НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ _Михайло НОВОТАРСЬКИЙ_
(підпис)

“ 9 ” червня 2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою

“Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Багатокористувацька гра-вікторина з використанням мікросервісної архітектури

Виконав : студент 4 курсу, групи ІО-13
(шифр групи)

_____Полтавський Володимир Дмитрович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент Русінов В.В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант ___ (нормоконтроль) асистент, Пономаренко А. М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____(доцент) Шимкович В. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“ ”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“ 9 ” червня 2025 р.

ЗАВДАННЯ

на бакалаврський дипломний проєкт студента

Полтавського Володимира Дмитровича

1. Тема проєкту Багатокористувацька гра-вікторина з використанням мікросервісної архітектури керівника проєкту Русінов Володимир
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
Володимирович, ассистент, затверджені наказом по університету від 11 травня 2021 року
2. Термін здачі студентом закінченого проєкту 9 червня 2025 р..
3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
   Розділ 1. Аналіз підходів до реалізації застосунків з real-time взаємодією.
   Розділ 2. Проєктування та вибір технологій.
   Розділ 3. Реалізація.

Розділ 4. Тестування та подальші дії.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) схема архітектури системи, схема баз даних та брокерів повідомлень.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Пономаренко А. М.		
Рецензія	Шимкович В. М.		

7. Дата видачі завдання « 24 » січня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*25.10.2024- 26.10.2024*	
2.	*Вивчення та аналіз завдання*	*13.04.2025- 20.04.2025*	
3.	*Розробка архітектури та загальної структури системи*	*21.04.2025- 27.04.2025*	
4.	*Розробка структур окремих підсистем*	*28.04.2025- 02.05.2025*	
5.	*Програмна реалізація системи*	*03.05.2025- 30.05.2025*	
6.	*Оформлення пояснювальної записки*	*11.05.2025- 01.06.2025*	
7.	*Захист програмного продукту*	*18.05.2025*	
8.	*Передзахист*	*02.06.2025*	
9.	*Захист*	*16.06.2025*	

Студент-дипломник _____ Володимир ПОЛТАВСЬКИЙ
(підпис)

Студент-дипломник _____ Володимир РУСІНОВ
(підпис)

АНОТАЦІЯ

У цій роботі проведено розробку веб-застосунку з real-time взаємодією - багатокористувацької гри-вікторини на базі музики.

Спершу було проаналізовано вже аналогічні системи: використані технології та підходи до розробки, переваги та недоліки певних рішень та методів.

Також були досліджені різні ресурси для інтеграції медіатеки. Зібрано ключову інформацію: політику використання, можливості API, ліміти та ціни, якщо вони наявні.

Результати аналізу дозволили обґрунтовано обрати відповідні технології, архітектуру та зовнішні API. У результаті, розроблена гра-вікторина що дає можливість позмагатися за статус кращого меломана. Весь процес від початку проєктування до етапів тестування та розгортання описано у цій роботі, кожен вибір аргументований. Також описано деталі реалізації системи, основні аспекти написання коду та проблеми, що виникли під час цього процесу.

Остаточна версія продукту є стабільною та готовою для безпечної експлуатації: вона протестована за допомогою різних методів. Застосунок легко масштабується горизонтально під різні навантаження.

Ключові слова: WebSocket, real-time взаємодія, мікросервісна архітектура, RabbitMQ, Elasticsearch, Spotify API, музична вікторина, ігрові сесії, асинхронна комунікація, веб-застосунок, горизонтальне масштабування, TypeScript.

ANNOTATION

This paper demonstrate a developing of web application with real-time interaction - a multiplayer music quiz game.

At first, similar existing systems were analyzed: the technologies and development approaches used, as well as the advantages and disadvantages of certain solutions and methods.

Various resources for media library integration were also studied. Key information was gathered, including usage policies, API capabilities, limitations, and pricing where available.

The results of the analysis made it possible to reasonably select appropriate technologies, architecture, and external APIs. As a result, a music quiz game was developed that allows users to compete for the title of top music expert. The entire process - from initial design to testing and deployment - is described in this work, and every decision is supported with reasoning. The implementation details, key aspects of the code, and challenges faced during development are also discussed.

The final version of the product is stable and ready for secure operation; it has been tested using various methods. The application can be easily horizontally scaled to handle different levels of load.

Keywords: WebSocket, real-time interaction, microservice architecture, RabbitMQ, Elasticsearch, Spotify API, music quiz, game sessions, asynchronous communication, web application, horizontal scaling, TypeScript

ОПИС АЛЬБОМУ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Багатокористувацька гра-вікторина з використанням мікросервісної архітектури»

Київ – 2025

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Багатокористувацька гра-	4		
			вікторина з використанням			
			мікросервісної архітектури			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Багатокористувацька гра-	101		
			вікторина з використанням			
			мікросервісної архітектури			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Багатокористувацька гра-	1		
			вікторина з використанням			
			мікросервісної архітектури			
			Схема архітектури системи			
	А4	***ІАЛЦ.467200.005 Д2***	Багатокористувацька гра-	1		
			вікторина з використанням			
			мікросервісної архітектури			
			Схема RabbitMQ			
	А4	***ІАЛЦ.467200.006 Д3***	Багатокористувацька гра-	4		
			вікторина з використанням			
			мікросервісної архітектури			
			Схематика бази даних			
	А4	***ІАЛЦ.467200.007 Д4***	Багатокористувацька гра-	5		
			вікторина з використанням			
			мікросервісної архітектури			
			Мапінг ElasticSearch			
	А4	***ІАЛЦ.467200.008 Д5***	Багатокористувацька гра-	20		
			вікторина з використанням			
			мікросервісної архітектури			
			Лістинг коду			

					ІАЛЦ.467200.001 ОА
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	
Розроб		Полтавський В. Д.			*Багатокористувацька гра-вікторина з використанням мікросервісної архітектури* Опис альбому
Перев		Русінов В.В.			

Літ.	Аркуш	Аркушів
	1	1

НТУУ "КПІ" ФІОТ ІО-13

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Багатокористувацька гра-вікторина з використанням мікросервісної архітектури*»

Київ – 202

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку багатокористувацької гри-вікторини з розпізнаванням музичних треків за допомогою real-time взаємодії та її подальшу підтримку.

Область застосування даного застосунку – розважальні онлайн-платформи, де користувачі можуть перевірити знання музики, змагатися з іншими гравцями. Система може використовуватися як самостійний сервіс, так і інтегруватися в ширші платформи для тематичних ігор.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки багатокористувацької гри-вікторини з real-time взаємодією є завдання на виконання кваліфікаційної роботи освітнього ступеня «бакалавр» спеціальності «Комп’ютерна інженерія», затверджене відповідною кафедрою факультету вищого навчального закладу.

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою цієї роботи є створення масштабованого, стабільного та інтерактивного веб-застосунку, що забезпечує реальновчасну взаємодію користувачів у межах ігрових сесій, побудованих на змаганнях у впізнаванні музичних треків. Призначенням системи є забезпечення якісного геймінгового досвіду через використання сучасної мікросервісної архітектури, асинхронного обміну повідомленнями та API музичних медіатек.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерела для розробки приведеної системи є документація використаних технологій (Symfony, MongoDB, Spotify API, Google Data APIs, тощо), науково-технічна література, статі з програмної інженерії (Medium, Dou).

5 ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Надання REST та WebSocket API з повною документацією.
- Можливість створення ігрових сесій для будь-яких користувачів.
- Імпорт плейлистів користувачів із сервісу Spotify.
- Механізм періодичного оновлення бази з принаймні 10 000 найпопулярніших артистів.
- Можливість масштабування кожного мікросервісу незалежно.

5.2 Вимоги до програмного забезпечення

- ОС Windows (з WSL2), Linux або MacOS.
- Редактор коду з підтримкою LSP для PHP, JavaScript, TypeScript.
- Docker Engine + Docker Compose.

5.3 Вимоги до апаратної частини

- ЦП не менше AMD Ryzen 5 1600.
- ROM не менше 64 ГБ.
- RAM не менше 8 ГБ.
- Доступ до стабільного інтернет-з’єднання для використання сторонніх API.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	25.10.2024 - 26.10.2024
Вивчення та аналіз завдання	13.04.2025 - 20.04.2025
Розробка архітектури та загальної структури системи	21.04.2025 - 27.04.2025
Розробка структур окремих частин системи	28.04.2025 - 02.05.2025
Програмна реалізація системи	03.05.2025 - 30.05.2025
Виправлення помилок	11.05.2025 - 01.06.2025
Оформлення пояснювальної записки	25.10.2024 - 26.10.2024

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Багатокористувацька гра-вікторина з використанням мікросервісної архітектури*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API (Application Programming Interface) — інтерфейс прикладного програмування, який дозволяє програмам взаємодіяти між собою.

AMQP (Advanced Message Queuing Protocol) — розширений протокол черг повідомлень, який визначає стандарти обміну повідомленнями між сервісами.

CDN (Content Delivery Network) — мережа доставки контенту, що забезпечує швидку роздачу статичних ресурсів користувачам.

DDD (Domain-Driven Design) — проєктування, орієнтоване на предметну область, що дозволяє чітко відокремити бізнес-логіку в системі.

FPM (FastCGI Process Manager) — менеджер пулу процесів для PHP, що дозволяє ефективно обробляти HTTP-запити.

gRPC (Google Remote Procedure Call) — фреймворк від Google для віддаленого виклику процедур, який працює поверх HTTP/2.

HTTP (Hypertext Transfer Protocol) — протокол передачі гіпертексту, основа для обміну даними в інтернеті.

JSON (JavaScript Object Notation) — текстовий формат обміну даними, зручний для людини і широко підтримуваний у веброзробці.

PHP (Hypertext Preprocessor) — мова програмування для серверної логіки, особливо популярна у веброзробці.

Pub/Sub (Publish/Subscribe) — шаблон обміну повідомленнями, де видавець не знає про підписників, а повідомлення розсилаються через брокер.

RPC (Remote Procedure Call) — віддалений виклик процедур, що дозволяє виконувати функції на іншому сервері, як локальні.

Swoole — асинхронний сервер для PHP, який дозволяє працювати з неблокуючими операціями.

TCP/IP (Transmission Control Protocol / Internet Protocol) — набір мережевих протоколів, що лежать в основі інтернету.

TypeScript (TS) — надбудова над JavaScript з системою статичної типізації.

VM (Virtual Machine) — віртуальна машина, середовище виконання, у якій програми інтерпретуються або компілюються.

WebSocket — протокол для постійного двостороннього зв’язку між клієнтом і сервером.

XML (eXtensible Markup Language) — розширювана мова розмітки, використовується для зберігання й передачі структурованих даних.

ВСТУП

Із кожним днем ми спостерігаємо стрімкий розвиток цифрових технологій, унаслідок чого дедалі більше систем стикаються з проблемами високого навантаження та зростанням кількості користувачів. У таких умовах інженери змушені шукати нові підходи для вирішення проблем великих затримок, масштабування, слабкої відказостійкості та збереження цілісності даних. Це зумовлює розвиток архітектурних підходів, технологій збереження даних, методів комунікації між сервісами та способів організації програмного коду. Усе частіше розробники відмовляються від класичних рішень, що вважалися «золотим стандартом» протягом десятиліть, на користь нових концепцій, які дозволяють ефективно масштабувати продукт як у межам окремих модулів, так і всієї системи в цілому.

Одним з найактуальніших напрямів сучасної веб-розробки є створення систем із real-time взаємодією, що забезпечують майже миттєвий обмін даними між користувачами та сервером. Особливої популярності останнім часом набули браузерні онлайн-ігри, які не потребують встановлення додаткового програмного забезпечення - достатньо лише відкрити сайт і розпочати гру. Такі продукти часто є цифровими адоптаціями класичних настільних ігор, як-от Alias, Codenames, Мафія. Успішним прикладом подібного рішення є гра Gartic Phone, яка об’єднала значну кількість користувачів у інтерактивному ігровому просторі. Тож розробка веб-застосунку з real-time функціональністю є вкрай актуальною темою, що відповідає сучасним викликам.

У цій роботі ставиться за мету реалізація багатокористувацької гри-вікторини з можливістю масштабування системи під різні навантаження. Аналіз готових рішень та побудова на їх базі системи, достатньої для експлуатації та майбутньої підтримки – теж є важливою частиною цього проекту.

Для досягнення поставленої задачі необхідні виконати наступні пункти:

1. Провести аналіз готових веб-застосунків з real-time взаємодією.
2. На базі досліджень побудувати архітектуру з урахуванням вимог до масштабування та обрати оптимальні технології.
3. Спроектувати та реалізувати логіку кожного сервісу.
4. Протестувати систему та переконатися у її стабільності та готовності до експлуатації.

Структура роботи складається з чотирьох розділів. Перший розділ присвячено аналізу сучасних real-time веб-застосунків - розглядаються архітектурні рішення, технології, способи реалізації ігрової логіки. У другому розділі наведено обґрунтований вибір мов програмування, фреймворків, архітектурного стилю, баз даних, методів комунікації та зовнішніх API. Також описано логіку кожного сервісу та взаємодію між компонентами системи. Третій розділ містить технічні аспекти реалізації, опис труднощів, що виникли під час розробки, а також підготовку до тестування та розгортання. Четвертий розділ присвячений оцінці стабільності системи та проведенню автоматизованого тестування. Усі розділи логічно пов’язані між собою та комплексно розкривають тему, демонструючи повний шлях від аналізу до створення робочого продукту.

РОЗДІЛ 1
АНАЛІЗ ПІДХОДІВ ДО РЕАЛІЗАЦІЇ ЗАСТОСУНКІВ З REAL-TIME ВЗАЄМОДІЄЮ

Для розробки продукту з можливістю масштабування та майбутньої підтримки необхідно добре орієнтуватися у сучасних підходах та практиках щодо створення схожих систем. Аналіз застосунків зі схожою предметною областю може допомогти у правильному виборі архітектурного стилю та технологій.

Цей розділ присвячено дослідженню підходів до реалізації багатокористувацької гри з real-time взаємодією. Розглянуто браузерну гру з відкритим кодом Woogles.io, суть дослідження – відібрати найкращі практики для реалізації гри-вікторина.

Також у цьому розділі проаналізовано відкриті API з можливістю отримання метаданих артистів, альбомів, плейлистів, треків та відповідних аудіо файлів. Проведено дослідження функціональності API, лімітів, політики використання та ціноутворення. Результати аналізу дають можливість перейти до проеєктування системи, яка відповідатиме усім заявленим цілям роботи.

1.1 Приклад популярних проектів з real-time взаємодією

Побудова систем на базі готових рішень – класичний прийом у розробці. Рішення які перевірені на практиці є надійнішими, аніж ті, які працюють у теорії. Першим що зроблено у цьому розділі – аналіз актуальних рішень.

1.1.1 Розбір багатокористувацької гри Woogles.io

Woogles.io – це багатокористувацький браузерний аналог настільної гри “Ерудит”. Проект створений некомерційною організацією Woogle, яка складається з волонтерів. Woogles.io почав світ у 2021 році і досі підтримується розробниками. Гра має стабільний онлайн та позитивні відгуки гравців. На цих підставах можна зробити висновок, що технічним рішенням

розробників можна довіряти, оскільки якість продукту задовольняє вимоги користувачів.

Woogles.io вибраний для аналізу, оскільки має повністю відкритий програмний код на GitHub[1].

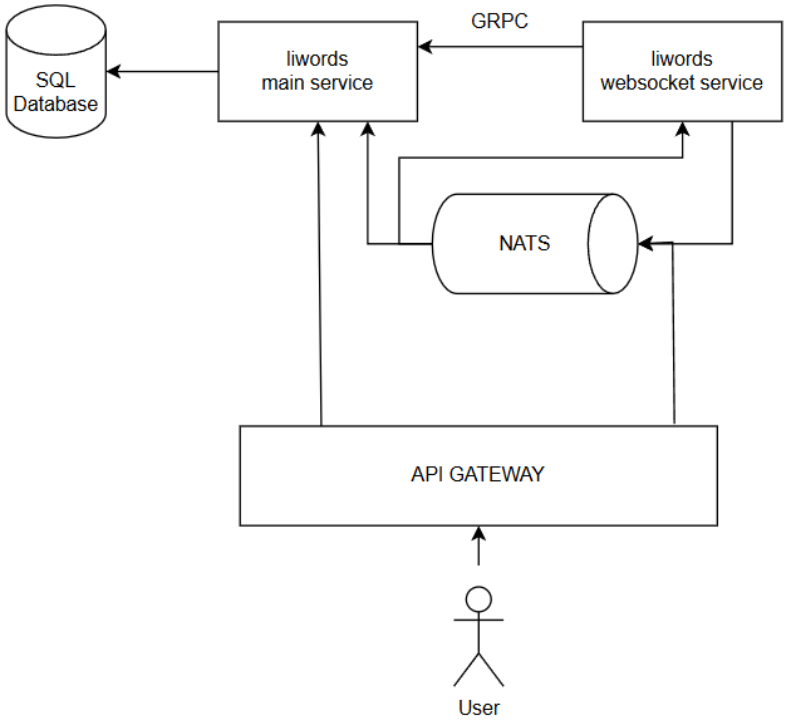


Рисунок 1.1 - Архітектура Woogles.io

Розглянемо деякі з компонентів системи окремо.

Сервіс з ігровими сесіями

Сервіс з ігровими сесіями – серце всього проекту, він відповідає за предметну область самого процесу гейміфікації.

Протокол для real-time взаємодії

Сервіс використовує протокол WebSocket. На відміну звичайного HTTP/HTTPS, цей протокол підтримує постійний зв’язок з користувачем та дає можливість відправляти дані не тільки з серверу на клієнт, а й з клієнту на сервер. WebSocket ідеально підходить для браузерних ігор, оскільки 95% браузерів підтримують його API.

Stateful підхід

Також взято за увагу те, що сервіс з ігровими сесіями є Stateful. Основною і головною перевагою такого підходу є збереження всіх даних у RAM. Це дає нам можливість працювати з інформацією на рівні застосунку та не використовувати базу даних взагалі. Недоліком же є більш складна масштабованість.

Pub/Sub

Варто звернути увагу також на те, що сервіс реалізує патерн Pub/Sub за допомогою NATS для клієнтських запитів та відправки статистики та статусу лобі. Такий підхід допомагає виконувати операції асинхронно, щоб клієнт не дочікуючись їх повного виконання. Це підвищує швидкість самого сервісу, витрачаючи ресурси на більш пріоритетні задачі.

GRPC

При цьому сам сервісу під час створення сесії потрібна інформацію з основного сервісу: дані про користувачів, статичні данні (макет рівня наприклад). Для таких цілей Woogles.io використовує GRPC. На відміну від популярнішого REST, GRPC швидше, оскільки використовує другу версію HTTP та обмінюється даними у бінарному форматі (коли у REST це в основному JSON).

Чат

Іншою із головних функцій даного сервісу є чат. Він ділиться на два типи: глобальний та локальний у межам гри. Чат є класичним прикладом реалізації WebSocket серверу. На відміну від самого ігрового процесу, відправка повідомлення спочатку йде на основний сервіс, де запис фіксується у базі даних, а потім вже після фіксації транзакції інформує клієнтів WebSocket серверу через NATS.

Клієнт

Важливу роль у сервісі з ігровими сесіями відіграє клієнт. Для нього відведено окремий модуль, у якому описано логіку взаємодії з NATS та основі

обмеження кількості запитів за одиницю часу та відправлених даних (структури даних та їх розміру). Це підвищує безпеку сервісу, оскільки отримує від користувача тільки валідні данні, та ефективність: rate limits запобігають DDOS-атакам.

Архітектурні проблеми

Головним недоліком цього сервісу є сильна зв'язаність. Аналізуючи програмний код, легко побачити, що предметна область не відділена від представлення у вигляді WebSocket серверу. Це ускладнює майбутню підтримку проекту, читаність коду погіршується: "розмазана" бізнес логіка не дає розуміння цілісності моделі.

Основний сервіс

Woogles.io має централізований сервіс, який виконує більшість задач. Автентифікація користувачів

Застосунок дозволяє зареєструватися/автентифікуватись кожному відвідувачу. Кожний користувач має ролі та відповідно певні привілеї.

Список привілеїв:

- Створення турнірів
- Менеджмент турнірів
- Модерація користувачів
- Створення шаблонів карт

Автентифікація використовує стандарт JWT. Цей вибір логічний: контроль над доступом користувача не є критично важливим, при цьому швидкість валідизації зростає, оскільки, на відміну від сесій, не потребується звернення до сховища – усі корисні данні зберігаються у самому токені доступу. Також це дуже спрощує автентифікацію для мікросервісної архітектури: токен, створений за допомогою асиметричного шифрування, може бути провалідований сервісами за допомогою публічного ключа, наявність секретного ключа не потребується.

Механізм зберігання даних

Woogles.io використовує лише одну базу даних – PostgreSQL. PostgreSQL – база даних з об’єкто-реляційною моделю, тому вона відповідає усім основним складовим реляційної теорії.

Система повідомлень

У застосунку присутній окремий модуль для відправки листів на електрону пошту. Механізм повідомлення користувача використовується для різних цілей: відправка коду підтвердження під час автентифікації, повідомлення про запрошення приєднатися до лобі, так далі. Модуль використовує протокол STMP для роботи з поштою та оброблює задачі з NATS, такий підхід робить систему легшою для масштабування: з’являється можливість збільшення кількості обробників.

1.1.2 Висновки

Після проведення аналізу багатокористувацької гри Woogles.io можна зібрати список переваг та недоліків тих чи інших технічних рішень. Основними перевагами системи є:

- Використання протоколу WebSocket для real-time взаємодії.
- NATS: системи обміну повідомленнями допомагають виконувати операції асинхронно, також застосунок краще масштабується під різні навантаження.
- Автентифікація за допомогою JWT: є одним з найкращим рішенням для подібних проектів.
- Stateful підхід для ігрових сесій: звільняє нас від інтеграції бази даних для зберігання стану, полегшує роботу з таймерами, асинхронними операціями.

Також важливо звернуту увагу на недоліки продукту:

- Відсутність ізоляції коду за відповідальністю: домена область сильно перев’язана з прямою реалізацією WebSocket серверу. Це робить проект складним для майбутньої підтримки.

Аналіз Woogles.io допоміг вибудувати основу для проектування цієї роботи. Хоч і моделі продуктів доволі сильно відрізняються одне від одного, певні механізми: використані протоколи, технології, побудова зв'язку між сервісами, майже ідеально підходять для реалізації поставленої мети.

У другому розділі даної роботи, під час створення архітектури системи, на вибір буде впливати проведений аналіз.

За мету поставлено не тільки адаптація певних технічних рішень гри Woogles.io, а й вдосконалення та виправлення недоліків.

1.2 Аналіз сторонніх API для роботи з музичним контентом

Одною з головних задач у цій роботі є реалізація наповнення бази даних артистами, альбомами, треками з відповідними аудіофайлами. Без наповненої медіатеки сама гра втрачає свій сенс, оскільки сам процес базується на музиці. Тому ця задача потребує відповідального підходу та аналізу кожної деталі потенційного рішення.

Основою проблемою реалізації системи наповнення музикою є авторські права. Кожен сервіс, який надає потрібний нам функціонал, має ряд обмежень щодо користування їх продуктом: заборона реплікації даних з API, відсутність відповідних аудіо-файлів, проблеми з монетизацією продукту. Для повного легального доступу зі всіма повноваженнями треба звертатися до авторів таких сервісів. Ця робота не ставить перед собою задачу налагодження контакту з власниками та отримання повноцінного функціоналу, тому сторонні API будуть розглянуті та використані тільки у навчальних цілях.

Метою цього підрозділу є аналіз API з метаданими артистів, альбомів, треків та користувацький плейлистів. Розглянуто такі рішення як Spotify API, Apple MusicKit та YouTube Data API. Досліджені можливості сервісів, складність їх інтеграції та потенційні проблеми з якими можна стикнутися під час реальної експлуатації: ліміти на запити, ціноутворення та політика використання. У кінці підрозділу буде приведене порівняння всіх рішень та

побудова основи вибору конкретних API для проектування архітектури у наступному розділі.

1.2.1 Spotify API

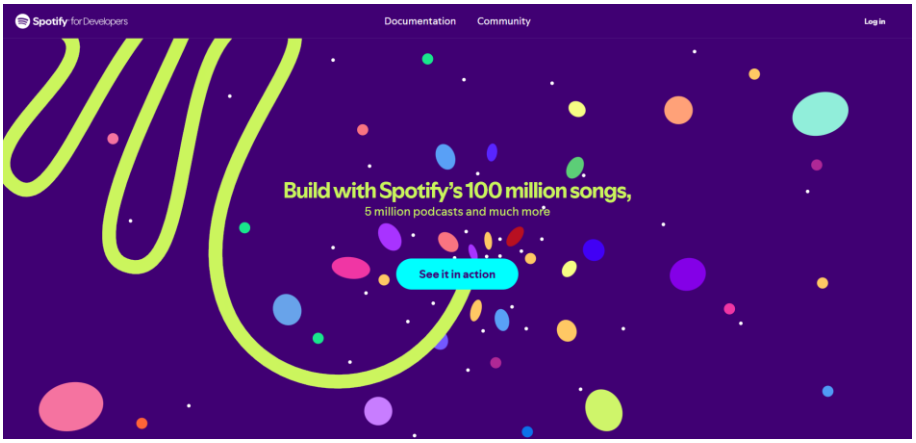


Рисунок 1.2 - Сторінка Spotify for Developers

Spotify - це одна з найпопулярніших музичних стримінгових платформ у світі, що надає користувачам доступ до мільйонів треків, альбомів, виконавців та подкастів. Вона була заснована у 2006 році Даніелем Еком (Daniel Ek) і Мартіном Лорентцоном (Martin Lorentzon) у Стокгольмі. Метою засновників було створити легальну, зручну й масштабовану альтернативу піратському розповсюдженню музики, яке домінувало в 2000-х роках. Сервіс офіційно запущено у 2008 році, спочатку в країнах Європи, а згодом - у США та на глобальному ринку.

На сьогодні Spotify налічує понад 600 мільйонів активних користувачів у всьому світі, з яких понад 200 мільйонів - платні передплатники. Платформа пропонує доступ до бібліотеки з понад 100 мільйонів треків, тисяч подкастів, аудіокниг і персоналізованих рекомендацій, сформованих на базі машинного навчання.

Окрім мобільного та веб-застосунків для прослуховування музики, Spotify активно розвиває екосистему для розробників. Однією з ключових складових цієї екосистеми є Spotify Web API, який дозволяє стороннім застосункам інтегрувати музичний функціонал - наприклад, створення

динамічних плейлистів, відображення даних про виконавців, аналіз треків або навіть управління відтворенням на пристроях користувача[2].

Особисто нас інтересує можливість отримати метадані користувачів, альбомів та треків.

Інформація артистів містить: ім’я, місце у топі, кількість підписників, основні його жанри та аватар. Отримати відповідні данні можна за ідентифікатором за конкретним URI або з інших ресурсів. Нажаль, API не має функції для отримання списку артистів, що робить задачу з реалізації механізму періодичного оновлення бази з принаймні 10 000 найпопулярніших авторів складнішою.

Інформація альбомів містить: назву, дату релізу, обкладинку, чи є реліз синглом чи міні-альбомом, неповну інформацію авторів, перші десять треків та гіпермедіа, достатнього для отримання усіх даних виконавців та треків.

Інформація треків містить: назву, неповну інформацію авторів, неповну інформацію альбома, тривалість та гіпермедіа, достатнього для отримання усіх даних виконавців та альбому.

Можна побачити, що API не надає ні файлу з повних треком, ні аудіо-фрагменту. Відповідне поле у описі схеми відповіді на запит помічене як “Deprecated”: отримати потрібний файл за допомогою одного цього API не є можливими.

Як і більшість великих API, Spotify встановлює обмеження на кількість запитів, які може виконати клієнт за певний проміжок часу. Це необхідно для захисту інфраструктури платформи від надмірного навантаження та забезпечення стабільної роботи сервісу для всіх користувачів. Хоча конкретні числові значення ліміту не задокументовані офіційно, на практиці спостерігається обмеження приблизно на рівні десяти запитів на секунду для одного токена доступу. У разі перевищення допустимої кількості запитів сервер повертає відповідь зі статусом 429 Too Many Requests, а також містить заголовок Retry-After, що вказує, через скільки секунд можна повторити запит.

Окрім загального ліміту швидкості, існують технічні обмеження на кількість об’єктів, які можна отримати або надіслати за один запит. Наприклад, максимальна кількість артистів, альбомів або треків, що можуть бути оброблені одночасно, обмежується п’ятдесятьма, тоді як у пошукових запитах результатів може бути до ста. Під час пагінації стандартне обмеження - до п’ятдесяти елементів на одну сторінку. Крім того, при додаванні треків до бібліотеки користувача або плейлиста також застосовується обмеження - не більше ста елементів за один запит.

Ще одним важливим аспектом є термін дії access token, який використовується для авторизованих запитів - він становить одну годину. Після закінчення цього часу токен потрібно оновити за допомогою refresh token. Деякі запити потребують окремих дозволів, які користувач повинен надати під час авторизації. Наприклад, щоб отримати доступ до приватних плейлистів або персоналізованих рекомендацій, необхідно, щоб користувач надав відповідні дозволи.

Spotify Web API надається для використання за умови дотримання офіційної політики, викладеної у Spotify Developer Terms of Service та Spotify Developer Policy. Основний акцент зроблено на забезпеченні безпеки користувачів, захисті інтелектуальної власності та дотриманні правил чесного використання. Платформа дозволяє інтегрувати базові функції - зокрема, відображення метаданих треків, альбомів та артистів, створення і керування публічними плейлистами, отримання рекомендацій, керування відтворенням, якщо користувач надає відповідні дозволи.

Однак Spotify чітко регламентує, які типи застосунків можуть використовувати API, і за яких умов. Зокрема, заборонено зберігати аудіо, кешувати великі обсяги даних, або використовувати API для створення комерційного продукту без офіційного дозволу. Уся взаємодія має відбуватися відповідно до політики конфіденційності, і розробники зобов’язані чітко

інформувати користувача про те, які дані збираються і як вони будуть використані.

Що стосується реалізації багатокористувацької гри-вікторини з realt-time взаємодією, яка потребує значного обсягу доступу до даних (зокрема, великої кількості артистів, метаданих треків, аудіо-фрагментів, персоналізованої інформації про користувачів або їхню музичну активність), стандартних можливостей публічного API не вистачить.

У таких випадках компанія дозволяє отримати розширені права лише після індивідуального погодження. Для цього розробник має зв’язатися безпосередньо з командою Spotify через офіційний канал.

1.2.2 Apple MusicKit

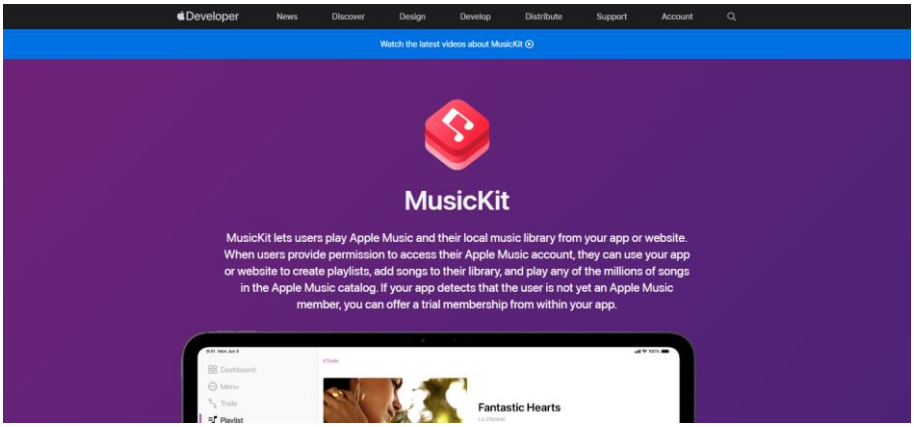


Рисунок 1.3 - Сторінка Apple MusicKit

Apple Music за аналогією з Spotify - музичний сервіс, створений компанією Apple у 2015 році як своя реалізація стримінгового сервісу для своєї продукції. Платформа налічує сотні тисяч релізів артистів різного рівня популярності. Також важливою функцією цього сервісу є створення користувацьких плейлистів як у Spotify. Сервіс позиціонується як преміальний, орієнтований на високу якість звуку та зручність використання в межах фірмового програмного й апаратного забезпечення.

Для досягнення мети цієї роботи компанія пропонує Apple MusicKit. Цей API дозволяє інтегрувати функціональність Apple Music у застосунок.

MusicKit складається з REST API для запитів до музичної бібліотеки та SDK для взаємодії з локальним відтворенням на пристрої користувача. Ми можемо отримати доступ до метаданих про альбоми, треки, виконавців, плейлисти, а також керувати відтворенням, якщо користувач наддасть відповідні дозволи через Apple ID. Корисні доступні метадані мають схожу структуру з Spotify API, тому аналіз схеми тіла відповідей на запит буде проігноровано[3].

Головною перевагою Music Kit над Spotify API є наявність аудіо-фрагменту відповідного треку. Це робить реалізацію задачі з наповнення бази простішою, оскільки приведення одного запису до валідного стану закривається за один запит до API, оскільки не потребується окремий пошук файлів відірвано від основного сервісу.

Але це рішення має один недолік для не експлуатаційного використання: для отримання доступу до Apple MusicKit потребується зареєструвати платний опобліковий запис Apple Music за 99$/рік. При цьому Spotify For Developers є безкоштновним. По-перше, для використання REST API розробник зобов’язаний зареєструвати платний обліковий запис Apple Developer, вартість якого становить $99 на рік. Також є важливим нюанс обмеженння кросплатформеність: MusicKit офіційно підтримується лише на пристроях Apple, і хоча існує веб-версія SDK, вона значно менш гнучка у порівнянні зі Spotify.

Як і у випадку будь-якого масштабного API, Apple MusicKit має технічні обмеження. REST API Apple Music підтримує стандартну систему лімітування запитів, працюючу на throttling - кількість дозволених запитів на певний проміжок часу залежить від типу запиту та статусу автентифікації. Конкретні значення лімітів не публікуються офіційно, проте при перевищенні допустимого порогу сервер повертає відповідь з кодом 429 Too Many Requests. У заголовках відповіді API вказується час, через який слід повторити запит, що дозволяє реалізувати адаптивну логіку запитів у клієнтській частині застосунку.

Окремої уваги заслуговують обмеження на обсяг даних в одному запиті. Для прикладу, при виконанні пошукового запиту максимальна кількість результатів за одну сторінку обмежена параметром limit, який не може перевищувати 25. При цьому пагінація реалізується за offset. Це дає змогу обробляти великі вибірки. Такий підхід дозволяє зберегти керовану навантаженість на інфраструктуру, але ускладнює побудову великих масивів даних

Apple дотримується суворого регламенту. Усі інтеграції через MusicKit повинні відповідати вимогам Apple Developer Agreement, App Store Review Guidelines та Apple Music API Terms of Use. У фокусі - захист авторських прав, контроль над поширенням контенту, а також повна прозорість у питанні збору і обробки персональних даних.

1.2.3 YouTube Data API

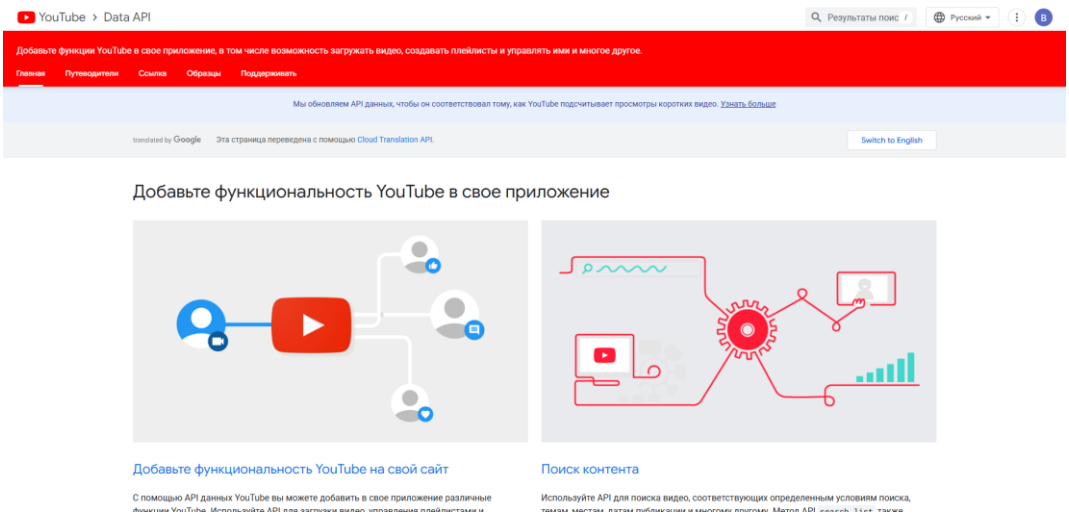


Рисунок 1.4 - Сторінка YouTube Data API

YouTube - це найбільша у світі платформа для обміну відео, що стала культурним, соціальним і технологічним феноменом. Заснована у 2005 році Стівом Ченом, Чадом Герлі та Джаведом Карімом, платформа стрімко набула популярності завдяки простоті публікації відео та широкому охопленню аудиторії. Уже в 2006 році YouTube була придбана компанією Google, що дало змогу масштабувати проєкт до глобального рівня та інтегрувати з іншими сервісами екосистеми. Станом на сьогодні YouTube налічує понад 2,5 мільярда

активних користувачів щомісяця, щодня переглядаються мільярди годин контенту - від музики, освітніх матеріалів і влогів до стрімів, коротких відео (Shorts) і подкастів.

На відміну від Spotify For Developers та Apple MusicKit, Youtube Data API не працює напряму з музикою, хоча і має окремий сервіс Youtube Music, який є аналогом описаних раніше сервісів. На жаль, Youtube Music не має публічного API, в той час як у самої оригінальної платформи він є. Треба зазначити, що у контексті сервісу не існує таких понять як артист, альбом та трек, оскільки Youtube є відеохостінгом, а сам музичний контент є підмножиною. Замість виконавців – канали, альбомів – плейлісти, треків – відео. Це одразу створює певні проблеми з експлуатацією: ми однозначно не можемо виявити чи є той чи інший автор насправді музикантом, чи є відео дійсно композицією. Якщо звернутися до документації та проаналізувати пошукові параметри у деяких методів, то можна виявити, що серед допустимих значень є topicId – фільтр за тематикою. Youtube надає список валідних ідентифікаторів і для музики, на момент написання роботи їх 15. Це допомагає підвищити шанс отримання коректного контенту, але не гарантує 100% успіху, оскільки така тематика торкається також, наприклад, обзорів[4].

У загальному випадку, YouTube не є найкращим джерелом даних для реалізації музичної гри, оскільки API платформи не має чіткої музичної структури, а формат відео сам по собі є надмірно загальним. Проте, незважаючи на це, YouTube може стати дуже корисним інструментом, якщо використовувати його у комбінації з іншими сервісами. Така робота буде продемонстровано у наступних розділах. Таким чином, YouTube виступає як додаткове візуальне або аудіо-доповнення, яке не забезпечує первинне джерело істини, але може підсилити інтерактивність або якість виводу для користувача.

Центральним методом для пошуку потрібного відеоконтенту є ендпоінт search. Він дозволяє надсилати запити за ключовими словами (наприклад, "Шугар – Тьотя official video") і фільтрувати тип результатів.

Ще одним важливим обмеженням є те, що YouTube API не дозволяє напряму завантажувати відео чи аудіо. Це серйозно ускладнює побудову функціоналу, що потребує фрагментів треків для локального аналізу або обробки. Офіційно доступ до контенту дозволено лише через вбудовані плеєри (iframe API) або пересилку на платформу. Це робить неможливим витяг аудіо у «фоновому режимі». Водночас, існують неофіційні інструменти, які дозволяють завантажити відео й аудіо потоково через прямі посилання. Варіанти таких утиліт та їх порівняння буде описано під час побудови архітектури.

Окремої уваги заслуговує система квот YouTube API. На відміну від Spotify або Apple MusicKit, де обмеження не відчуваються в типовому використанні, YouTube має доволі жорстку квотну політику, де кожен запит має певну "вартість" у балах. Щоденний ліміт - 10 000 одиниць на проєкт. Наприклад, звичайний search запит коштує 100 одиниць, а videos.list - 1. Це означає, що масовий пошук відео або збір великої кількості метаданих швидко «з'їдає» ліміт. У рамках експериментів або вузької задачі це не створює критичних проблем, але при масштабуванні до повноцінного сервісу або автоматизованого збору інформації квоти стають стримуючим фактором. Збільшення ліміту можливе лише через ручу перевірку Google із поясненням, для чого саме потрібне підвищення, що теж ускладнює реалізацію.

У плані політики використання, YouTube є досить закритою платформою: будь-яке системне використання API повинно відповідати їхнім правилам сервісу, серед яких особливо жорстко регулюється завантаження, перетворення та кешування контенту. Умови Google Cloud Console прямо забороняють зберігати відео або аудіо, отримані через API, без згоди YouTube. Якщо ж йдеться про наукову роботу або навчальні цілі - використання API,

навіть із певними обмеженнями, є виправданим. Але якщо стоїть питання про платний доступ до API, реєстрацію комерційного додатку або масштабну інтеграцію - це точно не та платформа, з яку варто використовувати. У цьому випадку Spotify або Apple MusicKit є значно кращим варіантом.

1.2.4 Порівняння готових рішень

1.3 Побудова процесу розробки системи

Чітка поетапність у розробці - це не просто нав'язана формальність, а дуже важлива річ, що допомагає не збитися з курсу, правильно розставити пріоритети і рухатись правильно за логікою. Наявність структура, легше контролювати прогрес, помічати проблеми ще до того, як вони стають катастрофами, і вчасно вносити правки, якщо щось пішло не так. За основу було взято модель SDLC, яка використовується у реальному бізнесі і є базою у розробці продукту[5]. Перші два етапи уже було пройдено - планування та аналіз. Збір вимог, розгляд схожого рішення, на якому можна базуватись, та дослідження сторонніх API, потрібних для реалізації певних механік, вже описано до кінця цього розділу. Тому роботу над системою було поділено на три основні фази, кожна з яких має свою мету, свій підхід і свою роль у загальній картині:

- Архітектура (розділ 2).

Починається все з фундаменту. Тут розглядається, як виглядатиме система з висоти пташиного польоту: які модулі будуть, як вони взаємодіють, де живуть дані і хто з ким говорить. Це той момент, коли ще нічого не написано, але вже зрозуміло, що буде створено.

- Реалізація (розділ 3).

Далі - код. Саме тут усе оживає: реалізується логіка гри, підключаються API, створюється інтерфейс, обробляються помилки і додається все, що потрібно для повноцінної роботи системи. Це найбільший і найбільш «живий» етап, у якому починає вимальовуватись реальний продукт.

- Тестування, документація, розгортання (розділ 4).

Фінальний штрих. Після написання - перевірка: що працює, що зламалось, що можна покращити. Паралельно пишеться документація, щоб у майбутньому не довелося заново розбиратись, що і чому було зроблено. І в кінці - розгортання: перенесення всього на живе середовище, де система вже доступна для користувача.

Останній етап у вигляді обслуговування та підтримки не буде розглянутий, оскільки виходить за рамки даної роботи.

ВИСНОВОК ДО РОЗДІЛУ

Перший розділ закладає фундамент всього проекту: чітко визначено мету проекту та побудований шлях повної реалізації системи. Першою метою було пошук аналогічних багатокористувацьких ігор для аналізу та розробки основ майбутнього проектування. За основу взято проект Woogles.io, оскільки розробники відкрили повний програмний код, що полегшило дослідження технічних рішень. Проведено аналіз більшої частини модулів, підмічено цікаві рішення, які використані під час побудови архітектури. Також приведено недоліки системи та опис потенційних проблем при їх наявності.

Наступним важливим кроком стало дослідження сторонніх API для інтеграції артистів, альбомів, плейлістів та треків. Розібрано 3 основних рішення: Spotify for Developers, Apple MusicKit та YouTube Data API. Можливості, схематика API, ціноутворення, ліміти – все це стало предметом аналізу. Дослідження показало, що кожне рішення має свою унікальні переваги: Spotify має велику аудиторію та безкоштовним доступ до API, MusicKit містить аудіо-фрагменти композицій, YouTube дає доступ до пошуку повноцінних треків. Але кожне з рішень має серйозні недоліки: Spotify – відсутність аудіо-файлів, MusicKit – ціна, Youtube – нелояльну політику та малі ліміти. Після порівняння кандидатів було обрано комбінований підхід Spotify + YouTube – це найкращий варіант для використання у навчальних цілях.

Останнім завдання стало побудова наступних етапів розробки. У цьому сильно допоміг набір принципів SDLC. Розроблено конкретних шлях від проектування до розгортання застосунку.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА ВИБІР ТЕХНОЛОГІЙ

Цей розділ присвячено, мабуть, найголовнішому етапу розробки будь-якої системи – проектуванню. Після збору вимог та аналізу готових рішень почато побудову архітектури застосунку та вибір технологій. Цей процес складається з дослідження існуючих практик, їх порівняння та відбір найкращих кандидатів. На це виділено велику частину часу, оскільки підсумкові рішення значно впливають на швидкість розробки, цілісність проекту, складність реалізації та майбутню підтримку.

З кожним новим кроком у цьому розділі рішення будуть наближати нас до фінального результати, такого, якого буде достатньо, щоб перейти на наступний етап реалізації. Кожний наступний процес базується на попередніх: проектування починається з вибору архітектурного стиля, потім проаналізовано різні мови програмування та обрані ті, які краще підходять під нього, вибір фреймворків базується на мовах і т.д. Такий підхід приводить нас з абстракцій до конкретики.

База даних - це один із ключових елементів, який потрібно ретельно підібрати перед початком реалізації. Наша задача - визначити, яка модель зберігання краще відповідає системи: реляційна чи документно-орієнтована. Необхідно врахувати, як будуть зберігатися зв’язки між сутностями (наприклад, треки, автори, питання), наскільки просто буде масштабуватися рішення, та як забезпечити індексацію і швидкий доступ до даних. Окремо варто звернути увагу на підтримку транзакцій і які інструменти база даних дає для вирішення проблем конкурентних запитів.

Ще одне важливе питання - організація комунікації між сервісами. Система має складатися з окремих модулів, які виконують ізольовані функції, але при цьому ефективно взаємодіють між собою. Тут потрібно прийняти рішення: яку модель обміну даними використовувати - синхронну чи асинхронну. Синхронна модель проста у реалізації, але вразлива до проблем із

мережею: у разі недоступності одного з сервісів уся операція може зірватися. Асинхронний підхід складніший, але дозволяє гарантувати доставку повідомлень і підвищує стійкість системи до відмов. Наша задача - розібратися, де в системі варто використовувати кожен із цих підходів, і побудувати комунікацію так, щоб не втрачати дані навіть у критичних ситуаціях.

Крім того, потрібно сформулювати чітку відповідальність кожного сервісу. Важливо розподілити функціональність таким чином, щоб сервіси не дублювали один одного, а комунікація між ними була простою, стабільною і передбачуваною. Також потрібно врахувати можливість повторного виконання операцій у разі збоїв, реалізувати механізми підтвердження обробки повідомлень і способи автоматичного відновлення.

Також важливою метою цієї роботи є забезпечення легкої масштабованості системи. Тому аналіз окрім звичайних нюансів повинен будуватися на можливостях адаптації під різні навантаження.

2.1 Аналіз та вибір архітектурного стилю

Вибір архітектурного стилю задає наступний вектор розвитку застосунку. Від цього залежить як компоненти взаємодіють між собою, наскільки система піддається масштабуванню, ізоляція сервісів, спосіб їх взаємодії та багато інших факторів.

Розглянуто кілька архітектур – монолітну, мікросервісну та шарову.

2.1.1 Монолітна архітектура

Монолітна архітектура - це архітектурний стиль, при якому вся бізнес-логіка, обробка запитів, інтеграції, служби та інтерфейси розміщуються в одному цілісному застосунку. Умовно кажучи, маємо єдину велику програму, яка містить усе одразу - від обробки запитів до доступу до бази даних.

Це класика жанру. Монолітами починали майже всі великі компанії, і цей підхід використовують уже десятки років. Його популярність пояснюється просто: розробити моноліт швидше, легше та дешевше. Один репозиторій,

одна збірка, одна точка розгортання. Розробнику не потрібно думати про складні механізми зв’язку між модулями чи налаштування інфраструктури.

Але за простотою криються серйозні недоліки. По мірі зростання проєкту код стає важким для підтримки: функціональність тісно пов’язана, і зміна в одному місці часто тягне помилки в інших. Порушується принцип єдиної відповідальності, а розробка новою функціональності становиться дуже складною.

Особливо критичним недоліком моноліту є неможливість ефективного масштабування. Це дуже серйозна проблема для нашого застосунку, оскільки одною з метою роботу є забезпечення адаптації системи під різні навантаження. В монолітній архітектурі ми не можемо масштабувати окремі частини застосунку відповідно до навантаження. У нашій системі це, наприклад, сервіс парсингу пісень та менеджер створення ігрових сесій. Усе доводиться масштабувати одразу, що витратно й неефективно. Але найсерйозніша проблема - база даних. У монолітах вона зазвичай єдина, не розділена за контекстами, і це означає: масштабувати її та сегментувати складно, часто майже неможливо. Якщо ми не зможемо масштабувати зберігання та обробку даних, то єдиний вузол не впорається з навантаженнями[6].

Попри всі недоліки класичного моноліту, існує практика, яка дозволяє його використання навіть у складних системах - але не як фінальну архітектуру, а як проміжну фазу розвитку. Такий підхід має назву Monolith First.

Ідея проста: спочатку будується моноліт, але не хаотичний, а структурований за доменами, з чітким розділенням відповідальностей між модулями та ізоляцією контекстів. Взаємодія між цими частинами відбувається через внутрішні або зовнішні інтерфейси. Оскільки весь код знаходиться у одному місці, ми можем визивати сервіс напряму, але при цьому реалізувати будь-яку іншу взаємодію (REST API, наприклад). Таким чином,

застосунок архітектурно поводиться як мікросервісна система, хоча фактично ще залишається єдиним монолітом.

У цьому - головна перевага: ми отримуємо зручність моноліту, тобто одну базу даних, повну підтримку транзакцій, спрощене розгортання, і водночас готуємося до подальшої сегментації. Це - компроміс між чистою мікросервісною архітектурою і класичним монолітом, який дозволяє: швидше запустити MVP, зберігати стабільність, уникнути зайвої інфраструктурної складності, з часом винести частини в окремі мікросервіси без рефакторингу всієї системи. Підхід Monolith First сьогодні активно використовується в індустрії, особливо в стартапах та R&D проєктах.

Втім, у цій роботі ми не будемо розглядати та брати як варіант цей підхід, оскільки завдання передбачає побудову архітектури, яка одразу буде розрахована на горизонтальне масштабування, високий рівень автономності компонентів та незалежне розгортання частин системи.

2.1.2 Мікросервісна архітектура

Мікросервісна архітектура - це підхід до розробки програмного забезпечення, за якого система складається з набору незалежних або частково залежних один від одного сервісів, кожен з яких виконує конкретну бізнес-функцію і взаємодіє з іншими через чітко визначені інтерфейси.

На відміну від класичного моноліту, мікросервісна модель є відносно новою. Вона набула популярності не просто так - її поява стала відповіддю на потребу ізолювати контексти всередині великих застосунків. З часом стало зрозуміло, що жорстко зв’язані частини системи створюють значні труднощі в підтримці, масштабуванні й розвитку. Мікросервіси вирішили цю проблему: кожен контекст став незалежним, що дозволило командам працювати паралельно, масштабувати сервіси окремо та розгортати їх незалежно один від одного.

Один із ключових наслідків переходу до мікросервісів - це сегментація бази даних. Хоч ми й не завжди одразу застосовуємо шардинг чи реплікацію,

вже саме розбиття на окремі бази даних за сервісами покращує продуктивність. Навантаження починає розподілятись між вузлами з різними контекстами, а це - шлях до більш передбачуваної масштабованості. Крім того, ще на етапі проектування ми закладаємо логічну структуру, яка робить майбутнє масштабування набагато легшим. Та попри всі переваги, у мікросервісної архітектури вистачає і мінусів. У першу чергу це значно вища складність. Комунікація між сервісами, логування, моніторинг, безпека, синхронізація - усе це потребує додаткових зусиль. Друге - дебагінг та пошук помилок. В одній системі з десятками сервісів навіть знайти джерело проблеми дуже важко, та нерідка ситуація, коли логічно зв’язаний потік проходить купу сервісів і прослідкувати за цим процесом, особливо коли є проблемно реалізоване логування[6].

І, напевно, головне - відмова від сильних сторін моноліту, таких як транзакції у базах даних. Тепер атомарність операцій перекладається на розробника. Щоб гарантувати узгодженість між сервісами, потрібно впроваджувати розподілені транзакції: або через складний двофазний коміт (2PC), або через більш гнучкий, але теж непростий, патерн Saga. Іншими словами, мікросервіси - це не магічне рішення, а відповідальність за гнучкість і масштабованість, яка завжди має свою ціну. Але в нашому випадку, коли система має рости, обробляти великі об’єми даних та розвиватись модулями, цей шлях є найбільш доцільним ніж монолітна архітектура.

2.1.3 Шарова архітектура

Шарова архітектура - це підхід до організації коду всередині окремого сервісу, який передбачає чітке розділення відповідальностей між різними логічними рівнями системи. На відміну від мікросервісної архітектури, яка визначає взаємодію між сервісами, шарова архітектура описує внутрішній устрій одного сервісу, незалежно від того, чи є він частиною моноліту чи мікросервісної системи.

У старих проектах було поширеною практикою напряму пов’язувати бізнес-логіку, наприклад, із HTTP-клієнтом або запитами до бази даних. Такий підхід призводив до високої зв’язаності коду: зміна в одному місці могла несподівано зламати інше, навіть не пов’язане напряму. Наприклад, правка у логіці маршрутизації запитів могла вплинути на модель даних - а це вже серйозне порушення принципів стійкості системи. Щоб уникнути цього, з'явився підхід, за яким код ділиться на шари відповідно до їхньої відповідальності. Кількість шарів може бути довільною, але класична реалізація зазвичай містить три основних:

- Шар доменної області - містить предметну логіку, яка описує "правила гри" у рамках конкретного бізнес-контексту. Це серце застосунку.
- Шар сервісів - реалізує сценарії використання, оркеструє взаємодію між доменом, інфраструктурою та зовнішніми джерелами.
- Шар представлення - приймає запити (наприклад, HTTP), трансформує дані для відображення, працює з контролерами, адаптерами тощо.

Ключовий принцип тут полягає в тому, що верхній шар знає про нижній, але не навпаки. Це забезпечує ізоляцію, яку легко протестувати, розширити або замінити частинами без ризику порушити всю систему.

Шарова архітектура особливо добре поєднується з методологією DDD (Domain-Driven Design), яка вимагає чіткого відокремлення предметної області від інфраструктурної логіки. Однак сам DDD виходить за рамки роботи, тому розглядатися не буде[7].

2.1.4 Висновок

Після розгляду кількох архітектурних стилів постає завдання -ухвалити остаточне рішення щодо того, на яких принципах будується наша система. Якщо уважно проаналізувати функціональні вимоги, можна помітити, що логіка застосунку природно ділиться на окремі контексти. Це означає, що різні частини системи працюють з незалежними наборами даних, бізнес-правилами та сценаріями використання.

Такий поділ відкриває перед нами одразу кілька переваг:

- високий рівень ізольованості між компонентами
- масштабованість кожного блоку окремо залежно від навантаження
- гнучкість у розробці та можливість паралельної роботи команд

Усе це робить мікросервісну архітектуру найкращим вибором для нашого проєкту. Саме вона стане фундаментом побудови системи. Кожен сервіс виконуватиме окрему функцію, буде відповідати за свій контекст та зможе масштабуватись незалежно від інших.

Окрім розподілу на сервіси, важливим є також внутрішній устрій кожного з них. Щоб забезпечити зручність підтримки та подальшої розробки, ми обираємо шарову архітектуру. Вона дозволяє чітко розділити код за відповідальністю, зменшуючи зв’язаність і підвищуючи стійкість до змін. Кожен сервіс нашої системи буде побудовано на трьох основних шарах:

- Доменний шар: містить сутності, бізнес-логіку, value-об’єкти, а також інтерфейси сервісів, які працюють з компонентами доменної області, але мають реалізацію на шарах вище.
- Сервісний шар: реалізує сценарії використання, керує модельними об’єктами, зв’язує домен із зовнішніми технологіями. Це мозок сервісу, який оркеструє взаємодію між шарами.
- Інфраструктурний шар: містить реалізації інтерфейсів із доменного шару, а також інші прикладні компоненти, такі як контролери, конфігураційні файли, консольні команди, адаптери до сторонніх API тощо. Усе це можна умовно назвати технічною оболонкою сервісу.

Такий підхід дозволяє будувати гнучку, масштабовану й зручну в підтримці систему, яка чітко розділяє відповідальність на всіх рівнях. У наступному підрозділі ми детальніше розглянемо, як буде проведено декомпозицію системи: які сервіси виділено, які задачі вони виконують і як між собою взаємодіють.

2.2 Функціональне розбиття системи

Після того як було обрано мікросервісну архітектуру як основний підхід до побудови системи, наступним логічним кроком є визначення меж кожного окремого сервісу.

Щоб коректно провести декомпозицію, спочатку проаналізувано ключові функціональні блоки, які наша система має реалізовувати. Якщо поглянути на загальні вимоги до застосунку, можна виділити п’ять основних завдань, які він вирішує:

- Парсинг медіатеки зі сторонніх API (Spotify API та Youtube Data API)
- Збереження та обробка медіатеки, яка надалі використовується в ігровому процесі
- Навігація по медіатеці для клієнта (фільтрація, пошук, сортування)
- Керування ігровими сесіями: створення, менеджмент, підготовка сесій до старту
- Проведення самих ігрових сесій — логіка гри, облік результатів, взаємодія з користувачем

Прийнято рішення — кожне з цих завдань винести в окремий сервіс, що дозволить зберігати чіткість контексту, забезпечити незалежний розвиток кожної частини системи та легко масштабувати будь-який компонент за потреби. У цьому підпункті детально описано кожен із сервісів: яку саме відповідальність він на себе бере та яку роль відіграє у загальній структурі застосунку; та як він спілкується з іншими сервісами системи.

У цьому підпункті зроблено невеликий опис кожного із сервісів: яку саме відповідальність він на себе бере, яку роль відіграє у загальній структурі застосунку, та як він комунікує з іншими сервісами системи.

2.2.1 Song Parser

Ключовий сервіс, що відповідає за інформаційне наповнення всієї системи. В його основі лежить механізм взаємодії з зовнішніми API, зокрема Spotify API та YouTube Data API. Сервіс регулярно оновлює заздалегідь

визначений список артистів та виконує глибокий парсинг: від артиста до альбомів і треків. Основна взаємодія відбувається з Core-сервісом, якому він передає підготовлену інформацію для збереження, а також приймає від нього запити на подальшу обробку — наприклад, отримання альбомів певного артиста.

2.2.2 Core

Центральний сервіс, що виконує функцію основного сховища даних у системі. Уся ключова інформація (артисти, треки, альбоми, плейлисти) зберігається саме тут. Core активно взаємодіє з майже кожним іншим сервісом:

- Song Parser — для прийому та збереження нових даних
- Search — для реплікації інформації у форматі, зручному для пошуку
- Game — для передачі медіатеки, яка використовується під час ігрового процесу.

2.2.3 Search

Сервіс, відповідальний за підготовку та зберігання інформації у форматі, максимально оптимізованому для пошуку на стороні клієнта. Отримує дані майже по всім сутностям із Core та перетворює їх для зручної навігації, фільтрації та сортування.

2.2.4 Matchmaker

Сервіс-оркестратор, що керує ігровими сесіями. Відповідає за моніторинг запущених ігор та обробку запитів на виділення нових ігрових кімнат. Саме він визначає, коли і як запускати нові сесії відповідно до навантаження та запитів користувачів.

2.2.5 Game

Головний сервіс усієї системи, який відповідає за real-time взаємодію між клієнтами та запуск самої гри. Отримує доступ до медіатеки через Core-сервіс і забезпечує виконання логіки ігрового процесу, керує обробкою відповідей, підрахунком балів тощо.

2.3 Вибір основних технологій

2.3.1 Мови програмування

Після того як ми поділили нашу систему на окремі сервіси, настав момент вибору, мабуть, найважливішої технічної технології — мови програмування. Це дуже відповідальне рішення, оскільки саме мова визначає набір доступних інструментів і технологій, які ми зможемо використовувати, а також те, як архітектура самої мови вплине на розробку. У межах цієї роботи буде використано декілька мов програмування. Чому ми не можемо обійтись лише однією — буде пояснено нижче.

Варто зазначити, що для реалізації поставленої мети підходить майже будь-яка сучасна мова, яка активно підтримується як розробниками, так і спільнотою. Але після аналізу було прийнято рішення зупинитись на двох мовах програмування: PHP та JavaScript (TypeScript).

PHP

PHP — це інтерпретована мова програмування, яка з'явилась ще в середині 90-х років і довгий час була одним із головних інструментів для створення вебзастосунків.

PHP має динамічну типізацію. Це означає, що тип змінної визначається під час виконання програми і може змінюватися прямо в процесі роботи. Такий підхід значно пришвидшує розробку, оскільки зменшує об’єм рутинної роботи пов’язаної з типізацією, але при цьому призводить до проблем з продуктивністю: падає швидкість виконання, збільшується споживання пам’яті. Це пов’язано з тим, що ми не можемо скомпілювати код напряму, як у C, бо компілятор просто не знатиме, як працювати зі структурами, тип яких наперед не визначений. Замість цього PHP компілює свій код у проміжні операційні коди для віртуальної машини Zend VM, яка вже і виконує сам код.

Окремо варто згадати проблему асинхронності. У випадку з PHP-FPM, мова не підтримує асинхронну модель із коробки. Хоча в самому PHP існують рішення для конкурентного або асинхронного виконання (наприклад Swoole) їх використання є оверхедом[8].

Незважаючи на вищезгадані недоліки, PHP чудово підходить для реалізації Core та Search сервісів. Це пов’язано з тим, що модель PHP-FPM працює за принципом: отримали запит, запустили процес, віддали відповідь, завершили процес. Такий підхід знімає з нас відповідальність за менеджмент пам’яті, оскільки після кожного запиту процес помирає і вся пам’ять автоматично очищується. Це ідеально лягає на задачі Core та Search, де основна логіка зводиться до роботи з БД і віддачі готових даних без складних обчислень чи взаємодій.

У цій роботі PHP буде використовуватися у двох різних режимах. Перший — це використання в парі з PHP-FPM, який є менеджером пулу процесів PHP із підтримкою FastCGI. Цей підхід буде застосований для обробки зовнішніх HTTP-запитів до API. PHP-FPM чудово підходить для таких задач, оскільки кожен запит обробляється окремим процесом, який завершується після виконання, автоматично очищаючи пам’ять.

Другий варіант використання — це запуск звичайного PHP-коду на постійному з’єднанні через сокет, що дозволяє реалізувати воркерів для обробки асинхронних повідомлень із черги. Така реалізація буде застосовуватись для взаємодії з брокером повідомлень.

Javascript (Typescript)

Окрім реалізації стандартного API, однією з важливих задач цієї роботи є підтримка механізму real-time взаємодії. Для реалізації такого функціоналу, опираючись на аналіз архітектури Woogles.io, ідеальним вибором є WebSocket. Проте, оскільки ми використовуємо PHP-FPM як основний інструмент для обробки HTTP-запитів, реалізувати WebSocket без суттєвих обмежень на PHP не вийде. Саме тому виникла потреба у виборі іншої мови програмування, яка краще підходить для довготривалих з’єднань і високої конкурентності. В результаті було обрано JavaScript.

JavaScript — це інтерпретована мова програмування з динамічною типізацією, яка з самого початку розроблялася для роботи в браузері, але з

часом, завдяки появі Node.js, отримала широке застосування і на бекенді. Як і PHP, JavaScript дозволяє швидко розробляти застосунки за рахунок відсутності жорсткої типізації, що значно скорочує час розробки. Але, на відміну від PHP, який має системи підказок типів та інші механізми структуризації, JavaScript не пропонує з коробки аналогічного функціоналу.

Цю проблему вирішує TypeScript — надмножина JavaScript, яка вводить систему статичної типізації. Його складно назвати повноцінною окремою мовою програмування, оскільки TypeScript компілюється у звичайний JavaScript. Однак він має низку додаткових можливостей, таких як enum-и, декоратори та інші функції, пов'язані з автогенерацією коду. TypeScript значно підвищує стабільність проектів, дозволяє краще контролювати типи на етапі компіляції і спрощує розробку.

У межах цієї роботи буде використовуватися як JavaScript, так і TypeScript. JavaScript буде застосовуватись у невеликому модулі, що відповідає за відстеження процесу парсингу плейлиста та реалтайм-сповіщення користувачів про хід цього процесу. Натомість TypeScript буде основною мовою в більш складному сервісі Game, де за рахунок типізації можна чітко описати внутрішню ігрову модель, правила гри, обробку станів та комунікацію між клієнтами.

2.3.2 Фреймворки та бібліотеки

Мови програмування самі по собі, попри всі їхні можливості, рідко здатні повністю задовольнити потреби сучасного застосунку. Хоча в базовій поставці часто є необхідні інструменти, наприклад, HTTP-клієнти, для побудови повноцінної системи цього недостатньо. Система, яку ми проєктуємо, потребує великої кількості допоміжних механізмів: роутінг запитів, валідація даних, взаємодія з базами даних через ORM, реалізація патернів на кшталт командної шини, подієвих диспетчерів, та багато іншого.

До вибору зовнішніх залежностей необхідно підходити з особливою обережністю. Хоча вони значно прискорюють розробку й часто вирішують

типові задачі, повністю покладатися на сторонній код — ризиковано. Зовнішні бібліотеки можуть містити помилки, мати приховані вразливості або в будь-який момент перестати підтримуватися спільнотою чи авторами.

У цій роботі ми свідомо уникаємо надмірного використання залежностей. Буде обрано тільки найнеобхідніші та перевірені рішення, які реально приносять користь у контексті нашої архітектури. Жодного фанатизму чи бездумного додавання модулів — лише те, що критично важливо для стабільної та масштабованої реалізації.

Щоб зручно підключати зовнішні залежності, слідкувати за їх версіями, керувати оновленнями та перевіряти їх на безпеку, необхідно використовувати менеджери пакетів. У цій роботі для PHP буде використовуватись Composer, а для JavaScript/TypeScript — npm. Важливо зазначити, що в рамках цього проєкту Composer буде використовуватись не лише як менеджер пакетів, але й як механізм автозавантаження класів (через autoload), що значно спрощує структуру та підтримку коду.

Symfony та Laravel

Оскільки основні сервіси нашої системи будуть реалізовані на PHP, надзвичайно важливо обрати фреймворк, який найкраще підійде під потреби проєкту. На сьогодні два найпопулярніші фреймворки в PHP-екосистемі — це Symfony та Laravel. Хоча з першого погляду вони дуже схожі (Laravel навіть використовує багато компонентів Symfony під капотом), між ними є принципові відмінності у філософії розробки, рівні гнучкості та орієнтованості на різні типи проєктів.

Laravel зазвичай використовується для створення менш складних систем. Його головний фокус — це швидкість розробки та простота входу для початківців. У фреймворку вже вбудовано багато рішень, які, хоча й дозволяють дуже швидко почати, не завжди легко адаптуються під складні або нестандартні архітектури. Наприклад, у Laravel основною ORM є Eloquent, яка реалізує патерн Active Record. Це робить її дуже простою у використанні, але

в той же час така архітектура порушує низку принципів (наприклад, SOLID) і складніше масштабується в складних системах.

З іншого боку, Symfony орієнтований на гнучкість, масштабованість і суворе дотримання принципів проєктування ПЗ. Його ORM/ODM — Doctrine — реалізує патерн Data Mapper, який хоч і складніший у початковому використанні, зате краще підходить для складних, великомасштабних застосунків і відповідає класичним підходам у побудові архітектур. Подібний підхід спостерігається майже у всіх компонентах Symfony: замість нав'язування єдиного шляху, Symfony дозволяє гнучко налаштовувати систему під власні потреби.

Беручи до уваги те, що наша мета — побудова стабільної, підтримуваної та масштабованої системи, вибір був зроблений на користь Symfony. Детально розглядати кожен компонент фреймворку, який буде використано, немає сенсу в межах цієї роботи, але нижче подано список основних компонентів Symfony, які будуть застосовуватись у нашій системі:

- HttpClient
- Router
- Doctrine ODM
- Messenger
- Serializer

Node.js та бібліотека ws

Для використання JavaScript поза межами браузера, у рамках окремого середовища виконання, у цій роботі буде застосовуватись Node.js. Це середовище дозволяє запускати JS-код на сервері завдяки використанню рушія V8.

Бібліотека ws є однією з найпопулярніших WebSocket-реалізацій для Node.js. Вона відома своєю простотою використання, високою продуктивністю та мінімалістичним дизайном, який не нав’язує зайвої логіки.

2.3.3 Бази даних

Після того як ми визначилися з вибором мов програмування та необхідних бібліотек, настав час зробити ще одне ключове архітектурне рішення — вибір бази даних. Це рішення впливає не лише на те, як зберігатимуться дані, а й на ефективність роботи з ними, можливість масштабування та забезпечення цілісності інформації. У професійному середовищі прийнято ділити бази даних на два великі класи: SQL та NoSQL. У нашому випадку вибір припав одразу на два кардинально різні підходи — MySQL і MongoDB.

MySQL — це класична реляційна база даних, яка вже багато років є стандартом де-факто у світі розробки. Її використовують тисячі компаній по всьому світу, і вона перевірена часом. Основні переваги реляційних баз — це транзакційна система, строгі обмеження цілісності та нормалізована модель зберігання.

Транзакції дозволяють виконувати набір операцій як єдине ціле — або всі операції виконуються, або жодна. Це гарантує, що база даних не потрапить у проміжний або непередбачуваний стан. Констрейнти допомагають підтримувати зв’язність і логічну правильність між таблицями. Саме зовнішній ключ — це одна з найпотужніших функцій реляційних баз, якої фактично не існує в MongoDB, бо вона просто не передбачає жорсткого зв’язування між документами. Крім того, реляційні БД зберігають дані в нормалізованому вигляді, що зменшує дублювання та допомагає уникати логічних дефектів у структурі.

Проте ці ж переваги можуть перетворитися на недоліки при потребі горизонтального масштабування. Транзакції та зовнішні ключі працюють лише в межах одного вузла, що ускладнює масштабування. MySQL з коробки не надає таких механізмів, і хоча існують сторонні рішення, більшість із них змушують відмовитися від того самого функціоналу, які й роблять реляційну базу настільки корисною.

На допомогу в таких випадках приходять NoSQL бази, зокрема MongoDB. Вона зберігає дані у вигляді BSON-документів, які по суті є розширеною формою JSON. Такий підхід дозволяє зберігати денормалізовані дані, що особливо зручно для роботи зі складними вкладеними структурами.

Так, MySQL також має підтримку JSON, проте вона не така гнучка: неможливо частково оновити вкладені поля, є проблеми з індексацією, а структура швидко стає громіздкою при зростанні складності даних.

MongoDB, навпаки, відмінно масштабується. Вона підтримує реплікацію, шардинг та інші механізми масштабування прямо з коробки. Це робить її ідеальним вибором для систем із високим навантаженням і динамічно змінною структурою даних. Але тут теж є компроміси: наприклад, транзакції, хоч і з'явилися, досі обмежені й працюють лише в межах одного вузла чи документа. Також потрібно змиритися з відсутністю нормалізації й самим відповідати за узгодженість даних між документами[9].

Якщо ж проаналізувати, які саме дані зберігатимуться у нашій системі, то можна впевнено сказати: MongoDB — це ідеальний вибір для зберігання медіатеки, яку ми отримуємо зі Spotify API. Денормалізована структура дозволить не ускладнювати модель, відповідальність за консистентність лежить на зовнішніх джерелах, а потреба в транзакціях тут мінімальна.

Окрім задачі постійного зберігання даних, важливою ціллю є забезпечення зручної та гнучкої навігації по медіатеці та пошуку для кінцевих користувачів. Після вибору основної бази даних — MongoDB, логічним було б використати вбудований текстовий індекс, який Mongo надає з коробки. Він дійсно працює: розбиває текст на терми та зберігає їх у так званому зворотному індексі, що дозволяє здійснювати пошук по текстовому вмісту документів.

Але, незважаючи на зовнішню простоту, вбудований пошук у MongoDB має низку серйозних обмежень, які не дозволяють повноцінно реалізувати вимоги до нашої системи.

По-перше, це погана підтримка української мови. У Mongo відсутня якісна морфологічна обробка для української, що унеможливлює ефективний пошук українських виконавців, пісень чи альбомів — слова не розпізнаються у різних формах, немає стемінгу та адекватного розбиття на терми.

По-друге, Mongo не дозволяє гнучко налаштовувати логіку пошуку. Неможливо задати, як саме має аналізуватися текст, які токенізатори чи фільтри застосовуються. Через це реалізувати пошук з автодоповненням, з урахуванням опечаток чи з пріоритезацією результатів практично неможливо.

Для вирішення таких задач використовуються спеціалізовані пошукові рушії, і в нашому випадку вибір зупинився на Elasticsearch.

Elasticsearch — це потужна система повнотекстового пошуку на базі Apache Lucene. Вона зберігає дані у вигляді документів і дозволяє виконувати надзвичайно швидкі та точні пошукові запити. Головною перевагою Elasticsearch є підтримка аналізаторів — інструментів, які дозволяють детально налаштовувати процес обробки тексту: від розбиття на слова до фільтрації, нормалізації й врахування мовних особливостей. Є можливість як використовувати готові аналізатори (зокрема й для української), так і створювати власні.

У рамках реалізації пошукового функціоналу, описаного в третьому розділі, буде використано n-gram аналізатор, який розбиває слова на підрядки — це дозволить реалізувати пошук з автодоповненням, який оновлює результати в реальному часі під час введення тексту.

Крім того, Elasticsearch добре масштабується — його легко масштабувати як вертикально, так і горизонтально: через шардування, реплікацію та розподілення навантаження.

Нарешті, розділення основної бази (MongoDB) і пошукової системи (Elasticsearch) дозволяє незалежно масштабувати та оптимізувати кожну з них під свої задачі: Mongo відповідає за зберігання складних структур з медіа-даними, а Elasticsearch — за ефективний повнотекстовий пошук по цим даним.

У пункті, присвяченому проєктуванню самих сервісів, буде детально описано процес проєктування баз даних, їх структура, а також налаштування, які дозволять ефективно взаємодіяти з даними в умовах навантаження, розподілу ролей сервісів та особливостей обраних СУБД.

2.4 Взаємодія між сервісами

Після того як було визначено склад мікросервісів та їх функціональні зони відповідальності, постає наступне ключове питання. Правильно обраний спосіб комунікації безпосередньо впливає на стабільність, продуктивність тамасштабованість усієї системи. У сучасних розподілених системах існує два основні підходи до взаємодії між компонентами — синхронний та асинхронний.

Синхронна взаємодія гарантує суворий порядок виконання інструкцій. Для нашої системи це означає, що сервіс не переходить до наступного кроку, поки не отримає відповідь на надісланий запит.

Асинхронна взаємодія, навпаки, не вимагає очікування відповіді — запит або повідомлення відправляється, після чого програма продовжує роботу. Це є дуже корисним для реалізації Pub/Sub: ми гарантує лише відправку нашого повідомлення, чи отримає хоч якась система його - не важливо. Якщо ж сервіс відповідає на запит, то повідомлення обробляється окремо. Такий підхід підвищує продуктивність та дозволяє досягти кращої масштабованості.

У цьому пункті буде розглянуто різні технології для реалізації обох типів взаємодії. Ми порівняємо їх сильні та слабкі сторони, і на основі цього оберемо найбільш відповідні рішення для нашої системи[9].

2.4.1 GRPC та REST

У багатьох сценаріях мікросервісної системи нам потрібно отримати конкретну відповідь на конкретний запит. Наприклад, коли ігрова сесія потребує інформацію про конкретний плейліст — її можна просто витягнути з основного сховища. Для таких задач не потрібне постійне з’єднання —

достатньо встановити зв’язок, звернутися до потрібного ресурсу, отримати відповідь і закрити з’єднання. Саме для цього чудово підходить HTTP.

HTTP — це основа сучасного інтернету. Це стандарт, який лежить в основі всіх взаємодій у мережі Інтернет. Він працює поверх протоколу TCP/IP, що гарантує доставку пакетів, дотримання порядку отримання даних та коректне з’єднання між клієнтом і сервером. Саме тому HTTP залишається універсальним, зрозумілим і добре підтримуваним вибором для побудови синхронних API.

Визначившись із тим, що ми будемо використовувати HTTP як транспорт, наступний крок — вибрати підхід до побудови самого API. Серед усіх варіантів розглянуті два найрелевантніші для нашого випадку: REST та gRPC.

REST

REST — це архітектурний стиль взаємодії в розподілених системах, який був запропонований Роєм Філдінгом у його дисертації в 2000 році. REST визначає набір обмежень, дотримання яких дозволяє створювати масштабовані, надійні й добре структуровані API поверх HTTP. REST не є протоколом чи стандартом у класичному сенсі, це архітектурна парадигма, яка дозволяє будувати API, використовуючи стандартні можливості HTTP (GET, POST, PUT, DELETE тощо) без додаткових надбудов. Для того щоб сервіс вважався RESTful, він має відповідати п’яти основним обмеженням:

- Модель клієнт-сервер: REST базується на чіткому розділенні ролей між клієнтом та сервером. Клієнт ініціює запити і не зберігає стан, а сервер відповідає на них та управляє бізнес-логікою і даними. Це розділення важливе не лише для масштабування, але й для розуміння наступних принципів REST.
- Відсутність стану: Кожен запит клієнта до сервера має містити всю інформацію, необхідну для його обробки. Сервер не зберігає ніякого контексту між запитами одного й того ж клієнта.

- Кешування: REST дозволяє відповідям бути позначеними як кешовані або ні. Це зменшує кількість запитів до сервера, підвищує продуктивність і знижує затримки. Кешування особливо важливе для нашої високонавантаженої системи, оскільки зменшує кількість запитів до оригінального застосунку або бази даних (залежить на якому рівні ми використовуємо кешування), тим самим зменшуючи навантаження та шанси на відказ, але при цьому жертвуючи узгодженістю.
- Єдинообразний інтерфейс: REST накладає обмеження на структуру взаємодії, що робить API передбачуваним, стандартизованим та простим у використанні. Всі ресурси повинні бути доступні через URI, а дії над ними — через стандартні HTTP-методи.
- Багаторівнева система: у REST дозволено впровадження проміжних шарів між клієнтом і сервером. Цей принцип критично важливий для нашого застосунку, оскільки вона є мікросервісною, ми не хочемо, щоб клієнт взаємодіяв із кожним мікросервісом напряму. Замість цього — клієнт звертається до Nginx, який приховує внутрішню структуру системи. Приклад: ми маємо ресурс, який можемо ідентифікувати за */api/v1/artists*. У нас є можливість звернутися до нього за допомогою *GET /api/v1/artists/{id}* та *GET /api/v1/artists/search?q={text}.* Клієнт сприймає це як один ресурс, хоча за API ховається додатковий шар, який відправляє запити на Core та Search відповідно.

У REST найчастіше використовується формат JSON для передачі інформації. Він є текстовим, людинозрозумілим, ідеально підходить для обробки на фронтенді та легко читається іншими сервісами або сторонніми інтеграторами. Також можуть використовуватись XML, YAML, але JSON є де-факто стандартом у сучасних веб-системах.

Перевага текстових форматів — у тому, що вони зручні як для розробників, так і для налагодження системи. Якщо API буде відкритим для

сторонніх розробників або інтеграцій, REST з JSON — це майже завжди найпростіший і найзрозуміліший вибір[10].

GRPC

gRPC — це сучасний фреймворк для виклику віддалених процедур (Remote Procedure Call), який був розроблений компанією Google у 2015 році. Основна його мета — забезпечити ефективну, масштабовану та типізовану взаємодію між сервісами. На відміну від REST, gRPC реалізує підхід не до роботи з ресурсами, а до безпосереднього виклику методів.

gRPC використовує як транспортний рівень протокол HTTP/2, що дає йому ряд переваг у порівнянні з HTTP/1.1, який зазвичай використовується у REST.

Крім транспортного рівня, важливо також відзначити, що gRPC використовує бінарний протокол обміну даними, а не текстовий. Це значно підвищує швидкість серіалізації та десеріалізації, зменшує розмір переданих повідомлень, але при цьому робить протокол нечитабельним для людини.

gRPC побудований на основі Protocol Buffers (Protobuf) — це мова опису структур даних, яка дозволяє створювати контракти між сервісами. Основна робота з protobuf зводиться до написання .proto файлів.

Після написання цих файлів вони компілюються спеціальним плагіном, який генерує вихідний код для обох сторін — клієнта й сервера — на потрібній мові програмування. Такий підхід дозволяє суворо контролювати типи, а також значно спрощує взаємодію між сервісами, навіть якщо вони написані на різних мовах.

Після аналізу двох основних підходів до побудови API, gRPC та REST, можна зробити висновок, що хоча gRPC дійсно має переваги у продуктивності та зручності типізованої взаємодії між мікросервісами, він не є універсальним рішенням. Його використання для запитів зі сторони фронтенду або браузера є практично неможливим без додаткових адаптацій. REST у цьому плані значно гнучкіший: одна й та сама реалізація може використовуватись як для

міжсервісної взаємодії, так і для клієнтських запитів з фронтенду або мобільних додатків.

2.4.2 Kafka та RabbitMQ

Іноді в процесі виконання задач виникає потреба відправити повідомлення не одному сервісу, а одразу кільком. Уявімо ситуацію: ми хочемо повідомити про певну подію декілька інших мікросервісів — наприклад, що з’явився новий трек у медіатеці. Якщо підходити до цього класично, через HTTP-запити, нам доведеться послідовно звернутись до кожного сервісу та дочекатись відповіді, що може займати багато часу й сильно впливати на загальну продуктивність системи.

Проте в більшості подібних випадків відповідь нас не цікавить взагалі. Ми не хочемо отримати якусь інформацію — ми просто хочемо сповістити про те, що щось відбулось. Тут і приходить на допомогу асинхронна взаємодія: ми публікуємо подію — і йдемо далі. Підписники отримають її, коли зможуть, і оброблять у себе.

Однак така гнучкість приносить із собою нову складність — гарантію доставки. Якщо повідомлення втратиться — ми не дізнаємось про це. Якщо підписник був тимчасово недоступний — подія може не потрапити до нього взагалі. Щоб уникнути цих проблем, використовуються брокери повідомлень.

Брокер повідомлень — це проміжне ПЗ, яке приймає повідомлення від одного сервісу та доставляє їх одному або кільком іншим сервісам, гарантуючи черговість, повторні спроби доставки, збереження у черзі та інші важливі механізми[11].

Проведено аналіз двох найпопулярніших рішень: Kafka та RabbitMQ. Розглянуто їх переваги, недоліки, а також обґрунтовано вибір найкращого варіанту саме для нашої системи.

RabbitMQ

RabbitMQ — один із найпопулярніших брокерів повідомлень. Спочатку розроблений компанією Pivotal (нині частина VMware), написаний мовою

Erlang, що забезпечує високу надійність і стійкість до збоїв. RabbitMQ підтримує велику кількість протоколів обміну повідомленнями, але для нашої системи найбільш цікаві — це AMQP та STOMP.

AMQP - це бінарний протокол, який використовується RabbitMQ за замовчуванням. Його архітектура дуже добре підходить для реалізації Pub/Sub.

STOMP - це текстовий протокол, який можна обгорнути у WebSocket і використовувати на клієнті. Це є ідеальним варіантом для real-time повідомлень клієнтів щодо процесу імпорту їх плейлістів, що є одною з задач цієї роботи.

RabbitMQ має хорошу архітектуру, але з одним важливим обмеженням: масштабування на запис практично неможливе. Причина в тому, що черги завжди прив’язані до конкретного вузла, і всі повідомлення на запис мають потрапляти саме туди. Навіть у режимі реплікації (HA) запис залишається централізованим.

Kafka

Apache Kafka — це розподілена платформа для потокової передачі даних з відкритим кодом, яку спочатку розробили в LinkedIn, а згодом передали до Apache Software Foundation. Вона призначена для обробки великих обсягів даних у реальному часі.

Архітектура Kafka досить проста і навіть примітивна у порівнянні з RabbitMQ: відсутність аналогів exchange та bindings, додаткових функцій, наприклад, топіків смерті, куди потрапляють повідомлення, які консьюмер не зміг обробити. Відповідальність за увесь цей фунціона лягає на самого розробника, що збільшує час розробки.

Важливо зазначити, що повідомлення у Kafka можуть зберігатися дуже довго — час зберігання можна налаштовувати, або взагалі зберігати безстроково. Це робить Kafka схожою на журнал подій або систему логування.

На відміну від RabbitMQ, де повідомлення видаляються після обробки, Kafka зберігає всю історію повідомлень.

Незважаючи на доволі примітивну архітектуру, Kafka є однією з найкращих систем для масштабування. Завдяки партиціям топік можна розбити на десятки чи сотні частин, які обробляються паралельно, що дозволяє обробляти мільйони повідомлень за секунду. Це робить Kafka ідеальним рішенням для високонавантажених проєктів з великим потоком даних[12].

Незважаючи на всі переваги Kafka та її чудову здатність масштабуватись, для нашої системи прийнято рішення використовувати RabbitMQ. Основна причина більш розширений набір функцій, який дозволяє уникнути потреби реалізовувати багато речей самостійно. Такі речі, як черги з повторними спробами, маршрутизація та підтримка STOMP — усе це вже є в RabbitMQ і дуже просто налаштовується. Хоча RabbitMQ і має гіршу масштабованість на запис, цього буде більш ніж достатньо навіть для великих навантажень, які очікуються в нашій системі. До того ж, при потребі забезпечити високу доступність та цілісність даних, можна буде реалізувати кластеризацію у режимі High Availability, що забезпечить реплікацію черг на кілька вузлів та стійкість до збоїв.

2.5 Проектування сервісів

Після того, як ми визначили ключові технології, розбили систему на окремі сервіси та чітко розмежували їхні функції, настав час перейти до проєктування самих сервісів. У цьому розділі буде детально описано структуру кожного сервісу, а також принципи їхньої роботи і взаємодії.

2.5.1 Song Parser

Проектування сервісу Song Parser було розпочате з огляду на його критичну роль у системі — саме він відповідає за наповнення нашої медіатеки через зовнішні API, зокрема Spotify API та YouTube Data API. Вся логіка сервісу розбита на чотири основні компоненти: взаємодія з API, команди, обробники команд та командна шина.

Взаємодія з зовнішніми сервісами реалізована через HTTP-клієнт, який отримує дані, серіалізує їх у внутрішній формат і передає в RabbitMQ через подієву шину Symfony. Сам процес парсингу артистів реалізований як послідовність окремих етапів.

Компонент Scheduler від Symfony щотижня ініціює оновлення інформації про 10 000 артистів. Кожен артист відправляється у чергу та обробляється незалежними воркерами. Воркери отримують дані про артиста зі Spotify API, зберігають його аватар у S3 (що згодом буде використано в Core-сервісі) та надсилають інформацію до черги на збереження у Core. Після підтвердження збереження артиста сервіс отримує список відсутніх альбомів, за якими знову звертається до Spotify API, і повторює процес: серіалізація → черга → Core.

На завершальному етапі, коли Core повертає підтвердження про збереження альбому, повідомлення містить список вже існуючих треків, аби уникнути дублювання. Далі відбувається побудова запиту до YouTube API у вигляді “Ім’я артиста - Назва треку (Official Audio)”. За отриманим результатом за допомогою утиліти yt-dlp завантажується аудіо та зберігається в S3. Після цього трек з відповідними метаданими надсилається в RabbitMQ для остаточного збереження у Core.

Ще однією важливою задачею сервісу Song Parser є парсинг плейлистів, який реалізований за тим самим принципом, що й основний процес обробки артистів, альбомів та треків. Основна різниця полягає в тому, що тут необхідно повідомляти користувача про перебіг обробки та завершення кожного з етапів. Для цього використовується протокол STOMP, обгорнутий у WebSocket-з’єднання. Завдяки цьому протоколу ми можемо в реальному часі надсилати повідомлення клієнту.

Головним зв'язуючим елементом у всьому процесі парсингу є RabbitMQ — саме він забезпечує ефективну, масштабовану та надійну доставку повідомлень між сервісами. Щоб досягти максимальної ефективності та

продуктивності системи, потрібно відповідально підійти до проєктування схеми маршрутизації повідомлень усередині брокера.

Основна мета — забезпечити рівномірне балансування навантаження між воркерами. Проте просте випадкове розподілення повідомлень між чергами може спричинити проблему з конкуренцією: менш актуальне повідомлення може обробитися раніше за більш пріоритетне, якщо воно потрапить до менш завантаженої черги. Щоб уникнути подібної ситуації, у RabbitMQ передбачено спеціальний тип обмінника — x-modulus-hash, який входить до офіційної бібліотеки rabbitmq_stream. Даний обмінник хешує routing-key повідомлення і за модулем розподіляє його в одну з доступних черг. Такий підхід гарантує, що всі повідомлення з однаковим routing-key (у нашому випадку це унікальний ідентифікатор сутності) потрапляють у ту саму чергу, тим самим забезпечуючи послідовну обробку та відсутність конфліктів.

У рамках проекту було прийнято рішення створити по 4 черги для кожної операції — обробки артистів, альбомів та плейлістів. Винятком стали треки, адже на цьому етапі виконується найбільш ресурсомістка операція — завантаження аудіо через утиліту yt-dlp[13]. Ця утиліта працює синхронно, блокує потік на деякий час і, відповідно, сповільнює загальну обробку. Тому для треків було створено одразу 16 черг, що дозволяє обробляти запити паралельно на більшій кількості воркерів і запобігає утворенню вузького місця у системі. Схематику RabbitMQ подано у додатку 2.

Окрім правильно налаштованої маршрутизації, дуже важливо забезпечити послідовну обробку повідомлень, що стосуються однієї й тієї ж сутності. Для цього під час надсилання повідомлень було прийнято рішення не видаляти повідомлення з черги одразу після отримання, а дозволити повторну обробку у разі невдалої спроби. Такий підхід дає змогу гарантувати, що важливі дані не будуть втрачені через тимчасові збої — наприклад, якщо сторонній API тимчасово недоступний. Після відновлення роботи API воркер зможе знову отримати повідомлення й успішно його обробити.

Це рішення є природним для обробки артистів та альбомів, адже для них характерна саме така категорія помилок — тимчасова недоступність стороннього сервісу. Проте ситуація з треками дещо складніша. У цьому випадку можливі такі помилки, які не зникнуть із часом. Один із яскравих прикладів — відео з віковими обмеженнями 18+, які утиліта yt-dlp не зможе завантажити без спеціальних обхідних механізмів. Якщо продовжити намагатися обробити таке повідомлення, ми лише заблокуємо всю чергу, і жоден інший трек не зможе бути оброблений, поки ця проблема не буде усунена вручну.

Щоб уникнути такого блокування, для обробки треків було реалізовано механізм перенаправлення невдалих повідомлень у спеціальну мертву. У разі виявлення критичної помилки воркер скасовує поточну обробку, не намагаючись повторити її нескінченно, й перенаправляє повідомлення до DLQ. Далі це повідомлення може бути зафіксоване в логах для подальшого аналізу, або ж, якщо обробка відбувалася у рамках збору плейлісту користувача — надіслано відповідне повідомлення з помилкою назад через STOMP.

Більше деталей щодо реалізації цього сервісу буде описано у третьому розділі роботи.

2.5.2 Core

Після того як була спроектована архітектура сервісу парсингу медіатеки, логічним наступним кроком стала реалізація центрального елементу всієї системи — сервісу Core. Саме він виконує роль головного сховища всієї інформації, надає її зовнішнім споживачам через REST API та відповідає за реплікацію даних на інші сервіси, зокрема на Search.

Основна взаємодія з сервісом Core відбувається асинхронно. Дані надходять у вигляді повідомлень із RabbitMQ, де оброблюються відповідними воркерами. Після обробки повідомлення, дані зберігаються у MongoDB. Завдяки механізму тригерів на зміну в базі, одразу після запису нової сутності

формується подальше повідомлення у чергу — наприклад, для продовження процесу парсингу або реплікації до Search-сервісу.

Кожна сутність у сервісі Core має окрему колекцію в MongoDB, що дозволяє зберігати дані структуровано та оптимізовано під специфіку запитів.

Артист (Artist) описується наступними полями: унікальний ідентифікатор, ім’я, назва аватару в S3-сховищі, масив жанрів та джерело у форматі "сервіс:ідентифікатор_у_сервісі"(наприклад,"spotify:0xByDfltDVpk6LDsUMHyI2").

Альбом (Album) містить ідентифікатор, назву, ім’я файлу обкладинки у S3, дату релізу, а також масив спрощених артистів. Під «спрощеними» розуміються артисти, для яких зберігається лише ім’я та джерело. Це дозволяє оперувати навіть не до кінця спаршеними даними — у разі, якщо відповідний артист ще не існує в системі, його ім’я все одно буде збережено. Якщо ж артист уже присутній у колекції артистів, встановлюється зв'язок між ними на основі значення джерела.

Трек (Track) складається з ідентифікатора, назви, імені аудіофайлу в S3, джерела та масиву спрощених артистів, аналогічно до альбомів.

Плейлист (Playlist) включає в себе ідентифікатор, назву, ім’я обкладинки у S3, джерело, ідентифікатор операції, у рамках якої був створений плейлист (це детальніше розкривається у розділі 3), опис і масив ідентифікаторів треків, що входять до нього.

У випадках, коли потрібне з’єднання між різними колекціями — наприклад, для підвантаження повної інформації про артистів у альбомі — використовується механізм агрегації MongoDB з оператором $lookup. Це дозволяє ефективно виконувати join-подібні операції між колекціями, зберігаючи при цьому гнучкість NoSQL-моделі.

У додатку 3-6 представлено схематичну структуру кожної з сутностей у MongoDB — включно з артистами, альбомами, треками та плейлістами.

У третьому розділі описано конкретну реалізацію предметної області, API та роботу з шиною подій.

2.5.3 Search

Однією з задач, поставлених перед системою, є реалізація повнотекстового пошуку з підтримкою автодоповнення та механізмів релевантного сортування. Для цього було спроєктовано окремий сервіс — Search, який побудовано на основі пошукового рушія Elasticsearch. Основна його мета — забезпечити швидкий і якісний пошук по медіаконтенту з урахуванням можливих помилок у введенні, морфології мови та очікувань користувача.

Згідно з архітектурою, Search-сервіс отримує дані з черги RabbitMQ, в яку відправляються події від сервісу Core під час створення або оновлення сутностей. Таким чином досягається асинхронна синхронізація між основним сховищем даних і пошуковим індексом.

Варто підкреслити, що Elasticsearch, як документо-орієнтований пошуковий рушій, не підтримує операції, подібні до join або $lookup, які є у MongoDB. Тому вся необхідна інформація дублюється безпосередньо в межах одного документа. Наприклад, документ альбому включає не лише свої поля, а й імена артистів, а документ плейліста — базову інформацію про треки. Це дозволяє проводити швидкий пошук без необхідності звернення до інших сутностей.

Також важливо зазначити, що структура даних у Elasticsearch є спрощеною в порівнянні з Core, адже ціль полягає не в зберіганні всіх атрибутів, а в наданні стислого результату, достатнього для попереднього відображення у користувацькому інтерфейсі. Усі додаткові дані (наприклад, повна інформація про трек чи альбом) можуть бути запитані окремо безпосередньо з Core-сервісу.

У додатках 7–8 буде представлено мапінг документів у Elasticsearch, а в наступному розділі буде детально описано процес токенізації (розбиття на

терми), налаштування аналізаторів та логіку формування запиту, яка лежить в основі реалізації пошуку з урахуванням релевантності.

2.5.4 MatchMaker

Сервіс MatchMaker відіграє надзвичайно важливу роль у маршрутизації гравців до відповідних ігрових сесій. Стандартні рішення на кшталт Nginx чи балансувальників навантаження виявляються занадто примітивними для таких цілей — вони не дозволяють ефективно відстежувати стан кожної ігрової кімнати, контролювати її доступність або забезпечувати підключення за приватним ключем. Саме для цього й був спроектований окремий сервіс MatchMaker.

Основна мета сервісу — керування Docker-контейнерами, всередині яких запускаються окремі ігрові сесії. Кожен контейнер містить HTTP-сервер, що дозволяє підняти WebSocket-з'єднання та має endpoint для перевірки свого статусу — чи готова сесія приймати нових гравців, чи вже зайнята. Кожна незайнята сесія надсилає heartbeat-запит до Redis кожні 5 секунд, оновлюючи інформацію про свій стан: ідентифікатор контейнера , скільки стоїть без діла. MatchMaker постійно працює з Redis і завжди має актуальну картину: які сесії доступні, які вже використовуються, а які можна призначити новим гравцям.

Коли надходить запит на підключення до сесії або створення нової, MatchMaker приймає його, аналізує поточний стан і або надає адресу вже існуючої вільної кімнати, або піднімає нову. Це дозволяє мінімізувати затримки при старті ігрового процесу та забезпечити швидке підключення гравця.

Окрему увагу в проєктуванні приділено автоскейлінгу. MatchMaker завжди підтримує певну кількість активних (резервних) контейнерів, які можуть бути використані негайно при зростанні навантаження. Якщо кількість вільних контейнерів падає нижче певного порогу — запускаються нові. Якщо ж навпаки — система починає простоювати, MatchMaker зупиняє частину контейнерів, щоб зекономити ресурси.

2.5.5 Game

Час перейти до найважливішого сервісу всієї архітектури — Game. Усі попередні модулі були створені саме для підтримки та обслуговування цього компонента. Саме тут відбувається ігровий процес у реальному часі. В якості основи для архітектурного рішення був проаналізований сервіс Woogles.io. Після детального аналізу вирішено наслідувати її загальну ідею, але уникнути низки помилок, притаманних оригінальній реалізації.

Одразу було прийнято рішення використовувати WebSocket як основний протокол для двостороннього real-time зв’язку між клієнтом і сервером. Кожен екземпляр ігрового сервера є stateful — тобто тримає усі дані сесії в пам’яті — і обслуговує лише одну ігрову сесію. Такий підхід дозволяє повністю ізолювати сесії одна від одної.

Важливою проблемою, яку було помічено у Woogles.io, є сильна зв’язаність WebSocket-протоколу з доменною логікою гри. Це ускладнює розуміння моделі, робить підтримку проєкту важкою та заважає повторному використанню логіки гри. Щоб цього уникнути, ми поділили сервіс на два рівні: шар представлення (WebSocket-з’єднання) та доменну модель (безпосередньо механіка гри).

Архітектура гри побудована на таких основних компонентах:

- Game — головний об’єкт-медіатор, який координує всі інші частини ігрової логіки.
- Stage — окрема сцена або крок гри, яка відповідає за свою ізольовану логіку.
- StagePool — список доступних сцен, керує їх зміною та ініціалізацією. State — клас зі станами, які автоматично синхронізуються з клієнтом через EchoService.
- GlobalState — як і State, але зберігається протягом усієї гри, а не лише однієї сцени.

- EchoService — сервіс над WebSocket, який відповідає за надсилання повідомлень клієнтам.

Розглянемо коротко деякі ключові частини:

State — це звичайний клас із відкритими полями. Основна його особливість — здатність відстежувати зміни певних полів і автоматично синхронізувати їх з клієнтом. Таким чином, коли на сервері щось змінюється, клієнт отримує оновлення без жодного додаткового коду. Повідомлення про зміну відправляються через EchoService.

Stage — поточна сцена гри, яка має чітко визначені методи життєвого циклу, реагує на підключення/відключення гравців, приймає повідомлення і може запускати наступну сцену через StagePool. Вона також може мати власний State, який зникає після завершення сцени.

StagePool — простий менеджер черги сцен. Дозволяє зручно переходити від однієї сцени до іншої, змінювати їх порядок або повторно запускати.

Game — обгортає StagePool і глобальні події гри. Має методи, які викликаються до старту першої сцени й після завершення останньої

Окрім цих компонентів, сервіс містить низку допоміжних модулів, які спрощують розробку, але вони будуть описані окремо в наступному розділі. Окрім внутрішньої логіки, кожен ігровий контейнер періодично оновлює свій статус у Redis, надсилаючи ping-запити кожні 5 секунд. Це дозволяє сервісу MatchMaker знати, чи доступна сесія, і швидко маршрутизувати користувачів. Також кожен екземпляр має окремий HTTP endpoint, за допомогою якого можна перевірити, чи контейнер зайнятий.

ВИСНОВОК ДО РОЗДІЛУ

Розділ 2 став другим кроком після проведення аналізу та збору вимог у процесі розробки багатокористувацької гри з підтримкою взаємодії в реальному часі. Основним завданням цього етапу було визначення архітектурної моделі системи та вибір ключових технологій, які забезпечать масштабованість, надійність і гнучкість під час подальшого розвитку проєкту.

У межах розділу було обґрунтовано вибір мікросервісної архітектури як основного стилю побудови системи. Такий підхід дозволяє досягти високого рівня ізоляції між окремими компонентами, що, у свою чергу, спрощує масштабування, забезпечує стабільність у випадку збоїв окремих сервісів і сприяє незалежному оновленню частин системи.

Після цього було проведено аналіз мов програмування, які можуть бути використані для реалізації проєкту, та обґрунтовано вибір на користь PHP та JavaScript (TypeScript). Вибір цих мов зумовлений підтримкою розвиненої екосистеми, наявністю потужних інструментів, фреймворків і активного ком'юніті. Одночасно з цим було визначено основні бібліотеки й фреймворки, що значно пришвидшать розробку та покращать структуру програмного коду.

Окрему увагу було приділено питанню вибору бази даних. Було порівняно документо-орієнтовану MongoDB та реляційну MySQL. В результаті, враховуючи специфіку задач проєкту, перевага була віддана MongoDB, яка краще підходить для зберігання неструктурованих даних та забезпечує необхідну гнучкість для реалізації бізнес-логіки.

Для забезпечення міжсервісної взаємодії були обрані відповідні засоби синхронної та асинхронної комунікації. REST було обрано для реалізації HTTP-запитів до API, тоді як RabbitMQ став основним брокером повідомлень для обміну подіями між сервісами.

Заключним етапом цього розділу стала проектування окремих сервісів системи. Кожен сервіс отримав чітко визначену відповідальність, було описано його бізнес-логіку, взаємодію з іншими компонентами та загальні

принципи реалізації. Це створило фундамент для подальшої деталізованої реалізації системи, яка буде описана в наступних розділах.

РОЗДІЛ 3
РЕАЛІЗАЦІЇ

Після того як була сформована загальна архітектура системи, обрані технології, розділена логіка на окремі сервіси та розставлені зони відповідальності, настав наступний етап — реалізація. У цьому розділі буде докладно показано, як відбувався перехід від теорії до практики, як інтегрувалися черги повідомлень, бази даних, зовнішні API, внутрішні бібліотеки та фреймворки.

Через велику кількість деталей, нюансів та специфічних технічних рішень, не було сенсу описувати абсолютно все, замість цього увага буде приділена ключовим моментам, які вплинули на якість системи, її стабільність та відповідність початковим цілям. Опис буде побудований поетапно, у логічній черговості реалізації самих сервісів.

Першим кроком став Song Parser — сервіс, що працює з даними зовнішніх API. Саме з нього почалась практична розробка, оскільки він дозволяє одразу зіткнутись із реальністю, перевірити достовірність і повноту сторонньої документації. Досвід інтеграції подібних сервісів часто показує, що на 100% довіряти API — не найкраще рішення: кожна система має свої винятки, які не завжди описані в офіційних джерелах. Тому створення цього сервісу — це не лише про отримання інформації за URI та перенесення до бази, а й про глибоке занурення в структуру даних, з якою доведеться працювати всій системі.

Наступним був Core — центральне ядро, через яке проходить вся логіка. Після цього — Search, як функціонально близький та тісно пов’язаний сервіс. Поки ще свіжа пам’ять про внутрішні механізми Core, розробка схожого сервісу буде бистрішою.

Після цього в роботу пішла сама гра з real-time взаємодією на WebSocket, що стала найдинамічнішою частиною системи. Завершує цю серію сервісів Matchmaker — логіка, яка керує сесіями та з’єднує гравців у матчі.

В окремому підрозділі буде детально розглянуто проблеми, які виникали під час розробки. Частина з них була вирішена одразу, інші — відкладені з поясненнями і планом подальших дій. Розділ завершиться коротким підсумком, який дозволить ще раз оцінити досягнутий результат, співвіднести його з початковими завданнями та зрозуміти, наскільки ефективною виявилася обрана архітектура та обрані технічні рішення.

3.1 Деталі реалізації

У цьому розділі будуть приведені деталі реалізації сервісів. Акцент був зроблений на найцікавіших моментах.

3.1.1 Інтеграція командної шини у проект

У попередньому розділі вже згадувалося, що в сервісі Song Parser для обробки подій буде використовуватись брокер повідомлень. Після проведеного порівняння з Apache Kafka, вибір зупинився на RabbitMQ. Але тут одразу постає питання: як правильно його інтегрувати?

Найпростіший шлях — реалізувати всю логіку самостійно. Пряме підключення до брокера, реалізація механізмів публікації та споживання повідомлень, мінімальні залежності — звучить досить привабливо. Такий підхід дозволяє швидко стартувати й не потребує нічого, крім базового клієнта для AMQP-протоколу. Але це рішення має серйозні недоліки. Найбільша проблема — жорстка прив’язаність до конкретної технології брокеру повідомлень. Навіть якщо сховати реалізацію за абстрактними інтерфейсами, у реальності це все одно залишиться залежністю, яку важко замінити або протестувати. Окреме питання — тестування: перевірка повної логіки передачі повідомлення, обробки, маршрутизації — усе це стає або дуже складним, або взагалі неможливим без запуску реального брокера.

Щоб уникнути цих проблем, у системах, що працюють по принципу pub/sub, часто використовують патерн Message Bus. Це високорівнева абстракція, яка дозволяє працювати з повідомленнями на логічному рівні, не переймаючись деталями реалізації. У PHP існує декілька бібліотек, що

реалізують цей підхід, але найпопулярнішим рішенням є Symfony Messenger[14]. Його популярність настільки висока, що його використовують навіть у сторонніх фреймворках замість особистих реалізацій саме як універсальний інструмент для роботи з повідомленнями.

Messenger розділяється на кілька ключових компонентів:

- Sender — інтерфейс для відправки повідомлень у зовнішнє середовище
- Receiver — інтерфейс для отримання повідомлень
- Handler — логіка обробки повідомлень
- Envelope — обгортка навколо повідомлення, що дозволяє додавати мета-дані
- Аналогія з поштою: лист — це повідомлення, а конверт із марками — це Envelope
- Middleware — проміжні компоненти, що змінюють або логують повідомлення під час проходження через шину.

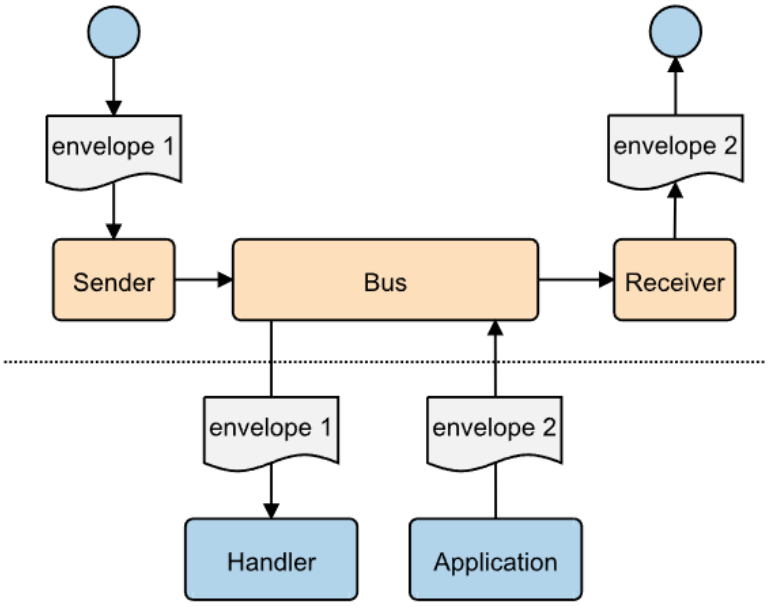


Рисунок 3.1 – Архітектура Symfony Messenger

Візуально архітектура Messenger виглядає дуже зріло та гнучко. Найголовніше — вона дозволяє повністю абстрагуватись від брокера. Можна перемикатися між синхронною та асинхронною логікою, не змінюючи

жодного рядка в бізнес-коді. Тестування також спрощується в рази: повідомлення можна запускати локально, без черг і сервісів, симулюючи повну обробку.

Symfony Messenger з коробки підтримує AMQP через свій транспорт. Транспорт — це логічне з'єднання Sender та Receiver, об'єднане в єдину конфігурацію. Але реалізація транспорту AMQP у Messenger має свої нюанси: функціональність доволі базова: відсутня конфігурація великої кількості опцій. Саме тому в рамках цієї роботи була реалізована власна версія транспорту, яка буде детально розглянута в підрозділі «Проблеми та їх вирішення».

Ще одна велика перевага Messenger — це гнучка маршрутизація та автоматична конфігурація. Класи-повідомлення не потребують жодних інтерфейсів або маркерів. Достатньо, щоб їхній обробник мав метод із відповідним параметром та помітку анотацією — система сама зрозуміє, яке повідомлення якому обробнику передати.

```php
#[AsMessageHandler]  WhtsPoint
final readonly class UpdateAlbumCommandHandler
{
    public function __construct(  WhtsPoint
        private TrackRepositoryInterface $trackRepository,
        private AlbumUpdater $updater,
        private CommandBusInterface $bus
    ) {
    }

    public function __invoke(UpdateAlbumCommand $command): void  no usages  WhtsPoint
    {
        $albumData = $command->dto;
        $album = $this->updater->fromDto($albumData);

        $this->bus->dispatch(new OnUpdateAlbumCommand(
            $album->getId(),
            $albumData->source,
            $this->trackRepository->getAllIdsByAlbum($album->getId())
        ));
    }
}
```

Рисунок 3.2 – Приклад класу-обробника

При цьому для самих повідомлень достатньо в файлі конфігурації вибрати конкретний транспорт.

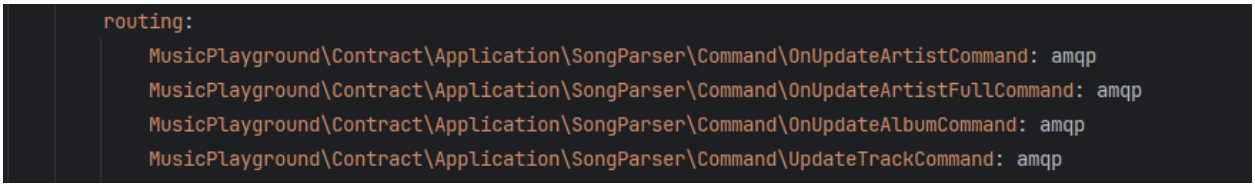


Рисунок 3.3 – Приклад конфігурації повідомлень

Важливо також зазначити, що Messenger використовується не лише в Song Parser, а і в інших сервісах системи. Окрім пришвидшення розробки, він виступає уніфікатором між сервісами — особливо в частині (де)сериалізації повідомлень. Тут ми стикаємось із ще однією складністю: у PHP не можна просто передати об’єкт у чергу. Необхідно перетворити його у формат, придатний для передачі — зазвичай це string. PHP надає функції serialize() та unserialize(), але вони мають важливе обмеження: щоб десеріалізація спрацювала, у приймаючому сервісі має бути повністю ідентичний клас із тим самим namespace та структурою.

Для вирішення цієї проблеми сумісності було прийнято рішення винести всі повідомлення, які виходять за межі одного сервісу, в окремий репозиторій контрактів і підключати його як окремий пакет через Composer. Composer дозволяє легко підключати сторонні репозиторії — достатньо лише правильно сконфігурувати composer.json як у головному проєкті, так і в самому пакеті.

Ключовим моментом є явне визначення namespace для кожного класу-повідомлення. Це дозволяє уникнути конфліктів і забезпечити стабільну десеріалізацію: якщо повідомлення серіалізоване у вигляді об'єкта, приймаючий сервіс повинен мати точну копію класу з таким самим неймспейсом. Завдяки підключеному пакету контрактів нам більше не потрібно дублікувати класи вручну чи постійно валідувати повідомлення — ми повністю дотримуємось принципу єдиного джерела істини.

Варто зазначити, що не всі повідомлення в системі передаються у вигляді серіалізованих об'єктів. У випадках, коли відбувається взаємодія між PHP та JavaScript, десеріалізувати PHP-клас у JS неможливо. У таких

ситуаціях використовується компонент Symfony Serializer, який за допомогою рефлексії автоматично перетворює об’єкт у звичайний JSON.

Подібна конфігурація командної шини зробила нашу систему не лише більш гнучкою, а й значно серйознішою з точки зору підтримки та тестування. Будь-які зміни в логіці маршрутизації або передачі повідомлень тепер повністю ізольовані від бізнес-логіки сервісів. Тестування стало простішим, адже повідомлення можна обробляти синхронно, не запускаючи брокер або повну інфраструктуру.

3.1.2 Інтеграція сторонніх API

Після того як була інтегрована командна шина, настав момент переходу до написання бізнес-логіки, а саме — до взаємодії з зовнішніми API. Першим із них, як і було заплановано, став Spotify API, який виконує роль основного джерела метаданих для треків, артистів, альбомів та плейлистів.

Перша ж проблема, яка одразу виникла під час реалізації — відсутність офіційного SDK для PHP. Через це було прийнято рішення реалізувати власну обмежену версію SDK, яка забезпечує необхідний функціонал для нашого проєкту. Цей SDK винесено в окремий репозиторій, що дозволяє його переюзати або модифікувати незалежно від основного коду. Реалізація охоплює роботу з доменами артистів, альбомів, треків і плейлистів, надаючи чіткі інтерфейси для зручної взаємодії.

Після створення SDK важливо було грамотно інтегрувати його в бізнес-логіку. Для цього було запроваджено патерн інтерфейсного репозиторію, який приховує конкретну реалізацію API. Таким чином, бізнес-логіка не залежить від Spotify напряму, а працює через абстракції. Усі відповіді від API мапляться у внутрішні сутності нашого домену — тобто, ми "підганяємо" дані ззовні під правила і структуру нашої системи.

Далі реалізовано механізм регулярного парсингу топ-10 000 артистів, ID яких зберігаються в окремому файлі. Парсинг виконується щотижня за допомогою механізму кронів, однак важливо зазначити, що ми не

використовуємо звичайний крон-сервіс. Було інтегровано компонент Symfony Scheduler, який дозволяє запускати завдання всередині нашої командної шини. Це значно зменшує залежність від середовища й робить запуск задач частиною загальної інфраструктури командної шини.

```php
final class UpdateAllArtistsSchedule implements ScheduleProviderInterface 1 usage  WhtsPoint *
{
    private Schedule $schedule; 1 usage

    public function getSchedule(): Schedule  no usages  WhtsPoint *
    {
        return $this->schedule ??= (new Schedule())
            ->with(
                RecurringMessage::every('1 weeks', new Envelope(new UpdateAllArtistsCommand(), [new BusNameStamp('messenger.bus.handle')])))
            );
    }
}
```

Рисунок 3.4 – Клас для створення регулярної оброки артистів

Кожен артист передається у чергу для подальшої обробки, що дозволяє не блокувати загальний процес і гарантує масштабовану обробку. Уся логіка подальшої роботи реалізована у вигляді послідовного ланцюга: отримали артиста → віддали в Core → підтвердили збереження → перейшли до альбомів і так далі.

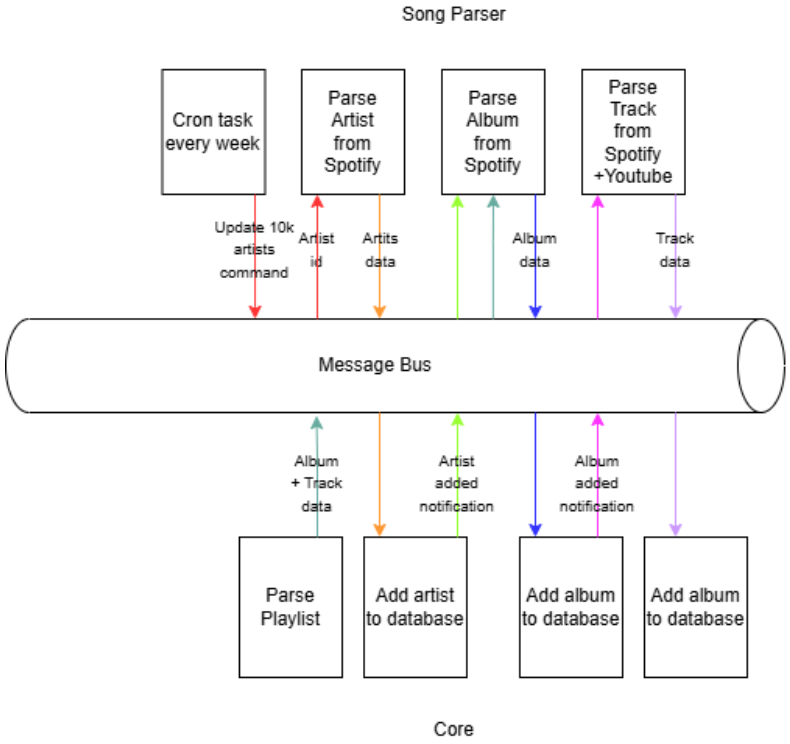


Рисунок 3.5 – Схема процесу парсингу

Окремо варто зупинитись на парсингу треків, адже це найбільш комплексна частина. Щоб отримати аудіофайл, необхідно виконати три основні кроки:

- Знайти трек за текстовим запитом на кшталт artist name - track name official audio.
- Отримати відео-ID треку.
- Завантажити аудіо і зберегти його в S3.

На цьому етапі ми звертаємося до YouTube Data API, який, на відміну від Spotify, має повноцінний SDK для PHP. Його було підключено через Composer, після чого використано функцію пошуку для отримання найбільш релевантного результату. Тепер у нас є ID відео, яке потенційно містить потрібний трек.

Як вже згадувалося в розділі 1, YouTube не надає можливості безпосереднього завантаження відео. Тут у гру вступає yt-dlp — популярна бібліотека для стримінгу. Це CLI-інструмент, тому ми інтегрували його безпосередньо в застосунок, викликаючи через shell_exec.

```php
    private function findVideoByName(string $name): string  1 usage  WhtsPoint
    {
        $response = $this->youtubeService->search->listSearch('snippet', [
            'maxResults' => 1,
            'order' => 'relevance',
            'q' => $name,
            'type' => 'video',
            'videoCategoryId' => 10,
        ]);

        $videoId = $response->getItems()[0]?->getId()->getVideoId();

        if ($videoId === null) {
            throw new RuntimeException('No video found for this track');
        }

        return $videoId;
    }

    private function downloadAudioFromVideo(string $fileName, string $videoId): string  1 usage  WhtsPoint
    {
        $filePath = "$this->filesPath/$fileName.mp3";

        shell_exec("yt-dlp -x --audio-format mp3 -o \"$filePath\" https://www.youtube.com/watch?v=$videoId");

        if (is_file($filePath) === false) {
            throw new RuntimeException('Bad attempt to download video');
        }

        return $filePath;
    }
}
```

Рисунок 3.6 – Методи з завантаженням треку

Проблема в тому, що такий запуск блокує I/O, що унеможливлює паралельну обробку великої кількості запитів. І саме тут розкривається потенціал RabbitMQ: ми просто масштабували обробку треків, створивши декілька черг і додавши більше воркерів (у поточній версії їх 16). Це дозволило ефективно завантажувати треки без серйозних затримок і з високою надійністю.

Після завантаження файл зберігається в Amazon S3, після чого дані про нього повертаються назад у Core для подальшої обробки. Також варто згадати про окремий цікавий процес — парсинг плейлистів, але він буде розглянутий у наступному підпункті. У другому підрозділі цього розділу ми детальніше поговоримо про типові проблеми API та способи їх вирішення, які були виявлені під час інтеграції.

3.1.3 Система відстеження асинхронного процесу в real-time

Уявімо ситуацію: користувач хоче завантажити власний плейлист у систему, щоб зіграти з ним у гру. Він надсилає HTTP-запит і, як зазвичай, очікує на відповідь з кодом 200 — підтвердження, що його запит успішно оброблено. Але у випадку з нашою системою все не так просто: процес парсингу плейлисту — асинхронний і довготривалий, він не вкладається у часові рамки одного HTTP-запиту, що неминуче призводить до тайм-ауту і переривання запиту.

Очевидно, що в таких умовах стандартний підхід — неефективний. Якщо проаналізувати логіку парсингу артистів, можна побачити схожість із парсингом плейлистів: обидва процеси асинхронні, запускаються через командну шину і мають схожі етапи обробки. Логічно, що і парсинг плейлистів також слід запускати через командну шину. Це розв'язує проблему тайм-ауту, але створює нову — як інформувати користувача про статус обробки?

На перший погляд, рішення очевидне — використовувати вебсокети. Ми можемо прослуховувати події з RabbitMQ і транслювати їх у вебсокет-канал,

який підписаний на конкретного користувача. Але цей підхід потребує створення окремого сервісу-проксі, який буде брати повідомлення з черги й передавати їх у браузер. Реалізація подібного рішення потребує додаткового часу, ресурсів та підтримки інфраструктури. Альтернативним, більш елегантним і простим варіантом є використання протоколу STOMP (Simple Text Oriented Messaging Protocol). Його перевага полягає в тому, що клієнт (тобто браузер) може напряму підключатись до RabbitMQ і підписуватись на повідомлення з черг без потреби в додатковому бекенд-сервісі.

Щоб активувати STOMP, достатньо встановити плагін rabbitmq_stomp і налаштувати конфігурацію для вхідних підключень, включно з автентифікацією, обмеженням доступу до певних черг та throttle-лімітами.

Коли користувач надсилає запит на парсинг плейліста, система генерує унікальний UUID і повертає його користувачу. Через нього клієнт може підписатися на відповідну чергу STOMP і в реальному часі отримувати повідомлення про етапи обробки. В системі реалізовано два ключові статуси: початок парсингу та додавання кожної окремої пісні. Завдяки такому рішенню забезпечується прозорий механізм зворотного зв’язку для користувача без потреби оновлювати сторінку чи очікувати завершення всього процесу. Приклад роботи цього механізму буде продемонстровано у розділі з тестуванням нашої системи.

3.1.4 Розробка Core

Сам по собі Core-сервіс не є складним і не містить великої кількості бізнес-логіки. Його основна задача — бути централізованим сховищем усієї медіатеки проєкту: зберігати дані про сутності (артистів, альбоми, треки, плейлісти) та надавати їх через API іншим сервісам. Водночас, навіть попри свою просту архітектуру, Core має низку технічних рішень, про які варто згадати окремо.

Після того як сервіс парсингу завершує обробку певної сутності, він надсилає повідомлення з усіма даними у чергу, де воно в подальшому

потрапляє в Core. Завдання Core — зберегти отриману сутність у базі даних. Як уже було вирішено раніше, головною базою даних у системі є MongoDB.

MongoDB надає офіційне PHP-розширення, яке відкриває повноцінний типізований SDK для взаємодії з базою: можна виконувати CRUD-операції, створювати колекції, працювати з індексами тощо. На перший погляд, цього достатньо, щоби розпочати роботу. Але при безпосередньому використанні SDK з'являється низка проблем:

- Відсутність мапінгу об'єктів — ми працюємо з класами, а MongoDB повертає сирі масиви. Це вимагає ручного мапінгу або використання кастомних гідраторів
- Проблеми з інкапсульованими полями — якщо об'єкти мають private/protected властивості, потрібно окремо вирішувати, як їх серіалізувати та десеріалізувати
- Відсутність подій — немає вбудованої можливості відстежувати вставки, оновлення, видалення, що важливо для механізму реплікації
- Відсутність опису схеми — інформацію про колекції, їх поля, індекси, доводиться вести вручну

Реалізація всіх цих механізмів власноруч потребувала би значних зусиль і часу, а результат не гарантував би стабільності чи гнучкості. Щоб уникнути цього, було вирішено використати ODM, аналог ORM, але для документоорієнтованих баз даних. Symfony не вимагає використання конкретного ODM, але найбільш поширеним та офіційно інтегрованим є Doctrine MongoDB ODM — інструмент, який дозволяє описувати структуру об'єктів у вигляді анотацій, XML або YAML, і повністю бере на себе перетворення між документами в Mongo та PHP-об'єктами.

```xml
<doctrine-mongo-mapping xmlns="http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping"
                        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xsi:schemaLocation="http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping
                    http://doctrine-project.org/schemas/odm/doctrine-mongo-mapping.xsd"
>
    <document name="App\Core\Domain\Entity\Album" db="core" collection="albums">
        <id />
        <field field-name="name" type="string" />
        <field field-name="coverId" type="string" />
        <field field-name="genres" type="collection" />
        <field field-name="source" type="id_source" />
        <embed-many target-document="App\Core\Domain\Entity\SimpleArtist" field="artists" />
        <field field-name="releaseDate" type="date_immutable" />
        <indexes>
            <index unique="true">
                <key name="source" />
            </index>
            <index>
                <key name="artists.source" />
            </index>
        </indexes>
    </document>
</doctrine-mongo-mapping>
```

Рисунок 3.7 – Приклад мапінгу однієї з сутностей

Таким чином, завдяки Doctrine ODM, ми отримали готовий і стабільний інструмент для роботи з MongoDB, що значно спростив реалізацію Core-сервісу. Це дозволило зосередитися на логіці роботи самої системи, не витрачаючи зайвий час на низькорівневу реалізацію інфраструктурних компонентів.

Хоча Doctrine ODM значно спрощує роботу з MongoDB, варто пам’ятати, що це все одно конкретна реалізація доступу до даних, яка створює залежність між нашою доменною моделлю та інфраструктурою зберігання. Для того щоб зменшити цей зв’язок і забезпечити більшу гнучкість системи, було вирішено застосувати паттерн Repository з DDD[7].

Repository створює абстрактний шар між доменною логікою та інфраструктурним кодом, дозволяючи змінити механізм збереження без значного впливу на бізнес-логіку. Усі запити до бази даних тепер йдуть через спеціально створені класи репозиторіїв, які інкапсулюють логіку читання та збереження даних. Таким чином, наша система стає менш залежною від Doctrine ODM і водночас набуває кращої тестованості та масштабованості.

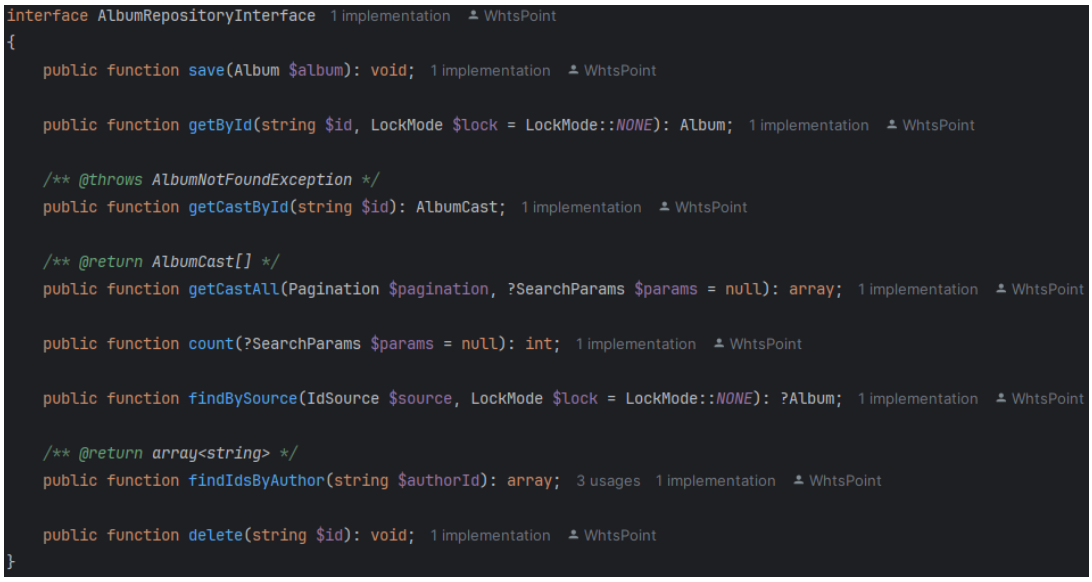


Рисунок 3.8 – Приклад інтерфейсу репозиторію

Тепер, після прийняття повідомлення з даними, достатньо скористатися відповідним репозиторієм, щоб зберегти сутність у базі даних. Це дозволяє ізолювати логіку збереження від самої обробки повідомлення та спростити зміну інфраструктурних рішень у майбутньому.

Якщо ж повідомлення є частиною процесу парсингу, після успішного збереження сутності в базі потрібно надіслати нове повідомлення до Song Parser, щоб продовжити ланцюжок. Таким чином, ми реалізуємо поетапну обробку з використанням черг, що дозволяє легко масштабувати систему та гарантувати цілісність даних на кожному кроці.

Після реалізації збереження даних необхідно було надати доступ до них через REST API, як для зовнішніх клієнтів, так і для внутрішніх сервісів. Symfony має зручну систему маршрутизації та HTTP-контролерів, що дозволяє легко налаштувати API. Для цього достатньо створити окремий клас-контролер, який відповідає за зв'язок між HTTP-запитами та бізнес-логікою, і позначити його методи анотаціями, що описують маршрути. Symfony автоматично зареєструє ці маршрути та забезпечить їхнє коректне оброблення.

Детальну документацію до кожного з реалізованих REST-ендпоінтів можна буде переглянути за допомогою Swagger UI, який буде реалізований у фінальному розділі.

3.1.5 Реалізація пошуку з автодоповненням

Після того як для реалізації пошукової системи в межах медіатеки було обрано Elasticsearch, першим кроком стала необхідність детально ознайомитися з, мабуть, найважливішим компонентом цього інструменту — аналізатором. Саме він відповідає за те, як текст буде підготовлений до індексації, розбитий на складові частини, і зрештою, як ефективно працюватиме пошук.

Ключовим поняттям у контексті Elasticsearch є зворотний індекс. Це структура, яка дозволяє дуже швидко знайти всі документи, що містять певне слово або фрагмент. Інакше кажучи, замість зберігання списку документів як у класичних реляційних базах даних, Elasticsearch створює словник, де кожен термін (тобто слово або токен) вказує, у яких саме документах він зустрічається.

Сам аналізатор в Elasticsearch складається з трьох частин: фільтр символів (може бути декілька), токенізатор (лише один) і фільтри токенів (також необмежена кількість). На першому етапі фільтри символів змінюють вхідний текст — наприклад, видаляють або замінюють символи, які не мають значення для пошуку (типу розділових знаків, спецсимволів тощо). Далі йде токенізатор — головна частина, яка розбиває текст на менші одиниці (токени), зазвичай це окремі слова або словоформи.

Завершальний етап — фільтри токенів. Вони відповідають за подальшу трансформацію вже розбитих елементів: нормалізацію до нижнього регістру, видалення стоп-слів, обмеження довжини, синонімію тощо. Саме ці фільтри дозволяють привести всі терміни до одного вигляду, щоб при пошуку користувач отримував максимально точні й релевантні результати незалежно від регістру, мови чи граматичної форми введених слів[15].

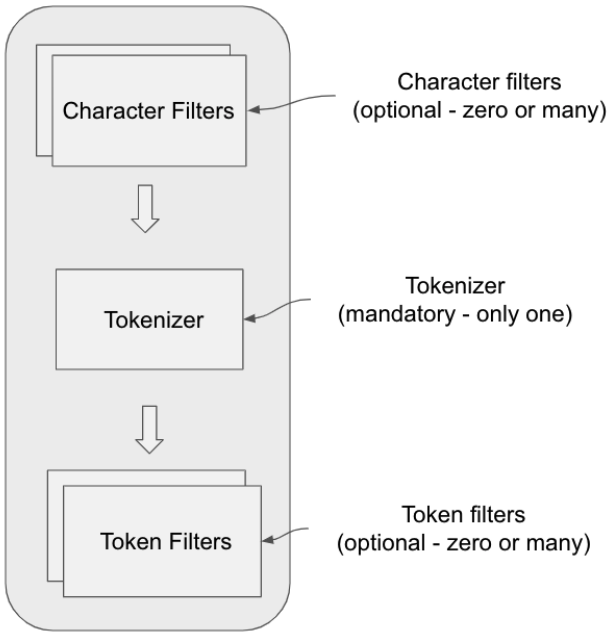


Рисунок 3.9 – логічна схема обробки тексту в Elasticsearch

Наша задача полягає у створенні пошуку з автодоповненням. Для кожної сутності — артистів, альбомів, плейлистів, треків — використовується однаковий аналізатор, який працює за однаковою логікою: для артистів це ім’я або псевдонім, для альбомів, плейлистів і треків — назва. На вхід системи подається саме назва або ім’я, тому додаткові фільтри не потрібні й будуть опущені.

Що стосується аналізатора, то найприродніше було б обрати n-gram, який розбиває текст на послідовні підрядки фіксованої довжини. Однак така логіка має недолік: розбиття відбувається по всьому рядку, а не по окремих словах. Наприклад, для назви альбому Teen Dream n-gram генерує токени типу «n D», « Dr», «en » тощо, що не завжди відповідає очікуваному пошуку.

Для уникнення цієї проблеми доцільно застосувати стандартний аналізатор, який спочатку розбиває текст на окремі слова. Далі завдання розбиття кожного слова на підрядки перекладається на спеціальний токен-фільтр, який формує n-gram без змішування слів між собою. Таким чином, ми зберігаємо логіку автодоповнення, уникаючи хаотичного формування токенів, що ускладнюють пошук.

При цьому всі токени приводяться до нижнього регістру, щоб пошук не залежав від регістру введених даних та був більш гнучким і зручним для користувачів.

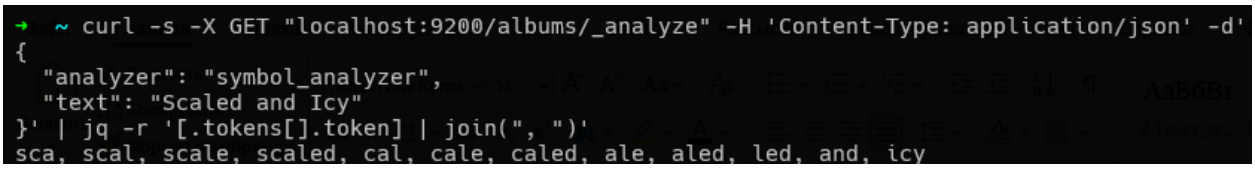


Рисунок 3.10 – Приклад роботи аналізатору

Під час формування запиту в Elasticsearch з умовою match, текст, що вводиться користувачем, проходить той самий процес аналізу, що й під час індексації даних. Це означає, що пошук відбувається не за повними значеннями, а за окремими токенами, і якщо хоча б один з них збігається з тими, що зберігаються в індексі, система повертає відповідний результат.

Однак важливо розуміти, що для користувача цінність становить не лише наявність збігів, а й їхня релевантність — тобто наскільки результати схожі на введений запит. Для цього в Elasticsearch реалізовано механізм оцінки релевантності. Кожен документ у результатах пошуку отримує спеціальне поле _score, яке містить кількість балів, нарахованих відповідно до внутрішніх критеріїв системи. Це дозволяє повертати результати у порядку спадання схожості, надаючи користувачу найбільш релевантні варіанти в першу чергу. Припустімо, що користувач хоче знайти треки або альбоми певного артиста. У цьому випадку він вводить ім’я артиста, однак у відповідь отримує лише сам об’єкт артиста — без треків чи альбомів. Це пов’язано з тим, що у базовій моделі даних, яку зберігає Core-сервіс, альбоми та плейлисти не містять посилань на імена виконавців у своїй структурі.

На перший погляд може здатися, що Elasticsearch має механізм, аналогічний до JOIN у SQL або $lookup у MongoDB. Проте насправді таких можливостей у нього немає. Саме тому виникає потреба у денормалізації даних — тобто дублюванні певної інформації. У нашому випадку це означає, що у документах альбомів і плейлистів слід зберігати також імена виконавців.

Це дозволить здійснювати пошук серед цих сутностей за ім’ям артиста, використовуючи той самий аналізатор, що й для основного індексу.

Зрозуміло, що зберігати повну інформацію про артиста у кожному пов’язаному документі недоцільно — це займе багато місця та ускладнить оновлення даних. Тому ми обмежимось збереженням лише необхідного мінімуму — того, що потрібно для відображення результатів пошуку. Це рішення зменшує використання пам’яті, проте призводить до складнішого процесу повторного обчислення даних (recalculate), якщо структура артиста буде розширена.

Ще один важливий момент — контроль ваги впливу певних полів на загальну релевантність. Наприклад, якщо збіг відбувся лише за іменем артиста, а не за назвою альбому, то оцінка _score має бути нижчою. Для цього в запиті можна задати значення boost для кожного окремого поля, регулюючи таким чином їхній внесок у загальний бал.

Сам процес денормалізації реалізується на рівні Core-сервісу, оскільки саме він має доступ до повної інформації про всі сутності. Утім, такий підхід створює проблему конкуренції запитів, яка буде розглянута та вирішена в наступному підрозділі.

У результаті ми отримуємо повноцінний пошуковий сервіс, який дозволяє швидко та зручно знаходити потрібну інформацію, враховуючи не лише точність, а й релевантність результатів.

3.2 Проблеми та їх вирішення

У процесі реалізації настільки масштабного функціоналу виникали труднощі, які було складно передбачити заздалегідь або які були випадково упущені через людський фактор. Деякі з них стали очевидними лише під час інтеграції окремих компонентів у єдину систему. У цьому підрозділі розглянуто ключові проблеми, з якими довелося зіткнутися під час розробки, проаналізовано можливі варіанти їх вирішення та обґрунтовано вибір остаточного підходу, що був імплементований у проєкті.

3.2.1 Обмеженість AMQP транспорту Symfony Messenger

Перші труднощі виникли ще на етапі інтеграції командної шини, зокрема — під час використання рішення від Symfony. Незважаючи на гнучкість та численні переваги компонента Messenger, він має низку концептуальних обмежень, які створюють серйозні перешкоди для реалізації критично важливих функцій системи. Однією з головних вимог до системи парсингу медіатеки є збереження цілісності обробки. Це означає, що всі етапи парсингу мають відбуватися в суворій послідовності, без втрат та без паралельної обробки взаємозалежних повідомлень. Будь-яке порушення цього порядку може призвести до невалідного стану системи (наприклад, якщо менш актуальна версія альбому перезапише оновлену інформацію). За замовчуванням RabbitMQ при неуспішній обробці повідомлення (тобто у випадку, коли воркер викидає виняток) відкидає його та переходить до наступного. Навіть якщо таке повідомлення потім потрапить до мертвої черги й буде повторно оброблене, порядок операцій вже буде порушено.

Для вирішення подібних ситуацій RabbitMQ підтримує механізм nack із прапорцем requeue, що дозволяє повернути повідомлення до тієї ж черги для повторної обробки з затримкою. Проте реалізація транспорту amqp-messenger у Symfony не дозволяє явно задавати цей параметр. У коді бібліотеки він жорстко захардкожений, і змінити його стандартними засобами неможливо. Це фактично унеможливлює використання Symfony Messenger у критичних системах, де важливий порядок обробки. Друга проблема полягає у неможливості динамічно вказувати exchange для окремих повідомлень.

Symfony Messenger передбачає використання одного загального exchange, який самостійно маршрутизує повідомлення до відповідних черг. Проте такий підхід створює зайву залежність від одного конкретного обмінника та серйозно обмежує можливість масштабування — ми не можемо логічно сегментувати наші черги за призначенням чи типом навантаження. Хоч друга проблема є неприємною, її ще можна обійти. Перша ж — критична,

тому було прийнято рішення реалізувати власну модифікацію транспорту amqp під потреби проєкту. Завдяки високій гнучкості компонентів Symfony це стало можливим. Кожен важливий клас у Messenger має власний інтерфейс, що визначає контракт, тож ми змогли частково перевизначити поведінку пакета amqp-messenger. Першим кроком стало створення окремого Stamp, який дозволяє вказати, до якого саме exchange має бути надіслане повідомлення.

Як відомо, у Symfony Messenger повідомлення можна обгортати у Envelope, що містить Stamp — спеціальні об'єкти з додатковими даними. У нашому випадку ми перевіряємо наявність такого Stamp перед відправленням і динамічно встановлюємо необхідний exchange. Другим кроком було додавання можливості встановлювати прапорець requeue для nack. Достатньо вказати в заголовках повідомлення requeue: true, після чого система коректно обробляє повторну відправку при помилці. Модифіковані класи вирішили обидві критичні проблеми — як з дотриманням порядку обробки, так і з можливістю гнучко керувати маршрутизацією.

Варто також згадати про реалізацію механізму повторної обробки повідомлень у самому Symfony Messenger. На жаль, спроба реалізувати такий функціонал на рівні самого застосунку створює високий ризик втрати повідомлень. Наприклад, якщо воркер починає обробку, кидає виняток і намагається переслати повідомлення до спеціально створеної тимчасової черги — і в цей момент сам процес падає — повідомлення буде втрачено.

Крім того, Symfony додає до "мертвих" повідомлень велику кількість метаданих, таких як трасування винятку, що значно збільшує розмір повідомлення. Якщо таких повторних спроб, скажімо, 10 — одне повідомлення може вирости у десятки кілобайт. Зважаючи на це, у проекті було прийнято рішення використовувати стандартний механізм RabbitMQ з мертвими чергами, який забезпечує гарантію збереження повідомлення та не створює надлишкових накладних витрат на рівні структури.

3.2.2 Обмеження сторонніх API

Як вже зазначалося у розділі 1, для реалізації механізму парсингу було використано сторонні API — зокрема Spotify API та YouTube Data API. Обидва сервіси мають обмеження на кількість запитів, що можливо виконати за певний проміжок часу, тобто працюють на основі квот. Це створює суттєві труднощі при роботі з великими обсягами даних.

Політика обмежень у Spotify є досить лояльною, однак сервіс все ж не дозволяє виконувати необмежену кількість запитів. Це особливо критично при масштабному імпорті великої кількості артистів. Наприклад, при імпорті 10 000 артистів, кожен з яких має у середньому по 5 альбомів і по 10 треків у кожному, загальна кількість треків становитиме близько 500 000. Такий обсяг парсингу неможливо реалізувати в рамках обмежень API. Spotify дозволяє отримувати максимум 50 записів за один запит. Тобто для одного артиста необхідно виконати:

- 1 запит — отримання основної інформації про артиста
- 1 запит — отримання списку альбомів; приблизно 5 запитів — по одному на кожен альбом
- 5 запитів — отримання треків для кожного альбому

Таким чином, в середньому потрібно близько 12 запитів на одного артиста. У випадку з 10 000 артистів це вже 120 000 запитів — кількість, яка суттєво перевищує добову квоту.

Після аналізу процесу оновлення даних було встановлено, що альбоми рідко змінюються після виходу — у більшості випадків не додаються нові треки, не змінюються обкладинки або назви. Враховуючи це, було прийнято рішення реалізувати механізм передачі додаткових даних про вже наявні альбоми та треки на етапі імпорту артиста.

Таким чином, коли сервіс Core надсилає повідомлення про успішне додавання нового артиста, до нього додається список вже існуючих альбомів і треків. Це дозволяє парсеру обробляти лише новий контент — тобто:

- 1 запит — отримання артиста
- 1 запит — отримання списку альбомів
- +2 запити на кожен новий альбом

Завдяки цьому підходу вдалося значно зменшити кількість необхідних запитів та оптимізувати навантаження на API.

Що стосується YouTube API, то тут ситуація ще складніша — кількість запитів жорстко обмежена системою квот. Найефективнішим способом економії є попередня перевірка наявності треку в локальному сховищі перед виконанням запиту до YouTube. Якщо пісня вже існує, немає необхідності дублювати її парсинг. Це особливо важливо при імпорті плейлистів, де часто зустрічаються одні й ті самі композиції.

Найбільша проблема виникає під час першої ініціалізації системи, коли база даних ще не містить жодного треку. Якщо спробувати одразу спарсити весь каталог, це створить надмірне навантаження на API Spotify, що потенційно може призвести до блокування або тимчасового відключення ключа доступу.

Щоб уникнути цього, було вирішено розбити першу ітерацію парсингу на кілька десятків менших підзадач з поступовою обробкою артистів. Такий підхід дозволив рівномірно розподілити навантаження в часі й уникнути перевищення квот, хоч і потребував більше часу для повного парсингу всієї медіатеки.

3.2.3 Проблема конкурентності запитів під час реплікації

Як було описано раніше, для побудови проєкцій сутностей у сервісі Search було реалізовано денормалізацію даних. Ідея цієї реалізації була натхненна підходом асинхронної реплікації в MySQL, однак замість зчитування даних з binary log, використовується асинхронна передача повідомлень через чергу.

Після того як у сервісі Core зберігаються або оновлюються документи, бібліотека Doctrine ODM генерує відповідні події postPersist() (для створення)

та postUpdate() (для оновлення). У тілі цих подій передається сам об’єкт сутності, який було збережено. Для обробки цих подій був реалізований спеціальний event listener, який отримує сутність, збирає усі необхідні додаткові дані та надсилає повідомлення до черги.

З метою підвищення продуктивності та масштабованості було прийнято рішення створювати окрему чергу для кожного типу сутності (наприклад, артисти, альбоми, треки тощо) та реплікувати їх незалежно.

Однак цей підхід також створює критичну проблему конкурентності обробки, яка без спеціальних заходів може призвести до неконсистентного стану даних у реплікованому сховищі.

Прикладом може слугувати ситуація, коли в систему додається артистка Надя Дорофєєва та її альбом Heartbeat. У момент створення альбому інформація про артистку ще не була збережена в базі даних, тому повідомлення про альбом формується без повного набору даних. Після цього зберігається сама артистка, і вже її повідомлення надходить у чергу. Якщо обробка артистів відбувається швидше, то її повідомлення обробляється раніше, оновлює відповідні альбоми, але не включає ще не збережений Heartbeat. Коли ж нарешті оброблюється альбом, він потрапляє у сховище без інформації про артиста, що призводить до невалідного або неповного стану документа.

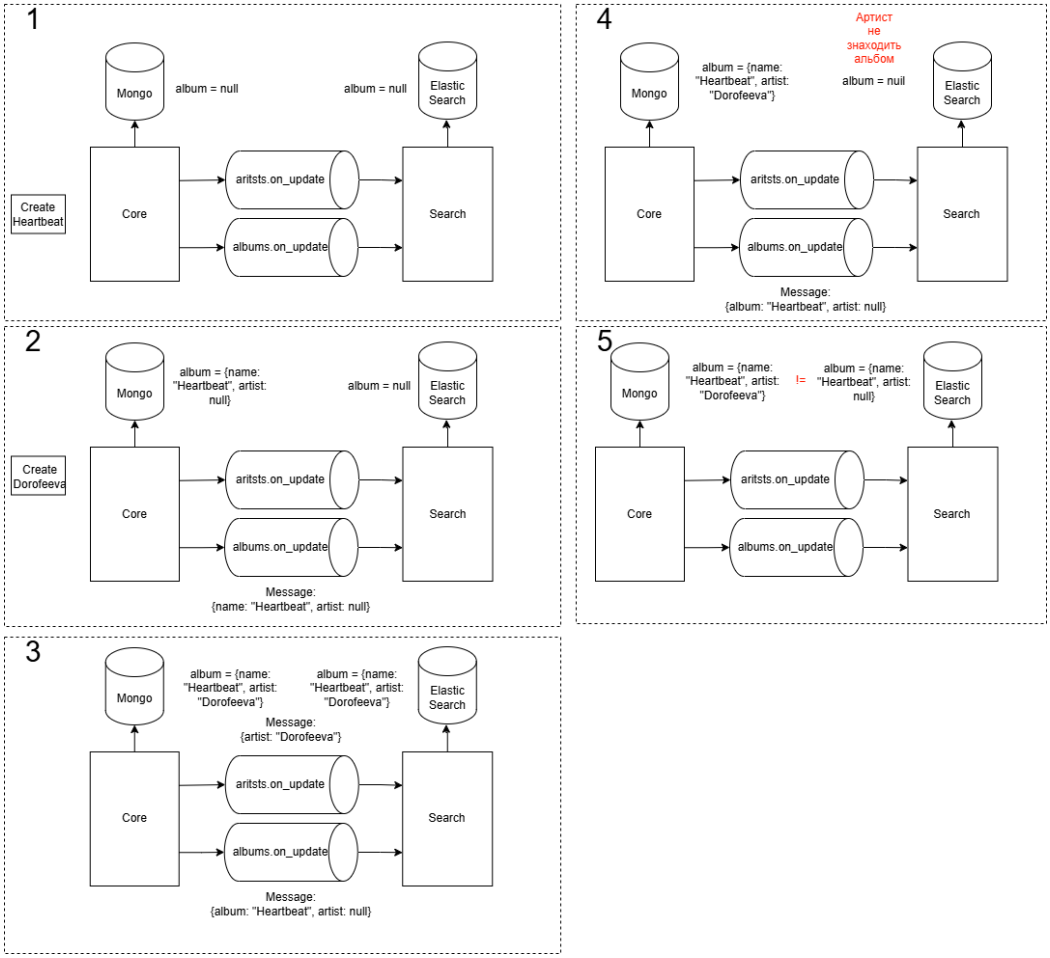


Рисунок 3.11 – Приклад конкурентної реплікації

Існує два можливих підходи до вирішення проблеми конкурентної реплікації.

Перший — це спростити архітектуру та використовувати одну загальну чергу для всіх типів сутностей. Такий підхід усуває проблему порядку обробки повідомлень, оскільки всі вони опрацьовуються послідовно. Проте це значно знижує масштабованість системи. Втрачається можливість додавати незалежні черги для кожної сутності, що не дозволяє рівномірно розподіляти навантаження та масштабувати систему в міру зростання обсягів даних і трафіку. У результаті система перетворюється на вузьке горлечко, особливо при високому потоці інформації.

Другий, більш гнучкий підхід — це збирання сутності по частинах. Коли повідомлення з артистом обробляється раніше, ніж повідомлення з альбомом, воно створює частковий запис в індексі альбомів. Цей запис містить ID

альбому, коротку інформацію про артиста та спеціальний прапорець, який позначає, що запис є неповним і невалідним — він не буде включений до результатів пошуку чи відображення користувачеві. Коли пізніше приходить повідомлення з повними даними альбому, воно знаходить цю заготовку, доповнює її відсутньою інформацією та переводить прапорець у валідний стан. Такий підхід вимагає, щоб під час формування повідомлення про артиста на Core до нього прикріплювався список ID усіх пов’язаних з ним альбомів. Це дозволяє заздалегідь підготувати шаблони для кожної пов’язаної сутності. Навіть якщо в майбутньому у артиста з’являться нові альбоми, кожен з них зможе підтягнути актуальну інформацію про артиста та створити повноцінний запис без порушення цілісності даних.

Таким чином, ми вирішили проблему конкурентності обробки повідомлень і водночас зберегли можливість горизонтального масштабування системи. Такий підхід дозволяє ефективно розподіляти навантаження між чергами, не втрачаючи цілісності даних у процесі реплікації.

ВИСНОВОК ДО РОЗДІЛУ

Цей розділ описав процес розробки нашої системи, зосереджуючи увагу на найбільш цікавих і технічно складних аспектах. Спершу було розгорнуто командну шину за допомогою Symfony Messenger, після чого інтегровано сторонні API, розглянуто їхні обмеження, проблеми, які вони створюють, та способи адаптації до цих умов. Далі було реалізовано основний сервіс Core: описано ключові деталі реалізації CRUD-операцій, роботу з Doctrine ODM та створення REST-ендпоінтів.

Наступним кроком стала реплікація даних для пошукового сервісу, його налаштування та особливості роботи з аналізатором. Було вирішено проблему конкурентних запитів при реплікації, що дозволило зберегти цілісність і масштабованість системи. В завершення була реалізована сама гра на WebSocket і механізм підбору матчів Matchmaker.

На поточному етапі система готова до експлуатації та повністю покриває всі цілі, що були поставлені на початку роботи. Проте попереду ще один важливий етап — тестування.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ПОДАЛЬШІ ДІЇ

Після завершення реалізації системи та її готовності до експлуатації необхідно здійснити завершальні кроки, які підтвердять правильність функціонування кожного з компонентів та забезпечать стабільне розгортання на продакшн-середовище. У цьому розділі буде описано процес тестування всієї системи, а також етапи підготовки до розгортання.

Насамперед буде проведене тестування, яке дозволяє перевірити загальну працездатність основних функціональних модулів. Це забезпечить впевненість у тому, що система запускається, приймає запити, взаємодіє з базою даних та чергами повідомлень, а також коректно відповідає на API-запити.

Далі буде написано автоматизовані тести для основних компонентів системи.

Також буде згенеровано документацію до REST API за допомогою Swagger. Це значно полегшить роботу майбутніх розробників, особливо під час реалізації клієнтської частини. Документація дозволяє швидко зрозуміти структуру запитів, можливі відповіді, формат полів, а також передбачені HTTP-методи. Також буде розглянуто процес автоматичного розгортання.

Завершальною частиною розділу стане оцінка масштабованості системи. Буде проаналізовано можливість горизонтального масштабування компонентів та перспективи розподіленого зберігання. Також буде описано потенційні напрями розвитку системи.

4.1 Тестування

Рано чи пізно будь-який розробник, який супроводжує систему, стикається з необхідністю змінювати вже існуючий код. Такі зміни завжди несуть ризик порушення працездатності системи, особливо якщо вона з самого початку не була покрита тестами. Навіть дотримання принципу

відкритості/закритості не гарантує повного захисту від ненавмисних помилок, особливо в складних архітектурах з великою кількістю залежностей. У таких випадках ключовим інструментом перевірки стабільності є тестування.

Тестування — це комплексна і багатогранна тема, яка включає безліч типів: юніт-тести, інтеграційні, end-to-end, навантажувальні, візуальні та інші. Проте наша система хоч і потребує тестування, однак не вимагає покриття всіма можливими типами тестів — це було б надмірно ресурсозатратно та малоефективно в контексті поточних задач.

Тому було прийнято рішення зосередитись на двох підходах: димових тестах, які дозволяють швидко перевірити базову працездатність системи, та автоматизованих тестах, що покривають критично важливу бізнес-логіку. Такий підхід забезпечує баланс між якістю, швидкістю розробки та ефективністю підтримки системи.

4.1.1 Димове тестування

Димове тестування проводиться майже щоразу, коли розробник впроваджує новий функціонал або змінює вже існуючий код, який не покритий автоматизованими тестами. Це є невід'ємною частиною процесу розробки та перевірки працездатності системи на базовому рівні.

У контексті цієї роботи димове тестування було проведено з метою демонстрації коректної роботи системи з наочним результатом, який можна представити у пояснювальній записці. На відміну від автоматичних тестів, які зазвичай лише сигналізують про успішність або провал перевірки, смоук-тестування дозволяє візуально переконатися в тому, що основні функції системи працюють належним чином. Також автоматичні тести, як правило, не зберігають жодних змін у системі — всі дані, які були створені або змінені під час виконання тесту, видаляються одразу після його завершення, щоб уникнути побічних ефектів, які можуть вплинути на інші тести або саму систему. Тому передивитися зміни у стані системи не вийде[].

Насамперед було протестовано найскладнішу функцію системи з тривалим процесом виконання — парсинг медіа. Для демонстрації було обрано групу Guns N' Roses. Її парсинг дозволяє перевірити не лише роботу API, але й усю логіку обробки, черги повідомлень та взаємодію з іншими сервісами.

Щоб запустити процес, ідентифікатор виконавця (Spotify ID), який можна отримати на офіційному сайті або в застосунку Spotify, було додано до файлу з регулярними оновленнями.

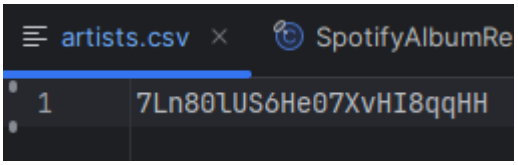


Рисунок 4.1 – Приклад файлу з артистами для парсингу

Після цього запустимо обробку файлу власноруч. Цей тест дозволяє переконатися, що вся ланка взаємодії — від отримання даних з API до збереження у сховище та відправлення у чергу — працює стабільно та без збоїв.

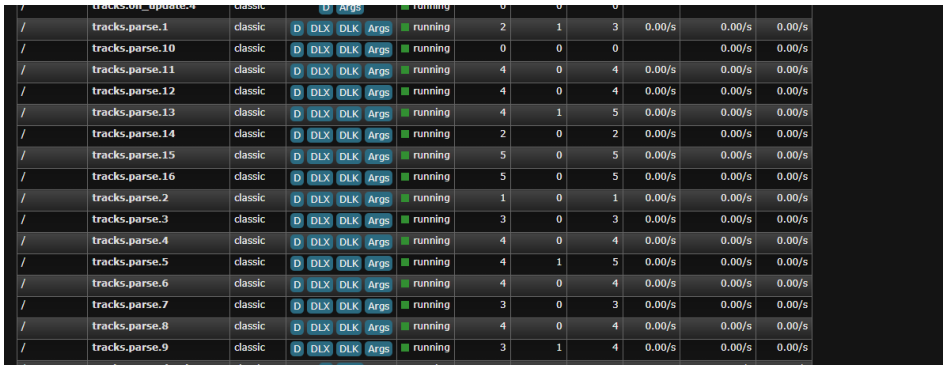
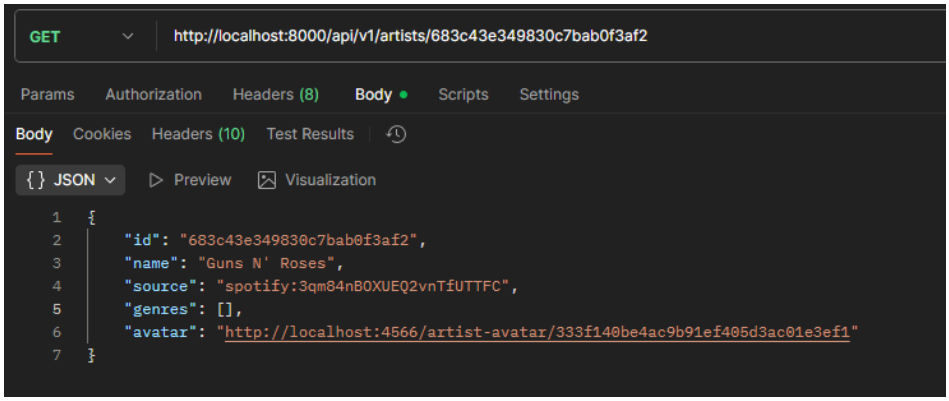


Рисунок 4.2 – Заповнені черги під час парсингу

Після того як перші пісні були додані до бази даних, можемо перевірити API



Рисунокы 4.3 – Відображення артиста у API

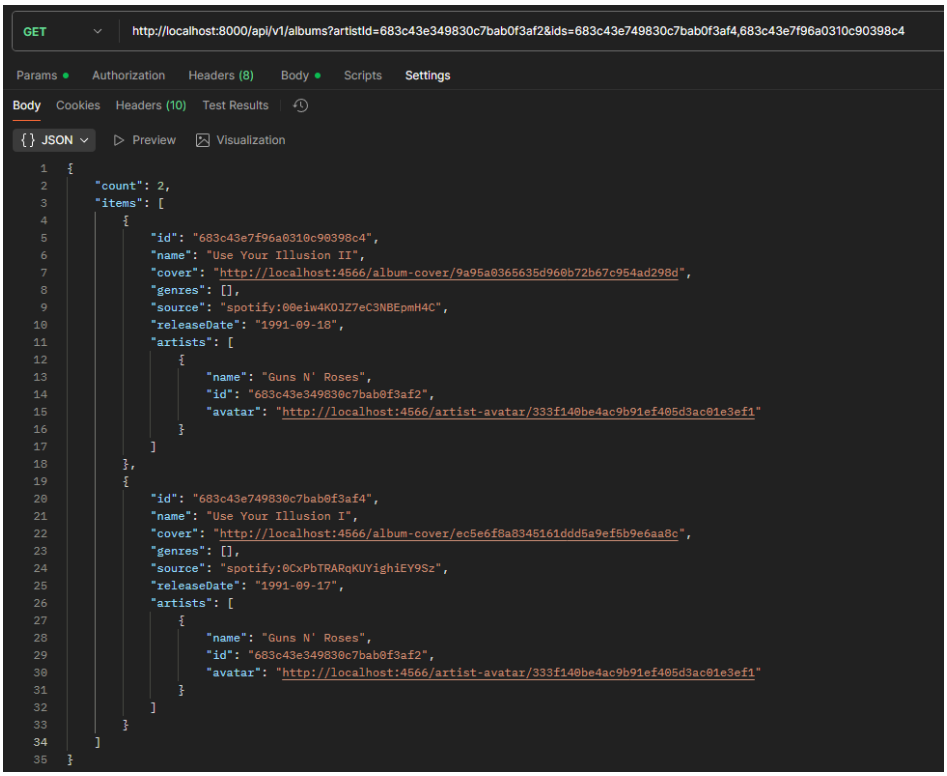


Рисунок 4.4 – Відображення альбомів у API

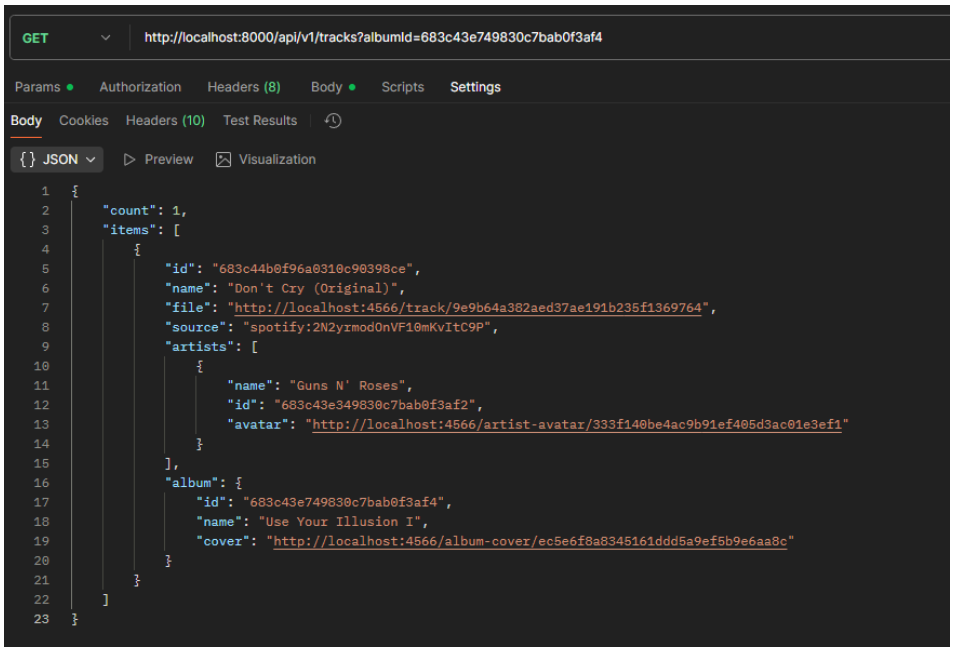


Рисунок 4.5 – Відображення треків у API

Тепер перевіримо правильність роботи реплікації на сервіс Search.

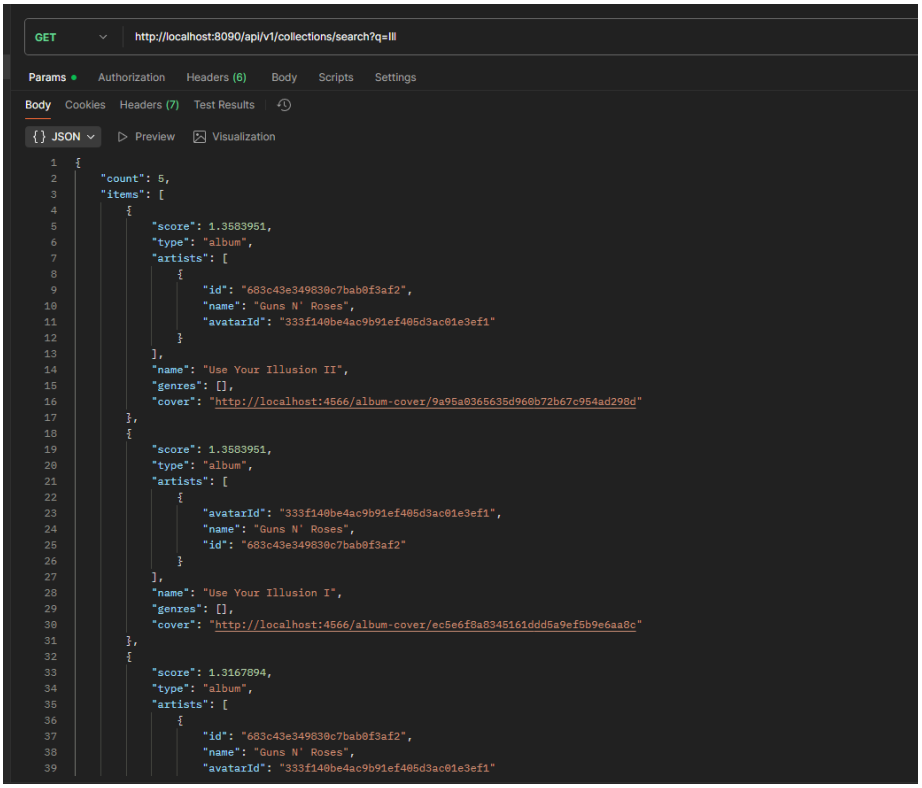


Рисунок 4.6 – Робота пошукового сервісу

Таким чином ми прослідкували за повним процесом парсингу артиста, хоч і для повного заповнення потрібний час, ми все одне переконалися в працездатності нашої системи.

4.1.2 Автоматизоване тестування

Димове тестування не потребує написання коду, проте має один критичний недолік — відсутність автоматизації. Навіть невеликий застосунок містить багато важливих частин логіки, які повинні залишатися працездатними. Постійно тримати їх у пам’яті та вручну перевіряти під час кожної зміни — практично неможливо. Саме для цього існують автоматизовані тести.

Автоматичні тести поділяються на багато категорій за різними ознаками, але найпоширенішими є:

- Модульні тести — перевіряють окрему одиницю логіки, зазвичай одну функцію чи метод. Вони є дуже швидкими та ізольованими, але ефективні лише тоді, коли система має складну внутрішню бізнес-логіку
- Інтеграційні тести — перевіряють взаємодію між кількома модулями або сервісами. Наприклад, перевірка запису в базу даних або відправки повідомлення в брокер повідомлень
- End-to-End тести — перевіряють повний сценарій використання системи, симулюючи реальну поведінку користувача (наприклад, HTTP-запити до API)

З формальної точки зору, логічно було б почати з написання модульних тестів. Проте, після аналізу архітектури проєкту, було вирішено, що юніт-тестування є надмірним і неефективним[]. Це пов'язано з тим, що більшість логіки — це інтеграція з зовнішніми сервісами: API, чергами повідомлень, базами даних тощо.

Багато внутрішніх сервісів системи виступають лише посередниками між цими ресурсами, не містячи власної бізнес-логіки, а отже не мають сенсу

для юніт-тестування. У зв’язку з цим було прийнято рішення зосередитися лише на інтеграційних та E2E тестах.

Багато внутрішніх сервісів системи виступають лише посередниками між цими ресурсами, не містячи власної бізнес-логіки, а отже не мають сенсу для юніт-тестування. У зв’язку з цим було прийнято рішення зосередитися лише на інтеграційних та E2E тестах.

Таким чином, вдалося охопити тестами значну частину застосунку: сервіси Song Parser, Core та Search. Тестування сервісів Matchmaker та Game було вирішено поки що пропустити через високу складність сценаріїв, але вони заплановані до покриття в рамках майбутньої підтримки системи.

4.2 Документація

Наш основний сервіс Core містить REST API для взаємодії з артистами, альбомами, треками та плейлистами. Без знання внутрішньої реалізації майже неможливо зрозуміти, який ендпоінт відповідає за яку функцію, за яким URI він доступний і яку поведінку має. Тому гарною практикою є створення окремої документації до REST API.

Проблема полягає в тому, що розробка власної документації "з нуля" потребує значних зусиль: потрібно створити інтерфейс користувача, продумати логіку взаємодії, забезпечити актуальність описів, структурувати відповіді й помилки, а також передбачити зручний механізм оновлення. Таку документацію розробляють, зокрема, великі компанії на кшталт Spotify чи YouTube — їхні API мають офіційно підтримувану документацію з веб-інтерфейсом, прикладами та можливістю тестування запитів. Однак для нас це надто ресурсозатратно, особливо з урахуванням того, що документація орієнтована передусім на розробників, і створення власного UI наразі лише планується в наступних етапах.

В умовах обмежених ресурсів найкращим рішенням стає використання стандарту OpenAPI і Swagger UI — популярного інструменту для автоматичної генерації та інтерактивного перегляду REST API-документації.

OpenAPI дозволяє описати всі ендпоінти API у форматі YAML або JSON, зазначити методи запитів, тіла, параметри, коди відповідей, а також структуру об’єктів. Swagger UI на його основі створює зручний веб-інтерфейс, який дозволяє не лише переглядати API, а й тестувати його безпосередньо з браузера.

У нашій реалізації для зручності та зменшення ручної роботи ми використали анотації. Анотації, які вже активно застосовуються в проекті для позначення обробників повідомлень у Symfony Messenger, також чудово підходять для OpenAPI. Завдяки ним можна позначити будь-який метод контролера, що обробляє HTTP-запит, відповідною метаінформацією. Надалі — за допомогою механізму рефлексії — ми автоматично зчитуємо всі такі анотації, формуємо конфігураційний файл у форматі OpenAPI, який є цілком сумісним зі Swagger UI.

Крім того, щоб уникнути необхідності вручну перевиготовляти конфігурацію після кожної зміни, можна реалізувати динамічний ендпоінт, який під час запиту автоматично зчитує структуру проєкту, аналізує анотації і генерує свіжу версію OpenAPI-опису в реальному часі. Це рішення підвищує гнучкість, зменшує технічний борг і дозволяє підтримувати документацію в актуальному стані без зайвих зусиль.

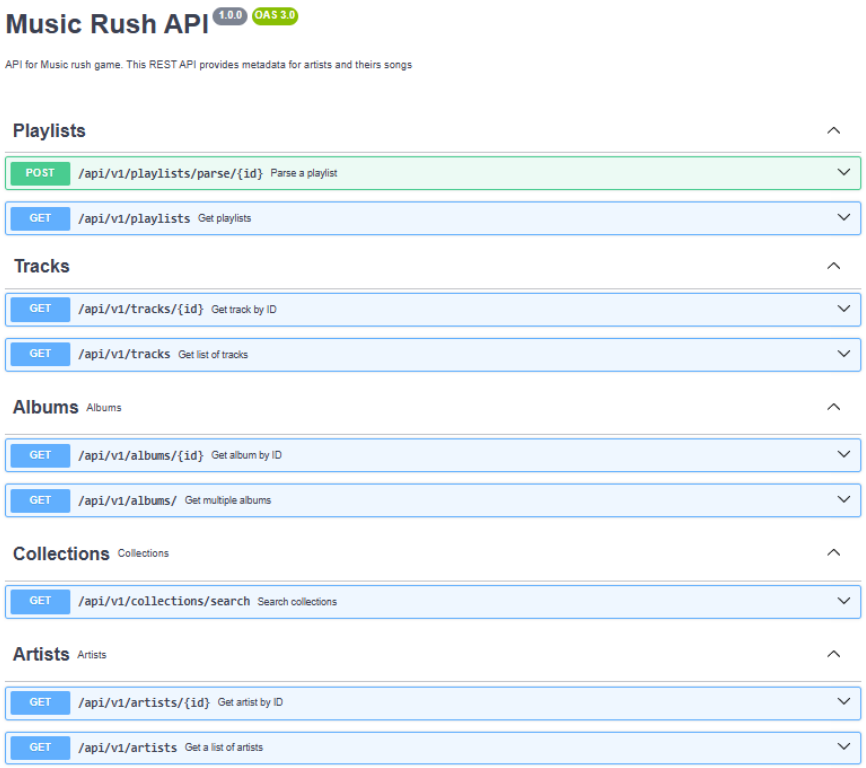


Рисунок 4.7 – Сторінка Swagger з API проекту

4.3 Масштабованість

Однією з ключових характеристик якісної системи є здатність масштабуватися безболісно при зміні навантаження. Оскільки можливість масштабування була врахована ще на етапі проектування, поточна архітектура системи дозволяє ефективно масштабувати її по горизонталі. Мікросервісна структура та сегментована база даних забезпечують незалежність окремих компонентів, що дає змогу розглядати масштабування на рівні кожного сервісу окремо.

4.3.1 Song Parser

Це один з найнавантаженіших сервісів системи. Основним фактором для масштабування є зростання запитів на парсинг плейлистів. Головна складність — зовнішні API, з якими сервіс тісно інтегрований. Щоб збільшити пропускну здатність, перш за все, необхідно домовитися з провайдерами API щодо підвищення квот і лімітів доступу. Сторонні API, які ми використовуємо, вже масштабовані на високий рівень, тому основна складність — юридична, а не

технічна. З боку внутрішньої обробки RabbitMQ масштабуються легко — достатньо створити додаткові черги та підключити до них нових воркерів. Також можливе горизонтальне масштабування RabbitMQ шляхом розподілу черг між вузлами, що суттєво підвищить пропускну здатність.

4.3.2 Core

Однією з причин вибору MongoDB, як згадувалося у розділі 2, була вбудована підтримка масштабування. Mongo підтримує як реплікацію, так і шардинг, що дозволяє адаптуватися до зростання навантаження. Очікується, що навантаження буде зростати саме на читання, тому буде достатньо розгорнути декілька реплік, які оброблятимуть запити читання в режимі eventual consistency. Актуальність даних у цій частині системи не критична, тому затримки в оновленні допустимі. Для масштабування самого застосунку досить підняти додаткові PHP-контейнери, щоб рівномірно розподіляти навантаження.

4.3.3 Search

Архітектурно аналогічний до Core, але масштабувати ElasticSearch ще простіше. Необхідно створити кластер з кількох нод, після чого переіндексувати існуючі індекси із новими налаштуваннями (кількість шард і реплік). Такий підхід дозволяє гнучко керувати ресурсами залежно від інтенсивності пошукових запитів.

4.3.3 Game

Це WebSocket-сервіс, відповідальний за відображення ігрового процесу. Його реалізація передбачає, що кожен окремий Node.js-процес обслуговує лише одну сесію. Таким чином, кожна сесія є повністю ізольованою. Для масштабування достатньо виділяти додаткові вузли з ресурсами та запускати на них нові контейнери для обробки нових ігрових сесій.

4.3.4 MatchMaker

Цей сервіс оркестрації масштабувати найпростіше, оскільки його завдання зводиться до отримання активних сесій з Redis. Redis, у свою чергу,

легко масштабується через шардинг, адже є key-value сховищем. Що стосується контролю над неактивними контейнерами — достатньо виділити декілька воркерів, які періодично перевірятимуть вузли та зупинятимуть непотрібні ігрові контейнери.

4.4 Можливості майбутнього розвитку

Після завершення основного етапу розробки, підготовки системи до використання та проведення тестування, логічним наступним кроком є визначення можливих напрямків майбутнього розвитку. Незважаючи на стабільну роботу поточного функціоналу, існує низка аспектів, які є обов'язковими або опціональними для реалізації перед повним запуском системи у Інтернеті.

4.4.1 Створення UI

Незважаючи на повністю готову серверну частину, використання системи реальними користувачами наразі неможливе через відсутність графічного інтерфейсу. Без зручного та зрозумілого UI взаємодія з додатком взагалі стає недоступною для більшості користувачів. Тому першим кроком подальшого розвитку системи має стати реалізація фронтенд-частини.

На момент написання роботи було прийнято рішення використовувати React.js як основну бібліотеку для побудови інтерфейсу. Для керування станом додатку буде застосовано Redux у поєднанні з Redux Saga. Така архітектура дозволяє відокремити бізнес-логіку від інтерфейсу, забезпечити масштабованість і легкість супроводу, а також спростити обробку асинхронних процесів.

4.4.2 Отримання повноважень для сторонніх API

Одним із критичних компонентів системи є інтеграція зі сторонніми API, які використовуються для парсингу даних про артистів, плейлистів, альбомів та треків. На поточному етапі розробки використано тестові або обмежені облікові записи, які мають знижені ліміти на кількість запитів та доступ до функцій. Для переходу в продуктивне середовище та

обслуговування ширшої аудиторії користувачів, необхідно отримати офіційні повноваження. Окрім технічної вигоди у вигляді збільшених квот, стабільної роботи та офіційної підтримки API, отримання таких повноважень також дозволить легально експлуатувати нашу систему. Це означає, що розроблене рішення зможе бути офіційно опубліковане, масштабоване, просуване як продукт, а також не порушуватиме юридичні норми використання сторонніх ресурсів.

ВИСНОВОК ДО РОЗДІЛУ

Останній розділ цієї роботи присвячено фінальному етапу розробки — тестуванню. Було проведено перевірку працездатності всієї системи за допомогою двох основних підходів: димового тестування та автоматизованого тестування.

На початку розділу продемонстровано приклад парсингу артиста та його композицій, що дозволило наочно показати, що система функціонує правильно й готова до експлуатації. Далі увагу було зосереджено на автоматизованому тестуванні. Було реалізовано набір інтеграційних тестів для перевірки репозиторіїв, командної шини та надсилання повідомлень у RabbitMQ. Окрім того, було додано end-to-end тести для перевірки коректності роботи REST API.

Після успішного проходження всіх тестів можна впевнено стверджувати, що додаток працює коректно та готовий до подальшої підтримки й розвитку. Обов’язковим кроком також стало генерування документації API за стандартом OpenAPI, що було реалізовано за допомогою відповідного PHP-пакета. Цей підхід у поєднанні з типізацією та використанням анотацій значно спростив процес документування та покращив читабельність коду.

Окрему увагу приділено оцінці масштабованості системи. Поточна мікросервісна архітектура дозволяє масштабувати кожен окремий сервіс незалежно, що забезпечує гнучкість і готовність до зростання навантаження.

Насамкінець, було окреслено напрямки подальшого розвитку, серед яких створення повноцінного користувацького інтерфейсу, отримання офіційних повноважень для роботи зі сторонніми API та вихід проєкту на реальний ринок.

ВИСНОВКИ

Ця робота була присвячена розробці багатокористувацької гри з взаємодією в реальному часі, суть якої полягає у вгадуванні музичних треків. Увесь процес розробки було поділено на низку послідовних етапів.

На першому етапі було зібрано вимоги до системи та проаналізовано існуючі рішення. Зокрема, було розглянуто відкритий проект Woogles.io, визначено його переваги та недоліки. Успішні аспекти стали фундаментом для нашої розробки, а виявлені недоліки було враховано та усунуто на етапі проектування. Далі було проведено аналіз сторонніх API для формування медіатеки. Обрано Spotify API та YouTube Data API як основні джерела даних, з можливістю подальшого розширення.

Другий етап було присвячено архітектурному проектуванню. Було обрано відповідні мови програмування, фреймворки, архітектурний стиль, типи баз даних, черги повідомлень і формат побудови REST API. Уся система була поділена на окремі мікросервіси, що зробило її більш гнучкою, масштабованою та відмовостійкою. Окремо описано кожен сервіс — від парсера медіа до вебсокет-сервісу гри.

Наступним етапом стала реалізація функціоналу. У роботі зосереджено увагу на найважливіших моментах. Розробка розпочалась з парсера: було реалізовано командну шину, інтегровано сторонні API, налагоджено кожен етап обробки плейлистів. Описано реалізацію механізму асинхронного сповіщення користувача про статус довготривалої операції. Для цього було інтегровано протокол STOMP, що дозволяє зручно працювати з чергами через WebSocket.

Далі було розроблено центральний сервіс Core, визначено найефективніші підходи до читання і запису в базу даних, створено REST API для взаємодії з іншими сервісами. Одразу після цього реалізовано пошуковий сервіс із підтримкою автодоповнення. Детально описано принципи токенізації та налаштування аналізаторів у ElasticSearch. Також у розділі реалізації

приділено увагу проблемам, які виникали під час розробки, з детальним описом шляхів їх вирішення.

Наступним кроком було створення самої гри на базі WebSocket-сервісу, де описано основні компоненти, загальну архітектуру гри та механізми синхронізації стану з клієнтом. Завершальним мікросервісом став Matchmaker — оркестратор підключень до сесій. Розглянуто його механізм обробки одночасних запитів, актуалізації сесій та підняття додаткових контейнерів для нових ігрових кімнат.

Фінальним етапом стало тестування системи. Було проведено димне тестування, яке продемонструвало працездатність усіх частин системи, а також реалізовано автоматизовані тести — інтеграційні та E2E, що покривають більшість функціональних частин. Крім того, створено документацію до REST API за допомогою OpenAPI.

Окрему увагу було приділено масштабованості системи, яка завдяки мікросервісній архітектурі дозволяє масштабувати кожен сервіс незалежно. Також наведено плани подальшого розвитку: створення користувацького інтерфейсу та отримання офіційного доступу до сторонніх API.

Підсумовуючи, можна зробити висновок, що всі поставлені на початку роботи цілі було досягнуто. Створено повністю готову серверну частину багатокористувацького застосунку з реальним часом, з урахуванням подальшої підтримки, масштабування та виходу на ринок.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Woogles.io application’s open-source code [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/woogles-io/
2. Spotify For Developers WEB API documentation [Електронний ресурс]. – Режим доступу до ресурсу: https://developer.spotify.com/documentation/web-api
3. Apple Music Kit [Електронний ресурс]. – Режим доступу до ресурсу: https://developer.apple.com/documentation/musickit
4. YouTube Data API [Електронний ресурс]. – Режим доступу до ресурсу: https://developers.google.com/youtube/v3/docs
5. Sommerville Ian, Software Engineering. Boston. – 1988. – с. 28–76.
6. Newman Sam, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, Sebastopol. – 2019. с. 1–17.
7. Evans Eric, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, Boston. – 2003. – с. 49–52, 106–113.
8. How a PHP interpreter works [Електронний ресурс]. – Режим доступу до ресурсу: https://www.droptica.com/blog/how-php-interpreter-works/
9. Kleppmann Martin, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Sebastopol. – 2017. с. 28–42, 439–451.
10. Fielding R. Architectural Styles and the Design of Network-based Software Architectures [Електронний ресурс]. – Режим доступу до ресурсу: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
11. Ben Stopford, Designing Event-Driven Systems. O'Reilly Media, Sebastopol. – 2018. – с. 29–44.
12. What’s the Difference Between Kafka and RabbitMQ? [Електронний ресурс] – AWS Documentation. – Режим доступу до ресурсу: https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka/?nc1=h_ls

13. Yt-dlp utility source code [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/yt-dlp/yt-dlp
14. Symfony Messenger Component Documentation [Електронний ресурс]. – Режим доступу до ресурсу: https://symfony.com/doc/current/components/messenger.html
15. Pranav Shukla, Sharat Kumar, Elasticsearch, Kibana, Logstash - The Elastic Stack Essentials. Packt Publishing, Birmingham. – 2017. - с. 76–103.
16. Кравченко А. How to do smoke testing: a step-by-step guide [Електронний ресурс]. – Режим доступу до ресурсу: https://luxequality.com/blog/how-to-do-smoke-testing/

17. Vocke H. The Practical Test Pyramid [Електронний ресурс]. – Режим доступу: https://martinfowler.com/articles/practical-test-pyramid.html

Додаток 1

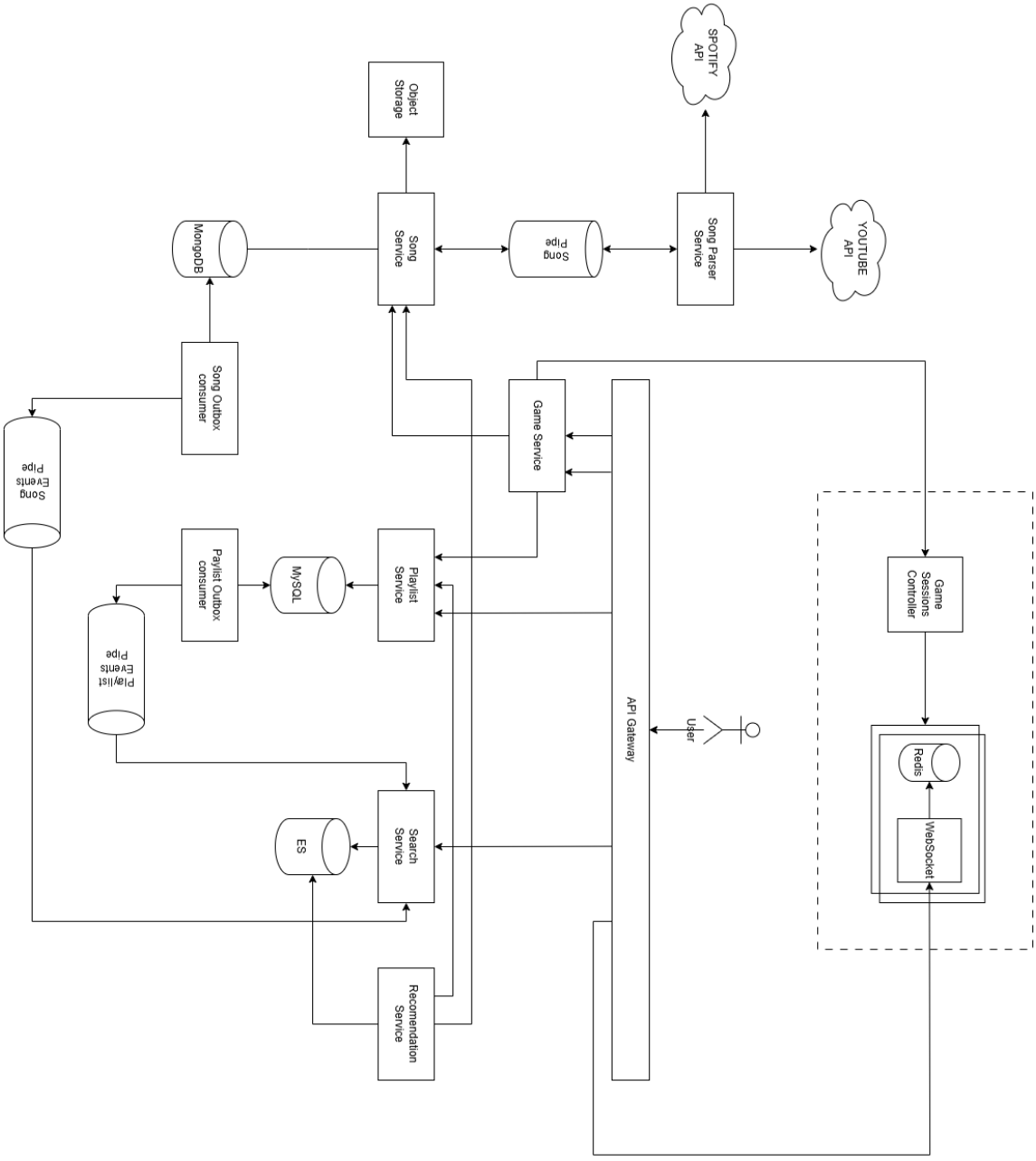
Багатокористувацька гра-вікторина з використанням мікросервісної архітектури

Схема архітектури системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ – 2025



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Полтавський В.Д.			*Багатокористувацька гра-вікторина з використанням мікросервісної архітектури* Схема архітектури системи	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Русінов В. В.					
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13	
Н. Контр.							
Затвердив							

Додаток 2

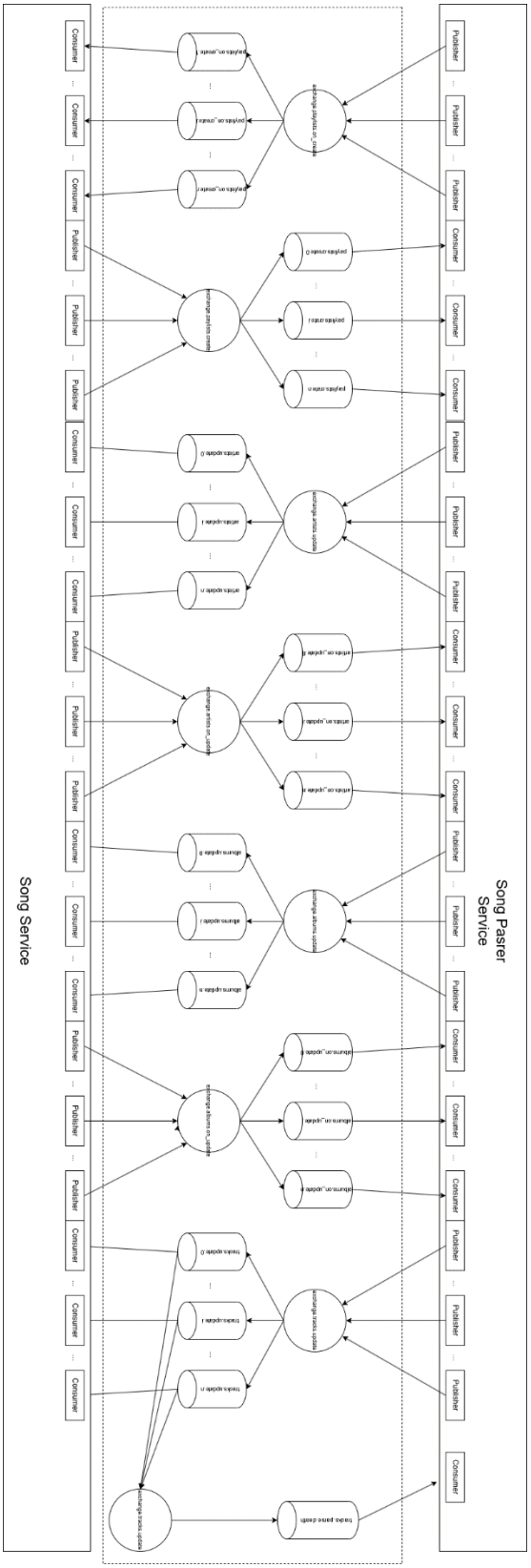
Багатокористувацька гра-вікторина з використанням мікросервісної архітектури

Схема RabbitMQ

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ - 2025 р



					ІАЛЦ.467200.005 Д2
		№ докум.	Підпис	Дата	
Розробив		Полтавський В.Д.			*Багатокористувацька гравікторина з використанням мікросервісної архітектури* Схема RabbitMQ
Перевірив		Русінов В. В.			
Н. Контр.					
Затвердив					

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13

Додаток 3

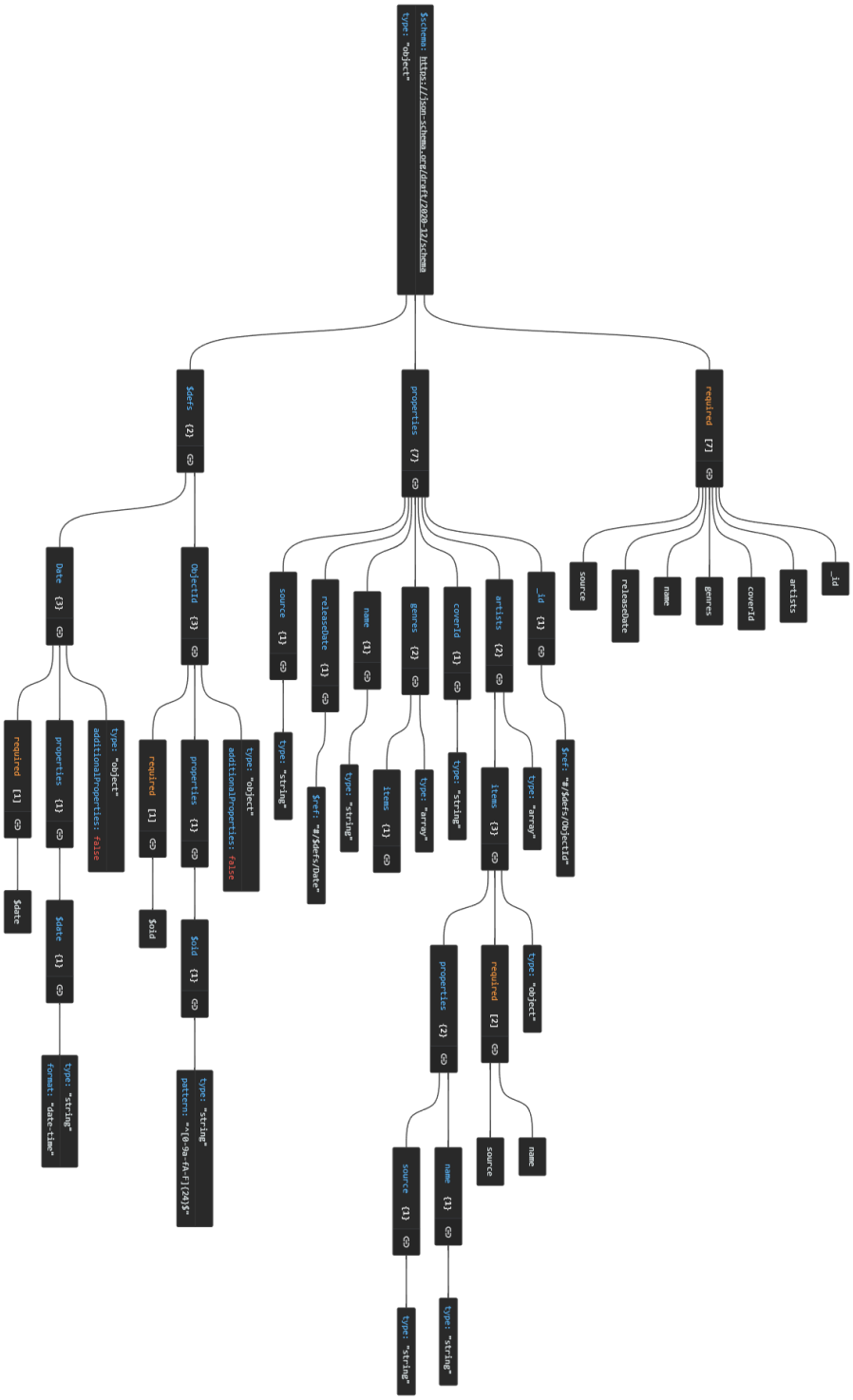
Багатокористувацька гра-вікторина з використанням мікросервісної архітектури

Схематика бази даних

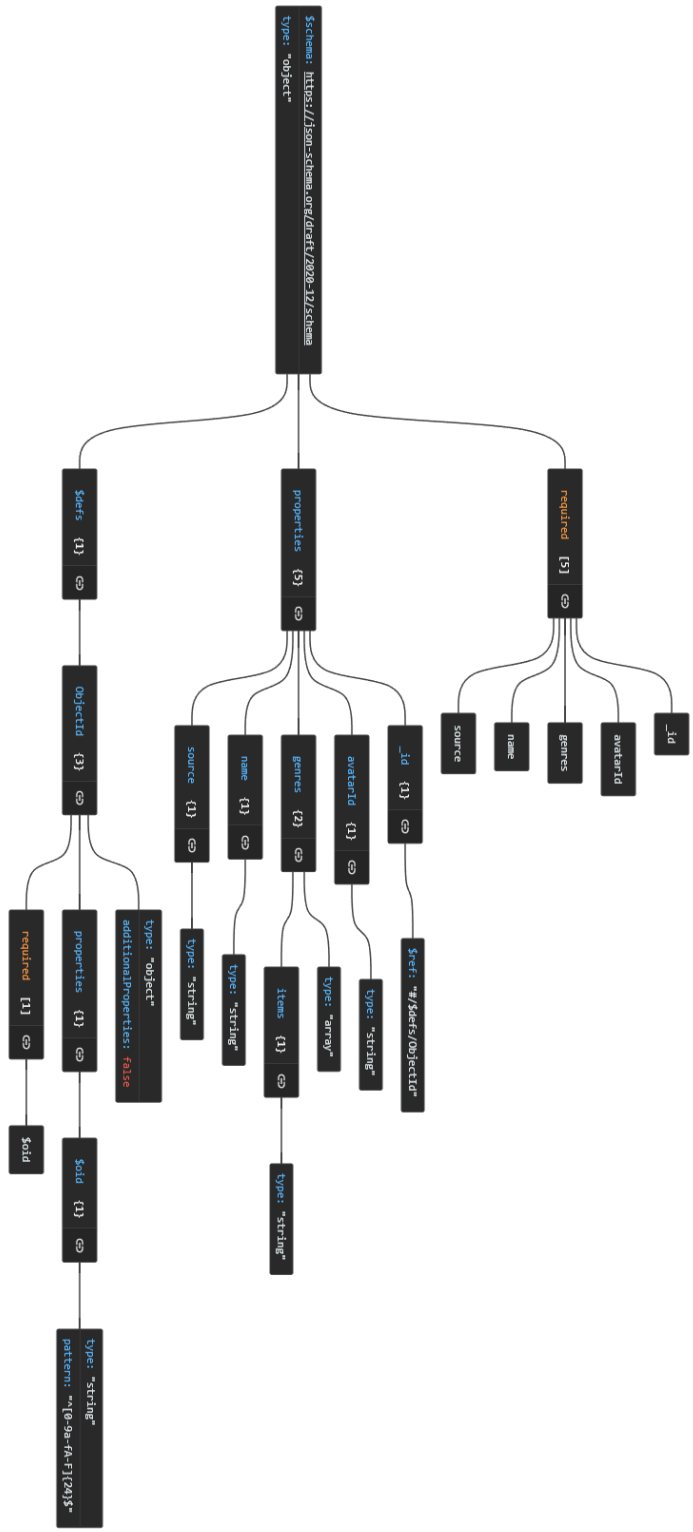
ІАЛЦ.467200.006 ДЗ

Аркушів 4

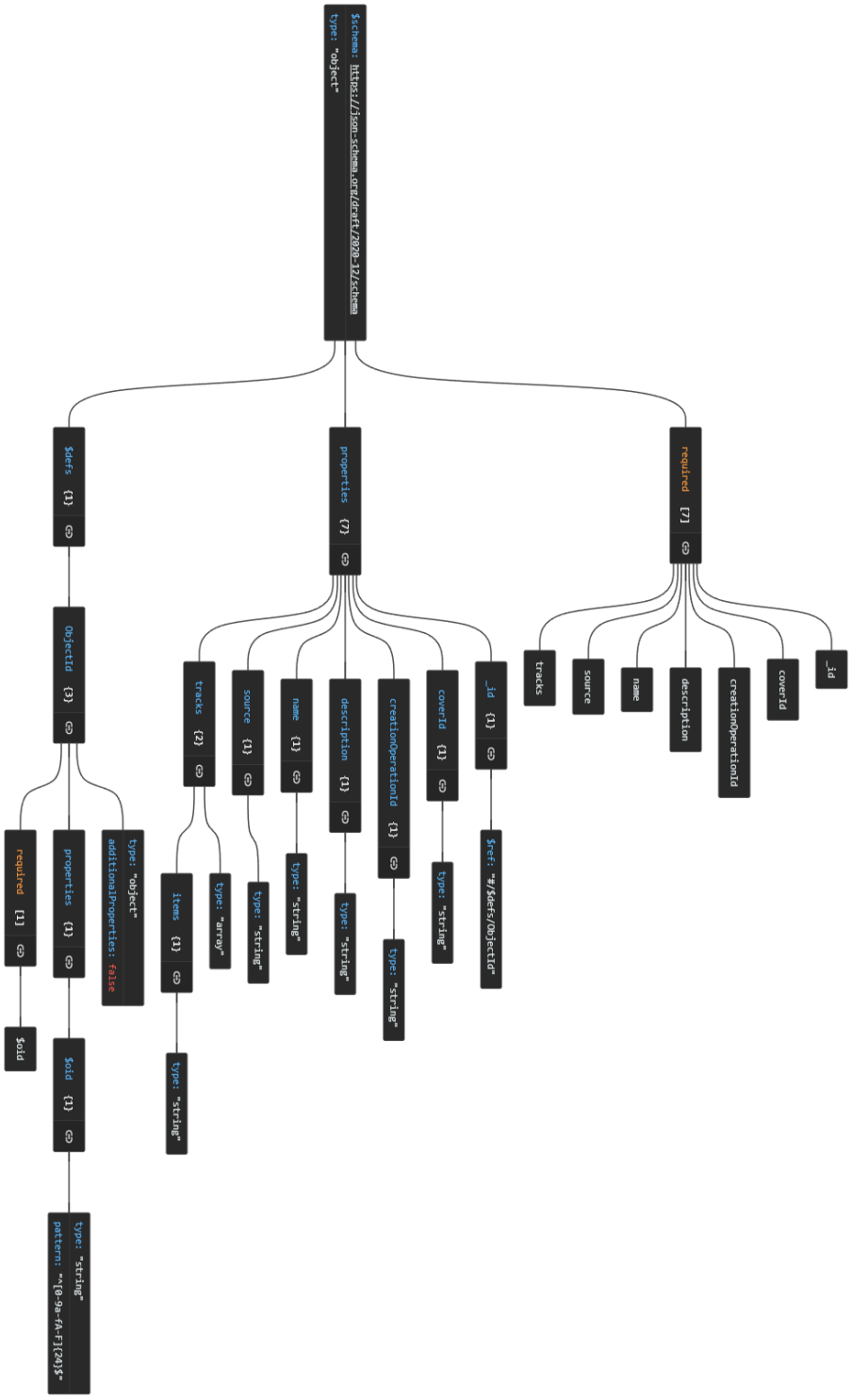
Київ - 2025



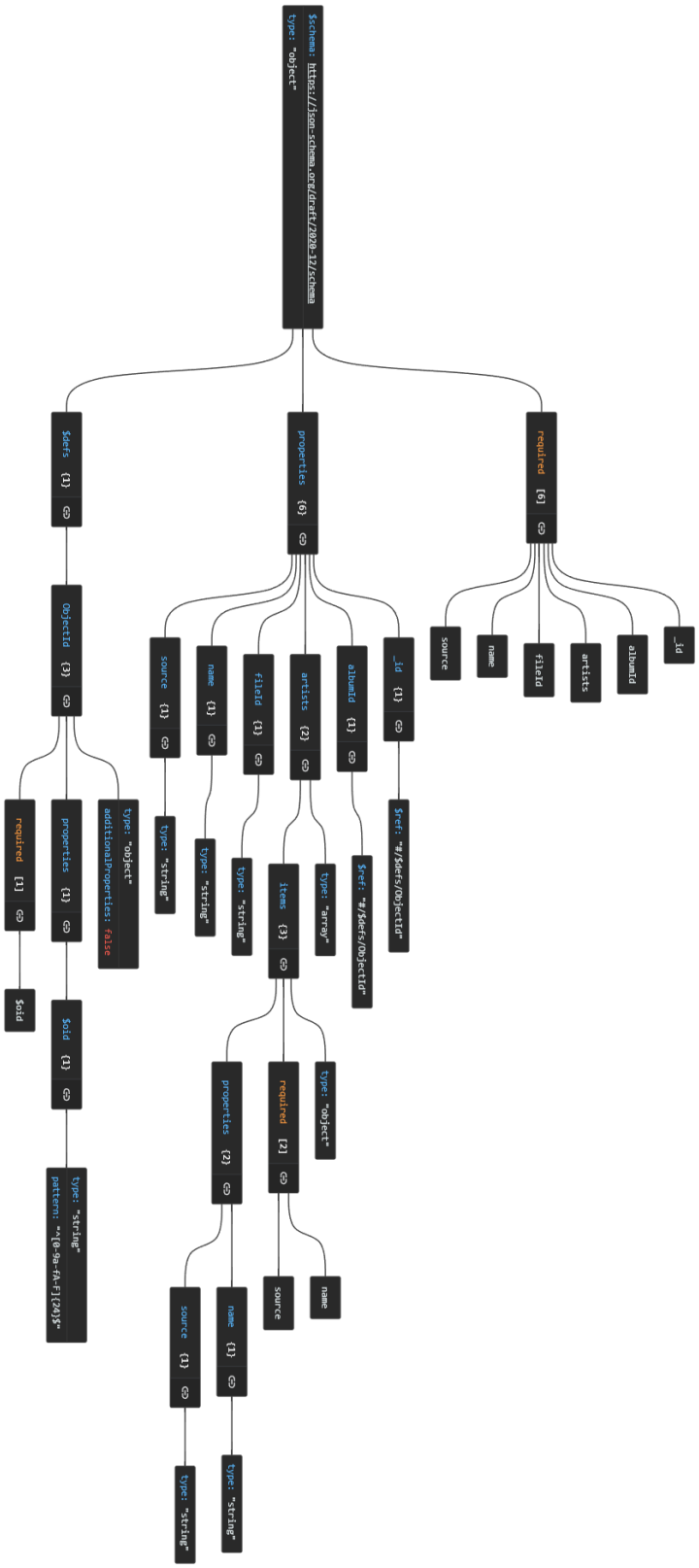
					ІАЛЦ.467200.006 ДЗ	
		№ докум.	Підпис	Дата		
Розробив		Полтавський В.Д.			*Багатокористувацька гравікторина з використанням мікросервісної архітектури* Схематика бази даних	Літ. / Аркуш 1 / Аркушів 4
Перевірив		Русінов В. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13
Н. Контр.						
Затвердив						



					ІАЛЦ.467200.006 ДЗ	Арк.
						2
Зм.	Арк.	№ докум.	Підпис	Дата		



					ІАЛЦ.467200.006 ДЗ	Арк.
						3
Зм.	Арк.	№ докум.	Підпис	Дата		



					ІАЛЦ.467200.006 ДЗ	Арк.
						4
Зм.	Арк.	№ докум.	Підпис	Дата		

Додаток 4

Багатокористувацька гра-вікторина з використанням мікросервісної архітектури

Мапінг ElasticSearch

ІАЛЦ.467200.007 Д4

Аркушів 5

Київ - 2025

```
{
   "settings": {
      "number_of_shards": 1,
      "max_ngram_diff": 17,
      "index": {
         "analysis": {
            "analyzer": {
               "symbol_analyzer": {
                  "type": "custom",
                  "tokenizer": "standard",
                  "filter": [
                     "lowercase",
                     "limited_ngram"
                  ]
               }
            },
            "filter": {
               "limited_ngram": {
                  "type": "ngram",
                  "min_gram": 3,
                  "max_gram": 20
               }
            }
         }
      }
   }
```

					ІАЛЦ.467200.007 Д4		
		№ докум.	Підпис	Дата			
Розробив		Полтавський В.Д.			*Багатокористувацька гра-вікторина з використанням мікросервісної архітектури* Технічне завдання	Літ.	Аркуш 1 / Аркушів 5
Перевірив		Русінов В. В.					
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13	
Н. Контр.							
Затвердив							

```
    },
    "mappings": {
      "properties": {
        "id": {
          "type": "keyword"
        },
        "name": {
          "type": "text",
          "analyzer": "symbol_analyzer",
          "search_analyzer": "standard"
        },
        "avatarId": {
          "type": "keyword"
        },
        "genres": {
          "type": "keyword"
        }
      }
    }
}
{
    "settings": {
      "number_of_shards": 1,
      "max_ngram_diff": 17,
      "index": {
```

```
        "analysis": {
          "analyzer": {
            "symbol_analyzer": {
              "type": "custom",
              "tokenizer": "standard",
              "filter": [
                "lowercase",
                "limited_ngram"
              ]
            }
          },
          "filter": {
            "limited_ngram": {
              "type": "ngram",
              "min_gram": 3,
              "max_gram": 20
            }
          }
        }
      },
      "mappings": {
        "properties": {
          "id": {
            "type": "keyword"
```

```
},
"isFull": {
    "type": "boolean"
},
"name": {
    "type": "text",
    "analyzer": "symbol_analyzer",
    "search_analyzer": "standard"
},
"coverId": {
    "type": "keyword"
},
"genres": {
    "type": "keyword"
},
"tracks": {
    "type": "object",
    "properties": {
        "id": {
            "type": "keyword"
        },
        "name": {
            "type": "text",
            "analyzer": "symbol_analyzer",
            "search_analyzer": "standard"
```

```
                }
            }
        },
        "artists": {
            "type": "object",
            "properties": {
                "id": {
                    "type": "keyword"
                },
                "name": {
                    "type": "text",
                    "analyzer": "symbol_analyzer",
                    "search_analyzer": "standard"
                },
                "avatarId": {
                    "type": "keyword"
                }
            }
        }
    }
}
```

Додаток 5

Багатокористувацька гра-вікторина з використанням мікросервісної архітектури

Лістинг коду

ІАЛЦ.467200.008 Д5

Аркушів 20

Київ - 2025

```php
<?php

namespace App\ArtistParser\Application\Handler;

use App\ArtistParser\Application\Parser\AlbumParser;
use App\ArtistParser\Application\Serializer\TrackSerializer;
use App\ArtistParser\Domain\Entity\TrackWithAlbum;
use App\ArtistParser\Domain\Repository\PlaylistRepositoryInterface;
use App\Shared\Application\Interface\CommandBusInterface;
use
MusicPlayground\Contract\Application\Playlist\Command\OnCreatedPlaylistCom
mand;
use
MusicPlayground\Contract\Application\SongParser\Command\UpdateAlbumForPl
aylistCommand;

final readonly class OnCreatePlaylistCommandHandler
{
    public function __construct(
        private CommandBusInterface $bus,
        private PlaylistRepositoryInterface $repository,
        private AlbumParser $albumParser,
        private TrackSerializer $trackSerializer,
        private int $maxTrackCount = 30
    ) {
    }
```

```
    public function __invoke(OnCreatedPlaylistCommand $command): void
    {
        $track = $this->repository->getTracks($command->source->id);
        for ($trackNum = 0; $trackNum < $this->maxTrackCount && $track->valid()
== true; $trackNum++ & $track->next()) {
            /** @var TrackWithAlbum $current */
            $current = $track->current();

            $this->bus->dispatch(new UpdateAlbumForPlaylistCommand(
                $command->operationId,
                $command->id,
                $this->albumParser->parse($current->getAlbum()),
                $this->trackSerializer->toPartialDTO($current->getTrack())
            ));
        }
    }
}

<?php
declare(strict_types = 1);

namespace App\ArtistParser\Application\Handler;

use App\ArtistParser\Application\Command\ParseTrackCommand;
use App\ArtistParser\Domain\Repository\TrackRepositoryInterface;
use App\Shared\Application\Interface\CommandBusInterface;
use
MusicPlayground\Contract\Application\SongParser\Command\OnUpdateAlbumC
ommand;
```

```
final readonly class OnUpdateAlbumCommandHandler
{
    public function __construct(
        private CommandBusInterface $bus,
        private TrackRepositoryInterface $service
    ) {
    }

    public function __invoke(OnUpdateAlbumCommand $command): void
    {
        $tracks = $this->service->getByAlbumsId($command->source->id);

        foreach ($tracks as $track) {
            if (in_array($track->getSource()->getId(), $command->containsTracks,
true) === false) {
                $this->bus->dispatch(new ParseTrackCommand($command->albumId,
$track));
            }
        }
    }
}
<?php

declare(strict_types = 1);

namespace App\ArtistParser\Application\Handler;

use App\ArtistParser\Application\Parser\AlbumParser;
```

```
use App\ArtistParser\Domain\Repository\AlbumRepositoryInterface;
use App\Shared\Application\Interface\CommandBusInterface;
use
MusicPlayground\Contract\Application\SongParser\Command\OnUpdateArtistCo
mmand;
use
MusicPlayground\Contract\Application\SongParser\Command\UpdateAlbumCom
mand;

final readonly class OnUpdateArtistCommandHandler
{
   public function __construct(
      private CommandBusInterface $bus,
      private AlbumRepositoryInterface $albumService,
      private AlbumParser $parser
   ) {
   }

   public function __invoke(OnUpdateArtistCommand $command): void
   {
      foreach ($this->albumService->getByArtistId($command->source->id) as
$album) {
         if (in_array($album->getSource()->getId(), $command->containsAlbums)
=== false) {
            $this->bus->dispatch(new UpdateAlbumCommand($this->parser-
>parse($album)));
         }
      }
   }
```

```
}

<?php

declare(strict_types = 1);

namespace App\ArtistParser\Application\Handler;

use App\ArtistParser\Application\Command\ParseArtistCommand;
use App\ArtistParser\Application\Saver\FileSaver;
use App\ArtistParser\Application\Serializer\ArtistSerializer;
use App\ArtistParser\Domain\Repository\ArtistRepositoryInterface;
use App\Shared\Application\Interface\CommandBusInterface;
use
MusicPlayground\Contract\Application\SongParser\Command\UpdateArtistComm
and;

final readonly class ParseArtistCommandHandler
{
    private ArtistSerializer $serializer;

    public function __construct(
        private CommandBusInterface $bus,
        private ArtistRepositoryInterface $artistService,
        private FileSaver $avatarSaver
    ) {
        $this->serializer = new ArtistSerializer();
    }
```

```
    public function __invoke(ParseArtistCommand $command): void
    {
        $artist = $this->artistService->getById($command->getId());
        $avatarUrl = $artist->getAvatarUrl();
        $avatarId = $avatarUrl !== null ? $this->avatarSaver->saveByUrl($avatarUrl)
: null;

        $this->bus->dispatch(new UpdateArtistCommand($this->serializer-
>toDTO($artist, $avatarId)));
    }
}
```

```
<?php

namespace App\ArtistParser\Application\Handler;

use App\ArtistParser\Application\Saver\FileSaver;
use App\ArtistParser\Application\Serializer\PlaylistSerializer;
use App\ArtistParser\Domain\Exception\PlaylistInvalidIdException;
use App\ArtistParser\Domain\Exception\PlaylistNotFoundException;
use App\ArtistParser\Domain\Repository\PlaylistRepositoryInterface;
use App\Shared\Application\Event\OperationExceptionEvent;
use App\Shared\Application\Interface\CommandBusInterface;
use
MusicPlayground\Contract\Application\Playlist\Command\CreatePlaylistComman
d;
use
MusicPlayground\Contract\Application\SongParser\Command\ParsePlaylistComm
and;
```

```
use Psr\EventDispatcher\EventDispatcherInterface;

final readonly class ParsePlaylistCommandHandler
{
    public function __construct(
        private CommandBusInterface $bus,
        private PlaylistRepositoryInterface $repository,
        private PlaylistSerializer $serializer,
        private EventDispatcherInterface $eventDispatcher,
        private FileSaver $fileSaver
    ) {
    }

    public function __invoke(ParsePlaylistCommand $command): void
    {
        try {
            $playlist = $this->repository->getById($command->playlistId);
            $coverId = $playlist->cover !== null ? $this->fileSaver-
>saveByUrl($playlist->cover) : null;
        } catch (PlaylistNotFoundException|PlaylistInvalidIdException $exception) {
            $this->eventDispatcher->dispatch(new
OperationExceptionEvent($command, $exception));

            return;
        }

        $this->bus->dispatch(new CreatePlaylistCommand(
            $command->operationId,
            $this->serializer->toDto($playlist, $coverId)
```

```
        ));
    }
}

<?php

declare(strict_types = 1);

namespace App\ArtistParser\Infrastructure\Parser;

use App\ArtistParser\Domain\Entity\SimpleArtist;
use App\ArtistParser\Domain\Entity\Track;
use App\ArtistParser\Domain\Parser\TrackParserInterface;
use App\Shared\Application\Interface\ObjectStorageInterface;
use Google\Service\Exception;
use Google\Service\YouTube;
use RuntimeException;

final readonly class YoutubeTrackParser implements TrackParserInterface
{
    public function __construct(
        private YouTube $youtubeService,
        private ObjectStorageInterface $objectStorage,
        private string $filesPath,
        private string $cookiePath
    ) {
    }

    /**
```

```
     * @throws Exception
     */
    public function parseFile(Track $dto): string
    {
        $artistsIds = array_map(fn (SimpleArtist $artist) => $artist->getName(), $dto->getArtists());
        $name = join(' & ', $artistsIds) . ' - ' . $dto->getName();
        $fileName = md5($name);

        if ($this->objectStorage->has($fileName) === true) {
            return $fileName;
        }

        $videoId = $this->findVideoByName($name);
        $filePath = $this->downloadAudioFromVideo($fileName, $videoId);
        $file = file_get_contents($filePath);

        unlink($filePath);
        $this->objectStorage->save($fileName, $file);

        return $fileName;
    }

    /**
     * @throws Exception
     */
    private function findVideoByName(string $name): string
    {
        $response = $this->youtubeService->search->listSearch('snippet', [
```

```
            'maxResults' => 1,
            'order' => 'relevance',
            'q' => $name,
            'type' => 'video',
            'videoCategoryId' => 10,
        ]);

        $videoId = $response->getItems()[0]?->getId()->getVideoId();

        if ($videoId === null) {
            throw new RuntimeException('No video found for this track');
        }

        return $videoId;
    }

    private function downloadAudioFromVideo(string $fileName, string $videoId):
string
    {
        $filePath = "$this->filesPath/$fileName.mp3";

        shell_exec("yt-dlp -x --audio-format mp3 -o \"$filePath\"
https://www.youtube.com/watch?v=$videoId");

        if (is_file($filePath) === false) {
            throw new RuntimeException('Bad attempt to download video');
        }

        return $filePath;
```

```
    }
}

<?php

declare(strict_types = 1);

namespace App\ArtistParser\Infrastructure\Repository;

use App\ArtistParser\Domain\Entity\Album;
use App\ArtistParser\Domain\Entity\SimpleArtist;
use App\ArtistParser\Infrastructure\Mapper\AlbumMapper;
use MusicPlayground\Elephy\Entity\SimpleArtist as SpotifySimpleArtist;
use App\ArtistParser\Domain\Repository\AlbumRepositoryInterface;
use App\ArtistParser\Domain\ValueObject\AlbumSource;
use App\ArtistParser\Domain\ValueObject\ArtistSource;
use DateTimeImmutable;
use Generator;
use InvalidArgumentException;
use MusicPlayground\Elephy\Interface\SpotifyApiInterface;

final readonly class SpotifyAlbumRepository implements
AlbumRepositoryInterface
{
    public function __construct(
        private SpotifyApiInterface $spotifyApi,
        private AlbumMapper $mapper
    ) {
    }
```

```
    public function getByArtistId(string $id): Generator
    {
        $albumsPagination = $this->spotifyApi->getArtistsAlbums(
            $id, includeGroups: ['album', 'single']
        );

        foreach ($albumsPagination as $body) {
            yield from array_map(fn ($album) => $this->mapper->map($album),
$body->albums);
        }
    }
}

<?php

namespace App\Core\Application\EventListener;

use App\Core\Application\Event\DomainInsertEvent;
use App\Core\Application\Event\DomainUpdateEvent;
use App\Core\Application\Serializer\AlbumSerializer;
use App\Core\Domain\Entity\Album;
use App\Core\Domain\Repository\Artist\ArtistRepositoryInterface;
use App\Shared\Application\Interface\CommandBusInterface;
use
MusicPlayground\Contract\Application\SongParser\Command\OnUpdateAlbumFu
llCommand;
use Symfony\Component\EventDispatcher\Attribute\AsEventListener;
```

```
#[AsEventListener(event: DomainInsertEvent::class, method: 'onInsert')]
#[AsEventListener(event: DomainUpdateEvent::class, method: 'onUpdate')]
final readonly class DispatchOnAlbumUpdateFullCommand
{
    public function __construct(
        private ArtistRepositoryInterface $artistRepository,
        private AlbumSerializer $serializer,
        private CommandBusInterface $bus
    ) {
    }

    public function onInsert(DomainInsertEvent $event): void
    {
        $this->dispatch($event->entity);
    }

    public function onUpdate(DomainUpdateEvent $event): void
    {
        $this->dispatch($event->entity);
    }

    private function dispatch(object $album): void
    {
        if ($album instanceof Album === false) {
            return;
        }

        $allArtists = $album->getSimpleArtists();
        $artists = $this->artistRepository->concat($allArtists);
```

```
        $this->bus->dispatch(new OnUpdateAlbumFullCommand(
            $this->serializer->toFullDto($album, $artists)
        ));
    }
}
<?php

namespace App\Core\Application\EventListener;

use App\Core\Application\Event\DomainInsertEvent;
use App\Core\Application\Event\DomainUpdateEvent;
use App\Core\Domain\Entity\Track;
use App\Shared\Application\Interface\CommandBusInterface;
use
MusicPlayground\Contract\Application\SongParser\Command\OnUpdateTrackCo
mmand;
use Symfony\Component\EventDispatcher\Attribute\AsEventListener;

#[AsEventListener(event: DomainInsertEvent::class, method: 'onInsert')]
#[AsEventListener(event: DomainUpdateEvent::class, method: 'onUpdate')]
final readonly class DispatchOnTrackUpdateCommand
{
    public function __construct(
        private CommandBusInterface $bus
    ) {
    }

    public function onInsert(DomainInsertEvent $event): void
```

```
    {
        $this->dispatch($event->entity);
    }

    public function onUpdate(DomainUpdateEvent $event): void
    {
        $this->dispatch($event->entity);
    }

    private function dispatch(object $track): void
    {
        if ($track instanceof Track === false) {
            return;
        }

        $this->bus->dispatch(new OnUpdateTrackCommand(
            $track->getId(),
            $track->getName(),
            $track->getAlbumId()
        ));
    }
}
```

```
<?php

namespace App\Core\Infrastructure\Doctrine\Repository;

use App\Core\Domain\Entity\Artist;
use App\Core\Domain\Entity\ArtistCast;
```

```
use App\Core\Domain\Entity\SimpleArtist;
use App\Core\Domain\Exception\ArtistNotFoundException;
use App\Core\Domain\Repository\Artist\ArtistRepositoryInterface;
use App\Core\Domain\Repository\Artist\SearchParams;
use App\Core\Domain\ValueObject\ArtistAvatar;
use App\Core\Domain\ValueObject\IdSource;
use App\Core\Infrastructure\Util\PreviewArtistsFactory;
use App\Shared\Domain\Repository\LockMode;
use App\Shared\Domain\ValueObject\Pagination;
use Doctrine\ODM\MongoDB\DocumentManager;
use Doctrine\ODM\MongoDB\LockException;
use Doctrine\ODM\MongoDB\Mapping\MappingException;
use Doctrine\ODM\MongoDB\MongoDBException;
use Doctrine\ODM\MongoDB\Repository\DocumentRepository;

final readonly class MongoArtistRepository implements ArtistRepositoryInterface
{
    private DocumentRepository $repository;

    public function __construct(
        private DocumentManager $dm,
        private PreviewArtistsFactory $previewArtistsFactory,
    ) {
        $this->repository = $this->dm->getRepository(Artist::class);
    }

    public function save(Artist $artist): void
    {
        $this->dm->persist($artist);
```

```
    }

    /**
     * @throws MappingException
     * @throws LockException
     * @throws ArtistNotFoundException
     */
    public function getById(string $id, LockMode $lock = LockMode::NONE):
Artist
    {
        $artist = $this->repository->find($id);

        if ($artist === null) {
            throw new ArtistNotFoundException();
        }

        return $artist;
    }

    /**
     * @throws MappingException
     * @throws LockException
     */
    public function findById(string $id, LockMode $lock = LockMode::NONE):
?Artist
    {
        return $this->repository->find($id);
    }
```

```
    /**
     * @throws ArtistNotFoundException
     * @throws MappingException
     * @throws LockException
     */
    public function getCastById(string $id): ArtistCast
    {
        $artist = $this->getById($id);

        return $this->artistToCast($artist);
    }

    public function getCastAll(Pagination $pagination, ?SearchParams $params =
null): array
    {
        if ($pagination->getCount() === 0) return [];

        $artists = $this->repository->findBy(
            $params?->sources !== null ? ['source' => ['$in' => $params->sources]] : [],
            ['id' => -1],
            $pagination->getCount(),
            $pagination->getFrom()
        );

        return array_map(fn (Artist $artist) => $this->artistToCast($artist), $artists);
    }

    /**
     * @throws MongoDBException
```

```
     */
    public function count(): int
    {
        return $this->repository->createQueryBuilder()->find()->count()-
>getQuery()->execute();
    }

    public function concat(array $artists): array
    {
        $sources = array_map(fn (SimpleArtist $artist) => $artist->getSource(),
$artists);

        $existingArtists = $this->repository->findBy([
            'source' => ['$in' => $sources]
        ]);

        return $this->previewArtistsFactory->create($artists, $existingArtists);
    }

    /**
     * @throws MappingException
     * @throws LockException
     * @throws ArtistNotFoundException
     */
    public function delete(string $id): void
    {
        $artist = $this->getById($id, LockMode::PESSIMISTIC_EXCLUSIVE);

        $this->dm->remove($artist);
```

```
    }

    private function artistToCast(Artist $artist): ArtistCast
    {
        $source = $artist->getSource();

        return new ArtistCast(
            $artist->getId(),
            $artist->getName(),
            (string)$source,
            $artist->getGenres(),
            new ArtistAvatar($artist->getAvatarId())
        );
    }

    public function findBySource(IdSource $source, LockMode $lock =
LockMode::NONE): ?Artist
    {
        return $this->repository->findOneBy(['source' => $source])
```