

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)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБЛЕННЯ ВЕБ ЗАСТОСУНКУ З МОЖЛИВІСТЮ
АВТОГЕНЕРАЦІЇ РЕКОМЕНДАЦІЙ ПЛЕЙЛИСТІВ. DEVELOPING A WEB
APPLICATION WITH AUTOGENERATION OF PLAYLISTS
RECOMMENDATIONS.

Виконав: студент(ка) 4 курсу, групи КНТ-112сп

Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ВОЛОВ М.М.

(ПРІЗВИЩЕ та ініціали)

Керівник ТАБУНЦИК Г. В.

(ПРІЗВИЩЕ та ініціали)

Рецензент ШИТІКОВА О.В.

(ПРІЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 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
 “ ” 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ВОЛОВА Микити Миколайовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розроблення веб застосунку з можливістю автогенерації рекомендацій плейлистів. Developing a web application with autogeneration of playlists recommendations.

керівник проєкту (роботи) к.т.н., професор, ТАБУНЩИК Г. В.,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 28 ” квітня 2025 року № 209

2. Строк подання студентом проєкту (роботи) 30 травня 2025 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. Матеріали і методи. 3. Програмна реалізація 4. Тестування застосунку.

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

Слайди презентації

6. Консультанти розділів проекту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ТАБУНЩИК Г.В. професор		
Нормоконтроль	ДЕЙНЕГА Л. Ю. ст. викл.		

7. Дата видачі завдання 5 травня 2025 р.

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.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

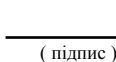
Студент(ка)


(підпис)

Микита ВОЛОВ

(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)


(підпис)

Галина ТАБУНЩИК

(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до випускної кваліфікаційної роботи бакалавра:
80 с., 21 табл., 21 рис., 3 дод., 9 джерел.

ВЕБ-ЗАСТОСУНОК, СИСТЕМА РЕКОМЕНДАЦІЙ, АНАЛІЗ
МУЗИКИ, YOUTUBE, LIBROSA, FASTAPI, REACT

Об'єкт дослідження – веб-застосунки з рекомендаційною системою.

Предмет дослідження – архітектура, принципи побудови та технології реалізації веб-застосунку для музичних рекомендацій на основі контентного аналізу треків.

Мета роботи – створення веб-застосунку, який дозволяє користувачам створювати персоналізовані музичні плейлисти з автоматичною генерацією рекомендацій на основі схожості аудіофайлів.

Матеріали, методи та технічні засоби: бібліотека аудіоаналізу Librosa, TypeScript (React) для клієнтської частини, Python (FastAPI) для серверної частини, база даних PostgreSQL, середовища розробки Visual Studio Code.

Результати. Розроблено повноцінний веб-застосунок, що дозволяє реєстрацію користувачів, створення та редагування плейлистів, додавання YouTube-треків із витяганням аудіо, проведення аудіоаналізу та побудову персоналізованих рекомендацій за допомогою content-based алгоритму.

Висновки. Реалізовано сучасну систему автоматичної генерації рекомендацій до плейлистів, що демонструє здатність до масштабування, адаптації під користувача та обґрунтованої рекомендації музичних композицій на основі їх характеристик.

Галузь використання – рекомендація аудіо на основі вмісту.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 80 pages, 21 tables, 21 figures, 3 appendixes, 9 sources.

WEB APPLICATION, RECOMMENDATION SYSTEM, MUSIC ANALYSIS, YOUTUBE, LIBROSA, FASTAPI, REACT

The object of research is web applications with a recommendation system.

The subject of the research is the architecture, design principles, and implementation technologies of a web application for music recommendations based on track content analysis.

The purpose of this work is to create a web application that allows users to create personalized music playlists with automatic generation of recommendations based on the similarity of audio files.

Materials, methods, and technical means: Librosa audio analysis library, TypeScript (React) for the client side, Python (FastAPI) for the server side, PostgreSQL database, Visual Studio Code development environment.

Results. A full-fledged web application has been developed that allows user registration, creation and editing of playlists, adding YouTube tracks with audio extraction, performing audio analysis, and building personalized recommendations using a content-based algorithm.

Conclusions. A modern system for automatically generating recommendations for playlists has been implemented, demonstrating the ability to scale, adapt to the user, and provide reasonable recommendations for musical compositions based on their characteristics.

Field of application – content-based audio recommendation.

ЗМІСТ

	С.
РЕФЕРАТ	4
ABSTRACT	5
ЗМІСТ	6
ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК	8
ВСТУП	9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	10
1.1 Сучасні веб-застосунки	10
1.2 Системи рекомендацій контенту	13
1.2.1 Поняття системи рекомендацій	13
1.2.2 Огляд сучасних систем рекомендацій контенту	14
1.3 Постанова задачі	15
1.3.1 Загальні вимоги	15
1.3.2 Вимоги до клієнтської частини	16
1.3.3 Вимоги до серверної частини	17
1.3.4 Експлуатаційні вимоги	20
1.4 Висновки за розділом 1	21
2 МАТЕРІАЛИ ТА МЕТОДИ	22
2.1 Вибір та обґрунтування мови програмування	22
2.1.1 Вибір мови для клієнтської частини веб-застосунку	22
2.1.2 Вибір мови для серверної частини веб-застосунку	25
2.2 Вибір та обґрунтування фреймворку	27
2.2.1 Вибір фреймворку для клієнтської частини веб-застосунку	27
2.2.2 Вибір фреймворку для серверної частини веб-застосунку	30

2.3 Вибір та обґрунтування системи рекомендацій	32
2.4 Вибір та обґрунтування середовища розробки	35
2.5 Висновки за розділом 2	38
3 ПРОГРАМНА РЕАЛІЗАЦІЯ	39
3.1 Детальний розгляд архітектури застосунку	39
3.1.1 Вимоги до деплоюменту та запуску	42
3.2 Робота з базою даних	44
3.2.1 Структура бази даних	44
3.2.2 Програмна взаємодія з базою даних	48
3.3 Створення RESTful API	51
3.4 Система рекомендацій	53
3.5 Робота з відкладеними завданнями	55
3.6 Створення веб-клієнту	57
3.7 Висновки за розділом 3	60
4 ТЕСТУВАННЯ ЗАСТОСУНКУ	61
ВИСНОВКИ	64
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ	65
ДОДАТОК А	66
ДОДАТОК Б	68
ДОДАТОК В	71

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

Front-end	– клієнтська частина веб-застосунку;
Back-end	– серверна частина веб-застосунку;
HTML	– Hypertext Markup Language;
CSS	– Cascade Style Sheets;
JS	– JavaScript;
API	– Application Programming Interface;
REST	– Representational State Transfer;
gRPC	– google Remote Procedure Calls;
JSON	– JavaScript Object Notation;
HTTP/HTTPS	– Hypertext Transfer Protocol / Secure;
ПЗ	– Програмне забезпечення;

ВСТУП

У сучасному цифровому середовищі веб-застосунки стали основним способом взаємодії користувачів із сервісами, контентом і даними. Висока доступність браузерів, розвиток інтернет-інфраструктури та велика кількість фреймворків зробили створення повноцінних веб-застосунків набагато простішим, ніж ще кілька років тому. Сьогодні навіть індивідуальний розробник має змогу реалізувати масштабовану систему з клієнтською та серверною частиною, використовуючи відкриті бібліотеки та інструменти.

Особливу популярність у веб-індустрії здобули системи рекомендацій, які дозволяють покращити користувацький досвід завдяки персоналізованим пропозиціям. Такі системи активно використовуються у медіа, електронній комерції, навчанні, кіноіндустрії та інших сферах, де існує потреба у швидкому відборі релевантного контенту з великої кількості варіантів.

Метою даної кваліфікаційної роботи є розробка сучасного веб-застосунку з можливістю автоматичної генерації музичних рекомендацій на основі контентного аналізу треків. Робота охоплює повний цикл створення ПЗ: від аналізу предметної області до реалізації системи та її тестування.

Проект демонструє практичну реалізацію системи рекомендацій з мінімальними ресурсами, що є актуальним для багатьох прикладних завдань сучасної розробки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Сучасні веб-застосунки

Веб-застосунки у сучасному світі у топі популярності за сукупністю факторів таких як: гнучкість, масштабованість, малий поріг входу та велика кількість інструментів для розробки. Веб-застосунки також мають набір архітектурних рішень для побудови.

На даний час основними підходами до архітектури вебзастосунків є монолітна, мікросервісна, serverless та Jamstack-архітектури. Кожна з моделей характеризується специфічними перевагами та недоліками.

Монолітна архітектура передбачає об'єднання усіх компонентів застосунку в єдиний програмний блок. Вона відзначається відносною простотою впровадження та розгортання, проте має обмеження у масштабованості при збільшенні обсягу функціональності.

Мікросервісна архітектура базується на поділі застосунку на незалежні сервіси із чіткими межами відповідальності. Цей підхід забезпечує високу масштабованість та гнучкість системи, але значно ускладнює її розробку, підтримку та супровід.

Serverless-архітектура організована навколо виконання окремих функцій у хмарних середовищах за запитом. Вона оптимізована для застосунків із нерівномірним навантаженням, однак передбачає ризики залежності від обраного постачальника хмарних послуг.

Jamstack-архітектура (JavaScript, APIs, Markup) орієнтована на попередню генерацію статичних сторінок із подальшим динамічним збагаченням через API. Даний підхід забезпечує високу швидкість завантаження та рівень безпеки, водночас ускладнюючи реалізацію інтерактивних функціональних можливостей.

У таблиці 1.1 наведено порівняння основних характеристик архітектур.

Таблиця 1.1 – Порівняння архітектур

Критерій	Монолітна	Мікросервісна	Serverless	Jamstack
Швидкість початкової розробки	Висока	Низька	Висока	Середня
Необхідність складної інфраструктури	Низька	Висока	Низька	Низька
Початкові фінансові витрати	Низькі	Високі	Низькі	Низькі
Складність подальшої підтримки	Зростає зі збільшенням функціоналу	Висока	Помірна	Низька
Можливість масштабування	Обмежена	Висока	Висока	Помірна
Доцільність використання для малих проектів	Висока	Низька	Висока	Висока

Для цього проекту було б раціонально використати монолітну архітектуру, її можливо умовно розділити на сукупність клієнтської та серверної частин. Або як більш прийнято у професіональних колах front-end та back-end.

Клієнтська частина (Front-end) являє собою:

- інтерфейс користувача: вікна, кнопки, поля вводу, таблиці, діаграми тощо;
- логіку взаємодії: реакція на кліки, скролінг, запити до сервера, валідацію форм і т.д.;
- візуальні ефекти: анімації, зміна станів елементів без перезавантаження сторінки.

Типовими формами клієнтської частини є багатосторінкові застосунки на яких кожна дія користувача призводить до перезавантаження сторінки, односторінкові застосунки у яких весь інтерфейс працює у межах однієї сторінки, динамічно оновлюючи вміст без перезавантаження, прогресивні веб-застосунки, вони виглядають і працюють майже як нативні мобільні додатки: підтримують офлайн-режим, пуш-нотифікації тощо.

Також до цих форм можна віднести мобільні застосунки на веб-технологія у яких клієнтська частина реалізується як мобільний додаток, але створена за допомогою HTML/CSS/JS.

Серверна частина (back-end) забезпечує обробку запитів клієнтів, виконання бізнес-логіки, зберігання і захист даних, а також взаємодію з іншими сервісами.

Серверна частина виконує декілька ключових функцій. По-перше, вона відповідає за обробку запитів клієнтів, що включає прийняття запитів, їх маршрутизацію та формування відповідей.

По-друге, серверна частина реалізує бізнес-логіку застосунку: виконує перевірку правил, обробляє транзакції та керує процесами відповідно до бізнес-вимог.

Також важливою функцією є аутентифікація та авторизація користувачів, тобто перевірка особистості користувача та визначення його прав доступу до ресурсів системи. Крім того, сервер забезпечує управління базами даних – здійснює зберігання, отримання, оновлення та видалення даних. Нарешті, серверна частина здійснює інтеграцію із зовнішніми сервісами, наприклад, платіжними шлюзами чи сторонніми API, що розширюють функціональні можливості застосунку.

Структура серверної частини зазвичай включає кілька ключових компонентів. Першим елементом є веб-сервер, який приймає HTTP або HTTPS-запити від клієнтів та перенаправляє їх на сервер застосунку для подальшої обробки. Найпоширенішими прикладами веб-серверів є Nginx та Apache.

Другим елементом є сервер застосунків, який реалізує основну бізнес-логіку системи. Він обробляє запити, керує сесіями користувачів, забезпечує виконання бізнес-процесів та взаємодіє із зовнішніми або внутрішніми ресурсами. Для створення серверів застосунків

використовуються такі технології, як Node.js, Django, FastAPI, Flask або Spring Boot.

Третім компонентом є база даних, яка відповідає за довгострокове зберігання структурованої або неструктурованої інформації. Бази даних можуть бути реляційними, такими як PostgreSQL і MySQL, або нереляційними, такими як MongoDB і Redis.

Ще одним важливим елементом є API-шари, що забезпечують стандартизовану взаємодію між клієнтською і серверною частинами. Через API клієнт надсилає запити до сервера і отримує структуровані відповіді, найчастіше у форматі JSON. Найпоширенішими підходами до побудови API є REST, GraphQL та gRPC.

1.2 Системи рекомендацій контенту

1.2.1 Поняття системи рекомендацій

Система рекомендацій – це інтелектуальна технологія, що аналізує дані про користувачів та об'єкти для надання персоналізованих пропозицій. Вона допомагає користувачам знаходити релевантну інформацію серед великого обсягу даних, значно скорочуючи час пошуку та підвищуючи якість взаємодії з контентом.

Рекомендаційні системи можна класифікувати за трьома основними підходами, кожен з яких має свої особливості, переваги та обмеження.

Колаборативна фільтрація базується на аналізі поведінки користувачів та їхніх уподобань. Вона передбачає, що люди, які погоджувалися в минулому, погоджуватимуться і в майбутньому. Наприклад, Netflix пропонує фільми на основі того, що дивились користувачі з подібними смаками. Цей метод поділяється на два підтипи: заснований на користувачах (визначає схожість між користувачами) та заснований на об'єктах (визначає схожість між об'єктами).

Контентна фільтрація (Content-based filtering) аналізує властивості самих об'єктів для знаходження схожих. Spotify, наприклад, аналізує музичні характеристики пісень – темп, гучність, інструментальність – і рекомендує композиції з подібними параметрами. Цей підхід не залежить від даних про інших користувачів, що робить його ефективним для нових елементів та унікальних смаків.

Гібридні системи поєднують переваги обох попередніх підходів, компенсуючи їхні недоліки. Amazon використовує такий підхід, аналізуючи як характеристики товарів, так і поведінку користувачів, що дозволяє пропонувати релевантніші рекомендації навіть для нових користувачів чи товарів.

1.2.2 Огляд сучасних систем рекомендацій контенту

Сучасні системи рекомендацій все частіше використовують глибоке навчання для обробки складних неструктурованих даних. Нейронні мережі дозволяють автоматично виявляти приховані закономірності в даних без необхідності ручного проектування ознак.

Контекстно-орієнтовані рекомендації враховують ситуативні фактори, такі як час доби, погода або місцезнаходження. Наприклад, сервіс доставки їжі може рекомендувати гарячі страви в холодну погоду або прохолодні напої влітку.

Онлайн-навчання дозволяє системам адаптуватися до змін у реальному часі, отримуючи зворотний зв'язок від користувачів і відповідно коригуючи рекомендації. Це особливо важливо для динамічних середовищ, таких як новинні платформи або соціальні мережі.

Приклади успішних рекомендаційних систем:

YouTube використовує складну рекомендаційну систему, що аналізує історію переглядів, час перегляду, вподобання та інші фактори для

пропонування відеоконтенту. За даними компанії, рекомендації генерують понад 70% загального часу перегляду на платформі [1].

Spotify розробив систему Discover Weekly, яка щотижня створює персоналізований плейлист із 30 пісень для кожного активного користувача. Система поєднує аналіз аудіохарактеристик пісень із даними про поведінку користувачів, враховуючи не лише те, що слухав конкретний користувач, але й музику, яку обирають люди з подібними смаками [2].

Amazon використовує рекомендації для збільшення середнього чеку та частоти покупок. Система аналізує покупки, перегляди товарів, рейтинги та навіть час, проведений на сторінці товару. За оцінками експертів, до 35% продажів компанії генеруються саме рекомендаційними алгоритмами [3].

1.3 Постанова задачі

Головною задачею цієї дипломної роботи є дослідження сучасних засобів розробки веб-застосунків та систем рекомендацій контенту на прикладі розробки веб-застосунку з можливістю рекомендації музичних композицій до створеного користувачем плейлісту.

Також другою ціллю цієї дипломної роботи я бачу демонстрацію малої практичної складності реалізації реального робочого застосунку з урахуванням сучасних методів та засобів розробки ПЗ.

1.3.1 Загальні вимоги

До системи висуваються наступні загальні вимоги:

- система повинна надавати можливість користувачам реєструватися та входити в систему, використовуючи автентифікацію через електронну пошту/пароль;

- система повинна дозволяти користувачам створювати, редагувати та видаляти плейлисти;
- система повинна забезпечувати можливість додавання музичних відео з YouTube до плейлистів користувачів за допомогою URL-адрес;
- система повинна витягувати аудіо з відео на YouTube за допомогою Librosa для контент-аналізу;
- система повинна генерувати рекомендації на основі схожості змісту треків, що вже є в плейлистах користувачів;
- система повинна забезпечувати базовий функціонал пошуку музики, яка вже є в базі даних системи;
- система повинна дозволяти користувачам відтворювати відео з YouTube безпосередньо в інтерфейсі програми;
- система повинна відображати основні метадані для кожного треку (назва, виконавець, тривалість);
- система повинна зберігати історію прослуханих треків для кожного користувача;
- система повинна надавати інформаційну панель, що відображає найпопулярніші жанри та виконавців, які найчастіше відтворюються користувачами.

1.3.2 Вимоги до клієнтської частини

До системи висуваються наступні вимоги, що до клієнтської частини:

- клієнтська частина повинна надавати чистий, інтуїтивно зрозумілий інтерфейс для створення та управління плейлистами;
- клієнтська частина повинна реалізовувати компонент відеоплеєра YouTube для відтворення музики в додатку;
- клієнтська частина повинна відображати рекомендації поряд з поточним плейлистом;

- клієнтська частина повинна надавати форму для додавання URL-адрес YouTube до плейлистів;
- клієнтська частина повинна реалізовувати адаптивний дизайн для десктопних та мобільних пристроїв;
- клієнтська частина повинна відображати індикатори завантаження під час процесів вилучення аудіофайлів.
- клієнтська частина повинна надавати візуальний зворотній зв'язок про схожість між треками;
- клієнтська частина повинна реалізовувати простий інтерфейс пошуку треків у системі;
- клієнтська частина повинна відображати помилки, коли URL-адреси YouTube невірні або відео недоступні;
- клієнтська частина повинна показувати базову аналітику треків (кількість відтворень, дата додавання тощо) для кожного плейлиста.

1.3.3 Вимоги до серверної частини

До системи висуваються наступні вимоги, що до серверної частини:

- серверна частина повинна реалізовувати безпечну автентифікацію користувачів та управління сесіями;
- серверна частина повинна зберігати дані користувачів, плейлисти та інформацію про треки в базі даних;
- серверна частина повинна інтегруватися з API YouTube для вилучення метаданих відео;
- серверна частина повинна реалізувати Librosa для вилучення аудіо з відео на YouTube;
- серверна частина повинна розробити алгоритм фільтрації на основі вмісту, використовуючи витягнуті аудіо-дані;

- на стороні сервера повинні бути надані кінцеві точки RESTful API для всіх операцій на стороні клієнта;
- на стороні сервера має бути реалізовано кешування часто використовуваних даних для підвищення продуктивності;
- серверна частина повинна реєструвати взаємодію з користувачем для подальшого аналізу та вдосконалення алгоритму;
- серверна сторона повинна належним чином обробляти обмеження швидкості YouTube API;
- на стороні сервера має бути реалізована обробка помилок для невдалої обробки відео або недоступного контенту.

На таблицях 1.2, 1.3, 1.4 демонструються матриці трасування вимог.

Таблиця 1.2 – Матриця трасування вимог

Вимога	Опис Вимоги	Тест-кейс	Опис Тест-кейсу
3-1	Реєстрація та вхід через email/пароль	ТС-01	Перевірка реєстрації нового користувача
3-1		ТС-02	Перевірка входу існуючого користувача
3-2	Створення, редагування, видалення плейлистів	ТС-03	Створення нового плейлиста
3-2		ТС-04	Редагування існуючого плейлиста
3-2		ТС-05	Видалення плейлиста
3-3	Додавання відео з YouTube за URL	ТС-06	Додавання валідного URL YouTube
3-3		ТС-07	Додавання невалідного URL (очікувана помилка)
3-4	Витяг аудіо через Librosa	ТС-08	Витяг аудіо з доданого відео
3-5	Генерація рекомендацій на основі змісту	ТС-09	Перевірка релевантності рекомендацій
3-6	Базовий пошук музики	ТС-10	Пошук треку по назві

Продовження таблиці 1.2.

Вимога	Опис Вимоги	Тест-кейс	Опис Тест-кейсу
3-7	Відтворення відео в інтерфейсі	ТС-11	Відтворення відео у вбудованому плеєрі
3-8	Відображення метаданих треку	ТС-12	Перевірка відображення назви, виконавця, тривалості
3-9	Історія прослуханих треків	ТС-13	Перевірка запису прослуханого треку
3-10	Інформаційна панель популярних виконавців	ТС-14	Відображення виконавців

Далі подивимось на таблицю 1.3 де представлено матрицю трасування вимог до клієнтської частини.

Таблиця 1.3 – Матриця трасування вимог до клієнтської частини

Вимога	Опис Вимоги	Тест -кейс	Опис Тест-кейсу
К-1	Інтерфейс для управління плейлистами	ТС-15	Оцінка інтуїтивності створення/управління
К-2	Відеоплеєр YouTube в додатку	ТС-16	Відтворення музики через плеєр
К-3	Відображення рекомендацій поряд із плейлистом	ТС-17	Наявність блоку рекомендацій
К-4	Форма для додавання URL	ТС-18	Коректність роботи форми URL
К-5	Адаптивний дизайн	ТС-19	Тестування вигляду на мобільному/десктопі
К-6	Відображення нових треків	ТС-20	Перевірка відображення додавання нових треків до системи
К-7	Візуальний фідбек про схожість треків	ТС-21	Відображення ступеня схожості треків у рекомендаціях відповідно до плейлісту
К-8	Інтерфейс пошуку треків	ТС-22	Тест пошуку треків через інтерфейс
К-9	Відображення помилок для некоректних URL	ТС-23	Перевірка повідомлень про помилку
К-10	Аналітика треків для плейлистів	ТС-24	Відображення кількості відтворень, дати додавання

Далі подивимось на таблицю 1.4 де представлено матрицю трасування вимог до серверної частини.

Таблиця 1.4 – Матриця трасування вимог до серверної частини

Вимога	Опис Вимоги	Тест-кейс	Опис Тест-кейсу
C-1	Безпечна автентифікація і сесії	ТС-25	Тестування безпеки сесій і паролів
C-2	Збереження даних користувачів і плейлистів	ТС-26	Перевірка збереження в базі даних
C-3	Інтеграція з YouTube API	ТС-27	Отримання метаданих через API
C-4	Витяг аудіо через Librosa	ТС-28	Перевірка цілісності витягнутого аудіо
C-5	Алгоритм фільтрації на основі вмісту	ТС-29	Оцінка точності рекомендацій
C-6	RESTful API для клієнта	ТС-30	Тестування всіх кінцевих точок API
C-7	Кешування популярних даних	ТС-31	Перевірка швидкості доступу до кешованих даних

1.3.4 Експлуатаційні вимоги

Мінімальні вимоги до програмно-технічних засобів користувача:

- процесор з тактовою частотою не менше за 2.5 ГГц;
- наявність оперативної пам'яті не менше 8 Гб;
- наявність сучасного браузеру;
- наявність екрану;
- наявність пристроїв типу клавіатура і миша;
- широкополосний доступ до мережі інтернет.

Мінімальні вимоги до програмно-технічних засобів серверу:

- підтримка віртуалізації;
- встановлений та працюючий Docker;
- процесор з тактовою частотою не менше за 2.5 ГГц;
- наявність оперативної пам'яті не менше 8 Гб;
- дискове сховище не менше 40 Гб;
- широкополосний доступ до мережі інтернет.

1.4 Висновки за розділом 1

У цьому розділі було розглянуто складові сучасних веб-застосунків та загальні риси систем рекомендації контенту. Визначено, що сучасні веб-застосунки складаються з клієнтської та серверної частин, кожна з яких має свої специфічні компоненти та функції. Досліджено основні підходи до побудови систем рекомендацій контенту, включаючи колаборативну фільтрацію, контентну фільтрацію та гібридні підходи. На основі проведеного аналізу сформульовано функціональні та експлуатаційні вимоги до розроблюваної системи. Розроблені матриці трасування вимог.

2 МАТЕРІАЛИ ТА МЕТОДИ

У цьому розділі буде проведено вибір та обґрунтування мови програмування, фреймворку, типу системи рекомендацій та середовища розробки з приведенням порівняльних таблиць.

2.1 Вибір та обґрунтування мови програмування

2.1.1 Вибір мови для клієнтської частини веб-застосунку

Серед усіх сучасних мов програмування, що використовуються для розробки клієнтської частини веб-застосунків, фактично домінує одна мова – JavaScript. Це єдина мова, яка підтримується нативно всіма сучасними браузерами без необхідності додаткових плагінів або компіляторів. Саме через цю універсальність JavaScript став стандартом де-факто для фронтенду.

JavaScript є мовою з динамічною слабкою типізацією, що означає можливість гнучкого оперування змінними без обов'язкового попереднього оголошення типів. Наприклад, один і той самий оператор може застосовуватись до рядків, чисел або булевих значень без жорсткого контролю над типами. Така гнучкість на перших етапах може здатись зручною для швидкої розробки, однак вона створює ризики виникнення складних для відстеження помилок у великих проєктах. Одним із класичних прикладів є неочікувана поведінка при виконанні операцій над різними типами, зокрема автоматичне перетворення чисел у рядки або навпаки, що може призвести до непередбачуваних результатів. На рисунку 2.1 демонструється типовий приклад такої контрінтуїтивної поведінки.

```
console.log(2 + 2);  
// 4  
console.log(2 + "2");  
// 22  
console.log("2" + + 2);  
// 22  
console.log(2 + + "2");  
// 4  
console.log(2 + 2 - 2);  
// 2  
console.log("2" + "2" - "2");  
// 20
```

Рисунок 2.1 – Приклад непередбачуваного результату через слабку типізацію JavaScript

У невеликих або прототипних застосунках JavaScript може бути достатнім завдяки простоті використання та швидкому запуску. Однак для великих масштабованих веб-застосунків, де важлива передбачуваність, читабельність коду та його підтримуваність, виникає потреба у більш строго контрольованому середовищі.

Рішенням цієї проблеми став TypeScript – надмножина JavaScript із підтримкою статичної сильної типізації. TypeScript вимагає явного оголошення типів змінних і не дозволяє виконання операцій між несумісними типами без попереднього перетворення, що значно знижує кількість логічних помилок. На відміну від JavaScript, TypeScript є компільованою мовою: перед запуском код компілюється у чистий JavaScript, придатний для виконання в браузері. Це дозволяє не тільки виявляти помилки ще на етапі розробки, а й використовувати сучасні можливості мови (класи, інтерфейси, дженерики), які в результаті будуть трансформовані у сумісний JavaScript-код, що підтримується будь-яким середовищем виконання.

Варто також зазначити, що більшість сучасних фреймворків і бібліотек, таких як React, Angular або Vue, мають повну або часткову підтримку TypeScript, що робить його кращим вибором для розробки великих

застосунків, де важлива командна розробка, перевірка типів, інтеграція з IDE та наявність зрозумілої документації.

Підсумовуючи, TypeScript забезпечує баланс між гнучкістю JavaScript та надійністю класичних типізованих мов програмування, таких як Java або C#. Саме тому у розробці клієнтської частини сучасних веб-застосунків, особливо масштабованих або критично важливих, рекомендовано обирати TypeScript як основну мову розробки.

У таблиці 2.1 наведено порівняння основних характеристик JavaScript та TypeScript.

Таблиця 2.1 – Порівняння JavaScript та TypeScript

Критерій	JavaScript	TypeScript
Типізація	Динамічна, слабка	Статична, сильна
Обробка помилок	Переважно виявлення у рантаймі	Виявлення помилок на етапі компіляції
Надійність у великих проєктах	Низька: висока ймовірність неочевидних помилок	Висока: контроль типів зменшує кількість помилок
Зручність підтримки	Обмежена: важко масштабувати код	Висока: спрощена підтримка завдяки типам та інтерфейсам
Інструментальна підтримка	Базова: автодоповнення залежить від середовища	Повна: розширене автодоповнення, рефакторинг, навігація
Навчальна крива	Полога: легкий старт	Складніша: потребує знання типізації та компіляції
Сумісність із браузерами	Повна: виконується напямую	Повна (після компіляції у JavaScript)
Підтримка сучасних стандартів	Часткова: залежить від середовища виконання	Повна: підтримує останні можливості через трансляцію
Уніфікованість коду в команді	Низька: відсутність строгих правил	Висока: типізація задає чіткі контракти між компонентами
Поширеність у сучасних фреймворках	Поширений, але менш бажаний у великих проєктах	Стандарт де-факто у React, Angular, Vue

Отже, з наведеного порівняння можна зробити висновок, що хоча JavaScript залишається основною мовою для клієнтської частини

веб-застосунків завдяки своїй сумісності з усіма браузерами, його слабка типізація та відсутність компіляції роблять його менш придатним для розробки великих і масштабованих проєктів. У свою чергу, TypeScript, як надмножина JavaScript, забезпечує всі переваги останнього, доповнюючи їх строгою типізацією, компіляцією та потужними засобами інструментального супроводу. Це робить його кращим вибором для сучасної фронтенд-розробки, де важливі надійність, підтримуваність та ефективність командної роботи.

2.1.2 Вибір мови для серверної частини веб-застосунку

У сучасних веб-застосунках, які включають системи рекомендацій, вимоги до серверної частини є комплексними: необхідна обробка великого обсягу даних, інтеграція алгоритмів машинного навчання, швидка відповідь на запити, масштабованість, безпека та стабільність. Однією з найважливіших мов, яка забезпечує ці потреби у контексті штучного інтелекту, є Python. Його екосистема пропонує вражаючу кількість бібліотек для побудови моделей (TensorFlow, PyTorch, scikit-learn), обробки даних (NumPy, pandas), а також зручні веб-фреймворки (FastAPI, Flask), які дають змогу швидко створювати прототипи та перетворювати їх у продакшен-рішення.

Основною перевагою Python є надзвичайна гнучкість у побудові та тестуванні алгоритмів рекомендацій. Його синтаксис простий, що прискорює розробку MVP, зменшує кількість помилок і полегшує командну співпрацю. У поєднанні з такими фреймворками як FastAPI Python може досягати високих показників у обробці HTTP-запитів, особливо у мікросервісній архітектурі, де обчислювально важка логіка може бути ізольована та масштабована окремо. Так, хоча інтерпретована природа мови і наявність GIL обмежують продуктивність у багатопоточності, ця проблема вирішується за допомогою асинхронності, multiprocessing або розподілення навантаження між процесами чи контейнерами. Також, у випадку CPU-bound обробки, критичні

частини можуть бути винесені у C/C++ розширення через Cython або використовуватись у вигляді зовнішніх сервісів.

На противагу Python, C# і Java пропонують строгішу типізацію та вищу продуктивність у багатопоточному середовищі, що корисно для високонавантажених застосунків. Java має чудову підтримку масштабування (Spring Boot, Vert.x) і стабільність, але вимагає більше конфігурації та є менш гнучкою для ML. C# у .NET середовищі вигідно поєднує високу продуктивність з простотою розробки, але обмежена за гнучкістю роботи з ML-екосистемами – хоча ML.NET розвивається, він все ще суттєво відстає від Python.

Go демонструє високу ефективність у побудові швидких та простих сервісів. Завдяки легким горутинам і мінімалістичному підходу Go ідеальний для інфраструктурного рівня (наприклад, маршрутизація або логіка кешування). Однак його можливості для роботи з ML вкрай обмежені, і це критично для рекомендаційної системи.

JavaScript/TypeScript на сервері (Node.js) виграють у тому, що одна мова обслуговує і клієнт, і сервер. Однак їх обчислювальна модель (однопоточний event-loop) не підходить для інтенсивних розрахунків, характерних для рекомендаційних алгоритмів. TypeScript зменшує ризик помилок, але не компенсує відсутність повноцінної ML-екосистеми.

C/C++ і Rust – це мови, орієнтовані на максимальну продуктивність. Їх доцільно застосовувати для реалізації обчислювальних ядер або модулів, які вимагають контролю над пам'яттю і часовими затримками. Проте загальна логіка веб-застосунку на цих мовах є надто громіздкою, розробка потребує значно більше часу, а поріг входу дуже високий.

Порівняння мов можна побачити у таблиці 2.2.

Таблиця 2.2 – Порівняння мов програмування для серверної частини

Мова	Продуктивність	ML/AI Екосистема	Швидкість розробки	Асинхронність / Паралелізм	Простота підтримки	Готовність до продакшену
Python	Середня	Відмінна	Висока	Середня (GIL, multiprocessing)	Висока	Висока
Java	Висока	Середня	Середня	Висока	Середня	Висока
C#	Висока	Обмежена	Середня	Висока	Висока	Висока
Go	Висока	Слабка	Висока	Висока	Висока	Висока
JS/TS	Середня	Слабка	Висока	Висока (але без CPU-bound)	Висока	Висока
C/C++	Дуже висока	Обмежена	Низька	Висока	Низька	Висока (модульно)
Rust	Дуже висока	Обмежена	Низька	Висока (безпечна)	Середня	Зростаюча

Python є найбільш збалансованим вибором для серверної частини веб-застосунку з рекомендаційною системою, особливо коли важлива гнучкість, швидкість розробки, підтримка бібліотек ML та легкість інтеграції з хмарними сервісами. У поєднанні з асинхронними фреймворками (FastAPI, Starlette), кешами (Redis) та зовнішніми обчислювальними модулями (наприклад, на Rust або C++) Python дозволяє створити продуктивну та масштабовану систему, не втрачаючи при цьому темпу розробки.

2.2 Вибір та обґрунтування фреймворку

2.2.1 Вибір фреймворку для клієнтської частини веб-застосунку

При виборі фреймворку для реалізації клієнтської частини сучасного веб-застосунку надзвичайно важливо враховувати не лише популярність технології, а й її відповідність до архітектурних вимог проєкту, таких як масштабованість, інтеграція зі сторонніми бібліотеками, продуктивність рендерингу та зручність у підтримці коду. У контексті застосунку з системою рекомендацій, який передбачає активну взаємодію користувача з даними, динамічну маршрутизацію та компонентну архітектуру, доцільним є використання фреймворків класу SPA (Single Page Application).

Найбільш поширеними фреймворками або бібліотеками для фронтенд-розробки є React, Angular, Vue.js, а також менш відомі, але прогресивні рішення як Svelte або SolidJS. Усі вони вирішують схожі задачі – побудову інтерфейсу, управління станом і маршрутизацією, але мають відмінності у філософії, складності, продуктивності та гнучкості.

React – це не зовсім повноцінний фреймворк, а радше бібліотека для побудови UI, однак завдяки своїй гнучкості, екосистемі та підтримці інструментів (як-от TanStack Router, Redux, React Query) він перетворюється на фреймворк широкого призначення. React реалізує декларативну модель розробки через компоненти, що суттєво спрощує масштабування та підтримку. У поєднанні з TypeScript він забезпечує сувору типізацію компонентів, пропсів і стану, знижуючи кількість помилок ще на етапі розробки. Бібліотека TanStack Router дозволяє ефективно організувати клієнтську маршрутизацію, підтримуючи вкладені маршрути, лейаут-систему та асинхронне завантаження контенту. Завдяки широкій спільноті та підтримці, React є безпечним вибором для проєктів будь-якого масштабу.

Angular – це повноцінний фреймворк з усіма вбудованими рішеннями: маршрутизація, форма, HTTP-запити, DI (ін'єкція залежностей), валідація, навіть build-система. Angular побудований на TypeScript та орієнтований на корпоративні застосунки з високими вимогами до структури коду та суворих патернів. Його головною перевагою є узгодженість, однак це робить його менш гнучким для швидких змін або нестандартних рішень. Навчальна крива Angular є значно вищою порівняно з React або Vue.

Vue.js – це прогресивний фреймворк, який об'єднує найкращі ідеї з React та Angular. Він легкий у засвоєнні, підтримує реактивність «з коробки» та має офіційні інструменти для маршрутизації (Vue Router) і управління станом (Pinia, раніше Vuex). У проєктах малого та середнього масштабу Vue показує високу ефективність, але для масштабних застосунків потребує детального налаштування архітектури, що не завжди стандартизовано.

Svelte – це сучасний компілятор, який трансформує декларативні компоненти у оптимізований JavaScript-код ще на етапі збірки. Його основна перевага – швидкість рендерингу і мінімальний розмір бандла. Однак екосистема ще молода, і бракує інструментів рівня React або Angular для великих командних проєктів.

SolidJS – ще одна новітня розробка, орієнтована на продуктивність і реактивність. Вона має найшвидший DOM-рендеринг серед конкурентів, але майже не має офіційної підтримки розширених інструментів, що обмежує його застосування у складних проєктах.

Порівняльну характеристику найбільш популярних фреймворків наведено у таблиці 2.3.

Таблиця 2.3 – Порівняння фреймворків

Критерій	React	Angular	Vue.js	Svelte	SolidJS
Типізація	Підтримка через TypeScript	Вбудована TypeScript	Опційна (через TS)	Опційна	Опційна
Маршрутизація	TanStack Router	Angular Router	Vue Router	SvelteKit Routing	Низькорівнева
Навчальна крива	Помірна	Висока	Низька	Низька	Висока
Продуктивність	Висока	Середня	Висока	Дуже висока	Найвища
Екосистема	Найбільша	Велика	Помірна	Молода	Дуже молода
Масштабованість	Висока	Висока	Середня	Обмежена	Обмежена
Офіційна підтримка рішень	Частково (через окремі пакети)	Повна «з коробки»	Часткова	Часткова	Низька
Гнучкість	Висока	Низька	Висока	Висока	Висока

У результаті порівняння можна стверджувати, що React у поєднанні з TypeScript і TanStack Router є оптимальним вибором для розробки клієнтської частини веб-застосунку з системою рекомендацій. Такий стек забезпечує баланс між продуктивністю, масштабованістю, типовою безпекою, багатством екосистеми та гнучкістю. React дозволяє поступово розширювати

функціональність застосунку без жорстких архітектурних обмежень, водночас підтримуючи високий рівень якості та зрозумілість коду.

2.2.2 Вибір фреймворку для серверної частини веб-застосунку

Серверна частина веб-застосунку з системою рекомендацій повинна забезпечувати високу швидкість обробки запитів, масштабованість, простоту інтеграції з базами даних, підтримку асинхронних операцій та можливість легкої розширюваності API. У контексті розробки на Python існує декілька зрілих фреймворків, серед яких варто розглянути Django, Flask, FastAPI та Tornado.

Django – це потужний фреймворк рівня "все в одному", що забезпечує повний набір інструментів для розробки веб-застосунків: ORM для роботи з базами даних, систему автентифікації, маршрутизацію, форму обробки запитів, підтримку шаблонів. Django чудово підходить для розробки складних монолітних застосунків, однак для системи рекомендацій, яка може потребувати великої кількості асинхронних запитів і взаємодії через API, його монолітна архітектура може бути обмеженням.

Flask – це мікрофреймворк, що надає мінімальну базу для побудови веб-сервера. Flask дуже гнучкий, має велику кількість розширень для підключення баз даних, валідації запитів та автентифікації. Його мінімалізм робить його хорошим вибором для невеликих застосунків або API, однак для складних систем потрібно самостійно інтегрувати додаткові компоненти, що може збільшити час розробки.

FastAPI – це сучасний фреймворк для створення API на базі Python 3.7+ із підтримкою асинхронних функцій та автоматичною генерацією документації OpenAPI (Swagger). Його головною перевагою є надзвичайно висока швидкість розробки завдяки вбудованій валідації даних через типізацію (pydantic), а також підтримка асинхронного програмування, що

критично для високонавантажених застосунків. FastAPI оптимізований для побудови RESTful сервісів і мікросервісної архітектури, що ідеально підходить для систем рекомендацій.

Tornado – це фреймворк, орієнтований на обробку великої кількості одночасних з'єднань завдяки використанню неблокуючого I/O [4]. Він дуже продуктивний у специфічних задачах реального часу (наприклад, чати, стрімінгові сервіси), однак має складніший синтаксис і менш зручну для початківців модель розробки, ніж FastAPI чи Flask.

Порівняльну характеристику серверних фреймворків наведено у таблиці 2.4.

Таблиця 2.4 – Порівняння серверних фреймворків

Критерій	Django	Flask	FastAPI	Tornado
Архітектура	Монолітна	Мікрофреймворк	Мікрофреймворк	Мікрофреймворк
Підтримка асинхронності	Обмежена	Можлива через розширення	Повна, вбудована	Повна, вбудована
Продуктивність	Середня	Висока	Дуже висока	Дуже висока
Документація API	Через сторонні пакети	Через сторонні пакети	Автоматична OpenAPI	Немає автоматичної
Простота створення API	Середня	Висока	Дуже висока	Складна
Навчальна крива	Висока	Низька	Помірна	Висока
Масштабованість	Висока	Середня	Висока	Висока
Екосистема	Дуже велика	Велика	Велика та швидко зростаюча	Невелика

Беручи до уваги вимоги до продуктивності, підтримку асинхронного програмування, легкість створення, документування та масштабування API, найкращим вибором для серверної частини веб-застосунку із системою рекомендацій є FastAPI. Цей фреймворк поєднує в собі сучасний підхід до типізації, високу продуктивність та простоту розробки, що забезпечує оптимальне середовище для побудови ефективного, масштабованого та легкого в супроводі бекенду.

2.3 Вибір та обґрунтування системи рекомендацій

Рекомендаційна система – підклас системи фільтрації інформації, яка будує рейтинговий перелік об'єктів (фільми, музика, книги, новини, вебсайти), яким користувач може надати перевагу [5].

У контексті розробки музичної рекомендаційної системи важливо детально розглянути існуючі підходи та обрати оптимальний варіант, враховуючи особливості предметної галузі та доступні ресурси.

Кожен з основних підходів до побудови рекомендаційних систем має свої переваги та обмеження, які необхідно враховувати при розробці проекту. Розглянемо кожен метод з огляду на його застосовність для музичної предметної галузі.

Колаборативна фільтрація базується на аналізі взаємодій користувачів з музичним контентом. Цей підхід дозволяє виявляти неочевидні зв'язки між музичними композиціями на основі колективного досвіду слухачів. Однак він потребує значного обсягу даних про користувацькі вподобання та страждає від проблеми "холодного старту" – неможливості робити якісні рекомендації для нових композицій або користувачів, щодо яких ще немає достатньо інформації.

Content-based фільтрація аналізує безпосередньо характеристики музичних творів, такі як темп, гармонія, тембр та інші аудіоознаки. Даний підхід особливо ефективний для музичних рекомендацій, оскільки аудіосигнал містить багато параметрів, які можна виміряти та порівняти. Важливою перевагою є можливість працювати з новими композиціями без необхідності накопичення користувацьких даних, що робить його ідеальним для академічних проектів з обмеженими ресурсами.

Гібридні системи поєднують переваги колаборативного та контентного підходів, дозволяючи компенсувати недоліки кожного з них. Такі системи зазвичай демонструють вищу точність рекомендацій, але вимагають

складнішої архітектури та більше обчислювальних ресурсів. Також для них необхідно мати як дані про контент, так і інформацію про користувацькі вподобання.

Системи на основі знань використовують експертні правила та онтології для формування рекомендацій. У контексті музики це можуть бути музикознавчі правила щодо гармонічної сумісності або історичних зв'язків між жанрами. Хоча такі системи можуть надавати обґрунтовані рекомендації, вони потребують значних експертних знань для налаштування та менш гнучкі порівняно з іншими підходами.

Порівняльну характеристику систем рекомендацій наведено у таблиці 2.5.

Таблиця 2.5 – Порівняння систем рекомендацій

Критерій	Колаборативна фільтрація	Content-based фільтрація	Гібридні системи	Системи на основі знань
Вимоги до даних	Велика кількість даних про користувацькі вподобання	Тільки характеристики контенту	Обидва типи даних	Експертні знання та правила
Проблема "холодного старту"	Значна	Відсутня для нового контенту	Частково вирішена	Мінімальна
Обчислювальна складність	Середня до високої	Низька до середньої	Висока	Низька до середньої
Здатність пояснювати рекомендації	Низька	Висока	Середня	Дуже висока
Персоналізація	Висока	Середня	Дуже висока	Середня
Різноманітність рекомендацій	Може бути низькою	Середня	Висока	Залежить від правил
Придатність для музичної галузі	Добра за наявності даних	Відмінна завдяки аудіоознакам	Дуже добра, але складна	Добра, але потребує експертизи
Складність реалізації	Середня	Низька	Висока	Середня до високої
Масштабованість	Проблематична	Добра	Складна	Добра
Адаптивність до зміни вподобань	Висока	Низька до середньої	Висока	Низька

На основі проведеного аналізу можна зробити висновок, що для розробки системи музичних рекомендацій в рамках дипломного проекту найдоцільнішим є використання Content-based фільтрації. Цей вибір обґрунтовується наступними факторами:

- доступність даних: Для Content-based підходу необхідні лише аудіохарактеристики музичних композицій, які можна отримати з таких наборів даних як Million Song Dataset або шляхом безпосереднього аналізу аудіофайлів за допомогою бібліотеки LibROSA;

- відсутність проблеми "холодного старту": Система може ефективно працювати з новими музичними композиціями, для яких ще немає історії прослуховувань;

- прозорість рекомендацій: Content-based підхід дозволяє чітко пояснити, чому певна композиція була рекомендована, на основі схожості її аудіопараметрів з іншими творами;

- технічна простота реалізації: Порівняно з гібридними системами, контентна фільтрація має нижчу складність реалізації, що важливо для академічного проекту з обмеженими термінами;

- відповідність предметній галузі: Музика має багато вимірюваних характеристик (темп, тембр, гармонічний склад, ритмічні патерни), які добре підходять для контентного аналізу.

Хоча колаборативна фільтрація та гібридні системи можуть демонструвати вищу персоналізацію, вони потребують значного обсягу даних про користувацькі вподобання, збір яких виходить за межі можливостей дипломного проекту. Системи на основі знань, хоча і перспективні, вимагають глибокої музикознавчої експертизи для налаштування правил рекомендацій.

Таким чином, Content-based фільтрація представляє собою оптимальний баланс між технічною складністю, вимогами до даних та якістю рекомендацій для академічного проекту з музичними рекомендаціями. Цей

підхід дозволить розробити функціональну систему, яка знаходитиме музичні композиції зі схожими аудіохарактеристиками, що є одним з ключових факторів формування музичних вподобань слухачів.

Проведений порівняльний аналіз демонструє, що для розробки музичної рекомендаційної системи в рамках дипломного проекту найбільш доцільним є використання Content-based фільтрації. Цей підхід забезпечує оптимальний баланс між технічною складністю реалізації, вимогами до даних, якістю рекомендацій та можливістю пояснити принципи роботи системи.

Перевага Content-based фільтрації полягає у можливості безпосередньо аналізувати аудіохарактеристики музичних творів, що дозволяє виявляти неочевидні зв'язки між композиціями на основі їхніх акустичних властивостей. Такий підхід не потребує великих масивів даних про користувацькі вподобання та дозволяє робити рекомендації для нових композицій, для яких ще немає історії прослуховувань.

З огляду на доступність інструментів для аналізу аудіоданих, таких як бібліотека LibROSA, та наявність структурованих наборів даних, таких як Million Song Dataset, Content-based фільтрація є найбільш практичним і реалістичним підходом для реалізації в рамках дипломного проекту. Цей метод забезпечить створення функціональної системи рекомендацій з потенціалом для подальшого розширення та вдосконалення.

2.4 Вибір та обґрунтування середовища розробки

Розробка сучасних веб-застосунків вимагає використання потужних та гнучких середовищ розробки, які підтримують зручну роботу як із клієнтською (JavaScript/TypeScript), так і з серверною (Python/FastAPI) частинами. Важливо, щоб середовище розробки забезпечувало високу продуктивність праці, підтримку систем контролю версій, інтеграцію із

зовнішніми інструментами, автодоповнення коду та інструменти для налагодження.

Для клієнтської частини на основі JavaScript та TypeScript серед популярних середовищ розробки варто розглянути Visual Studio Code, WebStorm та Sublime Text. Для серверної частини на Python із використанням FastAPI найбільш поширеними є Visual Studio Code, PyCharm та Vim (із налаштуваннями для Python-розробки).

Visual Studio Code – це легке, безкоштовне та надзвичайно розширюване середовище розробки, яке підтримує обидві мови –JavaScript/TypeScript та Python – на високому рівні. Завдяки великій кількості розширень (таких як Python, Pylance, Prettier, ESLint, FastAPI extension) і вбудованій підтримці налагодження, автодоповнення та інтеграції з Git, VSCode дозволяє створити єдине інтегроване середовище для розробки обох частин застосунку. Важливою перевагою також є мультиплатформеність та низьке споживання ресурсів.

WebStorm – це комерційне середовище розробки від JetBrains, орієнтоване переважно на фронтенд-технології, особливо JavaScript та TypeScript. Воно має глибоку інтеграцію з багатьма фреймворками, такими як React, Angular та Vue. Незважаючи на високу якість автодоповнення та рефакторингу, WebStorm вимагає окремого налаштування для повноцінної роботи з Python-проектами або використання іншого середовища.

PyCharm – спеціалізоване середовище для розробки на Python, також розроблене JetBrains. PyCharm забезпечує виняткову підтримку Python-проектів, включаючи FastAPI, однак його використання для фронтенд-розробки є менш зручним через обмежену інтеграцію із сучасними інструментами JavaScript/TypeScript.

Sublime Text – дуже легке та швидке текстове середовище розробки, яке через додаткові плагіни може підтримувати як Python, так і

JavaScript/TypeScript. Однак його можливості для налагодження та інтеграції є набагато менш потужними у порівнянні з VSCode або IDE JetBrains.

Vim – потужний текстовий редактор для ентузіастів, який при правильному налаштуванні дозволяє повністю підтримувати розробку як на Python, так і на JavaScript/TypeScript. Проте крута навчальна крива та необхідність глибокої кастомізації роблять його менш практичним вибором для стандартних проектів веб-розробки.

Порівняння основних середовищ розробки наведено у таблиці 2.6.

Таблиця 2.6 – Порівняння середовищ розробки

Критерій	Visual Studio Code	WebStorm	PyCharm	Sublime Text	Vim
Підтримка Python	Відмінна	Обмежена	Відмінна	Задовільна	Висока (з налаштуванням)
Підтримка JavaScript/TypeScript	Відмінна	Відмінна	Обмежена	Задовільна	Висока (з налаштуванням)
Підтримка FastAPI	Є розширення	Потрібна ручна настройка	Відмінна	Через плагіни	Через плагіни
Продуктивність	Висока	Висока (вимоглива до ресурсів)	Висока (вимоглива до ресурсів)	Дуже висока	Дуже висока
Інтеграція з Git	Вбудована	Вбудована	Вбудована	Через плагіни	Через плагіни
Наявність безкоштовної версії	Так	Ні	Так (Community Edition)	Так	Так
Зручність налаштування	Висока	Дуже висока	Висока	Середня	Низька

Аналіз показав, що для одночасної розробки клієнтської частини (на JavaScript/TypeScript з React) та серверної частини (на Python/FastAPI) найкращим вибором є Visual Studio Code. Це середовище об'єднує гнучкість, потужну інтеграцію з необхідними мовами і фреймворками, велику екосистему розширень і високу продуктивність, що дозволяє ефективно працювати над усіма аспектами проекту у єдиному просторі.

2.5 Висновки за розділом 2

У результаті аналізу технологічних рішень для створення веб-застосунку з системою рекомендацій було обґрунтовано вибір мови TypeScript для клієнтської частини та Python для серверної. Для реалізації інтерфейсу обрано React із бібліотекою TanStack Router, а для побудови серверної логіки – FastAPI як сучасний асинхронний фреймворк. Основою рекомендаційної системи стала Content-based фільтрація завдяки оптимальному співвідношенню точності та простоти реалізації.

Середовищем розробки обрано Visual Studio Code, що дозволяє ефективно працювати над обома частинами проєкту в єдиному просторі. Обраний стек технологій забезпечує гнучкість, масштабованість та високу продуктивність розробки дипломного проєкту.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Детальний розгляд архітектури застосунку

Архітектурне рішення базується на класичній клієнт-серверній моделі, у межах якої фронтенд та бекенд реалізовані як окремі, проте логічно взаємопов'язані компоненти. Загальний підхід обраний у вигляді модульного моноліта – усі частини системи взаємодіють тісно всередині єдиного репозиторію та функціонують як єдине цілісне програмне середовище, попри використання різних технологічних стеків. Схему архітектури застосунку можна побачити на рисунку 3.1

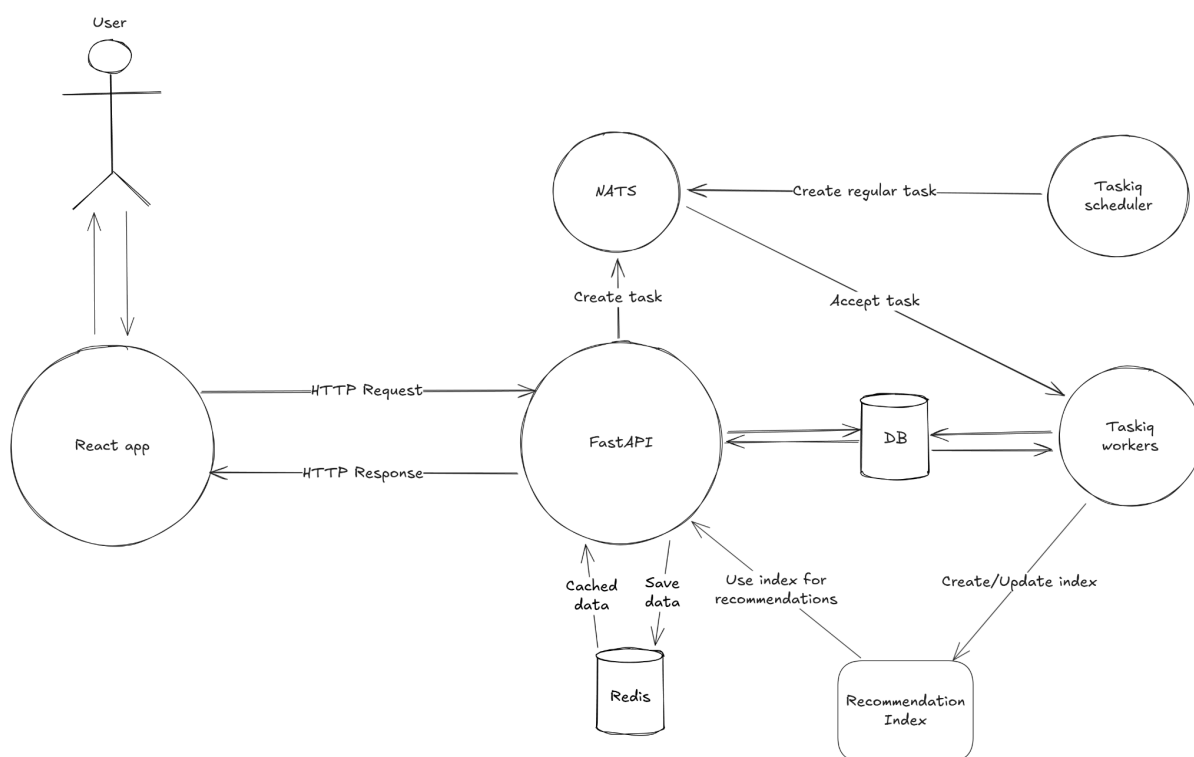


Рисунок 3.1 – Схема архітектури застосунку

Проект має наступну файлову структуру відповідно до клієнтської та серверної частин яку можна побачити на рисунку 3.2.

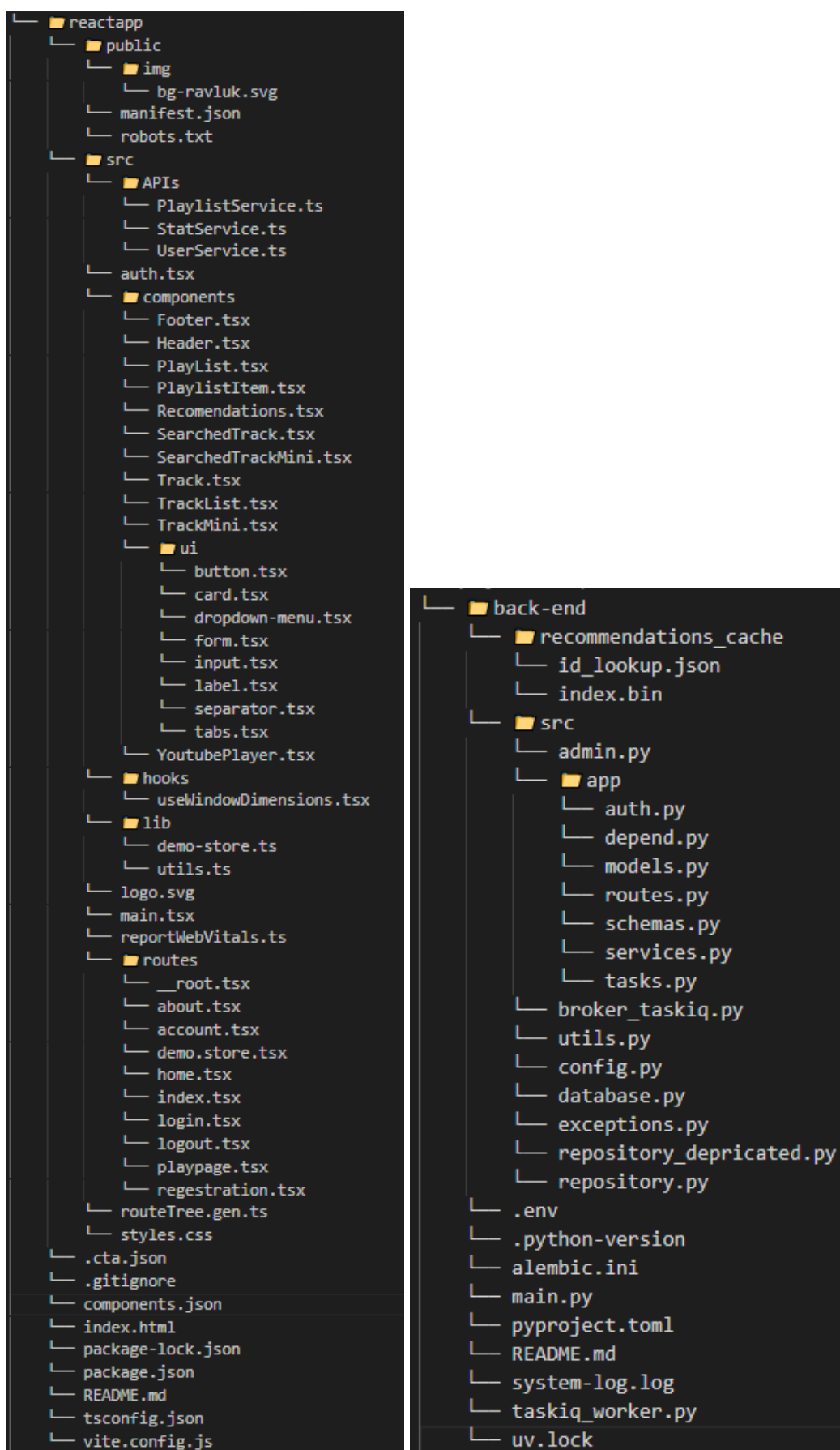


Рисунок 3.2 – Файлова структура клієнтської та серверної частин

Проект поділений на клієнтську (front-end) та серверну (back-end) частини, кожна з яких має власну структуру директорій, що забезпечує логічну організацію коду та зручність підтримки.

Усі файли клієнтської частини зберігаються в каталозі `src`. Основні компоненти інтерфейсу користувача розміщені у папці `components/`. Вона містить як окремі візуальні елементи (наприклад, `Track.tsx`, `PlaylistItem.tsx`, `Recommendations.tsx`), так і підкаталог `ui/`, де зібрано багаторазові компоненти для форми, кнопок, інпутів, вкладок тощо.

Логіка взаємодії з сервером винесена в `APIs/` – тут розташовані сервіси `PlaylistService.ts`, `UserService.ts`, `StatService.ts`, що формують HTTP-запити до API. У `routes/` описані сторінки застосунку: головна (`home.tsx`), вхід (`login.tsx`), реєстрація (`registration.tsx`), акаунт користувача (`account.tsx`) тощо.

Окремі гілки – це `hooks/` для користувацьких хуків, `lib/` для утиліт, а також `auth.tsx` – файл для авторизації, `main.tsx` – точка входу в застосунок, і `routeTree.gen.ts` – автоматично згенероване дерево маршрутів. Загальні стилі зберігаються в `styles.css`.

Серверна частина зосереджена у папці `back-end/`. Основний код розміщений у `src/`. Підкаталог `app/` містить логіку програми: `routes.py` – реалізація API-ендпоїнтів, `auth.py` – обробка авторизації, `schemas.py` – Pydantic-схеми, `models.py` – SQLAlchemy-моделі, `services.py` – бізнес-логіка, `tasks.py` – асинхронні завдання.

Кореневий каталог містить `main.py` – точку входу до застосунку, `config.py` – конфігурацію середовища, `database.py` – підключення до БД, `exceptions.py` – обробку помилок, `taskiq_worker.py` – запуск воркерів для обробки фонових задач, і `broker_taskiq.py` – налаштування брокера повідомлень.

Папка `recommendations_cache/` зберігає попередньо згенерований кеш індексу схожості треків (`index.bin`, `id_lookup.json`). Сервіс використовує бібліотеки ML для контентного аналізу аудіо, зокрема `librosa`. Проект також

включає файл `alembic.ini` для міграцій БД, `.env` для середовища, `pyproject.toml` для залежностей.

3.1.1 Вимоги до деплою та запуску

Для того щоб виконати успішний деплоймент даного веб-застосунку необхідно виконувати наступні вимоги:

1) Для серверної частини необхідні наступні пакети:

- python версій 3.13;
- aiocache 0.12.3;
- alembic 1.15.2;
- asyncpg 0.30.0;
- fastapi[standard] 0.115.12;
- hnswlib 0.8.0;
- itsdangerous 2.2.0;
- librosa 0.11.0;
- loguru 0.7.3;
- pandas 2.2.3;
- pydantic-settings 2.9.1;
- pydantic[email] 2.11.3;
- pydub 0.25.1;
- pyjwt версії 2.10.1;
- pysoundfile версії 0.9.0.post1;
- python-dotenv версії 1.1.0;
- pytubefix версії 8.13.1;
- sqlalchemy версії 0.20.1;
- sqlalchemy[asyncio] версії 2.0.40;
- taskiq версії 0.11.17;
- taskiq-nats версії 0.5.1;

- uuid6 версії 2024.7.10;
- uvicorn[all] версії 0.34.2.

2) Для клієнтської частини необхідні наступні пакети:

- node-js версії 20.11.0;
- radix-ui/react-dropdown-menu версії 2.1.14;
- radix-ui/react-label версії 2.1.6;
- radix-ui/react-separator версії 1.1.6;
- radix-ui/react-slot версії 1.2.2;
- radix-ui/react-tabs версії 1.1.11;
- tailwindcss/vite версії 4.0.6;
- tanstack/react-router версії 1.114.3;
- tanstack/react-router-devtools версії 1.114.3;
- tanstack/react-store версії 0.7.0;
- tanstack/router-plugin версії 1.114.3;
- tanstack/store версії 0.7.0;
- axios версії 1.9.0;
- lucide-react версії 0.510.0;
- react версії 19.0.0;
- react-youtube версії 10.1.0;
- tailwindcss версії 4.0.6;
- zod версії 3.24.4.

Також на сервері повинен бути доступ до NATS та Redis для нормальної роботи серверної частини веб-застосунку.

Виконання цих вимог в купі з вимогами до серверного обладнання які описані у пункті 1.3.4 забезпечує нормальну роботу веб-застосунку.

3.2 Робота з базою даних

3.2.1 Структура бази даних

Внаслідок аналізу предметної області та висунутих вимог було розроблено структуру бази даних що є достатньою для виконання поставлених цілей та достатньо зрозумілою щоб її можна було назвати простою.

У предметній області можна відразу зазначити наступні сутності:

- користувач;
- музична композиція;
- плейліст.

Це базові структури у обраній предметній області, але через функціональні вимоги до проекту також потрібно зазначити наступні сутності бази даних:

- сесії;
- статистика прослуховувань;
- проміжна сутність “музична композиція – плейлист”;
- історія прослуховувань;
- результати аналізу музичних композицій.

Цей список дає змогу зрозуміти з якими сутностями потрібно буде працювати під час розробки та підтримки проекту. У наступних таблицях 3.1 – 3.8 буде описано структуру цих сутностей.

Таблиця 3.1 – Сутність користувач.

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор користувача
username	String(50)	Так	Унікальне ім'я користувача
email	String(100)	Так	Унікальна електронна адреса
password	String(255)	Так	Хеш пароля користувача
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Таблиця 3.2 – Сутність музична композиція.

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор треку
title	String	Так	Назва треку
artist	String	Так	Виконавець
duration	Integer	Так	Тривалість у секундах
yt_id	String(25)	Так	Унікальний YouTube-ідентифікатор
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Таблиця 3.3 – Сутність плейліст.

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор плейлиста
name	String(100)	Так	Назва плейлиста
description	String(500)	Ні	Опис плейлиста
tracks_amount	Integer	Ні	Кількість треків у плейлісті
owner_id	UUID	Так	Зовнішній ключ на users.id
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Таблиця 3.4 – Сутність сесія.

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор сесії
user_id	UUID	Так	Зовнішній ключ на users.id
expires_at	DateTime	Так	Дата і час закінчення дії сесії
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Таблиця 3.5 – Сутність статистика прослуховувань.

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор запису
yt_id	String	Так	Зовнішній ключ на tracks.yt_id
user_id	String	Так	Зовнішній ключ на users.id
track_playback	Integer	Так	Кількість прослуховувань треку
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Таблиця 3.6 – Сутність результат аналізу музичних композицій.

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор
yt_id	String	Так	Зовнішній ключ на tracks.yt_id
chroma_mean	Float	Так	Середнє значення хроматичності
chroma_var	Float	Так	Варіація хроматичності
rms_mean	Float	Так	Середнє значення кореня середнього квадрата
rms_var	Float	Так	Варіація RMS
spectral_centroids_*	Float	Так	Центроїд спектру (середнє та варіація)
spectral_bandwidth_*	Float	Так	Ширина спектру (середнє та варіація)

Продовження таблиці 3.6.

Назва поля	Тип даних	Обов'язковість	Опис
spectral_rolloff_*	Float	Так	Roll-off спектру
spectral_contrast_*	Float	Так	Контраст спектру
zero_crossing_rate_*	Float	Так	Частота переходів через нуль
tempo	Float	Так	Темп
mfcc_N_mean, mfcc_N	Float	Так	MFCC-коефіцієнти від 1 до 20 (середня варіація)
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Таблиця 3.7 – Проміжна сутність “музична композиція – плейлист”

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор
playlist_id	UUID	Так	Ідентифікатор плейлиста
track_id	UUID	Так	Ідентифікатор музичної композиції
position	int	Так	Позиція у плейлисті
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Таблиця 3.8 – Сутність історія прослуховувань

Назва поля	Тип даних	Обов'язковість	Опис
id	UUID	Так	Унікальний ідентифікатор
user_id	UUID	Так	Ідентифікатор користувача
yt_id	String	Так	Ідентифікатор композиції на Youtube
created_at	DateTime	Так	Дата створення запису
updated_at	DateTime	Так	Дата останнього оновлення

Розроблена структура бази даних не є складною але повністю покриває усі потреби такого застосунку. На рисунку 3.1 можна побачити структуру зв'язків між сутностями.

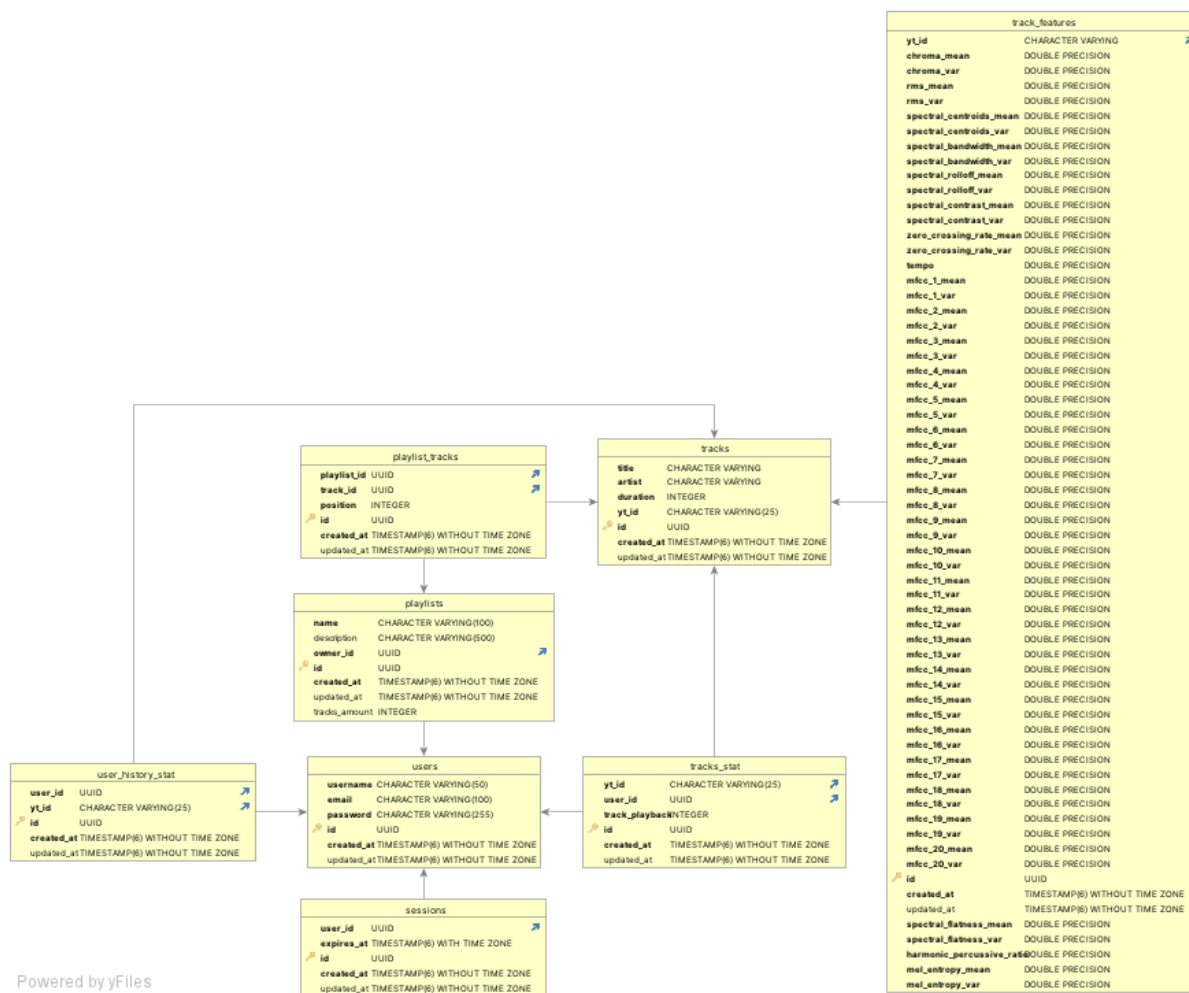


Рисунок 3.1 – Структура бази даних

3.2.2 Програмна взаємодія з базою даних

Для надійної та безперервної взаємодії з базою даних з коду програми у даному проєкті буде використовуватися ORM (Object-Relational Mapping), а саме SQLAlchemy (SQLA) як найпоширеніша та найсучасніша ORM для Python.

Щоб налагодити взаємодію для початку потрібно передати до SQLA посилання для підключення до бази даних. Далі необхідно створити рушій (engine) через який буде відбуватися взаємодія з базою даних.

Після того як був налаштований рушій необхідно створити програмні моделі таблиць за допомогою SQLA. У SQLA для створення моделей використовується Base (declarative_base), після її оголошення створюється клас який успадковується від Base та потребує мінімального налаштування, а саме визначити ім'я таблиці та поля таблиці.

```
class ModelName(Base):
    __tablename__ = "model_names"
    field: Mapped[field_type]
    ...
```

Таке оголошення таблиць дозволяє дуже просто та ефективно створювати, оновлювати, редагувати та слідкувати за відношеннями у таблицях.

Для того щоб реалізувати більше потенціалу роботи з SQLA було створено шаблонний клас який реалізує всі основні операції CRUD та може бути легко розширений додатковими методами. Приклад класу з 1 методом:

```
class ModelProtocol(Protocol):
    __tablename__: object

SQLAlchemyModel = TypeVar("SQLAlchemyModel", bound=ModelProtocol)
PydanticSchema = TypeVar("PydanticSchema", bound=BaseModel)

class CrudMeta(type):
    pass

class AsyncCrud(Generic[SQLAlchemyModel], metaclass=CrudMeta):
    model_class: Type[SQLAlchemyModel]
```

```

def __init__(self, model: Type[SQLAlchemyModel]):
    self.model_class = model

@classmethod
async def create(
    cls,
    session: AsyncSession,
    data: PydanticSchema,
) -> SQLAlchemyModel:
    """Create a single entity"""
    try:
        db_model = cls.model_class(**data.model_dump(exclude_unset = True))
        session.add(db_model)
        await session.commit()
        await session.refresh(db_model)
        return db_model
    except IntegrityError as e:
        await session.rollback()
        raise IntegrityConflictException(
            f"{cls.model_class.__tablename__} conflicts with existing data:
{e}",
            ) from e
    except Exception as e:
        await session.rollback()
        raise SnippetException(f"Failed to create
{cls.model_class.__tablename__}: {e}") from e

def crud_factory(model_class: Type[SQLAlchemyModel]) ->
Type[AsyncCrud[SQLAlchemyModel]]:
    if model_class is None:
        raise TypeError("Crud class must be initialized with 'model_class'")

    new_class_name = f"{model_class.__name__}Crud"

    new_cls = CrudMeta(new_class_name, (AsyncCrud,), {"model_class":
model_class})
    return cast(Type[AsyncCrud[SQLAlchemyModel]], new_cls)

```

У цьому прикладі за допомогою функції `crud_factory` ми можемо передати SQLA модель, наприклад `User` який був продемонстрований вище та отримати клас у якого буде метод `create` який буде створювати об'єкт `User` у базі даних у відповідності до тих даних які були передані у параметрі `data` та повертати об'єкт класу `User`. Тип `PydanticSchema` говорить о класі який успадковується від `BaseModel` з бібліотеки `pydantic` яка є один из основних модулів `FastAPI`.

Варто зазначити що за допомогою використання мета класу та шаблонів досягається підтримка автодоповнення та розуміння середою розробки який саме об'єкт повертається.

3.3 Створення RESTful API

Головною частиною серверної частини можна назвати веб-сервер який приймає та обробляє усі запити що надходять від клієнтів та реалізує `RESTful API`.

`RESTful API` – це веб-інтерфейс програмування, який дотримується архітектурних принципів `REST` (`Representational State Transfer`) і використовує стандартні `HTTP`-методи (`GET`, `POST`, `PUT`, `DELETE`) для взаємодії між клієнтом і сервером, забезпечуючи просту, масштабовану та зрозумілу модель обміну даними, зазвичай у форматі `JSON` [6].

У даному проєкті ця частина реалізується за допомогою `FastAPI` - мікрофреймоврку на основі `starlet` та `pydantic`.

Для створення такого застосунку спочатку потрібно створити файл `main.py` у якому оголосити застосунок як об'єкт `FastAPI`.

У базовому варіанті цього достатньо для роботи, тепер щоб наш сервер зміг приймати та обробляти запити до `app` потрібно додати об'єкт роутера за допомогою якого ми оголосимо певну функцію як обробник запита.

У такому базовому варіанті ми додали до нашого застосунку роутер який може обробляти запит та повертати до клієнта повідомлення.

Функції на які був викликаний декоратор `@router.get()` можуть приймати різноманітні аргументи, наприклад параметри запита, `cookie`, об'єкти `Request` та `Response`, за допомогою `pydantic` та схем можна приймати `body` параметри та багато чого іншого.

Потрібно зазначити що декоратор `@router.get()` має усі варіації запитів як `get`, `post`, `put`, `patch`, `delete`.

Однією з важливіших речей при створенні RESTful API є налаштування CORS за допомогою яких можна налаштовувати правила взаємодії з вашим API. У FastAPI це реалізується через додавання до застосунку “middleware” у якому передається об'єкт `CORSMiddleware` та налаштовуються те які “origin” можуть відправляти запити, чи дозволено використовувати в запитах додаткову інформацію, які дозволені методи запитів та які заголовки запитів.

Для запуску сервера потрібно викликати команду `fastapi dev` знаходясь у віртуальному середовищі пайтон.

Потрібно зазначити що FastAPI впроваджує автоматичну документацію маршрутів, цей функціонал доступен за адресою `/docs` коли сервер працює. Він документує всі маршрути у застосунку та дає змогу легко тестувати їх.

За допомогою FastAPI ми створюємо усі необхідні маршрути у застосунку та створюємо функціонал відповідно до функціональних вимог, створюємо модуль авторизації який буде керувати сесіями та надавати інформацію про користувачів у функції обробники за допомогою чого стає легко опрацьовувати індивідуальний контент для кожного користувача, будь то плейлисти чи рекомендації. Та створюємо модуль сервісу який буде обробляти дані від функцій обробників та надавати можливість керувати логікою застосунку.

3.4 Система рекомендацій

Для створення системи рекомендацій потрібно пройти наступні кроки:

- отримання аудіо музичних композицій;
- аналіз аудіо;
- створення індексу;
- рекомендація на основі кількох музичних композицій.

Для отримання аудіо використовується бібліотека `pytube` за допомогою якої ми завантажуюмо аудіо дані у пам'ять.

На виході з функції отримуємо об'єкт `AudioData` у якому знаходяться `audio_data` та `samplerate`. З цими даними буде працювати наступна функція яка використовує бібліотеку `librosa` для аналізу аудіо, а саме вона витягне з `audio_data` такі характеристики як:

- `chroma_mean` та `chroma_var`;
- `rms_mean` та `rms_var`;
- `spectral_centroids_mean` та `spectral_centroids_var`;
- `spectral_bandwidth_mean` та `spectral_bandwidth_var`;
- `spectral_rolloff_mean` та `spectral_rolloff_var`;
- `spectral_contrast_mean` та `spectral_contrast_var`;
- `zero_crossing_rate_mean` та `zero_crossing_rate_var`;
- `tempo`;
- `spectral_flatness_mean` та `spectral_flatness_var`;
- `harmonic_percussive_ratio`;
- `mel_entropy_mean` та `mel_entropy_var`;
- `mfcc_1_mean` та `mfcc_1_var`, від 1 до 20.

У сукупності 60 аудіо характеристик.

В результаті отримуємо словник з усіма фічами треку які потім зберігаємо у базу даних. Ці дані музичних композицій використовуються для побудови Index з `hnswlib` - це бібліотека яка впроваджує `NearestNeighbors` з

певними оптимізаціями за рахунок яких стає неймовірно швидкою при пошуку. Також необхідно створювати мапу музичних композицій бо `hnswlib` приймає лише числові `id`.

Ось алгоритм який реалізує створення `Index` та мапу музичних композицій та їх збереження:

- 1) Перетворити усі композиції на їх векторне представлення та нормалізувати.

- 2) Створити мапу композицій де ключ – номер, а значення – ідентифікатор Youtube.

- 3) Створити об'єкт `Index` з типом простору “cosine” та розмірністю відповідно розмірності входящих векторів.

- 4) Ініціалізувати `Index` та додати в нього усі вектори та список позначень.

- 5) Зберігти `Index` та мапу композицій.

Для того щоб мати змогу проводити рекомендації за кількома композиціями потрібно агрегувати їхні властивості до середнього та знаходити декілька сусідів які найбільш схожі але також гарно додавати кілька композицій які більш відрізняються. Ось алгоритм який це реалізує:

- 1) Якщо плейліст порожній – повертаємо порожній список.

- 2) Отримуємо вектори ознак для всіх треків у плейлісті.

- 3) Обчислюємо середній вектор та нормалізуємо його.

- 4) Знаходимо найближчі треки до середнього вектора через `Index`.

- 5) Відбираємо ті з них, яких немає у плейлісті, поки не наберемо основну частину рекомендацій.

- 6) Для кожного треку з плейліста шукаємо найближчі треки та рахуємо, які найчастіше зустрічаються.

- 7) Вибираємо найпопулярніші з них для додавання різноманітності.

- 8) Об'єднуємо основні рекомендації з різноманітними та повертаємо результат.

3.5 Робота з відкладеними завданнями

Було розглянуто обробку запитів до серверу але навіть у сучасному світі з дуже потужними комп'ютерами є задачі які потребують значний по міркам HTTP запиту час, наприклад витяг аудіо фіч з музичних композицій. Для того щоб обійти це обмеження використовують відкладенні завдання які будуть виконані іншою програмою незалежно від основного сервера.

У даному проєкті буде використовуватися менеджер відкладених завдань Taskiq та сервіс NATS який буде виконувати роль черги у нашій архітектурі.

Завдяки обраній мові програмування створювати функції що будуть виконуватися як відкладенні дуже просто з використанням декораторів з Taskiq але з початку потрібно створити об'єкт брокер який буде описувати підключення до черги, наприклад:

```
broker = NatsBroker(
    servers=[
        "nats://localhost:4222",
    ],
    queue="track_analisis",
)

@broker.task
async def track_features(yt_id: str):
    service = RecomendationService()
    a_dta = service.download_audio(yt_id)
    if not a_dta:
        raise ValueError(f"yt_id is invalid: {yt_id}")
    logger.info("✅ Track download complete. Start analysis...")

    features = service.get_sample_features(a_dta)
    logger.info("✅ Track analysis complete. Saving results...")
    features["yt_id"] = yt_id
```

```

    async with async_session_maker() as session:
        await crud_features.create(session,
TrackFeatures.model_validate(features))
        logger.info("✅ Result seved.")

logger.info("✅ Task complete.")

```

При використанні декоратора `@broker.task` відбувається реєстрація функції як обробника. Щоб надати завдання у коді потрібно викликати наступну функцію:

```
await track_features.kiq(yt_id)
```

Саме функція `kiq` відповідає за створення завдання.

Для того щоб налаштувати повторення завдань у певний проміжок часу у `Taskiq` використовується `scheduler`, приклад оголошення:

```

scheduler = TaskiqScheduler(
    broker=broker,
    sources=[LabelScheduleSource(broker)],
)

```

Через передавання у `scheduler` об'єкта `broker` реєстрація обробників відбувається тим же чином що і звичайних завдань але для повторення у деякий час цей час необхідно вказати при оголошенні у форматі `cron`, наприклад повторювати у кожну п'яту хвилину:

```
@broker.task(schedule=[{"cron": "*/5 * * * *"}])
```

Після оголошення необхідно імпортувати `broker`, `scheduler` та функції які були зареєстровані у окремий файл, наприклад:

```
from src.broker_taskiq import broker, scheduler
from src.app.tasks import build_recommendation_index, track_features
```

Це дозволяє запускати виконавців завдань за допомогою описаних у документації команд, у цьому випадку команди будуть наступні:

```
taskiq worker taskiq_worker:broker
taskiq scheduler taskiq_worker:scheduler
```

Також необхідно у застосунку FastAPI налагодити запуск та зупинку broker, наприклад:

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    await broker.startup()
    yield
    await broker.shutdown()

app = FastAPI(lifespan=lifespan)
```

Після запуску виконавців, черги та застосунку можна говорити про вирішення проблеми відкладених завдань у цьому проекті.

3.6 Створення веб-клієнту

Для створення веб-клієнту у цьому проекті за допомогою React і Tanstack Router необхідно створити проект командою `npx create-vite@latest` та обрати у меню React і Tanstack Router після чого переходити до створення кодової бази.

Tanstack Router – це сучасний, типобезпечний маршрутизатор для React, що дозволяє декларативно описувати маршрути у вигляді дерева, забезпечуючи глибоку інтеграцію з TypeScript. Він підтримує вкладені макети (layouts), завантажувачі даних (loaders), дії (actions) та забезпечує автоматичне виведення типів для параметрів URL, query-параметрів і даних. Завдяки цьому зменшується кількість помилок і покращується розробницький досвід, особливо у великих SPA або SSR-застосунках.

Також Tanstack Router вміє автоматично створювати маршрути зпираючись на назви файлів сторінок.

Для створення повноцінного клієнту необхідно створити сторінки застосунку, компоненти які цю сторінку будуть наповнювати та сервіси звертання до серверу для роботи з даними.

Спочатку створимо сторінку у нашому клієнті, у React це функція що повертає jsx елемент який є програмною прокладкою над звичайним HTML, приклад простої сторінки:

```
import { createFileRoute } from '@tanstack/react-router'

export const Route = createFileRoute('/home') ({
  component: RouteComponent,
})

function RouteComponent() {
  return <div>Hello "/home"!</div>
}
```

У цьому прикладі також створено Route який утворюється за допомогою createFileRoute у якому відбувається опис роботи Route, одним з параметрів є component який буде повертатися користувачу та він є обов'язковим.

У jsx файлах навіть такий компонент як `RouteComponent` можна використовувати як просту частину HTML, взявши його у скобки: `<RouteComponent />`. Та він буде відпрацьовувати як можна сподіватися, а тобто повертати свій компонент: `<div>Hello "/home"!</div>`

Також потрібно зазначити що у цих функціях створюється логіка відображення та контент який вони повертаються є динамічним бо React піклується о оновленні компонентів самостійно.

Завдяки цьому механізму стає можливо розбивати веб-клієнт на компоненти та зручно їми маніпулювати для досягнення необхідного вигляду інтерфейсу.

Розглянемо звертання до сервера за даними, для того щоб звернутися до сервера клієнту потрібно відправити HTTP запит, для цього використовується функція `fetch` чи більш просунена бібліотека `axios`.

Для створення сервісу запитів необхідно створити клас да додати до нього статичні методи що будуть реалізовувати ті чи інші запити, наприклад:

```
export type LoginData = {
  email: string;
  password: string;
};

export default class UserService {

  static async login(data: LoginData): Promise<boolean> {
    try {
      await axios.post(`${BASIC_API_URL}/login`, data, {
        withCredentials: true,
      });
      return true;
    } catch (error) {
      return false;
    }
  }
}
```

```
}
```

```
}
```

У цьому класі є метод `login` який приймає об'єкт типу `LoginData` та повинен мати поля `email` типу `string` та `password` типу `string`, це необхідно через використання `TypeScript` який вимагає відповідність полів та типів але забезпечує безпеку з точки зору типізації.

У результаті можливо створювати сторінки, компоненти та сервіси з яких буде створено веб-клієнт.

3.7 Висновки за розділом 3

В цьому розділі проведено огляд архітектурі, розібрана структура бази даних, створення універсального програмного засобу для взаємодії з базою даних, розібрано як створювати застосунок на `FastAPI` та підключати виконання відкладених завдань. Також розібрано основні компоненти для створення веб-клієнту.

4 ТЕСТУВАННЯ ЗАСТОСУНКУ

Тестування веб-застосунків малого розміру яким є даний проект зазвичай відбувається у ручному режимі з використанням чек-листів та списків вимог.

Чек-листи створюються з переліка функціональних вимог та являються у табличному вигляді для зручності проведення тестування.

У цьому проекті чек-листи описуються у таблицях 4.1 – 4.3.

Таблиця 4.1 – Загальний чек-лист

ID	Опис	Результат	Примітка
ТС-01	Перевірка реєстрації нового користувача	Пройдено	
ТС-02	Перевірка входу існуючого користувача	Пройдено	
ТС-03	Створення нового плейлиста	Пройдено	
ТС-04	Редагування існуючого плейлиста	Пройдено	
ТС-05	Видалення плейлиста	Пройдено	
ТС-06	Додавання валідного URL YouTube	Пройдено	
ТС-07	Додавання невалідного URL (очікувана помилка)	Пройдено	
ТС-08	Витяг аудіо з доданого відео	Пройдено	
ТС-09	Перевірка релевантності рекомендацій	Пройдено	
ТС-10	Пошук треку по назві	Пройдено	
ТС-11	Відтворення відео у вбудованому плеєрі	Пройдено	
ТС-12	Перевірка відображення назви, виконавця, тривалості	Пройдено	
ТС-13	Перевірка запису прослуханого треку	Пройдено	
ТС-14	Відображення виконавців	Пройдено	

Таблиця 4.2 – Чек-лист клієнтської частини

ID	Опис	Результат	Примітка
ТС-15	Оцінка інтуїтивності створення/управління	Пройдено	
ТС-16	Відтворення музики через плеєр	Пройдено	
ТС-17	Наявність блоку рекомендацій	Пройдено	
ТС-18	Коректність роботи форми URL	Пройдено	
ТС-19	Тестування вигляду на мобільному/десктопі	Пройдено	
ТС-20	Перевірка відображення додавання нових треків до системи	Пройдено	
ТС-21	Відображення ступеня схожості треків у рекомендаціях відповідно до плейлісту	Пройдено	
ТС-22	Тест пошуку треків через інтерфейс	Пройдено	
ТС-23	Перевірка повідомлень про помилку	Пройдено	
ТС-24	Відображення кількості відтворень, дати додавання	Пройдено	

Таблиця 4.3 – Чек-лист серверної частини

ID	Опис	Результат	Примітка
ТС-25	Тестування безпеки сесій і паролів	Пройдено	
ТС-26	Перевірка збереження в базі даних	Пройдено	
ТС-27	Отримання метаданих через API	Пройдено	
ТС-28	Перевірка цілісності витягнутого аудіо	Пройдено	
ТС-29	Оцінка точності рекомендацій	Пройдено	
ТС-30	Тестування всіх кінцевих точок API	Пройдено	
ТС-31	Перевірка швидкості доступу до кешованих даних	Пройдено	
ТС-32	Логування взаємодії користувачів	Пройдено	
ТС-33	Симуляція перевищення квоти API	Пройдено	
ТС-34	Перевірка правильного повідомлення про помилку	Пройдено	

Проведені тести охоплюють функціональне тестування основного сценарію роботи системи, тестування інтерфейсу користувача, інтеграційне

тестування взаємодії компонентів, а також перевірки на обробку помилок. Окремі тести охоплюють аспекти продуктивності й надійності системи. Основним методом тестування був метод чорного ящика, який дозволяє оцінити поведінку системи на основі зовнішніх проявів без занурення в код. Перелік типів тестування до проведених тестів можна побачити у таблиці 4.4.

Таблиця 4.4 – Перелік типів тестування

Тип тестування	ID тест-кейсів
Функціональне	ТС-01 – ТС-05, ТС-06, ТС-09 – ТС-10, ТС-12 – ТС-14
Інтеграційне	ТС-06, ТС-11, ТС-27 – ТС-32
UI/UX тестування	ТС-15 – ТС-19, ТС-21 – ТС-22, ТС-24
Негативне тестування	ТС-07, ТС-23, ТС-33, ТС-34
Безпекове (auth, БД)	ТС-25, ТС-26
Нефункціональне (кеш, API)	ТС-31, ТС-33

Тестування проводилося вручну, із використанням чек-листів і сценаріїв, що охоплюють основні функціональні та нефункціональні вимоги.

Загалом веб-застосунок продемонстрував стабільну та передбачувану поведінку. Інтерфейс виявився інтуїтивно зрозумілим навіть для користувачів без спеціальної підготовки. Результати тестування підтвердили, що система виконує свої функції у повному обсязі та може бути використана до подальшого використання або розвитку. Більш детальний опис тест-кейсів у додатку Б.

ВИСНОВКИ

Результатом проекту є повністю робочий та протестований веб-застосунок який готовий до пакування у контейнери та розгортання на сервері. Під час виконання проекту було виконано наступні етапи розробки:

- аналіз предметної області;
- вибір та обґрунтування програмних засобів;
- програмна реалізація веб-застосунку;
- ручне тестування за тейст-кейсами.

Розроблена система рекомендацій базується на підході Content-based фільтрації, що дозволяє аналізувати аудіохарактеристики музичних треків за допомогою бібліотеки Librosa. Такий підхід виявився ефективним для персоналізації рекомендацій навіть без наявності великих обсягів даних про вподобання користувачів.

Для клієнтської використано мову TypeScript та фреймворк React із маршрутизатором TanStack Router, а для серверної – мову Python та асинхронний фреймворк FastAPI. Це забезпечило високу продуктивність, масштабованість та зручність розробки. Тестування проводилося вручну відповідно до чек-листів і охоплювало функціональні, інтеграційні та сценарні кейси.

Загалом, потрібно зазначити, що у дипломному проекті було продемонстровано не лише теоретичні знання, а й практичні навички розробки повнофункціонального веб-застосунку з сучасною архітектурою та можливістю подальшого вдосконалення. Створений застосунок може бути використаний як основа для подальших досліджень або комерційного впровадження у сфері рекомендаційних сервісів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Covington P., Adams J., Sargin E. Deep neural networks for YouTube recommendations // Proceedings of the 10th ACM Conference on Recommender Systems. – 2016. – С. 191–198.
2. Jacobson K., Murali V., Newett E., Whitman B., Yon R. Music personalization at Spotify // Proceedings of the 10th ACM Conference on Recommender Systems. – 2016. – С. 373–373.
3. Linden G., Smith B., York J. Amazon.com recommendations: Item-to-item collaborative filtering // IEEE Internet Computing. – 2003. – Т. 7, № 1. – С. 76–80.
4. Tornado Web Server [Електронний ресурс]. – Режим доступу: <https://www.tornadoweb.org/en/stable/>.
5. Рекомендаційна система [Електронний ресурс] // Вікіпедія. – Режим доступу: https://uk.wikipedia.org/wiki/Рекомендаційна_система.
6. Філдінг Р. Архітектурні стилі та проектування програмного забезпечення для мережі: дис. ... канд. наук: 00.00.00 / Рой Томас Філдінг. – Ірвін: Каліфорнійський ун-т, 2000. – 162 с.
7. Zhao Z., Hong L., Wei L., Chen J., Nath A., Andrews S. та ін. Recommending what video to watch next: a multitask ranking system // Proceedings of the 13th ACM Conference on Recommender Systems. – 2019. – С. 43–51.
8. Ricci F., Rokach L., Shapira B. Recommender systems handbook. 3rd ed. – Springer, 2022.
9. Aggarwal C. C. Recommender systems: The textbook. – Springer, 2016.

ДОДАТОК А
Посилання на репозиторій

Посилання на репозиторій з повним кодом веб-застосунку:
<https://github.com/DustGalaxy/diplom>

ДОДАТОК Б
Опис тест-кейсів

Таблиця Б.1 – Опис тест-кейсів

ID	Назва тесту	Суть перевірки	Тип тестування
ТС-01	Реєстрація нового користувача	Перевірка створення облікового запису через email/пароль	Функціональне
ТС-02	Вхід користувача	Перевірка автентифікації з існуючим обліковим записом	Функціональне
ТС-03	Створення плейлиста	Тест на створення нового плейлиста	Функціональне
ТС-04	Редагування плейлиста	Оновлення назви чи опису плейлиста	Функціональне
ТС-05	Видалення плейлиста	Видалення наявного плейлиста	Функціональне
ТС-06	Додавання URL	Вставлення коректного посилання YouTube для завантаження треку	Інтеграційне
ТС-07	Некоректний URL	Очікуване повідомлення про помилку при невалідному посиланні	Негативне
ТС-08	Витяг аудіо з відео	Перевірка роботи модуля Librosa	Функціональне
ТС-09	Генерація рекомендацій	Перевірка точності рекомендованих треків на основі вмісту	Функціональне
ТС-10	Пошук треку	Пошук музики за назвою через інтерфейс	Функціональне
ТС-11	Відтворення відео	Перевірка роботи вбудованого плеєра YouTube	Інтеграційне
ТС-12	Відображення метаданих	Назва, виконавець, тривалість треку	Функціональне
ТС-13	Історія прослуханого	Перевірка, чи записується факт прослуховування	Функціональне
ТС-14	Панель виконавців	Відображення популярних авторів	Функціональне
ТС-15	Інтерфейс плейлистів	Наскільки інтуїтивним є управління плейлистами	UI/UX
ТС-16	Відтворення через плеєр	Можливість слухати треки через плеєр	UI/UX
ТС-17	Рекомендації поряд з плейлистом	Візуальна наявність блоку рекомендацій	UI/UX
ТС-18	Робота форми додавання URL	Перевірка валідації введеного посилання	UI/UX
ТС-19	Адаптивний дизайн	Коректне відображення на різних екранах	UI/UX
ТС-20	Додавання нових треків	Перевірка, чи з'являються нові треки в інтерфейсі	UI/UX
ТС-21	Схожість треків	Відображення показника схожості в рекомендаціях	UI/UX
ТС-22	Пошук через інтерфейс	Перевірка зручності користувацького пошуку	UI/UX
ТС-23	Обробка помилки URL	Повідомлення про помилку на невалідне посилання	Негативне
ТС-24	Аналітика треків	Вивід статистики: дата додавання, кількість відтворень	UI/UX

Продовження таблиці Б.1.

ID	Назва тесту	Суть перевірки	Тип тестування
ТС-25	Безпека автентифікації	Сесії, шифрування паролів	Безпекове
ТС-26	Збереження даних	Перевірка запису користувача, плейлистів у БД	Безпекове
ТС-27	YouTube API	Отримання метаданих по відео	Інтеграційне
ТС-28	Аудіоаналіз	Цілісність витягнутого аудіо (Librosa)	Функціональне
ТС-29	Рекомендаційний модуль	Точність рекомендацій на основі фіч	Функціональне
ТС-30	REST API	Відповідність і стабільність ендпоінтів API	Інтеграційне
ТС-31	Кешування	Швидкість доступу до збережених даних	Нефункціональне
ТС-32	Логування	Запис подій взаємодії користувача	Інтеграційне
ТС-33	Перевищення квоти API	Перевірка обробки обмеження YouTube API	Нефункціональне
ТС-34	Помилка при невдалому аналізі	Повідомлення про помилку при обробці відео	Негативне

ДОДАТОК В
Слайди презентації

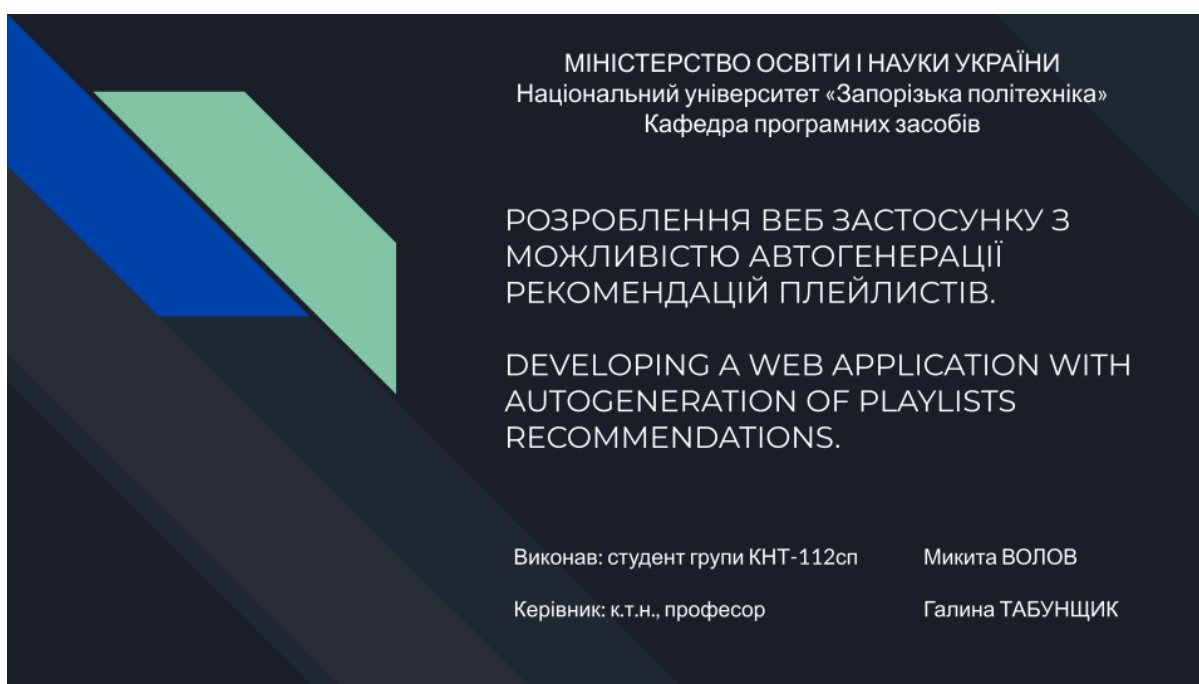


Рисунок В.1 – Слайд 1

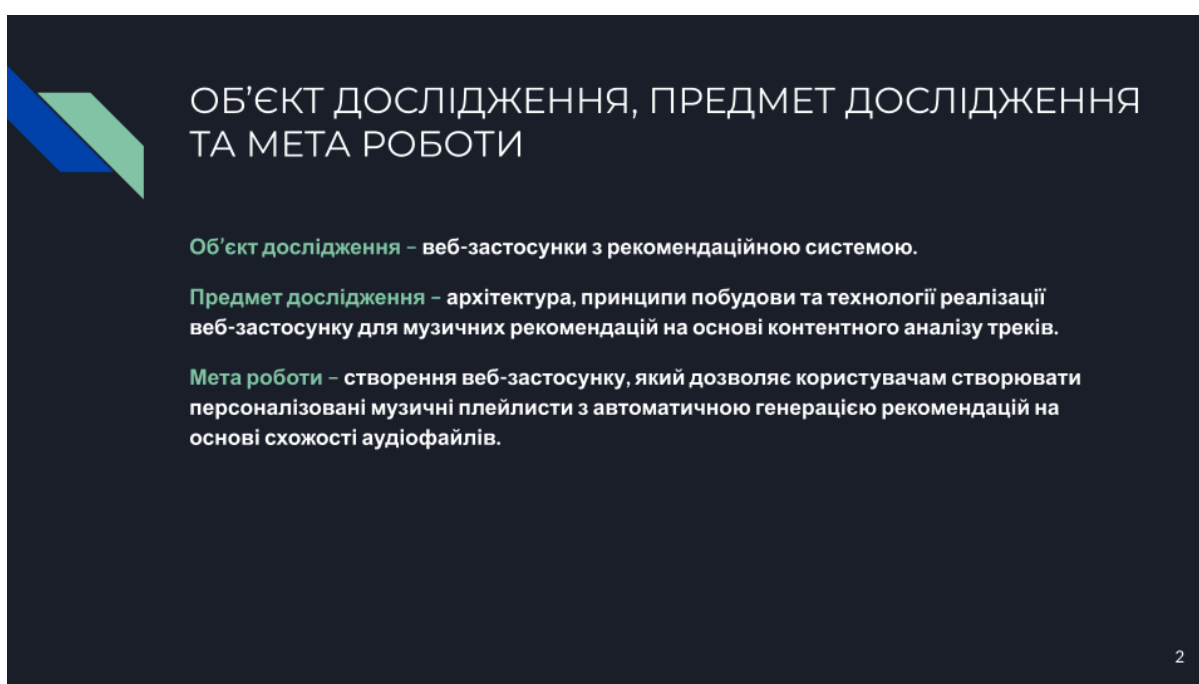


Рисунок В.2 – Слайд 2

ЗАДАЧІ ДО РОБОТИ

- виконати аналіз предметної області та програмних засобів для створення веб-застосунку
- розробити архітектуру взаємодії компонентів системи
- спланувати та розробити структуру бази даних
- розробити програмне забезпечення для серверної частини системи
- розробити програмне забезпечення для клієнтської частини системи
- розробити систему рекомендацій на основі вмісту
- виконати тестування основних частин системи

3

Рисунок В.3 – Слайд 3

ВИБІР МОВИ ПРОГРАМУВАННЯ ДЛЯ КЛІЄНТСЬКОЇ ЧАСТИНИ

Критерій	JavaScript	TypeScript
Типізація	Динамічна, слабка	Статична, сильна
Обробка помилок	Переважно виявлення у рантаймі	Виявлення помилок на етапі компіляції
Надійність у великих проєктах	Низька: висока ймовірність неочевидних помилок	Висока: контроль типів зменшує кількість помилок
Зручність підтримки	Обмежена: важко масштабувати код	Висока: спрощена підтримка завдяки типам та інтерфейсам
Інструментальна підтримка	Базова: автодоповнення залежить від середовища	Повна: розширене автодоповнення, рефакторинг, навігація
Навчальна крива	Полога: легкий старт	Складніша: потребує знання типізації та компіляції
Сумісність із браузерами	Повна: виконується напямую	Повна (після компіляції у JavaScript)
Підтримка сучасних стандартів	Часткова: залежить від середовища виконання	Повна: підтримує останні можливості через трансляцію
Поширеність у сучасних фреймворках	Поширений, але менш бажаний у великих проєктах	Стандарт де-факто у React, Angular, Vue

4

Рисунок В.4 – Слайд 4

ВИБІР МОВИ ПРОГРАМУВАННЯ ДЛЯ СЕРВЕРНОЇ ЧАСТИНИ

Мова	Продуктивність	ML/AI Екосистема	Швидкість розробки	Асинхронність / Паралелізм	Простота підтримки	Готовність до продакшену
Python	Середня	Відмінна	Висока	Середня (GIL, multiprocessing)	Висока	Висока
Java	Висока	Середня	Середня	Висока	Середня	Висока
C#	Висока	Обмежена	Середня	Висока	Висока	Висока
Go	Висока	Слабка	Висока	Висока	Висока	Висока
JS/TS	Середня	Слабка	Висока	Висока (але без CPU-bound)	Висока	Висока
C/C++	Дуже висока	Обмежена	Низька	Висока	Низька	Висока (модульно)
Rust	Дуже висока	Обмежена	Низька	Висока (безпечна)	Середня	Зростаюча

5

Рисунок В.5 – Слайд 5

ВИБІР ФРЕЙМВОРКА ДЛЯ КЛІЄНТСЬКОЇ ЧАСТИНИ

Критерій	React	Angular	Vue.js	Svelte	SolidJS
Типізація	Підтримка через TypeScript	Вбудована TypeScript	Опційна (через TS)	Опційна	Опційна
Маршрутизація	TanStack Router	Angular Router	Vue Router	SvelteKit Routing	Низькорівнева
Навчальна крива	Помірна	Висока	Низька	Низька	Висока
Продуктивність	Висока	Середня	Висока	Дуже висока	Найвища
Екосистема	Найбільша	Велика	Помірна	Молода	Дуже молода
Масштабованість	Висока	Висока	Середня	Обмежена	Обмежена
Офіційна підтримка рішень	Частково (через окремі пакети)	Повна «з коробки»	Часткова	Часткова	Низька
Гнучкість	Висока	Низька	Висока	Висока	Висока

6

Рисунок В.6 – Слайд 6

ВИБІР ФРЕЙМВОРКА ДЛЯ СЕРВЕРНОЇ ЧАСТИНИ

Критерій	Django	Flask	FastAPI	Tornado
Архітектура	Монолітна	Мікрофреймворк	Мікрофреймворк	Мікрофреймворк
Підтримка асинхронності	Обмежена	Можлива через розширення	Повна, вбудована	Повна, вбудована
Продуктивність	Середня	Висока	Дуже висока	Дуже висока
Документація API	Через сторонні пакети	Через сторонні пакети	Автоматична OpenAPI	Немає автоматичної
Простота створення API	Середня	Висока	Дуже висока	Складна
Навчальна крива	Висока	Низька	Помірна	Висока
Масштабованість	Висока	Середня	Висока	Висока
Екосистема	Дуже велика	Велика	Велика та швидко зростаюча	Невелика

7

Рисунок В.7 – Слайд 7

ВИБІР СЕРЕДОВИЩА РОЗРОБКИ

Критерій	Visual Studio Code	WebStorm	PyCharm	Sublime Text	Vim
Підтримка Python	Відмінна	Обмежена	Відмінна	Задовільна	Висока (з налаштуванням)
Підтримка JavaScript/TypeScript	Відмінна	Відмінна	Обмежена	Задовільна	Висока (з налаштуванням)
Підтримка FastAPI	Є розширення	Потрібна ручне налаштування	Відмінна	Через плагіни	Через плагіни
Продуктивність	Висока	Висока (вимоглива до ресурсів)	Висока (вимоглива до ресурсів)	Дуже висока	Дуже висока
Інтеграція з Git	Вбудована	Вбудована	Вбудована	Через плагіни	Через плагіни
Наявність безкоштовної версії	Так	Ні	Так (Community Edition)	Так	Так
Зручність налаштування	Висока	Дуже висока	Висока	Середня	Низька

8

Рисунок В.8 – Слайд 8

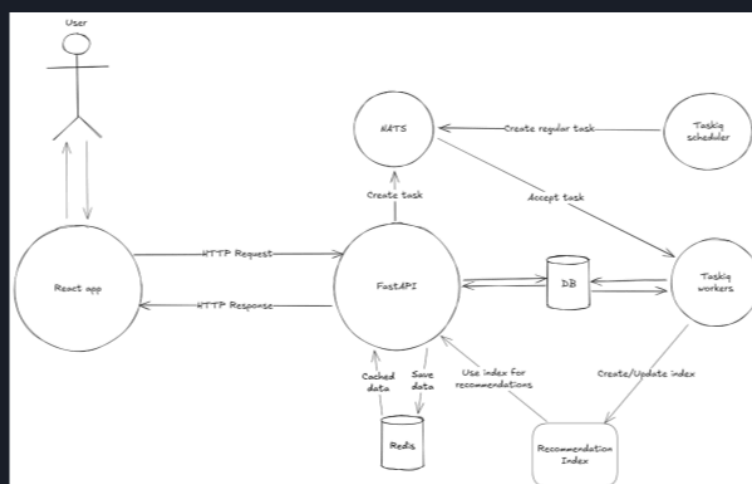
ВИБІР ТИПУ СИСТЕМИ РЕКОМЕНДАЦІЙ

Критерій	Колaborативна фільтрація	Content-based фільтрація	Гібридні системи	Системи на основі знань
Вимоги до даних	Велика кількість даних про користувачів і вподобання	Тільки характеристики контенту	Обидва типи даних (користувачів і вподобання та характеристики контенту)	Експертні знання та правила
Проблема "холодного старту"	Значна	Відсутня для нового контенту	Частково вирішена	Мінімальна
Обчислювальна складність	Середня до високої	Низька до середньої	Висока	Низька до середньої
Здатність пояснювати рекомендації	Низька	Висока	Середня	Дуже висока
Персоналізація	Висока	Середня	Дуже висока	Середня
Різноманітність рекомендацій	Може бути низькою	Середня	Висока	Залежить від правил
Придатність для музичної галузі	Добра за наявності даних	Відмінна завдяки аудіо характеристикам	Дуже добра, але складна	Добра, але потребує експертизи
Складність реалізації	Середня	Низька	Висока	Середня до високої
Масштабованість	Проблематична	Добра	Складна	Добра
Адаптивність до зміни вподобань	Висока	Низька до середньої	Висока	Низька

9

Рисунок В.9 – Слайд 9

СХЕМА АРХІТЕКТУРИ СИСТЕМИ



10

Рисунок В.10 – Слайд 10

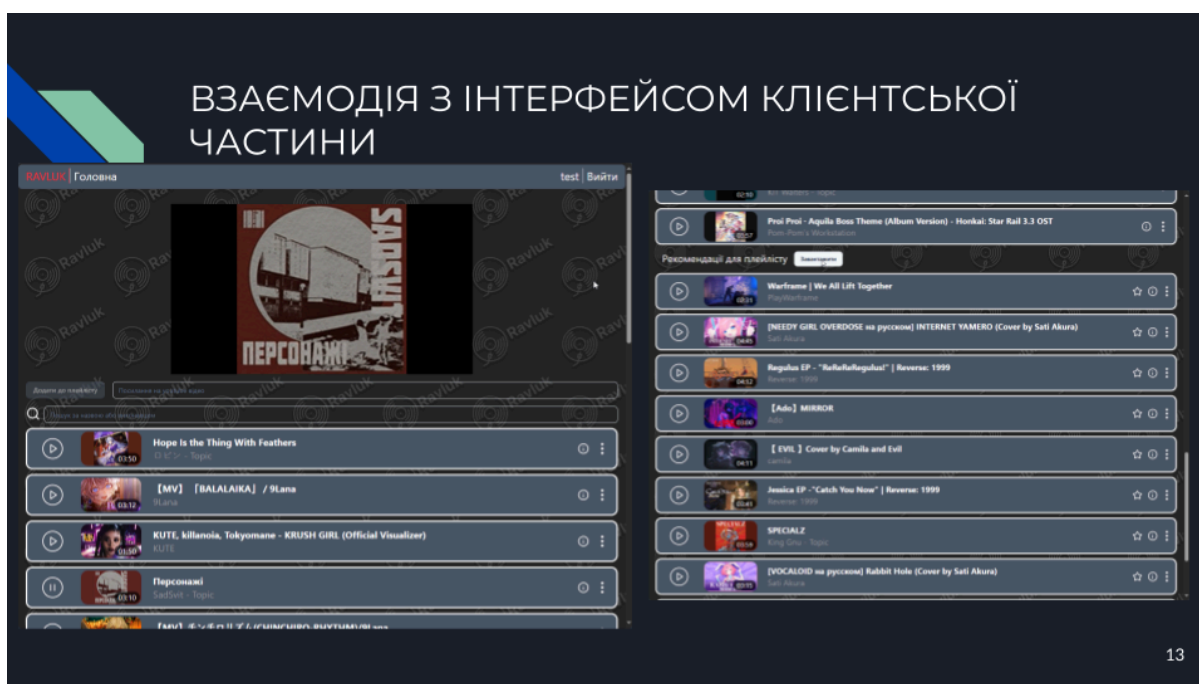


Рисунок В.13 – Слайд 13

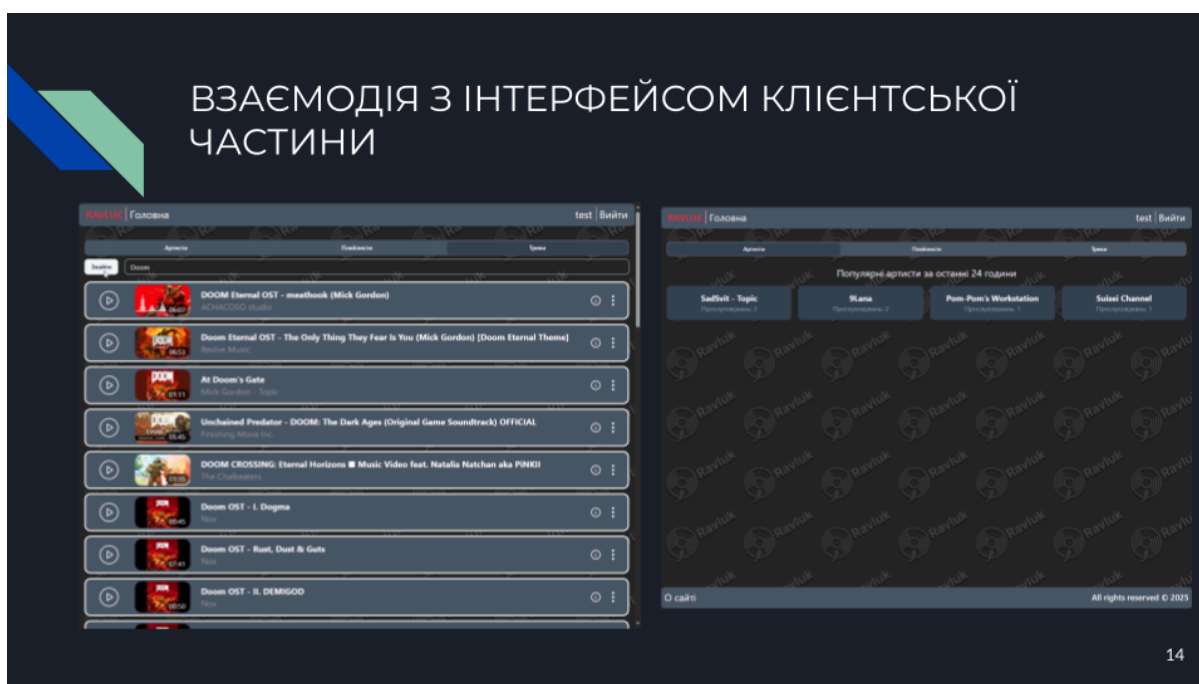


Рисунок В.14 – Слайд 14

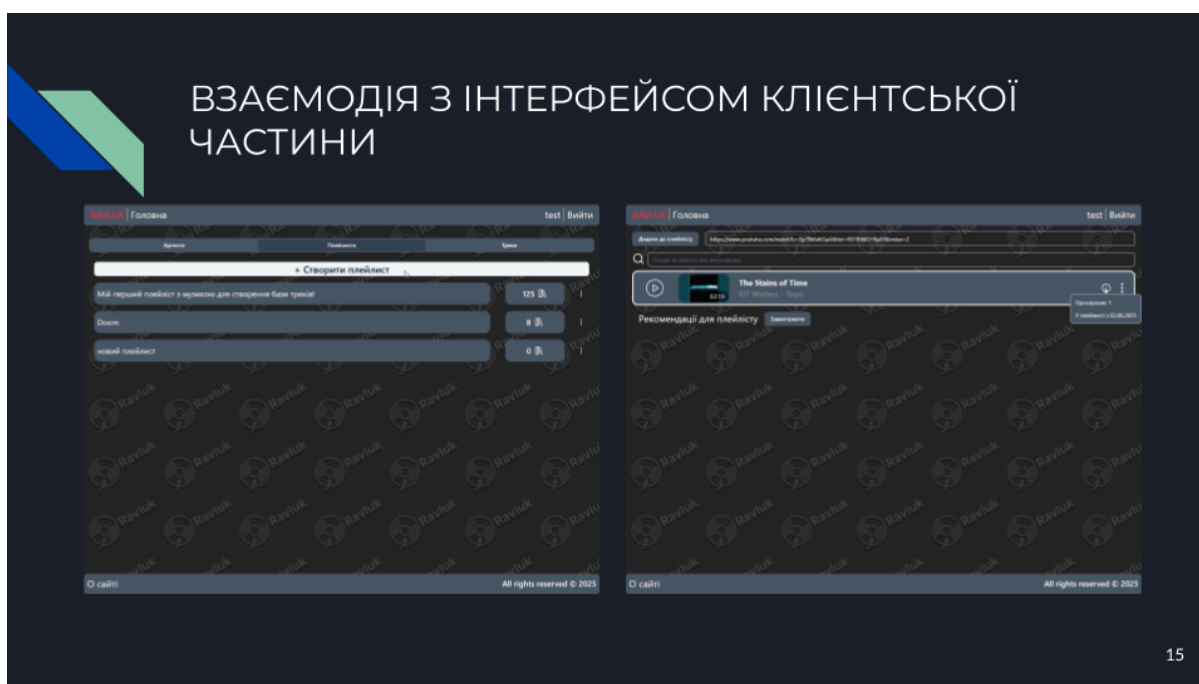


Рисунок В.15 – Слайд 15

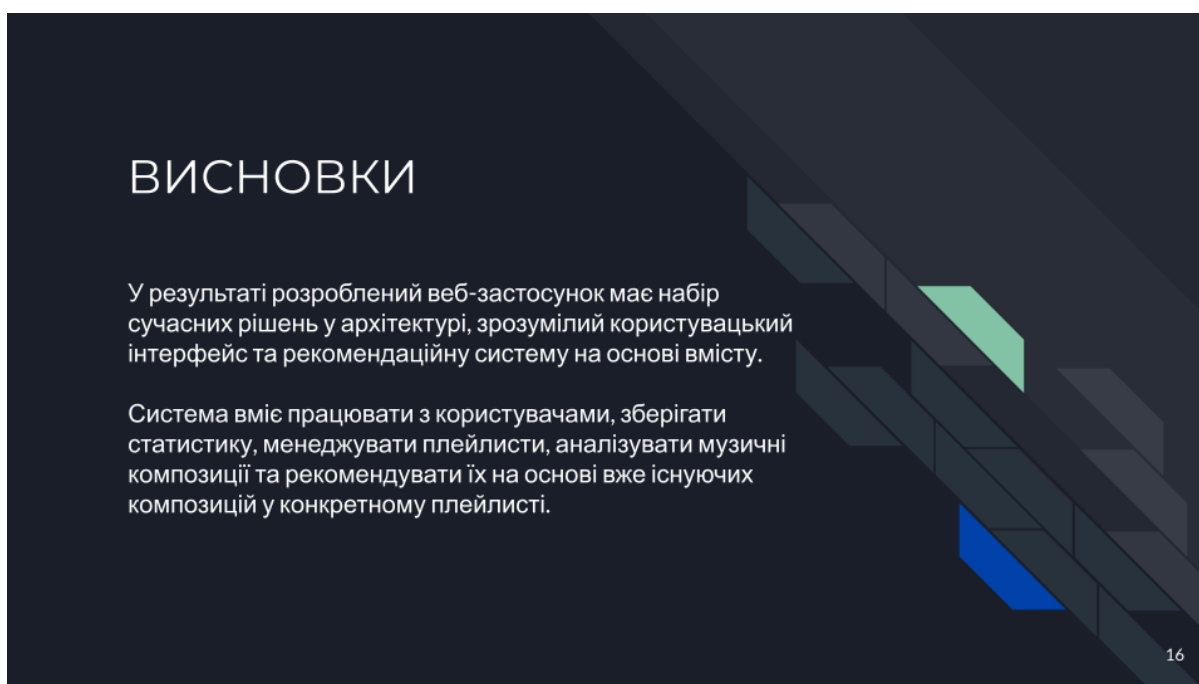


Рисунок В.16 – Слайд 16

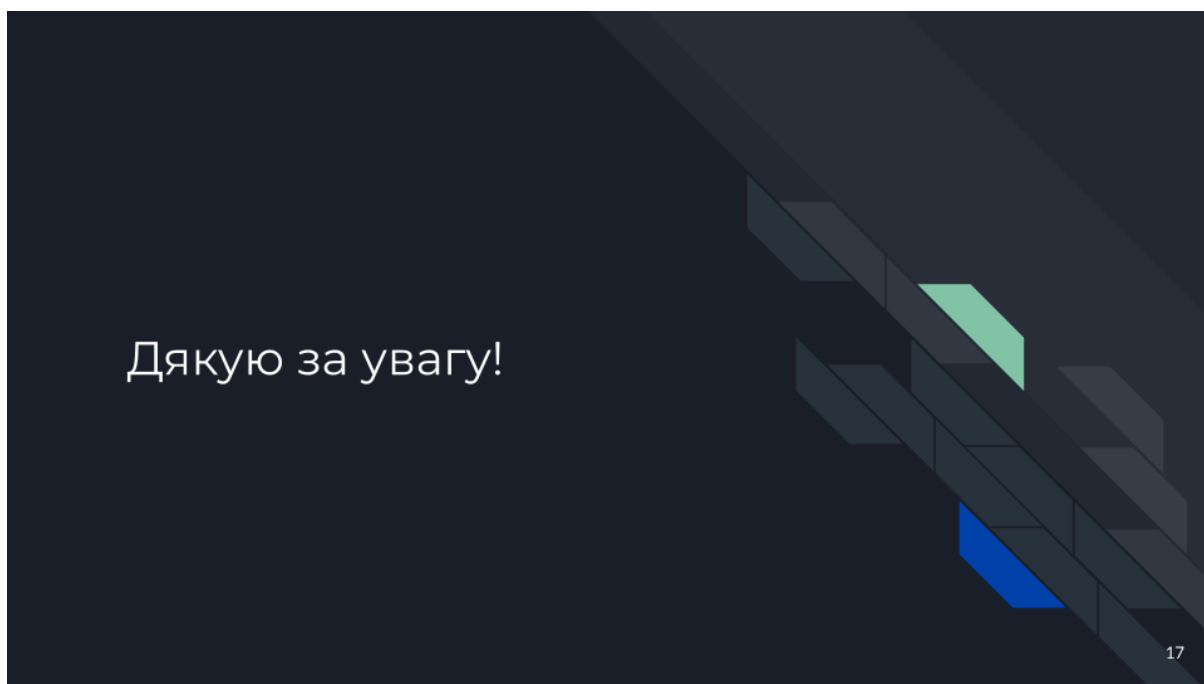


Рисунок В.17 – Слайд 17