МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Острозька академія»

Навчально-науковий інститут

інформаційних технологій та бізнесу

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: **“Розробка Discord -бота для аудіоконтенту: аудіо -фільтри FFmpeg , рекомендації на основі історії та веб - інтерфейс”**

Виконав: студент 4 курсу, групи КН-41

першого (бакалаврського) рівня вищої освіти

спеціальності 122 Комп’ютерні науки

освітньо-професійної програми «Комп’ютерні науки»

Ковінський Дмитро Вікторович

Керівник: викладач кафедри інформаційних технологій та аналітики даних

Ляховчук Сергій Васильович

Рецензент: кандидат технічних наук, доцент,

доцент кафедри прикладної математики

Донецького національного університету

імені Василя Стуса

Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _____ (проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від « 20 » травня 2026 р.

Острог, 2026

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Розробка Discord -бота для аудіоконтенту: аудіо -фільтри FFmpeg , рекомендації на основі історії та веб - інтерфейс

Автор: ***Ковінський Дмитро Вікторович***

Науковий керівник: викладач кафедри інформаційних технологій та аналітики даних

Ляховчук Сергій Васильович

Захищена «……»…… 2026 року.

Пояснювальна записка до кваліфікаційної роботи:69 с., 23 рис., 18 табл., 1 додатків, 22 джерел.

Ключові слова: Discord-бот, Python, FFmpeg, discord.py, потокове відтворення аудіо, Slash-команди, Automix, Auto-Resume, SQLite, Docker.

Короткий зміст праці:

У кваліфікаційній роботі розглянуто процес проєктування та розробки музичного бота для платформи Discord. Актуальність теми зумовлена зростанням популярності голосових платформ для комунікації, а також потребою у створенні стабільних програмних рішень для потокового відтворення аудіоконтенту. Після припинення роботи низки популярних музичних ботів користувачі Discord-серверів почали активно використовувати self-hosted рішення, які дозволяють самостійно керувати функціональністю та розгортанням системи.

У роботі проаналізовано сучасні підходи до розробки Discord-ботів та особливості організації потокового відтворення аудіо. Розглянуто проблеми, пов’язані з асинхронною обробкою подій, керуванням чергою відтворення, відновленням стану після перезапуску застосунку та взаємодією користувача

з інтерфейсом бота. Окрему увагу приділено реалізації інтерактивного користувацького інтерфейсу на основі Slash-команд та кнопкових елементів керування Discord API.

У теоретичній частині обґрунтовано вибір технологічного стеку для реалізації програмного рішення. Для розробки бота використано мову програмування Python та бібліотеку discord.py, що забезпечує асинхронну взаємодію з Discord API. Для отримання та обробки аудіопотоків використано yt-dlp і FFmpeg. Зберігання історії прослуховувань, налаштувань серверів та стану черги реалізовано за допомогою SQLite та асинхронного драйвера aiosqlite.

Архітектура застосунку побудована за модульним принципом із використанням системи Cogs, сервісного шару та розділення відповідальностей між компонентами системи. Основні функціональні модулі реалізують керування чергою відтворення, обробку аудіопотоків, інтерактивний інтерфейс користувача та механізм автоматичного підбору треків (Automix).

У практичній частині роботи реалізовано Discord-бота з підтримкою потокового відтворення аудіо, кнопкового керування плеєром та системи автоматичного відновлення черги (Auto-Resume). Механізм Auto-Resume дозволяє відновлювати стан відтворення після перезапуску застосунку шляхом серіалізації даних черги у базі даних. Також реалізовано систему контекстних повідомлень DJ Persona, яка формує текстові повідомлення залежно від стану відтворення та дій користувачів.

Для підвищення стабільності роботи застосунку реалізовано обробку помилок, механізми очищення FFmpeg-процесів та перевірку коректності користувацького вводу. Окрему увагу приділено асинхронній роботі сервісів і мінімізації блокування основного event loop.

У роботі використано сучасні практики автоматизації розробки та тестування програмного забезпечення. Для перевірки працездатності основних модулів системи використано фреймворк pytest. Автоматичний

запуск тестів і перевірка збірки проєкту реалізовані за допомогою GitHub Actions. Для спрощення розгортання застосунку використано Docker та Docker Compose.

У результаті виконання роботи створено програмний продукт, який забезпечує відтворення аудіоконтенту у Discord-середовищі, підтримує інтерактивне керування та дозволяє автоматизувати процес відтворення музики.

У межах поточної реалізації основну увагу приділено серверній частині Discord-бота та механізмам потокового відтворення аудіо. Реалізація повноцінного веб-інтерфейсу адміністрування та розширених аудіофільтрів розглядається як напрям подальшого розвитку системи.

SUMMARY

Keywords: Discord bot, Python, FFmpeg, discord.py, audio streaming, slash commands, Automix, Auto-Resume, SQLite, Docker.

Abstract:

The qualification work considers the process of designing and developing a music bot for the Discord platform. The relevance of the topic is due to the growing popularity of voice platforms for communication, as well as the need to create stable software solutions for streaming audio content. After the termination of a number of popular music bots, Discord server users began to actively use self-hosted solutions that allow them to independently manage the functionality and deployment of the system.

The work analyzes modern approaches to the development of Discord bots and the features of organizing audio streaming. The problems associated with asynchronous event processing, playback queue management, state recovery after application restart, and user interaction with the bot interface are considered. Special

attention is paid to the implementation of an interactive user interface based on Slash commands and Discord API button controls.

The theoretical part justifies the choice of a technological stack for implementing a software solution. The bot was developed using the Python programming language and the discord.py library, which provides asynchronous interaction with the Discord API. yt-dlp and FFmpeg were used to receive and process audio streams. Storing listening history, server settings, and queue status was implemented using SQLite and the asynchronous aiosqlite driver.

The application architecture is built on a modular principle using the Cogs system, a service layer, and separation of responsibilities between system components. The main functional modules implement playback queue management, audio stream processing, an interactive user interface, and an automatic track selection mechanism (Automix).

In the practical part of the work, a Discord bot was implemented with support for streaming audio playback, button player control, and an automatic queue resumption system (Auto-Resume). The Auto-Resume mechanism allows you to restore the playback state after restarting the application by serializing queue data in the database. The DJ Persona contextual message system was also implemented, which generates text messages depending on the playback status and user actions.

To increase the stability of the application, error handling, mechanisms for cleaning FFmpeg processes, and checking the correctness of user input were implemented. Special attention was paid to asynchronous operation of services and minimizing blocking of the main event loop.

The work used modern practices for automating software development and testing. The pytest framework was used to check the operability of the main system modules. Automatic test launch and verification of the project assembly were implemented using GitHub Actions. Docker and Docker Compose were used to simplify the deployment of the application.

As a result of the work, a software product was created that provides audio content playback in the Discord environment, supports interactive control, and allows you to automate the music playback process.

Within the current implementation, the main attention was paid to the server part of the Discord bot and audio streaming mechanisms. The implementation of a full-fledged web administration interface and advanced audio filters is considered as a direction for further development of the system.

Зміст

ВСТУП 10

РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ 13

1.1. Аналіз предметної області та актуальність розробки 13

1.2. Порівняльний аналіз програм-аналогів 14

1.3. Обґрунтування вибору технологій розробки 16

1.4. Формування функціональних та нефункціональних вимог. 17

ВИСНОВКИ ДО РОЗДІЛУ 1 17

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 18

2.1. Аналіз предметної області 18

2.2. Проектування системи 22

2.3. Побудова концептуальної моделі. 29

2.4. Математичне та алгоритмічне забезпечення 31

ВИСНОВКИ ДО РОЗДІЛУ 2 36

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ 38

3.1 Засоби розробки та обґрунтування вибору технологій (Клієнтська логіка та інтерфейс) 38

3.2 Вимоги до технічного та програмного забезпечення 40

3.3 Опис програмної реалізації бізнес-логіки та інтерфейсу 43

3.4 Керівництво користувача 56

ВИСНОВКИ ДО РОЗДІЛУ 3 59

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 65

ДОДАТКИ 67

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API — Application Programming Interface;

UI — User Interface;

UX — User Experience;

CI/CD — Continuous Integration / Continuous Deployment;

UDP — User Datagram Protocol;

DSP — Digital Signal Processing;

MVP — Minimum Viable Product;

ER — Entity Relationship;

DI — Dependency Injection.

ВСТУП

Актуальність теми.

Платформа Discord активно використовується для голосової та текстової комунікації в ігрових, освітніх та професійних спільнотах. Одним із поширених сценаріїв використання Discord є спільне прослуховування аудіоконтенту за допомогою спеціалізованих музичних ботів. Після припинення роботи низки популярних публічних музичних ботів виникла потреба у створенні автономних програмних рішень, які можуть бути самостійно розгорнуті користувачами та забезпечувати стабільне відтворення аудіо.

Під час розробки музичних Discord-ботів виникають задачі, пов’язані з асинхронною обробкою подій, керуванням чергою відтворення, підтримкою потокового аудіо, відновленням стану після перезапуску застосунку та організацією зручного користувацького інтерфейсу. Актуальність роботи полягає у створенні програмного рішення, яке поєднує підтримку потокового відтворення аудіо, інтерактивне керування, автоматичне продовження відтворення та модульну архітектуру, придатну до подальшого розширення.

Мета роботи — проєктування та програмна реалізація музичного бота для платформи Discord із підтримкою потокового відтворення аудіо, інтерактивного керування та автоматизації роботи черги відтворення.

Об’єкт дослідження — процес розробки програмного забезпечення для платформи Discord.

Предмет дослідження — методи, архітектурні рішення та програмні засоби реалізації асинхронного музичного Discord-бота.

Завдання роботи:

1. Проаналізувати предметну область та існуючі аналоги музичних ботів для Discord.
2. Визначити функціональні вимоги до програмного рішення та обрати засоби реалізації системи.
3. Спроектувати архітектуру застосунку, структуру модулів та базу даних для збереження налаштувань і стану черги.
4. Реалізувати модулі відтворення аудіо, керування чергою, інтерактивного користувацького інтерфейсу та автоматичного підбору треків.
5. Реалізувати механізм автоматичного відновлення стану відтворення після перезапуску застосунку.
6. Провести тестування програмного продукту та налаштувати автоматичну перевірку збірки проєкту.

Методи дослідження.

Для розв’язання поставлених завдань використано методи системного аналізу, об’єктно-орієнтованого проєктування, асинхронного програмування та модульної розробки програмного забезпечення. Для реалізації програмного продукту використано мову програмування Python, бібліотеку discord.py, засоби обробки аудіо FFmpeg та систему автоматизованого тестування pytest.

Практичне значення отриманих результатів полягає у створенні програмного продукту для потокового відтворення аудіоконтенту в Discord-середовищі. Розроблене рішення підтримує модульну архітектуру, автоматизоване тестування та контейнеризоване розгортання за допомогою Docker. Програмний продукт може бути використаний як self-hosted рішення для музичного супроводу Discord-

серверів та слугувати основою для подальшого розвитку функціональності системи.

Межі реалізації роботи.

У межах кваліфікаційної роботи основну увагу зосереджено на реалізації серверної частини музичного Discord-бота, потоковому відтворенні аудіоконтенту, інтерактивному керуванні плеєром, збереженні стану черги та автоматизації відтворення за допомогою механізму Automix. У роботі реалізовано використання FFmpeg для обробки та передавання аудіопотоку у форматі, сумісному з голосовими каналами Discord. Повноцінна система користувацьких аудіофільтрів, таких як еквалайзер, Bassboost або Nightcore, розглядається як напрям подальшого розвитку.

Веб-інтерфейс адміністрування також не входить до завершеної реалізації поточної версії застосунку. Його створення передбачено як перспективу розвитку системи для зручного керування налаштуваннями серверів, чергами відтворення, статистикою та параметрами Automix через браузер. Отже, у межах цієї роботи реалізовано основну серверну логіку Discord-бота, а веб-панель керування та розширені аудіофільтри визначено як функціональні напрями майбутнього доопрацювання.

РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Аналіз предметної області та актуальність розробки

Процес цифровізації комунікацій сприяв активному розвитку платформ для голосового та текстового спілкування, однією з найпоширеніших серед яких є Discord. Платформа використовується для організації комунікації в ігрових, навчальних та тематичних спільнотах. Одним із поширених напрямків використання Discord є організація спільного прослуховування аудіоконтенту за допомогою спеціалізованих музичних ботів.

Предметна область даного дослідження охоплює розробку програмних рішень для потокового відтворення аудіоконтенту в середовищі Discord із підтримкою інтерактивного керування та автоматизації процесу відтворення.

Актуальність розробки зумовлена декількома факторами. Після припинення роботи популярних музичних ботів Rythm та Groovy виникла потреба у self-hosted рішеннях, які користувачі можуть самостійно розгортати та налаштовувати. Крім того, значна частина сучасних публічних ботів використовує модель платного доступу до окремих функцій або має обмеження безкоштовного використання.

Важливою складовою сучасних Discord-ботів є користувацький інтерфейс. Текстові команди поступово замінюються інтерактивними елементами керування, такими як кнопки, випадаючі меню та Slash-команди, що забезпечують більш зручну взаємодію користувача із системою.

Окрему увагу під час розробки музичних ботів приділяють автоматизації процесу відтворення аудіо. У межах даної роботи реалізовано механізм Automix, який дозволяє автоматично продовжувати відтворення після завершення користувацької черги на основі пов’язаних композицій та історії прослуховувань.

Також у роботі реалізовано механізм Auto-Resume, що забезпечує збереження стану черги та автоматичне відновлення відтворення після перезапуску застосунку.

1.2. Порівняльний аналіз програм-аналогів

1.2.1. Бот Jockie Music. Jockie Music є одним із найпоширеніших музичних ботів для Discord. Система підтримує відтворення аудіо з різних джерел та дозволяє використовувати декілька екземплярів бота на одному сервері. До переваг рішення належать підтримка кількох аудіоплатформ та стабільна робота під час потокового відтворення. Недоліками є закритий вихідний код, обмеження безкоштовного функціоналу та відсутність self-hosted моделі розгортання.

1.2.2. Бот MEE6 (Music Plugin). MEE6 є багатофункціональним Discord-ботом, який містить модуль для відтворення музики. Серед переваг можна виділити наявність веб-панелі керування та інтеграцію з іншими інструментами адміністрування серверів. Водночас значна частина музичного функціоналу доступна лише у платній версії, а сам інтерфейс містить велику кількість допоміжних модулів, які не пов’язані безпосередньо з відтворенням аудіо.

1.2.3. Бот Hydra. Hydra є музичним Discord-ботом із підтримкою інтерактивного керування плеєром. Основною особливістю рішення є використання кнопкового інтерфейсу для взаємодії з користувачем. Після змін політики YouTube підтримка частини функціоналу була обмежена, а бот переорієнтовано на інші джерела аудіоконтенту.

Таблиця 1.1 Порівняльна характеристика

Критерій	Jockie Music	MEE6	Hydra	Проектний застосунок
Підтримка YouTube/Spotify	+	+	частково	+
Безкоштовний Automix	-	-	-	+
Інтерактивний UI (Кнопки)	частково	-	+	+
Open Source / Self-Hosted	-	-	-	+

Висновок за підрозділом:

Аналіз існуючих музичних ботів для Discord показав, що більшість доступних рішень мають обмеження, пов’язані з платним функціоналом, закритим вихідним кодом або відсутністю self-hosted розгортання. Це визначає доцільність розробки програмного рішення з підтримкою інтерактивного керування, автоматизації відтворення та можливістю самостійного розгортання користувачами.

1.3. Обґрунтування вибору технологій розробки

Вибір програмно-технологічного стеку є критичним етапом, оскільки він визначає здатність системи швидко обробляти аудіопотоки в реальному часі без затримок (lag) та витримувати паралельне навантаження від різних серверів.

1.3.1. Мова програмування Python та фреймворк discord.py.

Для реалізації програмного продукту обрано мову програмування Python та бібліотеку discord.py. Використання асинхронної моделі програмування дозволяєє організувати одночасну обробку кількох типів подій без блокування основного event loop. Бібліотека discord.py забезпечує взаємодію з Discord API та підтримує Slash-команди, кнопкові елементи керування і систему подій.

1.3.2. Обробка аудіопотоків: yt-dlp та FFmpeg.

Для отримання аудіопотоків використовується бібліотека yt-dlp, яка дозволяє працювати з різними джерелами аудіоконтенту. Для обробки та транскодування аудіо використовується FFmpeg, що забезпечує підтримку формату Opus для голосових каналів Discord та можливість застосування базових аудіофільтрів.

1.3.3. Локальне збереження даних (aiosqlite).

Для локального збереження даних використано SQLite та асинхронний драйвер aiosqlite. База даних використовується для збереження налаштувань серверів, історії прослуховувань та стану черги відтворення. Обраний підхід спрощує розгортання застосунку та не потребує окремого серверу баз даних..

1.3.4. Контейнеризація та CI/CD.

Для спрощення розгортання застосунку використовується Docker та Docker Compose. Це дозволяє забезпечити однакове середовище виконання на різних системах. Для автоматизації перевірки працездатності проєкту використовується GitHub Actions та фреймворк pytest.

1.4. Формування функціональних та нефункціональних вимог.

Функціональні вимоги:

1. Підключення до голосових каналів Discord та потокове відтворення аудіо.
2. Реалізація інтерактивного інтерфейсу керування плеєром.
3. Підтримка черги відтворення та керування її елементами.
4. Реалізація механізму автоматичного продовження відтворення (Automix).
5. Реалізація системи збереження та відновлення стану черги (Auto-Resume).
6. Збереження історії прослуховувань та статистики користувачів.
7. Обробка помилок та некоректного користувацького вводу.

Нефункціональні вимоги:

1. Підтримка асинхронної обробки подій без блокування основного потоку виконання.
2. Забезпечення стабільного відтворення аудіо та коректної обробки помилок.
3. Підтримка роботи застосунку на кількох Discord-серверах одночасно.
4. Можливість контейнеризованого розгортання за допомогою Docker.
5. Підтримка автоматизованого тестування та перевірки збірки проєкту.

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі проведено аналіз предметної області та розглянуто особливості розробки музичних Discord-ботів. Виконано порівняння існуючих програм-аналогів та визначено основні функціональні можливості, які необхідно реалізувати у програмному продукті. Обґрунтовано вибір технологічного стеку та сформовано функціональні й нефункціональні вимоги до системи. Отримані результати стали основою для подальшого проєктування архітектури та програмної реалізації застосунку.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Аналіз предметної області

Виділення об’єктів дослідження.

Програмна реалізація системи базується на взаємодії декількох основних компонентів, які забезпечують відтворення аудіоконтенту, обробку команд користувача та керування станом системи.

1. **Голосовий клієнт (Voice Client)** — компонент, який забезпечує підключення до голосових каналів Discord та передачу аудіопотоку у форматі Opus. Голосовий клієнт відповідає за встановлення та підтримку з’єднання із голосовими серверами Discord.
2. **Черга відтворення (Queue)** — структура даних, яка використовується для збереження порядку відтворення аудіотреків у межах окремого Discord-сервера. Черга підтримує операції додавання, видалення, перемішування та автоматичного переходу між треками.
3. **Аудіоджерело (Audio Source)** — об’єкт, який формується під час отримання аудіопотоку за допомогою yt-dlp та FFmpeg. Аудіоджерело містить посилання на потік відтворення та метадані треку, зокрема назву, тривалість і автора.
4. **Інтерактивний інтерфейс користувача (Music Controls View)** — компонент системи, який реалізує кнопкове керування плеєром та обробку взаємодії користувача із системою через Discord API.
5. **База даних (Database)** — локальне сховище на основі SQLite, яке використовується для збереження налаштувань серверів, історії прослуховувань та стану черги відтворення.

У процесі аналізу предметної області було виділено основні інформаційні об’єкти, які беруть участь у роботі музичного Discord-бота. Кожен із цих об’єктів відповідає за окрему частину функціональності системи: обробку

команд користувача, керування аудіопотоком, збереження стану черги або взаємодію з базою даних. Узагальнену характеристику основних об’єктів інформаційної моделі наведено в таблиці 2.1

Таблиця 2.1. Основні об’єкти інформаційної моделі системи

Об’єкт	**Призначення**	**Основні дані**	**Взаємодія з іншими компонентами**
Користувач Discord	Ініціює команди та взаємодіє з ботом через інтерфейс Discord	Ідентифікатор користувача, команда, параметри запиту	Взаємодіє з Discord API та модулями команд
Discord-сервер	Середовище, у межах якого працює окрема черга відтворення	Ідентифікатор сервера, налаштування, історія	Пов’язаний із QueueService, PlayerService та базою даних
Голосовий канал	Канал, у якому здійснюється відтворення аудіо	Ідентифікатор каналу, стан підключення	Використовується Voice Client для передавання аудіопотоку
Аудіотрек	Одиниця аудіоконтенту, яка додається до черги	Назва, автор, тривалість, URL, ідентифікатор джерела	Обробляється yt-dlp, FFmpeg, QueueService та HistoryService
Черга відтворення	Зберігає порядок треків для конкретного Discord-сервера	Список треків, поточна позиція, стан відтворення	Взаємодіє з PlayerService, AutomixService та базою даних
Налаштування сервера	Зберігають параметри роботи бота для окремого сервера	Гучність, стан Automix, параметри DJ Persona	Використовуються сервісами під час обробки команд
Історія прослуховувань	Зберігає інформацію про відтворені треки	Ідентифікатор треку, користувач, дата й час відтворення	Використовується HistoryService та AutomixService
Стан черги	Дані для відновлення роботи після	Ідентифікатор сервера, список	Використовується механізмом Auto-Resume

	перезапуску застосунку	треків, голосовий канал	

Наведені об’єкти формують основу інформаційної моделі застосунку. Важливою особливістю системи є те, що дані зберігаються окремо для кожного Discord-сервера. Це дозволяє одному екземпляру бота обслуговувати декілька серверів одночасно без змішування черг, історії прослуховувань і налаштувань.

Окрему роль відіграють об’єкти «Черга відтворення» та «Стан черги», оскільки саме вони забезпечують роботу механізму Auto-Resume. Після зміни черги її актуальний стан зберігається у базі даних, що дозволяє відновити відтворення після перезапуску застосунку. Об’єкт «Історія прослуховувань» використовується не лише для статистики, а й для фільтрації повторів під час роботи Automix.

Побудова інформаційної моделі

Інформаційна модель системи побудована за принципом подійно-орієнтованої архітектури. Основним джерелом взаємодії із системою є події Discord API, які генеруються під час виконання Slash-команд або взаємодії користувача з інтерактивними елементами керування.

Після отримання події Discord API передає її до відповідного модуля системи (Cog), який виконує обробку запиту та взаємодіє із сервісним шаром застосунку. Сервісний шар реалізує основну бізнес-логіку системи, зокрема керування чергою відтворення, отримання аудіопотоків та оновлення стану плеєра.

Під час запуску відтворення система використовує yt-dlp для отримання інформації про аудіоджерело та FFmpeg для обробки аудіопотоку у форматі, сумісному з голосовими каналами Discord.

Стан черги та налаштування серверів зберігаються у базі даних SQLite. Це дозволяє реалізувати механізм Auto-Resume, який відновлює стан відтворення після перезапуску застосунку.

У разі зміни стану плеєра система автоматично оновлює інтерактивний інтерфейс користувача та відображає актуальний стан відтворення у Discord-каналі.

Опис існуючих обмежень на вхідні та вихідні дані

Вхідні дані

Вхідними даними системи є текстові запити користувачів, Slash-команди Discord та URL-посилання на аудіоконтент. Система підтримує обробку посилань на зовнішні медіаресурси та пошук аудіотреків за текстовими запитами.

Одним із основних обмежень є часові вимоги Discord API до обробки Slash-команд. Для уникнення перевищення допустимого часу відповіді використовується механізм відкладеної відповіді (Deferred Response), який дозволяє виконувати тривалі операції пошуку та отримання аудіоданих без переривання взаємодії з користувачем.

Також обмеження стосуються коректності вхідних посилань, доступності зовнішніх джерел аудіоконтенту та підтримуваних форматів медіаданих.

Вихідні дані

Основними вихідними даними системи є аудіопотоки для голосових каналів Discord, повідомлення користувацького інтерфейсу та дані, що зберігаються у базі даних.

Передавання аудіо здійснюється у форматі Opus відповідно до вимог Discord API. Обробка аудіоданих виконується за допомогою FFmpeg із підтримкою параметрів, сумісних із голосовими каналами Discord.

Дані, які зберігаються у SQLite, обмежуються структурою таблиць бази даних та використовуються для збереження історії прослуховувань, налаштувань серверів і стану черги відтворення. Для забезпечення стабільної роботи застосунку використовується асинхронна взаємодія з базою даних за допомогою aiosqlite.

2.2. Проектування системи

Архітектура програмного застосунку побудована за модульним принципом із розділенням відповідальностей між окремими компонентами. Такий підхід спрощує підтримку коду, тестування окремих частин системи та подальше розширення функціональності.

У межах проєкту використано трирівневу архітектуру, яка складається з рівня представлення, рівня бізнес-логіки та рівня доступу до даних.

Рівень представлення відповідає за взаємодію користувача із системою через Discord API. До цього рівня належать модулі команд, реалізовані за допомогою Cogs, а також інтерактивні елементи інтерфейсу, зокрема кнопки, модальні вікна та випадаючі списки. Основним завданням цього рівня є прийом команд користувача, первинна перевірка введених даних і передача запитів до сервісного шару.

Рівень бізнес-логіки містить основні сервіси застосунку, які реалізують роботу плеєра, черги відтворення, автоматичного підбору треків та обробки історії прослуховувань. До цього рівня належать такі компоненти, як PlayerService, QueueService, AutomixService та HistoryService. Вони відповідають за керування станом відтворення, взаємодію з FFmpeg, обробку подій завершення треку та формування наступного елемента черги.

Рівень доступу до даних забезпечує взаємодію застосунку з локальною базою даних SQLite. Для асинхронної роботи з базою даних використовується aiosqlite. Цей рівень відповідає за збереження налаштувань серверів, історії прослуховувань і стану черги відтворення.

У застосунку також використано підхід, близький до патерну впровадження залежностей. Сервіси передаються між компонентами через ініціалізацію об’єктів, що дозволяє зменшити зв’язаність модулів і спростити тестування окремих частин системи.

Така архітектура відповідає задачам проєкту, оскільки Discord-бот має одночасно обробляти події користувачів, підтримувати аудіопотік, зберігати стан черги та реагувати на зміну стану голосового каналу.

2.2.1. Структура програмного проєкту

Програмний застосунок має модульну структуру, побудовану за принципом розділення відповідальностей між окремими компонентами системи. Такий підхід дозволяє спростити підтримку коду, тестування окремих модулів та подальше розширення функціональності застосунку.

Структура проєкту організована у вигляді окремих каталогів, кожен із яких відповідає за певний функціональний компонент системи. Основні модулі програмного застосунку наведено в таблиці 2.2.

Таблиця 2.1. Основні каталоги та модулі програмного проєкту

Каталог / модуль	**Призначення**
bot/	Основний модуль запуску Discord-бота та ініціалізації компонентів системи
cogs/	Модулі Slash-команд та взаємодії з Discord API
services/	Реалізація бізнес-логіки застосунку
ui/	Інтерактивні елементи керування плеєром
repository/	Робота з базою даних SQLite
database/	Файли локальної бази даних та міграції

tests/	Unit- та integration-тести
docker/	Конфігурації Docker та Docker Compose
utils/	Допоміжні функції та службові модулі
.github/workflows/	Конфігурації CI/CD GitHub Actions

Каталог cogs містить модулі команд Discord-бота, які відповідають за взаємодію користувача із системою через Slash-команди та інтерактивні компоненти Discord API. Основна бізнес-логіка винесена до каталогу services, що дозволяє мінімізувати залежність між рівнем представлення та внутрішньою логікою застосунку.

Модулі PlayerService, QueueService, AutomixService та HistoryService реалізують ключові функції системи: відтворення аудіо, керування чергою, автоматичне продовження відтворення та формування статистики прослуховувань.

Каталог repository використовується для ізоляції SQL-запитів та взаємодії з локальною базою даних SQLite через асинхронний драйвер aiosqlite. Це дозволяє відокремити логіку доступу до даних від інших компонентів системи.

Інтерактивні елементи керування плеєром реалізовані в каталозі ui. До них належать кнопки керування відтворенням, випадаючі списки та модальні вікна Discord API.

Для забезпечення стабільності роботи та автоматизації перевірки коду в проєкті використовується система автоматизованого тестування pytest, а також GitHub Actions для запуску CI/CD-процесів після внесення змін до репозиторію.

Контейнеризоване розгортання застосунку реалізовано за допомогою Docker та Docker Compose, конфігураційні файли яких розміщено в каталозі docker.

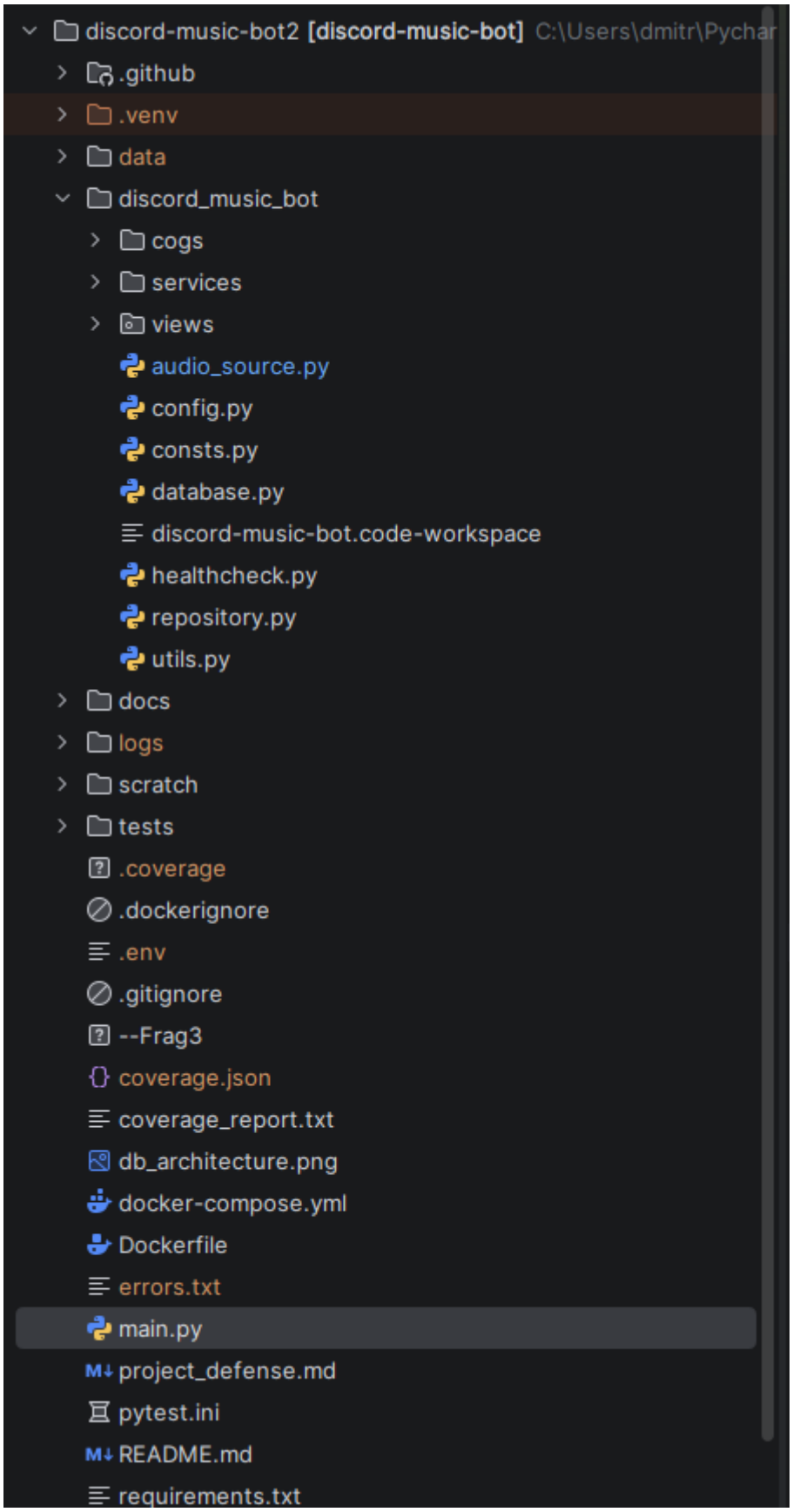

Рис.2.1. Структура програмного проєкту

2.2.2. Проектування бази даних

Для збереження стану системи, історії прослуховувань та налаштувань серверів у програмному застосунку використовується локальна реляційна база даних SQLite. Вибір SQLite обумовлений простотою розгортання, відсутністю необхідності використання окремого серверу баз даних та достатньою продуктивністю для self-hosted Discord-бота.

Взаємодія із базою даних реалізована за допомогою асинхронного драйвера aiosqlite, що дозволяє виконувати SQL-запити без блокування основного event loop застосунку.

База даних використовується для:

- збереження історії прослуховувань;
- збереження налаштувань серверів;
- реалізації механізму Auto-Resume;
- формування статистики користувачів;
- збереження параметрів роботи плеєра.

Основні таблиці бази даних

Основні таблиці бази даних наведено в таблиці 2.3.

Таблиця 2.2. Основні таблиці бази даних

Таблиця	**Призначення**
history	Зберігання історії прослуховувань треків
guild_settings	Налаштування Discord-серверів
queue_state	Збереження стану черги для Auto-Resume
user_stats	Дані статистики користувачів
player_state	Поточний стан плеєра
automix_cache	Тимчасові дані для механізму Automix

Опис таблиці history

Таблиця history використовується для збереження інформації про відтворені треки. Дані цієї таблиці використовуються для формування статистики та роботи механізму Automix.

Таблиця 2.3.Опис таблиці history

Поле	**Тип даних**	**Призначення**
id	INTEGER	Первинний ключ
guild_id	TEXT	Ідентифікатор Discord-сервера
user_id	TEXT	Ідентифікатор користувача
track_title	TEXT	Назва треку
track_url	TEXT	URL аудіоджерела
played_at	DATETIME	Час відтворення

Опис таблиці queue_state

Таблиця queue_state використовується для реалізації механізму Auto-Resume. У таблиці зберігається поточний стан черги відтворення для кожного Discord-сервера.

Таблиця 2.4. Структура таблиці queue_state

Поле	**Тип даних**	**Призначення**
guild_id	TEXT	Ідентифікатор Discord-сервера
voice_channel_id	TEXT	Голосовий канал
queue_data	TEXT	Серіалізований JSON стан черги
updated_at	DATETIME	Час останнього оновлення

Опис таблиці guild_settings

Таблиця guild_settings використовується для збереження налаштувань окремих Discord-серверів.

Таблиця 2.5. Структура таблиці guild_settings

Поле	**Тип даних**	**Призначення**
guild_id	TEXT	Ідентифікатор Discord-сервера
volume	INTEGER	Рівень гучності
automix_enabled	BOOLEAN	Стан Automix
dj_persona_enabled	BOOLEAN	Стан системи контекстних повідомлень
autoplay_enabled	BOOLEAN	Автоматичне продовження відтворення

Структура бази даних спроєктована таким чином, щоб мінімізувати обсяг операцій запису та забезпечити швидке отримання даних під час роботи застосунку. Для збереження складних структур даних, зокрема стану черги відтворення, використовується серіалізація у формат JSON.

Окрему роль у роботі системи відіграє таблиця queue_state, яка забезпечує відновлення черги після перезапуску застосунку. Після кожної зміни черги PlayerService або QueueService оновлюють відповідний запис у базі даних.

Таблиця history використовується для формування статистики прослуховувань та фільтрації повторів під час роботи механізму Automix.

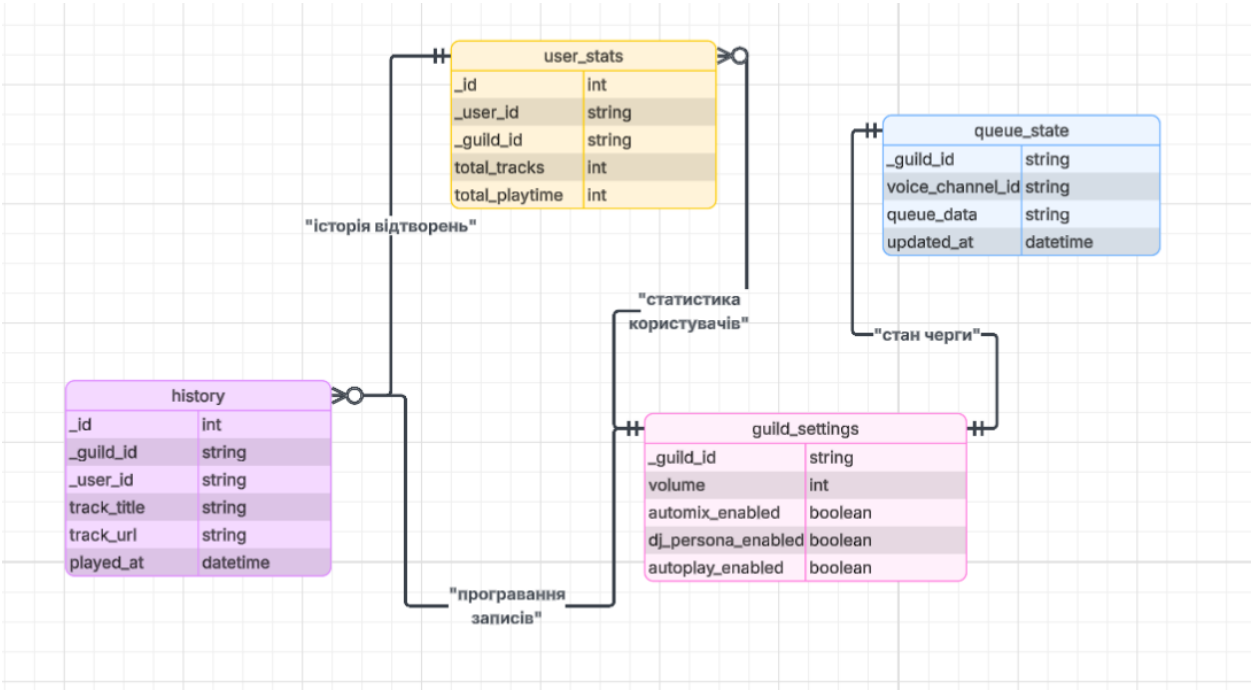

Рис.2.2. ER-діаграма бази даних

2.3. Побудова концептуальної моделі.

Концептуальна модель системи описує взаємодію користувача, Discord API, сервісного шару, зовнішніх джерел аудіоконтенту та бази даних.

Основним сценарієм роботи системи є запуск відтворення аудіотреку за допомогою Slash-команди. Користувач викликає команду /play і передає текстовий запит або URL-посилання. Після цього Discord API формує подію взаємодії та передає її відповідному модулю команд. Модуль команд виконує первинну перевірку запиту, зокрема перевіряє наявність користувача у голосовому каналі, після чого передає запит до сервісного шару.

Сервіс отримання аудіоджерела звертається до yt-dlp для пошуку треку або отримання метаданих за посиланням. Після отримання інформації про трек формується об’єкт аудіоджерела, який передається до QueueService. QueueService додає трек до черги відповідного Discord-сервера. Якщо на момент додавання черга була порожньою, PlayerService ініціює запуск відтворення через FFmpeg.

Після завершення поточного треку PlayerService обробляє подію завершення відтворення. Дані про відтворений трек передаються до HistoryService для збереження в історії. Далі система перевіряє наявність наступного треку в черзі. Якщо черга порожня і активовано режим Automix, AutomixService формує наступний трек на основі пов’язаного контенту та історії прослуховувань.

Окремим елементом концептуальної моделі є механізм Auto-Resume. Після зміни стану черги система зберігає її актуальний стан у базі даних. У разі перезапуску застосунку бот отримує збережені дані, відновлює чергу та може продовжити роботу з попереднім станом.

Таким чином, концептуальна модель системи охоплює повний цикл взаємодії: від отримання команди користувача до відтворення аудіо, оновлення інтерфейсу, збереження історії та автоматичного відновлення стану системи.

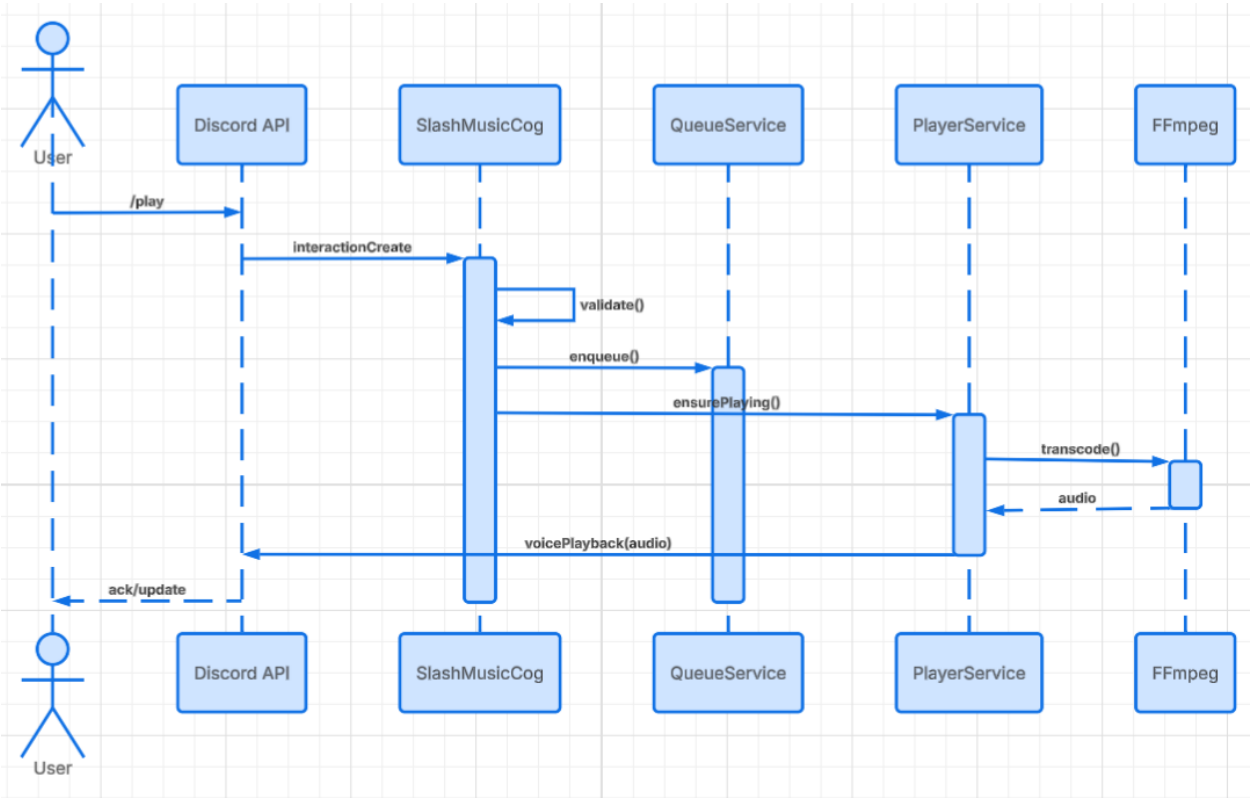

Рис.2.3. Блок-схема команди /play

2.4. Математичне та алгоритмічне забезпечення

Алгоритм обробки аудіопотоку

Після отримання команди від користувача система виконує пошук або обробку URL-посилання за допомогою бібліотеки yt-dlp. Отримані метадані та аудіопотік передаються до FFmpeg для подальшої обробки та конвертації у формат Opus, який підтримується голосовими каналами Discord.

Під час відтворення застосунок використовує потокову передачу аудіо без попереднього повного завантаження файлу на диск. Це дозволяє зменшити використання дискового простору та скоротити час очікування перед початком відтворення.

Для забезпечення стабільної роботи використовуються механізми автоматичного перепідключення та буферизації аудіопотоку.

Алгоритм роботи системи Automix

Механізм Automix використовується для автоматичного продовження відтворення після завершення користувацької черги.

Після завершення поточного треку система:

1. Отримує інформацію про поточний трек.
2. Формує список пов’язаних композицій.
3. Перевіряє історію прослуховувань.
4. Відфільтровує треки, які вже відтворювались раніше.
5. Додає новий трек до черги відтворення.

Такий підхід дозволяє автоматизувати процес відтворення музики та зменшити кількість повторів однакових композицій.

Алгоритм Auto-Resume

Механізм Auto-Resume забезпечує автоматичне відновлення стану плеєра після перезапуску застосунку.

Після кожної зміни черги система:

1. серіалізує список треків;
2. зберігає стан відтворення у базі даних SQLite;
3. записує ідентифікатор голосового каналу та параметри відтворення.

Після запуску застосунку бот:

1. Отримує збережені дані з бази даних.
2. Відновлює чергу відтворення.
3. Підключається до голосового каналу.
4. Продовжує відтворення збереженого треку.

Алгоритм формування статистики

Для формування статистики прослуховувань система використовує записи з локальної бази даних.

Під час виконання команд статистики:

1. виконується підрахунок кількості відтворень треків;
2. формується список найбільш популярних композицій;
3. обчислюється активність користувачів.

Для обробки даних використовуються SQL-запити з операціями агрегації та сортування.

Рис.2.4.Вивід статистики сесії

2.4.1. Псевдокод алгоритму Automix

Для реалізації механізму Automix використовується алгоритм автоматичного відбору наступного треку на основі пов’язаних композицій та історії прослуховувань.

Псевдокод алгоритму Automix наведено нижче.

```
function AUTOMIX(last_track, history, queue):

related_tracks ← get_related_tracks(last_track)

filtered_tracks ← []

for track in related_tracks:

if track not in history
and track not in queue:
filtered_tracks.append(track)

if filtered_tracks is empty:
```

```
return None

next_track ← random_choice(filtered_tracks)

return next_track
```

Алгоритм працює за принципом отримання списку пов’язаних композицій для поточного треку та подальшої фільтрації повторів. Для перевірки повторів використовується історія прослуховувань і поточна черга відтворення.

Обчислювальна складність алгоритму залежить від способу перевірки наявності треку в історії. У базовому варіанті при послідовному порівнянні кожного елемента складність становить:

$O(n \cdot m)$,

де:

- n — кількість пов’язаних треків;
- m — кількість елементів історії прослуховувань.

Для оптимізації перевірки історія може бути представлена у вигляді множини (set), що дозволяє зменшити складність перевірки до:

$O(n)$,

оскільки перевірка входження елемента у множину виконується за константний час.

2.4.2. Псевдокод механізму Auto-Resume

Механізм Auto-Resume забезпечує автоматичне відновлення стану плеєра після перезапуску застосунку.

Під час кожної зміни черги система виконує серіалізацію поточного стану та записує його до бази даних SQLite.

Псевдокод алгоритму збереження стану наведено нижче.

```
function SAVE_QUEUE_STATE(guild_id, queue):

serialized_queue ← serialize(queue)

database.update(
guild_id,
serialized_queue
)
```

Після запуску застосунку виконується процедура відновлення стану:

```
function RESTORE_QUEUE_STATE(guild_id):

saved_state ← database.read(guild_id)

if saved_state exists:

queue ← deserialize(saved_state)

connect_to_voice_channel()

start_playback(queue.first_track)
```

Для серіалізації використовується формат JSON, який містить:

- список треків;
- ідентифікатор голосового каналу;
- параметри відтворення;
- поточну позицію у черзі.

Складність операції серіалізації залежить від кількості елементів черги та становить:

$O(n)$,

де n — кількість треків у черзі відтворення.

ВИСНОВКИ ДО РОЗДІЛУ 2

У другому розділі виконано проєктування інформаційного та алгоритмічного забезпечення музичного Discord-бота. У межах розділу визначено основні компоненти системи, описано взаємодію між сервісами застосунку та сформовано концептуальну модель роботи програмного продукту.

У процесі проєктування виділено основні об’єкти системи, зокрема модулі керування відтворенням, чергою, аудіоджерелами та базою даних. Побудовано інформаційну модель застосунку на основі подійно-орієнтованого підходу, який забезпечує обробку команд користувачів та зміну стану системи в режимі реального часу.

Проведено аналіз вхідних та вихідних даних системи, а також визначено основні обмеження, пов’язані з роботою Discord API, передаванням аудіопотоків та асинхронною взаємодією з базою даних SQLite.

У розділі також обґрунтовано використання трирівневої архітектури із розділенням рівня представлення, бізнес-логіки та доступу до даних. Такий підхід дозволяє спростити підтримку коду, тестування окремих компонентів та подальше розширення функціональності системи.

Окрему увагу приділено алгоритмам обробки аудіопотоків, автоматичного продовження відтворення (Automix), збереження стану черги (Auto-Resume) та формування статистики прослуховувань. Розроблені

алгоритми забезпечують стабільну роботу застосунку та відповідають функціональним вимогам, сформованим у першому розділі.

Отримані результати стали основою для подальшої програмної реалізації системи, опис якої наведено у наступному розділі.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Засоби розробки та обґрунтування вибору технологій (Клієнтська логіка та інтерфейс)

Для реалізації музичного Discord-бота було використано набір сучасних програмних засобів та бібліотек, орієнтованих на створення асинхронних мережевих застосунків і обробку мультимедійного контенту. Основною мовою програмування обрано Python, оскільки вона має розвинену екосистему бібліотек для роботи з Discord API, аудіообробкою та автоматизацією розгортання.

Базовим інструментом взаємодії з платформою Discord є бібліотека discord.py версії 2.x. Її використання дозволяє реалізувати підтримку Slash-команд, інтерактивних кнопок, модальних вікон та інших компонентів Discord API. Завдяки асинхронній моделі роботи бібліотека забезпечує одночасну обробку подій без блокування основного потоку виконання.

Для отримання аудіопотоків із зовнішніх джерел використовується бібліотека yt-dlp, яка дозволяє отримувати метадані треків та прямі посилання на аудіопотоки без необхідності завантаження медіафайлів на локальний диск.

Обробка та конвертація аудіо реалізована за допомогою FFmpeg. Цей інструмент використовується для транскодування аудіопотоку у формат Opus, який підтримується голосовими каналами Discord.

Для збереження налаштувань серверів, історії прослуховувань та стану черги використовується SQLite. Взаємодія із базою даних реалізована через асинхронний драйвер aiosqlite.

Для автоматизації перевірки працездатності системи використовується фреймворк pytest. Автоматичний запуск тестів та перевірка збірки реалізовані за допомогою GitHub Actions.

Контейнеризоване розгортання застосунку реалізовано за допомогою Docker та Docker Compose.

Таблиця 3.1. Використані технології та засоби розробки

Технологія	**Призначення**
Python	Основна мова програмування
discord.py	Взаємодія з Discord API
yt-dlp	Отримання аудіопотоків
FFmpeg	Обробка та транскодування аудіо
SQLite	Локальна база даних
aiosqlite	Асинхронна взаємодія з БД
pytest	Автоматизоване тестування
Docker	Контейнеризація застосунку
Docker Compose	Керування контейнерами
GitHub Actions	CI/CD автоматизація
Git	Контроль версій

Для розробки програмного продукту використовувалося середовище PyCharm та система контролю версій Git. Управління залежностями реалізовано через файл requirements.txt.

Використання Docker дозволило забезпечити однакове середовище виконання застосунку незалежно від операційної системи. GitHub Actions використовується для автоматичної перевірки працездатності застосунку після внесення змін до репозиторію.

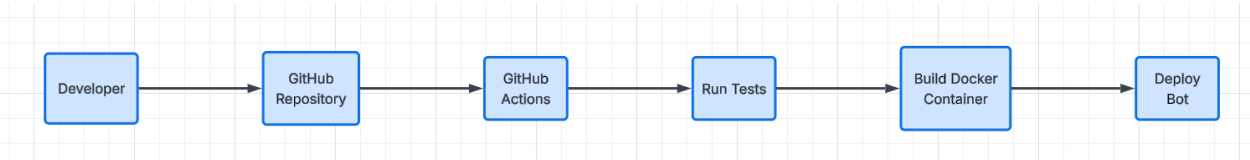

Рис. 3.1. Схема процесу CI/CD та автоматизації тестування

На рисунку 3.1 наведено загальну схему процесу автоматизації перевірки та розгортання програмного застосунку. Після внесення змін до репозиторію GitHub Actions автоматично запускає тестування та перевірку збірки Docker-контейнера.

3.2 Вимоги до технічного та програмного забезпечення

Для коректної роботи музичного Discord-бота необхідна наявність стабільного мережевого з’єднання, підтримка сучасної версії Python та встановлених компонентів для обробки аудіопотоків. Програмний продукт може працювати як на локальному комп’ютері користувача, так і на віддаленому сервері або VPS.

Однією з особливостей застосунку є використання потокового відтворення аудіо та асинхронної обробки подій. У зв’язку з цим система висуває вимоги до стабільності Інтернет-з’єднання та доступності Discord API.

Для забезпечення роботи голосових каналів Discord застосунок використовує FFmpeg та бібліотеку PyNaCl. Відсутність або некоректне налаштування цих компонентів унеможливлює передачу аудіо.

Таблиця 3.2. Мінімальні системні вимоги

Компонент	**Мінімальні вимоги**
Операційна система	Windows 10 / Linux
Python	Версія 3.11 або новіша
Оперативна пам’ять	2 GB
Процесор	2 ядра
Інтернет-з’єднання	Стабільний доступ до мережі
FFmpeg	Встановлений у системі
Docker	Версія 24+ (опціонально)

Таблиця 3.3. Рекомендовані системні вимоги

Компонент	**Рекомендовані вимоги**
Операційна система	Linux Server / Ubuntu
Python	Версія 3.11+
Оперативна пам’ять	4 GB
Процесор	4 ядра
SSD-накопичувач	Від 10 GB
Інтернет-з’єднання	Низька затримка до серверів Discord

Для встановлення всіх необхідних бібліотек використовується файл requirements.txt. Основні залежності встановлюються автоматично за допомогою менеджера пакетів pip.

Під час контейнеризованого розгортання Docker автоматично створює середовище виконання та встановлює необхідні компоненти системи. Це дозволяє мінімізувати кількість помилок, пов’язаних із налаштуванням середовища.

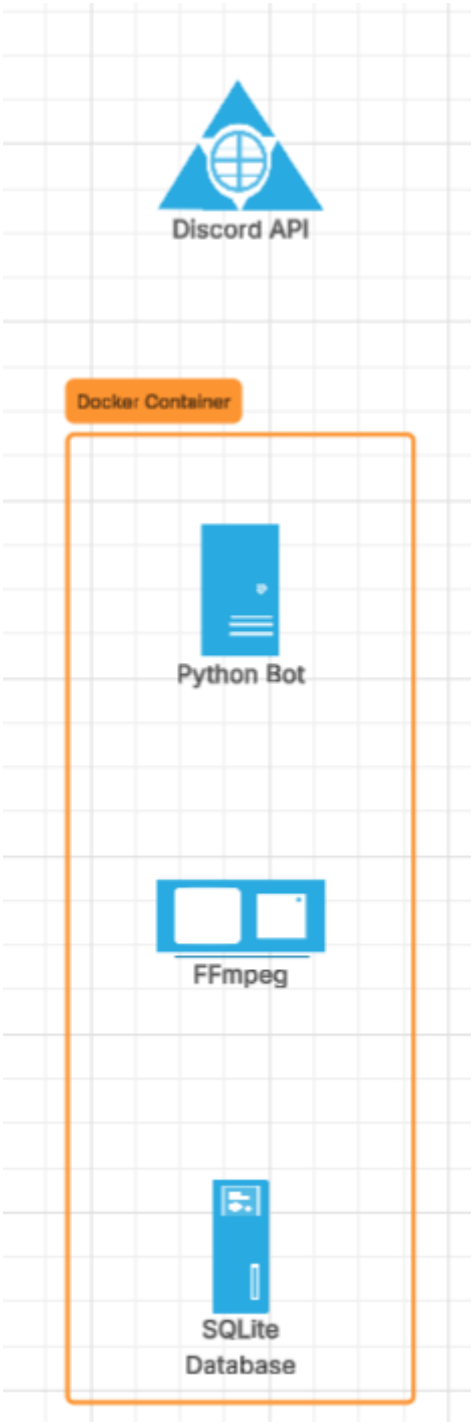

Рис. 3.2. Схема Docker-розгортання програмного застосунку

На рисунку 3.2 наведено загальну схему контейнеризованого розгортання музичного Discord-бота. Основна логіка застосунку працює всередині Docker-контейнера, який взаємодіє з Discord API, FFmpeg та локальною базою даних SQLite.

Для забезпечення стабільної роботи Discord-бота рекомендується використовувати Linux-сервер або VPS із постійним доступом до мережі Інтернет.

3.3 Опис програмної реалізації бізнес-логіки та інтерфейсу

3.3.1 Архітектура модуля команд (Slash Cogs)

Вхідною точкою взаємодії користувача із системою є модуль Slash-команд, реалізований за допомогою механізму app_commands бібліотеки discord.py. Використання Slash-команд дозволяє забезпечити інтеграцію з нативним інтерфейсом Discord та автоматичну валідацію параметрів ще до надсилання запиту до сервера застосунку.

Основним компонентом цього рівня є модуль SlashMusicCog, який відповідає за:

- реєстрацію команд;
- прийом interaction-подій;
- перевірку вхідних параметрів;
- передачу запитів до сервісного шару.

Перед виконанням команди система виконує перевірку:

- наявності користувача у голосовому каналі;
- коректності введених параметрів;
- доступності голосового клієнта;
- поточного стану плеєра.

Після успішної перевірки запит передається до відповідного сервісу бізнес-логіки.

Таблиця 3.4. Основні Slash-команди системи

Команда	**Призначення**
/play	Додавання треку до черги
/pause	Призупинення відтворення
/resume	Продовження відтворення
/skip	Перехід до наступного треку
/queue	Перегляд черги
/history	Перегляд історії прослуховувань
/mix	Налаштування Automix та DJ Persona
/volume	Зміна гучності
/stop	Повне завершення відтворення

Лістинг 3.1 Реєстрація Slash-команди `/play`

Приклад реєстрації Slash-команди `/play` наведено у лістингу 3.1.

```
@app_commands.command(
name="play",
description="Відтворити аудіо"
)
async def play(
self,
interaction: discord.Interaction,
query: str
):

await interaction.response.defer()

await self.player_service.play(
interaction,
query
)
```

У наведеному фрагменті коду використовується механізм Slash-команд бібліотеки discord.py. Після отримання interaction-події система передає запит до сервісного шару застосунку.

Використання Slash-команд дозволяє зменшити кількість помилок користувача під час введення параметрів та спрощує взаємодію із системою. Discord-клієнт автоматично відображає доступні параметри команд та забезпечує автозаповнення під час введення.

Рис. 3.3. Використання Slash-команди /play у Discord

На рисунку 3.3 наведено приклад використання Slash-команди /play у середовищі Discord. Інтерфейс Discord автоматично відображає доступні параметри команди та спрощує процес взаємодії користувача із системою.

3.3.2 Реалізація інтерактивного плеєра (Music Controls UI)

Інтерактивний інтерфейс плеєра реалізовано за допомогою класу discord.ui.View, який дозволяє створювати кнопки, випадаючі списки та інші елементи взаємодії безпосередньо у Discord-повідомленнях.

Після запуску відтворення бот автоматично формує Embed-повідомлення з інформацією про поточний трек та генерує набір кнопок керування:

- Play/Pause;
- Skip;
- Queue;
- Volume;
- Mix.

Натискання кнопок обробляється через callback-функції, які взаємодіють із сервісним шаром застосунку.

Під час зміни стану відтворення інтерфейс автоматично оновлюється. Наприклад, після призупинення музики кнопка Pause змінює текст та іконку на Resume.

Для обробки помилок використовуються ephemeral-повідомлення Discord API. Такі повідомлення бачить лише користувач, який виконав дію, що дозволяє уникнути перевантаження текстового каналу службовими повідомленнями.

Таблиця 3.5. Основні елементи інтерфейсу плеєра

Елемент	**Призначення**
Play/Pause	Керування відтворенням
Skip	Перехід до наступного треку
Queue	Перегляд черги
Volume	Налаштування гучності
Mix	Налаштування Automix

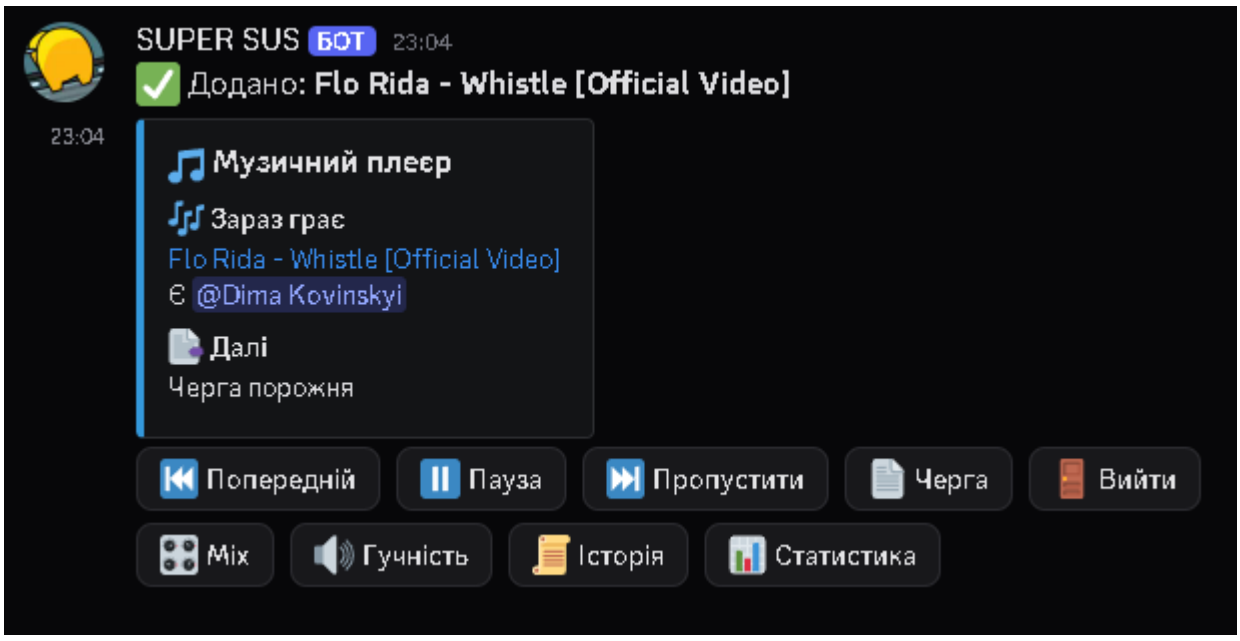

Рис. 3.4. Інтерактивний плеєр під час відтворення музики

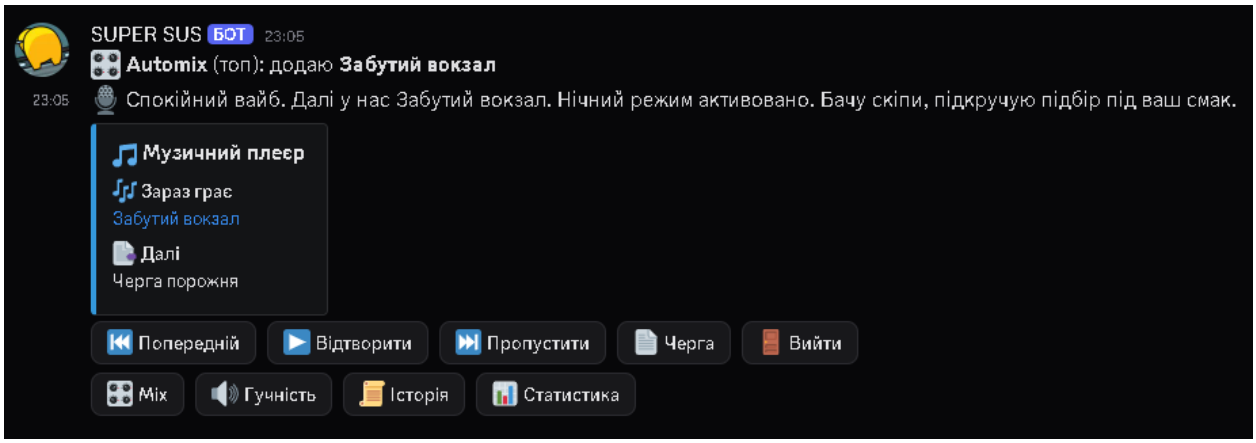

Рис. 3.5. Динамічна зміна кнопки Pause/Resume

На рисунках 3.4–3.5 наведено приклад роботи інтерактивного плеєра. Інтерфейс автоматично оновлює стан кнопок залежно від поточного режиму роботи плеєра.

Лістинг 3.2 Обробка натискання кнопки Pause

Приклад callback-функції для кнопки призупинення відтворення наведено у лістингу 3.2.

```
@discord.ui.button(
label="Pause",
style=discord.ButtonStyle.secondary
)
async def pause_button(
self,
interaction: discord.Interaction,
button: discord.ui.Button
):

await self.player.pause()

button.label = "Resume"

await interaction.response.edit_message(
```

```
view=self
)
```

Після натискання кнопки система виконує призупинення відтворення та автоматично оновлює інтерфейс користувача.

3.3.3 Реалізація системи Automix

Для автоматичного продовження відтворення музики у застосунку реалізовано окремий сервіс AutomixService. Основним завданням цього компонента є автоматичне додавання нових композицій до черги після завершення користувацького списку відтворення.

Після завершення поточного треку система перевіряє:

- чи активовано режим Automix;
- чи залишились елементи у черзі;
- чи доступне аудіоджерело для формування рекомендацій.

Якщо черга порожня, AutomixService отримує список пов’язаних композицій на основі поточного або останнього відтвореного треку. Для цього використовується бібліотека yt-dlp.

Отриманий список проходить додаткову фільтрацію:

- видаляються треки, які вже містяться у черзі;
- виключаються композиції з історії прослуховувань;
- перевіряється доступність аудіоджерела.

Після фільтрації система автоматично додає новий трек до черги та запускає подальше відтворення.

Таблиця 3.6. Основні функції AutomixService

Функція	**Призначення**
Отримання рекомендацій	Формування списку пов’язаних композицій
Фільтрація повторів	Запобігання повторному відтворенню
Аналіз історії	Використання history для відбору треків
Автоматичне додавання	Додавання нового треку до QueueService
Підтримка безперервного відтворення	Робота режиму «нескінченної» черги

Використання Automix дозволяє автоматизувати процес відтворення музики та забезпечити безперервну роботу плеєра навіть після завершення користувацької черги.

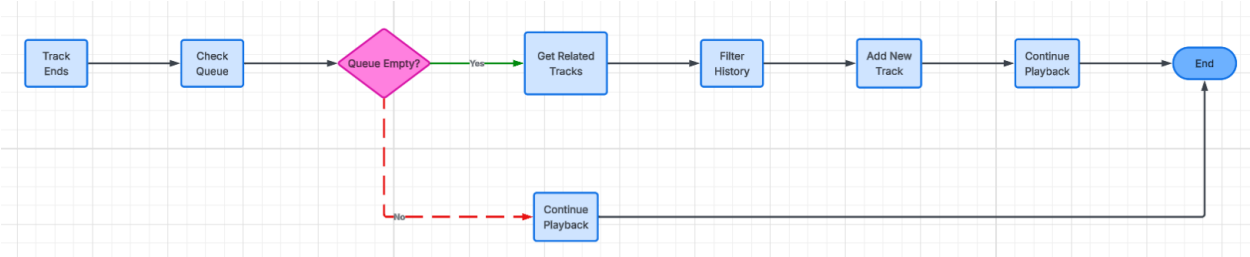

Рис. 3.6. Схема роботи механізму Automix

На рисунку 3.6 наведено спрощену схему роботи механізму Automix. Після завершення поточного треку система перевіряє стан черги, отримує список пов’язаних композицій та автоматично додає новий трек до відтворення.

3.3.4 Реалізація механізму Auto-Resume

Однією з ключових функцій застосунку є механізм Auto-Resume, який забезпечує автоматичне відновлення стану плеєра після перезапуску Discord-бота.

Після кожної зміни черги QueueService виконує серіалізацію поточного стану та записує його до бази даних SQLite. Зберігаються:

- список треків;
- поточна позиція у черзі;
- ідентифікатор голосового каналу;
- параметри відтворення.

Після запуску застосунку система виконує:

1. Зчитування збереженого стану із бази даних.
2. Відновлення черги відтворення.
3. Підключення до голосового каналу.
4. Відновлення відтворення поточного треку.

Механізм Auto-Resume дозволяє уникнути втрати черги у випадку перезапуску контейнера, помилки застосунку або короткочасного відключення сервера.

Таблиця 3.7. Основні етапи роботи Auto-Resume

Етап	**Призначення**
Serialize Queue	Збереження стану черги
Save State	Запис даних у SQLite
Restore Queue	Відновлення списку треків
Reconnect Voice	Підключення до голосового каналу
Resume Playback	Продовження відтворення

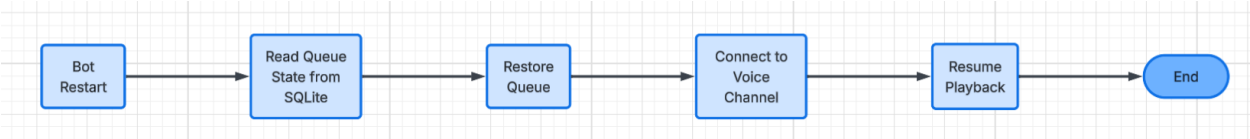

Рис. 3.7. Алгоритм роботи Auto-Resume

На рисунку 3.7 наведено послідовність відновлення стану плеєра після перезапуску застосунку. Механізм Auto-Resume дозволяє зберегти безперервність роботи системи та уникнути втрати поточної черги відтворення.

Лістинг 3.3 Серіалізація стану черги

Приклад збереження стану черги у базі даних наведено у лістингу 3.3.

```
async def save_queue_state(
self,
guild_id: int,
queue_data: dict
):

serialized_data = json.dumps(queue_data)

await self.db.execute(
"""
UPDATE queue_state
SET queue_data = ?
WHERE guild_id = ?
""",
(serialized_data, guild_id)
)

await self.db.commit()
```

Для збереження стану черги використовується серіалізація структури даних у формат JSON та асинхронна взаємодія з SQLite через aiosqlite.

3.3.5 Реалізація системи контекстних повідомлень DJ Persona

Для підвищення інтерактивності та покращення користувацького досвіду у застосунку реалізовано систему контекстних повідомлень DJ Persona. Основним призначенням цього компонента є формування текстових повідомлень залежно від стану відтворення та окремих подій під час роботи плеєра.

Система не використовує алгоритми штучного інтелекту або машинного навчання. Формування повідомлень реалізовано за допомогою набору шаблонів та правил, які враховують поточний контекст роботи застосунку.

Модуль DJService взаємодіє з QueueService та PlayerService і генерує повідомлення у таких випадках:

- початок нового треку;
- завершення черги;
- активація Automix;
- нічний режим відтворення;
- часте пропускання композицій користувачами.

Для реалізації різних стилів повідомлень використовується підхід, близький до патерну Strategy. Користувач може обрати один із доступних режимів:

- chill;
- energetic;
- funny.

Кожен режим використовує окремий набір шаблонів повідомлень.

Таблиця 3.8. Основні режими DJ Persona

Режим	**Особливості повідомлень**
chill	Спокійні нейтральні повідомлення
energetic	Активні та динамічні повідомлення
funny	Повідомлення з жартівливими фразами

Для підвищення атмосферності застосунку реалізовано врахування часу доби. Наприклад, у нічний період система може автоматично формувати повідомлення про активацію нічного режиму відтворення.

Використання системи контекстних повідомлень дозволяє зробити взаємодію з ботом більш інтерактивною без ускладнення основної логіки застосунку.

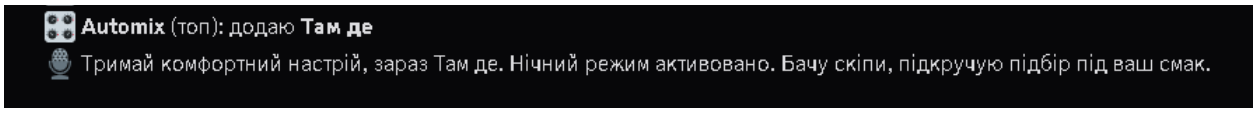

Рис. 3.8. Приклад повідомлень системи DJ Persona

На рисунку 3.8 наведено приклад роботи системи контекстних повідомлень DJ Persona під час відтворення аудіоконтенту.

3.3.6 Реалізація інтерфейсів налаштування

Для налаштування параметрів роботи застосунку використовуються інтерактивні елементи Discord API: модальні вікна (discord.ui.Modal) та випадаючі списки (discord.ui.Select).

Такий підхід дозволяє реалізувати введення параметрів без використання текстових команд складного формату.

За допомогою інтерфейсу налаштувань користувач може:

- змінювати рівень гучності;
- активувати або вимикати Automix;
- обирати стиль DJ Persona;
- змінювати окремі параметри плеєра.

Після введення значень система виконує:

- перевірку типу даних;
- валідацію допустимого діапазону;
- оновлення налаштувань у базі даних.

У разі некоректного введення користувач отримує повідомлення про помилку через Discord interaction API.

Таблиця 3.9. Основні елементи меню налаштувань

Елемент	**Призначення**
Volume Modal	Зміна гучності
Automix Toggle	Увімкнення/вимкнення Automix
DJ Persona Select	Вибір режиму повідомлень
Queue Controls	Керування чергою

Рис. 3.9. Меню налаштувань /mix

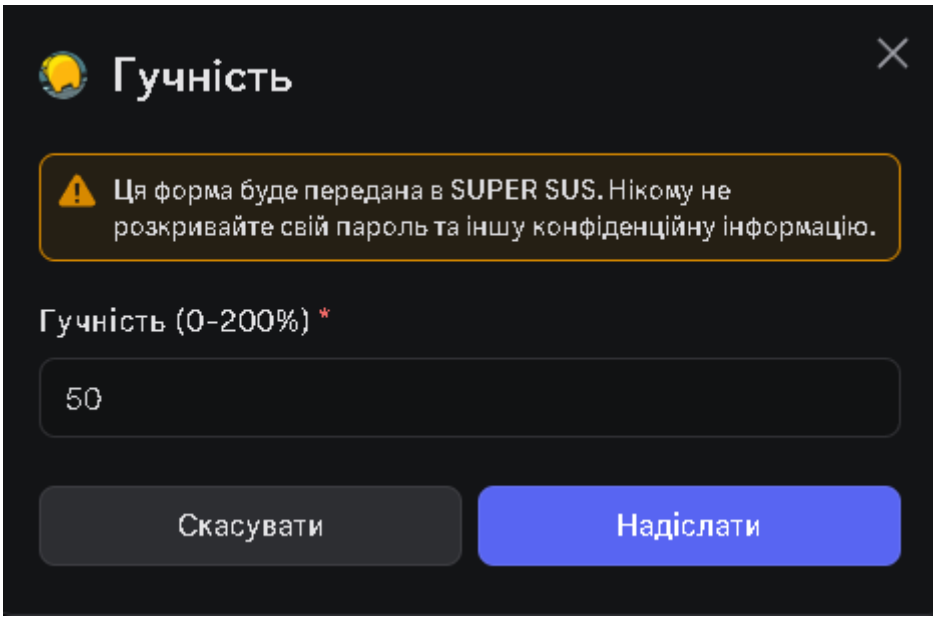

Рис. 3.10. Модальне вікно зміни гучності

На рисунках 3.9–3.10 наведено приклади роботи інтерактивних елементів налаштування застосунку. Використання модальних вікон та Select-компонентів дозволяє реалізувати введення параметрів без перевантаження текстового каналу службовими командами.

3.3.7 Тестування системи

Для перевірки працездатності програмного продукту використовувався фреймворк pytest. Автоматизоване тестування охоплює основні модулі бізнес-логіки застосунку.

Під час тестування перевірялися:

- коректність додавання треків до черги;
- робота механізму Auto-Resume;
- обробка помилок;
- коректність роботи Automix;
- взаємодія з базою даних SQLite;
- обробка interaction-подій.

Для автоматизації запуску тестів та перевірки збірки використовується GitHub Actions.

Таблиця 3.10. Основні сценарії тестування

Модуль	**Перевірка**
QueueService	Додавання та видалення треків
PlayerService	Запуск і зупинка відтворення
HistoryService	Збереження історії
AutoResume	Відновлення стану черги
UI Components	Обробка interaction-подій
AutomixService	Автоматичне додавання треків

Автоматизоване тестування дозволило виявити помилки логіки відтворення, перевірити коректність асинхронної взаємодії компонентів та забезпечити стабільність роботи застосунку.

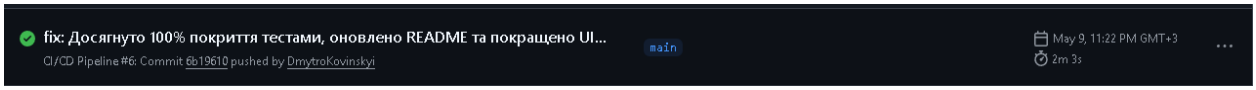

Рис. 3.11. Результат проходження тестів у GitHub Actions

На рисунку 3.11 наведено приклад автоматичного запуску тестів та перевірки збірки проєкту за допомогою GitHub Actions.

3.4 Керівництво користувача

У даному підрозділі наведено інструкції щодо використання основних функцій музичного Discord-бота. Взаємодія із системою здійснюється через Slash-команди та інтерактивні елементи Discord API.

3.4.1 Запуск відтворення музики

Для запуску відтворення користувачеві необхідно:

- Підключитися до голосового каналу Discord.
- Викликати Slash-команду /play.

- Ввести текстовий запит або URL-посилання на аудіоконтент.

Після обробки запиту система:

- виконує пошук аудіоджерела;
- формує Embed-повідомлення;
- додає трек до черги;
- запускає відтворення.

У разі успішного запуску відтворення в текстовому каналі автоматично відображається інтерактивний плеєр із кнопками керування.

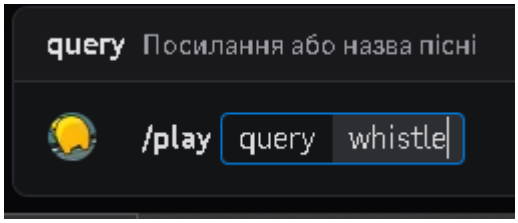

Рис. 3.12. Використання Slash-команди /play

На рисунку 3.12 наведено приклад запуску відтворення музики за допомогою Slash-команди /play.

3.4.2 Керування відтворенням

Після запуску треку користувач може взаємодіяти з плеєром через кнопковий інтерфейс.

Основні кнопки керування:

- Pause/Resume — призупинення та продовження відтворення;
- Skip — перехід до наступного треку;
- Queue — перегляд поточної черги;
- Volume — зміна гучності;
- Mix — відкриття меню налаштувань.

Після натискання кнопок інтерфейс автоматично оновлюється та відображає актуальний стан плеєра.

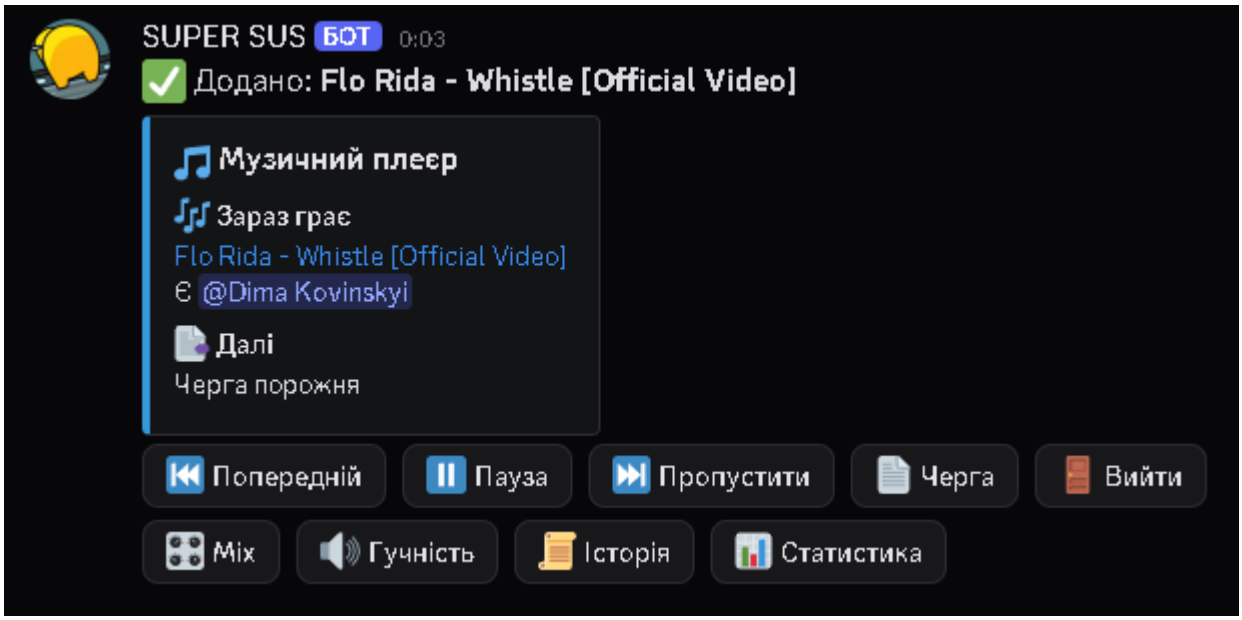

Рис. 3.13. Інтерактивний плеєр під час відтворення

На рисунку 3.13 наведено приклад роботи інтерактивного плеєра під час відтворення музики.

3.4.3 Обробка помилок та некоректних дій користувача

Під час роботи застосунок виконує перевірку коректності введених даних та поточного стану системи.

У разі виникнення помилок користувач отримує повідомлення про:

- відсутність підключення до голосового каналу;
- некоректне URL-посилання;
- порожню чергу;
- недоступність аудіоджерела;
- помилки взаємодії з Discord API.

Для службових повідомлень використовуються ephemeral-відповіді Discord API, які бачить лише користувач, що виконав дію.

Таблиця 3.11. Основні повідомлення про помилки

Ситуація	**Повідомлення**
Користувач не у voice channel	Необхідно підключитися до голосового каналу
Некоректне посилання	Неможливо отримати аудіоджерело
Порожня черга	Черга відтворення порожня
Втрата з’єднання	Спроба автоматичного перепідключення

ВИСНОВКИ ДО РОЗДІЛУ 3

У третьому розділі розглянуто програмну реалізацію музичного Discord-бота, описано архітектуру модулів, реалізацію бізнес-логіки та інтерактивного користувацького інтерфейсу.

У межах розділу обґрунтовано вибір засобів розробки, описано механізми потокового відтворення аудіо, автоматичного продовження відтворення (Automix), збереження стану черги (Auto-Resume) та систему контекстних повідомлень DJ Persona.

Окрему увагу приділено реалізації інтерактивного інтерфейсу на основі Slash-команд, кнопок та модальних вікон Discord API. Також розглянуто механізми тестування застосунку та контейнеризованого розгортання за допомогою Docker.

Розроблене програмне рішення забезпечує стабільне відтворення аудіоконтенту, підтримує інтерактивне керування та може бути використане як self-hosted музичний Discord-бот.

У межах виконання кваліфікаційної роботи було реалізовано музичного Discord-бота для потокового відтворення аудіоконтенту та інтерактивної взаємодії користувачів із системою. У процесі роботи проведено аналіз предметної області, досліджено існуючі програмні аналоги та визначено основні функціональні вимоги до програмного продукту.

У роботі обґрунтовано вибір технологічного стеку для реалізації застосунку, зокрема використання мови програмування Python, бібліотеки discord.py, засобів обробки аудіо FFmpeg та бібліотеки yt-dlp для роботи з аудіоджерелами. Для збереження стану системи та історії прослуховувань використано локальну базу даних SQLite з асинхронним драйвером aiosqlite.

У результаті виконання роботи спроєктовано архітектуру програмного застосунку з розділенням рівня представлення, бізнес-логіки та доступу до даних. Реалізовано основні функціональні модулі системи:

керування відтворенням аудіо;

- роботу з чергою;
- автоматичне продовження відтворення (Automix);
- механізм відновлення стану плеєра після перезапуску (Auto-Resume);
- систему збереження історії прослуховувань;
- інтерактивний користувацький інтерфейс на основі Slash-команд та Discord UI-компонентів.

Окрему увагу приділено реалізації інтерактивного інтерфейсу користувача. Для взаємодії із системою використано Slash-команди, кнопки, модальні вікна та випадаючі списки Discord API. Це дозволило спростити керування плеєром та забезпечити більш зручний користувацький досвід у порівнянні з класичними текстовими командами.

У застосунку реалізовано систему контекстних повідомлень DJ Persona, яка формує повідомлення залежно від поточного стану відтворення та окремих подій під час роботи плеєра. Також реалізовано механізм Auto-Resume, який забезпечує відновлення черги та повторне підключення до голосового каналу після перезапуску застосунку.

Для забезпечення стабільної роботи системи реалізовано механізми обробки помилок, перевірки користувацького вводу та очищення FFmpeg-

процесів. Автоматизоване тестування основних модулів застосунку виконувалося за допомогою фреймворку pytest, а автоматична перевірка збірки реалізована через GitHub Actions.

Для спрощення розгортання застосунку використано Docker та Docker Compose, що дозволяє запускати програмний продукт у контейнеризованому середовищі незалежно від операційної системи.

Практична цінність роботи полягає у створенні self-hosted програмного рішення для потокового відтворення аудіоконтенту в Discord-середовищі. Розроблений бот може бути використаний як основа для подальшого розвитку функціональності, зокрема реалізації веб-інтерфейсу адміністрування, розширення системи рекомендацій та інтеграції з додатковими джерелами аудіоконтенту.

Подальший розвиток системи може бути спрямований на:

1. впровадження веб-інтерфейсу керування ботом із використанням FastAPI або React;
2. розширення підтримки зовнішніх медіасервісів та інтеграцію з додатковими API;
3. реалізацію додаткових аудіоефектів та засобів цифрової обробки звуку;
4. впровадження систем моніторингу та аналізу продуктивності застосунку.

Таким чином, у кваліфікаційній роботі досягнуто поставленої мети — розроблено музичного Discord-бота, який забезпечує потокове відтворення аудіоконтенту, підтримує інтерактивне керування та реалізує механізми автоматизації роботи плеєра. Розроблений застосунок може використовуватися як self-hosted рішення для Discord-спільнот та слугувати основою для подальшого розширення функціональності системи.

ЗАГАЛЬНІ ВИСНОВКИ

У межах виконання кваліфікаційної роботи було реалізовано музичного Discord-бота для потокового відтворення аудіоконтенту та інтерактивної взаємодії користувачів із системою. У процесі роботи проведено аналіз предметної області, досліджено існуючі програмні аналоги та визначено основні функціональні вимоги до програмного продукту.

У роботі обґрунтовано вибір технологічного стеку для реалізації застосунку, зокрема використання мови програмування Python, бібліотеки discord.py, засобів обробки аудіо FFmpeg та бібліотеки yt-dlp для роботи з аудіоджерелами. Для збереження стану системи та історії прослуховувань використано локальну базу даних SQLite з асинхронним драйвером aiosqlite.

У результаті виконання роботи спроєктовано архітектуру програмного застосунку з розділенням рівня представлення, бізнес-логіки та доступу до даних. Реалізовано основні функціональні модулі системи:

керування відтворенням аудіо;

- роботу з чергою;
- автоматичне продовження відтворення (Automix);
- механізм відновлення стану плеєра після перезапуску (Auto-Resume);
- систему збереження історії прослуховувань;
- інтерактивний користувацький інтерфейс на основі Slash-команд та Discord UI-компонентів.

Окрему увагу приділено реалізації інтерактивного інтерфейсу користувача. Для взаємодії із системою використано Slash-команди, кнопки, модальні вікна та випадаючі списки Discord API. Це дозволило спростити керування плеєром та забезпечити більш зручний користувацький досвід у порівнянні з класичними текстовими командами.

У застосунку реалізовано систему контекстних повідомлень DJ Persona, яка формує повідомлення залежно від поточного стану відтворення та окремих подій під час роботи плеєра. Також реалізовано механізм Auto-Resume, який забезпечує відновлення черги та повторне підключення до голосового каналу після перезапуску застосунку.

Для забезпечення стабільної роботи системи реалізовано механізми обробки помилок, перевірки користувацького вводу та очищення FFmpeg-процесів. Автоматизоване тестування основних модулів застосунку виконувалося за допомогою фреймворку pytest, а автоматична перевірка збірки реалізована через GitHub Actions.

Для спрощення розгортання застосунку використано Docker та Docker Compose, що дозволяє запускати програмний продукт у контейнеризованому середовищі незалежно від операційної системи.

Практична цінність роботи полягає у створенні self-hosted програмного рішення для потокового відтворення аудіоконтенту в Discord-середовищі. Розроблений бот може бути використаний як основа для подальшого розвитку функціональності, зокрема реалізації веб-інтерфейсу адміністрування, розширення системи рекомендацій та інтеграції з додатковими джерелами аудіоконтенту.

Подальший розвиток системи може бути спрямований на:

1. впровадження веб-інтерфейсу керування ботом із використанням FastAPI або React;
2. розширення підтримки зовнішніх медіасервісів та інтеграцію з додатковими API;
3. реалізацію додаткових аудіоефектів та засобів цифрової обробки звуку;
4. впровадження систем моніторингу та аналізу продуктивності застосунку.

Таким чином, у кваліфікаційній роботі досягнуто поставленої мети — розроблено музичного Discord-бота, який забезпечує потокове відтворення аудіоконтенту, підтримує інтерактивне керування та реалізує механізми автоматизації роботи плеєра. Розроблений застосунок може використовуватися як self-hosted рішення для Discord-спільнот та слугувати основою для подальшого розширення функціональності системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційна документація мови програмування Python (версія 3.12). URL: https://docs.python.org/3/ (дата звернення: 09.05.2026).

2. Бібліотека discord.py — API Wrapper для Discord. URL: https://discordpy.readthedocs.io/en/stable/ (дата звернення: 09.05.2026).

3. Discord Developer Portal — Офіційна документація розробника Discord. URL: https://discord.com/developers/docs/intro (дата звернення: 09.05.2026).

4. yt-dlp — Multimedia download tool. URL: https://github.com/yt-dlp/yt-dlp (дата звернення: 09.05.2026).

5. Fmpeg Documentation. URL: https://ffmpeg.org/documentation.html (дата звернення: 09.05.2026).

6. aiosqlite — Асинхронний міст до модуля SQLite3 у Python. URL: https://aiosqlite.omnilib.dev/en/stable/ (дата звернення: 09.05.2026).

7. SQLite Official Documentation. URL: https://www.sqlite.org/docs.html (дата звернення: 09.05.2026).

8. pytest: framework makes it easy to write small tests. URL: https://docs.pytest.org/en/latest/ (дата звернення: 09.05.2026).

9. asyncio — Asynchronous I/O, event loop, coroutines and tasks. URL: https://docs.python.org/3/library/asyncio.html (дата звернення: 09.05.2026).

10. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 09.05.2026).

11. Docker Compose Overview. URL: https://docs.docker.com/compose/ (дата звернення: 09.05.2026).

12. GitHub Actions Documentation — Continuous Integration and Deployment. URL: https://docs.github.com/en/actions (дата звернення: 09.05.2026).

13. Discord Interactions and Application Commands (Slash Commands). URL:

https://discord.com/developers/docs/interactions/application-commands (дата звернення: 09.05.2026).

14. Патерни проєктування (Design Patterns) в Python. URL: https://refactoring.guru/uk/design-patterns/python (дата звернення: 09.05.2026).

15. Jockie Music — Discord music bot. URL: https://www.jockiemusic.com/ (дата звернення: 09.05.2026).

16. MEE6 — Discord bot platform. URL: https://mee6.xyz/ (дата звернення: 09.05.2026).

17. Hydra Bot — Discord music bot. URL: https://hydra.bot/ (дата звернення: 09.05.2026).

18. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Boston: Prentice Hall, 2017. — 432 p. (дата звернення: 09.05.2026).

19. Fowler M. Patterns of Enterprise Application Architecture. — Addison-Wesley Professional, 2002. — 533 p

20. Ramalho L. Fluent Python. — 2nd Edition. — O'Reilly Media, 2022. — 1014 p.

21. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1994. — 395 p.

22. Репозиторій програмного проєкту Discord Music Bot. URL: https://github.com/Alxcgs/discord-music-bot

ДОДАТКИ

Додаток А. Скріншоти інтерфейсу

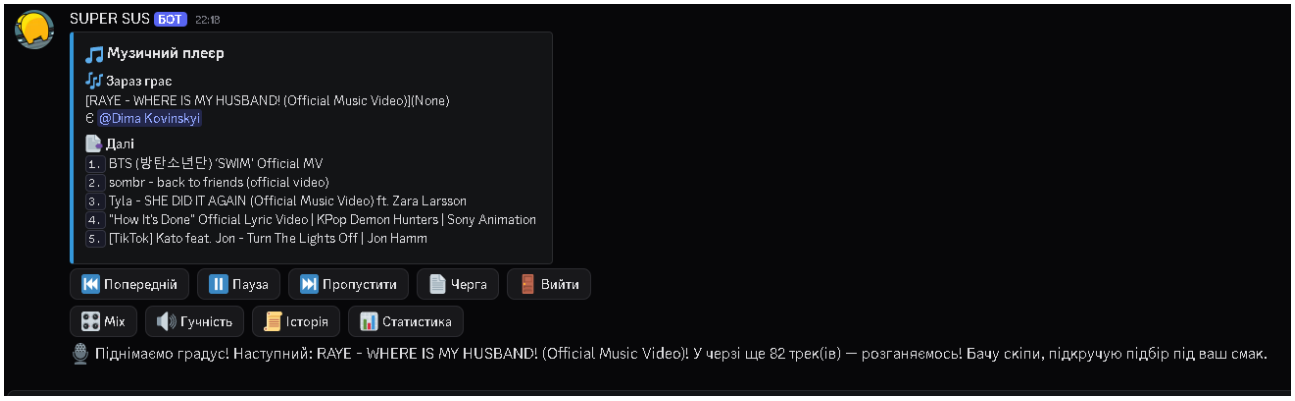

Рис.1. Основне вікно бота з Діджеєм

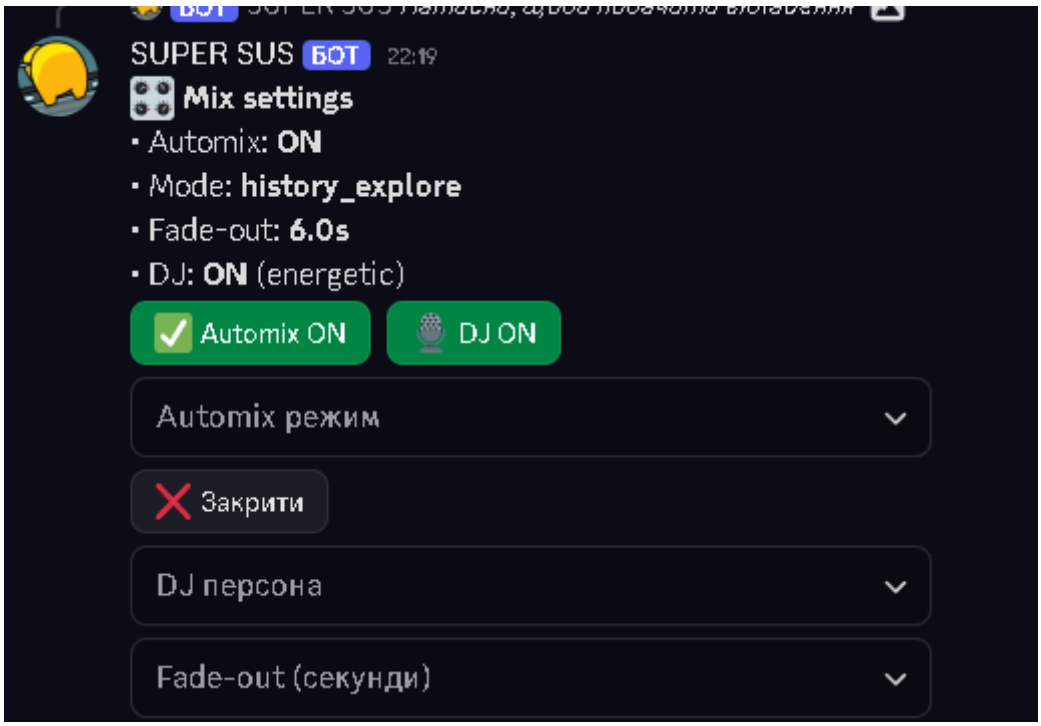

Рис.2.Діалогове вікно налаштувань Automix,DJ,та Fade-out

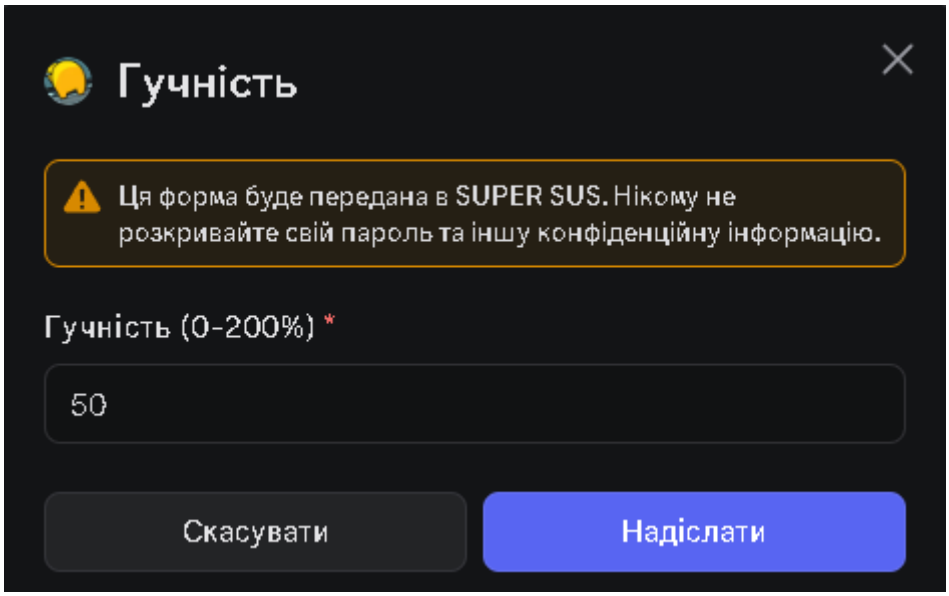

Рис.3. Діалогове вікно регулювання гучності відтворюваного аудіо

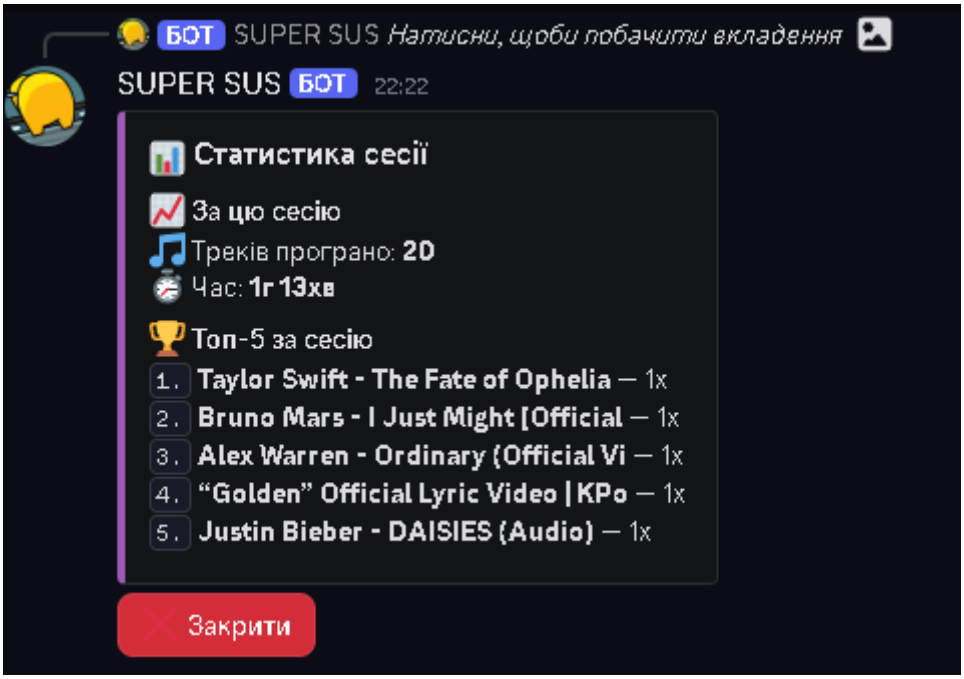

Рис.4. Вікно статистики відтворених треків за одну сесію

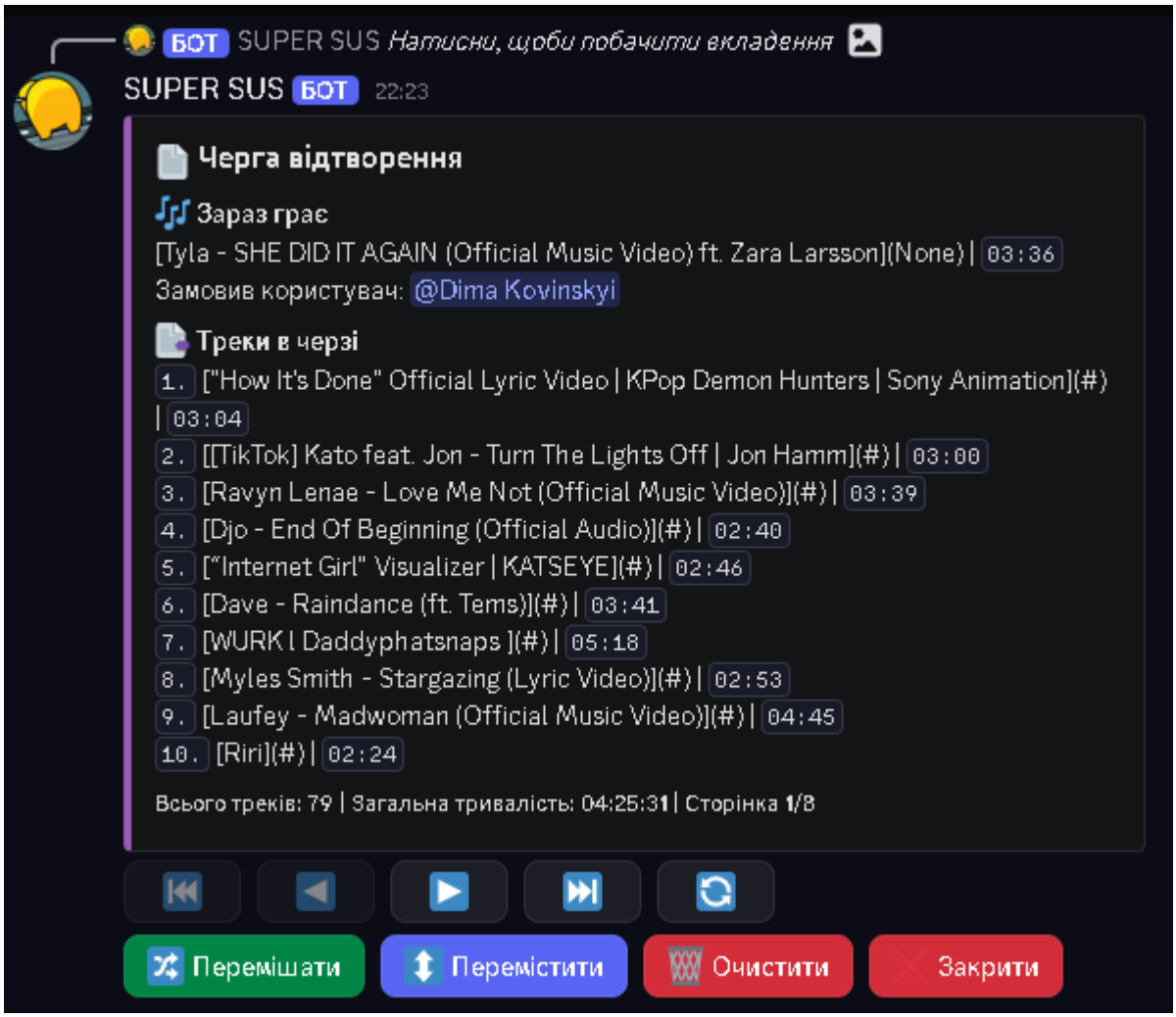

Рис.5. Вікно черги треків, з можливістю перемішування та переміщення треків з однієї позиції на іншу, також є можливість очистити чергу, пагінація між сторінками треків (якщо черга велика)

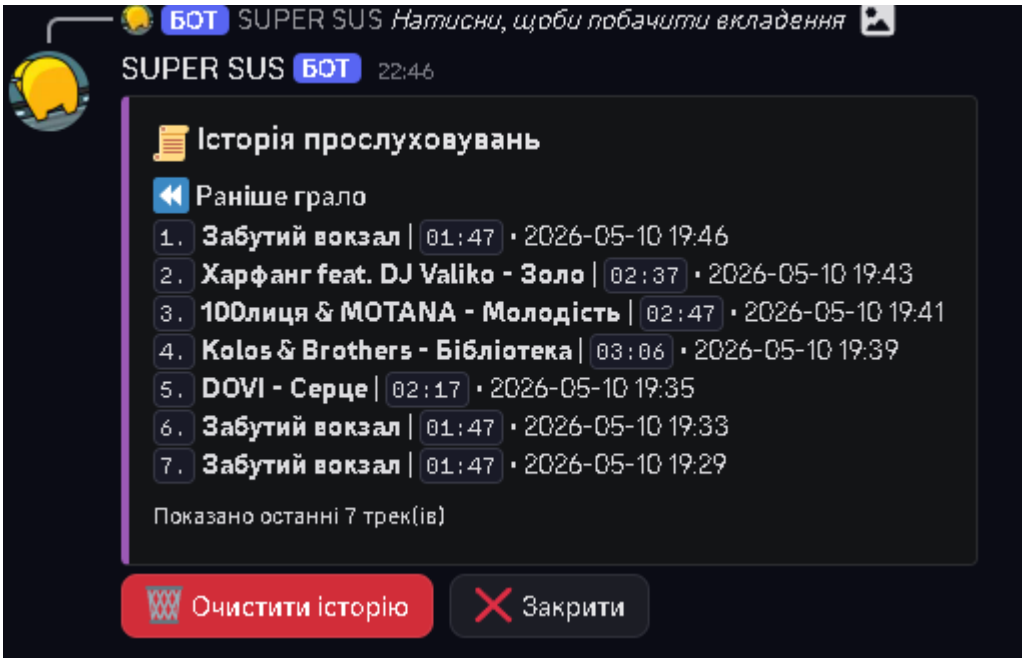

Рис.6. Вікно історії відтворених треків