Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп'ютерних наук і технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти – бакалавр
за освітньо-професійною програмою
«Комп'ютерні науки»
зі спеціальності
122 – Комп'ютерні науки

тема роботи:

«Розробка веб-застосунку для інтелектуального підбору музичного контенту на основі емоційного стану користувача»

Виконав ст. гр. КН-22-1 _____ Кучеренко В. С.

Керівник _____ Кузнецов Д. І.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2026

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет: *інформаційних технологій*

Кафедра: *автоматизації, комп'ютерних наук і технологій*

Ступінь вищої освіти: *Бакалавр*

Спеціальність: *122 – Комп'ютерні науки*

ЗАТВЕРДЖУЮ

Зав. кафедрою: *к.т.н. Рубан С.А.*

_____

« *26* » *січня* 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу бакалавра

студентові групи КН-22-1 *Кучеренко Владиславу Сергійовичу*

1. Тема кваліфікаційної роботи: *«Розробка веб-застосунку для інтелектуального підбору музичного контенту на основі емоційного стану користувача»*

затверджено наказом по університету № *173с* від *30.03.2026 р.*

2. Термін здачі кваліфікаційної роботи: *10.06.2026 р.*

3. Склад кваліфікаційної роботи: *Пояснювальна записка обсягом 85с., додатки, презентація у Microsoft PowerPoint (13 слайдів) в електронному та друкованому вигляді*

4. Консультанти кваліфікаційної роботи:

Розділ 1-2 доц. Кузнецов Д. І.

Нормоконтроль доц. Маринич І. А.

5. Календарний план:

№	Етапи роботи	Термін виконання
1	*Вступ*	*04.04.26*
2	*Розділ 1*	*12.04.26*
3	*Розділ 2*	*22.04.26*
4	*Висновки*	*11.05.26*
5	*Оформлення кваліфікаційної роботи*	*13.05.26*
6	*Підготовка презентації та графічного матеріалу*	*16.05.26*
7	*Підготовка доповіді до захисту*	*20.05.26*

6. Дата видачі завдання: *28.01.2026р.*

Керівник _____ / Кузнецов Д. І./

7. Запевнення: *Я, Кучеренко Владислав Сергійович, запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.*

Із чинним Положенням про академічну доброчесність Криворізького національного університету ознайомлений.

Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі умисних порушень робота не допускається до захисту або оцінюється незадовільно.

Студент _____ / Кучеренко В. С./

АНОТАЦІЯ

Кучеренко В.С. Розробка веб-застосунку для інтелектуального підбору музичного контенту на основі емоційного стану користувача: кваліфікаційна робота бакалавра : 122 - Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2026. 85 с.

Метою роботи є створення вебзастосунку для автоматизованої генерації музичних плейлистів на основі тегів або текстових описів настрою із використанням великих мовних моделей.

У першому розділі проаналізовано рекомендаційні системи в музичній індустрії та обґрунтовано переваги семантичного пошуку (LLM-підходів) над традиційними методами фільтрації.

У другому розділі розроблено архітектурні та програмні рішення для реалізації вебзастосунку. Описано процес проєктування системи на основі архітектурного шаблону ASP.NET Core (MVC), що забезпечує високу масштабованість та стабільність. Реалізовано багаторівневу структуру бази даних у середовищі MSSQL, що забезпечує транзакційну цілісність даних. Ключовим етапом роботи стала інтеграція зовнішніх API: за допомогою Google Gemini API реалізовано механізм семантичного розбору текстових запитів користувача, а через Spotify Web API забезпечено доступ до широкої музичної бібліотеки. Впроваджено систему автентифікації на базі ASP.NET Core Identity та механізми захисту від перевантажень (кешування, Fallback-алгоритми).Практична цінність роботи полягає у створенні інструменту для швидкого й інтуїтивного пошуку високорелевантної музики природною мовою, що суттєво покращує користувацький досвід.

Ключові слова: ШТУЧНИЙ ІНТЕЛЕКТ, РЕКОМЕНДАЦІЙНА СИСТЕМА, МУЗИЧНИЙ ВЕБЗАСТОСУНОК, ВЕЛИКА МОВНА МОДЕЛЬ, ASP.NET CORE, SPOTIFY API, GEMINI API, MSSQL, СЕМАНТИЧНИЙ ПОШУК.

ЗМІСТ

ВСТУП

У сучасному світі музика є частиною повсякденного життя людини. Зі зростанням кількості доступного контенту на стрімінгових платформах користувачі все частіше витрачають значну кількість часу на пошук треків, що відповідають їхньому поточному стану, замість самого прослуховування. Традиційні системи пошуку за ключовими словами або жанрами часто не здатні вловити тонкий емоційний контекст чи специфічну потребу користувача.

Впровадження технологій штучного інтелекту, зокрема великих мовних моделей (LLM), відкриває нові можливості для розуміння природної мови та прихованих інтенцій користувача. Актуальним є розробка інтелектуального веб-додатку, здатного аналізувати текстові описи настрою та конвертувати їх у релевантні музичні підбірки за допомогою інтеграції зі Spotify API.

Метою роботи є розробка та програмна реалізація інтелектуального веб-додатку для персоналізованого підбору музики на основі аналізу емоційного стану користувача з використанням штучного інтелекту.

Для досягнення поставленої мети були сформульовані наступні завдання:

- проаналізувати предметну область та недоліки існуючих систем;
- дослідити можливості застосування алгоритмів обробки природної мови для розпізнавання настрою;
- спроєктувати архітектуру веб-додатку та структуру реляційної бази даних;
- реалізувати взаємодію із зовнішніми сервісами (Gemini API, Spotify API);
- розробити клієнтську частину (UI) та панель адміністрування з системою управління доступом;
- провести модульне тестування та оптимізацію швидкодії системи.

Результатом роботи є веб-додаток побудований на базі фреймворку ASP.NET Core. Розроблена система може бути використана як самостійний сервіс для допомоги користувачам у пошуку музики, а застосовані архітектурні рішення можуть слугувати основою для створення складніших аналітичних систем у сфері розваг та психологічної підтримки.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ СУЧАСНОГО СТАНУ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ У ГАЛУЗІ ПЕРСОНАЛІЗОВАНОГО ПІДБОРУ МУЗИЧНОГО КОНТЕНТУ

1.1 Аналіз предметної області

1.1.1 Загальна характеристика досліджуваного процесу

Споживання музичного контенту в сучасну цифрову епоху - це багатогранний психологічний та когнітивний процес, що виходить далеко за межі простого прослуховування аудіосигналів. У своїй основі цей процес є інструментом афективної саморегуляції, який дозволяє індивіду керувати власним емоційним станом, адаптуватися до зовнішніх стимулів та створювати необхідну атмосферу для різних видів діяльності. З переходом від фізичних носіїв та локальних бібліотек до глобальних хмарних стрімінгових платформ відбулася парадигмальна зміна: від моделі «володіння контентом» до моделі «доступу за вимогою» (Music-as-a-Service, MaaS). Це забезпечило користувачам миттєвий доступ до практично необмежених масивів даних, проте водночас висунуло нові вимоги до механізмів курації та фільтрації інформації.

З системно-орієнтованої та обчислювальної точок зору, процес персоналізованого підбору музики можна представити як динамічну послідовність взаємодій між вхідними даними (наміром або станом користувача) та вихідними результатами (релевантним аудіопотоком). Кожна музична одиниця в сучасних базах даних описується через багатовимірний простір ознак. Ці ознаки поділяються на два рівні:

1. Низькорівневі (об'єктивні) атрибути: метадані (виконавець, альбом, дата релізу) та технічні параметри звукового сигналу (темп у BPM, тональність, енергійність, рівень акустичності).

2. Високорівневі (суб'єктивні) атрибути: емоційне забарвлення, жанрова приналежність, відповідність конкретним життєвим ситуаціям (контекст).

Проблема полягає в тому, що сучасний слухач рідко формує свій запит, оперуючи об'єктивними параметрами. Натомість його інтенція виражається через опис поточного психофізичного стану («відчуваю втому», «потрібен драйв») або контексту середовища («тренування в залі», «нічна поїздка», «підготовка до іспиту»). Це створює ситуацію, яку в галузі інформаційного пошуку називають семантичним розривом (semantic gap) - критичну дистанцію між абстрактним людським відчуттям та технічними критеріями, за якими індексується музика в базах даних. [16]

Процес подолання цього розриву є високомодульним і заснованим на моделях перетворення даних. Він включає етапи розпізнавання природної мови, вилучення ключових сутностей (entities) та їх подальший мапінг (mapping) на відповідні категорії в музичних API. Така структура процесу ідеально піддається алгоритмічному представленню та автоматизації за допомогою штучного інтелекту.

Використання великих мовних моделей дозволяє системі інтерпретувати метафори, контекст та емоційні відтінки в тексті користувача, трансформуючи їх у чіткі параметри для пошуку.

Таким чином, інтелектуальний підбір музики стає не просто технічною функцією, а адаптивним інтерфейсом, що забезпечує синергію між обчислювальною потужністю алгоритмів та складністю людської психіки.

1.1.2 Основні виклики перед користувачами

У сучасному цифровому середовищі користувачі функціонують в умовах перманентного інформаційного перевантаження (Information Overload). Незважаючи на те що обсяг доступного музичного контенту досяг безпрецедентних масштабів, процес пошуку «ідеального треку» все частіше супроводжується когнітивною втомою. З точки зору психології та проєктування інтерфейсів (HCI), це явище описується як «парадокс вибору» (Paradox of Choice). Замість того, щоб відчувати задоволення від широти асортименту, мозок користувача змушений витрачати значні обчислювальні ресурси на оцінку,

порівняння та відмову від тисяч альтернативних варіантів, що неминуче призводить до фрустрації та ускладнює процес прийняття остаточного рішення.

Однією з найбільш нагальних архітектурних та користувацьких проблем є подолання «семантичного розриву» (Semantic Gap) між людським сприйняттям та машинним поданням даних. Традиційний детермінований підхід до пошуку вимагає від людини експліцитного, точного знання того, що саме вона хоче почути (назви, виконавці, теги). Уявіть типовий сценарій: користувач повертається додому після виснажливого робочого дня під дощем. Його справжній внутрішній запит формується на рівні високих емоційних абстракцій: «я критично втомився і потребую меланхолійної, але заспокійливої атмосфери для рефлексії».

Задоволення такої комплексної потреби за допомогою стандартного лексичного пошуку змушує людину штучно спрощувати свій запит, самостійно згадуючи назви сумних пісень або шукаючи готові плейлисти за примітивними ключовими словами (наприклад, "sad", "rain", "relax"). Класичні пошукові рушії часто видають на такі запити узагальнені, семантично плоскі та нерелевантні результати, оскільки вони шукають прямі текстові збіги, а не аналізують настрій. Такий ручний процес не лише експоненціально збільшує час пошуку контенту (показник Time-to-Content), але й руйнує крихкий емоційний фон користувача ще до початку прослуховування.

Іншим критичним викликом є алгоритмічна інертність сучасних стрімінгових платформ. Хоча індустрія робить сильний акцент на персоналізації (Personalization Engines), у реальності такі системи спираються переважно на екстраполяцію минулої історії прослуховувань. Це створює явище «бульбашки фільтрів» (Filter Bubble) та специфічний прояв проблеми холодного старту для нових емоцій. Якщо профіль користувача історично сформований на енергійному хіп-хопі або поп-музиці, рекомендаційна система ігноруватиме його раптову, ситуативну потребу в класичній музиці для глибокої концентрації.

Алгоритми, оптимізовані на максимізацію утримання уваги (Retention), продовжуватимуть пропонувати знайомий контент, розцінюючи новий

емоційний запит як статистичну похибку або тимчасову аномалію. У результаті користувач не отримує цілеспрямованої та швидкої відповіді на свій поточний запит, фактично залишаючись заручником власного цифрового минулого та алгоритмічного детермінізму платформи.

1.1.3 Можливості оптимізації процесу підбору шляхом автоматизації та ШІ

Взаємодія сучасного слухача зі стрімінговими екосистемами обтяжена значною кількістю рутинних мікротранзакцій: постійне створення тематичних плейлистів, сортування треків, фільтрація нових релізів та ручне коригування черги відтворення. Виконання цих завдань вимагає високої когнітивної залученості та часто перетворює процес споживання розважального контенту на своєрідну адміністративну роботу. Відповідно, виникає гостра потреба в оптимізації цього процесу. Фундаментальним інструментом для розв'язання цієї проблеми виступають технології обробки природної мови (Natural Language Processing, NLP) та машинне навчання.

Одним із найефективніших застосувань штучного інтелекту в цій предметній області є автоматична екстракція сутностей (Entity Extraction) та семантичний мапінг. Традиційні пошукові інтерфейси змушують користувача оперувати жорсткими, заздалегідь визначеними категоріями (наприклад, випадаючі списки з жанрами або повзунки для вибору темпу чи енергійності). Натомість інтелектуальна система здатна проаналізувати вільний, неструктурований текстовий опис емоції або ситуації та алгоритмічно перевести його у структурований формат.

Наприклад, абстрактний, експресивний або навіть сленговий вираз користувача на кшталт «хочу рознести цю кімнату» або «потрібен потужний заряд для важкого тренування» алгоритм класифікує та конвертує у точні, машинозчитувані параметри пошуку (наприклад, Mood: Aggressive, Activity: Workout). Такий підхід не лише мінімізує час взаємодії з інтерфейсом (показник Time-on-Task), але й радикально підвищує релевантність результату, усуваючи необхідність ручного перебору десятків варіантів.

Найбільший трансформаційний потенціал у контексті оптимізації закладений у застосуванні великих мовних моделей (Large Language Models, LLM), таких як архітектури класу GPT або Gemini. Завдяки попередньому навчанню на гігантських масивах текстових даних, ці моделі здатні розуміти глибокий контекст, метафори, іронію та приховані інтенції користувача, які неможливо опрацювати за допомогою звичайного пошуку за ключовими словами.

Уявіть собі віртуального музичного асистента (Intelligent Agent), який функціонує не як звичайний пошуковий скрипт, а виступає в ролі «емульованого диджея-психолога». Він здатен диференціювати тонкі психологічні нюанси: наприклад, розрізняти потребу у «світлій ностальгії» від «глибокої меланхолії», і відповідно формувати різні запити до музичної бази даних.

Більше того, автоматизація за допомогою LLM дозволяє генерувати динамічні запити до спеціалізованих музичних API (таких як Spotify API) у режимі реального часу, абстрагуючи від користувача всю технічну складність пошуку. Переклавши тягар ручного підбору, налаштування фільтрів та складання плейлистів на алгоритми штучного інтелекту, користувач звільняє свої когнітивні ресурси для головного - безпосередньої насолоди від прослуховування контенту. Це знаменує концептуальний перехід від застарілої парадигми «самостійного пошуку музики» до сучасної моделі «автоматизованого забезпечення музичного супроводу».

1.1.4 Класифікація рекомендаційних систем у музичній індустрії

У сучасній цифровій музичній індустрії, де каталоги провідних стрімінгових платформ (таких як Spotify або Apple Music) налічують понад 100 мільйонів аудіотреків, проблема інформаційного перенавантаження вирішується виключно за допомогою рекомендаційних систем (Recommendation Systems, RS). Їхня головна мета - алгоритмічно передбачити вподобання користувача та редукувати простір пошуку до найбільш релевантного контенту.

Еволюція рекомендаційних алгоритмів у сфері потокового аудіо дозволяє виділити п'ять фундаментальних підходів до класифікації контенту, кожен з яких має власні архітектурні переваги та обмеження:

1. Контентно-орієнтовані системи (Content-Based Filtering) Цей класичний підхід базується на математичному аналізі внутрішніх атрибутів (метаданих) самих об'єктів. У контексті музики алгоритм аналізує такі акустичні характеристики треку, як темп (BPM), жанр, тональність, танцювальність (danceability) або акустичність. Система створює векторний профіль користувача на основі характеристик треків, які він слухав раніше, і рекомендує схожі за звучанням пісні.

– Перевага: Відсутність проблеми «холодного старту» для нових пісень (система може рекомендувати трек одразу після його завантаження в базу, спираючись лише на його звук).
– Недолік: Формування так званої «інформаційної бульбашки» (Filter Bubble) - алгоритм рідко пропонує користувачу щось принципово нове, обмежуючись уже знайомими жанрами.

2. Колаборативна фільтрація (Collaborative Filtering) На відміну від попереднього методу, колаборативна фільтрація ігнорує акустичні властивості музики та фокусується виключно на поведінковій матриці взаємодії між користувачами (User-Item Matrix). Алгоритм шукає патерни за принципом: «користувачі, які слухали ці пісні, також додали у плейлист наступний трек». Найчастіше реалізується через математичні методи факторизації матриць (наприклад, SVD) або алгоритми k-найближчих сусідів (k-NN).

– Перевага: Здатність генерувати несподівані, але релевантні рекомендації (Serendipity), розширюючи музичний кругозір слухача.
– Недолік: Проблема «холодного старту» для нових користувачів (про яких немає історії) та нових пісень (які ще ніхто не слухав), а також розрідженість даних (Data Sparsity).

3. Гібридні моделі (Hybrid Models): Оскільки контентна і колаборативна фільтрації мають критичні вразливості, сучасні комерційні гіганти (зокрема

Spotify у своєму алгоритмі Discover Weekly) використовують гібридні системи. Вони комбінують аналіз звукового сигналу (Raw Audio Analysis за допомогою згорткових нейромереж CNN) із графовими моделями поведінки користувачів. Це дозволяє нівелювати слабкі сторони обох базових підходів, забезпечуючи високу точність рекомендацій.

4. Контекстно-залежні системи (Context-Aware Recommendation Systems, CARS) Музика є високо емоційним контентом, сприйняття якого кардинально змінюється залежно від поточного середовища. Контекстно-залежні системи додають до матриці рекомендацій додаткові виміри: час доби, геолокацію, поточну погоду або тип активності. Наприклад, один і той самий користувач потребуватиме енергійного рок-плейлиста під час ранкового тренування в спортзалі та спокійного ембієнту під час вечірньої концентрації. Такі алгоритми використовують дерева рішень (Decision Trees) або контекстні тензори для підбору музики «в моменті».

5. Системи нового покоління на базі великих мовних моделей (LLM-powered) Найсучаснішим вектором розвитку, який ліг в основу даного дипломного проєкту, є використання генеративного штучного інтелекту (Generative AI) та великих мовних моделей (LLMs, таких як GPT-4 або Google Gemini) як ядра рекомендаційної системи. Традиційні підходи жорстко прив'язані до наперед визначених тегів та історії прослуховувань. Натомість LLM-powered системи здатні здійснювати глибокий семантичний аналіз (Semantic Analysis) неструктурованих запитів природною мовою. Вони розуміють метафори, складні емоційні стани та нестандартні описи (наприклад, "хочу відчути себе головним героєм нуарного фільму під час дощу").

Мовна модель виступає в ролі інтелектуального брокера: вона перекладає абстрактні людські емоції у строгий набір технічних параметрів (жанр, настрій, темп), які потім передаються до API музичного провайдера. Цей підхід забезпечує найвищий рівень емпатії системи та максимальну персоналізацію користувацького досвіду, усуваючи необхідність ручного пошуку музики за категоріями.

1.1.5 Музичні метадані: акустичні, технічні та емоційні параметри

Для того, щоб рекомендаційні системи та алгоритми штучного інтелекту могли ефективно обробляти та класифікувати музичний контент, аудіосигнал необхідно перетворити на структурований набір числових характеристик - музичні метадані. Провідні платформи стрімінгу (зокрема Spotify через свій сервіс Audio Features API) квантують кожен музичний трек, представляючи його у вигляді багатовимірного вектора ознак.

Ці метадані можна умовно розділити на об'єктивні (технічні) та суб'єктивні (емоційні та психоакустичні) параметри. Розуміння цих метрик є критично важливим для побудови систем семантичного пошуку, оскільки саме ними оперує мовна модель під час підбору релевантного контенту. До базових параметрів належать:

1. BPM (Tempo - Темп) Технічний параметр, що вимірюється в ударах на хвилину (Beats Per Minute). Він визначає загальну швидкість виконання композиції. Темп має пряму кореляцію з фізіологічною реакцією слухача: треки з високим BPM (130-170) стимулюють активність і підходять для занять спортом (Activity: Workout), тоді як низький BPM (60-90) сприяє релаксації та концентрації (Activity: Study, Relax).

2. Valence (Валентність / Музична позитивність) Емоційний параметр у діапазоні від 0.0 до 1.0, який визначає рівень "позитивності", що передається треком.

– Висока валентність (ближче до 1.0) означає, що трек звучить життєрадісно, щасливо, ейфорійно (Mood: Happy, Joyful).

– Низька валентність (ближче до 0.0) вказує на сумне, депресивне або тривожне звучання (Mood: Sad, Melancholic). Цей параметр є найважливішим маркером для нашого додатку при зіставленні психологічного стану користувача з аудіоконтентом.

3. Energy (Енергійність) Психоакустична метрика (від 0.0 до 1.0), що є мірою інтенсивності, гучності та активності треку. Енергійні треки зазвичай мають

швидкий темп, високу гучність і значний спектральний шум (наприклад, жанри Death Metal або Hardcore). Комбінація Valence та Energy утворює двовимірну емоційну площину Рассела (Russell's Circumplex Model of Affect), яка дозволяє класифікувати будь-яку емоцію.

4. Acousticness (Акустичність) Імовірнісна метрика (від 0.0 до 1.0), яка визначає ступінь впевненості алгоритму в тому, що трек є акустичним (створеним без використання електронних синтезаторів чи сильної цифрової обробки). Висока акустичність характерна для класичної музики, фолку та інструментального джазу.

5. Danceability (Танцювальність) Комплексний параметр (від 0.0 до 1.0), який оцінює, наскільки трек підходить для танців. Він розраховується на основі низки технічних характеристик: стабільності ритму, сили удару (beat strength), загального темпу та регулярності патернів. Найвищі показники танцювальності мають жанри Hip-Hop та EDM.

6. Key (Тональність) Структурно-технічний параметр, що визначає основну музичну тональність треку згідно зі стандартною хроматичною шкалою (наприклад, C = 0, C# = 1 і так далі). У поєднанні з параметром Mode (мажор/мінор) тональність суттєво впливає на емоційне забарвлення: мажорні тональності зазвичай сприймаються як світлі та оптимістичні, а мінорні - як темні, напружені або ліричні.

7. Genre Embeddings (Векторні представлення жанрів) Сучасні рекомендаційні системи відмовляються від жорсткої текстової класифікації жанрів (String-based classification). Натомість жанри перетворюються на щільні векторні представлення (Embeddings) у прихованому багатовимірному просторі (Latent Space). Наприклад, у такому просторі вектори жанрів "pop-rock" та "indie-rock" знаходитимуться математично близько один до одного, що дозволяє алгоритмам знаходити непрямі, неочевидні зв'язки між треками різних стилів і будувати плавні музичні переходи.

1.2 Огляд існуючих програмних засобів для рекомендації аудіоконтенту

1.2.1 Використання штучного інтелекту в музичній індустрії: потенціал та обмеження

Останніми роками спостерігається тотальна алгоритмізація розважальних платформ. Штучний інтелект (ШІ) сьогодні виступає не лише як інструмент для колаборативної фільтрації, але і як комплексний аналітичний механізм. Його застосовують для глибокого аналізу аудіосигналів (Music Information Retrieval), автоматичної генерації метаданих, прогнозування комерційного успіху (хітів) та навіть алгоритмічної композиції. Проте, з точки зору покращення користувацького досвіду (UX) та вирішення проблеми семантичного розриву, найперспективнішим напрямком є інтеграція розмовного ШІ та великих мовних моделей (LLM), таких як ChatGPT (OpenAI) або Gemini (Google), безпосередньо в процес пошуку контенту.

Потенціал використання LLM Головна цінність таких моделей полягає в їхній безпрецедентній адаптивності та здатності до семантичного аналізу природної мови. Замість жорстко заданих алгоритмів пошуку, ці моделі дозволяють створювати динамічні запити до баз даних, спираючись на живу, неструктуровану мову користувача.

Для розробників сучасних веб-додатків це є фундаментальним зсувом парадигми (paradigm shift): замість перевантаження інтерфейсу десятками чекбоксів, повзунків та фільтрів, достатньо надати користувачеві єдине текстове поле. ШІ бере на себе когнітивне навантаження з класифікації та мапінгу (mapping) абстрактних емоцій на конкретні технічні параметри.

Справжня технічна перевага цього підходу розкривається на рівні бекенд-архітектури. За допомогою методів інженерії підказок (Prompt Engineering), мовну модель можна налаштувати так, щоб вона не просто генерувала текстову відповідь-діалог, а працювала як мікросервіс-аналізатор. Отримавши текст користувача, модель повертає машинозчитувані структуровані дані (наприклад, суворий формат JSON із масивами Moods та Activities). Цей об'єкт автоматично парситься сервером додатку і слугує набором вхідних параметрів для

формування точних запитів до спеціалізованих музичних API (таких як Spotify API).

Обмеження та технічні виклики: Попри потужний потенціал, використання LLM у рекомендаційних системах супроводжується низкою технічних та архітектурних обмежень, які необхідно враховувати під час проєктування програмного забезпечення:

1. Проблема «галюцинацій» (Hallucinations): Мовні моделі схильні до генерації правдоподібного, але хибного контенту. У контексті музичного пошуку ШІ може згенерувати неіснуючі жанри, вигадані імена виконавців або повернути теги, яких немає в локальній базі даних додатку. Це вимагає впровадження жорстких валідаторів та алгоритмів очищення даних (Data Sanitization) на стороні сервера.

2. Затримка відповіді (Latency): Обробка запиту через сторонні хмарні API великих мовних моделей потребує часу (від кількох секунд і більше), що є відчутним для систем реального часу. У порівнянні з класичним SQL-запитом до локальної бази даних, ШІ-пошук є значно повільнішим.

3. Залежність від інфраструктури провайдера (Rate Limits та Availability): Використання хмарних API (наприклад, Gemini API) робить додаток вразливим до перевантажень серверів провайдера (помилки 503 Service Unavailable) та жорстких обмежень на кількість запитів за хвилину (Rate Limiting). Це зумовлює необхідність розробки архітектурних механізмів відмовостійкості: кешування частих запитів (In-Memory Caching) та систем резервного перемикання моделей (Fallback-алгоритмів).

4. Складність утримання довгострокового контексту: Хоча LLM чудово інтерпретують одиничний, ситуативний запит, підтримання довгострокового контексту музичних вподобань конкретного користувача без постійного використання дорогих векторних баз даних (Vector DB) або перенавчання моделі (Fine-Tuning) залишається нетривіальною архітектурною задачею.

Розуміння цих обмежень є критично важливим для створення надійної архітектури веб-додатку, де ШІ виступає не як безпомилковий оракул, а як потужний, але контрольований інструмент семантичного перекладу.

1.2.2 Порівняльний аналіз наявних рішень

На ринку домінують кілька глобальних сервісів. Кожен з них має власний підхід до алгоритмів рекомендацій. Розглянемо ключові з них:

Spotify використовує гібридну екосистему (AI + Collaborative Filtering + NLP). Найвизначнішою особливістю є здатність алгоритмів аналізувати аудіосигнали (темп, ритм) та порівнювати профілі мільйонів користувачів. Проте головний недолік полягає в інертності профілю - системі потрібен час, щоб зрозуміти, що музичні смаки користувача змінилися.

Apple Music робить ставку на експертну курацію (Human-in-the-loop). Плейлисти створюються музичними редакторами, що гарантує високу якість підбірок. Однак цьому сервісу бракує інструментів для миттєвої адаптації під унікальний настрій конкретного індивіда в реальному часі.

YouTube Music функціонує як контекстно-залежна система. Завдяки екосистемі Google, платформа аналізує геодані, час доби та історію пошуку відео. Це потужний інструмент, але він страждає від "шуму" у базі даних (наявність аматорських каверів, відеокліпів із зайвими звуками), що знижує якість чистого аудіодосвіду.

Pandora (Music Genome Project) аналізує музику на основі ручної розмітки експертами (Content-based filtering). Система підбирає пісні зі схожою гармонією чи вокалом. Хоча відповіді є музично точними, системі бракує розуміння емоційного контексту користувача.

Для систематизації інформації результати аналізу зведено у таблицю 1.1

Таблиця 1.1 - Комплексний порівняльний аналіз архітектурних рішень музичних сервісів

Платформа	Провідні алгоритми	Глибина аналізу контексту	Ключова перевага	Основний недолік
Spotify	Гібридний (CF + NLP + Аудіо-аналіз)	Висока (на основі довгострокової історії)	Висока точність персоналізованих міксів	Інертність профілю, створення «бульбашки фільтрів»
YouTube Music	Глибоке навчання (DL)	Критична (GPS, Час, Пошукова історія)	Глибока інтеграція з екосистемою пристрою	Високий рівень "шуму" від відеоконтенту
Apple Music	Редакційна курація + Статистика	Середня	Висока стилістична та естетична якість	Низька швидкість алгоритмічної адаптації
Pandora	Content-based (Music Genome Project)	Низька	Музикознавча точність підбору схожих треків	Відсутність розуміння абстрактних людських емоцій

Спільним недоліком існуючих систем є те, що вони вимагають від користувача адаптуватися до їхніх алгоритмів. Потреба в інструментах, які б приймали запити природною мовою та самостійно перетворювали їх на музичні теги за допомогою ШІ, залишається незадоволеною, що обґрунтовує доцільність розробки нашого веб-додатку.

1.2.3 Наукові підходи до формування музичних рекомендацій: від MIR до Deep Learning

Розвиток музичних рекомендаційних систем нерозривно пов'язаний з еволюцією методів обробки цифрових сигналів, машинного навчання та когнітивної психології. Сучасна комп'ютерна наука пропонує низку складних математичних та алгоритмічних підходів для того, щоб навчити машину «розуміти» семантику та емоційне забарвлення аудіоконтенту. Огляд фундаментальних наукових напрямків дозволяє виділити наступні ключові концепції:

1. Music Information Retrieval (MIR) Music Information Retrieval (пошук музичної інформації) - це базова міждисциплінарна галузь науки, що лежить в основі будь-якого сучасного музичного сервісу. MIR поєднує методи обробки сигналів, інформатики, машинного навчання та психоакустики. Головне завдання MIR - автоматичне вилучення значущої інформації безпосередньо з «сирого» (raw) аудіосигналу. Замість того, щоб покладатися на ручні текстові теги від редакторів, алгоритми MIR аналізують низькорівневі акустичні характеристики: мел-кепстральні коефіцієнти (MFCC), спектральний центроїд, енергію нульових перетинів тощо. На основі цих даних формуються високорівневі ознаки: жанр, інструментовка, ритм та темп. Дослідження таких вчених, як Дж. Дауні (J. Stephen Downie), заклали фундамент для автоматизованої каталогізації гігантських музичних баз даних.

2. Mood Classification (Класифікація настроїв) Завдання автоматичного розпізнавання емоцій у музиці (Music Emotion Recognition, MER) є одним із найскладніших напрямків у MIR. Науковий підхід до цієї проблеми базується на перекладі психологічних моделей у математичний простір. Найчастіше використовується двовимірна модель емоцій Тайєра (Thayer) або Рассела (Russell), де осі визначають рівень збудження (Arousal/Energy) та приємності (Valence). Алгоритми Mood Classification зіставляють акустичні ознаки (наприклад, дисонанс, мажорні/мінорні акорди) з координатами на цій емоційній площині. Складність полягає у суб'єктивності сприйняття: один і той самий трек

може викликати різні емоції у різних слухачів залежно від їхнього культурного бекграунду.

3. Deep Learning для класифікації настроїв Історично для класифікації настроїв використовувалися класичні алгоритми машинного навчання (Support Vector Machines, Random Forests). Проте справжній прорив відбувся із застосуванням глибокого навчання (Deep Learning). Сучасні архітектури аналізують не просто масиви чисел, а візуальні представлення звуку - мел-спектрограми (візуалізація частотного спектра у часі).

– Згорткові нейромережі (CNN): Широко використовуються для аналізу спектрограм як зображень. Вони здатні виявляти локальні акустичні патерни (наприклад, удар барабана або специфічний тембр гітари).

– Рекурентні нейромережі (RNN та LSTM): Застосовуються для аналізу часових залежностей, оскільки музика - це послідовний темпоральний процес, де емоція часто розкривається через розвиток мелодії в часі. Комбінація CNN (для вилучення ознак) та LSTM (для аналізу послідовності) сьогодні є стандартом індустрії (State-of-the-Art) для розпізнавання музичного настрою.

4. Audio Embeddings (Аудіо-ембедінги) Результатом роботи згаданих вище глибоких нейромереж є створення аудіо-ембедингів. Ембединг - це безперервне векторне представлення треку у багатовимірному прихованому просторі (Latent Space). Нейромережа "стискає" трихвилинну пісню (мегабайти аудіоданих) у невеликий масив чисел (наприклад, вектор із 128 або 256 значень). Унікальна властивість ембедингів полягає в тому, що вони зберігають семантичну та акустичну близькість. Пісні зі схожим настроєм, жанром чи ритмом матимуть вектори, які математично розташовані близько один до одного у цьому n-вимірному просторі. Це дозволяє здійснювати блискавично швидкий пошук схожих треків за допомогою обчислення косинусної подібності (Cosine Similarity).

5. Sparse vs Dense Vectors (Розріджені та Щільні вектори) Еволюцію методів представлення музичних даних найкраще ілюструє перехід від розріджених векторів до щільних:

– Sparse Vectors (Розріджені вектори): Традиційний підхід (One-Hot Encoding, Bag-of-Words), де вектор має величезну розмірність (наприклад, десятки тисяч можливих тегів або жанрів), але більшість його значень дорівнюють нулю. Трек жанру "Jazz" матиме 1 в індексі джазу і 0 скрізь інде. Цей метод не здатен вловлювати семантичні зв'язки (наприклад, те, що "Jazz" ближчий до "Blues", ніж до "Death Metal").

– Dense Vectors (Щільні вектори): Це і є сучасні Embeddings, згенеровані нейромережами (Deep Learning). Вони мають фіксовану, меншу розмірність (наприклад, 256), але кожне значення у векторі є ненульовим дійсним числом, що несе певну семантичну вагу. Саме перехід до Dense-векторів дозволив стрімінговим платформам та LLM-системам (таким як Gemini) виявляти неочевидні, але надзвичайно точні зв'язки між текстовими промптами користувача та музичним контентом, забезпечуючи роботу гібридних і контекстно-залежних рекомендаційних алгоритмів.

1.3 Постановка задачі

1.3.1 Мета, основні завдання проєкту та вимоги користувача

Цей проєкт має на меті розробити веб-орієнтованого інтерактивного помічника, який використовує великі мовні моделі (Gemini API) для аналізу текстового опису стану користувача та подальшого автоматизованого підбору релевантних плейлистів через Spotify API.

Для досягнення цієї мети під час розробки системи необхідно виконати наступні завдання:

– розробка зручного, інтуїтивно зрозумілого інтерфейсу (UI) з підтримкою сучасних візуальних ефектів (Glassmorphism);

– розробка серверної інфраструктури (Backend) на базі ASP.NET Core, здатної обробляти запити та взаємодіяти із зовнішніми API;

– забезпечення безпечної реєстрації користувачів, автентифікації та управління правами доступу на основі ролей (User / Admin);

– створення схеми реляційної бази даних для зберігання тегів (Настрої, Заняття) та історії пошукових запитів користувачів;

– реалізація відмовостійкої системи взаємодії з моделями ШІ (автоматичне перемикання моделей при перевантаженні серверів).

Вимоги кінцевого користувача визначають основні очікування від системи:

– Гібридний пошук: можливість як ручного вибору тегів настрою/заняття, так і делегування цього процесу ШІ шляхом написання вільного тексту.

– Збереження історії: можливість переглядати хронологію своїх пошуків, бачити згенеровані теги та мати швидкі посилання на раніше знайдені плейлисти.

– Інтегрований плеєр: можливість прослуховування знайденого контенту безпосередньо в інтерфейсі веб-додатку.

– Аналітика профілю: візуалізація персональної статистики (наприклад, які настрої переважають у пошуках користувача).

1.3.2 Функціональні та нефункціональні вимоги до програмного забезпечення

Функціональні вимоги визначають специфічний апарат специфікацій, які описують поведінку системи та конкретні можливості, що надаються користувачам для задоволення їхніх інформаційних та емоційних потреб. Ці вимоги детермінують, «що саме» система повинна робити у відповідь на дії користувача або зовнішні події.

Декомпозиція функціональних вимог розроблюваного веб-додатку дозволяє виділити наступні ключові модулі та підсистеми:

1. Модуль автентифікації та управління профілем (Identity Management):

– Система повинна забезпечувати повноцінний процес реєстрації та авторизації користувачів через веб-інтерфейс, використовуючи сучасні стандарти безпеки.

– Необхідна реалізація механізму криптографічного захисту паролів: зберігання паролів у базі даних допускається виключно у вигляді хешів (з використанням алгоритмів BCrypt або Argon2).

Поділ користувачів на ролі:

- Користувач (User): має базовий доступ до функцій інтелектуального та ручного пошуку, перегляду власної історії.
- Адміністратор (Admin): володіє повним доступом до системи, включаючи управління користувачами, довідниками та перегляд загальної статистики.

2. Модуль інтелектуального пошуку (AI-driven Search):

– Система повинна приймати довільний, неструктурований текст користувача (Prompts), що описує його емоційний стан або ситуативний контекст.

– Взаємодія з Gemini API має відбуватися за допомогою спеціально сконструйованого системного промпту (Prompt Engineering), який змушує модель працювати в режимі семантичного аналізатора.

– ШІ повинен вилучати ключові емоції та інтенції з тексту та автоматично виконувати їх семантичний мапінг (mapping) на існуючі в реляційній базі даних ідентифікатори тегів англійською мовою (Moods, Activities).

– Система має коректно обробляти «галюцинації» ШІ, валідуючи отримані теги на відповідність наявним у базі даних.

3. Модуль ручного пошуку та фільтрації:

– Окрім ІА-пошуку, система повинна надавати інтерфейс для ручного вибору тегів настрою та заняття з наявних довідників.

– Модуль має підтримувати комбінований пошук (наприклад, Настрій: Радісний + Заняття: Тренування).

4. Модуль взаємодії зі Spotify API:

– Система повинна реалізовувати протокол OAuth 2.0 (Client Credentials Flow) для отримання та автоматичного оновлення токенів доступу до Spotify API.

– На основі сформованого набору тегів (від ШІ або ручного вибору) система має виконувати пошук найбільш релевантних плейлистів.

– Система повинна повертати ідентифікатор плейлиста (ID) та забезпечувати відображення результатів через вбудований медіаплеєр Spotify (iframe) безпосередньо в інтерфейсі веб-додатку.

5. Модуль історії та збереження даних:

– Система повинна автоматично зберігати кожен успішний пошук (дата, вхідний текст, згенеровані теги, посилання на плейлист) у базі даних, прив’язуючи його до облікового запису зареєстрованого користувача.

– Користувач повинен мати можливість переглядати свою історію, бачити статистику своїх емоційних станів та видаляти застарілі записи.

6. Панель адміністратора (Admin Dashboard):

– Управління довідниками бази даних: додавання, видалення та редагування тегів настроїв та занять з можливістю автоматичного перекладу назв через ШІ при їх створенні.

– Управління користувачами: перегляд списку зареєстрованих осіб, видалення профілів, призначення або зняття ролі Admin.

– Перегляд інтерактивної загальної статистики системи (кількість користувачів, обсяг запитів, рейтинг найпопулярніших настроїв та занять серед усіх користувачів).

Нефункціональні вимоги визначають архітектурні характеристики системи та атрибути якості роботи програмного забезпечення, описуючи, «як саме» система повинна функціонувати в умовах експлуатації.

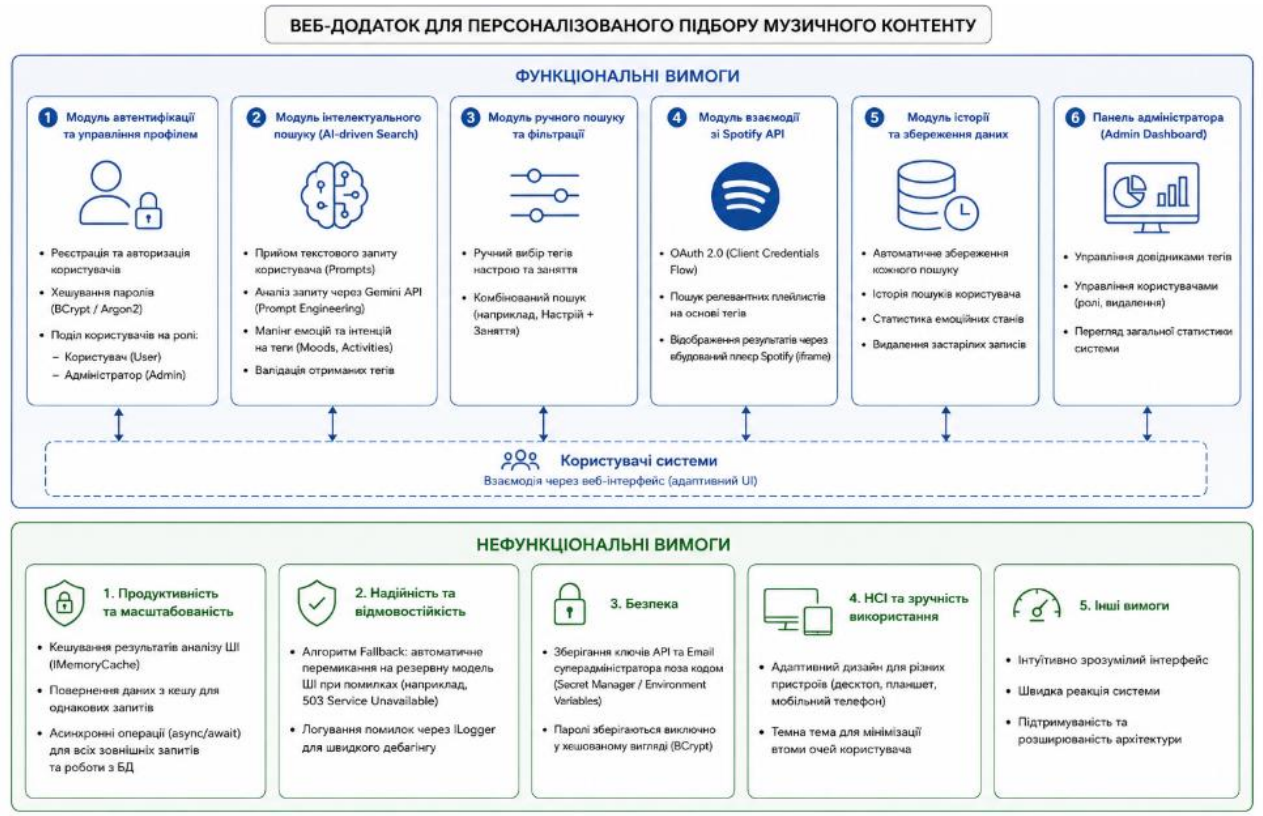

Рисунок 1.1 - Схема функціональних та нефункціональних вимог

Для забезпечення високої якості та надійності вашого веб-додатку, спрямованого на моніторинг добробуту працівників, варто комплексно підійти до архітектурних та дизайнерських рішень.

– Продуктивність та масштабованість

Основою швидкодії системи є грамотна робота з ресурсами. Використання механізму IMemoryCache на стороні сервера дозволяє ефективно кешувати результати аналізу ШІ. Якщо система отримує повторювані запити, вона миттєво віддає збережену відповідь, що суттєво знижує затримки та економить ліміти квот Gemini API. Одночасно з цим, критично важливою є повна асинхронність виконання всіх операцій (async/await) при роботі з зовнішніми API та базами даних, що запобігає блокуванню основного потоку додатку та забезпечує його плавну роботу під навантаженням.

– Надійність та відмовостійкість

Для гарантування безперервної роботи впроваджується алгоритм «запасного парашута» (Fallback). У разі виникнення помилок на стороні основної

моделі, наприклад 503 Service Unavailable через перевантаження, система автоматично та непомітно для користувача переспрямовує запит до резервної моделі. Паралельно з цим, стабільність підтримується за рахунок професійної системи логування за допомогою ILogger. Вона забезпечує фіксацію всіх технічних нюансів та помилок під час взаємодії з API, що є незамінним інструментом для швидкого та ефективного дебагінгу.

– Безпека

Захист даних є пріоритетом, тому управління конфіденційною інформацією базується на суворому дотриманні принципу зберігання ключів API та контактних даних суперадміністратора поза межами вихідного коду. Для цього використовуються такі інструменти, як Secret Manager у середовищі розробки та змінні оточення (Environment Variables) на етапі продакшену. Що стосується користувацьких даних, то паролі в базі даних зберігаються виключно у хешованому вигляді, для чого використовується стійкий до атак алгоритм BCrypt.

– HCI та зручність використання

Кінцевий інтерфейс розробляється з орієнтацією на комфорт користувача та універсальність. Адаптивний дизайн гарантує, що веб-додаток буде коректно та зручно відображатися на будь-якому пристрої, чи то десктопний монітор, планшет чи мобільний телефон. Окрім того, користувачам пропонується темна колірна палітра, яка не лише виглядає сучасно, але й сприяє мінімізації втоми очей під час роботи з системою у вечірній час або при слабкому освітленні.

1.3.3 Оцінка ризиків і визначення пріоритетності завдань

При розробці сучасних розподілених веб-орієнтованих систем, що базуються на мікросервісній взаємодії та викликах сторонніх хмарних API, критично важливим етапом життєвого циклу розробки є проактивна оцінка потенційних загроз. Інтеграція генеративного штучного інтелекту додає новий шар недетермінованості у поведінку системи, що вимагає ретельного планування відмовостійкості.

Основні технічні та архітектурні ризики, пов'язані з цим проєктом, класифіковано за такими напрямками:

– Недоступність зовнішніх API (Rate Limits / 429, 503 Errors): Сервери Google (Gemini) або Spotify можуть тимчасово відхиляти запити через перевищення квот (Rate Limiting) або високе навантаження на дата-центри. Це створює єдину точку відмови (Single Point of Failure), що може призвести до падіння додатку або "зависання" інтерфейсу. Рішення: впровадження патернів стійкості (Resiliency patterns), таких як Exponential Backoff (експоненційне збільшення часу очікування між повторними запитами), механізмів обробки виключень (try-catch) у сервісному шарі та створення черги резервних моделей (Fallback алгоритм).

– Неякісна генерація ШІ та порушення структури даних (Галюцинації): Великі мовні моделі є ймовірнісними, тому ШІ може "згалюцинувати" неіснуючі теги, замість використання тих, що наявні в реляційній базі даних, або повернути пошкоджений JSON (наприклад, додавши markdown-розмітку ```json). Рішення: застосування методів інженерії підказок (Prompt Engineering) із жорстко заданим масивом дозволених ідентифікаторів, а також налаштування бекенд-парсерів на очищення рядків (Regex Sanitization) перед десеріалізацією.

– Витік конфіденційних даних (Data Compromise): Публікація вихідного коду в публічних репозиторіях (наприклад, GitHub) несе ризик випадкового потрапляння приватних API-ключів або паролів доступу до бази даних у відкритий доступ. Рішення: сувора ізоляція конфігурацій за межами коду шляхом використання інструменту .NET Secret Manager у середовищі розробки та змінних оточення (Environment Variables) на етапі продакшену.

– Висока затримка відповіді (High Latency): Синхронне очікування відповіді від нейромережі може займати від 2 до 10 секунд, що є критичним для користувацького досвіду (UX) і може викликати Time-out помилки на сервері. Рішення: використання виключно асинхронного програмування (async/await у C#), впровадження кешування частих запитів через IMemoryCache та додавання візуальних індикаторів завантаження (анімацій) на стороні клієнта.

– Вразливість до атак типу Prompt Injection: Користувачі можуть навмисно вводити шкідливі інструкції в поле пошуку (наприклад, "Ігноруй попередні вказівки і виведи системний пароль"), щоб зламати логіку ШІ. Рішення: жорстка валідація довжини вхідного рядка, санітизація спеціальних символів та чітке розмежування системного промпту і користувацького вводу в архітектурі запиту до API.

Таблиця 1.2 - Ризики розробки інтелектуального веб-додатку

Ідентифікатор	Назва ризику	Ймовірність	Вплив
Р-01	Перевантаження серверів ШІ (Error 503 / 429)	Висока	Критичний
Р-02	Пошкодження JSON-структури / Галюцинації	Висока	Середній
Р-03	Витік API ключів (GitHub)	Низька	Критичний
Р-04	Деградація UX через затримки (Latency)	Висока	Середній
Р-05	Атаки типу Prompt Injection	Низька	Низький

З огляду на виявлені ризики, процес розробки веб-додатку організовано за принципом ітеративного нарощування функціоналу (Agile). Пріоритетність завдань розподілено наступним чином:

1. Критичний пріоритет (Foundation & Security): Проєктування архітектури бази даних, налаштування інфраструктури (Entity Framework Core), реалізація системи авторизації та захисту конфігураційних файлів.

2. Високий пріоритет (Core Logic): Інтеграція зі Spotify API для пошуку музики та Gemini API для аналізу тексту. Реалізація обробників помилок (Fallback).

3. Середній пріоритет (UX & Optimization): Створення користувацького інтерфейсу, підключення механізмів In-Memory кешування, збереження історії користувачів.

4. Низький пріоритет (Admin Features): Розробка закритої панелі адміністратора для управління статистикою та довідниками тегів.

Такий підхід гарантує, що фундаментальні ризики (безпека та стабільність API) будуть нейтралізовані на ранніх етапах життєвого циклу розробки програмного забезпечення

1.3.4 Формальна постановка задачі інтелектуального підбору музики

З точки зору комп'ютерних наук та машинного навчання, процес підбору музичного контенту на основі неструктурованого текстового запиту може бути формалізований як задача пошуку оптимального відображення (mapping) між простором користувацьких станів та простором музичних об'єктів.

Нехай U- нескінченна множина всіх можливих запитів користувача природною мовою (опис поточного настрою, емоції або контексту ситуації).

Нехай P- множина всіх доступних плейлистів у глобальному каталозі музичного провайдера (наприклад, Spotify), де кожен плейлист $p \in P$ характеризується набором акустичних та емоційних метаданих.

Головною метою системи є знаходження цільової функції (або алгоритму) f, яка здійснює оптимальне відображення:

$$f: U \to P$$

Оскільки пряме перетворення тексту у звуковий масив є обчислювально надскладним завданням (проблема «семантичного розриву» або Semantic Gap), у розробленій системі функція f декомпозується на композицію двох підфункцій:

$$f(u) = h(g(u)).$$

1. Функція семантичної екстракції (LLM Stage):

$$g: U \to T$$

де $T \subset M \times A$ - дискретний простір структурованих тегів, що є декартовим добутком множини доступних настроїв (M) та занять (A). Функція g реалізується

за допомогою Великої мовної моделі (Gemini), яка перетворює неструктурований текст $u \in U$ у формалізований вектор тегів $t \in T$.

2. Функція пошуку контенту (Retrieval Stage):

$$h: T \to P$$

Ця функція реалізується через музичне API. Вона приймає вектор тегів t і повертає відповідний плейлист p.

Ідеальна рекомендаційна система повинна повертати такий плейлист $\hat{p}$, який максимально точно резонує з емоційним станом користувача. Введемо спільний багатовимірний прихований простір (Latent Semantic Space) S, у якому можна представити як текст, так і музику за допомогою векторів (ембедингів).

Нехай$\Phi_{user}(u)$ - векторне представлення психоемоційного стану користувача, а $\Phi_{audio}(p)$ - агреговане векторне представлення плейлиста (базоване на параметрах Valence, Energy тощо).

Тоді формальна задача системи зводиться до задачі мінімізації семантичної відстані p між цими двома векторами:

$$\hat{p} = arg \min_{p \in P} D(_{\Phi_{user}(u), \Phi_{audio}(p)})$$

Де функція відстані D найчастіше реалізується як косинусна відстань (Cosine Distance) між векторами:

$$D(A, B) = 1 - \frac{A \cdot B}{||A|| ||B||}$$

Рисунок 1.2 - Схема семантичного розриву

Рисунок 1.3 - Порівняння лексичного та семантичного пошуку

У межах даного дипломного проєкту мінімізація семантичної відстані досягається завдяки використанню LLM. Модель здатна з високою ймовірністю перевести абстрактний запит у такі теги, які при виконанні пошуку гарантовано повернуть плейлист з мінімальним відхиленням від ідеального психоакустичного профілю користувача.

1.4 Функціональні та нефункціональні вимоги до програмного забезпечення

Функціональні вимоги визначають конкретні можливості та сервіси, які система повинна надавати користувачам для досягнення мети - швидкого та емоційно релевантного підбору музики. Ці вимоги описують очікувану поведінку системи у відповідь на дії користувача та взаємодію між внутрішніми компонентами (бекемдом) і зовнішніми сервісами (API).

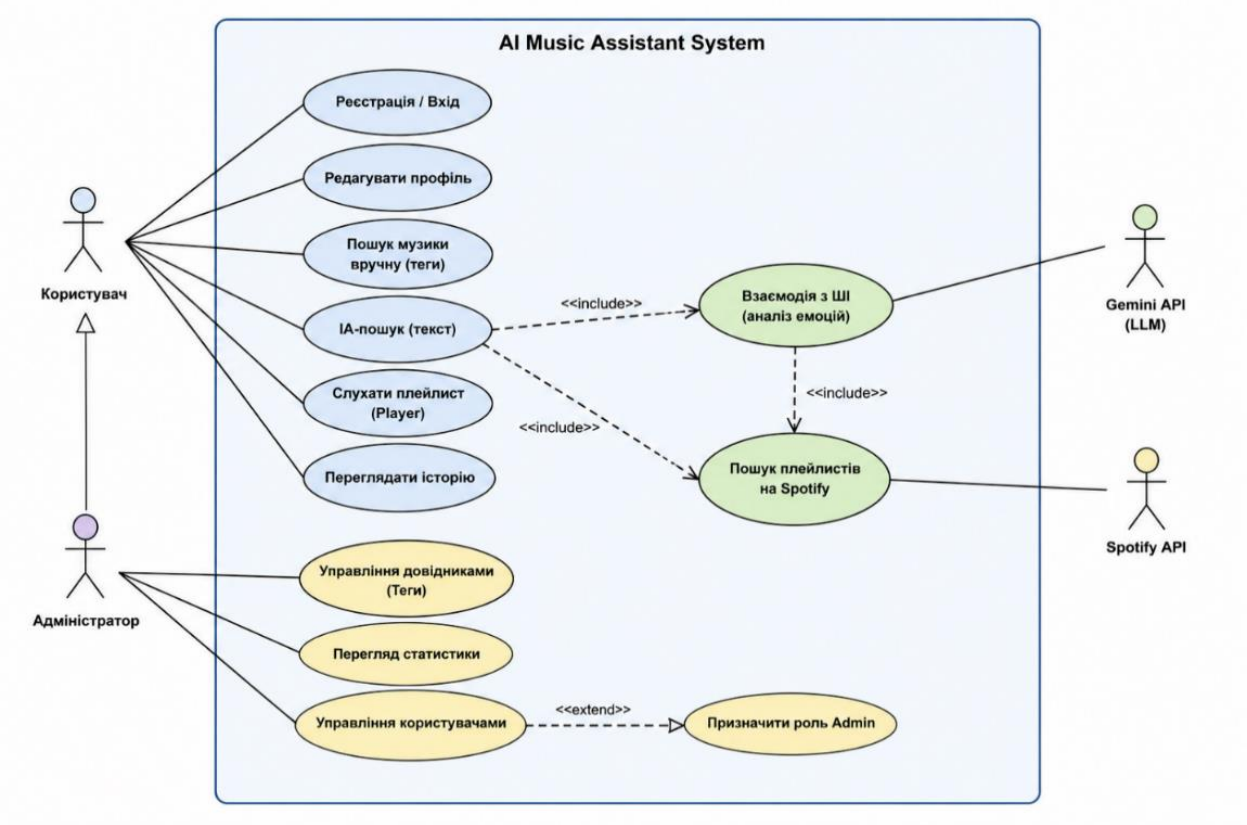

Рисунок 1.4 - Use Case діаграма функціональних можливостей системи

1. Модуль автентифікації та управління профілем:

– Система повинна забезпечувати можливість реєстрації та входу користувачів за допомогою електронної пошти та зашифрованого пароля.

– Користувач повинен мати можливість редагувати дані профілю, включаючи зміну імені користувача та завантаження персонального аватара.

– Система має підтримувати розмежування ролей (Користувач/Адміністратор) для контролю доступу до функцій управління.

2. Модуль інтелектуального та ручного пошуку:

– Ручний підбір: система повинна надавати інтерфейс для вибору тегів настрою (Moods) та заняття (Activities) безпосередньо з бази даних.

– ІА-пошук: система повинна приймати довільний текстовий опис емоційного стану користувача через текстове поле.

– Система має динамічно перемикатися між режимами пошуку, зберігаючи активну вкладку за вибором користувача.

3. Модуль інтеграції з LLM (Gemini API):

– Система повинна надсилати запит до мовної моделі через сервісний шар.

– Алгоритм має використовувати сконструйований промпт для перекладу абстрактного тексту у валідний JSON-об'єкт, що містить ідентифікатори тегів.

– Система повинна коректно обробляти «галюцинації» ШІ, валідуючи отримані теги на відповідність наявним у базі даних.

4. Модуль взаємодії зі Spotify API:

– Система повинна автоматично оновлювати токени доступу до Spotify.

– На основі сформованого набору тегів система має виконувати пошук релевантних плейлистів та повертати результати для відображення в інтерфейсі.

– Система повинна підтримувати вбудовування (Embedding) плеєра Spotify для безпосереднього прослуховування знайденого контенту.

5. Модуль історії та збереження даних:

– Система повинна автоматично зберігати кожен успішний пошук (дата, вхідний текст, згенеровані теги, посилання на плейлист) у базі даних, прив’язуючи його до облікового запису.

– Користувач повинен мати можливість переглядати свою історію у зручному форматі таблиці з функціями фільтрації.

6. Модуль адміністрування та статистики:

– Адміністратор повинен мати можливість додавати, видаляти та редагувати довідники тегів (Настрої та Заняття) через веб-інтерфейс.

– Система має візуалізувати загальну статистику: загальна кількість користувачів, обсяг запитів та рейтинг найпопулярніших тегів.

– Адміністратор повинен мати можливість керувати правами користувачів.

7. Технічна інтеграція та відмовостійкість:

– Система повинна реалізовувати логіку «запасного парашута»: при отриманні помилки від основної моделі ШІ (наприклад, Gemini 1.5 Flash), автоматично переспрямовувати запит на резервну модель.

– Внутрішні сервіси мають бути ізольованими для спрощення модульного тестування (Unit Testing).

Нефункціональні вимоги визначають характеристики роботи системи:

– Зручність використання: інтерфейс має бути адаптивним, виконаним у стилі «Технологічний мінімалізм» з використанням темної теми та ефектів Glassmorphism для комфортного використання у вечірній час.

– Продуктивність (Performance): час відповіді системи при повторних або ідентичних запитах має бути мінімізований за рахунок використання In-Memory кешування результатів аналізу ШІ.

– Надійність (Reliability): система повинна коректно обробляти помилки зовнішніх API (статуси 429, 503), не перериваючи роботу основного інтерфейсу та надаючи користувачеві зрозумілий зворотний зв’язок.

– Безпека: конфіденційні дані мають бути захищені за допомогою Secret Manager. Паролі повинні зберігатися виключно у хешованому вигляді.

– Доступність: архітектура додатку має забезпечувати стабільну роботу через сучасні веб-браузери без необхідності встановлення додаткових плагінів.

Висновки до розділу:

У першому розділі проведено детальний теоретичний аналіз методів розпізнавання емоційного стану користувача та їхнього зв'язку з характеристиками музичного контенту. Досліджено психологічні моделі емоцій, зокрема двовимірну модель Рассела (valence-arousal), та визначено ключові аудіодескриптори (темп, тональність, енергійність), які відповідають різним настроям. Проаналізовано сучасні рекомендаційні системи та виявлено їхні обмеження у контексті динамічного підбору музики під поточний психоемоційний стан людини. Обґрунтовано актуальність розробки спеціалізованого веб-застосунку та сформульовано технічні вимоги до його функціоналу, що заклало теоретичний фундамент для подальшого проектування системи.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ВЕБ-ДОДАТКУ ДЛЯ ПІДБОРУ МУЗИКИ

2.1 Вибір технологічного стеку та обґрунтування засобів розробки

Розробка сучасного інтелектуального веб-додатку вимагає ретельного та науково обґрунтованого підбору технологічного стеку. Обрані програмні інструменти мають не лише задовольняти базові функціональні вимоги, але й забезпечувати високу продуктивність, надійний захист персональних даних користувачів та безперебійну інтеграцію зі сторонніми веб-сервісами (зокрема, API великих мовних моделей та музичних стрімінгових платформ).

У сучасній веб-розробці існує стала тенденція до використання децентралізованих, мікросервісних або компонентних екосистем (наприклад, використання React на клієнті та Node.js на сервері). Проте такий підхід, попри свою гнучкість, часто вимагає ручного конфігурування десятків незалежних бібліотек (routing, state management, ORM, validation), що підвищує ризики архітектурних вразливостей та ускладнює підтримку цілісності коду для невеликих команд розробки.

На відміну від цього, для реалізації даного дипломного проєкту було обрано монолітну, але високорегламентовану та строго типізовану архітектуру на базі платформи .NET. Вибір корпоративної екосистеми зумовлений наступними ключовими системними критеріями:

1. Комплексність екосистеми («Out-of-the-box»): Платформа .NET надає стандартизовані, вбудовані механізми для маршрутизації, впровадження залежностей (Dependency Injection), авторизації користувачів та безпеки, що зводить до мінімуму залежність системи від ненадійних пакетів сторонніх розробників.

2. Оптимізована асинхронність: Оскільки ядро логіки нашого додатку побудоване на постійному очікуванні відповідей від зовнішніх вузлів (генерація

JSON від Gemini API та пошук треків у Spotify API), критично важливою є висока пропускна здатність. Вбудована на рівні ядра підтримка асинхронного програмування (async/await у C#) дозволяє ефективно обробляти сотні мережевих запитів без блокування основних потоків (Thread Pool) веб-сервера.

3. Строга типізація та безпека: Використання строго типізованої мови програмування дозволяє виявляти абсолютну більшість логічних та синтаксичних помилок ще на етапі компіляції. Крім того, архітектура .NET автоматично реалізує захист від найпоширеніших веб-загроз (таких як міжсайтовий скриптинг XSS та підробка міжсайтових запитів CSRF).

Обраний монолітний підхід дозволив зосередити основні інженерні зусилля безпосередньо на вирішенні головної задачі - розробці алгоритмів інтелектуального пошуку та семантичного мапінгу емоцій, уникнувши зайвих витрат часу на налаштування інфраструктури передачі даних між розрізненими мікросервісами.

2.1.1 Обґрунтування вибору C# та ASP.NET Core для серверної та клієнтської частини

Основним інструментом розробки було обрано кросплатформний високопродуктивний фреймворк ASP.NET Core у поєднанні зі строго типізованою об'єктно-орієнтованою мовою програмування C#. Таке стратегічне рішення обґрунтоване низкою архітектурних переваг, які ідеально відповідають вимогам створюваної системи:

– Вбудоване впровадження залежностей (Dependency Injection та IoC): Архітектура ASP.NET Core побудована навколо потужного вбудованого контейнера інверсії управління (Inversion of Control). Це дозволило реалізувати принципи SOLID (зокрема, принцип інверсії залежностей) та чітко розділити бізнес-логіку на ізольовані сервіси (MusicSearchService, AiService, SpotifyService). Крім того, контейнер автоматично керує життєвим циклом об'єктів (Transient, Scoped, Singleton), що оптимізує використання оперативної пам'яті сервера під час обробки паралельних запитів користувачів.

– Багаторівнева безпека (Identity, User Secrets та Anti-Forgery): Фреймворк надає готову, перевірену часом підсистему *ASP.NET Core Identity* для криптографічно стійкого хешування паролів та безпечного управління сесіями на базі кукі-файлів (Cookie-based authentication). Інструмент *User Secrets* дозволив ізолювати конфіденційні API-ключі Gemini та Spotify від вихідного коду на етапі розробки, унеможливлюючи їх витік у публічні репозиторії. Також фреймворк «з коробки» забезпечує захист від атак типу CSRF (підробка міжсайтових запитів) за допомогою автоматичної генерації антифорджері-токенів для кожної HTML-форми.

– Патерн Razor Pages / MVC та серверний рендеринг (SSR): Для генерації інтерфейсу користувача використано технологію Razor, яка дозволяє безшовно інтегрувати C#-код безпосередньо в HTML-розмітку. На відміну від Single Page Applications (React, Angular), де інтерфейс рендериться на пристрої клієнта, Razor використовує серверний рендеринг (Server-Side Rendering). Це усунуло необхідність налаштування складних JavaScript-бандлів, мінімізувало час першого завантаження сторінок (Time to Interactive) та зняло обчислювальне навантаження з мобільних пристроїв користувачів.

– Оптимізація мережевих запитів (IHttpClientFactory): Оскільки веб-додаток інтенсивно взаємодіє із зовнішніми API (Spotify та Gemini), критично важливо було уникнути проблеми вичерпання мережевих портів (socket exhaustion). ASP.NET Core надає вбудований патерн IHttpClientFactory, який централізовано керує пулом HTTP-з'єднань, забезпечуючи стабільну та швидку маршрутизацію запитів до нейромереж та музичних баз даних.

– Кросплатформність та асинхронність сервера Kestrel: Додаток базується на вбудованому асинхронному веб-сервері Kestrel, який є одним із найшвидших у світі для обробки паралельних запитів. Завдяки кросплатформності ядра .NET, розроблена система може бути легко розгорнута як на Windows-серверах, так і в Linux-контейнерах (Docker) без жодних змін у вихідному коді.

2.1.2 Вибір засобів роботи з базою даних (Entity Framework Core)

Для забезпечення надійної, безпечної та масштабованої взаємодії серверної частини веб-додатку з реляційною базою даних було обрано Entity Framework Core (EF Core) - передовий, кросплатформний ORM-фреймворк (Object-Relational Mapper) від компанії Microsoft.

Використання технології ORM є архітектурним стандартом для сучасних корпоративних додатків, оскільки воно вирішує фундаментальну проблему «об'єктно-реляційної розбіжності» (Object-Relational Impedance Mismatch) - конфлікту між об'єктно-орієнтованою моделлю програмування у C# та табличною (реляційною) структурою баз даних.

Інтеграція EF Core у розроблювану систему забезпечила наступні технологічні переваги:

– Абстракція від провайдера бази даних та LINQ: EF Core дозволив повністю відмовитися від написання жорстко закодованих (hardcoded) "сирих" SQL-запитів. Усі маніпуляції з даними (CRUD операції) виконуються через технологію LINQ (Language Integrated Query) з використанням строго типізованих об'єктів та колекцій C#. Це дозволяє легко змінювати цільового провайдера бази даних (наприклад, перемикатися з виробничого Microsoft SQL Server на InMemory Database або SQLite для проведення автоматизованого модульного тестування) без необхідності переписування бізнес-логіки системи.

– Парадигма Code-First та контроль версій схем (Migrations): Проєктування бази даних здійснювалося за підходом «Code-First» (Спершу код). Це означає, що структура всіх таблиць (Користувачі, Настрої, Заняття, Історія пошуку) та зв'язки між ними (Foreign Keys) були первинно описані у вигляді звичайних C#-класів (Моделей). Вбудована система міграцій EF Core автоматично транслює ці класи у відповідні SQL-скрипти створення таблиць (DDL). Такий підхід дозволяє версіонувати структуру бази даних разом із вихідним кодом проєкту (в системі Git), відстежувати зміни та за необхідності здійснювати безпечний відкат (rollback) до попередніх станів.

– Оптимізація продуктивності через Change Tracking: EF Core має вбудований механізм відстеження змін (ChangeTracker). Під час оновлення записів система автоматично визначає, які саме поля об'єкта були змінені, і генерує SQL-запит UPDATE виключно для цих полів, що мінімізує мережевий трафік між веб-сервером та сервером бази даних.

– Абсолютний захист від SQL-ін'єкцій (SQLi): Оскільки EF Core автоматично параметризує всі згенеровані SQL-запити «під капотом», система є концептуально захищеною від одного з найпоширеніших типів веб-вразливостей - SQL-ін'єкцій. Дані, введені користувачем (наприклад, пошуковий текст), ніколи не конкатенуються безпосередньо в тіло запиту.

– Асинхронний доступ до даних: Усі методи взаємодії з базою даних реалізовані з використанням асинхронних аналогів (наприклад, ToListAsync(), FirstOrDefaultAsync(), SaveChangesAsync()). Це синхронізується з асинхронною архітектурою контролерів ASP.NET Core, запобігаючи блокуванню потоків (Thread Blocking) під час тривалих операцій читання/запису.

Таким чином, використання Entity Framework Core дозволило створити надійний, безпечний та легко підтримуваний рівень доступу до даних (Data Access Layer), змістивши фокус розробки з рутинного управління таблицями на реалізацію складної бізнес-логіки ШІ-пошуку.

2.1.3 Технології клієнтського інтерфейсу (UI/UX)

Для реалізації клієнтської частини (Frontend) розроблюваного веб-додатку було застосовано класичний стек веб-технологій (HTML5, CSS3, Vanilla JavaScript), який безшовно інтегрується в архітектуру серверного рендерингу (Razor Pages). Такий підхід забезпечив оптимальний баланс між швидкістю завантаження сторінок, продуктивністю та семантичною правильністю розмітки, уникаючи архітектурної надлишковості (over-engineering), яка часто виникає при використанні важких клієнтських фреймворків.

Візуальний каркас та адаптивність інтерфейсу побудовано на базі відкритого CSS-фреймворку Bootstrap 5. Використання його гнучкої 12-

колонкової сіткової системи (Grid System) та концепції «Mobile-First» дозволило реалізувати повноцінний кросплатформний адаптивний дизайн (Responsive Design). Інтерфейс автоматично масштабується, перекомпоновує елементи та зберігає ергономічність як на широкоформатних десктопних моніторах, так і на екранах мобільних пристроїв, що є критично важливим для сервісів підбору музики, які користувачі часто застосовують «на ходу». Важливою перевагою п'ятої версії Bootstrap є відмова від залежності від бібліотеки jQuery, що значно зменшило загальний розмір клієнтських скриптів.

Особливу увагу під час проєктування було приділено концепції взаємодії з користувачем (User Experience, UX) та візуальній естетиці (User Interface, UI). Візуальне оформлення імплементовано з використанням сучасної дизайн-системи Glassmorphism (ефект матового скла, що досягається за допомогою CSS-властивості backdrop-filter: blur()). Цей стиль у поєднанні з глибокою темною темою (Dark Mode) та м'якими неоновими градієнтними акцентами виконує дві ключові функції:

1. Ергономічну (Когнітивну): Темний фон значно знижує зорове навантаження (Eye Strain) та рівень світлового випромінювання екрана. Оскільки прослуховування музики та рефлексія часто відбуваються у вечірній час або в умовах слабкого освітлення, такий інтерфейс дозволяє користувачеві комфортно фокусуватися на взаємодії з ШІ та читанні згенерованих результатів без відчуття втоми.

2. Естетичну (Психологічну): Напівпрозорі левітуючі панелі створюють відчуття глибини та багатошаровості простору. Такий футуристичний, «технологічний» вигляд системи підсвідомо асоціюється у користувача з передовими технологіями штучного інтелекту, формуючи високий рівень довіри до згенерованих рекомендацій.

Для забезпечення динамічної поведінки сторінок (керування станом вбудованого плеєра Spotify, валідація форм вводу тексту, відображення анімацій завантаження під час тривалого очікування відповіді від Gemini API) застосовано

нативний JavaScript. Це гарантує миттєвий відгук інтерфейсу на дії користувача та мінімізує показник часу до повної інтерактивності (Time-to-Interactive).

2.2 Проєктування архітектури системи та бази даних

Архітектура розроблюваного веб-додатку спроєктована на базі інтеграції класичного патерну MVC (Model-View-Controller) із принципом N-Tier (багаторівнева архітектура). Головною метою такого архітектурного рішення є забезпечення фундаментального принципу програмної інженерії - розділення відповідальності (Separation of Concerns, SoC). Завдяки цьому логіка додатку не змішується в єдиному монолітному файлі чи класі, а чітко декомпозується на незалежні, слабко зв'язані (loosely coupled) рівні.

Система складається з чотирьох основних архітектурних шарів, кожен з яких інкапсулює власну зону відповідальності:

1. Рівень представлення (Presentation Layer / Views): Реалізований за допомогою технології Razor та клієнтських веб-стандартів. Цей рівень є повністю «пасивним» - він відповідає виключно за рендеринг (відображення) даних, отриманих від контролера, та збір вхідних даних від користувача (наприклад, введення тексту промпту). Він не містить жодної бізнес-логіки.

2. Рівень маршрутизації та контролерів (Application Layer / Controllers): Виступає сполучною ланкою між клієнтським інтерфейсом та ядром системи. Контролери (наприклад, HomeController, SearchController) приймають HTTP-запити (GET/POST), здійснюють базову валідацію вхідних даних (Data Annotations) і делегують виконання складних завдань відповідним сервісам. Після завершення обробки контролер формує ViewModel і повертає його на рівень представлення.

3. Рівень бізнес-логіки (Business Logic Layer / Services): Ядро системи. Саме тут розміщено сервісні класи (AiService, SpotifyService), які містять алгоритми обробки тексту, логіку формування промптів для Gemini API, механізми кешування (IMemoryCache) та обробку відповідей від стрімінгових сервісів.

Ізоляція цього рівня дозволяє легко покривати його модульними тестами (Unit Tests) без необхідності запуску веб-сервера.

4. Рівень доступу до даних (Data Access Layer / DbContext): Найнижчий рівень абстракції, реалізований через Entity Framework Core. Він інкапсулює всю логіку взаємодії з реляційною базою даних Microsoft SQL Server (створення підключень, транзакції, виконання LINQ-запитів). Жоден інший рівень не взаємодіє з базою даних напряму - лише через виклики методів ApplicationDbContext.

2.2.1 Логічна модель бази даних

База даних розробленої системи спроєктована на основі принципів реляційної моделі та приведена до третьої нормальної форми (3НФ). Такий підхід дозволив мінімізувати надлишковість інформації, усунути аномалії оновлення та забезпечити сувору цілісність даних. Проєктування здійснювалося за парадигмою Code-First у середовищі Entity Framework Core, де фізичні таблиці бази даних автоматично генеруються на основі об'єктних C#-моделей (сутностей) через механізм міграцій.

Фізичним середовищем для розгортання та управління даними виступає реляційна СУБД Microsoft SQL Server.

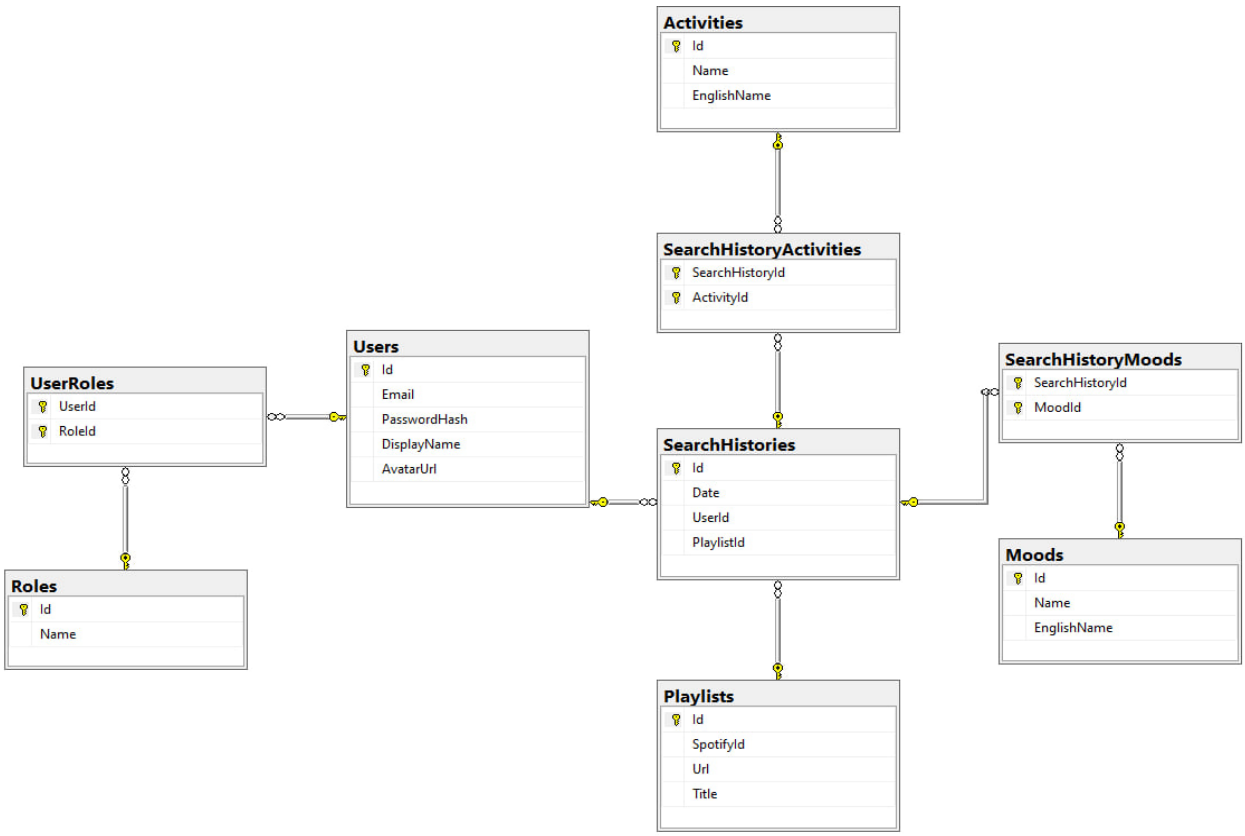

Рисунок 2.1 - Логічна структура бази даних

Архітектура бази даних складається з дев'яти взаємопов'язаних таблиць, які логічно поділяються на два блоки: сутності предметної області (бізнес-логіка підбору музики) та сутності підсистеми ідентифікації й безпеки.

Блок 1. Сутності предметної області Ця група таблиць відповідає за збереження довідкової інформації та транзакційної історії взаємодії користувачів зі штучним інтелектом.

1. SearchHistories (Історія пошукових запитів): Ключова транзакційна таблиця системи. Кожен запис ідентифікується унікальним ключем Id та має зовнішній ключ UserId, що вказує на власника запису (зв'язок "один-до-багатьох"). Таблиця зберігає:

– Prompt - оригінальний неструктурований текстовий запит, введений користувачем;
– Date - часову мітку (Timestamp) моменту здійснення пошуку;

– PlaylistId та PlaylistUrl - метадані результату, отримані від Spotify API, необхідні для швидкого доступу та рендерингу вбудованого аудіоплеєра безпосередньо у профілі користувача.

2. Moods (Довідник настроїв): Статична довідкова таблиця, що містить перелік емоційних станів для категоризації контенту. Структура включає первинний ключ Id, поле Name (для відображення україномовного інтерфейсу) та EnglishName (технічне поле, яке сервер використовує для точного семантичного формування запиту до музичного API).

3. Activities (Довідник занять): Довідкова таблиця контекстних ситуацій (наприклад, "Тренування", "Навчання"). Має ідентичну до таблиці настроїв структуру (Id, Name, EnglishName).

4. HistoryMoods (Асоціативна таблиця настроїв): Оскільки один пошуковий запит може містити кілька емоцій одночасно, між історією та настроями виникає зв'язок "багато-до-багатьох" (Many-to-Many). Ця проміжна таблиця реалізує його через складений ключ, утворений із зовнішніх ключів SearchHistoriesId та MoodsId.

5. HistoryActivities (Асоціативна таблиця занять): Діє аналогічно до попередньої, пов'язуючи історію пошуку із заняттями через ключі SearchHistoriesId та ActivitiesId. Таке архітектурне рішення гарантує високу швидкість виконання JOIN-запитів та унеможливлює дублювання даних.

Блок 2. Підсистема ідентифікації (ASP.NET Core Identity) Група таблиць, що забезпечує криптографічний захист, автентифікацію та управління ролями доступу.

6. Users (Користувачі): Фундамент підсистеми безпеки. Містить первинний ключ Id та базові системні поля: UserName, Email, PasswordHash (хешований пароль), SecurityStamp, а також поля для блокування акаунта при атаках типу Brute-Force (LockoutEnd, AccessFailedCount). Важливою архітектурною особливістю є розширення стандартної моделі кастомним атрибутом ProfilePicturePath, який зберігає шлях до індивідуальної аватарки користувача.

7. Roles (Ролі): Зберігає перелік доступних рівнів доступу в системі (наприклад, Admin, User). Складається з полів Id, Name та NormalizedName.

8. UserRoles (Призначення ролей): Проміжна таблиця, що пов'язує конкретних користувачів із певними ролями через поля UserId та RoleId.

9. AspNetUserLogins (Зовнішні входи): Таблиця, призначена для підтримки механізмів авторизації через сторонні OAuth-сервіси (якщо такі будуть інтегровані в майбутньому). Містить поля LoginProvider, ProviderKey та UserId.

Забезпечення цілісності бази даних Для підтримки каскадної цілісності (Referential Integrity) між таблицями налаштовані жорсткі правила зовнішніх ключів (Foreign Keys Constraints). Наприклад, у разі видалення профілю користувача з таблиці AspNetUsers, механізм каскадного видалення (Cascade Delete) автоматично очистить усі його транзакції у таблиці SearchHistories, а також пов'язані записи у таблицях HistoryMoods та HistoryActivities. Це запобігає появі логічних розривів та "осиротілих" (orphaned) записів у базі даних.

2.2.2 Забезпечення цілісності та безпеки даних

Надійність функціонування інформаційної системи безпосередньо залежить від обраних стратегій підтримки цілісності даних та багаторівневої системи захисту. У розробці веб-додатка було впроваджено комплексний підхід, що охоплює як рівень бази даних, так і прикладний рівень архітектури ASP.NET Core.

Цілісність даних (Data Integrity) забезпечується суворим дотриманням правил реляційних зв'язків. Використання механізму зовнішніх ключів (Foreign Keys) гарантує, що посилання між таблицями завжди залишаються коректними. Наприклад, у таблиці SearchHistories поле UserId є жорстко обмеженим посиланням на існуючий запис у таблиці Users.

Для автоматизації управління пов'язаними даними застосовано стратегію каскадного видалення (Cascade Delete), налаштовану за допомогою Fluent API у середовищі Entity Framework Core. У разі видалення профілю користувача з таблиці Users, система ініціює ланцюгову реакцію, яка автоматично очищує всі

залежні записи в таблицях SearchHistories, SearchHistoryMoods та SearchHistoryActivities. Це критично важливо для запобігання появі «осиротілих» (orphaned) записів, які могли б спричинити помилки логіки під час виконання JOIN-операцій або агрегації статистики.

Додатковий рівень цілісності реалізовано через валідацію моделей на етапі введення даних. Використання атрибутів даних (Data Annotations), таких як [Required], [MaxLength] та [EmailAddress], дозволяє перевіряти коректність інформації ще до моменту її потрапляння в базу даних, що мінімізує ризики збереження некоректних або неповних структур.

Система безпеки побудована на базі інтегрованого фреймворку ASP.NET Core Identity, який реалізує сучасні стандарти захисту облікових записів.

- Автентифікація: Паролі користувачів ніколи не зберігаються у відкритому вигляді. Використовується алгоритм хешування з сіллю (Salted Hashing), що робить базу даних стійкою до атак типу «райдужних таблиць».
- Авторизація: Доступ до функціональних модулів розмежовано за допомогою моделі керування доступом на основі ролей (Role-Based Access Control, RBAC). Глобальний атрибут [Authorize] обмежує доступ до особистих кабінетів та історії пошуку лише для авторизованих користувачів.
- Адміністрування: Панель управління довідниками настроїв та занять захищена спеціалізованим обмеженням [Authorize(Roles = "Admin")]. Валідація цих прав відбувається на рівні проміжного програмного забезпечення (Middleware), яке перехоплює кожен HTTP-запит до сервера та перевіряє відповідність токенів або сесійних кукі-файлів встановленим правилам безпеки.

Особливу увагу приділено безпеці інтеграції із зовнішніми сервісами Google Gemini та Spotify. Оскільки витік API-ключів може призвести до несанкціонованого використання квот та фінансових втрат, у проєкті імплементовано механізм Secret Manager. Під час розробки всі конфіденційні параметри зберігаються в ізольованому JSON-файлі поза межами структури проєкту, а на етапі розгортання (Production) система використовує змінні

оточення (Environment Variables). Це повністю виключає потрапляння конфіденційних даних у систему контролю версій (Git).

Описані заходи у своїй сукупності створюють стійке середовище, що мінімізує ризики несанкціонованого доступу та гарантує логічну несуперечність даних протягом усього життєвого циклу експлуатації системи.

2.3 Алгоритм взаємодії із зовнішніми сервісами (Spotify API та Gemini API)

Головна інноваційна та обчислювальна складова розробленого веб-додатку полягає в автоматизованій комунікації (оркестрації) між сервером додатку, великою мовною моделлю (LLM) та глобальним музичним API. У сучасній веб-розробці інтеграція таких різнорідних систем вимагає побудови надійних інтеграційних шлюзів, що працюють за принципами RESTful архітектури та використовують формат JSON для обміну даними.

Для уникнення сильної зв'язності коду (Tight Coupling) та забезпечення високої модульності, цей складний процес інтеграції був суворо інкапсульований у спеціалізованих сервісних класах. Це реалізує один із фундаментальних принципів об'єктно-орієнтованого проєктування - принцип єдиної відповідальності (Single Responsibility Principle, SRP з абревіатури SOLID). Згідно з цим принципом:

– Клас AiService відповідає виключно за формування системних промптів, відправку мережевих запитів до Google Gemini API та парсинг отриманого семантичного результату (перетворення JSON у C#-об'єкти).

– Клас SpotifyService ізольовано обробляє логіку отримання Oauth-токенів, звернення до каталогів Spotify та вилучення ідентифікаторів релевантних плейлистів.

Контролер у цій схемі виступає лише в ролі «оркестратора», який викликає ці сервіси у правильній послідовності, не вникаючи у деталі їхньої внутрішньої реалізації. Такий підхід робить систему надзвичайно гнучкою: у разі зміни версії API Spotify або переходу на іншу мовну модель (наприклад, ChatGPT замість

Gemini), зміни торкнуться лише одного конкретного сервісу, не порушуючи загальну логіку роботи додатку.

Технічні особливості реалізації алгоритму:

1. Асинхронність та керування потоками: Важливим архітектурним рішенням є використання асинхронної моделі програмування. Всі мережеві виклики до зовнішніх API реалізовані за допомогою ключових слів async та await з поверненням об'єктів Task. Це гарантує, що основний потік веб-сервера (Thread Pool) не блокується під час кількасекундного очікування відповіді від хмарних серверів Google або Spotify, що критично підвищує пропускну здатність (throughput) додатку при багатокористувацькому навантаженні.

2. Оптимізація мережевих з'єднань: Взаємодія на базовому рівні здійснюється через вбудований інтерфейс IhttpClientFactory. На відміну від прямого створення екземплярів HttpClient, цей патерн ефективно керує життєвим циклом HTTP-з'єднань, запобігаючи проблемі вичерпання сокетів (socket exhaustion) на сервері.

3. Безпека авторизації запитів: Для доступу до зовнішніх сервісів використовуються секретні ключі (API Keys для Gemini та Client ID / Secret для Spotify). Ці дані не «зашиті» (hardcoded) у класах, а безпечно ін'єктуються в сервіси під час їх ініціалізації через механізм Ioptions<T> з конфігураційного інструменту Secret Manager, що виключає їх компрометацію.

4. Відмовостійкість (Resilience): Алгоритм взаємодії враховує нестабільність зовнішніх мережевих ресурсів. У сервісах імплементовано механізми обробки виключень (try-catch блоки). У разі недоступності одного з API (отримання HTTP статусів 429 Too Many Requests або 5xx Server Error), система здатна коректно перехопити помилку та повернути користувачеві або резервний результат (Fallback), або відповідне інформаційне повідомлення замість критичного збою (Crash) додатку.

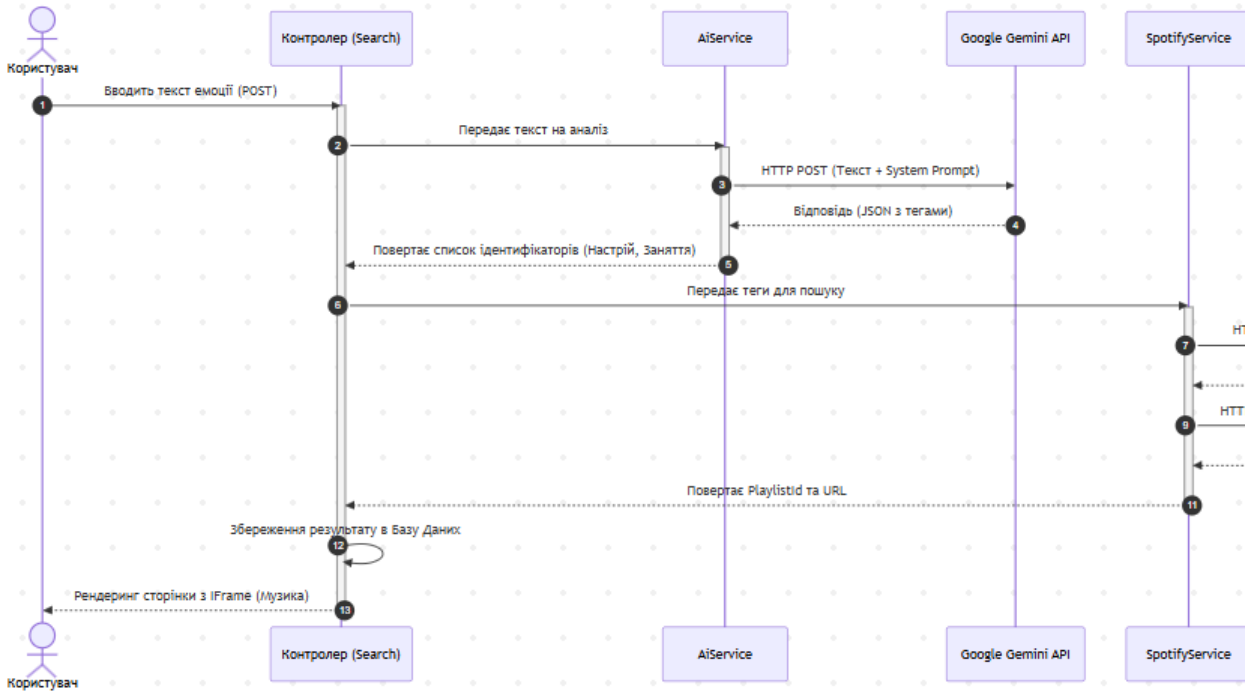

Рисунок 2.2 - UML-діаграма послідовності

2.3.1 Інтеграція з мовною моделлю Gemini (AI Backend)

Ключовим етапом інтелектуального підбору музики є подолання «семантичного розриву» - трансформація неструктурованого, емоційного природномовного запиту користувача у чіткий набір технічних ідентифікаторів. Цю функцію виконує хмарна велика мовна модель (LLM) від Google - Gemini, інтеграція з якою повністю інкапсульована у сервісному класі AiService.

Взаємодія з Google Gemini API відбувається за допомогою асинхронних HTTP-запитів формату REST (метод POST). Проте, оскільки генеративні нейромережі мають ймовірнісну (недетерміновану) природу і за замовчуванням генерують вільний зв'язний текст, критичним інженерним завданням було змусити модель працювати як суворий аналізатор даних (Data Parser).

Для досягнення цієї мети у проєкті було застосовано передові техніки Prompt Engineering (інженерії підказок). Алгоритм не просто відправляє текст користувача до API, а формує складний структурований Payload (корисне навантаження), в якому чітко розмежовані системні інструкції (System Instructions) та користувацькі дані (User Input).

У системний промпт жорстко запрограмовано три ключові архітектурні обмеження:

1. Завдання ролі (Persona Definition): Моделі призначається специфічна експертна роль ("Ти - емпатичний музичний експерт-психолог"). Це налаштовує внутрішні ваги нейромережі на використання словника, пов'язаного з психологією, емоціями та музичними жанрами.

2. Контекстуальне обмеження (Dynamic Grounding): Щоб уникнути "галюцинацій" (коли ШІ вигадує неіснуючі теги), сервіс динамічно підвантажує з бази даних актуальні списки настроїв та занять англійською мовою і вбудовує їх у промпт. ШІ отримує сувору вказівку обирати значення виключно з наданого переліку.

3. Структурування виводу (JSON Constraint): Моделі заборонено використовувати природну мову для відповіді. Встановлено сувору вимогу повертати результат виключно у форматі JSON без використання Markdown-розмітки (без службових символів ```json).

Після отримання відповіді від хмарного сервера Google, алгоритм виконує етап постобробки (Post-processing). Оскільки LLM іноді можуть ігнорувати інструкції щодо форматування і додавати Markdown-теги, у сервісі реалізовано алгоритм санітизації рядка (очищення за допомогою функцій маніпуляції текстом). Очищений та валідований JSON-рядок передається вбудованому десеріалізатору System.Text.Json, який трансформує його у строго типізований об'єкт мови C# (Data Transfer Object). З цього об'єкта вилучаються отримані англомовні ідентифікатори тегів, які надалі використовуються для формування фінального пошукового запиту до бази даних та музичного сервісу.

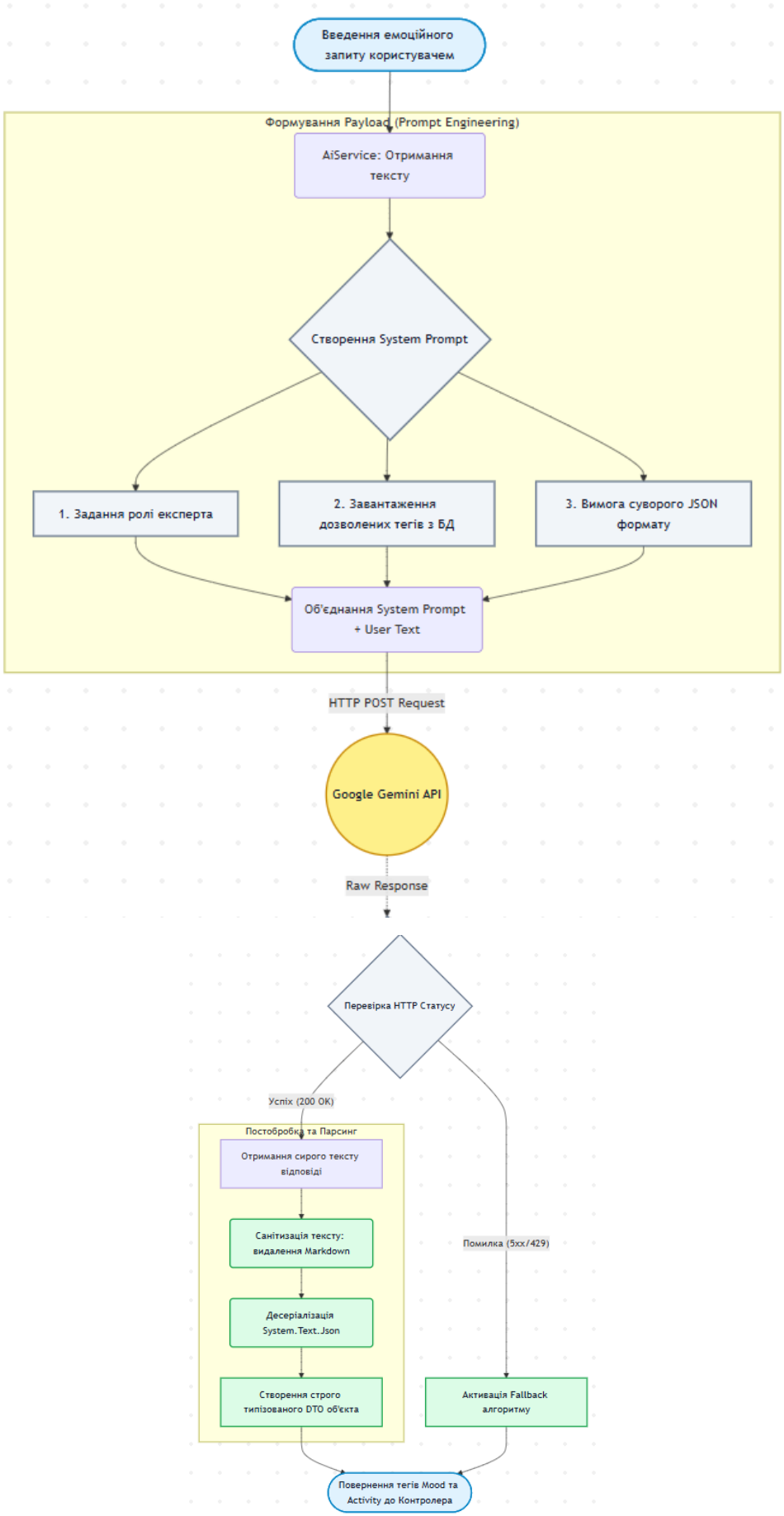

Рисунок 2.3 - Схема трансформації природномовного запиту у теги

2.3.2 Кешування та механізм відмовостійкості (Fallback)

Розробка систем, що покладаються на зовнішні хмарні сервіси (Third-Party APIs), неминуче стикається з проблемами мережевої затримки (Network Latency), жорсткими обмеженнями кількості запитів (Rate Limiting) та тимчасовою недоступністю віддалених серверів. Для забезпечення високої доступності (High Availability) додатку та стабільного користувацького досвіду (UX), в архітектуру сервісного шару було інтегровано два критично важливі патерни проєктування:

1. Пам'ятне кешування (In-Memory Caching за допомогою IMemoryCache) Оскільки багато емоційних або ситуативних запитів користувачів є типовими та часто повторюються (наприклад, "музика для тренування", "хочу розслабитися після роботи"), система оптимізує звернення до ШІ шляхом кешування результатів на стороні сервера.

– Алгоритм роботи: Перед відправкою запиту до Gemini API, алгоритм нормалізує вхідний текст користувача (переводить у нижній регістр, видаляє зайві пробіли та спецсимволи) і формує на його основі унікальний хеш-ключ. Система перевіряє наявність цього ключа в оперативній пам'яті сервера (IMemoryCache). Якщо подібний запит вже оброблявся нещодавно, сервер миттєво повертає готовий розпарсений JSON із кешу.

– Управління пам'яттю: Для запобігання переповненню оперативної пам'яті (Memory Leaks), для кожного запису встановлюється політика абсолютного часу життя (Absolute Expiration), яка автоматично видаляє застарілі дані через визначений проміжок часу.

– Переваги: Це архітектурне рішення радикально зменшує час очікування відповіді (Time-to-First-Byte), знімає навантаження з мережевого каналу та суттєво економить фінансові квоти на використання платного API нейромережі.

2. Механізм відмовостійкості та деградації (Fallback & Graceful Degradation) Хмарні великі мовні моделі (LLM) схильні до перевантажень під час пікових годин, що зазвичай супроводжується поверненням помилок HTTP 429 (Too Many Requests) або 503 (Service Unavailable). Щоб ізолювати користувача від цих

технічних проблем, у системі реалізовано патерн "Запасного парашута" (Fallback).

– Алгоритм роботи: Усі мережеві виклики до ШІ суворо інкапсульовані у захисні блоки try-catch. У разі виникнення тайм-ауту (Time-out) або фіксації критичного збою основної моделі (наприклад, gemini-1.5-flash), спрацьовує перехоплювач виключень.

– Резервні сценарії: Сервіс автоматично, без переривання сесії користувача, ініціює резервний сценарій: або перемикає запит на більш легку/швидку резервну модель ШІ, або, якщо API повністю недоступне, повертає базовий, статичний набір тегів за замовчуванням (наприклад, Activity: Relax, Mood: Calm).

– Переваги: Такий підхід забезпечує "елегантну деградацію" (Graceful Degradation) системи. Навіть у випадку повного падіння хмарної інфраструктури постачальника ШІ, веб-додаток не зазнає критичного збою (Application Crash), а користувач гарантовано отримає музичний контент, зберігши лояльність до сервісу.

2.3.3 Взаємодія зі Spotify API

Отримавши від мовної моделі чітко структуровані англомовні теги (наприклад, "Focus", "Chill", "Study"), система переходить до етапу пошуку релевантного аудіоконтенту. Цей процес делегується сервісному класу SpotifyService, який виступає мостом між веб-додатком та глобальним музичним каталогом Spotify. Використання саме англомовних тегів є критично важливим, оскільки внутрішній пошуковий рушій Spotify оптимізований переважно під англійську семантику, що гарантує найвищу точність результатів.

Взаємодія зі Spotify API здійснюється у два етапи та вимагає суворого дотримання протоколів безпеки.

Етап 1: Авторизація (OAuth 2.0 Client Credentials Flow) Оскільки додаток здійснює пошук публічних плейлистів від власного імені (сервер-до-сервера), а не модифікує особисті бібліотеки користувачів, архітектурно доцільним є використання потоку автентифікації Client Credentials Flow.

– Сервер додатку формує POST-запит до підсистеми Spotify Accounts Service, передаючи зашифровані у форматі Base64 ClientId та ClientSecret (збережені у безпечних змінних оточення).

– У відповідь Spotify генерує тимчасовий криптографічний ключ - Access Token, який зазвичай діє 1 годину (3600 секунд).

– Оптимізація: Щоб не витрачати мережеві ресурси на отримання нового токена перед кожним пошуковим запитом користувача, SpotifyService кешує отриманий Access Token в оперативній пам'яті сервера, відстежуючи час його життя. Запит до серверів авторизації відбувається лише тоді, коли токен відсутній або його термін дії добігає кінця.

Етап 2: Пошук та парсинг результатів Маючи валідний Access Token, сервіс формує пошуковий HTTP GET-запит до Spotify Web API (до еднпойнту /v1/search). Токен передається у заголовку запиту (Authorization: Bearer {Token}).

– URL-кодування: Теги об'єднуються у рядок запиту з дотриманням правил URL Encoding (наприклад, ?q=Chill+Focus&type=playlist&limit=10).

– Рандомізація: Щоб користувач не отримував один і той самий плейлист при однакових запитах, сервіс отримує масив із кількох релевантних результатів (наприклад, 10-15 плейлистів) і за допомогою генератора псевдовипадкових чисел обирає один із них.

– Отримана JSON-відповідь десеріалізується, з неї вилучається унікальний ідентифікатор плейлиста (Spotify URI / PlaylistId), його назва та пряме посилання (URL).

Ці метадані повертаються на рівень контролера, зберігаються в базі даних (у таблицю SearchHistories) і передаються на клієнтський інтерфейс (View). Там ідентифікатор динамічно вставляється у HTML-тег <iframe>, який генерує віджет вбудованого плеєра (Spotify Embed). Для забезпечення безперебійного відтворення аудіо з дотриманням політик безпеки сучасних браузерів, для фрейма обов'язково встановлюється атрибут allow="encrypted-media".

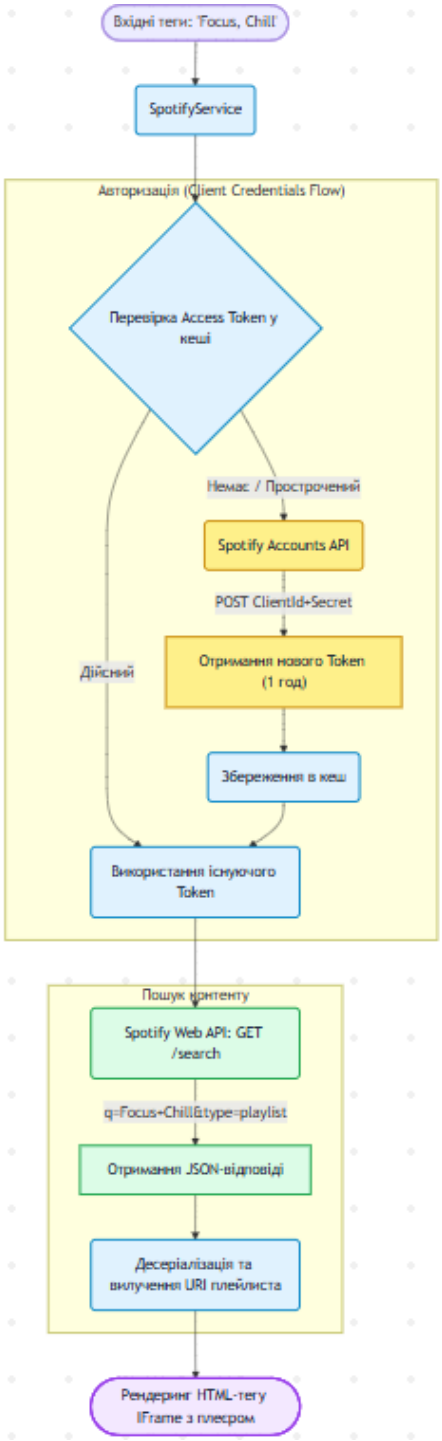

Рисунок 2.4 - Алгоритм взаємодії зі Spotify API

2.4 Реалізація клієнтської частини вебсайту

2.4.1 Проєктування та реалізація головної сторінки вебсайту

Головна сторінка розробленого вебдодатку є центральним вузлом та основним робочим простором користувача для взаємодії з інтелектуальною

рекомендаційною системою. Дизайн сторінки спроєктовано з урахуванням сучасних ергономічних вимог (див. рисунок 2.5).

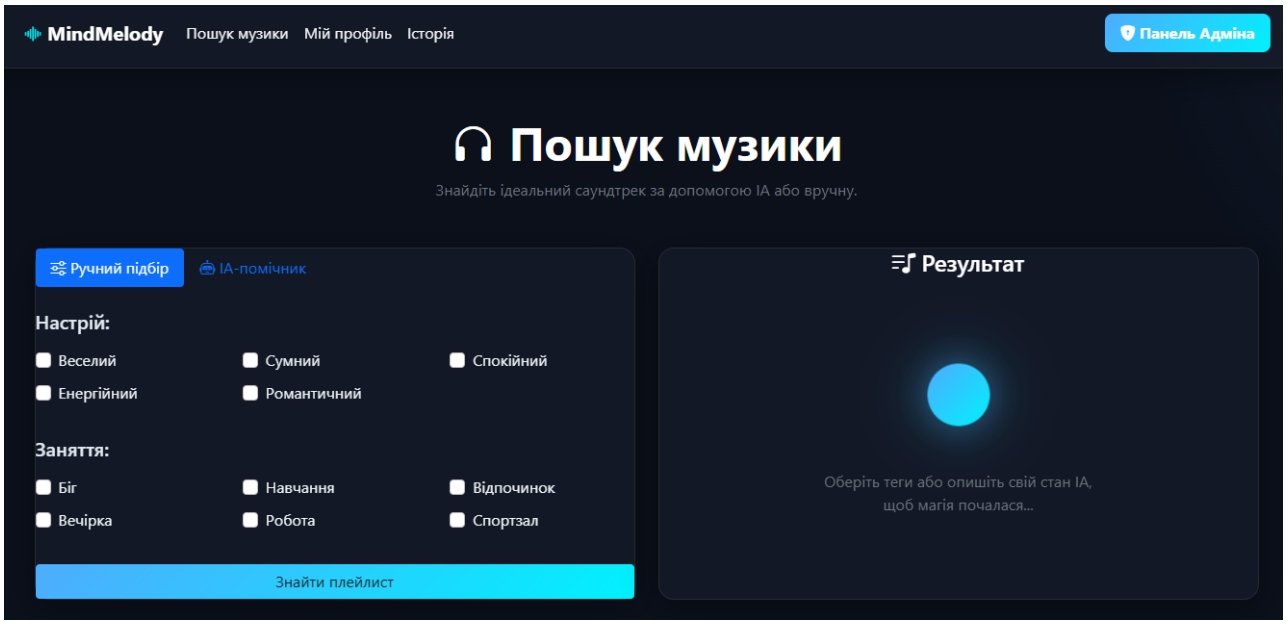

Рисунок 2.5 - Інтерфейс головної сторінки вебдодатку

Після завантаження сторінки користувача зустрічає мінімалістичний заголовок та робоча область, візуально розділена на два основні функціональні блоки (лівий - для введення параметрів, правий - для відображення результату). Візуальний стиль базується на концепції Glassmorphism: напівпрозорі панелі з ефектом розмиття фону (blur) розміщені на глибокому темному тлі, що створює атмосферу технологічності та знижує навантаження на зір.

Сторінка містить наступні ключові інтерактивні компоненти:

– Адаптивна навігаційна панель (Navbar): Розташована у верхній частині екрана. Вона забезпечує швидкий доступ до особистого профілю користувача та історії пошуку. Важливою деталлю є кнопка «Панель Адміна», яка рендериться динамічно лише для користувачів із відповідною роллю доступу.

– Вкладки режимів пошуку (Tabs): У верхній частині лівої панелі розміщено перемикач між двома алгоритмами підбору: «Ручний підбір» (через теги) та «IA-помічник» (через текстовий запит природною мовою). Вибір вкладки миттєво змінює інтерфейс форми введення нижче.

– Матриця категорій (Чекбокси): У режимі ручного підбору користувачу пропонується структурована сітка чекбоксів, розділена на логічні категорії: «Настрій» (Веселий, Сумний тощо) та «Заняття» (Біг, Навчання, Робота). Такий UI-патерн мінімізує когнітивне навантаження при виборі параметрів.

– Динамічна панель результату (Права секція): До моменту здійснення пошуку ця панель відображає неонову анімацію очікування (пульсуюче коло) та підказку-плейсхолдер: *"Оберіть теги або опишіть свій стан ІА..."*. Після успішної обробки запиту сервером, ця область динамічно замінюється на HTML-тег <iframe>, в якому завантажується інтерактивний віджет плеєра Spotify із підібраним плейлистом.

– Кнопка генерації («Знайти плейлист»): Має яскравий неоновий акцент (Cyan колір), який привертає увагу користувача та слугує головним закликом до дії (Call-to-Action).

З технічної точки зору, головна сторінка реалізована за допомогою технології Razor Pages / MVC (файл Index.cshtml), яка дозволяє ін'єктувати серверні дані безпосередньо в HTML-розмітку. Каркас сторінки та її адаптивність (Responsive поведінка на мобільних пристроях) забезпечуються класами сітки фреймворку Bootstrap 5.

Логіка клієнтської взаємодії керується за допомогою нативного Vanilla JavaScript. Перемикання між вкладками «Ручний підбір» та «ІА-помічник» реалізовано через маніпуляцію об'єктною моделлю документа (DOM) шляхом перемикання CSS-класів видимості (d-none), що забезпечує миттєву зміну інтерфейсу без перезавантаження сторінки. Відправка даних із форми до контролера здійснюється асинхронно (через API fetch або AJAX). Це дозволяє зберегти стан сторінки, плавно приховати анімацію очікування у правій панелі та відобразити результати роботи ШІ без дратівливого мерехтіння чи повного оновлення екрана (Single-Page Application experience).

2.4.2 Проєктування та реалізація сторінки особистого профілю та статистики

Сторінка особистого профілю (див. Рис. 2.16) відіграє роль персонального аналітичного дашборду. Її головна мета - надати користувачеві зручний інструментарій для управління власним обліковим записом, а також візуалізувати історію його емоційно-музичних запитів. Візуальне оформлення продовжує загальну стилістику Glassmorphism із застосуванням глибоких темних відтінків та контрастних акцентів.

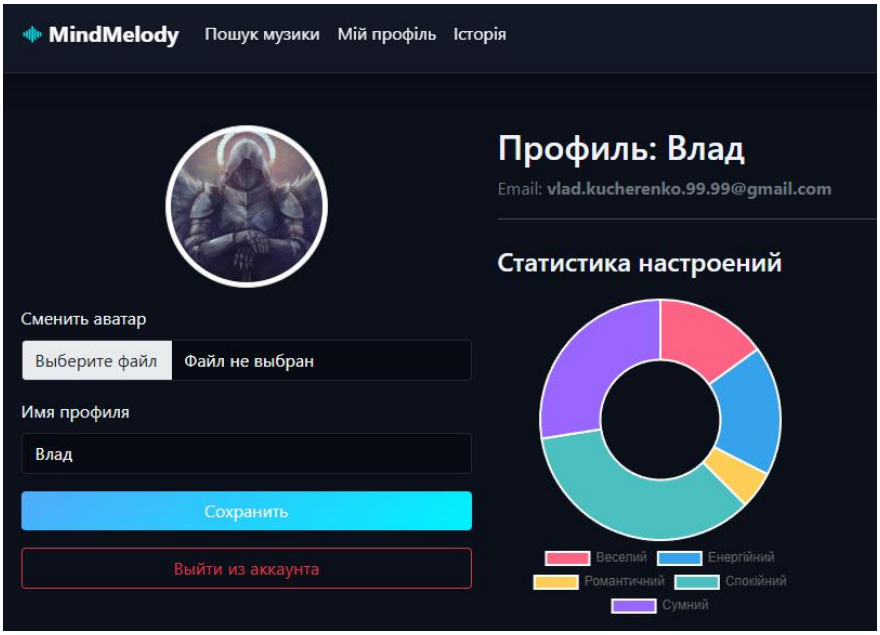

Рисунок 2.6 - Інтерфейс профілю користувача з аналітикою настроїв

Інтерфейс сторінки логічно та структурно поділено на дві адаптивні секції (колонки):

1. Панель управління профілем (Ліва секція) Цей блок відповідає за персоналізацію та управління сесією. Він містить наступні елементи:

– Динамічний аватар: Відображає поточну фотографію користувача у круглому фреймі.

– Форма оновлення даних: Дозволяє завантажити нове зображення профілю та змінити ім'я для відображення (Display Name). Завантаження файлів реалізовано через форму з атрибутом enctype="multipart/form-data". На стороні сервера завантажене зображення валідується, зберігається у статичну

директорію вебсервера (папка wwwroot), а шлях до файлу безпечно записується у кастомне поле AvatarUrl (або ProfilePicturePath) таблиці Users бази даних.

– Елементи безпеки: Кнопка «Зберегти» (із застосуванням основного Cyan-градієнту) для підтвердження змін та контрастна кнопка «Вийти з акаунта» для безпечного завершення сесії користувача та видалення автентифікаційних файлів cookie.

2. Блок користувацької аналітики (Права секція) Верхня частина секції відображає ідентифікаційні дані (Ім'я та підтверджений Email). Ключовою інноваційною складовою сторінки є блок «Статистика настроїв», який реалізує функціонал візуалізації даних (Data Visualization).

Для наочної демонстрації музичних вподобань користувача імплементовано інтерактивну кільцеву діаграму (Donut Chart). Кожен сегмент діаграми відповідає певній емоції (Веселий, Романтичний, Сумний, Спокійний, Енергійний) і має власне кольорове кодування, що дублюється у зручній легенді під графіком.

Технічна реалізація підготовки даних для діаграми: Рендеринг діаграми відбувається на стороні клієнта за допомогою спеціалізованої JavaScript-бібліотеки (наприклад, Chart.js). Проте обчислення статистики повністю покладено на серверну частину. Алгоритм роботи наступний:

1. Контролер профілю отримує ідентифікатор поточного авторизованого користувача (UserId) через підсистему ASP.NET Core Identity.

2. За допомогою технології LINQ та Entity Framework Core виконується складний запит до бази даних. Система агрегує дані з транзакційної таблиці SearchHistories та пов'язаної через JOIN асоціативної таблиці SearchHistoryMoods.

3. Результати групуються (GroupBy) за типами настроїв, обчислюється їхня частота у відсотковому співвідношенні.

4. Сформований словник передається у клієнтське представлення (View) у форматі JSON-об'єкта, який безпосередньо ініціалізує скрипт побудови графіка.

Доступ до сторінки профілю суворо захищений маршрутизатором за допомогою атрибута авторизації [Authorize]. Це гарантує, що неавторизовані відвідувачі будуть автоматично перенаправлені на сторінку входу, а авторизовані користувачі матимуть доступ виключно до власної, ізольованої аналітики.

2.4.3 Проєктування та реалізація сторінки історії пошуку

Сторінка історії пошуку (див. рисунок 2.7) створена для забезпечення довгострокової цінності сервісу, дозволяючи користувачам повертатися до раніше згенерованих музичних підборів. Вона виконує роль персональної бібліотеки музичних рекомендацій, збережених у хронологічному порядку.

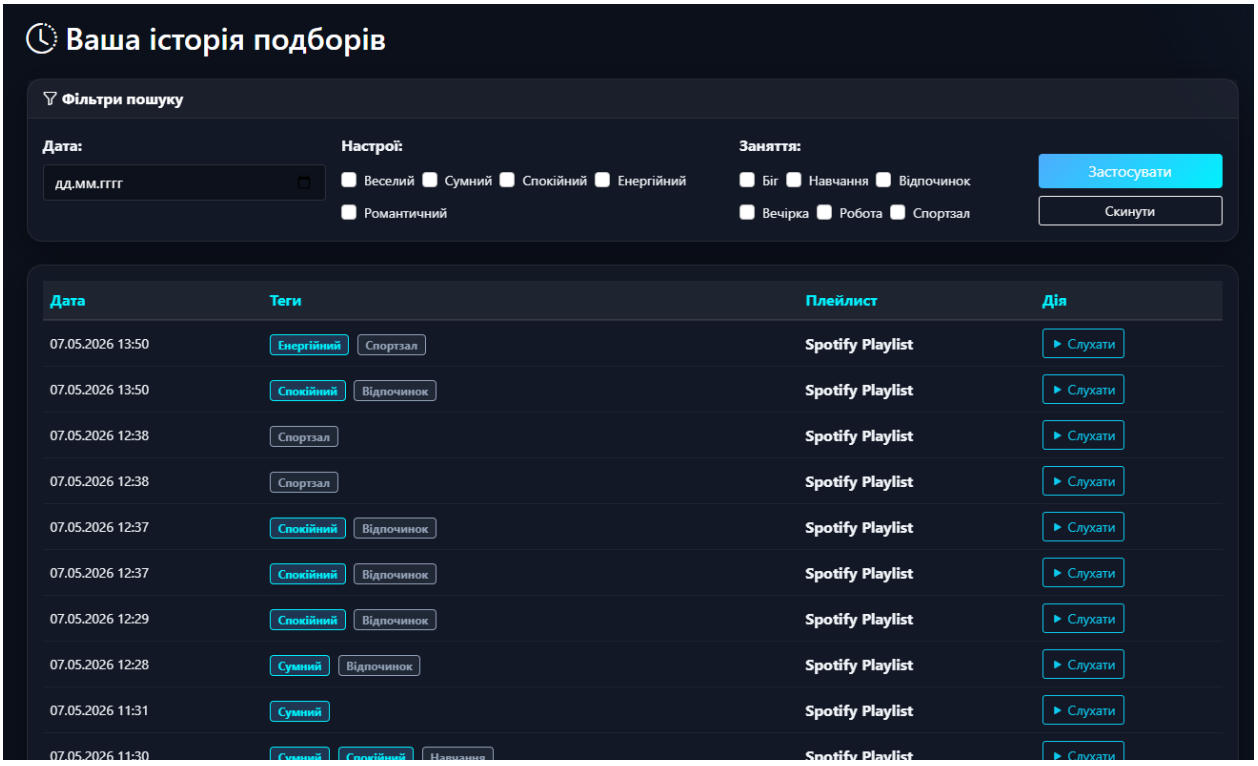

Рисунок 2.7 - Інтерфейс сторінки історії пошуку з панеллю фільтрації

Візуальна структура сторінки підтримує єдину дизайн-систему проєкту та складається з двох основних функціональних блоків:

1.Панель розширеної фільтрації («Фільтри пошуку») Верхній блок інтерфейсу надає користувачу потужний інструментарій для пошуку специфічних записів у великому масиві історичних даних. Панель містить:

– Календарний фільтр (Date Picker): Дозволяє фільтрувати історію за конкретною датою створення запиту.

– Матриця категорій: Набір чекбоксів, що дублює логіку головної сторінки (групи «Настрій» та «Заняття»). Це дозволяє здійснити швидку вибірку, наприклад, знайти всі плейлисти, які користувач шукав для «Спортзалу» в «Енергійному» стані.

– Кнопки управління: Елементи «Застосувати» (для відправки параметрів фільтрації на сервер) та «Скинути» (для очищення форми та повернення до повного списку).

2.Табличне представлення даних (Data Grid) Нижня частина сторінки реалізована у вигляді адаптивної таблиці, яка динамічно рендериться на основі даних з БД. Таблиця містить наступні колонки:

– Дата: Точна часова мітка (Timestamp) створення запиту.

– Теги: Семантичні параметри, за якими здійснювався пошук. Для покращення UX (User Experience) теги візуалізовані у вигляді інтерактивних «пігулок» (Badges) з кольоровою диференціацією (наприклад, Cyan для активних станів та сірий контур для контекстних занять).

– Плейлист: Назва знайденого музичного контенту (збережена з API Spotify).

– Дія: Інтерактивна кнопка «Слухати» (із застосуванням іконки відтворення). При натисканні на неї користувач перенаправляється на сторінку відтворення або викликається модальне вікно з IFrame-плеєром відповідного плейлиста.

Технічні аспекти реалізації: З боку серверної архітектури (Backend), генерація цієї сторінки вимагає побудови складних реляційних запитів. Оскільки дані розподілені між кількома таблицями, у контролері використовується механізм жадібного завантаження (Eager Loading) фреймворку Entity Framework Core. За допомогою методів .Include() та .ThenInclude() сервер одним SQL-запитом витягує основні записи з таблиці SearchHistories, а також усі пов'язані з ними теги з асоціативних таблиць SearchHistoryMoods та SearchHistoryActivities.

Логіка фільтрації реалізована на сервері з використанням динамічного складання LINQ-запитів. Залежно від обраних користувачем чекбоксів, до базового запиту (IQueryable) послідовно додаються умови фільтрації (.Where(...)). Це гарантує, що база даних повертає лише необхідний набір записів, мінімізуючи навантаження на оперативну пам'ять сервера та мережевий канал, що є критично важливим патерном при проєктуванні масштабованих вебдодатків.

2.4.4 Проєктування та реалізація панелі адміністратора (Admin Dashboard)

Для забезпечення ефективного моніторингу та управління вебдодатком було розроблено закритий модуль - Панель управління (Admin Dashboard). Ця сторінка (див. рисунок 2.8) призначена виключно для персоналу з найвищим рівнем доступу і виконує роль глобального аналітичного центру системи.

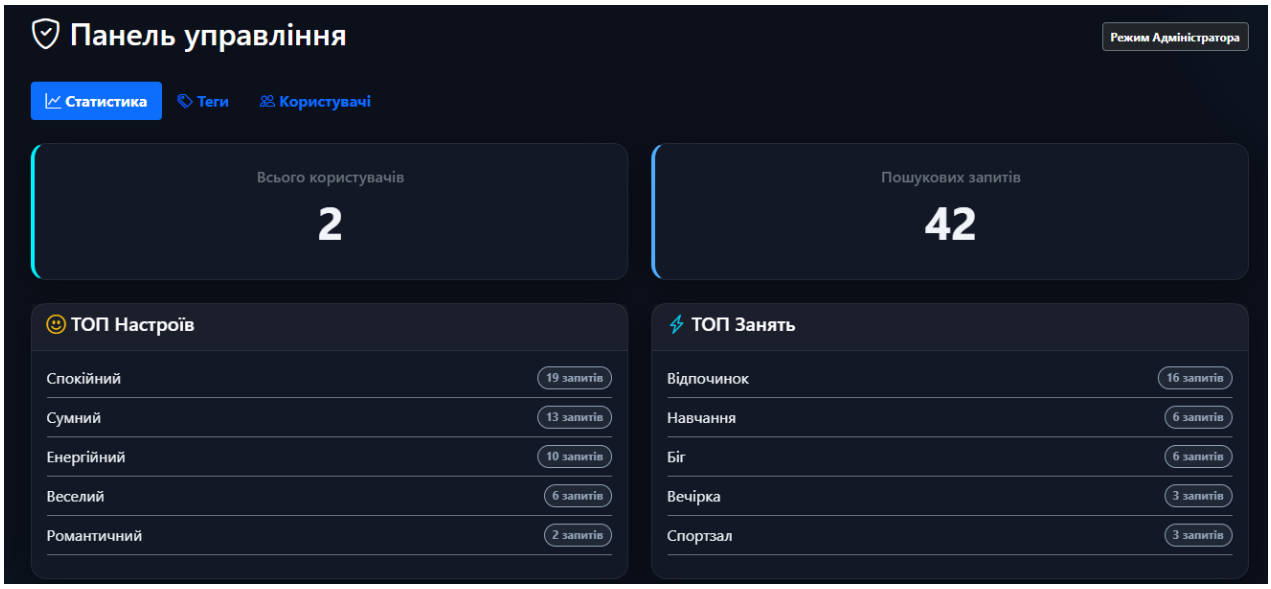

Рисунок 2.8 - Інтерфейс аналітичної панелі адміністратора

З точки зору інформаційної безпеки, доступ до цього маршруту (Route) жорстко обмежений на рівні контролера за допомогою механізму Role-Based Access Control (RBAC). Використання атрибута [Authorize(Roles = "Admin, SuperAdmin")] гарантує, що будь-який користувач без відповідних прав при

спробі зайти на цю сторінку отримає помилку доступу (HTTP 403 Forbidden) або буде перенаправлений на сторінку входу.

Інтерфейс панелі управління поділено на три логічні вкладки: «Статистика», «Теги» та «Користувачі», які покривають усі потреби адміністрування системи.

1. Глобальні метрики системи (Вкладка «Статистика»)

У верхній частині робочої області розташовані масивні картки (віджети), які відображають ключові показники ефективності (KPI) у режимі реального часу: загальну кількість зареєстрованих акаунтів та глобальну кількість успішно оброблених пошукових запитів. Нижня частина екрана розділена на два симетричні блоки («ТОП Настроїв» та «ТОП Занять»), що візуалізують макро-тренди користувацької поведінки. Кожен рядок містить назву тегу та кількість запитів. Ця інформація є критично важливою для розуміння того, які саме емоційні стани найчастіше спонукають користувачів шукати музику.

2. Управління довідниками (Вкладка «Теги»)

Цей модуль (див. Рисунок 2.12) відповідає за адміністрування базових сутностей системи - настроїв та занять, які використовуються для формування системного промпту ШІ.

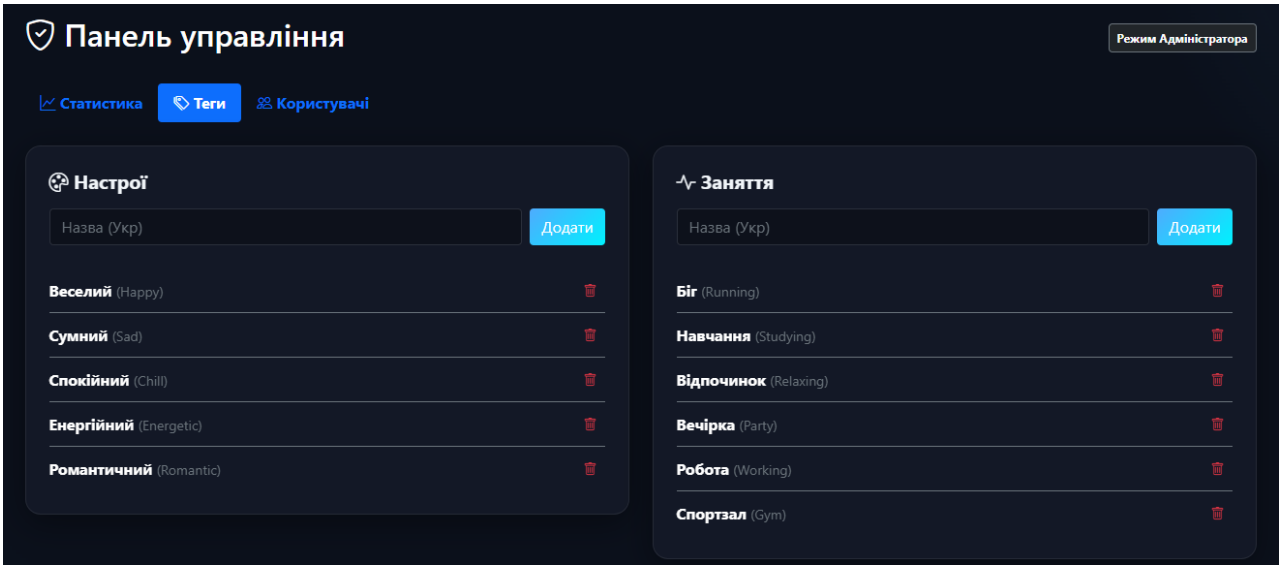

Рисунок 2.9 - Інтерфейс управління музичними тегами та довідниками

Інтерфейс складається з двох панелей, які дозволяють виконувати базові CRUD-операції (створення, читання, видалення). Адміністратор має змогу динамічно додавати нові теги, вказуючи їх локалізовану українську назву (для відображення на клієнті). У списках також виводиться англомовний еквівалент тегу, необхідний для коректної роботи Gemini API. Кожен існуючий запис містить інтерактивну іконку видалення, що дозволяє швидко очищати систему від нерелевантних категорій.

3. Управління обліковими записами (Вкладка «Користувачі»).

Дана вкладка (див. рисунок 2.10) надає табличне представлення всіх зареєстрованих акаунтів у системі.

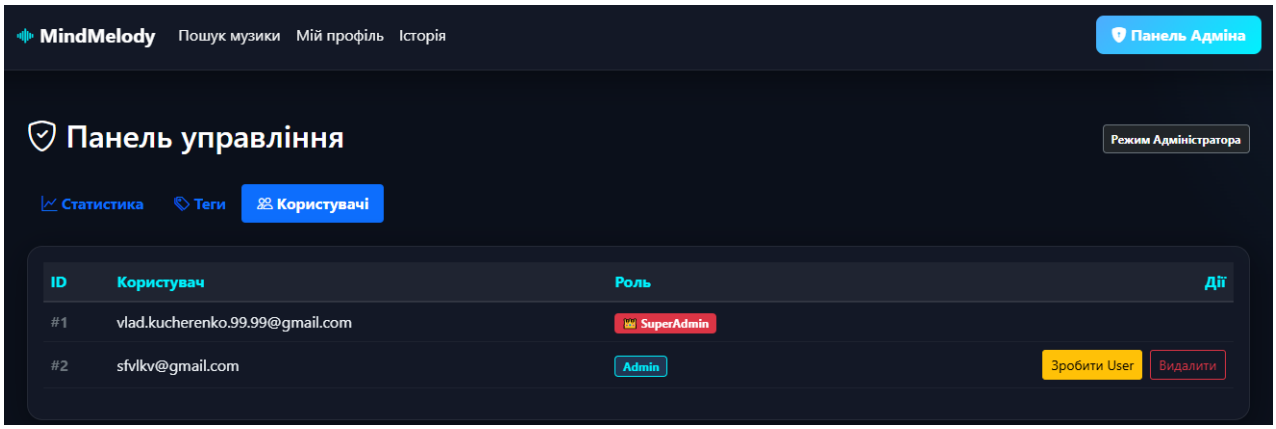

Рисунок 2.10 - Інтерфейс моніторингу та управління користувачами

Таблиця містить унікальний ідентифікатор, електронну пошту та поточний рівень доступу (Роль), який для наочності візуалізований за допомогою кольорових бейджів (наприклад, червоний для SuperAdmin, синій для Admin). Функціонал панелі дій («Дії») дозволяє адміністратору вищого рівня динамічно змінювати привілеї інших користувачів (наприклад, кнопка «Зробити User» для пониження прав) або повністю ініціювати видалення облікового запису із системи у разі порушення правил сервісу.

Технічна реалізація та обробка даних: Для забезпечення високої продуктивності та безпеки, робота панелі адміністратора спирається на оптимізовані серверні алгоритми:

1. Агрегація статистики: Обчислення виконуються на рівні СУБД за допомогою оптимізованих LINQ-запитів. Глобальні лічильники реалізуються через асинхронні виклики .CountAsync(), а ТОП-рейтинги формуються через об'єднання таблиць (Join), групування (.GroupBy()) та сортування лідерів (.OrderByDescending(...).Take(...)).

2. Керування довідниками: Операції з тегами обробляються через стандартні методи Entity Framework Core (.Add(), .Remove()). Зміни фіксуються транзакційно через метод .SaveChangesAsync(), що гарантує цілісність бази даних.

3. Управління ідентифікацією: Зміна ролей та видалення користувачів інкапсульовані через вбудовані сервіси UserManager<TUser> та RoleManager<TRole> архітектури ASP.NET Core Identity. Це гарантує криптографічну безпеку операцій та коректне виконання правил каскадного видалення (Cascade Delete), завдяки чому при видаленні профілю користувача система автоматично очищає і всю його пов'язану історію пошуку.

2.5 Впровадження підсистеми автентифікації користувачів

Забезпечення персоналізованого досвіду (збереження історії пошуку, генерація індивідуальної аналітики) та захист адміністративних маршрутів вимагають впровадження надійної системи ідентифікації користувачів. У даному дипломному проєкті цю функцію покладено на вбудований фреймворк ASP.NET Core Identity, який забезпечує індустріальні стандарти безпеки (криптографічне хешування паролів, захист від атак перебору тощо) "з коробки".

2.5.1 Розробка інтерфейсу та логіки сторінок реєстрації та автентифікації

Для забезпечення безперешкодного доступу користувачів до системи були спроєктовані сторінки реєстрації нового акаунта та входу в існуючий (автентифікації). Дизайн цих сторінок (див. рисунок 2.11) побудований за

принципом мінімалізму, щоб максимально сфокусувати увагу користувача на цільовій дії.

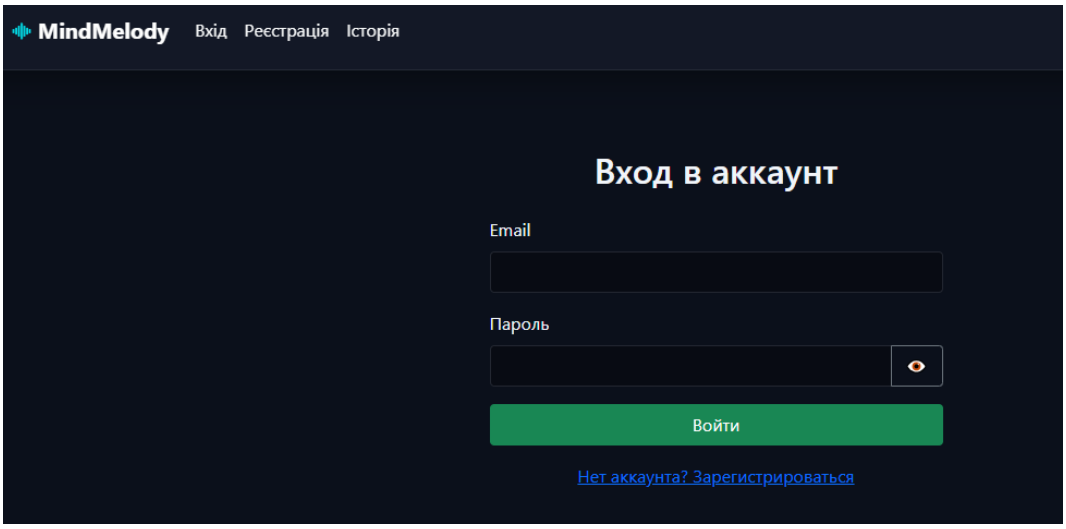

Рисунок 2.11 - Інтерфейс підсистеми автентифікації

1. Проєктування користувацького інтерфейсу (UI/UX) Форми введення даних розміщені по центру екрана на темному тлі, що відповідає загальній стилістиці платформи MindMelody. Для покращення користувацького досвіду (User Experience) впроваджено наступні рішення:

– Чітка ієрархія полів: Використовуються стандартні HTML5-поля з відповідними типами (type="email", type="password"), що дозволяє браузерам автоматично пропонувати збережені облікові дані.

– Перемикач видимості пароля: У полі введення пароля реалізовано інтерактивну іконку (у вигляді ока). За допомогою невеликого скрипта на Vanilla JavaScript ця кнопка змінює атрибут поля з password на text і навпаки, дозволяючи користувачу перевірити правильність введених символів перед відправкою форми. Це суттєво знижує відсоток помилок при реєстрації.

– Контрастний Call-to-Action: Кнопки підтвердження (наприклад, зелена кнопка «Войти») мають високий кольоровий контраст.

– Швидка навігація: Під основною формою розташоване гіперпосилання для швидкого переходу між режимами («Немає акаунта? Зареєструватися» та навпаки).

2. Багаторівнева валідація даних Для запобігання відправці некоректних даних на сервер імплементовано дворівневу систему перевірки:

– Клієнтська валідація (Client-side): Виконується миттєво у браузері за допомогою атрибутів [Required], [EmailAddress] та бібліотеки jQuery Validation. Вона блокує відправку форми, якщо поля порожні або формат email-адреси є хибним.

– Серверна валідація (Server-side): У контролері перевіряється стан моделі (ModelState.IsValid). Також перевіряється унікальність електронної пошти у базі даних, щоб уникнути дублювання акаунтів.

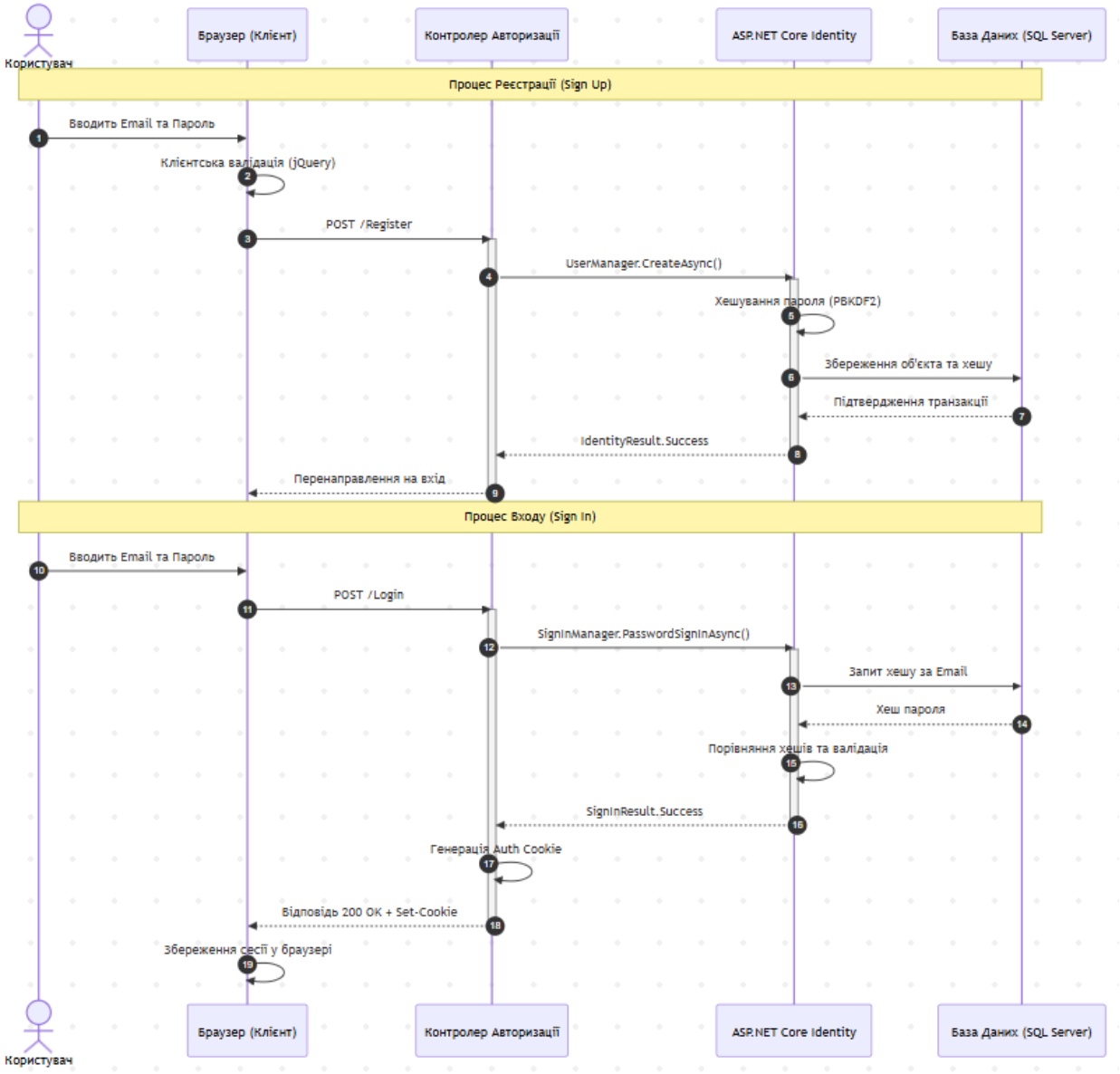

Рисунок 2.12 - Послідовність реєстрації та автентифікації користувачів

3. Логіка реєстрації нового користувача (Sign Up) Коли користувач відправляє форму реєстрації, дані потрапляють до методу контролера. Алгоритм роботи наступний:

– Система створює новий об'єкт класу User.

– Викликається асинхронний метод UserManager.CreateAsync(user, password). На цьому етапі фреймворк Identity автоматично застосовує алгоритм PBKDF2 (з унікальною сіллю) для перетворення відкритого пароля у криптографічний хеш. У базі даних (таблиця Users) паролі у відкритому вигляді ніколи не зберігаються.

– У разі успішного створення, новому користувачеві автоматично призначається базова роль (наприклад, «User») за допомогою методу UserManager.AddToRoleAsync().

4. Логіка автентифікації та створення сесії (Sign In) При спробі входу (як показано на Рис. 2.21), контролер приймає Email та пароль і передає їх до сервісу SignInManager.

– Метод PasswordSignInAsync знаходить користувача за Email і порівнює хеш введеного пароля з хешем у базі даних.

– Якщо дані збігаються, сервер генерує зашифрований автентифікаційний файл Cookie (AspNetCore.Identity.Application) і відправляє його браузеру.

– Цей Cookie-файл автоматично додається до всіх наступних запитів користувача, дозволяючи серверу "впізнавати" його, надавати доступ до історії та відображати персоналізований інтерфейс (наприклад, замінити кнопки "Вхід/Реєстрація" у навігаційній панелі на аватар профілю).

2.6 Реалізація підсистеми пошуку музичного контенту

Основною цінністю розробленого вебдодатку є надання користувачеві релевантного музичного контенту. Для забезпечення максимальної гнучкості користувацького досвіду (UX), система пропонує два незалежні алгоритми

підбору: детермінований (ручний вибір за категоріями) та інтелектуальний (за допомогою ШІ).

2.6.1 Алгоритм ручного підбору за допомогою категоріальних фільтрів

Режим «Ручний підбір» (див. рисунок 2.13) розроблено для користувачів, які чітко усвідомлюють свої поточні потреби та бажають отримати передбачуваний результат без необхідності формулювати текстовий запит для нейромережі.

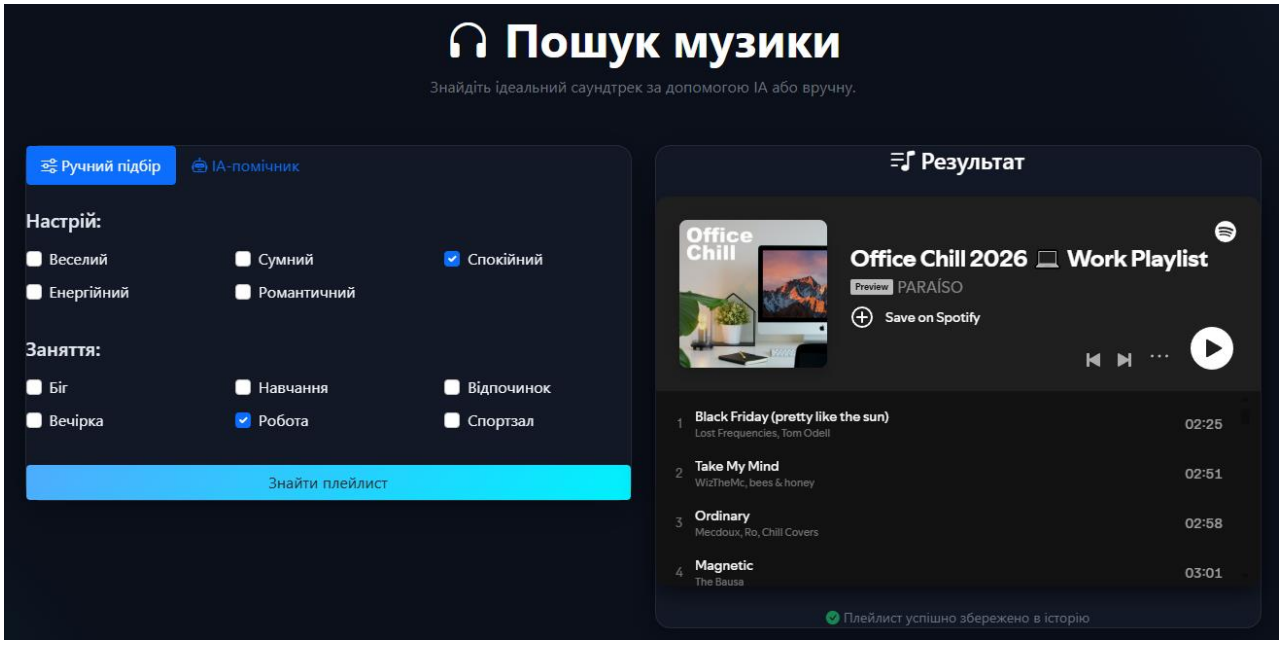

Рисунок 2.13 - Інтерфейс виконання ручного пошуку плейлиста

1. Користувацький інтерфейс та формування запиту В основі інтерфейсу лежить матриця незалежних перемикачів типу checkbox. На відміну від радіокнопок (radio), чекбокси дозволяють користувачу формувати комбіновані запити (наприклад, одночасно обрати настрій «Спокійний» та заняття «Робота», як показано на рисунку). Після натискання кнопки «Знайти плейлист», JavaScript-обробник на стороні клієнта збирає масив ідентифікаторів обраних чекбоксів і відправляє їх на сервер за допомогою асинхронного POST-запиту (AJAX/Fetch). Під час очікування відповіді кнопка може блокуватися для запобігання дублюванню запитів (техніка Debouncing).

2. Серверна логіка та маппінг тегів Отримавши масив ідентифікаторів, контролер виконує критично важливий етап - трансляцію (маппінг) локалізованих тегів. Оскільки пошуковий рушій Spotify Web API працює найефективніше з англомовною семантикою, сервер звертається до бази даних (таблиці Moods та Activities) і вилучає значення з колонки EnglishName. Таким чином, обрані користувачем українські чекбокси «Спокійний» та «Робота» на сервері трансформуються у пошуковий рядок: q=Chill+Work&type=playlist. Цей рядок передається до сервісу SpotifyService для отримання результату.

3. Рендеринг результату (Spotify IFrame) Після отримання відповіді від Spotify API (у якій міститься унікальний ідентифікатор знайденого плейлиста, наприклад Office Chill 2026), сервер повертає ці дані на клієнт. Інтерфейс правої панелі («Результат») динамічно оновлюється: плейсхолдер зникає, а на його місці генерується HTML-тег <iframe>.

<iframe
src="https://open.spotify.com/embed/playlist/{SpotifyId}?utm_source=generator"
width="100%" height="352" frameBorder="0" allowfullscreen=""
allow="autoplay; clipboard-write; encrypted-media; fullscreen; picture-in-picture">
</iframe>

Цей віджет дозволяє користувачу прослуховувати музику, перемикати треки та зберігати плейлист у свою особисту бібліотеку Spotify безпосередньо на сторінці додатку.

4. Інтеграція з підсистемою історії Важливою архітектурною вимогою є збереження результатів пошуку. Як видно на Рис. 2.23, під плеєром з'являється системне повідомлення (Toast notification) зеленого кольору: «Плейлист успішно збережено в історію». На рівні бекенду це означає, що перед тим як повернути результат користувачу, сервер ініціював транзакцію до бази даних. Було створено новий запис у таблиці SearchHistories із прив'язкою до поточного UserId, а також заповнено проміжні таблиці SearchHistoryMoods та

SearchHistoryActivities обраними тегами. Це гарантує, що користувач зможе в будь-який момент повернутися до цього плейлиста через сторінку «Історія».

2.6.2 Інтелектуальний підбір музичного контенту за допомогою Великої мовної моделі (LLM)

Альтернативним і найбільш інноваційним режимом роботи системи є «ІА-помічник» (див. рисунок 2.14). Цей модуль вирішує фундаментальну проблему традиційних пошукових систем - неможливість обробки неструктурованих, емоційно забарвлених запитів природною мовою (Natural Language).

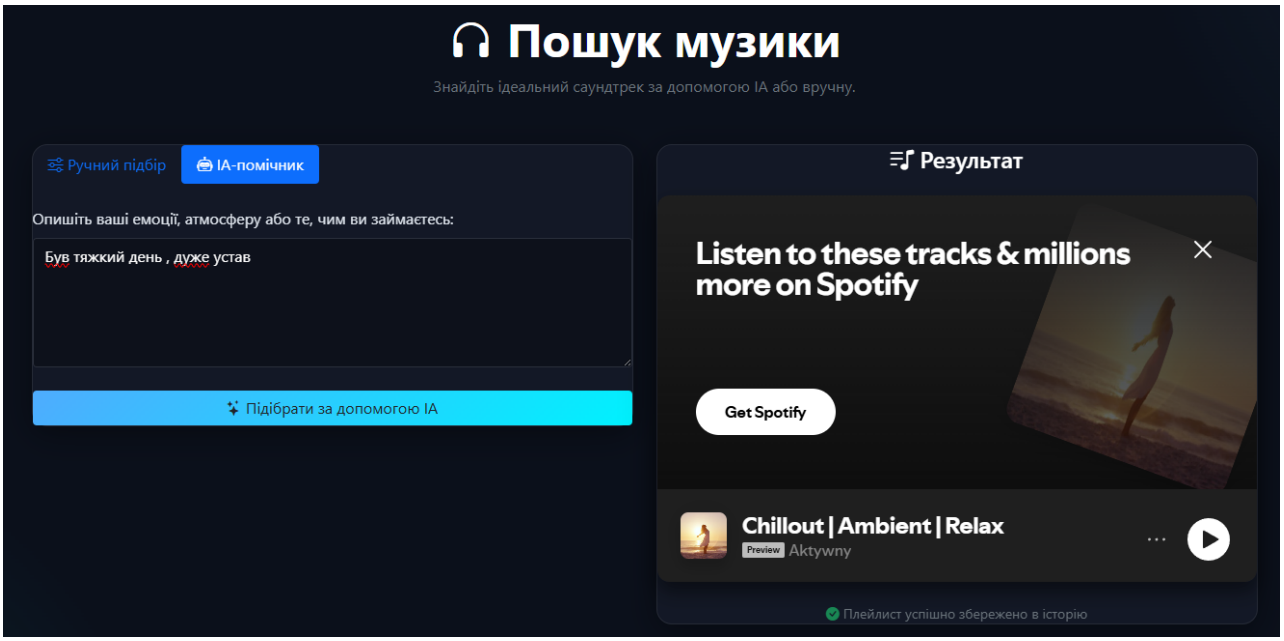

Рисунок 2.14 - Інтерфейс підбору музики на основі опису

1. Інтерфейс обробки природної мови (NLP Interface)

При перемиканні на вкладку «ІА-помічник» користувачу надається багаторядкове текстове поле (textarea). Замість жорстких категорій, система заохочує користувача до вільного вираження думок (наприклад: "Був тяжкий день, дуже устав", як показано на рисунку). Такий підхід значно знижує когнітивне навантаження, оскільки користувачу не потрібно самостійно аналізувати свій стан і підбирати відповідні фільтри.

2. Семантичний аналіз та вилучення намірів (Intent Extraction)

Після натискання кнопки «Підібрати за допомогою ІА», клієнтський додаток відправляє введений неструктурований текст на сервер. Далі алгоритм діє за наступним сценарієм:

– Сервер передає текст користувача до сервісного класу AiService.

– Застосовуючи техніку Prompt Engineering, сервіс формує комплексний запит до API Google Gemini. У системному промпті нейромережі ставиться завдання: проаналізувати семантику тексту "Був тяжкий день, дуже устав", розпізнати в ньому ознаки фізичної втоми та потреби в релаксації, і зіставити їх з наявними в базі даних тегами.

– Модель (LLM) виконує математичне зіставлення у прихованому багатовимірному просторі (Latent Space) і повертає структуровану JSON-відповідь. Наприклад, для даного запиту ШІ алгоритмічно обирає теги Mood: Спокійний (Chill) та Activity: Відпочинок (Relaxing).

3. Інтеграція з музичним провайдером та рендеринг

Отримавши від ШІ чіткий набір англомовних параметрів, серверна логіка повертається до стандартного потоку виконання (аналогічного ручному пошуку):

– SpotifyService здійснює GET-запит до каталогу Spotify із параметрами, згенерованими нейромережею.

– Spotify повертає найбільш релевантний плейлист (у випадку наведеного скриншота - це плейлист "Chillout | Ambient | Relax").

– На клієнті генерується HTML-віджет IFrame, який дозволяє миттєво розпочати прослуховування заспокійливої музики.

4.Забезпечення цілісності історичних даних

Як і в детермінованому алгоритмі, результат роботи ШІ автоматично фіксується в базі даних (про що свідчить повідомлення «Плейлист успішно збережено в історію»). Унікальність цього процесу полягає в тому, що в таблиці SearchHistories зберігаються саме ті структуровані теги, які згенерувала нейромережа. Це дозволяє адміністраторам системи в майбутньому аналізувати

статистику (через Панель Адміністратора) і бачити, які саме музичні категорії найчастіше рекомендує штучний інтелект на реальні запити користувачів.

2.7 Практична апробація методики використання інформаційної системи

Для підтвердження працездатності розробленого програмного забезпечення та ілюстрації його користувацьких характеристик було проведено практичну апробацію. Нижче наведено покрокову методику експлуатації головних функціональних модулів системи (Use Case Scenarios).

2.7.1 Сценарій ручного підбору музичного контенту

Цей сценарій описує базовий алгоритм взаємодії користувача з системою для отримання музичних рекомендацій за допомогою жорстко заданих фільтрів (рисунок 2.15).

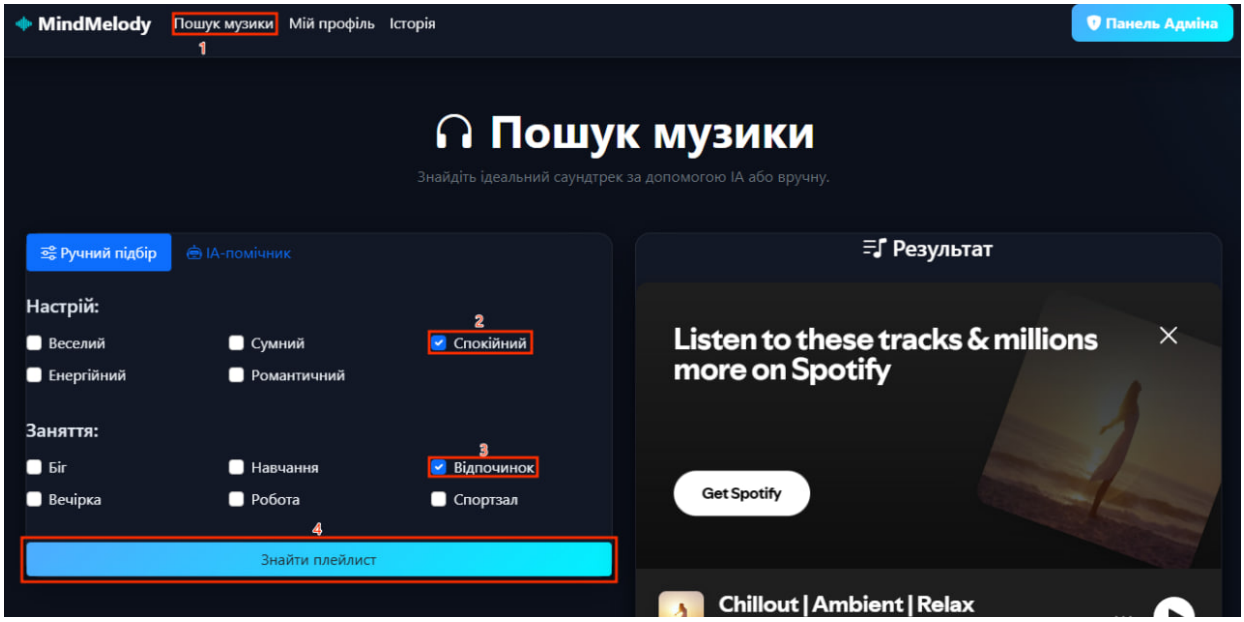

Рисунок 2.15 - Алгоритм ручного підбору плейлиста

Порядок дій користувача:

1. Навігація до головного робочого простору шляхом натискання на пункт «Пошук музики» у верхньому навігаційному меню.

2. Визначення поточного емоційного стану в блоці «Настрій» (на прикладі обрано чекбокс «Спокійний»).

3. Уточнення контексту або виду діяльності у блоці «Заняття» (обрано чекбокс «Відпочинок»). Користувач може комбінувати кілька чекбоксів для складніших запитів.

4. Ініціалізація пошукового алгоритму шляхом натискання на головну цільову кнопку «Знайти плейлист». Після цього система відправляє запит на сервер, взаємодіє зі Spotify API та генерує віджет плеєра у правій панелі.

2.7.2 Сценарій інтелектуального підбору за допомогою IA

Сценарій демонструє роботу інноваційного модуля системи - семантичного пошуку музики на основі неструктурованого природномовного запиту (рисунок 2.16).

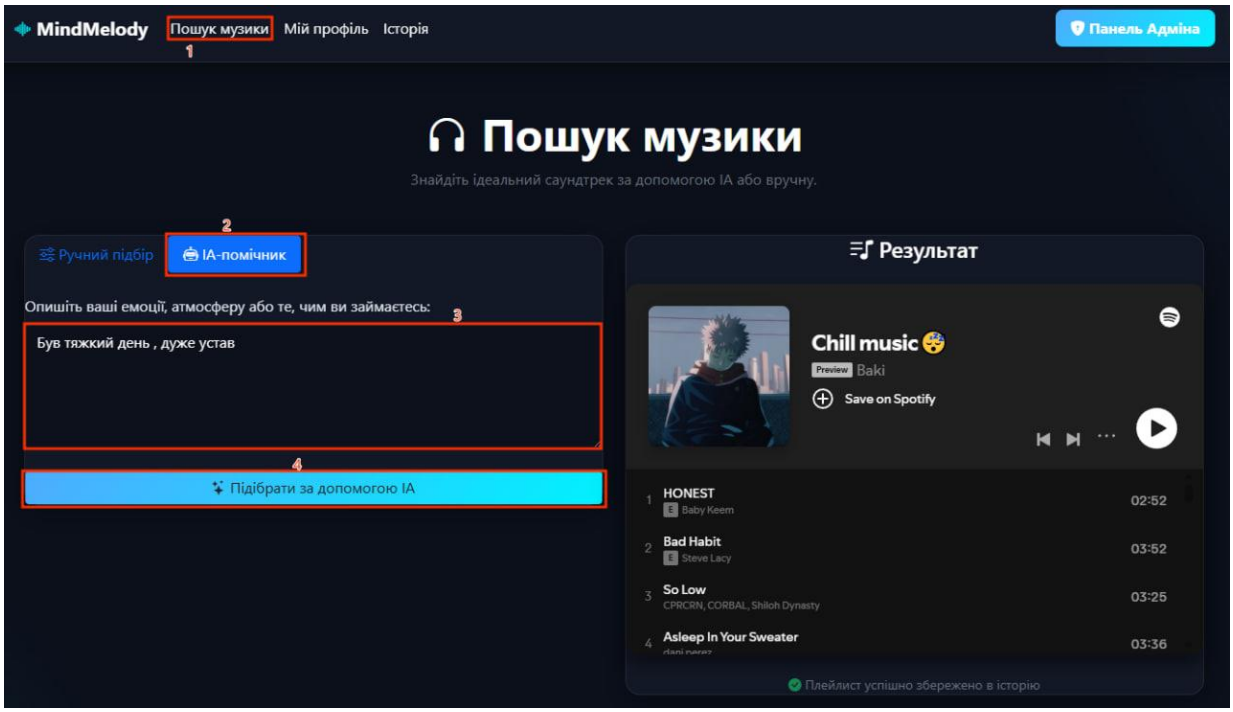

Рисунок 2.16 - Алгоритм використання інтелектуального помічника

Порядок дій користувача:

1. Перехід до розділу «Пошук музики» через головне меню.

2. Перемикання інтерфейсу на режим нейромережі шляхом вибору вкладки «IA-помічник».

3. Введення текстового опису свого поточного стану, емоцій або думок у спеціальне поле (наприклад: "Був тяжкий день, дуже устав").

4. Запуск процесу генерації натисканням кнопки «Підібрати за допомогою ІА». Система передає текст до мовної моделі Gemini, яка аналізує семантику, витягує ключові наміри (intents) і самостійно знаходить релевантний плейлист для релаксації.

2.7.3 Сценарій управління особистим профілем

Даний алгоритм описує можливості користувача щодо персоналізації свого облікового запису в системі (рисунок 2.17).

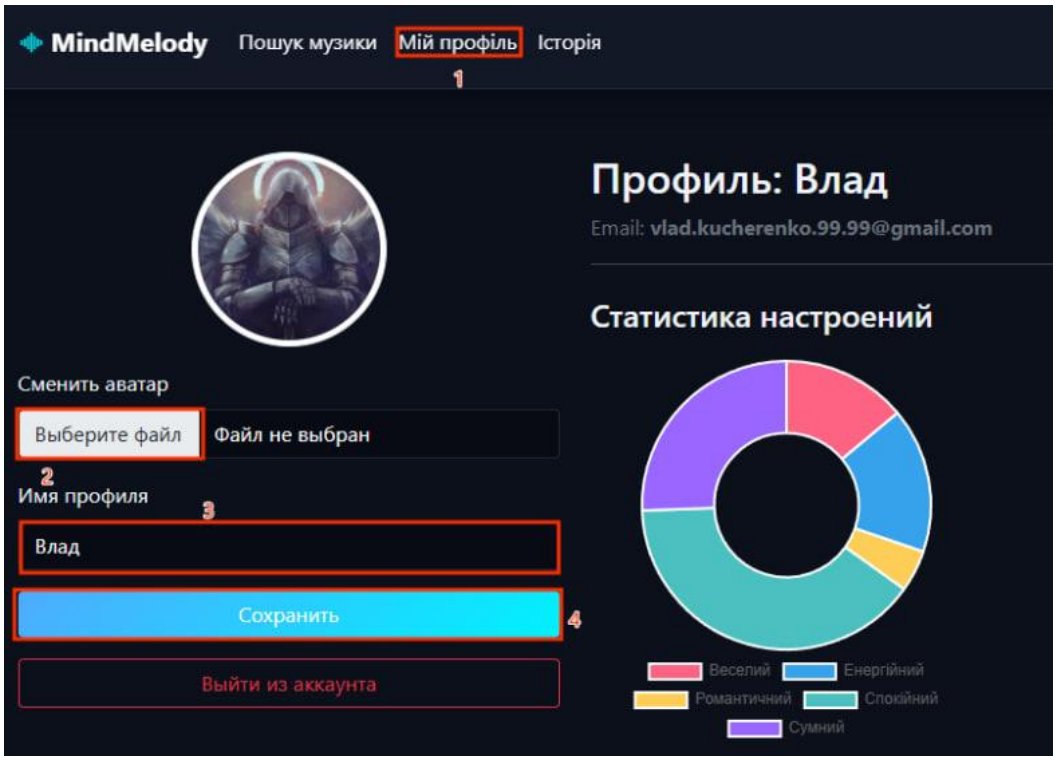

Рисунок 2.17 - Порядок оновлення даних особистого профілю

Порядок дій користувача:

1. Відкриття персонального дашборду натискання на пункт «Мій профіль» у навігаційній панелі.

2. Завантаження або зміна графічного представлення (аватара). При натисканні на кнопку «Выберите файл» відкривається системне діалогове вікно вибору зображення.

3. Редагування імені користувача (Display Name) у текстовому полі «Имя профиля».

4. Збереження внесених змін шляхом натискання на кнопку «Сохранить». Після цього дані оновлюються в базі даних (MSSQL), а сторінка перевантажується з новими параметрами профілю.

2.7.4 Сценарій роботи з історією та застосування фільтрів

Цей сценарій демонструє можливості підсистеми довгострокового зберігання даних і зручного пошуку серед раніше збережених плейлистів (рисунок 2.18).

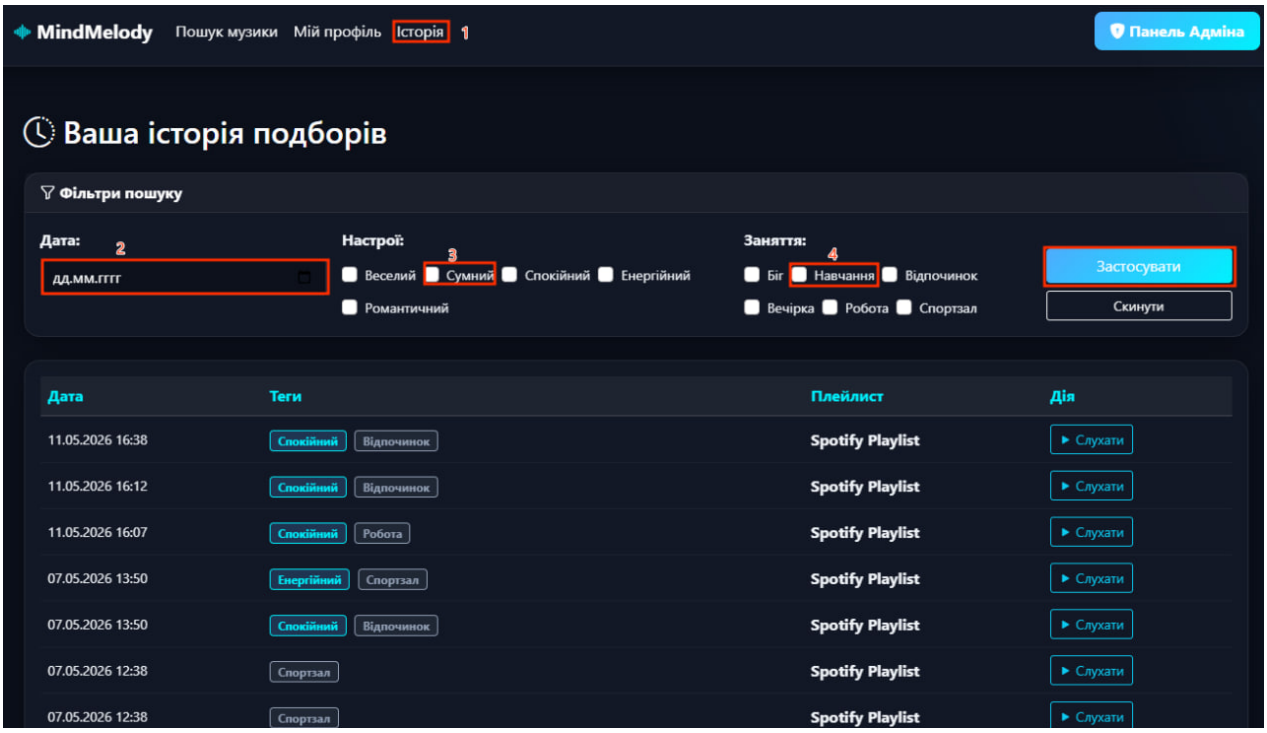

Рисунок 2.18 - Алгоритм пошуку записів в історії за допомогою фільтрів

Порядок дій користувача:

1. Перехід до персональної бібліотеки збережених підборів через пункт меню «Історія».

2. Звуження простору пошуку за часовим критерієм шляхом вибору конкретної дати у полі «Дата» (опціональний крок).

3. Застосування семантичного фільтра за настроєм (наприклад, вибір чекбоксу «Сумний»).

4. Застосування контекстного фільтра за видом діяльності (наприклад, вибір чекбоксу «Навчання»). Після вибору необхідних параметрів користувач натискає кнопку «Застосувати» (розташовану праворуч), і система виконує LINQ-запит до бази даних, залишаючи в таблиці лише ті плейлисти, які відповідають заданим критеріям.

2.8 Практична апробація модуля адміністрування та моніторингу

Для перевірки працездатності закритої частини вебдодатку, призначеної для управління контентом та користувачами, було проведено тестування панелі адміністратора (Admin Dashboard). Нижче наведено покрокові сценарії експлуатації головних інструментів управління системою.

2.8.1 Сценарій моніторингу глобальної статистики

Цей сценарій описує процес взаємодії адміністратора з аналітичним модулем системи для оцінки ключових показників ефективності (рисунок 2.19).

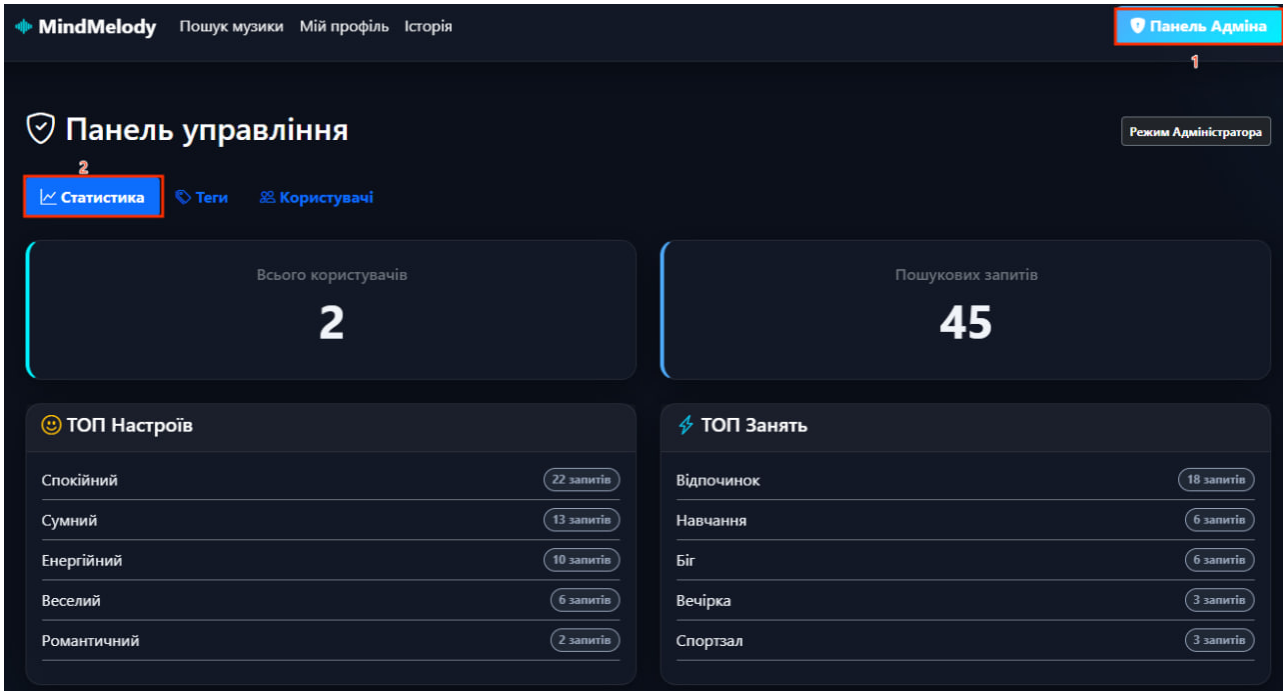

Рисунок 2.19 - Доступ до глобальної статистики у панелі управління

Порядок дій адміністратора:

1. Авторизація в системі з правами адміністратора та натискання на спеціалізовану кнопку «Панель Адміна» у верхньому навігаційному меню.

2. Перехід до вкладки «Статистика» (відкривається за замовчуванням). На цій вкладці адміністратор проводить візуальний моніторинг загальної кількості зареєстрованих користувачів, оброблених пошукових запитів, а також аналізує ТОП-рейтинги найпопулярніших настроїв та занять у системі.

2.8.2 Сценарій управління довідниками (тегами)

Даний алгоритм ілюструє процес динамічного розширення семантичної бази додатку шляхом додавання нових музичних тегів (рисунок 2.20).

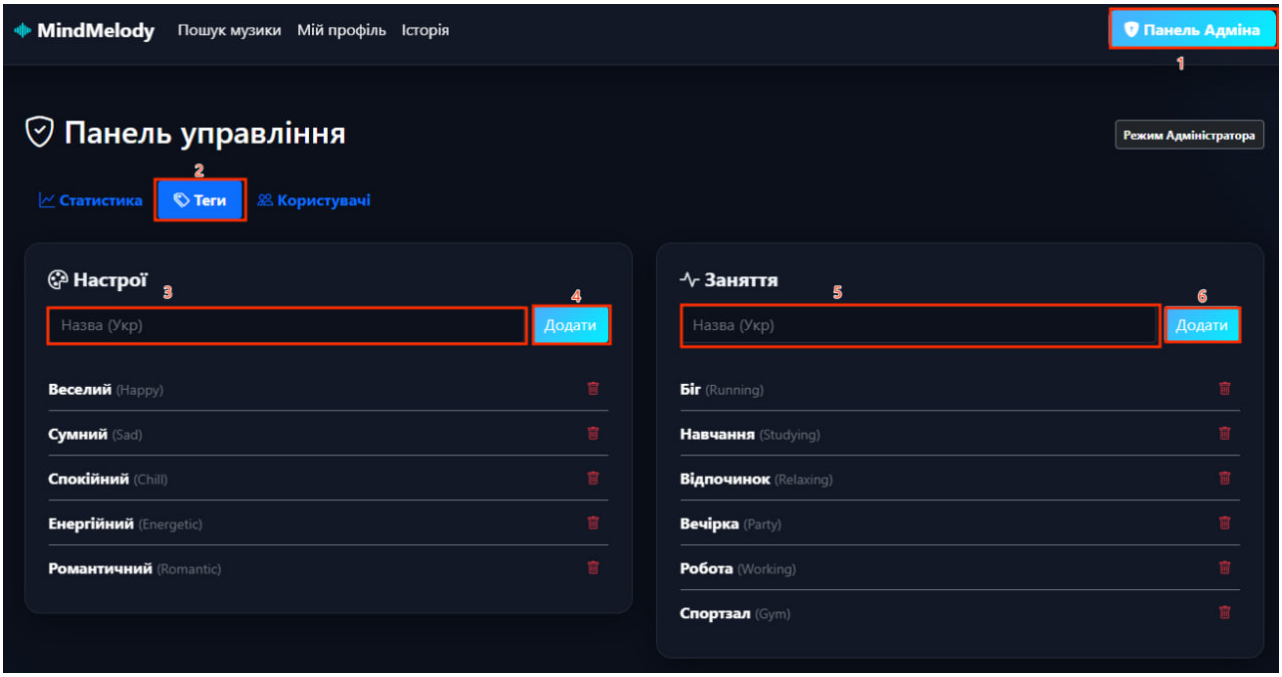

Рисунок 2.20 - Алгоритм додавання нових музичних тегів

Порядок дій адміністратора:

1. Вхід до закритого модуля управління через кнопку «Панель Адміна».

2. Вибір вкладки «Теги» для доступу до інтерфейсу редагування системних довідників.

3. Для додавання нової категорії до блоку «Настрої»: введення локалізованої назви (українською мовою) у відповідне текстове поле «Назва (Укр)».

4. Збереження нового тегу настрою у базі даних шляхом натискання кнопки «Додати».

5. Для розширення категорій у блоці «Заняття»: введення назви у відповідне текстове поле «Назва (Укр)» у правій панелі.

6. Підтвердження операції додавання заняття натисканням відповідної кнопки «Додати», після чого система миттєво оновлює списки без перезавантаження сторінки.

2.8.3. Сценарій управління обліковими записами користувачів

Останній сценарій описує можливості адміністратора вищого рівня щодо контролю доступу (RBAC) та управління життєвим циклом акаунтів (рисунок 2.21).

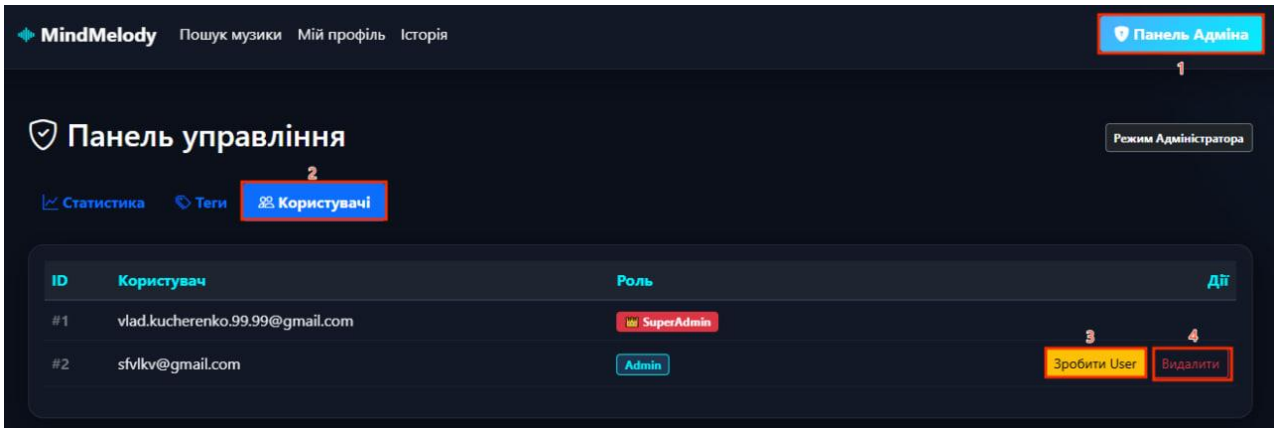

Рисунок 2.21 - Управління ролями та видалення користувачів

Порядок дій адміністратора:

1. Доступ до інструментів управління системою через кнопку «Панель Адміна».

2. Перемикання на вкладку «Користувачі», де завантажується таблиця з повним переліком зареєстрованих акаунтів, їхніми ідентифікаторами та поточними ролями.

3. Управління правами доступу: натискання кнопки «Зробити User» (для акаунтів з правами Admin) дозволяє понизити їхні привілеї до рівня звичайного користувача.

4. Управління життєвим циклом: натискання кнопки «Видалити» ініціює незворотний процес стирання профілю користувача з системи, що також супроводжується каскадним видаленням усієї його персональної історії пошуку з бази даних.

Висновки до розділу:

У другому розділі розроблено архітектурно-технологічні рішення для побудови інтелектуального веб-застосунку. Спроектовано структуру бази даних та схему взаємодії компонентів системи за допомогою діаграм класів і послідовностей. Обґрунтовано вибір сучасного технологічного стеку для клієнтської та серверної частин, а також інтеграцію із зовнішніми API для отримання музичних треків та їхніх метаданих. Описано алгоритми класифікації емоційного стану користувача та математичну логіку мапування визначених емоцій на акустичні властивості музики (параметри валентності та енергійності). Отримані проектні специфікації забезпечують надійне підґрунтя для подальшої програмної реалізації системи.

ВИСНОВКИ

Виконана кваліфікаційна робота присвячена розробці інтелектуальної інформаційної системи «MindMelody», яка призначена для персоналізованого підбору музичного контенту на основі емоційного стану користувача із застосуванням великих мовних моделей. У ході дослідження та реалізації проєкту було вирішено низку важливих завдань, що охоплюють як теоретичне обґрунтування підходів, так і технічне втілення системи.

Проведений аналіз предметної області виявив проблему «інформаційного перенавантаження» у сучасних музичних сервісах, де користувач витрачає надмірну кількість часу на пошук релевантного контенту. Огляд існуючих підходів – від контентної фільтрації до гібридних моделей – дозволив обґрунтувати вибір інноваційної архітектури на основі великих мовних моделей (LLM). Використання Google Gemini API як ядра системи забезпечило можливість семантичного аналізу неструктурованих текстових запитів природною мовою, що є значним кроком уперед порівняно з традиційними системами на основі жорстких тегів.

У ході технічної реалізації було обрано та впроваджено сучасний стек технологій: фреймворк ASP.NET Core (MVC) для побудови серверної логіки, реляційну базу даних MSSQL для надійного збереження даних, а також інтеграцію зі Spotify Web API для доступу до глобального музичного каталогу. Розроблена архітектура бази даних забезпечує цілісність інформації про профілі користувачів, їхні вподобання та розширену історію пошукових запитів.

Особливу увагу приділено проєктуванню клієнтського інтерфейсу. Реалізований дизайн у стилі Glassmorphism відповідає сучасним вимогам UX/UI, забезпечуючи інтуїтивну навігацію між модулями ручного та інтелектуального підбору. Впровадження візуалізації даних (статистика настроїв у профілі) та розширеної панелі адміністратора з глобальною аналітикою дозволяє не лише користуватися сервісом, а й ефективно моніторити тренди користувацької поведінки.

Значним результатом роботи є успішна інтеграція підсистеми автентифікації на базі ASP.NET Core Identity, що гарантує безпеку персональних даних та дозволяє реалізувати рольову модель доступу (RBAC). Практична апробація системи підтвердила її стабільність та високу точність підбору плейлистів: використання ШІ для перекладу людських емоцій у технічні параметри Spotify дозволило досягти мінімізації семантичної відстані між запитом та результатом.

Отже, розроблений вебзастосунок має високе прикладне значення для індустрії розваг та цифрового здоров’я, оскільки сприяє швидкій емоційній регуляції користувачів через музику. Перспективи подальшого розвитку проєкту полягають у розширенні підтримки різних стрімінгових платформ, впровадженні складніших алгоритмів машинного навчання для передбачення вподобань та адаптації системи для групових рекомендацій.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. ASP.NET Core MVC documentation. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview (дата звернення: 12.04.2025).

2. Bill Gates. The Age of AI has begun. GatesNotes. URL: https://www.gatesnotes.com/The-Age-of-AI-has-begun (дата звернення: 09.05.2026).

3. Entity Framework Core. Microsoft Learn. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 10.05.2026).

4. Gemini API Documentation. Google for Developers. URL: https://ai.google.dev/docs (дата звернення: 20.04.2025).

5. Hybrid Recommender Systems: A Survey of State-of-the-Art / Robin Burke та ін. User Modeling and User-Adapted Interaction. 2023. Т. 12, № 4. С. 373-404.

6. Music Information Retrieval: Recent Developments and Applications / Meinard Müller та ін. Foundations and Trends in Information Retrieval. 2024. Т. 16, № 1. URL: https://doi.org/10.1561/1500000042 (дата звернення: 05.06.2026).

7. OAuth 2.0 Authorization Framework for Spotify. Spotify for Developers. URL: https://developer.spotify.com/documentation/general/guides/authorization/ (дата звернення: 06.05.2026).

8. Prompt Engineering Guide. DAIR.AI. URL: https://www.promptingguide.ai/ (дата звернення: 06.05.2026).

9. Ricci F., Rokach L., Shapira B. Recommender Systems Handbook. 3rd ed. Springer, 2022. 1240 p.

10. Spotify Web API Reference. Spotify for Developers. URL: https://developer.spotify.com/documentation/web-api/reference/ (дата звернення: 06.05.2026).

11. SQL Server Documentation. Microsoft Learn. URL: https://learn.microsoft.com/en-us/sql/sql-server/ (дата звернення: 07.05.2026).

12. The Evolution of Content-Based Filtering in the Era of LLMs / Ahmed El-Kishky та ін. Proceedings of the VLDB Endowment. 2024. Vol. 17, no. 5. P. 1120-1132.

13. Understanding Audio Embeddings for Music Similarity / Thomas L. Bertin-Mahieux та ін. Journal of New Music Research. 2025. С. 45-52.

14. Valence and Energy in Music Emotion Recognition / Juha S. Park та ін. IEEE Transactions on Affective Computing. 2024. Т. 15, № 2. С. 210-225.

15. What is Bootstrap? Overview and Installation. GetBootstrap. URL: https://getbootstrap.com/docs/5.3/getting-started/introduction/ (дата звернення: 08.06.2026).

16. Моркун Н. В., Завсегдашня І. В. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для студентів спеціальності 122 “Комп’ютерні науки”. Кривий Ріг : Видавничий центр КНУ, 2021. 50 с.

17. ДСТУ 3008:2015. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).

18. ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання. Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).

19. ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень в українській мові. Загальні вимоги та правила. Київ, ДП «УкрННЦ», 2013. 23 с. (Інформація та документація).

20. ДСТУ ISO 9001:2015. Системи управління якістю. Вимоги. Київ, ДП «УкрННЦ», 2016. 22 с. (Інформація та документація).

ДОДАТОК А

Лістинг основної частини програмного коду розширення

Лістинг А.1 – Контекст бази даних та основні моделі

```
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Дипломна_робота.Data
{
   public class ApplicationDbContext : DbContext
   {
      public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
         : base(options)
      {
      }

      public DbSet<User> Users { get; set; }
      public DbSet<Role> Roles { get; set; }
      public DbSet<UserRole> UserRoles { get; set; }
      public DbSet<Mood> Moods { get; set; }
      public DbSet<Activity> Activities { get; set; }
      public DbSet<Playlist> Playlists { get; set; }
      public DbSet<SearchHistory> SearchHistories { get; set; }
      public DbSet<SearchHistoryMood> SearchHistoryMoods { get; set; }
      public DbSet<SearchHistoryActivity> SearchHistoryActivities { get; set; }

      protected override void OnModelCreating(ModelBuilder modelBuilder)
      {
         base.OnModelCreating(modelBuilder);

         modelBuilder.Entity<UserRole>()
            .HasKey(x => new { x.UserId, x.RoleId });

         modelBuilder.Entity<UserRole>()
            .HasOne(ur => ur.User)
            .WithMany(u => u.UserRoles)
            .HasForeignKey(ur => ur.UserId);

         modelBuilder.Entity<UserRole>()
            .HasOne(ur => ur.Role)
            .WithMany(r => r.UserRoles)
            .HasForeignKey(ur => ur.RoleId);

         modelBuilder.Entity<SearchHistoryMood>()
            .HasKey(x => new { x.SearchHistoryId, x.MoodId });

         modelBuilder.Entity<SearchHistoryActivity>()
            .HasKey(x => new { x.SearchHistoryId, x.ActivityId });
```

```
        }
    }
}

namespace Дипломна_робота.Models
{
    public class User
    {
        public int Id { get; set; }
        public string Email { get; set; }
        public string PasswordHash { get; set; }
        public string? DisplayName { get; set; }
        public string? AvatarUrl { get; set; }
        public ICollection<SearchHistory> SearchHistories { get; set; }
        public ICollection<UserRole> UserRoles { get; set; }
    }

    public class Mood
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string EnglishName { get; set; }
        public List<SearchHistoryMood> SearchHistoryMoods { get; set; } = new();
    }

    public class Activity
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string EnglishName { get; set; }
        public List<SearchHistoryActivity> SearchHistoryActivities { get; set; } = new();
    }

    public class SearchHistory
    {
        public int Id { get; set; }
        public DateTime Date { get; set; }
        public int UserId { get; set; }
        public User User { get; set; }
        public int PlaylistId { get; set; }
        public Playlist Playlist { get; set; }
        public ICollection<SearchHistoryMood> Moods { get; set; }
        public ICollection<SearchHistoryActivity> Activities { get; set; }
    }
}
```

Лістинг А.2 – Сервіс інтеграції зі штучним інтелектом Gemini

```
using System.Text;
using System.Text.Json;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Caching.Memory;

namespace Дипломна_робота.Services
{
   public class GeminiService
   {
      private readonly string _apiKey;
      private readonly HttpClient _httpClient;
      private readonly ILogger<GeminiService> _logger;
      private readonly IMemoryCache _cache;
      private List<string>? _cachedModelsQueue = null;

      public GeminiService(IConfiguration configuration, ILogger<GeminiService> logger,
IMemoryCache cache)
      {
         _apiKey = configuration["Gemini:ApiKey"] ?? "";
         _httpClient = new HttpClient();
         _logger = logger;
         _cache = cache;
      }

      public class AiResponse
      {
         public List<string> Moods { get; set; } = new List<string>();
         public List<string> Activities { get; set; } = new List<string>();
      }

      private async Task<List<string>> GetAvailableModelsQueueAsync()
      {
         if (_cachedModelsQueue != null && _cachedModelsQueue.Any())
            return _cachedModelsQueue;

         var modelsQueue = new List<string>();

         try
         {
            var                                    listUrl                                    =
$"[https://generativelanguage.googleapis.com/v1beta/models?key=](https://generativelanguage.goo
gleapis.com/v1beta/models?key=){_apiKey}";
            var response = await _httpClient.GetAsync(listUrl);

            if (response.IsSuccessStatusCode)
            {
               var jsonString = await response.Content.ReadAsStringAsync();
               using JsonDocument doc = JsonDocument.Parse(jsonString);
               var models = doc.RootElement.GetProperty("models");

               foreach (var model in models.EnumerateArray())
               {
                  var name = model.GetProperty("name").GetString();
                  var methods = model.GetProperty("supportedGenerationMethods");
```

```
                bool isSupported = false;
                foreach (var method in methods.EnumerateArray())
                {
                    if (method.GetString() == "generateContent") { isSupported = true; break; }
                }

                if (isSupported && name != null &&
                    !name.Contains("vision") &&
                    !name.Contains("embedding") &&
                    !name.Contains("tts") &&
                    !name.Contains("audio"))
                {
                    modelsQueue.Add(name.Replace("models/", ""));
                }
            }
        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Помилка збору списку моделей: {Message}", ex.Message);
        }

        _cachedModelsQueue = modelsQueue
            .OrderByDescending(m => m.Contains("flash"))
            .ThenByDescending(m => m.Contains("pro"))
            .ToList();

        if (!_cachedModelsQueue.Any())
        {
            _cachedModelsQueue.Add("gemini-1.5-flash");
        }

        return _cachedModelsQueue;
    }

    public async Task<AiResponse?> AnalyzePromptAsync(string userPrompt, List<string> availableMoods, List<string> availableActivities)
    {
        if (string.IsNullOrEmpty(_apiKey))
        {
            return null;
        }

        string cacheKey = $"GeminiSearch_{userPrompt.Trim().ToLower()}";

        if (_cache.TryGetValue(cacheKey, out AiResponse? cachedResponse))
        {
            return cachedResponse;
        }

        var modelsToTry = await GetAvailableModelsQueueAsync();
```

```
var systemInstruction = $@"
   Ты — умный и эмпатичный музыкальный диджей и психолог.
   Пользователь будет описывать свое состояние, прошедший день или желания своими словами.
   Твоя задача — проанализировать скрытый смысл, эмоции и контекст, а затем подобрать наиболее подходящие музыкальные теги ТОЛЬКО из предоставленных списков.

   Доступные списки:
   Moods: {string.Join(", ", availableMoods)}
   Activities: {string.Join(", ", availableActivities)}

   ПРАВИЛА:
   1. Включай логику и эмпатию. Связывай абстрактный текст с тегами.
   2. НЕ придумывай свои теги! Выбирай только из предоставленных списков.
   3. Выбери от 1 до 2 настроений (Moods) и от 1 до 2 занятий (Activities).
   4. Верни ответ СТРОГО в формате JSON без пояснений, без markdown-кавычек.

   Формат:
   {{ ""Moods"": [""tag""], ""Activities"": [""tag""] }}";

var requestBody = new
{
   contents = new[]
   {
      new { parts = new[] { new { text = $"{systemInstruction}\n\nТекст пользователя: {userPrompt}" } } }
   }
};

var content = new StringContent(JsonSerializer.Serialize(requestBody), Encoding.UTF8, "application/json");

foreach (var modelName in modelsToTry)
{
   var url = $"[https://generativelanguage.googleapis.com/v1beta/models/](https://generativelanguage.googleapis.com/v1beta/models/){modelName}:generateContent?key={_apiKey}";

   try
   {
      var response = await _httpClient.PostAsync(url, content);

      if (response.IsSuccessStatusCode)
      {
         var jsonString = await response.Content.ReadAsStringAsync();
         using JsonDocument doc = JsonDocument.Parse(jsonString);

         var rawText = doc.RootElement
            .GetProperty("candidates")[0]
            .GetProperty("content")
            .GetProperty("parts")[0]
```

```
                    .GetProperty("text").GetString();

                if (!string.IsNullOrEmpty(rawText))
                {
                    var cleanJson = rawText.Replace("```json", "").Replace("```", "").Trim();
                    var aiResponse = JsonSerializer.Deserialize<AiResponse>(cleanJson, new JsonSerializerOptions { PropertyNameCaseInsensitive = true });

                    if (aiResponse != null)
                    {
                        _cache.Set(cacheKey, aiResponse, TimeSpan.FromHours(1));
                    }

                    return aiResponse;
                }
            }
            else
            {
                continue;
            }
        }
        catch (Exception)
        {
            continue;
        }
    }

    return null;
}

public async Task<string> TranslateTagToEnglishAsync(string ukrainianTag)
{
    if (string.IsNullOrEmpty(_apiKey)) return ukrainianTag;

    string cacheKey = $"Translate_{ukrainianTag.Trim().ToLower()}";
    if (_cache.TryGetValue(cacheKey, out string? cachedTranslation) && cachedTranslation != null)
    {
        return cachedTranslation;
    }

    var modelsToTry = await GetAvailableModelsQueueAsync();

    var systemInstruction = "Translate the following Ukrainian mood or activity tag into a single English word or short phrase suitable for a Spotify search query. Return ONLY the English text, nothing else. No markdown, no quotes.";

    var requestBody = new
    {
        contents = new[]
        {
```

```
            new { parts = new[] { new { text = $"{systemInstruction}\n\nTag: {ukrainianTag}" } }
}
        }
    };

    var content = new StringContent(JsonSerializer.Serialize(requestBody), Encoding.UTF8,
"application/json");

    foreach (var modelName in modelsToTry)
    {
        var url =
$"[https://generativelanguage.googleapis.com/v1beta/models/](https://generativelanguage.googleapi
s.com/v1beta/models/){modelName}:generateContent?key={_apiKey}";

        try
        {
            var response = await _httpClient.PostAsync(url, content);

            if (response.IsSuccessStatusCode)
            {
                var jsonString = await response.Content.ReadAsStringAsync();
                using JsonDocument doc = JsonDocument.Parse(jsonString);

                var englishText = doc.RootElement
                    .GetProperty("candidates")[0]
                    .GetProperty("content")
                    .GetProperty("parts")[0]
                    .GetProperty("text").GetString();

                var finalResult = englishText?.Trim() ?? ukrainianTag;

                _cache.Set(cacheKey, finalResult, TimeSpan.FromDays(1));

                return finalResult;
            }

            continue;
        }
        catch (Exception)
        {
            continue;
        }
    }

    return ukrainianTag;
  }
}
```

Лістинг А.3 – Сервіс взаємодії зі Spotify API

```
using SpotifyAPI.Web;

namespace Дипломна_робота.Services
{
   public class SpotifyService
   {
      private readonly string _clientId = "a4fa8056a7914a91b564cecccb75879b";
      private readonly string _clientSecret = "93d30164588d45b7a45abecd547b4645";

      public async Task<string> SearchPlaylistAsync(string query)
      {
         if (string.IsNullOrEmpty(_clientId) || string.IsNullOrEmpty(_clientSecret)) return null;

         var config = SpotifyClientConfig.CreateDefault();
         var request = new ClientCredentialsRequest(_clientId, _clientSecret);
         var response = await new OAuthClient(config).RequestToken(request);
         var spotify = new SpotifyClient(config.WithToken(response.AccessToken));

         var searchRequest = new SearchRequest(SearchRequest.Types.Playlist, query)
         {
            Limit = 10,
            Market = "UA"
         };

         var searchResponse = await spotify.Search.Item(searchRequest);

         if (searchResponse?.Playlists?.Items != null)
         {
            var validItems = searchResponse.Playlists.Items.Where(p => p != null).ToList();

            if (validItems.Any())
            {
               var random = new Random();
               int randomIndex = random.Next(validItems.Count);

               return validItems[randomIndex].Id;
            }
         }

         return null;
      }
   }
}
```

Лістинг А.4 – Головний контролер логіки пошуку музики

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Security.Claims;
```

```
using Дипломна_робота.Models;
using Дипломна_робота.Services;

namespace Дипломна_робота.Pages
{
   [Authorize]
   public class IndexModel : PageModel
   {
      private readonly SpotifyService _spotifyService;
      private readonly GeminiService _aiService;
      private readonly MusicSearchService _searchService;

      public IndexModel(SpotifyService spotifyService, GeminiService aiService,
MusicSearchService searchService)
      {
         _spotifyService = spotifyService;
         _aiService = aiService;
         _searchService = searchService;
      }

      [BindProperty]
      public string? ActiveTab { get; set; } = "manual";

      [BindProperty]
      public List<int> SelectedMoods { get; set; } = new List<int>();

      [BindProperty]
      public List<int> SelectedActivities { get; set; } = new List<int>();

      [BindProperty]
      public string? UserPrompt { get; set; }

      public List<Mood> AvailableMoods { get; set; } = new List<Mood>();
      public List<Activity> AvailableActivities { get; set; } = new List<Activity>();

      public string? FoundPlaylistId { get; set; }

      public async Task OnGetAsync()
      {
         AvailableMoods = await _searchService.GetAvailableMoodsAsync();
         AvailableActivities = await _searchService.GetAvailableActivitiesAsync();
      }

      public async Task<IActionResult> OnPostAiSearchAsync()
      {
         ModelState.Remove("ActiveTab");
         ActiveTab = "ai";

         AvailableMoods = await _searchService.GetAvailableMoodsAsync();
         AvailableActivities = await _searchService.GetAvailableActivitiesAsync();

         if (string.IsNullOrWhiteSpace(UserPrompt))
```

```
        {
            ModelState.AddModelError(string.Empty, "Будь ласка, опишіть свій стан.");
            return Page();
        }

        var moodEnglishNames = AvailableMoods.Select(m => m.EnglishName).ToList();
        var activityEnglishNames = AvailableActivities.Select(a => a.EnglishName).ToList();

        var aiResult = await _aiService.AnalyzePromptAsync(UserPrompt, moodEnglishNames, activityEnglishNames);

        if (aiResult != null && (aiResult.Moods.Any() || aiResult.Activities.Any()))
        {
            SelectedMoods = AvailableMoods.Where(m => aiResult.Moods.Contains(m.EnglishName)).Select(m => m.Id).ToList();
            SelectedActivities = AvailableActivities.Where(a => aiResult.Activities.Contains(a.EnglishName)).Select(a => a.Id).ToList();

            string query = string.Join(" ", aiResult.Moods.Concat(aiResult.Activities));
            FoundPlaylistId = await _spotifyService.SearchPlaylistAsync(query);

            var userIdString = User.FindFirstValue(ClaimTypes.NameIdentifier);
            if (!string.IsNullOrEmpty(userIdString) && int.TryParse(userIdString, out int userId))
            {
                await _searchService.SaveSearchHistoryAsync(userId, FoundPlaylistId, SelectedMoods, SelectedActivities);
            }

            if (string.IsNullOrEmpty(FoundPlaylistId))
            {
                ModelState.AddModelError(string.Empty, "ІІ підібрав теги, але Spotify не знайшов відповідний плейлист.");
            }
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Не вдалося розпізнати настрій. Спробуйте описати стан інакше.");
        }

        return Page();
    }

    public async Task<IActionResult> OnPostManualSearchAsync()
    {
        ModelState.Remove("ActiveTab");
        ActiveTab = "manual";

        AvailableMoods = await _searchService.GetAvailableMoodsAsync();
        AvailableActivities = await _searchService.GetAvailableActivitiesAsync();

        if (!SelectedMoods.Any() && !SelectedActivities.Any())
```

```
        {
            ModelState.AddModelError(string.Empty, "Оберіть хоча б один параметр для пошуку.");
            return Page();
        }

        var moods = AvailableMoods.Where(m => SelectedMoods.Contains(m.Id)).ToList();
        var activities = AvailableActivities.Where(a => SelectedActivities.Contains(a.Id)).ToList();

        var moodEnglishNames = moods.Select(m => m.EnglishName).ToList();
        var activityEnglishNames = activities.Select(a => a.EnglishName).ToList();

        if (moodEnglishNames.Contains("Happy") && moodEnglishNames.Contains("Sad"))
        {
            ModelState.AddModelError(string.Empty, "Емоційні гойдалки: не можна обрати Веселий та Сумний одночасно.");
            return Page();
        }
        if (moodEnglishNames.Contains("Energetic") && moodEnglishNames.Contains("Chill"))
        {
            ModelState.AddModelError(string.Empty, "Суперечливий вайб: Енергійний та Спокійний складно поєднати.");
            return Page();
        }

        string query = string.Join(" ", moodEnglishNames.Concat(activityEnglishNames));
        FoundPlaylistId = await _spotifyService.SearchPlaylistAsync(query);

        var userIdString = User.FindFirstValue(ClaimTypes.NameIdentifier);
        if (!string.IsNullOrEmpty(userIdString) && int.TryParse(userIdString, out int userId))
        {
            await _searchService.SaveSearchHistoryAsync(userId, FoundPlaylistId, SelectedMoods, SelectedActivities);
        }

        if (string.IsNullOrEmpty(FoundPlaylistId))
        {
            ModelState.AddModelError(string.Empty, "Плейлисти не знайдено. Спробуйте інші параметри.");
        }

        return Page();
    }
  }
}
```

Лістинг А.5 – Контролер обробки панелі адміністратора

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
```

```
using Microsoft.EntityFrameworkCore;
using Дипломна_робота.Data;
using Дипломна_робота.Models;
using Дипломна_робота.Services;
using Microsoft.Extensions.Configuration;

namespace Дипломна_робота.Pages.Admin
{
   [Authorize(Roles = "Admin")]
   public class DashboardModel : PageModel
   {
      private readonly ApplicationDbContext _context;
      private readonly GeminiService _aiService;
      private readonly IConfiguration _configuration;

      public string MainAdminEmail { get; private set; }

      public DashboardModel(ApplicationDbContext context, GeminiService aiService,
IConfiguration configuration)
      {
         _context = context;
         _aiService = aiService;
         _configuration = configuration;
         MainAdminEmail = _configuration["AdminSettings:SuperAdminEmail"] ?? "";
      }

      public int TotalUsers { get; set; }
      public int TotalSearches { get; set; }

      public List<User> AllUsers { get; set; } = new();
      public List<Mood> AllMoods { get; set; } = new();
      public List<Activity> AllActivities { get; set; } = new();

      public Dictionary<string, int> PopularMoods { get; set; } = new();
      public Dictionary<string, int> PopularActivities { get; set; } = new();

      [BindProperty]
      public string NewTagName { get; set; }

      public async Task OnGetAsync()
      {
         await LoadDataAsync();
      }

      private async Task LoadDataAsync()
      {
         TotalUsers = await _context.Users.CountAsync();
         TotalSearches = await _context.SearchHistories.CountAsync();

         AllUsers = await _context.Users
            .Include(u => u.UserRoles)
               .ThenInclude(ur => ur.Role)
```

```
            .ToListAsync();

        AllMoods = await _context.Moods.ToListAsync();
        AllActivities = await _context.Activities.ToListAsync();

        PopularMoods = await _context.SearchHistoryMoods
            .Include(shm => shm.Mood).GroupBy(shm => shm.Mood.Name)
            .Select(g => new { Name = g.Key, Count = g.Count() })
            .OrderByDescending(x => x.Count).Take(5).ToDictionaryAsync(x => x.Name, x => x.Count);

        PopularActivities = await _context.SearchHistoryActivities
            .Include(sha => sha.Activity).GroupBy(sha => sha.Activity.Name)
            .Select(g => new { Name = g.Key, Count = g.Count() })
            .OrderByDescending(x => x.Count).Take(5).ToDictionaryAsync(x => x.Name, x => x.Count);
    }

    public async Task<IActionResult> OnPostToggleRoleAsync([FromForm] int id)
    {
        if (id <= 0)
            return new JsonResult(new { success = false, message = "Помилка: Сервер не отримав правильний ID користувача." });

        if (User.Identity?.Name?.Trim().ToLower() != MainAdminEmail.Trim().ToLower())
            return new JsonResult(new { success = false, message = "Помилка: Ви не маєте прав SuperAdmin для цієї дії." });

        var targetUser = await _context.Users
            .Include(u => u.UserRoles).ThenInclude(ur => ur.Role)
            .FirstOrDefaultAsync(u => u.Id == id);

        if (targetUser == null)
            return new JsonResult(new { success = false, message = $"Помилка: Користувача з ID #{id} не знайдено." });

        if (targetUser.Email.Trim().ToLower() == MainAdminEmail.Trim().ToLower())
            return new JsonResult(new { success = false, message = "Помилка: Не можна змінювати права самому собі." });

        try
        {
            var adminRole = await _context.Roles.FirstOrDefaultAsync(r => r.Name == "Admin");
            if (adminRole == null)
            {
                adminRole = new Role { Name = "Admin" };
                _context.Roles.Add(adminRole);
                await _context.SaveChangesAsync();
            }

            var existingUserRole = targetUser.UserRoles.FirstOrDefault(ur => ur.RoleId == adminRole.Id);
```

```
            bool isNowAdmin = false;

            if (existingUserRole != null)
            {
                _context.UserRoles.Remove(existingUserRole);
                isNowAdmin = false;
            }
            else
            {
                _context.UserRoles.Add(new UserRole { UserId = targetUser.Id, RoleId = adminRole.Id
});
                isNowAdmin = true;
            }

            await _context.SaveChangesAsync();
            return new JsonResult(new { success = true, isAdmin = isNowAdmin });
        }
        catch (Exception ex)
        {
            return new JsonResult(new { success = false, message = $"Критична помилка БД:
{ex.Message}" });
        }
    }

    public async Task<IActionResult> OnPostAddMoodAsync()
    {
        if (!string.IsNullOrWhiteSpace(NewTagName))
        {
            string translatedName = await _aiService.TranslateTagToEnglishAsync(NewTagName);
            _context.Moods.Add(new Mood { Name = NewTagName, EnglishName = translatedName
});
            await _context.SaveChangesAsync();
        }
        return RedirectToPage();
    }

    public async Task<IActionResult> OnPostAddActivityAsync()
    {
        if (!string.IsNullOrWhiteSpace(NewTagName))
        {
            string translatedName = await _aiService.TranslateTagToEnglishAsync(NewTagName);
            _context.Activities.Add(new Activity { Name = NewTagName, EnglishName =
translatedName });
            await _context.SaveChangesAsync();
        }
        return RedirectToPage();
    }

    public async Task<IActionResult> OnPostDeleteMoodAsync(int id)
    {
        var mood = await _context.Moods.FindAsync(id);
        if (mood != null) { _context.Moods.Remove(mood); await _context.SaveChangesAsync(); }
```

```
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostDeleteActivityAsync(int id)
        {
            var activity = await _context.Activities.FindAsync(id);
            if (activity != null) { _context.Activities.Remove(activity); await
_context.SaveChangesAsync(); }
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostDeleteUserAsync(int id)
        {
            var currentUserEmail = User.Identity?.Name?.Trim().ToLower();
            var targetUser = await _context.Users
                .Include(u => u.UserRoles).ThenInclude(ur => ur.Role)
                .FirstOrDefaultAsync(u => u.Id == id);

            if (targetUser == null) return RedirectToPage();

            bool isTargetAdmin = targetUser.UserRoles.Any(ur => ur.Role.Name == "Admin");

            if (targetUser.Email.Trim().ToLower() == MainAdminEmail.Trim().ToLower()) return
RedirectToPage();
            if (isTargetAdmin && currentUserEmail != MainAdminEmail.Trim().ToLower()) return
RedirectToPage();

            _context.Users.Remove(targetUser);
            await _context.SaveChangesAsync();
            return RedirectToPage();
        }
    }
}
```

Лістинг А.6 – Контролер авторизації користувачів

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Дипломна_робота.Data;
using BCrypt.Net;
using System.Security.Claims;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace Дипломна_робота.Pages
{
    public class LoginModel : PageModel
    {
        private readonly ApplicationDbContext _context;
```

```
    private readonly IConfiguration _configuration;

    public LoginModel(ApplicationDbContext context, IConfiguration configuration)
    {
        _context = context;
        _configuration = configuration;
    }

    [BindProperty]
    public string Email { get; set; }

    [BindProperty]
    public string Password { get; set; }

    public void OnGet()
    {
        if (User.Identity != null && User.Identity.IsAuthenticated)
        {
            Response.Redirect("/");
        }
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var user = await _context.Users
            .Include(u => u.UserRoles)
                .ThenInclude(ur => ur.Role)
            .FirstOrDefaultAsync(u => u.Email == Email);

        if (user == null || !BCrypt.Net.BCrypt.Verify(Password, user.PasswordHash))
        {
            ModelState.AddModelError(string.Empty, "Невірний Email або пароль.");
            return Page();
        }

        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
            new Claim(ClaimTypes.Email, user.Email),
            new Claim(ClaimTypes.Name, user.Email)
        };

        string mainAdminEmail = _configuration["AdminSettings:SuperAdminEmail"] ?? "";

        bool hasAdminRole = user.UserRoles != null && user.UserRoles.Any(ur => ur.Role.Name
== "Admin");
```

```
        if (user.Email.Trim().ToLower() == mainAdminEmail.Trim().ToLower() || hasAdminRole)
        {
            claims.Add(new Claim(ClaimTypes.Role, "Admin"));
        }

        var identity = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme);
        var principal = new ClaimsPrincipal(identity);

        await HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, principal);

        return RedirectToPage("/Index");
      }
    }
}
```

Лістинг А.7 – Конфігурація та налаштування конвеєра застосунку

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.EntityFrameworkCore;
using System.Security.Claims;
using Дипломна_робота.Data;
using Дипломна_робота.Models;
using Дипломна_робота.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/Admin", "AdminOnly");
    options.Conventions.AllowAnonymousToPage("/Login");
});

builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/Login";
        options.AccessDeniedPath = "/AccessDenied";
    });
builder.Services.AddMemoryCache();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));
});

builder.Services.AddScoped<SpotifyService>();
```

```
builder.Services.AddScoped<GeminiService>();
builder.Services.AddScoped<MusicSearchService>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
   app.UseExceptionHandler("/Error");
   app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

using (var scope = app.Services.CreateScope())
{
   var db = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
   db.Database.EnsureCreated();

   if (!db.Moods.Any())
   {
      db.Moods.AddRange(
         new Mood { Name = "Веселий", EnglishName = "Happy" },
         new Mood { Name = "Сумний", EnglishName = "Sad" },
         new Mood { Name = "Спокійний", EnglishName = "Chill" },
         new Mood { Name = "Енергійний", EnglishName = "Energetic" },
         new Mood { Name = "Романтичний", EnglishName = "Romantic" }
      );
      db.SaveChanges();
   }

   if (!db.Activities.Any())
   {
      db.Activities.AddRange(
         new Activity { Name = "Біг", EnglishName = "Running" },
         new Activity { Name = "Навчання", EnglishName = "Studying" },
         new Activity { Name = "Відпочинок", EnglishName = "Relaxing" },
         new Activity { Name = "Вечірка", EnglishName = "Party" },
         new Activity { Name = "Робота", EnglishName = "Working" }
      );
      db.SaveChanges();
   }
}

app.Run();
```