

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему: ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ
ВЕБЗАСТОСУНКУ ЗБЕРІГАННЯ ТА ПРОГРАВАННЯ МУЗИКИ
З ЗАСТОСУВАННЯМ КЛІЄНТ-СЕРВЕРНОЇ АРХІТЕКТУРИ.
RESEARCH AND SOFTWARE IMPLEMENTATION OF A WEB
APPLICATION FOR STORING AND PLAYING MUSIC USING A
CLIENT-SERVER ARCHITECTURE
(назва теми)

Виконав: студент 2 курсу, групи КНТ-133м

Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

СИНІЦІН С.С.

(ПРИЗВИЩЕ та ініціали)

Керівник ЛЕОЩЕНКО С.Д.

(ПРИЗВИЩЕ та ініціали)

Рецензент СКРУПСЬКИЙ С.Ю.

(ПРИЗВИЩЕ та ініціали)

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет ФКНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 20__ року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

СИНИЦИНА Семена Сергійовича

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація вебзастосунку зберігання та програвання музики з застосуванням клієнт-серверної архітектури.

Research and Software Implementation of a Web Application for Storing and Playing Music Using a Client-Server Architecture.

керівник проєкту (роботи) д-р.філос., доц. каф. ПЗ, ЛЕОЩЕНКО Сергій Дмитрович

(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 07 ” жовтня 2024 року № 398

2. Строк подання студентом проєкту (роботи) 2 грудня 2024 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Вибір програмного забезпечення та його обґрунтування. 3. Основні рішення щодо реалізації компонентів системи. 4. Керівництво програміста. 5. Дослідження і результати.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) 20 слайдів презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ЛЕОЩЕНКО С. Д., д-р. філос., доц. каф. ПЗ		
Нормоконтролер	ДЕЙНЕГА Л.Ю., ст.виклад.		

7. Дата видачі завдання “3” вересня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2 тижні	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів вирішення задачі.	3 тижні	Розділ 2
4	Розробка архітектури програми.	4 тиждень	Розділ 3
5	Розробка програми.	5-7 тижні	Розділ 4
6	Тестування та експериментальне дослідження програмного забезпечення.	8 тиждень	Розділ 5
7	Оформлення пояснювальної записки та документів до неї.	9-10 тижні	Додатки
8	Нормоконтроль та рецензування.	11 тижні	
9	Захист роботи.	12 тиждень	

Студент

(підпис) Семен СИНИЦИН
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Сергій ЛЕОЩЕНКО
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
152 с., 8 табл., 49 рис., 3 дод., 62 джерел.

MUSIC, HTTP, ANGULAR, SERVER, NEST, FRONTEND, BACKEND,
NODE.JS, TYPESCRIPT, JAVASCRIPT.

Об'єкт дослідження – програмні засоби для використання та зберігання музики.

Предмет дослідження – вебзастосунки для зберігання та відтворення музики.

Мета роботи – створення вебзастосунку на клієнт-серверній архітектурі для зберігання та відтворення музики.

Матеріали, методи та технічні засоби: Персональний комп'ютер під керуванням Windows 11, редактор коду Visual Studio Code, серверне середовище Node.js, фреймворки Angular, Nest.js. База даних Mongo DB. Мова програмування TypeScript. Для тестування – браузер Opera, Mozilla Firefox, Chrome.

Результати. Розроблено вебзастосунок для прослуховування музики, та включає основні функції: облікові записи користувачів, музичний програвач, історію взаємодії, алгоритм рекомендацій, пошук і завантаження аудіо.

Висновки. Розроблений вебзастосунок для прослуховування музики успішно реалізує поставлені цілі. Застосунок протестовано, функції працюють стабільно. Проєкт має потенціал для розширення, включаючи соціальні функції та аналітику.

Галузь використання – відтворення та поширення музики за допомогою мережі інтернет.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 152 pages, 49 figures, 8 tables, 3 appendixes, 62 sources.

MUSIC, HTTP, ANGULAR, SERVER, NEST, FRONTEND, BACKEND, NODE.JS, TYPESCRIPT, JAVASCRIPT.

Object of research is software for using and storing music.

Subject of research is web applications for storing and playing music.

Purpose of the work is to create a web application based on client-server architecture for storing and playing music.

Materials, methods and technical means: Personal computer running Windows 11, Visual Studio Code code editor, Node.js server environment, Angular frameworks, Nest.js. Mongo DB database. TypeScript programming language. For testing – the browser Opera, Mozilla Firefox, Chrome.

The results. A web application for listening to music is developed, and includes the main functions: user accounts, music player, interaction history, recommendation algorithm, audio search and download.

Conclusions. The developed web application for listening to music successfully realizes the set goals. The application has been tested, the functions work stably. The project has the potential to expand, including social features and analytics.

The field of use is the reproduction and distribution of music via the Internet.

ЗМІСТ

	С.
Вступ.....	10
1 Аналіз предметної області.....	12
1.1 Актуальність проблеми	12
1.2 Протоколи HTTP та HTTPS	13
1.3 Огляд аналогів	26
2 Вибір програмного забезпечення та його обґрунтування.....	41
2.1 Аналіз поширених фреймворків для фронтенду	41
2.2 Аналіз поширених фреймворків для бекенду	45
2.3 Аналіз поширених баз даних	49
3 Основні рішення щодо реалізації компонентів системи.....	53
3.1 Опис функціоналу програми.....	53
3.2 Навігація між сторінками застосунку	63
3.3 База даних	64
3.4 Реалізація API controllers.....	68
3.5 Оптимізація програми.....	71
4 Керівництво програміста.....	74
4.1 Характеристики програми.....	74
4.2 Залежності проєкту	77
4.3 Встановлення застосунків локально	82
4.4 Розгортання застосунків на сервері.....	85
4.5 Можливі способи покращення програми	88
5 Дослідження і результати.....	96
5.1 Опис методики проведення дослідження	96
5.2 Unit тестування у Angular.....	96

5.3 Manual тестування.....	103
Висновки	114
Перелік джерел посилання	116
Додаток А Технічне завдання	121
Додаток Б Текст програми	124
Додаток В Слайди презентації.....	142

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
AWS	– Amazon Web Services;
CI/CD	– Continuous Integration / Continuous Deployment;
CSS	– Cascading Style Sheets;
DB	– Database;
DOM	– Document Object Model;
GCP	– Google Cloud Platform;
HTML	– HyperText Markup Language;
HTTP	– HyperText Transfer Protocol;
HTTPS	– HyperText Transfer Protocol Secure;
ID3	– Identification;
JPEG	– Joint Photographic Experts Group;
JS	– JavaScript;
JSON	– JavaScript Object Notation;
JSX	– JavaScript XML;
MIME	– Multipurpose Internet Mail Extensions;
MP3	– MPEG-1 Audio Layer III;
NG	– Angular;
NPM	– Node Package Manager;
PNG	– Portable Network Graphics;
REST API	– Representational State Transfer API;
RxJS	– Reactive Extensions for JavaScript;
SCSS	– Sassy CSS;
SSL	– Secure Sockets Layer;
TLS	– Transport Layer Security;
TS	– TypeScript;

- UML – Unified Modeling Language;
- URI – Uniform Resource Identifier;
- URL – Uniform Resource Locator;

ВСТУП

З розвитком інформаційних технологій та Інтернету наша життєва активність стала невід'ємною частиною цифрового світу. Один із найпопулярніших аспектів цієї взаємодії – це зберігання та відтворення музики. З ростом популярності цифрових медіа та стрімінгових платформ, музика стала легко доступною для мільйонів людей по всьому світу. Однак, для багатьох, особливо для тих, хто бажає організувати свою велику музичну бібліотеку та створити персоналізовані плейлисти, виникає необхідність в розробці вебзастосунків, які забезпечували б ефективне управління аудіоконтентом, пропонуючи зручний інтерфейс.

Метою даної дипломної роботи є розгляд основних аспектів створення музичного вебзастосунку, від вивчення архітектурних рішень та вимог до функціональності, до проєктування та розробки програмного продукту. Основні завдання включають в себе створення інтерфейсу для користувача, реалізацію функціональності завантаження та зберігання аудіоконтенту на сервері, а також можливість відтворення цього контенту на клієнтському пристрої.

Особливу увагу приділено використанню технологій Angular та Nest.js, що відомі своєю ефективністю в розробці сучасних вебзастосунків.

Angular – це популярний фреймворк для розробки клієнтської частини вебзастосунків, який надає повний набір інструментів для створення багатофункціональних інтерактивних інтерфейсів користувача. Nest.js – це фреймворк для розробки серверної частини вебзастосунків, побудований на Node.js, який надає потужні інструменти для створення ефективних та масштабованих серверних застосунків. Разом ці два фреймворки надають повне рішення для створення сучасних вебзастосунків, починаючи з клієнтської частини і закінчуючи серверною логікою.

Також було вивчено та розглянуто кращі практики в галузі зберігання та обробки аудіоконтенту, зокрема розробка серверної інфраструктури для забезпечення безпеки та швидкодії передачі даних.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність проблеми

У сучасному цифровому світі музика стає все більш доступною для широкої аудиторії завдяки різноманітним вебзастосункам, які дозволяють зберігати, організовувати та програвати улюблені музичні треки з будь-якого пристрою, підключеного до інтернету. Особливо важливу роль відіграють вебзастосунки з клієнт-серверною архітектурою, які забезпечують надійність, безпеку та масштабованість системи.

Останні роки спостерігається значне зростання популярності музичних стрімінгових сервісів, таких як Spotify, Apple Music та інших. Це обумовлено не лише зростанням кількості користувачів інтернету, а й зміною способів споживання музичного контенту. Сучасні користувачі бажають мати доступ до своєї музичної бібліотеки з будь-якого місця і в будь-який час, що обумовлює необхідність розробки надійних та зручних вебзастосунків [1].

Збереження музичних файлів на локальних пристроях має ряд недоліків, серед яких ризик втрати даних через пошкодження пристрою або випадкове видалення. Клієнт-серверна архітектура дозволяє уникнути цих проблем, забезпечуючи збереження даних на сервері, що значно підвищує рівень безпеки та надійності зберігання. Використання сучасних методів шифрування та захисту даних гарантує конфіденційність особистої інформації користувачів.

Сучасні музичні вебзастосунки надають можливість персоналізації контенту відповідно до вподобань користувачів. Алгоритми машинного навчання аналізують поведінку користувачів та створюють індивідуальні рекомендації, що підвищує рівень задоволення від використання сервісу.

Розробка вебзастосунків з клієнт-серверною архітектурою вимагає глибоких знань у галузі програмування та системного проектування.

Використання новітніх технологій, таких як хмарні обчислення та мікросервіси, дозволяє створювати масштабовані та високопродуктивні системи. Інноваційні рішення, такі як застосування штучного інтелекту для аналізу музичних вподобань користувачів, відкривають нові можливості для покращення сервісу.

Розвиток музичних вебзастосунків значно впливає на музичну індустрію, надаючи нові способи поширення музики та монетизації контенту. Музиканти та виконавці мають змогу самостійно публікувати свої роботи та взаємодіяти з фанатами безпосередньо через вебзастосунки, що відкриває нові перспективи для творчості та розвитку кар'єри.

Таким чином, розробка вебзастосунку для зберігання та програвання музики з використанням клієнт-серверної архітектури є вкрай актуальною темою, що має широкий спектр застосувань та перспектив для подальшого розвитку як з технічної, так і з комерційної точок зору.

1.2 Протоколи HTTP та HTTPS

HTTP – це протокол прикладного рівня в моделі мережевого зв'язку Open Systems Interconnection (OSI), протокол передачі даних у мережі Інтернет між вебресурсом та сервером. З його допомогою запити користувача (через браузер) передаються серверу, а сервер формує відповіді, які повертає браузеру. Це базовий набір правил передачі [2].

HTTP забезпечує незашифрований обмін даними, що може призвести до перехоплення інформації третьою стороною. Так, наприклад, платіжна інформація, введена на HTTP-сайті, може бути перехоплена зловмисниками під час передачі на сервер.

HTTPS (Hypertext Transfer Protocol Secure) – це той самий протокол з важливим доповненням. HTTPS має розширення, зване SSL (Secure Sockets Layer) [3].

Сертифікат SSL – це технологія захисту даних, яка працює через ключі шифрування. Він розміщується між браузером та сервером [4].

Цей сертифікат забезпечує безпеку передачі даних у мережі. Він дозволяє передавати інформацію в зашифрованому вигляді, щоб запобігти читанню даних. Кожне нове відвідування сайту HTTPS формує захищене з'єднання між браузером і сервером [5].

Оригінальна версія HTTP, випущена в 1996-1997 роках, називалася HTTP/1.1. HTTP/2 та HTTP/3 є оновленими версіями власне протоколу. Система передачі була модифікована для підвищення її ефективності. Наприклад, HTTP/2 обмінюється даними в двійковому, а не текстовому форматі. Це також дозволяє серверам заздалегідь передавати відповіді в кеші клієнтів, а не чекати на новий HTTP-запит. HTTP/3 – відносно нова версія. Мета HTTP/3 – ефективніша підтримка потокової передачі в режимі реального часу та інших сучасних вимог до передачі даних.

HTTPS приділяє пріоритетну увагу питанням безпеки даних у HTTP. Сучасні системи використовують HTTP/2 з SSL/TLS як HTTPS.

1.2.1 Принципи роботи протоколу HTTP

У клієнт-серверній комунікації клієнти надсилають HTTP-запити на сервер та отримують HTTP-відповіді. За допомогою HTTP-запитів GET клієнт може запитувати ресурси із сервера. При надсиланні запиту HTTP клієнт може надати серверу додаткову інформацію, таку як файли cookie або дані авторизації. Щоб надіслати деяку інформацію, наприклад, заповнити контактну форму, виконується запит PUT. Надсилання даних із форм або завантаження зображень на сервер виконується за допомогою HTTP-запитів POST. Сервери надсилають запитані ресурси клієнту з HTTP-відповідями. Якщо клієнт запитує сервер виконати якусь операцію, то з HTTP-відповіддю сервер повертає результат операції або повідомляє про помилку [6].

Так само сервер відправляє різні типи HTTP-відповідей як числових кодів і даних [7].

Нижче наведено деякі приклади:

- 200 – ОК;
- 400 – невірний запит;
- 404 – ресурс не знайдено.

Ця взаємодія у вигляді запиту та відповіді зазвичай проходить невидима для користувачів. Цей спосіб зв'язку використовують браузер та вебсервери [8].

1.2.2 Принципи роботи протоколу HTTPS

HTTPS поєднує запити HTTP та відповіді з технологіями SSL і TLS. Вебсайти HTTPS повинні отримати сертифікат SSL/TLS від незалежного центру сертифікації. Ці вебсайти надсилають сертифікат браузеру, а потім обмінюються даними для встановлення довіри. Також SSL-сертифікат містить криптографічну інформацію, тому сервер та браузери можуть обмінюватися зашифрованими даними.

Процес працює наступним чином:

- клієнт відкриває вебсайт HTTPS, ввівши формат URL-адреси `https://` в адресному рядку браузера;
- браузер перевіряє справжність сайту, запросивши SSL-сертифікат сервера;
- у відповідь сервер надсилає сертифікат SSL, який містить відкритий ключ;
- сертифікат SSL вебсайту підтверджує особистість сервера. Як тільки браузер задоволений, він використовує відкритий ключ для шифрування та надсилання повідомлення, що містить секретний ключ сеансу;

- вебсервер використовує свій закритий ключ для розшифрування повідомлення та отримання ключа сеансу. Потім він шифрує сеансовий ключ і відправляє підтвердження в браузер;
- тепер і браузер, і вебсервер переходять на використання одного і того ж сеансового ключа для безпечного обміну повідомленнями [8].

1.2.3 Порівняння HTTP та HTTPS

Переваги HTTPS перед HTTP:

- безпека – HTTP-повідомлення є звичайним текстом, тому неавторизовані особи можуть легко отримати доступ до них і прочитати їх через Інтернет. Навпаки, HTTPS передає всі дані у зашифрованому вигляді. Коли користувачі надсилають конфіденційні дані, вони можуть бути впевнені, що ніякі треті сторони не зможуть перехопити їх по мережі. Для захисту потенційно конфіденційної інформації, наприклад даних кредитної картки або особистих даних клієнтів, краще вибрати HTTPS;
- сертифікаційний орган – пошукові системи зазвичай оцінюють вміст вебсайтів HTTP нижче, ніж вебсторінки HTTPS, тому що HTTP менш заслуговує на довіру. Клієнти також віддають перевагу вебсайтам HTTPS, а не HTTP. Браузер позначає підключення HTTPS, помістивши значок замка в адресному рядку браузера поруч із URL-адресою вебсайту. Користувачі надають перевагу вебсайтам і програмам HTTPS завдяки цим додатковим факторам безпеки та довіри;
- продуктивність та аналітика – вебпрограми HTTPS завантажуються швидше, ніж HTTP-програми. Також HTTPS краще відстежує реферальні посилання. Реферальний трафік – це трафік на сайт із інших джерел, таких як реклама або зворотні посилання в соціальних мережах. Для того щоб аналітичне програмне забезпечення точно визначало надійні джерела трафіку необхідно включити HTTPS.

Результати порівняння обох протоколів можна побачити в таблиці 1.1.

Таблиця 1.1 – Порівняння HTTP та HTTPS протоколів [8]

Параметр	Протоколи	
	HTTP	HTTPS
Опис	Протокол передачі гіпертексту	Захищений протокол передачі гіпертексту
Базові протоколи	HTTP/1 та HTTP/2 використовують TCP/IP. HTTP/3 використовує протокол QUIC.	Використовує HTTP/2 з SSL/TLS для подальшого шифрування HTTP-запитів та відповідей
Порт	За замовчуванням 80	За замовчуванням 443
Використання	Для старих текстових вебсайтів	Для всіх сучасних вебсайтів
Безпека	Жодних додаткових функцій безпеки	Використовує SSL-сертифікати для шифрування з відкритим ключем
Переваги	Дає можливість спілкування через Інтернет	Підвищує авторитет сайту, довіру та рейтинг у пошукових системах.

1.2.4 Структура HTTP-запиту

HTTP-запит містить такі елементи: рядок запиту, заголовки HTTP-запиту та тіло запиту [6].

Рядок запиту складається з трьох основних частин:

- метод HTTP – це однослівні команди, наприклад GET, POST, PUT, DELETE тощо, які повідомляють серверу, що робити з ресурсами;
- URI (Uniform Resource Identifier) – компонент шляху URL для запиту. Шлях ідентифікує ресурс на сервері. Тобто це адреса ресурсу на сервері, до якого клієнт хоче отримати доступ, наприклад, /index.html;
- версія HTTP – відображення специфікації HTTP, зазвичай HTTP/1.1 або HTTP/2.

Приклад рядка HTTP-запиту:

```
GET /echo HTTP/1.1
```

Заголовки запиту включають інформацію про хост, контент, тип контенту та інші параметри, необхідні для коректної обробки сервером. Заголовки містять метадані, що використовуються для передачі додаткової інформації між клієнтом та сервером. Вони можуть включати інформацію про тип браузера (User-Agent), тип даних, які клієнт може приймати (Accept), тип вмісту запиту (Content-Type), параметри авторизації, cookie (cookies) та багато іншого. Кожен заголовок HTTP складається з пари "ключ: значення". Заголовки HTTP для запиту клієнта містять інформацію, яку сервер може використовувати для прийняття рішення про те, як відповісти на запит [7].

Приклад заголовків HTTP-запиту [9]:

```
Content-Type: application/json
```

Тіло запиту (body) містить дані, що передаються методом application/x-www-form-urlencoded, типовим для форм, що надсилаються через браузер. Тут передаються такі параметри, як anchor, logintoken, username та password [10].

Тіло повідомлення запиту містить тіло об'єкта, яке може бути у вихідному стані або закодовано. Тіла повідомлень підходять для деяких методів запиту та не підходять для інших. Наприклад, запит із методом HTTP POST, який надсилає вхідні дані на сервер, має тіло повідомлення, що містить дані. Запит з методом HTTP GET, який просить сервер відправити ресурс, не має тіла повідомлення.

Коли клієнт (наприклад, браузер) відправляє HTTP-запит серверу, сервер обробляє цей запит і надсилає відповідь (HTTP response), який також має власну структуру зі статусним рядком (status line), заголовками відповіді та, можливо, тілом відповіді.

1.2.5 Структура HTTP-відповіді

HTTP-відповідь є повідомленням, яке сервер надсилає клієнту у відповідь на запит. HTTP-відповідь містить такі елементи: рядок статусу, заголовки відповіді та тіло відповіді.

Стартовий рядок HTTP-відповіді називається рядком статусу (status line), і він містить три елементи:

- версія HTTP – відображення версії протоколу (HTTP/2 або HTTP/1.1);
- код статусу – стан, що вказує на результат запиту;
- пояснення або текст статусу – короткий текстовий опис до коду стану, узагальнює значення коду статусу у зрозумілому для людини тексті.

Приклад рядка статусу відповіді HTTP:

HTTP/1.1 200 OK

Повний список кодів стану доступний у специфікації до протоколу, нижче у таблиці 1.2 наведено лише найпоширеніші коди відповідей [11].

Таблиця 1.2 – Опис найпоширеніших кодів відповідей HTTP/HTTPS протоколу

Категорія	Опис
1	2
200 OK	Повертається у разі успішної обробки запиту, причому тіло відповіді зазвичай містить запитаний ресурс.
302 Found	Перенаправляє клієнта на інший URL. Наприклад, цей код може прийти, якщо клієнт успішно пройшов процедуру автентифікації і тепер може перейти на сторінку свого облікового запису.
400 Bad Request	Цей код можна побачити, якщо запит сформовано з помилками. Наприклад, у ньому були відсутні символи завершення рядка.

Продовження таблиці 1.2

1	2
403 Forbidden	Означає, що клієнт не має достатніх прав доступу до запитаного ресурсу. Також цей код можна зустріти, якщо сервер виявив шкідливі дані, надіслані клієнтом у запиті.
404 Not Found	Цей код можна побачити, якщо запитати у сервера ресурс, якого немає на сервері.
500 Internal Error	Цей код повертається сервером, коли він не може з певних причин обробити запит.

Response Headers, або заголовки відповіді, використовуються для того, щоб уточнити відповідь, і ніяк не впливають на вміст тіла. Ця інформація може допомогти клієнту відобразити відповідь користувачу та зберегти (або кешувати) відповідь для майбутнього використання. Якщо запит не вдається, заголовки можуть повідомити клієнта, що потрібно зробити для успішної відповіді. Заголовки існують у форматі «Ім'я-Значення» з двокрапкою (:) в якості роздільника [12].

Приклад заголовків відповіді HTTP:

`Content-Type: text/html; charset=utf-8`

В таблиці 1.3 показано заголовки, що найчастіше зустрічаються у відповідях.

Таблиця 1.3 – заголовки відповідей

Категорія	Опис
Server	Містить інформацію про сервер, який обробив запит. Наприклад: Server: nginx.
Set-Cookie	Містить куки, потрібні для ідентифікації клієнта. Браузер парсить куки та зберігає їх у своєму сховищі для подальших запитів. Наприклад: Set-Cookie:PHPSSID=bf64982f
WWW-Authenticate	Повідомляє клієнта про тип аутентифікації, який необхідний для доступу до запитуваного ресурсу. Наприклад: WWW-Authenticate: BASIC realm="localhost"

Незважаючи на те, що у більшості відповідей тіло є присутнім, воно не є обов'язковим. Виняток є випадки, коли сервер відповідає на клієнтський запит за допомогою методу HTTP HEAD (який запитує заголовки, але не тіло відповіді) і коли сервер використовує певні коди статусу. Наприклад, у кодів "201 Created" або "204 No Content" тіло відсутнє, оскільки достатню інформацію для відповіді на запит вони передають у заголовку [13].

1.2.5 Загальні заголовки HTTP та їх використання

Є три практичних використання заголовків HTTP:

- безпека – посилити веббезпеку можна за допомогою таких заголовків, як Content-Security-Policy, який контролює ресурси, які дозволено завантажувати вебпереглядачу, і X-Frame-Options, який запобігає атакам клацання;

- стратегії кешування – покращити продуктивність і зменшити навантаження на сервер можна за допомогою Cache-Control, який керує тим, як кешуються ресурси, і ETag, який допомагає браузерам перевіряти кешовані ресурси;

- узгодження вмісту – можна ефективно керувати форматами даних за допомогою Accept, який повідомляє серверу, які типи вмісту може обробляти клієнт, і Content-Type, що визначає тип медіа відповіді.

Є багато заголовків HTTP, які використовуються для доставки цієї конкретної сторінки. Їх можна перевірити на вкладці "Мережа веббраузера". Нижче буде розписано заголовки, що найчастіше використовують.

Заголовок Content-Type визначає медіа-тип ресурсу у відповіді HTTP. Він інформує одержувача, як інтерпретувати та обробляти отримані дані.

Приклад:

```
Content-Type: application/json
```

Припустимо, створюється RESTful API, і клієнт надсилає HTTP-запит для отримання даних. Сервер відповідає заголовком Content-Type, встановленим як application/json. Це сигналізує клієнту, що тіло відповіді містить дані JSON, дозволяючи клієнту проаналізувати та обробити їх відповідним чином.

Заголовок авторизації використовується для автентифікації в HTTP. Він містить облікові дані, як правило, у формі маркера або облікових даних, щоб підтвердити особу клієнта.

Приклад:

```
Authorization: Bearer token321
```

У контексті автентифікації OAuth2 клієнт запитує доступ до захищеного ресурсу з сервера. Сервер очікує, що клієнт міститиме заголовок авторизації з дійсним маркером доступу (наприклад, Bearer token321). Якщо маркер дійсний, сервер надає доступ до запитуваного ресурсу [14].

Заголовок User-Agent визначає клієнтське програмне забезпечення або агент користувача, який ініціював запит HTTP. Це допомагає серверам адаптувати свої відповіді на основі можливостей або характеристик клієнта. Приклад:

```
User-Agent: Mozilla/6.0 (Windows NT 10.0; Win64;  
x64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/94.0.4607.81 Safari/538.37
```

Вебсервер отримує HTTP-запит із заголовком User-Agent, який вказує на те, що клієнт використовує браузер Google Chrome в операційній системі Windows. Сервер може використовувати цю інформацію для оптимізації макета вебсторінки та функцій для цього конкретного браузера [15].

Заголовок `Accept` інформує сервер про типи медіафайлів, які клієнт може обробляти або яким надає перевагу. Він направляє сервер у виборі відповідного формату відповіді.

Приклад:

```
Accept: application/json, text/html
```

Клієнт надсилає HTTP-запит на сервер доставки вмісту, включаючи заголовок `Accept`, що вказує на його перевагу для отримання даних у форматі JSON (`application/json`) або HTML (`text/html`). Сервер відповідає запитуваними даними у вказаному форматі [16].

Заголовок `Cache-Control` контролює та спрямовує механізми кешування, дозволяючи клієнтам і проміжним серверам кешувати відповіді та оптимізувати майбутні запити.

Приклад:

```
Cache-Control: max-age=3600, public
```

Вебсайт електронної комерції встановлює для заголовка `Cache-Control` на своїх зображеннях продуктів значення `max-age=3600`, вказуючи, що ці зображення можуть кешуватися клієнтами та проміжними серверами протягом однієї години. Це зменшує навантаження на сервер і прискорює час завантаження сторінки для відвідувачів, які повертаються. Числове значення тут представляє секунди [17].

Заголовок `Cookie` надсилається клієнтом, щоб включити файли `cookie` (невеликі фрагменти даних), які раніше зберігалися на стороні клієнта. Заголовок `Set-Cookie` використовується сервером для встановлення або оновлення файлів `cookie` на клієнті.

Приклад запиту клієнта:

```
Cookie: sessionId=34567; username=johnsmith
```

Приклад відповіді сервера:

```
Set-Cookie: sessionId=84250; expires=Thu, 26 Oct 2023 19:00:00 GMT; path=/; domain=.example.com
```

Користувач входить на платформу соціальних мереж, і сервер відповідає заголовком Set-Cookie, що містить ідентифікатор сеансу та термін дії. Клієнт зберігає цей файл cookie та надсилає його назад на сервер із кожним наступним запитом через заголовок Cookie, щоб підтримувати сеанс користувача [18].

Заголовок Content-Length вказує на розмір (у байтах) тіла повідомлення в HTTP-запиті або відповіді. Це допомагає одержувачам визначити, коли вони отримали повне повідомлення.

Приклад:

```
Content-Length: 1024
```

У сценарії завантаження файлу клієнт надсилає запит HTTP POST на сервер, щоб завантажити файл розміром 1 МБ. Сервер включає в запит заголовок Content-Length, який вказує на те, що розмір тіла повідомлення становить 1 024 000 байт. Це дозволяє серверу виділяти необхідні ресурси для точної обробки вхідних даних [19]-[20].

1.2.6 MIME типи

MIME (Multipurpose Internet Mail Extension, Багатоцільові розширення пошти Інтернету) – стандарт, що вказує формат файлу, фундаментальна характеристика цифрового ресурсу, яка впливає на його здатність бути

доступним та використовуватися з часом. Це специфікація для передачі по мережі файлів різного типу: зображень, музики, текстів, відео, архівів та ін. Вказівка MIME-типу використовується в HTML зазвичай при передачі даних форм і вставки на сторінку різних об'єктів в Content Types [21].

Типи вмісту можна загалом класифікувати на кілька категорій. Немає офіційної класифікації, наступна класифікація базується на шаблоні використання:

- Text Types – використовуються для представлення простого тексту, вмісту HTML, таблиць стилів CSS і коду JavaScript. Прикладами є text/plain, text/html, text/css, text/javascript, тощо;
- Image Types – використовується для різних форматів зображень. Наприклад: image/jpeg, image/png, image/gif, тощо;
- Audio та Video Types – використовується для обробки аудіо- та відео- файлів. Наприклад: audio/mpeg, video/mp4, тощо;
- Application Types – використовується для різних видів двійкових даних та документів, таких як корисні дані JSON в API, дані XML та PDF. Наприклад: application/json, application/xml, application/pdf, тощо;
- Multipart Types – використовується для підтримки даних, які розділені на частини, наприклад для завантаження файлів. Це може бути multipart/form-data, multipart/byteranges, тощо;
- Message Types – використовується для інкапсуляції таких повідомлень, як запити/відповіді HTTP або повідомлення електронної пошти. Прикладом є message/http, message/rfc822 [22].

В таблиці 1.4 можна побачити найпоширеніші MIME типи.

Таблиця 1.4 – найпоширеніші MIME типи [23]

Ім'я формату	MIME-type	Розширення файлу
1	2	3
Зображення		

Продовження таблиці 1.4

1	2	3
JPEG	image/jpeg	.jpeg
PNG	image/png	.png
GIF	image/gif	.gif
Generic Bitmap	image/bmp	.bmp
Microsoft Bitmap	image/x-ms-bmp	.bmp
Текст		
PDF	application/pdf	.pdf
JSON	application/json	.json
Відео		
MPEG-4 Video	video/mp4	.mp4
Open Web Media Project – Video	video/webm	.webm
M4v	video/x-m4v	.m4v
Quicktime video	video/quicktime	.mov, .qt
Аудіо		
MP3 or other MPEG format	audio/mpeg	.mp3
Waveform Audio	audio/x-wav	.wav, .wave

У розробленій програмі для зберігання та програвання музики користувачі можуть завантажувати музичні файли у форматі MP3. Фронтенд застосунку парсить дані тегів (наприклад, ID3 теги) з файлу MP3, щоб отримати інформацію про трек, таку як назва, виконавець, альбом та обкладинка (cover). Це дозволяє автоматично відображати метадані треку і обкладинку альбому у вебінтерфейсі [24].

1.3 Огляд аналогів

Для розробки вебзастосунку було обрано такі існуючі аналоги: Spotify, YouTubeMusic, Deezer та SoundCloud.

1.3.1 Spotify

Spotify – це цифровий музичний сервіс, який надає доступ до мільйонів пісень без реклами. Можна використовувати Spotify на будь-якому пристрої, включаючи комп'ютери, телефони, планшети та інші [25].

Spotify пропонує цифровий записаний аудіоконтент з обмеженими авторськими правами, у тому числі понад 100 мільйонів пісень та п'ять мільйонів подкастів від звукозаписних компаній та медіа-компаній. У межах умовно-безкоштовної послуги базові функції безкоштовні з рекламою та обмеженим контролем, а додаткові функції, такі як прослуховування в автономному режимі та прослуховування без реклами, пропонуються за платною підпискою. Користувачі можуть шукати музику за виконавцем, альбомом або жанром, а також створювати, редагувати та ділитися плейлистами [26].

Spotify доступний на більшості пристроїв, включаючи комп'ютери під керуванням Windows, macOS та Linux, смартфони та планшети iOS та Android, пристрої розумного будинку, такі як лінійки продуктів Amazon Echo та Google Nest, а також цифрові медіаплеєри, такі як Roku. Приклад інтерфейсу можна побачити на рисунку 1.1.

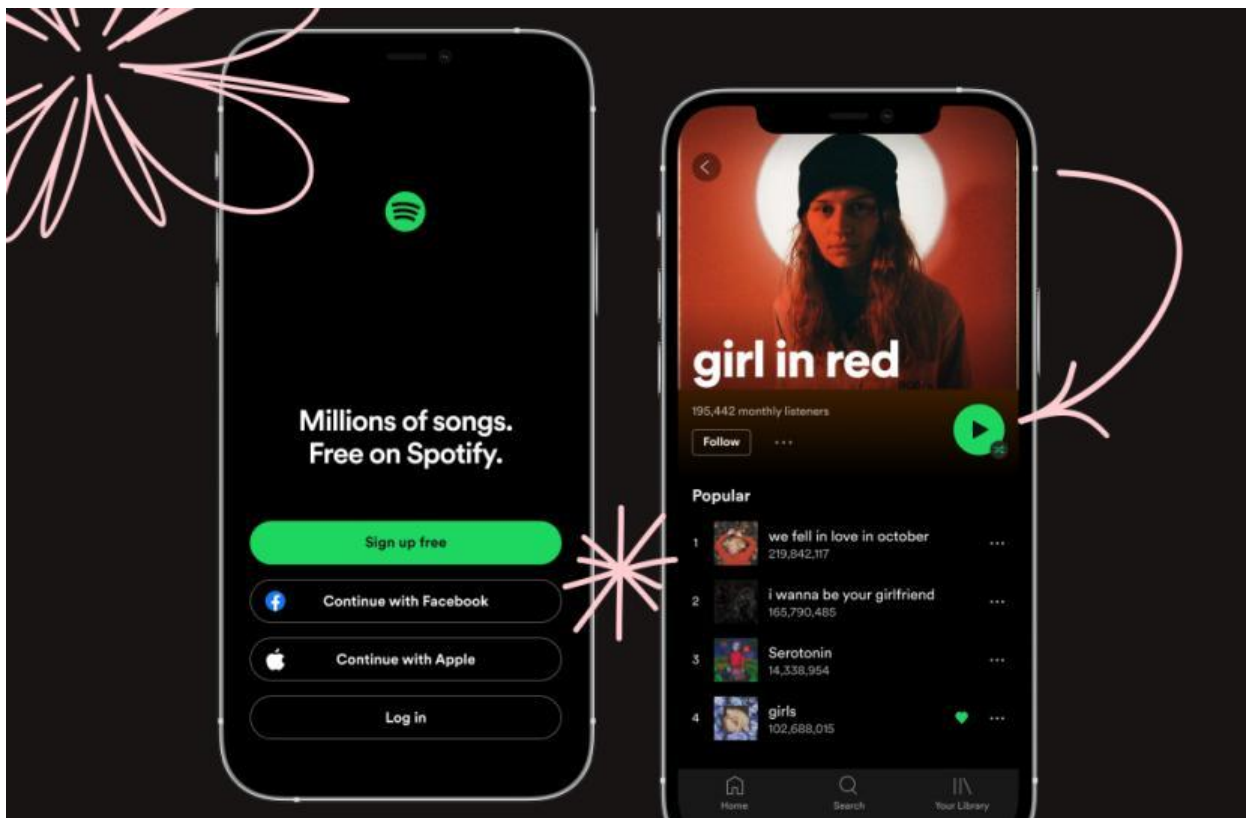


Рисунок 1.1 – Інтерфейс Spotify [27]

До переваг застосунку віднесено:

- простий інтерфейс, а саме навігація всіма його функціями, такими як списки відтворення, відкриття нової музики, прослуховування радіо та багато іншого;
- працює незалежно від платформи, можна підкреслити справедливу та збалансовану доступність платформи. Spotify на сьогоднішній день є найбільш незалежною від платформи доступною службою потокового аудіо та пропонує досвід аудіомузики на пристроях iOS, Android або Windows. Він також сумісний з найновішими версіями операційної системи. Можна використовувати лише один обліковий запис для керування всіма своїми пристроями;
- велика музична бібліотека. Користувачі можуть вибирати між безкоштовною та преміальною версіями. Незважаючи на це, користувачі мають доступ до значних музичних архівів. Крім того, приблизно 3,2

мільйона подкастів наразі пропонуються на Spotify, і ще більше додається щодня. Одна підписка на преміум функції працює на п'яти різних пристроях. Крім того, можливе прослуховування офлайн зі збереженими списками відтворення;

– обмін музикою та інтеграція програм. Безкоштовний обліковий запис Spotify можна пов'язати з Facebook або Twitter. Друзі та родина можуть бачити, що слухає користувач, якщо він підключив свій Spotify до інших платформ соціальних мереж. Користувачі можуть легко ділитися списками прослуховування та піснями на вимогу. Spotify дозволяє підключати його до інших програм і сервісів, таких як Rdio, Songkick, Deezer і Google Play. Ці мобільні програми можуть керувати музичною бібліотекою Spotify, що означає, що користувач може отримати доступ як до колекції програми, так і до музики у своєму обліковому записі Spotify, що є однією з головних переваг Spotify;

– функція радіо, яке збирає колекцію музики на основі музиканта, стилю музики, чи настрою, доступне з безкоштовним або платним обліковим записом. Ця функція чудова для людей, які не впевнені, що вони хочуть слухати, і їм потрібно щось, щоб показати їм схожі пісні;

– пропонує безкоштовні та преміум плани. На відміну від інших музичних потокових сервісів, Spotify не вимагає від нових користувачів негайної оплати преміум підписки. Користувач може продовжувати транслювати свою улюблену музику, створювати списки відтворення та знаходити нові аудіо за допомогою безкоштовного облікового запису Spotify. Більшість людей люблять тестувати послугу, перш ніж вирішити, платити чи ні за преміум послуги;

– Spotify Connect, одне з останніх удосконалень, надає нову можливість передавати музику через аудіосистему. Можна підключити планшет або смартфон до потокових пристроїв, таких як Chromecast, Apple

TV і Amazon Fire TV. Також можна слухати музику в дорозі за допомогою динаміків Bluetooth;

– Spotify чудово підходить для артистів. Користувачі можуть завантажити свої творіння на Spotify, незалежно від того, чи є вони музикантами-початківцями, розважальними подкастерами або колосальними виконавцями звукозапису. Spotify може допомогти як малим, так і видатним виконавцям створити аудиторію, збільшити прибутки в Інтернеті, зменшити піратство, розширити їхній репертуар і підвищити свої позиції в ключових чартах по всьому світу. Низька винагорода та упереджені рейтинги є двома найпоширенішими проблемами серед виконавців, які публікують публікації на Spotify. Spotify допоміг багатьом незалежним виконавцям і подкастерам у самовиданні та отриманні значного доходу від їхньої роботи [26].

До недоліків застосунку віднесено:

– вартість. Іноді люди сперечаються про те, чи варто оформлювати преміальну підписку на Spotify за 9,99 доларів. Хоча користувачі можуть слухати необмежену кількість музики, безкоштовне членство Spotify обмежує певною кількістю треків і містить рекламу, яка заважає музичному досвіду. Це головний недолік Spotify;

– наявність країни. Spotify все ще недоступний у багатьох регіонах. Деякі користувачі використовують служби VPN, щоб продовжити користуватися послугою, але більшість людей не підуть на це. Хоча його було запущено в 56 країнах, не всі з них мають однакову доступність. Spotify, наприклад, доступний у Болівії, Еквадорі, США, Канаді, Киргизстані, Туреччині та Україні, але не в Індії та Непалі;

– користувачі не можуть володіти піснями. Spotify Premium дозволяє транслювати та відтворювати музику без доступу до Інтернету. Однак користувачі не можуть завантажити та прослухати пісню;

– у безкоштовній версії немає опції без реклами. У Spotify є безкоштовні облікові записи, які вимагають від користувачів ввести дані

своїєї кредитної картки або приєднатися через Facebook, але під час гри на них відображається реклама. Деякі люди вважають рекламні ролики дратуючими, тоді як інші можуть захотіти їх переглянути, оскільки вони служать функцією перемішування. Навіть якщо вибрати версію Spotify із підтримкою реклами, час від часу можна помічати низьку якість звуку, що є одним із основних недоліків музики Spotify;

– рейтинг пісень. Spotify оцінює пісні на основі їх популярності, а не якості. Це означає, що популярні або активно рекламовані пісні звукозаписуючих компаній швидко з'являться в результатах пошуку. Таким чином важко знайти відповідні пісні, тому що багато користувачів ігнорують їх через те, що вони нерозголошенні або непопулярні;

– низька якість звуку для безкоштовних облікових записів. Користувачі з безкоштовними обліковими записами також стикаються з низькою якістю аудіопотоку при оптимальному бітрейті 128 Кбіт/с на комп'ютері та 160 Кбіт/с на мобільному пристрої. Якщо потрібна високоякісна потокова передача, преміум-опція пропонує потокову аудіо музику зі швидкістю 320 Кбіт/с на мобільних телефонах і комп'ютерах [28].

1.3.2 YouTube Music

YouTube Music – це сервіс потокової передачі музики, розроблений Google. Він був запусканий у травні 2018 року та розроблений, щоб конкурувати з іншими популярними платформами потокового передавання музики, такими як Spotify та SoundCloud. YouTube Music пропонує величезний каталог пісень і музичних відео, до яких користувачі можуть отримати доступ безкоштовно за допомогою реклами або через платну підписку [29]. Приклад інтерфейсу можна побачити на рисунку 1.2.

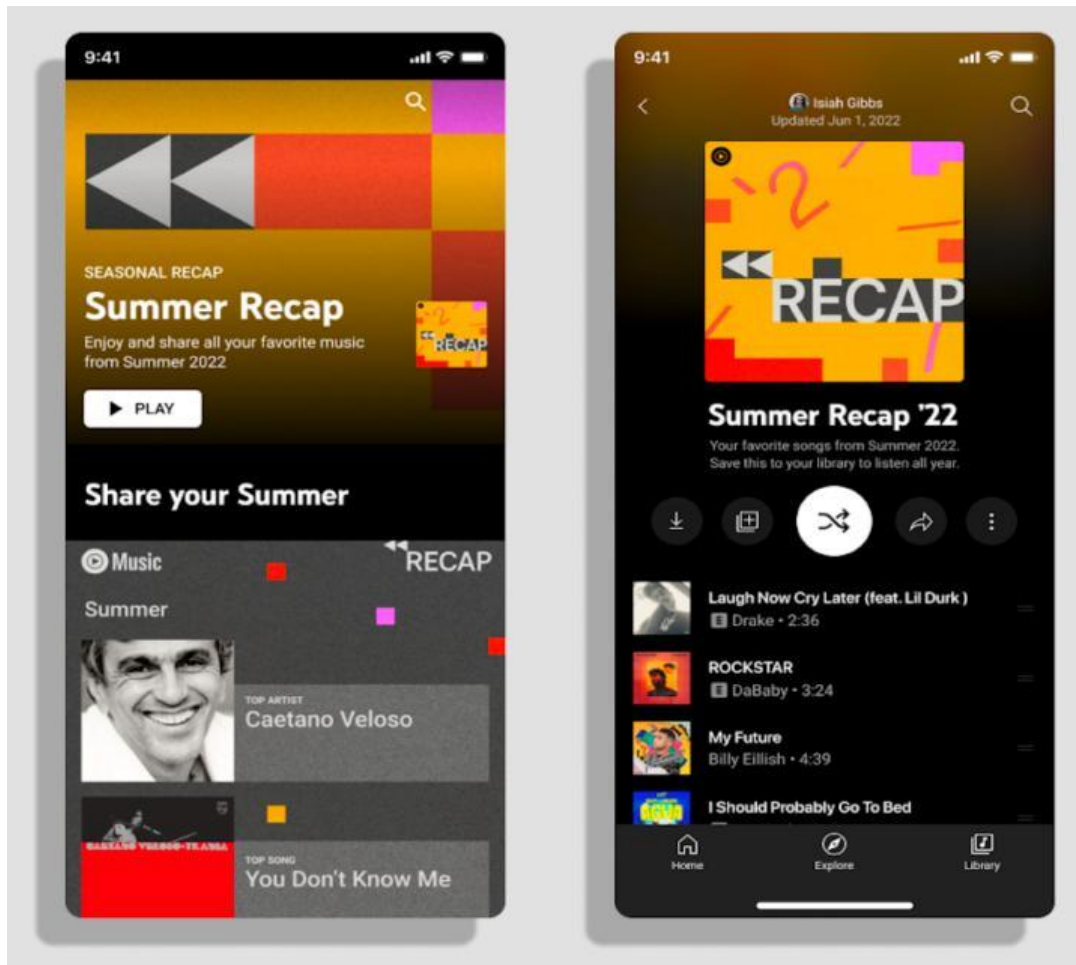


Рисунок 1.2 – Інтерфейс YouTube Music [30]

До переваг застосунку віднесено:

– значущість його музичної колекції. Однією з головних переваг користування музичним сервісом YouTube Music є безсумнівно його вражаюча музична бібліотека. З легкістю можна знайти практично будь-яку композицію, що приходить на думку. На відміну від стандартних синглів, альбомів та плейлистів, що зосереджені навколо видатних артистів цієї галузі, YouTube Music також надає можливість знаходити місцевих виконавців і мікстейпи. Схожий на Spotify, YouTube Music володіє вражаючим асортиментом понад сімдесяти мільйонів музичних треків. Проте, слід зазначити, що YouTube Music вирізняється своєю унікальною здатністю надавати індивідуально підібрані рекомендації для треків,

виконавців та альбомів. Ця особливість пов'язана з використанням алгоритму, ідентичного тому, який використовується на платформі YouTube;

– реклама, яку можна пропустити, у безкоштовній версії. Специфічно для безкоштовної версії провідних музичних стрімінгових сервісів, що безкоштовна версія YouTube Music включає рекламу, але відзначається можливістю її пропускати, що є унікальністю цієї платформи на тлі інших. Додатково, користувачі можуть безліч разів пропускати треки та переміщати пісні у відтворенні. Важливо зауважити, що ця можливість недоступна в безкоштовній версії Spotify, надаючи YouTube Music значний конкурентний перевагу. Після оновлення до преміум плану користувачі отримують доступ до функцій прослуховування в автономному режимі, фонового відтворення та режиму "картинка в картинці" для відео;

– можливість перегляду музичних відео. Платформа відзначається наявністю вбудованої підтримки відтворення відео, що надає велику перевагу для тих, хто віддає перевагу одночасному програванню музичних відомостей і відтворенню відео. Користувачі можуть обирати між двома режимами: режимом пісні та режимом відео. Для активації режиму відео достатньо знайти бажану пісню і вибрати відповідну опцію в верхній частині екрану. Цей підхід дозволяє насолоджуватися музикою на всіх рівнях, об'єднуючи аудіо та відео в єдиному просторі [31].

До недоліків застосунку віднесено:

– пропонує нижчу якість потокового передавання. Завдяки поширенню навушників із шумопоглинанням і таких функцій, як Spatial Audio та Lossless Playback, музика високої якості стала доступнішою, ніж будь-коли. Spotify забезпечує максимальний бітрейт 320 Кбіт/с для відтворення аудіо, а YouTube Music – 256 Кбіт/с. Хоча це може здатися не головною різницею, але якщо дбати про найкращі враження від прослуховування, це варто зауважити;

– у безкоштовній версії YouTube Music немає фонового відтворення музики. Хоча безкоштовна версія Spotify також досить дратує, принаймні вона має фонове відтворення. Той факт, що безкоштовна версія YouTube Music не підтримує його, означає, що музика припинить відтворення, коли користувач вийде з програми. Це незрозуміле рішення Google і головний недолік для тих, хто хоче випробувати безкоштовну версію [32].

1.3.3 Deezer

Deezer – це сервіс потокової передачі музики, який надає користувачам доступ до величезного каталогу пісень, альбомів і списків відтворення. Він був заснований у Парижі, Франція, у 2007 році та став одним із головних гравців у світовій індустрії потокового передавання музики, конкуруючи з такими сервісами, як Spotify і YouTube Music [33]. Приклад інтерфейсу Deezer можна побачити на рисунку 1.3.

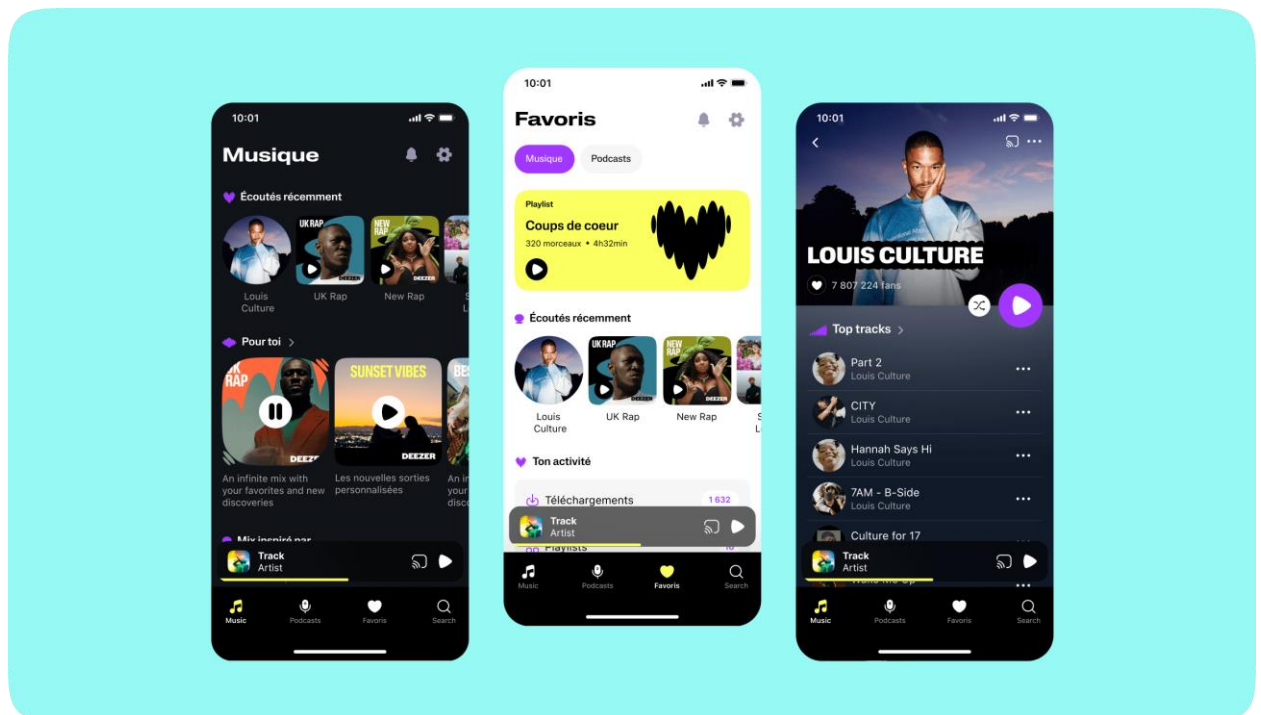


Рисунок 1.3 – Інтерфейс Deezer [34]

До переваг застосунку віднесено:

- доступність і величезна колекція музики. Deezer доступний у 182 країнах. Він містить понад 73 мільйони пісень у своєму каталозі, що робить його одним із найбільших сервісів потокового передавання музики;

- функції відтворення і персоналізації. Deezer має функцію лірики, за допомогою якої користувачі можуть бачити текст пісні під час відтворення вибраного треку. Deezer пропонує схожу на радіо службу під назвою Flow, яка відтворює випадкову музику на основі улюблених треків користувача. Також є вбудована функція ідентифікації музики;

- доступний на великій кількості платформ. Deezer доступний на широкому діапазоні пристроїв, таких як Windows, Mac, iOS, Android, Windows Mobile, Smart TV і має вебпрограму, але з преміум підпискою;

- можливості відтворення музики. Можна завантажувати та відтворювати вибрану музику в режимі офлайн. А також імпортувати файл MP3 для прослуховування як звичайну пісню та додати його до своїх списків прослуховування;

- підтримка подкастів і вебверсія. Користувачам ПК не потрібно завантажувати Deezer, оскільки вебверсія така сама, як і програма для комп'ютера. У 2015 році Deezer почав включати подкасти для кількох країн, а з тих пір розширив їх для всіх країн і створив спеціальний розділ, що пропонує 160 тисяч подкастів [35].

До недоліків застосунку віднесено:

- не всі треки прослуховуються. Деякі пісні пропускаються, навіть для преміум користувачів;

- обмеження в прослуховуванні музики. Можна слухати лише 30-секундні кліпи пісень без підписки;

- неякісні записи та невідтворення їх до кінця. Багато пісень не відтворюються до останніх акордів, а багато інших є записами дуже низької якості, які демонструють спотворення звуку [36].

1.3.4 SoundCloud

SoundCloud – це онлайн платформа потокового передавання музики, яка дозволяє як відомим, так і незалежним артистам завантажувати, рекламувати та ділитися своєю музикою. Він був заснований у 2007 році та став одним із найбільших сервісів потокового передавання музики в усьому світі, відомим своєю зосередженістю на новітній та андеграундній музиці [37]. Приклад інтерфейсу Spotify можна побачити на рисунку 1.4.

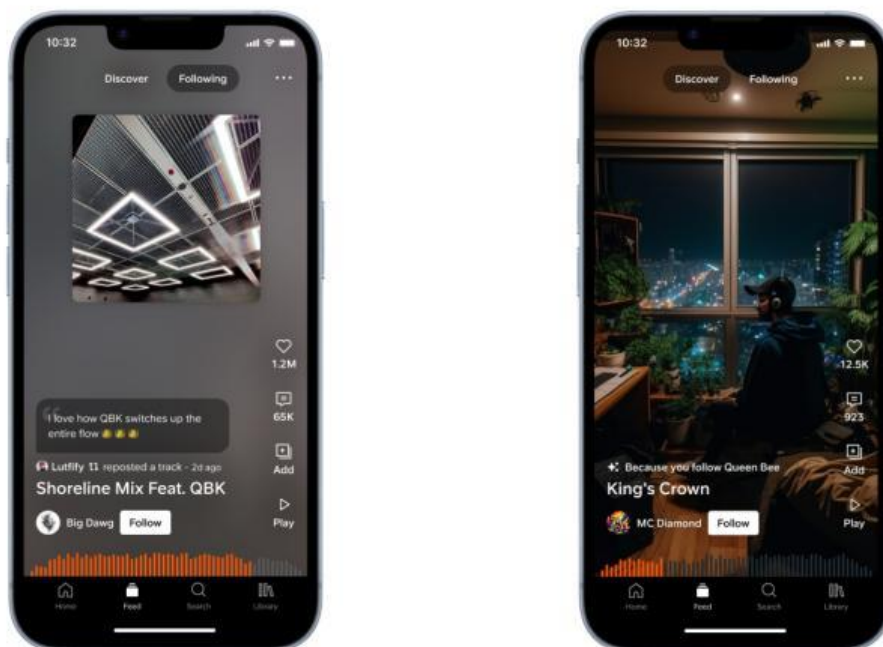


Рисунок 1.4 – Інтерфейс SoundCloud [38]

До переваг застосунку віднесено:

– зручність. SoundCloud простий у використанні. Він має чудові можливості навігації та дозволяє легко обмінюватися вбудованими посиланнями на файли між платформами. Для новачка, який шукає ефективний спосіб завоювати більшу аудиторію та можливість заробляти на своїй роботі з мінімальними інвестиціями на соціальній платформі, цей вебсайт справді повний плюсів;

– підтримує завантаження аудіо та відеофайлів. Більшість інших платформ обміну музикою – це переважно візуальні платформи. Хоча можливо завантажити відео для окремого музичного файлу, часто фінансово не вигідно створювати та завантажувати якісні музичні відео для кожного файлу, створеного виконавцем. Для музичного виконавця-початківця можливість завантажувати подкасти, аудіофайли та файли змішаного формату за допомогою того самого набору інструментів є перевагою;

– легко отримати доступ на різних пристроях. Отримати доступ до файлів, завантажених у SoundCloud, можна з однаковою легкістю через ноутбуки, настільні комп'ютери та мобільні пристрої. SoundCloud доступний як через програми для Android, так і для iOS і сумісний з кількома браузерами. Це дозволяє слухачам відтворювати музику виконавця будь-де та будь-коли. Виконавцям-початківцям не варто боятися втрати потенційних постійних слухачів через вибір платформи для обміну музикою за допомогою SoundCloud;

– вебсайт з глобальним охопленням. Музичні лейбли по всьому світу постійно шукають платформу для нових талантів. Припустимо, що у виконавця є якісний контент і він може зібрати достатню кількість прослуховувань SoundCloud, щоб зробити свою музику трендовою. У такому випадку треки з більшою ймовірністю будуть виявлені шукачами талантів, продюсерами, блогерами та діджеями на SoundCloud, ніж на більшості інших платформ потокового передавання музики;

– дозволяє виконавцю заробляти на своїх треках. Якщо контент може генерувати та підтримувати кілька органічних прослуховувань SoundCloud, виконавці можуть заробляти гроші та популярність на своїх завантажених файлах;

– створює списки для прослуховування. Якщо виконавець позначає свою роботу правильними мітками, у нього є високий шанс потрапити в список прослуховування, створений SoundCloud для його слухачів.

Автоматично створені списки прослуховування, як «Прослухати схожі треки», «50 найкращих» або «Нові треки», пропонують музичній спільноті як досвідчених, так і нових виконавців і, таким чином, надають новим музикантам більше можливостей для реклами;

- SoundCloud надає кращі можливості для соціальних мереж порівняно з більшістю інших платформ. Це може бути на користь новачкам, оскільки виконавець має створити базу шанувальників як онлайн, так і офлайн із нуля. Слухачі можуть лайкати, ділитися та репостити файли. Слухачі також можуть робити закладки для улюблених і завантажувати вибрані треки виконавця. Це забезпечує органічне просування саундтреків автора [39].

До недоліків застосунку віднесено:

- якість контенту. Хоча SoundCloud має широкий спектр музики, включаючи незалежну та експериментальну, якість записів може варіюватися значно. Багато завантажень є низької якості або неофіційними реміксами;

- реклама та обмеження у безкоштовному плані. Безкоштовну версію SoundCloud супроводжує реклама, яка може бути нав'язливою для користувачів. Також існують обмеження, такі як обмежена кількість пропусків треків. Деякі корисні функції, такі як відтворення в режимі офлайн і прослуховування без реклами, доступні лише з платною підпискою;

- не дозволяє створювати альбоми. Хоча вебсайт корисний для обміну окремими файлами, виконавці не можуть групувати свої файли в колекції чи альбоми. Це означає, що кожен пісню потрібно завантажити, позначити тегом, передати чи повторно опублікувати як окремі файли. Слухачі, які хочуть прослухати набір файлів, не можуть знайти або завантажити їх разом через цей недолік SoundCloud. Ця функція може збентежити багатьох слухачів, і це може відштовхнути їх від взаємодії з більшою кількістю робіт виконавця [40].

1.3.5 Загальне порівняння аналогів

Для порівняння було створено таблицю, в якій будуть основні параметри порівняння платформ для зберігання та відтворення звуку. В таблиці 1.5 відображено порівняння популярних програмних продуктів, таких як Spotify, YouTubeMusic, Deezer та SoundCloud. Було виявлено основні можливості цих застосунків.

Таблиця 1.5 – Порівняння усіх програмних продуктів

Параметр	Spotify	YouTubeMusic	Deezer	SoundCloud
Кількість треків	70 млн	70 млн	73 млн	76 млн
Якість звуку	До 320 кбіт/с	До 256 кбіт/с	До 1411 кбіт/с	До 128 кбіт/с
Підтримка платформ	Windows, macOS, Linux, Android, iOS, Xbox, PlayStation та інші	Windows, macOS, Linux, Android, iOS та інші	Windows, macOS, Linux, Android, iOS та інші	Windows, macOS, Linux, Android та інші
Вартість підписки на преміум версію	Від \$9.99/місяць	Від \$9.99/місяць	Від €9.99/місяць	Безкоштовно або від \$4.99/місяць
Можливості безкоштовної версії	Обмежена функціональність з рекламою та обмеженою якістю звуку	Обмежена функціональність з рекламою та обмеженою якістю звуку	Обмежена функціональність з рекламою та обмеженою якістю звуку	Обмежена функціональність з рекламою
Географічна доступність	Доступно в багатьох країнах	Доступно в багатьох країнах	Доступно в багатьох країнах	Доступно в багатьох країнах
Підтримка відео	Немає	Так	Так	Так (переважно від користувачів)
Офлайн-режим	Так	Так	Так	Так

Всі чотири платформи дозволяють користувачам слухати музику онлайн, без необхідності завантажувати її на пристрій.

Кожна з цих платформ використовує алгоритми для рекомендацій музики на основі історії прослуховування користувача, уподобань та поведінки.

Користувачі можуть створювати власні плейлисти, групувати пісні за темами та ділитися ними з іншими користувачами.

Користувачі можуть підписуватися на улюблених артистів, щоб слідкувати за їхньою новою музикою та отримувати оновлення про них.

Deezer і SoundCloud пропонують радіостанції та подкасти на різні теми.

Кожна з платформ має великий вибір треків і альбомів з різних жанрів.

З платною підпискою, користувачі можуть завантажувати музику для прослуховування в офлайн, коли немає доступу до Інтернету.

SoundCloud надає можливість взаємодії з музикантами та іншими користувачами, залишати коментарі та взаємодіяти через обговорення [41].

2 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ЙОГО ОБҐРУНТУВАННЯ

2.1 Аналіз поширених фреймворків для фронтенду

У сучасній веброзробці використання фреймворків для створення клієнтської частини вебзастосунків стало стандартною практикою. Вибір фреймворку для розробки клієнтської частини вебзастосунку є важливим кроком, який впливає на структуру, продуктивність, масштабованість та підтримку проєкту. Фреймворки дозволяють значно спростити розробку, тестування та підтримку застосунків. У цьому розділі буде розглянуто три найпопулярніші фреймворки: Angular, React та Vue, які мають свої унікальні особливості та застосування [42].

2.1.1 Angular

Angular є одним з найвідоміших фреймворків, створених компанією Google.

Його основними перевагами є:

- масштабованість та модульність – Angular базується на модульній архітектурі, що дозволяє легко розширювати та підтримувати великі проєкти;
- розширені можливості – Angular включає в себе багато внутрішніх функцій, таких як двостороннє зв'язування даних, маршрутизація, управління формами, роутінг та http клієнт;
- підтримка TypeScript – Angular використовує TypeScript, що забезпечує статичний аналіз коду та допомагає уникнути багатьох помилок ще на етапі розробки;
- активне співтовариство та підтримка – як проєкт Google, Angular має велику базу користувачів, розробників та багатий набір документації [43].

2.1.2 React

React, розроблений компанією Facebook, є бібліотекою для створення користувацьких інтерфейсів, яку часто використовують у парі з іншими бібліотеками та фреймворками.

До переваг можна віднести:

- гнучкість – React забезпечує високу гнучкість завдяки своїй компонентній архітектурі та можливості інтеграції з іншими інструментами;
- віртуальний DOM – реалізація віртуального DOM дозволяє досягти високої продуктивності при рендерингу;
- велике співтовариство – React має широку базу користувачів та розробників, що забезпечує велику кількість сторонніх бібліотек та інструментів;
- JSX – використання JSX для опису користувацьких інтерфейсів дозволяє поєднувати HTML та JavaScript, спрощуючи розробку [44].

2.1.3 Vue

Vue – це прогресивний фреймворк, створений для спрощення розробки користувацьких інтерфейсів.

До переваг відносяться:

- простота у використанні – Vue має низький поріг входження завдяки зрозумілій документації та простому синтаксису;
- гнучкість – як і React, Vue забезпечує високу гнучкість та можливість інтеграції з іншими інструментами;
- продуктивність – Vue використовує віртуальний DOM для оптимізації продуктивності;

– активне співтовариство – хоч Vue і молодший за Angular та React, його співтовариство швидко зростає, забезпечуючи підтримку та багатий набір інструментів [45].

2.1.4 Порівняння фреймворків

В таблиці 2.1 наведено порівняння популярних фреймворків для клієнтської частини проєкту.

Таблиця 2.1 – Порівняння фреймворків для фронтенду

Критерій	Angular	React	Vue
1	2	3	4
Тип	Повний фреймворк	Бібліотека для інтерфейсів	Прогресивний фреймворк
Мова	TypeScript	JavaScript	JavaScript
Основні концепції	Декларативне програмування, компонентна архітектура, об'єктно-орієнтоване програмування, реактивність	Компонентний підхід, декларативне програмування, віртуальний DOM	Декларативне програмування, компонентний підхід, реактивність
Повноцінність	Повний набір інструментів для розробки (роутінг, сервісів, форм, HTTP клієнтів)	Потребує додаткових бібліотек для роутінгу, керування станом тощо	Може потребувати додаткових бібліотек для повноцінної роботи
Типізація	Використання TypeScript, статичний аналіз коду	Використання JavaScript або TypeScript	Використання JavaScript або TypeScript
Гнучкість	Висока, але орієнтована на TypeScript	Дуже гнучка, інтеграція з будь-якими іншими бібліотеками	Гнучка, легко інтегрується у існуючі проєкти

Продовження таблиці 2.1

1	2	3	4
Крива навчання	Висока, особливо для новачків у TypeScript та об'єктно-орієнтованому програмуванні	Середня, необхідно звикнути до JSX. Але через велику свободу новачки пишуть дуже погану архітектуру.	Низька, гарна документація і простота використання
Продуктивність	Може впливати на продуктивність при першому завантаженні, але при правильному використанню може бути більш швидкою за інші бібліотеки.	Висока завдяки віртуальному DOM	Легка вага
Підтримка	Постійні оновлення та підтримка від Google	Постійні оновлення та підтримка від Facebook	Підтримка від спільноти та розробника
Документація	Вичерпна та структурована	Гарна, але може бути складною через велику кількість бібліотек	Простота для розуміння та багато прикладів
Популярність	Висока	Дуже висока	Висока, але менша ніж у React і Angular

Обрані характеристики для порівняння дозволяють отримати всебічне розуміння кожного фреймворку, його сильних і слабких сторін, що є важливим для прийняття обґрунтованого рішення при виборі інструментів для розробки клієнтської частини вебзастосунків [46].

2.1.1 Обґрунтування вибору фреймворка для фронтенду

Для розробки клієнтської частини проєкта було обрано Angular за такими критеріями:

- швидкість розробки – Angular забезпечує швидкий старт проєктів завдяки зменшенню кількості налаштувань, пов'язаних з модулями;
- підтримка від Google – постійні оновлення та підтримка від компанії Google гарантують стабільний розвиток фреймворку;
- типізація – використання TypeScript забезпечує високий рівень надійності коду завдяки статичній типізації;
- модульна архітектура – незважаючи на можливість уникнення модулів, Angular залишається модульним фреймворком, що дозволяє організувати код і робить його більш читабельним і зрозумілим.

Також цей вибір був зумовлений тим, що найкращий фреймворк для розробки – той, на якому програміст пише якісний та оптимізований код, та має досвід розробки.

2.2 Аналіз поширених фреймворків для бекенду

У процесі розробки серверної частини вебзастосунків важливо обрати фреймворк, що відповідає вимогам проєкту за продуктивністю, масштабованістю та зручністю використання. Розглянемо два найпоширеніші фреймворки для цієї мети – Nest.js та Express.js.

2.2.1 Express.js

Express.js є одним із найпопулярніших і найширше використовуваних фреймворків для Node.js.

Основні його характеристики та переваги:

- простота та мінімалізм – Express.js надає мінімальний набір інструментів, залишаючи розробнику свободу вибору додаткових компонентів;
- гнучкість – завдяки своїй простоті, Express.js можна використовувати у поєднанні з будь-якими іншими бібліотеками та фреймворками;
- велике співтовариство – Express.js має велику базу користувачів, що забезпечує багатий набір сторонніх модулів та розширень;
- продуктивність – завдяки своїй легковажності, Express.js забезпечує високу продуктивність навіть під великим навантаженням [47].

2.2.2 Nest.js

Nest.js є прогресивним фреймворком для Node.js, побудованим на базі TypeScript та натхненим архітектурою Angular.

Основні його характеристики та переваги:

- модульна архітектура – Nest.js підтримує модульну архітектуру, що сприяє зручному розподілу коду та його масштабованості;
- використання TypeScript – використання TypeScript забезпечує статичний аналіз коду та допомагає уникнути багатьох помилок ще на етапі розробки;
- декораторний підхід – Nest.js підтримує використання декораторів для визначення компонентів, що спрощує структуру коду та його читабельність;
- підтримка та документація – Nest.js має активне співтовариство та детальну документацію, що сприяє швидкому освоєнню фреймворку [48].

2.2.3 Порівняння фреймворків

В таблиці 2.2 наведено порівняння популярних фреймворків для серверної частини проєкту.

Таблиця 2.2 – Порівняльна таблиця фреймворків

Характеристика	Express.js	Nest.js
Модульна архітектура	Відсутня	Присутня
Мова програмування	JavaScript	TypeScript
Розширені можливості	Мінімальні, потребують додаткових модулів	Вбудовані, включаючи підтримку WebSockets, мікросервісів та GraphQL
Простота використання	Легка для новачків	Вимагає знань TypeScript та Angular
Співтовариство та підтримка	Велике, багатий набір сторонніх модулів	Активне, детальна документація
Продуктивність	Висока	Висока

Порівняння фреймворків на основі зазначених характеристик дозволяє охопити ключові аспекти, які впливають на вибір фреймворку і забезпечують повноцінне розуміння їх можливостей та обмежень [49].

Модульна архітектура – дозволяє структурувати код у зручний для розробки і підтримки спосіб. Це важливо для великих проєктів, де потрібна чітка організація та розподіл функціональності.

Мова програмування – може впливати на швидкість розробки, зручність кодування та інтеграцію з іншими системами.

Розширені можливості – наявність вбудованих функцій значно спрощує реалізацію складних проєктів і знижує необхідність у використанні сторонніх бібліотек.

Простота використання – легкість освоєння фреймворку є важливою для швидкого старту проєкту.

Співтовариство та підтримка – велике і активне співтовариство означає наявність багатого набору сторонніх модулів, розширень та документації, що значно полегшує розробку та вирішення проблем.

Продуктивність – визначає, як швидко фреймворк може обробляти запити і наскільки ефективно використовує ресурси сервера, що особливо важливо для високонавантажених застосунків.

2.2.4 Обґрунтування вибору фреймворка для бекенду

Для розробки серверної частини проєкта було обрано Nest.js за такими критеріями:

- сумісність з Angular – Nest.js та Angular мають подібну архітектуру та використовують TypeScript, що полегшує взаємодію між клієнтською та серверною частинами застосунку;
- масштабованість та модульність – завдяки модульній архітектурі Nest.js можна легко розширювати та підтримувати великі проєкти;
- розширені можливості – Nest.js пропонує широкий спектр вбудованих функцій, таких як підтримка WebSockets, мікросервісів та GraphQL, що дозволяє зосередитися на логіці проєкту, а не на налаштуваннях;
- структура та організація коду – використання декораторів та модульної системи спрощує розподіл коду на логічні частини та сприяє його читабельності;
- підтримка TypeScript – Nest.js забезпечує всі переваги TypeScript, включаючи статичний аналіз коду, що підвищує надійність та стабільність проєкту, запобігає багатьом помилкам на етапі розробки.

2.3 Аналіз поширених баз даних

Вибір системи управління базами даних (СУБД) є одним з критичних рішень при розробці програмного забезпечення, особливо для серверної частини застосунку. Від цього вибору залежить продуктивність, масштабованість та загальна архітектура системи. У цьому розділі буде розглянуто три найбільш популярні СУБД: MongoDB, SQLite та MySQL [50].

2.3.1 MongoDB

MongoDB – це документоорієнтована нереляційна база даних (NoSQL) база даних, яка зберігає дані у форматі BSON (Binary JSON). Ця СУБД відрізняється високою масштабованістю, гнучкою схемою, хорошою підтримкою великих обсягів даних та простотою використання. Це дозволяє зберігати різноманітні типи даних, включаючи вкладені документи та масиви, без необхідності визначення строгої схеми. MongoDB ідеально підходить для зберігання великих обсягів неструктурованих даних, таких як логі, аналітичні дані та дані соціальних мереж [51].

До переваг належать:

- гнучкість – легко адаптується до змінних вимог завдяки відсутності жорсткої схеми;
- масштабованість – легко горизонтально масштабується для обробки великих обсягів даних;
- висока продуктивність – забезпечує високу швидкість запитів, особливо для операцій читання;
- агрегація – потужні можливості для агрегації даних, що дозволяє виконувати складні аналітичні запити;
- використання – великі вебзастосунки, аналітика даних, IoT, мобільні застосунки.

2.3.2 SQLite

SQLite – це легка, вбудована СУБД, яка зберігає дані у вигляді таблиць. Вона часто використовується в невеликих проєктах, мобільних застосунках та як вбудована база даних в інших програмних продуктах. SQLite проста у використанні, але має обмежені можливості масштабування та транзакцій [52].

До переваг належать:

- простота – легко вбудовується в різні застосунки, не вимагає окремого сервера;
- компактність – займає мало місця на диску;
- швидкість – забезпечує високу швидкість роботи для невеликих баз даних;
- переносність – бази даних SQLite можуть бути легко переміщені між різними платформами;
- використання – мобільні застосунки, вбудовані системи, невеликі вебзастосунки, де не потрібна висока масштабованість.

2.3.3 MySQL

MySQL – це реляційна СУБД з відкритим кодом, яка використовує мову SQL для взаємодії з даними. Вона широко застосовується для створення вебзастосунків, систем управління контентом та інших корпоративних систем. MySQL відрізняється високою продуктивністю, надійністю та підтримкою складних запитів [53].

До переваг належать:

- стабільність – одна з найстаріших і найбільш вивчених СУБД, що забезпечує високу стабільність;

- продуктивність – забезпечує високу продуктивність для широкого спектру застосунків;
- масштабованість – може бути масштабована як вертикально, так і горизонтально;
- багато можливостей – підтримує широкий спектр функцій, включаючи транзакції, тригери, зберігаємі процедури;
- використання – великі вебзастосунки, електронна комерція, системи управління базами даних для підприємств.

2.3.4 Порівняння баз даних

Для вибору порівняно такі бази даних MongoDB, SQLite, MySQL. Порівняння виконано у таблиці 2.3.

Таблиця 2.3 – Порівняння баз даних

Параметри	MongoDB	SQLite	MySQL
Модель даних	Документоорієнтована	Реляційна	Реляційна
Масштабованість	Висока	Обмежена	Висока
Гнучкість схеми	Висока	Низька	Середня
Продуктивність	Висока	Середня	Висока
Транзакції	Підтримуються	Обмежена підтримка	Підтримуються
Використання	Великі вебзастосунки, аналітика даних	Мобільні застосунки, невеликі проекти	Великі вебзастосунки, електронна комерція

Наведені характеристики є найбільш загальними і важливими для порівняння різних СУБД [54].

Модель даних – ця характеристика визначає, як дані організовані в базі даних. Реляційні бази (MySQL) використовують таблиці зі строго

визначеними стовпцями, а документоорієнтовані (MongoDB) зберігають дані у форматі JSON-подібних документів, що надає більшу гнучкість.

Масштабованість – ця характеристика показує, як база даних може збільшувати свої обчислювальні потужності для обробки більших обсягів даних. MongoDB добре масштабується горизонтально (додавання нових серверів), тоді як SQLite має обмежені можливості в цьому плані.

Гнучкість схеми – реляційні бази мають строго визначену схему, тоді як документоорієнтовані дозволяють змінювати структуру даних динамічно.

Продуктивність – цей показник визначає, наскільки швидко база даних може виконувати запити на читання та запис даних.

Транзакції – характеристика визначає, наскільки база даних гарантує атомність, узгодженість, ізоляцію та довговічність транзакцій.

2.3.5 Обґрунтування вибору MongoDB

Було обрано MongoDB з наступних причин:

- застосунок потребує зберігання різноманітних даних, які можуть змінюватися з часом. Гнучка схема MongoDB дозволяє легко адаптуватися до цих змін;
- очікується, що обсяг даних буде зростати, тому потрібна база даних, яка легко масштабується;
- для застосунку важлива швидкість доступу до даних, особливо при великих обсягах. MongoDB демонструє відмінні результати в цьому аспекті;
- MongoDB пропонує зручний API для роботи з даними, що спрощує розробку застосунку;
- частина наших даних має неструктурований характер (наприклад, текстові описи, файли), і MongoDB ефективно зберігає такі дані.

3 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Опис функціоналу програми

На рисунку 3.1 наведено діаграму прецедентів, де визначено функціонал застосунка відтворення музики «Apollo».

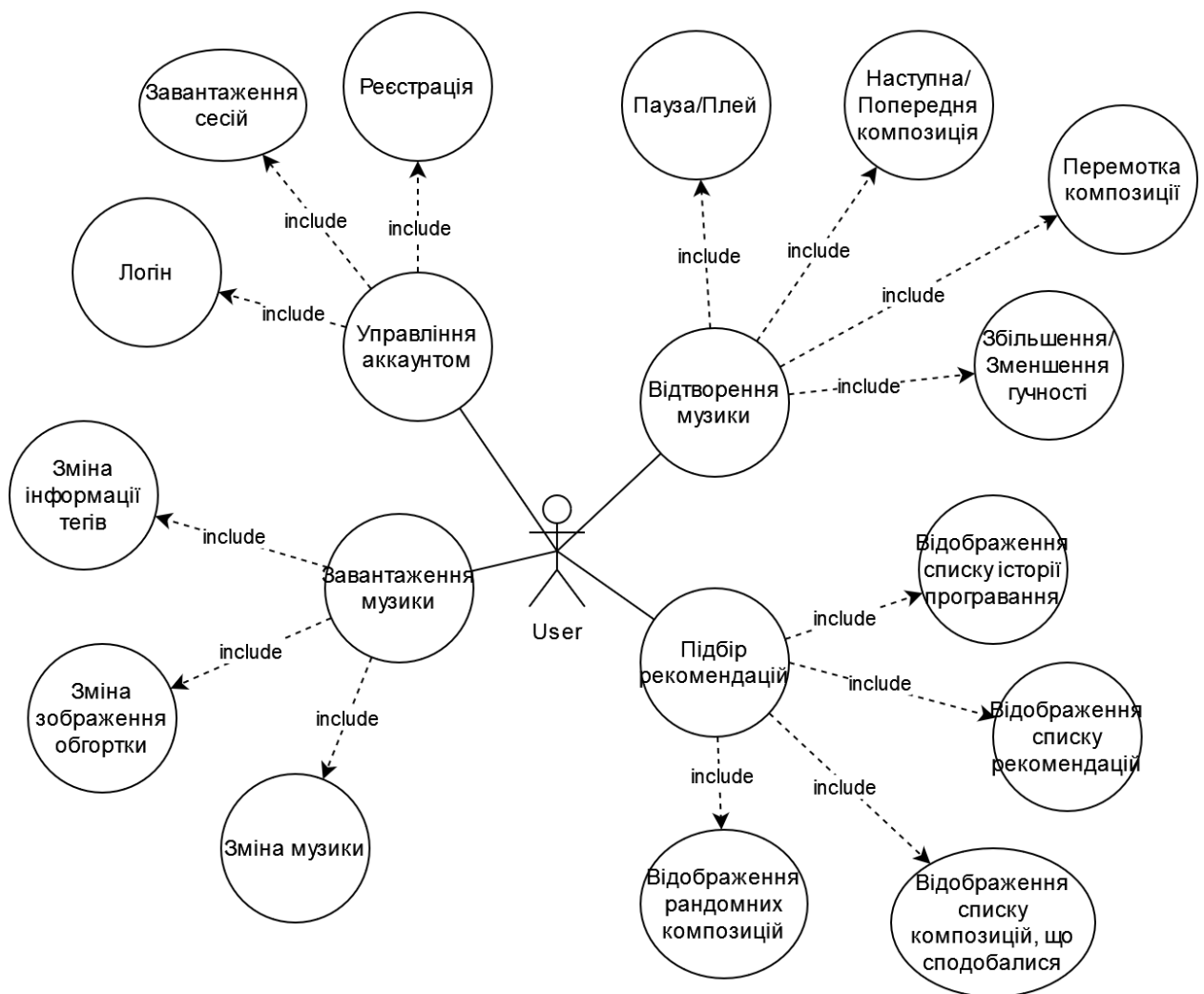


Рисунок 3.1 – Use Case діаграма основних можливостей застосунку

Далі буде наведено етапи основного потоку подій для користувача. Основний виконавець (актор) – користувач.

Потоки подій:

а) робота з акаунтом:

- 1) логін;
 - 2) реєстрація;
 - 3) управління акаунтом;
- б) завантаження сеансів:
- 1) завантаження сесій;
- в) завантаження та управління музикою:
- 1) завантаження музики;
 - 2) зміна музики;
 - 3) зміна інформації тегів;
 - 4) зміна зображення обкладинки;
- г) відтворення та управління музичними композиціями:
- 1) відтворення музики;
 - 2) пауза/плей;
 - 3) наступна/попередня композиція;
 - 4) перемотка композиції;
 - 5) збільшення/зменшення гучності;
- д) перегляд рекомендацій та історії:
- 1) підбір рекомендацій;
 - 2) відображення списку історії програвання;
 - 3) відображення списку рекомендацій;
 - 4) відображення випадкових композицій;
 - 5) відображення списку композицій, що сподобались.

Ці етапи складають основні функціональні можливості для користувача під час взаємодії з застосунком, забезпечуючи повноцінний досвід роботи з музичною бібліотекою та управління акаунтом.

У наступних розділах буде детально описано основний реалізований функціонал програми.

3.1.1 Взаємодія із функціоналом відтворення музики

AudioService у Angular застосунку надає функціональність аудіоплеєра, призначеного для обробки аудіофайлів, управління їх відтворенням та інтеграції із зовнішніми сервісами для збереження даних користувача. Основний акцент у сервісі зроблено на збереженні стану та керуванні подіями відтворення у реальному часі. Для відстеження поточного треку, статусу відтворення та списку доступних треків використовуються реактивні потоки, такі як BehaviorSubject і Observable. Це дозволяє динамічно оновлювати дані плеєра і відображати їх актуальний стан у інтерфейсі користувача, підтримуючи тісний зв'язок з діями користувача.

Клас також реалізує керування подіями аудіооб'єкта, відстежуючи такі моменти, як старт відтворення, пауза, завершення треку та помилки. Ці події обробляються в реальному часі і дозволяють аудіоплеєр реагувати на зміни в поточному статусі аудіофайлу. Наприклад, при завершенні поточного треку аудіоплеєр може автоматично переключитися на наступний у списку, що створює безперервне відтворення. Всі події прив'язані до відповідних потоків, що дозволяє інтерфейсу плавно синхронізуватися зі станом програвача.

Для управління відтворенням AudioService надає методи, які надають користувачеві контроль над аудіоплеєром. Наприклад, можна перемикати трек на паузу та назад на відтворення, змінювати поточну позицію всередині треку, регулювати гучність та перемикати треки у плейлисті. Ці функції роблять аудіоплеєр гнучким та чуйним до дій користувача, дозволяючи йому налаштовувати процес відтворення під свої потреби. Налаштування, такі як гучність, зберігаються в localStorage, що робить плеєр адаптованим до переваг користувача і зберігає досвід користувача між сеансами.

Окрім керування відтворенням, AudioService також інтегрується із зовнішніми API для запису історії прослуховувань та керування обраними

треками. При зміні треку інформація про прослуханий трек надсилається на сервер для додавання в історію, а також перевіряється статус "обраного". Це дозволяє зберігати дані про уподобання користувача, забезпечуючи персоналізований досвід та можливість додавати треки у вибране або видаляти їх звідти.

Сервіси реактивного програмування дозволяють постійно оновлювати стан плеєра в інтерфейсі користувача, роблячи можливою тісну інтеграцію. Наприклад, коли користувач перемикає трек або змінює гучність, інтерфейс негайно відображає зміни завдяки потокам, що відстежують параметри програвача, такі як тривалість, час, гучність і статус паузи. Цей підхід робить взаємодію з плеєром інтуїтивним, плавним і чуйним, забезпечуючи високу якість користувацького досвіду в умовах динамічних змін.

3.1.2 Взаємодія з функціоналом авторизації

AccountService надає комплексний функціонал для керування обліковим записом користувача та сесіями в застосунку, включаючи реєстрацію, авторизацію, вихід з облікового запису та оновлення стану сесії. Разом з SessionTokenInterceptor та AuthComponent, цей сервіс створює цілісну систему управління аутентифікацією та авторизацією користувача, а також організовує зберігання даних акаунту та токена сесії в localStorage.

AccountService починає роботу з завантаження з localStorage інформації про поточний обліковий запис і сесію. Реактивні потоки account\$ та session\$ зберігають дані користувача та активної сесії відповідно. Ці дані автоматично зберігаються в localStorage, коли оновлюються, що робить їх доступними між сеансами користувача. Важливим аспектом є перевірка терміну дії сесії. Якщо сесія застаріла, сервіс може ініціювати потрібні дії, наприклад, вийти з акаунта або запросити оновлення токена.

Сервіс надає основні методи для роботи з акаунтом: `signup`, `login` та `logout`. Метод `signup` використовується для реєстрації нового користувача. При реєстрації дані облікового запису відправляються на сервер, і в разі успішної відповіді сесія та дані облікового запису оновлюються і зберігаються. Метод `login` приймає облікові дані, відправляє їх на сервер, а у разі успішного входу оновлює токен сесії та завантажує дані облікового запису. `logout` очищає локальні дані облікового запису та сесії, а також надсилає запит на сервер для завершення сесії.

Для автоматичного додавання токена до заголовків запитів використовується `SessionTokenInterceptor`. Цей перехоплювач перевіряє наявність активного токена сесії та додає його в заголовок `session-token` кожного HTTP-запиту. Така реалізація дозволяє всім запитам, які потребують аутентифікації, коректно передавати токен, спрощуючи структуру програми та забезпечуючи прозорість роботи для користувача.

`AuthComponent` реалізує інтерфейс користувача для аутентифікації. Компонент надає форму, яка може бути використана для входу або реєстрації, залежно від того, чи зареєстрований користувач. У ньому передбачено логіку для активації та деактивації полів форми (наприклад, імені та опису) залежно від режиму роботи – реєстрації або авторизації. При надсиланні форми відбувається перевірка її валідності; якщо вона коректна, викликається або метод `login`, або `signup` залежно від статусу. У разі помилки валідації, всі поля форми позначаються як вимагають заповнення.

Таким чином, `AccountService`, `SessionTokenInterceptor` та `AuthComponent` працюють у зв'язці, організуючи комплексну структуру для управління аутентифікацією та авторизацією в застосунку.

3.1.3 Взаємодія із завантаженням аудіо та обкладинки музики

UploadMusicComponent реалізує повний цикл завантаження аудіофайлу та обкладинки на сервер, забезпечуючи попередню обробку метаданих та зображень, а також подальше збереження даних для інтеграції до бібліотеки користувача. Основні функції компонента включають завантаження аудіо та зображення, парсинг метаданих, відправлення даних на сервер, а також валідацію полів та їх оновлення залежно від вибраного контексту.

Процес починається із завантаження аудіофайлу через метод loadAudio, який використовує об'єкт FileReader для асинхронного читання файлу. У момент, коли файл завантажений, він конвертується в ArrayBuffer, який відправляється в потік audio\$. Потім аудіофайл обробляється через parseBlob, щоб витягти метадані, такі як назва, артист, альбом, жанр і рік випуску. Ці дані записуються у форму, щоб користувачеві не потрібно було вводити їх вручну. Компонент також намагається витягти обкладинку з метаданих аудіофайлу, використовуючи функцію selectCover. Якщо обкладинка знайдена, вона відображається в компоненті, перетворюючись на формат Blob і зв'язуючись з елементом `<input type="file">` за допомогою спеціального контейнера DataTransfer. Це дозволяє використовувати як локальні зображення, так і вбудовані в аудіофайл обкладинки.

Для завантаження обкладинки компонента окремо реалізований метод loadImg, який дозволяє завантажувати зображення вручну, якщо його немає в метаданих аудіофайлів. Метод також використовує FileReader, щоб прочитати файл зображення у двох форматах – ArrayBuffer для зберігання даних та DataURL для відображення зображення в інтерфейсі та вилучення типу зображення, яке буде використовуватися надалі при відправці на сервер.

Метод send виконує валідацію форми і якщо форма заповнена коректно, починає процес завантаження файлів на сервер. Створюються

FormData об'єкти для аудіофайлу та обкладинки, до яких додаються файли та відповідні метадані. Потім `http.post` відправляє аудіофайл на сервер, при цьому в заголовках запиту передаються дані форми, такі як назва, виконавець, альбом, жанр та рік. На стороні сервера метод `uploadMusic` контролера, використовуючи `AuthGuard`, перевіряє доступ до ресурсу, а за допомогою `FileInterceptor` – валідує тип завантажуваного файлу, приймаючи тільки аудіо. Після успішного завантаження метод `createMusic` сервісу створює запис про музику, а сервер повертає підтвердження у відповіді.

Коли аудіофайл успішно завантажений, починається завантаження обкладинки за допомогою `switchMap`, що дозволяє спочатку дочекатися відповіді від сервера, що містить ідентифікатор завантаженого аудіофайлу, необхідний для прив'язування обкладинки до треку. `upload/cover` метод на стороні сервера також захищений `AuthGuard` та `FileInterceptor`, приймаючи тільки зображення. При успішному завантаженні обкладинки метод `addCover` прив'язує її до аудіозапису, і сервер повертає підтвердження про виконання операції.

Ця комплексна структура з методів завантаження та компонентів, таких як `FileInterceptor`, `AuthGuard` та `SessionTokenInterceptor`, забезпечує надійну обробку аудіо- та графічних даних. Інтерактивність досягається за допомогою реактивних потоків, які забезпечують динамічне оновлення даних, відображення зображень та оновлення інформації про музику на основі метаданих.

3.1.4 Взаємодія з History Favorite

Сервіси та контролери `HistoryService` та `FavoriteService` реалізують функції ведення історії прослуховування та управління обраними треками для користувача. Ці сервіси працюють у зв'язці з контролерами для обробки HTTP-запитів та підтримки актуального стану бібліотеки користувача,

додаючи або видаляючи записи, запитуючи історію та перевіряючи статус обраного. Обидва сервіси використовують захист AuthGuard, який перевіряє права доступу, а також SessionService для ідентифікації користувача через sessionToken.

HistoryService відповідає за додавання треку до історії прослуховування та запит списку раніше прослуханих треків. Метод addRecord приймає sessionToken та musicId та виконує пошук облікового запису користувача через sessionService, щоб прив'язати трек до конкретного користувача. Потім створює новий об'єкт історії з датою прослуховування і зберігає їх у базі даних. Метод getListRecords повертає останні записи історії прослуховування для поточного користувача. Він обмежує кількість записів, що повертаються (за замовчуванням до 50) і сортує їх у порядку від останніх до перших. Для кожного треку з історії historyModel викликає musicModel, який знаходить та повертає повний об'єкт музики, включаючи метадані, щоб користувачеві відображалися всі деталі запису.

FavoriteService керує додаванням, видаленням, перевіркою та отриманням обраних треків користувача. Метод createFavorite перевіряє, чи існує вже запис у вибраному для даного треку та користувача, і, якщо ні, додає трек у вибране, записуючи ID треку, ID користувача та дату додавання. Метод deleteFavorite видаляє трек із вибраного, а isFavorite перевіряє, чи міститься цей трек у вибраному. Метод getFavorites повертає список всіх обраних треків, запитуючи дані через musicModel для отримання повної інформації про кожного з них.

Контролер FavoriteController підтримує функціональність роботи з обраними треками, використовуючи методи FavoriteService. Запити POST і DELETE додають та видаляють треки з обраного відповідно, де в заголовку передається session-token, що ідентифікує користувача. Метод getFavorite повертає список обраних треків поточного користувача, а метод isFavorite перевіряє статус обраного для зазначеного musicId. Контролер

HistoryController взаємодіє з HistoryService, дозволяючи додавати треки в історію через addRecord, а також отримувати останні прослухані треки через getList. Обидва контролери формують відповіді у форматі JSON та обробляють можливі помилки, надсилаючи відповідні коди статусів у разі невдачі.

3.1.5 Завантаження з бекенда

Функціонал бекенда завантаження файлів забезпечує доступ до аудіофайлів та зображень обкладинок, роблячи їх доступними для потокової передачі або скачування. Метод, що відповідає за аудіофайли, приймає ідентифікатор музичного файлу та, використовуючи сервіс даних, отримує його інформацію. На основі цих даних створюється потік для передачі вмісту файлу, а також додаються необхідні метадані для вказівки імені та типу контенту, що передається. Це дозволяє клієнту коректно сприймати файл, чи це відтворення чи завантаження.

Аналогічний процес застосовується для методу, який повертає файл обкладинки. Спочатку отримується інформація про зображення, потім ініціюється потік, який супроводжується заголовками для коректного відображення формату та імені. Обидва методи забезпечують контрольовану та безпечну передачу мультимедійних даних. У разі помилок передається відповідна відповідь з кодом статусу, який повідомляє про проблеми, що виникли.

3.1.6 Пропозиції списків музики

Методи бекенда також надають функціонал для пошуку музики та отримання списку нещодавно прослуханих треків, зберігаючи спрощений та зручний доступ до інформації про прослуховування.

Метод `getListRecords` орієнтований на надання списку прослуханих композицій, враховуючи історію користувача. Використовуючи переданий токен сесії, метод ідентифікує користувача, отримує записи історії прослуховувань, відсортовані за датою від останнього до першого, і обмежує кількість записів (за замовчуванням – 50). Потім він збирає повну інформацію про кожен трек з історії, формуючи та повертаючи масив об'єктів музики, що спрощує доступ до недавніх прослуховувань.

Також є алгоритм, який виконує функцію пошуку схожих користувачів на основі їх музичних уподобань. Програма починається з отримання інформації про обліковий запис користувача сеансового токена. Потім система отримує останні 50 унікальних ідентифікаторів музики з історії користувача. Після цього алгоритм перевіряє, які з цих пісень знаходяться у вибраному користувачеві, щоб створити набір ідентифікаторів вибраних композицій.

Далі, програма знаходить користувачів зі схожими музичними уподобаннями, порівнюючи їх вибрані треки з вибраними треками поточного користувача. Алгоритм використовує безліч для зберігання ідентифікаторів користувачів зі схожими смаками, збільшуючи при цьому лічильник збігів для кожного користувача.

Після збору всіх даних система сортує користувачів за кількістю збігів і створює список рекомендованих музичних треків. Програма уникає повторних рекомендацій треків, які вже були в історії користувача або в обраному, і обмежує список рекомендацій 50 треками. Зрештою, алгоритм повертає список музичних рекомендацій, ґрунтуючись на уподобаннях користувачів зі схожими смаками.

3.2 Навігація між сторінками застосунку

Для відображення процесу авторизації користувача у вебзастосунку та доступу до його функціональних можливостей застосовано UML діаграму діяльності, яка демонструє логічний потік операцій від початкового стану до завершення авторизації та подальшого доступу до певних розділів застосунку, таких як списки музики, пошук та завантаження музики, на рисунку 3.2.

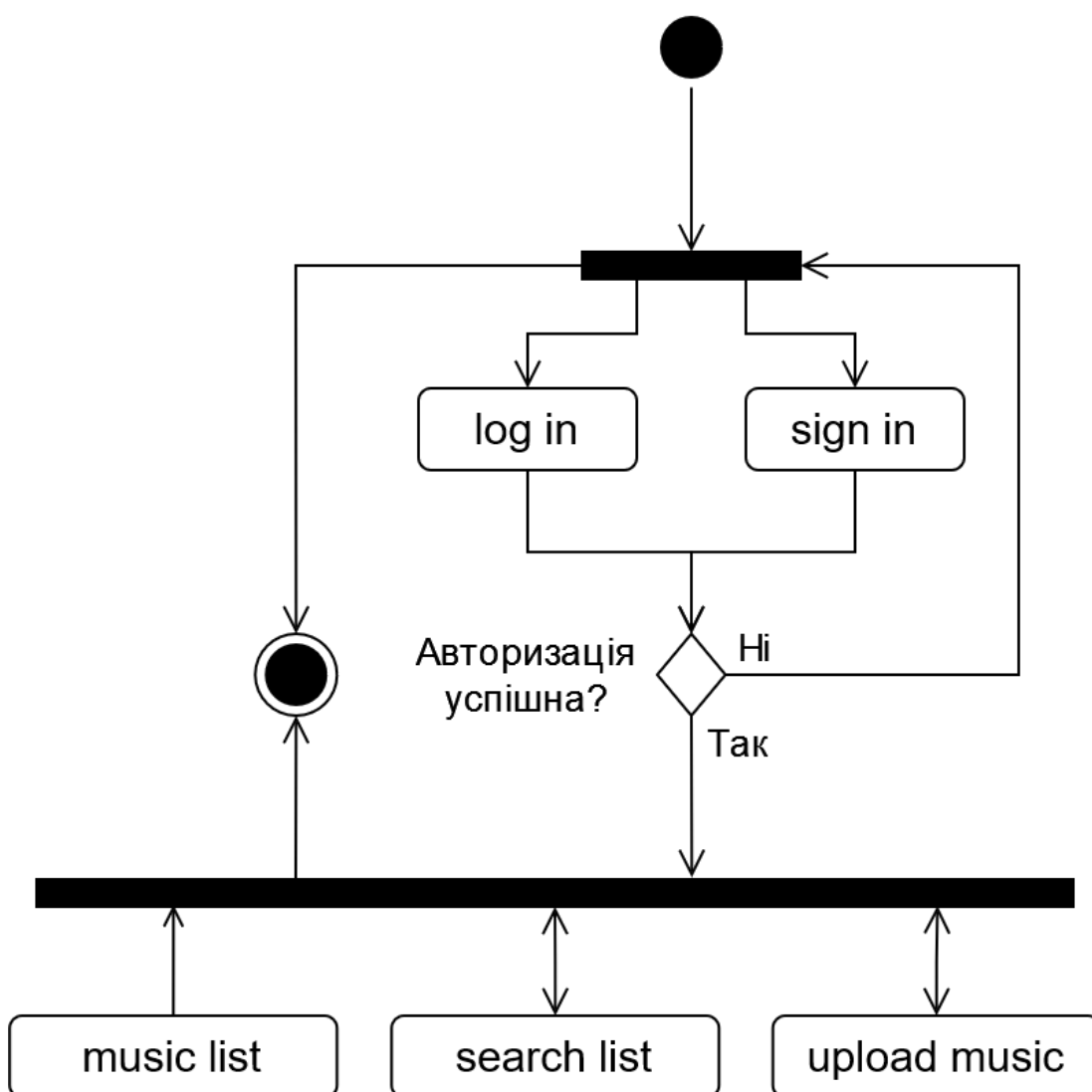


Рисунок 3.2 – Діаграма навігації

Початковий вузол позначає старт процесу, після якого користувач має вибір між двома діями: виконати вхід у систему (log in) або зареєструватися

(sign in). Обидва процеси ведуть до етапу перевірки успішності авторизації. Цей етап представлений умовним вузлом, де приймається рішення: якщо авторизація успішна, користувач переходить до наступних етапів; у разі невдачі процес повертається до початку.

У разі успішної авторизації користувач отримує доступ до основних функцій застосунку, представлених у вигляді трьох окремих блоків:

- music list – відображення списку музичних треків;
- search list – можливість пошуку та перегляду відповідних результатів;
- upload music – функція завантаження музичних файлів у систему.

Потік управління в діаграмі забезпечує чітку структуру переходу між станами та демонструє, як користувач може взаємодіяти з системою залежно від результату авторизації. Цей підхід забезпечує високу інформативність процесу та дозволяє деталізувати послідовність виконання операцій, важливих для підтримки доступу та взаємодії користувача з застосунком.

3.3 База даних

У розробленому застосунку структура бази даних розроблена для керування обліковими записами користувачів, взаємодією з музичним контентом та збереженням історії дій. У цій системі реалізовано декілька ключових схем, кожна з яких відображає певні аспекти взаємодії користувача з музичними об'єктами. Підхід, що використовує Mongoose та MongoDB, надає гнучкість для обробки даних, включаючи повне керування користувачами та їх перевагами, реєстрацію активності та зберігання музичних файлів. На рисунку 3.3 зображено діаграму бази даних.

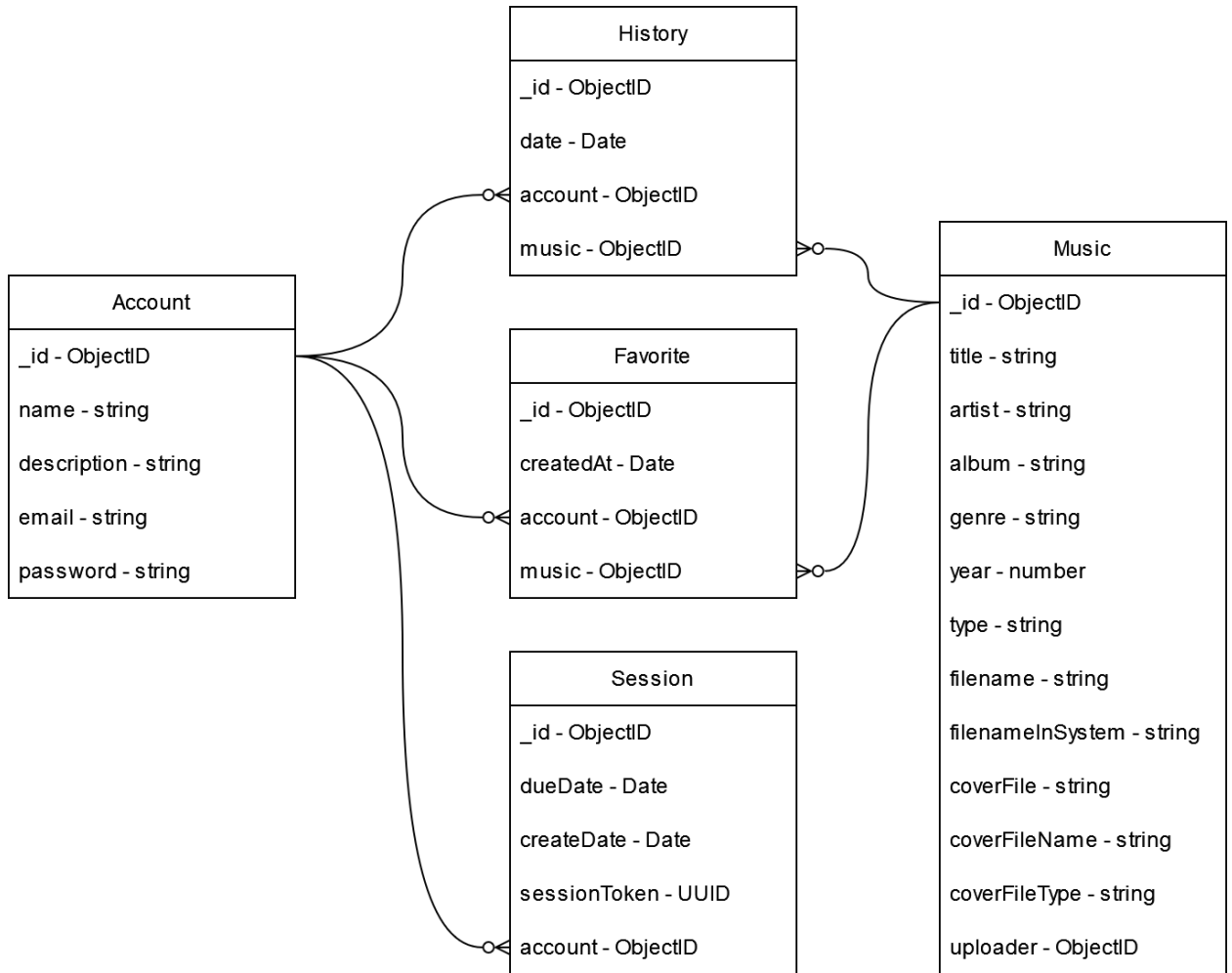


Рисунок 3.3 – Діаграма бази даних

Ця структура бази даних організована з урахуванням модульності та високої зв'язності даних, підтримуючи посилання між основними сутностями через ObjectID. Підхід з використанням індексації та посилань забезпечує оптимізацію запитів, високу продуктивність при великих обсягах даних, а також гнучкість, дозволяючи розширювати схему з мінімальними змінами структури бази даних.

3.3.1 Схема Account: Управління даними користувачів

Схема Account призначена для зберігання основних даних про користувачів:

- поля name та description зберігають ідентифікуючу інформацію про користувача;
- поля email та password унікальні та потрібні для забезпечення безпеки облікових записів;
- поле email є унікальним ідентифікатором облікового запису, забезпечуючи його однозначну ідентифікацію.

Структура підтримує унікальні обмеження на рівні бази даних для унеможливлення дублювання облікових записів, а також забезпечує строгість даних, вимагаючи обов'язкового заповнення всіх полів.

3.3.2 Схеми Favorite: Система вибраних треків

Схеми Favorite відстежує музику, додану користувачами до списку вибраного:

- поле createdAt фіксує дату та час додавання запису до вибраного;
- поле account пов'язує запис з конкретним користувачем через посилання на схему Account, а поле music – з конкретним треком, посилаючись на схему Music.

Така структура забезпечує доступ до вибраних треків користувачів, а також надає базу даних для аналізу музичних уподобань, що часто використовується для персоналізації та рекомендацій.

3.3.3 Схеми History: Історія прослуховувань

Схеми History організована для зберігання історії прослуховувань користувача:

- поле date вказує дату та час прослуховування;
- поля music та account забезпечують зв'язок з прослуханим треком та обліковим записом користувача відповідно.

Дана схема також зберігає дані про дії користувача, які можуть бути використані для покращення системи рекомендацій та створення звітів про прослуховування.

3.3.4 Схема Music: Структура музичного контенту

Схема Music відповідає за зберігання та керування даними про музичні композиції. Вона містить такі поля:

- основні метадані треків, такі як title, artist, album, genre, year, що дає можливість користувачеві шукати треки за декількома параметрами;
- поля filename та filenameInSystem містять назву файлу та його системне уявлення, забезпечуючи доступ до вмісту на рівні файлової системи;
- поля coverFile, coverFileName та coverFileType дозволяють зберігати обкладинки треків, покращуючи візуальне подання контенту;
- поле uploader пов'язує кожен трек з обліковим записом, який його завантажив, створюючи базу даних для відстеження володіння та прав доступу;
- схема також має індекс для текстового пошуку по полях title, artist, album та genre, що забезпечує покращені можливості пошуку, особливо при великому обсязі даних.

3.3.5 Схема Session: Управління сесіями користувачів

Сесію користувача організовано у вигляді схеми Session, де реалізовано наступне:

- поле sessionToken зберігає унікальний токен для ідентифікації поточної сесії користувача, що є важливим для авторизації та безпеки;

- поля `createDate` та `dueDate` встановлюють тимчасові рамки дії сесії, забезпечуючи контроль та управління терміном активності;
- поле `account` пов'язує сесію з конкретним обліковим записом, гарантуючи, що активна сесія завжди асоціюється з одним користувачем.

3.4 Реалізація API controllers

3.4.1 AccountController

У контролері `AccountController` реалізовано комплексний інтерфейс для управління обліковими записами, що охоплює процеси аутентифікації, реєстрації, оновлення даних профілю та завершення сесій. Використовуючи інструкції та механізми авторизації, контролер забезпечує наступне:

- аутентифікація та реєстрація – методи `login` та `signup` обробляють запити на вхід та створення облікового запису, повертаючи статуси, що вказують на успішну авторизацію або наявність помилок (наприклад, вже існуючого запису);
- завершення сесії – метод `logout` дозволяє завершити активну сесію, забезпечуючи надійний вихід користувача із системи;
- оновлення даних профілю – метод `change` реалізує можливість внесення змін до профілю користувача, перевіряючи справжність сесії та наявність можливих конфліктів;
- отримання профілю та інформації – методи `getProfile` та `getProfileInfo` дозволяють отримати інформацію про поточний профіль та основну публічну інформацію про обліковий запис, що може використовуватись для відображення профілів інших користувачів в обмеженому вигляді;

Цей контролер є важливою частиною механізму безпеки та управління профілями у застосунку, підтримуючи авторизацію та аутентифікацію з використанням токенів та системи ролей.

3.4.2 FavoriteController

У контролері FavoriteController реалізований інтерфейс для управління списком улюбленої музики користувача. Використовуючи механізми аутентифікації, контролер надає такі функціональні можливості:

- додавання до улюблених – метод `addFavorite` обробляє запит на додавання музичного треку до списку улюблених. Він перевіряє дійсність сесії користувача і додає трек, якщо його ще немає у списку;
- видалення з улюблених – метод `deleteFavorite` дозволяє видалити конкретний трек з улюблених за наявності чинного сесійного токена;
- отримання списку улюблених треків – метод `getFavorite` надає користувачеві список всіх треків, які він позначив як улюблені;
- перевірка статусу треку – метод `isFavorite` перевіряє, чи є певний музичний трек у списку улюблених користувача.

Цей контролер є важливою складовою функціональності застосунку, забезпечуючи зручне управління музичними вподобаннями користувача та їх зберігання, а також дозволяючи швидкий доступ до вибраних треків.

3.4.3 HistoryController

У контролері HistoryController реалізовані методи для управління історією прослуховування музики користувачем, що дозволяє відстежувати та зберігати його музичні вподобання. Контролер виконує наступні функції:

- додавання запису до історії – метод `addRecord` дозволяє додати новий запис про прослуховування музичного треку. Після перевірки дійсності сесійного токена трек зберігається в історії користувача;
- отримання списку записів з історії – метод `getList` надає користувачеві можливість отримати повний список музичних треків, прослуханих ним за певний період.

Контролер є невід’ємною частиною функціоналу застосунка, оскільки забезпечує збереження історії прослуховування, що в свою чергу дозволяє покращувати персоналізацію рекомендацій та аналіз вподобань користувачів.

3.4.4 MusicController

Контролер MusicController у даному застосунку реалізує функціонал для управління завантаженням, пошуком та потоковою передачею музичних файлів і обкладинок. Основні функції цього контролера охоплюють:

- завантаження музичних файлів – метод `uploadMusic` обробляє завантаження аудіофайлів. Файл проходить перевірку на тип, і якщо це аудіо, дані про файл і метадані зберігаються в базі даних;
- завантаження обкладинок для музичних файлів – метод `addCover` забезпечує можливість завантаження зображень обкладинок, перевіряючи тип файлу як зображення;
- потокове передавання музики та обкладинок – методи `getFile` і `getCoverFile` дозволяють передавати користувачам музичні треки та обкладинки. Файли завантажуються з файлової системи та повертаються користувачам із належними заголовками для коректного відтворення.
- отримання списку музичних треків – метод `getList` надає користувачам список доступних музичних треків, що зберігаються в базі даних;
- пошук та рекомендації – методи `getListRecomendations` і `getSearchList` реалізують функціонал пошуку подібних треків на основі уподобань користувача та загального пошуку за ключовими словами. Це забезпечує персоналізований досвід прослуховування та зручну навігацію в музичному контенті.

Контролер забезпечує зручний інтерфейс для взаємодії користувачів із сервісом музики, зберігаючи при цьому необхідний рівень безпеки та перевірок.

3.5 Оптимізація програми

Оптимізація Angular застосунків включає декілька стратегій, що дозволяють підвищити продуктивність, зменшити час завантаження і зробити інтерфейс більш плавним і швидким. Ключові техніки включають використання `ChangeDetectionStrategy.OnPush`, `async` пайпу, `standalone` компонентів, `lazy loading` для модулів, і застосування пайпів для обробки даних [55]. Розглянемо кожен з них детально.

3.5.1 `ChangeDetectionStrategy.OnPush`

Однією з потужних стратегій оптимізації є `ChangeDetectionStrategy.OnPush`. За замовчуванням в Angular використовується стратегія `Default`, яка реагує на будь-які зміни в моделі даних, запускаючи детекцію змін по всій деревовидній структурі компонентів. Це може призводити до зайвого навантаження, особливо у великих застосунках. Застосування стратегії `OnPush` змінює поведінку Angular таким чином, що перевірка на зміни виконується лише тоді, коли змінюються вхідні значення (через `@Input`), коли відбуваються події DOM або коли компонент запускає асинхронну операцію (як-от `Observable`). `OnPush` дозволяє уникнути зайвих перевірок, оптимізуючи продуктивність компонентів, і особливо корисний для компонентів, які отримують дані через асинхронні потоки або мають стабільний набір даних [56].

3.5.2 async пайп

Async пайп – це спеціальний пайп, що дозволяє автоматично підписуватися і відписуватися від Observable чи Promise, спрощуючи роботу з асинхронними даними і підвищуючи зручність коду. Використовуючи async пайп, можна уникнути явного підписування на потоки через `.subscribe()`, що спрощує очищення підписок і зменшує ризик витоків пам'яті. Також, async автоматично відновлює значення при оновленні даних, що ідеально підходить для сценаріїв, коли компонент отримує потік даних і відображає їх у шаблоні. Це значно оптимізує рендерінг, так як Angular реагує лише на фактичні зміни даних [57].

3.5.3 Standalone компоненти

Запровадження standalone компонентів в Angular дозволяє створювати компоненти, які не залежать від модулів (NgModules). Це робить структуру коду легшою, адже зникає потреба в модульному обгортанні компонентів, що підвищує модульність і спрощує перевикористання. Standalone компоненти дозволяють на пряму імпортувати тільки необхідні залежності та створювати більш ізольовані і продуктивні елементи застосунку. Такі компоненти особливо корисні для створення невеликих, перевикористаних частин інтерфейсу і спрощують код для застосування компонентів у різних частинах застосунку [58].

3.5.4 Lazy Loading

Lazy loading, або ліниве завантаження модулів, є важливою стратегією оптимізації, яка дозволяє завантажувати модулі лише тоді, коли вони фактично потрібні користувачеві. У Angular це реалізується шляхом

динамічного завантаження модулів через маршрутизацію. Замість завантаження усіх модулів одразу при старті застосунку, *lazy loading* дозволяє завантажувати модулі лише на певних маршрутах. Це значно зменшує час завантаження сторінки, підвищуючи швидкість завантаження головного інтерфейсу та покращуючи загальний досвід користувачів. Застосування *lazy loading* до важких або рідко використовуваних модулів допомагає розподілити навантаження і знизити обсяг пам'яті, яку займає застосунок [59].

3.5.5 Пайпи (Pipes)

Пайпи – це невеликі функції для трансформації даних в Angular, що дозволяють обробляти дані безпосередньо в шаблоні, скорочуючи потребу у складних обчисленнях у компонентах. Це ефективний інструмент, особливо для відображення форматованих даних, таких як дати, числа або текстові трансформації. Крім того, кастомні пайпи можуть бути створені для реалізації специфічних перетворень, наприклад, фільтрації або сортування. Пайпи дозволяють спростити логіку компонентів, виносячи її у перевикористовувані та ізольовані функції, які Angular виконує лише тоді, коли є зміни у вхідних даних. Це зменшує кількість зайвих перерахунків та поліпшує продуктивність [60].

4 КЕРІВНИЦТВО ПРОГРАМІСТА

4.1 Характеристики програми

Розроблений застосунок складається з трьох основних елементів: frontend, backend та бази даних.

Backend відповідає за виконання різних взаємодій із базою даних і надає функціонал REST API для frontend-частини. Такий підхід забезпечує ефективний обмін даними та узгоджену роботу між компонентами застосунка.

На рисунку 4.1 зображено діаграму ієрархії збудованої архітектури.

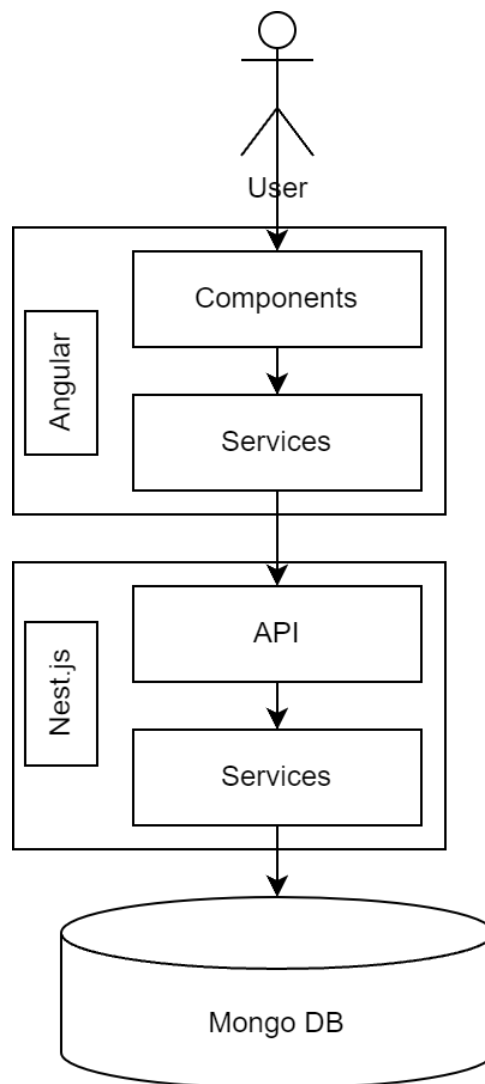


Рисунок 4.1 – Діаграма ієрархії збудованої архітектури

На рисунку 4.2 зображено файлову структуру frontend-y.

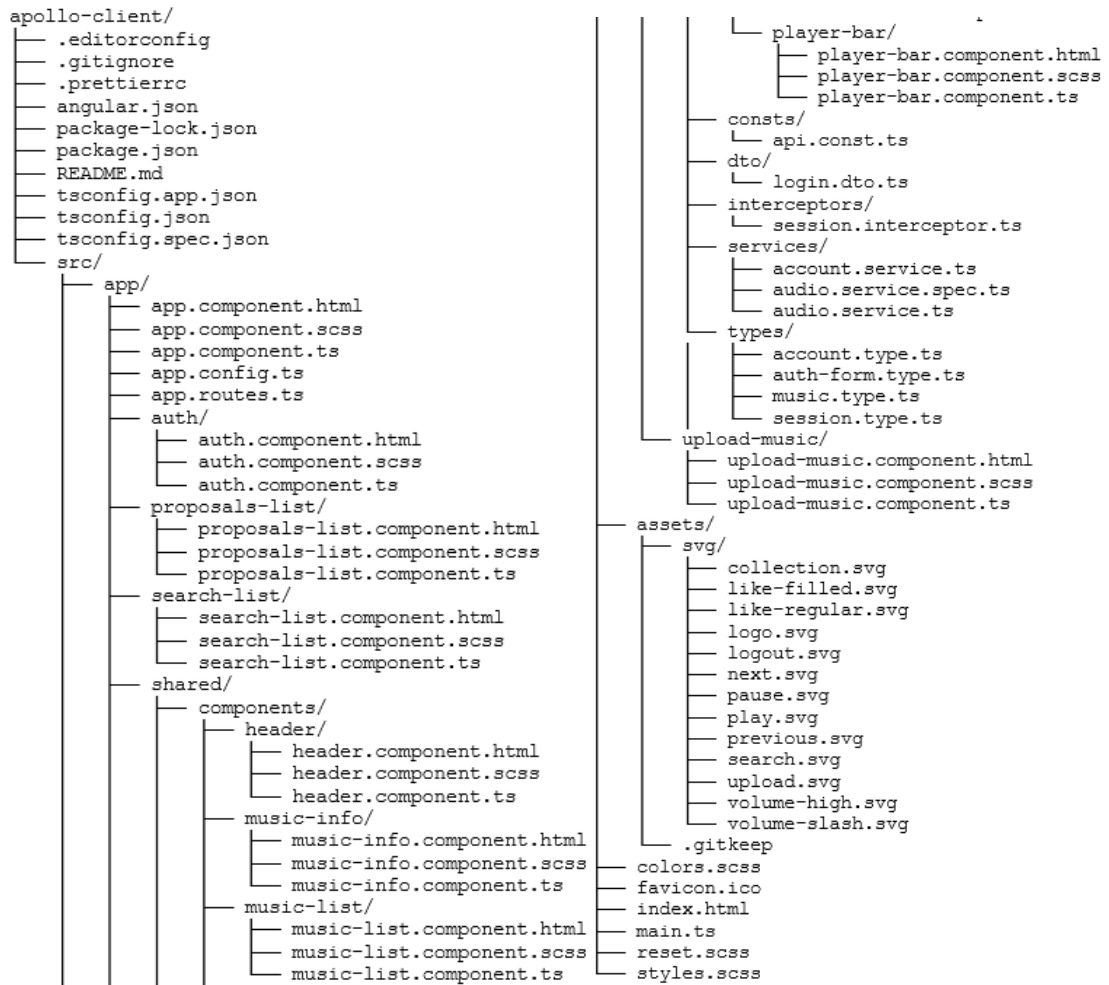


Рисунок 4.2 – Файлова структура frontend-y

На рисунку 4.3 зображено файлову структуру backend-y.

```

apollo-server/
├── .eslintrc.js
├── .gitignore
├── .prettierrc
├── nest-cli.json
├── package-lock.json
├── package.json
├── README.md
├── tsconfig.build.json
├── tsconfig.json
└── src/
    ├── app.controller.ts
    ├── app.module.ts
    ├── app.service.ts
    ├── main.ts
    ├── account/
    │   ├── account.controller.ts
    │   ├── account.module.ts
    │   ├── account.schema.ts
    │   └── account.service.ts
    │   └── dto/
    ├── favorite/
    │   ├── favorite.controller.ts
    │   ├── favorite.module.ts
    │   ├── favorite.schema.ts
    │   └── favorite.service.ts
    ├── history/
    │   ├── history.controller.ts
    │   ├── history.module.ts
    │   ├── history.schema.ts
    │   └── history.service.ts
    ├── music/
    │   ├── music.controller.ts
    │   ├── music.module.ts
    │   ├── music.schema.ts
    │   └── music.service.ts
    ├── session/
    │   ├── session.module.ts
    │   ├── session.schema.ts
    │   └── session.service.ts
    ├── shared/
    │   ├── consts/
    │   │   └── session.const.ts
    │   ├── guards/
    │   │   └── auth.guard.ts
    │   └── interceptors/

```

Рисунок 4.3 – Файлова структура backend-у

Код обох застосунків зберігається в єдиному Git-репозиторії, розміщеному на платформі GitHub. Такий підхід забезпечує централізоване управління версіями та полегшує працю над проєктом.

Під час процесу компіляції код, написаний на мові TypeScript, трансформується в JavaScript-файли, які згодом піддаються стисненню. Ця оптимізація має на меті зменшити розмір кінцевих файлів, що призводить до більш швидкого завантаження застосунків у браузерах користувачів та зниження навантаження на сервер.

Скомпільовані версії застосунків мають такі розміри:

- `apollo-server` (бекенд) – 266 КБ (273 269 байт). Бекенд-частина відповідає за обробку запитів клієнта та взаємодію з базою даних;
- `apollo-client` (фронтенд) – 559 КБ (573 396 байт). Фронтенд частина забезпечує взаємодію користувача з застосунком та відображення даних, отриманих від бекенду.

4.2 Залежності проєкту

4.2.1 Залежності клієнтського проєкту

Основу клієнтського застосунка формують його залежності, забезпечуючи потужний функціонал для розробки, тестування та роботи з медіафайлами. Вони підтримують ефективний процес розробки та виконання вимог застосунка до продуктивності та зручності використання.

Основні залежності `apollo-client`:

- `@angular/animations` – відповідає за реалізацію анімаційних ефектів у застосунку. Використовується для створення плавного інтерфейсу, зокрема під час переходів між компонентами чи станами;
- `@angular/common` – містить загальні функціональні можливості Angular, такі як інфраструктура для директив, пайпів і HTTP-запитів. Це базовий модуль для побудови компонентів;
- `@angular/compiler` – використовується для компіляції Angular-шаблонів у код, що виконується в браузері. Забезпечує динамічну генерацію коду для інтерактивних компонентів;
- `@angular/core` – серцевина фреймворку Angular, яка включає функціональність для роботи з компонентами, директивами та сервісами, необхідними для побудови застосунку;
- `@angular/forms` – забезпечує інструменти для створення та управління формами, включаючи реактивні та шаблонні підходи. Використовується для обробки даних користувача;

- @angular/platform-browser – відповідає за взаємодію Angular із браузером, забезпечуючи коректну обробку DOM-елементів;
 - @angular/platform-browser-dynamic – надає механізми для динамічного компілювання застосунків Angular у браузері. Важливий на етапі розробки;
 - @angular/router – реалізує систему маршрутизації в застосунку, дозволяючи організувати переходи між сторінками чи компонентами;
 - jsmediatags – бібліотека для зчитування метаданих із медіафайлів (ID3-теги). Використовується для обробки музичних файлів у застосунку;
 - music-metadata – надає розширені можливості для аналізу метаданих аудіофайлів. Корисна для отримання інформації про формат, трек чи альбом;
 - prettier – інструмент для форматування коду. Допомагає підтримувати єдиний стиль написання коду у проєкті;
 - rxjs – реактивна бібліотека, що використовується Angular для обробки асинхронних даних. Її Observable є основою для роботи з потоками даних;
 - tslib – збірка допоміжних бібліотек для TypeScript, що оптимізує роботу з функціями та класами;
 - zone.js – використовується для відстеження змін у зоні виконання JavaScript, що дозволяє Angular автоматично оновлювати UI.
- Залежності для розробки apollo-client:
- @angular-devkit/build-angular – інструмент для створення та розгортання Angular застосунків. Забезпечує зручність роботи з CLI;
 - @angular/cli – командний інтерфейс для генерації, тестування та розгортання Angular застосунків;
 - @angular/compiler-cli – забезпечує компіляцію Angular-шаблонів під час збирання застосунка;

- @types/jasmine – типи для тестування з Jasmine. Полегшують написання юніт-тестів у TypeScript;
- @types/node – типи для Node.js. Забезпечують інтеграцію з інструментами серверного оточення;
- jasmine core – ядро фреймворку Jasmine для написання юніт-тестів;
- karma – запускає юніт-тести у браузері, забезпечуючи зручну візуалізацію результатів;
- karma-chrome-launcher – надає підтримку запуску тестів у браузері Chrome;
- karma-coverage – інструмент для збору статистики про покриття коду тестами;
- karma-jasmine – інтеграція Jasmine із Karma для виконання тестів;
- karma-jasmine-html-reporter – репортер, який виводить результати тестування у зрозумілому HTML-форматі;
- typescript – мова програмування, яка компілюється у JavaScript. Angular базується на TypeScript, що надає йому строгий типізаційний контроль.

4.2.2 Залежності серверного проєкту

Основу серверної частини застосунка формують її залежності, які забезпечують масштабований, продуктивний та структурований процес розробки. Вони гарантують ефективну інтеграцію із базою даних, обробку HTTP-запитів та тестування. Наведемо детальний огляд основних і дев-залежностей.

Основні залежності серверної частини:

- @nestjs/common – містить загальні функціональні модулі NestJS, включаючи декоратори, життєвий цикл компонентів, інжектор залежностей і підтримку middleware;
- @nestjs/core – серцевина фреймворку NestJS, яка відповідає за маршрутизацію, модульність і структуру застосунка. Використовується для побудови сервісно-орієнтованої архітектури;
- @nestjs/mongoose – інтеграція NestJS із Mongoose для роботи з MongoDB. Забезпечує легкий доступ до бази даних через модельно-орієнтований підхід;
- @nestjs/platform-express – відповідає за інтеграцію NestJS із платформою Express.js. Дозволяє обробляти HTTP-запити і відповіді на сервері;
- mime-types – бібліотека для розпізнавання MIME-типів файлів. Використовується для визначення типу завантажуваних і оброблюваних файлів у застосунку;
- mongoose – бібліотека для роботи з MongoDB, що надає зручний інтерфейс для моделювання даних та виконання CRUD-операцій;
- music-metadata – інструмент для аналізу метаданих аудіофайлів. Застосовується для отримання детальної інформації про музичні треки, як-от альбом, виконавець чи тривалість;
- reflect-metadata – використовується для роботи з метаданими в TypeScript, необхідними для функціонування декораторів у NestJS;
- rxjs – реактивна бібліотека для обробки асинхронних операцій і потоків даних, що активно використовується у NestJS для роботи зі спостереженнями (Observables).

Залежності для розробки серверної частини:

- @nestjs/cli – командний інтерфейс для генерації компонентів NestJS, модулів та запуску застосунка;

- `@nestjs/schematics` – надає готові шаблони для генерації компонентів, контролерів, модулів та інших структур застосунка;
- `@nestjs/testing` – інструмент для модульного тестування компонентів та сервісів NestJS;
- `@types/express` – типи для Express.js, необхідні для коректної роботи з TypeScript у середовищі NestJS;
- `@types/jest` – типи для Jest – фреймворку тестування. Використовується для написання юніт-тестів;
- `@types/multer` – типи для бібліотеки Multer, що забезпечує завантаження файлів на сервер;
- `@types/node` – типи для Node.js, які допомагають працювати з його модулями, такими як fs, http тощо;
- `@types/supertest` – типи для Supertest – інструмента для тестування HTTP-запитів;
- `@typescript-eslint/eslint-plugin` – ESLint-плагін для забезпечення дотримання правил написання коду TypeScript;
- `@typescript-eslint/parser` – аналізує TypeScript-код і забезпечує підтримку правил ESLint;
- `eslint` – інструмент для перевірки стилю коду. Гарантує чистоту і читабельність коду;
- `eslint-config-prettier` – налаштування ESLint для сумісності з Prettier;
- `eslint-plugin-prettier` – інтегрує Prettier у ESLint для автоматичного форматування коду;
- `jest` – популярний фреймворк для написання та виконання модульних тестів;
- `prettier` – інструмент для автоматичного форматування коду відповідно до стилістичних вимог;

- source-map-support – забезпечує коректну обробку карт мапування (source maps) для зручнішого відлагодження;
- supertest – бібліотека для тестування HTTP-запитів до API, що дозволяє створювати інтеграційні тести;
- ts-jest – інструмент для використання TypeScript із Jest;
- ts-loader – завантажувач для інтеграції TypeScript із Webpack;
- ts-node – дозволяє виконувати TypeScript-код без попередньої компіляції у JavaScript;
- tsconfig-paths – забезпечує підтримку шляхів, визначених у файлі tsconfig.json;
- typescript – базова мова програмування для NestJS, яка гарантує строгий контроль типів і стабільність коду.

4.3 Встановлення застосунків локально

Для забезпечення ефективної роботи з застосунками у локальному середовищі необхідно здійснити низку кроків, кожен з яких має важливе значення для забезпечення належної роботи компонентів і служб.

Перш за все, одним із фундаментальних інструментів для роботи з репозиторіями є система контролю версій Git. Її встановлення є першочерговим завданням для будь-якого сучасного розробника. Завантажити останню версію можна з офіційного сайту (<https://git-scm.com/>). Після завершення процесу інсталяції рекомендується перезавантажити комп'ютер, щоб зміни у системних змінних середовища (environment variables) набрали чинності. Це забезпечує правильне функціонування Git як з командного рядка, так і в інших інструментах розробки.

Далі необхідно здійснити клонування репозиторію проєкту за допомогою команди `git clone <repo-domain>`. Цей крок дозволяє створити

локальну копію віддаленого сховища, що дає змогу працювати з кодом безпосередньо на пристрої.

Після отримання репозиторію важливо встановити базові програмні компоненти, що включають Node.js і MongoDB. Node.js – це середовище виконання JavaScript, що дозволяє запускати серверні скрипти. Його встановлення є необхідним для підтримки роботи сервісів та залежностей, що використовуються в застосунку. Завантаження здійснюється з офіційного сайту (<https://nodejs.org/en>). MongoDB, у свою чергу, є системою управління базами даних NoSQL і надає масштабованість і гнучкість у роботі з даними. Її можна завантажити за посиланням <https://www.mongodb.com/>. Після встановлення цих програмних компонентів рекомендується перезавантажити комп'ютер для автоматичного оновлення змінних середовища, що забезпечить доступність сервісів для інших процесів [51].

На наступному етапі необхідно скористатися менеджером пакетів Node Package Modules (npm) для встановлення глобальних залежностей, необхідних для роботи застосунку. Зокрема, для фронтенд-частини проєкту необхідно встановити Angular CLI командою `npm install -g @angular/cli`. Ця утиліта дозволяє створювати нові проєкти, запускати локальний сервер для розробки, збирати застосунок у виробничому середовищі тощо. Для бекенд-частини застосунку слід встановити NestJS CLI за допомогою команди `npm i -g @nestjs/cli`. NestJS забезпечує розробку серверних застосунків на основі Node.js, дозволяючи створювати модульні, масштабовані рішення.

Після глобальної установки необхідно перейти до встановлення усіх залежностей для обох проєктів, використовуючи команду `npm i` або `npm install`. У випадку виникнення проблем з інсталяцією можливо застосувати прапорці `-f` або `--force`, що дозволить примусове встановлення модулів. Це важливо для забезпечення сумісності залежностей, що можуть мати конфлікти версій. На рисунках 4.4-4.5 зображено запуск фронтенду та бекенду.

```

PS C:\Users\semen\git\apollo> cd .\apollo-client\
PS C:\Users\semen\git\apollo\apollo-client> ng s

Initial chunk files | Names | Raw size
polyfills.js | polyfills | 88.09 kB |
main.js | main | 31.53 kB |
chunk-P2IPZXGU.js | - | 4.04 kB |
styles.css | styles | 2.13 kB |
chunk-K56L2NUH.js | - | 1.72 kB |

Initial total | 127.50 kB

Lazy chunk files | Names | Raw size
chunk-SU5535XC.js | upload-music-component | 11.00 kB |
chunk-RT06SYN5.js | - | 8.15 kB |
chunk-LZQKZHAZ.js | proposals-list-component | 4.66 kB |
chunk-RLMSFA6K.js | search-list-component | 2.14 kB |

Application bundle generation complete. [2.410 seconds]

Watch mode enabled. Watching for file changes...
→ Local: http://localhost:4200/
→ press h + enter to show help

```

Рисунок 4.4 – Запуск фронтенду

```

[22:28:47] Starting compilation in watch mode...

[22:28:49] Found 0 errors. Watching for file changes.

[Nest] 5264 - 16.11.2024, 22:28:50 LOG [NestFactory] Starting Nest application...
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] MongooseModule dependencies initialized +20ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] AppModule dependencies initialized +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] MongooseCoreModule dependencies initialized +6ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] MongooseModule dependencies initialized +6ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] MongooseModule dependencies initialized +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] MongooseModule dependencies initialized +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] SessionModule dependencies initialized +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] MusicModule dependencies initialized +2ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] HistoryModule dependencies initialized +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] FavoriteModule dependencies initialized +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [InstanceLoader] AccountModule dependencies initialized +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RoutesResolver] AppController {/}: +3ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/, GET} route +2ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RoutesResolver] AccountController {/account}: +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/account/login, POST} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/account, POST} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/account/logout, HEAD} route +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/account, PATCH} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/account, GET} route +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/account/info, GET} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RoutesResolver] MusicController {/music}: +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/music/upload/audio, POST} route +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/music/upload/cover, POST} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/music/stream/:musicId, GET} route +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/music/cover/:musicId, GET} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/music/list, GET} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/music/similar-recomendations-list, GET} route +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/music/search, GET} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RoutesResolver] HistoryController {/history}: +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/history, POST} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/history/list, GET} route +1ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RoutesResolver] FavoriteController {/favorite}: +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/favorite, POST} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/favorite, DELETE} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/favorite/list, GET} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [RouterExplorer] Mapped {/favorite, GET} route +0ms
[Nest] 5264 - 16.11.2024, 22:28:50 LOG [NestApplication] Nest application successfully started +3ms

```

Рисунок 4.5 – Запуск бекенду

Наступний крок – запуск застосунків у двох окремих терміналах для сервісів `apollo-server` та `apollo-client`. Для запуску сервера використовується команда `nest start`, яка забезпечує виконання серверної частини. Якщо потрібно забезпечити автоматичну компіляцію і оновлення сервера під час редагування коду, рекомендується запускати сервер із прапорцем `--watch` або `-w`. Фронтенд застосунок (`apollo-client`) можна запустити командою `ng s`, `ng serve` або `npm start`, що запускає локальний сервер для розробки і дозволяє миттєво перевіряти зміни у браузері.

Завершальним етапом є відкриття браузера за адресою `http://localhost:4200/`, що дозволить запустити і протестувати застосунок у локальному середовищі. Це є важливою складовою циклу розробки, яка дозволяє виявити можливі проблеми на ранніх стадіях і забезпечити належну функціональність програми перед її розгортанням у виробничому середовищі.

4.4 Розгортання застосунків на сервері

Розгортання сучасних вебзастосунків на сервері є важливою частиною життєвого циклу розробки програмного забезпечення, яке забезпечує його доступність для кінцевих користувачів і стабільність у виробничому середовищі. Розроблено застосунки, що побудовані з використанням сучасних технологій, таких як `Angular` та `NestJS`, вимагають комплексного підходу до налаштування і підтримки серверного середовища. Для забезпечення оптимальної роботи та високої продуктивності, процес розгортання включає кілька етапів, що потребують особливої уваги та експертизи.

Першим етапом розгортання є підготовка серверного середовища. Це включає вибір відповідної платформи для хостингу, яка може бути як фізичним сервером, так і хмарним сервісом. Серед популярних варіантів для

хмарного хостингу слід виділити такі рішення, як AWS (Amazon Web Services), Microsoft Azure або GCP (Google Cloud Platform). Кожна з цих платформ надає широкий набір інструментів для масштабування та управління ресурсами, що дозволяє ефективно обробляти великий обсяг запитів і забезпечувати високу доступність застосунку.

Наступним важливим кроком є налаштування серверного оточення. Для застосунків необхідно забезпечити встановлення Node.js у виробничому середовищі, адже саме цей інструмент є основою для роботи як клієнтської, так і серверної частин застосунку. Рекомендується використовувати останню стабільну версію Node.js, що підтримує всі необхідні функції та безпекові оновлення. Інсталяція здійснюється за допомогою командного інтерфейсу шляхом завантаження пакету з офіційного сайту (<https://nodejs.org/>) або через менеджери пакетів, такі як NVM (Node Version Manager).

Для підтримки роботи з базою даних MongoDB слід розгорнути сервер MongoDB, що може бути здійснено на тому ж сервері або за допомогою спеціалізованих хмарних рішень, таких як MongoDB Atlas. Використання хмарного сервісу дозволяє забезпечити автоматичне резервне копіювання, масштабованість і високу доступність бази даних. Підключення до бази даних має бути належним чином захищене за допомогою налаштування шифрування з'єднання і обмеження доступу.

Для розгортання розроблених застосунків на сервері необхідно скопіювати файли проєкту за допомогою таких інструментів, як SCP (Secure Copy Protocol) або за допомогою налаштування автоматизованих процесів CI/CD (Continuous Integration/Continuous Deployment) з використанням сервісів, таких як GitHub Actions або GitLab CI/CD. Це забезпечує автоматизацію процесу доставки коду на сервер, що значно скорочує час розгортання і знижує ймовірність людських помилок.

Після перенесення файлів на сервер слід встановити всі необхідні залежності за допомогою команди `npm install`. Це дозволяє завантажити всі

залежності, необхідні для коректної роботи застосунку у виробничому середовищі. Щоб мінімізувати час простою і забезпечити стабільність, слід використовувати пакетні менеджери, такі як pm2 для Node.js, які дозволяють запускати застосунки в безперервному режимі, забезпечують автоматичний перезапуск у випадку помилки і дозволяють зручно моніторити стан застосунку за допомогою команд `pm2 start`, `pm2 stop`, `pm2 restart` тощо.

Для клієнтської частини застосунку на базі Angular потрібно зібрати виробничу версію за допомогою команди `ng build --prod`, яка створює оптимізований пакет з мінімізацією розміру і підвищеною продуктивністю. Після цього статичні файли можна розгорнути на вебсервері, такому як Nginx або Apache, який буде обробляти запити і забезпечувати доставку контенту користувачам. Налаштування конфігураційних файлів сервера має включати правила маршрутизації для правильного оброблення маршрутизації на стороні клієнта.

Безпека виробничого середовища є критичним аспектом. Необхідно налаштувати захищені з'єднання HTTPS, що забезпечується шляхом встановлення SSL-сертифікатів за допомогою таких сервісів, як Let's Encrypt. Це дозволяє шифрувати дані, що передаються між клієнтом і сервером, і підвищує загальний рівень безпеки застосунку. Додатково, для підвищення захисту серверу слід впроваджувати політики контролю доступу і налаштування файрволів.

На завершення, моніторинг роботи застосунку у виробничому середовищі є необхідним для підтримки високої продуктивності та швидкого реагування на можливі проблеми. Це включає використання таких інструментів, як Prometheus та Grafana для збору і візуалізації метрик, що допомагає виявити потенційні вузькі місця у системі і планувати подальшу оптимізацію інфраструктури.

4.5 Можливі способи покращення програми

Для покращення програми можна впровадити деякі покращення, які будуть перелічені далі.

4.5.1 Алгоритми рекомендацій

Алгоритми рекомендацій є потужним інструментом для підвищення якості взаємодії користувачів з застосунком, оскільки дозволяють персоналізувати контент та адаптувати його під індивідуальні уподобання. При розробці та впровадженні рекомендаційних алгоритмів у розроблений вебзастосунок можна застосовувати декілька підходів, кожен з яких має свої особливості та переваги залежно від характеру застосунка та обсягу даних.

Одним із найпоширеніших методів є алгоритми на основі контенту (content-based), які аналізують характеристики самого контенту і порівнюють його з вподобаннями користувача. Наприклад, якщо користувач слухає певний музичний жанр система може пропонувати контент, що відповідає подібним характеристикам. Це дозволяє забезпечити високий рівень релевантності рекомендацій без надмірної обробки великих обсягів даних.

Іншим підходом є колаборативна фільтрація (collaborative filtering), яка використовує поведінкові дані від інших користувачів для генерації рекомендацій. Такі алгоритми шукають користувачів зі схожими вподобаннями або дії, що часто збігаються, і на основі цього формують рекомендації. Наприклад, користувачу, який прослухав низку композицій, можна порекомендувати треки, які популярні серед інших слухачів з подібними смаками. Цей метод особливо ефективний для платформ із великою кількістю користувачів та даних про їхню взаємодію. Такий метод також реалізовано у програмі.

Також варто розглянути комбіновані або гібридні рекомендаційні системи, що поєднують контентну фільтрацію та колаборативну фільтрацію для досягнення кращої точності. Гібридні системи можуть пропонувати користувачам більше варіантів та дозволяють уникнути обмежень кожного з підходів окремо. Наприклад, такі системи можуть використовувати контентну фільтрацію для користувачів-новачків, у яких недостатньо даних про попередні взаємодії, та колаборативну фільтрацію для більш досвідчених користувачів із багатою історією переглядів.

Сучасні застосунки також застосовують алгоритми на базі машинного навчання та нейронних мереж, такі як глибоке навчання для побудови моделей, що розпізнають складні патерни в даних. Такі моделі здатні обробляти не лише інформацію про контент та поведінку користувачів, але й враховувати численні особисті фактори, тренди та сезонні вподобання. Нейронні мережі дозволяють створювати точні прогнози та робити високоточні рекомендації, особливо у випадку великих обсягів даних, які змінюються з часом.

Нарешті, перспективним напрямком розвитку є алгоритми на основі соціальних зв'язків (social-based recommendation), які враховують інформацію про коло спілкування користувачів. Цей метод може бути корисним, коли користувачі більшою мірою довіряють рекомендаціям своїх друзів та знайомих, особливо у музичних платформах.

Таким чином, рекомендаційні алгоритми є надзвичайно перспективним напрямком розвитку для розроблюваного застосунку. Їх адаптація під потреби користувача здатна значно покращити ефективність застосунку, посилити залучення користувачів та підвищити рівень їхньої задоволеності. На рисунку 4.6 зображено блок-схему поліпшеного алгоритму розрахунку рекомендацій.



Рисунок 4.6 – Блок-схема поліпшеного алгоритму розрахунку рекомендацій

4.5.2 Взаємодія з обліковим записом

В межах подальшого розвитку можливостей застосунку можна передбачити комплекс удосконалень для збагачення користувацького досвіду, орієнтованого на функціональну зручність, безпеку та інтеграцію з іншими сервісами:

- зміна пароля та підтвердження через електронну пошту. Функція зміни пароля дозволить користувачам надійно оновлювати дані, забезпечуючи актуальність облікової інформації. Підтвердження електронної пошти через верифікаційне посилання додає рівень захисту, що є особливо важливим для верифікації власника облікового запису та запобігання доступу до аккаунту третіми особами;

- авторизація через сторонні сервіси. Інтеграція із зовнішніми сервісами, як-от Google та Microsoft, дозволить користувачам входити в обліковий запис без необхідності запам'ятовування додаткових паролів, що зменшує ймовірність забуття даних для входу. Це також поліпшить загальний досвід користування застосунком, адже аутентифікація стає швидшою та більш зручною;

- функція спільного прослуховування. Користувачі зможуть разом слухати музичні треки або плейлисти в режимі реального часу, обговорюючи їх або просто насолоджуючись спільним прослуховуванням. Впровадження такої функціональності сприятиме соціальній інтеракції користувачів, розширюючи кордони індивідуального користування;

- можливість підписки на авторів дозволяє користувачам слідкувати за улюбленими музикантами, миттєво отримуючи оновлення про нові релізи або активність авторів, на яких вони підписані. Така функція розвиватиме інтерес до застосунку серед музичних ентузіастів і підтримуватиме залучення аудиторії;

– створення плейлистів, це забезпечить користувачам можливість персоналізації музичної бібліотеки відповідно до індивідуальних уподобань та настроїв. Розробка функції з гнучкими налаштуваннями сортування, додавання імен, описів та візуального оформлення плейлистів підвищить інтерактивність сервісу, адже користувачі зможуть поділитися своєю музикою з друзями та однодумцями.

4.5.3 Локалізація програми

Локалізація вебзастосунків є важливим аспектом сучасної розробки, який забезпечує підтримку багатомовності та дозволяє зробити застосунок доступним для ширшої аудиторії. У фронтенд застосунку можна використати бібліотеку ngx-translate, яка є популярним інструментом для ефективної локалізації в середовищі Angular. Вона забезпечує простий спосіб перекладу текстових елементів інтерфейсу і дозволяє підтримувати динамічні переклади без значного впливу на продуктивність застосунку. У цьому розділі буде розглянуто детальний процес інтеграції ngx-translate в Angular застосунок, а також описано основні практики, що забезпечують ефективне керування перекладами [61].

Для початку необхідно встановити ngx-translate та залежність @ngx-translate/http-loader, яка дозволяє завантажувати файли перекладів з сервера або локальної файлової системи. Це здійснюється за допомогою таких команд:

```
npm install @ngx-translate/core @ngx-  
translate/http-loader
```

Після цього необхідно імпортувати відповідні модулі в застосунок. У файлі app.config.ts слід додати модулі TranslateModule та TranslateLoader:

```
TranslateModule.forRoot({
  loader: {
    provide: TranslateLoader,
    useFactory: createTranslateLoader,
    deps: [HttpClient]
  }
})
```

Для локалізації необхідно створити JSON-файли з перекладами, які будуть розташовані в папці `assets/i18n`. Наприклад, для підтримки англійської та української мов можна створити файли `en.json` та `ua.json`:

`assets/i18n/en.json`:

```
{
  "WELCOME_MESSAGE": "Hello world!",
  "ADDITIONAL": "Some additional fields."
}
```

`assets/i18n/ua.json`:

```
{
  "WELCOME_MESSAGE": "Привіт, світ!",
  "ADDITIONAL": "Деякі додаткові поля."
}
```

Для використання перекладів у компонентах Angular, `ngx-translate` надає директиву `translate` та сервіс `TranslateService`. Наприклад, переклад можна легко відобразити у шаблоні за допомогою директиви `translate`:

```
<h1>{{ 'WELCOME_MESSAGE' | translate }}</h1>
<p>{{ 'ADDITIONAL' | translate }}</p>
```

Динамічне управління мовами здійснюється за допомогою сервісу `TranslateService`, який дозволяє змінювати мову у реальному часі:

```

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private translate: TranslateService)
{
    translate.addLangs(['en', 'ua']);
    translate.setDefaultLang('en');

    const browserLang =
translate.getBrowserLang();
    translate.use(browserLang.match(/en|ua/)
? browserLang : 'en');
  }

  switchLanguage(language: string) {
    this.translate.use(language);
  }
}

```

Відповідний HTML-шаблон може містити кнопку для зміни мови:

```

<button
(click)="switchLanguage('en')">English</button>
<button
(click)="switchLanguage('ua')">Українська</button>

```

Переваги використання ngx-translate:

- основною перевагою ngx-translate є його гнучкість та підтримка динамічного завантаження перекладів. Це особливо важливо для великих застосунків, де обсяг текстів може значно змінюватися. Крім того, ngx-translate дозволяє працювати з асинхронними запитами для завантаження перекладів, що підвищує продуктивність і дає змогу оновлювати переклади без перекомпіляції застосунку;

– іншою важливою функцією є можливість створення кастомних лодерів для перекладів, що дозволяє інтегрувати переклади з різних джерел, включаючи API.

5 ДОСЛІДЖЕННЯ І РЕЗУЛЬТАТИ

5.1 Опис методики проведення дослідження

В якості дослідження було проведено тестування працездатності застосунку. Для тестування використовувався персональний комп'ютер із такими системними характеристиками:

- центральний процесор – Intel Core i7-13700h;
- оперативна пам'ять – 32 гб;
- відеокарта – NVIDIA GeForce RTX 4060 Laptop.

Ручне тестування виконувалось у наступних програмних оточеннях:

- браузер – Mozilla Firefox 133.0 (64-розрядний);
- операційна система – Windows 11 Pro.

Також для тестування мобільної адаптації застосунка було використано смартфони Samsung Galaxy S24 та Samsung Galaxy S24Ultra.

Було проведено Юніт тестування та ручне тестування застосунка.

Юніт-тестування є одним із ключових підходів забезпечення надійності програмного забезпечення, особливо в складних вебзастосунках, побудованих на Angular [62]. Одним із важливих прикладів його застосування є тестування служб, що керують відтворенням аудіо файлів і обробкою пов'язаних з ними подій та станів.

У рамках ручного тестування було проведено повну перевірку функціональності клієнтського та серверного застосунків. Нижче подано докладний опис кожного етапу тестування, ілюстрований відповідними скріншотами.

5.2 Unit тестування у Angular

Розглянемо підхід до тестування служби AudioService в Angular за допомогою фреймворка Jasmine та інструментів для тестування HTTP-

запитів, таких як `HttpTestingController`. Мета цього тестування полягає в гарантуванні, що всі компоненти служби працюють відповідно до очікуваних сценаріїв, включаючи відтворення, призупинення, перемикання аудіо, а також інтеграцію з іншими системними компонентами, такими як локальне сховище та зовнішні API [62].

Розглянемо наступний код ініціалізації `AudioService`:

```
describe('AudioService', () => {
  let service: AudioService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        AudioService,
        provideHttpClient(),
        provideHttpClientTesting(),
      ],
    });
    service = TestBed.inject(AudioService);
    httpMock = TestBed.inject(HttpTestingController);

    spyOn(service.audioObject,
      'play').and.callFake(async () => {});
    spyOn(service.audioObject,
      'pause').and.callFake(async () => {});
  });

  afterEach(() => {
    httpMock.verify();
  });
});
```

Початкова конфігурація середовища тестування, що здійснюється через `TestBed.configureTestingModule`, є необхідним етапом, який дозволяє інкапсулювати середовище для тестування. Це забезпечує ізолюваність тестованого коду та емулює всі залежності, необхідні для повноцінного

функціонування служби, такі як HTTP-клієнт і система HTTP-запитів для тестування. Ін'єкція `HttpTestingController` забезпечує можливість перевірки HTTP-запитів і дає змогу тестовим сценаріям взаємодіяти з фальшивими сервісами замість реальних серверів. Таке налаштування є особливо важливим у випадках, коли реальні запити до сервера є небажаними або навіть неможливими через конфіденційність чи обмеження мережі.

Розглянемо наступний код початкового тестування запуску сервісу:

```
it('should be created', () => {
    expect(service).toBeTruthy();
});

it('should initialize with empty query and null
current audio', () => {
    expect(service.query$.value).toEqual([]);
    expect(service.currentAudio$.value).toBeNull();
});
```

Юніт-тестування в цьому контексті передбачає ретельну перевірку окремих аспектів функціональності служби. Наприклад, перевірка на успішне створення сервісу є фундаментальним аспектом, який дозволяє переконатися, що служба завантажена належним чином. Ця перевірка надзвичайно важлива для подальшої роботи, оскільки помилки на етапі створення можуть порушити функціонування всієї програми або призвести до несподіваних збоїв. Тестування початкових значень, таких як порожні значення в масиві запитів (`query$`) і `null` для поточного аудіо (`currentAudio$`), забезпечує додатковий рівень перевірки правильності початкової конфігурації.

Розглянемо наступний код ініціалізації перевірки `localStorage`:

```
it('should set volume from localStorage', () => {
    localStorage.setItem('volume', '0.5');
    service = TestBed.inject(AudioService);
});
```

```

    expect(service.audioObject.volume).toBe(0.5);
  });

  it('should set volume and save it to localStorage',
    () => {
      spyOn(localStorage, 'setItem');
      service.setVolume(0.7);
      expect(service.audioObject.volume).toBe(0.7);

      expect(localStorage.setItem).toHaveBeenCalled('volume', '0.7');
    });

```

Ще одним значним аспектом є тестування взаємодії з локальним сховищем (localStorage). Це гарантує, що всі налаштування користувача, наприклад, рівень гучності, зберігаються і відновлюються належним чином. За відсутності такої функціональності користувачам довелося б кожного разу налаштовувати гучність, що знижувало б зручність використання програми. Тестування локального сховища передбачає завантаження значень з localStorage, що симулює реальні умови використання, забезпечуючи максимальну реалістичність. Використання методу setVolume, що одночасно оновлює значення в аудіооб'єкті і зберігає його в localStorage, піддається ретельному аналізу. Цей тест підтверджує, що застосування служби залишатиметься незмінним, якщо користувачеві потрібно буде відновити рівень гучності після перезапуску програми.

Розглянемо наступний код тестування паузи та відтворення композиції:

```

it('should play and pause the audio', () => {
  service.playPauseMusic();

  expect(service.audioObject.play).toHaveBeenCalled();

  service.audioObject.pause();
  service.playPauseMusic();

```

```
expect(service.audioObject.pause).toHaveBeenCalled();
});

it('should set audio currentTime', () => {
  service.setTime(30);
  expect(service.audioObject.currentTime).toBe(30);
});
```

У цьому коді можна побачити тестування управління аудіоплеєром, зокрема функцій відтворення і паузи. Наприклад, метод `playPauseMusic` перевіряється на відповідне викликання функцій `play` та `pause` в аудіо об'єкті, що забезпечує належну поведінку служби. Такі тести виявляються корисними для підтримки стабільної роботи інтерфейсу користувача, де будь-які помилки у відтворенні можуть викликати розчарування або знизити загальну ефективність програми.

Розглянемо наступний код тестування :

```
it('should subscribe to currentAudio$ and update
audio source', () => {
  const mockMusic: Music = { _id: '1', title: 'Test
Music' } as Music;

  service.currentAudio$.next(mockMusic);

  expect(service.audioObject.src).toBe(`${API_MUSIC_STREA
M}/1`);

  const reqHistory = httpMock.expectOne(API_HISTORY);
  expect(reqHistory.request.method).toBe('POST');
  reqHistory.flush({});

  const reqFavorite =
httpMock.expectOne(`${API_FAVORITE}?musicId=1`);
  expect(reqFavorite.request.method).toBe('GET');
  reqFavorite.flush(true);

  expect(service.currentLiked$.value).toBe(true);
```

```
});
```

Підписка на зміну поточного аудіо через `currentAudio$` включає одночасне виконання запитів на сервер для додавання треку в історію і перевірки його статусу. Це допомагає переконатися, що користувачі можуть легко відслідковувати свою історію прослуховування та взаємодіяти з «вподобаними» треками.

Розглянемо наступний код тестування :

```
it('should select music and update query$', () => {
  const mockQuery: Music[] = [
    { _id: '1', title: 'Song 1' } as Music,
    { _id: '2', title: 'Song 2' } as Music,
  ];
  service.selectMusic(mockQuery);
  expect(service.query$.value).toEqual(mockQuery);

  expect(service.currentAudio$.value).toEqual(mockQuery[0]);

  const reqHistory = httpMock.expectOne(API_HISTORY);
  expect(reqHistory.request.method).toBe('POST');
  reqHistory.flush({});

  const reqFavorite = httpMock.expectOne(`${API_FAVORITE}?musicId=1`);
  expect(reqFavorite.request.method).toBe('GET');
  reqFavorite.flush(true);
});

const historyRequests = httpMock.match(API_HISTORY);
expect(historyRequests.length).toBeGreaterThan(0);
historyRequests.forEach(req => req.flush({}));

const favoriteRequests = httpMock.match(req =>
  req.url.includes(API_FAVORITE)
);
expect(favoriteRequests.length).toBeGreaterThan(0);
favoriteRequests.forEach(req => req.flush(true));
```

```
});
```

Тестування API-запитів на сервер для отримання інформації про історію і статус «вподобаного» аудіофайлу забезпечує коректність зв'язку між Angular застосунком та бекендом. Наявність можливості тестувати такі сценарії в реальних умовах з урахуванням імітованих HTTP-запитів гарантує, що застосунок зможе забезпечити ефективний і надійний користувацький досвід навіть за відсутності прямого зв'язку з сервером.

Розглянемо наступний код:

```
it('should go to next music', () => {
  const mockQuery: Music[] = [
    { _id: '1', title: 'Song 1' } as Music,
    { _id: '2', title: 'Song 2' } as Music,
  ];
  service.selectMusic(mockQuery);

  service.nextMusic();

  expect(service.currentAudio$.value).toEqual(mockQuery[1]);
});
```

Тестування переходу між треками (наприклад, функції nextMusic) допомагає переконатися, що переміщення по плейлисту працює коректно і всі необхідні зміни відбуваються належним чином. Це стосується як перемикання від одного аудіофайлу до іншого, так і поновлення відповідних даних, таких як історія і статус «вподобаного».

Загалом юніт-тестування служби AudioService у Angular сприяє підвищенню надійності застосунка за рахунок ретельної перевірки функцій відтворення аудіо, управління плейлистом, взаємодії з локальним сховищем і API-запитами. Використання HttpTestingController дозволяє налаштувати симуляцію запитів до сервера, що мінімізує залежність тестів від зовнішніх ресурсів і підвищує їхню швидкість. Цей процес допомагає переконатися, що

функціональні можливості сервісу відповідають очікуванням і забезпечують стабільність програми навіть при зміні API або внесенні інших суттєвих змін в архітектуру застосунка. Також було протестовано і інші сервіси, такі як AccountService, що відповідає за управління аккаунтом.

5.3 Manual тестування

Клієнтський інтерфейс застосунка був запущений за допомогою Angular CLI, результат можна побачити на рисунку 5.1.

```
PS C:\Users\semen\git\apollo\apollo-client> ng s
```

Initial chunk files	Names	Raw size
polyfills.js	polyfills	88.09 kB
main.js	main	31.53 kB
chunk-P2IPZXGU.js	-	4.04 kB
styles.css	styles	2.13 kB
chunk-K56L2NUH.js	-	1.72 kB
Initial total		127.50 kB

Lazy chunk files	Names	Raw size
chunk-SU5535XC.js	upload-music-component	11.00 kB
chunk-RT06SYN5.js	-	8.15 kB
chunk-LZQKZHAZ.js	proposals-list-component	4.66 kB
chunk-RLMSFA6K.js	search-list-component	2.14 kB

```

Application bundle generation complete. [2.600 seconds]

Watch mode enabled. Watching for file changes...
  → Local:   http://localhost:4200/
  → press h + enter to show help

```

Рисунок 5.1 – Запуск Angular застосунку

На серверній стороні було розгорнуто API за допомогою фреймворку Nest.js, результат можна побачити на рисунку 5.2.

```
PS C:\Users\semen\git\apollo\apollo-server> nest start
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [NestFactory] Starting Nest application...
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [InstanceLoader] MongooseModule dependencies initialized +19ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [InstanceLoader] AppModule dependencies initialized +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [InstanceLoader] MongooseCoreModule dependencies initialized +11ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/account, PATCH} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/account, GET} route +1ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/account/info, GET} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RoutesResolver] MusicController {/music}: +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/music/upload/audio, POST} route +1ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/music/upload/cover, POST} route +1ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/music/stream/:musicId, GET} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/music/cover/:musicId, GET} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/music/list, GET} route +1ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/music/similar-recomendations-list, GET} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/music/search, GET} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RoutesResolver] HistoryController {/history}: +1ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/history, POST} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/history/list, GET} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RoutesResolver] FavoriteController {/favorite}: +1ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/favorite, POST} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/favorite, DELETE} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/favorite/list, GET} route +1ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [RouterExplorer] Mapped {/favorite, GET} route +0ms
[Nest] 23876 - 25.11.2024, 21:03:53 LOG [NestApplication] Nest application successfully started +2ms
```

Рисунок 5.2 – Запуск Nest.js застосунку

Застосунок успішно відкривається за адресою localhost:4200 у браузері, результат можна побачити на рисунку 5.3.

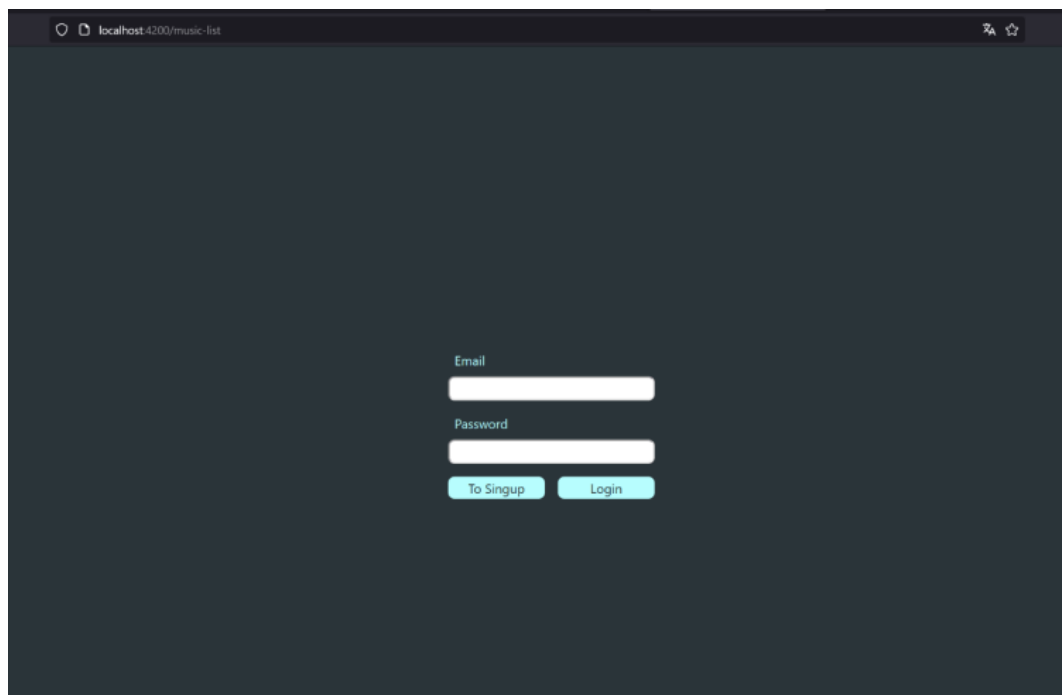
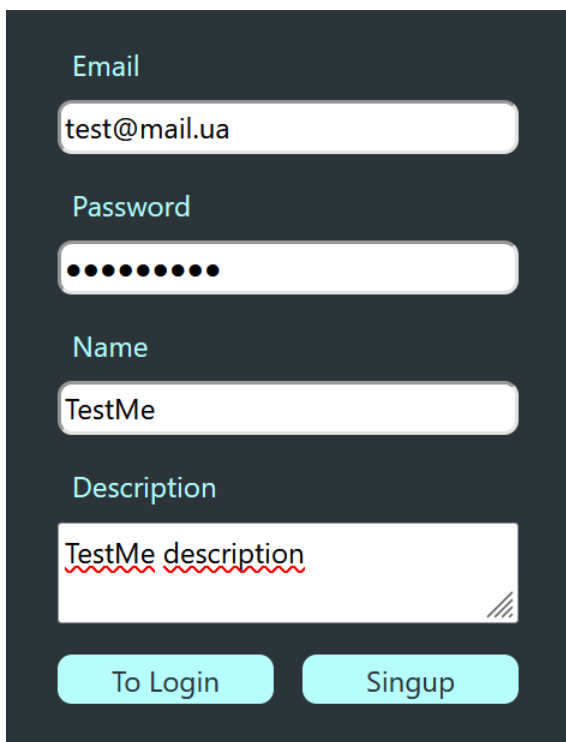


Рисунок 5.3. – Login сторінка Angular застосунку

Для перевірки функції реєстрації було здійснено заповнення форми та створення нового облікового запису, результат можна побачити на рисунку 5.4.



The image shows a dark-themed login/register form. It contains four input fields: 'Email' with the value 'test@mail.ua', 'Password' with masked characters, 'Name' with the value 'TestMe', and 'Description' with the value 'TestMe description'. Below the fields are two buttons: 'To Login' and 'Singup'.

Рисунок 5.4. – Login сторінка Angular застосунку

Було прослухано кілька композицій та перевірено, як відображаються композиції в історії прослуховувань.

Результат можна побачити на рисунку 5.5.

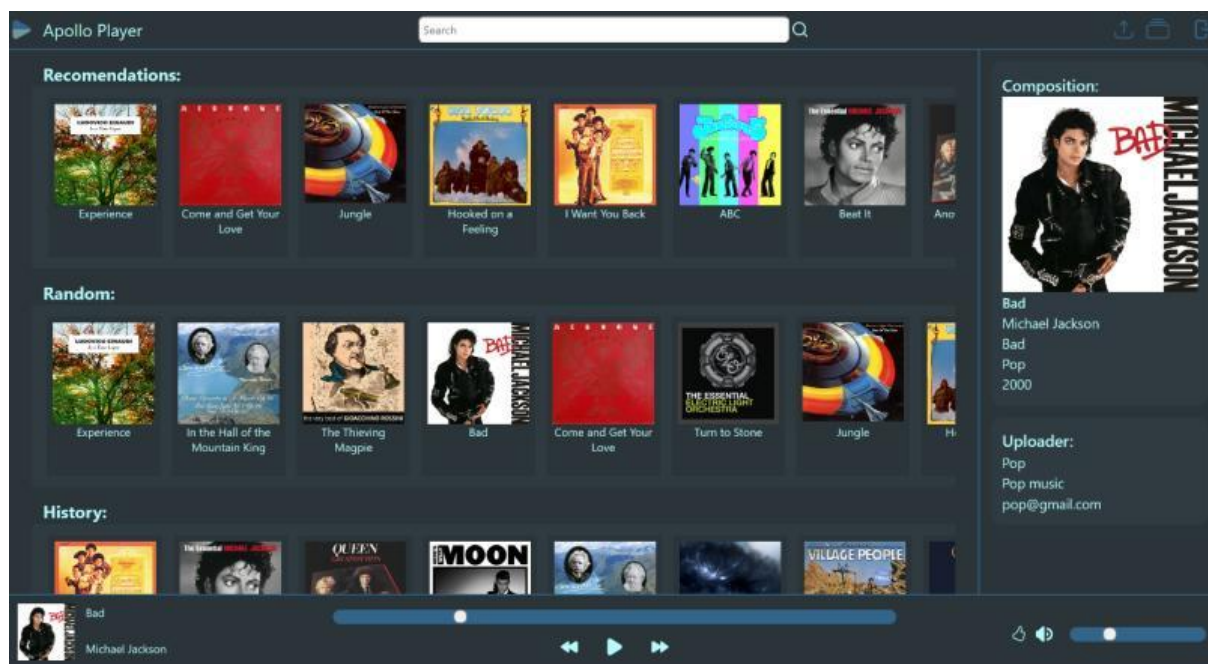


Рисунок 5.5 – Загальна сторінка для обрання музики

Було перевірено, чи генеруються рекомендації згідно з алгоритмом подібності. Також перевірено коректність відображення усіх прослуханих композицій в історії. Результат можна побачити на рисунку 5.6.

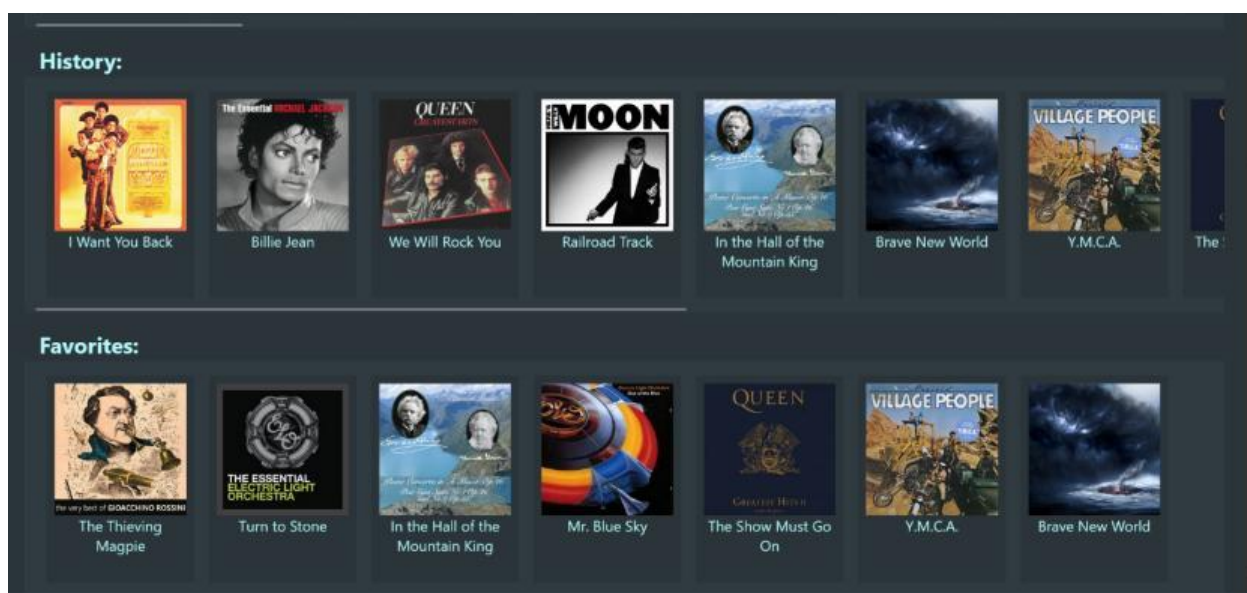


Рисунок 5.6 – Історія прослухань та обрані композиції

Обрана композиція була успішно додана до списку обраного після натискання відповідної кнопки. Кнопку яку потрібно нажати для додавання в обрані композиції можна побачити на рисунку 5.7.

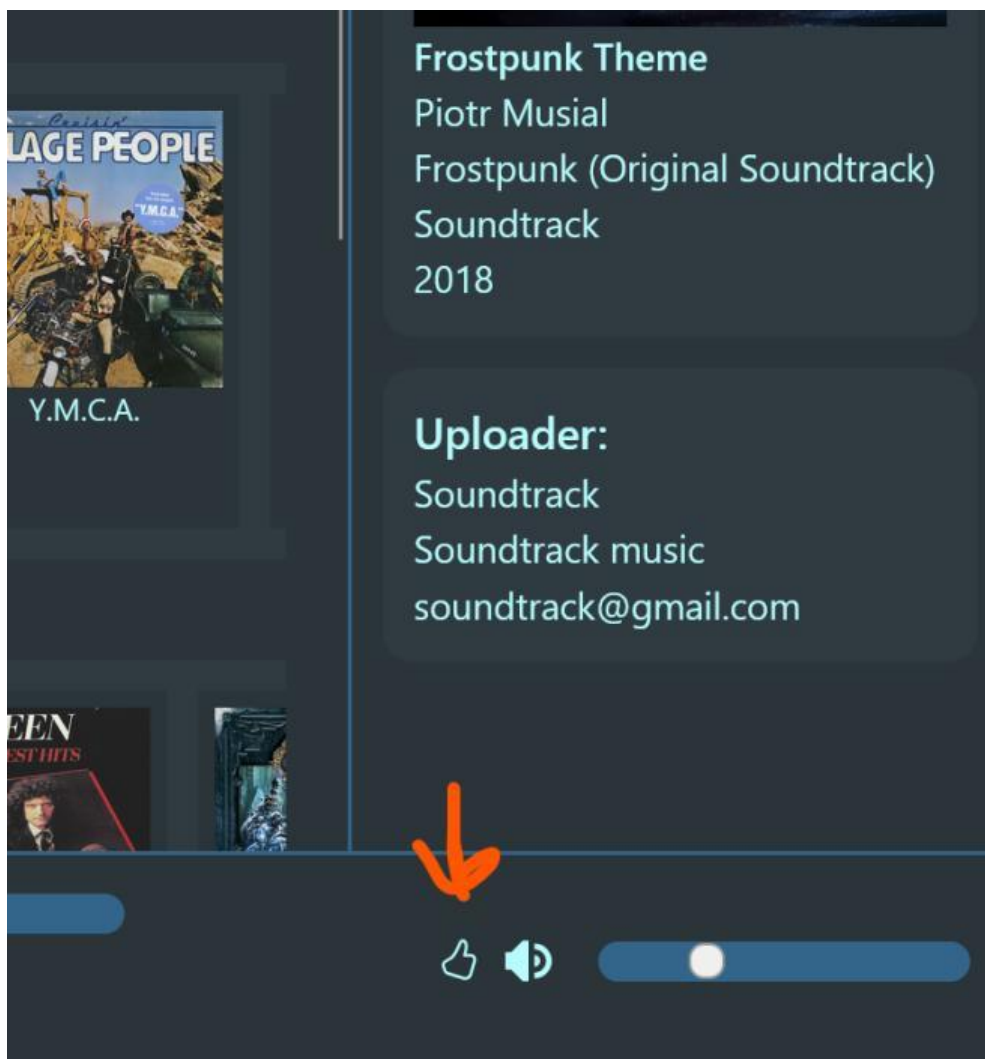


Рисунок 5.7 – Кнопка "Додати у обране"

Кнопка підсвітілась, і композиція додалася до списку. Результат можна побачити на 5.8.

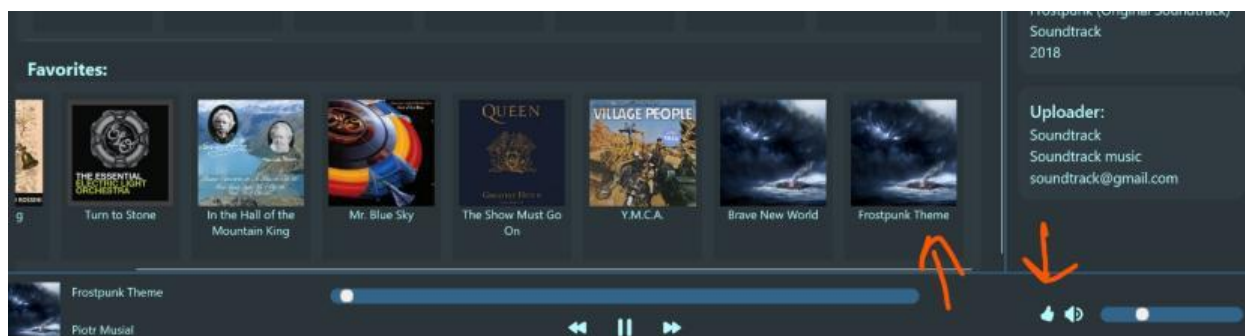


Рисунок 5.8 – Обрана композиція додалася у список

Було перевірено, що гучність успішно відрегульована. Та проведено перевірку після перезавантаження, що гучність залишилась на налаштованому рівні навіть після оновлення сторінки. Результати можна побачити на рисунку 5.9.

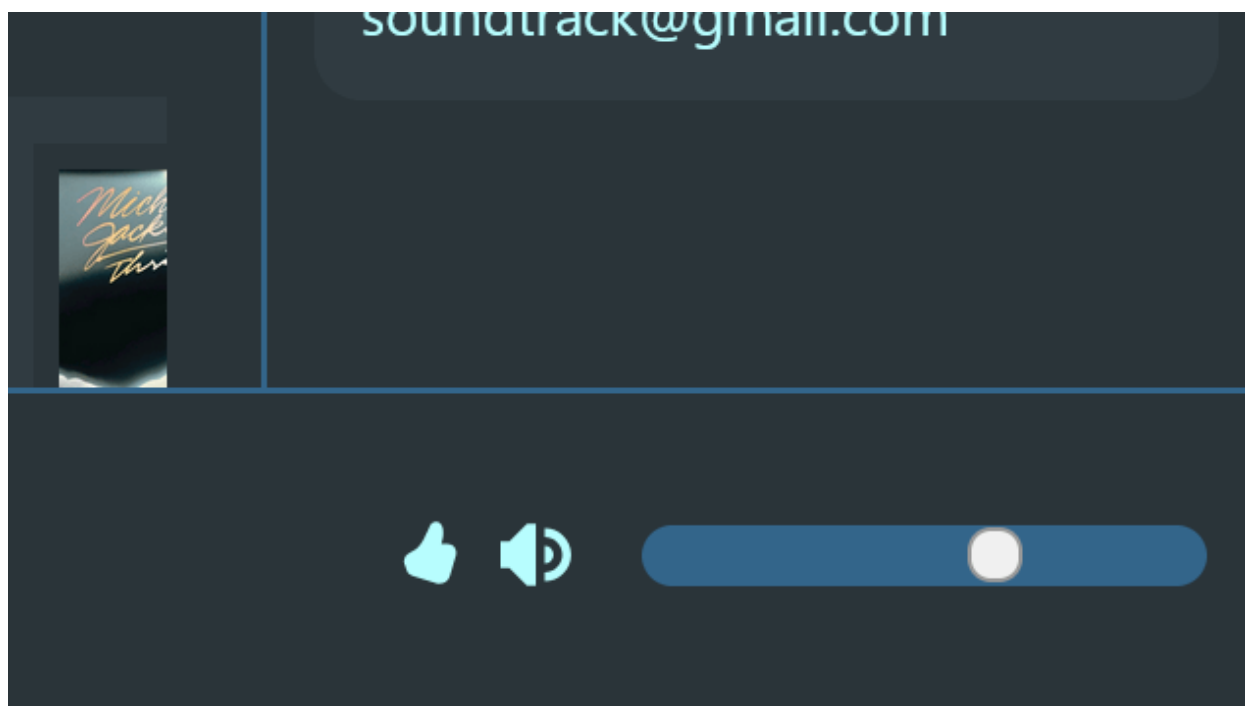


Рисунок 5.9 – Слайдер регулювання гучності

Перевірено, що при переході на наступну композицію користувача перенаправлено на наступний трек.

Перевірено, що при поверненні до попередньої композиції користувача перенаправлено до попередньої пісні. Результат можна побачити на рисунку 5.10.

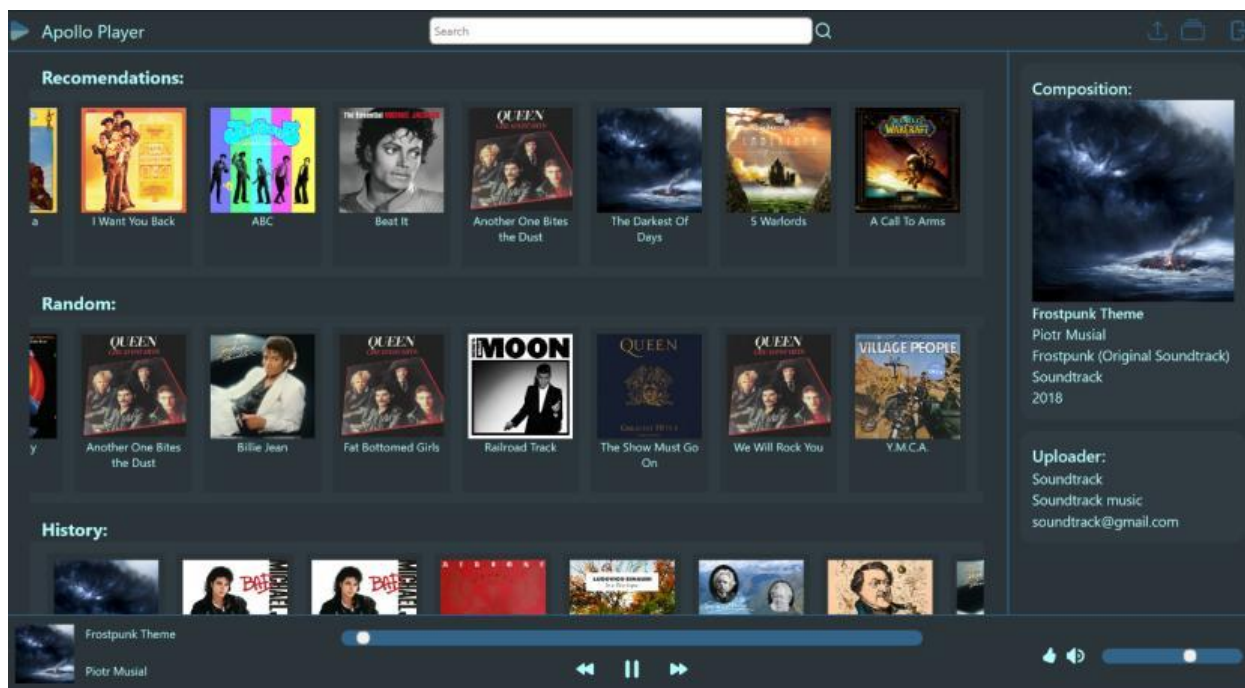


Рисунок 5.10 – Перехід між композиціями

Було протестовано функцію пошуку, ввівши ключове слово в пошуковому рядку.

Відображено список знайдених композицій відповідно до запиту, це можна побачити на рисунку 5.11.

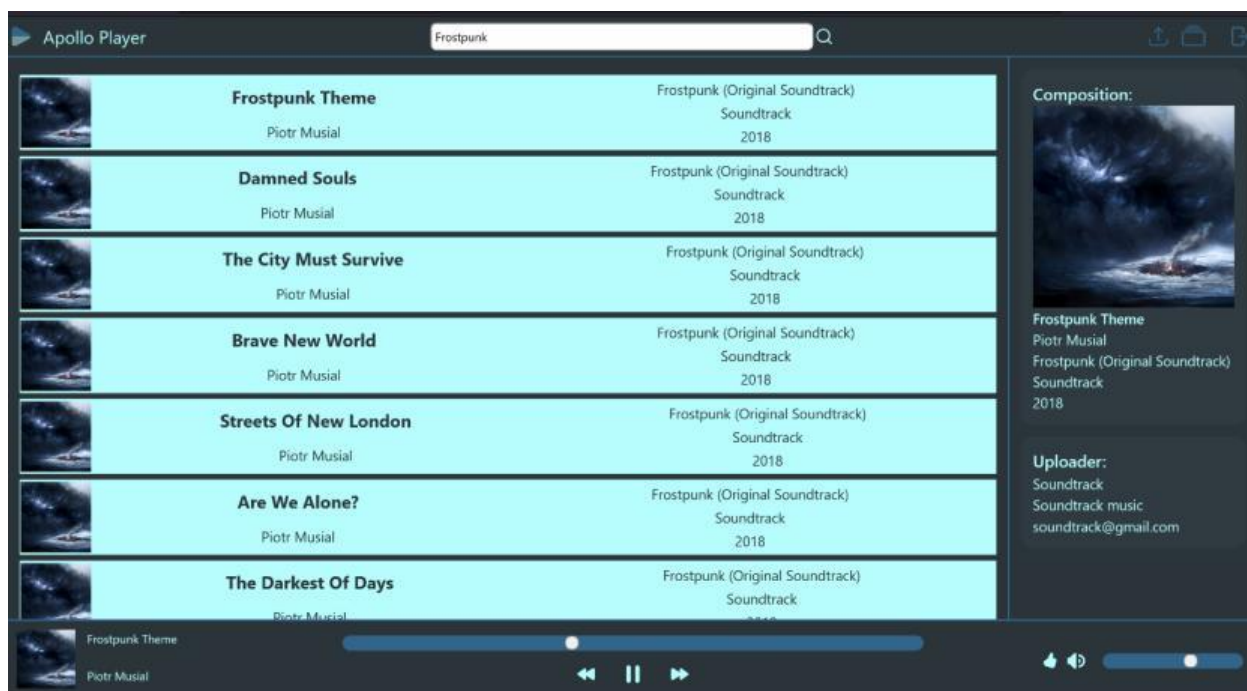
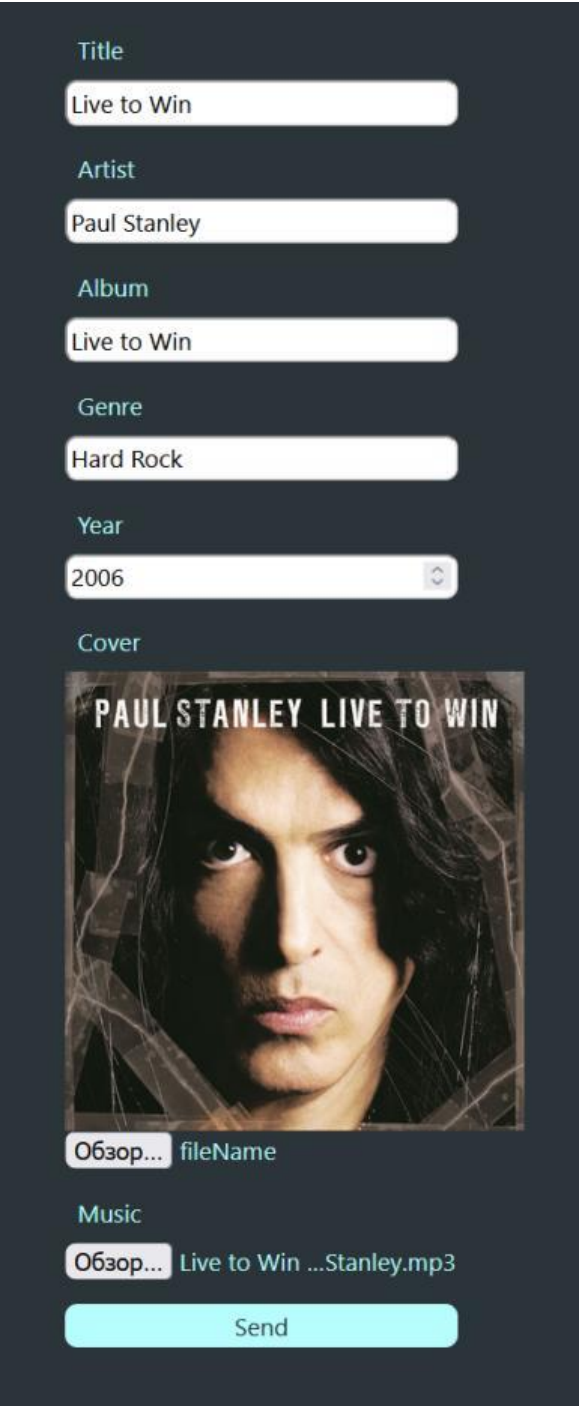


Рисунок 5.11 – Список результатів пошуку композицій

Обрано випадковий файл для завантаження. Теги й обкладинка треку автоматично підтягнулися. Результати можна побачити на рисунку 5.12.



Title

Live to Win

Artist

Paul Stanley

Album

Live to Win

Genre

Hard Rock

Year

2006

Cover

Обзор... fileName

Music

Обзор... Live to Win ...Stanley.mp3

Send

Рисунок 5.12 – Автозаповнення теги з композиції, та готові до відправлення на бекенд

Файл разом із метаданими успішно надісланий на сервер. Результат можна побачити на рисунку 5.13.

Статус	Метод	Домен	Файл	Инициатор	Тип	Передано	Размер	0 мс	80 мс
202	POST	localhost:3000	audio	polyfills.js:1724 (xhr)	json	569 Б	294 Б	17 мс	
204	OPTIONS	localhost:3000	audio	xhr	plain	360 Б	0 Б	6 мс	
	POST	localhost:3000	cover	polyfills.js:1724 (xhr)					
204	OPTIONS	localhost:3000	cover	xhr	plain	338 Б	0 Б	0 мс	

Рисунок 5.13 – Композиція та обгортка відправлені на бекенд

Завантажена композиція відображається в пошуку й коректно відтворюється. Результат можна побачити на рисунку 5.14.

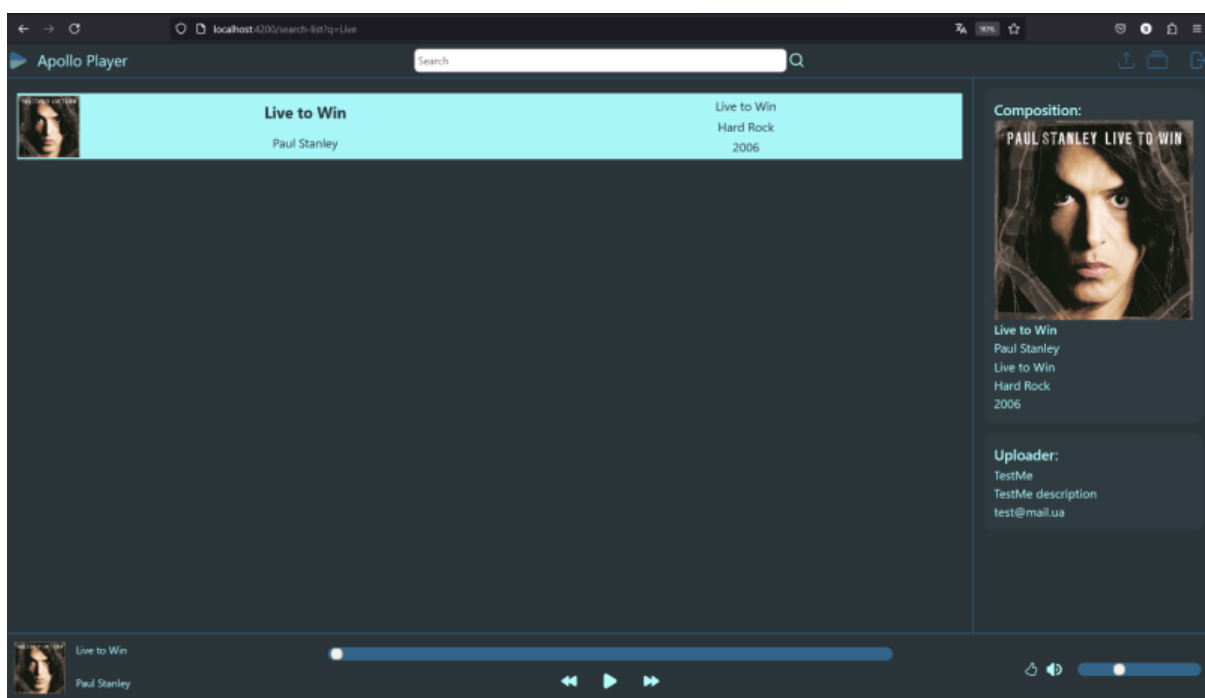


Рисунок 5.14 – Знайдена додана композиція

Композиція успішно зупиняється на паузу. Перемотування треку працює коректно. Результат можна побачити на рисунку 5.15.

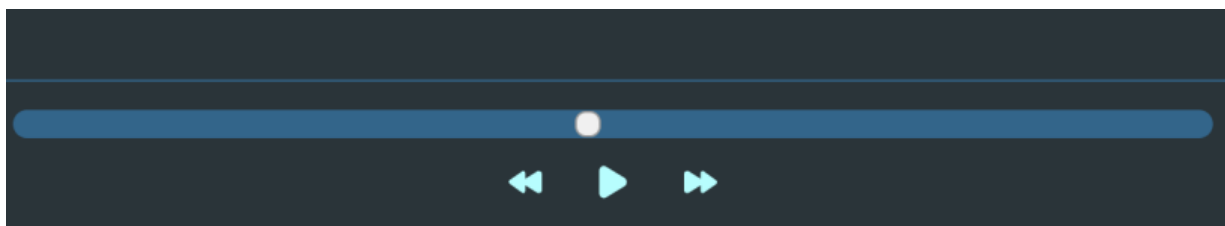


Рисунок 5.15 – Промотування композиції

Користувач успішно вийшов зі свого облікового запису. Також вхід у систему працює без проблем. Результат можна побачити на рисунку 5.16.

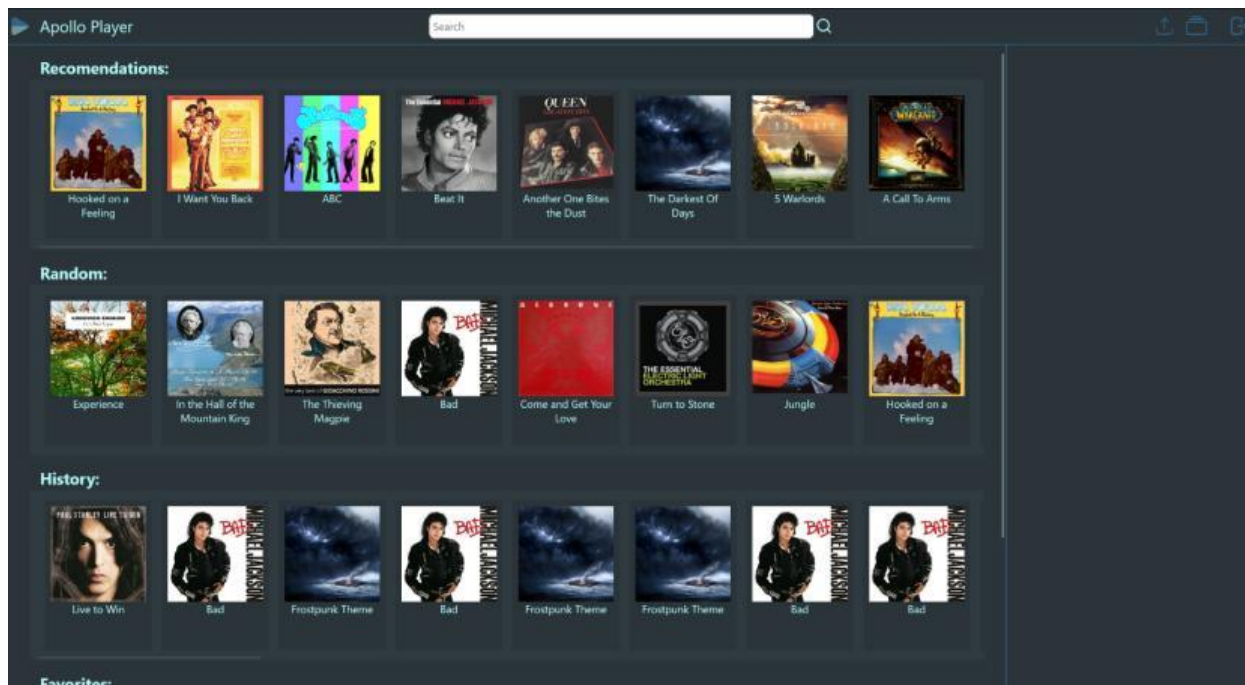


Рисунок 5.16 – Аккаунт відображає композиції для користувача

Ручне тестування показало, що всі основні функції програми працюють коректно. Функціональність клієнтського та серверного застосунків підтверджена на практиці.

ВИСНОВКИ

У процесі розробки програмного забезпечення було досягнуто поставлених цілей, які полягали у створенні багатофункціонального вебзастосунка для прослуховування музики та взаємодії користувачів та дослідження предметної області. У роботі застосовано сучасні підходи до розробки клієнтської та серверної частин застосунка, а також реалізовано алгоритми рекомендацій і управління користувацьким досвідом.

На основі проведеного аналізу було спроєктовано та реалізовано ключові функціональні можливості застосунку.

Користувачі можуть створювати облікові записи, виконувати вхід через електронну пошту.

Реалізовано базовий музичний програвач із підтримкою функцій відтворення, паузи, перемикання треків, регулювання гучності та відновлення налаштувань після перезавантаження сторінки.

Виконується збереження унікальних ідентифікаторів музичних треків для формування історії взаємодії користувача із застосунком.

Алгоритм пошуку схожих музичних вподобань базується на аналізі обраного та прослуханого контенту користувача, а також знаходженні спільних інтересів із іншими користувачами.

Реалізовано можливість додавати музичні треки до списку обраного та взаємодіяти з цим списком.

Реалізовано форму завантаження аудіо з автоматичним аналізом тегів і обкладинок для полегшення роботи користувача.

Створено інтуїтивний пошук із відображенням результатів за ключовими словами.

Розроблений вебзастосунок успішно протестовано в ручному та автоматичному режимі. Проведене тестування продемонструвало стабільну роботу основних модулів. Усі передбачені функції, включно з алгоритмами

рекомендацій та управління користувацьким досвідом, виконуються коректно.

Отримані результати підтверджують доцільність обраних технологій і підходів. Використання Angular для клієнтської частини, Nest.js для серверної частини, а також бази даних MongoDB забезпечило високу продуктивність і масштабованість системи.

Розроблений застосунок може бути основою для подальшого вдосконалення, зокрема інтеграції додаткових функцій, таких як соціальна взаємодія між користувачами, підписки на улюблених авторів або розширені аналітичні інструменти. Впровадження цих функцій дозволить підвищити користувацьку залученість і зробить систему конкурентоспроможною на ринку музичних сервісів.

Таким чином, виконана робота демонструє високий рівень відповідності поставленим цілям та завданням і робить вагомий внесок у розвиток вебтехнологій для індивідуального та колективного прослуховування музики.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. The Rise of Streaming Music and Implications for Music Production [Electronic resource]. – Access mode: https://www.researchgate.net/publication/327887722_The_Rise_of_Streaming_Music_and_Implications_for_Music_Production.
2. Overview [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>.
3. HTTPS [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Glossary/HTTPS>.
4. SSL [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Glossary/SSL>.
5. HTTP and HTTPS [Electronic resource]. – Access mode: <https://gravitec.net/blog/http-and-https/>.
6. HTTP request and response example [Electronic resource]. – Access mode: <https://reqbin.com/req/97vglfpm/http-request-and-response-example>.
7. HTTP Methods [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>.
8. The difference between HTTPS and HTTP [Electronic resource]. – Access mode: <https://aws.amazon.com/ru/compare/the-difference-between-https-and-http/#:~:text=HTTP%20messages%20are%20plaintext%2C%20which,the%20data%20over%20the%20network>.
9. HTTP Headers [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>.
10. Payload body [Electronic resource]. – Access mode: https://developer.mozilla.org/en-US/docs/Glossary/Payload_body.
11. HTTP Status [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.

12. Response header [Electronic resource]. – Access mode:
https://developer.mozilla.org/en-US/docs/Glossary/Response_header.
13. Response body [Electronic resource]. – Access mode:
<https://developer.mozilla.org/en-US/docs/Web/API/Response/body>.
14. RFC6749 [Electronic resource]. – Access mode:
<https://datatracker.ietf.org/doc/html/rfc6749>.
15. User-Agent Header [Electronic resource]. – Access mode:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent>.
16. Accept Header [Electronic resource]. – Access mode:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept>.
17. Cache-Control Header [Electronic resource]. – Access mode:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control>.
18. Cookie Header [Electronic resource]. – Access mode:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie>.
19. Content-Length Header [Electronic resource]. – Access mode:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Length>.
20. HTTP Headers [Electronic resource]. – Access mode:
<https://beeceptor.com/docs/concepts/http-headers/>.
21. MIME Type [Electronic resource]. – Access mode:
https://developer.mozilla.org/en-US/docs/Glossary/MIME_type.
22. Content-Type [Electronic resource]. – Access mode:
<https://beeceptor.com/docs/concepts/content-type/>.
23. Media Formats & MIME Types [Electronic resource]. – Access mode:
<https://pro.europeana.eu/page/media-formats-mime-types>.
24. ID3 [Electronic resource]. – Access mode: <https://id3.org/>.
25. Spotify [Electronic resource]. – Access mode: <https://open.spotify.com/>.
26. Spotify Review [Electronic resource]. – Access mode:
<https://www.pcmag.com/reviews/spotify>.
27. Better in Black: Rethinking our Most Important Buttons | Spotify Design

[Electronic resource]. – Access mode: <https://spotify.design/article/better-in-black-rethinking-our-most-important-buttons>.

28. Spotify Review [Electronic resource]. – Access mode: <https://www.techradar.com/reviews/spotify>.

29. YouTube Music [Electronic resource]. – Access mode: <https://music.youtube.com/>.

30. YouTube Music – New Features Highlights: Oct 2022 [Electronic resource]. – Access mode: <https://support.google.com/youtubemusic/thread/184976203/youtube-music-%E2%80%93-new-features-highlights-oct-2022?hl=en>.

31. YouTube Music Review [Electronic resource]. – Access mode: <https://www.pcmag.com/reviews/youtube-music>.

32. YouTube Music's 2024 Recaps [Electronic resource]. – Access mode: <https://techcrunch.com/2024/11/25/youtube-musics-2024-recaps-are-out-heres-how-to-see-yours/>.

33. Deezer [Electronic resource]. – Access mode: <https://www.deezer.com/ru/>.

34. The Design Crew Studio | Deezer [Electronic resource]. – Access mode: <https://www.thedesigncrew.co/en/studio-cases/deezer>.

35. Deezer Review [Electronic resource]. – Access mode: <https://www.pcmag.com/reviews/deezer>.

36. Deezer Review [Electronic resource]. – Access mode: <https://www.techradar.com/reviews/deezer>.

37. SoundCloud [Electronic resource]. – Access mode: <https://soundcloud.com/>.

38. SoundCloud update test features a Discover feed similar to TikTok - RouteNote Blog [Electronic resource]. – Access mode: <https://routenote.com/blog/soundcloud-update-test-similar-to-tiktok/>.

39. SoundCloud Review [Electronic resource]. – Access mode:

<https://www.pcmag.com/reviews/soundcloud>.

40. SoundCloud Review [Electronic resource]. – Access mode: <https://www.soundguys.com/soundcloud-review-51374/>.

41. I tested all the top streaming services – here is my opinion [Electronic resource]. – Access mode: https://www.reddit.com/r/YoutubeMusic/comments/nmew19/i_tested_all_the_top_streaming_services_here_is/.

42. Top Front-End Frameworks [Electronic resource]. – Access mode: <https://www.geeksforgeeks.org/top-front-end-frameworks/>.

43. Angular [Electronic resource]. – Access mode: <https://angular.dev/>.

44. React [Electronic resource]. – Access mode: <https://react.dev/learn>.

45. Vue.js Introduction [Electronic resource]. – Access mode: <https://vuejs.org/guide/introduction.html>.

46. Front-End Frameworks Roadmap [Electronic resource]. – Access mode: <https://roadmap.sh/frontend/frameworks>.

47. Express.js [Electronic resource]. – Access mode: <https://expressjs.com/>.

48. NestJS Documentation [Electronic resource]. – Access mode: <https://docs.nestjs.com/>.

49. NestJS vs. Express.js [Electronic resource]. – Access mode: <https://blog.logrocket.com/nestjs-vs-express-js/>.

50. MongoDB vs. MySQL vs. SQLite Comparison [Electronic resource]. – Access mode: <https://sourceforge.net/software/compare/MongoDB-vs-MySQL-vs-SQLite/>.

51. MongoDB [Electronic resource]. – Access mode: <https://www.mongodb.com/>.

52. SQLite [Electronic resource]. – Access mode: <https://www.sqlite.org/index.html>.

53. MySQL [Electronic resource]. – Access mode: <https://www.mysql.com/>.

54. DB-Engines Ranking [Electronic resource]. – Access mode: <https://db->

engines.com/en/system/MongoDB%3BMySQL%3BSQLite.

55. Complete Angular Performance Guide [Electronic resource]. – Access mode: <https://danielk.tech/home/complete-angular-performance-guide>.

56. OnPush Change Detection in Angular [Electronic resource]. – Access mode: <https://blog.angular-university.io/onpush-change-detection-how-it-works/>.

57. Angular Async Pipe [Electronic resource]. – Access mode: <https://malcoded.com/posts/angular-async-pipe/>.

58. Angular Standalone Migrations [Electronic resource]. – Access mode: <https://angular.dev/reference/migrations/standalone>.

59. Lazy Loading in Angular [Electronic resource]. – Access mode: <https://www.freecodecamp.org/news/lazy-loading-in-angular-intro-to-ngmodules/>.

60. Angular Optimization: Memoized Pipe Functions in Templates [Electronic resource]. – Access mode: <https://medium.com/@ineedsomemeat/angular-optimization-memoized-pipe-functions-in-templates-75f62e16df5a>.

61. ngx-translate [Electronic resource]. – Access mode: <https://github.com/ngx-translate>.

62. Angular Testing Guide [Electronic resource]. – Access mode: <https://v17.angular.io/guide/testing>.

ДОДАТОК А
Технічне завдання

Вступ

З розвитком цифрових технологій та популярністю стрімінгових платформ музика стала доступною для мільйонів користувачів. Однак виникає потреба у вебзастосунках, що забезпечують ефективне управління аудіоконтентом та створення персоналізованих плейлистів. У рамках цього проєкту розглядається розробка такого застосунку із використанням Angular для клієнтської та Nest.js для серверної частини, з акцентом на зручний інтерфейс, надійність і швидкодію.

A.1 Підстава для розробки

Підставою для розробки є завдання на дипломну кваліфікаційну роботу на тему «Дослідження та програмна реалізація вебзастосунку зберігання та програвання музики з застосуванням клієнт-серверної архітектури. Research and software implementation of a web application for storing and playing music using a client-server architecture», затверджене наказом Національного університету «Запорізька політехніка» № 398 від 07 жовтня 2024 р.

A.2 Призначення розробки

Розробка спрямована на створення багатофункціонального вебзастосунку для завантаження, пошуку, збереження та прослуховування музики. Система повина забезпечувати персоналізовані рекомендації музичного контенту, базуючись на вподобаннях користувачів та їхній активності. Важливими аспектами є масштабованість, висока продуктивність та зручність використання.

А.3 Основні вимоги до програми, що розробляється

А.3.1 Вимоги до функціональних характеристик

Головні функціональні вимоги включають підтримку реєстрації та авторизації користувачів через електронну пошту, базовий музичний програвач з ключовими функціями відтворення та управління треками, формування історії прослуховувань і списків обраного. Повинно бути реалізовано можливість завантаження аудіо з автоматичним підтягування метаданих, а також алгоритми пошуку й рекомендацій, засновані на спільних інтересах користувачів.

А.3.2 Вимоги до інтерфейсу програми

Програма повинна мати інтуїтивно зрозумілий і сучасний інтерфейс, який забезпечує швидкий доступ до основних функцій. Дизайн орієнтований на простоту навігації, адаптивність під різні розміри екранів і відповідність загальним стандартам UX/UI.

А.3.3 Вимоги до надійності

Програма повинна забезпечувати коректну роботу ключових функцій навіть за значного навантаження. Надійність гарантується використанням сучасних серверних технологій та систем тестування, які запобігають критичним збоям і забезпечують стабільність роботи.

А.3.4 Умови експлуатації

Для експлуатації програмного забезпечення потрібна наявність стабільного інтернет-з'єднання та браузера з підтримкою сучасних веб-

технологій. Серверна частина програми повинна функціонувати на потужностях, здатних обробляти запити від великої кількості користувачів.

A.3.5 Вимоги до складу та параметрів технічний засобів

Для експлуатації програмного забезпечення для хостингу вебзастосунку на сервері необхідно таке апаратне та програмне забезпечення:

- процесор – 1 ГГц з 4 або більше ядрами;
- оперативна пам'ять – 2 ГБ;
- HDD або SSD – 25 ГБ або більше;
- встановлене Node.JS.

Для експлуатації програмного забезпечення для кінцевого користувача необхідно таке апаратне та програмне забезпечення:

- процесор – 1 ГГц або швидше з 2 або більше ядрами;
- оперативна пам'ять – 4 ГБ;
- HDD або SSD – 64 ГБ або більше;
- дисплей – 720p, діагональ якого перевищує 9 дюймів, 8 біт на колірний канал;
- встановлений браузер, такий як Chrome, Edge, Firefox, Opera або Safari.

A.3.6 Вимоги до маркування і пакування

Маркування програми має включати її назву та версію. Пакування програмного забезпечення передбачає необхідно підготувати оптимізовані пакети для клієнтської та серверної частини. Також необхідно забезпечити наявність документації користувача та короткого опису функціональних можливостей для ознайомлення перед встановленням чи використанням.

ДОДАТОК Б
Текст програми

B.1 apollo-client\src\app\auth\auth.component.ts

```
export class AuthComponent {
  private readonly fb = inject(NonNullableFormBuilder);
  private readonly accountService = inject(AccountService);
  public isAccountAlreadyExist: boolean = true;
  public form: FormGroup<AuthForm> = this.fb.group<AuthForm>({
    {
      email: this.fb.control('', [Validators.required, Validators.email]),
      password: this.fb.control('', [Validators.required]),
      name: this.fb.control('', [Validators.required]),
      description: this.fb.control('', [Validators.required]),
    },
    {}
  });
  public submit() {
    this.checkDisableForm();
    if (this.form.valid) {
      if (this.isAccountAlreadyExist) {
        this.accountService.login({
          email: this.form.controls.email.value,
          password: this.form.controls.password.value,
        });
      } else {
        this.accountService.signup({
          email: this.form.controls.email.value,
          password: this.form.controls.password.value,
          name: this.form.controls.name.value,
          description: this.form.controls.description.value,
        });
      }
    } else {
      this.form.markAllAsTouched();
    }
  }
  public checkDisableForm() {
    if (this.isAccountAlreadyExist) {
      this.form.controls.description.disable();
      this.form.controls.name.disable();
    } else {
      this.form.controls.description.enable();
      this.form.controls.name.enable();
    }
  }
}
```

B.2 apollo-client\src\app\proposals-list\proposals-list.component.ts

```
export class ProposalsListComponent {
  private readonly http: HttpClient = inject(HttpClient);
  private readonly audioService: AudioService = inject(AudioService);
  public readonly musicList$ = this.http.get<Music[]>(API_MUSIC_LIST);
  public readonly historyList$ = this.http.get<Music[]>(API_HISTORY_LIST);
  public readonly recomendationsList$ = this.http.get<Music[]>(
    API_MUSIC_RECOMENDATIONS
  );
}
```

```

    );
    public readonly favoriteList$ = this.audioService.currentLiked$.pipe(
      switchMap(() => this.http.get<Music[]>(API_FAVORITE_LIST))
    );
  }

```

Б.3 apollo-client\src\app\search-list\search-list.component.ts

```

export class SearchListComponent {
  private readonly route = inject(ActivatedRoute);
  private readonly http = inject(HttpClient);
  public readonly searchList$ = this.route.queryParams.pipe(
    switchMap(queryParams =>
      this.http.get<Music[]>(API_MUSIC_SEARCH, {
        params: { searchString: queryParams['q'] },
      })
    )
  );
}

```

Б.4 apollo-client\src\app\shared\interceptors\session.interceptor.ts

```

@Injectable()
export class SessionTokenInterceptor implements HttpInterceptor {
  private readonly accountService: AccountService = inject(AccountService);
  public intercept(
    request: HttpRequest<unknown>,
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {
    request = request.clone({
      headers: request.headers.set(
        'session-token',
        this.accountService.session$.value?.sessionToken ?? ''
      ),
    });
    return next.handle(request);
  }
}

```

Б.5 apollo-client\src\app\shared\services\account.service.spec.ts

```

describe('AccountService', () => {
  let service: AccountService;
  let httpMock: HttpTestingController;
  const mockSession: Session = {
    dueDate: new Date().toISOString(),
    sessionToken: 'sessionToken',
    createDate: new Date().toISOString(),
  };
  const mockAccount: AccountWithCreds = {
    email: 'test@mail.com',
    password: 'test',
    name: 'test',
    description: 'description',
  };
};

```

```

const mockLoginDto: LoginDto = {
  email: 'test@mail.com',
  password: 'test',
};
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    providers: [AccountService],
  });

  service = TestBed.inject(AccountService);
  httpMock = TestBed.inject(HttpTestingController);
  localStorage.clear();
});
afterEach(() => {
  httpMock.verify();
});
it('should sign up and update session and account', () => {
  service.signup(mockAccount);
  const req = httpMock.expectOne(API_ACCOUNT);
  expect(req.request.method).toBe('POST');
  req.flush(mockSession);
  expect(service.session$.value).toEqual(mockSession);
  expect(service.account$.value).toEqual(mockAccount);
  expect(localStorage.getItem('session')).toEqual(
    JSON.stringify(mockSession)
  );
  expect(localStorage.getItem('account')).toEqual(
    JSON.stringify(mockAccount)
  );
});
it('should log in and fetch account', () => {
  service.login(mockLoginDto);
  const loginReq = httpMock.expectOne(API_ACCOUNT_LOGIN);
  expect(loginReq.request.method).toBe('POST');
  loginReq.flush(mockSession);
  expect(service.session$.value).toEqual(mockSession);
  const accountReq = httpMock.expectOne(API_ACCOUNT);
  expect(accountReq.request.method).toBe('GET');
  accountReq.flush(mockAccount);
  expect(service.account$.value).toEqual(mockAccount);
  expect(localStorage.getItem('session')).toEqual(
    JSON.stringify(mockSession)
  );
  expect(localStorage.getItem('account')).toEqual(
    JSON.stringify(mockAccount)
  );
});
it('should log out and clear session and account', () => {
  service.logout();
  const req = httpMock.expectOne(API_ACCOUNT_LOGOUT);
  expect(req.request.method).toBe('HEAD');
  req.flush(null);
  expect(service.account$.value).toBeNull();
  expect(service.session$.value).toBeNull();
  expect(localStorage.getItem('session')).toBeNull();
  expect(localStorage.getItem('account')).toBeNull();
});
});

```

B.6 apollo-client\src\app\shared\services\account.service.ts

```
export class AccountService {
  private readonly http: HttpClient = inject(HttpClient);
  public readonly account$: BehaviorSubject<AccountWithCreds | null> =
    new BehaviorSubject<AccountWithCreds | null>(null);
  public readonly session$: BehaviorSubject<Session | null> =
    new BehaviorSubject<Session | null>(null);
  public constructor() {
    this.account$.next(JSON.parse(localStorage.getItem('account')));
    this.session$.next(JSON.parse(localStorage.getItem('session')));
    if (
      this.session$.value?.dueDate &&
      new Date(this.session$.value?.dueDate) < new Date()
    ) {
    }
    this.account$.subscribe(account => {
      localStorage.setItem('account', JSON.stringify(account));
      if (account === null) {
        localStorage.removeItem('account');
      }
    });
    this.session$.subscribe(session => {
      localStorage.setItem('session', JSON.stringify(session));
      if (session === null) {
        localStorage.removeItem('session');
      }
    });
  }
  public signup(account: AccountWithCreds): void {
    this.http.post<Session>(API_ACCOUNT, account).subscribe(session => {
      this.session$.next(session);
      this.account$.next(account);
    });
  }
  public login(loginDto: LoginDto): void {
    this.http
      .post<Session>(API_ACCOUNT_LOGIN, loginDto)
      .pipe(
        switchMap(session => {
          this.session$.next(session);
          return this.http.get<AccountWithCreds>(API_ACCOUNT);
        })
      )
      .subscribe(account => this.account$.next(account));
  }
  public logout(): void {
    this.account$.next(null);
    this.session$.next(null);
    this.http.head<Session>(API_ACCOUNT_LOGOUT).subscribe();
  }
}
```

B.7 apollo-client\src\app\shared\services\audio.service.spec.ts

```
describe('AudioService', () => {
  let service: AudioService;
  let httpMock: HttpTestingController;
```

```

beforeEach(() => {
  TestBed.configureTestingModule({
    providers: [
      AudioService,
      provideHttpClient(),
      provideHttpClientTesting(),
    ],
  });
  service = TestBed.inject(AudioService);
  httpMock = TestBed.inject(HttpTestingController);
  spyOn(service.audioObject, 'play').and.callFake(async () => {});
  spyOn(service.audioObject, 'pause').and.callFake(async () => {});
});
afterEach(() => {
  httpMock.verify();
});
it('should subscribe to currentAudio$ and update audio source', () => {
  const mockMusic: Music = { _id: '1', title: 'Test Music' } as Music;
  service.currentAudio$.next(mockMusic);
  expect(service.audioObject.src).toBe(`${API_MUSIC_STREAM}/1`);
  const reqHistory = httpMock.expectOne(API_HISTORY);
  expect(reqHistory.request.method).toBe('POST');
  reqHistory.flush({});
  const reqFavorite = httpMock.expectOne(`${API_FAVORITE}?musicId=1`);
  expect(reqFavorite.request.method).toBe('GET');
  reqFavorite.flush(true);
  expect(service.currentLiked$.value).toBe(true);
});
it('should play and pause the audio', () => {
  service.playPauseMusic();
  expect(service.audioObject.play).toHaveBeenCalled();
  service.audioObject.pause();
  service.playPauseMusic();
  expect(service.audioObject.pause).toHaveBeenCalled();
});
it('should set audio currentTime', () => {
  service.setTime(30);
  expect(service.audioObject.currentTime).toBe(30);
});
it('should set volume and save it to localStorage', () => {
  spyOn(localStorage, 'setItem');
  service.setVolume(0.7);
  expect(service.audioObject.volume).toBe(0.7);
  expect(localStorage.setItem).toHaveBeenCalledWith('volume', '0.7');
});
it('should select music and update query$', () => {
  const mockQuery: Music[] = [
    { _id: '1', title: 'Song 1' } as Music,
    { _id: '2', title: 'Song 2' } as Music,
  ];
  service.selectMusic(mockQuery);
  expect(service.query$.value).toEqual(mockQuery);
  expect(service.currentAudio$.value).toEqual(mockQuery[0]);
  const reqHistory = httpMock.expectOne(API_HISTORY);
  expect(reqHistory.request.method).toBe('POST');
  reqHistory.flush({});
  const reqFavorite = httpMock.expectOne(`${API_FAVORITE}?musicId=1`);
  expect(reqFavorite.request.method).toBe('GET');
  reqFavorite.flush(true);
});
it('should go to next music', () => {

```

```

const mockQuery: Music[] = [
  { _id: '1', title: 'Song 1' } as Music,
  { _id: '2', title: 'Song 2' } as Music,
];
service.selectMusic(mockQuery);
service.nextMusic();
expect(service.currentAudio$.value).toEqual(mockQuery[1]);
const historyRequests = httpMock.match(API_HISTORY);
expect(historyRequests.length).toBeGreaterThan(0);
historyRequests.forEach(req => req.flush({}));
const favoriteRequests = httpMock.match(req =>
  req.url.includes(API_FAVORITE)
);
expect(favoriteRequests.length).toBeGreaterThan(0);
favoriteRequests.forEach(req => req.flush(true));
});
});

```

B.8 apollo-client\src\app\shared\services\audio.service.ts

```

export class AudioService {
  private readonly http: HttpClient = inject(HttpClient);
  public readonly query$: BehaviorSubject<Music[]> = new BehaviorSubject<
    Music[]
  >([]);
  public readonly currentAudio$: BehaviorSubject<Music | null> =
    new BehaviorSubject<Music | null>(null);
  public readonly currentLiked$: BehaviorSubject<boolean> =
    new BehaviorSubject<boolean>(false);
  public readonly audioEvents =
    ['ended', 'error', 'play', 'playing', 'pause', 'timeupdate', 'canplay', 'loadedmetad
    ata', 'loadstart', ];
  public audioObject = new Audio();
  public readonly stream$ = new Observable(observer => {
    const handler = (event: Event) => {
      observer.next(event);
    };
    this.addEvents(this.audioObject, this.audioEvents, handler);
    return () => {
      this.audioObject.pause();
      this.audioObject.currentTime = 0;
      this.removeEvents(this.audioObject, this.audioEvents, handler);
    };
  });
  constructor() {
    const volume = window.localStorage.getItem('volume');
    if (volume) {
      this.audioObject.volume = +volume;
    }
    this.currentAudio$.subscribe(audio => {
      if (audio) {
        this.audioObject.src = API_MUSIC_STREAM + '/' + audio._id;
        this.audioObject.load();
        this.audioObject.play();
        this.http
          .post(API_HISTORY, {
            musicId: audio._id,
          })
          .subscribe();
      }
    });
  }
}

```

```

        this.http
            .get<boolean>(API_FAVORITE, {
                params: { musicId: audio?._id ?? '' },
            })
            .subscribe(liked => {
                this.currentLiked$.next(liked);
            });
    }
    });
    this.stream$.subscribe(() => {
        if (this.audioObject.currentTime === this.audioObject.duration) {
            this.nextMusic();
        }
    });
}

    public readonly streamDuration$ = this.stream$.pipe(map(() =>
this.audioObject.duration));
    public readonly streamCurrentTime$ = this.stream$.pipe(map(() =>
this.audioObject.currentTime));
    public readonly streamVolume$ = this.stream$.pipe(map(() =>
this.audioObject.volume * 100));
    public readonly streamPaused$ = this.stream$.pipe(
        map(() => this.audioObject.paused)
    );
    private addEvents(obj: any, events: any, handler: any) {
        events.forEach((event: any) => {
            obj.addEventListener(event, handler);
        });
    }
    private removeEvents(obj: any, events: any, handler: any) {
        events.forEach((event: any) => {
            obj.removeEventListener(event, handler);
        });
    }
    }
    public playPauseMusic(): void {
        if (this.audioObject.paused) {
            this.audioObject.play();
        } else {
            this.audioObject.pause();
        }
    }
    }
    public setTime(time: number) {
        this.audioObject.currentTime = time;
    }
    }
    public setVolume(volume: number) {
        this.audioObject.volume = volume;
        localStorage.setItem('volume', '' + this.audioObject.volume);
    }
    }
    public selectMusic(query: Music[]): void {
        this.currentAudio$.next(query[0]);
        this.query$.next(query);
    }
    }
    public nextMusic(): void {
        const currentIndex = this.query$.value.findIndex(
            audio => audio === this.currentAudio$.value
        );
        if (this.query$.value.length - 1 >= currentIndex) {
            this.currentAudio$.next(this.query$.value[currentIndex + 1]);
        }
    }
    }

```

```

public prevMusic(): void {
  const currentIndex = this.query$.value.findIndex(
    audio => audio === this.currentAudio$.value
  );
  if (currentIndex !== 0) {
    this.currentAudio$.next(this.query$.value[currentIndex - 1]);
  }
}
public like(): void {
  if (this.currentLiked$.value) {
    this.http
      .delete(API_FAVORITE, {
        body: { musicId: this.currentAudio$.value?._id },
      })
      .subscribe(() => this.currentLiked$.next(false));
  } else {
    this.http
      .post(API_FAVORITE, { musicId: this.currentAudio$.value?._id })
      .subscribe(() => this.currentLiked$.next(true));
  }
}
}

```

B.9 apollo-client\src\app\upload-music\upload-music.component.ts

```

export class UploadMusicComponent {
  private readonly http: HttpClient = inject(HttpClient);
  private readonly fb: NonNullableFormBuilder =
    inject(NonNullableFormBuilder);
  private readonly cdr: ChangeDetectorRef = inject(ChangeDetectorRef);
  public audioBlob$: BehaviorSubject<any> = new BehaviorSubject('');
  public audio$: BehaviorSubject<any> = new BehaviorSubject('');
  public imgBlob$: BehaviorSubject<any> = new BehaviorSubject('');
  public imgFormat$: BehaviorSubject<any> = new BehaviorSubject('');
  public img$: BehaviorSubject<any> = new BehaviorSubject('');
  @ViewChild('imgFile') imgFile!: ElementRef;
  public readonly form = this.fb.group({
    title: this.fb.control('', Validators.required),
    artist: this.fb.control('', Validators.required),
    album: this.fb.control('', Validators.required),
    genre: this.fb.control('', Validators.required),
    year: this.fb.control(2000, Validators.required),
  });
  public async loadAudio($event: any) {
    const reader = new FileReader();
    reader.onload = e => {
      this.audio$.next(e.target!.result);
    };
    for (const file of $event.target.files) reader.readAsArrayBuffer(file);
    const blobAudio = new Blob($event.target.files, { type: 'audio/mpeg' });
    const metadata = await parseBlob(blobAudio);
    this.form.controls.title.setValue(metadata.common.title ?? '');
    this.form.controls.artist.setValue(metadata.common.artist ?? '');
    this.form.controls.album.setValue(metadata.common.album ?? '');
    this.form.controls.genre.setValue((metadata.common.genre ?? [''])[0]);
    this.form.controls.year.setValue(metadata.common.year ?? 2000);
    const picture = selectCover(metadata.common.picture);
    if (picture) {
      const blobCover = new Blob([picture.data], {

```

```

        type: picture.format,
    });
    this.imgBlob$.next(URL.createObjectURL(blobCover));
    let file = new File([blobCover], 'fileName', {
        type: picture.format,
        lastModified: new Date().getTime(),
    });
    let container = new DataTransfer();
    container.items.add(file);
    this.imgFile.nativeElement.files = container.files;
    this.img$.next(picture.data);
    this.imgFormat$.next(picture.format);
}
}
public loadImg($event: any) {
    const readerArrayBuffer = new FileReader();
    readerArrayBuffer.onload = e => {
        this.img$.next(e.target!.result);
    };
    for (const file of $event.target.files) {
        readerArrayBuffer.readAsArrayBuffer(file);
    }
    const readerBlob = new FileReader();
    readerBlob.onload = e => {
        const dataUrl = e.target!.result as string;
        this.imgFormat$.next(
            dataUrl.substring(dataUrl.indexOf(':') + 1, dataUrl.indexOf(';'))
        );
        this.imgBlob$.next(dataUrl);
    };
    for (const file of $event.target.files) {
        readerBlob.readAsDataURL(file);
    }
}
public async send() {
    if (this.form.valid) {
        const blobAudio = new Blob([this.audio$.value], { type: 'audio/mpeg'
    });
    const dataAudio = new FormData();
    dataAudio.append('file', blobAudio, this.form.controls.title.value);
    const blobCover = new Blob([this.img$.value], {
        type: this.imgFormat$.value,
    });
    const dataImg = new FormData();
    dataImg.append('file', blobCover, this.form.controls.title.value);
    this.http
        .post<Music>(API_MUSIC_UPLOAD_AUDIO, dataAudio, {
            headers: {
                title: this.form.controls.title.value,
                artist: this.form.controls.artist.value,
                album: this.form.controls.album.value,
                genre: this.form.controls.genre.value,
                year: '' + this.form.controls.year.value,
            },
        })
        .pipe(
            switchMap((music: Music) => {
                return this.http.post(API_MUSIC_UPLOAD_COVER, dataImg, {
                    headers: { musicId: music._id },
                });
            })
        )
    }
}

```

```

        )
        .subscribe(console.log);
    } else {
        this.form.markAllAsTouched();
    }
}
}
}

```

B.10 apollo-client\src\app\app.routes.ts

```

export const routes: Routes = [
  {
    path: '',
    redirectTo: 'music-list',
    pathMatch: 'full',
  },
  {
    path: 'upload',
    loadComponent: () => import('./upload-music/upload-music.component').then(c => c.UploadMusicComponent), },
  {
    path: 'music-list',
    loadComponent: () => import('./proposals-list/proposals-list.component').then(c => c.ProposalsListComponent),
  },
  {
    path: 'search-list',
    loadComponent: () => import('./search-list/search-list.component').then(c => c.SearchListComponent),
  },
];

```

B.11 apollo-server\src\account\account.service.ts

```

@Injectable()
export class AccountService {
  constructor(
    @InjectModel(Account.name) private readonly accountModel: Model<Account>,
    @InjectModel(Session.name) private readonly sessionModel: Model<Session>,
    private readonly sessionService: SessionService,
  ) {}

  async create(createAccountDto: CreateAccountDto): Promise<Session> {
    const account = await new this.accountModel(createAccountDto).save();
    return this.sessionService.createSession(account);
  }

  async change(
    sessionToken: string,
    changeAccountDto: Partial<CreateAccountDto>,
  ): Promise<boolean> {
    const account: Account =
      await this.sessionService.getAccountBySession(sessionToken);
    return (await this.accountModel.updateOne(account, changeAccountDto))
      .acknowledged;
  }

  public async getAccountById(id: string): Promise<Account> {
    const account: Account = await this.accountModel.findById(id).exec();
  }
}

```

```

    return account;
  }
  async login(loginAccountDto: LoginAccountDto): Promise<Session> {
    const account = await this.accountModel.findOne({
      email: loginAccountDto.email,
      password: loginAccountDto.password,
    });
    return this.sessionService.createSession(account);
  }
  async logout(sessionToken: string): Promise<boolean> {
    return (await this.sessionModel.deleteOne({ sessionToken }).exec())
      .acknowledged;
  }
}

```

B.12 apollo-server\src\music\music.controller.ts

```

@Controller('music')
export class MusicController {
  constructor(private readonly musicService: MusicService) {}
  @Post('upload/audio')
  @UseGuards(AuthGuard)
  @UseInterceptors(
    FileInterceptor('file', {
      dest: './music',
      fileFilter: (req: any, file: any, cb: any) => {
        console.log(file.mimetype);
        if (file.mimetype.match(/audio/)) {
          cb(null, true);
        } else {
          console.log('else');
          cb(
            new HttpException(
              `Unsupported file type ${extname(file.originalname)}`,
              HttpStatus.BAD_REQUEST,
            ),
            false,
          );
        }
      },
    }),
  )
  async uploadMusic(
    @Headers('session-token') sessionToken: string,
    @Headers('title') title: string,
    @Headers('artist') artist: string,
    @Headers('album') album: string,
    @Headers('genre') genre: string,
    @Headers('year') year: number,
    @UploadedFile() file: Express.Multer.File,
    @Res() res: Response,
  ) {
    try {
      const music = await
this.musicService.createMusic(sessionToken,title,artist,album,genre,year,file
.size,file.originalname,file.filename,file.mimetype);
      res.status(HttpStatus.ACCEPTED).json(music);
    } catch (e) {
      res.status(HttpStatus.BAD_REQUEST).json(e);
    }
  }
}

```

```

    }
  }

  @Post('upload/cover')
  @UseGuards(AuthGuard)
  @UseInterceptors(
    FileInterceptor('file', {
      dest: './cover',
      fileFilter: (req: any, file: any, cb: any) => {
        if (file.mimetype.match(/image/)) {
          cb(null, true);
        } else {
          cb(
            new HttpException(
              `Unsupported file type ${extname(file.originalname)}`,
              HttpStatus.BAD_REQUEST,
            ),
            false,
          );
        }
      },
    }),
  )
  async addCover(
    @Headers('session-token') sessionToken: string,
    @Headers('musicId') musicId: string,
    @UploadedFile() file: Express.Multer.File,
    @Res() res: Response,
  ) {
    try {
      await this.musicService.addCover(sessionToken, musicId, file);
      res.status(HttpStatus.ACCEPTED);
    } catch (e) {
      res.status(HttpStatus.BAD_REQUEST).json(e);
    }
  }

  @Get('/stream/:musicId')
  public async getFile(
    @Res({ passthrough: true }) res: Response,
    @Param('musicId') musicId: string,
  ): Promise<StreamableFile> {
    const music = await this.musicService.getMusic(musicId);
    const file = createReadStream(
      join(process.cwd(), './music/' + music.filenameInSystem),
    );
    res.set({
      'Content-Type': 'application/octet-stream',
      'Content-Disposition': `filename="${music.filename}"`,
    });
    return new StreamableFile(file);
  }

  @Get('/cover/:musicId')
  public async getCoverFile(
    @Res({ passthrough: true }) res: Response,
    @Param('musicId') musicId: string,
  ): Promise<StreamableFile> {
    try {
      const music = await this.musicService.getMusic(musicId);
      const file = createReadStream(
        join(process.cwd(), './cover/' + music.coverFile),
      );
    }
  }

```

```

        res.set({
            'Content-Type': music.coverFileType,
            'Content-Disposition': `filename="${music.coverFileName}"`,
        });
        return new StreamableFile(file);
    } catch (e) {
        res.status(HttpStatus.BAD_REQUEST).json(e);
    }
}

@Get('list')
@UseGuards(AuthGuard)
@UseGuards(AuthGuard)
public async getList(@Res() res: Response): Promise<void> {
    try {
        const list = await this.musicService.getList();
        res.status(HttpStatus.ACCEPTED).json(list);
    } catch (e) {
        res.status(HttpStatus.BAD_REQUEST).json(e);
    }
}

@Get('similar-recomendations-list')
@UseGuards(AuthGuard)
public async getListRecomendations(
    @Headers('session-token') sessionToken: string,
    @Res() res: Response,
): Promise<void> {
    try {
        const list = await this.musicService.searchSimilar(sessionToken);
        res.status(HttpStatus.ACCEPTED).json(list);
    } catch (e) {
        res.status(HttpStatus.BAD_REQUEST).json(e);
    }
}

@Get('search')
public async getSearchList(
    @Res() res: Response,
    @Query('searchString') searchString: string,
): Promise<void> {
    try {
        const list = await this.musicService.searchList(searchString);
        res.status(HttpStatus.ACCEPTED).json(list);
    } catch (e) {
        res.status(HttpStatus.BAD_REQUEST).json(e);
    }
}
}

```

B.13 apollo-server\src\music\music.service.ts

```

export class MusicService {
    constructor(
        @InjectModel(Favorite.name) private readonly favoriteModel:
        Model<Favorite>,
        @InjectModel(History.name) private readonly historyModel: Model<History>,
        @InjectModel(Music.name) private readonly musicModel: Model<Music>,
        private readonly sessionService: SessionService,
    ) {}
    public async createMusic(
        sessionToken: string,

```

```

    title: string,
    artist: string,
    album: string,
    genre: string,
    year: number,
    size: number,
    filename: string,
    filenameInSystem: string,
    type: string,
  ): Promise<Music> {
    const account = await
this.sessionService.getAccountBySession(sessionToken);
    console.log(account);

    return new
this.musicModel({title,artist,album,genre,year,type,size,filename,filenameInS
ystem, uploader: account._id, }).save();
  }
  public async addCover(
    sessionToken: string,
    musicId: string,
    file: Express.Multer.File,
  ): Promise<void> {
    const account = await
this.sessionService.getAccountBySession(sessionToken);
    await this.musicModel
      .findOneAndUpdate(
        {
          uploader: account._id,
          _id: musicId,
        },
        {
          coverFile: file.filename,
          coverFileName: file.originalname,
          coverFileType: file.mimetype,
        },
      )
      .exec();
  }
  public async getMusic(id: string): Promise<Music> {
    return this.musicModel.findById(id).exec();
  }
  public async getList(): Promise<Music[]> {
    return this.musicModel.find({}, {}, { limit: 50 }).exec();
  }
  public async searchList(searchString: string): Promise<Music[]> {
    return await this.musicModel
      .find({ $text: { $search: searchString } })
      .exec();
  }
  public async searchSimilar(sessionToken: string) {
    const account = await
this.sessionService.getAccountBySession(sessionToken);
    const lastHistories = await this.historyModel
      .find({ account: account._id }, {}, { limit: 50, sort: { date: -1 } })
      .exec();
    const uniqueMusicIds = [
      ...new Set(lastHistories.map((history) => history.music.toString())),
    ];
    const favoriteMusicIds = await this.favoriteModel
      .find({ music: { $in: uniqueMusicIds } })
      .exec();
  }

```

```

const favoriteMusicSet = new Set(
  favoriteMusicIds.map((favorite) => favorite.music.toString()),
);
const accountWithSimilar = new Map<string, number>();
for (const musicId of favoriteMusicSet) {
  let accountIds = (
    await this.favoriteModel.find({ music: musicId }).exec()
  ).map((account) => account.account.toString());
  accountIds = accountIds.filter(
    (accountId) => accountId !== account._id.toString(),
  );
  for (let i = 0; i < accountIds.length; i++) {
    const history = await this.historyModel.find(
      {
        account: accountIds[i],
        music: musicId,
      },
      {},
      { limit: 100, sort: { date: -1 } },
    );
    if (!history) {
      accountIds = accountIds.filter(
        (accountId) => accountId !== accountIds[i],
      );
    }
  }
  accountIds.forEach((account) => {
    const accountStringId = account.toString();
    accountWithSimilar.set(
      accountStringId,
      (accountWithSimilar.get(accountStringId) || 0) + 1,
    );
  });
}
const accountWithSimilarSorted = Array.from(accountWithSimilar).sort(
  ([, countA], [, countB]) => countB - countA,
);
const recommendationListIds: string[] = [];
for (const [account] of accountWithSimilarSorted) {
  const favorites = await this.favoriteModel.find({ account }).exec();
  if (favorites) {
    for (const favorite of favorites) {
      const musicId = favorite.music.toString();
      if (
        !uniqueMusicIds.includes(musicId) &&
        !recommendationListIds.includes(musicId)
      ) {
        recommendationListIds.push(musicId);
      }
      if (recommendationListIds.length >= 50) break;
    }
  }
  if (recommendationListIds.length >= 50) break;
}
const recommendationList = await this.musicModel
  .find({ _id: { $in: recommendationListIds } })
  .exec();
return recommendationList;
}
}

```

B.14 apollo-server\src\session\session.service.ts

```

@Injectable()
export class SessionService {
  constructor(
    @InjectModel(Session.name) private sessionModel: Model<Session>,
    @InjectModel(Account.name) private accountModel: Model<Account>,
  ) {}

  public async isValid(sessionToken: string): Promise<boolean> {
    if (!sessionToken) return false;
    const session: Session = await this.sessionModel
      .findOne({
        sessionToken,
      })
      .exec();
    const isValid: boolean = session.dueDate.getTime() > Date.now();
    if (!isValid) {
      this.sessionModel.deleteOne(session);
    }
    return isValid;
  }

  public async getBySession(sessionToken: string): Promise<
    Account & {
      _id: Types.ObjectId;
    }
  > {
    const session: Session = await this.sessionModel
      .findOne({ sessionToken })
      .exec();
    return this.accountModel.findById(session.account);
  }

  public async createSession(
    account: Account & {
      _id: Types.ObjectId;
    },
  ): Promise<Session> {
    return new this.sessionModel({
      sessionToken: randomUUID(),
      createDate: new Date(),
      dueDate: new Date(Date.now() + SESSION_DUE_TIME),
      account: account._id,
    }).save();
  }
}

```

B.15 apollo-server\src\shared\guards\auth.guard.ts

```

export class AuthGuard implements CanActivate {
  public constructor(private readonly sessionService: SessionService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const sessionToken = request.headers['session-token'];
    const isValid =
      await this.sessionService.isValid(sessionToken);
    return isValid;
  }
}

```

ДОДАТОК В
Слайди презентації

ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ ЗБЕРІГАННЯ ТА ПРОГРАВАННЯ МУЗИКИ З ЗАСТОСУВАННЯМ КЛІЄНТ-СЕРВЕРНОЇ АРХІТЕКТУРИ

Виконав:
ст. гр. КНТ-133м

Синицин С.С.

Керівник:
д-р.філос., доц. каф. ПЗ

Леоценко С.Д.

2024

Рисунок В.1 – Слайд №1

{ Об'єкт та предмет дослідження, мета роботи }

- **Об'єкт дослідження** – програмні засоби для використання та зберігання музики.
- **Предмет дослідження** – вебзастосунки для зберігання та відтворення музики.
- **Мета роботи** – створення вебзастосунку на клієнт-серверній архітектурі для зберігання та відтворення музики.

2

Рисунок В.2 – Слайд №2

{ Огляд аналогів }

Параметр	Spotify	YouTubeMusic	Deezer	SoundCloud
Кількість треків	70 млн	70 млн	73 млн	76 млн
Якість звуку	До 320 кбіт/с	До 256 кбіт/с	До 1411 кбіт/с (FLAC)	До 128 кбіт/с
Підтримка платформ	Windows, macOS, Linux, Android, iOS, Xbox, PlayStation та інші ¹	Windows, macOS, Linux, Android, iOS та інші	Windows, macOS, Linux, Android, iOS та інші	Windows, macOS, Linux, Android та інші
Вартість підписки на преміум-версію	Від \$9.99/місяць	Безкоштовно або від \$9.99/місяць	Від €9.99/місяць	Безкоштовно або від \$4.99/місяць
Можливості безкоштовної версії	Обмежена функціональність з рекламою та обмеженою якістю звуку	Обмежена функціональність з рекламою та обмеженою якістю звуку	Обмежена функціональність з рекламою та обмеженою якістю звуку	Обмежена функціональність з рекламою
Географічна доступність	Доступно в багатьох країнах	Доступно в багатьох країнах	Доступно в багатьох країнах	Доступно в багатьох країнах
Підтримка відео	Немає	Так	Так	Так (переважно від користувачів)
Офлайн-режим	Так	Так	Так	Так

3

Рисунок В.3 – Слайд №3

{ Актуальність роботи }

Актуальність роботи обумовлена зростанням популярності музичних веб-додатків, які мають забезпечувати високу продуктивність, персоналізований користувацький досвід і масштабованість.

У сучасних умовах важливо створювати ефективні алгоритми рекомендацій, які можуть обробляти великі обсяги даних і адаптуватися до потреб користувачів, водночас залишаючись оптимальними з точки зору ресурсів.

Це дозволяє забезпечити конкурентоспроможність сервісів і покращити взаємодію з користувачами.

4

Рисунок В.4 – Слайд №4

{ Порівняння фронтенд фреймворків }

Критерій	Angular	React	Vue
Тип	Повний фреймворк	Бібліотека для інтерфейсів	Прогресивний фреймворк
Мова	TypeScript	JavaScript	JavaScript
Основні концепції	Декларативне програмування, компонентна архітектура, об'єктно-орієнтоване програмування, реактивність	Компонентний підхід, декларативне програмування, віртуальний DOM	Декларативне програмування, компонентний підхід, реактивність
Повноцінність	Повний набір інструментів для розробки (рутуінг, сервісів, форм, HTTP клієнтів)	Потребує додаткових бібліотек для рутуінгу, керування станом тощо	Може потребувати додаткових бібліотек для повноцінної роботи
Типізація	Використання TypeScript, статичний аналіз коду	Використання JavaScript або TypeScript	Використання JavaScript або TypeScript
Гнучкість	Висока, але орієнтована на TypeScript	Дуже гнучка, інтеграція з будь-якими іншими бібліотеками	Гнучка, легко інтегрується у існуючі проекти
Крива навчання	Висока, особливо для новачків у TypeScript та об'єктно-орієнтованому програмуванні	Середня, необхідно звикнути до JSX. Але через велику свободу новачки пишуть дуже погану архітектуру.	Низька, гарна документація і простота використання
Продуктивність	Може впливати на продуктивність при першому завантаженні, але при правильному використанні може бути бистрішою за інші бібліотеки.	Висока завдяки віртуальному DOM	Легка вага
Підтримка	Постійні оновлення та підтримка від Google	Постійні оновлення та підтримка від Facebook	Підтримка від спільноти та розробника
Документація	Вичерпна та структурована	Гарна, але може бути складною через велику кількість бібліотек	Простота для розуміння та багато прикладів
Популярність	Висока	Дуже висока	Висока, але менша ніж у React і Angular

Рисунок В.5 – Слайд №5

{ Порівняння бекенд фреймворків }

Характеристика	Express.js	Nest.js
Модульна архітектура	Відсутня	Присутня
Мова програмування	JavaScript	TypeScript
Розширені можливості	Мінімальні, потребують додаткових модулів	Вбудовані, включаючи підтримку WebSockets, мікросервісів та GraphQL
Простота використання	Легка для новачків	Вимагає знань TypeScript та Angular
Співтовариство та підтримка	Велике, багатий набір сторонніх модулів	Активне, детальна документація
Продуктивність	Висока	Висока

Рисунок В.6 – Слайд №6

{ Порівняння баз даних }

Параметри	MongoDB	SQLite	MySQL
Модель даних	Документоорієнтована	Реляційна	Реляційна
Масштабованість	Висока	Обмежена	Висока
Гнучкість схеми	Висока	Низька	Середня
Продуктивність	Висока	Середня	Висока
Транзакції	Підтримуються	Обмежена підтримка	Підтримуються
Використання	Великі веб-додатки, аналітика даних	Мобільні додатки, невеликі проекти	Великі веб-додатки, електронна комерція

7

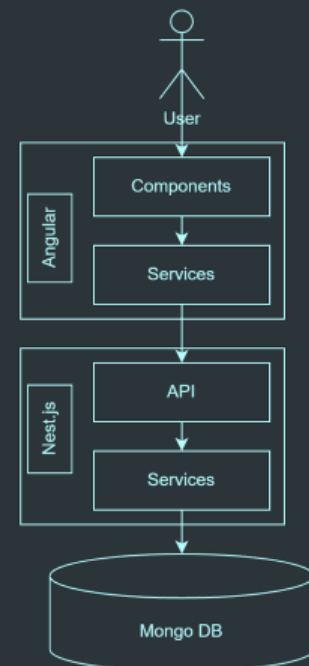
Рисунок В.7 – Слайд №7

{ Характеристики програми }

```

apollo-client/
├── Конфігураційні файли (.editorconfig, .gitignore, angular.json тощо)
├── package.json та пов'язані файли (package-lock.json, tsconfig*.json)
├── src/
│   ├── app/
│   │   ├── Основні компоненти (app.component, app.routes)
│   │   ├── Сторінки (auth, proposals-list, search-list, upload-music)
│   │   └── shared/
│   │       ├── Загальні компоненти (header, music-info, music-list, player-bar)
│   │       └── Константи, DTO, сервіси, типи (api.const, login.dto, account.service тощо)
│   ├── assets/
│   │   ├── Графічні ресурси (svg-іконки, favicon)
│   │   ├── Глобальні стилі та ресурси (colors.scss, reset.scss, styles.scss)
│   └── Вхідні файли (index.html, main.ts)
└──
apollo-server/
├── Конфігураційні файли (.eslintrc.js, .gitignore, nest-cli.json тощо)
├── package.json та пов'язані файли (package-lock.json, tsconfig*.json)
├── src/
│   ├── Основні модулі та компоненти (app.controller, app.module, app.service, main.ts)
│   ├── Функціональні модулі:
│   │   ├── account (контролери, модулі, схеми, сервіси, DTO)
│   │   ├── favorite (контролери, модулі, схеми, сервіси)
│   │   ├── history (контролери, модулі, схеми, сервіси)
│   │   ├── music (контролери, модулі, схеми, сервіси)
│   │   └── session (модуль, схема, сервіс)
│   ├── shared/
│   │   ├── Константи (session.const)
│   │   └── Гварди (auth.guard)
└──

```



8

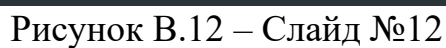
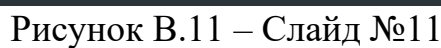
Рисунок В.8 – Слайд №8



Рисунок В.9 – Слайд №9



Рисунок В.10 – Слайд №10



{ Оптимізація програми }

ChangeDetectionStrategy

```
changeDetection: ChangeDetectionStrategy.OnPush,
```

standalone

```
standalone: true,
```

async pipe

```
@if (recommendationsList$ | async; as musicList) {
  <h2 class="title-list">Recommendations:</h2>
  <app-music-list [musicList]="musicList" />
}
```

Lazy loading

```
export const routes: Routes = [
  {
    path: '',
    redirectTo: 'music-list',
    pathMatch: 'full',
  },
  {
    path: 'upload',
    loadComponent: () =>
      import('./upload-music/upload-music.component').then(
        c => c.UploadMusicComponent
      ),
  },
  {
    path: 'music-list',
    loadComponent: () =>
      import('./proposals-list/proposals-list.component').then(
        c => c.ProposalsListComponent
      ),
  },
  {
    path: 'search-list',
    loadComponent: () =>
      import('./search-list/search-list.component').then(
        c => c.SearchListComponent
      ),
  },
];
```

13

Рисунок В.13 – Слайд №13

{ Unit Тестування програми }

Karma v 6.4.4 - connected; test: complete; DEBUG

Chrome 131.0.0.0 (Windows 10) is idle

 **Jasmine** 4.6.1 Options

12 specs, 0 failures, randomized with seed 30417 Finished in 0.037s

```

AudioService
  • should select music and update query$
  • should set audio currentTime
  • should play and pause the audio
  • should be created
  • should go to next music
  • should initialize with empty query and null current audio
  • should subscribe to currentAudio$ and update audio source
  • should set volume and save it to localStorage

AccountService
  • should log out and clear session and account
  • should log in and fetch account
  • should sign up and update session and account
  • should initialize with account and session from localStorage

```

14

Рисунок В.14 – Слайд №14

{ Скріншоти програми / авторизація та реєстрація }

The image shows two screenshots of a web application interface. The left screenshot is the 'Singup' form, which includes fields for Email (test@mail.ua), Password (masked with dots), Name (TestMe), and Description (TestMe description). It has 'To Login' and 'Singup' buttons. The right screenshot is the 'Login' form, which includes fields for Email and Password, and 'To Singup' and 'Login' buttons.

15

Рисунок В.15 – Слайд №15

{ Скріншоти програми / використання програми }



16

Рисунок В.16 – Слайд №16

{ Скріншоти програми / пошук композицій }



Рисунок В.17 – Слайд №17

{ Скріншоти програми / завантаження композицій }

Form fields for uploading a composition:

- Title:
- Artist:
- Album:
- Genre:
- Year:
- Cover:
- File name:
- Music:
- Send button

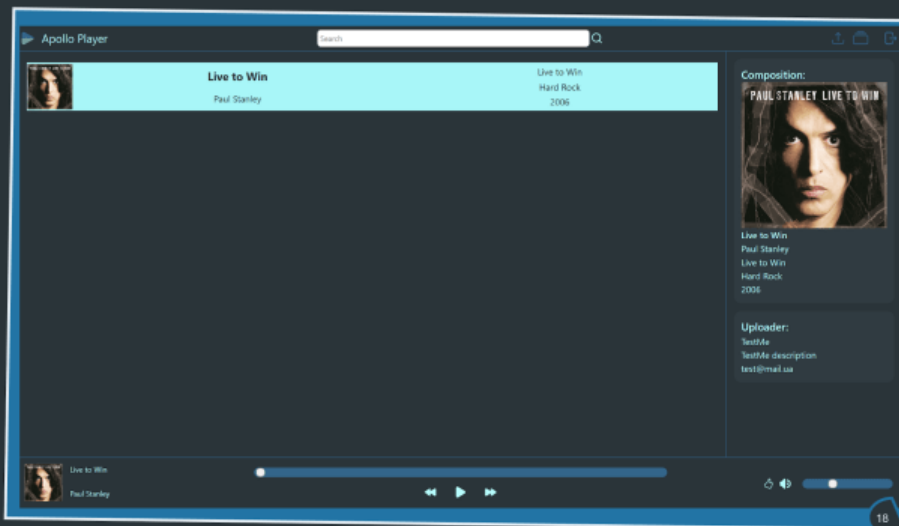
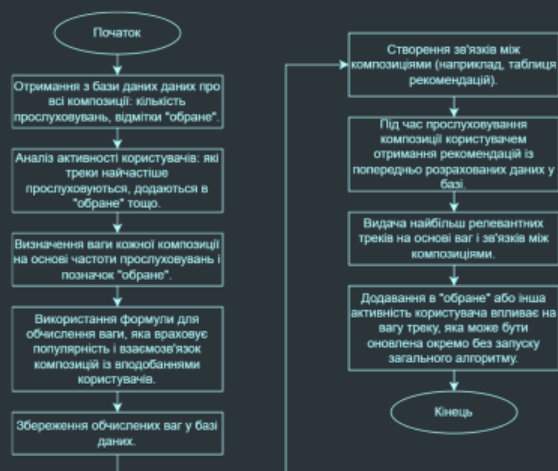


Рисунок В.18 – Слайд №18

{ Інформаційна модель розширення функціональності }



19

Рисунок В.19 – Слайд №19

{ Висновок }

Розроблений веб-додаток для прослуховування музики успішно реалізує поставлені цілі, включаючи основні функції:

- облікові записи користувачів
- музичний програвач
- історію взаємодії
- алгоритм рекомендацій
- пошук і завантаження аудіо.

Використання Angular, Nest.js та MongoDB забезпечило продуктивність і масштабованість. Додаток протестовано, функції працюють стабільно, а обрані технології довели свою ефективність. Проєкт має потенціал для розширення, включаючи соціальні функції та аналітику.

20

Рисунок В.20 – Слайд №20