -c project-service/docker-compose.yml project

echo "💬 Deploying comment-service..."
docker stack deploy -c comment-service/docker-compose.yml comment

echo "📄 Deploying application-service..."
docker stack deploy -c application-service/docker-compose.yml application

echo "🌐 Deploying api-gateway-service..."
docker stack deploy -c api-gateway-service/docker-compose.yml gateway

echo "✅ All services built and deployed successfully."
```

Ctrl+L to chat, Ctrl+K to generate

Рисунок 3.1 – Вміст файлу deploy_all.sh

Незважаючи на те, що кожен стек розгортається окремо, мікросервіси безперебійно з'єднані, щоб утворити єдину платформу. Ця організована інфраструктура гарантує, що платформа залишається модульною, стійкою та зручною для обслуговування, одночасно забезпечуючи автономні командні робочі процеси та послідовну поведінку системи.

3.4 Організація структури сервісів

Усі мікросервіси на платформі мають модульну та узгоджену структуру, що забезпечує чіткий розподіл обов'язків і незалежний розвиток. Кожен сервіс розташований у власному каталозі та відповідає за певну сферу діяльності — наприклад, керування користувачами, проектами чи коментарями. Внутрішня структура стандартизована з такими директоріями, як /app/ для вихідного коду, /tests/ для тестових випадків і файлами конфігурації, такими як Dockerfile, .env і docker-compose.yml.

3.4.1 API шлюз (api-gateway-service)

Структура директорії:

```
api-gateway-service/
│
├── app/
│   ├── __init__.py
│   ├── main.py            # Точка входу у FastAPI-додаток
│   ├── routers/           # Маршрути API (наприклад, gateway.py)
│   ├── middleware/        # Middleware (наприклад, auth, ratelimit)
│   ├── gateway/           # Основна логіка шлюзу (наприклад, proxy.py)
│   ├── core/              # Основні утиліти (config, security, logging)
│   └── api/               # API-специфічна логіка (наприклад, api.py)
│
├── tests/                 # Юніт-тести на базі Pytest
│   ├── test_main.py
│   ├── conftest.py
│   └── __init__.py
│
├── poetry.lock            # Файл блокування залежностей Poetry
├── pyproject.toml         # Конфігурація проєкту для Poetry
├── docker-compose.yml     # Оркестрація сервісу
├── Dockerfile             # Інструкції для збирання контейнера
└── README.md              # Документація до сервісу
```

Структура коду, конфігурації та запуску.

Основним файлом програми є app/main.py, який ініціалізує програму FastAPI, реєструє маршрутизатори та налаштовує проміжне програмне забезпечення.

Всі ендпоінти API організовані в каталозі app/routers/. Наприклад, gateway.py визначає основні маршрути шлюзу.

Кастомне проміжне ПЗ для автентифікації та тротлінгу знаходиться в app/middleware/ (auth.py, ratelimit.py).

Каталог app/gateway/ містить основну логіку проксі-сервера та агрегації - proxy.py.

Каталог app/core/ містить конфігурацію (config.py), утиліти безпеки (security.py) та конфігурацію ведення журналів (log_config.yaml).

Каталог app/api/ містить додаткову логіку або схеми API.

Файли pyproject.tml та poetry.lock керують залежностями Python через Poetry. Змінні оточення завантажуються з файлу .env (посилання на який міститься в docker-compose.yml). Логування налаштовується через app/core/log_config.yml.

Всі тести знаходяться в каталозі tests/, з використанням Pytest, з принаймні одним основним тестовим файлом і файлом conftest.py для виправлень. Результат модульного тестування мікросервісу наведений в рисунку 3.2.

Рисунок 3.2 – Скріншот результату модульного тестування сервісу api-gateway-service

Вміст файлу api-gateway-service/docker-compose.yml наведений нижче:

```
services:
  api-gateway:
    image: api-gateway-service:latest
```

```
    command: poetry run uvicorn app.main:app --host 0.0.0.0 --port 8000 --log-
                config app/core/log_config.yaml
    env_file:
      - .env
    volumes:
      - ./app/log:/code/app/log
    networks:
      - backend
    ports:
      - "8080:8000"
    deploy:
      replicas: 1
      resources:
        limits:
          memory: 512M
          cpus: '0.5'
        reservations:
          memory: 256M
          cpus: '0.2'

networks:
  backend:
    external: true
```

Опис команд з файлу оркестрації мікросервісу:

- Запускає сервіс за допомогою Uvicorn + Poetry.
- Завантажує змінні середовища з .env.
- Монтує каталог журналу для постійного журналювання.

- Відкриває порт 8080 (хост), зіставлений з 8000 (контейнер).
- Підключається до внутрішньої мережі Docker для зв’язку між сервісами.
- Встановлює обмеження ресурсів і резервування пам’яті та ЦП.

3.4.2 Сервіс керування користувачами (user-service)

Структура директорії:

```
user-service/
│
├── app/
│   ├── main.py           # Точка входу для FastAPI-додатку
│   ├── users/    # Логіка користувачів (маршрути, сервіси, схеми, моделі)
│   ├── teams/    # Логіка команд (маршрути, сервіси, схеми, моделі)
│   ├── core/     # Основні утиліти
│   └── api/      # Залежності API та основна логіка API
│
├── tests/            # Unit-тести з використанням Pytest
│   ├── users/            # Тести для функціоналу користувачів
│   └── teams/            # Тести для функціоналу команд
│
├── replication/          # Скрипти і конфіги налаштувань реплікації
│
├── alembic/              # Скрипти міграцій бази даних
│
├── poetry.lock           # Файл блокування залежностей Poetry
├── pyproject.toml        # Конфігурація проєкту Poetry
├── docker-compose.yml    # Оркестрація сервісу
├── Dockerfile            # Інструкції для створення контейнера
```

```
├── alembic.ini            # Конфігураційний файл Alembic
├── .dockerignore          # Правила ігнорування Docker
└── README.md              # Документація до сервісу (якщо наявна)
```

Структура коду, конфігурації та запуску.

Основним файлом програми є app/main.py, який ініціалізує програму FastAPI, включає маршрутизатори для користувачів і команд, а також налаштовує проміжне програмне забезпечення і конфігурацію.

Користувачі app/users/ та команди app/teams/, кожен з яких містить:

- router.py: Визначення маршрутів API для користувачів/команд.
- service.py: Бізнес-логіка для користувачів/команд.
- schemas.py: Підантичні моделі для перевірки запитів/відповідей.
- models.py: ORM-моделі для таблиць бази даних.

Основні утиліти знаходяться у каталозі app/core/:

- config.py: Управління конфігурацією (env, налаштування).
- db.py: Логіка підключення до бази даних.
- security.py: Утиліти безпеки (наприклад, хешування паролів, JWT).
- email_service.py: Логіка надсилання електронної пошти.
- log_config.yaml: Конфігурація ведення журналу.
- replication.py: Утиліти для підтримки реплікації.

Логіка API міститься в каталозі app/api/:

- main.py: Можливо, версії API або логіка кореневого API.
- deps.py: Впровадження залежностей для маршрутів.

Директорія alembic/ та файл alembic.ini використовуються для керування міграціями таблиць баз даних. Файли pyproject.tml та poetry.lock керують залежностями Python через Poetry. Змінні оточення завантажуються з файлу .env (посилання на який міститься в docker-compose.yml). Логування

налаштовується через app/core/log_config.yml. Всі тести знаходяться в каталозі tests/ за допомогою Pytest. Тести організовано за функціями (users/, teams/), кожна з яких має власні тестові файли. Результат модульного тестування мікросервісу наведений в рисунку 3.3.

Рисунок 3.3 – Скріншот результату модульного тестування сервісу user-service

Вміст файлу user-service/docker-compose.yml наведений нижче:

```
services:
  user-db:
    image: postgres:15
    env_file:
      - .env
    volumes:
      - user_data:/var/lib/postgresql/data
      - ./replication/master/postgresql.conf:/etc/postgresql/postgresql.conf
      - ./replication/master/pg_hba.conf:/var/lib/postgresql/pg_hba_custom.conf
    networks:
      - backend
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
```

```
      retries: 5

  user-db-replica:
    image: postgres:15
    env_file:
      - .env
    depends_on:
      - user-db
    volumes:
      - user_replica_data:/var/lib/postgresql/data
      - ./replication/replica:/var/lib/postgresql/conf
      - ./replication/init-replica.sh:/docker-entrypoint-initdb.d/init-replica.sh
    entrypoint: /bin/bash
    command: /docker-entrypoint-initdb.d/init-replica.sh
    networks:
      - backend
    ports:
      - "5433:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 15s
      timeout: 5s
      retries: 10

  user-service:
    image: user-service
    command: poetry run uvicorn app.main:app --host 0.0.0.0 --port 8000 --log-
config app/core/log_config.yaml
    volumes:
```

```
      - ./alembic/versions:/code/alembic/versions
      - ./app/log:/code/app/log
    ports:
      - "8001:8000"
    env_file:
      - .env
    networks:
      - backend
    depends_on:
      - user-db
      - user-db-replica

volumes:
  user_data:
  user_replica_data:

networks:
  backend:
    external: true
```

Опис команд з файлу оркестрації мікросервісу:

- Запускає сервіс за допомогою Uvicorn + Poetry.
- Завантажує змінні середовища з .env.
- Монтує каталоги для міграцій і журналів Alembic.
- Розкриває порт 8001 (хост), зіставлений з 8000 (контейнер).
- Підключається до внутрішньої мережі Docker для зв’язку між сервісами.

- Залежить від сервісу user-db (PostgreSQL 15), який використовує іменований том Docker для постійного зберігання.

3.4.3 Сервіс керування проєктами (project-service)

Структура директорії:

```
project-service/
│
├── app/
│   ├── main.py              # Точка входу для FastAPI-додатку
│   ├── projects/            # Логіка керування проєктами
│   ├── messaging/           # Обробники повідомлень/подій
│   ├── core/                # Основні утиліти
│   └── api/                 # Залежності API та основна логіка API
│
├── tests/                   # Unit-тести з використанням Pytest
│   └── projects/            # Тести для функціоналу проєктів
│
├── alembic/                 # Скрипти міграцій бази даних
│
├── poetry.lock              # Файл блокування залежностей Poetry
├── pyproject.toml           # Конфігурація проєкту Poetry
├── docker-compose.yml       # Оркестрація сервісу
├── docker-compose.elasticsearch.yml  # Оркестрація Elasticsearch
├── Dockerfile               # Інструкції для створення контейнера
├── alembic.ini              # Конфігураційний файл Alembic
└──.dockerignore             # Правила ігнорування Docker
```

Структура коду, конфігурації та запуску.

Основним записом програми є app/main.py, який ініціалізує програму FastAPI, включає маршрутизатори для проектів і налаштовує проміжне програмне забезпечення та конфігурацію.

Логіка керування проєктами в app/projects, директорія містить:

- router.py: визначення маршрутів API для проектів.
- service.py: бізнес-логіка для проєктів.
- search_service.py, search_models.py: Логіка пошуку та індексування з використанням Elasticsearch.
- image_service.py: логіка обробки зображень для проектів.
- schemas.py: Pydantic моделі для перевірки запитів/відповідей.
- models.py: моделі ORM для таблиць бази даних.

Роботу з подіями реалізовано в app/messaging/. Директорія містить comment_consumer.py для обробки подій/повідомлень коментарів.

Основні утиліти знаходяться в директорії app/core/, яка містить:

- config.py: керування конфігурацією (env, налаштування).
- db.py: логіка підключення до бази даних.
- elasticsearch.py: клієнт і помічники Elasticsearch.
- log_config.yaml: Конфігурація логування.

Логіка API знаходиться в директорії app/api/, яка містить:

- main.py: можливо, версії API або коренева логіка API.
- deps.py: впровадження залежностей для маршрутів.

Директорія alembic/ та файл alembic.ini використовуються для керування міграціями таблиць баз даних. Файли pyproject.toml і poetry.lock керують залежностями Python через Poetry. Змінні середовища завантажуються з файлу .env (на який посилається файл docker-compose.yml). Логування налаштовується за допомогою app/core/log_config.yaml. Усі тести знаходяться в

каталозі tests/ за допомогою Pytest. Тести впорядковано за функціями (projects/), кожна з яких має власні тестові файли. Результат модульного тестування мікросервісу наведений в рисунку 3.4.

Рисунок 3.4 – Скріншот результату модульного тестування сервісу project-service

Вміст файлу project-service/docker-compose.yml наведений нижче:

```
services:
  project-service:
    image: project-service
    command: poetry run uvicorn app.main:app --host 0.0.0.0 --port 8000 --log-
                config app/core/log_config.yaml
    volumes:
      - ./alembic/versions:/code/alembic/versions
      - ./app/log:/code/app/log
    ports:
      - "8002:8000"
    env_file:
      - .env
    networks:
      - backend
    depends_on:
      - project-db
      - rabbitmq
```

```
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

  project-db:
    image: postgres:15
    env_file:
      - .env
    volumes:
      - project_data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

volumes:
  project_data:

networks:
  backend:
    external: true
```

Опис команд з файлу оркестрації мікросервісу:

- Запускає сервіс за допомогою Uvicorn + Poetry.
- Завантажує змінні середовища з .env.

- Монтує каталоги для міграцій і журналів Alembic.
- Розкриває порт 8002 (хост), зіставлений з 8000 (контейнер).
- Підключається до внутрішньої мережі Docker для зв’язку між сервісами.
- Залежить від сервісу project-db (PostgreSQL 15) і rabbitmq для обміну повідомленнями. Сервіс бази даних використовує іменований том Docker для постійного зберігання.

Мікросервіс керування проєктами інтегрує Elasticsearch, надаючи розширені можливості повнотекстового пошуку, які значно покращують виявлення проектів на платформі. Інтеграція має багаторівневу архітектуру: базовий рівень (elasticsearch.py) керує стабільними з’єднаннями та операціями індексування, сервісний рівень (search_service.py) обробляє індексування, запити та логіку пропозицій, а рівень API забезпечує кінцеві точки RESTful (/projects/search, /projects/suggest, /projects/reindex). Програмний код ендпоінту /projects/search зображено на рисунку 3.5. Ця модульна структура забезпечує чіткий розподіл проблем і робить систему придатною для обслуговування та розширення.

```python
@router.get("/search", response_model=SearchResponse)
async def search_projects(
    q: str = Query(description="Search query"),
    offset: int = Query(default=0, ge=0, description="Search result offset"),
    limit: int = Query(default=20, ge=1, le=100, description="Maximum results to return"),
    sort_by: Optional[str] = Query(default=None, description="Field to sort by"),
    sort_order: Optional[str] = Query(default="desc", regex="^(asc|desc)$", description="Sort order"),
    user_id: Optional[str] = Query(default=None, description="Filter by user ID")
):
    """Search projects using Elasticsearch"""
    search_query = SearchQuery(
        query=q,
        offset=offset,
        limit=limit,
        sort_by=sort_by,
        sort_order=sort_order,
        user_id=user_id
    )

    return await search_service.search_projects(search_query)
```

Рисунок 3.5 – Програмний код ендпоінту /projects/search

3.4.4 Сервіс керування коментарями (comment-service)

Структура директорії:

```
comment-service/
│
├── app/
│   ├── main.py              # Точка входу для FastAPI-додатку
│   ├── comments/            # Логіка керування коментарями
│   ├── messaging/       # Відправлення повідомлень/подій
│   ├── core/       # Основні утиліти (конфігурація, база даних, логування)
│   └── api/                 # Залежності API та основна логіка API
│
├── tests/                  # Unit-тести з використанням Pytest
│   └── comments/            # Тести для функціоналу коментарів
│
├── alembic/                # Скрипти міграцій бази даних
│
├── poetry.lock             # Файл блокування залежностей Poetry
├── pyproject.toml          # Конфігурація проєкту Poetry
├── docker-compose.yml        # Оркестрація сервісу
├── Dockerfile              # Інструкції для створення контейнера
├── alembic.ini             # Конфігураційний файл Alembic
├── .dockerignore           # Правила ігнорування Docker
└── README.md               # Документація до сервісу (якщо наявна)
```

Структура коду, конфігурації та запуску.

Основним записом програми є app/main.py, який ініціалізує програму FastAPI, включає маршрутизатори для коментарів і налаштовує проміжне програмне забезпечення та конфігурацію.

Логіка керування коментарями знаходиться в app/comments/, яка містить:

- router.py: визначення маршрутів API для коментарів.
- service.py: бізнес-логіка для коментарів.
- schemas.py: Pydantic моделі для перевірки запитів/відповідей.
- models.py: моделі ORM для таблиць бази даних.

Директорія app/messaging/ містить виробників producer.py для зв’язку між мікросервісами, для надсилання подій/повідомлень іншим мікросервісам через RabbitMQ.

Основні утиліти знаходяться в каталозі app/core/, який містить:

- config.py: керування конфігурацією (env, налаштування).
- db.py: логіка підключення до бази даних.
- log_config.yaml: Конфігурація логування.

Логіка API знаходиться в каталозі app/api/, який містить:

- main.py: можливо, версії API або коренева логіка API.
- deps.py: впровадження залежностей для маршрутів.

Директорія alembic/ і файл alembic.ini використовуються для керування міграціями схем бази даних. Файли pyproject.toml і poetry.lock керують залежностями Python через Poetry. Змінні середовища завантажуються з файлу .env (на який посилається файл docker-compose.yml). Логування налаштовується за допомогою app/core/log_config.yaml. Усі тести знаходяться в каталозі tests/ і запускаються за допомогою Pytest. Результат модульного тестування мікросервісу наведений в рисунку 3.6.

Рисунок 3.6 – Скріншот результату модульного тестування сервісу comment-service

Вміст файлу comment-service/docker-compose.yml наведений нижче:

```
services:
  comment-service:
    image: comment-service
    command: poetry run uvicorn app.main:app --host 0.0.0.0 --port 8000 --log-
                config app/core/log_config.yaml
    volumes:
      - ./alembic/versions:/code/alembic/versions
      - ./app/log:/code/app/log
    ports:
      - "8003:8000"
    env_file:
      - .env
    networks:
      - backend
    depends_on:
      - comment-db
      - rabbitmq
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

  comment-db:
    image: postgres:15
    env_file:
      - .env
    volumes:
      - comment_data:/var/lib/postgresql/data
```

```
    networks:
      - backend
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

volumes:
  comment_data:

networks:
  backend:
    external: true
```

Опис команд з файлу оркестрації мікросервісу:

- Запускає сервіс за допомогою Uvicorn + Poetry.
- Завантажує змінні середовища з .env.
- Монтує каталоги для міграцій і журналів Alembic.
- Розкриває порт 8003 (хост), зіставлений з 8000 (контейнер).
- Підключається до внутрішньої мережі Docker для зв’язку між сервісами.
- Залежить від сервісу comment-db (PostgreSQL 15) і rabbitmq для обміну повідомленнями. Сервіс бази даних використовує іменований том Docker для постійного зберігання.
- Включає політику розгортання та перезапуску для стійкості.

3.4.5 Сервіс керування заявками (application-service)

Структура директорії:

```
application-service/
│
├── app/
│   ├── main.py              # Точка входу для FastAPI-додатку
│   ├── applications/        # Логіка керування заявками
│   ├── messaging/           # Видавці повідомлень/подій
│   ├── core/       # Основні утиліти (конфігурація, база даних, логування)
│   └── api/                 # Залежності API та основна логіка API
│
├── tests/                  # Unit-тести з використанням Pytest
│   └── applications/        # Тести для функціоналу заявок
│
├── alembic/                # Скрипти міграцій бази даних
│
├── poetry.lock             # Файл блокування залежностей Poetry
├── pyproject.toml          # Конфігурація проєкту Poetry
├── docker-compose.yml      # Оркестрація сервісу
├── Dockerfile              # Інструкції для створення контейнера
├── alembic.ini             # Конфігураційний файл Alembic
├── .dockerignore           # Правила ігнорування Docker
└── README.md               # Документація до сервісу (якщо наявна)
```

Структура коду, конфігурації та запуску.

Основним записом програми є app/main.py, який ініціалізує програму FastAPI, включає маршрутизатори для програм і налаштовує проміжне програмне забезпечення та конфігурацію.

Логіка керування заявками знаходиться в директорії app/applications/, яка містить:

- router.py: визначення маршрутів API для програм.
- service.py: бізнес-логіка для програм.
- schemas.py: Pydantic моделі для перевірки запитів/відповідей.
- models.py: моделі ORM для таблиць бази даних.

Каталог app/messaging/ містить publisher.py для надсилання подій/повідомлень іншим сервісами (наприклад, через RabbitMQ).

Основні утиліти:

Каталог app/core/ містить:

config.py: керування конфігурацією (env, налаштування).

db.py: логіка підключення до бази даних.

log_config.yaml: конфігурація логування.

Логіка API знаходиться в каталог app/api/, який містить:

- main.py: можливо, версії API або коренева логіка API.
- deps.py: впровадження залежностей для маршрутів.

Директорія alembic/ і файл alembic.ini використовуються для керування міграціями таблиць бази даних.

Файли pyproject.toml і poetry.lock керують залежностями Python через Poetry. Змінні середовища завантажуються з файлу .env (на який посилається файл docker-compose.yml). Логування налаштовується за допомогою app/core/log_config.yaml.

Усі тести знаходяться в каталозі tests/ за допомогою Pytest. Тести впорядковано за функціями (applications/), кожна з яких має власні тестові файли. Результат модульного тестування мікросервісу наведений в рисунку 3.7.

Рисунок 3.7 – Скріншот результату модульного тестування сервісу application-service

Вміст файлу application-service/docker-compose.yml наведений нижче:

```
services:
  application-service:
    image: application-service
    command: poetry run uvicorn app.main:app --host 0.0.0.0 --port 8000 --log-
                config app/core/log_config.yaml
    volumes:
      - ./alembic/versions:/code/alembic/versions
      - ./app/log:/code/app/log
    ports:
      - "8004:8000"
    env_file:
      - .env
    depends_on:
      - application-db
    networks:
      - backend
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
```

```
  application-db:
    image: postgres:15
    env_file:
      - .env
    volumes:
      - application_db_data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

volumes:
  application_db_data:

networks:
  backend:
    external: true
```

Опис команд з файлу оркестрації мікросервісу:

- Запускає мікросервіс за допомогою Uvicorn + Poetry.
- Завантажує змінні середовища з .env.
- Монтує каталоги для міграцій і журналів Alembic.
- Розкриває порт 8004 (хост), зіставлений з 8000 (контейнер).
- Підключається до внутрішньої мережі Docker для зв’язку між сервісами. Залежить від сервісу бази даних програми (PostgreSQL 15)

для постійного зберігання. Включає політику розгортання та перезапуску для стійкості.

3.5 Реалізація розподіленої СУБД

Архітектура розподіленої системи управління базами даних (СУБД) у цьому проєкті реалізована на основі мікросервісної моделі, де кожен сервіс відповідає за власний підмножину даних і взаємодіє з іншими сервісами через асинхронні повідомлення. У системі реалізована реплікація бази даних користувачів, а саме - балансування запитів запису в мастер-базу, а запитів читання в базу-репліку. Загалом, всі дані розділені горизонтально за доменами (наприклад, user-service має власну БД користувачів, а project-service — БД проєктів). Це дозволяє кожному мікросервісу масштабуватися незалежно й оптимізувати зберігання під конкретні задачі.

Redis інтегровано як сховище даних у пам’яті, щоб забезпечити швидкий доступ до даних сеансу, тимчасових станів і проміжних результатів. Наприклад, коли користувач створює коментар до проєкта, система виконує декілька скоординованих дій: перевіряє користувача, перевіряє доступність проекту, створює коментар та оновлює відповідну статистику. Усі проміжні кроки та їхні статуси успіху чи невдачі зберігаються в Redis, щоб забезпечити швидке відновлення або відкат, якщо щось піде не так під час процесу.

RabbitMQ служить основою для асинхронного зв’язку між сервісами. Кожен сервіс публікує та підписується на черги повідомлень відповідно до її доменних обов’язків. Наприклад, сервіс коментарів і сервіс заявок мають окремі черги для доставки повідомлень для створення коментарів або подання заявки. Навіть якщо мікросервіс відчуває тимчасовий збій, повідомлення залишаються в черзі та доставляються, коли сервіс знову стає доступним. Цей механізм покращує стійкість системи до розділення, забезпечуючи продовження роботи навіть під час тимчасових збоїв або нестабільності мережі.

Система сфокусована на доступності та толерантності до розділення, навмисно жертвуючи строгими гарантіями узгодженості, щоб дотримуватися компромісів теореми CAP. У межах однієї послуги узгодженість забезпечується оптимістичними механізмами блокування. Наприклад, під час оновлення лічильника коментарів проєкту програма порівнює номери версій, щоб виявити конфліктні оновлення. Такий підхід гарантує, що якщо відбудуться одночасні зміни, лише одне оновлення буде успішним, таким чином частково уникаючи неузгодженості даних.

3.6 Моніторинг, діагностика та логування

Кожен сервіс, включаючи допоміжні компоненти, такі як бази даних, Redis і RabbitMQ, має перевірку працездатності на рівні Docker. Ці перевірки справності регулярно перевіряють, чи контейнер працює належним чином, часто за допомогою ping /health ендпоінтів або за допомогою діагностичних команд. Сервіси, які не пройшли ці перевірки, можуть бути автоматично перезапущені Docker або позначені інструментами оркестрації для підтримки надійності системи.

Модуль monitor.py — це діагностичний сервіс на основі FastAPI для моніторингу працездатності та робочого стану ключових компонентів інфраструктури, зокрема PostgreSQL, Redis і RabbitMQ. Після запуску він ініціалізує постійні підключення до Redis, PostgreSQL, RabbitMQ і перевіряє їх доступність. Сервіс надає кінцеві точки REST, які надають звіти про працездатність у реальному часі (/health), детальну статистику бази даних (/database/stats) і зведені системні показники (/metrics).

Логіка моніторингу описана в класі DatabaseMonitor, який отримує та повертає структуровані показники, такі як кількість підключень до бази даних, активні запити, використання пам’яті в Redis. Ендпоінт /metrics об’єднує кілька перевірок служб в єдиний консолідований знімок стану, що дозволяє легко інтегрувати його в інформаційні панелі або системи оповіщення. Логіку

ендпоінту /metrics описано в методі get_system_metrics(), програмний код якого зображено на рисунку 3.8.

```
def get_system_metrics(self) -> Dict:
    """Get overall system metrics"""
    try:
        db_health = self.check_database_health()
        redis_health = self.check_redis_health()

        # Calculate overall health
        all_healthy = all(h.status == "healthy" for h in [db_health, redis_health])

        # Get saga stats
        with self.db_conn.cursor() as cursor:
            cursor.execute("SELECT COUNT(*) FROM saga_transactions WHERE status = 'started'")
            active_sagas = cursor.fetchone()[0]

            cursor.execute("SELECT COUNT(*) FROM saga_transactions")
            total_sagas = cursor.fetchone()[0]

        return {
            "overall_status": "healthy" if all_healthy else "degraded",
            "services": {
                "database": db_health.dict(),
                "redis": redis_health.dict()
            },
            "metrics": {
                "active_sagas": active_sagas,
                "total_sagas": total_sagas
            },
            "timestamp": datetime.now().isoformat()
        }

    except Exception as e:
        logger.error(f"Failed to get system metrics: {e}")
        return {
            "overall_status": "error",
            "error": str(e),
            "timestamp": datetime.now().isoformat()
        }
```

Рисунок 3.8 - Програмний код методу get_system_metrics()

Крім того, ендпоінт /database/stats (рисунок 3.9) надає інформацію про розміри таблиць і стани з'єднань у PostgreSQL, допомагаючи діагностувати уповільнення, спричинене великими таблицями або надмірно відкритими з'єднаннями. Дані витягуються за допомогою оптимізованих SQL-запитів у системних представленнях PostgreSQL (pg_stat_activity, pg_tables) і повертаються в зручному для читання форматі. У поєднанні з внутрішнім журналюванням і структурованими відповідями цілісності ця служба моніторингу сприяє покращенню спостережуваності та підтримує проактивне обслуговування та усунення несправностей. [7]

```python
@app.get("/database/stats")
async def get_database_stats():
    """Get detailed database statistics"""
    try:
        with monitor.db_conn.cursor() as cursor:
            # Get table sizes
            cursor.execute("""
                SELECT schemaname, tablename, pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) as size
                FROM pg_tables
                WHERE schemaname = 'public'
                ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC
            """)
            table_sizes = [{"schema": row[0], "table": row[1], "size": row[2]} for row in cursor.fetchall()]

            # Get connection info
            cursor.execute("""
                SELECT state, COUNT(*)
                FROM pg_stat_activity
                WHERE pid <> pg_backend_pid()
                GROUP BY state
            """)
            connection_states = [{"state": row[0], "count": row[1]} for row in cursor.fetchall()]

            return {
                "table_sizes": table_sizes,
                "connection_states": connection_states,
                "timestamp": datetime.now().isoformat()
            }

    except Exception as e:
        logger.error(f"Failed to get database stats: {e}")
        raise HTTPException(status_code=500, detail=str(e))
```

Рисунок 3.9 – Програмний код ендпоінту /database/stats

Ендпоінти працездатності на рівні програми (/health) відкриваються кожним мікросервісом через FastAPI. Вони забезпечують миттєвий зворотний зв’язок щодо робочого стану та готовності сервісу, а не лише процесу. Вони використовуються Docker Swarm і зовнішніми системами моніторингу для перевірки готовності програми.

Структуроване логування є ще одним ключовим елементом. Кожен сервіс використовує конфігурацію ведення логів в файлі log_config.yaml, вміст якого наведений на рисунку 3.10, для створення узгоджених записів із часовими мітками, рівнями серйозності, текстом трейсбеку. Логи записуються як до стандартного виводу, що підходить для контейнерних середовищ, так і до постійних файлів .log, змонтованих через томи Docker. [11]

user-service > app > core > log_config.yaml

```yaml
version: 1

formatters:
  default:
    format: "[%(asctime)s] [%(levelname)s] %(name)s: %(message)s"

handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: default
    stream: ext://sys.stdout

  file:
    class: logging.FileHandler
    level: INFO
    formatter: default
    filename: app/log/app.log
    mode: a

loggers:
  uvicorn:
    level: INFO
    handlers: [console, file]
    propagate: False

  uvicorn.error:
    level: INFO
    handlers: [console, file]
    propagate: False

  uvicorn.access:
    level: INFO
    handlers: [console]
    propagate: False

  app:
    level: DEBUG
    handlers: [console, file]
    propagate: True

root:
  level: INFO
  handlers: [console, file]

```

Рисунок 3.10 – Вміст файлу log_config.yaml

Перевірки працездатності сервісу також підтверджують підключення до зовнішніх залежностей, таких як бази даних і RabbitMQ. Якщо сервіс не може отримати критичні ресурси, вона позначає себе як несправну. Це комплексне налаштування забезпечує швидке виявлення та вирішення проблем, підтримує стабільність системи та надає важливу інформацію для реагування на інциденти.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі виконано і детально описано проектування архітектури розробленої системи, включаючи підхід до мікросервісів, оркестрацію Docker Swarm, організацію розподіленої бази даних, а також стійкість і масштабованість веб-сервісу. Кожен мікросервіс виконує окрему функціональну роль, має власну базу даних і забезпечує чітко визначену логіку бізнес-процесу. Це сприяє високому ступеню модульності та автономності, а також спрощує технічне обслуговування та розширення системи.

Використання Redis як високошвидкісного кешу та RabbitMQ як асинхронного каналу зв'язку покращує загальну продуктивність і масштабованість системи. Redis забезпечує миттєвий доступ до часто запитуваних даних, таким чином зменшуючи навантаження на базу даних. RabbitMQ дозволяє мікросервісам обмінюватися подіями без прямої залежності один від одного, забезпечуючи гнучкість і стійкість до пікових навантажень. Цей підхід сприяє побудові більш адаптивної архітектури, дотримуючись принципів кінцевої узгодженості.

Прийнята стратегія «база даних на сервіс» забезпечує фізичну та логічну ізоляцію даних, знижує ризик конфліктів і дозволяє окремим командам самостійно розробляти свої компоненти. У поєднанні з централізованим шлюзом API, який обробляє функції маршрутизації, частково автентифікації та обмеження трафіку, ця архітектура забезпечує чіткий розподіл обов'язків і узгоджену політику безпеки. Для підвищення надійності та масштабованості, у сервісі користувачів реалізовано реплікацію бази даних, що дозволяє розподіляти навантаження між читанням і записом.

Загалом реалізована архітектура демонструє збалансований підхід до розробки розподіленої системи, яка відповідає вимогам гнучкості, масштабованості та відмовостійкості.

ВИСНОВКИ

Отже, в результаті виконання дипломної роботи було реалізовано веб-сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури і розподіленої системи управління базами даних. Основною метою проєкту була розробка масштабованої та відмовостійкої платформи, яка забезпечує ефективну взаємодію між учасниками екосистеми стартапів – розробниками, менторами, інвесторами та іншими користувачами.

У першому розділі проаналізовано архітектурні підходи до побудови програмних систем, порівняно монолітну та мікросервісну архітектури, описано основні принципи створення мікросервісів та шаблони взаємодії, які забезпечують гнучкість, незалежність та масштабованість компонентів. Особливу увагу було приділено поняттям і особливостям розподілених СУБД, їх перевагам і недолікам, принципам організації доступу до даних у децентралізованому середовищі.

У другому розділі було обрано стек технологій для впровадження системи. Вибрані інструменти включають Python як основну мову програмування, FastAPI для створення REST API, Docker Swarm для оркестровки, RabbitMQ для асинхронної взаємодії, PostgreSQL як розподілену СУБД та інші допоміжні інструменти для автоматизації послуг, тестування та розгортання. Комплексно продемонстровано доцільність використання кожної технології в контексті мікросервісного підходу.

Третій розділ реалізує повну архітектуру системи, що складається з незалежних мікросервісів: сервісу користувачів, сервісу проєкту, коментарів, заявок, шлюзу API та допоміжних модулів. Застосовуються принципи ізоляції бази даних, асинхронної обробки подій і розгортання масштабованого контейнера. Особлива увага приділяється розподіленій СУБД: кожен сервіс організовує незалежне зберігання даних з урахуванням вимог доступності та відмовостійкості.

Результатом проєкту став гнучкий, розширюваний і надійний веб-сервіс, який демонструє переваги архітектури мікросервісів для побудови складних інформаційних систем. Розроблений сервіс може бути легко розширений клієнтською частиною платформи, або новими серверними модулями адаптованими до зростання кількості користувачів.

Таким чином, робота продемонструвала глибоке розуміння теоретичних основ побудови сучасних розподілених систем, практичну реалізацію мікросервісного підходу та вміння проектувати, розробляти та розгортати функціональні веб-сервіси з використанням галузевих стандартів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Mark Summerfield. Programming in Python 3: A Complete Introduction to the Python Language. – Addison-Wesley, 2010. – С. 25–40.
2. Sebastián Ramírez. FastAPI Documentation [Електронний ресурс]. – Режим доступу: https://fastapi.tiangolo.com/
3. Sam Newman. Building Microservices: Designing Fine-Grained Systems. – O'Reilly Media, 2015. – С. 30–65.
4. Michael Hausenblas, Stefan Schimanski. Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications. – O'Reilly Media, 2022. – С. 122–135.
5. Diogo Resende. Microservices Patterns and Best Practices. – Packt Publishing, 2018. – С. 77–95.
6. RabbitMQ Documentation [Електронний ресурс]. – Режим доступу: https://www.rabbitmq.com/documentation.html
7. PostgreSQL Global Development Group. PostgreSQL 14 Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/
8. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. – O'Reilly Media, 2017. – С. 45–72.
9. Alembic Documentation [Електронний ресурс]. – Режим доступу: https://alembic.sqlalchemy.org/
10. SQLModel Documentation by Sebastián Ramírez [Електронний ресурс]. – Режим доступу: https://sqlmodel.tiangolo.com/
11. Python Software Foundation. The Python Language Reference [Електронний ресурс]. – Режим доступу: https://docs.python.org/3/
12. Poetry Documentation – Python dependency management and packaging [Електронний ресурс]. – Режим доступу: https://python-poetry.org/docs/

13. Docker Documentation [Електронний ресурс]. – Режим доступу: https://docs.docker.com/
14. Brewer E. CAP Theorem in Distributed Systems [Електронний ресурс]. – Режим доступу: https://cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
15. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. – Addison-Wesley, 2004. – С. 154–17
16. Grinberg M. Flask Web Development: Developing Web Applications with Python and Flask. – O'Reilly Media, 2018. – Розділи про SQLAlchemy, с. 57–91
17. Розподілені бази даних: навчальний посібник. - К. ДУТ 2018. - 97с. [Електронний ресурс]. – Режим доступу: https://knute.edu.ua/file/MTcyNjQ=/860e81b460df8e3091ac44f8b2642a59.pdf

Додаток 1

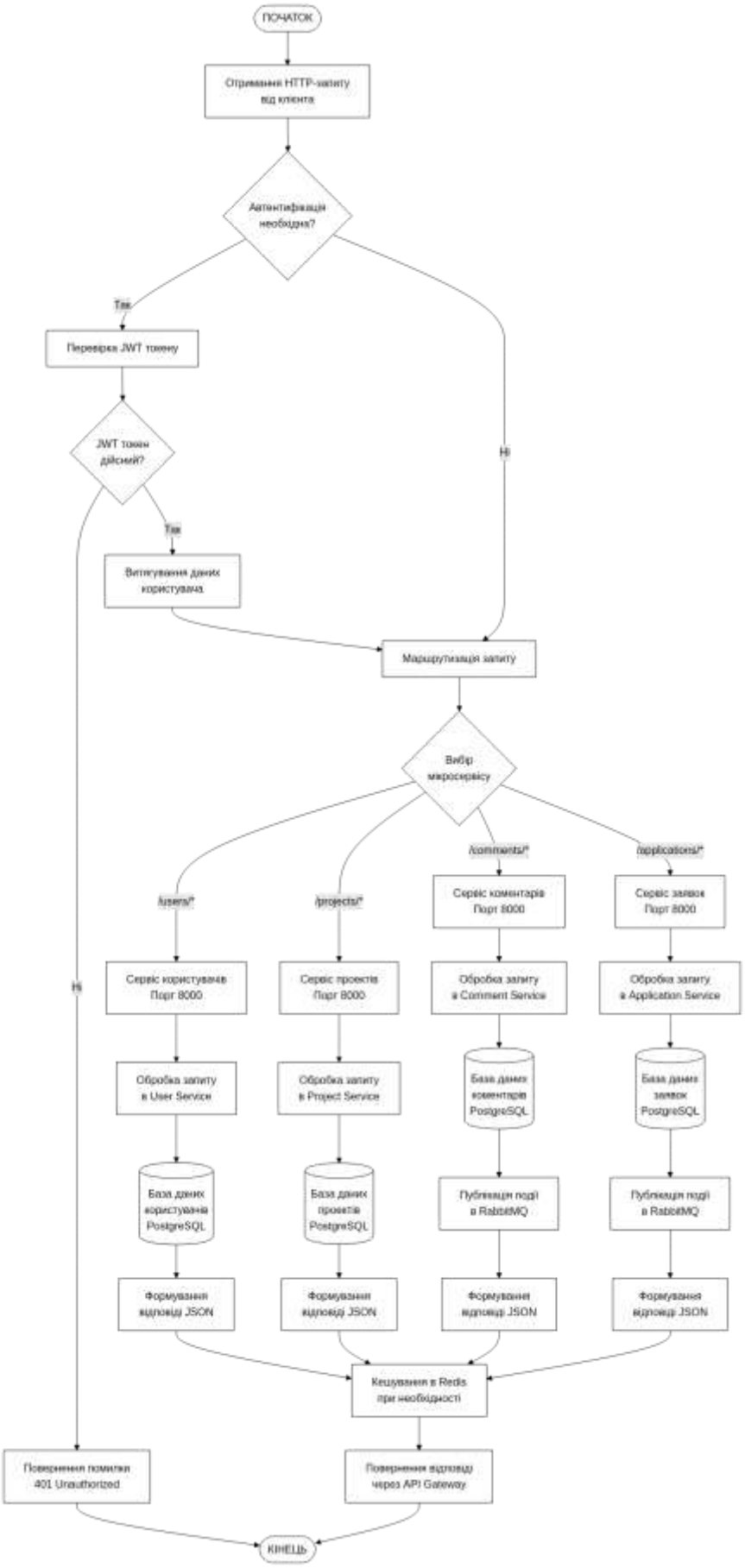
Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД

Принципова схема

ІАЛЦ.467100.004 Д1

Аркушів 1

Київ 2025



					ІАЛЦ.467100.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Жадько Н. Ю.			Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД Принципова схема	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Русінов В. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-15
Н. Контр.		Пономаренко А.М.				
Затвердив						

Додаток 2

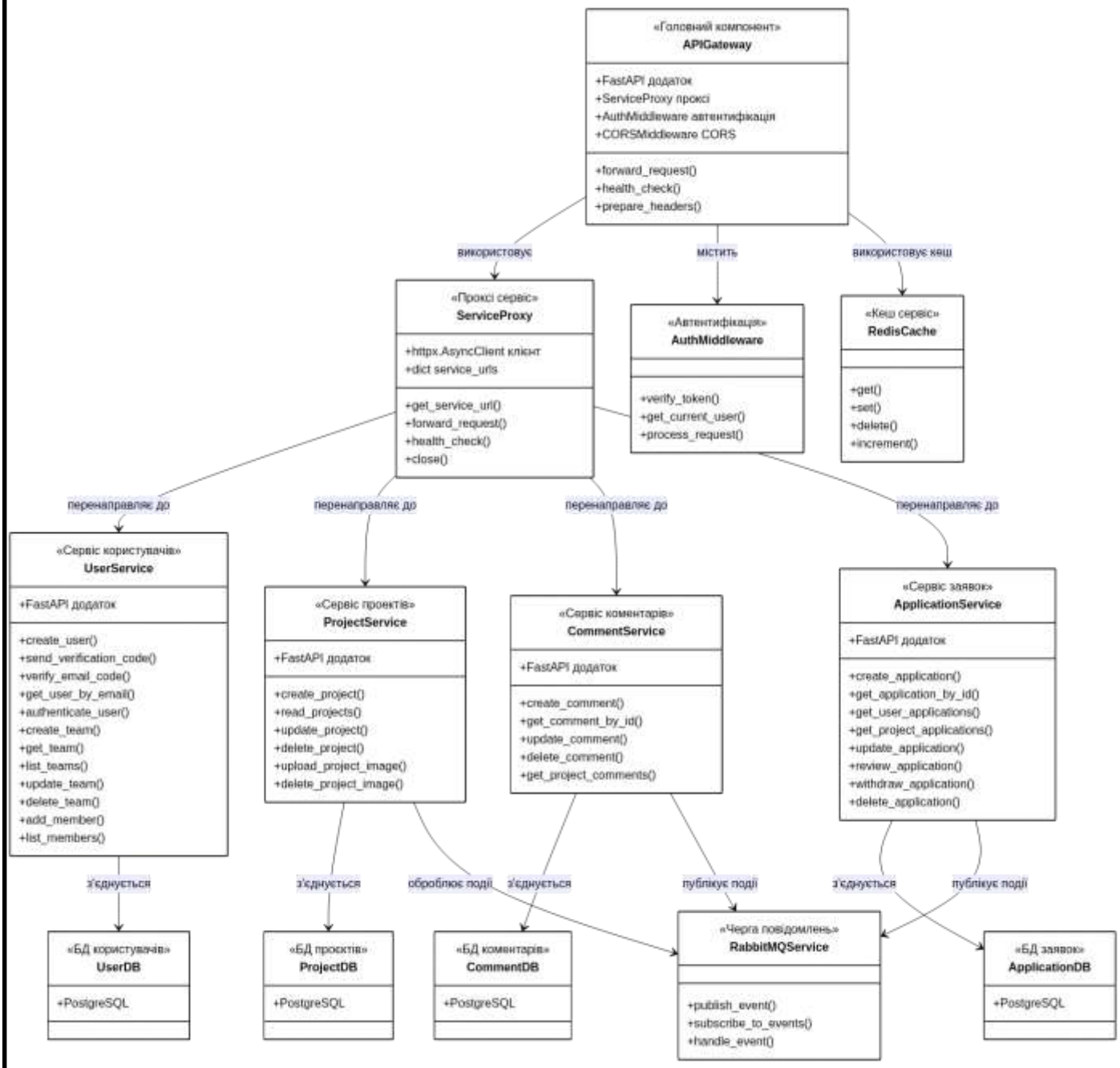
Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД

Функціональна схема

ІАЛЦ.467100.005 Д2

Аркушів 1

Київ 2025



					ІАЛЦ.467100.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Жадько Н. Ю.			Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД Функціональна схема	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Русінов В. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-15
Н. Контр.		Пономаренко А.М.				
Затвердив						

Додаток З

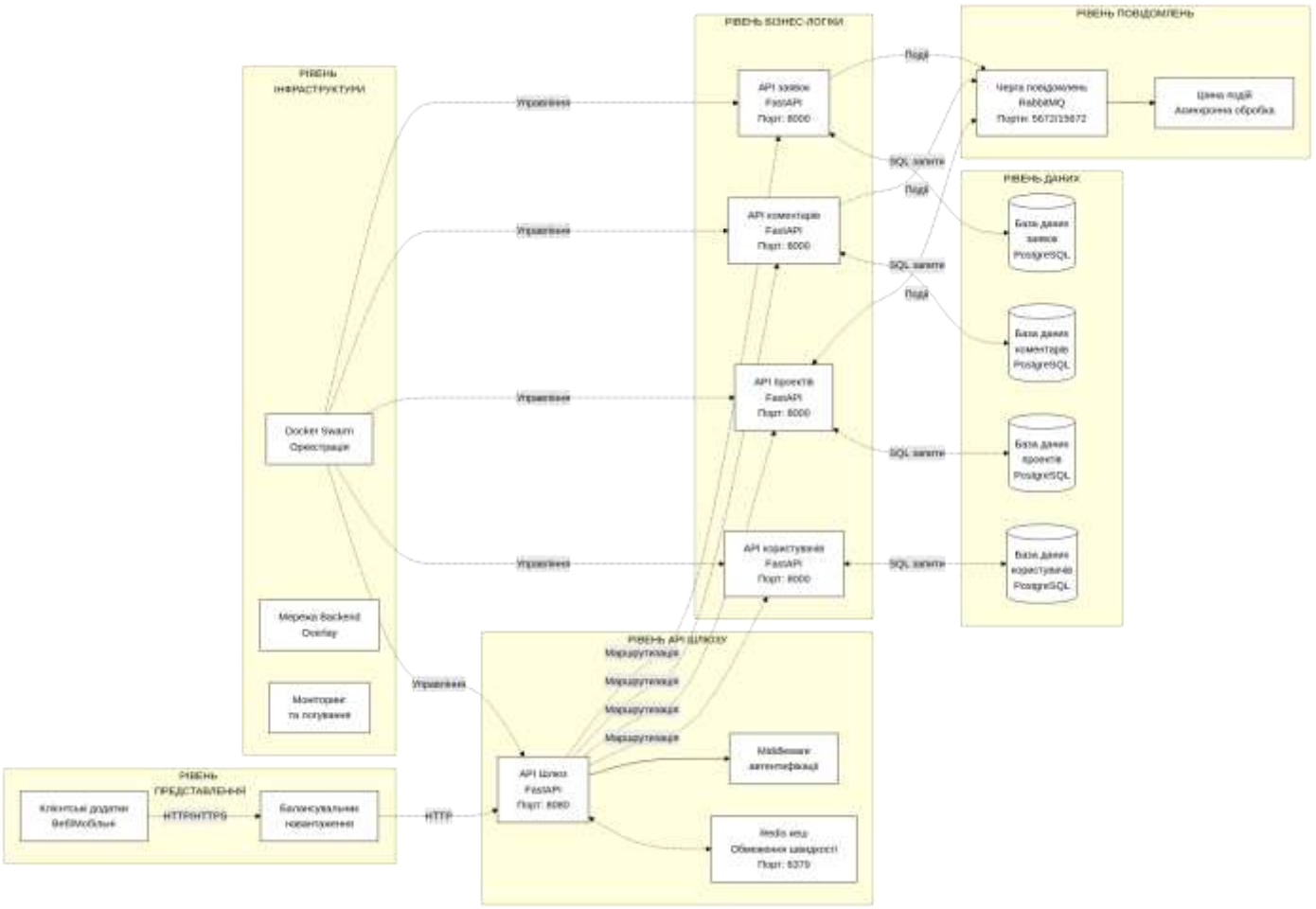
Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД

Структурна схема

ІАЛЦ.467100.006 ДЗ

Аркушів 1

Київ 2025



					ІАЛЦ.467100.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Жадько Н. Ю.			Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД Структурна схема	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Русінов В. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-15
Н. Контр.		Пономаренко А.М.				
Затвердив						

Додаток 4

Сервіс для взаємодії з особистими проєктами з використанням мікросервісної архітектури та розподіленої СУБД

Текст програмного коду

ІАЛЦ.467100.007 Д4

Аркушів 106

Київ 2025

api-gateway-service

app/main.py

```
from contextlib import asynccontextmanager

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from app.core.config import settings
from app.gateway.proxy import proxy
from app.routers import gateway
from app.middleware.auth import AuthMiddleware
from app.middleware.ratelimit import RateLimitMiddleware



@asynccontextmanager
async def lifespan(app: FastAPI):
    """Application lifespan events"""
    print("Starting API Gateway Service...")

    yield

    print("Shutting down API Gateway Service...")
    await proxy.close()
    print("API Gateway Service shutdown complete")



app = FastAPI(
    title=settings.PROJECT_NAME,
    version="1.0.0",
    description="API Gateway for Startup Project Management Platform",
    docs_url="/docs",
    redoc_url="/redoc",
    openapi_url=f"{settings.API_V1_STR}/openapi.json",
    lifespan=lifespan,
)
```

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=settings.all_cors_origins,
    allow_credentials=True,
    allow_methods=["GET", "POST", "PUT", "DELETE", "OPTIONS", "PATCH"],
    allow_headers=["Authorization", "Content-Type", "Accept", "X-Requested-With"],
)

app.add_middleware(RateLimitMiddleware)

app.add_middleware(AuthMiddleware)


@app.get("/")
async def root():
    """Root endpoint"""
    return {
        "message": "API Gateway Service",
        "version": "1.0.0",
        "status": "healthy",
        "docs": "/docs"
    }


@app.get("/health")
async def health():
    """Health check endpoint"""
    return {"status": "healthy"

app.include_router(gateway.router, tags=["gateway"])
```

app/core/security.py

```
from datetime import datetime, timedelta, timezone
from typing import Dict, Any, Optional

from fastapi import HTTPException, Request
from jose import jwt

from app.core.config import settings
```

```
        def create_jwt_token(
            user_id: str,
            expires_delta: timedelta,
            token_type: str = "access"
        ) -> str:
            now = datetime.now(timezone.utc)
            expire = now + expires_delta

            payload = {
                "sub": str(user_id),
                "iat": now,
                "exp": expire,
                "token_type": token_type
            }

            return jwt.encode(payload, settings.SECRET_KEY, algorithm=settings.ALGORITHM)


        def verify_jwt_token(token: str, expected_token_type: str = None) -> Dict[str, Any]:
            try:
                payload = jwt.decode(token, settings.SECRET_KEY,
    algorithms=[settings.ALGORITHM])

                exp = payload.get("exp")
                if exp is None or datetime.utcnow() > datetime.fromtimestamp(exp):
                    raise HTTPException(status_code=401, detail="Token has expired")

                if expected_token_type:
                    token_type = payload.get("token_type")
                    if token_type != expected_token_type:
                        raise HTTPException(
                            status_code=401,
                            detail=f"Invalid token type. Expected {expected_token_type}, got
    {token_type}"
                        )

                return payload

            except jwt.ExpiredSignatureError:
                raise HTTPException(status_code=401, detail="Token has expired")
            except jwt.JWTError:
```

```
            raise HTTPException(status_code=401, detail="Invalid token")


    def extract_token_from_cookies_or_header(request: Request) -> Optional[str]:
        access_token = request.cookies.get("access_token")
        if access_token:
            return access_token

        authorization = request.headers.get("authorization")
        if authorization and authorization.startswith("Bearer "):
            return authorization.split(" ")[1]

        return None


    def get_current_user_id(request: Request) -> str:
        token = extract_token_from_cookies_or_header(request)
        if not token:
            raise HTTPException(status_code=401, detail="Authentication token missing")

        payload = verify_jwt_token(token, "access")
        user_id = payload.get("sub")
        if not user_id:
            raise HTTPException(status_code=401, detail="Invalid token payload")

        return user_id


    def get_refresh_token(request: Request) -> str:
        refresh_token = request.cookies.get("refresh_token")
        if not refresh_token:
            raise HTTPException(status_code=401, detail="Refresh token missing")

        payload = verify_jwt_token(refresh_token, "refresh")
        user_id = payload.get("sub")
        if not user_id:
            raise HTTPException(status_code=401, detail="Invalid refresh token payload")

        return user_id
```

```
def create_tokens_for_user(user_id: str) -> Dict[str, str]:
    access_token = create_jwt_token(
        user_id=user_id,
        expires_delta=timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES),
        token_type="access"
    )

    refresh_token = create_jwt_token(
        user_id=user_id,
        expires_delta=timedelta(days=settings.REFRESH_TOKEN_EXPIRE_DAYS),
        token_type="refresh"
    )

    return {
        "access_token": access_token,
        "refresh_token": refresh_token
    }
```

app/gateway/proxy.py

```
from typing import Dict, Optional
from urllib.parse import urljoin

import httpx
from fastapi import HTTPException, Request, Response, status

from app.core.config import settings


class ServiceProxy:
    def __init__(self):
        self.client = httpx.AsyncClient(timeout=30.0)
        self.service_urls = {
            "users": settings.USER_SERVICE_URL,
            "projects": settings.PROJECT_SERVICE_URL,
            "comments": settings.COMMENT_SERVICE_URL,
            "applications": settings.APPLICATION_SERVICE_URL,
        }

    async def close(self):
        await self.client.aclose()
```

```
def get_service_url(self, service_name: str) -> str:
    if service_name not in self.service_urls:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Service '{service_name}' not found"
        )
    return self.service_urls[service_name]

def prepare_headers(self, request: Request) -> Dict[str, str]:
    headers = {}
    for name in [
        "authorization", "content-type", "accept", "user-agent",
        "x-forwarded-for", "x-real-ip"
    ]:
        value = request.headers.get(name)
        if value:
            headers[name] = value

    headers["x-forwarded-by"] = "api-gateway"
    headers["x-gateway-version"] = "1.0.0"

    if hasattr(request.state, "user_id") and request.state.user_id:
        headers["X-User-ID"] = str(request.state.user_id)

    if request.client and request.client.host:
        headers["x-forwarded-for"] = request.client.host

    return headers

async def forward_request(
    self,
    request: Request,
    service_name: str,
    path: str,
    method: str,
    body: Optional[bytes] = None,
    params: Optional[Dict] = None
) -> Response:
    service_url = self.get_service_url(service_name)
    target_url = urljoin(service_url.rstrip("/") + "/", path.lstrip("/"))
    headers = self.prepare_headers(request)
```

```
        query_params = dict(request.query_params) if not params else params

        try:
            response = await self.client.request(
                method=method,
                url=target_url,
                headers=headers,
                json=body,
                params=query_params,
                follow_redirects=True
            )

            response_headers = {
                name: response.headers.get(name)
                for name in [
                    "content-type", "content-length", "cache-control",
                    "etag", "last-modified", "location"
                ]
                if response.headers.get(name)
            }
            response_headers.update(self.get_cors_headers(request))

            return Response(
                content=response.content,
                status_code=response.status_code,
                headers=response_headers,
                media_type=response.headers.get("content-type")
            )

        except httpx.RequestError as e:
            raise HTTPException(
                status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
                detail=f"Service '{service_name}' is unavailable: {str(e)}"
            )
        except httpx.TimeoutException:
            raise HTTPException(
                status_code=status.HTTP_504_GATEWAY_TIMEOUT,
                detail=f"Service '{service_name}' request timeout"
            )

    def get_cors_headers(self, request: Request) -> Dict[str, str]:
```

```
                origin = request.headers.get("origin")
                headers = {
                    "Access-Control-Allow-Methods": "GET, POST, PUT, DELETE, OPTIONS,
PATCH",
                    "Access-Control-Allow-Headers": "Authorization, Content-Type, Accept, X-
Requested-With",
                    "Access-Control-Allow-Credentials": "true",
                    "Access-Control-Max-Age": "86400"
                }
                headers["Access-Control-Allow-Origin"] = (
                    origin if origin in settings.all_cors_origins else "*"
                )
                return headers

            async def health_check(self, service_name: str) -> Dict[str, str]:
                try:
                    health_url = urljoin(
                        self.get_service_url(service_name).rstrip("/") + "/", "health"
                    )
                    response = await self.client.get(health_url, timeout=5.0)
                    if response.status_code == 200:
                        return {"status": "healthy", "service": service_name}
                    return {
                        "status": "unhealthy",
                        "service": service_name,
                        "code": response.status_code
                    }
                except Exception as e:
                    return {
                        "status": "unreachable",
                        "service": service_name,
                        "error": str(e)
                    }


        proxy = ServiceProxy()
```

app/routers/gateway.py

```
from fastapi import APIRouter, Depends, Request, Body

from app.gateway.proxy import proxy
```

```
from app.middleware.auth import get_current_user, TokenData

router = APIRouter()


@router.get("/health")
async def health_check():
    return {"status": "healthy", "service": "api-gateway"}


@router.get("/health/services")
async def services_health_check():
    services = ["users", "projects", "comments", "applications"]
    health_status = {}

    for service in services:
        health_status[service] = await proxy.health_check(service)

    overall_status = "healthy"
    for service_health in health_status.values():
        if service_health["status"] != "healthy":
            overall_status = "degraded"
            break

    return {
        "status": overall_status,
        "services": health_status
    }


@router.api_route("/api/v1/users/me", methods=["GET", "PUT", "PATCH"])
async def user_profile_routes(
    request: Request,
    body: dict = Body(...),
    current_user: TokenData = Depends(get_current_user)
):
    return await proxy.forward_request(
        request=request,
        service_name="users",
        path=request.url.path,
        method=request.method,
```

```
            body=body
        )


    @router.api_route("/api/v1/projects/{path:path}", methods=["GET", "POST", "PUT",
"DELETE", "PATCH"])
    async def project_routes(
        request: Request,
        path: str,
        body: dict = Body(...),
        current_user: TokenData = Depends(get_current_user)
    ):
        full_path = f"/api/v1/projects/{path}" if path else "/api/v1/projects"
        return await proxy.forward_request(
            request=request,
            service_name="projects",
            path=full_path,
            method=request.method,
            body=body
        )


    @router.api_route("/api/v1/comments/{path:path}", methods=["GET", "POST", "PUT",
"DELETE", "PATCH"])
    async def comment_routes(
        request: Request,
        path: str,
        body: dict = Body(...),
        current_user: TokenData = Depends(get_current_user)
    ):
        full_path = f"/api/v1/comments/{path}" if path else "/api/v1/comments"
        return await proxy.forward_request(
            request=request,
            service_name="comments",
            path=full_path,
            method=request.method,
            body=body
        )
```

```
@router.api_route("/api/v1/applications/{path:path}", methods=["GET", "POST", "PUT",
"DELETE", "PATCH"])
async def application_routes(
    request: Request,
    path: str,
    body: dict = Body(...),
    current_user: TokenData = Depends(get_current_user)
):
    full_path = f"/api/v1/applications/{path}" if path else "/api/v1/applications"
    return await proxy.forward_request(
        request=request,
        service_name="applications",
        path=full_path,
        method=request.method,
        body=body
    )


@router.api_route("/api/v1/auth/{path:path}", methods=["POST", "GET"])
async def auth_routes(
    request: Request,
    path: str,
    body: dict = Body(...)
):
    full_path = f"/api/v1/users/{path}" if path else "/api/v1/users"
    return await proxy.forward_request(
        request=request,
        service_name="users",
        path=full_path,
        method=request.method,
        body=body
    )
```

app/middleware/auth.py

```
from datetime import datetime, timedelta
from typing import Optional

from fastapi import HTTPException, Request, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import JWTError, jwt
from pydantic import BaseModel
```

```
        from fastapi.responses import JSONResponse
        from starlette.middleware.base import BaseHTTPMiddleware

        from app.core.config import settings
        from app.core.security import extract_token_from_cookies_or_header, verify_jwt_token


        class TokenData(BaseModel):
            user_id: Optional[str] = None
            email: Optional[str] = None
            token_type: Optional[str] = None


        class JWTAuth:
            def __init__(self):
                self.secret_key = settings.SECRET_KEY
                self.algorithm = settings.ALGORITHM

            def verify_token(self, token: str, expected_token_type: str = "access") ->
    TokenData:
                try:
                    payload = jwt.decode(token, self.secret_key,
    algorithms=[self.algorithm])
                    user_id: str = payload.get("sub")
                    email: str = payload.get("email")
                    token_type: str = payload.get("token_type", "access")

                    if user_id is None:
                        raise HTTPException(
                            status_code=status.HTTP_401_UNAUTHORIZED,
                            detail="Invalid authentication credentials",
                            headers={"WWW-Authenticate": "Bearer"},
                        )

                    if token_type != expected_token_type:
                        raise HTTPException(
                            status_code=status.HTTP_401_UNAUTHORIZED,
                            detail=f"Invalid token type. Expected {expected_token_type}",
                            headers={"WWW-Authenticate": "Bearer"},
                        )
```

```
                return TokenData(user_id=user_id, email=email, token_type=token_type)
            except JWTError:
                raise HTTPException(
                    status_code=status.HTTP_401_UNAUTHORIZED,
                    detail="Invalid authentication credentials",
                    headers={"WWW-Authenticate": "Bearer"},
                )

        def create_access_token(self, data: dict, expires_delta: Optional[timedelta] =
None):
            to_encode = data.copy()
            expire = datetime.utcnow() + (
                expires_delta or timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)
            )
            to_encode.update({"exp": expire, "token_type": "access"})
            return jwt.encode(to_encode, self.secret_key, algorithm=self.algorithm)

        def create_refresh_token(self, data: dict, expires_delta: Optional[timedelta] =
None):
            to_encode = data.copy()
            expire = datetime.utcnow() + (
                expires_delta or timedelta(days=7)
            )
            to_encode.update({"exp": expire, "token_type": "refresh"})
            return jwt.encode(to_encode, self.secret_key, algorithm=self.algorithm)


    jwt_auth = JWTAuth()


    def get_current_user_from_cookie(request: Request) -> TokenData:
        access_token = request.cookies.get("access_token")
        if not access_token:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Access token not found in cookies",
                headers={"WWW-Authenticate": "Bearer"},
            )
        return jwt_auth.verify_token(access_token, "access")
```

```
def get_current_user_from_refresh_cookie(request: Request) -> TokenData:
    refresh_token = request.cookies.get("refresh_token")
    if not refresh_token:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Refresh token not found in cookies",
            headers={"WWW-Authenticate": "Bearer"},
        )
    return jwt_auth.verify_token(refresh_token, "refresh")


def get_optional_current_user_from_cookie(request: Request) -> Optional[TokenData]:
    access_token = request.cookies.get("access_token")
    if not access_token:
        return None
    try:
        return jwt_auth.verify_token(access_token, "access")
    except HTTPException:
        return None


class JWTBearer(HTTPBearer):
    def __init__(self, auto_error: bool = True):
        super().__init__(auto_error=auto_error)

    async def __call__(self, request: Request) -> Optional[TokenData]:
        credentials: HTTPAuthorizationCredentials = await super().__call__(request)
        if credentials:
            if credentials.scheme != "Bearer":
                raise HTTPException(
                    status_code=status.HTTP_401_UNAUTHORIZED,
                    detail="Invalid authentication scheme.",
                    headers={"WWW-Authenticate": "Bearer"},
                )
            return jwt_auth.verify_token(credentials.credentials)
        return None


def get_optional_current_user(request: Request) -> Optional[TokenData]:
    user = get_optional_current_user_from_cookie(request)
    if user:
```

```
            return user

        auth_header = request.headers.get("Authorization")
        if not auth_header:
            return None

        try:
            scheme, token = auth_header.split()
            if scheme.lower() != "bearer":
                return None
            return jwt_auth.verify_token(token, "access")
        except (ValueError, HTTPException):
            return None


    def get_current_user(request: Request) -> TokenData:
        try:
            return get_current_user_from_cookie(request)
        except HTTPException:
            pass

        auth_header = request.headers.get("Authorization")
        if not auth_header:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Authentication required - no access token found in cookies or
Authorization header",
                headers={"WWW-Authenticate": "Bearer"},
            )

        try:
            scheme, token = auth_header.split()
            if scheme.lower() != "bearer":
                raise HTTPException(
                    status_code=status.HTTP_401_UNAUTHORIZED,
                    detail="Invalid authentication scheme",
                    headers={"WWW-Authenticate": "Bearer"},
                )
            return jwt_auth.verify_token(token, "access")
        except (ValueError, HTTPException):
            raise HTTPException(
```

```
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Invalid authentication credentials",
                headers={"WWW-Authenticate": "Bearer"},
            )


class AuthMiddleware(BaseHTTPMiddleware):
    def __init__(self, app):
        super().__init__(app)
        self.exempt_paths = {
            "/",
            "/docs",
            "/openapi.json",
            "/api/v1/openapi.json",
            "/redoc",
            "/health",
            "/api/v1/auth/login",
            "/api/v1/auth/register",
            "/api/v1/auth/send-verification-code",
            "/api/v1/auth/verify-email",
        }

    async def dispatch(self, request: Request, call_next):
        path = request.url.path
        if path in self.exempt_paths:
            return await call_next(request)

        try:
            token = extract_token_from_cookies_or_header(request)
            if not token:
                return JSONResponse(
                    status_code=401,
                    content={"detail": "Authentication token missing"}
                )

            payload = verify_jwt_token(token, "access")
            user_id = payload.get("sub")
            if not user_id:
                return JSONResponse(
                    status_code=401,
                    content={"detail": "Invalid token payload"}
```

```
                )

                request.state.user_id = user_id

            except HTTPException as e:
                return JSONResponse(
                    status_code=e.status_code,
                    content={"detail": e.detail}
                )
            except Exception:
                return JSONResponse(
                    status_code=401,
                    content={"detail": "Authentication failed"}
                )

            return await call_next(request)
```

app/middleware/ratelimit.py

```
import time

from fastapi import Request
from starlette.middleware.base import BaseHTTPMiddleware
from starlette.responses import JSONResponse
import aioredis

from app.core.config import settings


class RateLimitMiddleware(BaseHTTPMiddleware):
    def __init__(self, app, max_requests: int = 60, window_seconds: int = 60):
        super().__init__(app)
        self.max_requests = max_requests
        self.window_seconds = window_seconds
        self.redis = None

    async def dispatch(self, request: Request, call_next):
        if self.redis is None:
            self.redis = await aioredis.from_url(
                settings.REDIS_URL,
                encoding="utf-8",
                decode_responses=True
```

```
            )

        ip = request.client.host
        now = int(time.time())
        window = now // self.window_seconds
        redis_key = f"ratelimit:{ip}:{window}"

        count = await self.redis.incr(redis_key)
        if count == 1:
            await self.redis.expire(redis_key, self.window_seconds)

        if count > self.max_requests:
            return JSONResponse(
                status_code=429,
                content={"detail": "Too Many Requests. Rate limit exceeded."}
            )

        return await call_next(request)
```

tests/test_main.py

```
import os
import pytest
from unittest.mock import AsyncMock, Mock, patch
from fastapi.testclient import TestClient
from fastapi import Response

os.environ["TESTING"] = "1"

from app.main import app
from app.middleware.auth import JWTAuth


class TestMainApp:
    def setup_method(self):
        self.client = TestClient(app)

    def test_root_endpoint(self):
        response = self.client.get("/")
        assert response.status_code == 200
        data = response.json()
        assert data["message"] == "API Gateway Service"
```

```
        assert data["version"] == "1.0.0"
        assert data["status"] == "healthy"
        assert data["docs"] == "/docs"

    def test_info_endpoint(self):
        response = self.client.get("/info")
        assert response.status_code == 200
        data = response.json()
        assert data["service"] == "api-gateway"
        assert data["version"] == "1.0.0"
        assert "backend_services" in data
        assert "users" in data["backend_services"]
        assert "projects" in data["backend_services"]
        assert "comments" in data["backend_services"]

    def test_health_endpoint(self):
        response = self.client.get("/health")
        assert response.status_code == 200
        data = response.json()
        assert data["status"] == "healthy"
        assert data["service"] == "api-gateway"

    @patch("app.gateway.proxy.proxy.health_check")
    def test_services_health_endpoint_all_healthy(self, mock_health_check):
        mock_health_check.side_effect = [
            {"status": "healthy", "service": "users"},
            {"status": "healthy", "service": "projects"},
            {"status": "healthy", "service": "comments"}
        ]
        response = self.client.get("/health/services")
        assert response.status_code == 200
        data = response.json()
        assert data["status"] == "healthy"
        assert data["services"]["users"]["status"] == "healthy"
        assert data["services"]["projects"]["status"] == "healthy"
        assert data["services"]["comments"]["status"] == "healthy"

    @patch("app.gateway.proxy.proxy.health_check")
    def test_services_health_endpoint_degraded(self, mock_health_check):
        mock_health_check.side_effect = [
            {"status": "healthy", "service": "users"},
```

```
                    {"status": "unhealthy", "service": "projects", "code": 500},
                    {"status": "healthy", "service": "comments"}
                ]
                response = self.client.get("/health/services")
                assert response.status_code == 200
                data = response.json()
                assert data["status"] == "degraded"
                assert data["services"]["projects"]["status"] == "unhealthy"

            def test_cors_preflight_request(self):
                response = self.client.options(
                    "/api/v1/users",
                    headers={
                        "Origin": "http://localhost:3000",
                        "Access-Control-Request-Method": "POST",
                        "Access-Control-Request-Headers": "Authorization"
                    }
                )
                assert response.status_code == 200
                assert "Access-Control-Allow-Origin" in response.headers

            def test_unknown_endpoint_returns_404(self):
                response = self.client.get("/unknown/endpoint")
                assert response.status_code == 404
                assert "Endpoint not found" in response.json()["detail"]

            def test_unknown_api_endpoint_returns_404(self):
                response = self.client.get("/api/v1/unknown-service/test")
                assert response.status_code == 404
                assert "Endpoint not found" in response.json()["detail"]

            def test_auth_route_forwarding(self, mock_proxy):
                mock_response = Response(
                    content='{"token": "test-token"}',
                    status_code=200,
                    media_type="application/json"
                )
                mock_proxy.forward_request.return_value = mock_response
                response = self.client.post("/api/v1/auth/login", json={"email":
"test@example.com", "password": "password"})
                assert response.status_code == 200
```

```
                mock_proxy.forward_request.assert_called_once()
                args = mock_proxy.forward_request.call_args[1]
                assert args["service_name"] == "users"
                assert args["path"] == "/api/v1/auth/login"
                assert args["method"] == "POST"

            def test_user_registration_forwarding(self, mock_proxy):
                mock_response = Response(
                    content='{"id": "user123"}',
                    status_code=201,
                    media_type="application/json"
                )
                mock_proxy.forward_request.return_value = mock_response
                response = self.client.post("/api/v1/users/register", json={"email":
"test@example.com", "password": "password"})
                assert response.status_code == 201
                mock_proxy.forward_request.assert_called_once()
                args = mock_proxy.forward_request.call_args[1]
                assert args["service_name"] == "users"
                assert args["path"] == "/api/v1/users/register"
                assert args["method"] == "POST"

            def test_protected_route_without_auth_returns_401(self):
                response = self.client.get("/api/v1/users/me")
                assert response.status_code == 401
                assert "Authentication required" in response.json()["detail"]

            def test_protected_route_with_auth_forwards(self, mock_proxy):
                token_data = {"sub": "user123", "email": "test@example.com"}
                token = JWTAuth().create_access_token(token_data)

                mock_response = Response(
                    content='{"id": "user123"}',
                    status_code=200,
                    media_type="application/json"
                )
                mock_proxy.forward_request.return_value = mock_response
                response = self.client.get("/api/v1/users/me", headers={"Authorization":
f"Bearer {token}"})
                assert response.status_code == 200
                mock_proxy.forward_request.assert_called_once()
```

```
        args = mock_proxy.forward_request.call_args[1]
        assert args["service_name"] == "users"
```

user-service

app/main.py

```
import logging

from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
from fastapi.routing import APIRoute
from starlette.middleware.cors import CORSMiddleware

from app.api.main import api_router
from app.core.config import settings

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
)
logger = logging.getLogger(__name__)


def custom_generate_unique_id(route: APIRoute) -> str:
    tag = route.tags[0] if route.tags else "default"
    return f"{tag}-{route.name}"


app = FastAPI(
    title=settings.PROJECT_NAME,
    openapi_url=f"{settings.API_V1_STR}/openapi.json",
    generate_unique_id_function=custom_generate_unique_id,
)


@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    logger.exception(f"Unhandled error: {exc}")
    return JSONResponse(
        status_code=500,
        content={"detail": "Internal server error"},
```

```
    )


if settings.all_cors_origins:
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.all_cors_origins,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

app.include_router(api_router, prefix=settings.API_V1_STR)


@app.get("/")
async def root():
    return {"message": "API is running"}


@app.get("/health")
async def health_check():
    return {"status": "healthy", "service": "user-service"}
```

app/core/config.py

```
import secrets
import warnings
from typing import Annotated, Any, Literal

from pydantic import (
    AnyUrl,
    BeforeValidator,
    EmailStr,
    PostgresDsn,
    computed_field,
    model_validator,
)
from pydantic_core import MultiHostUrl
from pydantic_settings import BaseSettings
from typing_extensions import Self
```

```
        def parse_cors(v: Any) -> list[str] | str:
            if isinstance(v, str) and not v.startswith("["):
                return [i.strip() for i in v.split(",")]
            elif isinstance(v, list | str):
                return v
            raise ValueError(v)


        class Settings(BaseSettings):
            # API & Auth
            API_V1_STR: str = "/api/v1"
            SECRET_KEY: str = secrets.token_urlsafe(32)
            ACCESS_TOKEN_EXPIRE_MINUTES: int = 15
            REFRESH_TOKEN_EXPIRE_DAYS: int = 30
            AUTH_CODE_EXPIRE_MINUTES: int = 10

            # Environment & CORS
            ENVIRONMENT: Literal["local", "staging", "production"] = "local"
            FRONTEND_HOST: str = "http://localhost:5173"
            BACKEND_CORS_ORIGINS: Annotated[list[AnyUrl] | str, BeforeValidator(parse_cors)]
    = []

            @computed_field
            @property
            def all_cors_origins(self) -> list[str]:
                return [
                    str(origin).rstrip("/") for origin in self.BACKEND_CORS_ORIGINS
                ] + [self.FRONTEND_HOST]

            # Email / SendGrid
            SENDGRID_API_KEY: str = ""
            SENDGRID_FROM_EMAIL: EmailStr = "noreply@example.com"
            SENDGRID_FROM_NAME: str = "YeIdea"

            # Redis
            REDIS_URL: str = "redis://redis:6379/0"

            # Project metadata
            PROJECT_NAME: str
```

```
    # PostgreSQL (main)
    POSTGRES_SERVER: str
    POSTGRES_PORT: int = 5432
    POSTGRES_USER: str
    POSTGRES_PASSWORD: str = ""
    POSTGRES_DB: str = ""

    # PostgreSQL (replica)
    POSTGRES_REPLICA_SERVER: str = ""
    POSTGRES_REPLICA_PORT: int = 5432

    @computed_field
    @property
    def SQLALCHEMY_DATABASE_URI_ASYNC(self) -> PostgresDsn:
        return MultiHostUrl.build(
            scheme="postgresql+asyncpg",
            username=self.POSTGRES_USER,
            password=self.POSTGRES_PASSWORD,
            host=self.POSTGRES_SERVER,
            port=self.POSTGRES_PORT,
            path=self.POSTGRES_DB,
        )

    @computed_field
    @property
    def SQLALCHEMY_DATABASE_URI_ASYNC_REPLICA(self) -> PostgresDsn:
        replica_server = self.POSTGRES_REPLICA_SERVER or self.POSTGRES_SERVER
        return MultiHostUrl.build(
            scheme="postgresql+asyncpg",
            username=self.POSTGRES_USER,
            password=self.POSTGRES_PASSWORD,
            host=replica_server,
            port=self.POSTGRES_REPLICA_PORT,
            path=self.POSTGRES_DB,
        )

    @computed_field
    @property
    def SQLALCHEMY_DATABASE_URI_SYNC(self) -> PostgresDsn:
        return MultiHostUrl.build(
            scheme="postgresql+psycopg",
```

```
            username=self.POSTGRES_USER,
            password=self.POSTGRES_PASSWORD,
            host=self.POSTGRES_SERVER,
            port=self.POSTGRES_PORT,
            path=self.POSTGRES_DB,
        )

    def _check_default_secret(self, var_name: str, value: str | None) -> None:
        if value == "changethis":
            message = (
                f'The value of {var_name} is "changethis", '
                "for security, please change it, at least for deployments."
            )
            if self.ENVIRONMENT == "local":
                warnings.warn(message, stacklevel=1)
            else:
                raise ValueError(message)

    @model_validator(mode="after")
    def _enforce_non_default_secrets(self) -> Self:
        self._check_default_secret("SECRET_KEY", self.SECRET_KEY)
        self._check_default_secret("POSTGRES_PASSWORD", self.POSTGRES_PASSWORD)
        return self


settings = Settings()
```

app/api/deps.py

```
from collections.abc import AsyncGenerator
from typing import Annotated, Optional

from fastapi import Depends, HTTPException, Header
from sqlmodel.ext.asyncio.session import AsyncSession
from sqlmodel import select

from app.core.db import get_read_session, get_write_session
from app.users.models import User


async def get_read_db() -> AsyncGenerator[AsyncSession, None]:
    """Get read-only database session (replica)"""
```

```
    async for session in get_read_session():
        yield session


async def get_write_db() -> AsyncGenerator[AsyncSession, None]:
    """Get write database session (master)"""
    async for session in get_write_session():
        yield session


async def get_db() -> AsyncGenerator[AsyncSession, None]:
    """Default database session - uses read session for backward compatibility"""
    async for session in get_read_session():
        yield session


ReadSessionDep = Annotated[AsyncSession, Depends(get_read_db)]
WriteSessionDep = Annotated[AsyncSession, Depends(get_write_db)]
SessionDep = Annotated[AsyncSession, Depends(get_db)]


def get_user_id_from_header(
    x_user_id: Annotated[Optional[str], Header()] = None
) -> str:
    """Extract user ID from X-User-ID header sent by API Gateway"""
    if not x_user_id:
        raise HTTPException(
            status_code=401,
            detail="Authentication required - missing user ID header"
        )
    return x_user_id


UserIDDep = Annotated[str, Depends(get_user_id_from_header)]


async def get_current_user_from_gateway(
    user_id: UserIDDep,
    session: ReadSessionDep
) -> User:
    """Get current user using user_id from API Gateway header"""
```

```
            try:
                statement = select(User).where(User.id == user_id)
                result = await session.exec(statement)
                user = result.first()

                if not user:
                    raise HTTPException(status_code=404, detail="User not found")

                if not user.is_active:
                    raise HTTPException(status_code=403, detail="User account is inactive")

                return user

            except Exception as e:
                raise HTTPException(status_code=500, detail="Failed to fetch user data")


        CurrentUserDep = Annotated[User, Depends(get_current_user_from_gateway)]


        async def get_current_user_for_write(
            user_id: UserIDDep,
            session: ReadSessionDep
        ) -> User:
            """Get current user for write operations - validates user before allowing
writes"""
            return await get_current_user_from_gateway(user_id, session)


        CurrentUserWriteDep = Annotated[User, Depends(get_current_user_for_write)]
```

app/core/email_service.py

```
        import logging

        from sendgrid import SendGridAPIClient
        from sendgrid.helpers.mail import Mail

        from app.core.config import settings

        logger = logging.getLogger(__name__)
```

```
        class EmailService:
            def __init__(self):
                self.sendgrid_client = None
                if settings.SENDGRID_API_KEY:
                    try:
                        self.sendgrid_client =
SendGridAPIClient(api_key=settings.SENDGRID_API_KEY)
                    except Exception as e:
                        logger.error(f"Failed to initialize SendGrid client: {e}")

            async def send_verification_code(self, email: str, code: str, user_name: str) ->
bool:
                if not self.sendgrid_client:
                    logger.error("SendGrid client not initialized - missing API key")
                    return False

                try:
                    subject = f"Your verification code - {settings.SENDGRID_FROM_NAME}"

                    html_content = f"""
                    <!DOCTYPE html>
                    <html>
                    <head>
                        <meta charset="utf-8">
                        <meta name="viewport" content="width=device-width, initial-scale=1">
                        <title>Email Verification</title>
                    </head>
                    <body style="font-family: Arial, sans-serif; line-height: 1.6; color:
#333; max-width: 600px; margin: 0 auto; padding: 20px;">
                        <div style="background-color: #f8f9fa; padding: 30px; border-radius:
10px; text-align: center;">
                            <h2 style="color: #2c3e50; margin-bottom: 30px;">Email
Verification</h2>
                            <p style="font-size: 16px; margin-bottom: 20px;">Hello
{user_name},</p>
                            <p style="font-size: 16px; margin-bottom: 30px;">Your
verification code is:</p>
                            <div style="background-color: #007bff; color: white; font-size:
32px; font-weight: bold; padding: 20px; border-radius: 8px; letter-spacing: 8px; margin:
30px 0;">
```

```
                            {code}
                        </div>
                        <p style="font-size: 14px; color: #666; margin-bottom: 20px;">
                            This code will expire in {settings.AUTH_CODE_EXPIRE_MINUTES}
minutes.
                        </p>
                        <p style="font-size: 14px; color: #666;">
                            If you didn't request this verification code, please ignore
this email.
                        </p>
                        <hr style="border: none; border-top: 1px solid #eee; margin:
30px 0;">
                        <p style="font-size: 12px; color: #999;">
                            © 2024 {settings.SENDGRID_FROM_NAME}. All rights reserved.
                        </p>
                    </div>
                </body>
                </html>
                """

                text_content = (
                    f"Email Verification\n\nHello {user_name},\n\n"
                    f"Your verification code is: {code}\n\n"
                    f"This code will expire in {settings.AUTH_CODE_EXPIRE_MINUTES}
minutes.\n\n"
                    f"If you didn't request this verification code, please ignore this
email.\n\n"
                    f"© 2024 {settings.SENDGRID_FROM_NAME}. All rights reserved."
                )

                message = Mail(
                    from_email=(settings.SENDGRID_FROM_EMAIL,
settings.SENDGRID_FROM_NAME),
                    to_emails=email,
                    subject=subject,
                    html_content=html_content,
                    plain_text_content=text_content,
                )

                response = self.sendgrid_client.send(message)
```

```
                    if response.status_code in [200, 201, 202]:
                        logger.info(f"Verification code sent successfully to {email}")
                        return True

                    logger.error(f"Failed to send email: {response.status_code} -
{response.body}")
                    return False

                except Exception as e:
                    logger.error(f"Error sending verification email to {email}: {e}")
                    return False


email_service = EmailService()
```

app/users/models.py

```
import uuid
from datetime import datetime
from typing import Optional

from sqlmodel import Field, SQLModel


class User(SQLModel, table=True):
    __tablename__ = "users"

    id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=True)
    email: str = Field(index=True, unique=True, max_length=255)
    full_name: str = Field(max_length=255)
    is_active: bool = False
    is_superuser: bool = False
    hashed_password: str
    email_verified: bool = False
    verification_code: Optional[str] = Field(default=None, max_length=6)
    verification_code_expires: Optional[datetime] = Field(default=None)
```

app/users/router.py

```
from datetime import timedelta

from fastapi import APIRouter, Depends, HTTPException
from fastapi.responses import JSONResponse
```

```
from sqlmodel import select

from app.api.deps import SessionDep, CurrentUserDep
from app.core.security import verify_password
from app.users.models import User
from app.users.schemas import (
    UserCreate,
    UserRead,
    UserLogin,
    EmailVerificationRequest,
    EmailVerificationCode,
    AuthResponse,
)
from app.users.service import (
    create_user,
    get_user_by_email,
    send_verification_code,
    verify_email_code,
)

router = APIRouter(prefix="/users", tags=["users"])


@router.post("/register", response_model=AuthResponse)
async def register(session: SessionDep, user_create: UserCreate):
    existing_user = await get_user_by_email(session=session,
email=user_create.email)
    if existing_user:
        raise HTTPException(status_code=400, detail="Email already registered")

    user = await create_user(session=session, user_create=user_create)
    await send_verification_code(session=session, email=user.email)

    return AuthResponse(
        message="User created. Please check your email for the verification code.",
        requires_verification=True,
    )


@router.post("/send-verification-code", response_model=AuthResponse)
```

```
        async def request_verification_code(session: SessionDep, request:
EmailVerificationRequest):
            await send_verification_code(session=session, email=request.email)

            return AuthResponse(
                message="Verification code sent to your email",
                requires_verification=True,
            )


        @router.post("/verify-email")
        async def verify_email(session: SessionDep, verification: EmailVerificationCode):
            user = await verify_email_code(session=session, email=verification.email,
code=verification.code)

            return {
                "message": "Email verified successfully",
                "requires_verification": False,
                "user": {
                    "id": str(user.id),
                    "email": user.email,
                    "full_name": user.full_name,
                    "is_active": user.is_active,
                },
            }


        @router.post("/login")
        async def login(session: SessionDep, user_login: UserLogin):
            user = await get_user_by_email(session=session, email=user_login.email)
            if not user or not verify_password(user_login.password, user.hashed_password):
                raise HTTPException(status_code=401, detail="Invalid credentials")

            if not user.email_verified:
                return {
                    "message": "Email not verified. Please verify your email first.",
                    "requires_verification": True,
                }

            if not user.is_active:
                raise HTTPException(status_code=403, detail="Account is inactive")
```

```
    return {
        "message": "Credentials validated successfully",
        "requires_verification": False,
        "user": {
            "id": str(user.id),
            "email": user.email,
            "full_name": user.full_name,
            "is_active": user.is_active,
        },
    }


@router.post("/logout")
def logout(user: CurrentUserDep):
    return {"message": "Logged out successfully"}


@router.get("/me", response_model=UserRead)
async def read_me(user: CurrentUserDep):
    return user


@router.post("/refresh")
async def refresh(user: CurrentUserDep, session: SessionDep):
    return {
        "message": "User validated for token refresh",
        "user": {
            "id": str(user.id),
            "email": user.email,
            "full_name": user.full_name,
            "is_active": user.is_active,
        },
    }
```

app/users/service.py

```
import secrets
from datetime import datetime, timedelta

from fastapi import HTTPException
from sqlmodel import select
```

```
from app.api.deps import ReadSessionDep, WriteSessionDep
from app.core.config import settings
from app.core.security import get_password_hash, verify_password
from app.core.email_service import email_service
from app.users.models import User
from app.users.schemas import UserCreate

ALGORITHM = "HS256"

def generate_verification_code() -> str:
    """Generate a secure 6-digit verification code."""
    return str(secrets.randbelow(900000) + 100000)


async def create_user(*, session: WriteSessionDep, user_create: UserCreate) -> User:
    """Create a new user - uses write session for database insert."""
    result = await session.exec(
        select(User).where(User.email == user_create.email)
    )
    existing_user = result.first()

    if existing_user:
        raise HTTPException(
            status_code=400,
            detail=f"User with email {user_create.email} already exists",
        )

    db_obj = User.model_validate(
        user_create,
        update={"hashed_password": get_password_hash(user_create.password)},
    )
    session.add(db_obj)
    await session.commit()
    await session.refresh(db_obj)
    return db_obj


async def send_verification_code(*, session: WriteSessionDep, email: str) -> bool:
    """Generate and send verification code to user's email - uses write session for
updates."""
```

```
            user = await get_user_by_email_for_write(session=session, email=email)
            if not user:
                raise HTTPException(status_code=404, detail="User not found")

            if user.email_verified:
                raise HTTPException(status_code=400, detail="Email already verified")

            verification_code = generate_verification_code()
            expiry_time = datetime.now(datetime.UTC) + timedelta(
                minutes=settings.AUTH_CODE_EXPIRE_MINUTES
            )

            user.verification_code = verification_code
            user.verification_code_expires = expiry_time

            session.add(user)
            await session.commit()

            email_sent = await email_service.send_verification_code(
                email=user.email,
                code=verification_code,
                user_name=user.full_name,
            )

            if not email_sent:
                raise HTTPException(status_code=500, detail="Failed to send verification
email")

            return user


       async def verify_email_code(*, session: WriteSessionDep, email: str, code: str) ->
User:
            """Verify the email verification code and activate user - uses write session for
updates."""
            user = await get_user_by_email_for_write(session=session, email=email)
            if not user:
                raise HTTPException(status_code=404, detail="User not found")

            if user.email_verified:
                raise HTTPException(status_code=400, detail="Email already verified")
```

```
            if not user.verification_code or not user.verification_code_expires:
                raise HTTPException(
                    status_code=400,
                    detail="No verification code found. Please request a new one.",
                )

            if datetime.now(datetime.UTC) > user.verification_code_expires:
                raise HTTPException(
                    status_code=400,
                    detail="Verification code has expired. Please request a new one.",
                )

            if user.verification_code != code:
                raise HTTPException(status_code=400, detail="Invalid verification code")

            user.email_verified = True
            user.is_active = True
            user.verification_code = None
            user.verification_code_expires = None

            session.add(user)
            await session.commit()
            await session.refresh(user)

            return user


        async def get_user_by_email(*, session: ReadSessionDep, email: str) -> User | None:
            """Get user by email - uses read session for query optimization."""
            result = await session.exec(select(User).where(User.email == email))
            return result.first()


        async def get_user_by_email_for_write(*, session: WriteSessionDep, email: str) ->
User | None:
            """Get user by email for write operations - uses write session for
consistency."""
            result = await session.exec(select(User).where(User.email == email))
            return result.first()
```

```
        async def authenticate(*, session: ReadSessionDep, email: str, password: str) ->
User:
            """Authenticate user - uses read session for login queries."""
            user = await get_user_by_email(session=session, email=email)
            if not user or not verify_password(password, user.hashed_password):
                raise HTTPException(status_code=401, detail="Invalid credentials")

            if not user.email_verified:
                raise HTTPException(
                    status_code=403,
                    detail="Email not verified. Please verify your email first.",
                )

            if not user.is_active:
                raise HTTPException(status_code=403, detail="Inactive user")

            return user
```

app/teams/models.py

```
import uuid
from sqlmodel import Field, SQLModel


class Team(SQLModel, table=True):
    id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=True)
    name: str
    creator_id: uuid.UUID


class TeamMembership(SQLModel, table=True):
    id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=True)
    team_id: uuid.UUID = Field(foreign_key="team.id")
    user_id: uuid.UUID = Field(foreign_key="user.id")
    role: str = Field(default="member")
```

app/teams/router.py

```
from uuid import UUID
from typing import List

from fastapi import APIRouter, Depends, HTTPException, status
```

```
from . import service, schemas
from app.api.deps import SessionDep

router = APIRouter(prefix="/teams", tags=["Teams"])


@router.post("/", response_model=schemas.TeamRead)
async def create_team(data: schemas.TeamCreate, session: SessionDep):
    return await service.create_team(session, data)


@router.get("/", response_model=List[schemas.TeamRead])
async def list_teams(session: SessionDep):
    return await service.list_teams(session)


@router.get("/{team_id}", response_model=schemas.TeamRead)
async def get_team(team_id: UUID, session: SessionDep):
    team = await service.get_team(session, team_id)
    if not team:
        raise HTTPException(status_code=404, detail="Team not found")
    return team


@router.patch("/{team_id}", response_model=schemas.TeamRead)
async def update_team(team_id: UUID, data: schemas.TeamUpdate, session: SessionDep):
    team = await service.update_team(session, team_id, data)
    if not team:
        raise HTTPException(status_code=404, detail="Team not found")
    return team


@router.delete("/{team_id}", status_code=status.HTTP_204_NO_CONTENT)
async def delete_team(team_id: UUID, session: SessionDep):
    success = await service.delete_team(session, team_id)
    if not success:
        raise HTTPException(status_code=404, detail="Team not found")


@router.post("/members/", response_model=schemas.TeamMembershipRead)
```

```
async def add_member(data: schemas.TeamMembershipCreate, session: SessionDep):
    return await service.add_member(session, data)


@router.get("/{team_id}/members", response_model=List[schemas.TeamMembershipRead])
async def list_members(team_id: UUID, session: SessionDep):
    return await service.list_members(session, team_id)
```

app/teams/service.py

```
from uuid import UUID
from sqlmodel import select
from sqlmodel.ext.asyncio.session import AsyncSession

from app.teams.schemas import TeamCreate, TeamUpdate, TeamMembershipCreate
from app.teams.models import Team, TeamMembership


async def create_team(session: AsyncSession, data: TeamCreate) -> Team:
    team = Team(**data.model_dump())
    session.add(team)
    await session.commit()
    await session.refresh(team)
    return team


async def get_team(session: AsyncSession, team_id: UUID) -> Team | None:
    return await session.get(Team, team_id)


async def list_teams(session: AsyncSession) -> list[Team]:
    result = await session.exec(select(Team))
    return result.all()


async def update_team(session: AsyncSession, team_id: UUID, data: TeamUpdate) ->
Team | None:
    team = await session.get(Team, team_id)
    if not team:
        return None

    for key, value in data.model_dump(exclude_unset=True).items():
```

```
                setattr(team, key, value)

            session.add(team)
            await session.commit()
            await session.refresh(team)
            return team


        async def delete_team(session: AsyncSession, team_id: UUID) -> bool:
            team = await session.get(Team, team_id)
            if not team:
                return False

            await session.delete(team)
            await session.commit()
            return True


        async def add_member(session: AsyncSession, data: TeamMembershipCreate) ->
TeamMembership:
            membership = TeamMembership(**data.model_dump())
            session.add(membership)
            await session.commit()
            await session.refresh(membership)
            return membership


        async def list_members(session: AsyncSession, team_id: UUID) ->
list[TeamMembership]:
            result = await session.exec(
                select(TeamMembership).where(TeamMembership.team_id == team_id)
            )
            return result.all()
```

tests/users/test_service.py

```
import uuid
from unittest.mock import AsyncMock, MagicMock, patch
import pytest
from fastapi import HTTPException

from app.users.service import (
```

```
    create_user, get_user_by_email, authenticate
)
from app.users.models import User
from app.users.schemas import UserCreate


class TestCreateUser:
    """Test cases for the create_user function"""

    @pytest.mark.asyncio
    async def test_create_user_expected_use(self):
        """Test creating a user with valid data"""
        mock_session = AsyncMock()

        mock_result = AsyncMock()
        mock_result.first = MagicMock(return_value=None)
        mock_session.exec = AsyncMock(return_value=mock_result)

        user_create = UserCreate(
            full_name="John Doe",
            email="john@example.com",
            password="password123"
        )

        mock_user = User(
            id=uuid.uuid4(),
            full_name="John Doe",
            email="john@example.com",
            hashed_password="hashed_password123",
            is_active=False,
            is_superuser=False,
            email_verified=False,
            verification_code=None,
            verification_code_expires=None
        )

        with patch('app.users.service.get_password_hash') as mock_hash:
            mock_hash.return_value = "hashed_password123"
            mock_session.add = MagicMock()
            mock_session.commit = AsyncMock()
            mock_session.refresh = AsyncMock()
```

```
                with patch.object(User, 'model_validate', return_value=mock_user):
                    result = await create_user(session=mock_session,
user_create=user_create)

                    mock_hash.assert_called_once_with("password123")
                    mock_session.add.assert_called_once_with(mock_user)
                    mock_session.commit.assert_called_once()
                    mock_session.refresh.assert_called_once_with(mock_user)

                    assert result == mock_user

        @pytest.mark.asyncio
        async def test_create_user_edge_case_max_length_fields(self):
            """Test creating a user with maximum length fields"""
            mock_session = AsyncMock()

            mock_result = AsyncMock()
            mock_result.first = MagicMock(return_value=None)
            mock_session.exec = AsyncMock(return_value=mock_result)

            user_create = UserCreate(
                full_name="x" * 255,
                email="test@example.com",
                password="x" * 40
            )

            mock_user = User(
                id=uuid.uuid4(),
                full_name="x" * 255,
                email="test@example.com",
                hashed_password="hashed_password",
                is_active=False,
                is_superuser=False,
                email_verified=False,
                verification_code=None,
                verification_code_expires=None
            )

            with patch('app.users.service.get_password_hash') as mock_hash:
                mock_hash.return_value = "hashed_password"
```

```
                    mock_session.add = MagicMock()
                    mock_session.commit = AsyncMock()
                    mock_session.refresh = AsyncMock()

                    with patch.object(User, 'model_validate', return_value=mock_user):
                        result = await create_user(session=mock_session,
user_create=user_create)

                        assert result is not None

            @pytest.mark.asyncio
            async def test_create_user_failure_case_existing_email(self):
                """Test creating a user with existing email"""
                mock_session = AsyncMock()
                existing_user = User(
                    id=uuid.uuid4(),
                    email="john@example.com",
                    full_name="Existing User",
                    hashed_password="hash",
                    is_active=True,
                    is_superuser=False,
                    email_verified=True,
                    verification_code=None,
                    verification_code_expires=None
                )

                mock_result = AsyncMock()
                mock_result.first = MagicMock(return_value=existing_user)
                mock_session.exec = AsyncMock(return_value=mock_result)

                user_create = UserCreate(
                    full_name="John Doe",
                    email="john@example.com",
                    password="password123"
                )

                with pytest.raises(HTTPException) as exc_info:
                    await create_user(session=mock_session, user_create=user_create)

                assert exc_info.value.status_code == 400
                assert "already exists" in exc_info.value.detail
```

```
        class TestGetUserByEmail:
            """Test cases for the get_user_by_email function"""

            @pytest.mark.asyncio
            async def test_get_user_by_email_expected_use(self):
                """Test getting user by email when user exists"""
                mock_session = AsyncMock()
                mock_user = User(
                    id=uuid.uuid4(),
                    email="john@example.com",
                    full_name="John Doe",
                    hashed_password="hash",
                    is_active=True,
                    is_superuser=False,
                    email_verified=True,
                    verification_code=None,
                    verification_code_expires=None
                )

                mock_result = AsyncMock()
                mock_result.first = MagicMock(return_value=mock_user)
                mock_session.exec = AsyncMock(return_value=mock_result)

                result = await get_user_by_email(session=mock_session,
    email="john@example.com")

                assert result == mock_user

            @pytest.mark.asyncio
            async def test_get_user_by_email_edge_case_nonexistent_email(self):
                """Test getting user by email when user doesn't exist"""
                mock_session = AsyncMock()

                mock_result = AsyncMock()
                mock_result.first = MagicMock(return_value=None)
                mock_session.exec = AsyncMock(return_value=mock_result)

                result = await get_user_by_email(session=mock_session,
    email="nonexistent@example.com")
```

```
            assert result is None

        @pytest.mark.asyncio
        async def test_get_user_by_email_failure_case_database_error(self):
            """Test getting user by email when database query fails"""
            mock_session = AsyncMock()
            mock_session.exec.side_effect = Exception("Database connection error")

            with pytest.raises(Exception, match="Database connection error"):
                await get_user_by_email(session=mock_session, email="test@example.com")


    class TestAuthenticate:
        """Test cases for the authenticate function"""

        @pytest.mark.asyncio
        async def test_authenticate_expected_use(self):
            """Test authentication with valid credentials and verified email"""
            mock_session = AsyncMock()
            mock_user = User(
                id=uuid.uuid4(),
                email="john@example.com",
                full_name="John Doe",
                hashed_password="hashed_password",
                is_active=True,
                is_superuser=False,
                email_verified=True,
                verification_code=None,
                verification_code_expires=None
            )

            with patch('app.users.service.get_user_by_email') as mock_get_user:
                with patch('app.users.service.verify_password') as mock_verify:
                    mock_get_user.return_value = mock_user
                    mock_verify.return_value = True

                    result = await authenticate(
                        session=mock_session,
                        email="john@example.com",
                        password="password123"
```

```
                        )

                        mock_get_user.assert_called_once_with(session=mock_session,
email="john@example.com")
                        mock_verify.assert_called_once_with("password123",
"hashed_password")
                        assert result == mock_user

            @pytest.mark.asyncio
            async def test_authenticate_edge_case_unverified_email(self):
                """Test authentication with unverified email"""
                mock_session = AsyncMock()
                mock_user = User(
                    id=uuid.uuid4(),
                    email="john@example.com",
                    full_name="John Doe",
                    hashed_password="hashed_password",
                    is_active=True,
                    is_superuser=False,
                    email_verified=False,
                    verification_code=None,
                    verification_code_expires=None
                )

                with patch('app.users.service.get_user_by_email') as mock_get_user:
                    with patch('app.users.service.verify_password') as mock_verify:
                        mock_get_user.return_value = mock_user
                        mock_verify.return_value = True

                        with pytest.raises(HTTPException) as exc_info:
                            await authenticate(
                                session=mock_session,
                                email="john@example.com",
                                password="password123"
                            )

                        assert exc_info.value.status_code == 403
                        assert "Email not verified" in exc_info.value.detail

            @pytest.mark.asyncio
            async def test_authenticate_edge_case_inactive_user(self):
```

```
        """Test authentication with inactive user"""
        mock_session = AsyncMock()
        mock_user = User(
            id=uuid.uuid4(),
            email="john@example.com",
            full_name="John Doe",
            hashed_password="hashed_password",
            is_active=False,
            is_superuser=False,
            email_verified=True,
            verification_code=None,
            verification_code_expires=None
        )

        with patch('app.users.service.get_user_by_email') as mock_get_user:
            with patch('app.users.service.verify_password') as mock_verify:
                mock_get_user.return_value = mock_user
                mock_verify.return_value = True

                with pytest.raises(HTTPException) as exc_info:
                    await authenticate(
                        session=mock_session,
                        email="john@example.com",
                        password="password123"
                    )

                assert exc_info.value.status_code == 403
                assert "Inactive user" in exc_info.value.detail

    @pytest.mark.asyncio
    async def test_authenticate_failure_case_nonexistent_user(self):
        """Test authentication with non-existent user"""
        mock_session = AsyncMock()

        with patch('app.users.service.get_user_by_email') as mock_get_user:
            mock_get_user.return_value = None

            with pytest.raises(HTTPException) as exc_info:
                await authenticate(
                    session=mock_session,
                    email="nonexistent@example.com",
```

```
                    password="password123"
                )

                assert exc_info.value.status_code == 401
                assert "Invalid credentials" in exc_info.value.detail

    @pytest.mark.asyncio
    async def test_authenticate_failure_case_wrong_password(self):
        """Test authentication with wrong password"""
        mock_session = AsyncMock()
        mock_user = User(
            id=uuid.uuid4(),
            email="john@example.com",
            full_name="John Doe",
            hashed_password="hashed_password",
            is_active=True,
            is_superuser=False,
            email_verified=True,
            verification_code=None,
            verification_code_expires=None
        )

        with patch('app.users.service.get_user_by_email') as mock_get_user:
            with patch('app.users.service.verify_password') as mock_verify:
                mock_get_user.return_value = mock_user
                mock_verify.return_value = False

                with pytest.raises(HTTPException) as exc_info:
                    await authenticate(
                        session=mock_session,
                        email="john@example.com",
                        password="wrongpassword"
                    )

                assert exc_info.value.status_code == 401
                assert "Invalid credentials" in exc_info.value.detail
```

project-service

app/main.py

```
import asyncio
```

```
import logging
from contextlib import asynccontextmanager

from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
from fastapi.routing import APIRoute
from starlette.middleware.cors import CORSMiddleware

from app.api.main import api_router
from app.core.config import settings
from app.messaging.comment_consumer import start_comment_consumer
from app.projects.search_service import search_service
from app.core.elasticsearch import es_client

logger = logging.getLogger(__name__)
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
)

consumer_task = None


def custom_generate_unique_id(route: APIRoute) -> str:
    tag = route.tags[0] if route.tags else "default"
    return f"{tag}-{route.name}"


@asynccontextmanager
async def lifespan(app: FastAPI):
    global consumer_task

    logger.info("Initializing Elasticsearch connection...")
    search_initialized = await search_service.initialize()
    if search_initialized:
        logger.info("✅ Elasticsearch search service initialized successfully")
    else:
        logger.warning("⚠ Elasticsearch search service initialization failed -
search features may not work")

    consumer_task = asyncio.create_task(start_comment_consumer())
```

```
    yield

    if consumer_task:
        consumer_task.cancel()
        try:
            await consumer_task
        except asyncio.CancelledError:
            print("● Comment consumer cancelled gracefully")

    logger.info("Closing Elasticsearch connection...")
    await es_client.disconnect()


app = FastAPI(
    title=settings.PROJECT_NAME,
    openapi_url=f"{settings.API_V1_STR}/openapi.json",
    generate_unique_id_function=custom_generate_unique_id,
    lifespan=lifespan,
)


@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    logger.exception(f"Unhandled error: {exc}")
    return JSONResponse(
        status_code=500,
        content={"detail": "Internal server error"},
    )


if settings.all_cors_origins:
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.all_cors_origins,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

app.include_router(api_router, prefix=settings.API_V1_STR)
```

```
@app.on_event("startup")
async def startup_event():
    loop = asyncio.get_event_loop()
    loop.create_task(start_comment_consumer())


@app.get("/")
async def root():
    return {"message": "API is running"}


@app.get("/health")
async def health_check():
    """Gateway health check endpoint"""
    es_healthy = await es_client.health_check()

    return {
        "status": "healthy",
        "service": "project-service",
        "elasticsearch": "healthy" if es_healthy else "unavailable"
    }
```

app/core/elasticsearch.py

```
import logging
from typing import Optional, Dict, Any
from elasticsearch import AsyncElasticsearch
from elasticsearch.exceptions import ConnectionError, NotFoundError

from app.core.config import settings

logger = logging.getLogger(__name__)


class ElasticsearchClient:
    """Elasticsearch client wrapper for managing connections and operations"""

    def __init__(self):
        self.client: Optional[AsyncElasticsearch] = None
        self.is_connected = False
```

```
    async def connect(self) -> bool:
        """Establish connection to Elasticsearch"""
        try:
            self.client = AsyncElasticsearch([settings.ELASTICSEARCH_URL])
            info = await self.client.info()
            logger.info(f"Connected to Elasticsearch: {info['version']['number']}")
            self.is_connected = True
            return True
        except ConnectionError as e:
            logger.error(f"Failed to connect to Elasticsearch: {e}")
            self.is_connected = False
            return False
        except Exception as e:
            logger.error(f"Unexpected error connecting to Elasticsearch: {e}")
            self.is_connected = False
            return False

    async def disconnect(self):
        """Close Elasticsearch connection"""
        if self.client:
            await self.client.close()
            self.is_connected = False
            logger.info("Disconnected from Elasticsearch")

    async def health_check(self) -> bool:
        """Check if Elasticsearch is healthy"""
        if not self.client or not self.is_connected:
            return False
        try:
            health = await self.client.cluster.health()
            return health["status"] in ["green", "yellow"]
        except Exception as e:
            logger.error(f"Elasticsearch health check failed: {e}")
            return False

    async def create_index_if_not_exists(self, index_name: str, mapping: Dict[str,
Any]) -> bool:
            """Create index with mapping if it doesn't exist"""
            if not self.client:
                return False
```

```
            try:
                exists = await self.client.indices.exists(index=index_name)
                if not exists:
                    await self.client.indices.create(index=index_name, body=mapping)
                    logger.info(f"Created Elasticsearch index: {index_name}")
                return True
            except Exception as e:
                logger.error(f"Failed to create index {index_name}: {e}")
                return False

        async def delete_index(self, index_name: str) -> bool:
            """Delete an index"""
            if not self.client:
                return False
            try:
                await self.client.indices.delete(index=index_name)
                logger.info(f"Deleted Elasticsearch index: {index_name}")
                return True
            except NotFoundError:
                logger.warning(f"Index {index_name} not found")
                return True
            except Exception as e:
                logger.error(f"Failed to delete index {index_name}: {e}")
                return False


    es_client = ElasticsearchClient()
```

app/messaging/comment_consumer.py

```
import asyncio
import json
import logging
import uuid

import aio_pika
from sqlmodel import select

from app.core.config import settings
from app.core.db import async_session
from app.projects.models import Project
```

```
        logger = logging.getLogger(__name__)


        async def handle_comment_event(message: aio_pika.IncomingMessage):
            """Handle comment events to update project comment count"""
            async with message.process():
                try:
                    data = json.loads(message.body)
                    event_type = data["type"]
                    payload = data["data"]
                    project_id = uuid.UUID(payload["project_id"])
                    comment_id = payload.get("comment_id")
                    user_id = payload.get("user_id")

                    logger.info(f"Processing {event_type} event for project {project_id}")

                    async with async_session() as session:
                        query = select(Project).where(Project.id == project_id)
                        result = await session.execute(query)
                        project = result.scalar_one_or_none()

                        if not project:
                            logger.warning(f"Project {project_id} not found for comment
event")
                            return

                        if event_type == "comment_created":
                            project.comment_count += 1
                            logger.info(f"Incremented comment count for project {project_id}
to {project.comment_count}")

                        elif event_type == "comment_deleted":
                            project.comment_count = max(0, project.comment_count - 1)
                            logger.info(f"Decremented comment count for project {project_id}
to {project.comment_count}")

                        elif event_type == "comment_updated":
                            logger.info(f"Comment {comment_id} updated for project
{project_id}")
                            return
```

```
                await session.commit()

        except json.JSONDecodeError as e:
            logger.error(f"Failed to decode message JSON: {e}")
        except ValueError as e:
            logger.error(f"Invalid UUID in message payload: {e}")
        except Exception as e:
            logger.error(f"Error processing comment event: {e}")


async def start_comment_consumer():
    """Start the comment consumer with retry logic"""
    retries = 5
    for i in range(retries):
        try:
            connection = await aio_pika.connect_robust(settings.RABBITMQ_URL)
            channel = await connection.channel()

            await channel.set_qos(prefetch_count=10)

            queue = await channel.declare_queue(
                "comments.events", durable=True
            )
            await queue.consume(handle_comment_event)
            logger.info("✅ Comment consumer started successfully.")
            return await asyncio.Future()

        except Exception as e:
            logger.warning(
                f"❗ Failed to connect to RabbitMQ (attempt {i+1}/{retries}): {e}"
            )
            await asyncio.sleep(5)

    logger.error("❌ Could not connect to RabbitMQ after several attempts.")
```

app/projects/models.py

```
import uuid
from datetime import datetime
from typing import Optional

from sqlmodel import Field, SQLModel
```

```
class Project(SQLModel, table=True):
    __tablename__ = "projects"
    id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=True)
    user_id: uuid.UUID = Field(index=True)
    slug: str = Field(max_length=255)
    name: str = Field(max_length=255)
    description: str = Field(max_length=2048)
    url: str = Field(max_length=255)
    is_deleted: bool = Field(default=False)
    comment_count: int = Field(default=0)

    image_url: Optional[str] = Field(default=None, max_length=512)
    image_filename: Optional[str] = Field(default=None, max_length=255)
    image_size: Optional[int] = Field(default=None)  # Size in bytes
    image_content_type: Optional[str] = Field(default=None, max_length=100)
    image_uploaded_at: Optional[datetime] = Field(default=None)
```

app/projects/router.py

```
from typing import Optional
import uuid

from fastapi import APIRouter, Depends, UploadFile, File, HTTPException, status,
Query
from fastapi.responses import FileResponse
from pathlib import Path

from app.api.deps import SessionDep, UserIDDep
from app.projects.schemas import (
    ProjectCreate,
    ProjectRead,
    ProjectUpdate,
    ImageUploadResponse
)
from app.projects.search_models import (
    SearchQuery,
    SearchResponse,
    SuggestionResponse
)
from app.projects.search_service import search_service
```

```
        from app.projects.service import (
            create_project as create_project_service,
            get_projects,
            get_project_by_id,
            update_project as update_project_service,
            delete_project as delete_project_service
        )
        from app.projects.image_service import image_service

        router = APIRouter(prefix="/projects", tags=["projects"])


        @router.get("/search", response_model=SearchResponse)
        async def search_projects(
            q: str = Query(description="Search query"),
            offset: int = Query(default=0, ge=0, description="Search result offset"),
            limit: int = Query(default=20, ge=1, le=100, description="Maximum results to
return"),
            sort_by: Optional[str] = Query(default=None, description="Field to sort by"),
            sort_order: Optional[str] = Query(default="desc", regex="^(asc|desc)$",
description="Sort order"),
            user_id: Optional[str] = Query(default=None, description="Filter by user ID")
        ):
            """Search projects using Elasticsearch"""
            search_query = SearchQuery(
                query=q,
                offset=offset,
                limit=limit,
                sort_by=sort_by,
                sort_order=sort_order,
                user_id=user_id
            )
            return await search_service.search_projects(search_query)


        @router.get("/suggest", response_model=SuggestionResponse)
        async def suggest_projects(
            q: str = Query(..., description="Search query for suggestions"),
            limit: int = Query(5, ge=1, le=20, description="Maximum suggestions to return")
        ) -> SuggestionResponse:
            """
```

```
    Get autocomplete suggestions for project search.

    - **q**: Partial search query
    - **limit**: Maximum number of suggestions (1-20)
    """
    return await search_service.suggest_projects(q, limit)


@router.post("/reindex", response_model=dict)
async def reindex_projects(
    session: SessionDep,
    user_id: UserIDDep
) -> dict:
    """
    Reindex all projects in Elasticsearch.
    This endpoint is for administrative purposes.
    """
    indexed_count = await search_service.reindex_all_projects(session)
    return {
        "message": f"Reindexing completed",
        "indexed_count": indexed_count
    }


@router.get("/", response_model=dict)
async def read_projects(
    session: SessionDep, offset: int = 0, limit: int = 100
) -> dict:
    projects = await get_projects(session=session, offset=offset, limit=limit)
    return {
        "projects": [ProjectRead.model_validate(p) for p in projects],
        "total": len(projects),
        "offset": offset,
        "limit": limit,
    }


@router.post("/", response_model=ProjectRead)
async def create_project(
    session: SessionDep,
    project: ProjectCreate,
```

```
    user_id: UserIDDep,
) -> ProjectRead:
    """
    Create a new project.
    """
    project = await create_project_service(
        session=session, project_create=project, user_id=user_id
    )
    await search_service.index_project(project)
    return ProjectRead.model_validate(project)


@router.get("/{project_id}", response_model=ProjectRead)
async def read_project(
    project_id: str,
    session: SessionDep,
) -> ProjectRead:
    """
    Get a project by ID.
    """
    project = await get_project_by_id(session=session, project_id=project_id)
    if not project:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Project not found"
        )
    return ProjectRead.model_validate(project)


@router.put("/{project_id}", response_model=ProjectRead)
async def update_project(
    project_id: str,
    project_data: ProjectUpdate,
    session: SessionDep,
    user_id: UserIDDep,
) -> ProjectRead:
    """
    Update a project by ID.
    """
    project = await get_project_by_id(session=session, project_id=project_id)
    if not project:
```

```
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Project not found"
            )
        project = await update_project_service(
            session=session,
            project=project,
            project_data=project_data,
            user_id=user_id
        )
        await search_service.update_project(project)
        return ProjectRead.model_validate(project)


@router.delete("/{project_id}", response_model=dict)
async def delete_project(
    project_id: str,
    session: SessionDep,
    user_id: UserIDDep,
) -> dict:
    """
    Delete a project by ID.
    """
    project = await get_project_by_id(session=session, project_id=project_id)
    if not project:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Project not found"
        )
    project = await delete_project_service(
        session=session,
        project=project,
        user_id=user_id
    )
    await search_service.update_project(project)
    return {"message": "Project deleted successfully"}


@router.post("/{project_id}/image", response_model=ImageUploadResponse)
async def upload_project_image(
    project_id: uuid.UUID,
```

```
            session: SessionDep,
            user_id: UserIDDep,
            file: UploadFile = File(...)
        ) -> ImageUploadResponse:
            """
            Upload an image for a project. Replaces existing image if present.

            - **project_id**: UUID of the project
            - **file**: Image file (JPEG, PNG, GIF, WebP, max 10MB)
            """
            try:
                result = await image_service.upload_project_image(
                    session=session,
                    project_id=project_id,
                    file=file,
                    user_id=uuid.UUID(user_id)
                )
                project = await get_project_by_id(session=session,
project_id=str(project_id))
                if project:
                    await search_service.update_project(project)
                return result
            except HTTPException:
                raise
            except Exception as e:
                raise HTTPException(
                    status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                    detail=f"Failed to upload image: {str(e)}"
                )


        @router.delete("/{project_id}/image", response_model=dict)
        async def delete_project_image(
            project_id: uuid.UUID,
            session: SessionDep,
            user_id: UserIDDep,
        ) -> dict:
            """
            Delete a project's image.
            """
            success = await image_service.delete_project_image(
```

```
                session=session,
                project_id=project_id,
                user_id=uuid.UUID(user_id)
            )
            if success:
                project = await get_project_by_id(session=session,
project_id=str(project_id))
                if project:
                    await search_service.update_project(project)
                return {"message": "Image deleted successfully"}
            else:
                raise HTTPException(
                    status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                    detail="Failed to delete image"
                )


        @router.get("/uploads/{filename}")
        async def serve_image(filename: str):
            """
            Serve uploaded images.
            """
            file_path = Path("uploads") / filename
            if not file_path.exists():
                raise HTTPException(
                    status_code=status.HTTP_404_NOT_FOUND,
                    detail="Image not found"
                )
            return FileResponse(file_path)
```

app/projects/service.py

```
import re
import uuid
import unicodedata
from typing import List, Optional

from fastapi import Cookie, HTTPException, status
from sqlmodel import select
from sqlmodel.ext.asyncio.session import AsyncSession

from app.api.deps import SessionDep
```

```
        from app.projects.models import Project
        from app.projects.schemas import ProjectCreate, ProjectUpdate

        ALGORITHM = "HS256"


        def slugify(value: str) -> str:
            """Generate URL-friendly slug from project name"""
            value = unicodedata.normalize("NFKD", value).encode("ascii",
"ignore").decode("ascii")
            value = re.sub(r"[^\w\s-]", "", value).strip().lower()
            return re.sub(r"[\s_-]+", "-", value).strip("-")


        async def create_project(
            *, session: AsyncSession, project_create: ProjectCreate, user_id: str
        ) -> Project:
            """Create a new project with auto-generated slug"""
            slug = f"{slugify(project_create.name)}-{uuid.uuid4().hex[:6]}"
            data = {**project_create.model_dump(), "user_id": user_id, "slug": slug}

            db_obj = Project.model_validate(data)
            session.add(db_obj)
            await session.commit()
            await session.refresh(db_obj)
            return db_obj


        async def get_projects(
            session: AsyncSession, offset: int = 0, limit: int = 100
        ) -> List[Project]:
            """Get list of projects with pagination"""
            statement = select(Project).offset(offset).limit(limit)
            result = await session.exec(statement)
            return result.all()


        async def get_project_by_id(session: AsyncSession, project_id: str) ->
Optional[Project]:
            """Get a project by its ID"""
            return await session.get(Project, project_id)
```

```
async def update_project(
    session: AsyncSession,
    project: Project,
    project_data: ProjectUpdate,
    user_id: str
) -> Project:
    """Update an existing project with ownership validation"""
    # Verify user owns the project
    if str(project.user_id) != user_id:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not authorized to update this project"
        )

    update_dict = project_data.model_dump(exclude_unset=True)
    for key, value in update_dict.items():
        setattr(project, key, value)

    session.add(project)
    await session.commit()
    await session.refresh(project)
    return project


async def delete_project(
    session: AsyncSession,
    project: Project,
    user_id: str
) -> Project:
    """Soft delete a project with ownership validation"""
    # Verify user owns the project
    if str(project.user_id) != user_id:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not authorized to delete this project"
        )

    project.is_deleted = True
    await session.commit()
```

```
        await session.refresh(project)
        return project


    def validate_project_ownership(project: Project, user_id: str) -> None:
        """Validate that the user owns the project"""
        if str(project.user_id) != user_id:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Not authorized to access this project"
            )
```

app/projects/search_models.py

```
    import uuid
    from datetime import datetime
    from typing import Optional, List, Dict, Any
    from pydantic import BaseModel, Field

    from app.projects.schemas import ProjectRead


    class ProjectSearchDocument(BaseModel):
        """Elasticsearch document model for projects"""

        id: str = Field(description="Project ID")
        user_id: str = Field(description="User ID who owns the project")
        slug: str = Field(description="Project slug")
        name: str = Field(description="Project name")
        description: str = Field(description="Project description")
        url: str = Field(description="Project URL")
        comment_count: int = Field(default=0, description="Number of comments")

        # Search-specific fields
        indexed_at: datetime = Field(default_factory=datetime.utcnow, description="When
document was indexed")

        @classmethod
        def from_project(cls, project) -> "ProjectSearchDocument":
            """Create search document from Project model"""
            return cls(
                id=str(project.id),
```

```
                    user_id=str(project.user_id),
                    slug=project.slug,
                    name=project.name,
                    description=project.description,
                    url=project.url,
                    comment_count=project.comment_count
                )


        class SearchQuery(BaseModel):
            """Search query parameters"""

            query: str = Field(description="Search query string")
            offset: int = Field(default=0, ge=0, description="Search result offset")
            limit: int = Field(default=20, ge=1, le=100, description="Maximum results to
return")
            sort_by: Optional[str] = Field(default=None, description="Field to sort by")
            sort_order: Optional[str] = Field(default="desc", pattern="^(asc|desc)$",
description="Sort order")
            user_id: Optional[str] = Field(default=None, description="Filter by user ID")


        class SearchResult(BaseModel):
            """Individual search result"""

            project: ProjectRead = Field(description="Project data")
            score: float = Field(description="Search relevance score")
            highlights: Optional[Dict[str, List[str]]] = Field(default=None,
description="Highlighted text matches")


        class SearchResponse(BaseModel):
            """Search response containing results and metadata"""

            results: List[SearchResult] = Field(description="Search results")
            total: int = Field(description="Total number of matching documents")
            offset: int = Field(description="Search result offset")
            limit: int = Field(description="Maximum results returned")
            query: str = Field(description="Original search query")
            took: int = Field(description="Search execution time in milliseconds")
```

```
            max_score: Optional[float] = Field(default=None, description="Maximum relevance
score")


        class SearchSuggestion(BaseModel):
            """Search suggestion/autocomplete result"""

            text: str = Field(description="Suggested text")
            score: float = Field(description="Suggestion relevance score")


        class SuggestionResponse(BaseModel):
            """Response for search suggestions"""

            suggestions: List[SearchSuggestion] = Field(description="List of suggestions")
            query: str = Field(description="Original query")


        # Elasticsearch index mapping for projects
        PROJECT_INDEX_MAPPING = {
            "mappings": {
                "properties": {
                    "id": {"type": "keyword"},
                    "user_id": {"type": "keyword"},
                    "slug": {"type": "keyword"},
                    "name": {
                        "type": "text",
                        "analyzer": "standard",
                        "fields": {
                            "keyword": {"type": "keyword"},
                            "suggest": {
                                "type": "completion",
                                "analyzer": "simple"
                            }
                        }
                    },
                    "description": {
                        "type": "text",
                        "analyzer": "standard",
                        "fields": {
                            "keyword": {"type": "keyword"}
```

```
                }
            },
            "url": {"type": "keyword"},
            "comment_count": {"type": "integer"},
            "indexed_at": {"type": "date"}
        }
    },
    "settings": {
        "analysis": {
            "analyzer": {
                "project_analyzer": {
                    "type": "custom",
                    "tokenizer": "standard",
                    "filter": ["lowercase", "stop", "snowball"]
                }
            }
        },
        "number_of_shards": 1,
        "number_of_replicas": 0
    }
}
```

app/projects/search_service.py

```
import logging
import uuid
from typing import List, Optional, Dict, Any

from elasticsearch.exceptions import NotFoundError
from sqlmodel.ext.asyncio.session import AsyncSession

from app.core.elasticsearch import es_client
from app.core.config import settings
from app.projects.models import Project
from app.projects.schemas import ProjectRead
from app.projects.search_models import (
    ProjectSearchDocument,
    SearchQuery,
    SearchResponse,
    SearchResult,
    SuggestionResponse,
    SearchSuggestion,
```

```
            PROJECT_INDEX_MAPPING
        )

        logger = logging.getLogger(__name__)


        class ProjectSearchService:
            def __init__(self):
                self.index_name = f"{settings.ELASTICSEARCH_INDEX_PREFIX}_projects"

            async def initialize(self) -> bool:
                if not es_client.is_connected:
                    if not await es_client.connect():
                        logger.error("Failed to connect to Elasticsearch")
                        return False
                success = await es_client.create_index_if_not_exists(
                    self.index_name,
                    PROJECT_INDEX_MAPPING
                )
                if success:
                    logger.info(f"Elasticsearch search service initialized with index:
{self.index_name}")
                return success

            async def index_project(self, project: Project) -> bool:
                if not es_client.client or not es_client.is_connected:
                    logger.warning("Elasticsearch not available for indexing")
                    return False
                try:
                    doc = ProjectSearchDocument.from_project(project)
                    await es_client.client.index(
                        index=self.index_name,
                        id=str(project.id),
                        body=doc.model_dump(),
                        refresh="wait_for"
                    )
                    logger.debug(f"Indexed project {project.id} in Elasticsearch")
                    return True
                except Exception as e:
                    logger.error(f"Failed to index project {project.id}: {e}")
                    return False
```

```
            async def update_project(self, project: Project) -> bool:
                return await self.index_project(project)

            async def delete_project(self, project_id: uuid.UUID) -> bool:
                if not es_client.client or not es_client.is_connected:
                    logger.warning("Elasticsearch not available for deletion")
                    return False
                try:
                    await es_client.client.delete(
                        index=self.index_name,
                        id=str(project_id),
                        refresh="wait_for"
                    )
                    logger.debug(f"Deleted project {project_id} from search index")
                    return True
                except NotFoundError:
                    logger.warning(f"Project {project_id} not found in search index")
                    return True
                except Exception as e:
                    logger.error(f"Failed to delete project {project_id} from search index:
{e}")
                    return False

            async def search_projects(self, search_query: SearchQuery) -> SearchResponse:
                if not es_client.client or not es_client.is_connected:
                    return SearchResponse(
                        results=[],
                        total=0,
                        offset=search_query.offset,
                        limit=search_query.limit,
                        query=search_query.query,
                        took=0
                    )
                try:
                    es_query = self._build_search_query(search_query)
                    response = await es_client.client.search(
                        index=self.index_name,
                        body=es_query,
                        from_=search_query.offset,
                        size=search_query.limit
```

```
            )
            results = []
            for hit in response["hits"]["hits"]:
                source = hit["_source"]
                score = hit["_score"]
                highlights = hit.get("highlight", {})
                project = ProjectRead(
                    id=uuid.UUID(source["id"]),
                    user_id=uuid.UUID(source["user_id"]),
                    slug=source["slug"],
                    name=source["name"],
                    description=source["description"],
                    url=source["url"],
                    comment_count=source["comment_count"]
                )
                results.append(SearchResult(
                    project=project,
                    score=score,
                    highlights=highlights if highlights else None
                ))
            return SearchResponse(
                results=results,
                total=response["hits"]["total"]["value"],
                offset=search_query.offset,
                limit=search_query.limit,
                query=search_query.query,
                took=response["took"],
                max_score=response["hits"]["max_score"]
            )
        except Exception as e:
            logger.error(f"Search failed: {e}")
            return SearchResponse(
                results=[],
                total=0,
                offset=search_query.offset,
                limit=search_query.limit,
                query=search_query.query,
                took=0
            )
```

```
            async def suggest_projects(self, query: str, limit: int = 5) ->
    SuggestionResponse:
                if not es_client.client or not es_client.is_connected:
                    return SuggestionResponse(suggestions=[], query=query)
                try:
                    suggest_body = {
                        "suggest": {
                            "project_suggest": {
                                "prefix": query,
                                "completion": {
                                    "field": "name.suggest",
                                    "size": limit
                                }
                            }
                        }
                    }
                    response = await es_client.client.search(
                        index=self.index_name,
                        body=suggest_body
                    )
                    suggestions = []
                    if "suggest" in response and "project_suggest" in response["suggest"]:
                        for suggestion_group in response["suggest"]["project_suggest"]:
                            for option in suggestion_group.get("options", []):
                                suggestions.append(SearchSuggestion(
                                    text=option["text"],
                                    score=option["_score"]
                                ))
                    return SuggestionResponse(suggestions=suggestions, query=query)
                except Exception as e:
                    logger.error(f"Suggestion failed: {e}")
                    return SuggestionResponse(suggestions=[], query=query)

            def _build_search_query(self, search_query: SearchQuery) -> Dict[str, Any]:
                query_body = {
                    "query": {
                        "bool": {
                            "must": [],
                            "filter": []
                        }
                    },
```

```
                "highlight": {
                    "fields": {
                        "name": {"fragment_size": 150, "number_of_fragments": 3},
                        "description": {"fragment_size": 150, "number_of_fragments": 3}
                    }
                }
            }
            if search_query.query.strip():
                query_body["query"]["bool"]["must"].append({
                    "multi_match": {
                        "query": search_query.query,
                        "fields": ["name^3", "description^2", "slug"],
                        "type": "best_fields",
                        "fuzziness": "AUTO"
                    }
                })
            else:
                query_body["query"]["bool"]["must"].append({"match_all": {}})
            if search_query.user_id:
                query_body["query"]["bool"]["filter"].append({
                    "term": {"user_id": search_query.user_id}
                })
            if search_query.sort_by:
                sort_field = search_query.sort_by
                sort_order = search_query.sort_order or "desc"
                field_mapping = {
                    "name": "name.keyword",
                    "created_at": "indexed_at",
                    "comment_count": "comment_count"
                }
                es_field = field_mapping.get(sort_field, sort_field)
                query_body["sort"] = [{es_field: {"order": sort_order}}]
            else:
                query_body["sort"] = [
                    "_score",
                    {"indexed_at": {"order": "desc"}}
                ]
            return query_body

        async def bulk_index_projects(self, projects: List[Project]) -> int:
            if not es_client.client or not es_client.is_connected:
```

```
                    logger.warning("Elasticsearch not available for bulk indexing")
                    return 0
                if not projects:
                    return 0
                try:
                    bulk_body = []
                    for project in projects:
                        doc = ProjectSearchDocument.from_project(project)
                        bulk_body.extend([
                            {"index": {"_index": self.index_name, "_id": str(project.id)}},
                            doc.model_dump()
                        ])
                    response = await es_client.client.bulk(body=bulk_body,
refresh="wait_for")
                    successful = sum(1 for item in response["items"] if "index" in item and
item["index"]["status"] in [200, 201])
                    logger.info(f"Bulk indexed {successful}/{len(projects)} projects")
                    return successful
                except Exception as e:
                    logger.error(f"Bulk indexing failed: {e}")
                    return 0

            async def reindex_all_projects(self, session: AsyncSession) -> int:
                logger.info("Starting full reindex of projects")
                try:
                    from sqlmodel import select
                    statement = select(Project)
                    result = await session.exec(statement)
                    projects = result.all()
                    await es_client.delete_index(self.index_name)
                    await es_client.create_index_if_not_exists(self.index_name,
PROJECT_INDEX_MAPPING)
                    indexed_count = await self.bulk_index_projects(projects)
                    logger.info(f"Reindexing completed: {indexed_count} projects indexed")
                    return indexed_count
                except Exception as e:
                    logger.error(f"Reindexing failed: {e}")
                    return 0


        search_service = ProjectSearchService()
```

comment-service

app/main.py

```
import logging

from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
from fastapi.routing import APIRoute
from starlette.middleware.cors import CORSMiddleware

from app.api.main import api_router
from app.core.config import settings

logger = logging.getLogger(__name__)
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
)


def custom_generate_unique_id(route: APIRoute) -> str:
    tag = route.tags[0] if route.tags else "default"
    return f"{tag}-{route.name}"


app = FastAPI(
    title=settings.PROJECT_NAME,
    openapi_url=f"{settings.API_V1_STR}/openapi.json",
    generate_unique_id_function=custom_generate_unique_id,
)


@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    logger.exception(f"Unhandled error: {exc}")
    return JSONResponse(
        status_code=500,
        content={"detail": "Internal server error"},
    )
```

```
if settings.all_cors_origins:
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.all_cors_origins,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

app.include_router(api_router, prefix=settings.API_V1_STR)


@app.get("/")
async def root():
    return {"message": "API is running"}


@app.get("/health")
async def health_check():
    """Gateway health check endpoint"""
    return {"status": "healthy", "service": "comment-service"}
```

app/messaging/producer.py

```
import json
import logging

import aio_pika

from app.core.config import settings


async def publish_event(event_type: str, payload: dict):
    try:
        connection = await aio_pika.connect_robust(settings.RABBITMQ_URL)
        async with connection:
            channel = await connection.channel()
            await channel.default_exchange.publish(
                aio_pika.Message(
                    body=json.dumps(
                        {
```

```
                            "type": event_type,
                            "data": payload,
                        }
                    ).encode()
                ),
                routing_key="comments.events",
            )
    except Exception as e:
        logging.getLogger(__name__).exception("Failed to publish RabbitMQ event")
```

app/comments/models.py

```
import uuid
from datetime import datetime
from typing import Optional

from sqlmodel import Field, SQLModel


class Comment(SQLModel, table=True):
    __tablename__ = "comments"
    id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=True)
    user_id: uuid.UUID = Field(index=True)
    project_id: uuid.UUID = Field(index=True)
    text: str = Field(max_length=1024)
    created_at: datetime = Field(default_factory=datetime.utcnow)
    updated_at: Optional[datetime] = Field(default=None)
```

app/commetns/router.py

```
import uuid
from typing import Optional

from fastapi import APIRouter, Depends, HTTPException, Query, status

from app.api.deps import SessionDep, UserIDDep
from app.comments.schemas import CommentCreate, CommentResponse, CommentUpdate
from app.comments.service import (
    create_comment as create_comment_service,
    update_comment as update_comment_service,
    delete_comment as delete_comment_service,
    get_comment_by_id,
    get_project_comments,
```

```
        )

        router = APIRouter(prefix="/comments", tags=["comments"])


        @router.post("/", response_model=CommentResponse,
    status_code=status.HTTP_201_CREATED)
        async def create_comment(
            session: SessionDep,
            comment: CommentCreate,
            current_user_id: UserIDDep,
        ) -> CommentResponse:
            """Create a new comment"""
            try:
                user_uuid = uuid.UUID(current_user_id)
                new_comment = await create_comment_service(
                    session=session,
                    comment_create=comment,
                    user_id=user_uuid
                )
                return new_comment
            except ValueError as e:
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=str(e))


        @router.get("/{comment_id}", response_model=CommentResponse)
        async def get_comment(
            comment_id: uuid.UUID,
            session: SessionDep,
            current_user_id: UserIDDep,
        ) -> CommentResponse:
            """Get a specific comment by ID"""
            comment = await get_comment_by_id(session, comment_id)

            if not comment:
                raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Comment
    not found")

            return comment
```

```
        @router.put("/{comment_id}", response_model=CommentResponse)
        async def update_comment(
            comment_id: uuid.UUID,
            comment_data: CommentUpdate,
            session: SessionDep,
            current_user_id: UserIDDep,
        ) -> CommentResponse:
            """Update a comment by ID (only by comment author)"""
            try:
                user_uuid = uuid.UUID(current_user_id)
                comment = await update_comment_service(
                    session, comment_id, user_uuid, comment_data
                )

                if not comment:
                    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
    detail="Comment not found")

                return comment
            except ValueError as e:
                raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail=str(e))


        @router.delete("/{comment_id}", status_code=status.HTTP_204_NO_CONTENT)
        async def delete_comment(
            comment_id: uuid.UUID,
            session: SessionDep,
            current_user_id: UserIDDep,
        ):
            """Delete a comment by ID (only by comment author)"""
            try:
                user_uuid = uuid.UUID(current_user_id)
                success = await delete_comment_service(
                    session, comment_id, user_uuid
                )

                if not success:
                    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
    detail="Comment not found")

            except ValueError as e:
```

```
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail=str(e))


@router.get("/project/{project_id}", response_model=list[CommentResponse])
async def get_project_comments_endpoint(
    project_id: uuid.UUID,
    session: SessionDep,
    current_user_id: UserIDDep,
    page: int = Query(1, ge=1, description="Page number"),
    size: int = Query(20, ge=1, le=100, description="Page size"),
) -> list[CommentResponse]:
    """Get all comments for a specific project with pagination"""
    comments, total = await get_project_comments(
        session, project_id, page, size
    )

    return comments
```

app/comments/service.py

```
import uuid
from datetime import datetime
from typing import Optional

from fastapi import Cookie, HTTPException
from sqlmodel import select
from sqlmodel.ext.asyncio.session import AsyncSession

from app.api.deps import SessionDep
from app.comments.models import Comment
from app.comments.schemas import CommentCreate, CommentUpdate
from app.messaging.producer import publish_event


async def create_comment(
    session: AsyncSession,
    comment_create: CommentCreate,
    user_id: uuid.UUID
) -> Comment:
    """Create a new comment"""
    comment = Comment(
        user_id=user_id,
```

```
            project_id=comment_create.project_id,
            text=comment_create.text,
            created_at=datetime.utcnow()
        )

        session.add(comment)
        await session.commit()
        await session.refresh(comment)

        await publish_event(
            event_type="comment_created",
            payload={
                "comment_id": str(comment.id),
                "project_id": str(comment.project_id),
                "user_id": str(comment.user_id)
            },
        )

        return comment


    async def get_comment_by_id(
        session: AsyncSession,
        comment_id: uuid.UUID
    ) -> Optional[Comment]:
        """Get a comment by ID"""
        query = select(Comment).where(Comment.id == comment_id)
        result = await session.execute(query)
        return result.scalar_one_or_none()


    async def update_comment(
        session: AsyncSession,
        comment_id: uuid.UUID,
        user_id: uuid.UUID,
        comment_update: CommentUpdate
    ) -> Optional[Comment]:
        """Update a comment (only by the comment author)"""
        comment = await get_comment_by_id(session, comment_id)

        if not comment:
```

```
            return None

        if comment.user_id != user_id:
            raise ValueError("User can only edit their own comments")

        comment.text = comment_update.text
        comment.updated_at = datetime.utcnow()

        await session.commit()
        await session.refresh(comment)

        await publish_event(
            event_type="comment_updated",
            payload={
                "comment_id": str(comment.id),
                "project_id": str(comment.project_id),
                "user_id": str(comment.user_id)
            },
        )

        return comment


async def delete_comment(
    session: AsyncSession,
    comment_id: uuid.UUID,
    user_id: uuid.UUID
) -> bool:
    """Delete a comment (only by the comment author)"""
    comment = await get_comment_by_id(session, comment_id)

    if not comment:
        return False

    if comment.user_id != user_id:
        raise ValueError("User can only delete their own comments")

    project_id = comment.project_id

    await session.delete(comment)
    await session.commit()
```

```
    await publish_event(
        event_type="comment_deleted",
        payload={
            "comment_id": str(comment_id),
            "project_id": str(project_id),
            "user_id": str(user_id)
        },
    )

    return True


async def get_project_comments(
    session: AsyncSession,
    project_id: uuid.UUID,
    page: int = 1,
    size: int = 20
) -> tuple[list[Comment], int]:
    """Get comments for a project with pagination"""
    count_query = select(Comment).where(Comment.project_id == project_id)
    count_result = await session.execute(count_query)
    total = len(count_result.scalars().all())

    query = (
        select(Comment)
        .where(Comment.project_id == project_id)
        .order_by(Comment.created_at.desc())
        .offset((page - 1) * size)
        .limit(size)
    )
    result = await session.execute(query)
    comments = result.scalars().all()

    return comments, total
```

application-service

app/main.py

```
import logging
```

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
from fastapi.routing import APIRoute
from starlette.middleware.cors import CORSMiddleware

from app.api.main import api_router
from app.core.config import settings
from app.messaging.publisher import message_publisher

logger = logging.getLogger(__name__)
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
)


def custom_generate_unique_id(route: APIRoute) -> str:
    tag = route.tags[0] if route.tags else "default"
    return f"{tag}-{route.name}"


app = FastAPI(
    title=settings.PROJECT_NAME,
    openapi_url=f"{settings.API_V1_STR}/openapi.json",
    generate_unique_id_function=custom_generate_unique_id,
)


@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    logger.exception(f"Unhandled error: {exc}")
    return JSONResponse(
        status_code=500,
        content={"detail": "Internal server error"},
    )


if settings.all_cors_origins:
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.all_cors_origins,
```

```
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )


@app.on_event("startup")
async def startup_event():
    """Initialize connections on startup"""
    try:
        await message_publisher.connect()
        logger.info("Application startup completed")
    except Exception as e:
        logger.error(f"Failed to initialize connections: {e}")


@app.on_event("shutdown")
async def shutdown_event():
    """Clean up connections on shutdown"""
    try:
        await message_publisher.disconnect()
        logger.info("Application shutdown completed")
    except Exception as e:
        logger.error(f"Error during shutdown: {e}")


app.include_router(api_router, prefix=settings.API_V1_STR)


@app.get("/")
async def root():
    return {"message": "Application Service API is running"}


@app.get("/health")
async def health_check():
    """Health check endpoint"""
    return {"status": "healthy", "service": "application-service"}
```

app/messaging/publisher.py

```
import json
```

```
import logging
import uuid
from typing import Any, Dict

import aio_pika
from aio_pika import Message

from app.core.config import settings

logger = logging.getLogger(__name__)


class MessagePublisher:
    def __init__(self):
        self.connection = None
        self.channel = None

    async def connect(self):
        try:
            self.connection = await aio_pika.connect_robust(settings.RABBITMQ_URL)
            self.channel = await self.connection.channel()
            self.exchange = await self.channel.declare_exchange(
                "application_events",
                aio_pika.ExchangeType.TOPIC,
                durable=True
            )
            logger.info("Connected to RabbitMQ successfully")
        except Exception as e:
            logger.error(f"Failed to connect to RabbitMQ: {e}")
            raise

    async def disconnect(self):
        if self.connection and not self.connection.is_closed:
            await self.connection.close()
            logger.info("Disconnected from RabbitMQ")

    async def publish_application_event(
        self,
        event_type: str,
        application_id: uuid.UUID,
        user_id: uuid.UUID,
```

```
                project_id: uuid.UUID,
                additional_data: Dict[str, Any] = None
            ):
                if not self.channel:
                    await self.connect()

                message_data = {
                    "event_type": event_type,
                    "application_id": str(application_id),
                    "user_id": str(user_id),
                    "project_id": str(project_id),
                    "timestamp": str(uuid.uuid4()),
                    **(additional_data or {})
                }

                message = Message(
                    json.dumps(message_data).encode(),
                    content_type="application/json",
                    message_id=str(uuid.uuid4()),
                )

                routing_key = f"application.{event_type}"

                try:
                    await self.exchange.publish(message, routing_key=routing_key)
                    logger.info(f"Published {event_type} event for application
{application_id}")
                except Exception as e:
                    logger.error(f"Failed to publish message: {e}")
                    raise

            async def publish_application_created(
                self,
                application_id: uuid.UUID,
                user_id: uuid.UUID,
                project_id: uuid.UUID,
                message: str = None
            ):
                await self.publish_application_event(
                    "created",
                    application_id,
```

```
            user_id,
            project_id,
            {"message": message}
        )

    async def publish_application_updated(
        self,
        application_id: uuid.UUID,
        user_id: uuid.UUID,
        project_id: uuid.UUID
    ):
        await self.publish_application_event(
            "updated",
            application_id,
            user_id,
            project_id
        )

    async def publish_application_reviewed(
        self,
        application_id: uuid.UUID,
        user_id: uuid.UUID,
        project_id: uuid.UUID,
        reviewer_id: uuid.UUID,
        status: str,
        review_message: str = None
    ):
        await self.publish_application_event(
            "reviewed",
            application_id,
            user_id,
            project_id,
            {
                "reviewer_id": str(reviewer_id),
                "status": status,
                "review_message": review_message
            }
        )

    async def publish_application_withdrawn(
        self,
```

```
            application_id: uuid.UUID,
            user_id: uuid.UUID,
            project_id: uuid.UUID
        ):
            await self.publish_application_event(
                "withdrawn",
                application_id,
                user_id,
                project_id
            )


message_publisher = MessagePublisher()
```

app/applications/models.py

```
import uuid
from datetime import datetime
from enum import Enum
from typing import Optional

from sqlmodel import Field, SQLModel


class ApplicationStatus(str, Enum):
    """Enumeration for application status values"""
    PENDING = "pending"
    APPROVED = "approved"
    REJECTED = "rejected"
    WITHDRAWN = "withdrawn"


class Application(SQLModel, table=True):
    """Model for user applications to participate in projects"""
    __tablename__ = "applications"

    id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=True)
    user_id: uuid.UUID = Field(index=True, description="ID of the user applying")
    project_id: uuid.UUID = Field(index=True, description="ID of the project being
applied to")
    status: ApplicationStatus = Field(default=ApplicationStatus.PENDING, index=True)
```

```
            message: Optional[str] = Field(default=None, max_length=2048,
description="Application message from user")

            application_date: datetime = Field(default_factory=datetime.utcnow, index=True)
            updated_at: datetime = Field(default_factory=datetime.utcnow)
            reviewed_at: Optional[datetime] = Field(default=None, description="When the
application was reviewed")

            reviewer_id: Optional[uuid.UUID] = Field(default=None, description="ID of the
user who reviewed the application")
            review_message: Optional[str] = Field(default=None, max_length=1024,
description="Review message from project owner")

            is_deleted: bool = Field(default=False, index=True)
```

app/applications/router.py

```
import uuid
from typing import Optional

from fastapi import APIRouter, Depends, HTTPException, Query, status

from app.applications.models import ApplicationStatus
from app.applications.schemas import (
    ApplicationCreate,
    ApplicationListResponse,
    ApplicationResponse,
    ApplicationReview,
    ApplicationUpdate,
)
from app.applications.service import (
    create_application as create_application_service,
    get_application_by_id,
    get_user_applications,
    get_project_applications,
    update_application as update_application_service,
    review_application as review_application_service,
    withdraw_application as withdraw_application_service,
    delete_application as delete_application_service,
    is_project_owner,
)
from app.api.deps import SessionDep, UserIDDep
```

```
        from app.messaging.publisher import message_publisher

        router = APIRouter()


        @router.post("/", response_model=ApplicationResponse,
    status_code=status.HTTP_201_CREATED)
        async def create_application(
            session: SessionDep,
            application_data: ApplicationCreate,
            current_user_id: UserIDDep
        ):
            try:
                user_uuid = uuid.UUID(current_user_id)
                application = await create_application_service(
                    session, user_uuid, application_data
                )
                await message_publisher.publish_application_created(
                    application.id,
                    application.user_id,
                    application.project_id,
                    application.message
                )
                return application
            except ValueError as e:
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=str(e))


        @router.get("/", response_model=ApplicationListResponse)
        async def get_user_applications_endpoint(
            session: SessionDep,
            current_user_id: UserIDDep,
            page: int = Query(1, ge=1, description="Page number"),
            size: int = Query(10, ge=1, le=100, description="Page size"),
            status_filter: Optional[ApplicationStatus] = Query(None, alias="status",
    description="Filter by status"),
        ):
            user_uuid = uuid.UUID(current_user_id)
            applications, total = await get_user_applications(
                session, user_uuid, page, size, status_filter
            )
```

```
            has_next = (page * size) < total
            return ApplicationListResponse(
                applications=applications,
                total=total,
                page=page,
                size=size,
                has_next=has_next
            )


        @router.get("/{application_id}", response_model=ApplicationResponse)
        async def get_application(
            session: SessionDep,
            application_id: uuid.UUID,
            current_user_id: UserIDDep
        ):
            application = await get_application_by_id(session, application_id)
            if not application:
                raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Application not found")
            user_uuid = uuid.UUID(current_user_id)
            if application.user_id != user_uuid:
                raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Access
denied")
            return application


        @router.put("/{application_id}", response_model=ApplicationResponse)
        async def update_application(
            session: SessionDep,
            application_id: uuid.UUID,
            update_data: ApplicationUpdate,
            current_user_id: UserIDDep
        ):
            try:
                user_uuid = uuid.UUID(current_user_id)
                application = await update_application_service(
                    session, application_id, user_uuid, update_data
                )
                if not application:
```

```
                    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Application not found")
                await message_publisher.publish_application_updated(
                    application.id,
                    application.user_id,
                    application.project_id
                )
                return application
            except ValueError as e:
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=str(e))


        @router.post("/{application_id}/review", response_model=ApplicationResponse)
        async def review_application(
            session: SessionDep,
            application_id: uuid.UUID,
            review_data: ApplicationReview,
            current_user_id: UserIDDep
        ):
            try:
                user_uuid = uuid.UUID(current_user_id)
                application = await review_application_service(
                    session, application_id, user_uuid, review_data
                )
                if not application:
                    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Application not found")
                await message_publisher.publish_application_reviewed(
                    application.id,
                    application.user_id,
                    application.project_id,
                    user_uuid,
                    application.status.value,
                    application.review_message
                )
                return application
            except ValueError as e:
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=str(e))


        @router.post("/{application_id}/withdraw", response_model=ApplicationResponse)
```

```
        async def withdraw_application(
            application_id: uuid.UUID,
            session: SessionDep,
            current_user_id: UserIDDep
        ):
            try:
                user_uuid = uuid.UUID(current_user_id)
                application = await withdraw_application_service(
                    session, application_id, user_uuid
                )
                if not application:
                    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
    detail="Application not found")
                await message_publisher.publish_application_withdrawn(
                    application.id,
                    application.user_id,
                    application.project_id
                )
                return application
            except ValueError as e:
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=str(e))


        @router.delete("/{application_id}", status_code=status.HTTP_204_NO_CONTENT)
        async def delete_application(
            application_id: uuid.UUID,
            session: SessionDep,
            current_user_id: UserIDDep
        ):
            try:
                user_uuid = uuid.UUID(current_user_id)
                success = await delete_application_service(
                    session, application_id, user_uuid
                )
                if not success:
                    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
    detail="Application not found")
            except ValueError as e:
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=str(e))
```

```
@router.get("/projects/{project_id}", response_model=ApplicationListResponse)
async def get_project_applications_endpoint(
    session: SessionDep,
    project_id: uuid.UUID,
    current_user_id: UserIDDep,
    page: int = Query(1, ge=1, description="Page number"),
    size: int = Query(10, ge=1, le=100, description="Page size"),
    status_filter: Optional[ApplicationStatus] = Query(None, alias="status",
description="Filter by status"),
):
    user_uuid = uuid.UUID(current_user_id)
    if not await is_project_owner(project_id, user_uuid):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Only project owners can view project applications"
        )
    applications, total = await get_project_applications(
        session, project_id, page, size, status_filter
    )
    has_next = (page * size) < total
    return ApplicationListResponse(
        applications=applications,
        total=total,
        page=page,
        size=size,
        has_next=has_next
    )
```

app/applications/service.py

```
import uuid
from datetime import datetime
from typing import Optional

import httpx
from sqlalchemy import and_, func, select
from sqlmodel.ext.asyncio.session import AsyncSession

from app.applications.models import Application, ApplicationStatus
from app.applications.schemas import ApplicationCreate, ApplicationReview,
ApplicationUpdate
from app.core.config import settings
```

```
        async def is_project_owner(project_id: uuid.UUID, user_id: uuid.UUID) -> bool:
            try:
                project_service_url = getattr(settings, 'PROJECT_SERVICE_URL',
    'http://project_project-service:8000')
                async with httpx.AsyncClient() as client:
                    response = await
    client.get(f"{project_service_url}/projects/{project_id}")
                    if response.status_code == 404:
                        return False
                    elif response.status_code == 200:
                        project_data = response.json()
                        return str(project_data.get("user_id")) == str(user_id)
                    else:
                        return False
            except httpx.RequestError:
                return False


        async def create_application(
            session: AsyncSession,
            user_id: uuid.UUID,
            application_data: ApplicationCreate
        ) -> Application:
            existing_query = select(Application).where(
                and_(
                    Application.user_id == user_id,
                    Application.project_id == application_data.project_id,
                    Application.status.in_([ApplicationStatus.PENDING,
    ApplicationStatus.APPROVED]),
                    Application.is_deleted == False
                )
            )
            existing_result = await session.execute(existing_query)
            if existing_result.first():
                raise ValueError("User already has an active application for this project")

            application = Application(
                user_id=user_id,
                project_id=application_data.project_id,
```

```
            message=application_data.message,
        )
        session.add(application)
        await session.commit()
        await session.refresh(application)
        return application


async def get_application_by_id(
    session: AsyncSession,
    application_id: uuid.UUID
) -> Optional[Application]:
    query = select(Application).where(
        and_(
            Application.id == application_id,
            Application.is_deleted == False
        )
    )
    result = await session.execute(query)
    return result.scalar_one_or_none()


async def get_user_applications(
    session: AsyncSession,
    user_id: uuid.UUID,
    page: int = 1,
    size: int = 10,
    status: Optional[ApplicationStatus] = None
) -> tuple[list[Application], int]:
    conditions = [
        Application.user_id == user_id,
        Application.is_deleted == False
    ]
    if status:
        conditions.append(Application.status == status)

    count_query = select(func.count(Application.id)).where(and_(*conditions))
    count_result = await session.execute(count_query)
    total = count_result.scalar()

    query = (
```

```
            select(Application)
            .where(and_(*conditions))
            .order_by(Application.application_date.desc())
            .offset((page - 1) * size)
            .limit(size)
        )
        result = await session.execute(query)
        applications = result.scalars().all()

        return applications, total


    async def get_project_applications(
        session: AsyncSession,
        project_id: uuid.UUID,
        page: int = 1,
        size: int = 10,
        status: Optional[ApplicationStatus] = None
    ) -> tuple[list[Application], int]:
        conditions = [
            Application.project_id == project_id,
            Application.is_deleted == False
        ]
        if status:
            conditions.append(Application.status == status)

        count_query = select(func.count(Application.id)).where(and_(*conditions))
        count_result = await session.execute(count_query)
        total = count_result.scalar()

        query = (
            select(Application)
            .where(and_(*conditions))
            .order_by(Application.application_date.desc())
            .offset((page - 1) * size)
            .limit(size)
        )
        result = await session.execute(query)
        applications = result.scalars().all()

        return applications, total
```

```
async def update_application(
    session: AsyncSession,
    application_id: uuid.UUID,
    user_id: uuid.UUID,
    update_data: ApplicationUpdate
) -> Optional[Application]:
    application = await get_application_by_id(session, application_id)
    if not application:
        return None
    if application.user_id != user_id:
        raise ValueError("User can only update their own applications")
    if application.status != ApplicationStatus.PENDING:
        raise ValueError("Can only update pending applications")

    if update_data.message is not None:
        application.message = update_data.message

    application.updated_at = datetime.utcnow()
    await session.commit()
    await session.refresh(application)
    return application


async def review_application(
    session: AsyncSession,
    application_id: uuid.UUID,
    reviewer_id: uuid.UUID,
    review_data: ApplicationReview
) -> Optional[Application]:
    application = await get_application_by_id(session, application_id)
    if not application:
        return None
    if application.status != ApplicationStatus.PENDING:
        raise ValueError("Can only review pending applications")

    application.status = review_data.status
    application.reviewer_id = reviewer_id
    application.review_message = review_data.review_message
    application.reviewed_at = datetime.utcnow()
```

```
    application.updated_at = datetime.utcnow()

    await session.commit()
    await session.refresh(application)
    return application


async def withdraw_application(
    session: AsyncSession,
    application_id: uuid.UUID,
    user_id: uuid.UUID
) -> Optional[Application]:
    application = await get_application_by_id(session, application_id)
    if not application:
        return None
    if application.user_id != user_id:
        raise ValueError("User can only withdraw their own applications")
    if application.status not in [ApplicationStatus.PENDING]:
        raise ValueError("Can only withdraw pending applications")

    application.status = ApplicationStatus.WITHDRAWN
    application.updated_at = datetime.utcnow()
    await session.commit()
    await session.refresh(application)
    return application


async def delete_application(
    session: AsyncSession,
    application_id: uuid.UUID,
    user_id: uuid.UUID
) -> bool:
    application = await get_application_by_id(session, application_id)
    if not application:
        return False
    if application.user_id != user_id:
        raise ValueError("User can only delete their own applications")

    application.is_deleted = True
    application.updated_at = datetime.utcnow()
    await session.commit()
```

```
            return True
```

distributed-db

monitor/monitor.py

```
        import logging
        import os
        from datetime import datetime
        from typing import Dict

        import psycopg2
        import redis
        from fastapi import FastAPI, HTTPException
        from pydantic import BaseModel
        import uvicorn

        logging.basicConfig(level=logging.INFO)
        logger = logging.getLogger(__name__)

        DB_URL = os.getenv("DB_URL",
    "postgresql://postgres:password@localhost:5432/distributed_db")
        REDIS_URL = os.getenv("REDIS_URL", "redis://localhost:6379")

        app = FastAPI(title="Database Monitor", version="1.0.0")


        class HealthStatus(BaseModel):
            service: str
            status: str
            details: Dict
            timestamp: datetime


        class DatabaseMonitor:
            def __init__(self):
                self.redis_client = None
                self.db_conn = None

            async def initialize(self):
                try:
                    self.redis_client = redis.from_url(REDIS_URL)
```

```
                    self.redis_client.ping()
                    logger.info("Connected to Redis")

                    self.db_conn = psycopg2.connect(DB_URL)
                    logger.info("Connected to database")
                except Exception as e:
                    logger.error(f"Failed to initialize connections: {e}")
                    raise

            def check_database_health(self) -> HealthStatus:
                try:
                    with self.db_conn.cursor() as cursor:
                        cursor.execute("SELECT 1")
                        cursor.execute("SELECT COUNT(*) FROM pg_stat_activity")
                        connection_count = cursor.fetchone()[0]

                        cursor.execute("SELECT COUNT(*) FROM pg_stat_activity WHERE state =
'active'")
                        active_queries = cursor.fetchone()[0]

                        cursor.execute("SELECT
pg_size_pretty(pg_database_size(current_database()))")
                        db_size = cursor.fetchone()[0]

                        details = {
                            "connections": connection_count,
                            "active_queries": active_queries,
                            "database_size": db_size
                        }

                        return HealthStatus(
                            service="database",
                            status="healthy",
                            details=details,
                            timestamp=datetime.now()
                        )

                except Exception as e:
                    logger.error(f"Database health check failed: {e}")
                    return HealthStatus(
                        service="database",
```

```
                        status="unhealthy",
                        details={"error": str(e)},
                        timestamp=datetime.now()
                    )

            def check_redis_health(self) -> HealthStatus:
                try:
                    info = self.redis_client.info()
                    details = {
                        "connected_clients": info.get("connected_clients", 0),
                        "used_memory": info.get("used_memory", 0),
                        "uptime_in_seconds": info.get("uptime_in_seconds", 0)
                    }

                    return HealthStatus(
                        service="redis",
                        status="healthy",
                        details=details,
                        timestamp=datetime.now()
                    )

                except Exception as e:
                    logger.error(f"Redis health check failed: {e}")
                    return HealthStatus(
                        service="redis",
                        status="unhealthy",
                        details={"error": str(e)},
                        timestamp=datetime.now()
                    )

            def get_system_metrics(self) -> Dict:
                try:
                    db_health = self.check_database_health()
                    redis_health = self.check_redis_health()
                    all_healthy = all(h.status == "healthy" for h in [db_health,
redis_health])

                    with self.db_conn.cursor() as cursor:
                        cursor.execute("SELECT COUNT(*) FROM saga_transactions WHERE status
= 'started'")
                        active_sagas = cursor.fetchone()[0]
```

```
                    cursor.execute("SELECT COUNT(*) FROM saga_transactions")
                    total_sagas = cursor.fetchone()[0]

                return {
                    "overall_status": "healthy" if all_healthy else "degraded",
                    "services": {
                        "database": db_health.dict(),
                        "redis": redis_health.dict()
                    },
                    "metrics": {
                        "active_sagas": active_sagas,
                        "total_sagas": total_sagas
                    },
                    "timestamp": datetime.now().isoformat()
                }

            except Exception as e:
                logger.error(f"Failed to get system metrics: {e}")
                return {
                    "overall_status": "error",
                    "error": str(e),
                    "timestamp": datetime.now().isoformat()
                }


monitor = DatabaseMonitor()


@app.on_event("startup")
async def startup_event():
    await monitor.initialize()


@app.get("/health")
async def health_check():
    return {"status": "ok", "timestamp": datetime.now()}


@app.get("/metrics")
async def get_metrics():
```

```
            return monitor.get_system_metrics()


        @app.get("/database/stats")
        async def get_database_stats():
            try:
                with monitor.db_conn.cursor() as cursor:
                    cursor.execute("""
                        SELECT schemaname, tablename,
pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) as size
                        FROM pg_tables
                        WHERE schemaname = 'public'
                        ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC
                    """)
                    table_sizes = [{"schema": row[0], "table": row[1], "size": row[2]} for
row in cursor.fetchall()]

                    cursor.execute("""
                        SELECT state, COUNT(*)
                        FROM pg_stat_activity
                        WHERE pid <> pg_backend_pid()
                        GROUP BY state
                    """)
                    connection_states = [{"state": row[0], "count": row[1]} for row in
cursor.fetchall()]

                    return {
                        "table_sizes": table_sizes,
                        "connection_states": connection_states,
                        "timestamp": datetime.now().isoformat()
                    }

            except Exception as e:
                logger.error(f"Failed to get database stats: {e}")
                raise HTTPException(status_code=500, detail=str(e))


        if __name__ == "__main__":
            uvicorn.run(app, host="0.0.0.0", port=8080)
```