Галицький фаховий коледж імені В’ячеслава Чорновола
відділення комп’ютерних технологій
циклова комісія інформатики та комп’ютерних дисциплін

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач відділення
комп’ютерних технологій
Наталія СТЕФУРАК /_____/
(підпис)
«___»_____2025р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи
освітньо-професійного ступеня «Фаховий молодший бакалавр»
зі спеціальності 122 «Комп’ютерні науки»

на тему: «Програмний засіб відтворення аудіофайлів
з розподіленого захищеного сховища»

Студент групи КН-41	Михайло МУЗИКА	_____ (підпис)
Керівник роботи	Степан ІВАСЬЄВ	_____ (підпис)
Консультанти:		
з техніко-економічного обґрунтування	Любов МЕЛЕНЧУК	_____ (підпис)
нормоконтролер	Оксана СИРОТЮК	_____ (підпис)

Тернопіль – 2025

Галицький фаховий коледж імені В’ячеслава Чорновола
відділення комп’ютерних технологій
циклова комісія інформатики та комп’ютерних дисциплін

ЗАТВЕРДЖУЮ
Завідувач відділення
комп’ютерних технологій
Наталія СТЕФУРАК /_____/
(підпис)
«___»_____20__р.

ЗАВДАННЯ

на кваліфікаційну роботу

на здобуття освітньо-професійного ступеня «фаховий молодший бакалавр»

студенту Музиці Михайлу Андрійовичу

1. Тема роботи Програмний засіб відтворення аудіофайлів з розподіленого захищеного сховища

затверджено наказом по коледжу від “25” листопада 2024 р., № 253а-н

2. Термін здачі студентом завершеного проєкту “___” _____ 20__ р.

3. Вихідні дані до проєкту Результати аналізу сучасних музичних плеєрів та їх обмежень, дослідження алгоритмів аудіообробки, вивчення протоколів безпечного P2P-обміну даними

4. Перелік питань, які повинні бути розроблені в проєкті:

а) основна частина Аналіз предметної області та постановка завдань. Проєктування архітектури застосунку. Реалізація та тестування системи

б) техніко-економічне обґрунтування Аналіз ринку музичних додатків Розрахунок витрат на розробку. Обґрунтування економічної доцільності

5. Перелік графічного матеріалу Схема архітектури музичного плеєра, Діаграма класів ядра програми, Діаграма P2P-взаємодії пристроїв, Схема роботи аудіоефектів

6. Консультанти проєкту _____

Розділ	Консультанти	Підпис, дата	
		Завдання видано	Завдання прийнято
З техніко-економічного обґрунтування	_____Меленчук Л. І._____ (вчена ступінь, звання П.І.Б _____ (консультант)		

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

№ п/п	Найменування етапу	Терміни	
		початку	завершення
1.	Вибір теми, аналіз вимог до кваліфікаційної роботи	20.11.2024	24.11.2024
2.	Огляд наявних рішень, написання аналітичного розділу	25.11.2024	04.12.2024
3.	Розробка функціональних та нефункціональних вимог реалізації застосунку	05.12.2024	07.12.2024
4.	Дослідження технологій реалізації	08.12.2024	12.12.2024
5.	Встановлення бібліотек та налаштування середовища розробки	13.12.2024	14.12.2024
6.	Проєктування системи	15.12.2024	26.12.2024
7.	Реалізація застосунку, аудіоефектів та мережевої взаємодії	27.12.2024	02.05.2025
8.	Тестування застосунку, написання розділу реалізації	03.05.2025	09.05.2025
9.	Написання техніко-економічного обґрунтування	10.05.2025	11.05.2025
10.	Оформлення пояснювальної записки	12.05.2025	13.06.2025
11.	Попередній захист кваліфікаційної роботи	13.06.2025	13.06.2025
12.	Підготовка до захисту кваліфікаційної роботи	14.06.2025	24.06.2025
13.	Захист кваліфікаційної роботи	25.06.2025	25.06.2025

7. Дата видачі “___” _____ 20__ р. Керівник _____/

Завдання прийняв до виконання _____/ Музика М. А.

Реферат

Кваліфікаційна робота. Програмний засіб відтворення аудіофайлів з розподіленого захищеного сховища. Михайло Музика. Галицький фаховий коледж імені В’ячеслава Чорновола. Відділення комп’ютерних технологій. 102с., 23 рисунків, 6 додатків.

Об'єкт дослідження – процес розробки програмного засобу відтворення аудіофайлів з розподіленого захищеного сховища.

Мета роботи полягає у створенні програмного засобу з використанням Qt/QML, C++ та сучасних аудіотехнологій, що забезпечує відтворення аудіофайлів, керування плейлистами, реалізацію аудіоефектів, адаптивний інтерфейс для різних пристроїв та мережевий обмін плейлистами через P2P-протокол.

Застосунок повинен забезпечувати високоякісне відтворення аудіо з інтуїтивним інтерфейсом та кросплатформовою підтримкою. Необхідно реалізувати систему захищеного P2P-обміну плейлистами через локальну мережу з використанням TLS-шифрування.

Результатом розробки є функціональний кросплатформний музичний плеєр з безпечним P2P-обміном плейлистами у локальній мережі через TLS-шифрування, зручним інтерфейсом користувача та відкритим вихідним кодом.

Ключові слова: AUDIO PLAYER, AUDIO PROCESSING, AUDIO EFFECTS, CROSS-PLATFORM, P2P, TLS, ENCRYPTION, ADAPTIVE UI, OPEN SOURCE, LOCAL NETWORK, AUDIO PLAYBACK, Qt, QML, C++, FFMPEG.

Abstract

Qualification work. Software for playing audio files from a distributed secure storage. Mykhailo Muzyka. Halytsky Applied College named after Vyacheslav Chornovil. Department of Computer Technologies. 102 pages, 23 figures, 6 appendices.

The object of research is the process of developing software for playing audio files from a distributed secure storage.

The aim of the work is to create software using Qt/QML, C++, and modern audio technologies that provides audio file playback, playlist management, audio effects implementation, an adaptive interface for different devices, and network playlist sharing via the P2P protocol.

The application should provide high-quality audio playback with an intuitive interface and cross-platform support. It is necessary to implement a secure P2P playlist exchange system over a local network using TLS encryption.

The result of the development is a functional cross-platform music player with secure P2P playlist sharing over a local network using TLS encryption, a user-friendly interface, and open source code.

Keywords: AUDIO PLAYER, AUDIO PROCESSING, AUDIO EFFECTS, CROSS-PLATFORM, P2P, TLS, ENCRYPTION, ADAPTIVE UI, OPEN SOURCE, LOCAL NETWORK, AUDIO PLAYBACK, Qt, QML, C++, FFMPEG.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОС – Операційна система

AES – Advanced Encryption Standard

API – Application Programming Interface

BSD – Berkeley Software Distribution

CMake – Cross-platform Make

FFmpeg – Fast Forward MPEG (некоректна розшифровка, див. нижче)

GUI – Graphical User Interface

ID3v2 – IDentify an MP3 v2

JSON – JavaScript Object Notation

MPEG – Moving Picture Experts Group

NAT – Network Address Translation

P2P – Peer-to-Peer

QML – Qt Meta-object Language

SHA – Secure Hash Algorithm

TCP – Transmission Control Protocol

TLS – Transport Layer Security

UDP – User Datagram Protocol

UML – Unified Modeling Languag

UPF – Universal Playlist Format

XDG – X Desktop Group

ВСТУП

У сучасному цифровому світі аудіоконтент займає провідне місце серед медіаданих, що споживаються користувачами щодня. Понад 80% інтернет-трафіку припадає на мультимедійний контент, значну частину якого становлять аудіофайли. За останні роки обсяг цифрового аудіоконтенту зростає експоненційно, що створює нові виклики для систем зберігання та управління медіаданими. Традиційні підходи до зберігання та відтворення аудіоконтенту характеризуються централізованою архітектурою, що створює ризики втрати даних, обмежує доступність контенту для користувачів різних платформ та не забезпечує належної масштабованості.

Розподілені системи зберігання даних набувають все більшої популярності завдяки підвищеній надійності, масштабованості та відмовостійкості. Використання принципів децентралізації дозволяє уникнути проблем єдиної точки відмови, забезпечити географічну розподіленість даних та підвищити загальну продуктивність системи. Однак існуючі рішення для роботи з аудіоконтентом у розподілених сховищах часто не забезпечують належного рівня безпеки та зручності використання. Більшість сучасних аудіопрогравачів розроблені для роботи з локальними файлами або централізованими сервісами, що не відповідає потребам користувачів у високодоступному та безпечному зберіганні контенту.

Відсутність комплексних програмних засобів, які б поєднували можливості відтворення аудіофайлів з ефективним управлінням розподіленим захищеним сховищем, створює потребу в розробці спеціалізованих рішень. Особливо гостро стоїть питання забезпечення конфіденційності та цілісності аудіоданих у розподіленому середовищі, де інформація зберігається на множині незалежних вузлів.

Сучасні криптографічні методи та технології розподілених систем дозволяють створювати безпечні та надійні сховища даних з

децентралізованою архітектурою. Інтеграція таких технологій з функціоналом аудіопрогравача може забезпечити користувачам безперебійний доступ до контенту при збереженні високого рівня захисту інформації. Використання алгоритмів шифрування, хешування та цифрових підписів у поєднанні з механізмами розподіленого консенсусу створює основу для побудови довірчих систем управління аудіоконтентом.

Метою роботи є розробка програмного засобу відтворення аудіофайлів з розподіленого захищеного сховища з підтримкою кросплатформності та сучасних стандартів інформаційної безпеки. Об'єктом дослідження виступають процеси зберігання, передачі та відтворення аудіоконтенту в розподілених захищених системах. Предметом дослідження є методи та технології інтеграції функціоналу аудіопрогравача з розподіленими сховищами даних.

Практичне значення результатів полягає у створенні програмного рішення, що забезпечить надійний та безпечний доступ до аудіоконтенту незалежно від місця розташування даних та типу пристрою користувача, а також дозволить ефективно управляти великими обсягами розподіленого аудіоконтенту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАНЬ

1.1 Опис предметної області

Аудіофайли є одним із найпоширеніших форматів медіаданих у мережі завдяки універсальності та оптимальному співвідношенню якості до розміру. Традиційні аудіопрогравачі зазвичай забезпечують лише базовий функціонал відтворення, без розширених інструментів для індивідуального налаштування звуку чи обміну контентом.

Пропоноване програмне рішення покликане забезпечити користувачів засобом для відтворення аудіофайлів, в якому передбачається реалізація зручного управління списками відтворення, а також швидкого обміну окремими аудіофайлами чи цілими списками відтворення через мережу. Високий рівень безпеки буде забезпечено завдяки інтеграції сучасних криптографічних методів шифрування даних.

Під час розробки особливу увагу буде приділено кросплатформності, оскільки сучасні користувачі очікують однаково зручного досвіду використання незалежно від операційної системи чи типу пристрою. Програмне рішення надаватиме підтримку Windows, Linux, macOS та мобільних платформ зі збереженням повної функціональності. Це буде досягнуто завдяки застосуванню фреймворку Qt6 1, що дозволить розширити аудиторію та спростить процес оновлення та підтримки застосунку.

Програмне рішення спрямоване на вирішення проблем безпечного зберігання та передачі аудіоконтенту. Для цього передбачається використання криптографічних алгоритмів, які запобігатимуть несанкціонованому доступу. Зручність прослуховування буде забезпечено завдяки високоточному еквалайзеру та сучасному інтуїтивному інтерфейсу. Крім того, програмний код буде розповсюджуватися за ліцензією «GNU GENERAL PUBLIC LICENSE v3» 2, що дасть змогу спільноті вільних розробників впроваджувати

нові функції, покращувати наявний функціонал та підтримувати актуальність системи.

1.2 Огляд наявних рішень

На ринку програмного забезпечення для програвання аудіоконтенту можна виокремити 3 популярних рішення: Groove, Audacious та MusicBee. Ці продукти демонструють різні підходи до реалізації функціоналу, організації інтерфейсу та забезпечення взаємодії з користувачем. У цьому розділі буде проведено аналіз ключових характеристик цих рішень: рівня безпеки даних, можливостей налаштування звуку, підтримки кросплатформності, та відкритості архітектури для модифікацій.

Сучасні музичні програвачі розвиваються в умовах постійно зростаючих вимог користувачів до якості звуку, зручності використання та функціональних можливостей. Традиційні підходи до проєктування аудіоплеєрів часто зосереджуються на базовому відтворенні файлів, залишаючи поза увагою потреби в соціальній взаємодії, обміні контентом та адаптації до різноманітних використовуваних сценаріїв. Водночас користувачі все частіше потребують інструментів, які поєднують високоякісне відтворення з можливостями персоналізації, гнучкого налаштування та безпечного обміну музичними колекціями.

Аналіз допоможе виявлено переваги та обмеження наявних продуктів, що дозволить сформулювати вимоги до розробки. Особливу увагу буде приділено аспектам, які залишаються недостатньо втіленими в сучасних аналогах: обміну контентом, гнучкості персоналізації та універсальності використання на різних платформах. Отримані під час аналізу дані стануть основою для подальшого проєктування програмного рішення та його розробки.

1.2.1 Аналіз аудіопрогравача Groove

Groove — це стандартний програвач який постачається із Windows 10 та 11, він сильно інтегрований із сервісами Microsoft. Його головна перевага — синхронізація з OneDrive та Xbox, що забезпечує доступ до музики на різних Windows/Xbox пристроях компанії.

Він поширюється як вбудований компонент Windows 10/11 із закритим кодом, що обмежує його модифікацію сторонніми розробниками. Ліцензія передбачає використання програми лише в рамках умов Microsoft, без права розподілу або змін.

Загалом застосунок має мінімалістичний та комфортний дизайн зроблений в стилі Microsoft, має великі кнопки управління, монохромну палітру та чітку типографіку. Він пропонує інтуїтивно зрозумілий інтерфейс, де основні функції (створення списків відтворення, пошук, сортування) доступні за 1–2 кліки. Головне вікно застосунку зображено на рис. 1.1.

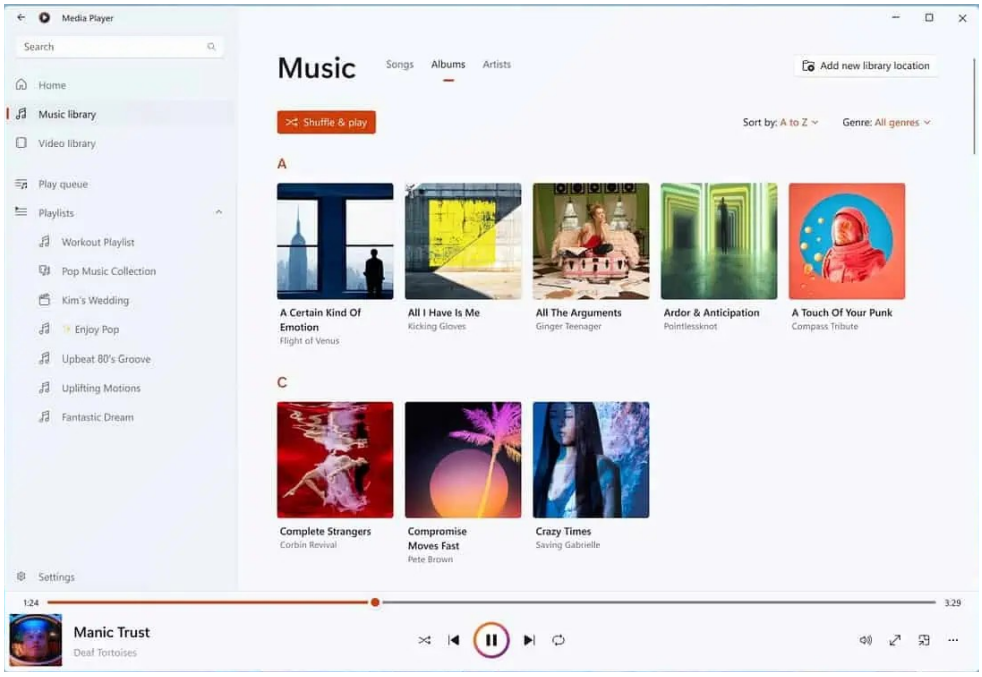

Рисунок 1.1 — Головне вікно аудіопрогравача Groove

Проте хоч інтерфейс і є інтуїтивно зрозумілим, він побудований таким чином, що користувач не може змінювати розташування елементів або

додавати власні модулі, що робить його менш привабливим для просунутих користувачів.

Основні елементи управління (відтворення, пауза, перемотка) розташовані зручно, що робить програму доступною для нових користувачів. Але створення динамічних списків відтворення або сортування треків за власними критеріями реалізовано менш очевидно, ніж у конкурентів, що може вимагати додаткового звикання.

1.2.2 Аналіз аудіопрогравача Audacious

Audacious4 — це легкий кросплатформний програвач, який орієнтований на мінімалізм та ефективність.

Інтерфейс Audacious нагадує класичні програвачі (на кшталт Winamp), з основним акцентом на компактність та швидкість роботи. Панелі управління містять базові кнопки (відтворення, пауза, гучність), але розширені функції (наприклад, редагування списків відтворення) приховані за контекстними меню, що може ускладнити роботу для нових користувачів. Головне вікно застосунку зображено на рис. 1.2.

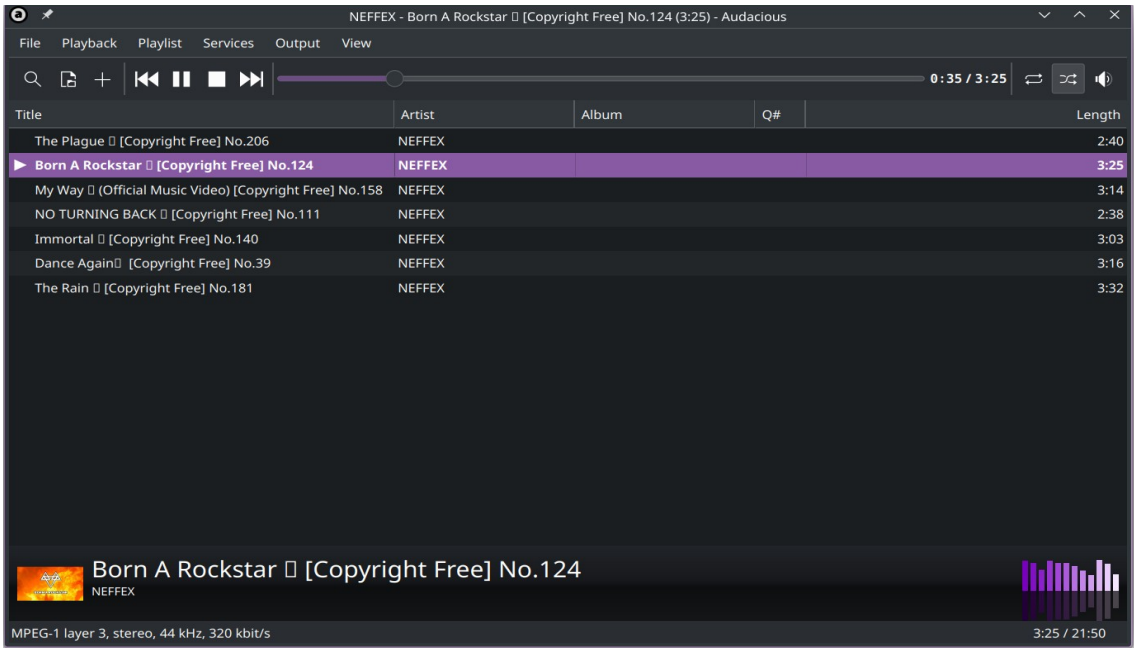

Рисунок 1.2 — Головне вікно аудіопрогравача Audacious

На відміну від Groove, тут доступне обмежене налаштування інтерфейсу користувача: можливість змінювати теми інтерфейсу, але розширені налаштування відсутні.

Audacious підтримує широкий спектр аудіоформатів і має базовий еквалайзер із ручним налаштуванням частот, що робить його гнучкішим за Groove.

Головна перевага — кросплатформність: програма працює на Windows, Linux, macOS, що робить її універсальним рішенням для користувачів з різними ОС. Однак функціонал обмежений базовими задачами: відтворення файлів та створення списків відтворення. Audacious не надає вбудованих інструментів для передачі файлів або спільного доступу до списків відтворення.

Audacious поширюється під ліцензією BSD5 з відкритим вихідним кодом, що дозволяє вільно модифікувати та розповсюджувати програму якщо дотримуються умови ліцензії. Це відкриває можливості для спільноти розробників, але на практиці активність спільноти досить низька, тому оновлення виходять рідко. На сам перед це пов’язано з недостатньо розвинутою документацією, що ускладнює розуміння коду та його модифікацію новими розробниками.

Отже, до головних недоліків можна віднести:

1) складність у використанні для новачків через "заховані" функції;
2) відсутність вбудованих механізмів обміну контентом;
3) мала активність спільноти розробників.

З полюсів можна виділити:

1) легкість та швидкодія;
2) кросплатформність (Windows, Linux, macOS);
3) відкритий код для модифікацій.

1.2.3 Аналіз аудіопрогравача Music Bee

MusicBee6 — спеціалізований програвач для Windows, орієнтований на управління великими музичними бібліотеками.

Інтерфейс MusicBee пропонує високий рівень налаштувань: користувач може змінювати розташування панелей, додавати віджети (наприклад, спектральний аналізатор) та вибирати теми оформлення. Головне вікно цього аудіопрогравача зображено на рис. 1.3.

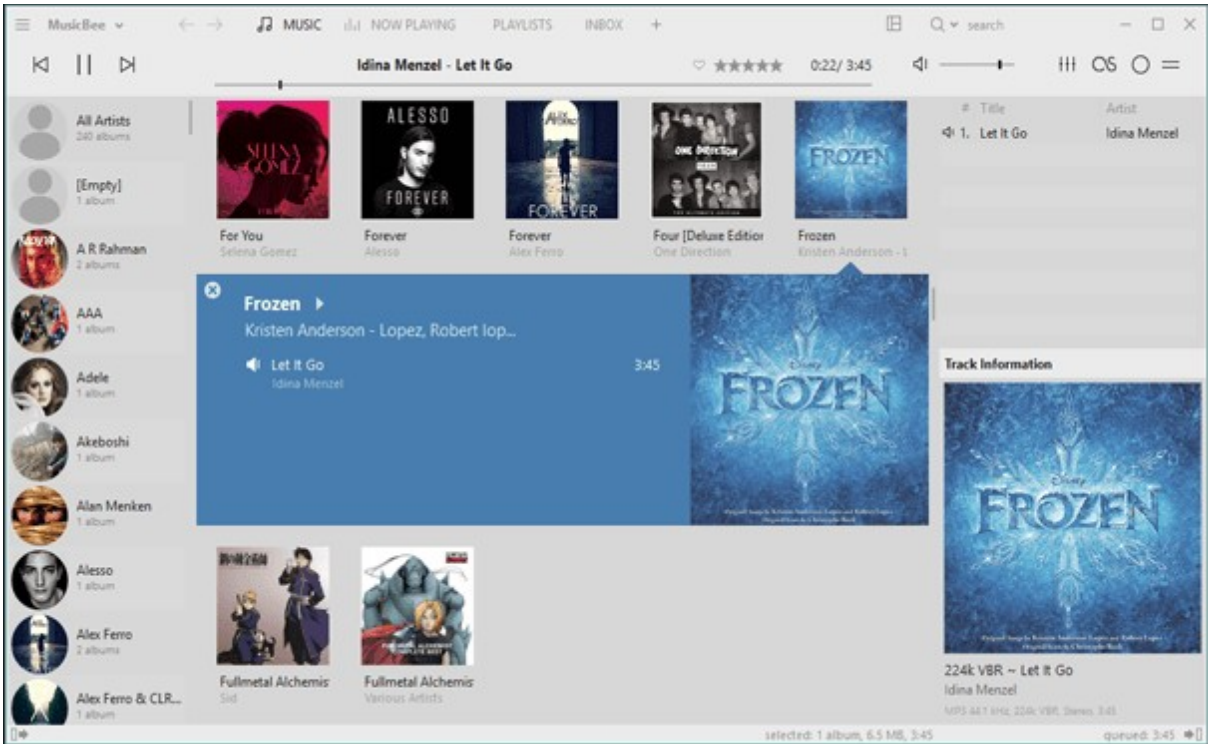

Рисунок 1.3 — Головне вікно аудіопрогравача MusicBee

Однак така гнучкість може ускладнювати процес використання для нових користувачів, налаштування потребують часу на знайомство з інтерфейсом. На відміну від мінімалістичного Audacious, MusicBee націлений на просунутих користувачів, які потребують детального контролю над бібліотекою.

Головною перевагою MusicBee є наявність потужних інструментів для роботи з аудіо, наприклад високоточний еквалайзер з готовими пресетами,

тоді як головним недоліком є те, що він сильно обмежений платформою Windows.

1.2.4 Порівняльний аналіз рішень

Для створення конкурентоспроможного програмного рішення було проведено порівняльний аналіз наявних аудіопрогравачів (Groove, Audacious, MusicBee) за ключовими критеріями: підтримкою форматів, кросплатформності, наявності функцій обміну даними, сучасним інтерфейсом, відкритістю коду та активною спільнотою розробників. Як результат в кожному з них було виявлено як переваги, так і недоліки. Результати представлені в таб. 1.1

Таблиця 1.1 — Порівняльний аналіз рішень

Критерії	Groove	Audacious	MusicBee
Підтримка форматів	Основні: wav, mp3	Популярні: flac, ogg, mp3 тощо✓	Найширша підтримка форматів
Кросплатформність	Лише Windows	Windows, Linux, macOS	Лише Windows
Обмін даними	Немає	Немає	Немає
Сучасний інтерфейс	Мінімалістичний	Застарілий дизайн	Потужний, але складний
Відкритість коду	Закритий	BSD ліцензія	Закритий
Спільнота розробників	Підтримка Microsoft	Мала активність	Закрита

На таблиці 1.1 видно, що жоден з проаналізованих програвачів не задовольняє всі сучасні вимоги. Groove обмежений Windows, Audacious не має сучасного інтерфейсу, а MusicBee, попри потужність, недоступний для інших ОС окрім Windows. Достатній рівень кросплатформності реалізовано лише в Audacious, але його застарілий дизайн і відсутність обміну даними роблять його менш привабливим. Усі три рішення не пропонують механізмів

безпечного обміну, а закритість коду Groove та MusicBee обмежує розвиток функціоналу. Audacious хоч і має відкритий код його спільнота майже неактивна.

1.3 Аналіз вимог до програмного засобу та постановка завдання

Під час аналізу наявних на ринку аудіопрогравачів було виявлено ключові прогалини, які обмежують їхню ефективність у сучасних умовах. Для розробки нового програмного рішення, що поєднує безпеку, кросплатформність та гнучкість, було сформульовано функціональні та нефункціональні вимоги.

Ключові функціональні аспекти включають підтримку великої кількості аудіоформатів, реалізацію механізмів безпечного обміну даними з використанням P2P-передачі із шифруванням, а також інтеграцію високоточного еквалайзера з можливістю ручного налаштування частот і шаблонними профілями. Особливу увагу необхідно приділити інтерфейсу, який має поєднувати інтуїтивність для новачків із розширеними налаштуваннями для досвідчених користувачів.

Нефункціональні вимоги включають забезпечення високого рівня безпеки на всіх етапах роботи з даними, кросплатформній сумісності (включаючи десктопні ОС, мобільні платформи та AndroidTV), оптимізації продуктивності навіть на слабких пристроях, а також архітектурній відкритості для майбутніх модифікацій спільнотою. Програмний продукт має бути масштабованим та модульним, що дозволить в майбутньому інтегрувати нові функції без порушення стабільності системи.

Створений застосунок має стати універсальним рішенням для користувачів, які потребують безпеки, гнучкості та доступності на будь-якій платформі. Впровадження методів захисту та обміну аудіофайлами, відкритості коду та підтримки всіх сучасних ОС забезпечить конкурентоспроможність на ринку аудіопрогравачів.

2 ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Вибір технологій реалізації

Пропонований музичний програвач — це десктопний додаток, який поєднує відтворення аудіо, обмін даними через P2P-мережі, обробку звуку та інтуїтивно зрозумілий інтерфейс. Технологій реалізації існує безліч, проте для забезпечення необхідного функціоналу необхідно розумно підібрати технології реалізації.

Для розробки інтерфейсу було вибрано декларативну мову програмування QML яка є компонентом фреймворку Qt. Вона дозволяє створювати динамічні та адаптивні інтерфейси, що сприяє більш інтуїтивному користувацькому досвіду.

Фреймворк Qt виступає центральним елементом у розробці графічного інтерфейсу завдяки своїй кросплатформності та високій продуктивності. Використання цього фреймворку гарантує, що додаток працюватиме стабільно на різних операційних системах та платформах.

Також враховуючи те що фреймворк Qt передбачає використання C++ як мови програмування для реалізації бізнес-логіки додатка, її й буде використано. Це дозволить забезпечити високу продуктивність та ефективне управління ресурсами. Такий вибір мови дозволить створити оптимізований і масштабований код, здатний впоратися з обчислювальними задачами, які виникають у процесі обробки аудіо даних та обміну інформацією через P2P-мережі.

Для декодування та обробки аудіо було розглянуто три популярні варіанти які широко використовуються: QtMultimedia, FFmpeg (libav) та sndfile.

Спочатку було проаналізовано можливості які надає модуль Qt для декодування та роботи з аудіо потоком: QtMultimedia. Цей підхід має інтегровану підтримку у самому Qt, що забезпечує зручність використання та

сумісність з іншими компонентами фреймворку. Однак, попри простоту інтеграції, його обмеження в гнучкості та менш широкий набір підтримуваних форматів можуть стати перешкодою при роботі з рідковживаними чи спеціалізованими аудіоформатами. Слід також зазначити, що внутрішня реалізація декодерів у QtMultimedia інколи не забезпечує достатньої оптимізації або можливості тонкого налаштування параметрів декодування, що може впливати на продуктивність у складних сценаріях.

Далі було проаналізовано бібліотеку libsndfile, яка спеціалізується на роботі з аудіо файлами та відома своєю стабільністю та простотою використання для базових завдань. Його перевагою є чистота API та орієнтація на читання/запис звукових даних. Проте, обмежений функціонал у плані підтримки складних операцій обробки, відсутність широкої підтримки різних кодеків і досить низька гнучкість у налаштуванні процесу декодування стають важливими недоліками при розробці сучасного музичного програвача з високими вимогами до якості звуку та адаптивності.

Найбільш універсальним рішенням, яке задовольнило вимоги проєкту, став FFmpeg. Це програмне забезпечення відоме своєю широкою підтримкою аудіо форматів та високою продуктивністю, що робить його ідеальним для інтеграції у систему, де потрібна обробка складних аудіо даних. FFmpeg надає розширені можливості налаштування, дозволяючи точно оптимізувати процес декодування відповідно до вимог додатка. Хоча інтеграція FFmpeg вимагає більше зусиль, ніж використання вбудованих рішень, її масштабованість, стабільність і можливість обробки великого кола форматів з високою якістю та ефективністю виявилися визначальними факторами при прийнятті остаточного рішення. Таким чином, обрана технологія найкраще відповідає завданням проєкту, забезпечуючи гнучкість, адаптивність і можливості для подальшого розширення функціоналу аудіообробки.

Жоден великий проєкт не обходиться без системи збірки, оскільки компіляція та відстеження кожного файлу з програмним кодом та підтримка

при цьому інших операційних систем була б дуже складним, а інколи непосильним завданням.

Для управління збіркою проєкту було обрано CMake, оскільки він забезпечує універсальність та зручність у конфігурації процесу компіляції. У ході аналізу також було проведено порівняння з іншими популярними системами збірки, зокрема Mason та QMake. Mason демонструє досить непогану інтеграцію з певними інструментами та є доволі цікавим проєктом, проте його можливості налаштування та інтеграцію з Qt обмежені у порівнянні з CMake, що ускладнює масштабування великих проєктів. QMake є власним засобом збірки для Qt проєктів, він надає простий та швидкий спосіб організації складання, проте його гнучкість значно поступається CMake, особливо при інтеграції сторонніх бібліотек і налаштуванні специфічних параметрів збірки. Завдяки своїй відкритості, підтримці, масштабованості та можливості точного налаштування процесу, в результаті порівняння було вирішено обрати саме CMake.

Крім того, для роботи з метаданими аудіофайлів було обрано бібліотеку taglib, яка не тільки дозволяє читати теги, але й редагувати їх. Здатність читати теги є базовою та необхідною функцією, що забезпечує відображення важливої інформації про треки (назва, виконавець та альбом тощо), тоді як редагування тегів, хоча і не є критичною вимогою, але надає користувачам приємну додаткову опцію для корегування метаданих.

2.2 Проєктування структури проєкту

Архітектурна концепція системи базується на модульному підході, де кожен компонент виконує специфічні функції, але водночас забезпечує тісну інтеграцію з іншими елементами. Основними модулями є аудіодвигун для декодування та відтворення файлів різних форматів, P2P-ядро для децентралізованого обміну контентом, система метаданих для управління музичною бібліотекою, модуль безпеки для шифрування та автентифікації, а

також адаптивний інтерфейс для різних платформ. Така архітектура дозволяє системі масштабуватися горизонтально, додаючи нові функції без порушення роботи наявних компонентів, і забезпечує стабільну роботу навіть при відмові окремих модулів.

Для реалізації проєкту було обрано рішення поділити весь проєкт на логічні одиниці — модулі, що дозволить чітко розділити відповідальність між компонентами, спростити тестування та полегшити подальше розширення функціоналу. Такий підхід має забезпечити гнучкість у розробці, дозволяючи незалежно оновлювати чи вдосконалювати окремі частини системи, не впливаючи на стабільність інших компонентів [10].

Система була поділена на такі основні модулі:

- «audio» — відповідає за роботу з аудіо;
- «effects» – плагінна система аудіоефектів;
- «common» — містить утиліти, спільні для всіх модулів;
- «core» — ядро системи, містить основну бізнес-логіку;
- «gui» — відповідає за інтерфейс користувача;
- «models» — визначає структури даних та моделі;
- «providers» – лагінна система «провайдерів» списків відтворення;
- «tags» — обробка метаданих аудіофайлів (читання/запис тегів).

Модулі додатка будуть реалізовані через систему «CMake QML Module API» 7.

CMake QML Module API — це набір інструментів і правил, що дозволяє організувати та інтегрувати QML-модулі в CMake-проєкти, забезпечуючи стандартизований підхід до управління залежностями та їх конфігурацією. Завдяки цьому підходу, кожен модуль буде розроблений як самостійна одиниця зі своєю відповідальністю, що спростить управління залежностями.

Модулі будуть пов'язані між собою через абстрактні інтерфейси (класи), що має забезпечити їх взаємодію без необхідності заглиблюватися у конкретні

деталі реалізації. Такий підхід сприятиме гнучкості та масштабованості системи, дозволяючи легко оновлювати або замінювати окремі компоненти без шкоди для загальної цілісності проєкту.

Для більш наочного розуміння поділу та взаємодії між модулями була створена структурна схема зв'язків (рис. 2.1), яка відображає логічну організацію системи.

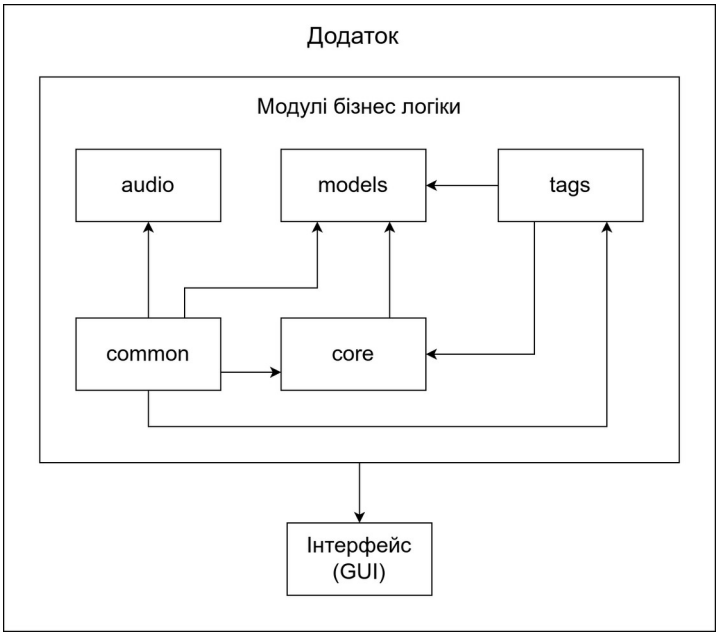

Рисунок 2.1 — Структурна схема зв'язків між модулями додатку

Кожен модуль, починаючи від audio, що відповідає за обробку звуку, і закінчуючи gui, який формує інтерфейс користувача, повинен інтегруватися через визначені інтерфейси взаємодії. Модуль core забезпечує бізнес-логіку та об'єднує функції роботи з аудіо, метаданими та P2P-комунікацією, тоді як модулі common і models виступають спільною базою для утиліт, структур даних і допоміжних функцій.

Такий підхід дозволить кожному компоненту ефективно обмінюватися інформацією з іншими модулями. Це має забезпечити достатній рівень стабільності системи навіть при зміні або оновленні окремих компонентів,

оскільки взаємодія між ними базуватиметься на чітко визначених інтерфейсах.

2.3 Проєктування аудіо системи

Аудіосистема виконує центральну роль у забезпеченні якісного декодування, обробки та відтворення звуку. Її архітектура організована за модульним принципом, що передбачає чітке розмежування функцій між окремими компонентами.

До ключових елементів системи входять:

1) підсистема декодування;
2) буферний механізм;
3) система обробки ефектів;
4) система виводу звуку.

Підсистема декодування покликана забезпечити роботу з різними аудіоформатами (MP3, WAV, FLAC, AAC тощо) завдяки інтеграції з такою потужною мультимедійною бібліотекою як FFmpeg. Цей компонент модуля відповідає за перетворення вхідних даних в один зі стандартизованих внутрішніх форматів (S32LE, F32LE, S16 тощо), зручних для подальшої обробки. Також в ньому будуть знаходитись механізми навігації по аудіо треку (пауза, перемотка тощо).

Буферний механізм — це проміжна ланка між декодером і системою виводу звуку. Цей компонент буферизує аудіо дані, що своєю чергою дозволить плавно та ефективно відтворювати аудіо навіть на повільних пристроях чи при поганому зв’язку з мережею.

Система обробки ефектів — це система яка надає інструменти для модифікації (постобробки) звукового потоку в реальному часі. Ефекти (еквалайзер, компресор, нормалізатор, реверберація тощо) будуть реалізовані як незалежні один від одного модулі, які можуть бути ввімкнені та вимкнені без втручання в основну логіку програми. Обробка відбуватиметься на рівні

проміжного буфера що зменшить шанси на виникнення різних артефактів під час одночасної обробки та відтворення.

Система виводу звуку відповідає за передачу обробленого аудіопотоку на фізичні пристрої (динаміки, навушники тощо). Вона взаємодіє з апаратними пристроями відтворення за допомогою стандартних API операційної системи (PipeWire чи PulseAudio для Linux, Core Audio для MacOS тощо). Його ключові функції передбачають синхронізацію з буферним механізмом, адаптацію до апаратних параметрів та обробка помилок відтворення.

На рисунку 2.2 зображено діаграму послідовностей роботи модуля «audio».

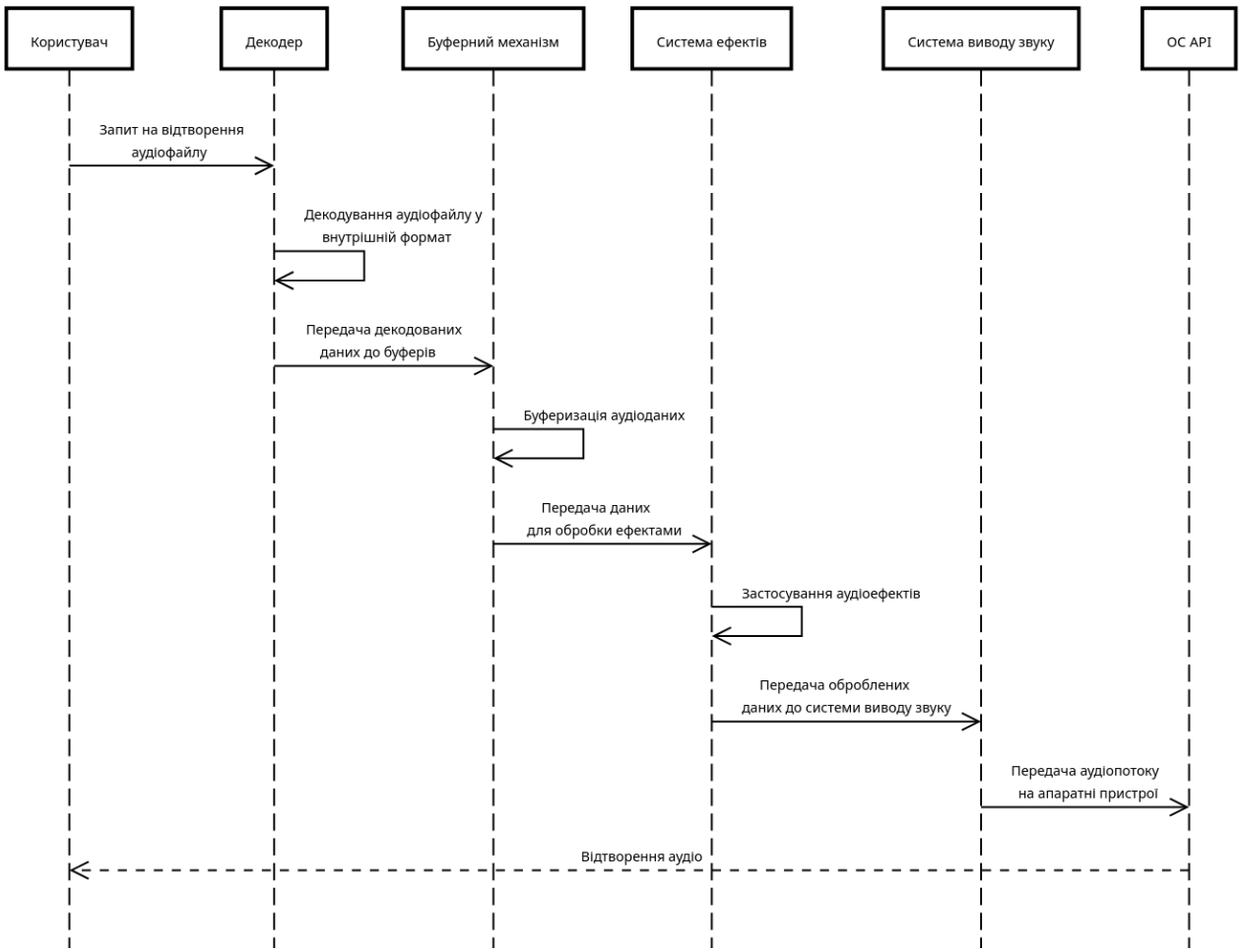

Рисунок 2.2 — Діаграма послідовностей роботи модуля «audio»

Зображена діаграма послідовностей ілюструє основні етапи відтворення аудіо. Спочатку користувач надсилає запит на відтворення аудіофайлу до декодера. Декодер самостійно розкодовує аудіофайл у внутрішній формат, після чого передає отримані дані до буферного механізму, де здійснюється їх буферизація для забезпечення безперебійного відтворення. Буферний механізм надсилає дані до системи ефектів, яка застосовує необхідні аудіоефекти. Після обробки ефектами, система передає дані до системи виводу звуку, що відправляє аудіопотік на апаратні пристрої через стандартний API операційної системи. Нарешті, API операційної системи забезпечує відтворення аудіо для користувача.

Загалом структура цього модуля має забезпечити стабільну та масштабовану роботу аудіосистеми, що своєю чергою дозволить легко інтегрувати нові формати та розширювати функціональність в майбутньому.

Підмодулі «providers» та «effects» утворюють ядро плагінної архітектури додатка, де providers відповідають за взаємодію з різними джерелами аудіоконтенту (локальні файли, мережеві ресурси, потокове мовлення тощо), а effects забезпечують обробку звукового сигналу в реальному часі. Кожен provider реалізує стандартизований інтерфейс для доступу до метаданих, декодування аудіоформатів та управління потоками, що дозволить системі уніфіковано працювати з різнорідними джерелами без зміни основного коду. Модулі effects, своєю чергою, мають бути організовані у вигляді конвеєра обробки (рис. 2.3), де кожен ефект може бути налаштований незалежно та комбінований з іншими для створення складних звукових перетворень.

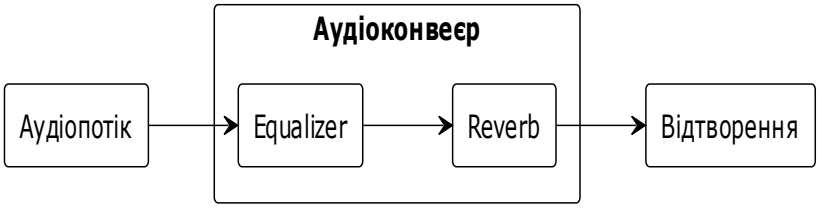

Рисунок 2.3 – Схема обробки ефектів

Система плагінів матиме гібридну архітектуру завантаження, яка поєднує статичне зв'язування критичних компонентів з динамічним підключенням розширювальних модулів, це дозволить забезпечити оптимальний баланс між продуктивністю та гнучкістю системи. Такий підхід суттєво полегшить залучення розробників до розширення функціональності програвача, оскільки вони зможуть створювати плагіни незалежно від основної кодової бази, не потребуючи доступу до вихідного коду всієї системи. Механізм додавання нових модулів без перекомпіляції чи перезапуску додатка дозволяє користувачам миттєво інтегрувати нові функції, просто скопіювавши файли плагінів у відповідну директорію, що значно знижує поріг входу для розробників та підвищує доступність розширення системи

Інтеграція з Qt6 забезпечує автоматичну серіалізацію налаштувань плагінів, підтримку локалізації інтерфейсів та безшовну роботу з системою сигналів і слотів для міжмодульної комунікації.

2.4 Проєктування мережевої взаємодії

Мережева взаємодія в аудіопрогравачі організована на принципах децентралізованого обміну даними, що забезпечує автономність клієнтів та мінімізацію залежності від зовнішніх серверів. Основним протоколом обрано P2P-з’єднання, яке дозволяє здійснювати пряму комунікацію між пристроями без проміжних ланок.

Серед альтернативних підходів до мережевої взаємодії також розглядалася клієнт-серверна архітектура. Однак P2P було обрано через його переваги в умовах локальних мереж: зменшення затримок, відсутність витрат на серверну інфраструктуру, можливість прямого обміну великими обсягами даних (наприклад, аудіофайлами) без обмежень пропускної здатності сторонніх сервісів. Крім того, децентралізований підхід підвищує надійність

системи — навіть при виході з ладу одного вузла, інші продовжуватимуть нормально функціонувати.

Для виявлення доступних списків відтворення у локальній мережі було вибрано механізм на основі UDP-трансляції (broadcast). Цей метод дозволяє ефективно знаходити інші пристрої, які пропонують доступні списки, не вимагаючи складної конфігурації або централізованих серверів. При кожному запиті на наявність нових списків відтворення, кожен клієнт відправляє пакети, що містять метадані в його локальній бібліотеці. Інші клієнти, що отримують ці пакети, можуть відобразити отриману інформацію в інтерфейсі для подальшого вибору. Для демонстрації алгоритму пошуку списків в локальній мережі було побудовано діаграму послідовностей (Рис. 2.4)

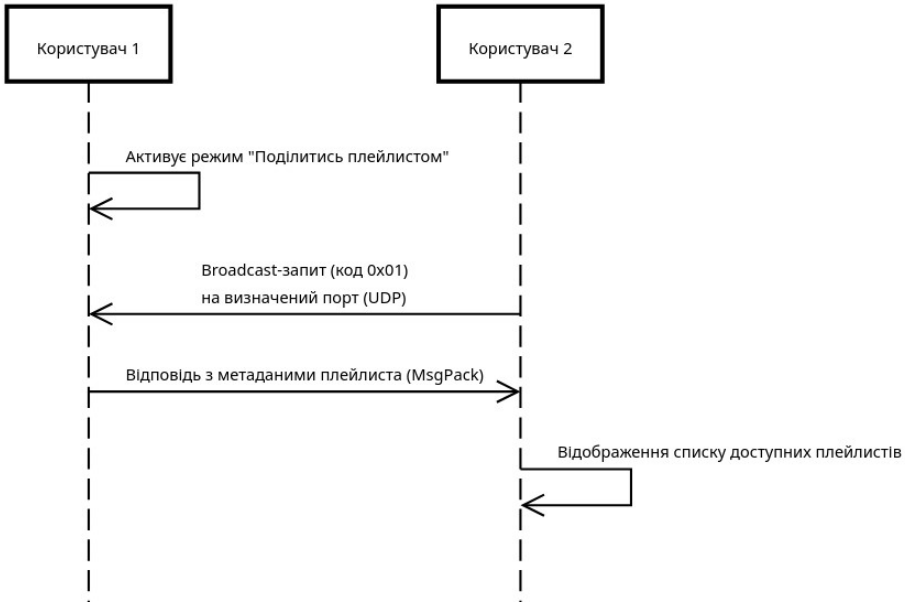

Рисунок 2.4 — Алгоритм пошуку плейлистів

Після вибору користувачем списку, система встановлює з’єднання між клієнтами. З'єднання здійснюється за допомогою алгоритму обміну ключами Діффі-Геллмана, що дозволяє безпечно обмінюватися публічними ключами та генерувати спільний секретний ключ для шифрування та дешифрування даних. Після успішного встановлення з'єднання між клієнтами передаються

метадані списку відтворення, які включають інформацію про доступні аудіофайли, їхні ідентифікатори та інші властивості.

Ці дані передаються у форматі MsgPack для мінімізації розміру та забезпечення ефективності передачі. Він дозволяє значно зменшити розмір передаваних даних у порівнянні з текстовими форматами, такими як JSON, що особливо важливо при роботі з великими аудіофайлами. Даний формат є бінарним і має високу швидкість серіалізації та десеріалізації, що робить його ідеальним для мережевих комунікацій.

Отримані метадані використовуються для відтворення обраного аудіофайлу. Після отримання запиту на відтворення, клієнт, який надає список, починає передачу даних про аудіофайл, використовуючи надійне шифрування з використанням алгоритму AES-256. Файл передається безпосередньо до клієнта, який відправив запит, і починається процес відтворення. Запит на відтворення аудіофайлу ініціюється від користувача, який бажає прослухати пісню з іншого списку відтворення. Після підтвердження з’єднання та передачі файлу, система перевіряє його цілісність і починає відтворення через локальний аудіоінтерфейс.

2.5 Проєктування інтерфейсу користувача

Серед всі варіантів взаємодії саме графічний інтерфейс стає основою, яка може забезпечити простоту використання та достатню гнучкість для користувача. Саме графічний інтерфейс закладає основу успішного програмного продукту, а враховуючи, що цей продукт — музичний програвач, його вагомість збільшується пропорційно до потреб інтерактивного керування та емоційного сприйняття, що безпосередньо впливає на задоволення користувача.

Графічний інтерфейс повинен бути достатньо гнучким в налаштуваннях для професійних користувачів та одночасно інтуїтивно зрозумілим для нових користувачів. Це дозволяє створити зручний продукт,

який адаптується до різних рівнів досвіду користувачів, забезпечує легкий доступ до основних функцій і надає можливість детального налаштування для оптимізації роботи з музичним контентом.

Основною функцією застосунку є прослуховування аудіофайлів, тому надзвичайно важливим є забезпечити користувачу зручний, інтуїтивно зрозумілий список відтворення та ефективний механізм пошуку в ньому. Ці елементи повинні органічно поєднуватись із сучасним дизайном, що відповідатиме актуальним тенденціям і водночас забезпечувати легкий доступ до необхідної користувачу інформації.

Крім того, для полегшення навігації по застосунку було спроєктовано навігаційну панель у вигляді списку, розташовану з лівої частини інтерфейсу. Ця панель дозволяє швидко перемикатися між категоріями.

На рисунку 2.5 зображено макет вище згаданих частин інтерфейсу.

Рисунок 2.5 — Макет основних частин інтерфейсу програми

Також враховуючи що метою є створення гнучкого в налаштуваннях застосунку, було спроєктовано зручне меню налаштувань, що дозволяє користувачу швидко змінювати параметри. В меню присутнє поле пошуку налаштувань, яке допомагає набагато швидше знаходити потрібні параметри

серед широкого набору опцій, забезпечуючи при цьому легкість використання.

Макет меню налаштувань зображено на рисунку 2.6.

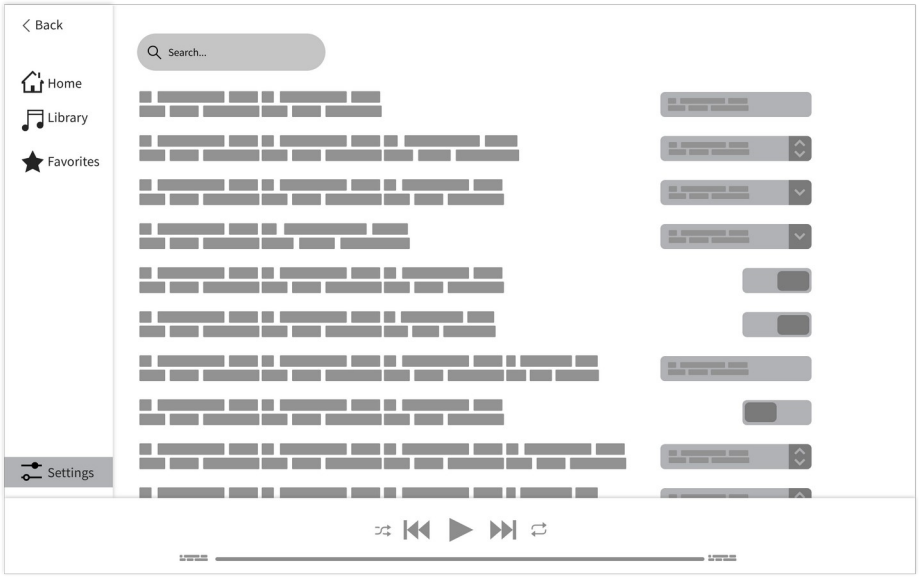

Рисунок 2.6 — Макет меню налаштувань

Також, враховуючи кросплатформову націленість застосунку, є життєво необхідним забезпечити його коректне відображення та функціональність на різних пристроях. Для цього буде використано адаптивні макети, які оптимізують інтерфейс під різні розміри екранів та орієнтацію пристроїв, гарантують єдиний користувацький досвід незалежно від апаратних характеристик.

Є багато людей, і всі вони різні – з різними побажаннями, вподобаннями та особливостями сприйняття інформації. Тому потрібно забезпечити підтримку тем оформлення, щоб кожен користувач міг адаптувати зовнішній вигляд застосунку під себе. Було вирішено створити власну тему, яка гармонійно поєднує в собі сучасний вигляд і зручність, а також буде реалізовано більш звичні – чорну та білу теми. Це дозволить не лише покращити візуальне сприйняття, а й зменшити навантаження на зір у темному чи яскравому середовищі.

Головна тема оформлена в темних відтінках синьо-сірого, що створюють спокійну та сучасну атмосферу. Основні та акцентні кольори помірно контрастні, а текст має м’які світлі тони для зручності читання. Така палітра забезпечує комфортне візуальне сприйняття та чітку ієрархію елементів.

Також буде реалізовано світлу тему з кремовим фоном і темним текстом, а також класичну темну тему з глибоким темним фоном та світлим кремовим текстом. Це дозволить користувачам обрати комфортний варіант відповідно до умов освітлення чи власних уподобань.

Отже, у цьому розділі було детально розглянуто етапи проєктування системи. Спочатку було обґрунтовано вибір технологій, які найкраще відповідають вимогам до кросплатформового мультимедійного застосунку з можливістю мережевої синхронізації. Далі було розроблено логічну структуру проєкту з розділенням на окремі модулі, що забезпечує зручність підтримки, масштабування та тестування. Особливу увагу приділено аудіопідсистемі – описано підхід до відтворення, обробки та контролю аудіопотоку із врахуванням якості та затримок.

Крім того, важливою частиною стало проєктування механізмів мережевої взаємодії між клієнтами, з акцентом на ефективний обмін даними та виявлення доступних користувачів у межах локальної мережі. Завершальним етапом розділу було формування вимог до графічного інтерфейсу користувача – проєктування панелей, тем та елементів керування, що мають забезпечити інтуїтивну взаємодію і зручність у щоденному використанні. Усі ці компоненти формують цілісну архітектуру застосунку, готову до реалізації та подальшого розвитку.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Реалізація основних модулів системи

Модуль «common» є утилітним компонентом архітектури програми, що реалізує принцип централізації допоміжної функціональності та забезпечує уніфікований доступ до загальних сервісів для всіх підсистем музичного програвача. Цей утилітний модуль спроєктовано як незалежну бібліотеку, що може бути використана будь-яким компонентом системи без створення циклічних залежностей, забезпечуючи чисту архітектуру та спрощуючи процес розробки. Лістинг CMakeLists.txt файлу цього модуля подано в лістингу 3.1.

Лістинг 3.1 – CMakeLists.txt файл модуля «common»

```
qt_add_qml_module(common
    STATIC
    VERSION 1.0
    URI "CustomPlayer.Common"
    SOURCES
        include/Log.h
        include/SongMetadata.h
        src/Utils.cpp include/Utils.h
    RESOURCE_PREFIX /
)
target_include_directories(common PUBLIC include)
target_link_libraries(common PRIVATE Qt6::Core)
```

Цей компонент містить просту самостійно написану систему логування, яка реалізована через набір макросів (Info, Warning, Error, Debug), що створюють ієрархічну систему повідомлень з власним призначенням та візуальним оформленням. Автоматичне додавання інформації про файл та рядок реалізовано через макроси __FILE__ та __LINE__, активні тільки в debug-режимі для оптимізації продуктивності реліз версій. Критичні помилки обробляються макросом Log_Error, який автоматично завершує програму,

запобігаючи пошкодженню даних. Код системи логування наведено в лістингу 3.2.

Лістинг 3.2 – Програмний код система логування

```
#ifdef NDEBUG
#define LOCATION ""
#else
#define LOCATION
QString("\n\t\033[1;37m(%1:%2)\033[0m").arg(__FILE__).arg(__LINE
__)
#endif
#define Log_Info(message) qDebug().noquote() <<
"[\033[1;37mInfo\033[0m]" << message << LOCATION
#define Log_Warning(message) qWarning().noquote() <<
"[\033[1;33mWarning\033[0m]" << message << LOCATION
#define Log_Error(message)qCritical().noquote() <<
"[\033[1;31mError\033[0m]" << message << LOCATION;exit(-1)
#ifdef NDEBUG
#define Log_Debug(message)
#else
#define Log_Debug(message) qDebug().noquote() <<
"[\033[1;32mDebug\033[0m]" << message << LOCATION
#endif
```

Також модуль common містить структуру даних SongMetadata, яка забезпечує централізоване зберігання інформації про музичні композиції з підтримкою Unicode-кодування. Основна інформація включає назву, альбом, жанр та виконавця для організації бібліотеки.

Технічні параметри (тривалість у мілісекундах, кількість каналів, бітрейт) використовуються аудіодвигуном для налаштування відтворення. Додаткові поля включають коментарі, рік випуску та URI для роботи з локальними файлами та мережевими ресурсами. Програмний код цієї структури наведено в лістингу 3.3.

Лістинг 3.3 – Програмний код структури SongMetadata

```
struct SongMetadata
{
        QString title;
        QString album;
        QString genre;
        QString comment;
        QString artist;
        int length;
        int channels;
        int bitrate;
        int sampleRate;
        unsigned int year;
        QString uri;
};
```

Наведена структура використовується в усіх підсистемах програми для передачі інформації про композиції між компонентами: аудіодвигуном для налаштування параметрів відтворення, системою управління бібліотекою для організації та пошуку треків, P2P логікою для обміну метаданими між користувачами та інтерфейсними компонентами для відображення інформації про поточну композицію. Уніфікований формат забезпечує сумісність між різними модулями та спрощує розширення функціональності без необхідності модифікації існуючих інтерфейсів.

Також система включає клас Utils який містить різні допоміжні методи необхідні для роботи система. Він реалізований як QML-інтегрований синглтон через макроси QML_ELEMENT та QML_SINGLETON.

Синглтон - це патерн проєктування, який забезпечує існування тільки одного екземпляра класу в межах програми та надає глобальну точку доступу до цього екземпляра, він використовується для ресурсів, які повинні бути унікальними в системі, таких як логгери, конфігураційні об'єкти або утилітарні класи.

Функція msecToDuration є членом класу Utils та містить макрос Q_INVOKABLE який використовується для позначення C++ методів, які можуть бути викликані з QML коду, забезпечуючи інтеграцію між C++ логікою та декларативним інтерфейсом користувача.

Ця функція забезпечує інтелектуальне форматування часу: "години:хвилини:секунди" для тривалих композицій та "хвилини:секунди" для коротких треків. Програмний код msecToDuration висвітлено в лістингу 3.4.

Лістинг 3.4 – Функція msecToDuration

```
QString Utils::msecToDuration(const int &durationInMsec)
{
    QTime time =
QTime::fromMSecsSinceStartOfDay(durationInMsec);
    if (time.hour() > 0)
        return time.toString("h:mm:ss");
    return time.toString("mm:ss");
}
```

Модуль «core» є центральним компонентом системи, що відповідає за управління критично важливими внутрішніми сервісами та забезпечення ефективної роботи з ресурсами програми. Лістинг програмного коду файлу CMakeLists.txt зображено в лістингу 3.5.

Лістинг 3.5 – CMakeLists.txt файл модуля «core»

```
qt_add_library(core STATIC
    src/DiskImageCache.cpp
    src/EventsHandler.cpp
    src/QueueManager.cpp
    src/ICache.cpp
    include/DiskImageCache.h
    include/ICache.h
    include/ISingleton.h
    include/IDiskCache.h)
target_include_directories(core PUBLIC include)
target_link_libraries(core PRIVATE common tags Qt6::Core
Qt6::Gui)
```

Цей модуль реалізує ключові механізми оптимізації продуктивності, управління життєвим циклом об'єктів та координації між різними підсистемами програми, забезпечуючи стабільну роботу всієї системи навіть при інтенсивному навантаженні.

Модуль організовано навколо чітко визначених інтерфейсів та їх конкретних реалізацій. Базовий інтерфейс ICache визначає загальний контракт для всіх операцій кешування, забезпечуючи уніфікований підхід до роботи з різними типами кешу (лістинг 3.6).

Лістинг 3.6 – Програмний код інтерфейсу ICache

```
class ICache
{
    public:
        virtual ~ICache() = default;
        virtual QString retrieve(const QString &uri) = 0;
    protected:
        ICache() = default;
        QString generateCacheKey(const QString &uri, const
QDateTime &time);
};
```

Цей клас загалом є лише інтерфейсом для інших класів, єдина функція яка в ньому реалізована це generateCacheKey, яка створює унікальний ідентифікатор (Хеш) для кешованих ресурсів на основі шляху до ресурсу та часу його останньої модифікації. Програмний код цієї функції зображено в лістингу 3.7.

Лістинг 3.7 – Програмний код функції generateCacheKey

```
QString ICache::generateCacheKey(const QString &uri, const
QDateTime &modTime)
{
    QString composite = uri + "_" +
modTime.toString(Qt::ISODate);
    return QCryptographicHash::hash(composite.toUtf8(),
QCryptographicHash::Sha1).toHex();
}
```

Ця функція використовується для забезпечення цілісності кешу. Коли файл змінюється, його час модифікації оновлюється, що призводить до генерації нового хеш-ключа, тим самим інвалідуючи старий кешований ресурс і забезпечуючи актуальність даних. Унікальність ключів гарантує відсутність колізій між різними ресурсами навіть при однакових назвах, а використання SHA-1 хешування забезпечує рівномірний розподіл ключів та швидкий пошук кешованих елементів.

У лістингу 3.8 зображено фрагмент коду інтерфейсу IDiskCache який розширює базовий інтерфейс ICache, додаючи специфічну функціональність для роботи з дисковим сховищем, тим самим вказуючи саме на те, що цей хеш саме файловий і призначений для організації кешу на постійному носії. Інтерфейс визначає абстрактний метод cacheDir(), який має повертати шлях до директорії кешування, дозволяючи різним реалізаціям використовувати власні стратегії розміщення кешованих файлів на диску.

Лістинг 3.8 – Програмний код інтерфейсу IDiskCache

```
class IDiskCache : public ICache
{
    public:
        virtual ~IDiskCache() = default;
        virtual QString cacheDir() const = 0;
    protected:
        IDiskCache() = default;
};
```

Також в модулі «core» реалізовано шаблонний інтерфейс ISingleton, який забезпечує типобезпечну реалізацію патерну Singleton для будь-якого класу, що його успадковує:

```
template <typename T> class ISingleton
{
    public:
        ISingleton(ISingleton &&) = delete;
        ISingleton(const ISingleton &) = delete;
        ISingleton &operator=(const ISingleton &) = delete;
```

```
        static T *instance()
        {
            static T instance;
            return &instance;
        }
    protected:
        ISingleton() = default;
        ~ISingleton() = default;
};
```

Цей інтерфейс використовує статичну локальну змінну в методі instance() для гарантування створення єдиного екземпляра класу, при цьому явно заборонює копіювання, переміщення та присвоєння об'єктів через видалені конструктори та оператори. Такий підхід дозволить забезпечити потокобезпечність без додаткової синхронізації, тому що в C++11 та новіших стандартах ініціалізація статичних локальних змінних гарантовано є потокобезпечною.

Основна реалізація кешування представлена класом DiskImageCache, який поєднує функціональність кешування з патерном Singleton, забезпечуючи централізоване управління зображеннями альбомів. Клас успадковується від QObject для інтеграції з системою сигналів і слотів Qt, IDiskCache для реалізації специфічних методів дискового кешування, та ISingleton<DiskImageCache> для гарантування єдиного екземпляра в межах програми.

Обов’язковий метод retrieve починається з перевірки існування файлу та отримання його метаданих:

```
QFileInfo info(filePath);
if (!info.exists()) {
    Log_Warning(QString("No exists file %1").arg(filePath));
    return QString();
}
QDateTime modTime = info.lastModified();
QString hashKey = generateCacheKey(filePath, modTime);
QString hashedFilePath = cacheDir() + "/" + hashKey + ".jpg";
```

Далі система перевіряє наявність кешованого зображення та повертає його шлях у разі існування:

```
if (Qfile::exists(hashedFilePath)) {
    return hashedFilePath;
}
```

Якщо кешований файл відсутній, система спочатку витягує зображення з метаданих аудіофайлу:

```
QImage image = id3v2_extract_image(filePath);
if (image.isNull()) {
    return QString();
}
```

Далі збереження зображення виконується в потокобезпечному режимі з автоматичним створенням директорії:

```
QMutexLocker locker(&m_mutex);
QDir _cacheDir(cacheDir());
if (!_cacheDir.exists()) {
    _cacheDir.mkpath(".");
}

if (!image.save(hashedFilePath, "JPG", 30)) {
    Log_Warning(QString("Failed to save image %1 to
cache").arg(hashedFilePath));
    return QString();
}
return hashedFilePath;
```

Використання QMutexLocker в цьому випадку забезпечує автоматичне блокування м'ютекса m_mutex на початку критичної секції та його гарантоване розблокування при виході з області видимості змінної locker. Потокобезпечність критично важлива в даному контексті, оскільки кілька потоків можуть одночасно намагатися створити ту саму директорію кешу або записати файли з ідентичними іменами.

Метод cacheDir у класі DiskImageCache визначає системну директорію для збереження кешованих зображень:

```
QString DiskImageCache::cacheDir() const {
    return
QStandardPaths::writableLocation(QStandardPaths::CacheLocation)
+ "/images";
}
```

Використання QStandardPaths::CacheLocation забезпечує відповідність стандартам кожної операційної системи для зберігання тимчасових файлів кешу.. На Linux-системах кешовані зображення зберігатимуться в директорії ~/.cache/CustomPlayer/images, що відповідає стандарту XDG Base Directory для тимчасових файлів у домашній теці користувача. В операційній системі Windows файли кешу розміщуватимуться за шляхом C:\Users\[Username] \AppData\Local\CustomPlayer\cache\images, використовуючи стандартну теку AppData для локальних даних програм. На macOS кешовані зображення зберігатимуться в ~/Library/Caches/[ApplicationName]/images, відповідно до рекомендацій Apple щодо структури файлової системи.

Система оптимізована для мінімізації навантаження на дискову підсистему та оперативну пам'ять. Зображення зберігаються з компресією 30%, що значно зменшує їх розмір без критичної втрати якості для відображення в інтерфейсі програвача. Кеш-директорія організована як плоска структура з хешованими іменами файлів, що забезпечує швидкий доступ навіть при великій кількості кешованих зображень.

Вона автоматично використовує наявні кешовані файли, уникаючи повторного декодування та збереження однакових зображень. Це особливо важливо для великих музичних колекцій, де одне зображення альбому може використовуватися для множини треків.

Модуль tags є спеціалізованим компонентом для професійної роботи з метаданими музичних файлів, побудованим на основі бібліотеки TagLib для

забезпечення парсингу та отримання інформації з різноманітних аудіоформатів. Модуль реалізує комплексний підхід до обробки як текстових метаданих, так і вбудованого візуального контенту.

Функція id3v2_read() яка знаходиться в цьому модулі спочатку виконує повне сканування ID3v2 тегів у MPEG-файлах:

```
SongMetadata id3v2_read(const QString &filePath) {
    SongMetadata data;
    TagLib::MPEG::File file(filePath.toStdString().c_str());
    if (!file.isValid())
        return data;
    TagLib::Tag *tag = file.tag();
    if (!tag) {
        return data;
    }
```

Далі перевіряється валідність файлу та існування тегів, запобігаючи помилкам при роботі з пошкодженими файлами. Далі відбувається вилучення основних метаданих з конвертацією в кодування UTF-8:

```
data.title = QString::fromStdString(tag-
>title().toCString(true));
data.artist.push_back(QString::fromStdString(tag-
>artist().toCString(true)));
data.album = QString::fromStdString(tag-
>album().toCString(true));
data.year = tag->year();
data.uri = filePath;
```

Використання toCString(true) забезпечує коректне перетворення рядків у UTF-8, це критично важливо для файлів з неанглійськими назвами. Всі технічні параметри аудіопотоку такі як семплрейт, кількість каналів та бітрейт отримуються через об'єкт AudioProperties:

```
data.length = file.audioProperties()->lengthInMilliseconds();
data.sampleRate = file.audioProperties()->sampleRate();
data.channels = file.audioProperties()->channels();
data.bitrate = file.audioProperties()->bitrate();
```

Також в модулі «tags» міститься функція id3v2_extract_image для пошуку та витягування зображень з ID3v2 тегів:

```
QImage id3v2_extract_image(const QString &filePath) {
    TagLib::MPEG::File file(filePath.toStdString().c_str());
    if (!file.isValid())
        return QImage();
    QImage image;
    TagLib::ID3v2::Tag *id3v2Tag = file.ID3v2Tag();
    if (id3v2Tag) {
        TagLib::ID3v2::FrameList frameList = id3v2Tag-
>frameList("APIC");
```

Алгоритм працює з фреймами типу "APIC" (Attached Picture), які містять обкладинки альбомів. Динамічне перетворення забезпечує безпечне приведення загальних фреймів у спеціалізовані об'єкти:

```
if (!frameList.isEmpty()) {
  TagLib::ID3v2::AttachedPictureFrame *coverFrame =
static_cast<TagLib::ID3v2::AttachedPictureFrame*>(frameList.fron
t());
  TagLib::ByteVector imageData = coverFrame->picture();
  image.loadFromData(reinterpret_cast<const
uchar*>(imageData.data()), static_cast<int>(imageData.size()));
  return image;
}
```

Декодування зображень оптимізовано для роботи з різними форматами через використання Qt для створення об'єктів QImage безпосередньо з бінарних даних, мінімізуючи витрати на проміжні копіювання у пам'яті.

Модуль models реалізує комплексну систему моделей даних для Qt/QML додатку, забезпечуючи ефективне управління плейлистами, фільтрацію даних та розширену плагінну архітектуру для роботи з різними форматами файлів.

Для забезпечення гнучкості та розширюваності системи управління списками відтворення було реалізовано комплексну систему провайдерів, що базується на модульній архітектурі з використанням інтерфейсів та

фабричного патерну. Ця система є ключовим компонентом модуля models та забезпечує уніфікований підхід до роботи з різноманітними форматами, дозволяючи легко додавати підтримку нових без необхідності внесення змін до основного коду програми.

Архітектурна концепція системи провайдерів ґрунтується на принципах інверсії залежностей та відкритості для розширення при закритості для модифікації. Це досягається через чітке розділення відповідальності між різними компонентами системи та використання абстрактних інтерфейсів для визначення контрактів взаємодії. Така організація коду забезпечує високий рівень модульності та дозволяє розробникам створювати незалежні модулі підтримки специфічних форматів списків відтворення.

Система провайдерів побудована на основі чотирьох ключових інтерфейсів, кожен з яких виконує специфічну роль у загальній архітектурі. Базовий інтерфейс IPlaylistProvider визначає фундаментальну функціональність роботи зі списками відтворення, включаючи операції завантаження, оновлення та доступу до метаданих:

```
class IPlaylistProvider : public QObject
{
        Q_OBJECT
    public:
        explicit IPlaylistProvider(QObject *parent = nullptr) :
QObject(parent){}
        virtual ~IPlaylistProvider() override = default;
        virtual QList<PlaylistFormat> uriFormats() const = 0;
        virtual QString uri() const = 0;
        virtual bool setUri(const QString &uri) = 0;
        virtual bool isReadOnly() const { return true; }
        virtual QString playlistName() const = 0;
        virtual int songsCount() const = 0;
        virtual SongMetadata songAt(int index) const = 0;
        virtual bool update() = 0;
        virtual bool load(QString uri = QString()) = 0;
    signals:
        void nameChanged(const QString &name);
        void reset();
```

```
};
Q_DECLARE_INTERFACE(IPlaylistProvider,
"CustomPlayer.EPlaylistProvider")
```

Цей інтерфейс забезпечує єдиний протокол взаємодії з плейлистами незалежно від їх внутрішнього формату або способу зберігання даних. Методи uriFormats() та uri() дозволяють визначати підтримувані формати та поточне розташування плейлиста, тоді як методи доступу до контенту забезпечують уніфікований спосіб отримання інформації про треки.

Розширений інтерфейс IEditablePlaylistProvider успадковує базовий функціонал та додає можливості редагування контенту плейлистів:

```
class IEditablePlaylistProvider : public IPlaylistProvider
{
        Q_OBJECT
        Q_INTERFACES(IPlaylistProvider)
    public:
        explicit IEditablePlaylistProvider(QObject *parent =
nullptr) : IPlaylistProvider(parent) {}
        virtual ~IEditablePlaylistProvider() override = default;
        virtual bool isReadOnly() const override{return false;}
        virtual bool setPlaylistName(const QString
&playlistName) = 0;
        virtual bool setSongAt(int index, const SongMetadata
&entry) = 0;
        virtual bool addSongs(const QList<SongMetadata> &songs)
= 0;
        virtual bool removeSong(int index) = 0;
        virtual bool save(QString uri = QString()) = 0;
    signals:
        void songsAdded(int index, int count);
        void songRemoved(int index);
        void songChanged(int index);
};
Q_DECLARE_INTERFACE(IEditablePlaylistProvider,
"CustomPlayer.EditablePlaylistProvider")
```

Цей інтерфейс передбачає методи для модифікації метаданих, додавання та видалення треків, а також збереження змін. Розділення на

базовий та розширений інтерфейси дозволяє підтримувати як статичні плейлисти, призначені лише для читання, так і повністю редаговані списки відтворення. Сигнали songsAdded, songRemoved та songChanged забезпечують детальне відстеження змін у плейлисті для синхронізації з користувацьким інтерфейсом.

Інтеграція з системою QML забезпечується через спеціалізований інтерфейс IPlaylistProviderInterface, який дозволяє провайдерам надавати власні компоненти користувацького інтерфейсу для налаштування специфічних параметрів списку відтворення при створенні:

```
class IPlaylistProviderInterface
{
    public:
        virtual ~IPlaylistProviderInterface() = default;
        Q_INVOKABLE virtual QObject *providerUi(QObject
*qmlContextParent) = 0;
        virtual bool isRequirementsProvided() const = 0;
};
#define IPlaylistProviderInterface_iid
"CustomPlayer.PlaylistProviderInterface"
Q_DECLARE_INTERFACE(IPlaylistProviderInterface,
IplaylistProviderInterface_iid)
```

Цей підхід було використано при розробці, щоб забезпечуючи високий рівень гнучкості у створенні унікальних інтерфейсів налаштувань для різних типів провайдерів, водночас зберігаючи узгодженість загального користувацького досвіду. Метод providerUi() дозволяє динамічно створювати QML-компоненти, специфічні для кожного типу провайдера, а isRequirementsProvided() перевіряє готовність провайдера до додавання списку відтворення.

Для управління життєвим циклом провайдерів було створено фабричний інтерфейс IPlaylistProviderFactory, який реалізує патерн Factory Method для створення екземплярів провайдерів:

```
class IPlaylistProviderFactory : public Qobject {
```

```
        Q_OBJECT
    public:
        virtual ~IPlaylistProviderFactory() override = default;
        virtual IPlaylistProvider *create(QObject *parent =
nullptr) const = 0;
        virtual int priority() const { return 0; };
};
#define IPlaylistProviderFactory_iid
"CustomPlayer.PlaylistProviderFactory"
Q_DECLARE_INTERFACE(IPlaylistProviderFactory,
IplaylistProviderFactory_iid)
```

Кожна реалізована фабрика має визначений пріоритет через метод priority(), що дозволяє системі автоматично вибирати найбільш відповідний провайдер для конкретного формату файлу. Фабрики також відповідають за створення екземплярів провайдерів через метод create() та управління їх життєвим циклом у контексті Qt-об'єктів.

Для уніфікованого представлення інформації про формати плейлистів використовується спеціальна структура PlaylistFormat:

```
#pragma once
#include <QString>
#include <QStringList>
struct PlaylistFormat
{
        QString name;
        QStringList nameFilters;
};
```

Ця структура забезпечує стандартизований спосіб опису підтримуваних форматів файлів, де name містить зрозумілу для людини назву формату, а nameFilters – список масок файлів для фільтрації у діалогах відкриття файлів, наприклад *.m3u, .pls чи URI адреси. Такий підхід дозволяє системі автоматично генерувати списки підтримуваних форматів, які потім можна використати в користувацькому інтерфейсі.

Система підтримує як локальні, так і віддалені списки відтворення, що реалізується через абстракцію URI. Провайдери можуть працювати з файлами в локальній файловій системі, мережевими ресурсами, або навіть з даними, що зберігаються в розподіленому захищеному сховищі. Така гнучкість досягається завдяки тому, що всі операції з даними абстраговані через інтерфейси, а конкретна реалізація залежить від типу провайдера.

Розширюваність системи забезпечується через можливість динамічного завантаження нових провайдерів через систему плагінів Qt. Нові провайдери можуть бути додані до системи без перекомпіляції основного додатка, що особливо важливо для підтримки нових або рідкісних форматів плейлистів. Система автоматично виявляє та реєструє нові провайдери при запуску програми завдяки використанню макросів Q_DECLARE_INTERFACE та відповідних ідентифікаторів інтерфейсів.

При розробці програми активно використовувалися підходи які передбачають надання моделі для відображення, що є фундаментальним принципом архітектури Qt Model/View/Delegate. Ця парадигма забезпечує чітке розділення логіки даних та їх візуального представлення, дозволяючи створювати гнучкі та масштабовані інтерфейси користувача.

Клас PlaylistModel є моделлю списку відтворення та центральним компонентом для відображення вмісту окремого плейлиста, забезпечуючи повну інтеграцію з QML-інтерфейсом та Qt Model/View архітектурою в якій він виступає саме моделлю. Цей клас успадковується від QAbstractListModel, що надає стандартний інтерфейс для роботи з упорядкованими колекціями даних та автоматично забезпечує підтримку всіх необхідних сигналів для оновлення інтерфейсу при зміні вмісту плейлиста.

Модель використовує патерн делегування, передаючи всю логіку роботи з даними провайдеру плейлиста:

```
PlaylistModel::PlaylistModel(QSharedPointer<IPlaylistProvider>
provider, QObject *parent)
```

```
    : QAbstractListModel(parent)
    , m_provider(provider){}
```

Конструктор за замовчуванням видалений, щоб забезпечити обов'язкову ініціалізацію моделі з провайдером. Використання QSharedPointer гарантує автоматичне управління пам'яттю та безпечний доступ до провайдера з різних частин програми.

Основні методи QAbstractListModel які реалізує PlaylistModel делегують виклики до провайдера:

```
int PlaylistModel::rowCount(const QModelIndex &parent) const
{
    if (parent.isValid() || !m_provider)
        return 0;
    return m_provider->songsCount();
}

QString PlaylistModel::playlistName() const
{
    return m_provider->playlistName();
}
```

Метод rowCount() перевіряє валідність батьківського індексу та існування провайдера перед поверненням кількості композицій, забезпечуючи стабільність роботи навіть при некоректних викликах.

Ролі в системах моделей Qt визначають, які дані можна отримати з кожного елемента моделі. Для доступу до специфічних властивостей аудіофайлів було реалізовано власні ролі, що розширюють стандартні Qt::DisplayRole та Qt::UserRole:

```
        enum SongRoles {
            TitleRole = Qt::UserRole + 1,
            ArtistRole,
            AlbumRole,
            GenreRole,
            CommentRole,
            LengthRole,
            ChannelsRole,
```

```
        BitRateRole,
        SampleRateRole,
        YearRole,
        UriRole,
        AlbumArtRole
    };
```

Але простої реалізації enum недостатньо, щоб це працювало, також було реалізовано метод roleNames(), який встановлює відповідність між числовими ідентифікаторами ролей та їх текстовими іменами, доступними в QML:

```
QHash<int, QByteArray> roleNames() const override
{
    return {
        {     TitleRole,      "title"},
        {    ArtistRole,     "artist"},
        {     AlbumRole,      "album"},
        {     GenreRole,      "genre"},
        {   CommentRole,    "comment"},
        {    LengthRole,     "length"},
        {  ChannelsRole,   "channels"},
        {   BitRateRole,    "bitrate"},
        {SampleRateRole, "samplerate"},
        {      YearRole,       "year"},
        {       UriRole,        "uri"},
        {  AlbumArtRole,   "albumArt"},
    };
}
```

Система моделей Qt автоматично використовує цей метод для створення зв'язку між C++ кодом та QML інтерфейсом, після чого в QML можна звертатися до властивостей композицій через зрозумілі імена типу model.title, model.artist, замість числових ідентифікаторів ролей.

Метод data() є центральним компонентом моделі - він відповідає за повернення конкретних даних для кожної ролі, це основний інтерфейс між моделлю та представленням. Програмний код методу data відображено в лістингу 3.9.

Лістинг 3.9 – Метод data() класу PlaylistModel

```
QVariant PlaylistModel::data(const QModelIndex &index, int role)
const {
    const int row = index.row();
    if (!index.isValid() || row < 0 || row >= m_provider-
>songsCount()) return QVariant();
    const auto &song = m_provider->songAt(row);
    switch (role) {
        case TitleRole: return !song.title.isEmpty() ?
song.title : QFileInfo(song.uri).completeBaseName();
        case ArtistRole:
return song.artist;
        case AlbumRole:
return song.album;
        case UriRole:
return song.uri;
        case AlbumArtRole:
return DiskImageCache::instance()->retrieve(song.uri);
        default:
return QVariant();
    }
}
```

Дані для кожної композиції отримуються з провайдера через метод songAt(row), який повертає об'єкт з усіма метаданими композиції.

Особливістю реалізації є fallback-логіка для назви композиції: якщо метадані не містять назви, використовується ім'я файлу без розширення, тому користувач завжди бачить зрозумілу назву. Роль AlbumArtRole інтегрується з системою кешування зображень DiskImageCache, що забезпечує швидке отримання обкладинок альбомів без повторного зчитування з диска.

Алгоритм додавання композицій підтримує як URL-адреси, так і прямі метадані файлів через універсальний метод:

```
bool PlaylistModel::addSongs(const QList<SongMetadata> &songs) {
    int oldCount = m_provider->songsCount();
    int newCount = oldCount + songs.size();
    auto editable = qobject_cast<IEditablePlaylistProvider
*>(m_provider.get());
    if (!editable)
```

```
        return false;
    if (!editable->addSongs(songs)) {
        return false;
}
    beginInsertRows(QModelIndex(), oldCount, newCount - 1);
    endInsertRows();
    editable->save();
    return true;
}
```

Перш за все метод обчислює поточну кількість композицій та визначає діапазон нових рядків, які будуть додані. Далі виконується перевірка, чи підтримує поточний провайдер редагування через qobject_cast до інтерфейсу IEditablePlaylistProvider – якщо провайдер доступний лише для читання, операція припиняється.

Після успішної передачі даних провайдеру модель повідомляє Qt про структурні зміни через пару beginInsertRows/endInsertRows, що забезпечує коректне оновлення всіх підключених представлень. Завершується процес автоматичним збереженням змін через виклик save(), гарантуючи персистентність даних.

Для нормалізації шляхів було реалізовано спеціалізовану функцію яка перетворює file:// URL у локальні шляхи :

```
static QString normalizeFilePath(const QUrl &url) {
    if (url.isValid() && url.scheme() == "file")
{ return url.toLocalFile();}
    return url.toString();
}
```

Для додавання пісень за URL-рядком було реалізовано функцію addSongs, яка спочатку нормалізує шляхи за допомогою функції normalizeFilePath, читає ID3v2 метадані та додає пісню, викликаючи addSongs за метаданими:

```
bool PlaylistModel::addSongs(const QList<QUrl> &urls) {
    QList<SongMetadata> songsToAdd;
```

```
    for (const auto &url : std::as_const(urls)) {
        QString normalizedPath = normalizeFilePath(url);
        auto data = id3v2_read(normalizedPath);
        if (!data.uri.isEmpty()) {
            songsToAdd.append(std::move(data));
        }
    }

    if (!songsToAdd.isEmpty()) {
        return addSongs(songsToAdd);
    }
    return false;
}
```

Ця функція забезпечує достатній рівень гнучкості, тому що можна додавати як файли з локальної файлової системи (через file:// URL), так і віддалені ресурси. Функція обробляє кожен URL окремо, витягує метадані через бібліотеку ID3v2, та збирає всі валідні композиції в один список для пакетного додавання. Таким чином досягається ефективність – замість окремих операцій для кожного файлу виконується одна операція для всього набору.

Також модель підтримує динамічну зміну назви плейлиста через провайдер:

```
bool PlaylistModel::setPlaylistName(const QString &playlistName)
{
    auto editable = qobject_cast<IEditablePlaylistProvider
*>(m_provider.get());
    if (!editable) {
        return false;
    }
    return editable->setPlaylistName(playlistName);
}
```

Ця операція доступна лише для редагованих списків відтворення, що забезпечує безпеку даних у випадку read-only провайдерів.

Для зручного пошуку аудіофайлів в списку відтворення було розроблено проксі клас FilterProxyModel. Він реалізує універсальну систему фільтрації для будь-яких Qt-моделей, забезпечуючи динамічний пошук за всіма доступними ролями. Самим основним в цьому класі є функція filterAcceptsRow, яка і виконує всю логіку фільтрування:

```
bool FilterProxyModel::filterAcceptsRow(int source_row, const
QModelIndex &source_parent) const {
    QHash<int, QByteArray> roles = sourceModel()->roleNames();
    for (auto it = roles.constBegin(); it != roles.constEnd(); +
+it) {QVariant data = sourceModel()->data(sourceModel()-
>index(source_row, 0, source_parent), it.key());
        if (data.toString().contains(m_filterString,
Qt::CaseInsensitive)) return true;
    }
    return false;
}
```

Ця функція перебирає всі доступні ролі моделі-джерела та перевіряє, чи містить значення кожної ролі пошуковий рядок без урахування регістру. Як тільки знайдено збіг у будь-якій ролі, функція повертає true, що означає, що рядок має бути показаний у відфільтрованому списку. Якщо жоден з атрибутів рядка не містить пошукового терміну, функція повертає false, і рядок буде приховано.

Така реалізація робить фільтрацію максимально гнучкою – незалежно від того, чи шукає користувач за назвою треку, виконавцем, альбомом чи будь-яким іншим атрибутом аудіофайлу, система знайде відповідні результати. Це особливо корисно для великих списків, де швидкий пошук значно покращує користувацький досвід та є необхідним.

Для зберігання та керування провайдерами в модулі models було створено регістр PlaylistProvidersRegistry, який виступає центральним компонентом системи динамічного завантаження плагінів. Цей клас реалізує механізм розширюваності функціоналу системи без необхідності

перекомпіляції основного застосунку, забезпечуючи гнучку архітектуру для підтримки різноманітних форматів списків відтворення.

Архітектурно регістр функціонує як посередник між основним додатком та зовнішніми плагінами провайдерів. Його основне завдання полягає у виявленні, завантаженні та управлінні життєвим циклом плагінів, які розширюють можливості програвача для роботи з різними джерелами музичного контенту. Система автоматично сканує стандартні директорії при ініціалізації, визначені через метод pluginSearchPaths:

```
QStringList PlaylistProvidersRegistry::pluginSearchPaths()
{
    return
QStringList({QStringLiteral(PLAYLIST_PROVIDER_PLUGIN_INSTALL_DIR
),
                         QCoreApplication::applicationDirPath() +
"/plugins/playlist_providers"});
}
```

Цей метод повертає список директорій які включають як системну директорію інсталяції, так і локальну теку плагінів поруч з виконуваним файлом. Змінна PLAYLIST_PROVIDER_PLUGIN_INSTALL_DIR визначається через механізми задання змінних під час компіляції системою CMake:

```
configure_file(
    "${CMAKE_SOURCE_DIR}/config.h.in"
    "${CMAKE_BINARY_DIR}/config.h"
)
```

Це дозволяє динамічно визначати шляхи до директорій плагінів на етапі збірки проекту, забезпечуючи гнучкість розгортання додатку в різних системних середовищах. Це працює таким чином, що CMake обробляє шаблонний файл config.h.in та генерує остаточний заголовковий файл config.h із підставленими значеннями реальних шляхів, що визначаються конфігурацією збірки.

Механізм підставляння використовує спеціальний синтаксис макросів препроцесора, де змінні CMake оточені символами @:

```
#define EFFECT_PLUGIN_INSTALL_DIR "@EFFECT_PLUGIN_INSTALL_DIR@"
#define PLAYLIST_PROVIDER_PLUGIN_INSTALL_DIR
"@PLAYLIST_PROVIDER_PLUGIN_INSTALL_DIR@"
#define PLUGIN_INSTALL_DIR "@PLUGIN_INSTALL_DIR@"
```

Під час виконання команди configure_file(), CMake замінює кожен маркер @VARIABLE_NAME@ на відповідне значення змінної, визначене в CMakeLists.txt або переданих параметрах збірки.

Така система забезпечує платформонезалежність збірки, оскільки шляхи можуть бути налаштовані відповідно до конвенцій конкретної операційної системи. На Linux це стандартні системні директорії /usr/lib, на Windows – директорії в Program Files, а на macOS – структури в Applications або Library. Це також дозволить легко адаптувати додаток для різних дистрибутивів Linux, кожен з яких може мати свої особливості організації файлової системи.

Списків відтворення в системі може бути безліч, а це значить, що необхідно забезпечити ефективне управління великими колекціями музичного контенту без втрати продуктивності інтерфейсу та споживання надмірних системних ресурсів. Модель PlaylistsModel розроблено з урахуванням масштабованості та оптимізації роботи з потенційно великими наборами даних через застосування ледачого завантаження та кешування. Також враховано специфіку системи провайдерів списків відтворення.

Клас PlaylistsModel успадковує від QAbstractListModel та містить повний набір властивостей для інтеграції з QML:

```
class PlaylistsModel : public QAbstractListModel {
    Q_OBJECT
    Q_PROPERTY(QString playlistsDir READ playlistsDir CONSTANT)
    Q_PROPERTY(int currentIndex READ currentIndex WRITE
setCurrentIndex NOTIFY currentIndexChanged)
```

```
    Q_PROPERTY(PlaylistModel* currentPlaylist READ
currentPlaylist NOTIFY currentPlaylistChanged)
    Q_PROPERTY(int count READ count NOTIFY countChanged)
    Q_PROPERTY(bool isEmpty READ isEmpty NOTIFY isEmptyChanged)
```

У лістингу 3.10 наведено систему ролей яка забезпечує структурований доступ до даних списків відтворення через QML інтерфейс, де кожна роль представляє конкретний аспект списку – від базових метаданих до складних об'єктів моделей пісень.

Лістинг 3.10 – Система ролей моделі PlaylistsModel

```
enum PlaylistRole {
    NameRole = Qt::UserRole + 1,
    UriRole,
    SongsModelRole
};
```

Таким чином з QML коду можна буде отримати доступ до назви списку відтворення, URI посилання та моделі з аудіофайлами.

Алгоритм завантаження плейлистів реалізує багатоетапний процес автоматичного виявлення та ініціалізації списків відтворення з файлової системи. Метод update() активує реєстр провайдерів для завантаження плагінів, після чого викликає loadPlaylists() для обробки файлів у вказаній директорії

```
void PlaylistsModel::update()
{
    m_providersRegistry->loadProviderPlugins();
    loadPlaylists(m_playlistsDir);
}
```

Це дозволяє чітко розділити відповідальність між компонентами системи, де реєстр провайдерів відповідає за управління плагінами, а модель плейлистів - за їх завантаження. Такий підхід забезпечує слабке зв'язування компонентів та спрощує тестування кожного модуля окремо.

Основний алгоритм завантаження перевіряє існування директорії та отримує список файлів, створюючи тимчасовий контейнер для нових плейлистів та отримуючи доступ до зареєстрованих фабрик провайдерів:

```
bool PlaylistsModel::loadPlaylists(const QString playlistsPath){
    QDir dir(playlistsPath);
    if (!dir.exists())
        return false;
    const QStringList files = dir.entryList(QDir::Files);
    QList<QSharedPointer<PlaylistModel>> newPlaylists;
    const auto factories = m_providersRegistry->factories();
```

Основний алгоритм перевіряє існування директорії та отримує список файлів, далі ітерує через усі зареєстровані фабрики провайдерів.

Використання патерну Factory дозволяє динамічно підтримувати різні формати плейлистів без зміни основного коду моделі. Це робить систему розширюваною та гнучкою, оскільки нові формати можуть бути додані просто через реєстрацію нових провайдерів без модифікації наявної логіки завантаження.

Процес розпізнавання форматів використовує регулярні вирази з підтримкою wildcards для порівняння імен файлів із шаблонами форматів:

```
for (const auto &factory : factories) {
    if (!factory.factory)continue;
    for (const QString &fileName : files) {
        const QString fullPath = dir.absoluteFilePath(fileName);
        for (const QVariant &formatVariant : factory.uriFormats)
{
            const QString pattern = formatVariant.toString();
            QRegularExpression re =
QRegularExpression::fromWildcard(pattern);
            if (!re.isValid() || !re.match(fileName).hasMatch())
                continue;
```

Використовується саме цей підхід, оскільки він дозволяє гнучко налаштовувати підтримувані формати через конфігурацію провайдерів, а не жорстке кодування розширень файлів. Використання wildcards забезпечує

інтуїтивний синтаксис для опису шаблонів файлів, наприклад ".m3u" або "playlist_.pls".

Після знаходження відповідності створюється екземпляр провайдера, встановлюється URI шлях до файлу та виконується завантаження даних:

```
QSharedPointer<IPlaylistProvider> provider(factory.factory-
>create());
            if (!provider || !provider->setUri(fullPath) || !
provider->load())
                continue;
            QSharedPointer<PlaylistModel> model =
                QSharedPointer<PlaylistModel>::create(provider);
            newPlaylists.append(model);
            break;
        }
    }
}
```

Використання розумних вказівників QSharedPointer забезпечує автоматичне управління пам'яттю та безпечне спільне використання об'єктів між компонентами. Послідовна перевірка кожного етапу (створення провайдера, встановлення URI, завантаження) гарантує, що до моделі потрапляють лише коректно ініціалізовані списки відтворення.

В кінці алгоритму використовуються сигнали Qt Model/View для коректного оновлення інтерфейсу:

```
if (newPlaylists.isEmpty())
    return false;
beginResetModel();
m_playlists += newPlaylists;
endResetModel();
return true;
```

Правильне використання beginResetModel() та endResetModel() забезпечує коректне оновлення всіх пов'язаних представлень без втрати стану або некоректного відображення. Це працює таким чином, що всі підключені до моделі представлення отримують сигнали про початок та завершення

зміни структури даних, що дозволяє їм правильно оновити свій стан та оновити інтерфейс.

3.2 Реалізація плагінів провайдерів

Враховуючи детально розроблену та імплементовану систему завантаження плагінів провайдерів списків відтворення, описану в попередньому розділі, логічним продовженням архітектурного рішення є реалізація самих плагінів. Система плагінів без конкретних реалізацій є лише каркасом, який не може функціонувати в реальних умовах експлуатації. Тому розробка базових плагінів провайдерів становить критично важливу частину загальної архітектури програмного забезпечення.

3.2.1 Реалізація утиліт

Центральним елементом утилітного інструментарію є CMake макрос add_playlist_provider, який інкапсулює всю логіку створення, конфігурування та встановлення плагінів провайдерів списків відтворення. Цей макрос забезпечує уніфікований підхід до створення плагінів та значно спрощує процес додавання нових провайдерів до системи.

Макрос призначений для використання в CMakeLists.txt файлах плагінів та приймає три обов'язкові параметри через іменовані аргументи. Параметр PROVIDER_NAME визначає базову назву плагіна, до якої автоматично додається суфікс "PlaylistProvider" для формування фінального імені бібліотеки. Параметр PROVIDER_SOURCES містить список всіх вихідних файлів, необхідних для компіляції плагіна, включаючи заголовочні файли та файли реалізації. Параметр INSTALL_DIR специфікує цільову директорію для встановлення скомпільованого плагіна в системі.

Детальна реалізація макросу наведена у Додатку А.

Детальна реалізація макроса наведена в додатку А. Внутрішня логіка макроса включає парсинг іменованих аргументів через ітерацію по списку

параметрів з відслідковуванням поточного режиму обробки, валідацію обов'язкових параметрів з генерацією фатальних помилок при їх відсутності, створення shared бібліотеки з автоматичним зв'язуванням необхідних Qt6 компонентів та спільної бібліотеки playlist_provider_plugin_common, конфігурування шляхів виводу для організованого розміщення згенерованих бібліотек, та налаштування правил встановлення для автоматичного копіювання файлів в цільові директорії при виконанні install target.

Практичне використання макроса демонструє його простоту та ефективність. Для створення нового плагіна необхідної включити файл з макросом та викликати його з відповідними параметрами:

```
include(cmake/PlaylistProviderPlugin.cmake)
add_playlist_provider(
  PROVIDER_NAME UPL
  PROVIDER_SOURCES UPL/UPLPlaylistProvider.h
UPL/UPLPlaylistProvider.cpp UPL/UPLPlaylistProviderFactory.cpp
  INSTALL_DIR ${PLAYLIST_PROVIDER_PLUGIN_INSTALL_DIR}
)
```

Реалізація такої системи значно спрощує процес створення нових плагінів, забезпечує консистентність структури та конфігурації проєктів, автоматизує рутинні операції збірки та встановлення, стандартизує процес тестування та валідації плагінів, та надає розробникам зручний API для швидкого старту розробки власних розширень системи. Ці інструменти є критично важливими для підтримки екосистеми плагінів та залучення сторонніх розробників до розширення функціональності аудіопрогравача.

3.2.2 Реалізація провайдерів

Провайдери плейлистів необхідно правильно реалізувати та продумати, оскільки вони є ключовими компонентами архітектури музичного плеєра, які забезпечують підтримку різноманітних форматів списків відтворення. Кожен провайдер має реалізувати підтримку для читання плейлистів, що є базовою

вимогою для функціонування системи. Підтримка запису необов'язкова, проте рекомендована, оскільки дозволяє користувачам редагувати та створювати власні плейлисти безпосередньо в додатку.

Правильна реалізація провайдерів передбачає дотримання принципів SOLID та використання стандартизованих інтерфейсів, що забезпечує інтероперабельність та розширюваність системи. Кожен провайдер повинен надавати детальну обробку помилок, підтримку метаданих пісень та ефективне управління пам'яттю. Використання патерну Factory для створення провайдерів дозволяє динамічно завантажувати підтримку нових форматів через плагінну архітектуру, що робить систему гнучкою та адаптивною до майбутніх потреб.

Спочатку постало питання вибору базового формату плейлистів за замовчуванням для системи, оскільки традиційні формати як M3U, PLS та XSPF мають суттєві обмеження у функціональності. Після аналізу доступних варіантів вибір пав на UPL (Universal Playlist) формат завдяки його унікальним перевагам. UPL формат забезпечує більшу гнучкість та повноту опису треків порівняно з існуючими форматами, дозволяючи кращий опис треків та додаткову функціональність в плеєрах. Використання JSON як основи робить формат широко сумісним між різними реалізаціями завдяки меншому набору функцій та суворішій специфікації, що веде до простіших, менших, швидших та більш сумісних бібліотек.

Ключовою перевагою UPL є система множинних ідентифікаторів, яка розв’язує фундаментальні проблеми традиційних форматів плейлистів:

```
"ids": {
  "md5": "6d522c2de22e678d590b9a2abae8f6d2",
  "sha2":
"a9a1c684b6e5d97e3bee3183a048080324dd834371e516a06ce1096b",
  "mbtrackid": "b00a2b97-53f1-485a-9121-1fe76b55e651",
  "filepath": [
    "Anciients/Following the Voice.mp3",
    "/home/user/music/Anciients/Following the Voice.mp3"
```

```
  ],
  "uri": [
    "nfs://example.com/music/ftv.mp3",
    "http://example.com/music/ftv.mp3"
  ]
}
```

Криптографічні хеші дозволяють програмам управління бібліотеками завжди знати, який запис відноситься до якого файлу, навіть якщо файли переміщуються або список відтворення синхронізується між комп'ютерами та різними плеєрами. MusicBrainz Track ID забезпечує однозначну ідентифікацію композицій незалежно від локального розташування файлів.

Для реалізації провайдера для UPL формату спочатку необхідно реалізувати фабричний клас, який забезпечить створення екземплярів провайдера через плагінну систему Qt:

```
class UPLPlaylistProviderFactory : public
IPlaylistProviderFactory{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID IPlaylistProviderFactory_iid FILE
"UPLPlaylistProvider.json")
    Q_INTERFACES(IPlaylistProviderFactory)
public:
    IPlaylistProvider *create(QObject *parent = nullptr) const
override { return new UPLPlaylistProvider(parent); }
    int priority() const override { return 1000; }
};
```

Використання фабрики дозволяє динамічно створювати провайдери через плагінну систему Qt, де макрос Q_PLUGIN_METADATA реєструє плагін у системі з посиланням на JSON-файл метаданих. Високий пріоритет (1000) гарантує першочергове використання цього провайдера серед інших можливих провайдерів для того ж формату.

Метадані плагіна містять критично важливу інформацію для роботи системи реєстрації та автоматичного розпізнавання формату:

```
{ "PluginId": "CustomPlayer.UPLPlaylistProvider",
```

```
  "UriFormatName": "UPL",
  "UriFormats": ["*.upl"] }
```

Структура метаданих забезпечує унікальну ідентифікацію плагіна через PluginId, що запобігає конфліктам при завантаженні кількох провайдерів. Поле UriFormatName надає зрозумілу назву формату для відображення в користувацькому інтерфейсі, а UriFormats містить шаблони файлів у форматі wildcards. Система реєстру використає ці метадані для автоматичного визначення відповідності файлів конкретним провайдерам та побудови списку підтримуваних форматів, що є основою для динамічного завантаження та вибору відповідного провайдера.

Заголовний файл провайдера визначає повний API для роботи з UPL-плейлистами, реалізуючи розширений інтерфейс редагування:

```
class UPLPlaylistProvider : public IeditablePlaylistProvider {
public:
    QString playlistName() const override;
    QList<QPair<QString, QStringList>> uriFormats() const;
    bool setUri(const QString &uri) override;
    int songsCount() const override;
    SongMetadata songAt(int index) const override;
    bool load(QString filePath = QString()) override;
    bool save(QString filePath = QString()) override;
    bool addSongs(const QList<SongMetadata> &songs) override;
    bool removeSong(int index) override;
    bool setPlaylistName(const QString &playlistName);};
```

Базові методи провайдера реалізують фундаментальний функціонал для ідентифікації та налаштування відповідно до специфікації UPL:

```
QString UPLPlaylistProvider::playlistName() const {
    return m_playlistName;
}
QList<QPair<QString, QStringList>>
UPLPlaylistProvider::uriFormats() const {
    return { {"UPL", {"*.upl"}} };
}
bool UPLPlaylistProvider::setUri(const QString &uri) {
```

```
    m_filePath = uri;
    return true;
}
int UPLPlaylistProvider::songsCount() const {
    return m_songs.size();
}
```

Метод uriFormats() повертає структуровану інформацію про підтримувані формати згідно з UPL специфікацією, де використовується розширення ".upl". Простота методу setUri() забезпечує швидке налаштування провайдера для роботи з конкретним файлом. Це дозволяє системі автоматично визначати можливості провайдера та правильно маршрутизувати файли до відповідних провайдерів на основі розширень файлів.

Алгоритм завантаження списку відтворення з файлу UPL формату реалізує комплексний парсинг JSON-структури відповідно до специфікації формату з детальною обробкою помилок на кожному етапі. Функція load виконує послідовну валідацію файлу: перевіряє існування шляху, відкриває файл для читання, парсить JSON-документ як масив та верифікує версію формату. У разі успішної валідації витягує назву плейлиста та ініціалізує контейнер для метаданих композицій.

Процес обробки кожного треку в плейлисті включає розбір об'єкта запису, де витягуються основні метадані (назва, виконавець, альбом, коментар, тривалість), аудіохарактеристики (канали, бітрейт, частота дискретизації), рік випуску та ідентифікатори.

У лістингу 3.11 відображено ключову частину алгоритму завантаження, де відбувається обробка масиву записів.

Лістинг 3.11 – Обробка масиву записів в алгоритмі звантаження

```
for (const QJsonValue &entry : pObj["entries"].toArray()) {
    QJsonObject eObj = entry.toObject(); SongMetadata song;
    song.title = eObj["title"].toString();
    song.artist = eObj["artist"].toString();
    song.album = eObj["album"].toString();
    song.comment = eObj["comment"].toString();
```

```
    song.length = eObj["duration"].toDouble() * 1000;
    QJsonObject audio = eObj["audio"].toObject();
    song.channels = audio["channels"].toInt();
    song.bitrate = audio["bitrate"].toInt();
    song.sampleRate = audio["sampleRate"].toInt();
    if (eObj.contains("year")) song.year = eObj["year"].toInt();
    song.uri = eObj["ids"].toObject()["filepath"].toArray()
[0].toString();
    songs.append(song);
}
```

У лістингу 3.11 зображено цикл обробки записів, де кожен елемент масиву entries перетворюється на об'єкт пісні. Тривалість конвертується з секунд у мілісекунди множенням на 1000, що відповідає внутрішньому формату зберігання часу в системі. Обов'язкова перевірка наявності поля year запобігає помилкам при обробці необов'язкових полів. Шлях до файлу визначається як перший елемент масиву filepath, що дозволяє провайдеру працювати з резервними шляхами, але використовувати лише первинний.

Після успішного парсингу всіх записів оновлюються внутрішній стан провайдера: назва плейлиста, шлях до файлу та список пісень. Фінальним етапом є викидання сигналу reset(), який сповіщає систему про повну зміну даних. Це призводить до оновлення інтерфейсу користувача та перезавантаження моделей даних.

Алгоритм збереження плейлиста у форматі UPL реалізує зворотний процес: перетворення внутрішнього стану провайдера у JSON-структуру, що відповідає специфікації. Для кожного треку в списку створюється об'єкт із повним набором метаданих, де тривалість конвертується з мілісекунд назад у секунди шляхом ділення на 1000.

У лістингу 3.12 відображено логіку формування JSON-об'єкту для окремого треку.

Лістинг 3.12 – формування JSON-об’єкту для окемого треку

```
QJsonObject eObj;
eObj["title"] = song.title; eObj["artist"] = song.artist;
```

```
eObj["album"] = song.album; eObj["comment"] = song.comment;
eObj["duration"] = song.length / 1000.0;
QJsonObject audio;
audio["channels"] = song.channels; audio["bitrate"] =
song.bitrate; audio["sampleRate"] = song.sampleRate;
eObj["audio"] = audio;
if (song.year > 0) eObj["year"] = song.year;
QJsonObject ids;
ids["filepath"] = QjsonArray({song.uri}); eObj["ids"] = ids;
```

У лістингу 3.12 продемонстровано структуру формування запису треку, де аудіопараметри об'єднуються у вкладений об'єкт audio, а шлях до файлу розміщується у масиві filepath в розділі ids. Застосування QJsonArray для filepath забезпечує сумісність зі специфікацією формату UPL, яка передбачає можливість зберігання декількох ідентифікаторів, попри те що наразі використовується тільки основний шлях до файлу.

Фінальна структура плейлиста формується як JSON-масив, що містить єдиний об'єкт з версією формату, назвою плейлиста та масивом записів:

```
QJsonObject playlist;
playlist["format"] = "UPL1";
playlist["name"] = m_playlistName;
playlist["entries"] = entries;
QJsonDocument doc({playlist});
```

Згідно зі специфікацією, файл має кореневий масив з одним об'єктом плейлиста. Така архітектура створює можливості для майбутнього розширення формату - додавання підтримки декількох плейлистів у межах одного файлу. Файл зберігається з форматуванням та відступами для забезпечення зручності читання.

Обидва процеси – завантаження та збереження - включають автоматичне перетворення одиниць часу. Це забезпечує відповідність стандарту UPL (де час вказується в секундах) та внутрішній логіці системи (яка працює з мілісекундами). Уніфікований підхід до роботи з часовими

значеннями спрощує математичні операції та процес відображення інформації.

Розроблений UPL провайдер забезпечує комплексну підтримку формату з повним дотриманням технічних вимог, оптимізованою обробкою інформації та стабільною системою контролю помилок. Це перетворює його на центральний елемент системної архітектури, здатний виконувати як стандартні операції з плейлистами, так і складні завдання управління музичною колекцією.

Крім локального управління, система також інтегрує механізми для безпечного обміну даними між пристроями. У цьому контексті мережевий провайдер P2P забезпечує захищений обмін плейлистами між пристроями в локальній мережі з використанням сучасних технологій шифрування. Реалізація базується на двоетапній архітектурі: пошук пристроїв через UDP широкомовні повідомлення та наступна передача даних через захищене TLS з'єднання.

Система використовує TLS 1.2+ для всіх з'єднань з автоматичною генерацією самопідписаних сертифікатів, що забезпечує базовий рівень довіри та шифрування без потреби в сторонній інфраструктурі сертифікації. У лістингу 3.13 наведено ключовий фрагмент ініціалізації SSL-сервера, який демонструє використання QSslServer разом із явно заданою конфігурацією безпеки.

Лістинг 3.13 – Ключовий фрагмент ініціалізації SSL

```
void P2PPlaylistProvider::setupNetwork() {
    m_sslServer = new QSslServer(this);
    QSslConfiguration sslConfig;
    sslConfig.setLocalCertificate(m_sslCertificate);
    sslConfig.setPrivateKey(m_sslPrivateKey);
    sslConfig.setProtocol(QSsl::TlsV1_2OrLater);
    m_sslServer->setSslConfiguration(sslConfig);
    m_sslServer->listen(QHostAddress::AnyIPv4, m_tcpPort);
}
```

Спочатку створюється сервер, що працює поверх SSL/TLS з обмеженням на використання протоколів версії не нижче TLS 1.2, що унеможливлює встановлення з’єднання через застарілі та небезпечні версії, як-от SSL 3.0 або TLS 1.0. Пара сертифікат/ключ встановлюється безпосередньо у QSslConfiguration, а сам сервер починає прослуховування на вказаному порту для будь-якої IPv4-адреси.

Використання QSslServer дозволяє безпосередньо інтегрувати механізми безпеки в рамках Qt-інфраструктури, забезпечуючи зручне керування параметрами з’єднання, автоматичну обробку рукостискання SSL та перевірку сертифікатів, що суттєво знижує ризик помилок при ручній реалізації криптографічних процедур.

Система автоматично створює самопідписаний сертифікат при першому запуску, якщо відповідний файл відсутній у каталозі налаштувань користувача. Це дозволяє забезпечити захищене TLS-з'єднання без залежності від зовнішніх центрів сертифікації. У лістингу 3.14 наведено фрагмент логіки генерації.

Лістинг 3.14 – Фрагмент логіки генерації сертифікату

```
void P2PPlaylistProvider::loadSslCertificate() {
    QString certPath = QStandardPaths::writableLocation(
        QStandardPaths::AppDataLocation) + "/server.crt";

    if (!QFile::exists(certPath)) {
        QSslKey key = QSslKey::generatePrivateKey(QSsl::Rsa,
2048);
        QSslCertificate cert =
QSslCertificate::generateSelfSigned(
            key, {}, {}, QSsl::Rsa, 2048,
QDateTime::currentDateTime().addYears(1));
        ...
    } else {
        ...
    }
}
```

Сертифікат створюється з ключем RSA 2048 біт, що відповідає сучасним вимогам до криптографічної стійкості. Термін дії встановлюється на один рік, це дозволяє зменшити ризики, пов’язані з тривалим використанням одного й того ж ключа, без потреби частої повторної генерації. Такий підхід є практичним компромісом між безпекою та автономністю системи, особливо в локальному мережевому середовищі без зовнішнього CA.

Система використовує комбінацію UDP для пошуку пристроїв та TLS для передачі даних:

```
void P2PPlaylistProvider::handleNewConnection() {
    QSslSocket *socket = qobject_cast<QSslSocket*>(
        m_sslServer->nextPendingConnection());
    connect(socket, &QSslSocket::encrypted, [this, socket]() {
        qDebug() << "SSL connection established";
    });
    socket->startServerEncryption();
    m_clientSockets.append(socket);
}
```

Цей метод обробляє нові клієнтські підключення до P2P-сервера. Спочатку витягується SSL-сокет з черги очікуючих з'єднань, після чого встановлюється обробник сигналу encrypted для підтвердження успішного SSL handshake. Виклик startServerEncryption() ініціює процес серверного шифрування, який забезпечує безпечний обмін ключами та встановлення захищеного каналу. В кінці, новий сокет додається до списку активних з'єднань для подальшого керування клієнтами.

Для обміну інформацією система використовує простий протокол на основі текстових команд:

```
void P2PPlaylistProvider::readClientData() {
    QSslSocket *socket = qobject_cast<QSslSocket*>(sender());
    QByteArray data = socket->readAll();
    if (data == "REQUEST_PLAYLIST") {
        sendPlaylistToClient(socket);
    }
```

```
}
```

Метод readClientData() забезпечує обробку вхідних повідомлень від підключених клієнтів. Система отримує посилання на сокет-відправник через функцію sender() та зчитує всі доступні дані методом readAll(). Розпізнавання команд відбувається через пряме порівняння отриманих байтів з еталонними текстовими значеннями.

При отриманні команди "REQUEST_PLAYLIST" система викликає відповідний обробник sendPlaylistToClient(), який формує та відправляє поточний плейлист клієнту. Текстовий формат команд забезпечує простоту налагодження та можливість ручного тестування протоколу. Архітектура дозволяє легко розширювати набір підтримуваних команд через додавання нових умовних перевірок у структурі розгалуження.

Для передачі структурованих даних використовується потужний механізм серіалізації Qt. У лістингу 3.15 відображено метод sendPlaylistToClient який відповідає за пакування композицій та їх відправку.

Лістинг 3.15 – Метод sendPlaylistToClient

```
void P2PPlaylistProvider::sendPlaylistToClient(
        QSslSocket *socket)
{
    QByteArray block;
    QDataStream stream(&block, QIODevice::WriteOnly);
    stream << m_playlistName << m_tracks.size();
    for (const SongMetadata &song : m_tracks) {
        stream << song;
    }
    socket->write("PLAYLIST_DATA:" + block);
}
```

Метод sendPlaylistToClient() реалізує серіалізацію плейлиста для передачі клієнту. Система створює буфер QByteArray та пов'язує з ним потік QDataStream у режимі запису. Серіалізація починається з запису назви

плейлиста та загальної кількості треків, що дозволяє клієнту підготувати структури для отримання даних.

Далі система послідовно записує метадані кожного треку через оператор потокового виводу. Після серіалізації готовий блок даних передається через захищене з'єднання з префіксом "PLAYLIST_DATA:", який дозволяє клієнту розпізнати тип повідомлення. Використання QDataStream гарантує незалежність від архітектури процесора та операційної системи при обміні даними між різними пристроями.

Оскільки мережевий провайдер є значно складнішим за прості провайдери локальних форматів і потребує налаштування параметрів з'єднання, моніторингу статусу підключених пристроїв та управління процесом обміну даними, система реалізує спеціалізований користувацький інтерфейс.

У лістингу 3.16 відображено метод providerUi який відповідає за динамічне створення інтерфейсу провайдера.

Лістинг 3.16 – Метод providerUi

```
QObject *P2PPlaylistProvider::providerUi(QObject *parent) {
    QQmlEngine *engine = qmlEngine(parent);
    QQmlComponent component(engine,
        QUrl("qrc:/P2PProviderUI.qml"));
    return component.createWithInitialProperties(
        {{"provider", QVariant::fromValue(this)}}, parent);
}
```

Метод providerUi() забезпечує створення графічного інтерфейсу провайдера через механізм QML-компонентів. Система отримує посилання на «двигун» QML з батьківського об'єкта та завантажує опис інтерфейсу з ресурсного файлу (qrc). Використання QQmlComponent дозволяє динамічно інстанціювати користувацький інтерфейс під час виконання програми.

Ключовою особливістю реалізації є передача поточного об'єкта провайдера як властивості «provider» через метод createWithInitialProperties(). Це забезпечує двосторонній зв'язок між QML-інтерфейсом та логікою провайдера, дозволяючи інтерфейсу викликати методи провайдера та отримувати сповіщення про зміни стану. Така архітектура дотримується принципів Model-View-ViewModel та забезпечує чітке розділення презентаційного рівня від бізнес-логіки.

Враховуючи асинхронний характер мережевих операцій та необхідність підтримки великої кількості одночасних з'єднань, система потребує грамотного керування життєвим циклом сокетів для запобігання накопиченню неактивних ресурсів. У лістингу 3.17 продемонстровано метод cleanupClients, який відповідає за очищення з'єднань.

Лістинг 3.17 – Функція cleanupClients

```
void P2PPlaylistProvider::cleanupClients() {
    auto it = m_clientSockets.begin();
    while (it != m_clientSockets.end()) {
        if ((*it)->state() != QTcpSocket::ConnectedState) {
            (*it)->deleteLater();
            it = m_clientSockets.erase(it);
        } else { ++it; }
    }
}
```

Метод cleanupClients() виконує періодичне очищення списку клієнтських з'єднань від неактивних сокетів. Система застосовує безпечний ітеративний підхід з перевіркою стану кожного сокета через властивість state(). При виявленні сокета у стані, відмінному від ConnectedState, система ініціює процес його видалення.

Використання deleteLater() замість прямого виклику деструктора забезпечує відкладене знищення об'єкта до наступної ітерації циклу подій Qt. Це гарантує коректне завершення всіх поточних операцій сокета та запобігає появі висячих покажчиків. Після планування видалення система видаляє

сокет зі списку через erase() та коригує ітератор, уникаючи пропуску наступних елементів. Такий підхід забезпечує ефективне управління пам'яттю в умовах інтенсивного створення та знищення мережевих з'єднань.

3.3 Реалізація аудіоефектів

Оскільки система обробки аудіоефектів базується на плагінній архітектурі для забезпечення динамічного розширення функціоналу без модифікації основного коду, необхідно реалізувати базові інтерфейси для взаємодії компонентів. Система потребує впровадження інтерфейсу IAudioEffect для стандартизації роботи з аудіоефектами.

Першим реалізованим ефектом є реверберація, яка моделює акустичні властивості приміщення через створення контрольованого відлуння аудіосигналу. Цей ефект потребує складних обчислень з використанням буферів затримки та алгоритмів змішування для створення природного звучання просторового середовища.

Реалізація ефекту реверберації у вигляді динамічної бібліотеки супроводжується JSON-файлом метаданих для інтеграції з системою завантаження плагінів:

```
{
  "PluginId": "CustomPlayer.ReverbEffect"
}
```

Структура метаданих містить ідентифікатор PluginId, який використовується для унікальної ідентифікації плагіна в системі під час процесу завантаження.

Аудіоефект успадковується від інтерфейсу IAudioEffect та використовує систему плагінів Qt для динамічної реєстрації в додатку:

```
class ReverbEffect : public IAudioEffect {
    Q_OBJECT
    Q_PLUGIN_METADATA(IID IAudioEffect_iid FILE
"ReverbEffect.json")
```

```
    Q_INTERFACES(IAudioEffect)
public:
    ReverbEffect(QObject *parent = nullptr);
    QString effectName() const override { return "Reverb"; }
    bool applyEffect(char *data, qint64 len, const QAudioFormat
&format) const override;
    QList<QSharedPointer<EffectParameter>> parameters() const
override;
};
```

Клас ReverbEffect реалізує повний інтерфейс аудіоефекту через наслідування від IAudioEffect. Макрос Q_PLUGIN_METADATA забезпечує зв'язування плагіна з його метаданими через вказівку на JSON-файл та ідентифікатор інтерфейсу. Декларація Q_INTERFACES інформує систему мета-об'єктів Qt про реалізовані інтерфейси, що необхідно для коректного приведення типів під час завантаження плагіна.

Кожен ефект визначає набір керованих параметрів через створення об'єктів EffectParameter з чітко визначеними діапазонами значень та типами даних:

```
m_decayTime = QSharedPointer<EffectParameter>::create(
    "Decay", 2.0, 0.1, 5.0, EffectParameter::Float, this);
m_damping = QSharedPointer<EffectParameter>::create(
    "Damping", 0.5, 0.0, 1.0, EffectParameter::Float, this);
m_wetLevel = QSharedPointer<EffectParameter>::create(
    "Wet", 0.3, 0.0, 1.0, EffectParameter::Float, this);
m_roomSize = QSharedPointer<EffectParameter>::create(
    "Room", 0.7, 0.0, 1.0, EffectParameter::Float, this);
```

В конструкторі створюються чотири ключові параметри реверберації з різними функціональними призначеннями. Параметр m_decayTime контролює час загасання відлуння з діапазоном від 0.1 до 5.0 секунд, що охоплює спектр від невеликих приміщень до великих залів. Параметр m_damping регулює демпфування високочастотних компонентів сигналу в

діапазоні від 0.0 до 1.0, моделюючи поглинання звуку різними матеріалами поверхонь.

Параметр m_wetLevel визначає пропорцію оброблюваного сигналу у фінальному міксі, дозволяючи плавно регулювати інтенсивність ефекту від повністю сухого до повністю обробленого звуку. Параметр m_roomSize моделює акустичні характеристики розміру приміщення, впливаючи на характер та густину ранніх відбиттів.

Використання QSharedPointer забезпечує автоматичне управління життєвим циклом параметрів та безпечний доступ з різних частин системи без ризику витоку пам'яті або доступу до звільнених ресурсів.

Ефект реверберації реалізується через комбінацію комб-фільтрів та алпас-фільтрів для моделювання складних акустичних процесів відбиття звуку в приміщенні:

```
void updateFilters(float sampleRate) const {
    const size_t combDelaysMs[] = {50, 56, 61, 68, 72, 78, 84,
90};

m_combs.resize(sizeof(combDelaysMs)/sizeof(combDelaysMs[0]));
    for (size_t i = 0; i < m_combs.size(); ++i) {
        size_t delaySamples = static_cast<size_t>(
            combDelaysMs[i] * sampleRate / 1000.0f);
        m_combs[i].buffer.resize(delaySamples, 0.0f);
    }
    // Аналогічно для алпас-фільтрів...
}
```

Система використовує вісім комб-фільтрів з різними затримками від 50 до 90 мілісекунд для створення густої структури відлунь. Кожен фільтр конвертує затримку з мілісекунд у кількість семплів відповідно до частоти дискретизації аудіосигналу через формулу:

$$delaySamples = delayMs * sampleRate / 1000.$$

Буфери затримки ініціалізуються нульовими значеннями для забезпечення чистого початкового стану.

Ефект реверберації імітує природне загасання звуку в приміщенні через комбінацію двох типів фільтрів. Комб-фільтри створюють серію затухаючих повторень за алгоритмами: output = buffer[n] + input * (1 — damp) та buffer[n] = input + output * feedback, де параметр демпфування контролює загасання високих частот, а зворотний зв'язок визначає інтенсивність відлунь.

Алпас-фільтри корегують частотну характеристику через обчислення output = -input + buffer[n] та buffer[n] = input + buffer[n] * 0.5, згладжуючи резонансні піки та створюючи більш природне звучання реверберації.

Основна логіка ефекту реалізується в методі applyEffect, який виконує послідовну обробку кожного аудіосемплу через ланцюг фільтрів:

```
for (int i = 0; i < totalSamples; ++i) {
    float input = samples[i];
    float combSum = 0.0f;
    for (auto &comb : m_combs)
        combSum += processComb(comb, input);
    combSum *= 0.1f;
    float allpassOut = combSum;
    for (auto &allpass : m_allpasses)
        allpassOut = processAllpass(allpass, allpassOut);
    samples[i] = input * (1.0f - wet) + allpassOut * wet * room;
}
```

Алгоритм обробки виконує багатоетапну трансформацію кожного семплу аудіосигналу. Спочатку вхідний сигнал надходить до всіх комб-фільтрів паралельно, де кожен фільтр генерує власну версію відлуння з унікальною затримкою та характеристикою загасання. Сума виходів комб-фільтрів масштабується коефіцієнтом 0.1 для запобігання перенасиченню та забезпечення стабільного рівня сигналу.

Результуючий сигнал послідовно проходить через ланцюг алпас-фільтрів, кожен з яких додатково корегує частотну характеристику та згладжує резонансні артефакти комб-фільтрів. Фінальне змішування поєднує

оригінальний сухий сигнал з обробленим відповідно до параметрів wet та room, де перший контролює співвідношення між сухим та обробленим сигналом, а другий модулює загальну інтенсивність ефекту залежно від модельованого розміру приміщення. Така архітектура забезпечує реалістичне відтворення акустичних властивостей різних просторових середовищ.

За аналогічними принципами архітектури плагінів, інтерфейсної абстракції та параметричного управління система реалізує додаткові аудіоефекти з різними алгоритмічними підходами. Фільтр нижніх частот застосовує цифрову фільтрацію для селективного підсилення низькочастотного спектра сигналу. Шістнадцятисмуговий еквалайзер використовує банк смугових фільтрів для незалежного регулювання амплітуди окремих частотних діапазонів аудіосигналу. Ефект підсилення Gain реалізує лінійне масштабування амплітуди сигналу з контролем динамічного діапазону.

Кожен з цих ефектів наслідує базову архітектуру через реалізацію інтерфейсу IAudioEffect, використання системи параметрів EffectParameter та інтеграцію з механізмом динамічного завантаження плагінів. Програмні реалізації цих ефектів детально представлені в додатках Б, В та Г відповідно, демонструючи різноманітність алгоритмічних підходів при збереженні уніфікованої архітектурної основи системи обробки звуку.

3.4 Реалізація користувацького інтерфейсу

Для реалізації користувацького інтерфейсу було обрано декларативну мову QML, оскільки проєкт побудований на базі фреймворку Qt та потребує створення сучасного адаптивного інтерфейсу. QML забезпечує розділення логіки додатку від презентаційного рівня, підтримує апаратне прискорення графіки через OpenGL, дозволяє створювати складні анімації та переходи, а також надає вбудовані засоби для адаптації під різні розміри екранів та роздільні здатності.

Користувацький інтерфейс реалізується через ієрархічну структуру QML-компонентів відповідно до архітектурних рішень, визначених у розділі проєктування інтерфейсу. Система забезпечує модульність та повторне використання елементів через компонентний підхід розробки. Основний файл main.qml визначає структуру додатку, яка включає бічну панель навігації, область контенту та панель керування відтворенням згідно з запроєктованою архітектурою інтерфейсу. Лістинг 3.4.1 демонструє програмний код ключової структури головного вікна.

Лістинг 3.17 – Програмний код структури головного вікна

```
ApplicationWindow {
    ColumnLayout {
        RowLayout {
            Rectangle { // Бічна панель
                Sidebar { /* ... */ }
}
            Loader {id: pageLoader}}
        ControlsPanel { // Панель керування відтворенням
}
    }
}
```

Архітектура інтерфейсу базується на вертикальному макеті ColumnLayout, що містить горизонтальний рядок з бічною панеллю та областю контенту через RowLayout, а також панель керування у нижній частині вікна. Використання компонента Loader забезпечує динамічне завантаження різних сторінок додатка без повного перезавантаження інтерфейсу, що підвищує продуктивність та забезпечує плавні переходи між розділами програми. Загальний вигляд реалізованого інтерфейсу представлено на рисунку 3.1

Рисунок 3.1 – Реалізований інтерфейс

Головний макет враховує платформу запуску для адаптації інтерфейсу відповідно до специфіки різних операційних систем та типів пристроїв:

```
readonly property bool isMobile: (Qt.platform.os === "android"
|| Qt.platform.os === "ios")
width: isMobile ? Screen.width : Screen.width * 0.8
height: isMobile ? Screen.height : Screen.height * 0.8
```

Система визначає тип платформи через порівняння значення Qt.platform.os з ідентифікаторами мобільних операційних систем. Властивість isMobile встановлюється як константа тільки для читання, що забезпечує стабільність поведінки інтерфейсу протягом всього життєвого циклу програми. На мобільних пристроях додаток займає повну площу екрана для максимального використання обмеженого простору дисплея.

На десктопних платформах система зменшує розміри вікна до 80% від розмірів екрана, забезпечуючи оптимальний користувацький досвід без необхідності створення окремих версій інтерфейсу. Адаптивне масштабування відбувається автоматично при запуску програми на основі детектованих характеристик операційного середовища.

Нижня частина інтерфейсу містить панель керування відтворенням, яка об'єднує основні елементи управління медіаплеєром (рис. 3.2).

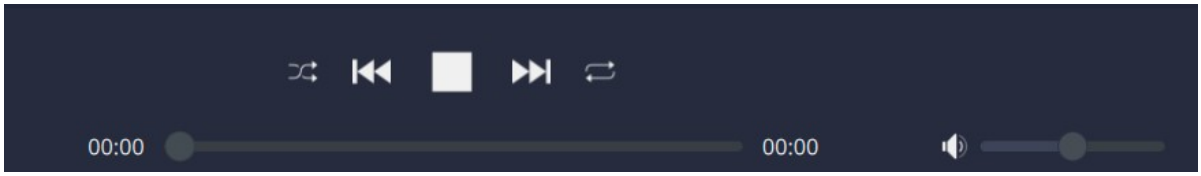

Рисунок 3.2 – Панель керування відтворенням

Панель включає кнопки навігації "назад" та "вперед" для переключення між треками у плейлисті, перемикачі режимів "повтор" та "перемішування" для контролю послідовності відтворення. Центральну позицію займає універсальна кнопка управління відтворенням, яка динамічно змінює свій вигляд та функціональність залежно від поточного стану плеєра, відображаючи іконки "програвати", "пауза" або "стоп" відповідно до активного режиму:

```
icon.name: {
    switch(MusicPlayer.playbackState) {
    case MusicPlayer.PlayingState: return "media-playback-pause"
    case MusicPlayer.PausedState: return "media-playback-start"
    case MusicPlayer.StoppedState: return "media-playback-stop"
    }
}
```

У реалізації інтерфейсу додатку активно використовується система іконок Freedesktop.org, яка дозволяє встановлювати іконки за їх стандартизованими назвами без необхідності зберігання власних графічних ресурсів. Ця стандартизована система іконок гарантує коректне відображення елементів управління в різних середовищах робочого столу та автоматичну адаптацію до налаштувань користувача.

Під рядом кнопок управління розташований слайдер позиціювання, який забезпечує візуальне відображення прогресу відтворення та дозволяє користувачу перемотувати трек до бажаної позиції. Слайдер супроводжується

текстовими індикаторами часу: зліва відображається поточна позиція відтворення у форматі хвилини:секунди, справа показується загальна тривалість активного треку. Така організація елементів забезпечує інтуїтивний контроль над процесом відтворення та надає користувачу повну інформацію про стан програвання медіаконтенту.

Лівий сайдбар програми містить організовану структуру функціональних елементів для керування контентом та налаштуваннями. У верхній частині панелі розміщується список доступних плейлистів з можливістю швидкого перемикання між ними. Центральна область включає розділи налаштувань додатка та управління аудіоефектами, що забезпечують доступ до основних параметрів конфігурації системи. У нижній частині панелі знаходиться елемент керування для створення нових плейлистів, що дозволяє користувачам розширювати колекцію музичного контенту безпосередньо з головного інтерфейсу. Інтерфейс лівого сайдбару зображено на рисунку 3.3.

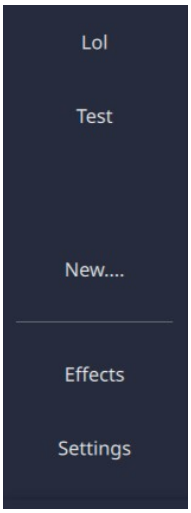

Рисунок 3.3 – Лівий сайдбар

Головним компонентом системи є списки аудіофайлів, які реалізуються через поєднання QML-компонента ListView та C++-моделі PlaylistModel. Ця архітектура забезпечує ефективне відображення великих колекцій музичних

треків з підтримкою віртуалізації та плавної прокрутки. Модель PlaylistModel наслідується від QAbstractListModel та інкапсулює логіку управління даними, включаючи завантаження метаданих файлів, сортування та фільтрацію елементів.

Компонент ListView виконує роль представлення даних з автоматичним створенням та знищенням візуальних елементів залежно від видимої області інтерфейсу. Такий підхід дозволяє системі обробляти плейлисти з тисячами треків без значного впливу на продуктивність інтерфейсу. Взаємодія між QML-компонентом та C++-моделлю відбувається через систему ролей Qt:

```
    PlaylistsModel {
        id: playlistsModel}
...
        SongsListView {
            model: playlistsModel.currentPlaylist;}
```

Ця архітектура реалізує патерн Model-View-Presenter з чітким розділенням відповідальності між логікою даних та презентаційним рівнем. Система автоматично синхронізує зміни в моделі з відображенням у компонентах через механізм сигналів та слотів Qt. Віртуалізація ListView забезпечує створення лише тих візуальних елементів, які потрапляють у видиму область екрана, що дозволяє ефективно працювати з великими наборами даних при мінімальному споживанні оперативної пам'яті.

ListView відображає делегат для кожного елемента моделі даних, що забезпечує уніфікований підхід до представлення інформації незалежно від типу контенту. Система делегатів дозволяє створювати гнучкі та переносимі компоненти, які можуть адаптуватися до різних структур даних без зміни основної логіки відображення. Такий підхід забезпечує можливість використання одного делегата для різних типів списків та спрощує підтримку консистентності візуального стилю в межах всього додатка.

У додатку Ґ відображено лістинг програмного коду делегату основного ListView, який відповідає за відображення списку аудіофайлів у головному

інтерфейсі програми. Делегат реалізує інтерактивний елемент списку з повним набором функціональних можливостей для взаємодії з музичними композиціями.

Цей компонент використовує складну структуру з RowLayout для горизонтального розташування елементів, включаючи обкладинку альбому через RoundedImage, текстову інформацію про композицію через ColumnLayout з двома Text елементами, та правостороннє розташування тривалості треку з кнопкою обраного. Система керування станом реалізується через властивості моделі даних, що дозволяє динамічно оновлювати відображення інформації про кожну композицію.

Інтерактивність забезпечується через MouseArea з підтримкою подвійного кліку для запуску відтворення та ховер-ефектів для візуального зворотного зв'язку. Анімаційні переходи кольорів створюють плавний користувацький досвід при наведенні миші та виборі елементів списку.

Зовнішній вигляд делегату списку з аудіофайлами зображено на рисунку 3.4.

Рисунок 3.4 – Делегат списку з аудіофайлами

Для конвертації мілісекунд в секунди використовується функція msecToDuration з модуля «common» класу Utils.

Також програма містить сторінки з налаштуваннями ефектів та загальних налаштувань програми. Сторінка з ефектами реалізується як список налаштовуваних елементів через елемент типу Repeater, який

динамічно створює інтерфейсні компоненти для кожного завантаженого аудіоефекту.

Система генерує окремі секції налаштувань для кожного ефекту з відповідними елементами контролю параметрів, що базуються на типах EffectParameter (рис. 3.5).

Рисунок 3.5 – Секція нелаштувань ефекта

Числові параметри відображаються через слайдери з точним контролем діапазону значень, логічні параметри через перемикачі, а текстові через поля введення. Кожен елемент керування автоматично прив'язується до відповідного параметра ефекту через систему властивостей Qt, забезпечуючи миттєве оновлення звукової обробки при зміні налаштувань.

Кожна секція також містить перемикач який дозволяє швидко вимикати конкретний ефект зі збереженням його стану після повторного ввімкнення. Інтерфейс реалізовано таким чином що якщо ефект вимкнути, то його секції з параметрами приховуються, це дозволяє значно зекономити місце та спростити пошук ефектів.

Сторінка загальних налаштувань організовується через групи функціональних параметрів включно з налаштуваннями аудіовиходу, поведінки інтерфейсу, управління бібліотекою та системи сповіщень. Всі налаштування зберігаються через механізм QSettings та автоматично відновлюються при наступному запуску програми.

Також для пошуку в списку плейлистів, налаштувань чи ефектів реалізовано спеціалізований елемент пошуку на основі TextField з розширеною функціональністю та адаптивним інтерфейсом. Компонент

інтегрує візуальні індикатори та інтерактивні елементи для забезпечення зручного користувацького досвіду при фільтрації контенту. Зовнішній вигляд цього елементу відображено на рисунку 3.6.

Рисунок 3.6 – Елемент пошуку

При перебуванні в якомусь плейлисті зверху зліва розташовується кнопка додавання аудіофайлів файлів (рис. 3.7).

Рисунок 3.7 – Кнопка додавання аудіофайлів

При натисканні на цю кнопку відкривається діалогове вікно множинного вибору аудіофайлів підтримуваних форматів через використання нативного системного інтерфейсу операційної системи. Система автоматично адаптується до платформи виконання, забезпечуючи користувачам звичний інтерфейс відповідно до стандартів їхньої операційної системи.

Для створення нового плейлиста система реалізує спеціалізоване діалогове вікно з динамічною архітектурою, яка дозволяє інтегрувати специфічні інтерфейси налаштувань від різних провайдерів плейлистів. Діалог автоматично адаптується до типу обраного провайдера через вбудовування його користувацького інтерфейсу (рис. 3.8), створеного методом providerUi(), що забезпечує гнучкість налаштування параметрів для кожного типу плейлиста. Лістинг програмного коду цього діалогу наведено в додатку Д.

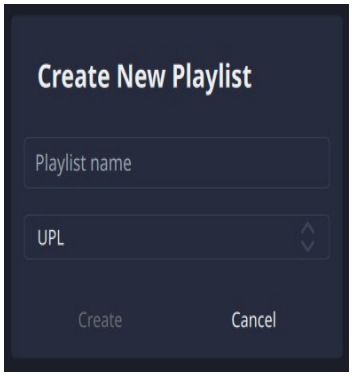

Рисунок 3.8 – Діалог створення плейлиста

3.5 Тестування програмного засобу

Тестування є невід'ємною частиною процесу розробки програмного засобу, що забезпечує перевірку коректності реалізації функціональних вимог та стабільності роботи системи в різних умовах експлуатації.

Функціональне тестування проводилось з метою перевірки відповідності програмного забезпечення встановленим функціональним вимогам та специфікаціям.

Спочатку було перевірено коректність роботи функції пошуку в додатку. Тестування здійснювалось шляхом введення назви конкретної пісні в поле пошуку з подальшою перевіркою відповідності отриманих результатів запиту. В результаті виконання тестового сценарію після введення назви композиції система успішно відобразила саме ту пісню, яку було запитано, що підтверджує правильність реалізації алгоритму пошуку та його здатність точно ідентифікувати та повертати релевантні результати. Коректність функціонування пошукової системи додатку демонструється на рисунку 3.9, де відображено успішне знаходження та відображення запитуваної музичної композиції відповідно до введених користувачем критеріїв пошуку.

Рисунок 3.9 – Перевірка функціонування систем пошуку

Далі було перевірено коректність роботи системи аудіоефектів шляхом вмикання аудіоефекту підсилення низьких частот та перевірки змін на слух та за допомогою програмного забезпечення Carla із набором плагінів LV2. Для забезпечення більш точного та наочного тестування було використано звук генератора замість музичних композицій, що дозволило краще візуалізувати та проаналізувати різницю у спектральних характеристиках на графіку до та після застосування аудіоефектів.

Спочатку було налаштовано програмне забезпечення Carla, адресуючи звук з виходу програми на входи плагіну LV2. Після чого було запущено звук спочатку без застосування аудіоефектів, а потім з активованим ефектом підсилення низьких частот. В результаті проведеного тестування було проаналізовано показники плагіну та порівняно спектральні характеристики сигналу у двох режимах роботи.

Тестування підтвердило коректність функціонування системи аудіоефектів – було зафіксовано чітке підсилення амплітуди у низькочастотному діапазоні при активації відповідного ефекту, що повністю відповідає очікуваній поведінці системи. Аудіоефект працює стабільно, забезпечуючи якісну обробку звукового сигналу згідно з налаштованими параметрами. Результат тестування зображено на рисунку 3.10.

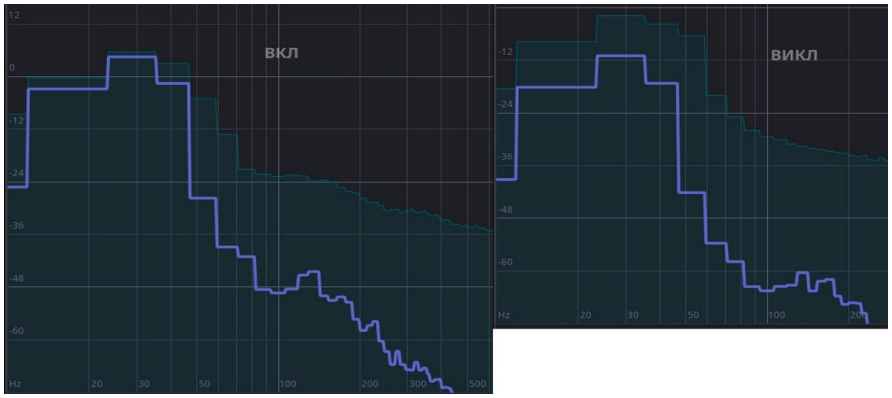

Рисунок 3.10 – Результати тестування системи ефектів

Далі було протестовано функцію додавання аудіофайлів до списку відтворення. Тестування проводилось поетапно: спочатку було натиснуто на кнопку "Додати файли", що призвело до відкриття стандартного діалогового вікна вибору файлів операційної системи. У відкритому вікні було здійснено вибір декількох аудіофайлів різних форматів та підтверджено вибір натисканням відповідної кнопки.

В результаті виконання тестового сценарію всі вибрані аудіофайли успішно з'явились у списку відтворення додатку (рис. 3.11), що підтверджує правильність реалізації функції імпорту медіафайлів. Система коректно обробила процес додавання файлів, забезпечивши їх відображення з правильними назвами та метаданими у користувацькому інтерфейсі програми. Функціональність додавання аудіофайлів працює стабільно та відповідає очікуваним вимогам.

Рисунок 3.11 – Додавання аудіофайлу в список

Також було протестовано функцію додавання (створення) плейлиста. Тестування здійснювалось шляхом натискання на кнопку "Новий плейлист", після чого в діалоговому вікні було обрано формат UPL (Universal Playlist) для створення плейлиста.

В результаті виконання операції новий порожній плейлист успішно з'явився у вікні програми (рис. 3.12), що підтверджує коректність функціонування системи управління плейлистами. Програма правильно обробила запит на створення плейлиста у вказаному форматі та відобразила його в користувацькому інтерфейсі у вигляді пустого контейнера, готового для подальшого наповнення аудіофайлами. Функція створення плейлистів працює стабільно та забезпечує підтримку стандартного формату UPL.

Далі було перевірено функціональність пошуку в P2P провайдері. Тестування проводилось у два етапи для комплексної перевірки роботи піринговой мережі.

Першим етапом у вікні додавання плейлиста було ініційовано сканування локальної мережі для виявлення доступних P2P клієнтів.

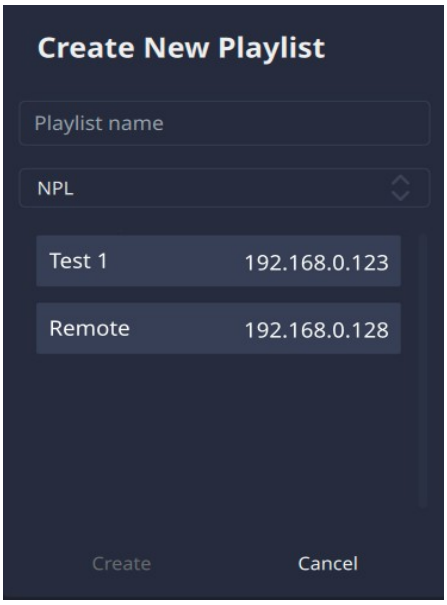

Рисунок 3.13 – Список пірів в локальній мережі

В результаті виконання цієї операції у вікні програми успішно відобразились всі активні локальні клієнти (рис 3.13), що знаходяться в межах мережевого сегмента, підтверджуючи коректність функціонування механізму автоматичного виявлення пірів.

Другим етапом було здійснено спробу підключення до виявлених клієнтів та отримання списку доступних аудіофайлів. Тестування показало успішне встановлення з'єднання з віддаленими клієнтами та отримання переліку музичних композицій, які знаходяться у їхніх колекціях (рис. 3.14).

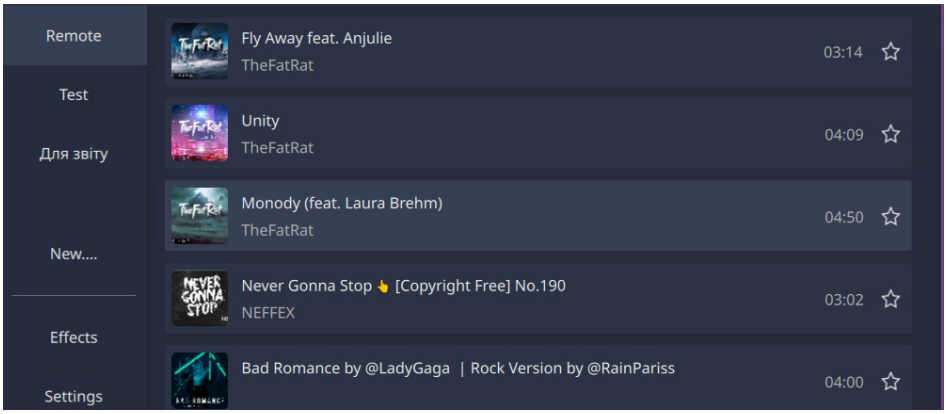

Рисунок 3.14 – Отриманий віддалений плейлист

Отже, було проведено комплексне функціональне тестування програмного засобу, яке охопило всі ключові компоненти системи. Результати тестування підтвердили повну функціональність розробленого програмного засобу та його відповідність встановленим вимогам.

Тестування показало коректну роботу функції пошуку, стабільне функціонування системи аудіоефектів з коректною обробкою звукового сигналу, надійне управління медіафайлами та плейлистами. Особливо важливим є успішне функціонування P2P провайдера, який забезпечує виявлення локальних клієнтів та обмін музичним контентом.

Усі протестовані функції демонструють стабільну роботу та відповідають технічним специфікаціям, що свідчить про готовність програмного засобу до практичного використання.

4 ТЕХНІЧНО-ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ

4.1 Аналіз ринку

Пропонований програмний продукт є кросплатформним аудіопрогравачем із розширеними функціями безпечного обміну даними через мережу. Він забезпечує підтримку основних операційних систем (Linux, Windows, macOS) та мобільних платформ завдяки використанню фреймворку Qt6, що робить його універсальним рішенням для різних пристроїв. Ключові експлуатаційні характеристики включають вбудований високоточний еквалайзер, шифрування аудіоконтенту під час передачі за допомогою алгоритмів AES-256 і TLS, а також інтуїтивно зрозумілий інтерфейс з адаптивним дизайном. Відкритий код проєкту, розповсюджений за ліцензією GPLv3, дозволяє спільноті впливати на його розвиток, що є суттєвою перевагою порівняно з закритими аналогами.

Хоча програмний продукт не є повністю новим на ринку, він пропонує унікальне поєднання функцій, які відсутні в наявних рішеннях. На відміну від традиційних аудіопрогравачів (наприклад, Groove чи Audacious), акцент зроблено на захисті даних під час мережевого обміну та повній кросплатформності, включаючи мобільні пристрої. Це робить його привабливим для широкого кола користувачів: від приватних осіб, які цінують конфіденційність, до організацій, що потребують безпечного внутрішнього обміну аудіоконтентом.

Глобальний ринок збуту охоплює країни з високими вимогами до кібербезпеки (ЄС, США), а також ніші, де переважають open-source рішення (спільноти Linux-користувачів). Попит на подібний продукт зростає через збільшення кількості кібератак (за даними Cybersecurity Ventures, на 15% щорічно 8) та зростання популярності відкритих технологій. Зокрема, в ЄС понад 60% державних установ вже використовують open-source програмне

забезпечення, а деякі країни навіть зобов’язують це на законодавчому рівні 9, що відкриває додаткові можливості для впровадження проєкту.

Реалізація продукту передбачає безплатне розповсюдження через платформи GitHub та F-Droid, а також розміщення у таких репозиторіях як pacman, apt, chocolatey тощо, для залучення більшої аудиторії. Сервісна підтримка базується на спільноті: документація, форуми та зворотний зв'язок через GitHub або GitLab. Для корпоративних клієнтів можливе надання платних послуг (наприклад, кастомізація або пріоритетна техпідтримка).

Головними конкурентами є Groove, Audacious та MusicBee. Переваги пропонованого рішення полягають у поєднанні безпеки, кросплатформності та відкритості коду, що відповідає сучасним вимогам до програмних продуктів. На відміну від конкурентів, які зосереджені лише на відтворенні, цей проєкт інтегрує інструменти для спільного використання контенту, що розширює його застосовність.

4.2 Розрахункова частина

4.2.1 Розрахунок витрат на розробку

Розробка програмного продукту включає комплексні витрати, пов'язані з оплатою праці, податками, закупівлею обладнання та іншими операційними витратами. Кожен компонент розрахований з урахуванням специфіки проєкту, зокрема необхідності тестування на різних платформах і забезпечення високої аудіоякості. Процес розробки передбачає багатоетапний підхід, що включає аналіз вимог, проєктування архітектури, реалізацію функціоналу та комплексне тестування системи.

Команда проєкту складається з трьох фахівців: мережевого інженера, full-stack розробника та QA-інженера. Мережевий інженер відповідає за розробку мережевої інфраструктури та оптимізацію передачі даних, full-stack розробник займається створенням мережевої (P2P) частини додатка, а QA-інженер здійснює комплексне тестування продукту та забезпечує

відповідність встановленим стандартам якості. Сумарні витрати враховують п'ятимісячний період розробки, протягом якого планується повна реалізація всіх функціональних можливостей програмного продукту. В таблиці 4.1 наведено деталізацію витрат на оплату праці кожного члена команди за увесь період роботи з указанням кількості годин, погодинних ставок та основної заробітної плати.

Таблиця 4.1 – Заробітна плата працівників

Посада	Погодинна ставка, грн	Кількість відпрацьованих годин, год	Заробітна плата, грн
Full-stack розробник	650	400	260 000
Мережевий інженер	450	200	90 000
QA-інженер	220	150	33 000
Загалом			383 000

При розрахунку повної вартості робочої сили підприємства необхідно враховувати не лише базову заробітну плату працівників, але й усі обов'язкові податкові відрахування, які суттєво збільшують загальні витрати на персонал. Як показано в таблиці 4.2, до таких відрахувань належать податок на доходи фізичних осіб (ПДФО) у розмірі 18%, військовий збір у розмірі 5% та єдиний соціальний внесок (ЄСВ) у розмірі 22% від фонду оплати праці.

Враховуючи чинну податкову систему України, роботодавець несе додаткове навантаження у вигляді єдиного соціального внеску, який становить значну частку від заробітної плати та спрямовується на фінансування загальнообов'язкового державного соціального страхування.

Таблиця 4.2 – Витрати на податки

Посада	ПДФО, грн	ЄСВ, грн	Військовий збір, грн	Загалом, грн
Full-stack розробник	46 800	57 200	13 000	117 000
Мережевий інженер	16 200	19 800	4 500	40 500
QA-інженер	5 940	7 260	1 650	14 850
Загалом	68 940	84 260	19 150	172 350

Згідно з розрахунками, фактичні витрати підприємства на утримання працівника значно перевищують його базову заробітну плату через необхідність сплати ЄСВ, який додається до основної суми. При цьому ПДФО та військовий збір утримуються із заробітної плати працівника, зменшуючи суму, яку він отримує на руки, але не впливаючи на загальні витрати роботодавця.

Енерговитрати становлять певну частину витрат при розробці програмного продукту та включають споживання електроенергії всім технічним обладнанням проєкту. Розрахунок виконано з урахуванням чинних тарифів Національної комісії, що здійснює державне регулювання у сферах енергетики та комунальних послуг, на 2025 рік, які становлять 2,64 гривні за кіловат-годину. При розрахунках також слід врахувати використання джерел безперебійного живлення з коефіцієнтом корисної дії 92%, що призводить до додаткових втрат електроенергії.

Розподіл обладнання між членами команди здійснено відповідно до специфіки їх робочих завдань, як показано в таблиці розподілу обладнання. Full-stack розробник та мережевий інженер використовують персональні комп'ютери з підвищеною продуктивністю для виконання складних обчислювальних завдань, тоді як QA-інженер працює на ноутбуці, що забезпечує необхідну мобільність для тестування на різних платформах.

Кожне робоче місце має бути обладнано відповідними моніторами для забезпечення комфортних умов праці.

Таблиця 4.3 – Розподіл обладнання між членами команди

Спеціаліст	Обладнання	Час роботи, год
Full-stack розробник	Комп'ютер + монітор	400
Мережевий інженер	Комп'ютер + монітор	200
QA-інженер	Ноутбук	150

Загальне споживання електроенергії всього обладнання з урахуванням втрат на ККД UPS становить 0,74 кВт/год за годину роботи, обчислено за формулою:

$$\text{Споживання\_факт} = (\text{Потужність} \times \text{Час роботи} \times \text{Кількість}) \div \text{ККД\_UPS}$$

Найбільше споживання припадає на персональні комп'ютери, що пов'язано з їх високою потужністю та тривалим часом використання протягом проєкту. Сумарні енерговитрати за весь період розробки становлять 1 250,06 гривень, обчислено за формулою:

$$\text{Вартість} = \text{Споживання\_факт} \times \text{Тариф}$$

Ця сума становить відносно невелику частку від загального бюджету проєкту, але має бути врахована при плануванні операційних витрат. Для зручності було створено таблицю розрахунку витрат на електроенергію (табл. 4.4).

Таблиця 4.4 – Розрахунок енерговитрат

Пристрій	Потужність, Вт	Споживання (1шт), кВт·год	Споживання сумарно, кВт·год	Час роботи, год	Кількість, шт	Загальна вартість для пристрою, грн
Комп'ютер	300	0,3	0,6	600	2	1057,54
Ноутбук	40	0,04	0,04	150	1	16,26

Пристрій	Потужність, Вт	Споживання (1шт), кВт·год	Споживання сумарно, кВт·год	Час роботи, год	Кількість, шт	Загальна вартість для пристрою, грн
Монітор	50	0,05	0,1	600	2	176,26
Всього			0,74			1250,06

Отже, загальні витрати на розробку програмного продукту включають витрати на оплату праці команди розробників з урахуванням обов'язкових податкових відрахувань (ПДФО 18%, військовий збір 5%, ЄСВ 22%), енерговитрати на забезпечення роботи технічного обладнання протягом п'ятимісячного періоду розробки, а також інші операційні витрати, пов'язані з реалізацією проєкту. Загальна сума витрат, розрахована за формулою: Витрати_загальні = Витрати_ЗП + Заг_ЄСВ + Енерговитрати, становить 468510 гривень, що відображає повну собівартість створення програмного продукту з урахуванням усіх необхідних ресурсів та обов'язкових відрахувань.

4.2.2 Розрахунок економічного ефекту від впровадження комп'ютерної системи чи пристрою

Створення застосунку для передачі аудіофайлів локальною мережею не передбачає отримання прямого економічного ефекту у вигляді грошових надходжень, враховуючи що вона буде мати відкритий код та розповсюджуватись за ліцензією GNU GENERAL PUBLIC LICENSE v3, що забезпечує безплатне використання для всіх категорій користувачів. Проте програмний продукт має потенціал для генерування значного непрямого економічного ефекту через оптимізацію робочих процесів та зменшення витрат на існуючі рішення.

Застосунок покликаний спростити прослуховування локальних аудіофайлів формату MP3, FLAC та інших через локальну мережу з

використанням P2P-підходу, що забезпечує високу якість звуку та швидкість передачі без необхідності централізованого сервера.

На відміну від традиційних медіаплеєрів, таких як Groove Music Player, Audacious або MusicBee, які обмежені роботою на одному пристрої, розроблений застосунок дозволяє організувати безперебійне прослуховування музики на всіх пристроях локальної мережі без необхідності дублювання файлів.

Економічний ефект від впровадження системи полягає у зменшенні потреби в додатковому дисковому просторі для зберігання копій музичних файлів на кожному пристрої, що може заощадити від 2000 до 8000 гривень на придбання додаткових накопичувачів. P2P-архітектура додатково усуває необхідність в придбанні та обслуговуванні централізованого сервера або NAS-пристрою для організації спільного доступу до музичної бібліотеки, вартість яких може сягати від 8000 до 25000 гривень, а також знижує витрати на електроенергію через відсутність сервера що постійно працює, це може заощадити до 1500 гривень на рік. Загальна економія від використання розробленого застосунку складатиме від 10000 до 34500 гривень одноразово та до 1500 гривень щорічно на електроенергію.

4.2.3 Окупність проєкту

Розрахунок періоду окупності дозволяє визначити тривалість часу, необхідного для повернення інвестованих коштів через отримуваний економічний ефект від використання розробленої системи. Цей показник характеризує фінансову ефективність проєкту та його здатність компенсувати початкові витрати. Для обчислення терміну окупності використовується співвідношення Термін_окупності = Срозр / Е, в якому Срозр відображає сукупні витрати на створення продукту, а Е представляє щорічний економічний ефект.

Базуючись на проведених обчисленнях, загальна вартість розробки становить 468510 гривень, тоді як економічна вигода варіюється від 11500 до

36000 гривень на рік (включаючи одноразову економію та щорічні заощадження). При песимістичному прогнозі: Ток = 468510 / 11500 = 40,7 років, при оптимістичному сценарії: Ток = 468510 / 36000 = 13,0 років.

4.3 Обґрунтування необхідності розробки

Отримані результати обрахунків вказують на значний період повернення вкладених ресурсів, що становить від 13 до 40,7 років залежно від інтенсивності використання системи. Такі показники відображають специфіку некомерційних проєктів з відкритим вихідним кодом, де головна мета полягає не в максимізації прибутку, а в створенні суспільно корисного продукту. Тривалий термін окупності компенсується стабільним довгостроковим ефектом та можливістю безкоштовного використання широким колом користувачів без обмежень ліцензійного характеру.

Проєкт з самого початку не задумувався як комерційне рішення, а спрямований на створення відкритого інструменту для спільноти користувачів, що підтверджується вибором ліцензії GNU GPL v3. Така концепція дозволяє максимізувати соціальну користь через безкоштовний доступ до функціоналу без обмежень використання.

Аналіз наявних на ринку рішень показав, що жоден з проаналізованих програвачів не задовольняє всі сучасні вимоги користувачів. Groove обмежений операційною системою Windows, Audacious не має сучасного інтерфейсу, а MusicBee, попри потужність, недоступний для інших ОС окрім Windows. Достатній рівень кросплатформності реалізовано лише в Audacious, але його застарілий дизайн і відсутність механізмів обміну даними роблять його менш привабливим для сучасних користувачів.

Усі три рішення не пропонують механізмів безпечного обміну файлами в локальній мережі, а закритість коду Groove та MusicBee обмежує можливості розвитку функціоналу спільнотою. Audacious хоч і має відкритий

код, але його спільнота розробників майже неактивна, що гальмує розвиток проєкту.

Пропонований застосунок покликаний задовольнити потреби користувачів у кросплатформному рішенні для прослуховування локальних аудіофайлів з можливістю безпечного обміну через локальну мережу. Економічний вплив проявляється через заощадження на придбанні комерційних аналогів, зменшення потреби в централізованих серверах та оптимізацію використання дискового простору. Основні напрямки отримання ефекту включають: зниження витрат на ліцензійне програмне забезпечення, економію на апаратному забезпеченні через P2P-архітектуру та зменшення енерговитрат через відсутність постійно працюючих серверів.

ВИСНОВКИ

В процесі виконання кваліфікаційної роботи створено програмний засіб для відтворення аудіоконтенту з децентралізованою архітектурою, який забезпечує обмін та програвання музичних файлів у локальних мережах без використання централізованих серверів. Всі поставлені завдання вирішено успішно: здійснено дослідження предметної області та аналіз існуючих рішень, розроблено детальну системну архітектуру з урахуванням специфіки P2P-технологій, впроваджено ключові функціональні компоненти включаючи механізми безпеки та аудіообробки, а також проведено комплексне функціональне тестування всіх підсистем.

Унікальність розробки полягає у створенні архітектурного рішення, яке об'єднує три критично важливі технології: розподілене зберігання інформації для забезпечення доступності даних без залежності від центральних вузлів, багаторівневий криптографічний захист на основі сучасних протоколів TLS/AES-256 для гарантування конфіденційності передачі, та гнучку систему динамічного розширення функціоналу через модульну архітектуру плагінів. Використання такого підходу повністю виключає проблеми єдиної точки відмови, характерні для традиційних клієнт-серверних архітектур, та гарантує повністю автономне функціонування кожного учасника мережі навіть при відключенні інших вузлів.

Практично реалізована система базується на потужному кросплатформному фреймворку Qt6 та інтегрує декілька ключових технологічних компонентів: P2P-мережу з mesh-топологією для забезпечення стійкості та масштабованості з'єднань, високопродуктивний модуль обробки аудіосигналу з підтримкою широкого спектра популярних та спеціалізованих форматів через бібліотеку FFmpeg, комплексний набір професійних звукових ефектів для покращення якості відтворення, та адаптивний кросплатформний інтерфейс користувача, оптимізований для різних типів пристроїв. Проведене

ретельне функціональне тестування в різних умовах експлуатації продемонструвало високу надійність та стабільність роботи всіх критично важливих підсистем: інтелектуального пошукового механізму з підтримкою складних запитів, системи реального часу аудіообробки з мінімальними затримками та механізмів P2P-взаємодії з автоматичним відновленням з'єднань.

Практичне значення результатів полягає у створенні інноваційного інструментарію для безпечного розповсюдження аудіоматеріалів у корпоративних та освітніх мережах зі збереженням якості звуку та метаданих. Відкрита ліцензія GPLv3 сприяє розвитку системи міжнародною спільнотою програмістів. Кросплатформна природа рішення забезпечує сумісність з Windows, Linux, macOS, Android та iOS.

Стратегічні напрямки майбутнього вдосконалення системи передбачають реалізацію передових технологій NAT-traversal для забезпечення безперешкодної роботи через глобальні мережі з подоланням обмежень мережевої інфраструктури, інтеграцію з популярними хмарними службами зберігання для створення гібридних архітектур, що поєднують переваги локального та віддаленого зберігання. Також заплановано створення SDK з детальною документацією та прикладами для спрощення розробки сторонніх розширень, формування централізованого каталогу додаткових модулів.

Створений програмний продукт являє собою технологічно передове та практично ефективне рішення для організації сучасного децентралізованого обміну високоякісним аудіоконтентом, яке демонструє значні перспективи широкомасштабного застосування у різноманітних бізнес-середовищах, навчальних закладах та творчих спільнотах, де критично важливими є питання безпеки, надійності та автономності роботи мультимедійних систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Qt | Tools for Each Stage of Software Development Lifecycle. URL: https://www.qt.io/ (дата звернення: 24.02.2025).
2. The GNU General Public License v3.0 - GNU Project - Free Software Foundation. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (дата звернення: 23.02.2025).
3. Groove Music - Wikipedia. URL: https://en.wikipedia.org/wiki/Groove_Music (дата звернення: 24.02.2025).
4. Audacious - An Advanced Audio Player. URL: https://www.audacious-media-player.org/ (дата звернення: 25.02.2025).
5. The 3-Clause BSD License – Open Source Initiative. URL: https://opensource.org/license/bsd-3-clause (дата звернення: 24.02.2025).
6. MusicBee - The Ultimate Music Manager and Player. URL: https://www.getmusicbee.com/ (дата звернення: 24.02.2025).
7. qt_add_qml_module | Qt Qml | Qt 6.9.0. URL: https://doc.qt.io/qt-6/qt-add-qml-module.html (дата звернення: 30.03.2025).
8. Cyber crime costs predicted to hit $10.5 trillion per year by 2025. URL: https://www.itpro.com/security/357769/cyber-crime-to-cost-businesses-105-trillion-per-year-by-2025 (дата звернення: 25.04.2025).
9. The European Public Sector Open Source Opportunity. URL: https://www.linuxfoundation.org/research/european-public-sector-opportunity (дата звернення: 25.04.2025).
10. Музика М. Розробка системи обробки аудіо у реальному часі з інтеграцією аудіоефектів. МАТЕРІАЛИ СТУДЕНТ. НАУКОВО-ПРАКТ. КОНФ., м. Тернопіль, 15 трав. 2025 р. Тернопіль, 2025.

ДОДАТКИ

Додаток А

Лістинг програмного коду утилітного скрипту для провайдера списку відтворення

```
macro(add_playlist_provider)
    set(_provider_name "")
    set(_provider_sources "")
    set(_install_dir "")

    set(_arg_mode "")
    foreach(arg ${ARGV})
        if("${arg}" STREQUAL "PROVIDER_NAME")
            set(_arg_mode "name")
        elseif("${arg}" STREQUAL "PROVIDER_SOURCES")
            set(_arg_mode "sources")
        elseif("${arg}" STREQUAL "INSTALL_DIR")
            set(_arg_mode "install")
        else()
            if("${_arg_mode}" STREQUAL "name")
                set(_provider_name "${arg}PlaylistProvider")
            elseif("${_arg_mode}" STREQUAL "sources")
                list(APPEND _provider_sources "${arg}")
            elseif("${_arg_mode}" STREQUAL "install")
                set(_install_dir "${arg}")
            else()
                message(FATAL_ERROR "Unknown argument or missing
keyword before '${arg}'")
            endif()
        endif()
    endforeach()

    if("${_provider_name}" STREQUAL "")
        message(FATAL_ERROR "PROVIDER_NAME is not set")
    endif()
    if("${_provider_sources}" STREQUAL "")
        message(FATAL_ERROR "PROVIDER_SOURCES is not set")
    endif()
    if("${_install_dir}" STREQUAL "")
        message(FATAL_ERROR "INSTALL_DIR is not set")
    endif()
```

```
    add_library(${_provider_name} SHARED ${_provider_sources})

    target_link_libraries(${_provider_name} PRIVATE
        Qt6::Core
        Qt6::Multimedia
        Qt6::Gui
        Qt6::QuickControls2
        playlist_provider_plugin_common
    )

    set_target_properties(${_provider_name} PROPERTIES
        LIBRARY_OUTPUT_DIRECTORY $
{CMAKE_RUNTIME_OUTPUT_DIRECTORY}/plugins/playlist_providers
    )

    install(TARGETS ${_provider_name}
        LIBRARY DESTINATION ${_install_dir}
        RUNTIME DESTINATION ${_install_dir}
    )
endmacro()
```

Додаток Б

Лістинг програмного коду аудіоефекту фільтра нижніх частот

```
#include <cmath>
#include <vector>

#include "EffectParameter.h"
#include "IAudioEffect.h"

class BassBoostEffect : public IAudioEffect
{
        Q_OBJECT
        Q_PLUGIN_METADATA(IID IAudioEffect_iid FILE
"BassBoostEffect.json")
        Q_INTERFACES(IAudioEffect)
    public:
        BassBoostEffect(QObject *parent = nullptr)
        {
            m_boost =
QSharedPointer<EffectParameter>::create("Boost", 1.0, 0.0, 5.0,
EffectParameter::Float, this);
            m_cutoff =
QSharedPointer<EffectParameter>::create("CutOffHz", 200.0, 20.0,
1000.0, EffectParameter::Float, this);

            emit parametersChanged();
        }

        QString effectName() const override { return "Bass
Boost"; }

        bool applyEffect(char *data, qint64 len, const
QAudioFormat &format) const override
        {
            const float boostVal = m_boost->value().toFloat();

            if (boostVal <= 0) {
                return true;
            }

            const float cutoffHz = m_cutoff->value().toFloat();
            const int channels = format.channelCount();
            const int sampleRate = format.sampleRate();
```

```
            // α = exp(-2π * fc / fs)
            const float alpha = std::exp(-2.0f * M_PI * cutoffHz
/ sampleRate);
            const float oneMinusAlpha = 1.0f - alpha;

            if (m_state.size() != static_cast<size_t>(channels))
                m_state.assign(channels, 0.0f);

            if (format.sampleFormat() == QAudioFormat::Float) {
                float *samples = reinterpret_cast<float
*>(data);
                const int sampleCount = len / sizeof(float);

                for (int i = 0; i < sampleCount; ++i) {
                    int ch = i % channels;
                    float x = samples[i];

                    float low = alpha * m_state[ch] +
oneMinusAlpha * x;
                    m_state[ch] = low;

                    samples[i] = x + boostVal * low;
                }
            } else if (format.sampleFormat() ==
QAudioFormat::Int16) {
                qint16 *samples = reinterpret_cast<qint16
*>(data);
                const int sampleCount = len / sizeof(qint16);

                for (int i = 0; i < sampleCount; ++i) {
                    int ch = i % channels;
                    float x = samples[i] / 32768.0f;

                    float low = alpha * m_state[ch] +
oneMinusAlpha * x;
                    m_state[ch] = low;

                    float y = x + boostVal * low;

                    int scaled = static_cast<int>(y * 32768);
                    samples[i] = qBound(-32768, scaled, 32767);
                }
            } else if (format.sampleFormat() ==
QAudioFormat::Int32) {
```

```
                qint32 *samples = reinterpret_cast<qint32
*>(data);
                const int sampleCount = len / sizeof(qint32);

                for (int i = 0; i < sampleCount; ++i) {
                    int ch = i % channels;
                    float x = samples[i] / 2147483648.0f;

                    float low = alpha * m_state[ch] +
oneMinusAlpha * x;
                    m_state[ch] = low;

                    float y = x + boostVal * low;
                    qint32 scaled = static_cast<qint32>(y *
2147483648.0f);
                    samples[i] = qBound<qint32>(INT32_MIN,
scaled, INT32_MAX);
                }
            } else {
                return false;
            }
            return true;
        }

        QList<QSharedPointer<EffectParameter>> parameters()
const override { return {m_boost, m_cutoff}; }

    private:
        QSharedPointer<EffectParameter> m_boost;
        QSharedPointer<EffectParameter> m_cutoff;
        mutable std::vector<float> m_state;
};

#include "BassBoostEffect.moc"
```

Додаток В

Лістинг програмного коду аудіоефекту шістнадцятисмугового еквалайзера

```
#include <vector>

#include "EffectParameter.h"
#include "IAudioEffect.h"

class EqualizerEffect : public IAudioEffect
{
        Q_OBJECT
        Q_PLUGIN_METADATA(IID IAudioEffect_iid FILE
"EqualizerEffect.json")
        Q_INTERFACES(IAudioEffect)
    public:
        EqualizerEffect(QObject *parent = nullptr)
        {
            for (size_t i = 0; i < m_centerFrequencies.size(); +
+i) {
                QString name = QString("%1
Hz").arg(m_centerFrequencies[i]);
                auto gain =
QSharedPointer<EffectParameter>::create(name, 0.0, -12.0, 12.0,
EffectParameter::Float, this);
                connect(gain.get(),
&EffectParameter::valueChanged, this,
&EqualizerEffect::onParameterChanged);
                m_bandGains.push_back(gain);
            }

            emit parametersChanged();
        }

        QString effectName() const override { return "16-Band
Equalizer"; }

        bool applyEffect(char *data, qint64 len, const
QAudioFormat &format) const override
        {
            const int channels = format.channelCount();
            const int sampleRate = format.sampleRate();
```

```
            if (m_lastSampleRate != sampleRate || m_coeffsDirty)
{
                precalculateCoefficients(sampleRate);
                m_lastSampleRate = sampleRate;
                m_coeffsDirty = false;
            }

            // init states
            if (m_states.empty() || m_states[0].size() !=
static_cast<size_t>(channels)) {
                m_states.resize(m_bandGains.size(),
std::vector<BiquadState>(channels));
            }

            if (format.sampleFormat() == QAudioFormat::Float) {
                float *samples = reinterpret_cast<float
*>(data);
                const int totalSamples = len / sizeof(float);

                for (int i = 0; i < totalSamples; ++i) {
                    const int channel = i % channels;
                    float sample = samples[i];

                    // Sequential processing across all bands
                    for (qsizetype band = 0; band <
m_bandGains.size(); ++band) {
                        const float gain = m_bandGains[band]-
>value().toFloat();
                        if (gain == 0.0)
                            continue;

                        // const float freq =
m_centerFrequencies[band];

                        sample = processBiquad(m_coeffs[band],
m_states[band][channel], sample);
                    }

                    samples[i] = sample;
                }
            } else {
                return false;
            }
            /**
```

```
             * @todo Realize process dor other formats
             */
            return true;
        }
        QList<QSharedPointer<EffectParameter>> parameters()
const override { return m_bandGains; }

    private slots:
        void onParameterChanged() { m_coeffsDirty = true; }

    private:
        struct BiquadCoefficients
        {
                float b0 = 0.0f, b1 = 0.0f, b2 = 0.0f;
                float a1 = 0.0f, a2 = 0.0f;
        };
        struct BiquadState
        {
                float x1 = 0.0f, x2 = 0.0f;
                float y1 = 0.0f, y2 = 0.0f;
        };
        const std::vector<float> m_centerFrequencies = {
            20.0f,    // 20 Hz
            40.0f,    // 40 Hz
            80.0f,    // 80 Hz
            160.0f,   // 160 Hz
            320.0f,   // 320 Hz
            640.0f,   // 640 Hz
            1300.0f,  // 1.3 kHz
            2500.0f,  // 2.5 kHz
            5000.0f,  // 5 kHz
            10000.0f, // 10 kHz
            12500.0f, // 12.5 kHz
            14000.0f, // 14 kHz
            16000.0f  // 16 kHz
        };

        QList<QSharedPointer<EffectParameter>> m_bandGains;
        const float Q = 1.0f;

        mutable bool m_coeffsDirty = true;
        mutable int m_lastSampleRate = -1;
        mutable std::vector<std::vector<BiquadState>> m_states;
        mutable std::vector<BiquadCoefficients> m_coeffs;
```

```
        void precalculateCoefficients(float sampleRate) const
        {
            m_coeffs.clear();
            for (qsizetype band = 0; band < m_bandGains.size();
++band) {
                const float freq = m_centerFrequencies[band];

m_coeffs.push_back(calculateCoefficients(m_bandGains[band]-
>value().toFloat(), freq, Q, sampleRate));
            }
        }
        BiquadCoefficients calculateCoefficients(float gainDB,
float freq, float Q, float sampleRate) const
        {
            BiquadCoefficients coeff;
            const float A = powf(10.0f, gainDB / 40.0f);
            const float omega = 2.0f * M_PI * freq / sampleRate;
            const float alpha = sinf(omega) / (2.0f * Q);
            const float a0 = 1.0f + alpha / A;
            // Calculating coeff for peaking EQ
            coeff.b0 = (1.0f + alpha * A) / a0;
            coeff.b1 = (-2.0f * cosf(omega)) / a0;
            coeff.b2 = (1.0f - alpha * A) / a0;
            coeff.a1 = (-2.0f * cosf(omega)) / a0;
            coeff.a2 = (1.0f - alpha / A) / a0;
            return coeff;
        }
        float processBiquad(const BiquadCoefficients &coeff,
BiquadState &state, float input) const
        {
            const float output =
                coeff.b0 * input + coeff.b1 * state.x1 +
coeff.b2 * state.x2 - coeff.a1 * state.y1 - coeff.a2 * state.y2;
            // Update state
            state.x2 = state.x1;
            state.x1 = input;
            state.y2 = state.y1;
            state.y1 = output;
            return output;
        }
};
#include "EqualizerEffect.moc"
```

Додаток Г

Лістинг програмного коду аудіоефекту підсилення

```
#include <QAudioFormat>

#include "EffectParameter.h"
#include "IAudioEffect.h"

class GainEffect : public IAudioEffect
{
        Q_OBJECT
        Q_PLUGIN_METADATA(IID IAudioEffect_iid FILE
"GainEffect.json")
        Q_INTERFACES(IAudioEffect)
    public:
        GainEffect(QObject *parent = nullptr)
        {
            m_gain =
QSharedPointer<EffectParameter>::create("Gain", 1.0, 0.0, 5.0,
EffectParameter::Float, this);
            emit parametersChanged();
        }

        QString effectName() const override { return "Gain"; }

        bool applyEffect(char *data, qint64 len, const
QAudioFormat &format) const override
        {
            auto gainVal = m_gain->value().toFloat();

            if (gainVal == 1.0) {
                return true;
            }

            switch (format.sampleFormat()) {
            case QAudioFormat::Float: {
                int sampleCount = len / sizeof(float);
                float *samples = reinterpret_cast<float
*>(data);
                for (int i = 0; i < sampleCount; ++i)
                    samples[i] *= gainVal;
                break;
            }
            case QAudioFormat::Int16: {
```

```
                int sampleCount = len / sizeof(qint16);
                qint16 *samples = reinterpret_cast<qint16
*>(data);
                for (int i = 0; i < sampleCount; ++i) {
                    int32_t scaled =
static_cast<int32_t>(samples[i] * gainVal);
                    samples[i] = std::clamp(scaled, -32768,
32767);
                }
                break;
            }
            case QAudioFormat::Int32: {
                int sampleCount = len / sizeof(qint32);
                qint32 *samples = reinterpret_cast<qint32
*>(data);
                for (int i = 0; i < sampleCount; ++i) {
                    int64_t scaled =
static_cast<int64_t>(samples[i] * gainVal);
                    samples[i] = std::clamp(scaled,
int64_t(INT32_MIN), int64_t(INT32_MAX));
                }
                break;
            }
            default:
                return false;
            }

            return true;
        }

        QList<QSharedPointer<EffectParameter>> parameters()
const override { return {m_gain}; }

    private:
        QSharedPointer<EffectParameter> m_gain;
};

#include "GainEffect.moc"
```

Додаток Ґ

Лістинг програмного коду делегату ListView відображення аудіофайлів

```
import QtQuick
import QtQuick.Layouts
import QtQuick.Controls
import CustomPlayer.Common
import CustomPlayer.Audio

import CustomPlayer

Item {
    id: item
    height: 60
    width: ListView.view.width - ListView.view.spacing - 5

    signal doubleClicked(int index)

    required property var model
    required property int index

    Rectangle {
        anchors.fill: parent
        radius: 2

        MouseArea {
            id: mouseArea
            anchors.fill: parent

            hoverEnabled: true

            onDoubleClicked: {
                item.doubleClicked(item.index)
                MusicPlayer.playFile(item.model.uri);
            }
        }

        color: mouseArea.containsMouse ||
item.ListView.isCurrentItem ? Colors.accent : Colors.secondary

        Behavior on color {
            ColorAnimation { duration: 100 }
        }
```

```
        RowLayout {
            anchors.fill: parent
            RoundedImage {
                id: img
                radius: 4

                Layout.margins: 6
                Layout.preferredHeight: item.height -
Layout.margins*2
                Layout.preferredWidth: Layout.preferredHeight

                source: "file://" + item.model.albumArt
                fillMode: Image.PreserveAspectCrop
                asynchronous: true
                cache: true
                smooth: true
            }
            ColumnLayout {
                Text {
                    text: item.model.album ? item.model.album +
" - " + item.model.title : item.model.title
                    color: Colors.text_p
                }
                Text {
                    text: item.model.artist
                    color: Colors.text_s
                }
            }
            Item { Layout.fillWidth: true }
            RowLayout {
                Text {
                    text:
Utils.msecToDuration(item.model.length)
                    color: Colors.text_s
                }
                Button {
                    id: favorite
                    icon.name: item.model.favorite ? "favorite-
favorited" : "favorite"
                    icon.color: Colors.text_p
                    onClicked: item.model.favorite = !
item.model.favorite

                    ToolTip.visible: hovered
```

```
                    ToolTip.delay: 1000
                    ToolTip.text: item.model.favorite ?
qsTr("Remove song from favorites") : qsTr("Add song to
favorites")
                }
            }

        }
    }
}
```

Додаток Д

Лістинг програмного коду діалогу додавання плейлиста

```
import QtQuick
import QtQuick.Layouts
import QtQuick.Controls
import QtQuick.Effects

import CustomPlayer

import "../components"

Dialog {
    id: dialog

    property bool isRequirementsProvided: false

    background: Rectangle {
        color: Colors.background
        radius: 4
        border.color: Colors.shadow
        border.width: 1

        layer.enabled: true
        layer.effect: MultiEffect {
            shadowEnabled: true
            shadowColor: Colors.shadow
            shadowBlur: 0.5
        }
    }


    header: Label {
        text: "Create New Playlist"
        font.pixelSize: 20
        font.bold: true
        leftPadding: 24
        topPadding: 24
        bottomPadding: 12
        color: Colors.text_p
    }

    property var providerUiObj: null
    property var providerFactory: null
```

```
    onAccepted: {
        playlistsModel.update();
    }

    contentItem: ColumnLayout {
        spacing: 16
        // padding: 24

        TextField {
            id: nameField
            Layout.fillWidth: true
            placeholderText: "Playlist name"
            font.pixelSize: 14
            color: Colors.text_p
            selectByMouse: true

            background: Rectangle {
                color: "transparent"
                border.color: Colors.accent
                border.width: 1
                radius: 4
            }
        }

        ComboBox {
            id: providerSelector
            Layout.fillWidth: true
            model: playlistsModel.providersRegistry
            textRole: "factoryName"
            visible: playlistsModel.providersRegistry.rowCount()
> 1
            onCurrentIndexChanged: {
                dialog.providerFactory =
playlistsModel.providersRegistry.getFactory(currentIndex)
                providerUiObj =
dialog.providerFactory.providerUi(dialog)
                if(dialog.providerUiObj) {
                    // obj.parent = uiLoader

uiLoader.setSourceObject(dialog.providerUiObj);
                    loaderScrollView.visible = true
                } else {
                    uiLoader.setSourceObject(null);
```

```
                // uiLoader.sourceComponent = null
                loaderScrollView.visible = false
            }
        }

        background: Rectangle {
            color: "transparent"
            border.color: Colors.accent
            border.width: 1
            radius: 4
        }
    }

    ScrollView {
        id: loaderScrollView
        Layout.fillWidth: true
        Layout.preferredHeight: 200
        visible: false
        ScrollBar.vertical.policy: ScrollBar.AlwaysOn

        ObjectLoader {
            id: uiLoader

            anchors.fill: parent
        }

        // Loader {
        //     clip: true
        //
        //     id: uiLoader
        //     anchors.fill: parent
        // }
    }
}


footer: DialogButtonBox {
    padding: 12
    background: Rectangle {
        color: "transparent"
    }

    Button {
        id: cancelButton
```

```
            text: "Cancel"
            DialogButtonBox.buttonRole:
DialogButtonBox.RejectRole
            onClicked: dialog.reject()
            flat: true
            background: Rectangle {
                radius: 6
                color: Colors.accent
                opacity: cancelButton.hovered ? 1.0 : 0.0

                Behavior on opacity {
                    NumberAnimation { duration: 75 }
                }
            }
        }

        Button {
            id: createButton
            text: "Create"
            enabled: nameField.text.length > 0 &&
(dialog.providerUiObj ?
dialog.providerFactory.isRequirementsProvided : true)
            DialogButtonBox.buttonRole:
DialogButtonBox.AcceptRole
            flat: true
            onClicked: dialog.accept()

            background: Rectangle {
                radius: 6
                color: Colors.accent
                opacity: createButton.hovered ? 1.0 : 0.0

                Behavior on opacity {
                    NumberAnimation { duration: 75 }
                }
            }
        }
    }
}
```