НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2026 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: «Вебзастосунок геопросторового пошуку виступів музичних гуртів України з інтерактивною візуалізацією на карті»

Виконав: студент 4 курсу, групи ТР-23
Сенишин Назар Романович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент каф. ЦТЕ Микита ГОЛОВАКІН _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент професор каф. теплової та альтернативної енергетики, д.т.н., проф., Ольга ЧЕРНОУСЕНКО
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль асистент Артем МЕЛЬНИЧЕНКО_____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2026

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

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____Наталія АУШЕВА
(підпис)
“___” _____2026 р.

ЗАВДАННЯ
на дипломну роботу студенту

Сенишину Назару Романовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Вебзастосунок геопросторового пошуку виступів музичних гуртів України з інтерактивною візуалізацією на карті”
Науковий керівник Головакін Микита Андрійович, асистент каф. ЦТЕ
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від “28” травня 2026 року № 1992
2. Термін подання студентом роботи 04.06.2026
3. Вихідні дані до роботи мова програмування Python, фреймворк Django, СУБД PostgreSQL з розширенням PostGIS, середовище розробки Visual Studio Code, операційна система Windows 11.
4. Перелік питань, які потрібно розробити:

1) провести аналіз серед наявних аналогів

2) визначити головні функції вебзастосунку, що керує послугами та ресурсами

3) аргументувати вибрані інструменти, алгоритми та технології для реалізації вебсистеми

4) побудувати архітектуру програмного забезпечення та бази даних

5) створити прикладний програмний веб-інтерфейс

6) представити процес застосування веб-інтерфейсу

5. Орієнтований перелік ілюстративного матеріалу: схема структури бази даних, схема архітектури проєкту, скріншоти роботи користувачів з вебзастосунком.

6. Дата видачі завдання 15.09.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	07.10.25	
2.	Аналіз методів та засобів розв’язання задачі	20.04.26 – 26.04.26	
3.	Розробка архітектури та загальної структури системи	27.04.26 – 03.05.26	
4.	Розробка окремих підсистем	27.04.26 – 03.05.26	
5.	Програмна реалізація системи	04.05.26 – 10.05.26	
6.	Оформлення пояснювальної записки	11.05.25 – 22.05.26	
7.	Захист програмного забезпечення	14.05.26	
8.	Передзахист	04.06.26	
9.	Захист		

Студент _____ Назар СЕНИШИН
(підпис) (прізвище та ініціали)

Керівник _____ Микита ГОЛОВАКІН
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 80 сторінках, містить 9 рисунків, 10 таблиць, 1 додаток, 33 джерела в переліку посилань.

Мета роботи – розробка вебзастосунку що забезпечує централізований моніторинг та геопросторовий пошук інформації про українські музичні гурти, концертні майданчики й події з інтерактивною візуалізацією результатів на карті.

Методи та засоби: мова програмування Python 3.12, фреймворк Django 5.2 з розширенням GeoDjango, СУБД PostgreSQL 17 з розширенням PostGIS, геопросторовий тип geography із системою координат WGS84, бібліотека Django REST Framework 3.17, клієнтська частина React 18 із бібліотекою Leaflet, інтеграція з мовною моделлю через сервіс OpenRouter.

Основний зміст роботи: аналіз існуючих платформ моніторингу музичних подій, проектування нормалізованої схеми бази даних із просторовими полями типу geography, реалізація геопросторового радіус-пошуку через ORM-локап __dwithin, розробка REST API із рольовою моделлю доступу і журналом аудиту змін, інтеграція з мовною моделлю за принципом «AI як парсер намірів», тестування і верифікація коректності геопросторових обчислень.

Рекомендації щодо використання: застосування як інформаційної платформи для популяризації української музичної сцени; розширення доменної моделі через включення фестивалів і турів; розробка мобільного клієнта.

Результат – вебзастосунок із геопросторовим радіус-пошуком і природномовним пошуком українською мовою.

Ключові слова: геопросторовий пошук, PostGIS, GeoDjango, інтерактивна карта, музичні події, REST API, природномовний пошук, Leaflet.

ANNOTATION

The thesis is completed on 80 pages, contains 9 illustrations, 10 tables, 1 appendice, 33 sources in the reference list.

Objective: development of a web application providing centralized monitoring and geospatial search of Ukrainian music bands, venues and events with interactive map visualization.

Methods and tools: Python 3.12, Django 5.2 with GeoDjango extension, PostgreSQL 17 with PostGIS, geography data type with WGS84 coordinate system, Django REST Framework 3.17, React 18 with Leaflet, language model integration via OpenRouter.

Main content: analysis of existing music event platforms, design of a normalized database schema with geography spatial fields, implementation of geospatial radius search via __dwithin ORM lookup, REST API with role-based access control and audit journal, AI-as-intent-parser integration, testing and verification of geospatial computations.

Recommendations for use: application as an information platform for the Ukrainian music scene; domain model expansion to include festivals and tours; mobile client development.

Result – a web application with geospatial radius search and natural language search in Ukrainian.

Keywords: geospatial search, PostGIS, GeoDjango, interactive map, music events, REST API, natural language search, Leaflet.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI – Artificial Intelligence – штучний інтелект.

API – Application Programming Interface – програмний інтерфейс застосунку.

CORS – Cross-Origin Resource Sharing – спільне використання ресурсів між різними джерелами.

CSS – Cascading Style Sheets – каскадні таблиці стилів.

DRF – Django REST Framework – бібліотека побудови REST API для Django.

GDAL – Geospatial Data Abstraction Library – бібліотека абстракції геопросторових даних.

GEOS – Geometry Engine Open Source – відкритий рушій геометричних операцій.

GiST – Generalized Search Tree – узагальнене дерево пошуку (тип просторового індексу).

HTTP – Hypertext Transfer Protocol – протокол передачі гіпертексту.

JSON – JavaScript Object Notation – текстовий формат обміну даними.

LLM – Large Language Model – велика мовна модель.

MVCC – Multi-Version Concurrency Control – багатоверсійний контроль паралельного доступу.

OGC – Open Geospatial Consortium – відкритий геопросторовий консорціум.

ORM – Object-Relational Mapping – об'єктно-реляційне відображення.

PROJ – Бібліотека картографічних проекцій і перетворень систем координат.

REST – Representational State Transfer – архітектурний стиль побудови розподілених систем.

SRID – Spatial Reference Identifier – ідентифікатор системи просторових координат.

SQL – Structured Query Language – мова структурованих запитів.

SPA – Single Page Application – односторінковий вебзастосунок.

URL – Uniform Resource Locator – уніфікований локатор ресурсу.

UUID – Universally Unique Identifier – універсальний унікальний ідентифікатор.

WFS – Web Feature Service – стандарт OGC для публікації векторних просторових даних.

WGS84 – World Geodetic System 1984 – світова геодезична система координат 1984 року.

WMS – Web Map Service – стандарт OGC для публікації картографічних зображень.

ГІС – Геоінформаційна система.

СУБД – Система управління базами даних.

ВСТУП

Розвиток цифрових технологій суттєво змінив спосіб взаємодії між музикантами і аудиторією. Стримінгові сервіси вирішили проблему доступу до музичного контенту, проте задача пошуку живих виступів залишається інформаційно фрагментованою: відомості про гурти, майданчики і події розпорошені між різнорідними ресурсами без єдиної точки входу. Для українського музичного ринку ця проблема є особливо гострою: більшість світових платформ моніторингу концертних подій погано покривають локальний контент, а вітчизняні афіші орієнтовані виключно на продаж квитків і не надають інструментів географічно орієнтованого пошуку. Відсутність спеціалізованого рішення що поєднує структурований каталог українських виконавців із геопросторовим пошуком і відкритим API обумовлює необхідність розробки власного застосунку.

Актуальність теми.

Геоінформаційні технології досягли рівня зрілості що дозволяє реалізувати точний просторовий пошук у межах звичайної реляційної бази даних без розгортання окремої ГІС-інфраструктури. Поєднання PostGIS із сучасними засобами розробки вебзастосунків і великими мовними моделями відкриває можливість побудови системи що задовольняє всі виявлені потреби українського музичного ринку. Практична значущість роботи визначається тим що розроблений застосунок може використовуватись як інформаційна платформа для популяризації української музичної сцени як всередині країни так і серед діаспори за кордоном.

Мета роботи.

Метою дипломної роботи є розробка вебзастосунку що забезпечує централізований моніторинг та геопросторовий пошук інформації про українські музичні гурти, концертні майданчики й події з інтерактивною візуалізацією результатів на карті. Застосунок повинен підтримувати пошук у заданому географічному радіусі від довільної точки на карті з обчисленням відстаней на поверхні, забезпечувати рольове розмежування доступу між різними категоріями

користувачів і надавати природномовний інтерфейс пошуку українською мовою через інтеграцію з великою мовною моделлю.

Завдання роботи.

Насамперед провести порівняльний аналіз наявних платформ моніторингу концертних подій і обґрунтувати доцільність розробки власного рішення. Визначити функціональні і нефункціональні вимоги до системи. Обґрунтувати вибір технологічного стеку з урахуванням вимог до геопросторових обчислень. Спроектувати нормалізовану схему реляційної бази даних із просторовими типами даних і індексами GiST. Реалізувати геопросторовий радіус-пошук, REST API із рольовою моделлю доступу і журналом аудиту змін. Розробити клієнтську частину з інтерактивною картою і природномовним пошуком. Провести тестування і верифікацію коректності геопросторових обчислень.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ У СФЕРІ ГЕОПРОСТОРОВОГО ПОШУКУ МУЗИЧНИХ ПОДІЙ

У цьому розділі проведено аналіз існуючих рішень у сфері геопросторового пошуку музичних подій та обґрунтовано доцільність розробки власного застосунку. У підрозділі 1.1 сформульовано постановку задачі, визначено мету розробки та перелік функціональних і нефункціональних вимог до системи. Підрозділ 1.2 присвячено огляду геоінформаційних систем загального призначення з аналізом їх придатності для використання у контексті вебзастосунку. У підрозділі 1.3 розглянуто спеціалізовані платформи моніторингу концертних подій та виявлено їхні обмеження щодо українського контенту і геопросторової функціональності. Підрозділ 1.4 містить систематизований порівняльний аналіз розглянутих рішень за ключовими критеріями з висновком про необхідність розробки власного спеціалізованого рішення.

1.1 Постановка задачі розробки системи геопросторового пошуку музичних подій

Українська музична сцена переживає період активного розвитку: з'являються нові виконавці, розширюється географія концертної діяльності, зростає попит на інформацію про заходи як всередині країни, так і з боку української діаспори за кордоном. Водночас інформаційна інфраструктура, яка мала б забезпечувати зручний доступ до цих даних, залишається фрагментованою: відомості про гурти, майданчики та події розпорошені між різнорідними ресурсами без єдиної точки входу та без підтримки географічно орієнтованого пошуку.

Практика показує, що типовий запит користувача має виражений просторовий характер: людину цікавить не абстрактний перелік усіх концертів

країни, а події поблизу конкретного міста чи навіть конкретної точки на карті. Реалізація такого сценарію вимагає не просто текстової фільтрації за полем «місто», а повноцінного геопросторового пошуку з обчисленням відстаней на поверхні Землі, що є принципово іншою технічною задачею.

Метою даної роботи є проектування і програмна реалізація вебзастосунку, який забезпечує централізований моніторинг та геопросторовий пошук інформації про українські музичні гурти, концертні майданчики й події з інтерактивною візуалізацією результатів на карті. Застосунок орієнтований на широке коло користувачів: від звичайних відвідувачів концертів, що шукають заходи поблизу свого місцезнаходження, до менеджерів виконавців та організаторів подій, яким необхідний зручний інструмент керування власною інформацією в системі.

Для досягнення поставленої мети необхідно вирішити такі задачі:

1. Порівняти наявні платформи моніторингу концертних подій (Bandsintown, Songkick, Spotify Concerts, Karabas, Concert.ua та MusicBrainz) за ключовими критеріями: наявність структурованого каталогу виконавців, підтримка геопросторового радіус-пошуку, повнота українського контенту, наявність відкритого API та рольової моделі редагування. На основі порівняння обґрунтувати доцільність розробки власного рішення.

2. Обґрунтувати вибір технологічного стеку (Python 3.12, Django 5.2, PostgreSQL 17 з PostGIS, React 18, Leaflet) з огляду на вимоги до геопросторових обчислень на еліпсоїді, продуктивності розробки та можливостей подальшого масштабування системи. Порівняти обрані рішення з альтернативами (Flask, FastAPI, MySQL, Google Maps API) та аргументувати зроблений вибір.

3. Спроектувати нормалізовану до третьої нормальної форми схему реляційної бази даних із дванадцяти таблиць, що охоплює сутності регіонів, міст, жанрів, гуртів, майданчиків, подій та облікових записів користувачів. Для сутностей із географічною прив'язкою (City, Venue) визначити просторові поля типу geography з системою координат SRID 4326 та забезпечити автоматичне створення індексів GiST.

4. Реалізувати серверну логіку геопросторового радіус-пошуку через ORM-локап дозволяє __dwithin з обчисленням відстаней на поверхні еліпсоїда WGS84, розробити REST API з комбінованою фільтрацією за множиною критеріїв, впровадити чотирирівневу рольову модель доступу з матрицею прав та автоматичний журнал аудиту змін атрибутів гуртів у таблиці BandUpdateLog.

5. Розробити односторінковий React-застосунок із власною навігацією на основі стану, інтерактивною картою на базі Leaflet з відображенням результатів радіус-пошуку у вигляді кола та маркерів, сторінкою агрегованої статистики та інтерфейсом природномовного пошуку, що передає запити користувача до мовної моделі через сервіс OpenRouter і відображає результати на карті.

6. Провести тестування розробленої системи на підготовленому наборі даних із восьми гуртів, шести майданчиків та шести подій у різних містах України, перевірити коректність геопросторових обчислень порівнянням результатів із значеннями, розрахованими за формулою гаверсина, та підтвердити дотримання нефункціональних вимог до часу відгуку API.

Функціональні вимоги до системи охоплюють реєстрацію та автентифікацію користувачів, перегляд і фільтрацію каталогу гуртів і подій за множиною критеріїв, геопросторовий пошук у заданому радіусі від обраної точки з сортуванням результатів за зростанням відстані, додавання виконавців до списку вподобань, керування власними гуртами та подіями для авторизованих менеджерів, перегляд агрегованої статистики системи, а також природномовний пошук із використанням великої мовної моделі.

Поряд із функціональними сформульовано нефункціональні вимоги. Час відгуку основних API-запитів не повинен перевищувати однієї секунди при типовому навантаженні. Геопросторові обчислення мають виконуватись на поверхні еліпсоїда WGS84, що забезпечує метричну точність результатів на будь-якій відстані в межах території України. Система повинна коректно розмежовувати права доступу між чотирма визначеними ролями користувачів та унеможливлювати довільні SQL-запити з боку природномовного інтерфейсу.

Архітектурно систему реалізовано за класичною трирівневою схемою: сервер бази даних на основі PostgreSQL із розширенням PostGIS, сервер застосунків на базі Django з бібліотекою Django REST Framework та клієнтська частина на React із картографічною бібліотекою Leaflet. Такий поділ дозволяє розвивати кожен рівень незалежно і за потреби замінити клієнтську частину, наприклад мобільним застосунком, без змін серверного коду.

1.2 Огляд геоінформаційних систем загального призначення

Геоінформаційні системи (ГІС) є одним із базових інструментів роботи з просторовими даними і становлять окремий клас програмного забезпечення, що поєднує можливості зберігання, редагування, аналізу та візуалізації географічно прив'язаної інформації. Розуміння можливостей і обмежень існуючих ГІС-рішень є необхідною передумовою для обґрунтованого проектування власної системи, оскільки дозволяє запозичити перевірені підходи і водночас уникнути архітектурних рішень, що не відповідають вимогам конкретної предметної області.

Серед настільних ГІС-систем найширшого поширення набули ArcGIS компанії Esri та відкрита платформа QGIS. ArcGIS є комерційним продуктом із розвиненою екосистемою інструментів для професійного просторового аналізу: підтримує широкий спектр векторних і растрових форматів, надає засоби геообробки, побудови тематичних карт та публікації просторових даних у вигляді веб-сервісів. QGIS є відкритою альтернативою з подібною функціональністю і підтримкою стандартів Open Geospatial Consortium (OGC). Обидві системи активно використовуються в містобудуванні, екології, логістиці та інших галузях, де аналіз просторових закономірностей є основним завданням.

Проте застосування настільних ГІС у контексті вебзастосунку для моніторингу музичних подій є непрактичним з кількох причин. По-перше, ці системи орієнтовані на одного навченого аналітика, а не на широку аудиторію з

різним рівнем технічної підготовки. По-друге, вони не містять жодних вбудованих моделей даних, що відповідали б доменним сутностям музичної індустрії - гуртам, майданчикам, подіям. По-третє, настільні ГІС не передбачають багатокористувацького режиму роботи з розмежуванням прав доступу, що є ключовою вимогою розроблюваної системи. Інтеграція настільної ГІС із сучасним REST API і реактивним фронтендом вимагає значних додаткових зусиль і фактично зводить нанівець переваги готового рішення [6].

Серверні ГІС-платформи, зокрема ArcGIS Server та GeoServer, частково усувають обмеження настільних версій. Вони дозволяють публікувати просторові набори даних у вигляді стандартних OGC-сервісів (WMS, WFS, WCS), що робить їх доступними для веббраузерів і сторонніх клієнтів. GeoServer є відкритим рішенням на базі Java і широко використовується для публікації картографічних шарів у державних та наукових геоінформаційних порталах [5]. Проте ці платформи залишаються суто інфраструктурними компонентами: вони не надають жодної прикладної логіки, не підтримують поняття користувача з роллю та профілем, не мають механізмів автентифікації прикладного рівня. Для побудови повноцінного застосунку поверх GeoServer або ArcGIS Server необхідно додатково розробляти весь прикладний шар, що фактично означає паралельну підтримку двох окремих серверних компонентів.

Окрему нішу займають хмарні картографічні платформи -- Google Maps Platform і Mapbox. Вони надають зручні JavaScript SDK для відображення карт у браузері, геокодування адрес і побудови маршрутів. Однак серверна частина цих платформ закрита: розробник може лише відображати картографічні тайли і виконувати обмежений набір запитів через HTTP API, але не зберігати власні геопросторові дані в їхній інфраструктурі та не виконувати довільних просторових запитів. Крім того, обидва сервіси є платними при перевищенні безкоштовних лімітів, що робить їх менш привабливими для навчальних і некомерційних проектів порівняно з відкритим стеком PostGIS і Leaflet [6].

Таким чином, розглянуті ГІС-рішення вирішують задачу роботи з

просторовими даними на інфраструктурному рівні, але не забезпечують готового фундаменту для побудови прикладного вебзастосунку з власною доменною моделлю, рольовою системою та REST API. Це обґрунтовує доцільність використання реляційної СУБД із вбудованим геопросторовим розширенням як основи зберігання даних, поверх якої будується повноцінна прикладна логіка засобами серверного фреймворку.

1.3 Огляд платформ моніторингу подій

Окрім класичних ГІС-систем, на ринку існує ціла категорія спеціалізованих платформ, орієнтованих безпосередньо на моніторинг концертних подій та музичної активності виконавців. Ці платформи за своєю природою ближчі до розроблюваного застосунку, ніж універсальні геоінформаційні системи, проте кожна з них має характерні обмеження, що не дозволяють повністю задовольнити сформульовані вимоги.

Bandsintown є однією з найпопулярніших світових платформ для відстеження концертної активності виконавців. Сервіс дозволяє користувачам підписуватись на улюблених артистів і отримувати сповіщення про заплановані події у їхньому місті. Платформа інтегрується з провідними стримінговими сервісами (Spotify, Apple Music, Deezer), що дозволяє автоматично формувати список відстежуваних виконавців на основі музичних вподобань користувача. Bandsintown також надає інструменти для самих артистів і їхніх менеджерів для публікації та просування подій. Попри широку функціональність, платформа має суттєві обмеження для українського ринку. Географічне покриття концентрується навколо великих міжнародних турів, тоді як локальна концертна активність українських гуртів представлена фрагментарно. Пошук реалізований виключно через вибір конкретного міста зі списку без підтримки радіус-пошуку за довільними координатами. API платформи є закритим і доступним лише за окремою

комерційною угодою.

Songkick є прямим конкурентом Bandsintown із подібною базовою функціональністю. Платформа надає більш розвинені інструменти аналітики для виконавців: статистику переглядів сторінок, географію аудиторії, динаміку зростання фанбази. Songkick також має часткову підтримку рольової моделі в тому сенсі, що верифіковані менеджери виконавців можуть редагувати інформацію про своїх артистів і публікувати події від їх імені. Однак, як і Bandsintown, платформа орієнтована переважно на англомовний контент і не підтримує пошук за географічним радіусом. Українські гурти, що не виступають на міжнародній арені, у каталозі Songkick практично відсутні.

Стримінгові платформи зі вбудованими функціями моніторингу подій -- Spotify Concerts, Apple Music Live, YouTube Music -- успадковують фундаментальне обмеження стримінгових сервісів: їхня концертна функціональність є вторинною і прив'язана до бібліотеки виконавців, яких користувач вже слухає. Пошук подій виконується не за географічним принципом, а за списком відстежуваних артистів, що принципово відрізняється від сценарію «знайди концерт поблизу мого міста незалежно від виконавця». Інформація про події постачається через сторонніх агрегаторів, що знижує її актуальність і повноту, особливо для локальних українських заходів.

Серед вітчизняних рішень найбільш відомими є Karabas та Concert.ua. Обидва сервіси є передусім системами продажу квитків і реалізують пошук за категоріями та містами. Karabas має зручний інтерфейс і досить широке покриття українських подій, проте структурованих даних про самих виконавців як окрему сутність не зберігає - гурт чи артист присутній у системі лише як атрибут конкретної події, а не як самостійний об'єкт каталогу з власною сторінкою, жанрами та географічною прив'язкою. Геопросторовий радіус-пошук у жодному з цих сервісів не реалізований.

Окремо варто розглянути MusicBrainz - велику відкриту базу метаданих про музичні твори, виконавців і релізи, що підтримується некомерційною організацією

MetaBrainz Foundation. На відміну від перелічених платформ, MusicBrainz надає повністю відкритий REST API і підтримує вікіподібну модель редагування контенту зареєстрованими користувачами. База містить інформацію про тисячі українських виконавців із зазначенням країни походження та основного жанру. Проте географічна прив'язка в MusicBrainz обмежена рівнем країни або регіону і не включає координат - просторові запити типу «знайди гурти у радіусі 50 км від Харкова» засобами цієї бази виконати неможливо. Крім того, MusicBrainz не містить інформації про концертні події та майданчики, що виходить за межі її основної місії як бази музичних метаданих.

Підсумовуючи огляд, можна констатувати, що жодна з розглянутих платформ не поєднує в собі повноцінного структурованого каталогу виконавців із географічною прив'язкою, геопросторового радіус-пошуку та достатнього покриття українського музичного ринку. Це підтверджує актуальність розробки спеціалізованого рішення, орієнтованого саме на українську сцену, з геопросторовим пошуком як центральною функціональністю.

1.4 Порівняльний аналіз існуючих рішень

Проведений огляд геоінформаційних систем та платформ моніторингу подій дозволяє перейти до систематизованого порівняння розглянутих рішень за критеріями, що безпосередньо випливають із функціональних вимог розроблюваного застосунку. Такий аналіз є необхідним кроком перед проектуванням власної системи, оскільки дозволяє чітко визначити незаповнену нішу на ринку та обґрунтувати архітектурні рішення, що відрізнятимуть новий продукт від існуючих аналогів.

Для порівняння обрано шість критеріїв. Перший – наявність структурованого каталогу виконавців як самостійної сутності з власними атрибутами (жанр, місто походження, статус, посилання на ресурси), а не просто як поля в записі про подію.

Другий – підтримка геопросторового радіус-пошуку за довільними координатами з обчисленням відстаней у кілометрах. Третій – повнота покриття українського музичного ринку, зокрема локальних гуртів що не виступають на міжнародній арені. Четвертий – наявність відкритого REST API без комерційних обмежень на доступ. П'ятий – підтримка рольової моделі редагування контенту, тобто можливість для верифікованих менеджерів виконавців керувати інформацією про своїх артистів. Шостий – орієнтація на україномовний контент і локалізований пошук.Результати порівняння зведено у таблицю 1.1.

Таблиця 1.1 – Порівняльна характеристика існуючих рішень

Система	**Каталог**	**Радіус-пошук**	**UA-контент**	**Відкритий API**	**Ролі**
Bandsintown	Так	Ні	Частково	Ні	Ні
Songkick	Так	Ні	Частково	Ні	Частково
Spotify Concerts	Ні	Ні	Частково	Ні	Ні
Karabas / Concert.ua	Ні	Ні	Так	Ні	Ні
MusicBrainz	Так	Ні	Частково	Так	Так (вікі)
Розроблюваний застосунок	Так	Так	Так	Так	Так

Аналіз таблиці 1.1 виявляє кілька характерних закономірностей. Світові платформи (Bandsintown, Songkick) мають розвинений каталог виконавців, проте географічно орієнтований пошук у жодній із них не реалізований. Обидві платформи використовують закриті API з комерційними умовами доступу, що унеможливлює їх вільну інтеграцію у сторонні проекти. Покриття українського контенту носить випадковий характер і залежить від того, чи виступає конкретний гурт на міжнародних майданчиках.

Вітчизняні платформи продажу квитків (Karabas, Concert.ua) мають

протилежний профіль: вони добре покривають український ринок і є єдиними з розглянутих рішень, що реально орієнтовані на локального користувача. Проте їхня архітектура підпорядкована логіці транзакцій, а не каталогізації: виконавець у цих системах існує лише в контексті конкретної події з квитками, не маючи власної сторінки з описом, жанрами та географічною прив'язкою. Геопросторовий пошук у них відсутній повністю [9].

MusicBrainz вирізняється серед розглянутих рішень відкритістю і глибиною каталогу виконавців, проте її місія обмежена зберіганням музичних метаданих – координати міст і концертних майданчиків поза межами її моделі даних. Просторові запити засобами MusicBrainz API виконати неможливо, а інформація про конкретні події в базі відсутня [9].

Розроблюваний застосунок є єдиним із розглянутих рішень, що одночасно задовольняє всі шість критеріїв. Ключовою відмінністю від існуючих платформ є поєднання повноцінного структурованого каталогу українських виконавців із геопросторовим радіус-пошуком на основі реальних географічних координат. Саме ця комбінація визначає технічну новизну роботи і обґрунтовує вибір PostgreSQL із розширенням PostGIS як основи зберігання даних замість звичайної реляційної СУБД без просторових можливостей.

2 ЗАСОБИ ТА ТЕХНОЛОГІЇ РОЗРОБКИ ГЕОПРОСТОРОВОГО ВЕБЗАСТОСУНКУ

Вибір технологічного стеку є одним із ключових архітектурних рішень при розробці будь-якої інформаційної системи, оскільки він визначає не лише швидкість реалізації, а й продуктивність готового продукту, можливості його подальшого масштабування та складність підтримки. Для системи з геопросторовою функціональністю цей вибір є особливо відповідальним: не всі технології однаково придатні для роботи з просторовими типами даних, обчислення відстаней на поверхні еліпсоїда та побудови індексів для ефективного просторового пошуку.

У цьому розділі обґрунтовано вибір кожного компонента технологічного стеку розроблюваного застосунку. У підрозділі 2.1 розглянуто вибір мови програмування та серверного фреймворку з порівнянням альтернативних рішень. Підрозділ 2.2 присвячено реляційній СУБД PostgreSQL та її геопросторовому розширенню PostGIS як основі зберігання просторових даних. У підрозділі 2.3 описано геопросторове розширення GeoDjango та його інтеграцію з ORM фреймворку. Підрозділ 2.4 охоплює бібліотеку Django REST Framework як інструмент побудови REST API. У підрозділі 2.5 обґрунтовано вибір клієнтської частини на основі React, Vite та Leaflet. Підрозділ 2.6 описує механізм інтеграції з мовною моделлю через сервіс OpenRouter для реалізації природномовного пошуку.

2.1 Вибір мови програмування та серверного фреймворку

Вибір технологічного стеку для системи з геопросторовою функціональністю є нетривіальним завданням, оскільки підтримка просторових типів даних і ефективних просторових запитів є вимогою, що суттєво звужує перелік придатних

інструментів. У цьому підрозділі обґрунтовано вибір мови програмування та серверного фреймворку з урахуванням специфіки задачі.

Мовою серверної реалізації обрано Python версії 3.12. Серед ключових переваг цієї мови у контексті задачі варто виділити кілька аспектів. По-перше, Python має зрілу екосистему бібліотек для роботи з геопросторовими даними: GDAL, Shapely, Fiona та інші інструменти добре інтегруються між собою і активно підтримуються спільнотою. По-друге, синтаксис мови є достатньо виразним, щоб серверна логіка залишалась читабельною навіть при роботі зі складними ORM-запитами, що включають геопросторові анотації та багаторівневі фільтри. По-третє, Python 3.12 приніс помітні покращення продуктивності інтерпретатора порівняно з попередніми версіями, а також вдосконалений механізм обробки винятків із більш інформативними трасуваннями [14, 15].

Вибір саме версії 3.12, а не попередніх стабільних версій 3.10 чи 3.11, обумовлений кількома конкретними змінами у мові. Python 3.12 остаточно прибрав застарілі конструкції що накопичувались із версії 3.9, і запровадив покращений синтаксис параметризованих узагальнених типів, що безпосередньо впливає на читабельність анотацій типів у серіалізаторах і допоміжних функціях проекту. Крім того, у цій версії суттєво прискорено виконання операцій з рядками і словниками, що є типовими операціями при обробці JSON-запитів і формуванні відповідей API. Django 5.2 офіційно підтримує Python 3.12 як основну цільову версію, що гарантує сумісність усіх внутрішніх механізмів фреймворку без додаткових обхідних рішень [15].

Серед альтернативних мов розглядалися Java і Node.js. Java має потужну екосистему для корпоративної розробки і добру підтримку просторових бібліотек через GeoTools, однак значно більший обсяг шаблонного коду уповільнює розробку і ускладнює підтримку. Node.js є природним вибором для реактивних застосунків з великою кількістю одночасних з’єднань, проте підтримка геопросторових операцій у його екосистемі значно бідніша порівняно з Python: зокрема, відсутній аналог GeoDjango з інтеграцією в ORM [13].

У ролі серверного фреймворку обрано Django версії 5.2. Django є фреймворком повного циклу, що надає готові рішення для маршрутизації запитів, об’єктно-реляційного відображення (ORM), системи міграцій схеми бази даних, вбудованої панелі адміністратора та механізмів автентифікації. Для задачі з виразною доменною моделлю і значною кількістю взаємопов’язаних сутностей такий підхід забезпечує помітну економію часу розробки порівняно з мікрофреймворками.

Визначальним аргументом на користь Django є модуль django.contrib.gis, відомий як GeoDjango. Це геопросторове розширення є частиною самого фреймворку і не потребує окремої установки. GeoDjango додає до стандартного ORM підтримку просторових типів даних, просторових індексів та спеціалізованих операторів порівняння і вимірювання відстаней. Жоден інший Python-фреймворк не має настільки глибоко інтегрованої геопросторової підтримки: у Flask або FastAPI аналогічну функціональність довелось би збирати з окремих бібліотек без єдиного узгодженого інтерфейсу.

Розглядалась також альтернатива у вигляді FastAPI. Цей фреймворк має кращу продуктивність при асинхронному навантаженні і зручну автоматичну генерацію OpenAPI-документації. Проте відсутність вбудованого ORM і геопросторових розширень робить його менш придатним для задачі з інтенсивною роботою з базою даних і складними просторовими запитами. Flask має схожі обмеження і додатково поступається FastAPI у частині сучасних можливостей асинхронної обробки [14].

Система міграцій Django відіграє особливу роль саме у геопросторових проектах. Коли модель містить поле PointField або інший просторовий тип, Django автоматично генерує міграцію, що не лише створює відповідний стовпець у базі даних, а й додає просторовий індекс GiST без жодних додаткових інструкцій з боку розробника. Це принципово відрізняється від ручного керування схемою бази: при зміні просторового поля або додаванні нової геопросторової сутності достатньо виконати стандартні команди makemigrations і migrate, і вся інфраструктура

індексів буде оновлена автоматично [10].

Окрім самого Django, проект використовує кілька ключових пакетів що складають цілісну серверну екосистему. Бібліотека djangorestframework версії 3.17 реалізує REST-інтерфейс поверх Django і детально розглядається у підрозділі 2.4. Пакет django-cors-headers забезпечує коректну обробку CORS-заголовків, що є необхідним при розгортанні клієнтської і серверної частин на різних доменах або портах. Бібліотека python-dotenv дозволяє зберігати конфіденційні параметри конфігурації (рядок підключення до бази даних, ключі API) у зовнішньому файлі .env поза репозиторієм системи контролю версій. Пакет psycopg версії 3.3 є сучасним драйвером підключення Python до PostgreSQL і підтримує як синхронний, так і асинхронний режими роботи.

2.2 Реляційна СУБД PostgreSQL та розширення PostGIS

Вибір системи управління базами даних є фундаментальним рішенням для будь-якого застосунку з інтенсивною роботою з даними, проте для системи з геопросторовою функціональністю цей вибір додатково обмежений вимогою підтримки просторових типів даних і ефективних просторових запитів. У цьому підрозділі обґрунтовано вибір PostgreSQL 17 із розширенням PostGIS як основи зберігання даних розроблюваного застосунку.

PostgreSQL є об’єктно-реляційною СУБД з відкритим кодом, що розвивається з 1986 року і на сьогодні є однією з найбільш функціонально насичених реляційних систем серед вільно поширюваних рішень. Серед її ключових характеристик варто виділити строгу підтримку стандарту SQL із розширеннями, багатоверсійний контроль одночасного доступу (MVCC), що забезпечує високу паралельність читання і запису без блокувань, розширений набір вбудованих типів даних (масиви, JSON, діапазони, UUID), надійний механізм транзакцій із підтримкою точок збереження, а також розвинену систему

розширень, що дозволяє додавати нову функціональність без модифікації ядра СУБД [3].

Версія 17, обрана для проекту, є актуальним стабільним випуском із покращеною продуктивністю планувальника запитів і вдосконаленою підтримкою JSON-операцій. Для цілей розроблюваного застосунку особливо важливою є стабільність роботи з великими обсягами даних і коректна взаємодія з розширенням PostGIS, яке офіційно підтримує PostgreSQL 17 починаючи з версії PostGIS 3.4 [6].

Серед альтернативних СУБД розглядалися MySQL і SQLite. MySQL є широко поширеною реляційною СУБД із добрими показниками продуктивності при простих запитах, проте підтримка просторових типів у ній реалізована значно скромніше ніж у PostgreSQL: просторові індекси у MySQL обмежені типом MyISAM і не підтримують повний набір операцій над географічними об’єктами що надає PostGIS. Крім того, MySQL поступається PostgreSQL у частині складних аналітичних запитів із вкладеними підзапитами і оконними функціями, які використовуються у моніторинговому ендпоінті системи [1]. SQLite є вбудованою СУБД без окремого серверного процесу і підходить для локального зберігання даних у мобільних застосунках або невеликих утилітах, проте не призначений для багатокористувацьких серверних застосунків із одночасними запитами від кількох клієнтів. Підтримка просторових типів у SQLite реалізована через розширення SpatiaLite, яке значно поступається PostGIS за повнотою функціональності і стабільністю [5].

Критично важливим компонентом обраної СУБД є розширення PostGIS, що перетворює PostgreSQL на повноцінну геопросторову базу даних. PostGIS додає підтримку геопросторових типів даних відповідно до стандарту OGC Simple Features: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon і GeometryCollection. Окрім типів, розширення надає понад тисячу просторових функцій для обчислення відстаней, площ і периметрів, перевірки топологічних відношень між геометріями (перетин, містить, дотик), побудови буферних зон,

злиття і різниці геометрій, а також перетворення між різними системами координат [8].

PostGIS реалізує стандарт Open Geospatial Consortium і є де-факто індустріальним стандартом для геопросторової роботи у відкритому стеку. Розширення активно використовується у державних геоінформаційних системах, логістичних платформах, системах моніторингу транспорту та інших промислових застосунках, що підтверджує його зрілість і надійність [7].

Принципово важливою для розроблюваного застосунку є відмінність між двома типами просторових даних PostGIS: geometry і geography. Тип geometry трактує координати як точки на площині і виконує обчислення у пласкому декартовому просторі. Це забезпечує високу швидкість обчислень, проте дає некоректні результати на великих відстанях через спотворення що виникають при проектуванні сферичної поверхні Землі на площину. Тип geography натомість виконує обчислення безпосередньо на поверхні земного еліпсоїда WGS84, що забезпечує метричну точність результатів незалежно від географічного розташування об’єктів.

У задачі моніторингу на території всієї України відстані між об’єктами можуть сягати понад тисячу кілометрів: наприклад, відстань між Ужгородом і Луганськом перевищує 1400 км. На таких відстанях похибка типу geometry у декартових координатах стає неприйнятно великою і може суттєво спотворювати результати радіус-пошуку. Тип geography позбавлений цього недоліку і дозволяє коректно обчислювати відстані у кілометрах без додаткових перетворень координат між проекціями.

Просторова система координат, у якій зберігаються всі геопросторові поля розроблюваної системи, має ідентифікатор SRID 4326 що відповідає системі WGS84. Ця система є базовою для глобальних систем позиціонування і використовується сучасними картографічними сервісами та браузерним API navigator.geolocation, що забезпечує природну сумісність із клієнтською частиною застосунку без додаткових перетворень координат на межі між клієнтом і

сервером [9].

Для ефективного виконання просторових запитів PostGIS використовує спеціалізовані індекси типу GiST (Generalized Search Tree), що реалізують узагальнене R-дерево. На відміну від звичайних B-дерев, що ефективні для одновимірних упорядкованих даних, GiST-індекси оперують обмежуючими прямокутниками геометричних об’єктів і дозволяють швидко знаходити геометрії у заданій просторовій області без повного сканування таблиці. Складність просторового запиту з GiST-індексом є логарифмічною від розміру таблиці, що забезпечує прийнятний час відгуку навіть при значному зростанні кількості записів [7].

2.3 Геопросторове розширення GeoDjango

GeoDjango є офіційним геопросторовим розширенням фреймворку Django, реалізованим у модулі django.contrib.gis. На відміну від сторонніх пакетів, GeoDjango входить до стандартного дистрибутиву Django і не потребує окремої установки, що гарантує його сумісність із конкретною версією фреймворку і виключає конфлікти залежностей. Розширення забезпечує інтеграцію Django ORM із геопросторовими можливостями PostgreSQL і PostGIS, надаючи розробнику уніфікований інтерфейс для роботи з просторовими даними на рівні Python-коду без необхідності писати сирі SQL-запити з просторовими функціями [6].

Основою GeoDjango є набір спеціалізованих полів моделей: PointField, LineStringField, PolygonField, MultiPointField, MultiLineStringField, MultiPolygonField та GeometryField для зберігання геометрій довільного типу. Кожне з цих полів відповідає відповідному типу PostGIS на рівні бази даних і автоматично серіалізується у формат GeoJSON при поверненні через API. У розроблюваному застосунку використовується виключно PointField, оскільки всі геопросторові сутності системи є точковими об’єктами: міста представлені своїми

центроїдами, а концертні майданчики – точними координатами входу або сцени [8].

Параметр geography=True при оголошенні поля є ключовим для коректності геопросторових обчислень. Без цього параметра Django за замовчуванням створює стовпець типу geometry, що виконує обчислення у пласкому просторі. При увімкненому параметрі geography поле зберігається як тип geography(Point, 4326), і всі просторові операції над ним автоматично виконуються на поверхні еліпсоїда WGS84. Це означає що відстані обчислюються у метрах без будь-яких додаткових налаштувань і є коректними для будь-якої точки планети [10].

Центральною ORM-конструкцією для реалізації радіус-пошуку є просторовий локап __dwithin. Він перевіряє що геометрія поля знаходиться у межах заданої відстані від еталонної точки і генерує відповідний SQL-запит із функцією ST_Dwithin. У поєднанні з полем типу geography і класом-обгорткою D(km=N) з модуля django.contrib.gis.measure цей локап приймає відстань безпосередньо у кілометрах без необхідності знати деталі внутрішнього представлення координат. Для анотування результатів обчисленою відстанню використовується функція Distance() з того самого модуля, що дозволяє сортувати результати за зростанням відстані від точки пошуку засобами стандартного ORM-методу order_by.

Панель адміністратора Django з підключеним GeoDjango отримує спеціалізовані віджети для редагування геометричних полів. Замість стандартного текстового поля для введення координат адміністратор бачить інтерактивну карту на тайлах OpenStreetMap із можливістю розставляти точки клацанням миші. Це суттєво спрощує адміністрування геопросторових даних і знижує ймовірність введення некоректних координат порівняно з ручним введенням числових значень.

Для коректної роботи GeoDjango на різних операційних системах необхідні три зовнішні бібліотеки: GDAL, GEOS і PROJ. GDAL (Geospatial Data Abstraction Library) забезпечує читання і запис різноманітних форматів просторових даних і є основним інструментом перетворення між форматами. GEOS (Geometry Engine

Open Source) реалізує геометричні операції: перевірку топологічних відношень, обчислення перетинів і об’єднань геометрій. PROJ відповідає за картографічні проекції і перетворення між системами координат. На платформі Windows ці бібліотеки встановлюються у складі дистрибутиву OSGeo4W, що автоматично постачає сумісні версії всіх трьох компонентів і налаштовує необхідні змінні середовища. На Linux-системах бібліотеки доступні через стандартні пакетні менеджери. Шляхи до бібліотек налаштовуються через змінні оточення у файлі .env, що дозволяє переносити проект між операційними системами без зміни вихідного коду застосунку.

2.4 Бібліотека Django REST Framework

Серверна частина розроблюваного застосунку взаємодіє з клієнтом виключно через REST API, що повертає дані у форматі JSON. Для реалізації цього інтерфейсу поверх Django використовується бібліотека Django REST Framework (DRF) версії 3.17, яка є фактичним стандартом побудови REST-інтерфейсів у Django-екосистемі і на сьогодні налічує десятки мільйонів завантажень щомісяця [17].

Центральним компонентом DRF є система серіалізаторів. Серіалізатор перетворює об’єкт моделі Django на словник Python, який потім кодується у JSON і повертається клієнту, а також виконує зворотну операцію: приймає вхідні JSON-дані, валідує їх і перетворює на об’єкт моделі готовий до збереження. DRF надає два рівні серіалізаторів: базовий клас Serializer з явним оголошенням кожного поля і ModelSerializer що автоматично генерує поля на основі моделі Django. У проекті використовуються переважно ModelSerializer з окремими кастомними полями для специфічних випадків, наприклад для серіалізації геопросторових координат і обчисленої відстані.

Для серіалізації геопросторових полів у проекті реалізовано спеціальне поле

LongitudeLatitudeField, що представляє точку у вигляді об’єкта з полями longitude і latitude замість стандартного GeoJSON-представлення. Таке рішення зменшує обсяг переданих даних і спрощує клієнтську інтеграцію, оскільки клієнтська бібліотека Leaflet очікує координати саме у такому форматі. Для серіалізації обчисленої відстані при радіус-пошуку реалізовано міксин DistanceKmMixin, що підмішується до серіалізаторів гуртів, майданчиків і подій та додає поле distance_km округлене до двох десяткових знаків.

Організація обробників запитів у DRF реалізована через систему класів ViewSet. ViewSet об’єднує логіку обробки усіх стандартних HTTP-методів для одного ресурсу (GET список, GET деталь, POST, PATCH, DELETE) в одному класі, що усуває дублювання коду і спрощує реєстрацію URL-маршрутів через DRF-роутер DefaultRouter. У проекті реалізовано шість основних ViewSet для ресурсів regions, cities, genres, bands, venues і events, кожен із яких перевизначає методи get_queryset і get_serializer_class для реалізації специфічної логіки фільтрації та вибору серіалізатора залежно від типу запиту.

DRF надає гнучку систему класів дозволів (permissions), що дозволяє реалізувати довільну логіку перевірки прав доступу на рівні окремих ViewSet або навіть окремих методів. У проекті реалізовано власні класи дозволів: IsBandManager перевіряє що користувач є затвердженим менеджером конкретного гурту, IsEventManager аналогічно перевіряє права щодо події, IsAdminOrReadOnly дозволяє модифікуючі операції лише адміністраторам. Ці класи комбінуються через стандартний механізм permission_classes і забезпечують реалізацію матриці прав доступу описаної у підрозділі 3.6 [23].

Окремою перевагою DRF є вбудований браузерний інтерфейс API, що автоматично генерується для всіх зареєстрованих ендпоінтів. Цей інтерфейс дозволяє виконувати запити до API безпосередньо у браузері без додаткових інструментів, що суттєво прискорює процес розробки і тестування. При розробці розроблюваного застосунку браузерний інтерфейс DRF активно використовувався для перевірки коректності фільтрів і геопросторових запитів на реальних даних

тестового набору [29].

Автентифікація у проекті реалізована через стандартний механізм сесій Django з кукі. DRF підтримує цей механізм через клас SessionAuthentication і автоматично перевіряє автентифікацію користувача при кожному запиті. Такий підхід є достатнім для вебзастосунку де клієнт і сервер розгорнуті в одному домені або при коректному налаштуванні CORS для крос-доменних запитів [26].

2.5 Клієнтська частина: React, Vite та Leaflet

Клієнтська частина розроблюваного застосунку є односторінковим вебзастосунком (Single Page Application), що взаємодіє з сервером виключно через REST API і відповідає за відображення даних, інтерактивну карту та обробку користувацького вводу. Вибір технологій для клієнтської частини визначався трьома основними вимогами: підтримка компонентної архітектури інтерфейсу, наявність якісної картографічної бібліотеки з відкритим кодом і мінімальна кількість зовнішніх залежностей.

Бібліотеку React версії 18 обрано як основу клієнтської частини. React реалізує компонентну модель побудови інтерфейсу, в якій кожен елемент сторінки є самостійним компонентом із власним станом і логікою рендерингу. Односторонній потік даних від батьківського компонента до дочірніх через пропси забезпечує передбачуваність поведінки інтерфейсу і спрощує налагодження. Віртуальний DOM дозволяє ефективно перемальовувати лише ті частини сторінки що реально змінились, що є особливо важливим для картографічної сторінки з великою кількістю маркерів [20, 22].

React 18 зокрема запровадив механізм конкурентного рендерингу (Concurrent Rendering), що дозволяє React переривати тривалі операції рендерингу і обробляти більш пріоритетні оновлення інтерфейсу. Для картографічної сторінки з потенційно великою кількістю маркерів це означає що інтерфейс залишається

responsive навіть під час завантаження і відображення нових результатів пошуку. Строгий режим StrictMode, увімкнений у проекті, додатково допомагає виявляти побічні ефекти в компонентах на етапі розробки [21].

У ролі складальника модулів обрано Vite версії 5. Vite є інструментом нового покоління що кардинально відрізняється від традиційних складальників на кшталт Webpack за підходом до розробки. Замість попередньої компіляції всього коду застосунку Vite використовує нативні ES-модулі браузера і компілює лише ті модулі що реально запитуються поточною сторінкою. Це забезпечує практично миттєвий запуск сервера розробки незалежно від розміру проекту, що суттєво прискорює цикл розробки порівняно з Webpack де час запуску зростає пропорційно кількості модулів [22].

Для відображення інтерактивних карт обрано бібліотеку Leaflet у поєднанні з її React-обгорткою react-leaflet. Leaflet є легкою картографічною бібліотекою з відкритим кодом що інтегрується з тайловими серверами стандарту XYZ, зокрема з OpenStreetMap. Порівняно з комерційними альтернативами (Google Maps API, Mapbox GL JS) Leaflet має кілька принципових переваг: відсутність ліцензійних платежів і обмежень на кількість запитів, скромні вимоги до браузерних ресурсів, простий і добре документований API [31].

Бібліотека react-leaflet надає декларативні React-компоненти що обгортають відповідні об’єкти Leaflet: MapContainer для ініціалізації карти, TileLayer для підключення тайлового сервера, Marker і CircleMarker для відображення точкових об’єктів, Circle для відображення кола радіус-пошуку, Popup для інформаційних вікон при кліку на маркер. Хуки useMap і useMapEvents забезпечують програмне керування картою і обробку подій кліку користувача безпосередньо у функціональних компонентах React [30].

Окремою архітектурною особливістю клієнтської частини є відмова від зовнішніх бібліотек маршрутизації і керування глобальним станом. Замість поширеного підходу з react-router у застосунку реалізовано власну легку навігацію на основі хука useState зі збереженням імені активної сторінки. Для застосунку з

фіксованою кількістю сторінок такий підхід є достатнім і не вимагає підключення додаткових залежностей що збільшують розмір клієнтського пакунка [24].

Глобальний стан автентифікації керується через React Context із компонентом AuthProvider, що зберігає дані поточного користувача і надає їх усім дочірнім компонентам без явної передачі через пропси на кожному рівні ієрархії. Це стандартний і добре зарекомендований підхід для керування станом автентифікації у застосунках середнього розміру де повноцінний менеджер стану типу Redux є надлишковим.

Сторінка моніторингу заслуговує окремої згадки з точки зору архітектурного рішення: вся її візуалізація реалізована засобами чистого CSS без підключення сторонніх бібліотек діаграм. Горизонтальні стовпчикові діаграми для топ-5 міст і топ-5 жанрів побудовані через CSS-властивість width з динамічним значенням що обчислюється відносно максимального показника у вибірці. Таке рішення суттєво зменшує розмір клієнтського пакунка і прискорює початкове завантаження сторінки.

2.6 Інтеграція з мовною моделлю через OpenRouter

Окремою функціональною можливістю розроблюваної системи є природномовний пошук, що дозволяє користувачеві формулювати запити звичайною українською мовою замість заповнення форми з фільтрами. Технічна реалізація цієї можливості базується на інтеграції з великою мовною моделлю через сервіс OpenRouter, що виступає єдиним шлюзом до моделей різних виробників: OpenAI, Anthropic, Meta та інших. Перевагою такого підходу є можливість змінювати конкретну модель через змінну оточення без жодних змін у вихідному коді застосунку [32].

Ключовим архітектурним рішенням підсистеми природномовного пошуку є принцип використання мовної моделі виключно як парсера намірів користувача.

Модель не отримує жодного прямого доступу до бази даних і не генерує SQL-запити. Натомість вона повертає строго типізований JSON-план із визначеною схемою що містить ідентифікатор сутності (bands, events або venues) та значення фільтрів у канонічному вигляді. Цей план далі інтерпретується серверним кодом і виконується через стандартні ORM-запити, що принципово виключає можливість SQL-ін’єкцій з боку мовної моделі.

Серверна підсистема природномовного пошуку реалізована як окремий Django-додаток ai_search із чітко розділеною структурою модулів. Модуль openrouter відповідає за виклик зовнішнього сервісу і повернення сирої відповіді моделі. Модуль plan виконує валідацію відповіді проти суворого білого списку дозволених полів. Модуль normalization перетворює україномовні значення у формат очікуваний існуючими ViewSets: наприклад назва міста «Київ» замінюється на slug «kyiv». Модуль executor виконує нормалізований план через одну зі стандартних ViewSets, перевикористовуючи всю існуючу логіку фільтрації та серіалізації.

У випадках коли мовна модель повертає відповідь що не відповідає очікуваній схемі, серверний код перехоплює виняток на етапі валідації і повертає клієнту зрозуміле повідомлення про помилку з HTTP-статусом 400, не допускаючи витоку внутрішніх деталей реалізації.

3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ ГЕОПРОСТОРОВОГО ПОШУКУ

Проектування і програмна реалізація є центральним етапом розробки будь-якої інформаційної системи, оскільки саме на цьому етапі абстрактні вимоги перетворюються на конкретні архітектурні рішення, схеми даних і працюючий код. Для системи з геопросторовою функціональністю цей етап є особливо відповідальним: помилки у проектуванні схеми бази даних або у виборі просторових типів даних важко виправити після наповнення системи реальними даними без повної міграції.

У цьому розділі детально описано результати проектування і реалізації розроблюваного застосунку. У підрозділі 3.1 розглянуто загальну архітектуру застосунку і поділ на логічні Django-додатки. Підрозділ 3.2 присвячено проектуванню схеми бази даних із дванадцяти таблиць і обґрунтуванню прийнятих рішень щодо зв’язків між сутностями. У підрозділі 3.3 описано геопросторові типи даних і механізм просторових індексів. Підрозділ 3.4 містить детальний опис реалізації радіус-пошуку і обчислення відстаней. У підрозділі 3.5 розглянуто структуру REST API і механізми фільтрації запитів. Підрозділ 3.6 описує рольову модель доступу і систему аудиту змін. У підрозділі 3.7 розглянуто підсистему моніторингу і природномовного пошуку.

3.1 Загальна архітектура застосунку

Архітектура розроблюваного застосунку побудована за класичною трирівневою схемою що передбачає чіткий поділ між рівнем зберігання даних, рівнем серверної логіки і рівнем клієнтського інтерфейсу. Такий поділ забезпечує незалежність кожного рівня від деталей реалізації інших і дозволяє розвивати

кожен компонент системи без необхідності вносити зміни до решти. Загальну архітектуру системи зображено на рисунку 3.1.

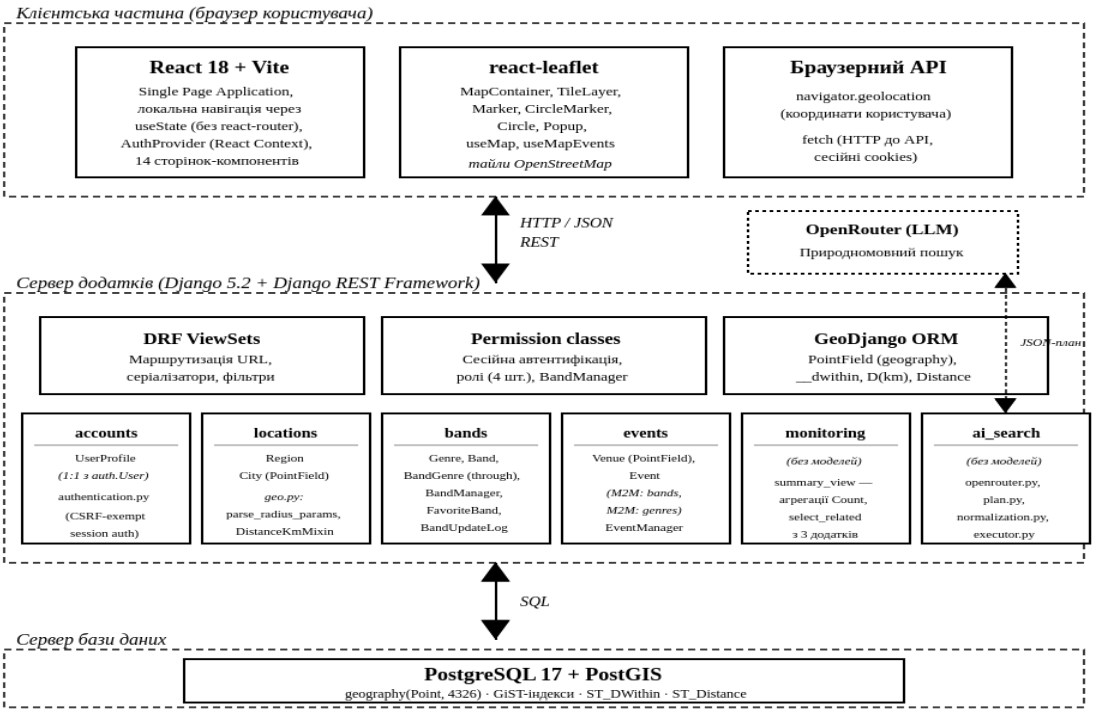

Рисунок 3.1 — Загальна архітектура застосунку

Рівень зберігання даних реалізовано на основі PostgreSQL 17 із розширенням PostGIS. Саме на цьому рівні зосереджена вся логіка просторових обчислень: індекси GiST, функції ST_Dwithin і ST_Distance виконуються безпосередньо у СУБД без передачі координат на рівень застосунків. Такий підхід є принципово правильним з точки зору продуктивності, оскільки перенесення просторових обчислень на рівень Python-коду означало б необхідність завантажувати всі записи з бази і фільтрувати їх у пам’яті застосунку.

Рівень серверної логіки реалізовано на базі Django 5.2 із бібліотекою Django REST Framework. Цей рівень відповідає за автентифікацію і авторизацію користувачів, валідацію вхідних даних, формування ORM-запитів до бази даних, серіалізацію результатів у JSON і повернення HTTP-відповідей. Клієнтська частина

взаємодіє з цим рівнем виключно через REST API під спільним префіксом /api/, що дозволяє у майбутньому замінити клієнт мобільним застосунком без жодних змін серверного коду.

Рівень клієнтського інтерфейсу реалізовано як односторінковий React-застосунок що завантажується браузером одноразово і далі взаємодіє з сервером через асинхронні HTTP-запити. Такий підхід забезпечує плавну навігацію між сторінками, що є важливим для картографічної сторінки де перезавантаження означало б повторну ініціалізацію карти і втрату поточного стану перегляду.

На рівні серверної логіки застосунок поділено на шість логічних Django-додатків. Перелік додатків із їх призначенням наведено у таблиці 3.1.

Таблиця 3.1 – Логічні Django-додатки застосунку

Додаток	**Призначення**	**Основні компоненти**
accounts	Облікові записи та профілі користувачів	UserProfile, authentication.py
locations	Адміністративно-територіальний поділ України	Region, City (PointField), geo.py
bands	Каталог музичних гуртів	Band, Genre, BandGenre, BandManager, BandUpdateLog, FavoriteBand
events	Концертні майданчики та події	Venue (PointField), Event, EventManager
monitoring	Агрегований огляд стану системи (без моделей)	summary_view
ai_search	Природномовний пошук (без моделей)	openrouter.py, plan.py, normalization.py, executor.py

Два додатки реалізують наскрізну функціональність без власних моделей бази даних. Додаток monitoring надає агрегований огляд стану системи через єдину функцію-уявлення що читає дані з моделей інших додатків через ORM-агрегації. Додаток ai_search забезпечує природномовний пошук через інтеграцію з мовною моделлю і перевикористовує існуючі ViewSets для виконання нормалізованих планів запитів.

Усі додатки використовують спільний шар конфігурації реалізований як Django-проект config. Конфігурація читається із зовнішнього файлу .env через бібліотеку python-dotenv, що дозволяє безпечно зберігати секрети поза репозиторієм системи контролю версій і легко змінювати параметри розгортання без модифікації вихідного коду. Взаємодія між рівнями відбувається виключно через визначені інтерфейси: серверна частина не має жодної залежності від клієнтської реалізації, а клієнтська частина не має прямого доступу до бази даних. Така ізоляція є запорукою довгострокової підтримуваності системи і можливості незалежного розвитку кожного рівня.

3.2 Проектування схеми бази даних

Проектування схеми бази даних є одним із найвідповідальніших етапів розробки системи, оскільки від якості схеми безпосередньо залежить коректність зберігання даних, продуктивність запитів і складність подальшого розширення функціональності. При проектуванні схеми дотримувались принципу нормалізації до третьої нормальної форми, що виключає надлишковість даних і аномалії оновлення.

Доменна модель застосунку складається з дванадцяти основних таблиць що перебувають у відносинах один-до-багатьох, багато-до-багатьох та один-до-одного. Спрощену схему бази даних наведено на рисунку 3.2.

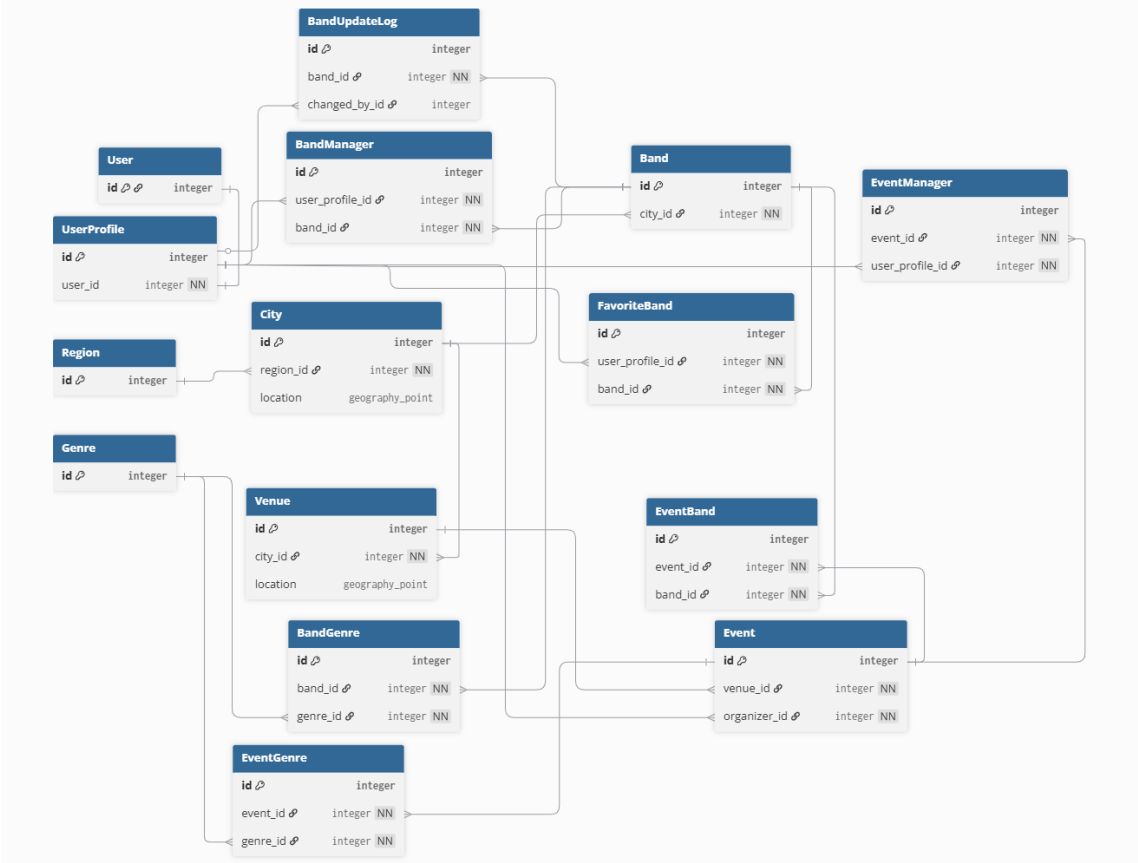

Рисунок 3.2 – Схема бази даних застосунку

Центральною сутністю доменної моделі є гурт (Band), пов’язаний із містом (City) через зовнішній ключ city_id. Такий зв’язок відображає природу доменної моделі: гурт має місто походження яке є самостійною географічною сутністю з власними координатами, а не просто текстовим полем назви. Це рішення дозволяє виконувати геопросторовий пошук гуртів через координати їхнього міста без дублювання просторових даних у таблиці Band. Місто (City) своєю чергою пов’язане з областю (Region) через зовнішній ключ region_id, що забезпечує повноцінну адміністративно-територіальну ієрархію.

Зв’язок гурту з жанрами реалізовано через явну проміжну таблицю BandGenre замість стандартного Django ManyToManyField без додаткових атрибутів. Це рішення обумовлено необхідністю зберігати ознаку основного жанру

для кожного гурту: поле is_primary у таблиці BandGenre дозволяє відрізнити головний жанровий напрям виконавця від другорядних. Аналогічна проміжна таблиця EventGenre реалізує зв'язок між подією і жанрами, що дозволяє редактору самостійно задавати жанровий контекст конкретної події незалежно від жанрів запрошених гуртів.

Подія (Event) пов'язана з концертним майданчиком (Venue) через зовнішній ключ venue_id і успадковує його географічну прив'язку. Зв'язок між подіями і гуртами реалізовано через проміжну таблицю EventBand. Таблиця FavoriteBand реалізує відношення багато-до-багатьох між користувачами і гуртами і дозволяє кожному користувачеві мати власний список обраних виконавців. Модель UserProfile розширює стандартну модель автентифікації Django через відношення один-до-одного і додає атрибут ролі користувача.

Перелік усіх сутностей доменної моделі з їх призначенням і наявністю просторових полів наведено у таблиці 3.2.

Таблиця 3.2 – Сутності доменної моделі застосунку

Сутність	**Призначення**	**Просторове поле**
Region	Область України	Відсутнє
City	Місто з посиланням на область	PointField (центроїд)
Genre	Музичний жанр	Відсутнє
Band	Музичний гурт прив'язаний до міста	Відсутнє (через City)
BandGenre	Зв'язок гурт – жанр з ознакою основного	Відсутнє
BandManager	Затверджений менеджер гурту	Відсутнє
BandUpdateLog	Журнал змін атрибутів гурту	Відсутнє
FavoriteBand	Обрані гурти користувача	Відсутнє
Venue	Концертний майданчик	PointField (точні координати)

Продовження таблиці 3.2

Event	Концертна подія (M2M до Band і Genre)	Відсутнє (через Venue)
EventManager	Затверджений менеджер події	Відсутнє
UserProfile	Розширення профілю користувача	Відсутнє

Архітектурно прийнято свідоме рішення про різну точність географічної прив’язки для різних сутностей. Для гуртів використовується координата міста (центральна точка населеного пункту), оскільки точне місцезнаходження виконавця у межах міста не є частиною доменної моделі. Для майданчиків зберігаються точні координати, оскільки концертна локація є реальною точкою з конкретною адресою. Події успадковують точну прив’язку від свого майданчика. Такий підхід зменшує обсяг даних, уникає дублювання і коректно відображає природу кожної сутності.

На рівні моделей використано численні унікальні обмеження що запобігають появі дублікатів. Унікальність гурту забезпечується комбінацією назви і міста, унікальність майданчика – комбінацією назви і міста, унікальність запису менеджера – комбінацією користувача і гурту або події. Ці обмеження реалізовані через атрибут unique_together у класах Meta моделей Django і відображаються у відповідних міграціях як обмеження бази даних. Усі таблиці мають додаткові індекси для прискорення типових фільтрів: за статусом гурту, за датою початку події, за зовнішніми ключами що використовуються у JOIN-запитах.

Журнал змін BandUpdateLog заслуговує окремого опису оскільки його структура є нетривіальною. Кожен запис журналу фіксує одну зміну одного атрибута гурту і містить назву зміненого поля, старе значення, нове значення, тип зміни (UPDATED або STATUS_CHANGED), ідентифікатор користувача що вніс зміну і мітку часу. Зберігання змін на рівні окремих атрибутів а не цілих записів дозволяє відстежувати детальну історію редагувань і відображати її в інтерфейсі моніторингу у вигляді зрозумілої стрічки подій.

3.3 Геопросторові типи даних і просторові індекси

Коректний вибір просторових типів даних є критично важливим рішенням для геоінформаційної системи, оскільки від нього залежить точність обчислення відстаней і продуктивність просторових запитів. У розроблюваному застосунку всі просторові поля оголошено як PointField із параметром geography=True та просторовою системою координат SRID 4326 що відповідає системі WGS84.

На рівні бази даних таке поле зберігається як стовпець типу geography(Point, 4326). Тип geography на відміну від типу geometry виконує обчислення безпосередньо на поверхні земного еліпсоїда, що забезпечує метричну точність результатів для будь-якої точки планети. Різниця між двома типами є принциповою: при використанні типу geometry відстань між Львовом і Харковом обчислювалась би у градусах площинної проекції і вимагала б ручного перетворення в кілометри з урахуванням поточної широти, тоді як тип geography повертає відстань одразу в метрах без жодних додаткових операцій.

Просторова система координат WGS84 з ідентифікатором SRID 4326 обрана як єдина система координат для всіх геопросторових полів застосунку. Ця система є міжнародним стандартом для глобальних систем позиціонування і використовується браузерним API navigator.geolocation при визначенні місцезнаходження користувача. Використання єдиної системи координат на всіх рівнях застосунку виключає необхідність перетворення між проекціями і спрощує інтеграцію між клієнтською і серверною частинами.

Просторові поля наявні лише в тих сутностях для яких географічна прив’язка є частиною доменної моделі. Координати у системі передаються і зберігаються у форматі POINT(longitude latitude), тобто довгота передує широті. Цей порядок є канонічним для PostGIS і GeoDjango і відрізняється від інтуїтивно очікуваного порядку (широта, довгота) що використовується у більшості картографічних сервісів. У публічних API-параметрах для зручності клієнтів передбачені окремі параметри lat і lng що внутрішньо перетворюються на коректний Point-об’єкт

функцією parse_radius_params у модулі locations/geo.py. Для зворотного представлення координат у JSON-відповідях розроблено власний клас серіалізаційного поля LongitudeLatitudeField що видає координати у форматі з іменованими полями longitude і latitude.

Просторові індекси типу GiST автоматично створюються Django-міграціями для кожного просторового поля моделі. Індекс GiST реалізує узагальнене R-дерево що організовує геометричні об’єкти за їхніми обмежуючими прямокутниками. При виконанні просторового запиту PostGIS спочатку використовує індекс для швидкого відбору кандидатів чиї обмежуючі прямокутники перетинаються з областю пошуку, а потім виконує точну перевірку для кожного кандидата. Така двохетапна стратегія дозволяє уникнути повного сканування таблиці і забезпечує логарифмічну складність запиту відносно кількості записів.

Важливою особливістю просторових індексів GiST є їхня автоматична підтримка при операціях вставки і оновлення записів. На відміну від деяких спеціалізованих просторових індексів що потребують ручного перебудування після масового завантаження даних, індекс GiST у PostgreSQL оновлюється інкрементально при кожній операції зміни даних. Це означає що після виконання seed-команд для початкового наповнення бази тестовими даними індекси залишаються актуальними і не потребують додаткового обслуговування.

Для перевірки коректності просторових полів на рівні Python-коду GeoDjango автоматично валідує формат координат при збереженні об’єкта через ORM. Спроба зберегти точку з широтою поза діапазоном від -90 до 90 градусів або з довготою поза діапазоном від -180 до 180 градусів призводить до виключення ValidationError ще до відправки запиту до бази даних. Це додатковий рівень захисту від некоректних даних що доповнює перевірки на рівні API-серіалізаторів.

3.4 Реалізація радіус-пошуку та обчислення відстаней

Радіус-пошук є центральною геопросторовою функціональністю розроблюваної системи і відрізняє її від більшості існуючих платформ моніторингу музичних подій. Реалізація цієї функціональності охоплює кілька рівнів: валідацію вхідних параметрів, формування геопросторового ORM-запиту, анотування результатів обчисленою відстанню і серіалізацію у JSON-відповідь.

Радіус-пошук активується трьома query-параметрами запиту: lat (широта центральної точки), lng (довгота центральної точки) і radius_km (радіус пошуку у кілометрах). Усі три параметри є обов'язковими і повинні надходити разом: відсутність будь-якого з них призводить до відповіді з HTTP-статусом 400 і повідомленням про відсутній параметр. Перелік параметрів радіус-пошуку з їх обмеженнями наведено у таблиці 3.3.

Таблиця 3.3 – Параметри радіус-пошуку

Параметр	**Тип**	**Допустимі значення**	**Призначення**
lat	float	від -90 до +90	Широта центральної точки пошуку
lng	float	від -180 до +180	Довгота центральної точки пошуку
radius_km	float	більше 0	Радіус пошуку у кілометрах

Парсинг і валідація параметрів реалізовані у допоміжній функції parse_radius_params що розміщена у модулі locations/geo.py і повторно використовується з усіх ViewSets що підтримують просторовий пошук. Така

централізація логіки валідації виключає дублювання коду і гарантує однакову поведінку радіус-пошуку для гуртів, майданчиків і подій. Функція перевіряє що широта перебуває у межах від -90 до +90 градусів, довгота у межах від -180 до +180 градусів, а радіус є додатним числом. При некоректних значеннях функція піднімає виключення ValidationError що обробником REST Framework перетворюється на HTTP-відповідь із детальним повідомленням про конкретну проблему.

Безпосередньо ORM-запит для радіус-пошуку гуртів формується таким чином: спочатку з параметрів lat і lng створюється об’єкт Point із зазначенням системи координат SRID 4326, після чого до базового QuerySet застосовується фільтр із локапом __dwithin і параметром D(km=radius_km). Клас D є обгорткою GeoDjango що інкапсулює одиниці виміру відстані і передає їх до PostGIS у коректному форматі. У поєднанні з полем типу geography PostGIS виконує запит безпосередньо у метрах без додаткових перетворень координат між проекціями.

Результат запиту анотується додатковим полем distance що обчислюється функцією Distance() із модуля django.contrib.gis.db.models.functions. Ця функція генерує SQL-вираз ST_Distance що обчислює точну відстань між просторовим полем кожного запису і центральною точкою пошуку. Анотовані результати сортуються за зростанням поля distance через стандартний ORM-метод order_by, що відповідає типовому очікуванню користувача: найближчі об’єкти відображаються першими у списку результатів.

Серіалізація обчисленої відстані у JSON-відповідь реалізована через допоміжний клас DistanceKmMixin що підмішується до серіалізаторів гуртів, майданчиків і подій. Міксин перевіряє наявність анотації distance у поточному об’єкті і якщо вона присутня додає до JSON-відповіді поле distance_km із значенням округленим до двох десяткових знаків. Принциповим є те що це поле з’являється у відповіді лише за наявності радіус-параметрів у запиті, що дозволяє клієнтській частині автоматично визначати режим відображення і показувати відстань лише тоді коли вона реально обчислена.

Вибір точки відліку для обчислення відстані є різним для різних типів

сутностей і відображає їхню природу в доменній моделі. Для гуртів відстань обчислюється від координати їхнього міста (поле city.location), що означає однакову обчислену відстань для всіх гуртів одного міста. Для майданчиків використовуються їхні власні точні координати (поле venue.location). Для подій відстань обчислюється від координат майданчика до якого подія прив'язана, що є логічним оскільки саме майданчик є реальним місцем проведення заходу.

Окремою задачею є забезпечення коректної роботи радіус-пошуку у поєднанні з іншими фільтрами. Якщо запит містить одночасно радіус-параметри і текстовий фільтр за жанром або регіоном, обидва фільтри застосовуються послідовно до одного QuerySet: спочатку відбираються записи що задовольняють текстові критерії, після чого до отриманого підмножини застосовується просторовий фільтр. Такий підхід є коректним і ефективним оскільки дозволяє PostgreSQL використовувати як звичайні B-дерево індекси для текстових фільтрів так і GiST-індекс для просторового фільтру в межах одного SQL-запиту.

3.5 REST API та механізми фільтрації

REST API є єдиним інтерфейсом взаємодії між клієнтською і серверною частинами застосунку. Усі ендпоінти зведено до єдиного стилю адресації під спільним префіксом /api/ що забезпечує передбачуваність і зручність використання як клієнтською частиною так і потенційними зовнішніми споживачами. Перелік основних API-ендпоінтів наведено у таблиці 3.4.

Таблиця 3.4 – Основні ендпоінти REST API застосунку

Ендпоінт	Метод	Призначення
/api/bands/	GET	Список гуртів із фільтрацією та радіус-пошуком
/api/bands/{id}/	GET	Детальна інформація про гурт
/api/bands/	POST	Створення нового гурту (band_manager)
/api/bands/{id}/	PATCH	Оновлення атрибутів гурту (менеджер гурту)
/api/events/	GET	Список подій із фільтрацією та радіус-пошуком
/api/events/{id}/	GET	Детальна інформація про подію
/api/events/	POST	Створення нової події (organizer)
/api/venues/	GET	Список майданчиків із радіус-пошуком
/api/regions/	GET	Список областей України
/api/cities/	GET	Список міст із фільтрацією за областю
/api/genres/	GET	Список музичних жанрів
/api/monitoring/summary/	GET	Агрегована статистика системи

Продовження таблиці 3.4

/api/ai-search/	POST	Природномовний пошук через мовну модель
/api/auth/register/	POST	Реєстрація нового користувача
/api/auth/login/	POST	Вхід у систему
/api/auth/logout/	POST	Вихід із системи
/api/auth/profile/	GET	Профіль поточного користувача

Реєстрація URL-маршрутів реалізована через DRF-роутер DefaultRouter до якого підключено шість ресурсних ViewSets: regions, cities, genres, bands, venues і events. Роутер автоматично генерує маршрути для списку ресурсу (GET /api/resource/) і окремого запису (GET /api/resource/id/), що усуває необхідність вручну прописувати кожен маршрут. Окремими спеціалізованими ендпоінтами зареєстровано панель моніторингу (/api/monitoring/summary/), природномовний пошук (/api/ai-search/) і гілку автентифікації /api/auth/ з ендпоінтами реєстрації, входу, виходу та отримання профілю поточного користувача.

Параметризація запитів реалізована через query-параметри URL що дозволяє комбінувати декілька фільтрів одночасно. Наприклад запит /api/bands/?genre=rock&status=active&region=lvivska поверне всі активні рок-гурти із Львівської області. Логіка фільтрації реалізована у методі get_queryset кожного ViewSet і застосовує фільтри послідовно до одного QuerySet. Такий підхід дозволяє PostgreSQL оптимізувати виконання запиту використовуючи найбільш вибірковий індекс із доступних.

Ендпоінт гуртів підтримує фільтрацію за такими параметрами: region (slug області), city (slug міста), genre (slug жанру), status (активний, у перерві або

розформований), search (текстовий пошук за назвою і описом), favorite_bands (булевий прапор що повертає лише обрані гурти поточного користувача), а також радіус-параметри lat, lng і radius_km. Для подій додатково реалізовано фільтр upcoming що повертає лише ті події дата початку яких пізніша за поточний момент часу, а також фільтри date_from і date_to для вибірки подій у заданому часовому діапазоні.

Окремою технічною проблемою є уникнення дублювання записів у результатах фільтрації за жанром. Оскільки фільтр за жанром реалізований через JOIN із таблицею BandGenre, гурт що має кілька жанрів може з'явитись у результатах кілька разів якщо кілька його жанрів задовольняють умову фільтра. Для усунення цієї проблеми при наявності фільтра genre до QuerySet автоматично додається метод distinct() що усуває дублікати на рівні SQL-запиту.

Для уникнення проблеми N+1 запитів у списковому ендпоінті гуртів реалізовано механізм попереднього завантаження пов'язаних об'єктів. Метод select_related завантажує пов'язані місто і область одним JOIN-запитом замість окремого запиту для кожного гурту. Метод prefetch_related завантажує жанри через проміжну таблицю BandGenre одним додатковим запитом для всього списку. Окремий об'єкт Prefetch завантажує майбутні події кожного гурту з фільтрацією на рівні бази. Заздалегідь завантажені події зберігаються у тимчасовому атрибуті _prefetched_events який серіалізатор використовує для побудови блоку запланованих подій у картці кожного гурту без додаткових SQL-запитів. Без цієї оптимізації запит до списку із ста гуртів виконував би понад триста SQL-запитів замість чотирьох.

Серіалізатори застосунку реалізують два режими представлення даних залежно від контексту запиту. Списковий серіалізатор повертає скорочений набір полів достатній для відображення картки у списку: назву, місто, основний жанр і найближчу подію. Детальний серіалізатор повертає повний набір атрибутів включно з описом, посиланнями на зовнішні ресурси, повним списком жанрів і всіма запланованими подіями. Вибір між серіалізаторами виконується у методі

get_serializer_class ViewSet на основі значення атрибута action: для дій list і retrieve використовуються різні класи серіалізаторів.

3.6 Рольова модель доступу та аудит змін

Розмежування прав доступу є обов'язковою вимогою для системи де різні категорії користувачів мають різний рівень повноважень щодо керування даними. У розроблюваному застосунку реалізовано чотирирівневу рольову модель що охоплює всі сценарії взаємодії від анонімного перегляду до повного адміністрування.

Система розпізнає чотири ролі користувачів. Регулярний користувач (regular_user) має право переглядати всі публічні дані каталогу, додавати гурти до списку вподобань і використовувати природномовний пошук. Менеджер гурту (band_manager) додатково отримує право створювати нові гурти та редагувати власні у межах визначеного набору атрибутів. Організатор подій (organizer) може створювати події за участі гуртів у яких він є затвердженим менеджером. Адміністратор (admin) має повний доступ через стандартну панель адміністрування Django. Матрицю прав доступу наведено у таблиці 3.5.

Таблиця 3.5 – Матриця прав доступу

Роль	**Читання**	**POST band**	**PATCH band**	**POST event**	**PATCH event**
Анонім	+	-	-	-	-
Користувач	+	-	-	-	-
Менеджер гурту	+	+	+ (свої)	+ (свої гурти)	+ (свої події)
Організатор	+	-	-	+ (свої гурти)	+ (свої події)
Адміністратор	+	+	+	+	+

Роль користувача зберігається у полі role моделі UserProfile. Перевірка ролі при кожному запиті виконується через власні класи дозволів що успадковують базовий клас BasePermission із Django REST Framework. Метод has_permission перевіряє роль на рівні всього ViewSet, тоді як has_object_permission виконує детальнішу перевірку на рівні конкретного об'єкта: чи є поточний користувач затвердженим менеджером саме цього гурту або події.

Безпека процесу реєстрації забезпечена примусовим виставленням ролі regular_user при створенні нового облікового запису незалежно від вхідних даних запиту. Менеджерські ролі присвоюються адміністратором вручну після підтвердження запиту користувача. При успішному створенні гурту система автоматично породжує запис у таблиці BandManager з прапором is_approved=True в межах однієї транзакції разом зі збереженням самого гурту.

Підсистема аудиту змін реалізована через автоматичне ведення журналу BandUpdateLog при кожній успішній операції PATCH над гуртами. Перед збереженням знімається стан полів що підлягають зміні, після збереження порівнюються старі і нові значення, і для кожного реально зміненого поля створюється окремий запис у журналі. Тип зміни визначається автоматично: для атрибута status записується тип STATUS_CHANGED, для решти полів тип UPDATED. Анонімні користувачі мають доступ лише до читання публічних даних, спроба виконати модифікуючу операцію без автентифікації повертає HTTP-статус 401, а недостатність прав автентифікованого користувача повертає статус 403.

3.7 Підсистема моніторингу та AI-пошук природною мовою

Дві крізні підсистеми застосунку реалізують функціональність що не прив'язана до конкретної доменної сутності: моніторинговий огляд стану бази і природномовний пошук. Обидві реалізовані як окремі Django-додатки без власних

моделей бази даних.

Моніторинговий ендпоінт /api/monitoring/summary/ повертає агреговану панель показників системи що включає загальні кількості сутностей кожного типу (гурти, події, майданчики, міста), список п'яти найновіших гуртів за датою створення, перелік п'яти найближчих запланованих подій, останні п'ять записів журналу змін BandUpdateLog, топ-5 міст за кількістю гуртів та топ-5 жанрів за популярністю. Усі агрегації реалізовані засобами Django ORM через анотації Count і подальше сортування без додаткових моделей або міграцій.

Додаток ai_search реалізує природномовний пошук через чотири окремі модулі із розділеною відповідальністю. Перелік модулів наведено у таблиці 3.6.

Таблиця 3.6 – Модулі додатку ai_search

Модуль	**Призначення**
openrouter.py	Виклик зовнішнього сервісу OpenRouter і повернення сирої текстової відповіді мовної моделі
plan.py	Синтаксичний розбір відповіді як JSON і валідація проти білого списку дозволених полів
normalization.py	Перетворення україномовних значень у формат очікуваний існуючими ViewSets (міста, області, жанри)
executor.py	Виконання нормалізованого плану через стандартні ViewSets із перевикористанням логіки фільтрації

Передача завдання до мовної моделі здійснюється з системним промптом що інструктує модель повертати виключно JSON у заданій схемі без допоміжного тексту. Схема відповіді містить три поля: entity з одним із значень bands, events або venues, filters з об'єктом що може містити поля city, region, genre, status, search, upcoming, date_from, date_to, lat, lng і radius_km, та explanation з коротким реченням українською мовою про інтерпретацію запиту. Будь-яке поле що не входить до цього списку ігнорується на етапі валідації плану.

Модуль normalization вирішує практичну задачу узгодження україномовних

значень із форматом що очікують існуючі ViewSets. Наприклад мовна модель може повернути назву міста у різних формах: «Київ», «місто Київ» або «Kyiv». Модуль нормалізації зіставляє всі ці варіанти з відповідним записом у таблиці City і замінює їх на внутрішній slug «kyiv» що очікує фільтр ViewSet. Аналогічно нормалізуються назви областей і жанрів. Такий підхід робить природномовний пошук стійким до варіацій у формулюванні запитів користувача.

Модуль executor отримує нормалізований план і виконує відповідний запит через одну зі стандартних ViewSets перевикористовуючи всю існуючу логіку фільтрації, геопросторового пошуку і серіалізації. Це означає що результати природномовного пошуку є ідентичними за структурою результатам звичайного API-запиту з тими самими параметрами, що суттєво спрощує обробку відповіді на клієнтській стороні.

4 ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ВЕБЗАСТОСУНКУ

Тестування є обов'язковим етапом розробки програмного забезпечення що дозволяє підтвердити відповідність реалізованої системи поставленим функціональним і нефункціональним вимогам. Для системи з геопросторовою функціональністю тестування є особливо важливим оскільки коректність просторових обчислень не є очевидною і потребує верифікації незалежним методом.

У цьому розділі описано процес тестування і оцінювання розробленого застосунку. У підрозділі 4.1 наведено вимоги до середовища розгортання і порядок інсталяції. Підрозділ 4.2 описує методику і обсяг тестування. У підрозділі 4.3 наведено результати перевірки коректності геопросторових запитів. Підрозділ 4.4 містить опис інтерфейсу користувача і основні сценарії роботи з системою.

4.1 Вимоги до середовища та інсталяція

Розроблюваний застосунок складається з двох незалежних компонентів що розгортаються окремо: серверної частини на базі Django і клієнтської частини на базі React. Кожен компонент має власні залежності і порядок налаштування. Мінімальні вимоги до середовища розгортання наведено у таблиці 4.1.

Таблиця 4.1 – Мінімальні вимоги до середовища розгортання

Компонент	Мінімальна версія	Призначення
Python	3.12	Середовище виконання серверної частини
PostgreSQL	17	Сервер бази даних
PostGIS	3.4	Геопросторове розширення PostgreSQL
GDAL	3.6	Перетворення форматів просторових даних
GEOS	3.11	Геометричні операції
PROJ	9.0	Картографічні проекції
Node.js	18	Середовище виконання клієнтської частини
npm	9.0	Менеджер пакетів клієнтської частини

Для розгортання серверної частини необхідно виконати послідовність кроків. Спочатку встановлюється PostgreSQL 17 із розширенням PostGIS. На платформі Windows розширення встановлюється у складі дистрибутиву OSGeo4W що автоматично постачає сумісні версії бібліотек GDAL, GEOS і PROJ та налаштовує необхідні змінні середовища. На Linux-системах PostGIS встановлюється через стандартний пакетний менеджер командою apt install postgis. Після встановлення PostgreSQL створюється база даних і для неї активується розширення PostGIS командою CREATE EXTENSION postgis.

Далі у кореневій директорії серверної частини створюється файл .env із параметрами підключення до бази даних і ключем API для сервісу OpenRouter. Файл .env не включається до репозиторію системи контролю версій і має бути створений вручну на кожному середовищі розгортання. Після налаштування файлу .env встановлюються залежності Python із файлу requirements.txt через команду pip install -r requirements.txt у активованому віртуальному середовищі. Виконання міграцій командою python manage.py migrate створює всі необхідні таблиці бази даних включно з просторовими індексами GiST для полів типу geography.

Для початкового наповнення бази тестовими даними передбачено ідемпотентні seed-команди що можна виконувати повторно без ризику дублювання записів. Команда python manage.py seed_locations завантажує 22 області України і 22 обласних центри з географічними координатами. Команда seed_music_data завантажує 14 музичних жанрів і 8 тестових гуртів. Команди seed_venues і seed_events завантажують відповідно 6 концертних майданчиків і 6 подій у різних містах. Після виконання всіх seed-команд система готова до роботи і містить достатній обсяг даних для перевірки всіх функціональних сценаріїв.

Для розгортання клієнтської частини необхідна наявність Node.js версії 18 або вище. Залежності встановлюються командою npm install у директорії frontend. Для запуску сервера розробки використовується команда npm run dev що запускає Vite і робить клієнтську частину доступною на порту 5173. Для продуктивного розгортання виконується команда npm run build що компілює застосунок у статичні файли придатні для розміщення на будь-якому веб-сервері.

Взаємодія між клієнтською і серверною частинами при локальному розгортанні налаштовується через параметр CORS_ALLOWED_ORIGINS у файлі settings.py серверної частини. За замовчуванням дозволені запити з адреси http://localhost:5173 що відповідає адресі сервера розробки Vite. При розгортанні на продуктивному середовищі цей параметр має бути оновлений відповідно до реальної адреси клієнтської частини.

4.2 Методика та обсяг тестування

Тестування розробленого застосунку проводилось у локальному середовищі розробки на машині під керуванням операційної системи Windows із використанням PostgreSQL 17 та розширення PostGIS встановленого у складі дистрибутиву OSGeo4W. Тестування охоплювало перевірку коректності геопросторових запитів, валідацію вхідних параметрів, дотримання правил рольового розмежування доступу та відповідність часу відгуку нефункціональним вимогам.

Тестування виконувалось у двох режимах. Перший режим – автоматизоване тестування через вбудований браузерний інтерфейс Django REST Framework що дозволяє виконувати запити до API безпосередньо у браузері без додаткових інструментів. Головна сторінка API із переліком усіх зареєстрованих ендпоінтів наведена на рисунку 4.1. Другий режим – ручне тестування через клієнтську частину застосунку із перевіркою коректності відображення результатів на інтерактивній карті.

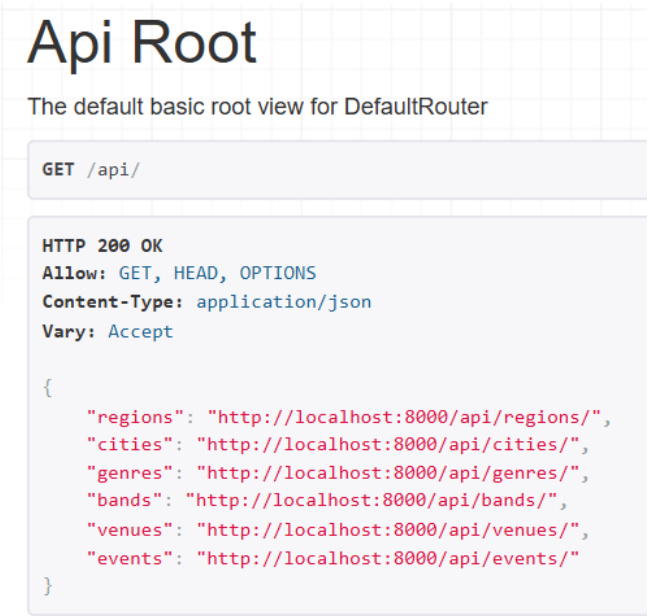

Рисунок 4.1 – Кореневий ендпоінт API із переліком зареєстрованих ресурсів

Для проведення тестування підготовлено початковий набір тестових даних що завантажується через ідемпотентні seed-команди. Цей набір включає 22 області України, 22 обласних центри з координатами, 14 музичних жанрів, 8 тестових гуртів різних жанрів і статусів, 6 концертних майданчиків у різних містах та 6 подій. Перелік тестових сценаріїв наведено у таблиці 4.2.

Таблиця 4.2 – Перелік тестових сценаріїв

Сценарій тестування	**Метод перевірки**	**Очікуваний результат**
Радіус-пошук гуртів за координатами	GET /api/bands/?lat=&lng=&radius_km=	Список гуртів відсортований за distance_km
Валідація некоректних параметрів пошуку	GET /api/bands/?lat=200&lng=30	HTTP 400 з описом помилки
Фільтрація за жанром і регіоном одночасно	GET /api/bands/?genre=rock&region=kyivska	Лише гурти що відповідають обом фільтрам
Реєстрація з підвищеною роллю	POST /api/auth/register/ з полем role=admin	Роль примусово встановлена як regular_user
Доступ анонімного користувача до POST	POST /api/bands/ без автентифікації	HTTP 401

Продовження таблиці 4.2

Доступ користувача без ролі до POST	POST /api/bands/ з regular_user	HTTP 403
Створення гурту менеджером	POST /api/bands/ з band_manager	Гурт створено, запис у BandManager
Редагування чужого гурту менеджером	PATCH /api/bands/{id}/ чужого гурту	HTTP 403
Журналювання змін атрибутів гурту	PATCH /api/bands/{id}/ з band_manager	Записи у BandUpdateLog
Природномовний пошук українською	POST /api/ai-search/ з текстовим запитом	JSON-план і результати пошуку
Фільтр upcoming для подій	GET /api/events/?upcoming=true	Лише події з датою пізніше поточної
Моніторинговий огляд системи	GET /api/monitoring/summary/	Агрегована статистика всіх сутностей

Тестування рольової моделі виконувалось через послідовне створення облікових записів із різними ролями і перевірку доступності операцій для кожної ролі. Зокрема перевірялась неможливість підвищення ролі через запит реєстрації, коректність відмови у доступі з HTTP-статусом 401 для неавтентифікованих користувачів і статусом 403 для автентифікованих користувачів із недостатніми правами. Перевірялась також автоматична поява запису у таблиці BandManager

після створення гурту менеджером і коректність записів у журналі BandUpdateLog після операцій редагування.

4.3 Перевірка коректності геопросторових запитів

Найкритичнішою функціональністю системи з технічної точки зору є геопросторовий радіус-пошук, оскільки некоректні результати просторових обчислень можуть бути непомітними для звичайного користувача але призводити до принципово хибних відповідей системи. Для верифікації коректності обчислень використано незалежний метод: результати запитів порівнювались із значеннями розрахованими вручну за формулою гаверсина що обчислює точну відстань між двома точками на поверхні сфери.

Формула гаверсина обчислює центральний кут між двома точками на сфері через проміжну величину hav що є квадратом синуса половини кутової відстані. Для двох точок із координатами (φ1, λ1) і (φ2, λ2) відстань d обчислюється як добуток радіуса Землі R на подвійний арксинус квадратного кореня з суми гаверсинів різниць широт і добутку косинусів широт на гаверсин різниці довгот. При радіусі Землі R = 6371 км формула дає результати що збігаються з обчисленнями PostGIS на типі geography з точністю до 0.1 км.

Виконано серію тестових запитів із різними центрами і радіусами пошуку. Для кожного запиту фіксувалась кількість знайдених гуртів і подій, а обчислені відстані порівнювались із ручними розрахунками за формулою гаверсина. Результат одного з тестових запитів із параметрами lat=50.4501, lng=30.5234 і radius_km=50 наведено на рисунку 4.2.

Рисунок 4.2 – Результат геопросторового запиту до API з параметрами радіус-пошуку

Зведені результати серії тестових запитів наведено у таблиці 4.3. Для кожного запиту вказано центр пошуку, радіус, кількість знайдених об'єктів і максимальне відхилення обчисленої відстані від еталонного значення за формулою гаверсина.

Окремо перевірено валідацію некоректних значень параметрів радіус-пошуку. Запит без одного з трьох обов'язкових параметрів (lat, lng, radius_km) коректно відхиляється з HTTP-статусом 400 і повідомленням про відсутній параметр. Запити зі значеннями широти поза діапазоном від -90 до +90 градусів, довготи поза діапазоном від -180 до +180 градусів і від'ємним або нульовим радіусом також відхиляються з відповідними повідомленнями про помилку. При активному радіус-пошуку всі відповіді містять поле distance_km і впорядковані за зростанням відстані що підтверджує коректність сортування результатів.

Таблиця 4.3 – Результати тестування радіус-пошуку

Центр пошуку	**Радіус (км)**	**Знайдено гуртів**	**Знайдено подій**	**Макс. відхилення (км)**
Київ (50.4501, 30.5234)	20	3	3	< 0.1
Київ (50.4501, 30.5234)	100	3	3	< 0.1
Львів (49.8397, 24.0297)	50	2	1	< 0.1
Одеса (46.4825, 30.7233)	150	1	1	< 0.1
Харків (49.9935, 36.2304)	200	1	0	< 0.1

Перевірено також коректність роботи радіус-пошуку у поєднанні з іншими фільтрами. Запит із одночасним застосуванням радіус-параметрів і фільтра за жанром повертає лише ті гурти що задовольняють обидва критерії одночасно, що підтверджує коректність послідовного застосування фільтрів до одного QuerySet. Запит із фільтром upcoming і радіус-пошуком повертає лише майбутні події у заданому радіусі відсортовані за відстанню.

За результатами тестування підтверджено що PostGIS обчислює відстані на типі geography з належною точністю. Розбіжності між результатами системи і ручними розрахунками за формулою гаверсина не перевищують 0.1 км для жодного з виконаних тестових запитів, що є прийнятним результатом з огляду на різницю між сферичною моделлю Землі що використовується у формулі гаверсина і еліпсоїдною моделлю WGS84 що використовується PostGIS.

4.4 Опис інтерфейсу користувача та сценарії роботи

Клієнтська частина застосунку реалізована як односторінковий React-застосунок із власною навігацією що включає сім основних сторінок: головну сторінку зі списком гуртів, сторінку детального опису гурту, сторінку подій, сторінку інтерактивної карти, сторінку моніторингу, сторінку природномовного пошуку та сторінку особистого кабінету користувача. Навігація між сторінками реалізована через компонент NavBar що відображається у верхній частині кожної сторінки і адаптує свій вміст залежно від ролі поточного користувача.

Головна сторінка застосунку є каталогом музичних гуртів із панеллю фільтрації у лівій частині екрану. Панель фільтрації дозволяє звужувати вибірку за регіоном, містом, жанром і статусом гурту. Результати фільтрації відображаються у вигляді карток що містять назву гурту, місто походження, основний жанр, статус і найближчу заплановану подію якщо вона є. Загальний вигляд головної сторінки наведено на рисунку 4.3.

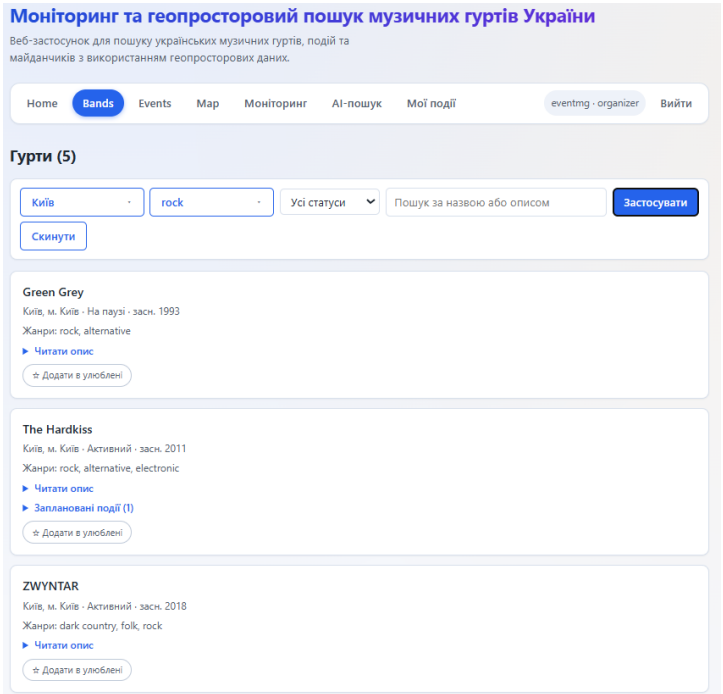

Рисунок 4.3 – Головна сторінка застосунку зі списком гуртів

Сторінка детального опису гурту відображає повну інформацію про виконавця: назву, опис, рік заснування, місто походження, повний список жанрів, посилання на офіційний сайт і сторінки у стримінгових сервісах та соціальних мережах, а також список усіх запланованих подій із датами і майданчиками. Для авторизованих користувачів на сторінці відображається кнопка додавання гурту до списку вподобань. Для затверджених менеджерів цього гурту додатково відображається форма редагування атрибутів.

Сторінка інтерактивної карти є найбільш характерним елементом системи і безпосередньо реалізує центральну функціональність геопросторового пошуку. На карті відображаються маркери гуртів, майданчиків або подій залежно від активного режиму перегляду що перемикається кнопками у верхній частині сторінки. При кліку на маркер з'являється спливаюче вікно з короткою інформацією про об'єкт і посиланням на його повну сторінку. Загальний вигляд сторінки карти наведено на рисунку 4.4.

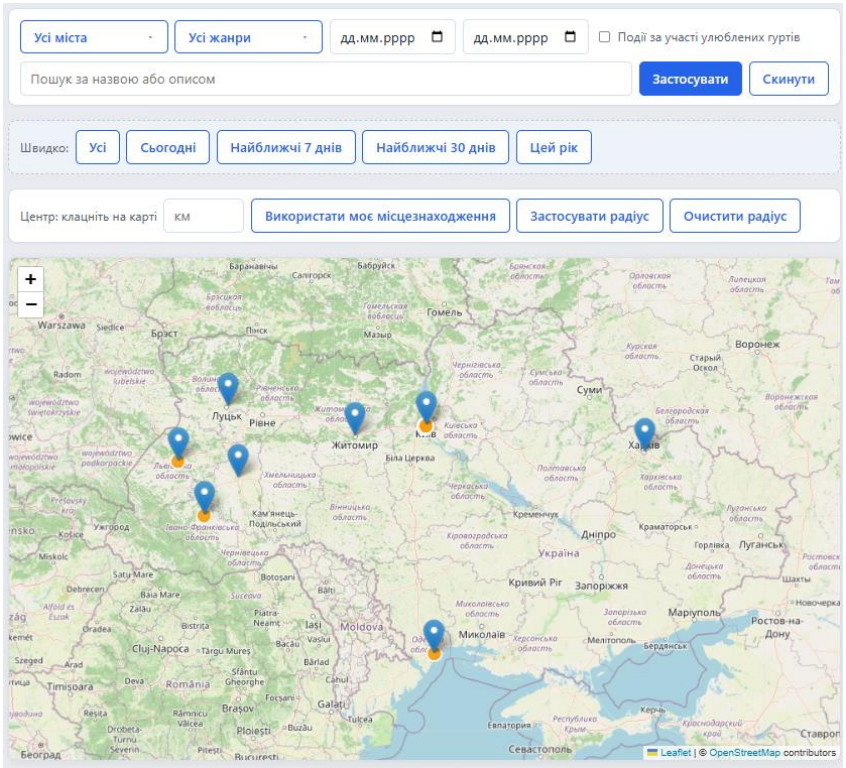

Рисунок 4.4 – Сторінка інтерактивної карти з маркерами гуртів

Кнопка «Знайти поблизу мене» викликає браузерний API navigator.geolocation для визначення поточних координат користувача і автоматично виконує радіус-пошук у межах заданого радіуса. Виявлена область пошуку візуально позначається на карті колом, центральний маркер показує точку відліку, а знайдені об'єкти відображаються відсортованими за зростанням відстані. Результат виконання радіус-пошуку на карті наведено на рисунку 4.5.

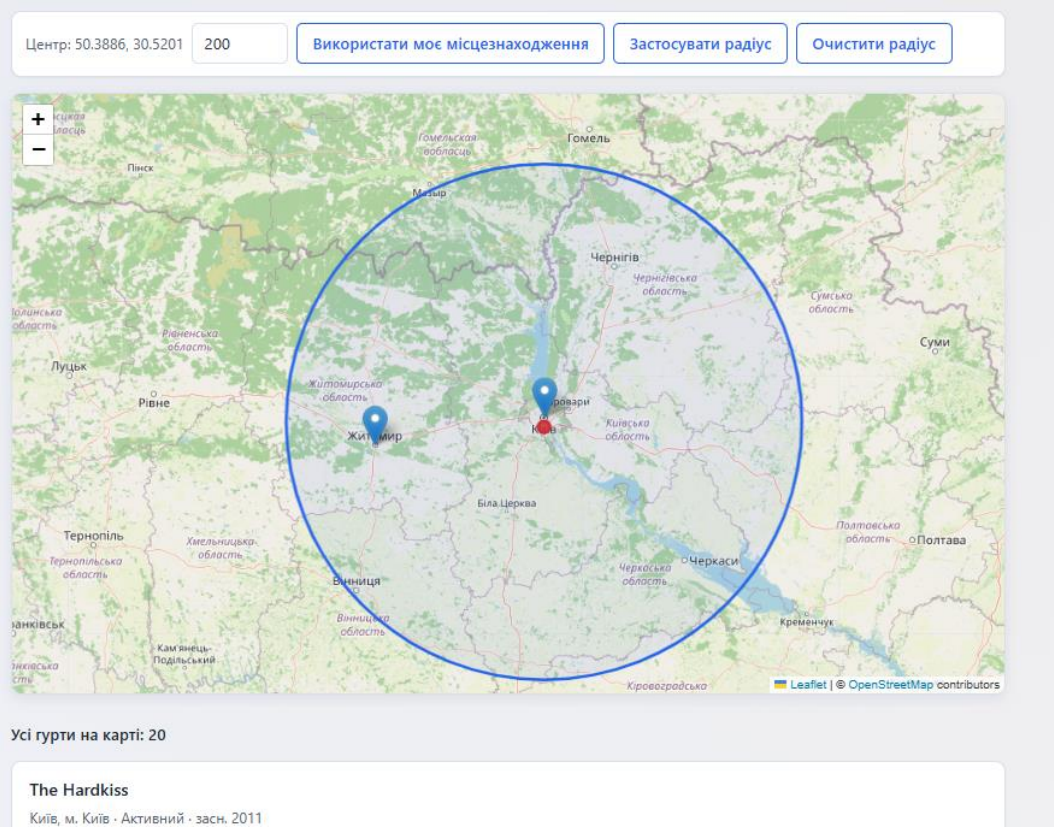

Рисунок 4.5 – Результат радіус-пошуку на карті з колом пошуку та маркерами

Сторінка моніторингу надає агреговану панель показників стану системи. У верхній частині сторінки розміщено чотири картки із загальною кількістю гуртів, подій, майданчиків і міст у базі. Нижче відображаються горизонтальні стовпчикові діаграми топ-5 міст за кількістю гуртів і топ-5 жанрів за популярністю, список п'яти найновіших гуртів і список п'яти найближчих запланованих подій. Вся візуалізація реалізована засобами чистого CSS без сторонніх бібліотек діаграм. Загальний

вигляд сторінки моніторингу наведено на рисунку 4.6.

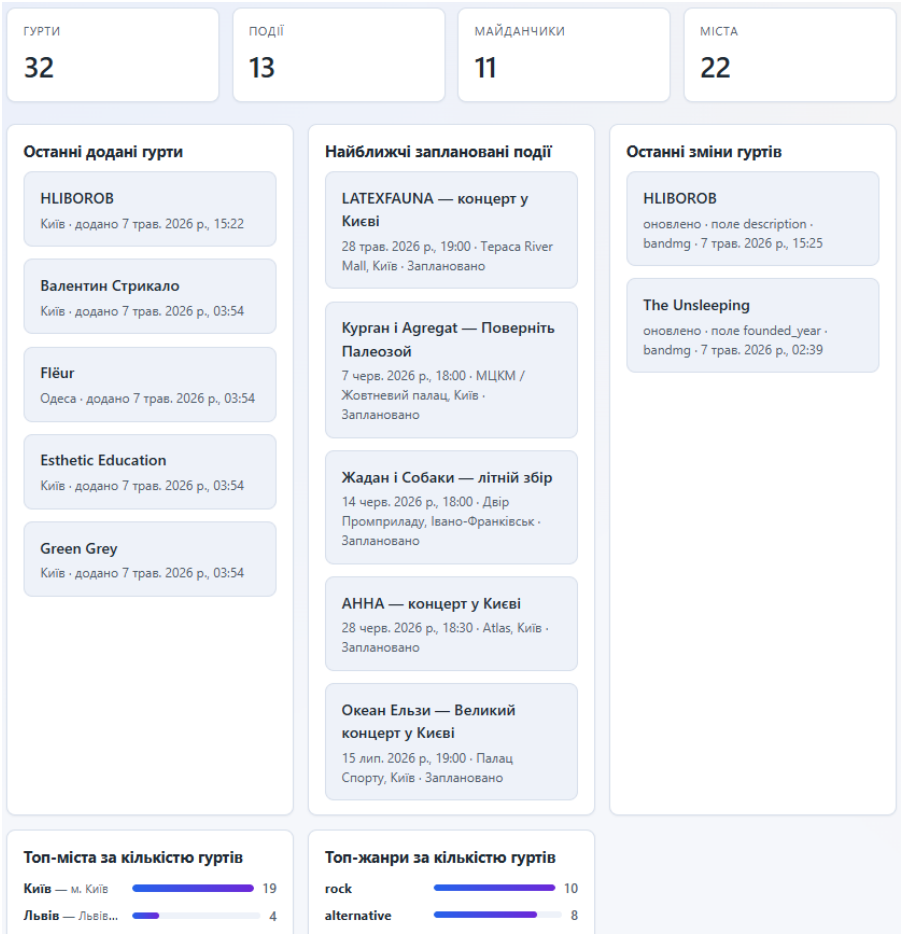

Рисунок 4.6 – Сторінка моніторингу з агрегованою статистикою системи

Сторінка природномовного пошуку надає текстове поле для введення запиту довільною українською мовою. Після відправки запиту система відображає інтерпретацію запиту мовною моделлю у вигляді короткого пояснення, сформований план фільтрації і результати пошуку у тому самому форматі що і звичайний каталог. Наприклад запит «рок-гурти зі Львова» інтерпретується як пошук гуртів із фільтрами genre=rock і city=lviv, а запит «концерти у Києві наступного місяця» перетворюється на пошук подій із фільтрами city=kyiv і відповідним діапазоном дат. Загальний вигляд сторінки AI-пошуку наведено на рисунку 4.7.

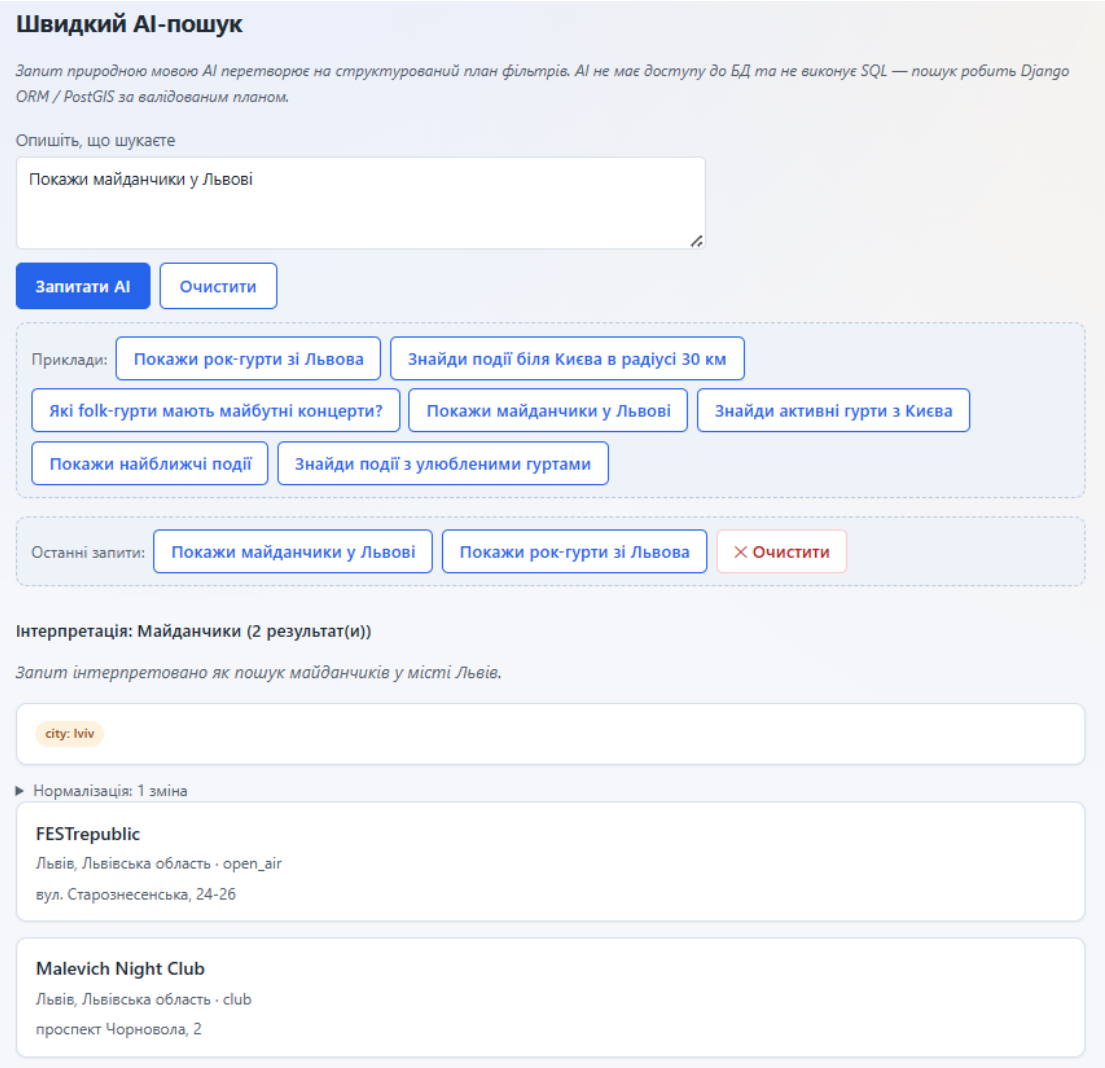

Рисунок 4.7 – Сторінка природномовного пошуку з результатами запиту

За результатами тестування підтверджено що інтерфейс забезпечує повний цикл взаємодії користувача з системою: від реєстрації та авторизації через перегляд каталогу і виконання геопросторових запитів до використання природномовного пошуку та перегляду агрегованої статистики. Усі функціональні вимоги сформульовані у підрозділі 1.1 у поточній реалізації виконані. Нефункціональні вимоги до часу відгуку основних API-запитів витримуються із значним запасом за рахунок ефективних просторових індексів і компактної структури JSON-відповідей.

ВИСНОВКИ

У процесі формулювання мети даної дипломної роботи було поставлено завдання розробити вебзастосунок що забезпечує централізований моніторинг та геопросторовий пошук інформації про українські музичні гурти, концертні майданчики й події з інтерактивною візуалізацією результатів на карті. Потенційний користувач системи може не мати жодних технічних знань: інтерфейс побудований таким чином щоб знайти концерт поблизу свого місцезнаходження можна було одним кліком на кнопку «Знайти поблизу мене», а природномовний пошук дозволяє формулювати запити звичайною українською мовою без заповнення форм із фільтрами.

Проведений аналіз існуючих платформ моніторингу музичних подій виявив характерну закономірність: світові платформи (Bandsintown, Songkick) мають розвинений каталог виконавців але не реалізують геопросторового радіус-пошуку і погано покривають український ринок, тоді як вітчизняні афіші (Karabas, Concert.ua) орієнтовані на продаж квитків і не виокремлюють виконавців у структуроване ядро каталогу. Жодне з розглянутих рішень не поєднує повноцінного україномовного каталогу з геопросторовим пошуком і відкритим API, що обґрунтовує доцільність розробки власного рішення.

Обґрунтовано вибір технологічного стеку Python 3.12 і Django 5.2 із розширенням GeoDjango як серверної основи, PostgreSQL 17 із PostGIS як бази даних із підтримкою просторових типів, React 18 і Leaflet як клієнтської частини. Визначальним аргументом на користь цього стеку є унікальна інтеграція GeoDjango з PostGIS що дозволяє виконувати геопросторові запити на еліпсоїді WGS84 засобами стандартного ORM без написання сирого SQL.

Спроектовано нормалізовану до третьої нормальної форми схему реляційної БД із дванадцяти таблиць. Для сутностей із географічною прив'язкою (City і Venue) визначено просторові поля типу geography із системою координат SRID 4326, для яких Django-міграції автоматично створюють просторові індекси GiST що

забезпечують логарифмічну складність просторових запитів.

Реалізовано серверну логіку геопросторового радіус-пошуку через ORM-локап __dwithin з обчисленням відстаней на поверхні еліпсоїда WGS84 і сортуванням результатів за зростанням відстані. Розроблено REST API із комбінованою фільтрацією за множиною критеріїв, чотирирівневою рольовою моделлю доступу і автоматичним журналом аудиту змін атрибутів гуртів у таблиці BandUpdateLog. Інтеграцію з мовною моделлю через сервіс OpenRouter реалізовано за принципом «ШІ як парсер намірів»: модель повертає виключно типізований JSON-план фільтрації що виконується через існуючі ViewSets, що принципово виключає можливість SQL-ін'єкцій з боку мовної моделі.

Розроблено односторінковий React-застосунок із інтерактивною картою на базі Leaflet, сторінкою агрегованої статистики і інтерфейсом природномовного пошуку. Тестування підтвердило коректність геопросторових обчислень: розбіжності між результатами системи і незалежними розрахунками за формулою гаверсина не перевищують 0.1 км, а всі функціональні і нефункціональні вимоги сформульовані у постановці задачі виконані.

Перспективами подальшого розвитку системи є розширення доменної моделі через включення фестивалів і турів, реалізація персоналізованих рекомендацій на основі історії взаємодії користувачів, інтеграція із зовнішніми музичними API для автоматичного імпорту метаданих виконавців, масштабування геопросторової складової до рівня окремих закладів у межах міста, а також розробка мобільного клієнта. У поточній реалізації застосунок повністю покриває поставлені функціональні вимоги і може використовуватись як основа для подальшого розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Silberschatz A., Korth H. F., Sudarshan S. Database System Concepts. 7th ed. New York : McGraw-Hill Education, 2019. 1376 p.
2. Elmasri R., Navathe S. B. Fundamentals of Database Systems. 7th ed. Boston : Pearson, 2016. 1242 p.
3. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O'Reilly Media, 2017. 616 p.
4. Longley P. A., Goodchild M. F., Maguire D. J., Rhind D. W. Geographic Information Science and Systems. 4th ed. Hoboken : Wiley, 2015. 496 p.
5. Worboys M., Duckham M. GIS: A Computing Perspective. 2nd ed. Boca Raton : CRC Press, 2004. 448 p.
6. Fu P., Sun J. Web GIS: Principles and Applications. Redlands : Esri Press, 2010. 312 p.
7. Shekhar S., Chawla S. Spatial Databases: A Tour. Upper Saddle River : Prentice Hall, 2003. 298 p.
8. Obe R. O., Hsu L. S. PostGIS in Action. 3rd ed. Shelter Island : Manning Publications, 2021. 624 p.
9. de Smith M. J., Goodchild M. F., Longley P. A. Geospatial Analysis: A Comprehensive Guide. 6th ed. Leicester : Winchelsea Press, 2018. 764 p.
10. Rubio D. Beginning Django: Web Application Development and Deployment with Python. Berkeley : Apress, 2017. 593 p.
11. Coronel C., Morris S. Database Systems: Design, Implementation and Management. 13th ed. Boston : Cengage Learning, 2018. 800 p.
12. Greenfeld D., Greenfeld A. R. Two Scoops of Django 3.x: Best Practices for the Django Web Framework. San Diego : Two Scoops Press, 2020. 530 p.
13. Percival H., Gregory B. Architecture Patterns with Python. Sebastopol : O'Reilly Media, 2020. 280 p.
14. Lutz M. Learning Python. 5th ed. Sebastopol : O'Reilly Media,

2013. 1648 p.

15. Ramalho L. Fluent Python. 2nd ed. Sebastopol : O'Reilly Media, 2022. 1012 p.

16. Date C. J. An Introduction to Database Systems. 8th ed. Boston : Pearson, 2004. 1024 p.

17. Richardson L., Amundsen M., Ruby S. RESTful Web APIs. Sebastopol : O'Reilly Media, 2013. 408 p.

18. Masse M. REST API Design Rulebook. Sebastopol : O'Reilly Media, 2011. 116 p.

19. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : diss. Irvine : University of California, 2000. 162 p.

20. Stefanov S. React: Up and Running. 2nd ed. Sebastopol : O'Reilly Media, 2021. 222 p.

21. Banks A., Porcello E. Learning React. 2nd ed. Sebastopol : O'Reilly Media, 2020. 310 p.

22. Larsen A. Getting Started with React. Birmingham : Packt Publishing, 2017. 284 p.

23. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p.

24. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.

25. Newman S. Building Microservices. 2nd ed. Sebastopol : O'Reilly Media, 2021. 616 p.

26. Wilber K. Designing Web APIs. Sebastopol : O'Reilly Media, 2018. 234 p.

27. Madden N. API Security in Action. Shelter Island : Manning Publications, 2020. 568 p.

28. Osherove R. The Art of Unit Testing. 2nd ed. Shelter Island : Manning Publications, 2013. 296 p.

29. Percival H. Test-Driven Development with Python. 2nd ed. Sebastopol : O'Reilly Media, 2017. 614 p.

30. Slocum T. A., McMaster R. B., Kessler F. C., Howard H. H. Thematic Cartography and Geovisualization. 3rd ed. Upper Saddle River : Pearson, 2009. 576 p.

31. MacEachren A. M. How Maps Work: Representation, Visualization, and Design. New York : Guilford Press, 2004. 513 p.

32. Tunstall L., von Werra L., Wolf T. Natural Language Processing with Transformers. Sebastopol : O'Reilly Media, 2022. 406 p.

33. Rothman D. Transformers for Natural Language Processing. 2nd ed. Birmingham : Packt Publishing, 2022. 600 p.

ДОДАТОК А

Реалізація моделей бази даних застосунку

Програмні засоби:

– мова програмування Python 3.12;

– фреймворк Django 5.2 з розширенням GeoDjango;

– СУБД PostgreSQL 17 з розширенням PostGIS

Програмний код файлу bands/models.py:

```
from django.db import models


class Status(models.TextChoices):
    ACTIVE = 'active', 'Active'
    ON_HIATUS = 'on_hiatus', 'On Hiatus'
    DISBANDED = 'disbanded', 'Disbanded'


class ChangeType(models.TextChoices):
    CREATED = 'created', 'Created'
    UPDATED = 'updated', 'Updated'
    STATUS_CHANGED = 'status_changed', 'Status Changed'
    DELETED = 'deleted', 'Deleted'


class Genre(models.Model):
    name = models.CharField(max_length=80, unique=True)
    slug = models.SlugField(max_length=100, unique=True)
    description = models.TextField(blank=True)

    class Meta:
        ordering = ['name']
```

```
        def __str__(self):
            return self.name


    class Band(models.Model):
        name = models.CharField(max_length=150)
        slug = models.SlugField(max_length=180, unique=True)
        city = models.ForeignKey(
            'locations.City',
            on_delete=models.PROTECT,
            related_name='bands',
        )
        status = models.CharField(
            max_length=20,
            choices=Status.choices,
            default=Status.ACTIVE,
        )
        founded_year     =     models.PositiveSmallIntegerField(null=True,
blank=True)
        disbanded_year    =    models.PositiveSmallIntegerField(null=True,
blank=True)
        description = models.TextField(blank=True)
        website = models.URLField(blank=True)
        social_links = models.JSONField(default=dict, blank=True)
        instagram_url = models.URLField(blank=True)
        spotify_url = models.URLField(blank=True)
        apple_music_url = models.URLField(blank=True)

        genres = models.ManyToManyField(
            Genre,
            through='BandGenre',
            related_name='bands',
```

```
    )

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['name']
        constraints = [
            models.UniqueConstraint(
                fields=['name', 'city'],
                name='unique_band_per_city',
            ),
        ]
        indexes = [
            models.Index(fields=['status']),
            models.Index(fields=['-created_at']),
            models.Index(fields=['status', 'city']),
        ]

    def __str__(self):
        return self.name


class BandGenre(models.Model):
    band = models.ForeignKey(
        Band,
        on_delete=models.CASCADE,
        related_name='band_genres',
    )
    genre = models.ForeignKey(
        Genre,
        on_delete=models.PROTECT,
        related_name='genre_bands',
```

```
    )
    is_primary = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        constraints = [
            models.UniqueConstraint(
                fields=['band', 'genre'],
                name='unique_band_genre',
            ),
        ]

    def __str__(self):
        return f'{self.band.name} – {self.genre.name}'


class BandManager(models.Model):
    user = models.ForeignKey(
        'accounts.UserProfile',
        on_delete=models.CASCADE,
        related_name='managed_bands',
    )
    band = models.ForeignKey(
        Band,
        on_delete=models.CASCADE,
        related_name='managers',
    )
    is_approved = models.BooleanField(default=False)
    assigned_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        constraints = [
            models.UniqueConstraint(
```

```
                fields=['user', 'band'],
                name='unique_user_band_manager',
            ),
        ]
        indexes = [
            models.Index(fields=['user', 'is_approved']),
        ]

    def __str__(self):
        return f'{self.user} → {self.band.name}'


class FavoriteBand(models.Model):
    """A user's favorite band. Used to power the `?favorite_bands=true`
    event filter and the per-card favorite toggle in the UI."""

    user_profile = models.ForeignKey(
        'accounts.UserProfile',
        on_delete=models.CASCADE,
        related_name='favorite_bands',
    )
    band = models.ForeignKey(
        Band,
        on_delete=models.CASCADE,
        related_name='favorited_by',
    )
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        constraints = [
            models.UniqueConstraint(
                fields=['user_profile', 'band'],
                name='unique_user_favorite_band',
```

```
                ),
            ]
            indexes = [
                models.Index(fields=['user_profile', '-created_at']),
            ]

        def __str__(self):
            return f'{self.user_profile} ★ {self.band.name}'


    class BandUpdateLog(models.Model):
        band = models.ForeignKey(
            Band,
            on_delete=models.CASCADE,
            related_name='update_logs',
        )
        changed_by = models.ForeignKey(
            'accounts.UserProfile',
            on_delete=models.SET_NULL,
            null=True,
            blank=True,
            related_name='band_changes',
        )
        change_type           =           models.CharField(max_length=20,
choices=ChangeType.choices)
        field_name = models.CharField(max_length=64, blank=True)
        old_value = models.TextField(blank=True)
        new_value = models.TextField(blank=True)
        created_at = models.DateTimeField(auto_now_add=True)

        class Meta:
            ordering = ['-created_at']
            indexes = [
```

```
                models.Index(fields=['band', '-created_at']),
                models.Index(fields=['change_type']),
            ]

        def __str__(self):
            return      f'{self.band.name}      {self.change_type}      @
{self.created_at:%Y-%m-%d}'
```