Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Інституту комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(рівень вищої освіти)

на тему: «Розроблення програмного забезпечення для інтелектуального аналізу даних гуртків для дітей засобами React JS»
(тема роботи)

Виконав: студент 4 курсу групи КН-41
Спеціальності 122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Лапчук Олександр Володимирович
(прізвище та ініціали)

Керівник Нечепуренко А.В., Яцишин С.І.
(прізвище та ініціали)

Рецензент Часковський О.Г.
(прізвище та ініціали)

Львів-2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра: комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 "Комп'ютерні науки"

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
Борецька І.Б.

"10" червня 2025р.

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Лапчук Олександр Володимирович
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення програмного забезпечення для інтелектуального аналізу даних гуртків для дітей засобами React JS / Software development for intelligent data analysis of children's clubs using React JS
керівник роботи *Яцишин С.І., к.т.н, доц, Нечепуренко А.В. ст. викл.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "15" листопада 2024 року №С-882
2. Термін подання студентом роботи 10 червня 2025 р.
3. Вихідні дані до роботи Розробити програмне забезпечення для централізованого збирання, збереження та представлення інформації про дитячі гуртки. Система має об'єднувати дані з різних джерел за допомогою парсерів та забезпечувати зручний інтерфейс користувача на базі React JS для перегляду, фільтрації та пошуку гуртків за різними критеріями (напрям, місцезнаходження тощо).
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
ВСТУП
РОЗДІЛ 1. Стан проблемної області
РОЗДІЛ 2. Інформаційне та математичне забезпечення
РОЗДІЛ 3. Програмне та технічне забезпечення
ВИСНОВКИ
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Підготовка матеріалу до доповіді .
6. Дата видачі завдання 18 листопада 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Збір та опрацювання матеріалу за темою дипломної роботи	18.11.2024-26.11.2024	Виконано
2	Проєктування програми	26.11.2024-14.12.2024	Виконано
3	Розробка програми	14.12.2024-26.02.2024	Виконано
4	Тестування програми	26.02.2024-20.03.2025	Виконано
5	Внесення правок у програму	20.03.2025-30.03.2025	Виконано
6	Розробка першого розділу пояснювальної записки	30.03.2025-20.04.2025	Виконано
7	Розробка другого розділу пояснювальної записки	20.04.2025-25.05.2025	Виконано
8	Розробка третього розділу пояснювальної записки	25.05.2025-05.06.2025	Виконано
9	Оформлення пояснювальної записки	05.06.2025-10.06.2025	Виконано

Студент _____ **Лапчук О.В.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Нечепуренко А.В.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Яцишин С.І.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

У дипломній роботі описано процес розроблення програмного забезпечення для централізованого збирання та відображення даних про дитячі гуртки. Основна увага приділена побудові системи, яка включає парсери для збору інформації з різних онлайн-ресурсів, об'єднання отриманих даних у спільну базу та створення повноцінного вебзастосунку. Фронтенд реалізовано засобами React JS, а серверну частину — з використанням Python FastAPI. Для зберігання даних використовується PostgreSQL, а для кешування та підвищення швидкодії — Redis. Розглянуто питання структурування даних, синхронізації з джерелами, побудови логіки фільтрації та пошуку. Вебінтерфейс забезпечує доступність інформації для широкого кола користувачів, включно з батьками та адміністраторами закладів. Систему протестовано на коректність роботи, оцінено її ефективність, масштабованість та зручність використання. ***Ключові слова****: дитячі гуртки, вебзастосунок, React JS, FastAPI, PostgreSQL, Redis, парсери, збирання даних.*

ABSTRACT

This diploma thesis presents the development of software for the centralized collection and display of data on children's clubs. The system integrates parsers to gather information from various online sources, unifies the collected data into a shared database, and delivers a full-featured web application. The frontend is developed using React JS, while the backend is implemented with Python FastAPI. PostgreSQL is used for data storage, and Redis is applied for caching and performance optimization. The work addresses data structuring, source synchronization, and logic for filtering and search. The web interface ensures accessible and user-friendly interaction for both parents and institution administrators. The system was tested for reliability, and its scalability and usability were evaluated. ***Keywords:*** *children's clubs, web application, React JS, FastAPI, PostgreSQL, Redis, parsers, data collection.*

ТЕХНІЧНЕ ЗАВДАННЯ

У рамках дипломної роботи необхідно реалізувати програмне забезпечення, яке забезпечує комплексне збирання, обробку, зберігання та представлення інформації про дитячі гуртки в межах одного міста. Основною метою системи є створення єдиної цифрової платформи, що агрегує дані з розрізнених онлайн-джерел, структурує їх і надає кінцевому користувачу у зручному для сприйняття інтерфейсі.

У межах цього завдання необхідно створити низку програмних модулів, що взаємодіють між собою у рамках єдиної архітектури. Ключовим етапом проєкту є реалізація механізму парсингу — програмного інструменту, що забезпечує щоденне вилучення структурованої інформації про гуртки з різноманітних онлайн-джерел. Збір має виконуватись автоматично за заданим розкладом із використанням інструментів асинхронного планування на мові Python. Самі парсери повинні ґрунтуватись на бібліотеках BeautifulSoup для обробки HTML-структур, httpx для асинхронної роботи з HTTP-запитами, а також Pydantic для моделювання та валідації отриманих даних.

Після отримання первинної інформації про гуртки необхідно реалізувати геолокаційну обробку адрес. За допомогою інтеграції з сервісами Google Maps API система повинна отримувати точні координати гуртків, а також визначати відповідні райони міста Львова, до яких належать вказані адреси. Отримані геодані повинні використовуватися як для візуалізації об'єктів на інтерактивній карті, так і для подальшого фільтрування за місцем розташування. З метою мінімізації кількості зовнішніх запитів та підвищення швидкодії, передбачити впровадження системи кешування за допомогою Redis, яка дозволить зберігати раніше оброблені адреси та результати геолокації, уникаючи дублювання обчислень.

Зібрана та опрацьована інформація має зберігатися у централізованій базі даних, створеній на основі PostgreSQL. Структура бази повинна включати сутності для опису гуртків, їхніх категорій, галерей, місць проведення (departments),

контактних відомостей та інших пов’язаних характеристик. Доцільно передбачити також додаткові таблиці, що дозволять реалізувати розширену фільтрацію й категоризацію даних у фронтенді.

Для доступу до збереженої інформації потрібно розробити серверну частину застосунку, з використанням вебфреймворку FastAPI. Ця частина програмного забезпечення повинна надавати RESTful API, яке дозволятиме клієнтам отримувати дані про гуртки, зображення з галерей, адресну та контактну інформацію. Окрім того, має бути реалізована система централізованого логування подій і помилок, що забезпечить відстеження роботи кожного з компонентів і дозволить оперативно виявляти збої в процесі збору чи обробки інформації. Уся серверна частина повинна бути розгорнута в ізольованому середовищі Docker, із окремими контейнерами для FastAPI, PostgreSQL та Redis, що гарантує масштабованість та стабільність системи.

Користувацький інтерфейс реалізувати у вигляді односторінкового застосунку (SPA), створеного за допомогою бібліотеки React у поєднанні з TailwindCSS. Інтерфейс має забезпечувати зручну навігацію каталогом гуртків, відображення детальної інформації про кожен об’єкт, підтримку пагінації, фільтрацію за напрямками та районами міста. Особливу роль у проєкті буде відігравати інтерактивна мапа, вбудована через Google Maps API, яка візуалізує географічне розташування гуртків і підвищує зручність взаємодії з інформацією.

Додатково, система має включати засоби для запобігання дублюванню даних, передбачити валідацію вхідних значень, гнучкість у масштабуванні та можливість подальшого розширення на інші регіони або види позашкільної активності. Очікується, що після реалізації програмного забезпечення буде проведене повноцінне тестування компонентів з метою перевірки їхньої стабільності, ефективності та відповідності поставленим вимогам.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) — програмний інтерфейс прикладного програмування

HTML (HyperText Markup Language) — мова гіпертекстової розмітки

HTTP (HyperText Transfer Protocol) — протокол передавання гіпертексту

SPA (Single Page Application) — односторінковий вебзастосунок

UI (User Interface) — інтерфейс користувача

UX (User Experience) — користувацький досвід

DB (Database) — база даних

JSON (JavaScript Object Notation) — текстовий формат обміну даними

REST (Representational State Transfer) — стиль архітектури вебсервісів

ID (Identifier) — ідентифікатор

SQL (Structured Query Language) — мова структурованих запитів

ORM (Object-Relational Mapping) — об’єктно-реляційне відображення, технологія для взаємодії з базою даних через об’єктну модель

JS (JavaScript) — мова сценаріїв для веброзробки

Redis (Remote Dictionary Server) — система зберігання даних типу "ключ-значення"

FastAPI — сучасний асинхронний вебфреймворк на Python

PostgreSQL — об’єктно-реляційна система управління базами даних

Docker — платформа для контейнеризації застосунків

TailwindCSS — утилітарна CSS-бібліотека для стилізації інтерфейсів

Google Maps API — інтерфейс для роботи з картографічними сервісами Google

Pydantic — бібліотека для створення схем перевірки даних у Python

BeautifulSoup — бібліотека Python для парсингу HTML та XML

httpx — асинхронна бібліотека HTTP-клієнта для Python

ВСТУП

Сучасне інформаційне суспільство вимагає все більшої цифровізації у сфері позашкільної освіти, зокрема в частині надання доступу до актуальної інформації про дитячі гуртки та секції. У великих містах, таких як Львів, існує безліч розрізнених ресурсів, що містять відомості про позашкільні освітні ініціативи, однак відсутність єдиної централізованої платформи значно ускладнює пошук, порівняння та вибір гуртків для батьків та дітей. Відтак виникає потреба в розробці автоматизованої системи, яка б дозволяла об'єднувати дані з різних джерел, структурувати їх, доповнювати геолокаційною інформацією та надавати користувачам у зручному вебформаті. Впровадження такого рішення сприятиме покращенню доступу до освітніх можливостей, а також підвищить прозорість та ефективність комунікації між організаторами гуртків і потенційними учасниками.

Об’єктом дослідження є процес організації, агрегації та подання інформації про дитячі гуртки у цифровому форматі.

Метою роботи є створення веборієнтованого програмного забезпечення, що забезпечує автоматичний збір даних з відкритих джерел, обробку адресної інформації, географічну класифікацію гуртків, зберігання даних у централізованій базі та їх подання через зручний інтерфейс.

Предметом дослідження є програмні засоби, бібліотеки та технології, які забезпечують реалізацію функціональних компонентів системи: зокрема, засоби асинхронного парсингу на Python, використання FastAPI для побудови серверної частини, PostgreSQL для управління даними, Redis для кешування та React з TailwindCSS для створення клієнтського інтерфейсу.

Практична значущість роботи полягає у розробці готової до використання системи, яка дозволяє зменшити фрагментованість інформації про позашкільні заклади, автоматизує збір і оновлення даних, спрощує пошук гуртків за ключовими параметрами та візуалізує їх розташування за допомогою інтерактивної карти.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Огляд проблемної області

У сучасних умовах швидкого розвитку інформаційних технологій спостерігається тенденція до цифровізації практично всіх сфер суспільного життя, включаючи освітню галузь. Однак незважаючи на активне впровадження цифрових рішень у формальну освіту, позашкільний освітній простір залишається недостатньо охопленим у цьому контексті. Зокрема, інформація про дитячі гуртки, секції та інші форми дозвілля для школярів часто розміщується на окремих, несистематизованих сайтах, сторінках у соціальних мережах або у вигляді оголошень, що значно ускладнює процес пошуку та порівняння варіантів.

У містах з великою кількістю закладів та варіантів позашкільної зайнятості, таких як Львів, батьки змушені витрачати значну кількість часу на ручний моніторинг розрізнених джерел. Це створює додаткові труднощі у прийнятті рішень, особливо коли йдеться про зіставлення за такими параметрами, як вік дитини, напрям підготовки, географічне розташування гуртка, наявність вільних місць тощо. Відсутність централізованого ресурсу унеможливлює комплексне охоплення ринку освітніх послуг для дітей, що, у свою чергу, знижує доступність важливої інформації для широкої аудиторії.

Ще однією актуальною проблемою є недостатня інтеграція геолокаційної інформації. Більшість доступних джерел подає лише назву вулиці або закладу без прив’язки до координат чи районів міста, що робить неможливим ефективне візуальне представлення даних на карті. У свою чергу, така прив’язка є критично важливою при виборі гуртків, особливо для сімей, які шукають найближчі до місця проживання заклади.

Окрему складність становить підтримання актуальності інформації. Оскільки більшість джерел оновлюється вручну або нерегулярно, користувачі часто стикаються з застарілими даними, що призводить до втрати часу і довіри до

ресурсів. Автоматизовані засоби збору й оновлення інформації наразі майже не застосовуються у цій сфері.

Враховуючи вищезазначене, постає об’єктивна потреба у створенні системи, яка б вирішувала проблеми фрагментованості, неструктурованості та обмеженої доступності інформації про дитячі гуртки. Така система має не лише централізувати дані, а й забезпечити їх регулярне оновлення, інтерактивну візуалізацію, а також зручну навігацію для користувачів. Це дозволить спростити процес пошуку освітніх можливостей, підвищити ефективність вибору гуртків для дітей і водночас створити технологічне підґрунтя для подальшої цифрової трансформації позашкільного освітнього простору.

1.2 Аналіз існуючих рішень та аналогічних систем

У сфері подання інформації про дитячі гуртки наразі відсутня універсальна система, що забезпечувала б централізований, актуальний і зручний для користувача доступ до всіх доступних позашкільних ініціатив міста. Натомість існує низка розрізнених онлайн-ресурсів, які частково вирішують це завдання, однак мають низку суттєвих обмежень.

Найпоширенішими є офіційні вебсайти окремих навчальних закладів, центрів творчості або спортивних шкіл. Вони, як правило, містять базову інформацію про перелік гуртків, вартість, розклад і контактні дані. Проте такі сайти зазвичай не підтримують сучасні засоби фільтрації, не мають інтерактивної карти та не дозволяють порівнювати гуртки між різними закладами. Крім того, оновлення інформації на таких ресурсах виконується вручну адміністраторами закладів, що часто призводить до застарілих або неповних даних.

Деякі локальні або комерційні сервіси намагаються агрегувати пропозиції гуртків на своїх платформах. Здебільшого вони орієнтовані на рекламну складову, де заклади самостійно додають інформацію про свої послуги. У таких випадках актуальність і точність даних повністю залежать від активності користувачів або адміністраторів сервісу. Часто відсутня нормалізована структура даних — опис, вік

дітей, адреса, район, години роботи можуть бути представлені в довільній формі, що унеможливлює коректну фільтрацію та обробку.

Окрему категорію становлять сторінки в соціальних мережах, де поширюється інформація про нові гуртки або набори. Хоча ці платформи мають широкий охоплення аудиторії, вони не є структурованими джерелами — ускладнений пошук, немає фіксованої бази даних або фільтрів за параметрами, і дані швидко втрачаються серед іншого контенту.

Таким чином, жодне з наявних рішень не забезпечує повної, системної, актуальної та інтерактивної моделі подання інформації про дитячі гуртки в межах одного міста. Це створює інформаційний вакуум, у якому кінцевий користувач змушений самостійно збирати, порівнювати та перевіряти дані з багатьох джерел. Відсутність єдиного агрегатора із сучасним вебінтерфейсом, геолокаційною підтримкою та автоматизованим оновленням інформації свідчить про необхідність створення нової системи, яка б відповідала актуальним технологічним вимогам і потребам громади.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення для агрегації, обробки та візуалізації інформації про дитячі гуртки потребує чіткого визначення обсягів і структури вхідних даних, правил їх трансформації, логіки фільтрації та математичних принципів, що забезпечують коректну обробку, класифікацію та подання результатів. На цьому етапі формуються вимоги до форм представлення інформації, визначаються алгоритми геолокаційної обробки, оптимізації пошуку й об'єднання дублікатів, а також методи реалізації фільтраційних механізмів, необхідних для зручного використання системи кінцевим користувачем.

2.1 Інформаційне забезпечення системи

Інформаційне забезпечення охоплює сукупність структурованих і неструктурованих даних, що використовуються для побудови бази знань про доступні гуртки. Основними джерелами є відкриті інтернет-ресурси закладів, платформ та публічних списків. Дані, які вилучаються під час парсингу, містять такі атрибути: назва гуртка, категорія або напрям діяльності, опис занять, адреса проведення, контактна інформація, дні та час роботи, а також зображення для формування галереї.

schemas.py

```python
class SCompany(BaseModel):
    link: str | None = None
    name: str | None = None
    org_id: str | None = None
    preview_image: str | None = None
    departments: list[SDepartment] = []
    description: str | None = ""
    carousel: list[SCarouselImage] = []
    categories: list[str] = []
```

Рисунок 2.1 – Схема атрибутів гуртка

Для зручності роботи ці дані нормалізуються у вигляді Pydantic-схем, що відповідають обраній структурі моделі. Після цього до адреси застосовується геолокаційна обробка з використанням Google Maps API, яка повертає координати та назву району міста Львова. Всі унікальні записи зберігаються в базі даних PostgreSQL. Інформаційне забезпечення також включає кеш-пам’ять Redis, у якій зберігаються повторно використовувані результати запитів до геосервісів, що знижує навантаження та прискорює обробку.

```
location_info = await location_resolver.resolve_location(address)
```

Рисунок 2.2 – Приклад виклику методу для отримання інформації про адресу використовуючи власний сервіс

2.2 Математичне забезпечення геолокаційної обробки та фільтрації

Математичне забезпечення системи базується на алгоритмах аналізу адресних даних і прив’язки об’єктів до координатної системи. Для кожної адреси виконується геокодування, в процесі якого визначаються широта й довгота. Отримані координати використовуються для побудови інтерактивної карти з маркерами, а також для групування гуртків за районами міста.

Окрім геолокації, важливим завданням є фільтрація. Вона реалізується на основі комбінаційних запитів до бази даних із врахуванням таких параметрів, як напрям гуртка (категорія), район, а також ключові слова у назві чи описі. Алгоритм передбачає багаторівневу логіку фільтрації, де умови можуть поєднуватись як за логікою "і", так і "або", забезпечуючи гнучкий підбір результатів.

```python
"""Build the main query for getting company details with pagination."""
query = (
    select(
        Company.name,
        Company.link,
        Company.org_id,
        func.coalesce(categories_subquery.c.category_names, []).label(
            "categories"
        ),
        Company.description,
        Company.preview_image,
        department_subquery.c.id,
        department_subquery.c.company_id,
        department_subquery.c.address_name,
        department_subquery.c.address_lat,
        department_subquery.c.address_long,
        department_subquery.c.district,
        department_subquery.c.phone_numbers,
        department_subquery.c.schedule,
    )
    .select_from(Company)
    .outerjoin(
        categories_subquery, Company.org_id == categories_subquery.c.company_id
    )
    .outerjoin(department_subquery, true())
)
if name:
    query = query.filter(Company.name.ilike(f"%{name}%"))
if category:
    if isinstance(category, list):
        query = query.filter(
            Company.categories.any(CompanyCategory.name.in_(category))
        )
    else:
        query = query.filter(
            Company.categories.any(CompanyCategory.name == category)
        )
if address:
    if isinstance(address, list):
        address_conditions = [
            department_subquery.c.address_name.ilike(f"%{addr}%")
            for addr in address
        ]
        query = query.filter(or_(*address_conditions))
    else:
        query = query.filter(
            department_subquery.c.address_name.ilike(f"%{address}%")
        )
if district:
    if isinstance(district, list):
        query = query.filter(department_subquery.c.district.in_(district))
    else:
        query = query.filter(department_subquery.c.district == district)

return query.where(Company.org_id.in_(base_query))
```

Рисунок 2.3 – Фрагмент репозиторію для отримання гуртків враховуючи адресу, район, категорію чи ключові слова.

2.3 Методи виявлення дублікатів та нормалізації

Під час автоматизованого збору інформації з різноманітних джерел система зіштовхується з проблемою дублювання записів. Це пов’язано з тим, що один і той самий гурток може бути представлений на кількох ресурсах або мати незначні варіації в описі. Для уникнення подібних повторів реалізовано перевірку унікальності записів на основі ключових полів, таких як назва гуртка, адреса, контактна інформація та координати розташування. Зіставлення виконується до моменту збереження в базу даних, що дозволяє виявляти дублікати ще на етапі обробки.

Важливою складовою забезпечення цілісності даних є попередня нормалізація. Перед додаванням до сховища інформація проходить процедуру уніфікації: формат телефонів, назв гуртків, адрес та описів приводиться до єдиного стандарту, очищується від зайвих символів і пробілів. Це дозволяє уникнути логічних суперечностей при подальшій обробці, фільтрації або виведенні даних на інтерфейс

користувача.

```
# -------------------------
# Methods for enriching companies departments data
# -------------------------

@staticmethod
def normalize_address(address: str) -> str:
    if not address:
        return None
    # вулиця Чукаріна, 3(Приміщення школи-ліцею "Оріяна" (ЗОШ №25)) => вулиця Чукаріна, 3
    return re.sub(r"\(.*\)", "", address).strip()
```

Рисунок 2.4 – Фрагмент коду нормалізації адреси під час парсингу

Окрім програмних механізмів, додатковим рівнем захисту від дублювання виступає грамотно спроєктована база даних. Завдяки накладеним обмеженням на унікальність комбінацій певних полів, система на рівні структури сховища запобігає запису ідентичних об'єктів. Це значно знижує ризик накопичення повторів та підвищує якість збереженої інформації.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Загальні функціональні та нефункціональні вимоги до системи

Під час створення програмного забезпечення для централізованого збору, обробки та візуального подання інформації про дитячі гуртки необхідно чітко окреслити як функціональні, так і нефункціональні вимоги, які визначають межі, очікувану поведінку та якість майбутньої системи.

До **функціональних вимог** належать можливості, які система зобов'язана реалізовувати у рамках свого призначення. У першу чергу, це автоматичний збір структурованої інформації з різних відкритих джерел за допомогою парсерів, які повинні регулярно виконуватися за встановленим розкладом. Система має також підтримувати обробку отриманих даних: нормалізацію, усунення повторів, доповнення інформації про координати та райони за допомогою інтеграції з картографічним сервісом (Google Maps API). Зібрані дані мають зберігатися у централізованій базі, а користувачам має бути доступний інтерфейс для перегляду, пошуку, фільтрації та взаємодії з інтерактивною мапою.

Серверна частина повинна забезпечувати API-доступ до всіх основних сутностей: гуртків, категорій, локацій, галерей тощо. Підтримка пагінації, фільтрації за параметрами (вік, напрям, район, тощо), а також наявність логування запитів і помилок — обов'язкові елементи функціональності. Фронтенд має надавати зручну навігацію каталогом, відображення деталей окремого гуртка, фільтраційну панель і карту з інтерактивними позначками.

До **нефункціональних вимог** належать характеристики, які визначають якість роботи системи, її масштабованість, продуктивність і зручність у використанні. Зокрема, система повинна бути побудована з урахуванням принципів модульності та розділення відповідальностей — кожен компонент (збір даних, логіка обробки, API, візуалізація) має функціонувати незалежно. Завдяки використанню контейнеризації (Docker), застосунок повинен легко розгортатися на

будь-якому серверному середовищі без складного налаштування. Збереження даних має здійснюватися у надійній реляційній СУБД PostgreSQL з урахуванням обмежень цілісності. Кешування відповідей Google Maps API повинне забезпечуватись через Redis, що зменшує навантаження та пришвидшує роботу системи.

Особливу увагу слід приділити стабільності: регулярний запуск парсерів не має порушувати роботу основного сервісу, а всі помилки повинні логуватися та бути легко відстежуваними. Інтерфейс має бути адаптивним і зручним як для перегляду з комп'ютера, так і з мобільних пристроїв. У майбутньому система має підтримувати легке розширення — як за рахунок додавання нових джерел, так і шляхом охоплення нових міст чи напрямів діяльності.

Таким чином, вимоги до системи визначають її як надійний, масштабований і зручний у користуванні інструмент для представлення повної та актуальної інформації про дитячі гуртки в одному місті.

3.2 Архітектура проєкту та організація коду

Структура застосунку була сформована з урахуванням принципів чистої архітектури та практичного поділу відповідальностей між логічними компонентами. Замість того, щоб об'єднувати всю логіку в кількох великих файлах, проєкт розбито на окремі, чітко визначені частини, кожна з яких відповідає за конкретний рівень — від низькорівневої інфраструктури до прикладного API.

Центральним ядром структури є папка *app/*, яка об'єднує всі основні модулі застосунку. У її межах виділено окремі директрії для API-інтерфейсів *(api/),* бізнес-логіки *(domain/),* інфраструктури *(infrastructure/),* фонових задач *(tasks/)* та спільних утиліт *(common/).* Такий поділ дозволяє зберігати гнучкість, швидко масштабувати окремі частини системи, а також підтримувати чистоту коду впродовж усієї розробки. [13, 15, 17]

Рівень API (api/) відповідає за взаємодію з зовнішнім світом. Саме тут знаходяться всі маршрути, залежності, схеми запитів і відповіді (через Pydantic), а

також окрема конфігурація запуску. Все, що «бачить» користувач чи фронтенд, проходить через цей рівень. Для ін’єкції залежностей застосовуються окремі модулі (dependencies.py, deps.py), що допомагає централізовано керувати ресурсами, які потрібні в різних частинах програми (наприклад, підключення до бази або кеша).

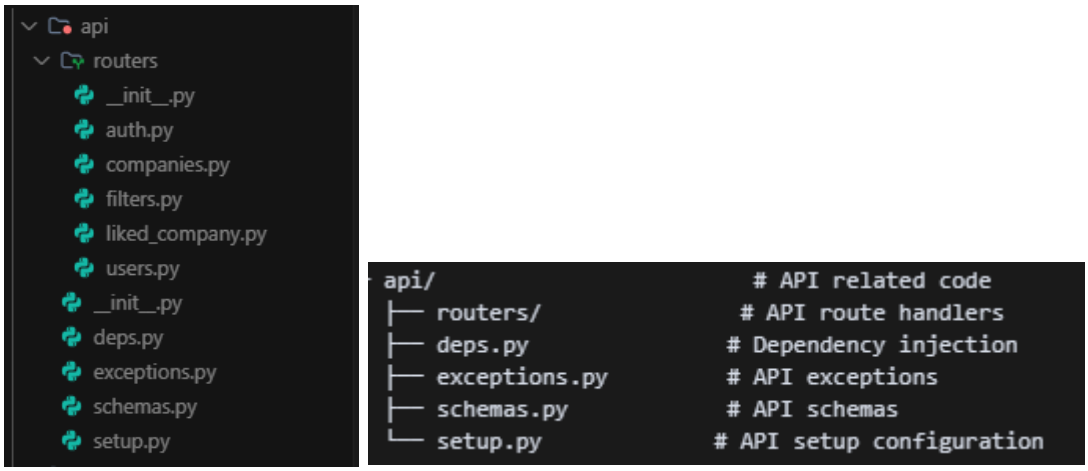


Рисунок 3.1 – Відображення структури модуля api/

Бізнес-логіка системи зосереджена в *domain/*. У межах цієї директорії зібрані окремі підрозділи, кожен із яких присвячений конкретному напрямку функціональності: автентифікація *(auth),* категорії гуртків *(category),* користувачі *(user),* логіка фільтрації *(filters),* взаємодія з компаніями чи закладами *(company)* тощо. Тут же зазвичай розміщується реалізація репозиторіїв, сервісів, схем даних та супутніх констант. Така структура дозволяє чітко розмежувати логіку і уникати надмірної залежності між модулями.

Рисунок 3.2 – Відображення структури бізнес логіки в domain/

Інфраструктурна частина *(infrastructure/)* містить усі компоненти, пов'язані з технічною реалізацією: доступ до бази даних *(database),* логіку роботи з Redis *(redis),* інтеграцію з Google Maps API (geolocation) тощо. Таким чином, доступ до зовнішніх сервісів ізольовано від основної логіки застосунку — що дозволяє замінювати або тестувати інфраструктурні залежності незалежно від решти системи.

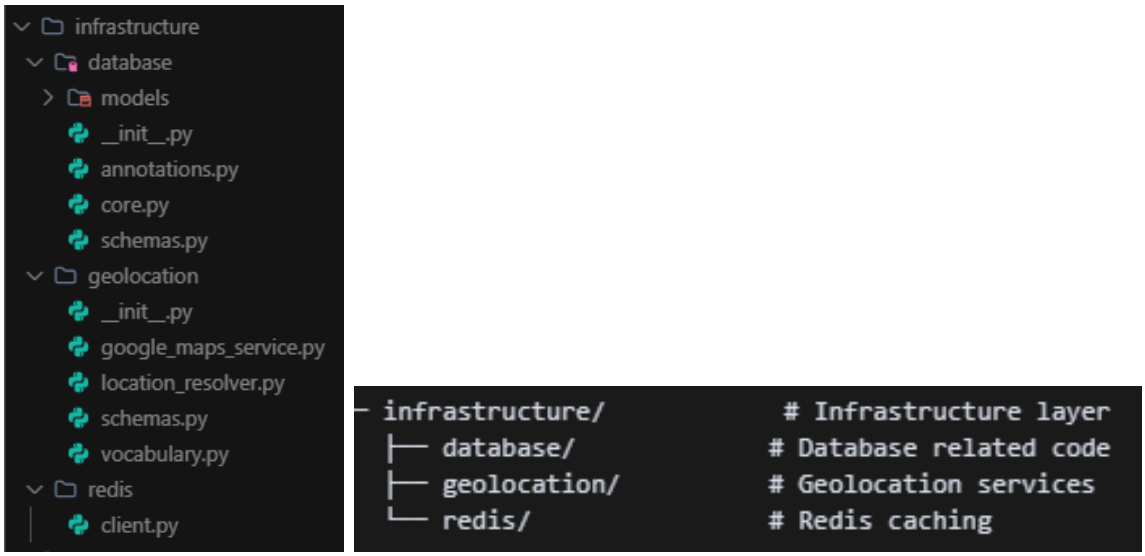


Рисунок 3.3 – Відображення структури infrastructure/

Окрему увагу приділено парсерам *(parsers/)* — ці компоненти відповідають за вилучення та попередню обробку інформації з відкритих джерел. Вони реалізовані у вигляді окремих сценаріїв, які працюють асинхронно, використовуючи httpx для запитів і BeautifulSoup для аналізу HTML-структури. Поряд із цим працює модуль *tasks/,* у якому містяться періодичні завдання, що запускаються згідно з розкладом через aiocron. Це дозволяє щодня без втручання людини оновлювати базу актуальними даними. [2, 9]

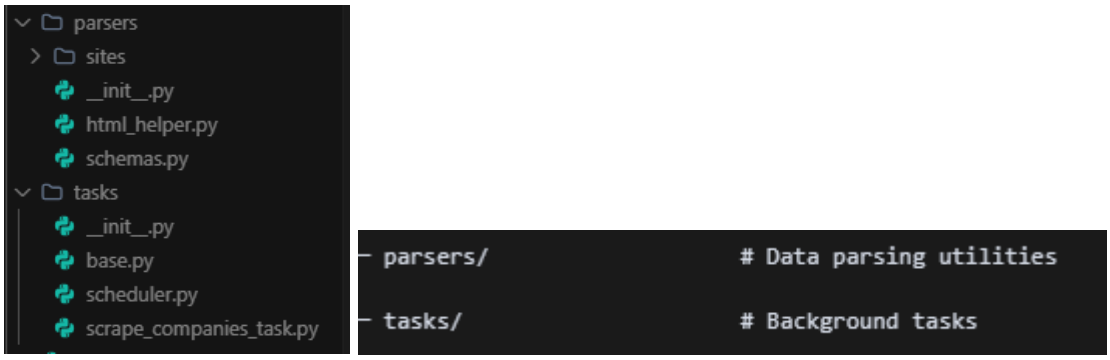


Рисунок 3.4 – Структура модулей /parser та /tasks

Головною точкою входу до застосунку виступає *main.py*, а початкове завантаження ресурсів централізовано виконуються через loader.py. Інші компоненти, як-от налаштування Docker, конфігурації Redis чи файл залежностей requirements.txt, винесено в кореневу частину проєкту. Таке розділення забезпечує легкість у розгортанні та мінімізацію помилок під час налаштування середовища.

```python
@asynccontextmanager
async def lifespan(app):
    # on_startup
    await redis_client.connect()

    logger.info("Application starting up...")
    await ScrapeCompaniesTask().execute()
    scheduler.register_task(
        "scrape_companies",
        ScrapeCompaniesTask().execute,
        "0 0 * * *",  # Run at midnight every day
    )
    scheduler.start()
    yield
    # on_shutdown
    logger.info("Application shutting down...")
    scheduler.stop()


app = create_app(lifespan=lifespan)

if __name__ == "__main__":
    uvicorn.run(
        "app.main:app",
        host=settings.APPLICATION_HOST,
        port=settings.APPLICATION_PORT,
        reload=not settings.PRODUCTION,
    )
```

Рисунок 3.5 – Реалізація точки входу main.py

```python
from app.infrastructure.geolocation.google_maps_service import GoogleMapsService
from app.infrastructure.geolocation.location_resolver import LocationResolver
from app.infrastructure.redis.client import RedisClient
from app.core import settings, logger

google_maps_service = GoogleMapsService(settings.GOOGLE_API_KEY, logger=logger)
redis_client = RedisClient(settings.redis_uri, logger=logger)
location_resolver = LocationResolver(google_maps_service, redis_client)

__all__ = ["google_maps_service", "redis_client", "location_resolver"]
```

Рисунок 3.6 – Початкове завантаження ресурсів у loader.py

Загалом, архітектура побудована так, щоб новий розробник міг швидко зорієнтуватись у проєкті, а масштабування, тестування та супровід не викликали зайвих труднощів. Це не просто набір скриптів — це повноцінна система з чіткою логікою, відокремленими рівнями та здоровою внутрішньою дисципліною коду.

3.2.1 Принципи взаємодії між модулями та підхід до організації залежностей

У процесі розробки системи одним із пріоритетів стала побудова прозорої та контрольованої взаємодії між логічними модулями. Проєкт реалізовано таким чином, щоб кожен шар не «знав» зайвого про інші частини системи, а залежності передавались явно, у передбачений спосіб. Це дозволяє уникати прихованих зв'язків, зменшує ймовірність неочікуваних помилок і спрощує підтримку.

Основною технікою керування залежностями в системі є **ін'єкція через функції**, яку підтримує FastAPI завдяки механізму Depends. Усі ключові залежності — зокрема, сесія бази даних, сервіси, кеш-клієнти Redis або зовнішні адаптери — передаються в ендпоінти явно через спеціально створені функції в deps.py. Це дозволяє централізовано контролювати життєвий цикл ресурсів (наприклад, відкривати й закривати сесію БД) і зменшує кількість дублювання коду. [1]

```python
async def get_session() -> AsyncSession:
    async with db_helper.get_db() as session:
        yield session


def CommaSeparatedList(param_name: str):
    def dependency(request: Request) -> list[str] | None:
        value = request.query_params.get(param_name)
        if not value:
            return None
        return value.split(",") if "," in value else [value]

    return dependency


SessionDep = Annotated[AsyncSession, Depends(get_session)]
PaginationDep = Annotated[SPagination, Depends(SPagination)]
```

Рисунок 3.7 – Фрагмент із api/deps.py

```python
@router.get("/")
async def get_companies(
    session: SessionDep,
    pagination: PaginationDep,
    name: str | None = None,
    category: Optional[list[str]] = Depends(CommaSeparatedList("category")),
    address: Optional[list[str]] = Depends(CommaSeparatedList("address")),
    district: Optional[list[str]] = Depends(CommaSeparatedList("district")),
) -> SCompanyShortListResponse:
```

Рисунок 3.8 – Приклад використання залежностей (dependencies)

Виклик основної логіки завжди виконується через сервіси, що розміщені в domain/. Жоден маршрут напряму не працює з базою даних або Redis — для цього створені відповідні сервіси й репозиторії, які абстрагують внутрішню реалізацію доступу до даних. Такий підхід дозволяє швидко замінити, наприклад, джерело координат або кеш-систему, не торкаючись логіки, що працює на рівні API.

```python
@router.get("/")
async def get_companies(
    session: SessionDep,
    pagination: PaginationDep,
    name: str | None = None,
    category: Optional[list[str]] = Depends(CommaSeparatedList("category")),
    address: Optional[list[str]] = Depends(CommaSeparatedList("address")),
    district: Optional[list[str]] = Depends(CommaSeparatedList("district")),
) -> SCompanyShortListResponse:

    return await CompanyService(session).get_short_companies_by_page(
        name=name,
        category=category,
        address=address,
        district=district,
        page=pagination.page,
        limit=pagination.limit,
    )
```

Рисунок 3.9 – Використання сервісу у контроллері

Окремим принципом побудови стало уникнення кругових залежностей. Усі компоненти викликають інші строго по напрямку: API → service → repository. Зворотних викликів немає. Це дає змогу уникнути плутанини в логіці викликів та полегшує покриття коду тестами.

Також було впроваджено низку структурних правил, наприклад:

– усі схеми для валідації запитів та відповідей мають зберігатися на відповідному рівні domain/СУТНІСТЬ/schemas (response.py – схеми для відповіді на запит, request.py – схема для отримання даних в контроллер)

– сервіси не містять HTTP-логіки, а працюють виключно з об'єктами Python,

– репозиторії — єдине місце, де дозволено роботу з ORM та SQL.

Такий підхід дозволив не лише розділити відповідальність, а й створити передумови для **масштабування** — у майбутньому можна буде додати нові модулі, джерела або сервіси, не порушуючи цілісність системи. Крім того, явне керування залежностями значно **полегшує модульне тестування**: кожен компонент може бути ізольовано перевірений із використанням мок-об’єктів без запуску всієї системи.

3.2.2 Взаємодія між шарами: обробка запитів, валідація, маршрутизація

Проєкт побудований за принципом багаторівневої взаємодії, де кожен шар виконує чітко визначену роль у життєвому циклі HTTP-запиту. Цей поділ дозволяє зберігати чистоту коду, зменшувати дублювання логіки та спрощувати тестування окремих компонентів. Передбачено строгий маршрут руху запиту: від зовнішнього HTTP-інтерфейсу до бізнес-логіки і назад — через серії послідовних перетворень, перевірок і викликів.

Коли клієнт (React-фронтенд) надсилає запит до API, він у першу чергу потрапляє до відповідного **роутера**, розташованого в директорії api/routers/. Там кожен маршрут визначено у вигляді окремої функції, до якої додаються залежності через механізм Depends — наприклад, сесія бази даних, авторизований користувач або кеш-клієнт. Роутери не містять жодної складної логіки — їх єдина функція — **переадресація запиту** до сервісів у шарі domain/, передаючи туди вже валідовані дані. [15]

Перед тим як запит буде передано в бізнес-логіку, дані, отримані від користувача, проходять через **валідацію за допомогою Pydantic-схем**. Ці схеми чітко описують типи, обов’язковість і формат кожного поля (наприклад, назва гуртка, координати, категорія тощо). У разі невідповідності — FastAPI автоматично повертає інформативну помилку клієнту, що підвищує стійкість і передбачуваність системи.

```python
@router.get("/{company_id}")
async def get_company(session: SessionDep, company_id: str) -> SCompany:
    response = await CompanyService(session).get_company_by_id(company_id)
    if not response:
        raise HTTPException(status_code=404, detail="Company not found")

    return response
```

Рисунок 3.10 – Роутер для отримання гуртка по його ID

Сервіси, які отримують запит, виконують основну логіку — перевіряють умови, формують SQL-запити використовуючи SqlAlchemy ORM до бази через репозиторії, здійснюють фільтрацію, сортування, агрегацію тощо. Після обробки результат повертається назад до рівня API, де перетворюється у відповідну відповідь, яка знову ж таки проходить перевірку за Pydantic-схемами — цього разу для вихідних даних, що гарантує послідовність структури відповіді для будь-якого споживача API.

Такий чітко організований ланцюг взаємодії — **маршрутизація → валідація → виклик сервісу → репозиторій → відповідь** — дозволяє гнучко керувати кожним кроком, легко додавати нові функції, змінювати поведінку логіки без втручання в інші шари та гарантувати стабільність усієї системи в цілому. Крім того, він полегшує написання документації: оскільки структури запитів і відповідей вже визначені схемами, FastAPI автоматично генерує OpenAPI-специфікацію, яка слугує базою для інтеграції з іншими сервісами або фронтендом. [13]

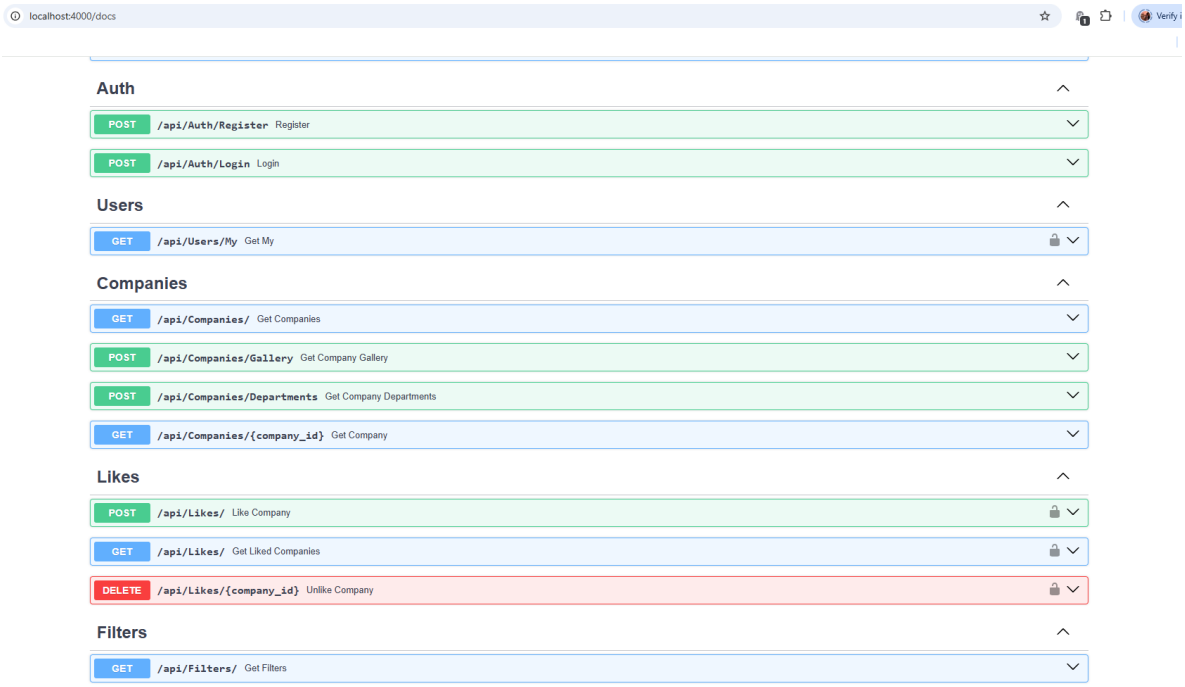


Рисунок 3.11 – Swagger (OpenAPI) автогенерована документація

3.3 Реалізація механізмів збору та обробки даних

Одним із найважливіших блоків у структурі системи є автоматизований механізм отримання та первинної обробки даних з відкритих джерел. Збір інформації про гуртки не виконується вручну чи в напівавтоматичному режимі, а реалізований як незалежний, самодостатній процес, який виконується у фоновому режимі відповідно до заданого графіка.

Парсинг реалізовано за допомогою мови Python з використанням комбінації бібліотек httpx для асинхронного надсилання HTTP-запитів та BeautifulSoup для розбору HTML-структури сторінок. Ці компоненти дозволяють ефективно працювати з різними сайтами, навіть якщо вони не мають відкритого API, що розширює кількість доступних джерел. Після отримання HTML-вмісту сторінки виконується вилучення необхідних даних — таких як назва гуртка, опис, графік занять, адреса, контактні телефони тощо. Кожен парсер реалізований як окремий

модуль, що дозволяє зручно масштабувати систему при додаванні нових джерел або зміні структури сторінки.

```python
class ListInUaParser(HTMLHelper):
    BASE_URL = "https://list.in.ua"
    SEARCH_URL = "https://list.in.ua/%D0%9B%D1%8C%D0%B2%D1%96%D0%B2/%D0%93%D1%83%D1%80%D1%82%D0%BA%D0%B8"

    def __init__(self):
        self.session = httpx.AsyncClient()
        self.request_count = 0

    # --------------------------
    # Methods for fetching data
    # --------------------------
    async def _request_with_retry(
        self, method: str, url: str, max_attempts: int = 3, delay: float = 2, **kwargs
    ) -> Response | None:
        for attempt in range(max_attempts):
            try:
                response = await self.session.request(
                    method,
                    url,
                    headers={
                        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 S
                    },
                    **kwargs,
                )
                response.raise_for_status()
                self.request_count += 1
                return response
            except Exception as exc:
                logger.debug(f"Помилка запиту {method} на {url}: {exc}")
                if attempt < max_attempts - 1:
                    await asyncio.sleep(delay)
                    delay *= 2
                else:
                    logger.debug(f"Всі спроби запиту {method} на {url} вичерпані.")
                    return None

    async def fetch_page(self, page: int) -> str | None:
        url = self.SEARCH_URL + f"/page/{page}" if page > 1 else self.SEARCH_URL

        response = await self._request_with_retry("GET", url, timeout=20)
        if not response:
            logger.debug(f"Не вдалося отримати сторінку номер {page}")
            return None
        return response.text
```

Рисунок 3.12 – Фрагмент модуля для парсингу сайту list in ua

```python
# --------------------------
# Methods for parsing a card
# --------------------------
async def _parse_card(self, card: Tag) -> SCompany:
    """
    Extract organisation info from one card
    """
    image_url = self.extract_image_url(card)
    org_id = self.extract_org_id(card)
    title = self.extract_title(card)
    link = self.extract_link(card)
    category = self.extract_category(card)

    return SCompany(
        org_id=org_id,
        name=title,
        link=link,
        categories=category,
        preview_image=image_url,
    )
```

```python
@staticmethod
def extract_category(card: Tag) -> List[str]:
    elements = card.select("a.js-service-filter")
    return [element.get_text(strip=True) for element in elements]

@staticmethod
def extract_title(card: Tag) -> Optional[str]:
    element = card.select_one("h2.item-search__title.mobile-hide a")
    return element.get_text(strip=True) if element else None

@staticmethod
def extract_link(card: Tag) -> Optional[str]:
    element = card.select_one("h2.item-search__title.mobile-hide a")
    link = element.get("href") if element else None
    return ListInUaParser.BASE_URL + link if link else None

@staticmethod
def extract_image_url(card: Tag) -> Optional[str]:
    element = card.select_one(".item-search__img img")
    return element.get("src") if element else None

@staticmethod
def extract_org_id(card: Tag) -> Optional[str]:
    return card.get("data-item")
```

Рисунок 3.13 – Фрагмент вилучення даних із об’єкту BeautifulSoup

Щоб процес оновлення інформації був безперервним і не вимагав ручного запуску, застосовано бібліотеку **aiocron**, яка працює як планувальник. Вона дозволяє задати точний розклад виконання кожного парсера — наприклад, щоденне оновлення о 03:00 ночі. Кожне завдання працює незалежно, а у разі виникнення помилки — логуються деталі події, щоб розробник міг оперативно реагувати.

```python
class ScrapeCompaniesTask(BaseTask):
    CACHE_KEY = "list_in_ua"
    CACHE_PREFIX = "parser"
    CACHE_EXPIRY = 24 * 60 * 60  # 24 hours in seconds

    async def run(self):
        """Execute the company scraping task."""
        logger.debug("[TASK] Run companies parser has been started!")

        if await self._is_already_parsed():
            logger.debug(
                "Skipping list in ua parser. The parser has already been launched during this day."
            )
            return

        await self._set_parser_cache()
        await self._process_companies()

    async def _is_already_parsed(self) -> bool:
        """Check if parser has already run today."""
        return await redis_client.exists(key=self.CACHE_KEY, prefix=self.CACHE_PREFIX)

    async def _set_parser_cache(self) -> None:
        """Set cache entry to track parser execution."""
        await redis_client.save_to_cache(
            key=self.CACHE_KEY,
            prefix=self.CACHE_PREFIX,
            value="-",
            expire_time=self.CACHE_EXPIRY,
        )

    async def _process_companies(self) -> None:
        """Process and store parsed companies."""
        companies = await self._parse_list_in_ua()
        async with db_helper.get_db() as session:
            company_service = CompanyService(session)
            await company_service.add_companies_with_departments(companies)

    @staticmethod
    @log_execution_time
    async def _parse_list_in_ua():
        """Parse companies from List.in.ua."""
        parser = ListInUaParser()
        return await parser.parse_all()
```

Рисунок 3.14 – Реалізація класу завдання, яке збирає інформацію раз в добу

```
scheduler.register_task(
    "scrape_companies",
    ScrapeCompaniesTask().execute,
    "0 0 * * *",  # Run at midnight every day
)
scheduler.start()
```

Рисунок 3.15 – Приклад запуску завдання

Після збору інформації починається фаза обробки та нормалізації. Система перетворює необроблені сирі дані у стандартизований вигляд відповідно до внутрішніх Pydantic-схем. Це включає очищення тексту, переведення телефонів у єдиний формат, видалення зайвих пробілів або символів, та уніфікацію структури записів.

```
@staticmethod
def extract_phone_numbers(raw: str) -> list[str]:
    pattern = re.compile(r"\+[\d\s\-\(\)]{7,}(?=\+|\(|$)")

    matches = pattern.findall(raw)
    return [m.strip() for m in matches]
```

```
def _extract_company_phones(self, department: Tag) -> list[str]:
    element = department.select_one(".company-phone")

    return (
        self.extract_phone_numbers(element.get_text(strip=True)) if element else []
    )
```

Рисунок 3.16 – Демонстрація методів для переведення телефонів у єдиний формат

Крім цього, важливу роль відіграє етап геолокаційної обробки. Використовуючи Google Maps API, система перетворює звичайні адреси на координати (широта і довгота), а також визначає до якого району міста Львова належить локація. Ці дані необхідні для подальшої побудови інтерактивної карти та фільтрації гуртків за районами. Щоб уникнути надмірного навантаження на API, упроваджено кешування результатів геолокаційних запитів за допомогою Redis.

Якщо адреса вже оброблялася раніше, результат буде взято з кешу без повторного звернення до Google.

```python
async def resolve_locations(self, addresses: list[str]) -> list[LocationInfo]:
    cached_results = await asyncio.gather(
        *[
            self.redis_client.get_cached(address, prefix="location")
            for address in addresses
        ]
    )

    cached_locations = {}
    uncached_addresses = []

    for address, cached in zip(addresses, cached_results):
        if cached is not None:
            cached_locations[address] = LocationInfo.model_validate_json(cached)
        else:
            uncached_addresses.append(address)

    async with self.google_maps_service as service:
        locations_from_google = await service.get_locations_info(uncached_addresses)

    await asyncio.gather(
        *[
            self.redis_client.set_non_expire(
                location.address,
                location.model_dump_json(),
                prefix="location",
            )
            for location in locations_from_google
        ]
    )

    combined_locations: dict[str, LocationInfo] = {
        loc.address: loc for loc in locations_from_google
    }

    for address, location in cached_locations.items():
        if address not in combined_locations:
            combined_locations[address] = location

    return list(combined_locations.values())
```

Рисунок 3.17 – Реалізація власного сервісу для отримання району та координат із адреси.

Після повної обробки й збагачення дані надходять у PostgreSQL-базу даних, де вони зберігаються у вигляді структурованих записів. На цьому етапі виконується остаточна перевірка на дублікати: якщо запис уже існує в системі (визначається за

комбінацією ключових полів), він не буде доданий повторно. Це дозволяє підтримувати базу даних у чистому, оновленому та логічно узгодженому стані.

```python
async def add_companies_with_departments(self, companies: list[SCompany]):
    company_values = []
    department_values = []
    category_links = []

    for company in companies:
        company = company.model_dump()
        org_id = company["org_id"]

        company_values.append(
            {
                "name": company["name"],
                "org_id": org_id,
                "link": company["link"],
                "preview_image": company["preview_image"],
                "description": company["description"],
                "carousel": company["carousel"],
            }
        )
        for category_name in company.get("categories", []):
            category_links.append(
                {"company_org_id": org_id, "category_name": category_name}
            )

        for department in company.get("departments", []):
            department_values.append(
                {
                    "address_name": department["address_name"],
                    "company_id": org_id,
                    "address_lat": department["address_lat"],
                    "address_long": department["address_long"],
                    "district": department["district"],
                    "phone_numbers": department["phone_numbers"],
                    "schedule": department["schedule"],
                }
            )

    try:
        await self.repository.add_companies_bulk(company_values)
        await self.repository.add_departments_bulk(department_values)
        category_names = [link["category_name"] for link in category_links]
        name_to_id = await self.category_repository.add_categories(category_names)
        await self.repository.add_company_category_links(category_links, name_to_id)
        await self.session.commit()
    except Exception as e:
        logger.error(f"Error adding companies: {e}")
```

Рисунок 3.18 – Реалізація методу у сервісі для добавлення нових локацій (гуртків) у БД

Завдяки цьому підходу до збору, нормалізації, доповнення та перевірки даних система може автоматично підтримувати свою актуальність, мінімізуючи втручання з боку розробника або адміністратора.

3.4 Робота з базою даних PostgreSQL

У рамках реалізації системи було обрано реляційну систему керування базами даних PostgreSQL, яка поєднує в собі високу продуктивність, гнучкість структур, підтримку складних запитів і надійність. У поєднанні з ORM-бібліотекою SQLAlchemy, PostgreSQL дозволяє будувати стабільну та логічно узгоджену модель даних, що відповідає вимогам багаторівневої архітектури проєкту.

Розробка структури бази даних велася з урахуванням розділення інформації за тематичними сутностями — окремо описані таблиці для гуртків, категорій, локацій (departments), а також допоміжні таблиці, які забезпечують гнучке групування, фільтрацію та прив’язку до геоданих. Усі моделі реалізовані за допомогою SQLAlchemy і використовують асинхронні сесії для роботи в межах подій FastAPI-застосунку. [5, 6, 14]

3.4.1 Структура таблиць: компанії, локації, категорії та фундамент для подальшого розширення

Модель даних проєкту побудована навколо центральної сутності — **організації або компанії**, яка на прикладному рівні представляє гурток чи заклад, що надає позашкільні послуги. Основу структури складає таблиця companies, де кожна компанія ідентифікується через унікальний *org_id*. Окрім базових текстових полів, таких як назва, опис, посилання на джерело та прев’ю-зображення, кожен запис має масив зображень (carousel), що зберігається у форматі JSONB для підтримки гнучких галерей.

```python
class Company(Base):
    __tablename__ = "companies"
    org_id: Mapped[str] = mapped_column(unique=True)

    name: Mapped[str | None] = mapped_column(default=None)
    link: Mapped[str | None] = mapped_column(nullable=True)
    preview_image: Mapped[str | None] = mapped_column()
    description: Mapped[str | None] = mapped_column(default="", nullable=True)
    carousel: Mapped[list] = mapped_column(JSONB, default=[])

    departments: Mapped[list["Department"]] = relationship(
        back_populates="company", cascade="all, delete-orphan"
    )

    categories: Mapped[list["CompanyCategory"]] = relationship(
        secondary=company_association_table,
        back_populates="companies",
    )

    liked_by_users: Mapped[list["LikedCompany"]] = relationship(
        back_populates="company", cascade="all, delete-orphan"
    )
```

Рисунок 3.19 – Модель Company (гурток) SQLAlchemy для БД

Кожна компанія може мати одну або кілька локацій, які описані в таблиці *departments*. Ця таблиця зберігає конкретні адреси, координати *(lat, long),* назву району міста Львова (district), а також розклад і контактні номери у вигляді масивів JSON. Локації пов'язані з компанією через зовнішній ключ *company*_id, а для запобігання дублюванню на рівні однієї організації встановлено обмеження унікальності за комбінацією *company_id + address_name*.

```python
class Department(Base):
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    company_id: Mapped[str] = mapped_column(
        ForeignKey("companies.org_id", ondelete="CASCADE")
    )

    address_name: Mapped[str | None] = mapped_column(nullable=True)
    address_lat: Mapped[str | None] = mapped_column(nullable=True)
    address_long: Mapped[str | None] = mapped_column(nullable=True)
    district: Mapped[str | None] = mapped_column(nullable=True, default=None)
    phone_numbers: Mapped[list[str]] = mapped_column(JSONB, default=list)
    schedule: Mapped[list[list[str]]] = mapped_column(JSONB, default=list)

    company: Mapped["Company"] = relationship(back_populates="departments")

    __table_args__ = (
        UniqueConstraint(
            "company_id", "address_name", name="uq_department_company_address"
        ),
    )
```

Рисунок 3.20 – Модель Department SQLAlchemy для БД

Для класифікації гуртків передбачено таблицю *company_categories*, яка містить перелік унікальних напрямків (наприклад, мистецтво, спорт, технічна творчість тощо). Реалізовано зв'язок багато-до-багатьох між компаніями та категоріями через проміжну таблицю *company_association_categories*. Це дозволяє гнучко комбінувати гуртки з кількома категоріями одночасно, що надалі використовується у системі фільтрації.

```
company_association_table = Table(
    "company_association_categories",
    Base.metadata,
    Column("company_id", ForeignKey("companies.org_id"), primary_key=True),
    Column("category_id", ForeignKey("company_categories.id"), primary_key=True),
)
```

Рисунок 3.21 – Проміжна таблиця для зв'язку many-to-many

Також у структурі бази передбачено механізм уподобань, реалізований через таблицю *liked_companies*. Вона дозволяє зберігати взаємозв'язки між користувачами та компаніями, які їм сподобались. Кожен запис містить *user_id* та *company_id*, і для уникнення повторних додавань однієї й тієї ж вподобаної компанії встановлено унікальне обмеження на цю пару. Цей функціонал є **фундаментом для подальшого розширення можливостей системи**, зокрема персоналізації, рекомендацій або побудови історії взаємодії користувачів із платформою.

```
class LikedCompany(Base):
    __tablename__ = "liked_companies"

    user_id: Mapped[str] = mapped_column(ForeignKey("users.id", ondelete="CASCADE"))
    company_id: Mapped[str] = mapped_column(
        ForeignKey("companies.org_id", ondelete="CASCADE")
    )

    # Relationships
    user = relationship("User", back_populates="liked_companies")
    company = relationship("Company", back_populates="liked_by_users")

    __table_args__ = (
        UniqueConstraint("user_id", "company_id", name="uq_user_company"),
    )
```

Рисунок 3.22 – Модель LikedCompany SQLAlchemy

Самі користувачі описані в таблиці *users*, де зберігаються їхні email-адреси, захешовані паролі, аватари, імена відображення та ролі. У поточній реалізації ця таблиця використовується переважно як заготовка для майбутніх функцій (реєстрація, вподобання, автентифікація), проте її структура вже дозволяє зручно масштабувати систему в напрямку користувацької взаємодії.

```python
class User(Base):
    __tablename__ = "users"

    email: Mapped[str] = mapped_column(unique=True)
    hashed_password: Mapped[str]

    display_name: Mapped[str | None] = mapped_column(nullable=True)
    picture: Mapped[str | None] = mapped_column(nullable=True)

    role: Mapped[UserRole] = mapped_column(
        SqlEnum(UserRole), nullable=False, default=UserRole.REGULAR
    )

    is_verified: Mapped[bool] = mapped_column(default=False)

    # Relationships
    liked_companies: Mapped[list["LikedCompany"]] = relationship(
        back_populates="user", cascade="all, delete-orphan"
    )
```

Рисунок 3.23 – Модель User SQLAlchemy

Загальна модель БД реалізована з урахуванням каскадного видалення *(ondelete="CASCADE")*, що забезпечує автоматичне очищення пов’язаних записів при видаленні основного об’єкта (наприклад, при видаленні компанії автоматично видаляються її локації та вподобання).

Таким чином, структура бази даних побудована з урахуванням **гнучкого розширення, логічної узгодженості між сутностями та продуктивності при складних вибірках**, а також закладає фундамент для майбутнього розвитку функціоналу користувачів.

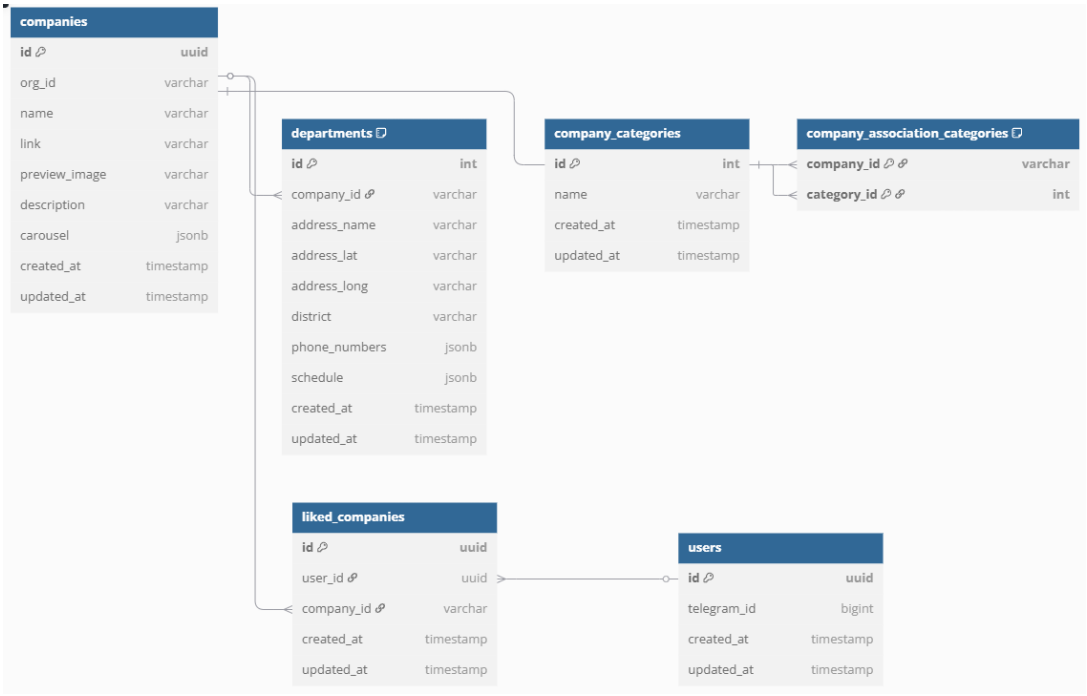


Рисунок 3.24 – ER-діаграма (Entity Relationship Diagrams)

3.4.2 Валідація даних та запобігання дублюванню

Однією з важливих задач при обробці й збереженні даних стало **усунення повторів і контроль цілісності інформації**. Навіть попри попередню обробку даних на рівні парсера, повторне надходження тих самих записів можливе — зокрема, при зміні форматування або незначних відмінностях у верстці джерел.

Для вирішення цієї проблеми реалізовано **дворівневу систему перевірки унікальності**. По-перше, кожна модель у Python містить логіку валідації: перед створенням нового запису система перевіряє, чи не існує вже схожого запису з такою ж назвою, адресою та координатами. По-друге, на рівні самої бази даних встановлені **унікальні обмеження** (*UNIQUE CONSTRAINT*) на ключові поля (або їх комбінації), які не дозволяють повторно додати ідентичний запис.

Такий підхід дозволяє зберігати цілісність даних навіть у разі збою логіки парсингу, одночасних запитів або імпорту з кількох джерел. Завдяки валідації на

обох рівнях — прикладному та структурному — система гарантує, що в базі не накопичуватимуться дублікати. [2]

3.4.3 Міграція за допомогою Alembic

Для керування структурними змінами бази даних було інтегровано **Alembic** — офіційний інструмент для міграцій у рамках SQLAlchemy. Він дозволяє фіксувати всі зміни в моделях у вигляді міграційних скриптів, які зручно застосовуються в середовищах розробки, тестування або продакшену.

Процес створення нових таблиць або зміни вже існуючих виконується не вручну, а через автоматичну генерацію міграцій (*alembic revision --autogenerate*). Це дозволяє підтримувати **синхронність між Python-моделями та структурою БД**, уникати помилок, пов’язаних із невідповідністю структури.

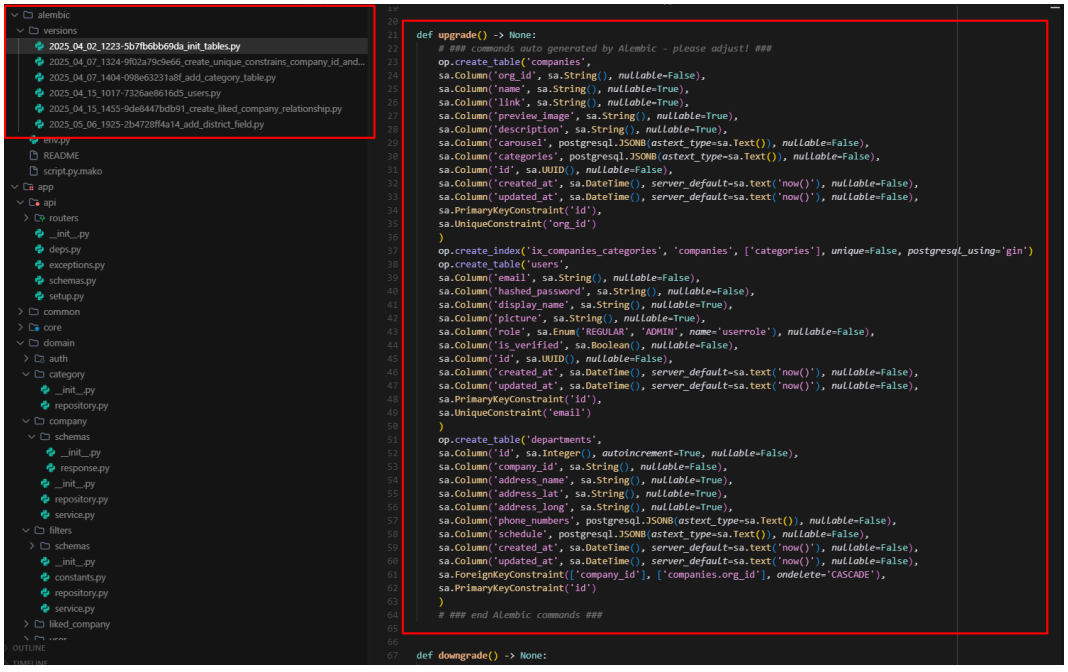


Рисунок 3.25 – Міграції alembic та приклад самої міграції

У проєкті реалізовано також **порядок застосування міграцій при розгортанні через Docker** — усі зміни структури застосовуються автоматично під час старту контейнера, що спрощує підтримку системи та запуск у новому

середовищі. Такий підхід дозволяє не лише вести контроль версій бази даних, а й легко повертатись до попереднього стану у разі потреби.

```
#!/bin/sh
alembic upgrade head
exec uvicorn app.main:app --host "${APPLICATION_HOST}" --port "${APPLICATION_PORT}"
```

Рисунок 3.26 – entrypoint.sh файл, який запускається при розгортанні докеру

3.5 Створення API за допомогою FastAPI

Однією з ключових переваг обраної архітектури є можливість швидкої розробки масштабованого і безпечного REST API завдяки фреймворку FastAPI. Завдяки використанню типізованих моделей, залежностей та автоматичної генерації документації, розробка та підтримка API значно спрощується, зберігаючи при цьому високу якість і передбачуваність інтерфейсу. [1]

3.5.1 Централізована конфігурація через клас Settings

Для підтримки чистоти коду та зручності масштабування системи було реалізовано окремий конфігураційний модуль, що відповідає за централізоване зчитування налаштувань. У якості основи застосовано бібліотеку Pydantic Settings, яка дозволяє типізовано і безпечно працювати з конфігураційними змінними, зчитуючи їх як з *.env*-файлу, так і з системного середовища.

Весь набір параметрів зосереджено в класі *Settings*, який виступає єдиною точкою істини для критичних налаштувань, таких як:

- параметри хосту та порту додатку;
- режим запуску (продакшн / розробка);
- секретні ключі для генерації JWT-токенів;
- дозволені origin-адреси для CORS;
- шляхи до лог-файлів;
- URI для PostgreSQL та Redis;

- ключі для Google Maps API.

Особливої уваги заслуговує логіка адаптації до середовища: методи *enable_production, disable_production* та *update_production_mode* дозволяють **динамічно перемикати додаток між розробницьким і бойовим режимом**, змінюючи критичні параметри (як-от *DB_ECHO, SHOW_SWAGGER*) без потреби дублювати конфігурації.

```
class Config:
    env_file = BASE_DIR / ".env"

def enable_production(self):
    self.PRODUCTION = True
    self.DB_ECHO = False
    self.SHOW_SWAGGER = False

def disable_production(self):
    self.PRODUCTION = False
    self.DB_ECHO = True

def update_production_mode(self):
    if self.PRODUCTION:
        self.enable_production()
    else:
        self.disable_production()

def __post_init_post_parse__(self):
    self.update_production_mode()
```

Рисунок 3.27 – Методи адаптації до середовища

Для зручності було передбачено й обчислювані властивості, які динамічно формують URL-адреси (*application_url, postgres_uri, redis_uri*), що спрощує їх подальше використання в будь-якій точці проєкту без ризику помилок при ручному складанні рядків підключення. [2]

```python
@property
def postgres_uri(self) -> str:
    return (
        f"postgresql+asyncpg://"
        f"{self.POSTGRES_USER}:"
        f"{self.POSTGRES_PASSWORD}@"
        f"{self.POSTGRES_HOST}:"
        f"{self.POSTGRES_PORT}/"
        f"{self.POSTGRES_DB}"
    )

@property
def redis_uri(self) -> str:
    return (
        f"redis://"
        f"{self.REDIS_USER}:"
        f"{self.REDIS_PASSWORD}@"
        f"{self.REDIS_HOST}:"
        f"{self.REDIS_PORT}"
    )
@property
def application_url(self) -> str:
    return f"{self.protocol}://{self.APPLICATION_HOST}:{self.APPLICATION_PORT}"
```

Рисунок 3.28 – Динамічні властивості, які формують URL-підключення для різних ресурсів

Таким чином, клас *Settings* став **надійною базою для керування параметрами** середовища й гарантує передбачувану поведінку застосунку у будь-яких умовах — локально, на *dev* чи у продакшн-оточенні.

Повна реалізація класу Settings:

```python
class Settings(BaseSettings):
    # --- API settings
    APPLICATION_PORT: int = 4000
    APPLICATION_HOST: str = "localhost"
    ALLOWED_ORIGIN: str | list
    SECURE_PROTOCOL: bool = False
    LIMIT_PER_PAGE: int = 12

    SHOW_SWAGGER: bool = True
    API_V1_STR: str = "/api"
    # 60 minutes * 24 hours * 8 days = 8 days
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 60 * 24 * 8
    JWT_SECRET_KEY: str = "ynnphjqwbuparbzzvvdiacnjsaoyiorxnwqgbqmewotkgjq"
    JWT_ALGORITHM: str = "HS256"

    PRODUCTION: bool = False
    LOG_LEVEL: str = "DEBUG"
    LOG_FILE_PATH: str | Path = BASE_DIR / "backend.log"
```

```
    # --- Database settings
    DB_ECHO: bool = False
    POSTGRES_HOST: str = "localhost"
    POSTGRES_PORT: int = 5432
    POSTGRES_DB: str = "test"
    POSTGRES_USER: str = "postgres"
    POSTGRES_PASSWORD: str = ""

    # --- Google Maps settings
    GOOGLE_API_KEY: str

    @property
    def postgres_uri(self) -> str:
        return (
            f"postgresql+asyncpg://"
            f"{self.POSTGRES_USER}:"
            f"{self.POSTGRES_PASSWORD}@"
            f"{self.POSTGRES_HOST}:"
            f"{self.POSTGRES_PORT}/"
            f"{self.POSTGRES_DB}"
        )

    # --- Redis settings
    REDIS_USER: str = ""
    REDIS_PASSWORD: str
    REDIS_HOST: str = "localhost"
    REDIS_PORT: int = 6379

    @property
    def redis_uri(self) -> str:
        return (
            f"redis://"
            f"{self.REDIS_USER}:"
            f"{self.REDIS_PASSWORD}@"
            f"{self.REDIS_HOST}:"
            f"{self.REDIS_PORT}"
        )

    @property
    def protocol(self):
        return {True: "https", False: "http"}[self.SECURE_PROTOCOL]

    @property
    def application_url(self) -> str:
        return f"{self.protocol}://{self.APPLICATION_HOST}:{self.APPLICATION_PORT}"

    @property
```

```
    def BASE_DIR(self) -> Path:
        return BASE_DIR

    class Config:
        env_file = BASE_DIR / ".env"

    def enable_production(self):
        self.PRODUCTION = True
        self.DB_ECHO = False
        self.SHOW_SWAGGER = False

    def disable_production(self):
        self.PRODUCTION = False
        self.DB_ECHO = True

    def update_production_mode(self):
        if self.PRODUCTION:
            self.enable_production()
        else:
            self.disable_production()

    def __post_init_post_parse__(self):
        self.update_production_mode()


settings = Settings()
```

3.5.2 Конфігурація CORS

Для того щоб frontend-частина, створена на React, могла безперешкодно взаємодіяти з бекендом, було налаштовано Cross-Origin Resource Sharing (CORS). Цей механізм дає змогу контролювати, які саме джерела (домен/порт) мають право надсилати запити до API. У процесі реалізації було задано список дозволених походжень (origins), а також відкрито доступ до необхідних HTTP-методів і заголовків. Це критично важливо в умовах, коли фронтенд і бекенд працюють на різних хостах або портах під час розробки чи розгортання в продакшн.

```python
_app.add_middleware(
    CORSMiddleware,
    allow_origins=[settings.ALLOWED_ORIGIN],
    allow_credentials=True,
    allow_methods=["GET", "POST"],
    allow_headers=["*"],
    expose_headers=["set-cookie"],
)
```

Рисунок 3.29 – Налаштування конфігурації CORS

3.5.3 Автоматична генерація документації API через OpenAPI та Swagger

Однією з ключових переваг FastAPI є автоматичне формування документації на основі специфікації OpenAPI, яка не тільки відображає повний перелік доступних маршрутів, але й дозволяє взаємодіяти з ними в реальному часі через візуальний інтерфейс **Swagger UI**. Це особливо корисно під час розробки або ручного тестування запитів, оскільки розробники та QA можуть одразу бачити, які дані очікує і повертає кожна точка доступу.

Також підтримується альтернатива у вигляді **ReDoc** — статичної, але більш презентабельної документації, яка ідеально підходить для технічної демонстрації або інтеграції сторонніх команд, що не мають доступу до коду.

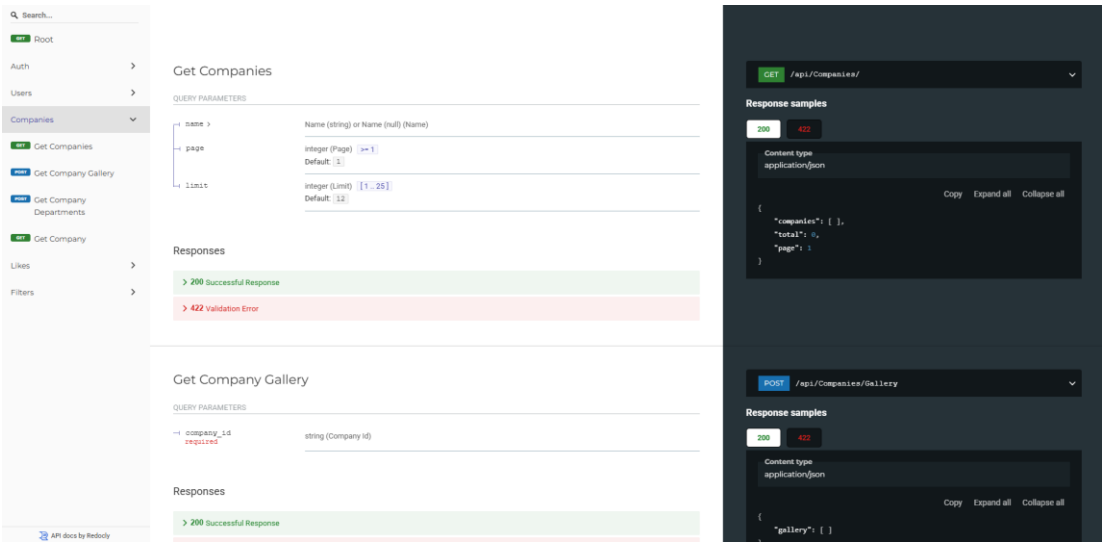


Рисунок 3.30 – Redoc документація

Важливо, що завдяки типізованим моделям **Pydantic**, документація не просто містить назви полів, а й автоматично включає приклади запитів та відповідей, дозволяючи миттєво зрозуміти формат payload'ів без необхідності вручну створювати OpenAPI-файли.

Щоб уникнути потенційних витоків інформації у продакшн-середовищі, було реалізовано гнучкий механізм відключення документації. За допомогою умовного параметра *SHOW_SWAGGER*, який конфігурується через *.env* або змінні середовища, система визначає, чи потрібно відкривати такі маршрути, як:

```
        docs_url="/docs" if settings.SHOW_SWAGGER else None,
        redoc_url="/redoc" if settings.SHOW_SWAGGER else None,
        openapi_url="/openapi.json" if settings.SHOW_SWAGGER else None,
```

Таким чином, у робочому середовищі, де API не має бути публічно задокументованим, всі інтерфейси доступу до документації **повністю вимикаються**, забезпечуючи додатковий рівень безпеки для внутрішніх систем або конфіденційних сервісів.

3.5.4 Журналювання подій, діагностика збоїв

Ефективне функціонування API передбачає не лише обробку вхідних запитів, а й глибоке логування внутрішніх процесів, що відбуваються під час їх виконання. З цією метою у проєкті реалізовано розширену систему ведення журналів подій за допомогою сторонньої бібліотеки **Loguru**, яка надає гнучкий та виразний інтерфейс для збору діагностичної інформації.

На відміну від класичного *logging* з великою кількістю конфігураційних складностей, Loguru дозволяє за лічені рядки коду налаштувати одночасне логування як у консоль, так і у файл. У нашому рішенні передбачено збереження логів у файл з автоматичною ротацією (до 10 МБ) та архівацією (.zip). Це дозволяє

вести довготривалий облік подій без ризику перевищення обсягу диску або втрати важливої інформації.

Кожен лог містить:

- час події (з точністю до секунди),

- рівень важливості (DEBUG, INFO, WARNING, ERROR),

- назву модуля та функції, яка його викликала,

- текст повідомлення.

Подібний підхід дозволяє не лише швидко знаходити першопричину збоїв у логах, а й використовувати журнал для аудиту, відстеження активності, навантаження чи потенційних точок атаки.

Таким чином, логування у цьому проєкті виступає не як формальність, а як інструмент контролю якості, безпеки та зручності супроводу впродовж усього життєвого циклу розробки. [15]

```python
class Logger:
    _logger = None

    @classmethod
    def get_logger(cls):
        loguru_logger.remove()
        if cls._logger is None:
            log_format = (
                "<green>{time:YYYY-MM-DD HH:mm:ss}</green> | "
                "<level>{level}</level> | "
                "<cyan>{name}</cyan>:<cyan>{function}</cyan>:<cyan>{line}</cyan> - "
                "<level>{message}</level>"
            )

            loguru_logger.add(sys.stdout, format=log_format, level=settings.LOG_LEVEL)

            loguru_logger.add(
                settings.LOG_FILE_PATH,
                format=log_format,
                level=settings.LOG_LEVEL,
                rotation="10 MB",
                compression="zip"
            )

            cls._logger = loguru_logger

        return cls._logger


logger = Logger.get_logger()
```

Рисунок 3.31 – Реалізація централізованого логеру

Приклад повідомлення в логах: 2025-06-06 20:49:12 2025-06-06 17:49:12 | DEBUG | app.infrastructure.redis.client:save_to_cache:74 - ✅ Cache saved: parser:list_in_ua -> - (TTL: 86400)

3.6 Клієнтська частина: React + TailwindCSS

Фронтенд застосунку реалізовано за допомогою сучасного JavaScript-фреймворку React, у зв'язці з високошвидкісним білдером **Vite**, що забезпечує блискавичну збірку та зручність у розробці. Для стилізації інтерфейсу використано **TailwindCSS** — утилітарну CSS-бібліотеку, яка дозволяє створювати адаптивний і мінімалістичний дизайн без необхідності писати вручну великі обсяги CSS-коду.

Основи конфігурації та структура проєкту

Проєкт підтримує строгі стандарти якості коду завдяки **ESLint**, а обробка стилів здійснюється через **PostCSS**, що дозволяє розширювати CSS через плагіни та оптимізації. Центральне місце займає *vite.config.js*, який конфігурує поведінку білдера, включно з проксі, плагінами та шляхами. Налаштування стилів Tailwind знаходяться у *tailwind.config.js*, де визначаються кольорові палітри, шрифти та кастомні класи.

Архітектура застосунку

Структура коду чітко сегментована на функціональні блоки:

- **Файли запуску** (main.jsx, App.jsx) задають вхідну точку програми, ініціалізують маршрутизацію та загальне компонування.

```jsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient();

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  </StrictMode>,
)
```

Рисунок 3.32 – main.jsx

```jsx
function App() {
  return (
    <Router>
      <SearchFiltersProvider>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about/:id" element={<AboutCompany />} />
          <Route path="/liked" element={<LikedPage />} />
        </Routes>
      </SearchFiltersProvider>
    </Router>
  )
}
```

Рисунок 3.33 – Компонента App.jsx

- **Глобальні стилі** організовано через index.css та App.css, а також розширено Tailwind-утилітами.

Компонентна модель

Вся логіка та UI розбито на модулі з розділенням відповідно до зони відповідальності:

- **Макетні компоненти** (*Header, Footer, Wrapper, Sidebar*) відповідають за структуру сторінки, шапку, футер та бічне меню.

- **Функціональні блоки** (напр., *Gallery, LikedPage, AboutCompany, OrganizationHero, Home*) реалізують основні сценарії взаємодії з платформою — перегляд компаній, збереження обраного, виведення інформації.
- **Локаційні компоненти** *(LocationMap, LocationContainer, CircleMarker*) працюють з відображенням географічних даних, використовуючи інтеграцію з Google Maps API.
- **Універсальні UI-компоненти** (*Button, Input, Spinner, Pagination*) є багаторазовими елементами, що можуть бути використані у різних частинах застосунку.
- **Фільтри та візуальні секції** (*FiltersSidebar, HeroBanner, Content*) створені для забезпечення зручної навігації, пошуку та представлення контенту. [11, 12, 16]

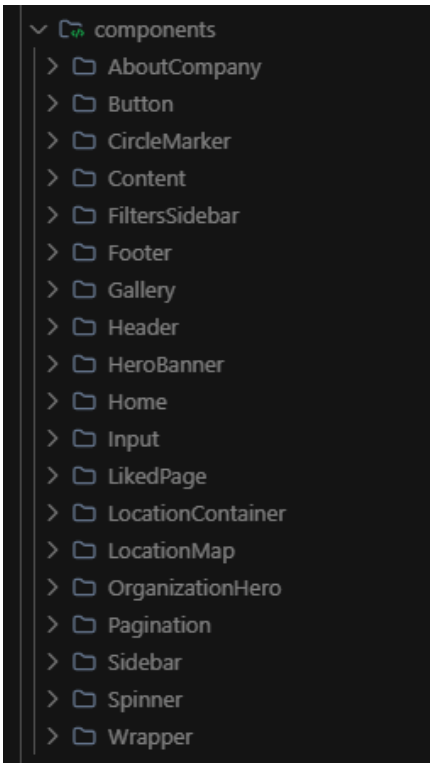


Рисунок 3.34 – Перелік власних компонентів React WebApp

3.6.1 API-взаємодія: Axios + React Query

Для взаємодії з бекендом клієнтська частина використовує бібліотеку **Axios** як HTTP-клієнт, а також ***@tanstack/react-query*** для керування кешуванням, станами запитів і повторною спробою отримання даних. Це дозволяє досягти високої ефективності при рендері та уникає повторних запитів до API, якщо дані вже були отримані нещодавно.

Усі запити до API інкапсульовано у спеціальні сервіси (*api/companies.js, api/filters.js*), що спрощує повторне використання логіки запитів і розмежовує відповідальність компонентів. Наприклад, *useCompanies(params)* дозволяє автоматично кешувати результати запиту */api/Companies* і забезпечує автоматичну інвалідацію даних при зміні фільтрів чи параметрів.

```
export const useCompanies = (params = {}) => {
  return useQuery({
    queryKey: ['companies', params],
    queryFn: () => fetchCompanies(params),
    staleTime: 1000 * 60 * 60,
    cacheTime: 1000 * 60 * 60 * 2,
  });
};
```

Рисунок 3.35 – Приклад використання useQuery із tanstack/react-query для кешування запитів

Цей підхід забезпечує кращу масштабованість застосунку та відокремлення логіки отримання даних від компонентів UI.

3.6.2 Система фільтрації: глобальний контекст та костомні хуки

Для роботи з фільтрами реалізовано глобальний контекст *SearchFiltersContext*, який дозволяє централізовано зберігати стан вибраних категорій, районів, текстового пошуку та поточної сторінки. Це забезпечує

синхронізацію між різними компонентами (наприклад, боковою панеллю фільтрів, відображенням картки компаній і пагінацією).

```
const SearchFiltersContext = createContext();
return (
    <SearchFiltersContext.Provider
      value={{
        searchQuery,
        setSearchQuery,
        filters,
        setFilters,
        page,
        setPage,
      }}
    >
      {children}
    </SearchFiltersContext.Provider>
  );
};

export const useSearchFilters = () => {
  const context = useContext(SearchFiltersContext);
  if (!context) throw new Error("useSearchFilters must be used within Provider");
  return context;
};
```

Рисунок 3.36 – Часткова демонстрація реалізації власного контексту

Функціональність маніпулювання фільтрами інкапсульована в кастомному хуку *useFilterState*, який містить методи *toggleDistrict, toggleCategory, removeSelectedMarker* та інші. Це дозволяє логіку фільтрації тримати окремо від візуального представлення, що покращує підтримку коду.

```
import { useSearchFilters } from "./useSearchFilters";

export function useFilterState(filtersData) {
  const {
    filters: selected,
    setFilters: setSelected,
    setPage
  } = useSearchFilters();

  const toggleDistrict = (district) => {
    const next = selected.districts?.includes(district)
      ? selected.districts.filter(r => r !== district)
      : [...(selected.districts || []), district];

    setSelected({ ...selected, districts: next });
    setPage(1);
  };

  const toggleCategory = (group, sub) => {
    const groupSelected = selected.categories?.[group] || [];
    const next = groupSelected.includes(sub)
      ? groupSelected.filter(i => i !== sub)
      : [...groupSelected, sub];

    const updatedCategories = {
      ...(selected.categories || {}),
      [group]: next,
    };
```

```
    const updated = { ...(selected.categories || {}) };
    for (const group in updated) {
      updated[group] = updated[group].filter(i => i !== value);
      if (updated[group].length === 0) delete updated[group];
    }

    setSelected({ ...selected, categories: updated });
  };

  const selectedMarkers = [
    ...(selected.districts || []),
    ...Object.values(selected.categories || {}).flat()
  ];

  return {
    selected,
    selectedMarkers,
    toggleDistrict,
    toggleCategory,
    toggleWholeCategory,
    removeSelectedMarker,
    isGroupChecked
  };
}
```

Рисунок 3.37 – Часткова демонстрація реалізації хуку useFilterState

Використання хуку:
const { toggleCategory, selectedMarkers } = useFilterState(filtersData);

3.6.3 Відображення на карті: компоненти з Google Maps

Геопросторове відображення реалізовано за допомогою **Google Maps JavaScript API** через бібліотеку *@react-google-maps/api*. Основний компонент LocationMap відповідає за ініціалізацію карти та розміщення маркерів на основі координат, отриманих від API. Після натискання на маркер відкривається *InfoWindow* з назвою організації.

Карта рендериться лише після повного завантаження координат. Компонент *LoadScriptNext* використовується для відкладеного завантаження скриптів Google Maps, що зменшує початкове навантаження на сторінку. Це покращує продуктивність і дозволяє уникати зайвих запитів до Google API. [10]

```
<LoadScriptNext
  googleMapsApiKey={import.meta.env.VITE_GOOGLE_MAPS_API_KEY}
  language="uk"
  region="UA"
>
  <GoogleMap
    mapContainerStyle={containerStyle}
    center={center}
    zoom={13}
  >
    {locations.map(loc => {
      if (!loc.department.address_lat || !loc.department.address_long) {
        return null;
      }
      return (
        <Marker
          key={loc.org_id}
          position={{lat: +loc.department.address_lat, lng: +loc.department.address_long}}
          onClick={() => setActiveMarker(loc.org_id)}
        >
          {activeMarker === loc.org_id && (
            <InfoWindow onCloseClick={() => setActiveMarker(null)}>
              <div className="text-sm font-semibold">{loc.name}</div>
            </InfoWindow>
          )}
        </Marker>
      );
    })}
  </GoogleMap>
</LoadScriptNext>
```

Рисунок 3.38 – Компонента для відображення інтерактивної карти

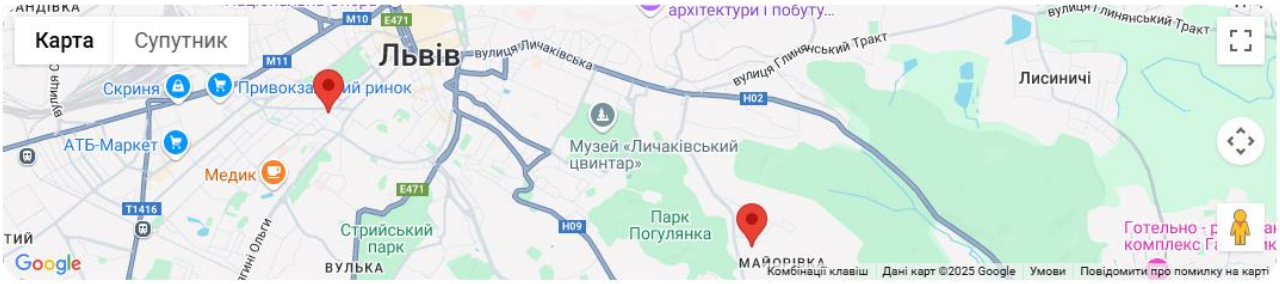


Рисунок 3.39 – Відображення карти на сайті

3.6.4 Обробка глобального стану URL та пагінація

Стан пошуку та пагінації синхронізується з URL за допомогою *useSearchParams* з *react-router-dom*. Це забезпечує стабільність стану між перезавантаженнями сторінки, дає змогу ділитися посиланнями на конкретні фільтри та сторінки, а також позитивно впливає на SEO та аналітику поведінки користувачів.

```
const [searchParams, setSearchParams] = useSearchParams();

const paramSearch = searchParams.get("s") || "";
const paramPage = Number(searchParams.get("p")) || 1;
```

Рисунок 3.40 – Отримання стану із URL параметрів

Для візуального перемикання сторінок використовується компонент Pagination, який підтримує динамічний рендер сторінок з багатьма варіантами (початок, середина, кінець), а також ручний перехід до потрібного номера сторінки. Це забезпечує UX-навігацію навіть при великій кількості результатів.

```
<Pagination
 totalPages={Math.ceil(locations.total / 5)}
 onPageChange={onPageChange}
 currentPage={locations.page}
/>
```

Рисунок 3.41 – Використання компоненти Pagination

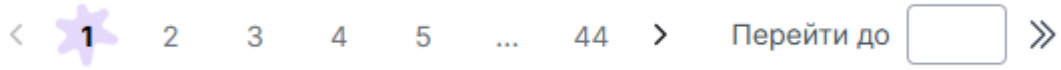


Рисунок 3.42 – Відображення пагінації на сайті

3.6.5 Локальне збереження вподобаних локацій (localStorage)

На поточному етапі застосунок підтримує можливість збереження вподобаних компаній на клієнтській стороні за допомогою *localStorage*. У компоненті *LocationItem* кнопка з іконкою серця дозволяє додати або прибрати локацію з вподобаних. Ідентифікатори обраних організацій зберігаються у локальному сховищі браузера (*liked_location_ids*).

Цей підхід не потребує авторизації, однак зберігає дані лише на одному пристрої. У майбутньому функціональність може бути перенесена на серверну частину — з прив'язкою до облікового запису користувача, що дозволить синхронізувати вподобання між різними пристроями.

```
const toggleLike = (e) => {
  e.stopPropagation();
  setLiked((prev) => {
    const stored = JSON.parse(localStorage.getItem("liked_location_ids") || "[]");
    let updated;

    if (prev) {
      updated = stored.filter((id) => id !== location.org_id);
    } else {
      updated = stored.includes(location.org_id) ? stored : [...stored, location.org_id];
    }

    localStorage.setItem("liked_location_ids", JSON.stringify(updated));
    return !prev;
  });
};
```

Рисунок 3.43 – Реалізація функції toggleLike для збереження/видалення локації із локального сховища браузера

3.7 Інтерфейс користувача: дизайн та зручність використання

Інтерфейс побудований з акцентом на простоту, логічність і зручність для користувача. Компоненти мають чітку ієрархію, великі інтерактивні зони та адаптивне компонування, що забезпечує комфортну взаємодію як на мобільних, так і на десктопних пристроях.

Основні елементи, такі як кнопки, форми, навігація та картки локацій, витримані в єдиному стилі та реагують на дії користувача — наприклад, зміну сторінки або додавання у вподобане. Всі важливі стани (завантаження, активні елементи) відображаються наочно.

Компоненти UI створено як незалежні блоки з можливістю повторного використання, що дозволяє швидко масштабувати інтерфейс і зберігати візуальну цілісність у всьому проєкті.

3.7.1 Головна сторінка

Головна сторінка є центральним хабом для взаємодії з платформою. Вона містить пошуковий рядок, блок із рекомендаціями, а також розширені фільтри за категоріями, районами та локацією, що дозволяють швидко звузити перелік доступних гуртків. Інтерфейс побудований так, щоб навіть новий користувач міг інтуїтивно орієнтуватися і швидко знаходити потрібне. Всі зміни у фільтрах одразу відображаються в результатах, що робить процес максимально динамічним.

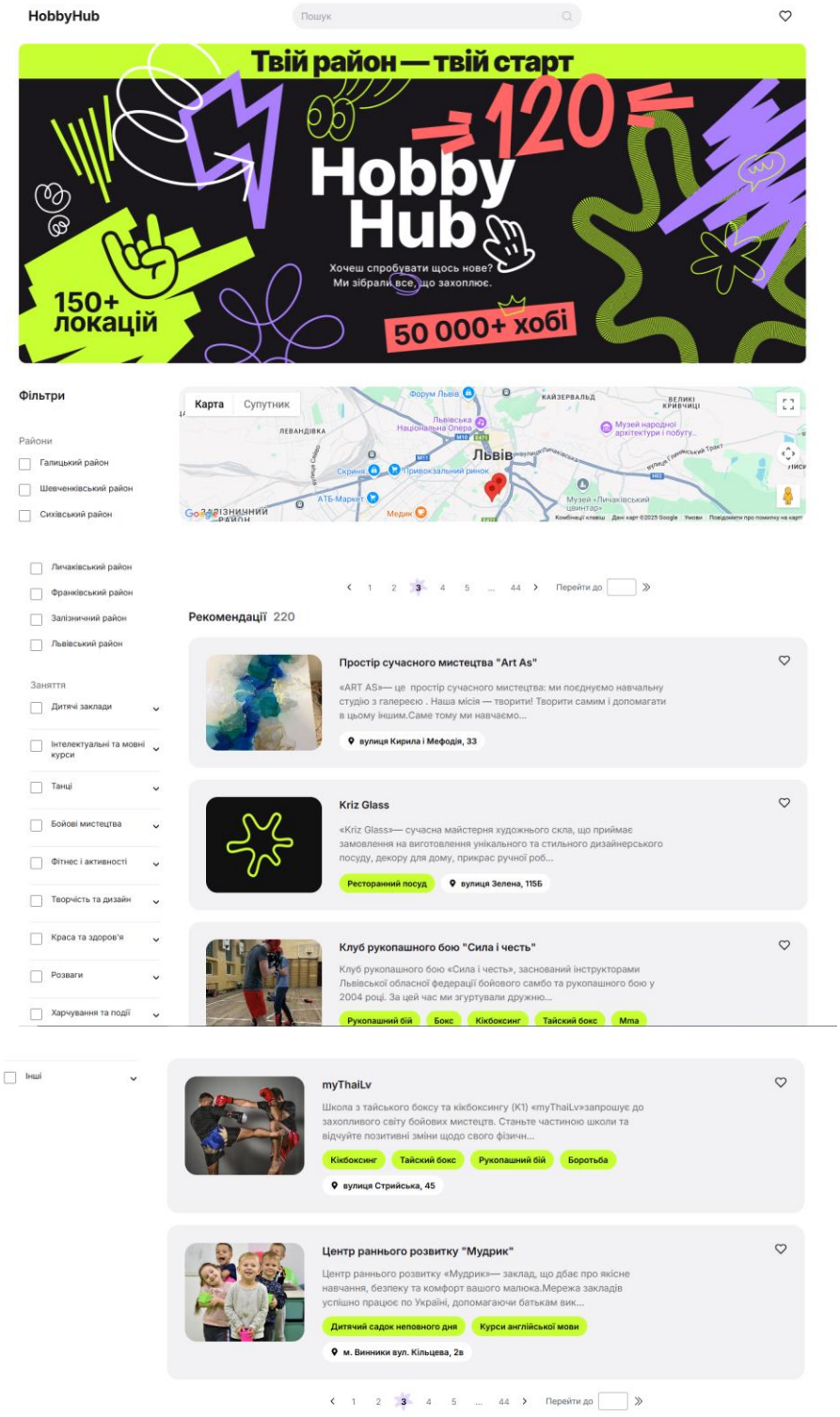


Рисунок 3.44 – Відображення головної сторінки

3.7.2 Перегляд гуртка

Сторінка перегляду гуртка містить детальну інформацію про вибрану організацію або секцію. Тут представлено опис, категорії, контактні дані, всі локації, а також візуальні елементи, що допомагають краще уявити діяльність гуртка.

Рисунок 3.45 – Сторінка перегляду гуртка

3.7.3 Галерея

Галерея — це візуальний розділ на сторінці гуртка, в якому зображення подані у вигляді адаптивної сітки. При натисканні на будь-яке зображення відкривається повноекранний перегляд із підтримкою перемикання між фото (пагінація), що дозволяє зручно гортати матеріали навіть на мобільних пристроях. Система забезпечує комфортне візуальне ознайомлення з діяльністю гуртка.

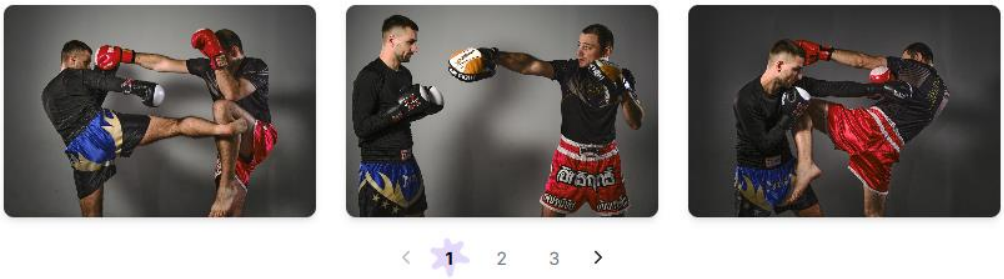


Рисунок 3.46 – Перегляд галереї на сторінці гуртка

3.7.4 Збережені локації

Ця сторінка відображає вподобані користувачем локації, які були збережені локально через інтерактивну кнопку «серце». Користувач може швидко переглянути всі свої обрані гуртки в одному місці. Як згадувалося раніше, дані зберігаються в *localStorage* без авторизації, однак у майбутньому ця функціональність може бути перенесена на бекенд для синхронізації між пристроями. Щоб перейти до «вподобаних» локацій потрібно натиснути на іконку (серця) справа зверху сторінки.

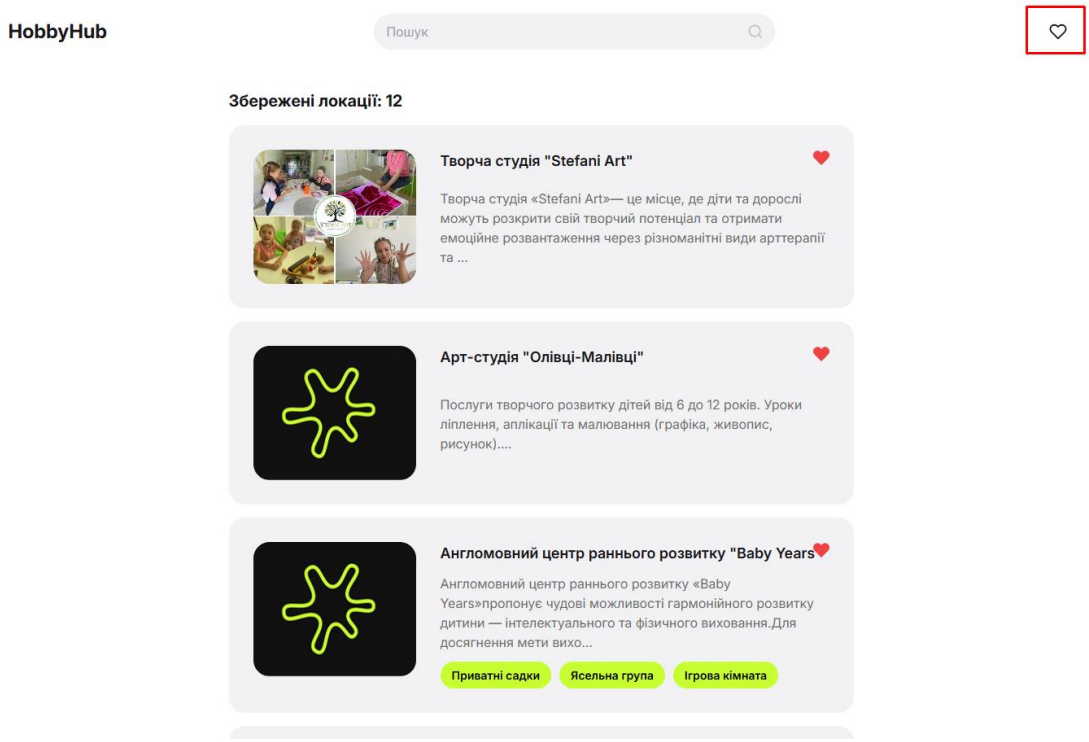


Рисунок 3.47 – Перегляд збережених локацій

3.8 Docker-орієнтована інфраструктура розгортання

Для ізольованого та відтворюваного запуску всіх ключових компонентів застосунку використовується Docker. У репозиторії присутні конфігураційні файли Dockerfile та docker-compose.yml, які забезпечують контейнеризацію бекенду, бази даних PostgreSQL і кеш-сервісу Redis.

Dockerfile: створення бекенд-образу

Файл Dockerfile описує побудову образу для Python-застосунку на основі офіційного образу python:3.11. У контейнер копіюються всі необхідні файли проєкту: скрипти міграцій Alembic, конфігурація Redis, список залежностей та скрипт запуску entrypoint.sh. Після встановлення залежностей (requirements.txt), контейнер стає повністю готовим до виконання FastAPI-застосунку.

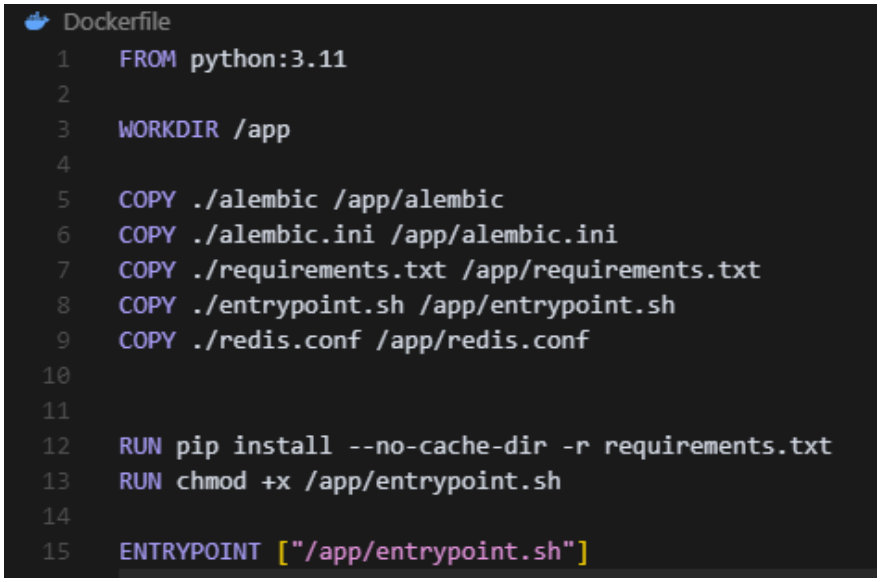


Рисунок 3.48 – Вміст файлу Dockerfile

Цей підхід дозволяє не лише автоматизувати процес налаштування середовища, але й легко масштабувати систему або переносити її на будь-який сервер без ручного налаштування Python-оточення.

docker-compose: управління сервісами

Файл *docker-compose.yml* організовує три ключові сервіси:

- Backend – сам FastAPI застосунок
- Db – база даних PostgreSQL
- Redis – кеш-сховище для оптимізації обробки запитів та сесій

Кожен сервіс ізольований у власному контейнері, але об'єднаний у спільну мережу backend, що забезпечує їхню взаємодію. [7, 8]

Рисунок 3.48 – Вміст файлу docker-compose.yml

Бекенд стартує лише після запуску Redis і PostgreSQL, що гарантує коректну ініціалізацію. При цьому всі змінні середовища зчитуються з .env-файлу, що дозволяє зручно конфігурувати поведінку системи без зміни коду.

Особливість: розділення фронтенду

Фронтенд-частина застосунку запускається окремо на іншому сервері, оскільки не є частиною цієї Docker-інфраструктури. Це рішення прийнято з міркувань масштабованості: воно дозволяє незалежно оновлювати клієнтську та серверну частину, а також зручно адаптувати їх під різні хостинг-умови (наприклад, фронт можна легко задеплоїти через Netlify, Vercel або окремий Nginx-сервер).

3.9 Тестування

Проєкт проходив кілька етапів тестування, спрямованих на виявлення логічних помилок, збоїв у роботі інтерфейсу, нестабільності API та недоліків взаємодії з користувачем. Оскільки автоматизовані юніт-тести не були реалізовані, основна увага була зосереджена на **ручному та інтеграційному тестуванні** на різних етапах розробки.

3.9.1 Ручне функціональне тестування

Після реалізації кожного модуля проводилось ручне тестування відповідного функціоналу. Особлива увага приділялась таким аспектам:

- правильність відображення фільтрів і їх взаємодія з картою та результатами;
- робота пагінації — як вона оновлює URL і дані;
- збереження та відновлення вподобаних локацій через localStorage;
- відповідність даних у компоненті перегляду гуртка отриманим через API.
- Тестування проводилося на кількох браузерах (Chrome, Firefox) і пристроях (мобільних/десктопних) для перевірки адаптивності та стабільності рендеру.

3.9.2 Інтеграційне тестування бекенд-фронтенд

Було протестовано взаємодію між клієнтською частиною і сервером у реальному середовищі Docker. Основні сценарії включали:

- запити до API компаній з параметрами фільтрації;
- динамічне оновлення компонентів при зміні searchParams у URL;
- правильність реакції на помилки сервера (наприклад, при відсутності даних);
- перевірка валідності CORS-заголовків при запуску фронтенду на іншому домені.

3.9.3 Тестування зручності (UX) і поведінкові тести

З метою покращення користувацького досвіду були залучені сторонні користувачі, які не брали участі в розробці. Їм було запропоновано знайти гуртки за фільтрами, перейти на сторінку перегляду, зберегти локацію, а також використати карту.

Було зібрано фідбек щодо:

- надто дрібних елементів інтерфейсу на мобільних;
- заплутаного повернення назад зі сторінки перегляду гуртка.

На основі цього було виконано оптимізацію візуальної ієрархії та покращено навігацію (добавлено кнопку назад на сторінці локації).

3.9.4 Виявлені баги та їх усунення

Під час тестування було знайдено кілька критичних і незначних помилок, які були оперативно виправлені:

Баг 1: При переході на іншу сторінку в пагінації скидалися активні фільтри.

→ Було впроваджено глобальний стан фільтрів через контекст *SearchFiltersContext*, який зберігає значення навіть після оновлення сторінки.

Баг 2: У деяких випадках бекенд повертав помилку 500, якщо користувач переходив за неіснуючим ID організації.

→ Було додано перевірку існування компанії перед спробою її відображення, з поверненням помилки 404 у разі відсутності.

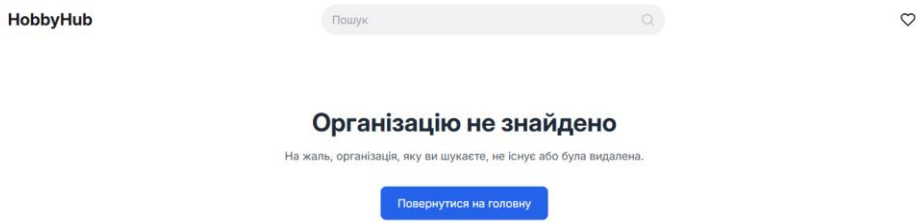


Рисунок 3.49 – Додано відображення повідомлення про те, що організацію не знайдено.

Баг 3: Некоректна обробка запиту з порожнім параметром category на бекенді не повертала жодного запису, замість всіх.

→ Було додано логіку, яка ігнорує порожні параметри фільтрації в запиті по замовчуванню вважаючи що на ці фільтри не звертати увагу.

Баг 4: На сторінці перегляду гуртка некоректно відображалася адреса, якщо вона містила дужки.

→ Реалізована утиліта removeBracketsFromAddress, що очищує адресу від непотрібних символів перед рендером.

```
export const removeBracketsFromAddress = (address) => {
    return address.replace(/\s*\(.*?\)\s*/g, '');
}
```

Рисунок 3.50 – Утиліта для видалення дужок

Баг 5: Після скидання фільтрів користувач все одно залишався на тій сторінці пагінації, на якій був.

→ Було реалізовано автоматичне скидання на першу сторінку (p=1) при зміні активних фільтрів.

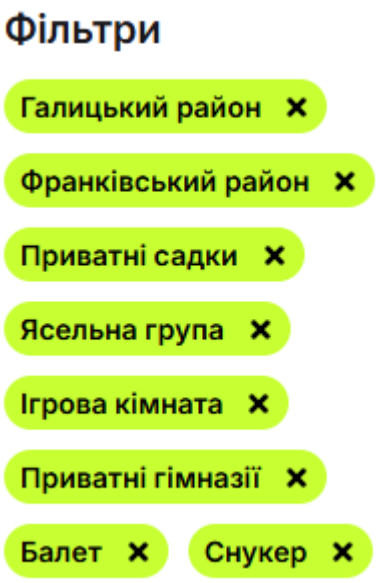


Рисунок 3.51 – Відображення вибраних фільтрів

ВИСНОВКИ

У процесі виконання дипломної роботи на тему «Розробка вебзастосунку для зручного пошуку гуртків та секцій із використанням сучасного стеку технологій» були реалізовані всі поставлені цілі та виконані ключові завдання дослідження.

Досягнуті результати:

- Проведено детальне дослідження предметної області — пошуку позашкільних освітніх послуг, визначено основні виклики користувачів і обґрунтовано потребу у простому, візуально зручному та фільтрованому інтерфейсі.
- Вибір технологій (FastAPI, PostgreSQL, React) дозволив побудувати ефективну архітектуру, що забезпечує швидкий обмін даними, масштабованість та хорошу підтримку в екосистемі.
- Створено зручний бекенд-API з документацією OpenAPI, налаштуванням CORS, логуванням подій, асинхронною взаємодією з базою даних і налаштованим середовищем Docker.
- Реалізовано повноцінний frontend-інтерфейс, що охоплює фільтрацію, перегляд детальної інформації про гуртки, адаптивну галерею, а також можливість зберігати улюблені локації — з потенціалом переходу до авторизованої взаємодії у майбутньому.
- Інтегровано карту Google Maps із підтримкою маркерів, інформаційних вікон та геолокаційних пошуків.
- Проведено тестування основних сценаріїв взаємодії користувача з системою, знайдено та усунуто технічні недоліки як на фронтенді, так і на бекенді.

Узагальнення:

Розроблений вебзастосунок є сучасним інструментом для швидкого та зручного пошуку гуртків за геолокацією, категорією та описом. Завдяки

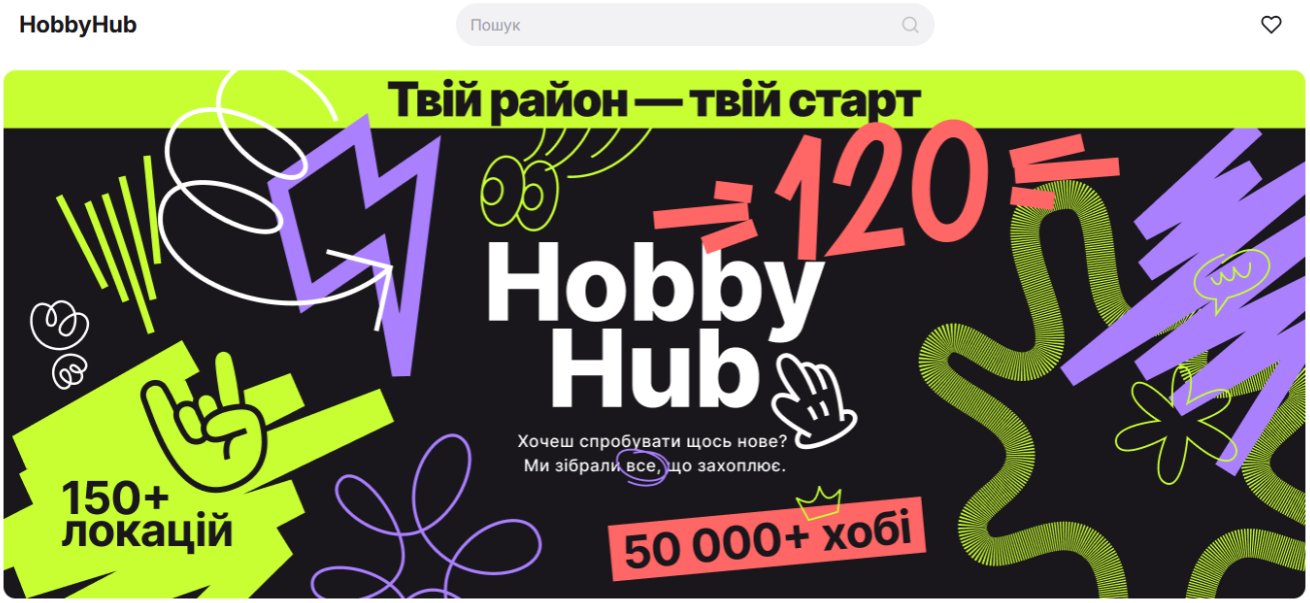
інтерактивному інтерфейсу, мапі, можливості збереження вподобаного, система має потенціал стати корисною платформою для батьків, дітей та організацій.

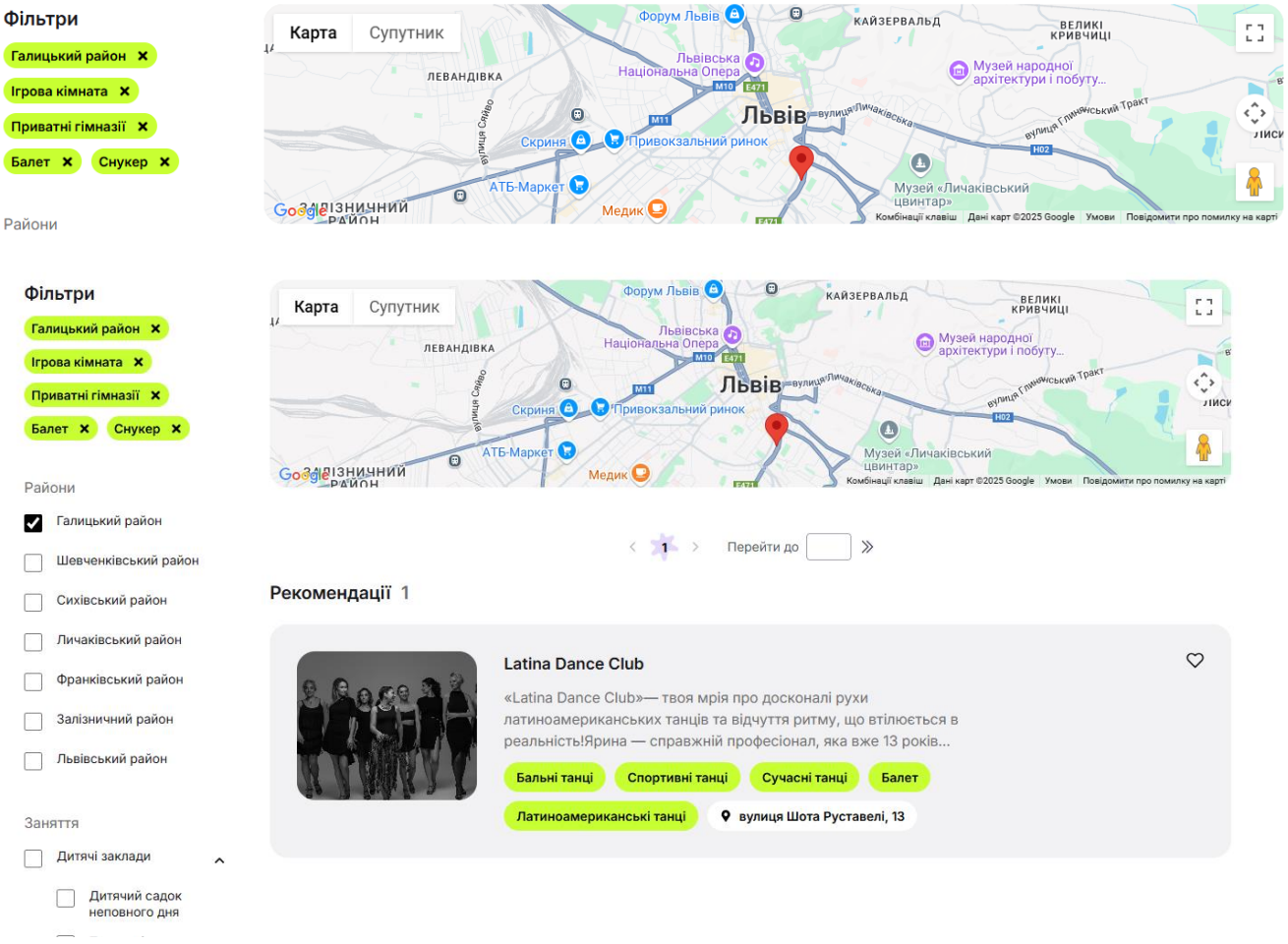
Таким чином, дипломна робота засвідчує практичну цінність запропонованого рішення, його адаптивність до реальних потреб користувачів та готовність до подальшого розвитку — включно з реалізацією авторизації, особистого кабінету та розширеного адміністрування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. FastAPI Official Documentation – https://fastapi.tiangolo.com/
2. Pydantic Official Documentation – https://docs.pydantic.dev/
3. Python Official Documentation – https://docs.python.org/3/
4. Рамальо Л. – "Fluent Python: Чітке, лаконічне та ефективне програмування" – O'Reilly Media, 1014 с., 2022.
5. PostgreSQL Official Documentation – https://www.postgresql.org/docs/
6. SQLAlchemy Documentation – https://docs.sqlalchemy.org/
7. Docker Documentation – https://docs.docker.com/
8. Docker Compose Specification – https://docs.docker.com/compose/
9. BeautifulSoup Documentation – https://www.crummy.com/software/BeautifulSoup/bs4/doc/
10. Google Maps JavaScript API Documentation – https://developers.google.com/maps/documentation/javascript
11. React Official Documentation – https://react.dev/
12. Tailwind CSS Documentation – https://tailwindcss.com/docs
13. Мартін Р. – "Clean Architecture: Архітектура чистого коду" – Prentice Hall, 432 с., 2017.
14. Beekeeper Studio Documentation – https://www.beekeeperstudio.io/docs
15. Мартін Р. – "Clean Code: Посібник із гнучкої розробки програмного забезпечення" – Prentice Hall, 464 с., 2008.
16. Vercel Deployment Documentation – https://vercel.com/docs
17. "Full Stack FastAPI and PostgreSQL" Tutorial Series by Sebastián Ramírez – https://github.com/tiangolo/full-stack-fastapi-postgresql

ДОДАТКИ


HobbyHub
Пошук
Твій район — твій старт
120
Hobby Hub
Хочеш спробувати щось нове?
Ми зібрали все, що захоплює.
150+ локацій
50 000+ хобі


Фільтри
Галицький район
Ігрова кімната
Приватні гімназії
Балет
Снукер
Райони
Карта
Супутник
Форум Львів
Львівська Національна Опера
Музей народної архітектури і побуту...
Львів
Скриня
Привокзальний ринок
АТБ-Маркет
Медик
Музей «Личаківський цвинтар»
Фільтри
Галицький район
Ігрова кімната
Приватні гімназії
Балет
Снукер
Райони
Галицький район
Шевченківський район
Сихівський район
Личаківський район
Франківський район
Залізничний район
Львівський район
Заняття
Дитячі заклади
Дитячий садок неповного дня
Карта
Супутник
Форум Львів
Львівська Національна Опера
Музей народної архітектури і побуту...
Львів
Скриня
Привокзальний ринок
АТБ-Маркет
Медик
Музей «Личаківський цвинтар»
Перейти до
Рекомендації 1
Latina Dance Club
«Latina Dance Club»— твоя мрія про досконалі рухи латиноамериканських танців та відчуття ритму, що втілюється в реальність!Ярина — справжній професіонал, яка вже 13 років...
Бальні танці
Спортивні танці
Сучасні танці
Балет
Латиноамериканські танці
вулиця Шота Руставелі, 13

HobbyHub

Пошук

Фільтри

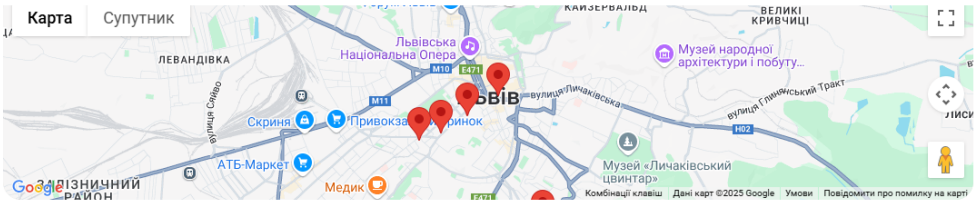
Галицький район ×

Райони

- [x] Галицький район
- [ ] Шевченківський район
- [ ] Сихівський район
- [ ] Личаківський район
- [ ] Франківський район
- [ ] Залізничний район
- [ ] Львівський район

Заняття

- [ ] Дитячі заклади
  - [ ] Дитячий садок неповного дня
  - [ ] Приватні садки
  - [ ] Ясельна група
  - [ ] Приватні школи
  - [ ] Приватні ліцеї
  - [ ] Ігрова кімната
  - [ ] Приватні гімназії



1 2 3 4 5 ... 10 Перейти до

Рекомендації 47

Мистецтво ідеї

Проведення занять, майстер-класів з живопису та малювання для дорослих і дітей. Гуртки....

вулиця Степана Бандери, 11

Львівський міський дитячий еколого-натуралістичний центр

Гуртки еколого-натуралістичного спрямування для дітей. Клуб акваріумістів. Зоопарк. Ботанічна пам'ятка природи місцевого значення "Дендропарк Бенедикта Дибовського". Літн...

вулиця Кубанська, 12

Etude Art Studio

Стильна«Etude Art Studio», розташована у самому центрі Львова, чекає на талановитих, захоплених мистецтвом та творчістю учнів. Якщо ви займаєтесь професійно рисунком, жив...

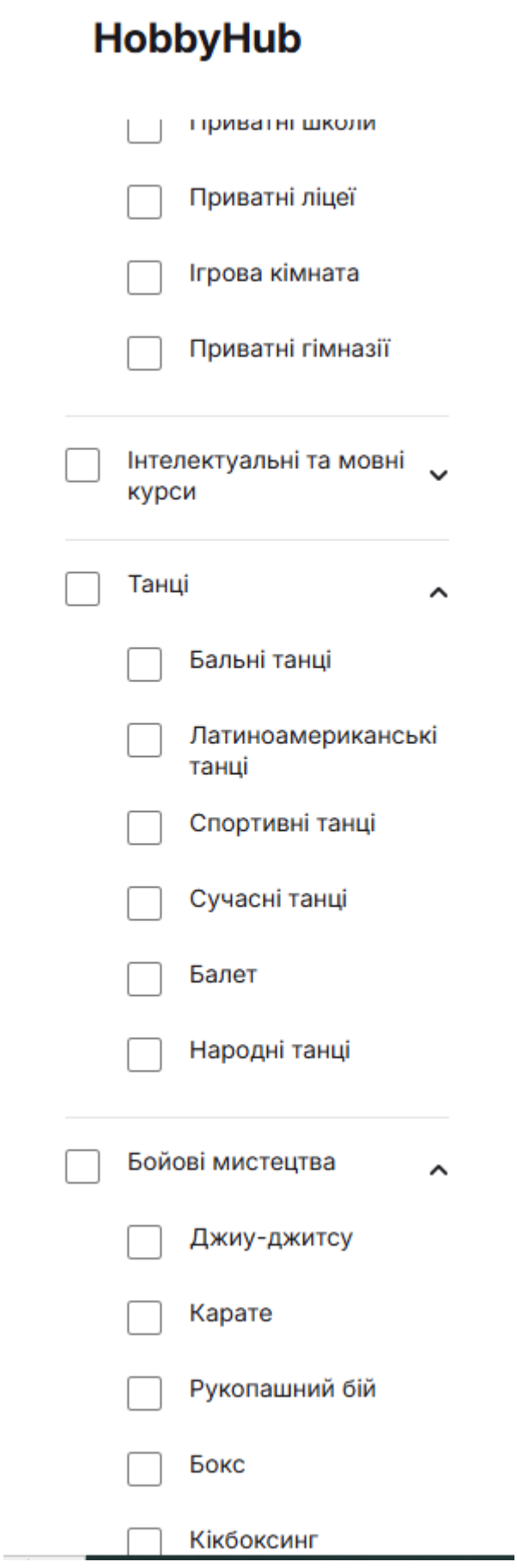
вулиця Валова, 5

Дитячий простір "Совенята"

Дитячий простір «Совенята»запрошує діток від 2 до 6 років на цікаві розвивальні заняття. Наш заклад спеціалізується на інклюзивній освіті, забезпечуючи індивідуальний під...

Дитячі психологи | Дитячий садок неповного дня | Ясельна група | Курси англійської мови | Приватні садки

с. Наварія вул. Стефаника, 1


HobbyHub
Приватні школи
Приватні ліцеї
Ігрова кімната
Приватні гімназії
Інтелектуальні та мовні курси
Танці
Бальні танці
Латиноамериканські танці
Спортивні танці
Сучасні танці
Балет
Народні танці
Бойові мистецтва
Джиу-джитсу
Карате
Рукопашний бій
Бокс
Кікбоксинг

ca

Рекомендації 3

СадОк Здорової Родини

«СадОк Здорової Родини»— це мережа ліцензованих приватних закладів дошкільної освіти, яка вже три роки працює на ринку Львова та забезпечує комплексний підхід до вихован...

Ясельна група | Дитячий садок неповного дня | Курси англійської мови | Бальні танці | Дитячі психологи | Приватні садки | проспект В'ячеслава Чорновола, 16к

Академія традиційних самурайських мистецтв Айкібудзюцу

Про нас:Як проходить пізнання Айкідо?Групами до 15 учнів (айкідок) під керівництвом досвідчених інструкторів іде навчання в додзьо (спортивних залах) та пізнання суті Айк...

Джиу-джитсу | Карате | вулиця Шухевича, 2

Творчий простір Оксани Журило Jurewel'

Відкрийте для себе світ мистецтва у творчому просторі«Jurewel'», створеномухудожницею та арттерапевткою Оксаною Журило. Тут кожен може знайти внутрішню гармонію, впевнені...

вулиця Коперника, 10

HobbyHub

ca

← Назад

Академія традиційних самурайських мистецтв Айкібудзюцу

Джиу-джитсу | Карате | вулиця Шухевича, 2 | вулиця Чукаріна, 3) | вулиця Хімічна, 7 | проспект Червоної Калини, 70

вулиця Шухевича, 2(Приміщення школи "Будокан")

Контакти

+38 (063) 307-00-04

Години роботи

Пн,Вт,Ср,Чт,Пт	15:00 - 22:00	Без перерви
Сб	13:00 - 2	Без пере

Про організацію

Про нас:Як проходить пізнання Айкідо?Групами до 15 учнів (айкідок) під керівництвом досвідчених інструкторів іде навчання в додзьо (спортивних залах) та пізнання суті Айкідо.Запам'ятовувати пози, рухи та прийоми — це лише зовнішня сторона справи.Пізнанням духу Айкідо — досягається розуміння цього мистецтва та карбування Айкідо у підсвідомості і реагування на зовнішні впливи на рівні безумовних рефлексів.Як навчитися Айкідо?Пізнання Айкідо — це університет. Пройшовши університетський курс та виконавши поточні завдання, через п'ять років Ви зможете досягти таємної мрії більшості новачків – отримати чорний пояс.Але це тільки перший крок на шляху Майстра. Останнього кроку не буває. Самовдосконалення безкінечне як безкінечний Всесвіт.Для багатьох заняття айкідо — це клуб, де зустрічаються з цікавими людьми, спільно планують та проводять дозвілля, допомагають один одному вирішувати життєві проблеми.Наша школа регулярно проводить міжнародні семінари за участю провідних та найбільш досвідчених майстрів з Японії, Великобританії, Канади, Чехії, Польщі та інших країн, атестації, літні та зимові табори.Прийти до нас і розпочати свій шлях Ви можете у будь-який час. Дружня

HobbyHub

ca

Години роботи

Пн,Вт,Ср,Чт,Пт	15:00 - 22:00	Без перерви
Сб	13:00 - 20:00	Без перерви
Нд	Вихідний	Вихідний

вулиця Чукаріна, 3(Приміщення школи-ліцею "Оріяна" (ЗОШ №25))

Контакти

+38 (063) 307-00-04

Години роботи

Пн,Вт,Ср,Чт,Пт	15:00 - 22:00	Без перерви
Сб	13:00 - 20:00	Без перерви
Нд	Вихідний	Вихідний

вулиця Хімічна, 7(ЗОШ №22)

Контакти

+38 (063) 307-00-01

Години роботи

університетський курс та виконавши поточні завдання, через п'ять років Ви зможете досягти таємної мрії більшості новачків – отримати чорний пояс.Але це тільки перший крок на шляху Майстра. Останнього кроку не буває. Самовдосконалення безкінечне як безкінечний Всесвіт.Для багатьох заняття айкідо — це клуб, де зустрічаються з цікавими людьми, спільно планують та проводять дозвілля, допомагають один одному вирішувати життєві проблеми.Наша школа регулярно проводить міжнародні семінари за участю провідних та найбільш досвідчених майстрів з Японії, Великобританії, Канади, Чехії, Польщі та інших країн., атестації, літні та зимові табори.Прийти до нас і розпочати свій шлях Ви можете у будь-який час. Дружня атмосфера допомагає швидко адаптуватися у колективі.Спортивні секції та групи:Такай яма ДодзьоКорю ЙосінканДодзьоКікентай ДодзьоДодзьо при Церкві Різдва Пресвятої БогородиціЗали:Сихів, Центральний зал Академії, Чукаріна, 3, приміщення молодшої школи «Оріяна», окремий вхідКонтактний телефон: 0633070004 ОльгаТренуванняпн, сер, пт18.00-19.30 — молодша група 5-9 р.19.30-21.00 — підліткова групавт, чт18.00-19.15 — молодша група19.30-21.00 — доросла групасб18.00-19.30 — Каторі Шінто рю (робота з японським мечем)Сихів, проспект Червоної Калини, 70 (Вхід зі сторони «Інтерсіті»)вт, чт, сб 20.00-22.00 — доросла групаконтакт.номер 0993132458В цих залах викладання Айкідо ЙошінканЦентр, Шухевича, 2, приміщення школи «Будокан», конт. номер 0633070004тренування Айкідо Йошінканпн, сер, пт16.00-17.00 — дитяча група, 7-10 р.18.00-19.30 — підліткова група, 11 р+вт, чт15.00-16.00 — дві групи паралельно, 5-7 р. та 8-10 р.16.00-17.00 — 9-12 р.17.00-18.30 — підліткисубота:12.00-13.00 — діти 5-7 р.13.00-14.00 — діти 9-12 р.14.00-15.30 — підліткитренування Джю-Джюцу:Хімічна 7, СШ 22, конт. номер 0633070001вт, чт18.30-20.00 — дорослісп, сер, пт16.00-17.00 діти 5-8 р.17.00-18.00 діти 9-13 р.18.00-20.00 підлітки та дорослі



1 2 3

HobbyHub

Пошук

Збережені локації: 12

Творча студія "Stefani Art"

Творча студія «Stefani Art»— це місце, де діти та дорослі можуть розкрити свій творчий потенціал та отримати емоційне розвантаження через різноманітні види арттерапії та ...

Арт-студія "Олівці-Малівці"

Послуги творчого розвитку дітей від 6 до 12 років. Уроки ліплення, аплікації та малювання (графіка, живопис, рисунок)....

Англомовний центр раннього розвитку "Baby Years"

Англомовний центр раннього розвитку «Baby Years»пропонує чудові можливості гармонійного розвитку дитини — інтелектуального та фізичного виховання.Для досягнення мети вихо...

Приватні садки | Ясельна група | Ігрова кімната

HobbyHub

Дослідити

Github ↗

Документація ↗

Слідкуйте

Instagram ↗

Facebook ↗

Не пропустіть нові можливості

Ввести ел. пошту | Підписатись