Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Стримінговий сервіс «Anifusion»

(тема)

Виконав:
здобувач 3 року навчання,
групи КІУКІу-22-2
Олексій МЕЩАНІНОВ
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Віталій СІТНІКОВ
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Мещанінову Олексію Вікторовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Стримінговий сервіс «Anifusion»

затверджена наказом по університету від “ 26 ” травня 2025 р. № 425 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 14 липня 2025 р.

3. Вхідні дані до роботи 1) Python як серверна мова програмування;
2) React для створення клієнтської частини;
3) REST API для взаємодії клієнта і сервера;
4) Docker для налаштування інфраструктури;
5) AWS для розгортання та сховища медіа даних;

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз предметної області;
2) аналіз та вибір технологій для розробки вебзастосунку;
3) опис програмної реалізації вебзастосунку;
4) інструкція користувача;
5) висновки;

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 11 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	01.02.25-01.03.25	
2	Вибір технології розробки та	02.03.25-06.03.25	
	інструментальних засобів		
3	Розробка алгоритмічного забезпечення	06.03.25-18.04.25	
4	Розробка сервісів	19.04.25-20.05.25	
5	Відлагодження програмних модулів	21.05.25-22.05.25	
6	Оформлення матеріалів кваліфікаційної роботи	22.05.25-10.06.25	
7	Подання кваліфікаційної роботи керівникові та	17.06.25-20.06.25	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	22.06.25-25.06.25	
	вання		

Дата видачі завдання " 09 " червня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Віталій СІТНІКОВ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 72 с., 14 рис., 1 дод., 10 джерел.

FASTAPI, SQlALCHEMY, POSTGRESQL, REDIS, RABBITMQ, DOCKER, AWS, MINIO, JWT, REACT, BOOTSTRAP, HLS.

Метою кваліфікаційної роботи є створення повноцінного стримінгового вебсервісу для перегляду фільмів і серіалів, який задовольняє сучасні вимоги до доступності, персоналізації, інтерактивності та масштабованості. Проєкт спрямований на реалізацію системи з чітко розмежованими мікросервісами та зручним користувацьким інтерфейсом.

У межах роботи було реалізовано мікросервісну архітектуру з використанням FastAPI, PostgreSQL, Docker, RabbitMQ та MinIO. Серверна частина включає сервіси авторизації (JWT), потокової трансляції відео, керування підписками, коментарями й обліковими записами. Обробка медіаконтенту виконується окремим сервісом, що забезпечує масштабованість і розподіл навантаження.

Клієнтський застосунок створено на основі React із сучасним та адаптивним інтерфейсом. Реалізовано можливість вибору мови аудіодоріжки, взаємодії з контентом, підписки на преміум-функції, а також зручну систему навігації. Хмарна архітектура дозволяє обробляти запити з високою швидкістю, забезпечуючи надійність і стабільність роботи сервісу.

Сервіс має потенціал для подальшого розвитку та впровадження персоналізованих рекомендацій, що сприятиме залученню нової аудиторії та підвищенню конкурентоспроможності.

ABSTRACT

Bachelor’s thesis: 72 pages, 14 figures, 1 appendix, 10 sources.

FASTAPI, SQlALCHEMY, POSTGRESQL, REDIS, RABBITMQ, DOCKER, AWS, MINIO, JWT, REACT, BOOTSTRAP, HLS.

The goal of this qualification project is to create a full-fledged streaming web service for watching movies and TV shows that meets modern requirements for accessibility, personalization, interactivity, and scalability. The project is focused on implementing a system with clearly separated microservices and a user-friendly interface.

As part of the work, a microservice architecture was implemented using FastAPI, PostgreSQL, Docker, RabbitMQ, and MinIO. The backend includes services for authorization (JWT), video streaming, subscription management, comments, and user accounts. Media content is processed by a separate service, ensuring scalability and load distribution.

The client application is built with React and features a modern, responsive interface. It supports audio track language selection, content interaction, premium feature subscriptions, and a convenient navigation system. The cloud-based architecture allows for high-speed request processing, ensuring reliability and service stability.

The platform has potential for further development, including the implementation of personalized recommendations, which will help attract a wider audience and increase competitiveness.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

СУБД – система управління базами даних

API – інтерфейс прикладного програмування (англ., Application Programming Interface)

CRUD – базовий набір операцій для роботи з даними в інформаційних системах (GET, POST, PUT, DELETE)

EC2 – сервіс в AWS для запуску віртуальних серверів (англ., Elastic Compute Cloud)

gRPC – фреймворк для віддаленого виклику процедур із використанням протоколу HTTP/2 і формату protobuf (англ., Google Remote Procedure Call)

AWS – хмарна платформа Amazon Web Services, що надає різноманітні сервіси для обчислень, зберігання, баз даних тощо

HLS – потоковий протокол для трансляції аудіо та відео контенту (англ., HTTP Live Streaming)

REST – архітектурний стиль для створення вебсервісів, що використовує HTTP-протокол та прості операції (GET, POST, PUT, DELETE)

HTTP – протокол передачі гіпертексту, основа для обміну інформацією у Всесвітній мережі (англ., HyperText Transfer Protocol)

JWT – стандартизований формат токенів авторизації (англ., JSON Web Token)

S3 – сервіс Amazon Simple Storage Service для об’єктного зберігання даних у хмарі

SES – сервіс Amazon Simple Email Service для масової та транзакційної відправки електронної пошти

SSL/TLS – протоколи для забезпечення безпечного шифрованого зв’язку в мережі (англ., Secure Sockets Layer / Transport Layer Security)

UI – інтерфейс користувача (англ., User Interface)

UX – досвід користувача (англ., User Experience)

ВСТУП

Зі створенням таких сервісів народжується ще одна важлива можливість – підтримка національного кінематографа. Власна стримінгова платформа може стати вікном у світ для українських режисерів, акторів, сценаристів, які нині часто залишаються в тіні західного контенту. Локальні сервіси – це шанс повернути глядача до власної культури, до свого – рідного, зрозумілого, близького.

Особливе значення такі проєкти мають в умовах війни. Вони здатні не лише розважати, а й надихати, підтримувати моральний дух, об’єднувати суспільство через спільні цінності та історії. Культурний фронт, як і інформаційний, сьогодні не менш важливий. А технології – це зброя, яку ми можемо використати для створення якісного, чесного й сучасного продукту.

Крім того, такі ініціативи відкривають нові горизонти для українського IT-сектору. Молоді розробники, студенти, ентузіасти мають змогу втілювати власні ідеї, реалізовувати креативні підходи, вчитися працювати з реальними технологіями – від фронтенду до складної логіки обробки даних. Кожен рядок коду в подібному проєкті – це не просто технічне завдання, а частина великої картини.

У технічному плані створення вебсервісу для перегляду відео – це ще й серйозний виклик. Потрібно продумати ефективну архітектуру, забезпечити масштабованість, оптимізувати швидкість завантаження та якість відео, реалізувати захист від піратства. Це вимагає глибоких знань, уважності до деталей і, звісно, любові до справи.

Окрему увагу варто приділити досвіду користувача (UX/UI). Саме від зручності інтерфейсу, логіки навігації, швидкості пошуку залежить, чи залишиться глядач на платформі, чи натомість повернеться до звичних Netflix або YouTube. А отже, завдання полягає не тільки в тому, щоб технічно реалізувати платформу, а й у тому, щоб зробити її по-справжньому зручною, цікавою та естетично привабливою.

Не менш важливою є й аналітика. Завдяки сучасним інструментам можна відстежувати, що саме дивляться користувачі, як довго, на яких пристроях. Ці дані дозволяють не тільки краще розуміти свою аудиторію, а й будувати прогнози, покращувати контент, персоналізувати пропозиції. Технології на службі у контенту – ось нова формула успіху.

Підсумовуючи, можна сказати, що створення стримінгового сервісу – це не просто кваліфікаційна робота чи навчальний проєкт. Це приклад того, як ідеї перетворюються на реальні речі, які можуть змінити світ навколо. Це шанс довести, що українські розробники здатні створювати конкурентоспроможні продукти, орієнтовані на майбутнє.

Такі ініціативи формують нову цифрову реальність, де комфорт користувача, доступ до якісного контенту та національна ідентичність можуть гармонійно поєднуватися. І якщо навіть один глядач після важкого дня знайде тут свій фільм і кілька хвилин спокою – значить, усе це недарма.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Ідея створення сервісу стримінгу

Історія стримінгових платформ тісно пов’язана з глобальними змінами, які сколихнули світ на початку 2020 року. Пандемія COVID-19 стала безпрецедентним викликом не лише для медицини чи економіки, а й для культурної індустрії. З початком карантинів кінотеатри по всьому світу змушені були зачинити свої двері. Прем'єри фільмів переносилися, фестивалі скасовувалися, глядачі залишилися вдома – без можливості насолоджуватись кіномистецтвом на великому екрані.

Для багатьох кінокомпаній це стало справжнім ударом. Витративши мільйони доларів на виробництво стрічок, вони стикнулися з відсутністю каналів їх дистрибуції. Доходи стрімко падали, а витрати залишалися. У цій критичній ситуації індустрія змушена була шукати нові шляхи взаємодії з глядачем. І саме тут з’явилась ідея цифрового прокату – швидкого, зручного, безпечного.

Так почалася нова ера онлайн-платформ. Деякі компанії, як-от Disney, Universal чи Warner Bros, пішли ще далі – вони почали одразу випускати фільми у стримінгових сервісах паралельно або навіть замість традиційного показу в кінотеатрах. Так виник формат підписки: коли користувач щомісяця сплачує фіксовану суму й отримує доступ до величезної бібліотеки контенту. Це стало вигідним і для споживача, і для компаній, які змогли стабілізувати прибутки навіть у найважчі часи.

Модель підписки (subscription-based model) виявилася зручною, гнучкою та передбачуваною. Люди, які вимушено сиділи вдома, почали активно користуватись новими цифровими сервісами. А компанії – розвивати інтерфейси, рекомендаційні алгоритми, розширювати каталоги, дублювати контент кількома мовами, включаючи українську.

Цікаво, що саме пандемія, попри весь її трагізм, стала каталізатором

цифрової трансформації у сфері розваг. Відбулася своєрідна "демократизація кіно": фільми стали ближчими, доступнішими, різноманітнішими. Не треба купувати дорогий квиток чи витрачати час на дорогу – усе вміщається в екрані смартфона або ноутбука.

Саме тоді багато розробників, стартапів і навіть студентів почали задумуватись над створенням власних відеоплатформ. Так виникла ідея даного проєкту – створити український вебзастосунок, який не лише відповідає вимогам сучасного ринку, а й враховує національні особливості, мовні уподобання, культурні запити. Бо хто, як не ми, може найкраще зрозуміти потреби українського глядача?

1.2 Порівняння існуючих платформ

Стримінг відео став ключовою технологією сучасного цифрового світу, забезпечуючи швидкий та зручний доступ до відеоконтенту без необхідності його попереднього завантаження. У сучасних умовах, коли користувачі очікують миттєвого доступу до високоякісного контенту, роль стримінгу стає ще більш значущою. Цей підхід повністю змінив уявлення про споживання відео: тепер, аби подивитись фільм чи серіал, достатньо лише смартфона й стабільного інтернет-з’єднання.

Сьогодні у світі існує десятки, якщо не сотні платформ, які базуються на стримінгових технологіях. Найбільш відомими серед них є Netflix, Hulu, Amazon Prime Video, Disney+, HBO Max. Кожна з них має свою стратегію, унікальний підхід до аудиторії та власний каталог ексклюзивного контенту. Саме конкуренція між цими гігантами і стала рушієм безперервного розвитку індустрії – з’явилися серіали з бюджетами, що перевищують повнометражні фільми, інноваційні підходи до зйомки, інтерактивні епізоди, персоналізовані рекомендації.

Першим цікавим севісом з перегляду фільмів та серівлів став "Netflix", Зовнішній вигляд застосунку можна побачити на рисунку 1.1.

Рисунок 1.1 – Сервіс Netflix

В якості переваг Netflix, безсумнівно, варто відзначити те, що ця платформа вже давно закріпила за собою статус одного з найвпливовіших і найпопулярніших стримінгових сервісів у світі. Її знають, їй довіряють, її відкривають мільйони користувачів щодня. Одним із головних козирів Netflix є широкий вибір контенту – від класичних фільмів до ексклюзивних оригінальних проєктів, які неможливо знайти на інших платформах. Саме власне виробництво (Netflix Originals) стало візитівкою сервісу. Такі серіали, як "Stranger Things", "The Crown", "Narcos" або "Wednesday" стали справжніми культурними явищами.

Ще однією сильною стороною сервісу є якісний багатомовний дубляж і субтитри. Платформа дбає про локалізацію, що дозволяє мільйонам людей у різних куточках світу комфортно споживати контент рідною мовою. Український дубляж також поступово з’являється, що є позитивним сигналом у контексті підтримки національної ідентичності в цифровому просторі.

Netflix також пропонує зручний інтерфейс, гнучку систему рекомендацій та кросплатформність – користувач може почати дивитись

серіал на телевізорі, продовжити на планшеті, а завершити на смартфоні дорогою на роботу.

Однак, попри вражаючу бібліотеку, у Netflix є й певні недоліки. Один із найпомітніших – обмеження за ліцензіями. Через авторські права частина контенту доступна лише в окремих регіонах, що викликає розчарування у користувачів. Наприклад, фільм може бути доступним у США, але недоступним для українських користувачів – або ж навпаки. Крім того, деякі культові стрічки або серіали, які належать іншим студіям, часто відсутні на платформі, оскільки їх розміщено на конкурентних сервісах – Disney+, HBO Max чи Amazon Prime.

До того ж останнім часом компанія почала активно боротися з розповсюдженням паролів між користувачами, що викликало хвилю критики. Багато глядачів вважають, що нова політика щодо обмеження спільного доступу до акаунтів зменшує зручність користування сервісом, особливо для сімей або друзів, які живуть окремо. Таким чином, хоча Netflix і є флагманом у світі стримінгу, його модель не позбавлена викликів. Аудиторія стає вибагливішою, конкуренція зростає, і навіть найуспішніші гравці мають постійно адаптуватися до змін цифрової епохи.

Другим цікавим сервісом було обрано "Amazon Prime Video" від компанії Amazon (рисунок 1.2). Amazon Prime Video – ще один потужний гравець на світовому ринку стримінгових сервісів, який упевнено тримає свою позицію поряд із такими гігантами, як Netflix та Disney+. Від початку свого існування платформа будувалася як частина глобальної екосистеми Amazon, тому доступ до відеосервісу часто є бонусом для передплатників Prime. Це вже є важливою перевагою – користувач, який оформив одну підписку, отримує не лише доступ до фільмів, а й низку додаткових сервісів, таких як швидка доставка товарів, електронні книги та музика.

Однією з головних сильних сторін Amazon Prime Video є широкий вибір контенту. Тут можна знайти як класичні фільми, так і нові релізи, авторське кіно, документалістику та культові серіали. Платформа активно

інвестує у власне виробництво (Amazon Originals), зокрема створює якісний контент із глибоким змістом, який не завжди отримує достатню увагу на інших сервісах. Прикладом може слугувати серіал "The Marvelous Mrs. Maisel" або блокбастерний "The Lord of the Rings: The Rings of Power", на який було витрачено рекордний бюджет.

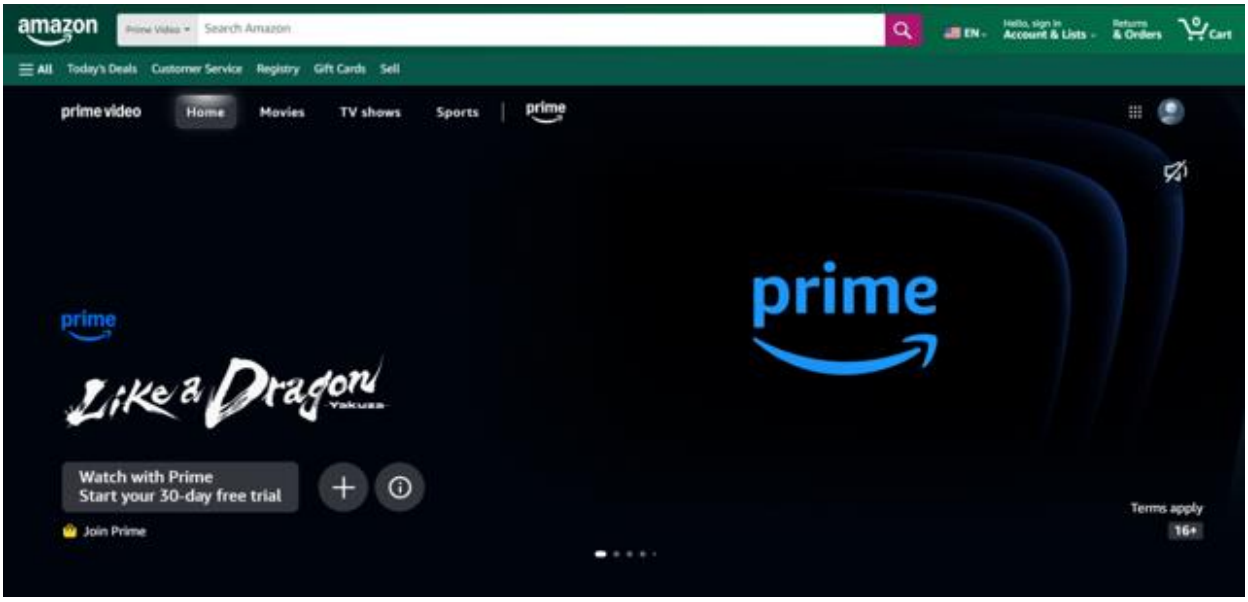

Рисунок 1.2 – Сервіс Amazon Prime Video

Ще однією перевагою Amazon є велика географія покриття. Сервіс доступний у багатьох країнах, що відкриває його контент для глобальної аудиторії. Крім того, він часто включає можливість купівлі або оренди окремих фільмів, яких немає в підписці – це дає користувачам додаткову гнучкість. Втім, платформа не позбавлена недоліків. Одним з основних є обмежена кількість мовних локалізацій. Для багатьох користувачів з країн, де англійська не є рідною, це створює бар'єри – адже відсутність дубляжу чи субтитрів у потрібній мові ускладнює перегляд. Український дубляж або субтитри, наприклад, представлені лише частково, що обмежує привабливість сервісу для вітчизняного глядача.

Ще один момент, який часто викликає критику – це інтерфейс платформи. Дизайн виглядає дещо застарілим, а структура навігації – перевантаженою. Знайти потрібний фільм чи серіал буває непросто, особливо

для нових користувачів. Розділення безкоштовного та платного контенту не завжди очевидне, що може викликати плутанину.

У підсумку, Amazon Prime Video – це потужний, багатофункціональний сервіс, який заслуговує на увагу. Але щоб залишатися конкурентоспроможним, він потребує вдосконалення: покращення інтерфейсу, розширення мовної підтримки та чіткішої комунікації щодо умов користування. У динамічному світі цифрових технологій навіть гіганти не можуть дозволити собі зупинятися на досягнутому.

Третій сервіс "Crunchroll" від компанії Amazon (рисунок 1.3) дозволяє переглядати анімаційні серіали та фільми.

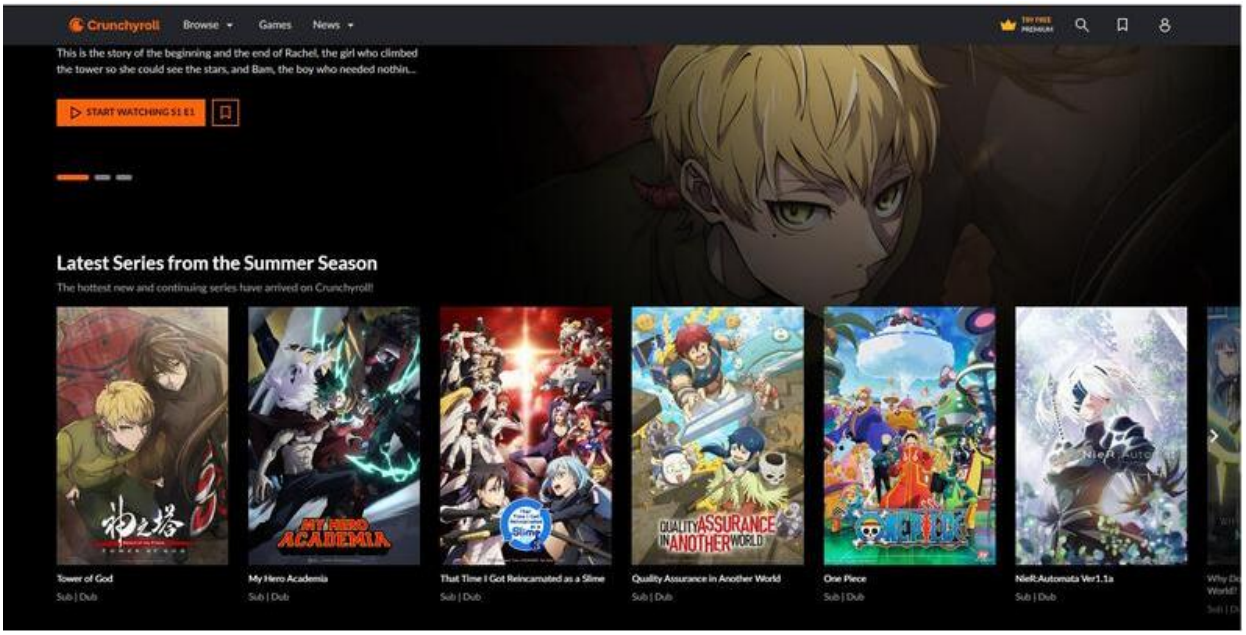

Рисунок 1.3 – Сервіс Crunchroll

Crunchyroll – це справжній рай для поціновувачів аніме. Якщо інші стримінгові платформи лише періодично додають окремі анімаційні стрічки до своєї бібліотеки, то Crunchyroll зробив ставку саме на цей жанр. І не прогадав: мільйони фанатів з усього світу обирають саме його для перегляду як новинок, так і класики японської анімації. Платформа стала важливою культурною точкою перетину – тут можна знайти унікальні проєкти, що не потрапляють у поле зору глядачів на інших сервісах. Наприклад, десятки сезонів культових франшиз або ексклюзивні трансляції серіалів одразу після

прем’єри в Японії.

До переваг Crunchyroll безумовно можна віднести величезний каталог аніме, постійне оновлення контенту, а також активну спільноту користувачів, яка створює рецензії, рейтинги та рекомендації. Це не просто відеосервіс, а частина великої фан-культури, яка об’єднує глядачів за інтересами. Попри свою популярність серед фанатів, сервіс має і слабкі сторони. Однією з основних проблем залишається мовне питання. Значна частина контенту доступна виключно японською мовою – або з англійськими субтитрами, що автоматично ускладнює перегляд для глядачів, які не володіють цими мовами. Українська локалізація – радше виняток, ніж правило.

Крім того, Crunchyroll рідко надає повноцінний дубляж. Для багатьох користувачів, зокрема молодшої аудиторії чи тих, хто звик до контенту в озвучці, це може стати суттєвою перешкодою. Ще однією проблемою є технічні обмеження: інтерфейс платформи не завжди інтуїтивно зрозумілий, а якість стримінгу залежить від регіону та швидкості з’єднання.

На основі аналізу існуючих стримінгових сервісів, можна виокремити кілька спільних недоліків, які вказують на потребу створення нових, адаптованих до різних аудиторій платформ:

- відсутність повноцінної підтримки багатьох мов;
- обмежений або повністю відсутній професійний дубляж;
- низький відсоток анімаційного контенту, окрім спеціалізованих сервісів;
- слабка персоналізація інтерфейсу для глядачів з різних культурних середовищ.

Ці обмеження відкривають широке поле для розробників. У світі, де глядач стає дедалі вибагливішим, а культурна різноманітність набуває особливої ваги, є актуальним створення стримінгового продукту з підтримкою локальних мов, зручним інтерфейсом та мультимедійним каталогом, який не тільки розважає, а й формує нові смисли в цифровому просторі.

2 ОГЛЯД ТЕХНОЛОГІЙ, ЯКІ ВИКОРИСТВУЮТЬСЯ ПРИ РОЗРОБЦІ СЕРВІСУ

2.1 Мова програмування Python

З розвитком технологій та появою нових задач, які вимагали простоти та гнучкості в розробці, у 1991 році з’явилася мова програмування Python. Її створив голландський програміст Гвідо ван Россум як інструмент, що поєднує простоту навчання з широкими можливостями для практичного використання. Python став інструментом, що змінив підходи до вирішення складних задач у програмуванні.

Назва Python походить не від змії, як можна подумати, а від британського комедійного шоу "Monty Python's Flying Circus". Гвідо ван Россум прагнув створити інструмент, що буде не лише ефективним, а й приємним у використанні.

Python – це не просто мова, це філософія. Її принципи, закладені в документі PEP 20 (The Zen of Python), наголошують на простоті, читабельності та відкритості. Це своєрідний універсальний інструмент у наборі програміста, що дозволяє створювати проєкти будь-якого рівня складності: від невеликих скриптів до великих корпоративних систем.

Перша версія Python з'явилася в 1991 році, але справжня популярність прийшла з виходом Python 2 у 2000 році. Основною відмінністю цієї версії стала підтримка Unicode, що значно розширило можливості роботи з текстовими даними різними мовами. У 2008 році була представлена третя версія мови – Python 3, яка внесла значні зміни в синтаксис та функціонал, що зробило мову ще більш потужною та адаптованою до сучасних потреб.

Python є високорівневою мовою програмування з динамічною типізацією, що робить її зрозумілою навіть для новачків. Її синтаксис дозволяє писати код, який виглядає майже як звичайна англійська мова, що суттєво полегшує розуміння логіки програми. Наприклад, замість

використання складних конструкцій, Python пропонує лаконічні рішення для роботи з колекціями, потоками даних та асинхронними процесами.

Python підтримує кілька парадигм програмування, включаючи:

- об'єктно-орієнтоване програмування (ООП) – створення структурованих програм з використанням класів та об'єктів;
- функціональне програмування – робота з функціями як об'єктами;
- імперативне програмування – можливість писати прості послідовності команд.

Python став незамінним інструментом у різних сферах:

- веброзробка: завдяки популярним фреймворкам, таким як Django, Flask та FastAPI, Python забезпечує створення масштабованих та безпечних вебзастосунків. Ці фреймворки спрощують роботу з базами даних, маршрутизацією запитів та управлінням користувачами;
- наука про дані та машинне навчання: бібліотеки, як-от NumPy, Pandas, TensorFlow, Scikit-learn, зробили Python стандартом для аналізу даних, створення моделей машинного навчання та роботи зі штучним інтелектом;
- автоматизація: Python активно використовується для автоматизації рутинних завдань, як-от обробка файлів, робота з везастосунками чи тестування програмного забезпечення;
- ігрова розробка: попри те, що Python не є основною мовою для створення ігор, бібліотеки Pygame та Panda3D дозволяють створювати захоплюючі 2D- та 3D-ігри;
- системне адміністрування: Python допомагає автоматизувати налаштування серверів, моніторинг системи та управління ресурсами.

Перевагами мови Python перед іншими мовами можливо зазначити:

- читабельність: код на Python легко зрозуміти навіть без глибокого знання мови;
- широку екосистему: величезна кількість бібліотек та модулів дозволяє швидко почати роботу над будь-яким проєктом;

- кросплатформеність: Python працює на всіх популярних операційних системах: Windows, macOS, Linux;
- велику спільноту: мільйони розробників по всьому світу підтримують Python, створюючи нові бібліотеки, інструменти та навчальні матеріали.

2.2 Фреймворк FastAPI

Для реалізації серверної частини вебсервісу було обрано фреймворк FastAPI – сучасний інструмент розробки API-застосунків на мові програмування Python [1]. Даний фреймворк активно набирає популярність завдяки поєднанню високої продуктивності, зручності використання та підтримки сучасних підходів до веброзробки, таких як асинхронність, типізація та автоматична генерація документації.

Однією з основних переваг FastAPI є його висока швидкодія, що досягається за рахунок використання асинхронної архітектури, побудованої на базі бібліотеки Starlette. Це особливо актуально у контексті розробки стримінгового сервісу, де очікується велика кількість одночасних запитів до серверу.

Крім того, FastAPI забезпечує автоматичну генерацію документації API у форматах Swagger UI та ReDoc, що значно спрощує процес тестування та взаємодії з клієнтською частиною застосунку. Завдяки інтеграції з бібліотекою Pydantic, фреймворк підтримує вбудовану валідацію та серіалізацію даних, що підвищує надійність програмного коду та знижує ризик помилок під час обробки запитів.

FastAPI також повністю підтримує асинхронне програмування (async/await), що дозволяє створювати масштабовані застосунки з високою продуктивністю. У випадку стримінгового сервісу це дає змогу ефективно реалізувати механізми пошуку, авторизації, персоналізованих рекомендацій та обробки користувацьких запитів у реальному часі.

До основних переваг використання FastAPI можна віднести:

- підтримку сучасного стандарту OpenAPI для опису структури API;
- можливість інтеграції з базами даних (PostgreSQL, MongoDB);
- легке масштабування сервісу;
- зручність налаштування системи автентифікації та авторизації;
- читабельний та структурований код з чіткою типізацією.

Фреймворк також демонструє високу ефективність у тестуванні. Його архітектура дозволяє легко реалізовувати модульні та інтеграційні тести, що є важливим фактором для розробки стабільного програмного продукту.

Таким чином, вибір FastAPI як основного фреймворку для реалізації серверної частини стримінгового вебзастосунку є технічно обґрунтованим та відповідає сучасним вимогам до продуктивності, безпеки та зручності масштабування.

2.3 Мікросервісна архітектура

Мікросервісна архітектура – це один із найпопулярніших підходів до розробки програмного забезпечення, який дозволяє створювати масштабовані, надійні та гнучкі системи. В основі цього підходу лежить принцип розбиття складного монолітного застосунку на набір незалежних компонентів, кожен з яких виконує окрему функцію та взаємодіє з іншими через стандартизовані інтерфейси.

Мікросервіс – це незалежний модуль програмного забезпечення, що відповідає за виконання однієї чітко визначеної бізнес-функції (рисунок 2.1). Наприклад, у системі онлайн-магазину окремі мікросервіси можуть відповідати за реєстрацію користувачів, управління кошиком покупок, оформлення замовлень або обробку платежів.

Мікросервіси розробляються з урахуванням наступних принципів:

- однофункціональність: кожен мікросервіс зосереджений на вирішенні однієї задачі;

- незалежність: сервіси працюють автономно, а їх розробка, тестування та розгортання виконуються окремо;

- децентралізація: управління даними та логікою розподілене між сервісами, що спрощує масштабування.

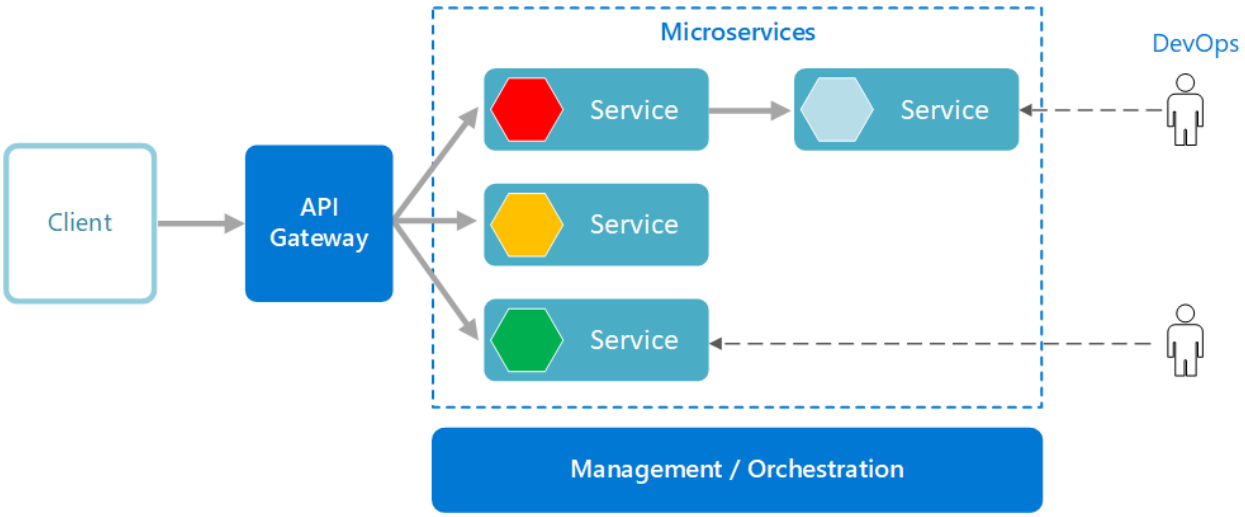

Рисунок 2.1 – Схема мікросервісної архітектури

Кожен мікросервіс у системі має власну базу даних, інтерфейс для взаємодії (наприклад, REST API або gRPC) і може бути реалізований на будь-якій мові програмування. Взаємодія між мікросервісами зазвичай здійснюється через мережні протоколи, такі як HTTP, або за допомогою систем обміну повідомленнями, наприклад RabbitMQ або Kafka.

Розподіл обов'язків між мікросервісами дозволяє досягти гнучкості в розробці. Наприклад, зміни в одному мікросервісі не впливають на роботу інших компонентів системи, що дозволяє уникнути ризиків, пов'язаних із модифікацією монолітного застосунку, де навіть незначна зміна може призвести до проблем у роботі всієї системи.

Ключовими перевагами мікросервісів є:

- масштабованість: мікросервісна архітектура дозволяє масштабувати лише ті частини системи, які зазнають високого навантаження. Наприклад, сервіс обробки замовлень в інтернет-магазині може масштабуватися окремо від сервісу управління користувачами;

- гнучкість у виборі технологій: кожен мікросервіс може бути реалізований на різних мовах програмування або з використанням різних баз даних. Наприклад, один сервіс може працювати на Python з PostgreSQL, а інший на Java з MongoDB;

- швидкість розробки: розподіл системи на незалежні компоненти дозволяє кільком командам працювати над різними мікросервісами одночасно, що прискорює випуск нових функцій;

- стійкість до помилок: збої в роботі одного мікросервісу зазвичай не впливають на функціонування інших компонентів системи. Наприклад, якщо сервіс аналітики тимчасово недоступний, основна система (наприклад, обробка замовлень) продовжує працювати;

- швидке впровадження змін: мікросервіси можна оновлювати незалежно, без необхідності розгортання всього застосунку. Це особливо корисно в умовах, коли потрібно швидко виправити помилку або додати новий функціонал.

В якості викликів та недоліків використання мікросервісної архітектури можливо зазначити:

- складність управління: велика кількість сервісів створює додаткові виклики для моніторингу, логування та відстеження залежностей, тому необхідно використовувати інструменти для оркестрації, такі як Kubernetes або Docker Swarm;

- мережні затримки: оскільки сервіси взаємодіють через мережу, можуть виникати затримки в обміні даними, що впливає на продуктивність системи;

- розподілені транзакції: якщо операція охоплює кілька сервісів, виникає потреба в управлінні транзакціями між ними, що може бути складною задачею;

- витрати на розробку та підтримку: мікросервісна архітектура вимагає більшого ресурсу для впровадження сучасної інфраструктури та навчання команди.

Для розгортання та підтримки мікросервісної архітектури зазвичай використовуються такі інструменти:

- Docker – для створення контейнерів, у яких розгортаються сервіси;
- Kubernetes – для оркестрації контейнерів та управління масштабуванням;
- Prometheus та Grafana – для моніторингу стану сервісів;
- RabbitMQ, Kafka – для організації черг повідомлень між сервісами.

2.4 Файловий стримінг

Стримінг – це спосіб передачі даних у режимі реального часу через мережу Інтернет. Він дозволяє користувачам переглядати відео, слухати музику чи грати в ігри без необхідності завантажувати весь контент на свій пристрій. Стримінг – це одна з ключових технологій сучасності, яка змінила підхід до споживання контенту. Завдяки стримінгу користувач має змогу миттєво отримувати доступ до музики, відео, ігор чи подій у будь-який час і з будь-якого місця. Попри виклики, такі як затримки чи залежність від якості інтернету, технологія продовжує вдосконалюватися та відкривати нові можливості для бізнесу та користувачів.

Стримінг базується на технології передачі даних у вигляді невеликих фрагментів (пакетів) з серверу до клієнта. Користувач починає отримувати контент одразу після початку передачі, без очікування завершення завантаження всього файлу. Це стало можливим завдяки використанню спеціальних протоколів передачі даних, таких як:

- HTTP Live Streaming (HLS);
- MPEG-DASH;
- RTMP (Real-Time Messaging Protocol).

Процес стримінгу включає:

- захоплення та обробку контенту (наприклад, відео чи аудіо);
- кодування даних для зменшення обсягу файлів;

- передачу потоків через сервери до кінцевих користувачів;
- декодування на стороні клієнта та відтворення у режимі реального часу.

Стримінг можна поділити на кілька основних категорій залежно від типу контенту та призначення:

- відеостримінг;
- музичний стримінг;
- ігровий стримінг;
- живі трансляції (Live Streaming).

Перевагами стримінгу є:

- миттєвий доступ до контенту: користувачі можуть почати перегляд чи прослуховування одразу, без завантаження файлу;
- економія місця: стримінг дозволяє відтворювати контент без його збереження на пристрої, що зменшує потребу у великому обсязі пам'яті;
- актуальність контенту: завдяки стримінгу можна транслювати події в реальному часі;
- персоналізація: багато стримінгових платформ пропонують рекомендації на основі уподобань користувача.

В якості недоліків стримінгу можливо зазначити:

- залежність від якості інтернету: для стабільного перегляду або прослуховування потрібне швидке та надійне з’єднання;
- високе навантаження на мережу: стримінг споживає значні обсяги інтернет-трафіку, що може бути проблемою для користувачів з обмеженим доступом;
- проблеми із затримками: затримка передачі даних (латентність) може бути критичною для трансляцій у реальному часі, особливо у випадках онлайн-ігор або конференцій;
- захист авторських прав: нелегальний стримінг або піратський контент є значною проблемою для індустрії.

2.5 Система управління базами даних PostgreSQL та SQLAlchemy

Кожна велика ідея потребує надійної основи. У випадку цифрового продукту такою основою є база даних – "серце" системи, яке зберігає всю інформацію: від назв фільмів до вподобань користувачів. Саме тому вибір системи управління базами даних (СУБД) – це не просто технічне рішення, а стратегічний крок, який безпосередньо впливає на стабільність, продуктивність і масштабованість застосунку.

Для реалізації проєкту було обрано PostgreSQL – одну з найпотужніших СУБД з відкритим кодом, яка вже давно зарекомендувала себе як надійне та ефективне рішення для великих та складних проєктів. PostgreSQL відома своєю стабільністю, гнучкістю та підтримкою складних запитів. Саме тому її обирають такі техногіганти, як Apple, Instagram, Spotify. Це рішення – не лише про зберігання даних, а про довіру до перевірених інструментів.

Але що таке сучасна база даних без зручного засобу взаємодії з нею? Тут на сцену виходить SQLAlchemy – бібліотека для Python, яка перетворює сухі SQL-запити у живу, об’єктно-орієнтовану мову програмування. Завдяки SQLAlchemy ми можемо працювати з базою даних як з набором логічних об’єктів – створювати, змінювати, пов’язувати між собою таблиці у вигляді моделей, як це роблять сучасні інженери програмного забезпечення.

Іншими словами, SQLAlchemy дозволяє розробнику зосередитися не на синтаксисі запитів, а на логіці застосунку. Вона бере на себе рутинну частину, автоматизує зв’язки, перевіряє типи, попереджає помилки. Завдяки цьому реалізація навіть складних функцій, як-от рекомендацій на основі історії перегляду, стає значно простішою.

У парі PostgreSQL та SQLAlchemy працюють як злагоджений механізм: один відповідає за швидкість та надійність зберігання даних, інший – за зручність та гнучкість доступу до них. Це технологічний дует, який ідеально підходить для створення сучасного стримінгового сервісу, де дані – не

просто цифри, а джерело розуміння користувачів, їхніх смаків та очікувань.

Завдяки цим інструментам було реалізовано повноцінну систему зберігання інформації про фільми, серіали, користувачів, оцінки, історію перегляду та персональні рекомендації. І все це – з надією, що технічне рішення стане не лише надійним фундаментом, а й інструментом створення якісного досвіду користувача.

2.6 Система обміну повідомленнями RabbitMQ

У світі, де швидкість реакції системи на події визначає її якість, асинхронна взаємодія між компонентами застосунку стає не просто бажаною – вона є критичною. Адже в уявному цифровому театрі подій усе має працювати злагоджено, без затримок та зависань. Саме в таких умовах на сцену виходить RabbitMQ – потужна система обміну повідомленнями, яка бере на себе роль невидимого диригента.

RabbitMQ працює за принципом "черги повідомлень": один компонент (наприклад, вебзастосунок) надсилає повідомлення в чергу, а інший (наприклад, фоновий процес обробки даних або сервіс рекомендацій) зчитує це повідомлення, коли буде готовий. Таким чином забезпечується асинхронність – процеси більше не блокують одне одного, а працюють незалежно, в ідеальному ритмі цифрового оркестру.

Це особливо важливо для стримінгових сервісів. Уявімо ситуацію: користувач натискає кнопку "Додати до списку перегляду" або "Поставити оцінку фільму". Замість того щоб одразу відправляти складні запити до бази даних або запускати механізм оновлення рекомендацій, система делегує це завдання RabbitMQ. Повідомлення надходить у чергу, а система продовжує працювати миттєво – користувач навіть не помічає жодної затримки.

RabbitMQ підтримує гнучку маршрутизацію повідомлень, масштабування, затримку обробки, а також збереження черг навіть у випадку тимчасового збою одного з сервісів. Іншими словами, ця система гарантує,

що жодне повідомлення не буде втрачене, а кожне завдання буде виконане – вчасно та коректно.

У проєкті, що розроблявся в межах цієї кваліфікаційної роботи, RabbitMQ виконує ключову роль у реалізації фонової логіки, зокрема:

- запис та обробка даних перегляду;
- генерація персональних рекомендацій на основі активності користувача;
- надсилання повідомлень про новинки або зміни в списку контенту.

Завдяки цьому технологічному рішенню вдалося розвантажити основний сервер, зробити систему масштабованою та гнучкою, а також забезпечити безперервний користувацький досвід.

RabbitMQ – це приклад того, як грамотне використання сучасних технологій здатне зробити сервіс не просто функціональним, а по-справжньому розумним, живим та чуйним до потреб кожного користувача.

2.7 Фреймворк React.js

Світ змінюється – змінюється й спосіб, у який ми взаємодіємо з цифровими платформами. Ще не так давно користувачі звикали до повільних оновлень сторінок, перезавантажень та громіздких вебінтерфейсів. Сьогодні ж інтерфейс повинен бути не просто швидким – він має бути інтуїтивним, живим, чуйним. І саме тут свою роль відіграє React.js – бібліотека, яка змінила правила гри.

React.js, розроблений у надрах компанії Facebook (тепер Meta), став одним з найпопулярніших інструментів для створення інтерфейсів користувача у вебзастосунках. Його основна ідея проста та геніальна водночас: розбити інтерфейс на незалежні, повторно використовувані компоненти, кожен з яких відповідає за свою частину сторінки. Це дозволяє створювати складні інтерфейси швидко, зручно і, головне, ефективно.

У контексті стримінгового сервісу React.js забезпечує саме той рівень

інтерактивності, якого очікує сучасний користувач: миттєвий пошук фільмів, плавна навігація між сторінками, оновлення рекомендацій без перезавантаження сайту. Все це – результат реактивного підходу, де зміни в даних автоматично оновлюють вигляд інтерфейсу. І користувач цього навіть не помічає – йому просто зручно.

Ще одна перевага React – можливість створення SPA (Single Page Application), тобто односторінкових застосунків. Завдяки цьому весь процес взаємодії з сервісом відбувається на одній сторінці, без дратівливих перезавантажень та очікувань. Це робить досвід користувача плавним та безшовним – саме таким, якого чекає користувач від сучасних цифрових платформ. React має велику спільноту, безліч готових бібліотек та компонентів, а також активно підтримується розробниками з усього світу. Це означає, що будь-яку складну задачу можна вирішити швидше – використовуючи перевірені рішення.

У межах цієї кваліфікаційної роботи React.js було обрано для реалізації клієнтської частини вебсервісу. Його використання дозволило створити інтерфейс, який є:

- сучасним та привабливим;
- адаптивним для різних пристроїв;
- швидким у роботі та гнучким у розвитку.

Таким чином, React – це не просто бібліотека, це філософія створення вебінтерфейсів, де на першому місці – користувач і його досвід. І саме ця філософія лягла в основу розробки даного застосунку – інструменту, який не лише надає доступ до відеоконтенту, а й робить цей доступ приємним, логічним і по-справжньому сучасним.

2.8 Ізольоване середовище Docker

Уявіть собі ідеальне середовище, де програма працює однаково добре та на локальному комп’ютері розробника, і на сервері в хмарі, і на машині

тестувальника, без помилок через різні версії бібліотек, налаштувань системи чи відсутніх залежностей. Це середовище не лише надійне, але й переносне, гнучке, ізольоване. Саме таку реальність створює Docker – технологія, що стала незамінною у світі DevOps та сучасної розробки.

Docker – це платформа, яка дозволяє створювати, розгортати та запускати застосунки в так званих контейнерах. Контейнер – це як мініатюрний комп'ютер усередині комп'ютера. Він містить усе необхідне для роботи застосунку: код, бібліотеки, конфігурації та навіть операційну систему. Завдяки цьому додаток працює однаково стабільно всюди, де запускається контейнер – незалежно від того, яка ОС чи середовище на машині.

У рамках кваліфікаційної роботи використання Docker мало стратегічне значення. Розробка системи перегляду фільмів передбачала роботу з кількома модулями: сервером на FastAPI, базою даних PostgreSQL, фронтендом на React, брокером повідомлень RabbitMQ. Docker дозволив об'єднати всі ці компоненти в єдине середовище, де кожен працює у своєму окремому контейнері, але взаємодіє з іншими без збоїв.

Серед переваг Docker варто виділити:

- ізоляцію середовища – кожен контейнер незалежний, що підвищує безпеку та надійність;
- масштабованість – легко запускати декілька екземплярів сервісу для обробки навантаження;
- автоматизацію розгортання – з Docker можна за кілька хвилин підняти весь проєкт на будь-якому сервері;
- контроль версій середовища – контейнер завжди містить точно ту конфігурацію, з якою він працював на етапі розробки.

Використання Docker дозволило зекономити час, уникнути проблем з сумісністю та забезпечити стабільну роботу системи в різних середовищах. І головне – це надало змогу сфокусуватися не на налаштуванні інфраструктури, а на тому, що справді важливо: створенні якісного,

сучасного продукту для перегляду відеоконтенту.

Docker – це не просто інструмент. Це підхід до розробки, який втілює головні цінності цифрової епохи: швидкість, гнучкість і надійність.

2.9 Зворотній проксі NGINX

У складному світі вебтехнологій, де на користувача чекає безліч мікросервісів, API, інтерфейсів та фонових процесів, виникає потреба в когось, хто зможе організувати цей цифровий оркестр. Саме такою роллю і наділений NGINX – один із найпотужніших вебсерверів, що з легкістю виконує роль зворотного проксі-сервера (reverse-proxy) у сучасних вебзастосунках.

На перший погляд, він може здатися звичайним інструментом для передачі запитів. Проте на практиці NGINX виконує набагато більше: він приймає запити від користувачів, розподіляє їх між різними модулями системи (наприклад, фронтендом, бекендом чи сервісами авторизації), кешує ресурси, зменшує навантаження на сервер і навіть підвищує безпеку, приховуючи внутрішню архітектуру застосунку.

У межах реалізації проєкту, присвяченого розробці стримінгового вебсервісу, NGINX виконує роль фронтового маршрутизатора, який:

- перенаправляє запити користувачів на React-застосунок;
- пересилає API-запити до FastAPI-сервера;
- виступає посередником між зовнішнім світом та внутрішньою інфраструктурою, створеною у Docker;
- забезпечує базову безпеку через SSL/TLS (за потреби).

Це дозволяє зменшити навантаження на основні сервіси, адже NGINX бере на себе частину обробки запитів – наприклад, кешує статичні файли, керує тайм-аутами, обробляє помилки та логірує події. А ще – виступає бар’єром між зовнішнім Інтернетом та внутрішніми мікросервісами, створюючи додатковий рівень безпеки. Його простота налаштування,

масштабованість та виняткова продуктивність роблять NGINX не просто корисним, а практично незамінним у сучасній архітектурі вебдодатків. Особливо у тих, де важлива швидкість доставки контенту та злагоджена робота кількох компонентів системи.

NGINX – це як диригент у великому технічному оркестрі. Він не грає сам, але саме завдяки йому кожен інструмент звучить у гармонії з іншими. І саме завдяки йому наш стримінговий сервіс працює швидко, злагоджено й надійно.

2.10 Хмарне середовище AWS

Світ IT давно залишив за спиною часи, коли кожен застосунок мав жити на одному фізичному сервері в окремій кімнаті з кондиціонером. Сьогодні програмне забезпечення вирушає "в хмари", де гнучкість, масштабованість та висока доступність – не просто бажані характеристики, а стандарт. Саме тому в межах кваліфікаційної роботи було обрано хмарну платформу Amazon Web Services (AWS) – світового лідера у сфері хмарних рішень.

Сервіс Amazon EC2 (Elastic Compute Cloud) став основною обчислювальною платформою, на якій розгорнуто серверну частину вебзастосунку. Це віртуальні машини, які можна масштабувати залежно від навантаження: збільшити потужність – у кілька кліків, зменшити – за потреби. EC2 забезпечує стабільну роботу FastAPI-сервера, RabbitMQ-брокера повідомлень та інші сервіси, розгорнуті в Docker-контейнерах. Надійність інфраструктури підтверджується роками досвіду AWS у забезпеченні високої доступності навіть для мільйонів користувачів.

У сучасному стримінговому сервісі зберігання контенту є критично важливим. Саме тут на допомогу приходить Amazon S3 (Simple Storage Service) – сервіс для зберігання великих обсягів даних. У межах проєкту S3 використовується для зберігання мультимедійних файлів, обкладинок

фільмів, прев’ю та інших статичних ресурсів. S3 не лише гарантує надійне зберігання та захист даних, а й забезпечує швидку доставку контенту через вбудовану систему кешування та інтеграцію з CDN.

Комунікація з користувачами – ще один важливий аспект цифрових сервісів. Для реалізації механізму надсилання підтверджень реєстрації, відновлення паролів та інформаційних повідомлень було інтегровано Amazon SES (Simple Email Service). На відміну від традиційних поштових рішень, SES забезпечує високу доставку листів, захист від спаму та просту інтеграцію з бекендом, що особливо актуально для стартапів та нових продуктів.

3 ОПИС РЕАЛІЗАЦІЇ ВЕБЗАСТОСУНКУ

3.1 Архітектура сервісів

3.1.1 Клієнт серверна архітектура

Клієнт-серверна архітектура – це модель організації обчислювального процесу, при якій завдання розподіляються між постачальниками ресурсів або сервісів та споживачами цих сервісів (рисунок 3.1). Такий підхід дозволяє ефективно масштабувати системи, розподіляти навантаження, підвищувати безпеку та забезпечувати централізоване керування компонентами системи.

У цій архітектурі клієнт ініціює запити до сервера, який обробляє ці запити та повертає необхідні дані або виконує відповідні дії. Клієнти можуть бути представлені у вигляді графічних інтерфейсів користувача (наприклад, вебзастосунки, мобільні додатки), а також інших програмних модулів або служб.

Сервер, у свою чергу, зазвичай працює на віддаленому серверному обладнанні й відповідає за зберігання та обробку даних, виконання бізнес-логіки, автентифікацію та авторизацію користувачів, а також за взаємодію з базами даних або іншими зовнішніми сервісами. Така модель забезпечує розділення обов’язків і підвищує гнучкість та надійність системи в цілому.

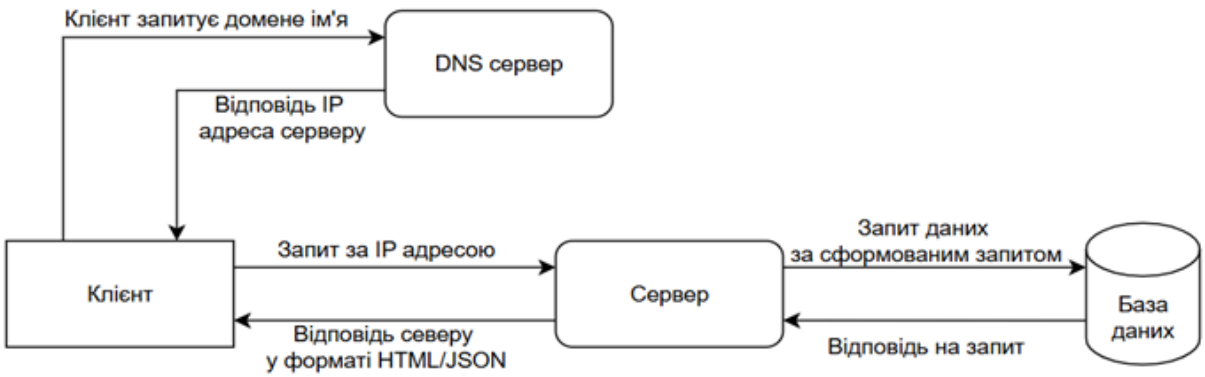

Рисунок 3.1 – Схема клієнт-серверної архітектури

3.1.2 Мікросервісна архітектура

Мікросервісна архітектура – це стиль побудови програмного забезпечення, в якому застосунок розбивається на набір невеликих, автономних сервісів, що взаємодіють між собою через чітко визначені інтерфейси, зазвичай за допомогою протоколів HTTP/REST, gRPC або систем обміну повідомленнями, таких як RabbitMQ чи Kafka [2]. Кожен мікросервіс виконує окрему бізнес-функцію та може бути розроблений, протестований, розгорнутий і масштабований незалежно від інших компонентів системи. Такий підхід дозволяє забезпечити високу гнучкість, надійність, масштабованість та спрощує підтримку й розвиток проєкту у довгостроковій перспективі.

У межах цього проєкту реалізовано повноцінну мікросервісну архітектуру, яка складається з семи окремих сервісів, кожен із яких відповідає за конкретний аспект бізнес-логіки (рисунок 3.2). До прикладу, окремі сервіси обробляють авторизацію, керування контентом, платіжну систему, генерацію підписаних URL для відео та аудіо, керування користувачами та коментарями.

Комунікація між сервісами відбувається переважно через брокер повідомлень RabbitMQ, що дозволяє реалізувати асинхронну взаємодію, зменшити зв’язність між компонентами та підвищити стійкість до збоїв. Для забезпечення незалежності та безпеки даних деякі мікросервіси використовують власні бази даних, що дозволяє зменшити ризики конфліктів при масштабуванні і підвищує загальну гнучкість структури.

Усі вхідні HTTP-запити до системи проходять через NGINX, який виконує роль зворотного проксі-сервера. Він здійснює маршрутизацію запитів до відповідних мікросервісів, а також виступає як балансувальник навантаження. Архітектурне рішення було розроблено з урахуванням подальшого масштабування системи та можливості легкої інтеграції нових сервісів у майбутньому.

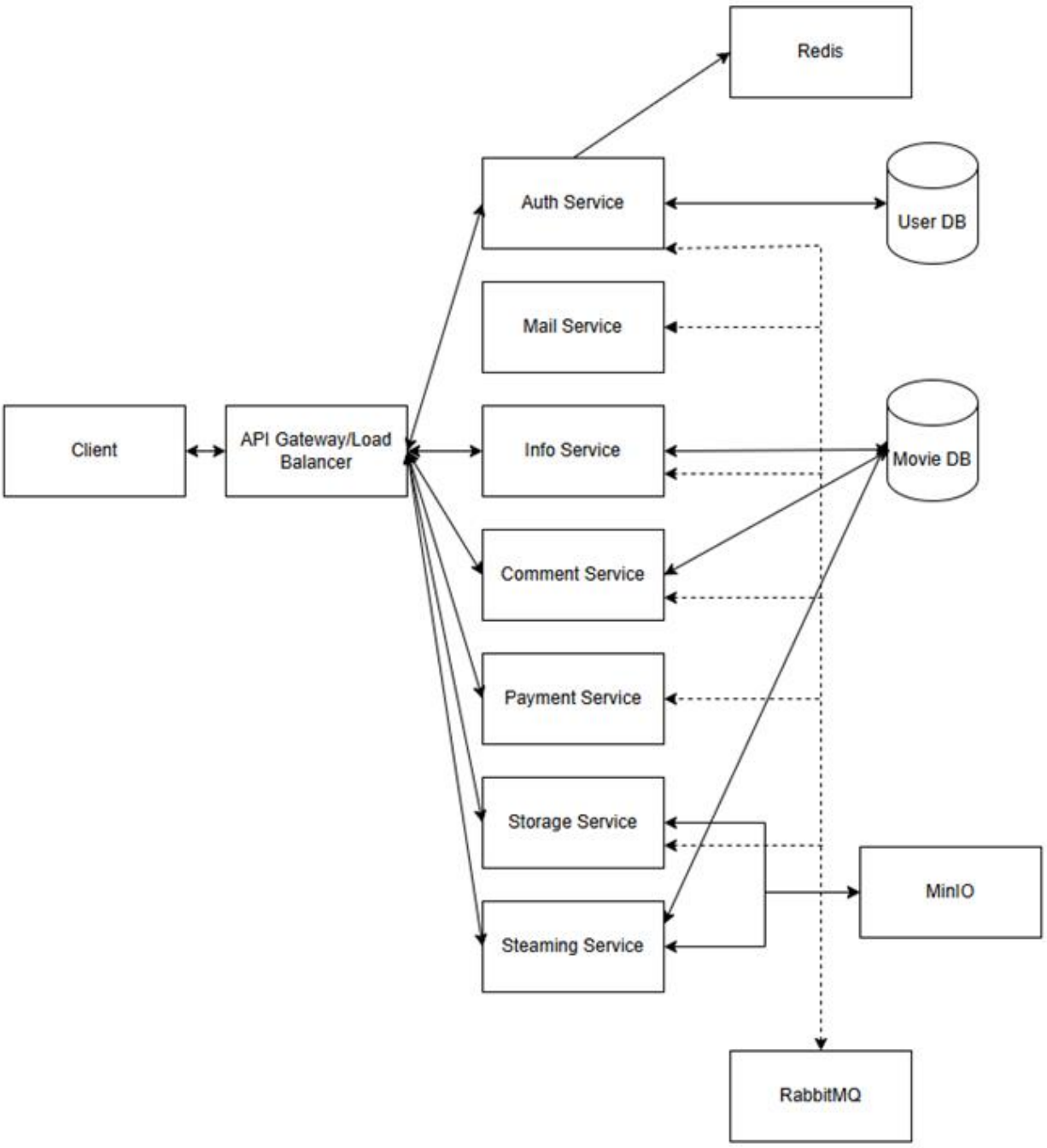

Рисунок 3.2 – Схема мікросервісної архітектури наведеної в проєкті

3.1.3 Архітектурний патерн "Репозиторій"

В рамках проєкту було застосовано архітектурний патерн "Репозиторій" (рисунок 3.3), який забезпечує абстракцію рівня доступу до даних [3]. Основною метою впровадження даного патерна є ізоляція бізнес-логіки від деталей реалізації роботи з базою даних, що, у свою чергу, підвищує модульність, тестованість та підтримуваність коду.

Репозиторій виступає у ролі посередника між шаром доменної логіки та шаром доступу до даних. Замість того, щоб компоненти системи напряму взаємодіяли з базою даних або ORM, вони звертаються до репозиторію, який надає уніфікований інтерфейс для виконання CRUD-операцій (створення, читання, оновлення, видалення).

Завдяки такому підходу зменшується зв'язність між компонентами системи, що є важливою вимогою до масштабованих мікросервісних архітектур. Крім того, це дозволяє легко замінювати джерело даних (наприклад, при переході з однієї СУБД на іншу), не змінюючи бізнес-логіку.

Використання патерну "Репозиторій" також сприяє впровадженню принципів SOLID, зокрема принципу єдиної відповідальності (Single Responsibility Principle) та інверсії залежностей (Dependency Inversion Principle), що є основою якісної та підтримуваної архітектури.

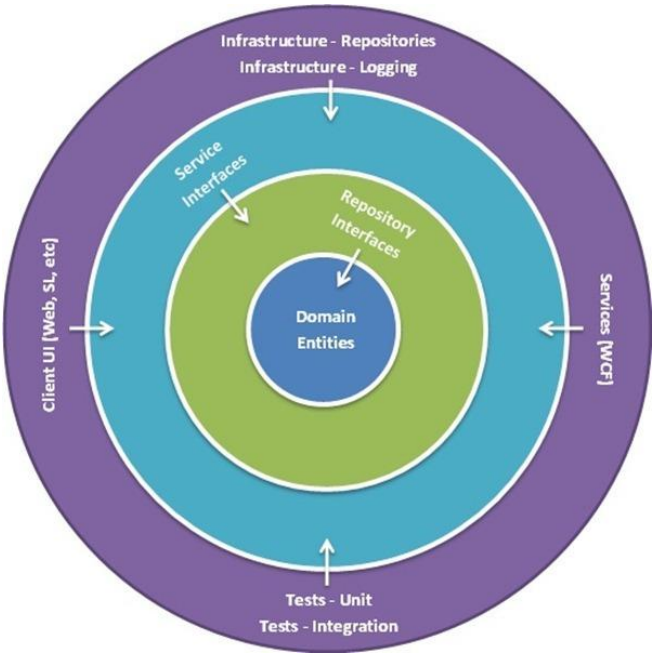

Рисунок 3.3 – Схематичне уявлення архітектурного патерну "Репозиторій"

Для повноцінної реалізації зазначеного патерна було розроблено структуровану файлову систему (рисунок 3.4), в якій чітко виділені окремі архітектурні шари. Кожен шар виконує власну функцію та не залежить від інших, що відповідає принципам низької зв’язності та високої когезії.

Такий підхід забезпечує логічну організацію проєкту, полегшує навігацію у кодовій базі, спрощує тестування окремих компонентів, а також сприяє масштабуванню та подальшій підтримці системи. Завдяки ізоляції шарів можлива незалежна розробка, модифікація або заміна окремих частин програми без ризику порушити загальну логіку застосунку.

```
app/
|   ├── api/
|   |   └── v1/
|   |   |   ├── endpoints/          # Визначення кінцевих точок API
|   |   |   └── dependencies.py     # Інжекція залежностей
|   |   └── routers.py              # Налаштування маршрутів API
|   ├── models/                     # Моделі бази даних (SQLAlchemy)
|   ├── repositories/               # Репозиторії для доступу до БД
|   ├── schemas/                    # Схеми валідації (Pydantic або Marshmallow)
|   ├── services/                   # Основна бізнес-логіка
|   ├── utils/                      # Утилітарні функції
|   ├── Dockerfile                  # Dockerfile для налаштування контейнера
|   ├── config.py                   # Конфігураційні налаштування
|   ├── database.py                 # Підключення до бази даних і її ініціалізація
|   └── main.py                     # Точка входу в застосунок
```

Рисунок 3.4 – Файлова структура кожного сервісу

3.2 Опис серверної частини

3.2.1 Контейнеризація та конфігурація сервісів

Для ізоляції, масштабування та зручного керування залежностями система використовує Docker з оркестрацією через Docker Compose (лістинг 3.1). Усі основні компоненти сервісної архітектури інкапсульовані в окремих контейнерах, які взаємодіють між собою через мережу app-network (якщо явно задана) [4]. Конфігурація зберігається у .env файлі, що дозволяє легко адаптувати середовище розгортання.

Лістинг 3.1 – Приклад розгортання бази даних і сервісу профілю

```
user-db:
    image: postgres:15
    container_name: postgres_user_db
    env_file:
      - .env
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB_USER}
    ports:
      - "5433:5433"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d
${POSTGRES_DB_USER}"]
      interval: 10s
      retries: 5
      start_period: 10s
profile-service:
    build:
      context: ./profile-service/app
      dockerfile: Dockerfile
    container_name: profile-service
    env_file:
      - .env
    volumes:
      - ./profile-service/app:/app
    expose:
      - "8000"
    environment:
      - PYTHONUNBUFFERED=1
    depends_on:
      user-db:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
```

В проєкті розгортаються дві окремі бази даних PostgreSQL 15:

- user-db – зберігає інформацію про користувачів;
- movie-db – містить дані, пов’язані з фільмами.

Обидві бази даних використовують образ postgres:15, налаштовуються через змінні середовища з .env, використовують загальний том postgres_data для зберігання даних, а також мають налаштовані healthcheck-перевірки через pg_isready.

Для реалізації бізнес-логіки застосовується набір мікросервісів, кожен з яких виконує окрему функцію в системі. Кожен мікросервіс збирається зі свого Dockerfile, монтує відповідну директорію з кодом та використовує змінні середовища:

- profile-service – відповідає за роботу з профілем користувача, залежить від user-db та rabbitmq;
- mail-service – сервіс для відправки електронної пошти. Використовує rabbitmq як брокер повідомлень;
- movie-service – обробляє інформацію про фільми, залежить від movie-db та rabbitmq;
- payment-service – відповідає за обробку платежів, залежить від rabbitmq;
- comment-service – відповідає за обробку коментарів користувачів, залежить від movie-db та rabbitmq;
- storage-service – відповідає за збереження та обслуговування файлів: постери до фільмів, зображення профілів, трейлери, обкладинки та інші мультимедійні ресурси;
- streaming-service – обслуговує потокову трансляцію відеоконтенту у форматі HLS.

Для забезпечення коректної роботи всі мікросервіси запускаються з параметром PYTHONUNBUFFERED=1, що дозволяє одразу виводити логи без буферизації, а також мають відкритий порт 8000 для внутрішнього доступу між контейнерами в межах мережі Docker.

Сервіси адміністрування та інфраструктури:

- pgAdmin – вебінтерфейс для роботи з PostgreSQL. Базується на образі dpage/pgadmin4, доступний на порту 5050, підключений до user-db та movie-db;
- Redis – інстанс Redis 7, використовується для кешування, запускається з режимом appendonly, доступний на порту 6379;
- RabbitMQ – брокер повідомлень, який забезпечує взаємодію між

мікросервісами. Використовує відкриті порти 5672 (для сервісів) та 15672 (панель управління), має налаштований healthcheck;

- MinIO – об’єктне сховище, сумісне з Amazon S3. Запускається на портах 9000 (для API) та 9001 (консоль керування), використовує окремий том minio_data.

3.2.2 Nginx конфігурація

У даній системі Nginx виконує роль реверс-проксі та за потреби може бути налаштований як балансувальник навантаження [6]. Він приймає всі вхідні HTTP-запити на порт 80 та, залежно від маршруту (/user/, /movie/ тощо), направляє їх до відповідних внутрішніх мікросервісів (лістинг 3.2), що дозволяє:

- централізовано керувати мережею запитів;
- приховати внутрішню архітектуру системи від зовнішнього світу;
- уніфікувати точку входу для клієнтів;
- збільшити безпеку та масштабованість системи.

У разі потреби можна проксирувати запити на декілька інстансів одного сервісу, що дає можливість розподіляти навантаження.

Лістинг 3.2 – Приклад розгортання сервісу профілю.

```
server {
    listen 80;
    server_name _;
    location /user/ {
        proxy_pass http://profile-service:8000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;    }
    location /movie/ {
        proxy_pass http://movie-service:8001/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
```

```
        proxy_set_header X-Forwarded-Proto $scheme;    }
    location /payment/ {
        proxy_pass http://payment-service:8002/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;    }
    location /comment/ {
        proxy_pass http://comment-service:8003/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;    }
    location /storage/ {
        proxy_pass http://storage-service:8004/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;    }
location /streaming/ {
        proxy_pass http://streaming-service:8005/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;    } }
```

Основні маршрути Nginx:

- /user/ → profile-service:8000 – проксирує запити, пов’язані з профілем користувача: авторизація, реєстрація, редагування профілю тощо;

- /movie/ → movie-service:8001 – обробка запитів до фільмового сервісу: перегляд списків фільмів, отримання інформації про конкретний контент, рекомендації тощо;

- /payment/ → payment-service:8002 – шлях для обробки платіжних операцій, транзакцій, підписок;

- /comment/ → comment-service:8003 – робота з коментарями: створення, видалення, отримання;

- /storage/ → storage-service:8004 – проксирування запитів до MinIO (об'єктного сховища файлів). Застосовується для збереження зображень, відео, документів та іншого контенту;

- /streaming-service/ – запити, пов’язані з потоковою трансляцією відео, обробляються відповідним сервісом потокового відео, який відповідає за доставку мультимедійного контенту в режимі реального часу.

3.2.3 Опис шару вза’ємодії з SQLAlchemy моделями.

У кожному сервісі програмного забезпечення, розробленого в межах цього проєкту, реалізовано окремий шар доступу до бази даних (лістинг 3.3). Цей шар побудований на базі ORM-фреймворку SQLAlchemy [7], що підтримує асинхронні сесії, та реалізований шляхом спадкування від спеціального абстрактного класу. У ньому описано загальні методи, які мають бути реалізовані в конкретних реалізаціях репозиторіїв. Варто зазначити, що абстрактний клас і класи взаємодії можуть мати різну структуру в залежності від специфіки кожного окремого сервісу.

Головна ідея впровадження такого шару полягає у відокремленні CRUD-операцій (створення, читання, оновлення, видалення) від бізнес-логіки. Завдяки цьому досягається гнучкість у розширенні та модифікації логіки взаємодії з базою даних без впливу на решту частин системи. Такий підхід дозволяє також підвищити безпеку застосунку, зокрема зменшити ризики SQL-ін’єкцій, навіть у випадках, коли запити формуються на клієнтському рівні – оскільки вони проходять валідацію як на рівні схем, так і через логіку обробки в бекенді.

Кожна модель взаємодіє з базою даних через спеціалізований репозиторій, який наслідує базовий клас SqlLayer. У цьому базовому класі визначено загальну логіку роботи з базою, яку можна повторно використовувати у різних сервісах. Це забезпечує ізоляцію доступу до даних від бізнес-логіки, що спрощує процеси тестування, масштабування та обслуговування системи.

Лістинг 3.3 – Метод get класу взаємодії роботи з базою даних

```
async def get(self, *args: Any, **kwargs: Any) -> Any:
       async with async_session_maker() as session:
           try:
               stmt = ( select(self.model)
                   .where(self._build_where_clause(kwargs))
.options(*self.get_all_relationships(self.model))            )
               res = await session.execute(stmt)
               return res.scalar_one_or_none()
           except Exception as e:
               raise Exception(f"Get Error in
{self.model.__class__.__name__}: {e}")
```

Базовий функціонал взаємодії з даними описано в абстрактному класі AbstractRepository, який визначає перелік методів, що мають бути реалізовані в дочірніх класах, зокрема insert, get, get_all, update, delete (лістинг 3.4).

Лістинг 3.4 – Абстрактний клас для взаємодії з базою даних

```
class AbstractRepository(ABC):
    @abstractmethod
    async def insert(self, data: dict) -> object:
        pass
    @abstractmethod
    async def get_all(self, *args: Any, **kwargs: Any) ->
list[object]:
        pass
    @abstractmethod
    async def get(self, *args: Any, **kwargs: Any) -> object:
        pass
    @abstractmethod
    async def update(self, data: dict, *args: Any, **kwargs: Any)
-> object:
        pass
    @abstractmethod
    async def delete(self, *args: Any, **kwargs: Any):
        pass
```

Для відкриття асинхронної сесії з базою даних використовується контекстний менеджер async_session_maker, який створюється з використанням об’єкту create_async_engine. Цей механізм є частиною SQLAlchemy 2.0 і забезпечує ефективну роботу з БД в асинхронному середовищі (лістинг 3.5).

Лістинг 3.5 – Ініціалізація асинхронної сесії

```
engine = create_async_engine(config_setting.DB_URI)
async_session_maker = async_sessionmaker(
    engine, expire_on_commit=False, class_=AsyncSession)
```

Для прив’язування моделей таблиць до логіки доступу використовується підхід із класами-репозиторіями. Кожен такий клас наслідує SqlLayer і в атрибуті model визначає відповідну ORM-модель, яка буде використовуватися для взаємодії з конкретною таблицею (лістинг 3.6).

Лістинг 3.6 – Класи репозиторіїв для моделей

```
 class ShowRepository(SqlLayer):
    model = ShowModel
class SeasonRepository(SqlLayer):
    model = SeasonModel
class SerieRepository(SqlLayer):
    model = SerieModel
```

Ці класи отримали інстанції моделей, які описанні в сервісах та будуть використовуватись у бізнес логіці для сервісу. Всі моделі описуються за допомогою SQLAlchemy в ORM стилі, за стандартом SQLAlchemy 2.0, де визначають всі необхідні поля для таблиць та зв’язки між ними за допомогою relationship (лістинг 3.7).

Лістинг 3.7 – Класи моделі для таблиці seasons

```
class SeasonModel(Base):
    __tablename__ = "seasons"
    id: Mapped[uuid.UUID] = mapped_column(default=uuid.uuid4,
primary_key=True)
    title: Mapped[str] = mapped_column(unique=True)
    preview: Mapped[str] = mapped_column(nullable=True)
    description: Mapped[str] = mapped_column()
    status: Mapped[str] = mapped_column(default="ongoing",
nullable=True)
    country_id: Mapped[uuid.UUID] =
mapped_column(ForeignKey("countries.id"))
    raiting: Mapped[float] = mapped_column(default=0.0)
    show_id: Mapped[uuid.UUID] =
mapped_column(ForeignKey("shows.id"))
    release_date: Mapped[datetime] = mapped_column()
    create_at: Mapped[datetime] =
mapped_column(default=datetime.now())
    serie: Mapped[List["SerieModel"]] =
relationship(back_populates="season")
```

```
    show: Mapped["ShowModel"] =
relationship(back_populates="season")
    country: Mapped["CountryModel"] =
relationship(back_populates="season")
    season_genres: Mapped[List["SeasonGenreModel"]] =
relationship(
        back_populates="season", cascade="all, delete-orphan",
passive_deletes=True    )
    genres: Mapped[List["GenreModel"]] = relationship(
        secondary="season_genres", viewonly=True,
back_populates="seasons"    )
    season_studios: Mapped[List["SeasonStudioModel"]] =
relationship(
        back_populates="season", cascade="all, delete-orphan",
passive_deletes=True    )
    studios: Mapped[List["StudioModel"]] = relationship(
        secondary="season_studios", viewonly=True,
back_populates="seasons")
```

Такий підхід дозволяє гнучко працювати з даними, організовувати складні зв’язки між сутностями, а також легко масштабувати проєкт у майбутньому. Крім того, структура моделей та взаємозв’язків робить систему зрозумілою для розробників і придатною до подальшої інтеграції з іншими компонентами програмної архітектури.

3.2.4 Опис класів бізнес логіки та взаємодія з утілітами

У кожному мікросервісі проєкту реалізовано окремі класи, які відповідають за бізнес-логіку відповідного сервісу (лістинг 3.8). Ці класи є незалежними від зовнішніх фреймворків або бібліотек і побудовані на принципах інверсії залежностей та розділення відповідальностей. Такий підхід дозволяє забезпечити високу модульність коду, легкість у тестуванні, а також спрощує процес розширення або зміни логіки без потреби модифікації інших компонентів системи.

Основною метою бізнес-класів є реалізація прикладної логіки, зокрема:

- обробка та валідація вхідних даних перед їх передачею до шару роботи з базою даних;

- перевірка існування пов’язаних сутностей, необхідних для коректного збереження інформації;

- взаємодія з чергами повідомлень (RabbitMQ), системами зберігання (MinIO) або іншими сервісами через абстрактні інтерфейси.

Конструктори таких класів приймають усі необхідні залежності через параметри, що дозволяє уникнути жорсткої прив’язки до конкретної реалізації. Це також забезпечує високу гнучкість – у випадку зміни інфраструктури (наприклад, перехід на іншу СУБД або черговий брокер), достатньо змінити реалізацію відповідного класу, не змінюючи саму бізнес-логіку.

Лістинг 3.8 – Клас бізнес логіки з атрибутами та методом створення

```
class SerieService(Protocol):
    def __init__(
        self, serie_repo, season_repo, queue_manager,
storage_manager, error_handler
    ) -> None:
        self.serie_repo: AbstractRepository = serie_repo()
        self.season_repo: AbstractRepository = season_repo()
        self.queue_manager: AbstractQueue = queue_manager()
        self.storage_manager: AbstractStorage = storage_manager()
        self.error_handler = error_handler
    async def create(self, data: dict, file) -> object:
        try:
            await self._check_relashions(data=data)
            if file:
                preview_url = await self._upload_to_minio("season-
preview", file=file)
            return await self.serie_repo.insert(data=data)
        except Exception:
            raise self.error_handler(status_code=500,
detail="Internal Server Error")
```

У межах деяких мікросервісів реалізовано обробку міжсервісної взаємодії через брокер RabbitMQ [7], що забезпечує асинхронну та безпечну передачу повідомлень у рамках приватної мережі (лістинг 3.9). Такий механізм дозволяє сервісам обмінюватися даними без прямої залежності один від одного, зменшуючи рівень зв’язаності між модулями системи.

Для роботи з чергами повідомлень передбачено створення окремих класів, які реалізують низку методів, необхідних для коректної взаємодії з RabbitMQ:

- connect – встановлення з’єднання з брокером;
- start_consuming – запуск прослуховування повідомлень із заданих черг;
- publish – відправка повідомлень до інших мікросервісів;
- rpc_node – реалізація RPC-повідомлень, які потребують відповіді.

Лістинг 3.9 – Абстрактний клас для взаємодії з RabbitMQ

```
class AbstractQueue(ABC):
    @abstractmethod
    async def connect(self) -> None:
        pass
    @abstractmethod
    async def publish(self, queue_name, message) -> None:
        pass
    @abstractmethod
    async def start_consuming(self) -> None:
        Pass
```

Ще одним компонентом, який активно використовується в сервісах, є система збереження медіафайлів. У даному проєкті для цієї мети застосовується MinIO [8] – S3-сумісне рішення для об'єктного зберігання, яке дозволяє інтегрувати з Amazon S3 або іншими сумісними сховищами (лістинг 3.10).

Класи для взаємодії з MinIO реалізують такі базові методи:

- створення підписаних URL-адрес для тимчасового доступу до файлів (наприклад, для завантаження відео);
- завантаження файлів на сервер;
- видалення файлів зі сховища.

Підписані посилання забезпечують захист контенту: кожен URL має обмеження в часі, і його не можна повторно використати після завершення терміну дії. Це особливо актуально при роботі з чутливими або комерційними даними.

Лістинг 3.10 – Клас абстракції для роботи з MinIO

```
class AbstractStorage(ABC):
    @abstractmethod
    async def presign_url(self, bucket_name: str, object_name:
str, expires_minutes: int) -> str:
        pass
    @abstractmethod
    async def upload_file(self, bucket_name: str, file_name: str,
content, content_type: str) -> str:
        pass
    @abstractmethod
    async def delete_file(self, bucket_name: str, file_name: str)
-> bool:
        pass
```

3.2.5 Залежності для FastAPI

У процесі реалізації програмного забезпечення було обрано архітектурний підхід, за якого бізнес-логіка сервісу не залежить безпосередньо від веб-фреймворку (лістинг 3.11). Для інтеграції з FastAPI застосовується механізм ін'єкції залежностей – потужний інструмент, що дозволяє передавати необхідні компоненти у функції обробки HTTP-запитів. Такий підхід сприяє кращому розділенню відповідальностей, покращує тестованість компонентів і спрощує супровід системи в майбутньому.

Одним із базових елементів є функція залежності для створення екземпляру сервісу, що реалізує бізнес-логіку. Наприклад, у мікросервісі, який обробляє інформацію про серіали, була реалізована функція serie_dep, яка повертає екземпляр класу SerieService.

Лістинг 3.11 – Функція залежності створення екземпляру класу SerieService

```
async def serie_dep() -> SerieService:
    return SerieService(
        serie_repo=SerieRepository,
        season_repo=SeasonRepository,
        queue_manager=RabbitMqManager,
        storage_manager=MinioStorage,
        error_handler=HTTPException, )
```

Кожна із переданих залежностей реалізує власну частину логіки:

- serie_repo – доступ до даних серіалів;

- season_repo – робота з сезонами;
- queue_manager – взаємодія з брокером повідомлень RabbitMQ;
- storage_manager – зберігання мультимедійних файлів через MinIO;
- error_handler – обробка помилок.

Такий спосіб організації дозволяє легко модифікувати або підмінювати окремі частини системи без необхідності вносити зміни у всю архітектуру.

Для перевірки на авторизацію теж використовується залежність is_admin та is_user (лістинг 3.12). В залежності від користувача будуть надаватися права доступу до енпоінтів на запис, оновлення, видалення, записів для фільмів, серіалів, серій, аудіо (тільки адміністратор може робити це, але користувач може видаляти коментарі, які він написав та редагувати свій профіль)

Лістинг 3.12 – Функція залежності для перевірки на адміністратора

```
async def is_admin(token: Annotated[str,
Depends(oauth2_scheme)]):
    try:
        security = JWTAuth()
        payload = await security.decode_token(token=token)
        user_id = payload.get("id", None)
        role = payload.get("role", None)
        if user_id:
            if role == "admin":
                return True
            else:
                raise HTTPException(status_code=403,
detail="Permission Denied")
        else:
            raise HTTPException(status_code=401,
detail="Unauthorized")
    except Exception:
        raise HTTPException(status_code=500, detail="Internal
Server Error")
```

Для кодування та декодування розроблений клас JWTAuth [9] (лістинг 3.13), який використовує алгоритм SHA256 та секретний ключ згенерований за допомогою Linux терміналу. При створенні токену виставляється час життя 15 хвилин для access-token та 7 днів для refresh-token. Коли access-token вичерпав час життя клієнт отримає HTTP помилку зі

статусом 401 та текстом "Unauthorized". Після чого клієнт повинен відправити запит на сервіс авторизації запит на оновлення пари токенів (система ротацій токенів) для отримання нової пари токенів. Всі токени перелаються виключно в cookie.

Лістинг 3.13 – Клас для кодування та декодування JWT токену

```
class JWTAuth(SecurityBase):
    SECRET_KEY = config_setting.SECRET_KEY
    ALGORITHM = config_setting.ALGORITHM
    async def create_token(self, data: dict) -> str:
        try:
            encoded_jwt = jwt.encode(
                data,
                self.SECRET_KEY,
                algorithm=self.ALGORITHM,          )
            return encoded_jwt
        except Exception as e:
            raise Exception(
                f"Create Token Error in
{self.create_token.__name__}: {e}"          )
    async def create_access_token(self, data: dict, time: int) ->
str:
        try:
            to_encode = data.copy()
            expire = datetime.now(timezone.utc) + timedelta(
minutes=config_setting.ACCESS_TOKEN_EXPIRE_MINUTES          )
            to_encode.update({"exp": expire})
            return await self.create_token(data=to_encode)
        except Exception as e:
            raise Exception(
                f"Create Access Token Error in
{self.create_access_token.__name__}: {e}"          )
    async def create_refresh_token(self, data: dict) -> str:
        try:
            to_encode = data.copy()
            expire = datetime.now(timezone.utc) + timedelta(
                days=config_setting.REFRESH_TOKEN_EXPIRE_DAYS )
            to_encode.update({"exp": expire})
            return await self.create_token(data=to_encode)
        except Exception as e:
            raise Exception(
                f"Create Refresh Token Error in
{self.create_refresh_token.__name__}: {e}"          )
    async def decode_token(self, token: str) -> dict:
        try:
            payload = jwt.decode(
                token,
                self.SECRET_KEY,
```

```
            algorithms=self.ALGORITHM,          )
        return payload
    except Exception as e:
        raise Exception(f"Decode Token Error in
{self.decode_token.__name__}: {e}")
```

3.2.6 Ендпоінти FastAPI

У фреймворку FastAPI побудова REST API здійснюється за допомогою оголошення асинхронних функцій, які описують логіку обробки HTTP-запитів. Для організації маршрутів (ендпоінтів) використовується клас APIRouter, екземпляр якого приймає параметри prefix (префікс URL-шляху) та tags (мітки для документації OpenAPI/Swagger). Такий підхід дозволяє логічно згрупувати функціонал за категоріями, підвищуючи читаємість та підтримуваність коду.

Методи, які можуть бути використані у роутері (@router.get, @router.post, @router.put, @router.patch, @router.delete), визначають HTTP-метод запиту. Вони дозволяють вказати шлях, код статусу відповіді за замовчуванням, схеми валідації за допомогою Pydantic, а також метадані для автоматичної генерації документації.

Захищені ендпоінти, які вимагають авторизації (наприклад, перевірки JWT-токену), реалізуються через механізм залежностей (Depends). Це дозволяє інтегрувати логіку авторизації або інші компоненти, як-от бізнес-логіку, не порушуючи принцип інверсії залежностей (лістинг 3.14).

Лістинг 3.14 – Захищений ендпоінт та зв’язаний с сервісом

```
router = APIRouter(prefix="/serie", tags=["Serie"])
serie_depend = Annotated[SerieService, Depends(serie_dep)]
admin_depend = Annotated[None, Depends(is_admin)]
@router.post("/create", status_code=status.HTTP_201_CREATED)
async def create_serie(data: CreateSerie, admin: admin_depend,
service: serie_depend) -> ReadSerie:
    service_action = await
service.create(data=data.model_dump(exclude_none=True))
return service_action
```

3.2.7 Схеми валідації JSON

FastAPI тісно інтегрований з бібліотекою Pydantic, яка забезпечує автоматичну валідацію даних, серіалізацію/десеріалізацію JSON та генерацію документації. Для кожної категорії запитів створюються окремі Pydantic-моделі, які забезпечують типобезпеку й перевірку вхідних даних (лістинг 3.15).

Основні типи схем:

- BaseScheme – містить базові атрибути, спільні для створення, оновлення та читання записів. Слугує батьківським класом для інших схем;
- CreateScheme – використовується для перевірки даних при створенні нового запису;
- UpdateScheme – дозволяє оновлювати лише частину полів, тому всі атрибути оголошуються як необов’язкові (Optional);
- ReadScheme – використовується при відправці відповіді клієнту. Завдяки параметру from_attributes = True можливе автоматичне заповнення даних з ORM-об’єкта SQLAlchemy.

Лістинг 3.15 – Захищений ендпоінт та зв’язаний с сервісом

```
class SerieBase(BaseModel):
    title: str
    season_id: Union[ShowBase, uuid.UUID]
    release_date: datetime
class CreateSerie(SerieBase):
    pass
class UpdateSerie(SerieBase):
    title: Optional[str] = None
    season_id: Optional[uuid.UUID] = None
    status: Optional[str] = None
    video_path: Optional[str] = None
    release_date: Optional[datetime] = None
class ReadSerie(SerieBase):
    id: uuid.UUID
    status: str
    create_at: datetime
    class Config:
        from_attributes = True
```

3.2.8 Ініціалізація серверу FastAPI

Була розроблена функція для ініціалізації серверу. При виклику цієї функції створюється інстанція FastAPI, створюються таблиці (якщо ще не існують), підключаються маршрути кластерів ендпоінтів, підключається до RabbitMQ та Redis (лістинг 3.16).

Лістинг 3.16 – Захищений ендпоінт та зв’язаний с сервісом

```
def get_application() -> FastAPI:
    async def startup():
        async with engine.begin() as conn:
            await conn.run_sync(Base.metadata.create_all)
    application = FastAPI(title="Profile Service")
    application.add_event_handler("startup", startup)
    for router in routers:
        application.include_router(router)
    return application
```

3.2.9 Ендпоінт стримінгу

Стримінговий сервіс відповідає за передачу відео- та аудіоконтенту користувачу у захищеному режимі. Для цього застосовується механізм генерації підписаних URL-адрес (signed URLs) із обмеженим часом дії (15 хвилин), що запобігає несанкціонованому доступу до файлів.

API-ендпоінт /series/{serie_id} приймає унікальний ідентифікатор серіалу та повертає користувачу (лістинг 3.17):

- підписане посилання на відеопотік;
- список аудіодоріжок з відповідними підписаними URL та мовними кодами.

Підписані URL-адреси генеруються через сервіс сховища (MinIO), який підтримує створення тимчасових посилань на об’єкти, що дозволяє клієнтській частині безпечно завантажувати або відтворювати контент, не отримуючи прямого доступу до сховища.

Такий підхід забезпечує баланс між зручністю користувача (можливістю вибирати аудіо в реальному часі) і захистом контенту від піратства та несанкціонованого копіювання.

Лістинг 3.17 – Ендпоінт для формування посилань для стримінгу

```
@router.get("/series/{serie_id}")
async def get_serie(serie_id: UUID, service: SerieService =
Depends()):
    serie = await service.get(id=serie_id)
    video_signed_url = await service.storage_manager.presign_url(
        bucket_name="video",
        object_name=serie.video_path,
        expires_minutes=15    )
    audio_list = []
    for audio in serie.audio:
        signed_audio_url = await
service.storage_manager.presign_url(
            bucket_name="audio",
            object_name=audio.file_path,
            expires_minutes=15        )
        audio_list.append({
            "lang": audio.language_code,
            "url": signed_audio_url       })
    return {
        "video_url": video_signed_url,
        "audios": audio_list    }
```

3.3 Опис відеоплеєра

Для реалізації клієнтської частини сервісу був розроблений компонент HLSPlayer (лістинг 3.18), побудований на основі React.js з використанням бібліотеки hls.js. Основна функція компонента – відтворення відео з потоковим протоколом HLS [10] із можливістю динамічного перемикання аудіодоріжок під час перегляду.

Компонент використовує React-хуки useRef та useState для управління елементом відео та станом аудіодоріжок:

- videoRef – посилання на DOM-елемент <video>, в якому відбувається відтворення:
- tracks – стан, що містить масив доступних аудіодоріжок, отриманих

із HLS-манифесту;

- hlsInstance – екземпляр класу Hls для керування процесом відтворення.

У функціональному хуке useEffect при ініціалізації компонента виконується перевірка підтримки HLS браузером за допомогою Hls.isSupported(). Якщо підтримка є, створюється новий об’єкт Hls, завантажується HLS-манифест за вказаним URL (masterUrl) та підключається до відеоелемента. Після розбору манифесту (MANIFEST_PARSED) отримується список доступних аудіодоріжок, який зберігається у стані tracks. Також відстежується подія перемикання аудіодоріжок (AUDIO_TRACK_SWITCHED) для логування обраної доріжки.

Для реалізації перемикання аудіодоріжок передбачена функція switchAudio(id), яка встановлює активну аудіодоріжку в об’єкті hlsInstance. Для забезпечення безперервності відтворення поточного відео зберігається та відновлюється час відтворення (currentTime), що усуває будь-які затримки або розриви при зміні аудіодоріжки.

За допомогою інтерфейсу користувача здійснюється можливість обирання відео- контенту. Компонент виводить елемент <video> з базовими контролами та шириною 800 пікселів. Під ним розміщуються кнопки для вибору аудіодоріжок, які генеруються динамічно на основі отриманих метаданих (name або lang аудіодоріжки). Користувач може легко переключати аудіо під час перегляду, що підвищує зручність використання та адаптивність сервісу для користувачів із різними мовними вподобаннями.

Лістинг 3.18 – Реалізація плеєра на клієнтській частині

```
import Hls from "hls.js";
import React, { useEffect, useRef, useState } from "react";
const HLSPlayer = ({ masterUrl }) => {
  const videoRef = useRef();
  const [tracks, setTracks] = useState([]);
  const [hlsInstance, setHlsInstance] = useState(null);
  useEffect(() => {
    if (Hls.isSupported()) {
      const hls = new Hls();
```

```
      hls.loadSource(masterUrl);
      hls.attachMedia(videoRef.current);
      hls.on(Hls.Events.MANIFEST_PARSED, () => {
        setTracks(hls.audioTracks);     });
      hls.on(Hls.Events.AUDIO_TRACK_SWITCHED, (e, data) => {
        console.log("Switched to:", data.id);     });
      setHlsInstance(hls);    }
  }, [masterUrl]);
  const switchAudio = (id) => {
    if (hlsInstance) {
      const currentTime = videoRef.current.currentTime;
      hlsInstance.audioTrack = id;
      videoRef.current.currentTime = currentTime;    }  };
  return (
    <div>
      <video ref={videoRef} controls width="800" />
      <div>
        {tracks.map((track, idx) => (
          <button key={idx} onClick={() => switchAudio(idx)}>
            {track.name || track.lang}
          </button>
        ))}
      </div>
    </div>   );  };
export default HLSPlayer;
```

Для забезпечення актуальності контенту у відеоплеєрі реалізовано механізм періодичного оновлення джерела потокового відео під час відтворення. Це особливо важливо для серіалів або трансляцій, де плейлист може змінюватись у процесі (лістинг 3.19).

Лістинг 3.19 – Автооновлення посилань на медіа ресурси

```
useEffect(() => {
  const interval = setInterval(async () => {
    const res = await fetch(`/series/123/playlist`);
    const { master_url } = await res.json();
    const currentTime = videoRef.current.currentTime;
    hlsInstance.loadSource(master_url);
    hlsInstance.attachMedia(videoRef.current);
    videoRef.current.currentTime = currentTime;
  }, 13 * 60 * 1000);
  return () => clearInterval(interval);
}, []);
```

Таким чином, завдяки періодичному оновленню HLS-плейлиста, контент залишається актуальним та захищеним від можливих проблем із

кешуванням або застарілими сегментами. Це забезпечує безперебійну трансляцію відео без втрати позиції відтворення. Крім того, користувач отримує можливість у реальному часі змінювати аудіодоріжки завдяки динамічному переключенню аудіотреків через інтерфейс плеєра, що підвищує зручність та індивідуалізацію перегляду.

4 ІНСТРУКЦІЯ КОРИСТУВАЧА

При переході на сайт користувача зустрічає головна сторінка, яка містить інтуїтивно зрозуміле меню навігації, логотип сервісу та пошукову панель для швидкого доступу до необхідного контенту (рисунок 4.1). Меню забезпечує легкий перехід між основними розділами сайту, а пошук дозволяє користувачам ефективно знаходити потрібну інформацію чи медіафайли, що підвищує загальний комфорт та зручність користування сервісом.

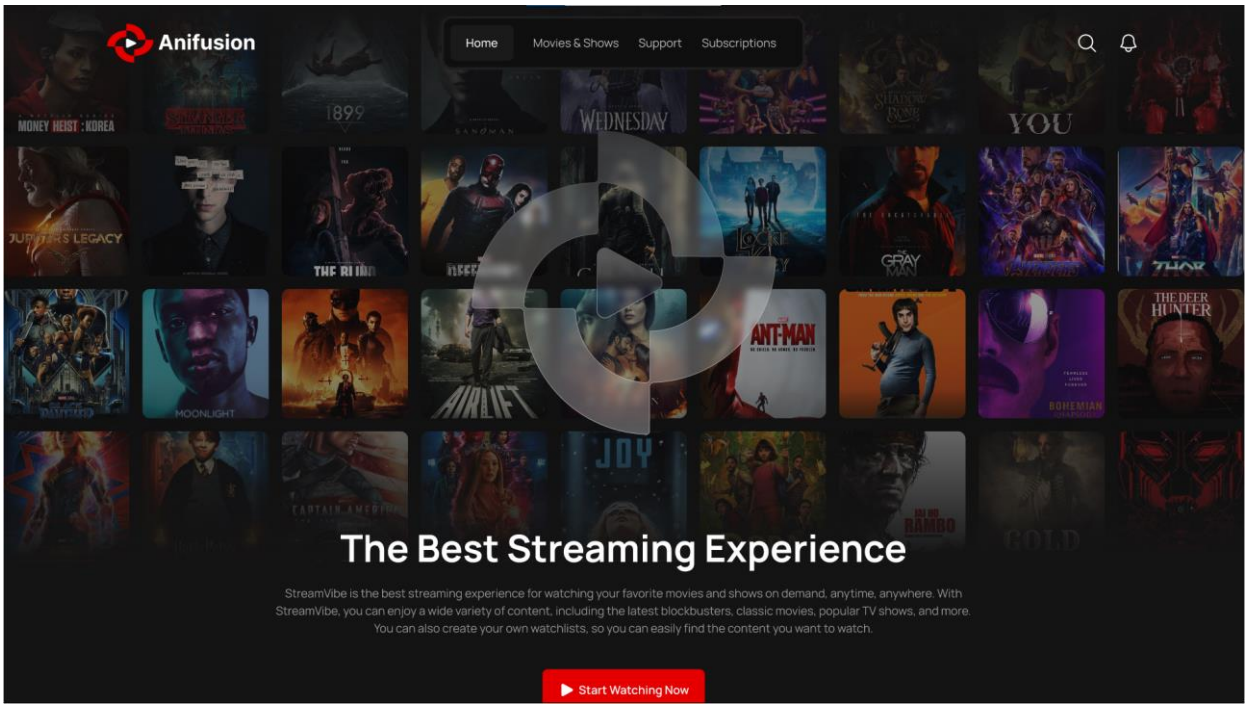

Рисунок 4.1 – Головна сторінка сайту

Нижче на сторінці користувачу пропонується обрати жанр, який його найбільше цікавить (рисунок 4.2). На представленому зображенні показані лише деякі з доступних жанрів, що ілюструє широкий вибір контенту для різних смаків. Окрім цього, на сторінці розміщені найпопулярніші запитання від інших користувачів із детальними відповідями, що допомагає швидко отримати необхідну інформацію та розвіяти можливі сумніви.

Також користувач має можливість ознайомитися з різними планами підписки на сервіс, серед яких він може обрати найбільш вигідний для себе

варіант (рисунок 4.3). Для оплати підписок інтегровано платіжний шлюз Stripe, який забезпечує безпечне та надійне проведення фінансових транзакцій, гарантує захист персональних та платіжних даних користувачів, а також підтримує зручний і швидкий процес оплати.

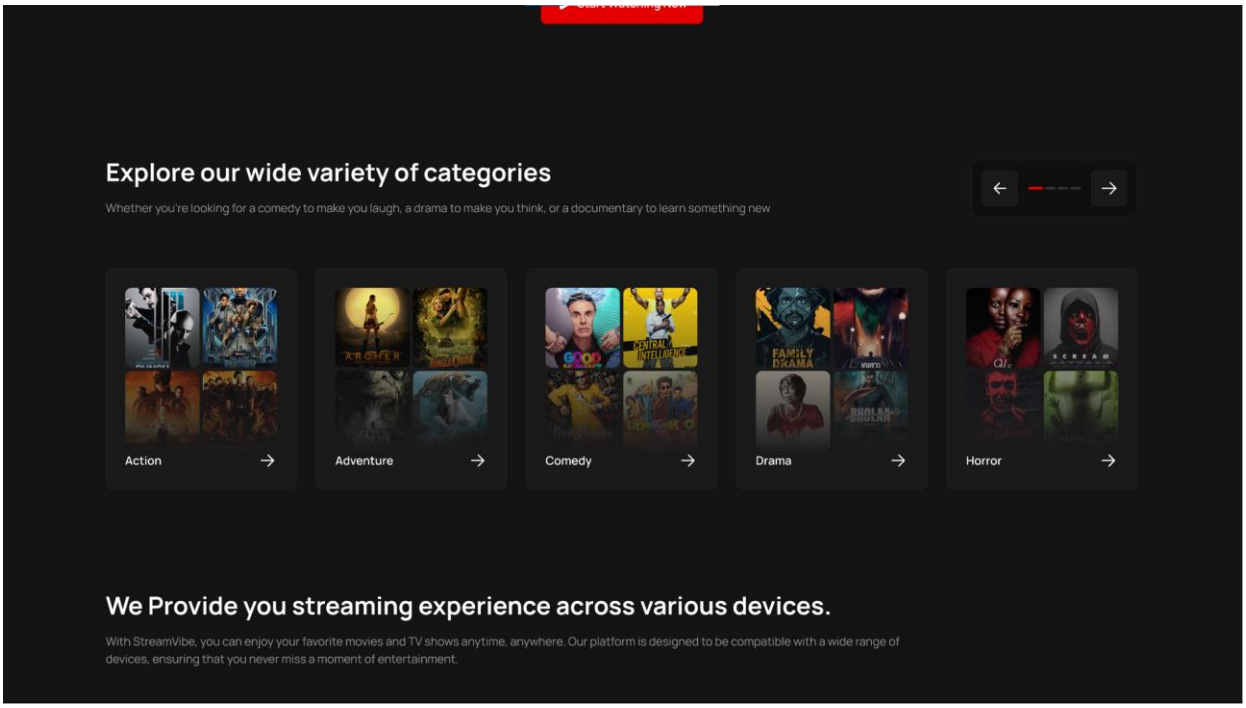

Рисунок 4.2 – Вибір жанру

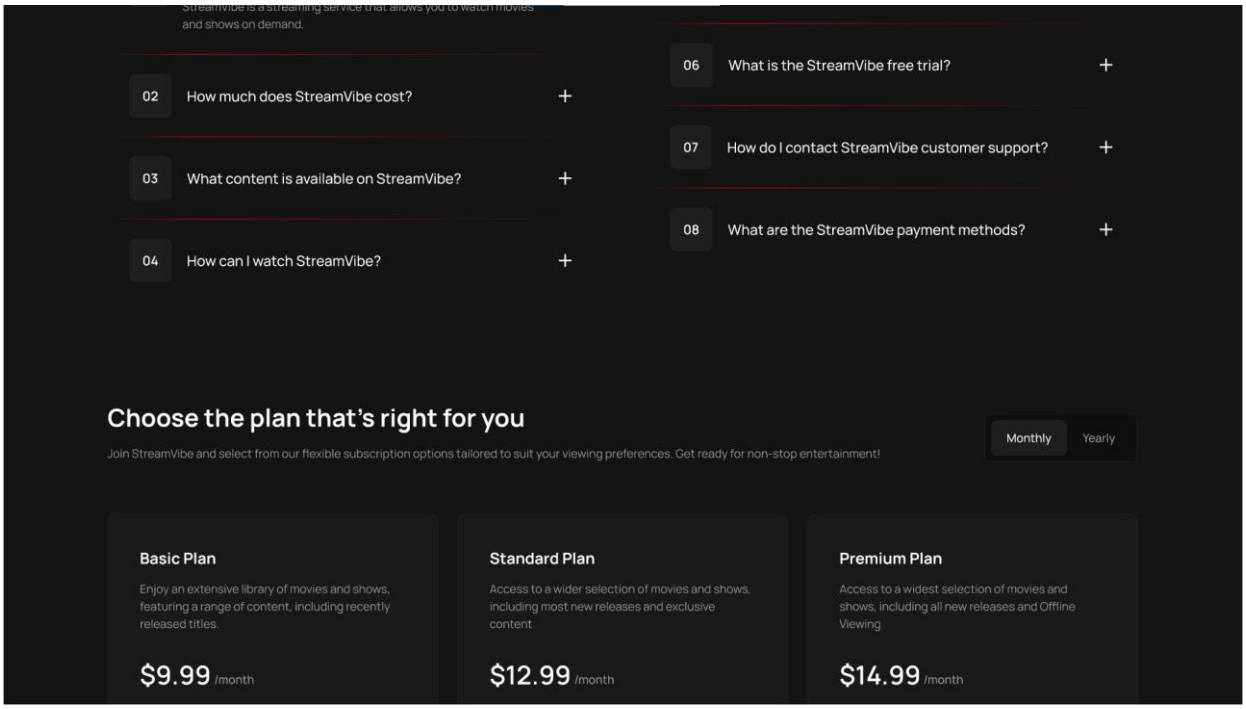

Рисунок 4.3 – Розповсюджені питання та плани підписок

На сторінці Movies & Shows користувачу надається можливість обрати жанр, який його цікавить (рисунок 4.4). Після вибору жанру відображається відповідний список фільмів та серіалів, що відповідають напрямку. Такий підхід дозволяє користувачу швидко знаходити потрібні йому відеоматеріали, оптимізуючи процес навігації по сервісу та підвищуючи загальний комфорт використання платформи.

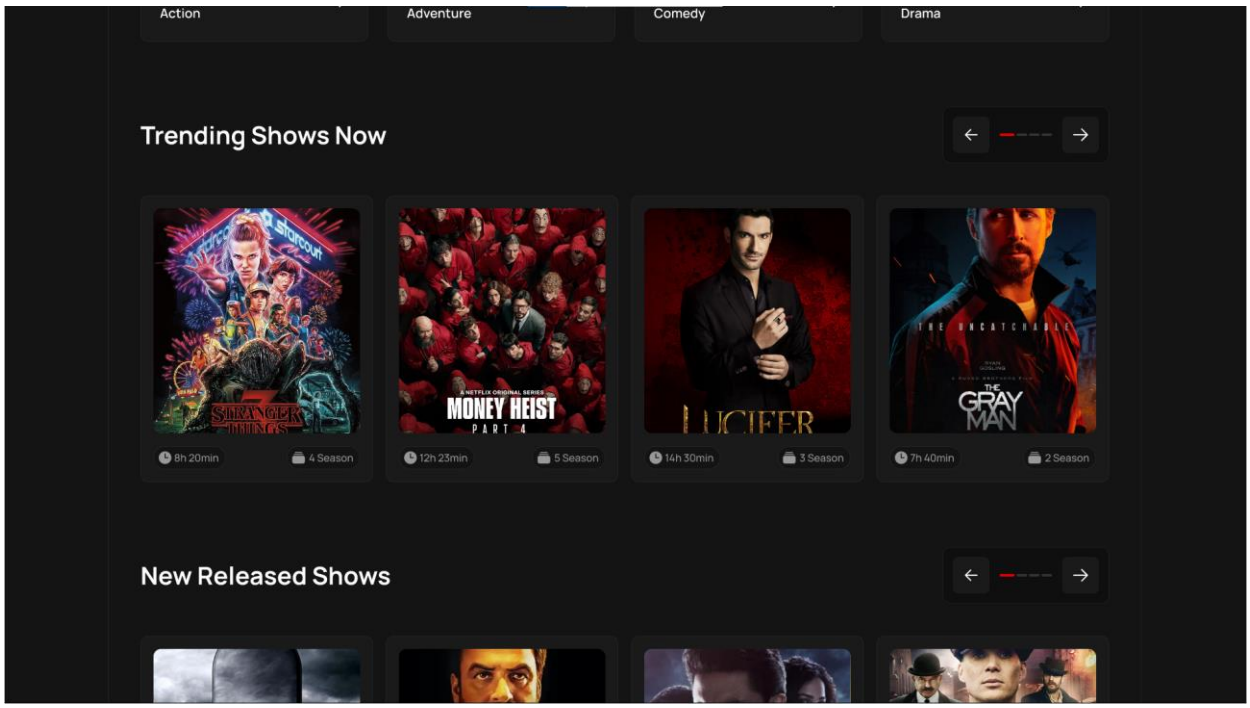

Рисунок 4.4 – Список фільмів та серіалів

Після того як користувач обирає певний фільм або серіал, він переходить на сторінку перегляду детальної інформації про обраний контент. На цій сторінці відображаються основні візуальні елементи: постери та зображення з фільму або серіалу, що дозволяє користувачу краще ознайомитися з атмосферою твору; представлена структурована інформація: назва, опис, жанр, рейтинг, тривалість, акторський склад, дата виходу тощо.

В залежності від типу контенту (серіал чи фільм), сторінка динамічно адаптується:

- для фільмів відразу доступна кнопка для перегляду (рисунок 4.5);
- для серіалів користувачу надається список сезонів та серій, які він може переглядати послідовно або вибірково (рисунок 4.6).

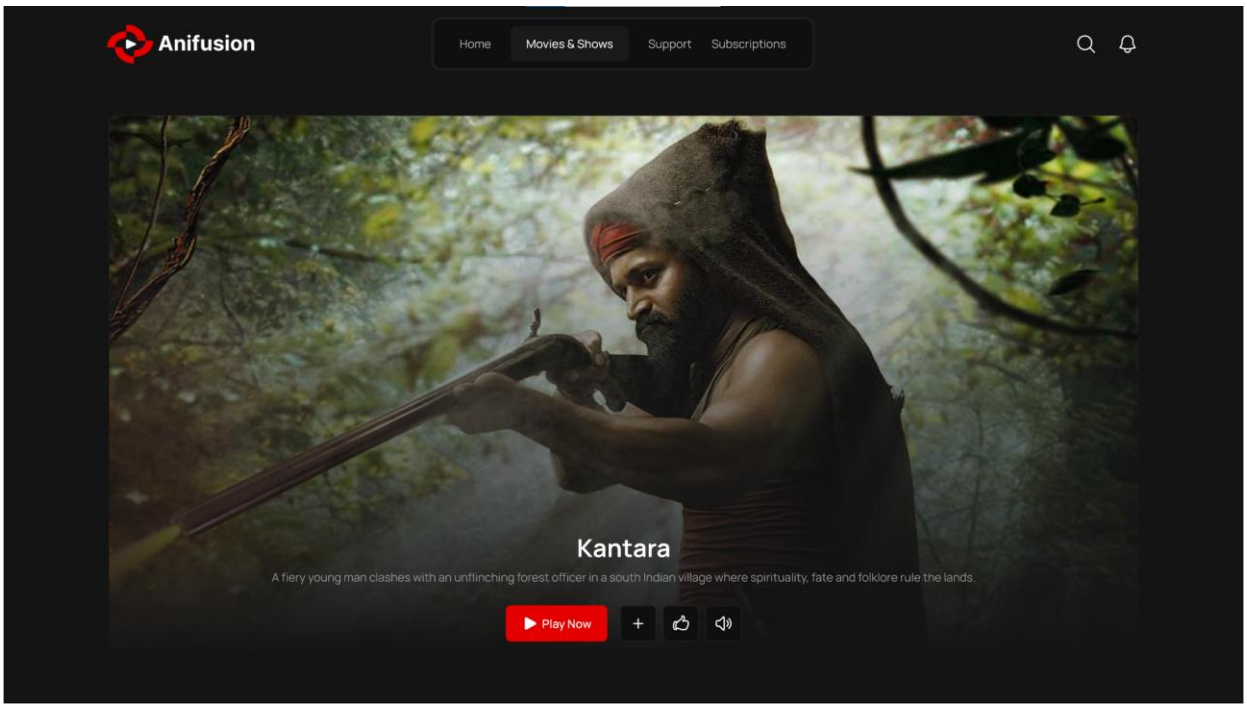

Рисунок 4.5 – Постер для фільмів та серіалів

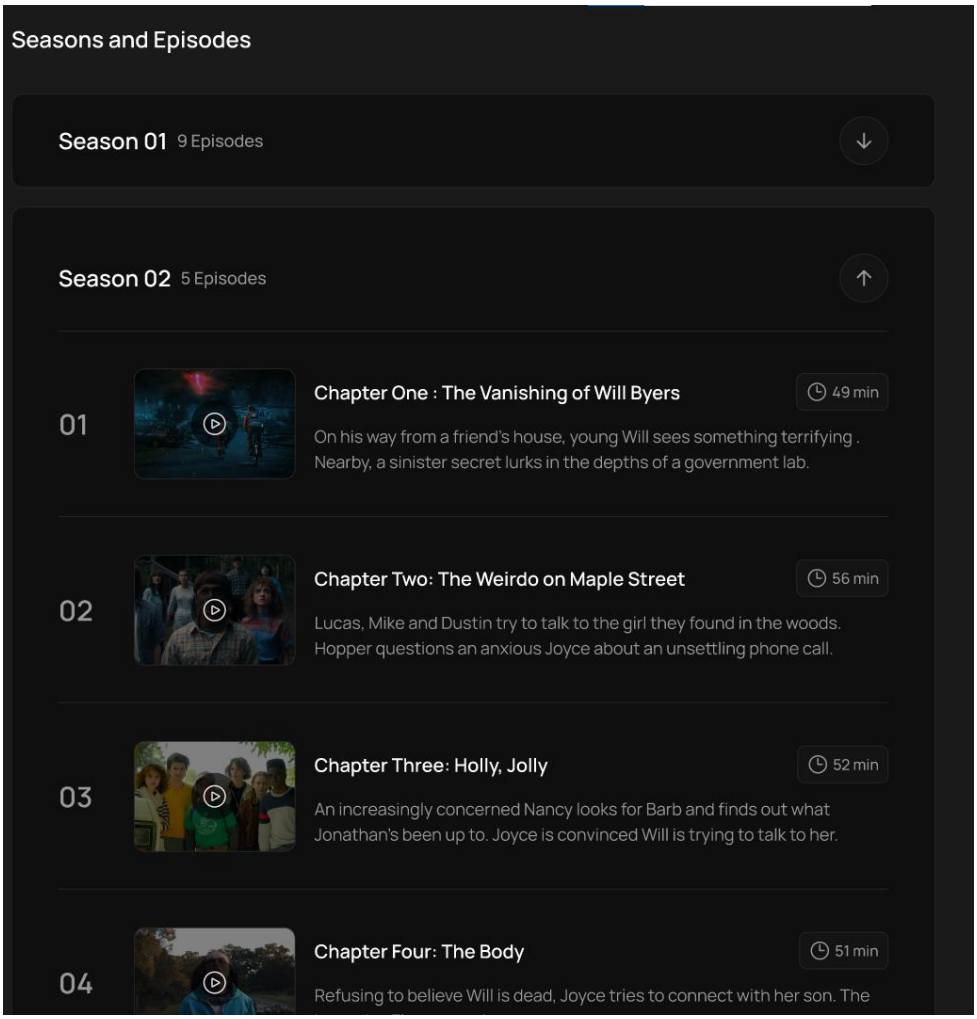

Рисунок 4.6 – Список серій

Нижче основного блоку інформації розташований розділ коментарів, де користувачі можуть переглядати думки інших глядачів або залишити власний коментар (рисунок 4.7). Це сприяє формуванню активної спільноти навколо контенту та підвищує залученість аудиторії до платформи.

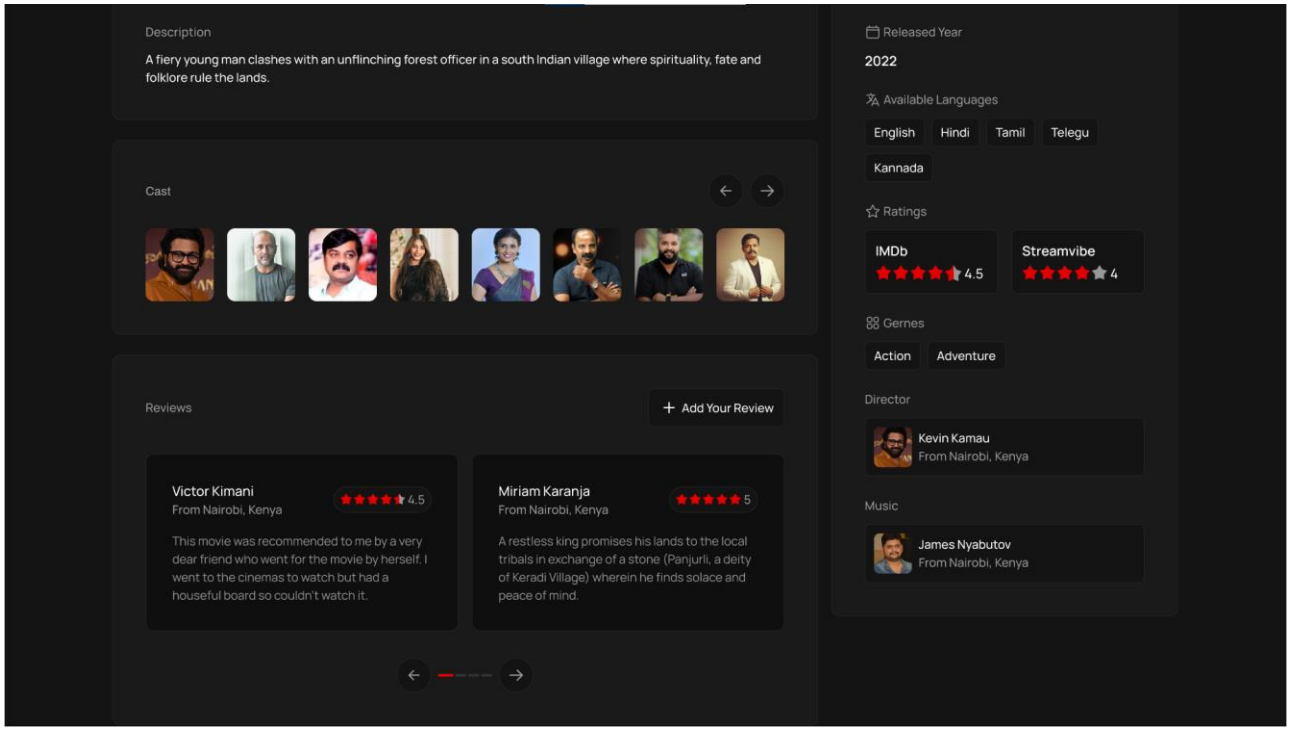

Рисунок 4.7 – Коментарі під відео

ВИСНОВКИ

Сучасна цифрова епоха висуває нові вимоги до способів споживання інформації та розваг. Пандемія стала каталізатором змін у поведінці користувачів, пришвидшивши перехід від традиційних форматів таких як кінотеатри до онлайн-середовищ. У цих умовах ідея створення стримінгового сервісу постала як цілком логічна відповідь на запити часу: доступність, комфорт, персоналізація та незалежність від фізичних обмежень.

Розробка власного вебсервісу для перегляду фільмів і серіалів дозволяє не лише задовольнити сучасні запити користувачів, а й створити конкурентну альтернативу вже існуючим платформам. Стрімкий розвиток хмарних технологій відкрив нові можливості для ефективного зберігання та доставки великого обсягу відеоконтенту. Застосування моделі підписки також забезпечує сталі джерела доходу та підтримує взаємодію з аудиторією.

У рамках цього проєкту було реалізовано повноцінний стримінговий сервіс, який дозволяє користувачам переглядати фільми та серіали онлайн з можливістю вибору мови аудіодоріжки, взаємодії з контентом через систему коментарів, оформлення підписок та персоналізації досвіду. Система побудована за мікросервісною архітектурою, що забезпечує розподіл відповідальностей, масштабованість та гнучкість у подальшому розвитку. Кожен мікросервіс виконує окрему функцію: від автентифікації користувачів до обробки відеопотоків та платежів.

Технологічний стек включає FastAPI, PostgreSQL, RabbitMQ, Docker, MinIO, React, а також клієнтську частину на React із сучасним та адаптивним інтерфейсом. Використання хмарного підходу дозволяє ефективно обробляти запити, масштабувати ресурси та забезпечувати високу доступність сервісу.

Таким чином, ідея проєкту є не просто технічно актуальною, а й соціально значущою – вона формує новий підхід до культури перегляду, сприяє розвитку цифрової інфраструктури, підтримує незалежних творців і повністю відповідає вимогам сучасного медіасередовища.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Tiangolo S. FastAPI Documentation. URL: https://fastapi.tiangolo.com.

2. Richardson L., Ruby S. RESTful Web APIs. O’Reilly Media, 1st Edition, 2013. 320 с.

3. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 560 с. (раздел «Repository Pattern»)

4. Rousseau D. Mastering SQLAlchemy. Packt Publishing, 2021. 370 с.

5. Merkel D. Docker: Up & Running: Shipping Reliable Containers in Production. O'Reilly Media, 2nd Edition, 2020. 332 с.

6. Joshi D. NGINX Cookbook: Over 70 practical recipes to design, deploy, and optimize NGINX-based applications. Packt Publishing, 2019. 276 с.

7. Videla A., Williams J. RabbitMQ in Action: Distributed Messaging for Everyone. Manning Publications, 2012. 320 с.

8. Amazon Web Services. AWS Documentation. URL: https://docs.aws.amazon.com

9. Auth0 Team. JWT Handbook. Auth0, 2022. URL: https://jwt.io/introduction

10. Abramov D., Clark A. The Road to React: Your Journey to Master Plain Yet Pragmatic React.js. Leanpub, 2019. URL: https://www.roadtoreact.com/