

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)
на тему:

**«Платформа з розміщення подій та заходів
(розробка бекенду)»**

Виконав студент групи ІПЗ-21-5
КОРНІЙЧУК Віталій Васильович

Керівник роботи:
ЄФРЕМОВ Юрій Миколайович

Рецензент:
ГРОМСЬКИЙ Олександр Олександрович

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна Вакалюк

«26» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **КОРНІЙЧУК Віталій Васильович**

Керівник роботи: **ЄФРЕМОВ Юрій Миколайович**

Тема роботи: **«Платформа з розміщення подій та заходів (розробка бекенду)»**,

затверджена Наказом закладу вищої освіти від **«14» лютого 2025 р., №61/с**

Вихідні дані для роботи: **бекенд платформи для розміщення подій для реклами, управління заходами та бронювання місць.**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Єфремов Ю. М.	27.02.2025	27.02.2025
2	Єфремов Ю. М.	27.02.2025	27.02.2025
3	Єфремов Ю. М.	27.02.2025	27.02.2025

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з бекенду для платформи з розміщення подій та пояснювальної записки. Пояснювальна записка до випускної роботи містить 53 сторінки, 33 ілюстрації, 4 таблиці.

Метою роботи є розробка бекенду для платформи з розміщення подій та заходів, що забезпечить зручне і безпечне використання як організаторами, так і відвідувачами.

Загальна архітектура платформи складається з Python застосунку, що написаний за допомогою Django REST фреймворку, PostgreSQL база даних, яка забезпечить зберігання нетипових даних (вектори, геодані), Docker, що дозволяє отримати однакові результати на різних етапах розробки.

Бекенд комунікує з фронтендом завдяки правилам визначеними як REST API — це спростить співпрацю з розробником клієнтської сторони.

КЛЮЧОВІ СЛОВА: БЕКЕНД, ПЛАТФОРМА, ПОДІЇ, DJANGO, PYTHON, REST API, POSTGRESQL, DOCKER.

					ІПЗ.КР.Б — 121 — 13 — ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Платформа з розміщення подій та заходів (розробка бекенду)	Літ.	Арк.	Архівів
Розроб.		Корнійчук В.В.						
Керівник		Єфремов Ю.М.					3	54
Рецензен		Громський О.О.				Житомирська політехніка, група ІПЗ-21-5		
Зав. каф.								

ABSTRACT

The bachelor's qualification thesis consists of a backend of event-advertising platform and an explanatory note. The explanatory note to the thesis contains 53 pages, 33 illustrations, 4 tables.

The goal of the work is to develop a backend of event-advertising that will ensure convenient and secure use by both organizers and visitors.

The overall architecture of the platform consists of a Python application written using the Django REST framework, a PostgreSQL database that will store non-standard data (vectors, geodata), and Docker, which allows you to get the same results at different stages of development.

The backend communicates with the frontend using rules defined as REST API — this will simplify cooperation with the client-side developer.

KEYWORDS: BACKEND, PLATFORM, EVENTS, DJANGO, PYTHON, REST API, POSTGRESQL, DOCKER.

		Корнійчук В.В.			ІІЗ.КР.Б — 121 — 13 — ІІЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

РЕФЕРАТ.....	3
ABSTRACT.....	4
ЗМІСТ.....	5
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ.....	6
ВСТУП.....	7
1. АНАЛІЗ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ РОЗМІЩЕННЯ ПОДІЙ ТА ЗАХОДІВ.....	8
1.1 Постановка задачі.....	8
1.2 Аналіз аналогів програмного продукту.....	9
1.3 Вибір архітектури.....	16
1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення.....	17
1.5 Розподілення обов’язків у команді.....	19
Висновки до розділу 1.....	21
2. РОЗРОБКА МОДЕЛІ ПРОГРАМНОГО КОМПЛЕКСУ НА ЛОГІЧНОМУ РІВНІ.....	23
2.1 Алгоритм роботи та стани програмної системи.....	23
2.2 Об’єктно-орієнтована модель системи.....	27
2.3 Інтеграційне тестування.....	29
2.4 Файлова структура проєкту.....	30
2.5 Комунікації і послідовність взаємодії об’єктів системи.....	32
Висновки до розділу 2.....	34
3. ПОРЯДОК РОБОТИ З БЕКЕНДОМ ДЛЯ ПЛАТФОРМИ З РОЗМІЩЕННЯ ПОДІЙ ТА ЗАХОДІВ.....	35
3.1 Середовище для тестування.....	35
3.2 Документація.....	38
3.3 Сценарії роботи з бекендом.....	40
Висновки до розділу 3.....	46
ВИСНОВКИ.....	48
СПИСКИ ВИКОРИСТАНИХ ДЖЕРЕЛ.....	50
ДОДАТКИ.....	53

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

БД – база даних

ПЗ – програмне забезпечення

REST – передача репрезентативного стану (Representational State Transfer)

API — прикладний програмний інтерфейс (Application Programming Interface)

JWT – JSON веб-токен

JSON – нотація об'єктів JavaScript (JavaScript Object Notation)

DRF — Django REST фреймворк (Django REST Framework)

VG — функціонал спрямований на покращення досвіду користування продуктом (Value Generator)

HTTP — протокол передачі гіпертекстом (Hypertext Transfer Protocol)

GIN — індекс стовпця у БД, що допомагає пошуку всередині рядків (Generalized Inverted Index)

MVT — модель вигляд шаблон (Model View Template)

UI — інтерфейс користувача (User Interface)

UX — досвід користувача (User Experience)

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		6

ВСТУП

Актуальність теми «Платформа з розміщення подій та заходів» зумовлена зростаючою конкуренцією серед бізнесів та очікуваннями споживачів про продукт. Сьогодні організатори подій стикаються з необхідністю не лише реклами, але й спрощенню процесів, такі як бронювання, що підвищить кількість клієнтів. Відвідувачі мають інтерес у зручних та швидких способах пошуку заходів, які відповідають їхнім інтересам, тому платформи, що забезпечують легкий доступ до інформації, стають надзвичайно важливими.

Мета випускної кваліфікаційної роботи бакалавра – розробка даного бекенду. Це включає в себе наступні завдання:

- проектування архітектури серверної частини
- реалізація функціоналу, що дозволить відвідувачам знаходити цікаві для них події, а організаторам ефективно просувати їхні заходи
- забезпечення безпеки, масштабованості при комунікації через API
- створення легкої інтеграції з клієнтською частиною

Об'єктом дослідження є платформи для реклами і розміщення подій.

Предметом дослідження є розробка бекенду для платформи, де розміщуються реклама та інформація про події та заходи.

Методи дослідження включають аналіз вимог до сучасних бекенд застосунків, проектування ПЗ, його розробка, тестування та оптимізація системи.

Практичним результатом є готовий бекенд, що є зручним та безпечним для використання організаторами та відвідувачами, легкий до використання іншими сервісами (особливо при розробці клієнтської частини) та задовільняє більшість сучасних вимог до ПЗ.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		7

1. АНАЛІЗ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ РОЗМІЩЕННЯ ПОДІЙ ТА ЗАХОДІВ

1.1 Постановка задачі

Платформа з розміщення подій та заходів дозволяє організаторам знаходити цільову аудиторію для своїх заходів, а відвідувачам обирати серед актуальних подій цікаві їм. Проте сучасні рішення, що використовуються в Україні, не задовільняють повний перелік вимог до якісного ПЗ.

Визначимо, що дані сервіси повинні мати:

- інструменти спрямовані для легкого створення, редагування подій та заходів, що включають визначення дати, місця проведення, фотографії та відео з події тощо.
- фільтрацію заходів за їх знаходженням, ціною, описом та іншими рисами, що дозволять відокремити актуальні і цікаві проєкти
- автоматизацію процесів, що є монотонними та довгими для людей і з якими програмний код впорається краще
- чіткий поділ подій за категоріями: вільний вхід, регулярність, онлайн тощо, що забезпечить швидке «сканування» інформації відвідувачами
- інтеграцію VG [1] рішень, що полегшують використання, але не є обов'язковими, такі як можливість впроваджувати розклад для записів на заходи, додавати інформацію про місце проведення

Розпишемо етапи при розробці бекенду для даної платформи:

1. Проєктування:

- визначення структури моделей бази даних
- опис вхідної та вихідної інформації для кінцевих точок API
- планування функціоналу, що є відкритим для користування через API та закритим

2. Розробка:

- створення моделей бази даних завдяки Django [2] ORM

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		8

- реалізація серіалізаторів для перетворення даних між моделями та JSON
- побудова кінцевих точок за допомогою DRF [3]
- налаштування автентифікації з використанням JWT
- впровадження обмеження доступів до різних моделей
- створення системи фільтрації, пагінації, пошуку
- розробка документації для використання іншими розробниками

3. Тестування та оптимізація:

- написання інтеграційних тестів із використанням PyTest [4]
- перевірка коректності роботи окремих кінцевих точок
- тестування безпеки, обробки помилок, виняткових ситуацій (edge cases)
- оптимізація на рівні БД (індексування, prefetch), застосунку та API (кешування)

4. Розгортання:

- використання зображень застосунку, БД, проксі тощо
- налаштування docker compose

1.2 Аналіз аналогів програмного продукту

У сучасному цифровому середовищі ринок платформ для просування подій активно розвивається. Поряд зі зростанням інтересу на живі події та заходи дедалі більше зростає попит на автоматизацію широкого набору функцій: бронювання квитків, перегляд подій за інтересами, перегляд статистики, широкому налаштуванню місця та часу подій, інтеграцію з соціальними мережами тощо.

Незалежно від наявності спеціалізованих глобальних рішень для розміщення подій (Eventbrite, Meetup, Ticketmaster) значна частина сучасних малих організаторів віддають перевагу загальнодоступним каналам комунікації — соціальні мережі (Instagram, Facebook, TikTok) або месенджери (Telegram, WhatsApp).

Проаналізуємо Instagram (рис. 1.1) як платформу для розміщення подій.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		9

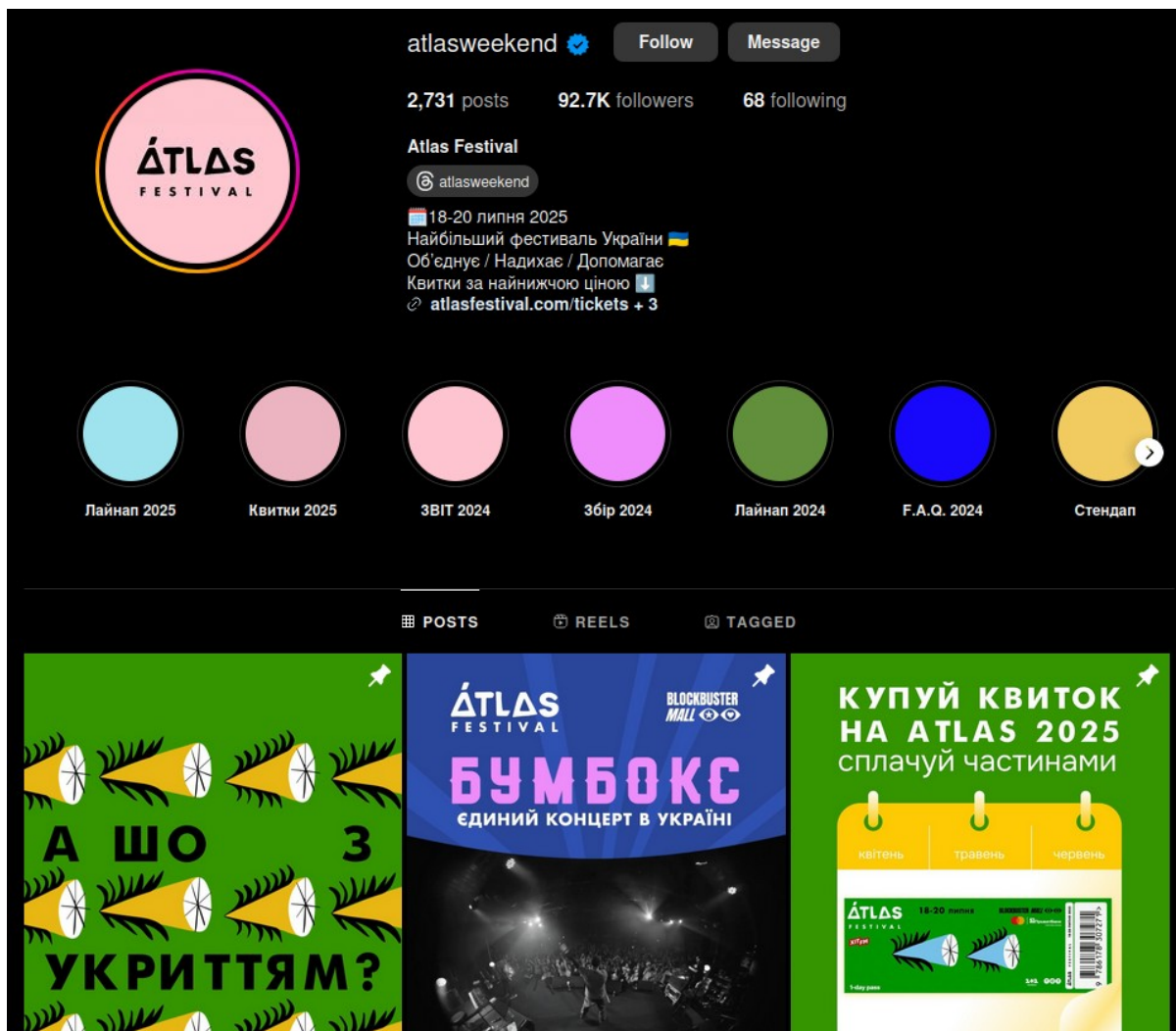


Рис. 1.1 — Instagram сторінка музичного фестивалю

Instagram [5] – це одна з найбільших соціальних мереж створена як платформа для розміщення фотографій. Організатори часто використовують її задля просування подій, сподіваючись на алгоритми рекомендацій, або задля створення портфоліо для фестивалів.

Переваги Instagram як платформи для розміщення подій:

- велика активна аудиторія різного віку і зацікавлень
- можливість органічного просування
- взаємодія з відвідувачами завдяки коментарям, опитуванням, прямим етерам
- функціонал для геоміток для позначень місця подій

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		10

- інтеграція з іншими соціальними мережами чи сайтами для продажу через посилання в описі профіля
- використання хештегів для фільтрації подій за типами

Недоліки:

- висока конкуренція серед акаунтів, що не пов'язані з подіями
- складність у структурованій подачі інформації
- відсутність вбудованого функціоналу для бронювання місць
- необхідність постійного оновлення контенту для підтримання інтересу аудиторія

Наступним рішенням є месенджер Telegram [6] (рис. 1.2). Ним користуються організатори вже з наявною аудиторією або разом з іншими рішеннями, де можливо розрекламувати захід.

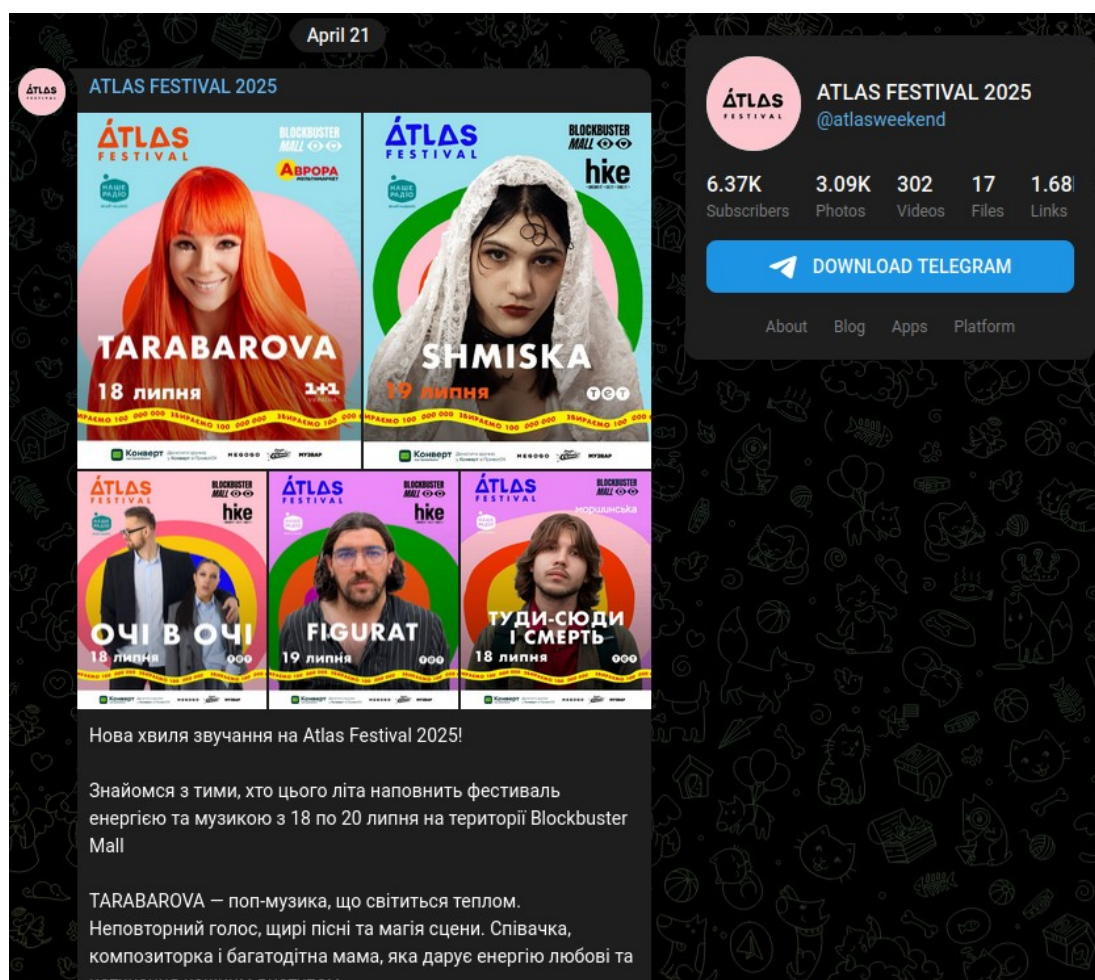


Рис. 1.2 — Telegram канал Atlas Weekend

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		11

Переваги Telegram як платформи для розміщення подій:

- підтримка миттєвих сповіщень для інформування щодо змін
- можливість створення ботів для автоматизації функціоналу бронювання місць
- організація тематичних чатів для спілкування та взаємодії з аудиторією

Недоліки:

- відсутність алгоритмів, що збільшують охоплення аудиторії
- складність створення ботів для недосвідчених людей
- відсутність інструментів аналітики

Попри наявність глобальних платформ для розміщення подій в Україні значно частіше використовуються регіональні рішення. Це зумовлено тим, що глобальні сервіси здебільшого орієнтовані на стандартизовані сценарії проведення великих заходів і не враховують гнучкість, яка потрібна для кожної окремої країни.

Також важливою причиною є відсутність повної та актуальної інформації про українські локації на глобальних сервісах. Іноземні платформи часто не мають широкої бази даних місць проведення подій в українських містах, а отже, організаторам доводиться вводити все вручну або оминати важливі поля, що знижує зручність користування. Це створює додатковий бар'єр і змушує користувачів шукати, які краще відображають локальну специфіку інфраструктури та реалії ринку.

Тому популярні події, такі як концерти відомих виконавців, виступи коміків або великі фестивалі, зазвичай розміщуються на спеціалізованих регіональних платформах. Ці рішення створені для того, щоб забезпечити широкий функціонал, який допомагає організаторам у просуванні заходів. Вони також сприяють залученню нових відвідувачів, розширюючи аудиторію події. Завдяки цьому великі події мають більше можливостей для успішної організації і популяризації.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		12

Одним з таких рішень є Concert [7] (рис. 1.3) — вебсайт, де розміщують концерти, тури, стендап виступи найбільші організатори країни.

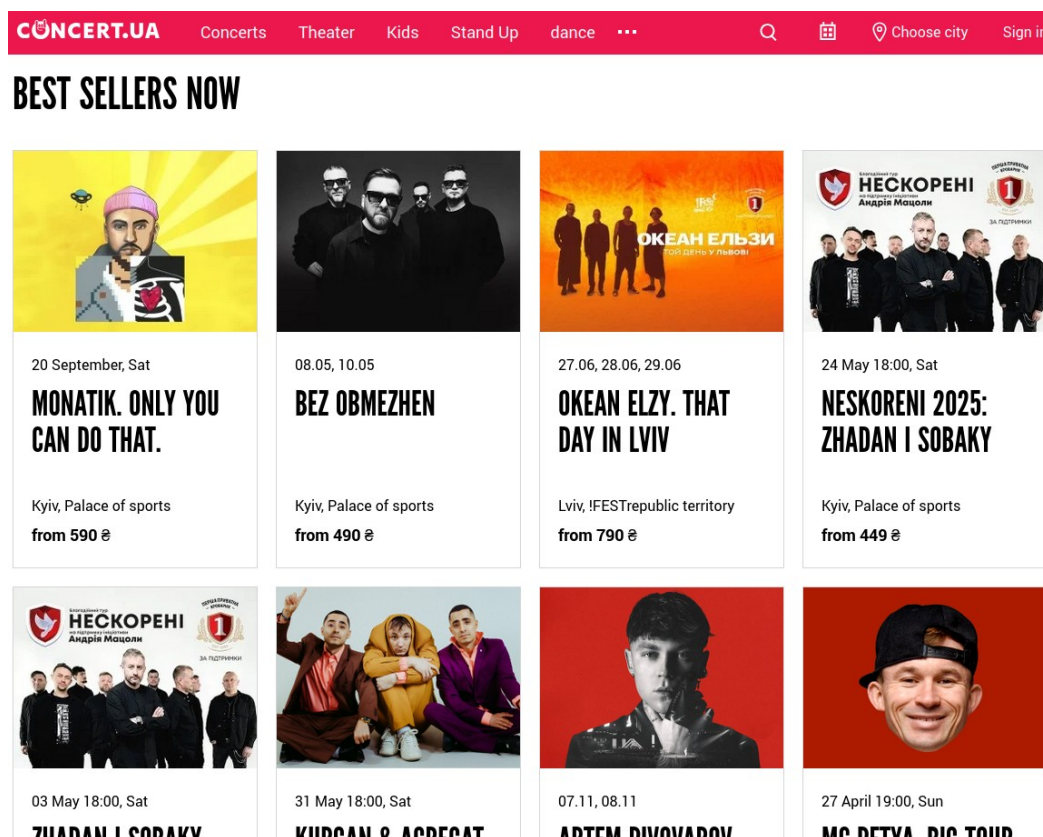


Рис. 1.3 — Головна сторінка Concert

Переваги Concert:

- велика аудиторія користувачів зацікавлена у відвідуванні подій
- зручна система пошуку і фільтрації заходів за містами, жанрами, датами
- легке додавання подій та автоматична обробка замовлень
- наявність системи електронних квитків
- інтегровані рекламні можливості для просування заходів в межах платформи

Недоліки:

- відсутність функціоналу для створення регулярних заходів
- орієнтація на продаж квитків, що не підходить для безкоштовних або закритих подій
- необхідність укладення договору для розміщення подій
- комісії, додаткові витрати при продажі квитків

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		13

- відсутність взаємодії з відвідувачами

Іншим рішенням для організаторів великих подій є Karabas [8] (рис. 1.4) — одна з найдосвідченіших квиткових платформ в Україні, яка має високий попит серед театральних подій.

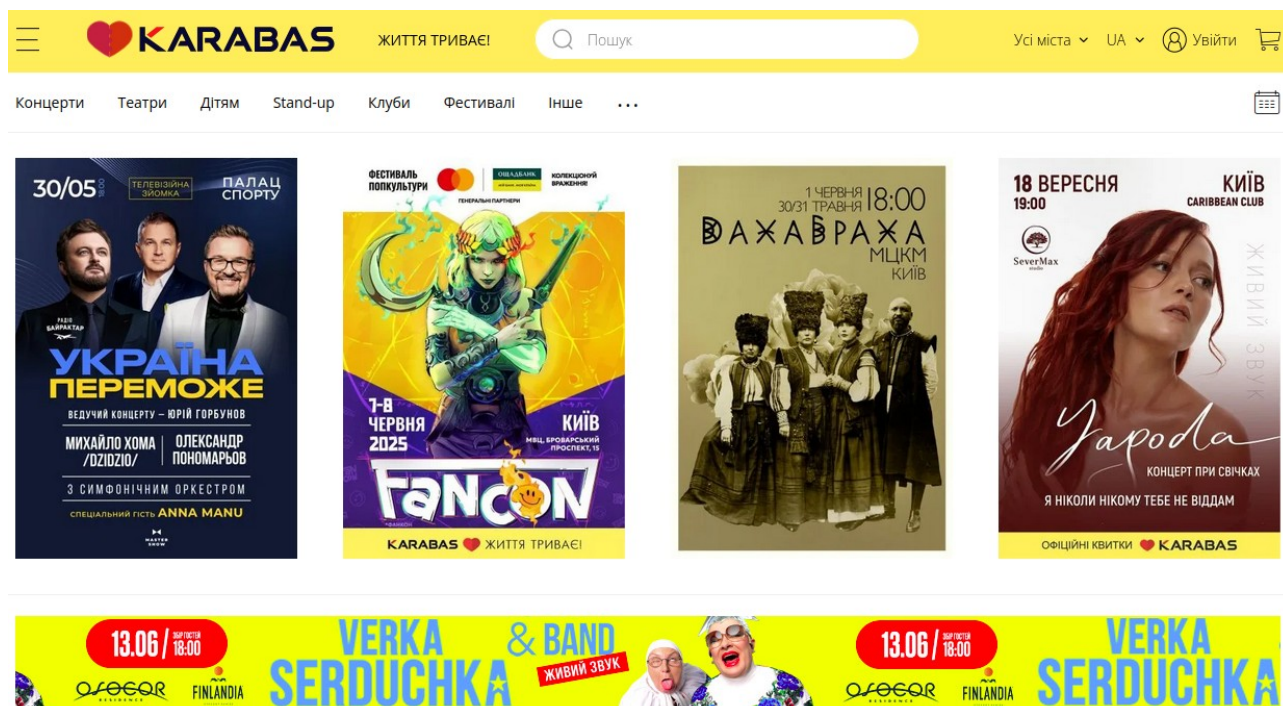


Рис. 1.4 — Головна сторінка Karabas

Переваги Karabas:

- відомий бренд із багаторічною репутацією серед відвідувачів і організаторів
- автоматизація продажу квитків та бронюванню місць
- можливість багатоетапних продажів (попередні продажі тощо)
- підтримка продажу через фізичні каси
- маркетингові інструменти для просування подій на платформі

Недоліки:

- фокус на продаж квитків, що унеможливорює мати безкоштовні події
- неможливість створення повторюваних подій
- висока вартість послуг для організаторів через комісії
- складний процес співпраці для малих чи нових організаторів

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		14

Для наглядного порівняння складемо таблицю з вищезгаданими рішеннями (таблиця 1.1).

Таблиця 1.1

Порівняння платформ з розміщення подій та заходів

Характеристика	Instagram	Telegram	Concert	Karabas	Наше рішення
Доступність для організаторів різних рівнів	+	+	-	-	+
Інтеграція з соціальними мережами та сайтами	+	+	-	-	+
Створення регулярних заходів	+-	+-	-	-	+
Підтримка безкоштовних подій	+	+	-	-	+
Охоплення серед можливих відвідувачів	+-	-	+	+	+
Інструменти для аналітики	+-	-	+	+	+
Фільтрація	+-	-	+	+	+
Встановлення геоміток	+	+	+	+	+

Проаналізувавши популярні сервіси з розміщення подій, можна відзначити, що кожен має власні переваги та обмеження. Instagram та Telegram забезпечують доступність для нових чи малих організаторів з підтримкою безкоштовних подій, але пропонують обмежені можливості для аналітики, налаштуванню, фільтрації. З іншого боку Concert та Karabas орієнтовані лиш на масштабні заходи, тому не є зручними для невеликих або безкоштовних ініціатив.

Нова платформа має враховувати найкращі практики існуючих рішень:

- доступність для широкого кола організаторів
- гнучку інтеграцію з соціальними мережами та сторонніми сайтами
- можливість легко створювати регулярні та одноразові події
- підтримка безкоштовних подій
- система фільтрації для зручного пошуку заходів
- впровадження базової аналітики
- встановлення геолокацій

На відміну від існуючих рішень, розроблювана платформа буде вирізнятись такими особливостями:

- відкритістю для подій будь-якого формату — від невеликих локальних заходів до масштабних фестивалів
- гнучким механізмом управління локаціями, розкладами та схемами розміщення, що дозволяє організаторам максимально точно адаптувати подію до своїх потреб
- технічною архітектурою, що дозволяє легко масштабувати рішення та інтегрувати нові функції без переривання основного функціоналу

Завдяки цьому платформа буде не лише відповідати сучасним вимогам, а й створювати умови для гнучкої, зручної та доступної організації подій різного масштабу.

1.3 Вибір архітектури

Для створення якісного бекенду важливо з самого початку закласти гнучку, масштабовану та надійну архітектуру, яка зможе витримувати зростання навантаження, адаптуватися до змін бізнес-логіки та забезпечити легку інтеграцію з зовнішніми сервісами. Такий підхід дозволяє не лише забезпечити стабільну роботу системи, а й значно спростити її підтримку та розвиток у майбутньому.

Основою нашої архітектури стане багаторівнева модель [9], яка забезпечує чітке розділення обов'язків і підвищує модульність системи. Вона включатиме в себе наступні рівні:

- бізнес-логіку, яка реалізує основні правила роботи з платформою, зберігаючи її незалежною від технічних деталей реалізації
- рівень API, який відповідає за обробку вхідних HTTP-запитів, маршрутизацію, авторизацію та валідацію даних, а також за передачу керування далі в бізнес-рівень

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		16

- доступ до бази даних через ORM, що забезпечує абстрактну роботу з моделями та дозволяє змінювати структуру даних без значних змін в інших частинах системи
- механізм кешування, який дозволить значно зменшити навантаження на базу даних, прискорити обробку повторюваних запитів і покращити загальний досвід користувача

Дана архітектура також побудована на основі принципів чистої архітектури [10] (Clean Architecture). Це означає, що всі залежності спрямовані в бік бізнес-логіки, а не навпаки, що забезпечує низьку зв'язаність між модулями та високу незалежність шарів. Такий підхід дозволяє легко замінювати або оновлювати окремі компоненти (наприклад, змінювати тип бази даних або зовнішній API) без необхідності глобального переписування системи. Також завдяки модульності, команді легше працювати паралельно над різними частинами проєкту, не створюючи конфліктів. У перспективі це дає змогу швидше впроваджувати нові можливості та підтримувати стабільну роботу продукту при зростанні навантаження.

У результаті ми отримуємо стабільну, підтримувану та легко масштабовану архітектуру, готову до розширення функціоналу й інтеграції з новими технологіями.

1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Для розробки бекенду одним з найважливіших аспектів при виборі фреймворку є спільнота. Велика активна спільнота забезпечує низький рівень помилок та проблем з безпекою. Тому було обрано Django та Django REST Framework — стек для швидкої побудови масштабованих та безпечних серверних рішень.

Також серед їх переваг є:

- вбудована ORM для легкої роботи з БД

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		17

- зручне визначення кінцевих точок
- чітка структура проєктів
- велика екосистема пакетів для розширення можливостей (такі як JWT автентифікація)

PostgreSQL [11] та доповнення PostGIS [12] було обрано як БД, що забезпечить надійне збереження реляційних даних разом з просторовими типами (географічними).

Серед переваг також є:

- офіційна інтеграція з Django проєктами
- надійність та перевірена стабільність на великих проєктах
- масштабованість для майбутнього зростання
- великий обсяг доповнень, що легко розширить можливості БД

Redis [13] використовуватиметься як система кешування, що значно знизить навантаження на БД та пришвидшить обробку запитів, результати яких уже зберігаються в тимчасовій пам'яті.

Docker [14] обрано як сервіс контейнеризації, що забезпечить стабільний запуск у різних середовищах.

Для інтеграційного тестування було обрано PyTest — бібліотеку з великим функціоналом для написання швидких (завдяки паралельному запуску) тестів до Python [15] застосунків.

Вимоги до апаратного забезпечення для розробки:

- процесор: Intel Core i5 або AMD Ryzen 5 (або еквівалент) не нижче 7-го покоління;
- оперативна пам'ять: мінімум 8 ГБ, рекомендовано 16 ГБ;
- дисковий простір: мінімум 20 ГБ вільного місця на SSD;
- операційна система: Windows 10/11, macOS 11+, або Linux (Ubuntu 20.04+);
- інтернет-з'єднання: стабільне підключення.

Вимоги до апаратного забезпечення серверу сильно залежать від кількості відвідувачів, але мінімальними є:

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		18

- оперативна пам'ять: мінімум 2 ГБ
- процесор: мінімум 1 vCPU
- дисковий простір: мінімум 10 ГБ
- мережа: стабільне підключення з мінімальною затримкою

Цей стек технологій дозволить створити ефективний, масштабований та безпечний серверний застосунок.

1.5 Розподілення обов'язків у команді

Розробка даного продукту є комплексною задачею, тому було прийнято рішення працювати в команді. Для успішної реалізації проєкту необхідно чітко визначити ролі та обов'язки кожного учасника команди. Ефективне розподілення завдань дозволяє оптимізувати робочий процес, уникнути дублювання зусиль і забезпечити своєчасне виконання всіх етапів розробки. Такі схеми роботи в команді мають назву підходи до керування проєктом [16] (project management approach). Це розвинена галузь на IT-ринку, що надає правильні умови для комфортної та ефективної роботи в команді.

Розглянемо найпопулярніші підходи, що широко використовуються в сучасних реаліях та мають практичну базу доказів про їх ефективність.

До прикладу найчастіше застосовані підходи:

- Водоспад (waterfall) — підхід, що вимагає виконання попередніх етапів до початку інших.
- Гнучкий (agile) — методологія, що базується на колаборації та відгуках замовника.
- Канбан — візуальна структура, що робить акцент на безперервних релізах, що не мають фіксованого порядку.
- Принц (Projects IN Controlled Environments) — робочий процес, що спирається на чітку ієрархію, підзвітність та контроль.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		19

Створимо таблицю для порівняння серед різних характеристик (таблиця 1.2).

Таблиця 1.2

Порівняння підходів до управління проектами

Характеристика	Водоспад	Гнучкий	Канбан	Принц
Базується на	Послідовності	Ітераціях	Потоці	Процесах
Гнучкість до змін	Дуже висока	Висока	Середня	Низька
Визначення ролей	Не вимагається	Не вимагається	Не вимагається	Власник продукту, менеджери, команда
Розмір команди	Будь-який	Малий	Будь-який	Великий
Потрібні інструменти	Будь-який трекер завдань	Будь-який трекер завдань	Jira або Trello	Jira або Trello
Легкість застосування	Висока	Середня	Висока	Низька

Проаналізувавши дані, легко відсіяти методології, що не підійдуть нам за обраними критеріями — принц не підходить за розміром команди, канбан вимагає платного програмного забезпечення, а водоспад вимагає послідовності, що може навпаки сповільнювати розробку продукту. Методом виключення ми обрали гнучкий підхід до управління проектом. Фундаментом методології є цикл з 6 стадій: аналіз, дизайн, розробка, тестування, розгортання і рецензія (рис. 1.5).

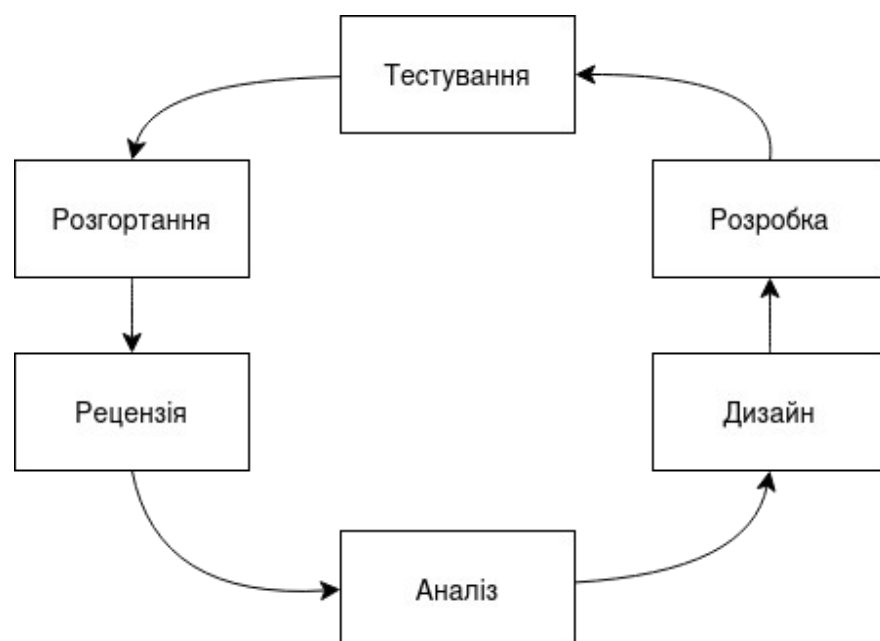


Рис. 1.5 — Цикл гнучкої методології

Наступним кроком є розподілення обов’язків. Незважаючи на те, що обраний підхід не вимагає чітко розділених ролей, проте варто визначити роботу, що буде виконувати кожен учасник команди. Це дозволить нам уникнути повторного виконання завдань, полегшити комунікацію та забезпечить своєчасне виконання етапів розробки.

Я, Корнійчук Віталій, маю більше досвіду та знань у сфері розробки серверної частини продукту, у той час як Пархомчук Руслан — клієнтської. Проте як визначив Марті Каган: «Кожна команда повинна мати менеджера продукту, дизайнера та інженерів. Це ядро розробки продукту» [17]. Тому кінцевий розподіл обов’язків є наступним (таблиці 1.3 та 1.4).

Таблиця 1.3

Обов’язки Пархомчука Руслана

Роль	Обов’язки	Сильні сторони
Дизайнер	Дослідження очікувань користувачів, створення UI/UX, прототипування та візуалізація інтерфейсів	Креативність та розуміння користувацького досвіду
Інженер клієнтської частини	Аналіз нового функціоналу, визначення вимог, написання коду та мануальне тестування	Досвід у розробці клієнтської частини, застосування стандартів індустрії та розуміння трендів

Таблиця 1.4

Обов’язки Корнійчука Віталія

Роль	Обов’язки	Сильні сторони
Менеджер продукту	Визначення бачення продукту, формування вимог, пріоритезація завдань та комунікація	Розуміння потреб користувачів та зацікавленість у розвитку в сфері бізнесу
Інженер серверної частини	Аналіз нового функціоналу, визначення вимог, написання коду, написання інтеграційних тестів та документування API	Досвід у розробці серверної частини, застосування стандартів індустрії та розуміння трендів

Висновки до розділу 1

У цьому розділі було визначено основні вимоги до сучасної платформи для розміщення подій, які спрямовані на полегшення процесу створення та просування заходів для організаторів, а також на зручність пошуку й вибору

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		21

подій для відвідувачів. Було запропоновано чіткий план розробки бекенд-частини платформи, що включає етапи проєктування, розробки, тестування та розгортання.

Аналіз існуючих рішень показав, що жодна з цих платформ не задовольняє повністю специфічні потреби організаторів малих і середніх заходів. Соціальні мережі забезпечують доступ до великої аудиторії, але мають обмежений функціонал для управління подіями. Спеціалізовані платформи, натомість, орієнтовані переважно на продаж квитків великих заходів і не пропонують зручностей для безкоштовних чи повторюваних подій.

Таким чином, актуальним є створення нової платформи, яка поєднає переваги існуючих рішень, мінімізуючи їх недоліки, та забезпечить ефективний інструмент для організаторів різного рівня.

За допомогою порівняння наявних методологій було обрано гнучкий підхід до керування проєктами. Завдяки розподілу ролей відповідно до навичок та сильних сторін учасників, ми сформували ефективну структуру, яка охоплює всі ключові аспекти розробки: планування, дизайн, клієнтську та серверну частини.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		22

2. РОЗРОБКА МОДЕЛІ ПРОГРАМНОГО КОМПЛЕКСУ НА ЛОГІЧНОМУ РІВНІ

2.1 Алгоритм роботи та стани програмної системи

Для розуміння взаємодії модулів та плин у процесу на моменті проєктування розробляють діаграми потоку процесу [18] (Flowchart). Це грає велику роль в уникненні помилок під час написання програмного коду.

Виконаємо діаграми для основних процесів, починаючи з повноцінної діаграми, яка описує взаємодію всіх модулів (рис. 2.1).

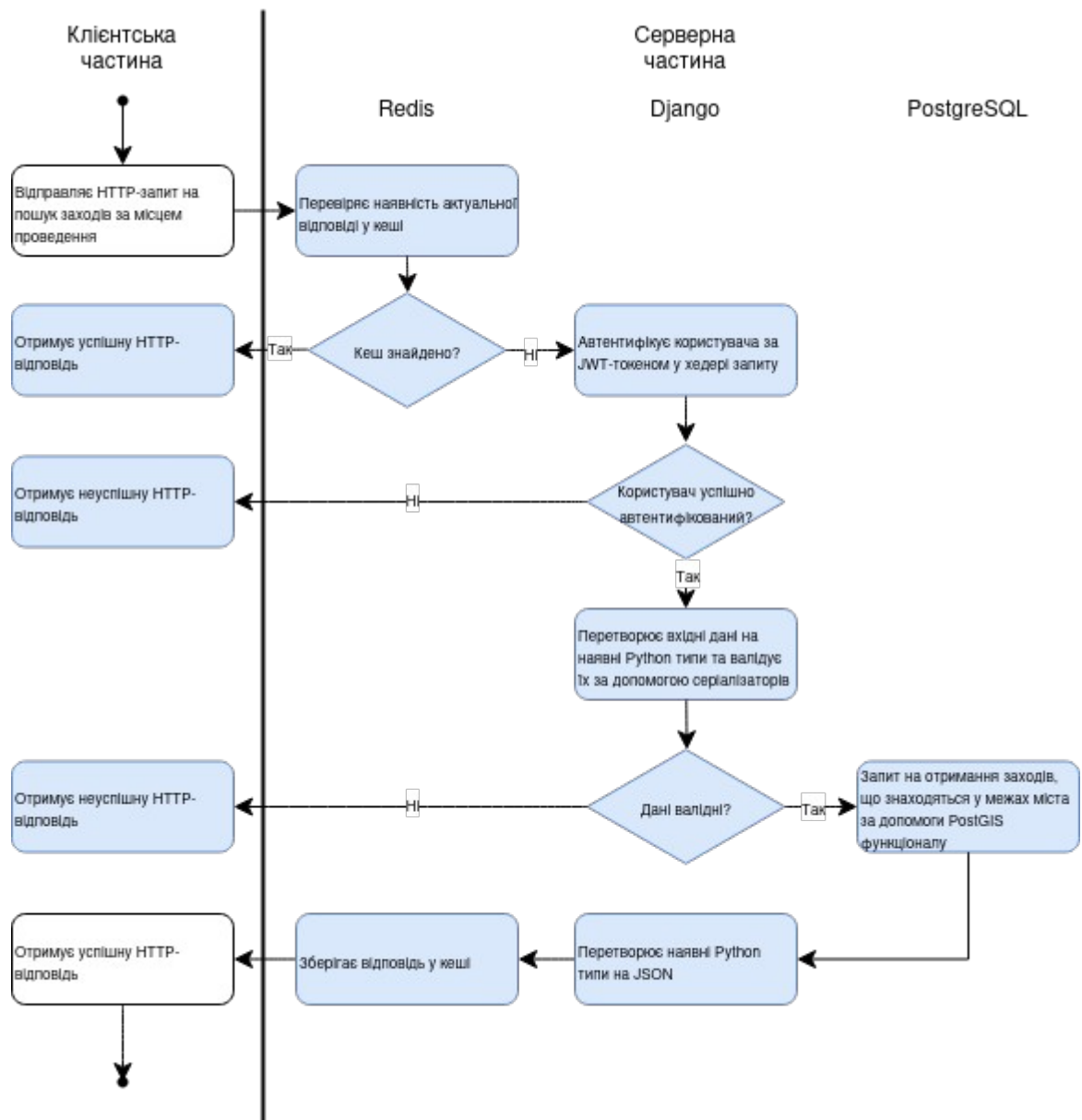


Рис. 2.1 — Діаграма для пошуку заходів за місцем проведення

На цій діаграмі показано як оптимізовано процес завдяки використанню кешування та попередній перевірці достовірності користувача. Якщо у кеші є актуальна відповідь, система швидко повертає результат без зайвого навантаження на базу даних. У разі відсутності кешу або необхідності актуалізації даних, запит обробляється на сервері з урахуванням перевірки прав доступу користувача та просторової фільтрації заходів за допомогою можливостей PostGIS. Усі проміжні дані серіалізуються для забезпечення стабільності та безпеки обробки, а отримані результати зберігаються для подальшого прискорення відповідей.

Для уникнення повторень блоки синього кольору будуть упущені в наступних прикладах, але на етапі розробки продукту вони також будуть реалізовані.

Іншим рівнем оптимізації є пришвидшення роботи БД (рис. 2.2).

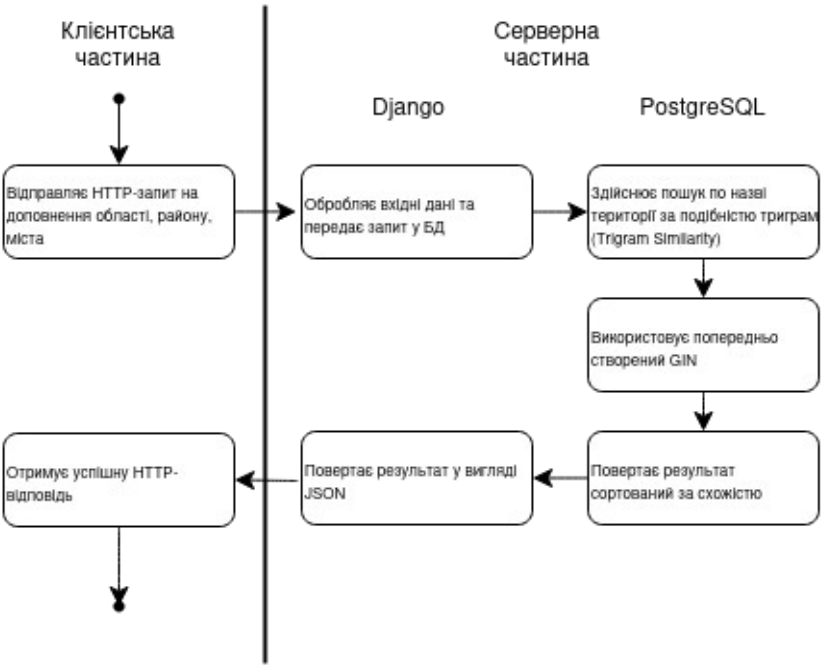


Рис. 2.2 — Діаграма доповнення назви території

На діаграмі показано, що для реалізації швидкого пошуку назв територій за неповним або неточним введенням використовується механізм триграмної схожості [19]. Триграма — це послідовність трьох послідовних символів у

слові. Наприклад, для слова "Житомир" триграми будуть (# позначає початок або кінець слова):

- #Жи
- Жит
- ито
- том
- оми
- мир
- ир#

Коли користувач вводить запит, система розбиває введений текст на триграми й порівнює їх з триграмами збережених назв у базі даних. Визначається ступінь подібності між словами, що дозволяє знайти відповідники навіть при орфографічних помилках, пропущених буквах або іншій неточності введення.

Щоб зробити такий пошук швидким і масштабованим, у базі даних попередньо створюється спеціальний індекс — GIN [20]. Цей тип індексу дуже добре підходить для зберігання й швидкого пошуку множин даних, таких як триграми. Замість повного перегляду всіх записів при кожному запиті, GIN дозволяє миттєво знаходити лише ті записи, де триграми відповідають заданому візерунку (pattern). Таким чином, навіть при великій кількості населених пунктів і територій час пошуку залишається мінімальним.

Використання триграмної схожості разом із GIN-індексацією суттєво покращує користувацький досвід, дозволяючи знаходити потрібні записи швидко, зручно та без вимог до ідеального введення тексту.

Іншим важливим фактором при розробці якісного ПЗ є правильна валідація даних. Для наглядного прикладу підійде діаграма бронювання місця на регулярну подію (рис. 2.3).

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		25

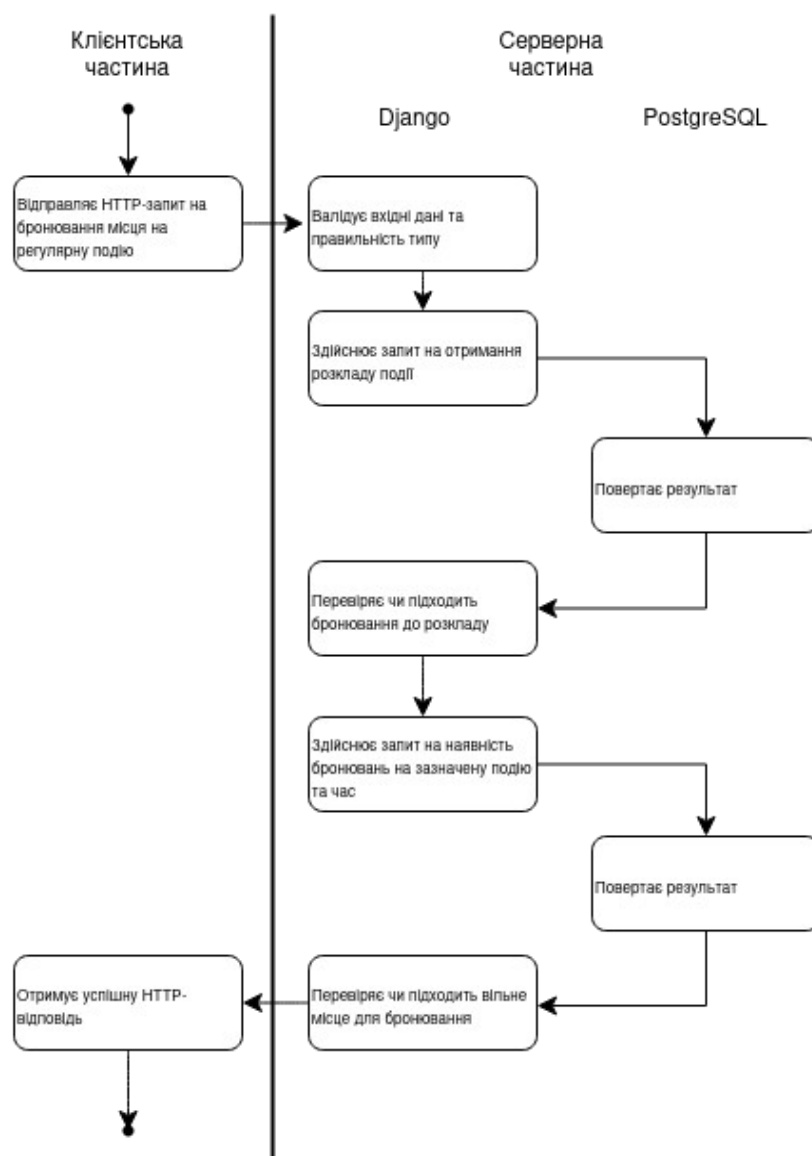


Рис. 2.3 — Діаграма бронювання місця на регулярну подію

На прикладі даної діаграми показано важливість правильної валідації. У процесі бронювання місця на регулярну подію враховуємо актуальність розкладу і перевірку конфліктів із вже здійсненими бронюваннями. Система виконує багаторівневу валідацію, щоб гарантувати коректність даних ще до реєстрації бронювання в базі.

Такий підхід дозволяє запобігти дублюванню місць та уникнути суперечностей у розкладі, забезпечуючи стабільну роботу навіть за умов високого навантаження.

2.2 Об'єктно-орієнтована модель системи

Основними об'єктами у Django є моделі, що відповідають за репрезентацію рядків у БД [21]. Правильне планування допоможе уникнути помилок при створенні відповідних таблиць, тому спроектуємо схему таблиць (рис. 2.4).

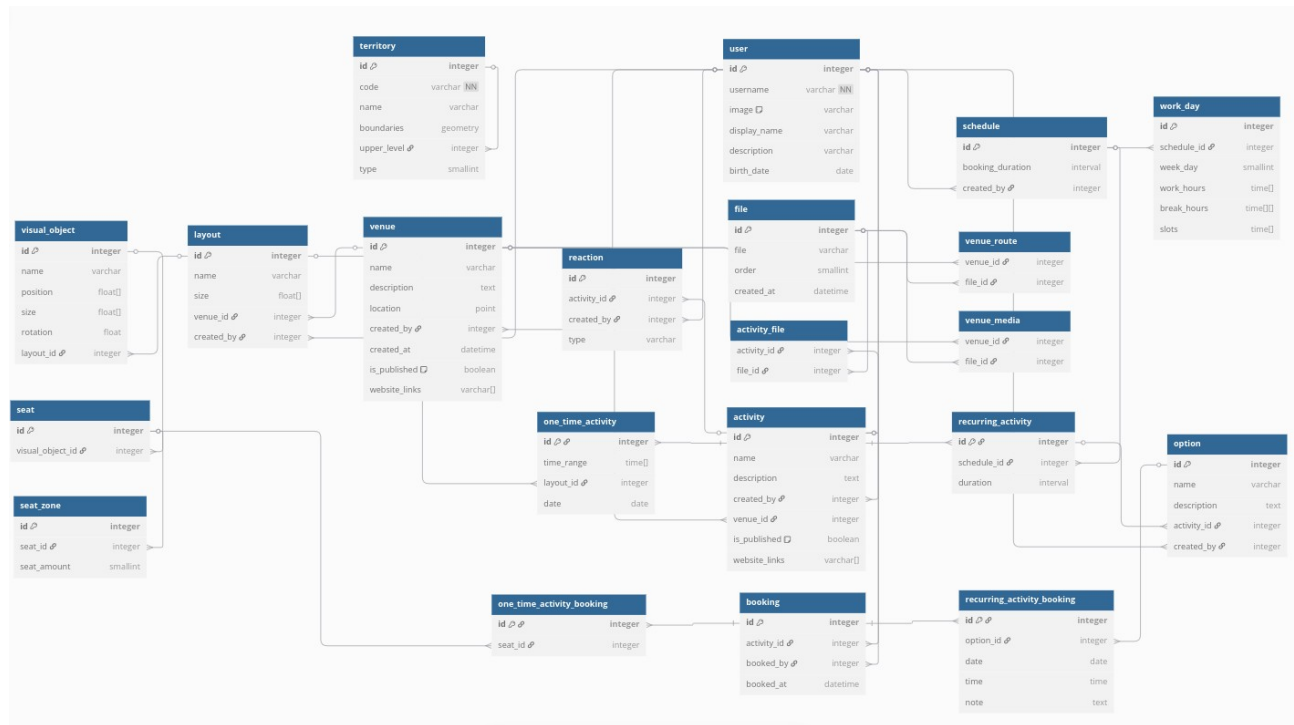


Рис. 2.4 — Схема бази даних

Було створено гнучку структуру бази даних для керування подіями, користувачами та місцями їх проведення, із чітким розділенням між одноразовими та регулярними активностями. Система підтримує приєднання медіафайлів і маршрутів до локацій, що покращує візуальне представлення і навігацію для користувачів. Ієрархічна модель територій із геопросторовими межами забезпечує можливість ефективної фільтрації за адміністративними рівнями. Завдяки зв'язку розкладів з робочими днями та перервами було реалізовано точний контроль доступності бронювання у межах допустимого часу.

Важливими перевагами в об'єктно-орієнтовному програмуванні [22] є наслідуваність та абстракція, що дозволяє нам писати код, що легко зрозуміти

та не вимагає аналізу всіх функцій та класів. Було створено головний клас серіалізатор `MainModelSerializer`, що розширює функціонал базового класу `ModelSerializer` за допомогою полів: `current_user_field`, `choice_display_fields`, `fk_serializers` та `multiple_file_fields`. Порівняємо використання поля `multiple_file_fields` з написанням цього функціоналу вручну.

```
class ExampleSerializer(MainModelSerializer):
    class Meta:
        ...
        multiple_file_fields = {
            "media1": {"validators": [media_content_type_validator]},
            "media2": {"validators": [media_content_type_validator]},
        }
```

Рис. 2.5 — Використання `multiple_file_fields`

Для прикладу візьмемо код з загальнодоступного джерела [23].

```
class ExampleSerializer(serializers.ModelSerializer):
    media1 = serializers.FileField(required=False, write_only=True)
    media2 = serializers.FileField(required=False, write_only=True)

    def create(self, validated_data):
        from django.core.files.uploadedfile import InMemoryUploadedFile

        validated_files = []
        for key, value in validated_data.copy().items():
            if isinstance(value, InMemoryUploadedFile):
                validated_files.append(value)
                validated_data.pop(key)
        instance = super().create(validated_data)
        for file in validated_files:
            File.objects.create(example=instance, file=file)
        return instance
```

Рис. 2.6 — Написання коду вручну

При тому що другий варіант є об'ємнішим він не дозволяє використання валідації, не реалізує завантаження файлів при PUT і PATCH запитах та ускладнює читання коду.

2.3 Інтеграційне тестування

Правильне функціонування серверної частини є дуже важливим, як на етапі розгортання проєкту, так і під час розробки. У випадку, коли бекенд працює некоректно, розробка клієнтської частини зупиняється також. Для уникнення таких ситуацій розробники пишуть тести. У даному проєкті особлива увага приділяється інтеграційному тестуванню [24], оскільки воно дозволяє перевірити взаємодію різних компонентів системи у реальних сценаріях.

На відміну від модульного тестування [25], яке перевіряє окремі функції або класи, інтеграційне охоплює повну логіку запиту. Таким чином, ми можемо переконатися, що серіалізатори, валідація, взаємодія з медіа, розкладами та локаціями працюють коректно в комплексі.

Для написання тестів використовується бібліотека PyTest. Це дозволяє нам створення фікстур [26] (рис. 2.7), які дозволяють зручно працювати з даними: реєструвати користувача, створювати захід, опубліковувати його тощо.

```
@pytest.fixture
def user_data():
    yield {
        "username": f"user-{uuid4().hex[:8]}",
        "password": "StrongP@ssw0rd!",
    }

@pytest.fixture
def signup_user(client, user_data):
    response = client.post("/auth/signup/", json=user_data)
    assert response.status_code == 201
    yield response

@pytest.fixture
def auth_client(client, signup_user):
    client.headers["Authorization"] = f"Bearer {signup_user.json()['access']}"
    yield client
```

Рис. 2.7 — Фікстура з реєстрації користувача

Основний набір тестів стосується перевірки кінцевої точки /activities/, який відповідає за роботою з заходами. Зокрема, перевіряються сценарії створення як регулярних, так і одноразових подій. Крім того, окремі тести охоплюють помилки валідації — наприклад, коли передано некоректний тип активності або файли, що не відповідають очікуваному формату (рис. 2.8).

```
def test_invalid_files(
    self, auth_client, recurring_activity_data, invalid_media_data
):
    response = auth_client.post(
        self.url,
        data=recurring_activity_data,
        files=invalid_media_data,
    )
    assert response.status_code == 400
    data = response.json()
    assert "0" in data["media_files"] and "1" in data["media_files"]
```

Рис. 2.8 — Тестування неправильного типу файлів

Така перевірка дозволяє впевнитися, що система коректно реагує на помилки і повертає очікувані повідомлення.

Інтеграційні тести в проєкті гарантують правильність роботи бекенду в реальних умовах, дозволяють виявити помилки в логіці або валідації, є основою для безпечного рефакторингу коду [27], покривають як успішні сценарії, так і виняткові випадки.

Завдяки інтеграційному тестуванню команда може бути впевненою, що основні функції системи залишаються стабільними навіть при швидкому темпі розробки.

2.4 Файлова структура проєкту

Проєкт побудовано за мікросервісною архітектурою [28] з чітко розділеними відповідальностями. Кожен компонент має власну директорію (рис. 2.9), конфігураційні файли та контекст збирання Docker-образу (рис. 2.10). Завдяки цьому проєкт є зручним для масштабування, тестування та супроводу.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		30

activity-finder

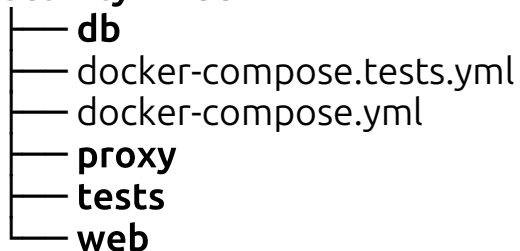


Рис. 2.9 — Структура першого рівня

web:

```
build:
  context: ./web
  dockerfile: Dockerfile
```

db:

```
build:
  context: ./db
  dockerfile: Dockerfile
```

proxy:

```
build:
  context: ./proxy
  dockerfile: Dockerfile
```

tests:

```
build:
  context: ./tests
  dockerfile: Dockerfile
```

Рис. 2.10 — Образи сервісів

Окремі конфігураційні файли для Docker (docker-compose.yml та docker-compose.tests.yml) дозволяють нам зручно запускати проєкт без або з викликом інтеграційних тестів (рис. 2.11).

docker compose -f **docker-compose.yml** -f **docker-compose.tests.yml** up — build

Рис. 2.11 — Команда запуску всіх сервісів разом з тестами

У папці web містяться окремі Django-модулі [29] (apps), кожен з яких відповідає за певну частину функціональності (рис. 2.12). Це спрощує

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		31

підтримку, дає змогу працювати над різними модулями паралельно, і забезпечує легке повторне використання коду. Крім того, таку структуру легко масштабувати, додаючи нові додатки без зміни існуючих.

web

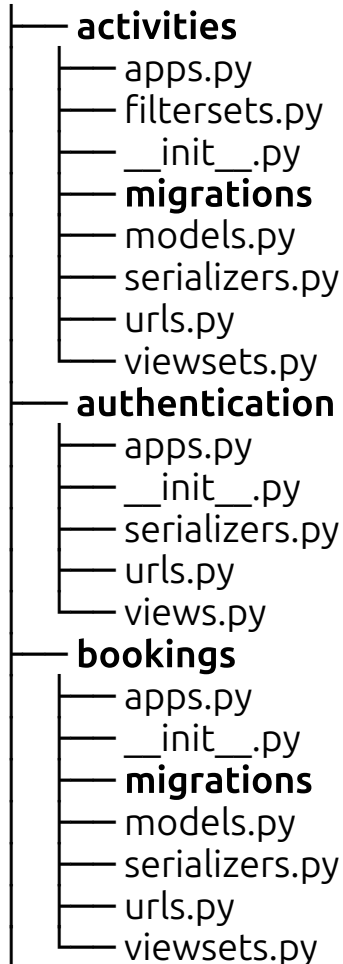


Рис. 2.12 — Структура модулів

Django та Django REST Framework вимагають чітку структуру та взаємодію між об'єктами. У нашому випадку цю систему можна категоризувати як «Модель вигляд шаблон» [30] (MVT). Проте важливо зазначити, що наш фреймворк вимагає іншу назву для шаблону, а саме серіалізатор.

Основні класи:

- модель, що відповідає за визначення та взаємодію даних у БД
- вигляд, що обробляє запити (перевіряє дозволи, виконує дії над даними)
- серіалізатор (шаблон), що конвертує дані з моделей у JSON та навпаки

Для наглядного розуміння, як ці компоненти взаємодіють між собою створимо діаграму послідовності (рис. 2.5).

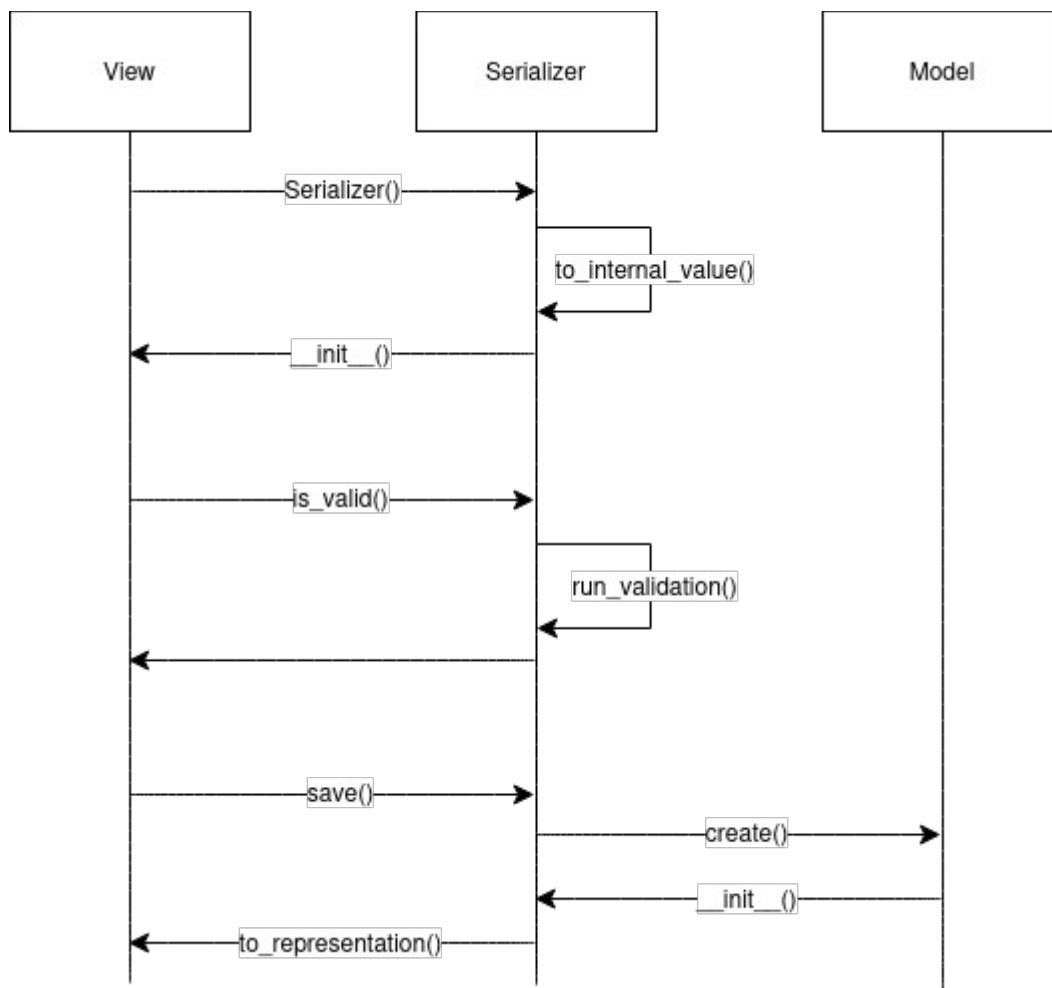


Рис. 2.10 – Діаграма послідовності

Ця діаграма описує внутрішню послідовність викликів під час обробки запиту, що створює об'єкт. Логіка:

- View ініціює екземпляр з переданими даними
- Serializer перетворює вхідні дані у внутрішні Python типи
- View викликає валідацію полів
- Serializer валідує їх (валідація відбувається на рівні всього серіалізатора, окремих полів та кожного поля окремо)
- після успішної валідації View викликає метод для збереження об'єкту
- Serializer викликає створення об'єкту у моделі
- Model повертає новостворений об'єкт

Висновки до розділу 2

У цьому розділі було детально розглянуто алгоритмічну та структурну організацію програмної системи. Було побудовано діаграми, які наочно демонструють ключові процеси взаємодії між модулями, зокрема пошук заходів, автозаповнення назв територій, а також бронювання місць на події. Завдяки використанню кешування, перевірки прав доступу, GIN-індексації та триграмного пошуку вдалося забезпечити високу швидкість та точність реакції системи на запити користувача.

Крім того, акцент зроблено на важливості багаторівневої валідації при створенні об'єктів, що забезпечує надійність та узгодженість даних. ООП-модель бази даних реалізована у вигляді чітко структурованих взаємозв'язків між об'єктами, що дозволяє легко масштабувати систему та адаптувати її під змінні бізнес-вимоги. Діаграма послідовності взаємодії об'єктів показала, як компоненти Django REST Framework співпрацюють між собою у процесі обробки запитів, валідації та збереження даних.

Окрему увагу було приділено інтеграційному тестуванню, яке відіграє ключову роль у підтримці цілісності системи під час активної розробки. Завдяки налаштованому середовищу тестування розробники можуть виявляти помилки на ранньому етапі, підвищуючи якість програмного продукту.

Також було проаналізовано файлову структуру проєкту, яка демонструє модульний підхід до організації коду. Чітке розділення логіки за мікросервісами та ізоляція середовищ за допомогою Docker значно спрощують масштабування, обслуговування та командну роботу. Такий підхід забезпечує гнучкість у розгортанні, можливість паралельної розробки окремих компонентів і підвищує прозорість структури для нових учасників команди.

Загалом, розділ заклав фундамент для реалізації надійної, масштабованої та продуктивної програмної системи на основі сучасних підходів до проєктування та розробки.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		34

3. ПОРЯДОК РОБОТИ З БЕКЕНДОМ ДЛЯ ПЛАТФОРМИ З РОЗМІЩЕННЯ ПОДІЙ ТА ЗАХОДІВ

3.1 Середовище для тестування

Для зручного тестування роботи бекенд-системи використовується середовище Postman [31] — потужний інструмент для формування, надсилання та аналізу HTTP-запитів, що дозволяє створювати структуровані колекції запитів, автоматизувати перевірку відповідей та швидко перевіряти нові зміни в API без потреби створювати окремі клієнтські інтерфейси.

Створимо колекції запитів та згрупуємо за категоріями. Це дозволить нам підтримувати впорядкованість і швидко орієнтуватися у великій кількості кінцевих точок (рис. 3.1).

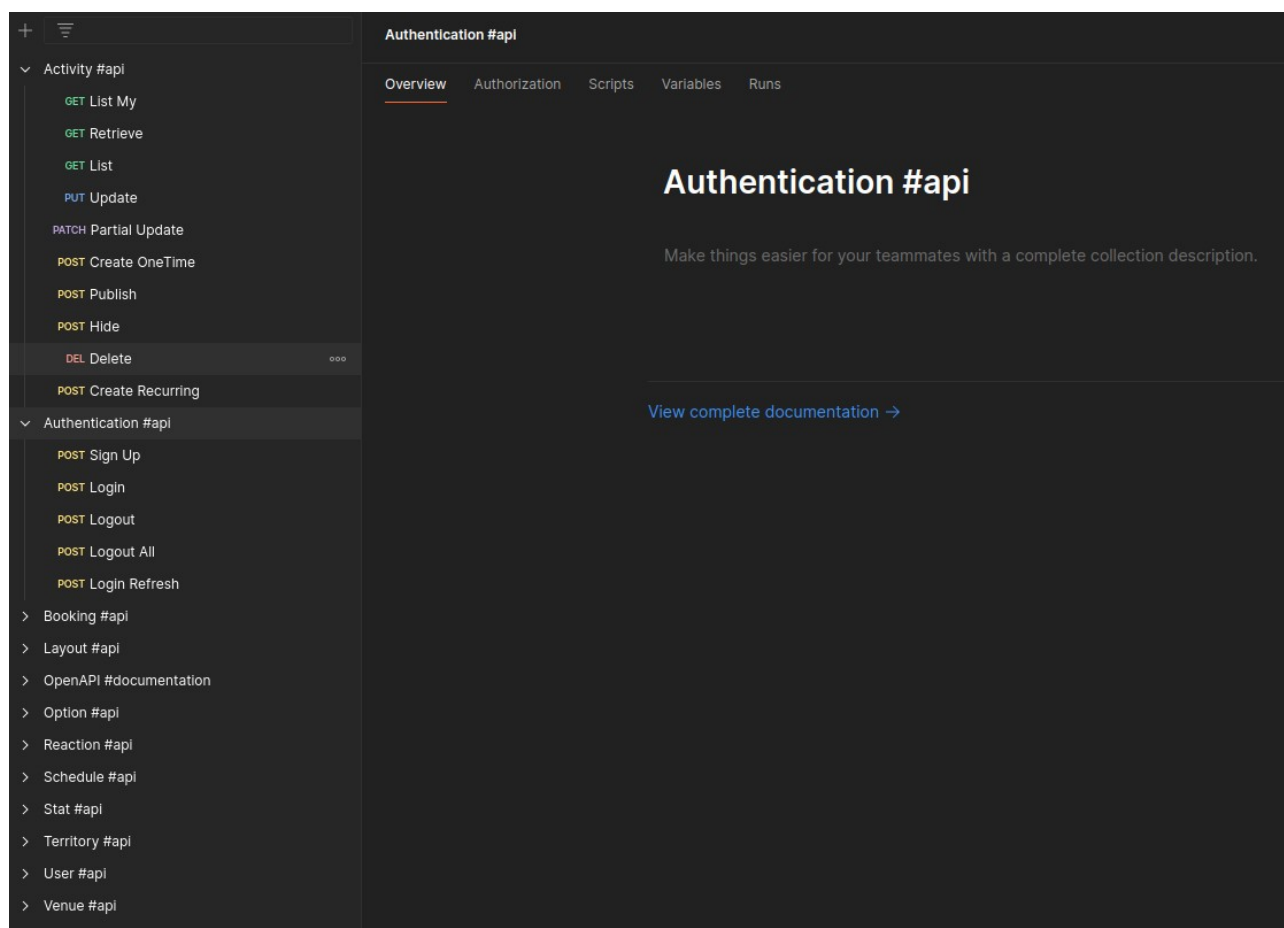
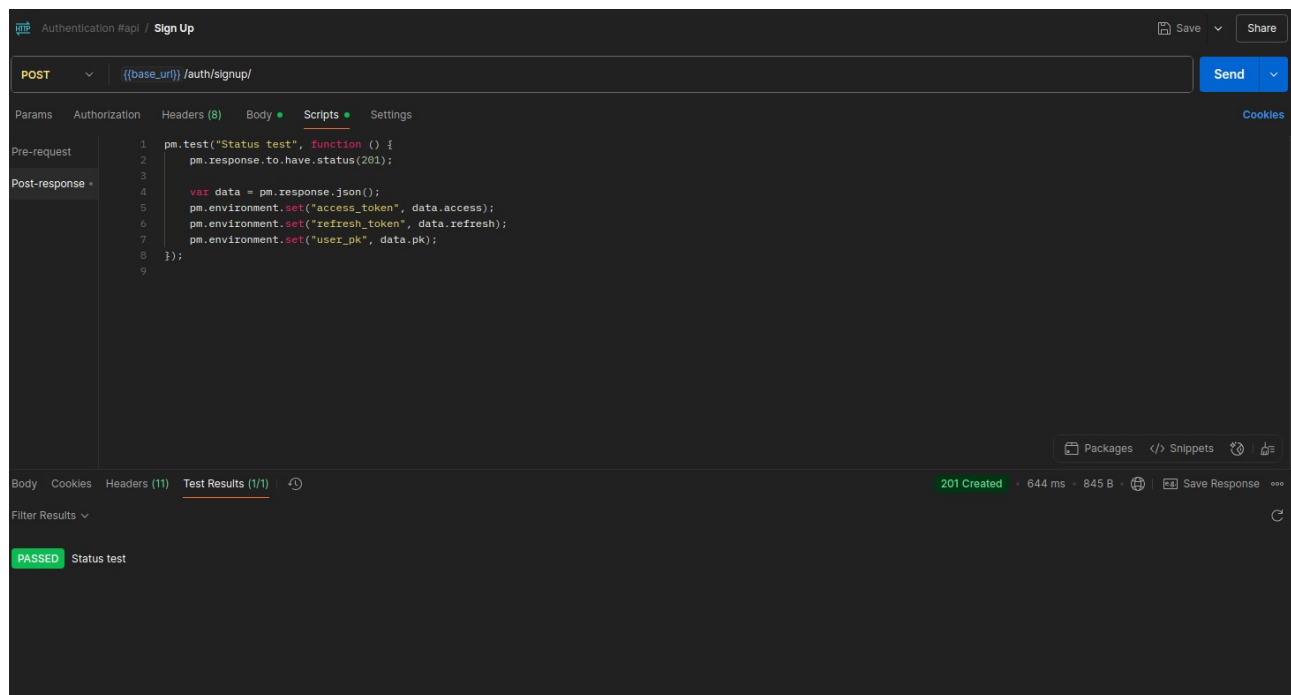


Рис. 3.1 — Запити для тестування

Одним із базових запитів є реєстрація нового користувача. POST-запит на створення облікового запису виконується через кінцеву точку /auth/sign-up/, яка

[illegible]

Для автоматизації та спрощення роботи з бекендом ми можемо використовувати функцію Scripts [32] для перевірки статусу відповіді та збереження змін у наше середовище (рис. 3.3).



		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				36
Змн.	Арк.	№ докум.	Підпис	Дата		

Збережені значення дозволяють нам налаштувати загальний заголовок (header) авторизації для всієї колекції запитів. Таким чином, Postman автоматично підставляє токен у кожен запит без необхідності ручного редагування (рис. 3.4).

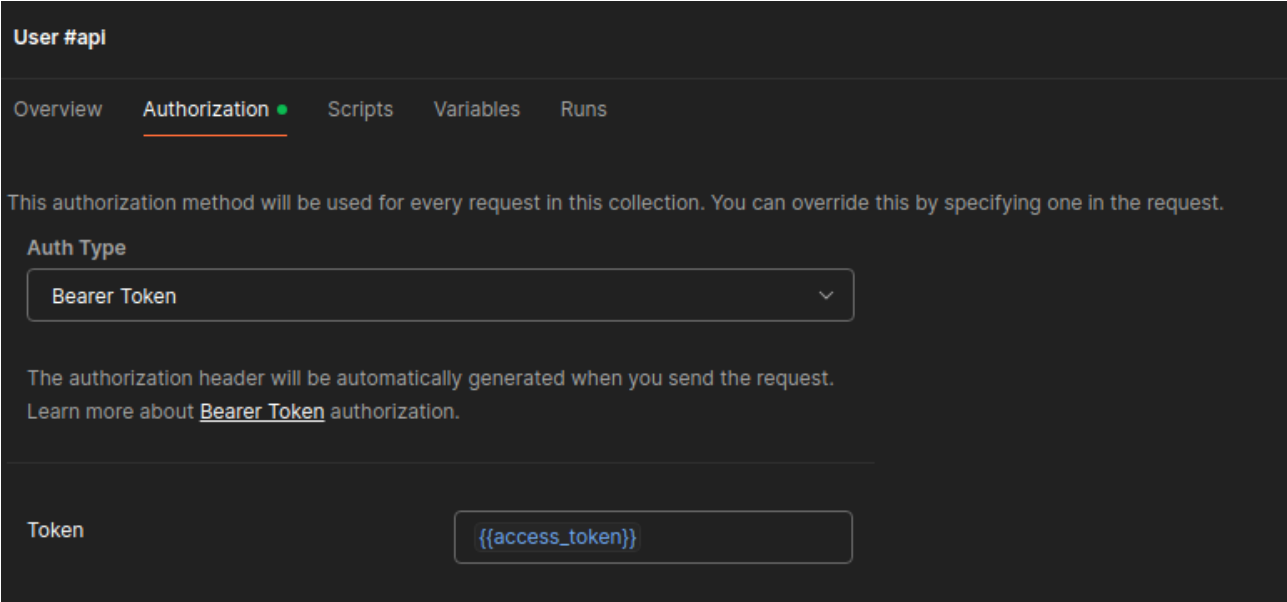


Рис. 3.4 — Автоматичне заповнення поля для авторизації

Запит на отримання даних поточного користувача є одним із прикладів авторизованих запитів. Без додаткових введень даних вручну ми можемо виконати цей запит завдяки попередньо налаштованому середовищу (рис. 3.5).

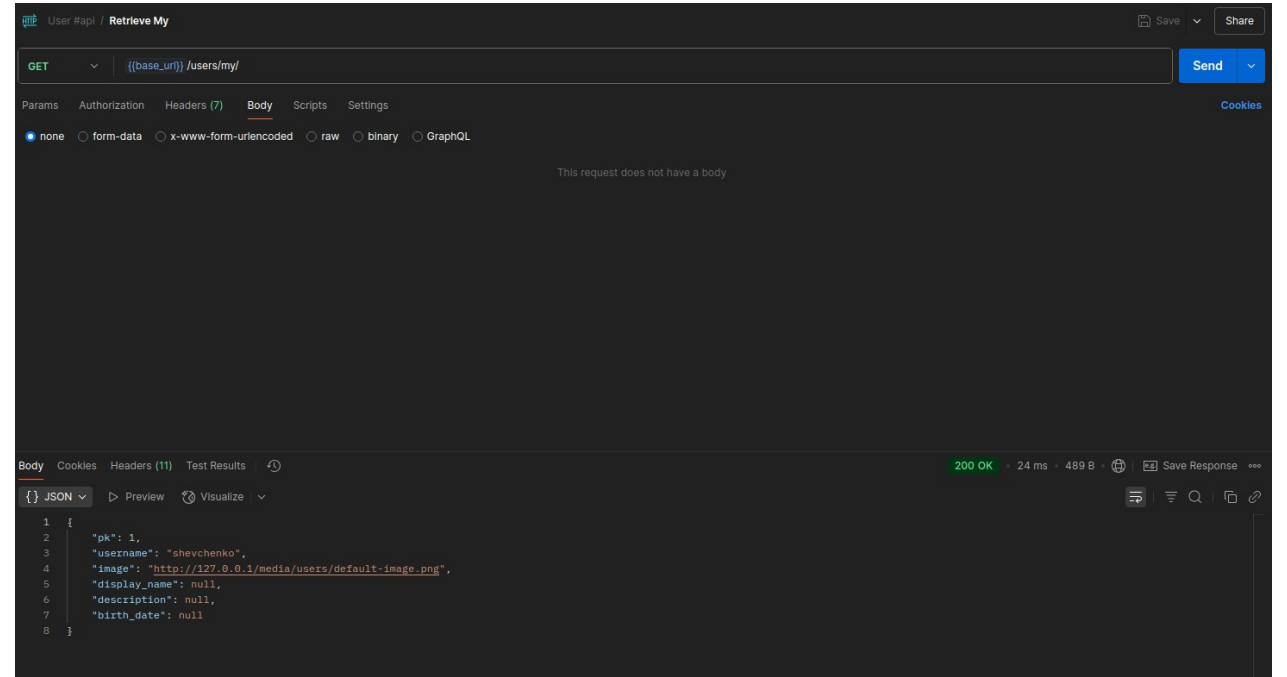


Рис. 3.5 — Запит на отримання поточного користувача

Використання налагодженого середовища Postman значно спрощує процес тестування нашого бекенду. Це дає змогу нам фокусуватись на правильності механізмів, захисту та валідації даних та отримання коректних відповідей.

3.2 Документація

Документація API — це важлива складова сучасної бекенд-розробки, яка значною мірою визначає зручність роботи як для фронтенд команди, так і для майбутніх розробників серверної частини. Для нашої платформи було реалізовано декілька способів доступу до документації: Swagger [33], ReDoc [34], а також у вигляді чистого YAML-файлу [35], що відповідає специфікації OpenAPI. Кожен з варіантів має своє призначення та переваги.

Swagger надає інтерактивну документацію, яка дозволяє переглядати всі доступні кінцеві точки, структуру запитів і відповідей, а також одразу ж протестувати API безпосередньо у веб-інтерфейсі. Це особливо зручно на етапі розробки, оскільки кожен учасник команди може миттєво перевірити роботу певного маршруту без потреби у сторонніх інструментах (рис. 3.6).

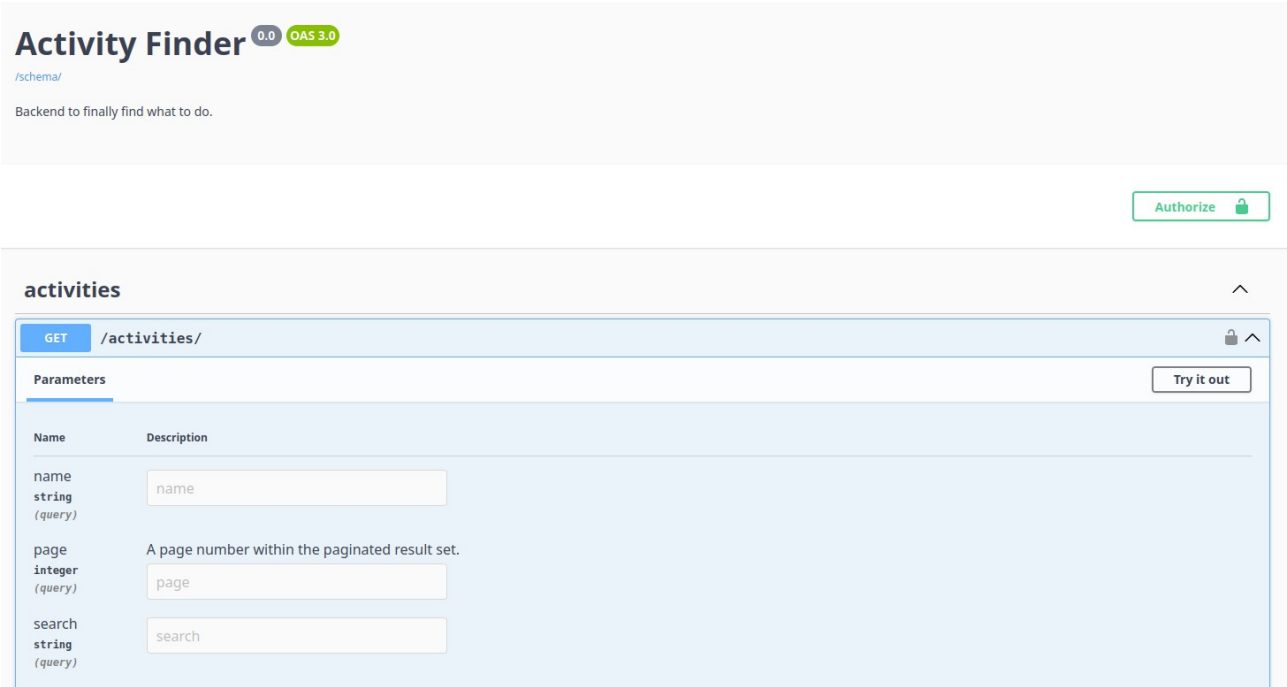


Рис. 3.6 — Інтерфейс Swagger

ReDoc надає більш лаконічний та читабельний інтерфейс, орієнтований не на тестування, а саме на вивчення API. Завдяки згортаним секціям, зручному пошуку та глибокій навігації, ReDoc ідеально підходить для зовнішньої документації або для ознайомлення нових учасників команди (рис. 3.7).

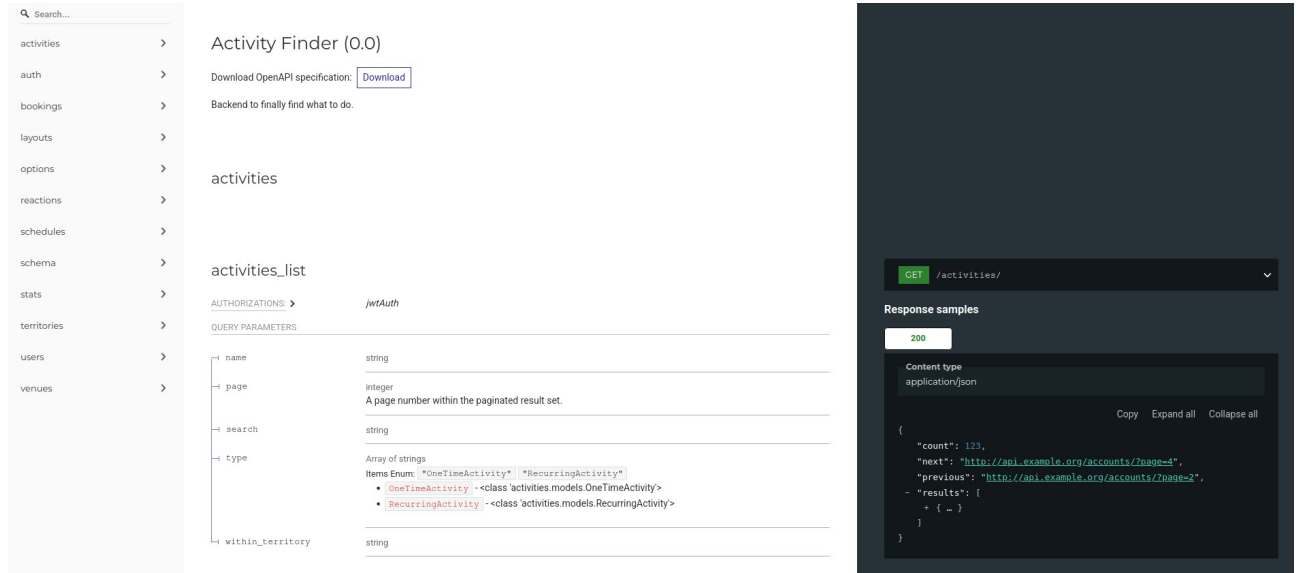


Рис. 3.7 — Інтерфейс ReDoc

Окрім інтерактивних інтерфейсів, проєкт також передбачає можливість експорту документації у форматі YAML-файлу, який містить повну схему бекенду згідно з OpenAPI 3.0 (рис. 3.8). Цей файл може бути використаний для генерації SDK, створення клієнтських бібліотек, а також для інтеграції з іншими платформами. Наявність такого файлу особливо корисна для CI/CD процесів [36], генерації контрактів і перевірок на сумісність при оновленнях API.

openapi: 3.0.3

info:

title: Activity Finder

version: '0.0'

description: Backend to finally find what to do.

paths:

/activities/:

get:

operationId: activities_list

parameters:

- in: query

name: name

schema:

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		39

- type:** string
- **name:** page
- required:** false
- in:** query
- description:** A page number within the paginated result set.
- schema:**
 - type:** integer
- **in:** query
- name:** search

Рис. 3.8 — Вигляд YAML-файлу

Таким чином, використання багатоформатної документації дозволяє задовольнити потреби як внутрішніх, так і зовнішніх користувачів системи, покращує супровідність проекту та прискорює процес впровадження нових функцій і інтеграцій.

3.3 Сценарії роботи з бекендом

Розглянемо ключові сценарії взаємодії користувача з бекендом: створення і налаштування місць проведення, подій та бронювання.

Місця проведення в API мають назву venue та є обов’язковим полем при створенні як одноразових подій, так і повторюваних. Надішлемо POST-запит на відповідну кінцеву точку (рис. 3.9).

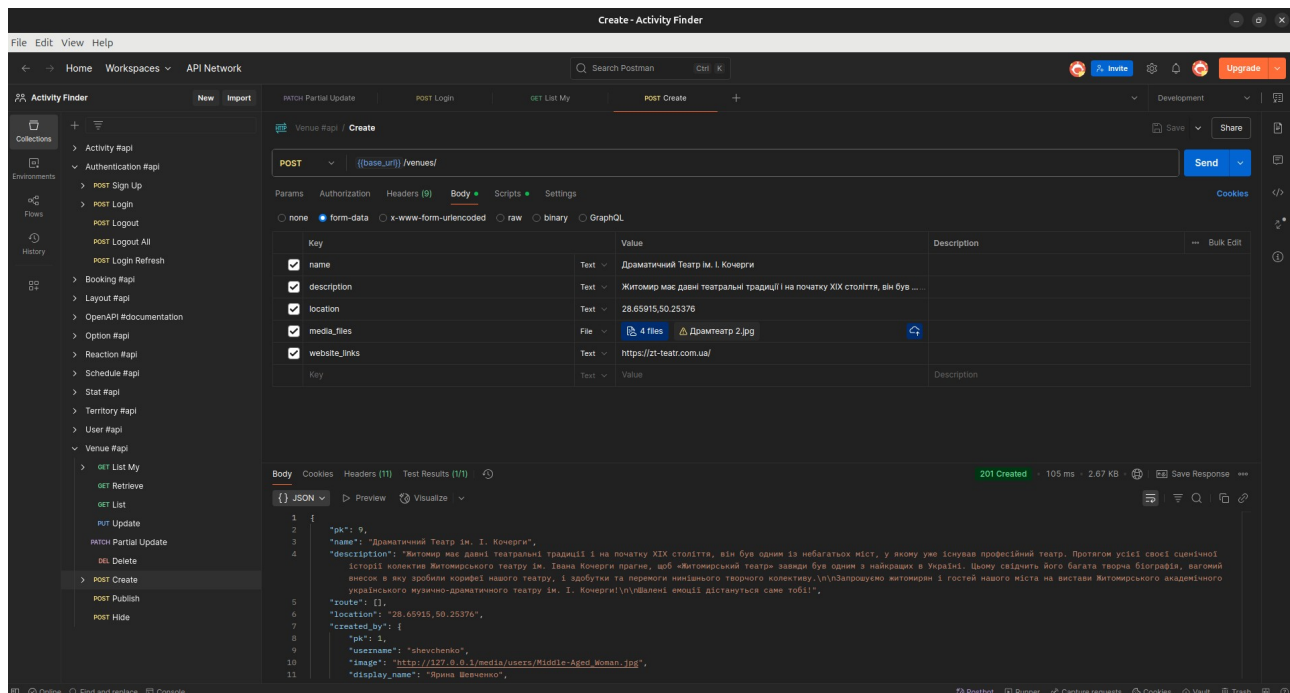


Рис. 3.9 — Створення місця проведення подій

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				40
Змн.	Арк.	№ докум.	Підпис	Дата		

Події та місця проведення мають два стани: опубліковані та приховані. Після створення об'єкти видимі лиш для користувача, що їх створив. Для опублікації вони також повинні бути заповненими. Це дозволяє створити додатковий шар валідації та розділити створення об'єктів на етапи, аби користувачі могли зберігати свої зміни, які не побачать інші користувачі.

Аби місце проведення подій було валідним для подальшої публікації, воно повинно містити щонайменше один план розміщення (layout). У разі спроби опублікувати venue без layout, система повертає помилку з відповідним повідомленням (рис. 3.10).

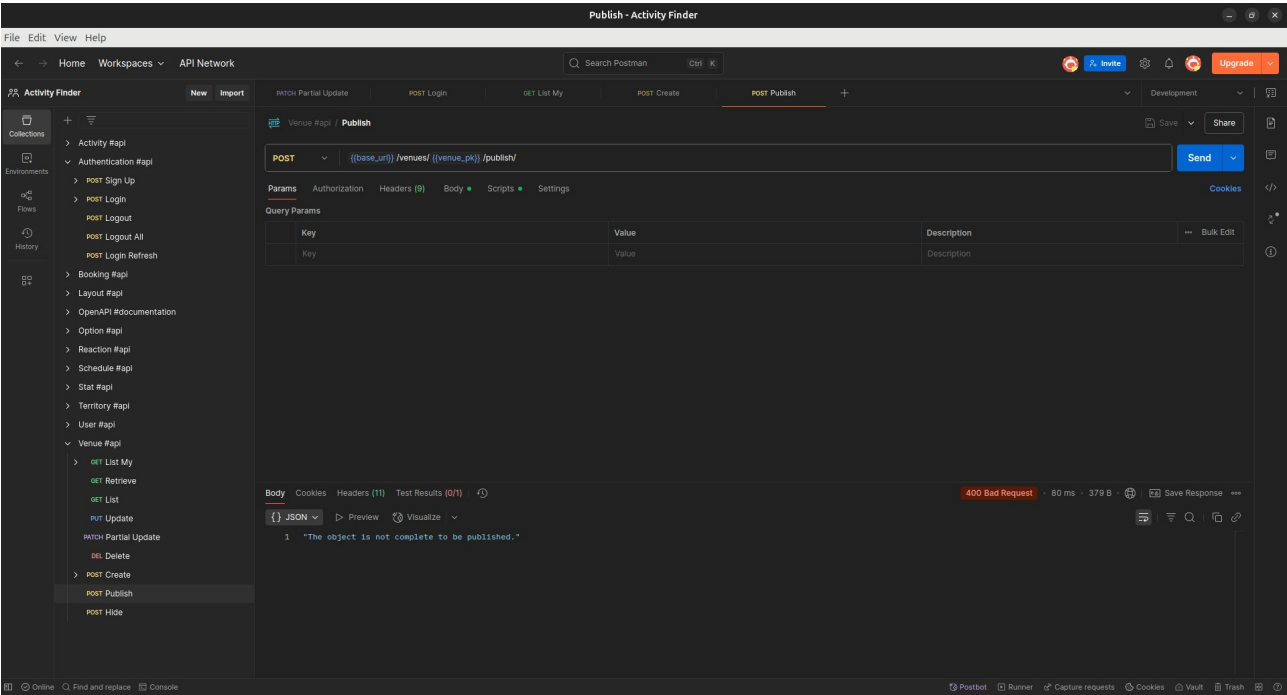


Рис. 3.10 — Відповідь при опублікуванні невалідного місця проведення

Після створення місця проведення події наступним важливим кроком є додавання layout — спеціальної схеми, яка визначає конфігурацію сидінь, секторів або зон у межах цього простору (рис. 3.11). Цей layout допомагає чітко організувати простір, забезпечуючи зручне розміщення відвідувачів і враховуючи особливості самого майданчика. Додавання layout є обов’язковою процедурою перед публікацією venue, адже без нього неможливо коректно відобразити структуру місця проведення для кінцевих користувачів. Схематична репрезентація venue надає користувачам клієнтської частини

можливість візуально ознайомитися з розташуванням сидінь та зон, що полегшує вибір місця під час придбання квитків.

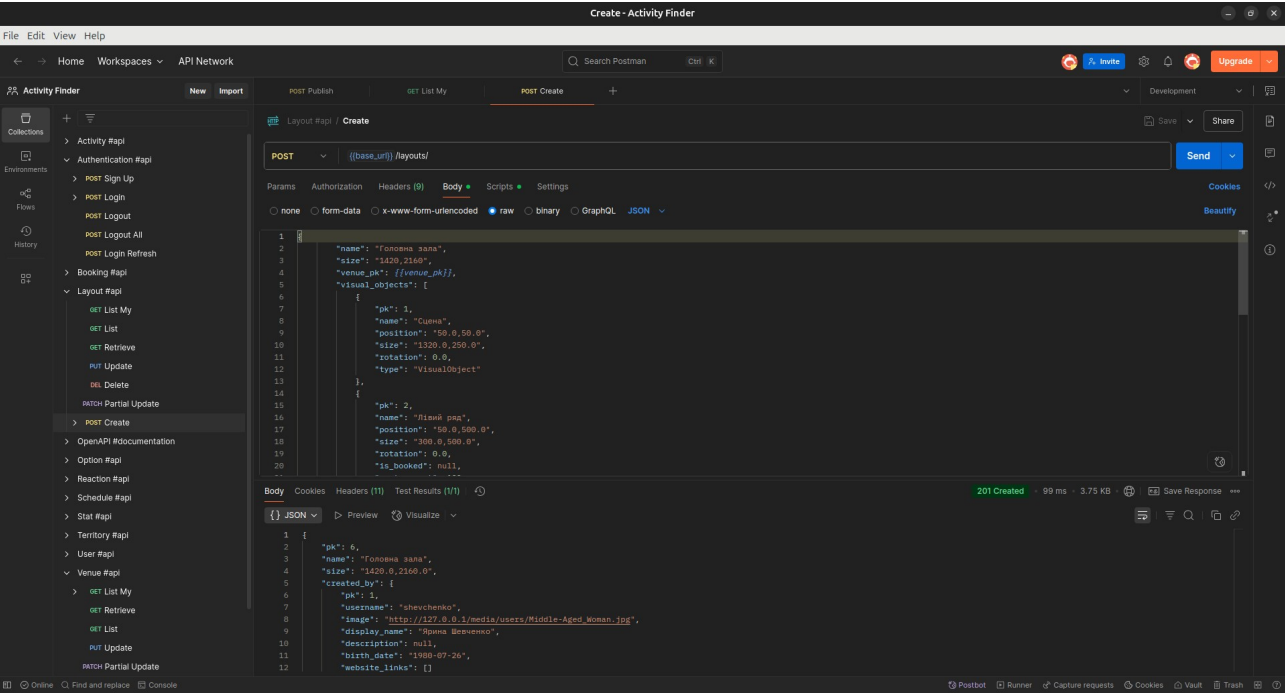


Рис. 3.11 — Створення плану розміщення

Після додавання хоча б одного layout, venue можна успішно опублікувати (рис. 3.12).

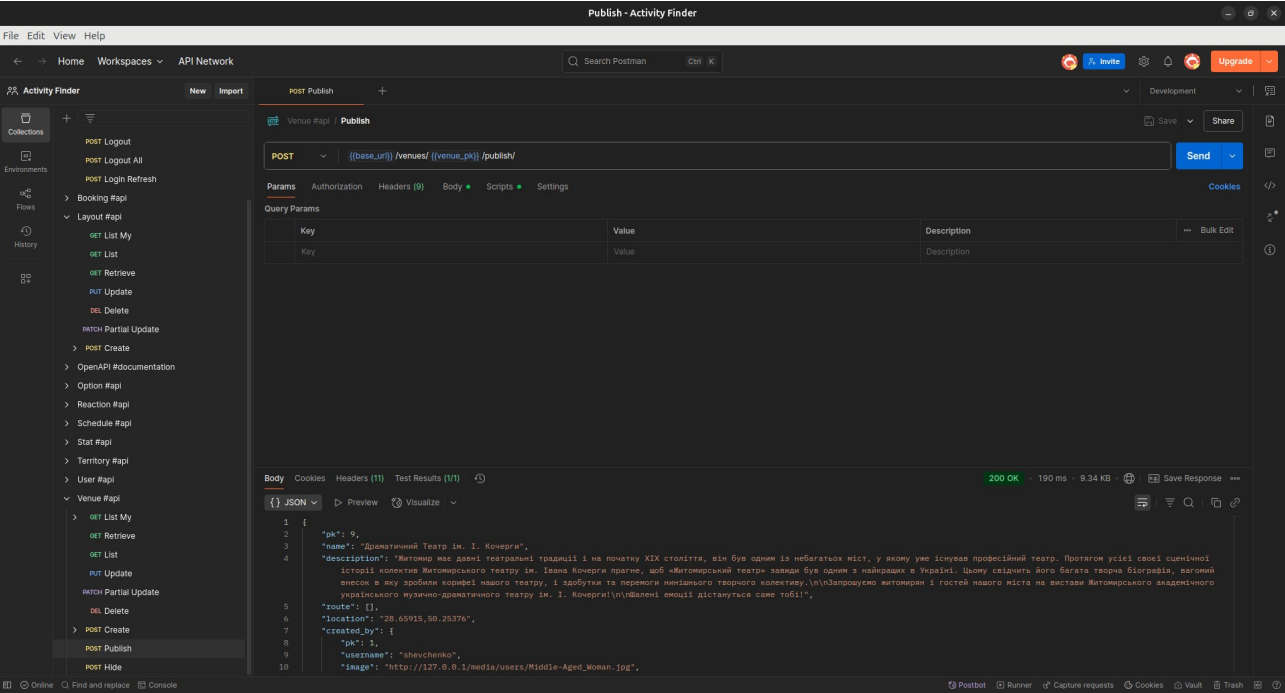


Рис. 3.12 — Успішне опублікування місця події

Наступним етапом є створення події. Платформа підтримує як одноразові (мають конкретний час і дату проведення), так і регулярні події (мають розклад,

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		42

на дозволений час якого можна забронювати місце). У прикладі нижче створюється одноразова подія з прив’язкою до опублікованого venue (рис. 3.13).

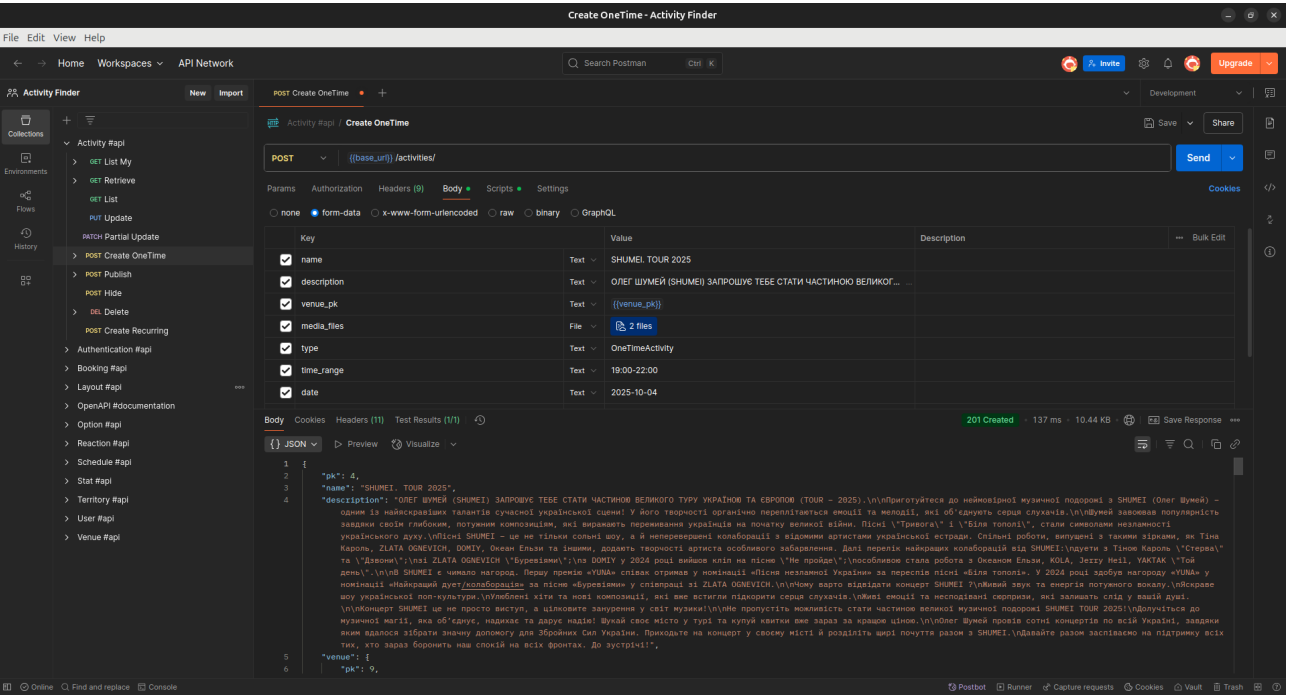


Рис. 3.13 — Створення одноразової події

Одноразова подія не вимагає додаткових етапів для публікації. Опублікуємо її, зробивши її доступною для перегляду та бронювання (рис. 3.14).

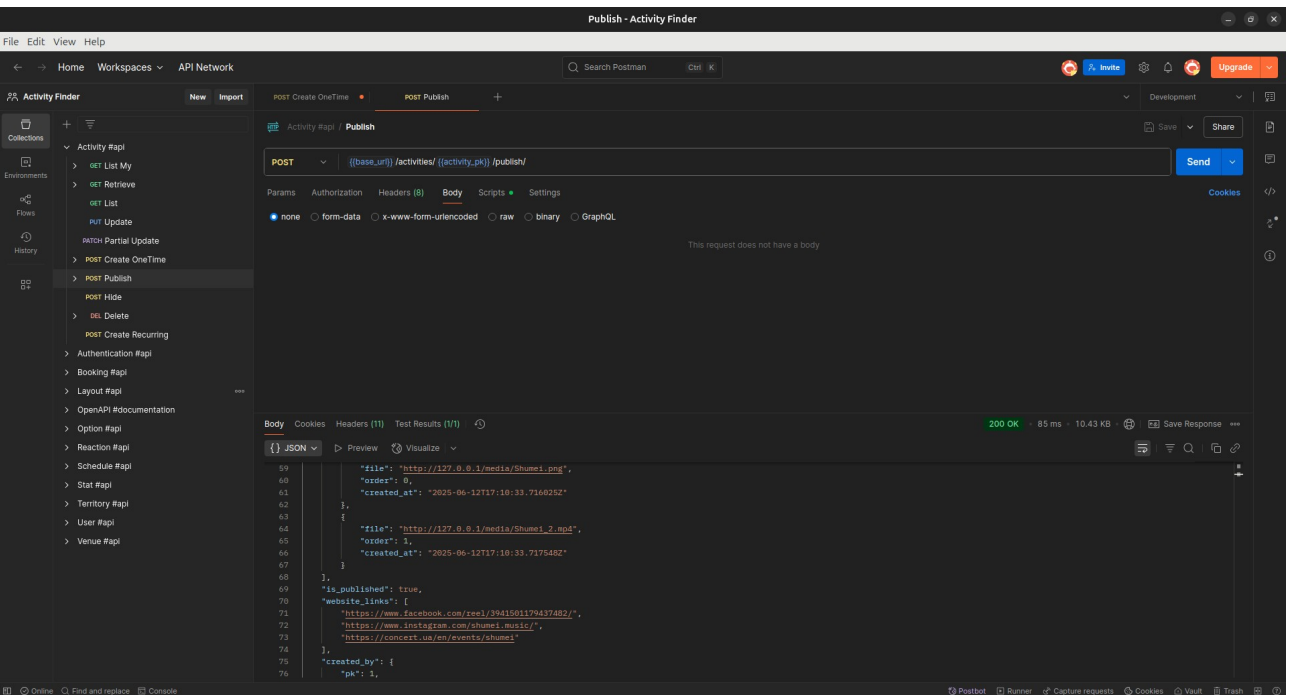


Рис. 3.14 — Опублікування події

Відвідувачі платформи матимуть свій унікальний сценарій взаємодії з бекендом, відмінний від інших ролей у системі. Зокрема, для них основними функціями будуть перегляд подій за місцем проведення та можливість створення бронювання на обрані заходи. Для реалізації цих функцій клієнтська частина застосунку може звертатися до спеціальних кінцевих точок (API), які оптимізовані під потреби користувачів.

Однією з таких точок є запит на автодоповнення назви території (рис. 3.15). Ця функція дозволяє відвідувачам швидко і зручно вибирати потрібну область, район або місто під час пошуку подій, що значно спрощує навігацію та підвищує зручність користування системою.

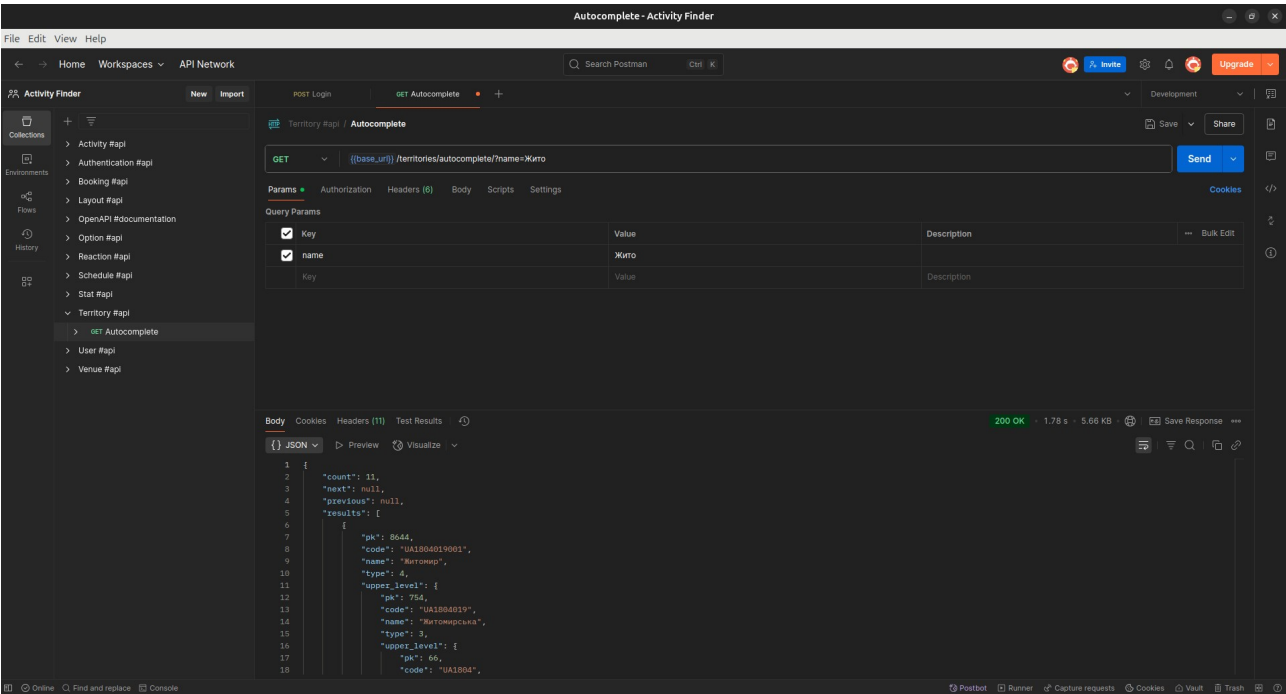


Рис. 3.15 — Запит на автодоповнення території

Отриману інформацію, зокрема первинний ключ (pk) обраної території, можна використати для фільтрації подій у системі. Завдяки цьому можна відокремити та відобразити лише ті заходи, які проводяться в конкретній області, громаді, регіоні, місті або селі, що цікавить користувача. Такий підхід значно полегшує пошук і вибір подій, адже користувачі отримують можливість швидко знаходити релевантні заходи у визначеній географічній зоні.

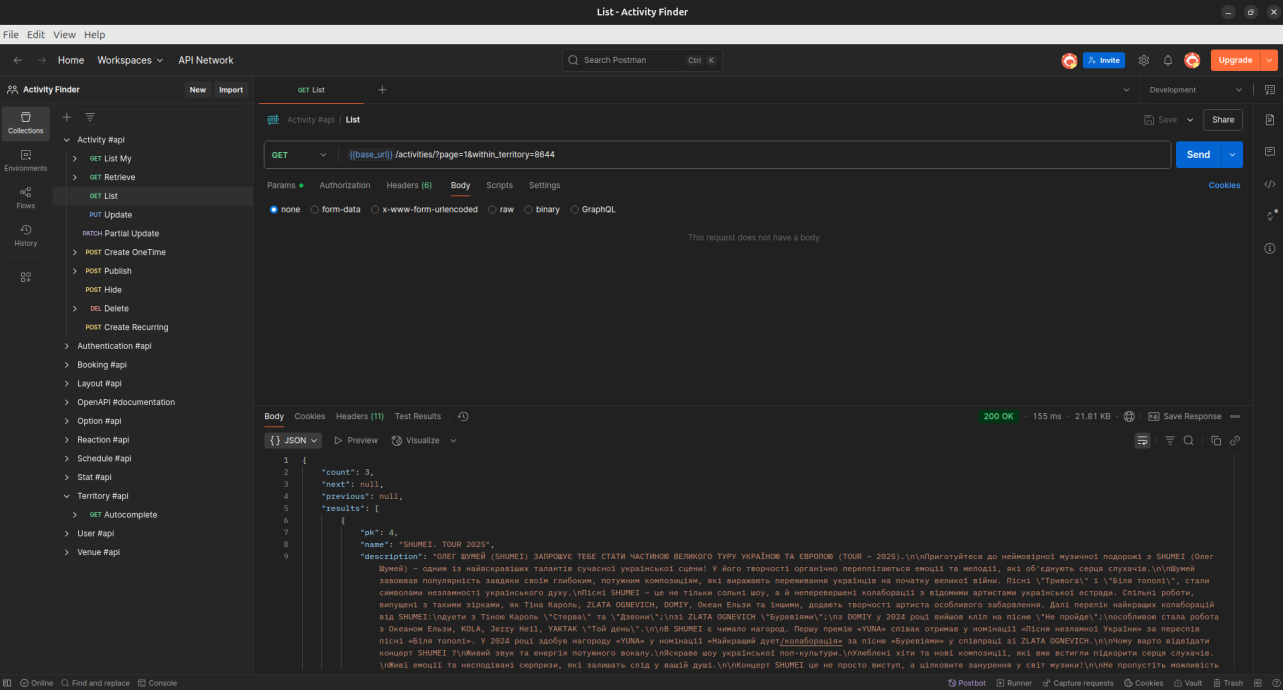


Рис. 3.16 — Фільтрація наявних подій за територією

Після цього користувач може забронювати місце на обрану подію. Бронювання враховує доступні місця відповідно до обраного layout.

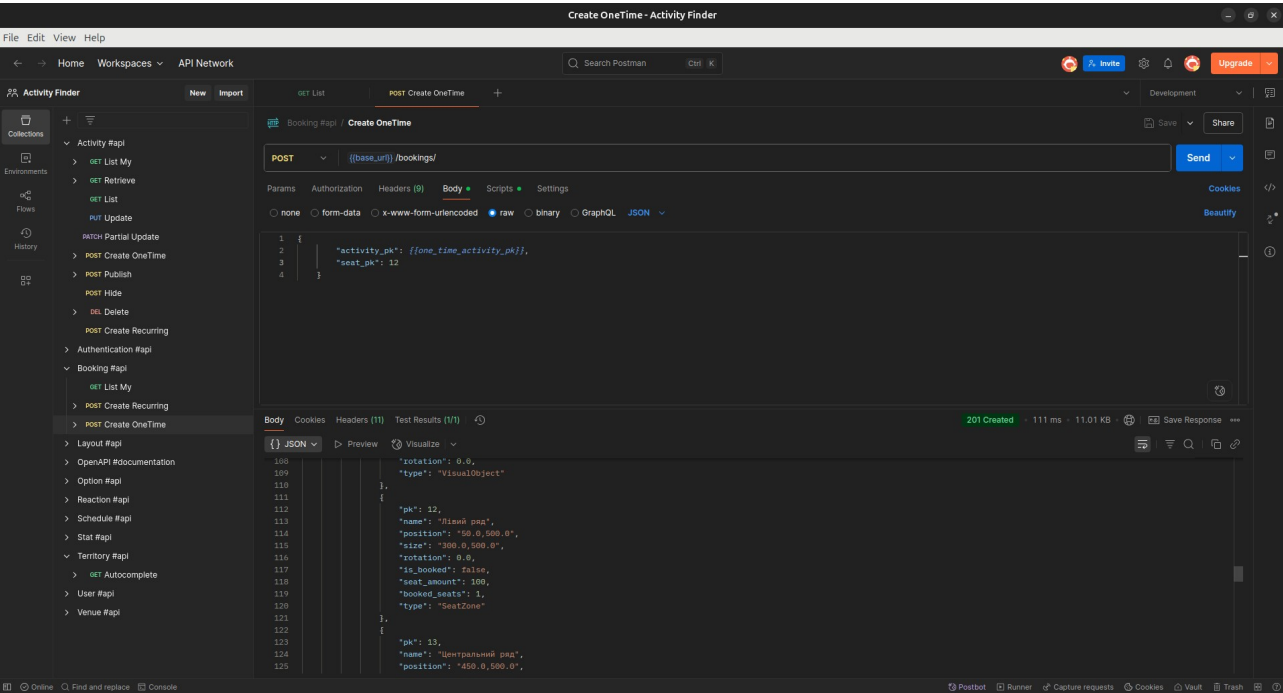


Рис. 3.17 — Бронювання місця на подію

Також варто звернути увагу на важливість кешування запитів, що значно підвищує ефективність роботи системи (рис. 3.18). Кешування дозволяє зберігати результати часто виконуваних запитів тимчасово, щоб при повторних зверненнях до тих самих даних не виконувати повний процес обробки заново.

Це знижує навантаження на сервер, прискорює час відповіді та покращує загальну продуктивність додатку

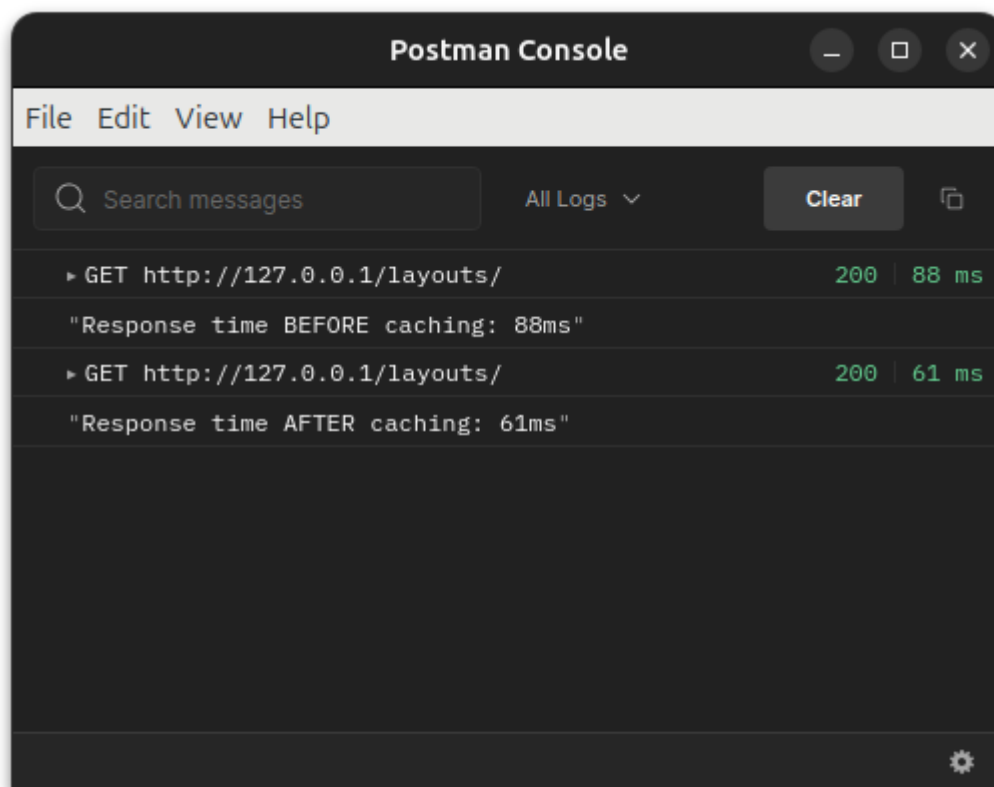


Рис. 3.18 — Результати швидкості відповіді при кешуванні

На даному рисунку видно, що час відповіді після повторного запиту (після кешування) зменшився на 27 мілісекунд, що дорівнює 30% приросту швидкості.

Висновки до розділу 3

У цьому розділі було розглянуто порядок взаємодії з бекендом системи через REST API, а також засоби, що спрощують тестування, перевірку та використання функціоналу. Середовище тестування дало змогу легко надсилати запити, перевіряти відповіді та автоматизувати рутинні операції під час роботи з сервісом.

Було представлено автоматично згенеровану документацію API — через Swagger, ReDoc та YAML-файл — що дозволяє швидко ознайомитися з доступними ендпоінтами, зрозуміти формат вхідних і вихідних даних, а також

відправляти запити прямо з браузера. Це значно полегшує взаємодію з системою.

Також були проаналізовані ключові сценарії використання: створення місця проведення, публікація події, фільтрація за територією, бронювання тощо. Кожен з них демонструє логіку роботи системи у відповідь на реальні потреби користувача, з урахуванням залежностей між об'єктами.

Таким чином, в третьому розділі показано, що робота з бекендом є послідовною, добре структурованою та зручною завдяки наявності супровідної документації, прикладів запитів і зрозумілих правил валідації. Це забезпечує комфортну взаємодію з системою навіть без поглиблених технічних знань про її внутрішню реалізацію.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		47

ВИСНОВКИ

У ході виконання кваліфікаційної роботи бакалавра було розроблено бекенд для розміщення подій та заходів, яка дозволяє організаторам публікувати події, а користувачам — здійснювати пошук і бронювання місць. Було проведено детальний аналіз предметної області, що дозволило виокремити основні вимоги до функціоналу системи, її архітектури та взаємодії між ключовими учасниками процесу — організаторами, відвідувачами та адміністраторами.

У межах роботи спроектовано архітектуру програмного забезпечення з урахуванням принципів масштабованості, безпеки та зручності розгортання. Було реалізовано повноцінний бекенд із REST API на базі Django REST Framework, розроблено структуру бази даних та моделі для обробки одноразових і повторюваних заходів, місць проведення, бронювань тощо. Система підтримує кешування для оптимізації швидкодії, перевірку прав доступу та складну валідацію даних.

Також у процесі роботи над проектом було організовано ефективну командну взаємодію. Після порівняння популярних методологій управління проектами було обрано гнучкий підхід, який найкраще відповідає вимогам до змін, масштабу команди та динаміки розробки. Незважаючи на відсутність суворого поділу ролей у цій методології, було визначено відповідальність кожного члена команди, що дозволило оптимізувати робочі процеси, зменшити кількість конфліктів та забезпечити стабільний поступ у розробці системи.

Важливу роль у забезпеченні якості системи відіграло налаштоване середовище інтеграційного тестування. Застосування Docker дало змогу ізолювати сервіси та створити незалежне тестове середовище, що дозволяє ефективно виявляти помилки на ранніх етапах. Особливу увагу приділено документації API, яка автоматично генерується через Swagger та ReDoc, забезпечуючи зручну навігацію по ендпоінтах і швидке ознайомлення з функціоналом.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		48

У роботі також було описано основні сценарії використання системи — від створення місць проведення заходів і публікації подій до фільтрації за територією, автозаповнення назв, а також бронювання місць. Для демонстрації та тестування взаємодії з системою використовувався Postman, що дозволило на практиці перевірити роботу кожного етапу користувацького сценарію.

Таким чином, розроблена платформа відповідає сучасним вимогам до бекенду. Вона є масштабованою, гнучкою до розширення функціоналу, забезпечує зручну взаємодію з користувачем та дозволяє ефективно керувати подієвим контентом. Отримані результати демонструють готовність системи до використання у реальному середовищі та подальшого розвитку.

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		49

СПИСКИ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Quality of Life to Value Generator comparison [Електронний ресурс] – Режим доступу до ресурсу: <https://demandmaven.substack.com/p/the-quality-of-life-improvement-trap>
2. Django Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.djangoproject.com/en/5.1/>
3. Django REST Framework [Електронний ресурс] – Режим доступу до ресурсу: <https://www.django-rest-framework.org/>
4. PyTest Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.pytest.org/en/stable/contents.html>
5. Instagram Atlas Weekend [Електронний ресурс] – Режим доступу до ресурсу: <https://www.instagram.com/atlasweekend/>
6. Telegram Atlas Weekend [Електронний ресурс] – Режим доступу до ресурсу: <https://t.me/atlasweekend>
7. Concert [Електронний ресурс] – Режим доступу до ресурсу: <https://concert.ua/>
8. Karabas [Електронний ресурс] – Режим доступу до ресурсу: <https://karabas.com/>
9. What is Multi-Layered Software Architecture [Електронний ресурс] – Режим доступу до ресурсу: <https://www.packtpub.com/en-us/learning/how-to-tutorials/what-is-multi-layered-software-architecture>
10. Clean Architecture [Електронний ресурс] – Режим доступу до ресурсу: <https://codilime.com/blog/clean-architecture/>
11. PostgreSQL Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://www.postgresql.org/docs/>
12. PostGIS Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://postgis.net/documentation/>
13. Redis Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://redis.io/docs/latest/>

14. Docker Documentation [Електронний ресурс] – Режим доступу до ресурсу:
<https://docs.docker.com/>
15. Python Documentation [Електронний ресурс] – Режим доступу до ресурсу:
<https://docs.python.org/3.12/>
16. Top 15 Most Popular Project Management Methodologies [Електронний ресурс] – Режим доступу до ресурсу:
<https://www.projectmanager.com/blog/project-management-methodology>
17. Shortform Books: Inspired by Marty Cagan [Електронний ресурс] – Режим доступу до ресурсу: <https://www.shortform.com/blog/inspired-marty-cagan/>
18. Significance of Flowchart [Електронний ресурс] – Режим доступу до ресурсу: <https://www.ucl.ac.uk/culture-online/case-studies/2021/jun/what-significance-flowchart>
19. How does Trigram Search work [Електронний ресурс] – Режим доступу до ресурсу: <https://www.cockroachlabs.com/docs/stable/trigram-indexes>
20. Understanding GIN Indexes [Електронний ресурс] – Режим доступу до ресурсу: <https://pganalyze.com/blog/gin-index>
21. Django models [Електронний ресурс] – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/python/django-models/>
22. OOP benefits [Електронний ресурс] – Режим доступу до ресурсу: <https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP>
23. How to upload multiple files in DRF: Answer [Електронний ресурс] – Режим доступу до ресурсу: <https://stackoverflow.com/a/71293707/20928849>
24. What is Integration Testing [Електронний ресурс] – Режим доступу до ресурсу: <https://katalon.com/resources-center/blog/integration-testing>
25. Disadvantages of Module testing [Електронний ресурс] – Режим доступу до ресурсу: <https://www.tencentcloud.com/techpedia/104357>
26. Usage of Fixtures [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.pytest.org/en/stable/how-to/fixtures.html>

27. Unit tests make refactoring harder [Електронний ресурс] – Режим доступу до ресурсу: <https://www.effective-software-testing.com/do-unit-tests-make-refactoring-harder>
28. What is Microservices Architecture [Електронний ресурс] – Режим доступу до ресурсу: <https://cloud.google.com/learn/what-is-microservices-architecture>
29. Why do we need Django apps: Discussion [Електронний ресурс] – Режим доступу до ресурсу: <https://forum.djangoproject.com/t/why-do-we-need-apps/827/3>
30. How Django's MVT Architecture Works [Електронний ресурс] – Режим доступу до ресурсу: <https://www.freecodecamp.org/news/how-django-mvt-architecture-works/>
31. Postman's Benefits [Електронний ресурс] – Режим доступу до ресурсу: <https://www.geeksforgeeks.org/what-are-the-key-benefits-of-using-postman-for-api-development/>
32. How to write Postman Scripts [Електронний ресурс] – Режим доступу до ресурсу: <https://learning.postman.com/docs/tests-and-scripts/write-scripts/intro-to-scripts/>
33. Swagger [Електронний ресурс] – Режим доступу до ресурсу: <https://swagger.io/>
34. ReDoc [Електронний ресурс] – Режим доступу до ресурсу: <https://redocly.com/redoc>
35. YAML OpenAPI specification [Електронний ресурс] – Режим доступу до ресурсу: <https://swagger.io/specification/>
36. YAML for CI/CD [Електронний ресурс] – Режим доступу до ресурсу: <https://medium.com/@dneimke/working-with-openapi-in-vs-code-and-ci-cd-pipelines-f328d6c39c14>

ДОДАТКИ

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		53

web/commons/serializers/fields.py:

```
from rest_framework import serializers
from django.contrib.gis.geos import Point
from drf_spectacular.utils import extend_schema_field
```

```
@extend_schema_field(
    {
        "type": "string",
        "description": "Coordinates in `~12.95432,25.43335` format.",
    }
)
class PointField(serializers.Field):
    def __init__(self, precision=6, dimension=None, *args, **kwargs):
        self.precision = precision
        self.dimension = dimension
        super().__init__(*args, **kwargs)
        self.error_messages["dimension"] = (
            "Invalid coordinate's dimension. Expected: {dimension}."
        )

    def bind(self, field_name, parent):
        super().bind(field_name, parent)
        if self.dimension is None:
            model_field = getattr(parent.Meta.model, field_name).field
            self.dimension = model_field.dim

    def to_representation(self, value):
        if not isinstance(value, Point):
            return None

        representation = [str(round(value, self.precision)) for value in
            value.coords]
        return ",".join(representation)

    def to_internal_value(self, data):
        if data is None:
            return None

        if not isinstance(data, str):
            raise serializers.ValidationError("Invalid type provided.")

        data = [float(value) for value in data.split(",")]
        if len(data) != self.dimension:
```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		54

```

        self.fail("dimension", dimension=self.dimension)

    return Point(data)

class PairField(serializers.ListField):
    separator = None
    child = None

    min_length = 2
    max_length = 2

    def to_representation(self, data):
        representations = []
        for obj in data:
            result = self.child.to_representation(obj)
            representations.append(str(result))
        return self.separator.join(representations)

    def to_internal_value(self, data: str):
        if isinstance(data, list):
            data = data[0]
        if not isinstance(data, str):
            raise serializers.ValidationError("Invalid type provided.")
        data = [obj.strip() for obj in data.split(self.separator)]
        return super().to_internal_value(data)

class WebsiteLinksField(serializers.ListField):
    child = serializers.URLField()
    separator = ","
    max_length = 10

    def to_internal_value(self, data: str):
        if isinstance(data, list):
            data = data[0]
        if not isinstance(data, str):
            raise serializers.ValidationError("Invalid type provided.")
        data = [obj.strip() for obj in data.split(self.separator)]
        return super().to_internal_value(data)

    @extend_schema_field(
        {
            "type": "string",
            "description": "Time range in `HH:MM[:SS]-HH:MM[:SS]` format.",
        }
    )
class TimeRangeField(PairField):

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		55

```

child = serializers.TimeField()
separator = "-"

class FloatPairField(PairField):
    separator = ","

    def __init__(self, **kwargs):
        self.min_value = kwargs.pop("min_value", None)
        self.max_value = kwargs.pop("max_value", None)
        kwargs["child"] = serializers.FloatField(
            min_value=self.min_value, max_value=self.max_value
        )
        super().__init__(**kwargs)
web/commons/serializers/utils.py:

from copy import copy
from uuid import uuid4
from rest_framework import serializers
from rest_framework.utils.field_mapping import get_relation_kwargs
from drf_spectacular.utils import extend_schema_field
from drf_spectacular.openapi import OpenApiTypes

from users.serializers import UserSerializer
from files.serializers import FileSerializer


def remove_serializer_fields(base_cls, remove_fields):
    original_fields = getattr(base_cls.Meta, "fields", ())

    class Meta:
        model = base_cls.Meta.model
        fields = tuple(f for f in original_fields if f not in remove_fields)

    return type(
        f"Nested{base_cls.__name__}{uuid4().hex}",
        (base_cls,),
        {
            "Meta": Meta,
            "get_fields": lambda self: {
                k: v
                for k, v in super(base_cls, self).get_fields().items()
                if k not in remove_fields
            },
        },
    )

```

```

def _add_fields(meta, *fields):
    if hasattr(meta, "fields"):
        fields_type = type(meta.fields)
        if not isinstance(fields, fields_type):
            fields = fields_type(fields)
        meta.fields += fields

def add_multiple_file_fields(attrs, multiple_file_fields, info):
    for serializer_field_name, extra_kwargs in multiple_file_fields.items():
        source = serializer_field_name
        files_field_name = serializer_field_name + "_files"

        attrs[serializer_field_name] = FileSerializer(
            many=True, read_only=True, required=False
        )

        files_field_kwargs = get_relation_kwargs(
            serializer_field_name, info.relations[source]
        )
        invalid_kwargs = ("view_name", "to_field", "source", "queryset", "many")
        for name in invalid_kwargs:
            files_field_kwargs.pop(name, None)

        attrs[files_field_name] = serializers.ListField(
            child=serializers.FileField(**extra_kwargs),
            write_only=True,
            **files_field_kwargs,
        )

        _add_fields(attrs["Meta"], serializer_field_name, files_field_name)

    return attrs

def add_fk_serializers(attrs, fk_serializers, info):
    for serializer_field_name, kwargs in fk_serializers.items():
        kwargs = copy(kwargs)
        serializer_field = kwargs.pop("serializer")
        pk_field_name = serializer_field_name + "_pk"
        source = serializer_field_name

        pk_field_kwargs = get_relation_kwargs(source, info.relations[source])
        invalid_kwargs = ("view_name", "to_field")
        for name in invalid_kwargs:
            pk_field_kwargs.pop(name, None)

        pk_field_kwargs["source"] = source

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		57

```

pk_field_kwargs = pk_field_kwargs | kwargs

pk_field = serializers.PrimaryKeyRelatedField(**pk_field_kwargs)

attrs[serializer_field_name] = serializer_field(read_only=True,
required=False)
attrs[pk_field_name] = pk_field

_add_fields(attrs["Meta"], serializer_field_name, pk_field_name)

return attrs

def add_current_user_serializer(attrs, source):
    serializer_field_name = source
    pk_field_name = source + "_pk"

    serializer_field = UserSerializer(read_only=True, required=False)
    pk_field = serializers.HiddenField(
        default=serializers.CurrentUserDefault(),
        source=source,
    )
    attrs[serializer_field_name] = serializer_field
    attrs[pk_field_name] = pk_field

    _add_fields(attrs["Meta"], serializer_field_name, pk_field_name)

    return attrs

def add_choice_display_fields(attrs, fields):
    for name in fields:
        display_property_name = f"{name}_display"
        attrs[display_property_name] = serializers.SerializerMethodField()

        display_method_name = f"get_{display_property_name}"

        def method(self, obj):
            return getattr(obj, display_method_name)()

        method.__name__ = display_method_name
        attrs[display_method_name] = method
    extend_schema_field(OpenApiTypes.STR)(method)

    _add_fields(attrs["Meta"], display_property_name)

    return attrs

```

web/commons/serializers/mixins.py:

```
from rest_framework import serializers
from rest_framework.utils.model_meta import get_field_info
from django.contrib.gis.db.models import PointField as ModelPointField,
Manager, Model
```

```
from commons.models import (
    TimeRangeField as ModelTimeRangeField,
    FloatPairField as ModelFloatPairField,
)
from files.models import File
```

```
from .utils import (
    add_choice_display_fields,
    add_current_user_serializer,
    add_fk_serializers,
    add_multiple_file_fields,
)
from .fields import TimeRangeField, PointField, FloatPairField
```

```
class MainModelSerializerMetaclass(serializers.SerializerMetaclass):
    def __new__(cls, name, bases, attrs):
        meta = attrs.get("Meta", None)
        if meta:
            model = getattr(meta, "model", None)
            if model:
                info = get_field_info(model)

                current_user_field = getattr(meta, "current_user_field", None)
                if current_user_field:
                    attrs = add_current_user_serializer(attrs, current_user_field)

                choice_fields = getattr(meta, "choice_fields", None)
                if choice_fields:
                    attrs = add_choice_display_fields(attrs, choice_fields)

                fk_serializers = getattr(meta, "fk_serializers", None)
                if fk_serializers and model:
                    attrs = add_fk_serializers(attrs, fk_serializers, info)

                multiple_file_fields = getattr(meta, "multiple_file_fields", None)
                if multiple_file_fields and model:
                    attrs = add_multiple_file_fields(attrs, multiple_file_fields, info)
```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		59

```

    return super().__new__(cls, name, bases, attrs)

class MainModelSerializerMixin:
    def __init__(self, *args, **kwargs):
        self.serializer_field_mapping[ModelTimeRangeField] = TimeRangeField
        self.serializer_field_mapping[ModelFloatPairField] = FloatPairField
        self.serializer_field_mapping[ModelPointField] = PointField
        super().__init__(*args, **kwargs)

    def get_current(self, field_name):
        field = self.fields[field_name]

        if field_name in self.initial_data:
            return field.to_internal_value(self.initial_data[field_name])

        if self.instance:
            value = field.get_attribute(self.instance)

            if isinstance(value, (Model, Manager)):
                value = field.to_representation(value)

            if value is None:
                return None

            return field.to_internal_value(value)

    def _save_multiple_file(self, instance, files):
        for field_name, files in files.items():
            objs = []
            for order, file in enumerate(files):
                objs.append(File(file=file, order=order))

            objs = File.objects.bulk_create(objs)
            getattr(instance, field_name).set(objs)

    def _pop_multiple_files(self, data):
        meta = getattr(self, "Meta", None)
        if meta is None:
            return

        multiple_file_fields = getattr(meta, "multiple_file_fields", None)
        if multiple_file_fields is None:
            return

        files = {}

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		60

```

for field_name in multiple_file_fields.keys():
    file_field_name = field_name + "_files"
    if file_field_name in data:
        files[field_name] = data.pop(file_field_name)

```

```

return files

```

```

def create(self, validated_data):
    files = self._pop_multiple_files(validated_data)
    instance = super().create(validated_data)
    if files is not None:
        self._save_multiple_file(instance, files)
    return instance

def update(self, instance, validated_data):
    files = self._pop_multiple_files(validated_data)
    instance = super().update(instance, validated_data)
    if files is not None:
        self._save_multiple_file(instance, files)
    return instance

```

web/commons/serializers/__init__.py:

```

from rest_framework import serializers
from drf_writable_nested.serializers import WritableNestedModelSerializer
from rest_polymorphic.serializers import PolymorphicSerializer

```

```

from .mixins import MainModelSerializerMixin,
MainModelSerializerMetaclass

```

```

class MainModelSerializer(
    MainModelSerializerMixin,
    serializers.ModelSerializer,
    metaclass=MainModelSerializerMetaclass,
):
    """

```

```

    Adds new attributes to `Meta`:
    - `current_user_field` takes a field and implicitly sets its value to current
    user's model
    - `choice_display_fields` takes a list of fields and adds `<name>_display`
    read-only field pair to each of them
    - `fk_serializers` takes a dict `{<field_name>: {"serializer":
    <serializer_class>}, **extra_kwargs}` and creates `<name>_pk` field pair that
    expects pk

```

```
- `multiple_file_fields` takes a dict `{<field_name>:
<extra_file_field_kwargs>}` and makes set fields to work for multiple file
upload
"""
```

```
pass
```

```
class MainWritableNestedModelSerializer(
    MainModelSerializerMixin,
    WritableNestedModelSerializer,
    metaclass=MainModelSerializerMetaclass,
):
    pass
```

```
class MainPolymorphicSerializer(PolymorphicSerializer):
    """
    Forbids changing the type on already created objects.
    """
```

```
def run_validation(self, data):
    if self.instance and self.resource_type_field_name in data:
        new_type = self._get_resource_type_from_mapping(data)
        old_type = self.instance.__class__.__name__
        if new_type != old_type:
            raise serializers.ValidationError(
                {
                    self.resource_type_field_name: f"You can't change
`{self.resource_type_field_name}` on created activity."
                }
            )
        return super().run_validation(data)
```

```
web/commons/viewsets.py:
```

```
from rest_framework.decorators import action
from rest_framework.viewsets import ModelViewSet
from rest_framework.permissions import IsAuthenticated
```

```
from .utils import resolve_nested_attribute
```

```
def get_user_or_401(self, request):
    permission = IsAuthenticated()
    if not permission.has_permission(request, self):
        self.permission_denied(
            request,
```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		62

```

        message=getattr(permission, "message", None),
        code=getattr(permission, "code", None),
    )
else:
    return request.user

def set_model_viewset_actions(base_cls, action_prefix=""):
    def new_partial_update(self, request, *args, **kwargs):
        kwargs["partial"] = True
        update = getattr(self, "update" + action_prefix)
        return update(request, *args, **kwargs)

    name_action_map = {
        "list": ModelViewSet.list,
        "retrieve": ModelViewSet.retrieve,
        "create": ModelViewSet.create,
        "update": ModelViewSet.update,
        "partial_update": new_partial_update,
        "destroy": ModelViewSet.destroy,
    }
    for name, func in name_action_map.items():
        setattr(base_cls, name + action_prefix, func)

    if not hasattr(base_cls, "perform_update"):
        setattr(base_cls, "perform_update", ModelViewSet.perform_update)
    if not hasattr(base_cls, "perform_create"):
        setattr(base_cls, "perform_create", ModelViewSet.perform_create)
    if not hasattr(base_cls, "perform_destroy"):
        setattr(base_cls, "perform_destroy", ModelViewSet.perform_destroy)

    return base_cls

def with_my_object_endpoint(
    pk_field_name=None,
    methods=["get", "put", "patch", "delete"],
    endpoint_name="my",
):
    """
    Adds a `my/` endpoint to a model's `ViewSet`, allowing interaction
    with current user's object.

    The object is resolved with `pk_field_name` from current user's model.
    For example, `profile.pk` covers direct or reverse relation to `Profile`
    model's pk.

```

Overwrite `perform\_update()`, `perform\_destroy()`

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		63

```

for `request.action == endpoint_name` to tweak the functionality.
"""

def decorator(base_cls):
    method_action_map = {
        "GET": "retrieve",
        "PUT": "update",
        "PATCH": "partial_update",
        "DELETE": "destroy",
    }
    prefix = "_my_object"

    def endpoint(self, request):
        current_user = get_user_or_401(self, request)
        self.kwargs["pk"] = resolve_nested_attribute(current_user,
pk_field_name)

        method_name = method_action_map[request.method] + prefix
        return getattr(self, method_name)(request, *self.args, **self.kwargs)

    endpoint.__name__ = endpoint_name
    my_action = action(detail=False, methods=methods)(endpoint)

    setattr(base_cls, endpoint_name, my_action)
    set_model_viewset_actions(base_cls, action_prefix=prefix)

    return base_cls

return decorator

def with_my_list_endpoint(
    field_name,
    methods=["get", "post"],
    endpoint_name="my",
    has_pagination=False,
):
    """
    Adds a `my/` endpoint to a model's `ViewSet`, allowing interaction
    with current user's list of objects.

    The object is resolved with `field_name` from current user's model.
    For example, `subscribers` covers direct or reverse relation to `Subscriber`
    model.

    Overwrite `.perform_create()` or any method that interacts with
    `queryset`

```

(except `.get_queryset()`) for `request.action == endpoint_name` to tweak the functionality.

```

def decorator(base_cls):
    method_action_map = {
        "GET": "list",
        "PUT": "create",
    }
    prefix = "_my_list"

    original_get_queryset = base_cls.get_queryset
    original_paginate_queryset = base_cls.paginate_queryset

    def new_paginate_queryset(self, queryset):
        current_action = getattr(self, "action", None)
        if current_action == endpoint_name:
            return None
        return original_paginate_queryset(self, queryset)

    def new_get_queryset(self):
        current_action = getattr(self, "action", None)
        if current_action == endpoint_name:
            return getattr(self, "queryset" + prefix)
        return original_get_queryset(self)

    def endpoint(self, request):
        current_user = get_user_or_401(self, request)
        queryset = resolve_nested_attribute(current_user, field_name).all()
        setattr(self, "queryset" + prefix, queryset)

        method_name = method_action_map[request.method] + prefix
        return getattr(self, method_name)(request, *self.args, **self.kwargs)

    if not has_pagination:
        new_paginate_queryset.__name__ = "paginate_queryset"
        setattr(base_cls, "paginate_queryset", new_paginate_queryset)

    new_get_queryset.__name__ = "get_queryset"
    setattr(base_cls, "get_queryset", new_get_queryset)

    endpoint.__name__ = endpoint_name
    my_action = action(detail=False, methods=methods)(endpoint)
    setattr(base_cls, endpoint_name, my_action)

    set_model_viewset_actions(base_cls, action_prefix=prefix)

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		65

```
return base_cls
```

```
return decorator
```

```
web/activities/models.py:
```

```
from django.contrib.gis.db import models
from django.contrib.postgres.fields import ArrayField
from polymorphic.models import PolymorphicModel
```

```
from users.models import User
from venues.models import Venue
from layouts.models import Layout
from commons.models import TimeRangeField
from schedules.models import Schedule
from files.models import File
```

```
class Activity(PolymorphicModel):
    class Meta:
        default_related_name = "activities"
        ordering = ["name", "pk"]

    name = models.CharField(max_length=64)
    description = models.TextField()
    created_by = models.ForeignKey(User, models.CASCADE)
    venue = models.ForeignKey(Venue, models.PROTECT)
    media = models.ManyToManyField(File)
    is_published = models.BooleanField(default=False, blank=True,
editable=False)
    website_links = ArrayField(models.URLField(), size=10, blank=True,
default=list)

    def __str__(self):
        return self.name

    def __repr__(self):
        return f"Activity({self.name})"

class OneTimeActivity(Activity):
    class Meta(Activity.Meta):
        default_related_name = "one_time_activities"

    time_range = TimeRangeField()
    layout = models.ForeignKey(Layout, models.PROTECT)
```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		66

```

date = models.DateField()

def __repr__(self):
    return f"OneTimeActivity({self.name})"

class RecurringActivity(Activity):
    class Meta(Activity.Meta):
        default_related_name = "recurring_activities"

    schedule = models.ForeignKey(Schedule, models.CASCADE,
related_name="activities")
    duration = models.DurationField()

def __repr__(self):
    return f"RecurringActivity({self.name})"

web/activities/serializers.py:

from rest_framework import serializers

from commons.serializers import MainModelSerializer,
MainPolymorphicSerializer

from commons.serializers.utils import remove_serializer_fields
from files.validators import media_content_type_validator
from users.validators import OwnedByCurrentUser
from schedules.serializers import ScheduleSerializer
from activities.models import Activity, OneTimeActivity, RecurringActivity
from venues.serializers import VenueSerializer
from layouts.serializers import LayoutSerializer

NestedScheduleSerializer = remove_serializer_fields(ScheduleSerializer,
["activities"])
NestedVenueSerializer = remove_serializer_fields(VenueSerializer,
["activities"])

class ActivitySerializer(MainModelSerializer):
    class Meta:
        model = Activity
        fields = (
            "pk",
            "name",
            "description",
            "venue",
            "media",

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		67

```

        "is_published",
        "website_links",
    )
    read_only_fields = ("pk", "is_published")
    current_user_field = "created_by"
    multiple_file_fields = {"media": {"validators":
[media_content_type_validator]}}
    fk_serializers = {"venue": {"serializer": NestedVenueSerializer}}

def validate_venue(self, venue):
    if not venue.is_published:
        raise serializers.ValidationError(
            {"venue_pk": "The object is not published."}
        )

class OneTimeActivitySerializer(ActivitySerializer):
    class Meta(ActivitySerializer.Meta):
        model = OneTimeActivity
        fields = ActivitySerializer.Meta.fields + (
            "time_range",
            "date",
            "layout",
        )
        fk_serializers = ActivitySerializer.Meta.fk_serializers | {
            "layout": {"serializer": LayoutSerializer}
        }

def validate(self, data):
    venue = data.get("venue") or self.get_current("venue")
    layout = data.get("layout") or self.get_current("layout")
    if venue != layout.venue:
        raise serializers.ValidationError(
            {"layout": "The object is not related to the current venue."}
        )
    return super().validate(data)

def to_representation(self, instance):
    self.fields["layout"].context["activity"] = instance
    return super().to_representation(instance)

class RecurringActivitySerializer(ActivitySerializer):
    class Meta(ActivitySerializer.Meta):
        model = RecurringActivity
        fields = ActivitySerializer.Meta.fields + (

```

```

        "schedule",
        "duration",
        "options",
    )
    fk_serializers = {
        "schedule": {
            "serializer": NestedScheduleSerializer,
            "validators": [OwnedByCurrentUser("created_by")],
        }
    }

def to_representation(self, instance):
    self.fields["schedule"].context["activity"] = instance
    return super().to_representation(instance)

def get_fields(self):
    from options.serializers import OptionSerializer

    fields = super().get_fields()
    NestedOptionSerializer = remove_serializer_fields(
        OptionSerializer, ["activity"]
    )
    fields["options"] = NestedOptionSerializer(many=True, read_only=True)
    return fields

class PolymorphicActivitySerializer(MainPolymorphicSerializer):
    model_serializer_mapping = {
        OneTimeActivity: OneTimeActivitySerializer,
        RecurringActivity: RecurringActivitySerializer,
    }
    resource_type_field_name = "type"

```

web/commons/filtersets.py:

```

from django.core.validators import EMPTY_VALUES
from django.contrib.postgres.search import SearchVector
from django_filters import rest_framework as filters

from territories.models import Territory

from .models import Activity

class SearchFilter(filters.CharFilter):
    def __init__(self, fields, *args, **kwargs):

```

```

    super().__init__(*args, **kwargs)
    self.search_fields = fields

def filter(self, qs, value):
    if value in EMPTY_VALUES:
        return qs

    qs = qs.annotate(search=SearchVector(*self.search_fields))
    return qs.filter(search=value)

def subclasses_as_choices(base_cls):
    choices = [(subclass.__name__, subclass) for subclass in
base_cls.__subclasses__()]
    return choices

class ActivityFilterSet(filters.FilterSet):
    class Meta:
        model = Activity
        fields = ("name",)

    search = SearchFilter(fields=("name", "description"))
    type = filters.MultipleChoiceFilter(
        method="filter_by_type",
        choices=subclasses_as_choices(Activity),
    )
    within_territory = filters.ModelChoiceFilter(
        method="filter_by_within_territory",
        queryset=Territory.objects.all(),
    )

    def filter_by_type(self, queryset, key, value):
        valid_choices = subclasses_as_choices(Activity)
        selected = [cls for name, cls in valid_choices if name in value]
        return queryset.instance_of(*selected)

    def filter_by_within_territory(self, queryset, key, value):
        return queryset.filter(venue__location__within=territory.boundaries)

web/activities/viewsets.py:

from rest_framework import viewsets, parsers, status
from rest_framework.decorators import action
from rest_framework.response import Response
from django.db.models import Q
from django.utils import timezone

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		70

```

from django_filters import rest_framework as filters

from users.permissions import OwnedByCurrentUserOrReadOnly
from commons.viewsets import with_my_list_endpoint

from .serializers import PolymorphicActivitySerializer, Activity,
OneTimeActivity
from .filtersets import ActivityFilterSet

@with_my_list_endpoint(field_name="activities", methods=["get"])
class ActivityViewSet(viewsets.ModelViewSet):
    serializer_class = PolymorphicActivitySerializer
    parser_classes = [parsers.MultiPartParser, parsers.FormParser]
    filter_backends = [filters.DjangoFilterBackend]
    filterset_class = ActivityFilterSet

    @action(["post"], detail=True)
    def publish(self, request, *args, **kwargs):
        instance = self.get_object()
        instance_type = instance.__class__.__name__
        type_method_mapping = {
            "OneTimeActivity": self.is_one_time_activity_publishable,
            "RecurringActivity": self.is_recurring_activity_publishable,
        }
        is_publishable = type_method_mapping[instance_type](instance)
        if is_publishable:
            instance.is_published = True
            instance.save()
            serializer = self.get_serializer(instance)
            return Response(serializer.data)

        return Response(
            "The object is not complete to be published.",
            status=status.HTTP_400_BAD_REQUEST,
        )

    @action(["post"], detail=True)
    def hide(self, request, *args, **kwargs):
        instance = self.get_object()
        instance.is_published = False
        instance.save()
        serializer = self.get_serializer(instance)
        return Response(serializer.data)

    def is_recurring_activity_publishable(self, instance):
        return len(instance.options.all()) > 0

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		71

```

def is_one_time_activity_publishable(self, instance):
    return True

def get_queryset(self):
    is_published = Q(is_published=True)
    today = timezone.now().date()
    is_relevant = Q(OneTimeActivity__date__gt=today) | ~Q(
        instance_of=OneTimeActivity
    )
    current_user = self.request.user
    is_mine = Q(created_by=current_user)

    all_filters = is_published
    if self.action != "retrieve":
        all_filters &= is_relevant
    if not current_user.is_anonymous:
        all_filters |= is_mine

    return (
        Activity.objects.select_related("venue", "created_by")
        .prefetch_related("media")
        .filter(all_filters)
        .all()
    )

def get_permissions(self):
    return [OwnedByCurrentUserOrReadOnly("created_by")]

```

Додаток Б

tests/fixtures/clients.py:

```

import pytest
import httpx

```

```

BASE_URL = "http://proxy/"

```

```

@pytest.fixture
def client():
    with httpx.Client(base_url=BASE_URL) as client:
        yield client

```

```

@pytest.fixture

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		72

```

def client1():
    with httpx.Client(base_url=BASE_URL) as client:
        yield client

@pytest.fixture
def auth_client(client, signup_user):
    client.headers["Authorization"] = f"Bearer {signup_user.json()['access']}"
    yield client

@pytest.fixture
def auth_client1(client1, signup_user1):
    client1.headers["Authorization"] = f"Bearer {signup_user1.json()['access']}"
    yield client1

tests/fixtures/users.py:

import pytest
from uuid import uuid4

@pytest.fixture
def user_data():
    yield {
        "username": f"user-{uuid4().hex[:8]}",
        "password": "StrongP@ssw0rd!",
    }

@pytest.fixture
def user1_data():
    yield {
        "username": f"user1-{uuid4().hex[:8]}",
        "password": "StrongP@ssw0rd!",
    }

@pytest.fixture
def signup_user(client, user_data):
    response = client.post("/auth/signup/", json=user_data)
    assert response.status_code == 201
    yield response

@pytest.fixture
def signup_user1(client, user1_data):
    response = client.post("/auth/signup/", json=user1_data)
    assert response.status_code == 201
    yield response

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		73

```
@pytest.fixture
def user_multiple_tokens(client, signup_user, user_data):
    tokens = []
    for _ in range(3):
        response = client.post("/auth/login/", json=user_data)
        tokens.append(response.json())
    yield tokens
```

tests/fixtures/venues.py:

```
import pytest
```

```
@pytest.fixture
def venue_data():
    yield {
        "name": "Big Club",
        "description": "The biggest club in your city!",
        "location": "-23.54321,5.54322",
    }
```

```
@pytest.fixture
def create_venue(auth_client, venue_data, media_data, route_data):
    files = media_data | route_data
    response = auth_client.post(
        "/venues/",
        data=venue_data,
        files=files,
    )
    assert response.status_code == 201
    yield response
```

```
@pytest.fixture
def publish_venue(auth_client, create_venue, create_layout):
    url = f"/venues/{create_venue.json()['pk']}/publish/"
    response = auth_client.post(url)
    assert response.status_code == 200
    yield response
```

tests/users/test_my.py:

```
from uuid import uuid4
```

```

class TestMyObjectUsers:
    url = "/users/my/"

    def test_successful_get(self, auth_client, user_data):
        response = auth_client.get(self.url)
        assert response.status_code == 200
        data = response.json()
        assert "username" in data and "image" in data
        assert data["username"] == user_data["username"]

    def test_successful_put(self, auth_client):
        new_username = f"user-{uuid4().hex[:6]}"
        response = auth_client.put(self.url, data={"username": new_username})
        assert response.status_code == 200
        data = response.json()
        assert "username" in data
        assert data["username"] == new_username

    def test_successful_patch(self, auth_client):
        new_description = "New description"
        response = auth_client.patch(self.url, data={"description":
new_description})
        assert response.status_code == 200
        data = response.json()
        assert "username" in data and "description" in data
        assert data["description"] == new_description

    def test_invalid_method(self, auth_client):
        response = auth_client.delete(self.url)
        assert response.status_code == 405

    def test_require_auth(self, client):
        response = client.get(self.url)
        assert response.status_code == 401

```

tests/venues/test_detail.py:

```

class TestDetailVenues:
    url = "/venues/"

    def build_url(self, pk):
        return f"{self.url}{pk}/"

    def test_successful_get(self, client, publish_venue):
        url = self.build_url(publish_venue.json()["pk"])

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		75

```

response = client.get(url)
assert response.status_code == 200
data = response.json()
assert isinstance(data, dict)

def test_hidden_get(self, client1, create_venue):
    url = self.build_url(create_venue.json()["pk"])
    response = client1.get(url)
    assert response.status_code == 404

def test_foreign_get(self, auth_client1, publish_venue):
    url = self.build_url(publish_venue.json()["pk"])
    response = auth_client1.get(url)
    assert response.status_code == 200
    data = response.json()
    assert isinstance(data, dict)

def test_successful_patch(self, auth_client, publish_venue):
    url = self.build_url(publish_venue.json()["pk"])
    data = {"name": "Normal club"}
    response = auth_client.patch(url, data=data)
    assert response.status_code == 200
    data = response.json()
    assert data["name"] == "Normal club"

def test_override_created_by(
    self,
    auth_client,
    auth_client1,
    signup_user1,
    publish_venue,
):
    url = self.build_url(publish_venue.json()["pk"])

    data = {"created_by": signup_user1.json()["pk"]}
    response = auth_client.patch(url, data=data)
    assert response.status_code == 200

    response = auth_client1.patch(url, data=data)
    assert response.status_code == 403

```

Додаток В

docker-compose.yml:

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		76

```

services:
  web:
    build:
      context: ./web
      dockerfile: Dockerfile
    env_file: .env
    volumes:
      - web-static:/var/www/static
      - web-media:/var/www/media
    depends_on:
      db:
        condition: service_healthy
      cache:
        condition: service_healthy

  db:
    build:
      context: ./db
      dockerfile: Dockerfile
    volumes:
      - db-data:/var/lib/data
    environment:
      - POSTGRES_USER=db-user
      - POSTGRES_DB=activity-finder
    env_file: .env
    healthcheck:
      test: ["CMD", "pg_isready", "-d", "activity-finder", "-U", "db-user"]
      interval: 10s
      timeout: 15s
      retries: 5

  cache:
    image: redis:7.0.5-alpine
    volumes:
      - cache-data:/var/cache/cache
    healthcheck:
      test: ["CMD", "redis-cli", "--raw", "incr", "ping"]
      interval: 10s
      timeout: 15s
      retries: 5

  proxy:
    build:
      context: ./proxy
      dockerfile: Dockerfile
    ports:

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		77

```

- 80:80
volumes:
  - web-static:/var/www/static:ro
  - web-media:/var/www/media
  - proxy-log:/var/log/proxy
depends_on:
  - web
healthcheck:
  test: ["CMD", "service", "nginx", "status"]
  interval: 10s
  timeout: 15s
  retries: 5

```

```

volumes:
  web-static:
  web-media:
  db-data:
  cache-data:
  proxy-log:

```

docker-compose.tests.yml:

services:

```

tests:
  build:
    context: ./tests
    dockerfile: Dockerfile
  depends_on:
    proxy:
      condition: service_healthy

```

web/Dockerfile:

```

# syntax = docker/dockerfile:experimental
FROM python:3.12-bookworm

```

```

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

```

```

WORKDIR /var/www/web

```

```

RUN adduser \
  --system \

```

		Корнійчук В.В.			ІПЗ.КР.Б — 121 — 13 — ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		78

```
--no-create-home \  
web-user
```

```
COPY ..
```

```
RUN sed -i 's/\r$//g' ./entrypoint.sh && \  
chmod +x ./entrypoint.sh
```

```
RUN mkdir -p /var/www/media /var/www/media/users  
/var/www/media/activities /var/www/static \  
&& chown -R web-user:users /var/www/media/users  
/var/www/media/activities /var/www/media /var/www/static \  
&& chmod -R 755 /var/www/media /var/www/media/users  
/var/www/media/activities /var/www/static \  
&& chmod -R g+s /var/www/media /var/www/media/users  
/var/www/media/activities /var/www/static
```

```
RUN --mount=target=/var/lib/apt/lists,type=cache,sharing=locked \  
--mount=target=/var/cache/apt,type=cache,sharing=locked \  
rm -f /etc/apt/apt.conf.d/docker-clean \  
&& apt-get update \  
&& apt-get install --no-install-recommends -y gdal-bin \  
&& rm -rf /var/lib/apt/lists/*
```

```
RUN --mount=type=cache,mode=0755,target=/root/.cache/pip \  
pip install --upgrade --root-user-action=ignore pip \  
&& pip install -r requirements.txt
```

```
USER web-user
```

```
ENTRYPOINT ["/entrypoint.sh"]
```

```
web/entrypoint.sh:
```

```
#!/bin/sh
```

```
python manage.py migrate --no-input  
python manage.py collectstatic --no-input
```

```
gunicorn web.wsgi:application --bind 0.0.0.0:8000
```