Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
________ Олена ОЛЬХОВСЬКА
(підпис)

«___»_________202_ р.

КВАЛІФІКАЦІЙНА РОБОТА

на тему

«РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КРОСПЛАТФОРМНОГО МІКРОСЕРВІСНОГО АГРЕГАТОРА ПОТОКОВОГО АУДІОКОНТЕНТУ»

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня бакалавра

Виконавець роботи Тернопольський Давід Ігорович
__________«____»______ 202_ р.

Науковий керівник к. ф.-м. н., доцент, Ольховська Олена Володимирівна
__________«____»______ 202_ р.

Рецензент ____

ПОЛТАВА 2026

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ, ТЕРМІНІВ.. 4

ВСТУП.. 5

1. ПОСТАНОВКА ЗАДАЧ.. 8

1.1. Аналіз предметної області та актуальність інтеграції музичних платформ8

2. ІНФОРМАЦІЙНИЙ ОГЛЯД АНАЛОГІВ 10

2.1. Огляд функціональності та обмежень платформеного інтегратора FreeYourMusic.. 10

2.2 Огляд функціональності та обмежень платформеного інтегратора Soundiiz 11

2.3 Висновки щодо аналізу аналогів 13

3. ТЕОРЕТИЧНА ЧАСТИНА ТА ПРОЄКТУВАННЯ СИСТЕМИ....... 15

3.1. Проєкт архітектури розподіленої мікросервісної системи на базі Nginx 15

3.2. Алгоритми обробки аудіопотоків та криптографічного дешифрування 16

3.2.1. Логіка парсингу та стримінгу AAC HLS потоків з SoundCloud 16

3.2.2. Алгоритм дешифрування аудіоданих Deezer FLAC за допомогою Blowfish та синхронізація з S3 17

3.3. Проєктування механізмів розподіленого кешування у Redis 19

3.4. Реляційне моделювання структури бази даних у PostgreSQL 21

3.5. Уніфікація гетерогенних медіаданих та патерн проєктування на базі Trait Into.. 25

4. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ІНТЕРФЕЙС КОРИСТУВАЧА.... 30

4.1. Обґрунтування вибору технологічного стека 30

4.1.1. Вибір мови Rust, фреймворку Axum та асинхронного драйвера SQLx для бекенду 30

4.1.2. Переваги клієнтського фреймворку Angular та контейнеризації через Docker 31

4.2. Опис програмної реалізації мікросервісів та бібліотек 33
4.2.1. Реалізація сервісів авторизації (auth-service) та керування медіатекою (library-service) 33
4.2.2. Розробка модулів інтеграції стримінгів (soundcloud-service, deezer-service) та спільних бібліотек (cache-lib, utils-lib, email-lib) 34
4.3. Конфігурація розгортання та маршрутизації в Docker-контейнерах через Nginx 35
4.4. Опис інтерфейсу користувача та результатів тестування застосунку 39
ВИСНОВКИ **41**
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ **44**
ДОДАТОК А **46**

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
API	API (Application Programming Interface)
CBC	(Cipher Block Chaining)
СКБД	Система керування базами даних

ВСТУП

Метою кваліфікаційної роботи є проєктування, програмна реалізація та розгортання високопродуктивної мікросервісної архітектури кросплатформного агрегатора музичних сервісів, яка забезпечує безшовну інтеграцію, дешифрування та синхронізоване відтворення аудіопотоків форматів AAC HLS та Deezer FLAC в єдиному користувацькому інтерфейсі з оптимізацією затримок через механізми розподіленого кешування та S3-сумісного об'єктного зберігання.

Для досягнення поставленої мети необхідно вирішити такі задачі дослідження:

1. Проаналізувати архітектурні особливості, формати передачі даних та криптографічні механізми захисту потокового аудіо в сучасних сервісах SoundCloud та Deezer.

2. Спроектувати розподілену мікросервісну архітектуру системи із чітким розділенням зон відповідальності (інтеграція платформ, авторизація, керування медіатекою) та розробити схему маршрутизації трафіку через зворотний проксі-сервер Nginx.

3. Розробити алгоритмічне та програмне забезпечення на мові Rust для взаємодії з API SoundCloud, що включає парсинг HLS маніфестів, та API Deezer, реалізувавши модуль симетричного дешифрування Blowfish-CBC в асинхронному режимі.

4. Проектувати реляційну модель бази даних PostgreSQL для збереження облікових даних та метаінформації треків, а також розробити схему організації об'єктного сховища MinIO (S3) для локального кешування аудіосегментів.

5. Розробити механізми розподіленого кешування у Redis для оптимізації доступу до високочастотних даних (JWT-токени, верифікаційні коди, стани черг відтворення, структури плейлистів).

6. Реалізувати реактивний клієнтський застосунок на базі фреймворку Angular із використанням архітектури State Management.

7. Провести контейнеризацію всієї інфраструктури за допомогою Docker та Docker Compose, виконати інтеграційне тестування системи в умовах симуляції мережевих запитів.

Об'єкт дослідження. Об'єктом дослідження є процеси агрегації, маршрутизації, криптографічної обробки та синхронізації різнорідних цифрових аудіопотоків у розподілених інформаційних системах.

Предмет дослідження. Предметом дослідження є мікросервісна архітектура, алгоритми асинхронного стримінгу, методи Blowfish-дешифрування, розподілені сховища даних (PostgreSQL, Redis, MinIO S3) та інструменти розробки програмного забезпечення (Rust/Axum, Angular, Nginx, Docker), що використовуються для створення кросплатформного музичного агрегатора.

Методи дослідження. Для вирішення поставлених задач у роботі застосовано комплекс наукових та інженерних методів:

- *методи системного аналізу та теорії інформаційних систем* — для дослідження предметної області, вивчення існуючих аналогів та формування технічних вимог до агрегатора;
- *методи об'єктно-орієнтованого аналізу та проектування (UML)* — для побудови архітектурних діаграм прецедентів, послідовності та структури бази даних;

- *методи прикладної криптографії та побітової обробки даних* — для реалізації алгоритму дешифрування Blowfish медіаконтенту на рівні байтових масивів;
- *методи асинхронного та конкурентного програмування (Async/Await, Actor Model концепції)* — для забезпечення паралельної обробки мережевих запитів без блокування потоків ОС на мові Rust;
- *методи реляційного моделювання та теорії кешування* — для оптимізації схем даних у PostgreSQL та структур "ключ-значення" в Redis;
- *методи компонентного вебпрограмування та реактивних потоків (RxJS)* — для побудови користувацького інтерфейсу на Angular;
- *методи контейнеризації та віртуалізації* — для пакування компонентів системи в ізольовані Docker-контейнери та налаштування локальної інфраструктури розгортання.

Наукова новизна та практичне значення отриманих результатів полягають у створенні відкритої архітектури медіа-агрегатора.

1. ПОСТАНОВКА ЗАДАЧ

1.1. Аналіз предметної області та актуальність інтеграції музичних платформ

Сучасний етап розвитку індустрії цифрового дистриб'ютування медіаконтенту характеризується глибокою фрагментацією ринку стримінгових послуг. Провідні платформи, такі як SoundCloud, Deezer, Spotify та Apple Music, функціонують як ізольовані екосистеми з власними закритими протоколами передачі даних, унікальними алгоритмами шифрування, специфічними API (Application Programming Interface) та несумісними структурами метаданих. Це створює суттєві незручності для кінцевого користувача, який змушений оперувати декількома клієнтськими застосунками, здійснювати монотонний ручний пошук аудіоматеріалів та сплачувати за кілька ізольованих підписок задля доступу до унікальних релізів, що представлені лише на конкретних майданчиках (наприклад, аматорські мікси на SoundCloud або lossless-аудіо на Deezer) [1-15].

З інженерної точки зору, розробка кросплатформного агрегатора музичних сервісів ускладнюється різнорідністю форматів потокового мовлення. Платформа SoundCloud переважно використовує протокол асинхронного стримінгу HLS (HTTP Live Streaming) із кодуванням аудіо в форматі AAC, що вимагає динамічного парсингу маніфестів .m3u8 та послідовного завантаження сегментів. У той же час сервіс Deezer надає доступ до високоякісних аудіопотоків у форматі FLAC, захищених пропрієтарним шифруванням на базі алгоритму Blowfish у режимі CBC (Cipher Block Chaining). Інтеграція цих технологій в межах єдиного середовища вимагає створення високопродуктивного бекенду, здатного

здійснювати дешифрування в реальному часі без затримок у відтворенні (jitter) та ефективно кешувати дані.

Традиційні монолітні архітектури не здатні забезпечити належний рівень масштабованості, відмовостійкості та низької затримки (latency) при високій інтенсивності мережевих запитів. Застосування мови програмування Rust, відомої своєю концепцією безпеки роботи з пам'яттю без використання збирача сміття (Garbage Collector) та високою ефективністю асинхронного розпаралелювання (компіляція на базі LLVM, рантайм tokio), дозволяє розробити мікросервісну систему з мінімальним споживанням системних ресурсів. Використання сучасного вебфреймворку axum та асинхронного драйвера sqlx забезпечує максимальну пропускну здатність при взаємодії з реляційною СУБД PostgreSQL. Оптимізація навантаження на зовнішні API реалізується через дворівневу систему збереження даних: оперативне кешування динамічних сутностей (сесій користувачів, черг відтворення, токенів авторизації) у СКБД Redis та довготривале об'єктне зберігання медіафайлів у S3-сумісному сховищі MinIO. На фронтенд-рівні фреймворк Angular дозволяє реалізувати реактивний, компонентно-орієнтований інтерфейс користувача для безшовного керування єдиним медіапростором.

Таким чином, розробка архітектури та програмного забезпечення мікросервісного агрегатора музичних платформ на базі Rust та Angular є актуальною науково-практичною задачею, спрямованою на вирішення проблеми інтеграції потокових сервісів у межах єдиної високопродуктивної інформаційної системи.

2. ІНФОРМАЦІЙНИЙ ОГЛЯД АНАЛОГІВ

2.1. Огляд функціональності та обмежень платформеного інтегратора FreeYourMusic

FreeYourMusic є спеціалізованим програмним забезпеченням, орієнтованим на трансфер, синхронізацію та резервне копіювання медіатек між більш ніж 20 музичними платформами, включаючи Deezer, SoundCloud, Spotify та Apple Music. Основним функціональним вузлом є алгоритм маппінгу (зіставлення) метаданих [6]. Програма сканує вихідний плейлист, виділяє теги (назва треку, виконавець, альбом) і за допомогою евристичних алгоритмів шукає найточніший збіг у базі даних цільового сервісу.

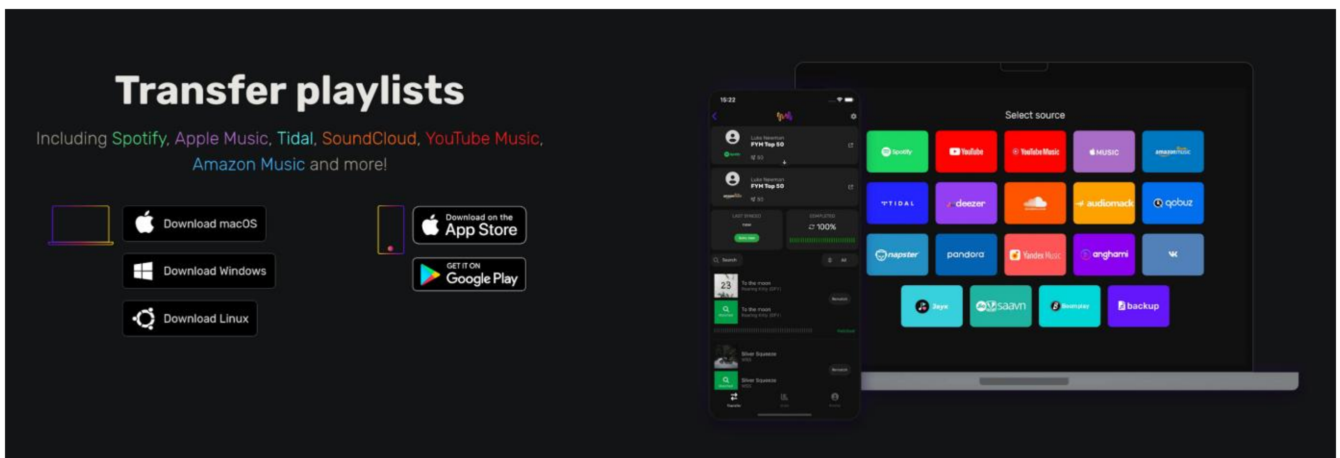

Рисунок 2.1 – **FreeYourMusic промо**

Переваги:

- Широке покриття стримінгових сервісів. Підтримка великої кількості комерційних майданчиків дозволяє здійснювати міграцію даних у будь-яких напрямках.

- Автоматизована синхронізація. Наявність фонових процесів для періодичного оновлення плейлистів між різними акаунтами користувача.
- Експорт даних. Можливість збереження списків відтворення у локальні файли форматів CSV або JSON для подальшого відновлення.

Недоліки:

- Відсутність функції безпосереднього відтворення. FreeYourMusic є виключно інструментом передачі даних. Він не функціонує як медіаплеєр і не дозволяє прослуховувати музику в реальному часі.
- Похибки алгоритмів маппінгу. При розбіжностях у метаданих або наявності реміксів алгоритм часто додає некоректні треки або взагалі не знаходить копію у цільовому сервісі.
- Клієнт-орієнтована архітектура. Більшість обчислювальних операцій та мережевих запитів виконуються безпосередньо на пристрої користувача, що створює високе навантаження на процесор та оперативну пам'ять, а також вимагає постійно відкритого застосунку для завершення синхронізації.

2.2 Огляд функціональності та обмежень платформеного інтегратора Soundiiz

Soundiiz — це хмарний сервіс для централізованого керування музичними акаунтами. Платформа дозволяє об'єднувати плейлисти, треки, артистів та альбоми з різних джерел у єдиному веб-інтерфейсі. Інструментарій включає інструменти для злиття плейлистів, їх розділення, а також інтелектуального аналізу структури медіатеки. Взаємодія зі стримінгами відбувається на рівні сервер-сервер через хмарну інфраструктуру Soundiiz.

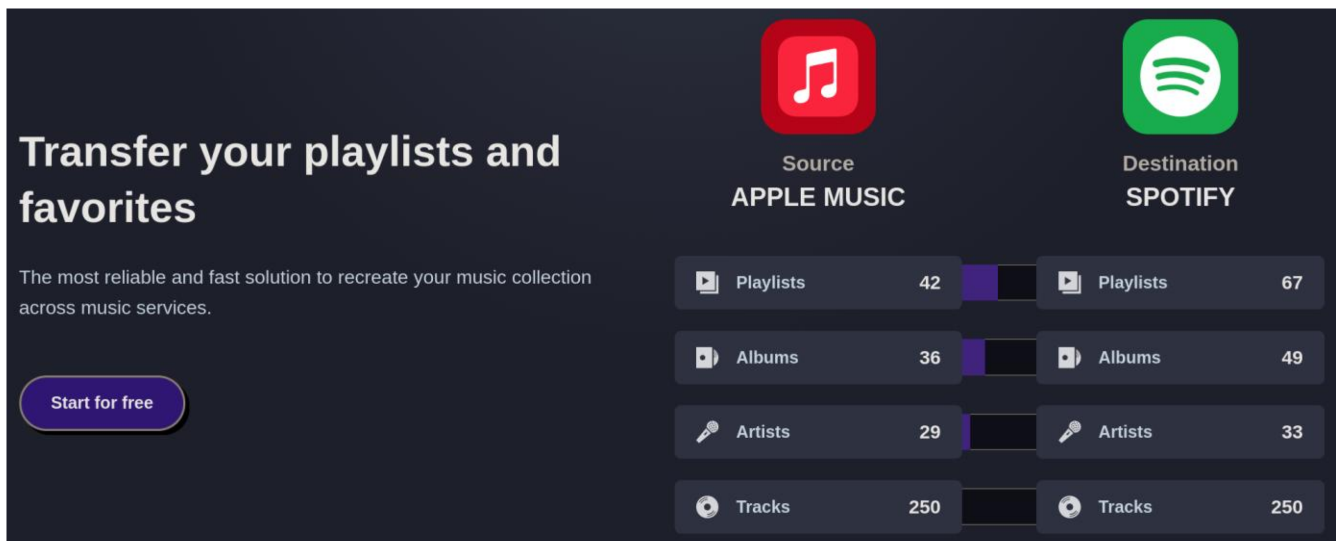

Рисунок 2.2 – **Soundiiz промо**

Переваги:

- Хмарна обробка даних. Усі операції з копіювання та синхронізації виконуються на серверах платформи, що звільняє ресурси клієнтського пристрою.
- Глибокий інструментарій корисних функцій. Наявність інструментів для пакетного редагування метаданих, видалення дублікатів та автоматичного злиття декількох списків відтворення в один.
- Стабільність з'єднання. Хмарна архітектура забезпечує постійний зв'язок з API музичних платформ з мінімальним індексом мережевих затримок.
- Недоліки:
- Платна модель підписки для автоматичних функцій. Більшість критично важливих функцій, таких як постійна автоматична синхронізація плейлистів у реальному часі, заблоковані за комерційною підпискою.

- Обмежені можливості стримінгу контенту. Веб-плеєр Soundiiz інтегрує лише короткі 30-секундні прев'ю-версії треків для більшості платформ через ліцензійні обмеження, що унеможливлює використання сервісу як повноцінного аудіоплеєра.
- Централізована монолітна база даних. Збереження конфіденційних токенів доступу (OAuth) користувачів відбувається у закритій централізованій базі даних, що підвищує ризики витоку інформації при комерційних кібератаках.

2.3 Висновки щодо аналізу аналогів

Проведений аналіз існуючих рішень демонструє чіткий технологічний розрив на ринку програмного забезпечення для агрегації медіаконтенту. Наявні системи чітко розділені на дві категорії: або це чисті «менеджери даних» без можливості аудіовідтворення (FreeYourMusic, Soundiiz), або це плеєри з поверхневою інтеграцією, які повністю обмежені рамками стандартних публічних API Жоден із розглянутих аналогів не забезпечує одночасного вирішення задачі наскрізного програвання потоків (AAC HLS та зашифрованого Blowfish FLAC) в межах єдиної черги відтворення [13].

Розробка власного унікального програмного забезпечення є технологічно необхідною з таких причин:

1. Потреба в низькорівневій обробці аудіопотоків на бекенді. Власне рішення на базі мови Rust дозволяє реалізувати кастомні модулі дешифрування Blowfish-CBC для сервісу Deezer та динамічний парсинг HLS-чанків для SoundCloud безпосередньо на стороні сервера. Це знімає обмеження щодо ліцензування віджетів та забезпечує передачу чистого аудіосигналу на клієнт.

2. Оптимізація продуктивності через мікросервісну архітектуру. Розподіл системи на ізольовані сервіси (auth, library, soundcloudОгляд функціональності та обмежень платформеного інтегратора, deezer) під керуванням Nginx гарантує горизонтальну масштабованість та стійкість до відмов, чого бракує монолітним або суто клієнтським аналогам.FreeYourMusic

3. Ефективне дворівневе збереження даних. Впровадження Redis для оперативної серіалізації черг відтворення та сесій користувачів у поєднанні з об'єктним S3-сховищем MinIO для кешування важких аудіосегментів мінімізує кількість запитів до зовнішніх API, захищаючи систему від блокувань (Rate Limiting) та гарантуючи миттєвий відгук інтерфейсу.

Таким чином, проектований мікросервісний агрегатор на стеку Rust/Axum + Angular є унікальним інженерним рішенням, що ліквідує функціональні обмеження комерційних аналогів та забезпечує створення високопродуктивного, незалежного медіасередовища.

3. ТЕОРЕТИЧНА ЧАСТИНА ТА ПРОЄКТУВАННЯ СИСТЕМИ

3.1. Проєкт архітектури розподіленої мікросервісної системи на базі Nginx

Проєктування сучасних високонавантажених платформ агрегації медіаконтенту вимагає відмови від монолітних архітектурних патернів. Розподіл системи на ізольовані функціональні блоки забезпечує високу відмовостійкість і спрощує горизонтальне масштабування. Центральним елементом маршрутизації в розробленій архітектурі виступить вебсервер Nginx. Він функціонує як зворотний проксі-сервер (Reverse Proxy) на сьомому рівні моделі OSI.

Nginx приймає вхідні HTTP-запити від клієнтського застосунку Angular і перенаправляє їх на відповідні бекенд-сервіси на основі аналізу URL-префіксів. Сервіси auth-service, library-service, soundcloud-service та deezer-service повністю автономні. Вони взаємодіють між собою та з базами даних в асинхронному безблокувальному режимі. Застосування Nginx дозволяє ефективно приховати внутрішню топологію мережі Docker-контейнерів від зовнішнього спостерігача. Це підвищує загальний рівень безпеки системи. Окрім маршрутизації, на проксі-сервер покладаються задачі балансування навантаження, термінації SSL-сертифікатів і стиснення контенту [10].

Кожен окремий мікросервіс реалізовано на базі асинхронного рантайму Tokio мови програмування Rust. Замість тривалого утримання потоків ОС під час очікування відповіді від мережі, сервіси використовують кооперативну багатозадачність. Це мінімізує споживання оперативної пам'яті. Зв'язок із персистентними сховищами (PostgreSQL, Redis, MinIO S3) здійснюється

через пули з'єднань, що нівелює накладні витрати на ініціалізацію TCP-сесій для кожного індивідуального клієнтського запиту.

3.2. Алгоритми обробки аудіопотоків та криптографічного дешифрування

3.2.1. Логіка парсингу та стримінгу AAC HLS потоків з SoundCloud

Аудіоконтент на платформі SoundCloud поширюється за допомогою протоколу потокової передачі HTTP Live Streaming (HLS). Цей стандарт передбачає розділення цілісного аудіофайлу на послідовність коротких фрагментів (медіа-сегментів), кодованих у форматі AAC. Взаємодія з такому потоком починається з надсилання запиту до API SoundCloud для отримання майстер-маніфесту у форматі файлу .m3u8.

Сервіс soundcloud-service виконує асинхронний парсинг отриманого маніфесту. Програма аналізує вміст файлу, витягує URL-адреси окремих чанків і формує чергу завантаження. Для забезпечення безперервного аудіовідтворення без затримок на стороні клієнта, отримані HLS-сегменти піддаються динамічній обробці. Оскільки сирі фрагменти можуть мати специфічні метадані контейнера Transport Stream (TS) або фрагментованого MP4, безпосередня передача їх у браузер часто викликає збої синхронізації аудіо-движка.

Для нівелювання цієї проблеми застосовано утиліту ffmpeg, інтегровану в pipeline обробки через асинхронні потоки (pipes) Rust. Бекенд зчитує вхідні мережеві чанки, передає їх у стандартний вхідний потік (stdin) процесу ffmpeg, де відбувається демультиплексування, вирівнювання часових міток (PTS/DTS) та ремультиплексування в уніфікований аудіопотік. Результат зчитується зі стандартного виходу (stdout) процесу і транслюється клієнту у

вигляді безперервного HTTP-стріму. Такий підхід гарантує стабільне відтворення незалежно від початкових параметрів кодування файлу на серверах SoundCloud.

3.2.2. Алгоритм дешифрування аудіоданих Deezer FLAC за допомогою Blowfish та синхронізація з S3

Робота з платформою Deezer відрізняється високою складністю через використання пропрієтарного криптографічного захисту аудіопотоків високої якості (FLAC). Сервіс не використовує стандартний HLS, а віддає статичні аудіофайли, які зашифровані блочним шифром Blowfish у режимі CBC (Cipher Block Chaining). Для реалізації безпосереднього програвання треків розроблено унікальний інженерний механізм дешифрування на льоту.

Першим етапом є генерація унікального 128-бітного криптографічного ключа. Метод generate_blowfish_key приймає ідентифікатор треку (track_id) та обчислює його MD5-хеш. Отриманий 32-символьний шістнадцятковий рядок перетворюється на масив байтів. Далі виконується побітова операція XOR між першою та дозволеною половинами хешу, а також статичною константою-секретом b"g4el58wc0zvf9na1". Схема генерації представлена наступним алгоритмом:

Rust

```
pub fn generate_blowfish_key(track_id: &str) -> [u8; 16] {
    const SECRET: &[u8] = b"g4el58wc0zvf9na1";
    let digest = md5::compute(track_id.as_bytes());
    let md5_id = format!("{:x}", digest);
    let md5_bytes = md5_id.as_bytes();
    let mut bf_key = [0u8; 16];
```

```
        for i in 0..16 {
            bf_key[i] = md5_bytes[i] ^ md5_bytes[i + 16] ^ SECRET[i];
        }
        bf_key
    }
```

Після генерації ключа сервіс виконує асинхронний запит до медіа-серверів Deezer через метод get_track_url та ініціює завантаження байтового масиву (bytes_stream). Особливість криптографічного захисту Deezer полягає в тому, що шифруванню піддається не весь файл, а лише кожен третій сегмент розміром 2048 байтів.

Для обробки такого потоку в пам’яті створюється стан-акумулятор (futures::stream::unfold). Мережеві чанки накопичуються у внутрішньому буфері Vec<u8>. Щойно розмір буфера досягає або перевищує 2048 байтів, з нього вилучається (drain) фіксований сегмент. Програма обчислює індекс сегмента за формулою segment_index = total_bytes_sent / 2048. Якщо segment_index % 3 == 0, сегмент розбивається на блоки по 8 байтів і дешифрується за допомогою об'єкта BlowfishCbcDec з використанням раніше згенерованого ключа та вектору ініціалізації (IV). Інші сегменти пропускаються без змін.

З метою оптимізації мережевого трафіку розроблено метод get_stream_and_save. Він використовує механізм багатоадресної трансляції tokio::sync::broadcast. Сервер створює канал, в який асинхронна задача (tokio::spawn) надсилає оброблені та дешифровані аудіосегменти. До цього каналу одночасно підключаються два споживачі: потік клієнта (Caller Stream), який обгортається в HTTP-відповідь Axum і негайно транслюється в браузер користувача, та завантажувач S3 (S3 Uploader).

Завантажувач S3 зчитує ті самі байтові дані, пакує їх в об'єкт SdkBody і надсилає в об'єктне сховище MinIO за допомогою методу put_object. Важливою архітектурною особливістю є те, що унікальний ідентифікатор файлу в сховищі S3 (s3_file_id) строго еквівалентний його первинному ідентифікатору в реляційній базі даних (db_id). Це усуває необхідність збереження додаткових таблиць маппінгу та прискорює пошук медіаресурсів. При обробці великих аудіофайлів (наприклад, тривалих треків або Lossless FLAC-записів) система задіює механізм потокового завантаження частинами (Multipart Upload) [5]. Файл розбивається на окремі мережеві чанки та завантажується кусками без повного виділення оперативної пам’яті під увесь об'єкт, що запобігає виникненню помилок переповнення пам'яті (Out-Of-Memory).

Завдяки такій схемі повторне прослуховування цього ж треку повністю виключає звернення до серверів Deezer та обчислювальні витрати на дешифрування. Файл миттєво віддається з локального S3-сховища. Ротація та оновлення застарілих сесійних токенів автоматизовані у методі update_token, захищеному асинхронним м'ютексом (token_update_lock), що унеможливлює виникнення стану гонитви (race condition) при одночасних запитах від багатьох потоків.

3.3. Проєктування механізмів розподіленого кешування у Redis

Оптимізація швидкодії мікросервісів та зниження навантаження на реляційну базу даних PostgreSQL досягається за допомогою проектування структурованого шару кешування в оперативній пам'яті на базі СКБД Redis. Для кожної сутності в системі жорстко регламентовано життєвий цикл (Time-To-Live, TTL) та стратегію валідації. Налаштовано такі константи часових обмежень:

- USER_CACHE_TTL: 7200 (2 години) — тривалість зберігання серіалізованих профілів користувачів та їх плейлистів.
- SESSION_CACHE_TTL: 3600 (1 година) — час життя активної сесії безпеки.
- ACCESS_TOKEN_TTL: 3600 (1 година) — період валідності токена доступу.
- REFRESH_TOKEN_TTL: 2592000 (30 діб) — тривалість сесії оновлення.
- VERIFY_TTL: 1800 (30 хвилин) — ліміт дії коду підтвердження реєстрації.
- VERIFY_ATTEMPTS: 3 — максимальна кількість спроб введення коду до блокування.

Механізм автентифікації базується на концепції сесійних серійних номерів (sn), які інтегруються безпосередньо у тіло JWT-токенів доступу та зберігаються у зв'язці з session_id. Метод create_session створює у Redis ключ виду session:{session_id}, записуючи туди значення sn із встановленим значенням SESSION_CACHE_TTL. При кожному запиті, що вимагає авторизації, викликається функція proof_session:

```
Rust
pub async fn proof_session(
    &self,
    session_id: Uuid,
    sn: Uuid,
) -> RedisResult<Result<(), SessionError>> {
    let mut conn = self.client.get_multiplexed_async_connection().await?;
    let sn_r: Option<String> = conn.get(format!("session:{}", session_id)).await?;

    match sn_r {
        Some(sn_r) => {
            if sn.to_string() != sn_r {
                return Ok(Err(SessionError::SessionWasUpdated));
```

```
            }
            Ok(Ok(()))
        }
        None => Ok(Err(SessionError::SessionNotFound)),
    }
}
```

Використання унікального серійного номера sn та його верифікація через оперативну пам'ять Redis виконує критично важливу оптимізаційну функцію: це дозволяє системі миттєво валідувати або скидати сесії користувачів без необхідності постійного оновлення індексів та виконання операцій запису у реляційній базі даних (PostgreSQL). Оскільки оновлення індексів на великих обсягах даних у таблицях сесій є вкрай ресурсомісткою операцією, яка викликає блокування та знижує загальну пропускну здатність системи, перенесення перевірки sn у Redis повністю вирішує проблему деградації продуктивності бази даних.

3.4. Реляційне моделювання структури бази даних у PostgreSQL

Для довготривалого збереження персистентних даних, забезпечення цілісності посилань та накопичення статистики прослуховувань розроблено нормалізовану схему реляційної бази даних PostgreSQL. На системному рівні ініціалізується розширення "pgcrypto", що дозволяє генерувати захищені псевдовипадкові ідентифікатори UUID на стороні СКБД.

SQL

```sql
CREATE EXTENSION IF NOT EXISTS "pgcrypto";
```

Архітектура таблиць розділена на три логічні блоки: ідентифікація користувачів, управління користувацькими кросплатформними списками відтворення та локальне дзеркалювання метаданих цільових стримінгових сервісів. Ядром системи автентифікації є таблиці app_users та app_sessions:

SQL

```
CREATE TABLE app_users (
   id            BIGSERIAL      PRIMARY KEY,
   email          VARCHAR(50)     NOT NULL    UNIQUE,
   username       VARCHAR(30)     NOT NULL    UNIQUE,
   password_hash   VARCHAR(255)    NOT NULL
);

CREATE TABLE app_sessions (
   id         UUID          PRIMARY KEY DEFAULT gen_random_uuid(),
   user_id     BIGINT           NOT NULL REFERENCES app_users(id) ON
DELETE CASCADE,
   sn         UUID          NOT NULL    UNIQUE,
   expires_at  TIMESTAMPTZ     NOT NULL,
   created_at  TIMESTAMPTZ     NOT NULL DEFAULT NOW()
);
```

Каскадне видалення (ON DELETE CASCADE) гарантує автоматичне очищення активних сесій та пов’язаних об’єктів при анулюванні облікового запису. Для реалізації ключової фічі агрегатора — об'єднання треків з різних джерел в межах одного списку відтворення — впроваджено перелічуваний тип track_platform та поліморфну структуру зв'язків через таблиці user_playlist та track_in_playlist:

```
SQL
CREATE TYPE track_platform AS ENUM ('deezer', 'soundcloud');

CREATE TABLE user_playlist (
   id              BIGSERIAL      PRIMARY KEY,
   title            VARCHAR(25)     NOT NULL,
   owner_id                 BIGINT              NOT  NULL      REFERENCES
app_users(id) ON DELETE CASCADE,
   last_track_added_at TIMESTAMPTZ     NULL
);

CREATE TABLE track_in_playlist (
   id            BIGSERIAL      PRIMARY KEY,
   playlist_id          BIGINT                  NOT  NULL        REFERENCES
user_playlist(id) ON DELETE CASCADE,
   track_id       BIGINT         NOT NULL,
```

```
  platform_id    track_platform  NOT NULL,
  position       INT             NOT NULL
);
```

Збереження структури метаданих для платформи Deezer враховує сувору ієрархію «Виконавець $\rightarrow$ Альбом $\rightarrow$ Трек». Кожен елемент містить посилання на зовнішній унікальний числовий ідентифікатор API Deezer. Окремо реалізовано таблицю listenings_deezer із розрахунком індексів для швидкої побудови аналітики:

```
SQL
CREATE TABLE authors_deezer (
 id       BIGINT  PRIMARY KEY,
 title    TEXT    NOT NULL,
 img      TEXT
);

CREATE TABLE albums_deezer (
 id       BIGINT  PRIMARY KEY,
 title    TEXT    NOT NULL,
 img      TEXT,
 author_id BIGINT  NOT NULL REFERENCES authors_deezer(id) ON
DELETE CASCADE
);

CREATE TABLE tracks_deezer (
 id       BIGINT  PRIMARY KEY,
 title    TEXT    NOT NULL,
 duration INT     NOT NULL,
 img      TEXT,
 album_id  BIGINT  NOT NULL REFERENCES albums_deezer(id) ON
DELETE CASCADE
);

CREATE TABLE listenings_deezer (
 id        BIGSERIAL PRIMARY KEY,
 track_id      BIGINT NOT NULL REFERENCES tracks_deezer(id) ON
DELETE CASCADE,
 listened_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
```

```
);
CREATE INDEX ON listenings_deezer (track_id);
```

Специфіка архітектури SoundCloud не має жорсткої прив'язки треків до альбомів. Вона фокусується на прямому зв'язку автора з контентом та публічних кастомних плейлистах. База даних відображає цю особливість через таблиці authors_soundcloud, tracks_soundcloud та playlist_soundcloud. Первинний ключ таблиці playlist_tracks_soundcloud є складеним (playlist_id, position), що дозволяє дублювати один і той самий трек на різних позиціях в межах одного списку відтворення:

```
SQL
CREATE TABLE authors_soundcloud (
  id       BIGINT  PRIMARY KEY,
  title    TEXT    NOT NULL,
  img      TEXT
);

CREATE TABLE tracks_soundcloud (
  id       BIGINT  PRIMARY KEY,
  title    TEXT    NOT NULL,
  duration  BIGINT  NOT NULL,
  img      TEXT,
  author_id  BIGINT  NOT  NULL  REFERENCES  authors_soundcloud(id)
ON DELETE CASCADE
);

CREATE TABLE playlist_soundcloud (
  id             BIGINT  PRIMARY KEY,
  title          TEXT    NOT NULL,
  img            TEXT,
  playlist_size   INT     NOT NULL,
  author_id                  BIGINT      NOT    NULL    REFERENCES
authors_soundcloud(id) ON DELETE CASCADE
);

CREATE TABLE playlist_tracks_soundcloud (
```

```
  playlist_id BIGINT NOT NULL REFERENCES playlist_soundcloud(id) ON DELETE CASCADE,
  track_id    BIGINT NOT NULL REFERENCES tracks_soundcloud(id) ON DELETE CASCADE,
  position    INT NOT NULL,
  PRIMARY KEY (playlist_id, position)
);
CREATE INDEX ON playlist_tracks_soundcloud (track_id);

CREATE TABLE listenings_soundcloud (
  id          BIGSERIAL PRIMARY KEY,
  track_id    BIGINT NOT NULL REFERENCES tracks_soundcloud(id) ON DELETE CASCADE,
  listened_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
CREATE INDEX ON listenings_soundcloud (track_id);
```

Для оптимізації масового імпорту даних з API Deezer на рівні СКБД створено спеціалізовані композитні типи даних: author_input_deezer, track_input_deezer та album_input_deezer. Вони використовуються в збережених процедурах для пакетної інсерції (Bulk Insert) через драйвер SQLx.

3.5. Уніфікація гетерогенних медіаданих та патерн проєктування на базі Trait Into

Головним технологічним викликом при розробці агрегатора є різнорідність моделей даних, що повертаються зовнішніми API. SoundCloud оперує текстовими та числовими ідентифікаторами змішаного типу, повертає об'єкти з варіативними структурами вкладеності. Deezer використовує виключно цілі числівники для ID та строго регламентує набір полів.

Для нівелювання цієї гетерогенності та забезпечення стабільної десеріалізації на фронтенді розроблено уніфікований шар абстракції моделей.

Усі зовнішні дані від різних майданчиків віддаються клієнту в єдиному уніфікованому вигляді. Ідентифікатори трансформуються в універсальний тип String. Належність контенту до конкретного провайдера визначається перелічуваним типом ApiServices:

```
Rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, Deserialize, Serialize)]
#[serde(rename_all = "lowercase")]
pub enum ApiServices {
   Deezer,
   Soundcloud,
}
```

Базовими елементами уніфікованого інтерфейсу є структури ApiArtist, ApiUser та ApiTrackInAlbum. Вони містять лише необхідний для рендерингу набір полів та маркер походження контенту:

```
Rust
#[derive(Serialize, Debug)]
pub struct ApiArtist {
   pub id: String,
   pub username: String,
   pub picture: Option<String>,
   pub is_dummy: bool,
}

#[derive(Serialize)]
pub struct ApiUser {
   pub id: String,
   pub img: String,
   pub username: String,
}

#[derive(Serialize)]
pub struct ApiTrackInAlbum {
```

```
    pub id: String,
    pub title: String,
    pub duration: i32,
    pub track_url: Option<String>,
    pub track_token: Option<String>,
}
```

Композитні сутності високого рівня (ApiAlbum, ApiTrack, ApiPlaylist) агрегують базові структури у вектори (Vec<T>). Це дає можливість будувати складні деревоподібні списки відтворення, абстрагуючись від фізичного джерела походження медіафайлу:

```
Rust
#[derive(Serialize)]
pub struct ApiAlbum {
    pub id: String,
    pub img: Option<String>,
    pub title: String,
    pub service: ApiServices,
    pub artists: Vec<ApiArtist>,
    pub tracks: Vec<ApiTrackInAlbum>,
}

#[derive(Serialize)]
pub struct ApiTrack {
    pub id: String,
    pub service: ApiServices,
    pub title: String,
    pub artists: Vec<ApiArtist>,
    pub alb_id: Option<String>,
    pub alb_title: Option<String>,
    pub duration: i64,
    pub platform: ApiServices,
    pub picture: Option<String>,
    pub track_url: Option<String>,
    pub track_token: Option<String>,
}

#[derive(Serialize)]
pub struct ApiPlaylist {
```

```
    pub id: String,
    pub title: String,
    pub parent_user_id: String,
    pub parent_username: String,
    pub parent_picture: Option<String>,
    pub platform: ApiServices,
    pub picture: Option<String>,
    pub tracks: Vec<ApiTrack>,
    pub size: u32,
}
```

Результати пошукових запитів консолідуються мікросервісами в єдину агреговану сторінку видачі ApiSearchPage. Вона передається на клієнтський UI Angular через єдиний уніфікований JSON-пакет:

```
Rust
#[derive(Serialize)]
pub struct ApiSearchPage {
    pub artists: Vec<ApiArtist>,
    pub albums: Vec<ApiAlbum>,
    pub tracks: Vec<ApiTrack>,
    pub playlists: Vec<ApiPlaylist>,
    pub users: Vec<ApiUser>,
}
```

Трансформація сирих структур відповідей зовнішніх провайдерів в уніфіковані моделі реалізована за допомогою стандартного та найбільш ефективного механізму мови Rust — патерну проєктування на базі трейту Into.

Замість написання об'ємних імперативних функцій маппінгу всередині обробників маршрутів Axum, архітектура використовує декларативний метод .into(). Це забезпечує винесення логіки конвертації на етап компіляції

програми, гарантує строгу типізацію, унеможливлює помилки часу виконання (Runtime Errors) та створює чисту абстракцію над гетерогенним медіа-середовищем. Надійна робота макросів #[derive(Serialize)] бібліотеки serde гарантує миттєве перетворення цих структур у JSON-потоки безпосередньо в момент відправки мережевої відповіді користувачу.

4. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ІНТЕРФЕЙС КОРИСТУВАЧА

4.1. Обґрунтування вибору технологічного стека

4.1.1. Вибір мови Rust, фреймворку Axum та асинхронного драйвера SQLx для бекенду

Проєктування серверної частини архітектури агрегатора потокового аудіо вимагає забезпечення мінімального часу затримки (latency) при обробці гетерогенних медіаданих та високої пропускної здатності в умовах інтенсивного введення-виведення (I/Obound навантаження). Для вирішення цих завдань було обрано мову програмування системного рівня Rust. Головною перевагою Rust є унікальна модель керування пам’яттю на основі концепцій володіння (ownership) та запозичення (borrowing), що перевіряються на етапі компіляції за допомогою механізму Borrow Checker. Це повністю виключає необхідність використання збирача сміття (Garbage Collector), усуваючи непередбачувані затримки у виконанні потоків дешифрування та ремультиплексування аудіочанків, а також гарантує повну безпеку роботи з пам'яттю (memory safety) та відсутність станів гонитви (race conditions) при багатопотоковому обході каналів мовлення.

Як базовий вебфреймворк інтегровано Axum, що розвивається в межах екосистеми асинхронного рантайму Tokio. Axum використовує декларативну модель маршрутизації, побудовану на базі потужної системи макросів та типах-екстракторах (Extractors). Це дозволяє безпечно витягувати параметри HTTP-запитів, JSON-корисне навантаження та сесійні дані безпосередньо у сигнатурах асинхронних функцій-обробників. Завдяки тісній інтеграції з архітектурою сервісних шарів Tower, Axum забезпечує модульне

підключення проміжного програмного забезпечення (Middleware) для логування, авторизації та лімітування частоти запитів (Rate Limiting).

Взаємодія з реляційною базою даних PostgreSQL реалізована за допомогою інструментарію SQLx. На відміну від класичних важковагових ORM-систем, які створюють додаткові обчислювальні накладні витрати та генерують неоптимальний SQL-код, SQLx є повністю асинхронним безблокувальним драйвером, що працює на базі пулу з'єднань. Ключовою інженерною перевагою SQLx є механізм компіляційної перевірки запитів (Compile-time checked queries). Під час збирання проєкту макроси sqlx::query! зв'язуються з екземпляром бази даних та валідують синтаксис SQL-коду і відповідність типів даних структури Rust, що повністю запобігає виникненню синтаксичних помилок чи невідповідності схем у промисловому середовищі (Runtime).

4.1.2. Переваги клієнтського фреймворку Angular та контейнеризації через Docker

Для побудови динамічного, чуйного та стійкого до навантажень інтерфейсу користувача (UI) застосовано фреймворк Angular. Організація клієнтської частини за архітектурним шаблоном Single Page Application (SPA) мінімізує обсяг мережевого трафіку за рахунок одноразового завантаження базового пакету ресурсів та подальшого асинхронного оновлення лише необхідних фрагментів DOM-дерева через REST API. Сувора типізація за допомогою TypeScript знижує ймовірність логічних помилок при обробці уніфікованих медіа-структур.

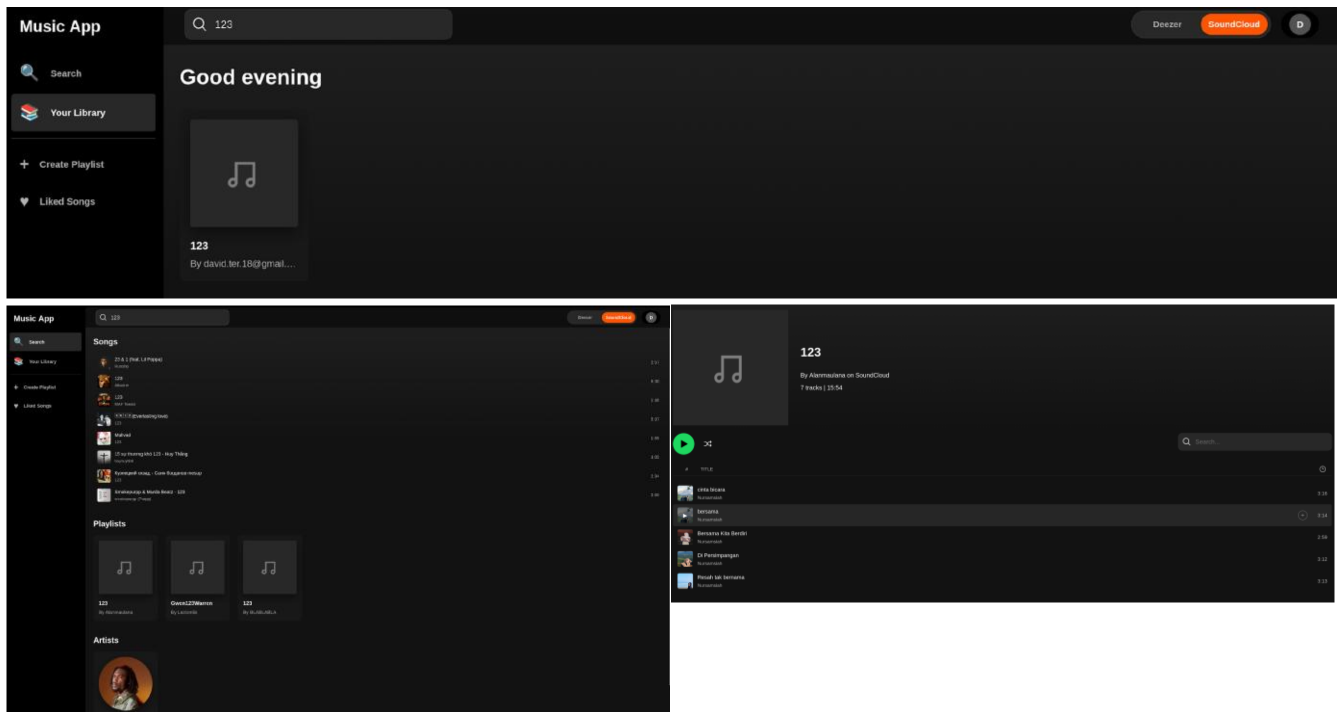

Рисунок 4.1.2 – Інтерфейс музикальной платформи

Компонентно-орієнтований підхід Angular спрощує масштабування інтерфейсу, дозволяючи ізолювати логіку аудіоплеєра, пошукової видачі та панелі керування списками відтворення. Вбудований механізм ін'єкції залежностей (Dependency Injection) забезпечує ефективне керування станом застосунку (State Management), а реактивні потоки RxJS дозволяють обробляти події відтворення, буферизації та зміни позиції треку в реальному часі у безблокувальному режимі за допомогою патерну Observer.

Ізоляція, конфігурування та розгортання всього ландшафту мікросервісів здійснюється засобами платформи контейнеризації Docker. Кожен функціональний модуль системи пакується в ізольований образ за технологією багатоетапного збирання (Multi-stage builds). Це дозволяє виконувати компіляцію важкого Rust-коду в проміжному образі із встановленими інструментами розробки, а у фінальний мінімальний образ (на базі Alpine Linux або Distroless) копіювати виключно скомпільований

бінарний файл та статичні конфігурації. Такий підхід скорочує розмір фінального контейнера до декількох десятків мегабайтів і мінімізує вектор потенційних атак на безпеку операційного середовища.

4.2. Опис програмної реалізації мікросервісів та бібліотек

4.2.1. Реалізація сервісів авторизації (auth-service) та керування медіатекою (library-service)

Мікросервіс auth-service є центральним компонентом системи безпеки. Він відповідає за реєстрацію, автентифікацію користувачів та верифікацію їх повноважень. Під час реєстрації сервіс приймає пароль у відкритому вигляді, генерує криптографічну сіль та виконує односпрямоване хешування за допомогою алгоритму Argon2id (через криптографічне розширення "pgcrypto" у базі даних або локальні бібліотеки), записуючи результат у таблицю app_users. Процес авторизації супроводжується генерацією пари токенів: короткоживучого Access Token та довгоживучого Refresh Token.

При кожному запиті клієнта сервіс генерує унікальний сесійний серійний номер sn (тип Uuid), який записується як у тіло токена, так і в розподілену пам'ять Redis із прив'язкою до ідентифікатора сесії. Уся робота з базовими структурами даних користувачів, конфігураціями авторизаційних запитів та помилками спирається на спільну системну бібліотеку **domain-lib**, яка виступає єдиним джерелом правди (Single Source of Truth) для всіх бізнес-сутностей та моделей предметної області в межах усього мікросервісного ландшафту.

Мікросервіс library-service керує персистентним станом користувацьких медіатек. Він реалізує CRUD-інтерфейс для взаємодії з таблицями user_playlist та track_in_playlist. Сервіс забезпечує атомарність операцій при

зміні порядку треків усередині списку відтворення шляхом виконання транзакційних SQL-запитів. Оновлення поля last_track_added_at здійснюється за допомогою тригерів або безпосередньо в тілі асинхронних транзакцій через SQLx пул з'єднань, гарантуючи коректне відображення 4.2.2 метаданих.

4.2.2. Розробка модулів інтеграції стримінгів (soundcloud-service, deezer-service) та спільних бібліотек (cache-lib, utils-lib, email-lib)

Спеціалізовані мікросервіси soundcloud-service та deezer-service інкапсулюють логіку взаємодії із відповідними зовнішніми API. Модуль soundcloud-service містить асинхронні HTTP-клієнти (на базі бібліотеки reqwest), які виконують завантаження HLS-маніфестів .m3u8, розбирають їх на індивідуальні посилання AAC-сегментів та здійснюють ремультиплексування через асинхронні канали утиліти ffmpeg.

Сервіс deezer-service реалізує криптографічний алгоритм дешифрування блоків Blowfish-CBC на льоту, використовуючи обчислені MD5-ключі для кожного третього 2048-байтового сегмента. Тут же розгорнуто механізм багатоадресної трансляції tokio::sync::broadcast, що дозволяє одночасно стрімити аудіодані користувачу і завантажувати їх кусками за схемою Multipart Upload в об'єктне сховеще MinIO S3, де унікальний s3_file_id жорстко прирівняний до первинного ключа треку в базі даних (db_id), запобігаючи дублюванню операцій дешифрування при повторних запитах.

Для винесення спільного функціоналу та запобігання дублюванню коду розроблено набір локальних бібліотек:

- domain-lib — фундаментальна бібліотека, яка зберігає в собі всі бізнес-сутності, переліки сервісів (ApiServices), структури відповідей (ApiTrack, ApiAlbum, ApiPlaylist, ApiSearchPage), спільні

типи помилок та валідаційні схеми. Вона підключається як залежність до кожного мікросервісу, забезпечуючи уніфікацію обміну даними.

- cache-lib — інкапсулює логіку взаємодії з Redis. Реалізує абстракції для запису, зчитування та атомарного оновлення кешу із заданими параметрами TTL, а також метод proof_session для швидкої перевірки сесійного номера sn без звернення до дискової підсистеми PostgreSQL.

- utils-lib — містить допоміжні утиліти для форматування рядків, обробки часових міток, налаштування CORS-політик та конфігурування глобального логування/трасування (через tracing-subscriber).

- email-lib — відповідає за асинхронне формування та відправку сервісних повідомлень (наприклад, кодів підтвердження реєстрації) по протоколу SMTP, взаємодіючи з чергами завдань.

4.3. Конфігурація розгортання та маршрутизації в Docker-контейнерах через Nginx

Оркестрація та координація життєвого циклу контейнерів реалізована за допомогою інструменту docker-compose. Опис інфраструктури містить декларативне визначення мереж (Networks) та томів даних (Volumes) для персистентного зберігання файлів PostgreSQL, Redis та MinIO. Всі мікросервіси об'єднані в ізольовану внутрішню мережу backend-net, доступ до якої ззовні повністю закритий, за винятком контейнера з вебсервером Nginx, який виступає єдиною точкою входу (Ingress Gateway).

Конфігураційний файл nginx.conf спроєктовано для оптимізації маршрутизації на сьомому рівні моделі OSI. Запити, що надходять від клієнтського застосунку Angular на порту 80 (або 443 при SSL), аналізуються

за контекстним шляхом (Location blocks). Нижче наведено фрагмент конфігурації маршрутизації Nginx:

```
Nginx
events {
   worker_connections 1024;
}

http {
   # Provide Docker's default DNS server
   resolver 127.0.0.11 valid=10s;

   server {
      listen 80;

      # Frontend Application
      location / {
         set $frontend "http://frontend:80";
         proxy_pass $frontend;
         proxy_http_version 1.1;
         proxy_set_header Upgrade $http_upgrade;
         proxy_set_header Connection "upgrade";
         proxy_set_header Host $host;
         proxy_set_header X-Real-IP $remote_addr;
         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
         proxy_set_header X-Forwarded-Proto $scheme;
      }

      # Catch-all for undefined API routes to prevent falling back to the frontend HTML
      location /api/ {
         return 404;
      }

      # Auth Service
      location /api/auth/ {
         # CORS Preflight
         if ($request_method = 'OPTIONS') {
            add_header 'Access-Control-Allow-Origin' $http_origin;
            add_header 'Access-Control-Allow-Credentials' 'true';
```

```
                add_header 'Access-Control-Allow-Methods' 'GET, POST,
OPTIONS, PUT, DELETE';
                add_header 'Access-Control-Allow-Headers' 'DNT,User-Agent,X-
Requested-With,If-Modified-Since,Cache-Control,Content-
Type,Range,Authorization';
                add_header 'Access-Control-Max-Age' 1728000;
                add_header 'Content-Type' 'text/plain; charset=utf-8';
                add_header 'Content-Length' 0;
                return 204;
            }

            # CORS Headers for actual requests
            add_header 'Access-Control-Allow-Origin' $http_origin always;
            add_header 'Access-Control-Allow-Credentials' 'true' always;

            set $auth "http://auth:3000";
            proxy_pass $auth;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # SoundCloud Service
        location /api/soundcloud/ {
            # CORS Preflight
            if ($request_method = 'OPTIONS') {
                add_header 'Access-Control-Allow-Origin' $http_origin;
                add_header 'Access-Control-Allow-Credentials' 'true';
                add_header 'Access-Control-Allow-Methods' 'GET, POST,
OPTIONS, PUT, DELETE';
                add_header 'Access-Control-Allow-Headers' 'DNT,User-Agent,X-
Requested-With,If-Modified-Since,Cache-Control,Content-
Type,Range,Authorization';
                add_header 'Access-Control-Max-Age' 1728000;
                add_header 'Content-Type' 'text/plain; charset=utf-8';
                add_header 'Content-Length' 0;
                return 204;
            }

            # CORS Headers for actual requests
```

```
            add_header 'Access-Control-Allow-Origin' $http_origin always;
            add_header 'Access-Control-Allow-Credentials' 'true' always;

            set $soundcloud "http://soundcloud:3000";
            proxy_pass $soundcloud;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # Library Service
        location /api/library/ {
            # CORS Preflight
            if ($request_method = 'OPTIONS') {
                add_header 'Access-Control-Allow-Origin' $http_origin;
                add_header 'Access-Control-Allow-Credentials' 'true';
                add_header 'Access-Control-Allow-Methods' 'GET, POST, 
OPTIONS, PUT, DELETE';
                add_header 'Access-Control-Allow-Headers' 'DNT,User-Agent,X-
Requested-With,If-Modified-Since,Cache-Control,Content-
Type,Range,Authorization';
                add_header 'Access-Control-Max-Age' 1728000;
                add_header 'Content-Type' 'text/plain; charset=utf-8';
                add_header 'Content-Length' 0;
                return 204;
            }

            # CORS Headers for actual requests
            add_header 'Access-Control-Allow-Origin' $http_origin always;
            add_header 'Access-Control-Allow-Credentials' 'true' always;

            set $library "http://library:3000";
            proxy_pass $library;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
}
```

4.4. Опис інтерфейсу користувача та результатів тестування застосунку

Інтерфейс користувача розроблено у відповідності до сучасних вимог ергономіки та адаптивного дизайну. Основне вікно застосунку розділене на три функціональні зони: ліва навігаційна панель (навігація по розділах медіатеки, доступ до створених плейлистів), центральна робоча область (динамічно відображає результати пошуку, профілі виконавців, списки треків або сторінки альбомів) та нижня панель медіаплеєра. Панель плеєра є персистентною і містить стандартні органи керування (Play/Pause, Скіп треків, Шкала прогресу відтворення, Регулятор гучності), що реалізовано через взаємодію сервісів Angular з native HTML5 Audio API елементом.

Уніфікація моделей через бібліотеку domain-lib дозволяє відображати треки з SoundCloud та Deezer в одному загальному списку. Кожен трек маркується невеликою іконкою відповідного сервісу, вказуючи на походження контенту. Пошукова панель реалізує технологію "живого пошуку" за допомогою оператора RxJS debounceTime(300), що запобігає надсиланню спам-запитів на бекенд під час введення символів користувачем.

Тестування розробленого програмного забезпечення здійснювалося на кількох рівнях. Юніт-тестування (Unit Testing) покриває критично важливу бізнес-логіку: перевірку алгоритму генерації криптографічних ключів Blowfish у deezer-service та коректність роботи патерну Into при перетворенні моделей у domain-lib. Тести запускалися за допомогою стандартної утиліти cargo test.

Інтеграційне тестування API (Integration Testing) проводилося за допомогою інструменту Postman та автоматизованих скриптів. Воно

включало перевірку сценаріїв наскрізної автентифікації, створення кросплатформних плейлистів та стабільності утримання HTTP-з'єднань при довготривалому стрімінгу аудіоданих у безблокувальному режимі. Результати тестування підтвердили високу стійкість системи, відсутність витоків пам'яті (Memory leaks) та коректність синхронізації розподіленого кешу Redis із персистентною базою даних PostgreSQL при валідації сесійних серійних номерів sn.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальне науково-практичне завдання з проєктування, програмної реалізації та розгортання високопродуктивної кросплатформної мікросервісної системи для агрегації та потокової передачі медіаконтенту із залученням сучасних технологій системного програмування, розподіленого кешування, реляційного та об’єктного зберігання даних.

Під час виконання роботи було отримано такі основні результати:

1. Проаналізовано предметну область та виявлено ключові архітектурні обмеження наявних рішень у сфері стрімінгу аудіоданих, зокрема проблеми високої затримки (latency) при обробці гетерогенних потоків та складнощі кросплатформної інтеграції. Обґрунтовано перехід до децентралізованої мікросервісної топології під керуванням зворотного проксі-сервера Nginx, що дозволило ізолювати бізнес-логіку авторизації, керування користувацькою медіатекою та модулів безпосередньої взаємодії із зовнішніми API (SoundCloud, Deezer).

2. Обґрунтовано вибір технологічного стека, де базовим інструментом розробки серверної частини виступила мова програмування Rust та асинхронний вебфреймворк Axum у поєднанні з рантаймом Tokio. Даний вибір дозволив реалізувати кооперативну багатозадачність, забезпечити нульову вартість абстракцій (zero-cost abstractions) та гарантувати безпеку роботи з пам'яттю без накладних витрат на збирання сміття, що критично важливо для стабільного дешифрування і стримінгу аудіочанків.

3. Розроблено унікальні алгоритми обробки аудіопотоків, що включають асинхронний парсинг AAC HLS маніфестів платформи SoundCloud із ремультиплексуванням «на льоту» через процеси ffmpeg, а також потокове криптографічне дешифрування блочного шифру Blowfish у режимі CBC для кодованих файлів високої якості (FLAC) сервісу Deezer. Спроєктовано систему багатоадресної трансляції, яка одночасно стрімить аудіо клієнту та завантажує його кусками (Multipart Upload) в об'єктне сховище MinIO S3, де унікальний ідентифікатор файлу строго еквівалентний його ID у персистентній базі даних PostgreSQL.

4. Оптимізовано підсистему збереження даних за рахунок проєктування дворівневого шару архітектури. Перший рівень базується на нормалізованій схемі СКБД PostgreSQL, що забезпечує цілісність посилань та накопичення аналітики прослуховувань. Другий рівень реалізовано як розподілений шар кешування на базі СКБД Redis. Впровадження механізму унікальних сесійних серійних номерів sn та їх валідація через пам'ять Redis дозволили миттєво відкликати або підтверджувати повноваження користувача без виконання ресурсомістких операцій оновлення індексів у PostgreSQL, усуваючи загрозу деградації продуктивності бази даних.

5. Розроблено шар уніфікації гетерогенних даних на базі спільної системної бібліотеки domain-lib. Застосування патерну проєктування на основі стандартного трейту мови Rust — Into дало змогу переносити різнорідні структури відповідей зовнішніх провайдерів в уніфікований вигляд клієнтського представлення ще на етапі компіляції. Це мінімізувало помилки часу виконання, забезпечило повну абстракцію над медіа-середовищем та спростило побудову динамічного інтерфейсу на базі Angular.

6. Проведено комплексне тестування розробленого програмного забезпечення на юніт-рівні (покриття криптографічних модулів та

трансформацій моделей) та інтеграційному рівні (валідація REST API ендпоінтів, стійкість утримання HTTP-з'єднань при тривалому мовленні). Результати тестування підтвердили високу надійність, низьке споживання системних ресурсів та повну відповідність системи сформованим функціональним вимогам.

Розроблена система демонструє високий потенціал для горизонтального масштабування і може бути використана як технологічна основа для створення комерційних агрегаторів мультимедійного контенту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бланді Д. Програмування мовою Rust. Офіційне видання / Д. Бланді, Д. Оренді. – К. : Форс Україна, 2022. – 544 с.

2. Документація асинхронного рантайму Tokio для мови Rust [Електронний ресурс]. – Режим доступу: https://tokio.rs/tokio/tutorial

3. Офіційна документація веб-фреймворку Axum [Електронний ресурс]. – Режим доступу: https://docs.rs/axum/latest/axum/

4. Асинхронна взаємодія з PostgreSQL за допомогою SQLx [Електронний ресурс]. – Режим доступу: https://github.com/launchbadge/sqlx

5. Документація розширення pgcrypto для СКБД PostgreSQL [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/current/pgcrypto.html

6. Розподілене кешування в оперативній пам'яті за допомогою СКБД Redis [Електронний ресурс]. – Режим доступу: https://redis.io/docs/latest/

7. Парсинг та обробка HTTP Live Streaming (HLS) маніфестів [Електронний ресурс]. – Режим доступу: https://datatracker.ietf.org/doc/html/rfc8216

8. Шифрування алгоритмом Blowfish: специфікація та криптоаналіз [Електронний ресурс]. – Режим доступу: https://www.schneier.com/academic/blowfish/

9. Офіційне керівництво з розробки клієнтських застосунків на фреймворку Angular [Електронний ресурс]. – Режим доступу: https://angular.dev/

10. Контейнеризація мікросервісних систем за допомогою Docker та Docker Compose [Електронний ресурс]. – Режим доступу: https://docs.docker.com/

11. Архітектура зворотного проксі-сервера Nginx та оптимізація стримінгу даних [Електронний ресурс]. – Режим доступу: https://nginx.org/en/docs/

12. Серіалізація та десеріалізація структур даних у Rust за допомогою Serde [Електронний ресурс]. – Режим доступу: https://serde.rs/

13. Інтеграційне тестування розподілених REST API застосунків [Електронний ресурс]. – Режим доступу: https://learning.postman.com/docs/writing29-scripts/intro-to-scripts/

14. Специфікація об'єктного сховища MinIO S3 та організація Multipart Upload [Електронний ресурс]. – Режим доступу: https://min.io/docs/minio/linux/index.html

15. Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання: ДСТУ 7.1-2006. – [Чинний від 2007-07-01]. – К. : Держспоживстандарт України, 2007. – 47 с.

ДОДАТОК А

Код програмного продукту:

```
import { Routes } from '@angular/router';
import { LoginComponent } from './pages/login/login.component';
import { RegisterComponent } from './pages/register/register.component';
import { VerifyComponent } from './pages/verify/verify.component';
import { MainLayoutComponent } from './components/layout/main-layout/main-layout.component';
import { LibraryComponent } from './pages/library/library.component';
import { SearchComponent } from './pages/search/search.component';

export const routes: Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'verify', component: VerifyComponent },
  { path: 'verify/:code', component: VerifyComponent },
  {
    path: '',
    component: MainLayoutComponent,
    children: [
      { path: '', redirectTo: 'library', pathMatch: 'full' },
      { path: 'library', component: LibraryComponent },
      { path: 'search', component: SearchComponent },
      {
        path: 'playlist/:platform/:id',
        loadComponent: () =>
          import('./pages/playlist/playlist.component').then((m) => m.PlaylistComponent),
      },
      {
        path: 'artist/:platform/:id',
        loadComponent: () =>
          import('./pages/artist/artist.component').then((m) => m.ArtistComponent),
      },
      {
        path: 'user-playlist/:id',
        loadComponent: () =>
          import('./pages/user-playlist/user-playlist.component').then(
            (m) => m.UserPlaylistComponent,
          ),
```

```
      },
    ],
  },
  { path: '', redirectTo: 'login', pathMatch: 'full' },
];

import { Component, inject, signal } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { AuthService } from '../../services/auth.service';
import { LibraryService } from '../../services/library.service';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './login.component.html',
  styleUrl: './login.component.css'
})
export class LoginComponent {
  private fb = inject(FormBuilder);
  private authService = inject(AuthService);
  private libraryService = inject(LibraryService);
  private router = inject(Router);

  ngOnInit() {
    this.libraryService.getMe().subscribe({
      next: () => {
        this.router.navigate(['/main']);
      },
      error: () => {
        // Not logged in, stay on login page
      }
    });
  }

  loginForm = this.fb.group({
    email: ['', [Validators.required, Validators.email]],
    password: ['', [Validators.required]]
  });
```

```
  errorMessage = signal<string | null>(null);

  onSubmit() {
    if (this.loginForm.valid) {
      this.errorMessage.set(null);
      this.authService.login(this.loginForm.value).subscribe({
        next: () => {
          // Navigate to home or dashboard after success
          // For now, maybe just log or reload, or navigate to root
          this.router.navigate(['/']);
        },
        error: (err) => {
          console.error(err);
          if (err.status === 406 && err.error?.problem_fields) {
            const problems = err.error.problem_fields;
            if (problems.Email) {
              this.loginForm.get('email')?.setErrors({ serverError: problems.Email });
            }
            if (problems.Password) {
              this.loginForm.get('password')?.setErrors({ serverError: problems.Password });
            }
          } else {
            this.errorMessage.set('Login failed. Please check your credentials.');
          }
        }
      });
    }
  }
}

import { Component, inject, OnInit, signal } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { AuthService } from '../../services/auth.service';
import { LibraryService } from '../../services/library.service';

@Component({
  selector: 'app-register',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
```

```
    templateUrl: './register.component.html',
    styleUrl: './register.component.css' // We can reuse the same style if we want, but for
now separate file
  })
  export class RegisterComponent implements OnInit {
    private fb = inject(FormBuilder);
    private authService = inject(AuthService);
    private libraryService = inject(LibraryService);
    private router = inject(Router);

    ngOnInit() {
      this.libraryService.getMe().subscribe({
        next: () => {
          this.router.navigate(['/main']);
        },
        error: () => {
          // Not logged in, proceed
        }
      });
    }

    registerForm = this.fb.group({
      username: ['', [Validators.required, Validators.minLength(3)]],
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required, Validators.minLength(6)]]
    });

    errorMessage = signal<string | null>(null);

    onSubmit() {
      if (this.registerForm.valid) {
        this.errorMessage.set(null);
        this.authService.register(this.registerForm.value).subscribe({
          next: () => {
            // On successful register, navigate to verify
            this.router.navigate(['/verify']);
          },
          error: (err) => {
            console.error(err);
            if (err.status === 406 && err.error?.problem_fields) {
              const problems = err.error.problem_fields;
              if (problems.Username) {
```

```
                        this.registerForm.get('username')?.setErrors({ serverError:
problems.Username });
                    }
                    if (problems.Email) {
                        this.registerForm.get('email')?.setErrors({ serverError: problems.Email
});
                    }
                    if (problems.Password) {
                        this.registerForm.get('password')?.setErrors({ serverError:
problems.Password });
                    }
                } else {
                    this.errorMessage.set('Registration failed. Username or Email might be
taken.');
                }
            }
        });
    }
}
}

import { Component, inject, OnInit, signal } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { AuthService } from '../../services/auth.service';
import { LibraryService } from '../../services/library.service';

@Component({
  selector: 'app-verify',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './verify.component.html',
  styleUrl: './verify.component.css'
})
export class VerifyComponent implements OnInit {
  private fb = inject(FormBuilder);
  private authService = inject(AuthService);
  private libraryService = inject(LibraryService);
  private route = inject(ActivatedRoute);
  private router = inject(Router);
```

```
verifyForm = this.fb.group({
  code: ['', [Validators.required, Validators.minLength(6)]]
});

status = signal<'idle' | 'verifying' | 'success' | 'error'>('idle');
message = signal<string>('Enter the verification code sent to your email.');

ngOnInit() {
  // Check if code is in the URL
  this.route.paramMap.subscribe(params => {
    const code = params.get('code');
    if (code) {
      this.verifyCode(code);
    } else {
      // Only check auth if not verifying a code immediately (optional logic, but safe)
      this.libraryService.getMe().subscribe({
        next: () => {
          this.router.navigate(['/main']);
        },
        error: () => {
          // Not logged in, proceed
        }
      });
    }
  });
}

onSubmit() {
  if (this.verifyForm.valid) {
    const code = this.verifyForm.value.code!;
    this.verifyCode(code);
  }
}

verifyCode(code: string) {
  this.status.set('verifying');
  this.message.set('Verifying your code...');

  this.authService.verify(code).subscribe({
    next: () => {
      this.status.set('success');
      this.message.set('Verification successful! Redirecting to login...');
```

```
        setTimeout(() => this.router.navigate(['/login']), 2000);
      },
      error: (err) => {
        console.error(err);
        this.status.set('error');
        this.message.set('Verification failed. Invalid or expired code.');
      }
    });
  }
}

import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { LibraryService, UserWithPlaylists } from '../../services/library.service';
import { PlaylistCardComponent } from '../../components/media-items/playlist-card/playlist-card.component';
import { Platform } from '../../services/platform.service';

@Component({
  selector: 'app-library',
  standalone: true,
  imports: [CommonModule, PlaylistCardComponent],
  templateUrl: './library.component.html',
  styleUrls: ['./library.component.css'],
})
export class LibraryComponent implements OnInit {
  user: UserWithPlaylists | null = null;
  greeting: string = 'Good day';
  readonly Platform = Platform;

  constructor(private libraryService: LibraryService) {}

  ngOnInit(): void {
    this.updateGreeting();
    this.libraryService.currentUser$.subscribe((data) => {
      this.user = data;
    });
  }

  updateGreeting(): void {
    const hour = new Date().getHours();
    if (hour < 12) {
```

```
      this.greeting = 'Good morning';
    } else if (hour < 18) {
      this.greeting = 'Good afternoon';
    } else {
      this.greeting = 'Good evening';
    }
  }

  deletePlaylist(id: number, event: Event): void {
    event.stopPropagation();
    if (confirm('Are you sure you want to delete this playlist?')) {
      this.libraryService.deletePlaylist(id).subscribe({
        error: (err) => console.error('Failed to delete playlist', err),
      });
    }
  }
}

import { Component, inject, input, signal } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { catchError, of } from 'rxjs';
import { rxResource, toSignal } from '@angular/core/rxjs-interop';
import { PlayerService } from '../../services/player.service';
import { PlatformService, Platform, ApiSearchPage } from '../../services/platform.service';
import { TrackItemComponent } from '../../components/media-items/track/track-item.component';
import { PlaylistCardComponent } from '../../components/media-items/playlist-card/playlist-card.component';
import { ArtistCardComponent } from '../../components/media-items/artist-card.component';
import checkParamsBeforeCall from '../../utils/check-params-before-call';
import { Router } from '@angular/router';

@Component({
  selector: 'app-search',
  standalone: true,
  imports: [
    CommonModule,
    FormsModule,
    TrackItemComponent,
    PlaylistCardComponent,
```

```
    ArtistCardComponent,
  ],
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.css'],
})
export class SearchComponent {
  query = input<string | undefined>(undefined, { alias: 'q' });

  private platformService = inject(PlatformService);
  private playerService = inject(PlayerService);
  private router = inject(Router);

  platform = toSignal(this.platformService.platform$);

  searchResource = rxResource<
    ApiSearchPage | null,
    { query: string | undefined; platform: Platform | undefined }
  >({
    params: () => {
      const q = this.query();
      const p = this.platform();
      if ((!q || !p) && this.platformService.lastPerformedSearch) {
        this.router.navigate(['/search'], {
          queryParams: {
            q: this.platformService.lastPerformedSearch.query,
          },
        });
        return { query: this.platformService.lastPerformedSearch.query, platform: p };
      }
      return { query: q, platform: p };
    },
    stream: ({ params }) =>
      checkParamsBeforeCall(
        this.platformService.search.bind(this.platformService),
        params.query,
        params.platform,
      ),
  });

  get searchResults() {
    return this.searchResource.value;
  }
```

```
  get isLoading() {
    return this.searchResource.isLoading;
  }

  playTrack(index: number) {
    const results = this.searchResults();
    if (results?.tracks) {
      this.playerService.setQueue(results.tracks, index);
    }
  }
}

import { Component, inject, input, effect } from '@angular/core';
import { rxResource } from '@angular/core/rxjs-interop';
import { PlayerService } from '../../services/player.service';
import { PlatformService, Platform, ApiPlaylist } from '../../services/platform.service'; //
Added ApiPlaylist import
import { TrackItemComponent } from '../../components/media-items/track/track-
item.component';
import { PlaylistHeaderComponent } from '../../sections/playlist-header/playlist-
header.component';
import { ToastService } from '../../services/toast.service';
import checkParamsBeforeCall from '../../utils/check-params-before-call';
import { PlaylistControlsComponent } from '../../sections/playlist-controls/playlist-
controls.component';

@Component({
  selector: 'app-playlist',
  standalone: true,
  imports: [TrackItemComponent, PlaylistHeaderComponent,
PlaylistControlsComponent],
  templateUrl: './playlist.component.html',
  styleUrl: './playlist.component.css',
})
export class PlaylistComponent {
  // Use specific types for inputs to help the Resource
  id = input<string | undefined>();
  platform = input<Platform | undefined>();

  private playerService = inject(PlayerService);
  private platformService = inject(PlatformService);
```

```
  private toastService = inject(ToastService);

  constructor() {
    effect(() => {
      const error = this.error();
      // Показываем тост только если ошибка реально существует
      if (error) {
        this.toastService.show(error.message || 'Failed to load playlist', 'error');
      }
    });
  }

  // Explicitly type the Resource <ValueType, RequestType>
  playlistResource = rxResource<
    ApiPlaylist,
    { id: string | undefined; platform: Platform | undefined }
  >({
    params: () => ({
      id: this.id(),
      platform: this.platform(),
    }),
    stream: ({ params }) =>
      checkParamsBeforeCall(
        this.platformService.getPlaylist.bind(this.platformService),
        params.id,
        params.platform,
      ),
  });

  // Derived signals
  playlist = this.playlistResource.value;
  isLoading = this.playlistResource.isLoading;
  error = this.playlistResource.error;
}

import { Component, effect, inject, input, OnInit, signal } from '@angular/core';
import { CommonModule } from '@angular/common';
import { PlayerService } from '../../services/player.service';
import {
  PlatformService,
  Platform,
  ApiTrack,
```

```
  ApiArtistDetails,
} from '../../services/platform.service';
import { TrackItemComponent } from '../../components/media-items/track/track-item.component';
import { PlaylistCardComponent } from '../../components/media-items/playlist-card/playlist-card.component';
import { rxResource } from '@angular/core/rxjs-interop';
import checkParamsBeforeCall from '../../utils/check-params-before-call';
import { ToastService } from '../../services/toast.service';

@Component({
  selector: 'app-artist',
  standalone: true,
  imports: [CommonModule, TrackItemComponent, PlaylistCardComponent],
  templateUrl: './artist.component.html',
  styleUrl: './artist.component.css',
})
export class ArtistComponent {
  // route params
  id = input<string | undefined>();
  platform = input<Platform | undefined>();

  private platformService = inject(PlatformService);
  private playerService = inject(PlayerService);
  private toastService = inject(ToastService);

  constructor() {
    effect(() => {
      const error = this.error();
      if (error) {
        this.toastService.show(error.message || 'Failed to load artist', 'error');
      }
    });
  }

  artistResource = rxResource<
    ApiArtistDetails,
    { id: string | undefined; platform: Platform | undefined }
  >({
    params: () => ({
      id: this.id(),
      platform: this.platform(),
```

```
    }),
    stream: ({ params }) =>
      checkParamsBeforeCall(
        this.platformService.getArtistDetails.bind(this.platformService),
        params.id,
        params.platform,
      ),
  });

  artist = this.artistResource.value;
  isLoading = this.artistResource.isLoading;
  error = this.artistResource.error;

  playAll() {
    const tracks = this.artist()?.tracks;
    if (tracks && tracks.length > 0) {
      this.playerService.setQueue(tracks);
    }
  }

  playTrackFromList(index: number) {
    const tracks = this.artist()?.tracks;
    if (tracks && tracks.length > 0) {
      this.playerService.setQueue(tracks, index);
    }
  }

  playTrack(track: ApiTrack) {
    this.playerService.playTrack(track);
  }
}

import { Component, effect, inject, input } from '@angular/core';
import {
  LibraryService,
  TransformedUserPlaylist,
  TransformedTrackInPlaylist,
  TransformedTrackInPlaylistFullData,
} from '../../services/library.service';
import { rxResource, toObservable, toSignal } from '@angular/core/rxjs-interop';
import { concat, concatMap, delay, from, of, scan, switchMap } from 'rxjs';
import checkParamsBeforeCall from '../../utils/check-params-before-call';
```

```
import { ToastService } from '../../services/toast.service';
import { Platform } from '../../services/platform.service';
import { PlayerService } from '../../services/player.service';
import { TrackItemComponent } from '../../components/media-items/track/track-item.component';
import { PlaylistHeaderComponent } from '../../sections/playlist-header/playlist-header.component';
import { PlaylistControlsComponent } from '../../sections/playlist-controls/playlist-controls.component';

@Component({
 selector: 'app-user-playlist',
 imports: [TrackItemComponent, PlaylistHeaderComponent, PlaylistControlsComponent],
 templateUrl: './user-playlist.component.html',
 styleUrl: './user-playlist.component.css',
})
export class UserPlaylistComponent {
 id = input<string | undefined>();

 protected readonly Platform = Platform;

 private libraryService = inject(LibraryService);
 private toastService = inject(ToastService);
 private playerService = inject(PlayerService);

 constructor() {
  effect(() => {
   const error = this.error();
   if (error) {
    this.toastService.show(error.message || 'Failed to load playlist', 'error');
   }
  });
 }

 playlistResource = rxResource<TransformedUserPlaylist, { id: string | undefined }>({
  params: () => ({ id: this.id() }),
  stream: ({ params }) =>
   checkParamsBeforeCall((id: string) => this.libraryService.getUserPlaylist(+id), params.id),
 });
```

```
playlist = toSignal(
  toObservable(this.playlistResource.value).pipe(
    switchMap((playlist) => {
      if (!playlist) return of(null);

      const partialTracks = playlist.tracks.filter(
        (t): t is TransformedTrackInPlaylist => t.type === 'partial',
      );

      if (partialTracks.length === 0) return of(playlist);

      const enriched$ = from(partialTracks).pipe(
        concatMap((track) => this.libraryService.getFullTrack(track).pipe(delay(500))),
        scan(
          (acc, fullTrack) => ({
            ...acc,
            tracks: acc.tracks.map((t) =>
              t.track_in_playlist_id === fullTrack.track_in_playlist_id ? fullTrack : t,
            ),
          }),
          playlist,
        ),
      );

      return concat(of(playlist), enriched$);
    }),
  ),
  { initialValue: null },
);

isLoading = this.playlistResource.isLoading;
error = this.playlistResource.error;

playPlaylist() {
  const data = this.playlist();
  if (!data) return;
  const fullTracks = data.tracks.filter(
    (t): t is TransformedTrackInPlaylistFullData => t.type === 'full',
  );
  if (fullTracks.length > 0) {
    this.playerService.setQueue(fullTracks, 0);
  }
```

```
  }
}
```