Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Музичний стримінговий сервіс на основі хмарних технологій
(тема)

Виконав:
здобувач 3 року навчання,
групи КІУКІу-22-2
Максим КАКУНЯ
(власне ім’я, прізвище)

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

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

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

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

Допускається до захисту

Завідувач кафедри ЕОМ ________________ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

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

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

Освітня програма Комп’ютерна інженерія
(повна назва)

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

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Какуні Максиму Вячеславовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Музичний стримінговий сервіс на основі хмарних технологій

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

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

3. Вхідні дані до роботи 1) функціональні вимоги до музичного стримінгового сервісу; 2) особливості використання хмарної інфраструктури AWS; 3) вимоги до архітектури клієнт-сервер (SPA + REST); 4) принципи мікросервісної архітектури; 5) специфіка 3-layers підходу у Web API та SRP; 6) API сторонніх сервісів (AWS MediaConvert, CloudFront, SES тощо); 7) інтерфейси користувача, натхненні сервісами Spotify; 8) рекомендації щодо реалізації авторизації та потокової передачі медіаконтенту; 9) технічне середовище розробки: C#, React, TypeScript, Docker, PostgreSQL тощо.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз проблеми та огляд існуючих рішень;
2) вибір технології розробки та інструментальних засобів;
3) розробка алгоритмічного забезпечення;
4) розробка сервісів;
5) відлагодження програмних модулів;
6) висновки.

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

Слайд-презентація – 13 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

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

Дата видачі завдання “ 09 ” червня 2025 р.

Здобувач ____________
(підпис)

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

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 86 с., 19 рисунків, 12 джерел.

ASP.NET CORE, EF CORE, JWT, REACT, ZUSTAND, JS, JSX, JOY UI, HTML, CSS, DOCKER, DOCKER-COMPOSE, PORTAINER, TRAEFIK, POSTGRESQL, SQL, REDIS, RABBITMQ, RPC, MESSAGEPACK, JETBRAINS RIDER, WEBSTORM, DATAGRIP, JIRA, BITBUCKET, GIT, AWS SES, AWS EC2, AWS ROUTE 53, МІКРОСЕРВІСНА АРХІТЕКТУРА, ВЕБ-ЗАСТОСУНОК, КЛІЄНТ, СЕРВЕР.

Об'єктом дослідження є аналіз існуючих музичних стримінгових платформ, зокрема функціональності пошуку та фільтрації контенту, включаючи плейлисти користувачів, альбоми та треки конкретних виконавців.

Метою кваліфікаційної роботи є розробка музичного стримінгового сервісу на основі вебтехнологій, який дозволить користувачам прослуховувати музику онлайн, створювати власні плейлисти та здійснювати пошук та фільтрацію контенту за різними критеріями: жанрами, виконавцями, альбомами.

Основна задача роботи полягає в розробці інтерфейсу, який дозволить користувачам зручно здійснювати пошук та фільтрацію треків, альбомів та плейлистів за заданими параметрами. Фільтрація та пошук будуть реалізовані на основі метаданих, таких як жанри, виконавці та назви альбомів, що дозволить значно покращити досвід користувача.

ABSTRACT

Bachelor's thesis: 86 pages, 19 figures, 12 sources.

ASP.NET CORE, EF CORE, JWT, REACT, ZUSTAND, JS, JSX, JOY UI, HTML, CSS, DOCKER, DOCKER-COMPOSE, PORTAINER, TRAEFIK, POSTGRESQL, SQL, REDIS, RABBITMQ, RPC, MESSAGEPACK, JETBRAINS RIDER, WEBSTORM, DATAGRIP, JIRA, BITBUCKET, GIT, AWS SES, AWS EC2, AWS ROUTE 53, MICROSERVICE ARCHITECTURE, WEB APPLICATION, CLIENT, SERVER.

Object of the research is the analysis of existing music streaming platforms, particularly the functionality of search and content filtering, including user playlists, albums, and tracks by specific artists.

The aim of the qualification project is to develop a music streaming service based on web technologies, which will allow users to listen to music online, create their own playlists, and perform search and filtering of content based on various criteria such as genres, artists, and albums.

The main task of the project is to develop an interface that enables users to conveniently search for and filter tracks, albums, and playlists according to specified parameters. Filtering and search will be implemented based on metadata such as genres, artists, and album titles, which will significantly enhance the user experience.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 9
ВСТУП 11
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 13
1.1 Аналіз функціональності музичних стримінгових платформ 13
1.2 Огляд механізмів пошуку та фільтрації контенту 16
1.3 Музичний стримінговий сервіс Spotify 18
1.4 Платформа для прослуховування музики YouTube Music 19
1.5 Висновок по розділу 21
2 ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ ТА ІНСТРУМЕНТИ ДЛЯ РОЗРОБКИ ВЕБЗАСТОСУНКУ 23
2.1 Серверна частина та база даних 23
2.1.1 Мова програмування C# 23
2.1.2 Фреймворк ASP.NET Core 24
2.1.3 ORM-фреймворк Entity Framework Core 24
2.1.4 Контейнеризація та Docker 25
2.1.5 Серіалізація даних та MessagePack 25
2.1.6 Обмін повідомленнями та RabbitMQ 26
2.1.7 Взаємодія сервісів через RabbitMQ RPC 26
2.1.8 Мова запитів SQL 27
2.1.9 Реляційна база даних PostgreSQL 27
2.1.10 Кешування та зберігання тимчасових даних у Redis 28
2.1.11 Авторизація на основі JWT 29
2.2 Клієнтська частина 30
2.2.1 Бібліотека React для розробки інтерфейсу користувача 30
2.2.2 Керування станом застосунку за допомогою Zustand 31
2.2.3 Мова програмування JavaScript та синтаксис JSX 31
2.2.4 Бібліотека компонентів Joy UI 32
2.2.5 Використання HTML та CSS у вебзастосунку 33

2.3 Хмарні технології AWS 33
2.3.1 Використання AWS EC2 для розгортання серверної частини 33
2.3.2 Використання AWS Route 53 для управління доменними іменами 34
2.3.3 Зберігання файлів у AWS S3 34
2.3.4 Оптимізація контенту через AWS CloudFront 35
2.3.5 Обробка медіафайлів через AWS MediaConvert 35
2.3.6 Автоматизація процесів за допомогою AWS Lambda 36
2.3.7 Надсилання електронних листів через AWS SES 36
2.3.8 Управління доступом через AWS IAM 37
2.4 Проксіювання та балансування навантаження у вебзастосунку 37
2.5 Середовище розробки та конфігурація сервера 38
2.5.1 Управління процесом розробки 38
2.5.2 Система контролю версій та репозиторії коду 39
2.5.3 Контейнеризація та управління Docker-образами 40
2.5.4 Інструменти розробки 40
2.5.5 Конфігурація серверного середовища 41
3 ВЗАЄМОДІЯ КОМПОНЕНТІВ ТА АРХІТЕКТУРА ВЕБЗАСТОСУНКУ 42
3.1 Архітектура отримання та конвертації медіаконтенту у AWS 42
3.2 Архітектура взаємодії Frontend із AWS для відтворення медіаконтенту 45
3.3 Взаємодія мікросервісів при отриманні Signed URL для відтворення медіаконтенту 47
3.4 Архітектура мікросервісів системи Soundify та призначення сервісів 49
3.5 Архітектура клієнтської частини системи Soundify 52
3.5.1 Загальна архітектура клієнтської частини 53
3.5.2 Шари архітектури та патерн State-Driven 54
3.5.3 Hosted Services (Back Tasks) 56

3.5.4 Dependency Injection (DI) 58
3.6 Архітектура баз даних системи Soundify 60
4 ІНСТРУКЦІЯ КОРИСТУВАЧА 66
ВИСНОВКИ 77
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 78
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 79

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОС – операційна система

ПК – персональний комп’ютер

СУБД – система управління базами даних

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

ASP.NET Core – фреймворк для розробки вебзастосунків від Microsoft

AWS – хмарна платформа (англ., Amazon Web Services)

AWS EC2 – служба для надання віртуальних серверів в хмарі AWS

AWS Route 53 – служба DNS в хмарі AWS

AWS SES – служба для відправки електронних листів в хмарі AWS

Bitbucket – вебсервіс для хостингу репозиторіїв Git, інтегрований з Jira

CLI – інтерфейс командного рядку (англ., Command Line Interface)

CSS – каскадні таблиці стилів (англ., Cascading Style Sheets)

Datagrip – інструмент для роботи з базами даних від JetBrains

Docker – платформа для контейнеризації застосунків

Docker-Compose – інструмент для опису та запуску багатоконтейнерних Docker-застосунків

EF Core – ORM для роботи з базами даних у .NET (англ., Entity Framework Core)

Git – система керування версіями

HTML – мова розмітки гіпертексту (англ., HyperText Markup Language)

JetBrains Rider – IDE для розробки .NET-додатків

Jira – інструмент для управління проєктами та відстеження задач

JS – мова програмування (англ., JavaScript)

JSON – формат обміну даними (англ., JavaScript Object Notation)

JSX – синтаксис розширення JavaScript для React (англ., JavaScript XML)

JWT – стандарт для безпечного обміну даними між сторонами (англ.,

JSON Web Token)

MessagePack – бінарний формат серіалізації даних

ORM – об’єктно-реляційне відображення (англ., Object-Relational Mapping)

Portainer – інтерфейс для управління контейнерами Docker

PostgreSQL – система управління базами даних

RabbitMQ – брокер повідомлень для обміну даними між сервісами

React – JavaScript-бібліотека для побудови інтерфейсу користувача

Redis – база даних для швидкого зберігання та обміну даними

RPC – віддалений виклик процедур (англ., Remote Procedure Call)

SQL – мова структурованих запитів (англ., Structured Query Language)

Traefik – реверсний проксі-сервер та балансувальник навантаження для контейнеризованих додатків

UI – інтерфейс користувача (англ., User Interface)

Webstorm – застосунок для веброзробки від JetBrains

Zustand – бібліотека для управління станом в React-застосунках

ВСТУП

Вебзастосунки стали невід'ємною частиною сучасного цифрового середовища, і серед них особливе місце займають музичні стримінгові сервіси, які надають користувачам можливість слухати музику без необхідності зберігати треки на своїх пристроях. Завдяки таким платформам, як Spotify, Apple Music, YouTube Music, користувачі можуть отримувати доступ до величезних бібліотек музичних композицій, що значно спрощує процес пошуку улюблених пісень та відкриття нових виконавців. Музичні сервіси зручні тим, що забезпечують можливість слухати музику на будь-якому пристрої – від смартфонів до комп'ютерів, що робить їх універсальними інструментами для прослуховування музики у будь-який час та в будь-якому місці.

Музичні стримінгові платформи об'єднують у собі безліч складних технологічних рішень, таких як передові методи обробки аудіофайлів, зберігання та маніпулювання метаданими (наприклад, інформацією про виконавців, альбоми, жанри та інші параметри), а також створення персоналізованих рекомендацій, що допомагають користувачам знаходити нові треки, які відповідають їх смакам. Розробка таких платформ вимагає використання новітніх вебтехнологій, здатних ефективно обробляти великі обсяги даних, а також інтеграцію з потужними серверними системами та хмарними сховищами, що забезпечують надійний та безперебійний доступ до контенту.

Однією з головних переваг вебзастосунків є їх здатність працювати на різних платформах та пристроях, що дозволяє користувачам слухати музику не тільки на стаціонарних комп'ютерах, а й на мобільних пристроях, планшетах, розумних годинниках та інших девайсах. Завдяки використанню сучасних вебтехнологій забезпечується висока інтерактивність інтерфейсу, що дає можливість користувачам створювати персоналізовані плейлисти, обирати улюблені треки, переглядати новинки, а також взаємодіяти з іншими

користувачами. Вебтехнології дозволяють також ефективно працювати з великими обсягами даних, наприклад, при відтворенні аудіофайлів у високій якості без значних затримок або відставань.

Актуальність створення музичного стримінгового сервісу обумовлена не лише розвитком технологій, а й змінами у споживчих звичках. В умовах стрімкого розвитку мобільних технологій та широкої доступності високошвидкісного інтернету, все більше користувачів віддають перевагу онлайн-сервісам для доступу до музики. Музичні платформи, зокрема Spotify, Apple Music та YouTube Music, активно змінюють індустрію музики, надаючи користувачам не лише доступ до широких музичних бібліотек, а й можливість створювати індивідуальні рекомендації, проводити інтеграцію з іншими сервісами та створювати різноманітні функціональні можливості для взаємодії з контентом.

Для створення такого сервісу необхідно враховувати багато аспектів, зокрема організацію роботи з великою кількістю файлів та метаданих, безпеку даних користувачів, а також зручність та швидкість роботи інтерфейсу. Сервіс має бути максимально простим та інтуїтивно зрозумілим для користувачів з різним рівнем технічної підготовки. Важливим етапом розробки є створення ефективних алгоритмів для персоналізації рекомендацій, а також інтеграція з різними хмарними технологіями для зберігання музичного контенту.

Метою цієї кваліфікаційної роботи є розробка вебзастосунку для музичного стримінгу, який дозволить користувачам слухати музику, створювати персоналізовані плейлисти, інтегрувати аудіофайли в хмарні сховища, а також отримувати рекомендації, що ґрунтуються на їхніх перевагах. Створення такого сервісу не лише дозволяє зручніше взаємодіяти з музичним контентом, а й сприяє розвитку інноваційних технологій у сфері веброзробки та музичного стримінгу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз функціональності музичних стримінгових платформ

У сучасному світі музичні стримінгові платформи стали важливим інструментом для прослуховування музики, пропонуючи користувачам доступ до величезних каталогів пісень, альбомів та плейлистів. Огляд таких популярних платформ, як Spotify, Apple Music та YouTube Music, дозволяє зрозуміти їхні архітектурні особливості, функціональність та моделі монетизації. Кожна з цих платформ має свою унікальну архітектуру та концепцію надання музичного контенту. Spotify є однією з найбільших у світі платформ для стримінгу, що пропонує як безкоштовний доступ з рекламою, так і преміум-підписку, яка дозволяє користуватися всіма можливостями сервісу без перерв на рекламу. Цей сервіс акцентує увагу на персоналізованих рекомендаціях, що реалізуються завдяки складним алгоритмам машинного навчання. Apple Music, у свою чергу, має тісну інтеграцію з іншими продуктами Apple та пропонує високоякісне аудіо, а також екслюзивний контент, доступний лише для користувачів цієї платформи. YouTube Music, як частина екосистеми Google, забезпечує не лише музику, але й відеокліпи, що дозволяє користувачам знайти відео та слухати музику в одному місці.

Усі ці сервіси базуються на розподілених архітектурах, де музика зберігається на віддалених серверах, а користувачі можуть отримати до неї доступ через інтерфейси на різних пристроях – смартфонах, планшетах, комп'ютерах і навіть телевізорах. Важливою частиною цих платформ є також платіжні системи, які дозволяють монетизувати послугу через підписки або рекламу.

Музичні стримінгові платформи надають своїм користувачам широкий спектр функцій, які значно покращують досвід використання. Кожен з цих сервісів має потужні можливості для пошуку та фільтрації контенту,

дозволяючи користувачам знаходити музику за різними параметрами, такими як жанр, артист або альбом. Також доступні фільтри за популярністю, новизною чи рекомендаціями, що робить пошук значно зручнішим.

Персоналізовані рекомендації – одна з ключових переваг таких платформ. Завдяки використанню алгоритмів машинного навчання, платформи здатні аналізувати слухацькі вподобання та пропонувати музику, яка, ймовірно, сподобається конкретному користувачу. Це дозволяє створювати індивідуальні рекомендації, що покращують досвід користувача та сприяють відкриттю нових артистів та жанрів.

Крім того, користувачі можуть створювати власні плейлисти або зберігати улюблені пісні для подальшого прослуховування. Більшість платформ дозволяє створювати багато плейлистів для різних настроїв чи подій, а також підтримує збереження музики для оффлайн-прослуховування, що особливо важливо для користувачів з обмеженим доступом до Інтернету. Всі ці функції створюють багатофункціональне середовище для прослуховування музики, що відповідає різним потребам та вимогам користувачів.

Що стосується монетизації, то існують різні моделі, які дозволяють платформам генерувати дохід. Підписка є основною моделлю для більшості стримінгових сервісів. Користувачі можуть обирати між безкоштовною версією з рекламою та преміум-підпискою, що надає доступ до додаткових функцій, таких як прослуховування без реклами, можливість збереження треків для оффлайн-прослуховування та доступ до ексклюзивного контенту. Наприклад, Spotify пропонує декілька типів підписок: від безкоштовного тарифу до різних преміум-опцій, включаючи сімейні та студентські плани.

Apple Music також спирається на підписку, яка дозволяє користувачам отримати доступ до музики без обмежень, причому в деяких випадках можливе поєднання підписки з іншими послугами Apple. YouTube Music у свою чергу пропонує безкоштовний доступ з рекламою, а також преміум-підписку для тих, хто бажає користуватися платформою без реклами.

Реклама є основним джерелом доходу для безкоштовних користувачів. Наприклад, YouTube Music надає можливість прослуховувати музику з рекламою, що дозволяє зберігати платформі стійкий потік доходу навіть від користувачів, які не готові платити за підписку. Однак реклама може бути нав'язливою та створювати неприємний досвід для користувачів.

Незважаючи на популярність, музичні стримінгові платформи стикаються з низкою проблем, які впливають на їх функціонування та досвід користувача. Одним з головних є питання зберігання даних, оскільки платформи мають справу з величезними обсягами музичного контенту, який потрібно зберігати та доставляти користувачам у найкращій якості. Для цього необхідно постійно модернізувати сервери та інфраструктуру для ефективної доставки музики.

Інша важлива проблема – це авторські права. Платформи повинні укладати договори з музичними лейблами та артистами, щоб уникнути порушення авторських прав, що вимагає значних юридичних та фінансових зусиль, адже порушення ліцензій може призвести до судових позовів або навіть закриття сервісу в певних країнах. Окрім цього, стримінгові сервіси мають враховувати регуляторні вимоги різних країн, зокрема локалізацію контенту, захист даних користувачів та оподаткування.

Ще однією важливою проблемою є ефективність алгоритмів пошуку та рекомендацій. Не завжди алгоритми правильно передбачають вподобання користувачів, що може призвести до непотрібних або нецікавих пропозицій, що змушує компанії постійно вдосконалювати свої системи для надання більш точних рекомендацій. І, звісно, на ринку існує висока конкуренція, що вимагає від платформ постійного вдосконалення функціоналу, залучення нових користувачів через маркетингові кампанії та укладання угод з артистами для надання ексклюзивного контенту. Таким чином, музичні стримінгові платформи є важливим елементом сучасної музичної індустрії, але вони стикаються з численними викликами, що вимагають постійного розвитку технологій та бізнес-моделей.

1.2 Огляд механізмів пошуку та фільтрації контенту

Механізми пошуку та фільтрації контенту на музичних стримінгових платформах відіграють вирішальну роль у створенні зручного досвіду користувача. Сучасні платформи забезпечують потужні інструменти для знаходження музики за різними критеріями, що дозволяє користувачам швидко знаходити потрібний контент, зберігаючи при цьому високу точність пошуку.

Один із основних підходів до пошуку музики полягає в можливості пошуку за жанрами. Жанри – це один з основних критеріїв класифікації музики, і вони допомагають користувачам орієнтуватися серед величезного каталогу музичних творів. Платформи, такі як Spotify чи Apple Music, пропонують широкий спектр жанрів, від попу та року до електронної музики та класики. Відповідно до цього користувачі обирають музику відповідно до їхніх уподобань або емоційного стану, а також допомагають платформам створювати рекомендовані плейлисти на основі жанрових переваг.

Не менш важливим є пошук за виконавцями, що є ще одним популярним способом знайти улюблену музику. Користувачі часто шукають конкретних артистів або групи, і платформи дозволяють робити це швидко та ефективно, пропонуючи додатково рекомендації схожих виконавців. Такий підхід допомагає не лише знаходити музику від конкретних артистів, але й відкривати нові імена, що можуть зацікавити користувача.

Пошук за альбомами – це ще один механізм. Багато слухачів віддають перевагу прослуховуванню повних альбомів, а не окремих пісень, тому платформи пропонують можливість пошуку та фільтрації саме за альбомами. Це дозволяє користувачам знаходити цілі альбоми, а не просто окремі треки, що значно полегшує досвід слухання для тих, хто хоче оцінити творчість артиста у повному обсязі.

Усі ці механізми пошуку значною мірою залежать від правильного використання метаданих. Метадані – це інформація, що супроводжує кожен

музичний трек, альбом чи виконавця, і вони визначають, як контент буде організований та доступний для пошуку. Для забезпечення високої ефективності пошуку важливо, щоб метадані були точними та структурованими. Зокрема, жанри є основними метаданими, які визначають тип музики та дозволяють класифікувати треки за різними категоріями, допомагаючи користувачам швидко знайти музику, що відповідає їхнім смакам.

Іншим важливим елементом є метадані про виконавців, оскільки вони дозволяють створити зручні інтерфейси для пошуку та фільтрації музики по артистах, що дозволяє користувачам знаходити конкретних виконавців, а також отримувати рекомендації схожих артистів.

Не менш важливими є метадані, що стосуються альбомів. Зокрема, вони дозволяють правильно організовувати та фільтрувати контент за повними релізами, даючи можливість користувачам знаходити цілі альбоми, а не просто окремі треки. Крім того, додаткові метадані у вигляді тегів, що описують настрій чи тему пісні (наприклад, "любов", "енергійний", "меланхолійний"), значно розширюють можливості пошуку, що дає користувачам змогу знаходити музику не лише за жанром чи виконавцем, але й за емоційним чи тематичним змістом. Це робить пошук ще більш персоналізованим.

Метадані також включають рік випуску, який дає змогу користувачам сортувати музику за часом її появи, що може бути важливим для тих, хто цікавиться новими релізами або хоче знайти музику певної епохи. Окрім того, популярність контенту є важливими метаданим, яка допомагає платформам створювати списки найпопулярніших треків чи альбомів, що завжди актуальні для слухачів, які хочуть бути в курсі музичних тенденцій.

Важливість метаданих для фільтрації контенту не можна переоцінити, оскільки саме вони дозволяють платформам здійснювати точний пошук та надавати користувачам тільки релевантний контент, що відповідає їхнім критеріям. Завдяки добре організованим метаданим, пошук музики стає

швидким та точним, що значно покращує загальний досвід користування платформами. Тому для будь-якої музичної стримінгової платформи ключовим фактором є забезпечення точності та повноти метаданих, що дозволяє створювати інтуїтивно зрозумілі та ефективні механізми пошуку та фільтрації контенту.

1.3 Музичний стримінговий сервіс Spotify

Сервіс Spotify (рисунок 1.1) є однією з найбільших та найпопулярніших музичних стримінгових платформ у світі. Платформа пропонує доступ до мільйонів музичних треків, плейлистів, подкастів та навіть відеокліпів. Вона підтримує як безкоштовний доступ з рекламою, так і преміум-підписку, яка дозволяє слухати музику без реклами та завантажувати треки для офлайн-прослуховування.

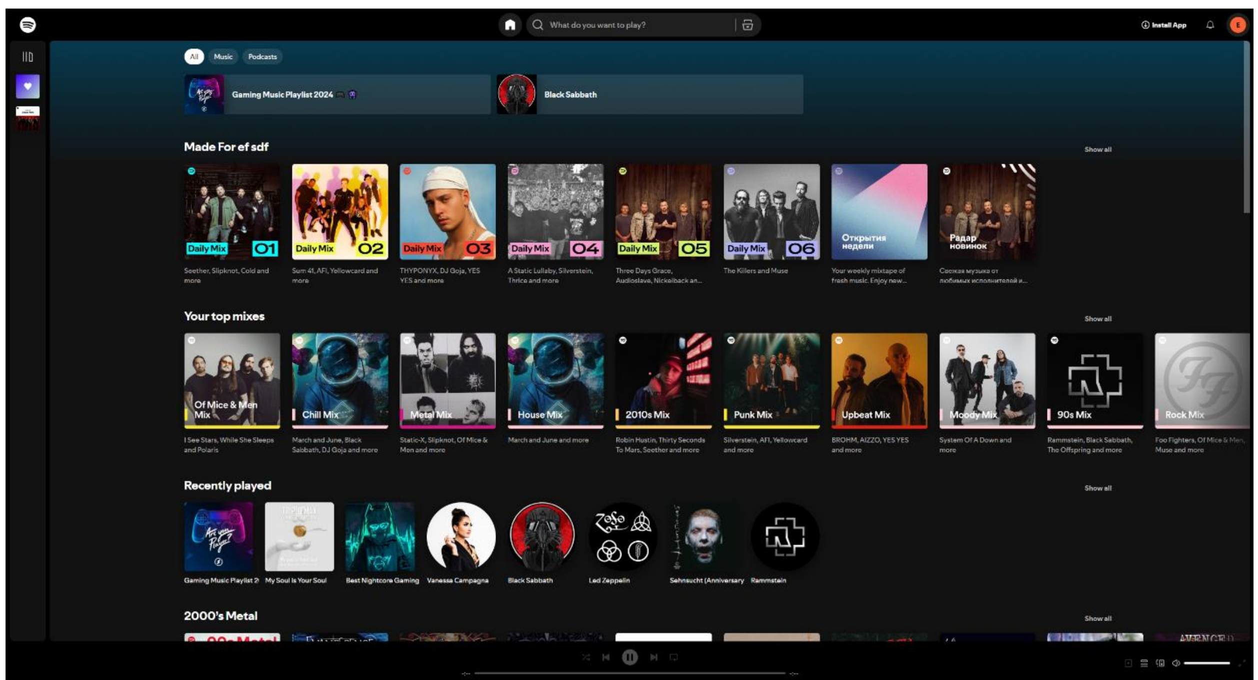

Рисунок 1.1 – Домашня сторінка Spotify

Spotify активно використовує складні алгоритми машинного навчання для персоналізації рекомендацій на основі історії прослуховувань та смаки

користувача. Платформа також пропонує багато функцій для створення та організації плейлистів, що дозволяє користувачам зібрати свої улюблені треки та створити тематичні колекції. Вона доступна на різних пристроях, включаючи смартфони, комп'ютери, смарт-колонки та навіть телевізори, що робить її надзвичайно зручною для використання в будь-яких умовах.

Сервіс Spotify надає розширений пошук, який дозволяє знаходити музику не тільки за виконавцем чи альбомом, але й за настроєм, популярністю та іншими персоналізованими критеріями. Платформа також пропонує фільтрацію за часом, новизною та рейтингами, що дозволяє користувачам швидко знаходити потрібні треки. Також Spotify дозволяє користувачам створювати власні плейлисти, зберігати треки для офлайн-прослуховування, а також пропонує додаткові функції, як-от персоналізовані рекомендації, що ґрунтуються на вподобаннях слухачів. Платформа також підтримує інтеграцію з різними пристроями, включаючи автомобільні системи та смарт-колонки.

Spotify використовує змішану модель монетизації, що включає безкоштовну версію з рекламою та преміум-підписку. Користувачі безкоштовної версії слухають музику з рекламою, але отримують доступ до широкого контенту. Преміум-підписка дає переваги, такі як прослуховування без реклами, можливість завантажувати треки та високоякісне відтворення.

1.4 Платформа для прослуховування музики YouTube Music

YouTube Music є популярною платформою для прослуховування музики, яка входить до екосистеми Google та поєднує можливості музичного стримінгу з переглядом відеокліпів (рисунок 1.2). Вона дозволяє користувачам шукати музику не тільки за назвами пісень та виконавцями, але й за текстами пісень, що є однією з її особливостей. YouTube Music також пропонує велику кількість ексклюзивного контенту, а також можливість переглядати відеокліпи, що робить платформу унікальною серед інших

музичних сервісів. Користувачі можуть створювати плейлисти, додавати улюблені пісні до бібліотеки та насолоджуватися відеоконтентом. Вона доступна на смартфонах, комп'ютерах та через смарт-колонки, що робить її універсальним інструментом для прослуховування музики.

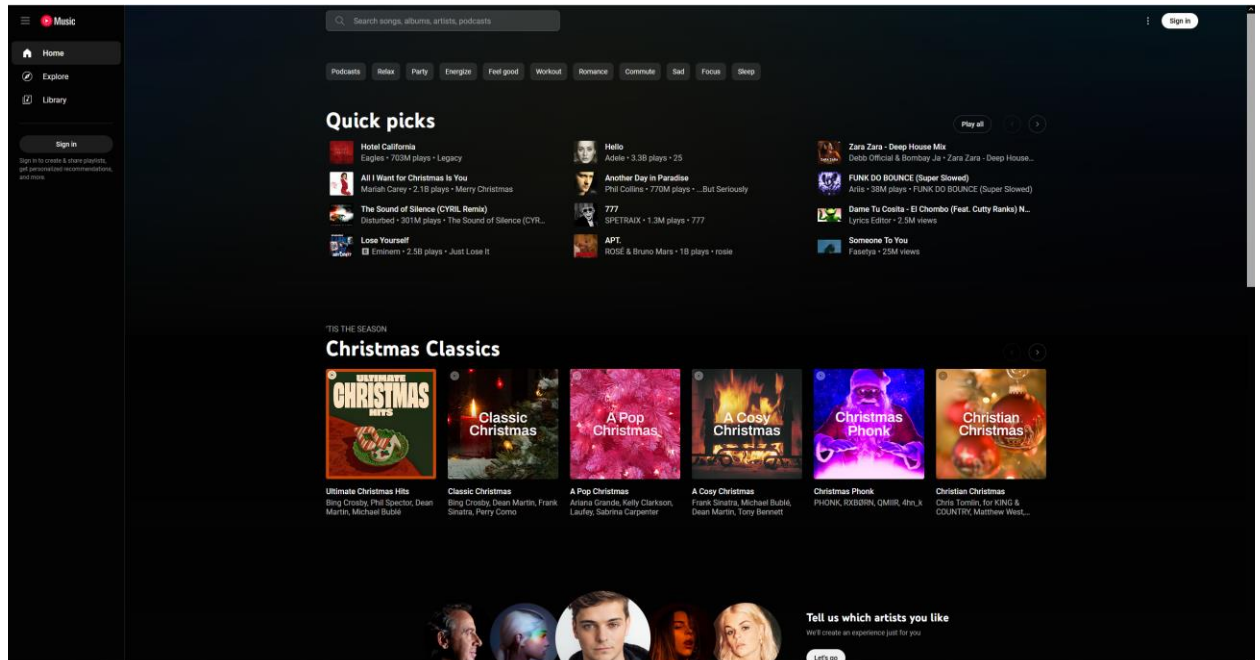

Рисунок 1.2 – Домашня сторінка YouTube Music

YouTube Music пропонує розширений пошук, що дозволяє знаходити музику не тільки за жанром, виконавцем чи альбомом, а й за текстами пісень. Це дає змогу користувачам знаходити композиції навіть за окремими словами чи фразами, що є перевагою порівняно з іншими платформами. Крім того, YouTube Music надає фільтрацію за популярністю, новизною, жанром та іншими параметрами, що дозволяє користувачам точно налаштувати пошук.

YouTube Music дозволяє користувачам не лише слухати музику, але й переглядати відеокліпи. Користувачі можуть шукати відеокліпи, дивитися їх у високій якості, а також переглядати повні альбоми у відеоформаті, що надає додаткові можливості для тих, хто хоче поєднувати аудіо та відео контент. Крім того, платформа підтримує створення персоналізованих

плейлистів, які можуть містити як аудіотреки, так і відео.

Платформа YouTube Music використовує модель, яка включає безкоштовний доступ з рекламою та преміум-підписку. Безкоштовні користувачі можуть слухати музику з перервами на рекламу, тоді як преміум-підписка дозволяє прослуховувати треки без реклами, а також надає додаткові функції, такі як завантаження для офлайн-прослуховування.

1.5 Висновок по розділу

Під час проведення попереднього аналізу існуючих рішень для прослуховування музичного контенту були виявлені недоліки, з якими стикаються користувачі. Результатом цього аналізу стало рішення розробити вебзастосунок, який позбавлений виявлених проблем.

Вебзастосунок орієнтований на користувачів, які шукають простий, доступний та ефективний інструмент для прослуховування музики. На відміну від великих стримінгових платформ, таких як Spotify та YouTube Music, що пропонують численні додаткові функції, сервіс зосереджений на спрощенні процесу використання. Це дозволяє користувачам отримувати лише найнеобхідніші функції, які відповідають базовим потребам у прослуховуванні музики.

Пошук та фільтрація музики в сервісі спрощені для зручності користувачів. Музику можна швидко знаходити за основними критеріями: жанр, виконавець, альбом. Такий підхід значно економить час і робить процес пошуку простим та інтуїтивно зрозумілим, без перевантаження зайвими опціями.

Щодо монетизації, сервіс працює на основі єдиної платної підписки, що є доступною завдяки спрощеній структурі функцій. Така модель дозволяє уникнути складних тарифних планів або додаткових платежів за додаткові можливості, характерні для великих платформ. Мета полягає у створенні максимально дешевої підписки при збереженні високої якості та

ефективності функціоналу. Користувачі отримують доступ до всіх основних функцій для пошуку та прослуховування музики за фіксовану низьку плату, що робить сервіс доступним для широкої аудиторії.

Цей підхід дозволяє створити простий та доступний музичний сервіс, орієнтуючись на потреби користувачів, які шукають ефективність та зручність без зайвих витрат на додаткові функції чи підписки. Сервіс стане ідеальним рішенням для тих, хто хоче отримати швидкий доступ до музики за мінімальну плату, без необхідності адаптуватися до великої кількості непотрібних опцій.

2 ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ ТА ІНСТРУМЕНТИ ДЛЯ РОЗРОБКИ ВЕБЗАСТОСУНКУ

Під час розробки сучасних вебзастосунків важливо зробити правильний вибір технологій, які забезпечать ефективність, масштабованість, безпеку та зручність використання. У цьому контексті було обрано технології для серверної та клієнтської частини, бази даних та розгортання, які добре поєднуються між собою, що дозволяє створювати стабільні та зручні додатки.

2.1 Серверна частина та база даних

2.1.1 Мова програмування C#

Мова програмування C# [1] є основою серверної частини проєкту завдяки своїй продуктивності, строгій типізації та широким можливостям для розробки високонавантажених застосунків. Вона була розроблена компанією Microsoft та постійно вдосконалюється, зберігаючи високу популярність серед розробників корпоративного та вебпрограмного забезпечення.

Однією з ключових переваг C# є підтримка об’єктно-орієнтованого програмування (ООП), що дозволяє структурувати код, підвищуючи його читабельність та підтримуваність. Також C# містить механізми для функціонального програмування, що спрощує роботу з колекціями та багатопотоковістю. Завдяки вбудованій підтримці асинхронного програмування, C# забезпечує ефективну роботу із запитами до бази даних, API та іншими сервісами, мінімізуючи блокування потоків. Це особливо важливо для масштабованих вебзастосунків, що обробляють велику кількість одночасних підключень. Вибір мови C# обумовлений її надійністю, великою спільнотою розробників та активною підтримкою з боку Microsoft, що гарантує актуальність та безпеку застосунку.

2.1.2 Фреймворк ASP.NET Core

Серверна частина вебзастосунку побудована на основі ASP.NET Core – сучасного високопродуктивного фреймворку від Microsoft для створення вебсервісів та API. Ця технологія є наступником класичного ASP.NET, має значні переваги, зокрема підвищену продуктивність, кросплатформеність та модульну архітектуру. Однією з основних особливостей ASP.NET Core є підтримка middleware-компонентів – окремих модулів, які обробляють HTTP-запити та відповіді. Це дозволяє легко додавати функціональність, таку як логування, автентифікацію, кешування або стиснення трафіку, без змін у базовій логіці застосунку.

ASP.NET Core надає ефективні механізми безпеки, включаючи аутентифікацію за допомогою JWT [2], що забезпечує захищений доступ до API без необхідності зберігати сесії на сервері. Завдяки мінімальному споживанню ресурсів та можливості роботи в мікросервісній архітектурі, ASP.NET Core є оптимальним вибором для побудови сучасного, продуктивного та безпечного вебзастосунку.

2.1.3 ORM-фреймворк Entity Framework Core

Для зручної роботи з БД використовується Entity Framework Core – ORM-бібліотека, яка спрощує доступ до даних у .NET-застосунках. Вона дозволяє писати код на C# не турбуючись про формування SQL-запитів вручну, що помітно прискорює розробку та мінімізує ризик помилок. EF Core підтримує кілька типів БД, автоматично відстежує зміни в об’єктах перед їх збереженням і надає інструменти для зручного оновлення схеми БД без втрати даних. Завдяки цьому, розробники можуть швидко вносити зміни в структуру проєкту, не витрачаючи час на складну міграцію даних.

Бібліотека чудово інтегрується .NET-екосистемою, що дозволяє створювати гнучкі та масштабовані рішення. Крім того, підтримка LINQ-

запитів робить роботу з даними максимально наближеною до об’єктно-орієнтованого стилю програмування.

2.1.4 Контейнеризація та Docker

Для ізоляції середовища виконання вебзастосунку використовується Docker – платформа для контейнеризації, яка дозволяє пакувати застосунок разом з усіма його залежностями в єдиний образ. Завдяки використанню Docker, застосунок може працювати стабільно на будь-якому сервері незалежно від налаштувань ОС. Це спрощує процес розгортання, усуває проблеми сумісності та забезпечує швидке масштабування.

Контейнери дозволяють легко оновлювати застосунок без ризику порушення роботи сервісу. Використання Docker Compose дає змогу автоматизувати розгортання декількох служб, наприклад, вебсервера, бази даних та кешуючих механізмів, об'єднуючи їх у спільне середовище. Також контейнеризація спрощує інтеграцію з CI/CD-процесами [3], оскільки образи Docker можна легко розгортати на будь-якому хмарному сервері або в Kubernetes-кластерах [4] для забезпечення високої доступності та навантажувальної стійкості.

2.1.5 Серіалізація даних та MessagePack

Для ефективного обміну даними між клієнтом та сервером застосовується MessagePack – високопродуктивний бінарний формат серіалізації, який є компактнішою та швидшою альтернативою JSON. Його головною перевагою є зменшений розмір даних, які передаються, що особливо важливо при роботі з великими обсягами інформації або в умовах обмеженої пропускної здатності мережі. Завдяки бінарному представленню MessagePack пришвидшує процес серіалізації та десеріалізації, що позитивно впливає на продуктивність вебзастосунку. Він також підтримує широкий

спектр типів даних, зберігаючи структуру об'єктів без втрати інформації. Це робить його зручним вибором для високонавантажених систем, які потребують швидкої та ефективної передачі даних у реальному часі.

2.1.6 Обмін повідомленнями та RabbitMQ

Для реалізації асинхронної взаємодії між мікросервісами використовується RabbitMQ – надійний брокер повідомлень, який дозволяє організувати ефективний обмін даними за допомогою черг. Його використання допомагає розвантажити сервери, покращити продуктивність системи та забезпечити стабільність виконання завдань навіть при високих навантаженнях. RabbitMQ підтримує різні моделі маршрутизації повідомлень, дозволяючи вибудовувати гнучку логіку обміну даними між сервісами. Він гарантує доставку повідомлень навіть у разі збоїв, що особливо важливо для критично важливих операцій. Інтеграція RabbitMQ у вебзастосунок підвищує масштабованість системи, оскільки мікросервіси можуть обмінюватися даними незалежно один від одного. Це дозволяє уникнути блокувань та забезпечити ефективну обробку запитів, зменшуючи затримки в роботі всього застосунку.

2.1.7 Взаємодія сервісів через RabbitMQ RPC

Для реалізації взаємодії між мікросервісами використовується RabbitMQ RPC [5], який дозволяє викликати віддалені процедури через механізм асинхронного обміну повідомленнями. Це забезпечує ефективну комунікацію між сервісами без необхідності прямого з’єднання між ними. Використання RPC через RabbitMQ дає змогу обробляти запити паралельно, розвантажуючи основні сервіси та підвищуючи швидкість виконання операцій. Завдяки чергам запитів та відповідей система отримує гнучкість у маршрутизації викликів, що зменшує затримки та покращує загальну

продуктивність. Такий підхід сприяє підвищенню відмовостійкості та масштабованості, оскільки мікросервіси можуть взаємодіяти незалежно один від одного. У разі збою одного з сервісів його запити залишаються в черзі, що дозволяє обробити їх після відновлення роботи, забезпечуючи надійність та стабільність системи.

2.1.8 Мова запитів SQL

SQL – це стандартна мова запитів, призначена для управління реляційними базами даних. Вона дозволяє виконувати CRUD-операції (створення, читання, оновлення та видалення даних), будувати складні вибірки, обробляти агреговані дані, а також виконувати аналітичні запити.

Мова SQL має вбудовані механізми оптимізації, такі як індексація, кешування, транзакції та паралельне виконання запитів, що забезпечує високу продуктивність та швидку обробку даних навіть у великих системах. Завдяки стандартизації SQL широко підтримується популярними СУБД, такими як PostgreSQL, MySQL, Microsoft SQL Server.

SQL використовується для ефективного управління базою даних вебзастосунку, забезпечуючи швидку обробку запитів та збереження цілісності даних. Завдяки підтримці складних операцій, таких як JOIN, GROUP BY, HAVING, WINDOW FUNCTIONS, можливо реалізовувати складні логічні вибірки та аналізувати великі обсяги інформації. Це дозволяє створювати гнучкі та масштабовані системи, що легко адаптуються до зростаючих навантажень.

2.1.9 Реляційна база даних PostgreSQL

PostgreSQL – це потужна об’єктно-реляційна система управління базами даних (СУБД) з відкритим вихідним кодом, яка відома своєю надійністю, гнучкістю та високою продуктивністю. Вона підтримує SQL-

стандарт та розширені можливості, такі як робота з JSON, складні транзакції, тригери, матеріалізовані представлення, збережені процедури та реплікація.

Завдяки механізмам MVCC [6] PostgreSQL ефективно керує конкурентним доступом до даних, забезпечуючи високу швидкодію без блокування читачів та записувачів. Це робить її оптимальним вибором для високонавантажених вебзастосунків, де важлива швидка обробка великої кількості запитів одночасно.

PostgreSQL підтримує масштабованість, що дозволяє легко керувати великими обсягами інформації та забезпечувати горизонтальне та вертикальне масштабування. Вбудовані механізми реплікації та резервного копіювання допомагають підвищити відмовостійкість і захистити дані від втрати. Завдяки своїм потужним інструментам та розширенням, таким як PostGIS (робота з геоданими), PostgreSQL підходить для складних систем, що потребують аналітики, роботи з великими масивами даних та гнучкого налаштування під специфічні задачі.

2.1.10 Кешування та зберігання тимчасових даних у Redis

Redis – це високопродуктивне розподілене сховище ключ-значення, яке використовується для кешування даних, обробки швидких запитів та зберігання тимчасової інформації. Його основною перевагою є операції в пам’яті (in-memory), що забезпечують надзвичайно швидкий доступ до даних порівняно з традиційними реляційними базами.

Однією з ключових особливостей Redis є підтримка різноманітних структур даних, включаючи рядки, списки, множини, хеші, відсортовані множини та інші, що дає можливість ефективно працювати з широким спектром задач. Redis підтримує механізми Time-To-Live, які дозволяють автоматично видаляти застарілі дані після закінчення заданого часу.

Завдяки реплікації (master-slave replication) Redis забезпечує відмовостійкість та балансування навантаження, а підтримка кластеризації

дозволяє горизонтально масштабувати систему, що є критично важливим для високонавантажених сервісів. Окрім кешування, Redis часто використовується для зберігання сесій користувачів, управління чергами завдань, обміну повідомленнями між сервісами та обробки аналітичних даних у реальному часі. Його низька затримка та висока продуктивність роблять його ідеальним вибором для вебзастосунків, де важлива швидка обробка запитів та мінімізація навантаження на основну базу даних.

2.1.11 Авторизація на основі JWT

JWT – це популярний метод аутентифікації та авторизації у вебзастосунках, який дозволяє передавати інформацію між клієнтом та сервером у вигляді компактного та захищеного токена. На відміну від традиційного зберігання сесій на сервері, JWT використовує безстанний підхід, що спрощує масштабування системи та підвищує її продуктивність.

Токен JWT складається з трьох частин: заголовка (header), корисного навантаження (payload) та підпису (signature). У заголовку міститься інформація про тип токена та алгоритм шифрування, у корисному навантаженні – закодовані дані про користувача та його права доступу, а підпис забезпечує цілісність токена та запобігає його підробці.

JWT працює за принципом видачі токена після успішної аутентифікації: сервер створює токен, підписує його секретним ключем і передає клієнту, який додає його до заголовка кожного наступного запиту. Це дозволяє серверу перевіряти автентичність користувача без необхідності звернення до бази даних під час кожної взаємодії, що суттєво покращує продуктивність та швидкість обробки запитів. JWT підтримує термін дії (expiration time), після закінчення якого токен стає недійсним, а також можливість освіження токена (refresh token), що забезпечує баланс між безпекою та зручністю використання. Завдяки своїм перевагам, JWT є оптимальним рішенням для масштабованих вебзастосунків, мікросервісних

архітектур та мобільних застосунків, де важлива автономність клієнтів та швидкий обмін даними без зайвих серверних запитів.

2.2 Клієнтська частина

2.2.1 Бібліотека React для розробки інтерфейсу користувача

React – це популярна JavaScript-бібліотека для створення динамічних інтерфейсів користувача. Вона розроблена Facebook (тепер Meta) та використовується для розробки швидких та інтерактивних вебзастосунків. Основною особливістю React є її компонентний підхід, що дозволяє розбивати інтерфейс на незалежні частини, які можливо багаторазово використовувати. Це сприяє зручності розробки, підтримки та масштабування застосунку.

Однією з головних переваг React є використання віртуального DOM [7], який значно прискорює оновлення інтерфейсу. Замість того, щоб змінювати всю сторінку при оновленні даних, React порівнює попередній та новий стан компонентів та вносить зміни лише в ті частини DOM, які дійсно змінилися. Це забезпечує високу продуктивність навіть у складних застосунках із великою кількістю динамічних елементів.

React також підтримує універсальну (ізоморфну) розробку, що дозволяє рендерити інтерфейс як на клієнті, так і на сервері, покращуючи швидкість завантаження сторінок та SEO-оптимізацію. Завдяки React Hooks стало можливим спрощене керування станом компонентів без використання класів, що робить код більш читабельним та зручним у супроводі. Ця бібліотека ідеально підходить для створення SPA [8], де швидкість взаємодії з користувачем відіграє важливу роль. React широко використовується в сучасних вебзастосунках завдяки своїй гнучкості, продуктивності та активній спільноті розробників.

2.2.2 Керування станом застосунку за допомогою Zustand

Zustand – це легка, продуктивна та гнучка бібліотека для керування станом у React-застосунках. Вона була створена як альтернатива громіздким рішенням на кшталт Redux, забезпечуючи простий API та мінімальні накладні витрати. На відміну від Redux, який вимагає створення редукторів, дій та мідлварів, Zustand дозволяє визначати глобальний стан усього в кількох рядках коду, використовуючи просту функцію створення сховища.

Бібліотека підтримує реактивний підхід, що досягається завдяки оптимізованому селекторному механізму, який дозволяє підписуватися лише на окремі частини стану. Таким чином, лише ті компоненти, які дійсно залежать від змінених даних, будуть перерендерені, що значно підвищує продуктивність застосунку. Ще однією ключовою особливістю Zustand є підтримка середовища без React. Завдяки цьому його можна використовувати для керування станом у мікросервісах або навіть у Node.js [9]. Крім того, бібліотека має вбудовану підтримку асинхронних запитів, що робить її зручною для роботи з API без потреби в додаткових мідлварах.

Простота, швидкодія та мінімалістичний API роблять Zustand ідеальним вибором для керування станом у React-застосунках будь-якої складності, особливо коли необхідно зменшити кількість зайвого коду та підвищити ефективність оновлення компонентів.

2.2.3 Мова програмування JavaScript та синтаксис JSX

JavaScript – це високорівнева, динамічна, прототипно-орієнтована мова програмування, яка є основним інструментом для створення інтерактивних вебзастосунків. Завдяки своїй гнучкості та широкій підтримці у браузерах, JavaScript став стандартом для розробки клієнтських інтерфейсів, дозволяючи створювати динамічні сторінки, обробляти події користувача та взаємодіяти з сервером без необхідності перезавантаження сторінки.

Мова підтримує асинхронне програмування, що робить її ідеальною для роботи з API, вебсокетами та іншими мережними запитами. Асинхронність реалізована за допомогою callback-функцій, промісів та async/await, що дозволяє ефективно керувати виконанням завдань без блокування головного потоку. Для опису структури інтерфейсу користувача в React застосовується JSX – розширення синтаксису JavaScript, яке дозволяє писати HTML-подібний код безпосередньо у файлах JS. JSX спрощує створення компонентів, роблячи код більш читабельним та декларативним. Відмінність JSX від звичайного HTML полягає в тому, що він компілюється у JavaScript-функції, які створюють віртуальний DOM, оптимізуючи рендеринг компонентів. Поєднання гнучкості JavaScript та зручності JSX дозволяє ефективно розробляти сучасні вебзастосунки, забезпечуючи швидку розробку, просту інтеграцію та високу продуктивність інтерфейсу.

2.2.4 Бібліотека компонентів Joy UI

Joy UI – це сучасна бібліотека інтерфейсних компонентів, яка була розроблена командою Material-UI та орієнтована на створення естетично привабливих, доступних та адаптивних інтерфейсів. Вона забезпечує гнучкість у налаштуванні стилів, підтримує CSS-змінні та дозволяє легко змінювати зовнішній вигляд компонентів завдяки системі темізації.

В основі Joy UI лежить концепція гнучкого дизайну, що дозволяє розробникам створювати сучасні UI-рішення без необхідності ручного написання великої кількості стилів. Бібліотека містить набір готових UI-компонентів – кнопки, форми, модальні вікна, списки та інші інтерактивні елементи, які легко кастомізуються та інтегруються у React-застосунки. Компоненти Joy UI оптимізовані для високої продуктивності, що зменшує час рендерингу та покращує досвід користувачів. Бібліотека підтримує темну та світлу тему, а також забезпечує адаптивність, що дозволяє створювати інтерфейси, які коректно відображаються на будь-яких пристроях.

Завдяки простоті використання, розширюваності та відповідності сучасним стандартам дизайну, Joy UI є чудовим вибором для швидкої розробки зручних та стильних вебзастосунків.

2.2.5 Використання HTML та CSS у вебзастосунку

HTML та CSS – це основні технології, які використовуються для створення та стилізації вебінтерфейсів. HTML відповідає за структуру сторінки, визначаючи елементи, такі як заголовки, текст, зображення, форми та кнопки, тоді як CSS контролює їхній зовнішній вигляд, включаючи кольори, шрифти, розташування та адаптивність. Сучасний CSS включає гнучкі інструменти для стилізації, такі як Flexbox та Grid, які спрощують створення адаптивних макетів без використання складних JavaScript-рішень. Додатково підтримуються CSS-змінні, які дозволяють керувати кольорами та шрифтами, що значно полегшує подальшу підтримку стилів.

У вебзастосунку HTML використовується для створення логічно організованої структури сторінок, що забезпечує зрозумілу семантику та покращує доступність для користувачів та пошукових систем. CSS застосовується для створення адаптивного дизайну, який коректно відображається на мобільних пристроях, планшетах і десктопах. Завдяки поєднанню HTML та CSS вебзастосунок отримує чітку структуру, привабливий вигляд та хорошу продуктивність, а також можливість швидкого внесення змін без складної логіки на стороні клієнта.

2.3 Хмарні технології AWS

2.3.1 Використання AWS EC2 для розгортання серверної частини

AWS EC2 – це масштабований сервіс віртуальних машин, що використовується для запуску серверної частини вебзастосунку, який надає

гнучкі можливості вибору апаратних ресурсів, операційної системи та рівня безпеки, що дозволяє налаштовувати інфраструктуру відповідно до вимог проєкту. Основні переваги EC2 – автоматичне масштабування, інтеграція з іншими сервісами AWS та гнучке керування доступом через IAM. Групи безпеки та правила брандмауера допомагають обмежувати небажаний трафік, забезпечуючи захищене середовище виконання серверних застосунків.

2.3.2 Використання AWS Route 53 для управління доменними іменами

Для керування DNS-записами та маршрутизації трафіку використовується AWS Route 53 – глобально-розподілений та масштабований DNS-сервіс. Він забезпечує швидкий та надійний розподіл запитів між серверами, підтримує балансування навантаження, а також легко інтегрується з іншими сервісами AWS, такими як EC2, S3 та CloudFront. Route 53 дозволяє налаштовувати перевірку працездатності серверів у реальному часі, що дає змогу автоматично перенаправляти трафік у разі збою одного з інстансів, підвищуючи загальну відмовостійкість системи. Сервіс без проблем масштабується під потреби проєкту, що робить його ефективним як для стартапів, так і для великих високонавантажених рішень з критичними вимогами до доступності. Завдяки зручному вебінтерфейсу, гнучким політикам маршрутизації та підтримці сучасних протоколів, Route 53 надає розробникам потужний інструмент для централізованого управління доменами та стабільної роботи вебресурсів у будь-яких умовах.

2.3.3 Зберігання файлів у AWS S3

Для збереження статичних файлів, таких як зображення, відео та інші ресурси, використовується AWS S3 – високодоступне та надійне хмарне сховище, яке забезпечує гнучке керування доступом, підтримує шифрування даних, автоматичне резервне копіювання та журналювання дій. У межах

вебзастосунку S3 виконує роль основного репозиторію для зберігання контенту користувача, організації доступу до ресурсів через унікальні URL-адреси, а також оптимізації доставки медіафайлів у поєднанні з CDN-сервісом AWS CloudFront. Завдяки своїй масштабованості та стабільності, сервіс ідеально підходить як для малих, так і для великих проєктів, де необхідна швидка та безпечна передача файлів у глобальному масштабі. Тісна інтеграція з іншими сервісами AWS дозволяє автоматизувати обробку, аналіз та керування даними, підвищуючи ефективність роботи всієї системи.

2.3.4 Оптимізація контенту через AWS CloudFront

AWS CloudFront – це сервіс доставки контенту (CDN) [10], який значно знижує затримку при завантаженні файлів завдяки розгалуженій глобальній мережі серверів кешування. CloudFront автоматично розподіляє статичний і динамічний контент між найближчими до користувачів edge-серверами, що забезпечує високу швидкість завантаження вебсторінок, зображень, відео та інших ресурсів незалежно від географічного розташування. Сервіс також підтримує автоматичне стиснення даних, що зменшує обсяг переданого трафіку та покращує продуктивність. Крім цього, CloudFront інтегрується з AWS Shield та AWS WAF, надаючи вбудований захист від DDoS-атак і додаткові засоби безпеки, що є критично важливим для сучасних вебзастосунків та високонавантажених систем. Завдяки масштабованості, високій доступності та простій інтеграції з іншими сервісами AWS, CloudFront є надійним рішенням для прискорення доставки контенту та забезпечення стабільної роботи проєктів будь-якого масштабу.

2.3.5 Обробка медіафайлів через AWS MediaConvert

AWS MediaConvert використовується для автоматичного перетворення аудіо- та відеофайлів у сучасні потокові формати, зокрема HLS [11], MPEG-

DASH та інші. Завдяки цьому контент можна адаптувати під різні типи пристроїв та швидкість інтернет-з’єднання користувача, що суттєво покращує якість перегляду та забезпечує безперебійне відтворення медіа навіть в умовах нестабільного зв’язку. Сервіс підтримує розширені функції, такі як автоматичне масштабування якості відео, додавання субтитрів, DRM-захист та створення попереднього перегляду. MediaConvert легко інтегрується з іншими компонентами AWS-інфраструктури, дозволяючи автоматизувати обробку відео у хмарі без потреби в окремому медіасервері чи складному налаштуванні інструментів для транскодування.

2.3.6 Автоматизація процесів за допомогою AWS Lambda

AWS Lambda – це безсерверний обчислювальний сервіс, що дозволяє запускати функції у відповідь на події без потреби в управлінні чи масштабуванні інфраструктури. Lambda використовується для автоматизованої обробки завантажених файлів, генерації прев’ю зображень, фільтрації даних, взаємодії з іншими сервісами AWS (наприклад, S3, DynamoDB, SNS), а також для побудови мікросервісної архітектури. Завдяки моделі "pay-per-use", функції виконуються лише тоді, коли це потрібно, що суттєво знижує витрати на обчислення та дозволяє уникнути простоїв. Крім того, Lambda підтримує кілька мов програмування, має високу масштабованість та легко інтегрується в CI/CD-процеси, що робить її ефективним інструментом для створення швидких, гнучких та економічно вигідних серверних рішень.

2.3.7 Надсилання електронних листів через AWS SES

AWS SES [12] використовується для автоматичної відправки транзакційних електронних листів, таких як підтвердження реєстрації, відновлення пароля, сповіщення про зміну налаштувань акаунта та інші

важливі повідомлення. SES забезпечує високу швидкість та надійність доставки, гарантуючи мінімальні затримки при відправці великих обсягів листів. Крім того, сервіс підтримує налаштування SPF, DKIM та DMARC, що дозволяє захистити репутацію домену та значно знижує ймовірність того, що листи потраплять у спам. Завдяки гнучким можливостям конфігурації та інтеграції з іншими сервісами AWS, SES є ідеальним рішенням для масштабованих та безпечних рішень по відправці електронної пошти в бізнес-застосунках.

2.3.8 Управління доступом через AWS IAM

AWS IAM використовується для налаштування прав доступу та управління безпекою хмарних ресурсів AWS. Завдяки IAM можна створювати ролі користувачів, групи, політики доступу та призначати їх для конкретних сервісів і ресурсів, що дозволяє ефективно контролювати та аудіювати всі операції в інфраструктурі AWS. IAM підтримує мультифакторну аутентифікацію (MFA), що додає додатковий рівень захисту для критичних облікових записів та ресурсів. Інтеграція з іншими сервісами AWS дозволяє централізовано керувати доступом до різноманітних хмарних ресурсів і забезпечувати відповідність до стандартів безпеки та вимог на всіх етапах життєвого циклу застосунків. За допомогою гнучких політик та можливості налаштування детальних прав доступу, IAM стає незамінним інструментом для побудови надійних, безпечних та масштабованих рішень у хмарі, допомагаючи знижувати ризики й підвищувати ефективність управління.

2.4 Проксіювання та балансування навантаження у вебзастосунку

Traefik – це динамічний проксі-сервер, призначений для маршрутизації трафіку та балансування навантаження. Він автоматично виявляє нові сервіси

та оновлює маршрути, що дозволяє значно зменшити потребу в ручному налаштуванні. Завдяки підтримці автоматичної SSL-термінації, Traefik спрощує процес розгортання та управління сертифікатами безпеки, що підвищує рівень захисту застосунків. Інтеграція з популярними оркестраторами контейнерів, такими як Docker та Kubernetes, дозволяє ефективно керувати мікросервісними архітектурами, автоматизуючи налаштування маршрутизації для нових або змінених сервісів без зупинки системи. Крім того, Traefik підтримує моніторинг та логування, що дозволяє оперативно відслідковувати трафік та діагностувати потенційні проблеми з продуктивністю, забезпечуючи надійне та безперебійне функціонування інфраструктури.

2.5 Середовище розробки та конфігурація сервера

Розглянуто опис системи контролю версій, середовища розробки, репозиторіїв коду та контейнерів, а також систему управління завданнями, яка забезпечує ефективне планування та контроль за виконанням проєкту. Окрему увагу приділено параметрам конфігурації серверного середовища, де детально описані характеристики AWS EC2-інстансу, операційна система, встановлене програмне забезпечення та інші налаштування, що забезпечують стабільну та ефективну роботу застосунку. Проведений аналіз дозволяє отримати чітке уявлення про основні інструменти та ресурси, які можуть бути використано на різних етапах розробки, що забезпечує зручне управління процесом та високий рівень автоматизації.

2.5.1 Управління процесом розробки

Для організації процесу розробки використовується Jira, яка дозволяє чітко розділити сервіси на окремі проєкти. Це створює зручну структуру, що дає змогу ефективно управляти задачами як на рівні кожного окремого

сервісу, так і на рівні всього проєкту. Такий підхід полегшує контроль за виконанням задач та дозволяє своєчасно стежити за прогресом роботи, виявляти вузькі місця та оптимізувати робочі процеси. В Jira реалізовано різні робочі процеси для кожного сервісу, що дозволяє ефективно розподіляти ресурси та гарантувати виконання задач у встановлені терміни. Окремо ведеться управління задачами, пов'язаними з налаштуванням, моніторингом та оптимізацією інфраструктури AWS, що дає змогу підтримувати чітке уявлення про стан інфраструктури та забезпечує стабільну роботу всіх сервісів. Завдяки гнучким можливостям інтеграції з іншими інструментами, такими як Bitbucket для контролю версій та автоматизації процесів, Jira є потужним інструментом для забезпечення високої ефективності та прозорості у розробці. Крім того, автоматизація процесів у Jira дозволяє значно зменшити час на рутинні операції, підвищуючи швидкість розробки та знижуючи ризик помилок.

2.5.2 Система контролю версій та репозиторії коду

Для зберігання вихідного коду використовується BitBucket, де кожен проєкт має власний репозиторій, що спрощує контроль за виконанням задач та забезпечує зручну інтеграцію з системою управління проєктами. Кожна гілка репозиторію може бути прив'язана до конкретних задач у Jira, що дозволяє тримати всі зміни під контролем та забезпечує прозорість у процесі розробки, відслідковуючи кожний етап роботи. Однією з головних переваг BitBucket є підтримка приватних репозиторіїв у безкоштовному тарифному плані, що гарантує високий рівень безпеки, захищаючи конфіденційний код від витоків. Інтеграція з іншими інструментами, такими як Jira, дає можливість автоматизувати зв'язок між задачами та змінами в коді, що значно підвищує ефективність управління проєктами. Крім того, підтримка процесів CI/CD дозволяє автоматизувати тестування, збірку та деплой, що знижує ризики помилок, підвищує якість коду та забезпечує швидший

випуск нових версій продукту. Це робить розробку більш ефективною, зручнішою та менш схильною до затримок.

2.5.3 Контейнеризація та управління Docker-образами

Для зберігання та розповсюдження образів Docker використовується Docker Hub, що дозволяє зручно отримувати, зберігати та розгортати релізи сервісів на сервері. Завдяки інтеграції з Docker Hub, процес деплою стає швидким та простим, оскільки всі образи доступні для завантаження безпосередньо з хмарного сховища. Для управління контейнерами використовуються Portainer – потужний вебінтерфейс, який значно спрощує процес розгортання контейнерів, дозволяючи управляти ними без необхідності підключення до оболонки EC2-інстанса.

Portainer доступний через інтернет, що дозволяє ефективно управляти контейнерами з будь-якої точки світу, а також підтримує розподіл прав доступу, забезпечуючи контроль над тим, хто може керувати контейнерами. Вбудована система авторизації дозволяє налаштувати різні рівні доступу для різних користувачів, що підвищує безпеку та забезпечує правильне управління контейнерами в команді.

2.5.4 Інструменти розробки

Для написання та налагодження коду використовуються JetBrains Rider та WebStorm. Rider є потужним середовищем розробки, оптимізованим для роботи з .NET-проєктами, пропонуючи багатий набір інструментів для розробки, налагодження та тестування коду, що значно прискорює процес розробки. WebStorm використовується для фронтенд-розробки, надаючи зручні інструменти для роботи з JavaScript, TypeScript, HTML та CSS, а також підтримку найновіших фреймворків та бібліотек, що дозволяє швидко створювати сучасні вебзастосунки. Для роботи з базами даних застосовується

JetBrains DataGrip, який забезпечує зручний інтерфейс для керування таблицями, написання SQL-запитів та взаємодії з різними системами управління базами даних. Завдяки підтримці великої кількості СУБД, DataGrip є незамінним інструментом для ефективної роботи з даними, спрощуючи налаштування підключень та оптимізацію запитів, що дозволяє розробникам фокусуватися на досягненні результату.

2.5.5 Конфігурація серверного середовища

Для розгортання серверної частини застосунку використовується AWS EC2 інстанс типу t2.micro, що має наступні характеристики:

- 1 vCPU – для забезпечення базової обчислювальної потужності, достатньої для невеликих навантажень;
- 1 GiB RAM, з можливістю розширення до 2 GiB за рахунок активного використання файлу підкачки, що дозволяє ефективно управляти пам'яттю при високих навантаженнях;
- 16 GB SSD – для зберігання операційної системи, застосунків та мінімальних даних. Вибір SSD забезпечує високу швидкість доступу до даних і підвищує загальну ефективність системи;
- операційна система: Amazon Linux, оптимізована для роботи в середовищі AWS, що надає легку інтеграцію з іншими сервісами AWS та підтримує автоматичне оновлення безпеки.

Вибір конфігурації був обумовлений безкоштовним тарифним планом AWS, що дозволяє оптимізувати витрати на початкових етапах тестування застосунку в реальному інтернет-середовищі. Це дає змогу протестувати функціональність та продуктивність застосунку без значних фінансових витрат. Крім того, обраний інстанс дозволяє масштабувати ресурси за потреби, забезпечуючи гнучкість у розвитку проєкту та можливість переходу до більш потужних інстансів у майбутньому.

3 ВЗАЄМОДІЯ КОМПОНЕНТІВ ТА АРХІТЕКТУРА ВЕБЗАСТОСУНКУ

У сучасних вебзастосунках легкість, масштабованість та розширюваність архітектури є ключовими вимогами для забезпечення їх ефективної роботи та подальшого розвитку. Для досягнення цих цілей широко застосовуються мікросервісна архітектура, хмарні технології, а також сучасні фреймворки та бібліотеки для створення клієнтських застосунків. У даній кваліфікаційній роботі в якості фронтенд-частини використовується React, що дозволяє створювати динамічний та зручний інтерфейс користувача за принципом SPA.

У розділі описано основні компоненти системи, а саме: клієнтська частина, серверна частина та хмарна інфраструктура AWS. Для кожного з компонентів наведено його внутрішню організацію, обрано підходи до побудови, а також використано технології та патерни проєктування.

Окрему увагу приділено механізмам взаємодії між цими компонентами. Зокрема, розглянуто, як здійснюється обмін даними між Frontend та Backend, між Backend та хмарними сервісами AWS, а також прямі взаємодії Frontend із AWS для оптимізованої роботи з медіаконтентом.

Додатково в розділі подано опис архітектури клієнтської частини, де впроваджено адаптовану трьохшарову модель (3-layers) у поєднанні з принципами SRP та Feature-Sliced Design (FSD). Такий підхід дозволяє досягти високої модульності, спрощує підтримку коду та забезпечує його зручну масштабованість. Важливим елементом клієнтської архітектури є використання патерну State-Driven, що дозволяє централізовано оновлювати стан застосунку із бізнес-логіки.

3.1 Архітектура отримання та конвертації медіаконтенту у AWS

У процесі отримання та конвертації медіаконтенту у AWS користувач завантажує MP3 файл безпосередньо у хмарне сховище AWS S3 за

допомогою тимчасового підписаного посилання (Signed URL), яке попередньо генерується через Storage Service (рисунок 3.1). Цей механізм дозволяє здійснювати безпечне пряме завантаження великих файлів без необхідності прокидати сам контент через серверну частину застосунку, що значно оптимізує використання мережевих ресурсів та зменшує навантаження на Backend.

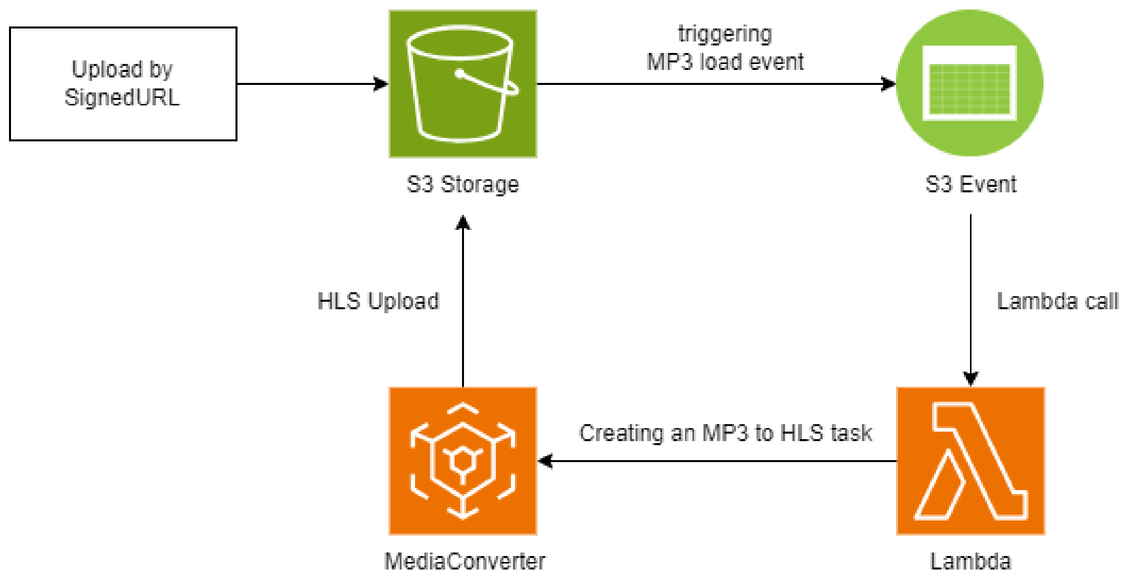

Рисунок 3.1 – Архітектура отримання та конвертації медіаконтенту у AWS

Після завершення завантаження MP3-файлу у S3, автоматично спрацьовує подія MP3 load event, яка запускає виконання AWS Lambda. Функція Lambda формує завдання для сервісу AWS MediaConvert, що виконує конвертацію аудіофайлу з формату MP3 у HLS (оптимізований для потокового відтворення у браузерах та мобільних застосунках). Готові HLS-сегменти завантажуються назад у AWS S3.

Після завершення запису HLS-сегментів у AWS S3, автоматично активується подія HLS Upload Event (рисунок 3.2), яка ініціює виконання додаткової Lambda-функції. Ця функція перевіряє наявність готового HLS-контенту у відповідній директорії S3 та, у разі успішної обробки, виконує видалення оригінального MP3-файлу, що забезпечує ефективне використання сховища та зменшення зайвого обсягу даних у S3. Таким чином,

представлена схема обробки медіаконтенту у хмарній інфраструктурі AWS дозволяє забезпечити повністю автоматизований та масштабований процес конвертації аудіофайлів у формат HLS, оптимізований для потокового відтворення.

Використання підписаних посилань (Signed URL) для завантаження та доступу до контенту гарантує безпеку та ефективність взаємодії між клієнтською частиною та хмарним сховищем, а автоматичне очищення оригінальних MP3-файлів сприяє раціональному використанню ресурсів S3.

Завдяки такій архітектурі, система готова до обробки великої кількості медіаконтенту та підтримує високі вимоги до продуктивності та масштабованості.

Реалізація автоматичного видалення оригінальних MP3-файлів після успішної конвертації дозволяє економити місце у хмарному сховищі, підвищити загальну ефективність використання ресурсів.

Такий підхід зменшує навантаження на інфраструктуру та мінімізує потребу в ручному адмініструванні процесів очищення, що особливо важливо у масштабованих багатокористувацьких системах.

У результаті система зберігає високу продуктивність при зростанні обсягів даних без необхідності постійного втручання адміністратора.

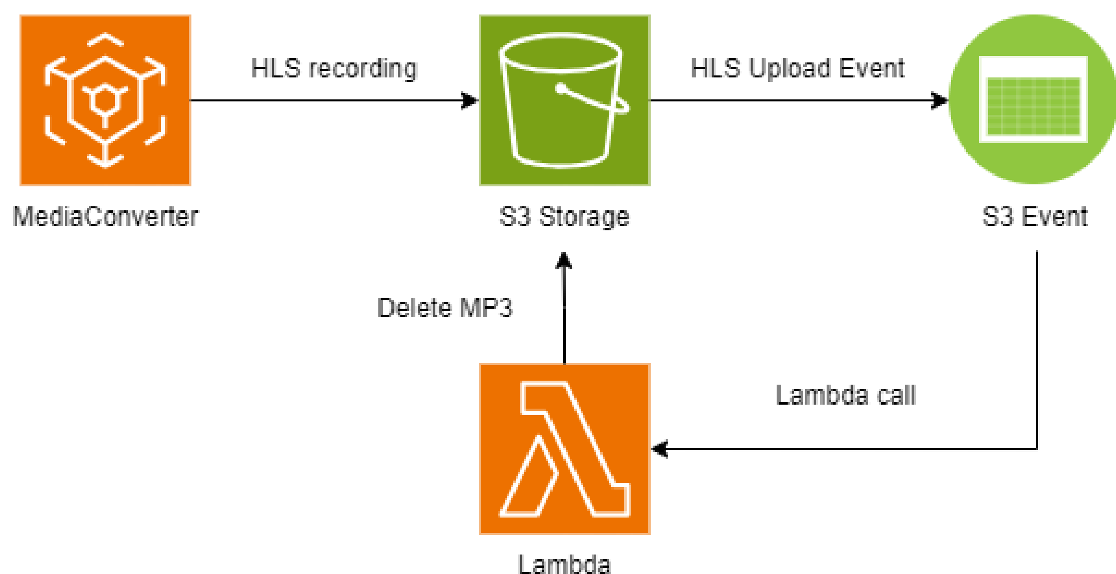

Рисунок 3.2 – Постобробка після конвертації: очищення оригінального MP3

3.2 Архітектура взаємодії Frontend із AWS для відтворення медіаконтенту

У процесі ініціації відтворення треку (рисунок 3.3), авторизований користувач натискає кнопку "Play" у клієнтському застосунку, що запускає ланцюг подій у Frontend-архітектурі, спрямованих на забезпечення безпечного доступу до медіаконтенту.

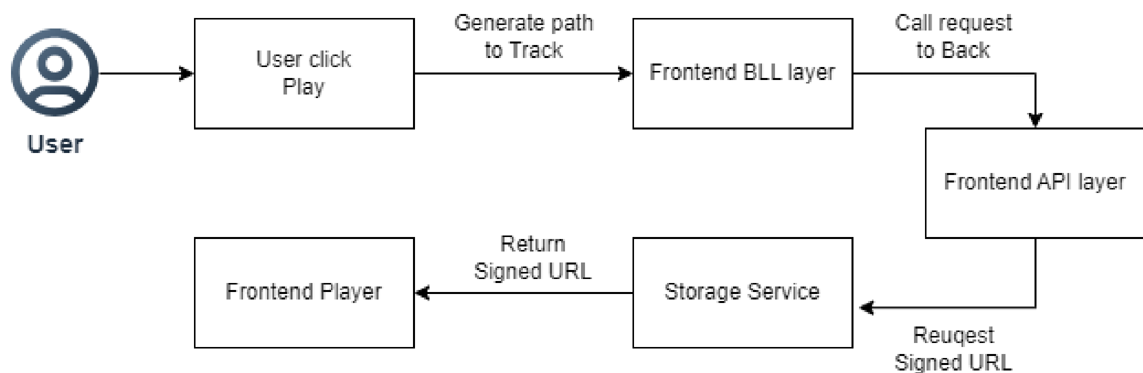

Рисунок 3.3 – Потік отримання Signed URL для відтворення медіаконтенту

На наступному етапі Business Logic Layer формує шлях до відповідного HLS-ресурсу на основі метаданих треку, збережених у внутрішньому стані застосунку. Формування цього шляху виконується безпосередньо у Frontend BLL для досягнення максимальної ізоляції бізнес-логіки від інфраструктурних залежностей. Це дозволяє забезпечити чітке розділення відповідальностей та дотримання принципу Single Responsibility Principle у архітектурі клієнтської частини.

Далі API Layer Frontend надсилає запит до Storage Service для отримання тимчасового підписаного посилання Signed URL, яке забезпечує доступ до контенту через AWS CloudFront. У відповідь Storage Service повертає Signed URL (рисунок 3.4), який передається у Frontend Player для подальшого використання. На цьому етапі Frontend Player вже має необхідний Signed URL та готовий до початку потокового відтворення.

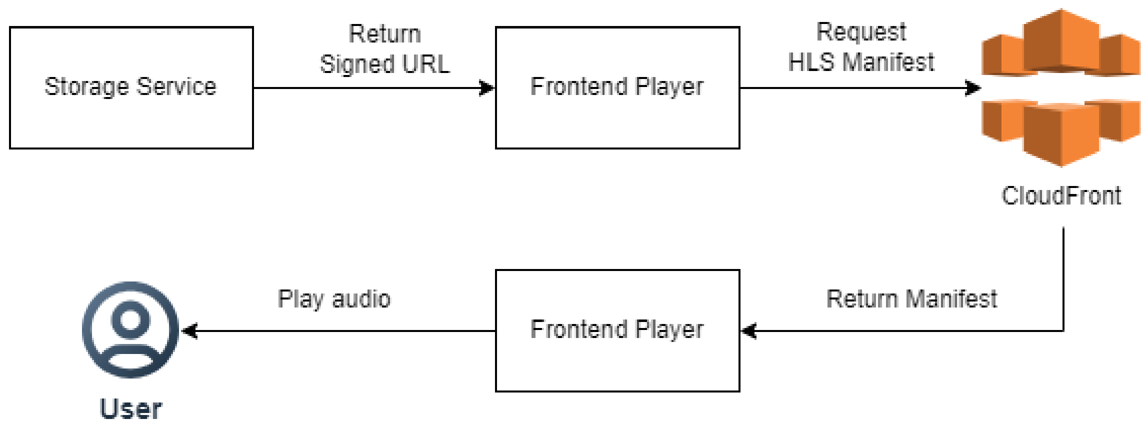

Рисунок 3.4 – Потік використання Signed URL для відтворення медіаконтенту

Після отримання Signed URL, Frontend Player, реалізований із використанням бібліотеки HLS.js, ініціює запит до AWS CloudFront для отримання HLS Manifest, який визначає структуру відтворення треку.

На цьому етапі CloudFront виступає як проміжний кешуючий рівень між клієнтським застосунком та основним сховищем контенту AWS S3. Завдяки цьому підходу значно зменшується затримка при відтворенні та оптимізується використання мережних ресурсів.

У випадку, якщо запитуваний HLS Manifest та пов’язані сегменти вже присутні у кеші CloudFront, плеєр отримує їх напряму із CDN, що забезпечує максимально швидкий старт відтворення та безперервний потік для кінцевого користувача.

Якщо ж необхідний контент відсутній у кеші, CloudFront автоматично звертається до AWS S3 для отримання відповідних HLS-сегментів. Після їх отримання CDN кешує ці сегменти локально та передає у Frontend Player для подальшого відтворення.

Процес відтворення здійснюється у режимі реального часу, де HLS.js Player динамічно підвантажує сегменти потоку відповідно до наявних мережних умов та можливостей клієнтського пристрою. Завдяки цьому підходу користувач отримує стабільний та якісний досвід потокового прослуховування музики.

Такий поділ процесу на два етапи (отримання Signed URL та використання Signed URL для потокового відтворення) дозволяє ефективно управляти доступом до медіаконтенту, забезпечуючи при цьому захищене, контрольоване та оптимізоване потокове відтворення за допомогою інфраструктури AWS CloudFront. Цей підхід також сприяє зменшенню навантаження на основне сховище AWS S3 завдяки використанню механізмів кешування на рівні CDN.

3.3 Взаємодія мікросервісів при отриманні Signed URL для відтворення медіаконтенту

Коли авторизований користувач із дійсним JWT-токеном відкриває вебзастосунок, автоматично ініціюється запит до Music Service (рисунок 3.5) для отримання актуального списку доступних треків.

У відповідь Music Service повертає список треків разом із необхідними метаданими, які включають інформацію для формування шляху до кожного треку у форматі: {artistId}/{albumId}/{trackId}/{trackId}.m3u8.

При натисканні користувачем кнопки "Play" на одному із треків, бізнес-логіка Frontend звертається до внутрішнього сховища стану (Zustand Store) для отримання метаданих обраного треку. На основі отриманих даних Business Logic Layer (BLL) формує шлях до відповідного HLS-ресурсу, який буде використано для подальшого запиту до Storage Service. Далі Frontend надсилає запит до Storage Service на отримання Signed URL для сформованого шляху треку.

На цьому етапі Storage Service перевіряє наявність дійсного JWT-токену у запиті та, у разі успішної авторизації, генерує тимчасове Signed URL, що надає безпечний доступ до HLS-контенту через AWS CloudFront. Отриманий Signed URL повертається у Frontend, де передається у плеєр для ініціації потокового відтворення.

Реалізована архітектура системи дозволяє використовувати

централізовану авторизацію: єдина точка видачі JWT-токенів спрощує керування сесіями та контролем доступу. Інші мікросервіси мають можливість самостійно виконувати перевірку авторизації на основі отриманого токену, без необхідності додаткових запитів до централізованої бази даних.

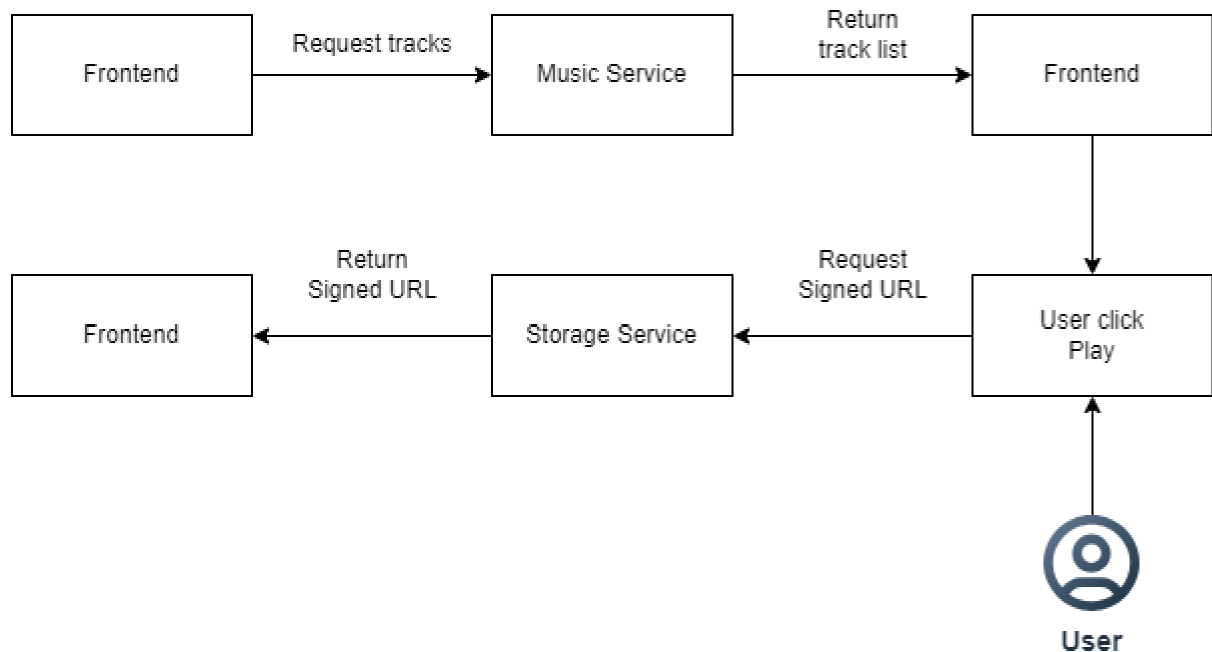

Рисунок 3.5 – Взаємодія мікросервісів при отриманні Signed URL для відтворення медіаконтенту

Такий підхід забезпечує високу незалежність мікросервісів, що суттєво покращує зручність розробки (Developer Experience, DX) – сервіси можуть запускатися, тестуватися та масштабуватися окремо, без потреби у розгортанні повного застосунку. Крім того, архітектура сприяє підвищенню відмовостійкості та захисту конфіденційної інформації: кожен сервіс зберігає лише ті секрети та налаштування, які необхідні йому безпосередньо для виконання своїх завдань.

Також забезпечено захищений та контрольований доступ до медіаконтенту за допомогою механізму Signed URL у поєднанні з авторизацією та кешуванням на рівні AWS CloudFront, що дозволяє досягти цього при мінімальних інфраструктурних витратах.

3.4 Архітектура мікросервісів системи Soundify та призначення сервісів

Система Soundify реалізована на основі мікросервісної архітектури (рисунок 3.6), що передбачає побудову системи з окремих сервісів, кожен із яких виконує чітко визначену бізнес-функцію. Взаємодія клієнтських застосунків із сервісами здійснюється через реверсивний проксі-сервер Traefik, який забезпечує маршрутизацію зовнішніх запитів та виконує SSL-термінацію. Взаємодія між мікросервісами у внутрішній мережі мінімізована: сервіси не виконують прямих викликів один до одного. Єдиним прикладом асинхронної взаємодії є взаємодія між IdentityCore та Notification Service за допомогою брокера повідомлень RabbitMQ, який забезпечує передачу подій (event-driven communication).

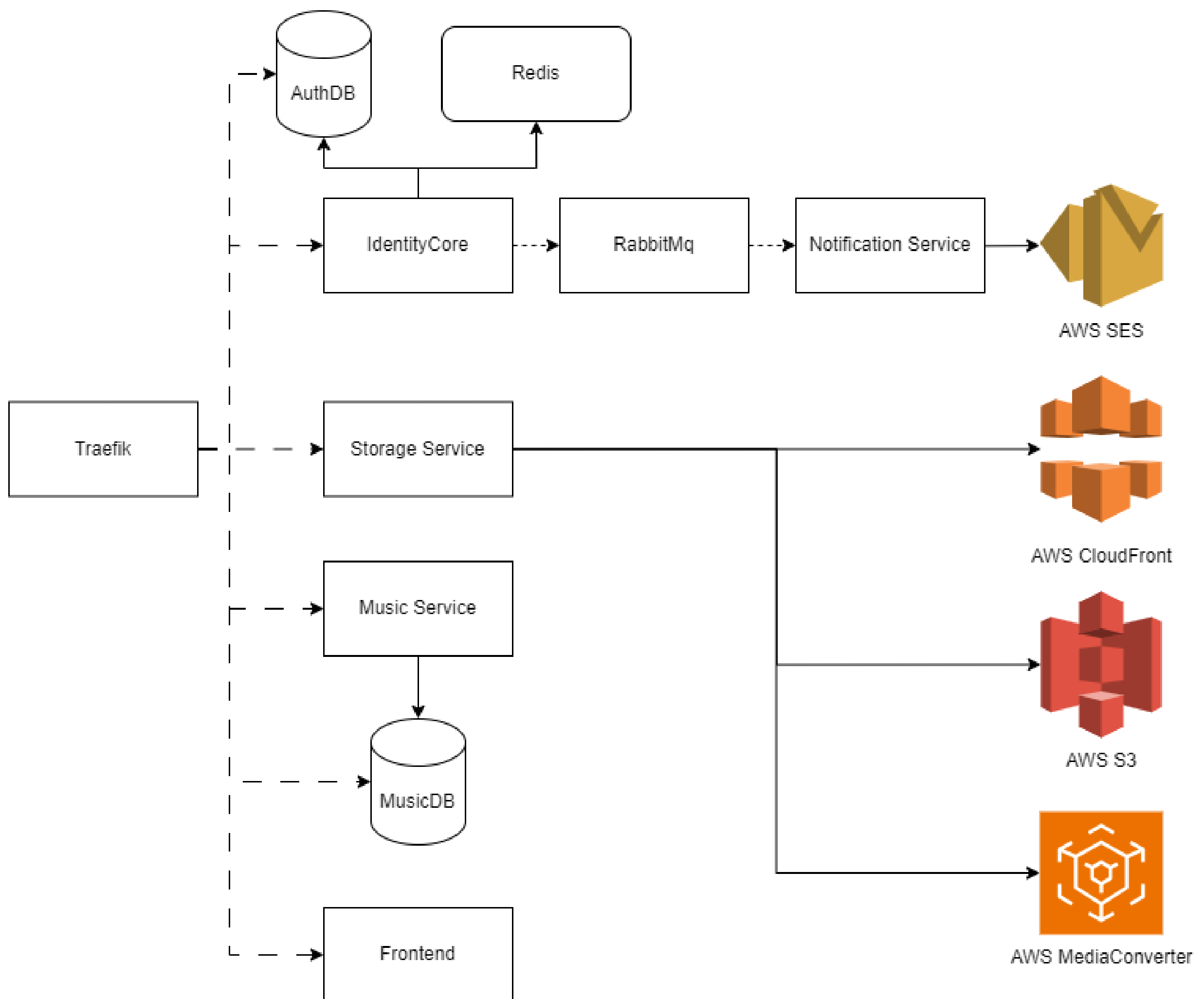

Рисунок 3.6 – Архітектура мікросервісів системи Soundify

Кожен мікросервіс розгортається у власному ізольованому середовищі, що спрощує процеси його оновлення, масштабування та моніторингу. Це також дозволяє забезпечити незалежне управління життєвим циклом сервісів, зокрема їх безпечне оновлення без простою всієї системи. Інфраструктура побудована таким чином, що окремі сервіси можуть розгортатися на різних вузлах кластеру, що підвищує загальну стійкість системи до збоїв та полегшує балансування навантаження.

Обрана архітектура також полегшує інтеграцію з зовнішніми сервісами та платформами. Завдяки чітко визначеним API та використанню стандартних протоколів взаємодії через RabbitMQ, система легко інтегрується з хмарною інфраструктурою AWS, що дає можливість ефективно використовувати такі сервіси, як S3, CloudFront, MediaConvert та SES, що спрощує розширення функціональності проєкту без необхідності суттєвої переробки існуючих компонентів. Такий підхід дозволяє досягти високої масштабованості, відмовостійкості та гнучкості розвитку системи.

Розглянемо більш докладно ключові компоненти мікросервісної архітектури системи Soundify, їх функціональне призначення, взаємозв’язки та внесок у загальну логіку роботи платформи.

AuthDB – база даних, яка відповідає за зберігання облікових записів користувачів, їх облікових даних (креденціалів), а також інформації про сесії.

MusicDB – база даних, яка зберігає метадані треків, альбомів, виконавців, а також інформацію про рейтинги, плейлисти та треки, позначені як улюблені.

Redis – кеш, що використовується для зберігання проміжних станів у процесах реєстрації користувачів, зміни пароля та електронної пошти.

RabbitMQ – брокер повідомлень, який підтримує чергу повідомлень для відправлення електронних листів користувачам через сервіс AWS SES.

Traefik – реверсивний проксі-сервер, що забезпечує можливість "гарячого" підключення нових сервісів, виконує SSL-термінацію та контролює мережевий трафік.

IdentityCore – мікросервіс авторизації, який видає JWT-токени після аутентифікації користувача. Відповідає за облікові записи користувачів, процеси реєстрації та входу. Під час реєстрації ініціює відправлення повідомлень через RabbitMQ, які потім обробляються Notification Service та доставляються користувачам на електронну пошту за допомогою AWS SES. Крім того, IdentityCore використовує Redis для зберігання стану користувача при скиданні пароля, зміні електронної пошти та підтвердженні реєстрації. Працює із базою даних AuthDB.

Music Service – мікросервіс для роботи з музичним контентом. Містить бізнес-логіку для управління виконавцями, альбомами, треками, плейлистами, улюбленими треками та рейтингами. Крім того, генерує токени для роботи з альбомами, виконавцями та треками у AWS S3 за посередництва Storage Service.

Storage Service – мікросервіс, що видає підписані посилання (Signed URL) та виконує операції CRUD над сервісами AWS (відповідно до архітектури системи). Основна його задача – це управління медіаконтентом у сховищі. Сервіс працює на основі токенів, отриманих із Music Service. Токени застосовуються при видаленні/завантаженні треків, обкладинок альбомів та зображень виконавців. Крім того він, надає API для ручного запуску MediaConvert Job, реалізує миттєву інвалідацію треків у AWS CloudFront за токенами видалення з Music Service, а також видає тимчасові підписані посилання для потокового відтворення музики через CDN.

Реалізована у роботі архітектура на основі мікросервісної моделі дозволяє досягти низького рівня зв’язаності між сервісами, що напряму сприяє підвищенню відмовостійкості та забезпечує легке горизонтальне масштабування за допомогою сучасних інструментів, таких як Kubernetes.

Розробка кожного сервісу може здійснюватися незалежно, навіть окремими командами розробників, що відкриває можливість оптимізації витрат на розробку та підтримку системи, що дозволяє залучати спеціалізовані команди з різних компаній для окремих компонентів

відповідно до їхнього функціонального призначення та рівня навантаження.

Крім того, глибока інтеграція з інфраструктурою AWS дала змогу відмовитися від необхідності створення власного рішення для стримінгового відтворення медіаконтенту. Це суттєво скорочує витрати на розробку, значно знижує вимоги до кваліфікації команди, а відповідно і вартість години роботи розробників.

Таким чином, в кваліфікаційній роботі було реалізовано не лише гнучку з технічної точки зору систему, а й ефективну з точки зору бізнес-управління архітектуру, яка спрощує процеси розвитку та експлуатації проєкту та забезпечує оптимальний баланс між якістю та витратами.

3.5 Архітектура клієнтської частини системи Soundify

Клієнтська частина системи Soundify була розроблена на основі кастомної архітектури, що поєднує принципи 3-layers моделі, Single Responsibility Principle (SRP) та ідеї Feature-Sliced Design (FSD), які оптимізовані під потреби проєкту.

При побудові архітектури клієнтської частини було використано класичний підхід 3-layers, який також реалізовано у Backend-частині системи. Під час аналізу Feature-Sliced Design (FSD) було встановлено, що його стандартна структура нерідко призводить до надмірного дублювання папок та повторення назв, що ускладнює навігацію та підтримку проєкту. З урахуванням цього було прийнято рішення оптимізувати архітектуру, зберігши ключові ідеї розподілу відповідальностей, але усунувши надлишкову структуру. У результаті була реалізована чиста модель, заснована на принципі єдиної відповідальності (Single Responsibility Principle, SRP): «Один шар – одна відповідальність», «Один Manager – одна сутність».

Технологічний стек клієнтської частини включає: React (побудова SPA), Zustand (централізоване управління станом), Material UI (MUI) (інтерфейс), а також повноцінно реалізовані DI-контейнер та Hosted Services.

При цьому логіка DI та HostedService була не адаптована, а повністю перенесена з .NET 8: код був переписаний з C# на TypeScript, повністю зберігаючи архітектурні принципи та життєвий цикл залежностей.

Ключовим елементом архітектури є патерн State-Driven, який забезпечує реактивну взаємодію між шарами клієнтської частини. Центральним ядром додатку виступає Zustand Store, де будь-які зміни стану ініціюються виключно через Managers, API Layer або Hosted Services. При цьому UI виконує тільки реактивне відображення стану, що забезпечує високу стабільність та передбачуваність поведінки інтерфейсу.

Особливу увагу в архітектурі приділено чіткому вертикальному розподілу шарів та відокремленню бізнес-логіки від презентаційного шару. Завдяки такому підходу було досягнуто гнучкість структури, простоту розширення та можливість масштабування команди розробки без ризику ускладнення архітектури.

3.5.1 Загальна архітектура клієнтської частини

Клієнтська частина системи Soundify реалізована у вигляді односторінкового застосунку (SPA) з використанням React. Структура проєкту побудована за принципами 3-layers архітектури, натхненої Backend-частиною системи, з чітким розділенням на:

- UI Layer (презентаційний шар);
- Business Logic Layer (BLL);
- Data Access Layer (API Layer).

Ядром управління станом застосунку виступає Zustand Store, який забезпечує централізоване реактивне зберігання стану та підтримку Event-Driven архітектури. У межах архітектури особливу увагу приділено чіткому розмежуванню відповідальностей між шарами:

- UI Layer виконує виключно функції відображення інтерфейсу та реагування на дії користувача;

- BLL Layer реалізовано у вигляді Managers (TrackManager, AlbumManager, AuthManager та інші), кожен з яких відповідає за бізнес-логіку певної сутності;
- API Layer інкапсулює усі взаємодії з Backend-сервісами через REST API, виконуючи роль Data Access Layer згідно з принципами 3-layers моделі.

Окрім класичних елементів SPA, у клієнтській частині реалізовано Hosted Services (Background Tasks), повністю перенесені з підходу .NET 8 (C# → TypeScript). Ці фонові сервіси працюють у зв’язці з State-driven патерном, підписуючись на оновлення стану Store та виконуючи довготривалі або періодичні задачі (наприклад, оновлення JWT-токену або керування відтворенням плеєра).

Крім того, у клієнтській частині було реалізовано повноцінний DI-контейнер, що дозволяє організовувати залежності між компонентами системи за принципами інверсії залежностей. При цьому використовується хук useInject, що забезпечує lazy-ініціалізацію залежностей у компонентах React. Завдяки такому підходу клієнтська частина системи має гнучку та модульну структуру, яка полегшує підтримку, розширення функціоналу та розвиток проєкту в майбутньому.

3.5.2 Шари архітектури та патерн State-Driven

При побудові клієнтської частини системи Soundify основною метою було створити архітектуру (рисунок 3.7), яка б забезпечувала чіткий розподіл відповідальностей між шарами, високу гнучкість та можливість легкого розширення. У якості базової моделі було обрано 3-layers архітектуру, яка використовується також у Backend-частині проєкту. В процесі проєктування були проаналізовані підходи Feature-Sliced Design (FSD), однак стандартна структура FSD виявилася надмірно громіздкою через дублювання папок та однакових назв. В результаті було розроблено оптимізований варіант архітектури, побудований за принципом Single Responsibility Principle (SRP).

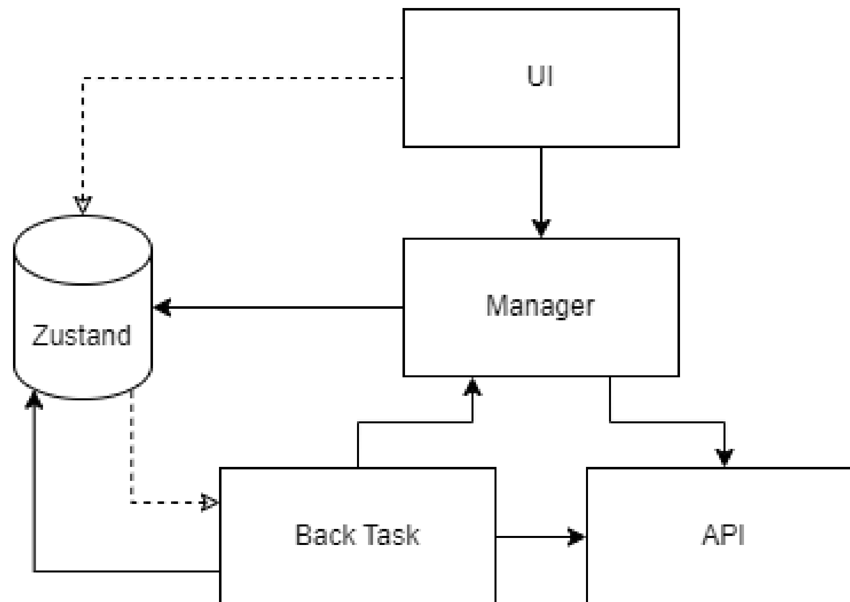

Рисунок 3.7 – Архітектура клієнтської частини системи Soundify (патерн State-Driven)

В рамках цієї архітектури клієнтська частина чітко розділена на такі основні шари:

- UI Layer відповідає виключно за відображення інтерфейсу та обробку взаємодії з користувачем. Будь-яка бізнес-логіка у UI відсутня;

- Business Logic Layer (BLL) реалізований у вигляді Managers (TrackManager, AlbumManager, AuthManager та ін.), кожен з яких відповідає за одну конкретну сутність. Managers не взаємодіють між собою напряму;

- Data Access Layer (API Layer) інкапсулює усі взаємодії з Backend, реалізовані через REST API;

- Zustand Store – централізоване сховище стану, яке забезпечує реактивність клієнтської частини;

- Hosted Services (Back Tasks) – фонові сервіси, що виконують довготривалі або періодичні задачі та реагують на зміни у Zustand Store.

В основі взаємодії між цими шарами лежить патерн State-Driven. Його ключова ідея – використання Zustand Store як єдиного медіатора стану між шарами системи.

Взаємодія будується таким чином: UI ініціює дії користувача та передає їх у відповідний Manager. Manager приймає рішення, які зміни необхідно внести у стан або які API-запити виконати. Всі зміни стану виконуються виключно через Zustand Store.

Managers не підписуються на Store і не реагують на його зміни, вони лише ініціюють оновлення, що дозволяє уникнути зворотних зв'язків та циклічних залежностей.

Hosted Services підписуються на відповідні Slice-и Zustand Store та реагують на зміни стану. Таким чином вони можуть виконувати асинхронні фонові задачі (наприклад, оновлення JWT або керування плеєром), не порушуючи чистої вертикальної архітектури. Важливою особливістю є те, що Hosted Services також можуть оновлювати Zustand Store, а UI автоматично реагує на оновлення за допомогою реактивної підписки.

Як видно зі схеми (рисунок 3.7), потоки даних у системі суворо впорядковані:

- дії користувача (UI) передаються у Manager;
- Manager ініціює зміну стану у Zustand Store та/або звернення до API;
- Back Tasks реагують на зміни у Store;
- UI реактивно оновлюється на підставі актуального стану Store.

Таким чином, архітектура забезпечує чітке вертикальне розділення, відсутність горизонтальних зв'язків та високу передбачуваність поведінки клієнтської частини. Крім того, такий підхід значно спрощує unit-тестування, оскільки тестові сценарії можуть емулювати зміни стану Store та перевіряти реакцію Hosted Services або UI.

3.5.3 Hosted Services (Back Tasks)

Однією з ключових особливостей клієнтської архітектури системи Soundify стало впровадження механізму Hosted Services (Back Tasks), що дозволяє реалізовувати у фронтенді повноцінну модель фонових задач.

Під час створення цього механізму було використано архітектурний підхід, що базується на практиках .NET 8, де Hosted Services виступають стандартним інструментом для запуску довготривалих або періодичних процесів. У даному проєкті логіка Hosted Services не була просто адаптована, а повністю перенесена з .NET 8: вихідний код з мови C# було переписано на TypeScript із урахуванням особливостей роботи у клієнтському середовищі.

У клієнтській архітектурі Hosted Services працюють у тісному зв'язку з State-Driven патерном та DI-контейнером. Hosted Services можуть отримувати у якості залежностей Managers та безпосередньо викликати бізнес-логіку BLL у разі потреби. Це дозволяє Hosted Services виконувати складні задачі, які вимагають повноцінної взаємодії з бізнес-рівнем, наприклад, ініціювати оновлення даних або запускати бізнес-процеси. Дана взаємодія реалізується через DI – Hosted Service отримує конкретний Manager у вигляді залежності (за контрактом), що дозволяє зберегти чисту архітектуру та чітке керування життєвим циклом компонентів.

Водночас, щоб уникнути зворотних жорстких залежностей, Managers не викликають Hosted Services напряму. Якщо Manager за підсумками взаємодії з користувачем або з API потребує ініціювати зміну поведінки Hosted Service, він просто оновлює стан Store. Hosted Service, у свою чергу, підписаний на відповідні частини Store та реагує на ці зміни. Саме для цього і було впроваджено State-Driven підхід, щоб Manager міг "керувати" Hosted Service опосередковано через зміну стану, не створюючи прямої залежності.

Таким чином, у системі формується логічна модель взаємодії. Managers виконують бізнес-логіку у відповідь на дії користувача, оновлюють стан. Hosted Services відстежують відповідні оновлення та реагують у фоні. Водночас Hosted Services при необхідності можуть напряму використовувати функціонал Managers через DI для складніших бізнес-операцій. В результаті досягається баланс між гнучкістю взаємодії та чистотою архітектури: усуваються циклічні залежності, але зберігається необхідна керованість потоками даних та подій.

У поточній реалізації системи прикладом такої взаємодії є AuthService, який у фоновому режимі слідкує за станом авторизації і за потреби може напряму використовувати AuthManager для оновлення токенів або ініціювання додаткових бізнес-дій. Іншим прикладом є AudioService, який слідкує за станом плеєра та управляє відтворенням, при цьому також має можливість викликати TrackManager або інші Managers через DI при обробці складних сценаріїв.

Завдяки такій архітектурі Hosted Services повністю інтегруються у клієнтську частину як рівноправні учасники бізнес-процесів, при цьому зберігається чіткий розподіл відповідальностей та контрольований потік залежностей. Крім того, такий підхід значно спрощує unit-тестування: у тестах легко емулювати зміни стану Store та перевіряти реакцію як Hosted Services, так і Managers у рамках передбачуваного життєвого циклу.

3.5.4 Dependency Injection (DI)

Ще однією важливою складовою архітектури клієнтської частини системи Soundify є власний механізм Dependency Injection (DI), що був розроблений на базі підходів .NET 8. При побудові DI було поставлено завдання не просто адаптувати ідею DI для TypeScript, а повністю перенести архітектурний патерн DI з .NET 8, максимально зберігши його можливості та зручність використання.

Особливістю реалізації у клієнтському середовищі є те, що інтерфейси TypeScript не існують у runtime, отже, їх неможливо використовувати як контракти для DI-контейнера. Для вирішення цієї проблеми було прийнято архітектурне рішення будувати контракти на основі abstract class. Це дозволило створити чітку ієрархію залежностей, підтримати типізацію на рівні TypeScript та забезпечити можливість контролю у runtime.

Всі ключові компоненти системи (Managers, Hosted Services, API) реєструються у DI-контейнері за відповідними abstract class-контрактами.

Отримання залежності можливе тільки через спеціальний getter DI-контейнера, при цьому компонент, що отримує залежність, передає у getter саме контракт (abstract class), а не конкретну реалізацію. Такий підхід дозволяє зберігати повну ізоляцію між компонентами та забезпечує слабке зв'язування.

Для зручної інтеграції DI у React-компоненти було розроблено спеціальний хук useInject, який дозволяє отримувати залежності у компонентах з підтримкою lazy-ініціалізації. Це дає можливість оптимізувати час ініціалізації та дозволяє ефективно використовувати залежності у великих SPA-додатках без перевантаження життєвого циклу компонентів.

Завдяки такій побудові DI вдалося досягти кількох важливих переваг. По-перше, було забезпечено чисту архітектуру без жорсткого зв'язування між компонентами, що суттєво полегшує розвиток та рефакторинг коду. По-друге, така структура ідеально підходить для unit-тестування, оскільки всі залежності впроваджуються за контрактами, їх легко можна підмінити на мок-реалізації під час тестів. По-третє, використання abstract class у якості контрактів дозволяє уникнути проблем з типізацією та сумісністю у TypeScript.

У підсумку можливо зазначити, що реалізований механізм DI став важливою основою архітектури клієнтської частини системи Soundify, забезпечуючи гнучкість, модульність та високу якість коду.

Як висновок можна відзначити, що завдяки поєднанню архітектурних патернів Backend (модель 3-layers) з кращими практиками Frontend (оптимізований підхід FSD) та використанню сучасного менеджера стану Zustand, вдалося створити легко масштабовану, тестовану та строго структуровану клієнтську архітектуру.

Особливо важливим є те, що ця архітектура орієнтована на зручність для розробників з бекграундом у .NET, оскільки в ній було повністю перенесено такі ключові концепції, як Dependency Injection (DI) та HostedService. При цьому мова йде не про спрощене копіювання, а про

повноцінну адаптацію цих патернів до реалій Frontend-розробки, з урахуванням специфіки TypeScript та клієнтського середовища.

Таким чином, створена архітектура забезпечує високу якість коду, передбачувану масштабованість та є зручною платформою для подальшого розвитку клієнтської частини системи Soundify.

3.6 Архітектура баз даних системи Soundify

У системі Soundify було реалізовано розділення баз даних на окремі AuthDB та MusicDB з метою оптимального розподілу навантаження та підвищення стійкості системи.

База даних AuthDB працює під постійним навантаженням, оскільки обробляє великий обсяг операцій, пов'язаних з оновленням JWT-токенів (refresh), керуванням профілем користувача та процесами скидання паролю. Додатково, через наявність відкритих endpoint-ів для оновлення JWT та відновлення доступу, саме AuthDB є більш вразливою до потенційних DDoS-атак.

Враховуючи ці особливості, було прийнято архітектурне рішення розділити бази даних: навіть у разі значного навантаження або DDoS-атаки на сервіс авторизації, користувачі зможуть продовжувати слухати музику, оскільки доступ до основного музичного контенту забезпечується через окрему базу даних MusicDB, що не залежить від AuthDB.

До того ж, в системі з самого початку передбачено механізм передчасного оновлення токенів (за 30 секунд до закінчення терміну дії JWT), що дозволяє користувачу підтримувати авторизований сеанс навіть за умови тимчасової недоступності Auth Service.

Таким чином, розділення баз даних не лише дозволяє ефективно балансувати навантаження між сервісами, а й значно підвищує відмовостійкість та стійкість усієї системи до атак.

3.6.1 База даних AuthDB

База даних AuthDB відповідає за зберігання усієї інформації, пов’язаної з обліковими записами користувачів та процесами авторизації у системі Soundify. Це окрема спеціалізована база даних, яка обслуговує лише Auth Service та забезпечує надійне управління автентифікацією та сесіями користувачів.

У AuthDB зберігаються профілі користувачів, хешовані паролі, інформація про способи автентифікації (у тому числі через сторонні провайдери), а також refresh-токени, які забезпечують безпечне оновлення сесій. Саме через цю базу реалізовано всі процеси керування обліковими записами: реєстрація, вхід, скидання пароля, підтвердження електронної пошти. Особливістю AuthDB є те, що вона працює під постійним навантаженням, оскільки обробляє високочастотні операції оновлення JWT-токенів та інші запити з відкритих endpoint-ів, що робить її найбільш чутливою до потенційних DDoS-атак. Саме тому AuthDB була винесена в окрему базу, ізольовану від основної MusicDB.

Взаємодія з AuthDB здійснюється виключно через Auth Service API. Інші сервіси системи не мають прямого доступу до цієї бази. Крім того, Auth Service взаємодіє з Notification Service за допомогою RabbitMQ: наприклад, для надсилання листів з підтвердженням реєстрації або скиданням пароля. Технологічно для доступу до AuthDB використовується Entity Framework Core, що дозволяє реалізовувати всі бізнес-процеси у рамках типізованої моделі. Модель бази даних AuthDB представлено на рисунку 3.8.

Основу структури AuthDB складають наступні ключові таблиці:

- Users – зберігає профілі користувачів, включаючи хешовані паролі, інформацію про ролі, статус облікового запису, способи автентифікації;

- RefreshTokens – таблиця для зберігання активних refresh-токенів, термінів їх дії та зв’язку з відповідним користувачем. Саме ця таблиця забезпечує механізм "плавного" оновлення JWT у системі.

У підсумку слід зазначити, що AuthDB реалізовано як окрему ізольовану базу даних, що дозволяє забезпечити високу стійкість до навантажень і атак, чіткий контроль доступу та надійне управління життєвим циклом облікових записів у системі Soundify.

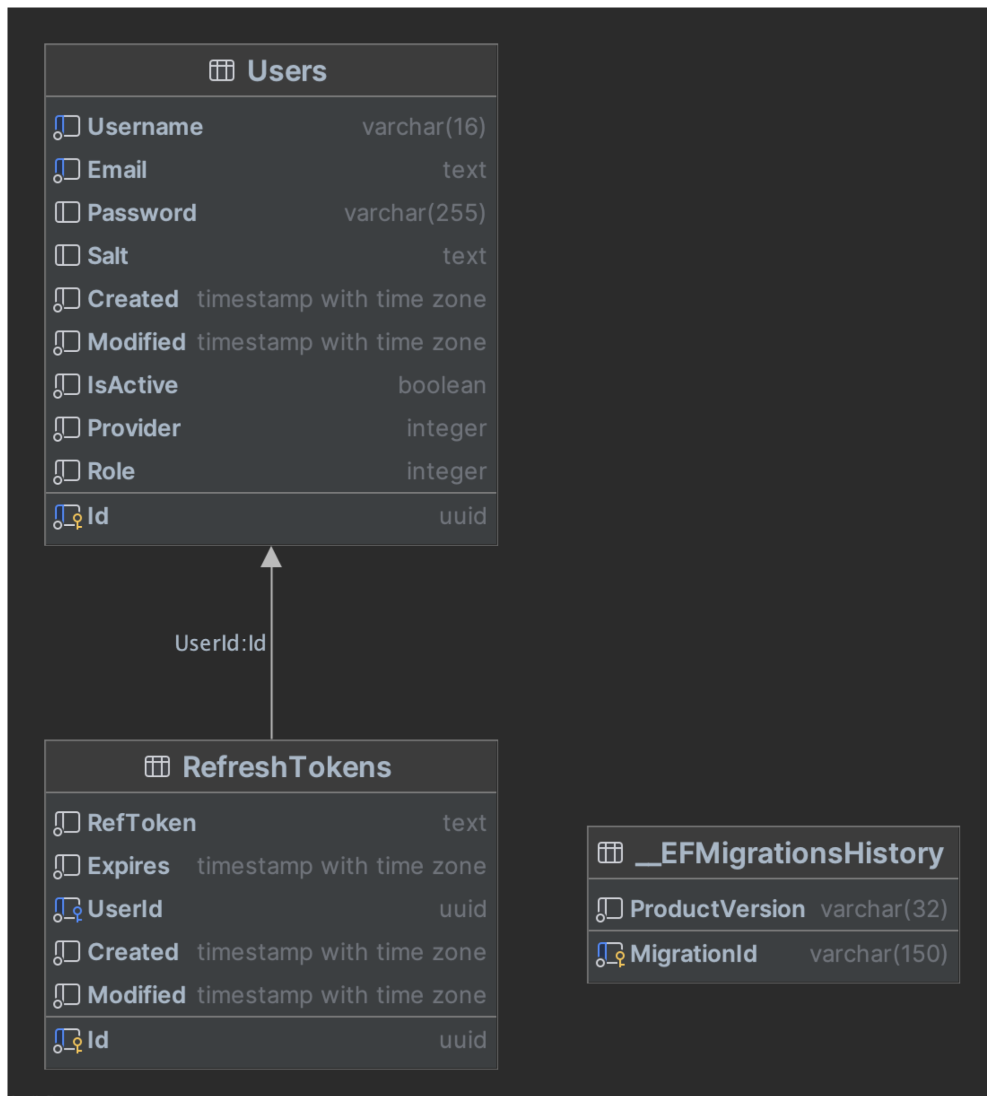

Рисунок 3.8 – Модель даних AuthDB

3.6.2 База даних MusicDB

База даних MusicDB є основним сховищем метаданих музичного контенту у системі Soundify. Вона обслуговує весь функціонал, пов’язаний із каталогізацією треків, альбомів, виконавців, а також із збереженням персоналізованої інформації користувачів, такої як рейтинги, улюблені треки та плейлисти.

На відміну від AuthDB, база MusicDB не перебуває під постійним високочастотним навантаженням з боку авторизаційних процесів, проте обробляє значний обсяг запитів при побудові користувацьких бібліотек, створенні персоналізованих добірок та відтворенні музичного контенту. Взаємодія з MusicDB відбувається виключно через Music Service API, що забезпечує чітке розмежування рівнів доступу та ізоляцію бізнес-логіки.

Структура MusicDB дозволяє ефективно моделювати як базові сутності музичного контенту (виконавці, альбоми, треки), так і допоміжні – жанри, рейтинги, плейлисти, улюблені треки. Також Music Service інтегровано з Storage Service, що дозволяє формувати тимчасові токени для доступу до медіаконтенту на базі метаданих із MusicDB. На рисунку 3.9 представлено модель бази даних MusicDB.

У складі MusicDB виділяються такі ключові таблиці:

- Artists містить інформацію про виконавців, включаючи базові метадані та посилання на соціальні мережі;

- Albums зберігає альбоми, пов’язані з виконавцями, а також обкладинки та додаткові дані;

- Tracks – основна таблиця для треків, яка включає посилання на файли, тривалість, метадані треку, а також зв’язок із альбомами та виконавцями;

- Genres – класифікація музики за жанрами;

- Playlists та PlaylistTracks – таблиці для збереження плейлистів користувачів та складу треків у кожному плейлисті;

- TrackRatings зберігає оцінки треків, виставлені користувачами;

- UserFavorites – таблиця для фіксації улюблених треків кожного користувача.

У підсумку слід зазначити, що MusicDB побудовано як оптимізовану предметно-орієнтовану базу даних, яка забезпечує високу продуктивність при виконанні запитів, гнучкість моделювання контенту та зручну інтеграцію з іншими мікросервісами системи Soundify.

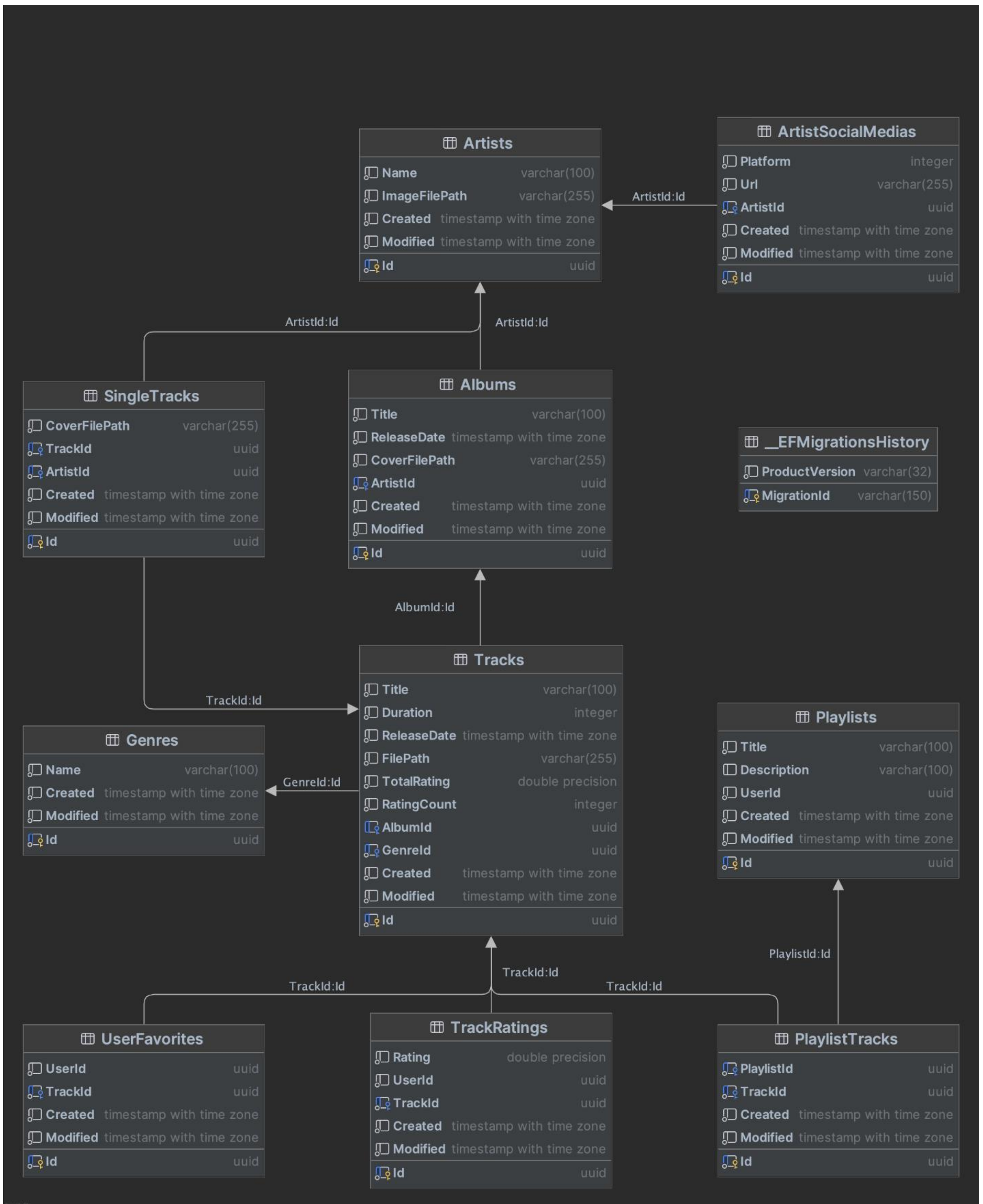

Рисунок 3.9 – Модель даних MusicDB

Таким чином, розділення баз даних у системі Soundify дозволило досягти оптимального балансу між продуктивністю, масштабованістю та стійкістю до атак. AuthDB забезпечує надійне та ізольоване управління обліковими записами та процесами авторизації, тоді як MusicDB обслуговує

основний контентний функціонал системи, пов’язаний із зберіганням та обробкою музичних метаданих. Таке архітектурне рішення сприяє підвищенню загальної відмовостійкості платформи та спрощує масштабування кожного окремого сервісу у рамках мікросервісної архітектури Soundify.

Підсумовуючи інформацію третього розділу, можна відзначити, що завдяки правильно обраним архітектурним підходам вдалося створити гнучку та легко масштабовану архітектуру системи Soundify. Використання мікросервісної моделі дозволяє ефективно розподіляти навантаження між сервісами та забезпечує високу відмовостійкість. Застосування 3-layers підходу у Web API-сервісах сприяє чіткому розділенню бізнес-логіки, доступу до даних та API-рівня, що значно спрощує підтримку та розвиток кожного сервісу.

На клієнтській частині було реалізовано оптимізований гібрид 3-layers та FSD-підходу, що забезпечує зрозумілу структуру коду та чітке розмежування відповідальностей. Крім того, глибока інтеграція з сервісами AWS дозволила уникнути необхідності розробки складних кастомних рішень для стримінгового відтворення аудіо та управління медіаконтентом. В результаті вдалося побудувати таку архітектуру, яка дозволяє підтримувати та розвивати систему навіть командами розробників середнього рівня кваліфікації. При цьому загальні витрати на розробку та підтримку будуть суттєво нижчими, ніж у випадку використання повністю самописних стримінгових рішень. Обрана архітектура створює надійний фундамент для подальшого розвитку платформи Soundify та її масштабування.

4 ІНСТРУКЦІЯ КОРИСТУВАЧА

4.1 Вхід до системи

На сторінці входу до системи користувачеві доступні два стандартних способи авторизації: за допомогою електронної пошти та пароля або через Google-акаунт (рисунок 4.1).

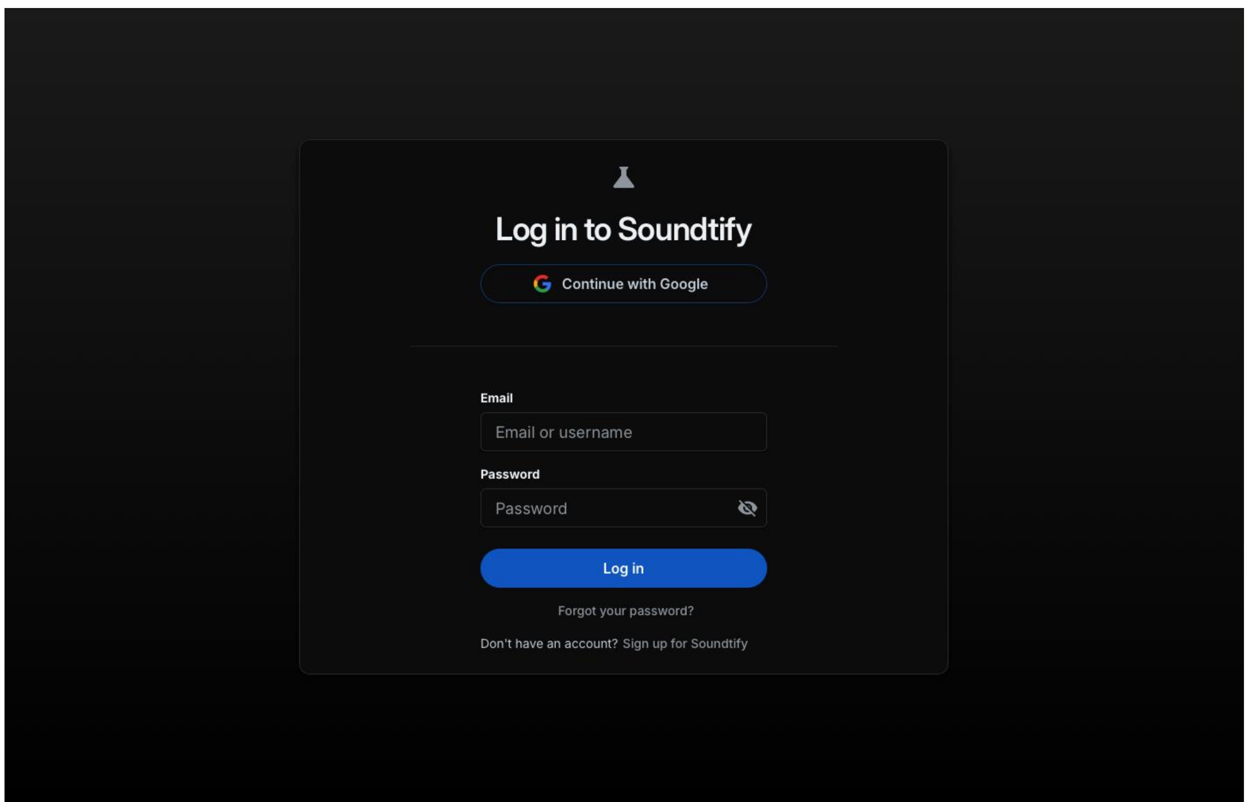

Рисунок 4.1 – Форма авторизації у системі Soundify

Інтерфейс авторизації реалізовано у мінімалістичному стилі, з акцентом на зручність взаємодії. У центрі екрана розміщено форму входу, що містить два поля: Email та Password. Поле для пароля підтримує стандартну функцію приховування/відображення символів, а також дозволяє вставити збережені облікові дані, якщо такі є у браузері.

Після заповнення полів користувач натискає кнопку Log in, і у разі успішної авторизації система перенаправляє його на головну сторінку або до

списку треків. У випадку невірного логіна або пароля виводиться відповідне повідомлення про помилку.

Альтернативно користувач може обрати авторизацію через обліковий запис Google, натиснувши кнопку Continue with Google. У цьому випадку відбувається перенаправлення до сторінки автентифікації Google OAuth 2.0, після чого користувач автоматично повертається до застосунку з активною сесією. Також на сторінці розміщено посилання Forgot your password? та Sign up for Soundify, які наразі не реалізовані: при спробі переходу за ними відкривається службова сторінка 404. Ці функції передбачені до реалізації у майбутніх версіях платформи.

4.2 Сторінка треків

Після успішної авторизації користувач автоматично перенаправляється на головну сторінку – сторінку треків, яка є основною точкою взаємодії з музичним контентом (рисунок 4.2).

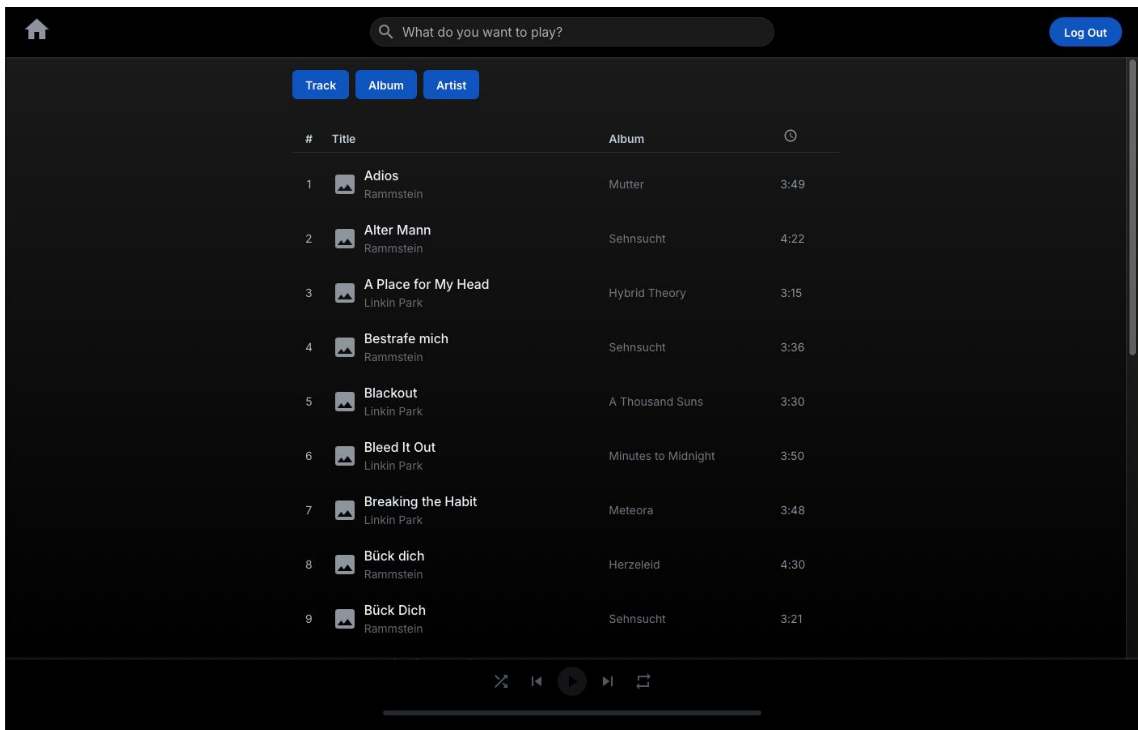

Рисунок 4.2 – Сторінка треків після входу в систему

Інтерфейс сторінки побудований за принципом класичного layout з фіксованими зонами:

- у верхній частині (header) розміщується панель навігації, що містить поле пошуку, кнопку виходу (Log Out) та три навігаційні кнопки: Track, Album і Artist;
- нижню частину екрана займає медіаплеєр, доступний на всіх сторінках після входу.

Натискання на одну з кнопок Track / Album / Artist ініціює перехід до відповідної сторінки результатів. Якщо поле пошуку порожнє, то система автоматично запитує останні додані сутності з бази даних, що дозволяє користувачу бачити найактуальніший контент без додаткових дій.

Основну частину екрану займає таблиця треків, яка відображає перелік доступних композицій із зазначенням назви, виконавця, альбому та тривалості. Кожен рядок таблиці містить наступні елементи:

- індекс треку у межах поточної таблиці, який автоматично змінюється на іконку відтворення (Play) при наведенні курсора;
- обкладинка (cover) треку, стилізована у вигляді ескізу;
- назва композиції – інтерактивна кнопка, яка відкриває сторінку відповідного треку;
- ім'я виконавця – інтерактивна кнопка, що веде на сторінку виконавця;
- кнопка додавання у плейлист – зарезервована для реалізації у наступних версіях системи;
- тривалість треку у форматі хвилини:секунди;
- кнопка додаткових дій (три крапки) – також планується до реалізації у майбутньому.

Таблиця є повністю інтерактивною: при наведенні курсора на рядок змінюється фон на прозоро-сірий, що створює чіткий контраст із темним фоном і дозволяє легко орієнтуватися. Всі елементи повертаються до стандартного вигляду після зняття наведення.

Ця поведінка формує зручний та інтуїтивно зрозумілий досвід користувача – система дозволяє максимально швидко знаходити та запускати музику без зайвих переходів та перезавантажень сторінок. Усі елементи інтерфейсу оформлено в єдиному стилі, що підтримує естетику та функціональність темної теми застосунку.

Крім інтерактивної кнопки відтворення, система також реалізує візуальне підсвічування активного рядка при наведенні курсора (рисунок 4.3). У момент наведення фон відповідного треку змінюється на прозоро-сірий, що створює чіткий контраст із темною темою інтерфейсу. Це дозволяє легко орієнтуватися у таблиці й точно відслідковувати, який саме трек готовий до взаємодії.

Праворуч у рядку також з'являється іконка додаткового меню. Загалом такий ефект робить взаємодію максимально інтуїтивною, передбачуваною та візуально зручною, не перевантажуючи користувача зайвими деталями. Користувач одразу розуміє, що за іконкою приховані додаткові дії, що сприяє кращій навігації та контролю.

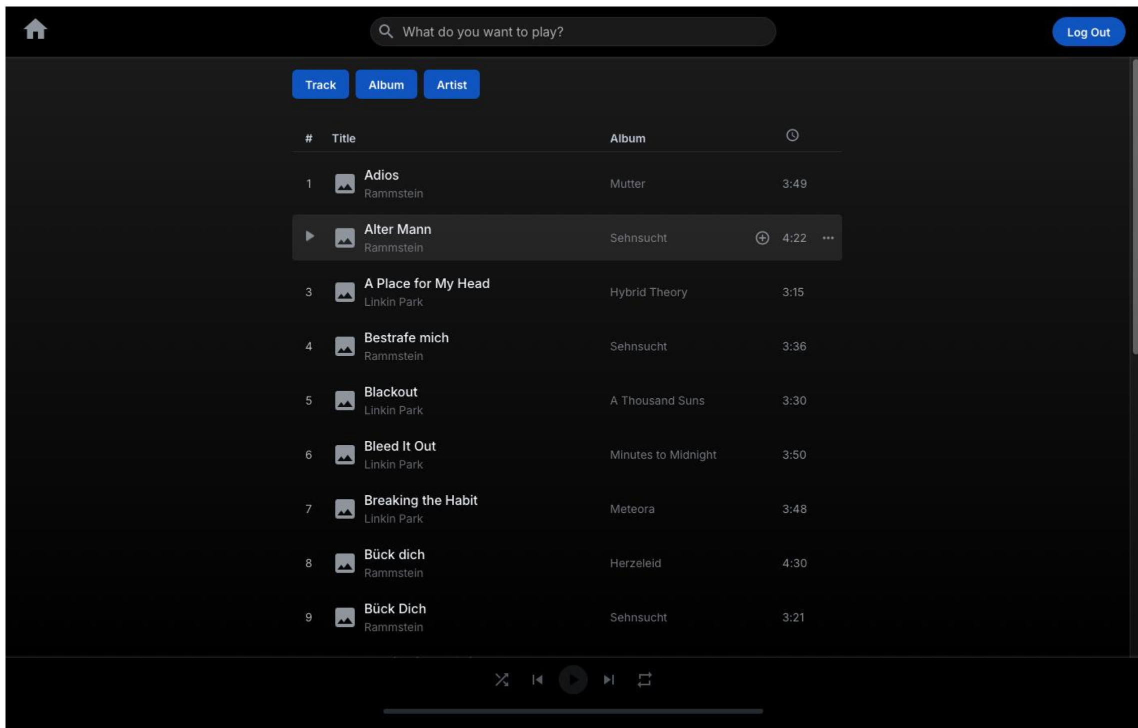

Рисунок 4.3 – Ефект наведення на рядок треку

Після натискання на кнопку Play у таблиці треків відбувається запуск аудіовідтворення. У цей момент hover-ефект фіксується на активному треку, що дозволяє користувачу легко бачити, який саме трек наразі грає. Одночасно у нижній частині інтерфейсу активується повноцінний медіаплеєр (рисунок 4.4).

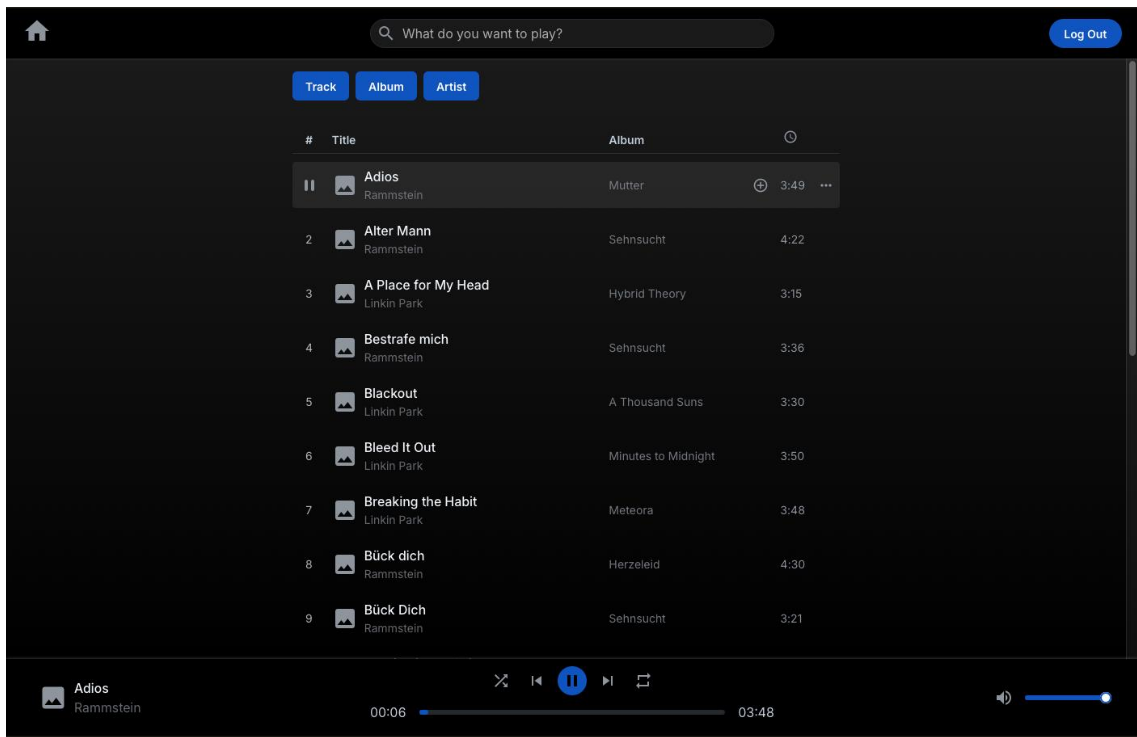

Рисунок 4.4 – Активний плеєр під час відтворення треку

У лівій частині плеєра з’являється коротка інформація про трек: назва композиції, виконавець та обкладинка (cover). Ці елементи є інтерактивними: при натисканні вони перенаправляють користувача на сторінку відповідного альбому або виконавця.

У центрі плеєра розміщується основна панель керування:

- кнопка відтворення/пауза;
- індикатор прогресу відтворення з поточним і загальним часом;
- інтерактивний повзунок, який дозволяє вручну переміщати позицію треку.

Праворуч внизу розміщено налаштування гучності:

- кнопка вимкнення/увімкнення звуку, яка дозволяє миттєво

заглушити звук і повернутися до попереднього рівня;

- повзунок гучності, що дозволяє точно підлаштувати рівень звуку для комфортного прослуховування.

Загалом, такий підхід дозволяє користувачу повністю контролювати процес відтворення, не покидаючи поточної сторінки та не втрачаючи контекст взаємодії. Усі елементи плеєра є реактивними, інтерактивними та адаптованими під темну тему інтерфейсу.

На сторінці треків реалізовано динамічну пошукову систему (рисунок 4.5), яка автоматично оновлює таблицю результатів. При введенні трьох і більше символів у поле пошуку система надсилає запит до серверу та відображає лише ті треки, що відповідають введеному запиту. Таким чином, пошук працює у режимі реального часу.

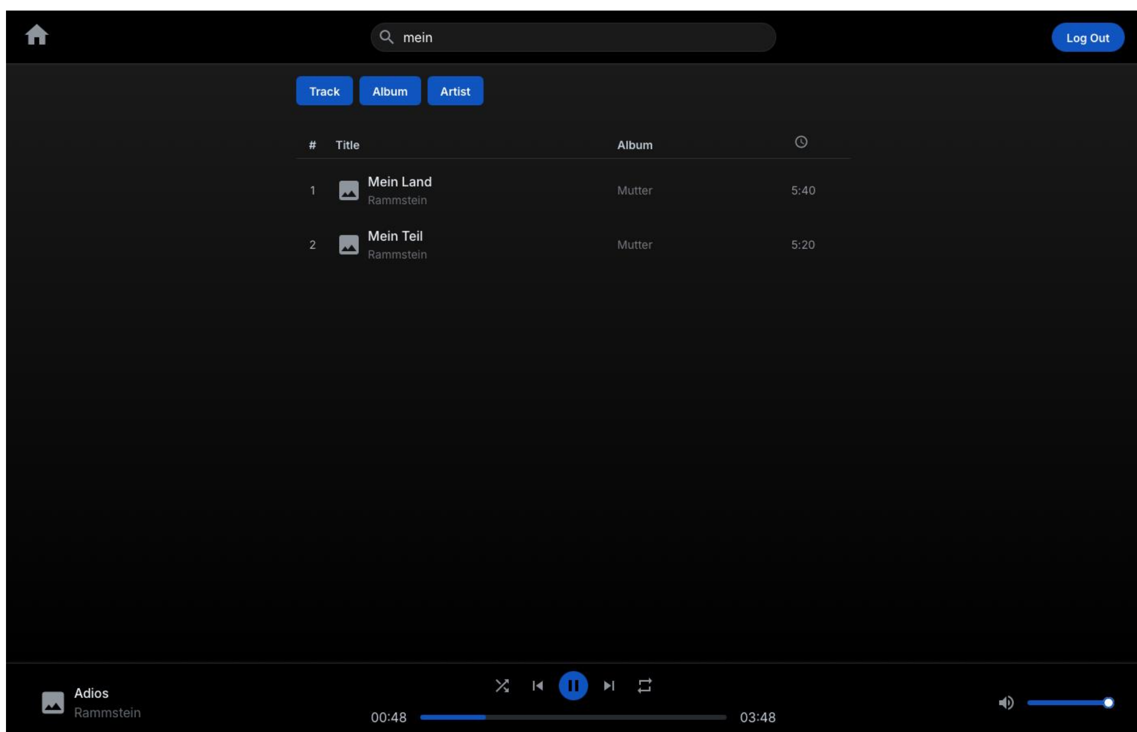

Рисунок 4.5 – Пошук і фільтрація треків за ключовими словами

У випадку, якщо користувач очищує поле пошуку, система автоматично повертається до відображення останніх доданих треків із бази даних, забезпечуючи зручну навігацію навіть без конкретного запиту.

Особливу увагу варто звернути на поведінку медіаплеєра: після запуску відтворення обраного треку, плеєр продовжує відтворення незалежно від подальших дій користувача на сторінці. Зміна сторінки, запуск пошуку чи скролінг жодним чином не переривають програвання. Це дозволяє зберегти континуїтет прослуховування, що є важливим UX-рішенням для стримінгового застосунку.

4.3 Сторінка альбомів

На сторінці альбомів (рисунок 4.6) зберігається загальна структура інтерфейсу, що вже була представлена раніше: у верхній частині – панель навігації з полем пошуку, кнопками Track / Album / Artist, а внизу – плеєр, що залишається активним під час усієї сесії користувача.

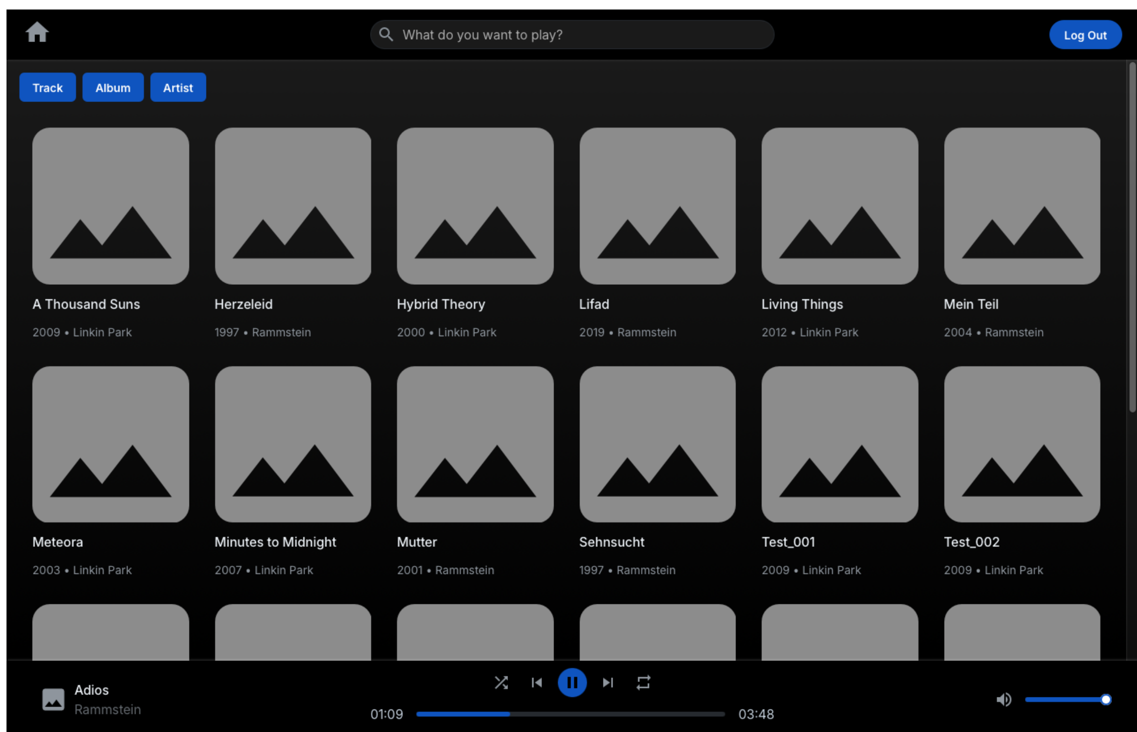

Рисунок 4.6 – Сторінка альбомів у системі Soundify

На відміну від таблиці треків, у цьому розділі контент подається у вигляді сітки карток, кожна з яких представляє окремий музичний альбом. Візуальне оформлення картки включає:

- обкладинку альбому (cover) у стилізованому квадратному форматі;
- назву альбому – інтерактивний елемент, при натисканні на який відбувається перехід до сторінки альбому з переліком його треків;
- рік релізу та виконавця – останній також є інтерактивним і веде на сторінку виконавця;

Реалізовано ефект наведення (hover), аналогічний до таблиці треків: при наведенні курсора на картку змінюється фон, створюючи чіткий візуальний акцент на об’єкті. Це забезпечує кращу навігацію по сторінці, особливо при великій кількості контенту.

Розташування альбомів здійснюється адаптивно: кількість карток у рядку залежить від ширини екрана користувача, що дозволяє однаково зручно переглядати сторінку на широкоформатних і стандартних дисплеях.

На сторінці альбомів також реалізовано динамічний пошук (рисунок 4.7), що працює за тим же принципом, що й на сторінці треків. Після введення трьох та більше символів у поле пошуку автоматично виконується фільтрація альбомів, і на екрані відображаються лише ті картки, які відповідають запиту.

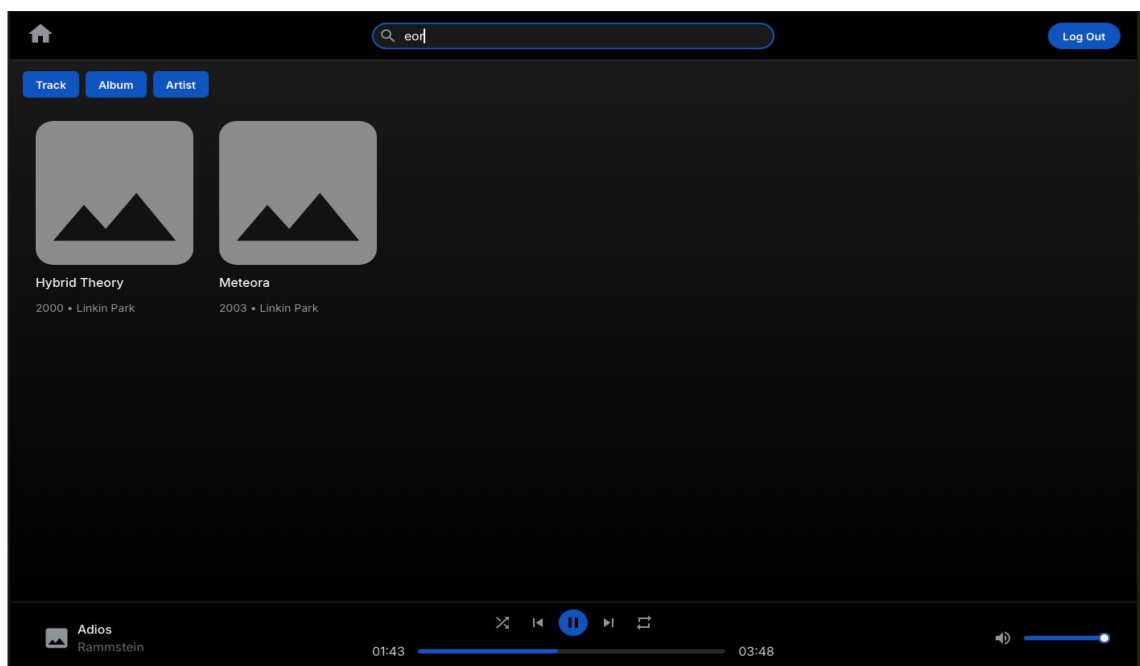

Рисунок 4.7 – Результати пошуку альбомів за ключовим словом

Ключова відмінність полягає у форматі відображення: замість рядків у таблиці користувач бачить відфільтровані картки альбомів, що дозволяє зберегти візуальну послідовність та однорідність інтерфейсу. При очищенні пошуку система повертає користувача до повного списку альбомів, як і у випадку з треками.

Таким чином, пошук є уніфікованим та передбачуваним на всіх сторінках, але адаптованим до конкретного типу вмісту – треків або альбомів.

4.4 Сторінка треків конкретного альбому

Сторінка треків конкретного альбому дозволяє користувачу ознайомитися з повним вмістом вибраного альбому, надаючи зручний доступ до всіх треків у компактному та знайомому форматі (рисунок 4.8).

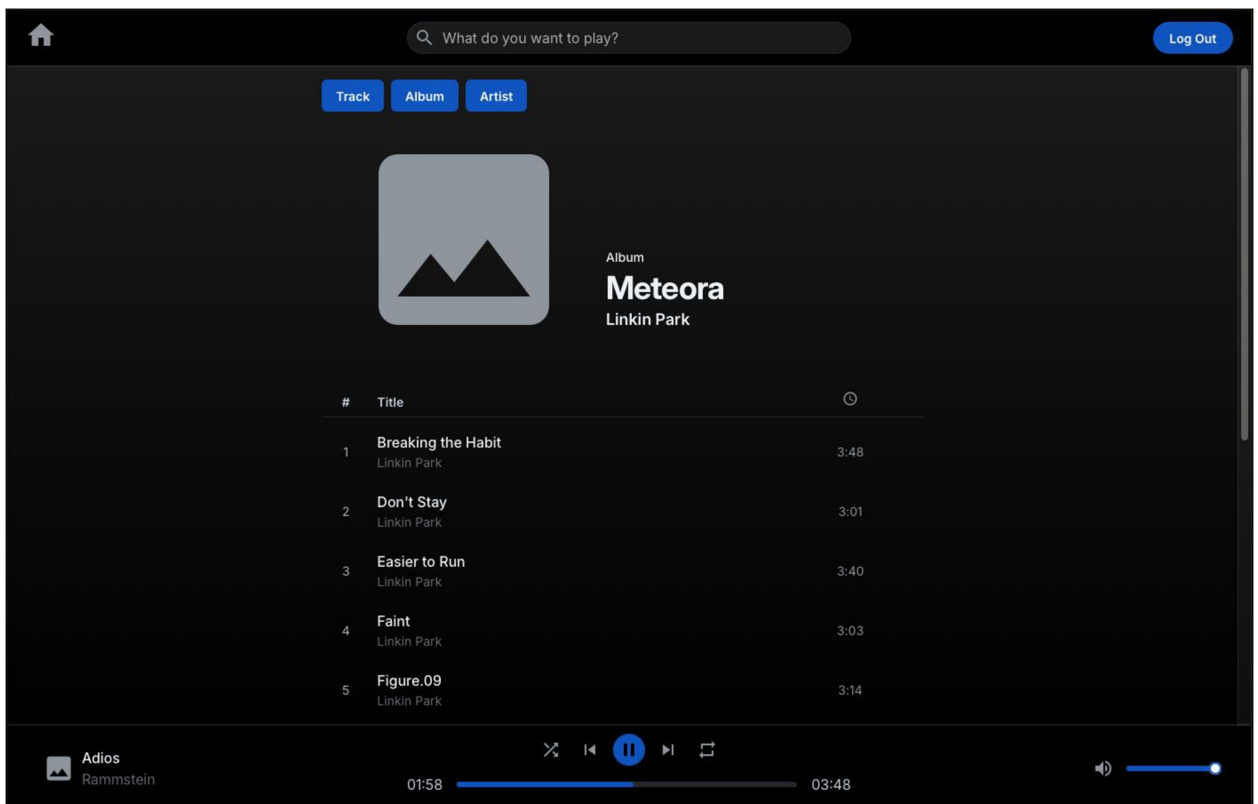

Рисунок 4.8 – Сторінка треків конкретного альбому

Інтерфейс зберігає загальну структуру та стиль сторінки треків, проте має кілька візуальних та функціональних відмінностей. У верхній частині відображається обкладинка альбому, а поруч – назва альбому та ім'я виконавця. Ці елементи виконують виключно інформативну функцію й не є інтерактивними, на відміну від таблиці нижче.

Основну частину екрану займає інтерактивна таблиця треків альбому. Вона має той самий функціонал, що й основна сторінка треків:

- при наведенні курсора на рядок з'являється кнопка «Play» замість індексу;
- активний трек підсвічується сірим фоном;
- плеєр у нижній частині автоматично оновлюється після натискання на трек;

На відміну від глобальної таблиці треків, у цьому списку не відображається обкладинка для кожного треку, оскільки всі вони належать одному й тому самому альбому: цю інформацію дублювати не потрібно.

Всі треки згруповані за альбомом та впорядковані відповідно до їх порядку в альбомі. Завдяки цьому реалізується логічна навігація, яка наближена до класичного прослуховування альбомів у цифрових бібліотеках.

Підсумовуючи розділ 4, можна відзначити, що у системі Soundify було реалізовано мінімалістичний та інтуїтивно зрозумілий інтерфейс, що дозволяє зосередити увагу користувача на головному – прослуховуванні музики. Завдяки збереженню логіки навігації, властивої популярним стримінговим платформам на кшталт Spotify, вдалося забезпечити зручність взаємодії з треками, альбомами та виконавцями, при цьому уникнувши перевантаження інтерфейсу зайвими візуальними або функціональними елементами.

На всіх сторінках збережено єдину архітектуру layout – з фіксованим плеєром у нижній частині та навігаційною панеллю у верхній, що створює послідовний і передбачуваний досвід користувача.

Інтерактивна поведінка елементів, реактивність інтерфейсу та

стабільна робота плеєра незалежно від переходів між сторінками гарантують високу якість UX. Реалізований функціонал уже сьогодні охоплює ключові потреби користувача, а заплановані доповнення (наприклад, плейлисти, улюблені треки, розширене керування контентом) стануть природним розширенням вже існуючої структури без порушення її цілісності.

Таким чином, інтерфейс користувача Soundify вже на поточному етапі відповідає вимогам сучасних SPA-застосунків: простий у використанні, швидкий у реакціях, стабільний у роботі та готовий до подальшого розвитку.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було створено повноцінний стримінговий застосунок Soundify, який поєднує в собі сучасні технології веброзробки та хмарні сервіси, було охоплено всі етапи розробки: від проєктування архітектури до реалізації клієнтської та серверної частин.

Серверна частина побудована з використанням ASP.NET Core з архітектурою 3-layers, що забезпечує чіткий розподіл відповідальностей між API, бізнес-логікою та доступом до даних. Для організації взаємодії між мікросервісами було застосовано RabbitMQ у двох режимах: черги повідомлень та RPC-запити з використанням MessagePack для ефективної передачі даних. Бази даних були поділені на AuthDB та MusicDB, що дозволило розподілити навантаження і підвищити безпеку системи.

Клієнтська частина реалізована за допомогою React, Zustand та Material UI. Архітектура фронтенду заснована на гібриді 3-layers та оптимізованого Feature-Sliced Design (FSD). Було реалізовано власний DI-контейнер, а також повністю перенесено логіку Hosted Services з .NET на TypeScript, що дозволило створити керовану, реактивну та зручно масштабовану архітектуру.

Контейнеризація всіх компонентів виконана через Docker, для локального управління контейнерами використовувався Portainer. Система була розгорнута на хмарних сервісах AWS EC2. Для обробки медіаконтенту застосовувалися AWS S3, AWS MediaConvert, CloudFront, а також AWS SES для розсилки електронних листів. Доступ до контенту реалізований через підписані посилання, що забезпечує безпеку та контроль.

У підсумку вдалося створити гнучку, продуктивну та масштабовану систему, яку можна підтримувати та розвивати навіть силами невеликої команди. Застосування мікросервісної архітектури, перевірених підходів 3-layers та SRP, а також глибока інтеграція з AWS забезпечують високий рівень надійності та готовність до реального комерційного використання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Price M.J. C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 8, Blazor, and EF Core 8. Publisher: Packt Publishing, 8th Edition, Kindle Edition. 2023. 21.8 MB

2. Blokdyk G. JSON Web Token Complete Self-Assessment Guide. Publisher: 5STARCooks. 2022. 306 p.

3. Blokdyk G. CI CD. A Complete Guide – 2021 Edition. Publisher: The Art of Service – CI CD Publishing, 2020. 312 p.

4. Luksa M. Kubernetes in Action. Publisher: Manning, First Edition, 2018. 624 p.

5. Costa L. RabbitMQ Simplified: A Beginner's Step-by-Step Guide. Kindle Edition, First Edition, 2023. 360 KB.

6. Bharadwaj V. Database Transactions and Concurrency Control: Isolation Levels and MVCC Demystified with Study of PostgreSQL. Kindle Edition, 2022. 15.4 MB.

7. WebLighters Studio. JavaScript and the DOM: Interacting with Web Pages. Kindle Edition, 2025. 1.5 MB.

8. Scott E. SPA Design and Architecture: Understanding single-page web applications. Publisher: Manning, 1st Edition, Kindle Edition. 2015. 9.5 MB.

9. Gascón U. Node.js for Beginners: A comprehensive guide to building efficient, full-featured web applications with Node.js. Publisher: Packt Publishing, 2024. 382 p.

10. Documentation Team. Amazon CloudFront Developer Guide. Publisher: Samurai Media Limited, 20218. 440 p.

11. Russell J., Cohn R. HTTP Live Streaming. Publisher: Book on Demand Ltd., 2012. 94 p.

12. Baranov S. From Algorithm to Digital System: HLS and RTL tool Synthagate in Digital System Design. Publisher: ISBN Canada, 2020. 4.3 MB.