Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання і програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 − Інженерія програмного забезпечення

На тему: Розробка веб-застосунку для потокового відтворення музики

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.

Студент групи ІПЗ-22-1

_____ / Є. Г. Кісельов /

Керівник кваліфікаційної роботи _____ / О. Г. Рибальченко /

Завідувач кафедри _____ / А. М. Стрюк /

Кривий Ріг
2026

Криворізький національний університет

Факультет	Інформаційних технологій
Кафедра	Моделювання та програмного забезпечення
Ступінь вищої освіти	бакалавр
Спеціальність	121 − Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Зав. кафедрою
к.п.н., доц. А.М. Стрюк
“____”_____20___р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-22-1 Кісельову Єгору Геннадійовичу

1. Тема: Розробка веб-застосунку для потокового відтворення музики затверджено наказом по КНУ №284-с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «10» червня 2026 р.
3. Вихідні дані по роботі:

Результати аналізу існуючих музичних стрімінгових сервісів (Spotify, Apple Music, YouTube Music), результати UX-дослідження, технічна документація використаних технологій (Node.js, TypeScript, React, PostgreSQL, OpenAI API, DeepL API, Stripe).

4. Зміст пояснювальної записки (перелік питань, що їх треба розробити):

Вступ. Розділ 1 – Аналіз сучасного стану музичних стрімінгових сервісів та обґрунтування актуальності роботи. Розділ 2 – Дослідження користувацького досвіду та обґрунтування функціоналу. Розділ 3 – Проєктування вебзастосунку. Розділ 4 – Реалізація та тестування вебзастосунку. Висновки. Список використаних джерел.

5. Перелік ілюстративного матеріалу: структурна схема, діаграма варіантів використання (прецедентів), контекстна діаграма, діаграма послідовностей, схема бази даних, вайрфрейми сторінок, знімки екрану розробленого вебзастосунку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Огляд літературних джерел відповідно до теми кваліфікаційної роботи	31.03.2026 – 04.04.2026
2	Пошук та проведення аналізу існуючих аналогів програмних продуктів	05.04.2026 – 10.04.2026
3	Формулювання актуальності та мети і визначення завдань роботи	11.04.2026 – 13.04.2026
4	Підготовка матеріалів першого розділу роботи	14.04.2026 – 16.04.2026
5	Проведення UX-дослідження: анкетування аудиторії та інтерв'ю	17.04.2026 – 19.04.2026
6	Розробка математичного апарату, діаграм архітектури продукту, проєктування БД	20.04.2026 – 21.04.2026
7	Створення вайрфреймів UI-компонентів вебзастосунку	22.04.2026 – 23.04.2026
8	Підготовка матеріалів другого розділу роботи	24.04.2026 – 26.04.2026
9	Створення і наповнення бази даних, розробка back-end частини вебзастосунку	27.04.2026 – 07.05.2026
10	Вибір технологій реалізації проєкту, розробка front-end частини вебзастосунку	08.05.2026 – 18.05.2026
11	Тестування розробленого програмного продукту, усунення виявлених дефектів	19.05.2026 – 23.05.2026
12	Підготовка матеріалів третього розділу роботи	24.05.2026 – 27.05.2026
13	Оформлення пояснювальної записки	28.05.2026 – 02.06.2026

Дата видачі завдання: «30» березня 2026р.

Студент: _____ / Є. Г. Кісельов /

Керівник роботи: _____ / О. Г. Рибальченко /

РЕФЕРАТ

ВЕБЗАСТОСУНОК, ПОТОКОВЕ ВІДТВОРЕННЯ МУЗИКИ, МУЗИЧНИЙ СЕРВІС, ПЛЕЙЛИСТИ, АУДІОКОНТЕНТ, REACT, NODE.JS, TYPESCRIPT

Пояснювальна записка: 74 с., 1 табл., 6 дод., 42 рис., 20 джерел.

Метою роботи є створення вебзастосунку «RedCloud», призначеного для потокового відтворення музики з інтелектуальною системою рекомендацій та інноваційними ШІ-функціями. Система повинна забезпечувати користувачам можливість зручного прослуховування аудіокомпозицій через вебпереглядач, організації власної музичної бібліотеки та формування персональних плейлистів. Розроблений сервіс має спростити доступ до музичного контенту та оптимізувати процес його пошуку і відтворення.

Об’єкт проєктування – вебзастосунок, що надає сучасний, інтуїтивно зрозумілий та адаптивний інтерфейс для взаємодії з музичним контентом, а також забезпечує ефективне управління аудіофайлами, плейлистами та особистою медіатекою користувача.

У даній кваліфікаційній роботі виконано дослідження програмних та інтерфейсних рішень, що використовуються у сфері потокових музичних сервісів, проведено аналіз існуючих аналогів та визначено їх основні переваги й недоліки. Проведено дослідження досвіду та побажань користувачів (UX), результати якого стали основою для обґрунтування функціоналу та проєктування інтерфейсу застосунку. На основі отриманих результатів розроблено вебзастосунок для онлайн-відтворення музики.

У процесі реалізації основну увагу приділено створенню функціональної серверної частини, що забезпечує обробку запитів користувачів, взаємодію з базою даних, управління інформацією про музичні композиції, виконавців, альбоми та плейлисти, а також передачу аудіоконтенту клієнтській частині. Реалізовано систему персоналізованих рекомендацій на основі AI-ембедингів з перемикачем між стандартним та експериментальним режимами, генерацію обкладинок плейлиста через ШІ з підтримкою власного запиту, транскрипцію

та лайв-переклад текстів пісень, а також генерацію обкладинок до пісень та плейлистів. Доступ до ШІ-функцій розмежовано системою підписки з відповідним візуальним блокуванням для користувачів без преміум-плану. Водночас реалізовано зручний та адаптивний інтерфейс користувача, спроєктований на основі дослідження досвіду та побажань цільової аудиторії, який дозволяє виконувати пошук музики, переглядати інформацію про треки й виконавців, керувати плейлистами та відтворювати аудіо безпосередньо у вебпереглядачі.

ABSTRACT

WEB APPLICATION, MUSIC STREAMING, MUSIC SERVICE, PLAYLISTS, AUDIO CONTENT, REACT, NODE.JS, TYPESCRIPT

Thesis in 74 p., 1 tab., 6 fig., 42 app., 20 references.

The purpose of this work is to develop the "RedCloud" web application, designed for music streaming with an intelligent recommendation system and innovative AI-powered features. The system must provide users with a convenient way to listen to audio tracks via a browser, organize their own music libraries, and create personal playlists. The developed service is intended to simplify access to music content and optimize the process of searching and playback.

The object of design is a web application that provides a modern, intuitive, and responsive interface for interacting with music content, as well as ensures efficient management of audio files, playlists, and the user's personal media library.

In this qualification work, a study of software and interface solutions used in the field of music streaming services was conducted, alongside an analysis of existing analogues to identify their main advantages and disadvantages. A user experience (UX) research study was carried out, the results of which formed the basis for justifying the application's functionality and designing its interface. Based on the results obtained, a web application for online music playback was developed.

During the implementation process, primary attention was paid to creating a functional server-side that handles user requests, interacts with the database, manages information about music tracks, artists, albums, and playlists, and ensures the delivery of audio content to the client-side. A personalized recommendation system based on AI embeddings was implemented, featuring a toggle between standard and experimental modes, along with AI-powered playlist cover generation with custom prompt support, as well as song lyrics transcription and live translation. Access to AI features is restricted by a subscription system with corresponding visual blocking for users without a premium plan. Simultaneously, a user-friendly and responsive interface was implemented, designed on the basis of research into

the experience and preferences of the target audience, allowing users to search for music, view information about tracks and artists, manage playlists, and play audio directly in the browser.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві цифрові технології відіграють важливу роль у повсякденному житті людей. Однією з найбільш поширених сфер використання вебтехнологій є доступ до мультимедійного контенту, зокрема музики. Завдяки розвитку інтернету, хмарних технологій та вебплатформ користувачі отримали можливість прослуховувати музичні композиції онлайн без необхідності їх попереднього завантаження на пристрій. Такий підхід забезпечує швидкий доступ до великої кількості аудіоконтенту та створює зручні умови для його організації й відтворення через вебпереглядач. Водночас стрімкий розвиток технологій штучного інтелекту відкриває нові можливості для музичних сервісів: інтелектуальні системи рекомендацій на основі векторних ембедингів, автоматична генерація обкладинок, транскрипція та лайв-переклад текстів пісень стають реальними інструментами, що суттєво підвищують цінність платформи для користувача.

З розвитком вебтехнологій та сучасних інструментів програмування з'являється можливість створювати функціональні вебзастосунки, які забезпечують зручну взаємодію користувача з мультимедійними сервісами. Музичні потокові платформи дозволяють не лише відтворювати композиції у режимі онлайн, а й керувати персональною музичною бібліотекою, здійснювати пошук треків, переглядати інформацію про виконавців і альбоми, а також формувати власні плейлисти. Проте для створення конкурентоспроможного продукту недостатньо лише реалізувати стандартний набір функцій – необхідно враховувати реальні потреби та очікування цільової аудиторії. Саме тому дослідження користувацького досвіду (UX) є ключовим етапом розробки, що дозволяє обґрунтовано приймати рішення щодо функціоналу та інтерфейсу застосунку і забезпечує його конкурентоспроможність на ринку потокових сервісів.

Метою даної роботи є створення вебзастосунку RedCloud для потокового відтворення музики на основі дослідження потреб та побажань цільової аудиторії. Для досягнення поставленої мети необхідно провести UX-

дослідження користувацького досвіду, проаналізувати існуючі програмні та інтерфейсні рішення у сфері потокових музичних сервісів, дослідити доцільність впровадження інноваційних функцій (зокрема, системи рекомендацій на основі AI-ембедингів, ШІ-генерації обкладинок до пісень і плейлистів, транскрипції текстів пісень та лайв-перекладу), спроєктувати архітектуру застосунку та реалізувати його функціональні можливості. Серверна частина застосунку буде розроблена з використанням платформи Node.js та мови програмування TypeScript і відповідатиме за обробку запитів користувачів, взаємодію з базою даних та передачу аудіоконтенту клієнтській частині. Клієнтський інтерфейс буде реалізовано за допомогою бібліотеки React, що дозволить створити зручний, адаптивний та інтерактивний вебінтерфейс для взаємодії з системою.

Об'єктом дослідження є процеси організації доступу до музичного контенту у вебсередовищі та досвід і побажання користувачів музичних стрімінгових сервісів, а предметом дослідження – методи та технології розробки вебзастосунків для потокового відтворення аудіо з урахуванням результатів UX-дослідження. Практичне значення роботи полягає у створенні функціонального вебзастосунку, який демонструє можливості використання сучасних вебтехнологій для реалізації музичних сервісів та може бути основою для подальшого розвитку подібних систем.

1. АНАЛІЗ СУЧАСНОГО СТАНУ МУЗИЧНИХ СТРІМІНГОВИХ СЕРВІСІВ ТА ОБҐРУНТУВАННЯ АКТУАЛЬНОСТІ РОБОТИ

1.1. Актуальний стан професійної області

У сучасних умовах стрімкого розвитку інформаційних технологій значна частина цифрового контенту споживається через мережу Інтернет. Однією з найбільш популярних категорій такого контенту є музика. Традиційні способи зберігання та відтворення аудіофайлів все більше поступаються місцем потоковим сервісам, які дозволяють користувачам отримувати доступ до музичних композицій у режимі реального часу без необхідності їх завантаження на пристрій. Такий підхід забезпечує зручність використання, швидкий доступ до великої кількості контенту та можливість прослуховування музики на різних пристроях.

Сфера потокових музичних сервісів активно розвивається та характеризується високою конкуренцією між платформами, що надають доступ до аудіоконтенту через вебінтерфейси та мобільні застосунки. Подібні системи надають користувачам широкий набір функціональних можливостей, зокрема пошук музичних композицій, перегляд інформації про виконавців та альбоми, створення і редагування плейлистів, а також організацію персональної музичної бібліотеки. Використання сучасних вебтехнологій дозволяє забезпечити швидку роботу сервісів, масштабованість системи та зручну взаємодію користувачів із контентом.

У процесі розробки подібних вебзастосунків широко застосовуються сучасні технології клієнтської та серверної розробки. Клієнтська частина зазвичай реалізується за допомогою популярних JavaScript-фреймворків та бібліотек, що дозволяють створювати динамічні та інтерактивні користувацькі інтерфейси. Серверна частина забезпечує обробку запитів користувачів, роботу з базами даних, управління аудіоконтентом та передачу необхідної інформації клієнтському застосунку через програмні інтерфейси. Важливими аспектами при створенні таких систем є ефективна організація потокового

відтворення аудіо, оптимізація продуктивності та забезпечення стабільності роботи сервісу.

Таким чином, розробка вебзастосунків для потокового відтворення музики є актуальним напрямом у сфері програмної інженерії. Використання сучасних технологій веброзробки дозволяє створювати функціональні, масштабовані та зручні у використанні музичні сервіси, що забезпечують ефективну взаємодію користувачів із аудіоконтентом. Разом із тим конкурентоспроможність подібних платформ значною мірою визначається відповідністю інтерфейсу та функціоналу реальним побажанням цільової аудиторії. Проведення досліджень користувацького досвіду (UX) є необхідною умовою для прийняття обґрунтованих рішень щодо проєктування інтерфейсу та пріоритизації функцій, що безпосередньо впливає на конкурентоспроможність розробленого вебзастосунку.

1.2. Аналіз існуючих аналогів

У сфері потокового відтворення музики існує значна кількість програмних сервісів, що надають користувачам можливість прослуховувати аудіоконтент через мережу Інтернет. Такі платформи забезпечують доступ до великої бібліотеки музичних композицій, дозволяють здійснювати пошук за виконавцями, альбомами та треками, а також створювати та керувати власними плейлистами. Розвиток подібних систем сприяв формуванню сучасних стандартів у сфері організації музичного контенту та взаємодії користувачів із мультимедійними сервісами.

Одним із найпоширеніших потокових музичних сервісів є Spotify, який надає користувачам доступ до великої бібліотеки музичних композицій, можливість створення персональних плейлистів та зручний інтерфейс для керування музичною бібліотекою. Платформа підтримує роботу на різних пристроях та забезпечує синхронізацію даних між ними. Сервіс вирізняється широким функціоналом і високою продуктивністю, проте значна частина можливостей доступна лише в межах платної підписки [1].

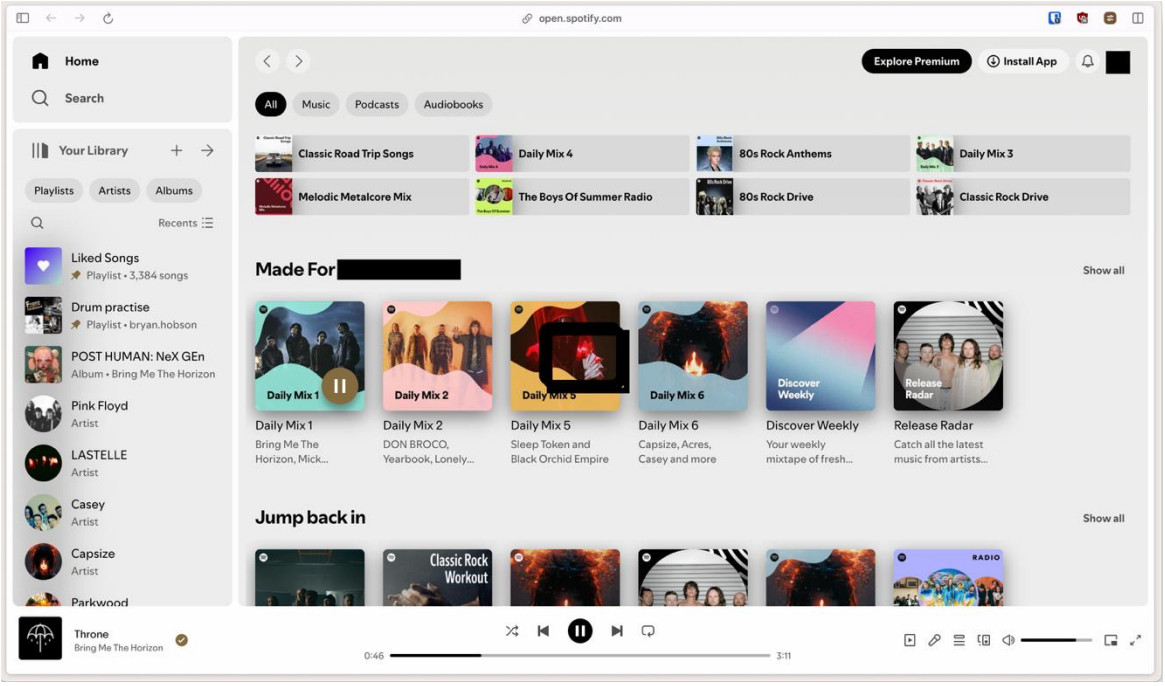

Рисунок 1.1 – Інтерфейс Spotify

Ще одним популярним сервісом є Apple Music, який інтегрований в екосистему пристроїв компанії Apple. Платформа пропонує доступ до великої музичної бібліотеки, зручний інтерфейс та підтримку відтворення музики на різних пристроях. Однак використання сервісу найефективніше саме в межах екосистеми Apple, що може обмежувати зручність для користувачів інших платформ [2].

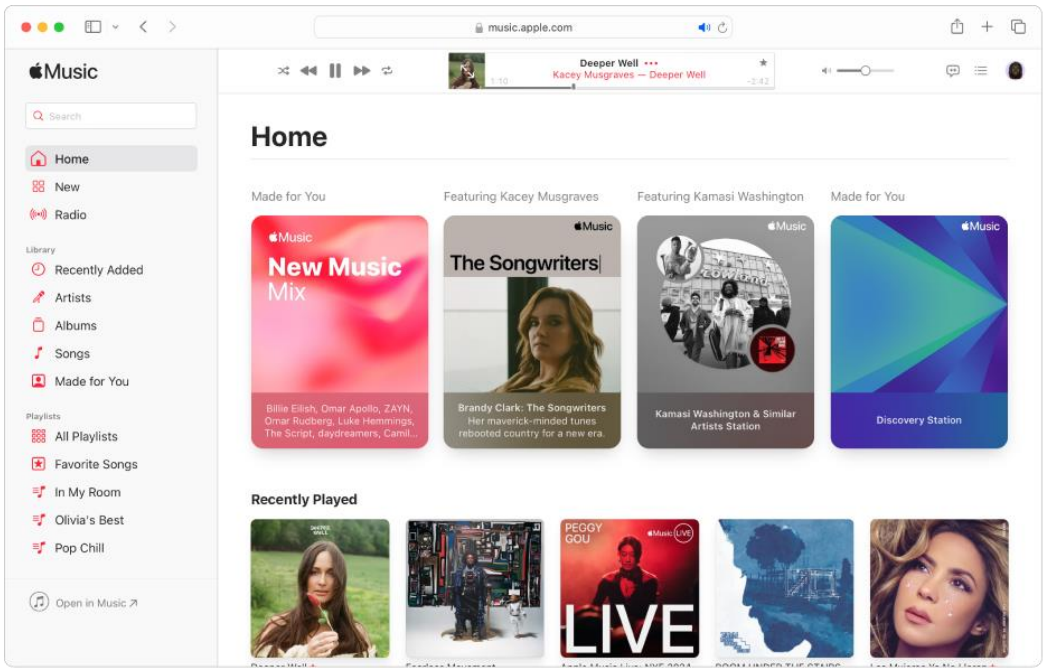

Рисунок 1.2 – Інтерфейс Apple Music

Серед відомих потокових сервісів також варто відзначити YouTube Music, який дозволяє прослуховувати музичні композиції та переглядати музичні відео через вебінтерфейс або мобільний застосунок. Основною перевагою сервісу є велика кількість доступного контенту та інтеграція з платформою YouTube. Разом з тим функціональність сервісу значною мірою залежить від підписки, а безкоштовна версія має певні обмеження [3].

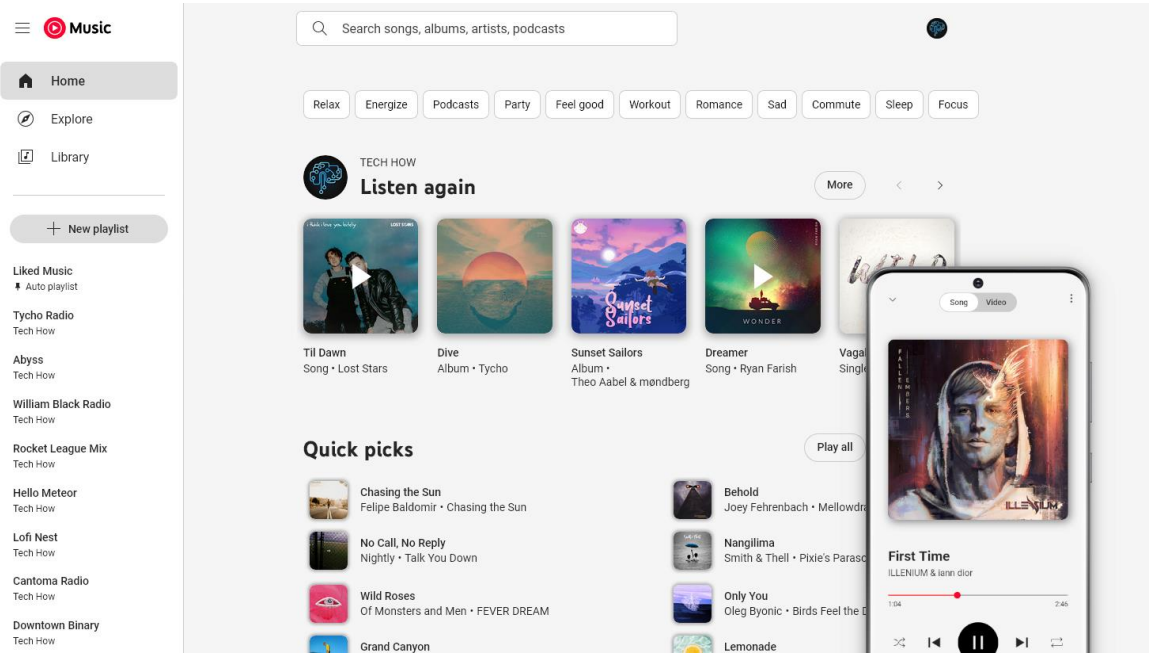

Рисунок 1.3 – Інтерфейс YouTube Music

Аналіз існуючих аналогів показує, що сучасні потокові музичні сервіси пропонують широкий набір функцій для взаємодії з аудіоконтентом, однак часто мають обмеження, пов’язані з платними можливостями або прив’язаністю до певних екосистем. Це створює передумови для розробки власних вебзастосунків, які реалізують основні функції потокового відтворення музики та демонструють використання сучасних технологій веброзробки. Одним із таких рішень є вебзастосунок RedCloud, що забезпечує доступ до музичного контенту через браузер та надає можливість керування музичними композиціями і плейлистами користувача. Результати порівняльного аналізу сучасного стану музичних потокових сервісів, їхніх функціональних та інтерфейсних рішень наведено в Таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз функціоналу потокових музичних сервісів

	Параметр	Наше рішення	Spotify	Apple Music	YouTube Music
Базовий функціонал	Кількість доступних треків	Велика	Велика	Велика	Велика
	Якість звуку	Висока	Висока	Висока	Висока
	Кросплатформеність	Так	Так	Так	Так
	Офлайн прослуховування	Ні	Так	Так	Так
Соціальні функції	Соціальна інтеграція	Так	Так	Так	Так
	Підтримка української мови	Так	Так	Ні	Так
Контент	Ексклюзивний контент	Ні	Так	Так	Так
	Відео контент	Ні	Ні	Ні	Так
Рекомендації	Персоналізовані рекомендації	Так	Так	Так	Так
	Експериментальні AI-рекомендації	Так	Ні	Ні	Ні
ШІ-функції	Лайв-переклад пісень	Так	Ні	Ні	Ні
	Транскрипція текстів пісень	Так	Ні	Ні	Ні
	ШІ-генерація обкладинок	Так	Ні	Ні	Ні

1.3. Формулювання актуальності та мети роботи

У сучасному цифровому середовищі доступ до мультимедійного контенту, зокрема музики, дедалі частіше здійснюється через мережу Інтернет. Розвиток вебтехнологій, високошвидкісних мереж та хмарних сервісів сприяв широкому поширенню потокових платформ, які забезпечують можливість прослуховування музики без необхідності її попереднього завантаження. Користувачі очікують швидкого доступу до аудіоконтенту, зручного керування музичною бібліотекою та можливості взаємодії із

сервісом через сучасний інтуїтивно зрозумілий інтерфейс. У зв'язку з цим розробка вебзастосунків для потокового відтворення музики із використанням сучасних технологій клієнтської та серверної розробки є актуальним завданням у галузі програмної інженерії.

Метою даної роботи є розробка вебзастосунку RedCloud для потокового відтворення музики, спроєктованого на основі дослідження досвіду та побажань цільової аудиторії. Застосунок має забезпечувати зручний доступ до аудіоконтенту через вебпереглядач, підтримувати пошук музичних композицій, перегляд інформації про виконавців і альбоми, створення та керування плейлистами, а також реалізовувати інноваційні функції – ШІ-генерацію обкладинок для пісень та плейлистів, транскрипцію текстів пісень, лайв-переклад текстів пісень та інтелектуальну систему рекомендацій на основі AI-ембедингів, доцільність яких визначається результатами UX-дослідження. Реалізація застосунку передбачає використання Node.js з TypeScript для серверної частини та React для клієнтського інтерфейсу.

Актуальність роботи обумовлена:

1. Стрімке зростання популярності потокових сервісів для доступу до музичного контенту через мережу Інтернет.
2. Потреба користувачів у швидкому та зручному доступі до великої кількості аудіокомпозицій без необхідності завантаження файлів на пристрій.
3. Розвиток сучасних вебтехнологій, що дозволяють створювати продуктивні та масштабовані мультимедійні вебзастосунки.
4. Необхідність забезпечення зручного управління музичними колекціями, плейлистами та аудіоконтентом через інтуїтивно зрозумілий вебінтерфейс.
5. Актуальність використання сучасних інструментів веброзробки, таких як Node.js, TypeScript та React, для створення інтерактивних і функціональних вебсервісів.

6. Необхідність врахування досвіду та побажань цільової аудиторії при проєктуванні інтерфейсу музичного сервісу для забезпечення його конкурентоспроможності.

7. Перспективність використання технологій штучного інтелекту у музичних сервісах, зокрема для генерації обкладинок, транскрипції текстів пісень, лайв-перекладу та побудови системи рекомендацій на основі AI-ембедингів.

1.4. Визначення завдань кваліфікаційної роботи

Завданнями даної кваліфікаційної роботи є комплексне вирішення питань, пов'язаних із створенням функціонального вебзастосунку для потокового відтворення музики на основі дослідження потреб користувачів. До основних завдань відноситься дослідження досвіду та побажань цільової аудиторії щодо функціоналу музичного сервісу, зокрема перевірка доцільності впровадження інноваційних можливостей – ШІ-генерації обкладинок для пісень та плейлистів, транскрипції текстів пісень, лайв-перекладу та системи рекомендацій на основі AI-ембедингів. Окрему увагу приділено аналізу сучасного стану музичних потокових сервісів, їхніх функціональних та інтерфейсних рішень, а також вибору технологій веброзробки для реалізації масштабованої серверної частини та інтерактивного клієнтського інтерфейсу

Завдання роботи:

1. Провести UX-дослідження потреб та побажань цільової аудиторії щодо функціоналу музичного стрімінгового сервісу.

2. Провести аналіз сучасних потокових музичних сервісів та визначити їх основні функціональні можливості та інтерфейсні рішення.

3. Визначити на основі результатів дослідження доцільність впровадження інноваційних функцій – ШІ-генерації обкладинок для пісень та плейлистів, транскрипції текстів пісень, лайв-перекладу та системи персоналізованих рекомендацій на основі AI-ембедингів.

4. Дослідити технології та інструменти, що використовуються для розробки вебзастосунків потокового відтворення музики.

5. Спроєктувати архітектуру вебзастосунку RedCloud.

6. Реалізувати серверну частину системи з використанням Node.js та TypeScript, що забезпечує обробку запитів користувачів та взаємодію з базою даних.

7. Розробити клієнтську частину застосунку за допомогою React.

8. Реалізувати функціональність пошуку музики, відтворення аудіоконтенту та керування плейлистами користувачів.

9. Забезпечити зручний та адаптивний користувацький інтерфейс для взаємодії із системою.

2. ДОСЛІДЖЕННЯ КОРИСТУВАЦЬКОГО ДОСВІДУ ТА ОБҐРУНТУВАННЯ ФУНКЦІОНАЛУ

2.1. Методологія дослідження

Для обґрунтування проєктних рішень вебзастосунку RedCloud було проведено комплексне дослідження користувацького досвіду (UX), що включало два етапи: онлайн-анкетування та глибинні інтерв'ю з представниками цільової аудиторії. Вибір саме такої комбінації методів зумовлений їхніми властивостями та відповідає підходу до людиноцентричного проєктування, за яким рішення мають ґрунтуватись на реальних потребах користувачів [20]. Анкетування дозволяє отримати кількісні дані та статистично значущі тенденції [7,8]. Глибинні інтерв'ю розкривають мотивацію, контекст та нюанси поведінки користувачів, які неможливо виявити за допомогою закритих питань [9].

Онлайн-анкетування проводилось за допомогою інструменту Google Forms та охоплювало питання щодо частоти використання сервісів, важливості конкретних функцій, ставлення до ШІ-інструментів, інтерфейсних уподобань та готовності до платної підписки. Анкета містила як закриті питання з варіантами відповідей, так і питання з оцінкою за шкалою, що дозволило кількісно виміряти ставлення респондентів до конкретних функцій [7].

Глибинні інтерв'ю проводились у форматі структурованої бесіди та охоплювали теми соціальної взаємодії в сервісі, алгоритмів рекомендацій, інтеграції ШІ, синхронізації між пристроями та немузичного контенту. Такий формат дозволив отримати розгорнуті відповіді, виявити приховані потреби та зафіксувати конкретні сценарії використання з реального досвіду респондентів [9].

Анкетування охопило 81 респондента, з яких 76 (93,8%) є активними користувачами музичних стрімінгових сервісів. Глибинні інтерв'ю проводились з 5 представниками цільової аудиторії віком від 20 до 27 років.

Географія учасників – Україна. Відбір респондентів здійснювався за критерієм регулярного використання стрімінгових сервісів, що забезпечило релевантність отриманих даних для проєктування RedCloud.

Усі зібрані дані були систематизовані та проаналізовані у розрізі ключових тематичних блоків: використання сервісів, функціональні потреби, ставлення до інновацій та інтерфейсні уподобання. Результати аналізу стали безпосередньою основою для прийняття проєктних рішень щодо функціоналу та інтерфейсу вебзастосунку RedCloud.

2.2. Визначення цільової аудиторії та профілі користувачів

За результатами анкетування сформовано детальний портрет цільової аудиторії вебзастосунку RedCloud та виявлено ключові характеристики поведінки користувачів музичних стрімінгових сервісів.

З 81 опитаного 76 (93,8%) є активними користувачами музичних стрімінгових сервісів – саме їхні відповіді стали основою подальшого аналізу. Вікова група – переважно від 17 до 22 років (73,1%), що відповідає основній демографічній групі споживачів стрімінгових сервісів. Саме ця аудиторія є найбільш технологічно залученою, готовою до використання нових функцій та схильною до регулярного споживання цифрового медіаконтенту.

Переважна більшість респондентів – 64,5% – слухають музику щодня, ще 28,9% – кілька разів на тиждень. Жоден із учасників не зазначив рідшої частоти використання. Це підтверджує, що цільова аудиторія RedCloud – це не випадкові, а постійні користувачі, для яких музичний сервіс є частиною щоденного цифрового середовища.

Найпопулярнішим сервісом виявився YouTube Music – його використовують 73,7% респондентів. На другому місці Spotify (60,5%), далі SoundCloud (21,1%) та Apple Music (9,2%). Ця картина свідчить про те, що серед молодої аудиторії YouTube Music є домінуючою платформою, тоді як Spotify залишається важливим орієнтиром при проєктуванні інтерфейсу та

функціоналу RedCloud – саме з цими двома сервісами користувачі будуть несвідомо порівнювати новий застосунок.

Цільова аудиторія вже сформувала стійкі звички та очікування щодо музичних сервісів – вони добре розуміють можливості існуючих платформ і здатні критично оцінити нові рішення. Це підтверджує необхідність уважного підходу до проєктування інтерфейсу та функціоналу RedCloud відповідно до виявлених потреб.

2.3. Результати анкетування

Аналіз відповідей на питання про важливість функцій виявив чітку ієрархію потреб цільової аудиторії [8]. Найвищий пріоритет отримало створення плейлистів – 88,2% (67 з 76 респондентів) вважають цю функцію важливою. Це підтверджує, що користувачі активно структурують свій музичний досвід і персоналізована організація контенту є базовою очікуваною можливістю будь-якого сучасного сервісу.

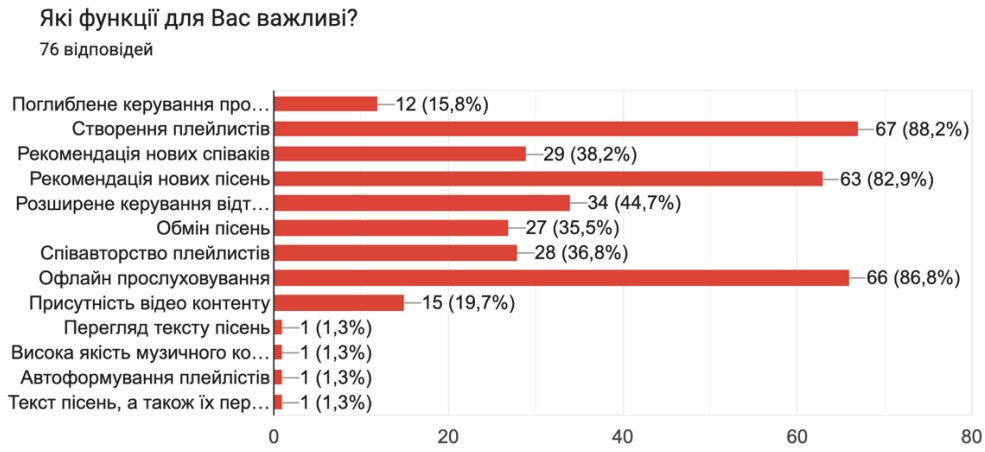

Рисунок 2.1 – Діаграма важливості функцій для користувачів

Офлайн-прослуховування обрали 86,8% респондентів – це другий за значущістю показник після плейлистів, що підкреслює критичну важливість доступу до музики за відсутності інтернет-з'єднання. Рекомендації нових пісень важливі для 82,9%, розширене керування відтворенням – для 44,7%, співавторство плейлистів та обмін піснями – для 36,8% та 35,5% відповідно.

Відео-контент є важливим лише для 19,7%, що узгоджується з позиціонуванням RedCloud як саме музичного, а не відео-сервісу.

Переважна більшість респондентів – 90,8% – задоволені своїм основним сервісом, і лише 9,2% – незадоволені. Це встановлює високу планку очікувань: користувачі вже звикли до якісних інтерфейсів і будь-яке відхилення від цього стандарту буде сприйматись негативно.

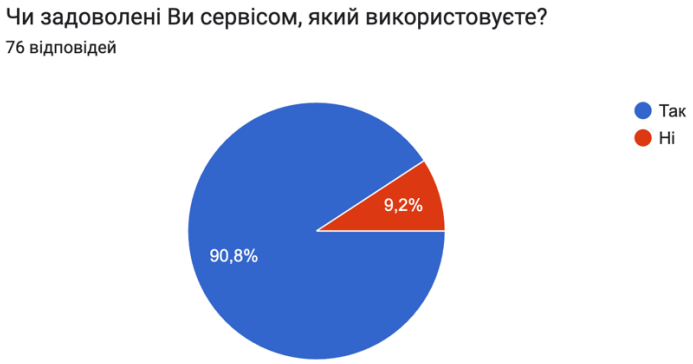

Рисунок 2.2 – Оцінка зручності використання сервісу

Аналіз важливості алгоритмічних рекомендацій музичного сервісу показав, що 86,8% вважають їх важливими. При цьому 72,9% відчували «зацикленість» алгоритмів – сервіс перестає пропонувати щось справді нове. Аж 92,9% підтримали б перемикач між персоналізованим та випадковим режимами рекомендацій – це найбільш однозначний результат усього анкетування, що безпосередньо обґрунтовує реалізований у RedCloud перемикач режимів.

Результати щодо соціальних функцій виявились неоднозначними. Більшість респондентів – 61,4% – вважають за краще ділитись музикою через зовнішні месенджери. Водночас 20% хотіли б вбудований чат, а 14,3% – лише тимчасовий чат під час спільного прослуховування. Щодо активності друзів – 61,4% цікаво бачити що слухають друзі в реальному часі, проте 20% вважають це зайвою функцією. Ці результати свідчать про те, що соціальні функції мають бути присутніми, але не нав'язливими, з можливістю вимкнення.

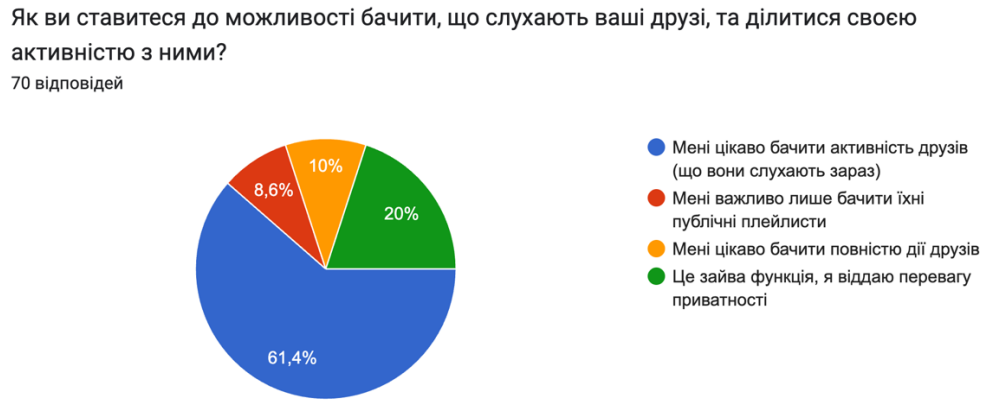

Рисунок 2.3 – Ставлення до системи друзів

Рисунок 2.4 – Ставлення до вбудованого чату

Щодо підписки – більшість респондентів, 53,9%, мають платну підписку з позитивним ставленням, ще 3,9% – з негативним. Серед тих, хто не має підписки, головними причинами є: «достатньо безкоштовного функціоналу» (20,9%), «занадто дорого» (20,9%) та «не бачу суттєвої різниці» (19,4%). Це вказує на важливість чіткого позиціонування переваг платної підписки в RedCloud – користувач має розуміти, що саме він отримає, перейшовши на преміум.

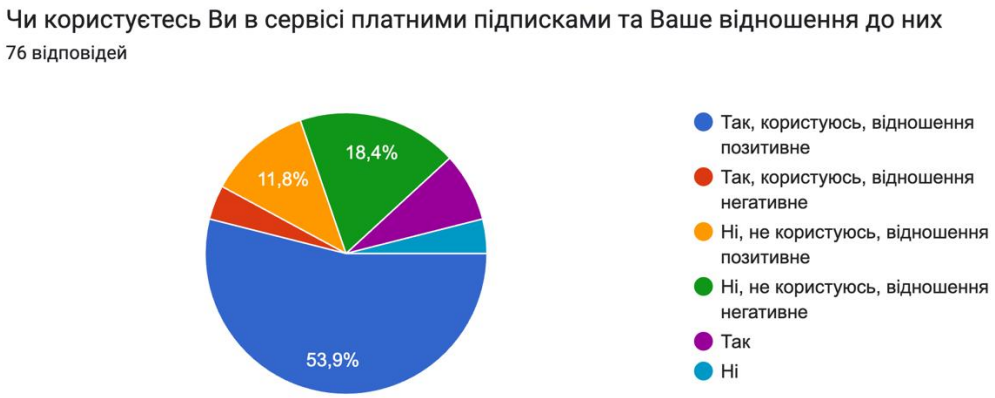

Рисунок 2.5 – Розподіл типів підписок серед респондентів

Окремим блоком анкетування стало вивчення ставлення до інноваційних функцій RedCloud. Щодо транскрипції тексту пісні за відсутності офіційного – позитивно ставляться 45,7% респондентів, нейтрально – 35,7%, негативно – лише 18,6%. Таким чином, переважна більшість – понад 80% – не заперечує проти цієї функції. Ідею ШІ-генерації обкладинок для плейлистів оцінили позитивно – 55,7% респондентів поставили 4 або 5 балів з 5, середня оцінка становить 3,49. Щодо ШІ-каверів думки розділились майже порівну: 50,7% підтримують, 49,3% – ні. Перемикач між персоналізованим та випадковим режимами рекомендацій підтримали 92,9% – це найбільш однозначний результат серед усіх інноваційних функцій.

Рисунок 2.6 – Ставлення респондентів до ШІ-генерації тексту пісні за відсутності офіційного

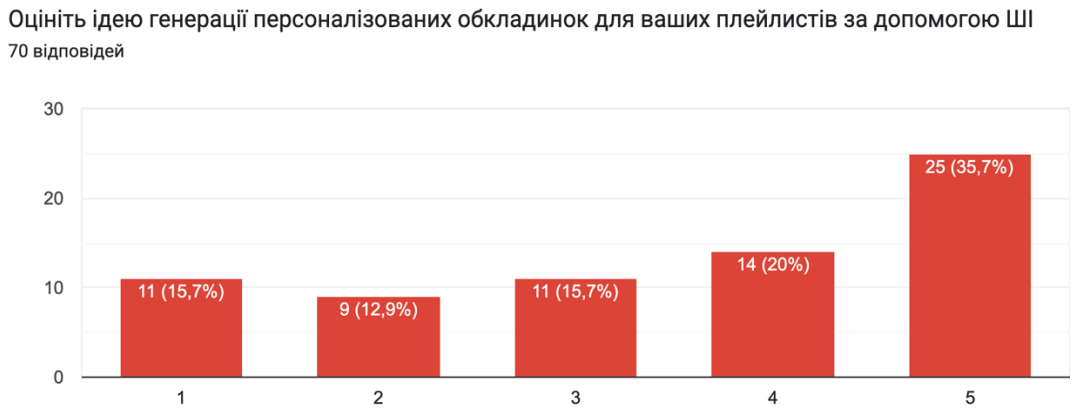

Рисунок 2.7 – Оцінка респондентами ідеї ШІ-генерації персоналізованих обкладинок для плейлистів

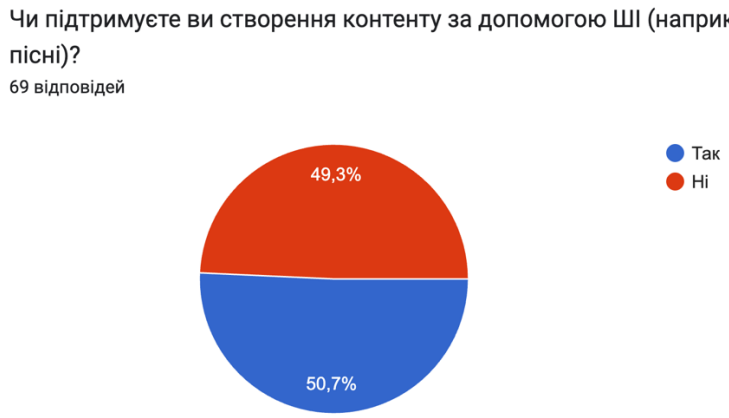

Рисунок 2.8 – Підтримка респондентами створення контенту за допомогою ШІ

2.4. Результати глибинних інтерв'ю

Глибинні інтерв'ю проводились з п'ятьма учасниками – Віталієм, Богданом, Анастасією, Віталієм та Валерією – віком від 20 до 27 років. Усі є активними користувачами Spotify або YouTube Music з досвідом від 6 місяців до 8 років. Формат структурованої бесіди охоплював шість тематичних блоків: щоденне використання, пошук нової музики, офлайн та технічні можливості, соціальні функції, штучний інтелект у сервісі та больові точки [9].

Усі п'ять учасників слухають музику щодня – вдома, в транспорті та під час навчання чи роботи. Тип підписки розділився: троє мають платну, двоє – безкоштовну. Головна причина відмови від платної підписки – відчуття, що

наявного функціоналу достатньо, хоча обидва відзначили конкретні незручності: обмеження на перемотування треків та реклама в навушниках у нічний час. Найчастіше використовувані функції – пошук, власні плейлисти та рекомендації.

Нову музику більшість учасників знаходить переважно через TikTok та Reels, а не через алгоритми сервісу. Усі п'ять відчували зацикленість рекомендацій. При цьому всі висловили бажання більшого контролю над алгоритмом.

Офлайн-режимом не користуються троє з п'яти учасників – переважно через наявність стабільного мобільного інтернету або відсутність платної підписки. Збереження черги відтворення при зміні пристрою більшість вважає приємним, але не критичним. Дистанційним керуванням між пристроями не користується жоден з учасників.

Обмін музикою серед учасників відбувається виключно через зовнішні месенджери – Telegram або Instagram, жоден не використовує вбудовані інструменти сервісу для цього. Вбудований чат всі вважають зайвим. Водночас деякі учасники цікавляться музичними смаками друзів та іноді відкривають нову музику саме через них, що підтверджує доцільність функції перегляду активності друзів як опціональної можливості.

Ставлення учасників до інноваційних ШІ-функцій виявилось переважно позитивним. Транскрипцію тексту пісень підтримали всі п'ять – один учасник залишився нейтральним. До AI-каверів ставлення більш стримане. Суперечність між стриманим ставленням в інтерв'ю та рівним розподілом в анкеті пояснюється різним формулюванням питань: в анкеті йшлося про підтримку ШІ-контенту загалом, тоді як в інтерв'ю обговорювались конкретні правові та якісні аспекти.

Серед больових точок існуючих сервісів учасники виділили: агресивну рекламу у безкоштовних версіях, обмеження на перемотування треків без підписки, зациклені рекомендаційні алгоритми та неможливість

синхронізувати власні треки між пристроями. Ці спостереження були враховані при формуванні функціональних вимог до RedCloud.

2.5. Висновки UX-дослідження та обґрунтування проєктних рішень

Проведене дослідження дозволило сформулювати конкретні висновки, що безпосередньо вплинули на проєктні рішення вебзастосунку RedCloud.

Лайв-переклад текстів пісень, ШІ-генерація обкладинок та система рекомендацій з перемикачем між режимами «For You» та «Discover» на головній сторінці, що дозволяє користувачу обирати між персоналізованими та експериментальними рекомендаціями на основі AI-ембедингів отримали підтримку аудиторії і є обґрунтованими відмінними рисами RedCloud на ринку. Ці функції закривають реальні незадоволені потреби, виявлені в дослідженні, та не мають прямих аналогів у досліджуваних конкурентах.

Дослідження підтвердило доцільність системи друзів та обміну плейлистами, проте соціальні функції мають бути опціональними – з можливістю вимкнення публічної активності. Вбудований чат не є пріоритетною потребою для цільової аудиторії і може бути розглянутий у наступних ітераціях продукту.

Більшість користувачів готова платити за підписку за умови чіткого розуміння переваг преміум-рівня. Модель з обмеженою, але функціональною безкоштовною версією та розширеними можливостями для платних користувачів відповідає очікуванням аудиторії і підтверджена досвідом конкурентів.

Таким чином, проведене UX-дослідження забезпечило доказову базу для всіх ключових рішень щодо функціоналу та інтерфейсу RedCloud. Кожна реалізована функція має обґрунтування, що спирається на реальні потреби та висловлені побажання цільової аудиторії, а не лише на технічні можливості або аналіз конкурентів.

2.6. Проєктування інтерфейсу на основі результатів дослідження

На основі результатів UX-дослідження та порівняльного аналізу інтерфейсних рішень конкурентів розроблено вайрфрейми основних сторінок застосунку RedCloud – спрощені схематичні зображення, що відображають структуру інтерфейсу та розташування ключових елементів. Основна увага приділяється не візуальному оформленню, а логіці побудови інтерфейсу: розміщенню навігації, блоків контенту, кнопок керування та інтерактивних елементів. Кожне інтерфейсне рішення обґрунтоване виявленими потребами користувачів та усталеними патернами галузі.

Розміщення навігаційної панелі ліворуч відповідає патерну Spotify і YouTube Music та забезпечує постійний доступ до бібліотеки без переходу між сторінками. Оскільки цільова аудиторія сформувала стійкі звички саме на цих платформах, відхилення від звичного розташування навігації підвищило б поріг освоєння застосунку. Фіксований плеєр у нижній частині екрана є галузевим стандартом усіх трьох досліджених конкурентів і знижує поріг освоєння для нових користувачів. Розміщення блоку рекомендацій у центральній частині головної сторінки з перемикачем режимів безпосередньо відповідає виявленому у дослідженні запиту на більший контроль над алгоритмами – цю потребу висловили 92,9% респондентів анкетування та всі п'ять учасників глибинних інтерв'ю [9]. Картковий інтерфейс для відображення плейлистів є усталеним патерном у Spotify та Apple Music і забезпечує швидке візуальне розпізнавання без необхідності читати назву, а також є логічним місцем для інтеграції кнопки ШІ-генерації обкладинки. Мінімалістичний дизайн сторінки авторизації з підтримкою входу через Google OAuth відповідає підходу провідних конкурентів і знижує бар'єр реєстрації.

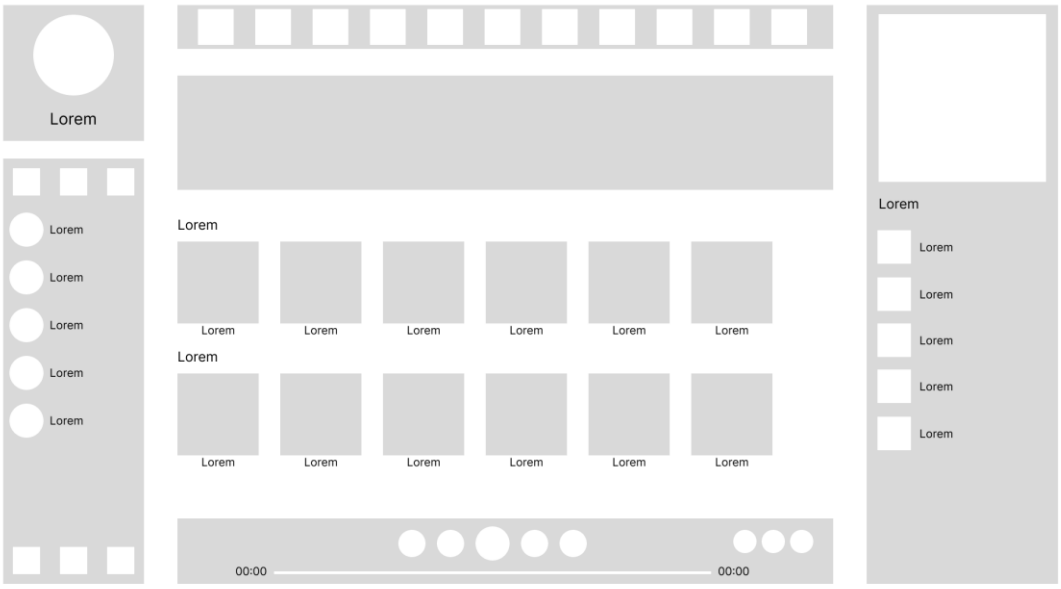

Рисунок 2.9 – Вайрфрейм головної сторінки музичного стримінгового сервісу

Головна сторінка є центральною частиною інтерфейсу та призначена для взаємодії користувача з контентом. У лівій частині розташовано панель профілю, що містить аватар користувача та навігаційне меню (перехід до бібліотеки, друзів, налаштувань тощо). У верхній частині сторінки знаходиться горизонтальний блок із плейлистами. У центральній частині розміщено основний контент: банер або рекомендаційний блок, а також декілька горизонтальних списків із рекомендованими піснями, плейлистами чи альбомами. Праворуч знаходиться панель поточного відтворення, яка містить інформацію про активну пісню та чергу відтворення. В нижній частині сторінки розташовано плеєр, який включає елементи керування відтворенням (play/pause, перемотування), індикатор часу та додаткові функції взаємодії з треком.

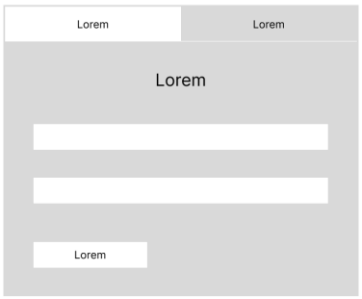

Рисунок 2.10 – Вайрфрейм сторінки авторизації

Сторінка авторизації містить форму введення облікових даних користувача, зокрема email та пароль. Передбачена можливість входу через сторонні сервіси, зокрема Google. Інтерфейс побудовано таким чином, щоб забезпечити швидкий і зручний доступ до системи. На великих екранах форма може розташовуватися поруч із ілюстративним блоком, тоді як на мобільних пристроях – по центру екрана.

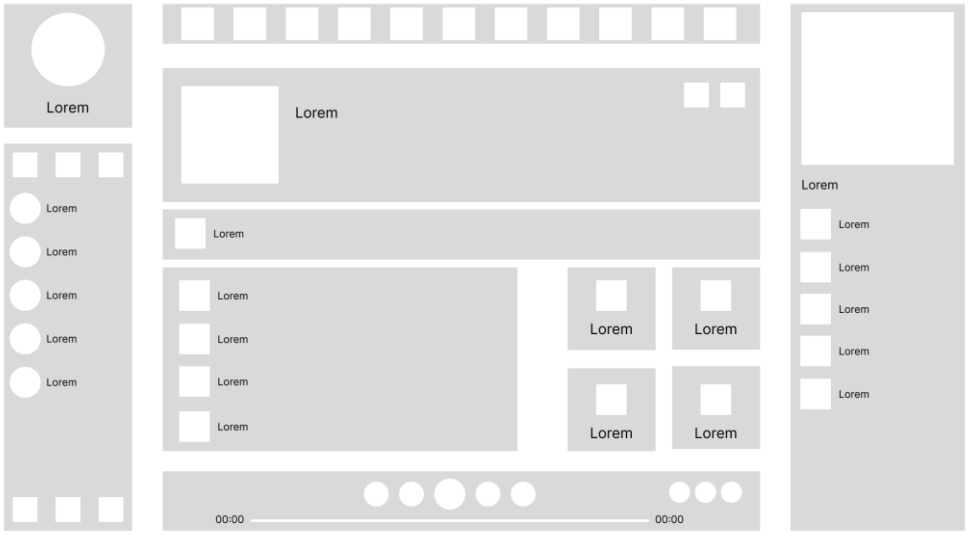

Рисунок 2.11 – Вайрфрейм сторінки профілю

Сторінка профілю призначена для перегляду та редагування інформації про користувача. У верхній частині відображаються основні дані – аватар, ім'я, додаткові відомості. Нижче розміщені списки активностей користувача,

зокрема створені плейлисти, вподобані пісні або статистика прослуховувань. Також передбачені елементи керування для редагування профілю та зміни налаштувань.

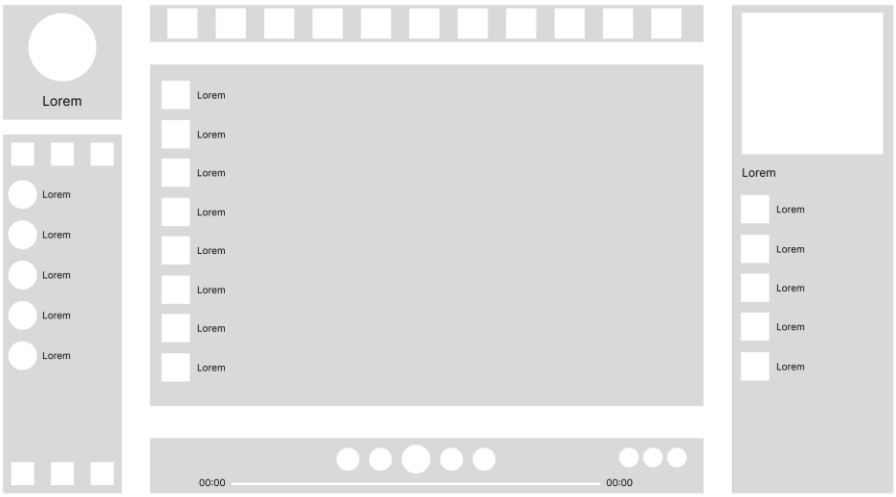

Рисунок 2.12 – Вайрфрейм сторінки списку плейлистів

Сторінка плейлистів призначена для відображення музичної бібліотеки користувача. У центральній частині інтерфейсу представлено набір карток плейлистів, кожна з яких містить обкладинку та назву, що забезпечує швидке візуальне розпізнавання. У верхній частині сторінки розташовані елементи керування – кнопки створення нового плейлиста та сортування. Вибір конкретного плейлиста зі списку відкриває окрему сторінку з детальною інформацією та переліком пісень, що до нього входять.

Представлені вайрфрейми є результатом синтезу двох джерел: даних UX-дослідження, що підтвердили очікування цільової аудиторії щодо навігації, відтворення та організації контенту, та аналізу інтерфейсних рішень провідних конкурентів. Такий підхід забезпечує відповідність інтерфейсу реальним потребам користувачів ще на етапі проєктування та знижує ризик помилкових рішень під час розробки.

3. ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ

3.1. Розробка математичної моделі

За результатами UX-дослідження підтверджено доцільність впровадження ШІ-функцій та системи підписки. Для формалізації ключових процесів, пов'язаних з їх реалізацією, розроблено математичну модель системи.

Математична модель системи описує формалізовані залежності між основними параметрами вебзастосунку RedCloud та дозволяє обґрунтувати ключові проєктні рішення. У межах даного проєкту доцільно виділити чотири основні підсистеми: обробка AI-операцій, система рекомендацій на основі векторних ембедингів, валідація прослуховування музичного контенту та управління підпискою.

Оскільки платформа інтегрує зовнішні AI-сервіси, кожна операція має певну вартість. Для обліку витрат використовується модель, що базується на кількості оброблених токенів.

Вартість однієї операції визначається як:

$$C = T_{in} \cdot c_{in} + T_{out} \cdot c_{out} \tag{3.1}$$

- T_{in}, T_{out} – кількість вхідних та вихідних токенів;
- c_{in}, c_{out} – вартість одного токена.

У випадку генерації зображень використовується спрощена модель:

$$C_{img} = N \cdot c_{img}, \tag{3.2}$$

де N – кількість згенерованих зображень.

Загальна вартість використання AI-функцій користувачем визначається як сума вартостей усіх виконаних операцій:

$$C_{total} = \sum_{i=1}^{k} C_i \tag{3.3}$$

Система експериментальних рекомендацій базується на векторному представленні музичного контенту. Кожному треку відповідає вектор ознак у n-вимірному просторі.

Схожість між двома треками визначається через косинусну подібність:

$$sim(i,j) = \frac{\overrightarrow{V_i} * \overrightarrow{V_j}}{\left|\overrightarrow{V_i}\right| * \left|\overrightarrow{V_j}\right|} \tag{3.4}$$

де значення sim ∈ [-1, 1], при цьому sim = 1 означає повну схожість.

Рейтинг рекомендації треку j для користувача u визначається як:

$$R(u,j) = \alpha * sim\_content(i,j) + (1 - \alpha) * sim\_social(u,j) \tag{3.5}$$

де:

- sim_content – схожість на основі AI-ембедингів контенту
- sim_social – схожість на основі соціальної активності (лайки, жанри)
- $\alpha \in [0,1]$ – коефіцієнт, що визначає режим рекомендацій (при $\alpha = 1$ – повністю експериментальний режим, при $\alpha = 0$ – персоналізований)

Саме параметр α відповідає перемикачу режимів рекомендацій в інтерфейсі застосунку.

Для коректного обліку статистики прослуховувань використовується модель, що враховує відсоток прослуханого треку.

$$P = \frac{D_{listened}}{D_{total}} \times 100\% \tag{3.6}$$

- $D_{listened}$ – фактичний час прослуховування;

- D_{total} – загальна тривалість треку.

Прослуховування вважається валідним, якщо:

$$P \geq P_{threshold} \tag{3.7}$$

- $P_{threshold} = 60\% \div 70\%$.

Такий підхід дозволяє уникнути фіктивних переглядів і підвищує точність аналітики.

Життєвий цикл підписки описується як часовий інтервал:

$$t_{end} = t_{start} + \Delta \tag{3.8}$$

- t_{start}– момент початку підписки;
- Δ– тривалість підписки.

Стан підписки визначається умовою:

$$Status = \begin{cases} Active, & t_{now} \leq t_{end} \\ Expired, & t_{now} > t_{end} \end{cases} \tag{3.9}$$

Для пробного періоду застосовується аналогічна модель із фіксованою тривалістю.

Запропонована математична модель дозволяє формалізувати ключові процеси вебзастосунку RedCloud, зокрема облік використання AI-функцій, систему рекомендацій на основі векторних ембедингів, валідацію прослуховувань та управління підпискою. Це забезпечує обґрунтованість прийнятих рішень і підвищує ефективність функціонування системи.

3.2. Визначення користувацьких ролей і рівнів доступу до системи

Визначення типів користувачів є важливим етапом проєктування системи, оскільки саме від цього залежить структура інтерфейсу, доступний функціонал та обмеження для кожної категорії користувачів. Це безпосередньо впливає на зручність використання, безпеку та ефективність роботи системи загалом.

У системі RedCloud використовується рольова модель доступу, яка передбачає розподіл користувачів на окремі ролі з відповідними правами. Такий підхід дозволяє централізовано керувати доступом до функціоналу системи та забезпечує гнучкість у подальшому розширенні.

У межах вебзастосунку RedCloud доцільно виділити чотири основні ролі користувачів.

1. Гість (Guest) – це роль для неавторизованого користувача, який має доступ лише до публічної частини системи. До доступного функціоналу належать: перегляд сторінок авторизації та реєстрації, ініціація входу через Google OAuth 2.0, скидання пароля, перегляд списку жанрів, тарифних планів та виконання пошуку по каталогу пісень. Будь-які спроби доступу до захищених ресурсів завершуються помилкою HTTP 401 (Unauthorized);

2. Авторизований користувач (User) – базова роль, яка автоматично призначається після реєстрації. Користувач отримує доступ до повного функціоналу платформи: управління профілем, завантаження та редагування музичного контенту, створення та редагування плейлистів, взаємодія з контентом (лайки, обране), пошук користувачів, отримання сповіщень у реальному часі, а також використання AI-функцій (генерація обкладинок для плейлистів та пісень, транскрипція та переклад текстів), офлайн-прослуховування музики. Додатково користувач може оформлювати та керувати підпискою;

3. Оператор (Operator) – розширена роль, яка призначається адміністратором. Окрім можливостей звичайного користувача, оператор має

доступ до управління довідниковими даними, зокрема каталогом жанрів (створення, редагування, видалення). Роль орієнтована на модерацію та підтримку структури контенту;

4. Адміністратор (Admin) – користувач із найвищим рівнем доступу, який має повний контроль над системою. До його повноважень входить управління користувачами (перегляд, редагування, зміна ролей), блокування та розблокування облікових записів, а також перегляд аналітичної інформації щодо активності користувачів.

Окрім рольової моделі доступу, у системі реалізовано додатковий механізм контролю у вигляді системи блокування користувачів. Заблокований користувач, незалежно від ролі, отримує помилку HTTP 403 (Forbidden) при спробі доступу до захищених ресурсів. Перевірка статусу блокування здійснюється на рівні middleware після автентифікації.

Також реалізовано контроль доступу на основі власності ресурсів: користувач може змінювати лише ті об’єкти, які були створені ним (наприклад, власні плейлисти або завантажені пісні).

Таким чином, використання рольової моделі з чітким розподілом прав доступу забезпечує ефективну організацію функціоналу системи, підвищує рівень безпеки та дозволяє адаптувати інтерфейс під різні категорії користувачів.

3.3. Створення структурної схеми та діаграми прецедентів

На рисунку 3.1 представлено структурну схему вебзастосунку RedCloud, яка відображає основні компоненти системи та їх взаємодію. Діаграма демонструє логічну архітектуру програмного продукту, розподілену на три ключові рівні: інтерфейс користувача, серверну логіку та рівень зберігання даних.

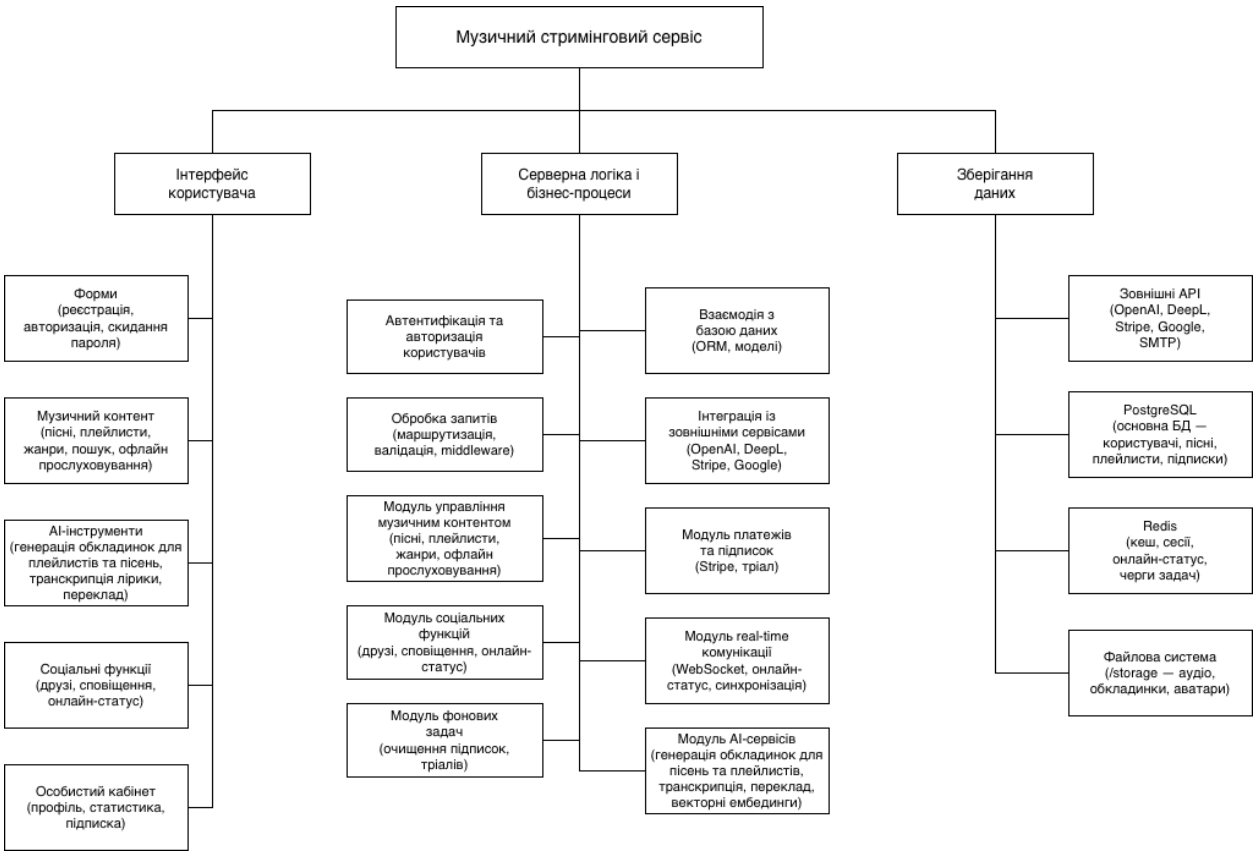

Рисунок 3.1 – Структурна схема вебзастосунку «RedCloud»

У верхній частині діаграми зображено загальний контекст системи – «Музичний стримінговий сервіс», що об’єднує всі підсистеми в єдину платформу.

Ліва частина діаграми відповідає за інтерфейс користувача. До цього рівня належать компоненти, з якими безпосередньо взаємодіє користувач: форми автентифікації (реєстрація, вхід, скидання пароля), модулі роботи з музичним контентом (пісні, плейлисти, жанри, пошук, офлайн-прослуховування), AI-інструменти (генерація обкладинок для пісень та плейлистів, транскрипція та переклад текстів), соціальні функції (друзі, сповіщення, онлайн-статус), а також особистий кабінет користувача, де доступна інформація про профіль, статистику та підписку.

Центральна частина діаграми відображає серверну логіку та бізнес-процеси, які забезпечують обробку всіх запитів користувача. До неї входять модулі автентифікації та авторизації, що відповідають за перевірку доступу, а також обробка HTTP-запитів із використанням маршрутизації, валідації та

middleware. Окремо виділено модулі управління контентом, соціальними функціями, платежами та підписками, а також модуль AI-сервісів, який реалізує генерацію обкладинок для пісень та плейлистів, транскрипцію та переклад текстів пісень, а також система персоналізованих рекомендацій з перемикачем між режимами «For You» та «Discover»

Крім того, серверна частина включає модуль взаємодії з базою даних через ORM, що забезпечує зручну роботу з даними, та модуль інтеграції із зовнішніми сервісами, такими як сервіси штучного інтелекту, платіжні системи та служби автентифікації. Реалізовано також модуль real-time комунікації, який забезпечує оновлення даних у реальному часі, та модуль фонових задач для виконання періодичних операцій, зокрема обробки підписок.

Права частина діаграми представляє рівень зберігання даних. Основним сховищем виступає реляційна база даних, у якій зберігаються дані користувачів, музичного контенту, плейлистів та підписок. Для оптимізації роботи системи використовується кешування та управління сесіями за допомогою Redis. Зберігання мультимедійних файлів (аудіо, зображення) реалізується через файлову систему або спеціалізоване сховище. Також передбачено використання зовнішніх API, що забезпечують додаткову функціональність, зокрема AI-обробку, переклад текстів, платежі та автентифікацію через сторонні сервіси.

Таким чином, представлена діаграма відображає багаторівневу архітектуру вебзастосунку RedCloud, де кожен рівень виконує окрему функцію та взаємодіє з іншими через чітко визначені інтерфейси. Такий підхід забезпечує масштабованість, гнучкість та зручність підтримки системи, що є важливим для сучасних вебзастосунків.

Діаграма варіантів використання відображає функціональні можливості вебзастосунку RedCloud та взаємодію користувачів із системою. Вона дозволяє визначити основні сценарії використання системи, а також ролі користувачів і їх доступ до відповідного функціоналу.

На рисунку 3.2 представлено основні актори системи, серед яких можна виділити неавторизованого користувача (гість), авторизованого користувача, оператора, та адміністратора. Кожен із них має різний рівень доступу до функцій системи.

Гість має можливість виконувати базові дії, зокрема реєстрацію та авторизацію в системі. Після успішної авторизації користувач отримує доступ до розширеного функціоналу платформи.

Авторизований користувач може керувати власним профілем, переглядати та редагувати персональні дані, змінювати налаштування та пароль. Також він має доступ до перегляду контенту, пошуку пісень, прослуховування музики офлайн та роботи з плейлистами. Передбачено можливість взаємодії з контентом, зокрема додавання або видалення пісень з обраного та редагування власних колекцій.

Окремо виділено функціонал роботи з контентом, який включає створення, редагування та перегляд пісень і плейлистів. Додатково система підтримує AI-інструменти, такі як генерація обкладинок для пісень та плейлистів, транскрипція та переклад текстів пісень, а також формування векторних ембедингів для системи рекомендацій.

Соціальна складова реалізована через систему друзів, яка дозволяє додавати та видаляти користувачів зі списку друзів, переглядати їхню музичну активність у реальному часі та онлайн-статус. Також передбачена система підписок, що включає оформлення та скасування підписки для доступу до розширених можливостей платформи.

Оператор є проміжною роллю між звичайним користувачем та адміністратором. Він має всі можливості авторизованого користувача, а також додаткові функції, пов’язані з управлінням контентом. Зокрема, оператор може редагувати музичний контент, керувати жанрами, модерувати пісні та плейлисти, що дозволяє підтримувати структуру та актуальність контенту в системі.

Адміністратор має розширені права доступу, що дозволяють керувати іншими користувачами. До його функцій належить блокування облікових записів та зміна ролей користувачів, що забезпечує контроль та безпеку системи.

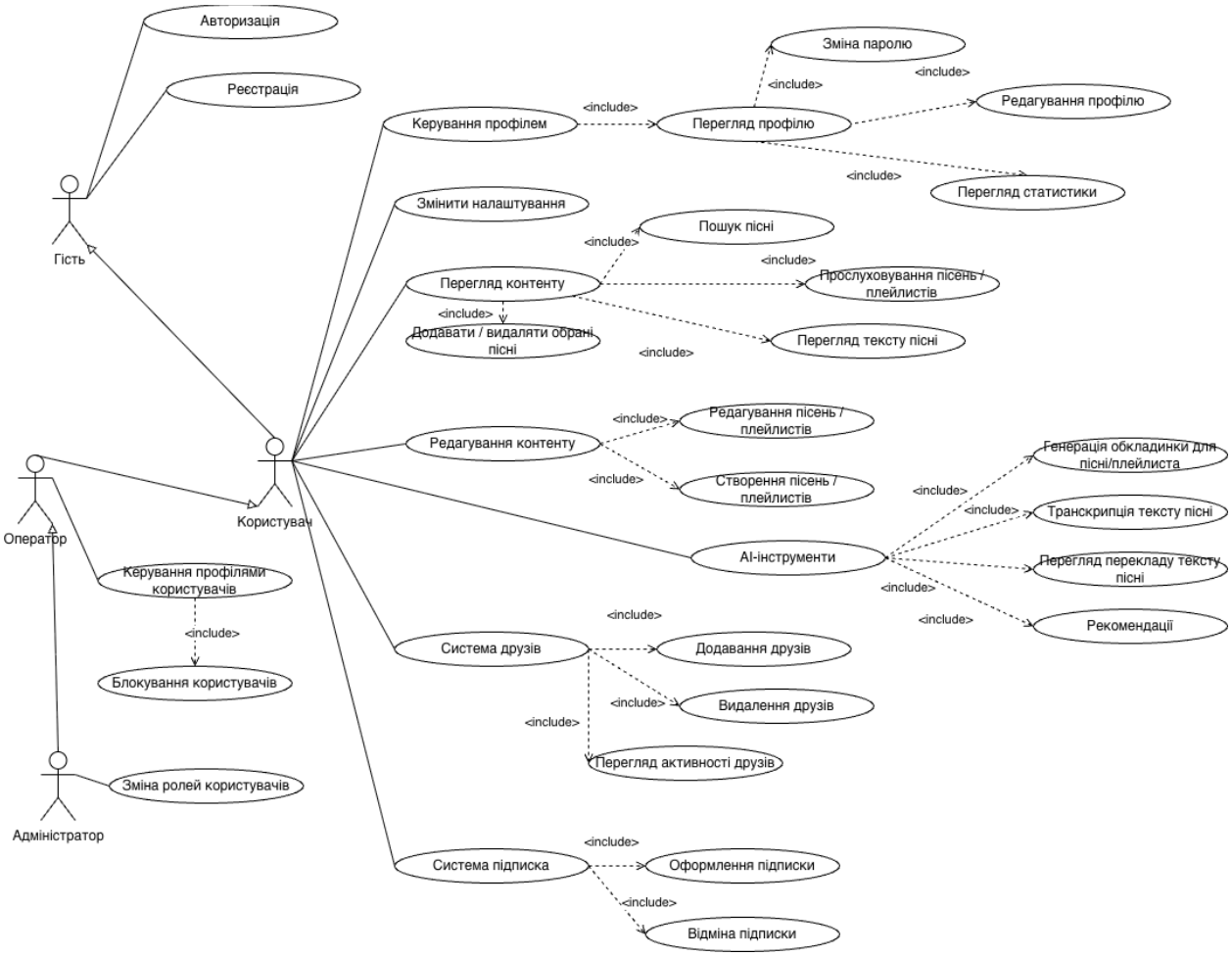

Рисунок 3.2 – Діаграма прецедентів створюваного вебзастосунку

На діаграмі також використано зв’язки типу «include», які показують, що деякі варіанти використання є складовими інших. Наприклад, перегляд контенту включає пошук пісень і їх прослуховування, а редагування профілю може включати зміну пароля.

Таким чином, діаграма варіантів використання наочно демонструє функціональну структуру системи RedCloud, розподіл прав доступу між різними ролями користувачів та взаємозв’язок між основними сценаріями роботи, що є важливим етапом проєктування програмного продукту.

3.4. Розробка функціональної діаграми

Функціональна модель дозволяє структуризувати основні процеси вебзастосунку RedCloud, визначити взаємозв'язки між ними та краще зрозуміти загальну логіку функціонування системи. Використання даної нотації дає змогу декомпозувати складні процеси на більш прості складові та чітко визначити їх взаємодію.

На рисунку 3.3 представлено контекстну діаграму (A-0) вебзастосунку RedCloud. Вона відображає систему на найвищому рівні абстракції у вигляді єдиного функціонального блоку «Управління музичною платформою». На цьому рівні показано взаємодію системи із зовнішнім середовищем.

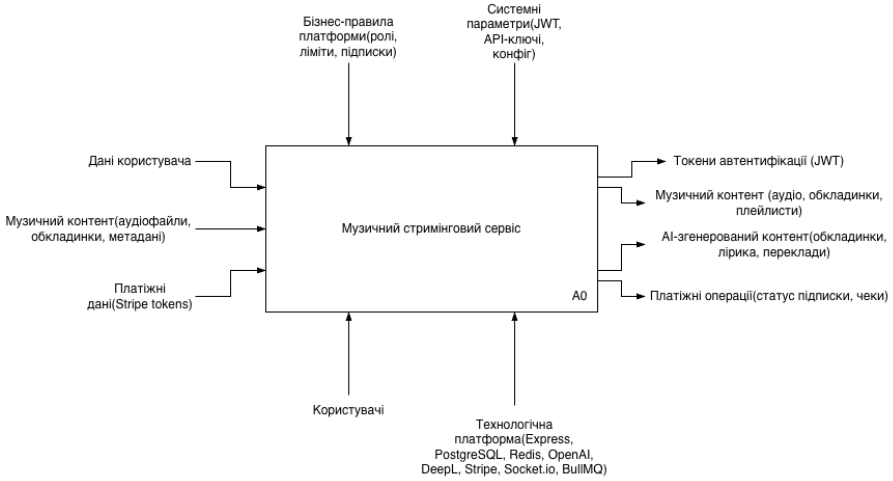

Рисунок 3.3 – Контекстне уявлення вебзастосунку музичного стримінгового сервісу

Вхідними даними системи є запити користувачів, мультимедійні файли (аудіо та зображення), текстова інформація (облікові дані, назви пісень, плейлистів), а також платіжні дані. Результатом роботи системи є JSON-відповіді API, JWT-токени, завантажені файли, сповіщення в реальному часі, AI-згенерований контент, переклади текстів та інформація про стан підписки.

Управління системою здійснюється за рахунок ролей користувачів, механізмів автентифікації та авторизації, правил валідації та бізнес-логіки. В якості механізмів реалізації виступають програмні та технологічні засоби, зокрема серверна частина, база даних, кешування, сторонні API та сервіси обробки даних.

Стрілками ліворуч на діаграмі позначено вхідні дані, праворуч – результати обробки, зверху – керуючі впливи, а знизу – механізми реалізації.

На рисунку 3.4 представлено діаграму першого рівня декомпозиції (A0), яка деталізує основну функцію системи. Загальний процес управління музичною платформою розбитий на сім основних підсистем: автентифікація та авторизація, управління контентом, соціальні функції, AI-сервіси, переклад лірики, управління підписками та real-time комунікація.

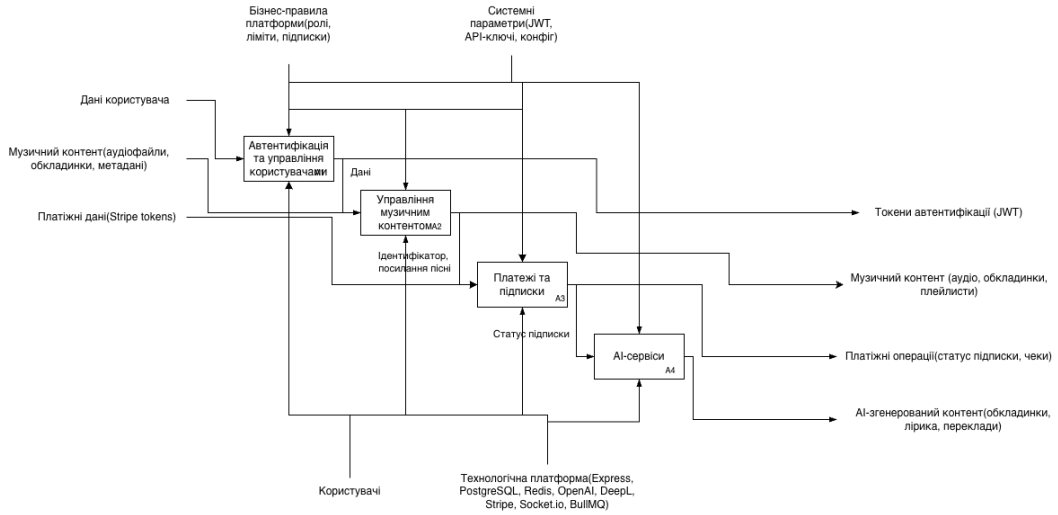

Рисунок 3.4 – Діаграма першого рівня декомпозиції контекстного представлення

Кожен із зазначених блоків виконує окрему функціональну роль у системі. Наприклад, блок автентифікації відповідає за обробку облікових даних користувачів та генерацію токенів доступу, тоді як блок управління контентом забезпечує роботу з музичними файлами, плейлистами та взаємодією користувачів із контентом.

Між блоками існують інформаційні зв’язки. Зокрема, результати автентифікації передаються до інших підсистем у вигляді даних користувача та токенів доступу. Дані контенту можуть використовуватись AI-сервісами для генерації або обробки інформації, а результати цих сервісів повертаються назад у систему. Таким чином забезпечується узгоджена робота всіх компонентів.

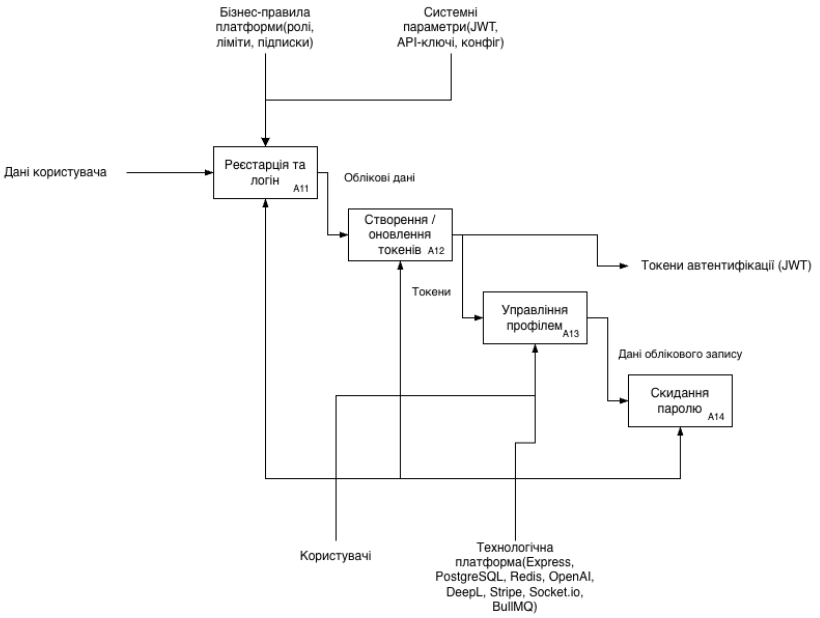

Рисунок 3.5 – Діаграма другого рівня декомпозиції авторизації

На рисунку 3.5 наведено діаграму другого рівня декомпозиції, яка деталізує один із функціональних блоків системи, зокрема підсистему автентифікації та авторизації. На цьому рівні показано окремі процеси, такі як реєстрація користувача, вхід до системи, авторизація через сторонні сервіси, оновлення токенів доступу та скидання пароля.

Кожен підпроцес має свої вхідні та вихідні дані, що дозволяє більш детально зрозуміти логіку роботи підсистеми. Наприклад, процес реєстрації використовує облікові дані користувача, а результатом є створення нового облікового запису та генерація токенів доступу.

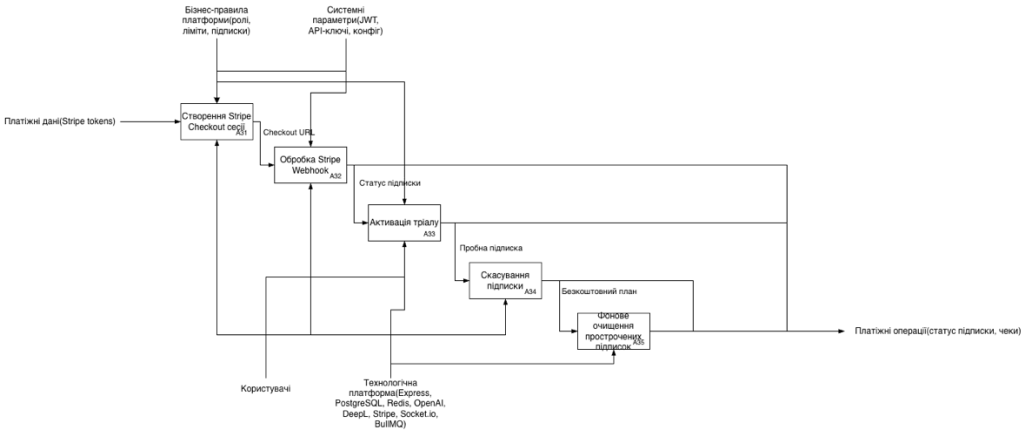

Рисунок 3.6 – Діаграма другого рівня декомпозиції системи оплати

На рисунку 3.6 представлено діаграму другого рівня декомпозиції блоку А3 «Платежі та підписки», яка описує фінансову логіку та механізми

монетизації сервісу RedCloud. Підсистема починається з ініціації Stripe Checkout сесії, де на основі обраного користувачем плану формується унікальний URL для безпечної оплати. Ключовим етапом є автоматизована обробка Webhook-повідомлень від платіжної системи, що дозволяє асинхронно оновлювати статус підписки в базі даних відразу після підтвердження транзакції.

Процес активації тріальних періодів та скасування платних послуг регулюється бізнес-правилами платформи, забезпечуючи автоматичний перехід користувача на безкоштовний план у разі завершення терміну дії Premium-доступу. Для підтримки актуальності даних використовується механізм фонового очищення прострочених підписок через BullMQ, що працює за принципом cron-задач. Технологічно підсистема інтегрована зі Stripe API та використовує PostgreSQL для зберігання фінансової історії, видаючи на виході актуальні статуси доступу та сформовані цифрові чеки.

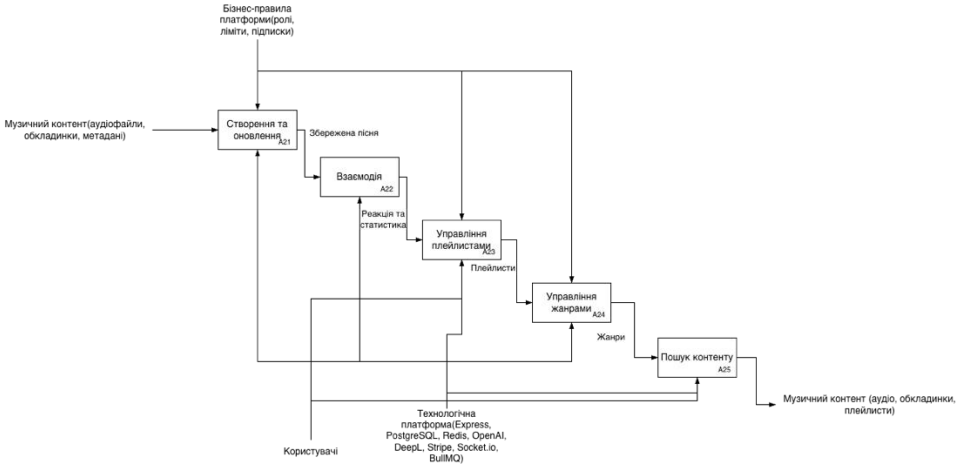

Рисунок 3.7 – Діаграма другого рівня декомпозиції елементів управління контентом

На рисунку 3.7 представлено діаграму другого рівня декомпозиції блоку A2 «Управління контентом», яка деталізує внутрішні операції з мультимедійними даними. Ця підсистема забезпечує повний життєвий цикл музичного контенту, починаючи від фізичного завантаження файлів через Multer-конфігурації до їх логічного структурування в базі даних PostgreSQL. Процес створення та оновлення пісень тісно пов'язаний із заповненням метаданих, що дозволяє системі коректно ідентифікувати авторів та жанри.

Взаємодія користувача з контґентом реалізована через механізми реакцій та збору статистики, що в подальшому впливає на формування персоналізованих плейлистів та черг відтворення. Окремий модуль управління жанрами дозволяє адміністраторам та операторам платформи структурувати бібліотеку, забезпечуючи точність глобального пошуку. Технологічна платформа, що включає Express.js та Objection.js [10,13], гарантує цілісність даних при виконанні CRUD-операцій, а використання Redis прискорює доступ до часто запитуваних списків обраного. Результатом роботи підсистеми є сформований медіа-архів із валідними URL-адресами обкладинок та аудіопотоків, готовий до стрімінгу та подальшої обробки AI-сервісами.

Рисунок 3.8 – Діаграма другого рівня декомпозиції AI-сервісів

На рисунку 3.8 наведено діаграму другого рівня декомпозиції для функціонального блоку A4 «AI-сервіси», що відповідає за автоматизовану обробку та генерацію медіа-контенту. Робота підсистеми ініціюється запитами користувачів на створення візуального або текстового супроводу до музичних композицій. Процес генерації обкладинок для пісень та плейлистів використовує інтеграцію з OpenAI, перетворюючи текстові запити в графічні файли, які автоматично зберігаються в системному сховищі та прив'язуються до відповідних сутностей у БД.

Паралельно з цим блок транскрипції лірики обробляє аудіофайли за допомогою моделі, формуючи текстову основу пісні, яка надалі передається до модуля перекладу. Важливою складовою підсистеми є трекінг активності та витрат, що дозволяє платформі контролювати доступ до AI-функцій залежно від тарифного плану користувача – у безкоштовному плані (Free) AI-інструменти недоступні, тоді як преміум-план (Pro) надає повний доступ до генерації обкладинок, транскрипції та перекладу текстів пісень. Керуючими впливами тут виступають системні параметри та API-ключі, а механізм реалізації базується на фонових задачах BullMQ, що запобігає перевантаженню основного потоку сервера. Виходом підсистеми є комплексний набір AI-згенерованих даних, що підвищує інформативність та привабливість контенту для кінцевого споживача.

3.5. Створення діаграми послідовностей

Діаграма послідовностей застосовується для візуалізації взаємодії між окремими компонентами системи в межах певного сценарію. Вона дає змогу відобразити, які саме повідомлення передаються між елементами системи, у якій послідовності це відбувається та як формується загальна логіка виконання процесу.

Основними складовими такої діаграми є учасники взаємодії, які розміщуються у верхній частині схеми, а також їхні часові лінії, що простягаються вниз і відображають тривалість існування об’єкта в межах сценарію. Кожен учасник відповідає окремому елементу системи, наприклад користувачу, клієнтському застосунку, серверу чи зовнішньому сервісу. Обмін інформацією між ними подається у вигляді стрілок, що ілюструють виклики, відповіді та інші типи взаємодій, зберігаючи при цьому чітку хронологію виконання дій.

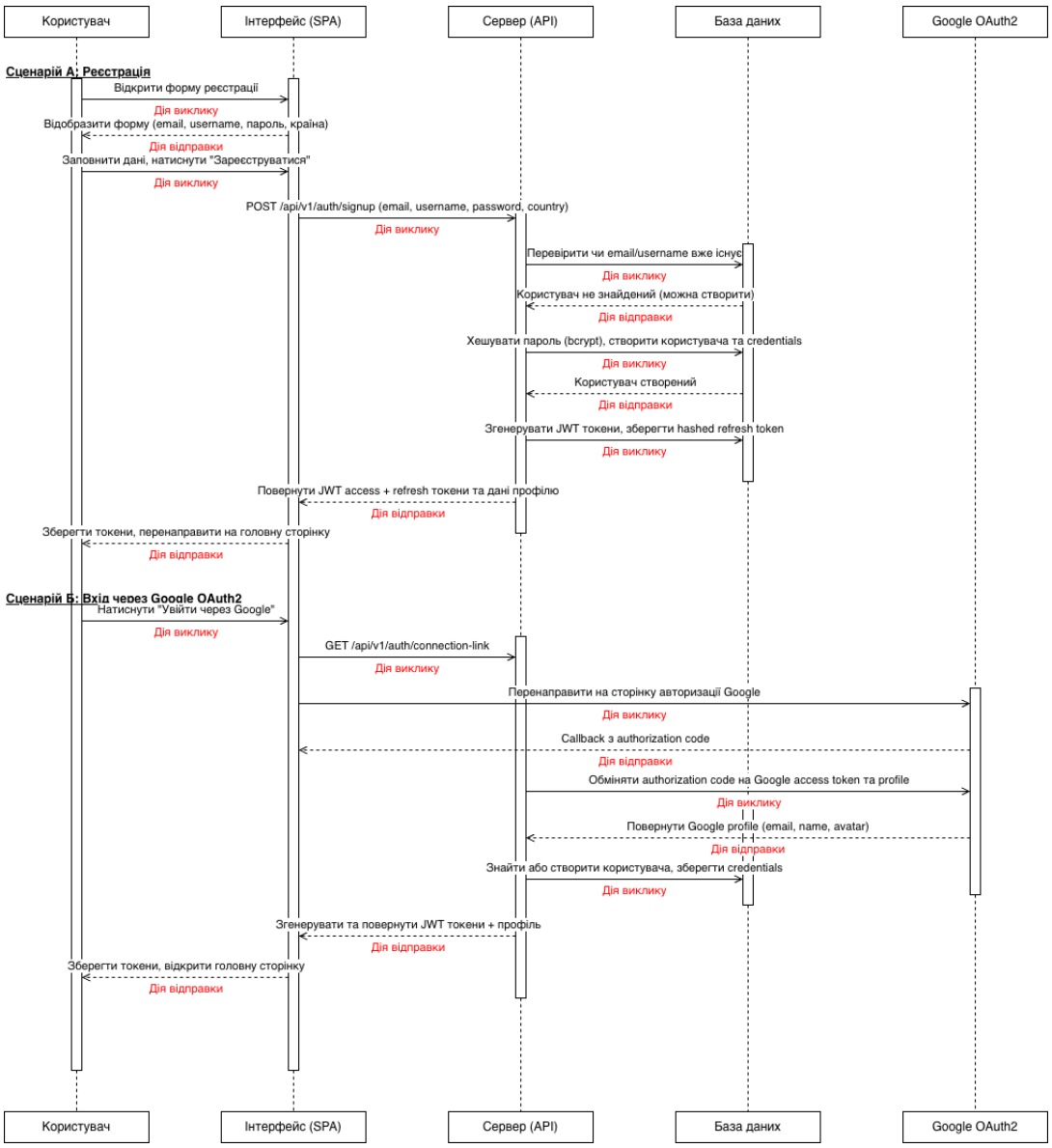

Рисунок 3.9 – Діаграма послідовностей авторизації у створюваному застосунку

На рисунку 3.9 представлена діаграма послідовностей процесу реєстрації користувача та авторизації через Google у вебзастосунку RedCloud. У діаграмі беруть участь такі основні компоненти: користувач, інтерфейс користувача (SPA), серверна частина (API), база даних та зовнішній сервіс Google OAuth2.

У першій частині діаграми показано процес стандартної реєстрації. Користувач відкриває форму реєстрації, вводить необхідні дані та надсилає запит через клієнтський інтерфейс до серверу. Сервер перевіряє наявність користувача в базі даних, після чого у разі відсутності створює новий

обліковий запис, виконує хешування пароля та генерує токени доступу. Отримані токени повертаються клієнту, де зберігаються для подальшої автентифікації.

Друга частина діаграми описує альтернативний сценарій авторизації через Google OAuth2. Користувач ініціює вхід через зовнішній сервіс, після чого відбувається перенаправлення на сторінку авторизації Google. Після успішного входу Google повертає код авторизації, який обробляється сервером для отримання профілю користувача. На основі отриманих даних система знаходить або створює користувача в базі даних та генерує JWT-токени, які передаються клієнту.

Завдяки наявності часових ліній на діаграмі можна простежити повну послідовність дій, а також визначити моменти взаємодії між клієнтською частиною, сервером, базою даних та зовнішнім API. Це дозволяє краще зрозуміти внутрішню логіку роботи системи та взаємозв’язки між її компонентами.

Таким чином, діаграма послідовностей дає можливість ще на етапі проєктування виявити потенційні проблеми у взаємодії компонентів системи, оптимізувати логіку обробки запитів та забезпечити коректну реалізацію процесів автентифікації та авторизації у вебзастосунку RedCloud.

3.6. Побудова моделі «сутність-зв’язок» бази даних вебзастосунку

Наведена діаграма 3.10 відображає структуру бази даних вебзастосунку музичного стримінгового сервісу RedCloud і демонструє взаємозв’язки між основними сутностями системи. Такий тип діаграми (ER-діаграма) використовується для проєктування та візуалізації логічної моделі даних, що дозволяє чітко зрозуміти, які саме дані зберігаються в системі, як вони пов’язані між собою та які обмеження накладаються на їх використання.

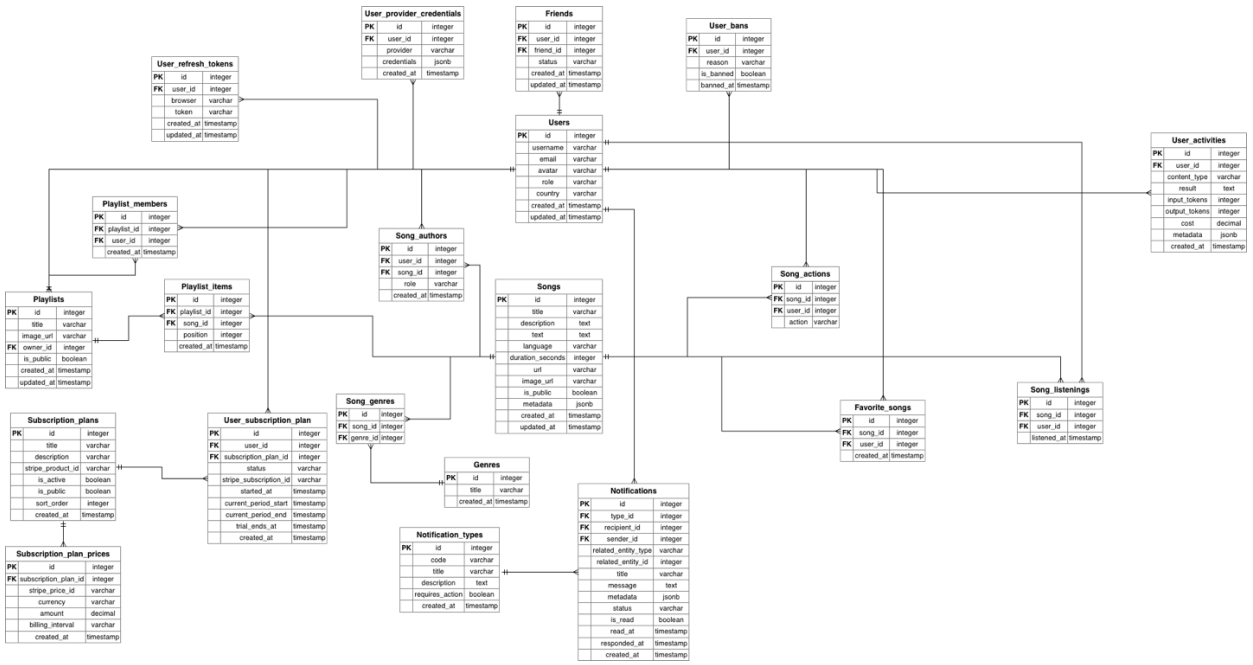

Рисунок 3.10 – ER-діаграма бази даних створюваного вебзастосунку

У центрі діаграми знаходиться таблиця Users, яка є ключовою сутністю системи. Вона містить основну інформацію про користувачів (username, email, avatar, роль, країна тощо) та пов’язана з більшістю інших таблиць. Це пояснюється тим, що практично всі процеси в системі (створення контенту, взаємодії, підписки, AI-активність) виконуються від імені користувача.

Модуль управління контентом представлений таблицями Songs, Playlists, Playlist_items, Song_authors, Song_genres та Genres. Таблиця Songs зберігає інформацію про музичні треки, включаючи метадані, URL файлів і тривалість. Зв’язки з таблицями жанрів та авторів дозволяють реалізувати гнучку систему категоризації та співавторства. Плейлисти реалізовані через окремі таблиці, що дозволяє зберігати порядок пісень і підтримувати багатьох користувачів у межах одного плейлиста.

Соціальні функції системи відображені через таблиці Friends та Notifications. Таблиця Friends реалізує механізм дружби між користувачами зі статусами (pending, accepted тощо), а Notifications дозволяє зберігати сповіщення різних типів із прив’язкою до відправника, отримувача та пов’язаних сутностей.

Взаємодія користувачів із контентом представлена таблицями Song_actions, Favorite_songs та Song_listenings. Вони відповідають за лайки/дизлайки, додавання в обране та фіксацію фактів прослуховування, що є важливим для аналітики та рекомендацій.

Підсистема підписок реалізована через таблиці Subscription_plans, Subscription_plan_prices та User_subscription_plan. Вони дозволяють зберігати інформацію про доступні тарифи, їх вартість і статус підписки кожного користувача, включаючи тріальний період і інтеграцію зі Stripe.

Безпека та автентифікація забезпечуються таблицями User_refresh_tokens та User_provider_credentials. Вони зберігають refresh-токени, інформацію про браузери та дані зовнішніх провайдерів (наприклад, Google OAuth), що дозволяє реалізувати як локальну, так і сторонню авторизацію.

Окремо виділено механізм контролю доступу через таблицю User_bans, яка дозволяє блокувати користувачів із зазначенням причини та часу блокування.

AI-функціональність системи відображена таблицею User_activities, яка зберігає інформацію про всі AI-операції (тип контенту, кількість токенів, вартість, результат), що дає змогу здійснювати облік витрат і формувати статистику використання.

Загалом, представлена структура бази даних є нормалізованою та модульною, що забезпечує масштабованість, зручність підтримки та ефективність виконання запитів. Така організація даних дозволяє реалізувати всі ключові функції платформи – від управління контентом і соціальної взаємодії до AI-сервісів і платіжної системи.

У базі даних переважають зв’язки типу один-до-багатьох **(1:N)**. Наприклад, між таблицями Users та Songs існує зв’язок 1:N через таблицю Song_authors, оскільки один користувач може бути автором багатьох пісень. Аналогічно, зв’язок 1:N спостерігається між Users та Playlists, де один користувач може створити декілька плейлистів.

Зв’язок між таблицями Playlists та Playlist_items також є 1:N, оскільки один плейлист містить багато записів із піснями. У свою чергу, таблиця Playlist_items формує зв’язок із таблицею Songs, що дозволяє реалізувати включення однієї пісні до різних плейлистів.

Для реалізації зв’язків типу багато-до-багатьох (N:M) використовуються проміжні таблиці. Наприклад:

1. Songs <-> Genres через таблицю Song_genres, оскільки одна пісня може належати до кількох жанрів, і один жанр може містити багато пісень;
2. Users <-> Songs через таблицю Favorite_songs, що дозволяє користувачам додавати багато пісень в обране, а кожна пісня може бути в обраному у багатьох користувачів;
3. Users <-> Users через таблицю Friends, яка реалізує двосторонній зв’язок дружби між користувачами.

Зв’язки типу один-до-одного (1:1) застосовуються рідше, але можуть бути логічно присутні, наприклад, між Users та User_subscription_plan, де кожен користувач має одну активну підписку в певний момент часу.

Таблиці, пов’язані з автентифікацією, також мають зв’язки 1:N:

1. Users <-> User_refresh_tokens – один користувач може мати декілька refresh-токенів (для різних пристроїв);
2. Users <-> User_provider_credentials – один користувач може мати декілька способів входу (наприклад, Google, email/password).

Для логування активності:

1. Users <-> User_activities – зв’язок 1:N, оскільки один користувач може виконати багато AI-операцій;
2. Songs <-> Song_listenings – також 1:N, що дозволяє зберігати історію прослуховувань.

У підсистемі сповіщень:

1. Users <-> Notifications – 1:N (один користувач може отримувати багато сповіщень);

2. Notification_types <-> Notifications – 1:N, що дозволяє класифікувати сповіщення за типами.

Таким чином, використання різних типів зв’язків (1:1, 1:N, N:M) у поєднанні з проміжними таблицями забезпечує гнучкість моделі даних, дозволяє уникнути дублювання інформації та підтримує масштабованість системи.

4. ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛІВ ПРОЄКТУ

4.1. Вибір технологій реалізації та середовища розробки

Вибір технологічного стеку для розробки вебзастосунку RedCloud ґрунтується на результатах аналізу існуючих рішень, вимогах до функціоналу, визначених у ході UX-дослідження, та практичних міркуваннях щодо продуктивності, масштабованості й підтримуваності системи.

Основним інструментом розробки обрано Visual Studio Code – один із найпоширеніших редакторів коду з розширеною підтримкою TypeScript, вбудованим відлагоджувачем, інтеграцією з Git та великою екосистемою розширень.

Для реалізації серверної частини обрано платформу Node.js – асинхронне середовище виконання JavaScript, що забезпечує високу продуктивність при обробці великої кількості одночасних запитів. Це особливо важливо для потокового відтворення аудіо, де сервер має одночасно обслуговувати багато активних з'єднань. Використання TypeScript на серверній стороні забезпечує статичну типізацію, що знижує ймовірність помилок та підвищує читабельність коду при роботі зі складними структурами даних, зокрема при обробці векторних ембедингів системи рекомендацій [4,5].

Інтерфейс застосунку реалізовано за допомогою бібліотеки React з TypeScript. React забезпечує компонентний підхід до побудови інтерфейсу, що дозволяє повторно використовувати UI-елементи – картки треків, плеєр, панель навігації – та ефективно управляти станом застосунку. Компонентна архітектура також спрощує подальше розширення функціоналу без переписування існуючого коду [6].

Для зберігання даних застосунку обрано PostgreSQL – реляційну систему управління базами даних з відкритим кодом. PostgreSQL забезпечує надійне зберігання структурованих даних: інформації про треки, виконавців, альбоми, плейлисти та користувачів [11].

Для реалізації функціоналу платної підписки інтегровано Stripe – провідну платіжну платформу з детально задокументованим API, підтримкою webhook-подій та вбудованими механізмами безпеки. Stripe забезпечує обробку рекурентних платежів, управління статусом підписки та автоматичне оновлення платіжних даних, що звільняє від необхідності самостійно реалізовувати критично важливу платіжну інфраструктуру [16].

Для реалізації ШІ-функцій застосунку використовується OpenAI API, що забезпечує чотири ключові можливості. Модель gpt-4o-transcribe застосовується для транскрипції аудіофайлів у текст – вона автоматично розпізнає мову та генерує текст пісні для треків, що не мають офіційних субтитрів. Модель gpt-4o-mini використовується для генерації текстових запитів на основі метаданих треку перед створенням зображення. Модель gpt-image-1-mini генерує обкладинки для пісень та плейлистів на основі користувацького запиту – вона оптимізована для швидкої генерації зображень із достатньою якістю для музичного сервісу. Модель text-embedding-3-small перетворює метадані треків у вектори, які зберігаються в базі даних та використовуються для обчислення косинусної подібності між треками в системі AI-рекомендацій. Використання єдиного провайдера для різних ШІ-задач спрощує управління API-ключами та контроль витрат [14].

Для реалізації функції лайв-перекладу текстів пісень обрано DeepL API – один із найточніших сервісів машинного перекладу, що використовує нейронні мережі власної розробки та демонструє значно вищу якість порівняно з альтернативами, особливо для художніх та поетичних текстів. DeepL підтримує понад 30 мов, включаючи українську, корейську та японську, що безпосередньо відповідає потребам цільової аудиторії, виявленим під час UX-дослідження [15].

Таким чином, обраний технологічний стек забезпечує реалізацію всього запланованого функціоналу RedCloud: від базового потокового відтворення музики до інноваційних ШІ-функцій. Кожна технологія обрана з

урахуванням конкретних технічних вимог проєкту, а їх сукупність формує цілісну, масштабовану архітектуру вебзастосунку.

4.2. Реалізація серверної частини (back-end)

Серверна частина вебзастосунку RedCloud реалізована на платформі Node.js з використанням TypeScript, що забезпечує статичну типізацію та знижує ймовірність помилок під час розробки складної бізнес-логіки. Для компіляції використовується розширений компілятор tspc з підтримкою path-аліасів, що спрощує імпорти між модулями проєкту. Автоматичний рестарт сервера при змінах забезпечується через nodemon, статичний аналіз коду – через ESLint, форматування – через Prettier.

Як веб-фреймворк обрано Express [10], на базі якого побудовано REST API застосунку. Архітектура побудована на шаруватому принципі, що відповідає усталеним патернам проєктування Node.js-застосунків [19]: кожен HTTP-запит від клієнта проходить через глобальні middleware – перевірку автентифікації та бан-фільтр – після чого потрапляє до відповідного роутера. Роутер передає запит контролеру, який валідує вхідні дані через відповідну Zod-схему та делегує виконання бізнес-логіки сервісу. Сервіс взаємодіє з моделями бази даних та повертає результат контролеру, який формує HTTP-відповідь. У разі виникнення помилки на будь-якому етапі вона передається до централізованого errorMiddleware через виклик next(error). Для обробки крос-доменних запитів підключено CORS-middleware, для роботи з завантаженням медіафайлів – Multer. Окремо реалізовано WebSocket-комунікацію в реальному часі через Socket.IO, що забезпечує синхронізацію стану плеєра між вкладками та підтримку спільного прослуховування.

Для роботи з базою даних PostgreSQL використовується двошарова архітектура: Knex.js [12] як SQL query builder та менеджер міграцій, і Objection.js [13] як ORM поверх Knex із повною типізацією моделей. Архітектура серверної частини побудована на принципі інверсії контролю з використанням контейнера InversifyJS – усі сервіси реєструються у файлі

ioc.builder.ts та впроваджуються через декоратори @injectable() та @inject(), що забезпечує слабку зв'язаність компонентів.

Потік аутентифікації починається з реєстрації користувача – пароль хешується через bcrypt та зберігається у базі даних, після чого на вказану email-адресу надсилається лист підтвердження через Nodemailer. Після підтвердження email користувач може увійти в систему: сервер генерує короткоживучий access-токен та довгоживучий refresh-токен. Refresh-токен хешується та зберігається у таблиці user_refresh_tokens, що забезпечує збереження сесій між перезапусками сервера. При оновленні токенів клієнт надсилає refresh-токен, сервер знаходить відповідний запис у БД, верифікує його через bcrypt.compare та видає нову пару токенів. При вході через Google OAuth клієнт отримує Google credential, який сервер верифікує через jose, витягує профіль користувача та або створює новий обліковий запис, або прив'язує Google-профіль до існуючого. Розмежування доступу забезпечується трьома middleware: authMiddleware для перевірки JWT, requireRole для контролю ролей та requireSubscription для захисту преміум-ендпоінтів ШІ-функцій.

CRUD-операції з основними сутностями – користувачами, піснями, плейлистами та жанрами – реалізовані через відповідні сервіси з урахуванням зв'язків many-to-many між таблицями. При завантаженні медіафайлів Multer зберігає файл у тимчасове сховище, після чого сервіс переміщує його до відповідної папки у storage/ та записує шлях до файлу в базу даних. Роздача файлів клієнту відбувається через express.static() без додаткової серверної обробки.

Для інтеграції зовнішніх ШІ-сервісів підключено OpenAI SDK [14], що забезпечує генерацію обкладинок через gpt-image-1-mini, транскрипцію аудіо через gpt-4o-transcribe, генерацію текстових запитів через gpt-4o-mini, а також отримання векторних ембедингів треків через text-embedding-3-small з розмірністю вектора 1536. Математична модель системи рекомендацій на основі косинусної подібності між векторними ембедингами, а також модель

обліку вартості AI-операцій формалізовані у підрозділі 3.1 та безпосередньо реалізовані у відповідних сервісах серверної частини. DeepL Node SDK [15] використовується для лайв-перекладу текстів пісень. Обробка платежів реалізована через Stripe [16] – маршрут POST /payment-webhook реєструється до підключення express.json(), що дозволяє отримати сире тіло запиту для верифікації підпису. Після успішної верифікації сервер обробляє відповідну подію – активацію, скасування або оновлення підписки – та автоматично оновлює статус у базі даних.

Для виконання ресурсомістких фонових завдань реалізовано систему черг на базі BullMQ [17] з Redis як брокером повідомлень. Надсилання транзакційних листів реалізовано через Nodemailer, структуроване логування – через Winston з рівнями info, warn та error. Централізована обробка помилок реалізована через errorMiddleware, підключений останнім у app.ts: ZodError повертає статус 400 з деталізованим списком порушень, AppError – відповідний HTTP-статус, будь-яка інша помилка – статус 500. Усі помилки логуються через Winston та зберігаються у файл логів.

4.3. Реалізація клієнтської частини (front-end)

Клієнтська частина вебзастосунку RedCloud реалізована як Single Page Application на базі React з TypeScript. Як інструмент збірки та dev-сервер використовується Vite, що забезпечує швидкий час запуску та миттєве оновлення модулів під час розробки. Стилізація реалізована через Tailwind CSS з utility-first підходом, умовне об'єднання класів здійснюється через бібліотеку classnames.

Маршрутизація застосунку побудована на react-router та охоплює 18 сторінок: головна, аутентифікація (вхід, OAuth-callback, відновлення паролю), профіль та редагування профілю, підписки, тексти пісень та їх переклад, редактор пісень, мої пісні, плейлист та список плейлистів, редактор плейлиста, пошук, обрані, налаштування, панель управління, про застосунок та сторінка 404. Захищені сторінки обгорнуті в Layout-роут, що забезпечує спільний шар

із сайдбаром, верхньою панеллю та плеєром. Сторінки аутентифікації винесені поза Layout і не містять елементів навігації.

Управління станом застосунку реалізовано через Redux Toolkit з RTK Query для роботи з серверними даними. Визначено 12 API-слайсів: auth, users, profile, songs, playlist, genres, lyrics, ai, recommendation, friends, search та subscription. Кожен слайс реєструється у store.ts як reducer та middleware. Для стану, що не потребує синхронізації з сервером, використовується React Context: AudioContext зберігає поточний стан плеєра, ThemeContext – обрану тему інтерфейсу.

HTTP-комунікація з сервером реалізована через Axios з кастомним axiosBaseQuery-адаптером для інтеграції з RTK Query. Request-інтерсептор автоматично додає до кожного запиту заголовок Authorization: Bearer <accessToken> та x-browser-id – UUID для ідентифікації конкретного пристрою користувача. Response-інтерсептор обробляє відповіді зі статусом 401 та 403: у такому випадку автоматично виконується запит на оновлення access-токена через окремий mainInstanceRetry інстанс, після чого оригінальний запит повторюється з новим токеном. Якщо оновлення токена завершилось невдачею, користувач перенаправляється на сторінку входу.

Real-time комунікація реалізована через socket.io-client у вигляді сервісу-сінглтону SocketService, що підключається до окремого WebSocket-сервера на порті 8081. Авторизація WebSocket-з'єднання здійснюється через передачу Bearer-токена в параметрі auth.token при ініціалізації. Через Socket.IO реалізовано два ключові сценарії: відображення онлайн-статусу друзів у реальному часі та синхронізація стану плеєра між різними пристроями одного користувача.

Форми реалізовано через Formik з валідацією на основі Zod-схем, що відповідають схемам серверної частини. Такий підхід забезпечує єдині правила валідації на клієнті та сервері. Локалізація інтерфейсу реалізована через i18next з автоматичним визначенням мови браузера через i18next-browser-languagedetector. Підтримується вісім мов: українська, англійська,

німецька, французька, іспанська, італійська, польська та португальська. Перемикач мови доступний у розділі налаштувань. Динамічні метадані сторінок керуються через react-helmet-async, що забезпечує унікальний тег title для кожної з 18 сторінок.

Бібліотека власних UI-компонентів налічує близько 20 багаторазово використовуваних елементів: базові елементи керування (Button, Input, Checkbox, Toggle, FileInput), елементи відображення контенту (Avatar, Song, SongSection, Player, List), навігаційні компоненти (Menu, MobileMenu, ContextMenu, PlaylistsMenu, StateSidebar), компоненти форм авторизації (AuthForm) та компоненти обмеження доступу до преміум-функцій (PremiumOverlay, PremiumFeatureLock). Останні два відповідають за візуальне блокування ШІ-функцій для користувачів без активної підписки. Додатково реалізовано три власні хуки: useSubscription для отримання статусу підписки, useFriends для роботи зі списком друзів та їх статусом онлайн, useContextMenu для керування контекстними меню.

Файлова структура клієнтської частини організована за функціональним принципом: сторінки у папці pages/, компоненти у components/, API-слайси у store/api/, axios-інстанс у api/, WebSocket-сервіс у services/, контексти у context/, хуки у hooks/, локалізація у locales/ з окремим JSON-файлом для кожної мови, TypeScript-типи у types/, Zod-схеми у validation/ та допоміжні утиліти у utils/.

4.4. Налаштування бази даних

Як система управління базами даних обрано PostgreSQL [11] з підключенням через connection string зі змінної середовища PG_DB_CONNECTION_STRING. Налаштовано connection pool з мінімум 2 та максимум 20 з'єднань, що забезпечує ефективне використання ресурсів під навантаженням. Конфігурація підтримує чотири середовища: development, test, staging та production – кожне з окремими параметрами підключення у файлі knexfile.cjs. Схема бази даних визначається через змінну середовища

DB_SCHEMA, що дозволяє розгортати застосунок у кастомній схемі без змін у коді міграцій.

Схема бази даних будувалась ітеративно через 22 міграції з timestamp-префіксами у форматі YYYYMMDDHHMMSS_<table>.cjs. Кожна міграція містить методи up() для застосування змін та down() для їх відкату, що дозволяє безпечно повертатись до попереднього стану схеми. Застосовані міграції відстежуються в окремій таблиці migrations. Запуск міграцій здійснюється через npm run knex [12] як обгортку над knex --knexfile=knexfile.cjs.

Схема розвивалась у кілька логічних етапів відповідно до функціоналу. Спочатку реалізовано базові таблиці аутентифікації та користувачів: users, user_provider_credentials, user_refresh_tokens та user_bans. Далі додано підписки: subscription_plans, subscription_plan_prices та user_subscription_plans. Контентний шар охоплює таблиці genres, songs, song_authors та song_genres. Взаємодії користувачів з контентом зберігаються у song_actions, favorite_songs та song_listenings. Соціальний функціонал реалізовано через таблиці playlists, playlist_items, playlist_members та friends. Сповіщення зберігаються у notification_types та notifications. Окремо виділено таблиці для ШІ-функціоналу: user_activities для логування використання та підрахунку вартості токенів, та song_embeddings для зберігання векторних представлень треків.

Зв'язки між таблицями організовані за стандартними реляційними принципами. One-to-many зв'язки реалізовані між користувачем та піснями, плейлистом та його елементами, користувачем та підписками. Many-to-many зв'язки реалізовані через проміжні таблиці: song_authors пов'язує пісні з авторами з урахуванням ролей, song_genres – пісні з жанрами, playlist_items – плейлисти з піснями, playlist_members – плейлисти зі співавторами. Таблиця friends реалізує зв'язок між користувачами системи для реалізації функціоналу підписки на друзів. Усі foreign keys налаштовано з onDelete: CASCADE для каскадного видалення залежних записів – наприклад, видалення пісні

автоматично видаляє її ембединг та всі пов'язані дії. На foreign keys визначено індекси для прискорення JOIN-операцій. Timestamps created_at та updated_at додаються через table.timestamps(true, true) з автоматичним оновленням при зміні запису.

Особливої уваги потребує зберігання векторних ембедингів. Для кожного треку система text-embedding-3-small генерує вектор розмірністю 1536 чисел з плаваючою точкою, який зберігається у колонці типу JSONB таблиці song_embeddings. Обчислення косинусної подібності між векторами при пошуку рекомендацій виконується на рівні Node.js, що забезпечує гнучкість реалізації.

Для швидкого розгортання робочого середовища реалізовано 10 seed-файлів з числовими префіксами для визначення порядку виконання. Сіди наповнюють базу початковими даними: тестові користувачі з OAuth credentials, жанри (Rock, Pop, Hip-Hop тощо), тарифні плани Free, Premium та Family з відповідними цінами, тестові пісні з авторами та жанрами, а також типи сповіщень.

4.5. Тестування серверної частини

Тестування серверної частини вебзастосунку RedCloud проводилось через інструмент Postman [18], що охоплює всі основні модулі системи: auth, songs, lyrics, ai, recommendations, playlists, friends, search та payment. Для кожного захищеного ендпоінту використовувався Bearer Token, отриманий після успішної аутентифікації.

Тестування аутентифікації підтвердило коректну роботу обох сценаріїв. При передачі правильних облікових даних сервер повертає статус 200 з парою access та refresh токенів (рисунок 4.1). При введенні невірного паролю повертається статус 403 з повідомленням «Wrong password» (рисунок 4.2).

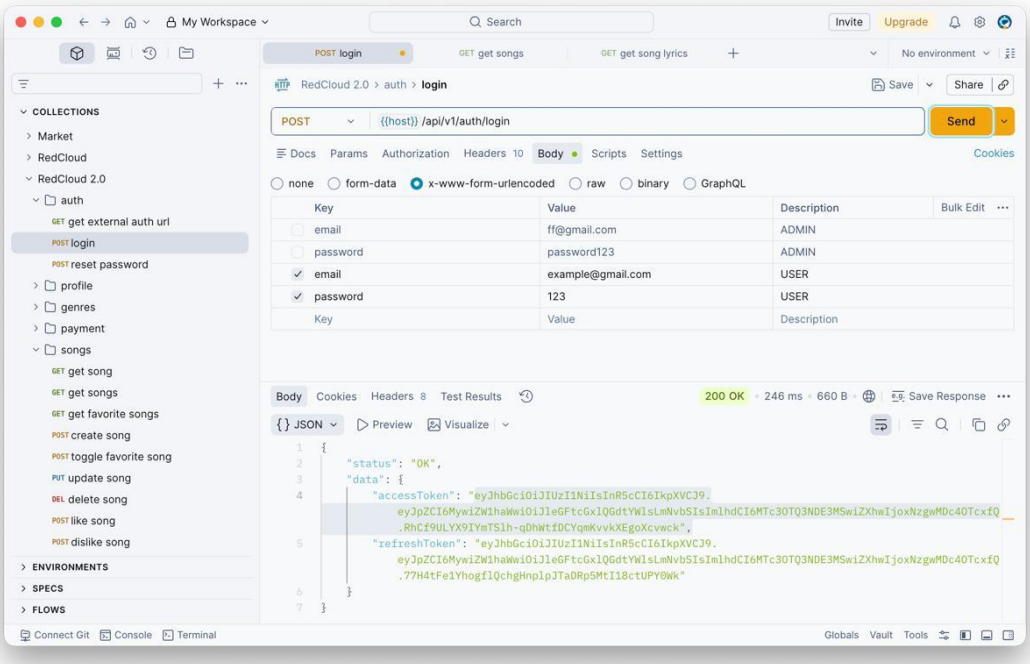

Рисунок 4.1 – Тестування аутентифікації: успішний вхід (200 OK)

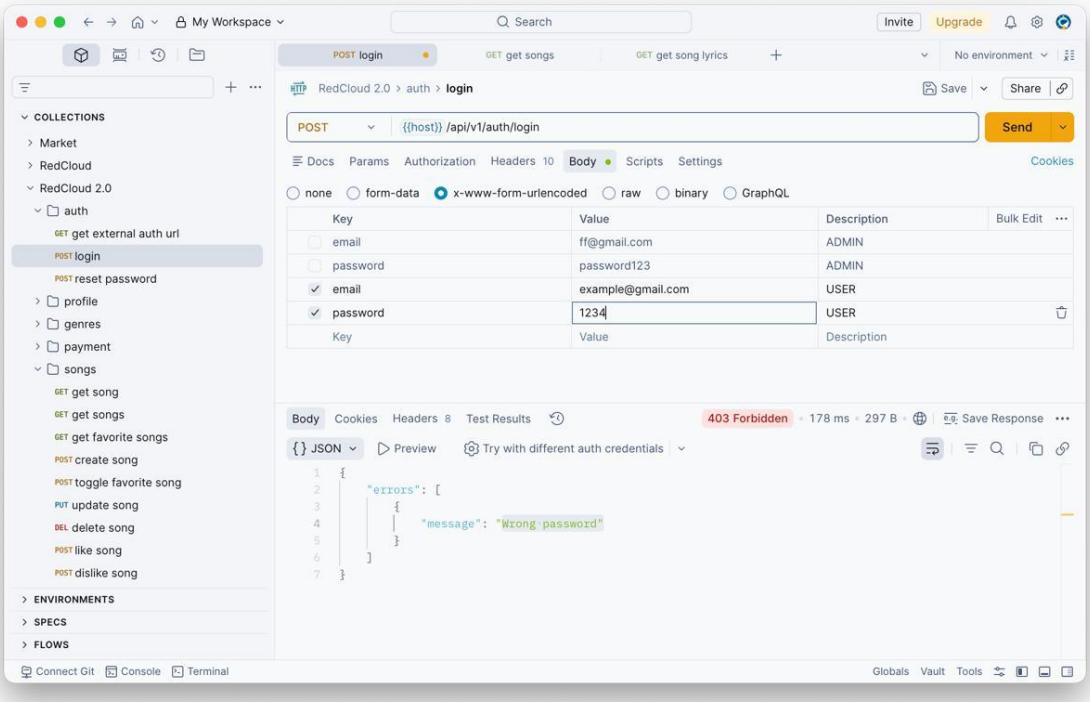

Рисунок 4.2 – Тестування аутентифікації: невірний пароль (403 Forbidden)

Тестування отримання контенту показало коректну роботу основних ендпоінтів. Запит GET /api/v1/songs повертає масив треків зі статусом 200 (рисунок 4.3). Запит GET /api/v1/lyrics/:songId повертає текст пісні з

відповідним songId та title (рисунок 4.4). Запит GET /api/v1/recommendations повертає персоналізований список треків з даними про тривалість, мову та URL аудіофайлу (рисунок 4.5).

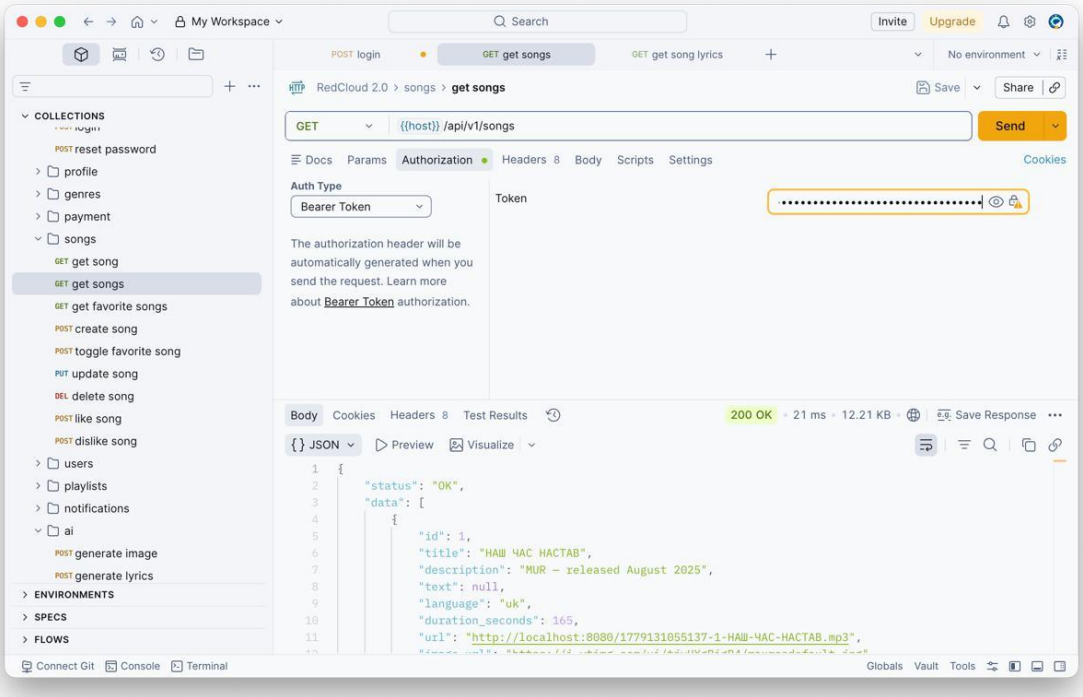

Рисунок 4.3 – Тестування отримання списку пісень (200 OK)

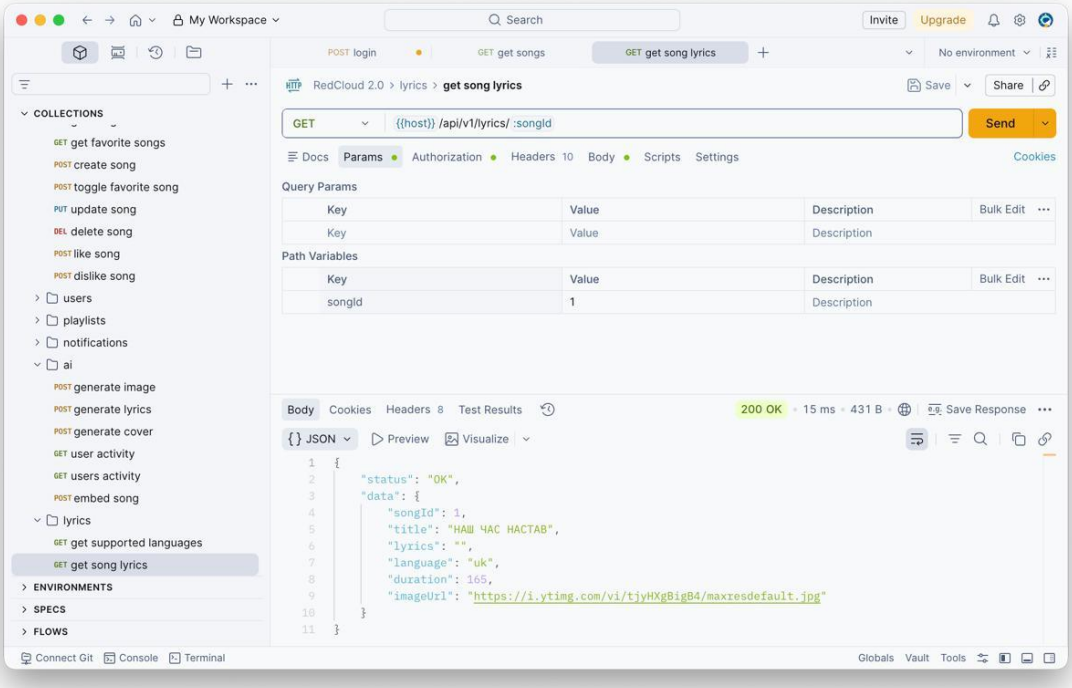

Рисунок 4.4 – Тестування отримання тексту пісні (200 OK)

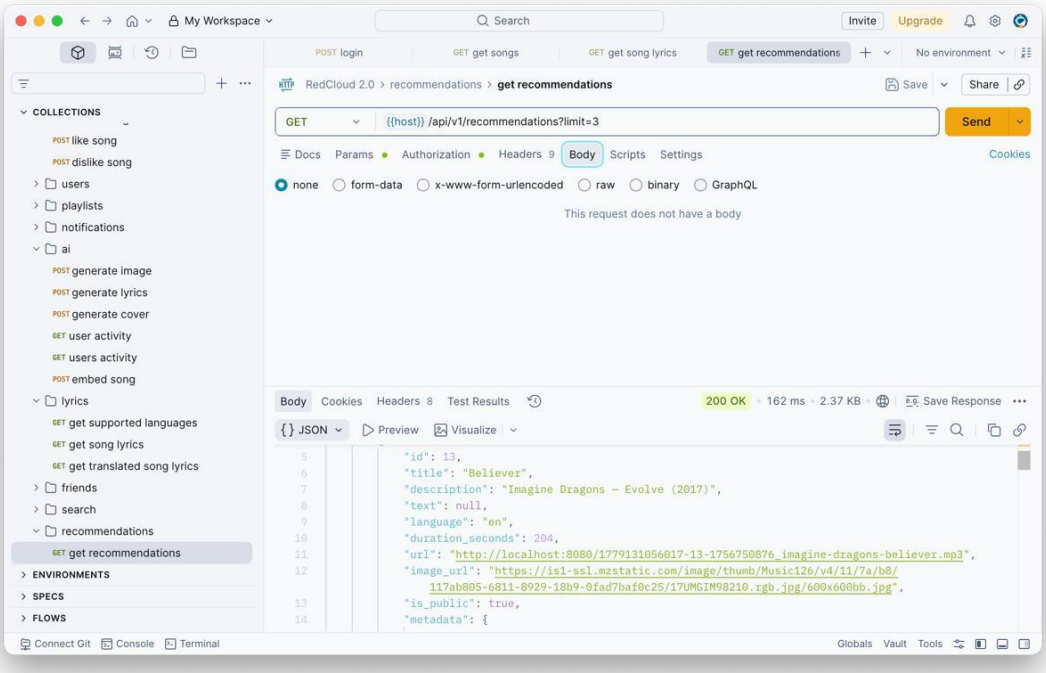

Рисунок 4.5 – Тестування системи рекомендацій (200 OK)

Тестування ШІ-функцій охопило три сценарії. Генерація тексту пісні через POST /api/v1/ai/generate-lyrics з параметром songId повертає повний текст треку зі статусом 200 (рисунок 4.6). Генерація зображення через POST /api/v1/ai/generate-image з активною підпискою повертає статус 200 з URL згенерованого зображення та даними провайдера (рисунок 4.7). Генерація обкладинки плейлиста через POST /api/v1/ai/playlists/:id/generate-cover з текстовим запитом повертає статус 200 з URL обкладинки (рисунок 4.8).

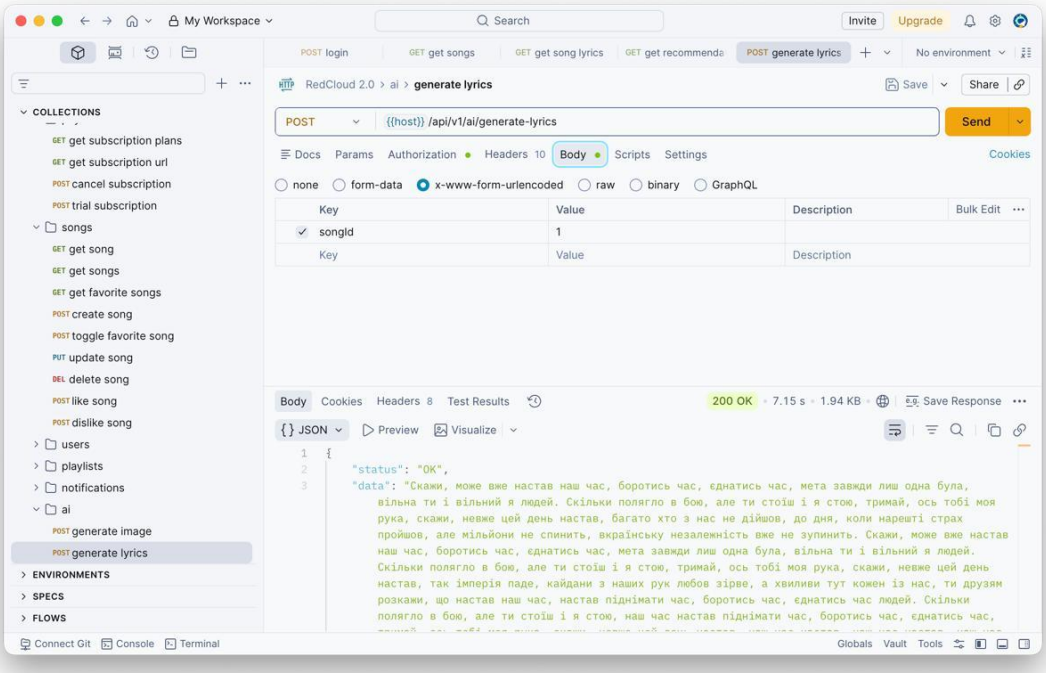

Рисунок 4.6 – Тестування генерації тексту пісні через ШІ (200 OK)

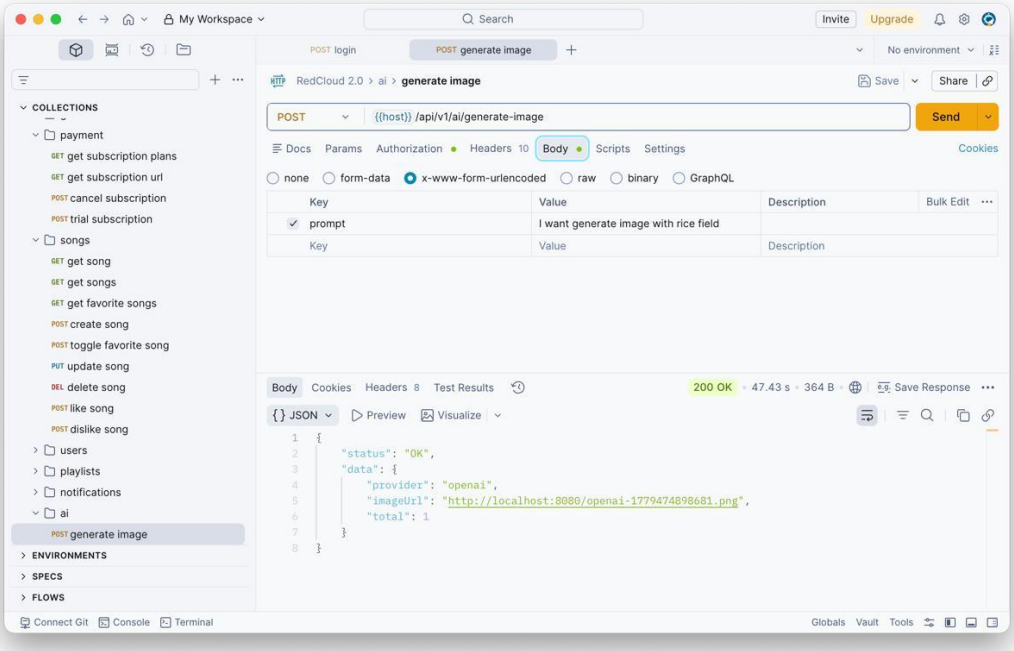

Рисунок 4.7 – Тестування генерації зображення (200 OK, 52с)

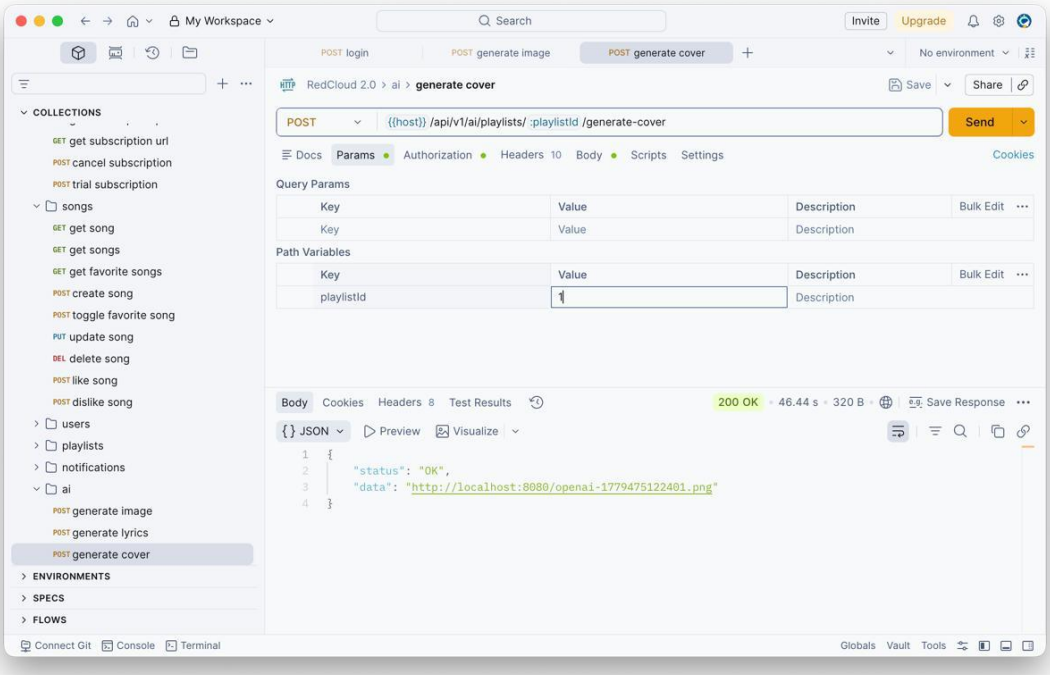

Рисунок 4.8 – Тестування генерації обкладинки із запитом (200 OK)

Тестування системи розмежування доступу підтвердило коректну роботу захисних middleware. Спроба викликати ендпоінт генерації обкладинки з простроченим токеном повертає статус 401 з повідомленням «Token has expired. Please log in again» (рисунок 4.9). Спроба викликати ШІ-функцію без активної підписки повертає статус 403 з повідомленням «This feature requires an active subscription» (рисунок 4.10).

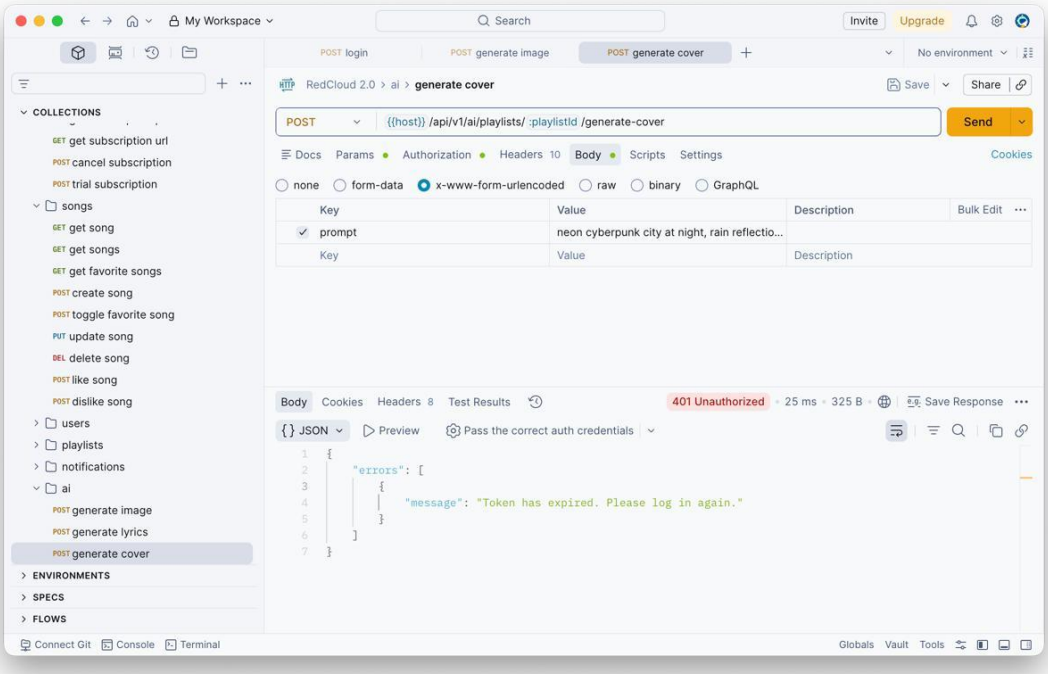

Рисунок 4.9 – Тестування генерації обкладинки без токена (401 Unauthorized)

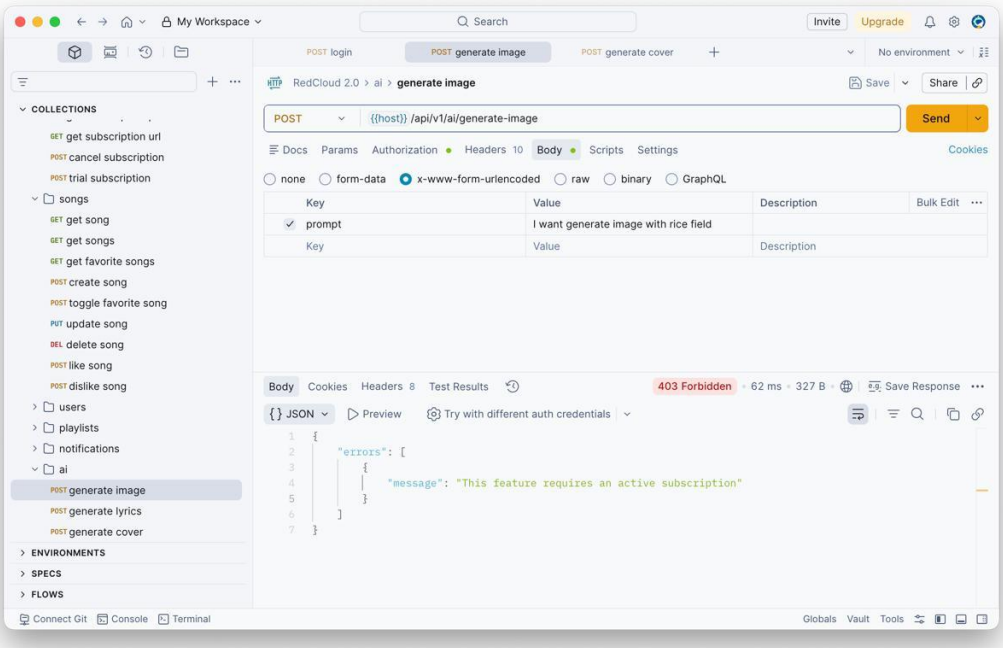

Рисунок 4.10 – Тестування доступу до ШІ без підписки (403 Forbidden)

За результатами тестування серверної частини всі перевірені ендпоінти повертають очікувані відповіді. Система коректно обробляє як успішні сценарії, так і граничні випадки – некоректні облікові дані, відсутній або

прострочений токен, спроба доступу до преміум-функцій без активної підписки.

4.6. Тестування користувацького інтерфейсу

Тестування користувацького інтерфейсу проводилось шляхом ручної перевірки основних сценаріїв взаємодії користувача із застосунком у браузері.

Сторінка авторизації відображає форму входу з полями email та пароль, чекбоксом «Remember me», посиланням на відновлення паролю та кнопкою входу через Google OAuth. Перемикання між вкладками «Authorization» та «Registration» працює коректно (рисунок 4.11).

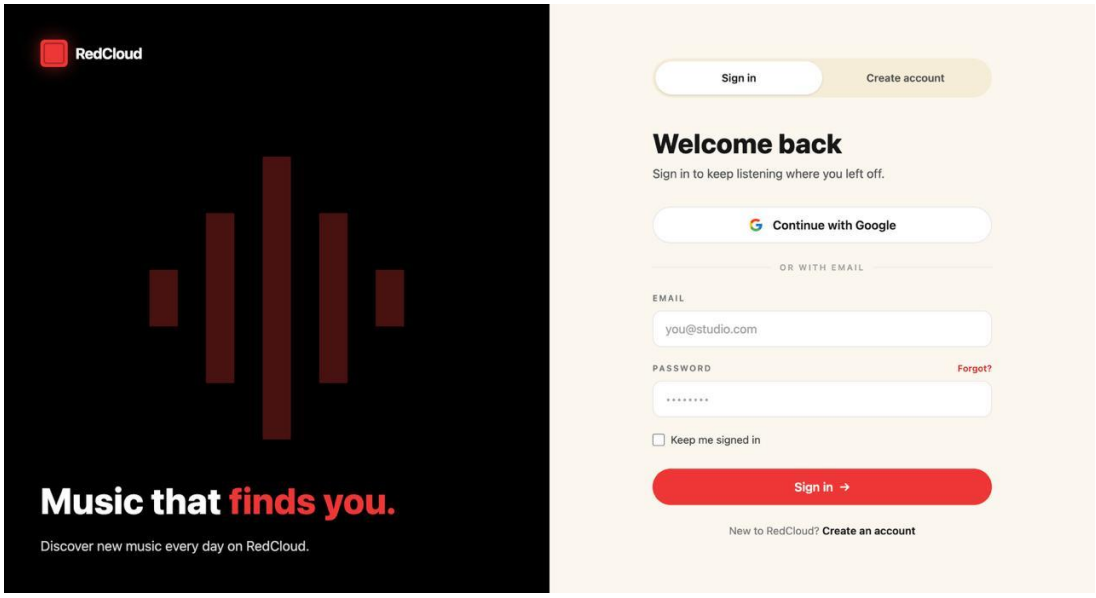

Рисунок 4.11 – Сторінка авторизації вебзастосунку RedCloud

Головна сторінка після успішного входу відображає банер, блок персоналізованих рекомендацій «Recommended for you» з горизонтальним прокручуванням та блок нових релізів «New releases». У правому куті блоку рекомендацій присутній перемикач між режимами «For You» та «Discover», що відповідає реалізованій системі AI-рекомендацій (рисунок 4.12).

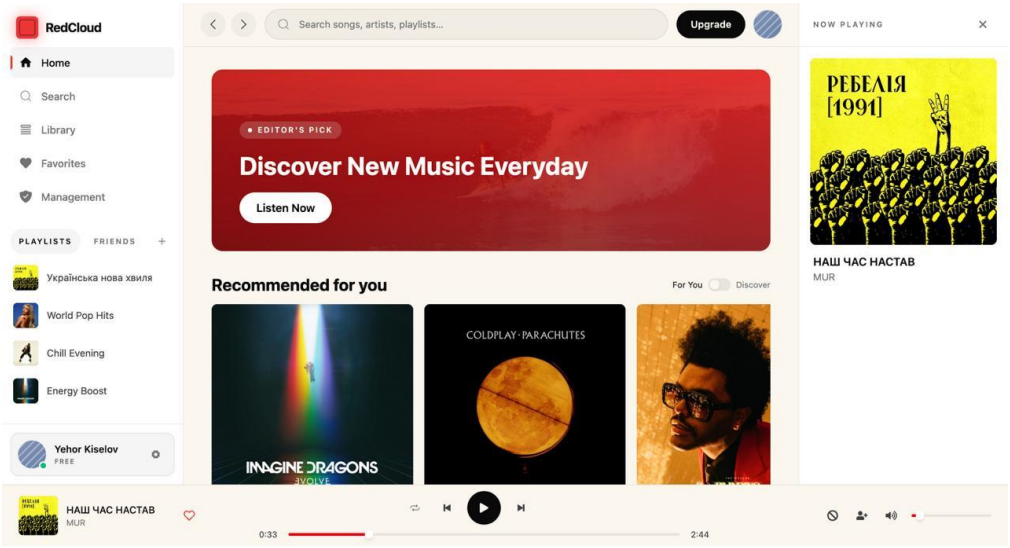

Рисунок 4.12 – Головна сторінка з блоком рекомендацій та перемикачем режимів

Сторінка бібліотеки відображає список плейлистів користувача з обкладинками та назвами, поле пошуку по плейлистах та кнопки створення і налаштування. Горизонтальна стрічка обкладинок у верхній частині забезпечує швидкий перехід між плейлистами (рисунок 4.13).

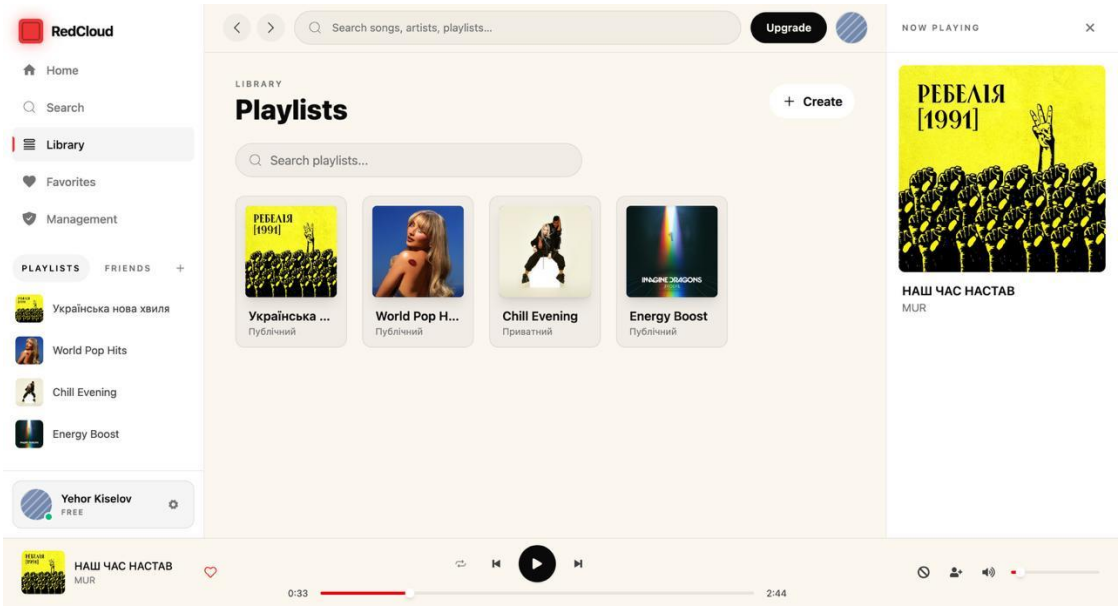

Рисунок 4.13 – Сторінка бібліотеки зі списком плейлистів користувача

Сторінка плейлиста відображає назву, кількість пісень та загальну тривалість, кнопку відтворення та список треків з обкладинками і тривалістю кожного (рисунок 4.14).

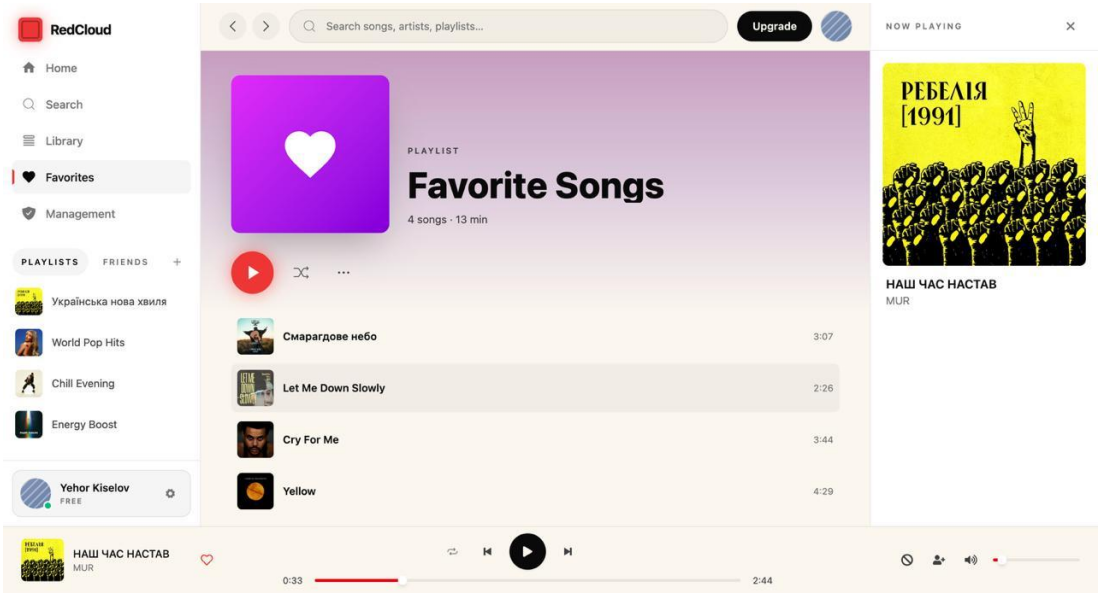

Рисунок 4.14 – Сторінка плейлиста «Улюблені пісні» з переліком треків

Сторінка текстів пісень відображає повний текст треку під час відтворення з кнопкою «Перекласти» у правому верхньому куті (рисунок 4.15). Після натискання кнопки перекладу інтерфейс розділяється на дві колонки – оригінальний текст ліворуч та переклад обраною мовою праворуч. Перемикач мови перекладу розміщено у верхньому правому куті. Перевірено переклад на українську мову через DeepL – результат відображається коректно (рисунок 4.16).

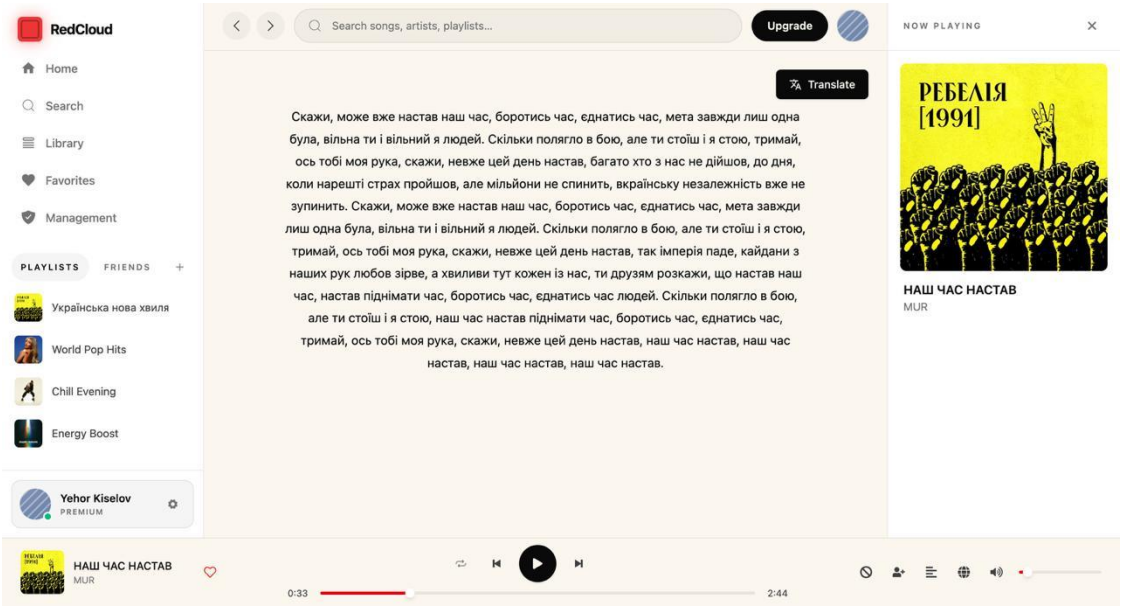

Рисунок 4.15 – Відображення тексту пісні під час відтворення

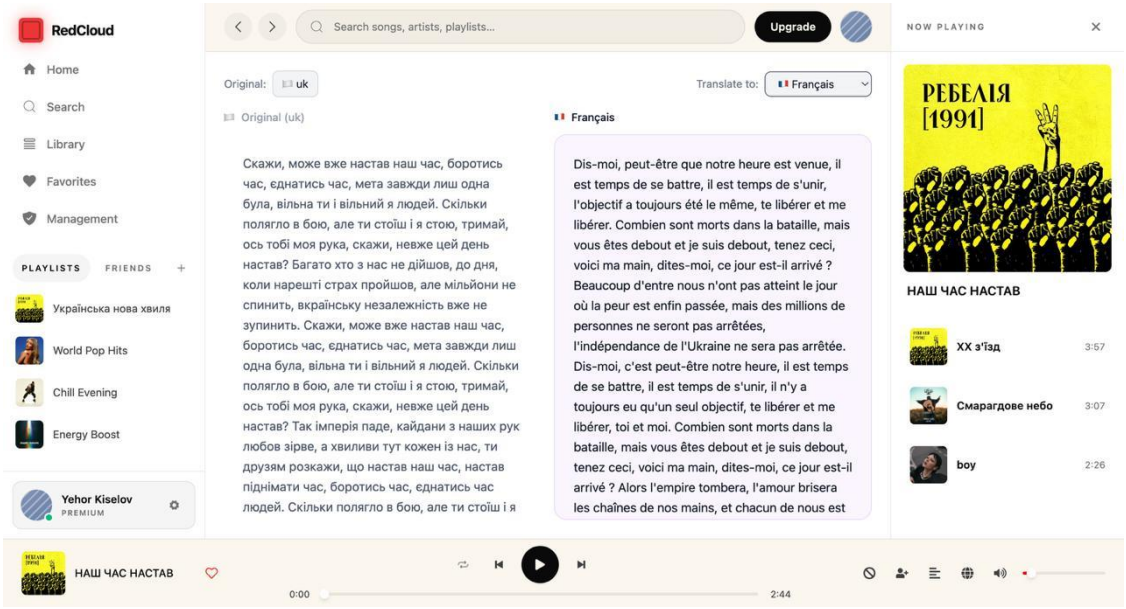

Рисунок 4.16 – Лайв-переклад тексту пісні на українську мову через DeepL

Перевірено блокування преміум-функцій для користувача без активної підписки. При спробі відкрити тексти пісень без преміум-плану відображається компонент PremiumFeatureLock з повідомленням «Lyrics are available only for Premium and Family subscribers» та кнопками «Upgrade to Premium» і «Go Back». Фіксований плеєр внизу продовжує відтворення під час відображення блокуючого екрану (рисунок 4.17).

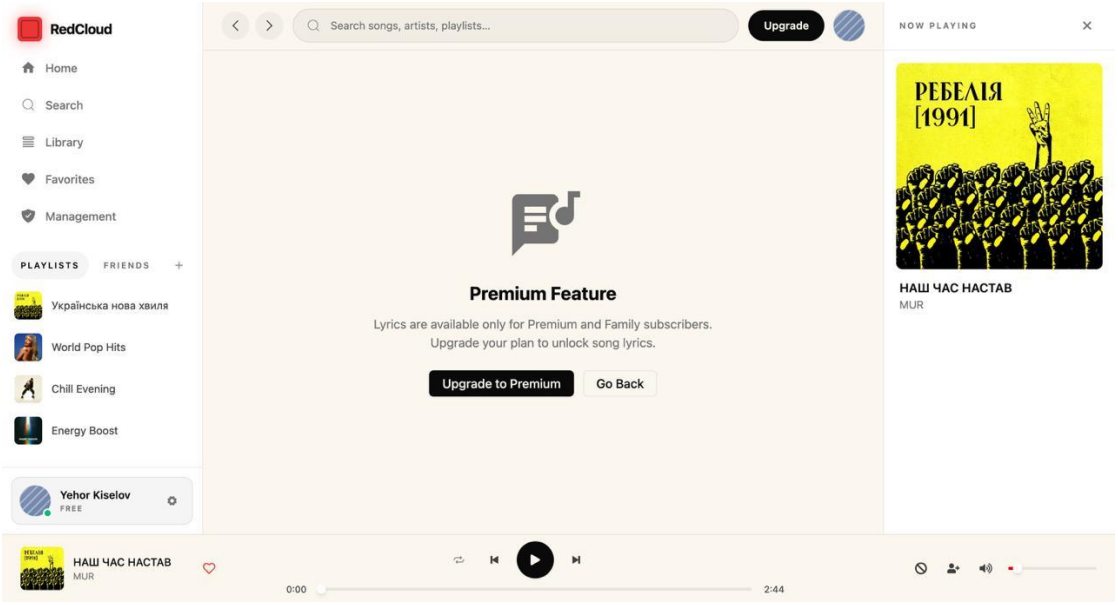

Рисунок 4.17 – Блокування преміум-функції для користувача без підписки

За результатами тестування інтерфейс застосунку працює коректно в усіх перевірених сценаріях. Навігація між сторінками, відтворення аудіо,

відображення текстів та їх переклад, а також система розмежування доступу функціонують відповідно до визначених вимог.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи розроблено вебзастосунок RedCloud для потокового відтворення музики, спроєктований на основі дослідження досвіду та побажань цільової аудиторії з використанням сучасних технологій веброзробки та штучного інтелекту. Поставлена мета досягнута, всі визначені завдання виконані.

Аналіз провідних музичних стрімінгових сервісів – Spotify, Apple Music та YouTube Music – дозволив визначити галузеві стандарти функціоналу та інтерфейсних рішень, виявити незайняті ніші та сформувати конкурентні переваги RedCloud. Встановлено, що жоден із досліджених сервісів не пропонує лайв-перекладу текстів пісень, ШІ-генерації обкладинок для пісень та плейлистів, транскрипції текстів і системи рекомендацій на основі AI-ембедингів одночасно, що визначає унікальне позиціонування розробленого застосунку.

Проведене UX-дослідження охопило 81 респондента та 5 глибинних інтерв'ю і підтвердило доцільність усіх запланованих інноваційних функцій: лайв-переклад підтримали понад 80% респондентів, ідею ШІ-генерації обкладинок позитивно оцінили 55,7%, а виявлений запит на контроль над алгоритмами рекомендацій (92,9% підтримали перемикач режимів) обґрунтував реалізований перемикач між персоналізованим режимом «For You» та експериментальним «Discover» [8]. Результати дослідження стали безпосередньою основою для проєктних рішень щодо функціоналу та інтерфейсу застосунку.

Реалізований застосунок охоплює повний стек сучасних технологій. Серверна частина на Node.js з TypeScript забезпечує обробку запитів, JWT-аутентифікацію з підтримкою Google OAuth, систему підписок через Stripe, real-time комунікацію через Socket.IO та фонові черги через BullMQ. Інтеграція OpenAI API забезпечує генерацію обкладинок через gpt-image-1-mini, транскрипцію аудіо через gpt-4o-transcribe та векторні ембединги через text-embedding-3-small, DeepL API – лайв-переклад текстів пісень. Клієнтська

частина на React з Redux Toolkit підтримує 18 сторінок, 8 мов інтерфейсу та близько 20 власних UI-компонентів. База даних PostgreSQL містить 22 таблиці з чітко визначеними зв'язками та підтримкою зберігання векторних ембедингів у форматі JSONB. Реалізовано офлайн-прослуховування музики для користувачів із активною підпискою.

Тестування підтвердило коректну роботу всіх реалізованих функцій: API повертає очікувані відповіді для успішних сценаріїв та граничних випадків, система розмежування доступу між безкоштовними та преміум-функціями працює належним чином, інтерфейс забезпечує зручну взаємодію з усіма можливостями застосунку.

Практичне значення роботи полягає у створенні конкурентоспроможного музичного сервісу, розробленого на основі реальних потреб користувачів, що може бути основою для подальшого комерційного розвитку продукту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Spotify: Music and Podcasts. Spotify AB. URL: https://www.spotify.com (дата звернення: 04.05.2026).
2. Apple Music. Apple Inc. URL: https://music.apple.com (дата звернення: 04.05.2026).
3. YouTube Music. Google LLC. URL: https://music.youtube.com (дата звернення: 04.05.2026).
4. Node.js Documentation. Node.js Foundation. URL: https://nodejs.org/en/docs (дата звернення: 04.05.2026).
5. TypeScript Documentation. Microsoft Corporation. URL: https://www.typescriptlang.org/docs (дата звернення: 04.05.2026).
6. React Documentation. Meta Platforms, Inc. URL: https://react.dev (дата звернення: 04.05.2026).
7. Анкета дослідження користувацького досвіду музичних стрімінгових сервісів. Google Forms, 2026. URL: https://docs.google.com/forms/d/e/1FAIpQLSeDry-yjzk3pUh7988oG9eqkTOCoSPvtBTpPH5sZUeamR2KHQ/viewform (дата звернення: 04.05.2026).
8. Результати анкетування користувачів музичних стрімінгових сервісів. Google Sheets, 2026. URL: https://docs.google.com/spreadsheets/d/1MC-Y71T7_5_hFwmmSPdrY0Y2NbdfAiz4TbtgaNrMd6k/edit?usp=sharing (дата звернення: 04.05.2026).
9. Результати глибинних інтерв'ю з користувачами музичних стрімінгових сервісів. Google Sheets, 2026. URL: https://docs.google.com/spreadsheets/d/1SiYiiiZ9vz_cCxj7m7mouaBcesdZgCuy/edit?usp=sharing (дата звернення: 04.05.2026).
10. Express.js Documentation. OpenJS Foundation. URL: https://expressjs.com (дата звернення: 04.05.2026).
11. PostgreSQL 16 Documentation. The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/16 (дата звернення: 04.05.2026).

12. Knex.js Query Builder Documentation. URL: https://knexjs.org/guide (дата звернення: 04.05.2026).

13. Objection.js ORM Documentation. URL: https://vincit.github.io/objection.js (дата звернення: 04.05.2026).

14. OpenAI API Reference. OpenAI, L.L.C. URL: https://platform.openai.com/docs/api-reference (дата звернення: 04.05.2026).

15. DeepL API Documentation. DeepL SE. URL: https://developers.deepl.com/docs (дата звернення: 04.05.2026).

16. Stripe API Reference. Stripe, Inc. URL: https://stripe.com/docs/api (дата звернення: 04.05.2026).

17. BullMQ Documentation. URL: https://docs.bullmq.io (дата звернення: 04.05.2026).

18. Postman API Platform. Postman, Inc. URL: https://www.postman.com (дата звернення: 04.05.2026).

19. Casciaro M., Mammino L. Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques. 3rd ed. Birmingham : Packt Publishing, 2020. 660 p.

20. Gothelf J., Seiden J. Lean UX: Designing Great Products with Agile Teams. 3rd ed. Sebastopol : O'Reilly Media, 2021. 248 p.

Додаток А

Функція створення Express-сервера

```
async function createServer(ioc: Container) {
  const app = express();

  app.use(
    cors({
      origin: ["http://localhost:5173"],
    }),
  );

  const paymentWebhookCtrl = ioc.get(PaymentController);
  app.post(
    "/payment-webhook",
    express.raw({ type: "application/json" }),
    paymentWebhookCtrl.webhookHandler,
  );

  app.use(express.urlencoded({ extended: true }));
  app.use(express.json({ limit: "20mb" }));
  app.use(express.static(storageFolder));

  app.use("/api/v1", createAPIV1Routes(ioc));

  // error handler
  app.use(errorMiddleware);

  return app;
}
```

Додаток Б

Точка входу застосунку (boot)

```
async function boot() {
  const appType: ApplicationType = (process.env.APP_TYPE ||
    ApplicationType.API) as ApplicationType;
  let _server: Application | undefined;
  let serverName: string;

  try {
    const ioc = await constructIOC();
    switch (appType) {
      case ApplicationType.Socket:
        await initSockets(ioc);
        serverName = ApplicationType.Socket;

        break;
      case ApplicationType.Worker:
        await initWorker(ioc);
        serverName = ApplicationType.Worker;
```

```
        break;
      case ApplicationType.API:
      default:
        _server = await createServer(ioc);

        serverName = ApplicationType.API;
        break;
    }

    const port = parseInt(process.env.PORT || "8080", 10);

    if (_server) {
      _server.listen(port, () => {
        logger().info(`APP (${serverName}) is running on port
${port}`);
      });
    } else {
      logger().info(`APP (${serverName}) initialized
successfully.`);
    }
  } catch (error) {
    logger().error(`Failed to boot the application: ${error}`);
    process.exit(1);
  }
}
```

Додаток В

Клас AuthController

```
@injectable()
export default class AuthController {
  constructor(@inject(AuthService) private authService:
AuthService) {
    this.getAuthUrl = this.getAuthUrl.bind(this);
    this.exchangeAuthCode = this.exchangeAuthCode.bind(this);
    this.signUp = this.signUp.bind(this);
    this.login = this.login.bind(this);
    this.refreshTokens = this.refreshTokens.bind(this);
    this.logout = this.logout.bind(this);
    this.refreshExternalTokens =
this.refreshExternalTokens.bind(this);
    this.resetPassword = this.resetPassword.bind(this);
    this.confirmResetPassword =
this.confirmResetPassword.bind(this);
  }

  public async getAuthUrl(req: Request, res: Response) {
    const { [browserHeader]: browser } =
getAuthUrlValidation.parse(
      req.headers,
    );

    const url = await this.authService.getAuthUrl(browser);
```

```
    res.json({
      status: "OK",
      data: url,
    });
  }

  public async exchangeAuthCode(req: Request, res: Response) {
    const { code, state } =
exchangeCodeValidation.parse(req.query);

    const { accessToken, refreshToken } =
      await this.authService.exchangeAuthCode(code, state);

    res.redirect(

`${process.env.UI_HOST_URL}${process.env.UI_EXTERNAL_PROVIDER_CA
LLBACK_PATH}?accessToken=${accessToken}&refreshToken=${refreshTo
ken}`,
    );
  }

  public async signUp(req: Request, res: Response) {
    const { [browserHeader]: browser, ...parse } =
signUpValidation.parse({
      ...req.headers,
      ...req.body,
    });

    const creds = await this.authService.signUp({ ...parse,
browser });

    res.json({
      status: "OK",
      data: creds,
    });
  }

  public async login(req: Request, res: Response) {
    const { [browserHeader]: browser, ...parse } =
loginValidation.parse({
      ...req.headers,
      ...req.body,
    });

    const creds = await this.authService.login({ ...parse,
browser });

    res.json({
      status: "OK",
      data: creds,
    });
```

```
  }

  public refreshTokens = async (req: Request, res: Response):
Promise<void> => {
    const {
      authorization: authHeader,
      userId,
      [browserHeader]: browser,
    } = refreshTokensValidation.parse({ ...req.headers, userId:
req.user?.id });

    const token = authHeader!.replace("Bearer ", "");

    const tokens = await this.authService.refreshTokens({
      token,
      userId,
      browser,
    });
    res.json({
      status: "OK",
      data: tokens,
    });
  };

  public async refreshExternalTokens(
    req: Request,
    res: Response,
  ): Promise<void> {
    const { userId } = userIdValidation.parse({ userId:
req.user?.id });
    await this.authService.refreshExternalTokens(userId);

    res.json({
      status: "OK",
    });
  }

  public async logout(req: Request, res: Response):
Promise<void> {
    const { userId, [browserHeader]: browser } =
logoutValidation.parse({
      ...req.headers,
      userId: req.user?.id,
    });
    await this.authService.logout({ userId, browser });

    res.json({
      status: "OK",
    });
  }
```

```
  public async resetPassword(req: Request, res: Response):
Promise<void> {
    const { email } = resetPasswordValidation.parse(req.body);

    await this.authService.resetPassword({ email });

    res.json({
      status: "OK",
    });
  }

  public async confirmResetPassword(
    req: Request,
    res: Response,
  ): Promise<void> {
    const { token, password } =
confirmResetPasswordValidation.parse(req.body);

    await this.authService.confirmResetPassword({
      token,
      newPassword: password,
    });

    res.json({
      status: "OK",
    });
  }
}
```

Додаток Г

Рекомендаційний сервіс (embeddings)

```
@injectable()
export class RecommendationService {
  private cosineSimilarity(a: number[], b: number[]): number {
    let dot = 0,
      magA = 0,
      magB = 0;
    for (let i = 0; i < a.length; i++) {
      dot += a[i] * b[i];
      magA += a[i] * a[i];
      magB += b[i] * b[i];
    }
    const denom = Math.sqrt(magA) * Math.sqrt(magB);
    return denom === 0 ? 0 : dot / denom;
  }

  private async getUserInteractedSongIds(userId: number):
Promise<{
    listenedIds: Set<number>;
    favoriteIds: Set<number>;
    dislikedIds: Set<number>;
  }> {
```

```
    const [listenings, favorites, actions] = await Promise.all([
      SongListeningsModel.query()
        .where("user_id", userId)
        .distinct("song_id")
        .select("song_id"),
      FavoriteSongsModel.query().where("user_id",
userId).select("song_id"),
      SongActionsModel.query()
        .where("user_id", userId)
        .select("song_id", "action"),
    ]);

    return {
      listenedIds: new Set(listenings.map((l) => l.song_id)),
      favoriteIds: new Set(favorites.map((f) => f.song_id)),
      dislikedIds: new Set(
        actions
          .filter((a) => a.action === SongAction.DISLIKE)
          .map((a) => a.song_id),
      ),
    };
  }

  private async getUserGenreScores(
    userId: number,
  ): Promise<Map<number, number>> {
    const scores = new Map<number, number>();

    type GenreCountRow = { genre_id: number; cnt: string };

    const [listenGenres, favoriteGenres] = await Promise.all([
      SongGenresModel.query()
        .join(
          SongListeningsModel.tableName,
          `${SongListeningsModel.tableName}.song_id`,
          `${SongGenresModel.tableName}.song_id`,
        )
        .where(`${SongListeningsModel.tableName}.user_id`,
userId)
        .select(`${SongGenresModel.tableName}.genre_id`)
        .count(`${SongGenresModel.tableName}.genre_id as cnt`)
        .groupBy(
          `${SongGenresModel.tableName}.genre_id`,
        ) as unknown as GenreCountRow[],

      SongGenresModel.query()
        .join(
          FavoriteSongsModel.tableName,
          `${FavoriteSongsModel.tableName}.song_id`,
          `${SongGenresModel.tableName}.song_id`,
        )
```

```
        .where(`${FavoriteSongsModel.tableName}.user_id`,
userId)
        .select(`${SongGenresModel.tableName}.genre_id`)
        .count(`${SongGenresModel.tableName}.genre_id as cnt`)
        .groupBy(
          `${SongGenresModel.tableName}.genre_id`,
        ) as unknown as GenreCountRow[],
    ]);

    for (const row of listenGenres) {
      const id = Number(row.genre_id);
      scores.set(id, (scores.get(id) ?? 0) + Number(row.cnt) *
1);
    }
    for (const row of favoriteGenres) {
      const id = Number(row.genre_id);
      scores.set(id, (scores.get(id) ?? 0) + Number(row.cnt) *
3);
    }

    return scores;
  }

  private async getFriendSongScores(
    userId: number,
  ): Promise<Map<number, number>> {
    const scores = new Map<number, number>();

    const friends = await FriendModel.query()
      .where((builder) => {
        builder
          .where({ user_id: userId, status:
FriendStatus.accepted })
          .orWhere({ friend_id: userId, status:
FriendStatus.accepted });
      })
      .select("user_id", "friend_id");

    const friendIds = friends.map((f) =>
      f.user_id === userId ? f.friend_id : f.user_id,
    );

    if (!friendIds.length) return scores;

    const [friendListenings, friendFavorites] = await
Promise.all([
      SongListeningsModel.query()
        .whereIn("user_id", friendIds)
        .distinct("song_id")
        .select("song_id"),
      FavoriteSongsModel.query()
        .whereIn("user_id", friendIds)
```

```
        .select("song_id"),
    ]);

    for (const l of friendListenings) {
      scores.set(l.song_id, (scores.get(l.song_id) ?? 0) + 2);
    }
    for (const f of friendFavorites) {
      scores.set(f.song_id, (scores.get(f.song_id) ?? 0) + 4);
    }

    return scores;
  }

  async generateSongEmbedding(songId: number): Promise<number[]>
{
    const song = await SongModel.query()
      .findById(songId)
      .withGraphFetched("genres");

    if (!song) throw new AppError(404, `Song ${songId} not
found`);

    const genreTitles = song.genres?.map((g) => g.title).join(",
") ?? "";
    const text = [
      song.title,
      genreTitles,
      song.description ?? "",
      song.text?.slice(0, 500) ?? "",
    ]
      .filter(Boolean)
      .join(". ");

    const response = await openAIClient().embeddings.create({
      model: AIModel.TEXT_EMBEDDING_3_SMALL,
      input: text,
    });

    const embedding = response.data[0].embedding;

    await SongEmbeddingModel.query()
      .insert({
        song_id: songId,
        embedding: JSON.stringify(embedding) as unknown as
number[],
        model: AIModel.TEXT_EMBEDDING_3_SMALL,
      })
      .onConflict("song_id")
      .merge();

    logger().info(`Embedding generated for song ${songId}`);
```

```
    return embedding;
  }

  private async getOrCreateEmbedding(songId: number):
Promise<number[] | null> {
    const existing = await SongEmbeddingModel.query()
      .where("song_id", songId)
      .first();

    if (existing) return existing.embedding;

    try {
      return await this.generateSongEmbedding(songId);
    } catch {
      return null;
    }
  }

  private async getContentScores(
    userTopSongIds: number[],
    candidateIds: number[],
  ): Promise<Map<number, number>> {
    const scores = new Map<number, number>();
    if (!userTopSongIds.length || !candidateIds.length) return
scores;

    const userEmbeddings = (
      await Promise.all(
        userTopSongIds.slice(0, 10).map((id) =>
this.getOrCreateEmbedding(id)),
      )
    ).filter((e): e is number[] => e !== null);

    if (!userEmbeddings.length) return scores;

    const dim = userEmbeddings[0].length;
    const tasteVector = new Array(dim).fill(0) as number[];
    for (const emb of userEmbeddings) {
      for (let i = 0; i < dim; i++) tasteVector[i] += emb[i];
    }
    for (let i = 0; i < dim; i++) tasteVector[i] /=
userEmbeddings.length;

    const candidateEmbeddings = await
SongEmbeddingModel.query().whereIn(
      "song_id",
      candidateIds,
    );

    const embeddedIds = new Set(candidateEmbeddings.map((e) =>
e.song_id));
```

```
    const missingIds = candidateIds.filter((id) =>
!embeddedIds.has(id));
    if (missingIds.length) {
      Promise.all(
        missingIds.slice(0, 20).map((id) =>
this.generateSongEmbedding(id)),
      ).catch((err) =>
        logger().warn("Failed to pre-generate candidate
embeddings", err),
      );
    }

    for (const ce of candidateEmbeddings) {
      const sim = this.cosineSimilarity(tasteVector,
ce.embedding);
      scores.set(ce.song_id, sim);
    }

    return scores;
  }

  async getRecommendations({
    userId,
    limit = 20,
    offset = 0,
    strategy = "mixed",
  }: {
    userId: number;
    limit?: number;
    offset?: number;
    strategy?: "genre" | "social" | "content" | "mixed";
  }): Promise<SongModel[]> {
    logger().info(`Getting recommendations`, { userId, strategy,
limit });

    const { listenedIds, favoriteIds, dislikedIds } =
      await this.getUserInteractedSongIds(userId);

    const excludeIds = new Set([...listenedIds,
...dislikedIds]);

    const [genreScores, socialScores] = await Promise.all([
      strategy !== "social"
        ? this.getUserGenreScores(userId)
        : Promise.resolve(new Map<number, number>()),
      strategy !== "genre" && strategy !== "content"
        ? this.getFriendSongScores(userId)
        : Promise.resolve(new Map<number, number>()),
    ]);

    const candidates = await SongModel.query()
      .where(`${SongModel.tableName}.is_public`, true)
```

```
      .modify((builder) => {
        if (excludeIds.size > 0) {
          builder.whereNotIn(`${SongModel.tableName}.id`,
[...excludeIds]);
        }
      })
      .withGraphFetched("genres");

    const candidateIds = candidates.map((s) => s.id);

    const userTopSongIds = [
      ...[...favoriteIds],
      ...[...listenedIds].slice(0, 20),
    ];

    const contentScores =
      strategy === "content" || strategy === "mixed"
        ? await this.getContentScores(userTopSongIds,
candidateIds)
        : new Map<number, number>();

    const maxGenre = Math.max(1, ...genreScores.values());
    const maxSocial = Math.max(1, ...socialScores.values());
    const maxContent = Math.max(1, ...contentScores.values());

    const scored = candidates.map((song) => {
      const genreRaw =
        song.genres?.reduce(
          (sum: number, g) =>
            sum + (genreScores.get(g.id as unknown as number) ??
0),
          0,
        ) ?? 0;

      const genreNorm = genreRaw / maxGenre;
      const socialNorm = (socialScores.get(song.id) ?? 0) /
maxSocial;
      const contentNorm = (contentScores.get(song.id) ?? 0) /
maxContent;

      let score: number;
      if (strategy === "genre") score = genreNorm;
      else if (strategy === "social") score = socialNorm;
      else if (strategy === "content") score = contentNorm;
      else score = 0.5 * contentNorm + 0.3 * genreNorm + 0.2 *
socialNorm;

      return { song, score };
    });

    return scored
      .sort((a, b) => b.score - a.score)
```

```
      .slice(offset, offset + limit)
      .map((s) => s.song);
  }
}
```

Додаток Д

Алгоритм косинусної подібності

```
private cosineSimilarity(a: number[], b: number[]): number {
  let dot = 0,
    magA = 0,
    magB = 0;
  for (let i = 0; i < a.length; i++) {
    dot += a[i] * b[i];
    magA += a[i] * a[i];
    magB += b[i] * b[i];
  }
  const denom = Math.sqrt(magA) * Math.sqrt(magB);
  return denom === 0 ? 0 : dot / denom;
}
```

Додаток Е

Сервіс друзів

```
@injectable()
export class FriendsService {
  constructor(
    @inject(NotificationService)
    private notificationService: NotificationService,
  ) {}

  public async getFriends({
    limit,
    offset,
    search,
    ids,
    userId,
  }: z.infer<typeof getAllFriendsSchema>) {
    const friends = await FriendModel.query()
      .where("user_id", userId)
      .where("status", FriendStatus.accepted)
      .withGraphFetched("friend")
      .modify((builder) => {
        if (ids?.length) {
          builder.whereIn("friend_id", ids);
        }

        if (search) {
          builder.whereExists(
            FriendModel.relatedQuery("friend")
              .whereILike("name", `%${search}%`)
              .orWhereILike("email", `%${search}%`),
          );
```

```
        }

        if (offset) {
          builder.offset(offset);
        }

        if (limit) {
          builder.limit(limit);
        }
      })
      .orderBy("id", "desc");

    return friends;
  }

  public async addFriend({
    friendId,
    userId,
  }: z.infer<typeof addFriendSchema>) {
    await FriendModel.query().insert({
      user_id: userId,
      friend_id: friendId,
      status: FriendStatus.pending,
    });

    const notificationType = await NotificationTypeModel.query()
      .where("code", "friend_request")
      .first();

    const sender = await UserModel.query().where("id",
userId).first();

    if (notificationType && sender) {
      await this.notificationService.createNotification({
        typeId: notificationType.id,
        recipientId: friendId,
        senderId: userId,
        relatedEntityType: "friend_request",
        relatedEntityId: null,
        title: "New Friend Request",
        message: `${sender.username} sent you a friend request`,
      });
    }
  }

  public async acceptFriendRequest({
    requestId,
    userId,
  }: z.infer<typeof acceptFriendRequestSchema>) {
    await FriendModel.transaction(async (trx) => {
      const request = await FriendModel.query(trx)
        .where("id", requestId)
```

```
        .andWhere("friend_id", userId)
        .first();

      if (!request) {
        throw new Error("Friend request not found");
      }

      await FriendModel.query(trx).insert({
        user_id: request.friend_id,
        friend_id: request.user_id,
        status: FriendStatus.accepted,
      });

      await FriendModel.query(trx)
        .where("id", requestId)
        .andWhere("friend_id", userId)
        .update({
          status: FriendStatus.accepted,
        });
    });
  }

  public async removeFriend({
    friendId,
    userId,
  }: z.infer<typeof removeFriendSchema>) {
    await FriendModel.query()
      .where("user_id", userId)
      .andWhere("friend_id", friendId)
      .delete();
  }
}
```