НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету	**Завідувач кафедри комп'ютерних наук**
_____ Ігор БОЛБОТ	_____ Белла ГОЛУБ
(підпис)	(підпис)
«___» _____ 2026 р.	«___» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему: «Програмне забезпечення веборієнтованої системи для обліку та відтворення музичного контенту»

Спеціальність «121 Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ **Ганна ВАЙГАНГ**
(підпис)

Керівник бакалаврської кваліфікаційної роботи
Асистент _____ **Тетяна Баранова**
(підпис)

Консультант бакалаврської кваліфікаційної роботи
к.т.н, доцент _____ **Ірина Бородкіна**
(підпис)

Виконав _____ **Мирослав Артюхов**
(підпис)

Київ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук
к.т.н., доцент _____ Белла ГОЛУБ
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Артюхову Мирославу Юрійовичу
(прізвище, ім'я, по батькові)

Спеціальність «121 Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»
Тема бакалаврської кваліфікаційної роботи

«Програмне забезпечення веборієнтованої системи для обліку та відтворення музичного контенту»

затверджена наказом від «19» грудня 2025 р. № 3204 «С»

Термін подання завершеної роботи на кафедру «22» травня 2026 р.

Вихідні дані до бакалаврської кваліфікаційної роботи: Інформаційні джерела з веброзробки, технологій Node.js, React, PostgreSQL, MongoDB, YouTube Data API та UML-моделювання.
Перелік питань, що підлягають дослідженню: Аналіз предметної області музичних стрімінгових платформ; формування функціональних та нефункціональних вимог; UML-моделювання системи; проєктування гібридної бази даних PostgreSQL та MongoDB; реалізація серверної та клієнтської частини вебзастосунку; інтеграція із зовнішнім YouTube Data API; реалізація механізмів пошуку, відтворення музики та збору статистики прослуховувань; тестування та оцінка ефективності роботи системи.

Перелік графічних документів (за необхідності).

_____

Дата видачі завдання «19» грудня 2026 р.

Керівник бакалаврської кваліфікаційної роботи
Асистент _____ **Тетяна Баранова**
(підпис)

Консультант бакалаврської кваліфікаційної роботи
к.т.н, доцент _____ **Ірина Бородкіна**
(підпис)

Завдання взяв до виконання _____ **Мирослав Артюхов**
(підпис)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) – інтерфейс програмування застосунків.

CRUD (Create, Read, Update, Delete) – базові операції роботи з даними.

DOM [25] (Document Object Model) – об’єктна модель документа вебсторінки.

ER-діаграма (Entity Relationship Diagram) – діаграма сутностей та зв’язків.

HLS (HTTP Live Streaming) – протокол потокового передавання мультимедійного контенту.

HTTP (HyperText Transfer Protocol) – протокол передавання гіпертексту.

ISO/IEC [22][23] – міжнародні стандарти Міжнародної організації зі стандартизації та Міжнародної електротехнічної комісії.

JSON [17] (JavaScript Object Notation) – текстовий формат обміну даними.

MPEG-DASH (Dynamic Adaptive Streaming over HTTP) – технологія адаптивного потокового передавання мультимедійного контенту.

ORM (Object Relational Mapping) – технологія об’єктно-реляційного відображення.

RBAC (Role-Based Access Control) – модель керування доступом на основі ролей.

SQL (Structured Query Language) – мова структурованих запитів до баз даних.

UML (Unified Modeling Language) – уніфікована мова моделювання.

UUID (Universally Unique Identifier) – універсальний унікальний ідентифікатор.

ВСТУП

Сьогодні музичні стрімінгові сервіси стали звичною частиною повсякденного життя. Користувачі слухають музику вдома, у транспорті, під час роботи чи відпочинку, використовуючи для цього вебплатформи та мобільні застосунки. За останні роки спосіб споживання музичного контенту суттєво змінився. Якщо раніше для прослуховування музики потрібно було купувати фізичні носії або завантажувати аудіофайли на пристрій, то сьогодні більшість користувачів надає перевагу онлайн-сервісам, які забезпечують швидкий доступ до великого каталогу треків через мережу Інтернет.

Сучасні музичні платформи, такі як Spotify [13] або YouTube Music [15], являють собою складні програмні системи, що поєднують засоби потокового відтворення аудіо, рекомендаційні алгоритми, пошукові механізми та системи збору статистики. Такі сервіси аналізують поведінку користувачів, формують персоналізовані рекомендації та забезпечують стабільне відтворення контенту навіть при великому навантаженні на систему.

Попри значну кількість готових музичних платформ, потреба у власних спеціалізованих системах залишається актуальною. Це особливо важливо для бізнесу, який потребує детального обліку відтворення музичного контенту та доступу до статистичних даних. Наприклад, для закладів громадського харчування або корпоративних мереж важливо мати можливість контролювати історію прослуховувань та отримувати аналітичну інформацію щодо використання музики.

У межах даної роботи розглядається розробка веборієнтованої системи обліку та відтворення музичного контенту. Основною особливістю системи є використання зовнішніх API для отримання музичних даних без необхідності локального зберігання великих обсягів аудіо файлів. Такий підхід дозволяє зменшити вимоги до серверної інфраструктури та забезпечити доступ до широкого каталогу музичного контенту.

Під час розробки системи виникла необхідність вирішення кількох технічних задач. Однією з них стало проєктування структури зберігання даних, яка забезпечувала б швидку роботу системи та можливість обробки великої кількості подій. Для цього було використано гібридний підхід із застосуванням реляційної бази даних PostgreSQL [1] та документо-орієнтованої бази MongoDB [2]. Окрему увагу приділено роботі із зовнішнім YouTube API [6] та механізму автоматичної зміни API-ключів на результати збережені у БД у випадку перевищення лімітів запитів.

Об’єктом дослідження є процес управління музичним контентом у веб середовищі, що включає пошук, відтворення та збір статистики прослуховувань. Предметом дослідження виступають програмні засоби, алгоритми та архітектурні рішення, необхідні для створення подібної системи.

Метою бакалаврської роботи є розробка програмного забезпечення веборієнтованої системи обліку та відтворення музичного контенту, яка забезпечує пошук треків, створення плейлистів, відтворення музики та збереження інформації про взаємодію користувача із системою.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. Проаналізувати сучасні музичні стрімінгові сервіси та визначити їхні особливості.
2. Сформувати функціональні та нефункціональні вимоги до системи.
3. Побудувати UML-моделі для опису структури та логіки роботи програмного забезпечення.
4. Розробити гібридну структуру зберігання даних із використанням PostgreSQL [1] та MongoDB [2].
5. Реалізувати механізм пошуку музичного контенту із використанням зовнішнього API.
6. Розробити клієнтську та серверну частини системи й виконати їх тестування.

Під час виконання роботи були використані методи системного аналізу, об’єктно-орієнтованого проєктування та сучасні підходи до побудови

вебзастосунків. Практична цінність роботи полягає у створенні програмної системи, яка може використовуватися для відтворення музичного контенту, збору статистики прослуховувань та подальшого аналізу поведінки користувачів.

Апробація результатів роботи. Апробація результатів бакалаврської кваліфікаційної роботи була здійснена шляхом підготовки та представлення матеріалів на тему «Програмне забезпечення веборієнтованої системи обліку та відтворення музичного контенту» на студентській науковій конференції кафедри комп'ютерних наук Національного університету біоресурсів і природокористування України у 2026 році.

Структура записки. Робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі виконано системний аналіз предметної області та сформовано вимоги до програмної системи. Другий розділ присвячений проєктуванню інформаційного та програмного забезпечення. У третьому розділі описано процес розробки програмного забезпечення та реалізації основних модулів системи. Четвертий розділ містить рекомендації щодо впровадження та експлуатації системи, а також результати тестування. У додатках наведено фрагменти програмного коду та допоміжні матеріали.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

У сучасному цифровому середовищі процес споживання музичного контенту зазнав суттєвих змін. Традиційні фізичні носії інформації по типу компакт-дисків, флеш-накопичувачів поступово втратили свою актуальність, поступившись місцем хмарним та веборієнтованим платформам, які забезпечують користувачам швидкий доступ до глобальних мультимедійних ресурсів. Цей перехід призвів до формування нового класу програмних систем – стрімінгових мультимедійних систем. Такі сервіси є складними програмними системами, що поєднують у собі функції потокового відтворення, розподіленого зберігання інформації та аналізу великих обсягів даних на основі музичних вподобань користувачів.

Предметною областю даної дипломної роботи є комплексні процеси обліку, пошуку, каталогізації та відтворення музичного контенту у веборієнтованому середовищі. Головною особливістю цієї предметної області є гостра необхідність інтеграції різнорідних джерел даних. Зокрема, йдеться про взаємодію із зовнішніми API, що надають легальний доступ до величезних масивів мультимедійного контенту.

На відміну від класичних мультимедійних систем минулого покоління, сучасні стрімінгові платформи зазвичай не зберігають сам аудіоконтент (mp3, wav чи flac файли) на своїх локальних серверах. Замість цього широко використовується архітектурний підхід агрегації даних. При такому підході розроблювана система оперує виключно метаданими – цифровими паспортами треків, що включають назву композиції, ім'я виконавця, унікальний глобальний ідентифікатор, тривалість у мілісекундах та посилання на графічну обкладинку. Сам же медіа-контент витягується та транслюється динамічно, в режимі реального часу через зовнішні глобальні сервіси, такі як YouTube Data API [6].

Застосування такої архітектури дозволяє зменшити вимоги до апаратної інфраструктури, знизити витрати на обслуговування дата-центрів та, що найважливіше з юридичної точки зору, зменшити ризики, які пов’язані з порушенням авторських прав на розповсюдження аудіозаписів.

У межах досліджуваної предметної області можна чітко виділити такі ключові структурні та логічні складові:

- Користувачі системи (слухачі, адміністратори) як основні користувачі системи;
- Музичні треки як основні інформаційні об’єкти системи, що підлягають обробці;
- Плейлисти як складні структури даних для агрегації та персональної організації контенту;
- Механізми інформаційного пошуку, фільтрації та генерації персоналізованих рекомендацій;
- Підсистема телеметрії та збору докладної статистики прослуховувань (веб-аналітика).

Особливо важливу роль відіграє саме збір аналітичних даних. Щоб система мала комерційну цінність, вона повинна з високою точністю фіксувати взаємодію користувача з контентом. Наприклад реєстрацію точного моменту початку відтворення у вигляді timestamp, тривалість прослуховування у секундах, а також усі ітеративні дії користувача по типу постановка на паузу, примусовий пропуск пісні, додавання до улюблених. Такі зібрані дані про поведінку користувачів можуть використовуватись для подальшого аналізу та побудови інтелектуальних рекомендаційних алгоритмів.

Разом з тим, значним інженерним викликом є жорстка залежність від зовнішніх інфраструктурних сервісів. Використання публічних API завжди накладає суворі ліміти у вигляді квоти на кількість запитів за одиницю часу. Цей факт вимагає реалізації серверних механізмів: кешування, балансування навантаження, ротації ключів та поступового зниження функціональності

замість повного краху системи у разі часткової недоступності зовнішніх ресурсів. Отже, предметна область характеризується високим рівнем мережевої інтеграції, необхідністю безперервної обробки великих обсягів даних та високими галузевими вимогами до надійності, відмовостійкості та горизонтальної масштабованості розроблюваної програмної системи.

1.2 Аналіз вимог до програмної системи

Формування вимог до програми – це базовий етап розробки. Від того, наскільки чітко вони будуть прописані, залежить кінцева якість продукту. Всі вимоги були поділені на дві категорії: функціональні та нефункціональні. Це було зроблено спираючись на міжнародні стандарти серії ISO/IEC 25010 [22].

Функціональні вимоги детально описують, що саме має робити система, які бізнес-функції виконувати та як реагувати на конкретні дії користувачів.

1. Платформа повинна підтримувати повноцінний цикл реєстрації та авторизації користувачів. Профілі мають бути надійно захищені сучасними криптографічними методами хешування та безпечними HTTP-Only сесіями.
2. Після успішної авторизації користувачі повинні отримати доступ до основного функціоналу музичної платформи, який включає пошук музичного контенту, потокове відтворення треків, формування персональної музичної бібліотеки, створення та управління плейлистами, додавання треків до улюблених, а також можливість підписки на виконавців. Система повинна забезпечувати зручний інтерфейс для взаємодії з контентом та персоналізації музичних уподобань користувача.
3. Особлива, увага в системі приділяється алгоритмам пошуку контенту. Система повинна здійснювати пошук гібридним методом. Тобто, при введенні запиту вона має одночасно сканувати локальну базу даних на предмет збігів серед плейлистів та відправляти форматований запит до глобального YouTube Data API [6] [3]. Цей механізм забезпечує користувачу великий вибір треків.

4. Крім того, пошукова система повинна мати механізм фільтрації. Вона зобов'язана ігнорувати всі відеоролики, тривалість яких становить менше ніж 30 секунд. Це важлива вимога для музичного сервісу, оскільки вона дозволяє відсіяти велику кількість нерелевантного контенту – YouTube Shorts, тизерів, рекламних вставок та інших відео, не призначених для повноцінного прослуховування музики.

5. Музичний контент повинен відтворюватися плавно з використанням технологій потокової буферизації без необхідності повного попереднього завантаження медіафайлів у пам’ять пристрою користувача. Паралельно у фоновому режимі має функціонувати підсистема телеметрії та збору статистики, яка автоматично фіксує ключові дії користувача: початок відтворення, пропуск треків, додавання треків до улюблених, взаємодію з плейлистами та інші події, необхідні для подальшої аналітики та персоналізації сервісу.

Нефункціональні вимоги, на відміну від функціональних, описують не те, що робить система, а те, як добре та за яких умов вона це робить. Стандарт ISO/IEC 25010 [22] виділяє кілька головних метрик якості:

- Продуктивність. Серверна частина повинна мати здатність швидко та без затримок обробляти значну кількість конкурентних запитів одночасно. Для досягнення цієї мети було впроваджено індексацію в базах даних (B-Tree індекси), що гарантує виконання пошуку по локальній базі за менше ніж 500 мілісекунд.
- Надійність та стійкість до відмов. Система має бути стійкою до зовнішніх збоїв. Оскільки платформа залежить від зовнішнього провайдера (Google), виникає ризик блокування за перевищення ліміту квот (HTTP 403 Quota Exceeded). Згідно з вимогами, сервер повинен вміти перехоплювати такі винятки і непомітно для користувача перемикатися результати які уже були збережені у базі даних.

- Ергономічність та зручність використання. Інтерфейс користувача має бути повністю адаптивним. Це означає, що DOM-дерево [25] сторінки повинно динамічно перебудовуватися, і коректно відображати контент як на великих широкоформатних моніторах, так і на малих екранах мобільних пристроїв.
- Безпека. Це один з основних критеріїв. Паролі користувачів категорично заборонено зберігати у відкритому текстовому вигляді, вони повинні криптографічно перетворюватися на хеші за допомогою алгоритмів bcrypt [8]. Для генерації унікальних системних ідентифікаторів повинні застосовуватися ідентифікатори формату UUIDv4, що робить неможливим їх підбір або передбачення зловмисниками.

Основні, найбільш критичні для успіху проєкту вимоги для наочності та зручності подальшого тестування ми систематизували та зібрали у таблицю 1.1.

Таблиця 1.1

Класифікація вимог до системи обліку музики

Категорія згідно ISO 25010	**Унікальний ідентифікатор**	**Опис вимоги**	**Рівень пріоритету**
Функціональна придатність	FR-01	Гібридний пошук музики через YouTube Data API.	Критичний
Функціональна придатність	FR-02	Автоматичний облік секунди, на якій користувач перервав пісню.	Високий
Функціональна придатність	FR-03	Управління плейлистами з налаштуванням приватності.	Високий
Продуктивність	NFR-01	Пошук у локальній базі має працювати швидше ніж 500 мілісекунд.	Високий
Надійність	NFR-02	Автоматична зміна API-ключів на результати з БД при помилках для стабільної роботи.	Критичний
Безпека	NFR-03	Зберігання паролів виключно у вигляді хешів.	Критичний

Ці вимоги стали основою для подальшого проєктування системи та написання коду.

1.3 Моделювання предметної області

Перед тим як приступати безпосередньо до кодування та розробки архітектури баз даних, інженеру необхідно чітко розібратися з бізнес-логікою майбутньої системи. У світовій інженерії програмного забезпечення для цього традиційно використовують методи UML-моделювання. Для даного проєкту обрано стандарт UML. Цей інструментарій дозволяє формалізувати вимоги до системи у вигляді графічних моделей.

Концептуально розроблена платформа функціонує як сучасна, високоінтерактивна цифрова радіостанція або корпоративний аудіо-хаб. У рамках цієї екосистеми чітко виділяються три основні ролі: звичайний слухач, куратор музичного ефіру – діджей та адміністратор. Щоб детально показати, як саме ці суб'єкти взаємодіють між собою, з системними ресурсами та безпосередньо з музичним контентом, ми побудували комплекс із кількох базових UML-діаграм, що описують систему під різними кутами зору.

1.3.1 Діаграма прецедентів предметної області. Діаграма прецедентів (Use Case Diagram) є типом UML-діаграми, яка демонструє систему з точки зору кінцевого користувача, описуючи що робить система, а не як вона це робить. Діаграма показує функції системи для різних користувачівміж трьома акторами.

- **Слухач:** Актор базового рівня. Його прецеденти включають авторизацію, використання пошукового модуля, відтворення музичних треків у плеєрі та прояв активності (натискання лайк або пропуск). Ці дії формують історію взаємодії користувача із системою. Кожен клік Слухача формує телеметричну історію, що слугує базисом для алгоритмів рекомендацій.
- **Діджей:** Актор з розширеними привілеями, що керує музичним потоком. Його прецеденти охоплюють створення плейлистів, наповнення їх новим контентом, а також обробку замовлень від слухачів. Окремим

прецедентом є "Отримання статистики". На діаграмі ми використали залежність включення (<<include>>), яка показує, що прецедент "Генерація звіту" обов'язково включає прецедент "Аналіз топових пісень та логів Слухачів".

- **Адміністратор:** Актор, віддалений від безпосереднього музичного процесу. Його головні прецеденти – управління технічними ресурсами. Це конфігурація API-ключів, моніторинг лімітів Google Cloud Console, управління базами даних та підписання/реєстрація нових ліцензій. Такий рольовий поділ (RBAC) забезпечує ізоляцію та безпеку: Слухач не має доступу до зміни чужих плейлистів, а Діджей не може випадково видалити API-ключ платформи.

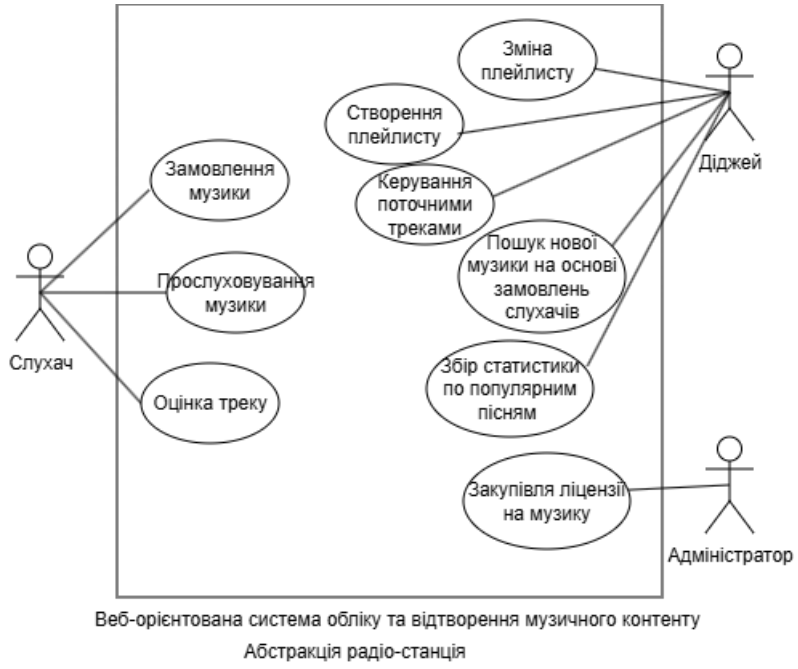


Рис. 1.1 Діаграма прецедентів веборієнтованої системи обліку та відтворення музичного контенту

1.3.2 Діаграма послідовності. Діаграма послідовності показує процеси в часі. Вона відображає повідомлення, які об'єкти передають один одному. Послідовність подій відображається зверху вниз.

На діаграмі показаний процес роботи з музикою на замовлення. Усе починається з підготовки: Діджей планує ефір і відправляє запит до

Адміністратора на перевірку прав. Адміністратор підтверджує, що треки доступні. Далі діджей створює плейлист у базі даних і формує чергу пісень.

Потім підключається Слухач. Він обирає пісню і надсилає повідомлення-замовлення. Діджей бачить це замовлення, ставить трек у плеєр, і сервер починає віддавати аудіопотік у браузер Слухача. Поки музика грає, Слухач може відправити реакцію (наприклад, натиснути подобається). Цей сигнал іде на сервер, а гібридна база даних зберігає цю оцінку. Ця діаграма чудово ілюструє повне коло обміну даними між клієнтом та бекендом.

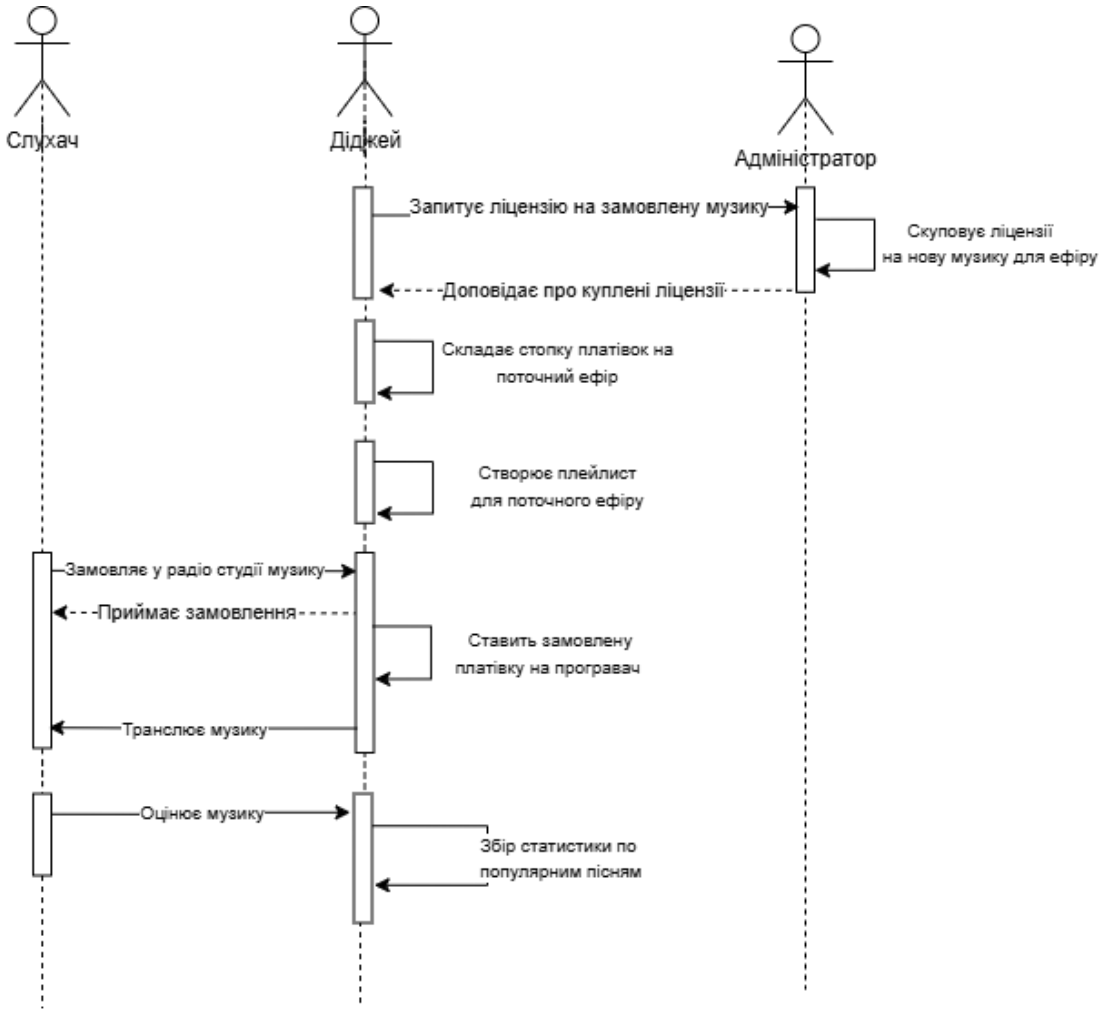


Рис. 1.2 Діаграма послідовності взаємодії акторів системи

1.3.3 Діаграма активності. Діаграма діяльності потрібна для опису алгоритмів. Вона за своєю структуроюдуже схожа на класичну блок-схему.

Ця діаграма показує логіку з умовними переходами (decision nodes). Ми детально зобразили алгоритм обробки музичного замовлення.

Робота починається з початкового вузла. Сервер отримує запит на пісню. Далі йде головний блок перевірки умов (у вигляді ромба). Система перевіряє: "Чи доступна ця пісня в базі або через API?".

Потік ділиться на дві гілки.

Позитивна гілка. Якщо пісня знайдена, програма додає її до черги. Трек завантажується в плеєр. Музика починає грати. Одночасно запускається паралельний процес – сервер починає фонове логування подій. Коли трек закінчується, цикл успішно завершується.

Негативна гілка. Пісні немає. Або зовнішній API повернув помилку. Система повинна коректно обробити помилку. Виконується альтернативна гілка сценарію. Програма генерує повідомлення про помилку. Вона повідомляє користувача, що трек недоступний. Після цього алгоритм повертається на початковий етап і чекає наступного запиту.

Таке детальне моделювання дозволяє зменшити кількість багів на етапі написання коду.

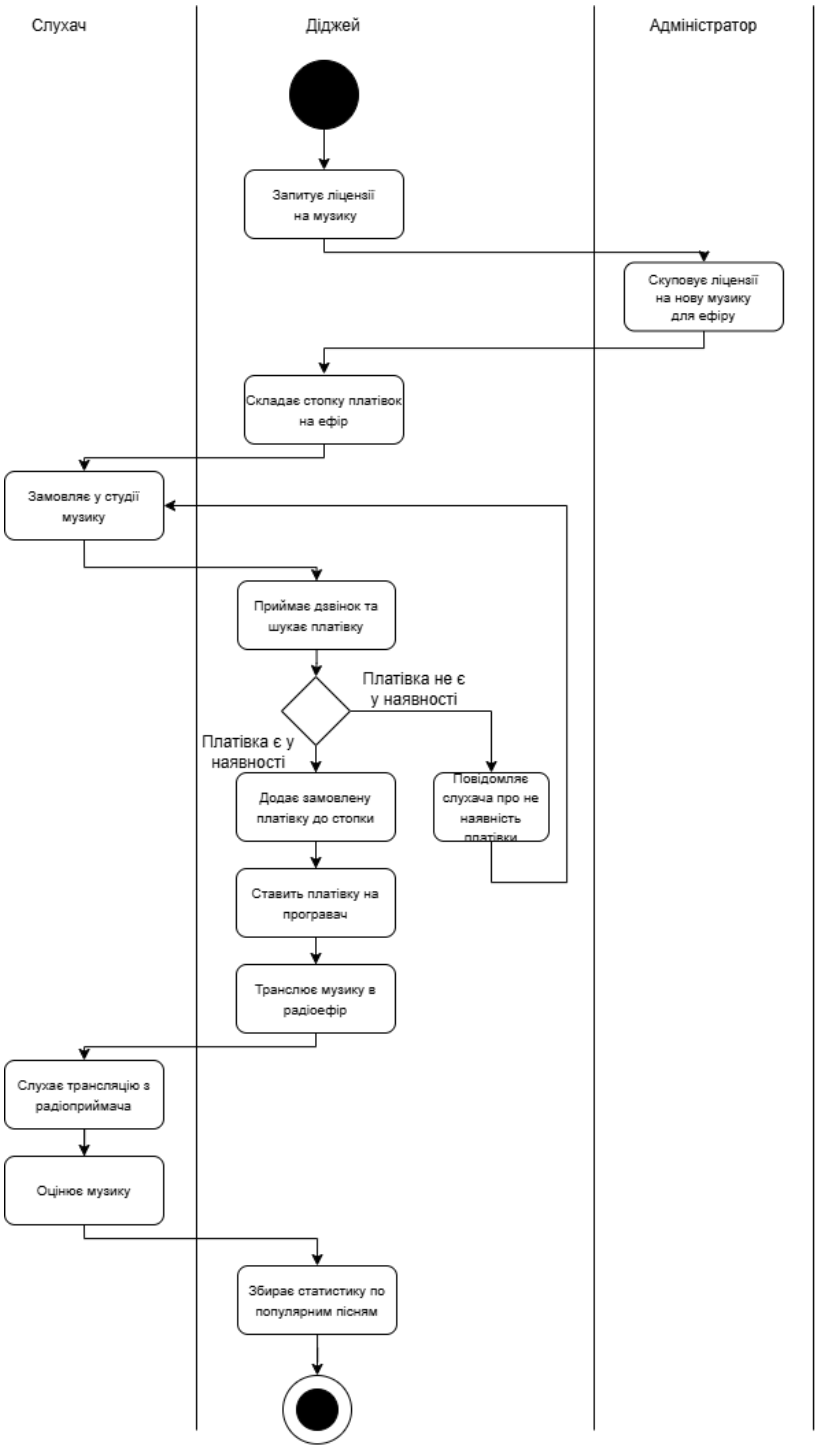


Рис. 1.3 Діаграма діяльності процесу замовлення та трансляції контенту

1.3.4 Абстракції предметної області. Перед тим як переходити до коду та баз даних, треба виділити головні сутності (абстракції) та прописати їхні властивості. Згідно з рисунком 1.4, ми виділили такі основні об'єкти:

- **Слухач.** Його властивості – ім'я та номер телефону. Його завдання: замовляти пісні, слухати ефір, писати відгуки та змінювати гучність.

- **Діджей.** Має ім'я та псевдонім. Його обов'язки значно ширші. Він приймає дзвінки, працює з плейлистами, шукає музику і збирає статистику.
- **Адміністратор.** Зберігає лише ім'я. Його головна функція – покупка ліцензій.
- **Плейлист.** Це логічний контейнер. Має назву ефірної програми, містить список пісень і посилання на діджея-куратора.
- **Музичний трек.** Зберігає інформацію про пісню: назву, автора, тривалість та якість звуку.
- **Музичний плеєр.** Суто технічний об'єкт. Його властивості – робоча частота, рівень гучності та потужність. Його завдання – приймати сигнал і відтворювати звук.

Рис. 1.4. Базові абстракції предметної області

1.3.5 Діаграма класів предметної області. Діаграма класів описує статичну структуру. Вона показує, як саме абстракції пов'язані між собою.

Тут ми бачимо типи зв'язків. Об'єкт "Плеєр" обслуговує багатьох "Слухачів". Цей зв'язок працює напряму.

Клас "Діджей" може створити сотні "Плейлистів". Тут використовується зв'язок типу "один-до-багатьох" (1..*).

Сам "Плейлист" складається з багатьох "Треків". Але один конкретний трек також може знаходитися у десяти різних плейлистах одночасно. Це складний зв'язок "багато-до-багатьох" (*..*). У базі даних PostgreSQL [1] для цього потрібна окрема таблиця-місток (наприклад, PlaylistTrack).

Ця діаграма була нашим кресленням під час створення реляційної бази.

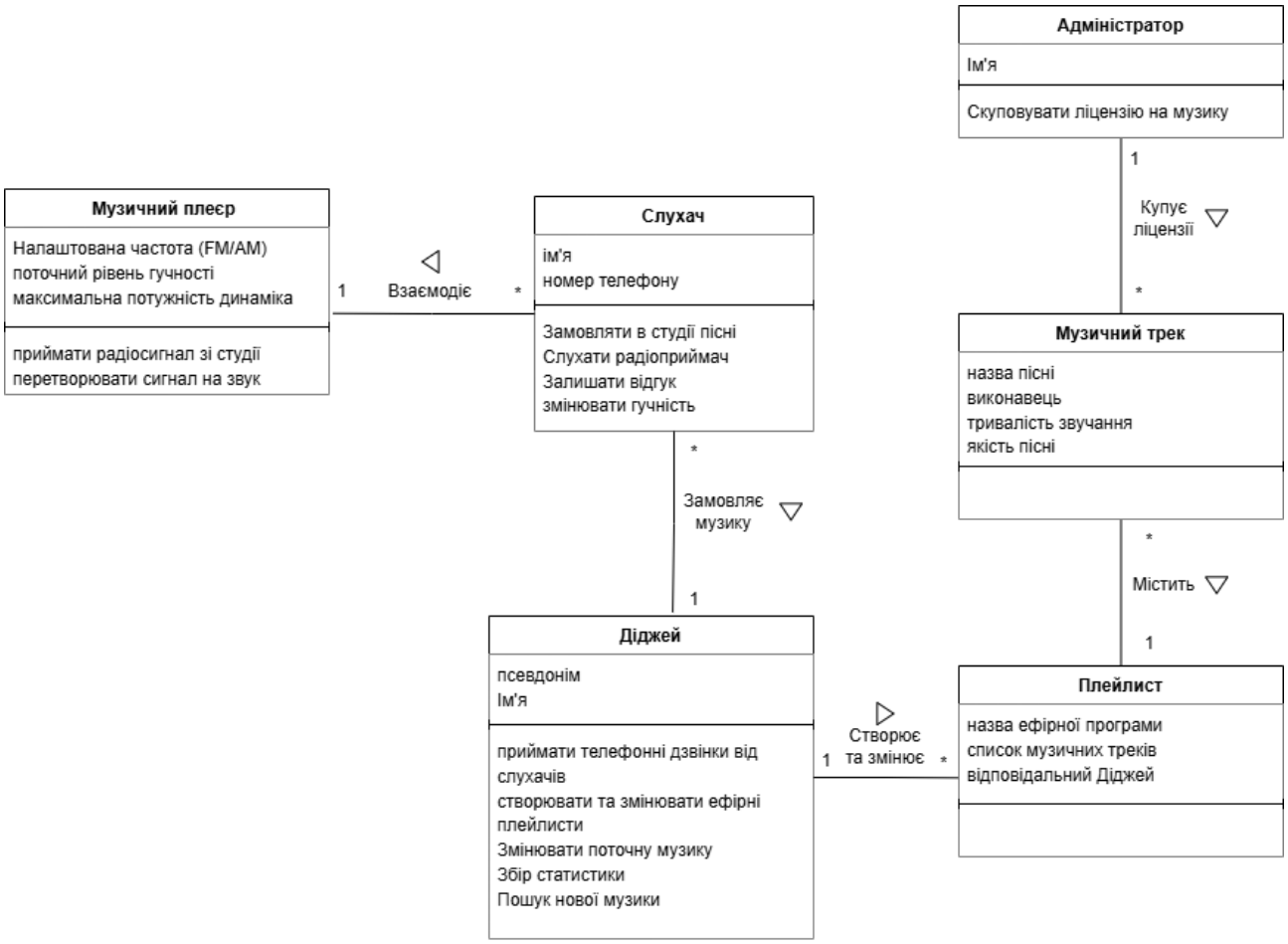


Рис. 1.5 Діаграма зв'язків між класами системи

1.3.6 Діаграма кооперацій предметної області. Діаграма кооперацій детальніше описує взаємодію між об’єктами системи, та показує процес взаємодії між ними. Вона демонструє не просто наявність відношень між абстракціями, а конкретні поведінкові функції, деталізуючи, який саме об'єкт ініціює дію та за які процеси відповідає.

Як видно з рисунка 1.6, взаємодія в системі чітко розділена на три логічні блоки (кооперації), що відповідають основним ролям системи:

- **Кооперація споживання контенту (Слухач).** Клас Слухач ініціює процес прослуховування. Він взаємодіє з системою через виклик методів +слухатиРадіо() та +змінюватиГучність(). Ці команди надходять до класу Радіо (який є абстракцією програмного плеєра). Своєю чергою, об'єкт Радіо самостійно викликає внутрішні методи +програвати плейлист() та +відтворюватиЗвук(), звертаючись до сутності Плейлист. Сам Плейлист логічно складається з об'єктів МузичнийТрек, які безпосередньо транслюються користувачу.
- **Кооперація управління ефіром (Діджей).** Клас Діджей виконує роль куратора контенту. Його головна задача – формування музичного потоку, що реалізується через виклик методу +створитиПлейлист(). У результаті спрацювання цього методу ініціалізується новий об'єкт класу Плейлист, який агрегує в собі необхідні композиції (клас МузичнийТрек).
- **Кооперація адміністрування ресурсів (Адміністратор).** Цей блок відображає роботу з юридичною та технічною базою платформи. Клас Адміністратор використовує метод +купити ліцензію на музичний трек(). Цей метод безпосередньо взаємодіє з класом «Музичний трек», що дозволяє легалізувати композицію, перш ніж вона стане доступною для використання діджеями у своїх плейлистах.

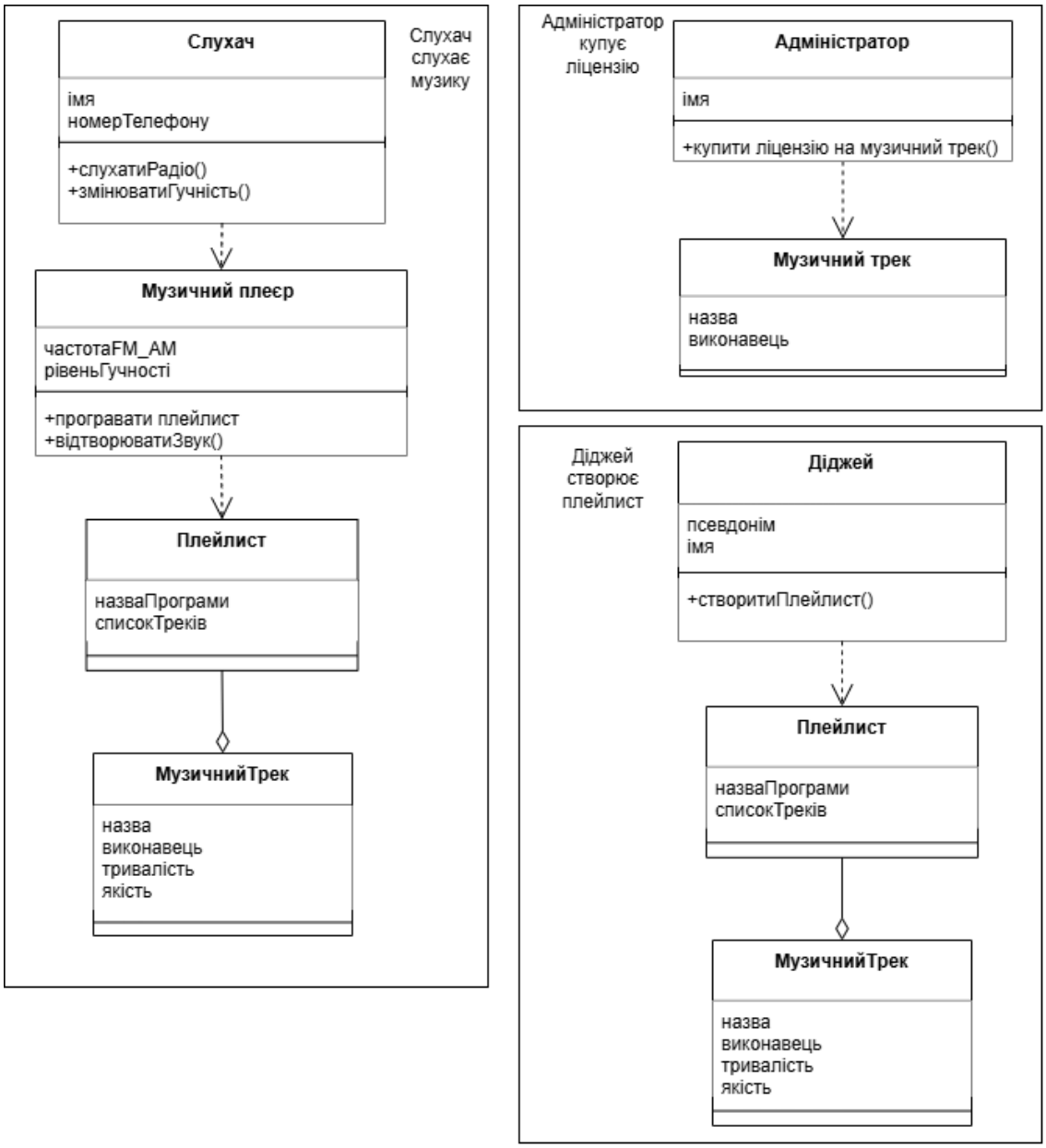


Рис. 1.6 Діаграми кооперацій із визначенням методів

1.4 Огляд інформаційних джерел та існуючих рішень

Щоб створити якісний продукт, треба розуміти, як працює сучасний ринок. Сфера потокового аудіо змінюється дуже швидко. Тому ми проаналізували останні тренди та розібрали популярні сервіси конкурентів.

1.4.1 Основні напрями розвитку музичних стрімінгових платформ. Давайте подивимося, як усе починалося. У 2000-х роках ринок фізичних носіїв

почав стрімко падати. Продажі дисків і касет обвалилися з 36,9 мільярда доларів до 15,9 мільярда. Людям набридло збирати коробки з дисками на полицях.

Тоді з'явився Napster. Це був легендарний піратський сервіс для обміну файлами. Він мав шалену популярність. До весни 2000 року там було вже 20 мільйонів користувачів. Згодом його закрили через нескінченні суди.

Проте звичка слухати музику онлайн залишилася. Індустрії довелося шукати легальну альтернативу.

Містком став магазин iTunes від компанії Apple. Там можна було купувати пісні поштучно. Це було досить зручно. Але справжня революція сталася трохи пізніше. Швидкість інтернету зросла. Мобільні пристрої стали потужнішими. З'явилися перші платформи, що працювали за підпискою. Це Spotify [13], Pandora та інші.

Вони запропонували абсолютно нову модель. Користувач сплачує фіксовану суму на місяць і отримуєш доступ до мільйонів треків.

Сьогодні це абсолютна норма. За даними IFPI [11] , у 2025 році доходи від стрімінгових сервісів перевищили 22 мільярди доларів та склали близько 70% світового ринку записаної музики. Музику більше не качають, її просто вмикають. Ми виділили три ключові тренди, за якими зараз рухається цей ринок:

1. Адаптивні технології доставки звуку. Сучасні платформи не змушують чекати. Вони розбивають пісню на невеликі сегменти (через протоколи HLS або MPEG-DASH). Якщо ваш інтернет працює погано, система миттєво знизить якість звуку. Але сама пісня не перерветься. Це робить процес прослуховування дуже комфортним.

2. Боротьба з кризою метаданих. Пісня в інтернеті – це не просто звук. Це великий набір супровідного тексту. Хто співає. Хто написав слова. Уому належать авторські права та роялті. Якщо ці бази даних не будуть впорядковані, музиканти просто не отримають свої роялті. Зараз індустрія активно стандартизує ці дані. Правильна архітектура бази даних – це фундамент будь-якого сучасного стрімінгу.

3. Зміна правового поля в Україні. Розробляти систему без огляду на закони неможливо. З початку 2023 року в Україні діє оновлений Закон про авторське право [12]. Він серйозно наблизив нас до стандартів ЄС. Установи поступово цифровізуються. Проте в нас досі є велика проблема з моніторингом музики. Наприклад, у кафе чи магазинах. Часто інспектори перевіряють це вручну. Тому бізнесу виникає потреба у програмних системах. Вони мають автоматично збирати логи прослуховувань для прозорої звітності.

1.4.2 Огляд існуючих програмних рішень. На ринку є кілька очевидних лідерів. Щоб зрозуміти їхні сильні та слабкі сторони, ми зробили детальне порівняння.

Spotify [13]**.** Це абсолютний гігант. База налічує понад 100 мільйонів ліцензійних треків. Його головна фішка – алгоритми штучного інтелекту. Вони точно вгадують настрій слухача. Вони самі генеруть персоналізовані плейлисти. Також додаток має дуже продуману навігацію. Але є проблема. Звичайний артист не може просто так завантажити туди свою музику. Для цього потрібен платний посередник (дистриб'ютор типу Distrokid). Крім того, для сторонніх розробників цей сервіс є закритим. Ви не можете підключитися до їхньої бази даних. Ви не зможете зробити на їхній основі власне онлайн-радіо для ресторану.

SoundCloud [14]**.** Це улюблена платформа для незалежних музикантів. Її база величезна – понад 320 мільйонів завантажень. Сюди будь-хто може завантажити свій звук абсолютно безкоштовно. Тут багато унікальних реміксів та каверів. Їх просто немає на інших сервісах. Ще одна сильна сторона – соціальність. Люди залишають коментарі прямо поверх звукової доріжки. Проте через таку відкритість там часто панує хаос. Назви пісень записані як завгодно. Також часто трапляються порушення авторських прав.

Apple Music. Сервіс робить ставку на дуже високу якість звуку без втрат (Lossless). Але з точки зору веброзробки він не дуже зручний. Цей продукт

глибоко прив'язаний до закритої екосистеми Apple. Він не має безкоштовної версії для масового тестування.

YouTube Music [15]**.** Для розробників це один із найбільш перспективних варіантів. Сервіс працює на базі найбільшого відеохостингу світу. Там можна знайти будь-що. І найголовніше – компанія Google дає доступ до свого офіційного API. Це робить YouTube ідеальним джерелом для створення власних легальних додатків. Але тут є суттєве обмеження. Це жорсткі ліміти. Кожному розробнику дають лише 10 000 одиниць квоти на день. А один простий пошук по базі забирає одразу 100 одиниць. Тобто ви можете зробити всього 100 пошукових запитів. Для реальної платформи цього замало. Нам довелося серйозно подумати, як обійти це обмеження на рівні коду.

1.4.2 Обґрунтування необхідності власної розробки. Незважаючи на наявність потужних комерційних стрімінгових платформ, створення власної веборієнтованої системи обліку та відтворення музичного контенту є доцільним з огляду на низку технічних та бізнесових факторів:

- **Повний контроль над даними активності користувача.** Глобальні сервіси, такі як Spotify [13] або YouTube Music [15], функціонують як закриті системи. Вони надають лише узагальнену статистику, не дозволяючи розробнику отримати доступ до первинних логів взаємодії. Власна розробка дозволяє реалізувати детальний облік кожної секунди прослуховування (параметр listenDurationSeconds), фіксувати точні моменти переривання треку та патерни поведінки користувачів. Зберігання цих даних у базі MongoDB [2] формує унікальний масив інформації для побудови власних рекомендаційних алгоритмів у майбутньому.
- **Оптимізація інфраструктурних витрат.** Завдяки архітектурному підходу агрегації метаданих, система не потребує оренди великих сховищ для зберігання аудіофайлів. Використання гібридної моделі даних (PostgreSQL [1] та MongoDB [2]) дозволяє досягти високої

продуктивності при мінімальних витратах на серверні потужності, що складніше реалізувати при використанні готових платформ.

З інженерної точки зору, реалізація такої системи є доволі таки складним викликом, що вимагає інтеграції різнорідних джерел даних та побудови відмовостійкої клієнт-серверної архітектури.

1.5 Постановка завдання

Після аналізу ринку та існуючих проблем, можна сформулювати завдання для нашої дипломної роботи

1.5.1 Формулювання загальної задачі. Основна ціль – розробити веборієнтовану систему для обліку та відтворення музики. Платформа має працювати за клієнт-серверною моделлю.

Для зберігання інформації ми використаємо гібридний підхід. Реляційна база PostgreSQL [1] відповідатиме за профілі користувачів. А база MongoDB [2] використовуватиметься для зберігання історії прослуховувань та метадані пісень. Система має вміти шукати треки через YouTube Data API [6] [3], приймати замовлення, плавно відтворювати аудіо і непомітно логувати всі дії користувача.

1.5.2 Вхідні дані. Для нормальної роботи програма буде приймати такі дані:

- Інформація для входу (email, паролі, імена або ID від Google OAuth [21]).
- Пошукові запити (текст, який користувач вводить у рядок пошуку).
- Дані від YouTube API [6] у форматі JSON [17]. До них входять ідентифікатор відео, назва, автор та тривалість треку.
- Фонові сигнали від музичного плеєра. Це інформація про те, що користувач зробив з піснею (розпочав відтворення, пропустив трек або поставив лайк) і на якій секунді це відбулося.

1.5.3 Вихідні дані. Обробляючи ці запити, система буде видавати:

- Готові вебсторінки з інтерфейсом (плейлисти, кнопки плеєра).

- Аудіопотік, який іде прямо в браузер користувача
- Структуровані відповіді у форматі JSON [17], щоб сторінки оновлювалися без перезавантаження.
- Записи в базах даних (наприклад, нові записи в PostgreSQL [1] та нові записи логів в MongoDB [2]). З цих записів потім формується аналітика.

1.5.4 Сценарій використання. Базовий сценарій використання системи описує типовий сценарій взаємодії користувача із системою: від моменту авторизації на платформі до прослуховування контенту та автоматичної фіксації аналітичних даних сервером. У стандартному режимі роботи повний цикл роботи виглядає наступним чином:

- **Авторизація:** Користувач відкриває клієнтський вебдодаток та проходить процедуру автентифікації. Це може бути класичний вхід за допомогою електронної пошти та пароля (який хешується алгоритмом bcrypt [8]) або авторизація через систему Google OAuth 2.0 [21].
- **Пошук контенту:** Користувач вводить назву пісні у рядок глобального пошуку. Система ініціює гібридний алгоритм: опитує локальний кеш та відправляє запит до YouTube Data API [6] [3]. Сервер автоматично відфільтровує нерелевантний контент, зокрема відеоролики тривалістю менше ніж 30 секунд (YouTube Shorts), повертаючи клієнту лише повноцінні композиції.
- **Організація бібліотеки:** Знайдений контент користувач може додати до улюблених (поставити лайк) або згрупувати у власні збірки. Через модальне вікно створюється персональний плейлист, інформація про який надійно фіксується у реляційній базі PostgreSQL [1].
- **Ініціалізація відтворення:** Користувач натискає на картку треку, після чого на нижній панелі активується глобальний музичний плеєр. Система завантажує ідентифікатор відео у прихований компонент YouTube IFrame та розпочинає трансляцію звуку. Завдяки технології Virtual DOM [25] (React [3]), інтерфейс оновлюється і без перезавантаження сторінки.

- **Фоновий збір телеметрії (Облік):** Щойно плеєр починає відтворювати музику, клієнтський додаток (через компонент PlayerContext) генерує асинхронний фоновий POST-запит на бекенд.
- **Збереження історії:** Отримавши сигнал про відтворення, сервер (Node.js [4]) виконує атомарну операцію upsert у базі MongoDB [2]: кешує метадані треку в колекції Songs та додає подію в колекцію ListeningHistory.
- **Фіксація переривань:** Якщо користувач вирішить зупинити або пропустити пісню до її завершення, система зафіксує точну секунду зупинки (параметр listenDurationSeconds) у системних логах InteractionLog, що може використовуватись для подальшого аналізу поведінки користувачів.

1.5.5 Кількісні критерії. Для забезпечення належного рівня якості програмного продукту та його відповідності інженерним стандартам, встановлюються строгі кількісні критерії та обмеження, наведені у таблиці 1.2.

Таблиця 1.2

Кількісні критерії оцінки якості функціонування системи

Критерій	**Значення**	**Навіщо це потрібно**
Ліміт пошуку (пагінація)	<= 15 записів	Для зменшення навантаження на клієнтський інтерфейс та від надлишку інформації під час відображення списків.
Швидкість запису логів	<= 100 мілісекунд	Щоб збір статистики не заважав плавному відтворенню музики.
Фільтр за тривалістю	> 30 секунд	Відсіювання коротких відеоформатів для отримання нормальних музичних треків.

1.5.6 Критерії успішності роботи. Проєкт вважається успішно реалізованим, якщо за результатами розробки та тестування виконуються наступні технічні та функціональні критерії:

1. **Стабільність роботи механізму пошуку контенту:** алгоритм гібридного пошуку стабільно функціонує за умов інтенсивного навантаження, забезпечуючи автоматичну та непомітну для користувача ротацію

результатів з API-ключа на данні збережені у базі даних при досягненні лімітів квот зовнішніх провайдерів.

2. **Узгодженість даних між PostgreSQL** [1] **та MongoDB** [2]**:** забезпечено повну синхронізацію та консистентність інформації між реляційною СУБД PostgreSQL [1] (управління профілями, плейлистами та підписками) та документо-орієнтованою СУБД MongoDB [2] (кешування метаданих треків та логування історії).
3. **Точність збору статистики прослуховувань:** підсистема аналітики стабільно фіксує кількісні метрики взаємодії, зокрема точний час прослуховування до моменту переривання треку (параметр listenDurationSeconds). Це підтверджує можливість подальшого використання даних для рекомендаційних алгоритмів на основі зібраних масивів даних.
4. **Ефективність фонових процесів:** запис логів взаємодії та історії прослуховувань виконується асинхронно, не спричиняючи блокувань основного потоку (Event Loop) та не погіршуючи швидкість відгуку користувацького інтерфейсу.
5. **Відповідність галузевим стандартам:** програмний продукт відповідаєфункціональним вимогам, визначеним згідно зі стандартом ISO/IEC 25010 [22].

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Процес проєктування є важливою стадією життєвого циклу розробки програмного забезпечення, на якій системні вимоги переходять у конкретні архітектурні рішення. У цьому розділі детально розглядається вибір стратегії зберігання даних, розробка моделей, розбиття системи на архітектурні модулі та їхня взаємодія.

2.1 Логічна модель даних у вигляді ER-діаграми

При проєктуванні архітектури даних для мультимедійної системи головним викликом стала проблема різного типу навантаження. Музична платформа одночасно обробляє транзакційні запити (оновлення профілю, зміна статусу плейлиста), які вимагають суворої консистентності (ACID), та інтенсивні потоки аналітичних даних (сотні тисяч подій play, pause, skip щогодини).

Використання виключно реляційної бази даних (наприклад, PostgreSQL [1] або MySQL) швидко призвело б до вузьких місць у продуктивності. Запис неструктурованих логів у реляційну таблицю з великою кількістю індексів і зовнішніх ключів блокує ресурси (table/row locking), що може спричиняти затримки в роботі клієнтського інтерфейсу.

Для вирішення цієї архітектурної проблеми було використано підхід Polyglot Persistence. Цей підхід передбачає децентралізацію відповідальності за дані та використання декількох різних типів СУБД у межах одного застосунку, де кожна база виконує ту роботу, до якої вона найкраще адаптована.

1. **Реляційна СУБД (PostgreSQL** [1]**):** Використовується для управління структурними та бізнес-критичними даними. Вона відповідає за користувачів, їхні налаштування, плейлисти та зв'язки між ними. PostgreSQL [1] гарантує строгу типізацію та підтримку каскадних

операцій: якщо користувач видаляє свій обліковий запис, база даних автоматично видаляє всі його персональні плейлисти, що спрощує реалізацію логіки цілісності даних для перевірки цілісності.

2. **Документо-орієнтована СУБД (MongoDB** [2]**):** Використовується для роботи з великими обсягами неструктурованих або слабко структурованих даних. У MongoDB [2] зберігаються метадані треків, закешовані із зовнішніх API (назви, посилання на зображення), а також щоденник системних логів (історія прослуховувань). Відсутність жорсткої схеми дозволяє швидко зберігати велику кількість JSON-документів [17].

ER-діаграма системи демонструє логічне розділення: реляційні таблиці з'єднані первинними і зовнішніми ключами, а зв'язок з MongoDB [2] здійснюється через логічний індекс – унікальний ідентифікатор youtubeId.

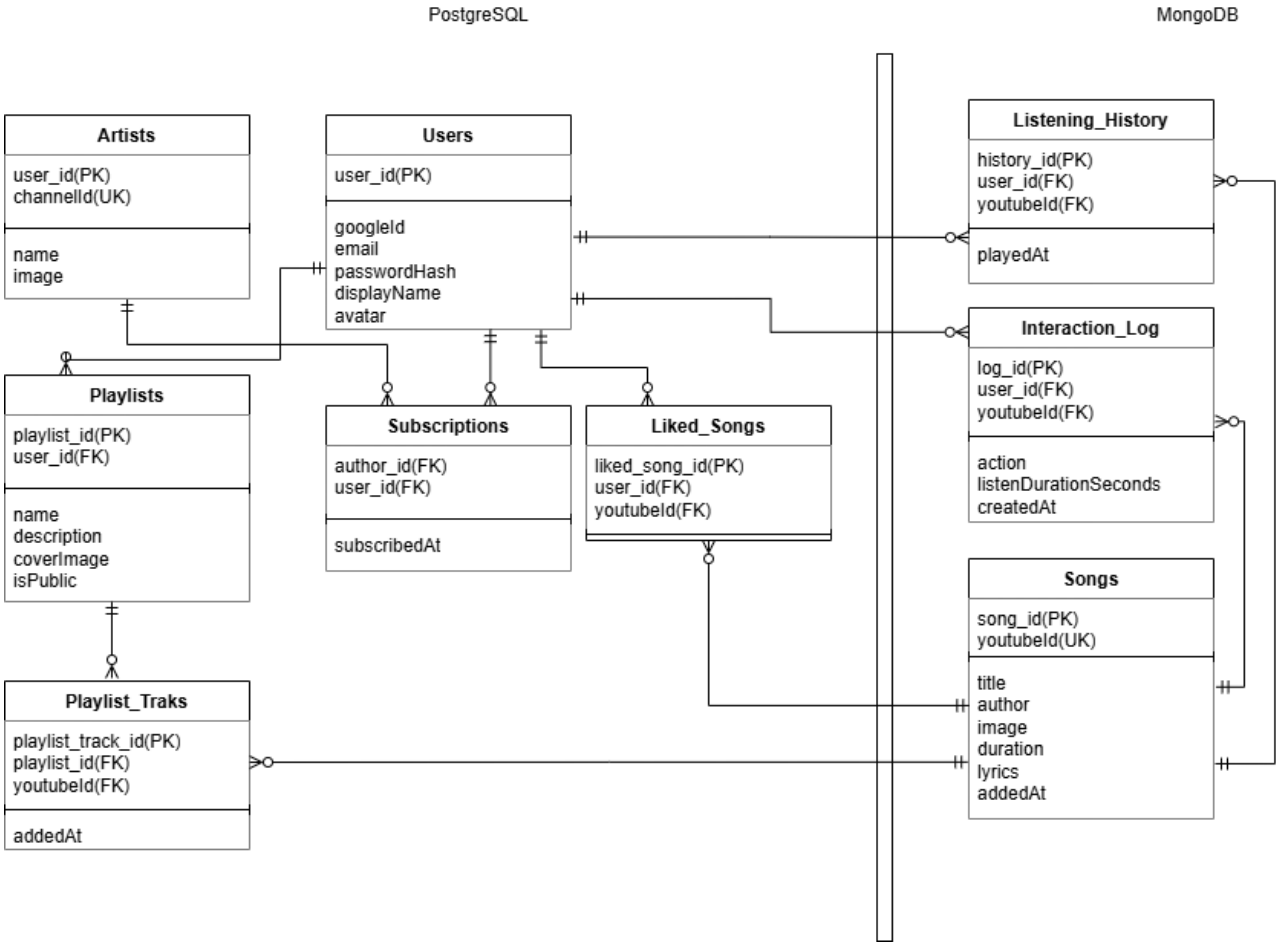


Рис. 2.1 ER-діаграма гібридної бази даних (PostgreSQL та MongoDB)

2.2 Діаграма класів та кооперацій

Програма не може бути суцільною стіною тексту. Код треба дробити на логічні частини. В інженерії для цього використовують класи та кооперації. Класи – це наші головні дійові особи. Кооперації – це сценарії, за якими ці особи спілкуються між собою.

2.2.1 Діаграма класів системи. Діаграма класів описує структуру об'єктів бекенду.

- **Клас User:** Інкапсулює дані користувача (id, displayName, email). Він містить методи для хешування пароля (за допомогою алгоритму bcrypt [8]) під час реєстрації, унеможливлюючи зберігання паролів у відкритому вигляді.
- **Клас Playlist:** Належить конкретному користувачу (зв'язок 1..* від User). Містить атрибути керування доступом (isPublic).
- **Клас PlaylistTrack:** Виконує роль сполучної ланки для реалізації зв'язку багато-до-багатьох між плейлистами та треками.
- **Клас Song (MongoDB** [2]**):** Відіграє роль локального кешу метаданих. Містить методи для автоматичного upsert (оновлення або створення), що дозволяє підтримувати актуальні метадані для плеєра.
- **Клас ListeningHistory (MongoDB** [2]**):** Акумулює записи телеметрії. Жоден інший клас не має прямої жорсткої залежності від нього, що дозволяє виконувати запис паралельно до основного потоку виконання програми.

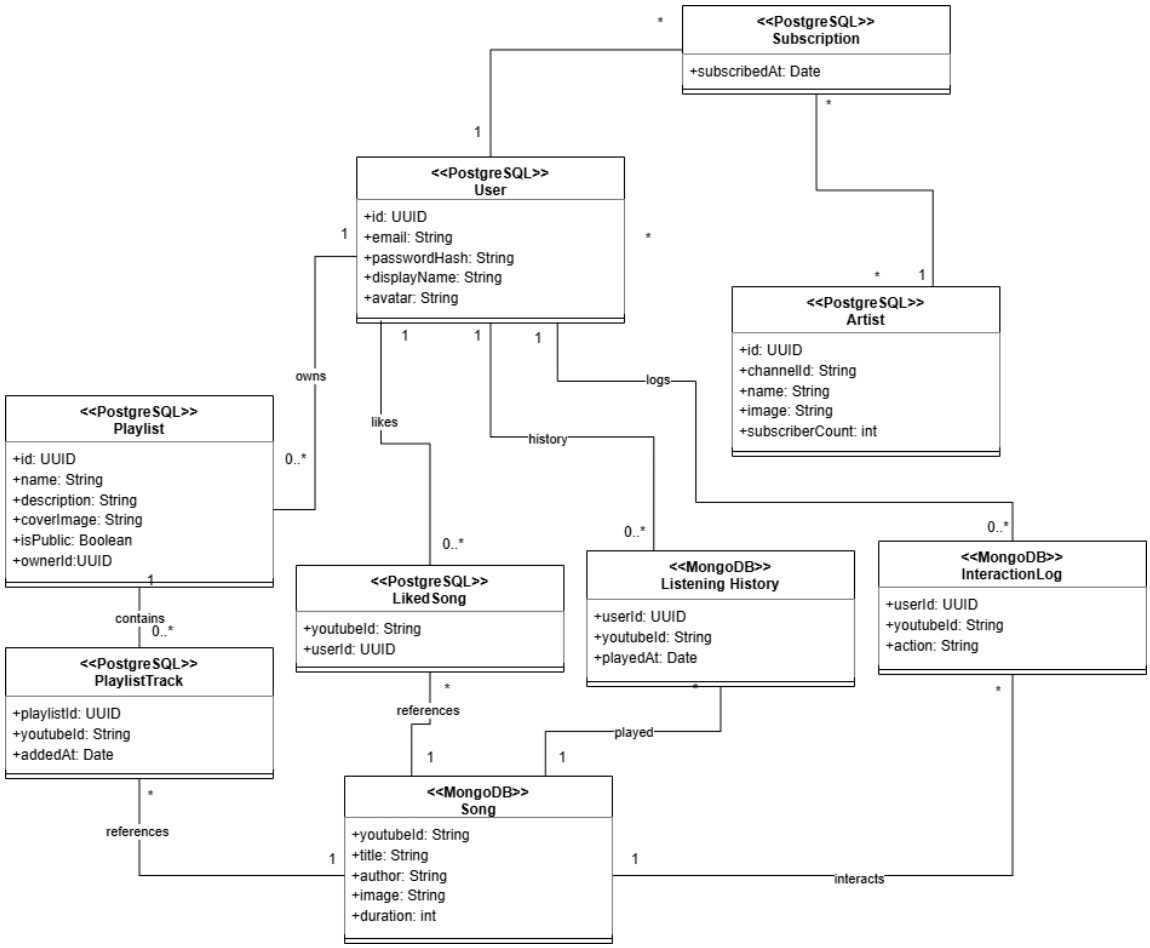


Рис. 2.2 Діаграма класів розроблюємої системи

2.2.2 Діаграми кооперацій системи. Кооперації показують, як класи працюють разом для виконання конкретної задачі. Ми докладно розписали п'ять найголовніших процесів нашої системи.

Кооперація 1: Процес авторизації (Auth) У системі реалізовано два способи авторизації. Користувач може ввести логін і пароль, або увійти через Google. Для цього ми використали бібліотеку Passport.js. Коли людина тисне "Увійти через Google", браузер перенаправляє її на сторінку підтвердження. Після згоди Google повертає серверу токен автентифікації. Система бере email із токена і шукає його в таблиці User (PostgreSQL [1]). Якщо такого email немає, створюється новий акаунт. Потім Passport.js бере ID користувача і зберігає його в сесію сервера. Для тих, хто реєструється поштою, працює інший алгоритм. Ми пропускаємо пароль через бібліотеку bcrypt [8]. Вона перетворює його на

криптографічний хеш. У базі зберігається лише цей хеш. Це гарантує повну безпеку.

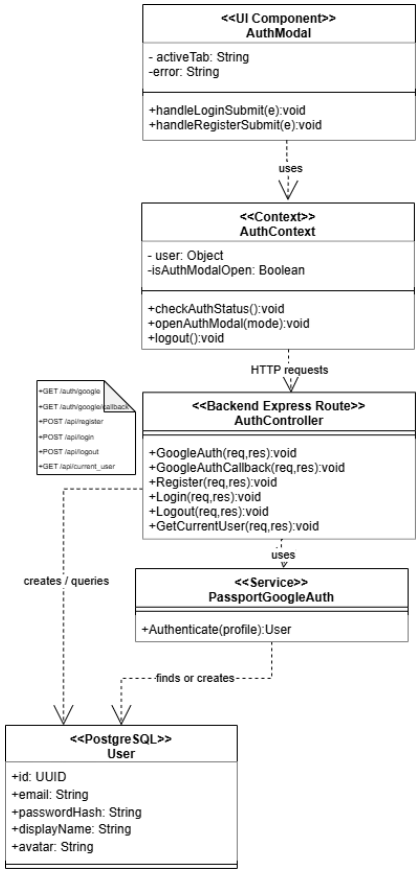


Рис. 2.3 Діаграма кооперації підсистеми авторизації

Кооперація 2: Історія та лайки (Interaction) Це дуже навантажений процес. Коли людина ставить лайк, спрацьовує гібридна логіка. Запит приходить у файл userProfile.js. Спочатку сервер іде в MongoDB [2]. Він створює там запис Song, щоб зберегти назву і картинку треку. Потім він іде в PostgreSQL [1] і створює запис у таблиці LikedSong. Механізм збереження історії працює окремо. Коли пісня починає грати, MongoDB [2] створює новий запис ListeningHistory.

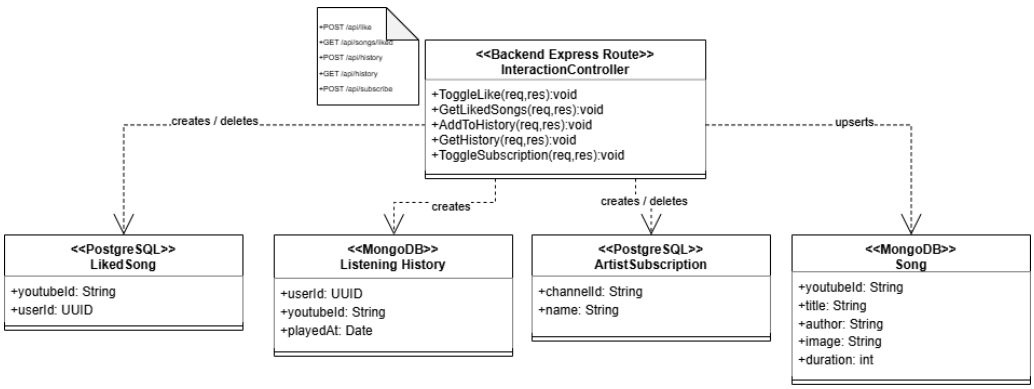


Рис. 2.4 Діаграма кооперації підсистеми взаємодії та обліку

Кооперація 3: Відтворення музики (Player) На рисунку 2.5 зображено архітектуру музичного плеєра. Тут зображено плеєр. Візуальний компонент Player завжди знає, яка пісня грає, завдяки PlayerContext. Цей контекст керує відтворенням через спеціальний інтерфейс IAudioPlayer (який ми реалізували через YoutubeIframeAdapter). Поки грає музика, наш плеєр робить запит у фоновому режимі до зовнішнього сервісу LRCLib API [9]. Звідти він тягне текст пісні, щоб показати його слухачу в режимі караоке.

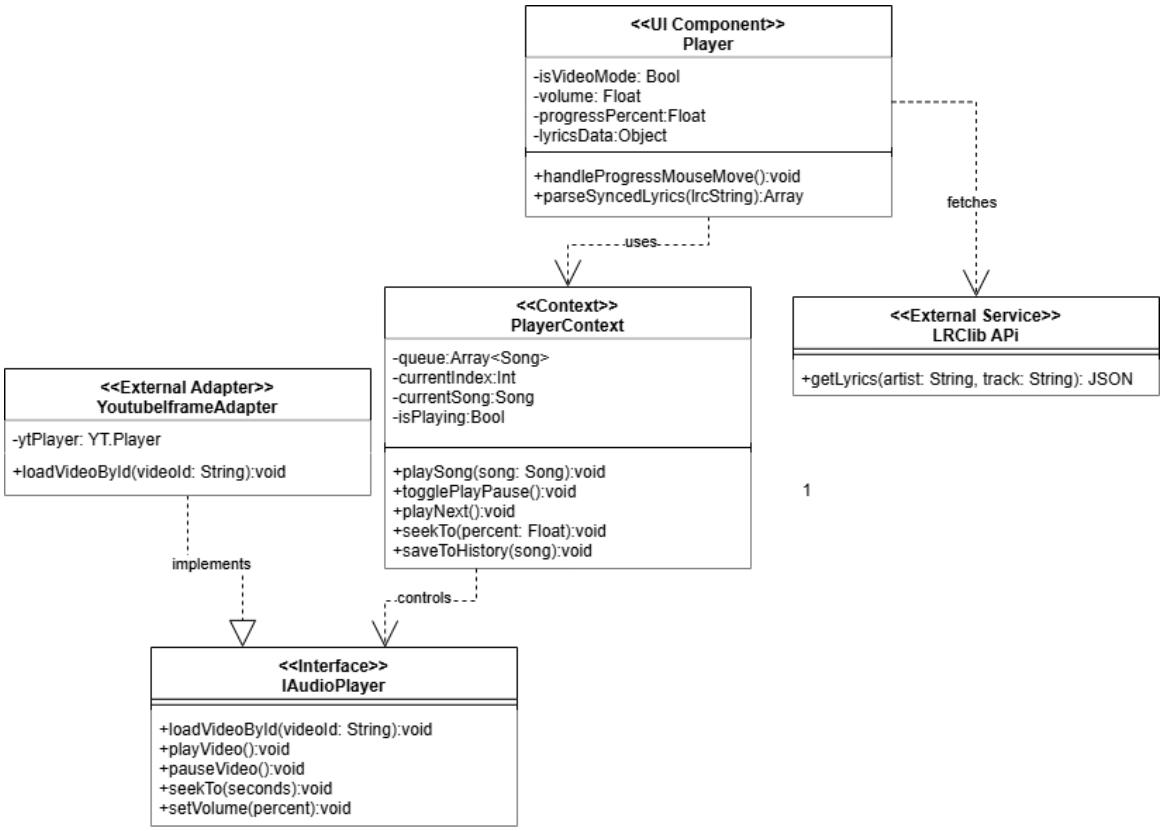


Рис. 2.5 Діаграма кооперації підсистеми відтворення музики

Кооперація 4: Робота з плейлистами (Playlist) Цей процес описаний у файлі playlists.js. Користувач тисне кнопку "Додати в плейлист". Контролер перевіряє права. Лише автор може змінювати свій список. Якщо пісень додається багато (наприклад, імпорт), система використовує метод bulkCreate у PostgreSQL [1]. Це дозволяє зберегти велику кількість треків за одну долю секунди. Коли ж треба показати плейлист на екрані, працює функція populateMongoSongs. Вона отримує ID з PostgreSQL [1], іде в MongoDB [2], отримує метадані треків та зображення, зліплює це все в один JSON-об'єкт [17] і відправляє у браузер.

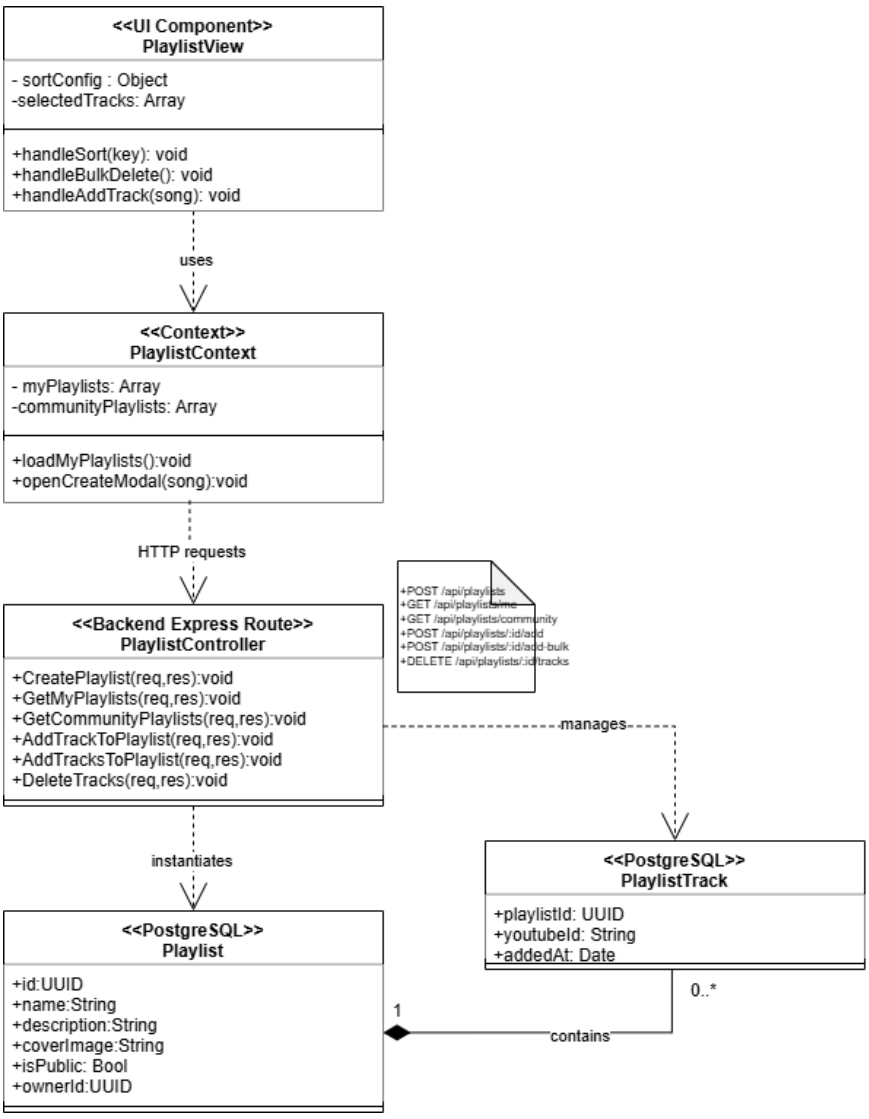


Рис. 2.6 Діаграма кооперації підсистеми керування плейлистами

Кооперація 5: Пошук контенту (Search) Тут реалізовано гібридний пошук. Алгоритм працює дуже розумно. Спочатку сервер робить запит до YouTube API [6]. YouTube не віддає тривалість відео у простому пошуку. Тому сервер збирає ID і робить другий, додатковий запит за статистикою. Тривалість приходить у дивному форматі ISO 8601 [23] (наприклад, PT3M45S). Програма перетворює його на звичайні секунди за допомогою регулярних виразів. Усі короткі відео (до 30 секунд) одразу відкидаються. Паралельно працює пошук по нашому сайту. Система шукає локальні плейлисти. Вона перевіряє, чи збігається назва, опис, або чи є всередині плейлиста пісня, яку шукає користувач.

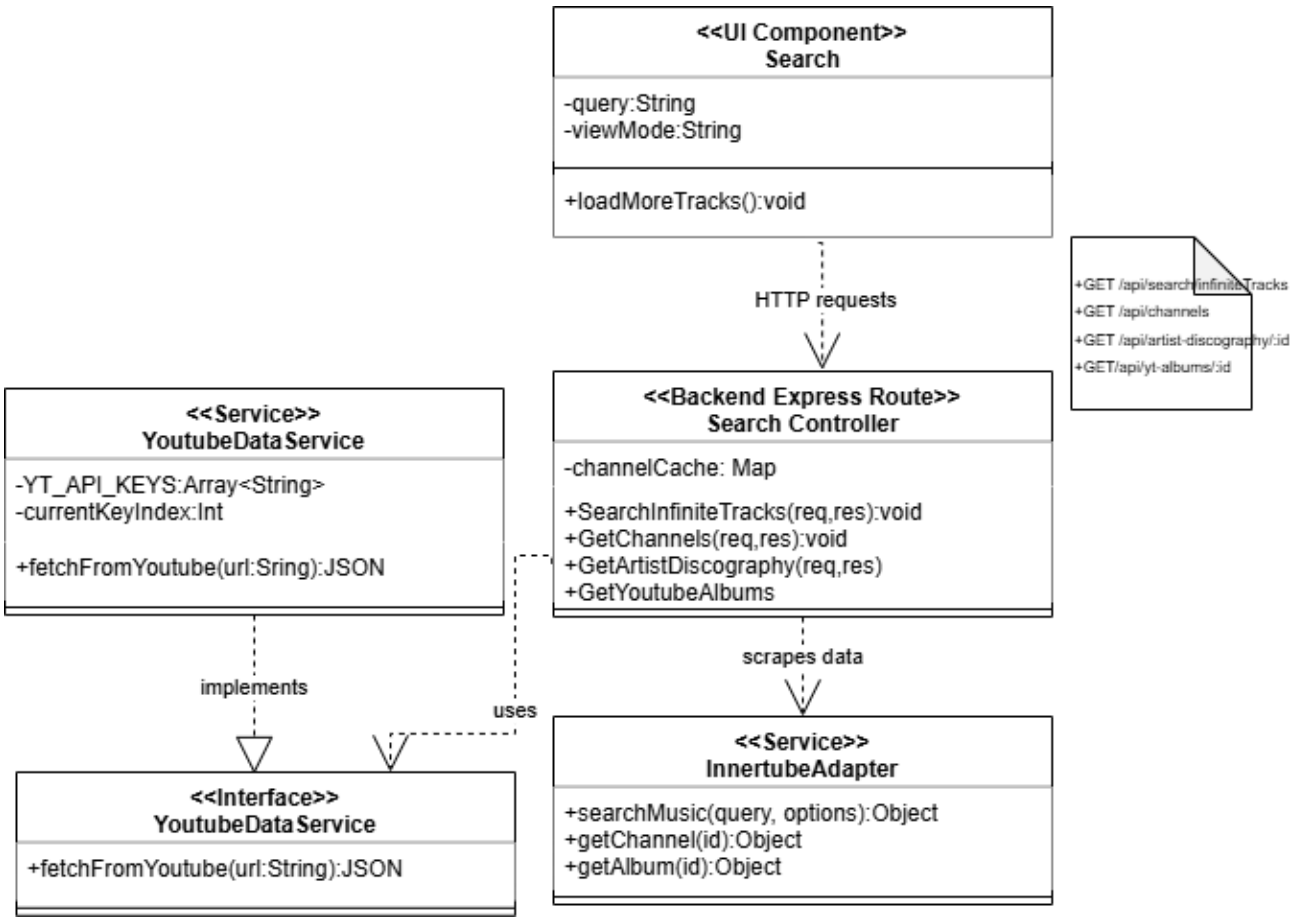


Рис. 2.7 Діаграма кооперації підсистеми пошуку контенту

2.3 Діаграма пакетів

Будь-який великий вебпроєкт потрібно розбивати на незалежні модулі (пакети). Це допомагає організувати структуру для спрощення підтримки та масштабування системи. Діаграма пакетів показує найвищий рівень архітектурної побудови нашої системи.

На рисунку 2.8 видно чотири головні блоки:

1. **Пакет Frontend (Клієнт).**

Пакет виконується у браузері користувача. Написаний на React [3]. Має компоненти сторінок (Pages) та глобальних станів (Contexts). Головний компонент тут – це PlayerContext. Він керує гучністю і постійно спілкується з сервером у фоновому режимі.

2. **Пакет Backend (Сервер).**

Цей пакет реалізує серверну логіку системи на Node.js [4]. Він поділений на маршрутизатори (Routes) та контролери.

Тут лежать наші головні компоненти-файли:

- auth.js – відповідає за безпеку та вхід.
- userProfile.js – оновлює імена, аватарки та історію.
- playlists.js – створює та видаляє збірки музики.
- search.js – відповідає за алгоритм пошуку.

3. **Пакет Databases (Бази даних).**

Цей пакет містить налаштування підключень. Він поділений на дві гілки: models/pg (для PostgreSQL [1]) та models/mongo (для MongoDB [2]). Там лежать схеми таблиць, які ми обговорювали в діаграмі класів.

4. **Пакет External API (Зовнішні сервіси).**

Цей пакет забезпечує взаємодію із зовнішніми сервісами. Тут підключений LRCLib [9] для текстів пісень. Тут живе офіційний клієнт YouTube Data API v3 [6]. Також тут є важливий файл channels.js. У ньому підключена бібліотека youtubei.js. Вона дозволяє витягувати аватарки артистів безпосередньо з API YouTube Music [15]. Цей же компонент має систему кешування. Він зберігає дані

артистів у файл channelCache.json. Термін життя кешу – 12 годин. Для зменшення кількості операцій запису, запис у файл відбувається із затримкою у 300 мілісекунд (механізм debounce). Це дозволяє зменшити навантаження на сервер.

Завдяки такому розділенню системи на фронтенд (React [3]) та бекенд (Node.js [4]) ми отримуємо високу масштабованість. Якщо кількість користувачів різко зросте, ми зможемо виділити більше ресурсів суто для серверної частини, без змін у клієнтській частині системи.

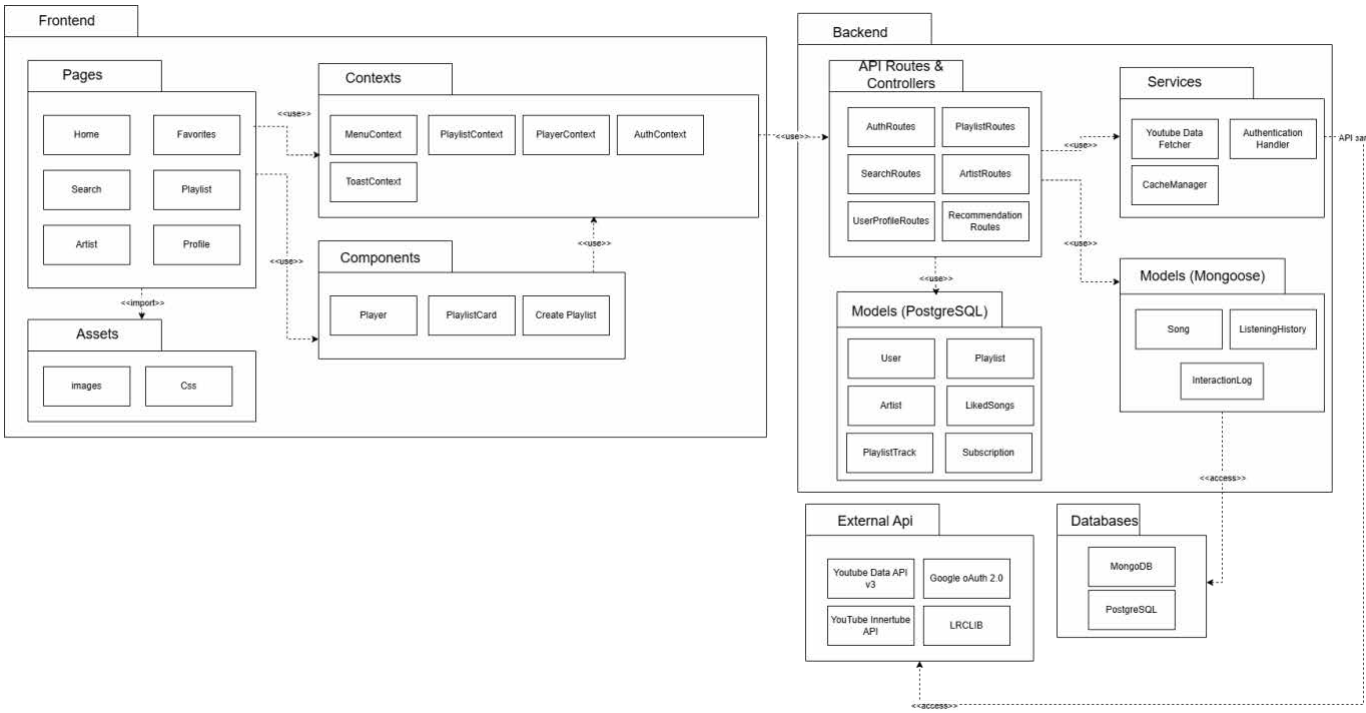


Рис. 2.8 Діаграма взаємодії пакетів та архітектурних модулів системи

2.4 Діаграма компонентів

Якщо діаграма пакетів показує загальну картину, то діаграма компонентів детальніше описує структуру реалізації системи. Вона демонструє, з яких саме робочих файлів та інструментів складається програма.

Згідно з рисунком 2.9, клієнтський додаток побудований на бібліотеці React [3]. Він логічно ділиться на:

Сторінки (Pages): Головна (Home), Пошук (Search), Профіль артиста, Улюблені треки (Favorites), Плейлист.

Контексти (Contexts): Це механізми збереження даних без перезавантаження сторінки. Наприклад, компонент PlayerContext відповідає за плеєр. Основною функцією цього компонента є те що він автоматично відправляє запити на збереження історії прослуховувань у фоновому режимі. Це дозволяє логувати дані, не блокуючи інтерфейс користувача.

UI-Компоненти: Нижня панель плеєра, картки плейлистів, спливаючі форми створення.

Ресурси (Assets): Картинки, іконки та CSS-стилі [10].

Серверна частина (Backend) містить наступні компоненти:

API Маршрути (Routes): AuthRoutes, SearchRoutes, PlaylistRoutes. Вони відповідають за прийом команд від клієнта.

Сервіси (Services): Youtube Data Fetcher відповідає за гібридний пошук музики. CacheManager зберігає тимчасові відповіді, для зменшення кількості запитів до API та оптимізації використання квот.

Моделі баз даних: Окремі файли, що описують структуру таблиць та колекцій (наприклад, Song та ListeningHistory).

Ця компонентна модель є основою для реалізації програмного коду. Вона наочно показує, від чого залежить кожна функція, і як відбувається передача даних від бази даних до екрана користувача.

Діаграма компонентів системи знаходиться у Додатку Б

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка систем потокового відтворення музичного контенту потребує використання сучасних вебтехнологій, ефективної організації баз даних та оптимізації серверної частини. Такі системи повинні забезпечувати швидку обробку запитів, стабільну роботу під навантаженням та мінімальні затримки під час відтворення мультимедіа.

У цьому розділі розглядається архітектура розробленої системи, структура баз даних, використаний технологічний стек та реалізація основних програмних модулів. Система побудована за клієнт-серверною архітектурою з використанням React [3], Node.js [4], PostgreSQL [1] та MongoDB [2]. Для отримання музичного контенту та метаданих використовуються зовнішні API-сервіси.

3.1 Система управління інформаційною базою

У межах розробленої веборієнтованої системи реалізовано гібридну модель зберігання даних, що поєднує реляційну СУБД PostgreSQL [1] та документо-орієнтовану MongoDB [2]. Такий підхід дозволив розділити обробку транзакційних операцій і високонавантажених процесів логування та кешування метаданих.

PostgreSQL [1] використовується як основне сховище структурованих даних системи. У реляційній базі зберігаються облікові записи користувачів, плейлисти, підписки на виконавців, а також зв’язки між користувачами та музичним контентом. Для роботи з PostgreSQL [1] у серверній частині застосовано ORM Sequelize [18], що забезпечує створення моделей, опис асоціацій між таблицями та виконання транзакційних операцій.

MongoDB [2] використовується як допоміжне сховище для роботи зі слабо структурованими даними. У колекціях MongoDB [2] зберігаються кешовані

метадані треків, історія прослуховувань, журнали взаємодій користувачів із системою та кеш головної сторінки застосунку.

Взаємодія між двома СУБД реалізована через спільний ідентифікатор youtubeId, який використовується для синхронізації метаданих треків та інформації про взаємодію користувачів із музичним контентом. Це дозволяє уникнути дублювання великих обсягів даних і забезпечує швидке отримання інформації під час формування відповіді клієнтському застосунку.

3.1.1 Використання реляційної бази даних. Реляційна система управління базами даних PostgreSQL [1] використовується для зберігання основних структурованих сутностей програмної системи. У базі даних реалізовано таблиці користувачів, плейлистів, підписок на виконавців, вподобаних треків та проміжних таблиць для підтримки реляційних зв’язків між сутностями.

Модель користувача User містить унікальний UUID-ідентифікатор, електронну пошту, дані OAuth-авторизації [21] Google та криптографічний хеш пароля. Під час локальної реєстрації пароль користувача обробляється за допомогою бібліотеки bcrypt [8] із генерацією salt-хешу, після чого у базі зберігається лише зашифроване значення passwordHash.

У системі реалізовано реляційні зв’язки типу один-до-багатьох та багато-до-багатьох. Наприклад, один користувач може створювати декілька плейлистів, що реалізовано через асоціації User.hasMany(Playlist) та Playlist.belongsTo(User). Для організації зв’язку між плейлистами та треками використовується проміжна таблиця PlaylistTrack, яка зберігає youtubeId треків та дату їх додавання до плейлиста.

Окремо реалізовано таблицю LikedSong, яка використовується для збереження вподобаних користувачем треків. Під час встановлення позначки “подобається” система перевіряє наявність відповідного запису в PostgreSQL [1] та створює або видаляє його залежно від поточного стану.

Для оптимізації масових операцій у системі використовуються bulkCreate-запити Sequelize [18]. Це дозволяє швидко додавати великі набори треків до плейлистів без значного навантаження на серверну частину.

3.1.2 Використання нереляційної бази даних. Для роботи зі слабо структурованими даними у програмній системі інтегровано документо-орієнтовану базу даних MongoDB [2]. У MongoDB [2] зберігаються метадані музичних треків, історія прослуховувань, журнали взаємодій користувачів та кешовані результати рекомендаційної системи.

Основною колекцією є Song, яка використовується як локальний кеш музичних метаданих. Документ містить youtubeId треку, назву композиції, ім’я автора, channelId каналу, посилання на зображення та тривалість у секундах. Збереження даних у MongoDB [2] дозволяє уникнути повторних звернень до зовнішнього YouTube API [6] та зменшує кількість використаних API-квот.

Для збору статистики прослуховувань використовується колекція ListeningHistory. Кожен запис містить ідентифікатор користувача, youtubeId композиції та час початку відтворення. Під час додавання нового запису система автоматично видаляє попередню появу цього ж треку в історії користувача та переносить його на початок списку останніх прослуханих треків.

Додатково реалізовано колекцію InteractionLog, що використовується для фіксації поведінкових подій користувача. Система реєструє дії типу play, skip, like, unlike та add_to_playlist, а також тривалість прослуховування композиції у секундах. Такі дані надалі використовуються для формування персоналізованих рекомендацій та аналітики активності користувачів.

MongoDB [2] також використовується для кешування головної сторінки користувача. Колекція HomeCache зберігає готові рекомендації, новинки, саундтреки та персоналізовані добірки альбомів, що дозволяє значно зменшити кількість повторних запитів до зовнішніх сервісів.

3.1.3 Комбінований підхід та міжсистемна синхронізація. Однією з основних особливостей архітектури розробленої системи є одночасне використання PostgreSQL [1] та MongoDB [2] у межах єдиного серверного

застосунку. Такий підхід дозволяє розподілити навантаження між двома типами СУБД та використовувати сильні сторони кожної технології.

Ключовим елементом синхронізації між реляційною та нереляційною базами даних виступає глобальний ідентифікатор youtubeId. Саме через нього система пов’язує записи PostgreSQL [1] із відповідними документами MongoDB [2].

Прикладом комбінованої роботи двох СУБД є механізм лайків та плейлистів. Під час додавання треку до улюблених система спочатку виконує операцію upsert у MongoDB-колекції [2] Song для кешування метаданих композиції, після чого створює реляційний запис у таблиці LikedSong PostgreSQL. Аналогічний підхід використовується і під час роботи з плейлистами.

Під час отримання плейлистів клієнтський застосунок використовує механізм крос-системної агрегації. Сервер спочатку отримує список youtubeId із PostgreSQL [1], після чого виконує запит до MongoDB [2] для отримання повних метаданих треків. На основі цих даних формується єдиний JSON-об’єкт [17] відповіді, який передається клієнтській частині застосунку.

Комбінований підхід також застосовується у рекомендаційній системі. Алгоритми аналізують вподобані композиції користувача з PostgreSQL [1] та історію прослуховувань із MongoDB [2], після чого формують персоналізовані рекомендації на основі поведінкових патернів і схожості музичного контенту.

3.2 Розробка інформаційної бази

Розробка структури інформаційної бази потребує аналізу предметної області та реалізації структури бази даних. У процесі розробки необхідно було визначити основні сутності системи, проаналізувавши сутності, які беруть участь у потоковому відтворенні музики, та зв'язки між ними. Зважаючи на використання комбінованої моделі, проєктування було розділено на два напрями: нормалізація реляційної схеми для забезпечення строгих зв'язків та проєктування документо-орієнтованих структур для швидкісної вибірки контенту.

Аналіз предметної області виділив такі ключові сутності: користувач (User), плейлист (Playlist), виконавець (Artist), музичний трек (Song), історія прослуховувань (ListeningHistory) та системний лог взаємодій (InteractionLog).

3.2.1 Проєктування реляційної структури даних. Реляційна частина інформаційної бази була спроєктована з дотриманням третьої нормальної форми (3NF), що гарантує залежність неключових атрибутів виключно від первинного ключа та мінімізує аномалії даних.

Таблиця користувачів: Ця таблиця є основною таблицею реляційної схеми, навколо якої будуються всі персоніфіковані зв'язки. Структура таблиці підтримує як класичну локальну авторизацію, так і федеративну авторизацію (через OAuth-провайдерів [21]).

Таблиця 3.1

Структура таблиці користувачів.

Поле таблиці	**Тип даних**	**Опис**
id	UUID / Integer	Первинний ключ, унікальний системний ідентифікатор.
displayName	String	Ім'я користувача для відображення в інтерфейсі (UI).
email	String	Електронна пошта. Обмеження: UNIQUE, NOT NULL.
passwordHash	String	Криптографічний хеш пароля. Nullable (якщо вхід через Google).
googleId	String	Ідентифікатор OAuth Google [21]. Nullable.
avatar	String	URL-посилання на аватар профілю.

Таблиця плейлистів: Плейлисти являють собою користувацькі музичні збірки. Ця сутність реалізує зв'язок "один-до-багатьох" (один користувач може мати необмежену кількість плейлистів, але плейлист має лише одного власника).

Таблиця 3.2

Структура таблиці плейлистів.

Поле таблиці	**Тип даних**	**Опис**
id	Integer	Первинний ключ плейлиста.
name	String	Користувацька назва музичної збірки.
description	Text	Детальний опис тематики або настрою збірки.
isPublic	Boolean	Статус доступу (true – публічний для глобального пошуку, false – приватний).
coverImage	String	URL обкладинки, сформований на основі треків плейлиста
ownerId	Integer	Зовнішній ключ (Foreign Key), посилається на User.id.

Таблиця виконавців: Таблиця виконавця використовується для збереження інформації про музичних авторів та канали, на які підписаний користувач для персоналізації контенту.

Таблиця 3.3

Структура таблиці виконавців.

Поле таблиці	**Тип даних**	**Опис**
id	UUID / Integer	Первинний ключ таблиці виконавців.
channelId	String	Зовнішній ідентифікатор каналу або артиста на платформі YouTube.
name	String	Назва виконавця або назва музичного каналу.
image	String	URL-посилання на зображення профілю артиста.
subs	String	Кількість підписників у відформатованому текстовому вигляді.

Асоціативні таблиці зв'язків Для реалізації зв’язків типу "багато-до-багатьох" (M:N) та "один-до-багатьох" (1:M) між основними сутностями спроєктовано проміжні таблиці.

Таблиця 3.4

Асоціативні таблиці зв'язків.

Назва таблиці	**Поля таблиці**	**Опис**
PlaylistTrack	playlistId, youtubeId, addedAt	Пов'язує плейлисти з конкретними треками. Поле addedAt (DateTime) використовується у хронологічному сортуванні вмісту.
LikedSong	userId, youtubeId	Фіксує вподобання користувачів. Складається з композитного ключа для уникнення багаторазового додавання дублікатів.
Subscription	userId, artistId, subscribedAt	Проміжна таблиця, яка поєднує користувачів із таблицею виконавців Artist для реалізації механізму підписок.

Проєктування нереляційної структури даних

Нереляційна модель MongoDB [2] побудована за принципом aggregate-oriented design. Колекції оптимізовано для зменшення кількості звернень до сервера (round-trips) під час читання інформації.

Колекція треків Song Ця колекція виконує функцію централізованого кешу метаданих для всієї системи. Замість того, щоб звертатися до лімітованого зовнішнього API при кожному відтворенні, сервер зберігає метадані треків.

Таблиця 3.5

Структура колекції треків.

Поле документа	**Тип даних**	**Опис**
youtubeId	String	Логічний первинний ключ, унікальний ідентифікатор відео.
title	String	Назва музичної композиції.
author	String	Назва каналу або ім'я виконавця треку.
image	String	Резолвлене посилання на найкращу доступну обкладинку.
duration	Number	Тривалість треку в секундах, необхідна для відображення прогресу відтворення у плеєрі

Колекція історії прослуховувань Історія прослуховувань використовується для роботи алгоритмів персоналізації.

Таблиця 3.6

Структура колекції історії прослуховувань.

Поле документа	**Тип даних**	**Опис**
userId	Number	Ідентифікатор користувача (забезпечує логічний зв'язок з PostgreSQL [1]).
song	Object	Вкладений об'єкт (embedded document) з повними метаданими треку.
playedAt	Date	Точна мітка часу відтворення для хронологічного сортування та витіснення старих записів.

Колекція логів взаємодії Ця колекція спроєктована для збереження часових рядів (time-series) подій. Її документи не оновлюються, а лише додаються (append-only), що забезпечує швидкий запис великої кількості подій. Саме ці дані можуть використовуватись для подальшого аналізу поведінки користувачів, здатних розпізнавати патерни поведінки користувачів.

Таблиця 3.7

Структура колекції логів взаємодії.

Поле документа	**Тип даних**	**Опис**
userId	Number	Ідентифікатор користувача, який здійснив взаємодію із системою.
youtubeId	String	Ідентифікатор конкретного треку, з яким відбулася взаємодія.
type	String	Тип події (відтворення, пропуск, встановлення лайка, додавання в плейлист).
listenDuration	Number	Кількісна метрика: реальний час прослуховування у секундах, що відрізняє випадковий клік від повноцінного прослуховування.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

Вибір інструментарію для розробки прикладного програмного забезпечення є важливим етапом, який визначає архітектуру системи, загальну продуктивність, здатність до горизонтального масштабування та рівень складності подальшого супроводу системи. Враховуючи вимоги до системи потокового відтворення, серед яких асинхронна обробка великої кількості I/O операцій, взаємодія з REST API [24], інтеграція різних СУБД та забезпечення інтерактивного користувацького інтерфейсу, було обрано єдиний технологічний стекна базі мови JavaScript.

Застосування єдиної JavaScript-архітектури, де JavaScript використовується як на клієнтському, так і на серверному рівнях, дозволяє розробникам повторно використовувати модулі валідації та логіки, стандартизувати формат обміну даними (JSON [17]) і значно знизити складність підтримки при підтримці кодової бази.

3.3.1 Обґрунтування серверної платформи. Для створення серверного ядра системи було обрано програмну платформу Node.js [4]. На відміну від традиційних багатопотокових серверних моделей наприклад, Java/Spring або PHP/Apache, де кожен мережевий запит породжує окремий потік операційної системи, Node.js [4] базується на рушії V8 та використовує однопотокову подієво-орієнтовану модель з неблокуючим вводом/виводом.

Ця архітектура ідеально підходить для розроблюваної системи. Оскільки стримінг музики та запити до баз даних є I/O-інтенсивними операціями то витрачається час на очікування відповіді від мережі або диска, Node.js [4] не блокує виконання програми під час очікування. Замість цього він передає операції системним механізмам та продовжує обробляти інші запити користувачів, обробляючи результати через механізм зворотних викликів Callbacks/Promises/async-await в циклі подій Event Loop. Це дозволяє серверу

обслуговувати велику кількість одночасних з'єднань із мінімальним споживанням оперативної пам'яті.

В якості каркасу поверх Node.js [4] застосовано веб-фреймворк Express.js [5]. Його мінімалістичний дизайн і гнучка архітектура проміжного програмного забезпечення дозволяють чітко розмежувати етапи обробки HTTP-запиту: парсинг тіла запиту, управління крос-доменними ресурсами (CORS), автентифікацію, бізнес-логіку та обробку помилок. Express [5] використовується для конструювання REST API [24], яке слугує точкою доступу до гібридної інформаційної бази.

3.3.2 Клієнтська архітектура та управління станом. Для реалізації користувацького інтерфейсу (UI) було задіяно бібліотеку React. Створення інтерактивного музичного плеєра вимагає швидкого оновлення інтерфейсу на дії користувача (оновлення прогрес-бара, перемикання станів кнопок, нескінченний скролінг) без перезавантаження сторінки – Single Page Application [20].

React [3] вирішує цю проблему за допомогою архітектури віртуального DOM (Virtual Document Object Model) [25]. При зміні стану плеєра система спочатку будує нове віртуальне представлення інтерфейсу в пам'яті, порівнює його з попередньою версією (процес Reconciliation) і обчислює мінімально необхідний набір змін. Потім ці зміни пакетно застосовуються до реального DOM [25] браузера. Цей підхід мінімізує кількість перерендерів DOM [25], що позитивно впливає на швидкість роботи інтерфейсу. Крім того, компонентна ізоляція дозволяє створювати незалежні модулі, такі як Sidebar, PlayerController, TrackList, які можна повторно використовувати в різних частинах застосунку.

Для управління глобальним станом наприклад, доступність даних про поточний трек для будь-якого компонента незалежно від його глибини у дереві була вирішена за допомогою вбудованого React [3] Context API. Це усунуло необхідність в інтеграції додаткових зовнішніх бібліотек наприклад, Redux та дозволило логічно поділити стан системи на домени:

- Контекст авторизації (перевірка сесій та ролей);
- Контекст плеєра (синхронізація відтворення музики та управління чергою);
- Контекст плейлистів (управління локальними бібліотеками);
- Контекст інтерфейсу (навігація та меню).

3.3.3 Механізми безпеки та зовнішня інтеграція. Для забезпечення безпеки та авторизації за допомогою впровадження спеціалізованих модулів. Бібліотека Passport.js застосовується як middleware для автентифікації. Її архітектура підтримує стратегії як для локального входу, захищеного криптографічним модулем bcryptjs [8], так і для авторизації через Google через протокол OAuth 2.0 [21], що спрощує процес входу для користувача. Захист сесій реалізовано шляхом серіалізації ідентифікаторів у HTTP-Only cookie з підписаними ідентифікаторами сесій, що захищає систему від міжсайтового скриптингу.

Для інтеграції з екосистемою YouTube та отримання музичного контенту застосовано комбінацію інструментів:

1. **YouTube Data API v3** [6]: використовується для формального пошуку, фільтрації відео за тривалістю наприклад, відсіювання коротких відео формату YouTube Shorts та завантаження первинних метаданих. Оскільки офіційний API накладає суворі ліміти на кількість запитів, у бекенді реалізовано автоматичну механізм ротації API-ключів, на данні які уже були збережні до бази даних.
2. **Бібліотека youtubei.js**: неофіційний клієнт (Innertube API), який розширює можливості системи. Він дозволяє без витрачання дорогих квот офіційного API отримувати специфічну музичну статистику з YouTube Music [15] наприклад, кількість щомісячних слухачів та отримувати додаткові метадані та інформацію про контент, що є важливим для забезпечення безперервного відтворення.

Таблиця 3.8

Вибраний технологічний стек для розробки системи.

Компонент системи	**Обрана технологія / Інструмент**	**Основне призначення**
Серверне середовище	Node.js [4]	Асинхронне виконання I/O операцій, мережевий сервер
Веб-фреймворк	Express.js [5]	Маршрутизація REST API [24], middleware
Клієнтський інтерфейс	React (з Context API) [3]	SPA [20], компонентний підхід, Virtual DOM [25]
Авторизація та криптографія	Passport.js, bcryptjs [8]	Хешування паролів, OAuth 2.0 [21], управління сесіями
Зовнішня інтеграція	YouTube Data API v3 [6], youtubei.js	Пошук, отримання інформації про контент, робота з Innertube API

3.4 Алгоритмізація та програмування програмних модулів

Основу логіки системи є комплекс розроблених алгоритмів. Алгоритмізація у контексті даної дипломної роботи передбачає не лише формалізацію логіки обробки даних, але й оптимізацію мережевих запитів, кешування та обробку помилок та відмов. Програмування цих модулів здійснювалося з урахуванням асинхронної моделі Node.js [4] та гібридної архітектури баз даних.

Нижче наведено детальний опис ключових алгоритмів, блок-схеми та фрагменти їх реалізації мовою JavaScript, що забезпечують повноцінне функціонування системи.

Весь код до зазначених нижче алгоритмів знаходиться у додатку А.

3.4.1 Алгоритм авторизації користувача. Алгоритм авторизації забезпечує перевірку облікових даних користувача, хешування паролів та створення безпечної сесії.

Алгоритм:

1. Користувач вводить електронну пошту та пароль у React-компоненті [3].

2. Клієнт формує POST-запит до серверного маршруту авторизації.
3. Сервер виконує нормалізацію електронної пошти за допомогою функцій trim() та toLowerCase().
4. Через ORM Sequelize [18] виконується пошук користувача у PostgreSQL [1].
5. Якщо користувача знайдено, викликається функція bcrypt.compare() [8] для перевірки пароля.
6. Після успішної перевірки Passport.js створює сесію та серіалізує ідентифікатор користувача.
7. Сервер повертає JSON-відповідь [17] із даними користувача та списком улюблених треків.

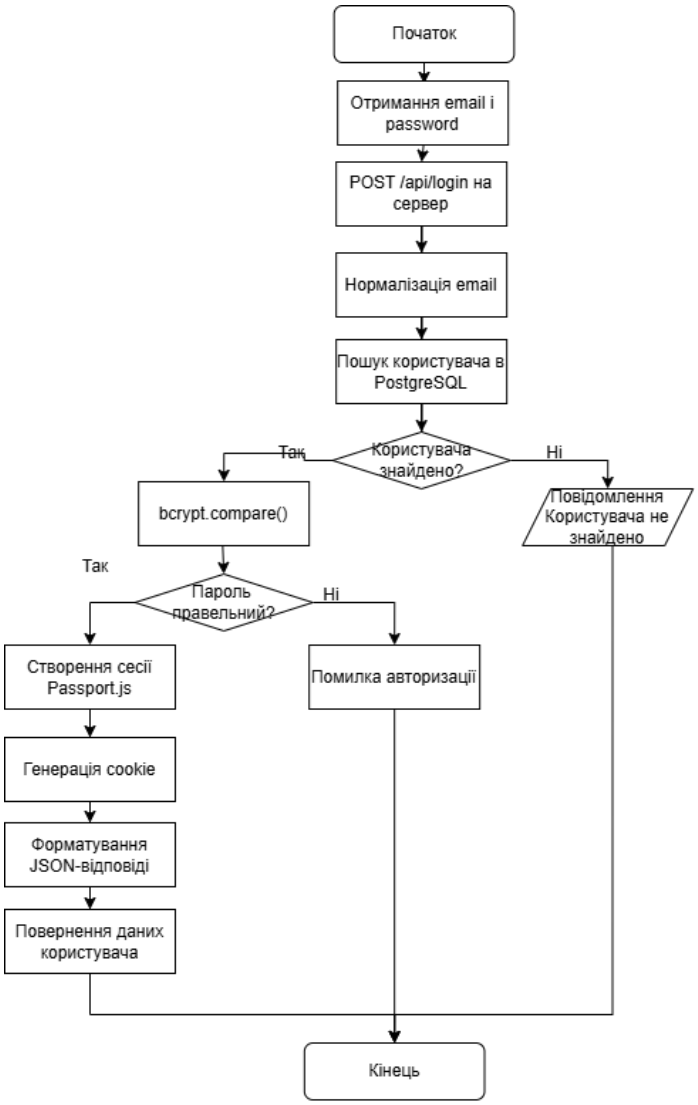


Рис. 3.1 Блок-схема алгоритму авторизації користувача

3.4.2 Гібридний алгоритм пошуку контенту. Пошуковий модуль системи використовує комбінований алгоритм, який поєднує локальний пошук та зовнішній YouTube Data API [6].

Алгоритм:

1. Сервер приймає текстовий пошуковий запит користувача.
2. Формується HTTP-запит до YouTube Data API [6].
3. Із відповіді витягуються videoId знайдених композицій.
4. Додатковим запитом отримуються метадані треків, зокрема тривалість композиції.
5. Виконується нормалізація даних та парсинг формату ISO 8601 [23].
6. Алгоритм відсіює відео тривалістю менше 30 секунд.
7. Підготовлений список треків повертається клієнту.

Таблиця 3.9

Параметри YouTube Data API v3, використані в алгоритмі пошуку.

Параметр YouTube API	**Значення / Логіка використання в алгоритмі**
part	snippet (для базового пошуку) або contentDetails,statistics (для метаданих)
q	Текстовий рядок запиту користувача
type	video (жорстке обмеження для виключення плейлистів та каналів з видачі)
pageToken	Забезпечує алгоритм "нескінченного скролу" (Infinite scroll)

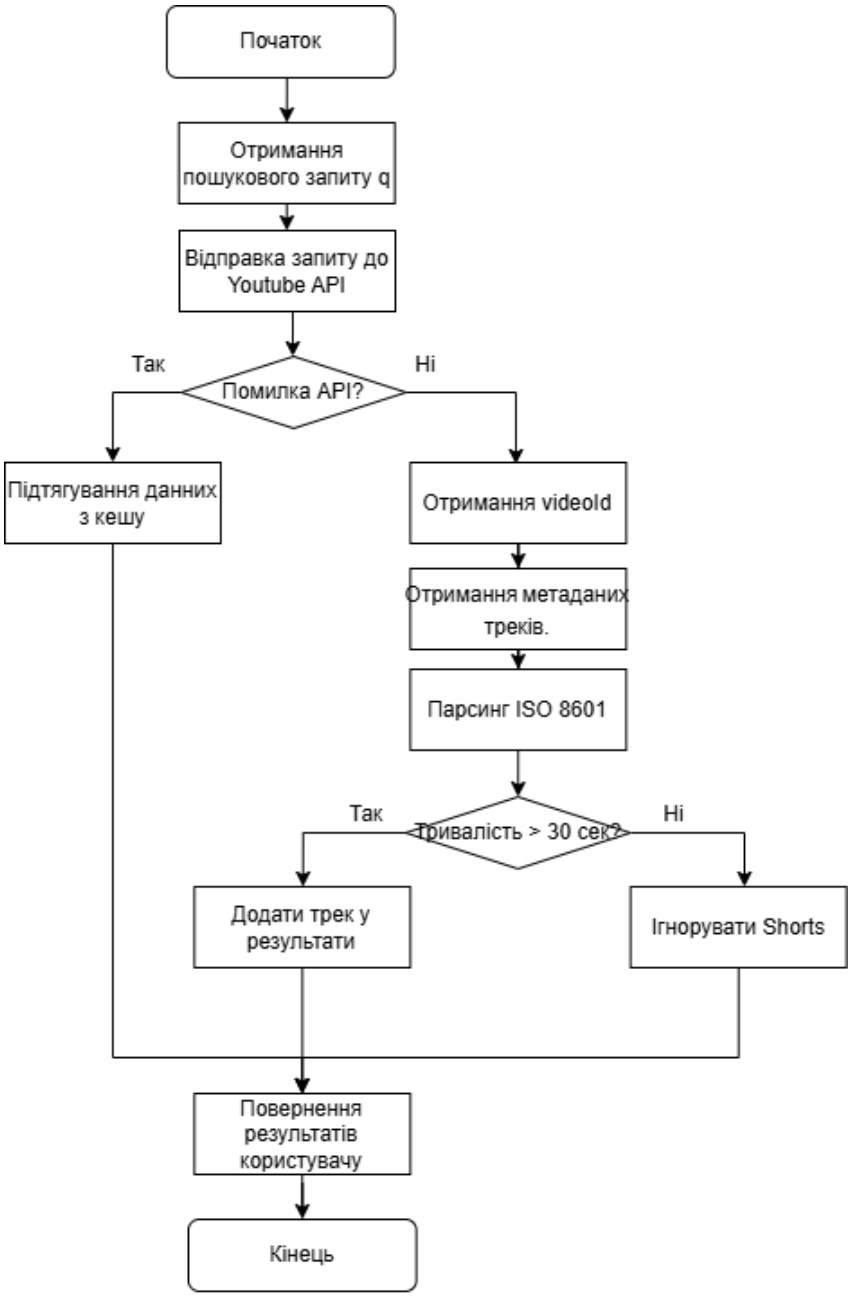


Рис. 3.2 Блок-схема алгоритму гібридного пошуку

3.4.3 Транзакційні алгоритми роботи з плейлистами. Алгоритм роботи з плейлистами забезпечує синхронізацію між PostgreSQL [1] та MongoDB [2] під час створення та оновлення музичних збірок.

Алгоритм:

1. Користувач надсилає запит із назвою плейлиста, описом та початковими треками.
2. Для кожного треку виконується операція findOneAndUpdate з параметром upsert у MongoDB [2].

3. Метадані композицій кешуються локально.
4. Зображення першого треку автоматично встановлюється як обкладинка плейлиста.
5. У PostgreSQL [1] створюється запис плейлиста.
6. Через таблицю PlaylistTrack створюються зв’язки між плейлистом і треками.
7. Після завершення транзакції користувач отримує оновлені дані.

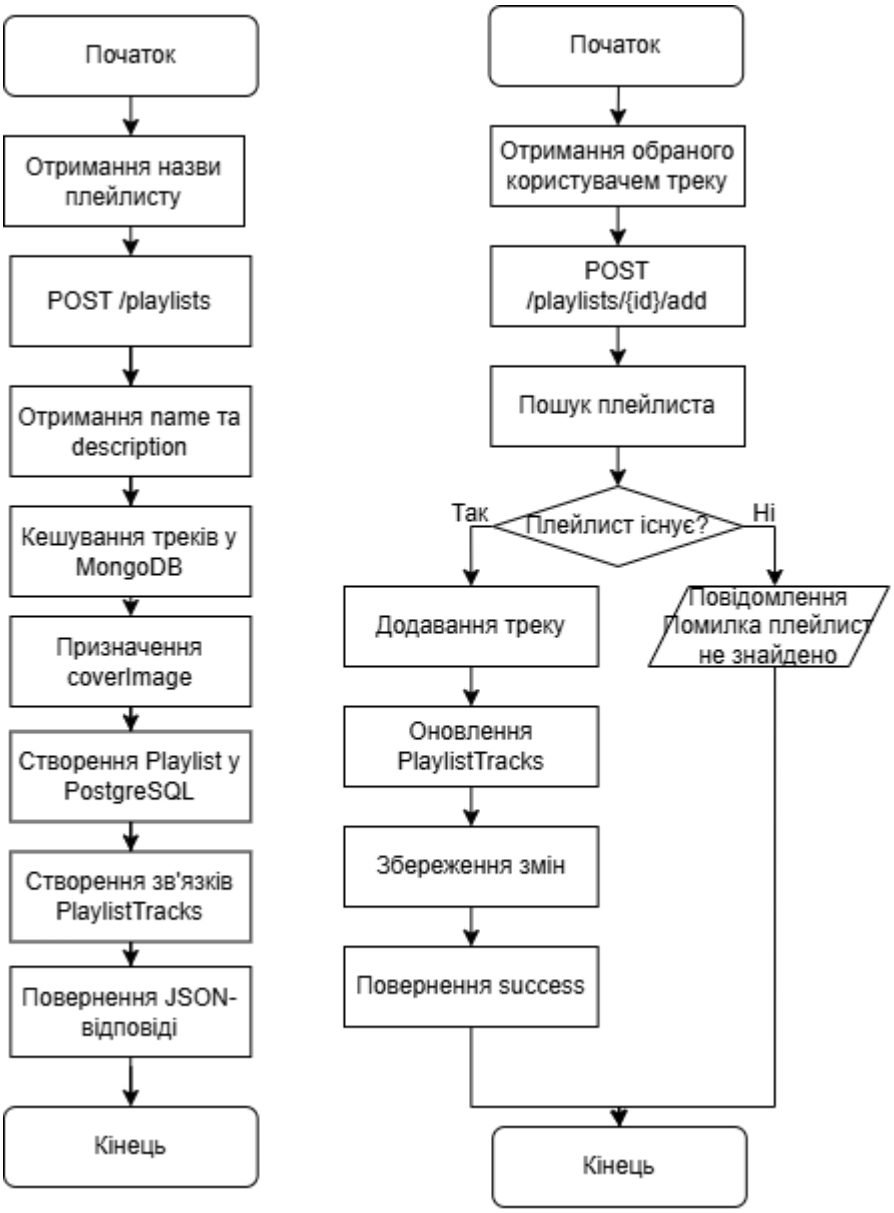


Рис. 3.3 Блок-схеми алгоритму створення та додавання пісні у плейлист

3.4.4 Алгоритм кешування інформації про виконавців. Для зменшення навантаження на зовнішні API та прискорення роботи системи реалізовано алгоритм кешування інформації про музичних виконавців.

Алгоритм:

1. Сервер отримує запит на завантаження сторінки виконавця.
2. Виконується перевірка наявності даних у локальному кеші.
3. Якщо кеш містить актуальні дані, вони повертаються без звернення до зовнішнього API.
4. Якщо запис відсутній або застарів, система виконує новий запит до YouTube API [6].
5. Отримані результати зберігаються у файловому кеші та оперативній пам’яті.
6. Для кожного запису встановлюється час життя TTL.
7. Після завершення TTL запис автоматично вважається неактуальним.

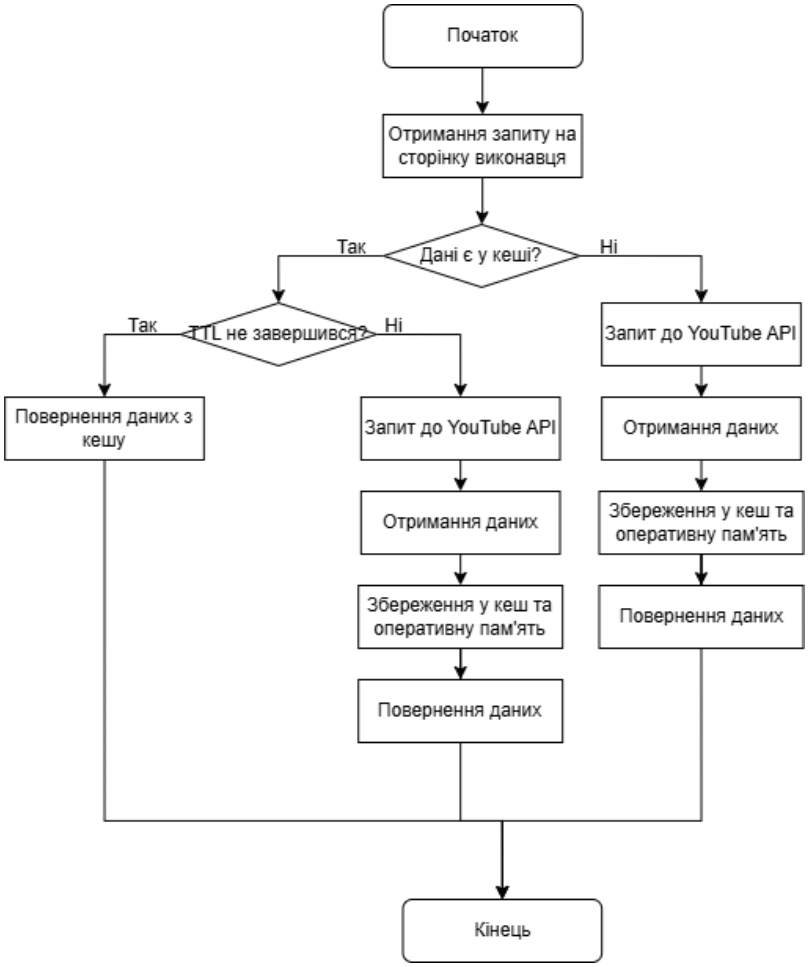


Рис. 3.4 Блок-схема алгоритму кешування інформації про виконавців

3.4.5 Алгоритм формування персональних рекомендацій. Для персоналізації контенту в системі реалізовано алгоритм рекомендацій на основі історії взаємодії користувача.

Алгоритм:

1. Система отримує список вподобаних композицій та історію прослуховувань користувача.
2. Формується початковий набір ідентифікаторів треків.
3. Виконується аналіз виконавців та жанрів композицій.
4. Для кожного виконавця генеруються пошукові запити до YouTube API [6].
5. Отримані результати проходять фільтрацію за релевантністю.
6. Із вибірки видаляються дублікати.
7. Система формує фінальний список персональних рекомендацій.

Блок-схема алгоритму наведена у додатку Б.

3.4.6 Алгоритми обробки поведінкових патернів. Система використовує окремі алгоритми для обробки взаємодії користувача з музичним контентом, зокрема лайків, історії прослуховувань та підписок.

Алгоритм лайків:

1. Користувач натискає кнопку вподобання композиції.
2. Сервер отримує POST-запит із даними треку.
3. Метадані композиції кешуються у MongoDB [2].
4. У таблиці LikedSong виконується перевірка наявності запису.
5. Якщо запис існує — виконується видалення лайка.
6. Якщо запис відсутній — створюється новий лайк.
7. Подія логування записується до колекції InteractionLog.

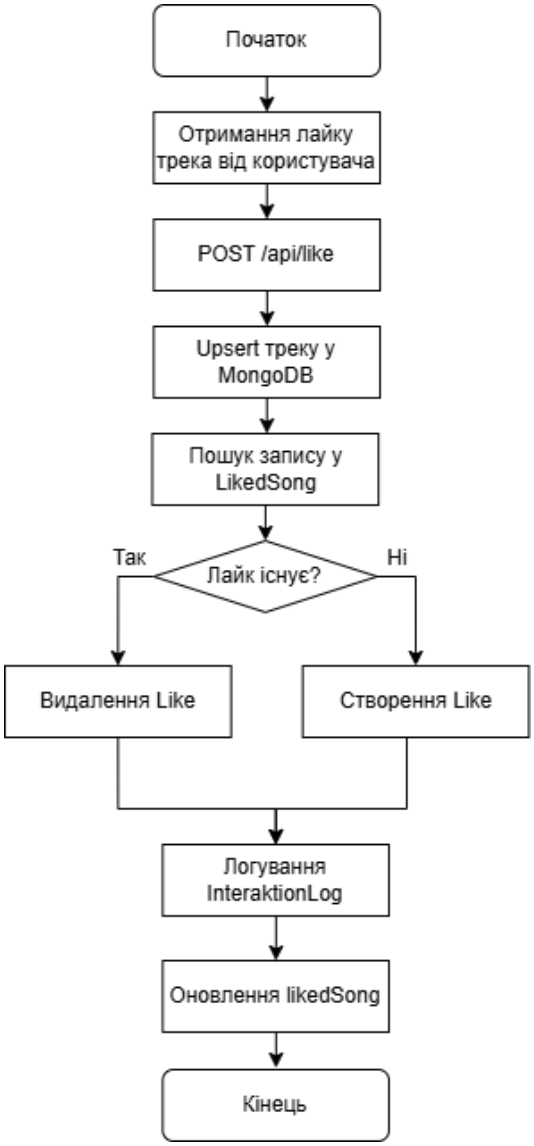


Рис. 3.4 Блок-схеми алгоритму лайків

Алгоритм збереження історії:

1. При запуску відтворення клієнт надсилає інформацію про трек на сервер.
2. Виконується пошук попереднього запису цього треку для поточного користувача.
3. Старий запис видаляється для уникнення дублювання.
4. Створюється новий документ із часовою міткою playedAt.
5. Дані зберігаються у колекції ListeningHistory.

Рис. 3.5 Блок-схема алгоритму запису історії

Алгоритм підписки на виконавця:

1. Користувач натискає кнопку підписки на сторінці виконавця.
2. Сервер отримує POST-запит із ідентифікатором артиста.
3. У PostgreSQL [1] створюється запис про підписку.
4. Інтерфейс автоматично оновлює стан підписки.
5. Користувач отримує актуальну інформацію про виконавця.

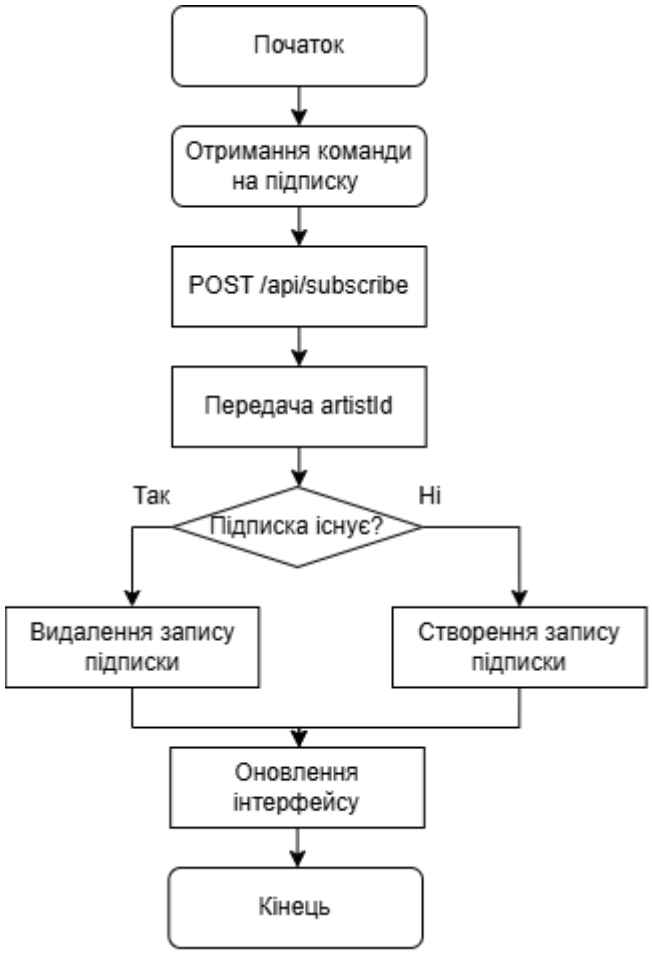


Рис. 3.6 Блок-схема алгоритму запису підписки

3.4.7 Допоміжні клієнтські алгоритми та управління плеєром. На стороні клієнтського інтерфейсу реалізовано набір локальних алгоритмів, що забезпечують швидку взаємодію користувача із системою.

Алгоритм фільтрації та сортування:

1. Отримується масив композицій та пошуковий рядок.
2. Метод filter перевіряє входження тексту у назву або автора композиції.
3. До відфільтрованих результатів застосовується метод sort.
4. Відсортований список передається для рендерингу у React-компонент [3].

Рис. 3.7 Блок-схема алгоритму фільтрації та сортування

Алгоритм управління чергою плеєра:

1. Користувач обирає трек для відтворення.
2. Трек додається у глобальну чергу відтворення.
3. Поточний стан плеєра зберігається у React Context API [3].
4. Внутрішній YouTube-плеєр завантажує відповідне відео.
5. Після завершення композиції система автоматично переходить до наступного треку.
6. При активованому режимі повтору композиція запускається повторно.

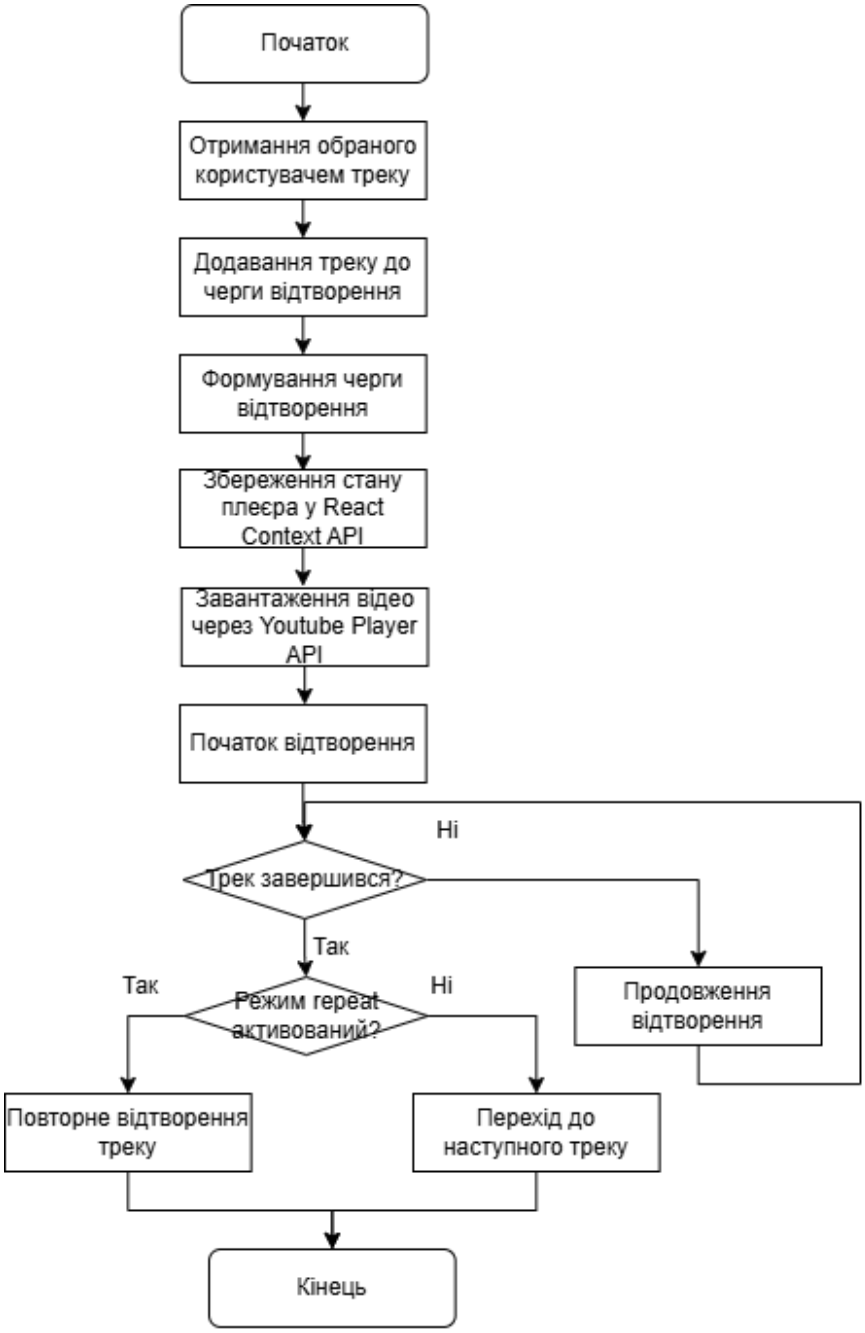


Рис. 3.8 Блок-схема алгоритму управління чергою відтворення

Алгоритм нескінченного підвантаження:

1. Користувач прокручує сторінку до нижньої межі списку.
2. Система перевіряє наявність nextPageToken.
3. Формується асинхронний запит на сервер.
4. Сервер повертає нову порцію треків.
5. Нові елементи об’єднуються з поточним списком.
6. Маркер nextPageToken оновлюється.

Рис. 3.9 Блок-схема алгоритму підвантаження треків

3.4.8 Алгоритм обробки помилок та аварійних ситуацій. Для підвищення стабільності роботи застосунку реалізовано централізований алгоритм обробки помилок.

Алгоритм:

1. Виконується асинхронний запит до сервера або зовнішнього API.
2. У межах конструкції try...catch перевіряється статус відповіді.
3. У випадку помилки створюється виняток throw new Error.
4. Помилка перехоплюється у блоці catch.
5. Система виконує логування помилки.
6. Користувач отримує безпечне повідомлення без аварійного завершення застосунку.
7. Інтерфейс продовжує роботу навіть при частковій недоступності сервісів.

Рис. 3.10 Блок-схема алгоритму обробки помилок

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

Розгортання та впровадження системи є важливим етапом розробки програмного забезпечення. Наша платформа працює на базі сучасного технологічного стеку Node.js [4] та React [3], і використовує одразу дві бази даних: PostgreSQL [1] та MongoDB [2].

Специфіка розробленої веборієнтованої системи обліку та відтворення музичного контенту полягає у використанні архітектури роботи з зовнішніми джерелами даних, де медіафайли не зберігаються локально, а відтворюються динамічно через зовнішні інтерфейси, зокрема YouTube Data API [6]. Крім того, система базується на JavaScript-стеку, що включає MongoDB [2], Express.js [5], React.js, Node.js та PostgreSQL. Подібна гібридна архітектура потребує окремих підходів до тестування, забезпечення інфраструктурних потреб та алгоритмізації процесів розгортання.

Головна мета цього розділу – показати, як ми шукали помилки під час тестування, яке саме залізо нам потрібно для стабільної роботи, і з яких конкретно файлів складається наш готовий проєкт.

4.1 Тестування системи

Тестування програмного забезпечення – процес перевірки розробленої системи з метою виявлення помилок, перевірки відповідності реалізованого функціоналу початковим вимогам та оцінки її поведінки в різних умовах експлуатації. Для багатокомпонентних вебдодатків, що спираються на гібридні бази даних та зовнішні інтеграції, класичного модульного тестування недостатньо.

Процес проведення тестування розробленої системи спирався на інтеграційний підхід та перевірку дотримання визначених кількісних критеріїв, сформованих на етапі системного аналізу відповідно до стандарту ISO/IEC 25010

[22]. Основний акцент тестування був зосереджений на перевірці механізму кешування музичних метаданих, алгоритму гібридного пошуку та продуктивності фонового логування.

Тестування механізму кешування та повторного використання музичних даних

Оскільки система використовує зовнішнє YouTube Data API v3 [6] для отримання музичних метаданих, важливим завданням стало зменшення кількості повторних запитів до зовнішнього сервісу. Для цього було реалізовано механізм локального кешування даних у MongoDB [2]. Після першого отримання інформації про трек його метадані (назва, автор, тривалість, обкладинка та youtubeId) автоматично зберігаються у локальній базі даних та можуть повторно використовуватись без необхідності повторного звернення до API.

Алгоритм роботи передбачає попередню перевірку локального кешу перед виконанням зовнішнього запиту. Якщо необхідні дані вже присутні у базі, система повертає їх без звернення до YouTube API [6], що дозволяє зменшити мережеве навантаження та пришвидшити роботу пошуку.

Таблиця 4.1

Тестові сценарії перевірки кешування та повторного використання музичних даних

Ідентифікатор	**Тип сценарію**	**Опис передумов та початкового стану**	**Очікувана поведінка системи**
ТС-1	Пошук нового треку	Дані про трек відсутні у MongoDB [2]	Сервер виконує запит до YouTube API [6] та зберігає отримані метадані у локальній базі
ТС-2	Повторний пошук	Інформація про трек уже присутня у MongoDB [2]	Система повертає дані з локального кешу без повторного API-запиту
ТС-3	Часткова відсутність даних	У базі відсутня частина метаданих	Сервер виконує оновлення запису та доповнює кеш
ТС-4	Недоступність зовнішнього API	YouTube API [6] тимчасово недоступний	Система використовує раніше збережені локальні дані для відображення результатів пошуку

Тестування алгоритмів гібридного пошуку контенту

Під час тестування гібридного пошуку перевірялися як зовнішньої інтеграції, так і крос-базової синхронізації між PostgreSQL [1] та MongoDB [2]. Крім того, перевірялося виконання функціональних вимог: пагінація не більше 15 записів та відсіювання відео тривалістю менше 30 секунд.

Таблиця 4.2

Тестові сценарії перевірки алгоритмів пошуку

Ідентифікатор	**Тип сценарію**	**Опис передумов та початкового стану**	**Очікувана поведінка системи**
ТС-5	Зовнішній пошук (Double-Fetch)	GET-запит із параметром q="Imagine Dragons" на ендпоінт infiniteTracks.	Система виконує подвійний запит, повертаючи треки, доповнені статистикою переглядів та тривалістю.
ТС-6	Валідація фільтра тривалості	Введення запиту, який у YouTube гарантовано повертає короткі відеокліпи (Shorts).	Модуль фільтрації коректно відкидає всі відео, коротші за 30 секунд.
ТС-7	Перевірка пагінації	Додавання nextPageToken для тестування функції нескінченного скролінгу.	API повертає наступну порцію результатів (до 15 записів), які не дублюють попередні.
ТС-8	Внутрішній пошук (За контентом)	Запит q="Linkin Park". У PostgreSQL [1] є плейлист без цих слів у назві, але треки гурту є всередині нього (в MongoDB [2]).	Гібридний механізм здійснює об'єднання даних із PostgreSQL [1] та MongoDB [2] і успішно повертає релевантний плейлист.

Також проводилося навантажувальне тестування продуктивності. Відповідно до вимоги NFR-01, час пошуку у локальній базі не повинен перевищувати 500 мілісекунд. Завдяки використанню B-Tree індексів у PostgreSQL [1], система успішно пройшла цей тест.

Тестування фонових процесів та телеметрії

Функціональна вимога FR-02 передбачає автоматичний облік кожної секунди прослуховування. Оскільки запис логів відбувається у документо-орієнтовану БД MongoDB [2], основна увага приділялася на перевірці швидкодії та механізмів витіснення даних.

Тестування підтвердило, що асинхронні POST-запити від компонента PlayerContext обробляються сервером за час <= 100 мілісекунд, що повністю відповідає заявленим кількісним критеріям. Операція upsert у MongoDB [2] не блокує Event Loop сервера.

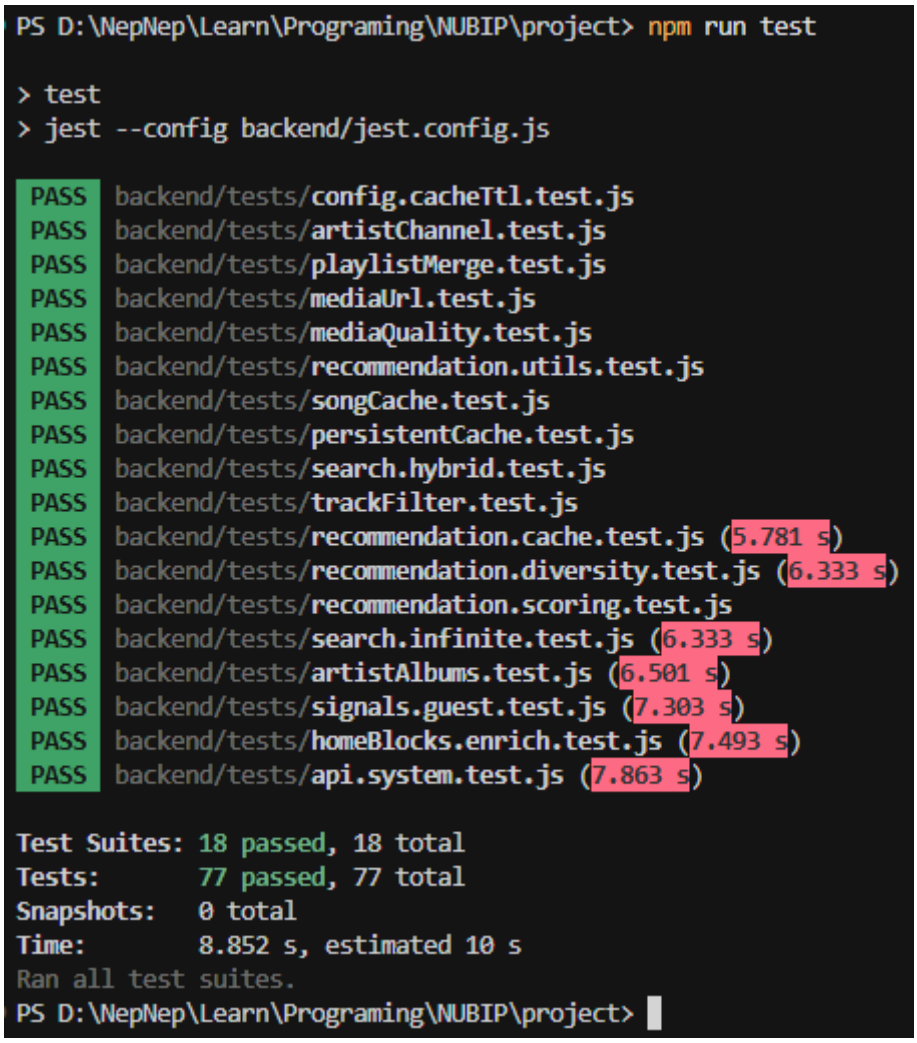


Рис. 4.1 – Результати автоматизованого тестування серверної частини системи

4.2 Вимоги до апаратного та програмного забезпечення

Описуючи вимоги до апаратного та програмного забезпечення гібридної системи, доцільно розпочати з опису топології системи. Саме розміщення описує основні вузли системи та місцезнаходження програмних компонентів відносно цих вузлів, дозволяючи сформувати точні специфікації.

Враховуючи обрану архітектуру на основі React [3], Node.js [4], Express.js [5], PostgreSQL [1] та MongoDB [2], топологія розробленої системи складається з наступних логічних та фізичних вузлів:

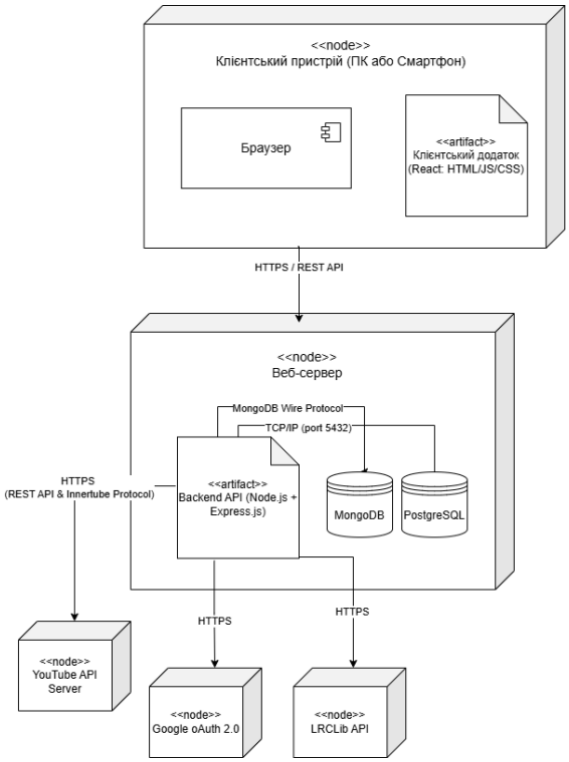


Рисунок 4.2 – Діаграма розгортання системи

1. **Клієнтський вузол (Client Node):** Пристрій кінцевого користувача (смартфон, ПК), на якому виконується браузер із завантаженим React-додатком [3].
2. **Вузол маршрутизації (Web Server Node):** Сервер, який діє як зворотний проксі та віддає статику. Зазвичай це Nginx.
3. **Вузол застосунку (App Server Node):** Серверне середовище виконання Node.js [4] з фреймворком Express.js [5], де реалізована основна бізнес-логіка .
4. **Вузли баз даних (Database Nodes):** Ізольовані середовища для реляційної СУБД (PostgreSQL [1]) та документо-орієнтованої СУБД (MongoDB [2]), які формують рівень управління даними.

Під час реального розгортання вузли 2, 3 та 4 можуть бути розгорнуті як на одному фізичному виділеному сервері, так і розміщені на окремих серверних інстансах для горизонтального масштабування.

Вимоги до клієнтського вузла

Для отримання доступу до музичної стрімінгової платформи користувачу не потрібно встановлювати жодного додаткового ПЗ.

Апаратні вимоги: Мінімальні вимоги включають процесор з тактовою частотою від 1.0 ГГц та щонайменше 1 ГБ оперативної пам'яті (RAM). Пам'ять є критичною через технологію Virtual DOM [25] у React [3], яка вимагає утримання структур дерева елементів. Також обов'язковою є наявність звукової карти та стабільного Інтернет-з'єднання для стабільного відтворення аудіо.

Програмні вимоги: Система є кросплатформною. Головною вимогою є наявність сучасного веб-браузера по типу Google Chrome, Safari, Firefox, що підтримує HTML5, CSS3 [10], ES6+ та Web Audio API. Включення JavaScript у браузері є обов'язковим.

Вимоги до серверних вузлів (Апаратне забезпечення)

Платформа Node.js [4] використовує однопотокову подієво-орієнтовану архітектуру, тоді як бази даних як PostgreSQL [1] вимагають значних обсягів пам'яті для кешування індексів. Тому вимоги до інфраструктури VPS/VDS або виділений сервер є суворими:

Таблиця 4.3

Апаратні вимоги до серверної інфраструктури

Апаратний компонент	**Мінімальні вимоги**	**Рекомендовані вимоги**	**Призначення в топології**
Процесор (vCPU)	1 - 2 ядра	4 ядра і більше	Забезпечення кластеризації процесів Node.js [4] для обробки конкурентних I/O запитів.
Оперативна пам'ять (RAM)	2 ГБ	8 - 16 ГБ	Необхідно для роботи shared_buffers у PostgreSQL [1] та WiredTiger кешу в MongoDB [2].
Дискова підсистема	10-25 ГБ SSD	40+ ГБ NVMe	Оскільки система не зберігає аудіофайли локально, обсяг не є критичним. Важливий високий показник IOPS для запису транзакцій.
Мережевий канал	100 Мбіт/с	1 Гбіт/с	Швидка передача JSON-даних [17] та взаємодія з YouTube Data API [6].

Системне програмне забезпечення серверів

Для забезпечення стабільної роботи згідно з топологією, на серверному вузлі має бути інстальовано наступне системне програмне забезпечення:

1. **Операційна система:** Ubuntu 20.04/22.04 LTS або Debian 12.
2. **Середовище виконання:** Node.js [4] LTS версії 18.x або 20.x разом з менеджером пакетів NPM.
3. **Системи управління базами даних:**
    - **PostgreSQL 14+:** Для транзакційних даних, підтримки ACID та реляційних зв’язків та транзакцій.
    - **MongoDB 6.0+:** Для кешування BSON-документів метаданих та швидкого запису історії.
4. **Вебсервер:** Nginx, що діятиме як зворотний проксі та забезпечуватиме SSL/TLS шифрування за допомогою утиліти Certbot.

5. **Менеджер процесів:** PM2 для демонізації процесу Node.js [4], його кластеризації по ядрах процесора та автоматичного відновлення після збоїв.

4.3 Склад інсталяційного пакету

Оскільки розроблена система є веборієнтованим клієнт-серверним застосунком, класичний інсталяційний пакет для кінцевого користувача у вигляді виконуваного файлу (.exe) відсутній. Для отримання доступу до функціоналу системи користувачу достатньо перейти за URL-адресою вебсервісу через браузер.

Проте з точки зору адміністратора або розробника система складається з набору програмних артефактів, необхідних для розгортання серверної та клієнтської частин платформи.

Інсталяційний пакет програмної системи включає такі компоненти:

1. **Клієнтська частина (Frontend)**

Містить вихідний код React-застосунку [3] та скомпільовану директорію dist, сформовану за допомогою інструмента Vite. До складу входять:

- HTML-файли;
- JavaScript-модулі;
- CSS-стилі [10];
- графічні ресурси (assets);
- React-компоненти (Player, Header, PlaylistCard тощо);
- контексти управління станом (PlayerContext, AuthContext).

2. **Серверна частина (Backend)**

Представлена Node.js-застосунком [4] на базі Express.js [5]. До складу входять:

- файл запуску сервера app.js;
- маршрутизатори API (routes);
- моделі PostgreSQL [1] та MongoDB [2] (models/pg, models/mongo);
- конфігураційні файли (config/database.js);

- кешувальні модулі;
- бізнес-логіка обробки запитів.

3. **Файли залежностей**

Для автоматичного встановлення бібліотек використовуються файли:

- package.json;
- package-lock.json.

Вони містять інформацію про всі необхідні програмні модулі: express[6], mongoose [19], sequelize [18], passport, bcryptjs [8], react [3] та інші.

4. **Файл конфігурації середовища (.env)**

Містить критично важливі змінні середовища:

- URI підключення до PostgreSQL [1];
- URI підключення до MongoDB [2];
- ключі YouTube Data API [6];
- параметри OAuth 2.0 [21];
- секрети сесій та токени.

5. **Бази даних**

Окремою частиною інсталяційного пакету є структури реляційної та нереляційної баз даних:

- таблиці PostgreSQL [1];
- колекції MongoDB [2];
- індекси та зв'язки між сутностями.

Процес розгортання системи включає:

1. Завантаження програмних файлів на сервер;
2. Встановлення залежностей командою npm install;
3. Налаштування змінних середовища;
4. Збірку клієнтської частини командою npm run build;
5. Запуск Node.js-сервера [4] через PM2 або інший менеджер процесів.

Після завершення розгортання система стає доступною через веббраузер та готовою до обробки користувацьких запитів.

ВИСНОВКИ

У результаті виконання бакалаврської кваліфікаційної роботи було розроблено програмне забезпечення веборієнтованої системи обліку та відтворення музичного контенту, яке забезпечує пошук, відтворення та організацію музичних треків, а також збір статистики взаємодії користувачів із системою.

Під час виконання роботи було проведено аналіз предметної області сучасних музичних стрімінгових платформ, визначено основні проблеми та особливості їх функціонування. На основі проведеного аналізу сформовано функціональні та нефункціональні вимоги до програмної системи, а також виконано моделювання предметної області за допомогою UML-діаграм. Це дозволило детально описати структуру системи, сценарії взаємодії користувачів та логіку обробки даних.

У процесі проєктування було реалізовано гібридний підхід до організації інформаційної бази із використанням PostgreSQL [1] та MongoDB [2]. Реляційна база даних використовується для зберігання структурованих даних, зокрема інформації про користувачів, плейлисти та зв’язки між ними, тоді як MongoDB [2] забезпечує ефективне зберігання історії прослуховувань, логів взаємодії та кешованих метаданих музичних треків. Такий підхід дозволив оптимізувати продуктивність системи та розподілити навантаження між різними типами даних.

У межах реалізації серверної частини системи було розроблено механізм гібридного пошуку музичного контенту із використанням YouTube Data API [6] та локальної бази даних. Для забезпечення стабільності роботи системи реалізовано механізм використання локально збережених даних та повторного використання кешованих результатів у випадку недоступності або обмеження зовнішнього сервісу. Крім цього, система підтримує фільтрацію нерелевантного контенту та дозволяє виконувати пошук із мінімальними затримками.

Клієнтська частина програмного забезпечення реалізована з використанням React [3] та побудована за принципом компонентної архітектури.

У системі реалізовано музичний плеєр, підтримку плейлистів, історії прослуховувань, механізми додавання треків до улюблених та адаптивний інтерфейс користувача. Використання React [3] Context дозволило організувати централізоване управління станом плеєра без необхідності перезавантаження сторінок.

Окрему увагу було приділено питанням безпеки та продуктивності. У системі реалізовано хешування паролів за допомогою bcrypt [8], підтримку безпечної авторизації через Google OAuth [21], а також асинхронну обробку логів взаємодії користувачів, що дозволяє уникнути блокування основного потоку виконання серверної частини застосунку.

Проведене тестування підтвердило працездатність розробленої системи та відповідність основним функціональним вимогам. Реалізована архітектура забезпечує можливість подальшого масштабування, інтеграції нових модулів та розширення функціоналу, зокрема впровадження рекомендаційних алгоритмів на основі накопичених даних про поведінку користувачів.

Отримані результати підтверджують, що поставлена мета бакалаврської роботи була досягнута, а розроблене програмне забезпечення може бути використане як основа для подальшого розвитку повноцінної музичної стрімінгової платформи з можливістю масштабування та інтеграції інтелектуальних рекомендаційних механізмів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. PostgreSQL Global Development Group. PostgreSQL Documentation – https://www.postgresql.org/docs/ (дата звернення: 18.09.2025).
2. MongoDB Documentation – https://www.mongodb.com/docs/ (дата звернення: 22.09.2025).
3. React Documentation – https://react.dev/ (дата звернення: 03.10.2025).
4. Node.js Documentation – https://nodejs.org/en/docs (дата звернення: 28.09.2025).
5. Express.js Documentation – https://expressjs.com/ (дата звернення: 01.10.2025).
6. Google Developers. YouTube Data API Documentation – https://developers.google.com/youtube/v3/guides/implementation (дата звернення: 14.10.2025).
7. Passport.js Documentation – https://www.passportjs.org/docs/ (дата звернення: 21.10.2025).
8. bcrypt Documentation – https://www.npmjs.com/package/bcrypt (дата звернення: 21.10.2025).
9. LRCLIB API Documentation – https://lrclib.net/docs (дата звернення: 17.11.2025).
10. W3C Cascading Style Sheets – https://www.w3.org/TR/css-2026/ (дата звернення: 05.09.2025).
11. IFPI Global Music Report 2025 – https://www.ifpi.org/ (дата звернення: 12.03.2026).
12. Закон України «Про авторське право і суміжні права» – https://zakon.rada.gov.ua/laws/show/2811-20#Text (дата звернення: 25.02.2026).
13. Spotify Official Website – https://spotify.com/ (дата звернення: 08.03.2026).
14. SoundCloud Official Website – https://soundcloud.com/ (дата звернення: 08.03.2026).

15. YouTube Music Official Website – https://music.youtube.com/ (дата звернення: 08.03.2026).
16. DEEZER Official Website – https://www.deezer.com/ (дата звернення: 08.03.2026).
17. JSON (JavaScript Object Notation) Specification – https://www.json.org/json-en.html (дата звернення: 20.11.2025).
18. Sequelize ORM Documentation – https://sequelize.org/ (дата звернення: 06.12.2025).
19. Mongoose ODM Documentation – https://mongoosejs.com/docs/ (дата звернення: 08.12.2025).
20. React Router Documentation – https://reactrouter.com/ (дата звернення: 15.01.2026).
21. Google OAuth 2.0 Documentation – https://developers.google.com/identity/protocols/oauth2 (дата звернення: 27.01.2026).
22. ISO/IEC 25010 –System and software quality models https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (дата звернення: 25.02.2026).
23. ISO 8601 –Date and time format standard https://www.iso.org/iso-8601-date-and-time-format.html звернення: 25.02.2026).
24. REST API Architectural Style (Roy Fielding Dissertation) https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (дата звернення: 01.10.2025).
25. DOM (Document Object Model) — MDN Web Docs https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (дата звернення: 12.10.2025).

ДОДАТКИ

ДОДАТОК А

Алгоритм 1. Фрагмент коду алгоритму авторизації користувача (auth.js)

```
//Серіалізація та десеріалізація сесії
passport.serializeUser((user, done) => {
    done(null, user.id);
});
passport.deserializeUser(async (id, done) => {
    try {
        const user = await User.findByPk(id);
        done(null, user);
    } catch (err) {
        done(err, null);
    }
});

//Алгоритм реєстрації з хешуванням
router.post('/api/register', async (req, res) => {
    try {
        const email = req.body.email.trim().toLowerCase();
        const { displayName, password } = req.body;

        let user = await User.findOne({ where: { email } });
        if (user) {
            return res.status(400).json({ error: 'Користувач з таким email вже існує' });
        }

        const salt = await bcrypt.genSalt(10);
        const hashedPassword = await bcrypt.hash(password, salt);

        user = await User.create({ displayName, email, passwordHash: hashedPassword });

        req.login(user, (err) => {
            if (err) return res.status(500).json({ error: 'Помилка авторизації' });
            res.json({ message: 'Реєстрація успішна', user });
        });
    } catch (err) {
        res.status(500).json({ error: 'Помилка сервера' });
```

```
    }
});

//Алгоритм входу
router.post('/api/login', async (req, res) => {
    try {
        const email = req.body.email.trim().toLowerCase();
        const { password } = req.body;

        const user = await User.findOne({
            where: { email },
            include: [{ model: LikedSong, as: 'likedSongs' }]
        });

        if (!user || !user.passwordHash) return res.status(400).json({ error: 'Невірний email або пароль' });

        const isMatch = await bcrypt.compare(password, user.passwordHash);
        if (!isMatch) return res.status(400).json({ error: 'Невірний email або пароль' });

        req.session.userId = user.id;

        const userObj = user.toJSON();
        userObj.likedSongs = userObj.likedSongs ? userObj.likedSongs.map(s => s.youtubeId) : [];
        delete userObj.passwordHash;

        res.json({ message: 'Вхід успішний', user: userObj });
    } catch (err) {
        res.status(500).json({ error: 'Помилка сервера' });
    }
});
```

Алгоритм 2. Фрагмент коду гібридного алгоритму пошуку контенту (search.js)

```
const YT_KEYS = ['KEY_1', 'KEY_2', 'KEY_3', 'KEY_4', 'KEY_5'];
let currentKeyIndex = 0;

async function fetchFromYouTube(baseUrl) {
    let attempts = 0;
    while (attempts < YT_KEYS.length) {
        const key = YT_KEYS[currentKeyIndex];
        const url = `${baseUrl}&key=${key}`;
```

```
        try {
            const response = await fetch(url);
            const data = await response.json();
            if (data.error) {
                currentKeyIndex = (currentKeyIndex + 1) % YT_KEYS.length;
                attempts++;
                continue;
            }
            return data;
        } catch (err) {
            attempts++;
            currentKeyIndex = (currentKeyIndex + 1) % YT_KEYS.length;
        }
    }
    throw new Error(Youtube Api Error');
}

//Алгоритм гібридного пошуку з фільтрацією тривалості
async function runInfiniteTracksSearch({ query = '', channelId = '', pageToken = '' }) {
    let url = `https://www.googleapis.com/youtube/v3/search?part=snippet&type=video&maxResults=30`;
    if (query) url += `&q=${encodeURIComponent(query)}`;
    if (channelId) url += `&channelId=${channelId}&order=viewCount`;
    if (pageToken) url += `&pageToken=${pageToken}`;

    const data = await fetchFromYouTube(url);
    if (!data.items) return { items: [], nextPageToken: null };

    const videoIds = data.items.map(item => item.id.videoId).join(',');
    const durationUrl =
`https://www.googleapis.com/youtube/v3/videos?part=contentDetails,statistics&id=${videoIds}`;
    const durationData = await fetchFromYouTube(durationUrl);

    const videoStatsMap = {};
    if (durationData.items) {
        durationData.items.forEach(video => {
            const match = video.contentDetails.duration.match(/PT(?:(\d+)H)?(?:(\d+)M)?(?:(\d+)S)?/);
            if (match) {
                const hours = parseInt(match[1]) || 0;
                const minutes = parseInt(match[2]) || 0;
                const seconds = parseInt(match[3]) || 0;
                videoStatsMap[video.id] = {
                    duration: hours * 3600 + minutes * 60 + seconds,
```

```
                    views: parseInt(video.statistics.viewCount) || 0
                };
            }
        });
    }

    const results = data.items
        .filter(item => {
            const duration = videoStatsMap[item.id.videoId]?.duration || 0;
            return duration > 30; //Відсіювання відео менше 30 секунд
        })
        .map(item => ({
            title: item.snippet.title,
            author: item.snippet.channelTitle,
            channelId: item.snippet.channelId || '',
            image: `https://i.ytimg.com/vi/${item.id.videoId}/mqdefault.jpg`,
            youtubeId: item.id.videoId,
            duration: videoStatsMap[item.id.videoId]?.duration || 0,
            views: videoStatsMap[item.id.videoId]?.views || 0,
            type: 'video'
        }));

    return { items: results, nextPageToken: data.nextPageToken || null };
}
```

Алгоритм 3. Фрагмент коду транзакційних алгоритмів роботи з плейлистами (playlists.js)

```
//Алгоритм створення плейлиста із синхронізацією баз даних
router.post('/api/playlists', isAuthenticated, async (req, res) => {
    try {
        const { name, description, isPublic, initialTrack } = req.body;
        const userId = req.session.userId || req.user.id;
        let coverImage = "https://via.placeholder.com/200/181818/ffffff?text=Playlist";
        const tracksToProcess = Array.isArray(initialTrack) ? initialTrack : (initialTrack ? [initialTrack] : []);

        //Зберігаємо пісні в MongoDB для кешу
        for (const trackData of tracksToProcess) {
            if (trackData && trackData.youtubeId) {
                await Song.findOneAndUpdate(
                    { youtubeId: trackData.youtubeId },
                    { title: trackData.title, author: trackData.author, image: trackData.image, duration:
```

```
trackData.duration || 0 },
                    { upsert: true, setDefaultsOnInsert: true }
                );
            }
        }

        if (tracksToProcess.length > 0 && tracksToProcess[0].image) coverImage = tracksToProcess[0].image;

        //Створюємо плейлист у PostgreSQL
        const newPlaylist = await Playlist.create({
            name, description, isPublic, ownerId: userId, coverImage
        });

        //Зв'язуємо треки з плейлистом у PostgreSQL
        if (tracksToProcess.length > 0) {
            const trackRecords = tracksToProcess.map(t => ({
                playlistId: newPlaylist.id,
                youtubeId: t.youtubeId
            }));
            await PlaylistTrack.bulkCreate(trackRecords);
        }

        res.status(201).json(newPlaylist);
    } catch (err) {
        res.status(500).json({ error: 'Помилка створення плейлиста' });
    }
});
```

Алгоритм 3. Фрагмент коду алгоритму кешування інформації про виконавців (channels.js)

```
const channelCache = new Map();
const CACHE_FILE_PATH = path.join(__dirname, '..', 'cache', 'channelCache.json');
const CACHE_TTL_MS = 1000 * 60 * 60 * 12; //TTL 12 годин
const now = () => Date.now();

//Отримання даних з кешу
const cacheGet = (key) => {
    const entry = channelCache.get(key);
    if (!entry) return undefined;
    if (entry && typeof entry === 'object' && entry.expiresAt) {
        if (entry.expiresAt <= now()) { //Перевірка TTL
            channelCache.delete(key);
```

```
            return undefined;
        }
        return entry.value;
    }
    return entry;
};

//Запис даних у кеш
const cacheSet = (key, value, ttlMs = CACHE_TTL_MS) => {
    channelCache.set(key, { value, expiresAt: now() + ttlMs });
    persistCacheDebounced(); //Синхронізація з файловою системою
};

//Використання кешування у маршруті
router.get('/api/channels/:id', async (req, res) => {
    const originalId = req.params.id;
    const cacheKey = `channel_v51_${originalId}`;

    const cached = cacheGet(cacheKey);
    if (cached) return res.json(cached); //Повернення з кешу

    try {
        const ytClient = await ytClientPromise;
        //Багато логіки для отримання даних з Youtube

        const result = { name: cleanName, image: image, subs: formatSubs(subs), channelId: realChannelId };
        cacheSet(cacheKey, result); //Збереження нових даних
        res.json(result);
    } catch (err) {
        res.status(500).json({ error: 'Помилка завантаження даних каналу' });
    }
});
```

Алгоритм 5. Фрагмент коду алгоритму формування персональних рекомендацій (recommendations.js)

```
//Аналіз історії та формування набору ідентифікаторів
async function getUserSignals(userId) {
    const [likes, history] = await Promise.all([
        LikedSong.findAll({ where: { userId } }),
        ListeningHistory.find({ userId }).sort({ playedAt: -1 }).limit(50)
    ]);
    const likedIds = likes.map(l => l.youtubeId);
    const playedIds = history.map(h => h.youtubeId);
    return { likedIds, playedIds, seedIds: uniq([...likedIds, ...playedIds]).slice(0, 50) };
}

//Формування рекомендацій на основі контенту
async function ytApiDiscoverFromSeeds(seedSongDocs, excludeSet) {
    const reps = pickRepresentativeSeedDocs(seedSongDocs);
    const scored = new Map();

    for (let i = 0; i < reps.length; i++) {
        const doc = reps[i];
        const author = doc.author;
        const hint = inferGenreHint(doc.title, doc.author);
        const seedCh = doc.channelId || '';
        let items = [];

        //Пошукові запити до YouTube API на основі жанрів та виконавців
        const quoted = `"${author.replace(/"/g, '')}"`;
        const hintQ = hint === 'anime-ost' ? `${quoted} anime ost` : hint === 'vocaloid' ? `${quoted}
vocaloid` : `${quoted} song`;

        const tx = await runInfiniteTracksSearch({ query: hintQ });
        for (const t of tx.items || []) {
            if (!excludeSet.has(t.youtubeId) && relevanceFilter(t, author, seedCh)) {
                items.push(t);
            }
        }

        for (const item of items) {
            if (excludeSet.has(item.youtubeId)) continue;
            scoreAdd(scored, item.youtubeId, 6 - i * 0.32, `yt-search|${normalizeArtistKey(author)}|${hint}`);
        }
```

```
    }

    //Сортування та відсіювання
    return Array.from(scored.entries())
        .sort((a, b) => b[1].score - a[1].score)
        .slice(0, 120)
        .map(([youtubeId, meta]) => ({ youtubeId, score: meta.score, reasons: meta.reasons }));
}
```

Алгоритм 6. Фрагмент коду алгоритмів обробки поведінкових патернів.

```
//Алгоритм обробки лайків (userProfile.js)
router.post('/api/like', isAuthenticated, async (req, res) => {
    const { song } = req.body;
    try {
        await Song.findOneAndUpdate(
            { youtubeId: song.youtubeId },
            { title: song.title, author: song.author, image: song.image, duration: song.duration || 0 },
            { upsert: true, setDefaultsOnInsert: true }
        );

        const userId = req.session.userId || req.user.id;
        const existingLike = await LikedSong.findOne({ where: { userId, youtubeId: song.youtubeId } });

        if (existingLike) {
            await existingLike.destroy();
            await InteractionLog.create({ userId: String(userId), youtubeId: song.youtubeId, action: 'unlike' });
        } else {
            await LikedSong.create({ userId, youtubeId: song.youtubeId });
            await InteractionLog.create({ userId: String(userId), youtubeId: song.youtubeId, action: 'like' });
        }
        const allLikes = await LikedSong.findAll({ where: { userId } });
        res.json({ likedSongs: allLikes.map(l => l.youtubeId) });
    } catch (err) { res.status(500).json({ error: 'Помилка' }); }
});

//Алгоритм збереження історії (userProfile.js)
router.post('/api/history', isAuthenticated, async (req, res) => {
    const { song } = req.body;
    try {
        const userId = String(req.session.userId || req.user.id);
```

```
        //Видалення старого запису цієї ж пісні для уникнення дублювання
        await ListeningHistory.deleteMany({ userId, youtubeId: song.youtubeId });

        //Створення нового запису
        await ListeningHistory.create({ userId, youtubeId: song.youtubeId, playedAt: new Date() });
        res.json({ message: 'Додано до історії' });
    } catch (err) { res.status(500).json({ error: 'Помилка' }); }
});

//Алгоритм підписки на виконавця (channels.js)
router.post('/api/subscribe', isAuthenticated, async (req, res) => {
    const { artist } = req.body;
    try {
        const userId = req.session.userId || req.user.id;
        const user = await User.findByPk(userId);

        let [dbArtist] = await Artist.findOrCreate({
            where: { channelId: artist.channelId },
            defaults: { name: artist.name, image: artist.image }
        });

        if (await user.hasSubscribedArtist(dbArtist)) await user.removeSubscribedArtist(dbArtist);
        else await user.addSubscribedArtist(dbArtist);

        const updatedUser = await User.findByPk(userId, { include: [{ model: Artist, as:
'subscribedArtists' }] });
        res.json({ subscribedArtists: updatedUser.subscribedArtists.map(a => ({ channelId: a.channelId })) });
    } catch (err) { res.status(500).json({ error: 'Помилка' }); }
});
```

Алгоритм 7. Фрагмент коду допоміжних клієнтських алгоритмів та управління плеєром.

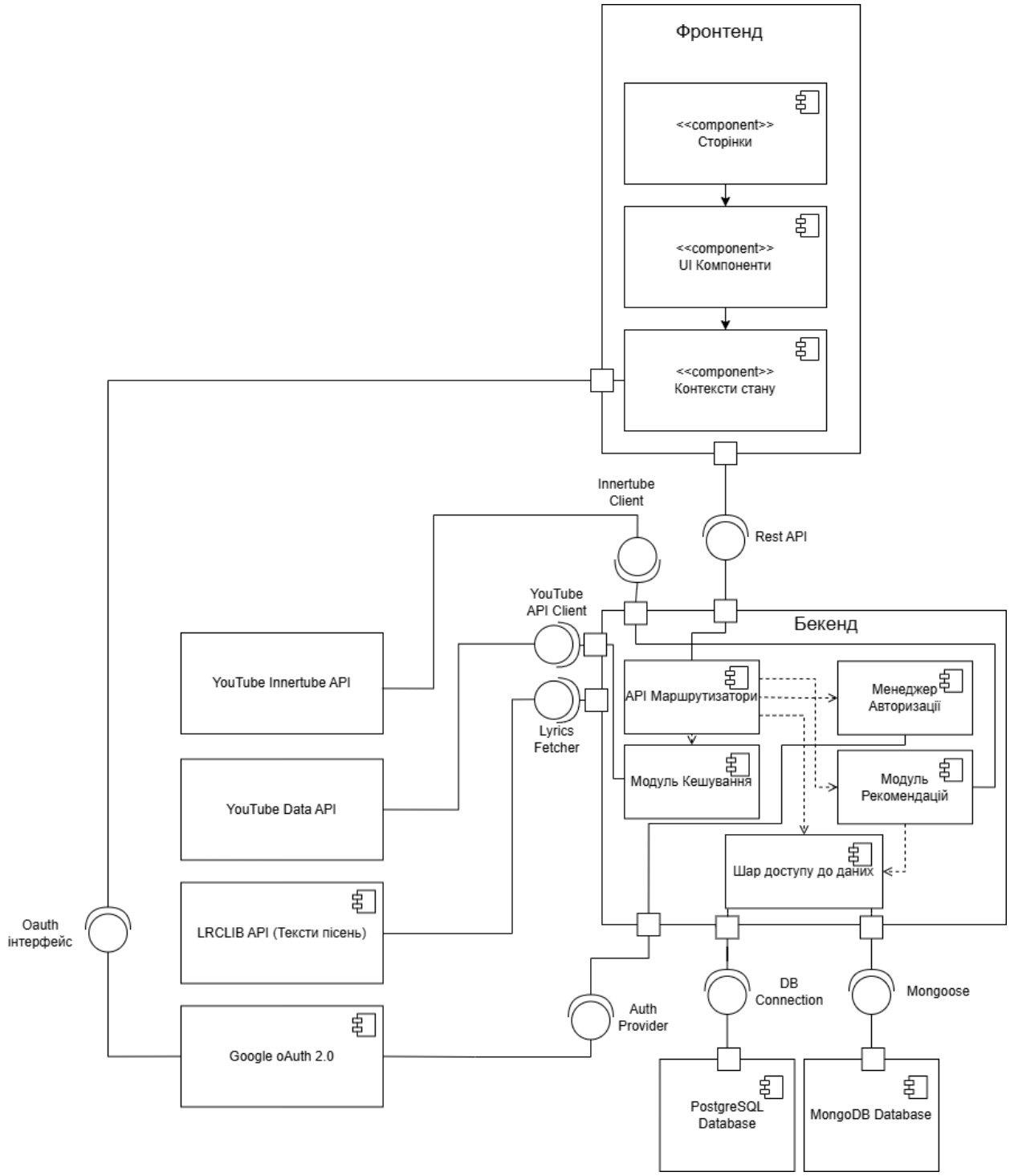
```
//Алгоритм фільтрації та сортування (Favorites.jsx)
const sortData = (data, config) => {
    if (!config.key) return data;
    return [...data].sort((a, b) => {
        let aVal = a[config.key] || '';
        let bVal = b[config.key] || '';

        if (config.key === 'addedAt' || config.key === 'playedAt') {
            aVal = new Date(aVal).getTime();
```

```
            bVal = new Date(bVal).getTime();
        } else if (config.key === 'duration') {
            aVal = Number(aVal);
            bVal = Number(bVal);
        } else {
            aVal = aVal.toString().toLowerCase();
            bVal = bVal.toString().toLowerCase();
        }

        if (aVal < bVal) return config.direction === 'asc' ? -1 : 1;
        if (aVal > bVal) return config.direction === 'asc' ? 1 : -1;
        return 0;
    });
};

const filteredTracks = likedTracks.filter(track =>
    track.title.toLowerCase().includes(searchQuery.toLowerCase()) ||
    track.author.toLowerCase().includes(searchQuery.toLowerCase())
);
const sortedTracks = sortData(filteredTracks, sortConfigTracks);

//Алгоритм управління чергою плеєра (PlayerContext.jsx)
const handlePlayerStateChange = (event) => {
    const YT = window.YT;
    if (event.data === YT.PlayerState.PLAYING) {
        setIsPlaying(true);
        setDuration(playerRef.current.getDuration());
    } else if (event.data === YT.PlayerState.ENDED) {
        if (isRepeatingRef.current && playerRef.current) {
            playerRef.current.seekTo(0);
            playerRef.current.playVideo();
        } else {
            playNext();
        }
    }
};

const playNext = () => {
    const nextIdx = currentIndexRef.current + 1;
    if (nextIdx < queueRef.current.length) {
        currentIndexRef.current = nextIdx;
        setCurrentIndex(nextIdx);
```

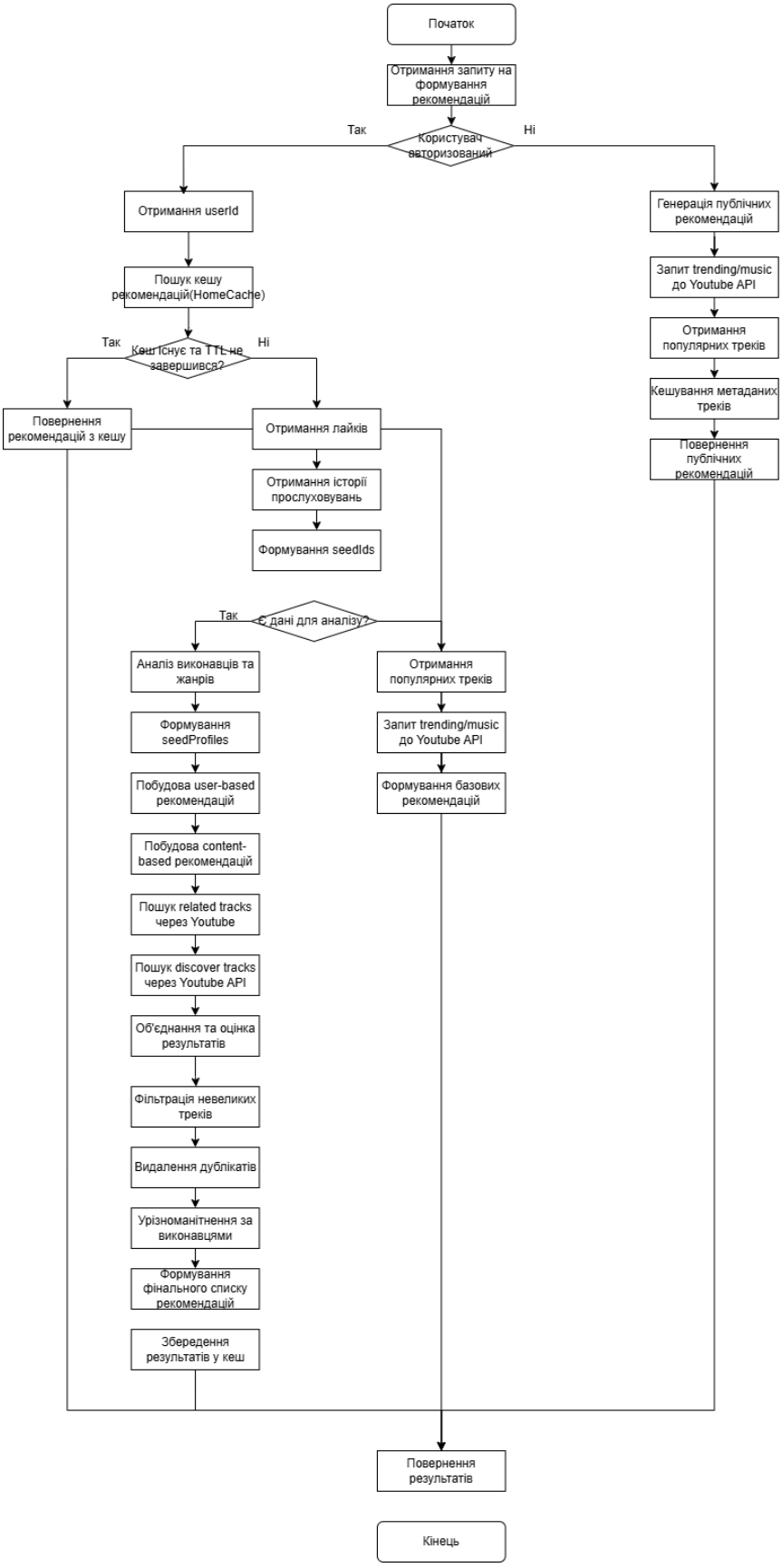
```
        const nextSong = queueRef.current[nextIdx];
        setCurrentSong(nextSong);
        setIsPlaying(true);
        saveToHistory(nextSong);
        if (playerRef.current) playerRef.current.loadVideoById(nextSong.youtubeId);
    }
};

//Алгоритм нескінченного підвантаження (Search.jsx)
const loadMoreTracks = async () => {
    if (isFetchingMore || !nextPageToken) return;
    setIsFetchingMore(true);
    try {
        const res = await
fetch(`/api/search/infiniteTracks?q=${encodeURIComponent(query)}&pageToken=${nextPageToken}`);
        const data = await res.json();

        setTracks(prev => {
            const existingIds = new Set(prev.map(t => t.youtubeId));
            const newUniqueTracks = (data.items || []).filter(t => !existingIds.has(t.youtubeId));
            return [...prev, ...newUniqueTracks]; // Об'єднання масивів
        });
        setNextPageToken(data.nextPageToken || null); //Оновлення маркера
    } catch (e) { console.error(e); } finally { setIsFetchingMore(false); }
};
```

ДОДАТОК Б

Діаграма компонентів системи


Фронтенд
<<component>>
Сторінки
<<component>>
UI Компоненти
<<component>>
Контексти стану
Innertube
Client
Rest API
YouTube
API Client
Бекенд
YouTube Innertube API
YouTube Data API
LRCLIB API (Тексти пісень)
Google oAuth 2.0
Lyrics
Fetcher
API Маршрутизатори
Менеджер
Авторизації
Модуль Кешування
Модуль
Рекомендацій
Шар доступу до даних
Oauth
інтерфейс
Auth
Provider
DB
Connection
Mongoose
PostgreSQL
Database
MongoDB Database

Блоксхема алгоритму формування персональних рекомендацій



ДОДАТОК В

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

Я, Артюхов Мирослав Юрійович, здобувач(ка) НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота на тему «Програмне забезпечення веборієнтованої системи для обліку та відтворення музичного контенту» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що (обрати необхідне):

- [✓] інструменти генеративного ШІ не використовувалися.
- [✓] інструменти ШІ (вказати назву: ChatGPT, Gemini, Claude, DeepL)

. ChatGPT, Gemini були використані для:

- [✓] перекладу іншомовних джерел;
- [✓] стилістичного редагування та перевірки граматики;
- [ ] допомоги у написанні/відлагодженні програмного коду;
- [ ] інше: _____.

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

«___»_____ 2026 р. _____ **Артюхов Мирослав**

(підпис)