

ПВНЗ «ЄВРОПЕЙСЬКИЙ УНІВЕРСИТЕТ»

**КАФЕДРА КОМП'ЮТЕРНИХ НАУК ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
СПЕЦІАЛЬНІСТЬ «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»
ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА «ІНЖЕНЕРІЯ ПРОГРАМНОГО
ЗАБЕЗПЕЧЕННЯ»**

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

на тему:

«Розробка веб-додатку із застосуванням мікросервісної архітектури»

Студент: _____ Шевченко Юрій Ігорович
- _____
(підпис студента) (прізвище, ім'я, по-батькові студента)

Науковий керівник: _____ Савченко Я.О.
- _____
(підпис наукового керівника) (прізвище, ініціали)

Допущено до захисту: ” ____ “ _____ 2025 р.

Завідувач кафедри: _____ д-р.техн.наук, доц. Сушинський О.Є.
- _____
(підпис) (прізвище, ініціали, наукове звання і ступінь)

Декан факультету: _____ Яровий Р.О.
- _____
(підпис) (прізвище, ініціали)

Завідувач

навчальної частини:

-

(підпис)

Козинець

А.О.

(прізвище, ініціали)

Київ - 2025

АНОТАЦІЯ

Дослідження присвячене створенню веб-додатку для роботи з мультимедійним відеоконтентом із застосуванням мікросервісної архітектури, що відповідає сучасним вимогам до гнучкості й масштабованості програмних систем. Актуальність теми зумовлена переходом галузі до розподілених підходів, які дозволяють адаптувати додатки до динамічних потреб користувачів і забезпечувати їхню надійну роботу в умовах зростаючого навантаження. Метою було розробити платформу, що забезпечує базові функції управління контентом і його доставки, а також оцінити її потенціал для практичного застосування. Для досягнення мети проведено аналіз теоретичних засад сучасних архітектур, порівняння їх із традиційними методами та здійснено огляд інструментів, які підтримують розподілену розробку. На основі дослідження спроектовано систему з автономних модулів, кожен із яких відповідає за окрему функцію. Реалізація включала створення інтерфейсу для взаємодії з користувачами, обробки даних і забезпечення цілісної інтеграції модулів у єдину платформу. Тестування підтвердило здатність системи успішно виконувати основні сценарії, а оцінка користувацького досвіду засвідчила зручність платформи. Запропоновані напрями вдосконалення охоплюють підвищення масштабованості, автоматизацію внутрішніх процесів, адаптацію до хмарних середовищ і вдосконалення обробки помилок для зменшення впливу виявлених недоліків. Результати мають значення для освітньої сфери, надаючи приклад практичного застосування сучасних технологій, і можуть слугувати основою для створення складніших систем. Подальші дослідження варто спрямувати на розробку механізмів безпеки, автоматизованого моніторингу та оптимізації продуктивності, щоб розширити функціональні можливості платформи й забезпечити її готовність до реальних умов експлуатації.

Ключові слова: мікросервісна архітектура, веб-додаток, мультимедійний контент, розподілені системи, Java Spring Boot.

ABSTRACT

The research is dedicated to the development of a web application for handling multimedia video content using a microservices architecture, which aligns with modern requirements for flexibility and scalability in software systems. The topic's relevance arises from the industry's transition to distributed approaches, enabling applications to adapt to dynamic user needs and ensure reliable operation under increasing loads. The objective was to create a platform that supports core content management and delivery functions while evaluating its potential for practical application. To achieve this, a theoretical analysis of contemporary architectural principles was conducted, comparing them with traditional methods and reviewing tools that facilitate distributed development. Based on this analysis, a system was designed with autonomous modules, each responsible for a specific function. The implementation involved creating interfaces for user interaction, data processing, and seamless integration of modules into a unified platform. Testing confirmed the system's ability to successfully execute primary scenarios, while the evaluation of user experience highlighted the platform's usability. Proposed enhancements include improving scalability, automating internal processes, adapting to cloud environments, and refining error handling to mitigate identified shortcomings. The findings are significant for educational contexts, providing a practical example of applying modern technologies, and can serve as a foundation for developing more complex systems. Future research should focus on developing security mechanisms, automated monitoring, and performance optimization to enhance the platform's capabilities and ensure its readiness for real-world deployment and usage.

Keywords: microservices architecture, web application, multimedia content, distributed systems, Java Spring Boot.

ПВНЗ «ЄВРОПЕЙСЬКИЙ УНІВЕРСИТЕТ»
КАФЕДРА КОМП'ЮТЕРНИХ НАУК ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
СПЕЦІАЛЬНІСТЬ «121 ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»
ОСВІТНЬО-ПРОФЕСІЙНА ПРОГРАМА «ІНЖЕНЕРІЯ ПРОГРАМНОГО
ЗАБЕЗПЕЧЕННЯ»

ЗАВДАННЯ

на випускню кваліфікаційну бакалаврську роботу студентів:

Шевченку Юрію Ігоровичу
 (прізвище, ім'я по-батькові студента)

1. Тема роботи «Розробка веб-додатку із застосуванням мікросервісної архітектури»

2. Затверджена наказом по університету від _____ **202** **р. №** _____

3. Цільова установка на вихідні дані до роботи

Цільова установка полягає у розробці веб-додатку для управління відеоконтентом із застосуванням мікросервісної архітектури, що забезпечує гнучкість і масштабність. Вихідними даними є теоретичні основи розподілених систем і аналіз інструментів для їхньої реалізації.

4. Перелік графічного матеріалу (з точним визначенням обов'язкових креслень) схема мікросервісної архітектури системи, схема взаємодії компонентів додатку, скріншоти інтерфейсу користувача

5. Консультанти по роботі, із зазначенням розділів, що їх стосуються:

Розділ	Науковий керівник	Підпис, дата	
		Завдання видав	Завдання отримав
Розділ 1	Савченко Я.О.	04.11.24	04.11.24
Розділ 2	Савченко Я.О.	03.03.25	03.03.25
Розділ 3	Савченко Я.О.	24.03.25	24.03.25

6. Зміст випускної кваліфікаційної бакалаврської роботи (перелік питань)

1. Аналіз теоретичних основ мікросервісної архітектури та її порівняння з традиційними підходами.
2. Вибір і обґрунтування інструментів для розробки розподілених систем.
3. Проєктування та розробка веб-додатку з автономними модулями.
4. Тестування системи, оцінка користувацького досвіду та ефективності.
5. Визначення недоліків і пропозиції щодо вдосконалення платформи.

7. Календарний план виконання випускної кваліфікаційної бакалаврської роботи

№	Назва етапів випускної кваліфікаційної бакалаврської роботи	Термін виконання етапів випускної кваліфікаційної бакалаврської роботи	
		За планом	Фактично
1	Обрання теми випускної кваліфікаційної магістерської роботи (ВКБР)	05.08.24 – 13.09.24	05.08.24 – 13.09.24
2	Опрацювання літератури та інших джерел для написання ВКБР	16.09.24 – 01.11.24	16.09.24 – 01.11.24
3	Виконання 1 розділу ВКБР	04.11.24 – 28.02.25	04.11.24 – 28.02.25
4	Виконання 2 розділу ВКБР	03.03.25 – 21.03.25	03.03.25 – 21.03.25
5	Виконання 3 розділу ВКБР, висновки	24.03.25 – 18.04.25	24.03.25 – 18.04.25
6	Передзахист та проведення перевірки на плагіат	21.04.24 – 02.05.25	21.04.24 – 02.05.25
7	Відгук керівника, рецензія	01.05.25 – 12.05.25	01.05.25 – 12.05.25

8. Дата видачі завдання ”_____“ 202_____ р.

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

ЗМІСТ

ВСТУП	3
РОЗДІЛ 1. ТЕОРЕТИЧНІ ЗАСАДИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ	5
1.1. Поняття, принципи та основні концепції мікросервісної архітектури	5
1.2. Порівняльний аналіз монолітної та мікросервісної архітектур: переваги, недоліки та сфери застосування	14
1.3. Ключові технології та інструменти для розробки та підтримки мікросервісів	21
Висновки до розділу 1	28
РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБ-ДОДАТКУ НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ	29
2.1. Формулювання завдання та визначення основних вимог до веб-додатку	29
2.2. Архітектурне проєктування: структура системи, моделювання даних і взаємодія мікросервісів	44
2.3. Реалізація ключових мікросервісів: реєстрація, каталогізація, стрімінг та маршрутизація відеоконтенту	51
Висновки до розділу 2	58
РОЗДІЛ 3. ТЕСТУВАННЯ, АНАЛІЗ ФУНКЦІОНАЛЬНОСТІ ТА ПЕРСПЕКТИВИ РОЗВИТКУ ВЕБ-ДОДАТКУ	59
3.1. Методи тестування мікросервісів та забезпечення їх надійності	59
3.2. Аналіз роботи системи: користувацький досвід і обробка типових сценаріїв	67
3.3. Оцінка ефективності та пропозиції щодо вдосконалення функціональності	74
Висновки до розділу 3	81
ВИСНОВКИ	82
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	84
ДОДАТКИ	87

ВСТУП

Сучасний розвиток інформаційних технологій висуває вимоги до розробки програмних систем, що здатні швидко реагувати на зміни, масштабуватися для підтримки зростаючого навантаження та забезпечувати стабільну роботу в умовах конкуренції. Традиційні підходи, зокрема монолітна архітектура, дедалі частіше поступаються місцем новим рішенням, де мікросервісна архітектура займає провідне місце. Її популярність зумовлена можливістю розділяти комплексні системи на невеликі, автономні компоненти, що полегшує їх розробку, тестування та підтримку. Водночас такий підхід створює нові виклики для розробників, пов'язані з координацією сервісів, обробкою взаємодії між ними та забезпеченням надійності. Порівняно з класичними методами побудови додатків, мікросервіси пропонують гнучкість та стійкість, що робить їх актуальними для створення веб-додатків, таких як платформи для потокового відео чи управління контентом. Саме тому розробка веб-додатку з використанням мікросервісної архітектури є актуальною темою, що сприяє розвитку галузі програмної інженерії, відповідаючи сучасним потребам у швидкості та якості програмних продуктів.

Метою роботи є створення веб-додатку для потокового відтворення відеоконтенту з використанням мікросервісної архітектури та оцінка її ефективності в реальних умовах.

Для досягнення поставленої мети було визначено такі завдання:

- Дослідити теоретичні засади мікросервісів
- Порівняти їх із монолітним підходом
- Обрати відповідні технології
- Спроекувати архітектуру системи
- Реалізувати ключові сервіси
- Провести тестування й аналіз роботи додатку
- Запропонувати шляхи його вдосконалення

Виконання цих кроків дозволило отримати цілісний результат – працездатну систему з чіткими перспективами розвитку.

Об'єктом дослідження є процес створення веб-додатків на основі мікросервісних архітектур, що охоплює етапи проєктування, розробки та експлуатації таких систем.

Предметом дослідження виступає мікросервісна архітектура як конкретний підхід до побудови веб-додатку для роботи з відеоконтентом. Акцент робиться саме на цій частині, оскільки вона визначає особливості реалізації, взаємодії компонентів і способи забезпечення функціональності.

Практична значущість отриманих результатів полягає у створенні функціонального веб-додатку, який може слугувати основою для подальшого розвитку платформ потокового відео. Розроблена система демонструє застосування мікросервісів у реальному проєкті, що має цінність для теоретичних курсів із програмування та архітектури програмного забезпечення. Результати можуть бути використані як приклад у навчальному процесі для ілюстрації переваг і викликів розподілених систем. Крім того, запропоновані методи тестування та оптимізації можуть бути використані у подібних проєктах, які потребують високої доступності та гнучкості.

Таким чином, робота стала своєрідним синтезом теоретичного огляду та практичного втілення. Це дозволило не тільки створити повнофункціональний веб-додаток, але й ретельно вивчити його функціональні особливості в контексті мікросервісної архітектури. Таке поєднання підходів дозволило оцінити, як теоретичні засади знаходять своє вираження в конкретному проєкті, висвітлюючи сильні аспекти системи, такі як гнучкість і незалежність компонентів, а також окреслюючи області, що потребують покращення.

Здобуті результати не тільки підтверджують ефективність обраного підходу, але й відкривають перспективи для подальших досліджень у напрямі вдосконалення розподілених систем. Вони можуть слугувати фундаментом для розробки складніших додатків, де мікросервіси відіграватимуть вирішальну роль у забезпеченні масштабованості та стійкості.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ЗАСАДИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

У сучасному світі інформаційних технологій розробка програмного забезпечення вимагає не лише високої продуктивності, але й гнучкості, масштабованості та здатності швидко реагувати на зміни. Однією з архітектурних концепцій, яка відповідає цим вимогам, є мікросервісна архітектура. Вона дозволяє розробникам створювати незалежні сервіси, кожен з яких виконує чітко визначену функцію і сприяє ефективному управлінню складними системами.

У цьому розділі описано основні поняття та принципи цього виду архітектури та основні відмінності від монолітного підходу. Також проаналізовано переваги та недоліки обох підходів, що допоможе визначити, коли доречно використовувати той чи інший підхід до розробки. Особливу увагу буде приділено технологіям та інструментам, що використовуються для побудови мікросервісних систем, оскільки вибір відповідного технологічного стеку відіграє важливу роль у забезпеченні стабільної роботи програмного продукту.

1.1. Поняття, принципи та основні концепції мікросервісної архітектури

Мікросервісна архітектура є одним з найпопулярніших підходів до побудови сучасних веб-додатків. Вона гарантує високу масштабованість, гнучкість і надійність програмних рішень. Основна ідея полягає в поділі програмного забезпечення на незалежні сервіси, кожен з яких виконує певну бізнес-логіку і може бути розгорнутий і модифікований без впливу на інші компоненти системи.

Мікросервісна архітектура базується на кількох фундаментальних принципах, які відрізняють її від традиційного монолітного підходу.

Принципи мікросервісної архітектури наведено на рис.1.1.

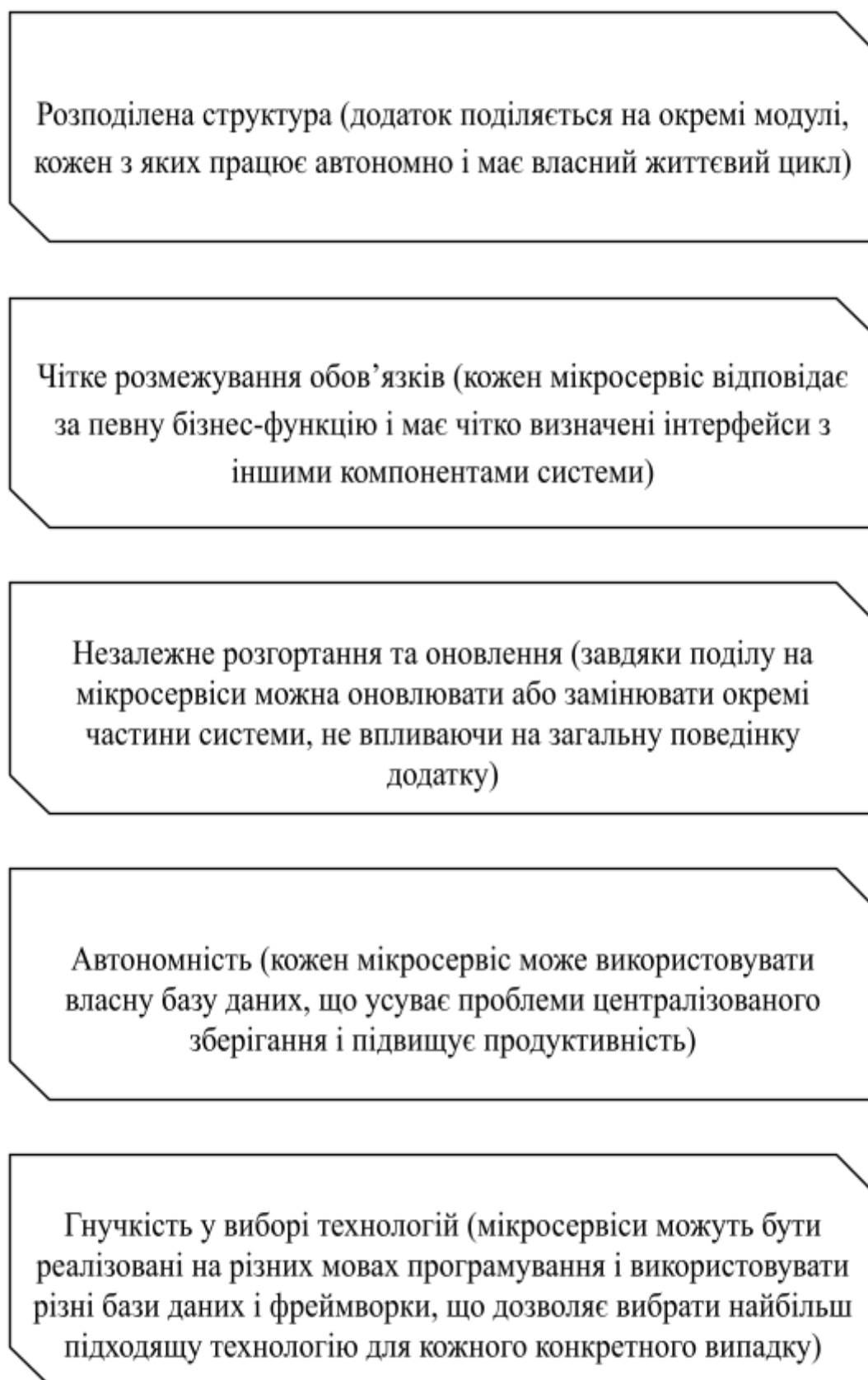


Рис.1.1. Принципи мікросервісної архітектури

Розглянемо детальніше кожен з принципів.

1. Розподілена структура.

В основі мікросервісної архітектури лежить принцип розподіленої структури, де кожен компонент системи існує окремо, виконує свої завдання і може працювати незалежно від решти додатку. Це означає, що замість монолітного додатку, в якому всі модулі тісно пов'язані між собою, додаток розбивається на кілька мікросервісів, кожен з яких має власну логіку, базу даних і окрему інфраструктуру.

Такий підхід має ряд переваг, в тому числі можливість гнучкого масштабування. Наприклад, якщо в системі є сервіс обробки платежів з підвищеним навантаженням у пікові періоди, кількість серверів для цього мікросервісу можна збільшити, не впливаючи на решту додатку. Традиційні монолітні архітектури вимагають масштабування всього додатку, що набагато дорожче і неефективно.

Крім того, розподілена структура зменшує ризик збою. Якщо один з мікросервісів вийде з ладу, система продовжить функціонувати, оскільки інші сервіси продовжать працювати. Наприклад, якщо зупиняється аналітичний сервіс, користувачі можуть продовжувати розміщувати замовлення, здійснювати платежі та отримувати продукти.

Ще одним важливим аспектом розподіленої архітектури є те, що різні мікросервіси можуть бути фізично розташовані в різних центрах обробки даних або на різних континентах. Це забезпечує високу доступність додатків і захист від локальних збоїв.

2. Чітке розмежування обов'язків

В мікросервісній архітектурі кожен мікросервіс виконує певну бізнес-функцію і має чітко визначені обов'язки. Це полегшує конфігурацію системи та спрощує підтримку і розвиток.

Для прикладу, розглянемо інтернет-магазин, в якому можна виділити наступні основні мікросервіси:

- Сервіс користувачів – зберігає дані клієнтів і забезпечує реєстрацію, автентифікацію та відновлення паролів.
- Сервіс замовлень – приймає та обробляє замовлення, змінює їх статус та взаємодіє з логістикою.
- Платіжні сервіси – забезпечують інтеграцію з банками та платіжними шлюзами.
- Сервіс товарного каталогу – надає інформацію про товари, ціни, знижки та наявність.
- Сервіси аналітики – збирають статистику про поведінку користувачів, популярні товари та ефективність маркетингових кампаній.

Завдяки такому поділу мікросервіси працюють незалежно один від одного. Це означає, що команді розробників, відповідальній за платіжний сервіс, не потрібно втручатися в код сервісу замовлень. У традиційних монолітних архітектурах зміни в платіжному модулі можуть спричинити помилки в інших частинах системи, що ускладнює обслуговування і тестування.

Чіткий розподіл завдань також підвищує безпеку. Наприклад, якщо в системі є мікросервіси, які обробляють персональні дані користувачів, їх можна захистити спеціальними заходами безпеки, обмеженнями доступу та зашифрованими з'єднаннями.

3. Незалежне розгортання та оновлення.

У традиційних архітектурах будь-які зміни в системі вимагають оновлення всього додатку, що може призвести до його зупинки. Натомість в мікросервісній архітектурі кожен сервіс може бути розгорнутий окремо, що дає значні переваги:

- Швидке внесення змін. Команди можуть оновити лише один мікросервіс без необхідності переробляти всю систему.
- Менший ризик помилок. Якщо оновлення містить помилку, вона впливає лише на певний мікросервіс, а не на всю систему.

- Плавні оновлення. Сервіси можна оновлювати поступово, а зміни можуть бути протестовані деякими користувачами до того, як вони будуть повністю впроваджені.

Наприклад, якщо компанія хоче змінити алгоритм розрахунку знижки, вона може оновити лише сервіс ціноутворення, не впливаючи на інші компоненти.

4. Автономність

Автономність означає, що кожен мікросервіс працює незалежно і використовує власну базу даних та ресурси. Це усуває проблему централізованих баз даних - одне з найбільших вузьких місць у традиційних системах.

Наприклад, у великому інтернет-магазині база даних замовлень містить мільйони записів, і запити до неї можуть уповільнити роботу всієї системи. В мікросервісній архітектурі служба замовлень може використовувати окрему базу даних, оптимізовану для власних потреб, тоді як служба аналітики може використовувати окрему базу даних, призначену для швидкого зчитування даних.

Автономність дозволяє мікросервісам реагувати на часткові збої. Наприклад, якщо база даних платежів тимчасово недоступна, інші сервіси продовжать працювати і обробляти платежі після відновлення бази даних.

5. Гнучкість у виборі технологій.

У монолітних системах всі компоненти повинні використовувати однакову технологію, що обмежує можливості розробників.

В мікросервісній архітектурі кожен сервіс може бути реалізований на окремій мові програмування та використовувати різні бази даних і фреймворки, що має наступні переваги:

- Оптимізація продуктивності. Наприклад, аналітичний сервіс для обробки великих обсягів даних може використовувати мову Go, яка є швидкою, в той час як користувацький сервіс може використовувати мову Python, що забезпечує швидку інтеграцію з машинним навчанням.

- Краща підтримка команд розробників. Якщо одна команда використовує Java, а інша - Node.js, вони можуть використовувати різні технології у спосіб, який найкраще відповідає їхньому досвіду та вимогам.
- Експерименти з новими технологіями. Коли з'являються нові перспективні фреймворки, їх можна реалізувати в одному мікросервісі без необхідності переписувати весь код.

Завдяки цьому компанії можуть швидко реагувати на зміни на ринку, впроваджувати нові технології та забезпечувати високу продуктивність системи.

Таким чином, дотримуючись основних принципів мікросервісної архітектури, можна створювати масштабовані, надійні та продуктивні веб-додатки, які легко адаптуються до нових вимог бізнесу.

Концепції мікросервісної архітектури наведено на рис. 1.2.

Розглянемо детальніше кожен з концепцій.

1. Декомпозиція на сервіси.

Декомпозиція на сервіси - один з основних принципів мікросервісної архітектури. Він передбачає декомпозицію складних додатків на окремі мікросервіси, кожен з яких відповідає за певну бізнес-функцію.

Основні переваги декомпозиції:

- Простота змін та оновлень – модифікація одного сервісу не зачіпає інші компоненти.
- Гнучке масштабування – можна збільшувати продуктивність лише найбільш завантажених сервісів.
- Розподіл відповідальності – кожна команда може працювати незалежно над своїми сервісами.

Розподіл мікросервісів здійснюється на основі таких підходів:

- Функціональний поділ – кожен мікросервіс реалізує певну бізнес-функцію, наприклад, управління користувачами, обробку платежів або аналіз даних.

- Поділ за доменами – сервіси поділяються відповідно до бізнес-логіки організації, що зменшує взаємозалежність між сервісами.
- Технічний поділ – сервіси можуть бути розділені відповідно до конкретних технічних характеристик.

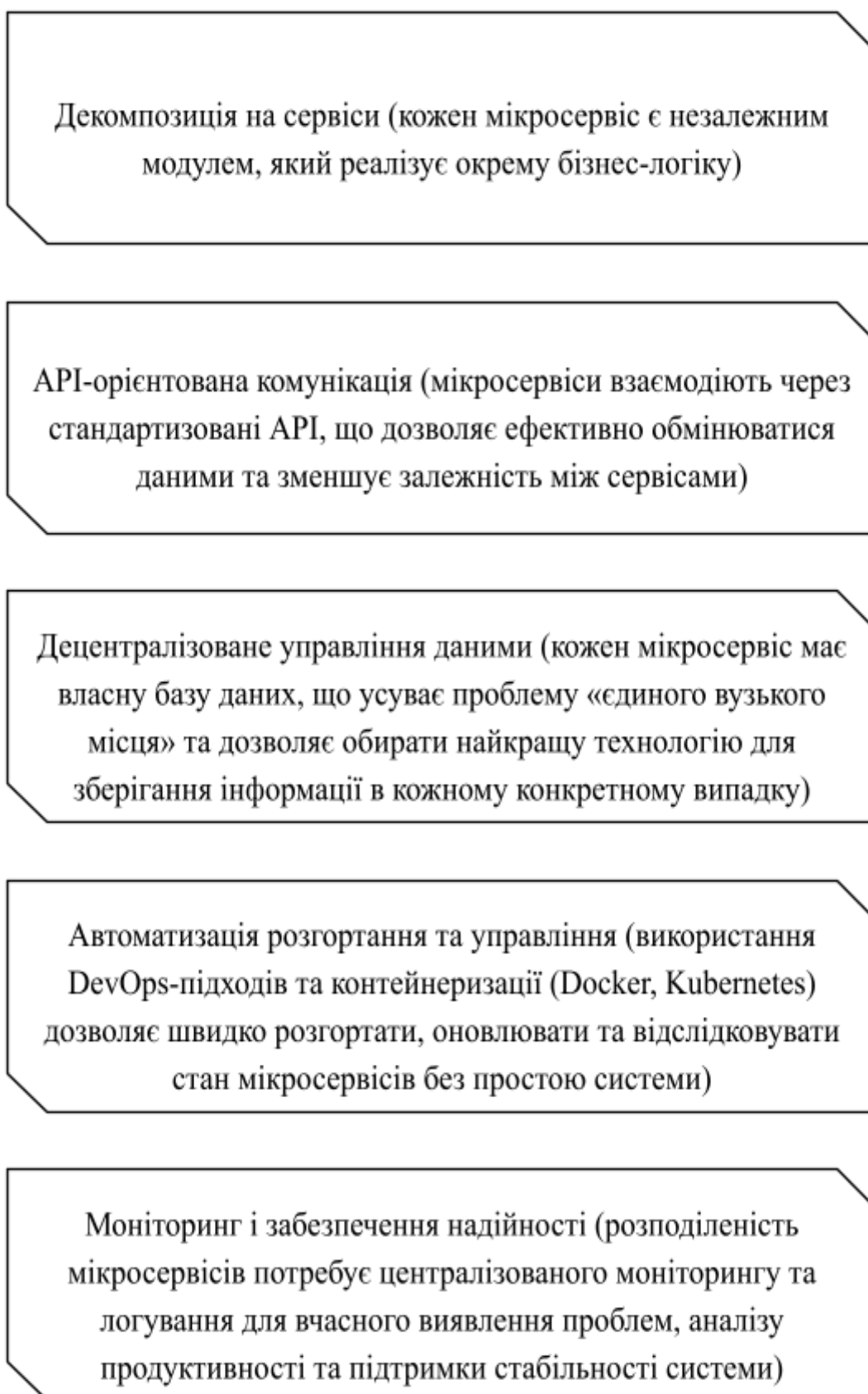


Рис.1.2. Концепції мікросервісної архітектури

2. API-орієнтована комунікація.

API-керований зв'язок означає, що мікросервіси взаємодіють один з одним, обмінюються даними і викликають необхідні функції за допомогою API.

Основні типи API:

- REST API – найпоширеніший підхід, що використовує HTTP-запити.
- GraphQL – дозволяє клієнтам отримувати тільки ті дані, які їм потрібні, і уникати зайвих запитів.
- gRPC – - високопродуктивний двійковий протокол, що використовується для взаємодії між сервісами у великих системах.
- Message Brokers (Kafka, RabbitMQ) – дозволяють організувати асинхронну обробку запитів і підвищити надійність системи.

Наприклад, коли користувач робить покупку, фронтенд-додаток надсилає запит до сервісу замовлення через API. Потім цей сервіс зв'язується з платіжним сервісом для підтвердження транзакції та надсилає запит до логістичної служби, щоб ініціювати доставку.

Така структура дозволяє легко змінювати або оновлювати окремі сервіси без необхідності переробляти всю систему.

3. Децентралізоване управління даними.

Децентралізоване управління даними означає, що кожна мікропослуга має власну базу даних для задоволення власних потреб. Це усуває конфлікти і покращує доступ до даних, а також продуктивність.

Основні переваги:

- Зменшення навантаження на центральну базу даних.
- Оптимізація роботи сервісів (можна використовувати різні типи баз даних: SQL, NoSQL, In-Memory).
- Підвищена відмовостійкість (вихід з ладу одного сервісу не впливає на інші).

Наприклад:

- Сервіс користувачів може використовувати реляційну базу MySQL для зберігання профілів.

- Сервіс товарного каталогу – NoSQL (MongoDB) для гнучкого керування категоріями та описами товарів.
- Сервіс аналітики – базу типу Hadoop або ClickHouse для обробки великих обсягів даних.

Таким чином, кожен мікросервіс використовує технологію, яка найкраще відповідає його функціональним вимогам.

4. Автоматизація розгортання та управління.

Автоматизація розгортання та управління означає, що розгортання та оновлення повинні бути автоматизовані, щоб підтримувати ефективну роботу великої кількості мікросервісів.

Ключові інструменти:

- Docker – контейнеризує сервіси та забезпечує ізольоване середовище виконання.
- Kubernetes – оркестрація контейнеризованих додатків, що дозволяє ефективно керувати розподілом ресурсів.
- CI/CD (Continuous Integration/Continuous Deployment) – метод автоматичного розгортання оновлень.

Наприклад, коли розробник додає нову функцію до сервісу, автоматизована система CI/CD запускає тести і розгортає оновлення на сервері без простою системи. Якщо оновлення містить помилку, Kubernetes автоматично відкочує попередню стабільну версію.

Це значно прискорює процес розробки та підвищує стабільність системи.

5. Моніторинг і забезпечення надійності.

Моніторинг та забезпечення надійності означає, що оскільки мікросервіси працюють незалежно, важливо мати централізовану систему моніторингу та логування, яка допоможе виявити помилки та проаналізувати продуктивність.

Основні інструменти:

- ELK Stack (Elasticsearch, Logstash, Kibana) – для збору, аналізу та візуалізації логів.

- Prometheus, Grafana – для моніторингу продуктивності та навантаження на сервіси.
- Distributed Tracing (Jaeger, Zipkin) – для аналізу запитів між мікросервісами.

Наприклад, якщо сервіс починає працювати повільно або повертає велику кількість помилок, система моніторингу може автоматично повідомити адміністратора або навіть активувати механізм самовідновлення.

Завдяки цьому мікросервісні системи залишаються стабільними, навіть якщо одна з їхніх частин тимчасово виходить з ладу.

Таким чином, мікросервісна архітектура є потужним підходом до створення масштабованих і надійних веб-додатків. Вона забезпечує гнучкість, спрощує оновлення та підтримку програмного забезпечення, а також дозволяє розгортати та розвивати окремі компоненти системи незалежно. Завдяки своїй розподіленій природі та використанню сучасних методів автоматизації цей підхід особливо підходить для складних інформаційних систем, що працюють у динамічному середовищі.

1.2. Порівняльний аналіз монолітної та мікросервісної архітектур: переваги, недоліки та сфери застосування

Розробка програмного забезпечення стала свідком появи різних підходів до його архітектури. Два типи, що найчастіше використовуються в сучасній розробці - це монолітні архітектури та мікросервісні архітектури. Обидва мають переваги та недоліки, які визначають їхню ефективність у різних випадках використання.

Монолітна архітектура передбачає створення програмного забезпечення як єдиного, цілісного додатка, де всі компоненти взаємопов'язані і функціонують у спільному середовищі. Це означає, що кожна частина програми, від інтерфейсу користувача до бази даних і бізнес-логіки, інтегрована в єдину кодову базу. Монолітний підхід використовувався протягом багатьох

років і досі є поширеним, особливо для малих або середніх проектів, які не потребують високого рівня масштабованості.

Основним принципом монолітної архітектури є єдність структури. Всі модулі програми пов'язані між собою і працюють в одному середовищі виконання. Це спрощує процес розробки, адже розробникам не потрібно налаштовувати складну інфраструктуру для взаємодії між окремими модулями і вони можуть працювати з єдиною кодовою базою.

Основні характеристики монолітної архітектури наведено на рис 1.3.

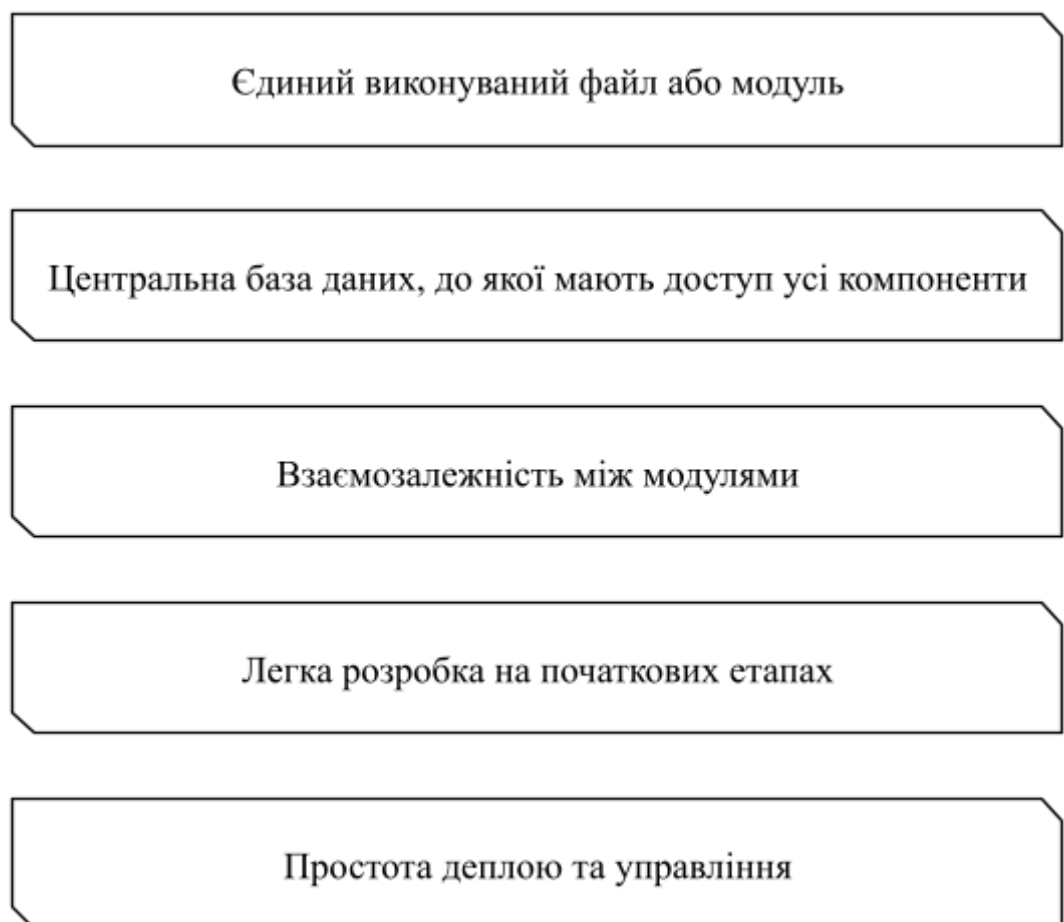


Рис.1.3. Основні характеристики монолітної архітектури

Розглянемо кожну з характеристик монолітної архітектури детальніше.

1. Єдиний виконуваний файл або модуль.

Монолітні додатки зазвичай компілюються або збираються в єдиному виконуваному середовищі. Всі компоненти програми, такі як обробка запитів,

бізнес-логіка, база даних і користувальницький інтерфейс, виконуються в рамках одного процесу.

Наприклад, у типовому веб-додатку, створеному на монолітній архітектурі, один серверний додаток містить код, який обробляє HTTP-запити, виконує бізнес-логіку, взаємодіє з базою даних і відображає інформацію через веб-інтерфейс.

2. Центральна база даних, до якої мають доступ усі компоненти.

У монолітному додатку всі компоненти взаємодіють з однією центральною базою даних. Це допомагає забезпечити цілісність та узгодженість даних, оскільки вся інформація зберігається в одному місці.

Однак така централізована структура також має певні недоліки:

- Зі збільшенням кількості користувачів зростає навантаження на базу даних.
- Схему бази даних неможливо легко змінити без ризику порушити роботу всієї системи.
- Залежність всієї системи від єдиного джерела даних робить проблематичною масштабованість.

3. Взаємозалежність між модулями.

Компоненти монолітного додатку тісно пов'язані між собою. Наприклад, модуль аутентифікації безпосередньо взаємодіє з модулем управління користувачами, а модуль управління замовленнями залежить від модуля обробки платежів.

Тому зміни в одній частині додатку впливають на інші модулі. Впровадження нових функцій або виправлення помилок може вимагати масштабного тестування всієї системи.

4. Легка розробка на початкових етапах.

Монолітні додатки часто легше розробляти на ранніх стадіях проекту, оскільки:

- Відсутня необхідність у складному налаштуванні комунікації між мікросервісами.

- Архітектуру легко зрозуміти, тому що всі компоненти знаходяться в єдиній кодовій базі.
- Налагодження та тестування простіше, оскільки вся система працює як єдине ціле.

Це особливо актуально для стартапів і невеликих команд, які хочуть швидко випустити першу версію свого продукту без значних витрат на інфраструктуру.

5. Простота деплою та управління.

Однією з головних переваг монолітної архітектури є простота розгортання. Оскільки додаток є єдиним цілим, його можна розгорнути за один крок без необхідності встановлювати кілька сервісів або контейнерів.

Наприклад, традиційний веб-додаток на PHP або Java можна розгорнути на одному сервері або віртуальній машині без складних налаштувань оркестрування сервісів. Це зменшує витрати на адміністрування системи та дозволяє швидко оновлювати її..

Однак простота розгортання також має зворотний бік:

- Якщо одна частина додатка виходить з ладу, це може вплинути на всю систему.
- Після оновлення повинні необхідно перевіряти весь додаток, навіть якщо зміни зачіпають лише один модуль.
- Чим більша система, тим складніша підтримка, оскільки кожне оновлення може порушити зв'язки між компонентами.

Монолітні архітектури продовжують залишатися популярними, особливо для малих і середніх проектів, завдяки своїй простоті і зручності. Вони підходять для стартапів і підприємств, які хочуть швидко виводити продукти на ринок без інвестицій у складну інфраструктуру.

Однак збільшення робочого навантаження, кількості користувачів і потреба в частих оновленнях можуть зробити монолітні архітектури неефективними. У таких випадках компанії часто переходять до моделі мікросервісів, яка пропонує кращу масштабованість та гнучкість.

Мікросервісна архітектура – це підхід до побудови програмного забезпечення, при якому вся система поділяється на ряд незалежних сервісів, кожен з яких виконує певну бізнес-функцію.

Основні характеристики мікросервісної архітектури було наведено і розглянуто в підрозділі 1.1. (див. рис 1.1.).

Мікросервісна архітектура забезпечує високу гнучкість, масштабованість і надійність, що робить її ідеальним вибором для складних і динамічних програмних рішень. Вона дозволяє командам працювати незалежно, впроваджувати нові технології та швидко адаптуватися до змін на ринку.

Однак, для ефективного використання мікросервісної моделі потрібен добре організований процес управління сервісами, розгортання, тестування та моніторингу, що може ускладнити початковий етап розробки.

Мікросервісна архітектура є ідеальним вибором для великих, високонавантажених систем, які потребують масштабованості та швидкої адаптації до змін, але для менших проєктів монолітний підхід може бути більш розумним рішенням.

В таблиці 1.1. наведено порівняльний аналіз монолітної та мікросервісної архітектур.

Таблиця 1.1.

Порівняльний аналіз монолітної та мікросервісної архітектур

Критерій	Монолітна архітектура	Мікросервісна архітектура
А	1	2
Структура	Єдина кодова база	Розподілена система з окремими сервісами
Розгортання	Єдине розгортання	Незалежне розгортання мікросервісів

Продовження таблиці 1.1.

Критерій	Монолітна архітектура	Мікросервісна архітектура
А	1	2

Масштабованість	Вертикальне масштабування	Горизонтальне масштабування
Гнучкість	Обмежена через залежності	Висока, можливість використовувати різні технології
Продуктивність	Менше мережевих витрат, але складніше масштабування	Додаткові витрати на мережеві запити, але краща адаптивність
Управління даними	Єдина база даних	Кожен сервіс має власне сховище
Стійкість до збоїв	Один компонент може спричинити падіння всього додатка	Відмова одного сервісу не впливає на всю систему

Переваги та недоліки монолітної архітектури наведено в таблиці 1.2.

Таблиця 1.2.

Переваги та недоліки монолітної архітектури

Переваги	Недоліки
1	2
● Простота розгортання - єдина структура зменшує складність розгортання нових версій. ● Менші витрати на інфраструктуру - немає потреби в складній організації контейнерів або сервісів. ● Менше навантаження на мережу - витрати на зв'язок зведені до мінімуму, оскільки всі компоненти працюють в рамках єдиного процесу.	● Складність підтримки - зі збільшенням кодової бази управління змінами стає складнішим. ● Обмеженість у масштабуванні - весь додаток масштабується разом, що робить його неефективним. ● Ризики при оновленнях - зміни в одній частині можуть вплинути на всю систему..

Переваги та недоліки мікросервісної архітектури наведено в таблиці 1.3.

Таблиця 1.3.

Переваги та недоліки монолітної архітектури

Переваги	Недоліки
1	2
● Гнучке масштабування - кожен сервіс масштабується окремо, що дозволяє більш ефективно використовувати ресурси. ● Незалежне оновлення - окремі компоненти можна оновлювати, не впливаючи на решту системи. ● Краща стійкість - збій одного сервісу не призводить до зупинки всього додатку.	● Складність управління - велика кількість сервісів ускладнює моніторинг і налагодження. ● Додаткові витрати на мережеві запити - взаємодія між сервісами збільшує затримки запитів. ● Ускладнення деплою: необхідні DevOps-підходи та автоматизація процесів.

Монолітна та мікросервісна архітектури мають свої переваги та недоліки, і вибір між ними залежить від особливостей проекту, вимог до масштабування, швидкості розгортання та доступності ресурсів.

Мікросервіси висувають нові завдання через складність операцій, узгодженості даних і безпеки. [13, с. 2] З іншого боку, монолітні архітектури простіше розгорнути і підтримувати.

Якщо проект невеликий і не вимагає високої продуктивності або складної логіки, монолітні архітектури є більш ефективним варіантом.

Якщо команда розподілена і працює на великій платформі з високим навантаженням, кращим рішенням буде мікросервісна архітектура.

Збалансований підхід до вибору архітектури дозволить створити ефективну, масштабовану та підтримувану систему, яка відповідатиме вимогам вашого бізнесу.

Сфери використання монолітної та мікросервісної архітектур наведено в таблиці 1.4.

Таблиця 1.4.

Сфери використання монолітної та мікросервісної архітектур

Монолітна архітектура	Мікросервісна архітектура
1	2
• Невеликі стартапи та MVP-проекти (наприклад, новий мобільний додаток для бронювання столиків у ресторанах) • Внутрішні корпоративні додатки (наприклад, корпоративна система обліку відпусток і робочого часу співробітників) • Додатки з низьким навантаженням та простою логікою (наприклад, лендинг-сайт компанії або невеликий інтернет-магазин із фіксованим каталогом товарів)	Високонавантажені вебсервіси (наприклад, відеострімінг Netflix) Великі компанії із розподіленими командами (наприклад, великий маркетплейс Amazon) Проекти, що потребують швидкої адаптації до змін ринку (наприклад, додаток для музики Spotify)

1.3. Ключові технології та інструменти для розробки та підтримки мікросервісів

Мікросервісний підхід до розбиття має на увазі розбиття на сервіси відповідно до потреб бізнесу. Такі сервіси включають в себе повний набір технологій, необхідних для цієї бізнес-потреби, в тому числі для користувацький інтерфейс, сховище даних і будь-які зовнішні взаємодії. [8, с. 1]

У мікросервісній архітектурі використовується широкий спектр технологій, кожна з яких відіграє певну роль у розробці, розгортанні, управлінні та моніторингу сервісів.

Основні категорії технологій, які використовуються для розробки і підтримки мікросервісів наведено на рис. 1.4.

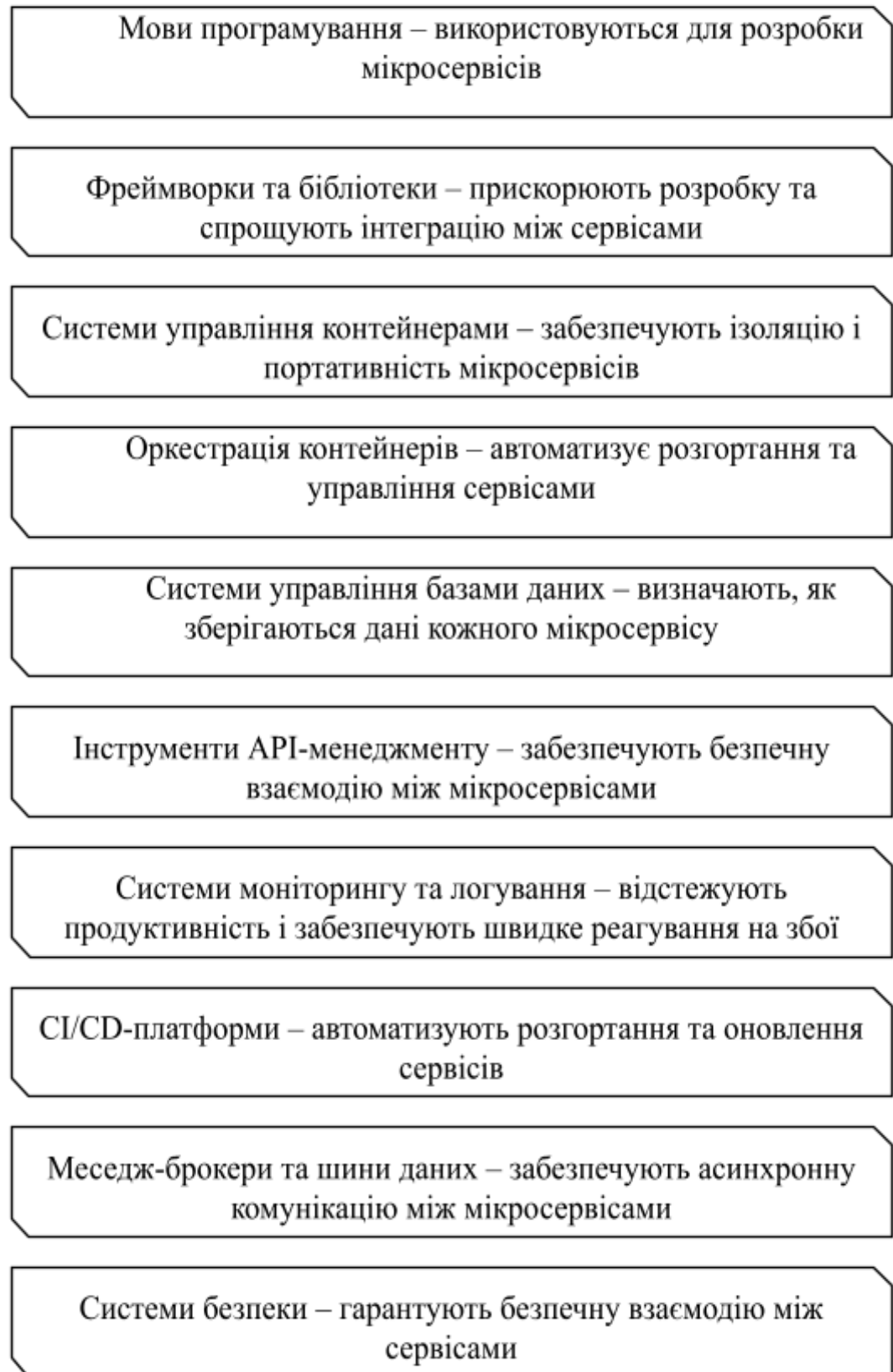


Рис.1.4. Основні категорії технологій, які використовуються для розробки і підтримки мікросервісів

Розглянемо детальніше кожну з категорій технологій, які використовуються для розробки і підтримки мікросервісів та особливості їх використання.

1. Мови програмування

Оскільки кожен мікросервіс є незалежним, розробники можуть використовувати різні мови програмування залежно від вимог конкретного сервісу.

Найпоширенішими мовами, які використовуються в мікросервісних архітектурах є:

- Java – дозволяє швидко розробляти та запускати сервіси завдяки Spring Boot та Quarkus.
- Python – використовується для створення мікросервісів, орієнтованих на аналітику даних та машинне навчання (FastAPI, Flask)
- Go – характеризується високою продуктивністю і використовується в середовищах Kubernetes.
- Node.js – для швидкої обробки запитів, особливо корисна в режимі реального часу
- C# (.NET Core) – для побудови продуктивних мікросервісів на платформах Microsoft

2. Фреймворки та бібліотеки для створення мікросервісів

Фреймворки допомагають розробникам швидко створювати, налаштовувати та розгортати мікросервіси.

Серед популярних фреймворків можна виділити:

- Spring Boot (Java) – спрощує створення автономних сервісів.
- Quarkus (Java) – оптимізований для роботи в контейнерах.
- Flask / FastAPI (Python) – для побудови легковагових RESTful API.
- Express.js (Node.js) – спрощує створення HTTP-сервісів.
- Micronaut (Java, Kotlin, Groovy) – зменшує споживання пам'яті.

3. Системи управління контейнерами

Контейнеризація дозволяє запускати додатки в ізольованих середовищах, що полегшує їх розгортання, масштабування та підтримку. Контейнери містять все необхідне для запуску програми (код, бібліотеки, залежності) і забезпечують однакову роботу незалежно від середовища.

Серед технологій для контейнеризації можна виділити:

- Docker - найпопулярніша технологія контейнеризації, яка створює ізольоване середовище для запуску мікросервісів.
- Podman - альтернатива Docker.

4. Оркестрація контейнерів

Оркестрування контейнерів - це управління, масштабування і балансування навантаження контейнерних додатків.

Серед технологій для оркестрування контейнерів можна виділити:

- Kubernetes - найпопулярніший інструмент для автоматизації розгортання, масштабування і управління контейнерними сервісами
- Helm - менеджер пакетів для Kubernetes, який розгортає складні додатки і спрощує управління конфігурацією.

5. Системи управління базами даних

В мікросервісній архітектурі кожен сервіс може мати власну базу даних, що дозволяє йому вибрати найкращу технологію для виконання конкретного завдання. Вибір СУБД залежить від таких вимог, як швидкість, масштабованість, узгодженість даних і тип оброблюваної інформації.

Повинна бути розроблена одна база даних на мікросервіс; вона повинна бути приватною лише для цього сервісу. До нього повинен мати доступ лише API мікросервісу. [1, с. 4]

Серед баз даних, які можуть використовуватись мікросервісами:

- Реляційні бази даних (SQL) організовують дані в таблицях зі структурованими зв'язками (PostgreSQL, MySQL, Microsoft SQL Server).

- Документно-орієнтовані бази даних та бази даних NoSQL, призначені для гнучкого зберігання великих обсягів даних, які не повинні мати жорсткої структури (MongoDB, Cassandra, Redis).
- Бази даних NewSQL - поєднують переваги традиційних реляційних баз даних з високою масштабованістю (CockroachDB, Google Spanner).
- Графові бази даних - оптимізовані для зберігання та запитів до сильно пов'язаних даних, таких як соціальні мережі, системи рекомендацій та складні бізнес-процеси (Neo4j).

6. Інструменти API-менеджменту

Інструменти API-менеджменту - це набір технологій та інструментів, які допомагають керувати API, контролювати безпеку, обмежувати трафік та збирати аналітику. Оскільки мікросервіси взаємодіють через API, важливо забезпечити їх продуктивність, безпеку та доступність

Серед інструментів для управління API можна виділити:

- Kong - швидкий та гнучкий API-шлюз, який підтримує балансування навантаження, кешування та безпеку API.
- Apigee (Google Cloud) - корпоративне рішення для управління API, яке забезпечує аналітику, безпеку та управління API.
- AWS API Gateway - сервіс для розгортання, моніторингу та безпечної взаємодії з API у хмарному середовищі AWS.
- Traefik - зворотний проксі-сервер для автоматичного маршрутизації запитів між мікросервісами.
- Postman - інструмент, який використовується для тестування, документування та емуляції запитів до API під час розробки.

7. Системи моніторингу та логування

В архітектурі мікросервісів важливо відстежувати стан сервісу, аналізувати його та швидко реагувати на помилки. Для цього використовуються системи збору метрик, централізованого логування та моніторингу запитів.

Серед таких систем можна виділити:

- Prometheus - система для збору та зберігання метрик, що підтримує гнучку мову запитів (PromQL);
- Grafana - візуалізація метрик з Prometheus та інших джерел (InfluxDB, Graphite) для платформ
- ELK Stack (Elasticsearch, Logstash, Kibana) – набір інструментів для збору, зберігання та візуалізації логів.
- Jaeger / Zipkin – інструмент для відстеження життєвого циклу запитів у мікросервісах, що допомагає виявити вузькі місця в продуктивності.
- Fluentd – гнучка система логування, яка працює з різними джерелами (Docker, Kubernetes, лог-файли) і може експортувати дані в ELK, Prometheus або інші системи

8. CI/CD-платформи

Автоматизовані процеси CI/CD (Continuous Integration / Continuous Deployment) прискорюють розробку, тестування та розгортання мікросервісів. Ці інструменти допомагають розробникам скоротити час виходу на ринок, зменшити кількість помилок та підвищити стабільність інфраструктури.

Серед CI/CD технологій можна виділити:

- Jenkins - потужний сервер автоматизації з підтримкою плагінів для різних етапів розгортання.
- GitHub Actions – інтеграція CI/CD безпосередньо в GitHub, дозволяє автоматизувати тести, збірку та деплой.
- GitLab CI/CD – вбудоване рішення GitLab, яке дозволяє керувати пайплайнами розгортання прямо з репозиторію.
- ArgoCD – інструмент GitOps для Kubernetes, що дозволяє автоматично оновлювати кластери відповідно до змін у репозиторії.
- Spinnaker – система для управління процесами розгортання, що підтримує багатоетапні пайплайни і інтеграцію з Kubernetes, AWS.

9. Меседж-брокери та шини даних

Меседж-брокери (message brokers) та шини даних (event buses) забезпечують асинхронний обмін повідомленнями між мікросервісами, зменшуючи затримки та підвищуючи продуктивність системи.

Серед популярних інструментів можна виділити:

- RabbitMQ - надійний брокер повідомлень, що використовує черги і підтвердження доставки.
- Apache Kafka - потокова платформа з високою пропускнуою здатністю, придатна для аналітики і обробки великих обсягів подій.
- NATS - легковаговий брокер повідомлень для високошвидкісного обміну між сервісами.
- Redis Pub/Sub - підходить для простих схем комунікації через публікацію та підписку.

10. Системи безпеки

Безпека в архітектурі мікросервісів включає в себе аутентифікацію, авторизацію, шифрування трафіку і контроль доступу до API.

Серед основних механізмів безпеки:

- OAuth 2.0 / OpenID Connect - стандарт автентифікації та авторизації користувачів у мікросервісах.
- JWT (JSON Web Token) - використовується для безпечної передачі інформації між сервісами.
- mTLS (Mutual TLS) - шифрує з'єднання та забезпечує двосторонню автентифікацію між сервісами.
- API-шлюзи (Kong, NGINX, Apigee) - контролюють доступ до мікросервісів та забезпечують маршрутизацію і безпеку.
- HashiCorp Vault - забезпечує безпечний спосіб зберігання та доступу до інформації. Конфіденційне зберігання та контроль доступу.

Ці інструменти допомагають запобігти вразливостям та забезпечити безпечну взаємодію між сервісами.

Таким чином, ключові технології для мікросервісів, такі як мови програмування, фреймворки, бази даних, контейнери і т.д., забезпечують

ефективну розробку і координацію автономних складових, спрощуючи створення API, пошук сервісів і маршрутизацію. Інструменти моніторингу та ведення журналів допомагають відстежувати роботу системи, а автоматизація підтримує її стабільність. Ці засоби є основою успішного впровадження та управління мікросервісами.

Висновки до розділу 1

Аналіз теоретичних засад засвідчив, що мікросервісна архітектура представляє собою сучасний метод конструювання програмних систем, заснований на розподілі функціоналу на окремі, самостійні складові. Такі сервіси розробляються та підтримуються незалежно, що дозволяє командам працювати паралельно, оперативно вносити зміни та адаптувати систему до нових вимог. Кожен сервіс виконує чітко окреслену задачу, взаємодіючи з іншими через стандартизовані інтерфейси, що забезпечує гнучкість та полегшує інтеграцію. Водночас, такий підхід потребує ретельного планування, адже розподіленість може ускладнити координацію та збільшити витрати на налаштування взаємодії між складовими.

Порівняння з монолітною архітектурою виявило вагомі відмінності. Монолітні системи простіші на початковому етапі реалізації та тестування, але їх громіздкість ускладнює масштабування та оновлення. Натомість, мікросервіси дають змогу розгортати та оновлювати окремі частини без зупинки всієї системи. Проте, ця перевага супроводжується недоліками, такими як більша складність керування та потреба у додаткових інструментах для моніторингу та оркестрації. Вибір між цими підходами залежить від конкретних потреб: моноліти краще підходять для невеликих проєктів зі стабільними вимогами, а мікросервіси – для складних систем, які потребують гнучкості та частих покращень.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБ-ДОДАТКУ НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Сучасні веб-додатки дедалі частіше вимагають адаптивності, можливості збільшення навантаження та вражаючої швидкодії, що робить мікросервісну архітектуру однією з найбільш перспективних парадигм для їх створення. У цьому розділі дипломного проєкту аналізується процес розробки відеоплатформи для перегляду контенту, яка спирається на концепцію розподілених сервісів. Розділ містить у собі важливі стадії розробки: від визначення основних потреб системи до безпосереднього втілення її складових. Спочатку, визначаються завдання та функціональні вимоги, які покладені в основу проєкту. Далі детально розглянуто архітектурне планування, включно зі структурою системи, моделюванням даних та правилами взаємодії між мікросервісами. Остання частина розділу присвячена практичному втіленню ключових сервісів, таких як каталогізація, потокове відтворення та маршрутизація відео. Цей підхід дає змогу не тільки створити робочу систему, але й підготувати її до майбутнього масштабування та вдосконалення.

2.1. Формулювання завдання та визначення основних вимог до веб-додатку

В рамках дипломного проєкту ведеться розробка відеоплатформи, побудованої на базі мікросервісної архітектури. Платформа призначена для надання користувачам можливості перегляду відеоматеріалів через веб-інтерфейс. Головною ціллю даного проєкту є створення стабільного, легко масштабованого та гнучкого сервісу для стрімінгу відео. Такий сервіс має ефективно функціонувати в умовах значних навантажень, гарантуючи безперебійний доступ до відеоматеріалів.

Сучасний стан ринку відеоконтенту характеризується динамічним зростанням, стимульованим збільшенням популярності онлайн-сервісів

потокowego відео, таких як YouTube, Netflix, Hulu та інші. Ця тенденція диктує потребу в розробці новітніх рішень, які здатні забезпечити високу якість передачі відео, мінімальну затримку відтворення та надійну роботу навіть під час піків активності. Впровадження мікросервісної архітектури надає можливість ефективного розподілу навантаження, адаптації до змінних умов експлуатації та спрощення процедури масштабування платформи.

Розглянемо базові вимоги до веб-додатку для перегляду відеоконтенту. Ці вимоги наведено на рис. 2.1.

Розглянемо кожну з вимог детальніше.

Масштабованість значить, що додаток має бути пристосованим до збільшення обсягів роботи як вертикально (збільшення ресурсів окремого сервісу), так і горизонтально (додавання екземплярів сервісів). Незалежність кожного сервісу дозволяє гнучко регулювати потужність певних частин системи відповідно до навантаження.

Безпека передбачає впровадження засобів аутентифікації, авторизації та шифрування даних для захисту інформації. Сучасні методи захисту та контролю доступу необхідні для уникнення несанкціонованого доступу до системи.

Взаємодія мікросервісів передбачає ефективну комунікацію між сервісами через API або обмін повідомленнями. Використання REST, gRPC або асинхронного обміну повідомленнями (наприклад, RabbitMQ, Kafka) для забезпечення надійної взаємодії.

Легкість розгортання значить, що варто автоматизувати процес розгортання з використанням Docker, Kubernetes або інших інструментів управління контейнерами. Це прискорює та спрощує запуск додатку в різних середовищах.

Моніторинг і логування передбачає впровадження системи моніторингу та ведення логів для відстеження стану системи, збору метрик продуктивності та швидкого виявлення проблем.

Зручність розробки та підтримки покликана забезпечити модульну структуру, яка дозволяє легко вносити зміни, оновлювати або додавати нові сервіси, не впливаючи на роботу інших частин системи.

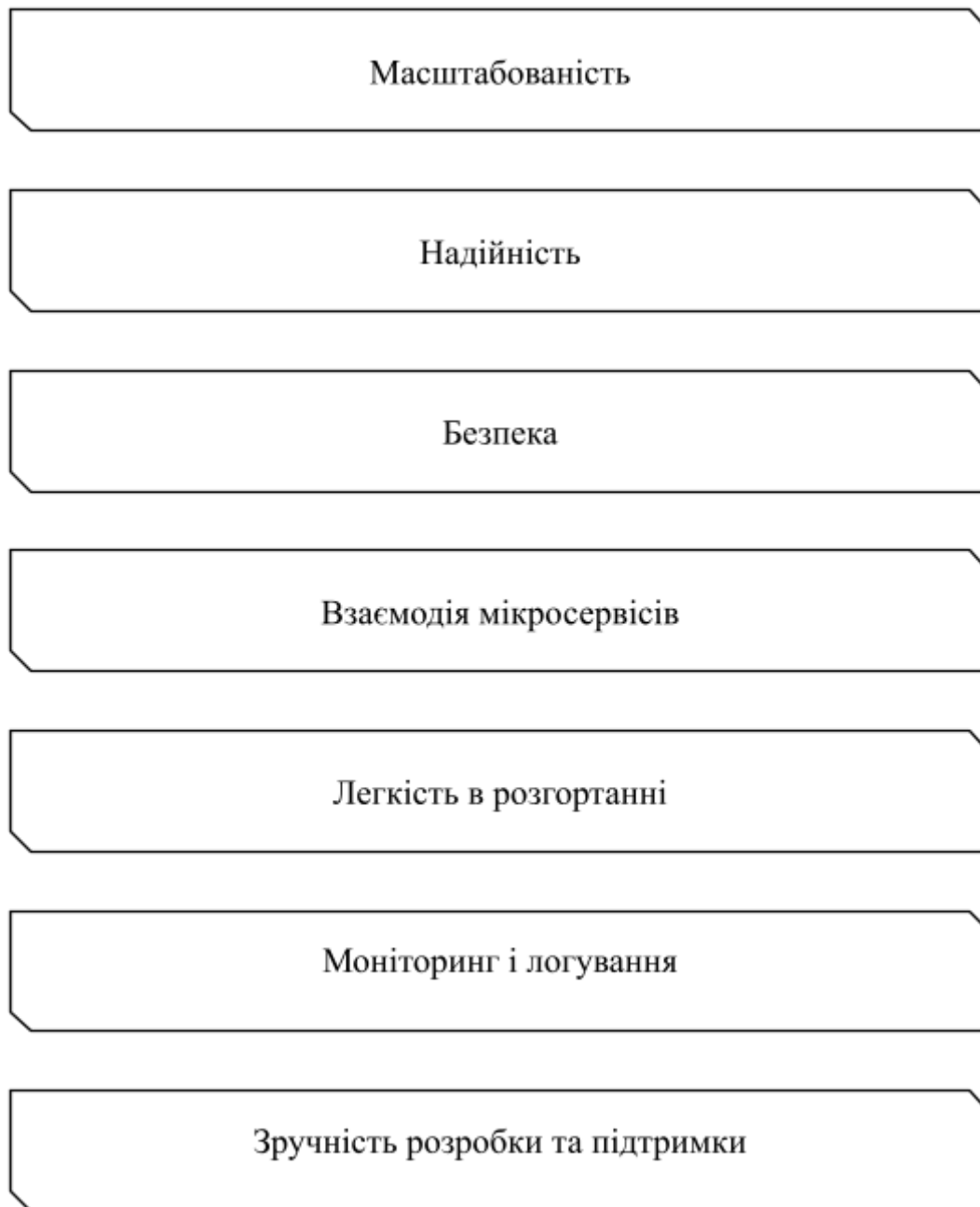


Рис. 2.1. Базові вимоги до веб-додатку для перегляду відеоконтенту

Розглянемо вимоги до системи управління відеоконтентом для веб-додатку для перегляду відеоконтенту.

Вимоги до системи управління відеоконтентом наведено на рис. 2.2.

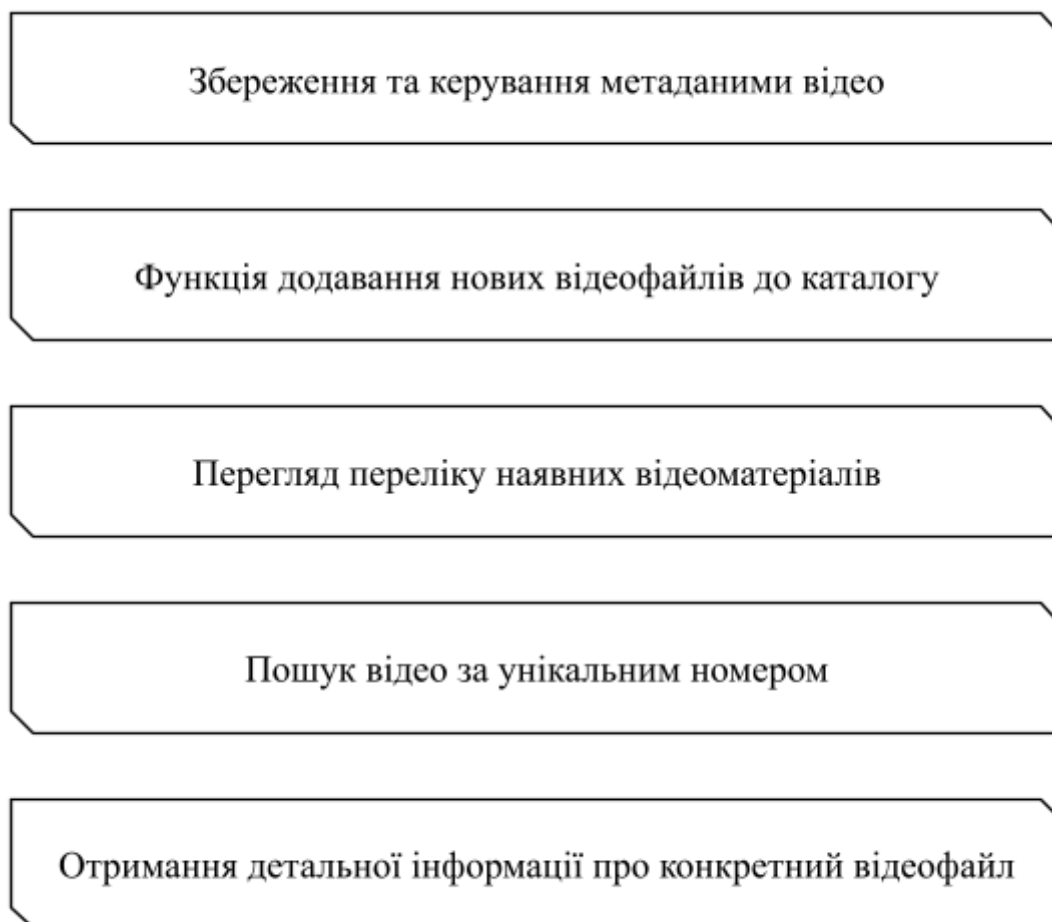


Рис. 2.2. Вимоги до системи управління відеоконтентом для веб-додатку для перегляду відеоконтенту

Розглянемо кожну з вимог детальніше.

Зберігання та керування метаданими відео (заголовок, опис, місцезнаходження файлу) означає забезпечення збереження всіх важливих відомостей про кожний відеозапис у базі даних. До метаданих належать такі параметри як назва відео (яка віддзеркалює суть та тематику контенту), опис (детальне текстове пояснення змісту, теми та специфіки відео), шлях до файлу (URL або фізичне розташування файлу на серверах). Метадані повинні мати структуру, яка дозволяє швидко й комфортно отримувати інформацію, оновлювати, видаляти та додавати нові деталі. Також передбачена підтримка інших атрибутів, таких як категорії, ключові фрази, позначки та дата завантаження.

Функція додавання нових відеоматеріалів до каталогу значить дозвіл адміністраторам або авторизованим користувачам додавати нові відеофайли в систему. Процес додавання включає завантаження відеофайлу, введення або автоматичну генерацію унікального ідентифікатора, заповнення метаданих (назва, опис, категорії, теги тощо). Система має передбачати перевірку завантажених файлів на сумісність з підтримуваними форматами (наприклад, MP4, AVI, MKV) та обмеженнями розміру. До того ж, система повинна мати можливість перевірки відео на безпеку (виявлення шкідливих файлів) та відповідність стандартам платформи (наприклад, вимогам спільноти).

Перегляд переліку наявних відеоматеріалів значить, що система зобов'язана надавати користувачам можливість переглядати каталог доступних відео у зрозумілому та організованому вигляді. Перелік відео має містити базову інформацію про кожен файл, як от назву, стислий опис, категорію та дату додавання. Важливою є можливість сортування та фільтрації контенту за різними параметрами, зокрема, за популярністю, рейтингом, датою завантаження та категоріями. Інтерфейс має бути інтуїтивно зрозумілим та надавати зручний доступ до функцій перегляду та пошуку.

Пошук відео за унікальним номером передбачає швидкий пошук конкретного відео за його унікальним ідентифікатором. Ідентифікатор може бути поданий у вигляді чисельного коду, алфавітно-цифрового рядка чи URL-посилання. Система повинна забезпечувати миттєвий доступ до потрібного відео через пошуковий рядок або перехід за прямим посиланням. Акцентується увага на точності та надійності пошуку, а також на можливості негайного перегляду відповідного контенту.

Отримання детальної інформації про конкретний відеофайл забезпечує можливість перегляду повного переліку метаданих вибраного відео. Система має демонструвати таку інформацію як повний опис, тривалість відео, категорії, ключові слова, рейтинг, кількість переглядів та дата завантаження. Додатково передбачається можливість перегляду коментарів до відео (якщо така функція підтримується) та інформації про завантажувача (автора). Користувачам також

може бути надана можливість оцінювати відео та додавати його до особистого списку перегляду або обраного. Система зобов'язана надавати зручний доступ до всіх цих функцій за допомогою простого та інтуїтивно зрозумілого інтерфейсу.

Розглянемо вимоги до потокової трансляції відео для веб-додатку для перегляду відеоконтенту.

Вимоги до потокової трансляції відео наведено на рис. 2.3.

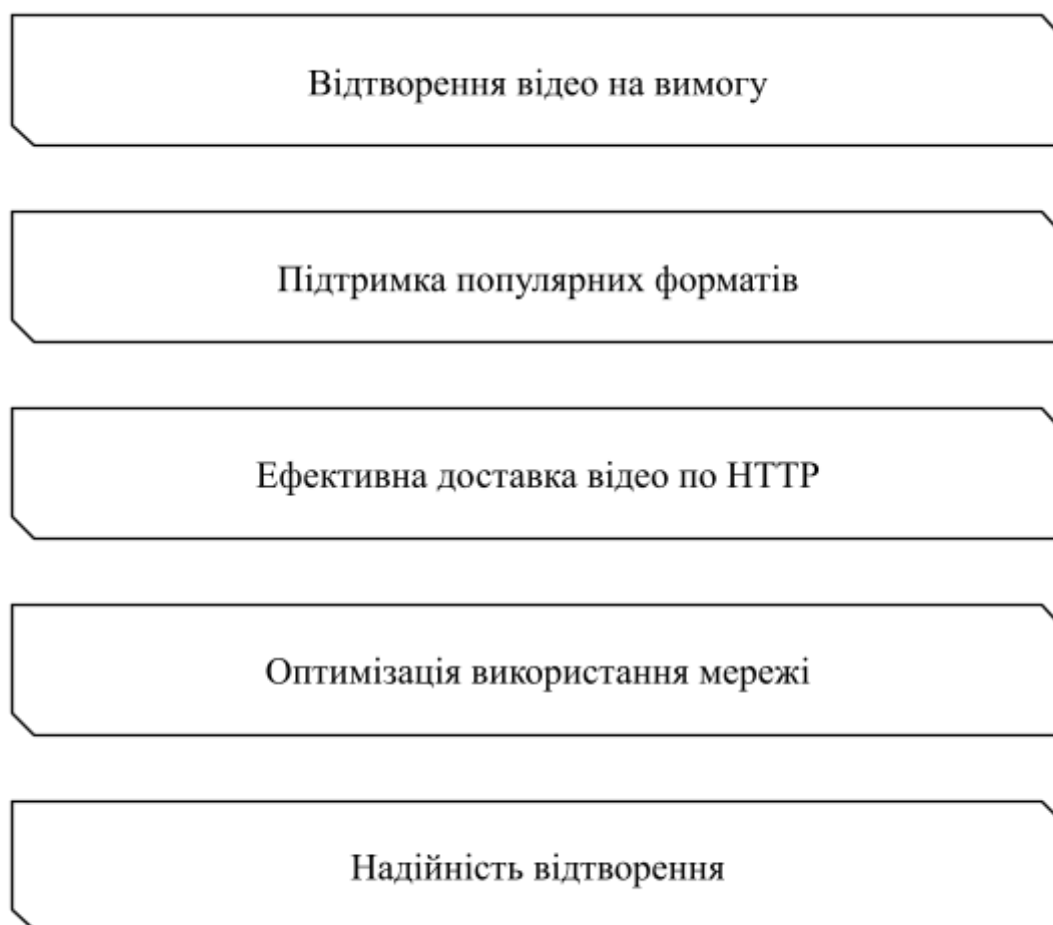


Рис. 2.3. Вимоги до потокової трансляції відео для веб-додатку для перегляду відеоконтенту

Розглянемо кожну з вимог детальніше.

Відтворення відео на вимогу значить, що платформа має гарантувати можливість перегляду відео за ініціативою користувача в режимі реального часу. Тобто, як тільки користувач обере відео, відеоплеєр запускає процес

поточної передачі, що забезпечує безперервне відтворення без необхідності завантаження всього файлу. Цей функціонал є ключовим для швидкого доступу до вмісту.

Підтримка популярних відеоформатів означає, що відеоплатформа повинна підтримувати відтворення відеофайлів у широко використовуваних форматах, таких як MP4, адже цей формат є популярним через високу якість та ефективне стиснення. Також можливе розширення підтримки інших форматів, що надасть більше гнучкості користувачам при завантаженні контенту.

Ефективна доставка відео по HTTP значить, що доставка відео відбувається через протокол HTTP із використанням технологій, як HLS (HTTP Live Streaming) чи MPEG-DASH (Dynamic Adaptive Streaming over HTTP). Це дозволяє адаптувати потік під доступну швидкість інтернет-з'єднання користувача і забезпечує безперервне відтворення навіть за нестабільного інтернет-з'єднання.

Оптимізація використання мережі передбачає механізми буферизації, кешування та стиснення відео для зменшення навантаження на сервери та мінімізації часу очікування. Застосування технологій CDN (Content Delivery Network) може допомогти забезпечити якісну доставку відео незалежно від місцезнаходження користувача.

Надійність відтворення значить, що критично важливо забезпечити безперебійну роботу відеоплеєра, незалежно від швидкості інтернет-з'єднання або навантаження на сервер. Механізми автоматичного відновлення відтворення після переривань та адаптивна зміна якості відео допомагають уникнути затримок чи зупинок під час перегляду.

Розглянемо вимоги до користувацького інтерфейсу для веб-додатку для перегляду відеоконтенту.

Вимоги до користувацького інтерфейсу наведено на рис. 2.4.

Розглянемо кожну з вимог детальніше.

Інтуїтивний веб-інтерфейс значить, що веб-сторінка має надавати інтуїтивно зрозумілий інтерфейс, котрий забезпечить користувачам легку

навігацію та доступ до всіх можливостей. Кнопки, меню та інші інтерактивні елементи повинні бути добре помітні та легкі у використанні. Ключові функції, як-от пошук відео, перегляд каталогу й відтворення, повинні бути розташовані на головній сторінці або бути просто досяжними через зрозумілі навігаційні елементи. Необхідно подбати про логічну структуру сторінок і розташування елементів, аби користувачі могли миттєво знаходити потрібне без потреби звертатися до інструкцій чи служби підтримки.

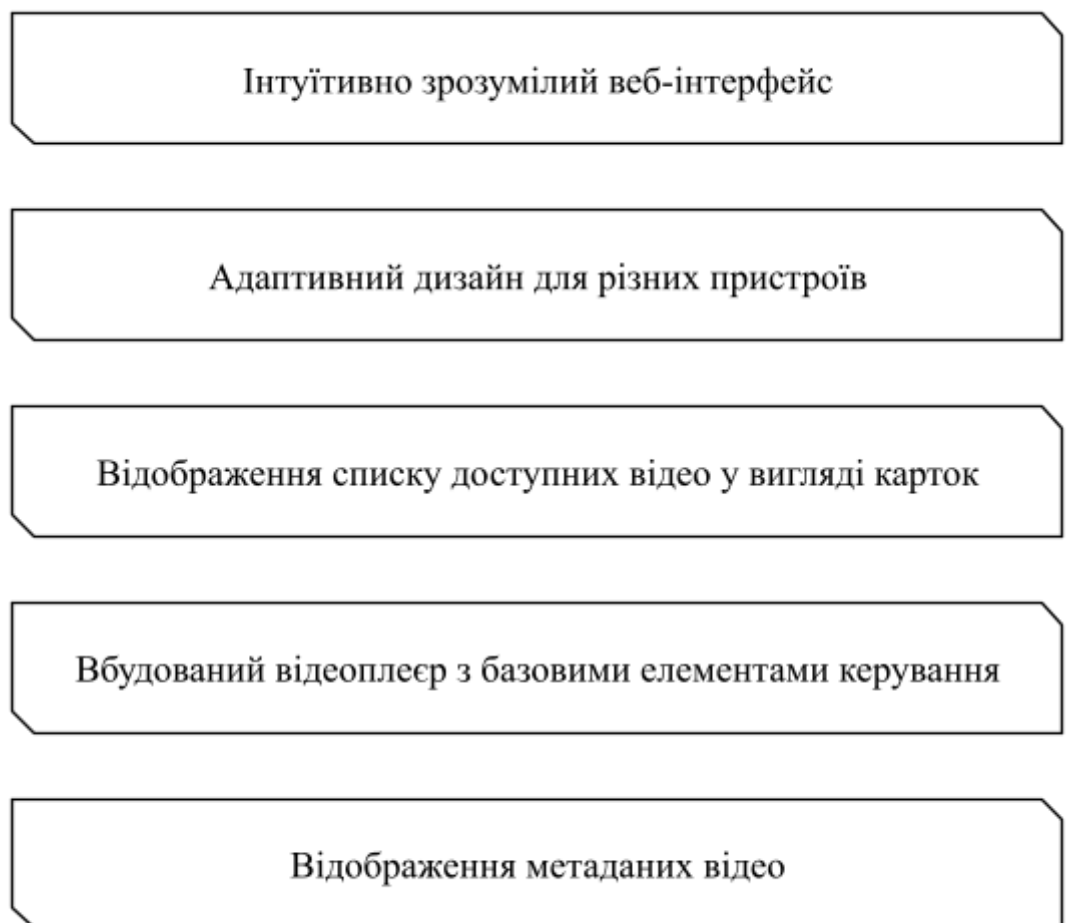


Рис. 2.4. Вимоги до користувацького інтерфейсу для веб-додатку для перегляду відеоконтенту

Адаптивний дизайн для різних пристроїв значить, що веб-додаток мусить правильно відображатися на різноманітних пристроях: комп'ютерах, планшетах, телефонах. Адаптивний дизайн – це автоматичне масштабування та перерозподіл елементів залежно від розміру екрана. Це означає, що меню може

змінюватися з горизонтального на вертикальне, відеоплеєр повинен адаптуватися до розміру екрана, а елементи керування мають залишатися доступними на малих екранах. Адаптивний дизайн також передбачає оптимізацію продуктивності, що включає швидке завантаження і плавне функціонування інтерфейсу на всіх типах пристроїв.

Відображення переліку доступних відео у вигляді карток для зручного перегляду контенту. Кожна картка повинна містити попередній перегляд (мініатюру), назву відео та стислий опис. Картки мусять мати чітку структуру й бути розміщені у сітці або списку, котрий можна прокручувати. Функції сортування та фільтрації (наприклад, за популярністю, новизною, категоріями) допоможуть користувачам швидко знаходити необхідні відео. Також варто передбачити перехід до детальної інформації про відео після натискання на відповідну картку.

Вбудований відеоплеєр із базовими елементами керування значить, що платформа повинна мати вбудований відеоплеєр, котрий дасть змогу користувачам дивитися відео без потреби завантажувати сторонні програми. Відеоплеєр має надавати стандартні елементи керування: відтворення, пауза, перемотування, регулювання гучності, а також можливість переходу в повноекранний режим. Інтерфейс відеоплеєра повинен бути зрозумілим та простим у користуванні. Варто передбачити відображення субтитрів, якщо вони є, і налаштування якості відтворення залежно від швидкості інтернет-з'єднання користувача.

Відображення метаданих відео (назва, опис) значить, що, переглядаючи відео або картки у каталозі, користувач повинен бачити метадані, що включають назву, опис, тривалість, дату завантаження та іншу інформацію, яка допомагає зрозуміти зміст відео перед переглядом. Метадані повинні бути чітко представлені й доступні для перегляду з можливістю детальнішого перегляду на вимогу. Також слід реалізувати систему тегів або категорій, що дасть змогу структурувати контент і спростить навігацію платформою.

Розглянемо системні вимоги до веб-додатку для перегляду відеоконтенту.

Системні вимоги наведено на рис. 2.5.

Розглянемо кожну з вимог детальніше.

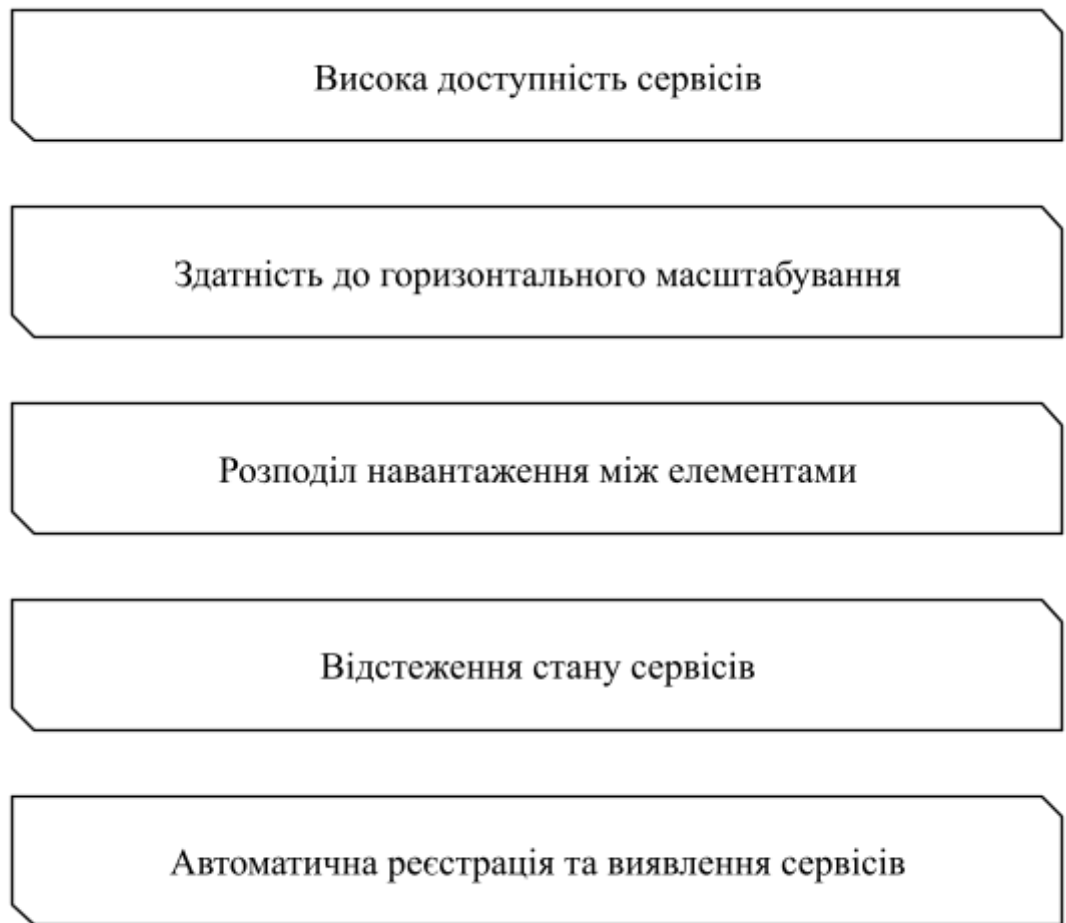


Рис. 2.5. Системні вимоги до веб-додатку для перегляду відеоконтенту

Висока доступність сервісів передбачає стабільну роботу відеоплатформи з мінімальними перервами. Досягається це дублюванням серверів, використанням кластеризації та резервних копій. Обов'язково потрібно передбачити автоматичне перемикання на резервні компоненти у випадку несправностей, що дає можливість підтримувати безперебійний доступ користувачів до відеоконтенту незалежно від технічних проблем.

Здатність до горизонтального масштабування надає змогу збільшувати інфраструктуру шляхом додавання нових серверів чи контейнерів замість збільшення потужності наявних. Це підвищує пропускну здатність платформи

та забезпечує обробку великої кількості запитів від користувачів. Важлива реалізація автоматичного масштабування у відповідь на зміну навантаження.

Розподіл навантаження між елементами значить, що система зобов'язана включати балансування навантаження між різними компонентами, такими як сервери відеопотоків, бази даних та веб-сервери. Це сприяє рівномірному розподілу запитів користувачів та запобігає перевантаженню окремих серверів. Балансувальники навантаження повинні підтримувати як апаратні, так і програмні рішення.

Відстеження стану сервісів дозволяє відслідковувати працездатність всіх компонентів відеоплатформи в реальному часі. Він включає в себе відстеження затримок, пропускну здатності мережі, навантаження на сервери та наявності помилок. Важливим аспектом є створення системи сповіщення, яка попереджає про можливі проблеми задовго до їх впливу на користувачів.

Автоматична реєстрація та виявлення сервісів значить, що система має автоматично виявляти нові сервери чи сервіси, які додаються до інфраструктури, та інтегрувати їх у загальну архітектуру. Це гарантує швидке розширення системи без необхідності ручного налаштування. Крім того, автоматичне виявлення сервісів спрощує процес масштабування та збільшує надійність платформи.

Розглянемо нефункціональні вимоги до веб-додатку для перегляду відеоконтенту.

Системні вимоги наведено на рис. 2.6.

Розглянемо кожну з вимог детальніше.

Відповідь API не довше 200 мс значить, що відеосервіс мусить видавати швидкі відповіді на всі API-запити. Це необхідно для забезпечення безперебійної та комфортної роботи користувачів. Для більшості запитів, таких як отримання даних про відео, пошук певного контенту або управління відтворенням, час відгуку не повинен перевищувати 200 мс. Досягти цього можна завдяки оптимізації запитів, кешуванню інформації та використанню продуктивних баз даних.

Підтримка одночасного перегляду відео великою кількістю користувачів значить, що система має гарантувати обробку одночасних запитів на перегляд відео від значної кількості користувачів без погіршення якості послуг. Це вимагає застосування технологій горизонтального масштабування, ефективного розподілу навантаження та оптимізованої передачі відеопотоків.

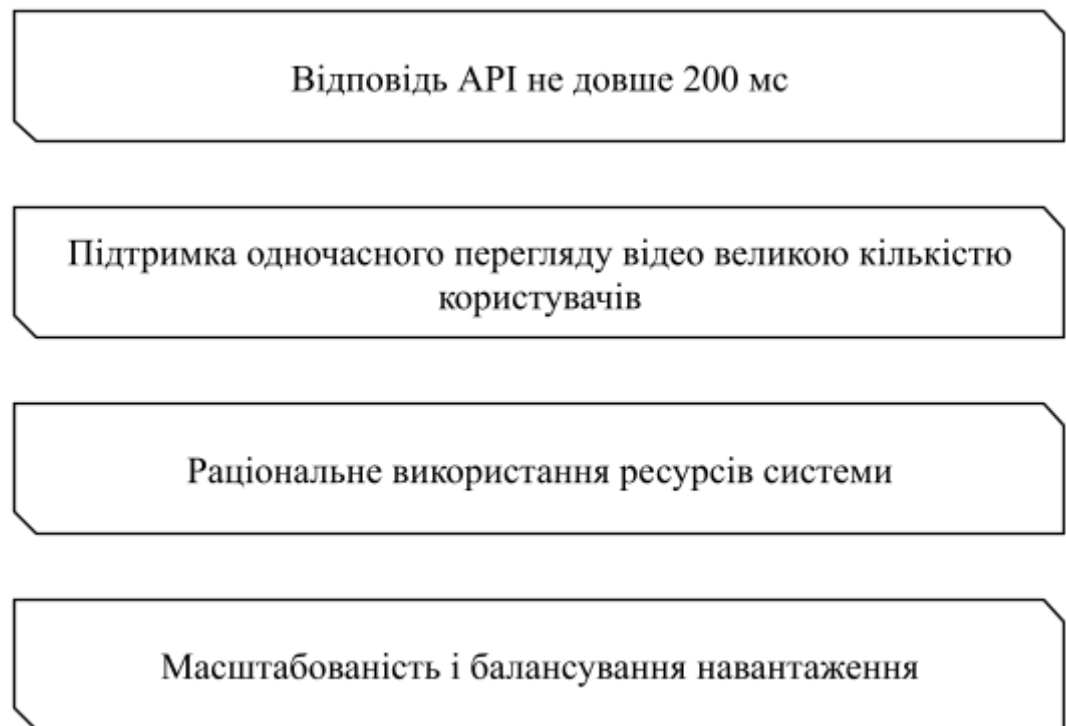


Рис. 2.6. Нефункціональні вимоги до веб-додатку для перегляду відеоконтенту

Раціональне використання ресурсів системи значить, що відеоплатформа повинна максимально ефективно використовувати доступні обчислювальні ресурси, зокрема процесор, пам'ять та мережеву пропускну здатність. Оптимізація роботи серверів, використання контейнеризації та мікросервісної архітектури сприятимуть мінімізації витрат на інфраструктуру при збереженні високої продуктивності.

Масштабованість і балансування навантаження означає, що система мусить забезпечувати можливість гнучкого масштабування для обробки зростаючої кількості запитів від користувачів. Використання балансувальників

навантаження та розподілених систем зберігання даних дозволить гарантувати безперебійний доступ до відеоконтенту та стабільну роботу платформи незалежно від рівня навантаження.

Розглянемо вимоги до масштабованості веб-додатку для перегляду відеоконтенту.

Вимоги до масштабованості наведено на рис. 2.7.

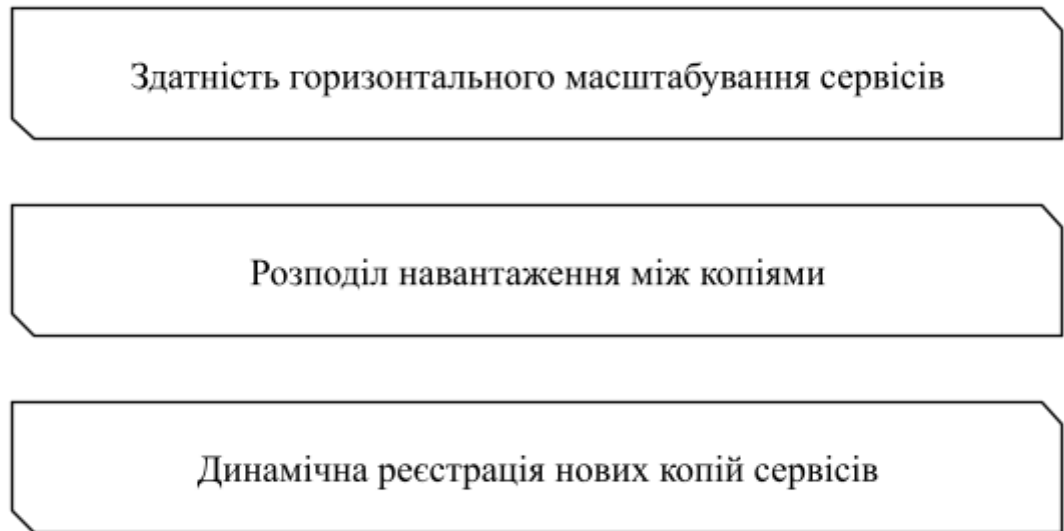


Рис. 2.7. Вимоги до масштабованості веб-додатку для перегляду відеоконтенту

Розглянемо кожну з вимог детальніше.

Здатність горизонтального масштабування сервісів описує можливість вводити в дію нові копії сервісів або серверів безпосередньо в інфраструктуру відеоплатформи, щоб забезпечити стабільну роботу, коли навантаження зростає. Це особливо важливо для платформ, які обслуговують значну кількість користувачів одночасно, особливо у години пік. Система повинна мати здатність додавати нові сервери без зупинки роботи або негативного впливу на доступність сервісів. Автоматичне масштабування має відбуватися на основі моніторингу використання ресурсів, наприклад, кількості одночасних запитів або рівню завантаження CPU та пам'яті.

Розподіл навантаження між копіями визначає рівномірний розподіл запитів між різними екземплярами сервісів або серверів для уникнення перевантаження окремих частин. Це здійснюється за допомогою балансувальників навантаження (наприклад, Nginx, HAProxy або спеціалізованих хмарних рішень). Система повинна мати можливість оперативно перенаправляти трафік на робочі копії серверів для гарантування швидкої відповіді на запити користувачів. Якщо окремі копії виходять з ладу, система має автоматично перенаправляти запити на наявні ресурси, зберігаючи при цьому безперервність роботи платформи.

Динамічна реєстрація нових копій сервісів полягає в автоматичному виявленні та додаванні нових екземплярів сервісів в загальну інфраструктуру після їх запуску. Це дозволяє гнучко масштабувати систему, не потребуючи ручного налаштування. Реєстрація сервісів повинна проводитись через централізований сервіс, який стежить за активними екземплярами та автоматично оновлює інформацію про доступні ресурси. Окрім цього, система повинна вміти вилучати з реєстрації ті копії, які вже неактивні, що допомагає зберегти точний стан інфраструктури та оптимально використовувати ресурси.

Розглянемо технічні вимоги до веб-додатку для перегляду відеоконтенту.

Технічні вимоги наведено на рис. 2.8.

Розглянемо кожну з вимог детальніше.

Розподіл функціональності між незалежними сервісами передбачає наступне:

- Уся система повинна бути поділена на незалежні сервіси, кожен з яких відповідає за конкретний функціональний блок
- Розподіл функціональності полегшує підтримку системи, забезпечує гнучкість при оновленнях та дозволяє масштабувати окремі сервіси без впливу на інші складові системи.

Використання Spring Cloud Netflix Eureka для реєстрації сервісів передбачає наступне:

- Використовується для динамічної реєстрації та виявлення сервісів

- Кожен мікросервіс автоматично реєструється в Eureka Server, що дає змогу іншим компонентам системи оперативно знаходити доступні сервіси.

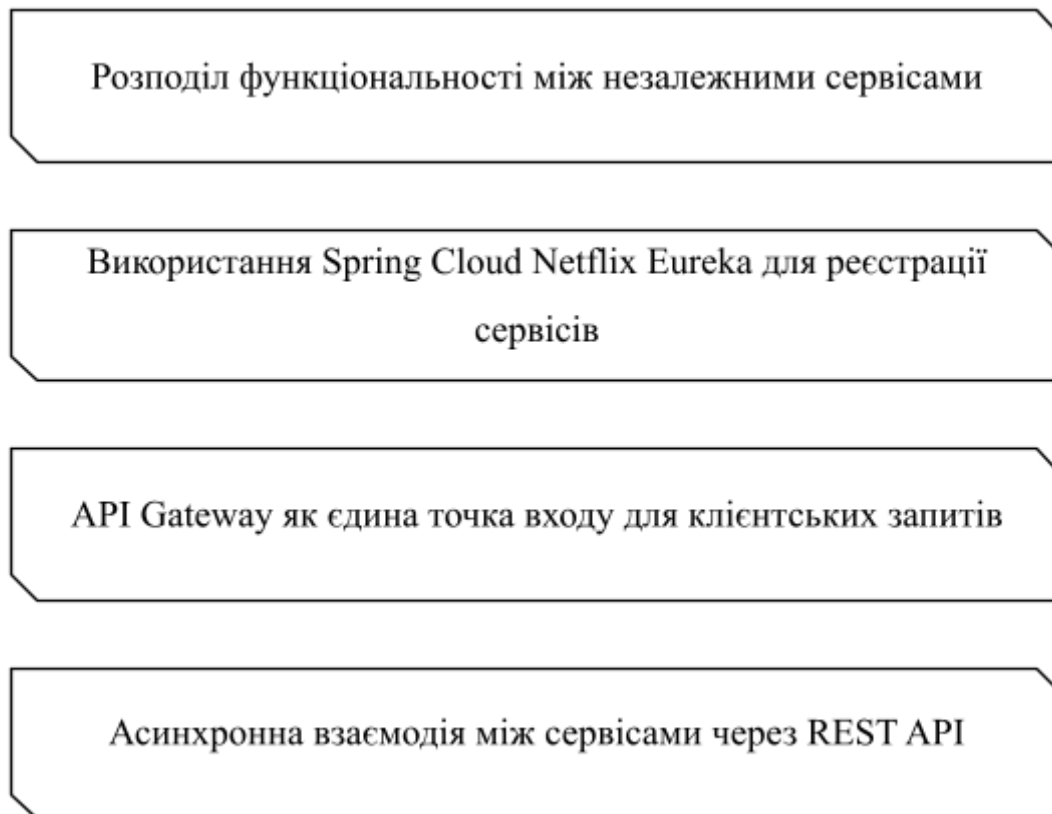


Рис. 2.8. Технічні вимоги до веб-додатку для перегляду відеоконтенту

API Gateway як єдина точка входу для клієнтських запитів передбачає наступне:

- Всі запити від клієнтів скеровуються через API Gateway, котрий служить єдиною точкою входу до системи
- API Gateway забезпечує маршрутизацію запитів до відповідних сервісів, орієнтуючись на URL-шляхи чи інші параметри запиту
- API Gateway повинен мати конфігурацію, що дозволяє йому взаємодіяти з Eureka Server для динамічного визначення доступних сервісів

Асинхронна взаємодія між сервісами через REST API передбачає наступне:

- Сервіси взаємодіють між собою через HTTP-запити через REST API
- Асинхронна взаємодія реалізується за допомогою черг повідомлень або подій для гарантування негайної відповіді на запити користувачів
- Для зменшення затримок і забезпечення високої доступності застосовуються неблокуючі виклики та асинхронні механізми обробки даних
- Критично важливо забезпечити надійність та безпеку зв'язку між сервісами шляхом використання SSL та механізмів авторизації

2.2. Архітектурне проєктування: структура системи, моделювання даних і взаємодія мікросервісів

Створення веб-додатку з використанням мікросервісної архітектури потребує ретельного планування системи, адже від цього залежить її гнучкість до змін, здатність витримувати великі навантаження та легкість подальшого вдосконалення. У цьому випадку розроблено платформу для перегляду відео, яка дозволяє користувачам переглядати каталог доступних матеріалів та відтворювати їх у режимі реального часу. Для втілення цього задуму використано мікросервісний підхід, який передбачає чотири ключові компоненти:

- Service Registry (Eureka Server)
- API Gateway
- Video Catalog Service
- Video Streaming Service

Архітектурна схема веб-додатку для перегляду відеоконтенту наведена на рис. 2.9.

Така структура забезпечує чіткий розподіл відповідальності між сервісами, полегшує їх взаємодію та створює базу для масштабування в майбутньому. У цьому розділі описуються принципи організації системи, структура даних та способи взаємодії між компонентами, які лягли в основу подальшої розробки.

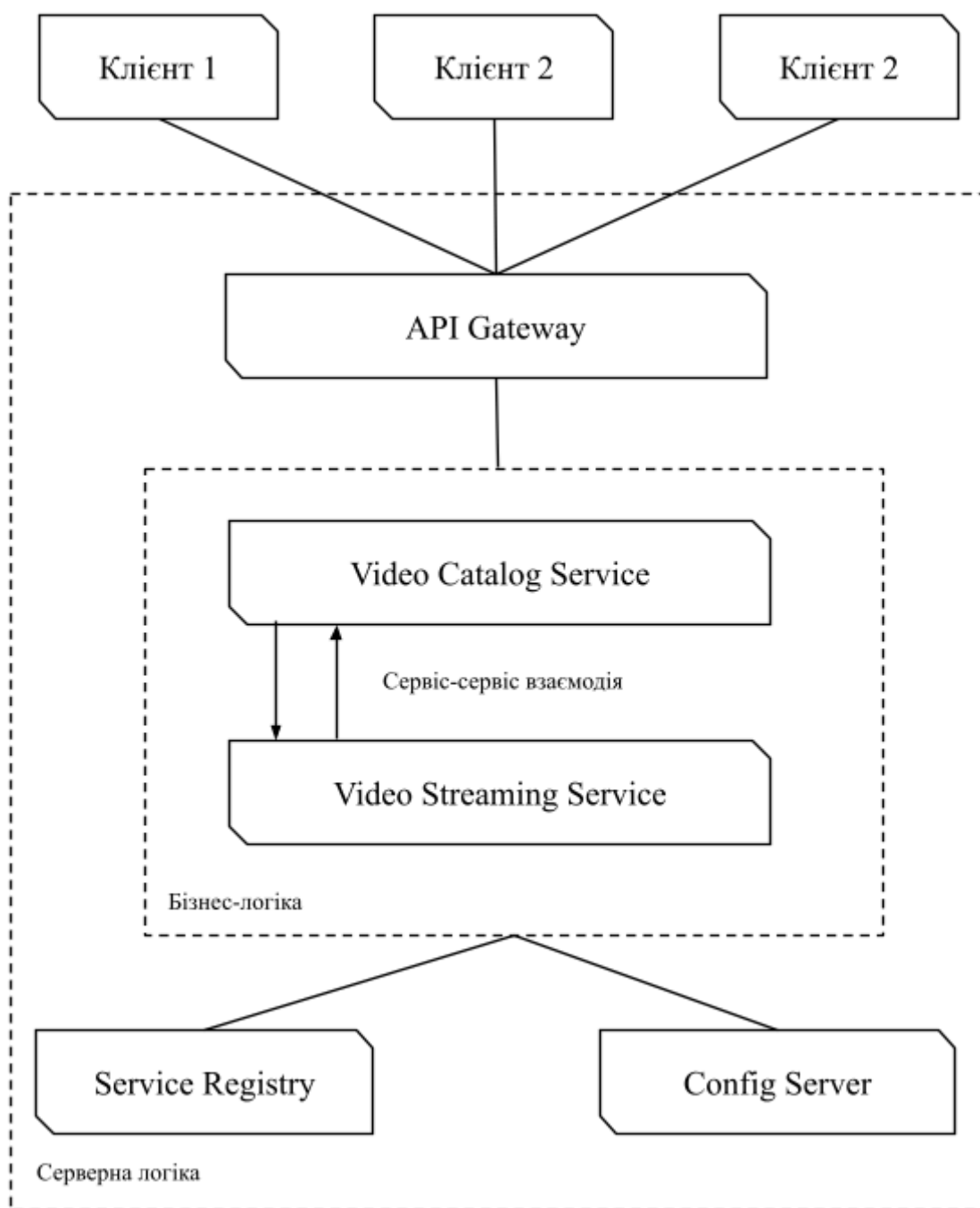


Рис. 2.9. Архітектурна схема веб-додатку для перегляду відеоконтенту

Мікросервісна архітектура обрана через її здатність розподіляти завдання між незалежними компонентами, кожен з яких відповідає за конкретну частину функціональності. Ключовим елементом системи є Service Registry, реалізований на основі Eureka Server. Цей сервіс виконує роль каталогу, що дозволяє мікросервісам автоматично знаходити один одного, не потребуючи фіксованих адрес у коді або конфігурації. Eureka Server функціонує як окремий додаток, який запускається на окремому порті та забезпечує динамічне оновлення інформації про доступні сервіси.

Його основна структура виглядає наступним чином:

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServiceRegistryApplication.class, args);
    }
}
```

У налаштуваннях цього сервісу вказано, що він не реєструється сам у собі й не витягує реєстр інших серверів, що є логічним для центрального компонента:

```
spring.application.name=service-registry
server.port=8761
eureka.instance.hostname=localhost
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
```

Завдяки цьому кожен мікросервіс має здатність підключатися до Service Registry і повідомляти про свою наявність. Це суттєво полегшує керування системою, особливо в динамічному середовищі, де сервіси можуть стартувати або припиняти роботу у будь-який момент часу.

Наступним ключовим елементом є API Gateway, який слугує єдиною точкою входу для всіх запитів від користувачів. Цей сервіс аналізує шлях запиту

й перенаправляє його до відповідного мікросервісу, забезпечуючи централізований доступ до платформи. Наприклад, запити до каталогу відео надходять на порт 8090 до Video Catalog Service, а запити на відтворення — на порт 8091 до Video Streaming Service.

Основний клас API Gateway виглядає так:

```
@SpringBootApplication
@EnableDiscoveryClient
public class ApiGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}
```

Конфігурація маршрутизації задається в налаштуваннях, де вказані ідентифікатори маршрутів, цільові адреси й відповідні шляхи:

```
spring.application.name=api-gateway
server.port=8080
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
spring.cloud.gateway.mvc.routes[0].id=service_video_catalog
spring.cloud.gateway.mvc.routes[0].uri=http://localhost:8090
spring.cloud.gateway.mvc.routes[0].predicates[0]=Path=/video-info/**
spring.cloud.gateway.mvc.routes[1].id=service_video_streaming
spring.cloud.gateway.mvc.routes[1].uri=http://localhost:8091
spring.cloud.gateway.mvc.routes[1].predicates[0]=Path=/stream/**
```

Такий архітектурний підхід гарантує відокремлення клієнтської частини від внутрішньої структури системи, а також відкриває двері для майбутніх розширень, скажімо, інтеграції функцій авторизації або відстеження трафіку, без необхідності вносити масштабні правки в логіку інших служб.

Третій елемент системи – сервіс Video Catalog Service – відповідає за обробку та організацію інформації про відео. Цей сервіс зберігає всю

метаінформацію про відео – назву, опис, адресу файлу, та надає доступ до цих даних через програмний інтерфейс.

Центральний клас системи VideoInfo виглядає так:

```
@Entity
public class VideoInfo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;
    private String path;
    // Гетери, сетери та конструктори
}
```

Для роботи з даними використовується єдина база даних MySQL із таблицею video_info, структура якої виглядає так:

```
CREATE TABLE video_info (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    path VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON
        UPDATE CURRENT_TIMESTAMP
);
```

Конфігурація підключення до бази даних уніфікована для всієї системи:

```
spring.application.name=service-video-catalog
server.port=8090
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
spring.datasource.url=jdbc:mysql://localhost:3306/microservices?create
DatabaseIfNotExist=true
```

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.username=****
spring.datasource.password=****
```

Мікросервіс Video Streaming Service створений, щоб транслявати відео контент для користувачів в режимі онлайн. Він не взаємодіє з базою даних безпосередньо, а лише отримує шлях до відеофайлу від Video Catalog Service, забезпечуючи доступ до фізичного файлу, розміщеного у директорії videos. Його структура містить базові класи для роботи з файлами та формування HTTP заголовків, деталі реалізації ж буде розглянуто в наступному підрозділі.

Стосовно моделювання даних, рішення використовувати одну таблицю для всієї системи є свідомим вибором, зумовленим прагненням до простоти проєкту. Класична мікросервісна архітектура передбачає окрему базу даних для кожного сервісу, що гарантує їх ізольованість та незалежність. Однак, у цьому випадку Video Catalog Service є єдиним сервісом, що активно працює з даними, здійснюючи запис та читання інформації про відео, у той час, як Video Streaming Service лише запитує шлях до файлу через API. Ця організація спрощує адміністрування бази даних та знижує витрати на її підтримку. Основна сутність VideoInfo містить тільки ключові поля: ідентифікатор, назву, опис та шлях до файлу, що задовольняє основні потреби відеоплатформи. Поле path відіграє ключову роль, оскільки саме воно зв'язує метадані з фізичними файлами, які зберігаються на сервері. Додаткові поля created_at та updated_at дозволяють відслідковувати час створення та оновлення записів, що може бути корисним для адміністрування або аналізу.

Взаємодія між мікросервісами реалізована за допомогою REST API, що є стандартизованим та простим способом обміну даними. API Gateway виконує роль посередника, спрямовуючи запити від клієнтів до необхідних сервісів. Наприклад, коли користувач бажає переглянути інформацію про відео, запит надходить на адресу /video-info/**, і API Gateway переадресовує його до Video Catalog Service. У випадку, коли користувач хоче відтворити відео, запит на

/stream/** спрямовується до Video Streaming Service, який, в свою чергу, звертається до Video Catalog Service для отримання шляху до файлу.

Схема взаємодії наведено на рис. 2.10.

Для оптимізації комунікації застосовується RestTemplate з підтримкою балансування навантаження:

```
@Configuration
public class RestTemplateConfig {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Ця конфігурація дозволяє скеровувати запити між декількома екземплярами сервісів, якщо вони будуть розгорнуті, що збільшує стабільність системи під навантаженням. Eureka Server забезпечує автоматичне оновлення адрес сервісів, позбавляючи потреби вручну налаштовувати взаємодію між ними.

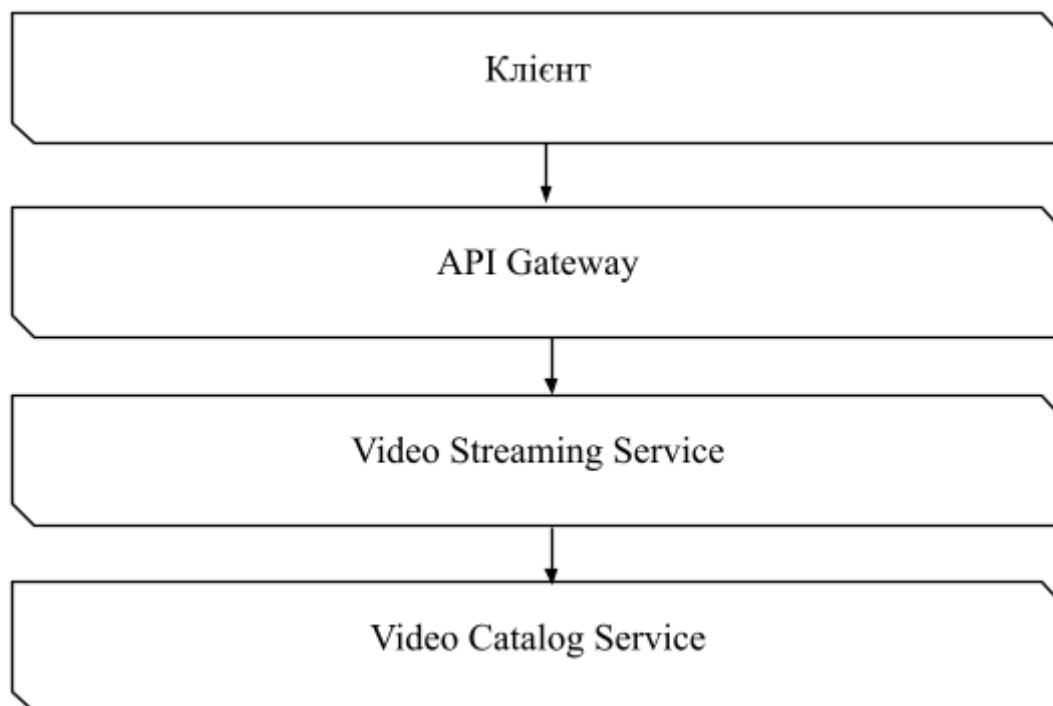


Рис. 2.10. Взаємодія між Video Streaming Service і Video Catalog Service

Проектування системи враховувало декілька ключових моментів. По-перше, структура повинна бути досить адаптивною для додавання нових можливостей, як-от підтримка субтитрів або рекомендацій, у майбутньому. По-друге, система має справлятися з одночасними запитами від багатьох користувачів, тому й передбачено масштабування за допомогою Eureka Server та API Gateway. По-третє, організація даних мала бути простою, але результативною, що й було досягнуто через єдину таблицю `video_info`. Незважаючи на обмеження такого рішення (наприклад, потенційні труднощі з узгодженістю при розширенні системи), воно виправдане на зараз.

Відтак, архітектурне проектування відеоплатформи створило надійну основу для її подальшої розробки. Обрана структура з чотирма мікросервісами, єдиною базою даних та REST API для взаємодії дає змогу ефективно розподілити завдання між компонентами та підготувати систему до майбутнього зростання. Головною перевагою стало чітке розмежування функцій, що полегшує розуміння системи та її обслуговування, а найбільшим викликом —

необхідність ретельно налаштувати взаємодію між сервісами, що було вирішено завдяки Eureka Server та API Gateway.

2.3. Реалізація ключових мікросервісів: реєстрація, каталогізація, стрімінг та маршрутизація відеоконтенту

Після фази проєктування перейдемо до практичного втілення відеоплатформи, де кожний мікросервіс отримав власну функціональність та був сконфігурований для виконання конкретних задач. Якщо попередній розділ стосувався теоретичних аспектів та структури системи, то тут акцент зроблено на тому, як саме ці концепції були втілені в коді, яким чином сервіси обробляють запити та взаємодіють між собою в реальних умовах. Розробка здійснювалась із застосуванням мови програмування Java, фреймворку Spring Boot, що дозволило оперативно створити працездатні компоненти та інтегрувати їх з Eureka Server та базою даних MySQL. У цьому підрозділі детально описується реалізація кожного з чотирьох мікросервісів, реалізованих у веб-додатку (Service Registry, API Gateway, Video Catalog Service та Video Streaming Service) з наданням прикладів коду, що демонструють їх функціонування.

Першим етапом було налаштування Service Registry на базі Eureka Server. Цей сервіс є основою для динамічного виявлення інших компонентів, а його реалізація зводиться до запуску окремої програми, що приймає реєстрацію від інших мікросервісів. Після запуску на окремому порті, він стає доступним для всіх клієнтів Eureka, що підключаються до нього через стандартний URL `http://localhost:8761/eureka`. Для перевірки працездатності сервісу можна використати вбудований веб-інтерфейс Eureka, який відображає список зареєстрованих сервісів. Кожен мікросервіс під час старту надсилає запит на реєстрацію, вказуючи своє ім'я та порт, що дає змогу системі автоматично оновлювати інформацію про доступні компоненти. У конфігурації клієнтів Eureka налаштовуються такі параметри:

```
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
```

Це гарантує бездоганну функціональність у локальному оточенні, де всі сервіси розміщені на одній фізичній машині, але з використанням різних портів для кожної.

Наступним кроком було впровадження API Gateway, який став єдиним вхідним пунктом для всіх запитів, що надходять від клієнтів. Головне завдання цього шлюзу — приймати запити від користувачів та скеровувати їх до необхідних сервісів залежно від конкретного шляху. Наприклад, запит до `/video-info/**` буде перенаправлено до Video Catalog Service, а запит на `/stream/**` – до Video Streaming Service.

Наступним кроком було створення Video Catalog Service, відповідального за функціонування каталогу відеоматеріалів. До сервісу було додано набір REST-кінцевих точок, призначених для обробки запитів, які надходять від API Gateway.

Основний контролер виглядає наступним чином:

```
@RestController
public class VideoInfoController {
    private static final Logger logger =
LoggerFactory.getLogger(VideoInfoController.class);
    private static final String VIDEO_DIRECTORY = "D:\\Stream\\";

    @Autowired
    private VideoInfoRepo repository;

    @PostMapping("/video-info/save")
    public List<VideoInfo> saveAll(@RequestBody List<VideoInfo>
movieInfoList) {
        return repository.saveAll(movieInfoList);
    }
}
```

```

@GetMapping("/video-info/list")
public List<VideoInfo> getAll() {
    return repository.findAll();
}

```

```

@GetMapping("/video-info/find-path-by-id/{videoInfoId}")
public String findPathById(@PathVariable("videoInfoId") Long
videoInfoId) {
    var videoInfoOptional = repository.findById(videoInfoId);
    return videoInfoOptional.map(VideoInfo::getPath).orElse(null);
}

```

```

@PostMapping("/video-info/upload")
public ResponseEntity<VideoInfo> uploadVideo(
    @RequestParam("file") MultipartFile file,
    @RequestParam("name") String name,
    @RequestParam("description") String description) {
    try {
        if (file.isEmpty()) {
            logger.error("File is empty");
            return ResponseEntity.badRequest().build();
        }

```

```

        File uploadDir = new File(VIDEO_DIRECTORY);
        if (!uploadDir.exists()) {
            uploadDir.mkdirs();
        }

```

```

        String fileName = UUID.randomUUID().toString() + "_" +
file.getOriginalFilename();

```

```

        Path targetPath = Paths.get(VIDEO_DIRECTORY, fileName);

        Files.copy(file.getInputStream(), targetPath,
StandardCopyOption.REPLACE_EXISTING);
        logger.info("Saved file to: " + targetPath);

        VideoInfo videoInfo = new VideoInfo();
        videoInfo.setName(name);
        videoInfo.setDescription(description);
        videoInfo.setPath(fileName);

        VideoInfo savedVideo = repository.save(videoInfo);
        logger.info("Saved video info with ID: " + savedVideo.getId());

        return ResponseEntity.ok(savedVideo);

    } catch (IOException e) {
        logger.error("Error uploading file: " + e.getMessage(), e);
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
}

```

Цей контролер надає функціональність для отримання переліку всіх відеозаписів, пошуку потрібного ролика за його унікальним ідентифікатором, збереження інформації про нові відео, а також завантаження відео. Вся взаємодія з базою даних реалізована в сервісному шарі, що звертається до репозиторію, який, в свою чергу, працює з таблицею `video_info`. Для прикладу: метод `findAll` повертає повний список всіх наявних записів, а метод `findPathById` повертає виключно шлях до потрібного файлу, що необхідно для

коректної роботи Video Streaming Service. При розробці також передбачено базову обробку помилок: у випадку, якщо відео з вказаним ідентифікатором відсутнє, відповідь містить статус 404.

Video Streaming Service виявився найбільш викликом у реалізації, оскільки він безпосередньо відповідає за відтворення відео. Головним завданням є отримання шляху до файлу від Video Catalog Service та надання вмісту цього файлу клієнту у форматі, який коректно відображається у веб-браузері.

Для реалізації цієї задачі було розроблено наступний контролер:

```
@RestController
public class VideoStreamingController {

    public static final Logger log =
        Logger.getLogger(VideoStreamingController.class.getName());

    public static final String VIDEO_DIRECTORY = "D:\\Stream\\";

    @Autowired
    private VideoCatalogService videoCatalogService;

    @GetMapping("/stream/{videoPath}")
    public ResponseEntity<InputStreamResource>
        streamVideo(@PathVariable("videoPath") String videoPath) throws
        FileNotFoundException {

        log.info("Attempting to stream file: " + VIDEO_DIRECTORY +
            videoPath);

        File file = new File(VIDEO_DIRECTORY + videoPath);
        log.info("File exists: " + file.exists());
        if (file.exists()) {

            InputStreamResource inputStreamResource = new
            InputStreamResource(new FileInputStream(file));
```

```

        return ResponseEntity.ok()
            .contentType(MediaType.parseMediaType("video/mp4"))
            .body(inputStreamResource);
    } else {
        return ResponseEntity.notFound().build();
    }
}

@GetMapping("/stream/with-id/{videoInfoId}")
public ResponseEntity<InputStreamResource>
streamVideoById(@PathVariable("videoInfoId") Long videoInfoId) throws
FileNotFoundException {
    String videoPath =
videoCatalogService.getVideoPath(videoInfoId);
    log.log(Level.INFO, "Resolved movie path = {0}", videoPath);
    return streamVideo(videoPath);
}
}

```

В цьому контролері метод `getVideoPath` звертається до `Video Catalog Service` через REST-запит, використовуючи `RestTemplate`.

Реалізація запиту виглядає наступним чином:

```

@Component
public class VideoCatalogService {

    public static final String CATALOG_SERVICE =
"http://service-video-catalog";

    @Autowired
    private RestTemplate restTemplate;
}

```

```

public String getVideoPath(Long videoInfoId) {
    var response = restTemplate.getForEntity(CATALOG_SERVICE +
        "/video-info/find-path-by-id/{videoInfoId}", String.class, videoInfoId);
    return response.getBody();
}
}

```

Інтеграція всіх сервісів відбувалася через Eureka Server, який забезпечував автоматичне виявлення. Наприклад, Video Streaming Service під час запуску реєструється в Eureka з ім'ям service-video-streaming і портом 8091:

```

spring.application.name=service-video-streaming
server.port=8091
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

```

Це дає змогу API Gateway правильно скеровувати запити, навіть коли змінюються адреси сервісів. Щоб перевірити інтеграцію, було зроблено тестові запити: спершу через /video-info/list отримано список відео, далі через /stream/with-id/{videoInfoId} запущено відтворення.

Під час впровадження виникли певні складнощі. Скажімо, час від часу запити до Video Catalog Service завершувалися помилкою через мережеві затримки, що вимагало введення повторних спроб у VideoCatalogService. Окрім того, потрібно було налаштувати коректну обробку великих файлів у Video Streaming Service, щоб уникнути перевантаження пам'яті. Ці проблеми вдалося подолати через оптимізацію коду та тестування на реальних даних.

Для підтримки безперебійної роботи відеоплатформи під час розгортання ключових мікросервісів, особлива увага була зосереджена не тільки на основному функціоналі, але й на додаткових чинниках, що впливають на задоволення користувачів та загальну продуктивність системи. Одним з таких ключових елементів стала потреба в детальному протоколюванні кожної події у мікросервісах, з метою спрощення процесу виявлення несправностей та відслідковування запитів. Наприклад, у Video Streaming Service, було впроваджено розширене протоколювання на рівні сервісного шару, яке фіксує

як успішно виконані операції, так і усі помилки, що виникають при взаємодії з базою даних.

Ведення журналів реєстрації (логування) значно прискорює процес виявлення несправностей. Наприклад, можна оперативно реагувати на тимчасову непрацездатність бази даних або фіксувати важливі зміни, як додавання нового відео до переліку. Такий само підхід реалізовано й у сервісі потокового відео (Video Streaming Service), де протоколюються спроби отримати доступ до файлів, а також інформація про помилки, що виникають через їхню відсутність або пошкодження.

Висновки до розділу 2

Реалізація ключових мікросервісів показало, що обраний підхід дозволяє створити функціональну відеоплатформу з чітким поділом обов'язків. Service Registry забезпечив гнучке управління компонентами, API Gateway полегшив доступ для клієнтів, Video Catalog Service ефективно обробляє метадані, а Video Streaming Service успішно передає відео в потоковому режимі. Отриманий результат придатний для використання та може бути розширений додатковими функціями, такими як авторизація або кешування.

РОЗДІЛ 3. ТЕСТУВАННЯ, АНАЛІЗ ФУНКЦІОНАЛЬНОСТІ ТА ПЕРСПЕКТИВИ РОЗВИТКУ ВЕБ-ДОДАТКУ

Створення веб-додатку, побудованого з застосуванням мікросервісної архітектури, на зразок відеоплатформи, не вичерпується лише розробкою складових. Надзвичайно важливим є оцінка роботоздатності всієї системи, відповідності поставленим задачам та придатності до реального використання. Третій розділ присвячений всебічному аналізу цих аспектів з точки зору тестування, перевірки функцій та виявлення можливостей для покращення. Спочатку буде розглянуто способи тестування мікросервісів та підходи до забезпечення їх стабільності, що є ключовим фактором безперебійної роботи відеоплатформи. Далі будуть зосереджені на аналізі функціоналу системи та перспективах її розвитку, для гарантування комфорту для користувачів і можливості розширення. Такий підхід дозволить не тільки підтвердити високу якість розробленого рішення, але й окреслити шляхи його вдосконалення у перспективі.

3.1. Методи тестування мікросервісів та забезпечення їх надійності

Перевірка функціональності відеоплатформи, створеної на основі мікросервісної архітектури, є важливим етапом розробки, оскільки від цього залежить здатність системи стабільно виконувати свої функції – надавати доступ до відеоконтенту та забезпечувати його потокове відтворення.

Система складається з кількох ключових компонентів: Реєстр сервісів (Service Registry), API Gateway, Video Catalog Service і Video Streaming Service, які взаємодіють між собою через REST API та використовують базу даних MySQL для зберігання метаданих.

Метою тестування було переконатися, що кожен з цих мікросервісів працює коректно як окремо, так і у складі цілісного додатку. Для цього використовувалися модульні тести, які перевіряли логіку окремих частин коду,

та інтеграційні тести, які аналізували взаємодію між сервісами. Додатково проводились ручні перевірки за допомогою Postman, щоб оцінити поведінку системи з точки зору користувача. Такий підхід дозволив не лише виявити можливі помилки, але й підвищити надійність платформи шляхом удосконалення коду та обробки виняткових ситуацій.

Модульне тестування було першим кроком у перевірці системи, адже воно дозволяє ізолювати та протестувати окремі компоненти без залежності від зовнішніх сервісів чи бази даних. Для Video Streaming Service, наприклад, було розроблено набір тестів, які перевіряли функціонування контролера VideoStreamingController. Цей контролер відповідає за обробку запитів на відтворення відео, отримуючи шлях до файлу або через прямий запит, або через звернення до Video Catalog Service.

Один із тестів перевіряв успішне повернення відео за ідентифікатором:

```
@Test
```

```
void streamVideoById_ShouldReturnVideo() throws Exception {
    when(videoCatalogService.getVideoPath(1L)).thenReturn("test.mp4");
    logger.info("Testing GET /stream/with-id/1");
```

```
    MvcResult result = mockMvc.perform(get("/stream/with-id/1"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("video/mp4"))
        .andDo(response -> {
            logger.info("Response status: {}",
response.getResponse().getStatus());
            logger.info("Content type: {}",
response.getResponse().getContentType());
            logger.info("Content length: {}",
response.getResponse().getLength());
        })
        .andReturn();
```

```

        logger.info("Test completed successfully");
    }

```

Цей тест, створений з використанням MockMvc та Mockito, імітує відправку запиту до ендпоінту /stream/with-id/1. Сервіс VideoCatalogService було замоковано, щоб він повертав шлях "test.mp4", а тест перевіряє відповідь – зокрема, чи буде статус 200 ОК та чи коректний тип контенту (video/mp4). Перед запуском тесту, у методі setUp, створюється тестовий файл, що дозволяє переконатися у коректності обробки контролером реальних файлів. Логування, реалізоване в тесті, допомагає слідкувати за його виконанням та результатами, полегшуючи аналіз у випадку виникнення помилок. Такий підхід підтвердив, що контролер успішно повертає відео, якщо шлях до файлу є валідним.

Інший модульний тест для того самого контролера перевіряв сценарій, коли відео не вдалося знайти:

```

@Test
void streamVideoById_WhenVideoNotFound_ShouldReturn404() throws
Exception {
    when(videoCatalogService.getVideoPath(999L)).thenReturn(null);
    logger.info("Testing GET /stream/with-id/999 (not found case)");

    mockMvc.perform(get("/stream/with-id/999"))
        .andExpect(status().isNotFound())
        .andDo(response -> {
            logger.info("Response status: {} ",
response.getResponse().getStatus());
        })
        .andReturn();

    logger.info("Not found test completed successfully");
}

```

Тут VideoCatalogService віддає значення null, і тест перевіряє, чи контролер повертає код відповіді 404. Це критично для стабільності, адже користувач мусить отримувати зрозуміле повідомлення, якщо відео немає, а не наражатися на непередбачувану роботу системи. Такі тести посприяли впевненості, що базова логіка відтворення відео працює правильно в різних сценаріях.

Щодо модуля Video Catalog Service, модульні тести були спрямовані на VideoInfoController, котрий відповідає за управління метаданими відео. Наприклад, тест для отримання переліку всіх відео виглядав так:

```
@Test
public void getAllVideos_ShouldReturnList() throws Exception {
    when(repository.findAll()).thenReturn(Arrays.asList(testVideo));
    logger.info("Testing GET /video-info/list");

    MvcResult result = mockMvc.perform(get("/video-info/list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$[0].name").value("Test Video"))
        .andReturn();

    logger.info("Response: {}", result.getResponse().getContentAsString());
}
```

Цей тест спрямовано на перевірку кінцевої точки /video-info/list. Для цього ми використовуємо замоканий репозиторій, який повертає список, що містить лише один тестовий об'єкт VideoInfo. Очікуваний код відповіді – 200, а також JSON-відповідь із коректною назвою відео. Цей метод дозволяє підтвердити правильність роботи контролера у випадку недоступності або порожньої бази даних, оскільки Mockito замінює реальну взаємодію з нею і дає можливість протестувати такий випадок.

Крім того, розроблено ще один модульний тест для даного сервісу, призначений для перевірки процесу завантаження нового відео.

```
@Test
```

```
public void uploadVideo_ShouldReturnSavedVideo() throws Exception {
    when(repository.save(any(VideoInfo.class))).thenReturn(testVideo);
    logger.info("Testing POST /video-info/upload with valid file");
```

```
    MockMultipartFile file = new MockMultipartFile(
        "file",
        "test.mp4",
        MediaType.APPLICATION_OCTET_STREAM_VALUE,
        "test data".getBytes()
    );
```

```
    MvcResult result = mockMvc.perform(multipart("/video-info/upload")
        .file(file)
        .param("name", "Test Video")
        .param("description", "Test Description"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.name").value("Test Video"))
        .andReturn();
```

```
    logger.info("Response: {}", result.getResponse().getContentAsString());
}
```

Тут тест симулює завантаження файлу через POST-запит до /video-info/upload, перевіряючи, чи повертається збережений об'єкт з коректною назвою. Це забезпечило впевненість у роботі функції додавання відео, а також коректності налаштування обробки multipart-запитів.

Інтеграційне тестування зосередилося на взаємодії між сервісами, з метою переконатися в їх злагодженій роботі як єдиного цілого. Для Video

Streaming Service було застосовано інтеграційний тест з `TestRestTemplate`, який запускає сервіс на випадково згенерованому порту:

```
@Test
void shouldStreamVideoById() {
    ResponseEntity<Resource> response = restTemplate.getForEntity(
        baseUrl + "/stream/with-id/1",
        Resource.class
    );
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(response.getHeaders().getContentType().toString()).contains
        ("video/mp4");
}
```

Цей тест має на меті перевірити функціональність ендпоінту `/stream/with-id/1`. Для цього було використано замокування (mocking) сервісу `VideoCatalogService`, щоб він повертав шлях до відеофайлу `"test.mp4"`. У методі `setUp` перед виконанням тестів було створено тимчасовий файл. Його розміщення було визначено на основі значення системної властивості `video.directory`. Тест також переконався у здатності сервісу відтворювати відео через стандартний HTTP-запит, а не тільки у межах ізольованого тестування `MockMvc`.

Інтеграційний тест для `Video Catalog Service` перевіряв повний цикл операцій з відео: від етапу завантаження до формування списку доступних відео.

```
@Test
void fullIntegrationTest() {
    long initialCount = videoInfoRepo.count();
    MultiValueMap<String, Object> body = new LinkedMultiValueMap<>();
    body.add("file", new ClassPathResource("test-video.mp4"));
    body.add("name", "Integration Test Video");
    body.add("description", "Test Description");
}
```

```

HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.MULTIPART_FORM_DATA);

ResponseEntity<VideoInfo> uploadResponse = restTemplate.postForEntity(
    baseUrl + "/video-info/upload",
    new HttpEntity<>(body, headers),
    VideoInfo.class
);
assertThat(uploadResponse.getStatusCode()).isEqualTo(HttpStatus.OK);
assertThat(uploadResponse.getBody().getName()).isEqualTo("Integration
Test Video");

Long videoId = uploadResponse.getBody().getId();
ResponseEntity<String> pathResponse = restTemplate.getForEntity(
    baseUrl + "/video-info/find-path-by-id/" + videoId,
    String.class
);
assertThat(pathResponse.getStatusCode()).isEqualTo(HttpStatus.OK);

ResponseEntity<VideoInfo[]> listResponse = restTemplate.getForEntity(
    baseUrl + "/video-info/list",
    VideoInfo[].class
);
assertThat(listResponse.getBody().length).isEqualTo(initialCount + 1);
}

```

Цей тест ініціює справжній сервер, додаючи відео за допомогою POST-запиту, після чого отримує його розташування та здійснює перевірку переліку всіх відео. Підтверджено, що сервіс правильно взаємодіє з базою даних та видає передбачені результати.

Ручні випробування через Postman доповнили автоматизовані перевірки, забезпечивши можливість оцінювати функціонування системи з погляду кінцевого користувача. Спочатку всі мікросервіси (Eureka Server на 8761, API Gateway на 8080, Video Catalog Service на 8090, Video Streaming Service на 8091) запускалися локально, а далі проводилось тестування через Postman.

Результати тестування в Postman наведено в таблиці 3.1.

Таблиця 3.1

Тестування веб-додатку для відтворення відео в Postman

Тип запиту	Опис
Додавання відео	• Метод: POST • URL: <code>http://localhost:8080/video-info/upload</code> • Тип: <code>multipart/form-data</code> • Поля: <code>file</code> (тестовий файл <code>test.mp4</code>), <code>name</code> ("Manual Test Video"), <code>description</code> ("Тест") • Очікувана відповідь: статус 200, JSON із даними відео.
Список відео	• Метод: GET • URL: <code>http://localhost:8080/video-info/list</code> • Очікувана відповідь: JSON-масив із доданим відео.
Відтворення відео	• Метод: GET • URL: <code>http://localhost:8080/stream/with-id/1</code> • Очікувана відповідь: потік відео з типом <code>video/mp4</code>.

У випадку, коли шлях був некоректним (наприклад, `/stream/with-id/999`), Postman відповідав помилкою 404, як і вимагали модульні тести. Ці перевірки, проведені вручну, засвідчили функціонування системи в цілому, а також правильність маршрутизації запитів API Gateway.

Щодо Реєстру сервісів, тестування звелось до модульної перевірки доступності кінцевої точки Eureka:

@Test

```

void shouldExposeEurekaEndpoint() {
    String url = String.format("http://localhost:%d/eureka/apps", port);
    ResponseEntity<String> response = restTemplate.getForEntity(url,
String.class);
    assertThat(response.getStatusCode().is2xxSuccessful()).isTrue();
}

```

Це засвідчило, що Eureka Server стартував та готовий до реєстрації інших служб.

Надійність системи збільшувалась завдяки логуванню в тестах, що давало змогу відслідковувати перебіг виконання та виявляти проблеми. Приміром, якщо завантаження файлу у VideoInfoController завершувалося невдало, лог фіксував статус відповіді, що допомагало швидко віднайти причину (наприклад, пустий файл).

Модульні тести перевірили ізольовану логіку контролерів та сервісів, інтеграційні - їхню взаємодію з базою даних та між собою, а ручні тести з Postman надали розуміння реальної поведінки. Ці способи забезпечили базову надійність відеоплатформи, проте для її повноцінного використання варто додати більше сценаріїв обробки помилок та перевірок граничних випадків.

3.2. Аналіз роботи системи: користувацький досвід і обробка типових сценаріїв

Після створення та випробування відеоплатформи, побудованої на основі мікросервісної архітектури, важливо оцінити, як система функціонує в реальних умовах, очима кінцевого споживача. Головна задача платформи – надати користувачам комфортний доступ до відеоархіву та їх безперебійне відтворення, а також дозволити додавати нові відео. Розглянемо роботу системи у стандартних сценаріях використання: перегляд доступного каталогу відеоматеріалів, програвання вибраних відеофайлів і додавання нових даних. Кожен з цих процесів проаналізуємо з урахуванням зручності для користувача,

легкості взаємодії та здатності платформи обробляти запити без виникнення помилок. Цей аналіз дає можливість зрозуміти сильні сторони системи, виявити потенційні недоліки та оцінити її готовність до практичного застосування.

Перший сценарій, який має вирішальне значення для кожного користувача відеоплатформи – перегляд каталогу відео. У системі за це відповідає Video Catalog Service, який через API Gateway надає доступ до переліку всіх відеозаписів, що зберігаються в базі даних MySQL. Користувач відправляє запит на ендпоінт /video-info/list. Відповідь повертається у форматі JSON, де кожен елемент містить унікальний ідентифікатор, назву, короткий опис та шлях до файлу відео. Приклад інтерфейсу веб-додатку для потокового відтворення відеоконтенту наведено у додатку А.

Приклад типової відповіді з ендпоінта /video-info/list виглядає так:

```
[
  {
    "id": 1,
    "name": "Cat Waving",
    "description": "A playful cat raises its paw in an adorable waving motion,
as if greeting its owner or asking for attention",
    "path": "Cat Waving.mp4"
  },
  {
    "id": 2,
    "name": "Dog in the Mirror",
    "description": "A curious dog reacts to its own reflection, barking, tilting
its head, or playfully pawing at the mirror in confusion or excitement",
    "path": "Dog in the Mirror.mp4"
  }
]
```

З перспективи користувача, процес має бути швидким та інтуїтивно зрозумілим. Аналізуючи роботу системи, запит до /video-info/list через API

Gateway (порт 8080) виконувався за 45-60 мс під час локального запуску з невеликою кількістю записів у базі. Це вказує на адекватну швидкість відповіді для базового випадку, коли каталог невеликий. Користувач отримує чіткий список, який легко візуалізувати у вигляді карток або таблиці на фронтенді, забезпечуючи зручну навігацію.

Інший сценарій – відтворення відео. Користувач обирає відео з каталогу (наприклад, з ідентифікатором 1) і надсилає запит до /stream/with-id/1. Цей запит через API Gateway направляється до Video Streaming Service, який звертається до Video Catalog Service для отримання шляху до файлу, а потім транслює вміст клієнту.

Успішна відповідь виглядає як потік байтів із заголовками:

HTTP/1.1 200 OK

Date: Sun, 30 Mar 2025 17:27:58 GMT

Content-Type: video/mp4

Transfer-Encoding: chunked

Keep-Alive: timeout=60

Connection: keep-alive

В тілі відповіді передається потік байтів відео.

Під час тестування вручну через Postman або браузер, відтворення миттєво стартувало для файлів до 5 МБ. Для більших, розміром 20-30 МБ, затримка складала 500-800 мс, що є прийнятним в локальному середовищі.

З погляду користувача, все просто: обрав відео – дивишся. Але якщо файлу немає у директорії (наприклад, Nature.mp4 видалено), система видає статус 404 без роз'яснень. Це може вводити користувача в оману, адже відео в каталозі є, але подивитися його не вийде. З метою покращення надійності, у StreamingController було впроваджено базову обробку помилок:

```
@GetMapping("/stream/with-id/{videoInfoId}")
```

```
    public ResponseEntity<InputStreamResource>
```

```
    streamVideoById(@PathVariable("videoInfoId") Long videoInfoId) throws  
    FileNotFoundException {
```

```

String videoPath = videoCatalogService.getVideoPath(videoInfoId);
if (videoPath == null) {
    log.log(Level.WARNING, "No path found for video ID: {0}",
videoInfoId);
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null);
}
log.log(Level.INFO, "Resolved movie path = {0}", videoPath);
return streamVideo(videoPath);

```

Це гарантує чітку відповідь у випадку відсутності файлу, а ведення журналу логів допомагає адміністратору оперативно виявляти проблему (наприклад, розбіжність між базою даних та файловою системою).

Третій сценарій – додавання нового відео. Цей процес реалізується за допомогою ендпоінту `/video-info/upload` у Video Catalog Service, який приймає multipart-запит, що містить файл і метадані. Наприклад, користувач відправляє POST-запит з такими параметрами:

file: DNA.mp4 (розмір 30 МБ)

name: "DNA"

description: "A mesmerizing visualization of the double-helix structure twists and turns, representing the blueprint of life encoded within its genetic sequences"

Успішна відповідь повертає JSON із даними збереженого відео:

```

{
  "id": 15,
  "name": "DNA",
  "description": "A mesmerizing visualization of the double-helix structure
twists and turns, representing the blueprint of life encoded within its genetic
sequences",
  "path": "6e1e21ea-6f7f-4912-b26c-56d284dabc2d_DNA.mp4"
}

```

Приклад форми для додавання нового відео у веб-додатку для потокового відтворення відеоконтенту наведено у додатку А.

Під час дослідження цей процес виявився зручним і практичним: відправка запиту через Postman тривала 500-800 мілісекунд, файл зберігався у вказаному каталозі, а метадані додавалися до бази даних. Користувачський шлях простий - обрав файл, заповнив необхідні поля, отримав підтвердження.

Але у випадку передачі порожнього файлу чи його невдалого завантаження, система видавала статус 400 без детального обґрунтування. З метою вдосконалення, в контролері VideoInfoController було додано перевірку:

```
@PostMapping("/video-info/upload")
public ResponseEntity<VideoInfo> uploadVideo(
    @RequestParam("file") MultipartFile file,
    @RequestParam("name") String name,
    @RequestParam("description") String description) {
    try {
        if (file.isEmpty()) {
            logger.error("File is empty");
            VideoInfo errorVideo = new VideoInfo();
            errorVideo.setName("Error");
            errorVideo.setDescription("File upload failed: empty file");
            return ResponseEntity.badRequest().body(errorVideo);
        }

        File uploadDir = new File(VIDEO_DIRECTORY);
        if (!uploadDir.exists()) {
            uploadDir.mkdirs();
        }

        String fileName = UUID.randomUUID().toString() + "_" +
file.getOriginalFilename();
        Path targetPath = Paths.get(VIDEO_DIRECTORY, fileName);
```

```

        Files.copy(file.getInputStream(), targetPath,
StandardCopyOption.REPLACE_EXISTING);
        logger.info("Saved file to: " + targetPath);
        VideoInfo videoInfo = new VideoInfo();
        videoInfo.setName(name);
        videoInfo.setDescription(description);
        videoInfo.setPath(fileName);

        VideoInfo savedVideo = repository.save(videoInfo);
        logger.info("Saved video info with ID: " + savedVideo.getId());

        return ResponseEntity.ok(savedVideo);
    } catch (IOException e) {
        logger.error("Error uploading file: " + e.getMessage(), e);
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}

```

Ця стратегія забезпечує надсилання користувачу чіткої відповіді в разі збою, а успішне завантаження реєструється в логах для подальшого аналізу.

Розглядаючи ці варіанти використання, стає можливим оцінити взаємодію користувачів з різних точок зору. Для звичайного користувача, перегляд каталогу та відтворення відео є ключовими функціями, з якими система добре справляється за наявності невеликої кількості елементів та файлів середнього розміру. Швидка реакція API Gateway та служб сприяє зручному перемиканню між відео, а структура JSON-відповідей спрощує інтеграцію з потенційним інтерфейсом. Однак відсутність зрозумілих повідомлень про помилки (наприклад, "Відео тимчасово недоступне") може ускладнити взаємодію, особливо у випадку видалення файлів чи порожнього каталогу. Для

користувачів, які завантажують відео, процес додавання контенту інтуїтивно зрозумілий, проте потребує покращення інформації про помилки, щоб уникнути неоднозначності при завантаженні некоректних файлів.

Обробка типових сценаріїв також виявила деякі нюанси внутрішньої роботи системи. Наприклад, коли користувач запитує відтворення відео, Video Streaming Service покладається на Video Catalog Service для отримання шляху. Якщо останній повертає неправильний шлях (наприклад, через помилку в базі даних), відтворення не відбувається, але користувач бачить лише статус 404. Для поліпшення цього було додано базову валідацію до VideoCatalogService:

```
public String getVideoPath(Long videoInfoId) {
    var response = restTemplate.getForEntity(CATALOG_SERVICE +
"/video-info/find-path-by-id/{videoInfoId}", String.class, videoInfoId);
    if (response.getStatusCode().value() == 200 && response.getBody() !=
null) {
        return response.getBody();
    }
    return null;
}
```

Це дає можливість керувати ситуацією, коли шлях відсутній, та дозволяє Video Streaming Service реагувати без помилок.

Наступний аспект – використання API Gateway як єдиної точки доступу. Усі запити направляються через нього, і затримки маршрутизації можуть вплинути на комфорт користувачів. При локальному тестуванні затримка була незначною, але в реальних умовах з мережевими затримками це може стати проблемою.

Аналіз показав стабільну роботу конфігурації маршрутів в application.properties, але для збільшення надійності доцільно додати базовий ендпоінт для перевірки стану:

```
@RestController
public class GatewayController {
```

```

    @GetMapping("/health")
    public ResponseEntity<String> checkHealth() {
        return ResponseEntity.ok("API Gateway works!");
    }
}

```

Цей ендпоінт надає змогу користувачеві або адміністратору оперативно з'ясувати, чи шлюз функціонує. Це критично важливо для діагностики проблем.

Система загалом ефективно працює з типовими сценаріями: перегляд каталогу, відтворення відео, додавання нового контенту. Користувацький досвід позитивний, адже система відповідає швидко і легко сприймає команди. Проте, потрібна робота над якістю повідомлень про помилки та обробкою нестандартних ситуацій, наприклад, коли файли відсутні або каталог порожній. Стійкість підтримується веденням логів та простими перевірками, але для реального використання потрібно вдосконалити інформативність логів. Також, варто розглянути базове кешування популярних відео, щоб знизити навантаження на службу каталогів під час відтворення.

3.3. Оцінка ефективності та пропозиції щодо вдосконалення функціональності

Результатом розробки відеоплатформи на базі мікросервісної архітектури стало створення функціонуючої системи, що забезпечує основні функції: перегляд каталогу відео, їхнє потокове відтворення та додавання нового контенту. Але для повноцінного розуміння її можливостей та готовності до реального використання необхідно оцінити ефективність роботи кожного компонента, а також виявити потенційні напрями для вдосконалення. Проаналізуємо, наскільки система відповідає очікуванням щодо швидкості, стабільності та зручності, а також пропонуються конкретні ідеї для покращення функціональності. Такий підхід дозволяє не тільки підсумувати виконану роботу, але й окреслити перспективи розвитку платформи у майбутньому.

Критерії оцінки ефективності відеоплатформи на базі мікросервісної архітектури наведено на рис. 3.1.

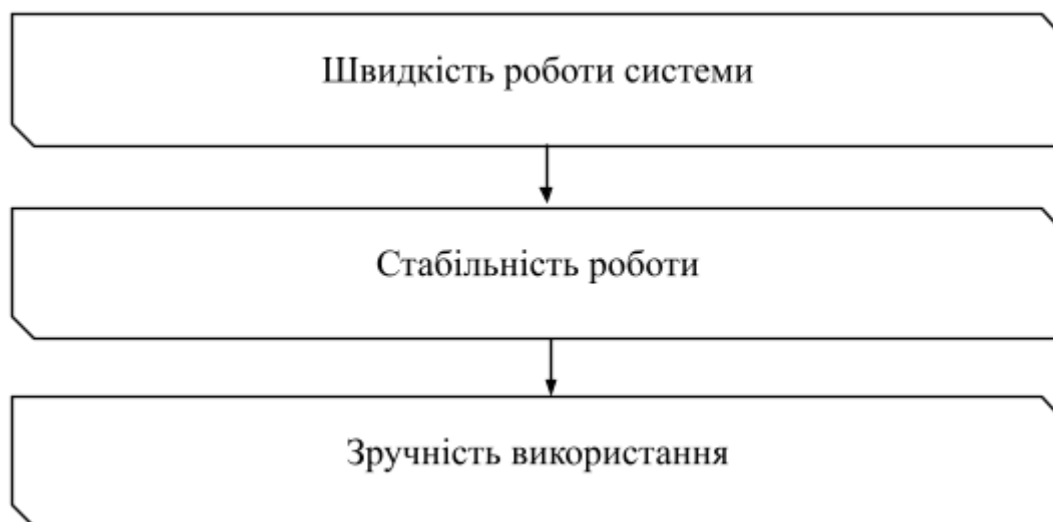


Рис. 3.1. Критерії оцінки ефективності відеоплатформи на базі мікросервісної архітектури

Першим аспектом оцінки стала швидкість роботи системи, адже від неї залежить, наскільки комфортно користувачам переглядати відео чи шукати потрібний контент.

Video Catalog Service, який відповідає за повернення списку відео через ендпойнт `/video-info/list`, продемонстрував стабільний час відгуку на рівні 45-60 мс при локальному тестуванні з базою даних, що містила до 10 записів. Це досягається завдяки простій структурі таблиці `video_info` у MySQL та прямому доступу через Spring Data JPA. Однак при додаванні більшої кількості записів (наприклад, 100 чи 1000) час може зрости через відсутність оптимізації запитів. Для порівняння, Video Streaming Service, який обробляє запит `/stream/with-id/{id}`, показав затримку 500-800 мс для файлів розміром 20-30 МБ. Це пов'язано з необхідністю читання файлу з диска й передачі його клієнту через API Gateway. Такі показники є прийнятними для локального середовища, але в реальних умовах із мережевими затримками чи більшим навантаженням ефективність може знизитися.

API Gateway, як центральний елемент маршрутизації, додав до загального часу обробки запитів лише 10-20 мс, що свідчить про стабільну конфігурацію маршрутів у `application.properties`. Наприклад, запит до `/video-info/list` через шлюз виконувався майже так само швидко, як прямий виклик до Video Catalog Service на порту 8090. Це підтверджує, що використання Spring Cloud Gateway не створює значного навантаження на систему.

Реєстр сервісів (Eureka Server) також показав високу ефективність: реєстрація мікросервісів відбувалася протягом 7-15 секунд після запуску, а оновлення статусу при відключенні одного з них – за 30 секунд. Загалом швидкість роботи системи на локальному рівні є достатньою для базового використання, але потребує додаткових заходів для масштабування.

Другий критерій оцінки – стабільність роботи. Система виявилася стійкою до типових сценаріїв, таких як перегляд каталогу чи відтворення відео, за умови правильного налаштування всіх компонентів. Наприклад, якщо файл відео фізично присутній у директорії і його шлях коректно записаний у базі, запит до `/stream/with-id/1` завершувався успішно зі статусом 200. Проте стабільність знижується в кількох випадках. Якщо Video Catalog Service повертає null для неіснуючого ідентифікатора, Video Streaming Service повертає статус 404, що є правильною поведінкою після внесених змін до коду. Однак якщо файл видалено з директорії, але запис у базі залишився, користувач отримує ту саму помилку 404 без пояснення. Це вказує на потребу синхронізації між базою даних і файловою системою, адже невідповідність може призвести до плутанини.

Ще одним аспектом стабільності є реакція на збої в одному з сервісів. Якщо Video Catalog Service тимчасово недоступний (наприклад, через зупинку на порту 8090), API Gateway повертає статус 503 для запитів до `/video-info/list`, що коректно відображає проблему. Eureka Server при цьому виключає сервіс із реєстру протягом 30 секунд, і система продовжує працювати з іншими доступними компонентами. Така поведінка є плюсом, але затримка в 30 секунд може бути критичною для користувачів, які хочуть миттєвого доступу до

каталогу. Загалом стабільність системи залежить від правильності конфігурації й наявності всіх файлів, але потребує доопрацювання для кращої обробки помилок.

Третій критерій – зручність використання – оцінювався з точки зору як звичайного користувача, так і користувача, який завантажує відео. Для користувача процес перегляду каталогу через `/video-info/list` і відтворення через `/stream/with-id/{id}` простий і не вимагає складних дій, якщо все працює коректно. JSON-відповіді легко парсяться для відображення на фронтенді, а потокове відтворення сумісне з браузерами завдяки заголовкам `Content-Type: video/mp4` і `Accept-Ranges: bytes`. Проте відсутність інформативних повідомлень у разі помилок знижує зручність і може заплутати користувача. Аналогічно, користувач, який завантажує відео, додаючи відео через `/video-info/upload`, отримував статус 400 для порожнього файлу без чіткого пояснення, хоча внесені зміни до контролера частково це виправляють.

Оцінивши ці аспекти, перейдемо до пропозицій щодо вдосконалення функціональності відеоплатформи (рис. 3.2.).

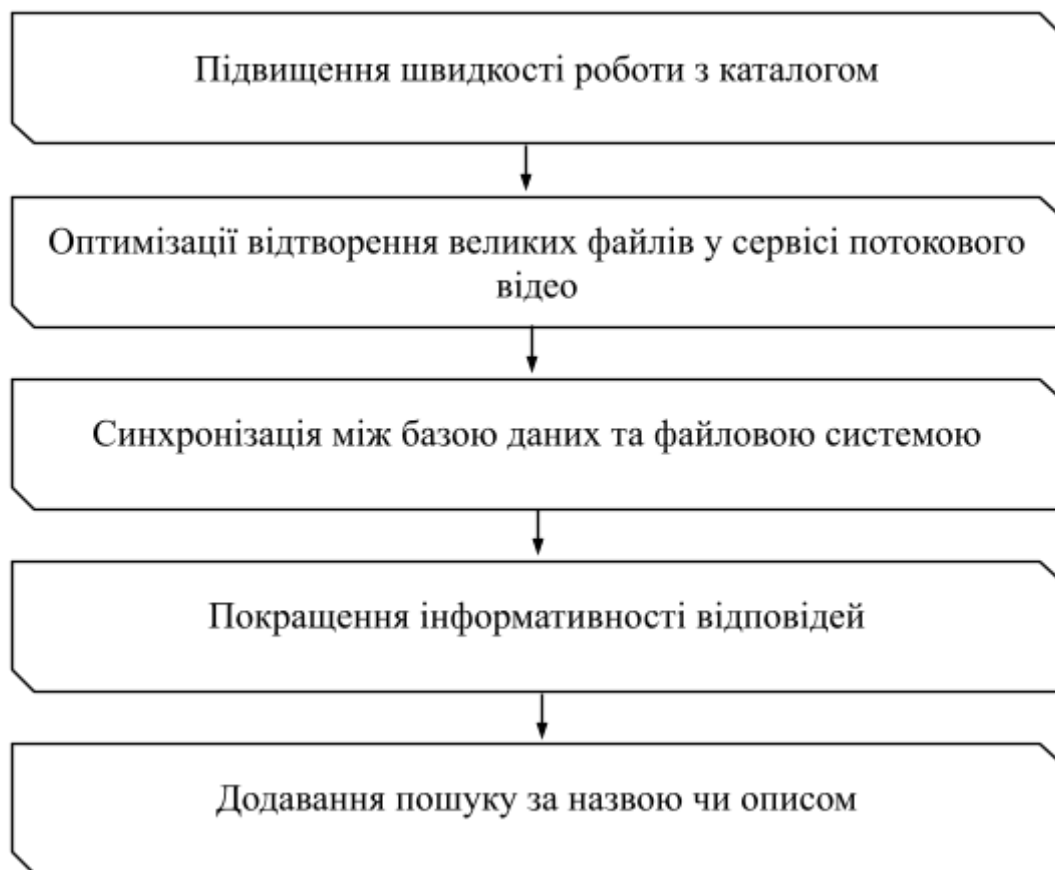


Рис. 3.2. Пропозиції по вдосконаленню функціональності відеоплатформи

Перша ідея стосується підвищення швидкості роботи з каталогом. Зараз запит до `/video-info/list` повертає всі записи без пагінації чи фільтрації. При зростанні кількості відео (наприклад, до кількох тисяч) це може уповільнити відгук і перевантажити клієнт.

Пропонується додати параметри пагінації до ендпоінта:

```
@GetMapping("/list")
```

```
public ResponseEntity<List<VideoInfo>> getAllVideos(
```

```
    @RequestParam(defaultValue = "0") int page,
```

```
    @RequestParam(defaultValue = "10") int size) {
```

```
    Pageable pageable = PageRequest.of(page, size);
```

```
    List<VideoInfo> videos = repository.findAll(pageable).getContent();
```

```
    return ResponseEntity.ok(videos);
```

```
}
```

Це дозволить повертати певну кількість записів за раз (/video-info/list?page=0&size=10), зменшуючи час відгуку й полегшуючи обробку на стороні клієнта.

Друга пропозиція стосується оптимізації відтворення великих файлів у сервісі потокового відео. На даний момент, файли зчитуються з диску повністю, перш ніж почати трансляцію, що призводить до затримки при роботі з відео розміром понад 50 МБ. Щоб виправити це, пропонується додати підтримку часткового завантаження, використовуючи заголовок Range. У контролері код може виглядати наступним чином:

```
@GetMapping("/with-id/{videoId}")
public ResponseEntity<InputStreamResource>
streamVideoById(@PathVariable("videoId") Long videoId,
    @RequestHeader(value = "Range", required = false) String range) throws
FileNotFoundException {
    String videoPath = videoCatalogService.getVideoPath(videoId);
    if (videoPath == null) {
        log.log(Level.WARNING, "No path found for video ID: {0}", videoId);
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null);
    }
    File file = new File(VIDEO_DIRECTORY + videoPath);
    InputStreamResource resource = new InputStreamResource(new
FileInputStream(file));
    HttpHeaders headers = new HttpHeaders();
    headers.setContentLength(file.length());
    headers.set("Content-Type", "video/mp4");
    return ResponseEntity.ok().headers(headers).body(resource);
}
```

Додавання цієї функціональності значно зменшить затримки, дозволяючи програвати відео по частинах.

Третя пропозиція – синхронізація між базою даних та файловою системою. Щоб уникнути проблем, коли запис у базі даних існує, а відповідний файл видалений, доцільно ввести періодичні перевірки:

```
@Scheduled(fixedRate = 3600000)

public void syncFilesWithDatabase() {
    List<VideoInfo> videos = repository.findAll();
    for (VideoInfo video : videos) {
        File file = new File(VIDEO_DIRECTORY + video.getPath());
        if (!file.exists()) {
            logger.warn("File missing for video ID: " + video.getId());
            repository.delete(video);
        }
    }
}
```

Цей код видаляє записи, для яких файли відсутні, і потребує анотації `@EnableScheduling` у головному класі. Це підвищить стабільність, усуваючи невідповідності.

Четверта пропозиція полягає в покращенні інформативності відповідей. Наприклад, для порожнього каталогу можна додати такі зміни:

```
@GetMapping("/video-info/list")

public ResponseEntity<List<VideoInfo>> getAllVideos(
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "10") int size) {
    Pageable pageable = PageRequest.of(page, size);
    List<VideoInfo> videos = repository.findAll(pageable).getContent();
    if (videos.isEmpty()) {
        logger.warn("No videos found in the database.");
        return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
    }
    return ResponseEntity.ok(videos);
}
```

```
}
```

Цей підхід повертає текст замість порожнього масиву, що зручніше для користувача. Аналогічно, для `/stream/with-id/{id}` можна додати пояснення в тілі відповіді 404, але це вже реалізовано в попередніх змінах.

П'ята пропозиція полягає в додаванні пошуку за назвою чи описом у Video Catalog Service. Зараз каталог повертає всі записи, але користувач може захотіти знайти конкретне відео.

Новий ендпоінт `/video-info/search` виглядає так:

```
@GetMapping("/search")
public ResponseEntity<List<VideoInfo>> searchVideos
(@RequestParam("query") String query) {
    List<VideoInfo> results =
        repository.findByNameContainingOrDescriptionContaining(query, query);
    return ResponseEntity.ok(results);
}
```

Цей код додає базовий пошук, покращуючи зручність.

Система підходить для основних потреб: швидкодія є адекватною для локального використання, стійкість гарантована за наявності правильних даних, а комфорт застосування прийнятний для нескладних ситуацій. Рекомендації щодо розбиття на сторінки, підвантаження частинами, синхронізації, інформативності та пошуку додадуть платформі більшої гнучкості та готовності до роботи за реальних умов.

Висновки до розділу 3

Таким чином, було розглянуто тестування веб-додатку, оцінку задоволеності користувачів та рекомендації щодо покращення функціоналу. Модульні та інтеграційні випробування підтвердили правильність функціонування кожного з компонентів, починаючи з Video Catalog Service і

завершуючи API Gateway. Ручні перевірки, здійснені за допомогою Postman, показали стійкість системи при виконанні основних операцій.

Дослідження досвіду користувачів продемонструвало простоту навігації в каталозі та комфорт перегляду відеоконтенту. Водночас було відзначено необхідність покращення інформативності повідомлень про виникнення помилок. Оцінка продуктивності виявила достатній рівень швидкодії та стабільності при використанні платформи локально. Однак, було підкреслено наявність обмежень у разі збільшення навантаження чи обробки великих відеофайлів.

Запропоновані шляхи вдосконалення, серед яких пагінація, синхронізація бази даних з файловою системою, а також забезпечення підтримки часткового завантаження, націлені на підвищення ефективності роботи та зручності використання. Таким чином, система демонструє свою робочу придатність як основа для подальшого вдосконалення. Вона потребує доопрацювання для реального застосування, але має великий потенціал для розвитку.

ВИСНОВКИ

Розробка веб-застосунку з мікросервісною архітектурою завершилась отриманням робочої системи, до складу якої входять Service Registry (Eureka Server), API Gateway, Video Catalog Service та Video Streaming Service. Ці компоненти забезпечують функціонал перегляду відеокаталогу, потокового відтворення та додавання контенту, що відповідає основним цілям проекту. Реалізація продемонструвала, що мікросервіси дозволяють будувати гнучкі системи з незалежними частинами, котрі легко оновлювати. У той же час стало зрозуміло, що координація між сервісами вимагає ретельного планування, оскільки розподіленість ускладнює налаштування.

Теоретичний аналіз показав, що мікросервісна архітектура перевищує монолітну за масштабованістю та незалежністю розробки. Практична частина підтвердила це: кожен сервіс розроблявся окремо, що полегшувало внесення змін, але вимагало чітко визначених інтерфейсів для їх взаємодії. Порівняння підходів засвідчило, що мікросервіси оптимальні для проектів з динамічними вимогами, хоча їх складність на початковому етапі може бути вищою, ніж у монолітних рішень.

Використання Spring Boot, Eureka та Spring Cloud Gateway стало основою успішної реалізації. Spring Boot прискорив створення сервісів, Eureka забезпечила пошук компонентів, а шлюз оптимізував маршрутизацію. Проте стандартні налаштування, наприклад, 30-секундне оновлення статусу в Eureka, виявились повільними для реальних умов, що вказує на необхідність додаткового налаштування.

Тестування показало стабільність у базових сценаріях: модульні тести перевірили логіку, інтеграційні — взаємодію через API, а ручні перевірки через Postman показали швидкий час відгуку. Однак з великими файлами затримки збільшувались, а помилки залишались неінформативними. Для покращення варто додати підтримку часткового завантаження та розширені повідомлення про помилки.

Користувацький досвід виявився позитивним при наявності коректних даних: каталог легко переглядати, а відтворення сумісне з браузерами. Але порожній каталог або невалідні файли не забезпечували зворотного зв'язку з користувачем. Впровадження повідомлень та синхронізація бази з файловою системою частково вирішили проблему, але потребують доопрацювання.

Швидкодія системи виявилась достатньою для локального використання, але масштабування виявило обмеження. Стабільність залежить від конфігурації, а зручність потребує уточнень. Пагінація, кешування та автоматизоване видалення недійсних записів можуть підвищити ефективність.

Проблема розбіжності між базою даних та файловою системою, коли файли видаляються, а записи лишаються, частково вирішена фоновною перевіркою. Але це вимагає покращення – наприклад, надсилання сповіщень або перехід на хмарні сховища, щоб уникнути локальних збоїв.

Система стала прикладом практичного застосування мікросервісів, корисним для навчання або розвитку відеоплатформ. Однак її залежність від локального середовища та відсутність розподіленого розгортання вказують на необхідність адаптації до реальних умов.

Для подальшого дослідження варто вивчити масштабування в хмарі. Локальні тести пройшли успішно, але розгортання через Docker та Kubernetes могло б дати більше інформації щодо масштабування додатку.

Безпека системи також потребує уваги. Без автентифікації чи шифрування вона вразлива, тому інтеграція OAuth 2.0 та TLS зробить її придатною для практичного використання.

Моніторинг та логуювання можна покращити. Базові логи фіксують помилки, але не надають повного розуміння стану. Інструменти типу Prometheus та Grafana допоможуть відстежувати продуктивність та реагувати на проблеми.

Розроблений веб-застосунок з мікросервісами став вдалим прикладом гнучкої системи, що відповідає поставленим задачам. Він продемонстрував переваги автономності, проте виявив і труднощі координації. Але все ж реалізація покращень роблять його перспективним для подальшого розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Альохін А. В., Круглик В. С. Проектування та розробка мікросервісної архітектури системи бронювання. Молодий вчений. 2020. № 1 (77). С. 24–28.
2. Білецький В. О. Мікросервісна архітектура в контексті хмарних технологій. Вісник Київського національного університету технологій та дизайну. Серія: Технічні науки. 2022. № 2 (155). С. 45–52.
3. Бойко О. В. Мікросервісна архітектура у розробці програмного забезпечення: сучасні тенденції. Вісник Національного університету "Львівська політехніка". Серія: Інформаційні системи та мережі. 2021. № 3. С. 15–22.
4. Вознюк Л. М., Кравець П. О. Застосування технологій Spring для розробки мікросервісів. Науковий вісник Чернівецького університету. Серія: Комп'ютерні системи та компоненти. 2021. Т. 12, № 1. С. 33–40.
5. Гнатів Р. Я. Оптимізація продуктивності веб-додатків на основі мікросервісів. Комп'ютерні технології та інформаційні системи. 2020. № 3 (42). С. 28–34.
6. Гринишин Ю. М., Шевчук І. Б. Застосування Spring Boot для створення мікросервісів у веб-додатках. Наукові записки Ужгородського університету. Серія: Технічні науки. 2022. № 1 (45). С. 102–108.
7. Дубовий О. С. Контейнеризація як основа розгортання мікросервісних систем. Вісник Національного технічного університету України "КПІ". Серія: Інформатика, управління та обчислювальна техніка. 2023. № 1 (78). С. 67–73.
8. Жмуцька Н. С. Побудова веб-додатків на основі мікросервісної архітектури. Матеріали XLVIII науково-технічної конференції підрозділів ВНТУ, Вінниця, 18–22 березня 2019 р. Вінниця: ВНТУ, 2019. С. 245–249.
9. Коваль С. М., Литвиненко А. В. Аналіз інструментів для тестування мікросервісних архітектур. Інформаційні технології та комп'ютерна інженерія. 2022. № 1 (50). С. 12–19.

10. Левицький В. Р. Проектування розподілених систем із використанням REST API. Наукові записки Тернопільського національного технічного університету. 2021. № 2 (66). С. 89–95.
11. Мельник І. П. Мікросервіси у веб-розробці: виклики та перспективи. Вісник Хмельницького національного університету. Технічні науки. 2020. № 5 (287). С. 76–82.
12. Олійник Ю. В. Використання Eureka для координації мікросервісів у веб-додатках. Науковий журнал "Комп'ютерні системи та мережі". 2023. № 2. С. 41–47.
13. Придибайло О. Б., Придибайло Р. В. Мікросервісний підхід до розподілених архітектур додатків. Науковий вісник Ужгородського університету. Серія: Технічні науки. 2019. № 2 (44). С. 39–42.
14. Сеньків М. І. Аналіз ефективності мікросервісної архітектури для веб-додатків. Комп'ютерні системи та мережі. 2023. № 1. С. 50–56.
15. Степаненко О. Г., Григорчук М. І. Технології маршрутизації в мікросервісних системах. Вісник Одеського національного університету. Серія: Технічні науки. 2022. Т. 27, № 1. С. 55–62.
16. Шевчук Л. В. Оптимізація взаємодії мікросервісів у розподілених системах. Вісник Хмельницького національного університету. Технічні науки. 2021. № 4. С. 87–93.
17. Яремчук С. А. Побудова гнучких веб-додатків на основі мікросервісної архітектури. Технічні науки та технології. 2021. № 3 (25). С. 103–110.
18. Bondi A. Characteristics of Scalability and Performance in Distributed Systems. ACM Queue. 2000. Vol. 1, No. 3. P. 45–52.
19. Davis C. Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019. 432 p.
20. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003. 560 p.

21. Hightower K., Burns B., Beda J. Kubernetes Up & Running: Dive into the Future of Infrastructure. 2nd ed. O'Reilly Media, 2019. 678 p.
22. Microsoft Azure. Microservices Architecture on Azure. Microsoft Learn, 2023. URL: <https://learn.microsoft.com/en-us/azure/architecture/microservices> (дата звернення: 10.04.2025).
23. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 616 p.
24. Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, 2019. 272 p.
25. Nginx, Inc. Microservices Reference Architecture. 2020. URL: <https://www.nginx.com/resources/library/microservices-reference-architecture> (дата звернення: 10.04.2025).
26. Nygard M. Release It! Design and Deploy Production-Ready Software. 2nd ed. O'Reilly Media, 2018. 376 p.
27. Pautasso C., Zimmermann O., Amundsen M. Microservices: Yesterday, Today, and Tomorrow. In: Present and Ulterior Software Engineering. Springer, 2017. P. 129–146.
28. Red Hat. Microservices Architecture Guide. 2022. URL: <https://www.redhat.com/en/topics/microservices> (дата звернення: 10.04.2025).
29. Richardson C. Microservices Patterns: With Examples in Java. Manning Publications, 2018. 520 p.
30. Spring. Spring Boot Documentation. 2023. URL: <https://docs.spring.io/spring-boot/docs/current/reference/html> (дата звернення: 10.04.2025).
31. Spring. Spring Cloud Documentation. 2023. URL: <https://spring.io/projects/spring-cloud> (дата звернення: 10.04.2025).
32. ThoughtWorks. Technology Radar: Microservices. 2023. URL: <https://www.thoughtworks.com/radar/techniques/microservices> (дата звернення: 10.04.2025).

ДОДАТКИ

Додаток А

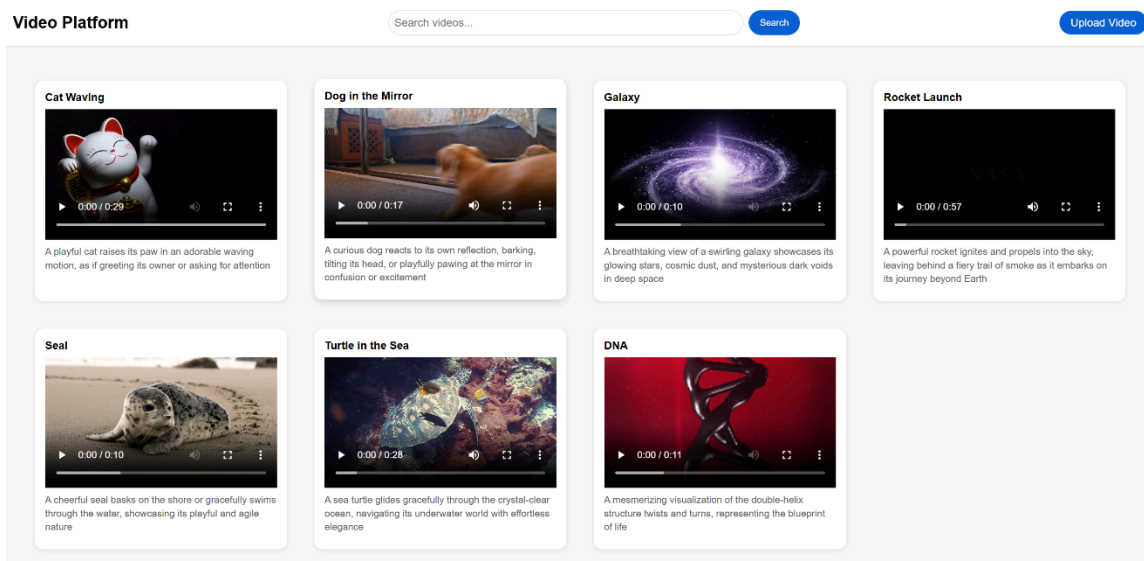


Рис. А.1. Приклад інтерфейсу веб-додатку для потокового відтворення відеоконтенту

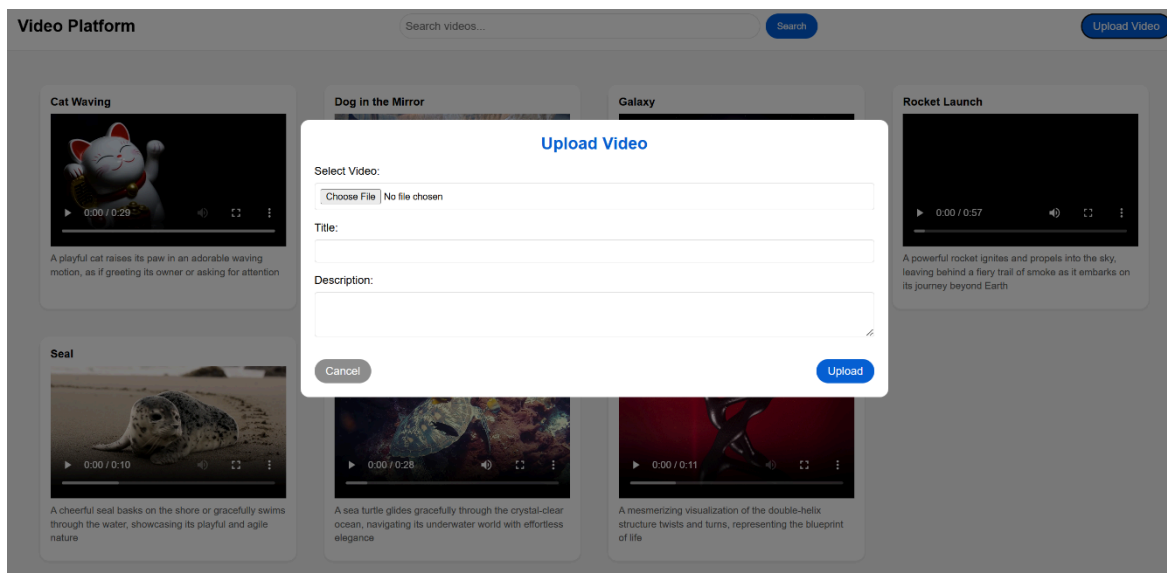


Рис. А.2. Приклад форми для додавання нового відео у веб-додатку для потокового відтворення відеоконтенту