НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра інформаційних систем та технологій

Тема “Музичний Програвач”

Курсова робота
З дисципліни «Технології розроблення програмного забезпечення»

Керівник
доц. Амонс О.А.
«Допущений до захисту»
_____

(Особистий підпис керівника)
« » _____2025р.
Захищений з оцінкою
_____
(оцінка)

Виконавець
ст. Тимчук Владислав
залікова книжка № 34 – 20
гр. ІА-34

(особистий підпис виконавця)
« »_____2025р.

Члени комісії:

_____ _____
(особистий підпис) (розшифровка підпису)

_____ _____
(особистий підпис) (розшифровка підпису)

Київ – 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
(назва навчального закладу)
Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ
Дисципліна «Технології розроблення програмного забезпечення»
Курс 3 Група ІА-34 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Тимчука Владислава Андрійовича

(прізвище, ім’я, по батькові)

1.Тема роботи: *Музичний програвач*

2. Строк здачі студентом закінченої роботи 04.12.2025

3. Вихідні дані до роботи:

Музичний програвач становить собою програму для програвання музичних файлів з можливістю створення, запам’ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіоформатів, еквалайзер.

4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці)

ВСТУП, 1 ПРОЄКТУВАННЯ СИСТЕМИ, 2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ, ВИСНОВКИ, ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ, ДОДАТКИ

Додатки:

_____

5.Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

Діаграма варіантів використання та описані сценарії варіантів використання (use-case) Діаграма класів системи (class diagram) Діаграма розгортання системи (deployment diagram) Діаграма станів, що протікають у системі (state diagram) Діаграма послідовностей у системі (state diagram)

6. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Підбір та вивчення літератури	30.09.2025	
2.	Проектування на написання розділу №1	31.10.2025	
3.	Розробка та написання розділу №2	20.11.2025	
4.	Подання курсової роботи на перевірку	25.11.2025	
5.	Захист курсової роботи	08.12.2025	
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			
15.			
16.			
17.			
18.			

Студент **Тимчук Владислав**
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник_____ Олександр АМОНС
(підпис) (Ім’я ПРІЗВИЩЕ)

«____»_____ 2025 р.

ЗМІСТ

ВСТУП

Музичні програвачі — це програми, до яких ми всі давно звикли. Вони потрібні не лише для того, щоб просто ввімкнути аудіофайл, але й щоб упорядкувати власну аудіотеку, зібрати улюблені треки у плейлісти та налаштувати звучання під себе.

Розробка власного музичного програвача — це цікавий та практичний виклик. Такий проєкт одразу змушує працювати з різними форматами файлів, проєктувати зручний інтерфейс, думати про обробку даних і те, як усе це надійно зберегти.

Метою цієї роботи якраз і є створення такого настільного плеєра. Реалізація буде розроблена на Java, а для графічного інтерфейсу - JavaFX. Для серверної логіки (наприклад, керування обліковими записами чи плейлістами) буде використаний Spring Boot, а всі дані зберігатимуться в базі MySQL.

Які функції матиме програвач: Відтворення локальних аудіофайлів; Створення та керування плейлістами; Базові елементи керування: перемішування (shuffle) та повтор треку.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

Сьогодні існує безліч програм для прослуховування музики. Вони дуже різні за можливостями та аудиторією, але загалом їх можна розділити на три групи: класичні десктопні плеєри та стримінгові сервіси.

Серед класичних програм для ПК варто згадати такі відомі інструменти:

- VLC Media Player — безкоштовний, працює на будь-якій системі й відкриває практично всі існуючі формати файлів.
- Audacious — кому потрібен легкий не сильно навантажений плеєр – цей варіант найкращий.
- Strawberry — гарний для колекціонерів, бо дозволяє зручно організувати велику бібліотеку треків та працювати з тегами.

Тепер коротко зачіпимо тему про стримінгові платформи (Spotify, YouTube Music). Вони зав’язані на хмарних технологіях і постійному доступі до інтернету, тому їхня архітектура є набагато складнішою.

Проаналізувавши існуючі рішення, стає зрозуміло, що навіть базовий плеєр повинен мати функції, до яких звикли користувачі: перемішування, повтор треку та еквалайзер. Також критично важливо вміти зчитувати й зберігати інформацію про пісню (виконавець, альбом, назва тощо), а це вже вимагає продуманої структури даних та підключення бази даних.

Часто бачив у відкритих проєктах музичних програвачів що реалізована лише клієнтська частина. Це відкриває чудову можливість для навчання: створити систему, де є і клієнт, і повноцінний сервер.

У своїй роботі я планую розробити саме такий застосунок: функціонально завершений плеєр із клієнт-серверною архітектурою та базою даних для плейлістів. Я хочу зосередитися на головному — правильному проєктуванні, чистій архітектурі та взаємодії компонентів.

1.2. Загальний опис проєкту

Наш проєкт — це десктопний музичний плеєр. Його головне завдання — дати користувачеві зручний інструмент для прослуховування музики та організації власної аудіотеки. Архітектурно система складається з двох частин: клієнтського застосунку та сервера (REST API), а всі дані зберігаються у реляційній базі даних.

Клієнтська частина: Це те, що бачить користувач. Тут реалізовано графічний інтерфейс та всі функції керування відтворенням: старт, пауза, перемикання треків, режими перемішування та повтору. Клієнт відповідає за роботу з локальними файлами (підтримуються формати MP3, WAV та інші) і спілкується з сервером, щоб завантажувати або зберігати інформацію.

Серверна частина: Виступає центральним сховищем даних. Сервер обробляє запити на пошук, додавання чи видалення треків. Через спеціальні API-ендпоїнти він дозволяє клієнту отримувати списки пісень, зберігати створені плейлісти та оновлювати їхню структуру.

Особливості: Користувач може створювати власні плейлісти, редагувати їх і зберігати на сервері для подальшого використання. Щоб зробити код надійним та гнучким, у процесі розробки ми плануємо застосовувати патерни проєктування там, де цього вимагатиме архітектура системи.

1.3. Вимоги до застосунків системи

1.3.1 Функціональні вимоги до системи

№	Функціональна вимога	Опис
1	Відтворення аудіофайлів	Система повинна відтворювати локальні аудіофайли форматів MP3, WAV, AAC
2	Керування відтворенням	Користувач має мати можливість запускати, зупиняти та призупиняти відтворення треку.
3	Зміна гучності	Система повинна надавати засоби зміни рівня гучності під час відтворення аудіо.В
4	Перемотування/зміна позиції відтворення	Користувач повинен мати можливість змінювати поточну позицію відтворення треку (перемотка).
5	Перехід між треками	Система повинна забезпечувати перехід до наступного та попереднього треку у плейлисті.
6	Режим перемішування	Користувач повинен мати можливість увімкнути випадкове відтворення треків плейлиста.
7	Режим повторення	Система повинна підтримувати режими повторення поточного треку та всього плейлиста.
8	Створення плейлиста	Користувач повинен мати можливість створювати нові плейлисти.
9	Редагування плейлиста	Система повинна дозволяти додавати, видаляти та змінювати порядок треків у плейлисті.
10	Збереження плейлиста	Плейлисти повинні зберігатися у базі даних через серверну частину.
11	Імпорт локальних аудіофайлів	Система повинна надавати можливість вибору та імпорту локальних аудіофайлів у бібліотеку.
12	Перегляд списку доступних треків	Користувач повинен бачити список доступних треків з основними метаданими (назва, виконавець тощо).
13	Отримання метаданих треку	Система повинна отримувати та відображати метадані треку (назва, виконавець, альбом, тривалість).
14	Client–Server синхронізація	Клієнтська частина повинна отримувати та зберігати дані про треки й плейлисти через REST API.
15	CRUD-операції над плейлистами на сервері	Серверна частина повинна підтримувати створення, читання, оновлення та видалення плейлистів.
16	CRUD-операції над треками на сервері	Серверна частина повинна підтримувати збереження та оновлення інформації про треки.
17	Пошук треків	Користувач повинен мати можливість здійснювати пошук треків за назвою.

Таблиця 1. – Функціональні вимоги до системи

1.3.2 Нефункціональні вимоги до системи

№	Нефункціональна вимога	Опис
1	Продуктивність відтворення	Відтворення аудіо має відбуватися без помітних затримок та переривань на цільовій платформі.
2	Продуктивність сервера	Час відповіді сервера на запити в повинен становити менше 200 мс.
3	Надійність обробки помилок	Система повинна коректно обробляти помилки без аварійного завершення.
4	Збереження даних	Плейлисти та інформація про треки мають зберігатися у базі даних і бути доступними після перезапуску системи.
5	Однорідність інтерфейсу	Усі елементи керування мають мати єдиний стиль та логічне розташування.
6	Розширюваність	Архітектура системи повинна дозволяти додавання нових аудіоформатів та розширення функціоналу без значної переробки існуючого коду.
7	Вимоги до інсталяції	Клієнтський застосунок повинен коректно працювати на операційній системі Windows
8	Логування	Сервер повинен виконувати базову валідацію вхідних даних і не допускати виконання некоректних або небезпечних запитів.
9	Підтримуваність	Система повинна забезпечувати базове логування ключових подій (помилки, звернення до сервера, критичні операції).

Таблиця 2. – Нефункціональні вимоги до системи

1.4 Сценарії використання системи

Сценарій 1: **Відтворення треку:**

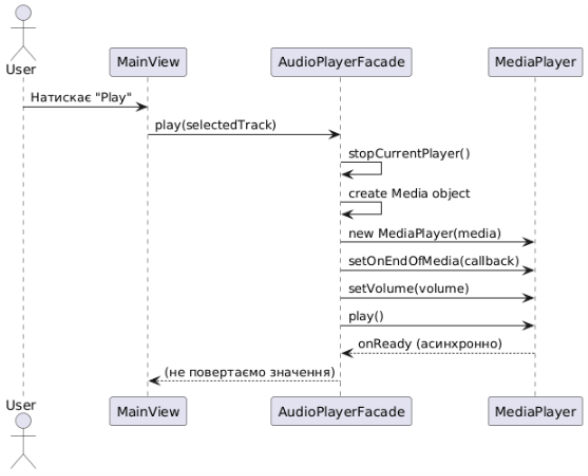

Рис. 1 – Сценарій №1 (Діаграма послідовностей)

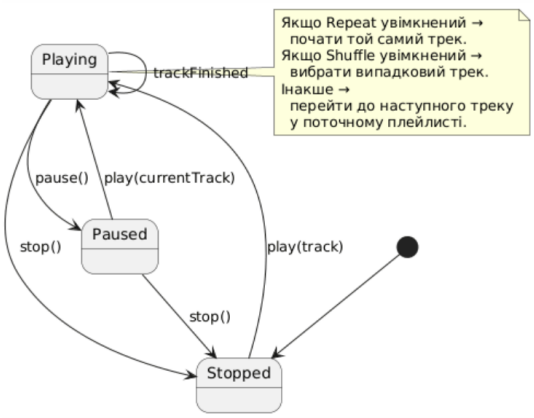

Рис. 2 – Сценарій №1 (Діаграма станів)

Сценарій 2: **Перехід до наступного/попереднього треку:**

Крок	Опис
1	Під час відтворення треку користувач натискає кнопку **Next** або **Prev**.
2	Система визначає наступний або попередній трек у поточному плейлисті.
3	Якщо активовано режим **Shuffle**, система обирає випадковий трек.
4	Поточний трек зупиняється.
5	Система запускає відтворення нового треку.
6	*Якщо трек був останнім - система переходить до першого і навпаки.*

Таблиця 3 – Сценарій №2

Сценарій 3: **Створення нового плейлиста:**

Крок	Опис дії
1	Користувач натискає кнопку Створити плейлист.
2	Система пропонує ввести назву нового плейлиста.
3	Користувач вводить назву та підтверджує створення.
4	Система створює порожній плейлист.
5	Користувач додає треки з провідника.
6	Зміни зберігаються локально у клієнтській частині.
7	Користувач за бажанням може надіслати плейлист на сервер для збереження.

Таблиця 4 – Сценарій №3

Сценарій 4: **Імпорт аудіофайлів:**

Крок	Опис дії
1	Користувач натискає кнопку **Import Tracks**.
2	Система відкриває провідник для вибору файлів.
3	Користувач вибирає аудіофайл
4	Система додає вибрані файли до внутрішньої музичної бібліотеки.
5	Оновлений список треків відображається в інтерфейсі користувача.

Таблиця 5 – Сценарій №4

Сценарій 5: **Збереження плейліста на сервер:**

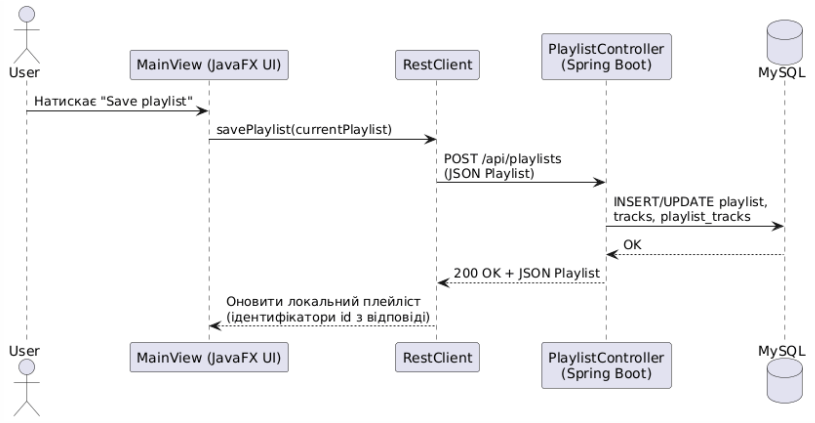

Рисунок 2 – Сценарій №5

1.5 Концептуальна модель системи

Концептуальна модель системи відображає основні сутності предметної області музичного програвача та зв’язки між ними. Вона описує логічну структуру даних, які будуть використовуватися

системою і є основою для побудови структури бази даних та програмної архітектури.

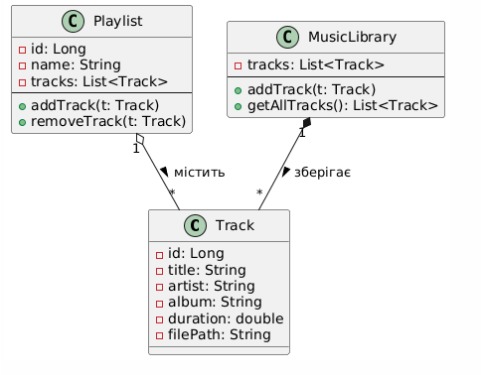

Рисунок 3 – Діаграма класів предметної області

Що представляє з себе ця діаграма:

1. **Track:**

   Містить інформацію про окремий аудіофайл: назву, виконавця, альбом та шлях до файлу.

2. **Playlist:**

   Буде представлений як набір треків, якого можна створювати, редагувати та зберігати. Плейлист має власний ідентифікатор, назву та дату створення. Зв’язок Playlist — Track є відношенням 1 до багатьох.

3. **MusicLibrary:**

   Представляє колекцію всіх доступних треків у клієнтській частині. Зберігає список треків, завантажених у застосунок через імпорт файлів.

1.6 Вибір бази даних

Для того, щоб наш програвач міг зберігати треки та плейлісти, потрібна надійна база даних. Для цієї роботи було обрано БД MySQL. Це перевірена часом реляційна СУБД з відкритим кодом, яка є своєрідним стандартом у розробці.

Чому саме MySQL:

1. **Популярність та екосистема:** Це одна з найпоширеніших баз даних у світі. Її підтримують майже всі сучасні фреймворки та хостинг-провайдери, тому проблем з інтеграцією точно не виникне.
2. **Надійність даних:** MySQL підтримує транзакції. Це означає, що коли користувач редагує плейліст або додає нові треки, ми можемо бути впевнені: дані збережуться коректно і нічого не зламається під час процесу.
3. **Оптимальна продуктивність:** де навантаження помірне, MySQL працює дуже швидко і не вимагає складних налаштувань зсередини.
4. **Зручність роботи:** Наявність таких інструментів як MySQL Workbench, значно спрощує життя. Там зручно проєктувати схеми, писати запити та робити бекапи.

Альтернативи: Звісно, можна було б обрати PostgreSQL (теж потужна система) або SQLite. Проте SQLite — це вбудоване рішення, яке чудово підходить для локальних програм, але погано вписується в нашу клієнт-серверну архітектуру. PostgreSQL — чудовий варіант, але для задач цього проєкту можливостей MySQL цілком достатньо, до того ж вона часто є більш звичним інструментом для старту.

1.7 Вибір мови програмування та середовища розробки

У межах цієїє роботи для розроблення клієнтської та серверної частин системи обрано мову програмування **Java**. Java є однією з

найбільш поширених універсальних мов, що забезпечує платформну незалежність, об’єктно-орієнтований підхід, розвинуту екосистему бібліотек та стабільну підтримку у сфері створення десктопних і серверних застосунків.

Чого Java:

1. **Працює всюди:** Завдяки віртуальній машині (JVM) не потрібно хвилюватися про операційну систему користувача. Написаний код однаково добре запуститься і на Windows, і на Linux, і на macOS.
2. **Потужна екосистема:** Для Java існує безліч готових бібліотек. Чи потрібно мені працювати з мережею, підключатися до бази даних чи обробляти мультимедіа — для всього вже є перевірені інструменти, що значно пришвидшує роботу.
3. **Надійність:** Java бере на себе складні питання керування пам’яттю та має сувору типізацію. Це допомагає уникнути багатьох помилок ще на етапі написання коду і робить додаток стабільним.

Інтерфейс: JavaFX:

Для створення клієнтської частини (того, шо бачить користувач) я зупинився на **JavaFX**. Це сучасний фреймворк, який ідеально підходить для мого завдання, і ось чому:

- **Вбудований медіаплеєр:** У JavaFX вже є компонент MediaPlayer. Це означає, що мені не потрібно писати обробку звуку з нуля — тобто фреймворк вміє відтворювати популярні формати.
- **Гнучкий дизайн:** Інтерфейс можна стилізувати за допомогою CSS, майже як вебсайт. Це дозволяє легко змінювати вигляд кнопок та панелей.
- **Правильна архітектура:** JavaFX підштовхує до використання патернів MVC/MVVM.
- **Зручне розгортання:** Є інструменти, що дозволяють запакувати готовий проєкт у звичайний .exe файл для Windows.

Серверна частина: Spring Boot:

Для створення серверного API було обрано **Spring Boot**. На сьогодні це фактичний стандарт у світі Java для вебзастосунків. Він дозволяє дуже швидко розгорнути REST-сервіс, маючи мінімум конфігурацій.

Середовище розробки

Писати код заплановано у **IntelliJ IDEA**. Хоча Java підтримують багато редакторів, саме IDEA є найбільш сильною: вона найкраще підказує код, допомагає миттєво знаходити помилки та має чудовий вбудований дебагер.

1.8 Проєктування розгортання системи

Проєктована система має клієнт-серверну архітектуру та складається з трьох основних компонентів:

1. **Клієнтський застосунок**, реалізований у середовищі IntelliJ IDEA за допомогою Java та JavaFX. Він запускається як настільна .exe програма й забезпечує інтерфейс взаємодії з користувачем.
2. **Серверна частина**, створена на основі Spring Boot, яка надає REST API для роботи з плейлистами та інформацією про треки.
3. **База даних MySQL**, що використовується сервером для збереження структурованих даних.

Розгортання системи передбачає, що клієнтський застосунок працює локально на комп’ютері користувача, у той час як серверна частина може бути розгорнута:

- або локально на тому ж комп’ютері,
- або на окремому сервері в локальній мережі чи хмарному середовищі.

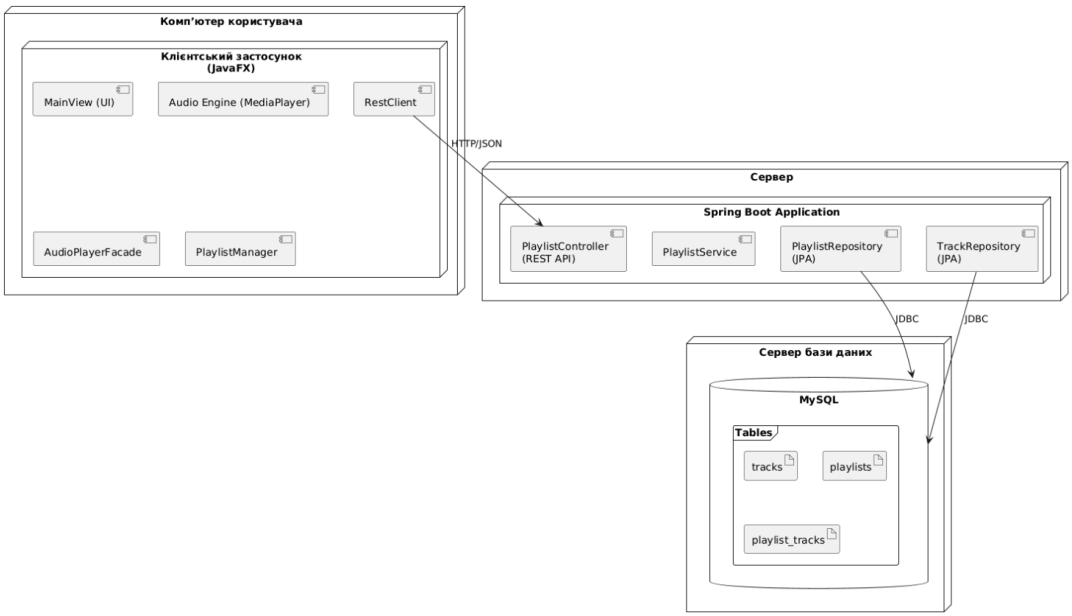

Рисунок 4 - Діаграма розгортання системи

Опис діаграми:

1. Комп’ютер користувача

На стороні користувача виконується клієнтський застосунок, реалізований на JavaFX.
Він складається з кількох основних компонентів:

- **MainView (UI)** — графічний інтерфейс користувача. Забезпечує взаємодію з плейлистами, треками, керування відтворенням, пошук, імпорт аудіофайлів.
- **Audio Engine (MediaPlayer)** — вбудований аудіодвигун JavaFX, що відповідає за відтворення аудіофайлів, перемотку, зміну гучності, перемикання треків.
- **AudioPlayerFacade** — фасад, який інкапсулює логіку керування відтворенням (play/pause/next/previous, shuffle, repeat).
- **PlaylistManager** — локальний менеджер плейлистів; відповідає за роботу з плейлистами на боці клієнта (вибір поточного плейлиста, додавання або видалення треків).
- **RestClient** — компонент, який виконує HTTP/JSON-запити до серверної частини системи.

Використовується для збереження та завантаження плейлистів, отримання списку треків із сервера.

2. Сервер

На сервері розгорнутий Spring Boot застосунок, який реалізує REST API та бізнес-логіку.

Він складається з таких модулів:

- **PlaylistController (REST API)** — обробляє HTTP-запити клієнтського застосунку, виконує маршрутизацію запитів до сервісного рівня, повертає JSON-відповіді.

- **PlaylistService** — реалізує бізнес-логіку роботи з плейлистами: створення, оновлення, видалення, додавання треків та робота зі зв’язками між ними.
- **PlaylistRepository (JPA)** — доступ до таблиці playlists за допомогою ORM-механізму Spring Data JPA.
- **TrackRepository (JPA)** — доступ до таблиці tracks, використовується при збереженні нових треків та формуванні складених структур плейлистів.
- **Data Access Layer (JPA)** — шар доступу до даних, який автоматично формує SQL-запити і взаємодіє з MySQL через JDBC.

3. Сервер бази даних

Це окремий вузол, на якому працює MySQL. У базі даних зберігається усі структуровані дані системи:

- **tracks** — таблиця з інформацією про треки (назва, артист, шлях до файлу).
- **playlists** — таблиця плейлистів користувача.
- **playlist_tracks** — проміжна таблиця для зв’язку *“багато-до-багатьох”* між плейлистами й треками.

Комунікація між Spring Boot сервером і MySQL здійснюється через JDBC, який використовується фреймворком JPA для виконання SQL-операцій.

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура бази даних

Структура БД включає три основні таблиці:

- tracks — зберігає інформацію про кожен аудіофайл;

- playlists — зберігає дані про плейлисти користувача;

- playlist_tracks — проміжна таблиця для зв’язку «багато-до-багатьох» між треками й плейлистами.

Опис таблиць:

Поле	Тип	Опис
id	INT PK	Унікальний ідентифікатор треку
title	VARCHAR(255)	Назва композиції
artist	VARCHAR(255)	Виконавець
album	VARCHAR(255)	Альбом
file_path	VARCHAR(500)	Шлях до файлу на локальній системі

Таблиця 6 – tracks

Поле	Тип	Опис
id	INT PK AI	Унікальний ідентифікатор плейлиста
name	VARCHAR(255)	Назва плейлиста
created_at	DATETIME	Дата створення

Таблиця 7 – playlists

Поле	Тип	Опис
playlist_id	INT FK	Посилання на `playlists.id`
track_id	INT FK	Посилання на `tracks.id`

Таблиця 8 – playlist_tracks

Зв’язки між таблицями:

- Один плейлист може містити багато треків.

- Один трек може входити до багатьох плейлистів.

- Реалізовано через таблицю playlist_tracks (many-to-many).

2.2. Архітектура системи

Архітектура розроблюваної системи ґрунтується на клієнт–серверній моделі з двома основними рівнями:

1. **Клієнтський рівень** (JavaFX-застосунок) — відповідає за взаємодію з користувачем, відтворення аудіо, керування плейлістами та виклик REST-запитів

2. **Серверний рівень** (Spring Boot) — надає REST API для роботи з треками та плейлістами і взаємодіє з базою даних MySQL.

У межах клієнтської частини виділяються компоненти для:

- користувацького інтерфейсу;
- логіки роботи з плейлистами та бібліотекою;
- логіки відтворення
- клієнта для HTTP-запитів до серверної частини.

Серверна частина структурована за принципом “controller–service–repository”.

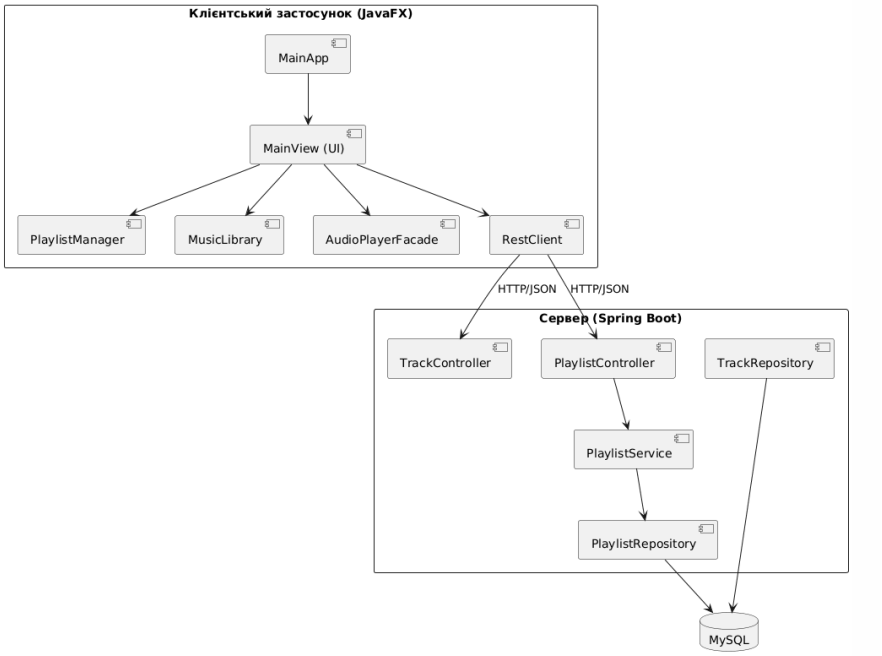

Рисунок 5 – Діаграма архітектури взаємодії

Пояснення:

- MainApp відповідає за запуск застосунку
- PlaylistManager та MusicLibrary керують локальною колекцією треків і плейлістів.
- AudioPlayerFacade інкапсулює роботу з MediaPlayer - патерн *Facade*
- RestClient відправляє HTTP-запити до серверних контролерів.
- На сервері контролери приймають запити, сервіси, містять бізнес-логіку, а репозиторії працюють із БД.

2.2.1. Специфікація системи

Клієнтська частина:

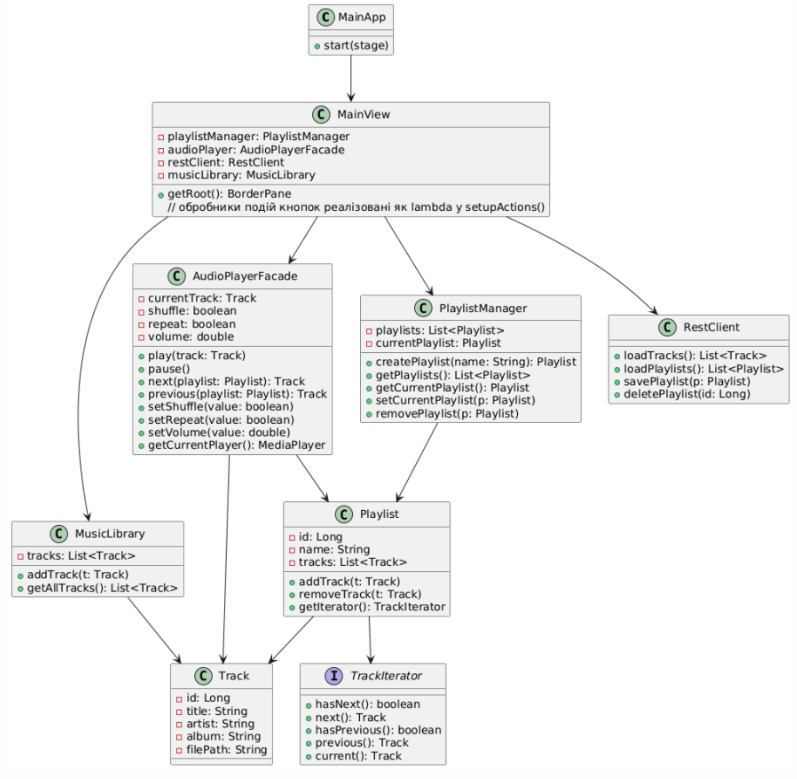

Рисунок 6 – Діаграма класів клієнтської частини.

Серверна частина:

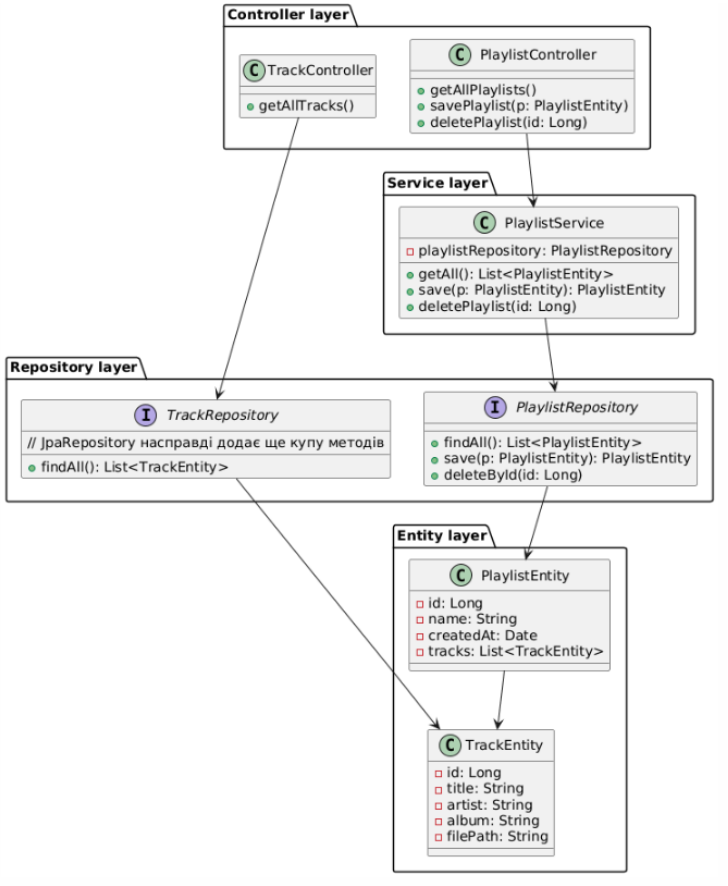

Рисунок 7 – Діаграма класів серверної частини

2.2.2. Вибір та обґрунтування патернів реалізації

1. Спрощення керування відтворенням аудіо (патерн Facade)
Проблема: безпосередня робота користувацького інтерфейсу з MediaPlayer призводила б до дублювання коду та змішування логіки UI й аудіодвигуна. Рішення: введено фасад AudioPlayerFacade, який інкапсулює усю логіку відтворення: створення й конфігурацію MediaPlayer, відтворення/паузу треку, перемикання на наступний/попередній трек, циклічний перехід по плейлисти, режими shuffle та repeat. Використання: клас MainView звертається до фасаду через методи play(), pause(), next(), previous(), setShuffle(), setRepeat(), не працюючи напряму з JavaFX Media API.

2. Обхід треків у плейлісті (патерн Iterator)
Проблема: логіка переходу між треками не повинна залежати від внутрішньої структури плейлиста.
Рішення: реалізовано інтерфейс TrackIterator та клас PlaylistTrackIterator, які інкапсулюють послідовний обхід списку треків. Клас Playlist надає метод getIterator(), що повертає новий екземпляр ітератора.
Використання: у методах next(playlist) та previous(playlist) класу AudioPlayerFacade використовується TrackIterator для пошуку наступного або попереднього треку відносно поточного, що дозволяє змінювати внутрішню реалізацію обходу без зміни коду фасаду й UI.

2.3. Інструкція користувача

Даний розділ описує порядок встановлення, запуску та використання клієнтського застосунку, а також правила взаємодії з основними елементами інтерфейсу.

Після ініціалізації пакувальника *jpackage* (або ж встановити за посиланням на репозиторії GitHub вже скомпільований архів-застосунок) користувач отримує папку застосунку в *client-side* директорії *(target\dist\MusicPlayer\),* що містить файли:

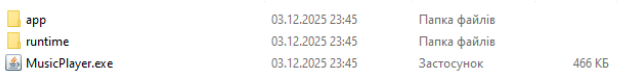

Рисунок 8 – Застосунок

Для запуску програвача необхідно відкрити exe-файл.

Після запуску відкривається головне вікно програми:

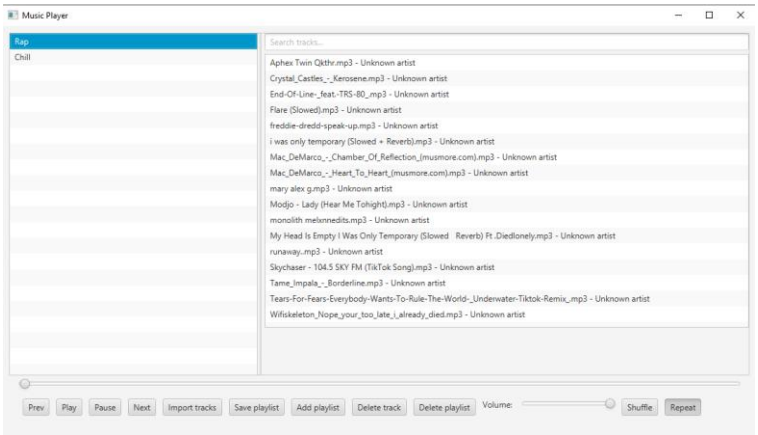

Рисунок 9 – Вікно програми

Інтерфейс користувача

Головне вікно складається з таких елементів:

1. Панель плейлістів

- список наявних плейлістів;
- кнопки:
  - **Add Playlist** — створення нового плейліста;
  - **Delete Playlist** — видалення вибраного плейліста.

2. Список треків (праворуч)

- відображає треки, що містяться у вибраному плейлісті;
- кнопка **Delete Track** видаляє трек зі списку.

3. Панель керування (внизу)

Містить кнопки:

- **Play** — запуск відтворення;

- **Pause** — пауза;
- **Next** — наступний трек;
- **Prev** — попередній трек.

Додаткові елементи:

- **Volume Slider** — зміна гучності;
- **Shuffle** — перемішане відтворення;
- **Repeat** — повтор поточного треку;
- **Position Slider** — прогрес відтворення.

3. Робота з плейлістами

3.1 Створення нового плейліста

1. Натисніть кнопку **Add Playlist**.
2. У діалоговому вікні введіть назву плейліста.
3. Підтвердіть створення.

Новий плейліст з’явиться у списку зліва.

3.2 Збереження плейліста

Щоб зберегти поточний плейліст на сервері:

1. Виберіть потрібний плейлист у списку.
2. Натисніть кнопку **Save Playlist**.

Плейліст буде відправлено на сервер, а його зміст — збережено в базі даних.

3.3 Видалення плейліста

1. Оберіть плейліст у списку.
2. Натисніть **Delete Playlist**.

Плейліст буде видалений як локально, так і на сервері.

Якщо плейліст містив треки, сервер також видалить зв’язки у таблиці playlist_tracks.

4. Додавання та видалення треків

4.1 Імпорт треків

1. Натисніть кнопку **Import Tracks**.
2. У провіднику оберіть один або кілька аудіофайлів.
3. Натисніть **Open**.

Треки будуть додані до списку треків поточного плейліста.

4.2 Видалення треку

1. Оберіть трек у списку.
2. Натисніть кнопку **Delete Track**.

Трек буде видалено лише з плейліста, але не з файлової системи.

5. Керування відтворенням

5.1 Відтворення треку

1. Виберіть трек у списку.
2. Натисніть **Play**.

На позиційному слайдері з’явиться прогрес відтворення.

5.2 Пауза та відновлення

- **Pause** — зупиняє відтворення, але не скидає позицію.
- Повторне натискання **Play** відновить відтворення з того ж місця.

5.3 Перемикання треків

- **Next** — переходить до наступного треку.
- **Prev** — повертається до попереднього.

Система підтримує циклічне перемикання:

- якщо вибрано останній трек → перехід на перший;
- якщо натиснути Prev на першому → перехід на останній.

5.4 Shuffle / Repeat

- **Shuffle** — вибір випадкового треку.
- **Repeat** — повторює поточний трек після завершення.

5.5 Регулювання гучності

Слайдер **Volume** змінює рівень відтворення від 0% до 100%.

6. Автоматичні переходи

Після завершення треку:

- якщо Shuffle увімкнено → вибирається випадковий трек;
- якщо Repeat увімкнено → той самий трек починає відтворюватися заново;
- якщо всі режими вимкнені → запускається наступний трек у списку.

Підготовка бази даних MySQL

Серверна частина використовує СУБД **MySQL**, у якій зберігаються треки, плейлісти та зв’язки між ними.

Перед першим запуском необхідно створити порожню базу даних.

Створення бази даних

1. Під’єднайтесь до локального MySQL-сервера.
2. Відкрийте консоль SQL.
3. Виконайте команду:

```
CREATE DATABASE music_player_db
  CHARACTER SET utf8mb4
```

```
  COLLATE utf8mb4_unicode_ci;
```

Після цього з’явиться нова пуста база `music_player_db`.

Примітка: таблиці для проєкта створюються автоматично під час запуску Spring Boot, завдяки Hibernate (`spring.jpa.hibernate.ddl-auto=update`).

Налаштування параметрів доступу

Параметри підключення до MySQL прописані у файлі:

```
src/main/resources/application.properties
```

Основні настройки:

```
spring.datasource.url=jdbc:mysql://localhost:3306/music_player_db
spring.datasource.username=YOUR_USERNAME
spring.datasource.password=YOUR_PASSWORD
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Необхідно змінити:

- `username`
- `password`
- номер порту (3306 за замовчуванням, можна не міняти)

Запуск серверної частини

Після того як проєкт зібраний командою:

```
mvn clean package
```

папка `target/` містить виконуваний JAR-файл сервера, скоріше за все з такою назвою:

```
music-player-server-1.0.0.jar
```

Разом з проєктом постачається файл:

```
run-server.bat
```

який автоматично запускає сервер з jar-файла, про якого розказувалося вище.

Запуск сервера

1. Запустіть файл:

```
run-server.bat
```

2. Відкриється консоль із логами Spring Boot.

Після запуску сервер чекає запитів клієнта за адресою:

```
http://localhost:8080/api
```

Важливо: консоль сервера повинна залишатися відкритою, поки користувач працює з клієнтським застосунком.

Після цього можна працювати з програмою.

ВИСНОВКИ

За виконанням цієї роботи було спроєктовано та реалізовано повноцінний клієнт-серверний музичний програвач, що поєднує сучасні програмні технології, інструменти та архітектурні підходи. У процесі розробки було виконано детальне проєктування системи, що включало опис предметної області, аналіз існуючих рішень, формування функціональних і нефункціональних вимог, моделювання сценаріїв використання та побудову необхідних UML-діаграм. Розроблені діаграми — варіантів використання, класів, розгортання, послідовностей та станів — дозволили сформувати цілісне бачення архітектури застосунку та структури взаємодії між його компонентами.

У результаті реалізації було створено два взаємопов’язані програмні модулі: **Клієнтський застосунок на JavaFX**, що забезпечує зручний графічний інтерфейс, імпорт треків, відтворення музики, роботу з плейлістами, керування відтворенням (Play, Pause, Next, Prev), а також режими Shuffle та Repeat; **Серверна частина на Spring Boot**, що реалізує REST API, керує плейлістами та треками та забезпечує їх збереження у базі даних MySQL.

Під час розробки були застосовані шаблони проєктування, що підвищили структурованість та гнучкість коду: **Facade** для спрощення роботи з JavaFX MediaPlayer, **Iterator** для проходження по треках у плейлісті, а також елементи патерну **Command** при обробці команд керування програвачем.

У кінцевому підсумку було отримано стабільний, працездатний застосунок, який має логічну архітектуру, коректно взаємодіє з сервером та базою даних, а також може бути упакований у виконуваний файл (.exe) для зручного запуску.

Результати роботи демонструють розуміння принципів побудови клієнт-серверних систем, практичне застосування JavaFX, Spring Boot та

MySQL, а також вміння використовувати UML-моделювання та шаблони проєктування для створення сучасних програмних рішень.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. **Oracle.** Java Platform, Standard Edition Documentation [Електронний ресурс].
   Режим доступу: https://docs.oracle.com/java/
2. **OpenJFX.** JavaFX Documentation [Електронний ресурс].
   Режим доступу: https://openjfx.io/
3. **OpenJFX.** JavaFX Media API — MediaPlayer Class [Електронний ресурс].
   Режим доступу: https://openjfx.io/javadoc/25/
4. **Spring Framework.** Spring Boot Reference Documentation [Електронний ресурс].
   Режим доступу: https://docs.spring.io/spring-boot/
5. **Spring Data.** Spring Data JPA — Reference Documentation [Електронний ресурс].
   Режим доступу: https://spring.io/projects/spring-data-jpa
6. **Oracle Corporation.** MySQL 8.0 Reference Manual [Електронний ресурс].
   Режим доступу: https://dev.mysql.com/doc/
7. **PlantUML.** User Guide and Syntax Reference [Електронний ресурс].
   Режим доступу: https://plantuml.com/
8. **Oracle.** jpackage Tool — Documentation [Електронний ресурс].
   Режим доступу: https://docs.oracle.com/en/java/javase/21/jpackage/

ДОДАТКИ

DDL код бази даних:

```
CREATE TABLE tracks (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255),
    artist VARCHAR(255),
    album VARCHAR(255),
    file_path VARCHAR(500)
);

CREATE TABLE playlists (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255),
    created_at DATETIME
);

CREATE TABLE playlist_tracks (
    playlist_id INT,
    track_id INT,
    PRIMARY KEY (playlist_id, track_id),
    FOREIGN KEY (playlist_id) REFERENCES playlists(id) ON DELETE
CASCADE,
    FOREIGN KEY (track_id) REFERENCES tracks(id) ON DELETE CASCADE
);
```

Код. Серверна частина:

```
package com.example.musicserver.controller;

import com.example.musicserver.entity.PlaylistEntity;
import com.example.musicserver.service.PlaylistService;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/playlists")
public class PlaylistController {

    private final PlaylistService playlistService;
```

```
    public PlaylistController(PlaylistService playlistService) {
        this.playlistService = playlistService;
    }

    @GetMapping
    public List<PlaylistEntity> getAll() {
        return playlistService.getAll();
    }

    @PostMapping
    public PlaylistEntity save(@RequestBody PlaylistEntity playlist) {
        return playlistService.save(playlist);
    }

    @DeleteMapping("/{id}")
    public void deletePlaylist(@PathVariable("id") Long id) {
        playlistService.deletePlaylist(id);
    }
}
```

```
package com.example.musicserver.controller;

import com.example.musicserver.entity.TrackEntity;
import com.example.musicserver.service.TrackService;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/tracks")
@CrossOrigin(origins = "*")
public class TrackController {

    private final TrackService trackService;

    public TrackController(TrackService trackService) {
        this.trackService = trackService;
    }

    @GetMapping
    public List<TrackEntity> getAllTracks() {
        return trackService.findAll();
    }

    @PostMapping
    public TrackEntity createTrack(@RequestBody TrackEntity track) {
        return trackService.save(track);
    }
}
```

```
package com.example.musicserver.entity;

import jakarta.persistence.*;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
```

```java
@Entity
@Table(name = "playlists")
public class PlaylistEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
            name = "playlist_tracks",
            joinColumns = @JoinColumn(name = "playlist_id"),
            inverseJoinColumns = @JoinColumn(name = "track_id")
    )
    private List<TrackEntity> tracks = new ArrayList<>();


    public PlaylistEntity() {
    }

    public PlaylistEntity(String name) {
        this.name = name;
        this.createdAt = LocalDateTime.now();
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public LocalDateTime getCreatedAt() {
        return createdAt;
    }

    public List<TrackEntity> getTracks() {
        return tracks;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setCreatedAt(LocalDateTime createdAt) {
        this.createdAt = createdAt;
    }

    public void setTracks(List<TrackEntity> tracks) {
        this.tracks = tracks;
    }
```

```
    public void addTrack(TrackEntity track) {
        this.tracks.add(track);
    }

    public void removeTrack(TrackEntity track) {
        this.tracks.remove(track);
    }
}
```

```
package com.example.musicserver.entity;

import jakarta.persistence.*;

@Entity
@Table(name = "tracks")
public class TrackEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    private String artist;

    private String album;


    @Column(name = "file_path")
    private String filePath;

    public TrackEntity() {
    }

    public TrackEntity(String title, String artist, String album, String
filePath) {
        this.title = title;
        this.artist = artist;
        this.album = album;
        this.filePath = filePath;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getArtist() {
```

```
        return artist;
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public String getAlbum() {
        return album;
    }

    public void setAlbum(String album) {
        this.album = album;
    }

    public String getFilePath() {
        return filePath;
    }

    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }
}
```

```
package com.example.musicserver.repository;

import com.example.musicserver.entity.PlaylistEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface PlaylistRepository extends JpaRepository<PlaylistEntity, Long>
{
}
```

```
package com.example.musicserver.repository;

import com.example.musicserver.entity.TrackEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface TrackRepository extends JpaRepository<TrackEntity, Long> {
}
```

```
package com.example.musicserver.service;

import com.example.musicserver.entity.PlaylistEntity;
import com.example.musicserver.repository.PlaylistRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
public class PlaylistService {
```

```
    private final PlaylistRepository playlistRepository;

    public PlaylistService(PlaylistRepository playlistRepository) {
        this.playlistRepository = playlistRepository;
    }

    public List<PlaylistEntity> getAll() {
        return playlistRepository.findAll();
    }

    public PlaylistEntity save(PlaylistEntity playlist) {
        return playlistRepository.save(playlist);
    }


    @Transactional
    public void deletePlaylist(Long id) {
        PlaylistEntity playlist = playlistRepository.findById(id)
                .orElseThrow(() -> new IllegalArgumentException("Playlist not
found: " + id));

        // очищаємо зв'язки, щоб видалити join rows
        playlist.getTracks().clear();

        playlistRepository.delete(playlist);
    }
}
```

```
package com.example.musicserver.service;

import com.example.musicserver.entity.TrackEntity;
import com.example.musicserver.repository.TrackRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class TrackService {

    private final TrackRepository trackRepository;

    public TrackService(TrackRepository trackRepository) {
        this.trackRepository = trackRepository;
    }

    public List<TrackEntity> findAll() {
        return trackRepository.findAll();
    }

    public TrackEntity save(TrackEntity track) {
        return trackRepository.save(track);
    }
}
```

```
package com.example.musicserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MusicPlayerServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(MusicPlayerServerApplication.class, args);
    }
}
```

application.properties:

```
spring.application.name=music-player-server
spring.datasource.url=jdbc:mysql://localhost:3306/music_player_db?useSSL=false&s
erverTimezone=UTC
spring.datasource.username=
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

server.port=8080
```

Код. Клієнтська частина:

```
package com.example.musicclient;

import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;

import java.io.File;
import java.util.Random;
import javafx.util.Duration;


public class AudioPlayerFacade {

    private MediaPlayer currentPlayer;
    private Track currentTrack;
    private boolean shuffle = false;
    private boolean repeat = true;

    private double volume = 1.0;

    private final Random random = new Random();

    public void play(Track track) {
        try {
            if (track == null) return;

            if (currentPlayer != null) {
                currentPlayer.stop();
```

```
            }

            this.currentTrack = track;
            Media media = new Media(new
File(track.getFilePath()).toURI().toString());
            currentPlayer = new MediaPlayer(media);
            currentPlayer.setVolume(volume);
            currentPlayer.play();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public MediaPlayer getCurrentPlayer() {
        return currentPlayer;
    }

    public Duration getCurrentTime() {
        if (currentPlayer != null) {
            return currentPlayer.getCurrentTime();
        }
        return Duration.ZERO;
    }

    public Duration getTotalDuration() {
        if (currentPlayer != null) {
            return currentPlayer.getTotalDuration();
        }
        return Duration.ZERO;
    }

    public void seek(Duration position) {
        if (currentPlayer != null && position != null) {
            currentPlayer.seek(position);
        }
    }

    public void setVolume(double volume) {
        this.volume = volume;
        if (currentPlayer != null) {
            currentPlayer.setVolume(volume);
        }
    }

    public double getVolume() {
        return volume;
    }

    public void pause() {
        if (currentPlayer != null) {
            var status = currentPlayer.getStatus();
            if (status == MediaPlayer.Status.PLAYING) {
                // якщо грає — ставимо на паузу
                currentPlayer.pause();
            } else if (status == MediaPlayer.Status.PAUSED) {
                currentPlayer.play();
            }
        }
    }
```

```
    public void resume() {
        if (currentPlayer != null) {
            currentPlayer.play();
        }
    }

    public void stop() {
        if (currentPlayer != null) {
            currentPlayer.stop();
        }
    }

    public void setShuffle(boolean shuffle) {
        this.shuffle = shuffle;
    }

    public boolean isShuffle() {
        return shuffle;
    }

    public void setRepeat(boolean repeat) {
        this.repeat = repeat;
    }

    public boolean isRepeat() {
        return repeat;
    }

    public Track getCurrentTrack() {
        return currentTrack;
    }

    public Track next(Playlist playlist) {
        if (playlist == null || playlist.getTracks().isEmpty()) {
            return null;
        }

        var tracks = playlist.getTracks();

        if (shuffle && tracks.size() > 1) {
            int currentIndex = currentTrack != null ?
tracks.indexOf(currentTrack) : -1;
            int nextIndex;
            do {
                nextIndex = random.nextInt(tracks.size());
            } while (nextIndex == currentIndex);
            Track nextTrack = tracks.get(nextIndex);
            play(nextTrack);
            return nextTrack;
        }

        TrackIterator iterator = playlist.getIterator();

        if (currentTrack == null) {
            if (iterator.hasNext()) {
                Track first = iterator.next();
                play(first);
                return first;
            }
            return null;
        }
```

```
        while (iterator.hasNext()) {
            Track t = iterator.next();
            if (t == currentTrack) {
                break;
            }
        }

        if (!iterator.hasNext()) {
            TrackIterator restartIterator = playlist.getIterator();
            if (restartIterator.hasNext()) {
                Track first = restartIterator.next();
                play(first);
                return first;
            }
            return null;
        }

        Track nextTrack = iterator.next();
        play(nextTrack);
        return nextTrack;
    }



    public Track previous(Playlist playlist) {
        if (playlist == null || playlist.getTracks().isEmpty()) {
            return null;
        }

        var tracks = playlist.getTracks();

        if (shuffle && tracks.size() > 1) {
            int currentIndex = currentTrack != null ?
tracks.indexOf(currentTrack) : -1;
            int prevIndex;
            do {
                prevIndex = random.nextInt(tracks.size());
            } while (prevIndex == currentIndex);
            Track prevTrack = tracks.get(prevIndex);
            play(prevTrack);
            return prevTrack;
        }

        TrackIterator iterator = playlist.getIterator();

        if (currentTrack == null) {
            Track last = null;
            while (iterator.hasNext()) {
                last = iterator.next();
            }
            if (last != null) {
                play(last);
            }
            return last;
        }

        while (iterator.hasNext()) {
            Track t = iterator.next();
            if (t == currentTrack) {
                break;
            }
        }
```

```
        if (iterator.hasPrevious()) {
            Track prevTrack = iterator.previous();
            play(prevTrack);
            return prevTrack;
        } else {
            Track last = null;
            while (iterator.hasNext()) {
                last = iterator.next();
            }
            if (last != null) {
                play(last);
            }
            return last;
        }
    }


}
```

```
package com.example.musicclient;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MainApp extends Application {

    @Override
    public void start(Stage primaryStage) {
        MainView mainView = new MainView();

        Scene scene = new Scene(mainView.getRoot(), 900, 600);
        primaryStage.setTitle("Music Player");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

```
package com.example.musicclient;

import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.scene.control.*;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.stage.FileChooser;
import javafx.scene.control.TextInputDialog;
import javafx.scene.control.Slider;
import javafx.util.Duration;
import java.io.File;
```

```
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.media.MediaPlayer;

public class MainView {

    private final BorderPane root;

    private final ListView<Track> trackListView;
    private final ListView<Playlist> playlistListView;

    private final Button playButton;
    private final Button pauseButton;
    private final Button nextButton;
    private final Button prevButton;
    private final Button importButton;

    private final Slider positionSlider;
    private final RestClient restClient;
    private final AudioPlayerFacade audioPlayer;
    private final MusicLibrary musicLibrary;
    private final PlaylistManager playlistManager;
    private final Button savePlaylistButton;

    private final Slider volumeSlider;
    private final ToggleButton shuffleButton;
    private final ToggleButton repeatButton;

    private final Button deletePlaylistButton;
    private final Button deleteTrackButton;
    private final Button addPlaylistButton;

    private final TextField searchField;


    public MainView() {
        this.root = new BorderPane();

        this.audioPlayer = new AudioPlayerFacade();
        this.musicLibrary = new MusicLibrary();
        this.playlistManager = new PlaylistManager();
        this.restClient = new RestClient("http://localhost:8080/api");

        this.trackListView = new ListView<>();
        this.playlistListView = new ListView<>();

        this.playButton = new Button("Play");
        this.pauseButton = new Button("Pause");
        this.nextButton = new Button("Next");
        this.prevButton = new Button("Prev");
        this.importButton = new Button("Import tracks");
        this.savePlaylistButton = new Button("Save playlist");

        this.volumeSlider = new Slider(0, 1, 1);
        this.shuffleButton = new ToggleButton("Shuffle");
        this.repeatButton = new ToggleButton("Repeat");

        this.deletePlaylistButton = new Button("Delete playlist");
        this.deleteTrackButton = new Button("Delete track");
        this.addPlaylistButton = new Button("Add playlist");

        this.positionSlider = new Slider(0, 100, 0);
```

```
        this.searchField = new TextField();
        this.searchField.setPromptText("Search tracks...");


        setupLayout();
        setupActions();
        loadDataFromServer();
    }

    public BorderPane getRoot() {
        return root;
    }

    private void setupLayout() {
        SplitPane centerPane = new SplitPane();
        centerPane.setOrientation(Orientation.HORIZONTAL);

        playlistListView.setPrefWidth(200);
        playlistListView.setPlaceholder(new Label("No playlists"));

        trackListView.setPlaceholder(new Label("No tracks"));

        VBox tracksBox = new VBox(5, searchField, trackListView);
        tracksBox.setPadding(new Insets(0, 0, 0, 5));

        centerPane.getItems().addAll(playlistListView, tracksBox);
        centerPane.setDividerPositions(0.3);

        HBox controls = new HBox(10,
                prevButton, playButton, pauseButton, nextButton,
                importButton, savePlaylistButton,
                addPlaylistButton, deleteTrackButton, deletePlaylistButton,
                new Label("Volume:"), volumeSlider,
                shuffleButton, repeatButton
        );
        controls.setPadding(new Insets(10));

        VBox bottomBox = new VBox(5, positionSlider, controls);
        bottomBox.setPadding(new Insets(5, 10, 10, 10));

        root.setCenter(centerPane);
        root.setBottom(bottomBox);
        root.setPadding(new Insets(10));

    }

    private void setupActions() {
        playButton.setOnAction(e -> {
            Track selected =
trackListView.getSelectionModel().getSelectedItem();
            if (selected != null) {
                audioPlayer.play(selected);
                configurePositionSliderForCurrentTrack();
            }
        });

        searchField.textProperty().addListener((obs, oldText, newText) -> {
            applyTrackFilter(newText);
        });

        pauseButton.setOnAction(e -> audioPlayer.pause());
```

```
        nextButton.setOnAction(e -> {
            Playlist current = playlistManager.getCurrentPlaylist();
            Track next = audioPlayer.next(current);
            if (next != null) {
                trackListView.getSelectionModel().select(next);
                configurePositionSliderForCurrentTrack();
            }
        });

        prevButton.setOnAction(e -> {
            Playlist current = playlistManager.getCurrentPlaylist();
            Track prev = audioPlayer.previous(current);
            if (prev != null) {
                trackListView.getSelectionModel().select(prev);
                configurePositionSliderForCurrentTrack();
            }
        });

        positionSlider.setOnMouseReleased(e -> {
            MediaPlayer player = audioPlayer.getCurrentPlayer();
            if (player != null) {
                double seconds = positionSlider.getValue();
                player.seek(Duration.seconds(seconds));
            }
        });

        importButton.setOnAction(e -> importTracks());

        savePlaylistButton.setOnAction(e -> {
            Playlist current = playlistManager.getCurrentPlaylist();
            if (current != null) {
                restClient.savePlaylist(current);
            }
        });


playlistListView.getSelectionModel().selectedItemProperty().addListener((obs,
oldVal, newVal) -> {
            if (newVal != null) {
                playlistManager.setCurrentPlaylist(newVal);
                searchField.clear();
                trackListView.getItems().setAll(newVal.getTracks());
            }
        });


        volumeSlider.valueProperty().addListener((obs, oldVal, newVal) -> {
            audioPlayer.setVolume(newVal.doubleValue());
        });

        shuffleButton.selectedProperty().addListener((obs, wasSelected,
isSelected) -> {
            audioPlayer.setShuffle(isSelected);
        });

        repeatButton.selectedProperty().addListener((obs, wasSelected,
isSelected) -> {
            audioPlayer.setRepeat(isSelected);
        });

        deletePlaylistButton.setOnAction(e -> {
```

```
            Playlist selected =
playlistListView.getSelectionModel().getSelectedItem();
            if (selected == null) {
                return;
            }

            restClient.deletePlaylist(selected.getId());

            playlistManager.removePlaylist(selected);
            playlistListView.getItems().remove(selected);

            Playlist current = playlistManager.getCurrentPlaylist();
            if (current != null) {
                trackListView.getItems().setAll(current.getTracks());
                playlistListView.getSelectionModel().select(current);
            } else {
                trackListView.getItems().clear();
            }
        });

        deleteTrackButton.setOnAction(e -> {
            Track selectedTrack =
trackListView.getSelectionModel().getSelectedItem();
            if (selectedTrack == null) {
                return;
            }

            Playlist current = playlistManager.getCurrentPlaylist();
            if (current != null) {
                current.getTracks().remove(selectedTrack);
            }

            trackListView.getItems().remove(selectedTrack);
        });

        addPlaylistButton.setOnAction(e -> {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setTitle("New playlist");
            dialog.setHeaderText("Create new playlist");
            dialog.setContentText("Playlist name:");

            dialog.showAndWait().ifPresent(name -> {
                String trimmed = name.trim();
                if (trimmed.isEmpty()) {
                    return;
                }

                boolean exists = playlistManager.getPlaylists().stream()
                        .anyMatch(p -> trimmed.equalsIgnoreCase(p.getName()));

                if (exists) {
                    System.err.println("Playlist with this name already
exists");
                    return;
                }

                Playlist newPlaylist = new Playlist(trimmed);
                playlistManager.getPlaylists().add(newPlaylist);
                playlistManager.setCurrentPlaylist(newPlaylist);

                playlistListView.getItems().add(newPlaylist);
                playlistListView.getSelectionModel().select(newPlaylist);
```

```
                trackListView.getItems().setAll(newPlaylist.getTracks());
            });
        });
    }

    private void importTracks() {
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Select audio files");
        fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Audio files", "*.mp3", "*.wav",
"*.aac")
        );
        var files =
fileChooser.showOpenMultipleDialog(root.getScene().getWindow());
        if (files != null) {
            for (File file : files) {
                Track track = new Track(
                        null,
                        file.getName(),
                        "Unknown artist",
                        "Unknown album",
                        file.getAbsolutePath()
                );
                musicLibrary.addTrack(track);
                trackListView.getItems().add(track);

                Playlist current = playlistManager.getCurrentPlaylist();
                if (current != null) {
                    current.addTrack(track);
                }
            }
        }
    }

    private void loadDataFromServer() {
        var tracksFromServer = restClient.loadTracks();
        for (Track t : tracksFromServer) {
            musicLibrary.addTrack(t);
        }
        trackListView.getItems().addAll(tracksFromServer);

        var playlistsFromServer = restClient.loadPlaylists();
        if (!playlistsFromServer.isEmpty()) {
            playlistListView.getItems().clear();
            playlistListView.getItems().addAll(playlistsFromServer);
            playlistManager.setCurrentPlaylist(playlistsFromServer.get(0));
            playlistListView.getSelectionModel().select(0);

trackListView.getItems().setAll(playlistsFromServer.get(0).getTracks());
        }
    }

    private void configurePositionSliderForCurrentTrack() {
        MediaPlayer player = audioPlayer.getCurrentPlayer();
        if (player == null) {
            positionSlider.setDisable(true);
            positionSlider.setValue(0);
            return;
        }

        positionSlider.setDisable(false);
```

```
        player.setOnReady(() -> {
            Duration total = player.getTotalDuration();
            positionSlider.setMax(total.toSeconds());
        });

        player.currentTimeProperty().addListener((obs, oldTime, newTime) -> {
            if (!positionSlider.isValueChanging()) {
                positionSlider.setValue(newTime.toSeconds());
            }
        });

        player.setOnEndOfMedia(() -> {
            if (repeatButton.isSelected()) {
                player.seek(Duration.ZERO);
                player.play();
                return;
            }

            Playlist current = playlistManager.getCurrentPlaylist();
            if (current != null) {
                Track next = audioPlayer.next(current);
                if (next != null) {
                    trackListView.getSelectionModel().select(next);
                    configurePositionSliderForCurrentTrack();
                }
            }
        });
    }


    private void applyTrackFilter(String query) {
        String trimmed = query == null ? "" : query.trim().toLowerCase();

        Playlist current = playlistManager.getCurrentPlaylist();
        if (current == null) {
            return;
        }

        if (trimmed.isEmpty()) {
            trackListView.getItems().setAll(current.getTracks());
            return;
        }

        var filtered = current.getTracks().stream()
                .filter(t -> {
                    String title = t.getTitle() != null ?
t.getTitle().toLowerCase() : "";
                    String artist = t.getArtist() != null ?
t.getArtist().toLowerCase() : "";
                    return title.contains(trimmed) || artist.contains(trimmed);
                })
                .toList();

        trackListView.getItems().setAll(filtered);
    }

}
```

```
package com.example.musicclient;

import java.util.ArrayList;
import java.util.List;

public class MusicLibrary {
    private final List<Track> tracks = new ArrayList<>();

    public void addTrack(Track track) {
        boolean exists = tracks.stream()
                .anyMatch(t -> t.getFilePath() != null
                        && t.getFilePath().equals(track.getFilePath()));
        if (!exists) {
            tracks.add(track);
        }
    }

    public List<Track> getAllTracks() {
        return tracks;
    }
}
```

```
package com.example.musicclient;

import java.util.ArrayList;
import java.util.List;
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;


@JsonIgnoreProperties(ignoreUnknown = true)
public class Playlist {
    private Long id;
    private String name;
    private final List<Track> tracks = new ArrayList<>();

    public Playlist() {
    }

    public Playlist(Long id, String name) {
        this.id = id;
        this.name = name;
    }

    public Playlist(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public List<Track> getTracks() {
        return tracks;
```

```
    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void addTrack(Track track) {
        // не додаємо трек, якщо вже є з таким самим шляхом до файлу
        boolean exists = tracks.stream()
                .anyMatch(t -> t.getFilePath() != null
                        && t.getFilePath().equals(track.getFilePath()));

        if (!exists) {
            tracks.add(track);
        }
    }

    @JsonIgnore
    public TrackIterator getIterator() {
        return new PlaylistTrackIterator(tracks);
    }


    public void removeTrack(Track track) {
        tracks.remove(track);
    }

    @Override
    public String toString() {
        return name;
    }
}
```

```
package com.example.musicclient;

import java.util.ArrayList;
import java.util.List;

public class PlaylistManager {
    private final List<Playlist> playlists = new ArrayList<>();
    private Playlist currentPlaylist;

    public Playlist createPlaylist(String name) {
        Playlist playlist = new Playlist(name);
        playlists.add(playlist);
        if (currentPlaylist == null) {
            currentPlaylist = playlist;
        }
        return playlist;
    }

    public List<Playlist> getPlaylists() {
        return playlists;
    }
```

```
    public Playlist getCurrentPlaylist() {
        return currentPlaylist;
    }

    public void setCurrentPlaylist(Playlist playlist) {
        this.currentPlaylist = playlist;
    }

    public void removePlaylist(Playlist playlist) {
        playlists.remove(playlist);
        if (currentPlaylist == playlist) {
            if (playlists.isEmpty()) {
                currentPlaylist = null;
            } else {
                currentPlaylist = playlists.get(0);
            }
        }
    }

}
```

```
package com.example.musicclient;

import java.util.List;

public class PlaylistTrackIterator implements TrackIterator {

    private final List<Track> tracks;
    private int currentIndex = -1;

    public PlaylistTrackIterator(List<Track> tracks) {
        this.tracks = tracks;
    }

    @Override
    public boolean hasNext() {
        return currentIndex + 1 < tracks.size();
    }

    @Override
    public Track next() {
        if (!hasNext()) {
            return null;
        }
        currentIndex++;
        return tracks.get(currentIndex);
    }

    @Override
    public boolean hasPrevious() {
        return currentIndex - 1 >= 0;
    }

    @Override
    public Track previous() {
        if (!hasPrevious()) {
            return null;
```

```
        }
        currentIndex--;
        return tracks.get(currentIndex);
    }

    @Override
    public Track current() {
        if (currentIndex < 0 || currentIndex >= tracks.size()) {
            return null;
        }
        return tracks.get(currentIndex);
    }
}
```

```
package com.example.musicclient;

import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.Collections;
import java.util.List;

public class RestClient {

    private final String baseUrl;
    private final HttpClient httpClient;
    private final ObjectMapper objectMapper;

    public RestClient(String baseUrl) {
        this.baseUrl = baseUrl;
        this.httpClient = HttpClient.newHttpClient();
        this.objectMapper = new ObjectMapper();
    }

    public List<Track> loadTracks() {
        try {
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(baseUrl + "/tracks"))
                    .GET()
                    .build();

            HttpResponse<String> response =
                    httpClient.send(request,
HttpResponse.BodyHandlers.ofString());

            if (response.statusCode() == 200) {
                return objectMapper.readValue(
                        response.body(),
                        new TypeReference<List<Track>>() {}
                );
            } else {
                System.err.println("Failed to load tracks, status = " +
response.statusCode());
            }
```

```
        } catch (Exception e) {
            System.err.println("Error loading tracks: " + e.getMessage());
        }
        return Collections.emptyList();
    }

    public List<Playlist> loadPlaylists() {
        try {
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(baseUrl + "/playlists"))
                    .GET()
                    .build();

            HttpResponse<String> response =
                    httpClient.send(request,
HttpResponse.BodyHandlers.ofString());

            if (response.statusCode() == 200) {
                return objectMapper.readValue(
                        response.body(),
                        new TypeReference<List<Playlist>>() {}
                );
            } else {
                System.err.println("Failed to load playlists, status = " +
response.statusCode());
            }
        } catch (Exception e) {
            System.err.println("Error loading playlists: " + e.getMessage());
        }
        return Collections.emptyList();
    }

    public void savePlaylist(Playlist playlist) {
        try {
            String json = objectMapper.writeValueAsString(playlist);

            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(baseUrl + "/playlists"))
                    .header("Content-Type", "application/json")
                    .POST(HttpRequest.BodyPublishers.ofString(json))
                    .build();

            HttpResponse<String> response =
                    httpClient.send(request,
HttpResponse.BodyHandlers.ofString());

            if (response.statusCode() == 200 || response.statusCode() == 201) {
                // Оновлюємо плейлист та треки з відповіді сервера (включаючи
id)
                Playlist updated = objectMapper.readValue(response.body(),
Playlist.class);

                playlist.setId(updated.getId());
                playlist.setName(updated.getName());

                playlist.getTracks().clear();
                playlist.getTracks().addAll(updated.getTracks());
            } else {
                System.err.println("Failed to save playlist, status = " +
response.statusCode());
            }
        } catch (Exception e) {
```

```
            System.err.println("Error saving playlist: " + e.getMessage());
        }
    }

    public void deletePlaylist(Long playlistId) {
        if (playlistId == null) {
            return;
        }
        try {
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(baseUrl + "/playlists/" + playlistId))
                    .DELETE()
                    .build();

            HttpResponse<String> response =
                    httpClient.send(request,
HttpResponse.BodyHandlers.ofString());

            if (response.statusCode() != 204 && response.statusCode() != 200) {
                System.err.println("Failed to delete playlist, status = " +
response.statusCode());
            }
        } catch (Exception e) {
            System.err.println("Error deleting playlist: " + e.getMessage());
        }
    }


}
```

```
package com.example.musicclient;

public class Track {
    private Long id;
    private String title;
    private String artist;
    private String album;
    private String filePath;

    public Track() {
    }

    public Track(Long id, String title, String artist, String album, String
filePath) {
        this.id = id;
        this.title = title;
        this.artist = artist;
        this.album = album;
        this.filePath = filePath;
    }

    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }
```

```
    public String getArtist() {
        return artist;
    }

    public String getAlbum() {
        return album;
    }


    public String getFilePath() {
        return filePath;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public void setAlbum(String album) {
        this.album = album;
    }


    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }

    @Override
    public String toString() {
        return title + " - " + artist;
    }
}
```

```
package com.example.musicclient;

public interface TrackIterator {
    boolean hasNext();
    Track next();
    boolean hasPrevious();
    Track previous();
    Track current();
}
```

pom.xml (server-side):

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>music-player-server</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <name>music-player-server</name>

    <properties>
       <java.version>17</java.version>
       <spring.boot.version>3.2.5</spring.boot.version>
    </properties>

    <dependencyManagement>
       <dependencies>
          <dependency>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-dependencies</artifactId>
             <version>${spring.boot.version}</version>
             <type>pom</type>
             <scope>import</scope>
          </dependency>
       </dependencies>
    </dependencyManagement>

    <dependencies>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
       </dependency>

       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-data-jpa</artifactId>
       </dependency>

       <dependency>
          <groupId>com.mysql</groupId>
          <artifactId>mysql-connector-j</artifactId>
          <scope>runtime</scope>
       </dependency>

       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
       </dependency>
    </dependencies>

    <build>
```

```
      <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>

            <configuration>

<mainClass>com.example.musicserver.MusicPlayerServerApplication</mainClass>
            </configuration>
            <executions>
               <execution>
                  <goals>
                     <goal>repackage</goal>
                  </goals>
               </execution>
            </executions>
         </plugin>
      </plugins>
   </build>


</project>
```

pom.xml (client-side):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>music-player-client</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>music-player-client</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <javafx.version>21.0.2</javafx.version>
    <junit.version>5.9.2</junit.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-controls</artifactId>
      <version>${javafx.version}</version>
    </dependency>

    <dependency>
```

```
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-graphics</artifactId>
      <version>${javafx.version}</version>
    </dependency>

    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-media</artifactId>
      <version>${javafx.version}</version>
    </dependency>

    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.17.1</version>
    </dependency>

    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.11.0</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
      </plugin>

      <plugin>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-maven-plugin</artifactId>
        <version>0.0.8</version>
        <configuration>
          <mainClass>com.example.musicclient.MainApp</mainClass>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Run-server.bat:

```
@echo off
cd /d "%~dp0"
echo Starting Music Player Server...
java -jar "target\music-player-server-1.0.0.jar"
pause
```