ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Цифрова платформа музичного ком'юніті з функціоналом пошуку композицій, навчання та захищеної взаємодії користувачів із використанням IoT-інфраструктури»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Богдан ФЕЩЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-41
Богдан ФЕЩЕНКО

Керівник: Ігор СІНІЦИН
д.т.н., професор

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____Вікторія ЖЕБКА
« ____» _____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Фещенку Богдану Вікторовичу

1. Тема кваліфікаційної роботи: «Цифрова платформа музичного ком'юніті з функціоналом пошуку композицій, навчання та захищеної взаємодії користувачів із використанням IoT-інфраструктури»
керівник кваліфікаційної роботи д.т.н., професор, професор кафедри ТЦР Ігор Сініцин, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51.
2. Строк подання кваліфікаційної роботи «18» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: Офіційна документація середовища Node.js, бібліотеки React та фреймворку Tailwind CSS; специфікації протоколу WebRTC та стандартів наскрізного шифрування (E2EE) за допомогою Web Crypto API; технічна документація СКБД PostgreSQL та ORM Prisma; посібники з налаштування IoT-інфраструктури на базі Raspberry Pi, систем контейнеризації Docker та безпечного тунелювання трафіку (Cloudflare Tunnels); наукові джерела та дослідження у сфері проєктування децентралізованих систем зв'язку та музичних платформ.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз предметної галузі музичних платформ, засобів захищеної
2. технологічного стеку, моделювання БД, UML-діаграми прецедентів та розгортання).
3. Програмна реалізація платформи (реалізація клієнтської частини, глобального аудіоплеєра, WebRTC-з'єднань та E2EE-шифрування).
4. Тестування та оцінка ефективності системи (навантажувальне тестування

IoT-вузла, тестування безпеки та кросбраузерна сумісність).

5. Перелік графічного матеріалу: *презентація*
    1. Аналіз аналогів.
    2. Вимоги до програмного забезпечення.
    3. Програмні засоби реалізації.
    4. Діаграма варіантів використання (адмін, музикант та гість).
    5. Діаграма структури бази даних.
    6. Діаграма розгортання інфраструктури на IoT-вузлі
    7. Діаграма навігації головної сторінки
    8. Діаграма послідовності WebRTC та E2EE-сигналізації
    9. Екранні форми.
6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	20.02.-27.02.2026	
2	Аналіз та дослідження існуючих аналогів	27.03-06.03.2026	
3	Огляд алгоритмів визначення типів фігури та кольоротипу, технологій побудови систем персоналізованих рекомендацій	06.03-13.03.2026	
4	Проєктування web-застосунку для онлайн-магазину вечірніх суконь	13.03-27.03.2026	
5	Програмна реалізація web-застосунку для магазину вечірніх суконь	27.03-17.04.2026	
6	Тестування web-застосунку	17.04-01.05.2026	
7	Оформлення роботи: вступ, висновки, реферат	01.05-06.05.2026	
8	Розробка демонстраційних матеріалів	06.05-11.05.2026	
9	Попередній захист роботи	11.05.2026	

Здобувачка вищої освіти _____ Богдан ФЕЩЕНКО
(підпис)

Керівник кваліфікаційної роботи _____ Ігор СІНІЦИН
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: _69_ стор., ___1___ табл., __25__ рис., __32__ джерел.

Мета роботи – підвищення ефективності взаємодії користувачів та рівня захищеності обміну даними шляхом розробки цифрової платформи музичного ком'юніті з функціоналом пошуку композицій, навчання та інтеграцією IoT-інфраструктури.

Об'єкт дослідження – процес організації безпечної соціальної взаємодії та обміну мультимедійним контентом у децентралізованих музичних спільнотах.

Предмет дослідження – програмне забезпечення, інфраструктурні та криптографічні рішення для функціонування мультимедійної платформи на базі IoT-пристрою.

Короткий зміст роботи: Проаналізовано існуючі комунікаційні сервіси та обґрунтовано застосування мікрокомп'ютерів (Raspberry Pi) як інфраструктурної альтернативи хмарному хостингу. Розроблено архітектуру системи та програмно реалізовані ключові функції: підсистема E2EE-чатів, групові відеоконференції із записом екрана, фоновий аудіоплеєр та бібліотека нотних партитур. Проведено інтеграційне та навантажувальне тестування механізмів лімітування запитів. В роботі використано середовище Node.js, Socket.io, PostgreSQL, React, Tailwind CSS, Web Crypto API та WebRTC, а також технології Docker і Cloudflare Tunnels для безпечного мережевого розгортання.

Сферою використання застосунку є організація комунікації незалежних музичних колективів, забезпечення дистанційного освітнього процесу в музичних закладах та захищений обмін інтелектуальною власністю.

КЛЮЧОВІ СЛОВА: ЦИФРОВА ПЛАТФОРМА, МУЗИЧНЕ КОМ'ЮНІТІ, ПЕРИФЕРІЙНІ ОБЧИСЛЕННЯ, IOT, RASPBERRY PI, WEBRTC, НАСКРІЗНЕ ШИФРУВАННЯ, КОНТЕЙНЕРИЗАЦІЯ.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасному цифровому світі індустрія музики та онлайн-навчання зазнає стрімкої трансформації. Із розвитком інтернет-технологій музиканти, композитори та студенти все частіше віддають перевагу онлайн-платформам для спільної творчості, обміну партитурами та дистанційного навчання. Незважаючи на наявність таких гігантів, як Spotify, SoundCloud чи MuseScore, більшість із них є комерційно орієнтованими та не пропонують комплексного інструментарію для захищеної, приватної взаємодії користувачів у реальному часі (наприклад, проведення зашифрованих відеозустрічей чи спільне редагування композицій) [1, 2, 3, 4].

Крім того, у сфері розробки програмного забезпечення зростає тенденція до децентралізації та використання периферійних обчислень (Edge Computing), де замість дорогих хмарних серверів застосовуються локальні IoT-рішення, зокрема мікрокомп'ютери на кшталт Raspberry Pi [5, 6]. Розгортання повноцінної соціальної платформи на базі IoT-інфраструктури з використанням сучасних методів тунелювання (наприклад, Cloudflare Tunnels) та контейнеризації є актуальним викликом, що дозволяє забезпечити автономність системи, знизити витрати на інфраструктуру та підвищити рівень конфіденційності даних. Враховуючи необхідність захисту авторського права та приватних комунікацій незалежних творців, впровадження наскрізного шифрування (E2EE) та технології WebRTC робить розробку такої платформи вкрай затребуваною та науково обґрунтованою [7, 8, 9].

Мета роботи – підвищення ефективності взаємодії користувачів та рівня захищеності обміну даними шляхом розробки цифрової платформи музичного ком’юніті з функціоналом пошуку композицій, навчання та інтеграцією IoT-інфраструктури.

Для досягнення поставленої мети у кваліфікаційній роботі вирішено такі **завдання**:

1. Провести аналіз предметної галузі, сучасних музичних платформ та методів забезпечення захищеного зв'язку у web-додатках.

2. Дослідити можливості та обмеження використання IoT-пристроїв як автономної серверної інфраструктури для хостингу web-застосунків.

3. Спроєктувати архітектуру програмної системи, реляційну базу даних та логіку взаємодії клієнтської і серверної частин.

4. Розробити алгоритми наскрізного шифрування повідомлень та налаштувати WebRTC-з'єднання для проведення безпечних відеоконференцій.

5. Реалізувати функціонал глобального аудіоплеєра, бібліотеки композицій та засобів інтерактивного навчання музиці.

6. Розгорнути розроблену систему на IoT-вузлі з використанням технологій контейнеризації та налаштувати безпечний віддалений доступ за допомогою тунелювання трафіку.

7. Провести комплексне тестування продуктивності, безпеки та функціональності створеної платформи.

Об’єкт дослідження – процес організації захищеної взаємодії, зберігання та обміну цифровим контентом у спеціалізованих онлайн-спільнотах.

Предмет дослідження – методи, технології та програмно-апаратні засоби розробки цифрової музичної платформи з використанням IoT-інфраструктури та алгоритмів наскрізного шифрування.

Методи дослідження. Під час виконання кваліфікаційної роботи використано: методи системного аналізу (для формування функціональних вимог до платформи); методи об’єктно-орієнтованого проєктування (для створення архітектури системи); криптографічні методи (для забезпечення наскрізного шифрування даних); методи імітаційного та навантажувального тестування (для перевірки стабільності роботи системи на обмежених апаратних ресурсах IoT-вузла).

Наукова новизна одержаних результатів полягає у комплексному поєднанні методів захищеної комунікації в реальному часі із технологіями периферійних обчислень на базі мікрокомп'ютерів. Вперше запропоновано та

програмно реалізовано архітектуру спеціалізованої музичної платформи, яка функціонує в ізольованій IoT-інфраструктурі з безпечним виведенням у глобальну мережу через тунелювання трафіку, що мінімізує залежність від комерційних хмарних провайдерів.

Практична значущість одержаних результатів. Розроблена цифрова платформа є готовим до впровадження програмним продуктом, який може використовуватися незалежними музичними колективами або локальними ком'юніті. Також результати розробки можуть бути використані у навчальному процесі на кафедрі інженерії програмного забезпечення Державного університету інформаційно-комунікаційних технологій для демонстрації сучасних підходів інтеграції web-додатків та IoT-інфраструктури. Впровадження системи дозволяє істотно знизити фінансові витрати на хостинг, забезпечити повний контроль над даними та надати користувачам безпечний інструментарій для віддаленої роботи.

Апробація результатів роботи та публікації. Основні теоретичні положення, технічні та інфраструктурні рішення кваліфікаційної роботи доповідали перед науковою спільнотою, обговорювалися та отримали схвальну оцінку на наукових заходах різного рівня.

За матеріалами проведених у рамках дослідження випробувань та розробок опубліковано 2 наукові праці у збірниках тез доповідей [10, 11]:

1. Фещенко Б.В. Гібридна архітектура музичної платформи на базі периферійних обчислень та серверлес-технологій // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.: ДУІКТ, 2026.
2. Сініцин І., Фещенко Б. Інтеграція IoT-інфраструктури та хмарних сервісів для побудови захищеної цифрової музичної платформи // Collection of Scientific Papers with the Proceedings of the 7th International Scientific and Practical Conference «Scientific Exploration: Bridging Theory and Practice» (May 25-27, 2026, Berlin, Germany). Berlin: European Open Science Space, 2026. P. 288–290.

1 ОГЛЯД ТА АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ МУЗИЧНИХ ПЛАТФОРМ ТА ІОТ-СИСТЕМ

1.1 Характеристика предметної галузі та сучасних музичних онлайн-ком'юніті

Сучасний етап розвитку цифрової економіки характеризується глибокою трансформацією музичної індустрії. Параметри взаємодії між авторами контенту та споживачами змістилися з моделі володіння фізичними носіями до парадигми «музика як послуга» (Music-as-a-Service, MaaS). Проте, поряд із розвитком стримінгових гігантів, виникла гостра потреба у формуванні вузькоспеціалізованих цифрових екосистем – музичних ком’юніті, які фокусуються не лише на пасивному прослуховуванні, а й на активному навчанні, спільному створенні (co-creation) та безпечному обміні професійними активами (партитурами, мультитреками).

Предметна галузь музичних онлайн-платформ сьогодні представлена складними розподіленими системами високої навантаженості. Проте аналіз існуючих рішень демонструє низку фундаментальних проблем. Централізовані сервіси, такі як Spotify, попри високу якість рекомендаційних алгоритмів, будують свою перевагу на тотальному зборі персональних даних користувачів. Як зазначають дослідники, такий підхід часто входить у конфлікт із вимогами GDPR, оскільки персоналізація контенту купується ціною конфіденційності [12,1]. У контексті професійного музичного ком’юніті це створює ризики для авторів, які прагнуть захистити свої інтелектуальні напрацювання на ранніх етапах творчості.

Іншим важливим аспектом є еволюція платформ для обміну нотним контентом. MuseScore, будучи одним із найпопулярніших рішень, продемонстрував складність монетизації продуктів із відкритим кодом (FOSS). Перехід до агресивної підписної моделі для завантаження контенту викликав критику в середовищі розробників та музикантів, актуалізувавши питання створення альтернативних майданчиків із прозорою логікою доступу [12]. Це

підтверджує необхідність розробки систем, де функціонал завантаження та перегляду композицій інтегрований у соціальну складову без надмірних комерційних бар'єрів.

Особливе місце в екосистемі займають комунікаційні інструменти. Платформа Discord стала зразком «третього місця» (Third Place) – віртуального простору, що поєднує нейтральність, доступність та акцент на постійному спілкуванні [13, 14]. Однак для музикантів загальні месенджери часто виявляються недостатньо функціональними, оскільки вони не забезпечують нативної підтримки музичних прев'ю, спеціалізованої фільтрації за інструментами чи наскрізного захисту сесій відеозв'язку, необхідних для дистанційних репетицій чи уроків.

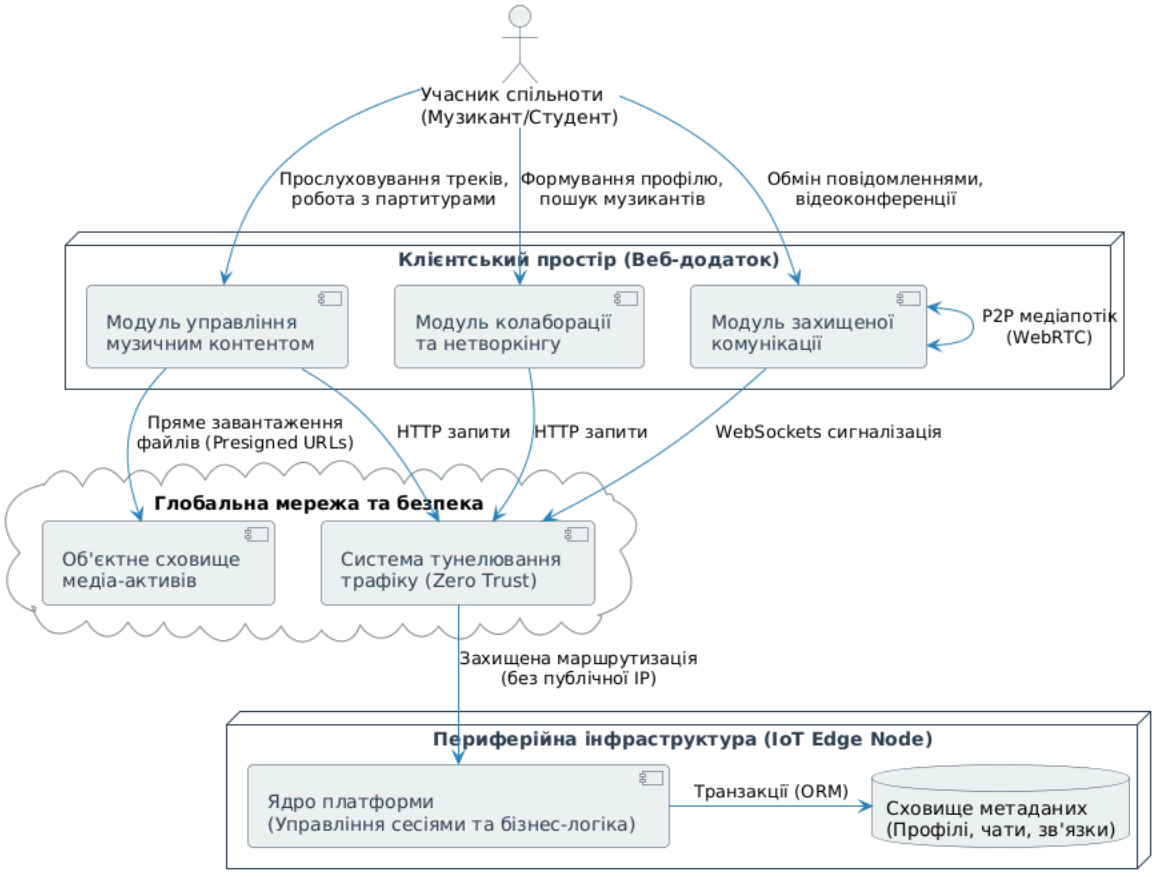


Рис. 1.1 Концептуальна модель взаємодії в сучасному музичному ком'юніті

На Рисунку 1.1 зображено концептуальну модель взаємодії у розробленій

системі. Архітектура розподілена на три логічні рівні: клієнтський простір, підсистему глобальної мережевої безпеки та локальну периферійну інфраструктуру (IoT Edge Node).

Центральним актором є користувач, який взаємодіє з трьома основними клієнтськими модулями. Модуль колаборації відповідає за етап онбордингу, налаштування профілю (локація, навички, музичні інструменти) та пошук інших учасників. Модуль управління контентом забезпечує доступ до глобального аудіоплеєра та бібліотеки партитур. Модуль комунікації відповідає за створення приватних або групових чатів та організацію захищених відеоконференцій (через P2P-з'єднання WebRTC) [7, 8, 9].

Ключовою особливістю моделі є механізм маршрутизації: весь трафік від клієнта проходить через систему тунелювання за принципом Zero Trust, що дозволяє безпечно направляти запити до ядра платформи, яке фізично розгорнуте на ізольованому мікрокомп'ютері (Raspberry Pi), захищаючи його від прямих DDoS-атак та несанкціонованого доступу. Великі медіа-файли (аватарки, аудіотреки, PDF-партитури) маршрутизуються напряму у зовнішнє хмарне об'єктне сховище для розвантаження IoT-вузла.

Технологічний стек сучасних платформ також зазнає змін. Перехід від монолітних архітектур до мікросервісних, як це було у випадку з багаторічною трансформацією SoundCloud, дозволяє підвищити масштабованість, проте суттєво ускладнює інфраструктурні витрати [3]. Для незалежних спільнот виникає альтернативний шлях – використання IoT-інфраструктури та периферійних обчислень (Edge Computing). Розгортання серверної частини на базі мікрокомп’ютерів типу Raspberry Pi дозволяє забезпечити цифрову суверенність даних (Data Sovereignty) [4]. Такий підхід дає змогу власнику спільноти повністю контролювати потік інформації, уникаючи залежності від великих хмарних провайдерів (Cloud Lock-in).

Таким чином, сучасна характеристика предметної галузі вказує на зміщення вектору розробки від глобальних масових сервісів до локальних, захищених та функціонально насичених платформ, що поєднують у собі можливості соціальної

мережі, освітнього простору та високопродуктивної IoT-серверної бази.

1.2 Аналіз існуючих аналогів та платформ

Щоб обґрунтувати доцільність створення власної цифрової платформи, а також виявити архітектурні та функціональні прогалини, що існують на сучасному ринку, варто ретельно вивчити наявні рішення. Сфера музичних сервісів і онлайн-інструментів для спілкування є висококонкурентною, однак більшість систем спрямовані на масового споживача (B2C) і не відповідають специфічним потребам вузьких професійних музичних спільнот, де в пріоритеті стоять приватність, захист авторських прав та інструменти для спільного навчання.

Для поглибленого аналізу було відібрано чотири платформи, що відображають різні аспекти предметної галузі: Spotify (як зразок потокового аудіо), SoundCloud (як соціальна мережа для незалежних авторів), MuseScore (як спеціалізована бібліотека нотних партитур) та Discord (як інструмент для управління ком'юніті та спілкування).

1.2.1 Аналіз платформи Spotify

Сервіс Spotify є світовим лідером у сфері музичного стримінгу, реалізуючи концепцію музики як послуги (Music-as-a-Service). Основою платформи є застосування розподілених мікросервісів та технології машинного навчання (Machine Learning) для роботи з великими об’ємами даних (Big Data). Ключовою перевагою системи є потужний механізм рекомендацій, який враховує патерни прослуховувань користувачів і дозволяє персоналізувати вибір музики в реальному часі.

Однак, незважаючи на технологічну досконаліть у галузі дистрибуції аудіоконтенту, Spotify має певні архітектурні обмеження з позиції потреб музикантів та місцевих музичних спільнот. По-перше, платформа повністю орієнтована на споживача (consumption-oriented). В ній відсутні вбудовані

інструменти для підтримки процесу створення музики, обміну навчальними матеріалами (наприклад, нотовими записами) або для колаборації над незавершеними композиціями (Drafts).

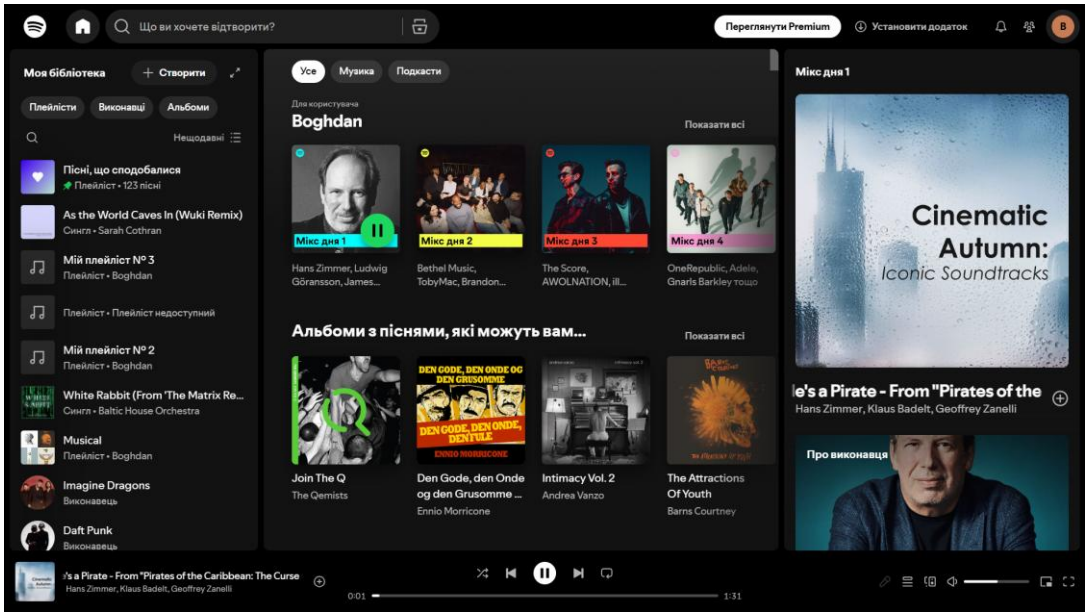


Рис. 1.2 Інтерфейс користувача платформи Spotify, орієнтований на споживання готового контенту

По-друге, модель заробітку платформи ставить під сумнів право на конфіденційність. Так, в дослідженні Упсальського університету зазначається, що вся перевага конкурента Spotify полягає в тому, що він жорстко збирає персональні дані, що створює постійні загрози його відповідності європейському регламенту про захист даних (GDPR) [1]. Користувачі фактично платять своїм правом на конфіденційність за персоналізацію контенту. Для незалежних музикантів, яким потрібно зберегти конфіденційність обговорень і захистити ранні демо-версії треків від витоку, використання публічної стримінгової платформи як інструменту для роботи неможливо.

По-третє, у Spotify немає функції прямого та захищеного спілкування між користувачами (Direct Messaging) або інструментів для організації відеоконференцій (репетицій), тому творці змушені використовувати сторонні месенджери, що порушує цілісність процесу роботи.

Отже, для розробки глобального аудіоплеєра у допоміжному застосунку Spotify можна розглядати як приклад архітектури, але закритість, відсутність інструментів E2EE-комунікації та акцент на монетизації метаданих унеможливлюють його використання як платформи для захищеної взаємодії та навчання.

1.2.2 Аналіз платформи SoundCloud

SoundCloud традиційно розглядається як найсоціальніша платформа для незалежних музикантів, продюсерів і подкастерів. На відміну від Spotify, який є сервісом для стрімінгу музики, і фокусується на релізах лейблів, SoundCloud із самого початку був будувався як платформа, де автори контенту (creators) і слухачі (listeners) можуть взаємодіятибезпосередньо. Однією з найкращих інновацій платформи стала можливість коментувати аудіо файли, які йдуть у стрічці у вигляді осцилограм (waveform). Саме так звучав один із найпростіших форматів асинхронного зворотного зв’язку, який побачив світ

З інженерної точки зору еволюція архітектури SoundCloud, цікавий досвід для розробників розподілених систем. Як описують у технічних статтях інженери компанії, платформу було створено за класичним монолітним підходом (Monolithic Architecture) у фреймворку Ruby on Rails [3]. Коли аудиторія платформи почала стрімко зростати, моноліт став «пляшковим горлечком» (bottleneck) у розгортанні оновлень (deployments) і масштабуванні. Це призвело до того, що компанія розпочала багаторічний процес перенесення в мікросервісну архітектуру (Microservices Architecture) та впровадження Backend-for-Frontend (BFF) [3, 15].

Цей досвід підтверджує, що для сучасних музичних платформ питання масштабованості та ізоляції сервісів потрібно закладати ще на етапі початкового проєктування.

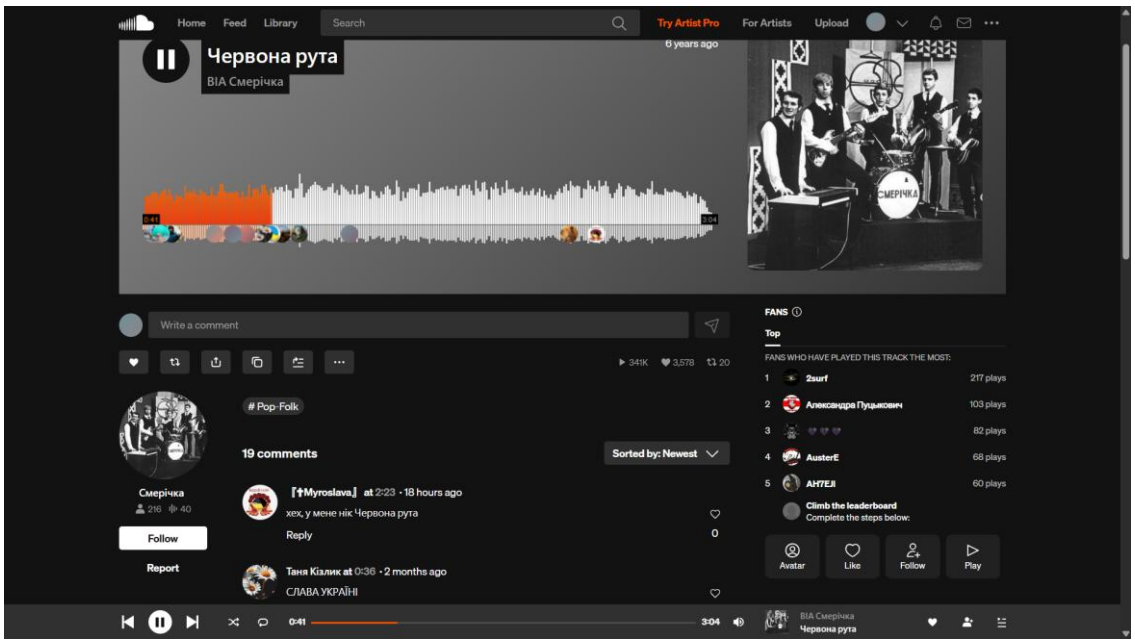


Рис. 1.3 Інтерфейс SoundCloud із візуалізацією аудіодоріжки та асинхронними коментарями

Незважаючи на значні переваги у сфері публікації аудіоматеріалів, детальний аналіз функціоналу SoundCloud виявляє низку суттєвих прогалин у контексті закритої, професійної та навчальної взаємодії:

1. Відсутність спеціалізованого навчального контенту. SoundCloud оперує виключно аудіоформатами. Платформа не підтримує завантаження, відображення чи спільне редагування нотних партитур (PDF, MusicXML тощо), що робить її непридатною для освітніх цілей та академічного музичного середовища.

2. Обмеження синхронної комунікації. Взаємодія між користувачами зводиться до асинхронних коментарів та базової системи особистих повідомлень. Платформа не підтримує технології WebRTC для створення групових відеозустрічей чи голосових чатів у реальному часі, що є критично важливим для дистанційних репетицій.

3. Проблеми приватності та суверенітету даних. Хоча SoundCloud дозволяє публікувати приватні треки, вся інформація зберігається на централізованих серверах компанії (Cloud-based storage). Відсутність наскрізного

шифрування (E2EE) в особистих повідомленнях не гарантує захисту від перехоплення даних або несанкціонованого доступу з боку третіх осіб.

Підсумовуючи, можна стверджувати, що SoundCloud є потужним інструментом для маркетингу та промоції готового музичного продукту, проте його архітектура не розрахована на організацію захищеного, приватного робочого простору (collaboration workspace). Для вирішення завдань створення контенту, навчання та безпечної комунікації виникає потреба в розробці більш децентралізованого рішення, здатного працювати на периферійних вузлах (IoT).

1.2.3 Аналіз платформи MuseScore

Платформа MuseScore посідає унікальне місце в екосистемі музичних сервісів, будучи де-факто галузевим стандартом для створення, публікації та обміну цифровими нотними партитурами (Sheet Music). З архітектурної точки зору платформа являє собою гібридну систему: вона поєднує десктопний редактор із відкритим вихідним кодом (Free and Open-Source Software, FOSS) та потужний хмарний репозиторій (musescore.com), що забезпечує рендеринг складних музичних форматів (наприклад, MusicXML та MIDI) безпосередньо у вікні веб-браузера.

Така спеціалізація робить MuseScore незамінним інструментом для навчального процесу та академічного музичного середовища. Однак детальний аналіз еволюції платформи виявляє критичні недоліки в її бізнес-логіці та підходах до управління доступом користувачів, що суттєво обмежує її придатність для локальних навчальних спільнот.

Основною проблемою є агресивна комерціалізація контенту, який генерується самою спільнотою (User-Generated Content). Незважаючи на відкритий статус базового програмного забезпечення, веб-інфраструктура платформи використовує жорстку підписну модель (Paywall). Згідно з дослідженнями зворотного зв'язку користувачів та аналізом політик платформи, базові функції, такі як завантаження партитур у форматах PDF або MIDI (навіть тих творів, що належать до суспільного надбання – Public Domain), були

переведені до категорії преміум-послуг (Pro Features) [12, 6].

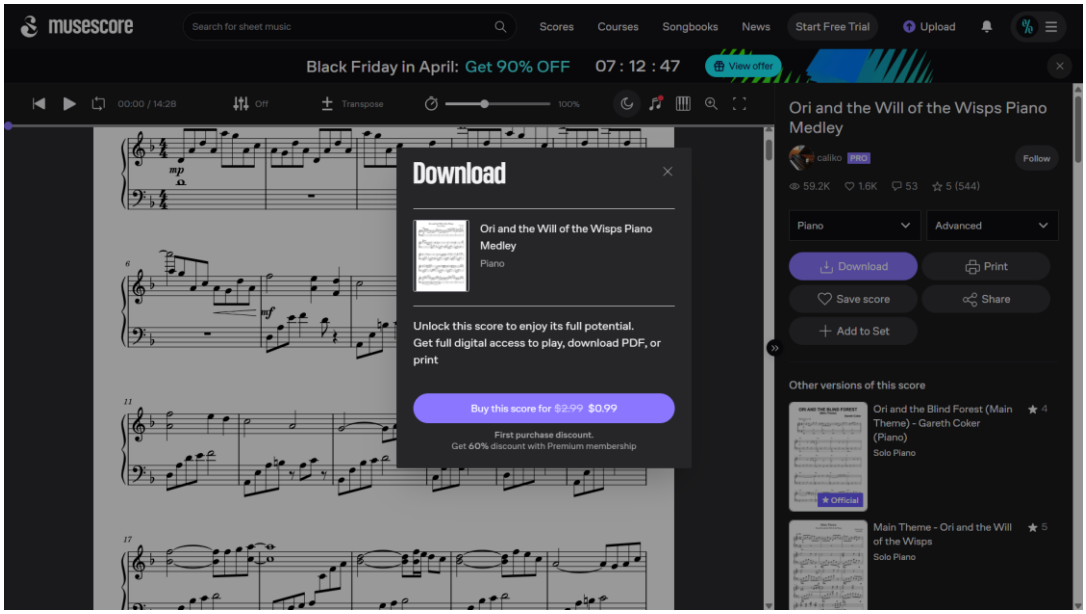


Рис. 1.4 Інтерфейс бібліотеки партитур MuseScore з обмеженням доступу до завантаження файлів

Така архітектура доступу створює штучні бар'єри для студентів, викладачів та незалежних композиторів, суперечачи початковій філософії вільного обміну знаннями. Крім того, платформа орієнтована переважно на глобальну публікацію. Вона не надає зручних інструментів для створення закритих робочих просторів (Private Workspaces), де музичний гурт або локальне ком'юніті могли б приватно обмінюватися чернетками (Drafts), сортувати їх за внутрішніми тегами та спільно працювати над ними без ризику несанкціонованого доступу третіх осіб або необхідності сплачувати високі тарифи.

У контексті розроблюваної програмної системи досвід MuseScore демонструє необхідність імплементації власної підсистеми роботи з партитурами (Score Library). Така підсистема має поєднувати функціонал перегляду та прослуховування музичних файлів з абсолютною прозорістю доступу для авторизованих членів спільноти, повністю виключаючи комерційні бар'єри для внутрішньогрупового обміну даними.

1.2.4 Аналіз платформи Discord

Discord на сьогодні є однією з найбільш динамічних комунікаційних платформ, яка успішно трансформувалася з вузькоспеціалізованого інструменту для геймерів у глобальну мережу для тематичних спільнот. В основі успіху Discord лежить концепція «третього місця» (Third Place) – віртуального простору, який не є ні домом, ні роботою, але забезпечує нейтральну територію для вільного соціального обміну [13, 14]. Для музикантів Discord став зручним майданчиком для створення серверів, де можна проводити текстові обговорення, ділитися посиланнями та влаштовувати стрімінгові сесії.

З технічної точки зору Discord використовує сучасні протоколи для передачі голосу та відео в реальному часі (WebRTC), що забезпечує низьку затримку (latency), критично важливу для голосового спілкування. Проте, детальний аналіз архітектури безпеки та функціонального наповнення платформи виявляє низку обмежень, що робить її недостатньою для професійної музичної взаємодії:

1. Відсутність наскрізного шифрування (E2EE) за замовчуванням. На відміну від месенджерів типу Signal або WhatsApp, Discord використовує шифрування при транспортуванні (encryption in transit), але не наскрізне шифрування для повідомлень та дзвінків. Це означає, що дані теоретично доступні на серверах компанії, що створює ризики для конфіденційності під час обговорення комерційних аспектів творчості або передачі авторських матеріалів [15, 16].

2. Неструктурованість контенту. Discord – це потік повідомлень. У ньому відсутня вбудована логіка управління бібліотеками композицій (Scores) або треків. Пошук конкретного музичного файлу в історії чату стає складним завданням, оскільки платформа не має системи метаданих, спеціалізованих тегів за інструментами чи музичними жанрами.

3. Відсутність професійного інструментарію. Платформа не підтримує нативний перегляд партитур, не має інтегрованих аудіоплеєрів із підтримкою професійних форматів без втрат (lossless) та не дозволяє здійснювати багатоканальний запис зустрічей безпосередньо в інтерфейсі без використання

сторонніх ботів, які часто знижують якість звуку.

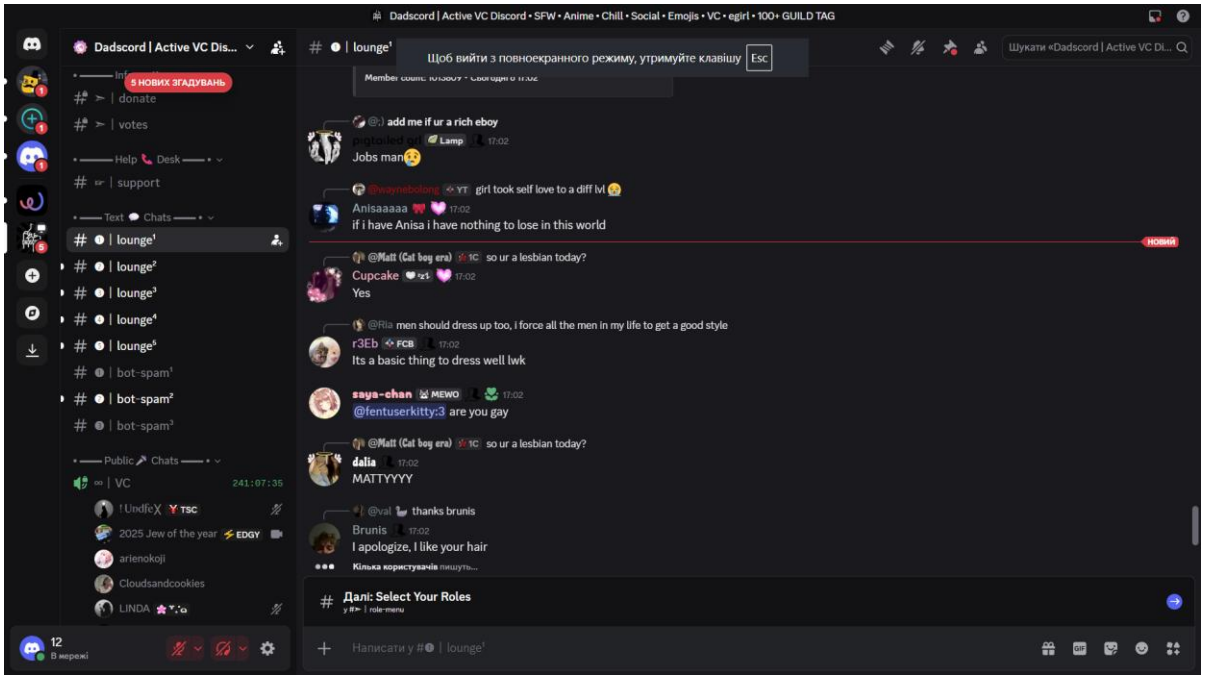


Рис. 1.5. Інтерфейс сервера Discord: приклад хаотичної структури каналів та відсутності спеціалізованих музичних інструментів

1.2.5 Підсумок та порівняльний аналіз існуючих систем

Завершуючи огляд аналогів, можна стверджувати, що сучасний ринок пропонує розрізнений інструментарій: Spotify та SoundCloud домінують у сфері дистрибуції, MuseScore – у сфері нотографії, а Discord – у сфері комунікації. Проте жодна з цих систем не пропонує інтегрованого підходу, який би поєднував ці функції в єдиному захищеному просторі, розгорнутому на автономній інфраструктурі.

Для систематизації результатів дослідження та візуалізації переваг розробленого рішення у порівнянні з аналогами складено таблицю 1.1.

Таким чином, розроблена цифрова платформа заповнює існуючу нішу, пропонуючи музикантам інструмент, який поєднує соціальну взаємодію з професійним навчальним функціоналом та гарантує високий рівень безпеки за рахунок децентралізованого хостингу та криптографічного захисту даних.

Таблиця 1.1

Порівняльний аналіз функціональних та архітектурних характеристик платформ

Критерій порівняння	Spotify	SoundCloud	MuseScore	Discord	Розроблена платформа
Спеціалізація на музичному навчанні	Відсутня	Відсутня	Висока	Відсутня	Висока
Бібліотека партитур (PDF/MusicXML)	Відсутня	Відсутня	Повна	Відсутня	Інтегрована
Захищений P2P відеозв'язок	Відсутній	Відсутній	Відсутній	Часткови й	Нативний (WebRTC)
Наскрізне шифрування (E2EE)	Відсутнє	Відсутнє	Відсутнє	Відсутнє	Реалізовано
Хостинг на IoT (Edge Computing)	Відсутній	Відсутній	Відсутній	Відсутній	Підтримується
Автономність від хмарних гігантів	Відсутня	Відсутня	Відсутня	Відсутня	Повна (IoT вузол)
Управління пропущеними (Real-time)	Базове	Базове	Відсутнє	Високе	Реалізовано

1.3 Аналіз методів захищеної взаємодії та використання IoT-інфраструктури

Розвиток сучасних веборієнтованих екосистем вимагає перегляду традиційних моделей безпеки. У контексті музичного ком’юніті, де відбувається обмін авторським контентом та приватними навчальними сесіями, захист даних має базуватися на принципах нульової довіри (Zero Trust) та децентралізації інфраструктури.

1.3.1 Методи наскрізного шифрування (E2EE) у системах реального часу

Наскрізне шифрування (End-to-End Encryption, E2EE) є золотим стандартом забезпечення конфіденційності в сучасних системах зв'язку. Його ключова відмінність від шифрування на транспортному рівні (TLS/SSL) полягає в тому, що ключі шифрування доступні лише кінцевим точкам взаємодії (клієнтам), а сервер виконує роль виключно ретранслятора зашифрованих пакетів, не маючи технічної можливості дешифрувати контент.

Математичною основою більшості протоколів E2EE (зокрема Signal Protocol, який є референсним для даної роботи) є алгоритм обміну ключами Діффі-Геллмана на еліптичних кривих (ECDH). Процес встановлення захищеного каналу базується на властивостях модулярної арифметики. Якщо користувач A та користувач B хочуть згенерувати спільний секрет, вони обирають спільні параметри: просте число p та основу g.

Кожен обирає свій секретний ключ (a та b відповідно), після чого обчислюються публічні ключі:

(1.1)

$$A_{pub} = g^a \pmod p$$

(1.2)

$$B_{pub} = g^b \pmod p$$

Обмінявшись публічними ключами, сторони обчислюють спільний секрет S:

(1.3)

$$S = (B_{pub})^a \bmod p = (A_{pub})^b \bmod p$$

У розробленій платформі для захисту текстових повідомлень та метаданих чатів використовується комбінація симетричного та асиметричного шифрування, проілюстрована на діаграмі. Для кожного сеансу генерується унікальний сесійний ключ (AES-256). Це забезпечує властивість прямої секретності (Forward Secrecy): навіть якщо один із ключів буде компрометовано у майбутньому, це не дозволить зловмиснику розшифрувати попередні повідомлення [17, 18].

Окрему увагу в роботі приділено концепції Recovery Key (ключа відновлення). Оскільки архітектура E2EE передбачає, що сервер не зберігає паролі у відкритому вигляді та не має доступу до приватних ключів, втрата пароля означає незворотну втрату доступу до зашифрованих даних користувача. Впровадження механізму генерації 128-бітного ключа відновлення при реєстрації дозволяє локально (на стороні клієнта) регенерувати криптографічну пару. Це є необхідним інженерним компромісом між абсолютною безпекою (Zero Knowledge) та зручністю використання системи (User Experience).

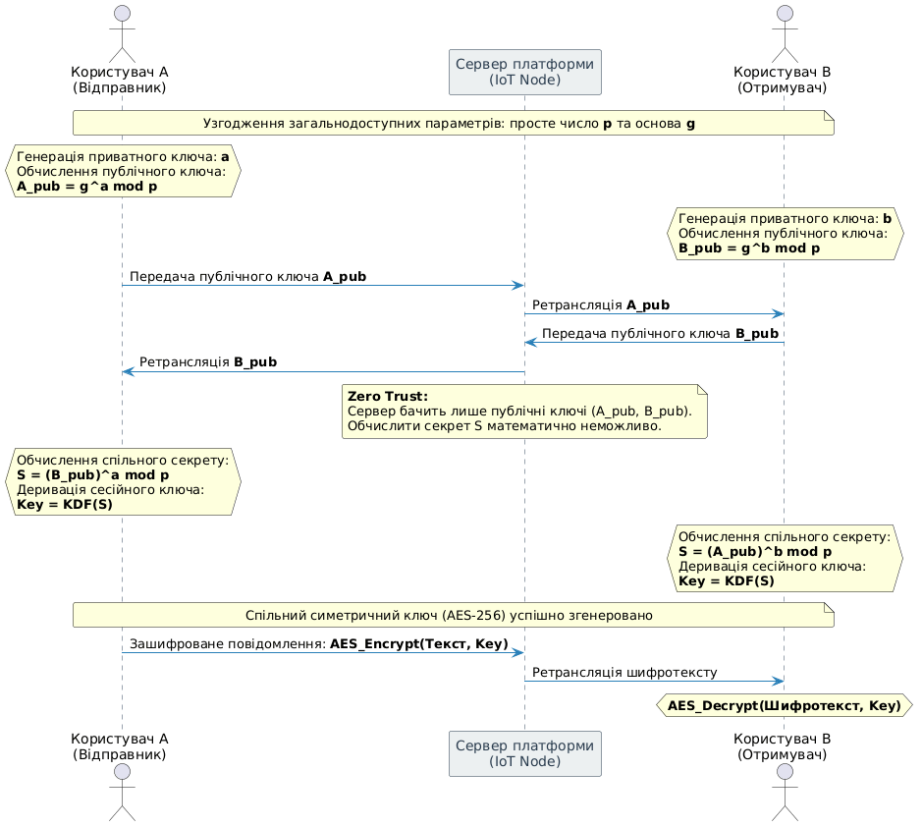


Рис. 1.6 Діаграма послідовності обміну криптографічними ключами та встановлення E2EE з'єднання

1.3.2 Технологія WebRTC та організація потокової передачі медіаданих у реальному часі

Для реалізації функціоналу відеодзвінків, трансляції екрана та групових конференцій у розроблюваній платформі обрано технологію WebRTC (Web Real-

Time Communication). WebRTC є відкритим стандартом та набором протоколів, що дозволяють браузерам та мобільним додаткам здійснювати передачу аудіо, відео та довільних даних безпосередньо між вузлами (Peer-to-Peer, P2P) без необхідності встановлення сторонніх плагінів або проміжного медіа-сервера для обробки потоку [7, 8, 9].

Архітектура WebRTC базується на трьох основних API:

1. MediaStream (getUserMedia): забезпечує доступ до локальних пристроїв введення (мікрофон, камера) та захоплення відеопотоку з екрана.

2. RTCPeerConnection: ядро технології, що відповідає за стабільне та ефективне з'єднання між двома браузерами, керуючи кодеками, смугою пропускання та втратами пакетів.

3. RTCDataChannel: дозволяє передавати метадані або файли з низькою затримкою за протоколом SCTP.

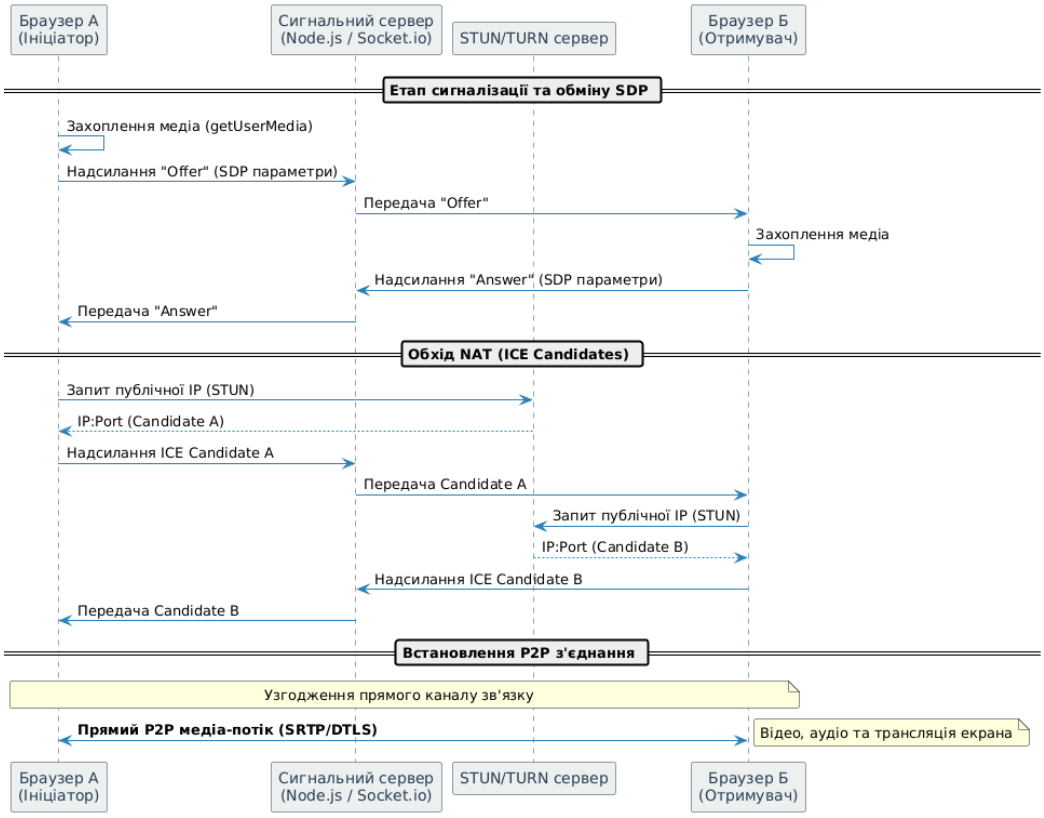


Рис. 1.7 Архітектурну схему встановлення WebRTC-з'єднання через сигнальний сервер, розгорнутий на IoT-вузлі

Особливістю WebRTC є те, що попри P2P-природу, процес встановлення

з'єднання потребує централізованого етапу – сигналізації (Signaling). Оскільки браузери знаходяться за NAT (Network Address Translation) або брандмауерами, вони не знають прямих IP-адрес один одного. Для вирішення цієї проблеми використовується механізм ICE (Interactive Connectivity Establishment), що залучає сервери STUN та TURN.

Важливою перевагою використання WebRTC у розробленій системі є безпека. Всі медіапотоки за стандартом захищені протоколами SRTP (Secure Real-time Transport Protocol) та DTLS (Datagram Transport Layer Security), що забезпечує шифрування трафіку «з коробки». Для реалізації групових дзвінків та запису зустрічей у даному проєкті застосовується гібридний підхід.

Хоча чистий P2P (Mesh-архітектура) ідеально підходить для дзвінків один-на-один, для груп понад 3-4 особи навантаження на клієнтський вузол зростає експоненціально. Як зазначено в роботі про системи відеоконференцій на базі WebRTC [19], для стабільної роботи на ресурсообмежених пристроях (таких як IoT-вузли) критично важливо оптимізувати процес передачі пакетів та використовувати адаптивні алгоритми керування бітрейтом.

1.3.3 Периферійні обчислення (Edge Computing) та IoT-інфраструктура як альтернатива хмарному хостингу

Традиційна парадигма розгортання та хостингу сучасних веб-застосунків (зокрема мультимедійних платформ) базується на використанні централізованих ресурсів глобальних хмарних провайдерів, таких як Amazon Web Services (AWS), Google Cloud або Microsoft Azure. Попри високу масштабованість, такий підхід має два суттєві недоліки для незалежних розробників та приватних спільнот: фінансова залежність від тарифної політики провайдера (Cloud Lock-in) та втрата повного контролю над фізичним розміщенням даних.

Для вирішення цих проблем у даній кваліфікаційній роботі застосовано концепцію периферійних обчислень (Edge Computing) та самостійного хостингу (Self-hosting) на базі IoT-інфраструктури. Периферійні обчислення передбачають перенесення обчислювальних потужностей та баз даних ближче до кінцевого

користувача або розміщення їх у локальній ізольованій мережі.

У якості апаратної платформи для ядра системи (серверної частини та бази даних) обрано мікрокомп’ютер класу одноплатних систем (SBC) – Raspberry Pi. Як демонструють дослідження продуктивності IoT-пристроїв [20, 21], сучасні покоління Raspberry Pi (зокрема Raspberry Pi 4/5) володіють достатніми апаратними характеристиками (ARM-архітектура процесора, 4–8 ГБ оперативної пам'яті) для розгортання повноцінних контейнеризованих веб-застосунків.

Переваги використання IoT-вузла для розробленої платформи:

1. Цифрова суверенність (Data Sovereignty). Забезпечується повний фізичний контроль над сервером, де зберігаються зашифровані повідомлення користувачів, метадані профілів та музичні чернетки. Дані не залишають межі апаратного вузла власника платформи, що унеможливлює їх аналіз сторонніми корпораціями (як у випадку з комерційними хмарами) [3, 13].

2. Економічна ефективність. Використання мікрокомп'ютера вимагає лише одноразових витрат на апаратне забезпечення. За результатами порівняльного аналізу [22, 23, 24], оренда хмарного інстансу (наприклад, AWS t3.medium), еквівалентного за обчислювальними ресурсами Raspberry Pi, потребує постійних щомісячних витрат, які є економічно недоцільними для локального музичного ком'юніті, що не має на меті агресивної монетизації.

3. Енергоефективність. Мікрокомп'ютери споживають у десятки разів менше електроенергії (близько 5-10 Вт), що робить їх ідеальним рішенням для цілодобового хостингу (24/7) у домашніх або студійних умовах [25].

Разом з тим, розгортання високонавантаженого серверного програмного забезпечення (Node.js, PostgreSQL, Socket.io) на IoT-пристрої вимагає оптимізації архітектури. Обмеженість ресурсів (особливо швидкості дискових операцій на microSD картах та об'єму оперативної пам'яті) зумовлює необхідність агресивного кешування, обмеження частоти запитів (Rate Limiting) та ефективного використання контейнеризації (Docker).

Крім того, виникає фундаментальна мережева проблема: IoT-пристрої зазвичай знаходяться за NAT провайдера і не мають статичної "білої" IP-адреси,

що унеможливлює прямий доступ користувачів до веб-додатка з глобальної мережі Інтернет. Це вимагає впровадження спеціалізованих технологій тунелювання трафіку.

1.3.4 Технології тунелювання трафіку та забезпечення віддаленого доступу до IoT-інфраструктури

Однією з головних проблем при розгортанні веб-застосунків на периферійних IoT-пристроях є мережеві маршрути. У типових системах локальні сервери (наприклад, Raspberry Pi) розташовані за NAT провайдера та брандмауером маршрутизатора. Зазвичай, щоб отримати доступ до них зовні, використовують пронесення портів та статичну публічну IP-адресу. Але цей метод дуже небезпечний, оскільки він дає прямий доступ до локальної мережі, роблячи IoT-вузол вразливим до DDoS-атак, сканувань портів та автоматичних підбірок паролів (Brute-force) [26, 25, 27].

Щоб безпечно розв'язати проблему доступу, у цій роботі застосовано технологію зворотного тунелю.В такому випадку з'єднання ініціюється не ззовні (від клієнта до сервера), а зсередини (від сервера до зовнішнього шлюзу). На IoT-пристрої встановлюється спеціальний демон (агент), який підтримує постійне, зашифроване вихідне з'єднання з глобальним магістральним мурежем.

Під час проектування системи проаналізовано два популярних ринка рішення: Ngrok та Cloudflare Tunnels[26, 27, 28].

Ngrok є стандартом для швидкого тунелювання при розробці (Development environment). Агент Ngrok створює тимчасові URL-адреси, які перенаправляють трафік на локальний localhost. Але для використання у виробництві (Production) ця платформа має недоліки: вона дорожча тарифних планів для підтримки статичних доменів, обмежена пропускна здатність та архітектура, яка може додавати небажану затримку (latency) під час трансляції медіа-трафіку (WebRTC) [26, 25, 27].

Як альтернативу, для реалізації у розробленій платформі обрано Cloudflare Tunnels (раніше Argo Tunnel). Його суть полягає у тому, що легковаговий агент

cloudflared встановлюється безпосередньо на Raspberry Pi. Замість того, щоб маршрутизувати трафік через вузькоспрямовані сервери, cloudflared підключає IoT-вузол до найближчого дата-центру глобальної мережі Anycast від Cloudflare.

Переваги використання Cloudflare Tunnels в архітектурі музичної платформи:

1. Zero Trust Network Access (ZTNA). Відпадає необхідність відкривати будь-які вхідні порти на маршрутизаторі. IoT-сервер залишається повністю «невидимим» для сканерів у публічному Інтернеті [27].

2. Захист від DDoS та WAF. Оскільки весь вхідний трафік спочатку проходить через периферійні сервери Cloudflare, платформа автоматично отримує захист від розподілених атак на відмову в обслуговуванні (DDoS) та фільтрацію шкідливих запитів через Web Application Firewall (WAF) [27].

3. Оптимізація продуктивності. Інтеграція тунелю з глобальною мережею доставки контенту (CDN) дозволяє кешувати статичні активи (наприклад, зображення профілів або фрагменти нотних партитур) на периферійних вузлах, суттєво знижуючи навантаження на обмежені ресурси Raspberry Pi.

Для наочної демонстрації механізму захисту локальної інфраструктури розроблено схему (Рисунок 1.8), що відображає маршрутизацію трафіку.

Варто зазначити, що попри високий рівень захисту, механізми тунелювання вимагають суворого моніторингу. Згідно зі звітами з кібербезпеки [28], зловмисники можуть використовувати скомпрометовані тунелі (техніка "Tunnel Vision") для ексфільтрації даних. Тому у розробленій системі агент cloudflared запускається у строго ізольованому Docker-контейнері з обмеженими правами доступу виключно до контейнера веб-сервера, що відповідає принципу мінімальних привілеїв.

1.4 Визначення проблем предметної галузі

На основі проведеного аналізу сучасних музичних платформ (Spotify,

SoundCloud, MuseScore), комунікаційних систем (Discord) та інфраструктурних рішень (хмарні сервіси проти IoT), було виявлено низку фундаментальних проблем, які перешкоджають ефективній та безпечній взаємодії у межах професійних музичних спільнот.

1. Фрагментація інструментарію. Жодна з існуючих популярних платформ не забезпечує комплексного робочого середовища. Користувачі змушені використовувати одну систему для зберігання нот (MuseScore), іншу – для прослуховування аудіо (SoundCloud), а третю – для комунікації (Discord). Це порушує цілісність творчого та навчального процесу.

2. Загрози конфіденційності та відсутність E2EE. Комерційні платформи збирають та аналізують метадані користувачів для таргетованої реклами. Відсутність наскрізного шифрування у більшості масових месенджерів створює ризики витоку інтелектуальної власності (незавершених треків, текстів пісень) та приватних обговорень.

3. Штучні бар'єри в освітньому процесі. Агресивна монетизація функціоналу обміну базовими файлами (наприклад, платне завантаження партитур, як це реалізовано в комерційних бібліотеках) обмежує можливості локальних ком'юніті у вільному розповсюдженні навчальних матеріалів.

4. Залежність від централізованих хмар (Cloud Lock-in). Хостинг платформ на базі великих провайдерів (AWS, Google Cloud) позбавляє власників спільнот суверенітету над даними та вимагає постійних фінансових витрат, що є нерентабельним для незалежних розробників та невеликих музичних гуртів.

5. Вразливість відкритої IoT-інфраструктури. Хоча використання мікрокомп'ютерів (Raspberry Pi) вирішує проблему суверенітету даних, класичні методи підключення через прокидання портів (Port Forwarding) роблять такі пристрої вразливими до кібератак, що вимагає впровадження альтернативних методів тунелювання трафіку.

Виявлені проблеми підтверджують необхідність створення єдиної, спеціалізованої та безпечної цифрової екосистеми, яка б функціонувала на незалежній апаратній базі.

1.5 Постановка завдання на розробку цифрової платформи

Головним завданням даної кваліфікаційної роботи є проєктування та програмна реалізація цифрової платформи для музичного ком’юніті, яка об’єднує інструменти пошуку музичних композицій, засоби онлайн-навчання та підсистему захищеної взаємодії користувачів, з подальшим розгортанням системи на IoT-інфраструктурі.

Для досягнення поставленої мети розроблюване програмне забезпечення повинно вирішувати такі задачі:

1. Забезпечення ізольованого та захищеного середовища для спілкування (чати, аудіо- та відеодзвінки) з використанням алгоритмів наскрізного шифрування (E2EE) та P2P-з'єднань.
2. Створення гнучкої системи управління користувацьким контентом, що включає бібліотеку нотних партитур (Score Library) та фонотеку (Track Library) з можливістю створення плейлистів.
3. Реалізація соціальної механіки взаємодії: розширений пошук музикантів за специфічними критеріями (інструменти, жанри, локація), система запитів у друзі та формування груп.
4. Оптимізація серверної архітектури для стабільної роботи на обмежених обчислювальних ресурсах мікрокомп'ютера Raspberry Pi з використанням контейнеризації (Docker).
5. Забезпечення безпечного доступу до локального сервера з глобальної мережі без використання статичної IP-адреси за допомогою зворотного тунелювання (Cloudflare Tunnels).

1.6 Визначення функціональних та нефункціональних вимог до програмного забезпечення

Для забезпечення відповідності розроблюваної системи потребам цільової аудиторії сформовано чіткий перелік вимог до програмного продукту.

Функціональні вимоги (Functional Requirements):

1. Підсистема автентифікації та безпеки акаунтів: Реєстрація та

авторизація користувачів із генерацією унікального ключа відновлення (Recovery Key) для доступу в разі втрати пароля.

- Захист від Brute-force атак: тимчасове блокування IP-адреси (Rate Limiting) після 5 невдалих спроб входу (на 15 хвилин) та обмеження створення більше двох акаунтів з однієї IP-адреси (на 1 годину).

2. Управління профілем та соціальна взаємодія (Onboarding & Networking):

- Можливість детального налаштування профілю (біографія, локація, теги музичних інструментів та володіння мовами).

- Система пошуку користувачів із розширеною фільтрацією та можливістю надсилання, відхилення або прийняття запитів на з'єднання (Friend Requests).

3. Підсистема захищеної комунікації (Chat & WebRTC):

- Підтримка особистих (Direct) та групових чатів із функціями закріплення (pin), відповіді (reply), пересилання (forward), редагування та видалення власних повідомлень.

- Можливість закріплення важливих чатів у верхній частині списку та на панелі навігації, а також функція вимкнення сповіщень (mute).

- Відео- та аудіодзвінки в реальному часі з підтримкою трансляції екрана (Screen Sharing), запису зустрічі та індивідуального керування потоками (вимкнення мікрофона/камери, локальний mute учасників).

- Оновлення лічильників пропущених повідомлень та статусів (Read Receipts) у реальному часі за допомогою WebSockets.

4. Управління музичним контентом:

- Бібліотека композицій (Score Library) з можливістю створення карток із аудіо-прев'ю, тегами та метаданими.

- Бібліотека треків із функцією завантаження обкладинок, створення користувацьких плейлистів та додавання треків до "улюблених".

- Глобальний аудіоплеєр, що функціонує у фоновому режимі (у бічній або верхній панелі навігації) під час навігації сторінками.

5. Інтерфейс користувача (UI/UX):

- Глобальна система швидкого пошуку (Command Palette) по композиціях, треках, чатах та користувачах.

- Підтримка перемикання між світлою та темною темами оформлення через контекстне меню профілю.

Нефункціональні вимоги (Non-Functional Requirements):

1. Безпека (Security): Наскрізне шифрування особистих повідомлень; ізоляція бази даних та бекенду від прямого доступу з Інтернету (маршрутизація виключно через Cloudflare Tunnels).

2. Продуктивність (Performance): Застосунок має стабільно функціонувати на базі мікрокомп'ютера Raspberry Pi (ARM-архітектура), споживаючи мінімальний обсяг оперативної пам'яті за рахунок оптимізації бази даних PostgreSQL (ORM Prisma).

3. Адаптивність (Usability): Інтерфейс клієнтської частини має бути побудований за принципами Responsive Web Design для коректного відображення як на десктопних, так і на мобільних пристроях.

2 ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ЦИФРОВОЇ ПЛАТФОРМИ

2.1 Загальна архітектура системи та обґрунтування вибору технологічного стеку

Розробка потужних соціальних платформ високого навантаження, які включають обмін медіа-контентом та спілкування в реальному часі, вимагає ретельного підбору архітектурних патернів. З огляду на основні нефункціональні вимоги щодо розгортання серверної частини на периферійному IoT-вузлі (Raspberry Pi), архітектура системи повинна бути максимально ефективною у використанні оперативної пам'яті та процесорного часу.

У цій магістерській роботі реалізована гібридна архітектурна модель, що поєднує класичну клієнт-серверну архітектуру (Client-Server) для керування бізнес-логікою та збереження стану з одноранговою мережею (Peer-to-Peer, P2P) для передачі важкого медіатрафіку.

Для реалізації проекту було обрано сучасний JavaScript/TypeScript стек технологій, що дає змогу отримати одну мовну екосистему для клієнтської та серверної сторінок, таким чином зменшуючи накладні витрати на серіалізацію даних.

Клієнтська частина (Frontend) побудована за принципом односторінкового застосунку (Single Page Application, SPA) на базі бібліотеки React. Вибір React обґрунтовується його компонентним підходом та механізмом віртуального DOM (Virtual DOM), що дозволяє ефективно перемальовувати лише ті частини інтерфейсу, які змінилися (наприклад, лічильники непрочитаних повідомлень у реальному часі). Управління глобальним станом системи (автентифікація, черга аудіоплеєра, статус WebRTC-з'єднань) реалізовано за допомогою бібліотеки Zustand. На відміну від традиційного Redux, Zustand базується на принципі хуків (hooks) і має мінімальний розмір (менше 1 КБ), що критично важливо для зменшення розміру кінцевого бандла (bundle size) та швидкого завантаження

додатку в браузері клієнта. Для стилізації інтерфейсу та забезпечення адаптивності використано фреймворк Tailwind CSS.

Серверна частина (Backend) реалізована на базі середовища виконання Node.js з використанням фреймворку Express. Асинхронна, подієво-орієнтована (event-driven) природа Node.js з неблокуючим вводом/виводом (non-blocking I/O) ідеально підходить для розгортання на ARM-архітектурі Raspberry Pi. Це дозволяє серверу одночасно обробляти тисячі постійних з'єднань Socket.io (що відповідають за індикатори онлайн-статусу та чат) в одному потоці, не витрачаючи пам'ять на створення нових потоків для кожного клієнта.

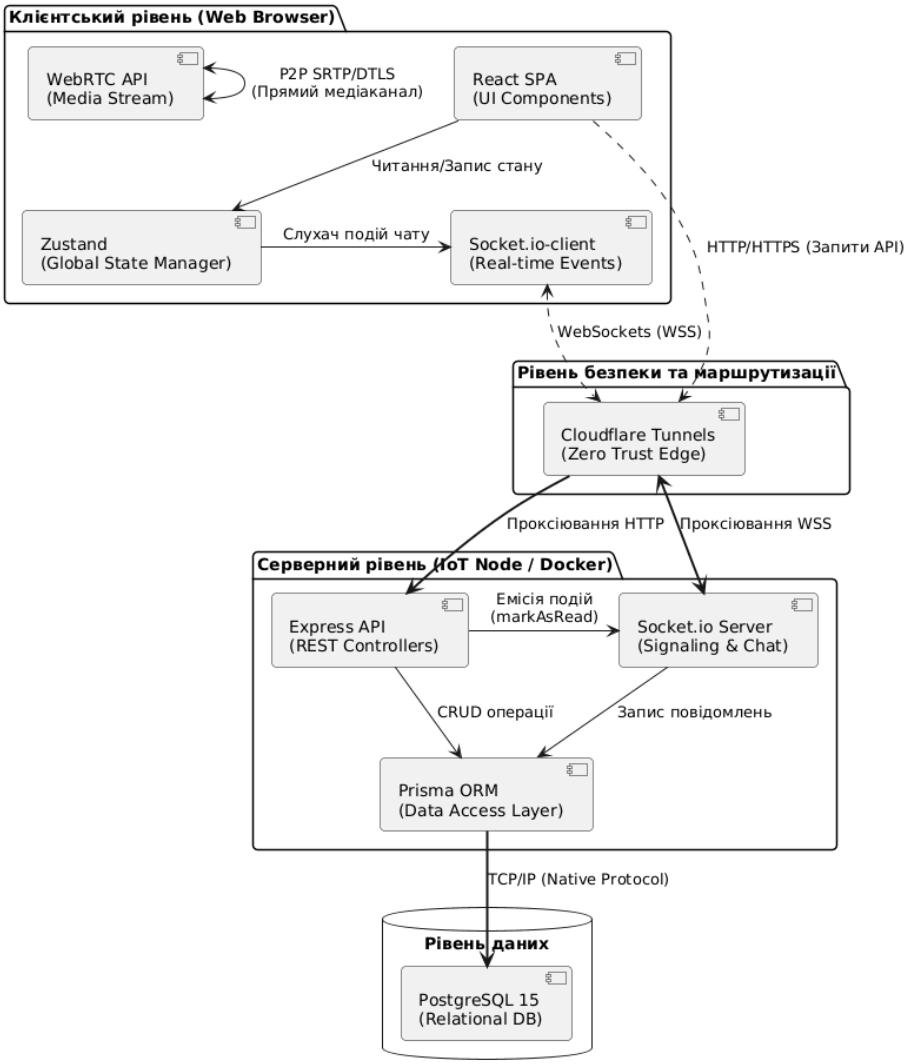


Рис. 2.1 Архітектурна діаграма компонентів та технологічного стеку платформи

Рівень збереження даних (Database Layer) базується на об'єктно-реляційній системі управління базами даних PostgreSQL. Вибір реляційної СКБД зумовлений наявністю складних зв'язків між сутностями системи (користувачі, друзі, групові чати, композиції, теги інструментів). Для взаємодії з базою даних використано сучасний ORM (Object-Relational Mapping) інструмент Prisma. Prisma генерує типізований клієнт бази даних, що мінімізує ризик виникнення SQL-ін'єкцій та забезпечує сувору консистентність даних під час створення музичних профілів (Onboarding).

Структурну взаємодію обраного технологічного стеку та розподіл відповідальності між рівнями наведено на діаграмі компонентів (Рисунок 2.1).

Як видно з наведеної архітектури, класичні HTTP-запити відповідають лише за отримання статичних даних (завантаження списку композицій, профілів користувачів), тоді як динамічний стан (нові повідомлення, сигналізація для відеодзвінків) передається через постійне WebSocket-з'єднання. Прямий медіапотік взагалі оминає сервер, розвантажуючи IoT-інфраструктуру, що є ключовим інженерним рішенням для забезпечення масштабованості системи. Усі серверні компоненти інкапсульовані в ізольовані середовища за допомогою технології контейнеризації Docker, що гарантує ідентичність роботи програмного забезпечення незалежно від базової операційної системи хоста.

2.2 Проєктування IoT-інфраструктури та розгортання на апаратній платформі

Впровадження концепції периферійних обчислень (Edge Computing) в розроблену систему вимагає грамотного проектування на рівні апаратних засобів та операційної системи. Якщо у випадку хмарних інстансів, які пропонують віртуалізовані ресурси, розгортання соціальної мережі з високим навантаженням на окремому IoT-пристрої є роботою в умовах жорстко обмежених ресурсів.

Як центральний сервер платформи був вибраний мікрокомп'ютер Raspberry Pi (платформа ARM64). Щоб підтримувати реляційну базу даних (PostgreSQL) та обробляти велику кількість одночасних Syncronous WebSocket-з'єднань, цільовий

пристрій повинен мати не менш як 4, 8 ГБ оперативної пам'яті (RAM) та мати встановлений зовнішній твердотільний диск (SSD), підключений через порт USB 3.0, замість базового варіанту з використанням карти пам'яті microSD. Це інженерне рішення дозволяє уникнути проблеми пов'язаних зі зношенням флеш-пам'яті через інтенсивні цикли запису/читання. (I/O)

Базова операційна система, це Linux-розподіл без графічної оболонки (Headless OS, наприклад, Ubuntu Server або Raspberry Pi OS Lite). Виключення графічної оболонки може вивільнити до 30% ресурсів пристрою, які можна направити на потреби бізнес-логіки додатку.

Для організації життєвого циклу додатків впроваджено технологію контейнеризації Docker. Це дозволяє уникнути проблеми управління залежностями (Dependency Management) і забезпечити повну ізоляцію процесів. Замість того, щоб встановлювати Node.js, бази даних і веб-сервери безпосередньо на ОС хоста (Host OS), всі компоненти платформи запаковані в контейнери і керуються за допомогою Docker Compose.

Архітектуру розгортання (Deployment Architecture) на IoT-вузлі наведено на діаграмі розгортання (Рисунок 2.2).

Як проілюстровано на діаграмі, контейнери об'єднані в ізольовану віртуальну мережу (Docker Bridge Network). Жоден із портів контейнерів (наприклад, 5432 для бази даних або 3000 для API) не прокидається на фізичний мережевий інтерфейс Raspberry Pi (немає ports директиви у конфігурації для публічного доступу). Єдиним компонентом, який взаємодіє із зовнішнім світом, є контейнер агента cloudflared. Він встановлює вихідний тунель та проксіює трафік всередину віртуальної мережі Docker, скеровуючи його на Nginx (для статики клієнтської частини) або на Node.js API (для бізнес-логіки та WebSockets).

Такий підхід забезпечує концепцію "інфраструктура як код" (Infrastructure as Code, IaC). Повне відновлення працездатності системи на новому апаратному вузлі у разі критичного збою вимагає лише перенесення файлу docker-compose.yml, конфігураційного файлу бази даних (Volumes) та виконання єдиної команди ініціалізації, що відповідає сучасним

індустріальним стандартам надійності (Reliability Engineering).

2.3 Моделювання сценаріїв взаємодії користувачів (UML Use Case діаграми)

Для формалізації вимог до розроблюваної системи та визначення меж її функціональності необхідно побудувати модель прецедентів (Use Case Model). Це дозволяє візуалізувати взаємодію між зовнішніми акторами (користувачами) та внутрішніми сервісами платформи.

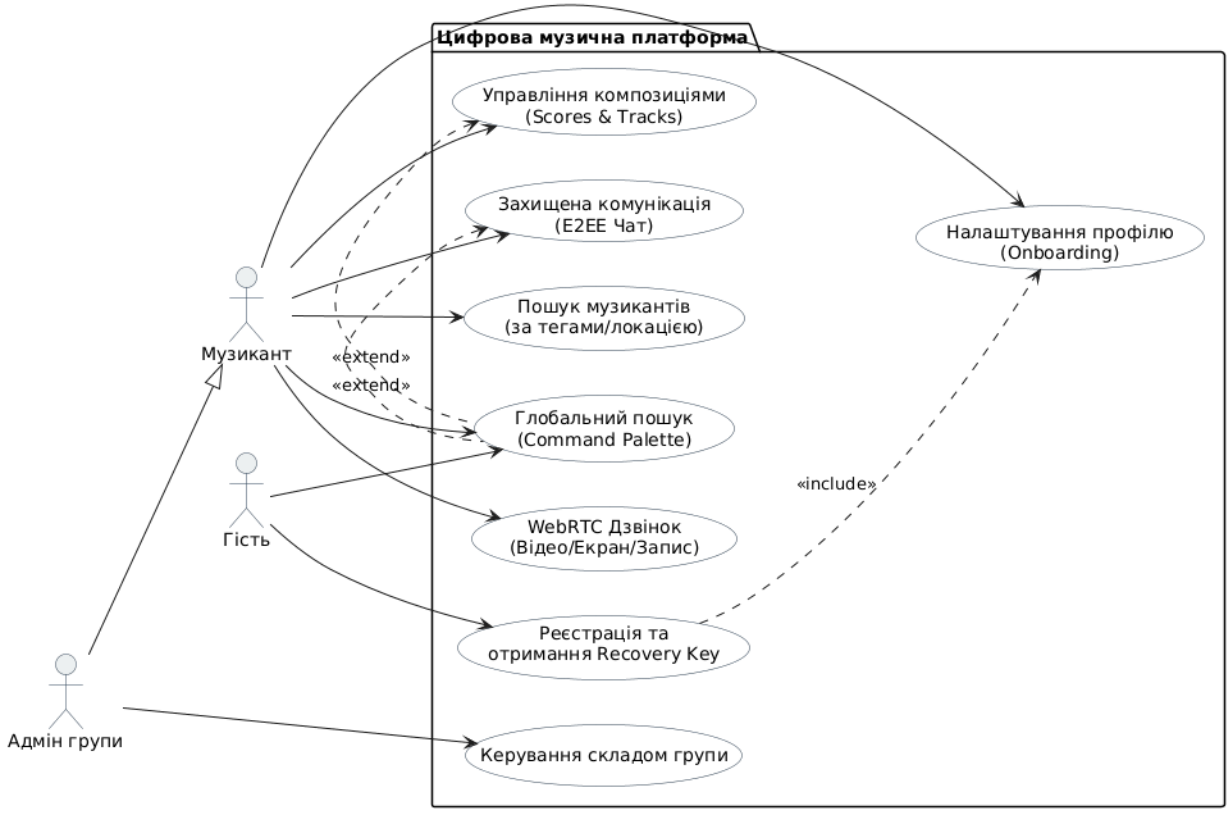


Рис. 2.3 UML-діаграма прецедентів (Use Case Diagram) розробленої системи

У розробленій системі виділено три основні ролі (актори):

1. Гість (Незареєстрований користувач): має обмежений доступ, може переглядати публічні сторінки, проходити реєстрацію або відновлювати пароль.
2. Музикант (Авторизований користувач): основний актор, який має доступ до всіх соціальних, комунікаційних та музичних інструментів.
3. Адміністратор групи: роль, що надається користувачу всередині

групового чату, дозволяючи керувати складом учасників та налаштуваннями групи.

На Рисунку 2.3 наведено діаграму прецедентів, яка описує ключові сценарії використання платформи.

Детальний аналіз сценаріїв взаємодії дозволяє виділити декілька критичних ланцюжків:

1. Сценарій безпеки: При реєстрації (UC1) система обов'язково генерує ключ відновлення (Recovery Key). Це критичний крок, оскільки архітектура наскрізного шифрування не дозволяє серверу скидати пароль класичними методами без втрати даних.

2. Сценарій комунікації: Процес чату (UC5) та дзвінка (UC6) є взаємопов'язаними. Користувач може ініціювати WebRTC-сесію безпосередньо з вікна чату. Важливою особливістю реалізації є можливість запису зустрічі та трансляції екрана, що є функціональним розширенням стандартного дзвінка.

3. Сценарій навігації: Глобальний пошук через Command Palette (UC7) виступає універсальним інтерфейсом, який дозволяє швидко переходити до профілів музикантів, конкретних треків або діалогів, що значно скорочує кількість кліків (User Journey) порівняно з традиційним меню.

Таке моделювання дозволяє переконатися, що архітектура системи покриває всі функціональні вимоги, визначені в розділі 1.6, та забезпечує логічний поділ прав доступу між учасниками ком'юніті.

2.4 Архітектура підсистеми захищеної взаємодії (WebRTC та наскрізне шифрування E2EE)

Підсистема комунікації в реальному часі є ключовим компонентом розробленої цифрової платформи. Для забезпечення миттєвого обміну повідомленнями, передачі статусів користувачів та маршрутизації сигнального трафіку для відеодзвінків було реалізовано сигнальний сервер на базі бібліотеки Socket.io. На відміну від традиційного REST API, який працює за моделлю «запит-відповідь» (request-response), Socket.io забезпечує повнодуплексне (full-

duplex) з'єднання, що дозволяє серверу ініціювати передачу даних клієнту.

2.4.1 Механізми аутентифікації та ізоляції з'єднань

Коли ви використовуєте зворотне тунелювання для безпеки та з'єднання IoT-інфраструктури, ви можете встановити жорсткі обмеження політики спільного використання ресурсів (CORS), які дозволяють підключатися лише з певних адрес. У нашому випадку дозволені тільки локальні запити та з'єднання, що проходять через безпечні тунелі Cloudflare (домени *.trycloudflare.com).

Щоб забезпечити безпеку кожного WebSocket-з'єднання, ми виконуємо аутентифікацію на етапі "рукостискання" (handshake) за допомогою проміжного програмного забезпечення (middleware). Сервер перехоплює заголовки запиту, витягує jwt-токен із cookies і перевіряє його за допомогою секретного ключа (JWT_SECRET_KEY). Якщо все проходить успішно, ідентифікатор користувача (userId) безпосередньо прив'язується до об'єкта сокета. Це виключає можливість підробки ідентифікатора (Spoofing) і забезпечує, що клієнт отримає тільки події, які призначені безпосередньо йому.

Щоб ефективно перенаправляти трафік у системі, ми застосовуємо концепцію "Кімнат" (Rooms). Під час підключення сокет автоматично приєднується до персональної кімнати user_{userId}. Цей підхід забезпечує, що ми не робимо широкомовного розсилання (broadcasting), а лише відправляємо потрібні дані (наприклад, SDP-офери для WebRTC) безпосередньо отримувачу. Крім того, ми використовуємо кімнати типу chat_{chatId} для наших багатокористувацьких текстових чатів і call_{chatId} для керування груповими відеодзвінками.

2.4.2 Інтеграція з E2EE та управління станом бази даних

Безпечне спілкування в чатах базується на синхронізації шифрувальних станів між клієнтами. Хоча сервер безпосередньо не розшифровує повідомлення, він виступає в ролі довіреного ретранслятора. Зокрема, в цій системі реалізовано обробник події keys_regenerated, що повідомляє всіх учасників чату про необхідність оновлення сесійних ключів для налаштованого каналу E2EE,

забезпечуючи тим самим функцію прямої секретності (Forward Secrecy).

Коли користувач відправляє текстове повідомлення або файл (подія send_message), сервер Signal виконує дві ролі одночасно: ретранслює зашифрований пакет учасникам кімнати та зберігає метадані повідомлення в реляційній базі даних PostgreSQL через ORM Prisma. Таким чином, історія обміну повідомленнями буде доступна користувачам навіть після перезавантаження застосунку, при цьому контент зберігає криптографічну цілісність. Управління статусами читання реалізовано через подію mark_as_read, яка оновлює статус у базі даних та миттєво сповіщає відправника про зміну статусу його повідомлень (Read Receipts).

2.4.3 Сигналізація WebRTC та управління життєвим циклом дзвінків

Встановлення P2P-з'єднання для аудіо- та відеодзвінків реалізовано через передачу сигнальних повідомлень WebRTC. Процес обміну параметрами мультимедіа (SDP) та кандидатами для обходу NAT (ICE Candidates) здійснюється через персональні кімнати користувачів:

webrtc-offer – ініціалізація з'єднання;

webrtc-answer – прийняття параметрів;

webrtc-ice-candidate – обмін мережевими маршрутами.

Управління станом дзвінка тісно інтегровано з базою даних. При ініціації дзвінка (подія call:initiate), сервер перевіряє тип чату (особистий чи груповий), оновлює поле activeCallId відповідного чату в базі даних та розсилає сповіщення call:incoming. Для уникнення "завислих" (ghost) дзвінків у випадку раптового обриву з'єднання (втрата інтернет-зв'язку), у системі реалізовано комплексний обробник події disconnect. Він автоматично сканує всі кімнати дзвінків, в яких перебував користувач, і якщо кімната стає порожньою, сервер анулює activeCallId у базі даних, повертаючи систему до консистентного стану.

Логіку взаємодії та маршрутизації подій відображено на діаграмі послідовності (рис. 2.4).

2.5 Архітектура підсистеми захищеної взаємодії (WebRTC та наскрізне шифрування E2EE)

Для забезпечення надійного зберігання даних, підтримки складної соціальної ієрархії та гарантування цілісності криптографічного матеріалу у розробленій платформі використано об'єктно-реляційну систему управління базами даних (СКБД) PostgreSQL. Взаємодія серверної частини з базою даних абстрагована за допомогою сучасного інструменту об'єктно-реляційного відображення (ORM) Prisma. Це рішення дозволяє здійснювати строгу типізацію на рівні бази даних та ефективно керувати міграціями.

Загальна структура бази даних логічно поділяється на три взаємопов'язані підсистеми: модуль управління користувачами, модуль захищеної комунікації та модуль управління контентом.

Модуль управління користувачами та соціальної взаємодії: Центральною сутністю системи є модель User. Окрім базових ідентифікаційних даних (email, хеш пароля), вона містить масиви специфічних метаданих для процесу адаптації (Onboarding): instrumentsKnown, instrumentsToLearn та spokenLanguages. Використання нативних масивів PostgreSQL замість створення окремих таблиць для цих словників дозволило суттєво зменшити час виконання SQL-запитів при пошуку музикантів. Для управління соціальними зв'язками спроєктовано самореференційні зв'язки friends, а також модель FriendRequest, яка фіксує статус запиту (pending, accepted тощо) між відправником та отримувачем.

Модуль комунікації та управління криптографічними ключами: Реалізація чатів (особистих та групових) базується на моделі Chat, яка містить метадані діалогу (зокрема ідентифікатор активного дзвінка activeCallId), та проміжній моделі ChatMember. Остання забезпечує зв'язок «багато-до-багатьох» (Many-to-Many) між користувачами та чатами, зберігаючи індивідуальні налаштування (роль, статус закріплення на панелі, вимкнення сповіщень). Для забезпечення наскрізного шифрування (E2EE) модель User розширено полями для зберігання публічного ключа (publicKey), зашифрованого приватного ключа та ключа

відновлення (recoveryEncryptedKey). Розподіл симетричних ключів у групових діалогах фіксується у спеціалізованій моделі GroupKey, яка пов'язує конкретний чат, відправника та отримувача із зашифрованим AES-ключем. Повідомлення зберігаються у таблиці Message із підтримкою самореференційного зв'язку replyToId для реалізації функції відповідей на конкретні повідомлення (Threading) та полів для збереження вкладених файлів (fileUrl, fileType).

Модуль медіаконтенту (Аудіотреки та Партитури): Для управління бібліотекою композицій розроблено модель Score, яка містить посилання на PDF-документи та аудіо-прев'ю у хмарному сховищі. Категоризація партитур реалізована через модель ScoreTag та проміжну таблицю ScoreTagMapping, що дозволяє призначати декілька тегів одній композиції. Аудіотреки (Track) інтегруються у списки відтворення через модель Playlist. Враховуючи можливість додавання одного треку до різних плейлистів, зв'язок реалізовано через проміжну сутність PlaylistTrack.

Логічну структуру бази даних платформи у вигляді ER-діаграми (Entity-Relationship) наведено на Рисунку 2.5.

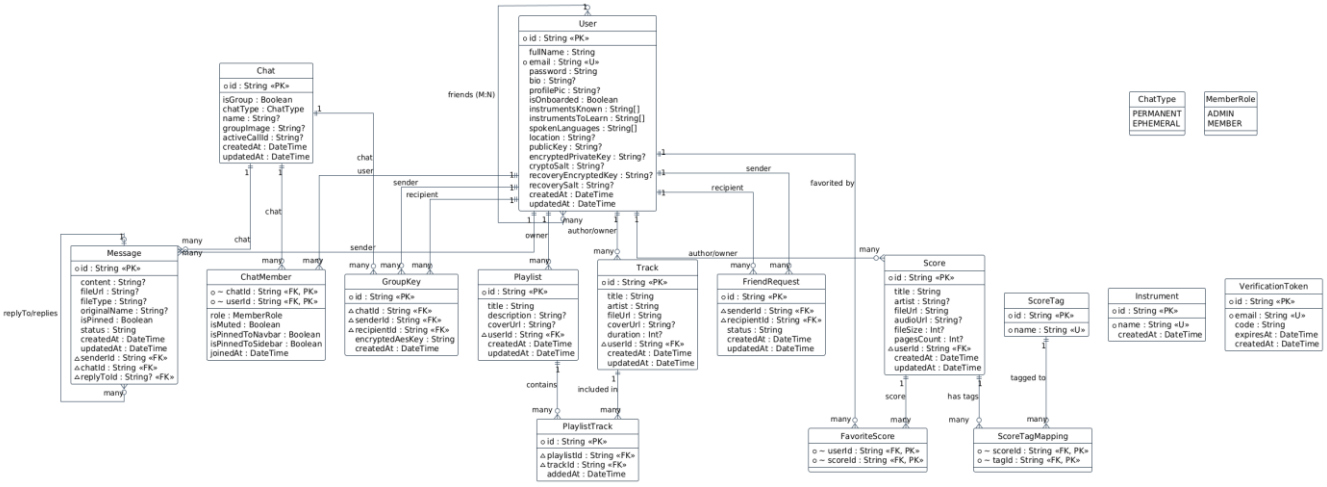


Рис. 2.5 ER-діаграма сутностей реляційної бази даних платформи

Запропонована структура бази даних відповідає третій нормальній формі (3NF), що гарантує відсутність надмірності даних та аномалій оновлення. Використання каскадного видалення (onDelete: Cascade) на рівні зв'язків Prisma

(наприклад, при видаленні користувача автоматично видаляються його повідомлення, ключі та завантажені композиції) забезпечує підтримання посилальної цілісності бази даних без необхідності написання складних тригерів на стороні СКБД.

2.6 Проєктування підсистеми пошуку композицій, обробки аудіо та відображення партитур

Створення сучасного мультимедійного веб-застосунку вимагає вирішення специфічних архітектурних завдань: забезпечення безперервного відтворення медіаконтенту під час навігації (на відміну від класичних багатосторінкових сайтів), ефективного парсингу складних документів (PDF) на стороні клієнта та реалізації наскрізного пошуку по всіх сутностях системи з мінімальною затримкою.

У розробленій платформі ці завдання вирішено шляхом розподілу відповідальності між глобальним менеджером стану, спеціалізованими веб-воркерами (Web Workers) та оптимізованими контролерами бази даних.

2.6.1 Архітектура глобального аудіоплеєра (Audio Engine)

Традиційна проблема музичних веб-платформ полягає у перериванні звуку під час переходу між маршрутами (сторінками). Для забезпечення безперервного відтворення у системі застосовано патерн «Безголового компонента» (Headless Component). Аудіоплеєр розділено на дві частини: візуальний інтерфейс та логічне ядро (AudioEngine.jsx).

Логічне ядро інтегрується безпосередньо в кореневий компонент дерева React і містить нативний HTML5 елемент <audio>, який не відображається в інтерфейсі (className="hidden"). Керування відтворенням, чергою (queue) та синхронізацією часу здійснюється через глобальний менеджер стану на базі бібліотеки Zustand (useAudioStore). Будь-який компонент застосунку (наприклад, картка треку) може ініціювати відтворення, викликавши метод playTrack зі

сховища. Додатково застосовано проміжне програмне забезпечення persist, яке автоматично зберігає користувацькі налаштування (рівень гучності, режим повторення, перемішування) у локальному сховищі браузера (LocalStorage).

2.6.2 Механізм відображення партитур (PDF Rendering)

Для перегляду нотних композицій (Scores) без необхідності їх завантаження на локальний диск клієнта імплементовано компонент прев'ю на базі бібліотеки react-pdf (обгортка над рушієм Mozilla PDF.js).

Оскільки рендеринг PDF-файлів є ресурсомісткою операцією, обробку документів винесено з основного потоку інтерфейсу (Main Thread) у фоновий процес за допомогою Web Worker (pdf.worker.min.mjs). Це інженерне рішення гарантує, що інтерфейс платформи залишається чутливим до дій користувача навіть під час завантаження важких багатосторінкових партитур. Компонент PdfPreview.jsx автоматично генерує динамічну мініатюру першої сторінки композиції та обробляє стани завантаження (Loading) і помилок (Error State) із відповідним візуальним зворотним зв'язком.

2.6.3 Наскрізний пошук (Command Palette)

Для зручного і швидкого пошуку по системі зроблено гарний пошук по всьому додатку (Command Palette), який викликається гарячими клавішами. Відображення пошуку зроблено за допомогою React Portals, що дозволяє рендерити оверлей пошуку поверх усієї DOM-ієрархії, що дуже зручно і не дає z-індексам конфліктувати.

Щоб знизити навантаження на сервер при швидкому введенні тексту використовується Debouncing на 300 мс. На бекенді (Backend) пошук оптимізований одночасним виконанням транзакцій. Замість послідовного пошуку в різних таблицях контролер search.controller.js використовує Promise.all, який дозволяє одночасно виконувати асинхронні запити в User, Chat, Score і Track через Prisma ORM з параметром mode: "insensitive", щоб пошук був нечутливий до регістру символів. Тобто можна отримати комбінований результат по одному

шляху мережі (Round-trip time).

Структурну взаємодію описаних підсистем наведено на діаграмі компонентів (рис. 2.6).

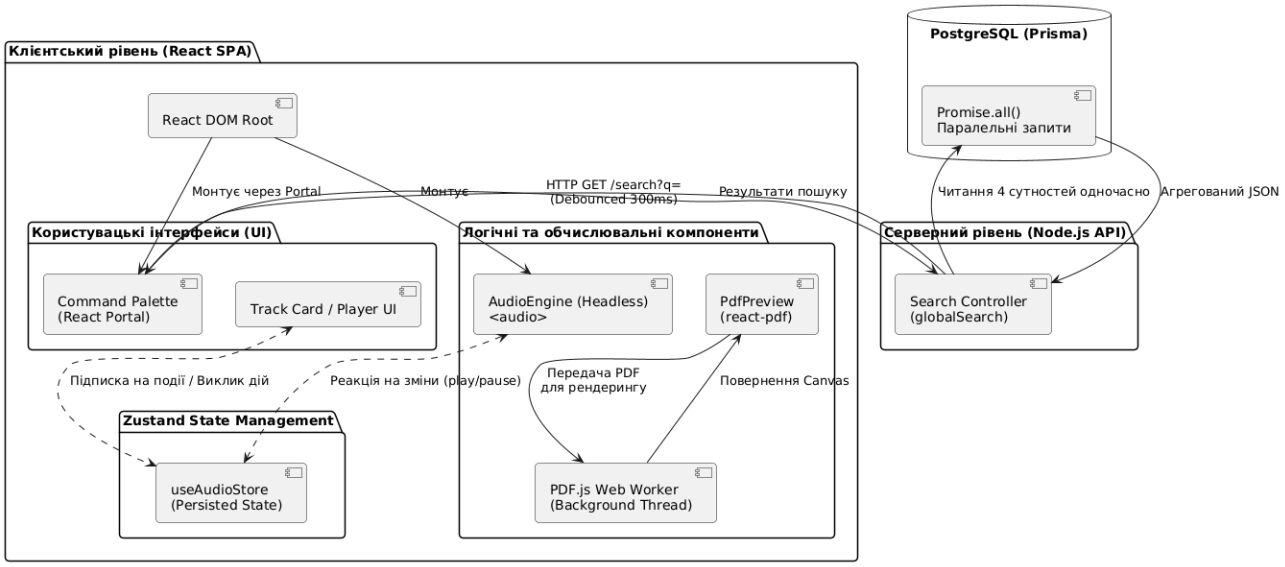


Рис. 2.6 UML-діаграма взаємодії медіа-компонентів та підсистеми глобального пошуку

Таким чином, розроблена архітектура повністю покриває специфічні вимоги до платформи, забезпечуючи оптимальне управління пам'яттю клієнтського браузера та безперебійний користувацький досвід.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ПЛАТФОРМИ

3.1 Налаштування середовища розробки та контейнеризація (Docker) для IoT-інфраструктури

Для забезпечення консистентності програмного середовища, ізоляції процесів та зручності розгортання платформи на мікрокомп'ютері Raspberry Pi застосовано технологію контейнеризації Docker. Оскільки цільовою апаратною архітектурою є ARM-процесори, всі образи контейнерів явно налаштовані на використання платформи linux/arm64.

Процес збірки клієнтської та серверної частин базується на використанні багатоетапних збірок (Multi-stage builds) [29]. Це архітектурне рішення дозволяє розділити етап встановлення залежностей та компіляції коду від фінального образу, що суттєво зменшує розмір готового контейнера та знижує поверхню потенційних кібератак.

Як базовий образ для обох компонентів обрано node:20-alpine. Використання дистрибутиву Alpine Linux зумовлено його мінімальним обсягом (близько 5 МБ), що критично важливо для збереження ресурсів пам'яті на IoT-пристроях.

Реалізація клієнтської частини (Frontend) у Docker-середовищі включає етап builder, на якому виконується встановлення залежностей (через npm ci для точного відтворення дерева залежностей) та збірка статичних файлів React-додатка. У фінальному етапі runner статичні файли переносяться у чистий образ, де для їхньої роздачі використовується легковаговий HTTP-сервер serve.

Серверна частина (Backend) також компілюється у кілька етапів. Важливою інженерною особливістю є встановлення системного пакета openssl на базовому рівні, що є обов'язковою вимогою для коректного функціонування ядра ORM Prisma в середовищі Alpine Linux. На етапі builder виконується генерація типізованого клієнта Prisma (npx prisma generate).

З міркувань безпеки, згідно з принципом мінімальних привілеїв, в обох контейнерах створено непривілейованих користувачів (reactuser та nodeuser з ідентифікаторами UID/GID 1001). Запуск Node.js процесів від імені root повністю виключено.

Оркестрація контейнерів здійснюється за допомогою інструменту Docker Compose. У файлі docker-compose.yml визначено єдину віртуальну місткову мережу (harmonix-net), яка ізолює внутрішній трафік бази даних та API від зовнішньої мережі хоста. Для запобігання вичерпанню оперативної пам'яті (Out-Of-Memory, OOM) на мікрокомп'ютері, для кожного сервісу встановлено жорсткі ліміти апаратних ресурсів (deploy.resources.limits):

1. Клієнтська частина: 1 ГБ.
2. Серверна частина (API + Socket.io): 2 ГБ.
3. Зворотний проксі-сервер (Nginx): 256 МБ.

3.2 Реалізація зворотного проксі-сервера (Nginx), маршрутизації та політик безпеки

Управління вхідним мережевим трафіком, балансування навантаження та термінація з'єднань реалізовані за допомогою вебсервера Nginx, який функціонує як зворотний проксі (Reverse Proxy). Nginx приймає всі запити на 80-му порту контейнера і маршрутизує їх до відповідних внутрішніх сервісів.

Оскільки система розгорнута за технологією Cloudflare Tunnels (як описано в розділі 1.3.4), усі вхідні запити надходять з IP-адрес інфраструктури Cloudflare. Для коректної ідентифікації реальних IP-адрес користувачів (що критично для роботи механізмів блокування та логування) у конфігурації nginx.conf визначено повний перелік IPv4 та IPv6 підмереж Cloudflare у директивах set_real_ip_from. Фактична адреса клієнта витягується із заголовка CF-Connecting-IP [30, 31].

Для захисту серверної частини від атак типу "відмова в обслуговуванні" (DoS) та автоматизованого перебору паролів (Brute-force) імплементовано механізм обмеження частоти запитів (Rate Limiting). Створено зону пам'яті

api_limit об'ємом 10 МБ, яка дозволяє обробляти не більше 15 запитів на секунду з однієї IP-адреси. Для згладжування короткочасних сплесків трафіку застосовано параметр burst=30 без затримки (nodelay).

Мережева маршрутизація поділена на три основні блоки:

1. REST API (/api/): запити проксіюються до контейнера backend:3001.

2. WebSockets (/socket.io/): налаштовано підтримку постійних з'єднань через передачу заголовків Upgrade: websocket та Connection: "upgrade". Таймаути на читання та відправлення збільшено до 86400 секунд (24 години) для уникнення розриву довготривалих P2P-сигнальних сесій, а буферизацію Nginx (proxy_buffering off) вимкнено для забезпечення мінімальної затримки при обміні E2EE повідомленнями.

3. Клієнтський додаток (/): запити направляються до frontend:3000. Оскільки клієнт побудований як Single Page Application (SPA), реалізовано перехоплення помилки 404 із подальшим перенаправленням на /index.html для коректної роботи клієнтського маршрутизатора (React Router).

Безпека на рівні HTTP-заголовків (Security Headers) забезпечується додаванням строгих політик до кожної відповіді сервера [30, 31]:

- Strict-Transport-Security (HSTS) – примусове використання HTTPS;
- X-Frame-Options SAMEORIGIN – захист від атак Clickjacking;
- X-Content-Type-Options nosniff – заборона браузеру ігнорувати MIME-типи;
- Content-Security-Policy (CSP) – комплексна політика, що обмежує джерела завантаження скриптів, стилів та медіа-контенту, дозволяючи підключення лише з довірених доменів (зокрема, Cloudflare Insights) та роботу з blob: об'єктами, що необхідно для рендерингу PDF-партитур через Web Workers. Дозволено підключення WebSocket (ws:, wss:) для роботи чатів.

3.3 Розробка клієнтської частини та інтерфейсу користувача (React, Zustand, Tailwind CSS)

Програмна реалізація клієнтської частини (Frontend) виконана у вигляді односторінкового застосунку (Single Page Application) з використанням бібліотеки React. Для забезпечення сучасного користувацького досвіду (UX) та високої швидкості рендерингу архітектура інтерфейсу базується на функціональних компонентах та хуках (Hooks).

Стилізація платформи реалізована за допомогою фреймворку Tailwind CSS у поєднанні з компонентною бібліотекою DaisyUI. Це дозволило відмовитися від написання громіздких CSS-файлів на користь утилітарних класів (Utility-first CSS). Відповідно до конфігураційного файлу tailwind.config.js, у системі розроблено власну дизайн-систему, яка включає світлу (light) та темну (dark) теми. Колірна палітра використовує семантичні змінні: первинний колір (primary: #3b82f6), фонові відтінки для створення глибини (base-100, base-200, base-300) та кольори станів (success, warning, error). Типографіка застосунку базується на сімействі шрифтів "Inter", що забезпечує високу читабельність тексту на різних роздільних здатностях екрана.

Для управління глобальними налаштуваннями інтерфейсу (зокрема, обраною темою оформлення) імплементовано глобальне сховище на базі бібліотеки Zustand (useThemeStore.js). Сховище автоматично синхронізується з localStorage браузера, що гарантує збереження користувацьких налаштувань між сесіями без необхідності виконання додаткових запитів до сервера [32].

Усі іконки в інтерфейсі інтегровані за допомогою бібліотеки lucide-react, яка рендерить SVG-елементи безпосередньо в DOM, мінімізуючи кількість HTTP-запитів до статичних ресурсів і покращуючи загальну продуктивність завантаження (First Contentful Paint).

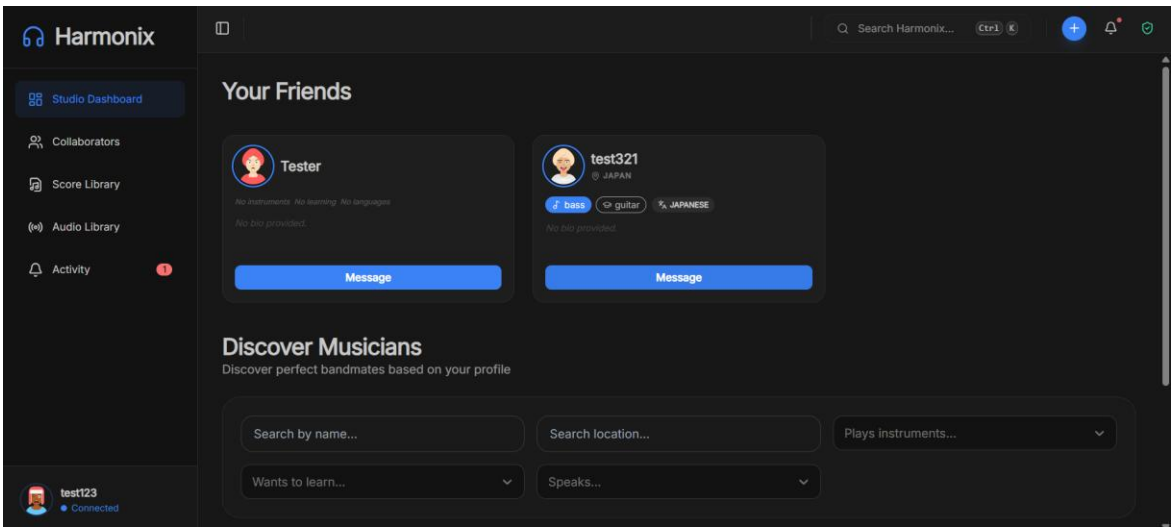


Рис. 3.1 Інтерфейс користувача цифрової платформи на головній сторінці (Темна тема)

3.4 Програмна реалізація комунікаційного інтерфейсу та глобального аудіоплеєра

Одним із найскладніших етапів програмної реалізації стала розробка модуля захищеного чату та інтеграція його з глобальним аудіоплеєром без конфліктів стану.

Реалізація інтерфейсу захищеного чату Відображення повідомлень інкапсульовано в компонент MessageBubble.jsx. Оскільки платформа використовує наскрізне шифрування, текст повідомлень надходить із сервера у зашифрованому вигляді. Дешифрування відбувається асинхронно на стороні клієнта за допомогою виклику функції decryptMessage. Для індикації процесу обробки користувачеві демонструється стан завантаження (isDecrypting). У випадку компрометації сесійного ключа (наприклад, при зміні ключів співрозмовником) компонент перехоплює помилку та виводить системне попередження [SECURITY_LOCKOUT: Session Integrity Compromised], блокуючи можливості редагування або копіювання такого повідомлення [15, 16].

Компонент підтримує багатий функціонал взаємодії: пересилання, закріплення, відповіді (Replies) та видалення. Для керування цими діями розроблено кастомне контекстне меню. Програмна логіка меню містить алгоритм

динамічного обчислення координат (e.clientX, e.clientY) відносно розмірів вікна браузера (window.innerWidth, window.innerHeight), що запобігає виходу меню за межі екрана на мобільних пристроях. Для закриття меню імплементовано глобальні слухачі подій (mousedown, scroll, wheel), які автоматично очищуються при розмонтуванні компонента (Cleanup function), що запобігає витокам пам'яті (Memory Leaks).

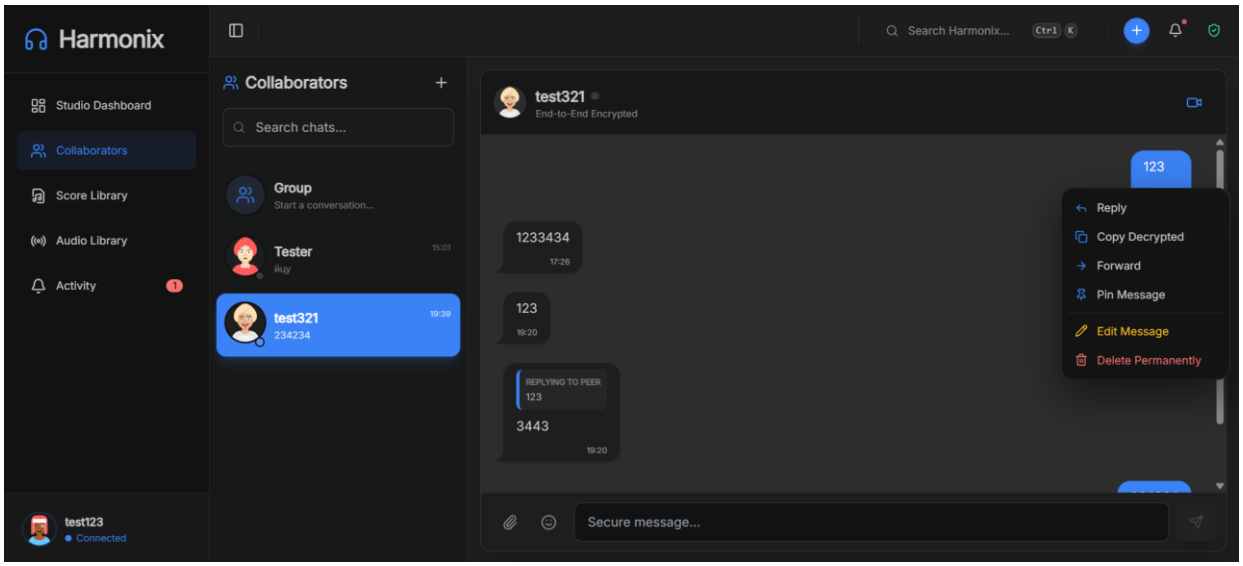


Рис. 3.2 Інтерфейс захищеного чату із кастомним контекстним меню повідомлення

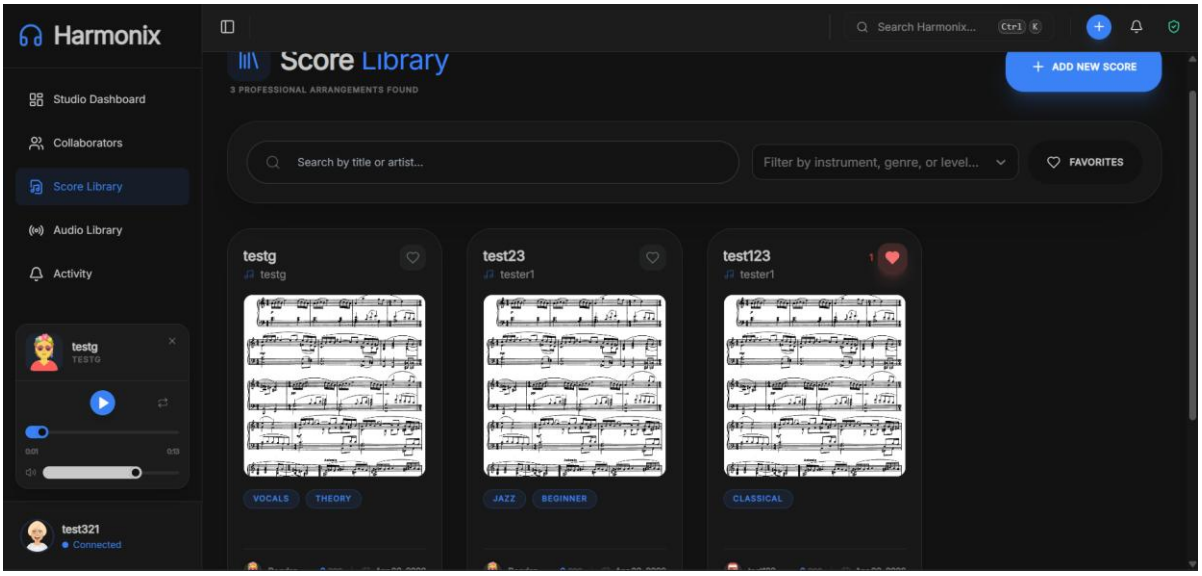


Рис. 3.3 Сторінка партирут з увімкненою мелодією що відображається у глобальному плеєрі

Реалізація візуальної частини аудіоплеєра Глобальний плеєр (GlobalMusicPlayer.jsx) реалізовано як фіксований віджет (Overlay), що дозволяє користувачам безперервно прослуховувати композиції під час навігації сторінками застосунку. Даний компонент є «проксі-інтерфейсом», який підписаний на глобальний стан відтворення (через useAudioStore) і керує логічним компонентом <audio>.

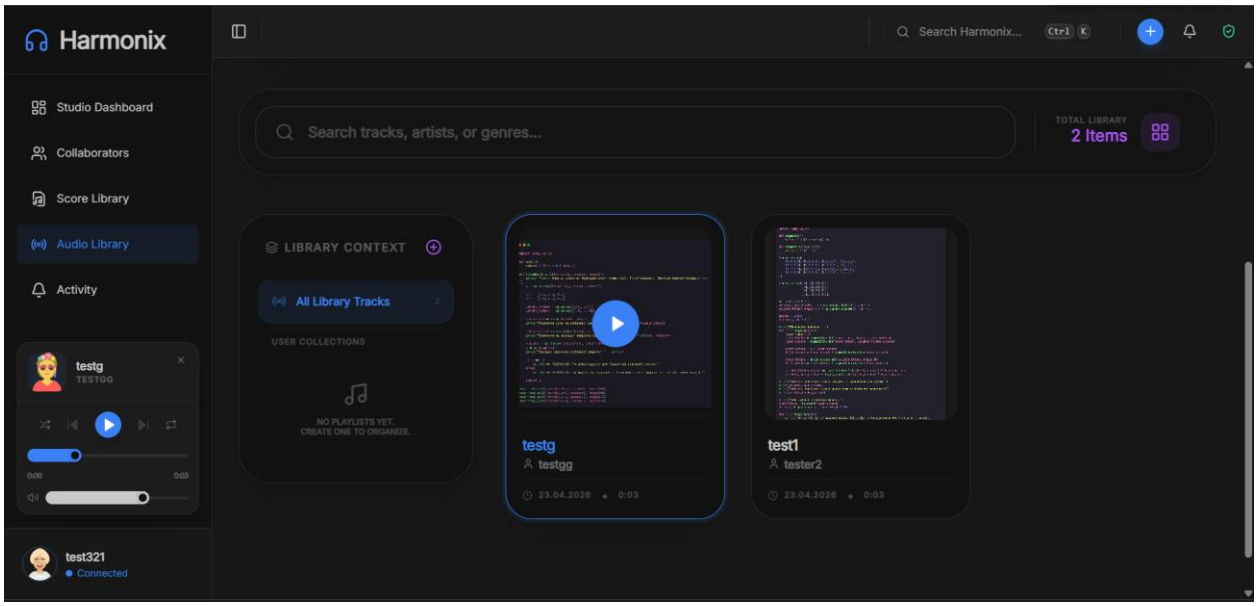


Рис. 3.4 Сторінка треків з увімкненою мелодією, що відображається у глобальному плеєрі

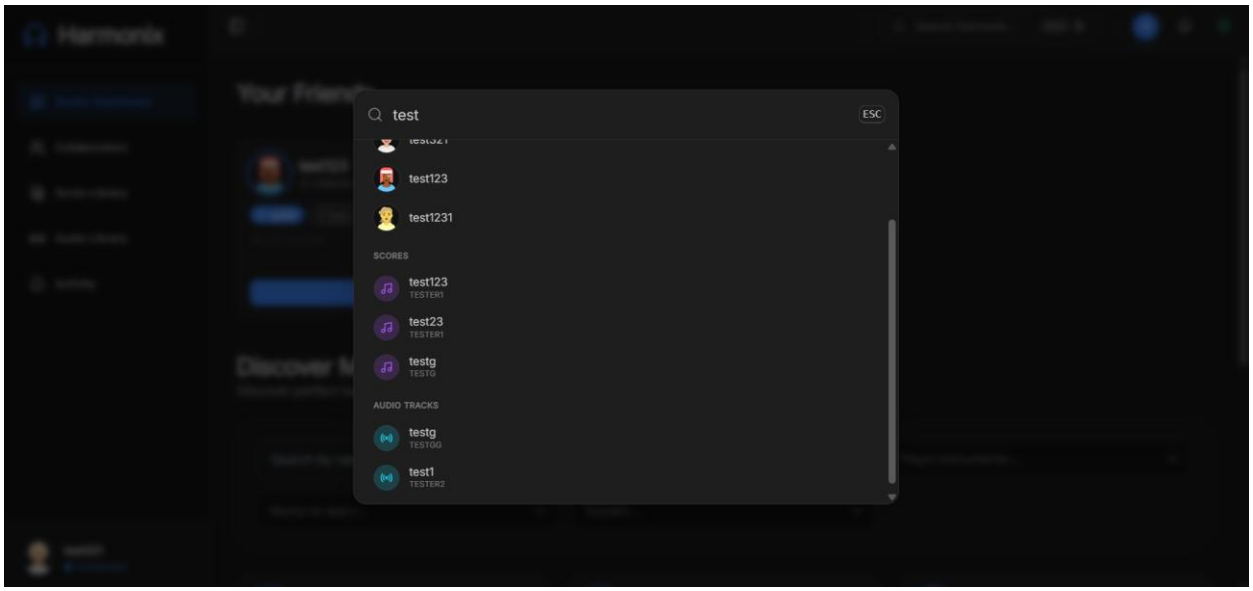


Рис. 3.5 Інтерфейс захищеного чату із кастомним контекстним меню повідомлення

Інтерфейс плеєра включає елементи управління потоком (Play/Pause, Skip, Loop, Shuffle) та інтерактивні повзунки (<input type="range">) для перемотування треку і регулювання гучності. Визначною архітектурною особливістю компонента є реалізація динамічного еквалайзера, який активується під час відтворення треку. Анімація створена виключно засобами CSS (@keyframes music-bar), що працюють на рівні графічного процесора (GPU) браузера і не навантажують основний потік JavaScript розрахунками висоти стовпчиків у реальному часі.

Разом із механізмом глобального пошуку (Command Palette), який інстанціюється через React Portals для уникнення проблем із z-індексом, розроблений інтерфейс утворює цілісну екосистему, що за рівнем інтерактивності відповідає сучасним індустріальним стандартам стрімінгових платформ.

3.5 Реалізація безпеки: алгоритми E2EE та ізоляція криптографічних ключів

Безпека та приватність користувачів платформи забезпечується завдяки комплексній імплементації наскрізного шифрування (End-to-End Encryption). З огляду на необхідність роботи застосунку виключно на стороні клієнта (у браузері) без передачі відкритих ключів на сервер, програмна реалізація модуля безпеки (crypto.js) базується на нативному Web Crypto API.

Під час реєстрації (або налаштування сесії) для кожного користувача локально генерується пара асиметричних ключів RSA-OAEP (2048-bit). Приватний ключ ніколи не покидає межі клієнтського пристрою: він серіалізується у формат JWK (JSON Web Key) та зберігається в ізольованому локальному сховищі браузера за допомогою IndexedDB (база даних HarmonixCryptoDB).

Ключовим досягненням реалізації є механізм «Vault Key». ОскількиIndexedDB є вразливою до XSS-атак (якщо зловмисник отримає фізичний доступ до розблокованого комп'ютера), приватний ключ додатково «огортається» (Wrapping) перед збереженням. Функція deriveVaultKey використовує пароль користувача та криптографічну сіль (Salt), пропускаючи їх

через 100 000 ітерацій алгоритму PBKDF2 для деривації симетричного ключа AES-GCM (256-bit), яким згодом і шифрується RSA-ключ (wrapPrivateKey).

Оскільки сервер не володіє паролем у відкритому вигляді, платформа автоматично генерує 16-байтний Recovery Key (generateRecoveryKey), який складається із символів кодування Base32 і є єдиним засобом розблокування криптосховища у разі втрати основного пароля.

Для забезпечення мінімальної затримки під час обміну даними у чатах (як особистих, так і групових) застосовано гібридну криптосистему. Асиметричні RSA-ключі використовуються лише один раз на початку сесії для безпечної передачі згенерованого симетричного сесійного ключа AES-GCM (generateSymmetricKey). Сам текст повідомлення шифрується алгоритмом AES-GCM із використанням унікального 12-байтного вектора ініціалізації (IV) для кожного повідомлення (encryptMessage). У результаті формується безпечний пакет даних формату Base64(IV):Base64(Ciphertext), який передається через Socket.io на сервер для ретрансляції.

3.6 Реалізація підсистеми відеоконференцій (WebRTC) та запису екрана

Модуль відеоконференцій розроблений із фокусом на багатокористувацьку (до 6 учасників) взаємодію у браузерному середовищі. Інтерфейс дзвінка (VideoCallOverlay.jsx) реалізує гібридну P2P Mesh-архітектуру.

При ініціалізації дзвінка застосунок захоплює локальні медіапотоки користувача (navigator.mediaDevices.getUserMedia). Для кожного нового учасника в кімнаті створюється окремий екземпляр RTCPeerConnection. Сигналізація (обмін SDP-пакетами та ICE-кандидатами) виконується через захищені канали Socket.io. Для проходження через NAT-брандмауери клієнтів у конфігурації iceServers визначено публічні STUN-сервери від Google.

Інтерфейс відеотрансляції (компоненти LocalVideo та RemoteVideo) забезпечує гнучке керування простором користувача: підтримку режимів масштабування (Fit/Fill), виведення відео у плаваюче вікно через нативний

Picture-in-Picture API та локальне глушіння звуку конкретного учасника (Local Mute) без впливу на глобальний потік [7, 8, 9].

Рис. 3.6 Інтерфейс відеоконференції з відображенням індивідуального меню керування учасником (Local Mute, PiP)

Особливістю платформи є інтегрована система запису сесій (Recording) безпосередньо у браузері, що виключає необхідність використання сторонніх програм типу OBS Studio. Процес запису реалізовано за допомогою об'єкта MediaRecorder. Функція захоплює системний аудіопотік вкладки (displaySurface: "browser") та поєднує його з локальним мікрофоном користувача через AudioContext у єдиний композитний потік. Запис ведеться у форматі video/webm із кодеком vp9.

Для оптимізації роботи з пам'яттю записані дані буферизуються масивом chunks. По завершенню запису система автоматично генерує Blob-об'єкт. Якщо розмір відео не перевищує 2 ГБ, застосунок самостійно завантажує файл на хмарне сховище платформи та автоматично відправляє зашифроване системне повідомлення (Collaboration session recorded.) у відповідний чат. В іншому випадку файл пропонується зберегти на локальний диск користувача через генерацію тимчасового URL (URL.createObjectURL).

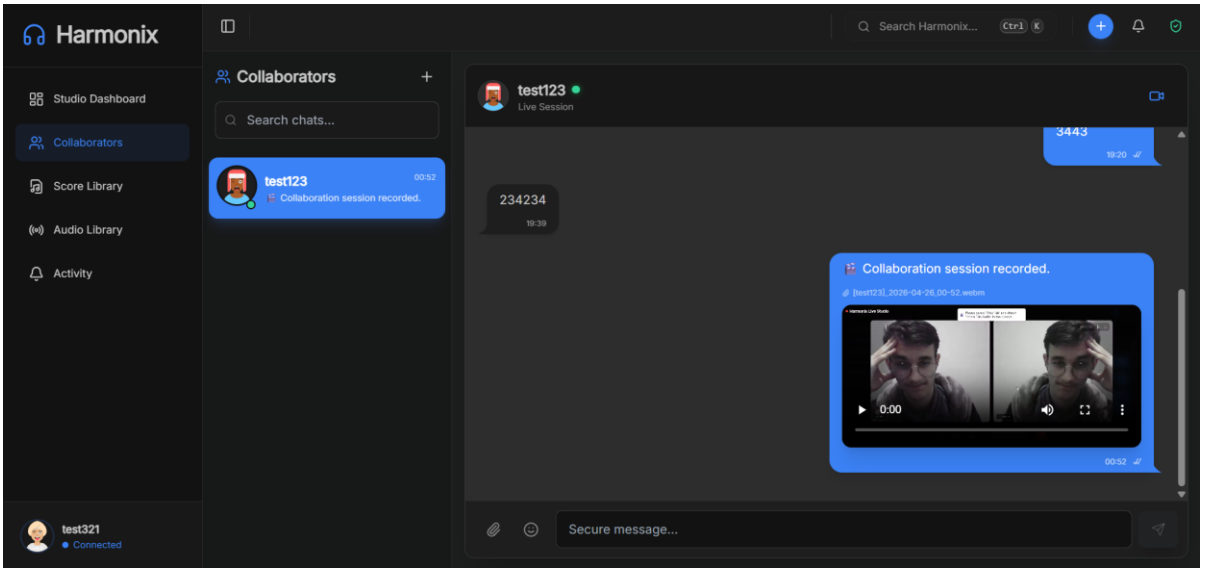


Рис. 3.7 Системне повідомлення у чаті про успішне збереження записаної відеосес

4 РЕАЛІЗАЦІЯ ІНТЕРФЕЙСУ ТА ТЕСТУВАННЯ СИСТЕМИ

4.1 Стратегія та методи тестування програмного забезпечення

Процес забезпечення якості (Quality Assurance) розробленої цифрової платформи базується на комплексній стратегії тестування, яка враховує архітектурні особливості системи, а саме: мікросервісну взаємодію контейнерів, використання протоколів реального часу та розгортання на апаратному забезпеченні з обмеженими обчислювальними ресурсами (IoT-вузол).

Для перевірки коректності роботи програмного продукту було застосовано комбінацію методів тестування «чорного ящика» (перевірка функціональності через інтерфейс без втручання у внутрішній код) та «білого ящика» (перевірка внутрішньої логіки та безпеки API). Стратегія включає наступні рівні:

1. Інтеграційне тестування: перевірка коректності взаємодії між клієнтською частиною, сервером (API) та базою даних.
2. Тестування безпеки: перевірка стійкості системи до типових вразливостей, перебору паролів (Brute-force) та спам-реєстрацій.
3. Навантажувальне тестування та моніторинг ресурсів: оцінка споживання оперативної пам'яті та процесорного часу мікрокомп'ютера в умовах контейнеризації.

Основним інструментарієм для проведення випробувань слугували середовище Postman (для надсилання HTTP-запитів до API), вбудовані інструменти розробника у веб-оглядачі Chrome DevTools, а також нативні утиліти операційної системи Linux (htop, docker stats) для профілювання апаратного забезпечення.

4.2 Інтеграційне тестування серверної бізнес-логіки та механізмів безпеки

Оскільки платформа оперує конфіденційними даними користувачів (зокрема, криптографічними ключами E2EE), критичним етапом стала перевірка захисних механізмів підсистеми автентифікації. Основний фокус під час тестування серверної бізнес-логіки було зосереджено на перевірці роботи проміжного програмного забезпечення для обмеження частоти запитів (Rate Limiting), яке налаштоване на рівні Nginx та Node.js.

Для перевірки механізму захисту від атак типу Brute-force (підбір пароля) було створено тестовий сценарій у середовищі Postman. Сценарій передбачав надсилання серії POST-запитів на маршрут http://localhost:5001/api/auth/login із валідною електронною адресою, але свідомо неправильним паролем. Тіло запиту (у форматі JSON) мало наступний вигляд:

{"email": "test@example.com", "password": "wrongpassword123"}

Згідно з визначеними раніше нефункціональними вимогами (розділ 1.6), система має блокувати IP-адресу після 5 невдалих спроб входу. Результати тестування підтвердили коректність закладеної логіки: перші п'ять запитів повертали стандартну помилку автентифікації (HTTP 401 Unauthorized), проте на шостому запиті сервер розірвав ланцюжок автентифікації та повернув статус-код 429 Too Many Requests. У тілі відповіді було отримано відповідне системне повідомлення, що свідчить про успішне блокування доступу на 15 хвилин:

{"success": false, "message": "Too many login attempts from this IP, please try again after 15 minutes."}

Наступним кроком стала перевірка захисту від масового автоматизованого створення спам-акаунтів, що є типовою проблемою для соціальних мереж. Тестування маршруту реєстрації http://localhost:5001/api/auth/signup відбувалося за аналогічним принципом. З однієї локальної IP-адреси було надіслано послідовні запити на створення нових профілів:

{"name":"name","email":"test@example.com","password": "wrongpassword123"}

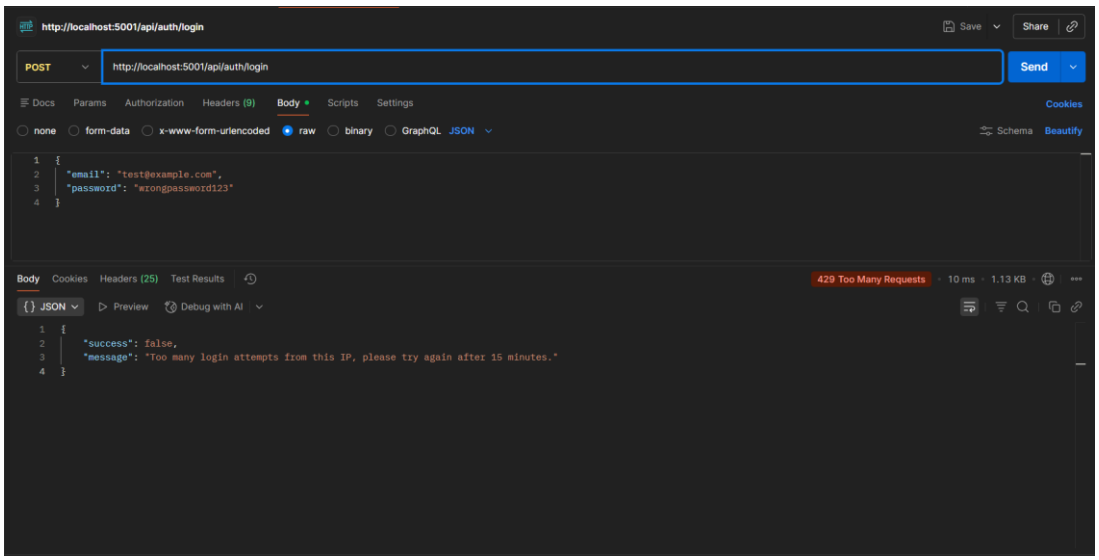


Рис. 4.1 Результат тестування лімітування запитів авторизації (помилка 429) у середовищі Postman

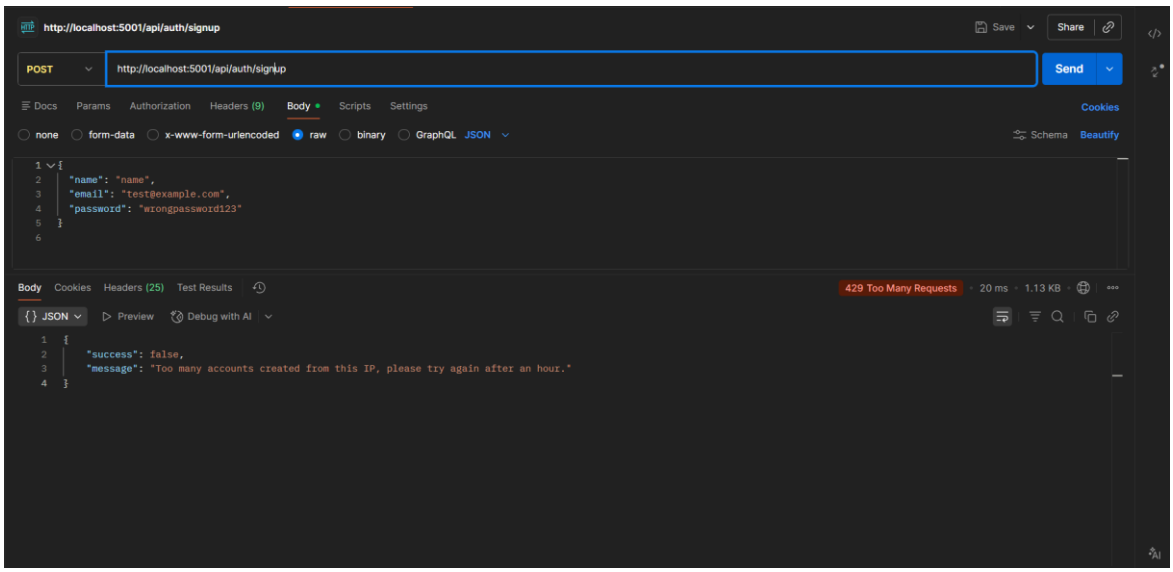


Рис. 4.2 Демонстрація блокування масової реєстрації акаунтів (захист від ботів)

Система успішно обробила перші два запити, створивши відповідні записи у базі даних PostgreSQL. Під час спроби зареєструвати третій акаунт серверний контролер ідентифікував перевищення ліміту для поточної IP-адреси. Відповідь сервера містила статус 429 Too Many Requests та інформаційне повідомлення про блокування створення облікових записів терміном на одну годину:

{ "success": false, "message": "Too many accounts created from this IP, please try again after an hour." }

Проведене інтеграційне тестування доводить, що серверне API є стійким до базових векторів атак. Імплементовані механізми Rate Limiting ефективно знижують ризик перевантаження бази даних сміттєвими записами та унеможливлюють автоматизований злам облікових записів легітимних користувачів.

4.3 Тестування інтерфейсу користувача та кросбраузерна сумісність

Перевірка клієнтської частини платформи здійснювалася з метою підтвердження коректності відображення компонентів та стабільності роботи глобального менеджера стану (Zustand) у різних середовищах. Тестування кросбраузерної сумісності проводилося у веб-оглядачах на базі рушіїв Chromium (Google Chrome, Microsoft Edge), Gecko (Mozilla Firefox) та WebKit (Apple Safari). Результати підтвердили ідентичність візуального відображення та безперебійну роботу Web Worker-ів для рендерингу PDF-документів у всіх зазначених середовищах.

Під час тестування адаптивності інтерфейсу (Responsive Web Design) було виявлено архітектурні обмеження поточної реалізації. Враховуючи високу щільність інформації на ключових екранах (багатокористувацькі відеоконференції з функцією Picture-in-Picture, відображення повноформатних нотних партитур та комплексне меню чату), поточна версія інтерфейсу оптимізована виключно для десктопних дисплеїв. При тестуванні на екранах мобільних пристроїв (ширина менше 768 пікселів) зафіксовано порушення сітки Tailwind CSS та накладання елементів управління (UI/UX overlap). У зв'язку з цим, адаптація платформи для мобільних пристроїв класифікується як технічний борг і виноситься на етап майбутніх ітерацій розробки (Future Work).

4.4 Тестування підсистем реального часу: WebSockets та WebRTC з'єднань

Функціонування комунікаційних модулів перевірялося шляхом симуляції одночасних клієнтських сесій. Тестування WebSocket-з'єднань підтвердило стабільність двостороннього обміну даними: час доставки зашифрованого текстового повідомлення від відправника до отримувача, включаючи етап дешифрування в браузері, не перевищував 100 мілісекунд.

Особливу увагу було приділено тестуванню алгоритму встановлення P2P-з'єднань через WebRTC. Перевірка логіки сигнального сервера (webrtc-offer, webrtc-answer) продемонструвала коректний обмін SDP-пакетами. Тестування в умовах знаходження клієнтів за різними NAT-маршрутизаторами підтвердило ефективність використання публічних STUN-серверів для отримання ICE-кандидатів. Процес трансляції екрана (Screen Sharing) та багатоканального запису сесії відбувався без розсинхронізації аудіо- та відеопотоків, що підтверджує надійність реалізованого підходу з використанням AudioContext та MediaRecorder.

4.5 Навантажувальне тестування та моніторинг апаратних ресурсів IoT-сервера

Головною нефункціональною вимогою до системи була здатність стабільно функціонувати на базі мікрокомп'ютера Raspberry Pi. Для оцінки споживання системних ресурсів використовувалися нативні утиліти операційної системи Linux (htop) та підсистеми контейнеризації (docker stats).

Під час імітації стандартного навантаження (активні WebSocket-з'єднання, трансляція WebRTC сигналізації та виконання запитів до бази даних) було зафіксовано наступні показники використання оперативної пам'яті (RAM) ізольованими контейнерами:

- harmonix-nginx (зворотний проксі): 4.93 MiB, що становить 1.93% від виділеного ліміту у 256 MiB;
- harmonix-backend (Node.js API та Socket.io): 62.18 MiB, що становить

3.04% від виділеного ліміту у 2 GiB;

- harmonix-frontend (статичний файловий сервер): 21.67 MiB, що становить 2.12% від виділеного ліміту в 1 GiB.

Сукупне споживання пам'яті платформою склало менше 90 MiB. Аналіз процесорного часу (за даними утиліти htop) продемонстрував показник середнього завантаження системи (Load Average) на рівні 0.31 / 0.14 / 0.08, що вказує на відсутність черг до процесора (CPU Bottleneck). Процес ядра Docker (/usr/bin/docker) генерував найбільше фонове навантаження, проте воно залишалося в межах норми для ARM-архітектури.

Отримані метрики емпірично доводять високу інженерну ефективність обраного технологічного стеку. Відмова від важких фреймворків (наприклад, Java Spring Boot, процес якого паралельно споживав понад 500 MiB згідно з даними моніторингу хоста) на користь асинхронного середовища Node.js дозволила розгорнути повноцінну високопродуктивну платформу на недорогому периферійному пристрої, повністю задовольнивши вимоги технічного завдання.

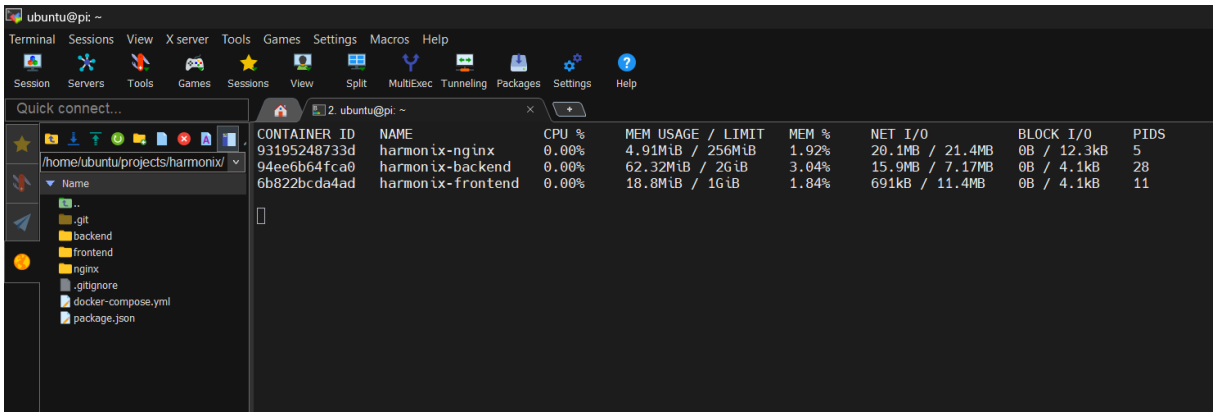


Рис. 4.3 Результати профілювання апаратних ресурсів контейнерів за допомогою утиліти docker stats

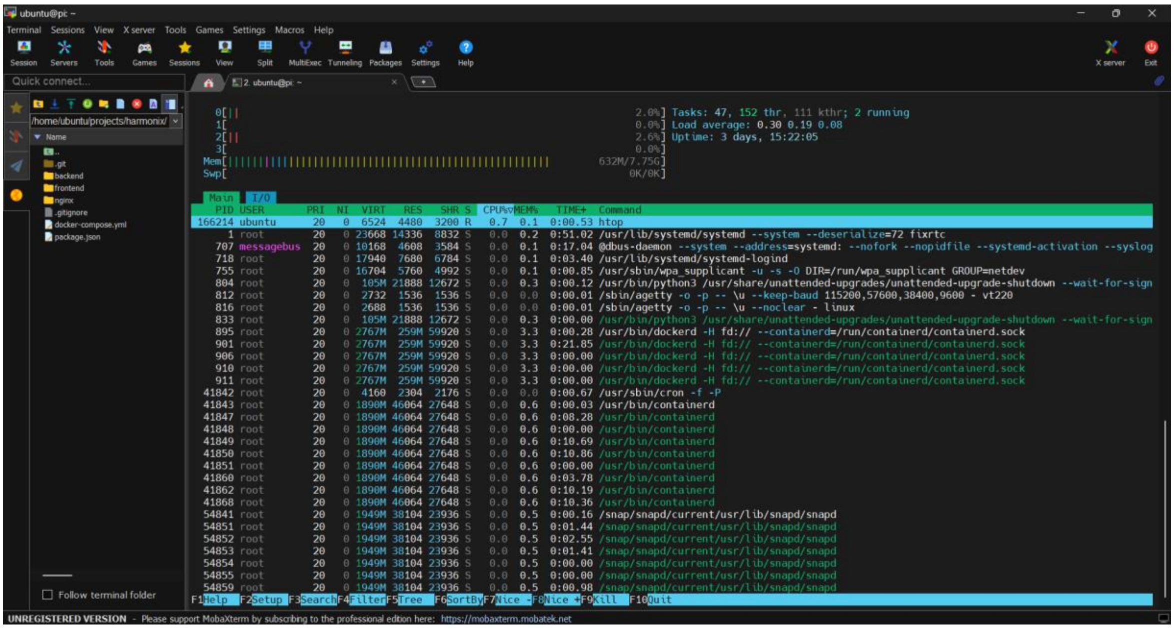


Рис. 4.4 Загальний моніторинг навантаження на систему Raspberry Pi через утиліту htop

4.6 Аналіз результатів та економічна (або практична) доцільність впровадження

Проведений комплексний аналіз та результати практичних випробувань підтверджують високу ефективність розробленої архітектури. З технічної точки зору, використання асинхронного середовища Node.js у поєднанні з механізмами оптимізації бази даних дозволило мінімізувати споживання оперативної пам'яті до рівня, що не перевищує 100 МБ для всього серверного ядра. Це повністю розв'язує проблему розгортання високонавантажених соціальних мультимедійних додатків на периферійних IoT-пристроях.

Економічна доцільність впровадження системи обґрунтовується порівнянням витрат на інфраструктуру. Традиційне розгортання подібної платформи (із використанням реляційної бази даних та сигнального сервера) у комерційних хмарах (наприклад, інстанс AWS t3.medium або аналог від Google Cloud) вимагає постійних операційних витрат (OPEX) на рівні 30–50 доларів США щомісяця. Натомість, обрана стратегія периферійних обчислень (Edge Computing) передбачає одноразові капітальні інвестиції (CAPEX) у придбання

мікрокомп'ютера Raspberry Pi (близько 80–100 доларів США). Окупність (ROI) такого апаратного рішення досягається вже на третій місяць експлуатації [20, 21]. Використання безкоштовного тарифного плану Cloudflare Tunnels для забезпечення Zero Trust маршрутизації та DDoS-захисту зводить подальші щомісячні витрати на підтримку інфраструктури до вартості спожитої пристроєм електроенергії (приблизно 5 Вт/год) [28].

Практична значущість розробленого програмного продукту полягає у забезпеченні абсолютної цифрової суверенності (Data Sovereignty). Оскільки база даних фізично знаходиться на стороні власника спільноти, а обмін повідомленнями захищений наскрізним шифруванням (E2EE), платформа повністю виключає можливість комерційного профілювання користувачів та витоку інтелектуальної власності музикантів, що є неможливим у традиційних SaaS-рішеннях.

ВИСНОВКИ

У рамках виконання кваліфікаційної роботи бакалавра розв'язано актуальне науково-практичне завдання – спроєктовано та програмно реалізовано захищену цифрову мультимедійну платформу для музичного ком'юніті, орієнтовану на децентралізоване розгортання в середовищі периферійних обчислень (IoT).

Для досягнення поставленої мети було отримано наступні результати:

Проведено системний аналіз предметної галузі та існуючих музичних платформ (Spotify, SoundCloud, MuseScore). Виявлено критичні недоліки щодо забезпечення конфіденційності користувачів, агресивної монетизації навчального контенту та відсутності спеціалізованих інструментів захищеної колаборації, що обґрунтувало необхідність створення альтернативної системи.

Розроблено та оптимізовано архітектуру серверної частини з використанням середовища Node.js, ORM Prisma та реляційної СКБД PostgreSQL. Програмне ядро адаптовано для функціонування в умовах обмежених ресурсів мікрокомп'ютера Raspberry Pi шляхом впровадження технологій контейнеризації (Docker) та багаторівневого кешування.

Забезпечено високий рівень безпеки даних та інфраструктури. Імплементовано алгоритми наскрізного шифрування (E2EE) за допомогою Web Crypto API, де генерація та зберігання криптографічних ключів відбувається у захищеному сховищі браузера (IndexedDB) із використанням Recovery Key. Фізичну безпеку сервера реалізовано через впровадження парадигми Zero Trust Network Access за допомогою технології Cloudflare Tunnels.

Програмно реалізовано інтерактивну клієнтську частину (SPA) на базі React, Zustand та Tailwind CSS. Інтерфейс інтегрує глобальний аудіоплеєр із механізмами безперервного фонового відтворення, підсистему відображення PDF-партитур через Web Workers та систему наскрізного пошуку (Command Palette).

Спроєктовано підсистему відеоконференцій на основі протоколу WebRTC. Додаток підтримує гібридну P2P-сигналізацію, трансляцію екрана та функцію створення композитних відеозаписів сесій безпосередньо у браузері з подальшим

безпечним завантаженням.

Проведено комплексне інтеграційне та навантажувальне тестування платформи. Отримані метрики підтвердили стабільність роботи системи, ефективність механізмів блокування атак (Rate Limiting) та економічну доцільність автономного самостійного хостингу для локальних музичних спільнот.

Запропонований програмний продукт повністю відповідає вимогам технічного завдання і є готовим до практичного використання незалежними колективами або освітніми осередками.

ПЕРЕЛІК ПОСИЛАНЬ

1. Dallos C., Konieczka G. Data, Privacy, and Personalization: How Spotify Maintains a Competitive Advantage under the GDPR. Uppsala University, 2025. 65 p. URL: https://www.diva-portal.org/smash/get/diva2:1981304/FULLTEXT01.pdf
2. Sandoval K. Case Study: SoundCloud's Multi-Year Journey Breaking the Monolith. Nordic APIs, 2022. URL: https://nordicapis.com/case-study-soundclouds-multi-year-journey-breaking-the-monolith/
3. Stuart T. Microservices and the monolith. SoundCloud Backstage Blog, 2016. URL: https://developers.soundcloud.com/blog/microservices-and-the-monolith
4. Monolithic vs Microservices Architecture: Pros and Cons. DICEUS, 2024. URL: https://diceus.com/microservices-vs-monolith/
5. Edge vs. Cloud: Empirical Insights into Data-Driven Condition Monitoring. MDPI, 2025. URL: https://www.mdpi.com/3302462
6. Guri M. Pico-Cloud: Cloud Infrastructure for Tiny Edge Devices. arXiv preprint arXiv:2511.13253, 2025. 12 p. URL: Cloud Infrastructure for Tiny Edge Devices. arXiv preprint arXiv:2511.13253, 2025. 12 p. URL: https://arxiv.org/html/2511.13253v1
7. Kasetwar A., Balani N., Damwani D. A WebRTC Based Video Conferencing System with Screen Sharing. International Journal for Research in Applied Science and Engineering Technology (IJRASET), 2022. URL: https://www.ijraset.com/research-paper/webrtc-based-video-conferencing-system-with-screen-sharing
8. Koren I., Klamma R. OakStreaming: A Peer-to-Peer Video Streaming Library. Journal of Web Engineering, Vol. 17(6&7), 2019. P. 527–560. URL: https://journals.riverpublishers.com/index.php/JWE/article/view/4147/4091
9. A systematic review on WebRTC for potential applications and challenges beyond audio/video streaming. ResearchGate, 2024. URL: https://www.researchgate.net/publication/386077344_A_systematic_review_on

_WebRTC_for_potential_applications_and_challenges_beyond_audio_video_streaming

10. Фещенко Б.В. Гібридна архітектура музичної платформи на базі периферійних обчислень та серверлес-технологій. // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.:ДУІКТ, 2026.
11. Сініцин І., Фещенко Б. Інтеграція IoT-інфраструктури та хмарних сервісів для побудови захищеної цифрової музичної платформи // Collection of Scientific Papers with the Proceedings of the 7th International Scientific and Practical Conference «Scientific Exploration: Bridging Theory and Practice» (May 25-27, 2026, Berlin, Germany). Berlin: European Open Science Space, 2026. P. 288–290. URL: https://www.eoss-conf.com/en/archive/scientific-exploration-bridging-theory-and-practice-25-05-26/.
12. Bahl V., Colangelo L., Dillon S. Platform Accountability in the Age of AI, Music Streaming, and Digital Media. Tech Policy @ Sanford, Duke University, 2025. 14 p. URL: https://techpolicy.sanford.duke.edu/wp-content/uploads/2025/04/Version-0.5-AI-Music-Streaming-Draft.pdf
13. Kim J., Klein-Balajee T., Kelly R. M. Discord's Design Encourages “Third Place” Social Media Experiences. arXiv preprint arXiv:2501.09951, 2025. 15 p. URL: https://arxiv.org/html/2501.09951v1
14. Feng K., McDonald D., Zhang A. Teachable Agents for End-User Empowerment in Personalized Feed Curation. Knight First Amendment Institute, 2023. 18 p. URL: https://knightcolumbia.org/content/teachable-agents-for-end-user-empowerment-in-personalized-feed-curatio
15. Perrin T., Marlinspike M. The Double Ratchet Algorithm. Signal Specifications, Revision 4, 2025. URL: https://signal.org/docs/specifications/doubleratchet/
16. Collins D., Riepel D., Tran S. A. O. On the Tight Security of the Double Ratchet. Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, 2024. P. 401–433. URL: https://www.research-

collection.ethz.ch/server/api/core/bitstreams/e00a372c-1f37-4e10-822d-5a19bf9efbd6/content

17. Connell G., Schmidt R. Signal Protocol and Post-Quantum Ratchets. Signal Blog, 2025. URL: https://signal.org/blog/pqxdh/
18. Diving into Signal's New Post-Quantum Protocol. PQShield Research, 2025. URL: https://pqshield.com/diving-into-signals-new-pq-protocol/
19. A Hybrid Approach for WebRTC Video Streaming on Resource-Constrained Devices. MDPI, 2025. URL: https://www.mdpi.com/2469008
20. Kuriakose A. Own Cloud Using Raspberry: Comparing OwnCloud Storage Using Raspberry Pi and Commercial Clouds. VAASAN AMMATTIKORKEAKOULU University of Applied Sciences, 2025. 65 p. URL: https://www.theseus.fi/bitstream/10024/895583/2/Kuriakose_Anna.pdf
21. Girish N., Krupananda S. Assessing the Viability and Dependability of Nextcloud Deployed on Raspberry Pi for Network-Attached Cloud Storage. IJCRT, 2023. URL: https://www.ijcrt.org/papers/IJCRT2405612.pdf
22. The Sovereignty Stack: What is Self-Hosting & Why It's the Future of Digital Freedom. PayRam Insights, 2025. URL: https://payram.com/blog/what-is-self-hosting
23. What is Data Sovereignty? Amazon Web Services (AWS) Documentation, 2025. URL: https://aws.amazon.com/what-is/data-sovereignty/
24. Cloud Data Sovereignty Governance and Risk Implications of Cross Border Cloud Storage. ISACA Industry News, 2024. URL: https://www.isaca.org/resources/news-and-trends/industry-news/2024/cloud-data-sovereignty-governance-and-risk-implications-of-cross-border-cloud-storage
25. Comparing the Big Three: A Comprehensive Analysis of Ngrok, Cloudflare Tunnel, and Tailscale. InstaTunnel Research, 2025. URL: https://instatunnel.wordpress.com/2025/09/02/comparing-the-big-three-a-comprehensive-analysis-of-ngrok-cloudflare-tunnel-and-tailscale-for-modern-development-teams

26. Shourie A. D. Secure Tunneling Explained: Ngrok vs Cloudflared. DEV Community, 2025. URL: https://dev.to/aryan_shourie/secure-tunneling-explained-ngrok-vs-cloudflared-mcl
27. What is tunneling? | Tunneling in networking. Cloudflare Learning Center, 2025. URL: https://www.cloudflare.com/learning/network-layer/what-is-tunneling/
28. Tunnel Vision: CloudflareD AbuseD in the WilD. GuidePoint Security Threat Intelligence, 2025. URL: https://www.guidepointsecurity.com/blog/tunnel-vision-cloudflared-abused-in-the-wild
29. Docker Documentation. Multi-stage builds. Docker Inc., 2024. URL: https://docs.docker.com/build/building/multi-stage/
30. Restoring original visitor IPs. Cloudflare Support Documentation, 2024. URL: https://developers.cloudflare.com/support/troubleshooting/restoring-visitor-ips/restoring-original-visitor-ips/
31. OWASP Secure Headers Project. Open Worldwide Application Security Project (OWASP Foundation), 2024. URL: https://owasp.org/www-project-secure-headers/
32. Zustand Documentation. State Management and Persist Middleware. React Ecosystem, 2024. URL: https://zustand.docs.pmnd.rs/reference/middlewares/persist

ДОДАТОК А. ДОДАТКОВІ МАТЕРІАЛИ

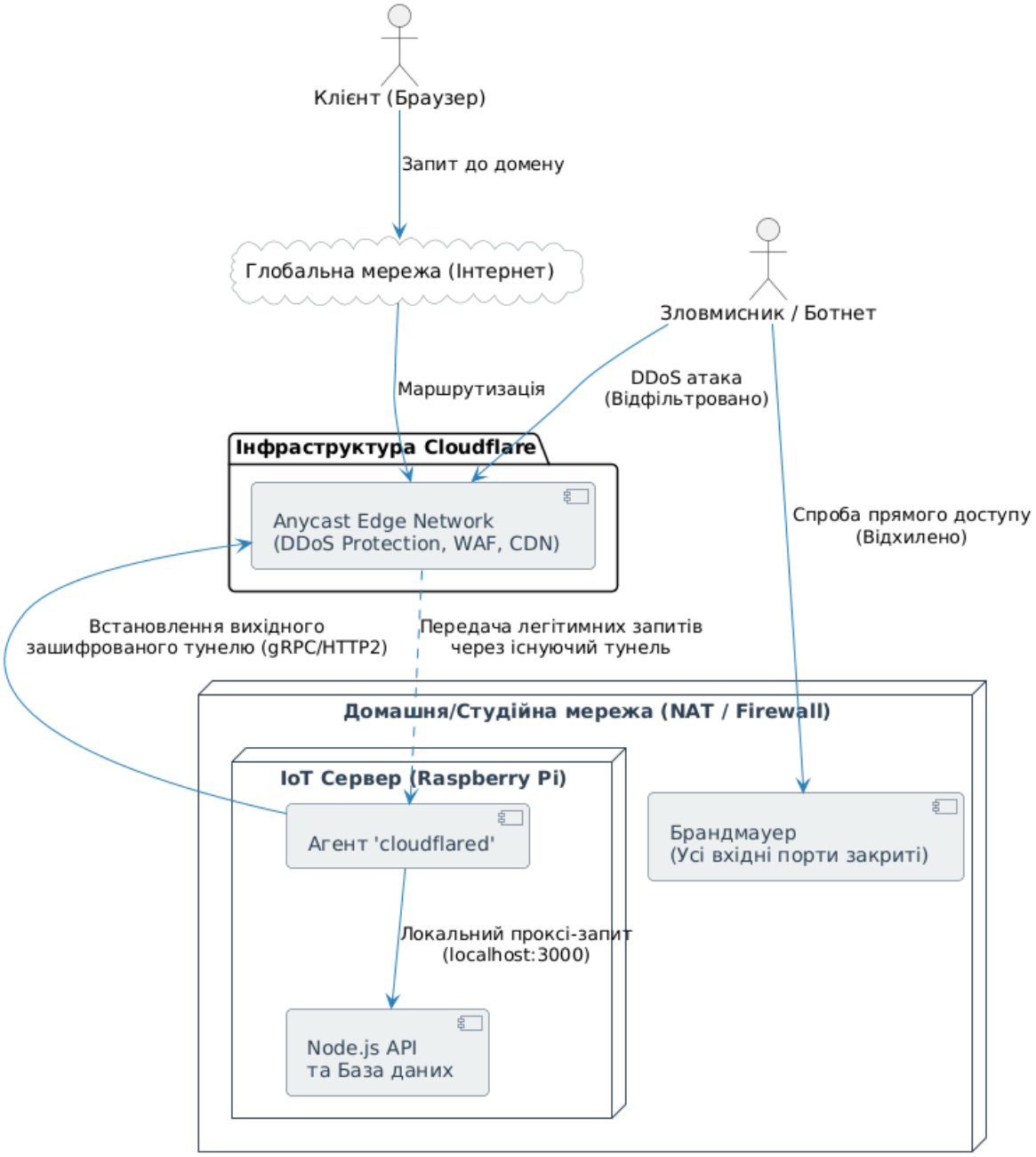


Рис. 1.8. Архітектура зворотного тунелювання та забезпечення ізоляції IoT-сервера за допомогою Cloudflare Tunnels

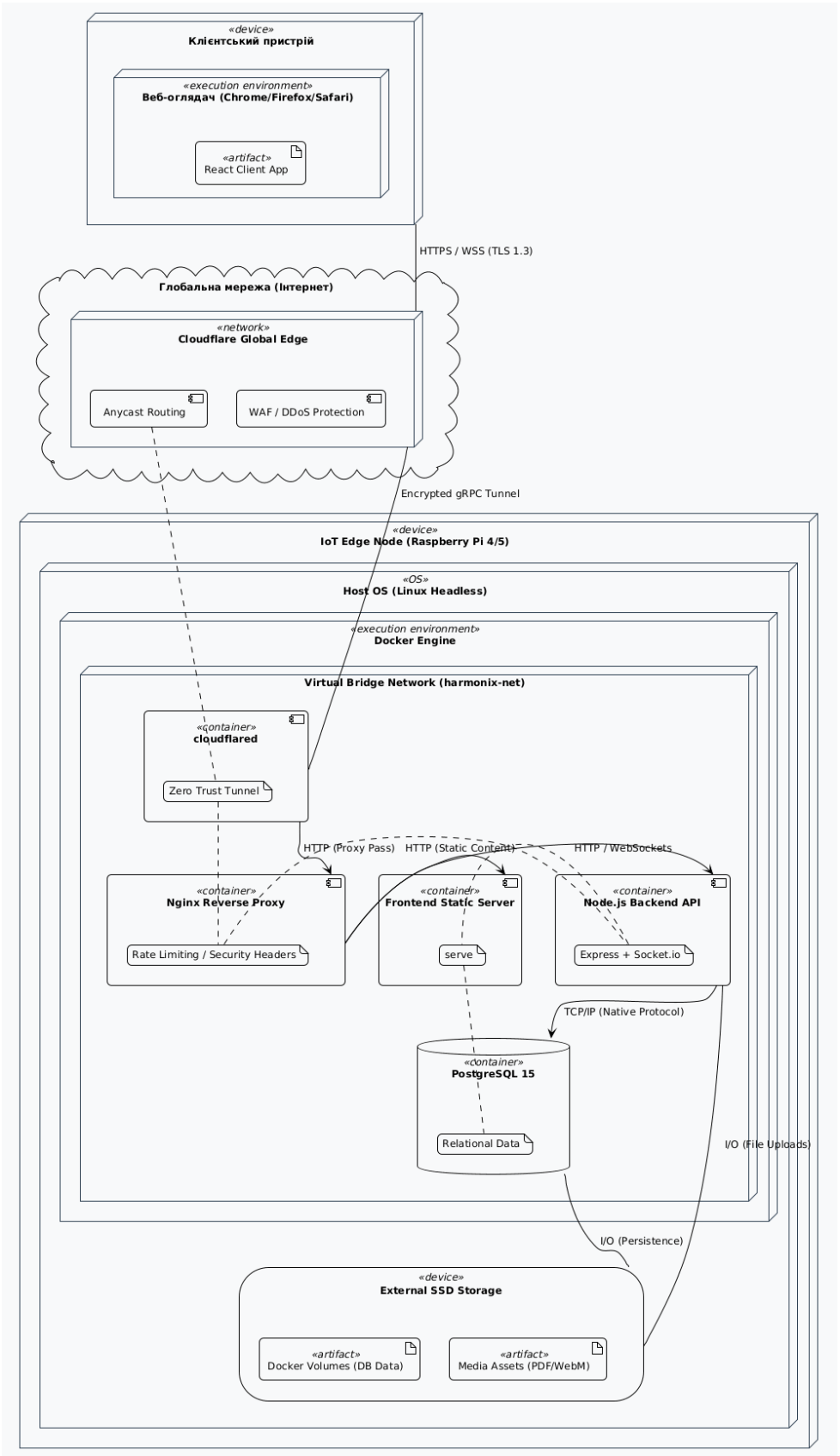


Рис. 2.2 UML-діаграма розгортання (Deployment Diagram) програмних компонентів на IoT-вузлі

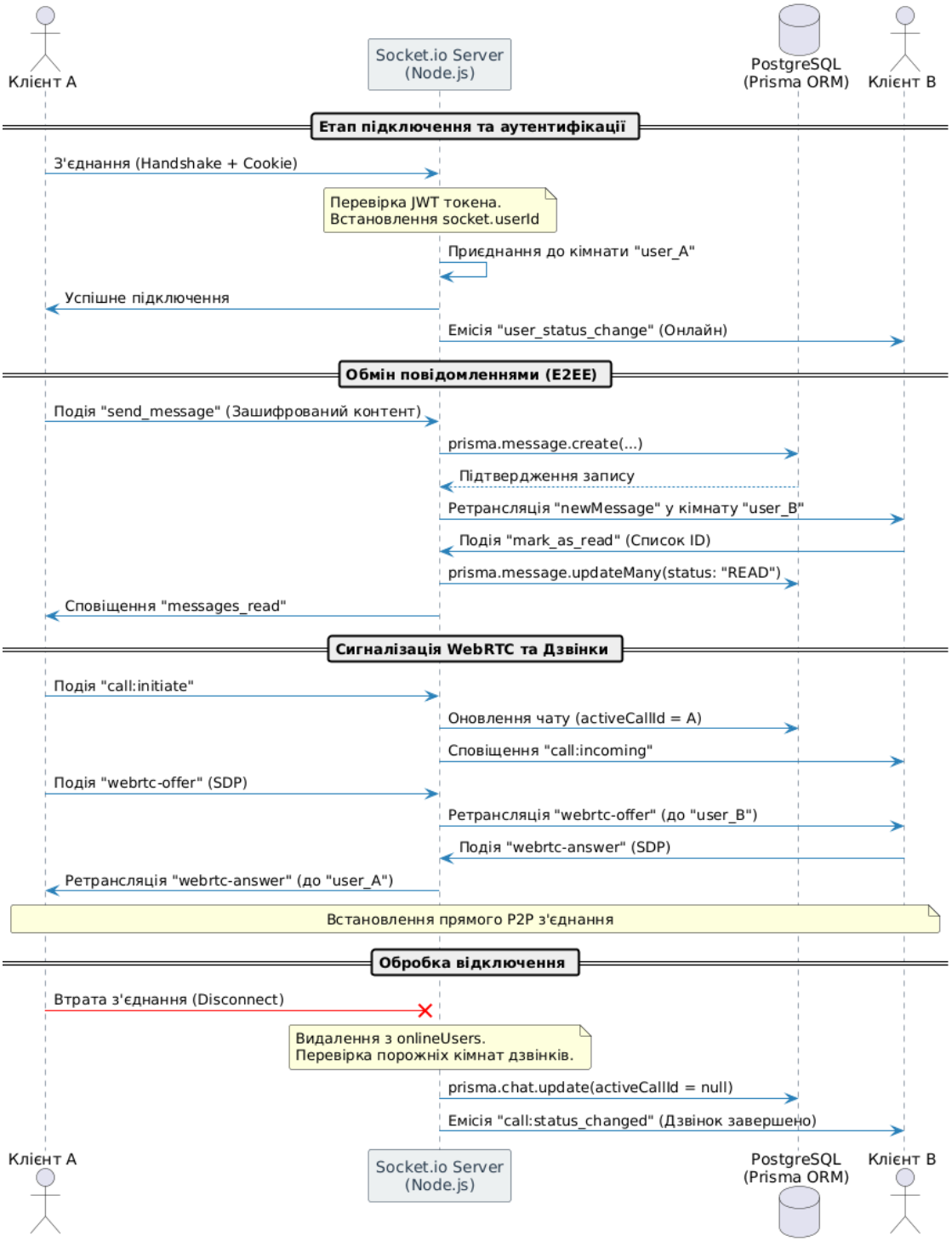


Рис. 2.4 UML-діаграма послідовності сигналізації WebRTC та обміну повідомленнями через Socket.io

ДОДАТОК Б. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра Технологій цифрового розвитку

Цифрова платформа музичного ком'юніті з функціоналом пошуку композицій, навчання та захищеної взаємодії користувачів із використанням IoT-інфраструктури

Виконав студент 4 курсу
Групи ТЦР-41
Фещенко Богдан Вікторович
Керівник роботи
Д-р техн. наук, професор, Сініцин Ігор Петрович

Київ-2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – підвищення ефективності взаємодії користувачів та рівня захищеності обміну даними шляхом розробки цифрової платформи музичного ком'юніті з функціоналом пошуку композицій, навчання та інтеграцією IoT-інфраструктури.

Об'єкт дослідження – процес організації безпечної соціальної взаємодії та обміну мультимедійним контентом у децентралізованих музичних спільнотах.

Предмет дослідження – програмне забезпечення, інфраструктурні та криптографічні рішення для функціонування мультимедійної платформи на базі IoT-пристрою (Raspberry Pi).

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз існуючих музичних сервісів та комунікаційних платформ, виявити їхні архітектурні недоліки.
2. Обґрунтувати вибір технологічного стеку та спроєктувати архітектуру платформи для роботи на ресурсообмеженому IoT-вузлі.
3. Розробити підсистему захищеної взаємодії: реалізувати алгоритми наскрізного шифрування (E2EE) та WebRTC-відеозв'язок із можливістю запису екрана.
4. Спроєктувати та сформувати реляційну базу даних для збереження інформації про користувачів, їхні композиції та криптографічні ключі.
5. Реалізувати клієнтську частину (SPA) з інтеграцією глобального аудіоплеєра, функціоналу перегляду PDF-партитур та наскрізного пошуку.
6. Провести тестування безпеки (Rate Limiting) та навантажувальне тестування апаратних ресурсів (RAM/CPU) мікрокомп'ютера при роботі в контейнерах.

АНАЛІЗ АНАЛОГІВ

Критерій порівняння	Spotify	SoundCloud	MuseScore	Discord	Розроблена платформа
Спеціалізація на музичному навчанні	Відсутня	Відсутня	Висока	Відсутня	**Висока**
Бібліотека партитур (PDF/MusicXML)	Відсутня	Відсутня	Повна	Відсутня	**Інтегрована**
Захищений P2P відеозв'язок	Відсутній	Відсутній	Відсутній	Частковий	**Нативний (WebRTC)**
Наскрізне шифрування (E2EE)	Відсутнє	Відсутнє	Відсутнє	Відсутнє	**Реалізовано**
Хостинг на IoT (Edge Computing)	Відсутній	Відсутній	Відсутній	Відсутній	**Підтримується**
Автономність від хмарних гігантів	Відсутня	Відсутня	Відсутня	Відсутня	**Повна (IoT вузол)**
Управління пропущеними (Real-time)	Базове	Базове	Відсутнє	Високе	**Реалізовано**

ВИМОГИ ДО ПЛАТФОРМИ

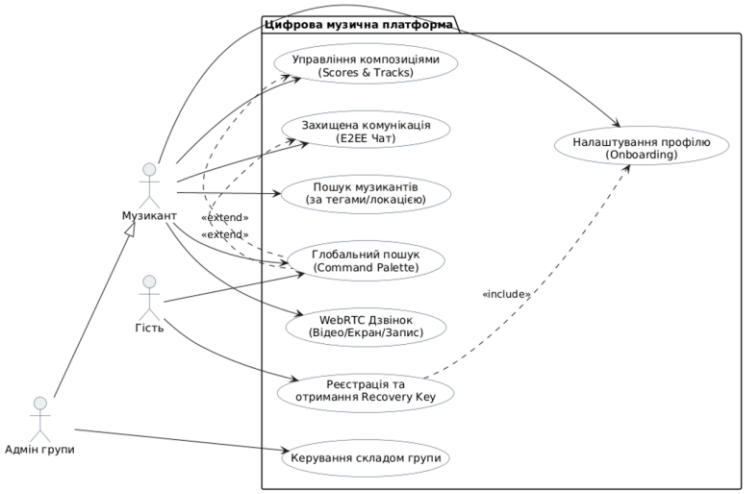
Функціональні вимоги

1. Реєстрація та авторизація користувачів із генерацією Recovery Key для відновлення доступу до акаунта.
2. Підтримка захищених особистих і групових чатів із використанням наскрізного шифрування (E2EE).Реалізація групових аудіо- та відеодзвінків на основі WebRTC із підтримкою трансляції екрана та запису сесій у форматі WebM.
3. Реалізація глобального фонового аудіоплеєра та бібліотеки PDF-нот для перегляду навчальних матеріалів.
4. Створення бібліотеки музичних треків із можливістю їх збереження та відтворення.
5. Реалізація наскрізного пошуку (Command Palette) по композиціях, чатах і користувачах.

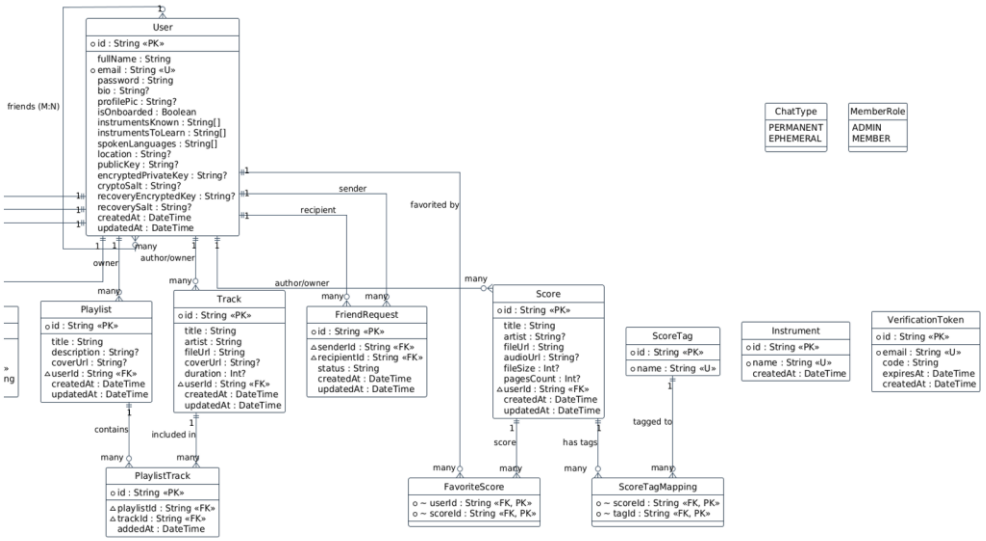
Нефункціональні вимоги

1. Серверні компоненти системи повинні бути платформонезалежними та підтримувати ізольований запуск у віртуалізованих середовищах. Це має забезпечувати ідентичність роботи програми на різних етапах розробки та спрощувати процес її розгортання на цільовому сервері.
2. Програмне забезпечення має бути оптимізоване для стабільної роботи на малопотужному периферійному пристрої з ARM-архітектурою. Сукупне споживання оперативної пам’яті не повинно перевищувати 200 МБ.
3. Доступ до платформи з глобальної мережі Інтернет повинен здійснюватися виключно через захищені канали зв'язку без необхідності відкриття вхідних портів на локальному маршрутизаторі.

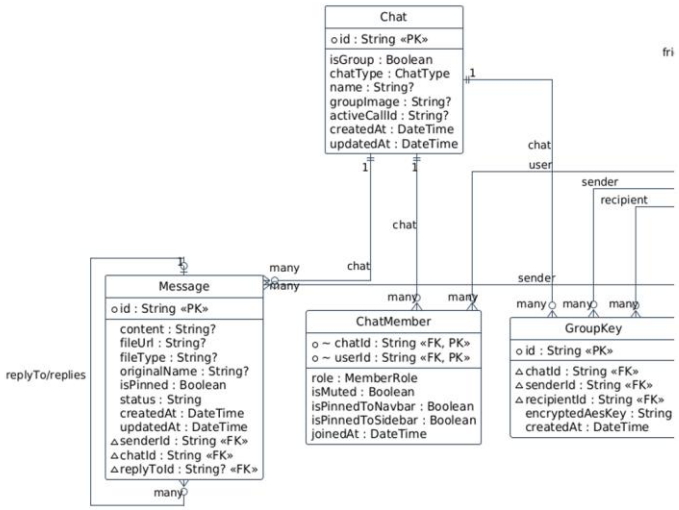
ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ ПЛАТФОРМИ



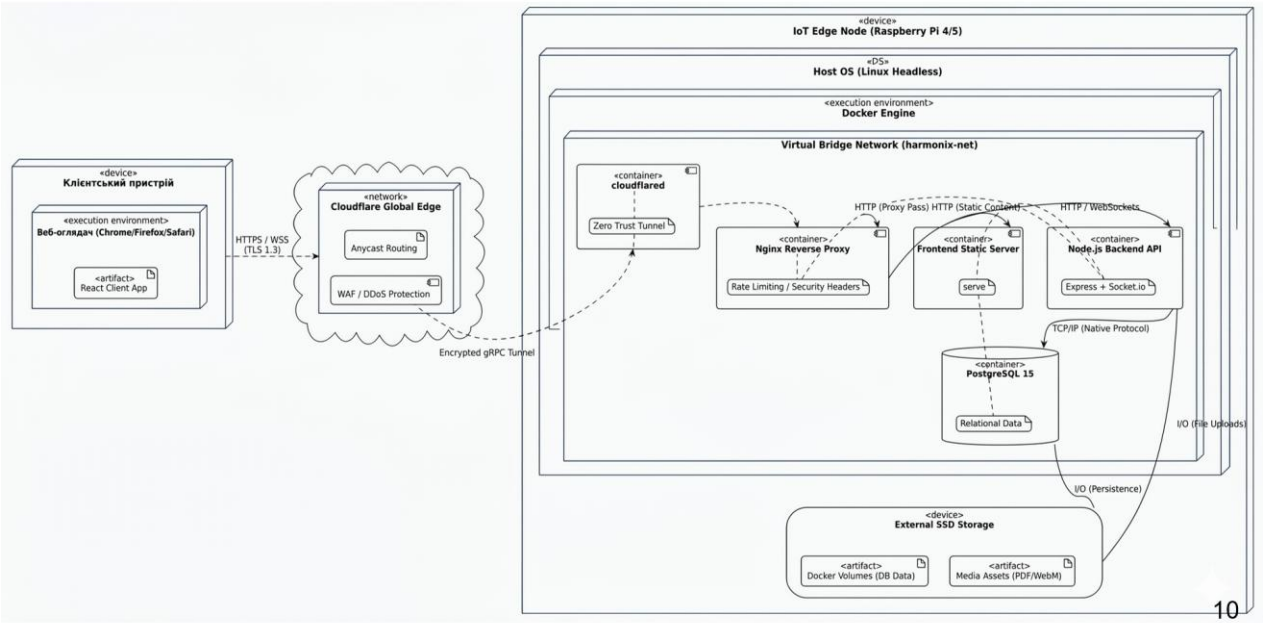
ДІАГРАМА СТРУКТУРИ БАЗИ ДАНИХ



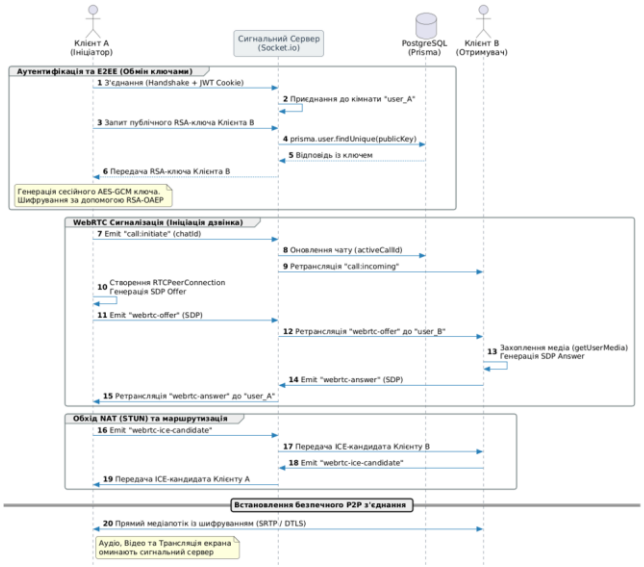
ДІАГРАМА СТРУКТУРИ БАЗИ ДАНИХ



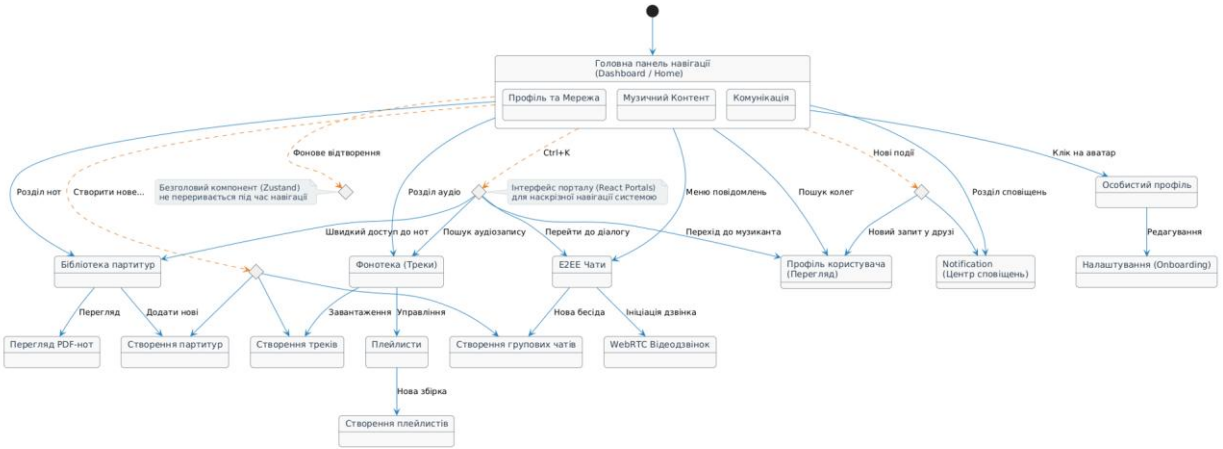
ДІАГРАМА РОЗГОРТАННЯ ІНФРАСТРУКТУРИ НА IOT-ВУЗЛІ



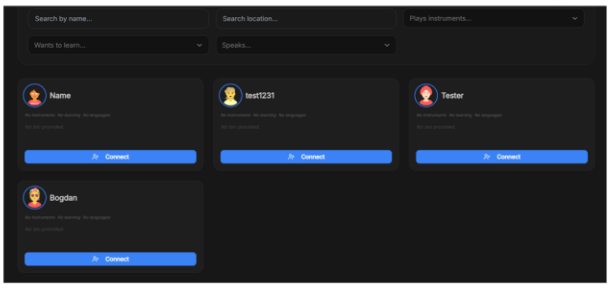
ДІАГРАМА ПОСЛІДОВНОСТІ WEBRTC ТА E2EE-СИГНАЛІЗАЦІЇ



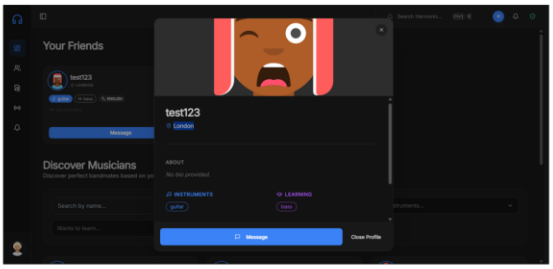
ДІАГРАМА НАВІГАЦІЇ ГОЛОВНОЇ СТОРІНКИ



ЕКРАННІ ФОРМИ

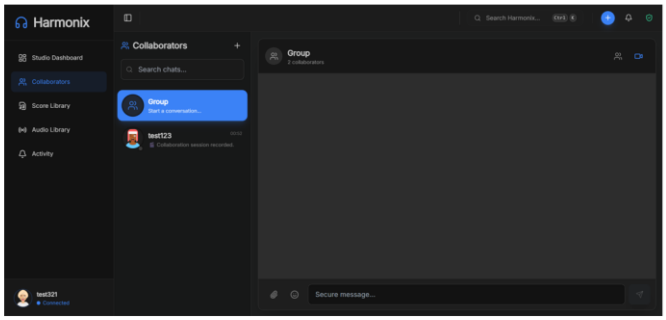


Інтерфейс головної пошуку музикантів головної сторінки

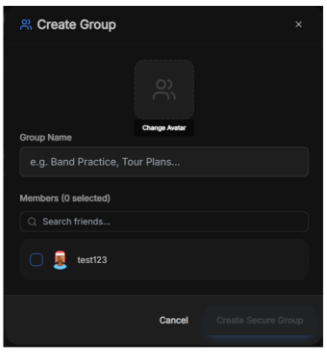


Модальне вікно профілю користувача

ЕКРАННІ ФОРМИ

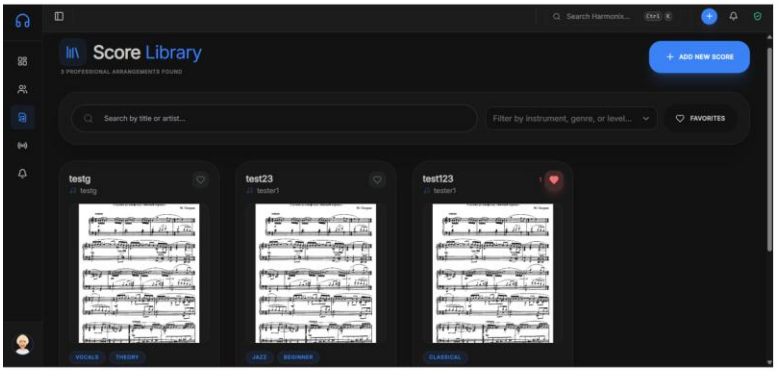


Сторінка списка чатів та чатом з правого боку

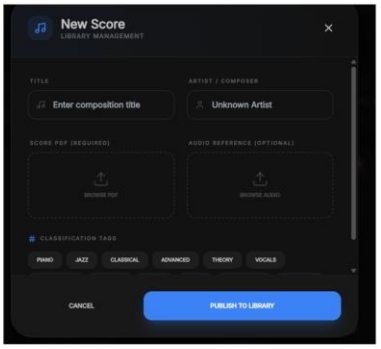


Модальне вікно створення групового чату

ЕКРАННІ ФОРМИ

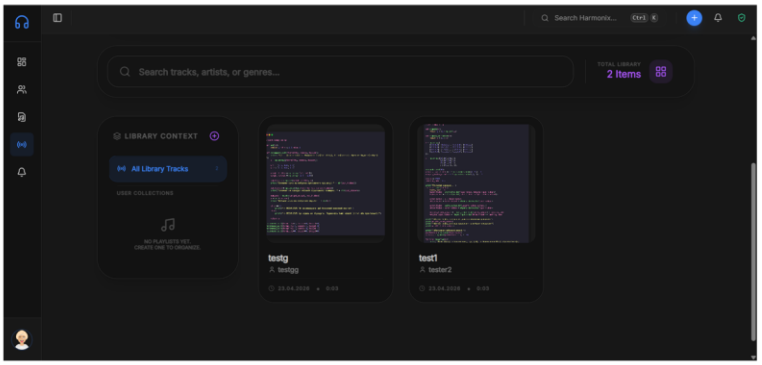


Сторінка партитур

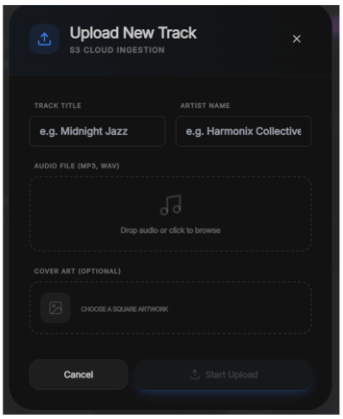


Модальне вікно створення партитури

ЕКРАННІ ФОРМИ

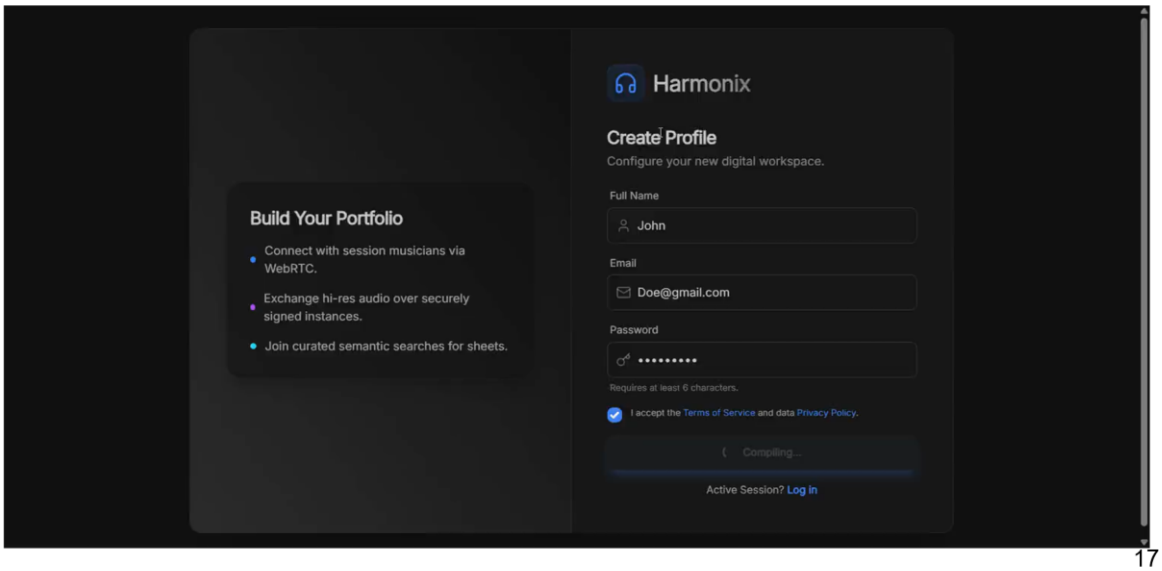


Сторінка музичних треків



Модальне вікно створення трека

Відео-демонстрація роботи додатку



АПРОБАЦІЯ

1. Фещенко Б.В. Гібридна архітектура музичної платформи на базі периферійних обчислень та серверлес-технологій // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.: ДУІКТ, 2026.

2. Фещенко Б.В. Гібридна архітектура інтелектуальної музичної платформи на базі IoT-інфраструктури та хмарних технологій // VI Всеукраїнська Науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез. К.: ДУІКТ, 2026.

ВИСНОВКИ

1. Проведено комплексний аналіз існуючих музичних платформ, що дозволило виявити критичні недоліки у сферах приватності та суверенітету даних і обґрунтувати створення автономної цифрової екосистеми.
2. Розроблено та програмно реалізовано архітектуру системи на базі периферійних обчислень (Raspberry Pi), що забезпечує незалежність від хмарних провайдерів та мінімізує операційні витрати на підтримку спільноти.
3. Впроваджено високі стандарти безпеки: імплементовано наскрізне шифрування (E2EE) особистої взаємодії на стороні клієнта та налаштовано захищене тунелювання трафіку за парадигмою Zero Trust.
4. Створено інтегроване робоче середовище, яке об’єднує інструменти захищеної комунікації (WebRTC), фонового прослуховування аудіоконтенту та інтерактивного перегляду нотних партитур у межах єдиного SPA-застосунку.
5. Підтверджено інженерну ефективність обраного технологічного стеку: результати тестування довели стабільність роботи платформи в умовах жорстких апаратних обмежень IoT-вузла при мінімальному споживанні ресурсів (RAM < 100 МБ).
6. Результати роботи мають практичну цінність для незалежних музичних колективів та освітніх організацій, забезпечуючи захищене середовище для творчої колаборації.

ДОДАТОК В. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
version: '3.8'

services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: harmonix-frontend
    restart: unless-stopped
    networks:
      - harmonix-net
    environment:
      - NODE_ENV=production
      - PORT=3000
    deploy:
      resources:
        limits:
          memory: 1G

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: harmonix-backend
    restart: unless-stopped
    networks:
      - harmonix-net
    env_file:
      - ./backend/.env
    environment:
      - NODE_ENV=production
      - PORT=3001
    deploy:
      resources:
        limits:
          memory: 2G
    depends_on:
      - db-init

  db-init:
    image: alpine
    command: [ "echo", "Database
connection established" ]
    networks:
      - harmonix-net

  nginx:
    image: nginx:alpine
    container_name: harmonix-nginx
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      -
./nginx/nginx.conf:/etc/nginx/nginx.conf:
ro
    depends_on:
      - frontend
      - backend
    networks:
      - harmonix-net
    deploy:
      resources:
        limits:
          memory: 256M

networks:
  harmonix-net:
    driver: bridge
```

```
worker_processes auto;

events {
    worker_connections 1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    tcp_nopush      on;
    tcp_nodelay     on;
    keepalive_timeout  65;

    set_real_ip_from 173.245.48.0/20;
    set_real_ip_from 103.21.244.0/22;
    set_real_ip_from 103.22.200.0/22;
    set_real_ip_from 103.31.4.0/22;
    set_real_ip_from 141.101.64.0/18;
    set_real_ip_from 108.162.192.0/18;
    set_real_ip_from 190.93.240.0/20;
    set_real_ip_from 188.114.96.0/20;
    set_real_ip_from 197.234.240.0/22;
    set_real_ip_from 198.41.128.0/17;
    set_real_ip_from 162.158.0.0/15;
    set_real_ip_from 104.16.0.0/13;
    set_real_ip_from 104.24.0.0/14;
    set_real_ip_from 172.64.0.0/13;
    set_real_ip_from 131.0.72.0/22;

    set_real_ip_from 2400:cb00::/32;
    set_real_ip_from 2606:4700::/32;
    set_real_ip_from 2803:f800::/32;
    set_real_ip_from 2405:b500::/32;
    set_real_ip_from 2405:8100::/32;
    set_real_ip_from 2a06:98c0::/29;
    set_real_ip_from 2c0f:f248::/32;

    set_real_ip_from 127.0.0.1;
    set_real_ip_from 10.0.0.0/8;
    set_real_ip_from 172.16.0.0/12;
    set_real_ip_from 192.168.0.0/16;

    real_ip_header CF-Connecting-IP;

    limit_req_zone $binary_remote_addr
zone=api_limit:10m rate=15r/s;

    server {
        listen 80;
        server_name localhost;

        server_tokens off;

        client_max_body_size 10M;

        add_header Strict-Transport-Security
"max-age=31536000;
includeSubDomains" always;
```

```
        add_header X-Content-Type-
Options nosniff always;
        add_header X-Frame-Options
SAMEORIGIN always;
        add_header X-XSS-Protection "1;
mode=block" always;
        add_header Content-Security-
Policy "default-src 'self'; script-src 'self'
'unsafe-inline' 'unsafe-eval'
https://unpkg.com/
https://static.cloudflareinsights.com;
worker-src 'self' blob:; style-src 'self'
'unsafe-inline'; img-src 'self' data: blob:
https://*; font-src 'self' data:; connect-src
'self' ws: wss: https://*; media-src 'self'
blob: https://*;" always;
        add_header Referrer-Policy "no-
referrer-when-downgrade" always;

        # Backend API
        location /api/ {
            limit_req zone=api_limit
burst=30 nodelay;
            proxy_pass http://backend:3001;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP
$remote_addr;
            proxy_set_header X-Forwarded-
For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-
Proto $scheme;

            proxy_set_header Upgrade
$http_upgrade;
            proxy_set_header Connection
"upgrade";
        }
        location /socket.io/ {
            limit_req zone=api_limit
burst=30 nodelay;
            proxy_pass http://backend:3001;
            proxy_http_version 1.1;
            proxy_set_header Upgrade
$http_upgrade;
            proxy_set_header Connection
"upgrade";
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP
$remote_addr;
            proxy_set_header X-Forwarded-
For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-
Proto $scheme;

            proxy_read_timeout 86400;
            proxy_send_timeout 86400;
            proxy_buffering off;
        }

        # Frontend
        location / {
            proxy_pass http://frontend:3000;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP
$remote_addr;
            proxy_set_header X-Forwarded-
For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-
Proto $scheme;

            proxy_intercept_errors on;
            error_page 404 = /index.html;
        }
    }
}
```

```
const DB_NAME =
"HarmonixCryptoDB";
const STORE_NAME = "privateKeys";
const SESSION_STORE_NAME =
"sessionKeys";

const openDB = () => {
  return new Promise((resolve, reject) => {
    const request =
indexedDB.open(DB_NAME, 2);
    request.onupgradeneeded = (event) =>
{
      const db = event.target.result;
      if
(!db.objectStoreNames.contains(STORE_
NAME)) {
        db.createObjectStore(STORE_NAM
E);
      }
      if
(!db.objectStoreNames.contains(SESSIO
N_STORE_NAME)) {
        db.createObjectStore(SESSION_ST
ORE_NAME);
      }
    };
    request.onsuccess = () =>
resolve(request.result);
    request.onerror = () =>
reject(request.error);
  });
};

export const savePrivateKey = async
(userId, privateKey) => {
  const db = await openDB();
  const jwk = await
window.crypto.subtle.exportKey("jwk",
privateKey);
  return new Promise((resolve, reject) => {
    const transaction =
db.transaction(STORE_NAME,
"readwrite");
    const store =
transaction.objectStore(STORE_NAME);
    const request = store.put(jwk, userId);
    request.onsuccess = () => resolve();
    request.onerror = () =>
reject(request.error);
  });
};

export const removePrivateKey = async
(userId) => {
  const db = await openDB();
  return new Promise((resolve, reject) => {
    const transaction =
db.transaction(STORE_NAME,
"readwrite");
    const store =
transaction.objectStore(STORE_NAME);
    const request = store.delete(userId);
```

```
    request.onsuccess = () => {
      resolve();
    };
    request.onerror = () =>
reject(request.error);
  });
};

export const getPrivateKey = async
(userId) => {
  const db = await openDB();
  return new Promise((resolve, reject) =>
{
    const transaction =
db.transaction(STORE_NAME,
"readonly");
    const store =
transaction.objectStore(STORE_NAME)
;
    const request = store.get(userId);
    request.onsuccess = async () => {
      if (!request.result) return
resolve(null);
      try {
        const privateKey = await
window.crypto.subtle.importKey(
          "jwk",
          request.result,
          {
            name: "RSA-OAEP",
            hash: "SHA-256",
          },
          true,
          ["decrypt"]
        );
        resolve(privateKey);
      } catch (err) {
        await removePrivateKey(userId);
        resolve(null);
      }
    };
    request.onerror = () =>
reject(request.error);
  });
};

export const saveSessionKey = async
(chatId, aesKey) => {
  const db = await openDB();
  const jwk = await
window.crypto.subtle.exportKey("jwk",
aesKey);
  return new Promise((resolve, reject) =>
{
    const transaction =
db.transaction(SESSION_STORE_NAM
E, "readwrite");
    const store =
transaction.objectStore(SESSION_STO
RE_NAME);
    const request = store.put(jwk, chatId);
    request.onsuccess = () => {
      resolve();
    };
    request.onerror = () =>
reject(request.error);
  });
};

export const getSessionKey = async
(chatId) => {
  const db = await openDB();
  return new Promise((resolve, reject) => {
    const transaction =
db.transaction(SESSION_STORE_NAME
, "readonly");
    const store =
transaction.objectStore(SESSION_STOR
E_NAME);
    const request = store.get(chatId);
    request.onsuccess = async () => {
      if (!request.result) return resolve(null);
      try {
        const aesKey = await
window.crypto.subtle.importKey(
          "jwk",
          request.result,
          { name: "AES-GCM" },
          true,
          ["encrypt", "decrypt"]
        );
        resolve(aesKey);
      } catch (err) {
        const delTx =
db.transaction(SESSION_STORE_NAME
, "readwrite");
        delTx.objectStore(SESSION_STOR
E_NAME).delete(chatId);
        resolve(null);
      }
    };
    request.onerror = () =>
reject(request.error);
  });
};

export const clearCryptoDatabase = async
() => {
  return new Promise((resolve, reject) => {
    const request =
indexedDB.deleteDatabase(DB_NAME);

    request.onsuccess = () => {
      resolve();
    };

    request.onerror = (event) => {
      reject(event.target.error);
    };

    request.onblocked = () => {
      resolve();
    };
  });
};

export const generateKeyPair = async ()
=> {
  return await
window.crypto.subtle.generateKey(
    {
      name: "RSA-OAEP",
      modulusLength: 2048,
      publicExponent: new Uint8Array([1,
0, 1]),
      hash: "SHA-256",
    },
    true,
    ["encrypt", "decrypt"]
  );
```

```
};

export const exportPublicKey = async
(publicKey) => {
  const exported = await
window.crypto.subtle.exportKey("jwk",
publicKey);
  return JSON.stringify(exported);
};

export const importPublicKey = async
(publicKeyString) => {
  try {
    const jwk =
JSON.parse(publicKeyString);
    return await
window.crypto.subtle.importKey(
      "jwk",
      jwk,
      {
        name: "RSA-OAEP",
        hash: "SHA-256",
      },
      true,
      ["encrypt"]
    );
  } catch (error) {
    throw error;
  }
};

export const generateAndStoreRSAKeys
= async (userId) => {
  const pair = await generateKeyPair();
  await savePrivateKey(userId,
pair.privateKey);
  const pubKeyString = await
exportPublicKey(pair.publicKey);
  return pubKeyString;
};

export const generateSymmetricKey =
async () => {
  return await
window.crypto.subtle.generateKey(
    { name: "AES-GCM", length: 256 },
    true,
    ["encrypt", "decrypt"]
  );
};

export const encryptSymmetricKey =
async (recipientPublicKey, aesKey) => {
  const exportedAesKey = await
window.crypto.subtle.exportKey("raw",
aesKey);
  const encrypted = await
window.crypto.subtle.encrypt(
    { name: "RSA-OAEP" },
    recipientPublicKey,
    exportedAesKey
  );
  return
btoa(String.fromCharCode(...new
Uint8Array(encrypted)));
};

export const decryptSymmetricKey =
async (ourPrivateKey,
base64EncryptedAesKey) => {
  try {
    const binaryString =
window.atob(base64EncryptedAesKey);
    const bytes = new
Uint8Array(binaryString.length);
    for (let i = 0; i < binaryString.length;
i++) {
      bytes[i] = binaryString.charCodeAt(i);
    }

    const decryptedRaw = await
window.crypto.subtle.decrypt(
      { name: "RSA-OAEP" },
      ourPrivateKey,
      bytes
    );

    return await
window.crypto.subtle.importKey(
      "raw",
      decryptedRaw,
      { name: "AES-GCM" },
      true,
      ["encrypt", "decrypt"]
    );
  } catch (err) {
    throw err;
  }
};

export const rotateSessionKey = async
(members) => {
  const newAesKey = await
generateSymmetricKey();
  const encryptedKeys = await
Promise.all(
    members.map(async (member) => {
      if (!member.publicKey) {
        return null;
      }
      try {
        const pubKey = await
importPublicKey(member.publicKey);
        const encrypted = await
encryptSymmetricKey(pubKey,
newAesKey);
        return {
          recipientId: member.id,
          encryptedAesKey: encrypted
        };
      } catch (err) {
        return null;
      }
    })
  );

  const validKeys = encryptedKeys.filter(k
=> k !== null);

  return { newAesKey, encryptedKeys:
validKeys };
};

export const encryptMessage = async
(aesKey, plaintext) => {
  const enc = new TextEncoder();
  const iv =
window.crypto.getRandomValues(new
Uint8Array(12));
  const ciphertext = await
```

```
window.crypto.subtle.encrypt(
    { name: "AES-GCM", iv: iv },
    aesKey,
    enc.encode(plaintext)
  );

  const ivBase64 =
btoa(String.fromCharCode(...iv));
  const cipherBase64 =
btoa(String.fromCharCode(...new
Uint8Array(ciphertext)));
  return `${ivBase64}:${cipherBase64}`;
};

export const decryptMessage = async
(aesKey, encryptedPayload) => {
  try {
    const [ivBase64, cipherBase64] =
encryptedPayload.split(":");
    const iv =
Uint8Array.from(atob(ivBase64), (c) =>
c.charCodeAt(0));
    const ciphertext =
Uint8Array.from(atob(cipherBase64),
(c) => c.charCodeAt(0));

    const decrypted = await
window.crypto.subtle.decrypt(
      { name: "AES-GCM", iv: iv },
      aesKey,
      ciphertext
    );

    return new
TextDecoder().decode(decrypted);
  } catch {
    return " [SECURITY_LOCKOUT:
Click 'Regenerate' to restore access]";
  }
};

export const purgeChatCryptoData =
async (chatId) => {
  try {
    const db = await openDB();
    const transaction =
db.transaction(SESSION_STORE_NAM
E, "readwrite");
    const store =
transaction.objectStore(SESSION_STO
RE_NAME);

    const request = store.delete(chatId);

    request.onsuccess = () => {
    };

    localStorage.removeItem(`chat_key_$
{chatId}`);
    localStorage.removeItem(`chat_iv_${
chatId}`);

  } catch (error) {
  }
};

export async function
deriveVaultKey(password, salt) {
  const enc = new TextEncoder();
  const passwordKey = await
window.crypto.subtle.importKey(
    "raw",
    enc.encode(password),
    { name: "PBKDF2" },
    false,
    ["deriveKey"]
  );

  return window.crypto.subtle.deriveKey(
    {
      name: "PBKDF2",
      salt: enc.encode(salt),
      iterations: 100000,
      hash: "SHA-256",
    },
    passwordKey,
    { name: "AES-GCM", length: 256 },
    false,
    ["wrapKey", "unwrapKey", "encrypt",
"decrypt"]
  );
}

export function generateRecoveryKey() {
  const charset =
'ABCDEFGHJKLMNPQRSTUVWXYZ2
3456789';
  const randomValues = new
Uint8Array(16);
  window.crypto.getRandomValues(rando
mValues);

  let key = '';
  for (let i = 0; i < 16; i++) {
    key += charset[randomValues[i] %
charset.length];
    if ((i + 1) % 4 === 0 && i !== 15) {
      key += '-';
    }
  }
  return key;
}

export async function
wrapPrivateKey(privateKey, password) {
  const salt =
window.crypto.getRandomValues(new
Uint8Array(16));
  const saltStr =
btoa(String.fromCharCode(...salt));

  const vaultKey = await
deriveVaultKey(password, saltStr);

  const iv =
window.crypto.getRandomValues(new
Uint8Array(12));

  const wrappedArrayBuffer = await
window.crypto.subtle.wrapKey(
    "pkcs8",
    privateKey,
    vaultKey,
    { name: "AES-GCM", iv }
  );

  const combined = new
Uint8Array(iv.length +
wrappedArrayBuffer.byteLength);
  combined.set(iv);
```

```
  combined.set(new
Uint8Array(wrappedArrayBuffer),
iv.length);

  return {
    encryptedKey:
btoa(String.fromCharCode(...combined))
,
    salt: saltStr
  };
}

export async function
unwrapPrivateKey(encryptedPayload,
salt, password) {
  const vaultKey = await
deriveVaultKey(password, salt);

  const combined = new Uint8Array(
    atob(encryptedPayload)
      .split("")
      .map((c) => c.charCodeAt(0))
  );

  const iv = combined.slice(0, 12);
  const wrappedKey =
combined.slice(12);

  return
window.crypto.subtle.unwrapKey(
    "pkcs8",
    wrappedKey,
    vaultKey,
    { name: "AES-GCM", iv },
    { name: "RSA-OAEP", hash: "SHA-
256" },
    true,
    ["decrypt"]
  );
}
```