ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО «ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «МІЖРЕГІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ ПЕРСОНАЛОМ»

Факультет комп'ютерно-інформаційних технологій
Кафедра комп'ютерних інформаційних систем і технологій

Чипігін Василь Сергійович

КВАЛІФІКАЦІЙНА РОБОТА

на тему: **«Розробка веб-застосунку для організації потокових трансляцій та взаємодії з аудиторією»**

Група: ІК-9-22-Б1ІПЗ(4.0д)

Спеціальність: Інженерія програмного забезпечення

Рівень вищої освіти: перший (бакалаврський)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів, текстів інших авторів мають посилання на відповідне джерело.

_____ __Чипігін Василь Сергійович____
(підпис) ПІБ (студента)

Науковий керівник

_____ __Дуднік Андрій Сергійович_____

(підпис)

Допущено до захисту ЕК

Завідувач кафедри _____ Сергій КАВУН , д.е.н., професор
(підпис)

Київ 2026

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 69 с., 22 рис., 2 табл., 1 додаток, 27 джерел.

ВЕБЗАСТОСУНОК ДЛЯ ВІДЕОТРАНСЛЯЦІЙ, ПОТОКОВЕ ВІДЕО, .NET, VERTICAL SLICE ARCHITECTURE, GRAPHQL, WEBSOCKETS, RTMP, HLS, POSTGRESQL, AWS S3.

Об’єкт дослідження – технології передачі живого відео та обміну повідомленнями. Мета роботи – створити швидку вебплатформу для проведення ефірів з мінімальними затримками. Методи розробки. Бекенд написано на .NET за принципами Vertical Slice Architecture. Великий акцент зроблено на тестуванні коду. Сервер приймає відеопотік через RTMP, а транслює клієнтам по HLS. Звичайні дані зберігаються у PostgreSQL. Важкі відеофайли винесено у хмару AWS S3. Для оптимізації запитів фронтенду впроваджено GraphQL замість класичного REST. Чат працює миттєво завдяки WebSockets. Платіжну систему Stripe підключено через вебхуки. Результат роботи. Створено повністю функціональну платформу для трансляцій. Вона стабільно приймає відео, адаптує якість під інтернет користувача та безпечно обробляє донати.
Ключові слова: РОЗРОБКА ВЕБЗАСТОСУНКУ, ПОТОКОВЕ ВІДЕО, .NET, VERTICAL SLICE ARCHITECTURE, GRAPHQL, WEBSOCKETS, RTMP, HLS.

WEB APPLICATION FOR VIDEO BROADCASTING, LIVE STREAMING, .NET, VERTICAL SLICE ARCHITECTURE, GRAPHQL, WEBSOCKETS, RTMP, HLS, POSTGRESQL, AWS S3.

The object of research is the technologies for live video transmission and real-time messaging. The aim of the work is to create a fast streaming web platform with

minimal latency. The development methods. The backend is written in .NET using Vertical Slice Architecture principles. A strong emphasis is placed on code testing. The server ingests video via RTMP and delivers it to clients via HLS. Regular data is stored in PostgreSQL. Heavy video files are offloaded to AWS S3 cloud storage. To optimize frontend requests, GraphQL is implemented instead of classic REST. The chat works instantly thanks to WebSockets. The Stripe payment system is integrated via webhooks. The result of the work. A fully functional broadcasting platform has been created. It stably receives video, adapts quality to the user's internet connection, and securely processes donations.

Key words: WEB APPLICATION DEVELOPMENT, LIVE VIDEO, .NET, VERTICAL SLICE ARCHITECTURE, GRAPHQL, WEBSOCKETS, RTMP, HLS.

ЗМІСТ

ВСТУП 6
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 8
1.1 Особливості організації потокового відеомовлення та огляд протоколів передачі медіаданих 9
1.2 Огляд та порівняльний аналіз існуючих платформ і систем потокового мовлення 11
1.3 Аналіз архітектурних патернів побудови сучасних веб-платформ 13
1.4 Підходи до організації зберігання структурованих даних та великих медіа-файлів 16
1.5 Аналіз технологій забезпечення інтерактивності та комунікації в реальному часі 18
1.6 Огляд сучасних підходів до автентифікації та захисту медіаплатформ 19
1.7 Аналіз методів монетизації контенту та інтеграції платіжних шлюзів 21
1.8 Аналіз підходів до проєктування програмних інтерфейсів (API) та оптимізації мережевої взаємодії 23
1.9 Висновки до першого розділу 25
РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБ-ЗАСТОСУНКУ ДЛЯ ТРАНСЛЯЦІЙ 25
2.1 Функціональні та нефункціональні вимоги до системи 26
2.2 Сценарії використання системи (Use Cases) 27
2.3 Проєктування архітектури застосунку 34
2.4. Проєктування моделі бази даних та структури зберігання контенту 36
2.4.1 Логічна та фізична модель реляційної бази даних 37
2.4.2 Структура зберігання контенту (AWS S3) 40
2.5 Висновки до другого розділу 40
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОНКУ 42
3.1. Мова програмування та платформа розробки C# / .NET. 43
3.2 Аналіз підходів до проєктування програмних інтерфейсів (REST vs GraphQL) 44
3.3 Платформа для вебінтерфейсу (Next.js / React) 44
3.4 Бази даних (PostgreSQL vs NoSQL) 45
3.5 Організація фонових задач (RabbitMQ) 46
3.8 Розгортання системи (Docker та Docker Compose) 47
3.9 Реалізація серверної частини та API на платформі .NET з використанням

GraphQL 48
3.10 Розробка клієнтського інтерфейсу на базі Next.js та інтеграція з Auth0 .. 49
3.11 Налаштування медіа-сервера MediaMTX 51
3.12 Реалізація сервісу обробки відео (VOD Processor) та інтеграція з AWS S3 51
3.13 Висновки до третього розділу 52
РОЗДІЛ 4. ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКУ 53
4.1 Юніт-тестування 54
4.2 Інтеграційне тестування взаємодії з базою даних 55
4.3 Верифікація нефункціональних вимог та системний аналіз 55
4.4 Висновки до четвертого розділу 57
РОЗДІЛ 5. РОБОТА КОРИСТУВАЧА З СИСТЕМОЮ 58
5.1 Автентифікація та доступ до платформи 58
5.2 Запуск трансляції 59
5.3. Платні підписки 62
5.4 Висновки до п'ятого розділу 64
ВИСНОВКИ 66
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 67
ДОДАТОК А 70

ВСТУП

Зараз важко знайти сферу, де б не використовували прямі ефіри. Сьогодні їх використовують для онлайн-освіти, бізнес-конференцій та навіть продажу товарів у реальному часі. За статистикою, відеоконтент — це вже понад 80% всього, що споживається в інтернеті [1].

Користуватися готовими платформами стає дедалі складніше через їхню корпоративну політику. Високі комісії, коли доводиться віддавати корпорації половину зароблено, раптове безпідставне блокування акаунтів без пояснень та закриті алгоритми — усе це стимулює розробників до створення власних self-hosted рішень. Власна платформа — це не просто «свій сайт з відеоконтентом», це повний контроль над даними, монетизацією та цільовою аудиторією.

Розробити такий сервіс — складно. Потрібно розв'язати проблему затримок, налаштувати передачу гігабайтів відео, щоб сервер витримував великі навантаження.

Використання сучасного стека, як-от .NET для логіки, GraphQL для гнучких запитів та AWS S3 для хмарного зберігання, дозволяє зібрати систему, яка буде працювати стабільно та масштабуватися. Поєднання цих складних інженерних задач і робить тему роботи актуальною для сучасного ринку.

Головна мета — спроєктувати та розробити вебзастосунок для трансляцій, який би автоматично обробляв відео, зберігав, дозволяв користувачам спілкуватися в реальному часі та мав прозору систему донатів.
Щоб дійти до результату, робота розбита на такі кроки:

- Дослідити теорію та протоколи. Потрібно детально розібрати роботу протоколів RTMP для прийому відео та HLS для його доставки. Також проаналізувати, як взагалі архітектурно будуються сучасні медіасервери.
- Продумати архітектуру системи. Вирішити, як саме основний бекенд на .NET буде спілкуватися з окремими сервісами, що відповідають за ресурсомістке транскодування відео. Це важливо, щоб не перевантажити

головний сервер.

- Розробити бекенд. Тут вирішено використати GraphQL, щоб фронтенд міг запитувати виключно необхідні дані. Це суттєво зменшить навантаження на мережу.
- Налаштувати роботу з відеопотоком. Треба реалізувати механізм, який приймає сигнал, у реальному часі його перекодовує і відразу відправляє готові записи ефірів (VOD) у хмарне сховище AWS S3.
- Розробити фронтенд. Основна задача — зробити зручний інтерфейс із плеєром та інтерактивним чатом. Чат має працювати в режимі реального часу, щоб повідомлення доходили з мінімальною затримкою.
- Налаштувати безпеку та платежі. Для надійної авторизації користувачів підключити сервіс Auth0, а також інтегрувати платіжний шлюз для обробки донатів від глядачів.
- Провести тестування. Перевірити правильність роботи всієї логіки, від трансляцій до фінансових транзакцій, і переконатися, що сервер витримує навантаження під час активних ефірів.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Предметна область проєкту фокусується на технологіях передачі потокового медіа (Video Streaming) та системах для інтерактивного зв'язку в режимі реального часу. Цифровізація кардинально змінила споживання контенту: класичні медіа формати фактично розчинилися в інтернет-платформах. Ефіри зараз інтегрувалися в інтернет-платформи — від онлайн-лекцій до корпоративних платформ і комерційних ефірів[1].

З погляду інженерної реалізації, мовлення — це безперервний технологічний процес. Процес ініціюється на боці джерела (зазвичай це програмне забезпечення, наприклад, OBS), яке безпосередньо передає медіапоток на сервер. Ключовим технічним викликом на цьому етапі є необхідність транскодування потоку в реальному часі відповідно до пропускної здатності мережі клієнта. Ця процедура є обов'язковою для того, щоб у користувача з низькою швидкістю з'єднання трансляція не переривалася через постійну буферизацію, у той час як користувачі з високошвидкісним доступом отримували відео в максимальній якості.

Окрім прямої роздачі сигналу, система має паралельно розв'язувати ще кілька задач:

- Забезпечення мінімальної затримки (low latency), щоб автор трансляції міг спілкуватися з аудиторією.
- Автоматичний запис ефірів та їх конвертація у файли для розділу VOD (Video on Demand).
- Зберігання великих масивів даних у розподілених сховищах.

Однак відео без інтерактивної взаємодії сьогодні втрачає актуальність. Глядачеві необхідний швидкий чат і система миттєвих реакцій чи мікротранзакцій. Для інженера це означає роботу з протоколом WebSockets, де потрібно підтримувати тисячі активних з'єднань одночасно, щоб сповіщення з’являлися на екрані без жодних затримок[2].

На сучасному етапі розвитку індустрії спостерігається стійка тенденція до децентралізації. Популярні платформи накладають на творців контенту суворі

обмеження: високі комісії, цензура та ризик видалення каналу без належного обґрунтування. Це закономірно стимулює попит на автономні (self-hosted) рішення. Власна платформа дає бізнесу чи незалежним розробникам переваги, недоступні в межах глобальних пропрієтарних сервісів — повний контроль над даними, архітектурою та фінансовими потоками. Розробка такої незалежної та відмовостійкої системи виступає головним завданням даного дипломного проєкту.

1.1 Особливості організації потокового відеомовлення та огляд протоколів передачі медіаданих

Технічно потокове відеомовлення (Live Streaming) кардинально відрізняється від звичайного завантаження файлів з інтернету. Тут немає можливості завантажити контент наперед. Дані мають передаватися безперервно від джерела до клієнта, і головна інженерна проблема — як мінімізувати затримку (latency), паралельно адаптуючи якість картинки під нестабільну мережу глядача.

Архітектуру будь-якої платформи потокового мовлення зазвичай ділять на три базові етапи:

- Ingest (Захоплення та передача). Це старт процесу. Програма на комп'ютері автора трансляції (наприклад, той самий OBS Studio) захоплює відео і відправляє його на вхідний вузол сервера.
- Processing (Обробка). Медіасервер отримує потік, транскодує його, змінює формати контейнерів і нарізає на дрібні сегменти для подальшої передачі та запису.
- Delivery (Доставка). Оброблені дані розподіляються між тисячами клієнтів. Щоб сервер не впав від навантаження, тут часто підключають CDN (мережі доставки контенту) або хмарні сховища.

Кожен із цих етапів вимагає свого протоколу. Для прийому відео на сервер (Ingest) індустрія досі масово використовує RTMP (Real-Time Messaging

Protocol). Цей протокол вважається старим, і сучасні браузери припинили його нативну підтримку. Але для зв'язку між автором трансляції та сервером він працює ефективно. RTMP базується на протоколі TCP, що гарантує надійність з'єднання, тримає затримку на рівні 2–5 секунд, є підтримку професійним бродкаст програмами (OBS, vMix тощо)[3].

Транслювати відео глядачам (етап Egress) через RTMP вже неефективно. Для доставки сьогодні стандартом є HLS (HTTP Live Streaming). Його механіка базується на тому, що безперервний потік розбивається на короткі файли (chunks) по кілька секунд. Клієнтський плеєр завантажує маніфест-файл (.m3u8), де вказано порядок цих шматків. Оскільки це звичайний HTTP-трафік, такі фрагменти дуже легко кешувати на проміжних серверах і зручно завантажувати в хмарні бакети, такі як AWS S3[3].

Значною перевагою HLS — це наявність технології ABR (Adaptive Bitrate Streaming). Якщо у глядача раптово знижається швидкість інтернету, плеєр автоматично підтягує шматки відео з меншою роздільною здатністю, і трансляція не зависає.

Варто також згадати WebRTC. Технологія має основну перевагу тим, що дає затримку менше секунди. Проте масштабувати Peer-to-Peer архітектуру на велику аудиторію технічно важко і фінансово затратним, тому WebRTC частіше залишають для відеоконференцій[4].

Зважаючи на все це, для побудови платформи обрано перевірену гібридну схему. Вхідний потік від авторів трансляції приймається через RTMP, а доставку глядачам реалізується через HLS. Це дає високу стабільність передачі, повну сумісність із

веб-плеєрами та суттєво спрощує роботу з архівами трансляцій (VOD)[4]. Блок-схему гібридної архітектури обробки та доставки потокового відеоконтенту зображено на рис. 1.1.

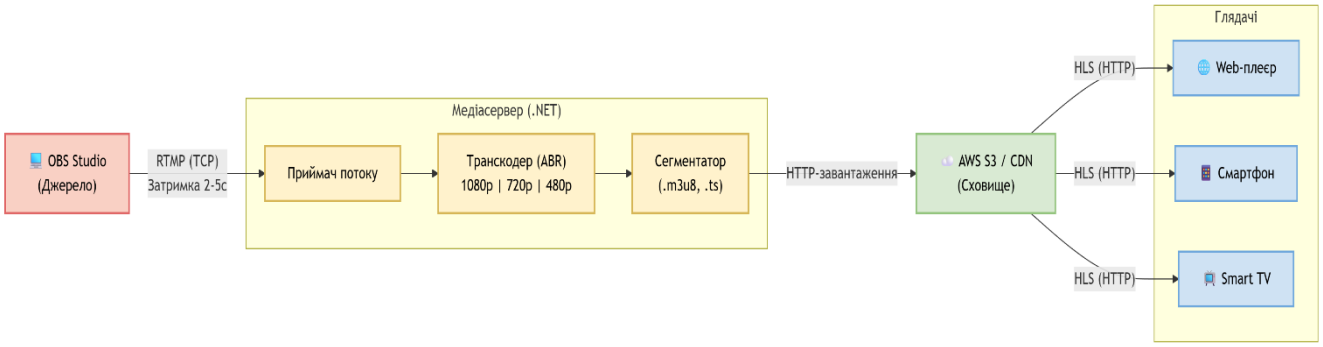

Рисунок 1.1 – Блок-схема архітектури обробки та доставки потокового відеоконтенту

1.2 Огляд та порівняльний аналіз існуючих платформ і систем потокового мовлення

Ринок потокового мовлення зараз жорстко поділений між кількома гігантами. Вони, формують технічні стандарти для всієї індустрії. Щоб сформулювати вимоги до системи, потрібно проаналізувати принцип роботи Twitch, YouTube Live. Також доцільно розібрати сегмент open-source self-hosted платформ на прикладі Owncast.

Twitch.tv сьогодні — це домінуюча платформа у сегменті потокового мовлення. Якщо проаналізувати архітектуру, вони використовують класичну схему: приймають вхідний сигнал по RTMP і транслюють його мільйонам клієнтів за допомогою HLS. Їх суттєва технологічна особливість у наднизькій затримці (Low Latency). Twitch досягає цього за рахунок глибокої кастомізації транспортних протоколів та використання величезної розподіленої інфраструктури. Проте для незалежних розробників архітектура таких платформ залишається непрозорою. Закритість системи позбавляє користувача можливості здійснювати контроль над власними даними та алгоритмами їх обробки[5].

Архітектура YouTube Live має принципові відмінності. Основний акцент у ній зроблено на забезпеченні максимальної якості зображення та безшовній інтеграції прямих трансляцій у глобальну базу відео за запитом (VOD). На відміну від Twitch, тут реалізовано повноцінний функціонал DVR, що дозволяє

глядачеві вільно переміщуватися за часовою шкалою ефіру назад. Технічно така реалізація потребує складних механізмів кешування та управління петабайтами даних.

Як альтернатива комерційним корпораціям існують Self-hosted рішення, серед яких найбільш відомим є Owncast. Це open-source продукт, який можна розгорнути на орендованому VPS-сервері за короткий проміжок часу. Такий підхід забезпечує повну автономність власника, проте створює обмеження на рівні кодової бази. Архітектурно Owncast побудований як класичний моноліт. Відповідно, якщо в процесі розвитку платформи виникне необхідність інтегрувати складну бізнес-логіку або винести сервіс чату на окремі екземпляри (інстанси) для підвищення масштабованості, розробник зіткнеться з серйозними труднощами[7].

Для зручності ключові технічні характеристики розглянутих платформ та проєкту розробки зведені у таблицю 1.1.

Таблиця 1.1 — Порівняльний аналіз існуючих систем та проєктуваного рішення

Критерій порівняння	Twitch	YouTube Live	Owncast
Архітектура	Розподілена (Microservices)	Розподілена (Microservices)	Монолітна
Протокол входу (Ingest)	RTMP	RTMP, HLS	RTMP
Протокол доставки	HLS (Low Latency)	MPEG-DASH, HLS	HLS
Збереження записів	Тимчасове	Постійне	Локальне
Інтерактивність (чат)	IRC / WebSocket	Polling / WebSocket	WebSocket
Складність підтримки	Дуже висока	Дуже висока	Низька

Комерційні платформи є надмірно складними та закритими системами,

тоді як доступні open-source рішення часто мають жорстку структуру, що суттєво ускладнює їхній подальший розвиток. Проведений аналіз обґрунтовує доцільність розробки власного вебзастосунку на базі архітектури модульного моноліту.

Використання підходу Vertical Slice Architecture дозволяє отримати оптимальне поєднання переваг: з одного боку, систему легко розгортати та адмініструвати, з іншого — програмний код розподіляється на незалежні модулі з високим рівнем зв’язності (High Cohesion). Це забезпечує можливість безперешкодного розширення функціональних можливостей у майбутньому, запобігаючи перетворенню проєкту на важкопідтримуваний монолітний код (legacy-code).

1.3 Аналіз архітектурних патернів побудови сучасних веб-платформ

Вибір архітектури — це основа. Якщо помилитися на цьому етапі, подальша підтримка та масштабування системи значно ускладняться. Для платформ, які мають справу з ресурсомістким медіаконтентом у реальному часі, шаблонні підходи часто виявляються неефективними, тому доводиться шукати оптимальні компромісні рішення між продуктивністю та складністю розробки.

Порівняння моноліту та мікросервісів. Класичний моноліт забезпечує швидкий початок розробки: вся логіка в одному місці, компоненти спілкуються між собою миттєво, а дебажити такий код — значно простіше. Але коли проєкт розростається, він перетворюється на надмірно складну та монолітну структуру. Будь-яка критична помилка в одному модулі призводить до критичної відмови всієї системи. Для потокового мовлення використання виключно монолітної архітектури є недоцільним. Це позбавляє можливості виділити додаткові потужності суто під конвертацію відео; доведеться клонувати весь застосунок цілком, що є нераціональним використанням ресурсів.

З іншого боку є мікросервісна архітектура. Це актуально, гнучко і

дозволяє розгортати систему по шматках. Проте для невеликої команди такий підхід є невиправдано складним. Витрати часу на підтримку інфраструктури, боротьба з мережевими затримками між мікросервісами та проблеми із синхронізацією розподілених баз даних приберуть усі переваги[9]. Структурне порівняння монолітного та мікросервісного підходів до побудови архітектури зображено на рис. 1.2.

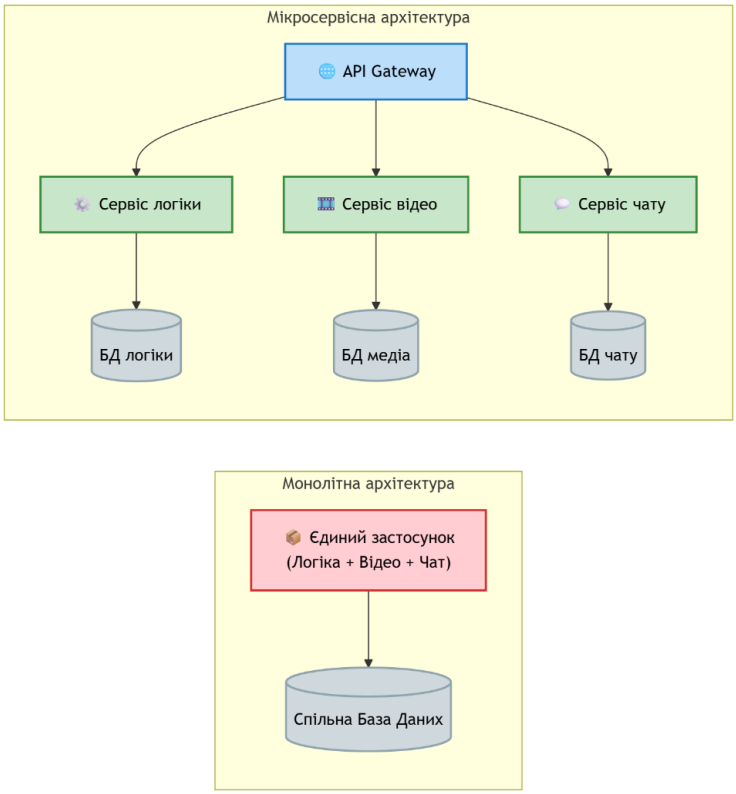

Рисунок 1.2 – Порівняння монолітної та мікросервісної архітектур

Тому найбільш хорошим рішенням для задачі виглядає гібридна (сервіс-орієнтована) архітектура. Ядро системи (де обробляються користувачі, фінанси та загальна бізнес-логіка) залишається модульним монолітом. А от специфічні та найважчі задачі — транскодування відеопотоків, виносяться в повністю ізольовані фонові сервіси (воркери). Вони спілкуються з основним ядром асинхронно, через черги повідомлень. Якщо навантаження зросте, з’являється можливість розгортання нові інстанси воркерів без необхідності модифікації чи зупинки основного сервера. Блок-схему взаємодії компонентів гібридної архітектури зображено на рис. 1.3.

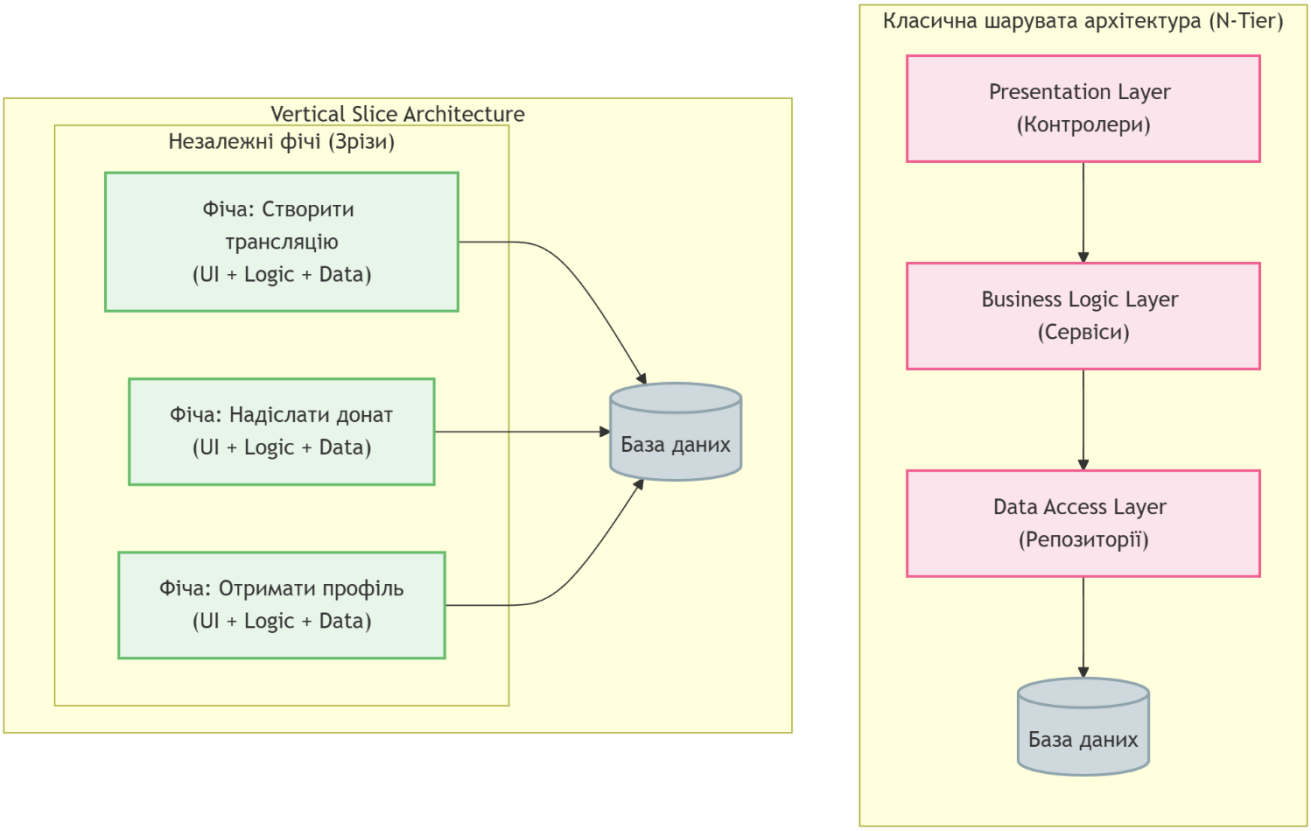

Рисунок 1.3 – Порівняння шаруватої архітектури та Vertical Slice Architecture

Вирішити, як сервіси спілкуються між собою — лише один з аспектів проєктування. Важливо те, як організований код всередині самого ядра. Традиційна шарувата (Layered) архітектура часто змушує розробника модифікувати код в різних місцях (контролери, сервіси, репозиторії), щоб додати одну дрібну фічу[8].

Більш ефективною альтернативою для сучасних систем є Vertical Slice Architecture. Тут здійснюється структурування коду не за технічними шарами, а за конкретним функціоналом. Кожен запит (наприклад, «Створити трансляцію») представлений у вигляді окремого вертикального зрізу. Такий підхід об'єднує всі необхідні компоненти: від валідації вхідних даних до збереження в базу. Це дає високу зв'язність коду (High Cohesion). Якщо треба щось змінити або реалізувати нову функціональну можливість, модифікація виконується локально, що мінімізує ризик виникнення регресійних помилок у системі [8].

Останній аспект — як саме фронтенд буде спілкуватися з бекендом.

Класичний REST-підхід сьогодні часто стає обмежувальним фактором. Виникає або надлишкове отримання величезної кількості непотрібних даних (over-fetching), або клієнту доводиться виконувати серію послідовних запитів, щоб сформувати необхідний стан інтерфейсу (under-fetching) [10].

Тому для цього застосунка обрано GraphQL. Його основна перевага полягає в тому, що клієнт самостійно визначає структуру, які саме дані йому потрібні. За один єдиний мережевий запит фронтенд може отримати профіль автора трансляції, статус його трансляції, кількість підписників та останні мікротранзакції. Це суттєво мінімізує навантаження на мережу і робить інтерфейс високопродуктивним, що для потокового мовлення є критичним показником. Відмінність підходів до вибірки даних між протоколами REST та GraphQL проілюстровано на рис. 1.4.

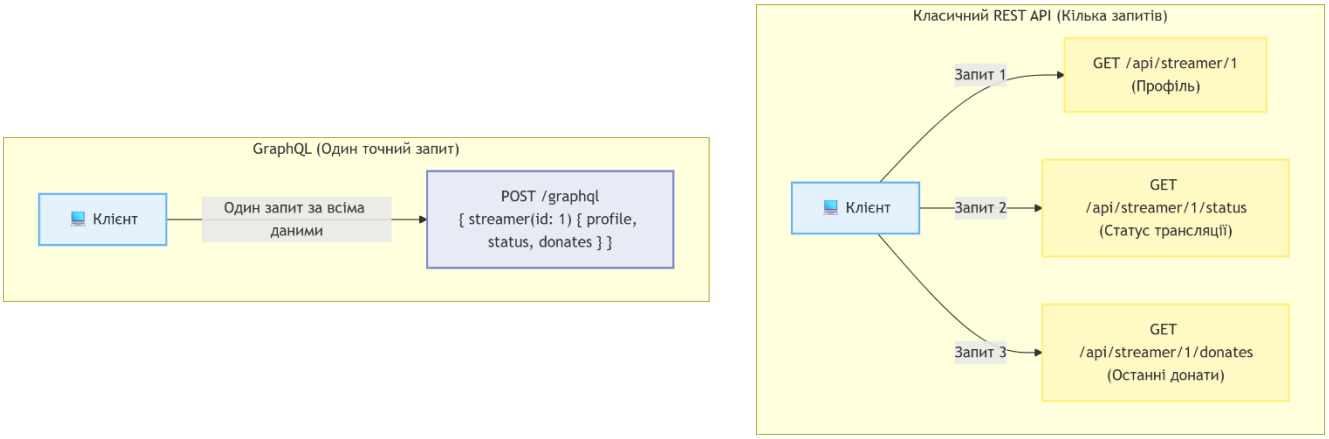

Рисунок 1.4 – Принцип вибірки даних: множинні запити REST проти єдиної кінцевої точки GraphQL

У підсумку маємо такий стек рішень: гібридна архітектура для масштабованості, Vertical Slice для чистоти і підтримки коду, та GraphQL для оптимізованої роботи клієнтської частини.

1.4 Підходи до організації зберігання структурованих даних та великих медіа-файлів

Будь-яка платформа для відеотрансляцій генерує величезну кількість даних. Щоб система витримала великі навантаження, важливо одразу розділити

ці дані на дві категорії. Перша — це структуровані метадані. Друга — безпосередньо важкі медіафайли, тобто BLOB-об'єкти. Зберігати і обробляти їх однаково неможливо, для кожного типу потрібен свій інженерний підхід.

Першочергово розглянемо логіку застосунку. Вона охоплює облікові записи, налаштування трансляцій, історія мікротранзакцій та повідомлення з чату. Для такої інформації на першому місці стоїть транзакційна цілісність (ACID) та можливість швидко виконувати складні запити. Відповідно, оптимальним рішенням є використання перевірених підходів: класичні реляційні СУБД ефективно вирішують подібні задачі. У рамках розробки було обрано СУБД PostgreSQL. Ця система функціонує як єдине джерело істини (Single Source of Truth), щоб гарантувати збереження кожної транзакції та кожного повідомлення [11].

Проблема роботи з відеоконтентом. Зовсім інша ситуація з відеозаписами ефірів (VOD). Зберігати гігабайти відеофайлів безпосередньо в таблицях бази даних у вигляді масивів байтів — є вкрай неефективним підходом. БД стрімко збільшиться до некерованих розмірів, час створення бекапів стане критичним, а паралельне читання об’ємних файлів може призвести до блокування інших процесів. Зберігати медіа локально на диску самого вебсервера — також має суттєві недоліки. У разі виникнення необхідності горизонтально масштабувати бекенд і додати ще кілька екземплярів сервера для розподілу навантаження, файли з першого вузла стануть недоступними для запитів, які потрапили на другий. Тобто система почне зберігати стан (stateful), а це суперечить принципам сучасної хмарної архітектури.

Перехід на об'єктні сховища (S3). Ефективне рішення це — використання хмарних об'єктних сховищ. PostgreSQL зберігає тільки посилання на файл: його ідентифікатор, URL-адресу, розмір та розширення. Самі ж відео відправляються в хмарне сховище, наприклад в AWS S3[12].

Такий підхід повністю вирішує проблему масштабування, адже обсяг хмарного сховища фактично необмежений і зростає сам по собі. До того ж основний бекенд взагалі не витрачає процесорний час на роздачу контенту —

глядачі завантажують відео напряму з AWS або через CDN. Провайдер також самостійно робить реплікацію даних між дата-центрами, тому ризик втратити архівні трансляції мінімальний. З огляду на це, комбінація PostgreSQL для структурованих даних та AWS S3 для мультимедіа є найбільш надійним рішенням для розроблюваної платформи.

1.5 Аналіз технологій забезпечення інтерактивності та комунікації в реальному часі

Будь-яка сучасна платформа потокового мовлення — це не просто трансляція відео. Щоб утримати аудиторію, критично необхідний живий інтерактив: швидкий чат, реакції та миттєві сповіщення. Але якщо спробувати побудувати таку взаємодію на класичній архітектурі «клієнт-сервер» (де ініціатором обміну завжди виступає клієнт) це презведе до слабкої продкутивності. Для режиму реального часу ця парадигма не підходить. Історично першим способом обійти це обмеження був HTTP Polling. Клієнтський застосунок програмували надсилати запити до сервера кожні кілька секунд запити щоб отримати нові дані. Це генерувало надлишковий обсяг мережевого трафіку через постійну передачу об’ємних HTTP-заголовків.

Сервер виконував непродуктивне використання обчислювальних ресурсів навіть тоді, коли в чаті була відсутність активності. Згодом підхід еволюціонував до Long Polling. Механіка змінилася: клієнт відправляє запит, а сервер не повертає порожню відповідь одразу, а утримує з'єднання відкритим до моменту появи нових даних (або закінчення тайм-ауту). Це суттєво зменшило кількість марних запитів, проте ключове архітектурне обмеження залишалося актуальним. Після отримання кожної порції повідомлень доводилося знову ініціювати TCP-з'єднання, що під великим навантаженням швидко вичерпувало ресурси сервеа[15].

Як альтернативу часто розглядають Server-Sent Events (SSE). Цей інструмент ефективно працює, коли потрібен однонаправлений потік інформації (наприклад, стрічка новин, котирування валют чи системні алерти)

через одне довготривале з'єднання. Але чат — це завжди двостороння взаємодія. Глядач одночасно і читає, і активно пише. Використовувати SSE архітектурно недоцільно, оскільки для відправки власних повідомлень користувачу все одно доведеться паралельно генерувати звичайні POST-запити[16]. Саме тому індустріальним стандартом для медіа-чатів є протокол WebSocket. Його логіка кардинально інша: після стартового рукостискання (Handshake) між клієнтом і сервером встановлюється стабільне повнодуплексне (двонаправлене) TCP-з'єднання. Відтепер обидві сторони можуть передавати дані в будь-яку мить. Завдяки відсутності HTTP-заголовків при кожній передачі повідомлення досягається мінімальна затримка (Low Latency) та максимально економне використання мережевого каналу. Проте самої лише підтримки протоколу WebSocket недостатньо. Коли мова йде про трансляції з десятками тисяч одночасних глядачів, сервери необхідно горизонтально масштабувати. Для управління таким трафіком на рівні бекенду застосовують патерн Publish/Subscribe (Pub/Sub). У цій моделі відправник (Publisher) не надсилає повідомлення напряму конкретним користувачам. Він публікує його у відповідний тематичний канал (Topic) — наприклад, у кімнату конкретного ефіру. А внутрішня інфраструктура вже автоматично та паралельно розповсюджує дані всім активним підписникам цього каналу. Такий підхід дозволяє легко додавати нові сервери, ізолює пули з'єднань і гарантує відмовостійкість системи під час масових ефірів[10].

1.6 Огляд сучасних підходів до автентифікації та захисту медіаплатформ

Безпека у платформах виходить далеко за межі звичайної форми логіну. Потрібно максимально жорстко розмежовувати права доступу. Автор трансляції має ексклюзивне право ініціювати ефір та генерувати ключі потоку. Звичайний глядач обмежений споживанням контенту, чатом і мікротранзакціями. Якщо архітектура дозволить цим ролям хоч десь перетнутися, платформа миттєво втратить довіру користувачів.

У класичних монолітах цю проблему вирішували через сесії (Session-

based Authentication). Після перевірки пароля сервер створює у своїй оперативній пам'яті запис і відправляє клієнту Cookie. Проте для високонавантажених систем це рішення неефективне. Сесії роблять архітектуру залежною від стану (*Stateful*). Під час масштабування бекенду та впровадження нових серверних вузлів, виникають ризики з розсинхронізацією. Запит авторизованого глядача може потрапити на новий сервер, який фізично нічого не знає про його сесію. Рішення цієї проблеми через підключення зовнішніх In-Memory баз (на кшталт Redis) лише ускладнює підтримку і робить інфраструктуру дорожчою[17].

Відповідно, зараз індустрія масово використовує автентифікацію на основі токенів. Стандартом де-факто став JSON Web Token (JWT). Це криптографічно підписаний рядок, у який уже зашиті ідентифікатор користувача, його рівень доступу та термін дії. Головна перевага JWT полягає в його повній незалежності від стану (*Stateless*). Серверу більше не потрібно при кожному мережевому запиті звертатися до бази даних чи кешу. Він валідує криптографічний підпис токена за допомогою локального алгоритму. Це кардинально знижує навантаження на систему баз даних і дозволяє масштабувати бекенд без обмежень[18]. Діаграму послідовностей, що демонструє різницю між сесійною (Stateful) та токенною (Stateless) моделями автентифікації, наведено на рис. 1.5.

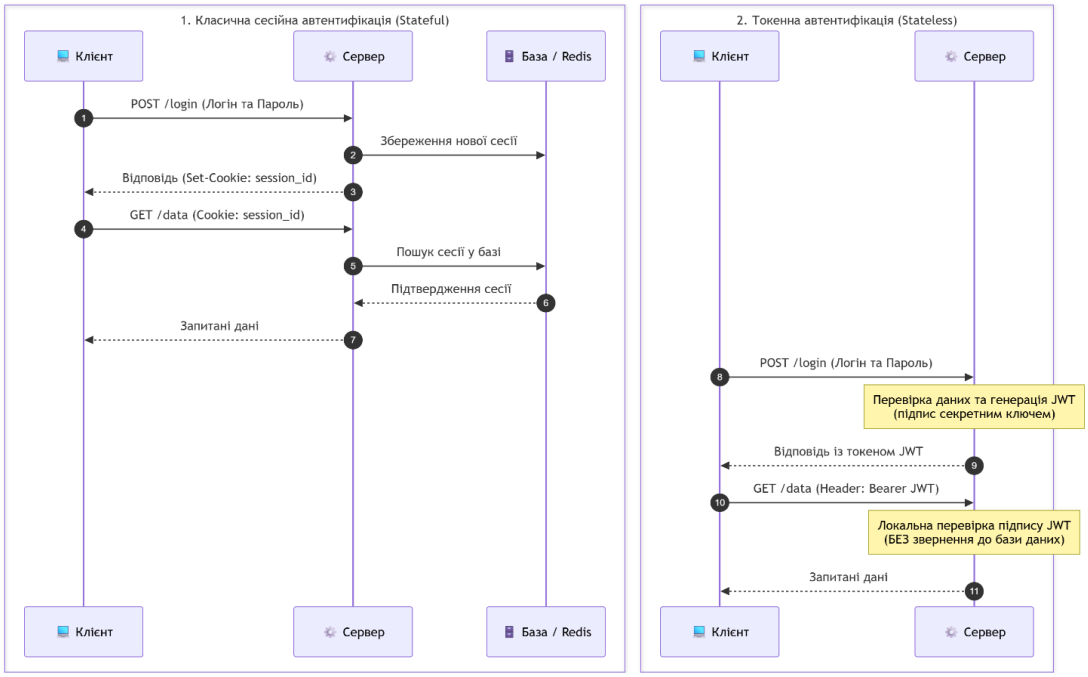

Рисунок 1.5 – Діаграма послідовностей: порівняння сесійної (Stateful) та токенної (Stateless) моделей автентифікації

Разом з тим, самостійна розробка модулів реєстрації та зберігання хешів паролів сьогодні вважається невиправданим ризиком. Це вимагає постійної боротьби з брутфорс-атаками та суворого дотримання нормативів захисту персональних даних. Ефективніше делегувати цей процес спеціалізованим хмарним провайдерам (концепція Identity-as-a-Service). Використовуючи протоколи OAuth 2.0 та OpenID Connect, веб-застосунок взагалі не працює з паролями користувачів. Бекенд виступає виключно в ролі споживача: він отримує від довіреного провайдера готовий JWT-токен і далі працює лише з ним. Це не тільки гарантує високий рівень безпеки, але й суттєво економить час на розробку основної бізнес-логіки платформи[19].

1.7 Аналіз методів монетизації контенту та інтеграції платіжних шлюзів

Будь-яка платформа не може працювати надійно та безпечно системи без надійної системи монетизації. У сучасній економіці авторів контенту фінансова модель зазвичай будується на трьох стовпах: добровільні мікротранзакції прямо під час ефіру, регулярні платні підписки (Subscriptions) та продаж доступу до конкретної події (Pay-Per-View).

З погляду інженерного проєктування, робота з фінансами — це

необхідність суворого дотримання стандарту безпеки PCI DSS. Якщо розробник здійснюватиме збереження нешифрованих даних банківських карток у таблицях власної бази даних, це є критичною архітектурною помилкою (антипатерном). Окрім значних правових ризиків, це створює пряму загрозу компрометації конфіденційних даних.

Тому сьогодні індустрія застосовує підхід делегування процесів зовнішнім платіжним шлюзам (Payment Gateways). Принцип функціонування базується на токенізації: коли глядач ініціює фінансову транзакцію (донат), фронтенд відправляє реквізити його картки безпосередньо на сервери платіжного провайдера. Бекенд не має доступу до цих конфіденційних даних. Натомість платіжний шлюз генерує та повертає безпечний криптографічний токен. Сервер використовує виключно цей ідентифікатор, щоб ініціювати списання коштів. Такий підхід гарантує безпеку та знімає з платформи необхідність самостійно проходити вимогливу процедуру сертифікації PCI DSS[20].

Оскільки система обслуговує безліч незалежних авторів, виникає необхідність вирішення додаткового завдання — автоматичного розподілу коштів (спліт-платежі). Сучасні API дозволяють розщеплювати транзакцію у реальному часі: основна сума одразу перераховується на верифікований рахунок автора трансляції, а комісійний відсоток автоматично утримується на балансі самої платформи.

Важливим технічним аспектом є те, що обробка платежів за своєю природою є асинхронною операцією. Транзакція часто вимагає перевірки від банку-емітента (наприклад, підтвердження через 3D Secure). Сервер не повинен блокувати потік виконання, чекаючи на відповідь. Тому для синхронізації використовуються Webhooks (вебхуки). Як тільки банк підтверджує успішність транзакції, платіжний провайдер самостійно ініціює POST-запит на заздалегідь підготовлений endpoint серверної частини застосунку. Після обробки цього сповіщення, сервер оновлює статистику в базі даних і миттєво відправляє подію через WebSockets [21]. Саме завдяки цьому ланцюжку аудиторія бачить

анімацію отриманого донату поверх відео без потреби перезавантажувати сторінку.

1.8 Аналіз підходів до проєктування програмних інтерфейсів (API) та оптимізації мережевої взаємодії

Традиційна архітектура, за якої сервер самостійно генерує та віддає готовий HTML-код, втратила свою ефективність для сучасних масштабованих проєктів. Головною причиною стало значне ускладнення бізнес-логіки та необхідність підтримувати інтеграцію з різними типами пристроїв, включаючи мобільні додатки і Smart TV. В умовах такого навантаження використання монолітного підходу сильно обмежує гнучкість системи та ускладнює її підтримку. Тому стандартом розробки став перехід до незалежної архітектури з повним логічним відокремленням бекенду від клієнтської частини.

Єдиною сполучною ланкою між цими підсистемами є API (Application Programming Interface). Фактично, він виступає як формалізований контракт. Він чітко визначає, які дані клієнт може запросити, у якому форматі отримає відповідь, і які операції (наприклад, старт ефіру чи відправлення повідомлення в чат) може ініціювати. Завдяки API створюється універсальне серверне ядро. Воно функціонує незалежно від того, хто саме до нього підключається і на якій платформі реалізований клієнтський застосунок. Структурну модель взаємодії різноманітних клієнтських платформ із серверним ядром через єдиний API-контракт зображено на рис. 1.6.

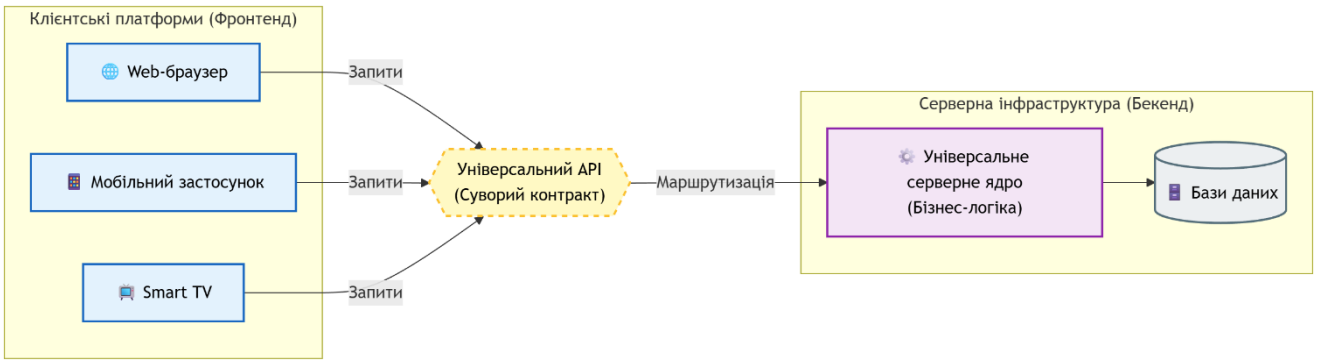

Рисунок 1.6 – Модель відокремлення клієнтських застосунків від бекенду за допомогою єдиного програмного інтерфейсу (API)

Але коли система стає високонавантаженою, проєктування API потребує вирішення складного завдання оптимізації мережевої взаємодії. Інтерфейс трансляції характеризується високою динамічністю. Щоб сформувати інтерфейс сторінки прямого ефіру, клієнтській частині необхідно отримати набір пов'язаних сутностей з бази даних: профіль автора трансляції, лічильник глядачів, статус підписки конкретного користувача, список останніх донатів тощо. У разі використання традиційних підходів з жорстко зафіксованими точками доступу (Endpoints), неминуче виникають дві фундаментальні проблеми:

Нестача даних (Under-fetching). При виконанні запиту до одного ендпоінту повертається лише базова інформація — наприклад, виключно ідентифікатор (ID) автора трансляції. Щоб сформувати повний набір даних для інтерфейсу, клієнт змушений ініціювати каскад додаткових послідовних запитів за іншими адресами (URL). Виникає архітектурна проблема «N+1 запитів». Через постійні мережеві затримки (latency) це суттєво збільшує час завантаження сторінки.

Надлишковість даних (Over-fetching). Це є протилежним архітектурним недоліком. Точка доступу завжди віддає ту структуру даних, яку жорстко визначив розробник бекенду. Наприклад, клієнтській частині для відображення мініатюри відео потрібні лише назва ефіру та нікнейм автора. Але сервер передає весь об'єкт цілком: разом з історією ефірів, довгим описом каналу, посиланнями на соцмережі та іншою інформацією, яка в даний момент є нерелевантною. Це не перевантажує мережевий канал надлишковим обсягом даних, а й змушує сервер непродуктивно використовувати процесорний час та оперативну пам'ять на серіалізацію.

Для подолання зазначених недоліків, сучасні підходи до проєктування передбачають відмову від імперативної маршрутизації і перехід до декларативної вибірки даних. У такій моделі весь API проєктується як один великий граф взаємопов'язаних сутностей. Клієнтській частині надається можливість самостійно формувати запит і явно визначати серверу, які саме

поля та зв'язки їй потрібні для поточної операції. А бекенд агрегує необхідні дані з різних таблиць і повертає їх у вигляді єдиного оптимізованого пакету. Це дозволяє комплексно оптимізувати систему: мінімізує кількість мережевих викликів, повністю усуває проблему надлишкового трафіку і дозволяє клієнтському інтерфейсу рендеритися з високою швидкодією [22].

1.9 Висновки до першого розділу

У процесі аналізу предметної області було визначено базовий технологічний стек та архітектурні патерни для проєктування системи. Відмова від класичної шаруватої архітектури на користь Vertical Slice Architecture обумовлена необхідністю ізолювати бізнес-логіку. Це дозволить в майбутньому масштабувати функціонал і покривати бекенд тестами без ризику зламати суміжні модулі.

Мережеву взаємодію спроєктовано на основі GraphQL. Традиційний REST виявився надто жорстким для складних клієнтських інтерфейсів, постійно провокуючи проблеми надлишкового трафіку. Для завдань, що вимагають миттєвої реакції та двостороннього обміну повідомленнями, задіяно протокол WebSocket.

Як основну базу даних затверджено PostgreSQL через її надійність у роботі з реляційними структурами. Систему автентифікації побудовано на безстановому підході з використанням JWT. Окремо вирішено архітектурну проблему обробки фінансів: замість прямого збереження чутливих даних процесинг платежів делегується зовнішнім шлюзам. Взаємодія з ними відбуватиметься через механізм Webhooks та криптографічні токени, що автоматично закриває питання відповідності стандартам PCI DSS. Затверджені рішення формують достатній інженерний фундамент для переходу до програмної реалізації проєкту.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБ-ЗАСТОСУНКУ ДЛЯ ТРАНСЛЯЦІЙ

Етап формалізації вимог є невід'ємною складовою життєвого циклу

розробки програмного забезпечення, оскільки саме він чітко окреслює межі проєкту та встановлює об’єктивні критерії якості готової системи. Увесь комплекс вимог до розроблюваної платформи відео мовлення структурно поділяється на дві базові категорії: функціональні (визначають бізнес-логіку та доступні операції) та нефункціональні (регламентують атрибути якості, такі як продуктивність, масштабованість і безпека даних).

2.1 Функціональні та нефункціональні вимоги до системи

Функціональні вимоги

Цей розділ описує логіку роботи вебзастосунку та набір операцій, доступних для трьох основних груп: авторів трансляції, глядачів та адміністраторів. Формалізація цих вимог є критично важливою, оскільки вона визначає межі розробки (scope) та дозволяє оцінити якість готового продукту.

Загалом вимоги до платформи відеомовлення поділяються на дві категорії: функціональні (що саме робить система) та нефункціональні (як стабільно вона працює).

- Навігація та пошук: формування списку активних трансляцій, сортування їх за кількістю глядачів та фільтрація за тематиками. Також реалізовано пошук каналів за назвою або нікнеймом.
- Керування доступом: реєстрація та логін через Auth0. Користувачі можуть редагувати профілі (імена, аватари), а система розмежовує права за ролями: від звичайного гостя до платного підписника.
- Організація ефірів: генерація Stream Key для підключення через OBS або vMix. Система автоматично створює сесію ефіру в базі при отриманні сигналу та дозволяє змінювати метадані (назву, категорію) без «дропу» з'єднання.
- Інтерактивні функції: чат з мінімальним пінгом, збереження історії чату для синхронного перегляду в записах (VOD) та система фоловінгів для сповіщень про нові ефіри.
- Монетизація: оформлення платної підписки, надання преміум-доступу до

закритих ефірів та автоматичний контроль активності підписки після завершення терміну оплати.

- Робота з медіа: прийом сигналу по RTMP, транскодування в HLS, автоматичний запис ефірів та вивантаження готових файлів у хмарне сховище AWS S3.

Нефункціональні вимоги

Ці вимоги задають технічні рамки, що гарантують стабільність платформи під навантаженням:

- Продуктивність: затримка ефіру в межах 15–20 секунд (стандарт HLS). Каталог має завантажуватися до 1 секунди, а повідомлення в чаті — долітати швидше, ніж за 0.5 секунди.
- Масштабованість: можливість додавати окремі сервери-воркери для обробки відео незалежно від основного бекенду. Обсяг відео в S3 не повинен впливати на швидкість роботи бази даних.
- Надійність: механізм «реконнекту» для відновлення ефіру при збоях мережі у автора трансляції та гарантоване збереження даних про всі фінансові операції.
- Безпека: обов'язкове використання HTTPS, захист ключів трансляції від перехоплення та валідація JWT-токенів для кожного запиту до API.

2.2 Сценарії використання системи (Use Cases)

Таблиця 2.1 – Сценарії використання.

Назва прецеденту	Прецедент UC-01: Реєстрація та ініціалізація профілю користувача
Опис	Процес автентифікації користувача через зовнішній провайдер ідентифікації (Auth0) із подальшим асинхронним створенням облікового запису в локальній базі даних та налаштуванням унікального імені (нікнейму).

Актори	Неавторизований користувач (Гість), Зовнішній провайдер ідентифікації (Auth0).
Передумови	Користувач не має активної сесії. Сервіси Auth0 та брокер повідомлень RabbitMQ функціонують у штатному режимі.
Базовий потік подій	1. Гість ініціює процес входу через відповідний елемент інтерфейсу на вебсторінці. 2. Система виконує перенаправлення на захищений шлюз авторизації Auth0. 3. Гість проходить процедуру автентифікації (за допомогою електронної пошти або інтеграції з Google/GitHub). 4. Auth0 здійснює валідацію облікових даних та генерує токен доступу. 5. Auth0 формує подію успішної реєстрації та передає її до брокера повідомлень RabbitMQ. 6. Внутрішній сервіс платформи обробляє повідомлення з черги та формує первинний запис у базі даних. 7. Система відображає форму для встановлення унікального нікнейму (оскільки Auth0 передає виключно технічний ідентифікатор). 8. Користувач вводить бажаний нікнейм і підтверджує дію. 9. Система фіксує нікнейм у базі даних та завершує процес ініціалізації профілю.
Альтернативні потоки	1. Крок 3: Гість перериває процес автентифікації. Система повертає його на головну сторінку, сесія

	не створюється. 2. Крок 8: Введений нікнейм вже існує в системі. Процес призупиняється з виведенням повідомлення про помилку валідації, система очікує введення коректних даних.
Виняткові ситуації	1. Відсутність зв'язку зі шлюзом Auth0 (помилка тайм-ауту). - Збій обробки повідомлення на рівні RabbitMQ (автентифікація успішна, проте локальний профіль не створено; вимагається втручання механізму повторних спроб).
Постумови	Користувач отримує статус авторизованого, його профіль збережено в базі даних та синхронізовано з провайдером ідентифікації.

Назва прецеденту	Прецедент UC-02: Налаштування метаданих та ініціалізація трансляції
Опис	Редагування інформації про канал (назва, категорія) та безпосередній запуск медіапотоку за допомогою зовнішнього програмного забезпечення для проведення трансляції.
Актори	Авторизований користувач (Автор трансляції).
Передумови	Користувач має активну авторизовану сесію та права на проведення трансляцій.
Базовий потік подій	1. Автор трансляції переходить до розділу управління трансляцією (Dashboard). 2. Вводить нову назву ефіру та обирає відповідну тематичну категорію з довідника. 3. Підтверджує збереження оновлених метаданих.

	4. Копіює унікальний криптографічний ключ трансляції (Stream Key) з панелі керування. 5. Інтегрує скопійований ключ у налаштування зовнішнього медіакодера (наприклад, OBS Studio) та ініціює передачу відеосигналу. 6. Медіасервер системи фіксує вхідний сигнал за протоколом RTMP/SRT. 7. Система автоматично оновлює статус каналу на «В ефірі» (Live) та активує розсилку сповіщень.
Альтернативні потоки	Відсутні.
Виняткові ситуації	1. Використання недійсного або скомпрометованого ключа трансляції (медіасервер відхиляє з'єднання на етапі "рукостискання"). - Втрата мережевого з'єднання з боку автора трансляції під час ініціалізації.
Постумови	Відеопотік стає доступним для перегляду на платформі, користувачі отримують відповідні системні сповіщення.

Назва прецеденту	Прецедент UC-03: Глобальний пошук контенту
Опис	Здійснення наскрізного пошуку за допомогою єдиного інтерфейсу запитів із паралельною фільтрацією профілів та тематичних категорій.
Актори	Будь-який користувач (Гість або Авторизований).
Передумови	Підсистема баз даних функціонує у штатному режимі.
Базовий потік подій	1. Користувач фокусується на елементі пошуку в навігаційній панелі.

	2. Вводить текстовий пошуковий запит. 3. Система ініціює паралельні транзакції для пошуку збігів у таблицях профілів (за нікнеймом) та категорій (за назвою). 4. Інтерфейс відображає структурований список результатів, згрупований за типом сутності. 5. Користувач обирає релевантний елемент зі списку. 6. Система виконує маршрутизацію на цільову сторінку.
Альтернативні потоки	1. Крок 3: За введеним критерієм не знайдено жодного запису. Система виводить стандартизоване повідомлення про відсутність результатів.
Виняткові ситуації	1. Перевищення ліміту часу на виконання запиту до БД (тайм-аут через пікове навантаження на сервер).
Постумови	Користувач успішно перенаправлений на сторінку обраного профілю або ігрової категорії.

Назва прецеденту	Прецедент UC-04: Оформлення платної підписки
Опис	Процес проведення фінансової транзакції, внаслідок якого користувач отримує преміальний статус (підписника) на обраному каналі.
Актори	Авторизований користувач (Глядач).
Передумови	Глядач авторизований у системі та знаходиться на сторінці цільового каналу.

Базовий потік подій	1. Глядач ініціює дію через кнопку оформлення підписки. 2. Система генерує інтерфейсне вікно з деталізацією тарифного плану та умов надання послуг. 3. Глядач підтверджує згоду на проведення фінансової операції. 4. Система виконує безпечне перенаправлення до зовнішнього платіжного шлюзу. 5. Після успішної авторизації платежу та отримання вебхука від емітента, система розширює права доступу користувача на відповідному каналі. 6. Інтерфейс клієнтського застосунку синхронізується з новим статусом (відображається активна підписка).
Альтернативні потоки	1. Крок 3: Глядач скасовує операцію в модальному вікні. Процес переривається без зміни стану системи. - Крок 1: Глядач вже володіє активною підпискою. Система модифікує інтерфейс, пропонуючи операції продовження терміну дії або дарування підписки іншому користувачу.
Виняткові ситуації	2. Відмова в проведенні транзакції на боці банку-емітента (недостатній баланс, блокування операції).
Постумови	Фінансова транзакція успішно зареєстрована, баланс платформи/автора трансляції оновлено, глядач отримав відповідні права доступу (розблокований чат,

	унікальні емодзі).

Назва прецеденту	Прецедент UC-05: Споживання медіаконтенту та інтерактивна взаємодія
Опис	Підключення клієнтського застосунку до активного відеопотоку та участь користувача в обміні текстовими повідомленнями.
Актори	Будь-який користувач (Гість або Авторизований).
Передумови	Цільовий канал перебуває у статусі «В ефірі» (Live).
Базовий потік подій	1. Користувач здійснює перехід на сторінку активної трансляції. 2. Клієнтський відеоплеєр завантажує HLS-маніфест та ініціює відтворення медіапотоку. 3. Браузер встановлює постійне WebSocket-з'єднання з сервером обміну повідомленнями. 4. Авторизований користувач формує текстове повідомлення та ініціює його відправку. 5. Сервер проводить санітизацію та валідацію повідомлення, після чого маршрутизує його всім активним клієнтам у поточній кімнаті. 6. Повідомлення синхронно відображається в інтерфейсі чату всіх підключених глядачів.
Альтернативні потоки	1. Крок 2: Політика безпеки браузера блокує автоматичне відтворення контенту зі звуком. Відтворення стартує у беззвучному режимі до ручного втручання користувача.
Виняткові ситуації	2. Автора трансляції раптово припиняє трансляцію

	(система переводить плеєр у стан очікування або відображає заглушку «Офлайн»). - Критичний розрив WebSocket-з'єднання (інтерфейс чату блокується для відправки повідомлень та ініціює механізм реконекту).
Постумови	Текстове повідомлення успішно зафіксовано в базі даних для подальшого VOD-відтворення, статистичні метрики ефіру (кількість глядачів) оновлено.

2.3 Проєктування архітектури застосунку

Під час проєктування архітектури було прийнято рішення відмовитися від класичного моноліту на користь розподіленої системи, де компоненти спілкуються між собою через черги повідомлень. Це дозволяє безболісно масштабувати проєкт і, що найголовніше, фізично ізолювати ресурсномісткі процеси обробки відео від звичайної бізнес-логіки та роботи з користувачами.

Усю систему можна розділити на сім логічних рівнів, зображено на рис. 2.1:

- Рівень клієнта (Client Layer). Реалізований як SPA (Single Page Application). Це єдина точка входу як для глядачів, так і для авторів трансляції. Фронтенд повністю бере на себе рендеринг інтерфейсу, відтворення потокового відео через HLS-плеєр та комунікацію з API.
- Рівень безпеки (Security Layer). Замість того, щоб самостійно реалізовувати зберігання паролів і наражатися на ризики витоку даних, автентифікацію делеговано зовнішньому провайдеру Auth0. Він працює за протоколами OAuth 2.0 / OIDC і повністю закриває питання безпечного входу та управління сесіями.
- Рівень монетизації (Monetization Layer). За процесинг банківських карток та регулярні підписки відповідає платіжний шлюз Stripe. Це знімає з нас необхідність проходити складну сертифікацію PCI DSS. Бекенд приймає від Stripe асинхронні вебхуки (Webhooks) і на їх основі оновлює фінансові статуси користувачів.

- Рівень проведення ефіру (Streaming Layer). Ядром відеотрансляцій виступає високопродуктивний медіасервер MediaMTX. Його головна задача — прийняти вхідний сигнал від автора (наприклад, по RTMP) і миттєво перепакувати (трансмуксувати) його у формат HLS для роздачі великій кількості глядачів.
- Рівень бізнес-логіки (Core Application Layer). Центральний Backend API обробляє всі основні запити: керує налаштуваннями профілів, чатом та метаданими ефірів. Щоб основний сервер не блокувався під час виконання тривалих задач, до системи додано брокер повідомлень RabbitMQ. Він забезпечує асинхронний зв'язок бекенду з іншими мікросервісами.
- Рівень фонової обробки (Processing Layer). Тут працює VOD Worker — ізольований фоновий сервіс. Він «лухає події з асинхроного брокера. Наприклад, коли трансляція завершується, воркер підхоплює запис, FFmpeg обробляє його для оптимізації, готує превью та готує файл до архівування.
- Рівень даних та зберігання (Data & Storage Layer). Інформаційні потоки чітко розділені. Всі структуровані дані (профілі, повідомлення, історія транзакцій) лежать у базі даних. А весь «важкий» неструктурований контент (відеоархіви, аватарки, обкладинки) відправляється у хмарне об'єктне сховище.

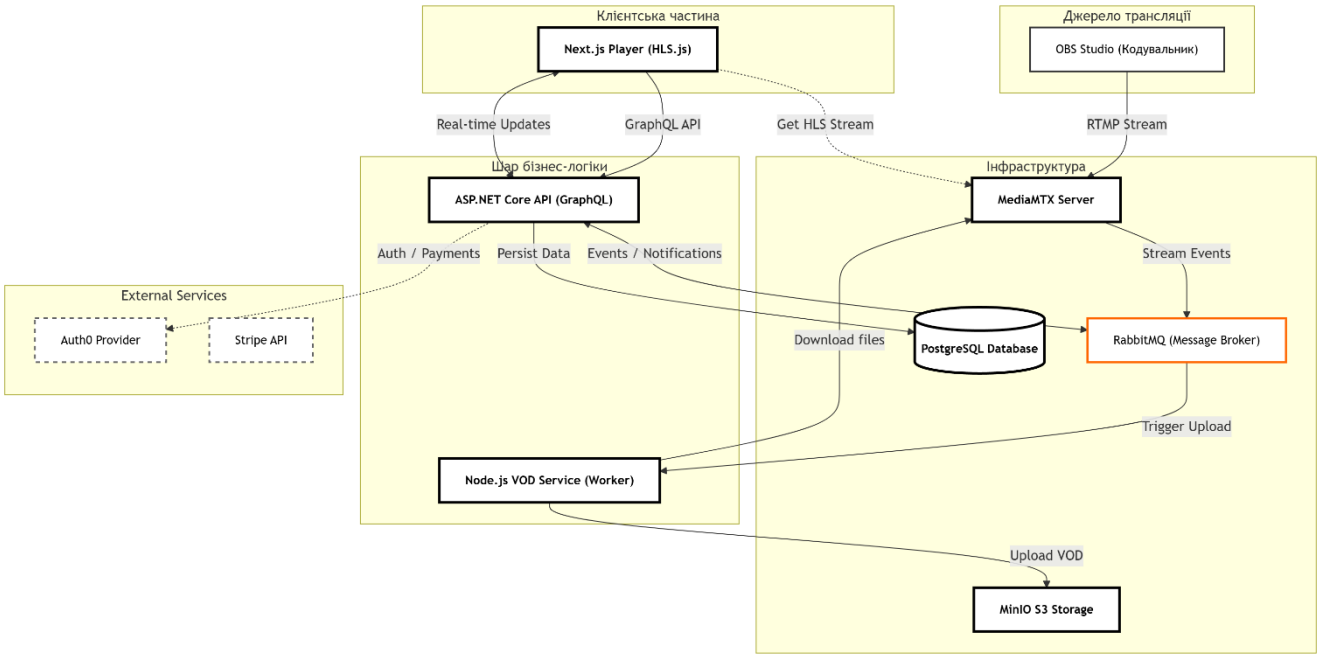

Рисунок 2.1 – Архітектура платформи

2.4. Проєктування моделі бази даних та структури зберігання контенту

Для роботи з даними у проєкті реалізована гібридна модель. Головна мета такого підходу — розподілити структуровані дані та медіаконтент по різних типах сховищ, щоб забезпечити швидку роботу інтерфейсу та стабільність при високих навантаженнях.

Оперативні дані (PostgreSQL). Реляційна база тримає все, що потребує суворих зв'язків та транзакційної цілісності. Це профілі користувачів, логіка підписок, метадані трансляцій та фінансова історія. Вибір PostgreSQL дозволяє будувати складні вибірки (наприклад, для глобального пошуку чи стрічки рекомендацій) і гарантувати, що дані про платежі або права доступу не розсинхронізуються[11].

Файлове сховище (AWS S3). Зберігати бінарні файли (відеозаписи, аватари, обкладинки). Зберігати це в базі даних — це архітектурна помилка, яка призвела б до розростання БД і падіння швидкодії. Тому весь медіаконтент винесено в об'єктне сховище S3. Бекенд записує в базу лише посилання на файл, а сама хмара бере на себе питання масштабування дискового простору та швидкої роздачі контенту користувачам[12].

Така структура дозволяє системі залишатися гнучкою. Це дозволяє

нарощувати потужність бази даних і окремо розширювати об'єм сховища для відеоархівів, не перевантажуючи при цьому головний сервер.

2.4.1 Логічна та фізична модель реляційної бази даних

Логічна та фізична моделі даних розроблені з фокусом на відмовостійкість та швидку роботу під навантаженням. Схема відповідає третій нормальній формі (3NF), що мінімізує дублювання та аномалії. Для спрощення масштабування та міграції даних у розподіленому середовищі замість звичайних інкрементних ID використано UUID як первинні ключі для всіх таблиць[11].

Структура бази даних розділена на кілька ключових блоків:

1. Управління користувачами та профілями

Центральною сутністю є таблиця користувачів (Users), де зберігаються облікові дані та налаштування каналу.

- Інтеграція зі Stripe: Для зв'язку з фінансовою частиною в таблиці зберігається Stripe_Customer_ID. Це дозволяє асоціювати локального юзера з його платіжним профілем без дублювання банківських даних.
- Оптимізація: Щоб інтерфейс працював швидше, застосовано часткову денормалізацію. Статус ефіру та лічильник фоловерів зберігаються прямо в основній таблиці — це позбавляє систему необхідності робити складні агрегаційні запити (COUNT) при кожному відкритті сторінки. При цьому важкі дані (біографія, соціальні лінки) винесені в окрему таблицю профілів, щоб не роздувати основну базу.

2. Контент та медіаархів

Система чітко розділяє живі ефіри та записи:

- Streams: Таблиця для активних сесій, яка містить актуальні метрики трансляції.
- Vods: Після завершення ефіру дані переходять у таблицю відеоархіву. Тут реалізовано гібридний підхід: у базі зберігаються лише метадані та посилання на маніфести HLS, а самі важкі відеофайли фізично лежать у

хмарі AWS S3.

- Класифікація: Для пошуку та фільтрації впроваджено систему тегів через зв'язок «багато-до-багатьох» (Many-to-Many).

3. Монетизація та права доступу

Підсистема платних підписок базується на таблиці Subscriptions. Вона виконує роль локального дзеркала (кешу) даних із Stripe.

- У базі фіксуються ID підписки, тарифний план та дата закінчення терміну дії.
- Це дозволяє миттєво перевіряти права доступу глядача до чату або бонусів без звернення до API платіжної системи при кожному кліку. Актуальність цього кешу підтримується через асинхронні вебхуки.

4. Соціальна взаємодія та безпека

Для чату та системи фоловерів створено таблиці, оптимізовані під інтенсивний запис (Write-Heavy).

- Чат: Індекси налаштовані так, щоб максимально швидко діставати історію повідомлень для конкретного каналу.
- Безпека: Система модерації (бани) винесена в окрему сутність, де фіксується причина обмеження та термін його дії.

Цілісність усієї структури підтримується на рівні бази через зовнішні ключі (Foreign Keys) та правила каскадного оновлення, що виключає появу «битих» посилань між користувачами, ефірами та платежами, зображено на рис 2.2.

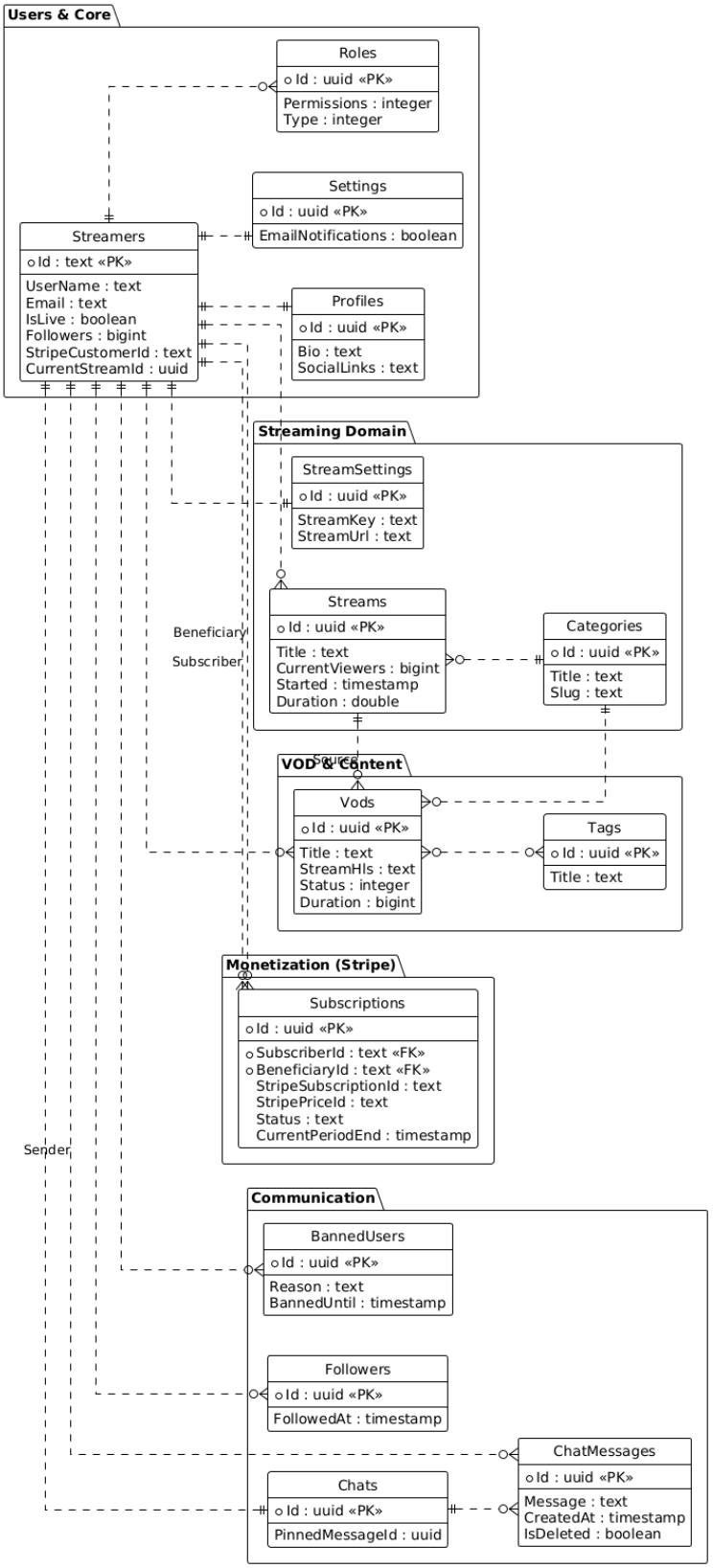

Рисунок 2.2 – Схема бази даних

2.4.2 Структура зберігання контенту (AWS S3)

Для зберігання медіафайлів у проєкті використано об’єктне сховище, що працює за стандартом Amazon S3. Головна перевага такого вибору — можливість легко підключити CDN (Content Delivery Network) для швидкої роздачі відео глядачам по всьому світу.

Щоб дані не змішувалися, а доступ до них був чітко розмежований, було розроблено ієрархічну структуру папок (префіксів).

Стратегія іменування файлів (Key Naming Strategy)

Відеоархіви (HLS):

- videos/{streamer_id}/{vod_id}/index.m3u8 — основний файл (маніфест), який плеєр зчитує першим.
- videos/{streamer_id}/{vod_id}/segment_001.ts — окремі короткі фрагменти відео, на які нарізається ефір.

Графічний контент:

- images/avatars/{streamer_id}_{timestamp}.jpg — фото профілів авторів трансляції (мітка часу в назві потрібна, щоб браузери не кешували старі фото після оновлення).
- images/previews/{vod_id}.jpg — автоматично згенеровані обкладинки для минулих трансляцій.

Така організація дозволяє гнучко налаштувати права доступу (Bucket Policies). Наприклад, це дозволяє конкретному користувачу завантажувати файли лише у його власну папку, тоді як CDN матиме доступ до читання всього контенту для трансляції глядачам. Це виключає ситуацію, коли один користувач може випадково або навмисно видалити чи змінити чужі записи.

2.5 Висновки до другого розділу

Другий розділ повністю присвячено технічному проєктуванню веб застосунка. Робота почалася з фіксації жорстких меж: система не просто має транслювати відео чи підтримувати чат, а робити це з конкретними таймінгами. Наприклад, затримка самого потоку не може перевищувати 20 секунд, а на

доставку повідомлення в чаті виділяється менше ніж півсекунди. Для кращого розуміння логіки роботи ці вимоги було перетворено на конкретні сценарії: від простої реєстрації глядача до налаштування ключів трансляції автором в умовному OBS.

На базі цих сценаріїв спроєктовано багаторівневу архітектуру застосунку. Від ідеї писати все в одному великому моноліті довелося відмовитися, бо така структура швидко "впала" б під навантаженням від трансляцій. Натомість систему розділено на незалежні блоки. Головний сервер займається виключно бізнес-логікою та клієнтськими запитами. Усі ресурсномісткі задачі, на зразок оптимізації відеозаписів чи створення обкладинок, винесені в ізольовані фонові сервіси. Вони отримують команди асинхронно через брокер повідомлень RabbitMQ. А безпосереднім прийомом та трансляцією відео займається окремий медіасервер MediaMTX. Завдяки цьому обробка гігабайтів відео ніяк не гальмує роботу бази даних чи клієнтського інтерфейсу.

Схожий принцип розділення застосовано і до системи зберігання інформації. Гібридна модель розводить структуровані дані та важкі медіафайли по різних сховищах. Текстова інформація — профілі, історія оплат, налаштування доступу — лежить у реляційній базі PostgreSQL. Для пришвидшення роботи інтерфейсу тут застосовано часткову денормалізацію. Наприклад, статус ефіру оновлюється прямо в основній таблиці профілю, що позбавляє сервер необхідності робити складні обчислення при кожному завантаженні сторінки.

Зберігати відеоархіви безпосередньо в базі даних технічно неправильно, тому весь медіаконтент автоматично відправляється в хмарне об'єктне сховище AWS S3. Розроблена ієрархія папок не лише сортує файли за ідентифікаторами авторів, а й дозволяє легко налаштувати правила безпеки та підключити CDN для швидкої роздачі відео по всьому світу.

Також архітектура знімає з проєкту зайві ризики щодо збереження чутливої інформації. Платформа взагалі не тримає в базі паролів чи номерів банківських карток. Авторизацію повністю делеговано сервісу Auth0, а

процесинг фінансів — платіжному шлюзу Stripe.

Створена архітектурна модель та структура бази даних ізолюють бізнес-логіку від обробки важкого медіаконтенту.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОНКУ

Для побудови платформи було обрано технологічний стек, який дозволяє максимально відокремити бізнес-логіку та ресурсномістку обробку відео.

Серверна частина базується на .NET 10 (C# 14). Це дало змогу використати Native AOT для швидкого запуску сервера та оптимізовану роботу з пам'яттю, що є критичним при великій кількості активних з'єднань. Замість звичного REST API взаємодія з клієнтом побудована на GraphQL (HotChocolate). Це розв’язало проблему зайвого трафіку, фронтенд запитує лише ті дані, які потрібні для рендерингу конкретного екрана. Також, GraphQL-підписки через WebSockets забезпечили оновлення чату без зайвих запитів до сервера.

Клієнтська сторона реалізована на фреймворку Next.js із використанням TypeScript. Такий підхід дозволив поєднати швидке завантаження сторінок (через Server-Side Rendering) та високу інтерактивність кабінету автора трансляції.

Використання TypeScript на обох кінцях системи гарантує, що структура даних, яку віддає бекенд, завжди відповідає тому, що очікує інтерфейс.

Медійна інфраструктура забезпечується на сервері MediaMTX. Він працює як вхідна точка для RTMP-потоків від авторів і перепаковує їх у формат HLS для глядачів. Усі фонові задачі — транскодування відео після ефіру, підготовка скріншотів та архіву (VOD) — виконуються через FFmpeg. Щоб ці ресурсомісткі процеси не гальмували основний API, було впроваджено брокер повідомлень RabbitMQ. Він працює як черга: коли трансляції завершується, бекенд надсилає подію в чергу, а окремий воркер підхоплює її та починає обробку відео в асинхронному режимі.

Зберігання даних було розділенл за гібридним принципом. Реляційна база PostgreSQL відповідає за профілі, права доступу та логіку підписок. Для роботи

з нею використано Entity Framework Core. Відеофайли та зображення — відвантажується у хмарне сховище AWS S3. Це дозволяє системі не залежати від дискового простору сервера та використовувати CDN для швидкої доставки відео контенту.

Питання безпеки та монетизації вирішені через інтеграцію із профільними сервісами. Автентифікація реалізована через Auth0 (OAuth 2.0 / OpenID Connect), що знімає з нас відповідальність за безпечне зберігання паролів. Фінансова частина, включаючи регулярні списання за підписки, працює через Stripe. Взаємодія з ним налаштована через систему вебхуків: як тільки проходить оплата, Stripe надсилає сигнал на сервер, і сервер автоматично оновлює статус користувача в базі даних.

3.1. Мова програмування та платформа розробки C# / .NET.

Серверну частину проєкту реалізовано мовою C# з використанням платформи .NET. Для платформи проведення потокового мовлення сервісу такий вибір має кілька практичних переваг. Найперше — це сувора типізація. У системі багато різних даних: відеопотоки, профілі, фінанси, ролі користувачів. Якщо розробляти серверну частину динамічними мовами, виникає можливість помилок типізації і програма буде некоректно працювати. C# перевіряє код на етапі компіляції та одразу виявляє такі помилки. Це зменшує час розробки. Інший важливий момент — це продуктивність. Робота з відео сильно навантажує сервер. У .NET добре налаштована асинхронність. Коли програма звертається до бази даних або зберігає файл, робочий потік не чекає на відповідь. Він іде обробляти запити інших глядачів. Тому платформа може тримати багато підключень одночасно і не гальмувати. Розробка велася в операційній системі Linux за допомогою середовища Rider. Сучасний .NET повністю кросплатформенний, тому готовий застосунок без проблем збирається в Docker-контейнери. Також платформа має багато готових інструментів. Наприклад, для бази даних використовується Entity Framework Core. Завдяки цьому не потрібно шукати сторонні бібліотеки, є можливість використовувати

стандартні рішення і реалізовувати логіку проєкту[24].

3.2 Аналіз підходів до проєктування програмних інтерфейсів (REST vs GraphQL)

Після вибору мови програмування потрібно було визначитися з тим, як фронтенд буде спілкуватися з бекендом. Зазвичай для цього використовують REST, але було обрано GraphQl

Головна причина — це гнучкість запитів. У системі на різних сторінках потрібні різні набори даних. Наприклад, в одному місці потрібно вивести повну інформацію про автора трансляції, а в іншому — тільки його картинку. Якщо використовувати REST, сервер завжди присилає фіксований набір даних, навіть якщо половина з них зараз не потрібна. Це неефективне витрачення трафіку.

GraphQL дозволяє клієнту самому вказати, які саме поля він хоче отримати.

Також GraphQL вирішує проблему зайвих запитів. Щоб зібрати інформацію для сторінки трансляції (дані про відео, автора, список коментарів), у REST часто доводиться робити три окремі запити на сервер. У GraphQL все це можна отримати за один раз. Це значно прискорює роботу інтерфейсу, особливо на мобільних пристроях або при слабкому інтернеті[10].

Завдяки чіткій схемі даних, розробник фронтенду завжди точно знає, які запити можна робити і які типи даних прийдуть у відповідь. Це спрощує розробку та зменшує кількість помилок при стикуванні двох частин застосунку.

3.3 Платформа для вебінтерфейсу (Next.js / React)

Для клієнтської частини застосунку обрано React та Next.js. React добре підходить для побудови інтерфейсу з ізольованих компонентів. Зручно один раз реалізувати елемент відеоплеєра чи блок чату і потім використати в інших місцях.

Проте, для повноцінної платформи цих можливостей виявилося недостатньо, тому було обрано фреймворк Next.js.

Основна причина такого рішення — серверний рендеринг (SSR). Якщо використовувати React, браузер спочатку вантажить порожній HTML-документ, а вже потім виконує скрипти і рендерить інтерфейс. Користувач у цей час дивиться на білий екран. Next.js генерує сторінки одразу на сервері. Клієнт отримує вже готовий код, тому візуально все завантажується набагато швидше. Для глядачів трансляцій будь-які затримки на старті дуже критичні. До того ж такий підхід повністю вирішує проблему з індексацією сторінок пошуковими системами[25].

Інший важливий момент стосується маршрутизації. Next.js має власний роутер, який працює на основі файлової системи. Через це відпадає необхідність підключати і конфігурувати сторонні пакети на зразок react-router-dom.

Окрему увагу необхідно приділити медіафайлам. Сервіс постійно підвантажує велику кількість картинок, таких як прев'ю трансляцій або аватари користувачів. Вбудовані інструменти Next.js автоматично стискають зображення і форматують їх під розмір екрана конкретного пристрою. Це ефективно розвантажує мережу і економить трафік.

3.4 Бази даних (PostgreSQL vs NoSQL)

Для збереження даних платформи було обрано PostgreSQL. На етапі проєктування також розглядалися NoSQL рішення (наприклад, MongoDB), але специфіка веб застосунка вимагає саме реляційного підходу.

Головна причина — це структура інформації. Платформа працює з чітко пов'язаними сутностями: користувачі, канали, підписки, ролі доступу та фінансові транзакції. Використання реляційной бази даних, зокрема PostgreSQL, є доцільним для таких задач, бо гарантує цілісність даних. Наприклад, база даних на рівні власної архітектури не дозволить створити запис про оплату, якщо такого користувача не існує в системі[11].

NoSQL також мають свої переваги. Вони швидкі і гнучкі. Їх було б зручно використовувати для неструктурованих даниї, наприклад, для історії

повідомлень у чаті трансляції. Але для фінансів, статусів підписок та прав доступу, потрібна максимальна надійність і строгі зв'язки. PostgreSQL забезпечує це завдяки чіткій роботі з транзакціями[11].

Також вагому роль відіграла інтеграція з іншими технологіями. PostgreSQL ефективно взаємодіє з обраною серверною частиною на .NET та інструментом Entity Framework Core. Налаштування бази, створення таблиць та оновлення їхньої структури здійснюється швидко і без потреби ручного написання складних SQL-запитів. Це дозволило зосередитись саме на логіці потокового мовлення, а не на адмініструванні бази даних.

3.5 Організація фонових задач (RabbitMQ)

Веб застосунок має багато ресурсномістких процесів. Функціонування платформи передбачає виконання низки фонових завдань: обробку відеопотоку, відправку сповіщень та верифікацію платежів. Концентрація всіх цих процесів на єдиному центральному сервері швидко призведе до його перевантаження та критичного збою системи.

Було обрано інструмент RabbitMQ. Він працює як черга повідомлень. Сервер не витрачає ресурси на складні задачі. Він створює запис про нову операцію і надсилає його в RabbitMQ. Після цього сервер одразу повертається до користувачів[26].

Задачі з черги забирає окремий фоновий сервіс і поступово виконує. Це ефективно розвантажує ресурси.

Додатковою вагомою перевагою застосування брокера повідомлень є підвищення загальної відмовостійкості системи. У випадку аварійного завершення роботи фонового сервісу під час виконання операції (наприклад, збереження файлу), завдання не втрачається, а надійно зберігається в черзі. Після відновлення працездатності програми процес обробки перерваного завдання автоматично поновлюється. Цей архітектурний підхід гарантує захист від втрати інформації у разі виникнення непередбачуваних апаратних чи програмних збоїв.

3.6 Організація середовища розробки (.NET Aspire)

Організація локального середовища розробки реалізована за допомогою інструмента .NET Aspire, який виконує функції локального оркестратора. Це дає змогу ініціалізувати залежні компоненти системи безпосередньо з робочого середовища.

Використання цього рішення знімає необхідність ручного управління інфраструктурою. Docker-контейнери бази даних та брокера повідомлень запускаються автоматично, після чого встановлюються мережеві зв'язки між сервісами. Повністю зникає потреба фіксувати порти чи облікові дані у конфігураційних файлах. Серверна частина отримує актуальні параметри підключення динамічно під час старту.

Сфера застосування даного оркестратора суворо обмежена етапом написання коду та локального налагодження. Розгортання платформи у промисловому середовищі базується на використанні конфігурацій Docker Compose. Відповідно, впровадження .NET Aspire вирішує виключно проблему економії часу на регулярний перезапуск мікросервісів у процесі розробки.

3.8 Розгортання системи (Docker та Docker Compose)

Платформа будується як сукупність ізольованих вузлів: API-сервера, СУБД PostgreSQL, брокера RabbitMQ та воркерів для фонової обробки. Традиційний метод розгортання через ручне встановлення компонентів тут недоцільний. Це зумовлено високим ризиком несумісності системних бібліотек та складністю синхронізації версій середовища.

Проблема вирішується впровадженням Docker. Технологія дозволяє інкапсулювати кожен сервіс разом із його специфічним оточенням у незалежний контейнер [27]. Це дає головну перевагу — гарантовану ідентичність роботи застосунку як на етапі розробки, так і у продуктивному середовищі.

Для координації роботи всієї групи контейнерів використано Docker Compose. Весь опис інфраструктури зосереджено в одному конфігураційному

файлі, де визначено ліміти ресурсів та правила мережевого доступу. Зокрема, Docker Compose автоматично створює ізольовану віртуальну мережу, що дозволяє сервісам звертатися один до одного за внутрішніми іменами хостів замість динамічних IP-адрес.

Особлива увага приділена збереженню даних. Оскільки внутрішній простір контейнера є ефемерним, для забезпечення персистентності бази даних задіяно механізм томів (volumes). Це дозволяє фізично винести файли бази на дисковий накопичувач сервера, убезпечивши інформацію від втрати при перезавантаженні сервісів.

У підсумку, процедура розгортання зводиться до перенесення одного файлу та виконання єдиної команди. Платформа самостійно підтягує образи та ініціалізує зв'язки, що нівелює людський фактор і значно прискорює реліз системи.

3.9 Реалізація серверної частини та API на платформі .NET з використанням GraphQL

Бекенд-частина системи реалізована на принципах **Vertical Slice Architecture**. На відміну від класичного горизонтального розшарування, де код розділений за технічними обов'язками, у цьому проєкті систему декомпоновано на автономні зрізи. Кожен такий зріз відповідає за конкретну бізнес-функцію (фічу) і містить усе необхідне: від вхідної точки GraphQL до логіки роботи з базою даних. Це забезпечує високу зв'язність коду (High Cohesion), оскільки логіка кожної окремої операції зосереджена в одному просторі імен.

Для побудови API застосовано підхід **Code-First**. Схема GraphQL автоматично формується на основі C#-класів, що гарантує сувору типізацію та повну синхронізацію між кодом і документацією. Центральною точкою налаштування API є клас Extensions.cs, повний лістинг якого наведено у додатку А (лістинг А.1) . У ньому реалізовано реєстрацію сервісів у DI-контейнері та побудовано конвеєр обробки запитів за допомогою бібліотеки

HotChocolate, що включає механізми авторизації та оптимізації SQL-запитів.

Обробка запитів базується на патерні CQRS (Command Query Responsibility Segregation). GraphQL-резолвери в системі не містять бізнес-логіки, а працюють як фасади, що делегують виконання запитів через медіатор (MediatR) до відповідних обробників. Приклад реалізації резолвера StreamQuery представлено у додатку А (лістинг А.2). Використання атрибутів [UsePaging], [UseFiltering] та [UseSorting] дозволяє клієнту гнучко формувати вибірку даних, які сервер автоматично конвертує в оптимізовані запити до PostgreSQL.

Для забезпечення інтерактивності в реальному часі використано механізм GraphQL Subscriptions поверх протоколу WebSockets. Система працює на базі динамічних топіків для маршрутизації повідомлень. Як показано у лістингу коду в додатку А(лістинг А.3), підписка на події чату параметризована ідентифікатором chatId. Це гарантує, що клієнт отримує дані лише з того каналу, який він переглядає в даний момент.

Бізнес-логіка інкапсульована в доменних сутностях згідно з принципами DDD (Domain-Driven Design). Клас Stream, код якого наведено у додатку А(лістинг А.4), реалізований як «багата модель» (Rich Domain Model), що самостійно контролює свій стан і валідацію. Важливою частиною архітектури є використання доменних подій (IDomainEvent). Наприклад, при зміні статусу трансляції сутність генерує подію StreamUpdated, на яку асинхронно реагують інші компоненти системи, не порушуючи принципу слабкої зв'язності коду.

3.10 Розробка клієнтського інтерфейсу на базі Next.js та інтеграція з Auth0

Фронтенд-застосунок побудований як SPA (Single Page Application) на базі фреймворку Next.js. Вибір стеку React + TypeScript дозволив створити типізовану архітектуру, де структура компонентів чітко відповідає моделям даних бекенду.

Для стилізації використано Tailwind CSS, що спростило розробку адаптивного дизайну: інтерфейс автоматично підлаштовується під роздільну

здатність екрана, від мобільних телефонів до широкоформатних моніторів.

Взаємодія з GraphQL-сервером реалізована через Apollo Client. Головна складність конфігурації полягала в необхідності працювати одночасно з двома протоколами: HTTP (для звичайних запитів) та WebSockets (для чату та живих оновлень). Для цього було застосовано механізм розділення посилань (Split Link). У додатку А(лістинг А.5) наведено код провайдера MyApolloProvider, де функція split автоматично маршрутизує трафік залежно від типу операції. Також у конфігурацію інтегровано authLink — цей механізм автоматично додає актуальний JWT-токен від Auth0 до кожного заголовка запиту, забезпечуючи безпеку без ручного прокидання токенів у кожному компоненті.

Щоб уникнути помилок при написанні запитів і вручну не описувати інтерфейси, у проєкті використано інструмент GraphQL Code Generator. Процес розробки базується на описі структури даних у .graphql-файлах приклад такої структури для чату наведено у Додатку А(лістинг А.6)), а утиліта автоматично генерує готові React-хуки, такі як useCreateMessageMutation. Це дає 100% відповідність типів: якщо на бекенді зміниться назва поля, клієнтський код не скомпілюється, що дозволяє виявляти помилки ще до запуску застосунку.

Ключовим елементом інтерфейсу є StreamPlayer — компонент для відтворення трансляцій. Він працює з бібліотекою react-hls-player, яка відповідає за обробку HLS-потоків. Як видно з лістингу в Додатку А(лістинг А.7), компонент не лише відтворює відео, а й підтримує адаптивний бітрейт, автоматично підлаштовуючи якість під швидкість інтернету користувача. Для зручності було додано шар оверлеїв, які з’являються лише під час активності користувача, щоб не перекривати контент під час перегляду.

Загальна структура сторінок тримається на компоненті MainLayout додаток А(лістинг А.8). Він відповідає за глобальну навігацію та динамічне керування бічною панеллю (Sidebar). Використання медіа-запитів дозволило реалізувати логіку, при якій на смартфонах панель навігації приховується в компактне меню, звільняючи місце для відеоплеєра та чату.

3.11 Налаштування медіа-сервера MediaMTX

Ядром підсистеми виступає сервер MediaMTX, який працює як Ingest-вузол: він приймає вхідний RTMP-потік від автора та конвертує його для роздачі глядачам. Основний акцент у конфігурації зроблено на мінімізацію затримки (Low Latency), що критично для живого спілкування в чаті. Для цього параметр hlsPartDuration встановлено на позначці 500 мс. Таке налаштування змушує сервер генерувати мікросегменти відео, завдяки чому плеєр глядача починає відтворення майже миттєво, не чекаючи накопичення повного плейлиста.

Безпека вхідного потоку реалізована через механізм authHTTPAddress. Перед тим як прийняти сигнал, MediaMTX звертається до API для верифікації токена автора, що виключає можливість несанкціонованих трансляцій. Взаємодія медіасервера з іншими частинами платформи побудована на системі хуків runOnReady та runOnNotReady. Замість того, щоб навантажувати сервер прямими запитами до бази даних, було використана проміжний Python-скрипт. Він відловлює моменти старту та зупинки ефіру і публікує відповідні події (liveStreamStarted / liveStreamEnded) у шину RabbitMQ. Це робить систему відмовостійкою: навіть якщо основний бекенд тимчасово недоступний, подія про ефір не зникне, а залишиться в черзі. Детальні налаштування сервера описані у конфігураційному файлі mediamtx.yml, який наведено у додатку А(Лістинг А.9).

3.12 Реалізація сервісу обробки відео (VOD Processor) та інтеграція з AWS S3

Для автоматичного створення архіву завершених трансляцій було розроблено окремий мікросервіс на Node.js — VOD Processor. Архітектурно це ізольований фоновий воркер, який активується лише після отримання повідомлення з RabbitMQ про завершення ефіру. Такий підхід дозволив винести всі ресурсномісткі операції з відео за межі основного сервера додатків.

Робочий цикл воркера починається із захоплення потоку через FFmpeg. Програма зчитує HLS-плейлист і склеює відеосегменти в один файл без

перекодування, щоб зберегти оригінальну якість і не витрачати зайвий час на рендеринг. Одночасно з цим FFmpeg генерує статичне зображення (thumbnail), яке стає обкладинкою для відео в каталозі.

Для передачі готових медіафайлів у хмарне сховище AWS S3 замість стандартних бібліотек SDK було інтегровано утиліту Rclone. Вона значно ефективніше працює з великими файлами, підтримує багатопотоковість і вміє автоматично дозавантажувати дані у разі розриву з'єднання. Після того як файли опиняються у хмарі, воркер відправляє в систему подію VodReady, сигналізуючи бекенду про доступність запису, і видаляє тимчасові дані з локального диска.

Повний код цього сервісу з описом логіки обробки представлено у додатку А(лістинг А.10). Така схема гарантує, що ресурсоємна обробка відео ніяк не впливає на швидкість роботи сайту чи стабільність трансляцій.

3.13 Висновки до третього розділу

Третій розділ описує процес перетворення спроєктованої архітектури на реальний робочий код. Головна ідея реалізації полягала в тому, щоб важкі процеси обробки відео фізично не перетиналися з основною бізнес-логікою і не гальмували сайт.

Основний бекенд написаний мовою C# на базі .NET 10. Замість стандартного розшарування коду було використано Vertical Slice Architecture.

Кожна функція платформи лежить у своїй окремій ізольованій папці. Це робить проєкт чистим і дозволяє легко додавати нові фічі. Для зв'язку бекенду з клієнтом замість REST API обрано GraphQL. Фронтенд не робить запити на сервера з купою непотрібної інформації, а чітко замовляє лише ті дані, які треба показати на екрані прямо зараз. Чат працює через WebSockets: повідомлення надходять миттєво без постійного оновлення сторінки.

Інтерфейс користувача зібраний за допомогою Next.js. Він формує сторінки ще на сервері, тому все працює ефективно. Щоб бекенд і фронтенд працювали разом, було підключено автоматичну генерацію коду. Утиліта

створює типи даних на основі запитів. Якщо на сервері щось зміниться, клієнтський код видасть помилку компіляції.

Центральним завданням при розробці платформи стала організація обробки відеопотоків. Сигнал від транслятора приймає медіасервер MediaMTX, який ділить потік на короткі HLS-сегменти. Це дозволяє скоротити затримку відтворення у клієнтському плеєрі до 500 мс. Після закінчення ефіру запускається процес архівування. Щоб не перевантажувати основний вузол обчисленнями, конвертацію відео реалізовано асинхронно через RabbitMQ. Бекенд лише публікує повідомлення про завершення ефіру в чергу, а фоновий сервіс забирає цю задачу, виконує транскодування через FFmpeg та готує фінальні файли з обкладинками.

Зберігання даних реалізовано за роздільним принципом. PostgreSQL використовується для роботи зі структурованою інформацією: профілями, транзакціями та метаданими. Водночас готові відеоархіви вивантажуються до хмарного сховища AWS S3, що знімає обмеження щодо об’єму жорстких дисків локального сервера.

Для автентифікації та фінансових операцій інтегровано сервіси Auth0 та Stripe. Платформа взаємодіє з ними через вебхуки, отримуючи лише результати дій. Такий підхід гарантує безпеку, оскільки конфіденційні паролі або реквізити карток не потрапляють до внутрішньої бази даних проєкту.

РОЗДІЛ 4. ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКУ

У цьому розділі наведено результати перевірки готового веб-застосунку. Головна мета етапу — виявити критичні помилки та переконатися, що платформа готова до запуску.

Процес тестування поділявся на кілька рівнів. Спочатку базова бізнес-логіка перевірялася ізольовано через юніт-тести. Після цього було додано інтеграційні тести для перевірки роботи з базою даних. Вони підтвердили, що Entity Framework Core формує правильні запити, а структура даних не ламається під час активного оновлення ефірів.

Комплексне тестування системи проводилося безпосередньо в браузері. Оцінювалася швидкість GraphQL-запитів та стабільність віддачі відео через HLS. Окрему увагу приділено затримці трансляції (latency): порівняння картинки в OBS та плеєрі показало мінімальне відставання. Це підтверджує правильне налаштування медіасервера MediaMTX. Також протестовано безпеку платформи. Будь-які спроби звернутися до закритого API без токена Auth0 миттєво відхилялися.

4.1 Юніт-тестування

Будь-яка зміна в проєкті може випадково зламати існуючу логіку. Щоб цього уникнути, базові алгоритми покриваються юніт-тестами. Їхня головна фішка в тому, що код перевіряється повністю автономно. Під час таких тестів програмі не потрібне підключення до бази даних чи інтернету. Перевіряється суто робота самого механізму.

Як приклад, у додатку А (лістинг А.11) зображено тестування сервісу SlugGenerator. Він виконує дуже специфічну задачу: бере довільний текст і генерує з нього чисте посилання для адресного рядка. Без цього інструменту неможлива нормальна пошукова оптимізація сайту.

Тести написані за допомогою фреймворку xUnit. Було спеціально вирішено відмовитися від написання десятка однакових методів під кожну дрібну перевірку. Натомість взято атрибут [Theory], передано йому набір вхідних даних через [InlineData] і отримано один гнучкий метод. Метод по черзі обробляє через себе різні життєві сценарії. Перевіряється: як сервіс очищає текст від спецсимволів, обрізає пробіли та як конвертує специфічні літери (наприклад, перетворює іспанську ñ на звичайну n).

Замість класичних методів порівняння застосовуються вирази формату result.Should().Be(expected), які сприймаються як логічні стверджувальні речення.

Це суттєво спрощує аналіз коду іншими учасниками команди розробки та полегшує розуміння цілей конкретного тесту [23]. Такий підхід мінімізує поріг

входження при ознайомленні з тестовим покриттям і підвищує загальну підтримуваність програмного забезпечення.

4.2 Інтеграційне тестування взаємодії з базою даних

Інтеграційні тести перевіряють роботу системи в комплексі. На цьому етапі вже підключається база даних. Це потрібно для того, щоб переконатися: програма не просто виконує логіку в пам'яті, а й правильно зберігає та зчитує інформацію через Entity Framework Core.

У додатку А (лістинг А.12) показано тестування функції створення повідомлень у чаті. Перед кожним запуском автоматично розгортається повністю чисте тестове середовище. Спочатку перевіряється успішний сценарій: створюється віртуальний користувач, проходить авторизація і відправляється текст. Після цього робиться прямий запит до бази. Це підтверджує, що запис дійсно з'явився у таблиці та має правильні зв'язки з чатом.

Окремо перевіряються всі можливі помилки. Написано тести для випадків, коли не знайдено ініціатора дії, вказано неправильний ідентифікатор чату або відсутнє повідомлення, на яке дається відповідь. Також тестуються ліміти: якщо текст довший за 250 символів, система має одразу повернути помилку валідації. Для зручного опису всіх цих умов використано бібліотеку FluentAssertions, яка робить код перевірок коротшим та більш читабельним[24].

4.3 Верифікація нефункціональних вимог та системний аналіз

Окрім перевірки логіки, тестувалася загальна швидкість і безпека платформи. Спочатку вимірювався час відповіді бекенду. Моніторинг показав, що в середньому запити до сервера йдуть близько 20 мілісекунд (мс). Це хороший показник, який дозволяє інтерфейсу реагувати на дії користувача без помітних затримок.

graphql	200	fetch	createHttpLink.js:146	229 B	9 ms
graphql	200	fetch	graphql.ts:6299	597 B	9 ms
graphql	200	fetch	graphql.ts:7523	2.7 kB	11 ms
graphql	200	fetch	graphql.ts:7616	530 B	12 ms
graphql	200	fetch	createHttpLink.js:146	408 B	15 ms
graphql	200	fetch	createHttpLink.js:146	758 B	13 ms
graphql	200	fetch	graphql.ts:4612	682 B	9 ms

Рисунок 4.**1** — Час обробки клієнтських запитів сервером

Далі перевірялася система відеотрансляцій. Головний параметр тут — затримка ефіру. Під час проведення ефіру через протокол HLS відставання відео в плеєрі від джерела (програми OBS) склало 15 секунд. Для технології HLS це стандартна і нормальна поведінка, оскільки протокол буферизує відео сегментами, щоб уникнути зависань у глядачів.

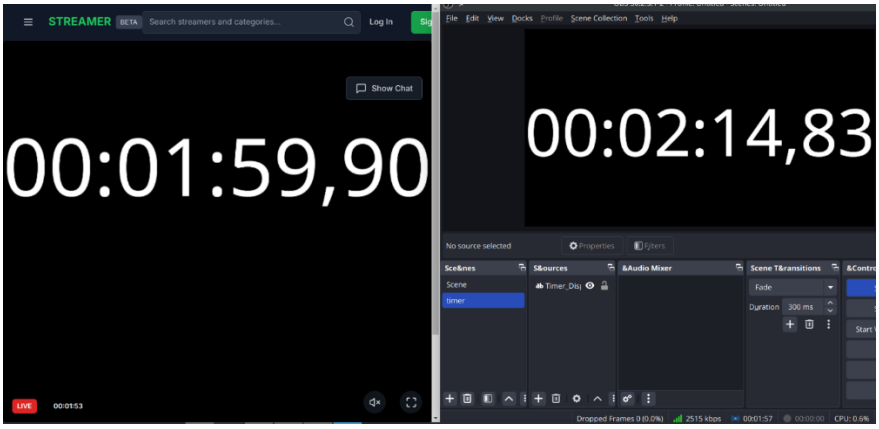

Рисунок 4.2 — Фіксація затримки відеотрансляції (HLS)

Наприкінці тестувався захист даних. Щоб переконатися в надійності системи авторизації, було зімітовано звернення до захищених ресурсів GraphQL взагалі без токена доступу. Сервер коректно заблокував цю дію і повернув статус помилки AUTH_NOT_AUTHENTICATED.

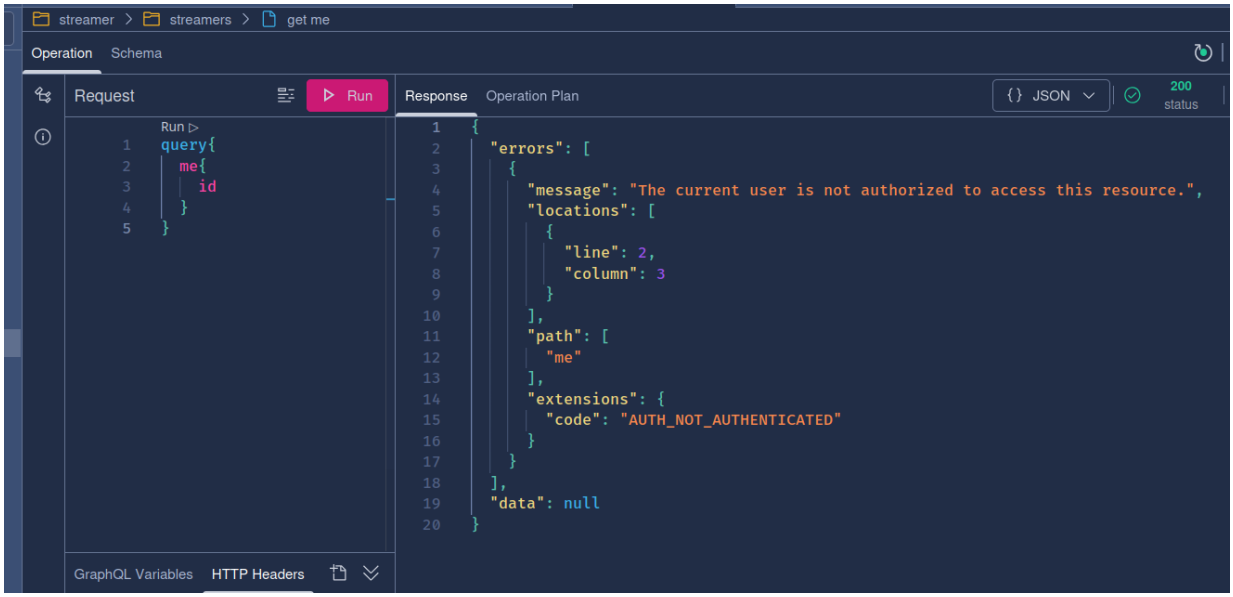

Рисунок 4.3 — Помилка 401 при неавторизованому GraphQL-запиті

4.4 Висновки до четвертого розділу

У четвертому розділі описано процес перевірки готової платформи. Головним завданням було знайти можливі баги та переконатися, що система витримає реальні умови експлуатації і готова до розгортання.

Перевірка йшла від простого до складного. Спочатку базові алгоритми покрили ізольованими юніт-тестами за допомогою xUnit та FluentAssertions. Це дозволило швидко прогнати різні сценарії використання і перевірити логіку без підключення до реальної бази. Після цього додалися інтеграційні тести. Вони підтвердили, що бекенд правильно спілкується з PostgreSQL: нові повідомлення в чат зберігаються без помилок, а всі обмеження, наприклад, максимальна довжина тексту, чітко контролюються системою.

Фінальним етапом стала перевірка живої системи безпосередньо в браузері. Заміри продуктивності показали, що сервер відповідає на клієнтські запити дуже швидко — в середньому за 20 мілісекунд. Завдяки цьому інтерфейс взагалі не гальмує. Окремо протестували роботу відеоплеєра. Відставання трансляції від реального часу склало 15 секунд. Для протоколу HLS це цілком нормальний робочий показник, який створює потрібний буфер і захищає глядачів від зависання картинки. Також перевірили безпеку: якщо

спробувати смикнути закриті API-маршрути без дійсного токена, сервер миттєво відбиває такий запит помилкою доступу.

У підсумку, всі етапи тестування пройшли успішно. Тести підтвердили, що код працює стабільно, компоненти системи правильно взаємодіють між собою, а сама платформа повністю готова до запуску на бойовому сервері.

РОЗДІЛ 5. РОБОТА КОРИСТУВАЧА З СИСТЕМОЮ

У цьому розділі описано принципи взаємодії користувачів із вебзастосунком. Розглянуто базові сценарії роботи: від авторизації та запуску трансляції до оформлення платних підписок. Опис базується на функціональних можливостях клієнтської частини та її інтеграції з бекендом.

5.1 Автентифікація та доступ до платформи

Веб-застосунок підтримує взаємодію без автентифікація, але в цьому випадку функціонал стає “обрізаним”, наприклад, він може переглядати трансляції, але писати в чат не зможе.

Коли користувач натискає кнопку Log In його перенаправляє на сайт Auth0, процес іде за допомогою протоколу OIDC. Де він повинен пройти процес автентифікації або реєстрації, рис 5.1 зображено вигляд форми.

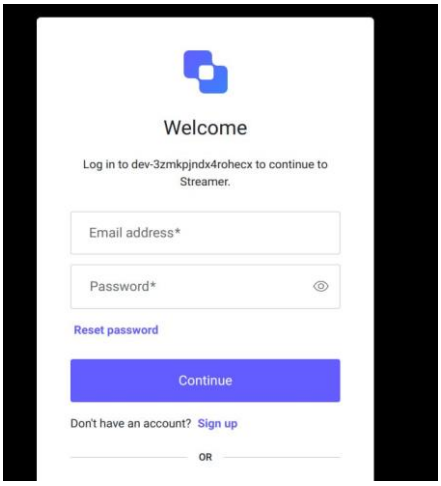

Рисунок 5.1 – Вікно автентифікації

Коли сервіс Auth0 підтверджує правильність введених даних, він генерує JWT-токен і автоматично повертає людину назад на сторінку застосунку. Далі в

роботу включається фронтенд: він забирає цей токен і кладе його в безпечне локальне сховище браузера. Відтепер система чітко розуміє, хто саме зайшов на сайт, і дозволяє користувачу працювати з усім функціоналом через його особистий профіль.

5.2 Запуск трансляції

Щоб розпочати трансляцію, автору спочатку потрібно встановити програму для ефіру, наприклад OBS Studio. Далі в особистому кабінеті треба скопіювати свій унікальний ключ і додати його в налаштування ПЗ. Це обов'язковий крок: саме за цим ідентифікатором медіасервер розуміє, від якого каналу надходить відеопотік. Інтерфейс сторінки з ключами показано на рис. 5.2.

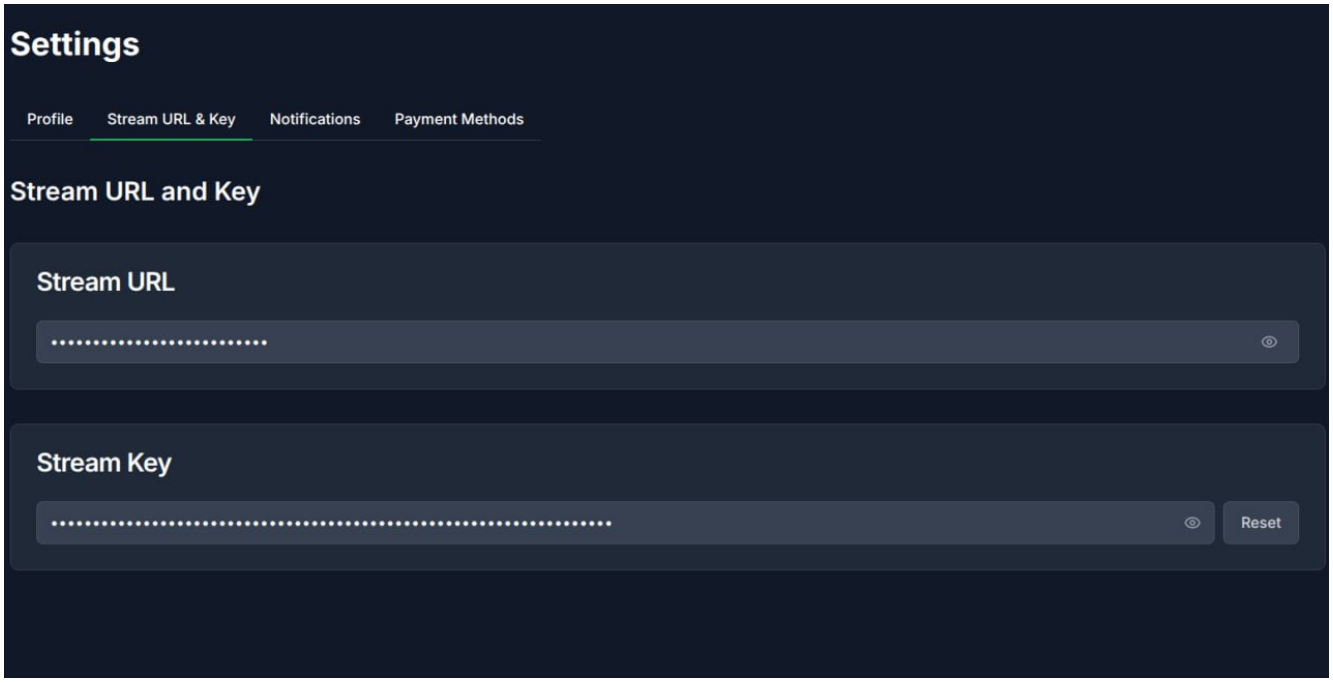

Рисунок 5.2 – Вікно ключів для запуска трансляції

Коли всі налаштування готові, автор трансляцаї натискає кнопку «Go Live» в OBS, і трансляція стартує. Відеопотік із програми одразу прямує на медіасервер. Саме він бере на себе всю важку роботу — підхоплює вхідний RTMP-сигнал і на льоту конвертує його в зручний для глядачів формат HLS.

Інтерфейс програми під час активного ефіру можна побачити на рис. 5.3.

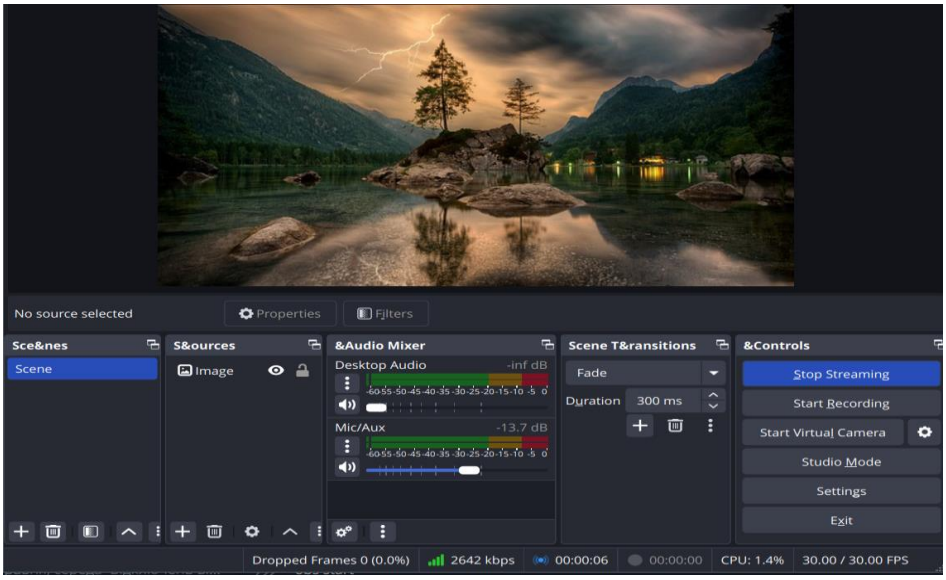

Рисунок 5.3 – Вікно OBS з активной трансляцією

Щойно трансляція успішно запустилась, на сторінці автора одразу активується відеоплеєр. Фронтенд-частина на Next.js починає безперервно звертатися до медіасервера, підвантажуючи свіжі відеофрагменти (чанки). Зовнішній вигляд сторінки в цей момент наведено на рис. 5.4. Варто також зазначити важливу архітектурну деталь: статус ефіру оновлюється в реальному часі завдяки технології GraphQL Subscriptions. Тобто глядачам взагалі не потрібно вручну оновлювати сторінку браузера, щоб побачити початок прямого включення.

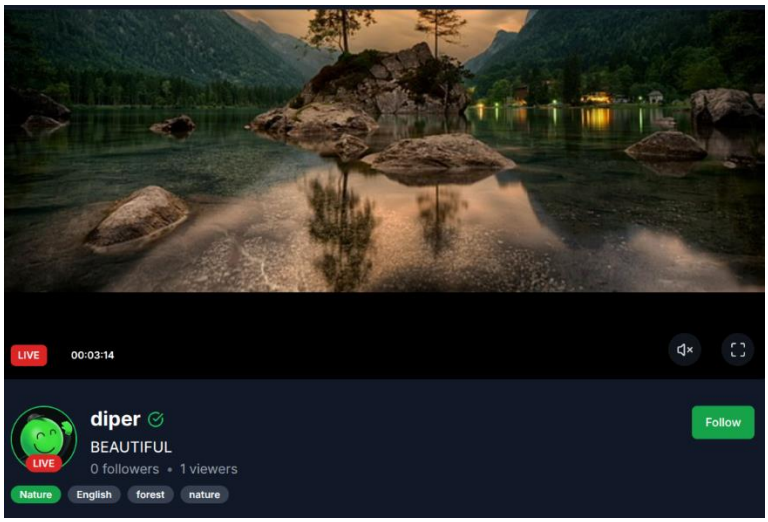

Рисунок 5.4 – Вікно користувача з активной трансляцією

Паралельно з цим на головній сторінці платформи автоматично з'являється картка нової трансляції (рис. 5.5). Цей блок містить свіжий кадр із прямого ефіру, назву ефіру, обрані теги, а також актуальний лічильник глядачів.

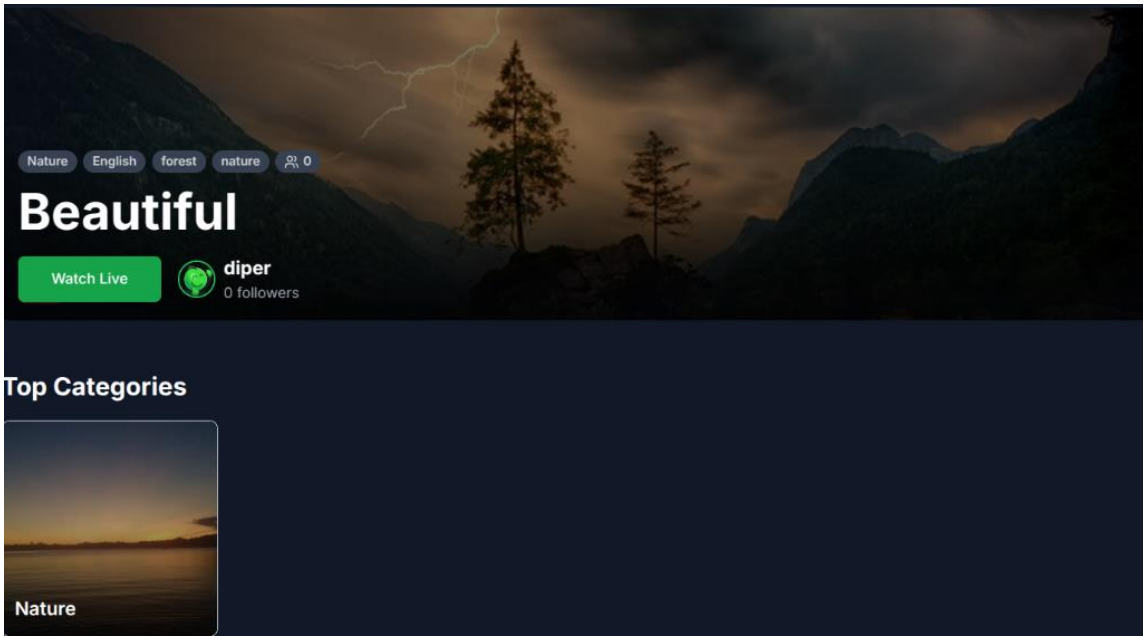

Рисунок 5.5 – Головна сторінка

Для зручності автора розроблено окрему панель керування (дашборд). На ній в одному місці зібрана вся ключова інформація про поточний ефір: автор може одночасно читати чат, бачити загальний час трансляції, слідкувати за динамікою глядачів та контролювати роботу модераторів. Інтерфейс цієї сторінки наведено на рис. 5.6.

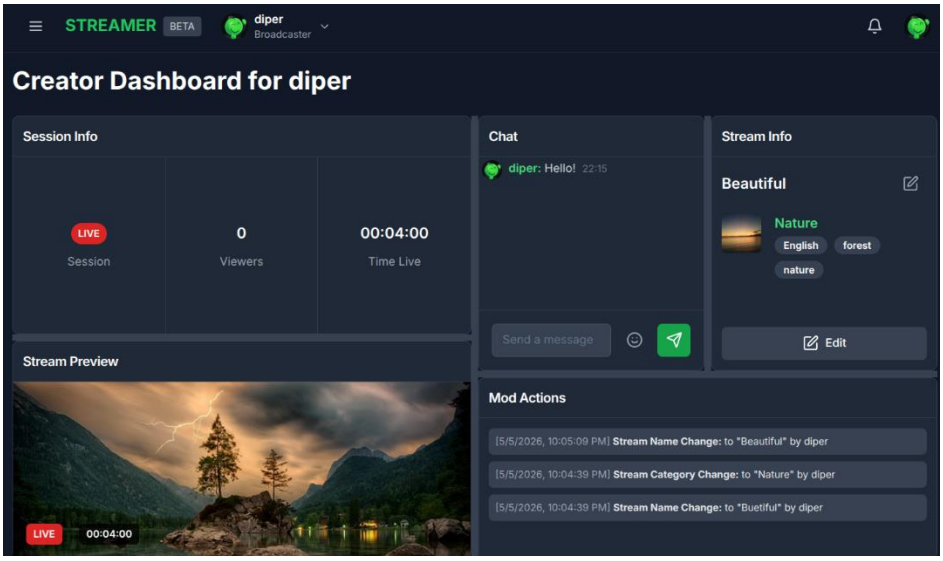

Рисунок 5.6 – Дашборд автора

5.3. Платні підписки

Функціонал платних підписок додано для того, щоб глядачі могли фінансово підтримувати своїх улюблених авторів. Коли користувач купує підписку, він отримує певні привілеї. Наприклад, його повідомлення в чаті починають виділятися іншим кольором, а поруч з'являється унікальний бейджик. Це гарно мотивує аудиторію і показує цінність платного контенту.

Щоб увімкнути прийом платежів, автор має обов'язково пройти верифікацію в системі Stripe Connect. Відповідна сторінка налаштувань показана на рис. 5.7. Коли автор натискає кнопку «Become a Partner», його перенаправляє на захищений сайт Stripe. Там потрібно вказати інформацію про себе: номер телефону, електронну пошту, адресу проживання і завантажити скан паспорта або ID-картки (рис. 5.8).

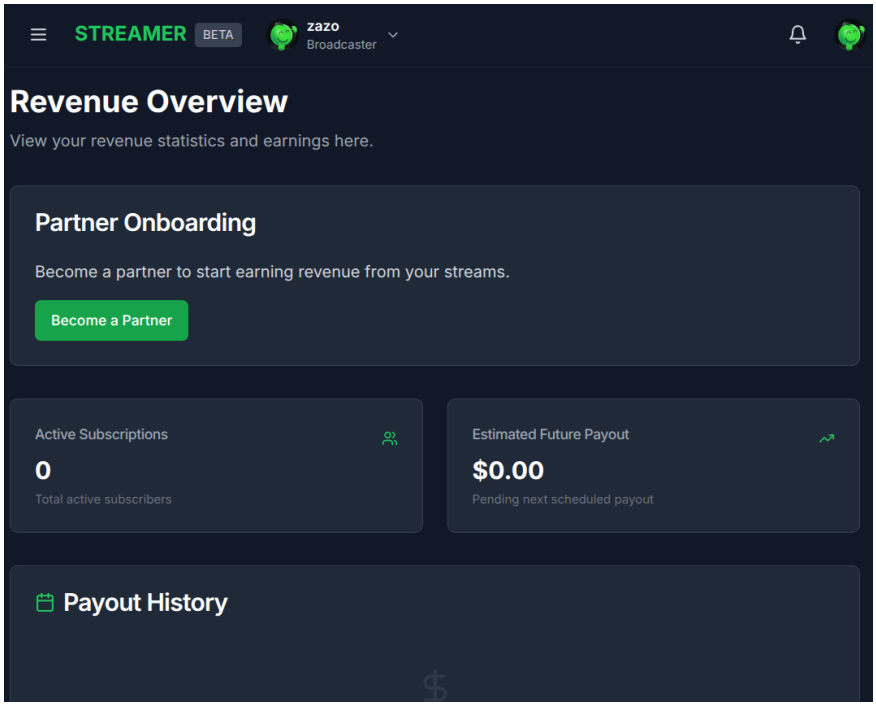

Рисунок 5.7 – Сторінка доходів

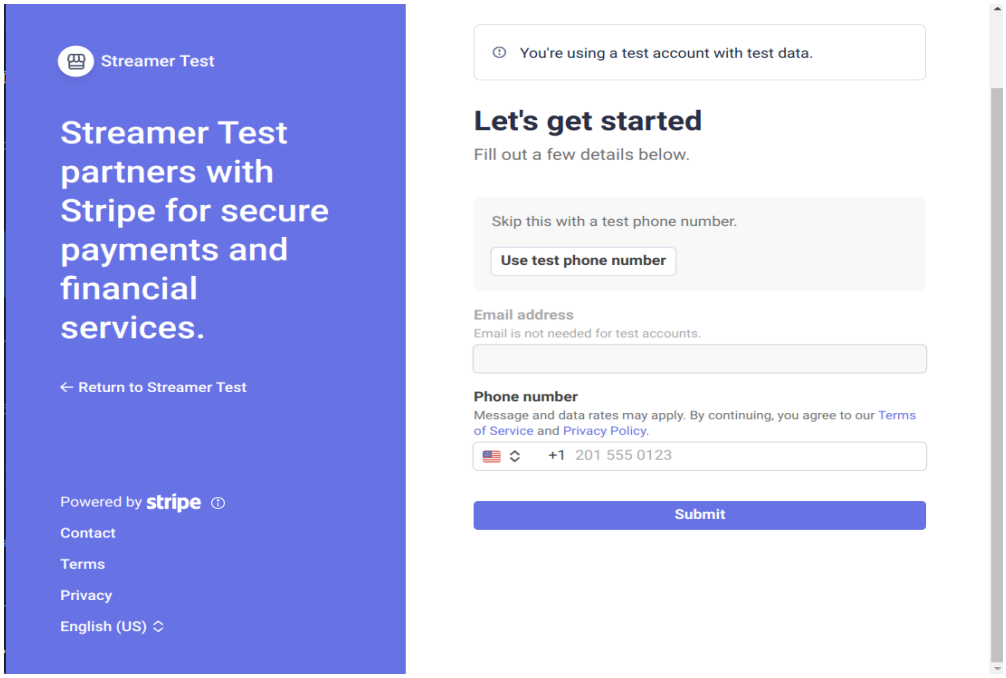

Рисунок 5.8 – Сторінка Stripe Connect

Після проходження верифікації на сторінці автора з’явилась кнопка для оформлення підписки рис 5.9

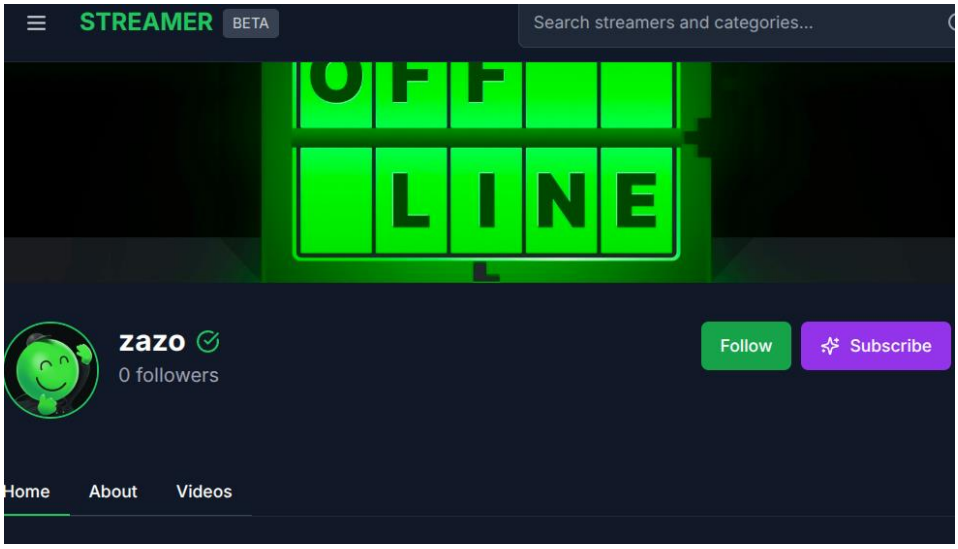

Рисунок 5.9 – Сторінка автора з підпискою

Після натискання цієї кнопки відкривається форма, де глядач прив'язує банківську картку і підтверджує платіж (рис. 5.10). Тут є важливий архітектурний момент: розроблена платформа взагалі не зберігає номерів

карток чи інших реквізитів. Уся робота з конфіденційною інформацією відбувається виключно на серверах Stripe, в базу записуються лише безпечні ідентифікатори. Про те, що гроші успішно списалися або статус підписки змінився, Stripe самостійно повідомляє бекенд за допомогою вебхуків

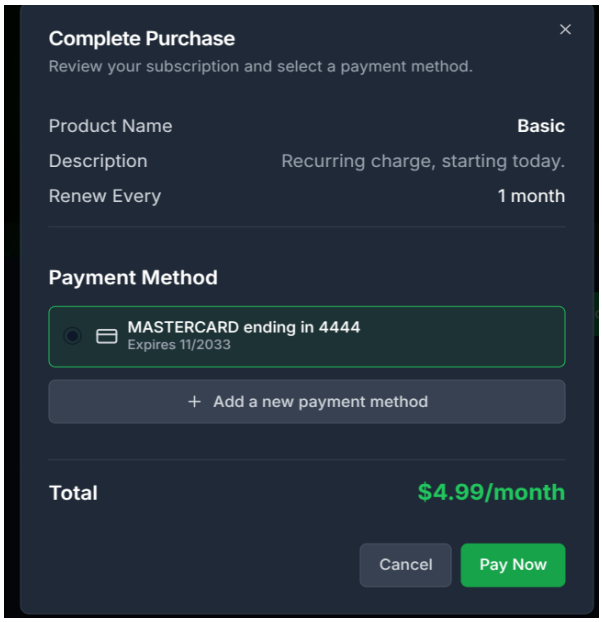

Рисунок 5.10 – Форма для оформлення підписки

Після успішного платежу автор у дашборді можна подивитись свої транзакції та кількість платних підписників рис 5.11

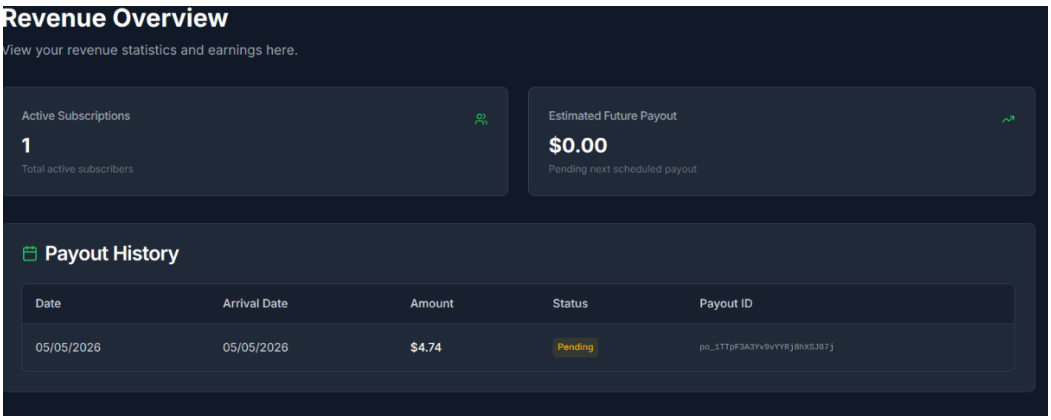

Рисунок 5.11 – Дашборд

5.4 Висновки до п'ятого розділу

У п'ятому розділі показано, як саме працює готова платформа з точки зору кінцевого користувача. Тут розібрано шлях від першого входу на сайт до

запуску власного ефіру та підключення монетизації.

Для реєстрації та входу підключили сервіс Auth0. Це повністю закриває питання безпеки паролів: сервер приймає та обробляє готові JWT-токени. При цьому сайт не змушує всіх обов'язково створювати акаунт — базовий перегляд ефірів доступний і звичайним гостям.

Процес запуску трансляції зроблено звичним для авторів. Автор копіює свій унікальний ключ в OBS, а далі система робить усе сама. Медіасервер приймає відео, конвертує його, а фронтенд завдяки технології GraphQL Subscriptions автоматично показує глядачам, що ефір почався. Жодних ручних оновлень сторінки не потрібно.

Окремо налаштовано прийом платежів для підписок. За це відповідає платформа Stripe. Головна перевага такої інтеграції — архітектура безпеки. Застосунок принципово не збирає реквізити банківських карток. Уся оплата проходить на стороні Stripe, а бекенд лише отримує вебхуки зі статусами транзакцій.

У результаті отримано повністю робочий клієнтський інтерфейс. Усі основні функції працюють стабільно, фронтенд миттєво реагує на події з сервера, а інтеграція зі сторонніми сервісами надійно захищає приватні дані користувачів.

ВИСНОВКИ

Результатом цієї роботи стала повністю робоча платформа для проведення відеотрансляцій. Вона об'єднує в собі швидкий бекенд на базі .NET, потужний медіасервер MediaMTX для обробки потокового відео та цілу інфраструктуру допоміжних баз даних і сервісів.

Вибір підходу Vertical Slice Architecture виправдав себе на всі сто відсотків. Кожна фіча платформи — будь то система повідомлень у чаті, налаштування ботів чи обробка збережених ефірів (VOD) — ізольована у своєму модулі. Це сильно спростило розробку і зробило код стійким до змін. Крім того, весь проєкт загорнутий у Docker-контейнери, тому розгортання на сервері проходить швидко і без зайвих проблем із налаштуванням середовища.
Щодо функціоналу, то зараз система успішно приймає сигнал через протокол RTMP, роздає його глядачам по HLS, автоматично завантажує записи ефірів у хмарне сховище (сумісне з AWS S3) та забезпечує живе спілкування в чатах. Авторизацію реалізовано через сервіс Auth0, що зняло зайвий головний біль із захистом облікових записів.

Етап тестування показав, що архітектура не просто гарно виглядає на папері, а й добре показує себе на практиці. Юніт- та інтеграційні тести підтвердили, що внутрішня логіка та взаємодія з базою даних працюють без збоїв. Системні перевірки видали чудові цифри: бекенд (GraphQL) відповідає в середньому за 20 мілісекунд. Відео по HLS йде із затримкою близько 15 секунд — це абсолютно нормальний показник для такого протоколу, який гарантує глядачам картинку без зависань. Механізми безпеки також відпрацьовують як треба, миттєво відбиваючи будь-які неавторизовані запити 401-ю помилкою.

У підсумку, вийшов не просто навчальний прототип, а цілком серйозний застосунок. Платформа працює стабільно, її легко підтримувати, вона готова до додавання нових фіч і цілком придатна для реального використання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper [Електронний ресурс] // Cisco Systems. – 2019. – Режим доступу: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html (дата звернення: 20.03.2026).
2. Fette I. RFC 6455. The WebSocket Protocol [Електронний ресурс] / I. Fette, A. Melnikov // Internet Engineering Task Force (IETF). – 2011. – Режим доступу: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 25.03.2026).
3. Understanding live streaming: RTMP [Електронний ресурс] // IPCamLive. – Режим доступу: https://www.ipcamlive.com/understanding-live-streaming-rtmp (дата звернення: 28.03.2026).
4. How WebRTC works [Електронний ресурс] // Cloudflare. – Режим доступу: https://www.cloudflare.com/learning/video/how-webrtc-works/ (дата звернення: 29.03.2026).
5. Low Latency Video [Електронний ресурс] // Twitch Help. – Режим доступу: https://help.twitch.tv/s/article/low-latency-video?language=en_US (дата звернення: 02.04.2026).
6. Harshit. Unveiling the Backbone of YouTube Live Streaming: A Deep Dive into YouTube's Architecture [Електронний ресурс] / Harshit // DEV Community. – Режим доступу: https://dev.to/wittedtech-by-harshit/unveiling-the-backbone-of-youtube-live-streaming-a-deep-dive-into-youtubes-architecture-and-real-time-video-processing-f6j (дата звернення: 04.04.2026).
7. Owncast Documentation [Електронний ресурс] // Owncast. – Режим доступу: https://owncast.online/docs/ (дата звернення: 05.04.2026).
8. Bogard J. Vertical Slice Architecture [Електронний ресурс] / J. Bogard // Jimmy Bogard Blog. – Режим доступу: https://www.jimmybogard.com/vertical-slice-architecture/ (дата звернення: 05.03.2026).

9. Fowler M. MonolithFirst [Електронний ресурс] / M. Fowler // martinfowler.com. – Режим доступу: https://martinfowler.com/bliki/MonolithFirst.html (дата звернення: 06.03.2026).
10. GraphQL FAQ [Електронний ресурс] // GraphQL. – Режим доступу: https://graphql.org/learn/#faq (дата звернення: 12.04.2026).
11. PostgreSQL: Documentation [Електронний ресурс] // PostgreSQL. – Режим доступу: https://www.postgresql.org/docs/current/index.html (дата звернення: 15.04.2026).
12. What is Object Storage? [Електронний ресурс] // Amazon Web Services. – Режим доступу: https://aws.amazon.com/what-is/object-storage/ (дата звернення: 20.04.2026).
13. WebSockets [Електронний ресурс] // Ably Realtime. – Режим доступу: https://ably.com/topic/websockets (дата звернення: 24.03.2026).
14. Cloud Pub/Sub overview [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/pubsub/docs/overview (дата звернення: 08.04.2026).
15. Understanding HTTP Long-Polling: Empowering Real-Time Communication [Електронний ресурс] // LinkedIn Pulse. – Режим доступу: https://www.linkedin.com/pulse/understanding-http-long-polling-empowering-real-time-communication-r-biufc/ (дата звернення: 22.03.2026).
16. Using server-sent events [Електронний ресурс] // MDN Web Docs. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events (дата звернення: 23.03.2026).
17. Session based authentication [Електронний ресурс] // SuperTokens. – Режим доступу: https://supertokens.com/blog/session-based-authentication (дата звернення: 10.04.2026).
18. Introduction to JSON Web Tokens [Електронний ресурс] // JWT.IO. – Режим доступу: https://www.jwt.io/introduction#what-is-json-web-token (дата звернення: 11.04.2026).

19. OAuth 2.0 Simplified [Електронний ресурс] // OAuth.com. – Режим доступу: https://www.oauth.com/ (дата звернення: 14.04.2026).
20. Official PCI Security Standards Council Site [Електронний ресурс] // PCI Security Standards Council. – Режим доступу: https://www.pcisecuritystandards.org/ (дата звернення: 24.04.2026).
21. Webhooks [Електронний ресурс] // Stripe Documentation. – Режим доступу: https://docs.stripe.com/webhooks (дата звернення: 26.04.2026).
22. Mosyan D. GraphQL vs REST API: similarities & differences [Електронний ресурс] / D. Mosyan // Medium. – Режим доступу: https://medium.com/@dmosyan/graphql-vs-rest-api-similarities-differences-857f9fc637a8 (дата звернення: 18.04.2026).
23. Fluent Assertions [Електронний ресурс] // Xceed. – Режим доступу: https://xceed.com/documentation/xceed-fluent-assertions-for-net/ (дата звернення: 02.05.2026).
24. .NET Documentation [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/ (дата звернення: 02.03.2026).
25. Learn Next.js [Електронний ресурс] // Next.js. – Режим доступу: https://nextjs.org/learn (дата звернення: 04.03.2026).
26. RabbitMQ Documentation [Електронний ресурс] // RabbitMQ. – Режим доступу: https://www.rabbitmq.com/docs (дата звернення: 10.03.2026).
27. Docker Documentation [Електронний ресурс] // Docker. – Режим доступу: https://docs.docker.com/ (дата звернення: 04.05.2026).

ДОДАТОК А

Лістинг програмного коду

Лістинг А.1 - Конфігурація та налаштування GraphQL-сервера

```
using HotChocolate.Types;
using Microsoft.Extensions.DependencyInjection;

namespace Streamers.Features.Shared.GraphQl;

public static class Extensions
{
   public static IServiceCollection AddGraphQl(this IServiceCollection services)
   {
      services
         .AddGraphQL()
         .AddGraphQLServer()
         .AddAuthorization()
         .AddFeaturesTypes()
         .AddProjections()
         .AddFiltering()
         .AddSorting()
         .AddSocketSessionInterceptor<AuthenticatedSocketSessionInterceptor>()
         .AddPagingArguments()
         .AddInMemorySubscriptions()
         .AddType<UploadType>()
         .ModifyRequestOptions(o => o.IncludeExceptionDetails = true)
         .ModifyCostOptions(o => o.EnforceCostLimits = false)
         .ModifyOptions(x => x.EnableFlagEnums = true);
      return services;
   }
```

```
}
```

Лістинг А.2 - Реалізація GraphQL-запитів для авторів.

```
 namespace Streamers.Features.Streams.GraphQl;

[QueryType]
public static partial class StreamQuery
{
   [Authorize]
   public static async Task<StreamSettingsDto> GetStreamSettings(IMediator mediat
or)
   {
      return await mediator.Send(new GetStreamSettings());
   }

   public static async Task<StreamDto> GetCurrentStream(string streamerId, IMedia
tor mediator)
   {
      return await mediator.Send(new GetCurrentStream(streamerId));
   }

   public static async Task<List<StreamDto>> GetTopStreams(IMediator mediator)
   {
      return await mediator.Send(new GetTopStreams());
   }
 }
```

Лістинг А.3 - Реалізація GraphQL-підписок для чату.

```
namespace Streamers.Features.Chats.GraphQl;

[SubscriptionType]
public static partial class ChatMessageSubscription
{
    [Subscribe]
    [Topic($"{nameof(ChatMessageCreated)}-{{{nameof(chatId)}}}")]
    public static ChatMessageDto ChatMessageCreated(
        Guid chatId,
        [EventMessage] ChatMessageDto message
    ) => message;

    [Subscribe]
    [Topic($"{nameof(ChatMessageDeleted)}-{{{nameof(chatId)}}}")]
    public static ChatMessageDto ChatMessageDeleted(
        Guid chatId,
        [EventMessage] ChatMessageDto message
    ) => message;

    [Subscribe]
    [Topic($"{nameof(ChatUpdated)}-{{{nameof(chatId)}}}")]
    public static ChatDto ChatUpdated(Guid chatId, [EventMessage] ChatDto chat) =>
 chat;
}
```

Лістинг А.4 - Опис доменної моделі ефіру.

```
namespace Streamers.Features.Streams.Models;
public record StreamUpdated(Stream Stream) : IDomainEvent;
public record StreamCreated(Stream Stream) : IDomainEvent;
```

```
public class Stream : Entity
{
    protected Stream() { }

    public Stream(
        Streamer streamer,
        string streamId,
        string title,
        DateTime started,
        List<StreamSource> streamSources,
        string language,
        List<Tag> tags,
        string? preview,
        Category? category
    )
    {
        Streamer = streamer;
        StreamerId = streamer.Id;
        StreamId = streamId;
        Title = title;
        Started = started;
        StreamSources = streamSources;
        Language = language;
        Tags = tags;
        Preview = preview;
        Active = true;
        Category = category;
        Raise(new StreamCreated(this));
        SetActive(Active);
    }
```

```
    public string StreamId { get; set; }
  …
    public string? Preview { get; set; }

    public void SetActive(bool active)
    {
        Active = active;
        if (!active)
        {
            Duration = DateTime.UtcNow.Subtract(Started).TotalSeconds;
        }
        Raise(new StreamUpdated(this));
    }

    public void SetCurrentViewers(long viewers)
    {
        CurrentViewers = viewers;
        Raise(new StreamUpdated(this));
    }
}
```

Лістинг А.5 - Налаштування Apollo Client для React.

```
export function MyApolloProvider({ children }: { children: React.ReactNode }) {
  const { getAccessTokenSilently, isAuthenticated } = useAuth0();

  const client = useMemo(() => {
    // HTTP посилання для стандартних запитів (Query/Mutation)
```

```
const httpLink = new HttpLink({
  uri: process.env.NEXT_PUBLIC_GRAPHQL_URI,
});

// Auth Middleware для додавання JWT-токена в хедери
const authLink = setContext(async (_, { headers }) => {
  let token = null;
  if (isAuthenticated) {
    try {
      token = await getAccessTokenSilently();
    } catch (e) {
      console.error("Auth Error:", e);
    }
  }
  return {
    headers: {
      ...headers,
      ...(token ? { Authorization: `Bearer ${token}` } : {}),
    },
  };
});

// WebSocket посилання для підписок (Subscriptions)
const wsLink = typeof window !== "undefined"
    ? new GraphQLWsLink(
        createClient({
          url: process.env.NEXT_PUBLIC_GRAPHQL_WS_URI!,
          connectionParams: async () => {
            const token = isAuthenticated ? await getAccessTokenSilently() : null;
            return token ? { Authorization: `Bearer ${token}` } : {};
```

```
            },
          })
      )
      : null;

  // Розділення трафіку (Split Link): WS для підписок, HTTP для решти
  const splitLink = typeof window !== "undefined" && wsLink
    ? split(
        ({ query }) => {
          const def = getMainDefinition(query);
          return (
            def.kind === "OperationDefinition" &&
            def.operation === "subscription"
          );
        },
        wsLink,
        authLink.concat(httpLink)
      )
      : authLink.concat(httpLink);

  return new ApolloClient({
    link: splitLink,
    cache: new InMemoryCache(),
  });
 }, [getAccessTokenSilently, isAuthenticated]);

 return <ApolloProvider client={client}>{children}</ApolloProvider>;
}
```

Лістинг А.6 - Приклади GraphQL-операцій для чату.

```
# Мутація для відправки повідомлення в чат
mutation CreateMessage($request: CreateMessageInput!) {
  createMessage(request: $request) {
    messageId
  }
}

# Запит для отримання історії повідомлень з пагінацією
query GetChatMessages(
  $chatId: UUID!
  $last: Int
  $before: String
) {
  chatMessages(
    chatId: $chatId
    last: $last
    before: $before
  ) {
    nodes {
      id
      message
      createdAt
      sender {
        id
        userName
        avatar
      }
    }
    pageInfo {
      hasPreviousPage
```

```
            startCursor
        }
    }
}

# Підписка на нові повідомлення в реальному часі (WebSockets)
subscription ChatMessageCreated($chatId: UUID!) {
    chatMessageCreated(chatId: $chatId) {
        id
        message
        createdAt
        sender {
            id
            userName
            avatar
        }
    }
}
```

Лістинг А.7 - Реалізація React-компонента плеєра для HLS-трансляцій.

```
"use client"

import React, { useRef, useState } from "react"
import ReactHlsPlayer from "react-hls-player";
import { Button } from "@/components/ui/button";
import { VolumeX, Volume2, Maximize, Minimize } from "lucide-react";

interface StreamPlayerProps {
  url: string;
  isLive?: boolean;
```

```
}

export const StreamPlayer = React.memo(function StreamPlayer({ url, isLive }: Stre
amPlayerProps) {
  const playerRef = useRef<HTMLVideoElement>(null);
  const containerRef = useRef<HTMLDivElement>(null);
  const [isMuted, setIsMuted] = useState(true);
  const [isFullscreen, setIsFullscreen] = useState(false);

  // Перемикання звуку
  const toggleMute = () => {
    if (playerRef.current) {
      playerRef.current.muted = !playerRef.current.muted;
      setIsMuted(playerRef.current.muted);
    }
  };

  // Перемикання повноекранного режиму
  const toggleFullscreen = () => {
    if (!document.fullscreenElement && containerRef.current) {
      containerRef.current.requestFullscreen();
      setIsFullscreen(true);
    } else {
      document.exitFullscreen();
      setIsFullscreen(false);
    }
  };

  return (
    <div ref={containerRef} className="relative w-full h-full bg-black group">
```

```
      <ReactHlsPlayer
        playerRef={playerRef}
        src={url}
        autoPlay={true}
        controls={false}
        width="100%"
        height="100%"
        muted={isMuted}
        hlsConfig={{ lowLatencyMode: true }}
      />

      {/* Кастомні елементи керування (Overlay) */}
      <div className="absolute bottom-0 left-0 right-0 p-4 bg-gradient-to-t from-
black/80 to-transparent opacity-0 group-hover:opacity-100 transition-opacity">
        <div className="flex items-center justify-between">
          <div className="flex items-center gap-2">
            <Button variant="ghost" size="icon" onClick={toggleMute} className="te
xt-white">
              {isMuted ? <VolumeX /> : <Volume2 />}
            </Button>
            {isLive && (
              <span className="bg-red-600 text-white text-xs px-2 py-1 rounded font-
bold uppercase">
                Live
              </span>
            )}
          </div>

          <Button variant="ghost" size="icon" onClick={toggleFullscreen} className
="text-white">
```

```
            {isFullscreen ? <Minimize /> : <Maximize />}
          </Button>
        </div>
      </div>
    </div>
  );
});
```

Лістинг А.8 - Реалізація головного макета з адаптивним сайдбаром.

```
"use client"
export function MainLayout({ children }: { children: React.ReactNode }) {
  const isMobile = useIsMobile();
  const [sidebarOpen, setSidebarOpen] = useState(true);

  // Автоматичне приховування сайдбару на мобільних пристроях
  useEffect(() => {
    setSidebarOpen(!isMobile);
  }, [isMobile]);

  return (
    <div className="min-h-screen bg-gray-900 text-white flex">
      {/* Бокова панель (Sidebar) з анімацією виїзду */}
      <div className={cn(
        "fixed inset-y-0 left-0 z-50 w-64 transform transition-transform duration-
200 ease-in-out",
        sidebarOpen ? "translate-x-0" : "-translate-x-full"
      )}>
        <Sidebar onClose={() => setSidebarOpen(false)} />
      </div>
```

```
      {/* Затемнення фону для мобільних */}
      {sidebarOpen && isMobile && (
        <div
          className="fixed inset-0 z-40 bg-black/50 lg:hidden"
          onClick={() => setSidebarOpen(false)}
        />
      )}

      {/* Основний контент зі зміщенням */}
      <div className={cn(
        "flex-1 flex flex-col transition-all duration-200",
        sidebarOpen && !isMobile ? "lg:ml-64" : "lg:ml-0"
      )}>
        <Navbar onMenuClick={() => setSidebarOpen(!sidebarOpen)} />

        <main className="flex-1 pt-16 overflow-y-auto">
          {children}
        </main>
      </div>
    </div>
  )
}
```

Лістинг А.9 - Конфігурація медіасервера MediaMTX із вебхуками.

```
api: yes
rtmp: yes
rtmpAddress: :1935
authMethod: http
authHTTPAddress: http://streamer-api/rtmp/checkToken
```

```
runOnConnectRestart: no
hlsPartDuration: 500ms
authInternalUsers:
  - user: admin
    pass: admin123
    permissions:
      - action: publish
      - action: read
      - action: api
  - user: any
    pass:
    ips: [ ]
    permissions:
      - action: read
        path:
authHTTPExclude:
  - action: api
  - action: read

pathDefaults:
  runOnReady: python3 /usr/local/bin/send_webhook.py "$MTX_SOURCE_ID" "live
StreamStarted" "$MTX_PATH" "vod" "cap.default.router" "mediamtx"
  runOnReadyRestart: no

  runOnNotReady: python3 /usr/local/bin/send_webhook.py "$MTX_SOURCE_ID" "
liveStreamEnded" "$MTX_PATH" "vod" "cap.default.router" "mediamtx"


  runOnRead: python3 /usr/local/bin/send_webhook.py "$MTX_SOURCE_ID" "strea
mRead" "$MTX_PATH" "vod" "cap.default.router" "mediamtx"
```

```
  runOnReadRestart: no

  runOnUnread: python3 /usr/local/bin/send_webhook.py "$MTX_SOURCE_ID" "str
eamUnread" "$MTX_PATH" "vod" "cap.default.router" "mediamtx"

paths:
  all_others:
```

Лістинг А.10 - Реалізація сервісу обробки та завантаження VOD у хмарне сховище.

```
const { runFfmpegHls, createPreview, getDurationMs } = require("./ffmpeg");
const { ensureTmpFolder } = require("./utils");
const { connectRabbit, EXCHANGE, FINISHED_KEY, QUEUE } = require("./rabbi
tmq");
const { createRcloneConfig, RCLONE_CONFIG_PATH } = require("./s3");
const { exec } = require("child_process");
const { rmSync } = require("node:fs");
const path = require("node:path");
const chokidar = require("chokidar");
const ONE_YEAR = 31536000;
const fs = require("fs");
const AWS_BUCKET = process.env.AWS_BUCKET;

function runRcloneSync(localPath, remotePath, cacheSeconds = 0) {
   return new Promise((resolve, reject) => {
```

```
        const cmd = [
            "rclone copy",
            `"${localPath}"`,
            `"${remotePath}"`,
            "--config /config/rclone.conf",
            "--s3-no-check-bucket",
            "--retries 3",
            "--low-level-retries 10",
            "--progress",
            "--s3-acl public-read",
            `--header "Cache-Control: ${cacheSeconds}"`
        ].join(" ");

        console.log("Running rclone:", cmd);

        const proc = exec(cmd, (err, stdout, stderr) => {
            if (err) {
                console.error("rclone failed:", stderr);
                return reject(err);
            }
            console.log("rclone output:", stdout);
            resolve();
        });

        proc.stdout.on("data", d => process.stdout.write(d));
        proc.stderr.on("data", d => process.stderr.write(d));
    });
}

async function main() {
```

```
createRcloneConfig();
const { channel } = await connectRabbit();

console.log(`[*] Waiting for messages in "${QUEUE}"`);
await channel.consume(QUEUE, async (msg) => {
  if (!msg) return;

  const payload = JSON.parse(msg.content.toString());
  console.log(" [x] Received:", payload);

  const vodId = payload.VodId;
  const hlsUrl = payload.HlsSource;
  const tmpDir = ensureTmpFolder(vodId);
  const remotePath = `aws:${AWS_BUCKET}/vods/${vodId}`;

  try {
    await runFfmpegHls(hlsUrl, tmpDir);
    console.log("HLS recording finished:", tmpDir);

    const hlsMainFile = path.join(tmpDir, "index.m3u8");
    const preview = await createPreview(hlsMainFile, tmpDir);
    console.log("Preview created:", preview.previewPath);

    await runRcloneSync(tmpDir, remotePath, ONE_YEAR);
    console.log("All files uploaded to S3");

    const durationMs = await getDurationMs(hlsMainFile);

    channel.publish(
      EXCHANGE,
```

```
            FINISHED_KEY,
            Buffer.from(JSON.stringify({
              VodId: vodId,
              Path: `vods/${vodId}/index.m3u8`,
              Preview: `vods/${vodId}/preview.jpg`,
              Duration: durationMs
            }))
          );
          console.log(" [x] Sent finished message");
        } catch (err) {
          console.error("Processing failed:", err);
        } finally {
          try {
            rmSync(tmpDir, { recursive: true, force: true });
          } catch (e) {
            console.error("Failed to clean tmp dir:", e);
          }
        }

        channel.ack(msg);
      }, { noAck: false });
}


main().catch(console.error);
```

Лістинг А.11 - Юніт-тести для генератора людинозрозумілих посилань (slug).

```
using FluentAssertions;
using Streamers.Features.Categories.Services;
using Xunit;
```

```
namespace Streamers.Features.UnitTests.Categories.Services;

public class SlugGeneratorTests
{
    private readonly ISlugGenerator _slugGenerator = new SlugGenerator();

    [Theory]
    [InlineData("Hello World", "hello-world")]
    [InlineData("  leading and trailing spaces  ", "leading-and-trailing-spaces")]
    [InlineData("Multiple   Spaces", "multiple-spaces")]
    [InlineData("Special!@#$%^&*()_+=Chars", "specialchars")]
    [InlineData("áéíóú", "aeiou")]
    [InlineData("Some Diacritics like ñ and ü", "some-diacritics-like-n-and-u")]
    [InlineData("An Already-Valid-Slug", "an-already-valid-slug")]
    [InlineData("", "")]
    [InlineData("  ", "")]
    public void GenerateSlug_ShouldReturnCorrectSlug(string input, string expected)
    {
        // Act
        var result = _slugGenerator.GenerateSlug(input);

        // Assert
        result.Should().Be(expected);
    }
}
```

Лістинг А.12 - Інтеграційні тести для перевірки створення повідомлень.

```
public class CreateMessageTests : BaseIntegrationTest
```

```
{
    public CreateMessageTests(StreamerWebApplicationFactory factory)
        : base(factory) { }

    [Fact]
    public async Task CreateMessage_ValidRequest_CreatesChatMessage()
    {
        // Arrange
        var sender = await CreateStreamer(Guid.NewGuid().ToString());
        // chatOwner is not needed here since Chat is created with sender

        CurrentUser.MakeAuthenticated(sender.Id);

        var command = new CreateMessage(sender.Chat.Id, "Hello chat!", null);

        // Act
        var response = await Sender.Send(command);

        // Assert
        response.Should().NotBeNull();
        response.MessageId.Should().NotBeEmpty();

        var chatMessage = await DbContext.ChatMessages.FirstAsync(m => m.Id ==
response.MessageId);
        chatMessage.Should().NotBeNull();
        chatMessage.Sender.Id.Should().Be(sender.Id);
        chatMessage.Message.Should().Be("Hello chat!");
        chatMessage.Chat.Id.Should().Be(sender.Chat.Id);
    }
```

```
    [Fact]
    public async Task
CreateMessage_SenderNotFound_ThrowsStreamerNotFoundException()
    {
        // Arrange
        // chatOwner and chat are not needed here

        CurrentUser.MakeAuthenticated("non-existent-sender");
        // Mock the permission service to avoid issues if a chat is found


        var command = new CreateMessage(Guid.NewGuid(), "Hello chat!", null); //
Use a random chat ID

        // Act
        var act = async () => await Sender.Send(command);

        // Assert
        await act.Should()
            .ThrowAsync<StreamerNotFoundException>()
            .WithMessage("Streamer with id 'non-existent-sender' was not found.");
    }

    [Fact]
    public async Task
CreateMessage_ChatNotFound_ThrowsChatNotFoundException()
    {
        // Arrange
        var sender = await CreateStreamer(Guid.NewGuid().ToString());
        CurrentUser.MakeAuthenticated(sender.Id);
```

```
        var command = new CreateMessage(Guid.NewGuid(), "Hello chat!", null);

        // Act
        var act = async () => await Sender.Send(command);

        // Assert
        await act.Should().ThrowAsync<ChatNotFoundException>();
    }

    [Fact]
    public async Task
CreateMessage_ReplyMessageNotFound_ThrowsMessageNotFoundException()
    {
        // Arrange
        var sender = await CreateStreamer(Guid.NewGuid().ToString());
        // chatOwner is not needed here

        CurrentUser.MakeAuthenticated(sender.Id);

        var command = new CreateMessage(sender.Chat.Id, "Hello chat!",
Guid.NewGuid());

        // Act
        var act = async () => await Sender.Send(command);

        // Assert
        await act.Should().ThrowAsync<MessageNotFoundException>();
    }
```

```
    [Fact]
    public async Task
CreateMessage_MessageTooLong_ThrowsValidationException()
    {
        // Arrange
        var sender = await CreateStreamer(Guid.NewGuid().ToString());

        CurrentUser.MakeAuthenticated(sender.Id);

        var longMessage = new string('a', 251); // Message > 250 chars
        var command = new CreateMessage(sender.Chat.Id, longMessage, null);

        // Act
        var act = async () => await Sender.Send(command);

        // Assert
        await act.Should().ThrowAsync<MyValidationException>();
    }
}
```