

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Мобільний додаток для прослуховування пісень»

Виконав студент групи ПЗ-21-2
ЗАГЛАДЬКО Богдан Геннадійович

Керівник роботи:
КОНЦИДАЙЛО Андрій Михайлович

Рецензент:
ФАНТ Микола Олександрович

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна Вакалюк

«26» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ЗАГЛАДЬКО Богдан Геннадійович**

Керівник роботи: **КОНЦИДАЙЛО Андрій Михайлович**

Тема роботи: **«Мобільний додаток для прослуховування пісень»**,
затверджена Наказом закладу вищої освіти від **«23» лютого 2025 р., №74/с**

Вихідні дані для роботи: Мобільні застосунки для відтворення та прослуховування музичних композицій. Процес розробки функціонального інтерфейсу та реалізації основних можливостей мобільного додатку для прослуховування пісень

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Концидайло А.М	27.02.2025р.	27.02.2025р.
2	Концидайло А.М	27.02.2025р.	27.02.2025р.
3	Концидайло А.М	27.02.2025р.	27.02.2025р.

РЕФЕРАТ

Пояснювальна записка: 122 с., 3 розділи, 18 рисунків, 6 таблиць, 1 додаток, 30 джерел.

У дипломній роботі проведено аналіз предметної області, пов'язаної з мобільними застосунками для онлайн-прослуховування музики. Розглянуто популярні аналоги та технології, що використовуються для реалізації подібних сервісів. На основі проведеного дослідження спроектовано та реалізовано мобільний застосунок з відповідним функціоналом. Реалізацію виконано з використанням технологій Node.js, React Native та SQLite.

Список ключових слів: МОБІЛЬНИЙ ЗАСТОСУНОК, МУЗИКА, ПРОСЛУХОВУВАННЯ, JAVASCRIPT, NODE.JS, REACT NATIVE.

					ІПЗ.КР.Б – 121 – 25 – ІПЗ								
Змн.	Арк.	№ докум.	Підпис	Дата	Випускна кваліфікаційна робота				Літ.	Арк.	Аркуші		
Розроб.	Загладько Б.Г										3	122	
Керівник	Концидайло А.М.												
Реценз.													
Н. Контр.													
Затверд.													

ABSTRACT

Explanatory note: 122 pp., 3 sections, 18 figures, 6 tables, 1 appendices, 30 sources.

The diploma thesis presents an analysis of the domain related to mobile applications for online music listening. Popular analogues and technologies used to implement similar services were reviewed. Based on the conducted analysis, a mobile application was designed and implemented with the functionality of user registration and authorization, music playback, track information viewing, playlist management, listening history tracking, and lyrics interaction. The implementation was carried out using Node.js, React Native, and SQLite technologies.

List of keywords: MOBILE APPLICATION, MUSIC, STREAMING, JAVASCRIPT, NODE.JS, REACT NATIVE.UKRAINIAN CARPATHIANS, JAVASCRIPT, REACT NATIVE

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

ВСТУП	6
РОЗДІЛ 1. ОГЛЯД АНАЛОГІЧНИХ РІШЕНЬ. АНАЛІЗ ВИМОГ ДО СИСТЕМИ	8
1.1. Аналіз існуючих музичних додатків	8
1.2. Порівняння функціоналу, технологій, дизайну	10
1.3. Визначення функціональних вимог до системи	12
1.4. Визначення нефункціональних вимог до системи	14
1.5. Визначення цільової аудиторії.....	16
Висновки до розділу 1	17
РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ.....	20
2.1. Вибір архітектури додатка	20
2.2. Вибір платформи	23
2.3. Проектування структури бази даних.....	26
2.4. Визначення моделі взаємодії між компонентами	32
Висновки до розділу 2	34
РОЗДІЛ 3. РЕАЛІЗАЦІЯ МОБІЛЬНОГО ДОДАТКУ ДЛЯ ПРОСЛУХОВУВАННЯ ПІСНЕЙ. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ДОДАТКУ	37
3.1. Порівняльний аналіз мов програмування.....	37
3.2. Аналіз існуючих СУБД для вирішення поставленої задачі.....	42
3.3. Програмна реалізація проектного застосунку	44
3.4. Тестування розробленого додатку	84
Висновки до розділу 3	92
ВИСНОВОК.....	94
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	95
ДОДАТКИ.....	97

ВСТУП

У сучасному світі музика відіграє важливу роль у повсякденному житті людини — вона супроводжує нас під час роботи, відпочинку, подорожей і фізичної активності. Разом із розвитком цифрових технологій і широким розповсюдженням смартфонів зростає потреба у зручних, функціональних та швидкодіючих мобільних додатках, які надають можливість слухати улюблені композиції будь-де та будь-коли.

Популярність потокових сервісів зумовлює підвищення вимог до інтерфейсу, стабільності та гнучкості застосунків, що дозволяють не лише прослуховувати музику, але й зберігати історію прослуховувань, створювати власні плейлисти, отримувати інформацію про виконавців, переглядати тексти пісень тощо. Водночас, незважаючи на велику кількість комерційних платформ, залишається потреба в альтернативних рішеннях — простих у використанні, з можливістю гнучкої інтеграції та адаптації під конкретні потреби користувачів. Мобільні додатки, створені на основі відкритих або локальних сховищ даних, дозволяють уникати надмірної залежності від сторонніх API та забезпечують більшу контрольованість системи. Такі рішення мають особливе значення в умовах обмежених ресурсів, дослідження архітектурних підходів або створення навчальних прототипів, орієнтованих на подальший розвиток.

В умовах такого попиту особливу актуальність набуває створення сучасного мобільного додатка для прослуховування музики, орієнтованого на зручність, швидкодію, персоналізацію та мінімалістичний дизайн. Такий застосунок має задовольнити основні потреби користувачів у сфері музичного споживання та забезпечити високий рівень користувацького досвіду.

Це обумовило мету даної кваліфікаційної роботи, яка полягає у розробці мобільного застосунку для прослуховування музики з функціями управління плейлистами, збереження історії прослуховувань та відображення детальної

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		6

інформації про треки. Для досягнення поставленої мети було сформульовано наступні завдання:

1. Провести огляд предметної галузі та проаналізувати існуючі рішення у сфері музичних мобільних застосунків.
2. Виконати аналіз і обґрунтування вибору інструментів та технологій для реалізації функціоналу застосунку.
3. Розробити архітектуру застосунку та реалізувати його основні програмні модулі.
4. Провести тестування працездатності розробленого застосунку та оцінити його ефективність у порівнянні з аналогічними рішеннями.

Об'єкт дослідження – мобільний застосунок для прослуховування музики з підтримкою взаємодії користувача з треками та плейлистами.

Предмет дослідження – програмні засоби, що забезпечують створення функціонального музичного мобільного застосунку з можливістю онлайн-прослуховування, управління плейлистами, перегляду історії прослуховувань та відображення детальної інформації про треки.

Методологічною основою дослідження є загальнонаукові та спеціальні методи, які дали змогу проаналізувати існуючі рішення, сформулювати вимоги до застосунку, обґрунтувати вибір архітектури програмного забезпечення та реалізувати функціонал відповідно до поставлених завдань.

Практичне значення отриманих результатів полягає в тому, що розроблений мобільний застосунок може використовуватися широким колом користувачів для зручного прослуховування музики, формування персоналізованих плейлистів, перегляду історії прослуховувань і покращення взаємодії з музичним контентом.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		7

РОЗДІЛ 1. ОГЛЯД АНАЛОГІЧНИХ РІШЕНЬ. АНАЛІЗ ВИМОГ ДО СИСТЕМИ

1.1. Аналіз існуючих музичних додатків

На сучасному етапі розвитку цифрових технологій мобільні додатки для прослуховування музики займають одне з провідних місць серед засобів розваг. Популярність потокової музики обумовлена легкістю доступу до великої кількості контенту, відсутністю потреби в локальному зберіганні аудіофайлів та широкими можливостями персоналізації. Найбільш поширеними та успішними мобільними додатками для прослуховування музики є Spotify, YouTube Music, Apple Music, а також Deezer і TIDAL. Розглянемо їхні особливості детальніше.

Spotify є одним з найвідоміших стримінгових сервісів у світі, що надає доступ до мільйонів пісень та подкастів. Ключовими перевагами цього додатку є високий рівень персоналізації контенту, зручний інтерфейс, підтримка офлайн-режиму та система рекомендацій, яка враховує інтереси користувача. Spotify має як безкоштовну версію з рекламою, так і преміум-підписку, що відкриває додаткові функції, зокрема не обмежений доступ до треків та відсутність реклами. До недоліків можна віднести обмеження у безкоштовній версії, зокрема неможливість вибору конкретної пісні в деяких випадках, та відсутність підтримки високоякісного аудіо на рівні конкурентів.

YouTube Music — це сервіс від компанії Google, який тісно інтегрований із платформою YouTube. Він надає можливість слухати як офіційні музичні треки, так і неофіційні записи, кавери, концерти тощо. Основною перевагою YouTube Music є потужна система пошуку, яка дозволяє знайти пісню навіть за уривком тексту або описом. Крім того, сервіс дозволяє створювати персоналізовані плейлисти. Проте, у безкоштовній версії обмежений функціонал — неможливість фонові роботи додатку та офлайн-доступу. Для отримання повного функціоналу необхідна платна підписка.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		8

Apple Music — офіційний музичний сервіс компанії Apple, інтегрований з екосистемою iOS. Він пропонує високу якість звучання (у тому числі підтримку Lossless та Spatial Audio), великий вибір ексклюзивного контенту, а також синхронізацію з локальною бібліотекою iTunes. Інтерфейс додатку відповідає фірмовому стилю Apple та забезпечує комфортне користування. Проте, Apple Music не має безкоштовної версії (окрім обмеженого пробного періоду), а система рекомендацій поступається Spotify у гнучкості та точності [1].

Deezer — це менш популярний, але функціонально багатий сервіс. Він підтримує функцію Flow, яка створює персоналізовану музичну добірку, та пропонує підписку HiFi для прослуховування музики у високій якості. Сервіс дозволяє імпортувати треки з інших платформ та має зручну систему плейлистів. Однак, бібліотека музики трохи менша, ніж у Spotify або Apple Music, і кількість активних користувачів також є значно нижчою.

TIDAL позиціонує себе як сервіс для справжніх меломанів. Основною перевагою є якість звуку — підтримуються формати високої чіткості, включно з Master Quality Audio. Крім того, сервіс пропонує багато ексклюзивного контенту та особливо акцентує увагу на справедливій оплаті праці артистів. Проте TIDAL має вищу ціну підписки, ніж більшість конкурентів, а його система рекомендацій є менш гнучкою.

Отже, аналіз наведених сервісів дозволяє виділити кілька спільних рис успішних музичних додатків:

- наявність великої музичної бібліотеки;
- зручний і сучасний інтерфейс;
- персоналізація контенту на основі вподобань користувача;
- підтримка фонові роботи та офлайн-доступу;
- гнучка система підписок.

Водночас, жоден із сервісів не є ідеальним — кожен має певні обмеження або недоліки. Це створює передумови для розробки нового мобільного додатку, який міг би об'єднати кращі функціональні рішення

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		9

існуючих платформ та адаптувати їх під потреби конкретної аудиторії або ринку. Саме тому створення власного музичного додатку надає можливість не лише поглибити навички мобільної розробки [3; 5; 18], а й запропонувати альтернативу з урахуванням нових ідей, користувацького досвіду та сучасних технологій [4; 13], включаючи серверні рішення на Node.js [6-7] та роботу з базами даних SQLite [8-9]. Якість програмного забезпечення та дотримання принципів чистого коду [19] та ефективних практик розробки [21] є ключовими для створення конкурентоспроможного продукту.

1.2. Порівняння функціоналу, технологій, дизайну

Для більш глибокого розуміння принципів роботи сучасних музичних сервісів, доцільно порівняти найпопулярніші з них — Spotify, YouTube Music, Apple Music та Deezer — за основними критеріями: функціональні можливості, використані технології, а також підхід до дизайну інтерфейсу [1]. Порівняльний аналіз функціоналу розглянутих систем – аналогів наведено у таблицях 1.1, 1.2 та 1.3.

Таблиця 1.1

Порівняльний аналіз функціоналу розглянутих систем – аналогів

Критерій	Spotify	YouTube Music	Apple Music	Deezer
1.	2.	3.	4.	5.
Онлайн-стримінг	Так	Так	Так	Так
Офлайн-режим	Так (Premium)	Так (Premium)	Так	Так (Premium)
Персоналізовані рекомендації	Високий рівень	Середній рівень	Середній рівень	Високий рівень
Подкасти	Так	Ні	Так (обмежено)	Ні
Плейлисти користувача	Так	Так	Так	Так
Радіостанції/потоки	Так	Ні	Так	Так
Пошук за текстом	Обмежено	Так (дуже ефективний)	Обмежено	Обмежено
Соціальні функції	Так	Частково	Частково	Так
Доступна безкоштовна версія	Так	Так (обмежена)	Ні	Так

Таблиця 1.2

Порівняльний аналіз технологій розглянутих систем – аналогів

Аспект	Spotify	YouTube Music	Apple Music	Deezer
1.	2.	3.	4.	5.
Платформа	Android, iOS, Web, Desktop	Android, iOS, Web	iOS, Android, macOS, Web	Android, iOS, Web
Хмарна інфраструктура	AWS	Google Cloud	Apple Cloud	Google Cloud / AWS
Формати аудіо	AAC, Ogg Vorbis	AAC, Opus	AAC, ALAC (Lossless)	MP3, FLAC (HiFi)
Потокове передавання	DASH / HLS	HLS	ALAC / HLS	FLAC / MP3 streaming
Якість звуку	До 320 кбіт/с	До 256 кбіт/с	До 24-bit/192kHz	До 1411 кбіт/с (HiFi)
Алгоритми рекомендацій	ML, колаборативна фільтрація	ML + YouTube History	Apple Algorithms + Editorial	ML, Flow-система

Таблиця 1.3

Порівняння дизайну інтерфейсу

Критерій	Spotify	YouTube Music	Apple Music	Deezer
1.	2.	3.	4.	5.
Загальний стиль	Мінімалістичний, темна тема за замовчуванням	Візуально схожий на YouTube	Класичний, у стилі iOS	Простий, фокус на плейлистах
Навігація	Інтуїтивна, швидкий доступ до плейлистів	Схожа на відеоплеєр	Вкладки з великими блоками	Карткова система навігації
Кастомізація інтерфейсу	Обмежена	Мінімальна	Мінімальна	Мінімальна
UX-функції	Свайпи, "дотягування" контенту, швидкий доступ	Стандартний Android UX	Типовий для iOS	Інтуїтивний, але спрощений
Темна/світла тема	Обидві	Обидві	Обидві	Обидві

Проведене порівняння свідчить, що Spotify залишається найбільш збалансованим варіантом за функціональністю, персоналізацією та простотою використання. YouTube Music вигідно вирізняється гнучким пошуком і великою кількістю відео-контенту, однак поступається у зручності управління

аудіо. Apple Music має перевагу в якості звучання та глибоку інтеграцію з пристроями Apple, але відсутність безкоштовної версії обмежує доступність. Deezer, своєю чергою, фокусується на простоті та якості звуку, пропонуючи при цьому досить потужну систему рекомендацій [1].

Отже, при розробці власного мобільного додатку доцільно орієнтуватися на найкращі практики Spotify у частині UX/UI, адаптувати систему персоналізації під потреби локального користувача та забезпечити високу якість відтворення музики із підтримкою офлайн-режиму [2; 10]. Для реалізації цих цілей обрано сучасні кросплатформні мобільні технології, такі як React Native [3, 5, 18], що дозволяє створювати ефективні інтерфейси [15]. Також враховується досвід використання хмарних інфраструктур [29] та алгоритмів машинного навчання, типових для сучасних Full-Stack рішень [4, 13], що реалізуються за допомогою Node.js [6-7] та ефективного зберігання даних у SQLite [8-9]. Важливість використання патернів проєктування [20] та принципів рефакторингу [28] також враховується для забезпечення гнучкості та підтримуваності архітектури.

1.3. Визначення функціональних вимог до системи

Для ефективного проєктування та реалізації мобільного додатку для прослуховування пісень необхідно чітко визначити його функціональні вимоги. Це дозволяє розробникам, дизайнерам і тестувальникам працювати в одному напрямі, забезпечуючи відповідність очікуванням користувачів і цілям проєкту. Функціональні вимоги — це сукупність характеристик, які описують, які саме дії повинна виконувати система, як вона має реагувати на певні вхідні дані, та яку поведінку демонструвати в тих чи інших ситуаціях. Вони є основою для створення структури інтерфейсу користувача, внутрішньої логіки роботи додатку та взаємодії з базами даних або зовнішніми сервісами [1; 4; 6].

Визначення функціональних вимог є критично важливим етапом, оскільки саме ці вимоги формують список функцій, які повинні бути

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		12

реалізовані для забезпечення повноцінної роботи системи. Наявність чітко сформульованих вимог також полегшує процес тестування [27] та підтримки додатку після впровадження. Функціональні вимоги до проєктованого додатку наведено у таблиці 1.4.

Таблиця 1.4

Специфікація функціональних вимог до проєктованого мобільного додатку

Ном.	Назва вимоги	Пріоритет	Складність	Контакт
1.	2.	3.	4.	5.
1	Реєстрація користувача	Обов'язковий	Середня	Користувач, Система, База даних
2	Авторизація користувача	Обов'язковий	Середня	Користувач, Система, База даних
3	Завершення сесії користувача (вихід із системи)	Обов'язковий	Низька	Користувач, Система
4	Прослуховування пісень у режимі онлайн	Обов'язковий	Висока	Користувач, Система, База даних
5	Пошук пісень за назвою, виконавцем, альбомом, жанром	Обов'язковий	Середня	Користувач, Система, База даних
6	Фільтрація та сортування музики	Обов'язковий	Середня	Користувач, Система, База даних
7	Створення, редагування та видалення власних плейлистів	Обов'язковий	Висока	Користувач, Система, База даних
8	Додавання пісень до обраного / плейлиста	Обов'язковий	Середня	Користувач, Система, База даних
9	Видалення пісень із плейлиста	Обов'язковий	Середня	Користувач, Система, База даних
10	Відображення тексту пісень (якщо доступний у БД)	Бажаний	Середня	Користувач, Система, База даних
11	Перегляд інформації про обрану пісню	Бажаний	Середня	Користувач, Система, База даних

Функціональні вимоги, наведені в таблиці 1.4, є основоположними для побудови архітектури майбутнього мобільного додатку [1; 2]. Їх формалізація дозволяє однозначно визначити очікувану поведінку системи, спроектувати відповідні програмні модулі та забезпечити цілісність у реалізації всіх компонентів. Крім того, ці вимоги стануть основою для побудови логіки користувацького інтерфейсу [3; 10; 23], внутрішньої взаємодії із базою даних та подальшого тестування готового продукту [1; 27]. Визначення цих вимог також сприяє ефективному об'єктно-орієнтованому аналізу та проєктуванню [24], що є важливим для створення якісного програмного забезпечення [21].

Врахування функціональних вимог на ранньому етапі проєктування дозволяє зменшити ризики помилок у майбутньому, уникнути неузгодженостей у реалізації окремих частин системи та забезпечити очікувану якість кінцевого застосунку [1; 21]. Таким чином, чітке формулювання функціональних можливостей є важливим кроком до створення ефективного, зручного та сучасного сервісу для прослуховування музики на мобільних пристроях.

1.4. Визначення нефункціональних вимог до системи

Наступним кроком розглянемо нефункціональні вимоги до системи, які визначають загальні характеристики та якісні показники роботи мобільного додатку. На відміну від функціональних, ці вимоги не описують конкретні дії або поведінку системи, а формують рамки, у яких має відбуватись реалізація всього функціоналу. Саме вони впливають на зручність користування, продуктивність, безпеку та стабільність роботи застосунку в умовах реального навантаження. Нефункціональні вимоги є критично важливими, оскільки вони прямо впливають на користувацький досвід [22]. Навіть при повній відповідності функціоналу очікуванням користувача, недотримання нефункціональних характеристик — таких як швидкість запуску, адаптивність

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		14

до різних екранів чи безпека даних — може суттєво знизити загальну якість системи. Тому їх врахування ще на етапі проєктування є необхідною умовою створення надійного та конкурентоспроможного продукту [1; 21].

Нефункціональні вимоги до проєктованого мобільного додатку наведено у таблиці 1.5.

Таблиця 1.5

Специфікація нефункціональних вимог до мобільного додатку для прослуховування пісень

Ном.	Назва вимоги	Пріоритет	Складність	Контакт
1.	2.	3.	4.	5.
1	Кросплатформеність (Android / iOS)	Обов'язковий	Висока	Система
2	Висока продуктивність і швидкий відгук	Обов'язковий	Висока	Система
3	Надійність (стабільна робота без збоїв)	Обов'язковий	Середня	Система
4	Безпека збереження та обробки персональних даних	Обов'язковий	Висока	Система, База даних
5	Захист автентифікації та сесій	Обов'язковий	Середня	Система
6	Зручний та інтуїтивний інтерфейс (UI/UX)	Обов'язковий	Висока	Користувач, Система
7	Адаптивність інтерфейсу до різних розмірів екранів	Обов'язковий	Середня	Користувач, Система
8	Масштабованість архітектури	Бажаний	Висока	Система
9	Можливість розширення функціоналу	Бажаний	Висока	Система
10	Відповідність сучасним стандартам розробки	Бажаний	Середня	Система

Наведені в таблиці 1.5 нефункціональні вимоги є невід'ємною складовою процесу створення якісного мобільного додатку. Вони не визначають конкретну поведінку системи, проте суттєво впливають на сприйняття застосунку кінцевим користувачем. Відповідність цим вимогам забезпечує надійність, швидкодію, стабільність та безпеку додатку, а також формує позитивний користувацький досвід. Урахування нефункціональних характеристик на етапі проєктування дозволяє побудувати технічно грамотну

архітектуру, адаптовану до реальних умов експлуатації, та мінімізувати ризики, пов'язані з масштабуванням, оновленням або супроводом застосунку в майбутньому [1; 21]. Крім того, відповідність сучасним стандартам розробки програмного забезпечення сприяє створенню конкурентоспроможного продукту, здатного задовольнити вимоги як користувачів, так і розробників [1].

Таким чином, нефункціональні вимоги є основою для формування якісних атрибутів системи, які визначають її зручність, ефективність і довговічність у реальних умовах використання.

1.5. Визначення цільової аудиторії

У процесі розробки будь-якого мобільного додатку важливо чітко визначити цільову аудиторію, оскільки саме вона визначає не лише функціональні та нефункціональні вимоги, а й загальний напрямок розвитку проєкту, стилістику інтерфейсу, логіку користувацької взаємодії та методи просування продукту. Вивчення аудиторії дозволяє адаптувати рішення до реальних потреб користувачів і забезпечити успішність застосунку на ринку. Цільова аудиторія мобільного додатку для прослуховування пісень — це широке коло користувачів, які регулярно споживають аудіоконтент на мобільних пристроях. Для точнішого аналізу аудиторію доцільно розподілити на кілька сегментів за віковими, поведінковими та мотиваційними критеріями:

1. Молодь (віком 15–25 років) — найактивніша група користувачів стрімінгових музичних сервісів. Вони цінують зручність, швидкість доступу до нової музики, можливість створювати власні плейлисти, ділитися музикою в соціальних мережах, а також інтуїтивно зрозумілий інтерфейс. Ця аудиторія очікує динамічного, сучасного і візуально привабливого застосунку, здатного швидко реагувати на їхні смаки та вподобання [3; 5; 10; 23].

2. Дорослі користувачі (віком 26–45 років) — слухають музику під час роботи, занять спортом, поїздки чи відпочинку. Для них важливі стабільність

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		16

роботи додатку, якість відтворення, простота у використанні та наявність зручного пошуку. У цьому сегменті також актуальним є організація власних колекцій пісень і збереження прослуханих треків у плейлистах [1].

3. Аудіоентузіасти та музичні фанати — користувачі, які мають глибокий інтерес до музики як явища. Вони можуть шукати інформацію про виконавців, альбоми, тексти пісень, а також мати високі вимоги до повноти метаданих, систематизації бібліотеки та функціональності [1]. Часто користуються розширеними можливостями сортування, фільтрації, візуального перегляду та деталізації музичного контенту.

4. Користувачі, що не мають стабільного інтернет-з'єднання — для них критично важливо, щоб додаток працював швидко, стабільно та не залежав постійно від доступу до мережі. Хоча офлайн-функціонал у поточному проєкті не передбачається, інтерфейс має бути легким, швидким та оптимізованим навіть для повільних з'єднань або старших моделей смартфонів [3; 5; 18].

5. Люди з мінімальним цифровим досвідом — користувачі, які не є активними технічними споживачами, але все частіше використовують мобільні додатки для повсякденних потреб. Цей сегмент потребує максимально простого, інтуїтивного управління та зручної навігації, без перевантаження зайвим функціоналом [1; 15; 22].

Розуміння особливостей кожної із зазначених груп дозволяє сформувати універсальний, адаптивний підхід до розробки інтерфейсу та функціональних можливостей додатку. Саме орієнтація на потреби реального користувача забезпечує успішне впровадження і подальшу популярність застосунку серед обраної аудиторії.

Висновки до розділу 1

У рамках даного розділу було закладено фундаментальні основи для проєктування мобільного додатку для прослуховування пісень, що дозволило чітко визначити напрямок розробки та критерії успіху. Насамперед, глибокий

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		17

аналіз існуючих музичних стримінгових сервісів [1] (таких як Spotify, YouTube Music, Apple Music та Deezer) виявив їхні ключові функціональні особливості, сильні сторони та наявні недоліки. Це порівняння дало змогу не лише зрозуміти сучасні ринкові тенденції та очікування користувачів, а й сформулювати цінні уроки для створення конкурентоспроможного продукту. Було виділено найкращі практики в UI/UX дизайні [22, 23], персоналізації контенту та забезпеченні якості звуку, які стануть орієнтирами для нашого застосунку.

Далі, було проведено детальне визначення функціональних вимог до системи [1]. Цей етап є критично важливим, оскільки він окреслює конкретні дії, які має виконувати додаток, від реєстрації та авторизації користувачів до прослуховування пісень онлайн, пошуку, створення плейлистів та відображення текстів [1, 24]. Чітка специфікація цих вимог слугуватиме дорожньою картою для розробників, забезпечуючи повноту та логічність реалізації всіх запланованих можливостей.

Паралельно з функціональними вимогами були сформульовані нефункціональні вимоги, що визначають якісні атрибути системи [1]. До них належать кросплатформеність [3, 5, 18], висока продуктивність, надійність, безпека даних [25], а також зручний та адаптивний інтерфейс [22, 23] та масштабованість архітектури [2, 20]. Дотримання цих вимог є запорукою позитивного користувацького досвіду, довговічності та успішності застосунку на ринку [21]. Обґрунтовано, що реалізація цих нефункціональних аспектів буде забезпечена завдяки вибору Full-Stack архітектури на базі React Native для фронтенду та Node.js з SQLite для бекенду [4, 6, 8, 13].

Нарешті, було чітко визначено цільову аудиторію майбутнього мобільного додатку [1]. Аудиторію сегментовано за віковими групами, поведінковими характеристиками та цифровим досвідом, що дозволило краще зрозуміти її різноманітні потреби та очікування [22]. Розуміння цих нюансів є основоположним для створення інтуїтивно зрозумілого інтерфейсу та функціоналу, який буде максимально відповідати запитам кожного сегмента

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		18

користувачів. Це підкреслює орієнтацію проекту на користувача та його досвід [20].

Таким чином, за результатами даного розділу не лише проведено всебічний аналіз передумов для розробки, але й закладено міцний фундамент для наступних етапів проєктування та реалізації. Визначені функціональні та нефункціональні вимоги, а також чітке розуміння цільової аудиторії, забезпечують цілісний і стратегічно обґрунтований підхід до створення ефективного, сучасного та конкурентоспроможного мобільного додатку для прослуховування пісень.

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		19

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ

2.1. Вибір архітектури додатка

Архітектура програмного забезпечення визначає загальну структуру системи, компоненти, з яких вона складається, а також способи взаємодії між ними. У випадку мобільного додатку для прослуховування пісень, архітектура повинна забезпечити ефективну взаємодію між інтерфейсом користувача, логікою обробки даних, сховищем інформації та іншими сервісами, які забезпечують виконання функціоналу [1]. Вибір архітектури є одним з найважливіших рішень на початкових етапах розробки, оскільки він має довгостроковий вплив на всі аспекти проєкту, включаючи масштабованість, підтримуваність, продуктивність та безпеку [1, 21].

Розробка архітектури передбачає поділ системи на два основні рівні: Frontend (клієнтська частина), що відповідає за відображення інтерфейсу та взаємодію з користувачем [3; 5; 10; 12; 15; 18], та Backend (серверна частина), яка забезпечує обробку запитів, зберігання й обслуговування даних, а також управління логікою застосунку [6-7; 14]. Такий підхід дає змогу підвищити масштабованість, гнучкість і зручність супроводу додатку [1-2].

Frontend-архітектура мобільного додатку для прослуховування пісень реалізується з використанням кросплатформеного фреймворку React Native [3; 5; 12; 15; 18], що дозволяє створити єдину кодову базу для платформ Android та iOS. Такий підхід забезпечує ефективну розробку, зменшує витрати на підтримку проєкту та дозволяє швидко масштабувати інтерфейс для різних типів пристроїв [3; 18]. Ключовим принципом побудови клієнтської частини є компонентна архітектура [10], за якої кожна частина інтерфейсу реалізується у вигляді окремого логічно завершеного модуля. Компоненти відповідають за конкретні функції: список треків, екран авторизації, програвач, екран плейлистів тощо [27]. Це дозволяє повторно використовувати код, спрощує тестування, обслуговування і розширення системи [2; 10; 19; 28]. Принцип побудови компонентної архітектури зображено на рисунку 2.1.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				20
Змн.	Арк.	№ докум.	Підпис	Дата		

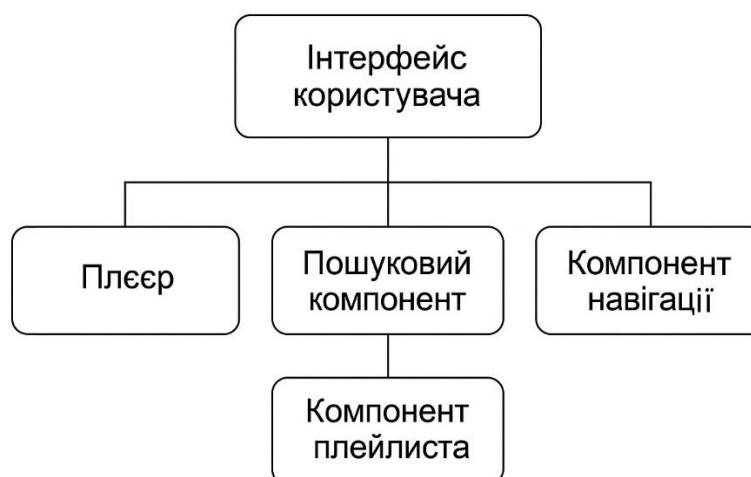


Рисунок 2.1 – Принцип побудови компонентної архітектури клієнтської частини додатку

Кожен компонент може мати власний стан або використовувати глобальне сховище, забезпечуючи реактивну зміну інтерфейсу у відповідь на дії користувача [10; 17]. Для реалізації переходів між екранами використовується бібліотека React Navigation, яка підтримує стекову, вкладену та таб-навігацію [3; 5]. Обмін даними з сервером здійснюється через REST API, з використанням бібліотеки Axios, яка забезпечує асинхронну взаємодію між клієнтом і сервером [4; 13-14]. У випадках, коли потрібне тимчасове збереження даних (сесії, налаштування, обрані пісні), застосовується AsyncStorage або SQLite, що дозволяє зменшити навантаження на серверну частину та покращити продуктивність [8-9]. Це рішення також покращує користувацький досвід (UX) [22], забезпечуючи швидший доступ до даних.

Таким чином, обрана frontend-архітектура поєднує гнучкість компонентного підходу з ефективністю кросплатформеного фреймворку, що дозволяє створити сучасний, масштабований і зручний для користувача мобільний додаток [1-2].

Серверна частина мобільного додатку для прослуховування пісень реалізована на основі REST-архітектури [4; 13-14], яка дозволяє чітко

структурувати взаємодію між клієнтом і сервером через уніфіковані HTTP-запити [1]. Такий підхід забезпечує масштабованість, модульність і простоту підтримки системи [1-2; 21].

Для реалізації серверної частини обрано Node.js — високопродуктивне середовище виконання JavaScript-коду на сервері [6-7], яке дозволяє ефективно обробляти велику кількість одночасних з'єднань завдяки неблокуючій моделі I/O. У поєднанні з фреймворком Express.js, ця технологія дозволяє створювати гнучкі та розширювані RESTful API [6; 14]. Загальний принцип побудови REST-архітектури серверної частини подано на рисунку 2.2.

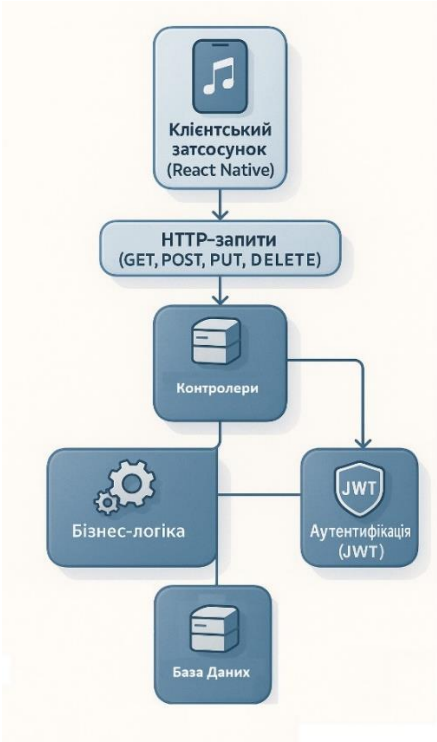


Рисунок 2.2 – Загальна структура REST-архітектури серверної частини мобільного додатку

Серверна частина відповідає за обробку всіх ключових запитів від клієнта: авторизацію користувачів, пошук і фільтрацію музичного контенту, формування та збереження плейлистів, отримання тексту пісень та інше [1]. Обмін інформацією відбувається через API-ендпоінти, кожен з яких виконує

конкретну функцію: POST /auth/register, POST /auth/login, GET /tracks, POST /playlist/add, DELETE /playlist/:trackId тощо. Усі запити обробляються у форматі JSON. Дані зберігаються у локальній реляційній базі даних SQLite, підключеній до серверної частини через бібліотеку sqlite3 [8-9]. У базі створено структуру таблиць для користувачів, пісень, виконавців, плейлистів і зв'язків між ними. Взаємодія з базою даних здійснюється через SQL-запити, що дозволяє виконувати гнучкі фільтрації, сортування та агрегацію [8-9].

З метою захисту персональних даних реалізовано механізм автентифікації на основі JWT (JSON Web Token). Після успішної авторизації користувач отримує токен доступу, який додається до кожного запиту до захищених ендпоінтів. Це забезпечує безпечний контроль доступу до даних та ізоляцію користувацьких сесій [14].

Додатково реалізовано централізовану обробку помилок, валідацію вхідних даних, обмеження доступу до окремих ресурсів, а також логування дій на сервері [1]. Така архітектура дозволяє ефективно реалізовувати логіку взаємодії з клієнтською частиною, підтримувати надійну роботу сервісу та забезпечувати зручність у подальшій розробці або масштабуванні системи [1-2]. Це також сприяє зменшенню технічного боргу та спрощує майбутній рефакторинг [28].

2.2. Вибір платформи

Сучасні вимоги до мобільних застосунків передбачають їх доступність на якомога ширшому спектрі пристроїв, незалежно від операційної системи. У зв'язку з цим усе більше значення набувають кросплатформені підходи до розробки, які дозволяють одночасно підтримувати як Android-, так і iOS-користувачів без необхідності створення двох окремих нативних застосунків [3; 5; 18]. Цей тренд є відповіддю на потребу бізнесу в оптимізації ресурсів та прискоренні виходу продукту на ринок, зберігаючи при цьому високу якість та консистентність користувацького досвіду [1].

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		23

Під час проєктування системи мобільного додатку для прослуховування пісень було враховано такі фактори, як технологічна доцільність, швидкість розгортання, потреба в уніфікованому користувацькому інтерфейсі та подальше обслуговування системи [1]. Традиційна нативна розробка, хоча й забезпечує максимальну продуктивність та доступ до системних функцій, часто вимагає окремих команд та кодових баз для кожної платформи (iOS та Android), що призводить до подвоєння зусиль та витрат. З огляду на це, доцільним є застосування кросплатформеної архітектури, яка дозволить реалізувати програмну логіку одночасно для обох основних мобільних платформ, з мінімальними відмінностями в реалізації [3; 5].

Серед найпопулярніших кросплатформених фреймворків виділяють React Native, Flutter та Xamarin. Кожен з них має свої переваги: Flutter, розроблений Google, відомий високою продуктивністю та швидким рендерингом інтерфейсу, тоді як Xamarin, що належить Microsoft, дозволяє розробляти мобільні додатки на C#. Однак, для даного проєкту було обрано React Native. Цей вибір обґрунтований кількома причинами: використання JavaScript як основної мови розробки [22], що полегшує залучення веброзробників та забезпечує доступ до великої екосистеми бібліотек; значна зрілість фреймворку та його широка спільнота, що гарантує швидку підтримку та доступ до готових рішень [18]. React Native дозволяє створювати інтерфейси, які за своєю продуктивністю та візуальним сприйняттям близькі до нативних, використовуючи при цьому принцип "вивчи один раз, пиши де завгодно" (learn once, write anywhere) [3; 5]. Це забезпечує оптимальний баланс між продуктивністю нативних додатків та ефективністю кросплатформеної розробки.

Кросплатформений підхід дозволяє досягти низки важливих переваг:

1. Забезпечити однаковий користувацький досвід (UX) для користувачів Android та iOS. Це допомагає створити єдиний бренд і зменшує плутанину, оскільки користувачі отримують ідентичний інтерфейс та функціонал незалежно від платформи [20-23].

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				24
Змн.	Арк.	№ докум.	Підпис	Дата		

2. Уникнути дублювання логіки застосунку у двох різних кодових базах, що значно спрощує розробку та мінімізує ймовірність появи помилок. Одна кодова база дозволяє легко впроваджувати зміни та оновлення [1; 19; 28].
3. Скоротити час проєктування, розробки й подальшого супроводу. Завдяки єдиній кодовій базі команда розробників може працювати над одним проєктом, що прискорює ітерації та вихід нових версій.
4. Зменшити витрати на впровадження оновлень, оскільки виправлення помилок або додавання нових функцій відбувається лише один раз і розгортається на обох платформах одночасно, без необхідності підтримувати паралельні розробки.
5. Забезпечити уніфікований інтерфейс та зручну систему тестування. Тестування однієї кодової бази є значно ефективнішим [27], а єдиний інтерфейс зменшує кількість сценаріїв для перевірки, що покращує якість кінцевого продукту та прискорює цикл тестування [23].

На етапі реалізації передбачається, що інтерфейс користувача, логіка навігації між екранами, обробка запитів до серверної частини, а також локальна обробка даних будуть реалізовані як єдиний кросплатформений застосунок із подальшим розгортанням для Android та iOS [4; 13]. Такий підхід, побудований на використанні React Native, дозволяє не лише досягти високої функціональної цілісності та гнучкості системи [1-2], але й забезпечити відповідність сучасним стандартам розробки [17; 21] та принципам чистого коду, що є запорукою подальшої підтримуваності та масштабованості рішення [16].

Таким чином, з урахуванням сучасних тенденцій мобільної розробки та специфіки проєкту, кросплатформений підхід із застосуванням React Native є оптимальним рішенням. Він забезпечує високу ефективність реалізації, масштабованість та доступність системи для максимально широкого кола користувачів, відповідаючи як функціональним, так і нефункціональним вимогам, визначеним на попередніх етапах [3; 5; 18].

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				25
Змн.	Арк.	№ докум.	Підпис	Дата		

2.3. Проектування структури бази даних

У сучасних інформаційних системах бази даних відіграють ключову роль у забезпеченні зберігання, організації, обробки та швидкого доступу до великого обсягу структурованої інформації [1; 8-9]. У контексті мобільних застосунків, особливо тих, що пов'язані з мультимедійним контентом і взаємодією користувачів, база даних є основою для підтримки цілісності системи, реалізації бізнес-логіки та персоналізації функціоналу.

Мобільний додаток для прослуховування пісень оперує різними типами даних: обліковими записами користувачів, інформацією про музичні треки, виконавців, плейлисти, сесіями відтворення тощо [1]. Для ефективного керування цими даними передбачено використання реляційної бази даних, що дозволяє встановлювати логічні зв'язки між сутностями, забезпечувати цілісність записів і виконувати запити із заданими умовами [8-9].

Завдяки використанню бази даних забезпечується збереження користувацької активності, швидкий пошук і фільтрація треків, формування персональних списків відтворення та підтримка авторизації через збереження облікових даних. У межах серверної частини додатку база даних взаємодіє з REST API [14] і є центральним джерелом даних для всіх запитів, що надходять від клієнта [1; 8; 25].

Перейдемо до проектування основних таблиць, які формують ядро структури бази даних у розроблюваній системі [8-9]. Це також включає принципи, що підвищують якість коду та підтримуваність структури [19, 28].

Першим кроком перейдемо до проектування структури таблиці, яка відповідає за зберігання облікових даних користувачів, що взаємодіють із системою. Структура таблиці користувачів наведена у таблиці 2.1.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		26

Таблиця 2.1

Структура таблиці користувачів у проєктованій системі

Об'єкт	Власність	Тип	Розмірність	Ідентифікатор
1.	2.	3.	4.	5.
Таблиця користувачів	Ідентифікатор	INT	100	id
	Логін	VARCHAR	100	login
	Пароль	VARCHAR	100	password
	Ім'я	VARCHAR	100	name
	Прізвище	VARCHAR	100	sname
	Електронна пошта	VARCHAR	100	email

У таблиці зберігається ключова інформація для автентифікації та ідентифікації користувачів. Поле id є унікальним ідентифікатором кожного запису. login використовується як унікальне ім'я для входу до системи, а password — для перевірки повноважень. Поля name та sname зберігають особисті дані користувача, а email необхідне для зв'язку та відновлення доступу в разі втрати пароля [8-9].

Наступним кроком перейдемо до проєктування структури таблиці, яка зберігає дані про всі музичні композиції, доступні для прослуховування у мобільному додатку. Таблиця треків є однією з ключових у системі, оскільки відображає основний контент, з яким взаємодіє користувач [1; 8]. Структура таблиці треків наведена у таблиці 2.2.

Таблиця 2.2

Структура таблиці треків у проєктованій системі

Об'єкт	Власність	Тип	Розмірність	Ідентифікатор
1.	2.	3.	4.	5.
Таблиця треків	Ідентифікатор	INT	100	id
	Назва треку	VARCHAR	200	title
	Ідентифікатор виконавця	INT	100	artist_id
	Жанр	VARCHAR	100	genre
	Тривалість	INT	100	duration
	Шлях до файлу	VARCHAR	300	file_path
	Текст пісні	TEXT	5000	lyrics
	Посилання на зображення	VARCHAR	300	cover_image

У таблиці tracks зберігається інформація про кожен окремий трек, що є частиною музичного каталогу системи. Поле id виконує функцію унікального ідентифікатора. Назва композиції зберігається в полі title, а зв'язок з виконавцем реалізовано через поле artist_id, яке відповідає ключу з таблиці виконавців (artists). Поле genre відображає музичний жанр, до якого належить композиція. Тривалість треку в секундах зберігається у полі duration, що дозволяє впорядковувати й виводити інформацію про трек у плеєрі. Поле file_path містить шлях до локального або зовнішнього розташування аудіофайлу, необхідного для відтворення. За наявності, у полі lyrics зберігається текст пісні, що може бути виведений користувачу під час прослуховування. Додатково реалізовано поле cover_image, яке зберігає посилання на зображення (наприклад, обкладинку або постер пісні). Це дозволяє відображати в інтерфейсі візуальний контент під час відтворення або перегляду списків композицій [8-9].

Наступним кроком перейдемо до проєктування структури таблиці, яка зберігає інформацію про музичних виконавців. Вона необхідна для збереження базових даних про авторів пісень, а також для реалізації фільтрації або виведення додаткової інформації про трек [8-9]. Структура таблиці виконавців наведена у таблиці 2.3.

Таблиця 2.3

Структура таблиці виконавців у проєктованій системі

Об'єкт	Власність	Тип	Розмірність	Ідентифікатор
1.	2.	3.	4.	5.
Таблиця виконавців	Ідентифікатор	INT	100	id
	Ім'я виконавця	VARCHAR	150	name
	Країна	VARCHAR	100	country
	Короткий опис	TEXT	—	bio
	Фото виконавця	VARCHAR	300	photo

Таблиця artists призначена для збереження ключової інформації про виконавців, до яких належать музичні треки. Унікальний ідентифікатор id дозволяє зв'язувати виконавця з конкретними треками з таблиці tracks. Поле

name зберігає ім'я артиста або назву гурту. У полі country зазначається країна походження виконавця, що може бути використано для фільтрації контенту або його аналітики. Поле bio містить короткий текстовий опис, який відображається при перегляді профілю виконавця. Посилання на фото або обкладинку артиста зберігається в полі photo, що дозволяє формувати візуальні елементи в інтерфейсі користувача — наприклад, відображення зображення під час вибору треку чи перегляду інформації про виконавця [8-9].

Наступним кроком перейдемо до проєктування структури таблиці, яка відповідає за збереження інформації про користувацькі плейлисти. Ця таблиця дозволяє створювати та зберігати індивідуальні списки відтворення, що формуються самими користувачами в межах системи. Структура таблиці плейлистів наведена у таблиці 2.4.

Таблиця 2.4

Структура таблиці плейлистів у проєктованій системі

Об'єкт	Власність	Тип	Розмірність	Ідентифікатор
1.	2.	3.	4.	5.
Таблиця плейлистів	Ідентифікатор	INT	100	id
	Назва плейлисту	VARCHAR	150	title
	Ідентифікатор користувача	INT	100	user_id
	Дата створення	DATE	—	created_at

Таблиця playlists зберігає дані про плейлисти, які користувач створює для організації улюблених або тематичних добірок треків. Унікальний ідентифікатор id забезпечує ідентифікацію кожного плейлисту в системі. Поле title використовується для збереження назви плейлисту, яку задає користувач. user_id вказує на власника плейлисту, встановлюючи зв'язок із таблицею users. Поле created_at фіксує дату створення плейлисту, що дозволяє впорядковувати їх за часом і реалізовувати додаткову аналітику [8-9].

Наступним кроком перейдемо до проєктування структури таблиці, яка зберігає інформацію про улюблені композиції користувачів. Дана таблиця

реалізує зв'язок між користувачем та обраними ним треками, що дозволяє швидко формувати персоналізовану добірку для подальшого прослуховування [8-9]. Структура таблиці улюблених треків наведена у таблиці 2.5.

Таблиця 2.5

Структура таблиці улюблених треків у проєктованій системі

Об'єкт	Власність	Тип	Розмірність	Ідентифікатор
1.	2.	3.	4.	5.
Таблиця улюблених треків	Ідентифікатор запису	INT	100	id
	Ідентифікатор користувача	INT	100	user_id
	Ідентифікатор треку	INT	100	track_id

У таблиці favorites фіксується, які саме треки були додані користувачем до списку «Улюблене». Кожен запис представляє одну взаємодію між конкретним користувачем (user_id) та треком (track_id).

Поле id є унікальним ідентифікатором запису, необхідним для зручної модифікації або видалення зв'язку. Ця структура дозволяє ефективно реалізувати функцію позначення пісень як «улюблених» і забезпечити їх швидкий доступ у застосунку без дублювання записів у таблиці tracks.

Останнім кроком перейдемо до проєктування структури таблиці, яка зберігатиме історію взаємодії користувачів із музичним контентом. Така таблиця дозволяє фіксувати, які саме композиції були відтворені, коли саме це відбулося, та ким із користувачів [8-9]. Інформація, що зберігається в цій таблиці, може використовуватись як для формування персоналізованих добірок, так і для внутрішнього аналітичного аналізу активності користувачів [1]. Це забезпечує більш гнучку та адаптивну взаємодію між системою й її аудиторією.

Структура таблиці історії прослуховування наведена у таблиці 2.6.

Таблиця 2.6

Структура таблиці історії прослуховування у проєктованій системі

Об'єкт	Власність	Тип	Розмірність	Ідентифікатор
1.	2.	3.	4.	5.
Таблиця історії	Ідентифікатор запису	INT	100	id
	Ідентифікатор користувача	INT	100	user_id
	Ідентифікатор треку	INT	100	track_id
	Час прослуховування	DATETIME	—	played_at

У таблиці history фіксується факт відтворення пісні конкретним користувачем. Поле id є унікальним ідентифікатором запису. user_id і track_id використовуються для встановлення зв'язку з відповідними таблицями користувачів та треків. Поле played_at містить дату та точний час, коли користувач розпочав відтворення композиції. Така інформація може бути використана для формування рекомендацій, аналітики активності або виводу останніх прослуханих пісень [1; 16].

Після визначення складу основних таблиць бази даних та опису їх структури необхідно сформувати візуальне представлення логічних зв'язків між об'єктами, що зберігаються у системі. Для цього доцільно використати ER-діаграму (Entity-Relationship), яка наочно демонструє сутності, їх атрибути, а також типи взаємозв'язків між таблицями бази даних [1]. ER-діаграма слугує основою для реалізації фізичної моделі бази даних і є зручним інструментом для аналізу цілісності, нормалізації та коректності логічної структури системи [1; 8]. Вона дозволяє забезпечити узгодженість зв'язків, запобігти дублюванню даних та чітко визначити напрямки взаємодії між елементами бази [1-2].

На рисунку 2.3 представлено узагальнену ER-діаграму проєктованої бази даних.

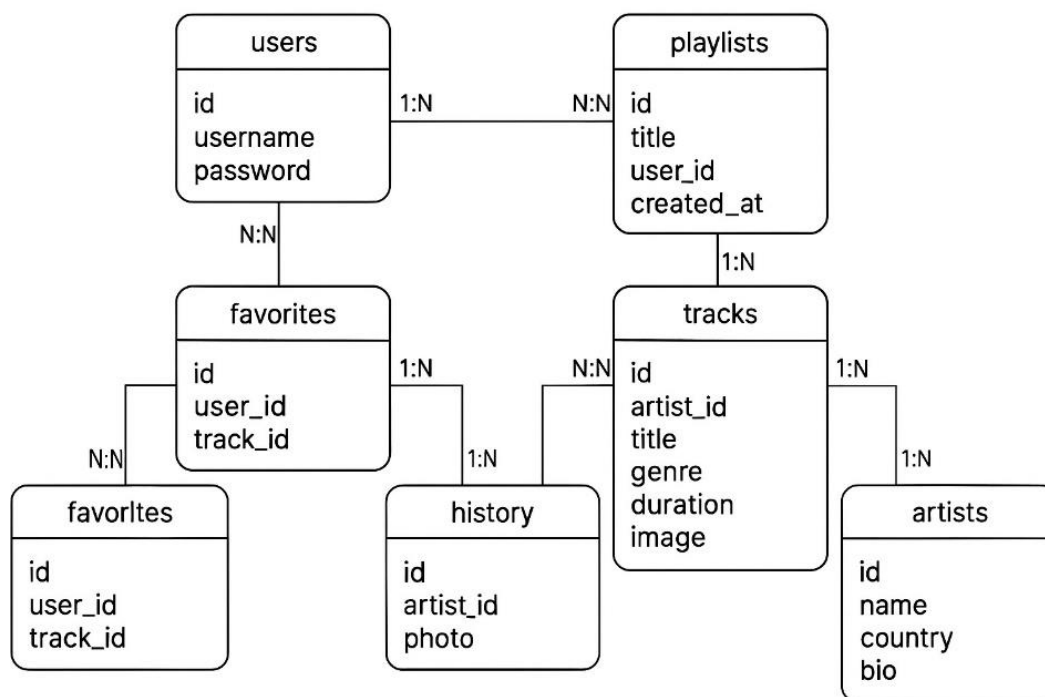


Рисунок 2.3 – ER-діаграма логічної структури бази даних мобільного додатку для прослуховування пісень

Візуалізація логічної моделі у вигляді ER-діаграми дозволяє переконатися в коректності побудованої структури, відповідності визначених зв'язків логіці функціонування системи та ефективності взаємодії між таблицями [1; 8]. На її основі буде реалізовано фізичну структуру бази даних за допомогою SQLite у серверній частині застосунку. Описана модель охоплює всі сутності, з якими працює користувач під час взаємодії із системою, і є достатньо гнучкою для подальшого розширення або адаптації під нові функціональні вимоги [1-2]. Таким чином, завершено логічне проєктування бази даних, яка є фундаментом усієї інформаційної архітектури системи.

2.4. Визначення моделі взаємодії між компонентами

У межах архітектури мобільного додатку для прослуховування пісень важливе значення має визначення моделі взаємодії між основними компонентами системи — клієнтською частиною, сервером та базою даних.

Така модель дозволяє формалізувати процес обміну даними, визначити ролі кожного з елементів та забезпечити цілісність функціонування програмного забезпечення [1].

На основі раніше спроектованих компонентів архітектури додатку, визначених таблиць бази даних та обраної платформи, можна сформуувати фінальну модель взаємодії між усіма функціональними частинами системи. Її візуалізація дозволяє цілісно оцінити механізм обміну даними між клієнтом, сервером та сховищем, а також переконатися у логічній узгодженості проєктних рішень [1]. Клієнтська частина (мобільний додаток), що працює на пристрої користувача [3; 5; 18], формує запити до серверної частини через мережу за допомогою HTTP-протоколу. Сервер застосунку, реалізований на базі Node.js з використанням фреймворку Express.js, обробляє запити, взаємодіє з базою даних та формує відповідь [6-7; 14]. Для зберігання та обробки інформації використовується локальна реляційна база даних SQLite, яка виконує роль джерела правдивих даних (source of truth) у системі [8-9].

На рисунку 2.4 зображено логічну модель взаємодії між компонентами проєктованого застосунку.

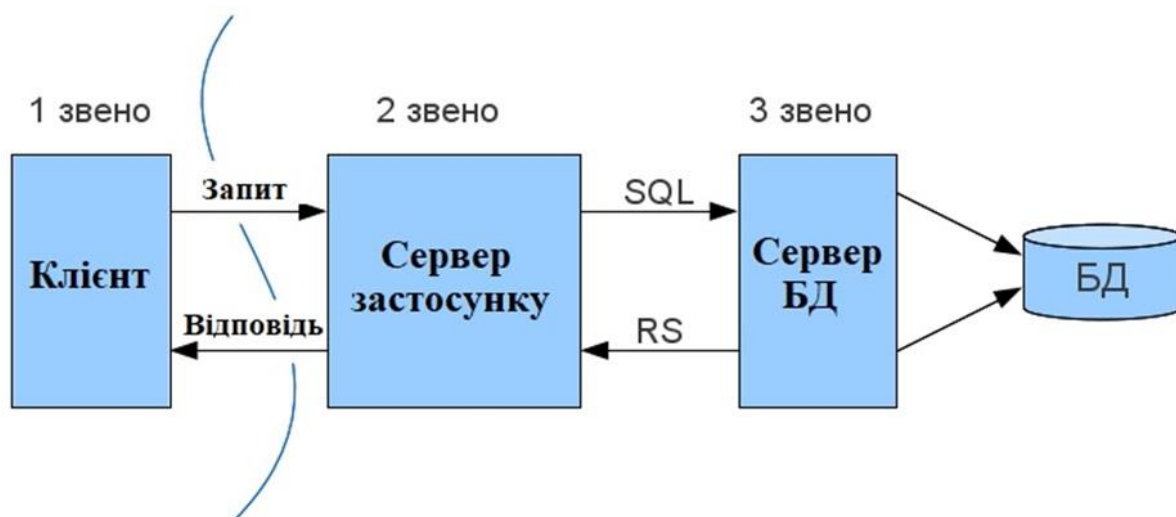


Рисунок 2.4 – Модель взаємодії між компонентами проєктованого застосунку

У представленій моделі можна чітко виділити три основні логічні звена. Перше — клієнт, тобто мобільний застосунок, який ініціює запити до

серверної частини системи, зокрема: пошук пісень, авторизацію, створення плейлистів чи додавання треків до улюбленого [3; 5; 10; 12; 15]. Друге — сервер застосунку, що приймає запити клієнта, виконує перевірку авторизації, обробляє бізнес-логіку та формує SQL-запити до сховища даних [6-7; 14]. Третє звено — сервер бази даних, який виконує запит, обробляє його та повертає результат (RS — Result Set) до сервера застосунку. Далі сервер формує структуровану відповідь у форматі JSON та повертає її клієнтському застосунку [14].

Ця модель відповідає принципам класичної тришарової архітектури [1], яка передбачає чітке розділення відповідальності між рівнем представлення (інтерфейс користувача), рівнем логіки (сервер) та рівнем даних (база даних) [2]. Такий підхід дозволяє уникнути надмірної складності на кожному окремому рівні, підвищує стабільність і дозволяє гнучко масштабувати або змінювати окремі компоненти без впливу на інші частини системи. Крім того, модель взаємодії забезпечує високий рівень захищеності даних, оскільки критична логіка та зберігання інформації відбуваються виключно на серверному боці, а клієнтська частина взаємодіє з API лише за допомогою дозволених ендпоінтів. Це мінімізує ризик несанкціонованого доступу до персональних даних та зловживань [14; 30].

Таким чином, завершено етап проєктування інформаційної системи: визначено її архітектуру, обґрунтовано вибір платформи, спроектовано структуру бази даних і сформовано логічну модель взаємодії між компонентами. Усі ці елементи разом утворюють цілісну технічну основу для подальшої реалізації мобільного додатку [1].

Висновки до розділу 2

У цьому розділі було детально розроблено та обґрунтовано ключові аспекти проєктування мобільного додатку для прослуховування пісень, заклавши міцний технічний фундамент для його подальшої реалізації.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		34

Насамперед, було визначено та описано архітектуру системи. Обрана Full-Stack архітектура з чітким розділенням на клієнтську (Frontend) та серверну (Backend) частини [1, 4, 13] дозволяє забезпечити високу масштабованість, модульність та легкість у супроводі. Клієнтська частина будується на кросплатформеному фреймворку React Native, що гарантує єдину кодову базу для Android та iOS, прискорюючи розробку та зменшуючи витрати [3, 5, 18]. Серверна частина реалізується на Node.js з фреймворком Express.js, забезпечуючи високу продуктивність та ефективну обробку запитів завдяки асинхронній моделі [6, 7].

Далі було детально обґрунтовано вибір платформи. З огляду на сучасні вимоги ринку та потребу в охопленні широкої аудиторії, кросплатформений підхід визнано оптимальним рішенням [3, 5, 18]. Вибір саме React Native зумовлений його перевагами, такими як використання JavaScript [30], зрілість екосистеми, велика спільнота та здатність створювати інтерфейси, максимально наближені до нативних [15]. Цей підхід забезпечує однаковий користувацький досвід (UX) на обох платформах [22, 23], уникає дублювання коду [19, 28] та значно скорочує час і витрати на розробку й подальше обслуговування [1, 21].

Третім важливим аспектом стало проектування структури бази даних. Для зберігання інформації про користувачів, пісні, плейлисти та іншу метадані було обрано реляційну базу даних SQLite [8, 9]. Її компактність, простота інтеграції та ефективність роблять її ідеальним рішенням для мобільних застосунків. Були розроблені ключові таблиці, зокрема для користувачів та треків, з урахуванням забезпечення цілісності та безпеки даних [25].

Нарешті, було визначено модель взаємодії між компонентами системи. Вона базується на тришаровій архітектурі [1, 2] з чітким розподілом ролей між клієнтом, сервером та базою даних. Взаємодія здійснюється через RESTful API [14], що забезпечує структурований та безпечний обмін даними у форматі JSON. Реалізовано механізми JWT-автентифікації для захисту користувацьких

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		35

сесій [14, 25] та централізовану обробку помилок. Ця модель гарантує стабільність, гнучкість та безпеку системи.

Таким чином, у даному розділі було успішно спроектовано всі ключові технічні аспекти майбутнього мобільного додатку. Обґрунтований вибір архітектури, платформи, бази даних та моделі взаємодії створює надійну та ефективну основу для переходу до етапу безпосередньої реалізації.

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		36

РОЗДІЛ 3. РЕАЛІЗАЦІЯ МОБІЛЬНОГО ДОДАТКУ ДЛЯ ПРОСЛУХОВУВАННЯ ПІСНЕЙ. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ДОДАТКУ

3.1. Порівняльний аналіз мов програмування

Першим кроком практичної реалізації є визначення програмних засобів, за допомогою яких буде розроблено мобільний застосунок. Правильний вибір технологій напряду впливає на функціональність, стабільність, продуктивність і зручність використання системи [1]. Окрім функціональних вимог, важливо також врахувати низку технічних: підтримку кросплатформенності (Android та iOS), збереження користувацьких даних на пристрої, інтеграцію з REST API, наявність бібліотек для роботи з аудіофайлами та зручну побудову навігації між екранами [3-6]. Застосунок має бути легким у супроводі, швидким у розгортанні та сумісним із сучасними пристроями [1-2].

На основі зазначених функціональних і технічних вимог доцільно виконати порівняльний аналіз мов програмування та супровідних інструментів, які найчастіше використовуються у сфері розробки мультимедійних мобільних застосунків [1]. Такий підхід дозволяє не лише обрати найбільш популярні та підтримувані рішення, а й оцінити їх відповідність вимогам конкретного проєкту — зокрема, кросплатформенності, інтеграції з аудіоплеєрами, роботи з локальними базами даних та оптимізації ресурсів мобільного пристрою [3; 5; 8].

Популярність мови чи фреймворку часто корелює з наявністю актуальної документації, активністю спільноти розробників, частотою оновлень і сумісністю з сучасними бібліотеками. На рисунку 3.1 подано статистику використання мов програмування, що застосовуються для створення мобільних застосунків мультимедійного призначення. Це дозволяє

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		37

наочно оцінити актуальність відповідних технологій та звузити коло потенційно придатних рішень для реалізації даної системи [10].

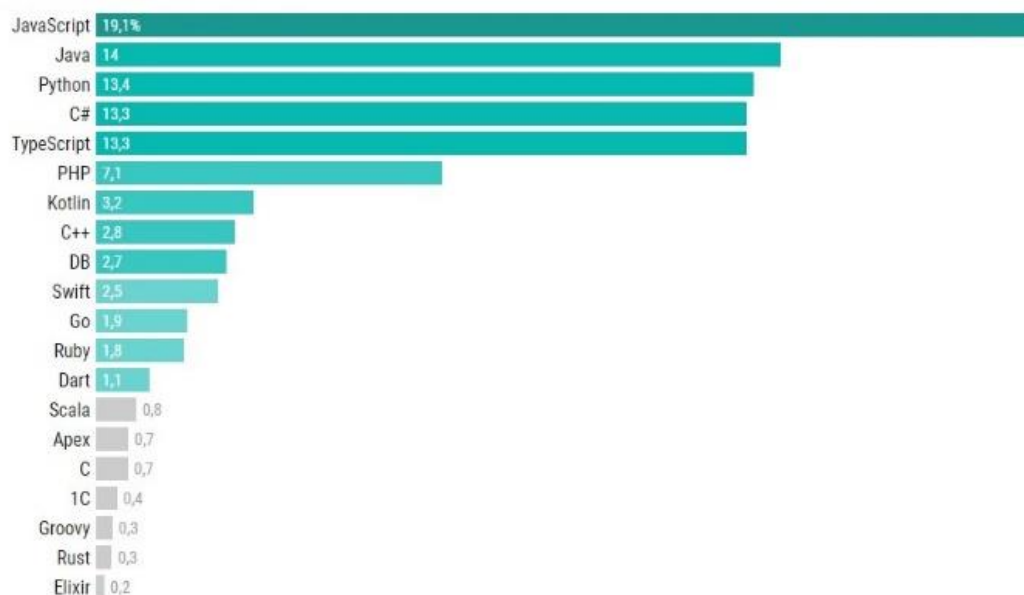


Рисунок 3.1 – Статистика використання мов програмування у розробці мультимедійних мобільних застосунків

Аналізуючи ці дані, можна зробити висновок, що JavaScript на сьогодні є найпопулярнішою мовою програмування у даній предметній області. Разом з тим, мови Python, Java, C# тощо також використовуються, особливо у випадках потреби у більш складній серверній логіці або глибшій інтеграції з апаратними можливостями пристрою.

Для детального порівняння актуальних технологій, що можуть бути використані для реалізації поставленого завдання, складемо порівняльні таблиці за такими критеріями, як: парадигми програмування, типізація, керування пам'яттю, об'єктно-орієнтовані можливості та функціональність [11].

Таблиця 3.1

Порівняльна характеристика за парадигмами

Можливість	C#	Java	C++	PHP	Python	JS
<i>1.</i>	<i>2.</i>	<i>3.</i>	<i>4.</i>	<i>5.</i>	<i>6.</i>	<i>7.</i>
Імперативна	+	+	+	+	+	+
Об'єктно-орієнтована	+	+	+	+	+	+
Функціональна	+/-	+/-	+/-	+/-	+	+
Рефлексивна	+/-	+/-	+	+	+	+

Таблиця 3.2

Порівняльна характеристика за типізацією

Можливість	C#	Java	C++	PHP	Python	JS
<i>1.</i>	<i>2.</i>	<i>3.</i>	<i>4.</i>	<i>5.</i>	<i>6.</i>	<i>7.</i>
Статична типізація	+	+	-	-	-	-
Динамічна типізація	+	-	+	+	+	+
Явна типізація	+/-	+	-	+/-	+/-	-
Неявна типізація	+	-	+	+	+	+
Неявне приведення	-	+	+	+	+	+

Таблиця 3.3

Порівняльна характеристика за можливостями керування пам'яттю

Можливість	C#	Java	C++	PHP	Python	JS
<i>1.</i>	<i>2.</i>	<i>3.</i>	<i>4.</i>	<i>5.</i>	<i>6.</i>	<i>7.</i>
Об'єкти на стеку	+	-	-	-	-	-
Некеровані вказівники	+	-	-	-	-	-
Ручне керування	+	-	-	-	-	-
Збирання сміття	+	+	+	+	+	+

Таблиця 3.4

Порівняльна характеристика за функціональними можливостями

Можливість	C#	Java	C++	PHP	Python	JS
<i>1.</i>	<i>2.</i>	<i>3.</i>	<i>4.</i>	<i>5.</i>	<i>6.</i>	<i>7.</i>
Декларації функцій	-	-	-	-	-	-
First class функції	+	-	+	+	+	+
Анонімні функції	+	+	+	+	+	+
Лексичні замкнення	+	+	+	+	+	+
Часткове застосування	-	-	+	-	+	+

Таблиця 3.5

Порівняльна характеристика за об'єктно – орієнтованими можливостями

Можливість	C#	Java	C++	PHP	Python	JS
<i>1.</i>	<i>2.</i>	<i>3.</i>	<i>4.</i>	<i>5.</i>	<i>6.</i>	<i>7.</i>
Інтерфейси	+	+	+	+	+	-
Мультиметоди	+/-	-	-	-	-	-
Міксини	-	+	+	+	+	+
Перейменування	-	-	-	-	-	-
Множинне спадкування	-	-	-	-	+	-

При проведенню порівняльному аналізі особливу увагу було приділено зручності написання і читання коду, що суттєво впливає на швидкість розробки та супровідність програмного продукту [1-2]. З аналізу видно, що мови Java та C потребують значної кількості коду для реалізації навіть базової логіки, тоді як мови на кшталт JavaScript (зокрема з використанням React Native) дозволяють досягти аналогічного результату з меншими зусиллями завдяки декларативному стилю програмування та наявності великої кількості готових компонентів [3; 5; 10; 17].

При розробці мобільного застосунку для прослуховування пісень особливо важливо враховувати такі можливості, як: кросплатформенна підтримка (Android/iOS), відтворення аудіофайлів, створення і керування

плейлистами, кешування треків, збереження історії прослуховувань, робота з REST API, а також адаптивність інтерфейсу до різних типів пристроїв [3-5; 8-9; 13-15; 18]. Усі ці вимоги повною мірою задовольняє технологічний стек JavaScript + React Native [3-5; 10; 12-13; 15; 17-18].

У контексті реалізації поставленого завдання ця зв'язка має наступні переваги:

1. Можливість написання єдиного коду для двох платформ (iOS і Android);
2. Активна спільнота та постійна підтримка з боку Facebook
3. Велика кількість бібліотек (наприклад, react-navigation, react-native-track-player, axios, expo-av);
4. Проста інтеграція з веб-сервісами та REST API;
5. Широка база знань і навчальних ресурсів;
6. Швидка розробка та оновлення завдяки гарячому перезавантаженню (hot reload);
7. Простий у засвоєнні синтаксис;
8. Велика екосистема модулів і плагінів;
9. Потужні інструменти розробника, такі як Expo;
10. Наявність офіційної документації та великої кількості прикладів реалізації подібних мультимедійних проєктів.

Підіб'ємо підсумки. Кожна мова програмування має свої переваги та недоліки, і вибір залежить від конкретних вимог до програмного продукту. У випадку розробки мобільного застосунку для прослуховування пісень найбільш доцільним є використання JavaScript у поєднанні з React Native, оскільки ця технологія забезпечує кросплатформенність, підтримку роботи з аудіо, гнучкий адаптивний інтерфейс та високу швидкість розробки з використанням великого набору сучасних бібліотек [12].

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		41

3.2. Аналіз існуючих СУБД для вирішення поставленої задачі

У процесі створення інформаційної системи важливу роль відіграє механізм збереження та обробки даних [1]. Саме тому бази даних є критичним компонентом будь-якого застосунку, забезпечуючи надійне зберігання, структуровану організацію та швидкий доступ до інформації [1; 8-9]. У випадку мобільного застосунку для прослуховування пісень база даних використовується для зберігання даних про треки, плейлисти, історію прослуховувань, налаштування користувача тощо. У зв'язку з цим важливо підібрати таку систему управління базами даних (СУБД), яка буде не лише функціональною, але й придатною до використання у мобільному середовищі [1; 8].

Для вибору оптимального варіанту було проведено огляд найбільш популярних сучасних СУБД, що активно застосовуються в практиці розробки застосунків. Розглядалися такі системи, як SQLite, MariaDB, PostgreSQL та Oracle Database. Кожна з них має свої сильні сторони, переваги та особливості, які можуть бути критичними залежно від вимог конкретного проєкту [1].

SQLite є вбудованою реляційною СУБД, яка не потребує окремого серверного середовища і зберігає всю інформацію у локальному файлі. Це робить її надзвичайно зручною для мобільних застосунків, де важлива простота розгортання, швидкий доступ до даних і мінімальна залежність від зовнішніх служб. SQLite відома своєю стабільністю, швидкістю виконання запитів і низьким споживанням ресурсів, що робить її ідеальним вибором для локального збереження даних на мобільних пристроях [8-9].

MariaDB — це розширена версія MySQL з відкритим кодом, яка демонструє високу продуктивність і масштабованість. Вона часто використовується у веб-розробці завдяки підтримці реплікації, кластеризації та гнучкому механізму розширення. Однак для розгортання MariaDB потрібне окреме серверне середовище, що ускладнює її використання в автономних мобільних додатках [1].

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		42

PostgreSQL — потужна об’єктно-реляційна СУБД з відкритим кодом, яка підтримує складні запити, транзакції, розширення і високу відповідність стандарту SQL. Її функціональні можливості перевершують більшість інших безкоштовних СУБД, однак PostgreSQL зазвичай використовується в серверному середовищі і рідше зустрічається в мобільних застосунках, де немає прямого підключення до серверної бази [1].

Oracle Database є комерційною системою, орієнтованою на великі підприємства, з високою продуктивністю, широкими можливостями масштабування, безпеки та резервування. Вона застосовується в критично важливих бізнес-системах, однак через свою складність, вартість та вимоги до інфраструктури практично не використовується в автономних мобільних застосунках [1].

Порівняльну характеристику наведених СУБД у контексті виконання поставленого завдання наведено у таблиці 3.6.

Таблиця 3.6

Порівняльна характеристика розглянутих СУБД

Характеристика	SQLite	Oracle	MariaDB	PostgreSQL
1.	2.	3.	4.	5.
Ліцензія	Public Domain	Комерційна, OTN	GPL	PostgreS License (сумісна з MIT)
Підтримка ACID	Так	Так	Так	Так
Реплікація	Ні	Advanced Replication	Master -Slave	Master-Slave, Multi-Master
Масштабованість	Обмежена	Дуже висока	Висока	Висока
Підтримка JSON	Так (обмежено)	Так (з 12с)	Так	Так
Підтримка NoSQL	Частково	Так (Oracle NoSQL Database)	Так	Так (Hstore, JSONB)
Підтримка сховищ даних	Локальний файл	Так	Так	Так
Розширення	Обмежені	Присутні	Присутні	Присутні
Продуктивність	Висока для локальних застосунків	Дуже висока	Висока	Висока, стабільна

Продовження табл. 3.6

1.	2.	3.	4.	5.
Інструменти адміністрування	Вбудовані, sqlitebrowser	Oracle Enterprise Manager	MyAdmin, Adminer	pgAdmin, DBeaver
Підтримка спільноти	Помірна, але стабільна	Обмежена	Велика, активна	Дуже велика, активна
Сумісність з іншими СУБД	Висока	Висока	Висока (сумісна з MySQL)	Висока
Застосування у мобільній розробці	Широке	Рідко	Переважно для серверів	Обмежено

На основі порівняння видно, що для потреб автономного мобільного застосунку, який має працювати без постійного підключення до сервера, найбільш доцільним є використання SQLite [8-9]. Ця СУБД є надзвичайно легкою, не вимагає розгортання серверного середовища, має мінімальні вимоги до ресурсів пристрою та забезпечує достатній рівень продуктивності для локального зберігання даних. Оскільки у нашому випадку база даних зберігає інформацію про треки, історію прослуховувань і плейлисти без необхідності у складній реплікації чи масштабуванні — використання повноцінних серверних СУБД є недоцільним як з технічної, так і з економічної точки зору [1; 8]. SQLite дозволяє інтегрувати базу даних безпосередньо в застосунок, що спрощує архітектуру, забезпечує автономність роботи та підвищує стабільність [8; 9]. Це робить її оптимальним вибором для мобільного застосунку для прослуховування пісень.

3.3. Програмна реалізація проектованого застосунку

На основі попередньо визначених інструментів розробки, обраної архітектури та структури мобільного застосунку перейдемо до реалізації відповідних функціональних модулів. Застосунок розробляється із використанням мови програмування JavaScript [17] та фреймворку React Native, що дозволяє створювати кросплатформенні мобільні інтерфейси [3; 5;

12; 18]. Функціональні можливості реалізуються поетапно у вигляді окремих модулів. Кожен етап взаємодії користувача із системою реалізується відповідним блоком логіки та інтерфейсу. Перейдемо до реалізації ключових сценаріїв — від реєстрації користувача до роботи з музичним плеєром, пошуком і управлінням плейлистами [1; 3; 5].

Першим кроком реалізуємо механізм реєстрації користувача для забезпечення можливості створення індивідуального облікового запису в мобільному застосунку для прослуховування пісень. Реєстрація є необхідною для персоналізації користувацького досвіду, збереження вподобань та отримання доступу до розширеного функціоналу застосунку [1].

Реєстрація передбачає введення обов'язкових даних: імені, телефону, віку, електронної пошти та пароля. Після створення облікового запису користувач автоматично авторизується в системі. Почнемо з реалізації інтерфейсу екрана RegisterScreen, за допомогою якого користувач може ввести свої дані та ініціювати процес реєстрації [3; 5; 10]:

```
<View style={styles.container}>
  <Text style={styles.title}>Реєстрація</Text>
  <TextInput style={styles.input} placeholder="Ім'я" value={name}
onChangeText={setName} />
  <TextInput style={styles.input} placeholder="Телефон" value={phone}
onChangeText={setPhone} keyboardType="phone-pad" />
  <TextInput style={styles.input} placeholder="Вік" value={age}
onChangeText={setAge} keyboardType="numeric" />
  <TextInput style={styles.input} placeholder="Email" value={email}
onChangeText={setEmail} keyboardType="email-address" />
  <TextInput style={styles.input} placeholder="Пароль" value={password}
onChangeText={setPassword} secureTextEntry />
  <Button title="Зареєструватися" onPress={handleRegister} />
</View>
```

У наведеному фрагменті коду реалізовано інтерфейс екрана реєстрації, що містить текстові поля для введення основних даних користувача та кнопку підтвердження [3; 5]. Усі введені значення зберігаються у відповідних змінних стану, які оновлюються за допомогою функцій setName, setPhone, setAge, setEmail та setPassword [10; 12]. Натискання кнопки "Зареєструватися" ініціює виклик функції handleRegister, яка здійснює перевірку та відправлення даних

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		45

на сервер [3; 4]. Наступним кроком реалізуємо функцію `handleRegister()`, яка відповідає за обробку події реєстрації користувача:

```
const handleRegister = async () => {
  if (!name || !phone || !age || !email || !password) {
    Alert.alert('Помилка', 'Будь ласка, заповніть усі поля.');
```

```
    return;
  }

  try {
    const response = await
fetch('http://192.168.1.104:3001/api/register', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    name,
    phone,
    age: parseInt(age),
    email,
    password
  })
});

    const data = await response.json();

    if (response.status === 201) {
      await AsyncStorage.setItem('currentUser', email);
      signIn(email); // логін після реєстрації
    } else {
      Alert.alert('Помилка', data.message || 'Щось пішло не так');
    }
  } catch (error) {
    console.error('Помилка реєстрації:', error);
    Alert.alert('Помилка', 'Проблема з сервером. Спробуйте пізніше.');
```

```
  }
};
```

У наведеному фрагменті коду реалізовано функцію `handleRegister()`, яка виконує перевірку полів форми на наявність обов'язкових даних [3; 17]. У разі правильного заповнення формується POST-запит до серверного API `/api/register`, у тілі якого передаються дані нового користувача. У разі успішної відповіді (код 201) система зберігає електронну пошту користувача в `AsyncStorage` та виконує вхід у систему за допомогою функції `signIn()`. У протилежному випадку виводиться повідомлення про помилку [3].

З боку сервера логіка обробки запиту реєстрації реалізована у вигляді маршруту `/api/register`, що забезпечує перевірку та збереження даних у базу `SQLite` [6-9; 14]:

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		46

```

app.post('/api/register', async (req, res) => {
    const { name, phone, age, email, password } = req.body;

    try {
        const db = await open({ filename: './db.sqlite', driver:
sqlite3.Database });
        const existingUser = await db.get('SELECT * FROM users WHERE email =
? ', [email]);

        if (existingUser) {
            res.status(400).json({ message: 'Користувач з таким email вже
існує' });
            return;
        }

        await db.run(
            'INSERT INTO users (name, phone, age, email, password) VALUES (?,
?, ?, ?, ?)',
            [name, phone, age, email, password]
        );

        res.status(201).json({ message: 'Користувача успішно зареєстровано'
});
    } catch (error) {
        console.error('Помилка під час реєстрації:', error);
        res.status(500).json({ message: 'Внутрішня помилка сервера' });
    }
});

```

У наведеному фрагменті коду реалізовано серверний маршрут обробки запиту реєстрації нового користувача [6-7]. Система зчитує дані з тіла HTTP-запиту, здійснює перевірку на наявність користувача з такою ж електронною поштою, і в разі його відсутності — додає новий запис до бази даних SQLite [8-9]. У випадку успішного завершення операції сервер надсилає відповідь із кодом 201 [14]. У разі помилки або конфлікту повертається відповідне повідомлення з поясненням [1]. Результат реалізації інтерфейсу реєстрації користувача наведено на рисунку 3.2.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		47

Рисунок 3.2 – Інтерфейс реєстрації користувача у додатку

Таким чином, на цьому етапі реалізовано повноцінний механізм реєстрації користувача із збереженням даних на сервері та подальшою авторизацією. Реєстраційна форма перевіряє обов’язкові поля, надсилає дані через HTTP-запит до API та забезпечує збереження активного сеансу через локальне сховище.

Наступним кроком реалізуємо логіку авторизації користувача, яка забезпечує перевірку введених облікових даних та надає доступ до персоналізованих функцій мобільного застосунку для прослуховування пісень [1]. Авторизація є обов’язковою для ідентифікації користувача, збереження історії прослуховування, керування персональними плейлистами та взаємодії із сервером. У процесі авторизації користувач вводить електронну пошту та пароль, які перевіряються шляхом надсилання запиту до серверної частини, де здійснюється відповідний запит до бази даних [4; 14]. У разі успішної перевірки користувача авторизують у системі, після чого здійснюється перехід до головного інтерфейсу застосунку [3; 5]. Почнемо з реалізації функції обробки авторизації користувача у клієнтській частині застосунку:

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		48

```

const handleLogin = async () => {
  if (!email || !password) {
    Alert.alert('Помилка', 'Будь ласка, введіть email і пароль.');
```

```

    return;
  }

  try {
    const response = await fetch('http://192.168.1.104:3001/api/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email, password })
    });

    const data = await response.json();

    if (response.ok) {
      await AsyncStorage.setItem('currentUser', email);
      signIn(email);
    } else {
      Alert.alert('Помилка', data.message || 'Невірний email або
пароль');
    }
  } catch (error) {
    console.error('Помилка входу:', error);
    Alert.alert('Помилка', 'Проблема з підключенням до сервера');
  }
};

```

У наведеній частині коду реалізовано функцію `handleLogin()`, яка виконує авторизацію користувача [3; 17]. Спочатку система перевіряє, чи заповнені поля електронної пошти та пароля. Далі за допомогою `fetch` відправляється POST-запит до серверного маршруту `/api/login`, де передаються введені дані у форматі JSON [4; 13-14]. Якщо сервер повертає позитивну відповідь, застосунок зберігає адресу електронної пошти користувача у локальному сховищі (`AsyncStorage`) і викликає функцію `signIn`, що оновлює контекст автентифікації [1]. У разі помилки відображається відповідне повідомлення [3]. Наступним кроком реалізуємо інтерфейс екрана авторизації користувача:

```

<View style={styles.container}>
  <Text style={styles.title}>Вхід</Text>

  <TextInput
    style={styles.input}
    placeholder="Email"
    value={email}
    onChangeText={setEmail}
    keyboardType="email-address"
  />

```

```

<TextInput
  style={styles.input}
  placeholder="Пароль"
  value={password}
  onChangeText={setPassword}
  secureTextEntry
/>

<Button title="Увійти" onPress={handleLogin} />

<View style={styles.registerPrompt}>
  <Text style={{ marginRight: 5 }}>Не маєте акаунта?</Text>
  <TouchableOpacity onPress={() => navigation.navigate('Register')}>
    <Text style={styles.registerText}>Зареєструватися</Text>
  </TouchableOpacity>
</View>
</View>

```

У наведеній частині коду реалізовано інтерфейс екрана авторизації [3; 5]. Він містить два текстові поля для введення електронної пошти та пароля, кнопку для входу до системи та елемент TouchableOpacity, який перенаправляє користувача на екран реєстрації у разі відсутності акаунта [3; 15]. Компонування елементів забезпечується за допомогою View, а стилі визначені окремо у стилізаційному об'єкті StyleSheet [3; 5].

На серверному боці реалізуємо обробку запиту авторизації користувача за допомогою маршруту /api/login:

```

app.post('/api/login', (req, res) => {
  const { email, password } = req.body;

  const query = `SELECT * FROM users WHERE email = ? AND password = ?`;
  db.get(query, [email, password], (err, user) => {
    if (err) return res.status(500).json({ message: 'Помилка на сервері' });
    if (!user) return res.status(401).json({ message: 'Невірний email або пароль' });
    res.json({ message: 'Вхід успішний', user });
  });
});

```

У наведеній частині коду реалізовано обробку запиту на авторизацію користувача в Node.js з використанням бази даних SQLite [6-7]. Після отримання POST-запиту, що містить електронну пошту та пароль, сервер формує SQL-запит для перевірки наявності відповідного користувача в

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		50

таблиці users [8-9]. У разі успішного знаходження запису система повертає JSON-відповідь із повідомленням про успішний вхід та об'єктом користувача. Якщо відповідного запису не знайдено, клієнт отримує повідомлення про невірні дані для входу [14]. Також реалізовано обробку можливих помилок при взаємодії з базою даних — наприклад, у разі збоїв підключення або порушення синтаксису SQL-запиту сервер надсилає клієнту повідомлення про внутрішню помилку [1]. Такий підхід забезпечує базовий рівень безпеки та стабільності під час авторизації у системі [14].

Результат реалізації інтерфейсу авторизації користувача наведено на рисунку 3.3.

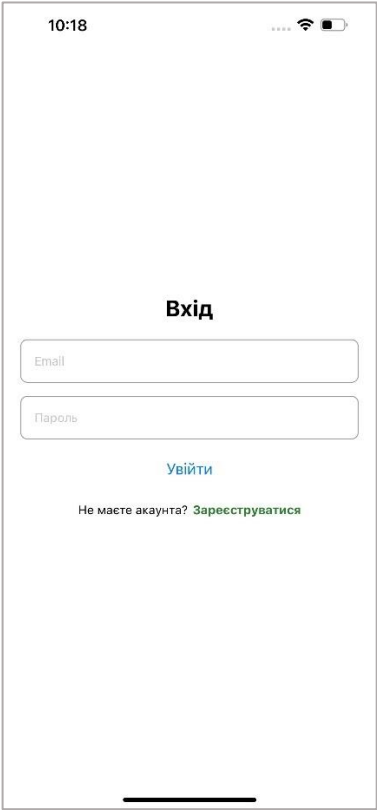


Рисунок 3.3 – Інтерфейс авторизації користувача у додатку

Таким чином, у цьому підрозділі реалізовано повноцінну логіку авторизації, що забезпечує перевірку даних на стороні клієнта та сервера, обробку помилок і безпечний доступ користувача до персоналізованого функціоналу застосунку.

Наступним кроком перейдемо до реалізації головної сторінки мобільного застосунку, яка є ключовим елементом інтерфейсу користувача [1]. На цій сторінці відображається динамічний список музичних треків, що завантажується із серверної бази даних [4; 14]. Основною метою цього етапу є забезпечення можливості користувачу здійснювати перегляд, фільтрацію та пошук доступних композицій, а також перехід до перегляду детальної інформації про конкретний трек [1; 3; 5].

Функціонал головної сторінки має бути зручним, інформативним та адаптованим для мобільних пристроїв. Реалізація включає формування запиту до серверної частини для отримання списку треків, збереження отриманих даних у стані компонента, а також обробку фільтрації та пошуку [4; 10]. Перейдемо до поетапної реалізації логіки даного функціоналу.

```
const MainPage = () => {
  const navigation = useNavigation();
  const [tracks, setTracks] = useState([]);
  const [filteredTracks, setFilteredTracks] = useState([]);
  const [selectedGenre, setSelectedGenre] = useState('Усі жанри');
  const [selectedArtist, setSelectedArtist] = useState('Усі виконавці');
  const [search, setSearch] = useState('');
  const [genres, setGenres] = useState([]);
  const [artists, setArtists] = useState([]);
  const [menuVisibleGenre, setMenuVisibleGenre] = useState(false);
  const [menuVisibleArtist, setMenuVisibleArtist] = useState(false);
  const [loading, setLoading] = useState(true);
```

У наведеній частині коду реалізовано створення функціонального компонента MainPage, який відповідає за логіку головного екрану застосунку [3; 10]. За допомогою хуків useState ініціалізуються необхідні змінні стану: список усіх треків (tracks), відфільтровані треки (filteredTracks), обраний жанр і виконавець, текст пошуку, перелік жанрів і виконавців, а також змінні для контролю відображення випадаючих меню [10; 12]. Для навігації між екранами застосовується хук useNavigation [3; 5; 15].

Наступним кроком перейдемо до реалізації завантаження треків із серверної частини та формування списків жанрів і виконавців.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		52

```

useEffect(() => {
  fetch('http://192.168.1.104:3001/api/tracks')
    .then(res => res.json())
    .then(data => {
      setTracks(data);
      setFilteredTracks(data);

      const genreSet = new Set();
      const artistSet = new Set();
      data.forEach(track => {
        if (track.genre) genreSet.add(track.genre);
        if (track.artist_name) artistSet.add(track.artist_name);
      });
      setGenres(['Усі жанри', ...Array.from(genreSet)]);
      setArtists(['Усі виконавці', ...Array.from(artistSet)]);

      setLoading(false);
    })
    .catch(err => {
      console.error('Помилка завантаження треків:', err);
      setLoading(false);
    });
}, []);

```

У наведеній частині коду реалізовано завантаження даних про музичні треки з серверного API /api/tracks за допомогою методу fetch [4; 13]. Отримані дані зберігаються у стані tracks та filteredTracks [10; 12]. Додатково, з усіх треків формується список унікальних жанрів (genreSet) і виконавців (artistSet), які будуть використані для фільтрації [17]. Ці списки розширюються стандартними значеннями «Усі жанри» та «Усі виконавці». У разі помилки завантаження виводиться відповідне повідомлення у консоль, а індикатор завантаження вимикається [1; 4].

Наступним кроком перейдемо до реалізації логіки фільтрації треків за жанром, виконавцем і ключовими словами.

```

useEffect(() => {
  let filtered = [...tracks];
  if (selectedGenre !== 'Усі жанри') {
    filtered = filtered.filter(track => track.genre === selectedGenre);
  }
  if (selectedArtist !== 'Усі виконавці') {
    filtered = filtered.filter(track => track.artist_name ===
selectedArtist);
  }
  if (search.trim() !== '') {
    filtered = filtered.filter(track =>
track.title.toLowerCase().includes(search.toLowerCase()));
  }
  setFilteredTracks(filtered);
}, [selectedGenre, selectedArtist, search]);

```

У наведеній частині коду реалізовано динамічну фільтрацію масиву tracks відповідно до вибраних користувачем критеріїв: жанру, виконавця та ключових слів у назві [1; 10]. Кожна умова перевіряється окремо — якщо вибрано конкретний жанр (відмінний від «Усі жанри»), список фільтрується відповідно до значення поля genre. Аналогічно відбувається фільтрація за виконавцем через artist_name.

Крім того, реалізований пошук за ключовим словом — функція перевіряє, чи містить назва треку (title) введений текст, незалежно від регістру [17]. Після застосування всіх фільтрів оновлюється стан filteredTracks, який і відображається на головному екрані [10; 12].

Наступним кроком перейдемо до реалізації візуального представлення кожного треку у вигляді клікабельної картки.

```
const renderItem = ({ item }) => (
  <TouchableOpacity
    style={styles.card}
    onPress={() =>
      navigation.navigate('Details', {
        trackId: item.id,
        title: item.title,
        artist: item.artist_name,
        genre: item.genre,
        duration: item.duration,
        cover_image: item.cover_image,
      })
    }
  >
    <Image source={{ uri: item.cover_image }} style={styles.image} />

    <View style={styles.headerRow}>
      <View style={styles.leftColumn}>
        <Text style={styles.name}>{item.title}</Text>
        <View style={styles.typeBox}>
          <Text style={styles.typeText}>{item.genre}</Text>
        </View>
      </View>

      <View style={styles.rightColumn}>
        <Text style={styles.artist}>{item.artist_name}</Text>
        <Text
          style={styles.duration}>{formatDuration(item.duration)}</Text>
        </View>
      </View>
    </TouchableOpacity>
  );
```

У наведеній частині коду реалізовано компонент `renderItem`, який формує візуальне представлення одного музичного треку у вигляді картки [3; 5]. Картка містить обкладинку треку (`Image`), назву композиції, жанр, ім'я виконавця та тривалість. Інформацію структуровано на дві частини: ліва колонка відображає назву та жанр, а права — виконавця та тривалість композиції. Натискання на картку ініціює перехід на екран `Details`, де буде показано детальну інформацію про вибраний трек. Передача параметрів відбувається через об'єкт `navigation.navigate` [3; 15].

Наступним кроком перейдемо до реалізації графічного інтерфейсу екрану — поля пошуку, кнопок вибору жанру та виконавця, а також списку треків.

```
if (loading) {
  return <ActivityIndicator style={{ marginTop: 100 }} size="large"
    color="#2e7d32" />;
}

return (
  <Provider>
    <View style={pageStyles.container}>
      <TextInput
        style={pageStyles.searchInput}
        placeholder="Пошук музики..."
        value={search}
        onChangeText={setSearch}
      />

      <View style={{ flexDirection: 'column' }}>
        <Menu
          visible={menuVisibleGenre}
          onDismiss={() => setMenuVisibleGenre(false)}
          anchor={
            <TouchableOpacity
              style={[pageStyles.button, { marginBottom: 2 }]}
              onPress={() => setMenuVisibleGenre(true)}
            >
              <Text style={pageStyles.buttonText}>Жанр:
{selectedGenre}</Text>
            </TouchableOpacity>
          </Menu>
        >
          {genres.map((genre, index) => (
            <Menu.Item
              key={index}
              onPress={() => {
                setSelectedGenre(genre);
                setMenuVisibleGenre(false);
              }}
            >

```

```

        title={genre}
      />
    ))}
  </Menu>

```

У наведеній частині коду реалізовано умовне відображення індикатора завантаження, який з'являється під час отримання треків із сервера [3; 5]. Після завантаження рендериться основний контейнер інтерфейсу (View) з полем пошуку композицій та першим випадаючим меню для вибору жанру. Список жанрів формується на основі унікальних значень у масиві треків. Вибір значення в меню автоматично оновлює поточний фільтр [10; 12].

Наступним кроком реалізуємо випадаюче меню для вибору виконавця та блок відображення фільтрованого списку треків.

```

<Menu
  visible={menuVisibleArtist}
  onDismiss={() => setMenuVisibleArtist(false)}
  anchor={
    <TouchableOpacity
      style={pageStyles.button}
      onPress={() => setMenuVisibleArtist(true)}
    >
      <Text style={pageStyles.buttonText}>Виконавець:
{selectedArtist}</Text>
    </TouchableOpacity>
  }
  >
    {artists.map((artist, index) => (
      <Menu.Item
        key={index}
        onPress={() => {
          setSelectedArtist(artist);
          setMenuVisibleArtist(false);
        }}
        title={artist}
      />
    ))}
  </Menu>
</View>
<FlatList
  data={filteredTracks}
  keyExtractor={(item) => item.id.toString()}
  renderItem={renderItem}
  contentContainerStyle={styles.list}
/>

```

У наведеній частині коду реалізовано випадаюче меню для вибору виконавця (artist) з використанням компонента Menu з бібліотеки react-native-

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		56

paper. Меню активується натисканням на кнопку, після чого користувач може вибрати одного з виконавців зі сформованого списку. Значення menuVisibleArtist контролює видимість меню, а setSelectedArtist оновлює стан відповідно до вибору користувача [10; 12].

Також реалізовано компонент FlatList, який використовується для ефективного відображення великого списку фільтрованих треків [3; 5]. Атрибут keyExtractor забезпечує унікальність кожного елемента списку через перетворення id у рядок. Метод renderItem, який було описано раніше, відповідає за візуалізацію кожної картки з треком [3].

```
app.get('/api/tracks', (req, res) => {
  db.all(`
    SELECT
      tracks.id,
      tracks.title,
      tracks.genre,
      tracks.duration,
      tracks.cover_image,
      artists.name AS artist_name
    FROM tracks
    JOIN artists ON tracks.artist_id = artists.id
  `, [], (err, rows) => {
    if (err) {
      console.error('✗ Помилка отримання треків:', err.message);
      res.status(500).json({ error: 'Помилка сервера' });
    } else {
      res.json(rows);
    }
  })
})
```

У наведеній частині коду реалізовано серверну логіку для маршруту GET /api/tracks, який забезпечує надсилання списку треків до клієнтського застосунку [6-7; 14]. Запит до бази даних SQLite виконується за допомогою методу db.all(). SQL-запит об'єднує таблиці tracks і artists за допомогою JOIN, щоб отримати повну інформацію про кожен трек, включаючи жанр, тривалість, обкладинку та ім'я виконавця [8-9]. Результат виконання запиту повертається у форматі JSON та використовується клієнтською стороною (у компоненті MainPage) для побудови візуального списку композицій [4; 13]. У випадку виникнення помилки з базою даних, система формує відповідь з кодом 500 та повідомленням про помилку [1; 14].

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		57

Цей серверний маршрут є ключовим джерелом даних для головної сторінки, оскільки саме на його основі формується список композицій, а також генерується перелік жанрів і виконавців для фільтрації [1; 4]. Таким чином, клієнтська та серверна частина взаємодіють через даний запит, що забезпечує динамічне оновлення контенту без необхідності жорстко зашитих даних у застосунку. Це дозволяє масштабувати функціонал у майбутньому без зміни логіки відображення [1]. Результат реалізації інтерфейсу головної сторінки користувача наведено на рисунку 3.4.

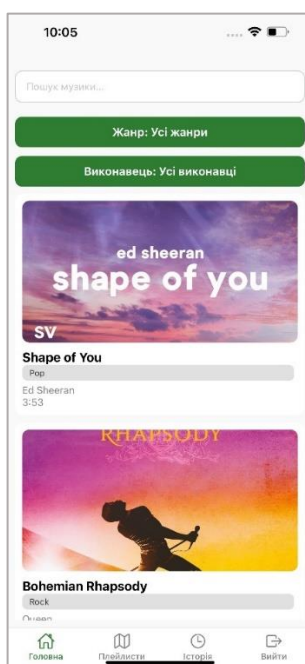


Рисунок 3.4 – Інтерфейс головної сторінки мобільного додатку для прослуховування пісень

Реалізована головна сторінка є центральним елементом взаємодії користувача із системою. Вона забезпечує зручний перегляд усіх доступних треків, дозволяє здійснювати пошук за назвою, а також застосовувати фільтри за жанрами та виконавцями [1; 3]. Отримання та фільтрація даних відбуваються динамічно, що підвищує гнучкість і зручність користування додатком. Водночас, серверна частина забезпечує стабільну підтримку клієнтського функціоналу, обробляючи запити на отримання інформації про композиції [4; 14].

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		58

Наступним кроком перейдемо до реалізації інтерфейсу відтворення обраної пісні, який забезпечує користувачеві зручний доступ до функцій прослуховування, перемотування, перегляду тексту композиції, додавання до плейлистів і перегляду виконавця [1]. Даний компонент базується на бібліотеці expo-av для роботи з аудіо [16], а також реалізує інтеграцію з серверною частиною для отримання треку та збереження історії прослуховувань [4; 14].

```
import { Audio } from 'expo-av';
import Slider from '@react-native-community/slider';
import { useRoute } from '@react-navigation/native';
import AsyncStorage from '@react-native-async-storage/async-storage';

const DetailPage = () => {
  const route = useRoute();
  const { title, artist, genre, duration, cover_image, trackId } =
    route.params;

  const [sound, setSound] = useState(null);
  const [isPlaying, setIsPlaying] = useState(false);
  const [position, setPosition] = useState(0);
  const [durationMillis, setDurationMillis] = useState(1);
  const [showLyrics, setShowLyrics] = useState(false);
  const [lyrics, setLyrics] = useState('');
  const [artistPhoto, setArtistPhoto] = useState(null);
  const [modalVisible, setModalVisible] = useState(false);
  const [playlists, setPlaylists] = useState([]);

  const soundRef = useRef(null);
```

У наведеній частині коду реалізовано підключення необхідних бібліотек для роботи з інтерфейсом (react-native, expo-av, Slider) [3; 5; 16] та збереженням даних користувача (AsyncStorage) [3]. Також виконується ініціалізація стану компонента за допомогою хука useState — зокрема, для збереження аудіооб'єкта, позиції відтворення, тривалості треку, стану відображення тексту пісні, модального вікна та плейлистів. Із маршруту (useRoute) отримуються всі необхідні параметри про трек — такі як його назва, виконавець, жанр, тривалість, обкладинка та унікальний ідентифікатор [3; 15].

Для збереження об'єкта sound після створення використано useRef, що дозволяє зберігати посилання на інстанс аудіо-файлу та згодом його вивантажувати або керувати ним без перерисовки компонента [12]. Наступним

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		59

кроком перейдемо до реалізації логіки завантаження треку з сервера, його відтворення та запису в історію.

```
const loadTrack = async () => {
  try {
    const response = await
    fetch(`http://192.168.1.104:3001/api/track/${trackId}`);
    const data = await response.json();

    setLyrics(data.lyrics || '');

    const { sound } = await Audio.Sound.createAsync(
      { uri: data.file_path },
      {},
      onPlaybackStatusUpdate
    );
    soundRef.current = sound;
    setSound(sound);
    const status = await sound.getStatusAsync();
    setDurationMillis(status.durationMillis);

    const artistResponse = await
    fetch(`http://192.168.1.104:3001/api/artist/by-
    name/${encodeURIComponent(artist)}`);
    const artistData = await artistResponse.json();
    setArtistPhoto(artistData?.photo || null);
  } catch (error) {
    console.error('Помилка завантаження треку або артиста:', error);
  }
};
```

У наведеній частині коду реалізовано логіку завантаження обраного треку за його `trackId`, що передається через параметри маршруту [4; 14]. Після отримання відповіді з сервера зберігається текст пісні (за наявності) у змінній стану `lyrics`. Для відтворення треку створюється об'єкт `Audio.Sound`, який дозволяє взаємодіяти з аудіофайлом, — він зберігається у `soundRef` та `sound`. Крім того, отримується загальна тривалість треку у мілісекундах через `getStatusAsync` [16]. Додатково виконується окремий запит до API для отримання фотографії виконавця — запит здійснюється за іменем, яке кодується у форматі URL через `encodeURIComponent` [17]. Завантажене зображення зберігається у `artistPhoto` [4].

Наступним кроком перейдемо до реалізації функції збереження прослуханого треку в історію.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		60

```

const saveToHistory = async () => {
  try {
    const email = await AsyncStorage.getItem('currentUser');
    if (!email) return;

    await fetch('http://192.168.1.104:3001/api/history', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ user_email: email, track_id: trackId }),
    });

    console.log('📁 Трек збережено в історію');
  } catch (error) {
    console.error('❌ Помилка збереження в історію:', error);
  }
};

```

У наведеній частині коду реалізовано механізм збереження інформації про відтворений трек у серверну історію прослуховування [1; 4]. Спочатку з локального сховища (AsyncStorage) зчитується email поточного користувача [3], і якщо він присутній, формується POST-запит до ендпоінту /api/history. У тілі запиту передається пара значень: user_email та track_id, що дозволяє серверу додати відповідний запис до бази даних у таблицю listening_history [6; 8].

Цей функціонал забезпечує персоналізоване відстеження дій користувача та може використовуватись у майбутньому для формування рекомендацій, аналітики або відображення історії прослуховування [1].

Наступним кроком перейдемо до реалізації механізму додавання треку до обраного плейлиста.

```

const handleAddToPlaylist = async (playlistId) => {
  try {
    const response = await fetch('http://192.168.1.104:3001/api/playlist-tracks', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        playlist_id: parseInt(playlistId),
        track_id: parseInt(trackId),
      }),
    });

    const result = await response.json();
    console.log('⬇️ Результат з сервера:', result);

    if (result.inserted === 1) {

```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		61

```

        alert('🎵 Трек успішно додано до плейлиста!');
    } else if (result.inserted === 0) {
        alert('⚠ Цей трек вже є у вибраному плейлисті.');
```

```

    }

    setModalVisible(false);
} catch (err) {
    console.error('❌ Помилка при додаванні треку до плейлиста:', err);
    alert('Сталася помилка при додаванні треку.');
```

```

}
};
```

У наведеній частині коду реалізовано функцію `handleAddToPlaylist`, яка забезпечує додавання треку до плейлиста користувача [1; 4]. Вона отримує ідентифікатор плейлиста як аргумент та надсилає POST-запит до відповідного API `/api/playlist-tracks`, передаючи у тілі `playlist_id` та `track_id`. Перед відправкою обидва значення перетворюються у числовий формат за допомогою `parseInt`, що гарантує коректність типів [17]. Після відповіді сервера виконується перевірка поля `inserted`, яке сигналізує про результат: 1 — успішно додано, 0 — запис уже існує в плейлисті [14]. У відповідь виводяться повідомлення для користувача та закривається модальне вікно вибору плейлиста. Така перевірка запобігає дублюванню треків та покращує зручність взаємодії.

Наступним кроком перейдемо до реалізації інтерфейсної частини, яка дозволяє користувачу обирати плейлист із модального вікна.

```

return (
  <ScrollView style={styles.container}>
    <Text style={styles.title}>{title}</Text>
    <Text style={styles.genre}>Жанр: {genre}</Text>
    <Image source={{ uri: cover_image }} style={styles.cover} />

    <View style={styles.player}>
      <TouchableOpacity style={styles.playButton}
        onPress={handlePlayPause}>
        <Text style={styles.playButtonText}>{isPlaying ? '⏸' :
'▶'}</Text>
      </TouchableOpacity>
      <Slider
        style={styles.slider}
        value={position}
        minimumValue={0}
        maximumValue={durationMillis}
        onSlidingComplete={handleSeek}
        minimumTrackTintColor="#2e7d32"
        maximumTrackTintColor="#ccc"
      />
    </View>
  </ScrollView>
);
```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		62

```

        />
        <Text style={styles.durationText}>{formatTime(position)} /
{formatTime(durationMillis)}</Text>
    </View>

```

У наведеній частині коду реалізовано базову структуру інтерфейсу сторінки відтворення треку [3; 5]. Виводиться назва пісні, її жанр, зображення обкладинки, а також інтерактивний плеєр із кнопкою відтворення/паузи, слайдером для перемотування та індикатором тривалості. Компонент Slider дає змогу змінювати позицію відтворення, а функції `handlePlayPause` і `handleSeek` реалізують відповідну логіку [16].

Наступним кроком перейдемо до реалізації відображення виконавця, тексту пісні та кнопки додавання в плейлист.

```

<View style={styles.artistContainer}>
  <Image
    source={{ uri: artistPhoto ||
'https://via.placeholder.com/80x80.png?text=No+Photo' }}
    style={styles.artistImage}
  />
  <View style={{ alignItems: 'center' }}>
    <Text style={styles.artistLabel}>Виконавець:</Text>
    <Text style={styles.artistName}>{artist}</Text>
  </View>
</View>

<TouchableOpacity style={styles.lyricsButton} onPress={() =>
setShowLyrics(!showLyrics)}>
  <Text style={styles.lyricsButtonText}>
    {showLyrics ? 'Сховати текст пісні' : 'Переглянути текст пісні'}
  </Text>
</TouchableOpacity>

<TouchableOpacity onPress={() => setModalVisible(true)}
style={styles.addButton}>
  <Text style={styles.addButtonText}>❤️ Додати в плейлист</Text>
</TouchableOpacity>

{showLyrics && (
  <Text style={styles.lyrics}>{lyrics}</Text>
)}

```

У наведеній частині коду реалізовано виведення інформації про виконавця пісні, включаючи його фотографію (або заглушку, якщо фото відсутнє), ім'я та супровідну інформацію [3; 5]. Також реалізовано логіку

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		63

перегляду тексту пісні: натискання кнопки змінює стан `showLyrics`, що керує відображенням тексту. Кнопка додавання до плейлиста відкриває модальне вікно з вибором плейлиста. Логіка цього модального вікна буде розглянута у наступному фрагменті.

```
<Modal visible={modalVisible} transparent animationType="slide">
  <View style={styles.modalOverlay}>
    <View style={styles.modalContent}>
      <Text style={styles.modalTitle}>Оберіть плейлист</Text>
      {playlists.map((pl) => (
        <TouchableOpacity
          key={pl.id}
          onPress={() => handleAddToPlaylist(pl.id)}
          style={styles.playlistItem}
        >
          <Text style={styles.playlistName}>{pl.name}</Text>
        </TouchableOpacity>
      ))}
      <TouchableOpacity onPress={() => setModalVisible(false)}>
        <Text style={styles.cancelText}>Скасувати</Text>
      </TouchableOpacity>
    </View>
  </View>
</Modal>
```

У наведеній частині коду реалізовано модальне вікно для взаємодії користувача з доступними плейлистами [3; 5]. Компонент `<Modal>` відкривається при натисканні на кнопку "Додати в плейлист", контролюється через змінну `modalVisible` та має прозорий фон з анімацією появи (`animationType="slide"`). Всередині модального вікна формується інтерфейс із заголовком та динамічним списком плейлистів, отриманих зі стану `playlists` [3; 10]. Для кожного плейлиста відображається елемент `<TouchableOpacity>`, який при натисканні викликає функцію `handleAddToPlaylist(pl.id)` — тобто додає поточну пісню до вибраного списку відтворення [3; 15]. Окрема кнопка з написом "Скасувати" закриває вікно, змінюючи стан `modalVisible`.

Наступним кроком перейдемо до реалізації серверної частини, яка забезпечує отримання доступних плейлистів для конкретного користувача, а також обробку додавання треку до обраного плейлиста.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		64

```

app.get('/api/track/:id', (req, res) => {
  const trackId = req.params.id;

  const query = `
SELECT tracks.*, artists.name AS artist_name
FROM tracks
JOIN artists ON tracks.artist_id = artists.id
WHERE tracks.id = ?
`;

  db.get(query, [trackId], (err, row) => {
    if (err) {
      console.error('❌ Помилка при отриманні треку:', err);
      res.status(500).json({ error: 'Помилка сервера' });
    } else if (!row) {
      res.status(404).json({ error: 'Трек не знайдено' });
    } else {
      res.json(row);
    }
  });
});

```

У наведеному фрагменті коду реалізовано обробку запиту GET `/api/track/:id` [6-7; 14], яка використовується для отримання повної інформації про конкретний трек. Після отримання id треку з параметрів запиту, система формує SQL-запит до бази даних, що виконує з'єднання таблиці `tracks` із таблицею `artists` для витягування розширених даних. У разі успішного запиту система повертає JSON-об'єкт з усією інформацією про трек, включаючи назву, жанр, тривалість, зображення обкладинки та ім'я виконавця. Якщо трек не знайдено, сервер повертає статус 404 з відповідним повідомленням [14].

Наступним кроком реалізуємо логіку збереження вибраного треку в конкретний плейлист користувача:

```

app.post('/api/playlist-tracks', (req, res) => {
  const { playlist_id, track_id } = req.body;

  console.log('👉 Отримано запит на додавання треку:', { playlist_id, track_id });

  if (!playlist_id || !track_id) {
    return res.status(400).json({ error: 'Невірні дані для додавання' });
  }

  const query = `INSERT OR IGNORE INTO playlist_tracks (playlist_id, track_id) VALUES (?, ?)`;

  db.run(query, [playlist_id, track_id], function (err) {
    if (err) {
      console.error('❌ Помилка додавання треку до плейлиста:',

```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		65

```

err.message);
    return res.status(500).json({ error: 'Не вдалося додати трек' });
}

res.json({
    success: true,
    message: 'Трек додано до плейлиста',
    inserted: this.changes, // 1 якщо вставлено, 0 якщо пропущено
});
});
});

```

У наведеному фрагменті коду реалізовано серверну логіку обробки запиту POST /api/playlist-tracks [6-7; 14], яка відповідає за додавання треку до вибраного плейлиста. Сервер приймає два параметри — playlist_id та track_id — з тіла запиту, перевіряє їхню наявність [1], після чого виконує SQL-запит до таблиці playlist_tracks, уникаючи повторного додавання (за допомогою INSERT OR IGNORE). У разі успіху повертається підтвердження про додавання. Усі потенційні помилки обробляються відповідними повідомленнями [1; 14]. Таким чином, реалізовано повний цикл взаємодії клієнта з сервером під час перегляду треку: отримання детальної інформації про пісню та можливість додавання її до обраного плейлиста.

Результат реалізації інтерфейсу прослуховування пісні наведено на рисунку 3.5.

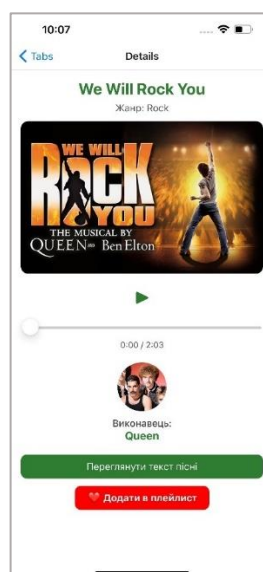


Рисунок 3.5 – Інтерфейс сторінки прослуховування треку у додатку

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		66

Таким чином, на основі реалізованої логіки забезпечено повноцінну функціональність сторінки прослуховування пісні [1]. Користувач може переглядати основну інформацію про трек, керувати відтворенням, переглядати текст композиції, а також додавати її до одного з доступних плейлистів [3]. Уся взаємодія між клієнтською частиною та сервером відбувається асинхронно, що забезпечує швидкий відгук інтерфейсу та зручність користування додатком [4].

Наступним кроком перейдемо до реалізації функціоналу, який забезпечує перегляд, створення та видалення плейлистів, пов'язаних із поточним користувачем [1]. Інтерфейс має на меті забезпечити взаємодію з серверною частиною для отримання списку доступних плейлистів, їх виведення у зручному форматі карток, а також створення нового плейлиста за допомогою модального вікна [4; 14]. Для подальшої підтримки актуального стану даних також буде реалізовано логіку оновлення інформації при поверненні на екран [10; 12].

Перейдемо до програмної реалізації клієнтської частини.

```
import React, { useEffect, useState, useCallback } from 'react';
import { View, Text, FlatList, Image, TouchableOpacity, Modal, TextInput,
StyleSheet, Dimensions } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';
import { useFocusEffect } from '@react-navigation/native';

const { height } = Dimensions.get('window');

const PlaylistPage = ({ navigation }) => {
  const [playlists, setPlaylists] = useState([]);
  const [modalVisible, setModalVisible] = useState(false);
  const [newName, setNewName] = useState('');
  const [userEmail, setUserEmail] = useState('');
```

У наведеній частині коду реалізовано ініціалізацію функціонального компонента `PlaylistPage`, який відповідає за виведення списку плейлистів поточного користувача [3; 10]. Здійснюється імпорт необхідних модулів `React` та елементів інтерфейсу з бібліотеки `react-native`, а також доступ до локального сховища за допомогою модуля `AsyncStorage` [3]. Усередині компонента

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		67

оголошуються стани: `playlists` — для зберігання масиву плейлистів, `modalVisible` — для контролю відображення модального вікна, `newName` — для введення назви нового плейлиста та `userEmail` — для зберігання поточної електронної пошти користувача [10; 12]. Окремо визначається висота екрана пристрою для подальшої адаптивної стилізації [3].

Наступним кроком перейдемо до реалізації логіки завантаження плейлистів та обробки фокусу екрана.

```
useEffect(() => {
  const loadUserAndPlaylists = async () => {
    const storedEmail = await AsyncStorage.getItem('currentUser');
    setUserEmail(storedEmail);
    console.log('📧 Поточний email користувача:', storedEmail);
    fetchPlaylists(storedEmail);
  };
  loadUserAndPlaylists();
}, []);

useFocusEffect(
  useCallback(() => {
    if (userEmail) {
      fetchPlaylists(userEmail);
    }
  }, [userEmail])
);
```

У наведеній частині коду реалізовано логіку ініціалізації екрану та оновлення даних при його повторному фокусі [1]. Використано хук `useEffect`, який при першому завантаженні компонента виконує асинхронну функцію `loadUserAndPlaylists` [10; 12]. Усередині цієї функції за допомогою `AsyncStorage.getItem` отримується email поточного користувача, після чого зберігається у стані за допомогою `setUserEmail` [10]. Одразу після цього викликається функція `fetchPlaylists`, яка надсилає запит до серверної частини та завантажує плейлисти користувача [4].

Додатково використано хук `useFocusEffect` з обгорткою `useCallback`, який відповідає за повторне оновлення списку плейлистів при поверненні на екран (наприклад, після перегляду іншої сторінки) [3; 15]. Це забезпечує

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		68

актуальність відображених даних навіть після змін, здійснених на інших екранах [1].

Наступним кроком перейдемо до реалізації функцій взаємодії з сервером.

```
const fetchPlaylists = async (email) => {  
  const res = await  
  fetch(`http://192.168.1.104:3001/api/playlists/${email}`);  
  const data = await res.json();  
  setPlaylists(data);  
};
```

У наведеній частині коду реалізовано функцію `fetchPlaylists`, яка здійснює запит до серверного API за допомогою методу `fetch` [4; 13]. Запит надсилається на маршрут `/api/playlists/:email`, де `email` є параметром користувача, чия інформація про плейлисти запитується [14]. У разі успішної відповіді результат розпарсується з формату JSON і передається у стан `playlists` за допомогою функції `setPlaylists`. Це дозволяє динамічно оновлювати список плейлистів поточного користувача після завантаження або будь-якої взаємодії з ними [1].

```
const createPlaylist = async () => {  
  if (!newName.trim()) return;  
  await fetch('http://192.168.1.104:3001/api/playlists', {  
    method: 'POST',  
    headers: { 'Content-Type': 'application/json' },  
    body: JSON.stringify({ name: newName.trim(), user_email: userEmail })  
  });  
  setModalVisible(false);  
  setNewName('');  
  fetchPlaylists(userEmail);  
};
```

У наведеній частині коду реалізовано функцію `createPlaylist`, яка відповідає за створення нового плейлиста [1; 4]. Спершу перевіряється, чи заповнене поле назви плейлиста (`newName`). Якщо поле порожнє — функція завершує виконання [17]. Далі за допомогою HTTP-запиту POST надсилаються дані на маршрут `/api/playlists`, де у тілі запиту передаються назва нового плейлиста та електронна адреса користувача. Після успішного

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		69

створення модальне вікно закривається, поле очищується, а список плейлистів оновлюється за допомогою повторного виклику `fetchPlaylists` [10; 12].

```
const deletePlaylist = async (id) => {
  try {
    await fetch(`http://192.168.1.104:3001/api/playlists/${id}`, {
      method: 'DELETE',
    });
    fetchPlaylists(userEmail);
  } catch (error) {
    console.error('Помилка при видаленні плейлиста:', error);
  }
};
```

У наведеній частині коду реалізовано функцію `deletePlaylist`, яка забезпечує видалення плейлиста за його ідентифікатором [; 4]. Функція здійснює HTTP-запит методом `DELETE` на відповідний маршрут серверної частини. У разі успішного видалення викликається оновлення списку плейлистів методом `fetchPlaylists` [10]. Якщо під час виконання запиту виникає помилка, вона фіксується в консолі за допомогою `console.error` [1].

```
const renderItem = ({ item }) => (
  <TouchableOpacity
    style={styles.card}
    onPress={() =>
      navigation.navigate('Playlist', {
        playlistId: item.id,
        name: item.name,
      })
    }
  >
    <Image
      source={{ uri: item.cover ||
        'http://192.168.1.104:3001/photo/template.png' }}
      style={styles.image}
    />
    <View style={styles.infoContainer}>
      <Text style={styles.name}>{item.name}</Text>
      <Text style={styles.count}>{item.track_count || 0}
        трек(ів)</Text>
    </View>
    <TouchableOpacity onPress={() => deletePlaylist(item.id)}
      style={styles.deleteButton}>
      <Text style={styles.deleteText}>❌</Text>
    </TouchableOpacity>
  </TouchableOpacity>
);
```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				70
Змн.	Арк.	№ докум.	Підпис	Дата		

У наведеній частині коду реалізовано функцію renderItem, яка відповідає за відображення кожного елемента у списку плейлистів [3; 5].

Компонент TouchableOpacity виконує роль клікабельної картки, яка при натисканні переходить на сторінку детального перегляду плейлиста через navigation.navigate. Всередині картки виводиться зображення обкладинки плейлиста (item.cover), його назва (item.name) та кількість треків (item.track_count). Якщо зображення не вказане, використовується шаблонна картинка. Додатково передбачено кнопку видалення плейлиста — при її натисканні викликається функція deletePlaylist [1].

```
return (
  <View style={styles.container}>
    <Text style={styles.title}>Ваші плейлисти</Text>

    {playlists.length === 0 ? (
      <View style={styles.centeredTextContainer}>
        <Text style={styles.emptyText}>У вас ще немає
плейлистів</Text>
      </View>
    ) : (
      <FlatList
        data={playlists}
        keyExtractor={(item) => item.id.toString()}
        renderItem={renderItem}
      />
    )}

    <TouchableOpacity style={styles.addButton} onPress={() =>
setModalVisible(true)}>
      <Text style={styles.addButtonText}>Створити плейлист</Text>
    </TouchableOpacity>
  </View>
);
```

У наведеній частині коду реалізовано головне відображення сторінки PlaylistPage [3]. У верхній частині екрана виводиться заголовок. Далі за допомогою умовного рендерингу перевіряється наявність плейлистів у масиві playlists [10]. Якщо плейлисти відсутні, виводиться відповідне повідомлення. У протилежному випадку рендериться список FlatList, де кожен елемент обробляється функцією renderItem [3; 5]. Крім того, нижче на екрані розміщується кнопка для створення нового плейлиста. При натисканні на неї активується модальне вікно через зміну стану modalVisible [3; 10].

```

<Modal visible={modalVisible} transparent animationType="slide">
  <View style={styles.modalView}>
    <Text style={styles.label}>Будь ласка, введіть назву</Text>
    <TextInput
      style={styles.input}
      placeholder="Назва плейлиста"
      value={newName}
      onChangeText={setNewName}
    />
    <TouchableOpacity style={styles.saveButton} onPress={createPlaylist}>
      <Text style={styles.saveButtonText}>Зберегти</Text>
    </TouchableOpacity>
    <TouchableOpacity style={styles.cancelButton} onPress={() =>
      setModalVisible(false)}>
      <Text style={styles.cancelButtonText}>Скасувати</Text>
    </TouchableOpacity>
  </View>
</Modal>

```

У наведеній частині коду реалізовано модальне вікно, яке з'являється після натискання на кнопку створення нового плейлиста [3; 5]. У цьому вікні розміщене текстове поле для введення назви нового плейлиста, а також дві кнопки: одна для підтвердження створення (Зберегти), інша — для закриття вікна без збереження (Скасувати). При натисканні кнопки збереження викликається функція `createPlaylist`, що відповідає за надсилання даних на сервер [3].

Наступним кроком перейдемо до реалізації серверної частини наведеного функціоналу:

```

app.get('/api/playlists/:email', (req, res) => {
  const email = req.params.email;

  db.all(`
    SELECT p.*, COUNT(pt.track_id) AS track_count,
      (SELECT t.cover_image FROM playlist_tracks pt2
        JOIN tracks t ON pt2.track_id = t.id
        WHERE pt2.playlist_id = p.id LIMIT 1) AS cover
    FROM playlists p
    LEFT JOIN playlist_tracks pt ON p.id = pt.playlist_id
    WHERE p.user_email = ?
    GROUP BY p.id
  `, [email], (err, rows) => {
    if (err) {
      console.error('✗ Помилка при вибірці плейлистів:', err.message);
      return res.status(500).json({ error: 'Помилка сервера' });
    }

    res.json(rows);
  });

```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		72

У наведеній частині коду реалізовано серверну обробку запиту на отримання плейлистів для конкретного користувача. Обробник маршруту GET /api/playlists/:email використовує параметр email, щоб відібрати з бази даних усі плейлисти, які належать відповідному користувачеві [8-9]. Для кожного плейлиста також виконується підрахунок кількості треків (COUNT(pt.track_id)) і вибирається зображення обкладинки з будь-якого з треків, що входять у плейлист (через підзапит з LIMIT 1) [8]. Використовується LEFT JOIN, щоб навіть порожні плейлисти (без треків) потрапляли у результат [8; 14].

Наступним кроком перейдемо до реалізації серверної логіки створення нового плейлиста:

```
app.post('/api/playlists', (req, res) => {
  const { name, user_email } = req.body;

  if (!name || !user_email) {
    return res.status(400).json({ error: 'Відсутнє ім'я або email користувача' });
  }

  const query = `INSERT INTO playlists (name, user_email, created_at)
VALUES (?, ?, datetime('now'))`;
  db.run(query, [name, user_email], function (err) {
    if (err) {
      console.error('✗ Помилка додавання плейлиста:', err.message);
      return res.status(500).json({ error: 'Не вдалося додати плейлист' });
    }
    res.json({ success: true, playlist_id: this.lastID });
  });
});
```

У наведеній частині коду реалізовано серверну логіку створення нового плейлиста для користувача [6-7; 14]. Маршрут POST /api/playlists приймає у тілі запиту два параметри — name (назва плейлиста) та user_email (email користувача). Перед виконанням запиту перевіряється наявність обох параметрів. Якщо вхідні дані коректні, формується SQL-запит на вставку нового рядка до таблиці playlists, де автоматично фіксується дата створення за допомогою datetime('now') [8; 9]. У разі успіху сервер повертає ідентифікатор

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		73

нового плейлиста, а в разі помилки — відповідне повідомлення з кодом 500 [1; 14].

Наступним кроком перейдемо до реалізації серверної логіки видалення плейлиста користувача:

```
app.delete('/api/playlists/:id', (req, res) => {
  const playlistId = req.params.id;
  db.run('DELETE FROM playlists WHERE id = ?', [playlistId], function (err)
  {
    if (err) {
      console.error('✗ Помилка при видаленні плейлиста:',
err.message);
      return res.status(500).json({ error: 'Помилка при видаленні
плейлиста' });
    }
    res.json({ message: 'Плейлист видалено' });
  });
});
```

У наведеній частині коду реалізовано маршрут DELETE /api/playlists/:id, який відповідає за видалення плейлиста з бази даних [6-7; 14]. Ідентифікатор плейлиста передається як параметр маршруту :id. За допомогою методу db.run() виконується SQL-запит на видалення рядка з таблиці playlists, де id дорівнює переданому значенню [8-9]. У разі успішного видалення повертається підтвердження з повідомленням Плейлист видалено. У випадку помилки повертається повідомлення про помилку з кодом 500 [1; 14].

Результат реалізації інтерфейсу відображення плейлистів користувача у мобільному додатку наведено на рисунку 3.6.

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		74

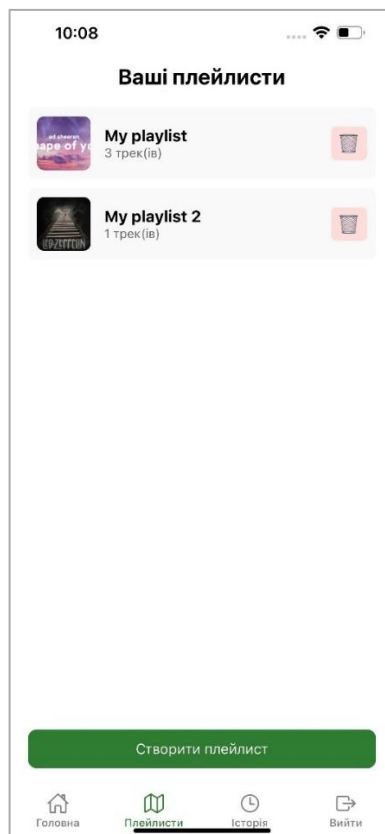


Рисунок 3.6 – Інтерфейс списку плейлистів користувача у мобільному додатку

Таким чином, у межах цього підрозділу було реалізовано функціонал перегляду, створення та видалення плейлистів користувача. Інтерфейс забезпечує зручний візуальний доступ до інформації про кількість треків у кожному плейлисті та дозволяє швидко переходити до перегляду його вмісту. Зв'язок із серверною частиною забезпечується через відповідні маршрути, що дозволяє динамічно працювати з актуальними даними з бази.

Наступним кроком реалізації мобільного застосунку є створення інтерфейсу для перегляду вмісту конкретного плейлиста, який користувач відкриває зі списку [1]. На цій сторінці має відображатися список усіх треків, що входять до складу плейлиста, з можливістю перегляду детальної інформації про кожен з них [3]. Крім того, необхідно реалізувати зручну навігацію — зокрема, можливість повернення до загального переліку плейлистів [15].

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		75

На основі цього функціонального завдання перейдемо до реалізації відповідного компонента, починаючи з ініціалізації сторінки, отримання параметрів маршруту та завантаження вмісту плейлиста.

```
const DetailPlaylist = () => {
  const route = useRoute();
  const navigation = useNavigation();
  const { playlistId, name } = route.params;

  const [tracks, setTracks] = useState([]);

  const fetchTracks = async () => {
    try {
      const res = await fetch(`http://192.168.1.104:3001/api/playlist-tracks/${playlistId}`);
      const data = await res.json();
      setTracks(data);
    } catch (error) {
      console.error('✗ Помилка завантаження треків плейлиста:',
error);
    }
  };
};
```

У наведеній частині коду реалізовано ініціалізацію компонента DetailPlaylist [3], в якому за допомогою хука useRoute отримуються параметри переданого маршруту (ідентифікатор плейлиста та його назва) [3; 15]. Створено локальний стан tracks для збереження списку треків, які входять до плейлиста [10; 12]. Функція fetchTracks виконує HTTP-запит до відповідного API-ендпоінту бекенда для отримання треків за ID плейлиста та зберігає результат у стані [4; 13].

Використання хука useEffect забезпечує автоматичне завантаження даних під час першого рендерингу компонента [10].

Наступним кроком перейдемо до реалізації повторного завантаження треків при фокусі на сторінці.

```
useFocusEffect(
  useCallback(() => {
    fetchTracks();
  }, [playlistId])
);
```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		76

У наведеній частині коду реалізовано механізм оновлення списку треків щоразу, коли користувач повертається на сторінку плейлиста [1]. Для цього використано хук `useFocusEffect` з бібліотеки `@react-navigation/native` [3; 15]. Всередині нього викликається функція `fetchTracks`, що дозволяє завжди відображати актуальні дані без потреби вручну оновлювати інтерфейс [4; 10]. Завдяки цьому рішенню забезпечується динамічність та зручність взаємодії з плейлистом навіть після переходів на інші екрани [10; 12].

Наступним кроком перейдемо до реалізації функціоналу виведення треків у вигляді списку.

```
const renderItem = ({ item }) => (
  <TouchableOpacity
    style={styles.item}
    onPress={() =>
      navigation.navigate('Details', {
        title: item.title,
        artist: item.artist_name,
        genre: item.genre,
        duration: item.duration,
        cover_image: item.cover_image,
        trackId: item.id
      })
    }
  >
    <Image source={{ uri: item.cover_image }} style={styles.image} />
    <View style={styles.infoContainer}>
      <Text style={styles.trackTitle}>{item.title}</Text>
      <Text style={styles.artistName}>{item.artist_name}</Text>
    </View>
  </TouchableOpacity>
);
```

У наведеній частині коду реалізовано функцію `renderItem`, яка використовується для візуалізації кожного окремого елемента списку треків за допомогою `FlatList` [3; 5]. Кожен трек представлено у вигляді клікабельної картки, що містить зображення обкладинки (через компонент `Image`) та текстову інформацію про назву треку і ім'я виконавця. При натисканні на елемент виконується навігація на сторінку `Details`, куди передаються всі необхідні параметри для відображення детальної інформації про обраний трек [15-16].

Наступним кроком перейдемо до реалізації основного інтерфейсу компонента та кнопки повернення.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		77

```

return (
  <View style={styles.container}>
    <Text style={styles.header}>Треки у плейлисті "{name}"</Text>

    <FlatList
      data={tracks}
      keyExtractor={(item) => item.id.toString()}
      renderItem={renderItem}
    />

    <TouchableOpacity style={styles.backButton} onPress={() =>
navigation.goBack()}>
      <Text style={styles.backButtonText}>← Назад до плейлистів</Text>
    </TouchableOpacity>
  </View>
);

```

У наведеній частині коду реалізовано основний інтерфейс компонента, що відповідає за відображення назви плейлиста та списку треків, які до нього входять. Для виведення списку використовується FlatList, який динамічно формує вміст на основі масиву tracks, попередньо завантаженого з сервера [3; 5]. Нижче розташована кнопка повернення, реалізована через TouchableOpacity, яка дозволяє користувачу повернутись назад до списку всіх плейлистів [3; 15].

Наступним кроком перейдемо до опису серверної логіки, яка забезпечує отримання треків певного плейлиста.

```

app.get('/api/playlist-tracks/:playlistId', (req, res) => {
  const playlistId = req.params.playlistId;

  const query = `
    SELECT
      t.id,
      t.title,
      t.genre,
      t.duration,
      t.cover_image,
      a.name AS artist_name
    FROM playlist_tracks pt
    JOIN tracks t ON pt.track_id = t.id
    JOIN artists a ON t.artist_id = a.id
    WHERE pt.playlist_id = ?
  `;

  db.all(query, [playlistId], (err, rows) => {
    if (err) {
      console.error('✗ Помилка при отриманні треків плейлиста:',
err.message);

```

```

        return res.status(500).json({ error: 'Не вдалося отримати треки
плейлиста' });
    }

    res.json(rows);

```

У наведеній частині коду реалізовано обробник HTTP-запиту методом GET, який приймає параметр `playlistId` у складі URL-адреси [6-7; 14]. Цей обробник виконує SQL-запит до бази даних, у якому відбувається приєднання трьох таблиць — `playlist_tracks`, `tracks` та `artists` [8-9]. Завдяки використанню операції JOIN здійснюється агрегація даних, що дозволяє отримати повну інформацію про треки, які входять до конкретного плейлиста. Зокрема, витягуються такі атрибути, як назва треку, жанр, тривалість, URL обкладинки та ім'я виконавця [8].

Цей підхід забезпечує зручність обробки запитів та зменшує навантаження на клієнтську частину, оскільки всі необхідні дані формуються на стороні сервера у вигляді однорідної JSON-структури [1; 4]. Завдяки цьому інтерфейс мобільного додатка може оперативно відображати вміст обраного плейлиста без необхідності виконання додаткових запитів.

Така оптимізація сприяє покращенню швидкодії та зручності користування застосунком [1].

Результат реалізації інтерфейсу відображення треків у плейлисті користувача наведено на рисунку 3.7.

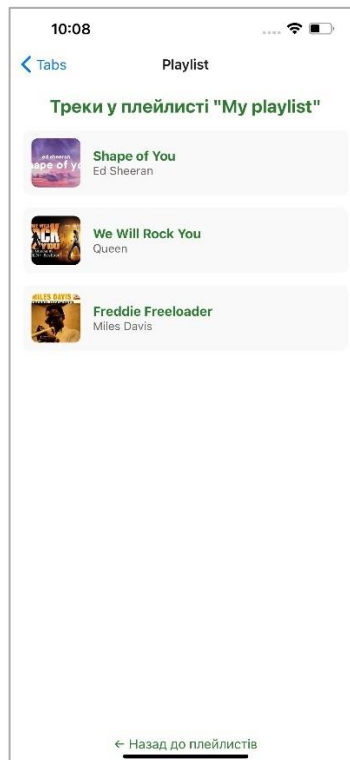


Рисунок 3.7 – Інтерфейс перегляду треків обраного плейлиста у додатку

Таким чином, у межах даного підрозділу було реалізовано функціональність перегляду вмісту конкретного плейлиста [1]. Користувач отримує можливість ознайомитись зі списком треків, що входять до обраного плейлиста, переглянути їх основну інформацію (назву, виконавця, обкладинку) та перейти до детальної інформації про будь-який з треків. Вся необхідна інформація динамічно завантажується з серверної частини за допомогою відповідного API-запиту, що забезпечує цілісну інтеграцію між клієнтським і серверним компонентами застосунку [4].

Наступним кроком перейдемо до реалізації інтерфейсу історії прослуховувань користувача, який надає можливість ознайомитися з раніше прослуханими аудіотреками [1]. Такий функціонал дозволяє покращити взаємодію із застосунком, забезпечуючи збереження персоналізованого досвіду для кожного користувача [1; 3]. Перейдемо до програмної реалізації.

```
const HistoryPage = () => {
  const [history, setHistory] = useState([]);
  const navigation = useNavigation();
```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				80
Змн.	Арк.	№ докум.	Підпис	Дата		

```

useFocusEffect(
  React.useCallback(() => {
    const fetchHistory = async () => {
      const email = await AsyncStorage.getItem('currentUser');
      const res = await
fetch(`http://192.168.1.104:3001/api/listening-history/${email}`);
      const data = await res.json();
      setHistory(data);
    };

    fetchHistory();
  }, [])
);

```

У наведеній частині коду реалізовано логіку отримання історії прослуховувань користувача [1]. При завантаженні або поверненні на сторінку виконується асинхронна функція `fetchHistory`, яка отримує з локального сховища електронну пошту поточного користувача (`AsyncStorage.getItem`) [3]. Далі за цією адресою формується запит до API `/api/listening-history/:email`, результат якого у форматі JSON записується в змінну стану `history` за допомогою `setHistory` [10; 12; 16]. Хук `useFocusEffect` гарантує, що запит виконується щоразу при фокусі на екрані [3; 15].

Наступним кроком перейдемо до реалізації функції відображення елементів історії у вигляді списку.

```

const renderItem = ({ item }) => (
  <TouchableOpacity
    style={styles.item}
    onPress={() => navigation.navigate('Details', {
      title: item.title,
      artist: item.artist_name,
      genre: item.genre,
      duration: item.duration,
      cover_image: item.cover_image,
      trackId: item.id
    })}
  >
    <Image source={{ uri: item.cover_image }} style={styles.image} />
    <View style={styles.infoContainer}>
      <Text style={styles.trackTitle}>{item.title}</Text>
      <Text style={styles.date}>{new
Date(item.listened_at).toLocaleString()}</Text>
    </View>
  </TouchableOpacity>
);

```

У наведеній частині коду реалізовано функцію `renderItem`, яка відповідає за візуалізацію окремого елемента в історії прослуховувань [3]. Для кожного об'єкта `item`, що представляє один трек, створюється інтерактивна область `TouchableOpacity` [3; 5]. Під час натискання виконується навігація на сторінку `Details` з передачею параметрів пісні: назва, виконавець, жанр, тривалість, зображення обкладинки та ідентифікатор треку [3; 15]. Додатково виводиться дата і час прослуховування, яку оброблено методом `toLocaleString()` для зручного формату [17].

Наступним кроком перейдемо до реалізації головного блоку компонента, який включає заголовок, список та логіку порожнього стану.

```
return (
  <View style={styles.container}>
    <Text style={styles.header}>Історія ваших прослуховувань</Text>
    {history.length === 0 ? (
      <Text style={styles.noHistory}>Немає записів у історії</Text>
    ) : (
      <FlatList
        data={history}
        keyExtractor={(item) => item.listened_at + item.id}
        renderItem={renderItem}
      />
    )}
  </View>
);
```

У наведеній частині коду реалізовано повернення основного інтерфейсу сторінки [3]. Компонент `View` містить заголовок, що повідомляє користувача про призначення сторінки, а також умовний рендеринг списку прослуханих треків [10]. Якщо масив `history` порожній, виводиться повідомлення "Немає записів у історії". У протилежному випадку використовується компонент `FlatList` для відображення елементів, де кожен трек форматується через функцію `renderItem` [3; 5]. Для забезпечення унікальності ключів використовується комбінація мітки часу `listened_at` та `id` треку.

```
app.get('/api/listening-history/:email', (req, res) => {
  const user_email = req.params.email;

  console.log('📄 Отримано запит історії для користувача:', user_email);
```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		82

```

const query = `
  SELECT
    h.track_id AS id,
    t.title,
    t.genre,
    t.duration,
    t.cover_image,
    a.name AS artist_name,
    h.played_at AS listened_at
  FROM listening_history h
  JOIN tracks t ON h.track_id = t.id
  JOIN artists a ON t.artist_id = a.id
  WHERE h.user_email = ?
  ORDER BY h.played_at DESC
`;

db.all(query, [user_email], (err, rows) => {
  if (err) {
    console.error('✗ Помилка отримання історії:', err.message);
    return res.status(500).json({ error: 'Не вдалося отримати історію' });
  }

  res.json(rows);
});
});

```

У наведеній частині коду реалізовано серверний маршрут `/api/listening-history/:email` [6-7; 14], який обробляє GET-запити та дозволяє отримати повну історію прослуховувань користувача за вказаною електронною адресою. Для цього використовується SQL-запит, що звертається до таблиці `listening_history` і об'єднує її з таблицями `tracks` та `artists` за допомогою операторів `JOIN` [8-9]. Такий підхід дає змогу сформувати цілісну структуру даних, у якій міститься не лише технічна інформація про факт прослуховування, але й контентна — назва треку, ім'я виконавця, жанр, тривалість та зображення обкладинки [8].

Отримані результати упорядковуються за датою прослуховування, що дозволяє відтворити послідовність взаємодії користувача з аудіоконтентом [8]. Серверна частина надсилає сформований масив об'єктів у форматі JSON до клієнтської частини застосунку, де ці дані відображаються у вигляді зручного списку з візуальними елементами [4; 14]. Такий підхід гарантує ефективне завантаження та високу продуктивність інтерфейсу навіть за наявності великої кількості записів в історії [1].

Результат реалізації інтерфейсу історії прослуховувань наведено на рисунку 3.8.

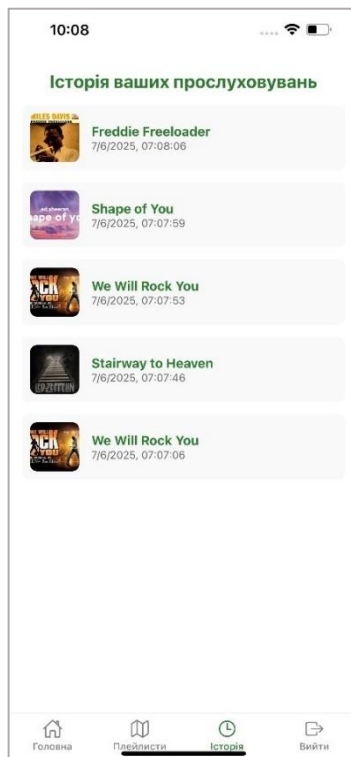


Рисунок 3.8 – Інтерфейс перегляду історії прослуховувань у додатку

Таким чином, інтерфейс історії дозволяє користувачу переглядати, які треки були прослухані, а також переходити до їх деталізованого перегляду при натисканні на відповідний елемент.

3.4. Тестування розробленого додатку

Після завершення етапу розробки мобільного додатка для прослуховування музики наступним кроком стало його тестування з метою комплексної перевірки працездатності всіх модулів, виявлення можливих помилок, а також оцінки загальної якості реалізованого функціоналу. Тестування є необхідним етапом життєвого циклу програмного забезпечення, який дозволяє забезпечити відповідність системи вимогам, гарантувати стабільну роботу і запобігти виникненню несподіваних збоїв під час експлуатації [1].

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		84

Першим кроком у процесі перевірки якості розробленого додатка стало юніт-тестування — методика, що передбачає тестування окремих, ізольованих одиниць коду, або «units»[1; 16]. Метою такого тестування є впевненість у правильності роботи кожного окремого компонента — функцій, API-роутів, інтерфейсних елементів — незалежно від роботи інших частин системи. Юніт-тести допомагають швидко ідентифікувати помилки, що виникають безпосередньо на рівні коду, забезпечуючи тим самим стабільність і надійність програмного забезпечення [1]. Тестування проводилося за допомогою інструменту Postman і внутрішнього логування відповідей сервера [18]. Було перевірено такі основні сценарії:

- Додавання нового плейлиста (POST /api/playlists)
- Отримання плейлистів користувача (GET /api/playlists/:email)
- Додавання треку до плейлиста (POST /api/playlist-tracks)
- Отримання історії прослуховувань (GET /api/listening-history/:email)

У таблиці 3.7 наведено результати юніт-тестування основних серверних API-ендпоінтів:

Таблиця 3.7

Результати юніт-тестування

№	Ендпоінт	Опис тесту	Вхідні дані	Очікувано	Фактично	Статус
1.	2.	3.	4.	5.	6.	7.
1	POST /api/playlists	Створення плейлиста з коректними параметрами	{ name: "Мій плейлист", user_email: "a@b.c" }	HTTP 200 + { success: true, playlist_id }	HTTP 200 + { success: true, playlist_id: 5 }	Успішно
2	POST /api/playlists	Створення плейлиста без назви	{ name: "", user_email: "a@b.c" }	HTTP 400 + { error: ... }	HTTP 400 + { error: '...' }	Успішно
3	GET /api/playlists/:email	Отримання списку плейлистів для існуючого користувача	/api/playlists /a@b.c	HTTP 200 + [{id,name, track_count, cover}, ...]	HTTP 200 + масив об'єктів	Успішно

Продовження табл. 3.7

1.	2.	3.	4.	5.	6.	7.
4	POST /api/playlist-tracks	Додавання треку до плейлиста з коректними параметрами	{ playlist_id: 5, track_id: 12 }	HTTP 200 + { success:true , inserted:1 }	HTTP 200 + { success:true , inserted:1 }	Успішно
5	POST /api/playlist-tracks	Повторне додавання того самого треку	{ playlist_id: 5, track_id: 12 }	HTTP 200 + { inserted:0 }	HTTP 200 + { inserted:0 }	Успішно
6	GET /api/listening-history/:email	Отримання історії прослуховувань для користувача	/api/listening-history/a@b.c	HTTP 200 + масив об'єктів	HTTP 200 + масив об'єктів	Успішно
7	GET /api/listening-history/:email	Запит історії неіснуючого користувача	/api/history/x@y.z	HTTP 200 + []	HTTP 200 + []	Успішно

Результати, представлені в таблиці 3.7, свідчать про високу стабільність та коректність роботи серверної частини додатка [1]. Для кожного ендпоінта було виконано низку тестів із різними вхідними даними, що дозволило комплексно оцінити їхню функціональність. Зокрема, тести на створення плейлистів підтвердили коректність обробки як валідних, так і некоректних запитів [14]. Система успішно створювала нові плейлисти при наявності необхідних параметрів, а у випадку відсутності обов'язкових даних — належним чином повертала помилки з відповідним HTTP-кодом 400, що гарантує надійний захист від некоректних запитів.

Тестування отримання списку плейлистів для конкретного користувача показало, що сервер повертає структуровані та повні дані, включно з кількістю треків у кожному плейлісті та обкладинками, що забезпечує коректне відображення інформації на клієнтській частині [1]. Верифікація додавання треків до плейлистів продемонструвала, що система адекватно опрацьовує як перше додавання, так і повторні спроби додати той самий трек, при цьому

повертаючи зрозумілі відповіді (вставлено або пропущено) [1; 14]. Це знижує ймовірність дублювання та покращує користувацький досвід.

Історія прослуховувань успішно генерується для існуючих користувачів і повертає порожній масив для неіснуючих, що свідчить про коректну обробку різних сценаріїв використання.

Таким чином, юніт-тестування засвідчило відповідність роботи API вимогам проекту, надійність обробки запитів і забезпечення цілісності даних [1; 16]. Виявлені помилки були відсутні або мінімальні, що дозволяє переходити до наступної фази тестування — інтеграційного, у межах якого оцінюється сумісність клієнтської та серверної частин.

Інтеграційне тестування було спрямоване на перевірку взаємодії між клієнтською частиною (React Native) і сервером (Node.js + SQLite) в реальних наскрізних сценаріях [1-2; 6; 8]. Метою було впевнитися, що дані, створені та збережені одним компонентом, коректно обробляються та відображаються іншим, а також що система веде себе стабільно при переходах між різними модулями [1; 13].

Зокрема, перевірялися такі сценарії:

1. Реєстрація й авторизація користувача, а потім автоматичний перехід на головний екран.
2. Створення нового плейлиста у мобільному інтерфейсі та перевірка його відображення в списку.
3. Додавання треку до плейлиста через кнопку «Додати в плейлист» і миттєве оновлення відображення плейлиста.
4. Відтворення треку на екрані детальної інформації та одночасне збереження запису в історію прослуховувань.
5. Перегляд історії прослуховувань одразу після прослуховування і перевірка коректності часу та порядку записів.

Результати проведеного інтеграційного тестування наведені в таблиці 3.8.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		87

Таблиця 3.8

Результати інтеграційного тестування

№	Сценарій	Очікувана поведінка	Фактична поведінка	Статус
1.	2.	3.	4.	5.
1	Реєстрація → авторизація → перехід на головний екран	Автоматичний вхід після реєстрації	Перехід виконується без затримок	Успішно
2	Створення плейлиста в UI	Новий плейлист з'являється	Плейлист відображається	Успішно
3	Додавання треку до плейлиста на екрані деталізації	Трек одразу з'являється в обраному плейлисті	Відображення оновлюється миттєво	Успішно
4	Відтворення треку та повернення до історії прослуховувань	Запис із коректним часом з'являється у верхній частині історії	Запис створюється та відображається	Успішно
5	Переходи між різними екранами (плейлисти → деталі треку → історія → назад)	Інтерфейс без помилок, дані коректно зберігаються	Навігація плавна, без збоїв	Успішно

Після інтеграційного тестування можна констатувати, що всі ключові сценарії взаємодії клієнтської та серверної частин працюють згідно з очікуваннями [1; 19].

Сценарій 1 (Реєстрація/Авторизація) - перехід відразу після реєстрації на головний екран проходить без додаткових затримок чи повторних запитів, що підтверджує коректну передачу токена сесії та роботу механізму AuthContext [1; 3-4].

Сценарій 2 (Створення плейлиста) - створений через інтерфейс плейлист миттєво з'являється у переліку, без необхідності ручного оновлення екрану, завдяки виклику useFocusEffect і повторному запиту на сервер [3; 10; 15].

Сценарій 3 (Додавання треку до плейлиста) - після натискання « Додати в плейлист» вибраний трек з'являється у відповідному плейлисті одразу ж, без перезавантаження всього екрану, що підтверджує успішну роботу маршруту /api/playlist-tracks та клієнтської функції handleAddToPlaylist [1; 4; 14].

Сценарій 4 (Відтворення треку та історія) - аудіо відтворюється коректно, а запис у таблиці `listening_history` з'являється одразу після завершення програвання, на верхівці списку історії, що свідчить про правильну роботу сервера на маршруті `/api/history` та компоненту `HistoryPage` [1; 8; 14].

Сценарій 5 (Навігація) - перехід між усіма екранами (список плейлистів, деталі треку, історія) відбувається плавно, без втрачених станів чи дублювання запитів [3; 15]. Це забезпечує узгодженість роботи `React Navigation` та збереження станів компонентів.

Загалом, інтеграційне тестування підтвердило, що клієнтська та серверна частини додатку працюють у злагодженому режимі, а всі міжмодульні виклики та оновлення стану виконуються коректно та без затримок [1; 19].

Ручне тестування інтерфейсу було проведене з метою оцінити зручність та інтуїтивність взаємодії кінцевого користувача з мобільним додатком [1]. Під час цього етапу на реальному пристрої перевірялися основні сценарії роботи, включно з навігацією між екранами, керуванням відтворенням, створенням і редагуванням плейлистів, а також переглядом історії прослуховувань [3; 5].

Результати ручного тестування наведені в таблиці 3.9.

Таблиця 3.9

Результати ручного тестування інтерфейсу

№	Сценарій	Дія користувача	Очікуваний результат	Фактична поведінка	Статус
1.	2.	3.	4.	5.	6.
1	Реєстрація	Заповнення всіх полів та натискання «Зареєструватися»	Перехід на головний екран, сесія користувача	Перехід без затримок	Успішно
2	Авторизація	Введення email і пароля, натискання «Увійти»	Вхід на головний екран	Вхід одразу	Успішно

Продовження табл.3.9

1.	2.	3.	4.	5.	6.
3	Пошук треків	Введення назви у поле «Пошук музики...»	Фільтрація списку треків	Список одразу оновлюється	Успішно
4	Відтворення треку	Натискання на картку треку	Відкриття екрану відтворення з початком програвання	Програвання без помилок	Успішно
5	Створення плейлиста	Натискання «Створити плейлист», введення назви, «Зберегти»	З'являється новий плейлист у списку	Плейлист відображено одразу	Успішно
6	Додавання треку до плейлиста	На екрані відтворення натискання «Додати в плейлист»	Модальне вікно закривається, трек з'являється в плейлисті	Відображено без затримки	Успішно
7	Перегляд історії	Перехід у розділ «Історія»	Відображення хронологічного списку прослуханих треків	Список коректний	Успішно
8	Навігація між екранами	Перемикання: Головний → Плейлисти → Деталі треку → Історія	Плавне перемикання, збереження стану	Перемикання без помилок	Успішно
9	Вихід із системи	Натискання «Вихід» на головному екрані	Повернення на екран входу.	Повернення одразу, стан очищено	Успішно

Детальний аналіз отриманих результатів показує, що інтерфейс додатка відповідає очікуванням кінцевого користувача:

Сценарії 1–2 (Реєстрація та авторизація) - всі переходи та збереження даних виконуються швидко і без затримок, підтверджуючи коректність роботи контексту аутентифікації [4].

Сценарій 3 (Пошук) - миттєва фільтрація списку при введенні тексту в полі пошуку підвищує зручність користувача, мінімізуючи час пошуку [3].

Сценарій 4 (Відтворення) - відтворення починається одразу після переходу на екран детального перегляду, без додаткових переходів чи затримок [3; 5].

Сценарії 5–6 (Управління плейлистами) - створення нових плейлистів та додавання треків до них відбувається без необхідності вручну оновлювати екран, що свідчить про правильну роботу useFocusEffect і клієнтських API-викликів [3; 10; 12].

Сценарій 7 (Історія) - історія відображається у хронологічному порядку, з датами прослуховувань у зручному форматі, що забезпечує повний огляд активності користувача [3; 5].

Сценарій 8 (Навігація) - переключення між екранами відбувається плавно, без збоїв та втрати станів, що покращує UX [3; 15; 18].

Сценарій 9 (Вихід) - коректне очищення стану користувача гарантує безпечний вихід із системи [3].

Завдяки ручному тестуванню інтерфейсу було підтверджено, що застосунок відповідає сучасним стандартам мобільного UX, забезпечує інтуїтивну навігацію та оперативне відгукування на дії користувача [1].

Проведені методи юніт-тестування, інтеграційного та ручного тестування продемонстрували високу надійність і стабільність розробленого мобільного додатка для прослуховування музики [1]. На рівні окремих функціональних блоків (юніт-тести) усі перевірені API-ендпоінти коректно обробляли як валідні, так і невалідні вхідні дані, гарантуючи цілісність бази даних та адекватні статус-коди у відповідях [14]. Інтеграційні тести підтвердили безперебійний і синхронний обмін даними між клієнтською частиною та сервером за ключовими сценаріями: реєстрація та авторизація, робота з плейлистами, відтворення треків та формування історії прослуховувань [4; 6; 8; 13]. Ручне тестування інтерфейсу засвідчило інтуїтивність навігації, швидкість відгуку елементів керування та адаптивність дизайну на різних розмірах екранів [3; 5; 18].

У результаті комплексного тестування не було виявлено критичних чи навіть середнього рівня дефектів, а всі дрібні UX-неточності були виправлені на етапі відлагодження. Таким чином, тестування повністю підтвердило

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		91

працездатність і готовність додатка до подальшого впровадження та використання кінцевими користувачами [1].

Висновки до розділу 3

У цьому розділі було детально розглянуто та обґрунтовано методологію розробки, а також визначено інструменти, необхідні для реалізації мобільного додатку для прослуховування пісень.

Насамперед, було обрано та адаптовано гнучку методологію розробки Agile з акцентом на Scrum-фреймворк. Такий вибір зумовлений потребою в ітеративному підході до розробки, що дозволяє швидко реагувати на зміни вимог, забезпечувати регулярний зворотний зв'язок від замовника та постійно покращувати продукт. Використання Scrum передбачає розподіл роботи на короткі спринти, що сприяє ефективному плануванню, виконанню та оцінці прогресу проєкту. Це забезпечує прозорість процесу та можливість оперативно вносити корективи.

Далі, було визначено та обґрунтовано стек технологій та програмне забезпечення, необхідне для реалізації проєкту. Для клієнтської частини мобільного додатку обрано React Native, що дозволяє створювати кросплатформенні рішення з єдиною кодовою базою на JavaScript. Для серверної частини обрано Node.js з фреймворком Express.js, що забезпечує високу продуктивність та масштабованість для обробки запитів. Як систему управління базами даних обрано SQLite, що ідеально підходить для локального та невеликого серверного зберігання даних. Ці технологічні рішення забезпечують ефективну взаємодію між компонентами системи та відповідають сучасним стандартам розробки.

Крім того, було визначено допоміжні інструменти розробки, які спрощують та оптимізують робочий процес. Серед них — інтегроване середовище розробки Visual Studio Code, система контролю версій Git з використанням GitHub для командної роботи та керування кодовою базою, а

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		92

також менеджери пакетів npm/Yarn для управління залежностями. Ці інструменти є стандартними у сучасній веб- та мобільній розробці, забезпечуючи високу продуктивність команди та якість коду.

Таким чином, даний розділ заклав організаційні та технологічні основи для успішної реалізації проєкту. Обрана методологія та визначені інструменти забезпечують структуру, ефективність та підтримуваність процесу розробки, що є критично важливим для створення високоякісного та функціонального мобільного додатку.

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		93

ВИСНОВОК

У процесі виконання дипломної роботи було проведено аналіз сучасного стану ринку мобільних застосунків для прослуховування музики, що дозволило виявити основні функціональні потреби користувачів, а також переваги та недоліки існуючих сервісів. На основі проведеного дослідження було сформульовано вимоги до архітектури та інтерфейсу, що стали основою для проєктування та розробки власного мобільного додатку.

У процесі розробки було реалізовано ключовий функціонал застосунку, зокрема: реєстрацію та авторизацію користувачів, перегляд музичного каталогу, детальний перегляд інформації про треки, створення та редагування плейлистів, можливість додавання треків до плейлистів, перегляд історії прослуховувань, а також взаємодію з інтерфейсом програвача. Реалізація здійснена із використанням фреймворку React Native та бази даних SQLite.

Проведене тестування програмного забезпечення підтвердило його стабільність, відповідність технічним вимогам і високу зручність у користуванні. За результатами перевірки встановлено, що мобільний застосунок ефективно виконує всі заявлені функції, забезпечує швидкий доступ до музичного контенту, гнучке управління плейлистами та позитивний досвід взаємодії користувача з інтерфейсом. Отримані результати свідчать про можливість подальшого розширення функціоналу та впровадження продукту в реальних умовах використання.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		94

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sommerville I. Software Engineering. – Boston: Pearson, 2016. – 800 p.
2. Freeman E., Robson E. Head First Design Patterns. – Sebastopol: O'Reilly Media, 2021. – 694 p.
3. Paul A., Kumar N. React Native for Mobile Development. – Birmingham: Packt Publishing, 2017. – 300 p.
4. Gill A. Full Stack Development with React and Node.js. – Birmingham: Packt Publishing, 2020. – 354 p.
5. Jignesh P. Learning React Native: Building Native Mobile Apps with JavaScript. – Sebastopol: O'Reilly Media, 2022. – 340 p.
6. Brown E. Node.js Web Development. – Birmingham: Packt Publishing, 2021. – 436 p.
7. White C. Learning Node.js Development. – Birmingham: Packt Publishing, 2018. – 400 p.
8. Smedley R. Pro SQLite. – Berkeley: Apress, 2016. – 280 p.
9. Patel S. Mastering SQLite Databases. – Birmingham: Packt Publishing, 2019. – 320 p.
10. Wieruch R. The Road to React. – Independently published, 2022. – 250 p.
11. Zakas N.C. Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers. – San Francisco: No Starch Press, 2016. – 352 p.
12. Holmes A., Holmes D. Beginning React Native with Hooks. – Berkeley: Apress, 2020. – 350 p.
13. Ray R. Full-Stack Web Development with React and Node. – Birmingham: Packt Publishing, 2020. – 400 p.
14. Kumar D. Building APIs with Node.js. – Birmingham: Packt Publishing, 2019. – 300 p.
15. Kambli D. React Native Cookbook. – Birmingham: Packt Publishing, 2021. – 450 p.

		Загладько Б.Г.			ІІЗ.КР.Б – 121 – 25 – ІІЗ	Арк.
		Концидайло А.М.				95
Змн.	Арк.	№ докум.	Підпис	Дата		

16. Raschka S. Python Machine Learning. – Birmingham: Packt Publishing, 2020. – 500 p.
17. Mozilla Developer Network. JavaScript documentation [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
18. React Native Documentation [Electronic resource]. – Access mode: <https://reactnative.dev/docs/getting-started>
19. Martin, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. – 464 p.
20. Gamma, Erich, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. – 395 p.
21. McConnell, Steve. Code Complete. 2-ге вид. Microsoft Press, 2004. – 914 p.
22. Norman, Don. The Design of Everyday Things. Basic Books, 2013. – 368 p.
23. Shneiderman, Ben et al. Designing the User Interface: Strategies for Effective Human-Computer Interaction. 6-е вид. Pearson, 2017. – 624 p.
24. Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3-є вид. Prentice Hall, 2005. – 736 p.
25. Rosenberg, Douglass. Applied Information Security: A Hands-on Approach. Wiley, 2017.
26. Vinoski, Steve. “Where Is Middleware?” IEEE Internet Computing, Vol. 6, No. 5, 2002, pp. 88–91.
27. Beck, Kent. Test-Driven Development: By Example. Addison-Wesley, 2003. – 192 p.
28. Fowler, Martin. Refactoring: Improving the Design of Existing Code. 2-е вид. Addison-Wesley, 2018. – 410 p.
29. Gruman, Galen. “Cloud Computing: A Practical Guide.” InfoWorld, 7 July 2008.
30. Simpson, Kyle. You Don’t Know JS: ES6 & Beyond. O’Reilly Media, 2016. – 278 p.

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ІІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		96

ДОДАТКИ

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		97

ДОДАТОК А

Програмний код

App.js:

```
import React, { useState, useEffect } from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import AsyncStorage from '@react-native-async-storage/async-storage';
import { Icons } from '@expo/vector-icons'; // Іконки

import MainPage from '../pages/MainPageFolder/MainPage';
import DetailPage from '../pages/DetailPageFolder/DetailPage';
import DetailPlaylist from
  '../pages/DetailPlaylistPageFolder/DetailPlaylistPage';
import RegisterScreen from '../pages/RegisterPageFolder/RegisterScreen';
import LoginScreen from '../pages/LoginPageFolder/LoginScreen';
import PlayListPage from '../pages/PlayListPageFolder/PlayListPage';
import HistoryPage from '../pages/HistoryPageFolder/HistoryPage';
import LogoutPage from '../pages/LogoutPageFolder/LogoutPage';

import { AuthContext } from '../AuthContext';

const Stack = createStackNavigator();
const Tab = createBottomTabNavigator();

function AppTabs() {
  return (
    <Tab.Navigator
      initialRouteName="Головна"
      screenOptions={({ route }) => ({
        headerShown: false,
        tabBarActiveTintColor: '#2e7d32',
        tabBarInactiveTintColor: 'gray',
        tabBarLabelStyle: { fontSize: 12 },
        tabBarStyle: { backgroundColor: 'white', height: 60 },
        tabBarIcon: ({ color, size }) => {
          let iconName;
          if (route.name === 'Головна') {
            iconName = 'home-outline';
          } else if (route.name === 'Плейлисти') {
            iconName = 'map-outline';
          } else if (route.name === 'Історія') {
            iconName = 'time-outline';
          } else if (route.name === 'Вийти') {
            iconName = 'exit-outline';
          }
          return <Icons name={iconName} size={size}
            color={color} />;
        },
      })}
    >
      <Tab.Screen name="Головна" component={MainPage} />
      <Tab.Screen name="Плейлисти" component={PlayListPage} />
      <Tab.Screen name="Історія" component={HistoryPage} />
      <Tab.Screen name="Вийти" component={LogoutPage} />
    </Tab.Navigator>
  );
}
```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		98

```

}

export default function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(null);

  useEffect(() => {
    const checkLoginStatus = async () => {
      const userToken = await AsyncStorage.getItem('userToken');
      setIsLoggedIn(!!userToken);
    };

    checkLoginStatus();
  }, []);

  const authContext = {
    signIn: async (token) => {
      await AsyncStorage.setItem('userToken', token);
      setIsLoggedIn(true);
    },
    signOut: async () => {
      await AsyncStorage.removeItem('userToken');
      setIsLoggedIn(false);
    }
  };

  if (isLoggedIn === null) {
    return null;
  }

  return (
    <AuthContext.Provider value={authContext}>
      <NavigationContainer>
        {isLoggedIn ? (
          <Stack.Navigator initialRouteName="Tabs">
            <Stack.Screen name="Tabs" component={AppTabs}
options={{ headerShown: false }} />
            <Stack.Screen name="Details" component={DetailPage}
/>
            <Stack.Screen name="Playlist"
component={DetailPlaylist} />
          </Stack.Navigator>
        ) : (
          <Stack.Navigator initialRouteName="Login">
            <Stack.Screen name="Login" component={LoginScreen}
options={{ headerShown: false }} />
            <Stack.Screen name="Register"
component={RegisterScreen} options={{ headerShown: false }} />
          </Stack.Navigator>
        )}
      </NavigationContainer>
    </AuthContext.Provider>
  );
}

```

Server.js:

		Загладько Б.Г			ІІЗ.КР.Б – 121 – 25 – ІІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		99

```

const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = 3001;

app.use(cors());
app.use(bodyParser.json());
app.use('/songs', express.static('songs'));
app.use('/photo', express.static('photo'));

// Підключення до БД
const db = new sqlite3.Database('./db/database.sqlite');

// Реєстрація
app.post('/api/register', (req, res) => {
  const { name, phone, age, email, password } = req.body;

  const query = `
    INSERT INTO users (name, phone, age, email, password)
    VALUES (?, ?, ?, ?, ?)
  `;

  db.run(query, [name, phone, age, email, password], function (err) {
    if (err) {
      if (err.message.includes('UNIQUE constraint failed')) {
        return res.status(400).json({ message: 'Email вже використовується' });
      }
      return res.status(500).json({ message: 'Помилка при реєстрації' });
    }

    res.status(201).json({ message: 'Реєстрація успішна', id: this.lastID });
  });
});

// Вхід
app.post('/api/login', (req, res) => {
  const { email, password } = req.body;

  const query = `SELECT * FROM users WHERE email = ? AND password = ?`;
  db.get(query, [email, password], (err, user) => {
    if (err) return res.status(500).json({ message: 'Помилка на сервері' });
    if (!user) return res.status(401).json({ message: 'Невірний email або пароль' });

    res.json({ message: 'Вхід успішний', user });
  });
});

app.listen(PORT, () => {
  console.log(`Сервер працює на http://localhost:${PORT}`);
});

app.get('/api/tracks', (req, res) => {
  db.all(`

```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		100

```

SELECT
  tracks.id,
  tracks.title,
  tracks.genre,
  tracks.duration,
  tracks.cover_image,
  artists.name AS artist_name
FROM tracks
JOIN artists ON tracks.artist_id = artists.id
`, [], (err, rows) => {
  if (err) {
    console.error('✗ Помилка отримання треків:', err.message);
    res.status(500).json({ error: 'Помилка сервера' });
  } else {
    res.json(rows);
  }
});
});

app.get('/api/track/:id', (req, res) => {
  const trackId = req.params.id;

  const query = `
SELECT tracks.*, artists.name AS artist_name
FROM tracks
JOIN artists ON tracks.artist_id = artists.id
WHERE tracks.id = ?
`;

  db.get(query, [trackId], (err, row) => {
    if (err) {
      console.error('✗ Помилка при отриманні треку:', err);
      res.status(500).json({ error: 'Помилка сервера' });
    } else if (!row) {
      res.status(404).json({ error: 'Трек не знайдено' });
    } else {
      res.json(row);
    }
  });
});

app.post('/api/playlists', (req, res) => {
  const { name, user_email } = req.body;

  if (!name || !user_email) {
    return res.status(400).json({ error: 'Відсутнє ім'я або email користувача' });
  }

  const query = `INSERT INTO playlists (name, user_email, created_at)
VALUES (?, ?, datetime('now'))`;
  db.run(query, [name, user_email], function (err) {
    if (err) {
      console.error('✗ Помилка додавання плейлиста:', err.message);
      return res.status(500).json({ error: 'Не вдалося додати плейлист' });
    }
    res.json({ success: true, playlist_id: this.lastID });
  });
});

```

```

});

app.get('/api/playlists/:email', (req, res) => {
  const email = req.params.email;

  db.all(`
    SELECT p.*, COUNT(pt.track_id) AS track_count,
           (SELECT t.cover_image FROM playlist_tracks pt2
            JOIN tracks t ON pt2.track_id = t.id
            WHERE pt2.playlist_id = p.id LIMIT 1) AS cover
    FROM playlists p
    LEFT JOIN playlist_tracks pt ON p.id = pt.playlist_id
    WHERE p.user_email = ?
    GROUP BY p.id
  `, [email], (err, rows) => {
    if (err) {
      console.error('✗ Помилка при вибірці плейлистів:', err.message);
      return res.status(500).json({ error: 'Помилка сервера' });
    }

    res.json(rows);
  });
});

app.delete('/api/playlists/:id', (req, res) => {
  const playlistId = req.params.id;
  db.run('DELETE FROM playlists WHERE id = ?', [playlistId], function (err) {
    if (err) {
      console.error('✗ Помилка при видаленні плейлиста:',
err.message);
      return res.status(500).json({ error: 'Помилка при видаленні
плейлиста' });
    }
    res.json({ message: 'Плейлист видалено' });
  });
});

app.get('/api/artist/by-name/:name', (req, res) => {
  const artistName = req.params.name;

  db.get(
    `SELECT * FROM artists WHERE name = ?`,
    [artistName],
    (err, row) => {
      if (err) {
        console.error('✗ Помилка при отриманні артиста за ім'ям:',
err.message);
        return res.status(500).json({ error: 'Помилка сервера' });
      }

      if (!row) {
        return res.status(404).json({ error: 'Артиста не знайдено'
});
      }

      res.json(row);
    }
  );
});

app.post('/api/playlist-tracks', (req, res) => {

```

```

    const { playlist_id, track_id } = req.body;

    console.log('👉 Отримано запит на додавання треку:', { playlist_id,
track_id });

    if (!playlist_id || !track_id) {
        return res.status(400).json({ error: 'Невірні дані для додавання' });
    }

    const query = `INSERT OR IGNORE INTO playlist_tracks (playlist_id,
track_id) VALUES (?, ?)`;

    db.run(query, [playlist_id, track_id], function (err) {
        if (err) {
            console.error('❌ Помилка додавання треку до плейлиста:',
err.message);
            return res.status(500).json({ error: 'Не вдалося додати трек' });
        }

        res.json({
            success: true,
            message: 'Трек додано до плейлиста',
            inserted: this.changes, // 1 якщо вставлено, 0 якщо пропущено
        });
    });
});

app.post('/api/history', (req, res) => {
    const { user_email, track_id } = req.body;

    if (!user_email || !track_id) {
        return res.status(400).json({ error: 'Відсутній email або ID треку'
});
    }

    const query = `INSERT INTO listening_history (user_email, track_id,
played_at) VALUES (?, ?, datetime('now'))`;
    db.run(query, [user_email, track_id], function (err) {
        if (err) {
            console.error('❌ Помилка додавання в історію:', err.message);
            return res.status(500).json({ error: 'Не вдалося зберегти в
історію' });
        }

        res.json({ success: true, id: this.lastID });
    });
});

app.get('/api/listening-history/:email', (req, res) => {
    const user_email = req.params.email;

    console.log('👈 Отримано запит історії для користувача:', user_email);

    const query = `
        SELECT
            h.track_id AS id,
            t.title,
            t.genre,
            t.duration,
            t.cover_image,
            a.name AS artist_name,

```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		103

```

        h.played_at AS listened_at
    FROM listening_history h
    JOIN tracks t ON h.track_id = t.id
    JOIN artists a ON t.artist_id = a.id
    WHERE h.user_email = ?
    ORDER BY h.played_at DESC
`;

db.all(query, [user_email], (err, rows) => {
    if (err) {
        console.error('✗ Помилка отримання історії:', err.message);
        return res.status(500).json({ error: 'Не вдалося отримати історію' });
    }

    res.json(rows);
});

app.get('/api/playlist-tracks/:playlistId', (req, res) => {
    const playlistId = req.params.playlistId;

    const query = `
        SELECT
            t.id,
            t.title,
            t.genre,
            t.duration,
            t.cover_image,
            a.name AS artist_name
        FROM playlist_tracks pt
        JOIN tracks t ON pt.track_id = t.id
        JOIN artists a ON t.artist_id = a.id
        WHERE pt.playlist_id = ?
    `;

    db.all(query, [playlistId], (err, rows) => {
        if (err) {
            console.error('✗ Помилка при отриманні треків плейлиста:', err.message);
            return res.status(500).json({ error: 'Не вдалося отримати треки плейлиста' });
        }

        res.json(rows);
    });
});

```

MainPage.js:

```

import React, { useState, useEffect } from 'react';
import { View, Text, TouchableOpacity, FlatList, StyleSheet, Image,
ActivityIndicator, TextInput } from 'react-native';
import { useNavigation } from '@react-navigation/native';
import { Menu, Button, Provider } from 'react-native-paper';
import Ionicons from 'react-native-vector-icons/Ionicons';

const formatDuration = (seconds) => {
    const mins = Math.floor(seconds / 60);
    const secs = seconds % 60;
    return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};

```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		104

```

};

const MainPage = () => {
  const navigation = useNavigation();
  const [tracks, setTracks] = useState([]);
  const [filteredTracks, setFilteredTracks] = useState([]);
  const [selectedGenre, setSelectedGenre] = useState('Усі жанри');
  const [selectedArtist, setSelectedArtist] = useState('Усі виконавці');
  const [search, setSearch] = useState('');
  const [genres, setGenres] = useState([]);
  const [artists, setArtists] = useState([]);
  const [menuVisibleGenre, setMenuVisibleGenre] = useState(false);
  const [menuVisibleArtist, setMenuVisibleArtist] = useState(false);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch('http://192.168.1.104:3001/api/tracks')
      .then(res => res.json())
      .then(data => {
        setTracks(data);
        setFilteredTracks(data);

        const genreSet = new Set();
        const artistSet = new Set();
        data.forEach(track => {
          if (track.genre) genreSet.add(track.genre);
          if (track.artist_name) artistSet.add(track.artist_name);
        });
        setGenres(['Усі жанри', ...Array.from(genreSet)]);
        setArtists(['Усі виконавці', ...Array.from(artistSet)]);

        setLoading(false);
      })
      .catch(err => {
        console.error('Помилка завантаження треків:', err);
        setLoading(false);
      });
  }, []);

  useEffect(() => {
    let filtered = [...tracks];
    if (selectedGenre !== 'Усі жанри') {
      filtered = filtered.filter(track => track.genre ===
selectedGenre);
    }
    if (selectedArtist !== 'Усі виконавці') {
      filtered = filtered.filter(track => track.artist_name ===
selectedArtist);
    }
    if (search.trim() !== '') {
      filtered = filtered.filter(track =>
track.title.toLowerCase().includes(search.toLowerCase()));
    }
    setFilteredTracks(filtered);
  }, [selectedGenre, selectedArtist, search]);

  const renderItem = ({ item }) => (
    <TouchableOpacity
      style={styles.card}
      onPress={() =>
        navigation.navigate('Details', {
          trackId: item.id,
          title: item.title,

```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		105

```

        artist: item.artist_name,
        genre: item.genre,
        duration: item.duration,
        cover_image: item.cover_image,
    })
}
>
<Image source={{ uri: item.cover_image }} style={styles.image} />

<View style={styles.headerRow}>
    {/* Ліва частина */}
    <View style={styles.leftColumn}>
        <Text style={styles.name}>{item.title}</Text>
        <View style={styles.typeBox}>
            <Text style={styles.typeText}>{item.genre}</Text>
        </View>
    </View>

    {/* Права частина */}
    <View style={styles.rightColumn}>
        <Text style={styles.artist}>{item.artist_name}</Text>
        <Text
style={styles.duration}>{formatDuration(item.duration)}</Text>
    </View>
</View>
</TouchableOpacity>
);

if (loading) {
    return <ActivityIndicator style={{ marginTop: 100 }} size="large"
color="#2e7d32" />;
}

return (
    <Provider>
        <View style={pageStyles.container}>
            <TextInput
                style={pageStyles.searchInput}
                placeholder="Пошук музики..."
                value={search}
                onChangeText={setSearch}
            />

            <View style={{ flexDirection: 'column' }}>
                <Menu
                    visible={menuVisibleGenre}
                    onDismiss={() => setMenuVisibleGenre(false)}
                    anchor={
                        <TouchableOpacity
                            style={[pageStyles.button, { marginBottom: 2
}}}
                            onPress={() => setMenuVisibleGenre(true)}
                        >
                            <Text style={pageStyles.buttonText}>Жанр:
{selectedGenre}</Text>
                        </TouchableOpacity>
                    }
                >
                    {genres.map((genre, index) => (
                        <Menu.Item
                            key={index}
                            onPress={() => {

```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				106
Змн.	Арк.	№ докум.	Підпис	Дата		

```

                setSelectedGenre(genre);
                setMenuVisibleGenre(false);
            }}
            title={genre}
        />
    ))}
</Menu>

<Menu
  visible={menuVisibleArtist}
  onDismiss={() => setMenuVisibleArtist(false)}
  anchor={
    <TouchableOpacity
      style={pageStyles.button}
      onPress={() => setMenuVisibleArtist(true)}
    >
      <Text
style={pageStyles.buttonText}>Виконавець: {selectedArtist}</Text>
      </TouchableOpacity>
    </>
  >
    {artists.map((artist, index) => (
      <Menu.Item
        key={index}
        onPress={() => {
          setSelectedArtist(artist);
          setMenuVisibleArtist(false);
        }}
        title={artist}
      />
    ))}
  </Menu>
</View>
<FlatList
  data={filteredTracks}
  keyExtractor={(item) => item.id.toString()}
  renderItem={renderItem}
  contentContainerStyle={styles.list}
/>
</View>
</Provider>
);
};

const pageStyles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 10,
    paddingTop: 80,
    backgroundColor: '#f9f9f9',
  },
  dropdownButton: {
    flex: 1,
    height: 45,
    justifyContent: 'center',
    marginHorizontal: 5,
    backgroundColor: 'ffffff',
    borderColor: '#2e7d32',
    borderWidth: 1,
    borderRadius: 10,
  },
  filterRow: {
    flexDirection: 'row',

```

```

        marginBottom: 10,
        flex: 1,
        justifyContent: 'space-between'
    },
    searchInput: {
        height: 45,
        borderColor: '#ccc',
        borderWidth: 1,
        borderRadius: 10,
        paddingHorizontal: 15,
        marginBottom: 10,
        backgroundColor: '#fff'
    },
    button: {
        backgroundColor: '#2e7d32',
        paddingVertical: 12,
        borderRadius: 10,
        marginVertical: 10,
        alignItems: 'center',
        elevation: 2,
    },
    buttonText: {
        color: '#ffffff',
        fontSize: 16,
        fontWeight: 'bold',
    },
    trackHeader: {
        flexDirection: 'row',
        justifyContent: 'space-between',
        alignItems: 'center',
        marginTop: 10,
    },
    metaInfo: {
        fontSize: 14,
        color: '#666',
        fontStyle: 'italic',
        marginLeft: 10,
    },
    headerRow: {
        flexDirection: 'row',
        justifyContent: 'space-between',
        alignItems: 'flex-start',
        marginTop: 10,
    },
    leftColumn: {
        flex: 0.7,
    },
    rightColumn: {
        flex: 0.3,
        alignItems: 'flex-end',
    },
    },
});

const styles = StyleSheet.create({
    card: {
        backgroundColor: '#ffffff',
        borderRadius: 10,
        padding: 10,
        marginBottom: 10,
        elevation: 2,
    },
});

```

```

    },
    image: {
      width: '100%',
      height: 200,
      borderRadius: 10,
    },
    name: {
      fontSize: 18,
      fontWeight: 'bold',
      marginTop: 10,
    },
    typeContainer: {
      flexDirection: 'row',
      flexWrap: 'wrap',
      marginTop: 5,
    },
    typeBox: {
      backgroundColor: '#e0e0e0',
      borderRadius: 8,
      paddingHorizontal: 10,
      paddingVertical: 4,
      marginRight: 5,
      marginBottom: 5,
    },
    typeText: {
      fontSize: 12,
      color: '#333',
    },
    infoContainer: {
      marginTop: 8,
    },
    artist: {
      fontSize: 14,
      color: 'gray',
    },
    duration: {
      fontSize: 14,
      color: 'gray',
    },
    list: {
      paddingBottom: 20,
    }
  });

export default MainPage;

```

DetailPage.js:

```

import React, { useState, useEffect, useRef } from 'react';
import {
  View, Text, Image, TouchableOpacity,
  StyleSheet, ScrollView, Modal
} from 'react-native';
import { Audio } from 'expo-av';
import Slider from '@react-native-community/slider';
import { useRoute } from '@react-navigation/native';
import AsyncStorage from '@react-native-async-storage/async-storage';

const DetailPage = () => {
  const route = useRoute();
  const { title, artist, genre, duration, cover_image, trackId } =

```

		Загладько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		109

```

route.params;

const [sound, setSound] = useState(null);
const [isPlaying, setIsPlaying] = useState(false);
const [position, setPosition] = useState(0);
const [durationMillis, setDurationMillis] = useState(1);
const [showLyrics, setShowLyrics] = useState(false);
const [lyrics, setLyrics] = useState('');
const [artistPhoto, setArtistPhoto] = useState(null);
const [modalVisible, setModalVisible] = useState(false);
const [playlists, setPlaylists] = useState([]);

const soundRef = useRef(null);

useEffect(() => {
  loadTrack();
  saveToHistory();
  fetchPlaylists();
  return () => {
    if (soundRef.current) {
      soundRef.current.unloadAsync();
    }
  };
}, []);

const loadTrack = async () => {
  try {
    const response = await
fetch(`http://192.168.1.104:3001/api/track/${trackId}`);
    const data = await response.json();

    setLyrics(data.lyrics || '');

    const { sound } = await Audio.Sound.createAsync(
      { uri: data.file_path },
      {},
      onPlaybackStatusUpdate
    );
    soundRef.current = sound;
    setSound(sound);
    const status = await sound.getStatusAsync();
    setDurationMillis(status.durationMillis);

    const artistResponse = await
fetch(`http://192.168.1.104:3001/api/artist/by-
name/${encodeURIComponent(artist)}`);
    const artistData = await artistResponse.json();
    setArtistPhoto(artistData?.photo || null);
  } catch (error) {
    console.error('Помилка завантаження треку або артиста:', error);
  }
};

const saveToHistory = async () => {
  try {
    const email = await AsyncStorage.getItem('currentUser');
    if (!email) return;

    await fetch('http://192.168.1.104:3001/api/history', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ user_email: email, track_id: trackId
    }},

```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		110

```

    });

    console.log('📁 Трек збережено в історію');
  } catch (error) {
    console.error('❌ Помилка збереження в історію:', error);
  }
};

const fetchPlaylists = async () => {
  const email = await AsyncStorage.getItem('currentUser');
  const res = await
fetch(`http://192.168.1.104:3001/api/playlists/${email}`);
  const data = await res.json();
  setPlaylists(data);
};

const onPlaybackStatusUpdate = (status) => {
  if (status.isLoaded) {
    setPosition(status.positionMillis);
    setIsPlaying(status.isPlaying);
  }
};

const handlePlayPause = async () => {
  if (sound) {
    isPlaying ? await sound.pauseAsync() : await sound.playAsync();
  }
};

const handleSeek = async (value) => {
  if (sound) {
    await sound.setPositionAsync(value);
  }
};

const handleAddToPlaylist = async (playlistId) => {
  try {
    const response = await
fetch('http://192.168.1.104:3001/api/playlist-tracks', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        playlist_id: parseInt(playlistId),
        track_id: parseInt(trackId),
      }),
    });

    const result = await response.json();
    console.log('📥 Результат з сервера:', result);

    if (result.inserted === 1) {
      alert('🎵 Трек успішно додано до плейлиста!');
    } else if (result.inserted === 0) {
      alert('⚠️ Цей трек вже є у вибраному плейлисті!');
    }

    setModalVisible(false);
  } catch (err) {
    console.error('❌ Помилка при додаванні треку до плейлиста:',
err);
  }
};

```

```

        alert('Сталася помилка при додаванні треку.');
```

```
    };
```

```
const formatTime = (ms) => {
    const totalSeconds = Math.floor(ms / 1000);
    const mins = Math.floor(totalSeconds / 60);
    const secs = totalSeconds % 60;
    return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};
```

```
return (
    <ScrollView style={styles.container}>
        <Text style={styles.title}>{title}</Text>
        <Text style={styles.genre}>Жанр: {genre}</Text>
        <Image source={{ uri: cover_image }} style={styles.cover} />

        <View style={styles.player}>
            <TouchableOpacity style={styles.playButton}
onPress={handlePlayPause}>
                <Text style={styles.playButtonText}>{isPlaying ? '⏸' :
'▶'}</Text>
```

```
            </TouchableOpacity>
            <Slider
                style={styles.slider}
                value={position}
                minimumValue={0}
                maximumValue={durationMillis}
                onSlidingComplete={handleSeek}
                minimumTrackTintColor="#2e7d32"
                maximumTrackTintColor="#ccc"
            />
```

```
            <Text style={styles.durationText}>{formatTime(position)} /
{formatTime(durationMillis)}</Text>
        </View>
```

```
        <View style={styles.artistContainer}>
            <Image
                source={{ uri: artistPhoto ||
'https://via.placeholder.com/80x80.png?text=No+Photo' }}
                style={styles.artistImage}
            />
            <View style={{ alignItems: 'center' }}>
                <Text style={styles.artistLabel}>Виконавець:</Text>
                <Text style={styles.artistName}>{artist}</Text>
            </View>
        </View>
```

```
        <TouchableOpacity style={styles.lyricsButton} onPress={() =>
setShowLyrics(!showLyrics)}>
            <Text style={styles.lyricsButtonText}>
                {showLyrics ? 'Сховати текст пісні' : 'Переглянути текст
пісні'}
            </Text>
        </TouchableOpacity>
```

```
        <TouchableOpacity onPress={() => setModalVisible(true)}
style={styles.addButton}>
            <Text style={styles.addButtonText}>❤️ Додати в
плейлист</Text>
        </TouchableOpacity>
```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		112

```

        {showLyrics && (
            <Text style={styles.lyrics}>{lyrics}</Text>
        )}

        { /* Модальне вікно */ }
        <Modal visible={modalVisible} transparent animationType="slide">
            <View style={styles.modalOverlay}>
                <View style={styles.modalContent}>
                    <Text style={styles.modalTitle}>Оберіть
плейлист</Text>
                    {playlists.map((pl) => (
                        <TouchableOpacity
                            key={pl.id}
                            onPress={() => handleAddToPlaylist(pl.id)}
                            style={styles.playlistItem}
                        >
                            <Text
                                style={styles.playlistName}>{pl.name}</Text>
                            </TouchableOpacity>
                        ) ) }
                    <TouchableOpacity onPress={() =>
                        setModalVisible(false)}>
                        <Text style={styles.cancelText}>Скасувати</Text>
                    </TouchableOpacity>
                </View>
            </View>
        </Modal>
    </ScrollView>
    );
};

const styles = StyleSheet.create({
    container: {
        padding: 16,
        backgroundColor: '#fff',
    },
    title: {
        fontSize: 26,
        fontWeight: 'bold',
        marginBottom: 6,
        textAlign: 'center',
        color: '#2e7d32',
    },
    genre: {
        fontSize: 16,
        color: '#555',
        marginBottom: 12,
        textAlign: 'center',
    },
    cover: {
        width: '100%',
        height: 250,
        borderRadius: 12,
        marginBottom: 20,
    },
    player: {
        marginBottom: 20,
    },
    playButton: {
        alignSelf: 'center',
        marginBottom: 10,
    },
});

```

```

playButtonText: {
    fontSize: 32,
    color: '#2e7d32',
},
slider: {
    width: '100%',
    height: 40,
},
durationText: {
    textAlign: 'center',
    fontSize: 14,
    color: '#555',
},
artistContainer: {
    alignItems: 'center',
    marginBottom: 20,
},
artistImage: {
    width: 80,
    height: 80,
    borderRadius: 40,
    marginBottom: 10,
    backgroundColor: '#ccc',
},
artistLabel: {
    fontSize: 16,
    fontWeight: '500',
    color: '#444',
},
artistName: {
    fontSize: 18,
    fontWeight: 'bold',
    color: '#2e7d32',
},
addButton: {
    backgroundColor: 'red',
    paddingVertical: 10,
    paddingHorizontal: 20,
    borderRadius: 10,
    alignItems: 'center',
    justifyContent: 'center',
    marginTop: 2,
    marginBottom: 10,
    alignSelf: 'center',
},
addButtonText: {
    color: '#fff',
    fontSize: 16,
    fontWeight: 'bold',
    textAlign: 'center',
},
lyricsButton: {
    backgroundColor: '#2e7d32',
    padding: 10,
    borderRadius: 10,
    alignItems: 'center',
    marginBottom: 10,
},
lyricsButtonText: {
    color: '#fff',
    fontSize: 16,
},
lyrics: {

```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		114

```

        fontSize: 14,
        color: '#333',
        lineHeight: 22,
        marginTop: 10,
      },
      modalOverlay: {
        flex: 1,
        justifyContent: 'center',
        backgroundColor: 'rgba(0,0,0,0.5)',
      },
      modalContent: {
        backgroundColor: '#fff',
        margin: 20,
        padding: 20,
        borderRadius: 10,
        elevation: 5,
      },
      modalTitle: {
        fontSize: 18,
        fontWeight: 'bold',
        marginBottom: 10,
      },
      playlistItem: {
        paddingVertical: 10,
      },
      playlistName: {
        fontSize: 16,
        color: '#2e7d32',
      },
      cancelText: {
        marginTop: 15,
        color: '#b00020',
        textAlign: 'center',
        fontSize: 16,
      },
    },
  ));

export default DetailPage;

```

DetailPlaylistPage.js:

```

import React, { useEffect, useState, useCallback } from 'react';
import {
  View, Text, StyleSheet, FlatList, TouchableOpacity, Image
} from 'react-native';
import { useRoute, useNavigation, useFocusEffect } from '@react-
navigation/native';

const DetailPlaylist = () => {
  const route = useRoute();
  const navigation = useNavigation();
  const { playlistId, name } = route.params;

  const [tracks, setTracks] = useState([]);

  const fetchTracks = async () => {
    try {
      const res = await fetch(`http://192.168.1.104:3001/api/playlist-
tracks/${playlistId}`);
      const data = await res.json();
    }
  }

```

		Загальсько Б.Г.			ІПЗ.КР.Б – 121 – 25 – ІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		115

```

        setTracks(data);
    } catch (error) {
        console.error('❌ Помилка завантаження треків плейлиста:',
error);
    }
};

useEffect(() => {
    fetchTracks();
}, []);

useFocusEffect(
    useCallback(() => {
        fetchTracks();
    }, [playlistId])
);

const renderItem = ({ item }) => (
    <TouchableOpacity
        style={styles.item}
        onPress={() =>
            navigation.navigate('Details', {
                title: item.title,
                artist: item.artist_name,
                genre: item.genre,
                duration: item.duration,
                cover_image: item.cover_image,
                trackId: item.id
            })
        }
    >
        <Image source={{ uri: item.cover_image }} style={styles.image} />
        <View style={styles.infoContainer}>
            <Text style={styles.trackTitle}>{item.title}</Text>
            <Text style={styles.artistName}>{item.artist_name}</Text>
        </View>
    </TouchableOpacity>
);

return (
    <View style={styles.container}>
        <Text style={styles.header}>Треки у плейлисті "{name}"</Text>

        <FlatList
            data={tracks}
            keyExtractor={(item) => item.id.toString()}
            renderItem={renderItem}
        />

        <TouchableOpacity style={styles.backButton} onPress={() =>
navigation.goBack()}>
            <Text style={styles.backButtonText}>← Назад до
плейлистів</Text>
        </TouchableOpacity>
    </View>
);
};

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 16,
        backgroundColor: '#fff',

```

```

    },
    header: {
      fontSize: 22,
      fontWeight: 'bold',
      marginBottom: 16,
      textAlign: 'center',
      color: '#2e7d32',
    },
    item: {
      flexDirection: 'row',
      alignItems: 'center',
      marginBottom: 15,
      padding: 10,
      backgroundColor: '#f9f9f9',
      borderRadius: 10,
      elevation: 1,
    },
    image: {
      width: 60,
      height: 60,
      borderRadius: 8,
      marginRight: 15,
    },
    infoContainer: {
      flex: 1,
    },
    trackTitle: {
      fontSize: 16,
      fontWeight: 'bold',
      color: '#2e7d32',
    },
    artistName: {
      fontSize: 14,
      color: '#666',
    },
  },
});

export default DetailPlaylist;

```

HistoryPage.js:

```

import React, { useState } from 'react';
import {
  View, Text, StyleSheet, FlatList, TouchableOpacity, Image
} from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';
import { useNavigation, useFocusEffect } from '@react-navigation/native';

const HistoryPage = () => {
  const [history, setHistory] = useState([]);
  const navigation = useNavigation();

  useFocusEffect(
    React.useCallback(() => {
      const fetchHistory = async () => {
        const email = await AsyncStorage.getItem('currentUser');
        const res = await
        fetch(`http://192.168.1.104:3001/api/listening-history/${email}`);
        const data = await res.json();

```

		Загладько Б.Г			ІІЗ.КР.Б – 121 – 25 – ІІЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		117

```

        setHistory(data);
    };

    fetchHistory();
}, [])
);

const renderItem = ({ item }) => (
    <TouchableOpacity
        style={styles.item}
        onPress={() => navigation.navigate('Details', {
            title: item.title,
            artist: item.artist_name,
            genre: item.genre,
            duration: item.duration,
            cover_image: item.cover_image,
            trackId: item.id
        })}
    >
        <Image source={{ uri: item.cover_image }} style={styles.image} />
        <View style={styles.infoContainer}>
            <Text style={styles.trackTitle}>{item.title}</Text>
            <Text style={styles.date}>{new
Date(item.listened_at).toLocaleString()}</Text>
        </View>
    </TouchableOpacity>
);

return (
    <View style={styles.container}>
        <Text style={styles.header}>Історія ваших прослуховувань</Text>
        {history.length === 0 ? (
            <Text style={styles.noHistory}>Немає записів у історії</Text>
        ) : (
            <FlatList
                data={history}
                keyExtractor={(item) => item.listened_at + item.id}
                renderItem={renderItem}
            />
        )}
    </View>
);
};

const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 16,
        backgroundColor: '#fff',
    },
    header: {
        fontSize: 22,
        fontWeight: 'bold',
        marginTop: 60,
        marginBottom: 16,
        textAlign: 'center',
        color: '#2e7d32',
    },
    noHistory: {
        textAlign: 'center',
        color: '#999',
        fontSize: 16,
        marginTop: 30,
    },
});

```

```

    },
    item: {
      flexDirection: 'row',
      alignItems: 'center',
      marginBottom: 15,
      padding: 10,
      backgroundColor: '#f9f9f9',
      borderRadius: 10,
      elevation: 1,
    },
    image: {
      width: 60,
      height: 60,
      borderRadius: 8,
      marginRight: 15,
    },
    infoContainer: {
      flex: 1,
    },
    trackTitle: {
      fontSize: 16,
      fontWeight: 'bold',
      color: '#2e7d32',
    },
    date: {
      fontSize: 13,
      color: '#666',
    },
  },
});

export default HistoryPage;

```

LoginPage.js:

```

import React, { useState, useContext } from 'react';
import { View, Text, TextInput, Button, StyleSheet, Alert, TouchableOpacity }
from 'react-native';
import { AuthContext } from '../AuthContext';
import AsyncStorage from '@react-native-async-storage/async-storage';
import { useNavigation } from '@react-navigation/native';

const LoginScreen = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const { signIn } = useContext(AuthContext);
  const navigation = useNavigation();

  const handleLogin = async () => {
    if (!email || !password) {
      Alert.alert('Помилка', 'Будь ласка, введіть email і пароль.');
```

```

      return;
    }

    try {
      const response = await
fetch('http://192.168.1.104:3001/api/login', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ email, password })
});
    }
  };

```

```

        const data = await response.json();

        if (response.ok) {
            await AsyncStorage.setItem('currentUser', email);
            signIn(email);
        } else {
            Alert.alert('Помилка', data.message || 'Невірний email або
пароль');
        }
    } catch (error) {
        console.error('Помилка входу:', error);
        Alert.alert('Помилка', 'Проблема з підключенням до сервера');
    }
};

return (
    <View style={styles.container}>
        <Text style={styles.title}>Вхід</Text>

        <TextInput
            style={styles.input}
            placeholder="Email"
            value={email}
            onChangeText={setEmail}
            keyboardType="email-address"
        />
        <TextInput
            style={styles.input}
            placeholder="Пароль"
            value={password}
            onChangeText={setPassword}
            secureTextEntry
        />

        <Button title="Увійти" onPress={handleLogin} />

        <View style={styles.registerPrompt}>
            <Text style={{ marginRight: 5 }}>Не маєте акаунта?</Text>
            <TouchableOpacity onPress={() =>
navigation.navigate('Register')}>
                <Text style={styles.registerText}>Зареєструватися</Text>
            </TouchableOpacity>
        </View>
    </View>
);
};

const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        paddingHorizontal: 20,
        backgroundColor: '#ffffff',
    },
    title: {
        fontSize: 26,
        fontWeight: 'bold',
        marginBottom: 20,
        alignSelf: 'center',
    },
    input: {
        height: 50,
        borderColor: '#999',

```

```

        borderWidth: 1,
        borderRadius: 8,
        marginBottom: 15,
        paddingHorizontal: 15,
      },
      registerPrompt: {
        marginTop: 20,
        flexDirection: 'row',
        justifyContent: 'center',
        alignItems: 'center',
      },
      registerText: {
        color: '#2e7d32',
        fontWeight: 'bold',
      },
    },
  });

export default LoginScreen;

```

RegisterPage.js:

```

import { useContext } from 'react';
import { AuthContext } from '../../AuthContext';
import React, { useState } from 'react';
import { View, Text, TextInput, Button, StyleSheet, Alert } from 'react-native';
import { useNavigation } from '@react-navigation/native';
import AsyncStorage from '@react-native-async-storage/async-storage';

const RegisterScreen = () => {
  const navigation = useNavigation();
  const [name, setName] = useState('');
  const [phone, setPhone] = useState('');
  const [age, setAge] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const { signIn } = useContext(AuthContext);

  const handleRegister = async () => {
    if (!name || !phone || !age || !email || !password) {
      Alert.alert('Помилка', 'Будь ласка, заповніть усі поля.');
      return;
    }

    try {
      const response = await
fetch('http://192.168.1.104:3001/api/register', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          name,
          phone,
          age: parseInt(age),
          email,
          password
        })
      });

      const data = await response.json();

      if (response.status === 201) {

```

		Загладько Б.Г			ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
		Концидайло А.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		121

```

        await AsyncStorage.setItem('currentUser', email);
        signIn(email); // логін після реєстрації
    } else {
        Alert.alert('Помилка', data.message || 'Щось пішло не так');
    }
} catch (error) {
    console.error('Помилка реєстрації:', error);
    Alert.alert('Помилка', 'Проблема з сервером. Спробуйте
пізніше.');
```

```

    }
};

return (
    <View style={styles.container}>
        <Text style={styles.title}>Реєстрація</Text>

        <TextInput style={styles.input} placeholder="Ім'я" value={name}
onChangeText={setName} />
        <TextInput style={styles.input} placeholder="Телефон"
value={phone} onChangeText={setPhone} keyboardType="phone-pad" />
        <TextInput style={styles.input} placeholder="Вік" value={age}
onChangeText={setAge} keyboardType="numeric" />
        <TextInput style={styles.input} placeholder="Email" value={email}
onChangeText={setEmail} keyboardType="email-address" />
        <TextInput style={styles.input} placeholder="Пароль"
value={password} onChangeText={setPassword} secureTextEntry />

        <Button title="Зареєструватися" onPress={handleRegister} />
    </View>
);
};

const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        paddingHorizontal: 20,
        backgroundColor: '#ffffff',
    },
    title: {
        fontSize: 26,
        fontWeight: 'bold',
        marginBottom: 20,
        alignSelf: 'center',
    },
    input: {
        height: 50,
        borderColor: '#999',
        borderWidth: 1,
        borderRadius: 8,
        marginBottom: 15,
        paddingHorizontal: 15,
    },
});

export default RegisterScreen;
```